

---

# Python ライブラリリファレンス

リリース 2.3.3

Guido van Rossum

Fred L. Drake, Jr., editor

日本語訳: Python ドキュメント翻訳プロジェクト

平成 16 年 6 月 28 日

**PythonLabs**

Email: [docs@python.org](mailto:docs@python.org)

Copyright © 2001, 2002, 2003 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. All rights reserved.

Copyright © 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

Translation Copyright © 2003 Python Document Japanese Translation Project. All rights reserved.

ライセンスおよび許諾に関する完全な情報は、このドキュメントの末尾を参照してください。

## 概要

Python は拡張性のあるインタプリタ形式のオブジェクト指向言語です。簡単なテキスト処理スクリプトから対話型の WWW ブラウザまで、幅広い用途に対応しています。

*Python* リファレンスマニュアルでは、言語に関する厳密な構文と意味づけを記述していますが、Python をすぐに役立てられるようにする上で大いに貢献する、言語とともに配付される標準ライブラリについては記述されていません。このライブラリには、ファイル I/O のように Python プログラムが直接アクセスできないシステム機能へのアクセス機能を提供する (C で書かれた) 組み込みモジュールや、Python で記述された、日々のプログラミングで生じる多くの問題に標準的な解決策を提供するモジュールが入っています。これらのモジュールのいくつかには、Python プログラムに移植性を持たせ、それを高めるという明確な意図があります。

このライブラリリファレンスマニュアルでは、Python の標準ライブラリだけでなく、多くのオプションのライブラリモジュール (プラットフォームがサポートしているかどうか、あるいはコンパイル時の設定により、使える場合と使えない場合がある) について説明しています。また、言語の標準の型、組み込みの関数と例外、Python リファレンスマニュアルに説明がなかったり不足している多くの点についても説明しています。

このマニュアルは、読者が Python 言語について基礎的な知識を持っていることを仮定しています。形式ばらない Python への入門には、*Python* チュートリアルを参照してください。*Python* リファレンスマニュアルは、高度な文法と意味づけについて疑問があるときに参照してください。最後に、*Python* インタプリタの拡張と組み込みと題されたマニュアルには、Python に新しい機能を追加する方法と、他のアプリケーションに Python を組み込む方法を記述しています。



# 目次

第 1 章	はじめに	1
第 2 章	組み込みオブジェクト	3
2.1	組み込み関数	3
2.2	非必須組み込み関数 (Non-essential Built-in Functions)	16
2.3	組み込み型	17
2.4	組み込み例外	36
2.5	組み込み定数	42
第 3 章	Python ランタイム サービス	45
3.1	sys — システムパラメータと関数	46
3.2	gc — ガベージコレクタ インターフェース	52
3.3	weakref — 弱参照	54
3.4	fpectl — 浮動小数点例外の制御	59
3.5	atexit — 終了ハンドラ	60
3.6	types — 組み込み型の名前	61
3.7	UserDict — 辞書オブジェクトのためのクラスラッパー	64
3.8	UserList — リストオブジェクトのためのクラスラッパー	64
3.9	UserString — 文字列オブジェクトのためのクラスラッパー	65
3.10	operator — 関数形式の標準演算子	66
3.11	inspect — 使用中のオブジェクトを調査	70
3.12	traceback — スタックトレースの表示や取り出し	74
3.13	linecache — テキストラインにランダムアクセスする	76
3.14	pickle — Python オブジェクトの整列化	77
3.15	cPickle — より高速な pickle	88
3.16	copy_reg — pickle サポート関数を登録する	88
3.17	shelve — Python オブジェクトの永続化	89
3.18	copy — 浅いコピーおよび深いコピー操作	91
3.19	marshal — 内部使用向けの Python オブジェクト整列化	92
3.20	warnings — 警告の制御	93
3.21	imp — import 内部へアクセスする	96
3.22	pkgutil — パッケージ拡張ユーティリティ	99
3.23	code — インタプリタ基底クラス	100
3.24	codeop — Python コードをコンパイルする	102
3.25	pprint — データ出力の整然化	103

3.26	<code>repr</code> — もう一つの <code>repr()</code> の実装	106
3.27	<code>new</code> — ランタイム内部オブジェクトの作成	107
3.28	<code>site</code> — サイト固有の設定フック	108
3.29	<code>user</code> — ユーザー設定のフック	109
3.30	<code>__builtin__</code> — 組み込み関数	110
3.31	<code>__main__</code> — トップレベルのスクリプト環境	110
3.32	<code>__future__</code> — Future ステートメントの定義	110
<b>第 4 章</b>	<b>文字列処理</b>	<b>113</b>
4.1	<code>string</code> — 一般的な文字列操作	113
4.2	<code>re</code> — 正規表現操作	117
4.3	<code>struct</code> — 文字列データをパックされたバイナリデータとして解釈する	128
4.4	<code>difflib</code> — 差異の計算を助ける	131
4.5	<code>fpformat</code> — 浮動小数点の変換	139
4.6	<code>StringIO</code> — ファイルのように文字列を読み書きする	140
4.7	<code>cStringIO</code> — 高速化された <code>StringIO</code>	140
4.8	<code>textwrap</code> — テキストの折り返しと詰め込み	141
4.9	<code>codecs</code> — codec レジストリと基底クラス	143
4.10	<code>unicodedata</code> — Unicode データベース	153
4.11	<code>stringprep</code> — インターネットのための文字列調製	154
<b>第 5 章</b>	<b>各種サービス</b>	<b>157</b>
5.1	<code>pydoc</code> — ドキュメント生成とオンラインヘルプシステム	157
5.2	<code>doctest</code> — ドキュメンテーション文字列に本当のことが書かれているか調べる	158
5.3	<code>unittest</code> — 単体テストフレームワーク	166
5.4	<code>test</code> — Python 用回帰テストパッケージ	178
5.5	<code>math</code> — 数学関数	182
5.6	<code>cmath</code> — 複素数のための数学関数	184
5.7	<code>random</code> — 擬似乱数を生成する	185
5.8	<code>whrandom</code> — 擬似乱数生成器	188
5.9	<code>bisect</code> — 配列二分法アルゴリズム	189
5.10	<code>heapq</code> — ヒープキューアルゴリズム	190
5.11	<code>array</code> — 効率のよい数値配列	193
5.12	<code>sets</code> — ユニークな要素の順序なしコレクション	196
5.13	<code>itertools</code> — 効率的なループ実行のためのイテレータ生成関数	199
5.14	<code>ConfigParser</code> — 設定ファイルの構文解析器	205
5.15	<code>fileinput</code> — 複数の入力ストリームをまたいだ行の繰り返し処理をサポートする。	208
5.16	<code>xreadlines</code> — ファイルの各行に対する効率のよい反復処理	209
5.17	<code>calendar</code> — 一般的なカレンダーに関する関数群	210
5.18	<code>cmd</code> — 行指向のコマンドインタプリタのサポート	211
5.19	<code>shlex</code> — 単純な字句解析	214
<b>第 6 章</b>	<b>汎用オペレーティングシステムサービス</b>	<b>219</b>
6.1	<code>os</code> — 雑多なオペレーティングシステムインタフェース	220
6.2	<code>os.path</code> — 共通のパス名操作	239
6.3	<code>dircache</code> — キャッシュされたディレクトリ一覧の生成	242
6.4	<code>stat</code> — <code>stat()</code> の返す内容を解釈する	242
6.5	<code>statcache</code> — <code>os.stat()</code> の最適化	244

6.6	statvfs — <code>os.statvfs()</code> で使われる定数群	245
6.7	filecmp — ファイルおよびディレクトリの比較	246
6.8	popen2 — アクセス可能な I/O ストリームを持つ子プロセス生成	247
6.9	datetime — 基本的な日付型および時間型	250
6.10	time — 時刻データへのアクセスと変換	269
6.11	sched — イベントスケジューラ	274
6.12	mutex — 排他制御	276
6.13	getpass — 可搬性のあるパスワード入力機構	276
6.14	curses — 文字セル表示のための端末操作	277
6.15	curses.textpad — curses プログラムのためのテキスト入力ウィジェット	293
6.16	curses.wrapper — curses プログラムのための端末ハンドラ	295
6.17	curses.ascii — ASCII 文字に関するユーティリティ	295
6.18	curses.panel — curses のためのパネルスタック拡張	298
6.19	getopt — コマンドラインオプションのパース	299
6.20	optparse — 強力なコマンドラインオプション解析器	301
6.21	tempfile — 一時的なファイルやディレクトリの生成	327
6.22	errno — 標準の errno システムシンボル	330
6.23	glob — UNIX 形式のパス名のパターン展開	336
6.24	fnmatch — UNIX ファイル名のパターンマッチ	336
6.25	shutil — 高レベルなファイル操作	337
6.26	locale — 国際化サービス	339
6.27	gettext — 多言語対応に関する国際化サービス	345
6.28	logging — Python 用ロギング機能	354
<b>第 7 章</b>	<b>オプションのオペレーティングシステムサービス</b>	<b>371</b>
7.1	signal — 非同期イベントにハンドラを設定する	371
7.2	socket — 低レベルネットワークインターフェース	374
7.3	select — I/O 処理の完了を待機する	384
7.4	thread — マルチスレッドのコントロール	386
7.5	threading — 高レベルスレッドインターフェース	387
7.6	dummy_thread — thread の代替モジュール	396
7.7	dummy_threading — threading の代替モジュール	396
7.8	Queue — 同期キュークラス	396
7.9	mmap — メモリマップファイル	397
7.10	anydbm — DBM 形式のデータベースへの汎用アクセスインタフェース	399
7.11	dbhash — BSD データベースライブラリへの DBM 形式のインタフェース	400
7.12	whichdb — どの DBM モジュールがデータベースを作ったかを推測する	402
7.13	bsddb — Berkeley DB ライブラリへのインタフェース	402
7.14	dumbdbm — 可搬性のある DBM 実装	405
7.15	zlib — <b>gzip</b> 互換の圧縮	406
7.16	gzip — <b>gzip</b> ファイルのサポート	408
7.17	bz2 — <b>bzip2</b> 互換の圧縮ライブラリ	409
7.18	zipfile — ZIP アーカイブの処理	411
7.19	tarfile — tar アーカイブファイルを読み書きする	415
7.20	readline — GNU readline のインタフェース	421
7.21	rlcompleter — GNU readline 向け補完関数	423

<b>第 8 章</b>	<b>Unix 独特のサービス</b>	<b>425</b>
8.1	posix — 最も一般的な POSIX システムコール群	425
8.2	pwd — パスワードデータベースへのアクセスを提供する	426
8.3	grp — グループデータベースへのアクセス	427
8.4	crypt — UNIX パスワードをチェックするための関数	428
8.5	dl — 共有オブジェクトの C 関数の呼び出し	428
8.6	dbm — UNIX dbm のシンプルなインタフェース	430
8.7	gdbm — GNU による dbm の再実装	430
8.8	termios — POSIX スタイルの端末制御	432
8.9	TERMIOS — termios モジュールで使われる定数	433
8.10	tty — 端末制御のための関数群	433
8.11	pty — 擬似端末ユーティリティ	434
8.12	fcntl — fcntl() および ioctl() システムコール	434
8.13	pipes — シェルパイプラインへのインタフェース	436
8.14	posixfile — ロック機構をサポートするファイル類似オブジェクト	437
8.15	resource — リソース使用状態の情報	439
8.16	nis — Sun の NIS (Yellow Pages) へのインタフェース	442
8.17	syslog — UNIX syslog ライブラリルーチン群	443
8.18	commands — コマンド実行ユーティリティ	444
<b>第 9 章</b>	<b>Python デバッガ</b>	<b>445</b>
9.1	デバッガコマンド	446
9.2	どのように動作しているか	449
<b>第 10 章</b>	<b>Python プロファイラ</b>	<b>451</b>
10.1	プロファイラとは	451
10.2	旧バージョンのプロファイラとの違い	451
10.3	インスタント・ユーザ・マニュアル	452
10.4	決定論的プロファイリングとは何か?	454
10.5	リファレンス・マニュアル	454
10.6	制限事項	457
10.7	キャリブレーション (補正)	458
10.8	拡張 — プロファイラの改善	459
10.9	hotshot — ハイパフォーマンス・ロギング・プロファイラ	459
10.10	timeit — 小さなコード断片の実行時間計測	461
<b>第 11 章</b>	<b>インターネットプロトコルとその支援</b>	<b>465</b>
11.1	webbrowser — 便利なウェブブラウザコントローラー	465
11.2	cgi — CGI (ゲートウェイインタフェース規格) のサポート	467
11.3	cgitb — CGI スクリプトのトレースバック管理機構	475
11.4	urllib — URL による任意のリソースへのアクセス	476
11.5	urllib2 — URL を開くための拡張可能なライブラリ	482
11.6	httpplib — HTTP プロトコルクライアント	490
11.7	ftplib — FTP プロトコルクライアント	493
11.8	gopherlib — gopher プロトコルのクライアント	497
11.9	poplib — POP3 プロトコルクライアント	497
11.10	imaplib — IMAP4 プロトコルクライアント	499
11.11	nntplib — NNTP プロトコルクライアント	504



11.12	smtplib — SMTP プロトコル クライアント	508
11.13	telnetlib — Telnet クライアント	512
11.14	urlparse — URL を解析して構成要素にする	515
11.15	SocketServer — ネットワークサーバ構築のためのフレームワーク	517
11.16	BaseHTTPServer — 基本的な機能を持つ HTTP サーバ	519
11.17	SimpleHTTPServer — 簡潔な HTTP リクエストハンドラ	522
11.18	CGIHTTPServer — CGI 実行機能付き HTTP リクエスト処理機構	523
11.19	Cookie — HTTP の状態管理	524
11.20	xmlrpclib — XML-RPC クライアントアクセス	528
11.21	SimpleXMLRPCServer — 基本的な XML-RPC サーバ	531
11.22	DocXMLRPCServer — セルフドキュメンティング XML-RPC サーバ	533
11.23	asyncore — 非同期ソケットハンドラ	534
11.24	asynchat — 非同期ソケット コマンド/レスポンス ハンドラ	537
<b>第 12 章</b>	<b>インターネット上のデータの操作</b>	<b>543</b>
12.1	formatter — 汎用の出力書式化機構	543
12.2	email — 電子メールと MIME 処理のためのパッケージ	547
12.3	mailcap — mailcap ファイルの操作	579
12.4	mailbox — 様々なメールボックス形式の読み出し	580
12.5	mhlib — MH のメールボックスへのアクセス機構	582
12.6	mimertools — MIME メッセージを解析するためのツール	584
12.7	mimetypes — ファイル名を MIME 型へマップする	586
12.8	MimeWriter — 汎用 MIME ファイルライター	588
12.9	mimify — 電子メールメッセージの MIME 処理	589
12.10	multifile — 個別の部分を含んだファイル群のサポート	590
12.11	rfc822 — RFC 2822 準拠のメールヘッダ読み出し	593
12.12	base64 — MIME base64 形式データのエンコードおよびデコード	597
12.13	binascii — バイナリデータと ASCII データとの間での変換	598
12.14	binhex — binhex4 形式ファイルのエンコードおよびデコード	599
12.15	quopri — MIME quoted-printable 形式データのエンコードおよびデコード	600
12.16	uu — uuencode 形式のエンコードとデコード	601
12.17	xdrllib — XDR データのエンコードおよびデコード	602
12.18	netrc — netrc ファイルの処理	605
12.19	robotparser — robots.txt のためのパーザ	606
12.20	csv — CSV ファイルの読み書き	606
<b>第 13 章</b>	<b>構造化マークアップツール</b>	<b>611</b>
13.1	HTMLParser — HTML および XHTML のシンプルなパーザ	611
13.2	sgmllib — 単純な SGML パーザ	614
13.3	htmlplib — HTML 文書の解析器	616
13.4	htmlentitydefs — HTML 一般エンティティの定義	618
13.5	xml.parsers.expat — Expat を使った高速な XML 解析	618
13.6	xml.dom — 文書オブジェクトモデル (DOM) API	626
13.7	xml.dom.minidom — 軽量の DOM 実装	637
13.8	xml.dom.pulldom — 部分的な DOM ツリー構築のサポート	642
13.9	xml.sax — SAX2 パーサのサポート	643
13.10	xml.sax.handler — SAX ハンドラの基底クラス	645

13.11	<code>xml.sax.saxutils</code> — SAX ユーティリティ	650
13.12	<code>xml.sax.xmlreader</code> — XML パーサのインターフェース	651
13.13	<code>xmllib</code> — XML ドキュメントのパーサ	655
<b>第 14 章</b>	<b>マルチメディアサービス</b>	<b>659</b>
14.1	<code>audioop</code> — 生のオーディオデータの操作	659
14.2	<code>imageop</code> — 生の画像データを操作する	662
14.3	<code>aifc</code> — AIFF および AIFC ファイルの読み書き	663
14.4	<code>sunau</code> — Sun AU ファイルの読み書き	666
14.5	<code>wave</code> — WAV ファイルの読み書き	668
14.6	<code>chunk</code> — IFF チャンクデータの読み込み	671
14.7	<code>colorsys</code> — 色体系間の変換	672
14.8	<code>rgbimg</code> — “SGI RGB” ファイルを読み書きする	673
14.9	<code>imghdr</code> — 画像の形式を決定する	673
14.10	<code>sndhdr</code> — サウンドファイルの識別	674
14.11	<code>ossaudiodev</code> — OSS 互換オーディオデバイスへのアクセス	675
<b>第 15 章</b>	<b>暗号関連のサービス</b>	<b>681</b>
15.1	<code>hmac</code> — メッセージ認証のための鍵付きハッシュ化	681
15.2	<code>md5</code> — MD5 メッセージダイジェストアルゴリズム	682
15.3	<code>sha</code> — SHA-1 メッセージダイジェストアルゴリズム	683
15.4	<code>mpz</code> — GNU 多倍長整数	684
15.5	<code>rotor</code> — エニグマ暗号機のような暗号化と復号化	685
<b>第 16 章</b>	<b>Tk を用いたグラフィカルユーザインターフェイス</b>	<b>687</b>
16.1	<code>Tkinter</code> — Tcl/Tk への Python インターフェイス	687
16.2	<code>Tix</code> — Tk の拡張ウィジェット	699
16.3	<code>ScrolledText</code> — スクロールするテキストウィジェット	705
16.4	<code>turtle</code> — Tk のためのタートルグラフィックス	705
16.5	<code>Idle</code>	707
16.6	他のグラフィカルユーザインターフェイスパッケージ	710
<b>第 17 章</b>	<b>制限実行 (restricted execution)</b>	<b>713</b>
17.1	<code>rexec</code> — 制限実行のフレームワーク	714
17.2	<code>Bastion</code> — オブジェクトに対するアクセスの制限	717
<b>第 18 章</b>	<b>Python 言語サービス</b>	<b>719</b>
18.1	<code>parser</code> — Python 解析木にアクセスする	719
18.2	<code>symbol</code> — Python 解析木と共に使われる定数	729
18.3	<code>token</code> — Python 解析木と共に使われる定数	729
18.4	<code>keyword</code> — Python キーワードチェック	730
18.5	<code>tokenize</code> — Python ソースのためのトークナイザ	730
18.6	<code>tabnanny</code> — あいまいなインデントの検出	731
18.7	<code>pyclbr</code> — Python クラスブラウザーサポート	732
18.8	<code>py_compile</code> — Python ソースファイルのコンパイル	733
18.9	<code>compileall</code> — Python ライブラリをバイトコンパイル	733
18.10	<code>dis</code> — Python バイトコードの逆アセンブラ	734
18.11	<code>distutils</code> — Python モジュールの構築とインストール	742

<b>第 19 章 Python コンパイラパッケージ</b>	<b>743</b>
19.1 基本的なインターフェイス	743
19.2 制限	744
19.3 Python 抽象構文	744
19.4 Visitor を使って AST をわたり歩く	749
19.5 バイトコード生成	750
<b>第 20 章 SGI IRIX 特有のサービス</b>	<b>751</b>
20.1 al — SGI のオーディオ機能	751
20.2 AL — al モジュールで使われる定数	753
20.3 cd — SGI システムの CD-ROM へのアクセス	753
20.4 fl — グラフィカルユーザーインターフェースのための FORMS ライブラリ	757
20.5 FL — fl モジュールで使用される定数	763
20.6 flp — 保存された FORMS デザインをロードする関数	763
20.7 fm — <i>Font Manager</i> インターフェース	763
20.8 gl — <i>Graphics Library</i> インターフェース	764
20.9 DEVICE — gl モジュールで使われる定数	766
20.10 GL — gl モジュールで使われる定数	766
20.11 imgfile — SGI imglib ファイルのサポート	766
20.12 jpeg — JPEG ファイルの読み書きを行う	767
<b>第 21 章 SunOS 特有のサービス</b>	<b>769</b>
21.1 sunaudiodev — Sun オーディオハードウェアへのアクセス	769
21.2 SUNAUDIODEV — sunaudiodev で使われる定数	770
<b>第 22 章 MS Windows 特有のサービス</b>	<b>773</b>
22.1 msvcrt — MS VC++ 実行時システムの有用なルーチン群	773
22.2 _winreg — Windows レジストリへのアクセス	774
22.3 winsound — Windows 用の音声再生インタフェース	779
<b>付 録 A ドキュメント化されていないモジュール</b>	<b>783</b>
A.1 フレームワーク	783
A.2 雑多な有用ユーティリティ	783
A.3 プラットフォーム特有のモジュール	783
A.4 マルチメディア関連	784
A.5 撤廃されたもの	784
A.6 SGI 特有の拡張モジュール	785
<b>付 録 B バグ報告</b>	<b>787</b>
<b>付 録 C 歴史とライセンス</b>	<b>789</b>
C.1 History of the software	789
C.2 Terms and conditions for accessing or otherwise using Python	790
<b>付 録 D 日本語訳について</b>	<b>793</b>
D.1 このドキュメントについて	793
D.2 翻訳者一覧 (敬称略)	793
<b>Module Index</b>	<b>795</b>



# はじめに

この“Python ライブラリ”には様々な内容が収録されています。

このライブラリには、数値型やリスト型のような、通常は言語の“核”をなす部分とみなされるデータ型が含まれています。Python 言語のコア部分では、これらの型に対してリテラル表現形式を与え、意味づけ上のいくつかの制約を与えていますが、完全にその意味づけを定義しているわけではありません。(一方で、言語のコア部分では演算子のスペルや優先順位のような構文法的な属性を定義しています。) このライブラリにはまた、組み込み関数と例外が納められています — 組み込み関数および例外は、全ての Python で書かれたコード上で、`import` 文を使わずに使うことができるオブジェクトです。これらの組み込み要素のうちいくつかは言語のコア部分で定義されていますが、大半は言語コアの意味づけ上不可欠なものではないのでここでしか記述されていません。

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

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

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

それでは、ショーの始まりです!



# 組み込みオブジェクト

組み込み例外名、関数名、各種定数名は専用のシンボルテーブル中に存在しています。シンボル名を参照するときこのシンボルテーブルは最後に参照されるので、ユーザーが設定したローカルな名前やグローバルな名前によってオーバーライドすることができます。組み込み型については参照しやすいようにここで説明されています。<sup>1</sup>

この章にある表では、オペレータの優先度を昇順に並べて表わして、同じ優先度のオペレータは同じ箱に入れています。同じ優先度の二項演算子は左から右への結合順序を持っています。(単項演算子は右から左へ結合しますが選択の余地はないでしょう。)<sup>2</sup>オペレータの優先順位についての詳細は *Python Reference Manual* の 5 章をごらんください。

## 2.1 組み込み関数

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

`__import__(name[, globals[, locals[, fromlist]])`

この関数は `import` 文によって呼び出されます。この関数の主な意義は、同様のインタフェースを持つ関数でこの関数を置き換え、`import` 文の意味を変更できるようにすることです。これを行う理由とやり方の例については、標準ライブラリモジュール `ihooks` および `rexec` を読んで下さい。また、組み込みモジュール `imp` についても読んでみて下さい。自分で関数 `__import__` を構築する際に便利な操作が定義されています。

例えば、文 `'import spam'` は結果として以下の呼び出し: `__import__('spam', globals(), locals(), [])` になります; 文 `'from spam.ham import eggs'` は `'__import__('spam.ham', globals(), locals(), ['eggs'])` です。`locals()` および `['eggs']` が引数で与えられますが、関数 `__import__()` は `eggs` という名のローカル変数を設定しないので注意してください; この操作はそれ以後の `import` 文のために生成されたコードで行われます。(実際、標準の実装では `locals` 引数を全く使わず、`import` 文のパッケージ文脈を決定するためだけに `globals` を使います。)

変数 `name` が `package.module` の形式であった場合、通常、`name` という名のモジュールではなくトップレベルのパッケージ (最初のドットまでの名前) が返されます。しかし、空でない `fromlist` 引数が与えられていれば、`name` と名づけられたモジュールが返されます。これは異なる種類の `import` 文に対して生成されたバイトコードと互換性をもたせるために行われます; `'import spam.ham.eggs'` とすると、トップレベルのパッケージ `spam` はインポートする名前空間に置かれなければならないませんが、`'from spam.ham import eggs'` とすると、変数 `eggs` を見つけるためには `spam.ham` サブパッケージを使わなくてはなりません。この振る舞いを回避するために、`getattr()` を使って必要なコンポーネントを展開してください。例えば、以下のようなヘルパー関数:

<sup>1</sup> ほとんどの説明ではそこで発生しうる例外については説明されていません。このマニュアルの将来の版で訂正される予定です。

<sup>2</sup> 訳者註: HTML 版では、変換の過程で表の区切り情報が消えてしまっているため、PS 版や PDF 版をごらんください。

```
def my_import(name):
    mod = __import__(name)
    components = name.split('.')
    for comp in components[1:]:
        mod = getattr(mod, comp)
    return mod
```

を定義することができます。

**abs(*x*)**

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

**basestring()**

この抽象型は、`str` および `unicode` のスーパークラスです。この型は呼び出したりインスタンス化したりはできませんが、オブジェクトが `str` や `unicode` のインスタンスであるかどうかを調べる際に利用できます。`isinstance(obj, basestring)` は `isinstance(obj, (str, unicode))` と同じです。2.3 で追加された仕様です。

**bool(*[x]*)**

標準の真値テストを使って、値をブール値に変換します。*x* が偽なら、`False` を返します; そうでなければ `True` を返します。`bool` はクラスでもあり、`int` のサブクラスになります。`bool` クラスはそれ以上サブクラス化できません。このクラスのインスタンスは `False` および `True` だけです。

2.2.1 で追加された仕様です。

2.3 で変更された仕様: 引数が与えられなかった場合、この関数は `False` を返します。

**callable(*object*)**

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

**chr(*i*)**

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

**classmethod(*function*)**

*function* のクラスメソッドを返します。

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

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

このメソッドはクラスで呼び出すこと (例えば `C.f()`) も、インスタンスとして呼び出すこと (例えば `C().f()`) もできます。インスタンスはそのクラスが何であるかを除いて無視されます。クラスメソッドが導出クラスに対して呼び出された場合、導出されたクラスオブジェクトが暗黙の第一引数として渡されます。

クラスメソッドは C++ や Java における静的メソッドとは異なります。そのような機能を求めているなら、`staticmethod()` を参照してください。2.2 で追加された仕様です。



`cmp(x, y)`

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

`compile(string, filename, kind[, flags[, dont_inherit]])`

`string` をコードオブジェクトにコンパイルします。コードオブジェクトは `exec` 文で実行したり、`eval()` を呼び出して評価することができます。`filename` 引数はコードを読み出すファイルを指定します; ファイルから読み出されない場合には、認識可能なある値を渡します (一般的には '`<string>`' が使われます)。引数 `kind` には、この種類のコードをコンパイルしなければならないのかを指定します; `string` が命令文の列からなる場合には '`exec`' を、単一の式からなる場合には '`eval`' が、単一の対話命令文からなる場合には '`single`' にします (後者の例では、評価する何らかの式が `None` 以外の場合にはその式が出力されます)。

複数行の命令文をコンパイルする時には、2 つの注意点があります: 行末は単一の改行文字 ('`\n`') で表さなければなりません。また、入力行は少なくとも 1 つの改行文字で終端されていなければなりません。行末が '`\r\n`' で表現されている場合、文字列に `replace()` メソッドを使って '`\n`' に変換してください。

オプションの引数 `flags` および `dont_inherit` (Python 2.2 で新たに追加) は、`string` のコンパイルにおいてどの `future` 文 (PEP 236 参照) が影響を受けるかを制御します。どちらも省略された場合 (または両方ともゼロの場合)、コンパイルを呼び出している側のコードで有効になっている `future` 文の内容を有効にして `string` をコンパイルします。`flags` が与えられており、かつ `dont_inherit` が与えられていない (またはゼロ) の場合、上の場合に加えて `flags` に指定された `future` 文が使われます。`dont_inherit` がゼロでない整数で `flags` はその値になります – この関数呼び出しに関する `future` 文は無視されます。

`future` 文はビットで指定され、互いにビット単位の論理和を取ることで複数の文を指定することができます。ある機能を指定するために必要なビットフィールドは、`__future__` モジュールの `_Feature` インスタンスにおける `compiler_flag` 属性で得ることができます。

`complex([real[, imag]])`

値 `real + imag*j` の複素数型数を生成するか、文字列または数値を複素数型に変換します。最初の引数が文字列の場合、文字列を複素数として変換します。この場合関数は二つ目の引数無しで呼び出さなければなりません。二つ目の引数は文字列であってはなりません。それぞれの引数は (複素数を含む) 任意の数値型をとることができます。`imag` が省略された場合、標準の値はゼロで、関数は `int`、`long()` および `float()` のような数値型への変換関数として動作します。全ての引数が省略された場合、`0j` を返します。

`delattr(object, name)`

`setattr()` の親戚となる関数です。引数はオブジェクトと文字列です。文字列はオブジェクトの属性のどれか一つの名前でなければなりません。この関数は与えられた名前の属性を削除しますが、オブジェクトがそれを許す場合に限りです。例えば、`delattr(x, 'foobar')` は `del x.foobar` と等価です。

`dict([mapping-or-sequence])`

オプションの場所にある引数が、キーワード引数の集合から、新しく辞書オブジェクトを初期化して返します。引数が指定されていない場合は、新しい空の辞書を返します。オプションの場所にある引数がマップ型のオブジェクトの場合、そのマップ型オブジェクトと同じキーと値を持つ辞書を返します。それ以外の場合、オプションの場所にある引数は配列型か、反復をサポートするコンテナ型か、イテレータオブジェクトでなければなりません。この場合引数中の要素もまた、上に挙げた型のどれかでなくてはならず、加えて正確に 2 個のオブジェクトを持っていないではなりません。最初の要素は新たな辞書のキーとして、二つ目の要素は辞書の値として使われます。同じキーが一度以上与えられた場合、新たな辞書中には最後に与えた値だけが関連付けられます。

キーワード引数が与えられた場合、キーワードとそれに関連付けられた値が辞書の要素として追加されます。オプションの場所にあるオブジェクト内とキーワード引数の両方で同じキーが指定されていた場合、辞書中にはキーワード引数の設定値の方が残されます。

例えば、以下のコードはどれも、`{"one": 2, "two": 3}` と同じ辞書を返します:

```
•dict({'one': 2, 'two': 3})
•dict({'one': 2, 'two': 3}.items())
•dict({'one': 2, 'two': 3}.iteritems())
•dict(zip(('one', 2), ('two', 3)))
•dict(['two', 3], ['one', 2])
•dict(one=2, two=3)
•dict([(('one', 'two')[i-2], i) for i in (2, 3)])
```

2.2 で追加された仕様です。 2.3 で変更された仕様: キーワード引数から辞書を構築する機能が追加されました

**dir([object])**

引数がない場合、現在のローカルシンボルテーブルにある名前の一覧を返します。引数がある場合、そのオブジェクトの有効な属性からなる一覧を返そうと試みます。この情報はオブジェクトの `__dict__` 属性が定義されている場合、そこから収集されます。また、クラスまたは型オブジェクトからも集められます。一覧は完全なものになるとは限りません。オブジェクトがモジュールオブジェクトの場合、一覧にはモジュール属性の名前も含まれます。オブジェクトが型オブジェクトやクラスオブジェクトの場合、一覧にはそれらの属性が含まれ、かつそれらの基底クラスの属性も再帰的にたどられて含まれます。それ以外の場合には、一覧にはオブジェクトの属性名、クラス属性名、再帰的にたどった基底クラスの属性名が含まれます。返される一覧はアルファベット順に並べられています。例えば:

```
>>> import struct
>>> dir()
['__builtins__', '__doc__', '__name__', 'struct']
>>> dir(struct)
['__doc__', '__name__', 'calcsize', 'error', 'pack', 'unpack']
```

注意: `dir()` は第一に対話プロンプトのために提供されているので、厳密さや一貫性をもって定義された名前のセットよりも、むしろ興味深い名前のセットを与えようとしています。また、この関数の細かい動作はリリース間で変わる可能性があります。

**divmod(a, b)**

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

2.3 で変更された仕様: 複素数に対する `divmod()` の使用は廃用されました。

**enumerate(iterable)**

列挙オブジェクトを返します。`iterable` は配列型、イテレータ型、あるいは反復をサポートする他のオブジェクト型でなければなりません。`enumerate()` が返すイテレータの `next()` メソッドは、(ゼ

口から始まる) カウント値と、値だけ *iterable* を反復操作して得られる、対応するオブジェクトを含むタプルを返します。 `enumerate()` はインデックス付けされた値の列: `(0, seq[0]), (1, seq[1]), (2, seq[2]), ...` を得るのに便利です。 2.3 で追加された仕様です。

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

引数は文字列とオプションの 2 つの辞書からなります。引数 *expression* は Python の表現式 (技術的にいうと、条件のリストです) として構文解釈され、評価されます。このとき辞書 *globals* および *locals* はそれぞれグローバルおよびローカルな名前空間として使われます。 *locals* 辞書が存在するが、`'__builtin__'` が欠けている場合、*expression* を解析する前に現在のグローバル変数を *globals* にコピーします。このことから、*expression* は通常標準の `__builtin__` モジュールへの完全なアクセスを有し、制限された環境が伝播するようになっています。 *locals* 辞書が省略された場合、標準の値として *globals* に設定されます。辞書が両方とも省略された場合、表現式は `eval` が呼び出されている環境の下で実行されます。構文エラーは例外として報告されます。

以下に例を示します:

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

この関数は (`compile()` で生成されるような) 任意のコードオブジェクトを実行するために利用することもできます。この場合、文字列の代わりにコードオブジェクトを渡します。このコードオブジェクトは引数 *kind* を `'eval'` にしてコンパイルされていなければなりません。

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

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

この関数は `exec` 文に似ていますが、文字列の代わりにファイルに対して構文解釈を行います。 `import` 文と違って、モジュール管理機構を使いません — この関数はファイルを無条件に読み込み、新たなモジュールを生成しません。<sup>3</sup>

引数は文字列とオプションの 2 つの辞書からなります。 *file* は読み込まれ、(モジュールのように) Python 文の配列として評価されます。このとき *globals* および *locals* がそれぞれグローバルおよびローカルな名前空間として使われます。 *locals* 辞書が省略された場合、標準の値として *globals* に設定されます。辞書が両方とも省略された場合、表現式は `execfiles` が呼び出されている環境の下で実行されます。戻り値は `None` です。

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

**file**(*filename*[, *mode*[, *bufsize*]])

新たなファイルオブジェクト (前の組み込み型の節で説明されています) を返します。最初の 2 つの引数は `studio` の `fopen()` と同じです: *filename* は開きたいファイルの名前で、*mode* はファイルをどのようにして開くかを指定します: 読み出しは `'r'`、書き込み (ファイルがすでに存在すれば切り詰められます) は `'w'`、追記書き込みは `'a'` です (いくつかの UNIX システムでは、全ての書き込みが現在のファイルシーク位置に関係なくファイルの末尾に追加されます)。

`'r+'`、`'w+'`、および `'a+'` はファイルを更新モードで開きます (`'w+'` はファイルがすでに存在すれば切り詰めるので注意してください)。パイナリとテキストファイルを区別するシステムでは、ファ

<sup>3</sup>この関数は比較利用されない方なので、将来構文にするかどうかは保証できません。

イルをバイナリモードで開くためには 'b' を追加してください (区別しないシステムでは 'b' は無視されます)。ファイルを開くことができなければ、`IOError` が送出されます。

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

*mode* が省略された場合、標準の値は 'r' になります。移植性を高めるためには、バイナリファイルを開くときには、*mode* の値に 'b' を追加しなければなりません。(バイナリファイルとテキストファイルを区別なく扱うようなシステムでも、ドキュメンテーションの代わりになるので便利です) オプションの *bufsize* 引数は、ファイルのために必要とするバッファのサイズを指定します: 0 は非バッファリング、1 は行単位バッファリング、その他の正の値は指定した値 (の近似値) のサイズをもつバッファを使用することを意味します。*bufsize* の値が負の場合、システムの標準を使います。通常、端末は行単位のバッファリングであり、その他のファイルは完全なバッファリングです。省略された場合、システムの標準の値が使われます。<sup>4</sup>

`file()` コンストラクタは Python 2.2 で新たに追加されました。以前の関数名である `open()` は互換性のために残されており、`file()` の別名となっています。

**filter**(*function*, *list*)

*list* のうち、*function* が真を返すような要素からなるリストを構築します。*list* は配列か、反復をサポートするコンテナか、イテレータです。*list* が文字列型かタプル型の場合、結果も同じ型になります。*function* が `None` の場合、恒等関数を仮定します。すなわち、*list* の偽となる要素 (ゼロまたは空) は除去されます。

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

**float**(*x*)

文字列または数値を浮動小数点数に変換します。引数が文字列の場合、十進の数または浮動小数点数を含んでいなければなりません。符号が付いていてもかまいません。また、空白文字中に埋め込まれていてもかまいません; これは `string.atof(x)` と同様の動作です。それ以外の場合、引数は通常整数、長整数、または浮動小数点数をとることができ、同じ値の浮動小数点数が (Python の浮動小数点精度で) 返されます。引数が指定されなかった場合、0.0 を返します。

注意: 文字列で値を渡す際、背後の C ライブラリによって NaN および Infinity が返されるかもしれません。これらの値を返すような特殊な文字列のセットは完全に C ライブラリに依存しており、バリエーションがあることが知られています。

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

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

<sup>4</sup> 現状では、`setvbuf()` を持っていないシステムでは、バッファサイズを指定しても効果はありません。バッファサイズを指定するためのインタフェースは `setvbuf()` を使っては行われていません。何らかの I/O が実行された後で呼び出されるとコアダンプすることがあり、どのような場合にそうなるかを決定する信頼性のある方法がないからです。



**globals()**

現在のグローバルシンボルテーブルを表す辞書を返します。常に現在のモジュールの辞書になります (関数またはメソッドの中ではそれらを定義しているモジュールを指し、この関数を呼び出したモジュールではありません)。

**hasattr(object, name)**

引数はオブジェクトと文字列です。文字列がオブジェクトの属性名の一つであった場合 1 を、そうでない場合 0 を返します (この関数は `getattr(object, name)` を呼び出し、例外を送出するかどうかを調べることで実装しています)。

**hash(object)**

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

**help([object])**

組み込みヘルプシステムを起動します (この関数は対話的な使用のためのものです)。引数を与えていない場合、対話的ヘルプシステムはインタプリタコンソール上で起動します。引数が文字列の場合、文字列はモジュール、関数、クラス、メソッド、キーワード、またはドキュメントの項目名として検索され、ヘルプページがコンソール上に印字されます。引数何らかのオブジェクトの場合、そのオブジェクトに関するヘルプページが生成されます。2.2 で追加された仕様です。

**hex(x)**

(任意のサイズの) 整数 を 16 進の文字列に変換します。結果は Python で有効な表現になります。注意: 値は常に符号無しのリテラルになります。例えば、32 ビットのマシンでは、`hex(-1)` は `'0xffffffff'` になります。同じワードサイズを持つ計算機で評価された場合、この文字列は -1 になります; 異なるワードサイズの計算機では、巨大な正の値に折り返されたり、例外 `OverflowError` を送出的るかもしれません。

**id(object)**

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

**input([prompt])**

`eval(raw_input(prompt))` と同じです。警告: この関数はユーザのエラーに対して安全ではありません! この関数では、入力には有効な Python の式であると期待しています; 入力が構文的に正しくない場合、`SyntaxError` が送付されます。式を評価する際にエラーが生じた場合、他の例外も送付されるかもしれません。(一方、この関数は時に、熟練者がすばやくスクリプトを書く際に必要なまさにそのものです)

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

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

**int([x[, radix]])**

文字列または数値を通常の整数に変換します。引数が文字列の場合、Python 整数として表現可能な十進の数でなければなりません。符号が付いていてもかまいません。また、空白文字中に埋め込まれていてもかまいません。`radix` 引数は変換の基数を表し、範囲 [2, 36] の整数またはゼロをとることができます。`radix` がゼロの場合、文字列の内容から適切な基数を推測します; 変換は整数リテラルと同じです。`radix` が指定されており、`x` が文字列でない場合、`TypeError` が送付されます。それ以外の場合、引数は通常整数、長整数、または浮動小数点数をとることができます。浮動小数点数から整数

へ変換では (ゼロ方向に) 値を丸めます。引数が通常整数の範囲を超えている場合、長整数が代わりに返されます。引数が与えられなかった場合、0 を返します。

`isinstance(object, classinfo)`

引数 *object* が引数 *classinfo* のインスタンスであるか、(直接または間接的な) サブクラスのインスタンスの場合に真を返します。また、*classinfo* が型オブジェクトであり、*object* がその型のオブジェクトである場合にも真を返します。*object* がクラスインスタンスや与えられた型のオブジェクトでない場合、この関数は常に偽を返します。*classinfo* をクラスオブジェクトでも型オブジェクトにもせず、クラスや型オブジェクトからなるタプルや、そういったタプルを再帰的に含むタプル (他の配列型は受理されません) でもかまいません。*classinfo* がクラス、型、クラスや型からなるタプル、そういったタプルが再帰構造をとっているタプルのいじれでもない場合、例外 `TypeError` が送出されます。2.2 で変更された仕様: 型情報をタプルにした形式のサポートが追加されました。

`issubclass(class, classinfo)`

*class* が *classinfo* の (直接または間接的な) サブクラスである場合に真を返します。クラスはそのクラス自体のサブクラスと *classinfo* はクラスオブジェクトからなるタプルでもよく、この場合には *classinfo* のすべてのエントリが調べられます。その他の場合では、例外 `TypeError` が送出されます。2.3 で変更された仕様: 型情報からなるタプルへのサポートが追加されました

`iter(o[, sentinel])`

イテレータオブジェクトを返します。2 つ目の引数があるかどうかで、最初の引数の解釈は非常に異なります。2 つ目の引数がない場合、*o* は反復プロトコル (`__iter__()` メソッド) か、配列プロトコル (引数が 0 から開始する `__getitem__()` メソッド) をサポートする集合オブジェクトでなければなりません。これらのプロトコルが両方ともサポートされていない場合、`TypeError` が送出されます。2 つ目の引数 *sentinel* が与えられていれば、*o* は呼び出し可能なオブジェクトでなければなりません。この場合に生成されるイテレータは、`next()` を呼び出す毎に *o* を引数無しで呼び出します。返された値が *sentinel* と等しければ、`StopIteration` が送出されます。そうでない場合、戻り値がそのまま返されます。2.2 で追加された仕様です。

`len(s)`

オブジェクトの長さ (要素の数) を返します。引数は配列型 (文字列、タプル、またはリスト) か、マップ型 (辞書) です。

`list([sequence])`

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

`locals()`

現在のローカルシンボルテーブルを表す辞書を更新して返します。警告: この辞書の内容は変更してはいけません; 値を変更しても、インタプリタが使うローカル変数の値には影響しません。

`long([x[, radix]])`

文字列または数値を長整数値に変換します。引数が文字列の場合、Python 整数として表現可能な十進の数でなければなりません。符号が付いていてもかまいません。また、空白文字中に埋め込まれていてもかまいません; これは `string.atol(x)` と同様の動作です。*radix* 引数は `int()` と同じように解釈され、*x* が文字列の時だけ与えることができます。それ以外の場合、引数は通常整数、長整数、または浮動小数点数をとることができ、同じ値の長整数が返されます。浮動小数点数から整数へ変換では (ゼロ方向に) 値を丸めます。引数が与えられなかった場合、0L を返します。

`map(function, list, ...)`

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

`max(s[, args...])`

単一の引数 *s* の場合、空でない配列 (文字列、タプルまたはリスト) の要素のうち最大のものを返します。1 個よりも引数が多い場合、引数間で最大のものを返します。

`min(s[, args...])`

単一の引数 *s* の場合、空でない配列 (文字列、タプルまたはリスト) の要素のうち最小のものを返します。1 個よりも引数が多い場合、引数間で最小のものを返します。

`object()`

ユーザ定義の属性やメソッドを持たない、新しいオブジェクトを返します。`object()` は新スタイルのクラスの、基底クラスです。これは、新スタイルのクラスのインスタンスに共通のメソッド群を持ちます。2.2 で追加された仕様です。

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

`oct(x)`

(任意のサイズの) 整数を 8 進の文字列に変換します。結果は有効な Python の表現形式になります。注意: 値は常に符号無しのリテラルになります。例えば、32 ビットのマシンでは、`oct(-1)` は `'037777777777'` になります。同じワードサイズを持つ計算機で評価された場合、この文字列は -1 になります; 異なるワードサイズの計算機では、巨大な正の値に折り返されたり、例外 `OverflowError` を送出するかもしれません。

`open(filename[, mode[, bufsize]])`

前述の関数 `file()` の別名です。

`ord(c)`

1 文字からなる文字列または Unicode 文字の ASCII 値を返します。例えば、`ord('a')` は整数 97 を返し、`ord(u'\u2020')` は 8224 を返します。この値は文字列に対する `chr()` の逆であり、Unicode 文字に対する `unichr()` の逆です。

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

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

`property([fget[, fset[, fdel[, doc]]]])`

新しい形式のクラス (object から導出されたクラス) におけるプロパティ属性を返します。

*fget* は属性値を取得するための関数で、同様に *fset* は属性値を設定するための関数です。また、*fdel*

は属性を削除するための関数です。以下に属性 `x` を扱う典型的な利用法を示します:

```
class C(object):
    def getx(self): return self.__x
    def setx(self, value): self.__x = value
    def delx(self): del self.__x
    x = property(getx, setx, delx, "I'm the 'x' property.")
```

2.2 で追加された仕様です。

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

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

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

**raw\_input**(`[prompt]`)

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

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

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

**reduce**(`function, sequence[, initializer]`)

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

**reload**(`module`)



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

いくつか補足説明があります:

モジュールは文法的に正しいが、その初期化には失敗した場合、そのモジュールの最初の `import` 文はモジュール名をローカルにはバインドしませんが、(部分的に初期化された) モジュールオブジェクトを `sys.modules` に記憶します。従って、モジュールをロードしなおすには、`reload()` する前にまず `import` (モジュールの名前を部分的に初期化されたオブジェクトにバインドします) を再度行わなければならない。

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

組み込みモジュールや動的にロードされるモジュールを再ロードすることは、不正なやり方ではありませんが、一般的にそれほど便利ではありません。例外は `sys`、`__main__` および `__builtin__` です。しかしながら、多くの場合、拡張モジュールは 1 度以上初期化されるようには設計されておらず、再ロードされた場合には何らかの理由で失敗するかもしれません。

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

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

**repr**(*object*)

オブジェクトの印字可能な表現を含む文字列を返します。これは型変換で得られる (逆クオートの) 値と同じです。通常関数としてこの操作にアクセスできるとたまに便利です。この関数は多くの型について、`eval()` に渡されたときに同じ値を持つようなオブジェクトを表す文字列を生成しようとします。

**round**(*x*[, *n*])

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

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

`getattr()` と対をなす関数です。引数はそれぞれオブジェクト、文字列、そして任意の値です。文字列はすでに存在する属性の名前でも、新たな属性の名前でもかまいません。この関数は指定した値を指定した属性に関連付けますが、指定したオブジェクトにおいて可能な場合に限りです。例えば、`setattr(x, 'foobar', 123)` は `x.foobar = 123` と等価です。

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

`range(start, stop, step)` で指定されるインデクスの集合を表すスライスオブジェクトを返します。

`range(start)` スライスオブジェクトを返します。引数 *start* および *step* は標準では `None` です。スライスオブジェクトは読み出し専用の属性 *start*、*stop* および *step* を持ち、これらは単に引数で使われた値 (または標準の値) を返します。これらの値には、その他のはっきりとした機能はありません; しかしながら、これらの値は Numerical Python およびその他のサードパーティによる拡張で利用されています。スライスオブジェクトは拡張されたインデックス指定構文が使われる際にも生成されます。例えば: `'a[start:stop:step]'` や `'a[start:stop, i]'` です。

**staticmethod(*function*)**

*function* の静的メソッドを返します。

静的メソッドは暗黙の第一引数を受け取りません。静的メソッドの宣言は、以下のように書き慣わされます:

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

このメソッドはクラスで呼び出すこと (例えば `C.f()`) も、インスタンスとして呼び出すこと (例えば `C().f()`) もできます。インスタンスはそのクラスが何であるかを除いて無視されます。

Python における静的メソッドは Java や C++ における静的メソッドと類似しています。より進んだ概念については、`classmethod()` を参照してください。2.2 で追加された仕様です。

**str([*object*])**

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

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

*start* と *sequence* の要素を左から右へ加算してゆき、総和を返します。*start* はデフォルトで 0 です。*sequence* の要素は通常は数値で、文字列であってはなりません。文字列からなる配列を結合する高速で、正しい方法は `" .join(sequence)` です。`sum(range(n), m)` は `reduce(operator.add, range(n), m)` と同等です。2.3 で追加された仕様です。

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

*type* の上位クラスを返します。返された上位クラスオブジェクトが非バインドの場合、二つめの引数は省略されます。二つめの引数がオブジェクトの場合、`isinstance(obj, type)` は真でなくてはなりません。二つ目の引数が型オブジェクトの場合、`issubclass(type2, type)` は真でなくてはなりません。`super()` は新スタイルのクラスにのみ機能します。

協調する上位クラスのメソッドを呼び出す典型的な利用法を以下に示します:

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

2.2 で追加された仕様です。

**tuple([*sequence*])**

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

**type(*object*)**

*object* の型を返します。返される値は型 オブジェクトです。標準モジュール `types` は、組み込み名を持っていない全ての組み込み型の名前を定義しています。引数が与えられない場合、新しい空のタプル `()` を返します。

例えば:

```
>>> import types
>>> x = 'abc'
>>> if type(x) is str: print "It's a string"
...
It's a string
>>> def f(): pass
...
>>> if type(f) is types.FunctionType: print "It's a function"
...
It's a function
```

オブジェクトの型のテストには、組み込み関数 `isinstance()` が推奨されています。

**`unichr(i)`**

Unicode におけるコードが整数 *i* になるような文字 1 文字からなる Unicode 文字列を返します。例えば、`unichr(97)` は文字列 `u'a'` を返します。この関数は Unicode 文字列に対する `ord()` の逆です。引数は両端を含めて `[0..65535]` の範囲でなければなりません。それ以外の値に対しては `ValueError` が送出されます。2.0 で追加された仕様です。

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

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

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

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

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

2.0 で追加された仕様です。 2.2 で変更された仕様: `__unicode__()` のサポートが追加されました

**`vars([object])`**

引数無しでは、現在のローカルシンボルテーブルに対応する辞書を返します。モジュール、クラス、またはクラスインスタンスオブジェクト (またはその他 `__dict__` 属性を持つもの) を引数として与えた場合、そのオブジェクトのシンボルテーブルに対応する辞書を返します。返される辞書は変更すべきではありません: 変更が対応するシンボルテーブルにもたらす影響は未定義です。<sup>5</sup>

---

<sup>5</sup>現在の実装では、ローカルな値のバインディングは通常は影響を受けませんが、(モジュールのような) 他のスコープから取り出した値は影響を受けるかもしれません。またこの実装は変更されるかもしれません。

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

この関数は `range()` に非常によく似ていますが、リストの代わりに“xrange オブジェクト”を返します。このオブジェクトは不透明な配列型で、対応するリストと同じ値を持ちますが、それらの値全てを同時に記憶しません。`range()` に対する `xrange()` の利点は微々たるものです (`xrange()` は要求に応じて値を生成するからです) ただし、メモリ量の厳しい計算機で巨大な範囲の値を使う時や、(ループがよく `break` で中断されるといったように) 範囲中の全ての値を使うとは限らない場合はその限りではありません。

**zip**(*seq1, ...*)

この関数はタプルのリストを返します。このリストの *i* 番目のタプルは各引数の配列中の *i* 番目の要素を含みます。少なくとも 1 つの配列が引数に必要で、そうでない場合には `TypeError` が送出されます。返されるリストは引数の配列のうち長さが最小のものの長さに切り詰められます。引数の配列が全て同じ長さの際には、`zip()` は初期値引数が `None` の `map()` と似ています。引数が単一の配列の場合、1 要素のタプルからなるリストを返します。2.0 で追加された仕様です。

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

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

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

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

引数 *function* は呼び出しができるオブジェクト (ユーザ定義および組み込みの関数またはメソッド、またはクラスオブジェクト) でなければなりません。*args* は配列型でなくてはなりません。*function* は引数リスト *args* を使って呼び出されます; 引数の数はタプルの長さになります。オプションの引数 *keywords* を与える場合、*keywords* は文字列のキーを持つ辞書でなければなりません。これは引数リストの最後に追加されるキーワード引数です。`apply()` の呼び出しは、単なる `function(args)` の呼び出しとは異なります。というのは、`apply()` の場合、引数は常に一つだからです。`apply()` は `function(*args, **keywords)` を使うのと等価です。上のような“拡張された関数呼び出し構文”は `apply()` と全く等価なので、必ずしも `apply()` を使う必要はありません。リリース 2.3 以降で撤廃された仕様です。上で述べられたような拡張呼び出し構文を使ってください。

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

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

**coerce**(*x, y*)

二つの数値型の引数を共通の型に変換して、変換後の値からなるタプルを返します。変換に使われる規則は算術演算における規則と同じです。

**intern**(*string*)

*string* を“隔離”された文字列のテーブルに入力し、隔離された文字列を返します – この文字列は *string* 自体のコピーです。隔離された文字列は辞書検索のパフォーマンスを少しだけ向上させるのに有効です – 辞書中のキーが隔離されており、検索するキーが隔離されている場合、(ハッシュ化後の) キーの比較は文字列の比較ではなくポインタの比較で行うことができるからです。通常、Python プログラム内で利用されている名前は自動的に隔離され、モジュール、クラス、またはインスタンス属性を保

持するための辞書は隔離されたキーを持っています。 2.3 で変更された仕様: 隔離された文字列の有効期限は (Python 2.2 またはそれ以前は永続的でしたが) 永続的ではなくなりました; `intern()` の恩恵を受けるためには、`intern()` の返す値に対する参照を保持しなければなりません

## 2.3 組み込み型

以下のセクションでは、インタプリタに組み込まれている標準の型について記述します。これまでの Python の歴史では、組み込み型はオブジェクト指向における継承を行う際に離型にできないという点で、ユーザ定義型とは異なっていました。リリース 2.2 からは状況が変わり始めましたが、目標とするユーザ定義型と組み込み方の一元化はまだまだ完成の域には達していません。

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

演算によっては、複数の型でサポートされているものがあります; 特に、全てのオブジェクトについて、比較、真値テスト、(``...`` 形式での) 文字列への変換を行うことができます。 `print` 使われた場合、最後の文字列への変換が暗黙のうちに行われます。(Information on `print` 文 やその他の文に関する情報は *Python* リファレンスマニュアル および *Python* チュートリアルで見つけることができます。)

### 2.3.1 真値テスト

どのオブジェクトも `if` または `while` 条件文の中や、以下のブール演算における被演算子として真値テストを行うことができます。以下の値は偽であると見なされます:

- `None`
- `False`
- 数値型におけるゼロ。例えば `0`、`0L`、`0.0`、`0j`。
- 空の配列型。例えば `"`、`()`、`[]`。
- 空のマッピング型。例えば `{}`。
- `__nonzero__()` または `__len__()` メソッドが定義されているようなユーザ定義クラスのインスタンスで、それらのメソッドが整数値ゼロまたは `bool` 値の `False` を返すとき。<sup>6</sup>

それ以外の値は全て真であると見なされます — 従って、ほとんどの型のオブジェクトは常に真です。

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

### 2.3.2 ブール演算

以下にブール演算子を示します。優先度の低いものから順に並んでいます。:

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

<sup>6</sup>これらの特殊なメソッドに関する追加情報は *Python* リファレンスマニュアルに記載されています。



注釈:

- (1) これらの演算子は、演算を行う上で必要がない限り、二つ目の引数を評価しません。
- (2) 'not' は非ブール演算子よりも低い演算優先度なので、`not a == b` は `not (a == b)` と評価され、`a == not b` は構文エラーとなります。

### 2.3.3 比較

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

以下のテーブルに比較演算をまとめます:

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

注釈:

- (1) `<>` および `!=` は同じ演算子を別の書き方にしたものです。 `!=` のほうが望ましい書き方です; `<>` は廃止すべき書き方です。

数値型間の比較が文字列間の比較でないかぎり、異なる型のオブジェクトを比較しても等価になることはありません; これらのオブジェクトの順番付けは一貫してはいませんが任意のものです (従って要素の型が一様でない配列をソートした結果は一貫したものになります)。さらに、(例えばファイルオブジェクトのように) 型によっては、その型の2つのオブジェクトの不等性だけの、縮退した比較の概念しかサポートしないものもあります。繰り返しますが、そのようなオブジェクトも任意の順番付けをされていますが、それは一貫したものです。被演算子が複素数の場合、演算子 `<`、`<=`、`>` および `>=` は例外 `TypeError` を送出します。

あるクラスのインスタンス間の比較は、そのクラスで `__cmp__()` メソッドが定義されていない限り等しくなりません。このメソッドを使ってオブジェクトの比較方法に影響を及ぼすための情報については *Python* リファレンスマニュアルを参照してください。

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

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

### 2.3.4 数値型

4つの異なる数値型があります: 通常の整数型、長整数型、浮動小数点型、および複素数型です。

さらに、bool 方は通常の整数方のサブタイプです。通常の整数 (単に 整数型 と呼ばれます) は C では long を使って実装されており、少なくとも 32 ビットの精度があります。長整数型には精度の制限がありません。浮動小数点型は C では double を使って実装されています。しかし使っている計算機が何であるか分からないなら、これらの数値型の精度に関して断言はできません。

複素数型は実数部と虚数部を持ち、それぞれの C では double を使って実装されています。複素数  $z$  から実数および虚数部を取り出すには、`z.real` および `z.imag` を使います。

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

Python は型混合の演算を完全にサポートします: ある 2 項演算子が互いに異なる数値型の被演算子を持つ場合、より“制限された”型の被演算子は他方の型に合わせて広げられます。ここで通常の整数は長整数より制限されており、長整数は浮動小数点数より制限されており、浮動小数点は複素数より制限されています。型混合の数値間での比較も同じ規則に従います。<sup>7</sup> コンストラクタ `int()`、`long()`、`float()`、および `complex()` を使って、特定の型の数を生成することができます。

全ての数値型 (`complex` は例外) は以下の演算をサポートします。これらの演算は優先度の低いものから順に並べられています (同じボックスにある演算は同じ優先度を持っています; 全ての数値演算は比較演算よりも高い優先度を持っています):

演算	結果	注釈
$x + y$	$x$ と $y$ の加算	
$x - y$	$x$ と $y$ の減算	
$x * y$	$x$ と $y$ の乗算	
$x / y$	$x$ と $y$ の除算	(1)
$x \% y$	$x / y$ の剰余	(4)
$-x$	$x$ の符号反転	
$+x$	$x$ の符号不変	
<code>abs(x)</code>	$x$ の絶対値または大きさ	
<code>int(x)</code>	$x$ の通常整数への変換	(2)
<code>long(x)</code>	$x$ の長整数への変換	(2)
<code>float(x)</code>	$x$ の浮動小数点数への変換	
<code>complex(re, im)</code>	実数部 $re$ 、虚数部 $im$ の複素数。 $im$ のデフォルト値はゼロ。	
<code>c.conjugate()</code>	複素数 $c$ の共役複素数	
<code>divmod(x, y)</code>	$(x / y, x \% y)$ からなるペア	(3)
<code>pow(x, y)</code>	$x$ の $y$ 乗	
$x ** y$	$x$ の $y$ 乗	

注釈:

- (1) (通常および長) 整数の割り算では、結果は整数になります。この場合値は常にマイナス無限大の方向に丸められます: つまり、 $1/2$  は  $0$ 、 $(-1)/2$  は  $-1$ 、 $1/(-1)$  は  $-1$ 、そして  $(-1)/(-2)$  は  $0$  になります。被演算子の両方が長整数の場合、計算値に関わらず結果は長整数で返されるので注意してください。
- (2) 浮動小数点数から (通常または長) 整数への変換では、C におけるのと同様の値の丸めまたは切り詰め

<sup>7</sup>この結果として、リスト `[1, 2]` は `[1.0, 2.0]` と等しいと見なされます。タプルの場合も同様です

が行われるかもしれませんが; きちんと定義された変換については、`math` モジュールの `floor()` および `ceil()` を参照してください。

(3) 完全な記述については、[2.1](#)、“組み込み関数”を参照してください。

(4) 複素数の切り詰め除算演算子、モジュロ演算子、および `divmod()`。

リリース 2.3 以降で撤廃された仕様です。適切であれば、`abs()` を使って浮動小数点に変換してください。

整数型におけるビット文字列演算

通常および長整数型ではさらに、ビット文字列に対してのみ意味のある演算をサポートしています。負の数はその値の 2 の補数の値として扱われます (長整数の場合、演算操作中にオーバーフローが起こらないように十分なビット数があるものと仮定します)。

2 進のビット単位演算は全て、数値演算よりも低く、比較演算子よりも高い優先度です; 単項演算 `'` は他の単項数値演算 (`+` および `-`) と同じ優先度です。

以下のテーブルでは、ビット文字列演算を優先度の低いものから順に並べています (同じボックス内の演算は同じ優先度です):

演算	結果	注釈
$x \mid y$	ビット単位の $x$ と $y$ の 論理和	
$x \wedge y$	ビット単位の $x$ と $y$ の 排他的論理和	
$x \& y$	ビット単位の $x$ と $y$ の 論理積	
$x \ll n$	$x$ の $n$ ビット左シフト	(1), (2)
$x \gg n$	$x$ の $n$ ビット右シフト	(1), (3)
$\sim x$	$x$ のビット反転	

注釈:

- (1) 負値のシフト数は不正であり、`ValueError` が送出されます。
- (2)  $n$  ビットの左シフトは、オーバーフローチェックを行わない `pow(2, n)` による乗算と等価です。
- (3)  $n$  ビットの右シフトは、オーバーフローチェックを行わない `pow(2, n)` による除算と等価です。

2.3.5 イテレータ型

2.2 で追加された仕様です。

Python はコンテナの内容にわたって反復処理を行う概念をサポートしています。この概念は 2 つの別々のメソッドを使って実装されています; これらのメソッドはユーザ定義のクラスで反復を行えるようにするために使われます。後に詳しく述べる配列型はすべて反復処理メソッドをサポートしています。

以下はコンテナオブジェクトに反復処理をサポートさせるために定義しなければならないメソッドです:

`__iter__()`

イテレータオブジェクトを返します。イテレータオブジェクトは以下で述べるイテレータプロトコルをサポートする必要があります。あるコンテナが異なる形式の反復処理をサポートする場合、それらの反復処理形式のイテレータを特定の要求するようなメソッドを追加することができます (複数の形式での反復処理をサポートするようなオブジェクトとして木構造の例があります。木構造は幅優先走査と深さ優先走査の両方をサポートします)。このメソッドは Python/C API において Python オブジェクトを表す型構造体の `tp_iter` スロットに対応します。



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

`__iter__()`

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

`next()`

コンテナ内の次の要素を返します。もう要素が残っていない場合、例外 `StopIteration` を送出します。このメソッドは Python/C API において Python オブジェクトを表す型構造体の `tp_iternext` スロットに対応します。

Python では、いくつかのイテレータオブジェクトを定義しています。これらは一般のおよび特殊化された配列型、辞書型、そして他のさらに特殊化された形式をサポートします。特殊型であることはイテレータプロトコルの実装が特殊になること以外は重要なことではありません。

このプロトコルの趣旨は、一度イテレータの `next()` メソッドが `StopIteration` 例外を送出した場合、以降の呼び出しでもずっと例外を送出しつづけるところにあります。この特性に従わないような実装は変則であるとみなされます (この制限は Python 2.3 で追加されました; Python 2.2 では、この規則に従うと多くのイテレータが変則となります)。

Python におけるジェネレータ (generator) は、イテレータプロトコルを実装する簡便な方法を提供します。コンテナオブジェクトの `__iter__()` メソッドがジェネレータとして実装されていれば、メソッドは `__iter__()` および `next()` メソッドを提供するイテレータオブジェクト (技術的にはジェネレータオブジェクト) を自動的に返します。

## 2.3.6 配列型

組み込み型には6つの配列型があります: 文字列、ユニコード文字列、リスト、タプル、バッファ、そして `xrange` オブジェクトです。

文字列リテラルは `'xyzzy'`、`"frobozz"` といったように、単引用符または二重引用符の中に書かれます。文字列リテラルについての詳細は、*Python* リファレンスマニュアルの第2章を読んで下さい。Unicode 文字列はほとんど文字列と同じですが、`u'abc'`、`u"def"` といったように先頭に文字 `'u'` を付けて指定します。リストは `[a, b, c]` のように要素をコンマで区切り角括弧で囲って生成します。タプルは `a, b, c` のようにコンマ演算子で区切って生成します (角括弧の中には入れません)。丸括弧で囲っても囲わなくてもかまいませんが、空のタプルは `()` のように丸括弧で囲わなければなりません。要素が一つのタプルでは、例えば `(d,)` のように、要素の後ろにコンマをつけなければなりません。

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

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

ほとんどの配列型は以下の演算操作をサポートします。`'in'` および `'not in'` は比較演算とおなじ優先度を持っています。`'+'` および `'*'` は対応する数値演算とおなじ優先度です。<sup>8</sup>

以下のテーブルは配列型の演算を優先度の低いものから順に挙げたものです (同じボックス内の演算は同じ優先度です)。テーブル内の *s* および *t* は同じ型の配列です; *n*、*i* および *j* は整数です:

<sup>8</sup>パーザが被演算子の型を識別できるようにするために、このような優先度でなければならないのです。

演算	結果	注釈
<code>x in s</code>	<code>s</code> のある要素 <code>x</code> と等しい場合 1、そうでない場合 0	(1)
<code>x not in s</code>	<code>s</code> のある要素が <code>x</code> と等しい場合 0、そうでない場合 1	(1)
<code>s + t</code>	<code>s</code> および <code>t</code> の結合	
<code>s * n, n * s</code>	<code>s</code> の浅いコピー <code>n</code> 個からなる結合	(2)
<code>s[i]</code>	<code>s</code> の 0 から数えて <code>i</code> 番目の要素	(3)
<code>s[i:j]</code>	<code>s</code> の <code>i</code> 番目から <code>j</code> 番目までのスライス	(3), (4)
<code>s[i:j:k]</code>	<code>s</code> の <code>i</code> 番目から <code>j</code> 番目まで、 <code>k</code> 毎のスライス	(3), (5)
<code>len(s)</code>	<code>s</code> の長さ	
<code>min(s)</code>	<code>s</code> の最小の要素	
<code>max(s)</code>	<code>s</code> の最大の要素	

注釈:

- (1) `s` が文字列または Unicode 文字列の場合、演算操作 `in` および `not in` は部分文字列の一致テストと同じように動作します。バージョン 2.3 以前の Python では、`x` は長さ 1 の文字列でした。Python 2.3 以降では、`x` はどの長さでもかまいません。
- (2) `n` が 0 以下の値の場合、0 として扱われます (これは `s` と同じ型の空の配列を表します)。コピーは浅いコピーなので注意してください; 入れ子になったデータ構造はコピーされません。これは Python に慣れていないプログラマをよく悩ませます。例えば以下のコードを考えます:

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

上のコードでは、`lists` はリスト `[[]]` (空のリストを唯一の要素として含んでいるリスト) の 3 つのコピーを要素とするリストです。しかし、リスト内の要素に含まれているリストは各コピー間で共有されています。以下のようにすると、異なるリストを要素とするリストを生成できます:

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

- (3) `i` または `j` が負の数の場合、インデックスは文字列の末端からの相対インデックスになります: `len(s) + i` または `len(s) + j` が代入されます。しかし `-0` は 0 のままなので注意してください。
- (4) `s` の `i` から `j` へのスライスは `i <= k < j` となるようなインデックス `k` を持つ要素からなる配列として定義されます。`i` または `j` が `len(s)` よりも大きい場合、`len(s)` を使います。`i` が省略された場合、0 を使います。`j` が省略された場合、`len(s)` を使います。`i` が `j` 以上の場合、スライスは空の配列になります。
- (5) `s` の `i` 番目から `j` 番目まで `k` 毎のスライスは、`0 <= n < abs(i-j)` となるような、インデックス `x = i + n*k` を持つ要素からなる配列として定義されます。`i` または `j` `len(s)` より大きい場合、`len(s)` を使

います。 *i* または *j* を省略した場合、“最後” (*k* の符号に依存) を示す値を使います。 *k* はゼロにできないので注意してください。

## 文字列メソッド

以下は 8 ビット文字列および Unicode オブジェクトでサポートされるメソッドです:

**capitalize()**

最初の文字を大文字にした文字列のコピーを返します。

**center(*width*)**

*width* の長さをもつ中央寄せされた文字列を返します。パディングには空白が使われます。

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

文字列 *S* [*start*:*end*] 中に部分文字列 *sub* が出現する回数を返します。オプション引数 *start* および *end* はスライス表記と同じように解釈されます。

**decode([*encoding* [, *errors*]])**

codec に登録された文字コード系 *encoding* を使って文字列をデコードします。*encoding* は標準でデフォルトの文字列エンコーディングになります。標準とは異なるエラー処理を行うために *errors* を与えることができます。標準のエラー処理は 'strict' で、エンコードに関するエラーは `ValueError` を送出します。他に利用できる値は 'ignore' および 'replace' です。2.2 で追加された仕様です。

**encode([*encoding* [, *errors*]])**

文字列のエンコードされたバージョンを返します。標準のエンコーディングは現在のデフォルト文字列エンコーディングです。標準とは異なるエラー処理を行うために *errors* を与えることができます。標準のエラー処理は 'strict' で、エンコードに関するエラーは `ValueError` を送出します。他に利用できる値は 'ignore' および 'replace' です。2.0 で追加された仕様です。

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

文字列の一部が *suffix* で終わるときに `True` を返します。そうでない場合 `False` を返します。オプション引数 *start* がある場合、文字列の *start* から比較を始めます。*end* がある場合、文字列の *end* で比較を終えます。

**expandtabs([*tabsize*])**

全てのタブ文字が空白で展開された文字列のコピーを返します。*tabsize* が与えられていない場合、タブ幅は 8 文字分と仮定します。

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

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

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

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

**isalnum()**

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

**isalpha()**

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

**isdigit()**

文字列中に数字しかない場合には真を返し、その他の場合は偽を返します。

**islower()**

文字列中の大小文字の区別のある文字全てが小文字で、かつ 1 文字以上ある場合には真を返し、そうでない場合は偽を返します。

**isspace()**

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

**istitle()**

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

**isupper()**

文字列中の大小文字の区別のある文字全てが大文字で、かつ 1 文字以上ある場合には真を返し、そうでない場合は偽を返します。

**join(seq)**

配列 *seq* 中の文字列を結合した文字列を返します。文字列を結合するときの区切り文字は、このメソッドを適用する対象の文字列になります。

**ljust(width)**

*width* の長さをもつ左寄せした文字列を返します。パディングには空白が使われます。*width* が `len(s)` よりも小さい場合、元の文字列が返されます。

**lower()**

文字列をコピーし、小文字に変換して返します。

**rstrip([chars])**

文字列をコピーし、文字列の先頭部分を除去して返します。*chars* が省略されるか `None` の場合、空白文字が除去されます。*chars* が与えられていてかつ `None` でない場合、*chars* は文字列でなければなりません; このメソッドを適用した対象の文字列の先頭部分から *chars* 中の文字が除去されます。  
2.2.2 で変更された仕様: 引数 *chars* をサポートしました

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

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

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

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

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

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

**rjust(width)**

*width* の長さをもつ右寄せした文字列を返します。パディングには空白が使われます。*width* が `len(s)` よりも小さい場合、元の文字列が返されます。

**rstrip([chars])**

文字列をコピーし、文字列の末尾部分を除去して返します。*chars* が省略されるか `None` の場合、空白文字が除去されます。*chars* が与えられていてかつ `None` でない場合、*chars* は文字列でなければなりません; このメソッドを適用した対象の文字列の末端部分から *chars* 中の文字が除去されます。  
2.2.2 で変更された仕様: 引数 *chars* をサポートしました

**split([sep[, maxsplit]])**

*sep* を単語の境界として文字列を単語に分割し、分割された単語からなるリストを返します。*maxsplit* が与えられた場合、最大で *maxsplit* 個になるように分割します。*sep* が指定されていないか `None` の場合、すべての空白文字が単語の境界になります。

**splitlines**(*[keepends]*)

文字列を改行部分で分解し、各行からなるリストを返します。*keepends* が与えられていて、かつその値が真でない限り、返されるリストには改行文字は含まれません。

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

文字列の一部が *prefix* で始まるときに `True` を返します。そうでない場合 `False` を返します。オプション引数 *start* がある場合、文字列の *start* から比較を始めます。*end* がある場合、文字列の *end* で比較を終えます。

**strip**(*[chars]*)

文字列をコピーし、文字列の先頭および末尾部分を除去して返します。*chars* が省略されるか `None` の場合、空白文字が除去されます。*chars* が与えられていてかつ `None` でない場合、*chars* は文字列でなければなりません; このメソッドを適用した対象の文字列の両端から *chars* 中の文字が除去されます。2.2.2 で変更された仕様: 引数 *chars* をサポートしました

**swapcase**()

文字列をコピーし、大文字は小文字に、小文字は大文字に変換して返します。

**title**()

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

**translate**(*table**[, deletechars]*)

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

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

**upper**()

文字列をコピーし、大文字に変換して返します。

**zfill**(*width*)

数値文字列の左側をゼロ詰めし、幅 *width* にして返します。*width* が `len(s)` よりも短い場合もこの文字列自体が返されます。2.2.2 で追加された仕様です。

## 文字列フォーマット操作

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



`format` が単一の引数しか要求しない場合、`values` はタプルでない単一のオブジェクトでもかまいません。<sup>9</sup>それ以外の場合、`values` はフォーマット文字列中で指定された項目と正確に同じ数の要素からなるタプルか、単一のマップオブジェクトでなければなりません。

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

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

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

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

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

変換フラグ文字を以下に示します:

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

精度長変換子として、`h`、`l`、および `L` を使うことができますが、Python では必要ないため無視されます。

変換型を以下に示します:

<sup>9</sup>従って、一個のタプルだけをフォーマット出力したい場合には出力したいタプルを唯一の要素とする単一のタプルを `values` に与えなくてはなりません。

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

注釈:

- (1) 別形式の出力にした場合、変換結果の先頭の数字がゼロ ('0') でないときには、数字の先頭と左側のパディングとの間にゼロを挿入します。
- (2) 別形式にした場合、変換結果の先頭の数字がゼロでないときには、数字の先頭と左側のパディングとの間に '0x' または '0X' (フォーマット文字が 'x' か 'X' かに依存します) が挿入されます。
- (3) %r 変換は Python 2.0 で追加されました。
- (4) オブジェクトや与えられた書式が unicode 文字列の場合、変換後の文字列も unicode になります。

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

安全上の理由から、浮動小数点数の精度は 50 桁でクリップされます; 絶対値が 1e25 を超える値の %f による変換は %g 変換で置換されます<sup>10</sup>。その他のエラーは例外を送出します。

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

## XRange 型

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

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

## 変更可能な配列型

リストオブジェクトはオブジェクト自体の変更を可能にする追加の操作をサポートします。他の変更可能な配列型 (を言語に追加する場合) も、それらの操作をサポートしなければなりません。文字列およびタプル

<sup>10</sup>この範囲に関する値はかなり適当なものです。この仕様は、正しい使い方では障害とならず、かつ特定のマシンにおける浮動小数点数の正確な精度を知らなくても、際限なく長く意味のない数字からなる文字列を印字しないですむようにするためのものです。

ルは変更不可能な配列型です: これらのオブジェクトは一度生成されたらそのオブジェクト自体を変更することができません。以下の操作は変更可能な配列型で定義されています (ここで  $x$  は任意のオブジェクトとします):

操作	結果	注釈
<code>s[i] = x</code>	$s$ の要素 $s$ を $x$ と入れ替えます	
<code>s[i:j] = t</code>	$s$ の $i$ から $j$ 番目までの要素を $t$ と入れ替えます	
<code>del s[i:j]</code>	<code>s[i:j] = []</code> と同じです	
<code>s[i:j:k] = t</code>	<code>s[i:j:k]</code> の要素を $t$ と入れ替えます	(1)
<code>del s[i:j:k]</code>	リストから <code>s[i:j:k]</code> の要素を削除します	
<code>s.append(x)</code>	<code>s[len(s):len(s)] = [x]</code> と同じです	(2)
<code>s.extend(x)</code>	<code>s[len(s):len(s)] = x</code> と同じです	(3)
<code>s.count(x)</code>	<code>s[i] == x</code> となる $i$ の個数を返します	
<code>s.index(x[, i[, j]])</code>	<code>s[k] == x</code> かつ $i \leq k < j$ となる最小の $k$ を返します。	(4)
<code>s.insert(i, x)</code>	$i \geq 0$ の場合の <code>s[i:i] = [x]</code> と同じです	(5)
<code>s.pop([i])</code>	$x = s[i]$ ; <code>del s[i]</code> ; <code>return x</code> と同じです	(6)
<code>s.remove(x)</code>	<code>del s[s.index(x)]</code> と同じです	(4)
<code>s.reverse()</code>	$s$ の値の並びを反転します	(7)
<code>s.sort(cmpfunc=None)</code>	$s$ の要素を並べ替えます	(7), (8), (9), (10)

Notes:

- (1)  $t$  は入れ替えるスライスと同じ長さでなければいけません。
- (2) かつての Python の C 実装では、複数パラメタを受理し、非明示的にそれらをタプルに結合していました。この間違った機能は Python 1.4 で廃用され、Python 2.0 の導入とともにエラーにするようになりました。
- (3)  $x$  がリストオブジェクトでない場合、例外を送出します。 `extend()` は実験的なメソッドであり、リスト以外の変更可能な配列型ではサポートされていません。
- (4)  $x$  が  $s$  中に見つからなかった場合 `ValueError` を送じます。 負のインデックスが二番目または三番目のパラメタとして `index()` メソッドに渡されると、これらの値にはスライスのインデックスと同様にリストの長さが加算されます。加算後もまだ負の場合、その値はスライスのインデックスと同様にゼロに切り詰められます。2.3 で変更された仕様: 以前は、`index()` は開始位置や終了位置を指定するのに負の数を使うことができませんでした
- (5) `insert()` の最初のパラメタとして負のインデックスが渡された場合、スライスのインデックスと同じく、リストの長さが加算されます。それでも負の値を取る場合、スライスのインデックスと同じく、0 に丸められます。2.3 で変更された仕様: 以前は、すべての負値は 0 に丸められていました。
- (5) 配列の先頭に新しい要素が追加されます。
- (6) `pop()` メソッドはリストおよびアレイ型のみでサポートされています。オプションの引数  $i$  は標準で  $-1$  なので、標準では最後の要素をリストから除去して返します。
- (7) `sort()` および `reverse()` メソッドは大きなリストを並べ替えたり反転したりする際、容量の節約のためにリストを直接変更します。副作用があることをユーザに思い出させるために、これらの操作は並べ替えまたは反転されたリストを返しません。



- (8) `sort()` メソッドはオプションの引数として比較関数をとります。この比較関数は二つの引数 (list items) をとり、最初の引数が二つ目の引数と比べてより小さいか、等しいか、より大きいかによって、それぞれ負、ゼロ、正の値を返します。この指定は並べ替え処理をかなり低速化するので注意してください;

たとえば、リストを逆順に並べ替えるためには、要素を反転して順序づけするような比較関数で `sort()` を使うより、`sort()` の後に `reverse()` を呼ぶほうがはるかに高速です。比較関数として `None` を渡すことは、比較関数なしで `sort()` を呼ぶのと同じ意味です。2.3 で変更された仕様: `None` を渡すのと、`varcmpfunc` を省略した場合とで、同等に扱うサポートを追加

`sort()` メソッドに `cmpfunc` 引数を使う例として、シーケンスのリストを、シーケンスの 2 つ目の要素でソートする場合を考えてみましょう。

```
def mycmp(a, b):
    return cmp(a[1], b[1])

mylist.sort(mycmp)
```

それなりにサイズの大きいデータ構造に対しては、より時間効率のよりアプローチが、頻繁に使われます。

```
tmplist = [(x[1], x) for x in mylist]
tmplist.sort()
mylist = [x for (key, x) in tmplist]
```

- (9) `sort()` メソッドが安定しているかどうかは言語上では定義されていません (比較結果が等しい要素の相対的な位置を変えないことを保証しているとき、並べ替えが安定しているといえます)。Python の C 実装では、Python 2.2 まで並べ替えはただ偶然に安定していました。Python 2.3 の C 実装は安定な `sort()` メソッドを導入しましたが、異なる Python の実装間やバージョンの間での可搬性を見込んだコードでは、この安定性に依存してはいけません。

- (10) リストが並べ替えられている間は、リストの変更はもとより、その値の閲覧すらその結果は未定義です。

Python 2.3 の C 実装では、この間リストは空に見えるようになり、並べ替え中にリストが変更されたことが検出されると `ValueError` が送出されます。

## 2.3.7 マップ型

*mapping* オブジェクトは変更不可能な値を任意のオブジェクトに対応付けます。対応付け事態は変更可能なオブジェクトです。現在のところは標準のマップ型、*dictionary* だけです。辞書のキーにはほとんど任意の値をつかうことができます。使うことができないのはリスト、辞書、その他の変更可能な型 (オブジェクトの一致ではなく、その値で比較されるような型) です。キーに使われた数値型は通常の数値比較規則に従います: 二つの数字を比較した時等価であれば (例えば 1 と 1.0 のように)、これらの値はお互いに同じ辞書のエントリを示すために使うことができます。

辞書は `key: value` からなるペアをカンマで区切ったリストを波括弧の中に入れて作ります。例えば: `{'jack': 4098, 'sjoerd': 4127}` または `{4098: 'jack', 4127: 'sjoerd'}` です。

以下の操作がマップ型で定義されています (ここで、*a* および *b* はマップ型で、*k* はキー、*v* および *x* は任意のオブジェクトです):

操作	結果	注釈
<code>len(a)</code>	<code>a</code> 内の要素の数です	
<code>a[k]</code>	キー <code>k</code> を持つ <code>a</code> の要素です	(1)
<code>a[k] = v</code>	<code>a[k]</code> を <code>v</code> に設定します	
<code>del a[k]</code>	<code>a</code> から <code>a[k]</code> を削除します	(1)
<code>a.clear()</code>	<code>a</code> から全ての要素を削除します	
<code>a.copy()</code>	<code>a</code> の (浅い) コピーです	
<code>a.has_key(k)</code>	<code>a</code> にキー <code>k</code> があれば <code>True</code> 、そうでなければ <code>False</code> です	
<code>k in a</code>	<code>a.has_key(k)</code> と同じです	(2)
<code>k not in a</code>	<code>not a.has_key(k)</code> と同じです	(2)
<code>a.items()</code>	<code>a</code> における (key, value) ペアのリストのコピーです	(3)
<code>a.keys()</code>	<code>a</code> におけるキーのリストのコピーです	(3)
<code>a.update(b)</code>	<code>for k in b.keys(): a[k] = b[k]</code>	
<code>a.fromkeys(seq[, value])</code>	<code>seq</code> からキーを作り、値が <code>value</code> であるような、新しい辞書を作成します	(7)
<code>a.values()</code>	<code>a</code> における値のリストのコピーです	(3)
<code>a.get(k[, x])</code>	<code>a[k]</code> if <code>k</code> in <code>a</code> , else <code>x</code>	(4)
<code>a.setdefault(k[, x])</code>	<code>a[k]</code> if <code>k</code> in <code>a</code> , else <code>x</code> (also setting it)	(5)
<code>a.pop(k[, x])</code>	<code>a[k]</code> if <code>k</code> in <code>a</code> , else <code>x</code> (and remove <code>k</code> )	(8)
<code>a.popitem()</code>	任意の (key, value) ペアを除去して返します	(6)
<code>a.iteritems()</code>	(key, value) ペアにわたるイテレータを返します	(2), (3)
<code>a.iterkeys()</code>	マップのキー列にわたるイテレータを返します	(2), (3)
<code>a.itervalues()</code>	マップの値列にわたるイテレータを返します	(2), (3)

注釈:

- (1) `k` がマップ内にない場合、例外 `KeyError` を送出します。
- (2) 2.2 で追加された仕様です。
- (3) キー及び値はランダムな順番でリストになっています。途中で辞書を変更せずに `items()`、`keys()`、`values()`、`iteritems()`、`iterkeys()`、and `itervalues()` を呼んだ場合、返されるリストは直接対応しています。これにより、(value, key) のペアを `zip()` を使って: `'pairs = zip(a.values(), a.keys())'` のように生成することができます。 `iterkeys()` および `itervalues()` メソッドの間でも同じ関係が成り立ちます: `'pairs = zip(a.itervalues(), a.iterkeys())'` は `pairs` と同じ値になります。同じリストを生成するもう一つの方法は `'pairs = [(v, k) for (k, v) in a.iteritems()]'` です。
- (4) `k` がマップ中になくても例外を送出せず、代わりに `x` を返します。 `x` はオプションです; `x` が与えられておらず、かつ `k` がマップ中になければ、`None` が返されます。
- (5) `setdefault()` は `get()` に似ていますが、`k` が見つからなかった場合、`x` が返されると同時に辞書の `k` に対する値として挿入されます。
- (6) `popitem()` は、集合アルゴリズムでよく行われるような、辞書を取り崩しながらの反復を行うのに便利です。
- (7) `fromkeys()` は、新しい辞書を返すクラスメソッドです。 `value` のデフォルト値は `None` です。 2.3 で追加された仕様です。
- (8) `pop()` は、デフォルト値が渡されず、かつ、キーが見つからない場合に、`KeyError` を送出します。 2.3 で追加された仕様です。

## 2.3.8 ファイルオブジェクト

ファイルオブジェクト は C の `stdio` パッケージを使って実装されており、2.1 節の“組み込み関数”で解説されている組み込みのコンストラクタ `file()` で生成することができます。<sup>11</sup>

ファイルオブジェクトはまた、`os.popen()` や `os.fdopen()`、ソケットオブジェクトの `makefile()` メソッドのような、他の組み込み関数およびメソッドによっても返されます。

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

ファイルは以下のメソッドを持ちます:

**close()**

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

**flush()**

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

**fileno()**

背後にある実装系がオペレーティングシステムに I/O 操作を要求するために用いる、整数の“ファイル記述子”を返します。この値は他の用途として、`fcntl` モジュールや `os.read()` やその仲間のような、ファイル記述子を必要とする低レベルのインタフェースで役に立ちます。注意: ファイル類似のオブジェクトが実際のファイルに関連付けられていない場合、このメソッドを提供すべきではありません。

**isatty()**

ファイルが `tty` (または類似の) デバイスに接続されている場合 `True` を返し、そうでない場合 `False` を返します。注意: ファイル類似のオブジェクトが実際のファイルに関連付けられていない場合、このメソッドを実装すべきではありません。

**next()**

ファイルオブジェクトはそれ自身がイテレータです。すなわち、`iter(f)` は ( $f$  が閉じられていない限り)  $f$  を返します。`for` ループ (例えば `for line in f: print line`) のようにファイルがイテレータとして使われた場合、`next()` メソッドが繰り返し呼び出されます。個のメソッドは次の入力行を返すか、または EOF に到達したときに `StopIteration` を送出します。ファイル内の各行に対する `for` ループ (非常によくある操作です) を効率的な方法で行うために、`next()` メソッドは隠蔽された先読みバッファを使います。先読みバッファを使った結果として、(`readline()` のような) 他のファイルメソッドと `next()` を組み合わせて使うとうまく動作しません。しかし、`seek()` を使ってファイル位置を絶対指定しなると、先読みバッファはフラッシュされます。

2.3 で追加された仕様です。

**read([size])**

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

<sup>11</sup> `file()` は Python 2.2 で新しく追加されました。古いバージョンの組み込み関数 `open()` は `file()` の別名です。

てください。また、非ブロック・モードでは、*size* パラメータが与えられなくても、要求されたよりも少ないデータが返される場合があることに注意してください。

**readline([size])**

ファイルから一行を読み出します。末尾の改行文字は文字列中に残されます<sup>12</sup>(ファイルが改行のない不完全な行の場合には、改行文字がないかもしれませんが)。引数 *size* が指定されていて非負の場合、(末尾の改行を含めて) 読み込む最大のバイト数です。この場合、不完全な行が返されるかもしれません。空文字列が返されるのは、直後に EOF に到達した場合 だけ です。注意: `stdio` の `fgets()` と違い、入力中にヌル文字 ('\\0') が含まれていれば、ヌル文字を含んだ文字列が返されます。

**readlines([sizehint])**

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

**xreadlines()**

個のメソッドは `iter(f)` と同じ結果を返します。2.1 で追加された仕様です。リリース 2.3 以降で撤廃された仕様です。代わりに `for line in file` を使ってください。

**seek(offset[, whence])**

`stdio` の `fseek()` と同様に、ファイルの現在位置を返します。*whence* 引数はオプションで、標準の値は 0 (絶対位置指定) です; 他に取り得る値は 1 (現在のファイル位置から相対的に seek する) および 2 (ファイルの末端から相対的に seek する) です。戻り値はありません。ファイルを追記モード(モード 'a' または 'a+') で開いた場合、書き込みを行うまでに行った `seek()` 操作はすべて元に戻されるので注意してください。ファイルが追記のみの書き込みモード('a') で開かれた場合、このメソッドは実質何も行いませんが、読み込みが可能な追記モード('a+') で開かれたファイルでは役に立ちます。ファイルをテキストモード('t') で開いた場合、`tell()` が返すオフセットのみが正しい値になります。他のオフセット値を使った場合、その振る舞いは未定義です。

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

**tell()**

`stdio` の `ftell()` と同様、ファイルの現在位置を返します。

**truncate([size])**

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

**write(str)**

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

**writelines(sequence)**

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

<sup>12</sup>改行を残す利点は、空の文字列が返ると EOF を示し、紛らわしくなくなるからです。また、ファイルの最後の行が改行で終わっているかそうでない(ありえることです!) か (例えば、ファイルを行単位で読みながらその完全なコピーを作成した場合には問題になります) を調べることができます。

ファイルはイテレータプロトコルをサポートします。各反復操作では `file.readline()` と同じ結果を返し、反復は `readline()` メソッドが空文字列を返した際に終了します。

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

#### **closed**

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

#### **encoding**

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

2.3 で追加された仕様です。

#### **mode**

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

#### **name**

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

#### **newlines**

Python をビルドするとき、`-with-universal-newlines` オプションが指定された場合 (デフォルト) この読み出し専用の属性が存在します。一般的な改行に変換する読み出しモードで開かれたファイルにおいて、この属性はファイルの読み出し中に遭遇した改行コードを追跡します。取り得る値は `'\r'`、`'\n'`、`'\r\n'`、`None` (不明または、まだ改行していない)、見つかった全ての改行文字を含むタプルのいずれかです。最後のタプルは、複数の改行慣例に遭遇したことを示します。一般的な改行文字を使う読み出しモードで開かれていないファイルの場合、この属性の値は `None` です。

#### **softspace**

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

### 2.3.9 他の組み込み型

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



## モジュール

モジュールに対する唯一の特殊な演算は属性へのアクセス: `m.name` です。ここで `m` はモジュールで、`name` は `m` のシンボルテーブル上に定義された名前にアクセスします。モジュール属性も代入することができます。(import 文は、厳密に言えば、モジュールオブジェクトに対する演算です; `import foo` は `foo` と名づけられたモジュールオブジェクトが存在することを必要とはせず、むしろ `foo` と名づけられた (外部の) モジュールの定義を必要とします。)

各モジュールの特殊なメンバは `__dict__` です。これはモジュールのシンボルテーブルを含む辞書です。この辞書を修正すると、実際にはモジュールのシンボルテーブルを変更しますが、`__dict__` 属性を直接代入することはできません (`m.__dict__['a'] = 1` と書いて `m.a` を 1 に定義することはできませんが、`m.__dict__ = {}` と書くことはできません)。

インタプリタ内に組み込まれたモジュールは、`<module 'sys' (built-in)>` のように書かれます。ファイルから読み出された場合、`<module 'os' from '/usr/local/lib/python2.3/os.pyc'>` と書かれます。

## クラスおよびクラスインスタンス

これらに関しては、Python リファレンスマニュアルの 3 章および 7 章を読んで下さい。

## 関数

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

関数オブジェクトには実際には 2 つの種: 組み込み関数とユーザ定義関数があります。両方とも同じ操作 (関数の呼び出し) をサポートしますが、実装は異なるので、オブジェクトの型も異なります。

関数の実装では、2 つの読み出し専用属性を追加しています: `f.func_code` は関数のコードオブジェクト (以下を参照) であり、`f.func_globals` は関数のグローバル名前空間として使われる辞書です (`f` が定義されているモジュールを `m` としたときの `m.__dict__` と同じです)。

関数オブジェクトはまた、例えば関数のメタ情報を関数に付属させるといったことに使える、任意の属性の取得や設定をサポートしています。それらの属性の取得や設定には、正規の属性をドット表記する方法が使われます。現在の実装では、関数の属性をサポートしているのはユーザ定義関数のみなので注意してください。組み込み関数の属性は将来サポートされるかもしれません。

関数はもう一つの特徴的な属性 `f.__dict__` (`f.func_dict` として知られています) を持っており、関数属性をサポートするために使われる名前空間を含んでいます。`__dict__` および `func_dict` は直接アクセスしたり、辞書オブジェクトを設定することができます。関数の辞書は削除することはできません。

## メソッド

メソッドは属性表記を使って呼び出される関数です。メソッドには二つの種類があります: (リストへの `append()` のような) 組み込みメソッドと、クラスインスタンスのメソッドです。組み込みメソッドはそれをサポートする型と一緒に記述されています。

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

クラスインスタンスメソッドには、メソッドがインスタンスからアクセスされるかクラスからアクセス



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

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

```
class C:
    def method(self):
        pass

c = C()
c.method.im_func.whoami = 'my name is c'
```

のように設定しなければなりません。詳しくは *Python* リファレンスマニュアルを読んで下さい。

## コードオブジェクト

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

コードオブジェクトは `exec` 文や組み込み関数 `eval()` に (ソースコード文字列の代わりに) 渡すことで、実行したり値評価したりすることができます。

詳しくは *Python* リファレンスマニュアルを読んで下さい。

## 型オブジェクト

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

型は `<type 'int'>` のように書き表されます。

## ヌルオブジェクト

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

`None` と書き表されます。

## 省略表記オブジェクト

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

`Ellipsis` と書き表されます。

## ブール値

ブール値とは二つの定数オブジェクト `False` および `True` です。これらは真偽値を表すために使われます (他の値も偽または真とみなされます) 数値処理のコンテキスト (例えば算術演算子の引数として使われた場合) では、これらはそれぞれ 0 および 1 と同様に振舞います。任意の値に対して真偽値を変換できる場合、組み込み関数 `bool()` は値をブール値にキャストするのに使われます (真値テストの節を参照してください)

これらはそれぞれ `False` および `True` と書き表されます。

## 内部オブジェクト

この情報については *Python* リファレンスマニュアル を読んで下さい。このオブジェクトではスタックフレーム、トレースバック、スライスオブジェクトを記述しています。

### 2.3.10 特殊な属性

実装では、いくつかのオブジェクト型に対して、数個の読み出し専用の特殊な属性を追加しています。それぞれ:

#### `__dict__`

オブジェクトの (書き込み可能な) 属性を保存するために使われる辞書または他のマップ型オブジェクトです。

#### `__methods__`

リリース 2.2 以降で撤廃された仕様です。オブジェクトの属性からなるリストを取得するには、組み込み関数 `dir()` を使ってください。この属性はもう利用できません。

#### `__members__`

リリース 2.2 以降で撤廃された仕様です。オブジェクトの属性からなるリストを取得するには、組み込み関数 `dir()` を使ってください。この属性はもう利用できません。

#### `__class__`

クラスインスタンスが属しているクラスです。

#### `__bases__`

クラスオブジェクトの基底クラスからなるタプルです。基底クラスを持たない場合、空のタプルになります。

## 2.4 組み込み例外

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

注意: 過去の Python のバージョンでは、文字列の例外がサポートされていました。Python 1.5 よりも新しいバージョンでは、全ての標準的な例外はクラスオブジェクトに変換され、ユーザにも同様にしよう奨励しています。文字列による例外は `PendingDeprecationWarning` を送出するようになります。将来のバージョンでは、文字列による例外のサポートは削除されます。

同じ値を持つ別々の文字列オブジェクトは異なる例外と見なされます。これはプログラマに対して、例外処理を指定する際に、文字列ではなく例外名を使わせるための変更です。組み込み例外の文字列値は全てその名前となりますが、ユーザ定義の例外やライブラリモジュールで定義される例外についてもそうす

るように要求しているわけではありません。

`try` 文の中で、`except` 節を使って特定の例外クラスについて記述した場合、その節は指定した例外クラスから導出されたクラスも扱います (指定した例外クラスを導出した元のクラスは含みません) サブクラス化の関係にない例外クラスが二つあった場合、それらに同じ名前を付けたとしても、等しくなることはありません。

以下に列挙した組み込み例外はインタプリタや組み込み関数によって生成されます。特に注記しないかぎり、これらの例外はエラーの詳しい原因を示している、“関連値 (associated value)” を持ちます。この値は文字列または複数の情報 (例えばエラーコードや、エラーコードを説明する文字列) を含むタプルです。この関連値は `raise` 文の二つ目の引数です。文字列の例外の場合、関連値自体は `except` 節 (あった場合) の二つ目の引数として与えた名前を持つ変数に記憶されます。クラス例外の場合、この値は例外クラスのインスタンスです。例外が標準のルートクラスである `Exception` から導出された場合、関連値は例外インスタンスの `args` 属性中に他の属性と同様に置かれます。

ユーザによるコードも組み込み例外を送出することができます。これは例外処理をテストしたり、インタプリタがある例外を送出する状況と“ちょうど同じような”エラー条件であることを報告させるために使うことができます。しかし、ユーザが適切でないエラーを送出するようコードするのを妨げる方法はないので注意してください。

組み込み例外クラスは新たな例外を定義するためにサブクラス化することができます; プログラマには、新しい例外を少なくとも `Exception` 基底クラスから導出するよう勧めます。例外を定義する上での詳しい情報は、*Python* チュートリアル の“ユーザ定義の例外”の項目にあります。

以下の例外クラスは他の例外クラスの基底クラスとしてのみ使われます。

#### **exception `Exception`**

例外のルートクラスです。全ての組み込み例外はこのクラスから導出されています。全てのユーザ定義例外はこのクラスから導出されるべきですが、(今のところまだ) それは強制ではありません。`str()` をこのクラス (またはほとんどの導出クラス) のインスタンスに適用すると、引数を文字列にしたが返されるか、インスタンスのコンストラクタが引数なしの場合には空の文字列が返されます。インスタンスを配列として使うと、コンストラクタに渡された引数にアクセスすることができます (古いコードとの互換性のために便利です)。引数はまた、インスタンスの `args` 属性として、タプルで得ることもできます。

#### **exception `StandardError`**

`StopIteration` および `SystemExit` 以外の、全ての組み込み例外の基底クラスです。`StandardError` 自体はルートクラス `Exception` から導出されています。

#### **exception `ArithmeticError`**

算術上の様々なエラーにおいて送出される組み込み例外: `OverflowError`、`ZeroDivisionError`、`FloatingPointError` の基底クラスです。

#### **exception `LookupError`**

マップ型または配列型に使ったキーやインデックスが無効な値の場合に送出される例外: `IndexError`、`KeyError` の基底クラスです。`sys.setdefaultencoding()` によって直接送出されることもあります。

#### **exception `EnvironmentError`**

Python システムの外部で起こっているはずの例外: `IOError`、`OSError` の基底クラスです。この型の例外が 2 つの要素をもつタプルで生成された場合、最初の要素はインスタンスの `errno` 属性で得ることができます (この値はエラー番号と見なされます)。二つめの要素は `strerror` 属性です (この値は通常、エラーに関連するメッセージです)。タプル自体は `args` 属性から得ることもできます。1.5.2 で追加された仕様です。

`EnvironmentError` 例外が 3 要素のタプルで生成された場合、最初の 2 つの要素は上と同様に得

ることができる一方、3 つ目の要素は `filename` 属性で得ることができます。しかしながら、以前のバージョンとの互換性のために、`args` 属性にはコンストラクタに渡した最初の 2 つの引数からなる 2 要素のタプルしか含みません。

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

以下の例外は実際に送出される例外です。

#### **exception AssertionError**

`assert` 文が失敗した場合に送出されます。

#### **exception AttributeError**

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

#### **exception EOFError**

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

#### **exception FloatingPointError**

浮動小数点演算が失敗した場合に送出されます。この例外はどの Python のバージョンでも常に定義されていますが、Python が `--with-fpectl` オプションをつけた状態に設定されているか、`'pyconfig.h'` ファイルにシンボル `WANT_SIGFPE_HANDLER` が定義されている場合にのみ送出されます。

#### **exception IOError**

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

このクラスは `EnvironmentError` から導出されています。この例外クラスのインスタンス属性に関する情報は上記の `EnvironmentError` に関する議論を参照してください。

#### **exception ImportError**

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

#### **exception IndexError**

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

#### **exception KeyError**

マップ型 (辞書型) オブジェクトのキーが、オブジェクトのキー集合内に見つからなかった場合に送出されます。

#### **exception KeyboardInterrupt**

ユーザが割り込みキー (通常は `Control-C` または `Delete` キーです) を押した場合に送出されます。割り込みが起きたかどうかはインタプリタの実行中に定期的に調べられます。組み込み関数 `input()` や `raw_input()` がユーザの入力を待っている間に割り込みキーを押しても、この例外が送出されます。

#### **exception MemoryError**

ある操作中にメモリが不足したが、その状況は (オブジェクトをいくつか消去することで) まだ復旧可能かもしれない場合に送出されます。例外に関連づけられた値は、どの種の (内部) 操作がメモリ

不足になっているかを示す文字列です。背後にあるメモリ管理アーキテクチャ (C の `malloc()` 関数) によっては、インタプリタが常にその状況を完璧に復旧できるとはかぎらないので注意してください; プログラムの暴走が原因の場合にも、やはり実行スタックの追跡結果を出力できるようにするために例外が送出されます。

#### **exception NameError**

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

#### **exception NotImplementedError**

この例外は `RuntimeError` から導出されています。ユーザ定義の基底クラスにおいて、そのクラスの導出クラスにおいてオーバーライドすることが必要な抽象化メソッドはこの例外を送出しなくてはなりません。1.5.2 で追加された仕様です。

#### **exception OSError**

このクラスは `EnvironmentError` から導出されており、主に `os` モジュールの `os.error` 例外で使われています。例外に関連付けられる可能性のある値については、上記の `EnvironmentError` を参照してください。1.5.2 で追加された仕様です。

#### **exception OverflowError**

算術演算の結果、表現するには大きすぎる値になった場合に送出されます。これは長整数の演算では起こりません (長整数の演算ではむしろ `MemoryError` が送出されることになるでしょう)。C では浮動小数点演算における例外処理の標準化が行われていないので、ほとんどの浮動小数点演算もチェックされていません。通常の整数では、オーバーフローを起こす全ての演算がチェックされます。例外は左シフトで、典型的なアプリケーションでは左シフトのオーバーフローでは例外を送出するよりもむしろ、オーバーフローしたビットを捨てるようにしています。

#### **exception ReferenceError**

`weakref.proxy()` によって生成された弱参照 (weak reference) プロキシを使って、ガーベジコレクションによって処理された後の参照対象オブジェクトの属性にアクセスした場合に送出されます。弱参照については `weakref` モジュールを参照してください。2.2 で追加された仕様: 以前は `weakref.ReferenceError` 例外として知られていました。

#### **exception RuntimeError**

他のカテゴリに分類できないエラーが検出された場合に送出されます。関連付けられた値は何が問題だったのかをより詳細に示す文字列です。(この例外はほとんど過去のバージョンのインタプリタにおける遺物です; この例外はもはやあまり使われることはありません)

#### **exception StopIteration**

イテレータの `next()` メソッドにより、それ以上要素がないことを知らせるために送出されます。この例外は、通常のアプリケーションではエラーとはみなされないで、`StandardError` ではなく `Exception` から導出されています。2.2 で追加された仕様です。

#### **exception SyntaxError**

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

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

#### **exception SystemError**

インタプリタが内部エラーを発見したが、その状況は全ての望みを棄てさせるほど深刻ではないように思われる場合に送出されます。関連づけられた値は (控えめな言い方で) 何がまずいのかを示す文



字列です。

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

#### **exception SystemExit**

この例外は `sys.exit()` 関数によって送出されます。この例外が処理されなかった場合、Python インタプリタは終了します; スタックのトレースバックは全く印字されません。関連付けられた値が通常の整数である場合、システム終了状態を指定しています (`exit()` 関数に渡されます); 値が `None` の場合、終了状態はゼロです; (文字列のような) 他の型の場合、そのオブジェクトの値が印字され、終了状態は 1 になります。

この例外のインスタンスは属性 `code` を持ちます。この値は終了状態またはエラーメッセージ (標準では `None` です) に設定されます。また、この例外は技術的にはエラーではないため、`StandardError` からではなく、`Exception` から導出されています。

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

#### **exception TypeError**

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

#### **exception UnboundLocalError**

関数やメソッド内のローカルな変数に対して参照を行ったが、その変数には値がバインドされていない場合に送出されます。`NameError` のサブクラスです。2.0 で追加された仕様です。

#### **exception UnicodeError**

Unicode に関するエンコードまたはデコードのエラーが発生した際に送出されます。`ValueError` のサブクラスです。2.0 で追加された仕様です。

#### **exception UnicodeEncodeError**

Unicode 関連のエラーがエンコード中に発生した際に送出されます。`UnicodeError` のサブクラスです。2.3 で追加された仕様です。

#### **exception UnicodeDecodeError**

Unicode 関連のエラーがデコード中に発生した際に送出されます。`UnicodeError` のサブクラスです。2.3 で追加された仕様です。

#### **exception UnicodeTranslateError**

Unicode 関連のエラーがコード翻訳に発生した際に送出されます。`UnicodeError` のサブクラスです。2.3 で追加された仕様です。

#### **exception ValueError**

組み込み演算や関数が、正しい型だが適切でない値を受け取った場合、および `IndexError` のように、より詳細な説明のできない状況で送出されます。

#### **exception WindowsError**

Windows 特有のエラーか、エラー番号が `errno` 値に対応しない場合に送出されます。`errno` および `strerror` 値は Windows プラットフォーム API の関数、`GetLastError()` と `FormatMessage()` の戻り値から生成されます。`OSError` のサブクラスです。2.0 で追加された仕様です。

#### **exception ZeroDivisionError**



除算またモジュロ演算における二つ目の引数がゼロであった場合に送出されます。関連付けられている値は文字列で、その演算における被演算子の型を示します。

以下の例外は警告カテゴリとして使われます; 詳細については `warnings` モジュールを参照してください。

**exception Warning**

警告カテゴリの基底クラスです。

**exception UserWarning**

ユーザコードによって生成される警告の基底クラスです。

**exception DeprecationWarning**

廃用された機能に対する警告の基底クラスです。

**exception PendingDeprecationWarning**

将来廃用されることになっている機能に対する警告の基底クラスです。

**exception SyntaxWarning**

曖昧な構文に対する警告の基底クラスです。

**exception RuntimeWarning**

あいまいなランタイム挙動に対する警告の基底クラスです。

**exception FutureWarning**

将来意味構成が変わることになっている文の構成に対する警告の基底クラスです。

組み込み例外のクラス階層は以下のようになっています:

```

Exception
+-- SystemExit
+-- StopIteration
+-- StandardError
|   +-- KeyboardInterrupt
|   +-- ImportError
|   +-- EnvironmentError
|       +-- IOError
|       +-- OSError
|       +-- WindowsError
+-- EOFError
+-- RuntimeError
|   +-- NotImplementedError
+-- NameError
|   +-- UnboundLocalError
+-- AttributeError
+-- SyntaxError
|   +-- IndentationError
|   +-- TabError
+-- TypeError
+-- AssertionError
+-- LookupError
|   +-- IndexError
|   +-- KeyError
+-- ArithmeticError
|   +-- OverflowError
|   +-- ZeroDivisionError
|   +-- FloatingPointError
+-- ValueError
|   +-- UnicodeError
|       +-- UnicodeEncodeError
|       +-- UnicodeDecodeError
|       +-- UnicodeTranslateError
+-- ReferenceError
+-- SystemError
+-- MemoryError
+---Warning
+-- UserWarning
+-- DeprecationWarning
+-- PendingDeprecationWarning
+-- SyntaxWarning
+-- OverflowWarning
+-- RuntimeWarning
+-- FutureWarning

```

## 2.5 組み込み定数

組み込み空間には少しだけ定数があります。以下にそれらの定数を示します:

### False

bool 型における、偽を表す値です。2.3 で追加された仕様です。

### True

bool 型における、真を表す値です。2.3 で追加された仕様です。

### None

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

**NotImplemented**

“特殊な比較 (rich comparison)” を行う特殊メソッド (`__eq__()`、`__lt__()`、およびその仲間) に対して、他の型に対しては比較が実装されていないことを示すために返される値です。

**Ellipsis**

拡張スライス文と同時に用いられる特殊な値です。



# Python ランタイム サービス

この章では、Python インタプリタや Python 環境に深く関連する各種の機能を解説します。以下に一覧を示します:

<code>sys</code>	システムパラメータと関数へのアクセス
<code>gc</code>	循環検出ガベージコレクタのインターフェース。
<code>weakref</code>	弱参照と弱辞書のサポート。
<code>fpectl</code>	浮動小数点例外処理の制御。
<code>atexit</code>	後始末関数の登録と実行。
<code>types</code>	組み込み型の名前
<code>UserDict</code>	辞書オブジェクトのためのクラスラッパー。
<code>UserList</code>	リストオブジェクトのためのクラスラッパー。
<code>UserString</code>	文字列オブジェクトのためのクラスラッパー。
<code>operator</code>	組み込み関数形式になっている全ての Python の標準演算子。
<code>inspect</code>	使用中のオブジェクトから、情報とソースコードを取得する。
<code>traceback</code>	スタックトレースの表示や取り出し。
<code>linecache</code>	このモジュールによりテキストファイルの各行にランダムアクセスできます。
<code>pickle</code>	Python オブジェクトからバイトストリームへの変換、およびその逆。
<code>cPickle</code>	<code>pickle</code> の高速バージョンですが、サブクラスはできません。
<code>copy_reg</code>	<code>pickle</code> サポート関数を登録する。
<code>shelve</code>	Python オブジェクトの永続化。
<code>copy</code>	浅いコピーおよび深いコピー操作。
<code>marshal</code>	Python オブジェクトをバイト列に変換したり、その逆を (異なる拘束条件下で) 行います。
<code>warnings</code>	警告メッセージを送出したり、その処理方法を制御したりします。
<code>imp</code>	<code>import</code> 文の実装へアクセスする。
<code>pkgutil</code>	パッケージの拡張をサポートするユーティリティです。
<code>code</code>	対話的 Python インタプリタのための基底クラス。
<code>codeop</code>	(完全ではないかもしれない)Python コードをコンパイルする。
<code>pprint</code>	Data pretty printer.
<code>repr</code>	大きさに制限のある別の <code>repr()</code> の実装。
<code>new</code>	ランタイム実装オブジェクトの作成のインターフェイス。
<code>site</code>	サイト固有のモジュールを参照する標準の方法。
<code>user</code>	ユーザー設定を参照するための標準的な方法を提供するモジュール
<code>__builtin__</code>	組み込み関数一式
<code>__main__</code>	トップレベルスクリプトが実行される環境。
<code>__future__</code>	Future ステートメントの定義

### 3.1 sys — システムパラメータと関数

このモジュールでは、インタプリタで使用・管理している変数や、インタプリタの動作に深く関連する関数を定義しています。このモジュールは常に利用可能です。

#### **argv**

Python スクリプトに渡されたコマンドライン引数のリスト。argv[0] はスクリプトの名前となりますが、フルパス名かどうかは、オペレーティングシステムによって異なります。コマンドライン引数に **-c** を付けて Python を起動した場合、argv[0] は文字列 **'-c'** となります。引数なしで Python を起動した場合、argv は長さ 0 のリストになります。

#### **byteorder**

プラットフォームのバイト順を示します。ビッグエンディアン (最上位バイトが先頭) のプラットフォームでは **'big'**、リトルエンディアン (最下位バイトが先頭) では **'little'** となります。2.0 で追加された仕様です。

#### **builtin\_module\_names**

コンパイル時に Python インタプリタに組み込まれた、全てのモジュール名のタプル (この情報は、他の手段では取得することができません。modules.keys() は、インポートされたモジュールのみのリストを返します。)

#### **copyright**

Python インタプリタの著作権を表示する文字列。

#### **dllhandle**

Python DLL のハンドルを示す整数。利用可能: Windows

#### **displayhook(value)**

value が None 以外の場合、value を sys.stdout に出力して \_\_builtin\_\_.\_ に保存します。

sys.displayhook は、Python の対話セッションで入力された式が評価されたときに呼び出されます。対話セッションの出力をカスタマイズする場合、sys.displayhook に引数の数が一つの関数を指定します。

#### **excepthook(type, value, traceback)**

指定したトレースバックと例外を sys.stderr に出力します。

例外が発生し、その例外が捕捉されない場合、インタプリタは例外クラス・例外インスタンス・トレースバックオブジェクトを引数として sys.excepthook を呼び出します。対話セッション中に発生した場合はプロンプトに戻る直前に呼び出され、Python プログラムの実行中に発生した場合はプログラムの終了直前に呼び出されます。このトップレベルでの例外情報出力処理をカスタマイズする場合、sys.excepthook に引数の数が三つの関数を指定します。

#### **\_\_displayhook\_\_**

#### **\_\_excepthook\_\_**

それぞれ、起動時の displayhook と excepthook の値を保存しています。この値は、displayhook と excepthook に不正なオブジェクトが指定された場合に、元の値に復旧するために使用します。

#### **exc\_info()**

この関数は、現在処理中の例外を示す三つの値のタプルを返します。この値は、現在のスレッド・現在のスタックフレームのものです。現在のスタックフレームが例外処理中でない場合、例外処理中のスタックフレームが見つかるまで次々とその呼び出し元スタックフレームを調べます。ここで、“例外処理中”とは“except 節を実行中、または実行した”フレームを指します。どのスタックフレームでも、最後に処理した例外の情報のみを参照することができます。



スタック上で例外が発生していない場合、三つの `None` のタプルを返します。例外が発生している場合、`(type, value, traceback)` を返します。`type` は、処理中の例外の型を示します (クラスオブジェクト)。`value` は、例外パラメータ (例外に関連する値または `raise` の第二引数。`type` がクラスオブジェクトの場合は常にクラスインスタンス) です。`traceback` は、トレースバックオブジェクトで、例外が発生した時点でのコールスタックをカプセル化したオブジェクトです (リファレンスマニュアル参照)。

`exc_clear()` が呼び出されると、現在のスレッドで他の例外が発生するか、又は別の例外を処理中のフレームに実行スタックが復帰するまで、`exc_info()` は三つの `None` を返します。

警告: 例外処理中に戻り値の `traceback` をローカル変数に代入すると循環参照が発生し、関数内のローカル変数やトレースバックが参照している全てのオブジェクトは解放されなくなります。特にトレースバック情報が必要ではなければ `exc_type, value = sys.exc_info()[:2]` のように例外型と例外オブジェクトのみを取得するようにして下さい。もしトレースバックが必要な場合には、処理終了後に `delete` して下さい。この `delete` は、`try ... finally ...` で行うと良いでしょう。注意: Python 2.2 以降では、ガベージコレクションが有効であればこのような到達不能オブジェクトは自動的に削除されます。しかし、循環参照を作らないようにしたほうが効率的です。

#### `exc_clear()`

この関数は、現在のスレッドで処理中、又は最後に発生した例外の情報を全てクリアします。この関数を呼び出すと、現在のスレッドで他の例外が発生するか、又は別の例外を処理中のフレームに実行スタックが復帰するまで、`exc_info()` は三つの `None` を返します。

この関数が必要となることは滅多にありません。ロギングやエラー処理などで最後に発生したエラーの報告を行う場合などに使用します。また、リソースを解放してオブジェクトの終了処理を起動するために使用することもできますが、オブジェクトが実際にされるかどうかは保障の限りではありません。2.3 で追加された仕様です。

#### `exc_type`

#### `exc_value`

#### `exc_traceback`

リリース 1.5 以降で撤廃された仕様です。 `exc_info()` を使用してください

これらの変数はグローバル変数なのでスレッド毎の情報を示すことができません。この為、マルチスレッドなプログラムでは安全に参照することはできません。例外処理中でない場合、`exc_type` の値は `None` となり、`exc_value` と `exc_traceback` は未定義となります。

#### `exec_prefix`

Python のプラットフォーム依存なファイルがインストールされているディレクトリ名 (サイト固有)。デフォルトでは、この値は `'/usr/local'` ですが、ビルド時に `configure` の `--exec-prefix` 引数で指定することができます。全ての設定ファイル (`'pyconfig.h'` など) は `exec_prefix + '/lib/pythonversion/config'` に、共有ライブラリは `exec_prefix + '/lib/pythonversion/lib-dynload'` にインストールされます (但し `version` は `version[:3]`)。

#### `executable`

Python インタープリタの実行ファイルの名前を示す文字列。このような名前が意味を持つシステムでは利用可能。

#### `exit([arg])`

Python を終了します。 `exit()` は `SystemExit` を送出するので、`try` ステートメントの `finally` 節に終了処理を記述したり、上位レベルで例外を捕捉して `exit` 処理を中断したりすることができます。オプション引数 `arg` には、終了ステータスとして整数 (デフォルトは 0) または整数以外の型のオブジェクトを指定することができます。整数を指定した場合、シェル等は 0 は “正常終了”、0 以外の整数を “異常終了” として扱います。多くのシステムでは、有効な終了ステータスは 0-127 で、こ

れ以外の値を返した場合の動作は未定義です。システムによっては特定の終了コードに個別の意味を持たせている場合がありますが、このような定義は僅かしかありません。UNIX プログラムでは文法エラーの場合には 2 を、それ以外のエラーならば 1 を返します。*arg* に *None* を指定した場合は、数値の 0 を指定した場合と同じです。それ以外のオブジェクトを指定すると、そのオブジェクトが `sys.stderr` に出力され、終了コードをして 1 を返します。エラー発生時には `sys.exit("エラーメッセージ")` と書くと、簡単にプログラムを終了することができます。

#### `exitfunc`

この値はモジュールに存在しませんが、ユーザプログラムでプログラム終了時に呼び出される終了処理関数として、引数の数が 0 の関数を設定することができます。この関数は、インタプリタ終了時に呼び出されます。`exitfunc` に指定することができる終了処理関数は一つだけですので、複数のクリーンアップ処理が必要な場合は `atexit` モジュールを使用してください。注意: 終了処理関数は、以下の場合には呼び出されません。シグナルで `kill` された場合・Python 内部で fatal エラーが発生した場合・`os._exit()` が呼び出された場合

#### `getcheckinterval()`

インタプリタの“チェックインターバル (check interval)”を返します; `setcheckinterval()` を参照してください。2.3 で追加された仕様です。

#### `getdefaultencoding()`

現在の Unicode 処理のデフォルトエンコーディング名を返します。2.0 で追加された仕様です。

#### `getdlopenflags()`

`dlopen()` で指定されるフラグを返します。このフラグは `dl` と `DLFCN` で定義されています。

利用可能: UNIX. 2.2 で追加された仕様です。

#### `getfilesystemencoding()`

Unicode ファイル名をシステムのファイル名に変換する際に使用するエンコード名を返します。システムのデフォルトエンコーディングを使用する場合には *None* を返します。

- Windows 9x では、エンコーディングは“mbcs”となります。
- OS X では、エンコーディングは“utf-8”となります。
- Unix では、エンコーディングはユーザの `nl_langinfo(CODESET)` の設定となります。`nl_langinfo(CODESET)` が失敗すると *None* を返します。
- Windows NT+では、Unicode をファイル名として使用することができるので変換の必要がありません。

2.3 で追加された仕様です。

#### `getrefcount(object)`

*object* の参照数を返します。*object* は(一時的に) `getrefcount()` から参照されるため、参照数は予想される数よりも 1 多くなります。

#### `getrecursionlimit()`

現在の最大再帰数を返します。最大再帰数は、Python インタプリタスタックの最大の深さです。この制限は Python プログラムが無限に再帰し、C スタックがオーバーフローしてクラッシュすることを防止するために設けられています。この値は `setrecursionlimit()` で指定することができます。

#### `_getframe([depth])`

コールスタックからフレームオブジェクトを取得します。オプション引数 *depth* を指定すると、スタックのトップから *depth* だけ下のフレームオブジェクトを取得します。*depth* がコールスタックよりも深ければ、`ValueError` が発生します。*depth* のデフォルト値は 0 で、この場合はコールスタックのトップのフレームを返します。

この関数は、内部的な、特殊な用途にのみ利用することができます。

#### `getwindowsversion()`

実行中の Windows のバージョンを示す、以下の値のタプルを返します：*major, minor, build, platform, text*。*text* は文字列、それ以外の値は整数です。

*platform* は、以下の値となります：

- 0 (VER\_PLATFORM\_WIN32s)Win32s on Windows 3.1.
- 1 (VER\_PLATFORM\_WIN32\_WINDOWS)Windows 95/98/ME
- 2 (VER\_PLATFORM\_WIN32\_NT)Windows NT/2000/XP
- 3 (VER\_PLATFORM\_WIN32\_CE)Windows CE.

この関数は、Win32 `GetVersionEx()` 関数を呼び出します。詳細はマイクロソフトのドキュメントを参照してください。

利用可能: Windows. 2.3 で追加された仕様です。

#### `hexversion`

整数にエンコードされたバージョン番号。この値は新バージョン (正規リリース以外であっても) ごとにならず増加します。例えば、Python 1.5.2 以降でのみ動作するプログラムでは、以下のようなチェックを行います。

```
if sys.hexversion >= 0x010502F0:
    # use some advanced feature
    ...
else:
    # use an alternative implementation or warn the user
    ...
```

‘hexversion’ は `hex()` で 16 進数に変換しなければ値の意味がわかりません。より読みやすいバージョン番号が必要な場合には `version_info` を使用してください。1.5.2 で追加された仕様です。

#### `last_type`

#### `last_value`

#### `last_traceback`

通常は定義されておらず、捕捉されない例外が発生してインタプリタがエラーメッセージとトレースバックを出力した場合にのみ設定されます。この値は、対話セッション中にエラーが発生したとき、デバッグモジュールをロード (例: ‘import pdb; pdb.pm()’) など。詳細は [9](#) を参照) して発生したエラーを調査する場合に利用します。デバッガをロードすると、プログラムを再実行せずに情報を取得することができます。

変数の意味は、上の `exc_info()` の戻り値と同じです。対話セッションを実行するスレッドは常に一つだけなので、`exc_type` のようにスレッドに関する問題は発生しません。

#### `maxint`

Python の整数型でサポートされる、最大の整数。この値は最低でも  $2^{31}-1$  です。最大の負数は  $-\text{maxint}-1$  となります。正負の最大数が非対称ですが、これは 2 の補数計算を行うためです。

#### `maxunicode`

Unicode 文字の最大のコードポイントを示す整数。この値は、オプション設定で Unicode 文字の保存形式として USC-2 と UCS-4 のいずれを指定したかによって異なります。

#### `modules`

ロード済みモジュールのモジュール名とモジュールオブジェクトの辞書。強制的にモジュールを再読み込みする場合などに使用します。この辞書からモジュールを削除するのは、`reload()` の呼び出しと等価ではありません。

## path

モジュールを検索するパスを示す文字列のリスト。PYTHONPATH 環境変数と、インストール時に指定したデフォルトパスで初期化されます。

開始時に初期化された後、リストの先頭 (path[0]) には Python インタープリタを起動するために指定したスクリプトのディレクトリが挿入されます。スクリプトのディレクトリがない (インタープリタで対話セッションで起動された時や、スクリプトを標準入力から読み込む場合など) 場合、path[0] には空文字列となり、Python はカレントディレクトリからモジュールの検索を開始します。スクリプトディレクトリは、PYTHONPATH で指定したディレクトリの前に挿入されますので注意が必要です。必要に応じて、プログラム内で自由に変更することができます。

2.3 で変更された仕様: Unicode 文字列が無視されなくなりました。

## platform

プラットフォームを識別する文字列 (例: 'sunos5', 'linux1' 等)。path にプラットフォーム別のサブディレクトリを追加する場合などに利用します。

## prefix

サイト固有の、プラットフォームに依存しないファイルを格納するディレクトリを示す文字列。デフォルトでは '/usr/local' になります。この値はビルド時に **configure** スクリプトの **--prefix** 引数で指定する事ができます。Python ライブラリの主要部分は prefix + '/lib/pythonversion' にインストールされ、プラットフォーム非依存なヘッダファイル ('pyconfig.h' 以外) は prefix + '/include/pythonversion' に格納されます (但し version は version[:3])。

## ps1

## ps2

インタープリタの一次プロンプト、二次プロンプトを指定する文字列。対話モードで実行中のみ定義され、初期値は '>>> ' と '... ' です。文字列以外のオブジェクトを指定した場合、インタープリタが対話コマンドを読み込むごとにオブジェクトの str() を評価します。この機能は、動的に変化するプロンプトを実装する場合に利用します。

## setcheckinterval(interval)

インタープリタの“チェック間隔”を示す整数値を指定します。この値はスレッドスイッチやシグナルハンドラのチェックを行う周期を決定します。デフォルト値は 100 で、この場合 100 の仮想命令を実行するとチェックを行います。この値を大きくすればスレッドを利用するプログラムのパフォーマンスが向上します。この値が ≤ 0 以下の場合、全ての仮想命令を実行するたびにチェックを行い、レスポンス速度と最大になりますがオーバーヘッドもまた最大となります。

## setdefaultencoding(name)

現在の Unicode 処理のデフォルトエンコーディング名を設定します。name に一致するエンコーディングが見つからない場合、LookupError が発生します。この関数は、site モジュールの実装が、sitecustomize モジュールから使用するためだけに定義されています。site から呼び出された後、この関数は sys から削除されます。2.0 で追加された仕様です。

## setdlopenflags(n)

インタープリタが拡張モジュールをロードする時、dlopen() で使用するフラグを設定します。sys.setdlopenflags(0) とすれば、モジュールインポート時にシンボルの遅延解決を行う事ができます。シンボルを拡張モジュール間で共有する場合には、sys.setdlopenflags(dl.RTLD\_NOW | dl.RTLD\_GLOBAL) と指定します。フラグの定義名は dl か DLFCN で定義されています。DLFCN が存在しない場合、h2py スクリプトを使って '/usr/include/dlfcn.h' から生成することができます。

利用可能: UNIX. 2.2 で追加された仕様です。

## setprofile(profilefunc)

システムのプロファイル関数を登録します。プロファイル関数は、Python のソースコードプロファイ

ルを行う関数で、Python で記述することができます。詳細は [10](#)を参照してください。プロファイル関数はトレース関数 (`settrace()` 参照) と似ていますが、ソース行が実行されるごとに呼び出されるのではなく、関数の呼出しと復帰時のみ呼び出されます (例外が発生している場合でも、復帰時のイベントは発生します)。プロファイル関数はスレッド毎に設定することができますが、プロファイルはスレッド間のコンテキスト切り替えを検出することはできません。従って、マルチスレッド環境でのプロファイルはあまり意味がありません。`setprofile` は常に `None` を返します。

#### `setrecursionlimit(limit)`

Python インタプリタの、スタックの最大の深さを *limit* に設定します。この制限は Python プログラムが無限に再帰し、C スタックがオーバーフローしてクラッシュすることを防止するために設けられています。

*limit* の最大値はプラットフォームによって異なります。深い再帰処理が必要な場合にはプラットフォームがサポートしている範囲内でより大きな値を指定することができますが、この値が大きすぎればクラッシュするので注意が必要です。

#### `settrace(tracefunc)`

システムのトレース関数を登録します。トレース関数は Python のソースデバッガを実装するために使用することができます。[9.2](#)の “How It Works,” を参照してください。トレース関数はスレッド毎に設定することができますので、デバッグを行う全てのスレッドで `settrace()` を呼び出し、トレース関数を登録してください。

#### `stdin`

#### `stdout`

#### `stderr`

インタプリタの標準入力・標準出力・標準エラー出力に対応するファイルオブジェクト。`stdin` はスクリプトの読み込みを除く全ての入力処理で使用され、`input()` や `raw_input()` も `stdin` から読み込みます。`stdout` は、`print` や式の評価結果、`input()`・`raw_input()` のプロンプトの出力先となります。インタプリタのプロンプトは (ほとんど) `stderr` に出力されます。`stdout` と `stderr` は必ずしも組み込みのファイルオブジェクトである必要はなく、`write()` メソッドを持つオブジェクトであれば使用することができます。`stdout` と `stderr` を別のオブジェクトに置き換えても、`os.popen()`・`os.system()`・`os` の `exec*()` などから起動されたプロセスが使用する標準 I/O ストリームは変更されません。

#### `__stdin__`

#### `__stdout__`

#### `__stderr__`

それぞれ起動時の `stdin`・`stderr`・`stdout` の値を保存します。終了処理時や、不正なオブジェクトが指定された場合に元の値に復旧するために使用します。

#### `tracebacklimit`

捕捉されない例外が発生した時、出力されるトレースバック情報の最大レベル数を指定する整数値 (デフォルト値は 1000)。0 以下の値が設定された場合、トレースバック情報は出力されず例外型と例外値のみが出力されます。

#### `version`

Python インタプリタのバージョンとビルド番号・使用コンパイラなどの情報を示す文字列で、`バージョン(#ビルド番号, ビルド日付, ビルド時間)[コンパイラ]` となります。先頭の三文字は、バージョンごとのインストール先ディレクトリ内を識別するために使用されます。例:



```
>>> import sys
>>> sys.version
'1.5.2 (#0 Apr 13 1999, 10:51:12) [MSC 32 bit (Intel)]'
```

#### **api\_version**

使用中のインタプリタの C API バージョン。Python と拡張モジュール間の不整合をデバッグする場合などに利用できます。2.3 で追加された仕様です。

#### **version\_info**

バージョン番号を示す 5 つの値のタプル: *major*, *minor*, *micro*, *releaselevel*, *serial releaselevel* 以外は全て整数です。 *releaselevel* の値は、'alpha', 'beta', 'candidate', or 'final' の何れかです。Python 2.0 の *version\_info* は、(2, 0, 0, 'final', 0) となります。2.0 で追加された仕様です。

#### **warnoptions**

この値は、warnings framework 内部のみ使用され、変更することはできません。詳細は warnings を参照してください。

#### **winver**

Windows プラットフォームで、レジストリのキーとなるバージョン番号。Python DLL の文字列リソース 1000 に設定されています。通常、この値は *version* の先頭三文字となります。この値は参照専用で、別の値を設定しても Python が使用するレジストリキーを変更することはできません。利用可能: Windows.

参考資料:

site モジュール (3.28 節):

This describes how to use .pth files to extend `sys.path`.

## 3.2 gc — ガベージコレクタ インターフェース

gc モジュールは、インタプリタのビルドオプションで循環ガベージコレクタを有効にした場合のみ使用することができます (デフォルトで有効)。もし無効になっている場合にこのモジュールをインポートすると、`ImportError` が発生します。

このモジュールは、循環ガベージコレクタの無効化・検出頻度の調整・デバッグオプションの設定などを行うインターフェースを提供します。また、検出した到達不能オブジェクトのうち、解放する事ができないオブジェクトを参照する事もできます。循環ガベージコレクタは Python の参照カウントを補うためのものですので、もしプログラム中で循環参照が発生しない事が明らかな場合には検出をする必要はありません。自動検出は、`gc.disable()` で停止する事ができます。メモリリークをデバッグするときには、`gc.set_debug(gc.DEBUG_LEAK)` とします。

gc モジュールは、以下の関数を提供しています。

#### **enable()**

自動ガベージコレクションを有効にします。

#### **disable()**

自動ガベージコレクションを無効にします。

#### **isenabled()**

自動ガベージコレクションが有効なら真を返します。

#### **collect()**

全ての検出を行います。全ての世代を検査し、検出した到達不可オブジェクトの数を返します。



`set_debug(flags)`

ガベージコレクションのデバッグフラグを設定します。デバッグ情報は `sys.stderr` に出力されます。デバッグフラグは、下の値の組み合わせを指定する事ができます。

`get_debug()`

現在のデバッグフラグを返します。

`get_objects()`

現在、追跡しているオブジェクトのリストを返します。このリストには、戻り値のリスト自身は含まれていません。2.2 で追加された仕様です。

`set_threshold(threshold0[, threshold1[, threshold2]])`

ガベージコレクションの閾値（検出頻度）を指定します。`threshold0` を 0 にすると、検出は行われません。

GC は、オブジェクトを、走査された回数に従って 3 世代に分類します。新しいオブジェクトは最も若い（0 世代）に分類されます。もし、そのオブジェクトがガベージコレクションで削除されなければ、次に古い世代に分類されます。もっとも古い世代は 2 世代で、この世代に属するオブジェクトは他の世代に移動しません。ガベージコレクタは、最後に検出を行ってから生成・削除したオブジェクトの数をカウントしており、この数によって検出を開始します。オブジェクトの生成数 - 削除数が `threshold0` より大きくなると、検出を開始します。最初は 0 世代のオブジェクトのみが検査されます。0 世代の検査が `threshold1` 回実行されると、1 世代のオブジェクトの検査を行います。同様に、1 世代が `threshold2` 回検査されると、2 世代の検査を行います。

`get_threshold()`

現在の検出閾値を、(`threshold0`, `threshold1`, `threshold2`) のタプルで返します。

`get_referrers(*objs)`

`objs` で指定したオブジェクトのいずれかを参照しているオブジェクトのリストを返します。この関数では、ガベージコレクションをサポートしているコンテナのみを返します。他のオブジェクトを参照していても、ガベージコレクションをサポートしていない拡張型は含まれません。

尚、戻り値のリストには、すでに参照されなくなっているが、循環参照の一部でまだガベージコレクションで回収されていないオブジェクトも含まれるので注意が必要です。有効なオブジェクトのみを取得する場合、`get_referrers()` の前に `collect()` を呼び出してください。2.2 で追加された仕様です。

`get_referents(*objs)`

引数で指定したオブジェクトのいずれかから参照されている、全てのオブジェクトのリストを返します。参照先のオブジェクトは、引数で指定したオブジェクトの C レベルメソッド `tp_traverse` で取得し、全てのオブジェクトが直接到達可能な全てのオブジェクトを返すわけではありません。`tp_traverse` はガベージコレクションをサポートするオブジェクトのみが実装しており、ここで取得できるオブジェクトは循環参照の一部となる可能性のあるオブジェクトのみです。従って、例えば整数オブジェクトが直接到達可能であっても、このオブジェクトは戻り値には含まれません。2.3 で追加された仕様です。

以下の変数は読み込み専用です。（変更することはできますが、再バインドする事はできません。）

`garbage`

到達不能であることが検出されたが、解放する事ができないオブジェクトのリスト（回収不能オブジェクト）。デフォルトでは、`__del__()` メソッドを持つオブジェクトのみが格納されます。<sup>1</sup>

`__del__()` メソッドを持つオブジェクトが循環参照に含まれている場合、その循環参照全体と、循環参照からのみ到達する事ができるオブジェクトは回収不能となります。このような場合には、Python

<sup>1</sup>Python 2.2 より前のバージョンでは、`__del__()` メソッドを持つオブジェクトだけでなく、全ての到達不能オブジェクトが格納されていた。）

は安全に `__del__()` を呼び出す順番を決定する事ができないため、自動的に解放することはできません。もし安全な解放順序がわかるのであれば、`garbage` リストを参照して循環参照を破壊する事ができます。循環参照を破壊した後でも、そのオブジェクトは `garbage` リストから参照されているため、解放されません。解放するためには、循環参照を破壊した後、`del gc.garbage[:]` のように `garbage` からオブジェクトを削除する必要があります。一般的には `__del__()` を持つオブジェクトが循環参照の一部とはならないように配慮し、`garbage` はそのような循環参照が発生していない事を確認するために利用する方が良いでしょう。

`DEBUG_SAVEALL` が設定されている場合、全ての到達不能オブジェクトは解放されずにこのリストに格納されます。

以下は `set_debug()` に指定することのできる定数です。

#### `DEBUG_STATS`

検出中に統計情報を出力します。この情報は、検出頻度を最適化する際に有益です。

#### `DEBUG_COLLECTABLE`

見つかった回収可能オブジェクトの情報を出力します。

#### `DEBUG_UNCOLLECTABLE`

見つかった回収不能オブジェクト（到達不能だが、ガベージコレクションで解放する事ができないオブジェクト）の情報を出力します。回収不能オブジェクトは、`garbage` リストに追加されます。

#### `DEBUG_INSTANCES`

`DEBUG_COLLECTABLE` か `DEBUG_UNCOLLECTABLE` が設定されている場合、見つかったインスタンスオブジェクトの情報を出力します。

#### `DEBUG_OBJECTS`

`DEBUG_COLLECTABLE` か `DEBUG_UNCOLLECTABLE` が設定されている場合、見つかったインスタンスオブジェクト以外のオブジェクトの情報を出力します。

#### `DEBUG_SAVEALL`

設定されている場合、全ての到達不能オブジェクトは解放されずに `garbage` に追加されます。これはプログラムのメモリリークをデバッグするときに便利です。

#### `DEBUG_LEAK`

プログラムのメモリリークをデバッグするときに指定します。（`DEBUG_COLLECTABLE` | `DEBUG_UNCOLLECTABLE` | `DEBUG_INSTANCES` | `DEBUG_OBJECTS` | `DEBUG_SAVEALL` と同じ。）

## 3.3 weakref — 弱参照

2.1 で追加された仕様です。

`weakref` モジュールは、Python プログラマがオブジェクトへの弱参照を作成できるようにします。

以下では、用語リファレント (*referent*) は弱参照が参照するオブジェクトを意味します。

オブジェクトに対する弱参照は、そのオブジェクトを生かしておくのに十分な条件にはなりません: あるリファレントに対する参照が弱参照しか残っていない場合、ガベージコレクション機構は自由にリファレントを破壊し、そのメモリを別の用途に再利用できます。弱参照の主な用途は、巨大なオブジェクトを保持するキャッシュやマップ型の実装において、キャッシュやマップ型にあるという理由だけオブジェクトを存続させたくない場合です。例えば、巨大なバイナリ画像のオブジェクトがたくさんあり、それぞれに名前を関連付けたいとします。Python の辞書型を使って名前を画像に対応付けたり画像を名前に対応付けたりすると、画像オブジェクトは辞書内のキーや値に使われているため存続しつづけることになります。`weakref` モジュールが提供している `WeakKeyDictionary` や `WeakValueDictionary` クラスはその

代用で、対応付けを構築するのに弱参照を使い、キャッシュやマップ型に存在するという理由だけでオブジェクトを存続させないようにします。例えば、もしある画像オブジェクトが `WeakValueDictionary` の値になっていた場合、最後に残った画像オブジェクトへの参照を弱参照マップ型が保持していれば、ガーベジコレクションはこのオブジェクトを再利用でき、画像オブジェクトに対する弱参照内の対応付けはそのまま削除されます。

`WeakKeyDictionary` や `WeakValueDictionary` は弱参照を使って実装されていて、キーや値がガーベジコレクションによって回収されたことを弱参照辞書に知らせるような弱参照オブジェクトのコールバック関数を設定しています。

ほとんどのプログラムが、いずれかの弱参照辞書型を使うだけで必要を満たせるはずです — 自作の弱参照辞書を直接作成する必要は普通はありません。とはいえ、弱参照辞書の実装に使われている低水準の機構は、高度な利用を行う際に恩恵をうけられるような `weakref` モジュールで公開されています。

すべてのオブジェクトを弱参照できるわけではありません。弱参照できるオブジェクトは、クラスインスタンス、(C ではなく) Python で書かれた関数、(束縛および非束縛の両方の) メソッドです。弱参照をサポートするために拡張型を簡単に作れます。詳細については、[3.3.3節 “拡張型における弱参照”](#) を読んでください。

**`ref(object[, callback])`**

`object` への弱参照を返します。リファレントがまだ生きているならば、元のオブジェクトは参照オブジェクトの呼び出しで取り出せず。リファレントがもはや生きていないならば、参照オブジェクトを呼び出したときに `None` を返します。`callback` を与えた場合、オブジェクトをまさに後始末処理しようとするときに呼び出します。このとき弱参照オブジェクトは `callback` の唯一のパラメータとして渡されます。リファレントはもはや利用できません。

同じオブジェクトに対してたくさんの弱参照を作れます。それぞれの弱参照に対して登録されたコールバックは、もっとも新しく登録されたコールバックからもっとも古いものへと呼び出されます。

コールバックが発生させた例外は標準エラー出力に書き込まれますが、伝搬させられません。それらはオブジェクトの `__del__()` メソッドが発生させる例外とまったく同様の方法で処理されます。

`object` がハッシュ可能ならば、弱参照はハッシュ可能です。それらは `object` が削除された後でもそれらのハッシュ値を保持します。`object` が削除されてから初めて `hash()` が呼び出された場合に、その呼び出しは `TypeError` を発生させます。

弱参照は等価性のテストをサポートしていますが、順序をサポートしていません。参照がまだ生きているならば、`callback` に関係なく二つの参照はそれらのリファレントと同じ等価関係を持ちます。リファレントのどちらか一方が削除された場合、参照オブジェクトが同じオブジェクトである場合に限り、その参照は等価です。

**`proxy(object[, callback])`**

弱参照を使う `object` へのプロキシを返します。弱参照オブジェクトとともに用いられる明示的な参照外しを要求する代わりに、これはほとんどのコンテキストにおけるプロキシの利用をサポートします。`object` が呼び出し可能かどうかに依存して、返されるオブジェクトは `ProxyType` または `CallableProxyType` のどちらか一方の型を持ちます。プロキシオブジェクトはリファレントに関係なくハッシュ可能ではありません。これによって、それらの基本的な変更可能という性質に関係する多くの問題を避けています。そして、辞書のキーとしてそれらの利用を妨げます。`callback` は `ref()` 関数の同じ名前のパラメータと同じものです。

**`getweakrefcount(object)`**

`object` を参照する弱参照とプロキシの数を返します。

**`getweakrefs(object)`**

`object` を参照するすべての弱参照とプロキシオブジェクトのリストを返します。

`class WeakKeyDictionary([dict])`

キーを弱く参照するマッピングクラス。もはやキーへの強い参照がなくなったときに、辞書のエントリは捨てられます。アプリケーションの他の部分が所有するオブジェクトへ属性を追加することもなく、それらのオブジェクトに追加データを関連づけるためにこれを使うことができます。これは属性へのアクセスをオーバーライドするオブジェクトに特に便利です。

注意: 注意: `WeakKeyDictionary` は Python 辞書型の上に作られているので、反復処理を行うときにはサイズ変更してはなりません。`WeakKeyDictionary` の場合、反復処理の最中にプログラムが行った操作が、(ガベージコレクションの副作用として)「魔法のように」辞書内の要素を消し去ってしまうため、確実なサイズ変更は困難なのです。

`class WeakValueDictionary([dict])`

値を弱く参照するマッピングクラス。値への強い参照がもはや存在しなくなったときに、辞書のエントリは捨てられます。

**ReferenceType**

弱参照オブジェクトのための型オブジェクト。

**ProxyType**

呼び出し可能でないオブジェクトのプロキシのための型オブジェクト。

**CallableProxyType**

呼び出し可能なオブジェクトのプロキシのための型オブジェクト。

**ProxyTypes**

プロキシのためのすべての型オブジェクトを含むシーケンス。これは両方のプロキシ型の名前付けに依存しないで、オブジェクトがプロキシかどうかのテストをより簡単にできます。

**exception ReferenceError**

プロキシオブジェクトが使われても、元のオブジェクトがガベージコレクションされてしまっているときに発生する例外。これは標準の `ReferenceError` 例外と同じです。

参考資料:

PEP 0205, “*Weak References*”

この機能の提案と理論的根拠。初期の実装と他の言語における類似の機能についての情報へのリンクを含んでいます。

### 3.3.1 弱参照オブジェクト

弱参照オブジェクトは属性あるいはメソッドを持ちません。しかし、リファレントがまだ存在するならば、呼び出すことでそのリファレントを取得できるようにします:

```
>>> import weakref
>>> class Object:
...     pass
...
>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
True
```

リファレントがもはや存在しないならば、参照オブジェクトの呼び出しは `None` を返します:

```
>>> del o, o2
>>> print r()
None
```

弱参照オブジェクトがまだ生きているかどうかのテストは、式 `ref() is not None` を用いて行われます。通常、参照オブジェクトを使う必要があるアプリケーションコードはこのパターンに従います:

```
# r は弱参照オブジェクト
o = r()
if o is None:
    # リファレントがガーベジコレクトされた
    print "Object has been allocated; can't frobnicate."
else:
    print "Object is still live!"
    o.do_something_useful()
```

“生存性 (liveness)” のテストを個々に行うと、スレッド化されたアプリケーションにおいて競合状態を作り出します。弱参照が呼び出される前に、他のスレッドは弱参照が無効になる原因となり得ます。上で示したイディオムは、シングルスレッド化されたアプリケーションと同じくスレッド化されたアプリケーションにおいて安全です。

### 3.3.2 例

この簡単な例では、アプリケーションが以前に参照したオブジェクトを取り出すためにオブジェクト ID を利用する方法を示します。オブジェクトに生きたままであることを強制することなく、オブジェクトの ID は他のデータ構造の中で使えます。しかし、そうする場合は、オブジェクトはまだ ID によって取り出せません。

```
import weakref

_id2obj_dict = weakref.WeakValueDictionary()

def remember(obj):
    oid = id(obj)
    _id2obj_dict[oid] = obj
    return oid

def id2obj(oid):
    return _id2obj_dict[oid]
```

### 3.3.3 拡張型における弱参照

実装の目的の一つは、弱参照によって恩恵を受けない数のような型のオブジェクトにオーバーヘッドを負わせることなく、どんな型でも弱参照メカニズムに加わることができるようにすることです。

弱く参照可能なオブジェクトに対して、弱参照メカニズムを使うために、拡張は `PyObject*` フィールドをインスタンス構造に含んでいなければなりません。オブジェクトのコンストラクタによって、それは `NULL` に初期化しなければなりません。対応する型オブジェクトの `tp_weaklistoffset` フィールドをフィールドのオフセットに設定することもしなければなりません。また、`Py_TPFLAGS_HAVE_WEAKREFS` を `tp_flags` スロットへ追加する必要もあります。例えば、インスタンス型は次のような構造に定義されます:

```
typedef struct {
    PyObject_HEAD
    PyClassObject *in_class;      /* クラスオブジェクト */
    PyObject      *in_dict;      /* 辞書 */
    PyObject      *in_weakreflist; /* 弱参照のリスト */
} PyInstanceObject;
```

インスタンスに対して静的に宣言される型オブジェクトはこのように定義されます:

```
PyTypeObject PyInstance_Type = {
    PyObject_HEAD_INIT(&PyType_Type)
    0,
    "module.instance",

    /* 簡単のためにたくさんのものを省略... */

    Py_TPFLAGS_DEFAULT | Py_TPFLAGS_HAVE_WEAKREFS /* tp_flags */
    0, /* tp_doc */
    0, /* tp_traverse */
    0, /* tp_clear */
    0, /* tp_richcompare */
    offsetof(PyInstanceObject, in_weakreflist), /* tp_weaklistoffset */
};
```

型コンストラクタは弱参照リストを NULL に初期化する責任があります:

```
static PyObject *
instance_new() {
    /* 簡単のために他の初期化を省略 */

    self->in_weakreflist = NULL;

    return (PyObject *) self;
}
```

さらに一つだけ追加すると、どんな弱参照でも取り除くためには、デストラクタは弱参照マネージャを呼び出す必要があります。オブジェクトの破壊のどんな他の部分が起きる前にこれを行うべきですが、弱参照リストが非 NULL である場合はこれが要求されるだけです:

```
static void
instance_dealloc(PyInstanceObject *inst)
{
    /* 必要なら一時オブジェクトを割り当ててください。
       しかし、まだ破壊しないでください。
       */

    if (inst->in_weakreflist != NULL)
        PyObject_ClearWeakRefs((PyObject *) inst);

    /* 普通にオブジェクトの破壊を進めてください。 */
}
```



### 3.4 fpectl — 浮動小数点例外の制御

ほとんどのコンピュータはいわゆる IEEE-754 標準に準拠した浮動小数点演算を実行します。実際のどんなコンピュータでも、浮動小数点演算が普通の浮動小数点数では表せない結果になることがあります。例えば、次を試してください。

```
>>> import math
>>> math.exp(1000)
inf
>>> math.exp(1000) / math.exp(1000)
nan
```

(上の例は多くのプラットフォームで動作します。DEC Alpha は例外かもしれません。) "Inf"は"infinity(無限)"を意味する IEEE-754 における特殊な非数値の値で、"nan"は"not a number(数ではない)"を意味します。ここで留意すべき点は、その計算を行うように Python に求めたときに非数値の結果以外に特別なことは何も起きないということです。事実、それは IEEE-754 標準に規定されたデフォルトのふるまいで、それで良ければここで読むのを止めてください。

いくつかの環境では、誤った演算がなされたところで例外を発生し、処理を止めることがより良いでしょう。fpectl モジュールはそんな状況で使うためのものです。いくつかのハードウェア製造メーカーの浮動小数点ユニットを制御できるようにします。つまり、IEEE-754 例外 Division by Zero、Overflow あるいは Invalid Operation が起きたときはいつでも SIGFPE が生成させるように、ユーザが切り替えられるようにします。あなたの python システムを構成している C コードの中へ挿入される一組のラッパーマクロと協力して、SIGFPE は捕捉され、Python FloatingPointError 例外へ変換されます。

fpectl モジュールは次の関数を定義しています。また、所定の例外を発生します:

**turnon\_sigfpe()**

SIGFPE を生成するように切り替え、適切なシグナルハンドラを設定します。

**turnoff\_sigfpe()**

浮動小数点例外のデフォルトの処理に再設定します。

**exception FloatingPointError**

turnon\_sigfpe() が実行された後に、IEEE-754 例外である Division by Zero、Overflow または Invalid operation の一つを発生する浮動小数点演算は、次にこの標準 Python 例外を発生します。

#### 3.4.1 例

以下の例は fpectl モジュールの使用を開始する方法とモジュールのテスト演算について示しています。

```

>>> import fpectl
>>> import fpetest
>>> fpectl.turnon_sigfpe()
>>> fpetest.test()
overflow          PASS
FloatingPointError: Overflow

div by 0          PASS
FloatingPointError: Division by zero
[ more output from test elided ]
>>> import math
>>> math.exp(1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
FloatingPointError: in math_1

```

### 3.4.2 制限と他に考慮すべきこと

特定のプロセッサを IEEE-754 浮動小数点エラーを捕らえるように設定することは、現在アーキテクチャごとの基準に基づきカスタムコードを必要とします。あなたの特殊なハードウェアを制御するために `fpectl` を修正することもできます。

IEEE-754 例外の Python 例外への変換には、ラッパーマクロ `PyFPE_START_PROTECT` と `PyFPE_END_PROTECT` があなたのコードに適切な方法で挿入されていることが必要です。Python 自身は `fpectl` モジュールをサポートするために修正されていますが、数値解析にとって興味ある多くの他のコードはそうではありません。

`fpectl` モジュールはスレッドセーフではありません。

参考資料:

このモジュールがどのように動作するのかについてより学習するときに、ソースディストリビューションの中のいくつかのファイルは興味を引くものでしょう。インクルードファイル `'Include/pyfpe.h'` では、このモジュールの実装について同じ長さで議論されています。`'Modules/fpetestmodule.c'` には、いくつかの使い方の例があります。多くの追加の例が `'Objects/floatobject.c'` にあります。

## 3.5 atexit — 終了ハンドラ

2.0 で追加された仕様です。

`atexit` モジュールは後始末関数を登録するための一つの関数を定義します。このような登録された関数はインタプリタが終了するときに自動的に実行されます。

注意: プログラムがシグナルで停止させられたとき、Python の致命的な内部エラーが検出されたとき、あるいは `os._exit()` が呼び出されたときには、このモジュールを通して登録された関数は呼び出されません。

これは `sys.exitfunc` 変数がはたす機能の代わりのインターフェイスです。

注意: `sys.exitfunc` を設定する他のコードとともに使用した場合には、このモジュールが正しく動作しないと思われます。特に、他のコア Python モジュールはプログラマの知らないところで `atexit` を自由に使えます。`sys.exitfunc` を使う著者は、代わりに `atexit` を使うコードに変換すべきです。`sys.exitfunc` を設定するコードを変換するもっとも簡単な方法は、`atexit` をインポートして `sys.exitfunc` へ束縛されていた関数を登録することです。

```
register(func[, *args[, **kargs]])
```

終了時に実行される関数として *func* を登録します。すべての *func* へ渡すオプションの引数を、`register()` へ引数としてわたさなければなりません。

通常のプログラムの終了時、例えば `sys.exit()` が呼び出されると、あるいは、メインモジュールの実行が完了したときに、登録されたすべての関数は後入先出順に呼び出されます。より低レベルのモジュールは通常より高レベルのモジュールより前にインポートされ、従ってより後で後始末するということが想定されています。

参考資料:

`readline` モジュール (7.20 節):

`readline` ヒストリファイルを読み書きするための `atexit` の有用な例。

### 3.5.1 `atexit` 例

次の簡単な例では、インポートされたときにモジュールがファイルからカウンタを初期化する方法を示しています。また、プログラムが終了したときにアプリケーションがこのモジュールを明示的に呼び出さなくても、自動的にカウンタの更新された値を保存する方法を示しています。

```
try:
    _count = int(open("/tmp/counter").read())
except IOError:
    _count = 0

def incrcounter(n):
    global _count
    _count = _count + n

def savecounter():
    open("/tmp/counter", "w").write("%d" % _count)

import atexit
atexit.register(savecounter)
```

`register()` に指定した位置パラメータとキーワードパラメータは、登録した関数を呼び出す際に渡されます。

```
def goodbye(name, adjective):
    print 'Goodbye, %s, it was %s to meet you.' % (name, adjective)

import atexit
atexit.register(goodbye, 'Donny', 'nice')

# or:
atexit.register(goodbye, adjective='nice', name='Donny')
```

## 3.6 `types` — 組み込み型の名前

このモジュールは標準の Python インタプリタで使われているオブジェクトの型について、名前を定義しています (拡張モジュールで定義されている型を除く)。このモジュールは以下に列挙している以外の名前をエクスポートしないので、`'from types import *'` のように使っても安全です。このモジュールの将来のバージョンで追加される名前は、`'Type'` で終わる予定です。

関数での典型的な利用方法は、以下のように引数の型によって異なる動作をする場合です:

```
from types import *
def delete(mylist, item):
    if type(item) is IntType:
        del mylist[item]
    else:
        mylist.remove(item)
```

Python 2.2 以降では、`int()` や `str()` のようなファクトリ関数は、型の名前となりましたので、`types` を使用する必要はなくなりました。上記のサンプルは、以下のように記述する事が推奨されています。

```
def delete(mylist, item):
    if isinstance(item, int):
        del mylist[item]
    else:
        mylist.remove(item)
```

このモジュールは以下の名前を定義しています。

#### **NoneType**

None の型です。

#### **TypeType**

type オブジェクトの型です (`type()` などによって返されます)。

#### **BooleanType**

bool の True と False の型です。これは組み込み関数の `bool()` のエイリアスです。

#### **IntType**

整数の型です (e.g. 1)。

#### **LongType**

長整数の型です (e.g. 1L)。

#### **FloatType**

浮動小数点数の型です (e.g. 1.0)。

#### **ComplexType**

複素数の型です (e.g. 1.0j)。Python が複素数のサポートなしでコンパイルされていた場合には定義されません。

#### **StringType**

文字列の型です (e.g. 'Spam')。

#### **UnicodeType**

Unicode 文字列の型です (e.g. u'Spam')。Python がユニコードのサポートなしでコンパイルされていた場合には定義されません。

#### **TupleType**

タプルの型です (e.g. (1, 2, 3, 'Spam'))。

#### **ListType**

リストの型です (e.g. [0, 1, 2, 3])。

#### **DictType**

辞書の型です (e.g. {'Bacon': 1, 'Ham': 0})。

#### **DictionaryType**

`DictType` の別名です。

#### **FunctionType**

ユーザー定義の関数または `lambda` の型です。

#### **LambdaType**

`FunctionType` の別名です。

#### **GeneratorType**

ジェネレータ関数の呼び出しによって生成されたイテレータオブジェクトの型です。2.2 で追加された仕様です。

#### **CodeType**

`compile()` 関数などによって返されるコードオブジェクトの型です。

#### **ClassType**

ユーザー定義のクラスの型です。

#### **InstanceType**

ユーザー定義のクラスのインスタンスの型です。

#### **MethodType**

ユーザー定義のクラスのインスタンスのメソッドの型です。

#### **UnboundMethodType**

`MethodType` の別名です。

#### **BuiltinFunctionType**

`len()` や `sys.exit()` のような組み込み関数の型です。

#### **BuiltinMethodType**

`BuiltinFunction` の別名です。

#### **ModuleType**

モジュールの型です。

#### **FileType**

`sys.stdout` のような `open` されたファイルオブジェクトの型です。

#### **XRangeType**

`xrange()` 関数によって返される `range` オブジェクトの型です。

#### **SliceType**

`slice()` 関数によって返されるオブジェクトの型です。

#### **EllipsisType**

`Ellipsis` の型です。

#### **TracebackType**

`sys.exc_traceback` に含まれるようなトレースバックオブジェクトの型です。

#### **FrameType**

フレームオブジェクトの型です。トレースバックオブジェクト `tb` の `tb.tb_frame` などです。

#### **BufferType**

`buffer()` 関数によって作られるバッファオブジェクトの型です。

#### **StringTypes**

文字列型のチェックを簡単にするための `StringType` と `UnicodeType` を含むシーケンスです。 `UnicodeType` は実行中の版の Python に含まれている場合にだけ含まれるので、2つの文字列型のシーケンスを使うよりこれを使う方が移植性が高くなります。例: `isinstance(s, types.StringTypes)`。  
2.2 で追加された仕様です。

### 3.7 UserDict — 辞書オブジェクトのためのクラスラッパー

注意: このモジュールは後方互換性のためだけに残されています。Python 2.2 より前のバージョンの Python で動作する必要のないコードを書いているのならば、組み込み dict 型から直接サブクラス化することを検討してください。

このモジュールは辞書オブジェクトのラッパーとして働くクラスを定義します。独自の辞書に似たクラスのために役に立つ基底クラスで、これを継承し既存のメソッドをオーバーライドしたり、あるいは、新しいものを追加したりすることができます。このような方法で、辞書に新しい振る舞いを追加できます。

最小限のマッピングインターフェイスをすでに持っているクラスのために、モジュールはすべての辞書メソッドを定義している mixin も定義しています。これによって、shelve モジュールのような辞書の代わりをする必要があるクラスを書くことが非常に簡単になります。

UserDict モジュールは UserDict クラスと DictMixin を定義しています:

```
class UserDict([initialdata])
```

辞書をシミュレートするクラス。インスタンスの内容は通常の辞書に保存され、UserDict インスタンスの data 属性を通してアクセスできます。initialdata が与えられれば、data はその内容で初期化されます。他の目的のために使えるように、initialdata への参照が保存されないことがあるということに注意してください。

マッピングのメソッドと演算 (節 2.3.7 を参照) に加えて、UserDict インスタンスは次の属性を提供します:

data

UserDict クラスの内容を保存するために使われる実際の辞書。

```
class DictMixin()
```

\_\_getitem\_\_、\_\_setitem\_\_、\_\_delitem\_\_ および keys といった最小の辞書インタフェースを既に持っているクラスのために、全ての辞書メソッドを定義する mixin です。

この mixin はスーパークラスとして使われるべきです。上のそれぞれのメソッドを追加することで、より多くの機能がだんだん追加されます。例えば、\_\_delitem\_\_ 以外の全てのメソッドを定義すると、使えないのは pop と popitem だけになります。

4 つの基底メソッドに加えて、\_\_contains\_\_、\_\_iter\_\_ および iteritems を定義すれば、順次効率化を果たすことができます。

mixin はサブクラスのコンストラクタについて何も知らないので、\_\_init\_\_() や copy() は定義していません。

### 3.8 UserList — リストオブジェクトのためのクラスラッパー

注意: このモジュールは後方互換性のためだけに残されています。Python 2.2 より前のバージョンの Python で動作する必要のないコードを書いているのならば、組み込み list 型から直接サブクラス化することを検討してください。

このモジュールはリストオブジェクトのラッパーとして働くクラスを定義します。独自のリストに似たクラスのために役に立つ基底クラスで、これを継承し既存のメソッドをオーバーライドしたり、あるいは、新しいものを追加したりすることができます。このような方法で、リストに新しい振る舞いを追加できます。

UserList モジュールは UserList クラスを定義しています:

```
class UserList([list])
```

リストをシミュレートするクラス。インスタンスの内容は通常のリストに保存され、UserList インスタンスの data 属性を通してアクセスできます。インスタンスの内容は最初に list のコピーに設



定されますが、デフォルトでは空リスト `[]` です。`list` は通常の Python リストか、`UserList`(またはサブクラス) のインスタンスのどちらかです。

変更可能シーケンスのメソッドと演算(節 2.3.6を参照)に加えて、`UserList` インスタンスは次の属性を提供します:

**data**

`UserList` クラスの内容を保存するために使われる実際の Python リストオブジェクト。

サブクラス化の要件: `UserList` のサブクラスは引数なしか、あるいは一つの引数のどちらかとともに呼び出せるコンストラクタを提供することが期待されます。新しいシーケンスを返すリスト演算は現在の実装クラスのインスタンスを作成しようとしています。そのために、データ元として使われるシーケンスオブジェクトである一つのパラメータとともにコンストラクタを呼び出せると想定しています。

導出クラスがこの要求に従いたくないならば、このクラスがサポートしているすべての特殊メソッドはオーバーライドされる必要があります。その場合に提供される必要のあるメソッドについての情報は、ソースを参考にしてください。

2.0 で変更された仕様: Python バージョン 1.5.2 と 1.6 では、コンストラクタが引数なしで呼び出し可能であることと変更可能な `data` 属性を提供するということも要求されます。Python の初期のバージョンでは、導出クラスのインスタンスを作成しようとはしません。

### 3.9 UserString — 文字列オブジェクトのためのクラスラッパー

注意: このモジュールの `UserString` クラスは後方互換性のためだけに残されています。Python 2.2 より前のバージョンの Python で動作する必要のないコードを書いているのならば、`UserString` を使う代わりに組み込み `str` 型から直接サブクラス化することを検討してください(組み込みの `MutableString` と等価なものはありません)。

このモジュールは文字列オブジェクトのラッパーとして働くクラスを定義します。独自の文字列に似たクラスのために役に立つ基底クラスで、これを継承し既存のメソッドをオーバーライドしたり、あるいは、新しいものを追加したりすることができます。このような方法で、文字列に新しい振る舞いを追加できます。

これらのクラスは実際のクラスやユニコードオブジェクトに比べてとても効率が悪いということに注意した方がよいでしょう。これは特に `MutableString` に対して当てはまります。

`UserString` モジュールは次のクラスを定義しています:

```
class UserString([sequence])
```

文字列またはユニコード文字列オブジェクトをシミュレートするクラス。インスタンスの内容は通常の文字列またはユニコード文字列オブジェクトに保存され、`UserString` インスタンスの `data` 属性を通してアクセスできます。インスタンスの内容は最初に `sequence` のコピーに設定されます。`sequence` は通常の Python 文字列またはユニコード文字列、`UserString`(またはサブクラス) のインスタンス、あるいは組み込み `str()` 関数を使って文字列に変換できる任意のシーケンスのいずれかです。

```
class MutableString([sequence])
```

このクラスは上の `UserString` から導出され、変更可能になるように文字列を再定義します。変更可能な文字列は辞書のキーとして使うことができません。なぜなら、辞書はキーとして変更不能なオブジェクトを要求するからです。このクラスの主な目的は、辞書のキーとして変更可能なオブジェクトを使うという試みを捕捉するために、継承と `__hash__()` メソッドを取り除く(オーバーライドする)必要があることを示す教育的な例を提供することです。そうしなければ、非常にエラーになりやすく、突き止めることが困難でしょう。

文字列とユニコードオブジェクトのメソッドと演算(節 8、“文字列メソッド”を参照)に加えて、`User-`

String インスタンスは次の属性を提供します:

**data**

UserString クラスの内容を保存するために使われる実際の Python 文字列またはユニコードオブジェクト。

## 3.10 operator — 関数形式の標準演算子

operator モジュールは、Python 固有の各演算子に対応している C 言語で実装された関数セットを提供します。例えば、`operator.add(x, y)` は式  $x+y$  と等価です。関数名は特殊なクラスメソッドとして扱われます; 便宜上、先頭と末尾の `'_'` を取り除いたものも提供されています。

これらの関数はそれぞれ、オブジェクトの比較、論理演算、数学演算、配列操作、および抽象型テストに分類されます。

オブジェクト比較関数は全てのオブジェクトで有効で、関数の名前はサポートする大小比較演算子からとられています:

```
lt(a, b)
le(a, b)
eq(a, b)
ne(a, b)
ge(a, b)
gt(a, b)
__lt__(a, b)
__le__(a, b)
__eq__(a, b)
__ne__(a, b)
__ge__(a, b)
__gt__(a, b)
```

これらは  $a$  および  $b$  の大小比較を行います。特に、`lt(a, b)` は  $a < b$ 、`le(a, b)` は  $a \leq b$ 、`eq(a, b)` は  $a == b$ 、`ne(a, b)` は  $a != b$ 、`gt(a, b)` は  $a > b$ 、そして `ge(a, b)` は  $a \geq b$  と等価です。

組み込み関数 `cmp()` と違って、これらの関数はどのような値を返してもよく、ブール代数值として解釈できてもできなくてもかまいません。大小比較の詳細については *Python* リファレンスマニュアルを参照してください。2.2 で追加された仕様です。

論理演算もまた全てのオブジェクトに対して適用することができ、真値テスト、同一性テストおよびブール演算をサポートします:

```
not(o)
__not__(o)
    not o の結果を返します。(オブジェクトのインスタンスには __not__() メソッドは適用されないので注意してください; この操作を定義しているのはインタプリタコアだけです。結果は __nonzero__() および __len__() メソッドによって影響されます。)
```

```
truth(o)
    o が真の場合 True を返し、そうでない場合 False を返します。この関数は bool のコンストラクタ呼び出しと同等です。
```

```
is(a, b)
    a is b を返します。オブジェクトの同一性をテストします。
```

`is_not(a, b)`

`a is not b` を返します。オブジェクトの同一性をテストします。

演算子で最も多いのは数学演算およびビット単位の演算です:

`abs(o)`

`__abs__(o)`

`o` の絶対値を返します。

`add(a, b)`

`__add__(a, b)`

数値 `a` および `b` について `a + b` を返します。

`and_(a, b)`

`__and__(a, b)`

`a` と `b` の論理積を返します。

`div(a, b)`

`__div__(a, b)`

`__future__.division` が有効でない場合には `a / b` を返します。“古い (classic)” 除算としても知られています。

`floordiv(a, b)`

`__floordiv__(a, b)`

`a // b` を返します。2.2 で追加された仕様です。

`inv(o)`

`invert(o)`

`__inv__(o)`

`__invert__(o)`

`o` のビット単位反転を返します。`~o` と同じです。Python 2.0 では名前 `invert()` および `__invert__()` が追加されました。

`lshift(a, b)`

`__lshift__(a, b)`

`a` の `b` ビット左シフトを返します。

`mod(a, b)`

`__mod__(a, b)`

`a % b` を返します。

`mul(a, b)`

`__mul__(a, b)`

数値 `a` および `b` について `a * b` を返します。

`neg(o)`

`__neg__(o)`

`o` の符号反転を返します。

`or_(a, b)`

`__or__(a, b)`

`a` と `b` の論理和を返します。

`pos(o)`

`__pos__(o)`

`o` の符号非反転を返します。

`pow(a, b)`

`__pow__(a, b)`

数値  $a$  および  $b$  について  $a ** b$  を返します。2.3 で追加された仕様です。

`rshift(a, b)`

`__rshift__(a, b)`

$a$  の  $b$  ビット右シフトを返します。

`sub(a, b)`

`__sub__(a, b)`

$a - b$  を返します。

`truediv(a, b)`

`__truediv__(a, b)`

`__future__.division` が有効な場合  $a / b$  を返します。除算としても知られています。2.2 で追加された仕様です。

`xor(a, b)`

`__xor__(a, b)`

$a$  および  $b$  の排他的論理和を返します

配列を扱う演算子には以下のようなものがあります:

`concat(a, b)`

`__concat__(a, b)`

配列  $a$  および  $b$  について  $a + b$  を返します。

`contains(a, b)`

`__contains__(a, b)`

$b$  in  $a$  を調べた結果を返します。演算子が反転しているので注意してください。関数名 `__contains__()` は Python 2.0 で追加されました。

`countOf(a, b)`

$a$  の中に  $b$  が出現する回数を返します。

`delitem(a, b)`

`__delitem__(a, b)`

$a$  でインデクスが  $b$  の要素を削除します。

`delslice(a, b, c)`

`__delslice__(a, b, c)`

$a$  でインデクスが  $b$  から  $c-1$  のスライス要素を削除します。

`getitem(a, b)`

`__getitem__(a, b)`

$a$  でインデクスが  $b$  の要素を削除します。Return the value of  $a$  at index  $b$ .

`getslice(a, b, c)`

`__getslice__(a, b, c)`

$a$  でインデクスが  $b$  から  $c-1$  のスライス要素を返します。

`indexOf(a, b)`

$a$  で最初に  $b$  が出現する場所のインデクスを返します。

`repeat(a, b)`

`__repeat__(a, b)`

配列  $a$  と整数  $b$  について  $a * b$  を返します。

`sequenceIncludes(...)`

リリース 2.0 以降で撤廃された仕様です。 `contains()` を使ってください。

`contains()` の別名です。

```
setitem(a, b, c)
```

```
__setitem__(a, b, c)
```

`a` でインデックスが `b` の要素の値を `c` に設定します。

```
setslice(a, b, c, v)
```

```
__setslice__(a, b, c, v)
```

`a` でインデックスが `b` から `c-1` のスライス要素の値を配列 `v` に設定します。

`operator` モジュールでは、オブジェクトの型を調べるための述語演算子も定義しています。注意: これらの関数が返す結果について誤って理解しないよう注意してください; インスタンスオブジェクトに対して常に信頼できる値を返すのは `isCallable()` だけです。例えば以下ようになります:

```
>>> class C:
...     pass
...
>>> import operator
>>> o = C()
>>> operator.isMappingType(o)
True
```

**isCallable(*o*)**

リリース 2.0 以降で撤廃された仕様です。 `callable()` を使ってください。

オブジェクト `o` を関数のように呼び出すことができる場合真を返し、それ以外の場合 `false` を返します。関数、バインドおよび非バインドメソッド、クラスオブジェクト、および `__call__()` メソッドをサポートするインスタンスオブジェクトは真を返します。

**isMappingType(*o*)**

オブジェクト `o` がマップ型インタフェースをサポートする場合に真を返します。辞書および全てのインスタンスオブジェクトに対しては、この値は真になります。警告: インタフェース自体が誤った定義になっているため、あるインスタンスが完全なマップ型プロトコルを備えているかを調べる信頼性のある方法は存在しません。このため、この関数によるテストはさほど便利ではありません。

**isNumberType(*o*)**

オブジェクト `o` が数値を表現している場合に真を返します。C で実装された全ての数値型、およびこれらのインスタンスオブジェクト全てに対して、この値は真になります。警告: インタフェース自体が誤った定義になっているため、あるインスタンスが完全な数値型のインタフェースをサポートしているかを調べる信頼性のある方法は存在しません。このため、この関数によるテストはさほど便利ではありません。

**isSequenceType(*o*)**

`o` が配列プロトコルをサポートする場合に真を返します。配列メソッドを C で定義している全てのオブジェクトに対して、この値は真になります。警告: インタフェース自体が誤った定義になっているため、あるインスタンスが完全な配列型のインタフェースをサポートしているかを調べる信頼性のある方法は存在しません。このため、この関数によるテストはさほど便利ではありません。

例: 0 から 256 までの序数を文字に対応付ける辞書を構築します。

```
>>> import operator
>>> d = {}
>>> keys = range(256)
>>> vals = map(chr, keys)
>>> map(operator.setitem, [d]*len(keys), keys, vals)
```

### 3.10.1 演算子から関数への対応表

下のテーブルでは、個々の抽象的な操作が、どのように Python 構文上の各演算子や `operator` モジュールの関数に対応しているかを示しています。

操作	構文	関数
加算	<code>a + b</code>	<code>add(a, b)</code>
結合	<code>seq1 + seq2</code>	<code>concat(seq1, seq2)</code>
包含テスト	<code>o in seq</code>	<code>contains(seq, o)</code>
除算	<code>a / b</code>	<code>__future__.division</code> が無効な場合の <code>div(a, b)</code> #
除算	<code>a / b</code>	<code>__future__.division</code> が有効な場合の <code>truediv(a, b)</code> #
除算	<code>a // b</code>	<code>floordiv(a, b)</code>
論理積	<code>a &amp; b</code>	<code>and_(a, b)</code>
排他的論理和	<code>a ^ b</code>	<code>xor(a, b)</code>
ビット反転	<code>~ a</code>	<code>invert(a)</code>
論理和	<code>a   b</code>	<code>or_(a, b)</code>
べき乗	<code>a ** b</code>	<code>pow(a, b)</code>
インデックス指定の代入	<code>o[k] = v</code>	<code>setitem(o, k, v)</code>
インデックス指定の削除	<code>del o[k]</code>	<code>delitem(o, k)</code>
インデックス指定	<code>o[k]</code>	<code>getitem(o, k)</code>
左シフト	<code>a &lt;&lt; b</code>	<code>lshift(a, b)</code>
剰余	<code>a % b</code>	<code>mod(a, b)</code>
積算	<code>a * b</code>	<code>mul(a, b)</code>
(算術) 否	<code>- a</code>	<code>neg(a)</code>
(論理) 否	<code>not a</code>	<code>not_(a)</code>
右シフト	<code>a &gt;&gt; b</code>	<code>rshift(a, b)</code>
配列の反復	<code>seq * i</code>	<code>repeat(seq, i)</code>
スライス指定の代入	<code>seq[i:j] = values</code>	<code>setslice(seq, i, j, values)</code>
スライス指定の削除	<code>del seq[i:j]</code>	<code>delslice(seq, i, j)</code>
スライス指定	<code>seq[i:j]</code>	<code>getslice(seq, i, j)</code>
文字列書式化	<code>s % o</code>	<code>mod(s, o)</code>
減算	<code>a - b</code>	<code>sub(a, b)</code>
真値テスト	<code>o</code>	<code>truth(o)</code>
順序付け	<code>a &lt; b</code>	<code>lt(a, b)</code>
順序付け	<code>a &lt;= b</code>	<code>le(a, b)</code>
等価性	<code>a == b</code>	<code>eq(a, b)</code>
不等性	<code>a != b</code>	<code>ne(a, b)</code>
順序付け	<code>a &gt;= b</code>	<code>ge(a, b)</code>
順序付け	<code>a &gt; b</code>	<code>gt(a, b)</code>

## 3.11 inspect — 使用中のオブジェクトを調査

2.1 で追加された仕様です。

`inspect` は、モジュール・クラス・メソッド・関数・トレースバック・フレームオブジェクト・コードオブジェクトなどのオブジェクトから情報を取得する関数を定義しており、クラスの内容を調べる、メソッド



ドのソースコードを取得する、関数の引数リストを取得して整形する、トレースバックから必要な情報だけを取得して表示する、などの処理を行う場合に利用します。

このモジュールの機能は、型チェック・ソースコードの取得・クラス／関数から情報を取得・インタプリタのスタック情報の調査、の 4 種類に分類する事ができます。

### 3.11.1 型とメンバ

`getmembers()` は、クラスやモジュールなどのオブジェクトからメンバを取得します。名前が “is” で始まる 11 個の関数は、`getmembers()` の 2 番目の引数として利用する事ができますし、以下のような特殊属性を参照できるかどうか調べる時にも使えます。

Type	Attribute	Description
module	__doc__	ドキュメント文字列
	__file__	ファイル名 (組み込みモジュールには存在しない)
class	__doc__	ドキュメント文字列
	__module__	クラスを定義しているモジュールの名前
method	__doc__	ドキュメント文字列
	__name__	メソッドが定義された時の名前
	im_class	メソッドを呼び出すために必要なクラスオブジェクト
	im_func	メソッドを実装している関数オブジェクト
	im_self	メソッドに結合しているインスタンス、または None
function	__doc__	ドキュメント文字列
	__name__	関数が定義された時の名前
	func_code	関数をコンパイルしたバイトコードを格納するコードオブジェクト
	func_defaults	引数のデフォルト値のタプル
	func_doc	(__doc__と同じ)
	func_globals	関数を定義した時のグローバル名前空間
traceback	func_name	(__name__と同じ)
	tb_frame	このレベルのフレームオブジェクト
	tb_lasti	最後に実行しようとしたバイトコード中のインストラクションを示すインデックス。
	tb_lineno	現在の Python ソースコードの行番号
frame	tb_next	このオブジェクトの内側 (このレベルから呼び出された) のトレースバックオブジェクト
	f_back	外側 (このフレームを呼び出した) のフレームオブジェクト
	f_builtins	このフレームで参照している組み込み名前空間
	f_code	このフレームで実行しているコードオブジェクト
code	f_exc_traceback	このフレームで例外が発生した場合にはトレースバックオブジェクト。それ以外なら None
	f_exc_type	このフレームで例外が発生した場合には例外型。それ以外なら None
	f_exc_value	このフレームで例外が発生した場合には例外の値。それ以外なら None
	f_globals	このフレームで参照しているグローバル名前空間
	f_lasti	最後に実行しようとしたバイトコードのインデックス。
	f_lineno	現在の Python ソースコードの行番号
	f_locals	このフレームで参照しているローカル名前空間
	f_restricted	制限実行モードなら 1、それ以外なら 0
	f_trace	このフレームのトレース関数、または None
	co_argcount	引数の数 (*、**引数は含まない)
	co_code	コンパイルされたバイトコードそのままの文字列
	co_consts	バイトコード中で使用している定数のタプル
code	co_filename	コードオブジェクトを生成したファイルのファイル名
	co_firstlineno	Python ソースコードの先頭行
	co_flags	以下の値の組み合わせ: 1=optimized   2=newlocals   4=*arg   8=**arg
	co_lnotab	文字列にエンコードした、行番号->バイトコードインデックスへの変換表
	co_name	コードオブジェクトが定義された時の名前
	co_names	ローカル変数名のタプル
	co_nlocals	ローカル変数の数
	co_stacksize	必要な仮想機械のスタックスペース
	co_varnames	引数名とローカル変数名のタプル
builtin	__doc__	ドキュメント文字列
	__name__	関数、メソッドの元々の名前
	__self__	メソッドが結合しているインスタンス、または None

Note:

(1) 2.2 で変更された仕様: `im_class` 従来、メソッドを定義しているクラスを参照するために使用していた

`getmembers(object[, predicate])`

オブジェクトの全メンバを、(名前, 値) の組み合わせのリストで返します。リストはメンバ名でソートされています。 *predicate* が指定されている場合、 *predicate* の戻り値が真となる値のみを返します。

`getmoduleinfo(path)`

*path* で指定したファイルがモジュールであればそのモジュールが Python でどのように解釈されるかを示す (*name*, *suffix*, *mode*, *mtype*) のタプルを返し、モジュールでなければ `None` を返します。*name* はパッケージ名を含まないモジュール名、*suffix* はファイル名からモジュール名を除いた残りの部分 (ドットによる拡張子とは限らない)、*mode* は `open()` で指定されるファイルモード ('r' または 'rb')、*mtype* は `imp` で定義している整数のいずれかが指定されます。モジュールタイプに付いては `imp` を参照してください。

`getmoduleinfo(path)`

*path* で指定したファイルの、パッケージ名を含まないモジュール名を返します。この処理は、インタプリタがモジュールを検索する時と同じアルゴリズムで行われます。ファイルがこのアルゴリズムで見つからない場合には `None` が返ります。

`ismodule(object)`

オブジェクトがモジュールの場合は真を返します。

`isclass(object)`

オブジェクトがクラスの場合は真を返します。

`ismethod(object)`

オブジェクトがメソッドの場合は真を返します。

`isfunction(object)`

オブジェクトが Python の関数、または無名 (lambda) 関数の場合は真を返します。

`istraceback(object)`

オブジェクトがトレースバックの場合は真を返します。

`isframe(object)`

オブジェクトがフレームの場合は真を返します。

`iscode(object)`

オブジェクトがコードの場合は真を返します。

`isbuiltin(object)`

オブジェクトが組み込み関数の場合は真を返します。

`isroutine(object)`

オブジェクトがユーザ定義か組み込みの関数・メソッドの場合は真を返します。

`ismethoddescriptor(object)`

オブジェクトがメソッドデスクリプタの場合に真を返しますが、 `ismethod()`、 `isclass()` または `isfunction()` が真の場合には真を返しません。

この機能は Python 2.2 から新たに追加されたもので、例えば `int.__add__` は真になります。このテストをパスするオブジェクトは `__get__` 属性を持ちますが `__set__` 属性を持ちません。しかしそれ以上に属性のセットには様々なものがあります。 `__name__` は通常見分けることが可能ですし、 `__doc__` も時には可能です。

デスクリプタを使って実装されたメソッドで、上記のいずれかのテストもパスしているものは、 `ismethoddescriptor()` では偽を返します。これは単に他のテストの方がもっと確実だからです – 例えば、

`ismethod()` をパスしたオブジェクトは `im_func` 属性 (など) を持っているとは期待できます。

`isdatadescriptor(object)`

オブジェクトがデータデスクリプタの場合に真を返します。

データデスクリプタは `__get__` および `__set__` 属性の両方を持ちます。データデスクリプタの例は (Python 上で定義された) プロパティや (C で定義された) `getset` やメンバです。通常、データデスクリプタは `__name__` や `__doc__` 属性を持ちます (プロパティ、`getset`、メンバは両方の属性を持っています) が、保証されているわけではありません。2.3 で追加された仕様です。

### 3.11.2 ソース参照

`getdoc(object)`

オブジェクトのドキュメンテーション文字列を取得します。タブはスペースに展開されます。コードブロックに合わせてインデントされている `docstring` を整形するため、2 行目以降では行頭の空白は削除されます。

`getcomments(object)`

オブジェクトがクラス・関数・メソッドの何れかの場合は、オブジェクトのソースコードの直後にあるコメント行 (複数行) を、単一の文字列として返します。オブジェクトがモジュールの場合、ソースファイルの先頭にあるコメントを返します。

`getfile(object)`

オブジェクトを定義している (テキストまたはバイナリの) ファイルの名前を返します。オブジェクトが組み込みモジュール・クラス・関数の場合は `TypeError` 例外が発生します。

`getmodule(object)`

オブジェクトを定義しているモジュールを推測します。

`getsourcefile(object)`

オブジェクトを定義している Python ソースファイルの名前を返します。オブジェクトが組み込みモジュール・クラス・関数の場合は、`TypeError` 例外が発生します。

## 3.12 traceback — スタックトレースの表示や取り出し

このモジュールは Python プログラムのスタックトレースを抽出し、書式を整え、表示するための標準インターフェースを提供します。モジュールがスタックトレースを表示するとき、Python インタープリタの動作を正確に模倣します。インタープリタの “ラッパー” の場合のように、プログラムの制御の元でスタックとレースを表示したいと思ったときに役に立ちます。

モジュールは `traceback` オブジェクトを使います — これは変数 `sys.exc_traceback` (非推奨) と `sys.last_traceback` に保存され、`sys.exc_info()` から三番目の項目として返されるオブジェクト型です。

モジュールは次の関数を定義します:

`print_tb(traceback[, limit[, file]])`

`traceback` から `limit` までスタックトレース項目を出力します。`limit` が省略されるか `None` の場合は、すべての項目が表示されます。`file` が省略されるか `None` の場合は、`sys.stderr` へ出力されます。それ以外の場合は、出力を受けるためのオープンしたファイルまたはファイルに類似したオブジェクトであるべきです。

`print_exception(type, value, traceback[, limit[, file]])`

例外情報と `traceback` から `limit` までスタックトレース項目を `file` へ出力します。これは次のようにす

ることで `print_tb()` とは異なります: (1) *traceback* が `None` でない場合は、ヘッダ ‘Traceback (most recent call last):’ を出力します。 (2) スタックトレースの後に例外 *type* と *value* を出力します。 (3) *type* が `SyntaxError` であり、*value* が適切な形式の場合は、エラーのおおよその位置を示すカレットを付けて構文エラーが起きた行を出力します。

`print_exc([limit[, file]])`

これは `print_exception(sys.exc_type, sys.exc_value, sys.exc_traceback, limit, file)` のための省略表現です。(非推奨の変数を使う代わりにスレッドセーフな方法で同じ情報を引き出すために、実際には `sys.exc_info()` を使います。)

`print_last([limit[, file]])`

これは `print_exception(sys.last_type, sys.last_value, sys.last_traceback, limit, file)` の省略表現です。

`print_stack([f[, limit[, file]]])`

この関数は呼び出された時点からのスタックトレースを出力します。オプションの *f* 引数は代わりの最初のスタックフレームを指定するために使えます。`print_exception()` に付いて言えば、オプションの *limit* と *file* 引数は同じ意味を持ちます。

`extract_tb(traceback[, limit])`

トレースバックオブジェクト *traceback* から *limit* まで取り出された“前処理済み”スタックトレース項目のリストを返します。スタックトレースの代わりの書式設定を行うために役に立ちます。*limit* が省略されるか `None` の場合は、すべての項目が取り出されます。“前処理済み”スタックトレース項目とは四つの部分からなる (*filename*, *line number*, *function name*, *text*) で、スタックトレースに対して通常出力される情報を表しています。*text* は前と後ろに付いている空白を取り除いた文字列です。ソースが使えない場合は `None` です。

`extract_stack([f[, limit]])`

現在のスタックフレームから生のトレースバックを取り出します。戻り値は `extract_tb()` と同じ形式です。`print_stack()` について言えば、オプションの *f* と *limit* 引数は同じ意味を持ちます。

`format_list(list)`

`extract_tb()` または `extract_stack()` が返すタブルのリストが与えられると、出力の準備を整えた文字列のリストを返します。結果として生じるリストの中の各文字列は、引数リストの中の同じインデックスの要素に対応します。各文字列は末尾に改行が付いています。その上、ソーステキスト行が `None` でないそれらの要素に対しては、文字列は内部に改行を含んでいるかもしれません。

`format_exception_only(type, value)`

トレースバックの例外部分の書式を設定します。引数は `sys.last_type` と `sys.last_value` のような例外の型と値です。戻り値はそれぞれが改行で終わっている文字列のリストです。通常、リストは一つの文字列を含んでいます。しかし、`SyntaxError` 例外に対しては、(出力されるときに) 構文エラーが起きた場所についての詳細な情報を示す行をいくつか含んでいます。どの例外が起きたのかを示すメッセージは、常にリストの最後の文字列です。

`format_exception(type, value, tb[, limit])`

スタックトレースと例外情報の書式を設定します。引数は `print_exception()` の対応する引数と同じ意味を持ちます。戻り値は文字列のリストで、それぞれの文字列は改行で終わり、そのいくつかは内部に改行を含みます。これらの行が連結されて出力される場合は、厳密に `print_exception()` と同じテキストが出力されます。

`format_tb(tb[, limit])`

`format_list(extract_tb(tb, limit))` の省略表現。

`format_stack([f[, limit]])`

`format_list(extract_stack(f, limit))` の省略表現。

`tb_lineno(tb)`

この関数はトレースバックオブジェクトに設定された現在の行番号をかえします。この関数は必要でした。なぜなら、`-O` フラグが Python へ渡されたとき、Python の 2.3 より前のバージョンでは `tb.tb_lineno` が正しく更新されなかったからです。この関数は 2.3 以降のバージョンでは役に立ちません。

### 3.12.1 トレースバックの例

この簡単な例では基本的な read-eval-print ループを実装します。それは標準的な Python の対話インタプリタループに似ていますが、Python のものより便利ではありません。インタプリタループのより完全な実装については、`code` モジュールを参照してください。

```
import sys, traceback

def run_user_code(envdir):
    source = raw_input(">>> ")
    try:
        exec source in envdir
    except:
        print "Exception in user code:"
        print '-'*60
        traceback.print_exc(file=sys.stdout)
        print '-'*60

envdir = {}
while 1:
    run_user_code(envdir)
```

## 3.13 linecache — テキストラインにランダムアクセスする

`linecache` モジュールは、キャッシュ(一つのファイルから何行も読んでおくのが一般的です)を使って、内部で最適化を図りつつ、任意のファイルの任意の行を取得するのを可能にします。

このモジュールは `traceback` モジュールで、インクルードしたソースをフォーマットされたトレースバックで復元するのに使われています。

`linecache` モジュールでは次の関数が定義されています:

**getline(filename, lineno)**

*filename* という名前のファイルから *lineno* 行目を取得します。この関数は決して例外を投げません — エラーの際には `"` を返します。(行末の改行文字は、見つかった行に含まれます。)

*filename* という名前のファイルが見つからなかった場合、モジュールの、つまり、`sys.path` でそのファイルを探します。

**clearcache()**

キャッシュをクリアします。それまでに `getline()` を使って読み込んだファイルの行が必要でなくなったら、この関数を使ってください。

**checkcache()**

キャッシュが有効かチェックします。キャッシュしたファイルにディスク上で変更があったかもしねなくて、更新が必要なときにこの関数を使ってください。

サンプル:



```
>>> import linecache
>>> linecache.getline('/etc/passwd', 4)
'sys:x:3:3:sys:/dev:/bin/sh\n'
```

## 3.14 pickle — Python オブジェクトの整列化

`pickle` モジュールでは、Python オブジェクトデータ構造を直列化 (serialize) したり非直列化 (de-serialize) するための基礎的ですが強力なアルゴリズムを実装しています。“Pickle 化 (Pickling)” は Python のオブジェクト階層をバイトストリームに変換する過程を指します。“非 Pickle 化 (unpickling)” はその逆の操作で、バイトストリームをオブジェクト階層に戻すように変換します。Pickle 化 (及び非 Pickle 化) は、別名“直列化 (serialization)”や“整列化 (marshalling)”<sup>2</sup>、“平坦化 (flattening)”として知られていますが、ここでは混乱を避けるため、用語として“Pickle 化”および“非 Pickle 化”を使います。

このドキュメントでは `pickle` モジュールおよび `cPickle` モジュールの両方について記述します。

### 3.14.1 他の Python モジュールとの関係

`pickle` モジュールには `cPickle` と呼ばれる最適化のなされた親類モジュールがあります。名前が示すように、`cPickle` は C で書かれており、このため `pickle` より 1000 倍くらいまで高速になる可能性があります。しかしながら `cPickle` では `Pickler()` および `Unpickler()` クラスのサブクラス化をサポートしていません。これは `cPickle` では、これらは関数であってクラスではないからです。ほとんどのアプリケーションではこの機能は不要であり、`cPickle` の持つ高いパフォーマンスの恩恵を受けることができます。その他の点では、二つのモジュールにおけるインタフェースはほとんど同じです; このマニュアルでは共通のインタフェースを記述しており、必要に応じてモジュール間の相違について指摘します。以下の議論では、`pickle` と `cPickle` の総称として“pickle”という用語を使うことにします。

これら二つのモジュールが生成するデータストリームは相互交換できることが保証されています。

Python には `marshal` と呼ばれるより原始的な直列化モジュールがありますが、一般的に Python オブジェクトを直列化する方法としては `pickle` を選ぶべきです。`marshal` は基本的に‘.pyc’ファイルをサポートするために存在しています。

`pickle` モジュールはいくつかの点で `marshal` と明確に異なります:

- `pickle` モジュールでは、同じオブジェクトが再度直列化されることのないよう、すでに直列化されたオブジェクトについて追跡情報を保持します。`marshal` はこれを行いません。

この機能は再帰的オブジェクトと共有オブジェクトの両方に重要な関わりをもっています。再帰的オブジェクトとは自分自身に対する参照を持っているオブジェクトです。再帰的オブジェクトは `marshal` で扱うことができず、実際、再帰的オブジェクトを `marshal` 化しようとする Python インタプリタをクラッシュさせてしまいます。共有オブジェクトは、直列化しようとするオブジェクト階層の異なる複数の場所で同じオブジェクトに対する参照が存在する場合に生じます。共有オブジェクトを共有のままにしておくことは、変更可能なオブジェクトの場合には非常に重要です。

- `marshal` はユーザ定義クラスやそのインスタンスを直列化するために使うことができません。`pickle` はクラスインスタンスを透過的に保存したり復元したりすることができますが、クラス定義をインポートすることが可能で、かつオブジェクトが保存された際と同じモジュールで定義されていなければなりません。

---

<sup>2</sup>`marshal` モジュールと間違えないように注意してください

- `marshal` の直列化フォーマットは Python の異なるバージョンで可搬性があることを保証していません。`marshal` の本来の仕事は `.pyc` ファイルのサポートなので、Python を実装する人々には、必要に応じて直列化フォーマットを以前のバージョンと互換性のないものに変更する権限が残されています。`pickle` 直列化フォーマットには、全ての Python リリース間で以前のバージョンとの互換性が保証されています。

`pickle` モジュールは誤りを含む、あるいは悪意を持って構築されたデータに対して安全にはされていません。信用できない、あるいは認証されていないデータ源から受信したデータを逆 `pickle` 化しないでください。

直列化は永続化 (persistence) よりも原始的な概念です; `pickle` はファイルオブジェクトを読み書きしますが、永続化されたオブジェクトの名前付け問題や、(より複雑な) オブジェクトに対する競合アクセスの問題を扱いません。`pickle` モジュールは複雑なオブジェクトをバイトストリームに変換することができ、バイトストリームを変換前と同じ内部構造をオブジェクトに変換することができます。このバイトストリームの最も明白な用途はファイルへの書き込みですが、その他にもネットワークを介して送信したり、データベースに記録したりすることができます。モジュール `shelve` はオブジェクトを DBM 形式のデータベースファイル上で `pickle` 化したり `unpickle` 化したりするための単純なインタフェースを提供しています。

### 3.14.2 データストリームの形式

`pickle` が使うデータ形式は Python 特有です。そうすることで、XDR のような外部の標準を持つ制限 (例えば XDR ではポインタの共有を表現できません) を課せられることがないという利点があります; しかしこれは Python で書かれていないプログラムが `pickle` 化された Python オブジェクトを再構築できない可能性があることを意味します。

標準では、`pickle` データ形式では印字可能な ASCII 表現を使います。これはバイナリ表現よりも少しかさばるデータになります。印字可能な ASCII の利用 (とその他の `pickle` 表現形式が持つ特徴) の大きな利点は、デバッグやリカバリを目的とした場合に、`pickle` 化されたファイルを標準的なテキストエディタで読めるということです。

現在、`pickle` 化に使われるプロトコルは、以下の 3 種類です。

- バージョン 0 のプロトコルは、最初の ASCII プロトコルで、以前のバージョンの Python と後方互換です。
- バージョン 1 のプロトコルは、古いバイナリ形式で、以前のバージョンの Python と後方互換です。
- バージョン 2 のプロトコルは、Python 2.3 で導入されました。新しいスタイルのクラスを、より効率よく `pickle` 化します。

詳細は PEP 307 を参照してください。

`protocol` を指定しない場合、プロトコル 0 が使われます。`protocol` に負値か `HIGHEST_PROTOCOL` を指定すると、有効なプロトコルの内、もっとも高いバージョンのものが使われます。

2.3 で変更された仕様: `bin` パラメータは非推奨で、後方互換性のためだけに提供されています。代わりに `protocol` を使うべきです。

`Pickler` コンストラクタや `dump()` および `dumps()` の引数 `bin` の値を真に設定することで、少しだけ効率の高いバイナリ形式を選ぶこともできます。`protocol version >= 1` であるとき、バイナリ形式を使用するからです。

### 3.14.3 使用法

オブジェクト階層を直列化するには、まず pickler を生成し、続いて pickler の `dump()` メソッドを呼び出します。データストリームから非直列化するには、まず unpickler を生成し、続いて unpickler の `load()` メソッドを呼び出します。pickle モジュールでは以下の定数を提供しています:

#### HIGHEST\_PROTOCOL

有効なプロトコルのうち、最も大きいバージョン。この値は、*protocol* として渡せます。2.3 で追加された仕様です。

この手続きを便利にするために、pickle モジュールでは以下の関数を提供しています:

**dump**(*object*, *file*[, *protocol*[, *bin*]])

すでに開かれているファイルオブジェクト *file* に、*object* を pickle 化したものを表現する文字列を書き込みます。Pickler(*file*, *protocol*, *bin*).dump(*object*) と同じです。

*protocol* を指定しない場合、プロトコル 0 が使われます。*protocol* に負値か HIGHEST\_PROTOCOL を指定すると、有効なプロトコルの内、もっとも高いバージョンのものが使われます。

2.3 で変更された仕様: *bin* パラメータは非推奨で、後方互換性のためだけに提供されています。代わりに *protocol* を使うべきです。

オプションの *bin* 引数が真の場合、バイナリ pickle 化形式が使われます; そうでない場合 (より低効率の) テキスト pickle 化形式が使われます (以前のバージョンとの互換性のため、こちらが標準になっています)。

*file* は、単一の文字列引数を受理する write() メソッドを持たなければなりません。従って、*file* としては、書き込みのために開かれたファイルオブジェクト、StringIO オブジェクト、その他前述のインタフェースに適合する他のカスタムオブジェクトをとることができます。

**load**(*file*)

すでに開かれているファイルオブジェクト *file* から文字列を読み出し、読み出された文字列を pickle 化されたデータ列として解釈して、もとのオブジェクト階層を再構築して返します。Unpickler(*file*).load() と同じです。

*file* は、整数引数をとる read() メソッドと、引数の必要ない readline() メソッドを持たなければなりません。これらのメソッドは両方とも文字列を返さなければなりません。従って、*file* としては、読み出しのために開かれたファイルオブジェクト、StringIO オブジェクト、その他前述のインタフェースに適合する他のカスタムオブジェクトをとることができます。

この関数はデータ列の書き込まれているモードがバイナリかそうでないかを自動的に判断します。

**dumps**(*object*[, *protocol*[, *bin*]])

*object* の pickle 化された表現を、ファイルに書き込む代わりに文字列で返します。

*protocol* を指定しない場合、プロトコル 0 が使われます。*protocol* に負値か HIGHEST\_PROTOCOL を指定すると、有効なプロトコルの内、もっとも高いバージョンのものが使われます。

2.3 で変更された仕様: *bin* パラメータは非推奨で、後方互換性のためだけに提供されています。代わりに *protocol* を使うべきです。

オプションの *bin* 引数が真の場合、バイナリの pickle 形式が使われます; そうでない場合、(より低効率の) テキストの pickle 形式が使われます (こちらが標準の設定です)。

**loads**(*string*)

pickle 化されたオブジェクト階層を文字列から読み出します。文字列中で pickle 化されたオブジェクト表現よりも後に続く文字列は無視されます。

pickle モジュールでは、以下の 3 つの例外も定義しています:

### exception PickleError

下で定義されている他の例外で共通の基底クラスです。Exception を継承しています。

### exception PicklingError

この例外は unpickle 不可能なオブジェクトが dump() メソッドに渡された場合に送出されます。

### exception UnpicklingError

この例外は、オブジェクトを unpickle 化する際に問題が発生した場合に送出されます。unpickle 化中には AttributeError、EOFError、ImportError、および IndexError といった他の例外 (これだけとは限りません) も発生する可能性があるので注意してください。

pickle モジュールでは、2 つの呼び出し可能オブジェクト<sup>3</sup>として、Pickler および Unpickler を提供しています:

```
class Pickler(file[, protocol[, bin]])
```

pickle 化されたオブジェクトのデータ列を書き込むためのファイル類似のオブジェクトを引数にとります。

protocol を指定しない場合、プロトコル 0 が使われます。protocol に負値か HIGHEST\_PROTOCOL を指定すると、有効なプロトコルの内、もっとも高いバージョンのものが使われます。

2.3 で変更された仕様: bin パラメータは非推奨で、後方互換性のためだけに提供されています。代わりに protocol を使うべきです。

オプションの bin を真にすると、pickler により高効率のバイナリ pickle 形式を使うように指示します。そうでない場合、ASCII 形式が使われます (こちらの方が標準です)。

file は単一の文字列引数を受理する write() メソッドを持たなければなりません。従って、file としては、書き込みのために開かれたファイルオブジェクト、StringIO オブジェクト、その他前述のインタフェースに適合する他のカスタムオブジェクトをとることができます。

Pickler オブジェクトでは、一つ (または二つ) の public なメソッドを定義しています:

```
dump(object)
```

コンストラクタで与えられた、すでに開かれているファイルオブジェクトに object の pickle 化された表現を書き込みます。コンストラクタに渡された bin フラグの値に応じて、バイナリおよび ASCII 形式が使われます。

```
clear_memo()
```

pickler の “メモ” を消去します。メモとは、共有オブジェクトまたは再帰的なオブジェクトが値ではなく参照で記憶されるようにするために、pickler がこれまでどのオブジェクトに遭遇してきたかを記憶するデータ構造です。このメソッドは pickler を再利用する際に便利です。

注意: Python 2.3 以前では、clear\_memo() は cPickle で生成された pickler でのみ利用可能でした。pickle モジュールでは、pickler は memo と呼ばれる Python 辞書型のインスタンス変数を持ちます。従って、pickler モジュールにおける pickler のメモを消去は、以下のようにしてできます:

```
mypickler.memo.clear()
```

以前のバージョンの Python での動作をサポートする必要のないコードでは、単に clear\_memo() を使ってください。

同じ Pickler のインスタンスに対し、dump() メソッドを複数回呼び出すことは可能です。この呼び出しは、対応する Unpickler インスタンスで同じ回数だけ load() を呼び出す操作に対応します。同じ

---

<sup>3</sup> pickle では、これらの呼び出し可能オブジェクトはクラスであり、サブクラス化してその動作をカスタマイズすることができます。しかし、cPickle モジュールでは、これらの呼び出し可能オブジェクトはファクトリ関数であり、サブクラス化することができません。サブクラスを作成する共通の理由の一つは、どのオブジェクトを実際に unpickle するかを制御することです。詳細については 3.14.6 を参照してください。

オブジェクトが `dump()` を複数回呼び出して pickle 化された場合、`load()` は全て同じオブジェクトに対して参照を行います<sup>4</sup>。

Unpickler オブジェクトは以下のように定義されています:

`class Unpickler(file)`

pickle データ列を読み出すためのファイル類似のオブジェクトを引数に取ります。このクラスはデータ列がバイナリモードかどうかを自動的に判別します。従って、Pickler のファクトリメソッドのようなフラグを必要としません。

`file` は、整数引数を取る `read()` メソッド、および引数を持たない `readline()` メソッドの、2つのメソッドを持ちます。両方のメソッドとも文字列を返します。従って、`file` としては、読み出しのために開かれたファイルオブジェクト、StringIO オブジェクト、その他前述のインタフェースに適合する他のカスタムオブジェクトをとることができます。

Unpickler オブジェクトは 1 つ (または 2 つ) の public なメソッドを持っています:

`load()`

コンストラクタで渡されたファイルオブジェクトからオブジェクトの pickle 化表現を読み出し、中に収められている再構築されたオブジェクト階層を返します。

`noload()`

`load()` に似ていますが、実際には何もオブジェクトを生成しないという点が違います。この関数は第一に pickle 化データ列中で参照されている、“永続化 id” と呼ばれている値を検索する上で便利です。詳細は以下の 3.14.5 を参照してください。

注意: `noload()` メソッドは現在 `cPickle` モジュールで生成された Unpickler オブジェクトのみで利用可能です。pickle モジュールの Unpickler には、`noload()` メソッドがありません。

### 3.14.4 何を pickle 化したり unpickle 化できるのか?

以下の型は pickle 化できます:

- None、True、および False
- 整数、長整数、浮動小数点数、複素数
- 通常文字列および Unicode 文字列
- pickle 化可能なオブジェクトからなるタプル、リスト、および辞書
- モジュールのトップレベルで定義されている関数
- モジュールのトップレベルで定義されている組み込み関数
- モジュールのトップレベルで定義されているクラス
- `__dict__` または `__setstate__()` を pickle 化できる上記クラスのインスタンス (詳細は 3.14.5 節を参照してください)

pickle 化できないオブジェクトを pickle 化しようとすると、`PicklingError` 例外が送出されます; この例外が起きた場合、背後のファイルには未知の長さのバイト列が書き込まれてしまいます。

<sup>4</sup> 警告: これは、複数のオブジェクトを pickle 化する際に、オブジェクトやそれらの一部に対する変更を妨げないようにするための仕様です。あるオブジェクトに変更を加えて、その後同じ Pickler を使って再度 pickle 化しようとしても、そのオブジェクトは pickle 化しなおされません — そのオブジェクトに対する参照が pickle 化され、Unpickler は変更された値ではなく、元の値を返します。これには 2 つの問題点: (1) 変更の検出、そして (2) 最小限の変更を整形化すること、があります。ガーベジコレクションもまた問題になります。



(組み込みおよびユーザ定義の)関数は、値ではなく“完全記述された”参照名として pickle 化されるので注意してください。これは、関数の定義されているモジュールの名前と一緒に併せ、関数名だけが pickle 化されることを意味します。関数のコードや関数の属性は何も pickle 化されません。従って、定義しているモジュールは unpickle 化環境で import 可能でなければならず、そのモジュールには指定されたオブジェクトが含まれていなければなりません。そうでない場合、例外が送出されます<sup>5</sup>。

クラスも同様に名前参照で pickle 化されるので、unpickle 化環境には同じ制限が課せられます。クラス中のコードやデータは何も pickle 化されないので、以下の例ではクラス属性 `attr` が unpickle 化環境で復元されないことに注意してください:

```
class Foo:
    attr = 'a class attr'

picklestring = pickle.dumps(Foo)
```

pickle 化可能な関数やクラスがモジュールのトップレベルで定義されていないのはこれらの制限のためです。

同様に、クラスのインスタンスが pickle 化された際、そのクラスのコードおよびデータはオブジェクトと一緒に pickle 化されることはありません。インスタンスのデータのみが pickle 化されます。この仕様は、クラス内のバグを修正したりメソッドを追加した後も、そのクラスの以前のバージョンで作られたオブジェクトを読み出せるように意図的に行われています。あるクラスの多くのバージョンで使われるような長命なオブジェクトを作ろうと計画しているなら、そのクラスの `__setstate__()` メソッドによって適切な変換が行われるようにオブジェクトのバージョン番号を入れておくともいかもしれません。

### 3.14.5 pickle 化プロトコル

この節では pickler/unpickler と直列化対象のオブジェクトとの間のインタフェースを定義する“pickle 化プロトコル”について記述します。このプロトコルは自分のオブジェクトがどのように直列化されたり非直列化されたりするかを定義し、カスタマイズし、制御するための標準的な方法を提供します。この節での記述は、unpickle 化環境を不信な pickle 化データに対して安全にするために使う特殊なカスタマイズ化についてはカバーしていません; 詳細は 3.14.6 を参照してください。

#### 通常のクラスインスタンスの pickle 化および unpickle 化

pickle 化されたクラスインスタンスが unpickle 化されたとき、`__init__()` メソッドは通常呼び出されません。unpickle 化の際に `__init__()` が呼び出される方が望ましい場合、クラスにおいてメソッド `__getinitargs__()` を定義することができます。このメソッドはクラスコンストラクタ (すなわち `__init__()`) に渡されるべきタプルを返さなければなりません。`__getinitargs__()` メソッドは pickle 時に呼び出されます; この関数が返すタプルはインスタンスの pickle 化データに組み込まれます。

クラスは、インスタンスの pickle 化方法にさらに影響を与えることができます; クラスが `__getstate__()` メソッドを定義している場合、このメソッドが呼び出され、返された状態値はインスタンスの内容として、インスタンスの辞書の代わりに pickle 化されます。`__getstate__()` メソッドが定義されていない場合、インスタンスの `__dict__` の内容が pickle 化されます。

unpickle 化では、クラスが `__setstate__()` も定義していた場合、unpickle 化された状態値とともに呼び出されます<sup>6</sup>。`__setstate__()` メソッドが定義されていない場合、pickle 化された状態は辞書型でなければならず、その要素は新たなインスタンスの辞書に代入されます。クラスが `__getstate__()` と

<sup>5</sup>送出される例外は `ImportError` や `AttributeError` になるはずですが、他の例外も起こりえます

<sup>6</sup>これらのメソッドはクラスインスタンスのコピーを実装する際にも用いられます



`__setstate__()` の両方を定義している場合、状態値オブジェクトは辞書である必要はなく、これらのメソッドは期待通りの動作を行います。<sup>7</sup>

警告: 新しいスタイルのクラスにおいて `__getstate__()` が負値を返す場合、`__setstate__()` メソッドは呼ばれません。

### 拡張型の pickle 化および unpickle 化

Pickler が全く未知の型の — 拡張型のような — オブジェクトに遭遇した場合、pickle 化方法のヒントとして 2 個所を探します。第一は `__reduce__()` メソッドを実装しているかどうかです。もし実装されていれば、pickle 化時に `__reduce__()` メソッドが引数なしで呼び出され、この呼び出しに対しては文字列またはタプルのどちらかが返されなくてはなりません。

文字列が返された場合、文字列は通常通りに pickle 化されるグローバルな変数を表します。タプルが返される場合、タプルの長さは 2 または 3 でなければならず、以下のような意味構成になっていなければなりません:

- 呼び出し可能なオブジェクトで、unpickle 化環境において、クラスか、“安全なコンストラクタ (safe constructor)” (下を参照してください) として登録されているか、属性 `__safe_for_unpickling__` を持ち値が真に設定されているような呼び出し可能なオブジェクトでなければなりません。そうでない場合、unpickle 化環境で `UnpicklingError` が送出されます。通常通り、呼び出しオブジェクト自体はその名前が pickle 化されます。
- 呼び出し可能なオブジェクトのための引数からなるタプルか、あるいは None リリース 2.3 以降で撤廃された仕様です。引数のタプルを使ってください。
- オプションとして、オブジェクトの状態。3.14.5 節で記述されているようにして、オブジェクトの `__setstate__()` メソッドに渡されます。オブジェクトが `__setstate__()` メソッドを持たない場合、上記のように、この値は辞書でなくてはならず、オブジェクトの `__dict__` に追加されます。

unpickle 化の際、(上の条件に合致する場合) 呼び出し可能オブジェクトは引数のタプルを渡して呼び出されます; オブジェクトは unpickle 化されたオブジェクトを返さなくてはなりません。

タプルの二つ目の要素が None だった場合、呼び出し可能オブジェクトを直接呼び出す代わりに、オブジェクトの `__basicnew__()` メソッドが引数なしで呼び出されます。オブジェクトは同様に unpickle 化されたオブジェクトを返さなければなりません。

リリース 2.3 以降で撤廃された仕様です。引数のタプルを使ってください。

pickle 化するオブジェクト上で `__reduce__()` メソッドを実装する代わりに、`copy_reg` モジュールを使って呼び出し可能オブジェクトを登録する方法もあります。このモジュールはプログラムに“縮小化関数 (reduction function)”とユーザ定義型のためのコンストラクタを登録する方法を提供します。縮小化関数は、単一の引数として pickle 化するオブジェクトをとることを除き、上で述べた `__reduce__()` メソッドと同じ意味とインタフェースを持ちます。

登録されたコンストラクタは上で述べたような unpickle 化については“安全なコンストラクタ”であると考えられます。

### 外部オブジェクトの pickle 化および unpickle 化

オブジェクトの永続化を便利にするために、pickle は pickle 化されたデータ列上にないオブジェクトに対して参照を行うという概念をサポートしています。これらのオブジェクトは“永続化 id (persistent id)”で

<sup>7</sup>このプロトコルはまた、`copy` で定義されている浅いコピーや深いコピー操作でも用いられます。

参照されており、この id は単に印字可能な ASCII 文字からなる任意の文字列です。これらの名前の解決方法は pickle モジュールでは定義されていません; オブジェクトはこの名前解決を pickler および unpickler 上のユーザ定義関数にゆだねます<sup>8</sup>。

外部永続化 id の解決を定義するには、pickler オブジェクトの `persistent_id` 属性と、unpickler オブジェクトの `persistent_load` 属性を設定する必要があります。

外部永続化 id を持つオブジェクトを pickle 化するには、pickler は自作の `persistent_id()` メソッドを持たなければなりません。このメソッドは一つの引数を取り、None とオブジェクトの永続化 id のうちどちらかを返さなければなりません。None が返された場合、pickler は単にオブジェクトを通常のように pickle 化だけです。永続化 id 文字列が返された場合、pickler はその文字列に対して、unpickler がこの文字列を永続化 id として認識できるように、マーカと共に pickle 化します。

外部オブジェクトを unpickle 化するには、unpickler は自作の `persistent_load()` 関数を持たなければなりません。この関数は永続化 id 文字列を引数にとり、参照されているオブジェクトを返します。

多分より理解できるようになるようなちょっとした例を以下に示します:

---

<sup>8</sup> ユーザ定義関数に関連付けを行うための実際のメカニズムは、pickle および cPickle では少し異なります。pickle のユーザは、サブクラス化を行い、`persistent_id()` および `persistent_load()` メソッドを上書きすることで同じ効果を得ることができます

```

import pickle
from cStringIO import StringIO

src = StringIO()
p = pickle.Pickler(src)

def persistent_id(obj):
    if hasattr(obj, 'x'):
        return 'the value %d' % obj.x
    else:
        return None

p.persistent_id = persistent_id

class Integer:
    def __init__(self, x):
        self.x = x
    def __str__(self):
        return 'My name is integer %d' % self.x

i = Integer(7)
print i
p.dump(i)

datastream = src.getvalue()
print repr(datastream)
dst = StringIO(datastream)

up = pickle.Unpickler(dst)

class FancyInteger(Integer):
    def __str__(self):
        return 'I am the integer %d' % self.x

def persistent_load(persid):
    if persid.startswith('the value '):
        value = int(persid.split()[2])
        return FancyInteger(value)
    else:
        raise pickle.UnpicklingError, 'Invalid persistent id'

up.persistent_load = persistent_load

j = up.load()
print j

```

cPickle モジュール内では、unpickler の persistent\_load 属性は Python リスト型として設定することができます。この場合、unpickler が永続化 id に遭遇しても、永続化 id 文字列は単にリストに追加されるだけです。この仕様は、pickle データ中の全てのオブジェクトを実際にインスタンス化しなくても、pickle データ列中でオブジェクトに対する参照を“嗅ぎ回る”ことができるようにするために存在しています<sup>9</sup>。リストに persistent\_load を設定するやり方は、よく Unpickler クラスの noload() メソッドと共に使われます。

### 3.14.6 Unpickler をサブクラス化する

デフォルトでは、逆 pickle 化は pickle 化されたデータ中に見つかったクラスを import することになります。自前の unpickler をカスタマイズすることで、何が unpickle 化されて、どのメソッドが呼び出されるかを厳

<sup>9</sup>Guide と Jim が居間に座り込んでピクルス (pickles) を嗅いでいる光景を想像してください。

密に制御することはできます。しかし不運なことに、厳密になにを行うべきかは `pickle` と `cPickle` のどちらを使うかで異なります<sup>10</sup>。

`pickle` モジュールでは、`Unpickler` からサブクラスを導出し、`load_global()` メソッドを上書きする必要があります。`load_global()` は `pickle` データ列から最初の 2 行を読まなければならない、ここで最初の行はそのクラスを含むモジュールの名前、2 行目はそのインスタンスのクラス名になるはずで、次にこのメソッドは、例えばモジュールをインポートして属性を掘り起こすなどしてクラスを探し、発見されたものを `unpickler` のスタックに置きます。その後、このクラスは空のクラスの `__class__` 属性に代入する方法で、クラスの `__init__()` を使わずにインスタンスを魔法のように生成します。あなたの作業は(もしその作業を受け入れるなら)、`unpickler` のスタックの上に `push` された `load_global()` を、`unpickle` しても安全だと考えられる何らかのクラスの既知の安全なバージョンにすることです。あるいは全てのインスタンスに対して `unpickling` を許可したくないならエラーを送出してください。このからくりがハックのように思えるなら、あなたは間違っていない。このからくりを動かすには、ソースコードを参照してください。

`cPickle` では事情は多少すっきりしていますが、十分というわけではありません。何を `unpickle` 化するかを制御するには、`unpickler` の `find_global` 属性を関数が `None` に設定します。属性が `None` の場合、インスタンスを `unpickle` しようとする試みは全て `UnpicklingError` を送出します。属性が関数の場合、この関数はモジュール名またはクラス名を受理し、対応するクラスオブジェクトを返さなくてはなりません。このクラスが行わなくてはならないのは、クラスの探索、必要な `import` のやり直しです。そしてそのクラスのインスタンスが `unpickle` 化されるのを防ぐためにエラーを送出することもできます。

以上の話から言えることは、アプリケーションが `unpickle` 化する文字列の発信元については非常に高い注意をはらわなくてはならないということです。

### 3.14.7 例

以下にあるクラスについてどうやって `pickle` 化の振る舞いを変更するかのれいを示します。`TextReader` クラスはファイルを開き、`readline()` メソッドが呼ばれるたびに行番号と行の内容を返します。`TextReader` インスタンスが `pickle` 化された場合、ファイルオブジェクト以外の全ての属性が保存されます。インスタンスが `unpickle` 化された際、ファイルは再度開かれ、以前のファイル位置から読み出しを再開します。上記の動作を実装するために、`__setstat__()` および `__getstate__()` メソッドが使われています。

---

<sup>10</sup> 注意してください: ここで記述されている機構は内部の属性とメソッドを使っており、これらは Python の将来のバージョンで変更される対象になっています。われわれは将来、この挙動を制御するための、`pickle` および `cPickle` の両方で動作する、共通のインタフェースを提供するつもりです。

```

class TextReader:
    """Print and number lines in a text file."""
    def __init__(self, file):
        self.file = file
        self.fh = open(file)
        self.lineno = 0

    def readline(self):
        self.lineno = self.lineno + 1
        line = self.fh.readline()
        if not line:
            return None
        if line.endswith("\n"):
            line = line[:-1]
        return "%d: %s" % (self.lineno, line)

    def __getstate__(self):
        odict = self.__dict__.copy() # copy the dict since we change it
        del odict['fh']              # remove filehandle entry
        return odict

    def __setstate__(self, dict):
        fh = open(dict['file'])      # reopen file
        count = dict['lineno']       # read from file...
        while count:                # until line count is restored
            fh.readline()
            count = count - 1
        self.__dict__.update(dict)  # update attributes
        self.fh = fh               # save the file object

```

使用例は以下のようなになります:

```

>>> import TextReader
>>> obj = TextReader.TextReader("TextReader.py")
>>> obj.readline()
'1: #!/usr/local/bin/python'
>>> # (more invocations of obj.readline() here)
... obj.readline()
'7: class TextReader:'
>>> import pickle
>>> pickle.dump(obj, open('save.p', 'w'))

```

`pickle` が Python プロセス間でうまく働くことを見たいなら、先に進む前に他の Python セッションを開始してください。以下の振る舞いは同じプロセスでも新たなプロセスでも起こります。

```

>>> import pickle
>>> reader = pickle.load(open('save.p'))
>>> reader.readline()
'8:      "Print and number lines in a text file."'

```

参考資料:

`copy_reg` モジュール (3.16 節):

拡張型を登録するための Pickle インタフェース構成機構。

`shelve` モジュール (3.17 節):

オブジェクトのインデックス付きデータベース; `pickle` を使います。

`copy` モジュール (3.18 節):

オブジェクトの浅いコピーおよび深いコピー。

`marshal` モジュール (3.19 節):

高いパフォーマンスを持つ組み込み型整列化機構。

### 3.15 cPickle — より高速な pickle

`cPickle` モジュールは Python オブジェクトの直列化および非直列化をサポートし、`pickle` モジュールとほとんど同じインタフェースと機能を提供します。いくつか相違点がありますが、最も重要な違いはパフォーマンスとサブクラス化が可能かどうかです。

第一に、`cPickle` は C で実装されているため、`pickle` よりも最大で 1000 倍高速です。第二に、`cPickle` モジュール内では、呼び出し可能オブジェクト `Pickler()` および `Unpickler()` は関数で、クラスではありません。つまり、`pickle` 化や `unpickle` 化を行うカスタムのサブクラスを導出することができないということです。多くのアプリケーションではこの機能は不要なので、`cPickle` モジュールによる大きなパフォーマンス向上の恩恵を受けられるはずです。`pickle` と `cPickle` で作られた `pickle` データ列は同じなので、既存の `pickle` データに対して `pickle` と `cPickle` を互換に使用することができます<sup>11</sup>。

`cPickle` と `pickle` の API 間には他にも些細な相違がありますが、ほとんどのアプリケーションで互換性があります。より詳細なドキュメンテーションは `pickle` のドキュメントにあり、そこでドキュメント化されている相違点について挙げています。

### 3.16 copy\_reg — pickle サポート関数を登録する

`copy_reg` モジュールは `pickle` と `cPickle` モジュールに対するサポートを提供します。その上、`copy` モジュールは将来これをつかう可能性が高いです。クラスでないオブジェクトコンストラクタについての設定情報を提供します。このようなコンストラクタはファクトリ関数か、またはクラスインスタンスでしょう。

`constructor(object)`

*object* を有効なコンストラクタであると宣言します。*object* が呼び出し可能でなければ (そして、それゆえコンストラクタとして有効でないならば)、`TypeError` を発生します。

`pickle(type, function[, constructor])`

*function* が型 *type* のオブジェクトに対する “リダクション” 関数として使うことを宣言します。*type* は “標準的な” クラスオブジェクトであってはいけません。(標準的なクラスは異なった扱われ方をします。詳細は、`pickle` モジュールのドキュメンテーションを参照してください。) *function* は文字列または二ないし三つの要素を含むタプルです。

オプションの *constructor* パラメータが与えられた場合は、ピクルス化時に *function* が返した引数のタプルとともに呼びだされたときにオブジェクトを再構築するために使われ得る呼び出し可能オブジェクトです。*object* がクラスであるか、または *constructor* が呼び出し可能でない場合に、`TypeError` を発生します。

*function* と *constructor* の求められるインターフェイスについての詳細は、`pickle` モジュールを参照してください。

<sup>11</sup>`pickle` データ形式は実際には小規模なスタック指向のプログラム言語であり、またあるオブジェクトをエンコードする際に多少の自由度があるため、二つのモジュールが同じ入力オブジェクトに対して異なるデータ列を生成することもあります。しかし、常に互いに他のデータ列を読み出せることが保証されています。



## 3.17 shelve — Python オブジェクトの永続化

“シェルフ (shelf, 棚)” は辞書に似た永続性を持つオブジェクトです。“dbm” データベースとの違いは、シェルフの値 (キーではありません!) は実質上どんな Python オブジェクトにも — pickle モジュールが扱えるなら何でも — できるということです。これにはほとんどのクラスインスタンス、再帰的なデータ型、沢山の共有されたサブオブジェクトを含むオブジェクトが含まれます。キーは通常の文字列です。

```
open(filename[, flag='c'[, protocol=None[, writeback=False[, binary=None]]]])
```

永続的な辞書を開きます。指定された *filename* は、根底にあるデータベースの基本ファイル名となります。副作用として、*filename* には拡張子がつけられる場合があります、ひとつ以上のファイルが生成される可能性もあります。デフォルトでは、根底にあるデータベースファイルは読み書き可能なように開かれます。オプションの *flag* パラメタは `anydbm.open` における *flag* パラメタと同様に解釈されます。

デフォルトでは、値を整列化するにはバージョン 0 の pickle 化が用いられます。pickle 化プロトコルのバージョンは *protocol* パラメタで指定することができます。2.3 で変更された仕様: *protocol* パラメタが追加されました。*binary* パラメタは撤廃され、以前のバージョンとの互換性のためにのみ提供されています

デフォルトでは、永続的な辞書の可変エントリに対する変更をおこなっても、自動的にファイルには書き戻されません。オプションの *writeback* パラメタが `True` に設定されていれば、アクセスされたすべてのエントリはメモリ上にキャッシュされ、ファイルを閉じる際に書き戻されます; この機能は永続的な辞書上の可変の要素に対する変更を容易にしますが、多数のエントリがアクセスされた場合、膨大な量のメモリがキャッシュのために消費され、アクセスされた全てのエントリを書き戻す (アクセスされたエントリが可変であるか、あるいは実際に変更されたかを決定する方法は存在しないのです) ために、ファイルを閉じる操作を非常に低速にしまいます。

shelve オブジェクトは辞書がサポートする全てのメソッドをサポートしています。これにより、辞書ベースのスクリプトから永続的な記憶媒体を必要とするスクリプトに容易に移行できるようになります。

### 3.17.1 制限事項

- どのデータベースパッケージが使われるか (例えば `dbm`、`gdbm`、`bsddb`) は、どのインタフェースが利用可能かに依存します。従って、データベースを `dbm` を使って直接開く方法は安全ではありません。データベースはまた、`dbm` が使われた場合 (不幸なことに) その制約に縛られます — これはデータベースに記録されたオブジェクト (の pickle 化された表現) はかなり小さくなくてはならず、キー衝突が生じた場合に、稀にデータベースを更新することができなくなるということを意味します。
- 実装に依存して、永続化した辞書を閉じるときには、変更がディスクに書き込まれるかもしれないし、必ずしも書き込まれないかもしれません。Shelf クラスの `__del__` メソッドは `close` メソッドを呼び出すので、プログラマは通常この作業を明示的に行う必要はありません。
- shelve モジュールは、シェルフに置かれたオブジェクトの並列した読み出し/書き込みアクセスをサポートしません (複数の同時読み出しアクセスは安全です)。あるプログラムが書き込みのために開かれたシェルフを持っているとき、他のプログラムはそのシェルフを読み書きのために開いてはいけません。この問題を解決するために UNIX のファイルロック機構を使うことができますが、この機構は UNIX のバージョン間で異なり、使われているデータベースの実装について知識が必要となります。

```
class Shelf(dict[, protocol=None[, writeback=False[, binary=None]]])
```

UserDict.DictMixin のサブクラスで、pickle 化された値を *dict* オブジェクトに保存します。

デフォルトでは、値を整列化するにはバージョン 0 の pickle 化が用いられます。pickle 化プロトコルのバージョンは *protocol* パラメタで指定することができます。pickle 化プロトコルについては

pickle のドキュメントを参照してください。2.3 で変更された仕様: *protocol* パラメタが追加されました。 *binary* パラメタは撤廃され、以前のバージョンとの互換性のためにのみ提供されています。 *writeback* パラメタが *True* に設定されていれば、アクセスされたすべてのエントリはメモリ上にキャッシュされ、ファイルを閉じる際に書き戻されます; この機能により、可変のエントリに対して自然な操作が可能になりますが、さらに多くのメモリを消費し、辞書をファイルと同期して閉じる際に長い時間がかかるようになります。

```
class BsdDbShelf(dict[, protocol=None[, writeback=False[, binary=None]]])
```

Shelf のサブクラスで、*first*、*next*、*previous*、*last* および *set\_location* メソッドを公開しています。これらのメソッドは *bsddb* モジュールでは利用可能ですが、他のデータベースモジュールでは利用できません。コンストラクタに渡された *dict* オブジェクトは上記のメソッドをサポートしていません。通常は、*bsddb.hashopen*、*bsddb.btopen* または *bsddb.rnopen* のいずれかを呼び出して得られるオブジェクトが条件を満たしています。オプションの *protocol*、*writeback*、および *binary* パラメタは Shelf クラスにおけるパラメタと同様に解釈されます。

```
class DbfilenameShelf(filename[, flag='c'[, protocol=None[, writeback=False[, binary=None]]]])
```

Shelf のサブクラスで、辞書様オブジェクトの代わりに *filename* を受理します。根底にあるファイルは *anydbm.open* を使って開かれます。デフォルトでは、ファイルは読み書き可能な状態で開かれます。オプションの *flag* パラメタは *open* 関数におけるパラメタと同様に解釈されます。オプションの *protocol*、*writeback*、および *binary* パラメタは Shelf クラスにおけるパラメタと同様に解釈されます。

### 3.17.2 使用例

インタフェースは以下のコードに集約されています (*key* は文字列で、*data* は任意のオブジェクトです):

```
import shelve

d = shelve.open(filename) # open -- file may get suffix added by low-level
                           # library

d[key] = data             # store data at key (overwrites old data if
                           # using an existing key)
data = d[key]             # retrieve a COPY of data at key (raise KeyError if no
                           # such key)
del d[key]                # delete data stored at key (raises KeyError
                           # if no such key)
flag = d.has_key(key)     # true if the key exists
list = d.keys()           # a list of all existing keys (slow!)

# as d was opened WITHOUT writeback=True, beware:
d['xx'] = range(4)        # this works as expected, but...
d['xx'].append(5)         # *this doesn't!* -- d['xx'] is STILL range(4)!!!
# having opened d without writeback=True, you need to code carefully:
temp = d['xx']             # extracts the copy
temp.append(5)            # mutates the copy
d['xx'] = temp            # stores the copy right back, to persist it
# or, d=shelve.open(filename,writeback=True) would let you just code
# d['xx'].append(5) and have it work as expected, BUT it would also
# consume more memory and make the d.close() operation slower.

d.close()                # close it
```

参考資料:

*anydbm* モジュール (7.10 節):

dbm スタイルのデータベースに対する汎用インタフェース。

bsddb モジュール (7.13 節):

BSD db データベースインタフェース。

dbhash モジュール (7.11 節):

bsddb をラップする薄いレイヤで、他のデータベースモジュールのように関数 `open` を提供しています。

dbm モジュール (8.6 節):

標準の UNIX データベースインタフェース。

dumbdbm モジュール (7.14 節):

dbm インタフェースの移植性のある実装。

gdbm モジュール (8.7 節):

dbm インタフェースに基づいた GNU データベースインタフェース。

pickle モジュール (3.14 節):

shelve によって使われるオブジェクト整列化機構。

cPickle モジュール (3.15 節):

pickle の高速版。

## 3.18 copy — 浅いコピーおよび深いコピー操作

このモジュールでは汎用の (浅い / 深い) コピー操作を提供しています。

以下にインタフェースをまとめます:

```
import copy

x = copy.copy(y)          # make a shallow copy of y
x = copy.deepcopy(y)      # make a deep copy of y
```

このモジュール固有のエラーに対しては、`copy.error` が送出されます。

浅い (shallow) コピーと深い (deep) コピーの違いが関係するのは、複合オブジェクト (リストやクラスインスタンスのような他のオブジェクトを含むオブジェクト) だけです:

- 浅いコピー (*shallow copy*) は新たな複合オブジェクトを作成し、その後 (可能な限り) 元のオブジェクト中に見つかったオブジェクトに対する参照 を挿入します。
- 深いコピー (*deep copy*) は新たな複合オブジェクトを作成し、その後元のオブジェクト中に見つかったオブジェクトの コピーを挿入します。

深いコピー操作には、しばしば浅いコピー操作の時には存在しない 2 つの問題がついてまわります:

- 再帰的なオブジェクト (直接、間接に関わらず、自分自身に対する参照を持つ複合オブジェクト) は再帰ループを引き起こします。
- 深いコピーでは、何もかも をコピーするため、例えば複数のコピー間で共有されるべき管理データ構造までも、余分にコピーしてしまいます。

`deepcopy()` 関数では、これらの問題を以下のようにして回避しています:

- 現在のコピー過程ですでにコピーされたオブジェクトからなる、“メモ” 辞書を保持します; かつ

- ユーザ定義のクラスでコピー操作やコピーされる内容の集合を上書きできるようにします。

現在のバージョンでは、モジュール、クラス、関数、メソッド、スタック追跡、スタック構造、ファイル、ソケット、ウィンドウ、アレイ、その他これらに類似の型をコピーしません。

クラスでは、pickle 化を制御するためのインタフェース: `__getinitargs__()`、`__getstate__()`、および `__setstate__()` と同じインタフェースをコピーの制御に使うことができます。これらのメソッドに関する情報は pickle モジュールの記述を参照してください。copy モジュールは pickle 用関数登録モジュール `copy_reg` を使いません。

クラス独自のコピー実装を定義するために、特殊メソッド `__copy__()` および `__deepcopy__()` を定義することができます。前者は浅いコピー操作を実装するために使われます; 追加の引数はありません。後者は深いコピー操作を実現するために呼び出されます; この関数には単一の引数としてメモ辞書が渡されます。 `__deepcopy__()` の実装で、内容のオブジェクトに対して深いコピーを生成する必要がある場合、`deepcopy()` を呼び出し、最初の引数にそのオブジェクトを、メモ辞書を二つ目の引数に与えなければなりません。

参考資料:

pickle モジュール (3.14 節):

オブジェクト状態の取得と復元をサポートするために使われる特殊メソッドについて議論されています。

### 3.19 marshal — 内部使用向けの Python オブジェクト整列化

このモジュールには Python 値をバイナリ形式で読み書きできるような関数が含まれています。このバイナリ形式は Python 特有のもので、マシンアーキテクチャ非依存のものです (つまり、Python の値を PC 上でファイルに書き込み、Sun に転送し、そこで読み戻すことができます)。バイナリ形式の詳細がドキュメントされていないのは故意によるものです; この形式は (稀にしかないので) Python のバージョン間で変更される可能性があるからです。<sup>12</sup>

このモジュールは汎用の“永続化 (persistence)”モジュールではありません。汎用的な永続化や、RPC 呼び出しを通じた Python オブジェクトの転送については、モジュール `pickle` および `shelve` を参照してください。marshal モジュールは主に、“擬似コンパイルされた (pseudo-compiled)”コードの‘.pyc’ファイルへの読み書きをサポートするために存在します。従って、Python のメンテナは、必要が生じれば marshal 形式を後方互換性のないものに変更する権利を有しています。Python オブジェクトを直列化および非直列化したい場合には、`pickle` モジュールを使ってください。

+

警告: + The marshal module is not intended to be secure against + erroneous or maliciously constructed data. Never unmarshal data + received from an untrusted or unauthenticated source. + +

警告: marshal モジュールは、誤ったデータや悪意を持って作成されたデータに対する安全性を考慮していません。信頼できない、もしくは認証されていない出所からのデータを非直列化してはなりません。

全ての Python オブジェクト型がサポートされているわけではありません; 一般的には、どの起動中の Python 上に存在するかに依存しないオブジェクトだけがこのモジュールで読み書きできます。以下の型: `None`、整数、長整数、浮動小数点数、文字列、Unicode オブジェクト、タプル、リスト、辞書、タプルとして解釈されるコードオブジェクト、がサポートされています。リストと辞書は含まれている要素もサポートされている型であるもののみサポートされています; 再帰的なリストおよび辞書は書き込んではいけません。

<sup>12</sup> このモジュールの名前は (特に) Modula-3 の設計者の間で使われていた用語の一つに由来しています。彼らはデータを自己充足的な形式で輸送する操作に“整列化 (marshalling)”という用語を使いました。厳密に言えば、“整列させる (to marshal)”とは、あるデータを (例えば RPC バッファのように) 内部表現形式から外部表現形式に変換することを意味し、“非整列化 (unmarshalling)”とはその逆を意味します。

ん (無限ループを引き起こしてしまいます)。

補足説明: C 言語の `long int` が (DEC Alpha のように) 32 ビットよりも長いビット長を持つ場合、32 ビットよりも長い Python 整数を作成することが可能です。そのような整数が整列化された後、C 言語の `long int` のビット長が 32 ビットしかないマシン上で読み戻された場合、通常整数の代わりに Python 長整数が返されます。型は異なりますが、数値は同じです。(この動作は Python 2.2 で新たに追加されたものです。それ以前のバージョンでは、値のうち最小桁から 32 ビット以外の情報は失われ、警告メッセージが出力されます。)

文字列を操作する関数と同様に、ファイルの読み書きを行う関数が提供されています。

このモジュールでは以下の関数を定義しています:

**dump**(*value*, *file*)

開かれたファイルに値を書き込みます。値はサポートされている型でなくてはなりません。ファイルは `sys.stdout` か、`open()` や `posix.popen()` が返すようなファイルオブジェクトでなくてはなりません。またファイルはバイナリモード ('wb' または 'w+b') で開かれていなければなりません。

値 (または値のオブジェクトに含まれるオブジェクト) がサポートされていない型の場合、`ValueError` 例外が送出されます — が、同時にごみのデータがファイルに書き込まれます。このオブジェクトは `load()` で適切に読み出されることはないはずです。

**load**(*file*)

開かれたファイルから値を一つ読んで返します。有効な値が読み出せなかった場合、`EOFError`、`ValueError`、または `TypeError` を送出します。ファイルはバイナリモード ('rb' または 'r+b') で開かれたファイルオブジェクトでなければなりません。警告: サポートされない型を含むオブジェクトが `dump()` で整列化されている場合、`load()` は整列化不能な値を `None` で置き換えます。

**dumps**(*value*)

`dump(value, file)` でファイルに書き込まれるような文字列を返します。値はサポートされている型でなければなりません。値がサポートされていない型 (またはサポートされていない型のオブジェクトを含むような) オブジェクトの場合、`ValueError` 例外が送出されます。

**loads**(*string*)

データ文字列を値に変換します。有効な値が見つからなかった場合、`EOFError`、`ValueError`、または `TypeError` が送出されます。文字列中の他の文字は無視されます。

## 3.20 warnings — 警告の制御

2.1 で追加された仕様です。

警告メッセージは一般に、ユーザに警告しておいた方がよいような状況下にプログラムが置かれているが、その状況は (通常は) 例外を送出したりそのプログラムを終了させるほどの正当な理由がないといった状況で発されます。例えば、プログラムが古いモジュールを使っている場合には警告を発したくなるかもしれません。

Python プログラムは、このモジュールの `warn()` 関数を使うことで警告を発することができます。(C 言語のプログラムは `PyErr_Warn()` を使います; 詳細は *Python/C API Reference Manual* を参照してください)。

警告メッセージは通常 `sys.stderr` に出力されますが、その処理方法は、全ての警告に対する無視する処理から警告を例外に変更する処理まで、柔軟に変更することができます。警告の処理方法は警告カテゴリ (以下参照)、警告メッセージテキスト、そして警告を発したソースコード上の場所に基づいて変更することができます。ソースコード上の同じ場所に対して特定の警告が繰り返された場合、通常は抑制されます。



警告制御には 2 つの段階 (stage) があります: 第一に、警告が発されるたびに、メッセージを出力すべきかどうか決定が行われます; 次に、メッセージを出力するなら、メッセージはユーザによって設定が可能なフックを使って書式化され印字されます。

警告メッセージを出力するかどうかの決定は、警告フィルタによって制御されます。警告フィルタは一致規則 (matching rule) と動作からなる配列です。filterwarnings() を呼び出して一致規則をフィルタに追加することができ、resetwarnings() を呼び出してフィルタを標準設定の状態にリセットすることができます。

警告メッセージの印字は showwarning() を呼び出して行うことができ、この関数は上書きすることができます; この関数の標準の実装では、formatwarning() を呼び出して警告メッセージを書式化しますが、この関数についても自作の実装を使うことができます。

### 3.20.1 警告カテゴリ

警告カテゴリを表現する組み込み例外は数多くあります。このカテゴリ化は警告をグループごととフィルタする上で便利です。現在以下の警告カテゴリクラスが定義されています:

クラス	記述
Warning	全ての警告カテゴリクラスの基底クラスです。Exception のサブクラスです。
UserWarning	warn() の標準のカテゴリです。
DeprecationWarning	その機能が廃用化されていることを示す警告カテゴリの基底クラスです。
SyntaxWarning	その文法機能があいまいであることを示す警告カテゴリの基底クラスです。
RuntimeWarning	その実行時システム機能があいまいであることを示す警告カテゴリの基底クラスです。
FutureWarning	その構文の意味付けが将来変更される予定であることを示す警告カテゴリの基底クラスです。

これらは技術的には組み込み例外ですが、概念的には警告メカニズムに属しているのでここで記述されています。

標準の警告カテゴリをユーザの作成したコード上でサブクラス化することで、さらに別の警告カテゴリを定義することができます。警告カテゴリは常に Warning クラスのサブクラスでなければなりません。

### 3.20.2 警告フィルタ

警告フィルタは、ある警告を無視すべきか、表示すべきか、あるいは (例外を送出する) エラーにするべきかを制御します。

概念的には、警告フィルタは複数のフィルタ仕様からなる順番付けられたリストを維持しています; 何らかの特定の警告が生じると、フィルタ仕様の一致するものが見つかるまで、リスト中の各フィルタとの照合が行われます; 一致したフィルタ仕様がその警告の処理方法を決定します。フィルタの各エントリは (action, message, category, module, lineno) からなるタプルです。ここで:

- action は以下の文字列のうちの一つです:

値	処理方法
"error"	一致した警告を例外に変えます
"ignore"	一致した警告を決して出力しません
"always"	一致した警告を常に出力します
"default"	一致した警告のうち、警告の原因になったソースコード上の場所ごとに、最初の警告のみ出力します
"module"	一致した警告のうち、警告の原因になったモジュールごとに、最初の警告のみ出力します。
"once"	一致した警告のうち、警告の原因になった場所にかかわらず最初の警告のみ出力します。



- *message* は正規表現を含む文字列で、メッセージはこのパターンに一致しなければなりません (照合時には常に大小文字の区別をしないようにコンパイルされます)。
- *category* はクラス (Warning のサブクラス) です。警告クラスはこのクラスのサブクラスに一致しなければなりません。
- *module* は正規表現を含む文字列で、モジュール名はこのパターンに一致しなければなりません (照合時には常に大小文字の区別をしないようにコンパイルされます)。
- *lineno* 整数で、警告が発生した場所の行番号に一致しなければなりません、すべての行に一致する場合には 0 になります。

Warning クラスは組み込みの Exception クラスから導出されているので、警告をエラーに変えるには単に `category(message)` を `raise` します。

警告フィルタは Python インタプリタのコマンドラインに渡される `-W` オプションで初期化されます。インタプリタは `-W` オプションに渡される全ての引数を `sys.warnoptions`; に変換せずに保存します; `warnings` モジュールは最初に `import` された際にこれらの引数を解釈します (無効なオプションは `sys.stderr` にメッセージを出力した後無視されます)。

### 3.20.3 利用可能な関数

**warn**(*message*[, *category*[, *stacklevel*]])

警告を発するか、無視するか、あるいは例外を送出します。*category* 引数が与えられた場合、警告カテゴリクラスでなければなりません (上を参照してください); 標準の値は `UserWarning` です。*message* を `Warning` インスタンスで代用することもできますが、この場合 *category* は無視され、`message.__class__` が使われ、メッセージ文は `str(message)` になります。発された例外が前述した警告フィルタによってエラーに変更された場合、この関数は例外を送出します。引数 *stacklevel* は Python でラップ関数を書く際に利用することができます。例えば:

```
def deprecation(message):
    warnings.warn(message, DeprecationWarning, stacklevel=2)
```

こうすることで、警告が参照するソースコード部分を、`deprecation()` 自身ではなく `deprecation()` を呼び出した側にできます (というのも、前者の場合は警告メッセージの目的を台無しにしてしまうからです)。

**warn\_explicit**(*message*, *category*, *filename*, *lineno*[, *module*[, *registry*]])

`warn()` の機能に対する低レベルのインタフェースで、メッセージ、警告カテゴリ、ファイル名および行番号、そしてオプションのモジュール名およびレジストリ情報 (モジュールの `__warningregistry__` 辞書) を明示的に渡します。モジュール名は標準で `.py` が取り去られたファイル名になります; レジストリが渡されなかった場合、警告が抑制されることはありません。*message* は文字列のとき、*category* は `Warning` のサブクラスでなければなりません。また *message* は `Warning` のインスタンスであってもよく、この場合 *category* は無視されます。

**showwarning**(*message*, *category*, *filename*, *lineno*[, *file*])

警告をファイルに書き込みます。標準の実装では、`formatwarning(message, category, filename, lineno)` を呼び出し、返された文字列を *file* に書き込みます。*file* は標準では `sys.stderr` です。この関数は `warnings.showwarning` に別の実装を代入して置き換えることができます。

**formatwarning**(*message*, *category*, *filename*, *lineno*)

警告を通常の方法で書式化します。返される文字列内には改行が埋め込まれている可能性があり、かつ文字列は改行で終端されています。

```
filterwarnings(action[, message[, category[, module[, lineno[, append]]]]])
```

警告フィルタのリストにエントリを一つ挿入します。標準ではエントリは先頭に挿入されます; *append* が真ならば、末尾に挿入されます。この関数は引数の型をチェックし、*message* および *module* の正規表現をコンパイルしてから、これらをタプルにして警告フィルタの先頭に挿入します。従って、以前に挿入されたエントリと後で挿入されたエントリの両方が特定の警告に合致した場合、後者が前者のエントリを上書きします。引数が省略されると、標準では全てにマッチする値に設定されます。

```
resetwarnings()
```

警告フィルタをリセットします。これにより、*-W* コマンドラインオプションによるものを含め、*filterwarnings* の呼び出しによる影響はすべて無効化されます。

## 3.21 imp — import 内部へアクセスする

このモジュールは *import* 文を実装するために使われているメカニズムへのインターフェイスを提供します。次の定数と関数が定義されています:

```
get_magic()
```

バイトコンパイルされたコードファイル (‘.pyc’ ファイル) を認識するために使われるマジック文字列値を返します。(この値は Python の各バージョンで異なります。)

```
get_suffixes()
```

三つ組みのリストを返します。それぞれはモジュールの特定の型を説明しています。各三つ組みは形式 (*suffix*, *mode*, *type*) を持ちます。ここで、*suffix* は探すファイル名を作るためにモジュール名に追加する文字列です。そのファイルをオープンするために、*mode* は組み込み *open()* 関数へ渡されるモード文字列です (これはテキストファイルに対しては ‘r’、バイナリファイルに対しては ‘rb’ となります)。*type* はファイル型で、以下で説明する値 *PY\_SOURCE*、*PY\_COMPILED*、あるいは、*C\_EXTENSION* の一つを取ります。

```
find_module(name[, path])
```

検索パス *path* 上でモジュール *name* を見つけようとします。*path* がディレクトリ名のリストならば、上の *get\_suffixes()* が返す拡張子のいずれかを伴ったファイルを各ディレクトリの中で検索します。リスト内の有効でない名前は黙って無視されます (しかし、すべてのリスト項目は文字列でなければならない)。*path* が省略されるか *None* ならば、*sys.path* のディレクトリ名のリストが検索されます。しかし、最初にいくつか特別な場所を検索します。所定の名前 (*C\_BUILTIN*) をもつ組み込みモジュールを見つつけようとします。それから、フリーズされたモジュール (*PY\_FROZEN*)、同様にいくつかのシステムと他の場所がみられます (Mac では、リソース (*PY\_RESOURCE*) を探します。Windows では、特定のファイルを指すレジストリの中を見ます)。

検索が成功すれば、戻り値は三つ組み (*file*, *pathname*, *description*) です。ここで、*file* は先頭に位置を合わされたオープンファイルオブジェクトで、*pathname* は見つかったファイルのパス名です。そして、*description* は *get\_suffixes()* が返すリストに含まれているような三つ組みで、見つかったモジュールの種類を説明しています。モジュールがファイルの中にあるならば、返された *file* は *None* で、*filename* は空文字列、*description* タプルはその拡張子とモードに対して空文字列を含みます。モジュール型は上の括弧の中に示されます。検索が失敗すれば、*ImportError* が発生します。他の例外は引数または環境に問題があることを示唆します。

この関数は階層的なモジュール名 (ドットを含んだ名前) を扱いません。*P.M*、すなわち、パッケージ *P* のサブモジュール *M* を見つけるためには、パッケージ *P* を見つけてロードするために *find\_module()* と *load\_module()* を使い、それから *P.\_\_path\_\_* に設定された *path* 引数とともに *find\_module()* を使ってください。*P* 自身がドット名のときは、このレシピを再帰的に適用してください。

```
load_module(name, file, filename, description)
```

`find_module()` を使って (あるいは、互換性のある結果を作り出す検索を行って) 以前見つけたモジュールをロードします。この関数はモジュールをインポートするという以上のことを行います: モジュールが既にインポートされているならば、`reload()` と同じです!。 `name` 引数は (これがパッケージのサブモジュールならばパッケージ名を含む) 完全なモジュール名を示します。 `file` 引数はオープンしたファイルで、 `filename` は対応するファイル名です。モジュールがファイルからロードされようとしていないとき、これらはそれぞれ `None` と `"` であっても構いません。 `get_suffixes()` が返すように `description` 引数はタプルで、どの種類のモジュールがロードされなければならないかを説明するものです。

ロードが成功したならば、戻り値はモジュールオブジェクトです。そうでなければ、例外 (たいていは `ImportError`) が発生します。

重要: `file` 引数が `None` でなければ、例外が発生した時でさえ呼び出し側にはそれを閉じる責任があります。これを行うには、`try ... finally` 文をつかうことが最も良いです。

`new_module(name)`

`name` という名前の新しい空モジュールオブジェクトを返します。このオブジェクトは `sys.modules` に挿入されません。

`lock_held()`

現在インポートロックが維持されているならば、`True` を返します。そうでなければ、`False` を返します。スレッドのないプラットフォームでは、常に `False` を返します。

スレッドのあるプラットフォームでは、インポートが完了するまでインポートを実行するスレッドは内部ロックを維持します。このロックは元のインポートが完了するまで他のスレッドがインポートすることを阻止します。言い換えると、元のスレッドがそのインポート (および、もしあるならば、それによって引き起こされるインポート) の途中で構築した不完全なモジュールオブジェクトを、他のスレッドが見られないようにします。

`acquire_lock()`

実行中のスレッドでインタープリタのインポートロックを取得します。スレッドセーフなインポートフックでは、インポート時にこのロックを取得します。スレッドのないプラットフォームではこの関数は何もしません。2.3 で追加された仕様です。

`release_lock()`

インタープリタのインポートロックを解放します。スレッドのないプラットフォームではこの関数は何もしません。2.3 で追加された仕様です。

整数値をもつ次の定数はこのモジュールの中で定義されており、`find_module()` の検索結果を表すために使われます。

`PY_SOURCE`

ソースファイルとしてモジュールが発見された。

`PY_COMPILED`

コンパイルされたコードオブジェクトファイルとしてモジュールが発見された。

`C_EXTENSION`

動的にロード可能な共有ライブラリとしてモジュールが発見された。

`PY_RESOURCE`

モジュールが Macintosh リソースとして発見された。この値は Macintosh でのみ返される。

`PKG_DIRECTORY`

パッケージディレクトリとしてモジュールが発見された。

`C_BUILTIN`

モジュールが組み込みモジュールとして発見された。

## PY\_FROZEN

モジュールがフリーズされたモジュールとして発見された (`init_frozen()` を参照)。

次の定数と関数は旧式のものです。それらの機能は `find_module()` や `load_module()` を使って利用できます。後方互換性のために残されています:

## SEARCH\_ERROR

使われていません。

## `init_builtin(name)`

`name` という名前の組み込みモジュールを初期化し、そのモジュールオブジェクトを返します。モジュールが既に初期化されている場合は、再度初期化されます。いくつかのモジュールは二度初期化することができません。 — これを再び初期化しようとする、`ImportError` 例外が発生します。`name` という名前の組み込みモジュールがない場合は、`None` を返します。

## `init_frozen(name)`

`name` という名前のフリーズされたモジュールを初期化し、モジュールオブジェクトを返します。モジュールが既に初期化されている場合は、再度初期化されます。`name` という名前のフリーズされたモジュールがない場合は、`None` を返します。(フリーズされたモジュールは Python で書かれたモジュールで、そのコンパイルされたバイトコードオブジェクトが Python の `freeze` ユーティリティを使ってカスタムビルト Python インタープリタへ組み込まれています。差し当たり、`'Tools/freeze'` を参照してください。)

## `is_builtin(name)`

`name` という名前の再度初期化できる組み込みモジュールがある場合は、`1` を返します。`name` という名前の再度初期化できない組み込みモジュールがある場合は、`-1` を返します (`init_builtin()` を参照してください)。`name` という名前の組み込みモジュールがない場合は、`0` を返します。

## `is_frozen(name)`

`name` という名前のフリーズされたモジュール (`init_frozen()` を参照) がある場合は、`True` を返します。または、そのようなモジュールがない場合は、`False` を返します。

## `load_compiled(name, pathname, file)`

バイトコンパイルされたコードファイルとして実装されているモジュールをロードして初期化し、そのモジュールオブジェクトを返します。モジュールが既に初期化されている場合は、再度初期化されます。`name` 引数はモジュールオブジェクトを作ったり、アクセスするために使います。`pathname` 引数はバイトコンパイルされたコードファイルを指します。`file` 引数はバイトコンパイルされたコードファイルで、バイナリモードでオープンされ、先頭からアクセスされます。現在は、ユーザ定義のファイルをエミュレートするクラスではなく、実際のファイルオブジェクトでなければなりません。

## `load_dynamic(name, pathname[, file])`

動的ロード可能な共有ライブラリとして実装されているモジュールをロードして初期化します。モジュールが既に初期化されている場合は、再度初期化します。いくつかのモジュールではそれができず、例外が発生するかもしれません。`pathname` 引数は共有ライブラリを指していなければなりません。`name` 引数は初期化関数の名前を作るために使われます。共有ライブラリの `'initname()'` という名前の外部 C 関数が呼び出されます。オプションの `file` 引数は無視されます。(注意: 共有ライブラリはシステムに大きく依存します。また、すべてのシステムがそれをサポートしているわけではありません。)

## `load_source(name, pathname, file)`

Python ソースファイルとして実装されているモジュールをロードして初期化し、モジュールオブジェクトを返します。モジュールが既に初期化されている場合は、再度初期化します。`name` 引数はモジュールオブジェクトを作成したり、アクセスしたりするために使われます。`pathname` 引数はソースファイルを指します。`file` 引数はソースファイルで、テキストとして読み込むためにオープンされ、先頭

からアクセスされます。現在は、ユーザ定義のファイルをエミュレートするクラスではなく、実際のファイルオブジェクトでなければなりません。(拡張子 ‘.pyc’ または ‘.pyo’ をもつ) 正しく対応するバイトコンパイルされたファイルが存在する場合は、与えられたソースファイルを構文解析する代わりにそれが使われることに注意してください。

### 3.21.1 例

次の関数は Python 1.4 までの標準 import 文 (階層的なモジュール名がない) をエミュレートします。(この実装はそのバージョンでは動作しないでしょう。なぜなら、find\_module() は拡張されており、また load\_module() が 1.4 で追加されているからです。)

```
import imp
import sys

def __import__(name, globals=None, locals=None, fromlist=None):
    # Fast path: see if the module has already been imported.
    try:
        return sys.modules[name]
    except KeyError:
        pass

    # If any of the following calls raises an exception,
    # there's a problem we can't handle -- let the caller handle it.

    fp, pathname, description = imp.find_module(name)

    try:
        return imp.load_module(name, fp, pathname, description)
    finally:
        # Since we may exit via an exception, close fp explicitly.
        if fp:
            fp.close()
```

階層的なモジュール名を実装し、reload() 関数を含むより完全な例はモジュール knee にあります。knee モジュールは Python のソースディストリビューションの中の ‘Demo/imputil/’ にあります。

## 3.22 pkgutil — パッケージ拡張ユーティリティ

2.3 で追加された仕様です。

警告: これは実験的なモジュールです。将来廃止されたり、Python 2.3 beta 1 のバージョンとはまったく別のものに書き換えられる可能性があります。

このモジュールは次の単一の関数を提供します。

**extend\_path**(path, name)

パッケージを構成するモジュールのサーチパスを拡張します。パッケージの ‘\_\_init\_\_.py’ で次のように書くことを意図したものです。

```
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

上記はパッケージの \_\_path\_\_ に sys.path の全ディレクトリのサブディレクトリとしてパッケージ名と同じ名前を追加します。これは 1 つの論理的なパッケージの異なる部品を複数のディレクトリに分けて配布したいときに役立ちます。



同時に '\*.pkg' の \* の部分が *name* 引数に指定された文字列に一致するファイルの検索もおこないます。この機能は `import` で始まる特別な行がないことを除き '\*.pth' ファイルに似ています (site の項を参照)。 '\*.pkg' は重複のチェックを除き、信頼できるものとして扱われます。 '\*.pkg' ファイルの中に見つかったエントリはファイルシステム上に実在するか否かを問わず、そのまますべてパスに追加されます。(このような仕様です。)

入力パスがリストでない場合 (フリーズされたパッケージのとき) は何もせずにリターンします。入力パスが変更されていなければ、アイテムを末尾に追加しただけのコピーを返します。

`sys.path` はシーケンスであることが前提になっています。`sys.path` の要素の内、実在するディレクトリを指す (ユニコードまたは 8 ビットの) 文字列となっていないものは無視されます。ファイル名として使ったときにエラーが発生する `sys.path` のユニコード要素がある場合、この関数 (`os.path.isdir()` を実行している行) で例外が発生する可能性があります。

### 3.23 code — インタプリタ基底クラス

`code` モジュールは `read-eval-print` (読み込み-評価-表示) ループを Python で実装するための機能を提供します。対話的なインタプリタプロンプトを提供するアプリケーションを作るために使える二つのクラスと便利な関数が含まれています。

```
class InteractiveInterpreter([locals])
```

このクラスは構文解析とインタプリタ状態 (ユーザの名前空間) を取り扱います。入力バッファリングやプロンプト出力、または入力ファイル指定を扱いません (ファイル名は常に明示的に渡されます)。オプションの *locals* 引数はその中でコードが実行される辞書を指定します。その初期値は、キー '`__name__`' が '`__console__`' に設定され、キー '`__doc__`' が `None` に設定された新しく作られた辞書です。

```
class InteractiveConsole([locals[, filename]])
```

対話的な Python インタプリタの振る舞いを厳密にエミュレートします。このクラスは `InteractiveInterpreter` を元に作られていて、通常の `sys.ps1` と `sys.ps2` をつけたプロンプト出力と入力バッファリングが追加されています。

```
interact([banner[, readfunc[, local]])
```

`read-eval-print` ループを実行するための便利な関数。これは `InteractiveConsole` の新しいインスタンスを作り、*readfunc* が与えられた場合は `raw_input()` メソッドとして使われるように設定します。*local* が与えられた場合は、インタプリタループのデフォルト名前空間として使うために `InteractiveConsole` コンストラクタへ渡されます。そして、インスタンスの `interact()` メソッドは見出しとして使うために渡される *banner* を受け取り実行されます。コンソールオブジェクトは使われた後捨てられます。

```
compile_command(source[, filename[, symbol]])
```

この関数は Python のインタプリタメインループ (別名、`read-eval-print` ループ) をエミュレートしようとするプログラムにとって役に立ちます。扱いにくい部分は、ユーザが (完全なコマンドや構文エラーではなく) さらにテキストを入力すれば完全になりうる不完全なコマンドを入力したときを決定することです。この関数はほとんどの場合に実際のインタプリタメインループと同じ決定を行います。

*source* はソース文字列です。*filename* はオプションのソースが読み出されたファイル名で、デフォルトで '`<input>`' です。*symbol* はオプションの文法の開始記号で、'`single`' (デフォルト) または '`eval`' のどちらかにすべきです。

コマンドが完全で有効ならば、コードオブジェクトを返します (`compile(source, filename, symbol)` と同じ)。コマンドが完全でないならば、`None` を返します。コマンドが完全で構文エラーを含む場合は、



`SyntaxError` を発生させます。または、コマンドが無効なリテラルを含む場合は、`OverflowError` もしくは `ValueError` を発生させます。

### 3.23.1 対話的なインタプリタオブジェクト

`runsource(source[, filename[, symbol]])`

インタプリタ内のあるソースをコンパイルし実行します。引数は `compile_command()` のものと同じです。`filename` のデフォルトは '`<input>`' で、`symbol` は '`single`' です。あるいはいくつかのことが起きる可能性があります:

- 入力がか正しくない。`compile_command()` が例外 (`SyntaxError` か `OverflowError`) を起こした場合。`showsyntaxerror()` メソッドの呼び出によって、構文トレースバックが表示されるでしょう。`runsource()` は `False` を返します。
- 入力が完全でなく、さらに入力が必要。`compile_command()` が `None` を返した場合。`runsource()` は `True` を返します。
- 入力が完全。`compile_command()` がコードオブジェクトを返した場合。( `SystemExit` を除く実行時例外も処理する) `runcode()` を呼び出すことによって、コードは実行されます。`runsource()` は `False` を返します。

次の行を要求するために `sys.ps1` か `sys.ps2` のどちらを使うかを決定するために、戻り値を利用できます。

`runcode(code)`

コードオブジェクトを実行します。例外が生じたときは、トレースバックを表示するために `showtraceback()` が呼び出されます。伝わるのが許されている `SystemExit` を除くすべての例外が捉えられます。

`KeyboardInterrupt` についての注意。このコードの他の場所でこの例外が生じる可能性がありますし、常に捕らえることができるとは限りません。呼び出し側はそれを処理するために準備しておくべきです。

`showsyntaxerror([filename])`

起きたばかりの構文エラーを表示します。複数の構文エラーに対して一つあるのではないため、これはスタックトレースを表示しません。`filename` が与えられた場合は、Python のパーサが与えるデフォルトのファイル名の代わりに例外の中へ入れられます。なぜなら、文字列から読み込んでいるときはパーサは常に '`<string>`' を使うからです。出力は `write()` メソッドによって書き込まれます。

`showtraceback()`

起きたばかりの例外を表示します。スタックの最初の項目を取り除きます。なぜなら、それはインタプリタオブジェクトの実装の内部にあるからです。出力は `write()` メソッドによって書き込まれます。

`write(data)`

文字列を標準エラー스트リーム (`sys.stderr`) へ書き込みます。必要に応じて適切な出力処理を提供するために、導出クラスはこれをオーバーライドすべきです。

### 3.23.2 対話的なコンソールオブジェクト

`InteractiveConsole` クラスは `InteractiveInterpreter` のサブクラスです。以下の追加メソッドだけでなく、インタプリタオブジェクトのすべてのメソッドも提供します。

`interact([banner])`

対話的な Python コンソールをそっくりのエミュレートします。オプションの `banner` 引数は最初のや

り通りの前に表示するバナーを指定します。デフォルトでは、標準 Python インタプリタが表示するものと同じようなバナーを表示します。それに続けて、実際のインタプリタと混乱しないように(とても似ているから!) 括弧の中にコンソールオブジェクトのクラス名を表示します。

`push(line)`

ソーステキストの一行をインタプリタへ送ります。その行の末尾に改行がついてはいけません。内部に改行を持っているかもしれません。その行はバッファへ追加され、ソースとして連結された内容が渡されインタプリタの `runsource()` メソッドが呼び出されます。コマンドが実行されたか、有効であることをこれが示している場合は、バッファはリセットされます。そうでなければ、コマンドが不完全で、その行が付加された後のままバッファは残されます。さらに入力が必要ならば、戻り値は `True` です。その行がある方法で処理されたならば、`False` です(これは `runsource()` と同じです)。

`resetbuffer()`

入力バッファから処理されていないソーステキストを取り除きます。

`raw_input([prompt])`

プロンプトを書き込み、一行を読み込みます。返る行は末尾に改行を含みません。ユーザが EOF キーシーケンスを入力したときは、`EOFError` を発生させます。基本実装では、組み込み関数 `raw_input()` を使います。サブクラスはこれを異なる実装と置き換えるかもしれません。

## 3.24 codeop — Python コードをコンパイルする

`code` モジュールで行われているような Python の read-eval-print ループをエミュレートするユーティリティを `codeop` モジュールは提供します。結果的に、直接モジュールを使いたいとは思わないかもしれません。あなたのプログラムにこのようなループを含めたい場合は、代わりに `code` モジュールを使うことをおそらく望むでしょう。

この仕事には二つの部分があります:

1. 入力の一行が Python の文として完全であるかどうかを見分けられること: 簡単に言えば、次が `'>>> '` か、あるいは `'... '` かどうかを見分けます。
2. どの `future` 文をユーザが入力したのかを覚えていること。したがって、実質的にそれに続く入力をこれらとともにコンパイルすることができます。

`codeop` モジュールはこうしたことのそれぞれを行う方法とそれら両方を行う方法を提供します。

前者は実行するには:

`compile_command(source[, filename[, symbol]])`

Python コードの文字列であるべき `source` をコンパイルしてみて、`source` が有効な Python コードの場合はコードオブジェクトを返します。このような場合、コードオブジェクトのファイル名属性は、デフォルトで `'<input>'` である `filename` でしょう。`source` が有効な Python コードではないが、有効な Python コードの接頭語である場合には、`None` を返します。

`source` に問題がある場合は、例外を発生させます。無効な Python 構文がある場合は、`SyntaxError` を発生させます。また、無効なリテラルがある場合は、`OverflowError` または `ValueError` を発生させます。

`symbol` 引数は `source` が文としてコンパイルされるか (`'single'`、デフォルト)、または式としてコンパイルされたかどうかを決定します (`'eval'`)。他のどんな値も `ValueError` を発生させる原因となります。

警告: ソースの終わりに達する前に、成功した結果をもってパーサは構文解析を止めることが(できそうではなく)できます。このような場合、後ろに続く記号はエラーとならずに無視されます。例えば、改行が後ろに付くバックスラッシュには不定のゴミが付いているかもしれません。パーサの API がより良くなればすぐに、これは修正されるでしょう。

```
class Compile()
```

このクラスのインスタンスは組み込み関数 `compile()` とシグネチャが一致する `__call__()` メソッドを持っていますが、インスタンスが `__future__` 文を含むプログラムテキストをコンパイルする場合は、インスタンスは有効なその文とともに続くすべてのプログラムテキストを'覚えていて'コンパイルするという違いがあります。

```
class CommandCompiler()
```

このクラスのインスタンスは `compile_command()` とシグネチャが一致する `__call__()` メソッドを持っています。インスタンスが `__future__` 文を含むプログラムテキストをコンパイルする場合に、インスタンスは有効なその文とともにそれに続くすべてのプログラムテキストを'覚えていて'コンパイルするという違いがあります。

バージョン間の互換性についての注意: `Compile` と `CommandCompiler` は Python 2.2 で導入されました。2.2 の `future-tracking` 機能を有効にするだけでなく、2.1 と Python のより以前のバージョンとの互換性も保ちたい場合は、次のようにかくことができます

```
try:
    from codeop import CommandCompiler
    compile_command = CommandCompiler()
    del CommandCompiler
except ImportError:
    from codeop import compile_command
```

これは影響の小さい変更ですが、あなたのプログラムにおそらく望まれないグローバル状態を導入します。または、次のように書くこともできます:

```
try:
    from codeop import CommandCompiler
except ImportError:
    def CommandCompiler():
        from codeop import compile_command
        return compile_command
```

そして、新たなコンパイラオブジェクトが必要となるたびに `CommandCompiler` を呼び出します。

## 3.25 pprint — データ出力の整然化

`pprint` モジュールを使うと、Python の任意のデータ構造をインタプリタへの入力で使われる形式にして“pretty-print”できます。フォーマット化された構造の中に Python の基本的なタイプではないオブジェクトがあるなら、表示できないかもしれません。Python の定数として表現できない多くの組み込みオブジェクトと同様、ファイル、ソケット、クラスあるいはインスタンスのようなオブジェクトが含まれていた場合は出力できません。

可能であればオブジェクトをフォーマット化して 1 行に出力しますが、与えられた幅に合わないなら複数行に分けて出力します。無理に幅を設定したいなら、`PrettyPrinter` オブジェクトを作成して明示してください。

`pprint` モジュールには 1 つのクラスが定義されています:

**class PrettyPrinter(...)**

PrettyPrinter インスタンスを作ります。このコンストラクタにはいくつかのキーワードパラメータを設定できます。

*stream* キーワードで出力ストリームを設定できます；このストリームに対して呼び出されるメソッドはファイルプロトコルの `write()` メソッドだけです。もし設定されなければ、PrettyPrinter は `sys.stdout` を使用します。さらに 3 つのパラメータで出力フォーマットをコントロールできます。そのキーワードは *indent*、*depth* と *width* です。

再帰的なレベルごとに加えるインデントの量は *indent* で設定できます；デフォルト値は 1 です。他の値にすると出力が少しおかしく見えますが、ネスト化されたところが見分け易くなります。

出力されるレベルは *depth* で設定できます；出力されるデータ構造が深いなら、指定以上の深いレベルのものは `'...'` で置き換えられて表示されます。デフォルトでは、オブジェクトの深さを制限しません。

*width* パラメータを使うと、出力する幅を望みの文字数に設定できます；デフォルトでは 80 文字です。もし指定した幅にフォーマットできない場合は、できるだけ近づけます。

```
>>> import pprint, sys
>>> stuff = sys.path[:]
>>> stuff.insert(0, stuff[:])
>>> pp = pprint.PrettyPrinter(indent=4)
>>> pp.pprint(stuff)
[ [  ' ',
    '/usr/local/lib/python1.5',
    '/usr/local/lib/python1.5/test',
    '/usr/local/lib/python1.5/sunos5',
    '/usr/local/lib/python1.5/sharedmodules',
    '/usr/local/lib/python1.5/tkinter'],
  ' ',
  '/usr/local/lib/python1.5',
  '/usr/local/lib/python1.5/test',
  '/usr/local/lib/python1.5/sunos5',
  '/usr/local/lib/python1.5/sharedmodules',
  '/usr/local/lib/python1.5/tkinter']
>>>
>>> import parser
>>> tup = parser.ast2tuple(
...     parser.suite(open('pprint.py').read()))[1][1][1]
>>> pp = pprint.PrettyPrinter(depth=6)
>>> pp.pprint(tup)
(266, (267, (307, (287, (288, (...)))))
```

PrettyPrinter クラスにはいくつかの派生する関数が提供されています：

**ppformat(*object*)**

*object* をフォーマット化して文字列として返します。フォーマット化にはデフォルトのパラメータが使用されます。

**pprint(*object*[, *stream*])**

*object* をフォーマット化して *stream* に出力し、最後に改行します。*stream* が省略されたら、`sys.stdout` に出力します。これは対話型のインタプリタ上で、求める値を `print` する代わりに使用できます。フォーマット化にはデフォルトのパラメータが使用されます。

```
>>> stuff = sys.path[:]
>>> stuff.insert(0, stuff)
>>> pprint.pprint(stuff)
[<Recursion on list with id=869440>,
 ',
 '/usr/local/lib/python1.5',
 '/usr/local/lib/python1.5/test',
 '/usr/local/lib/python1.5/sunos5',
 '/usr/local/lib/python1.5/sharedmodules',
 '/usr/local/lib/python1.5/tkinter']
```

**isreadable(object)**

*object* をフォーマット化して出力できる (“readable”) か、あるいは `eval()` を使って値を再構成できるかを返します。再帰的なオブジェクトに対しては常に `false` を返します。

```
>>> pprint.isreadable(stuff)
False
```

**isrecursive(object)**

*object* が再帰的な表現かどうかを返します。

さらにもう 1 つ、関数が定義されています：

**saferepr(object)**

*object* の文字列表現を、再帰的なデータ構造から保護した形式で返します。もし *object* の文字列表現が再帰的な要素を持っているなら、再帰的な参照は ‘<Recursion on *typename* with *id=number*>’ で表示されます。出力は他と違ってフォーマット化されません。

```
>>> pprint.saferepr(stuff)
"[<Recursion on list with id=682968>, ', '/usr/local/lib/python1.5', '/usr/local/lib/python1.5/test', '/usr/local/lib/python1.5/sunos5', '/usr/local/lib/python1.5/sharedmodules', '/usr/local/lib/python1.5/tkinter']"
```

### 3.25.1 PrettyPrinter オブジェクト

PrettyPrinter インスタンスには以下のメソッドがあります：

**pformat(object)**

*object* のフォーマット化した表現を返します。これは PrettyPrinter のコンストラクタに渡されたオプションを考慮してフォーマット化されます。

**pprint(object)**

*object* のフォーマット化した表現を指定したストリームに出力し、最後に改行します。

以下のメソッドは、対応する同じ名前の関数と同じ機能を持っています。以下のメソッドをインスタンスに対して使うと、新たに PrettyPrinter オブジェクトを作る必要がないのでちょっぴり効果的です。

**isreadable(object)**

*object* をフォーマット化して出力できる (“readable”) か、あるいは `eval()` を使って値を再構成できるかを返します。これは再帰的なオブジェクトに対して `false` を返すことに注意して下さい。もし PrettyPrinter の *depth* パラメータが設定されていて、オブジェクトのレベルが設定よりも深かったら、`false` を返します。

**isrecursive(object)**

オブジェクトが再帰的な表現かどうかを返します。

このメソッドをフックとして、サブクラスがオブジェクトを文字列に変換する方法を修正するのが可能になっています。デフォルトの実装では、内部で `saferepr()` を呼び出しています。

**format** (*object*, *context*, *maxlevels*, *level*)

3 つの値を返します：*object* をフォーマット化して文字列にしたもの、その結果が読み込み可能かどうかを示すフラグ、再帰が含まれているかどうかを示すフラグ。

最初の引数は表示するオブジェクトです。2 つめの引数はオブジェクトの `id()` をキーとして含むディクショナリで、オブジェクトを含んでいる現在の（直接、間接に *object* のコンテナとして表示に影響を与える）環境です。ディクショナリ *context* の中でどのオブジェクトが表示されたか表示する必要があるなら、3 つめの返り値は `true` になります。`format()` メソッドの再帰呼び出しではこのディクショナリのコンテナに対してさらにエントリを加えます。3 つめの引数 *maxlevels* で再帰呼び出しのレベルを設定します；もし制限しないなら、0 にします。この引数は再帰呼び出しでそのまま渡されます。4 つめの引数 *level* で現在のレベルを設定します；再帰呼び出しでは、現在の呼び出しより小さい値が渡されます。2.3 で追加された仕様です。

## 3.26 repr — もう一つの `repr()` の実装

`repr` モジュールは結果の文字列の大きさを制限したオブジェクト表現を作り出すための方法を提供します。これは Python デバッガで使われていますが、他の状況でも同じように役に立つかもしれません。

このモジュールはクラスとインスタンス、それに関数を提供します：

**class Repr** ()

組み込みクラス `repr()` によく似た関数を実装するために役に立つ書式化サービスを提供します。過度に長い表現を作り出さないように、異なるオブジェクト型に対する大きさの制限が追加されます。

**aRepr**

これは下で説明される `repr()` 関数を提供するために使われる `Repr` のインスタンスです。このオブジェクトの属性を変更すると、`repr()` と Python デバッガが使うサイズ制限に影響します。

**repr** (*obj*)

これは `aRepr` の `repr()` メソッドです。同じ名前の組み込み関数が返す文字列と似ていますが、最大サイズに制限のある文字列を返します。

### 3.26.1 Repr オブジェクト

`Repr` インスタンスは様々なオブジェクト型の表現にサイズ制限を与えるために使えるいくつかのメンバーと、特定のオブジェクト型を書式化するメソッドを提供します。

**maxlevel**

再帰的な表現を作る場合の深さ制限。デフォルトは 6 です。

**maxdict**

**maxlist**

**maxtuple**

指定されたオブジェクト型に対するエントリ表現の数についての制限。`maxdict` に対するデフォルトは 4 で、その他に対しては 6 です。

**maxlong**

長整数の表現における文字数の最大値。中央の数字が抜け落ちます。デフォルトは 40 です。

**maxstring**

文字列の表現における文字数の制限。文字列の“通常の”表現は文字の材料だということに注意して



ください: 表現にエスケープシーケンスが必要とされる場合は、表現が短縮されたときにこれらはマングルされます。デフォルトは 30 です。

**maxother**

この制限は Repr オブジェクトに利用できる特定の書式化メソッドがないオブジェクト型のサイズをコントロールするために使われます。maxstring と同じようなやり方で適用されます。デフォルトは 20 です。

**repr(obj)**

インスタンスが強制する書式化を使う組み込み repr() と等価なもの。

**repr1(obj, level)**

repr() が使う再帰的な実装。これはどの書式化メソッドを呼び出すかを決定するために obj の型を使い、それを obj と level に渡します。再帰呼び出しにおいて level の値に対して level - 1 を与える再帰的な書式化を実行するために、型に固有のメソッドは repr1() を呼び出します。

**repr\_type(obj, level)**

型名に基づく名前をもつメソッドとして、特定の型に対する書式化メソッドは実装されます。メソッド名では、type は string.join(string.split(type(obj).\_\_name\_\_, '\_')) に置き換えられます。これらのメソッドへのディスパッチは repr1() によって処理されます。再帰的に値の書式を整える必要がある型固有のメソッドは、'self.repr1(subobj, level - 1)' を呼び出します。

### 3.26.2 Repr オブジェクトをサブクラス化する

更なる組み込みオブジェクト型へのサポートを追加するためや、すでにサポートされている型の扱いを変更するために、Repr.repr1() による動的なディスパッチを使って Repr をサブクラス化することができます。この例はファイルオブジェクトのための特別なサポートを追加する方法を示しています:

```
import repr
import sys

class MyRepr(repr.Repr):
    def repr_file(self, obj, level):
        if obj.name in ['<stdin>', '<stdout>', '<stderr>']:
            return obj.name
        else:
            return 'obj'

aRepr = MyRepr()
print aRepr.repr(sys.stdin)          # prints '<stdin>'
```

## 3.27 new — ランタイム内部オブジェクトの作成

new モジュールはインタプリタオブジェクト作成関数へのインターフェイスを与えます。新しいオブジェクトを“魔法を使ったように”作り出す必要がある、通常の作成関数が使えないときに、これは主にマーシャル型関数で使われます。このモジュールはインタプリタへの低レベルインターフェイスを提供します。したがって、このモジュールを使うときには注意しなければなりません。

new モジュールは次の関数を定義しています:

**instance(class[, dict])**

この関数は\_\_init\_\_() コンストラクタを呼び出さずに辞書 dict をもつ class のインスタンスを作り出します。dict が省略されるか、None である場合は、新しいインスタンスのために新しい空の辞書

が作られます。オブジェクトがいつもと同じ状態であるという保証はないことに注意してください。

**instancemethod**(*function*, *instance*, *class*)

この関数は *instance* に束縛されたメソッドオブジェクトか、あるいは *instance* が `None` の場合に束縛されていないメソッドオブジェクトを返します。 *function* は呼び出し可能でなければなりません。

**function**(*code*, *globals*[, *name*[, *argdefs*]])

与えられたコードとグローバル変数をもつ (Python) 関数を返します。 *name* を与えるならば、文字列か `None` でなければなりません。文字列の場合は、関数は与えられた名前をもつ。そうでなければ、関数名は *code.co\_name* から取られる。 *argdefs* を与える場合はタプルでなければならず、パラメータのデフォルト値を決めるために使われます。

**code**(*argcount*, *nlocals*, *stacksize*, *flags*, *codestring*, *constants*, *names*, *varnames*, *filename*, *name*, *firstlineno*, *lnotab*)

この関数は `PyCode_New()` という C 関数へのインターフェイスです。

**module**(*name*)

この関数は *name* という名前の新しいモジュールオブジェクトを返します。 *name* は文字列でなければなりません。

**classobj**(*name*, *baseclasses*, *dict*)

この関数は新しいクラスオブジェクトを返します。そのクラスオブジェクトは (クラスのタプルであるべき) *baseclasses* から派生し、名前空間 *dict* を持ち、 *name* という名前です。

## 3.28 site — サイト固有の設定フック

このモジュールは初期化中に自動的にインポートされます。

Python の初期のバージョンでは (1.5a3 を含むそれまでは)、サイト固有のモジュールを使う必要のあるスクリプトやモジュールは `'import site'` をコードの先頭付近に置いていました。もはやこれは必要ありません。

これはサイト固有のパスをモジュール検索パスへ付け加えます。

前部と後部からなる最大で四つまでのディレクトリを作成することから始めます。前部には、 `sys.prefix` と `sys.exec_prefix` を使用します。空の前部は省略されます。後部には、(Macintosh や Windows では) 空文字列を使用し、(UNIX では) 最初に `'lib/python2.3/site-packages'` を使ってから `'lib/site-python'` を使います。別個の前部-後部の組み合わせのそれぞれに対して、それが存在するディレクトリを参照しているかどうかを調べ、もしそうならば `sys.path` へ追加します。そして、設定ファイルを新しく追加されたパスからも検索します。

パス設定ファイルは `'package.pth'` という形式の名前をもつファイルです。その内容は `sys.path` に追加される追加項目 (一行に一つ) です。存在しない項目は `sys.path` へは決して追加されませんが、項目が (ファイルではなく) ディレクトリを参照しているかどうかはチェックされません。項目が `sys.path` へ二回以上追加されることはありません。空行と `#` で始まる行は読み飛ばされます。 `import` で始まる行は実行されます。

例えば、 `sys.prefix` と `sys.exec_prefix` が `'/usr/local'` に設定されていると仮定します。そのとき Python 2.3.3 ライブラリは `'/usr/local/lib/python2.3'` にインストールされています (ここで、 `sys.version` の最初の三文字だけがインストールパス名を作るために使われます)。ここにはサブディレクトリ `'/usr/local/lib/python2.3/site-packages'` があり、その中に三つのサブディレクトリ `'foo'`、`'bar'` および `'spam'` と二つのパス設定ファイル `'foo.pth'` と `'bar.pth'` をもつと仮定します。`'foo.pth'` には以下のものが記載されていると想定してください:

```
# foo package configuration

foo
bar
bletch
```

また、‘bar.pth’ には:

```
# bar package configuration

bar
```

が記載されているとします。そのとき、次のディレクトリが `sys.path` へこの順番で追加されます:

```
/usr/local/lib/python2.3/site-packages/bar
/usr/local/lib/python2.3/site-packages/foo
```

‘bletch’ は存在しないため省略されるということに注意してください。‘bar’ ディレクトリは ‘foo’ ディレクトリの前に来ます。なぜなら、‘bar.pth’ がアルファベット順で ‘foo.pth’ の前に来るからです。また、‘spam’ はどちらのパス設定ファイルにも記載されていないため、省略されます。

これらのパス操作の後に、`sitecustomize` という名前のモジュールをインポートしようします。そのモジュールは任意のサイト固有のカスタマイゼーションを行うことができます。`ImportError` 例外が発生してこのインポートに失敗した場合は、何も表示せずに無視されます。

いくつかの非 UNIX システムでは、`sys.prefix` と `sys.exec_prefix` は空で、パス操作は省略されます。しかし、`sitecustomize` のインポートはそのときでも試みられます。

## 3.29 user — ユーザー設定のフック

ポリシーとして、Python は起動時にユーザー毎の設定を行うコードを実行することはしません (ただし対話型セッションで環境変数 `PYTHONSTARTUP` が設定されていた場合にはそのスクリプトを実行します。)

しかしながら、プログラムやサイトによっては、プログラムが要求した時にユーザーごとの設定ファイルを実行できると便利なこともあります。このモジュールはそのような機構を実装しています。この機構を利用したいプログラムでは、以下の文を実行してください。

```
import user
```

`user` モジュールはユーザーのホームディレクトリの ‘.pythonrc.py’ ファイルを探し、オープンできるならグローバル名前空間で実行します (`execfile()` を利用します)。この段階で発生したエラーは catch されません。`user` モジュールを `import` したプログラムに影響します。ホームディレクトリは環境変数 `HOME` が仮定されていますが、もし設定されていなければカレントディレクトリが使われます。

ユーザーの ‘.pythonrc.py’ では Python のバージョンに従って異なる動作を行うために `sys.version` のテストを行うことが考えられます。

ユーザーへの警告: ‘.pythonrc.py’ ファイルに書く内容には慎重になってください。どのプログラムが利用しているかわからない状況で、標準のモジュールや関数のふるまいを替えることはおすすめできません。

この機構を使おうとするプログラマへの提案: あなたのパッケージ向けのオプションをユーザーが設定できるようにするシンプルな方法は、‘.pythonrc.py’ ファイルで変数を定義して、あなたのプログラムでテス

トする方法です。たとえば、spam モジュールでメッセージ出力のレベルを替える `user.spam_verbose` 変数を参照するには以下のようにします:

```
import user
try:
    verbose = user.spam_verbose # ユーザーの設定値
except AttributeError:
    verbose = 0                 # デフォルト値
```

大規模な設定の必要があるプログラムではプログラムごとの設定ファイルを作るといいです。

セキュリティやプライバシーに配慮するプログラムではこのモジュールを `import` しないでください。このモジュールを使うと、ユーザーは `‘.pythonrc.py’` に任意のコードを書くことで簡単に侵入することができます。

汎用のモジュールではこのモジュールを `import` しないでください。import したプログラムの動作にも影響してしまいます。

参考資料:

site モジュール (3.28 節):

サイト毎のカスタマイズを行う機構

### 3.30 `__builtin__` — 組み込み関数

このモジュールは Python の全ての「組み込み」識別子を直接アクセスするためのものです。例えば `__builtin__.open` は `open()` 関数のための全ての組み込み関数を表示します。第 2.1 節, “組み込み関数” も参照してください。

### 3.31 `__main__` — トップレベルのスクリプト環境

このモジュールは Python インタプリタのメインプログラムがコマンドを実行する際の環境をあらわしています。このモジュールを利用することで、通常は無名のこの環境にアクセスすることができます。実行されるコマンドは標準入力、スクリプトファイルあるいは対話環境での入力プロンプトから入力されます。この環境は Python スクリプトをメインプログラムとして実行される際によく使われる “条件付きスクリプト” の一節が実行される環境です。

```
if __name__ == "__main__":
    main()
```

### 3.32 `__future__` — Future ステートメントの定義

`__future__` は実際にモジュールであり、3 つの役割があります。

- `import` ステートメントを解析する既存のツールを混乱させるのを避け、そのステートメントがインポートしようとしているモジュールを見つけられるようにするため。
- 2.1 以前のリリースで `future` ステートメントが実行されれば、最低でもランタイム例外を投げるようにするため。(`__future__` はインポートできません。というのも、2.1 以前にはそういう名前のモ

ジュールはなかったからです。)

- いつ互換でない変化が導入され、いつ強制的になる – あるいは、なった – のか文書化するため。これは実行できる形式で書かれたドキュメントでなので、`__future__` をインポートし、その中身を調べるようプログラムすれば確かめられます。

‘`__future__.py`’ の各ステートメントは次のような形をしています:

```
FeatureName = "_Feature(" OptionalRelease ", " MandatoryRelease ", "  
                  CompilerFlag ")"
```

ここで、普通は、OptionalRelease は MandatoryRelease より小さく、2 つとも `sys.version_info` と同じフォーマットの 5 つのタプルからなります。

```
(PY_MAJOR_VERSION, # the 2 in 2.1.0a3; an int  
PY_MINOR_VERSION, # the 1; an int  
PY_MICRO_VERSION, # the 0; an int  
PY_RELEASE_LEVEL, # "alpha", "beta", "candidate" or "final"; string  
PY_RELEASE_SERIAL # the 3; an int  
)
```

OptionalRelease はその機能が導入された最初のリリースを記録します。

まだ時期が来ていない MandatoryRelease の場合、MandatoryRelease はその機能が言語の一部となるリリースを記します。

その他の場合、MandatoryRelease はその機能がいつ言語の一部になったのかを記録します。そのリリースから、あるいはそれ以降のリリースでは、この機能を使う際に future ステートメントは必要ではありませんが、future ステートメントを使い続けても構いません。

MandatoryRelease は None になるかもしれません。つまり、予定された機能が破棄されたということです。

`_Feature` クラスのインスタンスには対応する 2 つのメソッド、`getOptionalRelease()` と `getMandatoryRelease()` があります。

`CompilerFlag` は動的にコンパイルされるコードでその機能を有効にするために、組み込み関数 `compile()` の第 4 引数に渡されなければならない (ビットフィールド) フラグです。このフラグは `_Future` インスタンスの `compiler_flag` 属性に保存されています。

`__future__` で解説されている機能のうち、削除されたものはまだありません。





# 文字列処理

この章で解説されているモジュールは文字列を操作するさまざまな処理を提供します。以下に概要を示します。

<b>string</b>	一般的な文字列操作
<b>re</b>	Perl 風の式シンタックスを用いた正規表現検索とマッチ操作。
<b>struct</b>	文字列データをパックされたバイナリデータとして解釈します。
<b>difflib</b>	オブジェクト同士の違いを計算する
<b>fpformat</b>	浮動小数点をフォーマットする汎用関数。
<b>StringIO</b>	ファイルのように文字列を読み書きする。
<b>cStringIO</b>	StringIO を高速にしたものだが、サブクラス化はできない。
<b>textwrap</b>	テキストの折り返しと詰め込み
<b>encodings.idna</b>	国際化ドメイン名実装
<b>unicodedata</b>	Access the Unicode Database.
<b>stringprep</b>	RFC 3453 による文字列調製

string オブジェクトのメソッドについては、8節の“文字列型のメソッド”もごらんください。

## 4.1 string — 一般的な文字列操作

このモジュールは、文字クラスのチェックに役立つ定数と、文字列を扱う関数を定義しています。正規表現を使った関数については re モジュールを参照してください。

このモジュールで定義される定数は以下のとおりです。

### ascii\_letters

後述の `ascii_lowercase` と `ascii_uppercase` を合わせたもの。この値はロケールに依存しません。

### ascii\_lowercase

小文字 'abcdefghijklmnopqrstuvwxyz'。この値はロケールに依存せず、変更されません。

### ascii\_uppercase

大文字 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'。この値はロケールに依存せず、変更されません。

### digits

文字列 '0123456789'。

### hexdigits

文字列 '0123456789abcdefABCDEF'。

### letters

後述の `lowercase` と `uppercase` を合わせた文字列。具体的な値はロケールに依存し、`locale.setlocale()` が呼ばれたときに更新されます。

### lowercase

小文字として扱われる文字全てを含む文字列。ほとんどのシステムにおいて、これは文字列 `'abcdefghijklmnopqrstuvwxyz'` です。この定義を変更してはいけません — `upper()` と `swapcase()` に対する影響が定義されていないからです。具体的な値はロケールに依存し、`locale.setlocale()` が呼ばれたときに更新されます。

#### `octdigits`

文字列 `'01234567'`。

#### `punctuation`

`'c'` ロケールにおいて、句読点として扱われる ASCII 文字の文字列。

#### `printable`

印刷可能な文字で構成される文字列。これは `digits`、`letters`、`punctuation`、`whitespace` を組み合わせたもの。

#### `uppercase`

大文字として扱われる文字全てを含む文字列。ほとんどのシステムにおいて、`'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` です。この定義を変更してはいけません — `lower()` と `swapcase()` に対する影響が定義されていないからです。具体的な値はロケールに依存し、`locale.setlocale()` が呼ばれたときに更新されます。

#### `whitespace`

空白扱いの文字全てを含む文字列。ほとんどのシステムにおいて、これはスペース、タブ、改行、リターン、改頁、垂直タブです。この定義を変更しないこと — `strip()` と `split()` に対する影響が定義されていないからです。

このモジュールで提供されるほとんどの関数は、string オブジェクトと Unicode オブジェクトのメソッドでも定義されています。詳細は“String メソッド” (8) を参照のこと。このモジュールで定義される関数は以下のとおりです。

#### `atof(s)`

リリース 2.0 以降で撤廃された仕様です。組み込み関数 `float()` を使いましょう。

文字列を浮動小数点型の数値に変換します。文字列は Python の浮動小数点リテラルの文法に従っている必要があります。先頭に符号 (`'+'` または `'-'`) が付くのは構いません。この関数は、組み込み関数 `float()` に文字列が渡された場合と同じようにふるまいます。

注意: 文字列が渡されたとき、内在する C ライブラリによって NaN や Infinity が返されることがあります。これらの値に変換される文字列がどれであるかは、完全に C ライブラリに依存するので、ライブラリによって異なることが知られています。

#### `atoi(s[, base])`

リリース 2.0 以降で撤廃された仕様です。組み込み関数 `int()` を使いましょう。

文字列 `s` を、`base` を基数とする整数に変換します。文字列は 1 桁以上の数字で構成されている必要があります。先頭に符号 (`'+'` または `'-'`) が付くのは構いません。`base` のデフォルト値は 10。これが 0 の場合は、(符号を省いた後の) 先頭文字によってデフォルト値が決定します。`'0x'` か `'0X'` なら 16、`'0'` なら 8、その他の場合は 10 が基数になります。`base` が 16 の場合、先頭の `'0x'` や `'0X'` は常に受け付けられますが、必須ではありません。この関数は、組み込み関数 `int()` に文字列が渡されたときと同じようにふるまいます。(注意: より柔軟な数値リテラル解釈が必要であれば、組み込み関数 `eval()` を使いましょう)

#### `atol(s[, base])`

リリース 2.0 以降で撤廃された仕様です。組み込み関数 `long()` を使いましょう。

文字列 `s` を、`base` を基数とする long 整数に変換します。文字列は 1 桁以上の数字で構成されている必要があります。先頭に符号 (`'+'` または `'-'`) が付くのは構いません。引数 `base` は `atoi()` の場

合と同じ意味。基数が 0 の場合を除いて、文字列末尾の 'l' や 'L' は認められません。base が与えられないとき、この値に 10 が指定されたとき、この関数は組み込み関数 long() に文字列が渡されたときと同じようにふるまいます。

**capitalize(word)**

先頭文字だけ大文字にした word のコピーを返します。

**capwords(s)**

引数で与えられた文字列を split() で単語に分割し、capitalize() で各単語の先頭文字を大文字化し、それを join() で連結します。この関数は連続した空白文字を、スペース 1 つに置換し、先頭と末尾の空白を削除することに注意しましょう。

**expandtabs(s[, tabsize])**

タブを展開、すなわちタブを 1 つ以上のスペースに置換します。スペースの個数は現在のコラム位置とタブ幅によって決定されます。コラム位置は文字列中に改行が出現する度に、0 にリセットされます。この関数は、改行以外の非表示文字やエスケープシーケンスを理解しません。タブ幅のデフォルト値は 8 です。

**find(s, sub[, start[, end]])**

s[start:end] の中で、部分文字列 sub が完全に含まれるもののうち、最初に見つかった位置を s のインデックスで返す。見つからなかった場合は -1 を返す。start と end のデフォルト値、および、負数が与えられた場合の解釈はスライスと同じ。

**rfind(s, sub[, start[, end]])**

find() と同じですが、最後に見つかったもののインデックスを返します。

**index(s, sub[, start[, end]])**

find() と同じですが、部分文字列が見つからなかったときに ValueError を発生させます。

**rindex(s, sub[, start[, end]])**

rfind() と同じですが、部分文字列が見つからなかったときに ValueError を発生させます。

**count(s, sub[, start[, end]])**

s[start:end] の中で、部分文字列 sub が (重ならず) に出現する回数を返します。start と end のデフォルト値、および、負数が与えられた場合の解釈はスライスと同じです。

**lower(s)**

大文字を小文字に変換した、s のコピーを返します。

**maketrans(from, to)**

translate() と regex.compile() に渡して、from に含まれる各文字を、to の同じ位置にある文字に置換するような変換テーブルを返します。from と to は同じ長さでなければいけません。

警告: lowercase と uppercase から得られる文字列を、引数にしてはいけません。ロケールによっては、これらは長さが異なる。大文字小文字の変換には、常に lower() と upper() を使いましょう。

**split(s[, sep[, maxsplit]])**

文字列 s に含まれる単語を、リストにして返します。第 2 引数 sep が指定されなかった場合と None が指定された場合には、各単語は任意の空白 (スペース、タブ、改行、リターン、改頁) で区切られます。第 2 引数 sep が指定され、かつ、それが None でなければ、その文字列が単語の区切りになる。返されるリストの要素数は、文字列内で重複しない区切り文字列の数より、1 個多くなります。第 3 引数 maxsplit のデフォルト値は 0。この引数が 0 以外の場合、多くとも maxsplit 箇所で区切り、残りの文字列は、リストの最後の要素として返されます (したがって、リストは多くとも maxsplit+1 個の要素を持ちます)。

**splitfields(s[, sep[, maxsplit]])**

この関数は `split()` と同じようにふるまいます (昔は `split()` は引数 1 つの場合でのみ使われ、`splitfields()` は引数 2 つの場合でのみ使われていました)。

`join(words[, sep])`

単語のリストまたはタプルの要素を、`sep` を間に入れて連結します。`sep` のデフォルト値はスペース 1 個。‘`string.join(string.split(s, sep), sep)`’ は、常に `s` です。

`joinfields(words[, sep])`

この関数は `join()` と同じふるまいをします (昔は、`join()` は引数 1 つの場合でのみ使われ、`joinfields()` は引数 2 つの場合でのみ使われていました)。オブジェクトには `joinfields()` メソッドがないことに注意してください。`join()` メソッドを使いましょう。

`lstrip(s[, chars])`

先頭の文字を取り除いた文字列のコピーを返します。`chars` が与えられない場合や、`chars` の値が `None` の場合、先頭の空白を取り除きます。`chars` が与えられ、その値が `None` でない場合、このメソッドを呼んだ文字列から、`chars` に含まれる文字を取り除きます。`chars` は文字列でなくてはなりません。2.2.3 で変更された仕様: `chars` パラメータを追加。初期の 2.2 バージョンでは、`versionschars` パラメータを渡せない

`rstrip(s[, chars])`

末尾の文字を取り除いた文字列のコピーを返します。`chars` が与えられない場合や、`chars` の値が `None` の場合、末尾の空白を取り除きます。`chars` が与えられ、その値が `None` でない場合、このメソッドを呼んだ文字列から、`chars` に含まれる文字を取り除きます。`chars` は文字列でなくてはなりません。2.2.3 で変更された仕様: `chars` パラメータを追加。初期の 2.2 バージョンでは、`versionschars` パラメータを渡せない

`strip(s[, chars])`

先頭と末尾の文字を取り除いた文字列のコピーを返します。`chars` が与えられない場合や、`chars` の値が `None` の場合、先頭と末尾の空白を取り除きます。`chars` が与えられ、その値が `None` でない場合、このメソッドを呼んだ文字列から、`chars` に含まれる文字を取り除きます。`chars` は文字列でなくてはなりません。2.2.3 で変更された仕様: `chars` パラメータを追加。初期の 2.2 バージョンでは、`versionschars` パラメータを渡せない

`swapcase(s)`

`s` の大文字と小文字を入れ替えたものを返します。

`translate(s, table[, deletechars])`

`s` の中から、(もし指定されていれば) `deletechars` に含まれる文字を削除し、`table` を使って文字を入れ替えたものを返します。`table` は 256 文字からなる文字列で、各文字の位置が、置換前の文字コードに対応します。

`upper(s)`

`s` に含まれる小文字を大文字に置換して返します。

`ljust(s, width)`

`rjust(s, width)`

`center(s, width)`

これらの関数は、与えられた文字列幅の中で、それぞれ左寄せ、右寄せ、中央寄せします。これらが返す文字列は、少なくとも `width` 文字分の長さがあり、`s` の右側、左側、または両側がスペースで埋まっている。文字列が切られることはない。

`zfill(s, width)`

数値文字列の左側を、与えられた幅に達するまで 0 で埋めます。符号で始まる場合も、正しく処理されます。

`replace(str, old, new[, maxreplace])`

`s` 内の部分文字列 `old` を全て `new` に置換したものを返します。`maxreplace` が指定された場合、最初に見つかった `maxreplace` 個分だけ置換します。

## 4.2 re — 正規表現操作

このモジュールは、Perl で見られるものと同様な正規表現マッチング操作を提供します。正規表現のパターン文字列にヌルバイトを含むことはできませんが、`\number` 記法を使って、ヌルバイトを指定することはできます。パターンおよび検索される文字列の両方で、8 ビット文字列と同じ様に Unicode 文字列を使うことができます。`re` モジュールはいつでも利用できます。

正規表現では、特殊な形式を表したり、特殊な文字の特別な意味を呼び出さずにその特殊な文字を使うことができるようにするために、バックスラッシュ文字 (`\`) を使います。このことは、Python の、文字列リテラルの同一文字は同一目的で使用するということと矛盾します；例えば、文字通りのバックスラッシュとマッチするには、パターン文字列として `'\\'` と書かなければなりません、というのは、正規表現は `'\'` でなければならない、各バックスラッシュは、正規な Python 文字列リテラル内では `'\\'` と表現しなければならないからです。

これは、正規表現パターンに Python の raw string 記法を使うことで解決されます；`'r'` を前に付けた文字列リテラル内では、バックスラッシュは、特別な風には全く処理されません。ですから、`r"\n"` は `'\n'` と `'n'` を含む 2 文字の文字列であり、一方 `"\n"` は、改行を含む一文字文字列です。Python コード中では、パターンは、ふつう、この raw string 記法を使って表現されます。

参考資料:

*Mastering Regular Expressions* 詳説 正規表現

Jeffrey Friedl 著、O'Reilly 刊の正規表現に関する本です。この本の第 2 版では Python については触れていませんが、良い正規表現パターンの書き方を非常に詳細に説明しています。

### 4.2.1 正規表現のシンタックス

正規表現 (すなわち RE) は、それとマッチする文字列の集合を指定します；このモジュールの関数によって、特別な文字列が与えられた正規表現とマッチするかどうか (あるいは与えられた正規表現が特別な文字列とマッチするかどうか、これは結局同じことになりますが) を検査できます。

正規表現は、連結して新しい正規表現を作ることができます；もし `A` と `B` が、ともに正規表現であれば、`AB` も正規表現です。一般的には、もし文字列 `p` が `A` とマッチし、別の文字列 `q` が `B` とマッチすれば、文字列 `pq` は `AB` にマッチします。ただし、この状況が成り立つのは `A` や `B` が優先度の低い演算や `A` と `B` との間の境界条件、あるいは番号付けされたグループ参照を含まない場合だけです。かくして、ここで述べるような、より簡単でプリミティブな正規表現から、複雑な正規表現を容易に構築することができます、正規表現に関する理論と実装の詳細については、上記の Friedl 本か、コンパイラ構造に関する大抵の教科書を調べて下さい。

以下で正規表現の形式に関する簡単な説明をしておきます。より詳細な情報やよりやさしい説明に関しては、<http://www.python.org/doc/howto/> からアクセスできる正規表現ハウツウを調べて下さい。

正規表現には、特殊文字と通常文字の両方を含めることができます。`'A'`、`'a'`、あるいは `'0'` のような殆どの通常文字は、もっとも簡単な正規表現です；それらは、単純にそれら自身とマッチします。通常文字を連結することもできますので、`'last'` は文字列 `'last'` とマッチします。(このセクションの残りでは、RE を普通引用符なしで「この特殊な形式」で書き、マッチされる文字列は、`'` 単一引用符内 `'` に書きます。)

`'|'` や `'()'` のようないくつかの文字は特殊文字です。特殊文字は通常文字クラスを表すか、あるいは通常文字に関する正規表現がどのように解釈されるかに影響します。

特殊文字は：



- ‘.’ (ドット。) デフォルトのモードでは、これは改行以外の任意の文字とマッチします。もし DOTALL フラグが指定されていれば、これは、改行も含むすべての文字とマッチします。
- ‘^’ (キャレット。) 文字列の先頭とマッチし、MULTILINE モードでは、各改行の直後とマッチします。
- ‘\$’ 文字列の末尾、あるいは文字列の末尾の改行の直前とマッチします。[foo] は、‘foo’ と ‘foobar’ の両方とマッチし、一方、正規表現 [foo\$] は、‘foo’ だけとマッチします。もっと興味深いことは、‘foo1\nfoo2\n’ で [foo.\$] を検索すると、普通は ‘foo2’ とマッチするのですが、MULTILINE モードでは ‘foo1’ とマッチします。
- ‘\*’ 結果の RE は、前にある RE を、0 回以上できるだけ多く繰り返したものとマッチするようになります。[ab\*] は、‘a’、‘ab’、あるいは ‘a’ に任意個数の ‘b’ を続けたものとマッチします。
- ‘+’ 結果の RE は、前にある RE を、1 回以上繰り返したものとマッチするようになります。[ab+] は、‘a’ に非ゼロ個の ‘b’ が続いたものとマッチします；これは、‘a’ だけとはマッチしません。
- ‘?’ 結果の RE は、前にある RE を、0 回か 1 回繰り返したものとマッチするようになります。[ab?] は、‘a’ あるいは ‘ab’ とマッチします。
- \*?, +?, ?? ‘\*’、‘+’、および ‘?’ 修飾子は、すべて 欲張り (*greedy*) です；それらはできるだけ多くのテキストとマッチします。時にはこの動作が望ましくない場合があります；もし RE ‘<. \*>’ を、‘<H1>title</H1>’ とマッチさせると、これは、全文字列とマッチし、‘<H1>’ だけとはマッチしません。‘?’ を修飾子の後に追加すると、控え目な (*non-greedy*) あるいは 最小 風のマッチをするようになります；できるだけ 少ない 文字とマッチします。前の式で ‘.\*?’ を使うと、‘<H1>’ だけとマッチします。
- {*m*} 前にある RE の *m* 回の正確なコピーとマッチすべきであることを指定します；マッチ回数が少なければ、RE 全体ではマッチしません。例えば、[a{6}] は、正確に 6 個の ‘a’ 文字とマッチしますが、5 個ではマッチしません。
- {*m*,*n*} 結果の RE は、前にある RE を、*m* 回から *n* 回まで繰り返したもので、できるだけ多く繰り返したものとマッチするように、マッチします。例えば、[a{3,5}] は、3 個から 5 個の ‘a’ 文字とマッチします。*m* を省略するとマッチ回数の下限として 0 を指定した事になり、*n* を省略することは、上限が無限であることを指定します；[a{4,}b] は aaaab や、千個の ‘a’ 文字に b が続いたものとマッチしますが、aaab とはマッチしません。コンマは省略できません、そうでないと修飾子が上で述べた形式と混同されてしまうからです。
- {*m*,*n*}? 結果の RE は、前にある RE の *m* 回から *n* 回まで繰り返したもので、できるだけ少なく繰り返したものとマッチするように、マッチします。これは、前の修飾子の控え目バージョンです。例えば、6 文字 文字列 ‘aaaaaa’ では、[a{3,5}] は、5 個の ‘a’ 文字とマッチしますが、[a{3,5}?] は 3 個の文字とマッチするだけです。
- ‘\’ 特殊文字をエスケープする (‘\*’ や ‘?’ 等のような文字とのマッチをできるようにする) か、あるいは、特殊シーケンスの合図です；特殊シーケンスは後で議論します。
- もしパターンを表現するのに raw string を使用していないのであれば、Python も、バックスラッシュを文字列リテラルでのエスケープシーケンスとして使っていることを覚えていて下さい；もしエスケープシーケンスを Python の構文解析器が認識して処理しなければ、そのバックスラッシュとそれに続く文字は、結果の文字列にそのまま含まれます。しかし、もし Python が結果のシーケンスを認識するのであれば、バックスラッシュを 2 回 繰り返さなければいけません。このことは複雑で理解しにくいので、最も簡単な表現以外は、すべて raw string を使うことをぜひ勧めます。



[ ] 文字の集合を指定するのに使用します。文字は個々にリストするか、文字の範囲を、2つの文字と '-' でそれらを分離して指定することができます。特殊文字は集合内では有効ではありません。例えば、[akm\$] は、文字 'a'、'k'、'm'、あるいは '\$' のどれかとマッチします；[a-z] は、任意の小文字と、[a-zA-Z0-9] は、任意の文字や数字とマッチします。(以下で定義する) \w や \s のような文字クラスも、範囲に含めることができます。もし文字集合に ']' や '-' を含めたいのなら、その前にバックスラッシュを付けるか、それを最初の文字として指定します。たとえば、パターン '[ ]]' は ']' とマッチします。

範囲内にない文字とは、その集合の補集合をとることでマッチすることができます。これは、集合の最初の文字として '^' を含めることで表すことができます；他の場所にある '^' は、単純に '^' 文字とマッチするだけです。例えば、[^5] は、'5' 以外の任意の文字とマッチし、[^^] は、'^' 以外の任意の文字とマッチします。

'|' A|B は、ここで A と B は任意の RE ですが、A か B のどちらかとマッチする正規表現を作成します。任意個数の RE を、こういう風に '|' で分離することができます。これはグループ (以下参照) 内部でも同様に使えます。検査対象文字列をスキャンする中で、'|' で分離された RE は左から右への順に検査されます。一つでも完全にマッチしたパターンがあれば、そのパターン枝が受理されます。このことは、もし A がマッチすれば、たとえ B によるマッチが全体としてより長いマッチになったとしても、B を決して検査しないことを意味します。言いかえると、'|' 演算子は決して貪欲 (greedy) ではありません。文字通りの '|' とマッチするには、\| を使うか、あるいはそれを '[|]' のように文字クラス内に入れます。

(...) 丸括弧の中にどのような正規表現があってもマッチし、またグループの先頭と末尾を表します；グループの中身は、マッチが実行された後に検索され、後述する \number 特殊シーケンス付きの文字列内で、後でマッチされます。文字通りの '(' や ')' とマッチするには、\( あるいは \) を使うか、それらを文字クラス内に入れます：'[ ( ) ]'。

(?... ) これは拡張記法です ('(' に続く '?' は他には意味がありません)。「?' の後の最初の文字が、この構造の意味とこれ以上のシンタックスがどのようなものであるかを決定します。拡張記法は普通新しいグループを作成しません；(?P<name>...) がこの規則の唯一の例外です。以下に現在サポートされている拡張記法を示します。

(?iLmsux) (集合 'i'、'L'、'm'、's'、'u'、'x' から 1 文字以上)。グループは空文字列ともマッチします；文字は、正規表現全体の対応するフラグ (re.I、re.L、re.M、re.S、re.U、re.X) を設定します。これはもし flag 引数を compile() 関数に渡さずに、そのフラグを正規表現の一部として含めたいならば役に立ちます。

「(?x)」フラグは、式が構文解析される方法を変更することに注意して下さい。これは式文字列内の最初か、あるいは 1 つ以上の空白文字の後で使うべきです。もしこのフラグの前に非空白文字があると、その結果は未定義です。

(?:...) 正規表現の丸括弧の非グループ化バージョンです。どのような正規表現が丸括弧内にあってもマッチしますが、グループによってマッチされたサブ文字列は、マッチを実行したあと検索されることも、あるいは後でパターンで参照されることもできません。

(?P<name>...) 正規表現の丸括弧と同様ですが、グループによってマッチされたサブ文字列は、記号グループ名 name を介してアクセスできます。グループ名は、正しい Python 識別子でなければならず、各グループ名は、正規表現内で一度だけ定義されなければなりません。記号グループは、グループに名前が付けられていない場合のように、番号付けされたグループでもあります。そこで上の例で 'id' という名前がついたグループは、番号グループ 1 として参照することもできます。

たとえば、もしパターンが「(?P<id>[a-zA-Z\_]\w\*)」であれば、このグループは、マッチオブジェクトのメソッドへの引数に、`m.group('id')` あるいは `m.end('id')` のような名前前で、またパターンテキスト内 (例えば、「(?P=id)») や置換テキスト内 (`\g<id>` のように) で名前で参照することができます。

(?P=name) 前に *name* と名前付けされたグループにマッチした、いかなるテキストにもマッチします。

(?#...) コメントです；括弧の内容は単純に無視されます。

(?=...) もし「...」が次に続くものとマッチすればマッチしますが、文字列をまったく消費しません。これは先読みアサーション (lookahead assertion) と呼ばれます。例えば、「Isaac (?=Asimov)」は、「Isaac」に「Asimov」が続く場合だけ、「Isaac」とマッチします。

(?!...) もし「...」が次に続くものとマッチしなければマッチします。これは否定先読みアサーション (negative lookahead assertion) です。例えば、「Isaac (?!Asimov)」は、「Isaac」に「Asimov」が続かない場合のみマッチします。

(?<=...) もし文字列内の現在位置の前に、現在位置で終わる「...」とのマッチがあれば、マッチします。これは肯定後読みアサーション (*positive lookbehind assertion*) と呼ばれます。「(?<=abc)def」は、「abcdef」にマッチを見つけます、というのは後読みが3文字をバックアップして、含まれているパターンとマッチするかどうか検査するからです。含まれるパターンは、固定長の文字列にのみマッチしなければなりません、ということは、「abc」や「a|b」は許されますが、「a\*」や「a{3,4}」は許されないことを意味します。肯定後読みアサーションで始まるパターンは、検索される文字列の先頭とは決してマッチしないことに注意して下さい；多分、`match()` 関数よりは `search()` 関数を使いたいでしょう：

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

この例ではハイフンに続く単語を探します：

```
>>> m = re.search('(?<=)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```

(?<!...) もし文字列内の現在位置の前に「...」とのマッチがないならば、マッチします。これは否定後読みアサーション (*negative lookbehind assertion*) と呼ばれます。肯定後読みアサーションと同様に、含まれるパターンは固定長さの文字列だけにマッチしなければいけません。否定後読みアサーションで始まるパターンは、検索される文字列の先頭とマッチすることができます。

特殊シーケンスは「\」と以下のリストにある文字から構成されます。もしリストにあるのが通常文字でないならば、結果の RE は2番目の文字とマッチします。例えば、「\\$」は文字「\$」とマッチします。

`\number` 同じ番号のグループの中身とマッチします。グループは1から始まる番号をつけられます。例えば、「(.+) \1」は、「the the」あるいは「55 55」とマッチしますが、「the end」とはマッチしません (グループの後のスペースに注意して下さい)。この特殊シーケンスは最初の99グループのうちの一つとマッチするのに使うことができます。もし *number* の最初の桁が0である、すなわち *number* が3桁の8進数であれば、それはグループのマッチとは解釈さ

れず、8進数値 *number* を持つ文字として解釈されます。文字クラスの '[' と ']' の中の数値エスケープは、文字として扱われます。

`\A` 文字列の先頭だけにマッチします。

`\b` 空文字列とマッチしますが、単語の先頭か末尾の時だけです。単語は英数字あるいは下線文字の並んだものとして定義されていますので、単語の末尾は空白あるいは非英数字、非下線文字によって表されます。`\b` は、`\w` と `\W` の間の境界として定義されているので、英数字であると思なされる文字の正確な集合は、UNICODE と LOCALE フラグの値に依存することに注意して下さい。文字の範囲の中では、`\b` は、Python の文字列リテラルと互換性を持たせるために、後退 (backspace) 文字を表します。

`\B` 空文字列とマッチしますが、それが単語の先頭あるいは末尾にない時だけです。これは `\b` のちょうど反対ですので、LOCALE と UNICODE の設定にも影響されます。

`\d` 任意の十進数とマッチします；これは集合 `[0-9]` と同じ意味です。

`\D` 任意の非数字文字とマッチします；これは集合 `[^0-9]` と同じ意味です。

`\s` 任意の空白文字とマッチします；これは集合 `[\t\n\r\f\v]` と同じ意味です。

`\S` 任意の非空白文字とマッチします；これは集合 `[^\t\n\r\f\v]` と同じ意味です。

`\w` LOCALE と UNICODE フラグが指定されていない時は、任意の英数文字および下線とマッチします；これは、集合 `[a-zA-Z0-9_]` と同じ意味です。LOCALE が設定されていると、集合 `[0-9_]` プラス 現在のロケール用に英数字として定義されている任意の文字とマッチします。もし UNICODE が設定されていれば、文字 `[0-9_]` プラス Unicode 文字特性データベースで英数字として分類されているものとマッチします。

`\W` LOCALE と UNICODE フラグが指定されていない時、任意の非英数文字とマッチします；これは集合 `[^a-zA-Z0-9_]` と同じ意味です。LOCALE が指定されていると、集合 `[0-9_]` になく、現在のロケールで英数字として定義されていない任意の文字とマッチします。もし UNICODE がセットされていれば、これは `[0-9_]` および Unicode 文字特性データベースで英数字として表されている文字以外のものとマッチします。

`\Z` 文字列の末尾とのみマッチします。

Python 文字列リテラルによってサポートされている標準エスケープのほとんども、正規表現パーザに認識されます：

<code>\a</code>	<code>\b</code>	<code>\f</code>	<code>\n</code>
<code>\r</code>	<code>\t</code>	<code>\v</code>	<code>\x</code>
<code>\\</code>			

8進エスケープは制限された形式で含まれています：もし第1桁が0であるか、もし8進3桁であれば、それは8進エスケープとみなされます。そうでなければ、それはグループ参照です。

#### 4.2.2 マッチング vs 検索

Python は、正規表現に基づく、2つの異なるプリミティブな操作を提供しています：マッチと検索です。もしあなたが Perl の記号に慣れているのであれば、検索操作があなたの求めるものです。 `search()` 関数と、コンパイルされた正規表現オブジェクトでの対応するメソッドを見て下さい。

マッチは、‘^’ で始まる正規表現を使うと、検索とは異なるかもしれないことに注意して下さい：‘^’ は文字列の先頭でのみ、あるいは MULTILINE モードでは改行の直後ともマッチします。“マッチ” 操作は、もしそのパターンが、モードに拘らず文字列の先頭とマッチするか、あるいは改行がその前にあるかどうかにかかわらず、省略可能な *pos* 引数によって与えられる先頭位置でマッチする場合のみ成功します。

```
re.compile("a").match("ba", 1)           # 成功
re.compile("^a").search("ba", 1)          # 失敗; 'a' は先頭でない
re.compile("^a").search("\na", 1)         # 失敗; 'a' は先頭でない
re.compile("^a", re.M).search("\na", 1)   # 成功
re.compile("^a", re.M).search("ba", 1)    # 失敗; \n が前にない
```

### 4.2.3 モジュール コンテンツ

このモジュールは以下の関数および定数、それに例外を定義しています：

**compile(*pattern*[, *flags*])**

正規表現パターンを正規表現オブジェクトにコンパイルします。このオブジェクトは、以下で述べる `match()` と `search()` メソッドを使って、マッチングに使うことができます。

式の動作は、*flags* の値を指定することで加減することができます。値は以下の変数を、ビットごとの OR ( | 演算子) を使って組み合わせることができます。

シーケンス

```
prog = re.compile(pat)
result = prog.match(str)
```

は、

```
result = re.match(pat, str)
```

と同じ意味ですが、`compile()` を使うバージョンの方が、その式を一つのプログラムで何回も使う時にはより効率的です。

**I**

**IGNORECASE**

大文字・小文字を区別しないマッチングを実行します；「[A-Z]」のような式は、小文字にもマッチします。これは現在のロケールには影響されません。

**L**

**LOCALE**

「\w」 「\W」 「\b」 および 「\B」 を、現在のロケールに従わせます。

**M**

**MULTILINE**

指定されると、パターン文字 ‘^’ は、文字列の先頭および各行の先頭 (各改行の直後) とマッチします；そしてパターン文字 ‘\$’ は文字列の末尾および各行の末尾 (改行の直前) とマッチします。デフォルトでは、‘^’ は、文字列の先頭とだけマッチし、‘\$’ は、文字列の末尾および文字列の末尾の改行の直前 (がもしあれば) とマッチします。

**S**

**DOTALL**

特殊文字 ‘.’ を、改行を含む任意の文字と、とにかくマッチさせます；このフラグがなければ、‘.’ は、改行 以外の任意の文字とマッチします。

## U

### UNICODE

`\w`, `\W`, `\b` および `\B` を、Unicode 文字特性データベースに従わせます。2.0 で追加された仕様です。

## X

### VERBOSE

このフラグによって、より見やすく正規表現を書くことができます。パターン内の空白は、文字クラス内にあるか、エスケープされていないバックスラッシュが前にある時以外は無視されます。また、行に、文字クラス内にもなく、エスケープされていないバックスラッシュが前にもない '#' がある時は、そのような '#' の左端からその行の末尾までが無視されます。

**search**(*pattern*, *string*[, *flags*])

*string* 全体を走査して、正規表現 *pattern* がマッチを発生する位置を探して、対応する MatchObject インスタンスを返します。もし文字列内に、そのパターンとマッチする位置がないならば、None を返します；これは、文字列内のある点で長さゼロのマッチを探すこととは異なることに注意して下さい。

**match**(*pattern*, *string*[, *flags*])

もし *string* の先頭で 0 個以上の文字が正規表現 *pattern* とマッチすれば、対応する MatchObject インスタンスを返します。もし文字列がパターンとマッチしなければ、None を返します；これは長さゼロのマッチとは異なることに注意して下さい。

注意: もし *string* のどこかにマッチを位置付けたいのであれば、代わりに `search()` を使って下さい。

**split**(*pattern*, *string*[, *maxsplit* = 0])

*string* を、*pattern* があるたびに分割します。もし括弧のキャプチャが *pattern* で使われていれば、パターン内のすべてのグループのテキストも結果のリストの一部として返されます。*maxsplit* がゼロでなければ、高々 *maxsplit* 個の分割が発生し、文字列の残りは、リストの最終要素として返されます。(非互換性ノート: オリジナルの Python 1.5 リリースでは、*maxsplit* は無視されていました。これはその後のリリースでは修正されました。)

```
>>> re.split('\W+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split('(\W+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', ', ', '']
>>> re.split('\W+', 'Words, words, words.', 1)
['Words', 'words, words.']
```

この関数は、古い `regsub.split()` と `regsub.splitx()` の機能を結合して拡張したものです。

**findall**(*pattern*, *string*)

*string* 内の *pattern* の重複しないマッチのすべてのリストを返します。もしパターンにグループが 1 つ以上あれば、グループのリストを返します；これは、もしパターンにグループが 1 つ以上あれば、タプルのリストとなります。他にマッチがなければ、空のマッチも結果に入ります。1.5.2 で追加された仕様です。

**finditer**(*pattern*, *string*)

*string* 内の RE *pattern* の重複しないマッチのすべてのイテレータを返します。各マッチごとに、イテレータはマッチオブジェクトを返します。他にマッチがなければ、空のマッチも結果に入ります。2.2 で追加された仕様です。

**sub**(*pattern*, *repl*, *string*[, *count*])

*string* 内で、*pattern* と重複しないマッチの内、一番左にあるものを置換 *repl* で置換して得られた文字列を返します。もしパターンが見つからなければ、*string* を変更せずに返します。*repl* は文字列でも関



数でも構いません；もしそれが文字列であれば、それにある任意のバックスラッシュエスケープは処理されます。すなわち、`'\n'` は単一の改行文字に変換され、`'\r'` は、行送りコードに変換されます、等々。`'\j'` のような未知のエスケープはそのままにされます。`'\6'` のような後方参照 (backreference) は、パターンのグループ 6 とマッチしたサブ文字列で置換されます。例えば：

```
>>> re.sub(r'def\s+([a-zA-Z_][a-zA-Z_0-9]*)\s*(\s*\n):',
...        r'static PyObject*\numpy_\1(void)\n{ ',
...        'def myfunc():')
'static PyObject*\numpy_myfunc(void)\n{'
```

もし *repl* が関数であれば、重複しない *pattern* が発生するたびにその関数が呼ばれます。この関数は一つのマッチオブジェクト引数を取り、置換文字列を返します。例えば：

```
>>> def dashrepl(matchobj):
....     if matchobj.group(0) == '-': return ' '
....     else: return '-'
>>> re.sub('-{1,2}', dashrepl, 'pro----gram-files')
'pro--gram files'
```

パターンは、文字列でも RE でも構いません；もし正規表現フラグを指定する必要があるれば、RE オブジェクトを使うか、パターンに埋込み修飾子を使わなければなりません；たとえば、`'sub("(?i)b+", "x", "bbbb BBBB")'` は `'x x'` を返します。

省略可能な引数 *count* は、置換されるパターンの出現回数の最大値です；*count* は非負の整数でなければなりません。もし省略されるかゼロであれば、出現したものがすべて置換されます。パターンのマッチが空であれば、以前のマッチと隣合わせでない時だけ置換されますので、`'sub('x*', '-', 'abc')'` は `'-a-b-c-'` を返します。

上で述べた文字エスケープや後方参照の他に、`'\g<name>'` は、`'(?P<name>...)'` のシンタックスで定義されているように、`'name'` という名前のグループとマッチしたサブ文字列を使います。`'\g<number>'` は対応するグループ番号を使います；それゆえ `'\g<2>'` は `'\2'` と同じ意味ですが、`'\g<2>0'` のような置換でもあいまいではありません。`'\20'` は、グループ 20 への参照として解釈されますが、グループ 2 にリテラル文字 `'0'` が続いたものへの参照としては解釈されません。後方参照 `'\g<0>'` は、RE とマッチするサブ文字列全体を置き換えます。

**subn**(*pattern*, *repl*, *string*[, *count* ])

`sub()` と同じ操作を行います、タプル (*new\_string*, *number\_of\_subs\_made*) を返します。

**escape**(*string*)

バックスラッシュにすべての非英数字をつけた *string* を返します；これはもし、その中に正規表現のメタ文字を持つかもしれない任意のリテラル文字列とマッチしたいとき、役に立ちます。

**exception error**

ここでの関数の一つに渡された文字列が、正しい正規表現ではない時 (例えば、その括弧が対になっていなかった)、あるいはコンパイルやマッチングの間になんらかのエラーが発生したとき、発生する例外です。たとえ文字列がパターンとマッチしなくても、決してエラーではありません。

#### 4.2.4 正規表現オブジェクト

コンパイルされた正規表現オブジェクトは、以下のメソッドと属性をサポートします：

**match**(*string*[, *pos*[, *endpos* ]])

もし *string* の先頭の 0 個以上の文字がこの正規表現とマッチすれば、対応する `MatchObject` インスタンスを返します。もし文字列がパターンとマッチしなければ、`None` を返します；これは長さゼロのマッチとは異なることに注意して下さい。



注意: もしマッチを *string* のどこかに位置付けたいければ、代わりに `search()` を使って下さい。

省略可能な第2のパラメータ *pos* は、文字列内の検索を始めるインデックスを与えます; デフォルトでは0です。これは、文字列のスライシングと完全に同じ意味だというわけではありません; `'^'` パターン文字は、文字列の実際の実の先頭と改行の直後とマッチしますが、それが必ずしも検索が開始するインデックスであるわけではないからです。

省略可能なパラメータ *endpos* は、どこまで文字列が検索されるかを制限します; あたかもその文字列が *endpos* 文字長であるかのようにしますので、*pos* から *endpos* - 1 までの文字が、マッチのために検索されます。もし *endpos* が *pos* より小さければ、マッチは見つかりませんが、そうでなくて、もし *rx* がコンパイルされた正規表現オブジェクトであれば、`rx.match(string, 0, 50)` は `rx.match(string[:50], 0)` と同じ意味になります。

**search**(*string*[, *pos*[, *endpos*]])

*string* 全体を走査して、この正規表現がマッチする位置を探して、対応する `MatchObject` インスタンスを返します。もし文字列内にパターンとマッチする位置がないならば、`None` を返します; これは文字列内のある点で長さゼロのマッチを探すこととは異なることに注意して下さい。

省略可能な *pos* と *endpos* パラメータは、`match()` メソッドのものと同じ意味を持ちます。

**split**(*string*[, *maxsplit* = 0])

`split()` 関数と同様で、コンパイルしたパターンを使います。

**findall**(*string*)

`findall()` 関数と同様で、コンパイルしたパターンを使います。

**finditer**(*string*)

`finditer()` 関数と同様で、コンパイルしたパターンを使います。

**sub**(*repl*, *string*[, *count* = 0])

`sub()` 関数と同様で、コンパイルしたパターンを使います。

**subn**(*repl*, *string*[, *count* = 0])

`subn()` 関数と同様で、コンパイルしたパターンを使います。

**flags**

*flags* 引数は、RE オブジェクトがコンパイルされたとき使われ、もし *flags* が何も提供されなければ0です。

**groupindex**

`'(?P<id>)'` で定義された任意の記号グループ名の、グループ番号への辞書マッピングです。もし記号グループがパターン内で何も使われていなければ、辞書は空です。

**pattern**

RE オブジェクトがそれからコンパイルされたパターン文字列です。

## 4.2.5 MatchObject オブジェクト

`MatchObject` インスタンスは以下のメソッドと属性をサポートします:

**expand**(*template*)

テンプレート文字列 *template* に、`sub()` メソッドがするようなバックスラッシュ置換をして得られる文字列を返します。`'\n'` のようなエスケープは適当な文字に変換され、数値の後方参照 (`'\1'`、`'\2'`) と名前付きの後方参照 (`'\g<1>'`、`'\g<name>'`) は、対応するグループの内容で置き換えられます。

**group**([*group1*, ...])

マッチした1個以上のサブグループを返します。もし引数で一つであれば、その結果は一つの文字列です; 複数の引数があれば、その結果は、引数ごとに一項目を持つタプルです。引数がなければ、

`group1` はデフォルトでゼロです (マッチしたもののすべてが返されます)。もし `groupN` 引数がゼロであれば、対応する戻り値は、マッチする文字列全体です；もしそれが範囲 [1..99] 内であれば、それは、対応する丸括弧つきグループとマッチする文字列です。もしグループ番号が負であるか、あるいはパターンで定義されたグループの数より大きければ、`IndexError` 例外が発生します。もしグループがマッチしなかったパターンの一部に含まれていれば、対応する結果は `None` です。もしグループが、複数回マッチしたパターンの一部に含まれていれば、最後のマッチが返されます。

もし正規表現が「`(?P<name>...)`」シンタックスを使うならば、`groupN` 引数は、それらのグループ名によってグループを識別する文字列であっても構いません。もし文字列引数がパターンのグループ名として使われていないものであれば、`IndexError` 例外が発生します。

適度に複雑な例題：

```
m = re.match(r"(?P<int>\d+)\.(\d*)", '3.14')
```

このマッチを実行したあとでは、`m.group(1)` は `m.group('int')` と同じく、`'3'` であり、そして `m.group(2)` は `'14'` です。

**groups([default])**

1 からどれだけ多くであろうがパターン内にあるグループ数までの、マッチの、すべてのサブグループを含むタプルを返します。`default` 引数は、マッチに加わらなかったグループ用に使われます；それはデフォルトでは `None` です。(非互換性ノート：オリジナルの Python 1.5 リリースでは、たとえばタプルが一要素長であっても、その代わりに文字列を返すことはありません。(1.5.1 以降の) 後のバージョンでは、そのような場合には、シングルトンタプルが返されます。)

**groupdict([default])**

すべての名前付きのサブグループを含む、マッチの、サブグループ名でキー付けされた辞書を返します。`default` 引数はマッチに加わらなかったグループ用に使われます；それはデフォルトでは `None` です。

**start([group])**

**end([group])**

`group` とマッチしたサブ文字列の先頭と末尾のインデックスを返します；`group` は、デフォルトでは (マッチしたサブ文字列全体を意味する) ゼロです。`group` が存在してもマッチに寄与しなかった場合は、`-1` を返します。マッチオブジェクト `m` およびマッチに寄与しなかったグループ `g` があって、グループ `g` とマッチしたサブ文字列 (`m.group(g)` と同じ意味ですが) は、

```
m.string[m.start(g):m.end(g)]
```

です。もし `group` がヌル文字列とマッチすれば、`m.start(group)` が `m.end(group)` と等しくなることに注意して下さい。例えば、`m = re.search('b(c?)', 'cba')` の後では、`m.start(0)` は 1 で、`m.end(0)` は 2 であり、`m.start(1)` と `m.end(1)` はともに 2 であり、`m.start(2)` は `IndexError` 例外が発生します。

**span([group])**

MatchObject `m` については、2-タプル (`m.start(group)`、`m.end(group)`) を返します。もし `group` がマッチに寄与しなかったら、これは `(-1, -1)` です。また `group` はデフォルトでゼロです。

**pos**

RegexObject の `search()` あるいは `match()` メソッドに渡された `pos` の値です。これは RE エンジンがマッチを探し始める位置の文字列のインデックスです。

**endpos**

RegexObject の `search()` あるいは `match()` メソッドに渡された `endpos` の値です。これは RE

エンジンがそれ以上は進まない位置の文字列のインデックスです。

**lastindex**

最後にマッチした取り込みグループの整数インデックスです。もしどのグループも全くマッチしなければ `None` です。

**lastgroup**

最後にマッチした取り込みグループの名前です。もしグループに名前がないか、あるいはどのグループも全くマッチしなければ `None` です。

**re**

その `match()` あるいは `search()` メソッドが、この `MatchObject` インスタンスを生成した正規表現オブジェクトです。

**string**

`match()` あるいは `search()` に渡された文字列です。

## 4.2.6 例

`scanf()` をシミュレートする

Python には現在のところ、`scanf()` に相当するものはありません。正規表現は、`scanf()` のフォーマット文字列よりも、一般的により強力であり、また冗長でもあります。以下の表に、`scanf()` のフォーマットトークンと正規表現の大体同等な対応付けを示します。

<code>scanf()</code> トークン	正規表現
<code>%c</code>	<code>「.」</code>
<code>%5c</code>	<code>「.{5}」</code>
<code>%d</code>	<code>「[-+]?\\d+」</code>
<code>%e, %E, %f, %g</code>	<code>「[-+]?((\\d+(\\.\\d*)?) \\d*\\.\\d+)([eE][-+]?\\d+)?」</code>
<code>%i</code>	<code>「[-+]?(0[xX][\\dA-Fa-f]+ 0[0-7]* \\d+)」</code>
<code>%o</code>	<code>「0[0-7]*」</code>
<code>%s</code>	<code>「\\S+」</code>
<code>%u</code>	<code>「\\d+」</code>
<code>%x, %X</code>	<code>「0[xX][\\dA-Fa-f]+」</code>

```
/usr/sbin/sendmail - 0 errors, 4 warnings
```

のような文字列からファイル名と数値を抽出するには、

```
%s - %d errors, %d warnings
```

のように `scanf()` フォーマットを使うでしょう。それと同等な正規表現は

```
(\\S+) - (\\d+) errors, (\\d+) warnings
```

再帰を避ける

エンジンに大量の再帰を要求するような正規表現を作成すると、`maximum recursion limit exceeded`(最大再帰制限を超過した) というメッセージを持つ `RuntimeError` 例外に出くわすかもしれません。たとえば、

```
>>> import re
>>> s = "Begin" + 1000 * 'a very long string' + 'end'
>>> re.match('Begin (\w| )*? end', s).end()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    File "/usr/local/lib/python2.3/sre.py", line 132, in match
      return _compile(pattern, flags).match(string)
RuntimeError: maximum recursion limit exceeded
```

再帰を避けるように正規表現を組みなおせることはよくあります。

Python 2.3 からは、再帰を避けるために「\*?」パターンの利用が特別扱いされるようになりました。したがって、上の正規表現は「Begin [a-zA-Z0-9\_ ]\*?end」に書き直すことで再帰を防ぐことができます。それ以上の恩恵として、そのような正規表現は、再帰的な同等のものよりもより速く動作します。

## 4.3 struct — 文字列データをパックされたバイナリデータとして解釈する

このモジュールは、Python の値と Python 上で文字列データとして表される C の構造体データとの間の変換を実現します。このモジュールでは、C 構造体のレイアウトおよび Python の値との間で行いたい変換をコンパクトに表現するために、フォーマット文字列を使います。このモジュールは特に、ファイルに保存されたり、ネットワーク接続を経由したバイナリデータを扱うときに使われます。

このモジュールは以下の例外と関数を定義しています：

### exception error

様々な状況で送出された例外です；引数は何が問題かを記述する文字列です。

### pack(*fmt*, *v1*, *v2*, ...)

値 *v1*, *v2*, ... が与えられたフォーマットで含まれる文字列データを返します。引数は指定したフォーマットが要求する型と正確に一致していなければなりません。

### unpack(*fmt*, *string*)

(おそらく `pack(fmt, ...)` でパックされた) 文字列データを、与えられた書式に従ってアンパックします。値が一つしかない場合を含め、結果はタプルで返されます。文字列データにはフォーマットが要求するだけのデータが正確に含まれていなければなりません (`len(string)` が `calcsizes(fmt)` と一致しなければなりません)。

### calcsizes(*fmt*)

与えられたフォーマットに対応する構造体のサイズ (すなわち文字列データのサイズ) を返します。

フォーマット文字 (format character) は以下の意味を持っています；C と Python の間の変換では、値は正確に以下に指定された型でなくてはなりません：

フォーマット	C での型	Python	備考
'x'	pad byte	no value	
'c'	char	長さ 1 の文字列	
'b'	signed char	整数型 (integer)	
'B'	unsigned char	整数型	
'h'	short	整数型	
'H'	unsigned short	整数型	
'i'	int	整数型	
'I'	unsigned int	long 整数型	
'l'	long	整数型	
'L'	unsigned long	long 整数型	
'q'	long long	long 整数型	(1)
'Q'	unsigned long long	long 整数型	(1)
'f'	float	浮動小数点型	
'd'	double	浮動小数点型	
's'	char[]	文字列	
'p'	char[]	文字列	
'P'	void *	整数型	

注意事項:

- (1) フォーマット文字 'q' および 'Q' は、プラットフォームの C コンパイラが C の long long 型、Windows では \_\_int64 をサポートする場合にのみ、プラットフォームネイティブの値との変換を行うモードだけで利用することができます。

2.2 で追加された仕様です。

フォーマット文字の前に整数をつけ、繰り返し回数 (count) を指定することができます。例えば、フォーマット文字列 '4h' は 'hhhh' と全く同じ意味です。

フォーマット文字間の空白文字は無視されます; count とフォーマット文字の間にはスペースを入れてはいけません。

フォーマット文字 's' では、count は文字列のサイズとして扱われます。他のフォーマット文字のように繰り返し回数ではありません; 例えば、'10c' が 10 個のキャラクタを表すのに対して、'10s' は 10 バイトの長さを持った 1 個の文字列です。文字列をバックする際には、指定した長さにフィットするように、必要に応じて切り詰められたりヌル文字で穴埋めされたりします。また特殊なケースとして、('0c' が 0 個のキャラクタを表すのに対して) '0s' は 1 個の空文字列を意味します。

フォーマット文字 'p' は "Pascal 文字列 (pascal string)" をコードします。Pascal 文字列は固定長のバイト列に収められた短い可変長の文字列です。count は実際に文字列データ中に収められる全体の長さです。このデータの先頭の 1 バイトには文字列の長さか 255 のうち、小さい方の数が収められます。その後に文字列のバイトデータが続きます。pack() に渡された Pascal 文字列の長さが長すぎた (count-1 よりも長い) 場合、先頭の count-1 バイトが書き込まれます。文字列が count-1 よりも短い場合、指定した count バイトに達するまでの残りの部分はヌルで埋められます。unpack() では、フォーマット文字 'p' は指定された count バイトだけデータを読み込みますが、返される文字列は決して 255 文字を超えることはないので注意してください。

フォーマット文字 'I'、'L'、'q' および 'Q' では、返される値は Python long 整数です。

フォーマット文字 'P' では、返される値は Python 整数型または long 整数型で、これはポインタの値を Python での整数にキャストする際に、値を保持するために必要なサイズに依存します。NULL ポインタは

常に Python 整数型の 0 になります。ポインタ型のサイズを持った値をパックするには、Python 整数型および long 整数型オブジェクトを使うことができます。例えば、Alpha および Merced プロセッサは 64 bit のポインタ値を使いますが、これはポインタを保持するために Python long 整数型が使われることを意味します; 32 bit ポインタを使う他のプラットフォームでは Python 整数型が使われます。

デフォルトでは、C では数値はマシンのネイティブ (native) の形式およびバイトオーダー (byte order) で表され、適切にアラインメント (alignment) するために、必要に応じて数バイトのパディングを行ってスキップします (これは C コンパイラが用いるルールに従います)。

これに代わって、フォーマット文字列の最初の文字を使って、バイトオーダーやサイズ、アラインメントを指定することができます。指定できる文字を以下のテーブルに示します:

文字	バイトオーダー	サイズおよびアラインメント
'@'	ネイティブ	ネイティブ
'='	ネイティブ	標準
'<'	リトルエンディアン	標準
'>'	ビッグエンディアン	標準
'!'	ネットワークバイトオーダー (= ビッグエンディアン)	標準

フォーマット文字列の最初の文字が上のいずれかでない場合、'@" でであるとみなされます。

ネイティブのバイトオーダーはビッグエンディアンかリトルエンディアンで、ホスト計算機に依存します。例えば、Motorola および Sun のプロセッサはビッグエンディアンです; Intel および DEC のプロセッサはリトルエンディアンです。

ネイティブのサイズおよびアラインメントは C コンパイラの sizeof 式で決定されます。ネイティブのサイズおよびアラインメントは大抵ネイティブのバイトオーダーと同時に使われます。

標準のサイズおよびアラインメントは以下ようになります: どの型に対しても、アラインメントは必要ありません (ので、パディングを使う必要があります); short は 2 バイトです; int と long は 4 バイトです; long long (Windows では \_\_int64) は 8 バイトです; float と double は順に 32-bit あるいは 64-bit の IEEE 浮動小数点数です。

'@" と '=' の違いに注意してください: 両方ともネイティブのバイトオーダーですが、後者のバイトサイズやバイトオーダーは標準のものに合わせてあります。

'!' 表記法はネットワークバイトオーダーがビッグエンディアンかリトルエンディアンか忘れちゃったという熱意に乏しい人向けに用意されています。

バイトオーダーに関して、「(強制的にバイトスワップを行う) ネイティブの逆」を指定する方法はありません; '<' または '>' のうちふさわしい方を選んでください。

'p' フォーマット文字はネイティブバイトオーダーでのみ利用可能です (デフォルトのネットワークバイトオーダーに設定するか、'@" バイトオーダー指定文字を指定しなければなりません)。'=' を指定した場合、ホスト計算機のバイトオーダーに基づいてリトルエンディアンとビッグエンディアンのどちらを使うかを決めます。struct モジュールはこの設定をネイティブのオーダー設定として解釈しないので、'p' を使うことはできません。

以下に例を示します (この例は全てビッグエンディアンのマシンで、ネイティブのバイトオーダー、サイズおよびアラインメントの場合です):



```
>>> from struct import *
>>> pack('hhl', 1, 2, 3)
'\x00\x01\x00\x02\x00\x00\x00\x03'
>>> unpack('hhl', '\x00\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
>>> calcsize('hhl')
8
```

ヒント: 特定の型によるアラインメント要求に従うように構造体の末端をそろえるには、count をゼロにした特定の型でフォーマットを終端します。例えば、フォーマット '`llh01`' は、long 型が 4 バイトを境界としてそろえられていると仮定して、末端に 2 バイトをパディングします。この機能は変換対象がネイティブのサイズおよびアラインメントの場合にのみ働きます; 標準に型サイズおよびアラインメントの設定ではいかなるアラインメントも行いません。

参考資料:

array モジュール (5.11 節):

一様なデータ型からなるバイナリ記録データのパック

xdrlib モジュール (12.17 節):

XDR データのパックおよびアンパック。

## 4.4 difflib — 差異の計算を助ける

2.1 で追加された仕様です。

**class SequenceMatcher**

柔軟性のあるクラスで、ハッシュ化できる要素の連続であれば、どんな型のものであっても比較可能です。基礎的なアルゴリズムは可塑的なものであり、1980 年代の後半に発表された Ratcliff と Obershelp によるアルゴリズム、大げさに名づけられた “ゲシュタルトパターンマッチング” よりはもう少し良さそうなものです。その考え方は、“junk” 要素を含まない最も長いマッチ列を探すことです (Ratcliff と Obershelp のアルゴリズムでは junk を示しません)。このアイデアは、下位のマッチ列から左または右に伸びる列の断片に対して再帰的にあてはまります。これは小さな文字列に対して効率良いものではありませんが、人間の目からみて「良く見える」ようにマッチする傾向があります。

タイミング: 基本的な Ratcliff-Obershelp アルゴリズムは、予想の 3 乗、最悪の場合でも 2 乗となります。SequenceMatcher オブジェクトは、最悪のケースに比べて 4 倍、予想される挙動は、シーケンスの中にどのくらいの要素があるのか (最良なのは一列の場合) というややこしい状況に依存しています。

**class Differ**

テキスト行からなるシーケンスを比較するクラスです。人が読むことのできる差異を作成します。Differ クラスは SequenceMatcher クラスを利用します。これらは、列からなるシーケンスを比較し、(ほぼ) 同一の列内の文字を比較します。

Differ クラスによる差異の各行は、2 文字のコードではじめられます。

コード	意味
'- '	列は文字列 1 にのみ存在する
'+ '	列は文字列 2 にのみ存在する
' ' '	列は両方の文字列で同一
'? '	列は入力文字列のどちらにも存在しない

'?' で始まる列は線内の差異に注意を向けようとします。その差異は、入力されたシーケンスのどこ

らにも存在しません。シーケンスがタブ文字を含むとき、これらのラインは判別しづらいものになることがあります。

```
context_diff(a, b[, fromfile[, tofile[, fromfiledate[, tofiledate[, n[, lineterm]]]]]])
```

*a* と *b* (文字列のリスト) を比較し、差異 (差異のある行を生成するジェネレータ) を、diff のコンテキスト形式で返します。

コンテキスト形式は、変更があった行に前後数行を加えてある、コンパクトな表現方法です。変更箇所は、変更前/変更後に分けて表します。コンテキスト (変更箇所前後の行) の行数は *n* で指定し、デフォルト値は 3 です。

デフォルトでは、diff の制御行 (\*\*\*) や -- を含む行) の最後には、改行文字が付加されます。この場合、入出力共、行末に改行文字を持つので、`file.readlines()` で得た入力から生成した差異を、`file.writelines()` に渡す場合に便利です。行末に改行文字を持たない入力に対しては、出力でも改行文字を付加しないように *lineterm* 引数に "" を渡してください。

diff コンテキスト形式は、通常、ヘッダにファイル名と変更時刻を持っています。この情報は、文字列 *fromfile*、*tofile*、*fromfiledate*、*tofiledate* で指定できます。変更時刻の書式は、通常、`time.ctime()` の戻り値と同じものを使います。指定しなかった場合のデフォルト値は、空文字列です。

‘Tools/scripts/diff.py’ は、この関数のコマンドラインのフロントエンド (インターフェイス) になっています。

2.3 で追加された仕様です。

```
get_close_matches(word, possibilities[, n[, cutoff]])
```

最も「十分」なマッチのリストを返します。*word* は、なるべくマッチして欲しい (一般的には文字列の) シーケンスです。*possibilities* は *word* にマッチさせる (一般的には文字列) シーケンスのリストです。

オプションの引数 *n* (デフォルトでは 3) はメソッドの返すマッチの最大数です。*n* は 0 より大きくなければなりません。

オプションの引数 *cutoff* (デフォルトでは 0.6) は、[0, 1] の間となる float の値です。可能性として、少なくとも *word* が無視されたのと同様の数値にはなりません。

可能性のある、(少なくとも *n* に比べて) 最もよいマッチはリストによって返され、同一性を表す数値に応じて最も近いものから順に格納されます。

```
>>> get_close_matches('appel', ['ape', 'apple', 'peach', 'puppy'])
['apple', 'ape']
>>> import keyword
>>> get_close_matches('wheel', keyword.kwlist)
['while']
>>> get_close_matches('apple', keyword.kwlist)
[]
>>> get_close_matches('accept', keyword.kwlist)
['except']
```

```
ndiff(a, b[, linejunk[, charjunk]])
```

*a* と *b* (文字列からなるリスト) を比較し、Differ オブジェクト形式の差異 (解析器は差異のある列) を返します。

オプションのパラメータ *linejunk* と *charjunk* は、filter 機能のためのキーワードです (使わないときは空にする)。

*linejunk*: string 型の引数ひとつを受け取る関数で、文字列が junk か否かによって true を (違うときには true を) 返します。Python 2.3 以降、デフォルトでは (None) になります。それまでは、モ

ジュールレベルの関数 `IS_LINE_JUNK()` であり、それは少なくともひとつのシャープ記号（'#'）をのぞく、可視のキャラクタを含まない行をフィルタリングします。Python 2.3 では、下位にある `SequenceMatcher` クラスが、雑音となるくらい頻繁に登場する列であるか否かを、動的に分析します。これは、バージョン 2.3 以前でのデフォルト値よりうまく動作します。

*charjunk*: 長さ 1 の文字を受け取る関数です。デフォルトでは、モジュールレベルの関数 `IS_CHARACTER_JUNK()` であり、これは空白文字列（空白またはタブ、注：改行文字をこれに含めるのは悪いアイデア！）をフィルタリングします。

'Tools/scripts/ndiff.py' は、この関数のコマンドラインのフロントエンド（インターフェイス）です。

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(1),
...              'ore\ntree\nemu\n'.splitlines(1))
>>> print ''.join(diff),
- one
?  ^
+ ore
?  ^
- two
- three
?  -
+ tree
+ emu
```

**restore**(*sequence, which*)

差異を生成したシーケンスのひとつを返します。

与えられる *sequence* は `Differ.compare()` または `ndiff()` によって生成され、ファイル 1 または 2（引数 *which* で指定される）によって元の列に復元され、行頭のプレフィクスが取りのぞかれます。

例:

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(1),
...              'ore\ntree\nemu\n'.splitlines(1))
>>> diff = list(diff) # materialize the generated delta into a list
>>> print ''.join(restore(diff, 1)),
one
two
three
>>> print ''.join(restore(diff, 2)),
ore
tree
emu
```

**unified\_diff**(*a, b[, fromfile[, tofile[, fromfiledate[, tofiledate[, n[, lineterm]]]]]]*)

*a* と *b*（共に文字列のリスト）を比較し、diff の unified 形式で、差異（差分行を生成するジェネレータ）を返します。

unified 形式は変更があった行に前後数行を加えた、コンパクトな表現方法です。変更箇所は（変更前/変更後を分離したブロックではなく）インライン・スタイルで表されます。コンテキスト（変更箇所前後の行）の行数は、*n* で指定し、デフォルト値は 3 です。

デフォルトでは、diff の制御行（--、++、@@ を含む行）は行末で改行します。この場合、入出力共、行末に改行文字を持つので、`file.readlines()` で得た入力进行处理して生成した差異を、`file.writelines()` に渡す場合に便利です。

行末に改行文字を持たない入力には、出力も同じように改行なしになるように、*lineterm* 引数を "" にセットしてください

diff コンテキスト形式は、通常、ヘッダにファイル名と変更時刻を持っています。この情報は、文字列 *fromfile*、*tofile*、*fromfiledate*、*tofiledate* で指定できます。変更時刻の書式は、通常、*time.ctime()* の戻り値と同じものを使います。指定しなかった場合のデフォルト値は、空文字列です。

‘Tools/scripts/diff.py’ は、この関数のコマンドラインのフロントエンド（インターフェイス）です。

2.3 で追加された仕様です。

**IS\_LINE\_JUNK**(*line*)

無視できる列のとき true を返します。列 *line* が空白、または ‘#’ ひとつのときには無視できます。それ以外の時には無視できません。ndiff() の引数 *linkjunk* としてデフォルトで使用されます。ndiff() の *linejunk* は Python 2.3 以前のものです。

**IS\_CHARACTER\_JUNK**(*ch*)

無視できる文字のとき true を返します。文字 *ch* が空白、またはタブ文字のときには無視できます。それ以外の時には無視できません。ndiff() の引数 *charjunk* としてデフォルトで使用されます。

参考資料:

*Pattern Matching: The Gestalt Approach* (パターンマッチング: 全体アプローチ)

John W. Ratcliff と D. E. Metzener による同一性アルゴリズムに関する議論。Dr. Dobbs's Journal 1988 年 7 月号掲載。

#### 4.4.1 SequenceMatcher オブジェクト

The SequenceMatcher クラスには、以下のようなコンストラクタがあります。:

**class SequenceMatcher**( [*isjunk*[, *a*[, *b*]] ] )

オプションの引数 *isjunk* は、( デフォルトの ) None または sequence エレメントが与えられた場合に true を返し、エレメントが “junk” の場合に限り無視される、ひとつの引数をもった関数である必要があります。 *b* に None が渡されるのは、`lambda x: 0`; が渡されるのと同じことです。言い換えると、どんな要素も無視されません。例えば以下のような引数を渡すことで、空白とタブ文字を無視してキャラクタの列を比較します。

```
lambda x: x in " \t"
```

オプションの引数 *a* と *b* は、比較される文字列です。デフォルトで、それらは空の文字列で、文字列の要素はハッシュ化できます。

SequenceMatcher オブジェクトは以下のメソッドを持ちます。

**set\_seqs**(*a*, *b*)

比較される 2 つの文字列を設定します。

SequenceMatcher オブジェクトは 2 つ目の文字列についての詳細な情報を算定し、保管します。そのため、ひとつの文字列をいくつもの文字列と比較する場合、まず *set\_seq2()* を使って文字列を設定しておき、別の文字列をひとつずつ比較するために、繰り返し *set\_seq()* を呼び出します。

**set\_seq1**(*a*)

比較を行うひとつ目の文字列を設定します。比較される 2 つ目の文字列は変更されません。

**set\_seq2**(*b*)

比較を行う 2 つめ目のシーケンスを設定します。比較されるひとつ目のシーケンスは変更されません。

**find\_longest\_match**(*alo*, *ahi*, *blo*, *bhi*)

*a*[*alo*:*ahi*] と *b*[*blo*: *bhi*] の中から、最長のマッチ列を探します。

*isjunk* が省略されたか None の時、*get\_longest\_match()* は *a*[*i*:*i+k*] が *b*[*j*: *j+k*] と等しいような (*i*, *j*, *k*) を返します。その値は *alo* <= *i* <= *i+k* <= *ahi* かつ *blo* <= *j* <= *j+k* <= *bhi*

となります。(i', j', k') でも、同じようになります。さらに  $k \geq k'$ ,  $i \leq i'$  が  $i == i'$ ,  $j \leq j'$  でも同様です。言い換えると、いくつかのマッチ列すべてのうち、*a* 内で最初に始まるものを返します。そしてその *a* 内で最初のマッチ列すべてのうち *b* 内で最初に始まるものを返します。

```
>>> s = SequenceMatcher(None, " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
(0, 4, 5)
```

引数 *isjunk* が与えられている場合、上記の通り、はじめに再長のマッチ列を判定します。ブロック内に junk 要素が見当たらないような追加条件の際はこれに該当しません。次にそのマッチ列を、その両側の junk 要素にマッチするよう、できる限り広げていきます。そのため結果となる列は、探している列のたまたま直前にあった同一の junk 以外の junk にはマッチしません。

以下は前と同じサンプルですが、空白を junk とみなしています。これは ' abcd' が 2 つ目の列の末尾にある ' abcd' にマッチしないようにしています。代わりに 'abcd' にはマッチします。そして 2 つ目の文字列中、一番左の 'abcd' にマッチします。

```
>>> s = SequenceMatcher(lambda x: x==" ", " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
(1, 0, 4)
```

どんな列にもマッチしない時は、(*alo*, *blo*, 0) を返します。

#### get\_matching\_blocks()

マッチしたシーケンス中で個別にマッチしたシーケンスをあらわす、3 つの値のリストを返します。それぞれの値は (*i*, *j*, *n*) という形式であらわされ、 $a[i:i+n] == b[j:j+n]$  という関係を意味します。3 つの値は *i* と *j* の間で単調に増加します。

最後のタプルはダミーで、( $\text{len}(a)$ ,  $\text{len}(b)$ , 0) という値を持ちます。これは  $n==0$  である唯一のタプルです。

```
>>> s = SequenceMatcher(None, "abxcd", "abcd")
>>> s.get_matching_blocks()
[(0, 0, 2), (3, 2, 2), (5, 4, 0)]
```

#### get\_opcodes()

*a* を *b* にするための方法を記述する 5 つのタプルを返します。それぞれのタプルは (*tag*, *i1*, *i2*, *j1*, *j2*) という形式であらわされます。最初のタプルは  $i1 == j1 == 0$  であり、*i1* はその前にあるタプルの *i2* と同じ値です。同様に *j1* は前の *j2* と同じ値になります。

*tag* の値は文字列であり、次のような意味です。

値	意味
'replace'	$a[i1:i2]$ は $b[j1:j2]$ に置き換えられる
'delete'	$a[i1:i2]$ は削除される。この時、 $j1 == j2$ である
'insert'	$b[j1:j2]$ が $a[i1:i1]$ に挿入される。この時 $i1 == i2$ である。
'equal'	$a[i1:i2] == b[j1:j2]$ (下位の文字列は同一)

例:

```

>>> a = "qabxcd"
>>> b = "abycdf"
>>> s = SequenceMatcher(None, a, b)
>>> for tag, i1, i2, j1, j2 in s.get_opcodes():
...     print ("%7s a[%d:%d] (%s) b[%d:%d] (%s)" %
...             (tag, i1, i2, a[i1:i2], j1, j2, b[j1:j2]))
delete a[0:1] (q) b[0:0] ( )
equal a[1:3] (ab) b[0:2] (ab)
replace a[3:4] (x) b[2:3] (y)
equal a[4:6] (cd) b[3:5] (cd)
insert a[6:6] ( ) b[5:6] (f)

```

**get\_grouped\_opcodes(*n*)**

最大 *n* 行までのコンテキストを含むグループを生成するような、ジェネレータを返します。

このメソッドは、get\_opcodes() で返されるグループの中から、似たような差異のかたまりに分け、間に挟まっている変更の無い部分を省きます。

グループは get\_opcodes() と同じ書式で返されます。

2.3 で追加された仕様です。

**ratio()**

[0, 1] の範囲の浮動小数点で、シーケンスの同一性を測る値を返します。

T が 2 つのシーケンスそれぞれがもつ要素の総数だと仮定し、M をマッチした数とすると、この値は  $2.0 * M / T$  であらわされます。もしシーケンスがまったく同じ場合、値は 1.0 となり、まったく異なる場合には 0.0 となります。

このメソッドは get\_matching\_blocks() または get\_opcodes() がまだ呼び出されていない場合には非常にコストが高く、この時より限定された機能をもった quick\_ratio() もしくは real\_quick\_ratio() を最初に試してみることができます。

**quick\_ratio()**

ratio() で測定する同一性をより素早く、限定された形で測ります。

このメソッドは ratio() より限定されており、これを超えるとは見なされませんが、高速に実行します。

**real\_quick\_ratio()**

ratio() で測定する同一性を非常に素早く測ります。

このメソッドは ratio() より限定されており、これを超えるとは見なされませんが、ratio() や quick\_ratio() より高速に実行します。

この文字列全体のマッチ率を返す 3 つのメソッドは、異なる近似値に基づく異なる結果を返します。とはいえ、quick\_ratio() と real\_quick\_ratio() は、常に ratio() より大きな値を示します。

```

>>> s = SequenceMatcher(None, "abcd", "bcde")
>>> s.ratio()
0.75
>>> s.quick_ratio()
0.75
>>> s.real_quick_ratio()
1.0

```

#### 4.4.2 SequenceMatcher の例

この例は 2 つの文字列を比較します。空白を “junk” とします。



```
>>> s = SequenceMatcher(lambda x: x == " ",
...                       "private Thread currentThread;",
...                       "private volatile Thread currentThread;")
```

`ratio()` は、 $[0, 1]$  の範囲の値を返し、シーケンスの同一性を測ります。経験によると、`ratio()` の値が 0.6 を超えると、シーケンスがよく似ていることを示します。

```
>>> print round(s.ratio(), 3)
0.866
```

シーケンスのどこがマッチしているかにだけ興味のある時には `get_matching_blocks()` が手軽でしょう。

```
>>> for block in s.get_matching_blocks():
...     print "a[%d] and b[%d] match for %d elements" % block
a[0] and b[0] match for 8 elements
a[8] and b[17] match for 6 elements
a[14] and b[23] match for 15 elements
a[29] and b[38] match for 0 elements
```

注意:最後のタプルは、`get_matching_blocks()` が常にダミーであることで返されるものです。`(len(a), len(b), 0)` であり、これは最後のタプルの要素 (マッチするようその数) がゼロとなる唯一のケースです。

はじめのシーケンスがどのようにして 2 番目のものになるのかを知るには、`get_opcodes()` を使います。

```
>>> for opcode in s.get_opcodes():
...     print "%6s a[%d:%d] b[%d:%d]" % opcode
equal a[0:8] b[0:8]
insert a[8:8] b[8:17]
equal a[8:14] b[17:23]
equal a[14:29] b[23:38]
```

See also the function `get_close_matches()` in this module, which shows how simple code building on `SequenceMatcher` can be used to do useful work. `SequenceMatcher` を使った、シンプルで使えるコードを知るには、このモジュールの関数 `get_close_matches()` を参照してください。

### 4.4.3 Differ オブジェクト

`Differ` オブジェクトによって抽出された差分は、最小単位の差分を見ても問題なく抽出されます。反対に、最小の差分の場合にはこれとは反対のように見えます。それらが、どこでも可能ときに同期するからです。時折、思いがけなく 100 ページもの部分にマッチします。隣接するマッチ列の同期するポイントを制限することで、より長い差異を算出する再帰的なコストでの、局所性の概念を制限します。

`Differ` は、以下のようなコンストラクタを持ちます。

```
class Differ([linejunk[, charjunk]])
```

オプションのパラメータ `linejunk` と `charjunk` は filter 関数のために指定します (もしくは `None` を指定)。

`linejunk`: ひとつの文字列引数を受け取れるべき関数です。文字列が `junk` のときに `true` を返します。デ

フォルトでは、None であり、どんな行であっても junk とは見なされません。

*charjunk*: この関数は (長さ 1 の) 文字列を引数として受け取り、文字列が junk であるときに true を返します。デフォルトは None であり、どんな文字列も junk とは見なされません。

Differ オブジェクトは、以下のひとつのメソッドによって使われます (違いを生成します)。

`compare(a, b)`

文字列からなる 2 つのシーケンスを比較し、差異 (を表す文字列からなるシーケンス) を生成します。

Each sequence must contain individual single-line strings ending with newlines. Such sequences can be obtained from the `readlines()` method of file-like objects. The delta generated also consists of newline-terminated strings, ready to be printed as-is via the `writelines()` method of a file-like object. それぞれのシーケンスは、改行文字によって終了する、独立したひと連なりの文字列でなければなりません。そのようなシーケンスは、ファイル形式オブジェクトの `readline()` メソッドによって得ることができます。(得られる) 差異は改行文字で終了する文字列として得られ、ファイル形式オブジェクトの `writeline()` メソッドによって出力できる形になっています。

#### 4.4.4 Differ の例

この例では 2 つのテキストを比較します。初めに、改行文字で終了する独立した 1 行の連続した (ファイル形式オブジェクトの `readlines()` メソッドによって得られるような) テキストを用意します。

```
>>> text1 = ''' 1. Beautiful is better than ugly.
... 2. Explicit is better than implicit.
... 3. Simple is better than complex.
... 4. Complex is better than complicated.
... '''.splitlines(1)
>>> len(text1)
4
>>> text1[0][-1]
'\n'
>>> text2 = ''' 1. Beautiful is better than ugly.
... 3. Simple is better than complex.
... 4. Complicated is better than complex.
... 5. Flat is better than nested.
... '''.splitlines(1)
```

次に Differ オブジェクトをインスタンス化します。

```
>>> d = Differ()
```

注意: Differ オブジェクトをインスタンス化するとき、“junk.” である列と文字をフィルタリングす関数を渡すことができます。詳細は `Differ()` コンストラクタを参照してください。

最後に、2 つを比較します。

```
>>> result = list(d.compare(text1, text2))
```

`result` は文字列のリストなので、pretty-print してみましょう。

```
>>> from pprint import pprint
>>> pprint(result)
['    1. Beautiful is better than ugly.\n',
 '-   2. Explicit is better than implicit.\n',
 '-   3. Simple is better than complex.\n',
 '+   3. Simple is better than complex.\n',
 '?    ++                                \n',
 '-   4. Complex is better than complicated.\n',
 '?      ^                        ---- ^ \n',
 '+   4. Complicated is better than complex.\n',
 '?      +++++ ^                      ^ \n',
 '+   5. Flat is better than nested.\n']
```

これは、複数行の文字列として、次のように出力されます。

```
>>> import sys
>>> sys.stdout.writelines(result)
1. Beautiful is better than ugly.
- 2. Explicit is better than implicit.
- 3. Simple is better than complex.
+ 3. Simple is better than complex.
?  ++
- 4. Complex is better than complicated.
?   ^                        ---- ^
+ 4. Complicated is better than complex.
?   +++++ ^                      ^
+ 5. Flat is better than nested.
```

## 4.5 `fpformat` — 浮動小数点の変換

**注意:** This module is unneeded: everything here could be done via the `%` string interpolation operator.

`fpformat` モジュールは浮動小数点数の表示を 100% 純粋に Python だけで行うための関数を定義しています。注意: このモジュールは必要ありません: このモジュールのすべてのことは、`%` を使って、文字列の補間演算により可能です。

`fpformat` モジュールは次にあげる関数と例外を定義しています。

**`fix(x, digs)`**

`x` を `[-]ddd.ddd` の形にフォーマットします。小数点の後ろに `digs` 桁と、小数点の前に少なくとも 1 桁です。`vardigs <= 0` の場合、小数点以下は切り捨てられます。

`x` は数字か数字を表した文字列です。

`digs` は整数です。

返り値は文字列です。

**`sci(x, digs)`**

`x` を `[-]d.dddE[+-]ddd` の形にフォーマットします。小数点の後ろに `digs` 桁と、小数点の前に 1 桁だけです。

`vardigs <= 0` の場合、1 桁だけ残され、小数点以下は切り捨てられます。

`x` は実数か実数を表した文字列です。

`digs` は整数です。

返り値は文字列です。

#### exception NotANumber

`fix()` や `sci()` にパラメータとして渡された文字列  $x$  が数字として認識できなかった場合、例外が発生します。標準の例外が文字列の場合、この例外は `ValueError` のサブクラスです。例外値は、例外を発生させた不適切にフォーマットされた文字列です。

例:

```
>>> import fpformat
>>> fpformat.fix(1.23, 1)
'1.2'
```

## 4.6 StringIO — ファイルのように文字列を読み書きする

このモジュールは、(メモリファイルとしても知られている) 文字列のバッファに対して読み書きを行うファイルのようなクラス、`StringIO`、を実装しています。

操作方法についてはファイルオブジェクトの説明を参照してください(セクション 2.3.8)。

`class StringIO([buffer])`

`StringIO` オブジェクトを作る際に、コンストラクターに文字列を渡すことで初期化することができます。文字列を渡さない場合、最初は `StringIO` はカラです。

`StringIO` オブジェクトはユニコードも 8-bit の文字列も受け付けますが、この 2 つを混ぜることに少し注意が必要です。この 2 つが一緒に使われると、`getvalue()` が呼ばれたときに、(8th ビットを使っている) 7-bit ASCII に解釈できない 8-bit の文字列は、`UnicodeError` を引き起こします。

次にあげる `StringIO` オブジェクトのメソッドには特別な説明が必要です:

`getvalue()`

`StringIO` オブジェクトの `close()` メソッドが呼ばれる前ならいつでも、“file” の中身全体を返します。ユニコードと 8-bit の文字列を混ぜることの説明は、上の注意を参照してください。この 2 つの文字コードを混ぜると、このメソッドは `UnicodeError` を引き起こすかもしれません。

`close()`

メモリバッファを解放します。

## 4.7 cStringIO — 高速化された StringIO

`cStringIO` モジュールは `StringIO` モジュールと同様のインターフェースを提供しています。`StringIO.StringIO` オブジェクトを酷使する場合、このモジュールにある `StringIO()` 関数をかわりに使うと効果的です。

このモジュールは、ビルトイン型のオブジェクトを返すファクトリー関数を提供しているので、サブクラス化して自分用の物を作ることはできません。そうした場合には、オリジナルの `StringIO` モジュールを使ってください。

`StringIO` モジュールで実装されているメモリファイルとは異なり、このモジュールで提供されているものは、プレーン ASCII 文字列にエンコードできないユニコードを受け付けることができません。

また、引数に文字列を指定して `StringIO()` 呼び出すと読み出し専用のオブジェクトが生成されますが、この場合 `cStringIO.StringIO()` では `write()` メソッドを持たないオブジェクトを生成します。

次にあげるデータオブジェクトも提供されています:

### InputType

文字列をパラメーターに渡して StringIO を呼んだときに作られるオブジェクトのオブジェクト型。

### OutputType

パラメーターを渡さずに StringIO を呼んだときに返されるオブジェクトのオブジェクト型。

このモジュールには C API もあります。詳しくはこのモジュールのソースを参照してください。

## 4.8 textwrap — テキストの折り返しと詰め込み

2.3 で追加された仕様です。

textwrap モジュールでは、二つの簡易関数 wrap() と fill()、そして作業のすべてを行うクラス TextWrapper とユーティリティ関数 dedent() を提供しています。単に一つや二つのテキスト文字列の折り返しまたは詰め込みを行っているならば、簡易関数で十分間に合います。そうでなければ、効率のために TextWrapper のインスタンスを使った方が良いでしょう。

**wrap**(text[, width[, ...]])

text(文字列) 内の段落を一つだけ折り返しを行います。したがって、すべての行が高々 width 文字の長さになります。最後に改行が付かない出力行のリストを返します。

オプションのキーワード引数は、以下で説明する TextWrapper のインスタンス属性に対応しています。width はデフォルトで 70 です。

**fill**(text[, width[, ...]])

text 内の段落を一つだけ折り返しを行い、折り返しが行われた段落を含む一つの文字列を返します。

fill() は

```
"\n".join(wrap(text, ...))
```

の省略表現です。

特に、fill() は wrap() とまったく同じ名前のキーワード引数を受け取ります。

wrap() と fill() の両方ともが TextWrapper インスタンスを作成し、その一つのメソッドを呼び出すことで機能します。そのインスタンスは再利用されません。したがって、たくさんのテキスト文字列を折り返し/詰め込みを行うアプリケーションのためには、あなた自身の TextWrapper オブジェクトを作成することでさらに効率が良くなるでしょう。

追加のユーティリティ関数である dedent() は、不要な空白をテキストの左側に持つ文字列からインデントを取り去ります。

**dedent**(text)

text の各行に対し、左側から一様に取り去ることができるような空白を除去します。

この関数は通常、三重引用符で囲われた文字列をスクリーン/その他の左端にそろえ、なおかつソースコード中でのインデントされた形式を損なわないようにするために使われます。

以下に例を示します:

```
def test():
    # end first line with \ to avoid the empty line!
    s = '''\
hello
    world
    '''
    print repr(s)          # prints 'hello\n    world\n'
    print repr(dedent(s))  # prints 'hello\n world\n'
```

```
class TextWrapper(...)
```

`TextWrapper` コンストラクタはたくさんのオプションのキーワード引数を受け取ります。それぞれの引数は一つのインスタンス属性に対応します。したがって、例えば、

```
wrapper = TextWrapper(initial_indent="* ")
```

は

```
wrapper = TextWrapper()  
wrapper.initial_indent = "* "
```

と同じです。

あなたは同じ `TextWrapper` オブジェクトを何回も再利用できます。また、使用中にインスタンス属性へ代入することでそのオプションのどれでも変更できます。

`TextWrapper` インスタンス属性 (とコンストラクタのキーワード引数) は以下の通りです:

**width**

(デフォルト: 70) 折り返しが行われる行の最大の長さ。入力行に `width` より長い単一の語が無い限り、`TextWrapper` は `width` 文字より長い出力行が無いことを保証します。

**expand\_tabs**

(デフォルト: `True`) もし真ならば、そのときは `text` 内のすべてのタブ文字は `text` の `expand_tabs()` メソッドを用いて空白に展開されます。

**replace\_whitespace**

(デフォルト: `True`) もし真ならば、タブ展開の後に残る (`string.whitespace` に定義された) 空白文字のそれぞれが一つの空白と置き換えられます。注意: `expand_tabs` が偽で `replace_whitespace` が真ならば、各タブ文字は一つの空白に置き換えられます。それはタブ展開と同じではありません。

**initial\_indent**

(デフォルト: `"`) 折り返しが行われる出力の一行目の先頭に付けられる文字列。一行目の折り返しの長さになるまで含められます。

**subsequent\_indent**

(デフォルト: `"`) 一行目以外の折り返しが行われる出力のすべての行の先頭に付けられる文字列。一行目以外の各行が折り返しの長さまで含められます。

**fix\_sentence\_endings**

(デフォルト: `False`) もし真ならば、`TextWrapper` は文の終わりを見つけようとし、確実に文がちょうど二つの空白で常に区切られているようにします。これは一般的に固定スペースフォントのテキストに対して望ましいです。しかし、文の検出アルゴリズムは完全ではありません: 文の終わりには、後ろに空白がある `'.'`、`'!'` または `'?'` の中の一つ、ことによると `"` あるいは `'` が付随する小文字があると仮定しています。これに伴う一つの問題は

```
[...] Dr. Frankenstein's monster [...]
```

の `"Dr."` と

```
[...] See Spot. See Spot run [...]
```

の `"Spot."` の間の差異を検出できないアルゴリズムです。

`fix_sentence_endings` はデフォルトで偽です。



文検出アルゴリズムは“小文字”の定義のために `string.lowercase` に依存し、同一行の文を区切るためにピリオドの後に二つの空白を使う慣習に依存しているため、英文テキストに限定されたものです。

#### `break_long_words`

(デフォルト: `True`) もし真ならば、そのとき `width` より長い行が確実にないようにするために、`width` より長い語は切られます。偽ならば、長い語は切られないでしょう。そして、`width` より長い行があるかもしれません。( `width` を超える分を最小にするために、長い語は単独で一行に置かれるでしょう。)

`TextWrapper` はモジュールレベルの簡易関数に類似した二つの公開メソッドも提供します:

#### `wrap(text)`

`text`(文字列) 内の段落を一つだけ折り返しを行います。したがって、すべての行は高々 `width` 文字です。すべてのラッピングオプションは `TextWrapper` インスタンスのインスタンス属性から取られています。最後に改行の無い出力された行のリストを返します。

#### `fill(text)`

`text` 内の段落を一つだけ折り返しを行い、折り返しが行われた段落を含む一つの文字列を返します。

## 4.9 codecs — codec レジストリと基底クラス

このモジュールでは、標準の Python codec (エンコーダとデコーダ) の基底クラスを定義し、codec およびエラー処理の検索手順を管理している、内部的な Python codec レジストリに対するアクセス手段を提供しています。

`codecs` では以下の関数を定義しています:

#### `register(search_function)`

codec 検索関数を登録します。検索関数は第 1 引数にアルファベットの小文字から成るエンコーディング名を取り、関数のタプル (`encoder`, `decoder`, `stream_reader`, `stream_writer`) を返すことになっています。戻り値の関数が取る引数は以下の通りです。

*encoder* と *decoder*: これらは、Codec インスタンスの `encode()` と `decode()` (Codec Interface 参照) と同じインターフェイスを持つ関数、またはメソッドでなければなりません。これらの関数・メソッドは状態なし (stateless) モードで動作しなければなりません。

*stream\_reader* と *stream\_writer*: これらは次のようなインターフェイスを持つファクトリ関数でなければなりません。

`factory(stream, errors='strict')`

これらのファクトリ関数は、それぞれの基底クラスである `StreamWriter` と `StreamReader` で定義されているインターフェイスを提供するオブジェクトを返さなければいけません。ストリーム `codecs` は状態を保持することができます。

`errors` が取り得る値は、`'strict'` (エンコーディングエラーの際に例外を発生)、`'replace'` (奇形データを `'?'` 等の適切な文字で置換)、`'ignore'` (奇形データを無視し何も通知せずに処理を継続)、`'xmlcharrefreplace'` (適切な XML 文字参照で置換 (エンコーディングのみ))、および `'backslashreplace'` (バックスラッシュによるエスケープシーケンス (エンコーディングのみ)) と、`register_error()` で定義されたその他のエラーハンドル名になります。

検索関数が与えられたエンコーディングを見つけられなかった場合には、`None` を返すべきです。

#### `lookup(encoding)`

Python codec レジストリから codec タプルを探し、上の項目で定義された関数のタプルを返します。

最初に、レジストリのキャッシュから `encoding` を探します。見つからなければ、登録されている検索関数のリストから探します。見つからなければ、`LookupError` が発生し、見つければ `codec` のタプルがキャッシュに保存され、それを呼び出し側に返します。

さまざまな `codec` へのアクセスを簡便化するために、このモジュールは次に挙げるような関数を提供します。これらは `codec` の検索に `lookup()` を使います。

**getencoder**(*encoding*)

`encoding` で指定された `codec` を検索し、エンコード関数を返します。

`encoding` が見つからなければ `LookupError` が発生します。

**getdecoder**(*encoding*)

`encoding` で指定された `codec` を検索し、デコード関数を返します。

`encoding` が見つからなければ `LookupError` が発生します。

**getreader**(*encoding*)

`encoding` で指定された `codec` を検索し、`StreamReader` クラス、またはファクトリ関数を返します。

`encoding` が見つからなければ `LookupError` が発生します。

**getwriter**(*encoding*)

`encoding` で指定された `codec` を検索し、`StreamWriter` クラス、またはファクトリ関数を返します。

`encoding` が見つからなければ `LookupError` が発生します。

**register\_error**(*name*, *error\_handler*)

エラー処理関数 *error\_handler* を名前 *name* で登録します。エンコード中およびデコード中にエラーが発生した場合、*name* が `errors` パラメタとして指定されていれば *error\_handler* が呼び出されます。

*error\_handler* はエラーの場所に関する情報の入った `UnicodeEncodeError` インスタンスとともに呼び出されます。エラー処理関数はこの例外を送出するか、別の例外を送出するか、または入力のエンコードができなかった部分の代替文字列、およびどこからエンコードを再開するかを指定するタプルを返さなければなりません。エンコーダは代替文字列をエンコードし、元の入力の指定された場所からエンコードを再開します。負の位置は入力文字列の末端からの相対位置として扱われます。返された位置が境界の外側にある場合には `IndexError` が送出されます。

デコードと翻訳は同様に働きますが、`UnicodeDecodeError` または `UnicodeTranslateError` がハンドラに渡される点が異なります。また、エラーハンドラから得られた置換文字列が出力に直接置かれる点も異なります。

**lookup\_error**(*name*)

既に、名前 *name* 以下に登録されているエラー処理関数を返します。

エラー処理関数が見つからなければ `LookupError` が発生します。

**strict\_errors**(*exception*)

strict エラー処理を実装しています。

**replace\_errors**(*exception*)

replace エラー処理を実装しています。

**ignore\_errors**(*exception*)

ignore エラー処理を実装しています。

**xmlcharrefreplace\_errors**(*exception*)

xmlcharrefreplace エラー処理を実装しています。

**backslashreplace\_errors**(*exception*)

backslashreplace エラー処理を実装しています。

エンコードされたファイルやストリームの処理を簡便化するため、このモジュールは次のようなユーティリティ関数を定義しています。

`open(filename, mode[, encoding[, errors[, buffering]]])`

指定された *mode* でエンコードされたファイルを開き、透過なエンコード・デコードを提供するような、ラップされた版のファイルオブジェクトを返します。

注意: ラップされた版は、その codec で定義されたフォーマットのオブジェクトのみ受け付けます。多くの組み込み codec において Unicode オブジェクトです。出力も codec に依存し、通常は Unicode オブジェクトです。

*encoding* は、ファイルに使われるエンコーディングを指定します。

*errors* を指定して、エラー処理を定義することもできます。デフォルトでは 'strict' で、エンコード時にエラーがあれば `ValueError` が発生します。

*buffering* は、組み込み関数 `open()` と同じです。デフォルトでは、ラインバッファです。

`EncodedFile(file, input[, output[, errors]])`

ラップされた版のファイルオブジェクトを返し、このオブジェクトは透過なエンコード変換を提供します。

ラップされたファイルに書かれた文字列は、与えられた *input* エンコーディングに従って変換され、*output* エンコーディングを使ってファイルに文字列として書き込まれます。中間エンコーディングは、通常 Unicode ですが、指定された codecs に依存します。

*output* が与えられなければ、*input* がデフォルトになります。

*errors* を与えて、エラー処理を定義することもできます。デフォルトでは 'strict' で、エンコード時にエラーがあれば `ValueError` が発生します。

このモジュールは以下のような定数を定義します。プラットフォーム依存なファイルを読み書きするのに役立ちます。

`BOM`

`BOM_BE`

`BOM_LE`

`BOM_UTF8`

`BOM_UTF16`

`BOM_UTF16_BE`

`BOM_UTF16_LE`

`BOM_UTF32`

`BOM_UTF32_BE`

`BOM_UTF32_LE`

ここで定義された定数は、様々なエンコーディングの Unicode のバイト・オーダー・マーク (BOM) で、UTF-16 と UTF-32 におけるデータストリームやファイルストリームのバイトオーダーを指定したり、UTF-8 における Unicode signature として使われます。BOM\_UTF16 は BOM\_UTF16\_BE と BOM\_UTF16\_LE のいずれかで、プラットフォームのネイティブ・バイトオーダーに依存します。BOM は BOM\_UTF16 のエイリアスです。同様に BOM\_LE は BOM\_UTF16\_LE の、BOM\_BE は BOM\_UTF16\_BE のエイリアスです。他は UTF-8 と UTF-32 エンコーディングの BOM を表します。

参考資料:

<http://sourceforge.net/projects/python-codecs/>

Python で使う、アジア文字用の codec のサポートに取り組む SourceForge のプロジェクト。この文書の執筆時点では開発の初期段階です。 — ダウンロード可能なファイルについては、プロジェクトの FTP エリア を見てください。

## 4.9.1 Codec 基底クラス

codecs は、インターフェイスを定義し、自前の Python 用 codec を、簡単に書くのに使える基底クラス群を定義してあります。

各 codec は、Python の codec として使えるように、4 つのインターフェイスを定義しなければなりません。状態なしエンコーダ、状態なしデコーダ、ストリームリーダ、ストリームライタです。ストリームリーダとライタは、通常、状態なしエンコーダとデコーダを再利用して、ファイル・プロトコルを実装します。

Codec クラスは、状態なしエンコーダ・デコーダのインターフェイスを定義します。

エラー処理を簡便化し、安定させるために、`encode()` メソッドと `decode()` メソッドは、`errors` 文字列引数を提供することで、異なるエラー処理の仕組みを実装してもかまいません。以下の文字列は全ての標準 Python codec で定義および実装されています。

Value	Meaning
'strict'	UnicodeError (または、そのサブクラス) を発生 – これがデフォルト。
'ignore'	その文字を無視し、次の文字から変換を再開
'replace'	適当な文字で置換 – Python の組み込み Unicode codec のデコード時には公式の U+FFFD REPLACEMENT CHARACTER
'xmlcharrefreplace'	適切な XML 文字参照で置換 (エンコードのみ)
'backslashreplace'	バックスラッシュ付きのエスケープシーケンスで置換 (エンコードのみ)

受け付ける値は、`register_error` を使って追加できます。

### Codec オブジェクト

Codec クラスは以下のメソッドを定義します。これらのメソッドは、状態を保持しないエンコーダとデコーダ関数のインターフェイスを定義します。

**`encode(input[, errors])`**

オブジェクト `input` エンコードし、(出力 オブジェクト, 消費した長さ) のタプルを返します。codecs は Unicode 専用ではありませんが、Unicode の文脈では、エンコーディングは Unicode オブジェクトを、特定の文字集合エンコーディング (たとえば cp1252 や iso-8859-1) を使って文字列オブジェクトに変換します。

`errors` は適用するエラー処理を定義します。'strict' 処理がデフォルトです。

このメソッドは Codec の状態を保持しなくてかまいません。効率よくエンコード/デコードするために、状態を保持しなければならない codecs には StreamCodec を使いましょう。

エンコーダは長さが 0 の入力を受け付け、その場合には、空のオブジェクトを出力オブジェクトとして返す必要があります。

**`decode(input[, errors])`**

オブジェクト `input` をデコードし、(出力オブジェクト, 消費した長さ) のタプルを返します。Unicode の文脈では、デコードは特定の文字集合エンコーディングでエンコードされた文字列を Unicode オブジェクトに変換します。

`input` は `bf_getreadbuf` バッファスロットを提供するオブジェクトでなければいけません。Python 文字列オブジェクト、バッファオブジェクト、メモリにマップされたファイルが、バッファスロットの例です。

`errors` は適用するエラー処理を定義します。'strict' がデフォルト値です。

このメソッドは、Codec インスタンス内部の状態を保持しなくてもかまいません。効率よくエンコード/デコードするために、状態を保持しなければならない codecs には StreamCodec を使いま

しょう。

エンコーダは長さが0の入力を受け付け、その場合には、空のオブジェクトを出力オブジェクトとして返す必要があります。

`StreamWriter` と `StreamReader` クラスは、新しいエンコーディングモジュールを、非常に簡単に実装するのに使用できる、一般的なインターフェイス提供します。実装例は `encodings.utf_8` をご覧ください。

## StreamWriter オブジェクト

`StreamWriter` クラスは `Codec` のサブクラスで、以下のメソッドを定義しています。全てのストリーム・ライターは、Python の `codec` レジストリとの互換性を保つために、これらのメソッドを定義する必要があります。

**class StreamWriter(*stream* [, *errors* ])**

Constructor for a `StreamWriter` instance. `StreamWriter` インスタンスのコンストラクタ。

全てのストリーム・ライターのコンストラクタは、このインターフェイスを提供しなければなりません。キーワード引数を追加しても構いませんが、Python の `codec` レジストリは、ここで定義される引数だけを使います。

*stream* は、(バイナリで)書き込み可能なファイル的なオブジェクトでなくてはなりません。

`StreamWriter` は、*errors* キーワード引数を受けて、異なったエラー処理の仕組みを実装しても構いません。以下のパラメータは、あらかじめ定義されています。

- `'strict'` `ValueError` (または、そのサブクラス) を発生。これがデフォルトです。
- `'ignore'` 文字を無視して、次の文字から続ける。
- `'replace'` 適切な置換文字で置換。
- `'xmlcharrefreplace'` 適切な XML 文字参照で置換。
- `'backslashreplace'` バックスラッシュ付きのエスケープシーケンスで置換。

*errors* 引数は、同名の属性に代入されます。この属性を変更することで、`StreamWriter` オブジェクトが生きている間に、異なるエラー処理に変更することができます。

*errors* 引数を取り得る値の種類は、`register_error()` で拡張できます。

**write(*object*)**

Writes the object's contents encoded to the stream. エンコードされた *object* の内容をストリームに書き出す。

**writelines(*list*)**

連結した、文字列リストを (おそらくは `write()` メソッドを再利用して) ストリームに書き出す。

**reset()**

状態保持に使われていた `codec` のバッファを、強制的に出力し、リセットする。

このメソッドの呼び出したとき、出力先データをきれいな状態にし、状態を元に戻すのにストリーム全体を再スキャンすることなく、新しくフラッシュするデータを追加できるように保障する必要があります。

ここまでで挙げたメソッドに加えて、`StreamWriter` は、元になっているストリームから、他の全てのメソッドや属性を、継承しなければなりません。



## StreamReader オブジェクト

StreamReader クラスは Codec のサブクラスで、以下のようなメソッドを定義します。Python の codec レジストリと互換性を保つには、これらのメソッド全てを定義しなければなりません。

**class StreamReader** (*stream* [, *errors*])

StreamReader インスタンスのコンストラクタ。

全てのストリーム・リーダは、このインターフェースを持つコンストラクタを定義しなければいけません。キーワード引数を追加しても構いませんが、ここで定義されているものだけが、Python の codec レジストリから使われます。

ここでは Python の codec レジストリが使うものを説明します。

*stream* は (バイナリ) データを読み込み可能なファイル的オブジェクトでなければいけません。

The StreamReader は、*errors* キーワード引数を受けて、異なったエラー処理の仕組みを実装しても構いません。以下のパラメータは、あらかじめ定義されています。

- 'strict' ValueError (または、そのサブクラス) を発生。これがデフォルトです。
- 'ignore' 文字を無視して、次の文字から続ける。
- 'replace' 適切な置換文字で置換。

*errors* 引数は、同名の属性に代入されます。この属性を変更することで、StreamReader オブジェクトが生きている間に、異なるエラー処理に変更することができます。

*errors* 引数を取り得る値の種類は、`register_error()` で拡張できます。

**read** ([*size*])

ストリームからのデータをデコードし、デコード済のオブジェクトを返す。

*size* は、デコードするためにストリームから読み込む、およその最大バイト数を意味します。デコーダは、この値を適切な値に変更できます。デフォルト値 -1 にすると、可能な限りたくさんのデータを読み込みます。*size* の目的は、巨大なファイルを 1 度にデコードするのを防ぐことです。

このメソッドは貪欲な読み込み戦略を取るべきです。すなわち、エンコーディング定義と *size* の値が許す範囲で、できるだけ多くのデータを読むべきだということです。たとえば、ストリーム上にエンコーディングの終端や状態の目印があれば、それも読み込みます。

**readline** ([*size*])

入力ストリームから 1 行読み込み、デコード済みのデータを返す。

`readlines()` とは異なり、このメソッドは改行情報を、元のストリームの `readline()` メソッドから継承します。- 行バッファの不足のため、現在、`cedec` のデコーダを使った改行のサポートはありません。しかし、サブクラスは可能であるなら、独自の改行情報を使って、このメソッドの実装を試みるべきです。

*size* が与えられた場合、ストリームにおける `readline()` の *size* 引数に渡されます。

**readlines** ([*sizehint*])

インプットストリームから全ての行を読み込み、行のリストとして返します。

改行は、`codec` デコーダのメソッド実装され、リスト要素の中に含まれます。

*sizehint* が与えられた場合、ストリームの `read()` メソッドに *size* 引数として渡されます。

**reset** ()

状態保持に使われた `codec` のバッファをリセットする。

ストリームの再配置を行うべきではないので注意してください。このメソッドはデコードの際のエラーから復帰できるようにすることを目的としたものです。



ここまでで挙げたメソッドに加えて、`StreamReader` は、元になっているストリームから、他の全てのメソッドや属性を、継承しなければなりません。

次に挙げる 2 つの基底クラスは、利便性のために含まれています。`codec` レジストリは、これらを必要としませんが、実際のところ、あると有用なものでしょう。

#### StreamReaderWriter オブジェクト

`StreamReaderWriter` を使って、読み書き両方に使えるストリームをラップできます。

`lookup()` 関数が返すファクトリ関数を使って、インスタンスを生成するという設計です。

```
class StreamReaderWriter(stream, Reader, Writer, errors)
```

`StreamReaderWriter` インスタンスを生成する。`stream` はファイル的オブジェクトです。`Reader` と `Writer` は、それぞれ `StreamReader` と `StreamWriter` インターフェイスを提供するファクトリ関数がファクトリクラスでなければなりません。エラー処理は、ストリーム・リーダーとライターで定義したものと同じように行われます。

`StreamReaderWriter` インスタンスは、`StreamReader` クラスと `StreamWriter` クラスを合わせたインターフェイスを継承します。元になるストリームからは、他のメソッドや属性を継承します。

#### StreamRecoder オブジェクト

`StreamRecoder` はエンコーディングデータの、フロントエンド-バックエンドを観察する機能を提供します。異なるエンコーディング環境を扱うとき、便利な場合があります。

`lookup()` 関数が返すファクトリ関数を使って、インスタンスを生成するという設計になっています。

```
class StreamRecoder(stream, encode, decode, Reader, Writer, errors)
```

双方向変換を実装する `StreamRecoder` インスタンスを生成します。`encode` と `decode` はフロントエンド (`read()` への入力と `write()` からの出力) を処理し、`Reader` と `Writer` はバックエンド (ストリームに対する読み書き) を処理します。

これらのオブジェクトを使って、たとえば、Latin-1 から UTF-8、あるいは逆向きの変換を、透過に記録することができます。

`stream` はファイル的オブジェクトでなくてははいけません。

`encode` と `decode` は `Codec` のインターフェイスに忠実でなくてははいけません。`Reader` と `Writer` は、それぞれ `StreamReader` と `StreamWriter` のインターフェイスを提供するオブジェクトのファクトリ関数がクラスでなくてははいけません。

`encode` と `decode` はフロントエンドの変換に必要で、`Reader` と `Writer` はバックエンドの変換に必要です。中間のフォーマットはコーデックの組み合わせによって決定されます。たとえば、Unicode コデックは中間エンコーディングに Unicode を使います。

エラー処理はストリーム・リーダーやライターで定義されている方法と同じように行われます。

`StreamRecoder` インスタンスは、`StreamReader` と `StreamWriter` クラスを合わせたインターフェイスを定義します。また、元のストリームのメソッドと属性も継承します。

### 4.9.2 標準エンコーディング

Python には数多くの `codec` が組み込みで付属します。これらは C 言語の関数、対応付けを行うテーブルの両方で提供されています。以下のテーブルでは `codec` と、いくつかの良く知られている別名と、エンコーディングが使われる言語を列挙します。別名のリスト、言語のリストともしらみつぶしに網羅されている

わけではありません。大文字と小文字、またはアンダースコアの代りにハイフンにただけの綴りも有効な別名です。

多くの文字セットは同じ言語をサポートしています。これらの文字セットは個々の文字 (例えば、EURO SIGN がサポートされているかどうか) や、文字のコード部分への割り付けが異なります。特に欧州言語では、典型的に以下の変種が存在します:

- ISO 8859 コードセット
- Microsoft Windows コードページで、8859 コード形式から導出されているが、制御文字を追加のグラフィック文字と置き換えたもの
- IBM EBCDIC コードページ
- ASCII 互換の IBM PC コードページ

Codec	別名	言語
ascii	646, us-ascii	イギリス
cp037	IBM037, IBM039	イギリス
cp424	EBCDIC-CP-HE, IBM424	ヘブライ
cp437	437, IBM437	イギリス
cp500	EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500	西ヨーロッパ
cp737		ギリシャ
cp775	IBM775	バルト沿岸国
cp850	850, IBM850	西ヨーロッパ
cp852	852, IBM852	中央および東ヨーロッパ
cp855	855, IBM855	ブルガリア、ベラルーシ、マケドニア、ロシア、セルビア
cp856		ヘブライ
cp857	857, IBM857	トルコ
cp860	860, IBM860	ポルトガル
cp861	861, CP-IS, IBM861	アイスランド
cp862	862, IBM862	ヘブライ
cp863	863, IBM863	カナダ
cp864	IBM864	アラビア
cp865	865, IBM865	デンマーク、ノルウェー
cp869	869, CP-GR, IBM869	ギリシャ
cp874		タイ
cp875		ギリシャ
cp1006		Urdu
cp1026	ibm1026	トルコ
cp1140	ibm1140	西ヨーロッパ
cp1250	windows-1250	中央および東ヨーロッパ
cp1251	windows-1251	ブルガリア、ベラルーシ、マケドニア、ロシア、セルビア
cp1252	windows-1252	西ヨーロッパ
cp1253	windows-1253	ギリシャ
cp1254	windows-1254	トルコ
cp1255	windows-1255	ヘブライ
cp1256	windows1256	アラビア
cp1257	windows-1257	バルト沿岸国

Codec	別名	言語
cp1258	windows-1258	ベトナム
latin_1	iso-8859-1, iso8859-1, 8859, cp819, latin, latin1, L1	西ヨーロッパ
iso8859_2	iso-8859-2, latin2, L2	中央および東ヨーロッパ
iso8859_3	iso-8859-3, latin3, L3	エスペラント、マルタ
iso8859_4	iso-8859-4, latin4, L4	バルト沿岸国
iso8859_5	iso-8859-5, cyrillic	ブルガリア、ベラルーシ、マケドニア、ロシア、セルビア
iso8859_6	iso-8859-6, arabic	アラビア
iso8859_7	iso-8859-7, greek, greek8	ギリシャ
iso8859_8	iso-8859-8, hebrew	ヘブライ
iso8859_9	iso-8859-9, latin5, L5	トルコ
iso8859_10	iso-8859-10, latin6, L6	北欧
iso8859_13	iso-8859-13	バルト沿岸国
iso8859_14	iso-8859-14, latin8, L8	ケルト
iso8859_15	iso-8859-15	西ヨーロッパ
koi8_r		ロシア
koi8_u		ウクライナ
mac_cyrillic	maccyrillic	ブルガリア、ベラルーシ、マケドニア、ロシア、セルビア
mac_greek	macgreek	ギリシャ
mac_iceland	maciceland	アイスランド
mac_latin2	maclatin2, maccentraleurope	中央および東ヨーロッパ
mac_roman	macroman	西ヨーロッパ
mac_turkish	macturkish	トルコ
utf_16	U16, utf16	全ての言語
utf_16_be	UTF-16BE	全ての言語 (BMP only)
utf_16_le	UTF-16LE	全ての言語 (BMP only)
utf_7	U7	全ての言語
utf_8	U8, UTF, utf8	全ての言語

数多くの codec は Python 特有なので、それらの codec 名は Python の外では無意味なものとなります。これらの codec のいくつかは Unicode 文字列からバイト文字列への変換を行わず、むしろ単一の引数をもつ全写像関数はエンコーディングとみなすことができるという Python codec の性質を利用しています。

以下に列挙した codec では、“エンコード”方向の結果は常にバイト文字列方向です。“デコード”方向の結果はテーブル内の被演算子型として列挙されています。

Codec	別名	被演算子の型	目的
base64_codec	base64, base-64	byte string	被演算子を MIME base64 に変換します
hex_codec	hex	byte string	被演算子をバイトあたり 2 桁の 16 進数の
idna		Unicode string	RFC 3490 を実装しています。2.3 で追加
mbcs	dbcs	Unicode string	Windows のみ: 被演算子を ANSI コードペ
palms		Unicode string	PalmOS 3.5 のエンコーディングです
punycode		Unicode string	RFC 3492 を実装しています。2.3 で追加
quopri_codec	quopri, quoted-printable, quotedprintable	byte string	被演算子を MIME quoted printable 形式に
raw_unicode_escape		Unicode string	Python ソースコードにおける raw Unicod
rot_13	rot13	byte string	被演算子のシーザー暗号 (Caesar-cypher)
string_escape		byte string	Python ソースコードにおける文字列リテ
undefined		any	全ての変換に対して例外を送出します。/
unicode_escape		Unicode string	Python ソースコードにおける Unicode リ
unicode_internal		Unicode string	被演算子の内部表現を返します。
uu_codec	uu	byte string	被演算子を uuencode を用いて変換します
zlib_codec	zip, zlib	byte string	被演算子を gzip を用いて圧縮します。

#### 4.9.3 encodings.idna — アプリケーションにおける国際化ドメイン名 (IDNA)

2.3 で追加された仕様です。

このモジュールでは RFC 3490 (アプリケーションにおける国際化ドメイン名, IDNA: Internationalized Domain Names in Applications) および RFC 3492 (Nameprep: 国際化ドメイン名 (IDN) のための stringprep プロファイル) を実装しています。このモジュールは punycode エンコーディングおよび stringprep の上に構築されています。

これらの RFC はともに、ドメイン名における非 ASCII 文字をサポートするためのプロトコルを定義しています。 (“www.Alliancefranaise.nu” のような) 非 ASCII 文字を含むドメイン名は ASCII と互換性のあるエンコーディング (ACE, “www.xn-alliancefranaise-npb.nu” のような形式) に変換されます。ドメイン名の ACE 形式は、DNS クエリ、HTTP Host: フィールドなどといった、プロトコルが任意の文字を使うことができない全ての場所で用いられます。この変換はアプリケーション内で行われます; 可能ならユーザからは不可視となります: アプリケーションは Unicode ドメインラベルをワイヤ上に載せる際に IDNA に、ACE ドメインラベルをユーザに提供する前に Unicode に、それぞれ透過的に変換しなければなりません。

Python ではこの変換をいくつかの方法でサポートします: idna codec は Unicode と ACE 間の変換を行います。さらに、socket モジュールは Unicode ホスト名を ACE に透過的に変換するため、アプリケーションはホスト名を socket モジュールに渡す際にホスト名の変換に煩わされることがありません。その上で、ホスト名を関数パラメタとして持つ、httpplib や ftplib のようなモジュールでは Unicode ホスト名を受理します (httpplib でもまた、Host: フィールドにある IDNA ホスト名を、フィールド全体を送信する場合に透過的に送信します)。

(逆引きなどによって) ワイヤ越しにホスト名を受信する際、Unicode への自動変換は行われません: こうしたホスト名をユーザに提供したいアプリケーションでは、Unicode にデコードしてやる必要があります。

encodings.idna ではまた、nameprep 手続きを実装しています。nameprep はホスト名に対してある正規化を行って、国際化ドメイン名の大小文字非区別を達成するとともに、類似の文字を一元化します。nameprep 関数は必要なら直接使うこともできます。

**nameprep**(label)

label を nameprep したバージョンを返します。現在の実装ではクエリ文字列を仮定しているので、AllowUnassigned は真です。true.

**ToASCII**(*label*)

RFC 3490 で指定されているようにして、ラベルを ASCII に変換します。UseSTD3ASCIIRules は偽であると仮定します。

**ToUnicode**(*label*)

RFC 3490 で指定されているようにして、ラベルを Unicode に変換します。

## 4.10 unicodedata — Unicode データベース

このモジュールは、全ての Unicode 文字の属性を定義している Unicode 文字データベースへのアクセスを提供します。このデータベース内のデータは、<ftp://ftp.unicode.org/> で公開されている ‘UnicodeData.txt’ ファイルのバージョン 3.2.0 に基づいています。

このモジュールは、UnicodeData ファイルフォーマット 3.2.0 (<http://www.unicode.org/Public/UNIDATA/UnicodeData.html> を参照) で定義されているものと、同じ名前と記号を使います。このモジュールで定義されている関数は、以下のとおりです。

**lookup**(*name*)

名前に対応する文字を探します。その名前の文字が見つかった場合、その Unicode 文字が返されます。見つからなかった場合には、`KeyError` を発生させます。

**name**(*unichr*[, *default*])

Unicode 文字 *unichr* に付いている名前を、文字列で返します。名前が定義されていない場合には *default* が返されますが、この引数が与えられていなければ `ValueError` を発生させます。

**decimal**(*unichr*[, *default*])

Unicode 文字 *unichr* に割り当てられている十進数を、整数で返します。この値が定義されていない場合には *default* が返されますが、この引数が与えられていなければ `ValueError` を発生させます。

**digit**(*unichr*[, *default*])

Unicode 文字 *unichr* に割り当てられている二進数を、整数で返します。この値が定義されていない場合には *default* が返されますが、この引数が与えられていなければ `ValueError` を発生させます。

**numeric**(*unichr*[, *default*])

Unicode 文字 *unichr* に割り当てられている数値を、`float` 型で返します。この値が定義されていない場合には *default* が返されますが、この引数が与えられていなければ `ValueError` を発生させます。

**category**(*unichr*)

Unicode 文字 *unichr* に割り当てられた、汎用カテゴリを返します。

**bidirectional**(*unichr*)

Unicode 文字 *unichr* に割り当てられた、双方向カテゴリを返します。そのような値が定義されていない場合、空の文字列が返されます。

**combining**(*unichr*)

Unicode 文字 *unichr* に割り当てられた正規結合クラスを返します。結合クラス定義されていない場合、0 が返されます。

**mirrored**(*unichr*)

Unicode 文字 *unichr* に割り当てられた、鏡像化のプロパティを返します。その文字が双方向テキスト内で鏡像化された文字である場合には 1 を、それ以外の場合には 0 を返します。

**decomposition**(*unichr*)

Unicode 文字 *unichr* に割り当てられた、文字分解マッピングを、文字列型で返します。そのようなマッピングが定義されていない場合、空の文字列が返されます。

**normalize**(*form*, *unistr*)

Unicode 文字列 `unistr` の正規形 `form` を返します。`form` の有効な値は、'NFC'、'NFKC'、'NFD'、'NFKD' です。

Unicode 規格は標準等価性 (canonical equivalence) と互換等価性 (compatibility equivalence) に基づいて、様々な Unicode 文字列の正規形を定義します。Unicode では、複数の方法で表現できる文字があります。たとえば、文字 U+00C7 (LATIN CAPITAL LETTER C WITH CEDILLA) は、U+0043 (LATIN CAPITAL LETTER C) U+0327 (COMBINING CEDILLA) というシーケンスとしても表現できます。

各文字には、2 つの正規形があり、それぞれ正規形 C と正規形 D といいます。正規形 D (NFD) は標準分解 (canonical decomposition) としても知られており、各文字を分解された形に変換します。正規形 C (NFC) は標準分解を適用した後、結合済文字を再構成します。

互換等価性に基づいて、2 つの正規形が加えられています。Unicode では、一般に他の文字との統合がサポートされている文字があります。たとえば、U+2160 (ROMAN NUMERAL ONE) は事実上 U+0049 (LATIN CAPITAL LETTER I) と同じものです。しかし、Unicode では、既存の文字集合 (たとえば gb2312) との互換性のために、これがサポートされています。

正規形 KD (NFKD) は、互換分解 (compatibility decomposition) を適用します。すなわち、すべての互換文字を、等価な文字で置換します。正規形 KC (NFKC) は、互換分解を適用してから、標準分解を適用します。

2.3 で追加された仕様です。

更に、本モジュールは以下の定数を公開します。

`unidata_version`

このモジュールで使われている Unicode データベースのバージョン。

2.3 で追加された仕様です。

## 4.11 stringprep — インターネットのための文字列調製

(ホスト名のような) インターネット上にある存在に識別名をつける際、しばしば識別名間の“等価性”比較を行う必要があります。厳密には、例えば大小文字の区別をするかしないかといったように、比較をどのように行うかはアプリケーションの領域に依存します。また、例えば“印字可能な”文字で構成された識別名だけを許可するといったように、可能な識別名を制限することも必要となるかもしれません。

RFC 3454 では、インターネットプロトコル上で Unicode 文字列を“調製 (prepare)”するためのプロシジャを定義しています。文字列は通信路に載せられる前に調製プロシジャで処理され、その結果ある正規化された形式になります。RFC ではあるテーブルの集合を定義しており、それらはプロファイルにまとめられています。各プロファイルでは、どのテーブルを使い、stringprep プロシジャのどのオプション部分がプロファイルの一部になっているかを定義しています。stringprep プロファイルの一つの例は nameprep で、国際化されたドメイン名に使われます。

stringprep は RFC 3453 のテーブルを公開しているに過ぎません。これらのテーブルは辞書やリストとして表現するにはバリエーションが大きすぎるので、このモジュールでは Unicode 文字データベースを内部的に利用しています。モジュールソースコード自体は `mkstringprep.py` ユーティリティを使って生成されました。

その結果、これらのテーブルはデータ構造体ではなく、関数として公開されています。RFC には 2 種類のテーブル: 集合およびマップ、が存在します。集合については、stringprep は“特性関数 (characteristic function)”、すなわち引数が集合の一部である場合に真を返す関数を提供します。マップに対しては、マップ関数: キーが与えられると、それに関連付けられた値を返す関数、を提供します。以下はこのモジュールで利用可能な全ての関数を列挙したものです。



`in_table_a1(code)`

`code` がテーブル A.1 (Unicode 3.2 における未割り当てコード点: unassigned code point) かどうか判定します。

`in_table_b1(code)`

`code` がテーブル B.1 (一般には何にも対応付けられていない: commonly mapped to nothing) かどうか判定します。

`map_table_b2(code)`

テーブル B.2 (NFKC で用いられる大小文字の対応付け) に従って、`code` に対応付けられた値を返します。

`map_table_b3(code)`

テーブル B.3 (正規化を伴わない大小文字の対応付け) に従って、`code` に対応付けられた値を返します。

`in_table_c11(code)`

`code` がテーブル C.1.1 (ASCII スペース文字) かどうか判定します。

`in_table_c12(code)`

`code` がテーブル C.1.2 (非 ASCII スペース文字) かどうか判定します。

`in_table_c11_c12(code)`

`code` がテーブル C.1 (スペース文字、C.1.1 および C.1.2 の和集合) かどうか判定します。

`in_table_c21(code)`

`code` がテーブル C.2.1 (ASCII 制御文字) かどうか判定します。

`in_table_c22(code)`

`code` がテーブル C.2.2 (非 ASCII 制御文字) かどうか判定します。

`in_table_c21_c22(code)`

`code` がテーブル C.2 (制御文字、C.2.1 および C.2.2 の和集合) かどうか判定します。

`in_table_c3(code)`

`code` がテーブル C.3 (プライベート利用) かどうか判定します。

`in_table_c4(code)`

`code` がテーブル C.4 (非文字コード点: non-character code points) かどうか判定します。

`in_table_c5(code)`

`code` がテーブル C.5 (サロゲーションコード) かどうか判定します。

`in_table_c6(code)`

`code` がテーブル C.6 (平文: plain text として不適切) かどうか判定します。

`in_table_c7(code)`

`code` がテーブル C.7 (標準表現: canonical representation として不適切) かどうか判定します。

`in_table_c8(code)`

`code` がテーブル C.8 (表示プロパティの変更または撤廃) かどうか判定します。

`in_table_c9(code)`

`code` がテーブル C.9 (タグ文字) かどうか判定します。

`in_table_d1(code)`

`code` がテーブル D.1 (双方向プロパティ “R” または “AL” を持つ文字) かどうか判定します。

`in_table_d2(code)`

`code` がテーブル D.2 (双方向プロパティ “L” を持つ文字) かどうか判定します。



## 各種サービス

この章では、Python のすべてのバージョンで利用可能な各種サービスについて説明します。以下に概要を示します。

<code>pydoc</code>	ドキュメント生成とオンラインヘルプシステム
<code>doctest</code>	ドキュメンテーション文字列内の例を検証するためのフレームワーク。
<code>unittest</code>	単体テストフレームワーク
<code>test.test_support</code>	Python 回帰テストのサポート
<code>math</code>	数学関数 ( <code>sin()</code> など)。
<code>cmath</code>	複素数のための数学関数です。
<code>random</code>	よく知られている様々な分布をもつ擬似乱数を生成する。
<code>whrandom</code>	浮動少数点数の擬似乱数生成器。
<code>bisect</code>	バイナリサーチ用の配列二分法アルゴリズム。
<code>heapq</code>	ヒープキュー (別名優先度キュー) アルゴリズム。
<code>array</code>	Efficient arrays of uniformly typed numeric values.
<code>sets</code>	ユニークな要素の集合の実装
<code>itertools</code>	効率的なループ実行のためのイテレータ生成関数。
<code>ConfigParser</code>	Configuration file parser.
<code>fileinput</code>	Perl のような複数の入力ストリームをまたいだ行の繰り返し処理をサポートする (その場で偽)
<code>xreadlines</code>	ファイルの各行に対する効率のよい反復処理。
<code>calendar</code>	UNIX の <code>cal</code> プログラム相当の機能を含んだカレンダーに関する関数群
<code>cmd</code>	行指向のコマンドインタプリタを構築
<code>shlex</code>	UNIX シェル類似の言語に対する単純な字句解析。

### 5.1 pydoc — ドキュメント生成とオンラインヘルプシステム

2.1 で追加された仕様です。

`pydoc` モジュールは、Python モジュールから自動的にドキュメントを生成します。生成されたドキュメントをテキスト形式でコンソールに表示したり、Web browser にサーバとして提供したり、HTML ファイルとして保存したりできます。

組み込み関数の `help()` を使うことで、対話型のインタプリタからオンラインヘルプを起動することができます。コンソール用のテキスト形式のドキュメントをつくるのにオンラインヘルプでは `pydoc` を使っています。`pydoc` を Python インタプリタからではなく、オペレーティングシステムのコマンドプロンプトから起動した場合でも、同じテキスト形式のドキュメントを見ることができます。例えば、以下を shell から実行すると

```
pydoc sys
```

`sys` モジュールのドキュメントを、UNIX の `man` コマンドのような形式で表示させることができます。`pydoc` の引数として与えることができるのは、関数名・モジュール名・パッケージ名、また、モジュールやパッケージ内のモジュールに含まれるクラス・メソッド・関数へのドット"."形式での参照です。`pydoc` への引数がパスと解釈されるような場合で (オペレーティングシステムのパス区切り記号を含む場合です。例えば UNIX ならば "/" (スラッシュ) 含む場合があります)、さらに、そのパスが Python のソースファイルを指しているなら、そのファイルに対するドキュメントが生成されます。

引数の前に `-w` フラグを指定すると、コンソールにテキストを表示させるかわりにカレントディレクトリに HTML ドキュメントを生成します。

引数の前に `-k` フラグを指定すると、引数をキーワードとして利用可能な全てのモジュールの概要を検索します。検索のやりかたは、UNIX の `man` コマンドと同様です。モジュールの概要というのは、モジュールのドキュメントの一行目のことです。

また、`pydoc` を使うことでローカルマシンに Web browser から閲覧可能なドキュメントを提供する HTTP サーバーを起動することもできます。`pydoc -p 1234` とすると、HTTP サーバーをポート 1234 に起動します。これで、好きな Web browser を使って `http://localhost:1234/` からドキュメントを見ることができます。

`pydoc` でドキュメントを生成する場合、その時点での環境とパス情報に基づいてモジュールがどこにあるのか決定されます。そのため、`pydoc spam` を実行した場合につくられるドキュメントは、Python インタプリタを起動して `'import spam'` と入力したときに読み込まれるモジュールに対するドキュメントになります。

## 5.2 doctest — ドキュメンテーション文字列に本当のことが書かれているか調べる

`doctest` モジュールは、モジュールのドキュメンテーション文字列 (docstring) から、対話的 Python セッションのように見えるテキストを探しだし、これらのセッションを実際に実行して、そこに書かれている通りに動作するか検証します。以下に小さな、しかし完全な例を示します:

```

"""
This is module example.

Example supplies one function, factorial.  For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    If the result is small enough to fit in an int, return an int.
    Else return a long.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> [factorial(long(n)) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    2652528598121910586363084800000000L
    >>> factorial(30L)
    2652528598121910586363084800000000L
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    2652528598121910586363084800000000L

    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """

```

```

import math
if not n >= 0:
    raise ValueError("n must be >= 0")
if math.floor(n) != n:
    raise ValueError("n must be exact integer")
if n+1 == n: # catch a value like 1e300
    raise OverflowError("n too large")
result = 1
factor = 2
while factor <= n:
    try:
        result *= factor
    except OverflowError:
        result *= long(factor)
    factor += 1
return result

def _test():
    import doctest, example
    return doctest.testmod(example)

if __name__ == "__main__":
    _test()

```

‘example.py’ をコマンドラインから直接実行すると、doctest はその魔法を働かせます:

```

$ python example.py
$

```

出力は何もありません！しかしこれが正常で、全ての例が正しく動作することを意味しています。スクリーンに `-v` を与えると、doctest は何を行おうとしているのかを記録した詳細なログを出力し、最後にまとめを出力します:

```

$ python example.py -v
Running example.__doc__
Trying: factorial(5)
Expecting: 120
ok
0 of 1 examples failed in example.__doc__
Running example.factorial.__doc__
Trying: [factorial(n) for n in range(6)]
Expecting: [1, 1, 2, 6, 24, 120]
ok
Trying: [factorial(long(n)) for n in range(6)]
Expecting: [1, 1, 2, 6, 24, 120]
ok
Trying: factorial(30)
Expecting: 2652528598121910586363084800000000L
ok

```

And so on, eventually ending with:



```

Trying: factorial(1e100)
Expecting:
Traceback (most recent call last):
...
OverflowError: n too large
ok
0 of 8 examples failed in example.factorial.__doc__
2 items passed all tests:
  1 tests in example
  8 tests in example.factorial
9 tests in 2 items.
9 passed and 0 failed.
Test passed.
$

```

これが、doctest を使って生産性の向上を目指す上で知っておく必要があることの全てです！さあやってみましょう。‘doctest.py’ 内のドキュメンテーション文字列には doctest の全ての側面についての詳細な情報が入っており、ここではより重要な点をカバーするだけにします。

### 5.2.1 通常の利用法

通常の利用法では、各モジュール M の最後を、以下:

```

def _test():
    import doctest, M          # replace M with your module's name
    return doctest.testmod(M)  # ditto

if __name__ == "__main__":
    _test()

```

のようにして締めくくります。

もしテスト対象のモジュールがメインモジュールであるなら、testmod() に M を指定する必要はありません; この場合、実行中のモジュールがテストされます。

次に、モジュールをスクリプトとして走らせ、ドキュメンテーション文字列を実行して検証します:

```
python M.py
```

ドキュメンテーション文字列に書かれた例の実行が失敗しない限り、何も表示されません。失敗すると、失敗した例と、その原因が (場合によっては複数) 標準出力に印字され、出力の最後の行が ‘Test failed.’ となります。

一方、-v スイッチをつけて走らせると:

```
python M.py -v
```

実行を試みた全ての例について詳細に報告し、最後に各種まとめをおこなった内容が標準出力に印字されます。

verbose=1 を testmod() に渡せば、詳細報告 (verbose) モードを強制することができます。また、verbose=0 とすれば禁止することができます。どちらの場合にも、testmod() は sys.argv 上のスイッチを調べません。

いずれの場合も、`testmod()` は整数 2 要素、 $(f, t)$  からなるタプルを返し、 $f$  が失敗したドキュメンテーション文字列内の例、 $t$  が実行を試みた例の総数となります。

## 5.2.2 どのドキュメンテーション文字列が検証されるのか？

全ての詳細は `'docstring.py'` のドキュメンテーション文字列を参照してください。混乱させるようなことはありません: モジュールのドキュメンテーション文字列、全ての関数、クラスおよびメソッドのドキュメンテーション文字列が検索されます。オプションとして、テストはプライベートな名前をもつオブジェクトに添付されたドキュメンテーション文字列を除外することができます。モジュールに `import` されたオブジェクトは検索されません。

加えて、`M.__test__` が存在し、"真の値を持つ" 場合、この値は辞書で、辞書の各エントリは (文字列の) 名前を関数オブジェクト、クラスオブジェクト、または文字列に対応付けていなくてはなりません。`M.__test__` から得られた関数およびクラスオブジェクトのドキュメンテーション文字列は、その名前がプライベートなもので検索され、文字列の場合にはそれがドキュメンテーション文字列であるかのように直接検索を行います。出力においては、`M.__test__` におけるキー `K` は、

```
<name of M>.__test__.K
```

のように表示されます。

検索中に見つかったクラスも同様に再帰的に検索が行われ、クラスに含まれているメソッドおよびネストされたクラスについてドキュメンテーション文字列のテストが行われます。`M` 内のグローバル変数から到達したプライベートな名前はオプションでスキップされますが、`M.__test__` から到達した名前は全て検索されます。

## 5.2.3 実行コンテキストとは何か？

デフォルトでは、`testmod` がテストを行うべきドキュメンテーション文字列を見つけるたびに、`M` のグローバルをコピーして 使うため、モジュール上でテストを動作させてもモジュールの実際のグローバル変数を変更することはありません。このため、`M` 内でテストを行った際に痕跡が残る (left behind crumbs)、偶発的に別のテストが作動することはないはずですが、すなわち、例では `M` 内のトップレベルで定義されたどんな名前も、ドキュメンテーション文字列が動作する以前に定義された名前も自由に使うことができます。

`testmod()` に `globals=your_dict` を渡すことで、自前の辞書を実行コンテキストとして使うこともできます。おそらくこの値は `M.__dict__` を他の `import` されたモジュール由来のグローバル変数とマージしたものになるでしょう。

## 5.2.4 例外については？

例で生成される出力がトレースバックのみである限り問題ありません。例えば:

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: list.remove(x): x not in list
>>>
```

例外型と値だけが比較される (詳しくはトレースバックの最後の行だけ) が比較されるので注意してください。途中の様々な "File" 行は (例の中のドキュメンテーション値に明示的に追加しない限り) 放っておか

れます。

## 5.2.5 進んだ使い方

doctest をどのように動作させるかを制御する、いくつかのモジュールレベルの関数が利用できます。

**debug**(*module*, *name*)

doctest を含む単一のドキュメンテーション文字列をデバッグします。

デバッグしたいドキュメンテーション文字列の入った *module* (またはドットで区切ったモジュール名) と、(モジュール内の) デバッグしたいドキュメンテーション文字列を持つオブジェクトの *name* を指定してください。

doctest の例が展開され (testsource() 関数を参照してください)、一次ファイルに書き込まれます。次に Python デバッガ pdb がこのファイルに対して起動されます。2.3 で追加された仕様です。

**testmod**()

この関数は doctest への基本的なインタフェースを提供します。この関数は Tester のローカルなインスタンスを生成し、このクラスの適切なメソッドを動作させ、結果をグローバルな Tester インスタンスである master に統合します。

testmod() が提供するよりも細かい制御を行うには、Tester のインスタンスを自作のポリシで作成するか、master のメソッドを直接呼び出します。詳細は Tester.\_\_doc\_\_ を参照してください。

**testsource**(*module*, *name*)

doctest の例をドキュメンテーション文字列から展開します。

展開したいテストの入った *module* (またはドットで区切られたモジュールの名前) と、展開したいテストの入った docstring を持つオブジェクトの (モジュール内の) *name* を与えます。

doctest 内の例は Python コードの入った文字列として返されます。例中での予想される出力のブロックは Python のコメントに変換されます。2.3 で追加された仕様です。

**DocTestSuite**(*[module]*)

モジュールにおける doctest のテストプログラムを unittest.TestSuite に変換します。

返される TestSuite は unittest フレームワークで動作するためのもので、モジュール内の各 doctest を走らせます。doctest のいずれかが失敗すると、生成された unittest が失敗し、該当するテストを含むファイルと (時に近似の) 行番号を表示する DocTestTestFailure 例外が送出されます。

オプションの *module* 引数はテストするモジュールを与えます。この値はモジュールオブジェクトか (場合によってはドットで区切られた) モジュール名となります。指定されていないければ、この関数を呼び出しているモジュールが使われます。

unittest モジュールが TestSuite を利用する数多くの方法のうちの一つを使った例を以下に示します:

```
import unittest
import doctest
import my_module_with_doctests

suite = doctest.DocTestSuite(my_module_with_doctests)
runner = unittest.TextTestRunner()
runner.run(suite)
```

2.3 で追加された仕様です。 警告: この関数は現在のところ `M.__test__` を検索せず、その検索テクニックはあらゆる点で `testmod()` と合致しません。将来のバージョンではこれら二つを収斂させる予定です。

## 5.2.6 ドキュメンテーション文字列内の例をどうやって認識するのか?

ほとんどの場合、対話コンソールセッション上でのコピー / ペーストはうまく動作します — 先頭部の空白が厳密に一貫性があるようにしてください (きちんとやるのが面倒ならタブとスペースを混在させることはできますが、`doctest` はあなたにとってのタブがどういう意味を持つのかを推測するようなことはしません)。

```
>>> # comments are ignored
>>> x = 12
>>> x
12
>>> if x == 13:
...     print "yes"
... else:
...     print "no"
...     print "NO"
...     print "NO!!!"
...
no
NO
NO!!!
>>>
```

出力結果例 (expected output) は、コードを含む最後の '`>>>`' or '`...`' 行の直下に続きます。また、出力結果例 (がある場合) は、次の '`>>>`' 行か、全て空白文字の行まで続きます。

細目事項:

- 出力結果例には、全て空白の行が入ってはいけません。そのような行は出力結果例の終了を表すと見なされるからです。
- `stdout` への出力は取り込まれますが、`stderr` は取り込まれません (例外発生時のトレースバックは別の方法で取り込まれます)。
- 対話セッションにおいて、バックスラッシュを用いて次の行に続ける場合や、その他の理由でバックスラッシュを用いる場合、ドキュメンテーション文字列内ではバックスラッシュを二重にしておく必要があります。これは、単に例を記述しているのが文字列内であり、バックスラッシュをそのままにしておくためにはエスケープする必要があるからです。例えば以下のように:

```
>>> if "yes" == "\\
...     "y" +   \\
...     "es":
...     print 'yes'
yes
```

- 開始カラムはどこでもかまいません:

```
>>> assert "Easy!"
>>> import math
>>> math.floor(1.9)
1.0
```

出力結果例の先頭部にある空白文字列は、出力をトリガしている '`>>>`' 行の先頭にある空白文字列と同じだけはぎとられます。

## 5.2.7 注意事項

1. `doctest` は出力結果例が厳密に一致するように厳しく要求します。1 文字でも一致しないと、テストは失敗します。このため、Python が出力に関して何を保証していて、何を保証していないかを正確に知っていないと幾度か混乱させられることでしょう。例えば、辞書を出力する際、Python はキーと値のペアが常に特定の順番で並ぶよう保証してはいません。従って、以下のようなテスト

```
>>> foo()
{"Hermione": "hippogryph", "Harry": "broomstick"}
>>>
```

は脆弱です! 回避法としては、

```
>>> foo() == {"Hermione": "hippogryph", "Harry": "broomstick"}
True
>>>
```

としてください。別のやり方としては、

```
>>> d = foo().items()
>>> d.sort()
>>> d
[('Harry', 'broomstick'), ('Hermione', 'hippogryph')]
```

のようになります。

他のやり方もありますが、自分で考えてみてください。

もう一つのよくない発想は、

```
>>> id(1.0) # certain to fail some of the time
7948648
>>>
```

のようなオブジェクトアドレスの埋め込みです。

Python は浮動小数点の書式化をプラットフォームの C ライブラリにゆだねており、C ライブラリはこの点においてプラットフォーム間で非常に違いが大きいため、浮動小数点数もまたプラットフォーム間での微妙な出力の違いの原因となります。

```
>>> 1./7 # risky
0.14285714285714285
>>> print 1./7 # safer
0.142857142857
>>> print round(1./7, 6) # much safer
0.142857
```

`I/2.**J` の形式になる数値は全てのプラットフォームで安全であり、私はしばしば `doctest` の例にはその形式の数値を使っています:

```
>>> 3./4 # utterly safe
0.75
```

単純な分数は人が理解し易いこともあり、よりよいドキュメント記述になります。

2. 一度だけしか実行してはならないコードには気をつけてください。

一度だけしか実行してはならないようなモジュールレベルのコードがあるなら、`_test()` のフルプルーフ性の高い定義は

```
def _test():
    import doctest, sys
    doctest.testmod()
```

のようになります。

## 5.2.8 提言

`doctest` の最初の言葉が "doc" になっていますが、この言葉、すなわちドキュメントを最新に保つこと: こそが、作者が `doctest` を書いた理由です。たまたま `doctest` はテスト環境として面白いユニットになっていますが、それが第一目的なわけではありません。

`docstring` の例は注意深く作成してください。これには学習を必要とする一種の芸術が存在します — 最初はすんなりできないでしょう。例というものはドキュメントに紛れ無しの価値を与えます。よい例はしばしばいくつもの言葉に値します。可能なら、通常の場合の例、境界条件的な場合の例、興味深い微妙な例を示し、そして送出されうる例外各々についての例を示してください。おそらく境界条件的な例や微妙な例を対話シェルでテストしていることでしょう: `doctest` ではこうしたセッションを取り込み、かつ今後ずっと設計通りに動作するよう検証する作業を可能な限り簡単にしたいと思っています。

注意深くやれば、例はユーザにとってはあまり意味のないものになるかもしれませんが、歳を経るにつれて、あるいは "状況が変わった" 際に何度も何度も正しく動作させるためにかかることになる時間を節約する形で見返りは選られます。私は今でも、自分の `doctest` で処理した例が "たいした事のない" 変更を行った際にうまく動作しなくなることに驚いています。

網羅的なテストや、ドキュメントに対する付加価値のまったくない退屈な例のテストについては、`__test__` 辞書の方に定義してください。`__test__` はそのための辞書です。

## 5.3 unittest — 単体テストフレームワーク

2.1 で追加された仕様です。



この Python 単体テストフレームワークは“PyUnit”とも呼ばれ、Kent Beck と Erich Gamma による JUnit の Python 版です。JUnit はまた Kent の Smalltalk 用テストフレームワークの Java 版で、どちらもそれぞれの言語で業界標準の単体テストフレームワークとなっています。

PyUnit では、テストの自動化・初期設定と終了処理の共有・テストの分類・テスト実行と結果レポートの分離などの機能を提供しており、`unittest` のクラスを使って簡単にたくさんのテストを開発できるようになっています。

PyUnit では、テストを以下のような構成で開発します。

## Fixture

*test fixture*(テスト設備)とは、テスト実行のために必要な準備や終了処理を指します。例:テスト用データベースの作成・ディレクトリ・サーバプロセスの起動など。

## テストケース

テストケースはテストの最小単位で、各入力に対する結果をチェックします。PyUnit では、`TestCase` を基底クラスとしてテストケースを作成します。

## テストスイート

テストスイートはテストケースとテストスイートの集まりで、同時に実行しなければならないテストをまとめる場合に使用します。

## テストランナー

テストランナーはテストの実行と結果表示を管理するコンポーネントです。ランナーはグラフィカルインターフェースでもテキストインターフェースでも良いですし、何も表示せずにテスト結果を示す値を返すだけの場合もあります。

PyUnit では、テストケースと fixture を、`TestCase` クラスと `FunctionTestCase` クラスで提供しています。`TestCase` クラスは新規にテストを作成する場合に使用し、`FunctionTestCase` は既存のテストを PyUnit に組み込む場合に使用します。fixture の設定処理と終了処理は、`TestCase` では `setUp()` メソッドと `tearDown()` をオーバーライドして記述し、`classFunctionTestCase` では初期設定・終了処理を行う既存の関数をコンストラクタで指定します。テスト実行時、まず fixture の初期設定が最初に実行されます。初期設定が正常終了した場合、テスト実行後にはテスト結果に関わらず終了処理が実行されます。`TestCase` の各インスタンスが実行するテストは一つだけで、fixture は各テストごとに新しく作成されます。

テストスイートは `TestSuite` クラスで実装されており、複数のテストとテストスイートをまとめる事ができます。テストスイートを実行すると、スイートと子スイートに追加されている全てのテストが実行されます。

テストランナーは `run()` メソッドを持つオブジェクトで、`run()` は引数として `TestCase` か `TestSuite` オブジェクトを受け取り、テスト結果を `TestResult` オブジェクトで戻します。PyUnit ではデフォルトでテスト結果を標準エラーに出力する `TextTestRunner` をサンプルとして実装しています。これ以外のランナー (グラフィックインターフェース用など) を実装する場合でも、特定のクラスから派生する必要はありません。

参考資料:

*PyUnit Web Site*

(<http://pyunit.sourceforge.net/>)

The source for further information on PyUnit.

*Simple Smalltalk Testing: With Patterns*

(<http://www.XProgramming.com/testfram.htm>)

Kent Beck's original paper on testing frameworks using the pattern shared by unittest.

### 5.3.1 基礎的な例

unittest モジュールには、テストの開発や実行の為に優れたツールが用意されており、この節では、その一部を紹介します。ほとんどのユーザにとっては、ここで紹介するツールだけで十分でしょう。

以下は、random モジュールの三つの関数をテストするスクリプトです。

```
import random
import unittest

class TestSequenceFunctions(unittest.TestCase):

    def setUp(self):
        self.seq = range(10)

    def testshuffle(self):
        # make sure the shuffled sequence does not lose any elements
        random.shuffle(self.seq)
        self.seq.sort()
        self.assertEqual(self.seq, range(10))

    def testchoice(self):
        element = random.choice(self.seq)
        self.assertIn(element, self.seq)

    def testsample(self):
        self.assertRaises(ValueError, random.sample, self.seq, 20)
        for element in random.sample(self.seq, 5):
            self.assertIn(element, self.seq)

if __name__ == '__main__':
    unittest.main()
```

テストケースは、unittest.TestCase のサブクラスとして作成します。メソッド名が test で始まる三つのメソッドがテストです。テストランナーはこの命名規約によってテストを行うメソッドを検索します。

これらのテスト内では、予定の結果が得られていることを確かめるために assertEquals() を、条件のチェックに assert\_() を、例外が発生する事を確認するために assertRaises() をそれぞれ呼び出しています。assert 文の代わりにこれらのメソッドを使用すると、テストランナーでテスト結果を集計してレポートを作成する事ができます。

setUp() メソッドが定義されている場合、テストランナーは各テストを実行する前に setUp() メソッドを呼び出します。同様に、tearDown() メソッドが定義されている場合は各テストの実行後に呼び出します。上のサンプルでは、それぞれのテスト用に新しいシーケンスを作成するために setUp() を使用しています。

サンプルの末尾が、簡単なテストの実行方法です。unittest.main() は、テストスクリプトのコマンドライン用インターフェースです。コマンドラインから起動された場合、上記のスクリプトから以下のような結果が出力されます:

```
...
-----
Ran 3 tests in 0.000s

OK
```

簡略化した結果を出力したり、コマンドライン以外からも起動する等のより細かい制御が必要であれば、`unittest.main()` を使用せずに別の方法でテストを実行します。例えば、上記サンプルの最後の2行は以下のように書くことができます:

```
suite = unittest.TestSuite()
suite.addTest(unittest.makeSuite(TestSequenceFunctions))
unittest.TextTestRunner(verbosity=2).run(suite)
```

変更後のスクリプトをインタプリタや別のスクリプトから実行すると、以下の出力が得られます:

```
testchoice (__main__.TestSequenceFunctions) ... ok
testsample (__main__.TestSequenceFunctions) ... ok
testshuffle (__main__.TestSequenceFunctions) ... ok

-----
Ran 3 tests in 0.110s

OK
```

以上が `unittest` モジュールでよく使われる機能で、ほとんどのテストではこれだけでも十分です。基礎となる概念や全ての機能については以降の章を参照してください。

### 5.3.2 テストの構成

単体テストの基礎となる構築要素は、テストケース — セットアップと正しさのチェックを行う、独立したシナリオ — です。PyUnit では、テストケースは `unittest` モジュールの `TestCase` クラスのインスタンスで示します。テストケースを作成するには `TestCase` のサブクラスを記述するか、または `FunctionTestCase` を使用します。

`TestCase` から派生したクラスのインスタンスは、このオブジェクトだけで一件のテストと初期設定・終了処理を行います。

`TestCase` インスタンスは外部から完全に独立し、単独で実行する事も、他の任意のテストと一緒に実行する事もできなければなりません。

以下のように、`runTest()` をオーバーライドし、必要なテスト処理を記述するだけで簡単なテストケースを書くことができます:

```
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def runTest(self):
        widget = Widget("The widget")
        self.failUnless(widget.size() == (50,50), 'incorrect default size')
```

何らかのテストを行う場合、ベースクラス `TestCase` の `assert*()` か `fail*()` メソッドを使用してください。テストケース実行時、テストが失敗すると例外が送出され、テストフレームワークはテスト結

果を *failure* とします。 `assert*()` と `fail*()` 以外からの例外が発生した場合、テスト結果は *errors* となります。

テストの実行方法については後述とし、まずはテストケースインスタンスの作成方法を示します。テストケースインスタンスは、以下のように引数なしでコンストラクタを呼び出して作成します。

```
testCase = DefaultWidgetSizeTestCase()
```

似たようなテストを数多く行う場合、同じ環境設定処理を何度も必要となります。例えば上記のような Widget のテストが 100 種類も必要な場合、それぞれのサブクラスで “Widget” オブジェクトを生成する処理を記述するのは好ましくありません。

このような場合、初期化処理は `setUp()` メソッドに切り出し、テスト実行時にテストフレームワークが自動的に実行するようにすることができます：

```
import unittest

class SimpleWidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget("The widget")

class DefaultWidgetSizeTestCase(SimpleWidgetTestCase):
    def runTest(self):
        self.failUnless(self.widget.size() == (50,50),
                        'incorrect default size')

class WidgetResizeTestCase(SimpleWidgetTestCase):
    def runTest(self):
        self.widget.resize(100,150)
        self.failUnless(self.widget.size() == (100,150),
                        'wrong size after resize')
```

テスト中に `setUp()` メソッドで例外が発生した場合、テストフレームワークはテストを実行することができないとみなし、`runTest()` を実行しません。

同様に、終了処理を `tearDown()` メソッドに記述すると、`runTest()` メソッド終了後に実行されます：

```
import unittest

class SimpleWidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget("The widget")

    def tearDown(self):
        self.widget.dispose()
        self.widget = None
```

`setUp()` が正常終了した場合、`runTest()` の結果に関わり無く `tearDown()` が実行されます。

このような、テストを実行する環境を *fixture* と呼びます。

JUnit では、多数の小さなテストケースを同じテスト環境で実行する場合、全てのテストについて `DefaultWidgetSizeTestCase` のような `SimpleWidgetTestCase` のサブクラスを作成する必要があります。これは時間のかかる、うんざりする作業ですので、PyUnit ではより簡単なメカニズムを用意しています：

```

import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget("The widget")

    def tearDown(self):
        self.widget.dispose()
        self.widget = None

    def testDefaultSize(self):
        self.failUnless(self.widget.size() == (50,50),
            'incorrect default size')

    def testResize(self):
        self.widget.resize(100,150)
        self.failUnless(self.widget.size() == (100,150),
            'wrong size after resize')

```

この例では `runTest()` がありませんが、二つのテストメソッドを定義しています。このクラスのインスタンスは `test*()` メソッドのどちらか一方の実行と、`self.widget` の生成・解放を行います。この場合、テストケースインスタンス生成時に、コンストラクタの引数として実行するメソッド名を指定します:

```

defaultSizeTestCase = WidgetTestCase("testDefaultSize")
resizeTestCase = WidgetTestCase("testResize")

```

PyUnit ではテストスイートによってテストケースインスタンスをテスト対象の機能によってグループ化することができます。テストスイートは、`unittest` の `TestSuite` クラスで作成します。

```

widgetTestSuite = unittest.TestSuite()
widgetTestSuite.addTest(WidgetTestCase("testDefaultSize"))
widgetTestSuite.addTest(WidgetTestCase("testResize"))

```

各テストモジュールで、テストケースを組み込んだテストスイートオブジェクトを作成する呼び出し可能オブジェクトを用意しておくと、テストの実行や参照が容易になります:

```

def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase("testDefaultSize"))
    suite.addTest(WidgetTestCase("testResize"))
    return suite

```

または:

```

class WidgetTestSuite(unittest.TestSuite):
    def __init__(self):
        unittest.TestSuite.__init__(self, map(WidgetTestCase,
            ("testDefaultSize",
             "testResize")))

```

(小心者は前者を使うべし)

一般的に、`TestCase` のサブクラスには良く似た名前のテスト関数が複数定義されます。そこで `unittest` モジュールには、テストケースクラスの全テストケースを使ってテストスイートを作成する `makeSuite()`

関数を用意しています。

```
suite = unittest.makeSuite(WidgetTestCase, 'test')
```

`makeSuite()` でテストスイートを作成した場合、テストケースの実行順序はテストケース関数名を `cmp()` 組み込み関数でソートした順番となります。

システム全体のテストを行う場合など、テストスイートをさらにグループ化したい場合がありますが、このような場合、`TestSuite` インスタンスには `TestSuite` と同じように `TestSuite` を追加する事ができます。

```
suite1 = module1.TheTestSuite()
suite2 = module2.TheTestSuite()
alltests = unittest.TestSuite((suite1, suite2))
```

テストケースやテストスイートは ('widget.py' のような) テスト対象のモジュール内にも記述できますが、テストは ('widgettests.py' のような) 独立したモジュールに置いた方が以下のような点で有利です:

- テストモジュールだけをコマンドラインから実行することができる。
- テストコードと出荷するコードを分離する事ができる。
- テストコードを、テスト対象のコードに合わせて修正する誘惑に駆られにくい。
- テストコードは、テスト対象コードほど頻繁に更新されない。
- テストコードをより簡単にリファクタリングすることができる。
- C で書いたモジュールのテストは、どっちにしろ独立したモジュールとなる。
- テスト戦略を変更した場合でも、ソースコードを変更する必要がない。

### 5.3.3 既存テストコードの再利用

既存のテストコードが有るとき、このテストを PyUnit で実行しようとする為に古いテスト関数をいちいち `TestCase` クラスのサブクラスに変換するのは大変です。

このような場合は、`TestCase` のサブクラスである `FunctionTestCase` クラスを使い、既存のテスト関数をラップします。初期設定と終了処理をラップする事もできます。

以下のテストコードがあった場合:

```
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

テストケースインスタンスは次のように作成します:

```
testcase = unittest.FunctionTestCase(testSomething)
```

初期設定、終了処理が必要な場合は、次のように指定します:



```

testcase = unittest.FunctionTestCase(testSomething,
                                     setUp=makeSomethingDB,
                                     tearDown=deleteSomethingDB)

```

注意: PyUnit は `AssertionError` によるテストの失敗検出もサポートしていますが、推奨されません。将来のバージョンでは、`AssertionError` は別の目的に使用される可能性が有ります。

### 5.3.4 クラスと関数

**class TestCase()**

`TestCase` クラスのインスタンスは、テストの最小実行単位を示します。このクラスをベースクラスとして使用し、必要なテストを具象サブクラスに実装します。`TestCase` クラスでは、テストランナーがテストを実行するためのインターフェースと、各種のチェックやテスト失敗をレポートするためのメソッドを実装しています。

**class FunctionTestCase(*testFunc*[, *setUp*[, *tearDown*[, *description*]]])**

このクラスでは `TestCase` インターフェースの内、テストランナーがテストを実行するためのインターフェースだけを実装しており、テスト結果のチェックやレポートに関するメソッドは実装していません。既存のテストコードを `unittest` によるテストフレームワークに組み込むために使用します。

**class TestSuite([*tests*])**

このクラスは、個々のテストケースやテストスイートの集約を示します。通常のテストケースと同じようにテストランナーで実行すると、テストスイート内の全てのテストケースとテストスイートを実行します。テストケース・テストスイートを追加するためのメソッドを用意しています。*tests* には、スイートに追加するテストのシーケンスを指定する事ができます。

**class TestLoader()**

モジュールまたは `TestCase` クラスから、指定した条件に従ってテストをロードし、`TestSuite` にラップして返します。モジュールからテストをロードする場合、全ての `TestCase` 派生クラスを抽出し、名前が 'test' で始まる全てのメソッドのインスタンスを作成します。

**defaultTestLoader**

`TestLoader` のインスタンスで、共用する事ができます。`TestLoader` をカスタマイズする必要がなければ、新しい `TestLoader` オブジェクトを作らずにこのインスタンスを使用します。

**class TextTestRunner([*stream*[, *descriptions*[, *verbosity*]]])**

実行結果を標準結果に出力する、単純なテストランナー。いくつかの設定項目がありますが、非常に単純です。グラフィカルなテスト実行アプリケーションでは、独自のテストランナーを作成してください。

**main([*module*[, *defaultTest*[, *argv*[, *testRunner*[, *testRunner*]]]])**

テストを実行するためのコマンドラインプログラム。この関数を使えば、次のように簡単に実行可能なテストモジュールを作成する事ができます。

```

if __name__ == '__main__':
    unittest.main()

```

場合によっては、`doctest` モジュールを使って書かれた既存のテストがあります。その場合、モジュールは既存のテストコードから `unittest.TestSuite` インスタンスを自動的に構築できる `DocTestSuite` クラスを提供します。2.3 で追加された仕様です。

### 5.3.5 TestCase オブジェクト

TestCase クラスのインスタンスは個別のテストをあらわすオブジェクトですが、TestCase の具象サブクラスには複数のテストを定義する事ができます — 具象サブクラスは、特定の fixture(テスト設備)を示している、と考えてください。fixture は、それぞれのテストケースごとに作成・解放されます。

TestCase インスタンスには、次の 3 種類のメソッドがあります: テストを実行するためのメソッド・条件のチェックやテスト失敗のレポートのためのメソッド・テストの情報収集に使用する問い合わせメソッド。

テストを実行するためのメソッドを以下に示します:

**setUp()**

テストを実行する直前に、fixture を作成する為に呼び出されます。このメソッドを実行中に例外が発生した場合、テストの失敗ではなくエラーとされます。デフォルトの実装では何も行いません。

**tearDown()**

テストを実行し、結果を記録した直後に呼び出されます。テスト実行中に例外が発生しても呼び出されますので、内部状態に注意して処理を行ってください。メソッドを実行中に例外が発生した場合、テストの失敗ではなくエラーとみなされます。このメソッドは、setUp() が正常終了した場合にはテストメソッドの実行結果に関わり無く呼び出されます。デフォルトの実装では何も行いません。

**run([result])**

テストを実行し、テスト結果を *result* に指定されたテスト結果オブジェクトに収集します。*result* が None か省略された場合、一時的な結果オブジェクトを生成して使用しますが呼び出し元には渡されません。このメソッドは、TestCase インスタンスの呼び出しと等価です。

**debug()**

テスト結果を収集せずにテストを実行します。例外が呼び出し元に通知されるため、テストをデバッグで実行することができます。

テスト結果のチェックとレポートには、以下のメソッドを使用してください。

**assert\_(expr[, msg])**

**failUnless(expr[, msg])**

*expr* が偽の場合、テスト失敗を通知します。*msg* にはエラーの説明を指定するか、または None を指定してください。

**assertEqual(first, second[, msg])**

**failUnlessEqual(first, second[, msg])**

*first* と *second* が等しくない場合、テスト失敗を通知します。エラー内容は *msg* に指定された値か、または None となります。failUnlessEqual() では *msg* のデフォルト値は *first* と *second* を含んだ文字列となりますので、failUnless() の第一引数に比較の結果を指定するよりも便利です。

**assertNotEqual(first, second[, msg])**

**failIfEqual(first, second[, msg])**

*first* と *second* が等しい場合、テスト失敗を通知します。エラー内容は *msg* に指定された値か、または None となります。failUnlessEqual() では *msg* のデフォルト値は *first* と *second* を含んだ文字列となりますので、failUnless() の第一引数に比較の結果を指定するよりも便利です。

**assertAlmostEqual(first, second[, places[, msg]])**

**failUnlessAlmostEqual(first, second[, places[, msg]])**

*first* と *second* を *places* で与えた少数位で値を丸めて差分を計算し、ゼロと比較することで、近似的に等価であるかどうかをテストします。指定少数位の比較というのは指定有効桁数の比較ではないので注意してください。値の比較結果が等しくなかった場合、テストは失敗し、*msg* で与えた説明か、None を返します。

**assertNotAlmostEqual(first, second[, places[, msg]])**

**failIfAlmostEqual**(*first*, *second*[, *places*[, *msg*]])

*first* と *second* を *places* で与えた少数位で値を丸めて差分を計算し、ゼロと比較することで、近似的に等価でないかどうかをテストします。指定少数位の比較というものは指定有効桁数の比較ではないので注意してください。値の比較結果が等しかった場合、テストは失敗し、*msg* で与えた説明か、*None* を返します。

**assertRaises**(*exception*, *callable*, ...)

**failUnlessRaises**(*exception*, *callable*, ...)

*callable* を呼び出し、発生した例外をテストします。assertRaises() には、任意の位置パラメータとキーワードパラメータを指定する事ができます。*exception* で指定した例外が発生した場合はテスト成功とし、それ以外の例外が発生するか例外が発生しない場合にテスト失敗となります。複数の例外を指定する場合には、例外クラスのタプルを *exception* に指定します。

**failIf**(*expr*[, *msg*])

failIf() は failUnless() の逆で、*expr* が真の場合、テスト失敗を通知します。エラー内容は *msg* に指定された値か、または *None* となります。

**fail**([*msg*])

無条件にテスト失敗を通知します。エラー内容は *msg* に指定された値か、または *None* となります。

**failureException**

test() メソッドが送出する例外を指定するクラス属性。テストフレームワークで追加情報を持つ等の特殊な例外を使用する場合、この例外のサブクラスとして作成します。この属性の初期値は AssertionError です。

テストフレームワークは、テスト情報を収集するために以下のメソッドを使用します:

**countTestCases**()

テストオブジェクトに含まれるテストの数を返します。TestCase インスタンスは常に 1 を返します。TestSuite クラスでは 1 以上を返します。

**defaultTestResult**()

オブジェクトが実行するテストの、デフォルトのテスト結果オブジェクトの型を返します。

**id**()

テストケースを特定する文字列を返します。通常、*id* はモジュール名・クラス名を含む、テストメソッドのフルネームを指定します。

**shortDescription**()

テストの説明を一行分、または説明がない場合には *None* を返します。デフォルトでは、テストメソッドの docstring の先頭の一行、または *None* を返します。

### 5.3.6 TestSuite オブジェクト

TestSuite オブジェクトは TestCase とよく似た動作をしますが、実際のテストは実装せず、一まとめに実行するテストのグループをまとめるために使用します。TestSuite には以下のメソッドが追加されています:

**addTest**(*test*)

TestCase 又は TestSuite のインスタンスをスイートに追加します。

**addTests**(*tests*)

シーケンス *test* に含まれる全ての TestCase 又は TestSuite のインスタンスをスイートに追加します。

run() メソッドは TestCase の run() メソッドと若干異なります:

**run(*result*)**

スイート内のテストを実行し、結果を *result* で指定した結果オブジェクトに収集します。TestCase.run() と異なり、TestSuite.run() では必ず結果オブジェクトを指定する必要があります。

通常、TestSuite の run() メソッドは TestRunner が起動するため、ユーザが直接実行する必要はありません。

### 5.3.7 TestResult オブジェクト

TestResult は、複数のテスト結果を記録します。TestCase クラスと TestSuite クラスのテスト結果を正しく記録しますので、テスト開発者が独自にテスト結果を管理する処理を開発する必要はありません。

unittest を利用したテストフレームワークでは、TestRunner.run() が返す TestResult インスタンスを参照し、テスト結果をレポートします。

TestResult インスタンスは、テストの実行件数と、テスト中に発生した失敗・エラーの情報を (*testcase*, *traceback*) のタプルで保持します。*traceback* は例外のトレースバック情報をフォーマットした文字列です。

以下の属性は、テストの実行結果を検査する際に使用することができます：

**errors**

テスト中に発生した例外の内、テスト失敗ではなくエラーとなった例外の情報のリスト。リストの要素は、TestCase と例外のトレースバック情報をフォーマットした文字列の組となります。2.2 で変更された仕様: sys.exc\_info() の結果ではなく、フォーマットしたトレースバックを保存

**failures**

テスト中に発生した失敗の情報のリスト。リストの要素は、TestCase と例外のトレースバック情報をフォーマットした文字列の組となります。2.2 で変更された仕様: sys.exc\_info() の結果ではなく、フォーマットしたトレースバックを保存

**testsRun**

開始したテストの数。

**wasSuccessful()**

これまでに実行したテストが全て成功していれば True を、それ以外なら False を返す。

以下のメソッドは内部データ管理用のメソッドですが、対話的にテスト結果をレポートするテストツールを開発する場合などにはサブクラスで拡張することができます。

**startTest(*test*)**

*test* を実行する直前に呼び出されます。

**stopTest(*test*)**

*test* の実行直後に、テスト結果に関わらず呼び出されます。

**addError(*test*, *err*)**

テスト実行中に、テストの失敗以外の例外が発生した場合に呼び出されます。*err* は sys.exc\_info() が返すタプル (*type*, *value*, *traceback*) です。

**addFailure(*test*, *err*)**

テストが失敗した場合に呼び出されます。*err* は sys.exc\_info() が返すタプル (*type*, *value*, *traceback*) です。

**addSuccess(*test*)**

テストが失敗しなかった場合に呼び出されます。*test* には、テストケースオブジェクトが指定されます。

TestResult オブジェクトには、さらにもう一つのメソッドがあります：

**stop()**

テスト中断のシグナルを送ります。このメソッドが呼び出されると、テストランナーは以降のテスト実行を中止し、呼び出し元に復帰します。TextTestRunner ではキーボードからの割り込みでテストを中断するためにこのメソッドを使用しており、独自のランナーを実装する場合にも同じように使用することができます。

### 5.3.8 TestLoader オブジェクト

TestLoader クラスは、クラスやモジュールからテストスイートを作成するために使用します。通常はこのクラスのインスタンスを作成する必要はなく、unittest モジュールのモジュール属性 defaultTestLoader を共用インスタンスとして使用することができます。

TestLoader オブジェクトには以下のメソッドがあります：

**loadTestsFromTestCase**( *testCaseClass* )

TestCase の派生クラス *testCaseClass* に含まれる全テストケースのスイートを返します。

**loadTestsFromModule**( *module* )

指定したモジュールに含まれる全テストケースのスイートを返します。このメソッドは *module* 内の TestCase 派生クラスを検索し、見つかったクラスのテストメソッドごとにクラスのインスタンスを作成します。

警告: TestCase クラスを基底クラスとしてクラス階層を構築すると fixture や補助的な関数をうまく共用することができますが、基底クラスに直接インスタンス化できないテストメソッドがあると、この loadTestsFromModule を使うことができません。この場合でも、fixture が全て別々で定義がサブクラスにある場合は使用することができます。

**loadTestsFromName**( *name*[, *module*] )

文字列で指定される全テストケースを含むスイートを返します。

*name* には“ドット修飾名”でモジュールかテストケースクラス、または TestCase か TestSuite のインスタンスを返す呼び出し可能オブジェクトを指定します。例えば SampleTests モジュールに TestCase から派生した SampleTestCase クラスがあり、SampleTestCase にはテストメソッド test\_one()・test\_two()・test\_three() があるとします。この場合、*name* に 'SampleTests.SampleTestCase' と指定すると、SampleTestCase の三つのテストメソッドを実行するテストスイートが作成されます。'SampleTests.SampleTestCase.test\_two' と指定すれば、test\_two() だけを実行するテストスイートが作成されます。インポートされていないモジュールやパッケージ名を含んだ名前を指定した場合は自動的にインポートされます。

また、*module* を指定した場合、*module* 内の *name* を取得します。

**loadTestsFromNames**( *names*[, *module*] )

loadTestsFromName() と同じですが、名前を一つだけ指定するのではなく、複数の名前のシーケンスを指定する事ができます。戻り値は *names* 中の名前指定されるテスト全てを含むテストスイートです。

**getTestCaseNames**( *testCaseClass* )

*testCaseClass* 中の全てのメソッド名を含むソート済みシーケンスを返します。

以下の属性は、サブクラス化またはインスタンスの属性値を変更して TestLoader をカスタマイズする場合に使用します。

**testMethodPrefix**

テストメソッドの名前と判断されるメソッド名の接頭語を示す文字列。デフォルト値は 'test' です。

**sortTestMethodsUsing**

getTestCaseNames() でメソッド名をソートする際に使用する比較関数。デフォルト値は組み込

み関数 `cmp()` です。None を指定するとソートを行いません。

#### **suiteClass**

テストのリストからテストスイートを構築する呼び出し可能オブジェクト。メソッドを持つ必要はありません。デフォルト値は `TestSuite` です。

## 5.4 test — Python 用回帰テストパッケージ

`test` パッケージには、Python 用の全ての回帰テストと、`test.test_support` および `test.regrtest` モジュールが入っています。`test.test_support` はテストを充実させるために使い、`test.regrtest` はテストスイートを駆動するのに使います。

`test` パッケージ内の各モジュールのうち、名前が `'test_'` で始まるものは、特定のモジュールや機能に対するテストスイートです。新しいテストはすべて `unittest` モジュールを使って書くようにしてください; 必ずしも `unittest` を使う必要はないのですが、`unittest` はテストをより柔軟にし、メンテナンスをより簡単にします。古いテストのいくつかは `doctest` を利用しており、“伝統的な” テスト形式になっています。これらのテスト形式をカバーする予定はありません。

参考資料:

`unittest` モジュール (5.3 節):

PyUnit 回帰テストを書く。

`doctest` モジュール (5.2 節):

ドキュメンテーション文字列に埋め込まれたテスト。

### 5.4.1 test パッケージのためのユニットテストを書く

`test` パッケージ用のテストを書く場合、`unittest` モジュールを使い、以下のいくつかのガイドラインに従うよう推奨します。一つは、テストモジュールの名前と同様、すべてのテストメソッドの名前を `'test_'` で始めることです。これはテスト駆動プログラムにそのメソッドをテストメソッドとして認識させるためです。また、テストメソッドにはドキュメンテーション文字列を入れるべきではありません。テストメソッドのドキュメント記述には、(`#True` あるいは `False` だけを返すテスト関数) のようなコメントを使ってください。これは、ドキュメンテーション文字列が存在する場合にはその内容が出力されるため、どのテストを実行しているのかをいちいち表示しなくするためです。

以下のような基本的な決まり文句を使います:



```

import unittest
from test import test_support

class MyTestCase1(unittest.TestCase):

    # Only use setUp() and tearDown() if necessary

    def setUp(self):
        ... code to execute in preparation for tests ...

    def tearDown(self):
        ... code to execute to clean up after tests ...

    def test_feature_one(self):
        # Test feature one.
        ... testing code ...

    def test_feature_two(self):
        # Test feature two.
        ... testing code ...

    ... more test methods ...

class MyTestCase2(unittest.TestCase):
    ... same structure as MyTestCase1 ...

... more test classes ...

def test_main():
    test_support.run_unittest(MyTestCase1,
                              MyTestCase2,
                              ... list other tests ...
                              )

if __name__ == '__main__':
    test_main()

```

この定型的なコードによって、テストスイートを `regtest.py` から起動できると同時に、スクリプト自体からも実行できるようになります。

回帰テストの目的はコードの分解です。そのためには以下のいくつかのガイドラインに従ってください:

- テストスイートはすべてのクラス、関数および定数を用いるべきです。これは外部に公開される外部 API だけでなく"非公開"コードも含んでいます。
- ホワイトボックス・テスト (テストを書くときに対象のコードをすぐテストする) を推奨します。ブラックボックス・テスト (最終的に公開されたユーザーインターフェイスだけをテストする) は、すべての境界条件と極端条件を確実にテストするには完全ではありません。
- 無効な値を含み、すべての取りうる値を確実にテストするようにしてください。そうすることで、全ての有効な値を受理するだけでなく、不適切な値を正しく処理することも確認できます。
- できる限り多くのコード経路を網羅してください。分岐が生じるテストし、入力を調整して、コードの全体に渡って取りうる限りの個々の処理経路を確実にたどらせるようにしてください。
- テスト対象のコードにどんなバグが発見された場合でも、明示的なテスト追加するようにしてください。そうすることで、将来コードを変更した際にエラーが再発しないようにできます。
- (一時ファイルをすべて閉じたり削除したりするといった) テストの後始末を必ず行ってください。

- import するモジュールをできるかぎり少なくし、可能な限り早期に import を行ってください。そうすることで、テストの外部依存性を最小限にし、モジュールの import による副作用から生じる変則的な動作を最小限にできます。
- コードの再利用を最大限に行うようにしてください。時として、テストの多様性はどんな型の入力を受け取るかの違いまで小さくなります。例えば以下のように、入力が指定されたサブクラスで基底テストクラスをサブクラス化して、コードの複製を最小化します:

```
class TestFuncAcceptsSequences(unittest.TestCase):

    func = mySuperWhammyFunction

    def test_func(self):
        self.func(self.arg)

class AcceptLists(TestFuncAcceptsSequences):
    arg = [1,2,3]

class AcceptStrings(TestFuncAcceptsSequences):
    arg = 'abc'

class AcceptTuples(TestFuncAcceptsSequences):
    arg = (1,2,3)
```

#### 参考資料:

##### *Test Driven Development*

コードより前にテストを書く方法論に関する Kent Beck の著書

### 5.4.2 test.regrtest を使ってテストを実行する

test.regrtest を使うと Python の回帰テストスイートを駆動できます。スクリプトを単独で実行すると、自動的に test パッケージ内のすべての回帰テストを実行し始めます。パッケージ内の名前が 'test\_' で始まる全モジュールを見つけ、それをインポートし、もしあるなら関数 test\_main を実行してテストを行います。実行するテストの名前もスクリプトに渡される可能性もあります。単一の回帰テストを指定 (python regrtest.py test\_spam.py) すると、出力を最小限にします。テストが成功したかあるいは失敗したかだけを出力するので、出力は最小限になります。

直接 test.regrtest を実行すると、テストに利用するリソースを設定できます。これを行うには、-u コマンドラインオプションを使います。すべてのリソースを使うには、python regrtest.py -uall を実行します。-u のオプションに all を指定すると、すべてのリソースを有効にします。(よくある場合ですが) 何か一つを除く全てが必要な場合、カンマで区切った不要なリソースのリストを all の後に並べます。コマンド python regrtest.py -uall,-audio,-largefile とすると、audio と largefile リソースを除く全てのリソースを使って test.regrtest を実行します。すべてのリソースのリストと追加のコマンドラインオプションを出力するには、python regrtest.py -h を実行してください。

テストを実行しようとするプラットフォームによっては、回帰テストを実行する別の方法があります。UNIX では、Python をビルドしたトップレベルディレクトリで make test を実行できます。Windows 上では、'PCBuild' ディレクトリから rt.bat を実行すると、すべての回帰テストを実行します。

### 5.4.3 test.test\_support — テストのためのユーティリティ関数

test.test\_support モジュールでは、Python の回帰テストに対するサポートを提供しています。

このモジュールは次の例外を定義しています:

**exception TestFailed**

テストが失敗したとき送出される例外です。

**exception TestSkipped**

TestFailed のサブクラスです。テストがスキップされたとき送出されます。テスト時に (ネットワーク接続のような) 必要なリソースが利用できないときに送出されます。

**exception ResourceDenied**

TestSkipped のサブクラスです。(ネットワーク接続のような) リソースが利用できないとき送出されます。requires 関数によって送出されます。

test.test\_support モジュールでは、以下の定数を定義しています:

**verbose**

冗長な出力が有効な場合は True です。実行中のテストについてのより詳細な情報が欲しいときにチェックします。verbose は test.regrtest によって設定されます。

**have\_unicode**

ユニコードサポートが利用可能ならば True になります。

**is\_jython**

実行中のインタプリタが Jython ならば True になります。

**TESTFN**

一時ファイルを作成するパスに設定されます。作成した一時ファイルは全て閉じ、unlink (削除) せねばなりません。

test.test\_support モジュールでは、以下の関数を定義しています:

**forget(*module\_name*)**

モジュール名 *module\_name* を sys.modules から取り除き、モジュールのバイトコンパイル済みファイル全てを削除します。

**is\_resource\_enabled(*resource*)**

*resource* が有効で利用可能ならば True を返します。利用可能なリソースのリストは、test.regrtest がテストを実行している間のみ設定されます。

**requires(*resource*[, *msg*])**

*resource* が利用できなければ、ResourceDenied を送出します。その場合、*msg* は ResourceDenied の引数になります。\_\_name\_\_ が "\_\_main\_\_" である関数にから呼び出された場合には常に真を返します。テストを test.regrtest から実行するときに使われます。

**findfile(*filename*)**

*filename* という名前のファイルへのパスを返します。一致するものが見つからなければ、*filename* 自体を返します。*filename* 自体もファイルへのパスでありえるので、*filename* が返っても失敗ではありません。

**run\_unittest(\**classes*)**

渡された unittest.TestCase サブクラスを実行します。この関数は名前が 'test\_' で始まるメソッドを探して、テストを個別に実行します。この方法をテストの実行方法として推奨しています。

**run\_suite(*suite*[, *testclass*=None])**

unittest.TestSuite のインスタンス *suite* を実行します。オプション引数 *testclass* はテストスイート内のテストクラスの一つを受け取り、指定するとテストスイートが存在する場所についてさらに詳細な情報を出力します。

## 5.5 math — 数学関数

このモジュールはいつでも利用できます。標準 C で定義されている数学関数にアクセスすることができます。

これらの関数で複素数を使うことはできません。複素数に対応する必要があるならば、`cmath` モジュールにある同じ名前の関数を使ってください。ほとんどのユーザーは複素数を理解するのに必要なだけの数学を勉強したくないので、複素数に対応した関数と対応していない関数の区別がされています。これらの関数では複素数が利用できないため、引数に複素数を渡されると、複素数の結果が返えるのではなく例外が発生します。その結果、プログラマは、そもそもどういった理由で例外がスローされたのかに早い段階で気づく事ができます。<sup>1</sup>

このモジュールでは次の関数が用意されています。特に明示的に指定されていなければ、戻り値は全て浮動小数点数となります:

**acos**(*x*)

*x* の逆余弦を返します。

**asin**(*x*)

*x* の逆余弦を返します。

**atan**(*x*)

*x* 逆余弦を返します。

**atan2**(*y, x*)

$\text{atan}(y / x)$  の逆正接を返します。

**ceil**(*x*)

*x* 以上の最も小さい整数を float 型で返します。

**cos**(*x*)

*x* の余弦を返します。

**cosh**(*x*)

*x* の双曲線余弦を返します。

**degrees**(*x*)

角 *x* をラジアンから度数に変換します。

**exp**(*x*)

$e^{**x}$  を返します。

**fabs**(*x*)

*x* の絶対値を返します。

**floor**(*x*)

*x* 以下の最も大きい整数を float 型で返します。

**fmod**(*x, y*)

プラットフォームの C ライブラリで定義されている `fmod(x, y)` を返します。Python の `x % y` という式が同じ結果を返さないかもしれないことに注意してください。

**frexp**(*x*)

*x* の仮数と指数を (*m*, *e*) のペアとして返します。 `x == m * 2**e` というように、*m* は float 型で *e* は int 型です。*x* がゼロの場合は、(0.0, 0) を返し、それ以外の場合は、`0.5 <= abs(m) < 1` です。

**hypot**(*x, y*)

---

<sup>1</sup> 訳注: 例外が発生しないで、計算結果が返ってしまうと、計算結果がおかしいことから、原因が複素数を渡したせいである事にプログラマが気づくのがおくれる可能性があります。

ユークリッド距離 ( $\text{sqrt}(x*x + y*y)$ ) を返します。

`ldexp(x, i)`

$x * (2^{**i})$  を返します。

`log(x[, base])`

$x$  の自然対数を返します。 *base* を底とした  $x$  の対数を返します。 *base* を省略した場合  $x$  の自然対数を返します。 2.3 で変更された仕様: *base* argument added

`log10(x)`

$x$  の 10 を底とした対数を返します。

`modf(x)`

$x$  の小数部分と整数部分を返します。両方の結果は  $x$  の符号を受け継ぎます。整数部は float 型で返されます。

`pow(x, y)`

$x^{**y}$  を返します。

`radians(x)`

角  $x$  を度数からラジアンに変換します。

`sin(x)`

$x$  の正弦を返します。

`sinh(x)`

$x$  の双曲線正弦を返します。

`sqrt(x)`

$x$  の平方根を返します。

`tan(x)`

$x$  の正接を返します。

`tanh(x)`

$x$  の双曲線正接を返します。

`frexp()` と `modf()` は C のものとは異なった呼び出し/返しパターンを持っていることに注意してください。引数を 1 つだけ受け取り、1 組のペアになった値を返すので、2 つ目の戻り値を ‘出力引数’(Python にそのようなものはありません) 経由で返したりはしません。

このモジュールでは 2 つの数学定数も定義されています。

`pi`

数学定数  $\pi$ 。

`e`

数学定数  $e$ 。

注意: `math` モジュールは、ほとんどが実行プラットフォームにおける C 言語の数学ライブラリ関数に対する薄いラップでできています。例外的な場合での挙動は、C 言語標準ではおおさっぱにしか定義されておらず、さらに Python は数学関数におけるエラー報告機能の挙動をプラットフォームの C 実装から受け継いでいます。その結果として、エラーの際(およびなんらかの引数がとにかく例外的であると考えられる場合)に送出される特定の例外については、プラットフォーム間やリリースバージョン間を通じて有意なものになっていません。例えば、`math.log(0)` が `-Inf` を返すか `ValueError` または `OverflowError` を送出するかは不定であり、`math.log(0)` が `OverflowError` を送出する場合において `math.log(0L)` が `ValueError` を送出するときもあります。

参考資料:

`cmath` モジュール (5.6 節):

これらの多くの関数の複素数版。

## 5.6 cmath — 複素数のための数学関数

このモジュールは常に利用することができます。このモジュールでは、複素数のための数学関数群へのアクセス手段を提供します。以下が提供されている関数群です：

**acos**( $x$ )

$x$  の逆余弦 (arc cosine) を返します。この関数には二つの branch cut があります：一つは 1 から右側に実数軸に沿って  $\infty$  へと延びていて、下から連続しています。もう一つは -1 から左側に実数軸に沿って  $-\infty$  へと延びていて、上から連続しています。

**acosh**( $x$ )

$x$  の逆双極線余弦を返します。branch cut が一つあり、1 の左側に実数軸に沿って  $-\infty$  へと延びていて、上から連続しています。

**asin**( $x$ )

$x$  の逆正弦を返します。acos() と同じ branch cut を持ちます。

**asinh**( $x$ )

$x$  の双極正弦を返します。2 つの branch cut があり、 $\pm 1j$  の左から  $\pm \infty j$  に延びており、両方とも上で連続しています。これらの branch cut は将来のリリースで修正されるべきバグとみなされています。正しい branch cut は虚数軸に沿って延びており、一つは  $1j$  から  $\infty j$  までで右から連続、もう一方は  $-1j$  から下って  $-\infty j$  までで、左から連続です。

**atan**( $x$ )

$x$  の逆正接を返します。2 つの branch cut があります：一つは  $1j$  から虚数軸に沿って  $\infty j$  へと延びており、左で連続です。もう一方は  $-1j$  から虚数軸に沿って  $-\infty j$  までで、左で連続です。(この仕様は上の branch cut が反対側から連続になるように変更されるかもしれません)。

**atanh**( $x$ )

$x$  の双極逆正接を返します。2 つの branch cut があります：一つは 1 から実数軸に沿って  $\infty$  までで、上で連続です。もう一方は -1 から実数軸に沿って  $-\infty$  までで、上で連続です。(この仕様は左側の branch cut が反対側から連続になるように変更されるかもしれません)。

**cos**( $x$ )

$x$  の余弦を返します。

**cosh**( $x$ )

$x$  の双極線余弦を返します。

**exp**( $x$ )

指数値  $e^{**x}$  を返します。

**log**( $x$ )

$x$  の自然対数を返します。branch cut を一つもち、0 から負の実数軸に沿って  $-\infty$  に延びており、上で連続しています。

**log10**( $x$ )

$x$  の底 10 対数を返します。log() と同じ branch cut を持ちます。

**sin**( $x$ )

$x$  の正弦を返します。

**sinh**( $x$ )

$x$  の双曲線正弦を返します。

**sqrt**( $x$ )



$x$  の平方根を返します。 `log()` と同じ branch cut を持ちます。

`tan(x)`

$x$  の正接を返します。

`tanh(x)`

$x$  の双曲線正接を返します。

このモジュールではまた、以下の数学上の定数も定義しています:

`pi`

数学上の定数  $\pi$  で、実数です。

`e`

数学上の定数  $e$  で、実数です。

提供される関数の選択は `math` と同様ですが、全く同じではないので注意してください。二つのモジュールにしているのは、ユーザによっては複素数には興味がなく、おそらく複素数とは何かすら知らないからです。そういった人たちはむしろ、`math.sqrt(-1)` が複素数を返すよりも例外を送出するほうを好みます。また、`cmath` で定義されている関数は常に、たとえ答えを実数として表現することができる (虚数部分がゼロの複素数) の場合でも、複素数を返すので注意してください。

branch cut に関する注釈: branch cut をもつ曲線上では、与えられた関数は連続でありえなくなります。これらは多くの複素関数における必然的な特性です。複素関数を計算する必要がある場合、これらの branch cut に関して理解しているものと仮定しています。悟りに至るために何らかの (到底基礎的とはいえない) 複素数に関する書をひもといってください。数値計算を目的とした branch cut の正しい選択方法についての情報としては、以下がよい参考文献となります:

参考資料:

Kahan, W: Branch cuts for complex elementary functions; or, Much ado about nothings's sign bit. In Iserles, A., and Powell, M. (eds.), *The state of the art in numerical analysis*. Clarendon Press (1987) pp165-211.

## 5.7 random — 擬似乱数を生成する

このモジュールでは様々な分布をもつ擬似乱数生成器を実装しています。整数用では、ある値域内の数の選択を一様にします。配列用には、配列からのランダムな要素の一様な選択、リストの要素の順列をランダムに置き換える関数、順列を入れ替えずにランダムに取り出す関数があります。

実数用としては、一様分布、正規分布 (ガウス分布)、対数正規分布、負の指数分布、ガンマおよびベータ分布を計算する関数があります。角度分布の生成用には、von Mises 分布が利用可能です。

ほとんど全てのモジュール関数は基礎となる関数 `random()` に依存します。この関数は半开区間  $[0.0, 1.0)$  の値域を持つ一様な浮動小数点数を生成します。Python は中心となる乱数生成器として Mersenne Twister を使います。これは 53 ビットの浮動小数点を生成し、周期が  $2^{19937}-1$ 、本体は C で実装されていて、高速でスレッドセーフです。Mersenne Twister は、現存する中で、最も大規模にテストされた乱数生成器のひとつです。しかし、完全に決定論的であるため、この乱数生成器は全ての目的に合致しているわけではなく、暗号化の目的には全く向いていません。

このモジュールで提供されている関数は、実際には `random.Random` クラスの隠蔽されたインスタンスのメソッドにバインドされています。内部状態を共有しない生成器を取得するため、自分で `Random` のインスタンスを生成することができます。異なる `Random` のインスタンスを各スレッド毎に生成し、`jumpahead()` メソッドを使うことで各々のスレッドにおいて生成された乱数配列が重複しないようにすれば、マルチスレッドプログラムを作成する上で特に便利になります。

自分で考案した基本乱数生成器を使いたいなら、クラス `Random` をサブクラス化することもできます: この場合、メソッド `random()`、`send()`、`getstate()`、`setstate()`、および `jumpahead()` をオー

パライドしてください。

サブクラス化の例として、random モジュールは WichmannHill クラスを提供します。このクラスは Python だけで書かれた代替生成器を実装しています。このクラスは、乱数生成器に Wichmann-Hill 法を使っていた古いバージョンの Python から得られた結果を再現するための、後方互換の手段になります。2.3 で変更された仕様: MersenneTwister を Wichmann-Hill の代わりに使う

保守関数:

**seed**(*x*)

基本乱数生成器を初期化します。オプション引数 *x* はハッシュ可能な任意のオブジェクトをとり得ます。*x* が省略されるか None の場合、現在のシステム時間が使われます; 現在のシステム時間はモジュールが最初にインポートされた時に乱数生成器を初期化するためにも使われます。*x* が None でも、整数でも長整数でもない場合、`hash(x)` が代わりに使われます。*x* が整数または長整数の場合、*x* が直接使われます。

**getstate**()

乱数生成器の現在の内部状態を記憶したオブジェクトを返します。このオブジェクトを `setstate()` に渡して内部状態を復帰することができます。2.1 で追加された仕様です。

**setstate**(*state*)

*state* は予め `getstate()` を呼び出して得ておかななくてはなりません。`setstate()` は `setstate()` が呼び出された時の乱数生成器の内部状態を復帰します。2.1 で追加された仕様です。

**jumpahead**(*n*)

内部状態を、現在の状態から、非常に離れているであろう状態に変更します。*n* は非負の整数です。これはマルチスレッドのプログラムが複数の Random クラスのインスタンスと結合されている場合に非常に便利です: `setstate()` や `seed()` は全てのインスタンスを同じ内部状態にするのに使うことができ、その後 `jumpahead()` を使って各インスタンスの内部状態を引き離すことができます。2.1 で追加された仕様です。2.3 で変更された仕様: *n* ステップ先の特定の状態になるのではなく、`jumpahead(n)` は何ステップも離れているであろう別の状態にする

整数用の関数:

**randrange**(*start*, *stop*, *step*)

`range(start, stop, step)` の要素からランダムに選ばれた要素を返します。この関数は `choice(range(start, stop, step))` と等価ですが、実際には `range` オブジェクトを生成しません。1.5.2 で追加された仕様です。

**randint**(*a*, *b*)

$a \leq N \leq b$  であるようなランダムな整数 *N* を返します。

配列用の関数:

**choice**(*seq*)

空でない配列 *seq* からランダムに要素を返します。

**shuffle**(*x*, *random*)

配列 *x* を直接変更によって混ぜます。オプションの引数 *random* は、値域が [0.0, 1.0) のランダムな浮動小数点数を返すような引数を持たない関数です; 標準では、この関数は `random()` です。

かなり小さい `len(x)` であっても、*x* の順列はほとんどの乱数生成器の周期よりも大きくなるので注意してください; このことは長い配列に対してはほとんどの順列は生成されないことを意味します。

**sample**(*population*, *k*)

母集団の配列から選ばれた長さ *k* の一意な要素からなるリストを返します。値の置換を行わないランダムサンプリングに用いられます。2.3 で追加された仕様です。

母集団自体を変更せずに、母集団内の要素を含む新たなリストを返します。返されたリストは選択

された順に並んでいるので、このリストの部分スライスもランダムなサンプルになります。これにより、くじの当選者を 1 等賞と 2 等賞 (の部分スライス) に分けるといったことも可能です。母集団の要素はハッシュ可能でなくても、ユニークでなくても、かまいません。母集団が繰り返しを含む場合、返されたリストの各要素はサンプルから選択可能な要素になります。整数の並びからサンプルを選ぶには、引数に `xrange` を使いましょう。特に、巨大な母集団からサンプルを取るとき、速度と空間効率が上がります。 `sample(xrange(10000000), 60)`

以下の関数は特殊な実数値分布を生成します。関数パラメタは対応する分布の公式において、数学的な慣行に従って使われている変数から取られた名前がつけられています; これらの公式のほとんどは多くの統計学のテキストに載っています。

**random()**

値域  $[0.0, 1.0)$  の次のランダムな浮動小数点数を返します。

**uniform(*a*, *b*)**

$a \leq N \leq b$  であるようなランダムな実数  $N$  を返します。

**betavariate(*alpha*, *beta*)**

ベータ分布です。引数の満たすべき条件は Beta distribution. Conditions on the parameters are  $\alpha > -1$  および  $\beta > -1$  です。0 から 1 の値を返します。

**cunifvariate(*mean*, *arc*)**

円形一様分布です。 *mean* は平均角度で、 *arc* は平均角を中心とした分布の範囲です。値は両方ともラジアンで表され、0 から  $\pi$  の値をとることができます。返される値の範囲は  $mean - arc/2$  から  $mean + arc/2$  になりますが、0 から  $\pi$  の間に正規化されます。

リリース 2.3 以降で撤廃された仕様です。 ( $mean + arc * (random.random() - 0.5) * \pi$  を代わりに使ってください。

**expovariate(*lambda*)**

指数分布です。 *lambda* は平均にしたい値で 1.0 を割ったものです。(このパラメタは “lambda” と呼ぶべきなのですが、Python の予約語なので使えません。) 返される値の範囲は 0 から正の無限大です。

**gammavariate(*alpha*, *beta*)**

ガンマ分布です。(ガンマ関数 ではありません ! ) 引数の満たすべき条件は  $\alpha > 0$  および  $\beta > 0$  です。

**gauss(*mu*, *sigma*)**

ガウス分布です。 *mu* は平均であり、 *sigma* は標準偏差です。この関数は後で定義する関数 `normalvariate()` より少しだけ高速です。

**lognormvariate(*mu*, *sigma*)**

対数正規分布です。この分布を自然対数を用いた分布にした場合、平均 *mu* で標準偏差 *sigma* の正規分布になるでしょう。 *mu* は任意の値を取ることができ、 *sigma* はゼロより大きくなければなりません。

**normalvariate(*mu*, *sigma*)**

正規分布です、 *mu* は平均で、 *sigma* は標準偏差です。

**vonmisesvariate(*mu*, *kappa*)**

*mu* は平均の角度で、0 から  $2\pi$  までのラジアンで表されます。 *kappa* は濃度パラメタで、ゼロまたはそれ以上でなければなりません。 *kappa* がゼロに等しい場合、この分布は範囲 0 から  $2\pi$  の一様でランダムな角度の分布に退化します。

**paretovariate(*alpha*)**

パレート分布です。 *alpha* は形状パラメタです。

**weibullvariate(*alpha*, *beta*)**

ワイブル分布です。 *alpha* はスケールパラメタで、 *beta* は形状パラメタです。

代替の乱数生成器

`class WichmannHill([seed])`

乱数生成器として Wichmann-Hill アルゴリズムを実装するクラスです。Random クラスと同じメソッド全てと、下で説明する `whseed` メソッドを持ちます。このクラスは、Python だけで実装されているので、スレッドセーフではなく、呼び出しと呼び出しの間にロックが必要です。また、周期が 6,953,607,871,644 と短く、独立した 2 つの乱数列が重複しないように注意が必要です。

`whseed([x])`

これは `obsolete` で、バージョン 2.1 以前の Python と、ビット・レベルの互換性のために提供されます。詳細は `seed` を参照してください。 `whseed` は、引数に与えた整数が異なっても、内部状態が異なることを保障しません。取り得る内部状態の個数が  $2^{24}$  以下になる場合もあります。

参考資料:

M. Matsumoto and T. Nishimura, “Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator”, *ACM Transactions on Modeling and Computer Simulation* Vol. 8, No. 1, January pp.3-30 1998.

Wichmann, B. A. & Hill, I. D., “Algorithm AS 183: An efficient and portable pseudo-random number generator”, *Applied Statistics* 31 (1982) 188-190.

## 5.8 whrandom — 擬似乱数生成器

リリース 2.1 以降で撤廃された仕様です。 `random` を代わりに使ってください。

注意: Python 2.1 以前のリリースでは、このモジュールは `random` モジュールの実装における一部でした。現在はもう使われていません。このモジュールを直接使わないでください; 代わりに `random` を使ってください。 This module was an implementation detail of the

このモジュールは Wichmann-Hill による擬似乱数生成器クラスを実装します。このクラスはまた、 `whrandom` と名づけられています。 `whrandom` クラスのインスタンスは?? 節で記述されている乱数生成器インタフェース (Random Number Generator インタフェース) に準拠しています。このモジュールではまた、Wichmann-Hill アルゴリズムに特有の以下のメソッドを提供しています:

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

整数 *x*、*y*、*z* から乱数生成器を初期化します。このモジュールが最初に取り込まれたときに、乱数は現在の時刻から取り出された値で初期化されます。*x*、*y*、および *z* が省略されるか 0 の場合、乱数のシードは現在のシステム時刻から計算されます。引数のうち 3 つ全てではなく、1 または 2 個だけが 0 の場合、ゼロに鳴っている値は 1 に置き換えられます。このことにより、一見して異なるシード値が同じ値になってしまい、乱数生成器から生成される擬似乱数列もこれに対応します。

`choice(seq)`

空でない配列 *seq* からランダムに要素を選んで返します。

`randint(a, b)`

$a \leq N \leq b$  であるような整数の乱数 *N* を返します。

`random()`

範囲 [0.0 ... 1.0) から次の浮動小数点数の乱数 *N* を返します。

`seed(x, y, z)`

整数 *x*、*y*、*z* から乱数生成器を初期化します。このモジュールが最初にインポートされた際、乱数は現在の時刻から取り出された値で初期化されます。

`uniform(a, b)`

$a \leq N < b$  であるようなランダムな実数  $N$  を返します。

`whrandom` モジュールがインポートされた時、`whrandom` クラスのインスタンスも生成され、このインスタンスのメソッドをモジュールレベルで利用できるようにします。従って、`N = whrandom.random()` を以下のコード:

```
generator = whrandom.whrandom()
N = generator.random()
```

のように書くこともできます。

乱数生成器の別々のインスタンスを使った場合、擬似乱数の配列は独立になるので注意してください。

参考資料:

`random` モジュール (5.7 節):

Generators for various random distributions and documentation for the Random Number Generator interface.

Wichmann, B. A. & Hill, I. D., "Algorithm AS 183: An efficient and portable pseudo-random number generator", *Applied Statistics* 31 (1982) 188-190.

## 5.9 bisect — 配列二分法アルゴリズム

このモジュールは、挿入の度にリストをソートすることなく、リストをソートされた順序に保つことをサポートします。大量の比較操作を伴うような、アイテムがたくさんあるリストでは、より一般的なアプローチに比べて、パフォーマンスが向上します。

動作に基本的な二分法アルゴリズムを使っているので、`bisect` と呼ばれています。ソースコードはこのアルゴリズムの実例として一番役に立つかもしれません (境界条件はすでに正しいです!)

次の関数が用意されています。

**bisect\_left**(*list*, *item*[, *lo*[, *hi*]])

ソートされた順序を保ったまま *item* を *list* に挿入するのに適した挿入点を探し当てます。リストの中から検索する部分集合を指定するには、パラメーターの *lo* と *hi* を使います。デフォルトでは、リスト全体が使われます。*item* がすでに *list* に含まれている場合、挿入点はどのエントリーよりも前(左)になります。戻り値は、`list.insert()` の第一引数として使うのに適しています。*list* はすでにソートされているものとします。2.1 で追加された仕様です。

**bisect\_right**(*list*, *item*[, *lo*[, *hi*]])

`bisect_left()` と似ていますが、*list* に含まれる *item* のうち、どのエントリーよりも後ろ (右) にくるような挿入点を返します。2.1 で追加された仕様です。。

**bisect**(...)

`bisect_right()` のエイリアス。

**insort\_left**(*list*, *item*[, *lo*[, *hi*]])

*item* を *list* にソートされた順序で (ソートされたまま) 挿入します。これは、`list.insert(bisect.bisect_left(list, item, lo, hi), item)` と同等です。*list* はすでにソートされているものとします。2.1 で追加された仕様です。

**insort\_right**(*list*, *item*[, *lo*[, *hi*]])

`insort_left()` と似ていますが、*list* に含まれる *item* のうち、どのエントリーよりも後ろに *item* を挿入します。2.1 で追加された仕様です。

**insort**(...)



`insort_right()` のエイリアス。

### 5.9.1 使用例

一般には、`bisect()` 関数は数値データを分類するのに役に立ちます。この例では、`bisect()` を使って、(たとえば) 順序のついた数値の区切り点の集合に基づいて、試験全体の成績の文字を調べます。区切り点は 85 以上は 'A'、75..84 は 'B'、などです。

```
>>> grades = "FEDCBA"
>>> breakpoints = [30, 44, 66, 75, 85]
>>> from bisect import bisect
>>> def grade(total):
...     return grades[bisect(breakpoints, total)]
...
>>> grade(66)
'C'
>>> map(grade, [33, 99, 77, 44, 12, 88])
['E', 'A', 'B', 'D', 'F', 'A']
```

`bisect` モジュールは `Queue` モジュールと一緒に使って、優先順序つき待ち行列を実装することができます。(Fredrik Lundh の好意による例です):

```
import Queue, bisect

class PriorityQueue(Queue.Queue):
    def _put(self, item):
        bisect.insort(self.queue, item)

# 使い方
queue = PriorityQueue(0)
queue.put((2, "second"))
queue.put((1, "first"))
queue.put((3, "third"))
priority, value = queue.get()
```

## 5.10 heapq — ヒープキューアルゴリズム

2.3 で追加された仕様です。

このモジュールではヒープキューアルゴリズムの一実装を提供しています。優先度キューアルゴリズムとしても知られています。

ヒープとは、全ての  $k$  に対して、ゼロから要素を数えていった際に、 $heap[k] \leq heap[2*k+1]$  かつ  $heap[k] \leq heap[2*k+2]$  となる配列です。比較のために、存在しない要素は無限大として扱われます。ヒープの興味深い属性は  $heap[0]$  が常に最小の要素になることです。

以下の API は教科書におけるヒープアルゴリズムとは 2 つの側面で異なります: (a) ゼロベースのインデクス化を行っています。これにより、ノードに対するインデクスとその子ノードのインデクスの関係がやや明瞭でなくなりますが、Python はゼロベースのインデクス化を使っているのでよりしっくりきます。(b) われわれの `pop` メソッドは最大の要素ではなく最小の要素 (教科書では "min heap: 最小ヒープ" と呼ばれています; 教科書では並べ替えをインプレースで行うのに適した "max heap: 最大ヒープ" が一般的です)。

これらの 2 点によって、ユーザに戸惑いを与えることなく、ヒープを通常の Python リストとして見るこ



とができます: `heap[0]` が最小の要素となり、`heap.sort()` はヒープを不変なままに保ちます!

ヒープを作成するには、`[]` に初期化されたリストを使うか、`heapify()` を用いて要素の入ったリストを変換します。

以下の関数が提供されています:

**heappush**(*heap*, *item*)

*item* を *heap* に push します。ヒープを不変に保ちます。

**heappop**(*heap*)

pop を行い、*heap* から最初の要素を返します。ヒープは不変に保たれます。ヒープが空の場合、`IndexError` が送出されます。

**heapify**(*x*)

リスト *x* をインブレース処理し、線形時間でヒープに変換します。

**heapreplace**(*heap*, *item*)

*heap* から最小の要素を pop して返し、新たに *item* を push します。ヒープのサイズは変更されません。ヒープが空の場合、`IndexError` が送出されます。この関数は `heappop()` に次いで `heappush()` を送出するよりも効率的で、固定サイズのヒープを用いている場合にはより適しています。返される値は *item* よりも大きくなるかもしれないので気をつけてください! これにより、このルーチンの合理的な利用法が制限されています。

使用例を以下に示します:

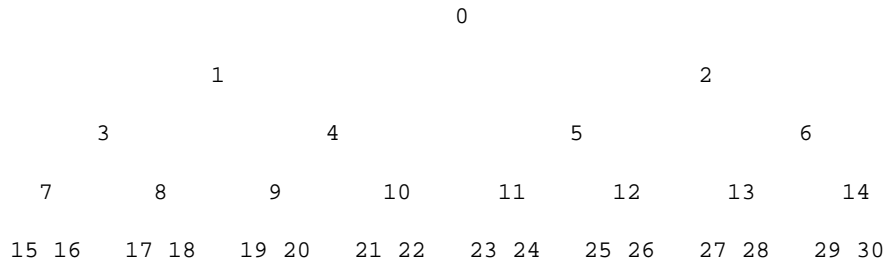
```
>>> from heapq import heappush, heappop
>>> heap = []
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> for item in data:
...     heappush(heap, item)
...
>>> sorted = []
>>> while heap:
...     sorted.append(heappop(heap))
...
>>> print sorted
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> data.sort()
>>> print data == sorted
True
>>>
```

### 5.10.1 理論

(説明は Fran 轍 is Pinard によるものです。このモジュールの Python コードは Kevin O'Connor の貢献によるものです。)

ヒープとは、全ての  $k$  について、要素を 0 から数えたときに、 $a[k] \leq a[2k+1]$  かつ  $a[k] \leq a[2k+2]$  となる配列です。比較のために、存在しない要素を無限大と考えます。ヒープの興味深い属性は `heap[0]` が常に最小の要素になることです。

上記の奇妙な不変式は、勝ち抜き戦判定の際に効率的なメモリ表現を行うためのものです。以下の番号は  $a[k]$  ではなく  $k$  とします:



上の木構造では、各セル  $k$  は  $2*k+1$  および  $2*k+2$  を最大値としています。スポーツに見られるような通常の 2 つ組勝ち抜き戦では、各セルはその下にある二つのセルに対する勝者となっていて、個々のセルの勝者を追跡していくことにより、そのセルに対する全ての相手を見ることができます。しかしながら、このような勝ち抜き戦を使う計算機アプリケーションの多くでは、勝歴を追跡する必要はありません。メモリ効率をより高めるために、勝者が上位に進級した際、下のレベルから持ってきて置き換えることにすると、あるセルとその下位にある二つのセルは異なる三つの要素を含み、かつ上位のセルは二つの下位のセルに対して "勝者" となります。

このヒープ不変式が常に守られれば、インデックス 0 は明らかに最勝者となります。最勝者の要素を除去し、"次の" 勝者を見つけるための最も単純なアルゴリズム的手法は、ある敗者要素 (ここでは上図のセル 30 とします) を 0 の場所に持っていく、この新しい 0 を濾過するようにしてツリーを下らせて値を交換してゆきます。不変関係が再構築されるまでこれを続けます。この操作は明らかに、ツリー内の全ての要素数に対して対数的な計算量となります。全ての要素について繰り返すと、 $O(n \log n)$  のソート (並べ替え) になります。

このソートの良い点は、新たに挿入する要素が、その最に取り出す 0 番目の要素よりも "良い値" でない限り、ソートを行っている最中に新たな要素を効率的に追加できるということです。

この性質は、シミュレーション的な状況で、ツリーで全ての入力イベントを保持し、"勝者となる状況" を最小のスケジュール時刻にするような場合に特に便利です。あるイベントが他のイベント群の実行をスケジュールする際、それらは未来にスケジュールされることになるので、それらのイベント群を容易にヒープに積むことができます。すなわち、ヒープはスケジューラを実装する上で良いデータ構造であるといえます (私は MIDI シーケンサで使っているものです :-) )。

これまでスケジューラを実装するための様々なデータ構造が広範に研究されています。ヒープは十分高速で、速度もおおむね一定であり、最悪の場合でも平均的な速度とさほど変わらないため良いデータ構造といえます。しかし、最悪の場合がひどい速度になることを除き、たいいていより効率の高い他のデータ構造表現も存在します。

ヒープはまた、巨大なディスクのソートでも非常に有用です。おそらくご存知のように、巨大なソートを行うと、複数の "ラン (run)" (予めソートされた配列で、そのサイズは通常 CPU メモリの量に関係しています) が生成され、続いて統合処理 (merging) がこれらのランを判定します。この統合処理はしばしば非常に巧妙に組織されています<sup>2</sup>。重要なのは、最初のソートが可能な限り長いランを生成することです。勝ち抜き戦はこれを行うための良い方法です。もし利用可能な全てのメモリを使って勝ち抜き戦を行い、要素を置換および濾過処理して現在のランに収めれば、ランダムな入力に対してメモリの二倍のサイズのランを生成することになり、大体順序づけがなされている入力に対してはもっと高い効率になります。

さらに、ディスク上の 0 番目の要素を出力して、現在の勝ち抜き戦に (最後に出力した値に "勝って" しまつために) 収められない入力を得たなら、ヒープには収まらないため、ヒープのサイズは減少します。解

<sup>2</sup>現在使われているディスクバランサ化アルゴリズムは、最近ではもはや巧妙というよりも目障りであり、このためにディスクに対するシーク機能が重要になっています。巨大な容量を持つテープのようにシーク不能なデバイスでは、事情は全く異なり、個々のテープ上の移動が可能な限り効率的に行われるように非常に巧妙な処理を (相当前もって) 行わねばなりません (すなわち、もっとも統合処理の "進行" に関係があります)。テープによっては逆方向に読むことさえでき、巻き戻しに時間を取られるのを避けるために使うこともできます。正直、本当に良いテープソートは見えていて素晴らしく驚異的なものです！ソートというのは常に偉大な芸術なのです！ :-)

放されたメモリは二つ目のヒープを段階的に構築するために巧妙に再利用することができ、この二つ目のヒープは最初のヒープが崩壊していくのと同じ速度で成長します。最初のヒープが完全に消滅したら、ヒープを切り替えて新たなランを開始します。なんと巧妙で効率的なのでしょう！

一言で言うと、ヒープは知って得するメモリ構造です。私はいくつかのアプリケーションでヒープを使っていて、‘ヒープ’モジュールを常備するのはいい事だと考えています。:-)

## 5.11 array — 効率のよい数値配列

このモジュールは基本的な値(文字、整数、浮動小数点数)の配列を効率よく表現できる新しいオブジェクト型を定義します。配列はシーケンス型であり、格納されるオブジェクトの型に制限があることを除けば、リストとまったく同じように振る舞います。オブジェクト生成時に一文字の型コードを用いて型を指定します。次の型コードが定義されています:

型コード	C の型	Python の型	最小サイズ (バイト単位)
'c'	char	文字 (str 型)	1
'b'	signed char	int 型	1
'B'	unsigned char	int 型	1
'u'	Py_UNICODE	Unicode 文字 (unicode 型)	2
'h'	signed short	int 型	2
'H'	unsigned short	int 型	2
'i'	signed int	int 型	2
'I'	unsigned int	long 型	2
'l'	signed long	int 型	4
'L'	unsigned long	long 型	4
'f'	float	float 型	4
'd'	double	float 型	8

値の実際の表現はマシンアーキテクチャ(厳密に言うと C の実装)によって決まります。itemsize 属性から実際のサイズが得られます。Python の通常の整数型では C の unsigned (long) 整数の最大範囲を表せないため、'L' と 'I' 要素として格納される値は Python の長整数として表されます。

モジュールは次の型を定義します:

**array**(*typecode*[, *initializer*])

要素のデータ型が *typecode* に限定される新しい配列を返します。オプションの値 *initializer* をわたすと初期値になりますが、リストまたは文字列でなければなりません。新しい配列に初期要素を追加するために、リストまたは文字列はその配列の `fromlist()`、`fromstring()` あるいは `fromunicode()` メソッド (以下を参照して下さい) へ渡されます。

**ArrayType**

もはや使われない array の別名。

配列オブジェクトは、添字付け、スライス、連結および反復といった通常のシーケンス演算をサポートします。スライス代入を使うとき、代入される値は同じ方コードの配列オブジェクトでなければなりません。他すべての場合には、`TypeError` が発生します。配列オブジェクトはバッファインターフェイスを実装しており、バッファオブジェクトがサポートされるのであればどこでも使われます。

次のデータアイテムとメソッドもサポートされます:

**typecode**

配列を作るときに使われる型コード文字です。

**itemsize**

内部表現における 1 つの配列要素のバイト単位の長さです。

**append(*x*)**

値 *x* をもつ新しい要素を配列の末尾に追加します。

**buffer\_info()**

配列の内容を保持するのに使われているバッファの現在のメモリアドレスと要素の長さを与えるタプル (*address*, *length*) を返します。`array.buffer_info()[1] * array.itemsize` で、バイト単位のメモリバッファの大きさを計算できます。メモリアドレスを必要とする、例えば `ioctl()` 操作のような低レベルな (そして、本質的に危険な) I/O インタフェースを使って作業する場合に、ときどき便利です。その配列が存在し、長さを変える演算が行われない限り、返される数は有効です。

注意: C あるいは C++ で書かれたコードから配列オブジェクトを使う (この情報を効率的に使うたった一つの方法) ときは、配列オブジェクトがサポートするバッファインターフェイスを使う方がより理にかなっています。このメソッドは後方互換性のために保守されており、新しいコードでの使用は避けるべきです。バッファインターフェイスは *Python/C API* リファレンスマニュアルで文書化されています。

**byteswap()**

配列のすべての要素に対して “バイトスワップ” (リトルエンディアンとビッグエンディアンの変換) を行います。これは大きさが 1、2、4 および 8 バイトである値に対してのみサポートされます。値の他の型に対しては、`RuntimeError` が発生します。異なるバイトオーダーをもつマシンで書かれたファイルからデータを読み込むときに役に立ちます。

**count(*x*)**

配列中の *x* の出現回数を返す。

**extend(*a*)**

*a* から配列の末尾へ配列要素を追加します。二つの配列は全く同じ型コードを持っていなければなりません。そうでなければ、`TypeError` が発生します。

**fromfile(*f*, *n*)**

ファイルオブジェクト *f* から (マシン依存のデータ形式そのまま) *n* 個の要素を読み込み、それらを配列の末尾に追加します。*n* 個の要素を読めなかったときは `EOFError` が発生しますが、そのときでも利用可能な要素は配列に追加されます。*f* は実際の組み込みファイルオブジェクトでなければなりません。`read()` メソッドをもつ他のものは動作しないでしょう。

**fromlist(*list*)**

リストから要素を追加します。型エラーが発生した場合に配列が変更されないことを除いて、`for x in list: a.append(x)` と同等です。

**fromstring(*s*)**

文字列から要素を追加します。(まるでファイルから `fromfile()` メソッドを使って読み込んだかのように) マシン依存のデータの配列として文字列を解釈しています。

**fromunicode(*s*)**

与えられたユニコード文字列からのデータで、この配列を拡張します。配列は型 `'u'` の配列でなければなりません。そうでなければ、`ValueError` が発生します。他の型の配列にユニコードデータを追加するには、`'array.fromstring(ustr.decode(enc))'` を使ってください。

**index(*x*)**

配列の中で *x* が初めて現れるインデックスである、もっとも小さい *i* を返します。

**insert(*i*, *x*)**

配列の位置 *i* の前に値 *x* をもつ新しい要素を挿入します。*i* が負数の場合、配列の末尾からの相対位

置として扱います。

`pop([i])`

配列からインデックス  $i$  の要素を取り除き、それを返します。デフォルトで最後の要素を取り除いて返すようすするために、オプションの引数の既定値は  $-1$  です。

`read(f, n)`

リリース 1.5.1 以降で撤廃された仕様です。 `fromfile()` メソッドを使ってください。

ファイルオブジェクト  $f$  から (マシン依存のデータ形式そのまま)  $n$  個の要素を読み込んで、それらを配列の末尾に追加します。  $n$  個より少ない要素しか読めなかった場合には `EOFError` が発生しますが、そのときでも利用可能な要素は配列に追加されます。  $f$  は実際の組み込みファイルオブジェクトでなければなりません。 `read()` をもつ他のものは動作しないでしょう。

`remove(x)`

配列から最初に現れる  $x$  を取り除きます。

`reverse()`

配列の要素の順番を逆にします。

`tofile(f)`

ファイルオブジェクト  $f$  にすべての要素を (マシン依存のデータ形式そのまま) 書き込みます。

`tolist()`

配列を同じ要素を持つ普通のリストに変換します。

`tostring()`

配列をマシン依存のデータの配列に変換し、文字列表現 (`tofile()` メソッドによってファイルに書き込まれるものと同じバイトシーケンス) を返します。

`tounicode()`

配列をユニコード文字列に変換します。配列は型 `'u'` の配列でなければなりません。そうでないならば、`ValueError` が発生します。他の型の配列からユニコード文字列を得るには、`array.tostring().decode(enc)` をを使ってください。

`write(f)`

リリース 1.5.1 以降で撤廃された仕様です。 `tofile()` メソッドを使ってください。

ファイルオブジェクト  $f$  に、全ての要素を (マシン依存のデータ形式そのまま) 書き込みます。

配列オブジェクトが表示される、あるいは文字列に変換されるとき、`array(typecode, initializer)` というように表現されます。配列が空ならば、`initializer` は省略されます。そうではなく `typecode` が `'c'` ならば、それは文字列になります。さもないと、数字のリストになります。`array()` 関数が `from array import array` である限り、文字列は逆クォーテーション (") を用いて同じデータ型と値を持つ配列に逆変換できることが保証されています。例:

```
array('l')
array('c', 'hello world')
array('u', u'hello \textbackslash u2641')
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14])
```

#### 参考資料:

`struct` モジュール (4.3 節):

異なる種類のバイナリデータをパックおよびアンパックします。

`xdrllib` モジュール (12.17 節):

遠隔手続き呼び出しシステムで使われる External Data Representation (XDR) データをパックおよびア



ンパックします。

*The Numerical Python Manual*

(<http://numpy.sourceforge.net/numdoc/HTML/numdoc.htm>)

Numeric Python 拡張 (NumPy) は、別の配列型を定義しています。Numerical Python についてのこれ以上の情報は、<http://numpy.sourceforge.net/> を参照してください。(NumPy マニュアルの PDF バージョンは <http://numpy.sourceforge.net/numdoc/numdoc.pdf> で手に入ります。

## 5.12 sets — ユニークな要素の順序なしコレクション

2.3 で追加された仕様です。

`sets` モジュールは、ユニークな要素の順序なしコレクションを構築し、操作するためのクラスを提供します。帰属関係のテストやシーケンスから重複を取り除いたり、積集合・和集合・差集合・対称差集合のような標準的な数学操作などを含みます。

他のコレクションのように、`x in set`, `len(set)`, `for x in set` をサポートします。順序なしコレクションは、挿入の順序や要素位置を記録しません。従って、インデックス・スライス・他のシーケンス的な振舞いをサポートしません。

ほとんどの集合のアプリケーションは、`__hash__()` を除いてすべての集合のメソッドを提供する `Set` クラスを使用します。ハッシュを要求する高度なアプリケーションについては、`ImmutableSet` クラスが `__hash__()` メソッドを加えているが、集合の内容を変更するメソッドは省略されます。`Set` と `ImmutableSet` は、何が集合 (`isinstance(obj, BaseSet)`) であるか決めるのに役立つ抽象クラス `BaseSet` から派生します。

集合クラスは辞書を使用して実装されます。その結果、集合はリストや辞書のような変更可能な要素を含むことができません。しかしそれらは、タプルや `ImmutableSet` のインスタンスのような不変コレクションを含むことができます。集合の集合の実装中の便宜については、内部集合が自動的に変更不可能な形式に変換されます。例えば、`Set([Set(['dog'])])` は `Set([ImmutableSet(['dog'])])` へ変換されます。

```
class Set([iterable])
```

新しい空の `Set` オブジェクトを構築します。もしオプション `iterable` が与えられたら、イテレータから得られた要素を備えた集合として更新します。`iterable` 中の全ての要素は、変更不可能であるか、または 5.12.3 で記述されたプロトコルを使って変更不可能なものに変換可能であるべきです。

```
class ImmutableSet([iterable])
```

新しい空の `ImmutableSet` オブジェクトを構築します。もしオプション `iterable` が与えられたら、イテレータから得られた要素を備えた集合として更新します。`iterable` 中の全ての要素は、変更不可能であるか、または 5.12.3 で記述されたプロトコルを使って変更不可能なものに変換可能であるべきです。

`ImmutableSet` オブジェクトは `__hash__()` メソッドを備えているので、集合要素または辞書キーとして使用することができます。`ImmutableSet` オブジェクトは要素を加えたり取り除いたりするメソッドを持っていません。したがって、コンストラクタが呼ばれたとき要素はすべて知られていなければなりません。

### 5.12.1 Set オブジェクト

`Set` と `ImmutableSet` のインスタンスはともに、以下の操作を備えています：



演算	等価な演算	結果
<code>len(s)</code>		集合 $s$ の濃度 (cardinality)
<code>x in s</code>		$x$ が $s$ に帰属しているかテストする
<code>x not in s</code>		$x$ が $s$ に帰属して否ないかテストする
<code>s.issubset(t)</code>	$s \leq t$	$s$ のすべての要素が $t$ にあるかテストする
<code>s.issuperset(t)</code>	$s \geq t$	$t$ のすべての要素が $s$ にあるかテストする
<code>s.union(t)</code>	$s   t$	$s$ と $t$ の両方の要素からなる新しい集合
<code>s.intersection(t)</code>	$s \& t$	$s$ と $t$ で共通する要素からなる新しい集合
<code>s.difference(t)</code>	$s - t$	$s$ にあるが $t$ にない要素からなる新しい集合
<code>s.symmetric_difference(t)</code>	$s \wedge t$	$s$ と $t$ のどちらか一方に属する要素からなる集合
<code>s.copy()</code>		$s$ の浅いコピーからなる集合

演算子を使わない書き方である `union()`、`intersection()`、`difference()`、および `symmetric_difference()` は任意のイテレータ可能オブジェクトを引数として受け取るのに対し、演算子を使った書き方の方では引数は集合型でなければならないので注意してください。これはエラーの元となる `Set('abc') & 'cbs'` のような書き方を排除し、より可読性のある `Set('abc').intersection('cbs')` を選ばせるための仕様です。2.3.1 で変更された仕様: 以前は全ての引数が集合型でなければなりませんでした。

加えて、`Set` と `ImmutableSet` は集合間の比較をサポートしています。二つの集合は、各々の集合のすべての要素が他方に含まれて (各々が他方の部分集合) いる場合、かつその場合に限り等価になります。ある集合は、他方の集合の真の部分集合 (proper subset、部分集合であるが非等価) である場合、かつその場合に限り、他方の集合より小さくなります。ある集合は、他方の集合の真の上位集合 (proper superset、上位集合であるが非等価) である場合、かつその場合に限り、他方の集合より大きくなります。

部分集合比較や等値比較では、完全な順序決定関数を一般化できません。たとえば、互いに素な2つの集合は等しくありませんし、互いの部分集合でもないのので、 $a < b$ 、 $a = b$ 、 $a > b$  はすべて `False` を返します。したがって集合は `__cmp__` メソッドを実装しません。

集合は一部の順序 (部分集合の関係) を定義するだけなので、集合のリストにおいて `list.sort()` メソッドの出力は未定義です。

以下は `ImmutableSet` で利用可能であるが `Set` にはない操作です:

演算	結果
<code>hash(s)</code>	$s$ のハッシュ値を返す

以下は `Set` で利用可能であるが `ImmutableSet` にはない操作です:

演算	等価な演算	結果
<code>s.union_update(t)</code>	$s  = t$	$t$ を加えた要素からなる集合 $s$ を返します
<code>s.intersection_update(t)</code>	$s \&= t$	$t$ でも見つかった要素だけを持つ集合 $s$ を返します
<code>s.difference_update(t)</code>	$s -= t$	$t$ にあった要素を取り除いた後の集合 $s$ を返します
<code>s.symmetric_difference_update(t)</code>	$s \wedge= t$	$s$ と $t$ のどちらか一方に属する要素からなる集合 $s$ を返します
<code>s.add(x)</code>		要素 $x$ を集合 $s$ に加えます
<code>s.remove(x)</code>		要素 $x$ を集合 $s$ から取り除きます; $x$ がなければ <code>KeyError</code> を返します
<code>s.discard(x)</code>		要素 $x$ が存在すれば、集合 $s$ から取り除きます
<code>s.pop()</code>		$s$ から要素を取り除き、それを返します; 集合が空なら <code>KeyError</code> を返します
<code>s.clear()</code>		集合 $s$ からすべての要素を取り除きます

演算子を使わない書き方である `union_update()`、`intersection_update()`、`difference_update()`、および `symmetric_difference_update()` は任意のイテレータ可能オブジェクト

トを引数として受け取るので注意してください。2.3.1 で変更された仕様: 以前は全ての引数が集合型でなければなりませんでした。

### 5.12.2 使用例

```
>>> from sets import Set
>>> engineers = Set(['John', 'Jane', 'Jack', 'Janice'])
>>> programmers = Set(['Jack', 'Sam', 'Susan', 'Janice'])
>>> managers = Set(['Jane', 'Jack', 'Susan', 'Zack'])
>>> employees = engineers | programmers | managers           # union
>>> engineering_management = engineers & managers            # intersection
>>> fulltime_management = managers - engineers - programmers # difference
>>> engineers.add('Marvin')                                   # add element
>>> print engineers
Set(['Jane', 'Marvin', 'Janice', 'John', 'Jack'])
>>> employees.issuperset(engineers)                          # superset test
False
>>> employees.union_update(engineers)                         # update from another set
>>> employees.issuperset(engineers)
True
>>> for group in [engineers, programmers, managers, employees]:
...     group.discard('Susan')                               # unconditionally remove element
...     print group
...
Set(['Jane', 'Marvin', 'Janice', 'John', 'Jack'])
Set(['Janice', 'Jack', 'Sam'])
Set(['Jane', 'Zack', 'Jack'])
Set(['Jack', 'Sam', 'Jane', 'Marvin', 'Janice', 'John', 'Zack'])
```

### 5.12.3 不変に自動変換するためのプロトコル

集合は変更不可能な要素だけを含むことができます。都合上、変更可能な Set オブジェクトは、集合要素として加えられる前に、自動的に `ImmutableSet` へコピーします。そのメカニズムはハッシュ可能な要素を常に加えることですが、もしハッシュ不可能な場合は、その要素は変更不可能な等価物を返す `__as_immutable__()` メソッドを持っているかどうかチェックされます。

Set オブジェクトは、`ImmutableSet` のインスタンスを返す `__as_immutable__()` メソッドを持っているので、集合の集合を構築することが可能です。

集合内のメンバーであることをチェックするために、要素をハッシュする必要がある `__contains__()` メソッドと `remove()` メソッドが、同様のメカニズムを必要としています。これらのメソッドは要素がハッシュできるかチェックします。もし出来なければ `__hash__()`, `__eq__()`, `__ne__()` のための一時的なメソッドを備えたクラスによってラップされた要素を返すメソッド `__as_temporarily_immutable__()` メソッドをチェックします。

代理メカニズムは、オリジナルの可変オブジェクトから分かれたコピーを組み上げる手間を助けてくれます。

Set オブジェクトは、新しいクラス `TemporarilyImmutableSet` によってラップされた Set オブジェクトを返す、`__as_temporarily_immutable__()` メソッドを実装します。

ハッシュ可能を与えるための2つのメカニズムは通常ユーザーに見えません。しかしながら、マルチスレッド環境下においては、`TemporarilyImmutableSet` によって一時的にラップされたものを持っているスレッドがあるときに、もう一つのスレッドが集合を更新することで、衝突を発生させることができます。言い換えれば、変更可能な集合の集合はスレッドセーフではありません。

## 5.13 itertools — 効率的なループ実行のためのイテレータ生成関数

2.3 で追加された仕様です。

このモジュールではイテレータを構築する部品を実装しています。プログラム言語 Haskell と SML からアイデアを得ていますが、Python に適した形に修正されています。

このモジュールは、高速でメモリ効率に優れ、単独でも組み合わせても使用することのできるツールを標準化したものです。標準化により、多数の個人が、それぞれの好みと命名規約で、それぞれ少しだけ異なる実装を行う為に発生する可読性と信頼性の問題を軽減する事ができます。

ここで定義したツールは簡単に組み合わせて使用することができるようになっており、アプリケーション固有のツールを簡潔かつ効率的に作成する事ができます。

例えば、SML の作表ツール `tabulate(f)` は `f(0)`, `f(1)`, ... のシーケンスを作成します。このツールボックスでは `imap()` と `count()` を用意しており、この二つを組み合わせると `imap(f, count())` とすれば同じ結果を得る事ができます。

同様に、`operator` モジュールの高速な関数とも一緒に使用することができるようになっています。

他にこのモジュールに追加したい基本的な構築部品があれば、開発者に提案してください。

イテレータを使用すると、Python で書いても C で書いてもリストを使用した同じ処理よりメモリ効率がよく、高速となります。これはデータをメモリ上に“在庫”しておくのではなく、必要に応じて作成する注文生産方式を採用しているためです。

イテレータによるパフォーマンス上のメリットは、要素の数が増えるにつれてより明確になります。一定以上の要素を持つリストでは、メモリキャッシュのパフォーマンスに対する影響が大きく、実行速度が低下します。

参考資料:

The Standard ML Basis Library, *The Standard ML Basis Library*.

Haskell, A Purely Functional Language, *Definition of Haskell and the Standard Libraries*.

### 5.13.1 Itertool 関数

以下の関数は全て、イテレータを作成して返します。無限長のストリームのイテレータを返す関数もあり、この場合にはストリームを中断するような関数がループ処理から使用しなければなりません。

`chain(*iterables)`

先頭の `iterable` の全要素を返し、次に 2 番目の `iterable` の全要素...と全 `iterable` の要素を返すイテレータを作成します。連続したシーケンスを、一つのシーケンスとして扱う場合に使用します。この関数は以下のスクリプトと同等です：

```
def chain(*iterables):
    for it in iterables:
        for element in it:
            yield element
```

`count([n])`

`n` で始まる、連続した整数を返すイテレータを作成します。`n` を指定しなかった場合、デフォルト値はゼロです。現在、Python の長整数はサポートしていません。`imap()` で連続したデータを生成する場合や `izip()` でシーケンスに番号を追加する場合などに引数として使用することができます。この関数は以下のスクリプトと同等です：

```
def count(n=0):
    while True:
        yield n
        n += 1
```

`count()` はオーバーフローのチェックを行いません。このため、`sys.maxint` を超えると負の値を返します。この動作は将来変更されます。

#### **`cycle(iterable)`**

`iterable` から要素を取得し、同時にそのコピーを保存するイテレータを作成します。`iterable` の全要素を返すと、セーブされたコピーから要素を返し、これを無限に繰り返します。この関数は以下のスクリプトと同等です：

```
def cycle(iterable):
    saved = []
    for element in iterable:
        yield element
        saved.append(element)
    while saved:
        for element in saved:
            yield element
```

`cycle` はこのツールキットの中で唯一、大きなメモリ領域を使用します。使用するメモリ量は `iterable` の大きさに依存します。

#### **`dropwhile(predicate, iterable)`**

`predicate` が真である限りは要素を無視し、その後は全ての要素を返すイテレータを作成します。このイテレータは、`predicate` が真の間は全く要素を返さないため、最初の要素を返すまでに長い時間がかかる場合があります。この関数は以下のスクリプトと同等です：

```
def dropwhile(predicate, iterable):
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

#### **`ifilter(predicate, iterable)`**

`predicate` が `True` となる要素だけを返すイテレータを作成します。`predicate` が `None` の場合、値が真であるアイテムだけを返します。この関数は以下のスクリプトと同等です：

```
def ifilter(predicate, iterable):
    if predicate is None:
        predicate = bool
    for x in iterable:
        if predicate(x):
            yield x
```

#### **`ifilterfalse(predicate, iterable)`**

`predicate` が `False` となる要素だけを返すイテレータを作成します。`predicate` が `None` の場合、値が偽であるアイテムだけを返します。この関数は以下のスクリプトと同等です：

```
def ifilterfalse(predicate, iterable):
    if predicate is None:
        predicate = bool
    for x in iterable:
        if not predicate(x):
            yield x
```

**imap**(*function*, \**iterables*)

*iterables* の要素を引数として *function* を呼び出すイテレータを作成します。*function* が *None* の場合、引数のタプルを返します。*map()* と似ていますが、最短の *iterable* の末尾まで到達した後は *None* を補って処理を続行するのではなく、終了します。これは、*map()* に無限長のイテレータを指定するのは多くの場合誤りですが (全出力が評価されてしまうため)、*imap()* の場合には一般的で役に立つ方法であるためです。この関数は以下のスクリプトと同等です：

```
def imap(function, *iterables):
    iterables = map(iter, iterables)
    while True:
        args = [i.next() for i in iterables]
        if function is None:
            yield tuple(args)
        else:
            yield function(*args)
```

**islice**(*iterable*, [*start*, ] *stop* [, *step* ])

*iterable* から要素を選択して返すイテレータを作成します。*start* が 0 以外であれば、*iterable* の先頭要素は *start* に達するまでスキップします。以降、*step* が 1 以下なら連続した要素を返し、1 以上なら指定された値分の要素をスキップします。*stop* が *None* であれば、無限に、もしくは *iterable* の全要素を返すまで値を返します。*None* 以外ならイテレータは指定された要素位置で停止します。通常のスライスと異なり、*start*、*stop*、*step* に負の値を指定する事はできません。配列化されたデータから関連するデータを取得する場合 (複数行からなるレポートで、三行ごとに名前が指定されている場合など) に使用します。この関数は以下のスクリプトと同等です：

```
def islice(iterable, *args):
    s = slice(*args)
    next, stop, step = s.start or 0, s.stop, s.step or 1
    for cnt, element in enumerate(iterable):
        if cnt < next:
            continue
        if stop is not None and cnt >= stop:
            break
        yield element
        next += step
```

**izip**(\**iterables*)

各 *iterable* の要素をまとめるイテレータを作成します。*zip()* に似ていますが、リストではなくイテレータを返します。複数のイテレート可能オブジェクトに対して、同じ繰り返し処理を同時に行う場合に使用します。この関数は以下のスクリプトと同等です：

```
def izip(*iterables):
    iterables = map(iter, iterables)
    while iterables:
        result = [i.next() for i in iterables]
        yield tuple(result)
```

2.3.1 で変更された仕様: イテレート可能オブジェクトを指定しない場合、TypeError 例外を送出する代わりに長さゼロのイテレータを返します。

**repeat**(*object*[, *times*])

繰り返し *object* を返すイテレータを作成します。*times* を指定しない場合、無限に値を返し続けます。*imap()* で常に同じオブジェクトを関数の引数として指定する場合に使用します。また、*izip()* で作成するタプルの全要素に常に同じオブジェクトを指定する場合にも使用することもできます。この関数は以下のスクリプトと同等です：

```
def repeat(object, times=None):
    if times is None:
        while True:
            yield object
    else:
        for i in xrange(times):
            yield object
```

**starmap**(*function*, *iterable*)

*iterables* の要素を引数として *function* を呼び出すイテレータを作成します。*function* の引数が単一の *iterable* にタプルとして格納されている場合 (“zip 済み”)、*imap()* の代わりに使用します。*imap()* と *starmap()* では *function* の呼び出し方法が異なり、*imap()* は *function(a,b)*、*starmap()* では *function(\*c)* のように呼び出します。この関数は以下のスクリプトと同等です：

```
def starmap(function, iterable):
    iterable = iter(iterable)
    while True:
        yield function(*iterable.next())
```

**takewhile**(*predicate*, *iterable*)

*predicate* が真である限り *iterable* から要素を返すイテレータを作成します。この関数は以下のスクリプトと同等です：

```
def takewhile(predicate, iterable):
    for x in iterable:
        x = iterable.next()
        if predicate(x):
            yield x
        else:
            break
```

## 5.13.2 例

以下に各ツールの一般的な使い方と、ツールの組み合わせの例を示します。



```

>>> amounts = [120.15, 764.05, 823.14]
>>> for checknum, amount in izip(count(1200), amounts):
...     print 'Check %d is for $%.2f' % (checknum, amount)
...
Check 1200 is for $120.15
Check 1201 is for $764.05
Check 1202 is for $823.14

>>> import operator
>>> for cube in imap(operator.pow, xrange(1,4), repeat(3)):
...     print cube
...
1
8
27

>>> reportlines = ['EuroPython', 'Roster', '', 'alex', '', 'laura',
                   '', 'martin', '', 'walter', '', 'samuele']
>>> for name in islice(reportlines, 3, None, 2):
...     print name.title()
...
Alex
Laura
Martin
Walter
Samuele

```

この節では、itertools を組み合わせてより強力な itertools を作り出す `enumerate()` と `iteritems()` の非常に効率的な実装は Python に含まれていますが、ここでは高レベルなツールを基本的な部品から作成する例として取り上げています。

```

def take(n, seq):
    return list(islice(seq, n))

def enumerate(iterable):
    return izip(count(), iterable)

def tabulate(function):
    "Return function(0), function(1), ..."
    return imap(function, count())

def iteritems(mapping):
    return izip(mapping.iterkeys(), mapping.itervalues())

def nth(iterable, n):
    "Returns the nth item"
    return list(islice(iterable, n, n+1))

def all(seq, pred=bool):
    "Returns True if pred(x) is True for every element in the iterable"
    return False not in imap(pred, seq)

def any(seq, pred=bool):
    "Returns True if pred(x) is True at least one element in the iterable"
    return True in imap(pred, seq)

def no(seq, pred=bool):
    "Returns True if pred(x) is False for every element in the iterable"
    return True not in imap(pred, seq)

def quantify(seq, pred=bool):
    "Count how many times the predicate is True in the sequence"
    return sum(imap(pred, seq))

def padnone(seq):
    "Returns the sequence elements and then returns None indefinitely"
    return chain(seq, repeat(None))

def ncycles(seq, n):
    "Returns the sequence elements n times"
    return chain(*repeat(seq, n))

def dotproduct(vec1, vec2):
    return sum(imap(operator.mul, vec1, vec2))

def window(seq, n=2):
    "Returns a sliding window (of width n) over data from the iterable"
    "  s -> (s0,s1,...s[n-1]), (s1,s2,...,sn), ..."
    it = iter(seq)
    result = tuple(islice(it, n))
    if len(result) == n:
        yield result
    for elem in it:
        result = result[1:] + (elem,)
        yield result

def tee(iterable):
    "Return two independent iterators from a single iterable"
    def gen(next, data={}, cnt=[0]):
        dpop = data.pop
        for i in count():
            if i == cnt[0]:
                item = data[i] = next()
                cnt[0] += 1
            else:
                item = dpop(i)
            yield item
    next = iter(iterable).next
    return (gen(next), gen(next))

```

## 5.14 ConfigParser — 設定ファイルの構文解析器

このモジュールでは、ConfigParser クラスを定義しています。ConfigParser クラスは、Microsoft Windows の INI ファイルに見られるような構造をもつ、基礎的な設定ファイルを実装しています。このモジュールを使って、エンドユーザーが簡単にカスタマイズできるような Python プログラムを書くことができます。

警告: このライブラリでは、Windows のレジストリ用に拡張された INI 文法はサポートしていません。

設定ファイルは 1 つ以上のセクションからなり、セクションは `'[section]'` ヘッダとそれに続く RFC 822 形式の `'name: value'` エントリからなっています。`'name=value'` という形式も使えます。値の先頭にある空白文字は削除されるので注意してください。オプションの値には、同じセクションか DEFAULT セクションにある値を参照するような書式化文字列を含めることができます。初期化時や検索時に別のデフォルト値を与えることもできます。`'#'` か `';'` で始まる行は無視され、コメントを書くために利用できます。

例:

```
[My Section]
foodir: %(dir)s/whatever
dir=frob
```

この場合 `'%(dir)s'` は変数 `'dir'` (この場合は `'frob'`) に展開されます。参照の展開は必要に応じて実行されます。

デフォルト値は ConfigParser のコンストラクタに辞書として渡すことで設定できます。追加の (他の値をオーバーライドする) デフォルト値は `get()` メソッドに渡すことができます。

**class RawConfigParser([defaults])**

基本的な設定オブジェクトです。defaults が与えられた場合、オブジェクトに固有のデフォルト値がその値で初期化されます。このクラスは値の置換をサポートしません。2.3 で追加された仕様です。

**class ConfigParser([defaults])**

RawConfigParser の派生クラスで値の置換を実装しており、`get()` メソッドと `items()` メソッドに省略可能な引数を追加しています。defaults に含まれる値は `'%( )s'` による値の置換に適当なものである必要があります。\_\_name\_\_ は組み込みのデフォルト値で、セクション名が含まれるので defaults で設定してもオーバーライドされます。

**class SafeConfigParser([defaults])**

ConfigParser の派生クラスでより安全な値の置換を実装しています。この実装のはより予測可能性が高くなっています。新規に書くアプリケーションでは、古いバージョンの Python と互換性を持たせる必要がない限り、このバージョンを利用することが望ましいです。2.3 で追加された仕様です。

**exception NoSectionError**

指定したセクションが見つからなかった時に起きる例外です。

**exception DuplicateSectionError**

同じ名前をもつセクションが複数見つかった時や、すでに存在するセクション名に対して `add_section()` が呼び出された際に起きる例外です。

**exception NoOptionError**

指定したオプションが指定したセクションに存在しなかった時に起きる例外です。

**exception InterpolationError**

文字列の置換中に問題が起きた時に発生する例外の基底クラスです。

**exception InterpolationDepthError**

InterpolationError の派生クラスで、文字列の置換回数が MAX\_INTERPOLATION\_DEPTH を越えたために完了しなかった場合に発生する例外です。

**exception InterpolationMissingOptionError**

InterpolationError の派生クラスで、値が参照しているオプションが見つからない場合に発生する例外です。

**exception InterpolationSyntaxError**

InterpolationError の派生クラスで、指定された構文で値を置換することができなかった場合に発生する例外です。2.3 で追加された仕様です。

**exception MissingSectionHeaderError**

セクションヘッダを持たないファイルを構文解析しようとした時に起きる例外です。

**exception ParsingError**

ファイルの構文解析中にエラーが起きた場合に発生する例外です。

**MAX\_INTERPOLATION\_DEPTH**

*raw* が偽だった場合の *get()* による再帰的な文字列置換の繰り返しの最大値です。ConfigParser クラスだけに関係します。

参考資料:

shlex モジュール (5.19 節):

UNIX のシェルに似た、アプリケーションの設定ファイル用フォーマットとして使えるもう一つの小型言語です。

### 5.14.1 RawConfigParser オブジェクト

RawConfigParser クラスのインスタンスは以下のメソッドを持ちます:

**defaults()**

インスタンス全体で使われるデフォルト値の辞書を返します。

**sections()**

利用可能なセクションのリストを返します。DEFAULT はこのリストに含まれません。

**add\_section(section)**

*section* という名前のセクションをインスタンスに追加します。同名のセクションが存在した場合、DuplicateSectionError が発生します。

**has\_section(section)**

指定したセクションがコンフィグレーションファイルに存在するかを返します。DEFAULT セクションは存在するとみなされません。

**options(section)**

*section* で指定したセクションで利用できるオプションのリストを返します。

**has\_option(section, option)**

与えられたセクションが存在してかつオプションが与えられていれば 1 を返し、そうでなければ 0 を返します。1.6 で追加された仕様です。

**read(filenames)**

ファイル名のリストを読んで解析します。もし *filenames* が文字列かユニコード文字列なら、1 つのファイル名として扱われます。*filenames* で指定されたファイルが開けない場合、そのファイルは無視されます。この挙動は設定ファイルが置かれる可能性のある場所 (例えば、カレントディレクトリ、

ホームディレクトリ、システム全体の設定を行うディレクトリ)を設定して、そこに存在する設定ファイルを読むことを想定して設計されています。設定ファイルが存在しなかった場合、ConfigParserのインスタンスは空のデータセットを持ちます。初期値の設定ファイルを先に読み込んでおく必要があるアプリケーションでは、readfp())をread()の前に呼び出すことでそのような動作を実現できます:

```
import ConfigParser, os

config = ConfigParser.ConfigParser()
config.readfp(open('defaults.cfg'))
config.read(['site.cfg', os.path.expanduser('~/.myapp.cfg')])
```

**readfp**(fp[, filename])

fpで与えられるファイルかファイルのようなオブジェクトを読み込んで構文解析します(readline()メソッドだけを使います)。もしfilenameが省略されてfpがname属性を持っていればfilenameの代わりに使われます。ファイル名の初期値は'<???'です。

**get**(section, option)

sectionのoption変数を取得します。

**getint**(section, option)

sectionのoptionを整数として評価する関数です。

**getfloat**(section, option)

sectionのoptionを浮動小数点数として評価する関数です。

**getboolean**(section, option)

指定したsectionのoption値をブール値に型強制する便宜メソッドです。optionとして受理できる値は、真(True)としては"1"、"yes"、"true"、"on"、偽(False)としては"0"、"no"、"false"、"off"です。これらの文字列値に対しては大文字小文字の区別をしません。その他の値の場合にはValueErrorを送出します。

**items**(section)

与えられたsectionのそれぞれのオプションについて(name, value)ペアのリストを返します。

**set**(section, option, value)

与えられたセクションが存在していれば、オプションを指定された値に設定します。セクションが存在しなければNoSectionErrorが発生します。1.6で追加された仕様です。

**write**(fileobject)

設定を文字列表現に変換してファイルオブジェクトに書き出します。この文字列表現はread()で読み込むことができます。1.6で追加された仕様です。

**remove\_option**(section, option)

指定されたsectionから指定されたoptionを削除します。セクションが存在しなければ、NoSectionErrorを起こします。存在するオプションを削除した時は1、そうでない時は0を返します。1.6で追加された仕様です。

**remove\_section**(section)

指定されたsectionを設定から削除します。もし指定されたセクションが存在すればTrue、そうでなければFalseを返します。

**optionxform**(option)

入力ファイル中に見つかったオプション名か、クライアントコードから渡されたオプション名optionを、内部で利用する形式に変換します。デフォルトではoptionを全て小文字に変換した名前が返されます。サブクラスではこの関数をオーバーライドすることでこの振舞いを替えることができます。

たとえば、このメソッドを `str()` に設定することで大小文字の差を区別するように変更することができます。

### 5.14.2 ConfigParser オブジェクト

`ConfigParser` クラスは `RawConfigParser` のインターフェースをいくつかのメソッドについて拡張し、省略可能な引数を追加しています。

`get(section, option[, raw[, vars]])`

`section` の `option` 変数を取得します。`raw` が真でない時には、全ての '%' 置換はコンストラクタに渡されたデフォルト値か、`vars` が与えられていればそれを元にして展開されてから返されます。

`items(section[, raw[, vars]])`

指定した `section` 内の各オプションに対して、`(name, value)` のペアからなるリストを返します。省略可能な引数は `get()` メソッドと同じ意味を持ちます。2.3 で追加された仕様です。

## 5.15 fileinput — 複数の入力ストリームをまたいだ行の繰り返し処理をサポートする。

このモジュールは標準入力やファイルの並びにまたがるループを素早く書くためのヘルパークラスと関数を提供しています。

典型的な使い方は以下の通りです:

```
import fileinput
for line in fileinput.input():
    process(line)
```

このプログラムは `sys.argv[1:]` に含まれる全てのファイルをまたいで繰り返します。もし該当するものがなければ、`sys.stdin` がデフォルトとして扱われます。ファイル名として '-' が与えられた場合も、`sys.stdin` に置き換えられます。別のファイル名リストを使いたい時には、`input()` の最初の引数にリストを与えます。単一ファイル名の文字列も受け付けます。

全てのファイルはテキストモードでオープンされます。オープン中あるいは読み込み中に I/O エラーが発生した場合には、`IOError` が発生します。

`sys.stdin` が 2 回以上使われた場合は、2 回目以降は行を返しません。ただしインタラクティブに利用している時や明示的にリセット (`sys.stdin.seek(0)`) を使う) を行った場合はその限りではありません。

空のファイルは開いた後すぐ閉じられます。空のファイルはファイル名リストの最後にある場合にしか外部に影響を与えません。

ファイルの最後が改行文字で終わっていない場合には、最後に改行文字を追加して返します。

以下の関数がこのモジュールの基本的なインタフェースです:

`input([files[, inplace[, backup]]])`

`FileInput` クラスのインスタンスを作ります。生成されたインスタンスは、このモジュールの関数群が利用するグローバルな状態として利用されます。この関数への引数は `FileInput` クラスのコンストラクタへ渡されます。

以下の関数は `input()` 関数によって作られたグローバルな状態を利用します。アクティブな状態が無い場合には、`RuntimeError` が発生します。



`filename()`

現在読み込み中のファイル名を返します。一行目が読み込まれる前は `None` を返します。

`lineno()`

最後に読み込まれた行の、累積した行番号を返します。1 行目が読み込まれる前は 0 を返します。最後のファイルの最終行が読み込まれた後には、その行の行番号を返します。

`filelineno()`

現在のファイル中での行番号を返します。1 行目が読み込まれる前は 0 を返します。最後のファイルの最終行が読み込まれた後には、その行のファイル中での行番号を返します。

`isfirstline()`

最後に読み込まれた行がファイルの 1 行目なら `True`、そうでなければ `False` を返します。

`isstdin()`

最後に読み込まれた行が `sys.stdin` から読まれていれば `True`、そうでなければ `False` を返します。

`nextfile()`

現在のファイルを閉じます。次の繰り返しでは (存在すれば) 次のファイルの最初の行が読み込まれます。閉じたファイルの読み込まれなかった行は、累積の行数にカウントされません。ファイル名は次のファイルの最初の行が読み込まれるまで変更されません。最初の行の読み込みが行われるまでは、この関数は呼び出されても何もみませんので、最初のファイルをスキップするために利用することはできません。最後のファイルの最終行が読み込まれた後にも、この関数は呼び出されても何もみません。

`close()`

シーケンスを閉じます。

このモジュールのシーケンスの振舞いを実装しているクラスのサブクラスを作することもできます。

`class FileInput([files[, inplace[, backup]]])`

`FileInput` クラスはモジュールの関数に対応するメソッド `filename()`、`lineno()`、`filelineno()`、`isfirstline()`、`isstdin()`、`nextfile()`、`close()` を実装しています。それに加えて、次の入力行を返す `readline()` メソッドと、シーケンスの振舞いの実装をしている `__getitem__()` メソッドがあります。シーケンスはシーケンシャルに読み込むことしかできません。つまりランダムアクセスと `readline()` を混在させることはできません。

その場で保存するオプション機能:

キーワード引数 `inplace=1` が `input()` か `FileInput` クラスのコンストラクタに渡された場合には、入力ファイルはバックアップファイルに移動され、標準出力が入力ファイルに設定されます (バックアップファイルと同じ名前のファイルが既に存在していた場合には、警告無しに置き換えられます)。これによって入力ファイルをその場で書き替えるフィルタを書くことができます。キーワード引数 `backup='.<拡張子>'` も与えられていれば、バックアップファイルの拡張子を決めることができます。デフォルトでは `'.bak'` になっています。出力先のファイルが閉じられればバックアップファイルは消されます。その場で保存する機能は、標準入力を読み込んでいる間は無効にされます。

警告: 現在の実装は MS-DOS の 8+3 ファイルシステムでは動作しません。

## 5.16 xreadlines — ファイルの各行に対する効率のよい反復処理

2.1 で追加された仕様です。

リリース 2.3 以降で撤廃された仕様です。 `for line in file` を代わりに使ってください。

このモジュールでは、あるファイル内の各行に対して効率的に反復を行えるような新たなオブジェクト型を定義します。 `xreadline` オブジェクトは `for` 文や関数 `filter()` で必要とされるような、0 から始ま

る単純かつ順序正しいインデックス付けを実装した配列型です。

つまり、以下のコード

```
import xreadlines, sys

for line in xreadlines.xreadlines(sys.stdin):
    pass
```

は、

```
while 1:
    lines = sys.stdin.readlines(8*1024)
    if not lines: break
    for line in lines:
        pass
```

とだいたい同じ速度とメモリ消費になります。ただし前者は `for` 文のままなのでより明快です。

**xreadlines**(*fileobj*)

*fileobj* の内容にわたって反復を行うような、新しい `xreadlines` オブジェクトを返します。*fileobj* は、引数 *sizehint* をサポートする `readlines()` メソッドを持っていないなりません。注意: `readlines()` メソッドはデータをバッファするので、ファイルオブジェクトを非バッファリングに設定しても無視されます。

`xreadlines` オブジェクト *s* は以下の配列操作をサポートします:

操作	結果
<i>s</i> [ <i>i</i> ]	<i>s</i> の <i>i</i> 行目

*i* の値が 0 から始まる連続した数でない場合、このコードは例外 `RuntimeError` を送出します。

ファイルの最後の行が読み出された後、このコードは例外 `IndexError` を送出します。

## 5.17 calendar — 一般的なカレンダーに関する関数群

このモジュールは UNIX の `cal` プログラムのようなカレンダー出力を行い、それに加えてカレンダーに関する有益な関数群を提供します。標準ではこれらのカレンダーは (ヨーロッパの慣例に従って) 月曜日を週の始まりとし、日曜日を最後の日としています。 `setfirstweekday()` を用いることで、日曜日 (6) や他の曜日を週の始まりに設定することができます。日付を表す引数は整数値で与えます。

このモジュールで提供する関数のほとんどは `datetime` に依存しており、過去も未来も現代のグレゴリオ暦を利用します。この方式は Dershowitz と Reingold の書籍「*Calendrical Calculations*」にある *proleptic Gregorian* 暦に一致しており、同書では全ての計算の基礎となる暦としています。

**setfirstweekday**(*weekday*)

週の最初の曜日 (0 は月曜日, 6 は日曜日) を設定します。定数 `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` 及び `SUNDAY` は便宜上提供されています。例えば、日曜日を週の開始日に設定するとき:

```
import calendar
calendar.setfirstweekday(calendar.SUNDAY)
```

2.0 で追加された仕様です。

**firstweekday()**

現在設定されている週の最初の曜日を返します。2.0 で追加された仕様です。

**isleap(year)**

year が閏年なら 1 を、そうでなければ 0 を返します。

**leapdays(y1, y2)**

範囲 (y1...y2) 指定された期間の閏年の回数を返します。ここで y1 や y2 は年を表します。2.0 で変更された仕様: Python 1.5.2 では、この関数は世紀をまたがった範囲では動作しません。

**weekday(year, month, day)**

year(1970-...), month (1-12), day(1-31) で与えられた日の曜日 (0 は月曜日) を返します。

**monthrange(year, month)**

year と month で指定された月の一日の曜日と日数を返します。

**monthcalendar(year, month)**

月のカレンダーを行列で返します。各行が週を表し、月の範囲外の日は 0 になります。それぞれの週は setfirstweekday() で設定をしていない限り月曜日から始まります。

**prmonth(theyear, themonth[, w[, l]])**

month() 関数によって返される月のカレンダーを出力します。

**month(theyear, themonth[, w[, l]])**

月のカレンダーを複数行の文字列で返します。w により日の列幅を変えることができ、それらはセンタリングされます。l により各週の表示される行数を変えることができます。週の最初の曜日は setfirstweekday() 関数の設定に依存します。2.0 で追加された仕様です。

**prcal(year[, w[, l[, c]]])**

calendar() 関数で返される一年間のカレンダーを出力します。

**calendar(year[, w[, l[, c]]])**

3 列からなる一年間のカレンダーを複数行の文字列で返します。任意の引数 w, l, 及び c はそれぞれ、日付列の表示幅、各週の行数及び月と月の間のスペースの数を変更するためのものです。週の最初の曜日は setfirstweekday() 関数の設定に依存します。出力されるカレンダーの起点となる年はプラットフォームに依存します。2.0 で追加された仕様です。

**timegm(tuple)**

関連はありませんが便利な関数で、time モジュールの gmtime() 関数の戻値のような時間のタプルを受け取り、1970 年を起点とし、POSIX 規格のエンコードによる UNIX のタイムスタンプに相当する値を返します。実際、time.gmtime() と timegm() は反対の動作をします。2.0 で追加された仕様です。

参考資料:

time モジュール (6.10 節):

低レベルの時間に関連した関数群。

## 5.18 cmd — 行指向のコマンドインタプリタのサポート

Cmd クラスでは、行指向のコマンドインタプリタを書くための簡単なフレームワークを提供します。テスト用の仕掛けや管理ツール、そして、後により洗練されたインターフェイスでラップするプロトタイプとして、こうしたインタプリタはよく役に立ちます。

**class Cmd([completekey], [stdin], [stdout])**

Cmd インスタンス、あるいはサブクラスのインスタンスは、行指向のインタプリタ・フレームワークです。Cmd 自身をインスタンス化することはありません。むしろ、Cmd のメソッドを継承したり、

アクションメソッドをカプセル化するために、あなたが自分で定義するインタープリタクラスのスーパークラスとしての便利です。

オプション引数 *completekey* は、補完キーの *readline* 名です。デフォルトは *Tab* です。*completekey* が *None* でなく、*readline* が利用できるならば、コマンド補完は自動的に行われます。

! The optional arguments *stdin* and *stdout* specify the ! input and output file objects that the *Cmd* instance or subclass ! instance will use for input and output. If not specified, they ! will default to *sys.stdin* and *sys.stdout*. ! オプション引数 *stdin* と *stdout* には、*Cmd* インスタンス又はサブクラスのインスタンスが入出力に使用するファイルオブジェクトを指定します。省略時には *sys.stdin* と *sys.stdout* を使用します。

2.3 で変更された仕様: 引数 *stdin* と *stdout* を追加

### 5.18.1 Cmd オブジェクト

*Cmd* インスタンスは、次のメソッドを持ちます:

**cmdloop**(*[intro]*)

プロンプトを繰り返し出し、入力を受け取り、受け取った入力から取り去った先頭の語を解析し、その行の残りを引数としてアクションメソッドへディスパッチします。

オプションの引数は、最初のプロンプトの前に表示されるバナーあるいは紹介用の文字列です (これはクラスメンバ *intro* をオーバーライドします)。

*readline* モジュールがロードされているなら、入力は自動的に *bash* のような履歴リスト編集機能を受け継ぎます (例えば、*Control-P* は直前のコマンドへのスクロールバック、*Control-N* は次のものへ進む、*Control-F* はカーソルを右へ非破壊的に進める、*Control-B* はカーソルを非破壊的に左へ移動させる等)。

入力のファイル終端は、文字列 '*EOF*' として渡されます。

メソッド *do\_foo()* を持っている場合に限って、インタープリタのインスタンスはコマンド名 '*foo*' を認識します。特別な場合として、文字 '?' で始まる行はメソッド *do\_help()* へディスパッチします。他の特別な場合として、文字 '!' で始まる行はメソッド *do\_shell()* へディスパッチします (このようなメソッドが定義されている場合)。

補完が有効になっているなら、コマンドの補完が自動的に行われます。また、コマンド引数の補完は、引数 *text*、*line*、*begidx*、および *endidx* と共に *complete\_foo()* を呼び出すことによって行われます。*text* は、我々がマッチしようとしている文字列の先頭の語です。返されるマッチは全てそれで始まっていなければなりません。*line* は始めの空白を除いた現在の入力行です。*begidx* と *endidx* は先頭のテキストの始まりと終わりのインデックスで、引数の位置に依存した異なる補完を提供するのに使えます。

*Cmd* のすべてのサブクラスは、定義済みの *do\_help()* を継承します。このメソッドは、(引数 '*bar*' と共に呼ばれたとすると) 対応するメソッド *help\_bar()* を呼び出します。引数がなければ、*do\_help()* は、すべての利用可能なヘルプ見出し (すなわち、対応する *help\_\**() メソッドを持つすべてのコマンド) をリストアップします。また、文書化されていないコマンドでも、すべてリストアップします。

**onecmd**(*str*)

プロンプトに答えてタイプしたかのように引数を解釈実行します。これをオーバーライドすることがあるかもしれませんが、通常は必要ないでしょう。便利な実行フックについては、*precmd()* と *postcmd()* メソッドを参照してください。戻り値は、インタープリタによるコマンドの解釈実行をやめるかどうかを示すフラグです。

**emptyline()**

プロンプトに空行が入力されたときに呼び出されるメソッド。このメソッドがオーバーライドされていないなら、最後に入力された空行でないコマンドが繰り返されます。

**default(line)**

コマンドの先頭の語が認識されないときに、入力行に対して呼び出されます。このメソッドがオーバーライドされていないなら、エラーメッセージを表示して戻ります。

**completedefault(text, line, begidx, endidx)**

利用可能なコマンド固有の `complete_*`() が存在しないときに、入力行を補完するために呼び出されるメソッド。デフォルトでは、空行を返します。

**precmd(line)**

コマンド行 *line* が解釈実行される直前、しかし入力プロンプトが作られ表示された後に実行されるフックメソッド。このメソッドは `Cmd` 内のスタブであって、サブクラスでオーバーライドされるために存在します。戻り値は `onecmd()` メソッドが実行するコマンドとして使われます。`precmd()` の実装では、コマンドを書き換えるかもしれないし、あるいは単に変更していない *line* を返すかもしれません。

**postcmd(stop, line)**

コマンドディスパッチが終わった直後に実行されるフックメソッド。このメソッドは `Cmd` 内のスタブで、サブクラスでオーバーライドされるために存在します。*line* は実行されたコマンド行で、*stop* は `postcmd()` の呼び出しの後に実行を停止するかどうかを示すフラグです。これは `onecmd()` メソッドの戻り値です。このメソッドの戻り値は、*stop* に対応する内部フラグの新しい値として使われます。偽を返すと、実行を続けます。

**preloop()**

`cmdloop()` が呼び出されたときに一度だけ実行されるフックメソッド。このメソッドは `Cmd` 内のスタブであって、サブクラスでオーバーライドされるために存在します。

**postloop()**

`cmdloop()` が戻る直前に一度だけ実行されるフックメソッド。このメソッドは `Cmd` 内のスタブであって、サブクラスでオーバーライドされるために存在します。

`Cmd` のサブクラスのインスタンスは、公開されたインスタンス変数をいくつか持っています:

**prompt**

入力を求めるために表示されるプロンプト。

**identchars**

コマンドの先頭の語として受け入れられる文字の文字列。

**lastcmd**

最後の空でないコマンドプリフィックス。

**intro**

紹介またはバナーとして表示される文字列。`cmdloop()` メソッドに引数を与えるために、オーバーライドされるかもしれません。

**doc\_header**

ヘルプの出力に文書化されたコマンドの部分がある場合に表示するヘッダ。

**misc\_header**

ヘルプの出力にその他のヘルプ見出しがある (すなわち、`do_*`() メソッドに対応していない `help_*`() メソッドが存在する) 場合に表示するヘッダ。

**undoc\_header**

ヘルプの出力に文書化されていないコマンドの部分がある (すなわち、対応する `help_*`() メソッド



を持たない `do_*`( ) メソッドが存在する) 場合に表示するヘッダ。

#### **ruler**

ヘルプメッセージのヘッダの下に、区切り行を表示するために使われる文字。空のときは、ルーラ行が表示されません。デフォルトでは、`'=`' です。

#### **use\_rawinput**

フラグ、デフォルトでは真。真ならば、`cmdloop()` はプロンプトを表示して次のコマンド読み込むために `raw_input()` を使います。偽ならば、`sys.stdout.write()` と `sys.stdin.readline()` が使われます。(これが意味するのは、`readline` を import することによって、それをサポートするシステム上では、インタプリタが自動的に Emacs 形式の行編集とコマンド履歴のキーストロークをサポートするということです。)

## 5.19 shlex — 単純な字句解析

1.5.2 で追加された仕様です。

`shlex` クラスは UNIX シェルを思わせる単純な構文に対する字句解析器を簡単に書けるようにします。このクラスはしばしば、Python アプリケーションのための実行制御ファイルのような、小規模言語を書く上で便利です。

参考資料:

`ConfigParser` モジュール (5.14 節):

Windows `'ini'` ファイルに似た設定ファイルのパパーザ。

### 5.19.1 モジュールの内容

`shlex` モジュールは以下の関数を定義します。

**split**(*s* [, *comments=False* ])

シェル類似の文法を使って、文字列 *s* を分割します。*comments* が `False` の場合、受理した文字列内のコメントを解析しません (`shlex` インスタンスの `commenters` メンバの値を空文字列にします)。この関数は POSIX モードで動作します。2.3 で追加された仕様です。

`shlex` モジュールは以下のクラスを定義します。

**class shlex**( [*instream=sys.stdin* [, *infile=None* [, *posix=False* ]]] )

`shlex` クラスとサブクラスのインスタンスは、字句解析器オブジェクトです。初期化引数を与えると、どこから文字を読み込むかを指定できます。指定先は `read()` メソッドと `readline()` メソッドを持つファイル/ストリーム類似オブジェクトか、文字列でなくてはなりません (文字列が受理されるようになったのは Python 2.3 以降)。引数が与えられなければ、`sys.stdin` から入力を受け付けます。第 2 引数は、ファイル名を表す文字列で、`infile` メンバの値の初期値を決定します。*instream* 引数が省略された場合や、この値が `sys.stdin` である場合、第 2 引数のデフォルト値は `"stdin"` になります。*posix* 引数は Python 2.3 で導入されました。これは動作モードを定義します。*posix* が真でない場合 (デフォルト) `shlex` インスタンスは互換モードで動作します。POSIX モードで動作中、`shlex` は、できる限り POSIX シェルの解析規則に似せようとします。5.19.2 を参照のこと。

### 5.19.2 shlex オブジェクト

`shlex` インスタンスは以下のメソッドを持っています:

**get\_token**( )

トークンを一つ返します。トークンが `push_token()` で使ってスタックに積まれていた場合、トー



クンをスタックからポップします。そうでない場合、トークンを一つ入力ストリームから読み出します。読み出し即時にファイル終了子に遭遇した場合、`self.eof` (非 POSIX モードでは空文字列 (")、POSIX モードでは `None`) が返されます。

`push_token(str)`

トークンスタックに引数文字列をスタックします。

`read_token()`

生 (raw) のトークンを読み出します。プッシュバックスタックを無視し、かつソースリクエストを解釈しません (通常これは便利なエントリポイントではありません。完全性のためにここで記述されています)。

`sourcehook(filename)`

`shlex` がソースリクエスト (下の `source` を参照してください) を検出した際、このメソッドはその後に続くトークンを引数として渡され、ファイル名と開かれたファイル類似オブジェクトからなるタプルを返すとされています。

通常、このメソッドはまず引数から何らかのクォートを剥ぎ取ります。処理後の引数が絶対パス名であった場合か、以前に有効になったソースリクエストが存在しない場合か、以前のソースが (`sys.stdin` のような) ストリームであった場合、この結果はそのままにされます。そうでない場合で、処理後の引数が相対パス名の場合、ソースインクルードスタックにある直前のファイル名からディレクトリ部分を取り出され、相対パスの前の部分に追加されます (この動作は C 言語プリプロセッサにおける `#include "file.h"` の扱いと同様です)。

これらの操作の結果はファイル名として扱われ、タプルの最初の要素として返されます。同時にこのファイル名で `open()` を呼び出した結果が二つ目の要素になります (注意: インスタンス初期化のときは引数の並びが逆になっています！)

このフックはディレクトリサーチパスや、ファイル拡張子の追加、その他の名前空間に関するハックを実装できるようにするために公開されています。‘close’ フックに対応するものではありませんが、`shlex` インスタンスはソースリクエストされている入力ストリームが EOF を返した時には `close()` を呼び出します。

ソーススタックをより明示的に操作するには、`push_source()` および `pop_source()` メソッドを使ってください。

`push_source(stream[, filename])`

入力ソースストリームを入力スタックにプッシュします。ファイル名引数が指定された場合、以後のエラーメッセージ中で利用することができます。`sourcehook` メソッドが内部で使用しているのと同じメソッドです。2.1 で追加された仕様です。

`pop_source()`

最後にプッシュされた入力ソースを入力スタックからポップします。字句解析器がスタック上の入力ストリームの EOF に到達した際に利用するメソッドと同じです。2.1 で追加された仕様です。

`error_leader([file[, line]])`

このメソッドはエラーメッセージの論述部分を UNIX C コンパイラエラーラベルの形式で生成します; この書式は `'"%s", line %d: '` で、‘%s’ は現在のソースファイル名で置き換えられ、‘%d’ は現在の入力行番号で置き換えられます (オプションの引数を使ってこれらを上書きすることもできます)。

このやり方は、`shlex` のユーザに対して、Emacs やその他の UNIX ツール群が解釈できる一般的な書式でのメッセージを生成することを推奨するために提供されています。

`shlex` サブクラスのインスタンスは、字句解析を制御したり、デバッグに使えるような public なインスタンス変数を持っています:

#### **commenters**

コメントの開始として認識される文字列です。コメントの開始から行末までのすべてのキャラクタ文字は無視されます。標準では単に '#' が入っています。

#### **wordchars**

複数文字からなるトークンを構成するためにバッファに蓄積していくような文字からなる文字列です。標準では、全ての ASCII 英数字およびアンダースコアが入っています。

#### **whitespace**

空白と見なされ、読み飛ばされる文字群です。空白はトークンの境界を作ります。標準では、スペース、タブ、改行 (linefeed) および復帰 (carriage-return) が入っています。

#### **escape**

エスケープ文字と見なされる文字群です。これは POSIX モードでのみ使われ、デフォルトでは '\' だけが入っています。2.3 で追加された仕様です。

#### **quotes**

文字列引用符と見なされる文字群です。トークンを構成する際、同じクオートが再び出現するまで文字をバッファに蓄積します (すなわち、異なるクオート形式はシェル中で互いに保護し合う関係にあります)。標準では、ASCII 単引用符および二重引用符が入っています。

#### **escapedquotes**

quotes のうち、escape で定義されたエスケープ文字を解釈する文字群です。これは POSIX モードでのみ使われ、デフォルトでは '"' だけが入っています。2.3 で追加された仕様です。

#### **whitespace\_split**

この値が True であれば、トークンは空白文字でのみで分割されます。たとえば shlex がシェル引数と同じ方法で、コマンドラインを解析するのに便利です。2.3 で追加された仕様です。

#### **infile**

現在の入力ファイル名です。クラスのインスタンス化時に初期設定されるか、その後のソースリクエストでスタックされます。エラーメッセージを構成する際にこの値を調べると便利ことがあります。

#### **instream**

shlex インスタンスが文字を読み出している入力ストリームです。

#### **source**

このメンバ変数は標準で None を取ります。この値に文字列を代入すると、その文字列は多くのシェルにおける 'source' キーワードに似た、字句解析レベルでのインクルード要求として認識されます。すなわち、その直後に現れるトークンをファイル名として新たなストリームを開き、そのストリームを入力として、EOF に到達するまで読み込まれます。新たなストリームの EOF に到達した時点で close() が呼び出され、入力元の入力ストリームに戻されます。ソースリクエストは任意のレベルの深さまでスタックしてかまいません。

#### **debug**

このメンバ変数が数値で、かつ 1 またはそれ以上の値の場合、shlex インスタンスは動作に関する冗長な進捗報告を出力します。この出力を使いたいなら、モジュールのソースコードを読めば詳細を学ぶことができます。

#### **lineno**

ソース行番号 (遭遇した改行の数に 1 を加えたもの) です。

#### **token**

トークンバッファです。例外を捕捉した際にこの値を調べると便利ことがあります。

#### **eof**

ファイルの終端を決定するのに使われるトークンです。非 POSIX モードでは空文字列 (" )、POSIX

モードでは None が入ります。

### 5.19.3 解析規則

非 POSIX モードで動作中の shlex は以下の規則に従おうとします。

- ワード内の引用符を認識しない (Do "Not" Separate は単一ワード Do "Not" Separate として解析されます)
- エスケープ文字を認識しない
- 引用符で囲まれた文字列は、引用符内の全ての文字リテラルを保持する
- 閉じ引用符でワードを区切る ("Do" Separate は、"Do" と Separate であると解析されます)
- whitespace\_split が False の場合、wordchar、whitespace または quote として宣言されていない全ての文字を、単一の文字トークンとして返す。True の場合、shlex は空白文字でのみ単語を区切る。
- 空文字列 (") で EOF を送出する
- 引用符に囲んであっても、空文字列を解析しない

POSIX モードで動作中の shlex は以下の解析規則に従おうとします。

- 引用符を取り除き、引用符で単語を分解しない ("Do "Not" Separate" は単一ワード DoNotSeparate として解析されます)
- 引用符で囲まれないエスケープ文字群 ('\' など) は直後に続く文字のリテラル値を保持する
- escapedquotes でない引用符文字 ('\' など) で囲まれている全ての文字のリテラル値を保持する
- 引用符に囲まれた escapedquotes に含まれる文字 ('\" など) は、escape に含まれる文字を除き、全ての文字のリテラル値を保持する。エスケープ文字群は使用中の引用符、または、そのエスケープ文字自身が直後にある場合のみ、特殊な機能を保持する。他の場合にはエスケープ文字は普通の文字とみなされる。
- None で EOF を送出する
- 引用符に囲まれた空文字列 (") を許す



# 汎用オペレーティングシステムサービス

本章に記述されたモジュールは、ファイルの取り扱いや時間計測のような (ほぼ) すべてのオペレーティングシステムで利用可能な機能にインターフェースを提供します。これらのインターフェースは、UNIX もしくは C のインターフェースを基に作られますが、ほとんどの他のシステムで同様に利用可能です。概要を以下に記述します。

<code>os</code>	雑多なオペレーティングシステムインタフェース。
<code>os.path</code>	共通のパス名操作。
<code>dircache</code>	キャッシュメカニズムを備えたディレクトリー一覧生成。
<code>stat</code>	<code>os.stat()</code> 、 <code>os.lstat()</code> および <code>os.fstat()</code> の返す内容を解釈するためのユーティリティ。
<code>statcache</code>	ファイルの <code>stat</code> を調べ、その結果を記憶します。
<code>statvfs</code>	<code>os.statvfs()</code> の返す値を解釈するために使われる定数群。
<code>filecmp</code>	ファイル群を効率的に比較します。
<code>popen2</code>	アクセス可能な I/O ストリームを持つ子プロセス生成。
<code>datetime</code>	基本的な日付型および時間型。
<code>time</code>	時刻データへのアクセスと変換
<code>sched</code>	一般的な目的のためのイベントスケジューラ
<code>mutex</code>	排他制御のためのロックとキュー
<code>getpass</code>	ポータブルなパスワードとユーザー ID の検索
<code>curses</code>	可搬性のある端末操作を提供する <code>curses</code> ライブラリへのインタフェース。
<code>curses.textpad</code>	<code>curses</code> ウィンドウ内での Emacs ライクな入力編集機能。
<code>curses.wrapper</code>	<code>curses</code> プログラムのための端末設定ラッパ。
<code>curses.ascii</code>	ASCII 文字に関する定数および集合帰属関数。
<code>curses.panel</code>	<code>curses</code> ウィンドウに深さの概念を追加するパネルスタック拡張。
<code>getopt</code>	ポータブルなコマンドラインオプションのパパーザ。長短の両方の形式をサポートします。
<code>optparse</code>	強力で柔軟、拡張性があり簡単に利用できるコマンドライン解析ライブラリ
<code>tempfile</code>	一時的なファイルやディレクトリを生成。
<code>errno</code>	標準の <code>errno</code> システムシンボル。
<code>glob</code>	UNIX シェル形式のパス名のパターン展開。
<code>fnmatch</code>	UNIX シェル形式のファイル名のパターンマッチ。
<code>shutil</code>	コピーを含む高レベルなファイル操作。
<code>locale</code>	国際化サービス。
<code>gettext</code>	多言語対応に関する国際化サービス。
<code>logging</code>	PEP 282 に基づく Python 用のロギングモジュール。

## 6.1 os — 雑多なオペレーティングシステムインタフェース

このモジュールでは、オペレーティングシステム依存の機能を利用する方法として、`posix` や `nt` といったオペレーティングシステム依存の組み込みモジュールを `import` するよりも可搬性の高い手段を提供しています。

このモジュールは、`mac` や `posix` のような、オペレーティングシステム依存の組み込みモジュールから関数やデータを検索して、見つかったものを取り出し (`export`) ます。Python における組み込みのオペレーティングシステム依存モジュールは、同じ機能を利用することができる限り、同じインタフェースを使います; たとえば、`os.stat(path)` は `path` についての `stat` 情報を (たまたま POSIX インタフェースに起源する) 同じ書式で返します。

特定のオペレーティングシステム固有の拡張も `os` を介して利用することができますが、これらの利用はもちろん、可搬性を脅かします！

最初の `os` の `import` 以後、`os` を介した関数の利用は、オペレーティングシステム依存組み込みモジュールにおける関数の直接利用に比べてパフォーマンス上のペナルティは全くありません。従って、`os` を利用しない理由は 存在しません！

### exception error

関数がシステム関連のエラー (引数の型違いや他のありがちなエラーではない) を返した場合この例外が発生します。これは `OSError` として知られる組み込み例外でもあります。付属する値は `errno` からとった数値のエラーコードと、エラーコードに対応する、C 関数 `perror()` により出力されるのと同じ文字列からなるペアです。背後のオペレーティングシステムで定義されているエラーコード名が収められている `errno` を参照してください。

例外がクラスの場合、この例外は二つの属性、`errno` と `strerror` を持ちます。前者の属性は C の `errno` 変数の値、後者は `strerror()` による対応するエラーメッセージの値を持ちます。(`chdir()` や `unlink()` のような) ファイルシステム上のパスを含む例外に対しては、この例外インスタンスは 3 つめの属性、`filename` を持ち、関数に渡されたファイル名となります。

### name

`import` されているオペレーティング・システム依存モジュールの名前です。現在次の名前が登録されています: `'posix'`、`'nt'`、`'dos'`、`'mac'`、`'os2'`、`'ce'`、`'java'`、`'riscos'`。

### path

`posixpath` や `macpath` のように、システムごとに対応付けられているパス名操作のためのシステム依存の標準モジュールです。すなわち、正しく `import` が行われるかぎり、`os.path.split(file)` は `posixpath.split(file)` と等価でありながらより汎用性があります。このモジュール自体が `import` 可能なモジュールでもあるので注意してください。: `os.path` として直接 `import` してもかまいません。

### 6.1.1 プロセスのパラメタ

これらの関数とデータ要素は、現在のプロセスおよびユーザに対する情報提供および操作のための機能を提供しています。

### environ

環境変数の値を表すマップ型オブジェクトです。例えば、`environ['HOME']` は (いくつかのプラットフォーム上での) あなたのホームディレクトリへのパスです。これは C の `getenv("HOME")` と等価です。

プラットフォーム上で `putenv()` がサポートされている場合、このマップ型オブジェクトは環境変数に対するクエリと同様に変更するために使うこともできます。`putenv()` はマップ型オブジェク



トが修正される時に、自動的に呼ばれることになります。

注意: FreeBSD と Mac OS X を含むいくつかのプラットフォームでは、`environ` の値を変更するとメモリリークの原因になる場合があります。システムの `putenv` に関するドキュメントを参照してください。

`putenv()` が提供されていない場合、このマッピングオブジェクトを適切なプロセス生成機能に渡して、子プロセスが修正された環境変数を利用するようにできます。

`chdir(path)`

`getcwd()`

これらの関数は、“ファイルとディレクトリ” (6.1.4 節) で説明されています。

`ctermid()`

プロセスの制御端末に対応するファイル名を返します。利用できる環境: UNIX。

`getegid()`

現在のプロセスの実行グループ id を返します。この id は現在のプロセスで実行されているファイルの ‘set id’ ビットに対応します。利用できる環境: UNIX。

`geteuid()`

現在のプロセスの実行ユーザ id を返します。利用できる環境: UNIX。

`getgid()`

現在のプロセスの実際のグループ id を返します。利用できる環境: UNIX。

`getgroups()`

現在のプロセスに関連づけられた従属グループ id のリストを返します。利用できる環境: UNIX。

`getlogin()`

現在のプロセスの制御端末にログインしているユーザ名を返します。ほとんどの場合、ユーザが誰かを知りたいときには環境変数 `LOGNAME` を、現在有効になっているユーザ名を知りたいときには `pwd.getpwuid(os.getuid())[0]` を使うほうが便利です。利用できる環境: UNIX。

`getpgrp()`

現在のプロセス・グループの id を返します。利用できる環境: UNIX。

`getpid()`

現在のプロセス id を返します。利用できる環境: UNIX, Windows。

`getppid()`

親プロセスの id を返します。利用できる環境: UNIX。

`getuid()`

現在のプロセスのユーザ id を返します。利用できる環境: UNIX。

`getenv(varname[, value])`

環境変数 `varname` が存在する場合にはその値を返し、存在しない場合には `value` を返します。`value` のデフォルト値は `None` です。利用できる環境: UNIX 互換環境、Windows。

`putenv(varname, value)`

`varname` と名づけられた環境変数の値を文字列 `value` に設定します。このような環境変数への変更は、`os.system()`、`popen()`、`fork()` および `execv()` により起動された子プロセスに影響します。利用できる環境: 主な UNIX 互換環境、Windows。

注意: FreeBSD と Mac OS X を含むいくつかのプラットフォームでは、`environ` の値を変更するとメモリリークの原因になる場合があります。システムの `putenv` に関するドキュメントを参照してください。

`putenv()` がサポートされている場合、`os.environ` の要素に対する代入を行うと自動的に

`putenv()` を呼び出します; しかし、`putenv()` の呼び出しは `os.environ` を更新しないので、実際には `os.environ` の要素に代入する方が望ましい操作です。

**setegid(*egid*)**

現在のプロセスに有効なグループ ID をセットします。利用できる環境: UNIX。

**seteuid(*eid*)**

現在のプロセスに有効なユーザ ID をセットします。利用できる環境: UNIX。

**setgid(*gid*)**

現在のプロセスにグループ id をセットします。利用できる環境: UNIX。

**setgroups(*groups*)**

現在のグループに関連付けられた従属グループ id のリストを *groups* に設定します。*groups* はシーケンス型でなくてはならず、各要素はグループを特定する整数でなくてはなりません。この操作は通常、スーパーユーザしか利用できません。Availability: UNIX. 2.2 で追加された仕様です。

**setpgrp()**

システムコール `setpgrp()` または `setpgrp(0, 0)` のどちらかのバージョンのうち、(実装されていれば) 実装されている方を呼び出します。機能については UNIX マニュアルを参照してください。利用できる環境: UNIX

**setpgid(*pid*, *pgrp*)**

システムコール `setpgid()` を呼び出して、*pid* の id をもつプロセスのプロセスグループ id を *pgrp* に設定します。利用できる環境: UNIX

**setreuid(*ruid*, *euid*)**

現在のプロセスに対して実際のユーザ id および実行ユーザ id を設定します。利用できる環境: UNIX

**setregid(*rgid*, *egid*)**

現在のプロセスに対して実際のグループ id および実行ユーザ id を設定します。利用できる環境: UNIX

**setsid()**

システムコール `setsid()` を呼び出します。機能については UNIX マニュアルを参照してください。利用できる環境: UNIX

**setuid(*uid*)**

現在のプロセスのユーザ id を設定します。利用できる環境: UNIX

**strerror(*code*)**

エラーコード *code* に対応するエラーメッセージを返します。利用できる環境: UNIX、Windows

**umask(*mask*)**

現在の数値 `umask` を設定し、以前の `umask` 値を返します。利用できる環境: UNIX、Windows

**uname()**

現在のオペレーティングシステムを特定する情報の入った 5 要素のタプルを返します。このタプルには 5 つの文字列: (*sysname*, *nodename*, *release*, *version*, *machine*) が入っています。システムによっては、ノード名を 8 文字、または先頭の要素だけに切り詰めます; ホスト名を取得する方法としては、`socket.gethostname()` を使う方がよいでしょう、あるいは `socket.gethostbyaddr(socket.gethostname())` でもかまいません。利用できる環境: UNIX 互換環境

### 6.1.2 ファイルオブジェクトの生成

以下の関数は新しいファイルオブジェクトを作成します。

**fdopen(*fd*[, *mode*[, *bufsize*]])**

ファイル記述子 *fd* に接続している、開かれたファイルオブジェクトを返します。引数 *mode* および *bufsize* は、組み込み関数 `open()` における対応する引数と同じ意味を持ちます。利用できる環境: Macintosh、UNIX、Windows

`popen(command[, mode[, bufsize]])`

*command* への、または *command* からのパイプ入出力を開きます。戻り値はパイプに接続されている開かれたファイルオブジェクトで、*mode* が 'r' (標準の設定です) または 'w' によって読み出しまたは書き込みを行うことができます。引数 *bufsize* は、組み込み関数 `open()` における対応する引数と同じ意味を持ちます。*command* の終了ステータス (`wait()` で指定された書式でコード化されています) は、`close()` メソッドの戻り値として取得することができます。例外は終了ステータスがゼロ (すなわちエラーなしで終了) の場合で、このときには `None` を返します。利用できる環境: UNIX、Windows

2.0 で変更された仕様: この関数は、Python の初期のバージョンでは、Windows 環境下で信頼できない動作をしていました。これは Windows に付属して提供されるライブラリの `_popen()` 関数を利用したことによるものです。新しいバージョンの Python では、Windows 付属のライブラリにある壊れた実装を利用しません

`tmpfile()`

更新モード ('w+b') で開かれた新しいファイルオブジェクトを返します。このファイルはディレクトリエントリ登録に関連付けられておらず、このファイルに対するファイル記述子がなくなると自動的に削除されます。利用できる環境: UNIX、Windows

以下の `popen()` の変種はどれも、*bufsize* が指定されている場合には I/O パイプのバッファサイズを表します。*mode* を指定する場合には、文字列 'b' または 't' でなければなりません; これは、Windows でファイルをバイナリモードで開くかテキストモードで開くかを決めるために必要です。*mode* の標準の設定値は 't' です。

以下のメソッドは子プロセスからリターンコードを取得できるようにはしていません。入出力ストリームを制御し、かつ終了コードの取得も行える唯一の方法は、`popen2` モジュールの `Popen3` と `Popen4` クラスを利用する事です。これらは UNIX 上でのみ利用可能です。

これらの関数の利用に関係して起きうるデッドロック状態についての議論は、“フロー制御問題” (section 6.8.2) を参照してください。

`popen2(cmd[, mode[, bufsize]])`

*cmd* を子プロセスとして実行します。ファイル・オブジェクト (*child\_stdin*, *child\_stdout*) を返します。利用できる環境: UNIX、Windows 2.0 で追加された仕様です。

`popen3(cmd[, mode[, bufsize]])`

*cmd* を子プロセスとして実行します。ファイルオブジェクト (*child\_stdin*, *child\_stdout*, *child\_stderr*) を返します。利用できる環境: UNIX、Windows 2.0 で追加された仕様です。

`popen4(cmd[, mode[, bufsize]])`

*cmd* を子プロセスとして実行します。ファイルオブジェクト (*child\_stdin*, *child\_stdout\_and\_stderr*) を返します。利用できる環境: UNIX、Windows 2.0 で追加された仕様です。

上記の機能は `popen2` モジュール内の同じ名前の関数を使っても実現できますが、これらの関数の戻り値は異なる順序を持っています。

### 6.1.3 ファイル記述子の操作

これらの関数は、ファイル記述子を使って参照されている I/O ストリームを操作します。

`close(fd)`

ファイルディスクリプタ *fd* を閉じます。利用できる環境: Macintosh、UNIX、Windows

注:この関数は低レベルの I/O のためのもので、`open()` や `pipe()` が返すファイル記述子に対して適用しなければなりません。組み込み関数 `open()` や `popen()`、`fdopen()` の返す“ファイルオブジェクト”を閉じるには、オブジェクトの `close()` メソッドを使ってください。

**dup**(*fd*)

ファイル記述子 *fd* の複製を返します。利用できる環境: Macintosh、UNIX、Windows.

**dup2**(*fd*, *fd2*)

ファイル記述子を *fd* から *fd2* に複製し、必要なら後者の記述子を前もって閉じておきます。利用できる環境: UNIX、Windows

**fdatasync**(*fd*)

ファイル記述子 *fd* を持つファイルのディスクへの書き込みを強制します。メタデータの更新は強制しません。利用できる環境: UNIX

**fpathconf**(*fd*, *name*)

開いているファイルに関連したシステム設定情報 (system configuration information) を返します。*name* には取得したい設定名を指定します; これは定義済みのシステム固有値名の文字列で、多くの標準 (POSIX.1、UNIX 95、UNIX 98 その他) で定義されています。プラットフォームによっては別の名前も定義しています。ホストオペレーティングシステムの関知する名前は `pathconf_names` 辞書で与えられています。このマップオブジェクトに入っていない設定変数については、*name* に整数を渡してもかまいません。利用できる環境: UNIX

もし *name* が文字列でかつ不明である場合、`ValueError` を送出します。*name* の指定値がホストシステムでサポートされておらず、`pathconf_names` にも入っていない場合、`errno.EINVAL` をエラー番号として `OSError` を送出します。

**fstat**(*fd*)

`stat()` のようにファイル記述子 *fd* の状態を返します。利用できる環境: UNIX、Windows

**fstatvfs**(*fd*)

`statvfs()` のように、ファイル記述子 *fd* に関連づけられたファイルが入っているファイルシステムに関する情報を返します。利用できる環境: UNIX

**fsync**(*fd*)

ファイル記述子 *fd* を持つファイルのディスクへの書き込みを強制します。UNIX では、ネイティブの `fsync()` 関数を、Windows では `MS_commit()` 関数を呼び出します。

Python のファイルオブジェクト *f* を使う場合、*f* の内部バッファを確実にディスクに書き込むために、まず *f.flush()* を実行し、それから `os.fsync(f.fileno())` してください。利用できる環境: UNIX、2.2.3 以降では Windows も。

**ftruncate**(*fd*, *length*)

ファイル記述子 *fd* に対応するファイルを、サイズが最大で *length* バイトになるように切り詰めます。利用できる環境: UNIX。

**isatty**(*fd*)

ファイル記述子 *fd* が開いていて、tty(のような) 装置に接続されている場合、1 を返します。そうでない場合は 0 を返します。利用できる環境: UNIX

**lseek**(*fd*, *pos*, *how*)

ファイル記述子 *fd* の現在の位置を *pos* に設定します。*pos* の意味は *how* で修飾されます: ファイルの先頭からの相対には 0 を設定します; 現在の位置からの相対には 1 を設定します; ファイルの末尾からの相対には 2 を設定します。利用できる環境: Macintosh、UNIX、Windows。

**open**(*file*, *flags*[, *mode*])

ファイル *file* を開き、*flag* に従って様々なフラグを設定し、可能なら *mode* に従ってファイルモードを

設定します。*mode* の標準の設定値は 0777 (8 進表現) で、先に現在の *umask* を使ってマスクを掛けます。新たに開かれたファイルのファイル記述子を返します。利用できる環境: Macintosh、UNIX、Windows。フラグとファイルモードの値についての詳細は C ランタイムのドキュメントを参照してください; (*O\_RDONLY* や *O\_WRONLY* のような) フラグ定数はこのモジュールでも定義されています (以下を参照してください)。

この関数は低レベルの I/O のためのものです。通常の利用では、*read()* や *write()* (やその他多くの) メソッドを持つ “ファイルオブジェクト” を返す、組み込み関数 *open()* を使ってください。 , which returns a “file

#### ***openpty()***

新しい擬似端末のペアを開きます。ファイル記述子のペア (*master*, *slave*) を返し、それぞれ *pty* および *tty* を表します。(少しだけ) より可搬性のあるアプローチとしては、*pty* モジュールを使ってください。利用できる環境: いくつかの UNIX 風システム。

#### ***pipe()***

パイプを作成します。ファイル記述子のペア (*r*, *w*) を返し、それぞれ読み出し、書き込み用に使うことができます。利用できる環境: UNIX、Windows。

#### ***read(fd, n)***

ファイル記述子 *fd* から最大で *n* バイト読み出します。読み出されたバイト列の入った文字列を返します。*fd* が参照しているファイルの終端に達した場合、空の文字列が返されます。利用できる環境: Macintosh、UNIX、Windows。

注: この関数は低レベルの I/O のためのもので、*open()* や *pipe()* が返すファイル記述子に対して適用しなければなりません。組み込み関数 *open()* や *popen()*、*fdopen()* の返す “ファイルオブジェクト”、あるいは *sys.stdin* から読み出すには、オブジェクトの *read()* メソッドを使ってください。

#### ***tcgetpgrp(fd)***

*fd* (*open()* が返す開かれたファイル記述子) で与えられる端末に関連付けられたプロセスグループを返します。利用できる環境: UNIX。

#### ***tcsetpgrp(fd, pg)***

*fd* (*open()* が返す開かれたファイル記述子) で与えられる端末に関連付けられたプロセスグループを *pg* に設定します。利用できる環境: UNIX。

#### ***ttyname(fd)***

ファイル記述子 *fd* に関連付けられている端末デバイスを特定する文字列を返します。*fd* が端末に関連付けられていない場合、例外が送出されます。利用できる環境: UNIX。

#### ***write(fd, str)***

ファイル記述子 *fd* に文字列 *str* を書き込みます。実際に書き込まれたバイト数を返します。利用できる環境: Macintosh、UNIX、Windows。

注: この関数は低レベルの I/O のためのもので、*open()* や *pipe()* が返すファイル記述子に対して適用しなければなりません。組み込み関数 *open()* や *popen()*、*fdopen()* の返す “ファイルオブジェクト”、あるいは *sys.stdout*、*sys.stderr* に書き込むには、オブジェクトの *write()* メソッドを使ってください。

以下のデータ要素は *open()* 関数の *flags* 引数を構築するために利用することができます。

*O\_RDONLY*

*O\_WRONLY*

*O\_RDWR*

*O\_NDELAY*



**O\_NONBLOCK**

**O\_APPEND**

**O\_DSYNC**

**O\_RSYNC**

**O\_SYNC**

**O\_NOCTTY**

**O\_CREAT**

**O\_EXCL**

**O\_TRUNC**

`open()` 関数の *flag* 引数のためのオプションフラグです。これらの値はビット単位 OR を取ることができます。利用できる環境:Macintosh、UNIX、Windows。

**O\_BINARY**

`open()` 関数の *flag* 引数のためのオプションフラグです。この値は上に列挙したフラグとビット単位 OR を取ることができます。利用できる環境:Macintosh、Windows。

## 6.1.4 ファイルとディレクトリ

**access(*path*, *mode*)**

実 uid/gid を使って *path* に対するアクセスが可能か調べます。ほとんどのオペレーティングシステムは実行 uid/gid を使うため、このルーチンは suid/sgid 環境において、プログラムを起動したユーザが *path* に対するアクセス権をもっているかを調べるために使われます。*path* が存在するかどうかを調べるには *mode* を F\_OK にします。ファイル操作許可 (permission) を調べるために R\_OK、W\_OK、X\_OK から一つまたはそれ以上のフラグと OR をとることもできます。アクセスが許可されている場合 1 を、そうでない場合 0 を返します。詳細は `access(2)` のマニュアルページを参照してください。利用できる環境:Macintosh、Windows。

**F\_OK**

`access()` の *mode* に渡すための値で、*path* が存在するかどうかを調べます。

**R\_OK**

`access()` の *mode* に渡すための値で、*path* が読み出し可能かどうかを調べます。

**W\_OK**

`access()` の *mode* に渡すための値で、*path* が書き込み可能かどうかを調べます。

**X\_OK**

`access()` の *mode* に渡すための値で、*path* が実行可能かどうかを調べます。

**chdir(*path*)**

現在の作業ディレクトリ (current working directory) を *path* に設定します。利用できる環境:Macintosh、UNIX、Windows。

**getcwd()**

現在の作業ディレクトリを表現する文字列を返します。利用できる環境: Macintosh、UNIX、Windows。

**chroot(*path*)**

現在のプロセスに対してルートディレクトリを *path* に変更します。利用できる環境: UNIX。2.2 で追加された仕様です。

**chmod(*path*, *mode*)**

*path* のモードを数値 *mode* に変更します。*mode* は、(stat モジュールで定義されている) 以下の値のいずれかを取り得ます:

•S\_ISUID



- S\_ISGID
- S\_ENFMT
- S\_ISVTX
- S\_IREAD
- S\_IWRITE
- S\_IEXEC
- S\_IRWXU
- S\_IRUSR
- S\_IWUSR
- S\_IXUSR
- S\_IRWXG
- S\_IRGRP
- S\_IWGRP
- S\_IXGRP
- S\_IRWXO
- S\_IROTH
- S\_IWOTH
- S\_IXOTH

利用できる環境: UNIX、Windows。

**chown**(*path*, *uid*, *gid*)

*path* の所有者 (owner) *id* とグループ *id* を、数値 *uid* および *gid* に変更します。利用できる環境: UNIX。

**link**(*src*, *dst*)

*src* を指しているハードリンク *dst* を作成します。利用できる環境: UNIX。

**listdir**(*path*)

ディレクトリ内のエントリ名が入ったリストを返します。リスト内の順番は不定です。特殊エントリ `‘.’` および `‘..’` は、それらがディレクトリに入っているリストには含められません。2.3 で変更された仕様: Windows NT/2k/XP と Unix では、*path* が Unicode オブジェクトの場合、Unicode オブジェクトのリストが返されます。利用できる環境: Macintosh、UNIX、Windows。

**lstat**(*path*)

`stat()` に似ていますが、シンボリックリンクをたどりません。利用できる環境: UNIX。

**mkfifo**(*path*[, *mode*])

数値で指定されたモード *mode* を持つ FIFO (名前付きパイプ) を *path* に作成します。*mode* の標準の値は 0666 (8 進) です。現在の `umask` 値が前もって *mode* からマスクされます。利用できる環境: UNIX。

FIFO は通常のファイルのようにアクセスできるパイプです。FIFO は (例えば `os.unlink()` を使って) 削除されるまで存在しつづけます。一般的に、FIFO は“クライアント”と“サーバ”形式のプロセス間でランデブーを行うために使われます: このとき、サーバは FIFO を読み出し用に関き、クライアントは書き込み用に関きます。`mkfifo()` は FIFO を開かない — 単にランデブーポイントを作成するだけ — なので注意してください。

**mknod**(*path*[, *mode*=0600, *device*])

*filename* という名前で、ファイルシステム・ノード (ファイル、デバイス特殊ファイル、または、名前付きパイプ) を作ります。*mode* は、作ろうとするノードの使用権限とタイプを、`S_IFREG`、`S_IFCHR`、

S\_IFBLK、S\_IFIFO (これらの定数は `stat` で使用可能) のいずれかと (ビット OR で) 組み合わせて指定します。S\_IFCHR と S\_IFBLK を指定すると、*device* は新しく作られたデバイス特殊ファイル (おそらく `os.makedev()` を使って) 定義し、指定しなかった場合には無視します。2.3 で追加された仕様です。

**major**(*device*)

生のデバイス番号から、デバイスのメジャー番号を取り出します。2.3 で追加された仕様です。

**minor**(*device*)

生のデバイス番号から、デバイスのマイナー番号を取り出します。2.3 で追加された仕様です。

**makedev**(*major*, *minor*)

*major* と *minor* から、新しく生のデバイス番号を作ります。2.3 で追加された仕様です。

**mkdir**(*path*[, *mode*])

数値で指定されたモード *mode* をもつディレクトリ *path* を作成します。*mode* の標準の値は 0777 (8 進) です。システムによっては、*mode* は無視されます。利用の際には、現在の `umask` 値が前もってマスクされます。利用できる環境: Macintosh、UNIX、Windows。

**makedirs**(*path*[, *mode*])

再帰的なディレクトリ作成関数です。mkdir() に似ていますが、末端 (leaf) となるディレクトリを作成するために必要な中間の全てのディレクトリを作成します。末端ディレクトリがすでに存在する場合や、作成ができなかった場合には `error` 例外を送出します。*mode* の標準の値は 0777 (8 進) です。(Windows システムにのみ関係することですが、Universal Naming Convention パスは、`'\\host\path'` という書式のパスです) 1.5.2 で追加された仕様です。

**pathconf**(*path*, *name*)

指定されたファイルに関するシステム設定情報を返します。*varname* には取得したい設定名を指定します; これは定義済みのシステム固有値名の文字列で、多くの標準 (POSIX.1、UNIX 95、UNIX 98 その他) で定義されています。プラットフォームによっては別の名前も定義しています。ホストオペレーティングシステムの関知する名前は `pathconf_names` 辞書で与えられています。このマップ型オブジェクトに入っていない設定変数については、*name* に整数を渡してもかまいません。利用できる環境: UNIX

もし *name* が文字列でかつ不明である場合、`ValueError` を送じます。*name* の指定値がホストシステムでサポートされておらず、`pathconf_names` にも入っていない場合、`errno.EINVAL` をエラー番号として `OSError` を送じます。

**pathconf\_names**

`pathconf()` および `fpathconf()` が受理するシステム設定名を、ホストオペレーティングシステムで定義されている整数値に対応付けている辞書です。この辞書はシステムでどの設定名が定義されているかを決定するために利用できます。利用できる環境: UNIX。

**readlink**(*path*)

シンボリックリンクが指しているパスを表す文字列を返します。返される値は絶対パスにも、相対パスにもなり得ます; 相対パスの場合、`os.path.join(os.path.dirname(path), result)` を使って絶対パスに変換することができます。利用できる環境: UNIX。

**remove**(*path*)

ファイル *path* を削除します。*path* がディレクトリの場合、`OSError` が送じます; ディレクトリの削除については `rmdir()` を参照してください。この関数は下で述べられている `unlink()` 関数と同一です。Windows では、使用中のファイルを削除しようと試みると例外を送じます; UNIX では、ディレクトリエントリは削除されますが、記憶装置上にアロケーションされたファイル領域は元のファイルが使われなくなるまで残されます。利用できる環境: Macintosh、UNIX、Windows。

**removedirs**(*path*)

再帰的なディレクトリ削除関数です。rmdir()と同じように動作しますが、末端ディレクトリがうまく削除できるかぎり、パスを構成する要素の右端となるディレクトリを刈り込んでゆき、指定したパス全体が削除されるかエラーが送出されるまで続けます (このエラーは通常、指定したディレクトリの親ディレクトリが空でないことを意味するだけなので無視されます)。末端のディレクトリがうまく削除できない場合には `error` を送出します。1.5.2 で追加された仕様です。

**rename**(*src*, *dst*)

ファイルまたはディレクトリ *src* を *dst* に名前変更します。*dst* がディレクトリの場合、`OSError` が送出されます。UNIX では、*dst* が存在し、かつファイルの場合、ユーザの権限があるかぎり暗黙のうちに元のファイルが削除されます。この操作はいくつかの UNIX 系において、*src* と *dst* が異なるファイルシステム上にあると失敗することがあります。ファイル名の変更が成功する場合、この操作は原子的 (atomic) 操作となります (これは POSIX 要求仕様です) Windows では、*dst* が既に存在する場合には、たとえファイルの場合でも `OSError` が送出されます; これは *dst* が既に存在するファイル名の場合、名前変更の原子的操作を実装する手段がないからです。利用できる環境: Macintosh、UNIX、Windows。

**renames**(*old*, *new*)

再帰的にディレクトリやファイル名を変更する関数です。rename() のように動作しますが、新たなパス名を持つファイルを配置するために必要な途中のディレクトリ構造をまず作成しようと試みます。名前変更の後、元のファイル名のパス要素は `removedirs()` を使って右側から順に枝刈りされてゆきます。注意: この関数はコピー元の末端のディレクトリまたはファイルを削除する権限がない場合には失敗します。1.5.2 で追加された仕様です。

**rmdir**(*path*)

ディレクトリ *path* を削除します。利用できる環境: Macintosh、UNIX、Windows。

**stat**(*path*)

与えられた *path* に対して `stat()` システムコールを実行します。戻り値はオブジェクトで、その属性が `stat` 構造体の以下に挙げる各メンバ: `st_mode` (保護モードビット)、`st_ino` (i ノード番号)、`st_dev` (デバイス)、`st_nlink` (ハードリンク数)、`st_uid` (所有者のユーザ ID)、`st_gid` (所有者のグループ ID)、`st_size` (ファイルのバイトサイズ)、`st_atime` (最終アクセス時刻)、`st_mtime` (最終更新時刻)、`st_ctime` (最終更新またはメタデータ変更時刻)、となっています。

2.3 で変更された仕様: もし `stat_float_times` が真を返す場合、時間値は浮動小数点で秒を計ります。ファイルシステムがサポートしていれば、秒の小数点以下の桁も含めて返されます。Mac OS では、時間は常に浮動小数点です。詳細な説明は `stat_float_times` を参照してください

(Linux のような) Unix システムでは、以下の属性: `st_blocks` (ファイル用にアロケーションされているブロック数)、`st_blksize` (ファイルシステムのブロックサイズ)、`st_rdev` (i ノードデバイスの場合、デバイスの形式)、も利用可能なときがあります。

Mac OS システムでは、以下の属性: `st_rsize`、`st_creator`、`st_type`、も利用可能なときがあります。

RISCOS システムでは、以下の属性: `st_fstype` (file type)、`st_attrs` (attributes)、`st_obtype` (object type)、も利用可能なときがあります。

後方互換性のために、`stat()` の戻り値は少なくとも 10 個の整数からなるタプルとしてアクセスすることができます。このタプルはもっとも重要な (かつ可搬性のある) `stat` 構造体のメンバを与えており、以下の順番、`st_mode`、`st_ino`、`st_dev`、`st_nlink`、`st_uid`、`st_gid`、`st_size`、`st_atime`、`st_mtime`、`st_ctime`、に並んでいます。

実装によっては、この後ろにさらに値が付け加えられていることもあります。Mac OS では、時刻の値は Mac OS の他の時刻表現値と同じように浮動小数点数なので注意してください。標準モジュール `stat` では、`stat` 構造体から情報を引き出す上で便利な関数や定数を定義しています。(Windows で

は、いくつかのデータ要素はダミーの値が埋められています。) 利用できる環境: Macintosh、UNIX、Windows。

2.2 で変更された仕様: 返されたオブジェクトの属性としてのアクセス機能を追加しました

`stat_float_times([newvalue])`

`stat_result` がタイムスタンプに浮動小数点オブジェクトを使うかどうかを決定します。 `newvalue` が真の場合、以後の `stat()` 呼び出しは浮動小数点を返し、偽の場合には整数を返します。 `newvalue` が省略された場合、現在の設定どおりの戻り値になります。

古いバージョンの Python と互換性を保つため、`stat_result` にタプルとしてアクセスすると、常に整数が返されます。また、Python 2.2 との互換性のため、タイムスタンプにフィールド名を指定してアクセスすると、整数で返されます。タイムスタンプの秒を小数点以下の精度で求めたいアプリケーションは、タイムスタンプを浮動小数点型にするために、この関数を使うことができます。実際に、小数点以下の桁に 0 以外の数値が得られるかどうかは、システムに依存します。

将来リリースされる Python は、この設定のデフォルト値を変更するでしょう。浮動小数点型のタイムスタンプを扱えないアプリケーションは、この関数を使って、その機能を停止させることができます。

この設定の変更は、プログラムの起動時に、`__main__` モジュールの中でのみ行うことを推奨します。ライブラリは決して、この設定を変更するべきではありません。浮動小数点型のタイムスタンプを処理すると、不正確な動作をするようなライブラリを使う場合、ライブラリが修正されるまで、浮動小数点型を返す機能を停止させておくべきです。

`statvfs(path)`

与えられた `path` に対して `statvfs()` システムコールを実行します。戻り値はオブジェクトで、その属性は与えられたパスが収められているファイルシステムについて記述したものです。かく属性は `statvfs` 構造体のメンバ: `f_frsize`、`f_blocks`、`f_bfree`、`f_bavail`、`f_files`、`f_ffree`、`f_favail`、`f_flag`、`f_namemax`、に対応します。利用できる環境: UNIX。

後方互換性のために、戻り値は上の順にそれぞれ対応する属性値が並んだタプルとしてアクセスすることもできます。標準モジュール `statvfs` では、配列としてアクセスする場合に、`statvfs` 構造体から情報を引き出す上便利な関数や定数を定義しています; これは属性として各フィールドにアクセスできないバージョンの Python で動作する必要のあるコードを書く際に便利です。2.2 で変更された仕様: 返されたオブジェクトの属性としてのアクセス機能を追加しました

`symlink(src, dst)`

`src` を指しているシンボリックリンクを `dst` に作成します。利用できる環境: UNIX。

`tempnam(dir[, prefix])`

一時ファイル (temporary file) を生成する上でファイル名として相応しい一意なパス名を返します。この値は一時的なディレクトリエントリを表す絶対パスで、`dir` ディレクトリの下か、`dir` が省略されたり `None` の場合には一時ファイルを置くための共通のディレクトリの下になります。 `prefix` が与えられており、かつ `None` でない場合、ファイル名の先頭につけられる短い接頭辞になります。アプリケーションは `tempnam()` が返したパス名を使って正しくファイルを生成し、生成したファイルを管理する責任があります; 一時ファイルの自動消去機能は提供されていません。警告: `tempnam()` を使うと、`symlink` 攻撃に対して脆弱になります; 代りに `tmpfile()` を使うよう検討してください。利用できる環境: UNIX、Windows。

`tmpnam()`

一時ファイル (temporary file) を生成する上でファイル名として相応しい一意なパス名を返します。この値は一時ファイルを置くための共通のディレクトリ下の一時的なディレクトリエントリを表す絶対パスです。アプリケーションは `tmpnam()` が返したパス名を使って正しくファイルを生成し、生成したファイルを管理する責任があります; 一時ファイルの自動消去機能は提供されていません。



警告: `tmpnam()` を使うと、symlink 攻撃に対して脆弱になります; 代りに `tmpfile()` を使うよう検討してください。利用できる環境: UNIX、Windows。この関数はおそらく Windows では使うべきではないでしょう; Microsoft の `tmpnam()` 実装では、常に現在のドライブのルートディレクトリ下のファイル名を生成しますが、これは一般的にはテンポラリファイルを置く場所としてはひどい場所です (アクセス権限によっては、この名前をつかってファイルを開くことすらできないかもしれません)。

#### **TMP\_MAX**

`tmpnam()` がテンポラリ名を再利用し始めるまでに生成できる一意な名前の最大数です。

#### **unlink(path)**

ファイル *path* を削除します。 `remove()` と同じです; `unlink()` の名前は伝統的な UNIX の関数名です。利用できる環境: Macintosh、UNIX、Windows。

#### **utime(path, times)**

*path* で指定されたファイルに最終アクセス時刻および最終修正時刻を設定します。 *times* が `None` の場合、ファイルの最終アクセス時刻および最終更新時刻は現在の時刻になります。そうでない場合、*times* は 2 要素のタプルで、(*atime*, *mtime*) の形式をとらなくてはなりません。これらはそれぞれアクセス時刻および修正時刻を設定するために使われます。2.0 で変更された仕様: *times* として `None` をサポートするようにしました。利用できる環境: Macintosh、UNIX、Windows。

#### **walk(top[, topdown=True[, onerror=None]])**

`walk()` は、ディレクトリツリー以下のファイル名を、ツリーをトップダウンとボトムアップの両方向に歩行することで生成します。ディレクトリ *top* を根に持つディレクトリツリーに含まれる、各ディレクトリ (*top* 自身を含む) から、タプル (*dirpath*, *dirnames*, *filenames*) を生成します。

*dirpath* は文字列で、ディレクトリへのパスです。 *dirnames* は *dirpath* 内のサブディレクトリ名のリスト ('.' と '..' は除く) です。 *filenames* は *dirpath* 内の非ディレクトリ・ファイル名のリストです。このリスト内の名前には、ファイル名までのパスが含まれないことに、注意してください。 *dirpath* 内のファイルやディレクトリへの (*top* からたどった) フルパスを得るには、`os.path.join(dirpath, name)` してください。

オプション引数 *topdown* が真であるか、指定されなかった場合、各ディレクトリからタプルを生成した後で、サブディレクトリからタプルを生成します。(ディレクトリはトップダウンで生成)。 *topdown* が偽の場合、ディレクトリに対応するタプルは、そのディレクトリ以下の全てのサブディレクトリに対応するタブルの後で (ボトムアップで) 生成されます

*topdown* が真のとき、呼び出し側は *dirnames* リストを、インプレースで (たとえば、`del` やスライスを使った代入で) 変更でき、`walk()` は *dirnames* に残っているサブディレクトリ内のみを再帰します。これにより、検索を省略したり、特定の訪問順序を強制したり、呼び出し側が `walk()` を再開する前に、呼び出し側が作った、または名前を変更したディレクトリを、`walk()` に知らせたりすることができます。 *topdown* が偽のときに *dirnames* を変更しても効果はありません。ボトムアップモードでは *dirnames* 自身が生成される前に *dirnames* 内のディレクトリの情報が生成されるからです。

デフォルトでは、`os.listdir()` 呼び出しから送出されたエラーは無視されます。オプションの引数 *onerror* を指定するなら、この値は関数でなければなりません; この関数は単一の引数として、`os.error` インスタンスを伴って呼び出されます。この関数ではエラーを報告して歩行を続けたり、例外を送出して歩行を中断したりできます。ファイル名は例外オブジェクトの *filename* 属性として取得することに注意してください。

注意: 相対パスを渡した場合、`walk()` の回復の間でカレント作業ディレクトリを変更しないでください。 `walk()` はカレントディレクトリを変更しませんし、呼び出し側もカレントディレクトリを変更しないと仮定しています。

注意: シンボリックリンクをサポートするシステムでは、サブディレクトリへのリンクが *dirnames* リ

ストに含まれますが、`walk()` はそのリンクをたどりません (シンボリックリンクをたどると、無限ループに陥りやすくなります)。リンクされたディレクトリをたどるには、`os.path.islink(path)` でリンク先ディレクトリを確認し、各ディレクトリに対して `walk(path)` を実行するとよいでしょう。

以下の例では、最初のディレクトリ以下にある各ディレクトリに含まれる、非ディレクトリファイルのバイト数を表示します。ただし、CVS サブディレクトリより下を見に行きません。

```
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/email'):
    print root, "consumes",
    print sum([getsize(join(root, name)) for name in files]),
    print "bytes in", len(files), "non-directory files"
    if 'CVS' in dirs:
        dirs.remove('CVS') # don't visit CVS directories
```

次の例では、ツリーをボトムアップで歩行することが不可欠になります; `rmdir()` はディレクトリが空になる前に削除させないからです:

```
import os
from os.path import join
# Delete everything reachable from the directory named in 'top'.
# CAUTION: This is dangerous! For example, if top == '/', it
# could delete all your disk files.
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(join(root, name))
    for name in dirs:
        os.rmdir(join(root, name))
```

2.3 で追加された仕様です。

### 6.1.5 プロセス管理

プロセスを生成したり管理するために、以下の関数を利用することができます。

様々な `exec*()` 関数が、プロセス内にロードされた新たなプログラムに与えるための引数からなるリストをとります。どの場合でも、新たなプログラムに渡されるリストの最初の引数は、ユーザがコマンドラインで入力する引数ではなく、プログラム自身の名前になります。C プログラムにとっては、これはプログラムの `main()` に渡される `argv[0]` になります。例えば、`'os.execv('/bin/echo', ['foo', 'bar'])` は、標準出力に `'bar'` を出力します; `'foo'` は無視されたかのように見えることでしょう。

**abort()**

SIGABRT シグナルを現在のプロセスに対して生成します。UNIX では、標準設定の動作はコアダンプの生成です; Windows では、プロセスは即座に終了コード 3 を返します。 `signal.signal()` を使って SIGABRT に対するシグナルハンドラを設定しているプログラムは異なる挙動を示すので注意してください。

利用できる環境: UNIX、Windows。

```
execl(path, arg0, arg1, ...)
execle(path, arg0, arg1, ..., env)
execlp(file, arg0, arg1, ...)
execclpe(file, arg0, arg1, ..., env)
execv(path, args)
execve(path, args, env)
```



`execvp(file, args)`

`execvpe(file, args, env)`

これらの関数はすべて、現在のプロセスを置き換える形で新たなプログラムを実行します; 現在のプロセスは戻り値を返しません。UNIX では、新たに実行される実行コードは現在のプロセス内にロードされ、呼び出し側と同じプロセス ID を持つことになります。エラーは `OSError` 例外として報告されます。

‘l’ および ‘v’ のついた `exec*()` 関数は、コマンドライン引数をどのように渡すかが異なります。‘l’ 型は、コードを書くときにパラメタ数が決まっている場合に、おそらくもっとも簡単に利用できます。個々のパラメタは単に `execl*()` 関数の追加パラメタとなります。‘v’ 型は、パラメタの数が可変の時に便利で、リストかタプルの引数が `args` パラメタとして渡されます。どちらの場合も、子プロセスに渡す引数は動作させようとしているコマンドの名前から始まらなくてはなりません。

末尾近くに ‘p’ をもつ型 (`execlp()`、`execlpe()`、`execvp()`、および `execvpe()`) は、プログラム `file` を探すために環境変数 `PATH` を利用します。環境変数が (次の段で述べる `exec*e()` 型関数で) 置き換えられる場合、環境変数は `PATH` を決定する上の情報源として使われます。その他の型、`execl()`、`execle()`、`execv()`、および `execve()` では、実行コードを探すために `PATH` を使いません。`path` には適切に設定された絶対パスまたは相対パスが入っていないなくてはなりません。

`execle()`、`execlpe()`、`execve()`、および `execvpe()` (全て末尾に ‘e’ がついていることに注意してください) では、`env` パラメタは新たなプロセスで利用される環境変数を定義するためのマップ型でなくてはなりません; `execl()`、`execlp()`、`execv()`、および `execvp()` では、全て新たなプロセスは現在のプロセスの環境を引き継ぎます。利用できる環境: UNIX、Windows。

`_exit(n)`

終了ステータス `n` でシステムを終了します。このときクリーンアップハンドラの呼び出しや、標準入出力バッファのフラッシュなどはいきません。利用できる環境: UNIX、Windows。

注意: システムを終了する標準的な方法は `sys.exit(n)` です。`_exit()` は通常、`fork()` された後の子プロセスでのみ使われます。

以下の終了コードは必須ではありませんが `_exit()` と共に使うことができます。一般に、メールサーバの外部コマンド配送プログラムのような、Python で書かれたシステムプログラムに使います。

`EX_OK`

エラーが起きなかったことを表す終了コード。利用できる環境: UNIX。2.3 で追加された仕様です。

`EX_USAGE`

誤った個数の引数が渡されたときなど、コマンドが間違っていて使われたことを表す終了コード。利用できる環境: UNIX。2.3 で追加された仕様です。

`EX_DATAERR`

入力データが間違っていたことを表す終了コード。利用できる環境: UNIX。2.3 で追加された仕様です。

`EX_NOINPUT`

入力ファイルが存在しなかった、または、読み込み不可だったことを表す終了コード。利用できる環境: UNIX。2.3 で追加された仕様です。

`EX_NOUSER`

指定されたユーザが存在しなかったことを表す終了コード。利用できる環境: UNIX。2.3 で追加された仕様です。

`EX_NOHOST`

指定されたホストが存在しなかったことを表す終了コード。利用できる環境: UNIX。2.3 で追加された仕様です。

#### **EX\_UNAVAILABLE**

要求されたサービスが利用できないことを表す終了コード。利用できる環境: UNIX。2.3 で追加された仕様です。

#### **EX\_SOFTWARE**

内部ソフトウェアエラーが検出されたことを表す終了コード。利用できる環境: UNIX。2.3 で追加された仕様です。

#### **EX\_OSERR**

fork できない、pipe の作成ができないなど、オペレーティング・システム・エラーが検出されたことを表す終了コード。利用できる環境: UNIX。2.3 で追加された仕様です。

#### **EX\_OSFILE**

システムファイルが存在しなかった、開けなかった、あるいはその他のエラーが起きたことを表す終了コード。利用できる環境: UNIX。2.3 で追加された仕様です。

#### **EX\_CANTCREAT**

ユーザには作成できない出力ファイルを指定したことを表す終了コード。利用できる環境: UNIX。2.3 で追加された仕様です。

#### **EX\_IOERR**

ファイルの I/O を行っている途中にエラーが発生したときの終了コード。利用できる環境: UNIX。2.3 で追加された仕様です。

#### **EX\_TEMPFAIL**

一時的な失敗が発生したことを表す終了コード。これは、再試行可能な操作の途中に、ネットワークに接続できないというような、実際にはエラーではないかも知れないことを意味します。利用できる環境: UNIX。2.3 で追加された仕様です。

#### **EX\_PROTOCOL**

プロトコル交換が不正、不適切、または理解不能なことを表す終了コード。利用できる環境: UNIX。2.3 で追加された仕様です。

#### **EX\_NOPERM**

操作を行うために十分な許可がなかった（ファイルシステムの問題を除く）ことを表す終了コード。利用できる環境: UNIX。2.3 で追加された仕様です。

#### **EX\_CONFIG**

設定エラーが起こったことを表す終了コード。利用できる環境: UNIX。2.3 で追加された仕様です。

#### **EX\_NOTFOUND**

“an entry was not found” のようなことを表す終了コード。利用できる環境: UNIX。2.3 で追加された仕様です。

#### **fork()**

子プロセスを fork します。子プロセスでは 0 が返り、親プロセスでは子プロセスの id が返ります。利用できる環境: UNIX。

#### **forkpty()**

子プロセスを fork します。このとき新しい擬似端末 (pseudo-terminal) を子プロセスの制御端末として使います。親プロセスでは (*pid*, *fd*) からなるペアが返り、*fd* は擬似端末のマスタ側 (master end) のファイル記述子となります。可搬性のあるアプローチを取るためには、pty モジュールを利用してください。利用できる環境: いくつかの UNIX 系。

#### **kill(*pid*, *sig*)**

プロセス *pid* をシグナル *sig* で kill します。ホストプラットフォームで利用可能なシグナルを特定する定数は signal モジュールで定義されています。利用できる環境: UNIX。

**killpg**(*pgid*, *sig*)

プロセスグループ *pgid* をシグナル *sig* 付きで kill する。利用できる環境: UNIX。2.3 で追加された仕様です。

**nice**(*increment*)

プロセスの “nice 値” に *increment* を加えます。新たな nice 値を返します。利用できる環境: UNIX。

**plock**(*op*)

プログラムのセグメント (program segment) をメモリ内でロックします。*op* (<sys/lock.h> で定義されています) にはどのセグメントをロックするかを指定します。利用できる環境: UNIX。

**popen**(...)

**popen2**(...)

**popen3**(...)

**popen4**(...)

子プロセスを起動し、子プロセスとの通信のために開かれたパイプを返します。これらの関数は [6.1.2](#) 節で記述されています。

**spawnl**(*mode*, *path*, ...)

**spawnle**(*mode*, *path*, ..., *env*)

**spawnlp**(*mode*, *file*, ...)

**spawnlpe**(*mode*, *file*, ..., *env*)

**spawnv**(*mode*, *path*, *args*)

**spawnve**(*mode*, *path*, *args*, *env*)

**spawnvp**(*mode*, *file*, *args*)

**spawnvpe**(*mode*, *file*, *args*, *env*)

新たなプロセス内でプログラム *path* を実行します。*mode* が P\_NOWAIT の場合、この関数は新たなプロセスのプロセス ID となります。; *mode* が P\_WAIT の場合、子プロセスが正常に終了するとその終了コードが返ります。そうでない場合にはプロセスを kill したシグナル *signal* に対して *-signal* が返ります。Windows では、プロセス ID は実際にはプロセスハンドル値になります。

‘l’ および ‘v’ のついた *spawn\**( ) 関数は、コマンドライン引数をどのように渡すかが異なります。‘l’ 型は、コードを書くときにパラメタ数が決まっている場合に、おそらくもっとも簡単に利用できます。個々のパラメタは単に *spawnl\**( ) 関数の追加パラメタとなります。‘v’ 型は、パラメタの数が可変の時に便利で、リストかタプルの引数が *args* パラメタとして渡されます。どちらの場合も、子プロセスに渡す引数は動作させようとしているコマンドの名前から始まらなくてはなりません。

末尾近くに ‘p’ をもつ型 (*spawnlp*( ), *spawnlpe*( ), *spawnvp*( ), および *spawnvpe*( )) は、プログラム *file* を探すために環境変数 PATH を利用します。環境変数が (次の段で述べる *spawn\*e*( ) 型関数で) 置き換えられる場合、環境変数は PATH を決定する上の情報源として使われます。その他の型、*spawnl*( ), *spawnle*( ), *spawnv*( ), および *spawnve*( ) では、実行コードを探すために PATH を使いません。*path* には適切に設定された絶対パスまたは相対パスが入っていないてはなりません。

*spawnle*( ), *spawnlpe*( ), *spawnve*( ), および *spawnvpe*( ) (全て末尾に ‘e’ がついていることに注意してください) では、*env* パラメタは新たなプロセスで利用される環境変数を定義するためのマップ型でなくてはなりません; *spawnl*( ), *spawnlp*( ), *spawnv*( ), および *spawnvp*( ) では、全て新たなプロセスは現在のプロセスの環境を引き継ぎます。

例えば、以下の *spawnlp*( ) および *spawnvpe*( ) 呼び出し:

```
import os
os.spawnlp(os.P_WAIT, 'cp', 'cp', 'index.html', '/dev/null')

L = ['cp', 'index.html', '/dev/null']
os.spawnvpe(os.P_WAIT, 'cp', L, os.environ)
```

は等価です。利用できる環境: UNIX、Windows。

`spawnlp()`、`spawnlpe()`、`spawnvp()` および `spawnvpe()` は Windows では利用できません。1.6 で追加された仕様です。

#### **P\_NOWAIT**

#### **P\_NOWAITO**

`spawn*()` 関数ファミリーに対する *mode* パラメタとして取れる値です。この値のいずれかを *mode* として与えた場合、`spawn*()` 関数は新たなプロセスが生成されるとすぐに、プロセスの ID を戻り値として返ります。利用できる環境: UNIX、Windows。1.6 で追加された仕様です。

#### **P\_WAIT**

`spawn*()` 関数ファミリーに対する *mode* パラメタとして取れる値です。この値を *mode* として与えた場合、`spawn*()` 関数は新たなプロセスを起動して完了するまで返らず、プロセスがうまく終了した場合には終了コードを、シグナルによってプロセスが kill された場合には *-signal* を返します。利用できる環境: UNIX、Windows。1.6 で追加された仕様です。

#### **P\_DETACH**

#### **P\_OVERLAY**

`spawn*()` 関数ファミリーに対する *mode* パラメタとして取れる値です。これらの値は上の値よりもやや可搬性において劣っています。P\_DETACH は P\_NOWAIT に似ていますが、新たなプロセスは呼び出しプロセスのコンソールから切り離され (detach) ます。P\_OVERLAY が使われた場合、現在のプロセスは置き換えられます; 従って `spawn*()` は返りません。利用できる環境: Windows。1.6 で追加された仕様です。

#### **startfile(path)**

ファイルを関連付けられたアプリケーションを使って「スタート」します。この動作は、Windows の Explorer 上でのファイルをダブルクリックや、コマンドプロンプト (interactive command shell) 上でのファイル名を **start** 命令の引数としての実行と同様です: ファイルは拡張子が関連付けされているアプリケーション (が存在する場合) を使って開かれます。

`startfile()` は関連付けされたアプリケーションが起動すると同時に返ります。アプリケーションが閉じるまで待機させるためのオプションはなく、アプリケーションの終了状態を取得する方法もありません。*path* 引数は現在のディレクトリからの相対で表します。絶対パスを利用したいなら、最初の文字はスラッシュ('/') ではないので注意してください; もし最初の文字がスラッシュなら、システムの背後にある Win32 ShellExecute() 関数は動作しません。`os.path.normpath()` 関数を使って、Win32 用に正しくコード化されたパスになるようにしてください。利用できる環境: Windows。2.0 で追加された仕様です。

#### **system(command)**

サブシェル内でコマンド (文字列) を実行します。この関数は標準 C 関数 `system()` を使って実装されており、`system()` と同じ制限があります。`posix.environ`、`sys.stdin` 等に対する変更を行っても、実行されるコマンドの環境には反映されません。

UNIX では、戻り値はプロセスの終了ステータスで、`wait()` で定義されている書式にコード化されています。POSIX は `system()` 関数の戻り値の意味について定義していないので、Python の `system` における戻り値はシステム依存となることに注意してください。

Windows では、戻り値は *command* を実行した後にシステムシェルから返される値で、Windows の環

境変数 COMSPEC となります: **command.com** ベースのシステム (Windows 95, 98 および ME) では、この値は常に 0 です; **cmd.exe** ベースのシステム (Windows NT, 2000 および XP) では、この値は実行したコマンドの終了ステータスです; ネイティブでないシェルを使っているシステムについては、使っているシェルのドキュメントを参照してください。

利用できる環境: UNIX、Windows。

**times()**

(プロセスまたはその他の) 積算時間を秒で表す浮動小数点数からなる、5 要素のタプルを返します。タプルの要素は、ユーザ時間 (user time)、システム時間 (system time)、子プロセスのユーザ時間、子プロセスのシステム時間、そして過去のある固定時点からの経過時間で、この順に並んでいます。UNIX マニュアルページ *times(2)* または対応する Windows プラットフォーム API ドキュメントを参照してください。利用できる環境: UNIX、Windows。

**wait()**

子プロセスの実行完了を待機し、子プロセスの *pid* と終了コードインジケータ— 16 ビットの数で、下位バイトがプロセスを kill したシグナル番号、上位バイトが終了ステータス (シグナル番号がゼロの場合) — の入ったタプルを返します; コアダンプファイルが生成された場合、下位バイトの最上桁ビットが立てられます。利用できる環境: UNIX。

**waitpid(*pid*, *options*)**

プロセス *id pid* で与えられた子プロセスの完了を待機し、子プロセスのプロセス *id* と (*wait()* と同様にコード化された) 終了ステータスインジケータからなるタプルを返します。この関数の動作は *options* によって影響されます。通常の操作では 0 にします。利用できる環境: UNIX。

*pid* が 0 よりも大きい場合、*waitpid()* は特定のプロセスのステータス情報を要求します。 *pid* が 0 の場合、現在のプロセスグループ内の任意の子プロセスの状態に対する要求です。 *pid* が -1 の場合、現在のプロセスの任意の子プロセスに対する要求です。 *pid* が -1 よりも小さい場合、プロセスグループ *-pid* (すなわち *pid* の絶対値) 内の任意のプロセスに対する要求です。

**WNOHANG**

子プロセス状態がすぐに取得できなかった場合にハングアップしてしまわないようにするための *waitpid()* のオプションです。利用できる環境: UNIX。

以下の関数は *system()*、*wait()*、あるいは *waitpid()* が返すプロセス状態コードを引数にとります。これらの関数はプロセスの配置を決めるために利用することができます。

**WIFSTOPPED(*status*)**

プロセスが停止された (stop) 場合に真を返します。利用できる環境: UNIX。

**WIFSIGNALED(*status*)**

プロセスがシグナルによって終了した (exit) 場合に真を返します。利用できる環境: UNIX。

**WIFEXITED(*status*)**

プロセスが *exit(2)* システムコールで終了した場合に真を返します。利用できる環境: UNIX。

**WEXITSTATUS(*status*)**

*WIFEXITED(status)* が真の場合、*exit(2)* システムコールに渡された整数パラメタを返します。そうでない場合、返される値には意味がありません。利用できる環境: UNIX。

**WSTOPSIG(*status*)**

プロセスを停止させたシグナル番号を返します。利用できる環境: UNIX。

**WTERMSIG(*status*)**

プロセスを終了させたシグナル番号を返します。利用できる環境: UNIX



### 6.1.6 雑多なシステム情報

#### `confstr(name)`

文字列形式によるシステム設定値 (system configuration value) を返します。*name* には取得したい設定名を指定します; この値は定義済みのシステム値名を表す文字列にすることができます; 名前は多くの標準 (POSIX.1、UNIX 95、UNIX 98 その他) で定義されています。ホストオペレーティングシステムの関知する名前は `confstr_names` 辞書で与えられています。このマップ型オブジェクトに入っていない設定変数については、*name* に整数を渡してもかまいません。利用できる環境: UNIX。

*name* に指定された設定値が定義されていない場合、空文字列を返します。

もし *name* が文字列でかつ不明である場合、`ValueError` を送出します。*name* の指定値がホストシステムでサポートされておらず、`confstr_names` にも入っていない場合、`errno.EINVAL` をエラー番号として `OSError` を送出します。

#### `confstr_names`

`confstr()` が受理する名前を、ホストオペレーティングシステムで定義されている整数値に対応付けている辞書です。この辞書はシステムでどの設定名が定義されているかを決定するために利用できます。利用できる環境: UNIX。

#### `getloadavg()`

過去 1 分、5 分、15 分間で、システムで走っているキューの平均プロセス数を返します。平均負荷が得られない場合には `OSError` を送出します。2.3 で追加された仕様です。

#### `sysconf(name)`

整数値のシステム設定値を返します。*name* で指定された設定値が定義されていない場合、`-1` が返されます。*name* に関するコメントとしては、`confstr()` で述べた内容が同様に当てはまります; 既知の設定名についての情報を与える辞書は `sysconf_names` で与えられています。利用できる環境: UNIX。

#### `sysconf_names`

`sysconf()` が受理する名前を、ホストオペレーティングシステムで定義されている整数値に対応付けている辞書です。この辞書はシステムでどの設定名が定義されているかを決定するために利用できます。利用できる環境: UNIX。

以下のデータ値はパス名編集操作をサポートするために利用されます。これらの値は全てのプラットフォームで定義されています。

パス名に対する高レベルの操作は `os.path` モジュールで定義されています。

#### `curdir`

現在のディレクトリ参照するためにオペレーティングシステムで使われる文字列定数です。例: POSIX では `'.'`、Machintosh では `'.'`。 `os.path` から利用できます。

#### `pardir`

親ディレクトリを参照するためにオペレーティングシステムで使われる文字列定数です。例: POSIX では `'..'`、Machintosh では `'..'`。 `os.path` から利用できます。

#### `sep`

パス名を要素に分割するためにオペレーティングシステムで利用されている文字で、例えば POSIX では `'/'` で、Machintosh では `':'` です。しかし、このことを知っているだけではパス名を解析したり、パス名同士を結合したりするには不十分です — こうした操作には `os.path.split()` や `os.path.join()` を使ってください — が、たまに便利なこともあります。 `os.path` から利用できます。

#### `altsep`

文字パス名を要素に分割する際にオペレーティングシステムで利用されるもう一つの文字で、分割文



字が一つしかない場合には `None` になります。この値は `sep` がバックスラッシュとなっている DOS や Windows システムでは `'/'` に設定されています。`os.path` から利用できます。

#### **extsep**

ベースのファイル名と拡張子を分ける文字。たとえば、`'os.py'` では `'.'` です。`os.path` から利用できます。2.2 で追加された仕様です。

#### **pathsep**

(PATH のような) サーチパス内の要素を分割するためにオペレーティングシステムが慣習的に用いる文字で、POSIX における `':`' や DOS および Windows における `';`' に相当します。`os.path` から利用できます。

#### **defpath**

`exec*p*()` や `spawn*p*()` において、環境変数辞書内に `'PATH'` キーがない場合に使われる標準設定のサーチパスです。`os.path` から利用できます。

#### **linesep**

現在のプラットフォーム上で行を分割 (あるいは終端) するために用いられている文字列です。この値は例えば POSIX での `'\n'` や MacOS での `'\r'` のように、単一の文字にもなりますし、例えば DOS や Windows での `'\r\n'` のように複数の文字列にもなります。

## 6.2 `os.path` — 共通のパス名操作

このモジュールには、パス名を操作する便利な関数が定義されています。

警告: これらの関数の多くは Windows の一律命名規則 (UNC パス名) を正しくサポートしていません。`splitunc()` と `ismount()` は正しく UNC パス名を操作できます。

#### **abspath(*path*)**

*path* の標準化された絶対パスを返します。たいていのプラットフォームでは、`normpath(join(os.getcwd(), path))` と同じ結果になります。1.5.2 で追加された仕様です。

#### **basename(*path*)**

パス名 *path* の末尾のファイル名を返します。これは `split(path)` で返されるペアの 2 番目の要素です。この関数が返す値は UNIX の `basename` とは異なります; UNIX の `basename` は `'/foo/bar/'` に対して `'bar'` を返しますが、`basename()` は空文字列 (`''`) を返します。

#### **commonprefix(*list*)**

パスの *list* 中の共通する最長のプレフィックスを (パス名の 1 文字 1 文字を判断して) 返します。もし *list* が空なら、空文字列 (`''`) を返します。これは一度に 1 文字を扱うため、不正なパスを返すことがあるかもしれませんので注意して下さい。

#### **dirname(*path*)**

パス *path* のディレクトリ名を返します。これは `split(path)` で返されるペアの最初の要素です。

#### **exists(*path*)**

*path* が存在するなら、`True` を返します。

#### **expanduser(*path*)**

与えられた引数の最初の `'~'` または `'~user'` を、*user* のホームディレクトリのパスに置き換えて返します。最初が `'~'` なら、環境変数の `HOME` の値に置き換えられます; `'~user'` なら、ビルトインモジュール `pwd` を使ってパスワードディレクトリから該当するものを検索します。もし置き換えに失敗したり、引数のパスがチルダで始まっていなかったら、パスをそのまま返します。Macintosh では *path* をそのまま返します。

#### **expandvars(*path*)**

引数のパスを環境変数に展開して返します。引数の中の '*\$name*' または '*\${name}*' の文字列が環境変数の *name* に置き換えられます。不正な変数名や存在しない変数名の場合には変換されず、そのまま返します。Macintosh では *path* をそのまま返します。

#### `getatime(path)`

*path* に最後にアクセスした時刻を、エポック (time モジュールを参照) からの経過時間を示す秒数で返します。ファイルが存在しなかったりアクセスできない場合は `os.error` を発生します。2.3 で変更された仕様: `os.stat_float_times()` が `True` を返す場合、戻り値は浮動小数点値となります。1.5.2 で追加された仕様です。

#### `getmtime(path)`

*path* の最終更新時刻を、エポック (time モジュールを参照) からの経過時間を示す秒数で返します。ファイルが存在しなかったりアクセスできない場合は `os.error` を発生します。2.3 で変更された仕様: `os.stat_float_times()` が `True` を返す場合、戻り値は浮動小数点値となります。1.5.2 で追加された仕様です。

#### `getctime(path)`

システムによって、ファイルの最終変更時刻 (UNIX のようなシステム) や作成時刻 (Windows のようなシステム) をシステムの `ctime` で返します。戻り値はエポック (time モジュールを参照) からの経過秒数を示す数値です。ファイルが存在しなかったりアクセスできない場合は `os.error` を発生します。2.3 で追加された仕様です。

#### `getsize(path)`

ファイル *path* のサイズをバイト数で返します。ファイルが存在しなかったりアクセスできない場合は `os.error` を発生します。1.5.2 で追加された仕様です。

#### `isabs(path)`

*path* が絶対パス (スラッシュで始まる) なら、`True` を返します。

#### `isfile(path)`

*path* が存在する正しいファイルなら、`True` を返します。シンボリックリンクの場合にはその実体をチェックするので、同じパスに対して `islink()` と `isfile()` の両方が `True` を返すことがあります。

#### `isdir(path)`

*path* が存在するなら、`True` を返します。シンボリックリンクの場合にはその実体をチェックするので、同じパスに対して `islink()` と `isfile()` の両方が `True` を返すことがあります。

#### `islink(path)`

*path* がシンボリックリンクなら、`True` を返します。シンボリックリンクがサポートされていないプラットフォームでは、常に `False` を返します。

#### `ismount(path)`

パス名 *path* がマウントポイント *mount point* (ファイルシステムの中で異なるファイルシステムがマウントされているところ) なら、`True` を返します: この関数は *path* の親ディレクトリである '*path/..*' が *path* と異なるデバイス上にあるか、あるいは '*path/..*' と *path* が同じデバイス上の同じ i-node を指しているかをチェックします— これによって全ての UNIX と POSIX 標準でマウントポイントが検出できます。

#### `join(path1[, path2[, ...]])`

パスに 1 つあるいはそれ以上のパスの要素をうまく結合します。付け加える要素に絶対パスがあれば、それより前の要素は全て破棄され、以降の要素を結合します。戻り値は *path1* と省略可能な *path2* 以降を結合したもので、*path2* が空文字列でないなら、ディレクトリの区切り文字 (`os.sep`) が各要素の間に挿入されます。Windows では各ドライブに対してカレントディレクトリがあるので、`os.path.join("c:", "foo")` によって、'*c:\foo*' ではなく、ドライブ '*C:*' 上のカレントディレクトリからの相対パス ('*c:foo*') が返されます。

**normcase**(*path*)

パス名の大文字、小文字をシステムの標準にします。UNIX ではそのまま返します。大文字、小文字を区別しないファイルシステムではパス名を小文字に変換します。Windows では、スラッシュをバックスラッシュに変換します。

**normpath**(*path*)

パス名を標準化します。余分な区切り文字や上位レベル参照を削除します。例えば、A//B、A/. /B、A/foo/. . /B は全て A/B になります。大文字、小文字は標準化しません（それには `normcase()` を使って下さい）。Windows では、スラッシュをバックスラッシュに変換します。

**realpath**(*path*)

パスの中のシンボリックリンクを取り除いて、標準化したパスを返します。利用可能：UNIX。2.2 で追加された仕様です。

**samefile**(*path1*, *path2*)

2つの引数であるパス名が同じファイルあるいはディレクトリを指していれば（同じデバイスナンバーとi-node ナンバーで示されていれば）、True を返します。どちらかのパス名で `os.stat()` の呼び出しに失敗した場合には、例外が発生します。利用可能：Macintosh、UNIX

**sameopenfile**(*fp1*, *fp2*)

*fp1* と *fp2* が同じファイルオブジェクトを指していたら、True を返します。2つのファイルオブジェクトが異なるファイルディスクリプタを示すこともあります。利用可能：Macintosh、UNIX

**samestat**(*stat1*, *stat2*)

stat タプル *stat1* と *stat2* が同じファイルを指していたら、True を返します。これらのタプルは `fstat()`、`lstat()` や `stat()` で返されたものでかまいません。この関数は、`samefile()` と `sameopenfile()` で使われるのと同様なものを背後に実装しています。利用可能：Macintosh、UNIX

**split**(*path*)

パス名 *path* を (*head* と *tail*) のペアに分割します。*tail* はパスの構成要素の末尾で、*head* はそれより前の部分です。*tail* はスラッシュを含みません；もし *path* の最後にスラッシュがあれば、*tail* は空文字列になります。もし *path* にスラッシュがなければ、*head* は空文字列になります。*path* が空文字列なら、*head* と *tail* のどちらも空文字列になります。*head* の末尾のスラッシュは、*head* がルートディレクトリ（1つ以上のスラッシュのみ）でない限り、取り除かれます。ほとんど全ての場合、`join(head, tail)` の結果が *path* と等しくなります（ただ1つの例外は、複数のスラッシュが *head* と *tail* を分けている時です）。

**splitdrive**(*path*)

パス名 *path* を (*drive*, *tail*) のペアに分割します。*drive* はドライブ名か、空文字列です。ドライブ名を使用しないシステムでは、*drive* は常に空文字列です。全ての場合に *drive* + *tail* は *path* と等しくなります。1.3 で追加された仕様です。

**splitext**(*path*)

パス名 *path* を (*root*, *ext*) のペアにします。*root* + *ext* == *path* になります。*ext* は空文字列か1つのピリオドで始まり、多くても1つのピリオドを含みます。

**walk**(*path*, *visit*, *arg*)

*path* をルートとする各ディレクトリに対して（もし *path* がディレクトリなら *path* も含みます）、(*arg*, *dirname*, *names*) を引数として関数 *visit* を呼び出します。引数 *dirname* は訪れたディレクトリを示し、引数 *names* はそのディレクトリ内のファイルのリスト (`os.listdir(dirname)` で得られる) です。関数 *visit* によって *names* を変更して、*dirname* 以下の対象となるディレクトリのセットを変更することもできます。例えば、あるディレクトリツリーだけ関数を適用しないなど。（*names* で参照されるオブジェクトは、`del` あるいはスライスを使って正しく変更しなければなりません。）

注意：ディレクトリへのシンボリックリンクはサブディレクトリとして扱われないので、`walk()`

による操作対象とはされません。ディレクトリへのシンボリックリンクを操作対象とするには、`os.path.islink(file)` と `os.path.isdir(file)` で識別して、`walk()` で必要な操作を実行しなければなりません。

注意: 新たに追加された `os.walk()` ジェネレータを使用すれば、同じ処理をより簡単に行う事ができます。

#### `supports_unicode_filenames`

任意のユニコード文字列を ( ファイルシステムの制限内で ) ファイルネームに使うことが可能で、`os.listdir` がユニコード文字列の引数に対してユニコードを返すなら、真を返します。2.3 で追加された仕様です。

## 6.3 dircache — キャッシュされたディレクトリー一覧の生成

`dircache` モジュールはキャッシュされた情報を使ってディレクトリー一覧を読み出すための関数を定義しています。キャッシュはディレクトリの *mtime* に応じて無効化されます。さらに、一覧中のディレクトリにスラッシュ ( `'/'` ) を追加することでディレクトリであると分かるようにするための関数も定義しています。

`dircache` モジュールは以下の関数を定義しています:

#### `listdir(path)`

`os.listdir()` によって得た *path* のディレクトリー一覧を返します。 *path* を変えない限り、以降の `listdir()` を呼び出してもディレクトリ構造を読み込みなおすことはしないので注意してください。

返されるリストは読み出し専用であると見なされるので注意してください (おそらく将来のバージョンではタプルを返すように変更されるはず? です)。

#### `opendir(path)`

`listdir()` と同じです。以前のバージョンとの互換性のために定義されています。

#### `annotate(head, list)`

*list* を *head* の相対パスからなるリストとして、各パスがディレクトリを指す場合には `'/'` をパス名の後ろに追加したものに置き換えます。

```
>>> import dircache
>>> a=dircache.listdir('/')
>>> a=a[:] # Copy the return value so we can change 'a'
>>> a
['bin', 'boot', 'cdrom', 'dev', 'etc', 'floppy', 'home', 'initrd', 'lib', 'lost+
found', 'mnt', 'proc', 'root', 'sbin', 'tmp', 'usr', 'var', 'vmlinuz']
>>> dircache.annotate('/', a)
>>> a
['bin/', 'boot/', 'cdrom/', 'dev/', 'etc/', 'floppy/', 'home/', 'initrd/', 'lib/
', 'lost+found/', 'mnt/', 'proc/', 'root/', 'sbin/', 'tmp/', 'usr/', 'var/', 'vm
linuz']
```

## 6.4 stat — `stat()` の返す内容を解釈する

`stat` モジュールでは、`os.stat()`、`os.lstat()` および `os.fstat()` (存在すれば) の返す内容を解釈するための定数や関数を定義しています。`stat()`、`fstat()`、および `lstat()` の関数呼び出しについての完全な記述はシステムのドキュメントを参照してください。

`stat` モジュールでは、特殊なファイル型を判別するための以下の関数を定義しています:

`S_ISDIR(mode)`

ファイルのモードがディレクトリの場合にゼロでない値を返します。

`S_ISCHR(mode)`

ファイルのモードがキャラクタ型の特殊デバイスファイルの場合にゼロでない値を返します。

`S_ISBLK(mode)`

ファイルのモードがブロック型の特殊デバイスファイルの場合にゼロでない値を返します。

`S_ISREG(mode)`

ファイルのモードが通常ファイルの場合にゼロでない値を返します。

`S_ISFIFO(mode)`

ファイルのモードが FIFO (名前つきパイプ) の場合にゼロでない値を返します。

`S_ISLNK(mode)`

ファイルのモードがシンボリックリンクの場合にゼロでない値を返します。

`S_ISSOCK(mode)`

ファイルのモードがソケットの場合にゼロでない値を返します。

より一般的なファイルのモードを操作するための二つの関数が定義されています:

`S_IMODE(mode)`

`os.chmod()` で設定することのできる一部のファイルモード — すなわち、ファイルの許可ビット (permission bits) に加え、(サポートされているシステムでは) スティッキービット (sticky bit)、実行グループ ID 設定 (set-group-id) および 実行ユーザ ID 設定 (set-user-id) ビット — を返します。

`S_IFMT(mode)`

ファイルの形式を記述しているファイルモードの一部 (上記の `S_IS*()` 関数で使われます) を返します。

通常、ファイルの形式を調べる場合には `os.path.is*()` 関数を使うことになります; ここで挙げた関数は同じファイルに対して複数のテストを同時に行いたい、`stat()` システムコールを何度も呼び出してオーバーヘッドが生じるのを避けたい場合に便利です。これらはまた、ブロック型およびキャラクタ型デバイスに対するテストのように、`os.path` で扱うことのできないファイルの情報を調べる際にも便利です。

以下の全ての変数は、`os.stat()`、`os.fstat()`、または `os.lstat()` が返す 10 要素のタプルにおけるインデックスを単にシンボル定数化したものです。

`ST_MODE`

I ノードの保護モード。

`ST_INO`

I ノード番号。

`ST_DEV`

I ノードが存在するデバイス。

`ST_NLINK`

該当する I ノードへのリンク数。

`ST_UID`

ファイルの所持者のユーザ ID。

`ST_GID`

ファイルの所持者のグループ ID。

`ST_SIZE`

通常ファイルではバイトサイズ; いくつかの特殊ファイルでは処理待ちのデータ量。

`ST_ATIME`

最後にアクセスした時刻。

**ST\_MTIME**

最後に変更された時刻。

**ST\_CTIME**

最後に状態が変更された時刻 (詳細についてはマニュアルページを参照してください)。

“ファイルサイズ”の解釈はファイルの型によって異なります。通常のファイルの場合、サイズはファイルの大きさをバイトで表したものです。ほとんどの UNIX 系 (特に Linux) における FIFO やソケットの場合、“サイズ”は `os.stat()`、`os.fstat()`、あるいは `os.lstat()` を呼び出した時点で読み出し待ちであったデータのバイト数になります; この値は時に有用で、特に上記の特殊なファイルを非ブロックモードで開いた後にポーリングを行いたいといった場合に便利です。他のキャラクタ型およびブロック型デバイスにおけるサイズフィールドの意味はさらに異なっていて、背後のシステムコールの実装によります。

例を以下に示します:

```
import os, sys
from stat import *

def walktree(top, callback):
    '''recursively descend the directory tree rooted at top,
    calling the callback function for each regular file'''

    for f in os.listdir(top):
        pathname = os.path.join(top, f)
        mode = os.stat(pathname)[ST_MODE]
        if S_ISDIR(mode):
            # It's a directory, recurse into it
            walktree(pathname, callback)
        elif S_ISREG(mode):
            # It's a file, call the callback function
            callback(pathname)
        else:
            # Unknown file type, print a message
            print 'Skipping %s' % pathname

def visitfile(file):
    print 'visiting', file

if __name__ == '__main__':
    walktree(sys.argv[1], visitfile)
```

## 6.5 statcache — `os.stat()` の最適化

リリース 2.2 以降で撤廃された仕様です。 キャッシュを使わずに `os.stat()` を直接使ってください; キャッシュはそれを使うアプリケーションに非常に高いレベルの脆弱性をもたらし、キャッシュ管理のサポートを追加することでアプリケーションのソースコードを難解にしまいます。

statcache モジュールは `os.stat()` を最後に呼び出した際の値を記憶することによる最適化機能を提供します。

statcache モジュールでは以下の関数を定義しています:

**stat(path)**

このモジュールの主要なエントリポイントです。

`os.stat()` と同じですが、将来同じ呼び出しが行われたときの為に結果を記憶しておきます。



その他の関数はキャッシュやその一部を消去するために用いられます。

**reset()**

キャッシュを消去します: これまでの全ての `stat()` 呼び出しによる結果を忘却します。

**forget(path)**

`stat(path)` のキャッシュが存在すれば、それを忘却します。

**forget\_prefix(prefix)**

`prefix` で始まる `path` に対する `stat(path)` の結果をすべて忘却します。

**forget\_dir(prefix)**

`prefix` で始まる `path` に対する `stat(path)` の結果を、`stat(prefix)` も含めてすべて忘却します。

**forget\_except\_prefix(prefix)**

`forget_prefix()` に似ていますが、`prefix` で始まらない全ての `path` について忘却します。

以下に例を示します:

```
>>> import os, statcache
>>> statcache.stat('.')
(16893, 2049, 772, 18, 1000, 1000, 2048, 929609777, 929609777, 929609777)
>>> os.stat('.')
(16893, 2049, 772, 18, 1000, 1000, 2048, 929609777, 929609777, 929609777)
```

## 6.6 statvfs — `os.statvfs()` で使われる定数群

`statvfs` モジュールでは、`os.statvfs()` の返す値を解釈するための定数を定義しています。`os.statvfs()` は“マジックナンバ”を記憶せずにタプルを生成して返します。このモジュールで定義されている各定数は `os.statvfs()` が返すタプルにおいて、特定の情報が収められている各エントリへのインデクスです。

**F\_BSIZE**

選択されているファイルシステムのブロックサイズです。

**F\_FRSIZE**

ファイルシステムの基本ブロックサイズです。

**F\_BLOCKS**

ブロック数の総計です。

**F\_BFREE**

空きブロック数の総計です。

**F\_BAVAIL**

非スーパーユーザが利用できる空きブロック数です。

**F\_FILES**

ファイルノード数の総計です。

**F\_FFREE**

空きファイルノード数の総計です。

**F\_FAVAIL**

非スーパーユーザが利用できる空きノード数です。

**F\_FLAG**

フラグで、システム依存です: `statvfs()` マニュアルページを参照してください。

**F\_NAMEMAX**

ファイル名の最大長です。

## 6.7 filecmp — ファイルおよびディレクトリの比較

filecmp モジュールでは、ファイルおよびディレクトリを比較するため、様々な時間 / 正確性のトレードオフに関するオプションを備えた関数を定義しています。

filecmp モジュールでは以下の関数を定義しています:

**cmp**(*f1*, *f2*[, *shallow*[, *use\_statcache*]])

名前が *f1* および *f2* のファイルを比較し、二つのファイルが同じらしければ **True** を返し、そうでなければ **false** を返します。

*shallow* が与えられておりかつ偽でなければ、`os.stat()` の返すシグネチャが一致するファイルは同じであると見なされます。2.3 で変更された仕様: *use\_statcache* は廃止され、指定しても無視されます。

この関数で比較されたファイルは `os.stat()` シグネチャが変更されるまで再び比較されることはありません。*use\_statcache* を真にすると、キャッシュ無効化機構を失敗させます — そのため、statcache のキャッシュから古いファイル stat 値が使われます。

可搬性と効率のために、個の関数は外部プログラムを一切呼び出さないで注意してください。

**cmpfiles**(*dir1*, *dir2*, *common*[, *shallow*[, *use\_statcache*]])

ファイル名からなる 3 つのリスト: *match*、*mismatch*、*errors* を返します。*match* には双方のディレクトリで一致したファイルのリストが含まれ、*mismatch* にはそうでないファイル名のリストが入ります。そして *errors* は比較されなかったファイルが列挙されます。ファイルによっては、ユーザにそのファイルを読む権限がなかったり、比較を完了することができなかった場合以外のその他諸々の理由により、*errors* に列挙されることがあります。

引数 *common* は両方のディレクトリにあるファイルのリストです。引数 *shallow* および *use\_statcache* はその意味も標準の設定も `filecmp.cmp()` と同じです。

例:

```
>>> import filecmp
>>> filecmp.cmp('libundoc.tex', 'libundoc.tex')
True
>>> filecmp.cmp('libundoc.tex', 'lib.tex')
False
```

### 6.7.1 dircmp クラス

dircmp のインスタンスは以下のコンストラクタで生成されます:

**class dircmp**(*a*, *b*[, *ignore*[, *hide*]])

ディレクトリ *a* および *b* を比較するための新しいディレクトリ比較オブジェクトを生成します。*ignore* は比較の際に無視するファイル名のリストで、標準の設定では ['RCS', 'CVS', 'tags'] です。*hide* は表示しない名前のリストで、標準の設定では [os.curdir, os.pardir] です。

dircmp クラスは以下のメソッドを提供しています:

**report**()

*a* および *b* の間の比較結果を (sys.stdout に) 出力します。

`report_partial_closure()`

*a* および *b* およびそれらの直下にある共通のサブディレクトリ間での比較結果を出力します。

`report_full_closure()`

*a* および *b* およびそれらの共通のサブディレクトリ間での比較結果を (再帰的に比較して) 出力します。

`dircmp` は、比較しているディレクトリツリーに関する様々な種類の情報を取得するために使えるような、多くの興味深い属性を提供しています。

`__getattr__()` フックを経由すると、全ての属性をのろのろと計算するため、速度上のペナルティを受けないのは計算処理の軽い属性を使ったときだけなので注意してください。

`left_list`

*a* にあるファイルおよびサブディレクトリです。 *hide* および *ignore* でフィルタされています。

`right_list`

*b* にあるファイルおよびサブディレクトリです。 *hide* および *ignore* でフィルタされています。

`common`

*a* および *b* の両方にあるファイルおよびサブディレクトリです。

`left_only`

*a* だけにあるファイルおよびサブディレクトリです。

`right_only`

*b* だけにあるファイルおよびサブディレクトリです。

`common_dirs`

*a* および *b* の両方にあるサブディレクトリです。

`common_files`

*a* および *b* の両方にあるファイルです。

`common_funny`

*a* および *b* の両方にあり、ディレクトリ間でタイプが異なるか、`os.stat()` がエラーを報告するような名前です。

`same_files`

*a* および *b* 両方にあり、一致するファイルです。

`diff_files`

*a* および *b* 両方にあるが、一致しないファイルです。

`funny_files`

*a* および *b* 両方にあるが、比較されなかったファイルです。

`subdirs`

`common_dirs` のファイル名を `dircmp` オブジェクトに対応付けた辞書です。

## 6.8 popen2 — アクセス可能な I/O ストリームを持つ子プロセス生成

このモジュールにより、UNIX および Windows でプロセスを起動し、その入力 / 出力 / エラー出力パイプに接続し、そのリターンコードを取得することができます。

Python 2.0 から、この機能は `os` モジュールにある関数を使って得ることができるので注意してください。`os` にある関数はこのモジュールにおけるファクトリ関数と同じ名前を持ちますが、戻り値に関する取り決めは `os` の関数の方がより直感的です。

このモジュールで提供されている第一のインタフェースは 3 つのファクトリ関数です。これらの関数のいずれも、`bufsize` を指定した場合、I/O パイプのバッファサイズを決定します。`mode` を指定する場合、文

字列 'b' または 't' でなければなりません; Windows では、ファイルオブジェクトをバイナリあるいはテキストモードのどちらで開くかを決めなければなりません。mode の標準の値は 't' です。

子プロセスからのリターンコードを取得するには、Popen3 および Popen4 クラスの poll() あるいは wait() メソッドを使うしかありません; これらの機能は UNIX でしか利用できません。この情報は popen2(), popen3(), および popen4() 関数、あるいは os モジュールにおける同等の関数の使用によって得ることができません。

**popen2**(cmd[, bufsize[, mode]])

cmd をサブプロセスとして実行します。ファイルオブジェクト (child\_stdout, child\_stdin) を返します。

**popen3**(cmd[, bufsize[, mode]])

cmd をサブプロセスとして実行します。ファイルオブジェクト (child\_stdout, child\_stdin, child\_stderr) を返します。

**popen4**(cmd[, bufsize[, mode]])

cmd をサブプロセスとして実行します。ファイルオブジェクト (child\_stdout\_and\_stderr, child\_stdin). 2.0 で追加された仕様です。

UNIX では、ファクトリ関数によって返されるオブジェクトを定義しているクラスも利用することができます。これらのオブジェクトは Windows 実装で使われていないため、そのプラットフォーム上で使うことはできません。

**class Popen3**(cmd[, capturestderr[, bufsize]])

このクラスは子プロセスを表現します。通常、Popen3 インスタンスは上で述べた popen2() および popen3() ファクトリ関数を使って生成されます。

Popen3 オブジェクトを生成するためにいずれかのヘルパー関数を使っていないのなら、cmd パラメータは子プロセスで実行するシェルコマンドになります。capturestderr フラグが真であれば、このオブジェクトが子プロセスの標準エラー出力を捕獲しなければならないことを意味します。標準の値は偽です。bufsize パラメータが存在する場合、子プロセスへの / からの I/O バッファのサイズを指定します。

**class Popen4**(cmd[, bufsize])

Popen3 に似ていますが、標準エラー出力を標準出力と同じファイルオブジェクトで捕獲します。このオブジェクトは通常 popen4() で生成されます。2.0 で追加された仕様です。

## 6.8.1 Popen3 および Popen4 オブジェクト

Popen3 および Popen4 クラスのインスタンスは以下のメソッドを持ちます:

**poll()**

子プロセスがまだ終了していない際には -1 を、そうでない場合にはリターンコードを返します。

**wait()**

子プロセスの状態コード出力を待機して返します。状態コードでは子プロセスのリターンコードと、プロセスが exit() によって終了したか、あるいはシグナルによって死んだかについての情報を符号化しています。状態コードの解釈を助けるための関数は os モジュールで定義されています; [6.1.5 節](#)の W\*() 関数ファミリーを参照してください。

以下の属性も利用可能です:

**fromchild**

子プロセスからの出力を提供するファイルオブジェクトです。Popen4 インスタンスの場合、この値は標準出力と標準エラー出力の両方を提供するオブジェクトになります。

**tochild**

子プロセスへの入力を提供するファイルオブジェクトです。

**childerr**

コンストラクタに *capturestderr* を渡した際には子プロセスからの標準エラーを提供するファイルオブジェクトで、そうでない場合 *None* になります。Popen4 インスタンスでは、この値は常に *None* になります。

**pid**

子プロセスのプロセス番号です。

## 6.8.2 フロー制御の問題

何らかの形式でプロセス間通信を利用している際には常に、制御フローについて注意深く考える必要があります。これはこのモジュール (あるいは *os* モジュールにおける等価な機能) で生成されるファイルオブジェクトの場合にもあてはまります。

親プロセスが子プロセスの標準出力を読み出している一方で、子プロセスが大量のデータを標準エラー出力に書き込んでいる場合、この子プロセスから出力を読み出そうとするとデッドロックが発生します。同様の状況は読み書きの他の組み合わせでも生じます。本質的な要因は、一方のプロセスが別のプロセスでブロック型の読み出しをしている際に、*\_PC\_PIPE\_BUF* バイトを超えるデータがブロック型の入出力を行うプロセスによって書き込まれることにあります。

こうした状況を扱うには幾つかのやりかたがあります。

多くの場合、もっとも単純なアプリケーションに対する変更は、親プロセスで以下のようなモデル:

```
import popen2

r, w, e = popen2.popen3('python slave.py')
e.readlines()
r.readlines()
r.close()
e.close()
w.close()
```

に従うようにし、子プロセスで以下:

```
import os
import sys

# note that each of these print statements
# writes a single long string

print >>sys.stderr, 400 * 'this is a test\n'
os.close(sys.stderr.fileno())
print >>sys.stdout, 400 * 'this is another test\n'
```

のようなコードにすることでしょう。

とりわけ、*sys.stderr* は全てのデータを書き込んだ後に閉じられなければならないということに注意してください。さもなければ、*readlines()* は返ってきません。また、*sys.stderr.close()* が *stderr* を閉じないように *os.close()* を使わなければならないことにも注意してください。(そうでなく、*sys.stderr* に関連付けると、暗黙のうちに閉じられてしまうので、それ以降のエラーが出力されません)。

より一般的なアプローチををサポートする必要があるアプリケーションでは、パイプ経由の I/O を *se-*

`lect()` ループでまとめるか、個々の `popen*()` 関数や `Popen*` クラスが提供する各々のファイルに対して、個別のスレッドを使って読み出しを行います。

## 6.9 datetime — 基本的な日付型および時間型

2.3 で追加された仕様です。

`datetime` モジュールでは、日付や時間データを簡単な方法と複雑な方法の両方で操作するためのクラスを提供しています。日付や時刻を対象にした四則演算がサポートされている一方で、このモジュールの実装では出力の書式化や操作を目的としたデータメンバの効率的な取り出しに焦点を絞っています。

日付および時刻オブジェクトには、“naive” および “aware” の 2 種類があります。この区別はオブジェクトがタイムゾーンや夏時間、あるいはその他のアルゴリズム的、政治的な理由による時刻の修正に関する何らかの表記をもつかどうかによるものです。特定の数字がメートルか、マイルか、質量を表すかといったことがプログラムの問題であるように、naive な `datetime` オブジェクトが標準世界時 (UTC: Coordinated Universal time) を表現するか、ローカルの時刻を表現するか、あるいは他のいずれかのタイムゾーンにおける時刻を表現するかは純粋にプログラムの問題となります。naive な `datetime` オブジェクトは、現実世界のいくつかの側面を無視するという犠牲のもとに、理解しやすく、かつ利用しやすくなっています。

より多くの情報を必要とするアプリケーションのために、`datetime` および `time` オブジェクトはオプションのタイムゾーン情報メンバ、`tzinfo` を持っています。このメンバには抽象クラス `tzinfo` のサブクラスのインスタンスが入っています。`tzinfo` オブジェクトは UTC 時刻からのオフセット、タイムゾーン名、夏時間が有効になっているかどうか、といった情報を記憶しています。`datetime` モジュールでは具体的な `tzinfo` クラスを提供していないので注意してください。必要な詳細仕様を備えたタイムゾーン機能を提供するのはアプリケーションの責任です。世界各国における時刻の修正に関する法則は合理的というよりも政治的なものであり、全てのアプリケーションに適した標準というものが存在しないのです。

`datetime` モジュールでは以下の定数を公開しています：

### **MINYEAR**

`date` や `datetime` オブジェクトで許されている、年を表現する最小の数字です。MINYEAR は 1 です。

### **MAXYEAR**

`date` や `datetime` オブジェクトで許されている、年を表現する最大の数字です。MAXYEAR は 9999 です。

参考資料：

`calendar` モジュール (5.17 節)：

汎用のカレンダー関連関数。

`time` モジュール (6.10 節)：

時刻へのアクセスと変換。

### 6.9.1 利用可能なデータ型

#### **class date**

理想化された naive な日付表現で、実質的には、これまでもこれからも現在のグレゴリ暦 (Gregorian calendar) であると仮定しています。属性: `year`、`month`、および `day`。

#### **class time**

理想化された時刻表現で、あらゆる特定の日における影響から独立しており、毎日厳密に 24\*60\*60 秒であると仮定します ("うるう秒: leap seconds" の概念はありません)。属性: `hour`、`minute`、`second`、



microsecond、および tzinfo。

#### **class datetime**

日付と時刻を組み合わせたもの。属性: year、month、day、hour、minute、second、microsecond、および tzinfo。

#### **class timedelta**

date、time、あるいは datetime クラスの二つのインスタンス間の時間差をマイクロ秒精度で表す経過時間値です。

#### **class tzinfo**

タイムゾーン情報オブジェクトの抽象基底クラスです。datetime および time クラスで用いられ、カスタマイズ可能な時刻修正の概念 (たとえばタイムゾーンや夏時間の計算) を提供します。

これらの型のオブジェクトは変更不可能 (immutable) です。

date 型のオブジェクトは常に naive です。

time や datetime 型のオブジェクト *d* は naive にも aware にもできます。*d* は *d.tzinfo* が None でなく、かつ *d.tzinfo.utcoffset(d)* が None を返さない場合に aware となります。*d.tzinfo* が None の場合や、*d.tzinfo* は None ではないが *d.tzinfo.utcoffset(d)* が None を返す場合には、*d* は naive となります。

naive なオブジェクトと aware なオブジェクトの区別は timedelta オブジェクトにはあてはまりません。サブクラスの関係は以下ようになります:

```
object
  timedelta
  tzinfo
  time
  date
    datetime
```

## 6.9.2 timedelta オブジェクト

timedelta オブジェクトは経過時間、すなわち二つの日付や時刻間の差を表します。

**class timedelta**(days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0, weeks=0)

全ての引数がオプションです。引数は整数、長整数、浮動小数点数にすることができ、正でも負でもかまいません。

days、seconds および microseconds のみが内部に記憶されます。引数は以下のようにして変換されます:

- 1 ミリ秒は 1000 マイクロ秒に変換されます。
- 1 分は 60 秒に変換されます。
- 1 時間は 3600 秒に変換されます。
- 1 週間は 7 日に変換されます。

その後、日、秒、マイクロ秒は値が一意に表されるように、

- 0 ≤ microseconds < 1000000
- 0 ≤ seconds < 3600\*24 (一日中の秒数)
- -999999999 ≤ days ≤ 999999999

で正規化されます。

引数のいずれかが浮動小数点であり、小数のマイクロ秒が存在する場合、小数のマイクロ秒は全ての引数から一度取り置かれ、それらの和は最も近いマイクロ秒に丸められます。浮動小数点の引数がない場合、値の変換と正規化の過程は厳密な (失われる情報がない) ものとなります。

日の値を正規化した結果、指定された範囲の外側になった場合には、`OverflowError` が送出されます。

負の値を正規化すると、一見混乱するような値になります。例えば、

```
>>> d = timedelta(microseconds=-1)
>>> (d.days, d.seconds, d.microseconds)
(-1, 86399, 999999)
```

クラス属性を以下に示します:

**min**

最小の値を表す `timedelta` オブジェクトで、`timedelta(-999999999)` です。

**max**

最大の値を表す `timedelta` オブジェクトで、`timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)` です。

**resolution**

`timedelta` オブジェクトが等しくない最小の時間差で、`timedelta(microseconds=1)` です。

正規化のために、`timedelta.max > -timedelta.min` となるので注意してください。 `-timedelta.max` は `timedelta` オブジェクトとして表現することができません。

以下に (読み出し専用の) インスタンス属性を示します:

属性	値
<code>days</code>	両端値を含む -999999999 から 999999999 の間
<code>seconds</code>	両端値を含む 0 から 86399 の間
<code>microseconds</code>	両端値を含む 0 から 999999 の間

サポートされている操作を以下に示します:

演算	結果
$t1 = t2 + t3$	$t2$ と $t3$ を加算します。演算後、 $t1 - t2 == t3$ および $t1 - t3 == t2$ は真になります。(1)
$t1 = t2 - t3$	$t2$ と $t3$ の差分です。演算後、 $t1 == t2 - t3$ および $t2 == t1 + t3$ は真になります。(1)
$t1 = t2 * i$ or $t1 = i * t2$	整数や長整数による乗算です。演算後、 $t1 // i == t2$ は $i != 0$ であれば真となります。一般的に、 $t1 * i == t1 * (i-1) + t1$ は真となります。(1)
$t1 = t2 // i$	端数を切り捨てて除算され、剰余 (がある場合) は捨てられます。(3)
$+t1$	同じ値を持つ <code>timedelta</code> オブジェクトを返します。(2)
$-t1$	<code>timedelta(-t1.days, -t1.seconds, -t1.microseconds)</code> 、および $t1 * -1$ と同じです。(1)(4)
$abs(t)$	$t.days >= 0$ のときには $+t$ 、 $t.days < 0$ のときには $-t$ となります。(2)

注釈:

(1) この操作は厳密ですが、オーバーフローするかもしれません。

(2) この操作は厳密であり、オーバーフローしないはずです。

(3) 0 による除算は `ZeroDivisionError` を送出します。

(4) `-timedelta.max` は `timedelta` オブジェクトで表現することができません。

上に列挙した操作に加えて、`timedelta` オブジェクトは `date` および `datetime` オブジェクトとの間で加減算をサポートしています (下を参照してください)。

`timedelta` オブジェクト間の比較はサポートされており、より小さい経過時間を表す `timedelta` オブジェクトがより小さい `timedelta` と見なされます。型混合の比較がデフォルトのオブジェクトアドレス比較となってしまうのを抑止するために、`timedelta` オブジェクトと異なる型のオブジェクトが比較されると、比較演算子が `==` または `!=` でないかぎり `TypeError` が送出されます。後者の場合、それぞれ `False` または `True` を返します。

`timedelta` オブジェクトはハッシュ可能 (辞書のキーとして利用可能) であり、効率的な pickle 化をサポートします。また、ブール演算コンテキストでは、`timedelta` オブジェクトは `timedelta(0)` に等しくない場合かつそのときに限り真となります。

### 6.9.3 date オブジェクト

`date` オブジェクトは日付 (年、月、および日) を表します。日付は理想的なカレンダー、すなわち現在のグレゴリ暦を過去と未来の両方向に無限に延長したもので表されます。1 年の 1 月 1 日は日番号 1、1 年 1 月 2 日は日番号 2、となっていくます。この暦法は、全ての計算における基本カレンダーである、Dershowitz と Reingold の書籍 *Calendrical Calculations* における "予期的グレゴリ (proleptic Gregorian)" 暦の定義に一致します。

`class date(year, month, day)`

全ての引数が必要です。引数は整数でも長整数でもよく、以下の範囲に入らなければなりません:

- `MINYEAR <= year <= MAXYEAR`
- `1 <= month <= 12`
- `1 <= day <=` 指定された月と年における日数

範囲を超えた引数を与えた場合、`ValueError` が送出されます。

他のコンストラクタ、および全てのクラスメソッドを以下に示します:

`today()`

現在のローカルな日付を返します。`date.fromtimestamp(time.time())` と等価です。

`fromtimestamp(timestamp)`

`time.time()` が返すような POSIX タイムスタンプに対応する、ローカルな日付を返します。タイムスタンプがプラットフォームにおける C 関数 `localtime()` でサポートされている範囲を超えている場合には `ValueError` を送出することがあります。この値はよく 1970 年から 2038 年に制限されていることがあります。うるう秒がタイムスタンプの概念に含まれている非 POSIX システムでは、`fromtimestamp()` はうるう秒を無視します。

`fromordinal(ordinal)`

予期的グレゴリ暦順序に対応する日付を表し、1 年 1 月 1 日が序数 1 となります。1 <= ordinal <= `date.max.toordinal()` でない場合、`ValueError` が送出されます。任意の日付 *d* に対し、`date.fromordinal(d.toordinal()) == d` となります。

以下にクラス属性を示します:

`min`

表現できる最も古い日付で、`date(MINYEAR, 1, 1)` です。

**max**

表現できる最も新しい日付で、`date(MAXYEAR, 12, 31)` です。

**resolution**

等しくない日付オブジェクト間の最小の差で、`timedelta(days=1)` です。

以下に (読み出し専用の) インスタンス属性を示します:

**year**

両端値を含む MINYEAR から MAXYEAR までの値です。

**month**

両端値を含む 1 から 12 までの値です。

**day**

1 から与えられた月と年における日数までの値です。

サポートされている操作を以下に示します:

演算	結果
$date2 = date1 + timedelta$	$date2$ はから $date1$ から $timedelta.days$ 日移動した日付です。 (1)
$date2 = date1 - timedelta$	$date2 + timedelta == date1$ であるような日付 $date2$ を計算します。 (2)
$timedelta = date1 - date2$	(3)
$date1 < date2$	$date1$ が時刻として $date2$ よりも前を表す場合に、 $date1$ は $date2$ よりも小さいと見なされま

注釈:

- (1)  $date2$  は  $timedelta.days > 0$  の場合進む方向に、 $timedelta.days < 0$  の場合戻る方向に移動します。演算後は、 $date2 - date1 == timedelta.days$  となります。 $timedelta.seconds$  および  $timedelta.microseconds$  は無視されます。 $date2.year$  が MINYEAR になってしまったり、MAXYEAR より大きくなってしまう場合には `OverflowError` が送出されます。
- (2) この操作は  $date1 + (-timedelta)$  と等価ではありません。なぜならば、 $date1 - timedelta$  がオーバーフローしない場合でも、 $-timedelta$  単体がオーバーフローする可能性があるからです。 $timedelta.seconds$  および  $timedelta.microseconds$  は無視されます。
- (3) この演算は厳密で、オーバーフローしません。 $timedelta.seconds$  および  $timedelta.microseconds$  は 0 で、演算後には  $date2 + timedelta == date1$  となります。
- (4) 別の言い方をすると、 $date1.toordinal() < date2.toordinal()$  であり、かつそのときに限り  $date1 < date2$  となります。型混合の比較がデフォルトのオブジェクトアドレス比較となってしまうのを抑止するために、 $timedelta$  オブジェクトと異なる型のオブジェクトが比較されると `TypeError` が送出されます。しかしながら、被比較演算子のもう一方が `timetuple` 属性を持つ場合には `NotImplemented` が返されます。このフックにより、他種の日付オブジェクトに型混合比較を実装するチャンスを与えています。そうでない場合、 $timedelta$  オブジェクトと異なる型のオブジェクトが比較されると、比較演算子が `==` または `!=` でないかぎり `TypeError` が送出されます。後者の場合、それぞれ `False` または `True` を返します。

`date` オブジェクトは辞書のキーとして用いることができます。ブール演算コンテキストでは、全ての `date` オブジェクトは真であるとみなされます。

以下にインスタンスメソッドを示します:

**replace(year, month, day)**

キーワード引数で指定されたデータメンバが置き換えられることを除き、同じ値を持つ `date` オブ

ジェクトを返します。例えば、`d == date(2002, 12, 31)` とすると、`d.replace(day=26)` `== date(2000, 12, 26)` となります。

#### `timetuple()`

`time.localtime()` が返す形式の `time.struct_time` を返します。時間、分、および秒は 0 で、DST フラグは -1 になります。`d.timetuple()` は `time.struct_time((d.year, d.month, d.day, 0, 0, 0, d.weekday(), d.toordinal() - date(d.year, 1, 1).toordinal() + 1, -1))` と等価です。

#### `toordinal()`

予測的グレゴリ暦における日付序数を返します。1 年の 1 月 1 日が序数 1 となります。任意の `date` オブジェクト `d` について、`date.fromordinal(d.toordinal()) == d` となります。

#### `weekday()`

月曜日を 0、日曜日を 6 として、曜日を整数で返します。例えば、`date(2002, 12, 4).weekday() == 2` であり、水曜日を示します。`isoweekday()` も参照してください。

#### `isoweekday()`

月曜日を 1、日曜日を 7 として、曜日を整数で返します。例えば、`date(2002, 12, 4).weekday() == 3` であり、水曜日を示します。`weekday()`、`isocalendar()` も参照してください。

#### `isocalendar()`

3 要素のタプル (ISO 年、ISO 週番号、ISO 曜日) を返します。

ISO カレンダーはグレゴリ暦の変種として広く用いられています。細かい説明については <http://www.phys.uu.nl/~vgent/calendar/isocalendar.htm> を参照してください。

ISO 年は完全な週が 52 または 53 週あり、週は月曜から始まって日曜に終わります。ISO 年でのある年における最初の週は、その年の木曜日を含む最初の (グレゴリ暦での) 週となります。この週は週番号 1 と呼ばれ、この木曜日での ISO 年はグレゴリ暦における年と等しくなります。

例えば、2004 年は木曜日から始まるため、ISO 年の最初の週は 2003 年 12 月 29 日、月曜日から始まり、2004 年 1 月 4 日、日曜日に終わります。従って、`date(2003, 12, 29).isocalendar() == (2004, 1, 1)` であり、かつ `date(2004, 1, 4).isocalendar() == (2004, 1, 7)` となります。

#### `isoformat()`

ISO 8601 形式、'YYYY-MM-DD' の日付を表す文字列を返します。例えば、`date(2002, 12, 4).isoformat() == '2002-12-04'` となります。

#### `__str__()`

`date` オブジェクト `d` において、`str(d)` は `d.isoformat()` と等価です。

#### `ctime()`

日付を表す文字列を、例えば `date(2002, 12, 4).ctime() == 'Wed Dec 4 00:00:00 2002'` のようにして返します。ネイティブの C 関数 `ctime()` (`time.ctime()` はこの関数を呼び出しますが、`date.ctime()` は呼び出しません) が C 標準に準拠しているプラットフォームでは、`d.ctime()` は `time.ctime(time.mktime(d.timetuple()))` と等価です。

#### `strftime(format)`

明示的な書式化文字列で制御された、日付を表現する文字列を返します。時間、分、秒を表す書式化コードは値 0 になります。`strftime()` のふるまいについてのセクションを参照してください。

## 6.9.4 datetime オブジェクト

datetime オブジェクトは date オブジェクトおよび time オブジェクトの全ての情報が入っている単一のオブジェクトです。date オブジェクトと同様に、datetime は現在のグレゴリ暦が両方向に延長されているものと仮定します; また、time オブジェクトと同様に、datetime は毎日が厳密に 3600\*24 秒であると仮定します。

以下にコンストラクタを示します:

```
class datetime(year, month, day, hour=0, minute=0, second=0, microsecond=0, tzinfo=None)
```

年、月、および日の引数は必須です。tzinfo は None または tzinfo クラスのサブクラスのインスタンスにすることができます。残りの引数は整数または長整数で、以下のような範囲に入ります:

- MINYEAR <= year <= MAXYEAR
- 1 <= month <= 12
- 1 <= day <= 与えられた年と月における日数
- 0 <= hour < 24
- 0 <= minute < 60
- 0 <= second < 60
- 0 <= microsecond < 1000000

引数がこれらの範囲外にある場合、ValueError が送出されます。

その他のコンストラクタ、およびクラスメソッドを以下に示します:

**today()**

現在のローカルな datetime を tzinfo が None であるものとして返します。これは datetime.fromtimestamp(time.time()) と等価です。now()、fromtimestamp() も参照してください。

**now(tz=None)()**

現在のローカルな日付および時刻を返します。オプションの引数 tz が None であるか指定されていない場合、このメソッドは today() と同様ですが、可能ならば time.time() タイムスタンプを通じて得ることができるより高い精度で時刻を提供します (例えば、プラットフォームが C 関数 gettimeofday() をサポートする場合には可能なことがあります)。

そうでない場合、tz はクラス tzinfo のサブクラスのインスタンスでなければならず、現在の日付および時刻は tz のタイムゾーンに変換されます。この場合、結果は tz.fromutc(datetime.utcnow().replace(tzinfo=tz)) と等価になります。today(), utcnow() も参照してください。

**utcnow()**

現在の UTC における日付と時刻を、tzinfo が None であるものとして返します。このメソッドは now() に似ていますが、現在の UTC における日付と時刻を naive な datetime オブジェクトとして返します。now() も参照してください。

**fromtimestamp(timestamp, tz=None)**

time.time() が返すような、POSIX タイムスタンプに対応するローカルな日付と時刻を返します。オプションの引数 tz が None であるか、指定されていない場合、タイムスタンプはプラットフォームのローカルな日付および時刻に変換され、返される datetime オブジェクトは naive なものになります。

そうでない場合、tz はクラス tzinfo のサブクラスのインスタンスでなければならず、現在の日付および時刻は tz のタイムゾーンに変換されます。この場合、結果は



`tz.fromutc(datetime.utcfromtimestamp(timestamp).replace(tzinfo=tz))` と等価になります。

タイムスタンプがプラットフォームの C 関数 `localtime()` や `gmtime()` でサポートされている範囲を超えた場合、`fromtimestamp()` は `ValueError` を送出することがあります。この範囲はよく 1970 年から 2038 年に制限されています。うるう秒がタイムスタンプの概念に含まれている非 POSIX システムでは、`fromtimestamp()` はうるう秒を無視します。このため、秒の異なる二つのタイムスタンプが同一の `datetime` オブジェクトとなることが起こり得ます。`utcfromtimestamp()` も参照してください。

#### **`utcfromtimestamp(timestamp)`**

`time.time()` が返すような POSIX タイムスタンプに対応する、UTC での `datetime` オブジェクトを返します。タイムスタンプがプラットフォームにおける C 関数 `localtime()` でサポートされている範囲を超えている場合には `ValueError` を送出することがあります。この値はよく 1970 年から 2038 年に制限されていることがあります。`fromtimestamp()` も参照してください。

#### **`fromordinal(ordinal)`**

1 年 1 月 1 日を序数 1 とする予測的グレゴリ暦序数に対応する `datetime` オブジェクトを返します。`1 <= ordinal <= datetime.max.toordinal()` でないかぎり `ValueError` が送出されます。結果として返されるオブジェクトの時間、分、秒、およびマイクロ秒はすべて 0 となり、`tzinfo` は `None` となります。

#### **`combine(date, time)`**

与えられた `date` オブジェクトと同じデータメンバを持ち、時刻と `tzinfo` メンバが与えられた `time` オブジェクトと等しい、新たな `datetime` オブジェクトを返します。任意の `datetime` オブジェクト `d` について、`d == datetime.combine(d.date(), d.timetz())` となります。`date` が `datetime` オブジェクトの場合、その時刻と `tzinfo` は無視されます。

以下にクラス属性を示します:

#### **`min`**

表現できる最も古い `datetime` で、`datetime(MINYEAR, 1, 1, tzinfo=None)` です。

#### **`max`**

表現できる最も新しい `datetime` で、`datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None)` です。

#### **`resolution`**

等しくない `datetime` オブジェクト間の最小の差で、`timedelta(microseconds=1)` です。

以下に (読み出し専用の) インスタンス属性を示します:

#### **`year`**

両端値を含む `MINYEAR` から `MAXYEAR` までの値です。

#### **`month`**

両端値を含む 1 から 12 までの値です。

#### **`day`**

1 から与えられた月と年における日数までの値です。

#### **`hour`**

`range(24)` 内の値です。

#### **`minute`**

`range(60)` 内の値です。

#### **`second`**

`range(60)` 内の値です。

## microsecond

range(1000000) 内の値です。

## tzinfo

datetime コンストラクタに *tzinfo* 引数として与えられたオブジェクトになり、何も渡されなかった場合には None になります。

以下にサポートされている演算を示します:

演算	結果
$datetime2 = datetime1 + timedelta$	(1)
$datetime2 = datetime1 - timedelta$	(2)
$timedelta = datetime1 - datetime2$	(3)
$datetime1 < datetime2$	datetime を datetime と比較します。(4)

(1) *datetime2* は *datetime1* から時間 *timedelta* 移動したもので、*timedelta.days* > 0 の場合進む方向に、*timedelta.days* < 0 の場合戻る方向に移動します。結果は入力 of *datetime* と同じ *tzinfo* を持ち、演算後には *datetime2* - *datetime1* == *timedelta* となります。*datetime2.year* が MINYEAR よりも小さいか、MAXYEAR より大きい場合には *OverflowError* が送出されます。入力が aware なオブジェクトの場合でもタイムゾーン修正は全く行われません。

(2) *datetime2* + *timedelta* == *datetime1* となるような *datetime2* を計算します。ちなみに、結果は入力 of *datetime* と同じ *tzinfo* メンバを持ち、入力が aware でもタイムゾーン修正は全く行われません。この操作は *date1* + (-*timedelta*) と等価ではありません。なぜならば、*date1* - *timedelta* がオーバーフローしない場合でも、-*timedelta* 単体がオーバーフローする可能性があるからです。

(3) *datetime* から *datetime* の減算は両方の被演算子が naive であるか、両方とも aware である場合にのみ定義されています片方が aware でもう一方が naive の場合、*TypeError* が送出されます。

両方とも naive か、両方とも aware で同じ *tzinfo* メンバを持つ場合、*tzinfo* メンバは無視され、結果は *datetime2* + *t* == *datetime1* であるような *timedelta* オブジェクト *t* となります。この場合タイムゾーン修正は全く行われません。

両方が aware で異なる *tzinfo* メンバを持つ場合、*a* - *b* は *a* および *b* をまず naive な UTC *datetime* オブジェクトに変換したかのようにして行います。演算結果は決してオーバーフローを起こさないことを除き、(*a*.replace(*tzinfo*=None) - *a*.utcoffset()) - (*b*.replace(*tzinfo*=None) - *b*.utcoffset()) と同じになります。

(4) *datetime1* が時刻として *datetime2* よりも前を表す場合に、*datetime1* は *datetime2* よりも小さいと見なされます。

被演算子の片方が naive でもう一方が aware の場合、*TypeError* が送出されます。両方の被演算子が aware で、同じ *tzinfo* メンバを持つ場合、共通の *tzinfo* メンバは無視され、基本の *datetime* 間の比較が行われます。両方の被演算子が aware で異なる *tzinfo* メンバを持つ場合、被演算子はまず (*self*.utcoffset() で得られる) UTC オフセットで修正されます。注意: 型混合の比較がデフォルトのオブジェクトアドレス比較となってしまうのを抑止するために、被演算子のもう一方が *datetime* オブジェクトと異なる型のオブジェクトの場合には *TypeError* が送出されます。しかしながら、被比較演算子のもう一方が *timetuple* 属性を持つ場合には *NotImplemented* が返されます。このフックにより、他種の日付オブジェクトに型混合比較を実装するチャンスを与えています。そうでない場合、*datetime* オブジェクトと異なる型のオブジェクトが比較されると、比較演算子が == または != でないかぎり *TypeError* が送出されます。後者の場合、それぞれ False または True を返します。

`datetime` オブジェクトは辞書のキーとして用いることができます。ブール演算コンテキストでは、全ての `datetime` オブジェクトは真であるとみなされます。

インスタンスメソッドを以下に示します:

**`date()`**

同じ年、月、日の `date` オブジェクトを返します。

**`time()`**

同じ時、分、秒、マイクロ秒を持つ `time` オブジェクトを返します。`tzinfo` は `None` です。`timetz()` も参照してください。

**`timetz()`**

同じ時、分、秒、マイクロ秒、および `tzinfo` メンバを持つ `time` オブジェクトを返します。`time()` メソッドも参照してください。

**`replace(year=, month=, day=, hour=, minute=, second=, microsecond=, tzinfo=)`**

キーワード引数で指定したメンバの値を除き、同じ値をもつ `datetime` オブジェクトを返します。メンバに対する変換を行わずに aware な `datetime` オブジェクトから naive な `datetime` オブジェクトを生成するために、`tzinfo=None` を指定することもできます。

**`astimezone(tz)`**

`datetime` オブジェクトを返します。返されるオブジェクトは新たな `tzinfo` メンバ `tz` を持ちます。`tz` は日付および時刻を調整して、オブジェクトが `self` と同じ UTC 時刻を持つが、`tz` におけるローカルな時刻を表すようにします。

`tz` は `tzinfo` のサブクラスのインスタンスでなければならず、インスタンスの `utcoffset()` および `dst()` メソッドは `None` を返してはなりません。`self` は aware でなくてはなりません (`self.tzinfo` が `None` であってはならず、かつ `self.utcoffset()` は `None` を返してはなりません)。

`self.tzinfo` が `tz` の場合、`self.astimezone(tz)` は `self` に等しくなります: 日付および時刻データメンバに対する調整は行われません。そうでない場合、結果はタイムゾーン `tz` におけるローカル時刻で、`self` と同じ UTC 時刻を表すようになります: `astz = dt.astimezone(tz)` とした後、`astz - astz.utcoffset()` は通常 `dt - dt.utcoffset()` と同じ日付および時刻データメンバを持ちます。`tzinfo` クラスに関する議論では、夏時間 (Daylight Saving time) の遷移境界では上の等価性が成り立たないことを説明しています (`tz` が標準時と夏時間の両方をモデル化している場合のみの問題です)。

単にタイムゾーンオブジェクト `tz` を `datetime` オブジェクト `dt` に追加したいだけで、日付や時刻データメンバへの調整を行わないのなら、`dt.replace(tzinfo=tz)` を使ってください。単に aware な `datetime` オブジェクト `dt` からタイムゾーンオブジェクトを除去したいだけで、日付や時刻データメンバの変換を行わないのなら、`dt.replace(tzinfo=None)` を使ってください。

デフォルトの `tzinfo.fromutc()` メソッドを `tzinfo` のサブクラスで上書きして、`astimezone()` が返す結果に影響を及ぼすことができます。エラーの場合を無視すると、`astimezone()` は以下のように動作します:

```
def astimezone(self, tz):
    if self.tzinfo is tz:
        return self
    # Convert self to UTC, and attach the new time zone object.
    utc = (self - self.utcoffset()).replace(tzinfo=tz)
    # Convert from UTC to tz's local time.
    return tz.fromutc(utc)
```

**`utcoffset()`**

`tzinfo` が `None` の場合、`None` を返し、そうでない場合には `self.tzinfo.utcoffset(self)` を返します。後者の式が `None` か、1 日以下の大きさを持つ経過時間を表す `timedelta` オブジェクトのいずれかを返さない場合には例外を送出します。

#### `dst()`

`tzinfo` が `None` の場合、`None` を返し、そうでない場合には `self.tzinfo.dst(self)` を返します。後者の式が `None` か、1 日以下の大きさを持つ経過時間を表す `timedelta` オブジェクトのいずれかを返さない場合には例外を送出します。

#### `tzname()`

`tzinfo` が `None` の場合、`None` を返し、そうでない場合には `self.tzinfo.tzname(self)` を返します。後者の式が `None` か文字列オブジェクトのいずれかを返さない場合には例外を送出します。

#### `timetuple()`

`time.localtime()` が返す形式の `time.struct_time` を返します。`d.timetuple()` は `time.struct_time((d.year, d.month, d.day, d.hour, d.minute, d.second, d.weekday(), d.toordinal() - date(d.year, 1, 1).toordinal() + 1, dst))` と等価です。返されるタプルの `tm_isdst` フラグは `dst()` メソッドに従って設定されます: `tzinfo` が `None` か `dst()` が `None` を返す場合、`tm_isdst` は `-1` に設定されます; そうでない場合、`dst()` がゼロでない値を返すと、`tm_isdst` は `1` となります; それ以外の場合には `tm_isdst` は `0` に設定されます。

#### `utctimetuple()`

`datetime` インスタンス `d` が naive の場合、このメソッドは `d.timetuple()` と同じであり、`d.dst()` の返す内容にかかわらず `tm_isdst` が `0` に強制される点だけが異なります。DST が UTC 時刻に影響を及ぼすことは決してありません。

`d` が aware の場合、`d` から `d.utcoffset()` が差し引かれて UTC 時刻に正規化され、正規化された時刻の `time.struct_time` を返します。`tm_isdst` は `0` に強制されます。`d.year` が `MINYEAR` や `MAXYEAR` で、UTC への修正の結果表現可能な年の境界を越えた場合には、戻り値の `tm_year` メンバは `MINYEAR-1` または `MAXYEAR+1` になることがあります。

#### `toordinal()`

予測的グレゴリ暦における日付序数を返します。`self.date().toordinal()` と同じです。

#### `weekday()`

月曜日を `0`、日曜日を `6` として、曜日を整数で返します。`self.date().weekday()` と同じです。`isoweekday()` も参照してください。

#### `isoweekday()`

月曜日を `1`、日曜日を `7` として、曜日を整数で返します。`self.date().isoweekday()` と等価です。`weekday()`、`isocalendar()` も参照してください。

#### `isocalendar()`

3 要素のタプル (ISO 年、ISO 週番号、ISO 曜日) を返します。`self.date().isocalendar()` と等価です。

#### `isoformat(sep='T')`

日付と時刻を ISO 8601 形式、すなわち `YYYY-MM-DDTHH:MM:SS.mmmmmmm` か、`microsecond` が `0` の場合には `YYYY-MM-DDTHH:MM:SS` で表した文字列を返します。`utcoffset()` が `None` を返さない場合、UTC からのオフセットを時間と分を表した (符号付きの) 6 文字からなる文字列が追加されます: すなわち、`YYYY-MM-DDTHH:MM:SS.mmmmmmm+HH:MM` となるか、`microsecond` が `0` の場合には `YYYY-MM-DDTHH:MM:SS+HH:MM` となります。オプションの引数 `sep` (デフォルトでは `'T'` です) は 1 文字のセパレータで、結果の文字列の日付と時刻の間に置かれます。例えば、

```
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     def utcoffset(self, dt): return timedelta(minutes=-399)
...
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat(' ')
'2002-12-25 00:00:00-06:39'
```

となります。

**\_\_str\_\_()**

datetime オブジェクト *d* において、`str(d)` は *d.isoformat(' ')* と等価です。

**ctime()**

日付を表す文字列を、例えば `datetime(2002, 12, 4, 20, 30, 40).ctime() == 'Wed Dec 4 20:30:40 2002'` のようにして返します。ネイティブの C 関数 `ctime()` (`time.ctime()` はこの関数を呼び出しますが、`datetime.ctime()` は呼び出しません) が C 標準に準拠しているプラットフォームでは、*d.ctime()* は `time.ctime(time.mktime(d.timetuple()))` と等価です。

**strftime(format)**

明示的な書式化文字列で制御された、日付を表現する文字列を返します。`strftime()` のふるまいについてのセクションを参照してください。

## 6.9.5 time オブジェクト

`time` オブジェクトは (ローカルの) 日中時刻を表現します。この時刻表現は特定の日の影響を受けず、`tzinfo` オブジェクトを介した修正の対象となります。

**class time(hour=0, minute=0, second=0, microsecond=0, tzinfo=None)**

全ての引数はオプションです。*tzinfo* は `None` または `tzinfo` クラスのサブクラスのインスタンスにすることができます。残りの引数は整数または長整数で、以下のような範囲に入ります:

- `0 <= hour < 24`
- `0 <= minute < 60`
- `0 <= second < 60`
- `0 <= microsecond < 1000000`.

引数がこれらの範囲外にある場合、`ValueError` が送出されます。

以下にクラス属性を示します:

**min**

表現できる最も古い datetime で、`time(0, 0, 0, 0)` です。The earliest representable time, `time(0, 0, 0, 0)`.

**max**

表現できる最も新しい datetime で、`time(23, 59, 59, 999999, tzinfo=None)` です。

**resolution**

等しくない datetime オブジェクト間の最小の差で、`timedelta(microseconds=1)` ですが、`time` オブジェクト間の四則演算はサポートされていないので注意してください。

以下に (読み出し専用の) インスタンス属性を示します:

**hour**

`range(24)` 内の値です。

**minute**

range(60) 内の値です。

**second**

range(60) 内の値です。

**microsecond**

range(1000000) 内の値です。

**tzinfo**

time コンストラクタに *tzinfo* 引数として与えられたオブジェクトになり、何も渡されなかった場合には None になります。

以下にサポートされている操作を示します:

- time と time の比較では、*a* が時刻として *b* よりも前を表す場合に *a* は *b* よりも小さいと見なされます。被演算子の片方が naive でもう一方が aware の場合、TypeError が送出されます。両方の被演算子が aware で、同じ tzinfo メンバを持つ場合、共通の tzinfo メンバは無視され、基本の datetime 間の比較が行われます。両方の被演算子が aware で異なる tzinfo メンバを持つ場合、被演算子はまず (self.utcoffset() で得られる) UTC オフセットで修正されます。型混合の比較がデフォルトのオブジェクトアドレス比較となってしまうのを抑止するために、time オブジェクトが他の型のオブジェクトと比較された場合、比較演算子が == または != でないかぎり TypeError が送出されます。後者の場合、それぞれ False または True を返します。
- ハッシュ化、辞書のキーとしての利用
- 効率的な pickle 化
- ブール演算コンテキストでは、time オブジェクトは、分に変換し、utcoffset() (None を返した場合) に 0 を差し引いて変換した後の結果がゼロでない場合、かつそのときに限って真とみなされます。

以下にインスタンスメソッドを示します:

**replace()**

hour=, minute=, second=, microsecond=, tzinfo=) キーワード引数で指定したメンバの値を除き、同じ値をもつ time オブジェクトを返します。メンバに対する変換を行わずに aware な datetime オブジェクトから naive な time オブジェクトを生成するために、tzinfo=None を指定することもできます。

**isoformat()**

日付と時刻を ISO 8601 形式、すなわち HH:MM:SS.mmmmmmm か、microsecond が 0 の場合には HH:MM:SS で表した文字列を返します。utcoffset() が None を返さない場合、UTC からのオフセットを時間と分を表した (符号付きの) 6 文字からなる文字列が追加されます: すなわち、HH:MM:SS.mmmmmmm+HH:MM となるか、microsecond が 0 の場合には HH:MM:SS+HH:MM となります。

**\_\_str\_\_()**

time オブジェクト *t* において、str(*t*) は *t.isoformat()* と等価です。

**strftime(format)**

明示的な書式化文字列で制御された、日付を表現する文字列を返します。strftime() のふるまいについてのセクションを参照してください。

**utcoffset()**

tzinfo が None の場合、None を返し、そうでない場合には self.tzinfo.utcoffset(None) を返します。後者の式が None か、1 日以下の大きさを持つ経過時間を表す timedelta オブジェクトのいずれかを返さない場合には例外を送出します。



**dst()**

tzinfo が None の場合、None を返し、そうでない場合には `self.tzinfo.dst(None)` を返します。後者の式が None か、1 日以下の大きさを持つ経過時間を表す `timedelta` オブジェクトのいずれかを返さない場合には例外を送出します。

**tzname()**

tzinfo が None の場合、None を返し、そうでない場合には `self.tzinfo.tzname(None)` を返します。後者の式が None か文字列オブジェクトのいずれかを返さない場合には例外を送出します。

## 6.9.6 tzinfo オブジェクト

tzinfo は抽象基底クラスです。つまり、このクラスは直接インスタンス化して利用しません。具体的なサブクラスを導出し、(少なくとも) 利用したい `datetime` のメソッドが必要とする tzinfo の標準メソッドを実装してやる必要があります。datetime モジュールでは、tzinfo の具体的なサブクラスは何ら提供していません。

tzinfo (の具体的なサブクラス) のインスタンスは `datetime` および `time` オブジェクトのコンストラクタに渡すことができます。後者のオブジェクトでは、データメンバをローカル時刻におけるものとして見ており、tzinfo オブジェクトはローカル時刻の UTC からのオフセット、タイムゾーンの名前、DST オフセットを、渡された日付および時刻オブジェクトからの相対で示すためのメソッドを提供します。

pickle 化についての特殊な要求事項: tzinfo のサブクラスは引数なしで呼び出すことのできる `__init__` メソッドを持たねばなりません。そうでなければ、pickle 化することはできますがおそらく unpickle 化することはできないでしょう。これは技術的な側面からの要求であり、将来緩和されるかもしれません。

tzinfo の具体的なサブクラスでは、以下のメソッドを実装する必要があります。厳密にどのメソッドが必要なのは、aware な `datetime` オブジェクトがこのサブクラスのインスタンスをどのように使うかに依存します。不確かならば、単に全てを実装してください。

**utcoffset(self, dt)**

ローカル時間の UTC からのオフセットを、UTC から東向きを正とした分で返します。ローカル時間が UTC の西側にある場合、この値は負になります。このメソッドは UTC からのオフセットの総計を返すように意図されているので注意してください; 例えば、tzinfo オブジェクトがタイムゾーンと DST 修正の両方を表現する場合、`utcoffset()` はそれらの合計を返さなければなりません。UTC オフセットが未知である場合、None を返してください。そうでない場合には、返される値は -1439 から 1439 の両端を含む値 ( $1440 = 24 \times 60$ ; つまり、オフセットの大きさは 1 日より短くなくてはなりません) が分で指定された `timedelta` オブジェクトでなければなりません。ほとんどの `utcoffset()` 実装は、おそらく以下の二つのうちの一つに似たものになるでしょう:

```
return CONSTANT                # fixed-offset class
return CONSTANT + self.dst(dt) # daylight-aware class
```

`utcoffset()` が None を返さない場合、`dst()` も None を返してはなりません。

`utcoffset()` のデフォルトの実装は `NotImplementedError` を送じます。

**dst(self, dt)**

夏時間 (DST) 修正を、UTC から東向きを正とした分で返します。DST 情報が未知の場合、None が返されます。DST が有効でない場合には `timedelta(0)` を返します。DST が有効の場合、オフセットは `timedelta` オブジェクトで返します (詳細は `utcoffset()` を参照してください)。DST オフセットが利用可能な場合、この値は `utcoffset()` が返す UTC からのオフセットには既に加算されているため、DST を個別に取得する必要がない限り `dst()` を使って問い合わせる必要はないので注意してください。例えば、`datetime.timetuple()` は tzinfo メンバの `dst()` メソッドを呼ん

で `tm_isdst` フラグがセットされているかどうか判断し、`tzinfo.fromutc()` は `dst()` タイムゾーンを移動する際に DST による変更があるかどうかを調べます。

標準および夏時間の両方をモデル化している `tzinfo` サブクラスのインスタンス `tz` は以下の式:

$$tz.utcoffset(dt) - tz.dst(dt)$$

が、`dt.tzinfo==tz` 全ての `datetime` オブジェクト `dt` について常に同じ結果を返さなければならないという点で、一貫性を持っていなければなりません。正常に実装された `tzinfo` のサブクラスでは、この式はタイムゾーンにおける "標準オフセット (standard offset)" を表し、特定の日や時刻の事情ではなく地理的な位置にのみ依存してはなりません。`datetime.astimezone()` の実装はこの事実に依存していますが、違反を検出することができません; 正しく実装するのはプログラマの責任です。`tzinfo` のサブクラスでこれを保証することができない場合、`tzinfo.fromutc()` の実装をオーバーライドして、`astimezone()` に関わらず正しく動作するようにしてもかまいません。ほとんどの `dst()` 実装は、おそらく以下の二つのうちの一つに似たものになるでしょう:

```
return timedelta(0)    # a fixed-offset class:  doesn't account for DST

or

# Code to set dston and dstoff to the time zone's DST transition
# times based on the input dt.year, and expressed in standard local
# time.  Then

if dston <= dt.replace(tzinfo=None) < dstoff:
    return timedelta(hours=1)
else:
    return timedelta(0)
```

デフォルトの `dst()` 実装は `NotImplementedError` を送出します。

**tzname(*self*, *dt*)**

`datetime` オブジェクト `dt` に対応するタイムゾーン名を文字列で返します。`datetime` モジュールでは文字列名について何も定義しておらず、特に何かを意味するといった要求仕様もまったくありません。例えば、"GMT"、"UTC"、"-500"、"-5:00"、"EDT"、"US/Eastern"、"America/New York" は全て有効な応答となります。文字列名が未知の場合には `None` を返してください。`tzinfo` のサブクラスでは、特に、`tzinfo` クラスが夏時間について記述している場合のように、渡された `dt` の特定の値によって異なった名前を返したい場合があるため、文字列値ではなくメソッドとなっていることに注意してください。

デフォルトの `tzname()` 実装は `NotImplementedError` を送出します。

以下のメソッドは `datetime` や `time` オブジェクトにおいて、同名のメソッドが呼び出された際に応じて呼び出されます。`datetime` オブジェクトは自身を引数としてメソッドに渡し、`time` オブジェクトは引数として `None` をメソッドに渡します。従って、`tzinfo` のサブクラスにおけるメソッドは引数 `dt` が `None` の場合と、`datetime` の場合を受理するように用意しなければなりません。

`None` が渡された場合、最良の応答方法を決めるのはクラス設計者次第です。例えば、このクラスが `tzinfo` プロトコルと関係をもたないということを表明させたいければ、`None` が適切です。標準時のオフセットを見つける他の手段がない場合には、標準 UTC オフセットを返すために `utcoffset(None)` を使うとっと便利かもしれません。

`datetime` オブジェクトが `datetime` メソッドの応答として返された場合、`dt.tzinfo` は `self` と同じオブジェクトになります。ユーザが直接 `tzinfo` メソッドを呼び出さないかぎり、`tzinfo` メソッドは `dt.tzinfo` と `self` が同じであることに依存します。その結果 `tzinfo` メソッドは `dt` がローカル時間であると解釈するので、他のタイムゾーンでのオブジェクトの振る舞いについて心配する必要がありません。

`fromutc(self, dt)`

デフォルトの `datetime.astimezone()` 実装で呼び出されます。 `datetime.astimezone()` から呼ばれた場合、 `dt.tzinfo` は `self` であり、 `dt` の日付および時刻データメンバは UTC 時刻を表しているものとして見えます。 `fromutc()` の目的は、 `self` のローカル時刻に等しい `datetime` オブジェクトを返すことにより日付と時刻データメンバを修正することにあります。

ほとんどの `tzinfo` サブクラスではデフォルトの `fromutc()` 実装を問題なく継承できます。デフォルトの実装は、固定オフセットのタイムゾーンや、標準時と夏時間の両方について記述しているタイムゾーン、そして DST 移行時刻が年によって異なる場合でさえ、扱えるくらい強力なものです。デフォルトの `fromutc()` 実装が全ての場合に対して正しく扱うことができないような例は、標準時の (UTC からの) オフセットが引数として渡された特定の日や時刻に依存するもので、これは政治的な理由によって起きることがあります。デフォルトの `astimezone()` や `fromutc()` の実装は、結果が標準時オフセットの変化にまたがる何時間かの中にある場合、期待通りの結果を生成しないかもしれません。

エラーの場合のためのコードを除き、デフォルトの `fromutc()` の実装は以下のように動作します:

```
def fromutc(self, dt):
    # raise ValueError error if dt.tzinfo is not self
    dtoff = dt.utcoffset()
    dtdst = dt.dst()
    # raise ValueError if dtoff is None or dtdst is None
    delta = dtoff - dtdst # this is self's standard offset
    if delta:
        dt += delta # convert to standard local time
        dtdst = dt.dst()
        # raise ValueError if dtdst is None
    if dtdst:
        return dt + dtdst
    else:
        return dt
```

以下に `tzinfo` クラスの使用例を示します:

```
from datetime import tzinfo, timedelta, datetime

ZERO = timedelta(0)
HOURL = timedelta(hours=1)

# A UTC class.

class UTC(tzinfo):
    """UTC"""

    def utcoffset(self, dt):
        return ZERO

    def tzname(self, dt):
        return "UTC"

    def dst(self, dt):
        return ZERO

utc = UTC()

# A class building tzinfo objects for fixed-offset time zones.
# Note that FixedOffset(0, "UTC") is a different way to build a
# UTC tzinfo object.
```

```

class FixedOffset(tzinfo):
    """Fixed offset in minutes east from UTC."""

    def __init__(self, offset, name):
        self.__offset = timedelta(minutes = offset)
        self.__name = name

    def utcoffset(self, dt):
        return self.__offset

    def tzname(self, dt):
        return self.__name

    def dst(self, dt):
        return ZERO

# A class capturing the platform's idea of local time.

import time as _time

STDOFFSET = timedelta(seconds = -_time.timezone)
if _time.daylight:
    DSTOFFSET = timedelta(seconds = -_time.altzone)
else:
    DSTOFFSET = STDOFFSET

DSTDIFF = DSTOFFSET - STDOFFSET

class LocalTimezone(tzinfo):

    def utcoffset(self, dt):
        if self._isdst(dt):
            return DSTOFFSET
        else:
            return STDOFFSET

    def dst(self, dt):
        if self._isdst(dt):
            return DSTDIFF
        else:
            return ZERO

    def tzname(self, dt):
        return _time.tzname[self._isdst(dt)]

    def _isdst(self, dt):
        tt = (dt.year, dt.month, dt.day,
              dt.hour, dt.minute, dt.second,
              dt.weekday(), 0, -1)
        stamp = _time.mktime(tt)
        tt = _time.localtime(stamp)
        return tt.tm_isdst > 0

Local = LocalTimezone()

# A complete implementation of current DST rules for major US time zones.

def first_sunday_on_or_after(dt):
    days_to_go = 6 - dt.weekday()
    if days_to_go:
        dt += timedelta(days_to_go)
    return dt

```

```

# In the US, DST starts at 2am (standard time) on the first Sunday in April.
DSTSTART = datetime(1, 4, 1, 2)
# and ends at 2am (DST time; 1am standard time) on the last Sunday of Oct.
# which is the first Sunday on or after Oct 25.
DSTEND = datetime(1, 10, 25, 1)

class USTimeZone(tzinfo):

    def __init__(self, hours, reprname, stdname, dstname):
        self.stdoffset = timedelta(hours=hours)
        self.reprname = reprname
        self.stdname = stdname
        self.dstname = dstname

    def __repr__(self):
        return self.reprname

    def tzname(self, dt):
        if self.dst(dt):
            return self.dstname
        else:
            return self.stdname

    def utcoffset(self, dt):
        return self.stdoffset + self.dst(dt)

    def dst(self, dt):
        if dt is None or dt.tzinfo is None:
            # An exception may be sensible here, in one or both cases.
            # It depends on how you want to treat them. The default
            # fromutc() implementation (called by the default astimezone()
            # implementation) passes a datetime with dt.tzinfo is self.
            return ZERO
        assert dt.tzinfo is self

        # Find first Sunday in April & the last in October.
        start = first_sunday_on_or_after(DSTSTART.replace(year=dt.year))
        end = first_sunday_on_or_after(DSTEND.replace(year=dt.year))

        # Can't compare naive to aware objects, so strip the timezone from
        # dt first.
        if start <= dt.replace(tzinfo=None) < end:
            return HOUR
        else:
            return ZERO

    Eastern = USTimeZone(-5, "Eastern", "EST", "EDT")
    Central = USTimeZone(-6, "Central", "CST", "CDT")
    Mountain = USTimeZone(-7, "Mountain", "MST", "MDT")
    Pacific = USTimeZone(-8, "Pacific", "PST", "PDT")

```

標準時間 (standard time) および夏時間 (daylight time) の両方を記述している tzinfo のサブクラスでは、回避不能の難解な問題が年に 2 度あるので注意してください。具体的な例として、東部アメリカ時刻 (US Eastern, UTC -5000) を考えます。EDT は 4 月の最初の日曜日の 1:59 (EST) 以後に開始し、10 月の最後の日曜日の 1:59 (EDT) に終了します:

UTC	3:MM	4:MM	5:MM	6:MM	7:MM	8:MM
EST	22:MM	23:MM	0:MM	1:MM	2:MM	3:MM
EDT	23:MM	0:MM	1:MM	2:MM	3:MM	4:MM
start	22:MM	23:MM	0:MM	1:MM	3:MM	4:MM
end	23:MM	0:MM	1:MM	1:MM	2:MM	3:MM

DST の開始の際 ("start" の並び) ローカルの壁時計は 1:59 から 3:00 に飛びます。この日は 2:MM の形式をとる時刻は実際には無意味となります。従って、`astimezone(Eastern)` は DST が開始する日には `hour==2` となる結果を返すことはありません。 `astimezone()` がこのことを保証するようにするには、`tzinfo.dst()` メソッドは "失われた時間" (東部時刻における 2:MM) が夏時間に存在することを考えなければなりません。

DST が終了する際 ("end" の並び) では、問題はさらに悪化します: 1 時間の間、ローカルの壁時計ではつきりと時刻をいえなくなります: それは夏時間の最後の 1 時間です。東部時刻では、その日の UTC での 5:MM に夏時間は終了します。ローカルの壁時計は 1:59 (夏時間) から 1:00 (標準時) に再び巻き戻されます。ローカルの時刻における 1:MM はあいまいになります。 `astimezone()` は二つの UTC 時刻を同じローカルの時刻に対応付けることでローカルの時計の振る舞いをまねます。東部時刻の例では、5:MM および 6:MM の形式をとる UTC 時刻は両方とも、東部時刻に変換された際に 1:MM に対応づけられます。 `astimezone()` がこのことを保証するようにするには、`tzinfo.dst()` は "繰り返された時間" が標準時に存在することを考慮しなければなりません。このことは、例えばタイムゾーンの標準のローカルな時刻に DST への切り替え時刻を表現することで簡単に設定することができます。

このようなあいまいさを許容できないアプリケーションは、ハイブリッドな `tzinfo` サブクラスを使って問題を回避しなければなりません; UTC や、他のオフセットが固定された `tzinfo` のサブクラス (EST (-5 時間の固定オフセット) のみを表すクラスや、EDT (-4 時間の固定オフセット) のみを表すクラス) を使う限り、あいまいさは発生しません。

## 6.9.7 `strftime()` の振る舞い

`date`、`datetime`、および `time` オブジェクトは全て、明示的な書式化文字列でコントロールして時刻表現文字列を生成するための `strftime(format)` メソッドをサポートしています。大雑把にいうと、`d.strftime(fmt)` は `time` モジュールの `time.strftime(fmt, d.timetuple())` のように動作します。ただし全てのオブジェクトが `timetuple()` メソッドをサポートしているわけではありません。

`time` オブジェクトでは、年、月、日の値がないため、それらの書式化コードを使うことができません。無理矢理使った場合、年は 1900 に置き換えられ、月と日は 0 に置き換えられます。

`date` オブジェクトでは、時、分、秒の値がないため、それらの書式化コードを使うことができません。無理矢理使った場合、これらの値は 0 に置き換えられます。

`naive` オブジェクトでは、書式化コード `%z` および `%Z` は空文字列に置き換えられます。

`aware` オブジェクトでは以下ようになります:

`%z` `utcoffset()` は +HHMM あるいは -HHMM の形式をもった 5 文字の文字列に変換されます。HH は UTC オフセット時間を与える 2 桁の文字列で、MM は UTC オフセット分を与える 2 桁の文字列です。例えば、`utcoffset()` が `timedelta(hours=-3, minutes=-30)` を返した場合、`%z` は文字列 '-0330' に置き換わります。

`%Z` `tzname()` が `None` を返した場合、`%Z` は空文字列に置き換わります。そうでない場合、`%Z` は返された値に置き換わりますが、これは文字列でなければなりません。



Python はプラットフォームの C ライブラリから `strptime()` 関数を呼び出し、プラットフォーム間のバリエーションはよくあることなので、サポートされている書式化コードの全セットはプラットフォーム間で異なります。Python の `time` モジュールのドキュメントでは、C 標準 (1989 年版) が要求する書式化コードをリストしており、これらのコードは標準 C 準拠の実装がなされたプラットフォームでは全て動作します。1999 年版の C 標準では書式化コードが追加されているので注意してください。

`strptime()` が正しく動作する年の厳密な範囲はプラットフォーム間で異なります。プラットフォームに関わらず、1900 年以前の年は使うことができません。

## 6.10 time — 時刻データへのアクセスと変換

このモジュールでは、時刻に関するさまざまな関数を提供します。ほとんどの関数が利用可能ですが、全ての関数が全てのプラットフォームで利用可能なわけではありません。このモジュールで定義されているほとんどの関数は、プラットフォーム上の同名の C ライブラリ関数を呼び出します。これらの関数に対する意味付けはプラットフォーム間で異なるため、プラットフォーム提供のドキュメントを読んでおくと便利でしょう。

まずいくつかの用語の説明と慣習について整理します。

- エポック (*epoch*) は、時刻の計測がはじまった時点のことです。その年の 1 月 1 日の午前 0 時に“エポックからの経過時間”が 0 になるように設定されます。UNIX ではエポックは 1970 年です。エポックがどうなっているかを知るには、`gmtime(0)` の値を見るとよいでしょう。
- このモジュールの中の関数は、エポック以前あるいは遠い未来の日付や時刻を扱うことができません。将来カットオフ (関数が正しく日付や時刻を扱えなくなる) が起きる時点は、C ライブラリによって決まります。UNIX ではカットオフは通常 2038 です。
- **2000 年問題 (Y2K):** Python はプラットフォームの C ライブラリに依存しています。C ライブラリは日付および時刻をエポックからの経過秒で表現するので、一般的に 2000 年問題を持ちません。時刻を表現する `struct_time` (下記を参照してください) を入力として受け取る関数は一般的に 4 桁表記の西暦年を要求します。以前のバージョンとの互換性のために、モジュール変数 `accept2dyear` がゼロでない整数の場合、2 桁の西暦年をサポートします。この変数の初期値は環境変数 `PYTHONY2K` が空文字列のとき 1 に設定されます。空文字列でない文字列が設定されている場合、0 に設定されます。こうして、`PYTHONY2K` を空文字列でない文字列に設定することで、西暦年の入力がすべて 4 桁の西暦年でなければならないようにすることができます。2 桁の西暦年が入力された場合には、POSIX または X/Open 標準に従って変換されます: 69-99 の西暦年は 1969-1999 となり、0-68 の西暦年は 2000-2068 になります。100-1899 は常に不正な値になります。この仕様は Python 1.5.2(a2) から新たに追加された機能であることに注意してください; それ以前のバージョン、すなわち Python 1.5.1 および 1.5.2a1 では、1900 以下の年に対して 1900 を足します。
- UTC は協定世界時 (Coordinated Universal Time) のことです (以前はグリニッジ標準時または GMT として知られていました)。UTC の頭文字の並びは誤りではなく、英仏の妥協によるものです。
- DST は夏時間 (Daylight Saving Time) のことで、一年のうち部分的に 1 時間タイムゾーンを修正することです。DST のルールは不可思議で (局所的な法律で定められています)、年ごとに変わることもあります。C ライブラリはローカルルールを記したテーブルを持っており (柔軟に対応するため、たいていはシステムファイルから読み込まれます)、この点に関しては唯一の真実の知識の源です。
- 多くの現時刻を返す関数 (real-time functions) の精度は、値や引数を表現するのに使う単位から想像されるよりも低いかも知れません。例えば、ほとんどの UNIX システムで、クロックの一刹那 (ticks)

の精度は 1 秒の 50 から 100 分の 1 に過ぎません。また、Mac では時刻は秒きっかりのとき以外正確ではありません。

- 反対に、`time()` および `sleep()` は UNIX の同等の関数よりましな精度を持っています: 時刻は浮動小数点で表され、`time()` は可能なかぎり最も正確な時刻を (UNIX の `gettimeofday()` があればそれを使って) 返します。また `sleep()` にはゼロでない端数を与えることができます (UNIX の `select()` があれば、それを使って実装しています)。
- `gmtime()`、`localtime()`、`strptime()` が返す時刻値、 および `asctime()`、`mktime()`、`strftime()` に与える時刻値はどちらも 9 つの整数からなる配列です。

Index	Attribute	Values
0	tm_year	(例えば 1993)
1	tm_mon	[1,12] の間の数
2	tm_mday	[1,31] の間の数
3	tm_hour	[0,23] の間の数
4	tm_min	[0,59] の間の数
5	tm_sec	[0,61] の間の数 <code>strftime()</code> の説明にある (1) を読んで下さい
6	tm_wday	[0,6] の間の数、月曜が 0 になります
7	tm_yday	[1,366] の間の数
8	tm_isdst	0, 1 または -1; 以下を参照してください

C の構造体と違って、月の値が 0-11 でなく 1-12 であることに注意してください。西暦年の値は上の ”2000 年問題 (Y2K)” で述べたように扱われます。夏時間フラグを -1 にして `mktime()` に渡すと、たいいていは正確な夏時間の状態を実現します。

`struct_time` を引数とする関数に正しくない長さの `struct_time` や要素の型が正しくない `struct_time` を与えた場合には、`TypeError` が送出されます。

2.2 で変更された仕様: 時刻値の配列はタプルから `struct_time` に変更され、それぞれのフィールドに属性名がつけられました。

このモジュールでは以下の関数とデータ型を定義します:

#### `accept2dyear`

2 桁の西暦年を使えるかを指定するブール型の値です。標準では真ですが、環境変数 `PYTHONY2K` が空文字列でない値に設定されている場合には偽になります。実行時に変更することもできます。

#### `altzone`

ローカルの夏時間タイムゾーンにおける UTC からの時刻オフセットで、西に行くほど増加する秒で表した値です (ほとんどの西ヨーロッパでは負になり、アメリカでは正、イギリスではゼロになります)。`daylight` がゼロでないときのみ使用してください。

#### `asctime([t])`

`gmtime()` や `localtime()` が返す時刻を表現するタプル又は `struct_time` を、`'Sun Jun 20 23:21:05 1993'` といった書式の 24 文字の文字列に変換します。`t` が与えられていない場合には、`localtime()` が返す現在の時刻が使われます。`asctime()` はロケール情報を使いません。注意: 同名の C の関数と違って、末尾には改行文字はありません。2.1 で変更された仕様: `tuple` を省略できるようになりました。

#### `clock()`

UNIX では、現在のプロセッサ時間秒を浮動小数点数で返します。時刻の精度および “プロセッサ時間 (processor time)” の定義そのものは同じ名前前の C 関数に依存します。いずれにせよ、この関数は Python のベンチマーク Python や計時アルゴリズムに使われています。

Windows では、最初にこの関数が呼び出されてからの経過時間を wall-clock 秒で返します。この関数は Win32 関数 `QueryPerformanceCounter()` に基づいていて、その精度は通常 1 マイクロ秒以下です。

**`ctime([secs])`**

エポックからの経過秒数で表現された時刻を、ローカルの時刻を表現する文字列に変換します。`secs` が与えられていない場合、`time()` が返す値が現在の時刻として使われます。`ctime(secs)` は `asctime(localtime(secs))` と同じです。`ctime()` はロケール情報を使いません。2.1 で変更された仕様: `secs` を省略できるようになりました。

**`daylight`**

DST タイムゾーンが定義されている場合ゼロでない値になります。

**`gmtime([secs])`**

エポックからの経過時間で表現された時刻を、UTC における `struct_time` に変換します。このとき `dst` フラグは常にゼロとして扱われます。`secs` が与えられていない場合、`time()` が返す値が現在の時刻として使われます。秒の端数は無視されます。`struct_time` のレイアウトについては上を参照してください。2.1 で変更された仕様: `secs` を省略できるようになりました

**`localtime([secs])`**

`gmtime()` に似ていますが、ローカルタイムに変換します。現在の時刻に DST が適用される場合、`dst` フラグは 1 に設定されます。2.1 で変更された仕様: `secs` を省略できるようになりました。

**`mktime(t)`**

`localtime()` の逆を行う関数です。引数は `struct_time` が完全な 9 つの要素全てに値の入ったタプル (`dst` フラグも必要です; 現在の時刻に DST が適用されるか不明の場合には -1 を使ってください) で、UTC ではなく ローカルの時刻を指定します。`time()` との互換性のために浮動小数点数の値を返します。入力の値が正しい時刻で表現できない場合、例外 `OverflowError` または `ValueError` が送出されます (どちらが送出されるかは Python および その下にある C ライブラリのどちらにとって無効な値が入力されたかで決まります)。この関数で生成できる最も昔の時刻値はプラットフォームに依存します。

**`sleep(secs)`**

与えられた秒数の間実行を停止します。より精度の高い実行停止時間を指定するために、引数は浮動小数点にしてもかまいません。何らかのシステムシグナルがキャッチされた場合、それに続いてシグナル処理ルーチンが実行され、`sleep()` を停止してしまいます。従って実際の実行停止時間は要求した時間よりも短くなるかもしれません。また、システムが他の処理をスケジューリングするため、実行停止時間が要求した時間よりも多少長い時間になることもあります。

**`strftime(format[, t])`**

`gmtime()` や `localtime()` が返す時刻値タプル又は `struct_time` を、`format` で指定した文字列形式に変換します。`t` が与えられていない場合、`localtime()` が返す現在の時刻が使われます。`format` は文字列でなくてはなりません。2.1 で変更された仕様: `t` を省略できるようになりました。

`format` 文字列には以下の指示語 (directive) を埋め込むことができます。これらはフィールド長や精度のオプションを付けずに表され、`strftime()` の結果の対応する文字列と入れ替えられます:

Directive	Meaning	Notes
%a	ロケールにおける省略形の曜日名。	(1)
%A	ロケールにおける省略なしの曜日名。	
%b	ロケールにおける省略形の月名。	
%B	ロケールにおける省略なしの月名。	
%c	ロケールにおける適切な日付および時刻表現。	
%d	月の始めから何日目かを表す 10 進数 [01,31]。	
%H	(24 時間計での) 時を表す 10 進数 [00,23]。	
%I	(12 時間計での) 時を表す 10 進数 [01,12]。	
%j	年の初めから何日目かを表す 10 進数 [001,366]。	
%m	月を表す 10 進数 [01,12]。	
%M	分を表す 10 進数 [00,59]。	
%p	ロケールにおける AM または PM に対応する文字列。	
%S	秒を表す 10 進数 [00,61]。	
%U	年の初めから何週目か (日曜を週の始まりとします) を表す 10 進数 [00,53]。年が明けてから最初の日曜日までの全ての曜日は 0 週目に属すると見なされます。	
%w	曜日を表す 10 進数 [0(日曜日),6]。	
%W	年の初めから何週目か (日曜を週の始まりとします) を表す 10 進数 [00,53]。年が明けてから最初の月曜日までの全ての曜日は 0 週目に属すると見なされます。	
%x	ロケールにおける適切な日付の表現。	
%X	ロケールにおける適切な時刻の表現。	
%Y	上 2 桁なしの西暦年を表す 10 進数 [00,99]。	
%Y	上 2 桁付きの西暦年を表す 10 進数。	
%Z	タイムゾーンの名前 (タイムゾーンがない場合には空文字列)。	
%%	文字 '%' 自体の表現。	

注意:

(1) 値の幅は間違いなく 0 to 61 です; これはうるう秒と、(ごく稀ですが) 2 重のうるう秒のためのものです。

以下に RFC 2822 インターネット電子メール標準で定義されている日付表現と互換の書式の例を示します。<sup>1</sup>

```
>>> from time import gmtime, strftime
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
'Thu, 28 Jun 2001 14:17:15 +0000'
```

いくつかのプラットフォームではさらにいくつかの指示語がサポートされていますが、標準 ANSI C で意味のある値はここで列挙したものだけです。

いくつかのプラットフォームでは、フィールドの幅や精度を指定するオプションが以下のように指示語の先頭の文字 '%' の直後に付けられるようになっていました; この機能も移植性はありません。フィールドの幅は通常 2 ですが、%j は例外で 3 です。

<sup>1</sup> 現在では %Z の利用は推奨されていません。しかしここで実現したい時間及び分オフセットへの展開を行ってくれる %Z エスケープは全ての ANSI C ライブラリでサポートされているわけではありません。また、オリジナルの 1982 年に提出された RFC 822 標準は西暦年の表現を 2 桁と要求しています (%Y でなく %y)。しかし実際には 2000 年になるだいが以前から 4 桁の西暦年表現に移行しています。4 桁の西暦年表現は RFC 2822 において義務付けられ、伴って RFC 822 での取り決めは撤廃されました。

**strptime**(*string* [, *format*])

時刻を表現する文字列をフォーマットに従って解釈します。返される値は `gmtime()` や `localtime()` が返すような `struct_time` です。*format* パラメタは `strptime()` で使うものと同じ指示語を使います; このパラメタの値はデフォルトでは "%a %b %d %H:%M:%S %Y" で、`ctime()` が返すフォーマットに一致します。*string* が *format* に従って解釈できなかった場合、例外 `ValueError` が送出されます。解析しようとする文字列が解析後に余分なデータを持っていた場合、`ValueError` が送出されます。欠落したデータはデフォルトの値で埋められ、その値は (1900, 1, 1, 0, 0, 0, 0, 1, -1) です。

%Z 指示語へのサポートは `tzname` に収められている値と `daylight` が真かどうかで決められます。このため、常に既知の (かつ夏時間でないと考えられている) UTC や GMT を認識する時意外はプラットフォーム固有の動作になります。

**struct\_time**

`gmtime()`、`localtime()` および `strptime()` が返す時刻値配列のタイプです。2.2 で追加された仕様です。

**time()**

時刻を浮動小数点数で返します。単位は UTC におけるエポックからの秒数です。時刻は常に浮動小数点で返されますが、全てのシステムが 1 秒より高い精度で時刻を提供するとは限らないので注意してください。この関数が返す値は通常減少していくことはありませんが、この関数を 2 回呼び出し、呼び出しの間にシステムクロックの時刻を巻き戻して設定した場合には、以前の呼び出しよりも低い値が返ることもあります。

**timezone**

(DST でない) ローカルタイムゾーンの UTC からの時刻オフセットで、西に行くほど増加する秒で表した値です (ほとんどの西ヨーロッパでは負になり、アメリカでは正、イギリスではゼロになります)。

**tzname**

二つの文字列からなるタプルです。最初の要素は DST でないローカルのタイムゾーン名です。ふたつめの要素は DST のタイムゾーンです。DST のタイムゾーンが定義されていない場合、二つ目の文字列を使うべきではありません。

**tzset()**

ライブラリで使われている時刻変換規則をリセットします。どのように行われるかは、環境変数 TZ で指定されます。2.3 で追加された仕様です。

利用できるシステム: UNIX。

注意: 多くの場合、環境変数 TZ を変更すると、`tzset` を呼ばない限り `localtime` のような関数の出力に影響を及ぼすため、値が信頼できなくなってしまう。

TZ 環境変数には空白文字を含めてはなりません。

環境変数 TZ の標準的な書式は以下です: (分かりやすいように空白を入れています)

`std offset [dst [offset[,start[/time], end[/time]]]]`

各値は以下のようにになっています:

`std` and `dst` 三文字またはそれ以上の英数字で、タイムゾーンの略称を与えます。この値は `time.tzname` になります。

`offset` オフセットは形式: `± hh[:mm[:ss]]` をとります。この表現は、UTC 時刻にするためにローカルな時間に加算する必要がある時間値を示します。'-' が先頭につく場合、そのタイムゾーンは本子



午線 (Prime Meridian) より東側にあります; それ以外の場合は本子午線の西側です。オフセットが dst の後ろに続かない場合、夏時間は標準時より一時間先行しているものと仮定します。

start[/time,end[/time]] いつ DST に移動し、DST から戻ってくるかを示します。開始および終了日時の形式は以下のいずれかです:

$Jn$  ユリウス日 (Julian day)  $n$  ( $1 \leq n \leq 365$ ) を表します。うるう日は計算に含められないため、2月28日は常に59で、3月1日は60になります。

$n$  ゼロから始まるユリウス日 ( $0 \leq n \leq 365$ ) です。うるう日は計算に含められるため、2月29日を参照することができます。

$Mm.n.dm$  月の第  $n$  週における  $d$  番目の日 ( $0 \leq d \leq 6, 1 \leq n \leq 5, 1 \leq m \leq 12$ ) を表します。週5は月における最終週の  $d$  番目の日を表し、第4週か第5週のどちらかになります。週1は日  $d$  が最初に現れる日を指します。日0は日曜日です。

時間はオフセットと同じで、先頭に符号 ('-' や '+') を付けてはいけないうところが違います。時刻が指定されていなければ、デフォルトの値 02:00:00 になります。

```
>>> os.environ['TZ'] = 'EST+05EDT,M4.1.0,M10.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'02:07:36 05/08/03 EDT'
>>> os.environ['TZ'] = 'AEST-10AEDT-11,M10.5.0,M3.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'16:08:12 05/08/03 AEST'
```

多くの Unix システム (\*BSD, Linux, Solaris, および Darwin を含む) では、システムの zoneinfo (*tzfile(5)*) データベースを使ったほうが、タイムゾーンごとの規則を指定する上で便利です。これを行うには、必要なタイムゾーンデータファイルへのパスをシステムの 'zoneinfo' タイムゾーンデータベースからの相対で表した値を環境変数 TZ に設定します。システムの 'zoneinfo' は通常 '/usr/share/zoneinfo' にあります。例えば、'US/Eastern'、'Australia/Melbourne'、'Egypt' ないし 'Europe/Amsterdam' と指定します。

```
>>> os.environ['TZ'] = 'US/Eastern'
>>> time.tzset()
>>> time.tzname
('EST', 'EDT')
>>> os.environ['TZ'] = 'Egypt'
>>> time.tzset()
>>> time.tzname
('EET', 'EEST')
```

#### 参考資料:

locale モジュール (6.26 節):

国際化サービス。ロケールの設定は time モジュールのいくつかの関数が返す値に影響をおよぼすことがあります。

calendar モジュール (5.17 節):

一般的なカレンダー関連の関数。timegm() はこのモジュールの gmtime() の逆の操作を行います。

## 6.11 sched — イベントスケジューラ

sched モジュールは一般的な目的のためのイベントスケジューラを実装するクラスを定義します:



**class scheduler**(*timefunc, delayfunc*)

`scheduler` クラスはイベントをスケジュールするための一般的なインターフェースを定義します。それは“外部世界”を実際に扱うための2つの関数を必要とします — *timefunc* は引数なしで呼出し可能であるべきで、そして数 (それは“time”です, どんな単位でもかまいません) を返すようにします。*delayfunc* は1つの引数 (*timefunc* の出力と互換) で呼出し可能であり、その時間だけ遅延しなければいけません。各々のイベントが、マルチスレッドアプリケーションの中で他のスレッドが実行する機会の許可を実行した後に、*delayfunc* は引数0で呼ばれるでしょう。

例:

```
>>> import sched, time
>>> s=sched.scheduler(time.time, time.sleep)
>>> def print_time(): print "From print_time", time.time()
...
>>> def print_some_times():
...     print time.time()
...     s.enter(5, 1, print_time, ())
...     s.enter(10, 1, print_time, ())
...     s.run()
...     print time.time()
...
>>> print_some_times()
930343690.257
From print_time 930343695.274
From print_time 930343700.273
930343700.276
```

### 6.11.1 スケジューラオブジェクト

`scheduler` インスタンスは以下のメソッドを持っています:

**enterabs**(*time, priority, action, argument*)

新しいイベントをスケジュールします。引数 *time* は、コンストラクタへ渡された *timefunc* の戻り値と互換な数値型でなければいけません。同じ *time* によってスケジュールされたイベントは、それらの *priority* によって実行されるでしょう。

イベントを実行することは、*action*(*\*argument*) を実行することを意味します。*argument* は *action* のためのパラメータを保持するシーケンスでなければいけません。

戻り値は、イベントのキャンセル後に使われるかもしれないイベントです (`cancel()` を見よ)。

**enter**(*delay, priority, action, argument*)

時間単位以上の *delay* でイベントをスケジュールします。そのとき、その他の関連時間、その他の引数、効果、戻り値は、`enterabs()` に対するものと同じです。

**cancel**(*event*)

キューからイベントを消去します。もし *event* がキューにある現在のイベントでないならば、このメソッドは `RuntimeError` を送出します。

**empty**()

もしイベントキューが空ならば、`True` を返します。

**run**()

すべてのスケジュールされたイベントを実行します。この関数は次のイベントを (コンストラクタへ渡された関数 *delayfunc* を使うことで) 待ち、そしてそれを実行し、イベントがスケジュールされなくなるまで同じことを繰り返します。

*action* あるいは *delayfunc* は例外を投げることができます。いずれの場合も、スケジューラは一貫した状態を維持し、例外を伝播するでしょう。例外が *action* によって投げられる場合、イベントは `run()` への呼出しを未来に行なわないでしょう。

イベントのシーケンスが、次イベントの前に、利用可能時間より実行時間が長いと、スケジューラは単に遅れることになるでしょう。イベントが落ちることはありません; 呼出しコードはもはや適切でないキャンセルイベントに対して責任があります。

## 6.12 mutex — 排他制御

`mutex` モジュールはロックの獲得と解除を通して排他制御を可能にするクラスを定義します。スレッドやマルチタスクを必要と (あるいは意味) しません、それらの目的に有益となることでしょう。

`mutex` モジュールは以下のクラスを定義します:

`class mutex()`

新しい (ロックされてない) `mutex` を作ります。

`mutex` は状態を 2 つ持ちます — “ロック” ビットとキューです。 `mutex` がロックされていないとき、キューは空です。 `mutex` がロックされているとき、キューは空、あるいはロックを獲得するために待っている関数 (またはメソッド) を示すペア (*function*, *argument*) を収容しています。キューが空でないときに `mutex` が解除されると、最初のキュー登録が削除されてその *function(argument)* が呼び出れ、それが今はロックを持ちます。

もちろんマルチスレッドの意味は全くなく、一度ロックを取得すると関数が呼ばれるので、`lock()` による風変わりなインターフェースとなりえます。

### 6.12.1 mutex オブジェクト

`mutex` は以下のメソッドを持ちます:

`test()`

`mutex` がロックされたか調べます。

`testandset()`

“アトミック” Test-and-Set 命令です。セットされていないならばロックを得て `True` を返し、そうでないならば `False` を返します。

`lock(function, argument)`

`mutex` がロックされていなければ、*function(argument)* を実行します。この場合にはその `mutex` がロックされて、関数と引数をキューに置きます。*function(argument)* が実行されたときの説明は、`unlock` を見てください。

`unlock()`

もしキューが空ならば、`mutex` を解除します。一方、空でないならば、キューの最初の要素を実行します。

## 6.13 getpass — 可搬性のあるパスワード入力機構

The `getpass` module provides two functions: `getpass` モジュールは二つの機能を提供します:

`getpass([prompt])`

エコーなしでユーザーにパスワードを入力させるプロンプト。ユーザーは *prompt* の文字列をプロンプトに使えます、デフォルトは `'Password: '` です。

利用できるシステム: Machintosh, Unix, Windows

`getuser()`

ユーザーの“ログイン名”を返します。有効性: UNIX、Windows

この関数は環境変数 LOGNAME USER LNAME USERNAME の順序でチェックして、最初の空ではない文字列が設定された値を返します。もし、なにも設定されていない場合は pwd モジュールが提供するシステム上のパスワードデータベースから返します。それ以外は、例外が上がります。

## 6.14 curses — 文字セル表示のための端末操作

1.6 で変更された仕様: ncurses ライブラリのサポートを追加し、パッケージに変換しました

curses モジュールは、可搬性のある端末操作を行うためのデファクトスタンダードである、curses ライブラリへのインタフェースを提供します。

UNIX 環境では curses は非常に広く用いられていますが、DOS、OS2、そしておそらく他のシステムのバージョンも利用することができます。この拡張モジュールは Linux および BSD 系の UNIX で動作するオープンソースの curses ライブラリである ncurses の API に合致するように設計されています。

参考資料:

curses.ascii モジュール (6.17 節):

ロケール設定に関わらず ASCII 文字を扱うためのユーティリティ。

curses.panel モジュール (6.18 節):

curses ウィンドウにデプス機能を追加するパネルスタック拡張。

curses.textpad モジュール (6.15 節):

Emacs ライクなキーバインディングをサポートする編集可能な curses 用テキストウィジェット。

curses.wrapper モジュール (6.16 節):

アプリケーションの起動時および終了時に適切な端末のセットアップとリセットを確実に行うための関数。

*Curses Programming with Python*

(<http://www.python.org/doc/howto/curses/curses.html>)

Andrew Kuchling および Eric Raymond によって書かれた、curses を Python で使うためのチュートリアルです。Python Web サイトで入手できます。

Python ソースコードの ‘Demo/curses/’ ディレクトリには、このモジュールで提供されている curses バインディングを使ったプログラム例がいくつか収められています。

### 6.14.1 関数

curses モジュールでは以下の例外を定義しています:

**exception error**

curses ライブラリ関数がエラーを返した際に送出される例外です。

注意: 関数やメソッドにおけるオプションの引数 *x* および *y* がある場合、標準の値は常に現在のカーソルになります。オプションの *attr* がある場合、標準の値は `A_NORMAL` です。

curses では以下の関数を定義しています:

`baudrate()`

端末の出力速度をビット / 秒で返します。ソフトウェア端末エミュレータの場合、これは固定の高い値を持つことになります。この関数は歴史的な理由で入れられています; かつては、この関数は時間

遅延を生成するための出力ループを書くために用いられ、行速度に応じてインタフェースを切り替えたりするために用いられ、していました。

**beep()**

注意を促す短い音を鳴らします。

**can\_change\_color()**

端末に表示される色をプログラマが変更できるか否かによって、真または偽を返します。

**cbreak()**

cbreak モードに入ります。cbreak モード (“rare” モードと呼ばれることもあります) では、通常の tty 行バッファリングはオフにされ、文字を一字一字読むことができます。ただし、raw モードとは異なり、特殊文字 (割り込み:interrupt、終了:quit、一時停止:suspend、およびフロー制御) については、tty ドライバおよび呼び出し側のプログラムに対する通常の効果をもっています。まず raw() を呼び出し、次いで cbreak() を呼び出すと、端末を cbreak モードにします。

**color\_content(color\_number)**

色 color\_number の赤、緑、および青 (RGB) 要素の強度を返します。color\_number は 0 から COLORS の間でなければなりません。与えられた色の R、G、B、の値からなる三要素のタプルが返されます。この値は 0 (その成分はない) から 1000 (その成分の最大強度) の範囲をとります。

**color\_pair(color\_number)**

指定された色の表示テキストにおける属性値を返します。属性値は A\_STANDOUT、A\_REVERSE、およびその他の A\_\* 属性と組み合わせられています。pair\_number() はこの関数の逆です。

**curs\_set(visibility)**

カーソルの状態を設定します。visibility は 0、1、または 2 に設定され、それぞれ不可視、通常、または非常に可視、を意味します。要求された可視属性を端末がサポートしている場合、以前のカーソル状態が返されます; そうでなければ例外が送出されます。多くの端末では、“可視 (通常)” モードは下線カーソルで、“非常に可視” モードはブロックカーソルです。

**def\_prog\_mode()**

現在の端末属性を、稼動中のプログラムが curses を使う際のモードである “プログラム” モードとして保存します。(このモードの反対は、プログラムが curses を使わない “シェル” モードです。) その後 reset\_prog\_mode() を呼ぶとこのモードを復旧します。

**def\_shell\_mode()**

現在の端末属性を、稼動中のプログラムが curses を使っていないときのモードである “シェル” モードとして保存します。(このモードの反対は、プログラムが curses 機能を利用している “プログラム” モードです。) その後 reset\_shell\_mode() を呼ぶとこのモードを復旧します。

**delay\_output(ms)**

出力に ms ミリ秒の一時停止を入れます。

**doupdate()**

物理スクリーン (physical screen) を更新します。curses ライブラリは、現在の物理スクリーンの内容と、次の状態として要求されている仮想スクリーンをそれぞれ表す、2 つのデータ構造を保持しています。doupdate() は更新を適用し、物理スクリーンを仮想スクリーンに一致させます。

仮想スクリーンは addstr() のような書き込み操作をウィンドウに行った後に noutrefresh() を呼び出して更新することができます。通常の refresh() 呼び出しは、単に noutrefresh() を呼んだ後に doupdate() を呼ぶだけです; 複数のウィンドウを更新しなければならない場合、全てのウィンドウに対して noutrefresh() を呼び出した後、一度だけ doupdate() を呼ぶことで、パフォーマンスを向上させることができ、おそらくスクリーンのちらつきも押さえることができます。

**echo()**

echo モードに入ります。echo モードでは、各文字入力はスクリーン上に入力された通りにエコーバックされます。

**endwin()**

ライブラリの非初期化を行い、端末を通常の状態に戻します。

**erasechar()**

ユーザの現在の消去文字 (erase character) 設定を返します。UNIX オペレーティングシステムでは、この値は curses プログラムが制御している端末の属性であり、curses ライブラリ自体では設定されません。

**filter()**

**filter()** ルーチンを使う場合、**initscr()** を呼ぶ前に呼び出さなくてはなりません。この手順のもとらす効果は以下の通りです: まず二つの関数の呼び出しの間は、**LINES** は 1 に設定されます; **clear**、**cup**、**cud**、**cudl**、**cuul**、**cuu**、**vpa** は無効化されます; **home** 文字列は **cr** の値に設定されます。これにより、カーソルは現在の行に制限されるので、スクリーンの更新も同様に制限されます。この関数は、スクリーンの他の部分に影響を及ぼさずに文字単位の行編集を行う場合に利用できます。

**flash()**

スクリーンをフラッシュ(flash) します。すなわち、画面を色反転 (reverse-video) にして、短時間でもとにもどします。人によっては、**beep()** で生成される可聴な注意音よりも、このような“可視ベル (visible bell)”を好みます。

**flushinp()**

全ての入力バッファをフラッシュします。この関数は、ユーザによってすでに入力されているが、まだプログラムによって処理されていない全ての先行入力文字 (typeahead) を捨て去ります。

**getmouse()**

**getch()** が **KEY\_MOUSE** を返してマウスイベントを通知した後、この関数を呼んで待ち行列 (queue) 上に置かれているマウスイベントを取得しなければなりません。イベントは (*id*, *x*, *y*, *z*, *bstate*) の 5 要素のタプルで表現されています。*id* は複数のデバイスを区別するための ID 値で、*x*、*y*、*z* はイベントの座標値です (現在 *z* は使われていません)。 *bstate* は整数値で、その各ビットはイベントのタイプを示す値に設定されています。この値は以下に示す定数のうち一つまたはそれ以上のビット単位 OR になっています。以下の定数の *n* は 1 から 4 のボタン番号を示します: **BUTTONn\_PRESSED**, **BUTTONn\_RELEASED**, **BUTTONn\_CLICKED**, **BUTTONn\_DOUBLE\_CLICKED**, **BUTTONn\_TRIPLE\_CLICKED**, **BUTTON\_SHIFT**, **BUTTON\_CTRL**, **BUTTON\_ALT**.

**getsyx()**

仮想スクリーンにおける現在のカーソル位置を *y* および *x* の順で返します。 **leaveok** が真に設定されていれば、-1、-1 が返されます。

**getwin(file)**

以前の **putwin()** 呼び出しでファイルに保存されている、ウィンドウ関連データを読み出します。次に、このルーチンはそのデータを使って新たなウィンドウを生成し初期化して、その新規ウィンドウオブジェクトを返します。

**has\_colors()**

端末が色表示を行える場合には真を返します。そうでない場合には偽を返します。

**has\_ic()**

端末が文字の挿入 / 削除機能を持つ場合に真を返します。この関数は、最近の端末エミュレータがどれもこの機能を持っているのと同じく、歴史的な理由だけのために含まれています。

**has\_il()**

端末が行の挿入 / 削除機能を持つか、領域単位のスクロールによって機能をシミュレートできる場合に真を返します。この関数は、最近の端末エミュレータがどれもこの機能を持っているのと同じく、



歴史的な理由だけのために含まれています。

**has\_key**(*ch*)

キー値 *ch* をとり、現在の端末タイプがその値のキーを認識できる場合に真を返します。

**halfdelay**(*tenths*)

半遅延モード、すなわち `cbreak` モードに似た、ユーザが打鍵した文字がすぐにプログラムで利用できるようになるモードで使われます。しかしながら、何も入力されなかった場合、*tenths* 十秒後に例外が送出されます。*tenths* の値は 1 から 255 の間でなければなりません。半遅延モードから抜けるには `nocbreak()` を使います。

**init\_color**(*color\_number*, *r*, *g*, *b*)

色の定義を変更します。変更したい色番号と、その後に 3 つ組みの RGB 値 (赤、緑、青の成分の大きさ) をとります。*color\_number* の値は 0 から `COLORS` の間でなければなりません。*r*、*g*、*b* の値は 0 から 1000 の間でなければなりません。`init_color()` を使うと、スクリーン上でカラーが使用されている部分は全て新しい設定に即時変更されます。この関数はほとんどの端末で何も行いません；`can_change_color()` が 1 を返す場合にのみ動作します。

**init\_pair**(*pair\_number*, *fg*, *bg*)

色ペアの定義を変更します。3 つの引数: 変更したい色ペア、前景色の色番号、背景色の色番号、をとります。*pair\_number* は 1 から `COLOR_PAIRS - 1` の間でなければなりません (0 色ペアは黒色背景に白色前景となるように設定されており、変更することができません)。*fg* および *bg* 引数は 0 と `COLORS` の間でなければなりません。色ペアが以前に初期化されていれば、スクリーンを更新して、指定された色ペアの部分を新たな設定に変更します。

**initscr**()

ライブラリを初期化します。スクリーン全体をあらわす `WindowObject` を返します。注意: 端末のオープン時にエラーが発生した場合、`curses` ライブラリによってインタープリタが終了される場合があります。

**isendwin**()

`endwin()` がすでに呼び出されている (すなわち、`curses` ライブラリが非初期化されている) 場合に真を返します。

**keyname**(*k*)

*k* に番号付けされているキーの名前を返します。印字可能な ASCII 文字を生成するキーの名前はそのキーの文字自体になります。コントロールキーと組み合わせたキーの名前は、キャレットの後ろに対応する ASCII 文字が続く 2 文字の文字列になります。Alt キーと組み合わせたキー (128-255) の名前は、先頭に 'M-' が付き、その後に対応する ASCII 文字が続く文字列になります。

**killchar**()

ユーザの現在の行削除文字を返します。UNIX オペレーティングシステムでは、この値は `curses` プログラムが制御している端末の属性であり、`curses` ライブラリ自体では設定されません。

**longname**()

現在の端末について記述している `terminfo` の長形式 `name` フィールドが入った文字列を返します。`verbose` 形式記述の最大長は 128 文字です。この値は `initscr()` 呼び出しの後でのみ定義されています。

**meta**(*yes*)

*yes* が 1 の場合、8 ビット文字を入力として許します。*yes* が 0 の場合、7 ビット文字だけを許します。

**mouseinterval**(*interval*)

ボタンが押されてから離されるまでの時間をマウスクリック一回として認識する最大の時間間隔を設定します。以前の内部設定値を返します。標準の値は 200 ミリ秒、または 5 分の 1 秒です。



**mousemask**(*mousemask*)

報告すべきマウスイベントを設定し、(*availmask*, *oldmask*) の組からなるタプルを返します。*availmask* はどの指定されたマウスイベントのどれが報告されるかを示します; どのイベント指定も完全に失敗した場合には 0 が返ります。*oldmask* は与えられたウィンドウの以前のマウスイベントマスクです。この関数が呼ばれない限り、マウスイベントは何も報告されません。

**napms**(*ms*)

*ms* ミリ秒スリープします。

**newpad**(*nlines*, *ncols*)

与えられた行とカラム数を持つパッド (pad) データ構造を生成し、そのポインタを返します。パッドはウィンドウオブジェクトとして返されます。

パッドはウィンドウと同じようなものですが、スクリーンのサイズによって制限されており、スクリーンの特定の部分に関連付けられていなくてもかまいません。大きなウィンドウが必要であり、スクリーンにはそのウィンドウの一部しか一度に表示しない場合に使うことができます。(スクロールや入力エコーなどによる) パッドに対する再描画は起こりません。パッドに対する `refresh()` および `noutrefresh()` メソッドは、パッド中の表示する部分と表示するために利用するスクリーン上の位置を指定する 6 つの引数が必要です。これらの引数は *pminrow*、*pmincol*、*sminrow*、*smincol*、*smaxrow*、*smaxcol* です; *p* で始まる引数はパッド中の表示領域の左上位置で、*s* で始まる引数はパッド領域を表示するスクリーン上のクリップ矩形を指定します。

**newwin**( [*nlines*, *ncols*, ] *begin\_y*, *begin\_x* )

左上の角が (*begin\_y*, *begin\_x*) で、高さ / 幅が *nlines/ncols* の新規ウィンドウを返します。

標準では、ウィンドウは指定された位置からスクリーンの右下まで広がります。

**nl**( )

`newline` モードに入ります。このモードはリターンキーを入力中の改行として変換し、出力時に改行文字を復帰 (return) と改行 (line-feed) に変換します。`newline` モードは初期化時にはオンになっています。

**nocbreak**( )

`cbreak` モードから離れます。行バッファリングを行う通常の “cooked” モードに戻ります。

**noecho**( )

`echo` モードから離れます。入力のエコーバックはオフにされます。

**nonl**( )

`newline` モードから離れます。入力時のリターンキーから改行への変換、および出力時の改行から復帰 / 改行への低レベル変換を無効化します (ただし、`addch( '\n' )` の振る舞いは変更せず、仮想スクリーン上では常に復帰と改行に等しくなります)。変換をオフにすることで、`curses` は水平方向の動きを少しだけ高速化できることがあります; また、入力中のリターンキーの検出ができるようになります。

**noqiflush**( )

`noqiflush` ルーチンを使うと、通常行われている `INTR`、`QUIT`、および `SUSP` 文字による入力および出力キューのフラッシュが行われなくなります。シグナルハンドラが終了した際、割り込みが発生しなかったかのように出力を続たい場合、ハンドラ中で `noqiflush()` を呼び出すことができます。

**noraw**( )

`raw` モードから離れます。行バッファリングを行う通常の “cooked” モードに戻ります。

**pair\_content**(*pair\_number*)

要求された色ペア中の色を含む (*fg*, *bg*) からなるタプルを返します。*pair\_number* は 0 から `COLOR_PAIRS - 1` の間でなければなりません。

**pair\_number**(*attr*)

*attr* に対する色ペアセットの番号を返します。color\_pair() はこの関数の逆に相当します。

**putp**(*string*)

tputs(*str*, 1, putchar) と等価です; 現在の端末における、指定された terminfo 機能の値を出力します。putp の出力は常に標準出力に送られるので注意して下さい。

**qiflush**( [*flag*] )

*flag* が偽なら、noqiflush() を呼ぶのと同じ効果です。*flag* が真か、引数を与えられていない場合、制御文字が読み出された後にキューはフラッシュされます。

**raw**( )

raw モードに入ります。raw モードでは、通常の行バッファリングと割り込み (interrupt)、終了 (quit)、一時停止 (suspend)、およびフロー制御キーはオフになります; 文字は curses 入力関数に一字ずつ渡されます。

**reset\_prog\_mode**( )

端末を “program” モードに復旧し、予め def\_prog\_mode() で保存した内容に戻します。

**reset\_shell\_mode**( )

端末を “shell” モードに復旧し、予め def\_shell\_mode() で保存した内容に戻します。

**setsyx**(*y*, *x*)

仮想スクリーンカーソルを *y*、*x* に設定します。*y* および *x* が共に -1 の場合、leaveok が設定されます。

**setupterm**( [*termstr*, *fd*] )

端末を初期化します。*termstr* は文字列で、端末の名前を与えます; 省略された場合、TERM 環境変数の値が使われます。*fd* は初期化シーケンスが送られる先のファイル記述子です; *fd* を与えない場合、sys.stdout のファイル記述子が使われます。

**start\_color**( )

プログラマがカラーを利用したい場合で、かつ他の何らかのカラー操作ルーチンを呼び出す前に呼び出さなくてはなりません。この関数は initscr() を呼んだ直後に呼び出すようにしておくといでしょう。

start\_color() は 8 つの基本色 (黒、赤、緑、黄、青、マゼンタ、シアン、および白) と、色数の最大値と端末がサポートする色ペアの最大数が入っている、curses モジュールにおける二つのグローバル変数、COLORS および COLOR\_PAIRS を初期化します。この関数はまた、色設定を端末のスイッチが入れられたときの状態に戻します。

**termattrs**( )

端末がサポートする全てのビデオ属性を論理和した値を返します。この情報は、curses プログラムがスクリーンの見え方を完全に制御する必要がある場合に便利です。

**termname**( )

14 文字以下になるように切り詰められた環境変数 TERM の値を返します。

**tigetflag**(*capname*)

terminfo 機能名 *capname* に対応する機能値をブール値で返します。*capname* がブール値で表される機能値でない場合 -1 が返され、機能がキャンセルされているか、端末記述上に見つからない場合には 0 を返します。

**tigetnum**(*capname*)

terminfo 機能名 *capname* に対応する機能値を数値で返します。*capname* が数値で表される機能値でない場合 -2 が返され、機能がキャンセルされているか、端末記述上に見つからない場合には -1 を返します。

**tigetstr**(*capname*)

terminfo 機能名 *capname* に対応する機能値を文字列値で返します。*capname* が文字列値で表される機能値でない場合や、機能がキャンセルされているか、端末記述上に見つからない場合には `None` を返します。

`tparm(str[,...])`

*str* を与えられたパラメタを使って文字列にインスタンス化します。*str* は terminfo データベースから得られたパラメタを持つ文字列でなければなりません。例えば、`tparm(tigetstr("cup"), 5, 3)` は `'\033[6;4H'` のようになります。厳密には端末の形式によって異なる結果となります。

`typeahead(fd)`

先読みチェックに使うためのファイル記述子 *fd* を指定します。*fd* が `-1` の場合、先読みチェックは行われません。

curses ライブラリはスクリーンを更新する間、先読み文字列を定期的に検索することで“行はみ出し最適化 (line-breakout optimization)”を行います。入力を得られ、かつ入力は端末からのものである場合、現在行おうとしている更新は `refresh` や `doupdate` を再度呼び出すまで先送りにします。この関数は異なるファイル記述子で先読みチェックを行うように指定することができます。

`unctrl(ch)`

*ch* の印字可能な表現を文字列で返します。制御文字は例えば `^C` のようにキャレットに続く文字として表示されます。印字可能文字はそのままです。

`ungetch(ch)`

*ch* をプッシュして、`getch()` を次に呼び出したときに返されるようにします。注意: `getch()` を呼び出すまでは *ch* は一つしかプッシュできません。

`ungetmouse(id, x, y, z, bstate)`

与えられた状態データが関連付けられた `KEY_MOUSE` イベントを入力キューにプッシュします。

`use_env(flag)`

この関数を使う場合、`initscr()` または `newterm` を呼ぶ前に呼び出さなくてはなりません。*flag* が偽の場合、環境変数 `LINES` および `COLUMNS` の値 (これらは標準の設定で使われます) の値が設定されていたり、curses がウィンドウ内で動作して (この場合 `LINES` や `COLUMNS` が設定されていないとウィンドウのサイズを使います) いても、terminfo データベースに指定された `lines` および `columns` の値を使います。

## 6.14.2 Window オブジェクト

上記の `initscr()` や `newwin()` が返すウィンドウは、以下のメソッドを持ちます:

`addch([y, x], ch[, attr])`

注意: ここで文字は Python 文字 (長さ 1 の文字列) *C* における文字 (ASCII コード) を意味します。(この注釈は文字について触れているドキュメントではどこでも当てはまります。) 組み込みの `ord()` は文字列をコードの集まりにする際に便利です。

(*y*, *x*) にある文字 *ch* を属性 *attr* で描画します。このときその場所に以前描画された文字は上書きされます。標準の設定では、文字の位置および属性はウィンドウオブジェクトにおける現在の設定になります。

`addnstr([y, x], str, n[, attr])`

文字列 *str* から最大で *n* 文字を (*y*, *x*) に属性 *attr* で描画します。以前ディスプレイにあった内容はすべて上書きされます。

`addstr([y, x], str[, attr])`

(*y*, *x*) に文字列 *str* を属性 *attr* で描画します。以前ディスプレイにあった内容はすべて上書きされます。

**attroff**(*attr*)

現在のウィンドウに書き込まれた全ての内容に対し“バックグラウンド”に設定された属性 *attr* を除去します。

**attron**(*attr*)

現在のウィンドウに書き込まれた全ての内容に対し“バックグラウンド”に属性 *attr* を追加します。

**attrset**(*attr*)

“バックグラウンド”の属性セットを *attr* に設定します。初期値は 0 (属性なし) です。

**bkgd**(*ch* [, *attr*])

ウィンドウ上の背景プロパティを、*attr* を属性とする文字 *ch* に設定します。変更はそのウィンドウ中の全ての文字に以下のようにして適用されます:

- ウィンドウ中の全ての文字の属性が新たな背景属性に変更されます。
- 以前の背景文字が出現すると、常に新たな背景文字に変更されます。

**bkgdset**(*ch* [, *attr*])

ウィンドウの背景を設定します。ウィンドウの背景は、文字と何らかの属性の組み合わせから成り立ちます。背景情報の属性の部分は、ウィンドウ上に描画されている空白でない全ての文字と組み合わせ (OR され) ます。空白文字には文字部分と属性部分の両方が組み合わせられます。背景は文字のプロパティとなり、スクロールや行 / 文字の挿入 / 削除操作の際には文字と一緒に移動します。

**border**( [*ls* [, *rs* [, *ts* [, *bs* [, *tl* [, *tr* [, *bl* [, *br*]]]]]]]] )

ウィンドウの縁に境界線を描画します。各引数には境界の特定部分を表現するために使われる文字を指定します; 詳細は以下のテーブルを参照してください。文字は整数または 1 文字からなる文字列で指定されます。

注意: どの引数も、0 を指定した場合標準設定の文字が使われるようになります。キーワード引数は使うことができません。標準の設定はテーブル中に示されています:

引数	記述	標準の設定値
<i>ls</i>	左側	ACS_VLINE
<i>rs</i>	右側	ACS_VLINE
<i>ts</i>	上側	ACS_HLINE
<i>bs</i>	下側	ACS_HLINE
<i>tl</i>	左上の角	ACS_ULCORNER
<i>tr</i>	右上の角	ACS_URCORNER
<i>bl</i>	左下の角	ACS_BLCORNER
<i>br</i>	右下の角	ACS_BRCORNER

**box**( [*vertch*, *horch*])

**border**() と同様ですが、*ls* および *rs* は共に *vertch* で、*ts* および *bs* は共に *horch* です。この関数では、角に使われる文字は常に標準設定の値です。

**clear**()

**erase**() に似ていますが、次に **refresh**() が呼び出された際に全てのウィンドウを再描画するようにします。

**clearok**(*yes*)

*yes* が 1 ならば、次の **refresh**() はウィンドウを完全に消去します。

**clrtoeol**()

カーソルの位置からウィンドウの端までを消去します: カーソル以降の全ての行が削除されるため、**clrtoeol**() が実行されたのとおなじになります。

`clrtoeol()`

カーソル位置から行末までを消去します。

`cursyncup()`

ウィンドウの全ての親ウィンドウについて、現在のカーソル位置を反映するよう更新します。

`delch([x, y])`

(*y*, *x*) にある文字を削除します。Delete any character at (*y*, *x*).

`deleteln()`

カーソルの下にある行を削除します。後続の行はすべて 1 行上に移動します。

`derwin([nlines, ncols,] begin_y, begin_x)`

“derive window (ウィンドウを導出する)” の短縮形です。derwin() は subwin() と同じですが、begin\_y および begin\_x はスクリーン全体の原点ではなく、ウィンドウの原点からの相対位置です。導出されたウィンドウオブジェクトが返されます。

`echochar(ch[, attr])`

文字 *ch* に属性 *attr* を付与し、即座に refresh() をウィンドウに対して呼び出します。

`enclose(y, x)`

与えられた文字セル座標をスクリーン原点から相対的なものとし、ウィンドウの中に含まれるかを調べて、真または偽を返します。スクリーン上のウィンドウの一部がマウスイベントの発生場所を含むかどうかを調べる上で便利です。

`erase()`

ウィンドウをクリアします。

`getbegyx()`

左上の角の座標をあらわすタプル (*y*, *x*) を返します。

`getch([x, y])`

文字を取得します。返される整数は ASCII の範囲の値となるわけではないので注意してください: ファンクションキー、キーボード上のキー等は 256 よりも大きな数字を返します。無遅延 (no-delay) モードでは、入力がない場合 -1 が返されます。

`getkey([x, y])`

文字を取得し、getch() のように整数を返す代わりに文字列を返します。ファンクションキー、キーボードキーなどはキー名の入った複数バイトからなる文字列を返します。無遅延モードでは、入力がない場合例外が送出されます。

`getmaxyx()`

ウィンドウの高さおよび幅を表すタプル (*y*, *x*) を返します。

`getparyx()`

親ウィンドウ中におけるウィンドウの開始位置を *x* と *y* の二つの整数で返します。ウィンドウに親ウィンドウがない場合 -1, -1 を返します。

`getstr([x, y])`

原始的な文字編集機能つきで、ユーザの入力文字列を読み取ります。

`getyx()`

ウィンドウの左上角からの相対で表した現在のカーソル位置をタプル (*y*, *x*) で返します。

`hline([y, x,] ch, n)`

(*y*, *x*) から始まり、*n* の長さを持つ、文字 *ch* で作られる水平線を表示します。

`idcok(flag)`

*flag* が偽の場合、curses は端末のハードウェアによる文字挿入 / 削除機能を使おうとしなくなります; *flag* が真ならば、文字挿入 / 削除は有効にされます。curses が最初に初期化された際には文字挿入 /



削除は標準の設定で有効になっています。

**idlok**(*yes*)

*yes* が 1 であれば、`curses` はハードウェアの行編集機能を利用しようと試みます。行挿入 / 削除は無効化されます。

**immedok**(*flag*)

*flag* が真ならば、ウィンドウイメージ内における何らかの変更があるとウィンドウを更新するようになります; すなわち、`refresh()` を自分で呼ばなくても良くなります。とはいえ、`wrefresh` を繰り返し呼び出すことになるため、この操作はかなりパフォーマンスを低下させます。標準の設定では無効になっています。

**inch**(*[x, y]*)

ウィンドウの指定の位置の文字を返します。下位 8 ビットが常に文字となり、それより上のビットは属性を表します。

**insch**(*[y, x, ] ch[, attr]*)

(*y, x*) に文字 *ch* を属性 *attr* で描画し、行の *x* からの内容を 1 文字分右にずらします。

**insdelln**(*nlines*)

*nlines* 行を指定されたウィンドウの現在の行の上に挿入します。その下にある *nlines* 行は失われます。負の *nlines* を指定すると、カーソルのある行以降の *nlines* を削除し、削除された行の後ろに続く内容が上に来ます。その下にある *nlines* は消去されます。現在のカーソル位置はそのままです。

**insertln**()

カーソルの下に空行を 1 行入れます。それ以降の行は 1 行ずつ下に移動します。

**insnstr**(*[y, x, ] str, n[, attr]*)

文字列をカーソルの下にある文字の前に (一行に収まるだけ) 最大 *n* 文字挿入します。*n* がゼロまたは負の値の場合、文字列全体が挿入されます。カーソルの右にある全ての文字は右に移動し、行の左端にある文字は失われます。カーソル位置は (*y, x* が指定されていた場合はそこに移動しますが、その後は) 変化しません。

**insstr**(*[y, x, ] str[, attr]*)

キャラクタ文字列を (行に収まるだけ) カーソルより前に挿入します。カーソルの右側にある文字は全て右にシフトし、行の右端の文字は失われます。カーソル位置は (*y, x* が指定されていた場合はそこに移動しますが、その後は) 変化しません。

**instr**(*[y, x] [, n]*)

現在のカーソル位置、または *y, x* が指定されている場合にはその場所から始まるキャラクタ文字列をウィンドウから抽出して返します。属性は文字から剥ぎ取られます。*n* が指定された場合、`instr()` は (末尾の NUL 文字を除いて) 最大で *n* 文字までの長さからなる文字列を返します。

**is\_linetouched**(*line*)

指定した行が、最後に `refresh()` を呼んだ時から変更されている場合に真を返します; そうでない場合には偽を返します。*line* が現在のウィンドウ上の有効な行でない場合、`curses.error` 例外を送出します。

**is\_wintouched**()

指定したウィンドウが、最後に `refresh()` を呼んだ時から変更されている場合に真を返します; そうでない場合には偽を返します。

**keypad**(*yes*)

*yes* が 1 の場合、ある種のキー (キーパッドやファンクションキー) によって生成されたエスケープシーケンスは `curses` で解釈されます。*yes* が 0 の場合、エスケープシーケンスは入力ストリームにそのままの状態に残されます。



**leaveok**(yes)

yes が 1 の場合、カーソルは“カーソル位置”に移動せず現在の場所にとどめます。これにより、カーソルの移動を減らせる可能性があります。この場合、カーソルは不可視にされます。

yes が 0 の場合、カーソルは更新の際に常に“カーソル位置”に移動します。

**move**(new\_y, new\_x)

カーソルを (new\_y, new\_x) に移動します。

**mvderwin**(y, x)

ウィンドウを親ウィンドウの中で移動します。ウィンドウのスクリーン相対となるパラメタ群は変化しません。このルーチンは親ウィンドウの一部をスクリーン上の同じ物理位置に表示する際に用いられます。

**mvwin**(new\_y, new\_x)

ウィンドウの左上角が (new\_y, new\_x) になるように移動します。

**nodelay**(yes)

yes が 1 の場合、getch() は非ブロックで動作します。

**notimeout**(yes)

yes が 1 の場合、エスケープシーケンスはタイムアウトしなくなります。

yes が 0 の場合、数ミリ秒の間エスケープシーケンスは解釈されず、入力ストリーム中にそのままの状態に残されます。

**noutrefresh**()

更新をマークはしますが待機します。この関数はウィンドウのデータ構造を表現したい内容を反映するように更新しますが、物理スクリーン上に反映させるための強制更新を行いません。更新を行うためには doupdate() を呼び出します。

**overlay**(destwin[, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol])

ウィンドウを destwin の上に重ね書き (overlay) します。ウィンドウは同じサイズである必要はなく、重なっている領域だけが複写されます。この複写は非破壊的 (non-destructive) です。これは現在の背景文字が destwin の内容を上書きしないことを意味します。

複写領域をきめ細かく制御するために、overlay() の第二形式を使うことができます。sminrow および smincol は元のウィンドウの左上の座標で、他の変数は destwin 内の矩形を表します。

**overwrite**(destwin[, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol])

destwin の上にウィンドウの内容を上書き (overwrite) します。ウィンドウは同じサイズである必要はなく、重なっている領域だけが複写されます。この複写は破壊的 (destructive) です。これは現在の背景文字が destwin の内容を上書きすることを意味します。

複写領域をきめ細かく制御するために、overlay() の第二形式を使うことができます。sminrow および smincol は元のウィンドウの左上の座標で、他の変数は destwin 内の矩形を表します。

**putwin**(file)

ウィンドウに関連付けられている全てのデータを与えられたファイルオブジェクトに書き込みます。この情報は後に getwin() 関数を使って取得することができます。

**redrawln**(beg, num)

beg 行から始まる num スクリーン行の表示内容が壊れており、次の refresh() 呼び出しで完全に再描画されなければならないことを通知します。

**redrawwin**()

ウィンドウ全体を更新 (touch) し、次の refresh() 呼び出しで完全に再描画されるようにします。

**refresh**([pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol])

ディスプレイを即時更新し (現実のウィンドウとこれまでの描画 / 削除メソッドの内容との同期をと

り)ます。

6つのオプション引数はウィンドウが `newpad()` で生成された場合にのみ指定することができます。追加の引数はパッドやスクリーンのどの部分が含まれるのかを示すために必要です。`pminrow` および `pmincol` にはパッドが表示されている矩形の左上角を指定します。`sminrow`, `smincol`, `smaxrow`, および `smaxcol` には、スクリーン上に表示される矩形の縁を指定します。パッド内に表示される矩形の右下角はスクリーン座標から計算されるので、矩形は同じサイズでなければなりません。矩形は両方とも、それぞれのウィンドウ構造内に完全に含まれていなければなりません。`pminrow`, `pmincol`, `sminrow`, または `smincol` に負の値を指定すると、ゼロを指定したものとして扱われます。

**scroll**(`[lines = 1]`)

スクリーンまたはスクロール領域を上 `lines` 行スクロールします。

**scrollok**(`flag`)

ウィンドウのカーソルが、最下行で改行を行ったり最後の文字を入力したりした結果、ウィンドウやスクロール領域の縁からはみ出して移動した際の動作を制御します。`flag` が偽の場合、カーソルは最下行にそのままにしておかれます。`flag` が真の場合、ウィンドウは1行上にスクロールします。端末の物理スクロール効果を得るためには `idlok()` も呼び出す必要があるので注意してください。

**setscrreg**(`top`, `bottom`)

スクロール領域を `top` から `bottom` に設定します。スクロール動作は全てこの領域で行われます。

**standend**( )

`A_STANDOUT` 属性をオフにします。端末によっては、この操作で全ての属性をオフにする副作用が発生します。

**standout**( )

`A_STANDOUT` 属性をオンにします。

**subpad**(`[nlines, ncols, ] begin_y, begin_x`)

左上の角が (`begin_y`, `begin_x`) にあり、幅 / 高さがそれぞれ `ncols/nlines` であるようなサブウィンドウを返します。

**subwin**(`[nlines, ncols, ] begin_y, begin_x`)

左上の角が (`begin_y`, `begin_x`) にあり、幅 / 高さがそれぞれ `ncols/nlines` であるようなサブウィンドウを返します。

標準の設定では、サブウィンドウは指定された場所からウィンドウの右下角まで広がります。

**syncdown**( )

このウィンドウの上位のウィンドウのいずれかで更新 (`touch`) された各場所をこのウィンドウ内でも更新します。このルーチンは `refresh()` から呼び出されるので、手動で呼び出す必要はほとんどないはずです。

**syncok**(`flag`)

`flag` を真にして呼び出すと、ウィンドウが変更された際は常に `syncup()` を自動的に呼ぶようになります。

**syncup**( )

ウィンドウ内で更新 (`touch`) した場所を、上位の全てのウィンドウ内でも更新します。

**timeout**(`delay`)

ウィンドウのブロックまたは非ブロック読み込み動作を設定します。`delay` が負の場合、ブロック読み出しが使われ、入力を無期限で待ち受けます。`delay` がゼロの場合、非ブロック読み出しが使われ、入力待ちの文字がない場合 `getch()` は -1 を返します。`delay` が正の値であれば、`getch()` は `delay` ミリ秒間ブロックし、ブロック後の時点で入力がない場合には -1 を返します。

**touchline**(`start`, `count`)

`start` から始まる `count` 行が変更されたかのように振舞わせます。

`touchwin()`

描画を最適化するために、全てのウィンドウが変更されたかのように振舞わせます。

`untouchwin()`

ウィンドウ内の全ての行を、最後に `refresh()` を呼んだ際から変更されていないものとしてマークします。

`vline([y, x], ch, n)`

$(y, x)$  から始まり、 $n$  の長さを持つ、文字  $ch$  で作られる垂直線を表示します。

### 6.14.3 定数

`curses` モジュールでは以下のデータメンバを定義しています:

**ERR**

`getch()` のような整数を返す `curses` ルーチンのいくつかは、失敗した際に **ERR** を返します。

**OK**

`napms()` のような整数を返す `curses` ルーチンのいくつかは、成功した際に **OK** を返します。

**version**

モジュールの現在のバージョンを表現する文字列です。 `__version__` でも取得できます。

以下に文字セルの属性を指定するために利用可能ないくつかの定数を示します:

属性	意味
<code>A_ALTCHARSET</code>	代用文字 (alternate character) モード。
<code>A_BLINK</code>	点滅モード。
<code>A_BOLD</code>	太字モード。
<code>A_DIM</code>	低輝度モード。
<code>A_NORMAL</code>	通常の属性。
<code>A_STANDOUT</code>	強調モード。
<code>A_UNDERLINE</code>	下線モード。

キーは `'KEY_'` で始まる名前をもつ整数定数です。利用可能なキーキャップはシステムに依存します。

キー定数	キー
<code>KEY_MIN</code>	最小のキー値
<code>KEY_BREAK</code>	ブレーク (Break, 信頼できません)
<code>KEY_DOWN</code>	下向き矢印 (Down-arrow)
<code>KEY_UP</code>	上向き矢印 (Up-arrow)
<code>KEY_LEFT</code>	左向き矢印 (Left-arrow)
<code>KEY_RIGHT</code>	右向き矢印 (Right-arrow)
<code>KEY_HOME</code>	ホームキー (Home, または上左矢印)
<code>KEY_BACKSPACE</code>	バックスペース (Backspace, 信頼できません)
<code>KEY_F0</code>	ファンクションキー 64 個までサポートされています。
<code>KEY_Fn</code>	ファンクションキー $n$ の値
<code>KEY_DL</code>	行削除 (Delete line)
<code>KEY_IL</code>	行挿入 (Insert line)
<code>KEY_DC</code>	文字削除 (Delete char)
<code>KEY_IC</code>	文字挿入、または文字挿入モードへ入る

キー定数	キー
KEY_EIC	文字挿入モードから抜ける
KEY_CLEAR	画面消去
KEY_EOS	画面の末端まで消去
KEY_EOL	行末端まで消去
KEY_SF	前に 1 行スクロール
KEY_SR	後ろ (逆方向) に 1 行スクロール
KEY_NPAGE	次のページ (Page Next)
KEY_PPAGE	前のページ (Page Prev)
KEY_STAB	タブ設定
KEY_CTAB	タブリセット
KEY_CATAB	全てのタブをリセット
KEY_ENTER	入力または送信 (信頼できません)
KEY_SRESET	ソフトウェア (部分的) リセット (信頼できません)
KEY_RESET	リセットまたはハードリセット (信頼できません)
KEY_PRINT	印刷 (Print)
KEY_LL	下ホーム (Home down) または最下行 (左下)
KEY_A1	キーパッドの左上キー
KEY_A3	キーパッドの右上キー
KEY_B2	キーパッドの中央キー
KEY_C1	キーパッドの左下キー
KEY_C3	キーパッドの右下キー
KEY_BTAB	Back tab
KEY_BEG	開始 (Beg)
KEY_CANCEL	キャンセル (Cancel)
KEY_CLOSE	閉じる (Close)
KEY_COMMAND	コマンド (Cmd)
KEY_COPY	コピー (Copy)
KEY_CREATE	生成 (Create)
KEY_END	終了 (End)
KEY_EXIT	終了 (Exit)
KEY_FIND	検索 (Find)
KEY_HELP	ヘルプ (Help)
KEY_MARK	マーク (Mark)
KEY_MESSAGE	メッセージ (Message)
KEY_MOVE	移動 (Move)
KEY_NEXT	次へ (Next)
KEY_OPEN	開く (Open)
KEY_OPTIONS	オプション (Options)
KEY_PREVIOUS	前へ (Prev)
KEY_REDO	やり直し (Redo)
KEY_REFERENCE	参照 (Ref)
KEY_REFRESH	更新 (Refresh)
KEY_REPLACE	置換 (Replace)
KEY_RESTART	再起動 (Restart)
KEY_RESUME	再開 (Resume)

キー定数	キー
KEY_SAVE	保存 (Save)
KEY_SBEG	シフト付き開始 Beg
KEY_SCANCEL	シフト付きキャンセル Cancel
KEY_SCOMMAND	シフト付き Command
KEY_SCOPY	シフト付き Copy
KEY_SCREATE	シフト付き Create
KEY_SDC	シフト付き Delete char
KEY_SDL	シフト付き Delete line
KEY_SELECT	選択 (Select)
KEY_SEND	シフト付き End
KEY_SEOL	シフト付き Clear line
KEY_SEXIT	シフト付き Dxit
KEY_SFIND	シフト付き Find
KEY_SHELP	シフト付き Help
KEY_SHOME	シフト付き Home
KEY_SIC	シフト付き Input
KEY_SLEFT	シフト付き Left arrow
KEY_SMESSAGE	シフト付き Message
KEY_SMOVE	シフト付き Move
KEY_SNEXT	シフト付き Next
KEY_SOPTIONS	シフト付き Options
KEY_SPREVIOUS	シフト付き Prev
KEY_SPRINT	シフト付き Print
KEY_SREDO	シフト付き Redo
KEY_SREPLACE	シフト付き Replace
KEY_SRIGHT	シフト付き Right arrow
KEY_SRSUME	シフト付き Resume
KEY_SSAVE	シフト付き Save
KEY_SSUSPEND	シフト付き Suspend
KEY_SUNDO	シフト付き Undo
KEY_SUSPEND	一時停止 (Suspend)
KEY_UNDO	元に戻す (Undo)
KEY_MOUSE	マウスイベント通知
KEY_RESIZE	端末リサイズイベント
KEY_MAX	最大キー値

VT100 や、X 端末エミュレータのようなソフトウェアエミュレーションでは、通常少なくとも 4 つのファンクションキー (KEY\_F1、KEY\_F2、KEY\_F3、KEY\_F4) が利用可能で、矢印キーは KEY\_UP、KEY\_DOWN、KEY\_LEFT および KEY\_RIGHT が対応付けられています。計算機に PC キーボードが付属している場合、矢印キーと 12 個のファンクションキー (古い PC キーボードには 10 個しかファンクションキーがないかもしれません) が利用できると思ってよいでしょう; また、以下のキーパッド対応付けは標準的なものです:

キーキャップ	定数
Insert	KEY_IC
Delete	KEY_DC
Home	KEY_HOME
End	KEY_END
Page Up	KEY_NPAGE
Page Down	KEY_PPAGE

代用文字 (alternative character) セットを以下の表に列挙します。これらは VT100 端末から継承したものであり、X 端末のようなソフトウェアエミュレーション上で一般に利用可能なものです。グラフィックが利用できない場合、curses は印字可能 ASCII 文字による粗雑な近似出力を行います。注意: これらは `initscr()` が呼び出された後でしか利用できません。

ACS コード	意味
ACS_BBSS	右上角の別名
ACS_BLOCK	黒四角ブロック
ACS_BOARD	白四角ブロック
ACS_BSBS	水平線の別名
ACS_BSSB	左上角の別名
ACS_BSSS	上向き T 字罫線の別名
ACS_BTEE	下向き T 字罫線
ACS_BULLET	黒丸 (bullet)
ACS_CKBOARD	チェッカーボードボタン (点描)
ACS_DARROW	下向き矢印
ACS_DEGREE	度
ACS_DIAMOND	ダイヤモンド
ACS_GEQUAL	より大きいか等しい
ACS_HLINE	水平線
ACS_LANTERN	ランタン (lantern) シンボル
ACS_LARROW	left arrow
ACS_LEQUAL	より小さいか等しい
ACS_LLCORNER	左下角
ACS_LRCORNER	右下角
ACS_LTEE	left tee
ACS_NEQUAL	等号否定
ACS_PI	パイ記号
ACS_PLMINUS	プラスマイナス記号
ACS_PLUS	大プラス記号
ACS_RARROW	右向き矢印
ACS_RTEE	右向き T 字罫線
ACS_S1	scan line 1
ACS_S3	scan line 3
ACS_S7	scan line 7
ACS_S9	scan line 9
ACS_SBBS	右下角の別名
ACS_SBSB	垂直線の別名
ACS_SBSS	右向き T 字罫線の別名



ACS コード	意味
ACS_SSBB	左下角の別名
ACS_SSBS	下向き T 字罫線の別名
ACS_SSSB	左向き T 字罫線の別名
ACS_SSSS	交差罫線または大プラス記号の別名
ACS_STERLING	ポンドスターリング記号
ACS_TTEE	上向き T 字罫線
ACS_UARROW	上向き矢印
ACS_ULCORNER	左上角
ACS_URCORNER	右上角
ACS_VLINE	垂直線

以下のテーブルは定義済みの色を列挙したものです:

定数	色
COLOR_BLACK	黒
COLOR_BLUE	青
COLOR_CYAN	シアン (薄く緑がかった青)
COLOR_GREEN	緑
COLOR_MAGENTA	マゼンタ (紫がかった赤)
COLOR_RED	赤
COLOR_WHITE	白
COLOR_YELLOW	黄色

## 6.15 curses.textpad — curses プログラムのためのテキスト入力ウィジェット

1.6 で追加された仕様です。

curses.textpad モジュールでは、curses ウィンドウ内での基本的なテキスト編集を処理し、Emacs に似た (すなわち Netscape Navigator, BBedit 6.x, FrameMaker, その他諸々のプログラムとも似た) キーバインドをサポートしている Textbox クラスを提供します。このモジュールではまた、テキストボックスを枠で囲むなどの目的のために有用な、矩形描画関数を提供しています。

curses.textpad モジュールでは以下の関数を定義しています:

**rectangle**(win, uly, ulx, lry, lrx)

矩形を描画します。最初の引数はウィンドウオブジェクトでなければなりません; 残りの引数はそのウィンドウからの相対座標になります。2 番目および 3 番目の引数は描画すべき矩形の左上角の y および x 座標です; 4 番目および 5 番目の引数は右下角の y および x 座標です。矩形は、VT100/IBM PC におけるフォーム文字を利用できる端末 (xterm やその他のほとんどのソフトウェア端末エミュレータを含む) ではそれを使って描画されます。そうでなければ ASCII 文字のダッシュ、垂直バー、およびプラス記号で描画されます。

### 6.15.1 Textbox オブジェクト

以下のような Textbox オブジェクトをインスタンス生成することができます:

**class Textbox**(win)

テキストボックスウィジェットオブジェクトを返します。win 引数は、テキストボックスを入れるための WindowObject でなければなりません。テキストボックスの編集カーソルは、最初はテキストボックスが入っているウィンドウの左上角に配置され、その座標は (0, 0) です。インスタンスの stripspaces フラグの初期値はオンに設定されます。

Textbox オブジェクトは以下のメソッドを持ちます:

**edit**([validator])

普段使うことになるエントリポイントです。終了キーストロークの一つが入力されるまで編集キーストロークを受け付けます。validator を与える場合、関数でなければなりません。validator はキーストロークが入力されるたびにそのキーストロークが引数となって呼び出されます; 返された値に対して、コマンドキーストロークとして解釈が行われます。このメソッドはウィンドウの内容を文字列として返します; ウィンドウ内の空白が含まれるかどうかは stripspaces メンバで決められます。

**do\_command**(ch)

単一のコマンドキーストロークを処理します。以下にサポートされている特殊キーストロークを示します:

キーストローク	動作
Control-A	ウィンドウの左端に移動します。
Control-B	カーソルを左へ移動し、必要なら前の行に折り返します。
Control-D	カーソル下の文字を削除します。
Control-E	右端 (stripspaces がオフのとき) または行末 (stripspaces がオンのとき) に移動します。
Control-F	カーソルを右に移動し、必要なら次の行に折り返します。
Control-G	ウィンドウを終了し、その内容を返します。
Control-H	逆方向に文字を削除します。(バックスペース)
Control-J	ウィンドウが 1 行であれば終了し、そうでなければ新しい行を挿入します。
Control-K	行が空白行ならその行全体を削除し、そうでなければカーソル以降行末までを消去します。
Control-L	スクリーンを更新します。
Control-N	カーソルを下に移動します; 1 行下に移動します。
Control-O	カーソルの場所に空行を 1 行挿入します。
Control-P	カーソルを上移動します; 1 行上に移動します。

移動操作は、カーソルがウィンドウの縁にあって移動ができない場合には何も行いません。場合によっては、以下のような同義のキーストロークがサポートされています:

定数	キーストローク
KEY_LEFT	Control-B
KEY_RIGHT	Control-F
KEY_UP	Control-P
KEY_DOWN	Control-N
KEY_BACKSPACE	Control-h

他のキーストロークは、与えられた文字を挿入し、(行折り返し付きで) 右に移動するコマンドとして扱われます。

**gather**()

このメソッドはウィンドウの内容を文字列として返します; ウィンドウ内の空白が含まれるかどうかは stripspaces メンバ変数で決められます。

**stripspaces**

このデータメンバはウィンドウ内の空白領域の解釈方法を制御するためのフラグです。フラグがオン

に設定されている場合、各行の末端にある空白領域は無視されます; すなわち、末端空白領域にカーソルが入ると、その場所の代わりに行の末尾にカーソルが移動します。また、末端の空白領域はウィンドウの内容を取得する際に剥ぎ取られます。

## 6.16 `curses.wrapper` — `curses` プログラムのための端末ハンドラ

1.6 で追加された仕様です。

このモジュールでは関数 `wrapper()` 一つを提供しています。これは `curses` 使用アプリケーションの残りの部分となるもう一つの関数です。アプリケーションが例外を送出した場合、`wrapper()` はスタックを整理してトレースバックを生成する前に端末を正常な状態に復元します。

`wrapper(func, ...)`

`curses` を初期化し、別の関数 *func* を呼び出、エラーが発生した場合には通常のキーボード / スクリーン動作に戻すラップ関数です。呼び出し可能オブジェクト *func* は主ウィンドウの `'stdscr'` に対する最初の引数として渡されます。その他の引数は `wrapper()` に渡されます。

フック関数を呼び出す前に、`wrapper()` は `cbreak` モードをオン、エコーをオフにし、端末キーパッドを有効にします。端末がカラーをサポートしている場合にはカラーを初期化します。(通常終了も例外による終了も) 終了時には `cooked` モードに復元し、エコーをオンにし、端末キーパッドを無効化します。

## 6.17 `curses.ascii` — ASCII 文字に関するユーティリティ

1.6 で追加された仕様です。

`curses.ascii` モジュールでは、ASCII 文字を指す名前定数と、様々な ASCII 文字区分についてある文字が帰属するかどうかを調べる関数を提供します。このモジュールで提供されている定数は以下の制御文字の名前です:

Name	Meaning
NUL	空
SOH	ヘディング開始、コンソール割り込み
STX	テキスト開始
ETX	テキスト終了
EOT	テキスト伝送終了
ENQ	問い合わせ、ACK フロー制御時に使用
ACK	肯定応答
BEL	ベル
BS	一文字後退
TAB	タブ
HT	TAB の別名: “水平タブ”
LF	改行
NL	LF の別名: “改行”
VT	垂直タブ
FF	改頁
CR	復帰
SO	シフトアウト、他の文字セットの開始
SI	シフトイン、標準の文字セットに復帰
DLE	データリンクでのエスケープ
DC1	装置制御 1、フロー制御のための XON
DC2	装置制御 2、ブロックモードフロー制御
DC3	装置制御 3、フロー制御のための XOFF
DC4	装置制御 4
NAK	否定応答
SYN	同期信号
ETB	ブロック転送終了
CAN	キャンセル
EM	媒体終端
SUB	代入文字
ESC	エスケープ文字
FS	ファイル区切り文字
GS	グループ区切り文字
RS	レコード区切り文字、ブロックモード終了子
US	単位区切り文字
SP	空白文字
DEL	削除

これらの大部分は、最近では実際に定数の意味通りに使われることがほとんどないので注意してください。これらのニーモニック符号はデジタル計算機より前のテレプリンタにおける慣習から付けられたものです。

このモジュールでは、標準 C ライブラリの関数を雛型とする以下の関数をサポートしています:

`isalnum(c)`

ASCII 英数文字かどうかを調べます; ‘`isalpha(c)` or ‘`isdigit(c)`’ と等価です。

`isalpha(c)`

ASCII アルファベット文字かどうかを調べます; ‘`isupper(c)` or ‘`islower(c)`’ と等価です。

`isascii(c)`

文字が 7 ビット ASCII 文字に合致するかどうかを調べます。

**isblank(*c*)**

ASCII 余白文字かどうかを調べます。

**iscntrl(*c*)**

ASCII 制御文字 (0x00 から 0x1f の範囲) かどうかを調べます。

**isdigit(*c*)**

ASCII 10 進数字、すなわち '0' から '9' までの文字かどうかを調べます。'*c* in string.digits' と等価です。

**isgraph(*c*)**

空白以外の ASCII 印字可能文字かどうかを調べます。

**islower(*c*)**

ASCII 小文字かどうかを調べます。

**isprint(*c*)**

空白文字を含め、ASCII 印字可能文字かどうかを調べます。

**ispunct(*c*)**

空白または英数字以外の ASCII 印字可能文字かどうかを調べます。

**isspace(*c*)**

ASCII 余白文字、すなわち空白、改行、復帰、改頁、水平タブ、垂直タブかどうかを調べます。

**isupper(*c*)**

ASCII 大文字かどうかを調べます。

**isxdigit(*c*)**

ASCII 16 進数字かどうかを調べます。'*c* in string.hexdigits' と等価です。

**isctrl(*c*)**

ASCII 制御文字 (0 から 31 までの値) かどうかを調べます。

**ismeta(*c*)**

非 ASCII 文字 (0x80 またはそれ以上の値) かどうかを調べます。

これらの関数は数字も文字列も使えます; 引数を文字列にした場合、組み込み関数 `ord()` を使って変換されます。

これらの関数は全て、関数に渡した文字列の最初の文字から得られたビット値を調べるので注意してください; 関数はホスト計算機で使われている文字列エンコーディングについて何ら関知しません。文字列エンコーディングについて関知する (そして国際化に関するプロパティを正しく扱う) 関数については、モジュール `string` を参照してください。

以下の 2 つの関数は、引数として 1 文字の文字列または整数で表したバイト値のどちらでもとり得ます; これらの関数は引数と同じ型で値を返します。

**ascii(*c*)**

ASCII 値を返します。*c* の下位 7 ビットに対応します。

**ctrl(*c*)**

与えた文字に対応する制御文字を返します (0x1f とビット単位で論理積を取ります)。

**alt(*c*)**

与えた文字に対応する 8 ビット文字を返します (0x80 とビット単位で論理和を取ります)。

以下の関数は 1 文字からなる文字列値または整数値を引数に取り、文字列を返します。

**unctrl(*c*)**

ASCII 文字 *c* の文字列表現を返します。もし *c* が印字可能文字であれば、返される文字列は *c* そのも

のになります。もし *c* が制御文字 (0x00-0x1f) であれば、キャレット (‘^’) と、その後ろに続く *c* に対応した大文字からなる文字列になります。*c* が ASCII 削除文字 (0x7f) であれば、文字列は ‘^?’ になります。*c* のメタビット (0x80) がセットされていれば、メタビットは取り去られ、前述のルールが適用され、‘!’ が前につけられます。

#### `controlnames`

0 (NUL) から 0x1f (US) までの 32 の ASCII 制御文字と、空白文字 ‘SP’ の二モニック符号名からなる 33 要素の文字列による配列です。

## 6.18 `curses.panel` — `curses` のためのパネルスタック拡張

パネルは深さ (depth) の機能が追加されたウィンドウです。これにより、ウィンドウをお互いに重ね合わせることができ、各ウィンドウの可視部分だけが表示されます。パネルはスタック中に追加したり、スタック内で上下移動させたり、スタックから除去することができます。

### 6.18.1 関数

`curses.panel` では以下の関数を定義しています:

#### `bottom_panel()`

パネルスタックの最下層のパネルを返します。

#### `new_panel(win)`

与えられたウィンドウ *win* に関連付けられたパネルオブジェクトを返します。

#### `top_panel()`

パネルスタックの最上層のパネルを返します。

#### `update_panels()`

仮想スクリーンをパネルスタック変更後の状態に更新します。この関数では `curses.doupdate()` を呼ばないので、ユーザは自分で呼び出す必要があります。

### 6.18.2 Panel オブジェクト

上記の `new_panel()` が返す Panel オブジェクトはスタック順の概念を持つウィンドウです。ウィンドウはパネルに関連付けられており、表示する内容を決定している一方、パネルのメソッドはパネルスタック中のウィンドウの深さ管理を担います。

Panel オブジェクトは以下のメソッドを持っています:

#### `above()`

現在のパネルの上にあるパネルを返します。

#### `below()`

現在のパネルの下にあるパネルを返します。

#### `bottom()`

パネルをスタックの最下層にプッシュします。

#### `hidden()`

パネルが隠れている (不可視である) 場合に真を返し、そうでない場合偽を返します。

#### `hide()`

パネルを隠します。この操作ではオブジェクトは消去されず、スクリーン上のウィンドウを不可視にするだけです。



`move(y, x)`

パネルをスクリーン座標  $(y, x)$  に移動します。

`replace(win)`

パネルに関連付けられたウィンドウを *win* に変更します。

`set_userptr(obj)`

パネルのユーザポインタを *obj* に設定します。このメソッドは任意のデータをパネルに関連付けるために使われ、任意の Python オブジェクトにすることができます。

`show()`

(隠れているはずの) パネルを表示します。

`top()`

パネルをスタックの最上層にプッシュします。

`userptr()`

パネルのユーザポインタを返します。任意の Python オブジェクトです。

`window()`

パネルに関連付けられているウィンドウオブジェクトを返します。

## 6.19 getopt — コマンドラインオプションのパーザ

このモジュールは `sys.argv` に入っているコマンドラインオプションの構文解析を支援します。‘-’ や ‘--’ の特別扱いも含めて、UNIX の `getopt()` と同じ記法をサポートしています。3 番目の引数 (省略可能) を設定することで、GNU のソフトウェアでサポートされているような長形式のオプションも利用することができます。このモジュールは 1 つの関数と例外を提供しています:

`getopt(args, options[, long_options])`

コマンドラインオプションとパラメータのリストを構文解析します。*args* は構文解析の対象になる引数リストです。これは先頭のプログラム名を除いたもので、通常 `sys.argv[1:]` で与えられます。*options* はスクリプトで認識させたいオプション文字と、引数が必要な場合にはコロン (‘:’) をつけます。つまり UNIX の `getopt()` と同じフォーマットになります。

注意: GNU の `getopt()` とは違って、オプションでない引数の後は全てオプションではないと判断されます。これは GNU でない、UNIX システムの挙動に近いものです。

*long\_options* は長形式のオプションの名前を示す文字列のリストです。名前には、先頭の ‘--’ は含めません。引数が必要な場合には名前の最後に等号 (‘=’) を入れます。長形式のオプションだけを受け付けるためには、*options* は空文字列である必要があります。長形式のオプションは、該当するオプションを一意に決定できる長さまで入力されていれば認識されます。たとえば、*long\_options* が `['foo', 'frob']` の場合、`--fo` は `--foo` に該当しますが、`--f` では一意に決定できないので、`GetoptError` が発生します。

返り値は 2 つの要素から成っています: 最初は `(option, value)` のタプルのリスト、2 つ目はオプションリストを取り除いたあとに残ったプログラムの引数リストです (*args* の末尾部分のスライスになります)。それぞれの引数と値のタプルの最初の要素は、短形式の時はハイフン 1 つで始まる文字列 (例: ‘-x’)、長形式の時はハイフン 2 つで始まる文字列 (例: ‘--long-option’) となり、引数が 2 番目の要素になります。引数をとらない場合には空文字列が入ります。オプションは見つかった順に並んでいて、複数回同じオプションを指定することができます。長形式と短形式のオプションは混在させることができます。

`gnu_getopt(args, options[, long_options])`

この関数はデフォルトで GNU スタイルのスキャンモードを使う以外は `getopt()` と同じように動

作します。つまり、オプションとオプションでない引数とを混在させることができます。getopt() 関数はオプションでない引数を見つけると解析をやめてしまいます。

オプション文字列の最初の文字が '+' にするか、環境変数 POSIXLY\_CORRECT を設定することで、オプションでない引数を見つけると解析をやめるように振舞いを変えることができます。

#### exception GetoptError

引数リストの中に認識できないオプションがあった場合か、引数が必要なオプションに引数が与えられなかった場合に発生します。例外の引数は原因を示す文字列です。長形式のオプションについては、不要な引数が与えられた場合にもこの例外が発生します。msg 属性と opt 属性で、エラーメッセージと関連するオプションを取得できます。特に関係するオプションが無い場合には opt は空文字列となります。

1.6 で変更された仕様: GetoptError は error の別名として導入されました。

#### exception error

GetoptError へのエイリアスです。後方互換性のために残されています。

UNIX スタイルのオプションを使った例です:

```
>>> import getopt
>>> args = '-a -b -cfoo -d bar a1 a2'.split()
>>> args
['-a', '-b', '-cfoo', '-d', 'bar', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'abc:d:')
>>> optlist
[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]
>>> args
['a1', 'a2']
```

長形式のオプションを使っても同様です:

```
>>> s = '--condition=foo --testing --output-file abc.def -x a1 a2'
>>> args = s.split()
>>> args
['--condition=foo', '--testing', '--output-file', 'abc.def', '-x', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'x', [
...     'condition=', 'output-file=', 'testing'])
>>> optlist
[('--condition', 'foo'), ('--testing', ''), ('--output-file', 'abc.def'), ('-x',
'')]
>>> args
['a1', 'a2']
```

スクリプト中での典型的な使い方は以下ようになります:

```

import getopt, sys

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help", "output="])
    except getopt.GetoptError:
        # ヘルプメッセージを出力して終了
        usage()
        sys.exit(2)
    output = None
    verbose = False
    for o, a in opts:
        if o == "-v":
            verbose = True
        if o in ("-h", "--help"):
            usage()
            sys.exit()
        if o in ("-o", "--output"):
            output = a
    # ...

if __name__ == "__main__":
    main()

```

## 6.20 optparse — 強力なコマンドラインオプション解析器

2.3 で追加された仕様です。

optparse モジュールは、Python による強力で柔軟、拡張性があり簡単に利用できるコマンドライン解析ライブラリです。optparse を使うことで、有能で洗練されたコマンドラインオプションの解析機能を少ない労力でスクリプトに追加できます。

以下は optparse を使っていくつかのコマンドラインオプションを簡単なスクリプトに追加する例です：

```

from optparse import OptionParser

parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
                  help="write report to FILE", metavar="FILE")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose", default=1,
                  help="don't print status messages to stdout")

(options, args) = parser.parse_args()

```

このようにわずかな行数のコードによって、スクリプトのユーザはコマンドライン上で以下のような "よくある使い方" を実行できるようになります：

```

$ <yourscript> -f outfile --quiet
$ <yourscript> -qfoutfile
$ <yourscript> --file=outfile -q
$ <yourscript> --quiet --file outfile

```

(これらの結果は全て options.filename == "outfile" および options.verbose == 0 ... つまり予想していた結果となります)

もっと気の利いたことに、ユーザは

```
$ <yourscript> -h
$ <yourscript> --help
```

のいずれかを実行することができ、`optparse` はスクリプトのオプションについて簡単にまとめた内容を出力します:

```
usage: <yourscript> [options]

options:
  -h, --help            show this help message and exit
  -fFILE, --file=FILE  write report to FILE
  -q, --quiet           don't print status messages to stdout
```

以上は `optparse` がコマンドライン解析に与える柔軟性のほんの一部にすぎません。

## 6.20.1 オプション解析への道

`optparse` は、私が考えている最も明瞭かつ素直でユーザフレンドリーなコマンドラインプログラムのインタフェースを設計するための方法を実装したものです。

簡単に言うと、私は引数解析の王道 (そしてたくさんの邪道) についてかなり頑固な考えを持っており、`optparse` はそういった考えの多くを反映しています。この節では上記の哲学について説明するつもりです。この哲学は UNIX と GNU ツールキットに強く影響されています。

### 用語

まずは用語の定義を行う必要があります。

#### 引数 (argument)

コマンドラインでユーザが入力するテキストの塊で、シェルが `execl()` や `execv()` に引き渡すものです。Python では、引数は `sys.argv[1:]` の要素となります。( `sys.argv[0]` は実行しようとしているプログラムの名前です。引数解析に関しては、この要素はあまり重要ではありません。) UNIX シェルでは、“語 (word)” という用語も使います。

場合によっては `sys.argv[1:]` 以外の引数リストを代入の方が望ましいことがあるので、“引数” は “`sys.argv[1:]` または `sys.argv[1:]` の代替として提供される別のリストの要素” と読解するべきでしょう。

#### オプション (option)

追加的な情報を与えるための引数で、プログラムの実行に対する教示やカスタマイズを行います。オプションには多様な文法が存在します; 伝統的な UNIX における書法は `-` の後ろに一文字が続くもので、例えば `-x` や `-F` です。また、伝統的な UNIX における書法では、複数のオプションを一つの引数にまとめることができ、例えば `-x -F` は `-xF` と等価です。GNU プロジェクトでは `--` の後ろにハイフンで区切られた語を続ける方法、例えば `--file` や `--dry-run` を導入しています。`optparse` ではこれら二種類のオプション書法だけを提供しています。

他に見られる他のオプション書法には以下のようなものがあります:

- ハイフンの後ろに数個の文字が続くもので、例えば `-pf`。(このオプションは複数のオプションを一つにまとめたものとは \*違います\*。)

- ハイフンの後ろに語が続くもので、例えば `-file`。(これは技術的には上の書式と同じですが、通常同じプログラム上で一緒に使われることはありません。)
- プラス記号の後ろに一文字、数個の文字、または語を続けたもので、例えば `+f`、`+rgb`。
- スラッシュ記号の後ろに一文字、数個の文字、または語を続けたもので、例えば `/f`、`/file`。

これらのオプション書法は `optparse` ではサポートされておらず、将来にわたってサポートされることはありません。(これらのオプション書法のいずれかを本当に使いたいのなら、`OptionParser` をサブクラス化して、難解な部分を全て上書きしなければならないでしょう。しかしできればやめてください! `optparse` は熟慮の上で伝統的な UNIX/GNU 的書法を採っています; 最初の 3 つの書法は場所によっては非標準ですし、最後の書法は MS-DOS/Windows かつ/または VMS をターゲットにしているときしか意味をなしません。)

#### オプション引数 (option argument)

あるオプションの後ろに続く引数で、そのオプションに密接な関連をもち、オプションと同時に引数リストから取り出されます。しばしば、オプション引数はオプションと一体の引数としてとりまわることがあります。例えば以下:

```
["-f", "foo"]
```

は以下の引数:

```
["-ffoo"]
```

と等価なことがあります。( `optparse` ではこの書法をサポートしています)

あるオプションは引数をとることがなく、またあるオプションは常に引数をとります。多くの人々が“オプションのオプション引数”機能を欲しています。これ、あるオプションについては引数が存在するときだけ引数を取り、そうでないときには引数をとらないようにするという機能です。この機能は引数解析をあいまいにするため、議論的となる点です: 例えば、もし `-a` がオプション引数を取り、`-b` がまったく別のオプションだとしたら、`-ab` をどうやって解析すればいいのでしょうか? `optparse` は今のところこの機能をサポートしていません。

#### 固定引数 (positional argument)

他のオプションが解析される、すなわち他のオプションとその引数が解析されて引数リストから除去された後に引数リストに置かれているものです。

#### 必須のオプション (required option)

コマンドラインで与えなければならないオプションです; “必須なオプション”という言葉は矛盾のある語法であり、通常は貧弱なユーザインタフェースデザインと考えられています。 `optparse` では必須オプションの実装を妨げはしませんが、とりたてて実装の役に立つようにはなっていません。 `optparse` で必須オプションを実装する方法については“拡張の例”(6.20.5 節)を参照してください。

例えば、以下のような架空のコマンドラインを考えてみてください:

```
prog -v --report /tmp/report.txt foo bar
```

`-v` と `--report` は両方ともオプションです。 `--report` が引数の一つとると仮定すると、“`/tmp/report.txt`” はオプション引数です。“`foo`” および “`bar`” は固定引数です。

オプションはどう使うのでしょうか？

オプションはプログラムの実行をチューニングしたり、カスタマイズしたりするための補足的な情報を与えるために使われます。補足すると、オプションは通常 選択可能 (*optional*) となっています。プログラムはいかなるオプションが指定されていなくても、きちんと動作できなくてはなりません。(UNIX や GNU ツールセットからプログラムをランダムに取り出してみてください。プログラムはオプション無しで動作して、意味のある結果になるでしょう？私の知っている例外は `find`、`tar`、`dd` といったプログラムです — これらは全て変種のはみだし者で、インタフェースが非標準で混乱をまねくものだと批判されてきました。)

大多数の人々が、プログラムに“必須オプション”を持たせたいと望んでいます。しかし考えてみてください。もしオプションが必須だというのなら、そのオプションは オプション:選択可能項目 などではありません！プログラムがうまく動作するために絶対的に必要とする情報があるとするなら、固定引数を使うべきです。(とはいえ、どうしても“必須オプション”をプログラムに組み込みたいというのなら、`optparse` を使った実装方法については“拡張の例”(6.20.5 節)を読んでください。

ファイルをコピーする地味なユーティリティ `cp` について考えてみましょう。ファイルをコピーしようと試みるときに、コピー先や少なくとも一つのコピー元を指定しなければ、このコマンドは意味をなしません。従って、`cp` は引数無しで実行すると失敗します。しかしながら、このコマンドは、拡張性のある便利な書法を持っており、オプションに全然頼っていません:

```
$ cp SOURCE DEST
$ cp SOURCE ... DEST-DIR
```

You can get pretty far with just that. ほとんどの `cp` の実装では、ファイルをどのようにしてコピーするかを的確に微調整するためのオプション:例えばファイルモードや最終更新時刻を変更しないようにしたり、シンボリックリンクの追跡を避けたり、既存のファイルをぶちこわし(clobber)にする前に同意を求めたり、などを一そろい提供しています。しかし、これらのオプションはいずれも、ファイルを別のファイルにコピーしたり N 個のファイルを他のディレクトリにコピーしたりするといった、`cp` の主要な役割を混乱させることはありません。

固定引数はどう使うのでしょうか？

上の例について補足します: 固定引数とは、プログラムを動作させるために、絶対的にかつ明確に必要とされる情報の断片です。

よいユーザインタフェースとは、可能な限り少ない固定引数をもつべきです。プログラムを正しく動作させるために 17 個もの別個の情報が必要だとしたら、その情報をユーザからどうやって引き出そうかということとはさほど問題にはなりません — ほとんどの人はプログラムを正しく動作させられるようになる前にあきらめて立ち去るでしょう。これはユーザインタフェースがコマンドラインでも、設定ファイルでも GUI やその他のいずれであっても当てはまることからです: 多くの要求をユーザに押し付ければ、ほとんどのユーザはただ音をあげるだけなのです。

要するに、ユーザが絶対に提供しなければならない情報だけに制限する — 可能な限りよく練られたデフォルト設定を使うよう試みてください。もちろん、プログラムには適度な柔軟性を持たせたいとも望むはずですが、それこそがオプションの果たす役割です。繰り返しますが、設定ファイルのエントリであろうが、GUI の“環境設定”ダイアログにあるチェックボックスであろうが、コマンドライン上のオプションであろうが関係ありません — より多くオプションを実装すればプログラムはより柔軟性を持ち、実装はより難解になります。高すぎる柔軟性でユーザ(あなた自身もですよ!)を閉口させるのは至って簡単なのです。



## 6.20.2 基本的な使い方

`optparse` はとても柔軟性がある一方、基本的なケースを動作させるために試練を受けたり大量のドキュメントを読む必要はありません。このドキュメントでは、いくつかの単純な使用パターンを実演して、スクリプトで `optparse` を使えるようにすることを目的としています。

コマンドラインを `optparse` で解析するためには、`OptionParser` インスタンスを作成して `populate` する必要があります。当然ながら、`optparse` を使うスクリプトはいずれも `OptionParser` クラスを `import` しなければなりません。

```
from optparse import OptionParser
```

メインプログラムの冒頭で、パーザを作成してください:

```
parser = OptionParser()
```

これで、パーザ内のオプションを移すことができます。各オプションは現実には同義のオプション文字列になります; ほとんどの場合、短いオプション文字列を一つと、長いオプション文字列を一つ — 例えば `-f` and `--file` — を持ちます:

```
parser.add_option("-f", "--file", ...)
```

もちろん、最も知りたいことは、オプション文字列の後に続くキーワード引数が何かということです。このドキュメントでは、キーワード引数として指定できる内容について 4 種類: *action*、*type*、*dest* (destination: 目的変数)、および *help* だけをカバーします。

### "store" アクション

アクションは、*action* が設定されているオプションがコマンドライン内に見つかったときに `optparse` が行うべきことを教えます。例えば、*store* はこういう意味です: 次の引数 (または現在の引数の残りの部分) を取り上げ、正しい型であるか確認して、決められた目的変数に保存します。

例として、先ほどのオプションの `"..."` を埋めてみましょう:

```
parser.add_option("-f", "--file",
                  action="store", type="string", dest="filename")
```

次に、架空のコマンドラインをでっち上げて、`optparse` に解析してもらいましょう:

```
args = ["-f", "foo.txt"]
(options, args) = parser.parse_args(args)
```

(引数リストを `parse_args()` に渡さなければ、自動的に `sys.argv[1:]` を使うので注意してください。)

`optparse` が `-f` を見つけると、次の引数 — “foo.txt” — を吸出し、特殊なオブジェクトの *filename* 属性に記憶します。このオブジェクトは `parse_args()` の戻り値の最初の値です。従って以下のコード:

```
print options.filename
```

は “foo.txt” を出力します。

optparse でサポートされている他のオプションの型は “int” と “float” です。以下に整数型の引数を要求するオプションの例を示します:

```
parser.add_option("-n", type="int", dest="num")
```

長い文字列によるオプションを与えていないことに注意してください。これはまったく問題なく受理されます。また、*action* を指定していません — これはアクションがデフォルトで “store” になるからです。

架空のコマンドラインをもう一つ解析してみましょう。今度は、オプション引数をオプションの右側にぴったりくっつけて一緒にたにします — **-n42** (一つの引数のみ) は **-n 42** (二つの引数からなる) と等価になります:

```
(options, args) = parser.parse_args(["-n42"])
print options.num
```

は “42” を出力します。

“float” 型を使った例は、読者のための練習課題として残しておきます。

型を指定しない場合、optparse は引数を “string” 型と仮定します。デフォルトのアクションが “store” であることも併せて考えると、最初の例はもっと短くなります:

```
parser.add_option("-f", "--file", dest="filename")
```

目的変数名を与えなければ、optparse はオプション文字列から気の利いたデフォルト値を決めます: 最初の長い形式のオプション文字列が **--foo-bar** だったとすると、デフォルトの目的変数は *foo\_bar* となります。長い形式のオプションがない場合、optparse は最初の短いオプションに注目します: **-f** に対するデフォルトの目的変数は *f* となります。

型を追加するのはかなり簡単です; [6.20.5 節](#)、“新しい型を追加する” を参照してください。

#### その他の “store\_” 操作

フラグオプション — 特定のオプションが見つかったときに、ある変数を真または偽にする — は、とても一般的な機能です。optparse では、フラグオプションを二つの別個のアクション “store\_true” と “store\_false” でサポートします。例えば、*verbose* フラグが **-v** でオンにされ、**-q** でオフにされるとします:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```

ここで二つのオプションを同じ目的変数に設定しましたが、これは全く問題ありません。(ただし、デフォルト値を設定する際に少しだけ注意する必要があります — 以下を参照してください。)

optparse が **-v** をコマンドライン内に見つけると、特殊な option values オブジェクトの *verbose* 属性を 1 に設定します; **-q** が見つかれば、*verbose* を 0 に設定します。

#### デフォルト値を設定する

上で述べてきた例は全て、あるコマンドラインオプションが見つかった際に何らかの変数 (“destination: 目的変数”) の設定を伴うものでした。ではこれらのオプションがなければどうなるのでしょうか? デフォルト

ト値を与えていないので、これらの値は全て None になります。これはこれでよいときもあります (デフォルトで None になるようにしているのはこのためです) が、もっときちんと制御したい場合もあります。その要求を満たすために、`optparse` では各目的変数ごとにデフォルトの値を指定できるようにしており、デフォルト値はコマンドラインが解析される前に代入されます。

まず、`verbose/quiet` の例を考えてみましょう。`optparse` に対して、`-q` がない限り `verbose` を 1 に設定させたいなら、以下のようにします:

```
parser.add_option("-v", action="store_true", dest="verbose", default=1)
parser.add_option("-q", action="store_false", dest="verbose")
```

奇妙なことに、上のコードは以下とまったく等価です: Oddly enough, this is exactly equivalent:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose", default=1)
```

これらのコードが等価なのは、オプションの 目的変数 に対してデフォルト値を指定しており、各コードの二つのオプションは同じ目的変数 (`verbose` 変数) を持っているからです。

以下のコード:

```
parser.add_option("-v", action="store_true", dest="verbose", default=0)
parser.add_option("-q", action="store_false", dest="verbose", default=1)
```

を考えてみましょう。繰り返しますが、`verbose` のデフォルト値は 1 です: 特定の目的変数に対するデフォルト値として有効なのは、最後に指定した値です。

## ヘルプを作成する

どのスクリプトでも使うことになる機能の最後の一つは、`optparse` の機能を使ったヘルプメッセージの生成です。やらなければならないことは、オプションを追加する際に `help` 値を指定することだけです。新たなパーザを作成し、ユーザに優しい (ドキュメント付きになっている) オプションを追加していきましょう:

```
usage = "usage: %prog [options] arg1 arg2"
parser = OptionParser(usage=usage)
parser.add_option("-v", "--verbose",
                  action="store_true", dest="verbose", default=1,
                  help="make lots of noise [default]")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose",
                  help="be vewwy quiet (I'm hunting wabbits)")
parser.add_option("-f", "--file", dest="filename",
                  metavar="FILE", help="write output to FILE"),
parser.add_option("-m", "--mode",
                  default="intermediate",
                  help="interaction mode: one of 'novice', "
                        "'intermediate' [default], 'expert'")
```

`optparse` がコマンドラインの解析中に `-h` か `--help` に遭遇するか、または単に `parser.print_help()` を呼び出すと、以下の内容を標準出力に出力します:

```
usage: <yourscript> [options] arg1 arg2

options:
  -h, --help            show this help message and exit
  -v, --verbose          make lots of noise [default]
  -q, --quiet            be vewwy quiet (I'm hunting wabbits)
  -fFILE, --file=FILE    write output to FILE
  -mMODE, --mode=MODE    interaction mode: one of 'novice', 'intermediate'
                        [default], 'expert'
```

optparse に可能な限り親切なヘルプメッセージを出力させる上で役に立つと思われることを以下にざっと述べます:

- スクリプト自体の使用法メッセージ (usage) を以下のようにして定義します:

```
usage = "usage: %prog [options] arg1 arg2"
```

optparse は、使用法メッセージ文字列中の "%prog" に現在のスクリプト名、すなわち `os.path.basename(sys.argv[0])` を展開します。展開された文字列は、詳細なオプションヘルプの前に出力されます。

使用法メッセージ文字列を与えなければ、optparse は面白みのない、とはいえ気の利いたデフォルト値: "usage: %prog [options]" を使います。スクリプトが固定引数を取っていないければ、この値でかまわないでしょう。

- 各オプションはヘルプ文字列を定義していますが、行の折り返しについて悩むことはありません — optparse が行を折り返し、ヘルプメッセージ出力の見栄えがよくなるよう気を配ります。
- 値をとるオプションについては、そのことを示す内容がヘルプメッセージ内に自動的に生成されます。例えば、“mode” オプションは:

```
-mMODE, --mode=MODE
```

となります。ここで、“MODE” はメタ変数と呼ばれます: この値はユーザが **-m/--mode** に対して指定するようスクリプトが期待している引数を表します。デフォルトでは、optparse は目的変数の名前を大文字に変換して、メタ変数として使います。時にこれは期待通りの値になりません — 例えば、上の例の *filename* オプションでは明示的に `metavar="FILE"` を設定しており、その結果自動生成されたオプション説明は以下のようになります:

```
-fFILE, --file=FILE
```

この機能は重要なもので、それは表示スペースを節約するといった理由にとどまりません: 上の例では、手作業で書かれたヘルプテキストの中では、メタ変数として“FILE”を使っています。その結果、ユーザに対して、堅苦しく表現された書法“-fFILE”とくだけた意味付け説明“write output to FILE”が対応しているという情報を与えます。これは、ヘルプテキストをエンドユーザに対してより明快でより有用にする上では、単純でありながら効果的な手法です。

数多くのオプションを扱うときには、オプションをグループ化するとヘルプメッセージ出力をよりよくする上で便利です。OptionParser は複数のオプショングループを収めることができ、各グループは複数のオプションを収めることができます。

上で定義されたパーザに続けて `OptionGroup` を追加するのは簡単です:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk."
                    " It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)
```

このコードは以下のヘルプ出力になります:

```
usage:  [options] arg1 arg2

options:
  -h, --help            show this help message and exit
  -v, --verbose          make lots of noise [default]
  -q, --quiet            be vewwy quiet (I'm hunting wabbits)
  -fFILE, --file=FILE   write output to FILE
  -mMODE, --mode=MODE   interaction mode: one of 'novice', 'intermediate'
                        [default], 'expert'

Dangerous Options:
  Caution: use of these options is at your own risk.  It is believed that
  some of them bite.
  -g                    Group option.
```

## バージョン番号を出力する

要約された使用方法メッセージ文字列と同様に、`optparse` はプログラムのバージョン文字列を出力できます。これを行うには、`version` 引数を `OptionParser` に文字列で渡さなければなりません:

```
parser = OptionParser(usage="%prog [-f] [-q]", version="%prog 1.0")
```

“%prog” が `usage` とまったく同じように展開されることに注意してください。またこれとは別に、`version` には何でも必要なものを入れることができます。`version` を指定すると、`optparse` は自動的に `--version` オプションをパーザに追加します。コマンドライン上でこのオプションが見つかると、`version` 文字列を (“%prog” を置換して) 展開し、標準出力に出力してから終了します。

例えば、スクリプトが `/usr/bin/foo` という名前なら、ユーザは以下のようにするはずで:

```
$ /usr/bin/foo --version
foo 1.0
$
```

## エラー処理

基本的な使い方をする上でもう一つ知っておかなければならないのは、`optparse` がコマンドライン上でエラーに遭遇した — 例えば、`-n` が整数値のオプションの時に `-n4x` が指定された — ときに、どのように振舞うかということです。

`optparse` は標準エラー出力に使用方法メッセージを出力し、有用かつ人間が読めるエラーメッセージを続けます。その後、プログラムをゼロでない終了状態で終了し (`sys.exit()` を呼び出し) ます。

この動作が気に入らないなら、`OptionParser` をサブクラスして `error()` メソッドを上書きしてくだ

さい。 [6.20.5 節](#)、“optparse を拡張する”を参照してください。

### 全てをつなぎ合わせる

optparse を使った私のスクリプトがだいたいどのようになっているかを以下に示します:

```
from optparse import OptionParser

...

def main ():
    usage = "usage: %prog [options] arg"
    parser = OptionParser(usage)
    parser.add_option("-f", "--file", type="string", dest="filename",
                      help="read data from FILENAME")
    parser.add_option("-v", "--verbose",
                      action="store_true", dest="verbose")
    parser.add_option("-q", "--quiet",
                      action="store_false", dest="verbose")
    # more options ...

    (options, args) = parser.parse_args()
    if len(args) != 1:
        parser.error("incorrect number of arguments")

    if options.verbose:
        print "reading %s..." % options.filename

    # go to work ...

if __name__ == "__main__":
    main()
```

## 6.20.3 進んだ使い方

ここからはリファレンスのドキュメントです。まだ [6.20.2 節](#) の基本的なドキュメントを読んでいないなら、まず読んでください。

### パーザを生成してオプションを追加する

パーザにオプションを加えていく方法はいくつかあります。一つ Options からなるリストを Option-Parser のコンストラクタに渡す方法です:

```
parser = OptionParser(option_list=[
    make_option("-f", "--filename",
                action="store", type="string", dest="filename"),
    make_option("-q", "--quiet",
                action="store_false", dest="verbose")])
```

(make\_option() は Option クラスの別名、すなわち Option のコンストラクタを呼び出すだけです。optparse の将来のバージョンでは、おそらく Option を複数のクラスに分割し、make\_option() は正しいクラスを取り出してインスタンス化するファクトリ関数になるはずです。)

長い形式のオプションリストについては、リストを別に生成する方がしばしば便利です:



```
option_list = [make_option("-f", "--filename",
                          action="store", type="string", dest="filename"),
               # 17 other options ...
               make_option("-q", "--quiet",
                          action="store_false", dest="verbose")]
parser = OptionParser(option_list=option_list)
```

また、OptionParser の add\_option() メソッドを使って、オプションを一つ一つ追加することもできます:

```
parser = OptionParser()
parser.add_option("-f", "--filename",
                 action="store", type="string", dest="filename")
parser.add_option("-q", "--quiet",
                 action="store_false", dest="verbose")
```

このメソッドは Option コンストラクタが送出する例外を簡単に追跡できるようにします。例外が発生するのはよくあることで、これは数多くのキーワード引数間で複雑な相互依存関係が発生するためです — うまく指定がされていない場合、optparse は OptionError を送出します。

add\_option() は以下の二つの方法のいずれかで呼び出すことができます:

- (make\_option() が返すような) Option インスタンスを渡す。
- make\_option() (すなわち Option のコンストラクタ) が受理できるような固定引数およびキーワード引数の組み合わせを渡し、Option インスタンスを生成する (上述)。

### オプションを定義する

各 Option インスタンスは、同じ意味を持つ一群のコマンドラインオプション、すなわち同じ意味と効果をもつが、つづりが異なるようなオプションを表します。任意の数の短いオプションと長いオプションを指定することができますが、少なくとも一つのオプション文字列を与えなければなりません。

短い形式のオプション文字列を一つだけもつオプションを定義するには以下のようにします:

```
make_option("-f", ...)
```

長い形式のオプション文字列を一つだけもつオプションを定義するには以下のようにします:

```
make_option("--foo", ...)
```

“...” は、Option オブジェクトの属性を定義する一群のキーワード引数を表しています。ある Option に対してどのようなキーワードを与えれば十分かという問題はかなり複雑です (信じられないというなら、Option の数々の \_check\_\*() メソッド群を調べてみてください) が、だいたい何がしか 指定しなければなりません。うまく指定されていない場合、optparse は OptionError 例外を送出し、犯した間違いを説明してくれます。

オプションの中でもっとも重要な属性はアクション、すなわちコマンドライン中でそのオプションが見つかったときにどうするかを決める属性です。とりうるアクションを以下に挙げます:

#### store

[default] このオプションに対する引数を値として保存します。

**store\_const**

定数値を保存します。

**store\_true**

真を表す値を保存します。

**store\_false**

偽を表す値を保存します。

**append**

このオプションに対する引数をリストに追加します。

**count**

カウンタを 1 増やします。

**callback**

指定された関数を呼び出します。

**help**

全てのオプションとそれに対するドキュメントを含めた、使用法メッセージを出力します。

(アクションを指定しなければ、デフォルト値は “store” になります。このアクションでは、*type* と *dest* キーワードを与えることができます; 以下を参照してください。)

ご覧のとおり、ほとんどのアクションは何らかの値を保存したり更新したりする処理を含んでいます。optparse はこの処理のために、常に特定のオブジェクト (Values クラスのインスタンス) を生成します。オプション引数 (とその他数々の値) は、`make_option()/add_option()` の引数 *dest* (目的変数) に従って、このオブジェクトの属性として保存されます。

例えば、以下の呼び出し:

```
parser.parse_args()
```

を行うと、optparse は最初に行うことのの一つとして、*values* オブジェクトを生成します:

```
values = Values()
```

このパーザのオプションの一つが、以下:

```
make_option("-f", "--file", action="store", type="string", dest="filename")
```

のように定義されており、解析しているコマンドライン中に以下:

```
-ffoo
-f foo
--file=foo
--file foo
```

の文字列が含まれていれば、optparse は **-f** や **--file** オプションを見つけた際に、以下のコード:

```
values.filename = "foo"
```

と同じ処理を行います。明らかに、*type* と *dest* 引数は (大抵) *action* とほぼ同じくらい大事なものです。*action* は \*全ての\* オプションで意味をなす唯一の属性ですが、最も重要な属性でもあります。

### オプションに対するアクション

オプションに対する様々なアクションは全て、わずかに異なる要求事項と効果を持っています。“help” アクションを除き、`Option` を生成する際には少なくとも一つの別のキーワード引数を指定する必要があります; 各アクションに対する厳密な要求事項を以下に挙げます。

#### store

[relevant: *type*, *dest*, *nargs*, *choices*]

オプションの後には必ず引数が続き、*type* に従った値に変換されて *dest* に保存されます。*nargs* > 1 の場合、複数の引数がコマンドラインから取り出されます; 引数は全て *type* に従って変換され、*dest* にタプルとして保存されます。以下の 6.20.3 節、“オプションの型”を参照してください。

*choices* を指定 (文字列のリストかタプル) していれば、型のデフォルト値は “choice” になります。

*type* が指定されなければ、デフォルト値は “string” になります。

*dest* が指定されない場合、`optparse` は目的変数を最初の長い形式のオプション文字列から導出します (例えば、`--foo-bar` → `foo_bar`)。長い形式のオプション文字列がない場合、`optparse` は最初の短い形式のオプションから目的変数名を導出します (例えば `-f` → `f`)。

例:

```
make_option("-f")
make_option("-p", type="float", nargs=3, dest="point")
```

以下のコマンド:

```
-f foo.txt -p 1 -3.5 4 -fbar.txt
```

を与えると、`optparse` は以下のように変数を設定します:

```
values.f = "bar.txt"
values.point = (1.0, -3.5, 4.0)
```

(実際のところ、`values.f` は二度値設定されますが、最終的には二回目の設定値だけが読み取れます。)

#### store\_const

[required: *const*, *dest*]

`Option` コンストラクタに与えられた *const* 値が *dest* に保存されます。

例:

```
make_option("-q", "--quiet",
            action="store_const", const=0, dest="verbose"),
make_option("-v", "--verbose",
            action="store_const", const=1, dest="verbose"),
make_option(None, "--noisy",
            action="store_const", const=2, dest="verbose"),
```

`--noisy` がなければ、`optparse` は以下のように値を設定します:

```
values.verbose = 2
```

#### **store\_true**

[required: *dest*]

“store\_const” の特殊なケースで、真値 (厳密には整数の 1) を *dest* に保存します。

#### **store\_false**

[required: *dest*]

“store\_true” と同じですが、偽値 (整数の 0) を保存します。

例:

```
make_option(None, "--clobber", action="store_true", dest="clobber")
make_option(None, "--no-clobber", action="store_false", dest="clobber")
```

#### **append**

[relevant: *type*, *dest*, *nargs*, *choices*]

オプションの後ろには引数が続かなければなりません。この引数は *dest* に入っているリストに追加されます。*dest* のデフォルト値が指定されていない (すなわち、デフォルト値が `None` である) 場合、`optparse` が最初にコマンドライン中にこのオプションを見つけた際に空のリストが自動的に生成されます。‘*nargs* > 1’ であれば、複数の引数を取り出され、長さ *nargs* のタプルが *dest* に追加されます。

*type* と *dest* のデフォルト値は “store” アクションの値と同じです。

例:

```
make_option("-t", "--tracks", action="append", type="int")
```

`-t3` がコマンドライン上で見つかると、`optparse` は以下と等価な処理を行います:

```
values.tracks = []
values.tracks.append(int("3"))
```

その後で ‘`-tracks=4`’ が見つかると、さらに以下の処理を行います:

```
values.tracks.append(int("4"))
```

`optparse` が ‘`-tracks=x`’ のような指定をどのように扱うかについての情報は、“エラー処理” (6.20.2 節) を参照してください。

#### **count**

[required: *dest*]

*dest* に保持されている整数値をインクリメント (1 増加) させます。*dest* は (デフォルトの値を指定しない限り) 最初にインクリメントを行う前にゼロに設定されます。

例:

```
make_option("-v", action="count", dest="verbosity")
```

最初に `-v` がコマンドライン上で見つかった際、`optparse` は以下と等価な処理:

```
values.verbosity = 0
values.verbosity += 1
```

を行います。その後 `-v` が見つかると、さらに以下の処理:

```
values.verbosity += 1
```

を行います。

### callback

[required: 'callback'; relevant: *type, nargs, callback\_args, callback\_kwargs*]

*callback* に指定された関数を呼び出します。この関数の用法は以下のようになっています:

```
func(option : Option,
      opt : string,
      value : any,
      parser : OptionParser,
      *args, **kwargs)
```

callback オプションについての詳細は、[6.20.4 節](#)、“callback オプション”でカバーされています。

### help

[required: none]

現在のオプションパーザ内にある全てのオプションに対する完全なヘルプメッセージを出力します。ヘルプメッセージは `OptionParser` のコンストラクタに渡された *usage* 文字列と、各オプションに渡された *help* 文字列から生成されます。

オプションに *help* 文字列が指定されていなくても、ヘルプメッセージ中には列挙されます。オプションを完全に表示させないようにするには、特殊な値 `optparse.SUPPRESS_HELP` を使います。

例:

```
from optparse import Option, OptionParser, SUPPRESS_HELP

usage = "usage: %prog [options]"
parser = OptionParser(usage, option_list=[
    make_option("-h", "--help", action="help"),
    make_option("-v", action="store_true", dest="verbose",
                help="Be moderately verbose"),
    make_option("--file", dest="filename",
                help="Input file to read data from"),
    make_option("--secret", help=SUPPRESS_HELP)
```

`optparse` がコマンドライン上に `--h` または `--help` を見つけると、以下のようなヘルプメッセージを標準出力に出力します:

```
usage: <yourscript> [options]

options:
  -h, --help            Show this help message and exit
  -v                    Be moderately verbose
  --file=FILENAME       Input file to read data from
```

ヘルプメッセージの出力後、`optparse` は `sys.exit(0)` でプロセスを終了します。

## version

[required: none]

`OptionParser` に指定されているバージョン番号を標準出力に出力して終了します。バージョン番号は、実際には `OptionParser` の `print_version()` メソッドで書式化されてから出力されます。通常、`OptionParser` のコンストラクタに `version` が指定されたときのみ関係のあるアクションです。

## オプションの型

`optparse` は6種類のすばらしいオプション型: `string` (文字列)、`int` (整数)、`long` (長整数)、`choice` (選択項目)、`float` (浮動小数点数) および `complex` (複素数) をサポートしています。(この中で、`string`、`int`、`float`、および `choice` が最もよく使われます – `long` と `complex` は主にオプション型を完全にそろえるために存在しています。) 新たなオプションの型は、`Option` クラスをサブクラス化することで簡単に追加できます; 6.20.5 節、“`optparse` を拡張する”を参照してください。

文字列オプションの引数は、チェックしたり変換したりはしません: コマンドライン上のテキストが目的変数にそのまま保存されます (またはコールバックに渡されます)。

整数引数は `int()` に渡され、Python 整数に変換されます。`int()` が失敗すると `optparse` も失敗に終わりますが、より役に立つエラーメッセージを伴います。内部的には、`optparse` は `optparse.check_builtin()` で `OptionValueError` を送出します; より高水準 (`OptionParser` 内) では、この例外が捕捉され、`optparse` はエラーメッセージを伴ってプログラムを停止させます。

同様に、浮動小数点数引数は `float()`、長整数引数は `long()`、複素数引数は `complex()` に渡されて変換されます。それ以外では、値は整数引数の場合と同様に扱われます。

選択項目オプションは文字列オプションのサブタイプです。マスタリストや選択項目のリスト (文字列) はオプションコンストラクタ (`make_option()` や `OptionParser.add_option()`) に “choices” キーワード引数として渡されなければなりません。選択項目オプション引数は `optparse.check_choice()` 内でマスタリストと比較され、マスタリストにない文字列が指定された場合には `OptionValueError` が送出されます。

## オプションパーザへのクエリと操作

自前のオプションパーザをつつきまわして、何が起こるかを調べると便利ことがあります。`OptionParser` では便利な二つのメソッドを提供しています:

`has_option(opt_str)`

`-q` や `--verbose` のようなオプション文字列に対して、`OptionParser` がオプションに対応している場合に真を返します。

`get_option(opt_str)`

指定されたオプションを実装している `Option` インスタンスを返します。実装されていない場合は `None` を返します。



```
remove_option(opt_str)
```

`OptionParser` が `opt_str` に対するオプションを持っている場合、そのオプションは削除されます。そのオプションが他の文字列で指定することができる場合、それらのオプション文字列もすべて無効になります。

`opt_str` がこの `OptionParser` オブジェクトのどのオプションにも属さない場合、`ValueError` を送出します。

## オプション間の衝突

注意が足りないと、衝突するオプションを定義しやすくなります:

```
parser.add_option("-n", "--dry-run", ...)
...
parser.add_option("-n", "--noisy", ...)
```

(標準的なオプションを実装済みの `OptionParser` から自前のサブクラスを定義した場合にはさらによく起きます)

`optparse` はこのような定義は通常間違いによるものと仮定しており、デフォルトでは例外 (`OptionConflictError`) を送出します。この誤りは簡単に修正できるプログラム上のエラーなので、例外を catch しようとしないうほうがせず — 間違いを直しておくべきです。

時に、故意に新たなオプションで以前のオプションを置き換えたいことがあります。これは以下の呼び出し:

```
parser.set_conflict_handler("resolve")
```

で行えます。この操作は `optparse` にオプションの衝突を賢く解決するよう指示します。

からくりはこうなっています: オプションを追加するたびに、`optparse` は以前に追加されたオプションとの衝突がないか調べます。何らかの衝突が見つかったら、`OptionParser` のコンストラクタでの指定:

```
parser = OptionParser(..., conflict_handler="resolve")
```

か、`set_conflict_handler()` で指定されたいずれかの衝突処理機構 (conflict-handling mechanism) が起動されます。

デフォルトの衝突処理メカニズムは “error” です。その他の唯一の選択肢は “ignore” で、これはバージョン 1.1 またはそれ以前の `optparse` の (おそらく壊れている) 挙動を修正します。

以下に例を示します: まず、衝突を解決するように設定された `OptionParser` を定義しましょう:

```
parser = OptionParser(conflict_handler="resolve")
```

ここでオプションを全て定義します:

```
parser.add_option("-n", "--dry-run", ..., help="original dry-run option")
...
parser.add_option("-n", "--noisy", ..., help="be noisy")
```

この時点で、`optparse` は以前に追加されたオプションですでに `-n` オプション文字列が使用されてい

ることを検出します。conflict\_handler == "resolve" であるため、この状況は **-n** を以前のオプションにおけるオプション文字列リストから削除することによって解決します。こうして、以前のオプションを有効にする方法は **--dry-run** だけとなります。ユーザがヘルプメッセージを要求すると、ヘルプ文字列は上の結果を反映します。例えば以下ようになります:

```
options:
  --dry-run      original dry-run option
  ...
  -n, --noisy    be noisy
```

以前に追加されたオプション文字列を切り詰めていって何も残らないようにすることは可能ですが、そのオプションをコマンドラインから起動する手段がなくなってしまうので注意してください。この場合、optparse はオプションを完全に除去してしまうので、こうしたオプションはヘルプテキストやその他のいずれにも表示されなくなります。例えば、現在の OptionParser で続けると:

```
parser.add_option("--dry-run", ..., help="new dry-run option")
```

この操作を行った時点で、最初の **-n/--dry-run** オプションはもはやアクセスできなくなり、optparse は最初のオプションを除去します。ユーザがヘルプを要求すると、以下のようなヘルプ:

```
options:
  ...
  -n, --noisy    be noisy
  --dry-run      new dry-run option
```

を読むことになるでしょう。

## 6.20.4 callback オプション

optparse の組み込みのアクションや型が望みにかなったものでなく、とはいえ optparse を拡張して独自のアクションや型を定義するほどではない場合、おそらく callback オプションを定義する必要があるでしょう。callback オプションの定義はきわめて簡単です; 注意を要するのは、よいコールバック関数 (callback function、optparse がオプションをコマンドライン上に見つけた際に呼び出される関数) を書くところです。

### callback オプションを定義する

例によって、コールバックオプションの定義は、直接 Option クラスをインスタンス化するか、OptionParser オブジェクトの add\_option() メソッドを使うことで定義できます。指定しなければならない属性は唯一 *callback* で、呼び出したい関数を指定します:

```
parser.add_option("-c", callback=my_callback)
```

ここでは関数オブジェクトを与えなければならないことに注意してください — つまり、callback オプションを定義する時点で、関数 my\_callback() はすでに定義済みにしていなければなりません。上の単純なケースでは、optparse は **-c** が必要とする引数について何の知識もありません。通常、これはオプションが何ら引数をとらない — コマンドライン上に **-c** が存在するかどうかだけが知りたいことの全てである — ことを意味します。しかし、状況によっては、コールバック関数が任意の数の引数をコマンドライン引数か

ら取り出したいと思うかもしれません。これがコールバック関数の記述を難しくしているところです; これについては本ドキュメントの後の方でカバーします。

オプションの属性を定義する際に指定できる属性はいくつかあります:

#### **type**

通常の意味を持ちます: “store” や “append” アクションのように、`optparse` に対して引数の一つ消費するように指示します。この引数は *type* に変換できなければなりません。`optparse` は値をどこかに保存するのではなく、*type* に変換してコールバック関数に渡します。

#### **nargs**

これも通常の意味を持ちます: この属性が与えられ、`'nargs > 1'` であるばあい、`optparse` は *nargs* 個の引数を取り出します。各値は *type* に変換可能でなくてはなりません。その後、変換された値からなるタプルがコールバック関数に渡されます。

#### **callback\_args**

コールバックに渡す外部の固定引数からなるタプルです。

#### **callback\_kwargs**

コールバックに渡す外部のキーワード引数です。

コールバック関数はどのように呼び出されるか

コールバックは全て以下のようにして呼び出されます:

```
func(option, opt, value, parser, *args, **kwargs)
```

ここで、

#### **option**

は、このコールバックを呼び出している `Option` のインスタンスです。

#### **opt**

は、コールバック呼び出しのきっかけとなったコマンドライン上のオプション文字列です。(長形式のオプションに対する省略形が使われている場合、*opt* は完全な、正式な形のオプション文字列となります — 例えば、ユーザが `--foobar` の短縮形として `--foo` をコマンドラインに入力した時には、*opt* は `--foobar` となります。)

#### **value**

は、このオプションの引数で、コマンドライン上に見つかったものです。`optparse` は、*type* が設定されている場合、単一の引数しかとりません;*value* の型はオプションの型 ( [6.20.3](#), “オプションの型” 参照) で示された型になります。このオプションに対する *type* が `None` である (引数はない) 場合、*value* は `None` になります。`'nargs > 1'` なら、*value* は適切な型をもつ値のタプルとなります。

#### **parser**

は、現在のオプション解析の全てを駆動している `OptionParser` インスタンスです。主に、この値を介して、インスタンス属性としていくつかの興味深いデータにアクセスできるために便利です:

#### **parser.rargs**

現在未処理の引数からなるリスト、すなわち、*opt* (と、あれば *value*) が除去され、それ以降の引数だけが残っているリストです。`parser.rargs` の変更、例えばさらに引数を取り出すといった変更は、自由に行ってかまいません。

### **parser.largs**

現在放置されている引数のセット、すなわち処理はされたが、オプション (やオプションの引数) として取り出されてはいない引数のセットです。parser.largs の変更、例えばさらに引数を追加するといった変更は、自由に行ってかまいません。

### **parser.values**

オプションの値がデフォルトで保存されるオブジェクトです。この値は、コールバック関数がオプションの値を記憶するために、他の optparse と同じ機構を使えるようにするので便利です; これでグローバル変数や閉包 (closure) を台無しにすることがありません。コマンドライン上にすでに現れているオプションの値にアクセスすることもできます。

### **args**

*callback\_args* オプション属性で与えられた任意の固定引数からなるタプルです。

### **kwargs**

*callback\_args* オプション属性で与えられた任意のキーワード引数からなるタプルです。

*args* と *kwargs* は省略可能 (*callback\_args* や *callback\_kwargs* をコールバック定義の際に与えたときのみ渡されます) なので、最小のコールバック関数定義は以下のようになります:

```
def my_callback (option, opt, value, parser):  
    pass
```

### **エラー処理**

オプションやその引数に関して問題が発生した場合、コールバック関数は `OptionValueError` を送出しなければなりません。optparse はこの例外を捕捉し、プログラムを終了します。このとき指定したエラーメッセージを標準出力に出力します。メッセージは明快、簡潔、正確で、問題が見つかったオプションについて触れなければなりません。そうでなければ、ユーザは指定したオプションの何が問題なのか解決するのに苦労するでしょう。

### **例**

以下に引数をとらず、単に見つかったオプションを記録するだけのコールバックオプションの例を示します:

```
def record_foo_seen (option, opt, value, parser):  
    parser.saw_foo = 1  
  
parser.add_option("--foo", action="callback", callback=record_foo_seen)
```

もちろん、“store\_true” アクションを使っても同じことができます。もう少し興味深い例もあります: **-a** がコマンドライン上に見つかった事実を記録するが、**-b** が見つかりと **-a** を消し去るというものです。

```
def check_order (option, opt, value, parser):  
    if parser.values.b:  
        raise OptionValueError("can't use -a after -b")  
    parser.values.a = 1  
    ...  
parser.add_option("-a", action="callback", callback=check_order)  
parser.add_option("-b", action="store_true", dest="b")
```

このコールバックを同様のオプションに対して複数回使いたい(オプションが見つかるたびにフラグが設定されるが、`-b`が見つかるたびに常に消し去られる)場合、少しだけ作業する必要があります: すなわち、エラーメッセージと設定されるフラグを一般化しておかなければなりません。

```
def check_order (option, opt, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use %s after -b" % opt)
    setattr(parser.values, option.dest, 1)
    ...
parser.add_option("-a", action="callback", callback=check_order, dest='a')
parser.add_option("-b", action="store_true", dest="b")
parser.add_option("-c", action="callback", callback=check_order, dest='c')
```

もちろん、ここでは任意の条件を入れることができます — できることは、すでに定義されたオプションの値に対するチェックにとどまりません。例えば、満月 (full moon) にならないと呼び出してはならないオプションがあったとすると、やらなければならない作業は以下ようになります:

```
def check_moon (option, opt, value, parser):
    if is_full_moon():
        raise OptionValueError("%s option invalid when moon full" % opt)
    setattr(parser.values, option.dest, 1)
    ...
parser.add_option("--foo",
                  action="callback", callback=check_moon, dest="foo")
```

(`is_full_moon()` の定義は読者の練習用に残しておきます。)

#### 固定個の引数

決まった数の引数をとるような callback オプションを定義する際には、状況は少し興味深いものとなります。callback オプションが引数をとるよう定義する操作は “store” や “append” オプションと同様です: *type* を定義すると、オプションは単一の引数を取り、この値は指定された型に変換可能でなくてはなりません; 加えて *nargs* を定義すると、オプションは指定数の引数をとるようになります。

標準的な “store” アクションをエミュレートするだけの例を以下に示します:

```
def store_value (option, opt, value, parser):
    setattr(parser.values, option.dest, value)
    ...
parser.add_option("--foo",
                  action="callback", callback=store_value,
                  type="int", nargs=3, dest="foo")
```

`optparse` は 3 つの引数を取り出し、整数に変換しようと配慮することに気を付けてください; これらの値を保存する必要があります (あるいは、何らかの処理を行う必要があります: 明らかに、この例ではコールバックを使う必要はありません。想像力を働かせてください!)

#### 可変個の引数

オプションが可変個の引数をとるようにしたいのなら、状況はやや困難になります。この場合、コールバックを書かなければなりません; `optparse` では可変個の引数に対する組み込みの機能はまったく提供していません。このため、慣習的な UNIX コマンドライン解析のための本格的な書法を扱わなくてはなりません。(以前は、`optparse` はこの問題に対する配慮を行っていましたが、これは不具合がありました。この不具合は、この種のコールバックをより複雑にすることで修正されました。) とりわけ、コールバックでは、裸の `--` や `-` 引数のことを考えなければなりません; やりかたは以下ようになります:

- 裸の `--` が存在し、何らかのオプションの引数でない場合、コマンドライン処理を停止させ、`--` 自体は消失します。
- 裸の `-` も同様にコマンドライン処理を停止させますが、`-` 自体は残されます。
- `--` か `-` のいずれかがオプションに対する引数となり得ます。

オプションが可変個の引数をとるようにさせたいなら、いくつかの巧妙で厄介な問題に配慮しなければなりません。どういう実装をとるかは、アプリケーションでどのようなトレードオフを考慮するかによります (このため、`optparse` では可変個の引数に関する問題を直接的に取り扱わないのです)。

とはいえ、可変個の引数をもつオプションに対するスタブ (stub、仲介インタフェース) を以下に示しておきます:

```
def varargs (option, opt, value, parser):
    assert value is None
    done = 0
    value = []
    rargs = parser.rargs
    while rargs:
        arg = rargs[0]

        # Stop if we hit an arg like "--foo", "-a", "-fx", "--file=f",
        # etc. Note that this also stops on "-3" or "-3.0", so if
        # your option takes numeric values, you will need to handle
        # this.
        if ((arg[:2] == "--" and len(arg) > 2) or
            (arg[:1] == "-" and len(arg) > 1 and arg[1] != "-")):
            break
        else:
            value.append(arg)
            del rargs[0]

    setattr(parser.values, option.dest, value)

...
parser.add_option("-c", "--callback",
                  action="callback", callback=varargs)
```

この実装固有の弱点は、`-c` 以後に続いて負の数を表す引数があった場合、`-c` の引数ではなく次のオプションとして解釈されるということです。この問題の修正は読者の練習課題としておきます。

### 6.20.5 `optparse` を拡張する

`optparse` がコマンドラインオプションを解析する手法における二つの主要な制御要素は、各オプションに対するアクションと型です。このため、ほとんどの拡張の方向性は、新たなアクションか型の追加になっています。

また、拡張例の節には `optparse` に対する異なったやり方での拡張を実演したものが収められています: 例えば、大小文字の区別をしないオプションパーザや、“必須オプション”を実装している二種類のオプションパーザです。

#### 新たな型を追加する

新たな型を追加するには、`optparse` の `Option` クラスから自前のサブクラスを定義する必要があります。このクラスは `optparse` の型を定義する二つの属性: `TYPES` と `TYPE_CHECKER` を持っています。



TYPES は型の名前からなるタプルです; サブクラスでは、標準のタプルに基づいて新たなタプル TYPES を定義するだけです。

TYPE\_CHECKER は型の名前を型チェック関数に対応付ける辞書です。型チェック関数は以下の形式をとります:

```
def check_foo (option : Option, opt : string, value : string)
    -> foo
```

この関数はどんな名前にしてもよく、どんな型を返すようにもできます。型チェック関数が返す値は OptionParser.parse\_args() が返す OptionValues インスタンスになるか、value パラメタとしてコールバック関数に渡されます。

自前の型チェック関数が問題に遭遇した場合、OptionValueError を送出しなければなりません。OptionValueError は単一の文字列引数を取り、この引数はそのまま OptionParser の error() メソッドに渡されます。error() メソッドはプログラム名と文字列 “error:” を渡された文字列の前に付け、プロセスを終了する前に全てを標準エラー出力に出力します。

以下のつまらない例では、コマンドライン上の Python 形式の複素数を解析するために “複素数 (complex)” 型を追加する操作を実演します。(この例は今ではさらにつまらなくなっていました。というのは、optparse 1.3 からは組み込みで [純粋に完備性のためだけに] 複素数のサポートが追加されるようになったからです。ですが、気にしないことにします。)

まず、必要な import を行います:

```
from copy import copy
from optparse import Option, OptionValueError
```

まず型チェック関数を定義する必要があります。というのは、型チェック関数は後で (自前の Option サブクラスにおける TYPE\_CHECKER 属性で) 参照されるからです:

```
def check_complex (option, opt, value):
    try:
        return complex(value)
    except ValueError:
        raise OptionValueError(
            "option %s: invalid complex value: %r" % (opt, value))
```

最後に、Option サブクラスを定義します:

```
class MyOption (Option):
    TYPES = Option.TYPES + ("complex",)
    TYPE_CHECKER = copy(Option.TYPE_CHECKER)
    TYPE_CHECKER["complex"] = check_complex
```

(Option.TYPE\_CHECKER の copy() を作成しなければ、optparse の Option クラスにおける TYPE\_CHECKER 属性を変更するだけで済みます。Python では、良いマナーと常識以外にこうした省略をとどめるものはありません。)

これで完成です! もう、新たなオプション型を使うようなスクリプトを他の optparse ベースのスクリプトと同じように書くことができます。ただし、OptionParser に Option ではなく MyOption を使うように指示しなければなりません:

```
parser = OptionParser(option_class=MyOption)
parser.add_option("-c", action="store", type="complex", dest="c")
```

別の方法として、自前のオプションリストを構築して、OptionParser に渡すこともできます; 上の方法で add\_option() を使わなければ、OptionParser にどのオプションクラスを使うかを教える必要はありません:

```
option_list = [MyOption("-c", action="store", type="complex", dest="c")]
parser = OptionParser(option_list=option_list)
```

### 新たなアクションを追加する

アクションの追加は少しだけ厄介です。というのは、optparse がアクションを二種類に分類していることを理解しなければならないからです:

#### "store" アクション

結果的に optparse の OptionValues インスタンスの属性に値を保存することになるアクションです; このアクションをとるオプションは Option のコンストラクタに 'dest' 属性を指定する必要があります。

#### "typed" アクション

コマンドラインから値を取り、それが特定の型か、あるいは特定の型に変換される文字列になるはずのアクションです。このアクションをとるオプションは Option のコンストラクタに 'type' 属性を指定する必要があります。

デフォルトの “store” アクションには、“store”、“store\_const”、“append”、および “count” があります。デフォルトの “typed” アクションは、“store”、“append”、および “callback” です。

アクションを追加する際、アクションが “store”、“typed”、どちらでもない、両方である、のいずれか決めなければなりません。Option (または Option サブクラス) の 3 つのクラス属性がこの決定を制御します:

#### ACTIONS

すべてのアクションは ACTIONS に文字列で列挙されていなければなりません。

#### STORE\_ACTIONS

“store” 系のアクションはここに列挙されていなければなりません。

#### TYPED\_ACTIONS

“typed” 系のアクションはここに列挙されていなければなりません。

自前の新たなアクションを実際に実装するには、Option の take\_action() メソッドを上書きして、自前のアクションを認識する場合を追加しなければなりません。

例として、“extend” アクションを追加してみましょう。このアクションは標準の “append” アクションに似ていますが、コマンドラインから単一の値をとって既存のリストに追加する代わりに、“extend” ではコマンドで区切られた単一の文字列から複数の値を取り、既存のリストに追加します。すなわち、--names が “extend” のタイプのオプション文字列であれば、以下のコマンドライン:

```
--names=foo,bar --names blah --names ding,dong
```

は以下のリスト:

```
["foo", "bar", "blah", "ding", "dong"]
```

になります。

再び `Option` のサブクラスを定義します:

```
class MyOption (Option):

    ACTIONS = Option.ACTIONS + ("extend",)
    STORE_ACTIONS = Option.STORE_ACTIONS + ("extend",)
    TYPED_ACTIONS = Option.TYPED_ACTIONS + ("extend",)

    def take_action (self, action, dest, opt, value, values, parser):
        if action == "extend":
            lvalue = value.split(",")
            values.ensure_value(dest, []).extend(lvalue)
        else:
            Option.take_action(
                self, action, dest, opt, value, values, parser)
```

以下のことに注意します:

- “extend” は値をコマンドラインから要求し、その値をどこかへ保存するという両方の操作を行うので、`STORE_ACTIONS` と `TYPED_ACTIONS` の両方に相当します。
- `MyOption.take_action()` ではこの新しいアクションだけを実装し、標準の `Option.take_action()` で `optparse` アクションへ制御を戻します。
- `values` は `Values` クラスのインスタンスで、非常に便利な `ensure_value()` メソッドを提供しています。`ensure_value()` は実質的には安全値 (safety value) のある `getattr()` です; このメソッドは以下のようにして呼び出されます:

```
values.ensure_value(attr, value)
```

`values` の `attr` 属性が存在しないか、または `None` の場合、`ensure_value()` は属性の値を `value` に設定してから `value` を返します。この機能は “extend”、“append”、および “count” で便利です。というのは、これらのアクションはすべて、データを変数に蓄積し、その変数が特定の型であるはず (最初の二つはリストで、最後の一つは整数) だからです。`ensure_value()` を使うことで、アクションを使うスクリプトにおいて、オプションの目的変数にデフォルトの値を設定するよう配慮しなくてよくなります; こうした目的変数はデフォルトを `None` にしておいてよく、`ensure_value()` が必要なときに正しい値にするよう面倒を見ることになります。

拡張を行う他の理由 `optparse`

新たな型やアクションの追加は、`optparse` を拡張しようとする大きな、かつ明白な理由です。その他に少なくとも二つの分野がかかわってくると考えています。

一つ目の理由は簡単なものです: `OptionParser` は必要に応じて、すなわちコマンドラインにエラーが生じたときはユーザがヘルプを要求した際に `sys.exit()` を呼び出して役に立とうとします。前者の場合、スクリプトをクラッシュさせてトレースバックを出力する伝統的なやり方は受け入れがたいものです; このような応答を行うと、ユーザはコマンドラインエラーを起こしたときにスクリプトにバグがあるのだと考えさせてしまいます。後者の場合、一般的にはヘルプを出力した後に継続する利点はあまりありません。

この動作が目障りなら、“修正”するのはさほど難しいことではありません。以下の実装をおこなわねばなりません。

1. `OptionParser` をサブクラス化して `error()` メソッドを上書きします
2. `Option` をサブクラス化して `take_action()` メソッドを上書きします— アクション "help" の処理を自前で提供し、`sys.exit()` を呼ばない用にする必要があるでしょう

もう一つの理由は、より複雑ですが、`optparse` で実装されているコマンドライン文法をオーバーライドするということです。この場合、オプションに対するアクションと型の機構はすべて捨ておき、`sys.argv` を処理するコードを上書きします。いずれの場合も、`OptionParser` をサブクラス化する必要があります; どれくらい徹底的に書き換えたいかによって、`parse_args()`、`_process_long_opt()`、および `_process_short_opts()` について、一つからすべてのメソッドを上書きする必要があるでしょう。

これらは両方とも、読者の練習課題として残しておきます。実は私は両方とも自分では書いたことがありません。というも、`optparse` のデフォルトの (自然な) 動作で十分ハッピーでいられるからです。

Happy hacking、そしてこの言葉を忘れないでください: ルークよ、source を使え (Use the Source, Luke)。

## 拡張の例

以下に `optparse` モジュールを拡張する例をいくつか示します。

まず、オプション解析で大小文字の区別をしないように変更しましょう:

```
from optparse import Option, OptionParser, _match_abbrev

# This case-insensitive option parser relies on having a
# case-insensitive dictionary type available. Here's one
# for Python 2.2. Note that a *real* case-insensitive
# dictionary type would also have to implement __new__(),
# update(), and setdefault() -- but that's not the point
# of this exercise.

class caseless_dict (dict):
    def __setitem__ (self, key, value):
        dict.__setitem__(self, key.lower(), value)

    def __getitem__ (self, key):
        return dict.__getitem__(self, key.lower())

    def get (self, key, default=None):
        return dict.get(self, key.lower())

    def has_key (self, key):
        return dict.has_key(self, key.lower())

class CaselessOptionParser (OptionParser):

    def _create_option_list (self):
        self.option_list = []
        self._short_opt = caseless_dict()
        self._long_opt = caseless_dict()
        self._long_opts = []
        self.defaults = {}

    def _match_long_opt (self, opt):
        return _match_abbrev(opt.lower(), self._long_opt.keys())
```

```

if __name__ == "__main__":
    from optik.errors import OptionConflictError

    # test 1: no options to start with
    parser = CaselessOptionParser()
    try:
        parser.add_option("-H", dest="blah")
    except OptionConflictError:
        print "ok: got OptionConflictError for -H"
    else:
        print "not ok: no conflict between -h and -H"

    parser.add_option("-f", "--file", dest="file")
    #print `parser.get_option("-f")`
    #print `parser.get_option("-F")`
    #print `parser.get_option("--file")`
    #print `parser.get_option("--filE")`
    (options, args) = parser.parse_args(["--FiLe", "foo"])
    assert options.file == "foo", options.file
    print "ok: case insensitive long options work"

    (options, args) = parser.parse_args(["-F", "bar"])
    assert options.file == "bar", options.file
    print "ok: case insensitive short options work"

```

次に、`optparse` で“必須オプション”を実装する方法を二つ挙げます。

その 1: アプリケーションが引数を解析した後に必ず呼び出すメソッドを `OptionParser` に追加します:

`_1.py`

その 2: `Option` を拡張して `required` 属性を追加します; `OptionParser` を拡張して、必須オプションが引数解析語に存在することを確認させます:

`_2.py`

## 6.21 `tempfile` — 一時的なファイルやディレクトリの生成

このモジュールを使うと、一時的なファイルやディレクトリを生成できます。このモジュールはサポートされている全てのプラットフォームで利用可能です。

バージョン 2.3 の Python では、このモジュールに対してセキュリティを高める為の見直しが行われました。現在では新たに 3 つの関数、`NamedTemporaryFile()`、`mkstemp()`、および `mkdtemp()` が提供されており、安全でない `mktemp` を使いつづける必要をなくしました。このモジュールで生成される一時ファイルはもはやプロセス番号を含みません; その代わりに、6 桁のランダムな文字からなる文字列が使われます。

また、ユーザから呼び出し可能な関数は全て、一時ファイルの場所や名前を直接操作できるようにするための追加の引数をとるようになりました。もはや変数 `tempdir` および `template` を使う必要はありません。以前のバージョンとの互換性を維持するために、引数の順番は多少変です; 明確さのためにキーワード引数を使うことをお勧めします。

このモジュールではユーザから呼び出し可能な以下の関数を定義しています:

`TemporaryFile([mode='w+b'] [, bufsize=-1] [, suffix] [, prefix] [, dir])`

一時的な記憶領域として使うことができるファイル (またはファイル類似の) オブジェクトを返しま

す。ファイルは `mkstemp` を使って生成されます。このファイルは閉じられると (オブジェクトがガーベジコレクションされた際に、暗黙のうちに閉じられる場合を含みます) すぐに消去されます。UNIX 環境では、ファイルが生成されるとすぐにそのファイルのディレクトリエントリは除去されてしまいます。一方、他のプラットフォームではこの機能はサポートされていません; 従って、コードを書くときには、この関数で作成した一時ファイルをファイルシステム上で見ることができる、あるいはできないということをあてにすべきではありません。

生成されたファイルを一旦閉じなくてもファイルを読み書きできるようにするために、`mode` パラメータは標準で `'w+b'` に設定されています。ファイルに記録するデータが何であるかに関わらず全てのプラットフォームで一貫性のある動作をさせるために、バイナリモードが使われています。`bufsize` の値は標準で `-1` で、これはオペレーティングシステムにおける標準の値を使うことを意味しています。

`dir`、`prefix` および `suffix` パラメータは `mkstemp()` に渡されます。

`NamedTemporaryFile([mode='w+b'], [bufsize=-1], [suffix], [prefix], [dir])`

この関数はファイルがファイルシステム上で見ることができるよう保証されている点を除き、`TemporaryFile()` と全く同じに働きます。(UNIX では、ディレクトリ エントリは `unlink` されません) ファイル名はファイルオブジェクトの `name` メンバから取得することができます。このファイル名を使って一時ファイルをもう一度開くことができるかどうかは、プラットフォームによって異なります。(UNIX では可能でしたが、Windows NT 以降では開く事ができません。)

2.3 で追加された仕様です。

`mkstemp([suffix], [prefix], [dir], [text=False])`

可能な限り最も安全な手段で一時ファイルを生成します。使用するプラットフォームで `os.open()` の `O_EXCL` フラグが正しく実装されている限り、ファイルの生成で競合条件が起こることはありません。このファイルは、ファイルを生成したユーザのユーザ ID からのみ読み書き可能です。使用するプラットフォームにおいて、ファイルを実行可能かどうかを示す許可ビットが使われている場合、ファイルは誰からも実行不可なように設定されます。このファイルのファイル記述子は子プロセスに継承されません。

`TemporaryFile()` と違って、`mkstemp()` で生成されたファイルが用済みになったときにファイルを消去するのはユーザの責任です。

`suffix` が指定された場合、ファイル名は指定された拡張子で終わります。そうでない場合には拡張子は付けられません。`mkstemp()` はファイル名と拡張子の間にドットを追加しません; 必要なら、`suffix` の先頭につけてください。

`prefix` が指定された場合、ファイル名は指定されたプレフィクス (接頭文字列) で始まります; そうでない場合、標準のプレフィクスが使われます。

`dir` が指定された場合、一時ファイルは指定されたディレクトリ下に作成されます; そうでない場合、標準のディレクトリが使われます。

`text` が指定された場合、ファイルをバイナリモード (標準の設定) かテキストモードで開くかを示します。使用するプラットフォームによってはこの値を設定しても変化はありません。

`mkstemp()` は開かれたファイルを扱うための OS レベルの値とファイルの絶対パス名が順番に並んだタプルを返します。2.3 で追加された仕様です。

`mkdtemp([suffix], [prefix], [dir])`

可能な限り安全な方法で一時ディレクトリを作成します。ディレクトリの生成で競合条件は発生しません。ディレクトリを作成したユーザ ID だけが、このディレクトリに対して内容を読み出したり、書き込んだり、検索したりすることができます。

`mkdtemp()` によって作られたディレクトリとその内容が用済みになった時、にそれを消去するのはユーザの責任です。



*prefix*、*suffix*、および *dir* 引数は `mkstemp()` のものと同じです。

`mkdtemp()` は新たに生成されたディレクトリの絶対パス名を返します。2.3 で追加された仕様です。

`mktemp([suffix][, prefix][, dir])`

リリース 2.3 以降で撤廃された仕様です。Use `mkstemp()` instead.

一時ファイルの絶対パス名を返します。このパス名は少なくともこの関数が呼び出された時点ではファイルシステム中に存在しなかったパス名です。*prefix*、*prefix*、*suffix*、および *dir* 引数は `mkstemp()` のものと同じです。

警告: この関数を使うとプログラムのセキュリティホールになる可能性があります。この関数が返したファイル名を返した後、あなたがそのファイル名を使って次に何かをしようとする段階に至る前に、誰か他の人間があなたにパンチをくらわせてしまうかもしれません。

このモジュールでは、一時的なファイル名の作成方法を指定する 2 つのグローバル変数を使います。これらの変数は上記のいずれかの関数を最初に呼び出した際に初期化されます。関数呼び出しをおこなうユーザはこれらの値を変更することができますが、これはお勧めできません; その代わりに関数に適切な引数を指定してください。

`tempdir`

この値が `None` 以外に設定された場合、このモジュールで定義されている関数全ての *dir* 引数に対する標準の設定値となります。

`tempdir` が設定されていないか `None` の場合、上記のいずれかの関数を呼び出した際は常に、Python は標準的なディレクトリ候補のリストを検索し、関数を呼び出しているユーザの権限でファイルを作成できる最初のディレクトリ候補を `tempdir` に設定します。リストは以下のようになっています:

1. 環境変数 `TMPDIR` で与えられているディレクトリ名。
2. 環境変数 `TEMP` で与えられているディレクトリ名。
3. 環境変数 `TMP` で与えられているディレクトリ名。
4. プラットフォーム依存の場所:
  - Macintosh では 'Temporary Items' フォルダ。
  - RiscOS では環境変数 `Wimp$ScrapDir` で与えられているディレクトリ名。
  - Windows ではディレクトリ '`C:\TEMP`'、'`C:\TMP`'、'`\TEMP`'、および '`\TMP`' の順。
  - その他の全てのプラットフォームでは、'`/tmp`'、'`/var/tmp`'、および '`/usr/tmp`' の順。
5. 最後の手段として、現在の作業ディレクトリ。

`gettempdir()`

現在選択されている、テンポラリファイルを作成するためのディレクトリを返します。`tempdir` が `None` でない場合、単にその内容を返します; そうでない場合には上で記述されている検索が実行され、その結果が返されます。

`template`

リリース 2.0 以降で撤廃された仕様です。代わりに `gettempprefix()` を使ってください。

この値に `None` 以外の値を設定した場合、`mktemp()` が返すファイル名のディレクトリ部を含まない先頭部分 (プレフィクス) を定義します。ファイル名を一意にするために、6 つのランダムな文字および数字がこのプレフィクスの後に追加されます。Windows では、標準のプレフィクスは '`~T`' です; 他のシステムでは '`tmp`' です。

このモジュールの古いバージョンでは、`os.fork()` を呼び出した後に `template` を `None` に設定することが必要でした; この仕様はバージョン 1.5.2 からは必要なくなりました。

`gettempprefix()`

一時ファイル生成の際に使われるファイル名の先頭部分を返します。この先頭部分にはディレクトリ部は含まれません。変数 *template* を直接読み出すよりもこの関数を使うことを勧めます。1.5.2で追加された仕様です。

## 6.22 `errno` — 標準の `errno` システムシンボル

このモジュールから標準の `errno` システムシンボルを取得することができます。個々のシンボルの値は `errno` に対応する整数値です。これらのシンボルの名前は、`'linux/include/errno.h'` から借用されており、かなり網羅的なはずですが。

**errorcode**

`errno` 値を背後のシステムにおける文字列表現に対応付ける辞書です。例えば、`errno.errorcode[errno.EPERM]` は `'EPERM'` に対応付けられます。

数値のエラーコードをエラーメッセージに変換するには、`os.strerror()` を使ってください。

以下のリストの内、現在のプラットフォームで使われていないシンボルはモジュール上で定義されていません。定義されているシンボルだけを挙げたリストは `errno.errorcode.keys()` として取得することができます。取得できるシンボルには以下のようなものがあります:

**EPERM**

許可されていない操作です (Operation not permitted)

**ENOENT**

ファイルまたはディレクトリがありません (No such file or directory)

**ESRCH**

指定したプロセスが存在しません (No such process)

**EINTR**

割り込みシステムコールです (Interrupted system call)

**EIO**

I/O エラーです (I/O error)

**ENXIO**

そのようなデバイスまたはアドレスはありません (No such device or address)

**E2BIG**

引数リストが長すぎます (Arg list too long)

**ENOEXEC**

実行形式にエラーがあります (Exec format error)

**EBADF**

ファイル番号が間違っています (Bad file number)

**ECHILD**

子プロセスがありません (No child processes)

**EAGAIN**

再試行してください (Try again)

**ENOMEM**

空きメモリがありません (Out of memory)

**EACCES**

許可がありません (Permission denied)

**EFAULT**  
不正なアドレスです (Bad address)

**ENOTBLK**  
ブロックデバイスが必要です (Block device required)

**EBUSY**  
そのデバイスまたは資源は使用中です (Device or resource busy)

**EEXIST**  
ファイルがすでに存在します (File exists)

**EXDEV**  
デバイス間のリンクです (Cross-device link)

**ENODEV**  
そのようなデバイスはありません (No such device)

**ENOTDIR**  
ディレクトリではありません (Not a directory)

**EISDIR**  
ディレクトリです (Is a directory)

**EINVAL**  
無効な引数です (Invalid argument)

**ENFILE**  
ファイルテーブルがオーバーフローしています (File table overflow)

**EMFILE**  
開かれたファイルが多すぎます (Too many open files)

**ENOTTY**  
タイプライタではありません (Not a typewriter)

**ETXTBSY**  
テキストファイルが使用中です (Text file busy)

**EFBIG**  
ファイルが大きすぎます (File too large)

**ENOSPC**  
デバイス上に空きがありません (No space left on device)

**ESPIPE**  
不正なシークです (Illegal seek)

**EROFS**  
読み出し専用ファイルシステムです (Read-only file system)

**EMLINK**  
リンクが多すぎます (Too many links)

**EPIPE**  
パイプが壊れました (Broken pipe)

**EDOM**  
数学引数が関数の定義域を越えています (Math argument out of domain of func)

**ERANGE**  
表現できない数学演算結果になりました (Math result not representable)

**EDEADLK**

リソースのデッドロックが起きます (Resource deadlock would occur)

**ENAMETOOLONG**  
ファイル名が長すぎます (File name too long)

**ENOLCK**  
レコードロッキングが利用できません (No record locks available)

**ENOSYS**  
実装されていない機能です (Function not implemented)

**ENOTEMPTY**  
ディレクトリが空ではありません (Directory not empty)

**ELOOP**  
これ以上シンボリックリンクを追跡できません (Too many symbolic links encountered)

**EWOLDBLOCK**  
操作がブロックします (Operation would block)

**ENOMSG**  
指定された型のメッセージはありません (No message of desired type)

**EIDRM**  
識別子が除去されました (Identifier removed)

**ECHRNG**  
チャネル番号が範囲を超えました (Channel number out of range)

**EL2NSYNC**  
レベル 2 で同期がとれていません (Level 2 not synchronized)

**EL3HLT**  
レベル 3 で終了しました (Level 3 halted)

**EL3RST**  
レベル 3 でリセットしました (Level 3 reset)

**ELNRNG**  
リンク番号が範囲を超えています (Link number out of range)

**EUNATCH**  
プロトコルドライバが接続されていません (Protocol driver not attached)

**ENOCSI**  
CSI 構造体がありません (No CSI structure available)

**EL2HLT**  
レベル 2 で終了しました (Level 2 halted)

**EBADE**  
無効な変換です (Invalid exchange)

**EBADR**  
無効な要求記述子です (Invalid request descriptor)

**EXFULL**  
変換テーブルが一杯です (Exchange full)

**ENOANO**  
陰極がありません (No anode)

**EBADRQC**  
無効なリクエストコードです (Invalid request code)

<b>EBADSLT</b>	無効なスロットです (Invalid slot)
<b>EDEADLOCK</b>	ファイルロックにおけるデッドロックエラーです (File locking deadlock error)
<b>EBFONT</b>	フォントファイル形式が間違っています (Bad font file format)
<b>ENOSTR</b>	ストリーム型でないデバイスです (Device not a stream)
<b>ENODATA</b>	利用可能なデータがありません (No data available)
<b>ETIME</b>	時間切れです (Timer expired)
<b>ENOSR</b>	streams リソースを使い切りました (Out of streams resources)
<b>ENONET</b>	計算機はネットワーク上にありません (Machine is not on the network)
<b>ENOPKG</b>	パッケージがインストールされていません (Package not installed)
<b>EREMOTE</b>	対象物は遠隔にあります (Object is remote)
<b>ENOLINK</b>	リンクが切られました (Link has been severed)
<b>EADV</b>	Advertise エラーです (Advertise error)
<b>ESRMNT</b>	Srmount エラーです (Srmount error)
<b>ECOMM</b>	送信時の通信エラーです (Communication error on send)
<b>EPROTO</b>	プロトコルエラーです (Protocol error)
<b>EMULTIHOP</b>	多重ホップを試みました (Multihop attempted)
<b>EDOTDOT</b>	RFS 特有のエラーです (RFS specific error)
<b>EBADMSG</b>	データメッセージではありません (Not a data message)
<b>EOVERFLOW</b>	定義されたデータ型にとって大きすぎる値です (Value too large for defined data type)
<b>ENOTUNIQ</b>	名前がネットワーク上で一意ではありません (Name not unique on network)
<b>EBADFD</b>	ファイル記述子の状態が不正です (File descriptor in bad state)
<b>EREMCHG</b>	

遠隔のアドレスが変更されました (Remote address changed)

**ELIBACC**

必要な共有ライブラリにアクセスできません (Can not access a needed shared library)

**ELIBBAD**

壊れた共有ライブラリにアクセスしています (Accessing a corrupted shared library)

**ELIBSCN**

a.out の .lib セクションが壊れています (.lib section in a.out corrupted)

**ELIBMAX**

リンクを試みる共有ライブラリが多すぎます (Attempting to link in too many shared libraries)

**ELIBEXEC**

共有ライブラリを直接実行することができません (Cannot exec a shared library directly)

**EILSEQ**

不正なバイト列です (Illegal byte sequence)

**ERESTART**

割り込みシステムコールを復帰しなければなりません (Interrupted system call should be restarted)

**ESTRPIPE**

ストリームパイプのエラーです (Streams pipe error)

**EUSERS**

ユーザが多すぎます (Too many users)

**ENOTSOCK**

非ソケットに対するソケット操作です (Socket operation on non-socket)

**EDESTADDRREQ**

目的アドレスが必要です (Destination address required)

**EMSGSIZE**

メッセージが長すぎます (Message too long)

**EPROTOTYPE**

ソケットに対して不正なプロトコル型です (Protocol wrong type for socket)

**ENOPROTOPT**

利用できないプロトコルです (Protocol not available)

**EPROTONOSUPPORT**

サポートされていないプロトコルです (Protocol not supported)

**ESOCKTNOSUPPORT**

サポートされていないソケット型です (Socket type not supported)

**EOPNOTSUPP**

通信端点に対してサポートされていない操作です (Operation not supported on transport endpoint)

**EPFNOSUPPORT**

サポートされていないプロトコルファミリです (Protocol family not supported)

**EAFNOSUPPORT**

プロトコルでサポートされていないアドレスファミリです (Address family not supported by protocol)

**EADDRINUSE**

アドレスは使用中です (Address already in use)

**EADDRNOTAVAIL**

要求されたアドレスを割り当てできません (Cannot assign requested address)



**ENETDOWN**

ネットワークがダウンしています (Network is down)

**ENETUNREACH**

ネットワークに到達できません (Network is unreachable)

**ENETRESET**

リセットによってネットワーク接続が切られました (Network dropped connection because of reset)

**ECONNABORTED**

ソフトウェアによって接続が終了されました (Software caused connection abort)

**ECONNRESET**

接続がピアによってリセットされました (Connection reset by peer)

**ENOBUFS**

バッファに空きがありません (No buffer space available)

**EISCONN**

通信端点がすでに接続されています (Transport endpoint is already connected)

**ENOTCONN**

通信端点が接続されていません (Transport endpoint is not connected)

**ESHUTDOWN**

通信端点のシャットダウン後は送信できません (Cannot send after transport endpoint shutdown)

**ETOOMANYREFS**

参照が多すぎます: 接続できません (Too many references: cannot splice)

**ETIMEDOUT**

接続がタイムアウトしました (Connection timed out)

**ECONNREFUSED**

接続を拒否されました (Connection refused)

**EHOSTDOWN**

ホストはシステムダウンしています (Host is down)

**EHOSTUNREACH**

ホストへの経路がありません (No route to host)

**EALREADY**

すでに処理中です (Operation already in progress)

**EINPROGRESS**

現在処理中です (Operation now in progress)

**ESTALE**

無効な NFS ファイルハンドルです (Stale NFS file handle)

**EUCLEAN**

(Structure needs cleaning)

**ENOTNAM**

XENIX 名前付きファイルではありません (Not a XENIX named type file)

**ENAVAIL**

XENIX セマフォは利用できません (No XENIX semaphores available)

**EISNAM**

名前付きファイルです (Is a named type file)

**EREMOTEIO**

遠隔側の I/O エラーです (Remote I/O error)

EDQUOT

ディスククォータを超えました (Quota exceeded)

## 6.23 glob — UNIX 形式のパス名のパターン展開

glob モジュールは UNIX シェルで使われているルールに従って指定されたパターンにマッチするすべてのパス名を見つけ出します。チルダ展開は使えませんが、\*、?と[]で表される文字範囲には正しくマッチします。これは os.listdir() 関数と fnmatch.fnmatch() 関数を一緒に使って実行されていて、実際に subshell を呼び出しているわけではありません。(チルダ展開とシェル変数展開を利用したければ、os.path.expand() と os.path.expandvars() を使ってください。)

glob(pathname)

pathname (パスの指定を含んだ文字列でなければいけません。)にマッチする空の可能性のあるパス名のリストを返します。

pathname は ('usr/src/Python-1.5/Makefile' のように) 絶対パスでもいいし、('./../Tools/\*.gif' のように) 相対パスでもよくて、シェル形式のワイルドカードを含んでいてもかまいません。

たとえば、次のファイルだけがあるディレクトリを考えてください: '1.gif'、'2.txt'、and 'card.gif'。glob() は次のような結果になります。パスに接頭するどの部分が保たれているかに注意してください。

```
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
```

参考資料:

fnmatch モジュール (6.24 節):

シェル形式の (パスではない) ファイル名展開

## 6.24 fnmatch — UNIX ファイル名のパターンマッチ

このモジュールは UNIX のシェル形式のワイルドカードへの対応を提供しますが、(re モジュールでドキュメント化されている) 正規表現と同じではありません。シェル形式のワイルドカードで使われる特別な文字は、

Pattern	Meaning
*	すべてにマッチします
?	任意の一文字にマッチします
[seq]	seq にある任意の文字にマッチします
[!seq]	seq にない任意の文字にマッチします

ファイル名のセパレーター (UNIX では '/' ) はこのモジュールに固有なものではないことに注意してください。パス名展開については、glob モジュールを参照してください (glob はパス名の部分にマッチさせるのに fnmatch() を使っています)。同様に、ピリオドで始まるファイル名はこのモジュールに固有ではなくて、\* と ? のパターンでマッチします。

**fnmatch**(*filename*, *pattern*)

*filename* の文字列が *pattern* の文字列にマッチするかテストして、真、偽のいずれかを返します。もしもオペレーティングシステムが大文字、小文字を区別するなら、比較を行う前に、すべて大文字、あるいは小文字に標準化されます。オペレーティングシステムに標準かどうかは別にして、大文字、小文字を区別する比較が必要なときには、かわりに `fnmatchcase()` を使ってください。

**fnmatchcase**(*filename*, *pattern*)

*filename* が *pattern* にマッチするかテストして、真、偽を返します。比較は大文字、小文字を区別します。

**filter**(*names*, *pattern*)

*pattern* にマッチする *names* のリストの部分集合を返します。[`n for n in names if fnmatch(n, pattern)`] と同じですが、もっと効率よく実装しています。2.2 で追加された仕様です。

参考資料:

glob モジュール (6.23 節):

UNIX シェル形式のパス展開。

## 6.25 shutil — 高レベルなファイル操作

shutil モジュールはファイルやファイルの収集に関する多くの高レベルな操作方法を提供します。特にファイルのコピーや削除のための関数が用意されています。

注意: MacOS においてはリソースフォークや他のメタデータは取り扱うことができません。

つまり、ファイルをコピーする際にこれらのリソースは失われたり、ファイルタイプや作成者コードは正しく認識されないことを意味します。

**copyfile**(*src*, *dst*)

*src* で指定されたファイル内容を *dst* で指定されたファイルへとコピーします。もし *dst* が存在したら置き換えられ、そうでなければ新規作成されます。キャラクタやブロックデバイス、パイプ等の特別なファイルはこの関数ではコピーできません。 *src* と *dst* には文字列としてパス名を与えられます。

**copyfileobj**(*fsrc*, *fdst*, [*length*])

ファイル形式のオブジェクト *fsrc* の内容を *fdst* へコピーします。整数値 *length* はバッファサイズを表します。特に負の *length* はチャンク内のソースデータを繰り返し操作することなくコピーします。つまり標準ではデータは制御不能なメモリ消費を避けるためにチャンク内に読み込まれます。

**copymode**(*src*, *dst*)

*src* から [*dst* へパーミッションをコピーします。ファイル内容や所有者、グループは影響を受けません。 *src* と *dst* には文字列としてパス名を与えられます。

**copystat**(*src*, *dst*)

*src* から *dst* へパーミッション最終アクセス時間、最終更新時間をコピーします。ファイル内容や所有者、グループは影響を受けません。 *src* と *dst* には文字列としてパス名を与えられます。

**copy**(*src*, *dst*)

ファイル *src* をファイルまたはディレクトリ *dst* へコピーします。もし、*dst* がディレクトリであればファイル名は *src* と同じものが指定されたディレクトリ内に作成（または上書き）されます。パーミッションはコピーされます。 *src* と *dst* には文字列としてパス名を与えられます。

**copy2**(*src*, *dst*)

`copy()` と類似していますが、最終アクセス時間や最終更新時間も同様にコピーされます。これは UNIX コマンドの `cp -p` と同様の働きをします。

**copytree**(*src*, *dst*[, *symlinks*])

*src* を起点としてディレクトリツリー全体を再帰的にコピーします。*dst* で指定されたディレクトリは既存のものではなく新規に作成されるものでなくてはなりません。個々のファイルは `copy2()` によってコピーされます。If *symlinks* が真であれば、元のディレクトリ内のシンボリックリンクはコピー先のディレクトリ内へシンボリックリンクとしてコピーされます。偽が与えられたり省略された場合は元のディレクトリ内のリンクの対象となっているファイルがコピー先のディレクトリ内へコピーされます。エラーが発生したときはエラー理由のリストを持った例外を引き起こします。

この関数のソースコードは道具としてよりも使用例として捉えられるべきでしょう。

2.3 で変更された仕様: コピー中にエラーが発生した場合、メッセージを出力するのではなく例外を引き起こすように変更。

**rmtree**(*path*[, *ignore\_errors*[, *onerror*]])

ディレクトリツリー全体を削除します。もし *ignore\_errors* が真であれば削除に失敗したことによるエラーは無視され、偽が与えられたり省略された場合はこれらのエラーは *onerror* で与えられたハンドラを呼び出して処理され、これが省略された場合は例外を引き起こします。

*onerror* が与えられた場合、それは 3 つのパラメータ *function*, *path* および *excinfo* を受け入れて呼び出し可能のものでなくてはなりません。最初のパラメータ *function* は例外を引き起こす関数で `os.remove()` や `os.rmdir()` 等が用いられるでしょう。二番目のパラメータは *path* は *function* へ渡らせるパス名です。三番目のパラメータ *excinfo* は `sys.exc_info()` で返されるような例外情報になるでしょう。*onerror* が引き起こす例外はキャッチできません。

**move**(*src*, *dst*)

再帰的にファイルやディレクトリを別の場所へ移動します。

もし移動先が現在のファイルシステム上であれば単に名前を変更します。そうでない場合はコピーを行い、その後コピー元は削除されます。

2.3 で追加された仕様です。

#### **exception Error**

この例外は複数ファイルの操作を行っているときに生じる例外をまとめたものです。`copytree` に対しては例外の引数は 3 つのタプル (*srcname*, *dstname*, *exception*) からなるリストです。

2.3 で追加された仕様です。

### 6.25.1 使用例

以下は前述の `copytree()` 関数のドキュメント文字列を省略した実装例です。本モジュールで提供される他の関数の使い方を示しています。

```
def copytree(src, dst, symlinks=0):
    names = os.listdir(src)
    os.mkdir(dst)
    for name in names:
        srcname = os.path.join(src, name)
        dstname = os.path.join(dst, name)
        try:
            if symlinks and os.path.islink(srcname):
                linkto = os.readlink(srcname)
                os.symlink(linkto, dstname)
            elif os.path.isdir(srcname):
                copytree(srcname, dstname, symlinks)
            else:
                copy2(srcname, dstname)
        except (IOError, os.error), why:
            print "Can't copy %s to %s: %s" % (srcname, dstname, str(why))
```

## 6.26 locale — 国際化サービス

locale モジュールは POSIX ロケールデータベースおよびロケール関連機能へのアクセスを提供します。POSIX ロケール機構を使うことで、プログラマはソフトウェアが実行される各国における詳細を知らなくても、アプリケーション上で特定の地域文化に関係する部分を扱うことができます。

locale モジュールは、`_locale` を被うように実装されており、ANSI C ロケール実装を使っている `_locale` が利用可能なら、こちらを先に使うようになっています。

locale モジュールでは以下の例外と関数を定義しています:

### exception Error

`setlocale()` が失敗したときに送出される例外です。

**setlocale**(*category*[, *locale*])

*locale* を指定する場合、文字列、(*language code*, *encoding*)、からなるタプル、または `None` をとることができます。*locale* がタプルの場合、ロケール別名解決エンジンによって文字列に変換されます。*locale* が与えられていて、かつ `None` でない場合、`setlocale()` は *category* の設定を変更します。変更することのできるカテゴリは以下に列記されており、値はロケール設定の名前です。空の文字列を指定すると、ユーザの環境における標準設定になります。ロケールの変更に失敗した場合、`Error` が送出されます。成功した場合、新たなロケール設定が返されます。

*locale* が省略されたり `None` の場合、*category* の現在の設定が返されます。

`setlocale()` はほとんどのシステムでスレッド安全ではありません。アプリケーションを書くとき、大抵は以下のコード

```
import locale
locale.setlocale(locale.LC_ALL, '')
```

から書き始めます。これは全てのカテゴリをユーザの環境における標準設定 (大抵は環境変数 `LANG` で指定されています) に設定します。その後複数スレッドを使ってロケールを変更したりしない限り、問題は起こらないはずです。

2.0 で変更された仕様: 引数 *locale* の値としてタプルをサポートしました。

**localeconv**()

地域的な慣行のデータベースを辞書として返します。辞書は以下の文字列をキーとして持っています:

キー名	カテゴリ	意味
LC_NUMERIC	'decimal_point'	小数点を表す文字です。
	'grouping'	'thousands_sep' が来るかもしれない場所を相対的に表した数からなる配列です。配列が CHAR_MAX で終端されている場合、それ以上の桁では桁数字のグループ化を行いません。配列が 0 で終端されている場合、最後に指定したグループが反復的に使われます。
	'thousands_sep'	桁グループ間を区切るために使われる文字です。
LC_MONETARY	'int_curr_symbol'	国際通貨を表現する記号です。
	'currency_symbol'	地域的な通貨を表現する記号です。
	'mon_decimal_point'	金額表示の際に使われる小数点です。
	'mon_thousands_sep'	金額表示の際に桁区切り記号です。
	'mon_grouping'	'grouping' と同じで、金額表示の際に使われます。
	'positive_sign'	正の値の金額表示に使われる記号です。
	'negative_sign'	負の値の金額表示に使われる記号です。
	'frac_digits'	金額を地域的な方法で表現する際の小数点以下の桁数です。
	'int_frac_digits'	金額を国際的な方法で表現する際の小数点以下の桁数です。

'p\_sign\_posn' および 'n\_sign\_posn' の取り得る値は以下の通りです。

値	説明
0	通貨記号および値は丸括弧で囲われます。
1	符号は値と通貨記号より前に来ます。
2	符号は値と通貨記号の後に続きます。
3	符号は値の直前に来ます。
4	符号は値の直後に来ます。
LC_MAX	このロケールでは特に指定しません。

**nl\_langinfo(option)**

ロケール特有の情報を文字列として返します。この関数は全てのシステムで利用可能なわけではなく、指定できる *option* もプラットフォーム間で大きく異なります。引数として使えるのは、locale モジュールで利用可能なシンボル定数を表す数字です。

**getdefaultlocale([envvars])**

標準のロケール設定を取得しようと試み、結果をタプル (*language code*, *encoding*) の形式で返します。POSIX によると、setlocale(LC\_ALL, "") を呼ばなかったプログラムは、移植可能な 'C' ロケール設定を使います。setlocale(LC\_ALL, "") を呼ぶことで、LANG 変数で定義された標準のロケール設定を使うようになります。Python では現在のロケール設定に干渉したくないので、上で述べたような方法でその挙動をエミュレーションしています。

他のプラットフォームとの互換性を維持するために、環境変数 LANG だけでなく、引数 *envvars* で指定された環境変数のリストも調べられます。*envvars* は標準では GNU gettext で使われているサーチパスになります; パスには必ず変数名 'LANG' が含まれているからです。GNU gettext サーチパスは 'LANGUAGE'、'LC\_ALL'、'LC\_CTYPE'、および 'LANG' が列挙した順番に含まれています。

'C' の場合を除き、言語コードは RFC 1766 に対応します。*language code* および *encoding* が決定できなかった場合、None になるかもしれません。



2.0 で追加された仕様です。

**getlocale**(*[category]*)

与えられたロケールカテゴリに対する現在の設定を、*language code*、*encoding* を含む配列で返します。*category* として LC\_ALL 以外の LC\_\* の値の一つを指定できます。標準の設定は LC\_CTYPE です。

'C' の場合を除き、言語コードは RFC 1766 に対応します。*language code* および *encoding* が決定できなかった場合、None になるかもしれません。

2.0 で追加された仕様です。

**getpreferredencoding**(*[do\_setlocale]*)

テキストデータをエンコードする方法を、ユーザの設定に基づいて返します。ユーザの設定は異なるシステム間では異なった方法で表現され、システムによってはプログラミング的に得ることができないこともあるので、この関数が返すのはただの推測です。

システムによっては、ユーザの設定を取得するために *setlocale* を呼び出す必要があるため、この関数はスレッド安全ではありません。*setlocale* を呼び出す必要がない、または呼び出したくない場合、*do\_setlocale* を False に設定する必要があります。2.3 で追加された仕様です。

**normalize**(*localename*)

与えたロケール名を規格化したロケールコードを返します。返されるロケールコードは *setlocale*() で使うために書式化されています。規格化が失敗した場合、もとの名前がそのまま返されます。

与えたエンコードがシステムにとって未知の場合、標準の設定では、この関数は *setlocale*() と同様に、エンコーディングをロケールコードにおける標準のエンコーディングに設定します。2.0 で追加された仕様です。

**resetlocale**(*[category]*)

*category* のロケールを標準設定にします。

標準設定は *getdefaultlocale*() を呼ぶことで決定されます。*category* は標準で LC\_ALL になっています。2.0 で追加された仕様です。

**strcoll**(*string1, string2*)

現在の LC\_COLLATE 設定に従って二つの文字列を比較します。他の比較を行う関数と同じように、*string1* が *string2* に対して前に来るか、後に来るか、あるいは二つが等しいかによって、それぞれ負の値、正の値、あるいは 0 を返します。

**strxfrm**(*string*)

文字列を組み込み関数 *cmp*() で使える形式に変換し、かつロケールに則した結果を返します。この関数は同じ文字列が何度も比較される場合、例えば文字列からなる配列を順序付けて並べる際に使うことができます。

**format**(*format, val[, grouping]*)

数値 *val* を現在の LC\_NUMERIC の設定に基づいて書式化します。書式は % 演算子の慣行に従います。浮動小数点数については、必要に応じて浮動小数点が変更されます。*grouping* が真なら、ロケールに配慮した桁数の区切りが行われます。

**str**(*float*)

浮動小数点数を *str(float)* と同じように書式化しますが、ロケールに配慮した小数点が使われます。

**atof**(*string*)

文字列を LC\_NUMERIC で設定された慣行に従って浮動小数点に変換します。

**atoi**(*string*)

文字列を LC\_NUMERIC で設定された慣行に従って整数に変換します。

**LC\_CTYPE**

文字タイプ関連の関数のためのロケールカテゴリです。このカテゴリの設定に従って、モジュール

string における関数の振る舞いが変わります。

#### LC\_COLLATE

文字列を並べ替えるためのロケールカテゴリです。locale モジュールの関数 `strcoll()` および `strxfrm()` が影響を受けます。

#### LC\_TIME

時刻を書式化するためのロケールカテゴリです。`time.strftime()` はこのカテゴリに設定されている慣行に従います。

#### LC\_MONETARY

金額に関係する値を書式化するためのロケールカテゴリです。設定可能なオプションは関数 `localeconv()` で得ることができます。

#### LC\_MESSAGES

メッセージ表示のためのロケールカテゴリです。現在 Python はアプリケーション毎にロケールに対応したメッセージを出力する機能はサポートしていません。`os.strerror()` が返すような、オペレーティングシステムによって表示されるメッセージはこのカテゴリによって影響を受けます。

#### LC\_NUMERIC

数字を書式化するためのロケールカテゴリです。関数 `format()`、`atoi()`、`atof()` および locale モジュールの `str()` が影響を受けます。他の数値書式化操作は影響を受けません。

#### LC\_ALL

全てのロケール設定を総合したものです。ロケールを変更する際にこのフラグが使われた場合、そのロケールにおける全てのカテゴリを設定しようと試みます。一つでも失敗したカテゴリがあった場合、全てのカテゴリにおいて設定変更を行いません。このフラグを使ってロケールを取得した場合、全てのカテゴリにおける設定を示す文字列が返されます。この文字列は、後に設定を元に戻すために使うことができます。

#### CHAR\_MAX

`localeconv()` の返す特別な値のためのシンボル定数です。

関数 `nl_langinfo` は以下のキーのうち一つを受理します。ほとんどの記述は GNU C ライブラリ中の対応する説明から引用されています。

#### CODESET

選択されたロケールで用いられている文字エンコーディングの名前を文字列で返します。

#### D\_T\_FMT

時刻および日付をロケール特有の方法で表現するために、`strftime(3)` の書式化文字列として用いることのできる文字列を返します。

#### D\_FMT

日付をロケール特有の方法で表現するために、`strftime(3)` の書式化文字列として用いることのできる文字列を返します。

#### T\_FMT

時刻をロケール特有の方法で表現するために、`strftime(3)` の書式化文字列として用いることのできる文字列を返します。

#### T\_FMT\_AMPM

時刻を午前 / 午後の書式で表現するために、`strftime(3)` の書式化文字列として用いることのできる文字列を返します。返される値は

#### DAY\_1 ... DAY\_7

1 週間中の `n` 番目の曜日名を返します。警告: ロケール US における、`DAY_1` を日曜日とする慣行に従っています。国際的な (ISO 8601) 月曜日を週の初めとする慣行ではありません。

**ABDAY\_1 ... ABDAY\_7**

1 週間中の n 番目の曜日名を略式表記で返します。

**MON\_1 ... MON\_12**

n 番目の月の名前を返します。

**ABMON\_1 ... ABMON\_12**

n 番目の月の名前を略式表記で返します。

**RADIXCHAR**

基数点 (小数点ドット、あるいは小数点コンマ、等) を返します。

**THOUSEP**

1000 単位桁区切り (3 桁ごとのグループ化) の区切り文字を返します。

**YESEXPR**

肯定 / 否定で答える質問に対する肯定回答を正規表現関数で認識するために利用できる正規表現を返します。警告: 表現は C ライブラリの `regex()` 関数に合ったものでなければならず、これは `re` で使われている構文とは異なるかもしれません。

**NOEXPR**

肯定 / 否定で答える質問に対する否定回答を正規表現関数で認識するために利用できる正規表現を返します。

**CRNCYSTR**

通貨シンボルを返します。シンボルを値の前に表示させる場合には "-"、値の後ろに表示させる場合には "+"、シンボルを基数点と置き換える場合には "." を前につけます。

**ERA**

現在のロケールで使われている年代を表現する値を返します。

ほとんどのロケールではこの値を定義していません。この値を設定しているロケールの例は日本です。日本では、日付の伝統的な表示法に、時の天皇に対応する元号名を含めます。

通常この値を直接指定する必要はありません。E を書式化文字列に指定することで、関数 `strftime` がこの情報を使うようになります。返される文字列の様式は決められていないので、異なるシステム間で様式に関する同じ知識が使えると期待してはいけません。

**ERA\_YEAR**

返される値はロケールでの現年代の年値です。

**ERA\_D\_T\_FMT**

返される値は `strftime` で日付および時間をロケール固有の年代に基づいた方法で表現するための書式化文字列として使うことができます。

**ERA\_D\_FMT**

返される値は `strftime` で日付をロケール固有の年代に基づいた方法で表現するための書式化文字列として使うことができます。

**ALT\_DIGITS**

返される値は 0 から 99 までの 100 個の値の表現です。

例:

```
>>> import locale
>>> loc = locale.setlocale(locale.LC_ALL) # get current locale
>>> locale.setlocale(locale.LC_ALL, 'de_DE') # use German locale; name might vary with platform
>>> locale.strcoll('f\xe4n', 'foo') # compare a string containing an umlaut
>>> locale.setlocale(locale.LC_ALL, '') # use user's preferred locale
>>> locale.setlocale(locale.LC_ALL, 'C') # use default (C) locale
>>> locale.setlocale(locale.LC_ALL, loc) # restore saved locale
```

### 6.26.1 ロケールの背景、詳細、ヒント、助言および補足説明

C 標準では、ロケールはプログラム全体にわたる特性であり、その変更は高価な処理であるとしています。加えて、頻繁にロケールを変更するようなひどい実装はコアダンプを引き起こすこともあります。このことがロケールを正しく利用する上で苦痛となっています。

そもそも、プログラムが起動した際、ロケールはユーザの希望するロケールにかかわらず 'C' です。プログラムは `setlocale(LC_ALL, "")` を呼び出して、明示的にユーザの希望するロケール設定を行わなければなりません。

`setlocale()` をライブラリルーチン内で呼ぶことは、それがプログラム全体に及ぼす副作用の面から、一般的によくはない考えです。ロケールを保存したり復帰したりするのもよくありません：高価な処理であり、ロケールの設定が復帰する以前に起動してしまった他のスレッドに悪影響を及ぼすからです。

もし、汎用を目的としたモジュールを作っていて、ロケールによって影響をうけるような操作（例えば `string.lower()` や `time.strftime()` の書式の一部）のロケール独立のバージョンが必要ということになれば、標準ライブラリルーチンを使わずに何とかしなければなりません。よりましな方法は、ロケール設定が正しく利用できているか確かめることです。最後の手段は、あなたのモジュールが 'C' ロケール以外の設定には互換性がないとドキュメントに書くことです。

`string` モジュールの大小文字の変換を行う関数はロケール設定によって影響を受けます。`setlocale()` 関数を呼んで `LC_CTYPE` 設定を変更した場合、変数 `string.lowercase`、`string.uppercase` および `string.letters` は計算しなおされます。例えば `from string import letters` のように、`'from ... import ...'` を使ってこれらの変数を使っている場合には、それ以降の `setlocale()` の影響を受けないので注意してください。

ロケールに従って数値操作を行うための唯一の方法はこのモジュールで特別に定義されている関数：`atof()`、`atoi()`、`format()`、`str()` を使うことです。

### 6.26.2 Python 拡張の作者と、Python を埋め込むようなプログラムに関して

拡張モジュールは、現在のロケールを調べる以外は、決して `setlocale()` を呼び出してはなりません。しかし、返される値もロケールの復帰のために使えるだけなので、さほど便利とはいえません（例外はおそらくロケールが 'C' かどうか調べることでしょう）。

Python があるアプリケーション内に埋め込まれており、アプリケーションが Python を初期化する前にロケールを設定した場合は一般的に問題はありません。このとき Python は設定されているロケールを使います。ただし、`LC_NUMERIC` ロケールは常に 'C' に設定されます。

`locale` モジュールの関数 `setlocale()` は、`LC_NUMERIC` ロケール設定が操作できるような印象を Python プログラマに与えますが、C のレベルではこれは当てはまりません。C コードでは、常に `LC_NUMERIC` ロケール設定は 'C' になります。これは、小数点文字がピリオド以外の別の文字に変更された場合にうまく動作しなくなるものがあまりにも多い（例えば Python パーザはうまく動作しません）からです。補足説明：Python のグローバルインタプリタをロックしないまま動作するスレッド間では、数値ロケール設定が一致しなくなることがあるかもしれません。このため、`LC_NUMERIC` ロケールの設定を実装するための可

搬性のある唯一の方法は、数値ロケールをユーザの希望するロケールに設定して、直接関係のある値を展開し、最後に ‘C’ 数値ロケールを復帰することです。

ロケールを変更するために Python コードで `locale` モジュールを使った場合、Python を埋め込んでいるアプリケーションにも影響を及ぼします。Python を埋め込んでいるアプリケーションに影響が及ぶことを望まない場合、‘`config.c`’ ファイル内の組み込みモジュールのテーブルから `_locale` 拡張モジュール(ここで全てを行っています)を削除し、共有ライブラリから `_locale` モジュールにアクセスできないようにしてください。

### 6.26.3 メッセージカタログへのアクセス

C ライブラリの `gettext` インタフェースが提供されているシステムでは、`locale` モジュールでそのインタフェースを公開しています。このインタフェースは関数 `gettext()`、`dgettext()`、`dcgettext()`、`textdomain()`、および `bindtextdomain()` からなります。これらは `gettext` モジュールの同名の関数に似ていますが、メッセージカタログとして C ライブラリのバイナリフォーマットを使い、メッセージカタログを探すために C ライブラリのサーチアルゴリズムを使います。

Python アプリケーションでは、通常これらの関数を呼び出す必要はないはずで、代わりに `gettext` を呼ぶべきです。例外として知られているのは、内部で `gettext()` または `cdgettext()` を呼び出すような C ライブラリにリンクするアプリケーションです。こうしたアプリケーションでは、ライブラリが正しいメッセージカタログを探せるようにテキストドメイン名を指定する必要があります。

## 6.27 gettext — 多言語対応に関する国際化サービス

`gettext` モジュールは、Python によるモジュールやアプリケーションの国際化 (I18N, I-nternationalizatio-N) および地域化 (L10N, L-ocalizatio-N) サービスを提供します。このモジュールは GNU `gettext` メッセージカタログへの API と、より高レベルで Python ファイルに適しているクラスに基づいた API の両方をサポートしています。以下で述べるインタフェースを使うことで、モジュールやアプリケーションのメッセージをある自然言語で記述しておき、翻訳されたメッセージのカタログを与えて他の異なる自然言語の環境下で動作させることができます。

ここでは Python のモジュールやアプリケーションを地域化するためのいくつかのヒントも提供しています。

### 6.27.1 GNU gettext API

`gettext` モジュールでは、以下の GNU `gettext` API に非常に良く似た API を提供しています。この API を使う場合、メッセージ翻訳の影響はアプリケーション全体に及ぼすことになります。アプリケーションが単一の言語しか扱わず、各言語に依存する部分をユーザのロケール情報によって選ぶのなら、ほとんどの場合この方法でやりたいことを実現できます。Python モジュールを地域化していたり、アプリケーションの実行中に言語を切り替えたい場合、おそらくクラスに基づいた API を使いたくなるでしょう。

`bindtextdomain(domain[, localedir])`

`domain` をロケール辞書 `localedir` に結び付け (bind) ます。具体的には、`gettext` は与えられたドメインに対するバイナリ形式の ‘.mo’ ファイルを、(UNIX では) ‘`localedir/language/LC_MESSAGES/domain.mo`’ から探します。ここで `languages` はそれぞれ環境変数 `LANGUAGE`、`LC_ALL`、`LC_MESSAGES`、および `LANG` の中から検索されます。

`localedir` が省略されるか `None` の場合、現在 `domain` に結び付けられている内容が返されます。<sup>2</sup>

---

<sup>2</sup> 標準でロケールが収められているディレクトリはシステム依存です; 例えば、RedHat Linux では ‘`/usr/share/locale`’ ですが、



`textdomain([domain])`

現在のグローバルドメインを調べたり変更したりします。`domain` が `None` の場合、現在のグローバルドメインが返されます。それ以外の場合にはグローバルドメインは `domain` に設定され、設定されたグローバルドメインを返します。

`gettext(message)`

現在のグローバルドメイン、言語、およびロケール辞書に基づいて、`message` の特定地域向けの翻訳を返します。通常、ローカルな名前空間ではこの関数に `_` という別名をつけます (下の例を参照してください)。

`dgettext(domain, message)`

`gettext()` と同様ですが、指定された `domain` からメッセージを探します。

`ngettext(singular, plural, n)`

`gettext()` と同様ですが、複数形の場合を考慮しています。翻訳文字列が見つかった場合、`n` の様式を適用し、その結果得られたメッセージを返します (言語によっては二つ以上の複数形があります)。翻訳文字列が見つからなかった場合、`n` が 1 なら `singular` を返します; そうでない場合 `plural` を返します。

複数形の様式はカタログのヘッダから取り出されます。様式は C または Python の式で、自由な変数 `n` を持ちます; 式の評価値はカタログ中の複数形のインデクスとなります。`.po` ファイルで用いられる詳細な文法と、様々な言語における様式については、GNU `gettext` ドキュメントを参照してください。

2.3 で追加された仕様です。

`dngettext(domain, singular, plural, n)`

`ngettext()` と同様ですが、指定された `domain` からメッセージを探します。

2.3 で追加された仕様です。

GNU `gettext` では `dcgettext()` も定義していますが、このメソッドはあまり有用ではないと思われるので、現在のところ実装されていません。

以下にこの API の典型的な使用法を示します:

```
import gettext
gettext.bindtextdomain('myapplication', '/path/to/my/language/directory')
gettext.textdomain('myapplication')
_ = gettext.gettext
# ...
print _('This is a translatable string.')
```

## 6.27.2 クラスに基づいた API

クラス形式の `gettext` モジュールの API は GNU `gettext` API よりも高い柔軟性と利便性を持っています。Python のアプリケーションやモジュールを地域化するにはこちらを使う方を勧めます。`gettext` では、GNU `.mo` 形式のファイルを解釈し、標準の 8 ビット文字列または Unicode 文字列形式でメッセージを返す“翻訳”クラスを定義しています。翻訳オブジェクトのインスタンスも組み込み名前空間に関数 `_()` として組み入れる (`install`) ことができます。

`find(domain[, localedir[, languages[, all]]])`

この関数は標準的な `.mo` ファイル検索アルゴリズムを実装しています。`textdomain()` と同じく、

---

Solaris では `'/usr/lib/locale'` です。`gettext` モジュールはこうしたシステム依存の標準設定をサポートしません; その代わりに `'sys.prefix/share/locale'` を標準の設定とします。この理由から、常にアプリケーションの開始時に絶対パスで明示的に指定して `bindtextdomain()` を呼び出すのが最良のやり方ということになります。



*domain* を引数にとります。オプションの *localedir* は `bindtextdomain()` と同じです。またオプションの *languages* は文字列を列挙したリストで、各文字列は言語コードを表します。

*localedir* が与えられていない場合、標準のシステムロケールディレクトリが使われます。<sup>3</sup>

*languages* が与えられなかった場合、以下の環境変数: `LANGUAGE`、`LC_ALL`、`LC_MESSAGES`、および `LANG` が検索されます。空でない値を返した最初の候補が *languages* 変数として使われます。この環境変数は言語名をコロンで分かち書きしたリストを含んでいなければなりません。`find()` はこの文字列をコロンで分割し、言語コードの候補リストを生成します。

`find()` は次に言語コードを展開および正規化し、リストの各要素について、以下のパス構成:

`'localedir/language/LC_MESSAGES/domain.mo'`

からなる実在するファイルの探索を反復的に行います。`find()` は上記のような実在するファイルで最初に見つかったものを返します。該当するファイルが見つからなかった場合、`None` が返されます。*all* が与えられていれば、全ファイル名のリストが言語リストまたは環境変数で指定されている順番に並べられたものを返します。

`translation(domain[, localedir[, languages[, class_[fallback]]]])`

`Translations` インスタンスを *domain*、*localedir*、および *languages* に基づいて生成して返します。*domain*、*localedir*、および *languages* はまず関連付けられている `'mo'` ファイルパスのリストを取得するために `find()` に渡されます。同じ `'mo'` ファイル名を持つインスタンスはキャッシュされます。実際にインスタンス化されるクラスは *class\_* が与えられていればそのクラスが、そうでない時には `GNUTranslations` です。クラスのコンストラクタは単一の引数としてファイルオブジェクトを取らなくてはなりません。

複数のファイルが発見された場合、後で見つかったファイルは前に見つかったファイルの代替で見なされ、後で見つかった方が利用されます。代替の設定を可能にするには、`copy.copy` を使ってキャッシュから翻訳オブジェクトを複製します; こうすることで、実際のインスタンスデータはキャッシュのものと共有されます。

`'mo'` ファイルが見つからなかった場合、*fallback* が偽 (標準の設定です) ならこの関数は `IOError` を送出し、*fallback* が真なら `NullTranslations` インスタンスが返されます。

`install(domain[, localedir[, unicode]])`

関数 `translation()` に渡した *domain* および *localedir* に基づいて、Python の組み込み名前空間に関数 `_` を組み入れます。*unicode* フラグは `translation()` の返した翻訳オブジェクトの `install` メソッドに渡されます。

以下に示すように、通常はアプリケーション中の文字列を関数 `_()` の呼び出しで包み込んで翻訳対象候補であることを示します:

```
print _('This string will be translated.')
```

利便性を高めるためには、`_()` 関数を Python の組み込み名前空間に組み入れる必要があります。こうすることで、アプリケーション内の全てのモジュールからアクセスできるようになります。

## NullTranslations クラス

翻訳クラスは、元のソースファイル中のメッセージ文字列から翻訳されたメッセージ文字列への変換を実際に実装しているクラスです。全ての翻訳クラスが基底クラスとして用いるクラスが `NullTranslations` です; このクラスでは独自の特殊な翻訳クラスを実装するために使うことができる基本的なインタフェースを以下に `NullTranslations` のメソッドを示します:

<sup>3</sup> 上の `bindtextdomain()` に関する脚注を参照してください。

`__init__([fp])`

オプションのファイルオブジェクト *fp* を取ります。この引数は基底クラスでは無視されます。このメソッドは“保護された (protected)” インスタンス変数 `_info` および `_charset` を初期化します。これらの変数の値は導出クラスで設定することができます。同様に `_fallback` も初期化しますが、この値は `add_fallback` で設定されます。その後、*fp* が `None` でない場合 `self._parse(fp)` を呼び出します。

`_parse(fp)`

基底クラスでは何もしない (no-op) になっています。このメソッドの役割はファイルオブジェクト *fp* を引数に取り、ファイルからデータを読み出し、メッセージカタログを初期化することです。サポートされていないメッセージカタログ形式を使っている場合、その形式を解釈するためにはこのメソッドを上書きしなくてはなりません。

`add_fallback(fallback)`

*fallback* を現在の翻訳オブジェクトの代替オブジェクトとして追加します。翻訳オブジェクトが与えられたメッセージに対して翻訳メッセージを提供できない場合、この代替オブジェクトに問い合わせることになります。

`gettext(message)`

代替オブジェクトが設定されている場合、`gettext` を代替オブジェクトに転送します。そうでない場合、翻訳されたメッセージを返します。導出クラスで上書きするメソッドです。

`ugettext(message)`

代替オブジェクトが設定されている場合、`gettext` を代替オブジェクトに転送します。そうでない場合、翻訳されたメッセージを Unicode 文字列で返します。導出クラスで上書きするメソッドです。

`ngettext(singular, plural, n)`

代替オブジェクトが設定されている場合、`ngettext` を代替オブジェクトに転送します。そうでない場合、翻訳されたメッセージを返します。導出クラスで上書きするメソッドです。2.3 で追加された仕様です。

`ungettext(singular, plural, n)`

代替オブジェクトが設定されている場合、`ungettext` を代替オブジェクトに転送します。そうでない場合、翻訳されたメッセージを Unicode 文字列で返します。導出クラスで上書きするメソッドです。2.3 で追加された仕様です。

`info()`

“保護されている” `_info` 変数を返します。

`charset()`

“保護されている” `_charset` 変数を返します。

`install([unicode])`

*unicode* フラグが偽の場合、このメソッドは `self.gettext()` を組み込み名前空間に組み入れ、`'_'` と結び付けます。*unicode* が真の場合、`self.gettext()` の代わりに `self.ugettext()` を結び付けます。標準では *unicode* は偽です。

この方法はアプリケーションで `_` 関数を利用できるようにするための最も便利な方法ですが、唯一の手段でもあるので注意してください。この関数はアプリケーション全体、とりわけ組み込み名前空間に影響するので、地域化されたモジュールで `_` を組み入れることができないのです。その代わりに、以下のコード:

```
import gettext
t = gettext.translation('mymodule', ...)
_ = t.gettext
```

を使って `_` を使えるようにしなければなりません。

この操作は `_` をモジュール内だけのグローバル名前空間に組み入れるので、モジュール内の `_` の呼び出しだけに影響します。

## GNUTranslations クラス

`gettext` モジュールでは `NullTranslations` から導出されたもう一つのクラス: `GNUTranslations` を提供しています。このクラスはビッグエンディアン、およびリトルエンディアン両方のバイナリ形式の GNU `gettext` ‘.mo’ ファイルを読み出せるように `_parse()` を上書きしています。また、このクラスはメッセージ id とメッセージ文字列の両方を Unicode に型強制します。

このクラスではまた、翻訳カタログ以外に、オプションのメタデータを読み込んで解釈します。GNU `gettext` では、空の文字列に対する変換先としてメタデータを取り込むことが慣習になっています。このメタデータは RFC 822 形式の `key: value` のペアになっており、`Project-Id-Version` キーを含んでいなければなりません。キー `Content-Type` があった場合、`charset` の特性値 (property) は“保護された” `_charset` インスタンス変数を初期化するために用いられます。値がない場合には、デフォルトとして `None` が使われます。エンコードに用いられる文字セットが指定されている場合、カタログから読み出された全てのメッセージ id とメッセージ文字列は、指定されたエンコードを用いて Unicode に変換されます。`ugettext()` は常に Unicode を返し、`gettext()` はエンコードされた 8 ビット文字列を返します。どちらのメソッドにおける引数 id の場合も、Unicode 文字列か US-ASCII 文字のみを含む 8 ビット文字列だけが受理可能です。国際化された Python プログラムでは、メソッドの Unicode 版 (すなわち `ugettext()` や `ungettext()`) の利用が推奨されています。

`key/value` ペアの集合全体は辞書型データ中に配置され、“保護された” `_info` インスタンス変数に設定されます。

‘.mo’ ファイルのマジックナンバーが不正な場合、あるいはその他の問題がファイルの読み出し中に発生した場合、`GNUTranslations` クラスのインスタンス化で `IOError` が送出されることがあります。

以下のメソッドは基底クラスの実装からオーバーライドされています:

**`gettext (message)`**

カタログから `message` id を検索して、対応するメッセージ文字列を、カタログの文字セットが既知のエンコードの場合、エンコードされた 8 ビット文字列として返します。`message` id に対するエントリがカタログに存在せず、フォールバックが設定されている場合、フォールバック検索はオブジェクトの `gettext()` メソッドに転送されます。そうでない場合、`message` id 自体が返されます。

**`ugettext (message)`**

カタログから `message` id を検索して、対応するメッセージ文字列を、Unicode でエンコードして返します。`message` id に対するエントリがカタログに存在せず、フォールバックが設定されている場合、フォールバック検索はオブジェクトの `ugettext()` メソッドに転送されます。そうでない場合、`message` id 自体が返されます。

**`ngettext (singular, plural, n)`**

メッセージ id に対する複数形を検索します。カタログに対する検索では `singular` がメッセージ id として用いられ、`n` にはどの複数形を用いるかを指定します。返されるメッセージ文字列は 8 ビットの文字列で、カタログの文字セットが既知の場合にはその文字列セットでエンコードされています。

メッセージ id がカタログ中に見つからず、フォールバックオブジェクトが指定されている場合、メッセージ検索要求はフォールバックオブジェクトの `ngettext()` メソッドに転送されます。そうでない場合、`n` が 1 ならば `singular` が返され、それ以外に対しては `plural` が返されます。

2.3 で追加された仕様です。

**`ungettext (singular, plural, n)`**

メッセージ id に対する複数形を検索します。カタログに対する検索では *singular* がメッセージ id として用いられ、*n* にはどの複数形を用いるかを指定します。返されるメッセージ文字列は Unicode 文字列です。

メッセージ id がカタログ中に見つからず、フォールバックオブジェクトが指定されている場合、メッセージ検索要求はフォールバックオブジェクトの `ungettext()` メソッドに転送されます。そうでない場合、*n* が 1 ならば *singular* が返され、それ以外に対しては *plural* が返されます。

以下に例を示します。:

```
n = len(os.listdir('.'))
cat = GNUTranslations(somefile)
message = cat.ungettext(
    'There is %(num)d file in this directory',
    'There are %(num)d files in this directory',
    n) % {'n': n}
```

2.3 で追加された仕様です。

### Solaris メッセージカタログ機構のサポート

Solaris オペレーティングシステムでは、独自の '.mo' バイナリファイル形式を定義していますが、この形式に関するドキュメントが手に入らないため、現時点ではサポートされていません。

### Catalog コンストラクタ

GNOME では、James Henstridge によるあるバージョンの `gettext` モジュールを使っていますが、このバージョンは少し異なった API を持っています。ドキュメントに書かれている利用法は:

```
import gettext
cat = gettext.Catalog(domain, locale_dir)
_ = cat.gettext
print _('hello world')
```

となっています。過去のモジュールとの互換性のために、`Catalog()` は前述の `translation()` 関数の別名になっています。

このモジュールと Henstridge のバージョンとの間には一つ相違点があります: 彼のカタログオブジェクトはマップ型の API を介したアクセスがサポートされていましたが、この API は使われていないらしく、現在はサポートされていません。

## 6.27.3 プログラムやモジュールを国際化する

国際化 (I18N, I-nternationalizatio-N) とは、プログラムを複数の言語に対応させる操作を指します。地域化 (L10N, L-ocalizatio-N) とは、すでに国際化されているプログラムを特定地域の言語や文化的な事情に対応させることを指します。Python プログラムに多言語メッセージ機能を追加するには、以下の手順を踏む必要があります:

1. プログラムやモジュールで翻訳対象とする文字列に特殊なマークをつけて準備します
2. マークづけをしたファイルに一連のツールを走らせ、生のメッセージカタログを生成します
3. 特定の言語へのメッセージカタログの翻訳を作成します

#### 4. メッセージ文字列を適切に変換するために gettext モジュールを使います

ソースコードを I18N 化する準備として、ファイル内の全ての文字列を探す必要があります。翻訳を行う必要のある文字列はどれも `_('...')` — すなわち関数 `_()` の呼び出しで包むことでマーク付けしなくてはなりません。例えば以下のようにします:

```
filename = 'mylog.txt'
message = _('writing a log message')
fp = open(filename, 'w')
fp.write(message)
fp.close()
```

この例では、文字列 `'writing a log message'` が翻訳対象候補としてマーク付けされており、文字列 `'mylog.txt'` および `'w'` はされていません。

Python の配布物には、ソースコードに準備作業を行った後でメッセージカタログの生成を助ける 2 つのツールが付属します。これらはバイナリ配布の場合には付属していたりしなかったりしますが、ソースコード配布には入っており、`'Tools/i18n'` ディレクトリにあります。

**pygettext** プログラム<sup>4</sup>は全ての Python ソースコードを走査し、予め翻訳対象としてマークした文字列を探し出します。このツールは GNU **gettext** プログラムと同様ですが、Python ソースコードの機微について熟知している反面、C 言語や C++ 言語のソースコードについては全く知りません。(C 言語による拡張モジュールのように) C 言語のコードも翻訳対象にしたいのでない限り、GNU **gettext** は必要ありません。

**pygettext** は、テキスト形式 Uniforum スタイルによる人間が判読可能なメッセージカタログ `'pot'` ファイル群を生成します。このファイル群はソースコード中でマークされた全ての文字列と、それに対応する翻訳文字列のためのプレースホルダを含むファイルで構成されています。**pygettext** はコマンドライン形式のスクリプトで、**xgettext** と同様のコマンドラインインタフェースをサポートします; 使用法についての詳細を見るには:

```
pygettext.py --help
```

を起動してください。

これら `'pot'` ファイルのコピーは次に、サポート対象の各自然言語について、言語ごとのバージョンを作成する個々の人間の翻訳者に頒布されます。翻訳者たちはプレースホルダ部分を埋めて言語ごとのバージョンをつくり、`'po'` ファイルとして返します。( `'Tools/i18n'` ディレクトリ内の) **msgfmt.py**<sup>5</sup> プログラムを使い、翻訳者から返された `'po'` ファイルから機械可読な `'mo'` バイナリカタログファイルを生成します。`'mo'` ファイルは、**gettext** モジュールが実行時に実際の翻訳処理を行うために使われます。

**gettext** モジュールをソースコード中でどのように使うかはアプリケーション全体を国際化するのか、それとも単一のモジュールを国際化するのかによります。

#### モジュールを地域化する

モジュールを地域化する場合、グローバルな変更、例えば組み込み名前空間への変更を行わないように注意しなければなりません。GNU **gettext** API ではなく、クラスベースの API を使うべきです。

仮に対象のモジュール名を `"spam"` とし、モジュールの各言語における翻訳が収められた `'mo'` ファイルが `'/usr/share/locale'` に GNU **gettext** 形式で置かれているとします。この場合、モジュールの最初で以下の

<sup>4</sup> 同様の作業を行う **xpot** と呼ばれるプログラムを François Pinard が書いています。このプログラムは彼の **po-utils** パッケージの一部で、<http://www.iro.umontreal.ca/contrib/po-utils/HTML/> で入手できます。

<sup>5</sup> **msgfmt.py** は GNU **msgfmt** とバイナリ互換ですが、より単純で、Python だけを使った実装がされています。このプログラムと **pygettext.py** があれば、通常 Python プログラムを国際化するために GNU **gettext** パッケージをインストールする必要はありません。



ようにします:

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.gettext
```

翻訳オブジェクトが '.po' ファイル中の Unicode 文字列を返すようになっているのなら、上の代わりに以下のようにします:

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.ugettext
```

### アプリケーションを地域化する

アプリケーションを地域化するのなら、関数 `_()` をグローバルな組み込み名前空間に組み入れなければならず、これは通常アプリケーションの主ドライバ (main driver) ファイルで行います。この操作によって、アプリケーション独自のファイルは明示的に各ファイルで `_()` の組み入れを行わなくても単に `_('...')` を使うだけで済むようになります。

単純な場合では、単に以下の短いコードをアプリケーションの主ドライバファイルに追加するだけです:

```
import gettext
gettext.install('myapplication')
```

ロケールディレクトリや *unicode* フラグを設定する必要がある場合、それらの値を `install()` 関数に渡すことができます:

```
import gettext
gettext.install('myapplication', '/usr/share/locale', unicode=1)
```

### 動作中 (on the fly) に言語を切り替える

多くの言語を同時にサポートする必要がある場合、複数の翻訳インスタンスを生成して、例えば以下のコード:

```
import gettext

lang1 = gettext.translation(languages=['en'])
lang2 = gettext.translation(languages=['fr'])
lang3 = gettext.translation(languages=['de'])

# start by using language 1
lang1.install()

# ... time goes by, user selects language 2
lang2.install()

# ... more time goes by, user selects language 3
lang3.install()
```



のように、インスタンスを明示的に切り替えてもかまいません。

### 翻訳処理の遅延解決

コードを書く上では、ほとんどの状況で文字列はコードされた場所で翻訳されます。しかし場合によっては、翻訳対象として文字列をマークはするが、その後実際に翻訳が行われるように遅延させる必要が生じます。古典的な例は以下のようなコードです：

```
animals = ['mollusk',
           'albatross',
           'rat',
           'penguin',
           'python',
           ]
# ...
for a in animals:
    print a
```

ここで、リスト `animals` 内の文字列は翻訳対象としてマークはしたいが、文字列が出力されるまで実際に翻訳を行うのは避けたいとします。

こうした状況进行处理する一つの方法を以下に示します：

```
def _(message): return message

animals = [_( 'mollusk' ),
           _( 'albatross' ),
           _( 'rat' ),
           _( 'penguin' ),
           _( 'python' ),
           ]

del _

# ...
for a in animals:
    print _(a)
```

ダミーの `_()` 定義が単に文字列をそのまま返すようになっているので、上のコードはうまく動作します。かつ、このダミーの定義は、組み込み名前空間に置かれた `_()` の定義で (`del` 命令を実行するまで) 一時的に上書きすることができます。もしそれまでに `_` をローカルな名前空間に持っていたら注意してください。

二つ目の例における `_()` の使い方では、“a” は文字列リテラルではないので、`pygettext` プログラムが翻訳可能な対象として識別しません。

もう一つの処理法は、以下の例のようなやり方です：

```
def N_(message): return message

animals = [N_('mollusk'),
           N_('albatross'),
           N_('rat'),
           N_('penguin'),
           N_('python'),
           ]

# ...
for a in animals:
    print _(a)
```

この例の場合では、翻訳可能な文字列を関数 `N_()` でマーク付けしており<sup>6</sup>、`_( )` の定義とは全く衝突しません。しかしメッセージ展開プログラムには翻訳対象の文字列が `N_()` でマークされていることを教える必要が出てくるでしょう。`pygettext` および `xpot` は両方とも、コマンドライン上のスイッチでこの機能をサポートしています。

#### 6.27.4 謝辞

以下の人々が、このモジュールのコード、フィードバック、設計に関する助言、過去の実装、そして有益な経験談による貢献をしてくれました:

- Peter Funk
- James Henstridge
- Juan David Ibáñez Palomar
- Marc-André Lemburg
- Martin von Löwis
- François Pinard
- Barry Warsaw

## 6.28 logging — Python 用ロギング機能

2.3 で追加された仕様です。このモジュールでは、アプリケーションのための柔軟なエラーログ記録 (logging) システムを実装するための関数やクラスを定義しています。

ログ記録は `Logger` クラスのインスタンス (以降 ロガー :logger) におけるメソッドを呼び出すことで行われます。各インスタンスは名前をもち、ドット (ピリオド) を区切り文字として表記することで、概念的には名前空間中の階層構造に配置されることになります。例えば、"scan" と名づけられたロガーは "scan.text"、"scan.html"、および "scan.pdf" ロガーの親ロガーとなります。ロガー名には何をつけてもよく、ログに記録されるメッセージの生成元となるアプリケーション中の特定の領域を示すことになります。

ログ記録されたメッセージにはまた、重要度レベル (level of importance) が関連付けられています。デフォルトのレベルとして提供されているものは `DEBUG`、`INFO`、`WARNING`、`ERROR` および `CRITICAL` です。簡便性のために、`Logger` の適切なメソッド群を呼ぶことで、ログに記録されたメッセージの重要性を指定

---

<sup>6</sup> この `N_()` をどうするかは全くの自由です; `MarkThisStringForTranslation()` などとしてもかまいません。

することができます。それらのメソッドとは、デフォルトのレベルを反映する形で、`debug()`、`info()`、`warning()`、`error()` および `critical()` となっています。これらのレベルを指定するにあたって制限はありません: `Logger` のより汎用的なメソッドで、明示的なレベル指定のための引数を持つ `log()` を使って自分自身でレベルを定義したり使用したりできます。

レベルもロガーに関連付けることができ、デベロッパが設定することも、保存されたログ記録設定を読み込む際に設定することもできます。ロガーに対してログ記録メソッドが呼び出されると、ロガーは自らのレベルとメソッド呼び出しに関連付けられたレベルを比較します。ロガーのレベルがメソッド呼び出しのレベルよりも高い場合、実際のログメッセージは生成されません。これはログ出力の冗長性を制御するための基本的なメカニズムです。

ログ記録されるメッセージは `LogRecord` クラスのインスタンスとしてコード化されます。ロガーがあるイベントを実際にログ出力すると決定した場合、ログメッセージから `LogRecord` インスタンスが生成されます。

ログ記録されるメッセージは、ハンドラ (*handlers*) を通して、処理機構 (dispatch mechanism) にかかけられます。ハンドラは `Handler` クラスのサブクラスのインスタンスで、ログ記録された (`LogRecord` 形式の) メッセージが、そのメッセージの伝達対象となる相手 (エンドユーザ、サポートデスクのスタッフ、システム管理者、開発者) に行き着くようにする役割を持ちます。ハンドラには特定の行き先に方向付けられた `LogRecord` インスタンスが渡されます。各ロガーはゼロ個、単一またはそれ以上のハンドラを (`Logger` の `addHandler` メソッド) で関連付けることができます。ロガーに直接関連付けられたハンドラに加えて、ロガーの上位にあるロガー全てに関連付けられたハンドラがメッセージを処理する際に呼び出されます。

ロガーと同様に、ハンドラは関連付けられたレベルを持つことができます。ハンドラのレベルはロガーのレベルと同じ方法で、フィルタとして働きます。ハンドラがあるイベントを実際に処理すると決定した場合、`emit()` メソッドが使われ、メッセージを発送先に送信します。ほとんどのユーザ定義の `Handler` のサブクラスで、この `emit()` をオーバーライドする必要があるでしょう。

基底クラスとなる `Handler` クラスに加えて、多くの有用なサブクラスが提供されています:

1. `StreamHandler` のインスタンスはストリーム (ファイル様オブジェクト) にエラーメッセージを送信します。
2. `FileHandler` のインスタンスはディスク上のファイルにエラーメッセージを送信します。files.
3. `RotatingFileHandler` のインスタンスは最大ログファイルのサイズ指定とログファイルの交替機能をサポートしながら、ディスク上のファイルにエラーメッセージを送信します。
4. `SocketHandler` のインスタンスは TCP/IP ソケットにエラーメッセージを送信します。TCP/IP sockets.
5. `DatagramHandler` のインスタンスは UDP ソケットにエラーメッセージを送信します。
6. `SMTPHandler` のインスタンスは指定された電子メールアドレスにエラーメッセージを送信します。
7. `SysLogHandler` のインスタンスは遠隔を含むマシン上の syslog デーモンにエラーメッセージを送信します。
8. `NTEventLogHandler` のインスタンスは Windows NT/2000/XP イベントログにエラーメッセージを送信します。
9. `MemoryHandler` のインスタンスはメモリ上のバッファにエラーメッセージを送信し、指定された条件でフラッシュされるようにします。
10. `HTTPHandler` のインスタンスは 'GET' か 'POST' セマンティクスを使って HTTP サーバにエラーメッセージを送信します。

StreamHandler および FileHandler クラスは、中核となるログ化機構パッケージ内で定義されています。他のハンドラはサブモジュール、`logging.handlers` で定義されています。(サブモジュールにはもうひとつ `logging.config` があり、これは環境設定機能のためのものです。)

ログ記録されたメッセージは `Formatter` クラスのインスタンスを介し、表示用に書式化されます。これらのインスタンスは `%` 演算子と辞書を使うのに適した書式化文字列で初期化されます。

複数のメッセージの初期化をバッチ処理するために、`BufferingFormatter` のインスタンスを使うことができます。書式化文字列(バッチ処理で各メッセージに適用されます)に加えて、ヘッダ (header) およびトレイラ (trailer) 書式化文字列が用意されています。

ロガーレベル、ハンドラレベルの両方または片方に基づいたフィルタリングが十分でない場合、`Logger` および `Handler` インスタンスに `Filter` のインスタンスを (`addFilter()` メソッドを介して) 追加することができます。メッセージの処理を進める前に、ロガーとハンドラはともに、全てのフィルタでメッセージの処理が許可されているか調べます。いずれかのフィルタが偽となる値を返した場合、メッセージの処理は行われません。

基本的な `Filter` 機能では、指定されたロガー名でフィルタを行えるようになっています。この機能が利用された場合、名前付けされたロガーとその下位にあるロガーに送られたメッセージがフィルタを通過できるようになり、その他のメッセージは捨てられます。

上で述べたクラスに加えて、いくつかのモジュールレベルの関数が存在します。

`getLogger([name])`

指定された名前のロガーを返します。名前が指定されていない場合、ロガー階層のルート (root) にあるロガーを返します。

与えられた名前に対して、この関数はどの呼び出しでも同じロガーインスタンスを返します。従って、ロガーインスタンスをアプリケーションの各部でやりとりする必要はなくなります。

`debug(msg[, *args[, **kwargs]])`

L レベル `DEBUG` のメッセージをルートロガーで記録します。`msg` はメッセージの書式化文字列で、`args` は `msg` に取り込むための引数です。キーワード文字列 `kwargs` からは `exc_info` のみが調べられ、この値の評価値が偽でない場合、(`sys.exc_info` から得た) 例外情報がログメッセージに追加されます。

`info(msg[, *args[, **kwargs]])`

レベル `INFO` のメッセージをルートロガーで記録します。引数は `debug()` と同じように解釈されます。

`warning(msg[, *args[, **kwargs]])`

レベル `WARNING` のメッセージをルートロガーで記録します。引数は `debug()` と同じように解釈されます。

`error(msg[, *args[, **kwargs]])`

レベル `ERROR` のメッセージをルートロガーで記録します。引数は `debug()` と同じように解釈されます。

`critical(msg[, *args[, **kwargs]])`

レベル `CRITICAL` のメッセージをルートロガーで記録します。引数は `debug()` と同じように解釈されます。

`exception(msg[, *args])`

レベル `ERROR` のメッセージをルートロガーで記録します。引数は `debug()` と同じように解釈されます。例外情報はログメッセージに追加されます。このメソッドは例外ハンドラからのみ呼び出されます。

`disable(lvl)`

全てのロガーに対して、ロガー自体のレベルに優先するような上書きレベル *lvl* を与えます。アプリケーション全体にわたって一時的にログ出力の頻度を押し下げる必要が生じた場合にはこの関数が有効です。

**addLevelName**(*lvl*, *levelName*)

内部辞書内でレベル *lvl* をテキスト *levelName* に関連付けます。これは例えば `Formatter` でメッセージを書式化する際のように、数字のレベルをテキスト表現に対応付ける際に用いられます。この関数は自作のレベルを定義するために使うこともできます。使われるレベルに対する唯一の制限は、レベルは正の整数でなくてはならず、メッセージの深刻さが上がるに従ってレベルの数も上がらなくてはならないということです。

**getLevelName**(*lvl*)

ログ記録レベル *lvl* のテキスト表現を返します。レベルが定義済みのレベル `CRITICAL`、`ERROR`、`WARNING`、`INFO`、あるいは `DEBUG` のいずれかである場合、対応する文字列が返されます。`addLevelName()` を使ってレベルに名前に関連づけていた場合、*lvl* に関連付けられていた名前が返されます。そうでない場合、文字列 `"Level %s" % lvl` が返されます。

**makeLogRecord**(*attrdict*)

属性が *attrdict* で定義された、新たな `LogRecord` インスタンスを生成して返します。この関数は pickle 化された `LogRecord` 属性の辞書を作成し、ソケットを介して送信し、受信端で `LogRecord` インスタンスとして再構成する際に便利です。

**makeLogRecord**(*attrdict*)

*attrdict* で属性を定義した、新しい `LogRecord` インスタンスを返します。この関数は、逆 pickle 化された `LogRecord` 属性辞書を socket 越しに受け取り、受信端で `LogRecord` インスタンスに再構築する場合に有用です。

**basicConfig**()

デフォルトのフォーマッタ (`formatter`) を持つ `StreamHandler` を生成してルートロガーに追加し、ログ記録システムの基本的な環境設定を行います。関数 `debug()`、`info()`、`warning()`、`error()`、および `critical()` は、ルートロガーにハンドラが定義されていない場合に自動的に `basicConfig()` を呼び出します。

**shutdown**()

ログ記録システムに対して、バッファのフラッシュを行い、全てのハンドラを閉じることで順次シャットダウンを行うように告知します。

**setLoggerClass**(*klass*)

ログ記録システムに対して、ロガーをインスタンス化する際にクラス *klass* を使うように指示します。指定するクラスは引数として名前だけをとるようなメソッド `__init__()` を定義していなければならず、`__init__()` では `Logger.__init__()` を呼び出さなければなりません。典型的な利用法として、この関数は自作のロガーを必要とするようなアプリケーションにおいて、他のロガーがインスタンス化される前にインスタンス化されます。

参考資料:

PEP 282, “A Logging System”

本機能を Python 標準ライブラリに含めるよう記述している提案書。

## 6.28.1 Logger オブジェクト

ロガーは以下の属性とメソッドを持ちます。ロガーを直接インスタンス化することはできず、常にモジュール関数 `logging.getLogger(name)` を介してインスタンス化するので注意してください。

**propagate**

この値の評価結果が偽になる場合、ログ記録しようとするメッセージはこのロガーに渡されず、また子ロガーから上位の(親の)ロガーに渡されません。コンストラクタはこの属性を 1 に設定します。

**setLevel(lvl)**

このロガーの閾値を *lvl* に設定します。ログ記録しようとするメッセージで、*lvl* よりも深刻でないものは無視されます。ロガーが生成された際、レベルは NOTSET (全てのメッセージがルートロガーで処理されるか、非ルートロガーの場合には親ロガーに処理を代行させる) に設定されます。

**isEnabledFor(lvl)**

深刻さが *lvl* のメッセージが、このロガーで処理されることになっているかどうかを示します。このメソッドはまず、`logging.disable(lvl)` で設定されるモジュールレベルの深刻さレベルを調べ、次にロガーの実効レベルを `getEffectiveLevel()` で調べます。

**getEffectiveLevel()**

このロガーの実効レベルを示します。NOTSET 以外の値が `setLevel()` で設定されていた場合、その値が返されます。そうでない場合、NOTSET 以外の値が見つかるまでロガーの階層をルートロガーの方向に追跡します。見つかった場合、その値が返されます。

**debug(msg[, \*args[, \*\*kwargs]])**

レベル DEBUG のメッセージをこのロガーで記録します。*msg* はメッセージの書式化文字列で、*args* は *msg* に取り込むための引数です。キーワード文字列 *kwargs* からは *exc\_info* のみが調べられ、この値の評価値が偽でない場合、(`sys.exc_info` から得た) 例外情報がログメッセージに追加されます。

**info(msg[, \*args[, \*\*kwargs]])**

レベル INFO のメッセージをこのロガーで記録します。引数は `debug()` と同じように解釈されます。

**warning(msg[, \*args[, \*\*kwargs]])**

レベル WARNING のメッセージをこのロガーで記録します。引数は `debug()` と同じように解釈されます。

**error(msg[, \*args[, \*\*kwargs]])**

レベル ERROR のメッセージをこのロガーで記録します。引数は `debug()` と同じように解釈されます。

**critical(msg[, \*args[, \*\*kwargs]])**

レベル CRITICAL のメッセージをこのロガーで記録します。引数は `debug()` と同じように解釈されます。

**log(lvl, msg[, \*args[, \*\*kwargs]])**

レベル *lvl* のメッセージをこのロガーで記録します。その他の引数は `debug()` と同じように解釈されます。

**exception(msg[, \*args])**

レベル ERROR のメッセージをこのロガーで記録します。引数は `debug()` と同じように解釈されます。例外情報はログメッセージに追加されます。このメソッドは例外ハンドラからのみ呼び出されます。

**addFilter(filt)**

指定されたフィルタ *filt* をこのロガーに追加します。

**removeFilter(filt)**

指定されたフィルタ *filt* をこのロガーから除去します。

**filter(record)**

このロガーのフィルタをレコード (*record*) に適用し、レコードがフィルタを透過して処理されることになる場合には真を返します。

**addHandler(hdlr)**



指定されたハンドラ *hdlr* をこのロガーに追加します。

**removeHandler**(*hdlr*)

指定されたハンドラ *hdlr* をこのロガーから除去します。

**findCaller**()

呼び出し元のソースファイル名と行番号を調べます。ファイル名と行番号を 2 要素のタプルで返します。

**handle**(*record*)

レコードをこのロガーおよびその上位ロガーに (*propagate* の値が偽になるまで) さかのぼった関連付けられている全てのハンドラに渡して処理します。このメソッドはソケットから受信した逆 pickle 化されたレコードに対してもレコードがローカルで生成された場合と同様に用いられます。*filter*() によって、ロガーレベルでのフィルタが適用されます。

**makeRecord**(*name, lvl, fn, lno, msg, args, exc\_info*)

このメソッドは、特殊な *LogRecord* インスタンスを生成するためにサブクラスでオーバーライドできるファクトリメソッドです。

## 6.28.2 Handler オブジェクト

ハンドラは以下の属性とメソッドを持ちます。Handler は直接インスタンス化されることはありません; このクラスはより便利なサブクラスの基底クラスとして働きます。しかしながら、サブクラスにおける *\_\_init\_\_*() メソッドでは、*Handler.\_\_init\_\_*() を呼び出す必要があります。

*\_\_init\_\_*(*level=NOTSET*)

レベルを設定して、Handler インスタンスを初期化します。空のリストを使ってフィルタを設定し、I/O 機構へのアクセスを直列化するために (*createLock*() を使って) ロックを生成します。

**createLock**()

スレッド安全でない根底の I/O 機能に対するアクセスを直列化するために用いられるスレッドロック (thread lock) を初期化します。

**acquire**()

*createLock*() で生成されたスレッドロックを獲得します。

**release**()

*acquire*() で獲得したスレッドロックを解放します。

**setLevel**(*lvl*)

このハンドラに対する閾値を *lvl* に設定します。ログ記録しようとするメッセージで、*lvl* よりも深刻でないものは無視されます。ハンドラが生成された際、レベルは *NOTSET* (全てのメッセージが処理される) に設定されます。

**setFormatter**(*form*)

このハンドラのフォーマッタを *form* に設定します。

**addFilter**(*filt*)

指定されたフィルタ *filt* をこのハンドラに追加します。

**removeFilter**(*filt*)

指定されたフィルタ *filt* をこのハンドラから除去します。

**filter**(*record*)

このハンドラのフィルタをレコードに適用し、レコードがフィルタを透過して処理されることになる場合には真を返します。

**flush**()

全てのログ出力がフラッシュされるようにします。このクラスのバージョンではなにも行わず、サブクラスで実装するためのものです。

**close()**

ハンドラで使われている全てのリソースを始末します。このクラスのバージョンではなにも行わず、サブクラスで実装するためのものです。

**handle(record)**

ハンドラに追加されたフィルタの条件に応じて、指定されたログレコードを発信します。このメソッドは I/O スレッドロックの獲得/開放を伴う実際のログ発信をラップします。

**handleError()**

このメソッドは emit() の呼び出し中に例外に遭遇した際にハンドラから呼び出されます。デフォルトではこのメソッドは何も行いません。すなわち、例外は暗黙のまま無視されます。ほとんどのログ記録システムでは、これがほぼ望ましい機能です - というのは、ほとんどのユーザはログ記録システム自体のエラーは気にせず、むしろアプリケーションのエラーに興味があるからです。しかしながら、望むならこのメソッドを自作のハンドラと置き換えることはできます。

**format(record)**

レコードに対する書式化を行います - フォーマッタが設定されていれば、それを使います。そうでない場合、モジュールにデフォルト指定されたフォーマッタを使います。

**emit(record)**

指定されたログ記録レコードを実際にログ記録する際の全ての処理を行います。このメソッドのこのクラスのバージョンはサブクラスで実装するためのものなので、`NotImplementedError` を送出します。

## StreamHandler

`StreamHandler` クラスはログ出力を `sys.stdout`、`sys.stderr` あるいは何らかのファイル類似オブジェクト (あるいは、もっと正確に言えば、`write()` および `flush()` メソッドをサポートする何らかのオブジェクト) といったストリームに送信します。

**class StreamHandler([strm])**

`StreamHandler` クラスの新たなインスタンスを返します。`strm` が指定された場合、インスタンスはログ出力先として指定されたストリームを使います; そうでない場合、`sys.stderr` が使われます。

**emit(record)**

フォーマッタが指定されていれば、フォーマッタを使ってレコードを書式化します。次に、レコードがストリームに書き込まれ、末端に改行がつけられます。例外情報が存在する場合、`traceback.print_exception()` を使って書式化され、ストリームの末尾につけられます。

**flush()**

ストリームの `flush()` メソッドを呼び出してバッファをフラッシュします。`close()` メソッドは `Handler` から継承しているため何も行わないので、`flush()` 呼び出しを明示的に行う必要があります。

## FileHandler

`FileHandler` クラスはログ出力をディスク上のファイルに送信します。このクラスは出力機能を `StreamHandler` から継承しています。

**class FileHandler(filename[, mode])**

`FileHandler` クラスの新たなインスタンスを返します。指定されたファイルが開かれ、ログ記録の

ためのストリームとして使われます。*mode* が指定されなかった場合、`'a'` が使われます。デフォルトでは、ファイルは無制限に大きくなりつづけます。

`close()`

ファイルを閉じます。

`emit(record)`

*record* をファイルに出力します。

## RotatingFileHandler

`RotatingFileHandler` クラスはディスク上のログファイルに対するローテーション処理をサポートします。

`class RotatingFileHandler(filename[, mode[, maxBytes[, backupCount]]])`

`RotatingFileHandler` クラスの新たなインスタンスを返します。指定されたファイルが開かれ、ログ記録のためのストリームとして使われます。*mode* が指定されなかった場合、`"a"` が使われます。デフォルトでは、ファイルは無制限に大きくなりつづけます。

あらかじめ決められたサイズでファイルをロールオーバー (*rollover*) させられるように、*maxBytes* および *backupCount* 値を指定することができます。指定サイズを超えそうになると、ファイルは閉じられ、暗黙のうちに新たなファイルが開かれます。ロールオーバーは現在のログファイルの長さが *maxBytes* に近くなると常に起きます。*backupCount* が非ゼロの場合、システムは古いログファイルをファイル名に `".1"`, `".2"` といった拡張子を追加して保存します。例えば、*backupCount* が 5 で、基本のファイル名が `'app.log'` なら、`'app.log'`、`'app.log.1'`、`'app.log.2'`、... と続き、`'app.log.5'` までを得ることになります。ログの書き込み対象になるファイルは常に `'app.log'` です。このファイルが満杯になると、ファイルは閉じられ、`'app.log.1'` に名称変更されます。`'app.log.1'`、`'app.log.2'` などが存在する場合、それらのファイルはそれぞれ `'app.log.2'`、`'app.log.3'` といった具合に名前変更されます。

`doRollover()`

上で記述したようにして、ロールオーバーを行います。

`emit(record)`

`setRollover()` で記述したようなロールオーバーを行いながら、レコードをファイルに出力します。

## SocketHandler

`SocketHandler` クラスはログ出力をネットワークソケットに送信します。基底クラスでは TCP ソケットを用います。

`class SocketHandler(host, port)`

アドレスが *host* および *port* で与えられた遠隔のマシンと通信するようにした `SocketHandler` クラスのインスタンスを生成して返します。

`close()`

ソケットを閉じます。

`handleError()`

`emit()`

レコードの属性辞書を pickle 化し、バイナリ形式でソケットに書き込みます。ソケット操作でエラーが生じた場合、暗黙のうちにパケットは捨てられます。前もって接続が失われていた場合、接続を再度確立します。受信端でレコードを逆 pickle 化して `LogRecord` にするには、`makeLogRecord` 関数を使ってください。

`handleError()`

`emit()` の処理中に発生したエラーを処理します。よくある原因は接続の消失です。次のイベント発生時に再度接続確立を試みることができるようにソケットを閉じます。

**makeSocket()**

サブクラスで必要なソケット形式を詳細に定義できるようにするためのファクトリメソッドです。デフォルトの実装では、TCP ソケット (`socket.SOCK_STREAM`) を生成します。

**makePickle(record)**

レコードの属性辞書を pickle 化して、長さを指定プレフィクス付きのバイナリにし、ソケットを介して送信できるようにして返します。

**send(packet)**

pickle 化された文字列 *packet* をソケットに送信します。この関数はネットワークが処理待ち状態の時に発生しうる部分的送信を行えます。

## DatagramHandler

DatagramHandler クラスは SocketHandler を継承しており、ログ記録メッセージを UDP ソケットを介して送れるようサポートしています。

**class DatagramHandler(host, port)**

アドレスが *host* および *port* で与えられた遠隔のマシンと通信するようにした DatagramHandler クラスのインスタンスを生成して返します。

**emit()**

レコードの属性辞書を pickle 化し、バイナリ形式でソケットに書き込みます。ソケット操作でエラーが生じた場合、暗黙のうちにパケットは捨てられます。前もって接続が失われていた場合、接続を再度確立します。受信端でレコードを逆 pickle 化して LogRecord にするには、makeLogRecord 関数を使ってください。

**makeSocket()**

ここで SocketHandler のファクトリメソッドをオーバーライドして UDP ソケット (`socket.SOCK_DGRAM`) を生成しています。

**send(s)**

pickle 化された文字列をソケットに送信します。

## SysLogHandler

SysLogHandler クラスでは、ログ記録メッセージを遠隔またはローカルの UNIX syslog に送信する機能をサポートしています。

**class SysLogHandler([address[, facility]])**

遠隔の UNIX マシンと通信するための、SysLogHandler クラスの新たなインスタンスを返します。マシンのアドレスは (*host*, *port*) のタプル形式をとる *address* で与えられます。*address* が指定されない場合、('localhost', 514) が使われます。アドレスは UDP ソケットを使って開かれます。*facility* が指定されない場合、LOG\_USER が使われます。

**close()**

遠隔ホストのソケットを閉じます。

**emit(record)**

レコードは書式化された後、syslog サーバに送信されます。例外情報が存在しても、サーバには送信されません。

**encodePriority(facility, priority)**

便宜レベル (facility) および優先度を整数に符号化します。値は文字列でも整数でも渡すことができます。文字列が渡された場合、内部の対応付け辞書が使われ、整数に変換されます。

## NTEventLogHandler

NTEventLogHandler クラスでは、ログ記録メッセージをローカルな Windows NT、Windows 2000、または Windows XP のイベントログ (event log) に送信する機能をサポートします。この機能を使えるようにするには、Mark Hammond による Python 用 Win32 拡張パッケージをインストールする必要があります。

```
class NTEventLogHandler ( appname[, dllname[, logtype ] ] )
```

NTEventLogHandler クラスの新たなインスタンスを返します。appname はイベントログに表示する際のアプリケーション名を定義するために使われます。この名前を使って適切なレジストリエントリが生成されます。dllname はログに保存するメッセージ定義の入った .dll または .exe ファイルへの完全に限定的な (fully qualified) パス名を与えなければなりません (指定されない場合、'win32service.pyd' が使われます - このライブラリは Win32 拡張とともにインストールされ、いくつかのプレースホルダとなるメッセージ定義を含んでいます)。これらのプレースホルダを利用すると、メッセージの発信源全体がログに記録されるため、イベントログは巨大になるので注意してください。logtype は 'Application'、'System' または 'Security' のいずれかであるか、デフォルトの 'Application' でなければなりません。

```
close()
```

現時点では、イベントログエントリの発信源としてのアプリケーション名をレジストリから除去することができます。しかしこれを行うと、イベントログビューアで意図したログをみることができなくなるでしょう - これはイベントログが .dll 名を取得するためにレジストリにアクセスできなければならないからです。現在のバージョンではこの操作を行いません (実際、このメソッドは何も行いません)。

```
emit ( record )
```

メッセージ ID、イベントカテゴリおよびイベント型を決定し、メッセージを NT イベントログに記録します。

```
getEventCategory ( record )
```

レコードに対するイベントカテゴリを返します。自作のカテゴリを指定したい場合、このメソッドをオーバーライドしてください。このクラスのバージョンのメソッドは 0 を返します。

```
getEventType ( record )
```

レコードのイベント型を返します。自作の型を指定したい場合、このメソッドをオーバーライドしてください。このクラスのバージョンのメソッドは、ハンドラの *typemap* 属性を使って対応付けを行います。この属性は `__init__()` で初期化され、DEBUG、INFO、WARNING、ERROR、および CRITICAL が入っています。自作のレベルを使っているのなら、このメソッドをオーバーライドするか、ハンドラの *typemap* 属性に適切な辞書を配置する必要があるでしょう。

```
getMessageID ( record )
```

レコードのメッセージ ID を返します。自作のメッセージを使っているのなら、ロガーに渡される *msg* を書式化文字列ではなく ID にします。その上で、辞書参照を行ってメッセージ ID を得ます。このクラスのバージョンでは 1 を返します。この値は 'win32service.pyd' における基本となるメッセージ ID です。

## SMTPHandler

SMTPHandler クラスでは、SMTP を介したログ記録メッセージの送信機能をサポートします。

```
class SMTPHandler ( mailhost, fromaddr, toaddrs, subject )
```

新たな SMTPHandler クラスのインスタンスを返します。インスタンスは email の from および to アドレス行、および subject 行とともに初期化されます。toaddrs はドメイン名 (mailost) を含まない文字列からなるリストでなければなりません。非標準の SMTP ポートを指定するには、mailhost 引数に (host, port) のタプル形式を指定します。文字列を使った場合、標準の SMTP ポートが使われます。

**emit(record)**

レコードを書式化し、指定されたアドレスに送信します。

**getSubject(record)**

レコードに応じたサブジェクト行を指定したいなら、このメソッドをオーバーライドしてください。

## MemoryHandler

MemoryHandler では、ログ記録するレコードをメモリ上にバッファし、定期的にその内容をターゲット (target) となるハンドラにフラッシュする機能をサポートしています。フラッシュ処理はバッファが一杯になるか、ある深刻さかそれ以上のレベルをもったイベントが観測された際に行われます。

MemoryHandler はより一般的な抽象クラス、BufferingHandler のサブクラスです。この抽象クラスでは、ログ記録するレコードをメモリ上にバッファします。各レコードがバッファに追加される毎に、shouldFlush() を呼び出してバッファをフラッシュすべきかどうか調べます。フラッシュする必要がある場合、flush() が必要にして十分な処理を行うものと想定しています。

**class BufferingHandler(capacity)**

指定し許容量のバッファでハンドラを初期化します。

**emit(record)**

レコードをバッファに追加します。shouldFlush() が真を返す場合、バッファを処理するために flush() を呼び出します。

**flush()**

このメソッドをオーバーライドして、自作のフラッシュ動作を実装することができます。このクラスのバージョンのメソッドでは、単にバッファの内容を削除して空にします。

**shouldFlush(record)**

バッファが許容量に達している場合に真を返します。このメソッドは自作のフラッシュ処理方針を実装するためにオーバーライドすることができます。

**class MemoryHandler(capacity[, flushLevel[, target]])**

MemoryHandler クラスの新たなインスタンスを返します。インスタンスはサイズ capacity のバッファとともに初期化されます。flushLevel が指定されていない場合、ERROR が使われます。target が指定されていない場合、ハンドラが何らかの有意義な処理を行う前に setTarget() でターゲットを指定する必要があります。

**close()**

flush() を呼び出し、ターゲットを None に設定してバッファを消去します。

**flush()**

MemoryHandler の場合、フラッシュ処理は単に、バッファされたレコードをターゲットがあれば送信することを意味します。違った動作を行いたい場合、オーバーライドしてください。

**setTarget(target)**

ターゲットハンドラをこのハンドラに設定します。

**shouldFlush(record)**

バッファが満杯になっているか、flushLevel またはそれ以上のレコードでないかを調べます。



## HTTPHandler

HTTPHandler クラスでは、ログ記録メッセージを 'GET' または 'POST' セマンティクスを使って Web サーバに送信する機能をサポートしています。

```
class HTTPHandler (host, url[, method])
```

HTTPHandler クラスの新たなインスタンスを返します。インスタンスはホストアドレス、URL および HTTP メソッドとともに初期化されます。*method* が指定されなかった場合 'GET' が使われます。

```
emit(record)
```

レコードを URL エンコードされた辞書形式で Web サーバに送信します。

### 6.28.3 Formatter オブジェクト

Formatter は以下の属性とメソッドを持っています。Formatter は LogRecord を (通常は) 人間か外部のシステムで解釈できる文字列に変換する役割を担っています。基底クラスの Formatter では書式化文字列を指定することができます。何も指定されなかった場合、'*%s(message)n*' の値が使われます。

Formatter は書式化文字列とともに初期化され、LogRecord 属性に入っている知識を利用できるようにします - 上で触れたデフォルトの値では、ユーザによるメッセージと引数はあらかじめ書式化されて、LogRecord の *message* 属性に入っていることを利用しているようにです。この書式化文字列は、Python 標準の % を使った変換文字列で構成されます。文字列整形に関する詳細は [8 “String Formatting Operations”](#) の章を参照してください。

現状で、LogRecord の有用な属性は以下に述べるようになっています:

Format	Description
<i>%(name)s</i>	ロガー (ログ記録チャンネル) の名前
<i>%(levelno)s</i>	メッセージのログ記録レベルを表す数字 (DEBUG, INFO, WARNING, ERROR, CRITICAL)
<i>%(levelname)s</i>	メッセージのログ記録レベルを表す文字列 ("DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL")
<i>%(pathname)s</i>	ログ記録の呼び出しが行われたソースファイルの全パス名 (取得できる場合)
<i>%(filename)s</i>	パス名中のファイル名部分
<i>%(module)s</i>	モジュール名 (ファイル名の名前部分)
<i>%(lineno)d</i>	ログ記録の呼び出しが行われたソース行番号 (取得できる場合)
<i>%(created)f</i>	LogRecord が生成された時刻 (time.time() の返した値)
<i>%(asctime)s</i>	LogRecord が生成された時刻を人間が読める書式で表したもの。デフォルトでは “2003-07-08 16:49:”
<i>%(msecs)d</i>	LogRecord が生成された時刻の、ミリ秒部分
<i>%(thread)d</i>	スレッド ID (取得できる場合)
<i>%(process)d</i>	プロセス ID (取得できる場合)
<i>%(message)s</i>	レコードが発信された際に処理された msg %args の結果

```
class Formatter ([fmt[, datefmt]])
```

Formatter クラスの新たなインスタンスを返します。インスタンスは全体としてのメッセージに対する書式化文字列と、メッセージの日付/時刻部分のための書式化文字列を伴って初期化されます。*fmt* が指定されない場合、'*%(message)s*' が使われます。*datefmt* が指定されない場合、ISO8601 日付書式が使われます。

```
format(record)
```

レコードの属性辞書が、文字列を書式化する演算で被演算子として使われます。書式化された結果の文字列を返します。辞書を書式化する前に、二つの準備段階を経ます。レコードの *message* 属性が *msg %*

*args* を使って処理されます。書式化された文字列が '*(asctime)*' を含むなら、*formatTime()* が呼び出され、イベントの発生時刻を書式化します。例外情報が存在する場合、*formatException()* を使って書式化され、メッセージに追加されます。

***formatTime(record[, datefmt])***

このメソッドは、フォーマッタが書式化された時間を利用したい際に、*format()* から呼び出されます。このメソッドは特定の要求を提供するためにフォーマッタで上書きすることができますが、基本的な振る舞いは以下ようになります: *datefmt* (文字列) が指定された場合、レコードが生成された時刻を書式化するために *time.strftime()* で使われます。そうでない場合、ISO8601 書式が使われます。結果の文字列が返されます。

***formatException(exc\_info)***

指定された例外情報 (*sys.exc\_info()* が返すような標準例外のタプル) を文字列として書式化します。デフォルトの実装は単に *traceback.print\_exception()* を使います。結果の文字列が返されます。

## 6.28.4 Filter オブジェクト

*Filter* は *Handler* と *Logger* によって利用され、レベルによる制御よりも洗練されたフィルタ処理を提供します。基底のフィルタクラスでは、ロガーの階層構造のある点よりも下層にあるイベントだけを通過させます。例えば、"A.B" で初期化されたフィルタはロガー "A.B"、"A.B.C"、"A.B.C.D"、"A.B.D" などでログ記録されたイベントを通過させます。しかし、"A.BB"、"B.A.B" などは通過させません。空の文字列で初期化された場合、全てのイベントを通過させます。

***class Filter([name])***

*Filter* クラスのインスタンスを返します。 *name* が指定されていれば、*name* はロガーの名前を表します。指定されたロガーとその子ロガーのイベントがフィルタを通過できるようになります。 *name* が指定されなければ、全てのイベントを通過させます。

***filter(record)***

指定されたレコードがログされているか？ されていなければゼロを、されていればゼロでない値を返します。適切と判断されれば、このメソッドによってレコードは *in place* で修正されることがあります。

## 6.28.5 LogRecord オブジェクト

何かをログ記録する際は常に *LogRecord* インスタンスが生成されます。インスタンスにはログ記録されることになっているイベントに関する全ての情報が入っています。インスタンスに渡される主要な情報は *msg* および *args* で、これらは *msg % args* を使って組み合わせられ、レコードのメッセージフィールドを生成します。レコードはまた、レコードがいつ生成されたか、ログ記録がソースコード行のどこで呼び出されたか、あるいはログ記録すべき何らかの例外情報といった情報も含んでいます。

*LogRecord* にはメソッドがありません; ログ記録イベントの情報を収めたただの容器 (repository) です。このオブジェクトが辞書でなくクラスになっている唯一の理由は、拡張を容易にするためです。

***class LogRecord(name, lvl, pathname, lineno, msg, args, exc\_info)***

関係のある情報とともに初期化された *LogRecord* のインスタンスを返します。 *name* はロガーの名前です; *lvl* は数字で表されたレベルです; *pathname* はログ記録呼び出しが見つかったソースファイルの絶対パス名です。 *msg* はユーザ定義のメッセージ (書式化文字列) です; *args* はタプルで、*msg* と合わせて、ユーザメッセージを生成します; *exc\_info* は例外情報のタプルで、*sys.exc\_info()* を呼び出して得られたもの (または、例外情報が取得できない場合には *None*) です。

## 6.28.6 スレッド安全性

`logging` モジュールは、クライアントで特殊な作業を必要としないかぎりスレッド安全 (thread-safe) になっています。このスレッド安全性はスレッドロックによって達成されています; モジュールの共有データへのアクセスを直列化するためのロックが一つ存在し、各ハンドラでも根底にある I/O へのアクセスを直列化するためにロックを生成します。

## 6.28.7 環境設定

### 環境設定のための関数

以下の関数では、`logging` モジュールを環境設定できるようにします。これらの関数を使えるようにするには、`logging.config` を import しなければなりません。これらの関数の使用はオプションです - `logging` モジュールは全て、(`logging` 自体で定義されている) 主要な API を呼び出し、`logging` か `logging.handlers` で宣言されているハンドラを定義することで設定することができます。

**fileConfig(*fname*[, *defaults*])**

ログ記録の環境設定をファイル名 *fname* の ConfigParser 形式ファイルから読み出します。この関数はアプリケーションから何度も呼び出すことができ、これによって、(設定の選択と、選択された設定を読み出す機構をデベロッパが提供していれば) 複数のお仕着せの設定からエンドユーザが選択するようになります。ConfigParser に渡すためのデフォルト値は *defaults* 引数で指定できます。

**listen([*port*])**

指定されたポートでソケットサーバを開始し、新たな設定を待ち受け (listen) ます。ポートが指定されなければ、モジュールのデフォルト設定である `DEFAULT_LOGGING_CONFIG_PORT` が使われます。ログ記録の環境設定は `fileConfig()` で処理できるようなファイルとして送信されます。Thread インスタンスを返し、サーバを開始するために `start()` を呼び、適切な状態で `join()` を呼び出すことができます。サーバを停止するには `stopListening()` を呼んでください。

**stopListening()**

`listen()` を呼び出して作成された、待ち受け中のサーバを停止します。通常 `listen()` の戻り値に対して `join()` が呼ばれる前に呼び出します。

### 環境設定ファイルの書式

`fileConfig` が解釈できる環境設定ファイルの形式は、ConfigParser の機能に基づいています。ファイルには、`[loggers]`、`[handlers]`、および `[formatters]` といったセクションが入っていなければならず、各セクションではファイル中で定義されている各タイプのエンティティを名前指定しています。こうしたエンティティの各々について、そのエンティティをどう設定するかを示した個別のセクションがあります。すなわち、`log01` という名前の `[loggers]` セクションにあるロガーに対しては、対応する詳細設定がセクション `[logger_log01]` に収められています。同様に、`hand01` という名前の `[handlers]` セクションにあるハンドラは `[handler_hand01]` と呼ばれるセクションに設定をもつことになり、`[formatters]` セクションにある `form01` は `[formatter_form01]` というセクションで設定が指定されています。ルートロガーの設定は `[logger_root]` と呼ばれるセクションで指定されていなければなりません。

ファイルにおけるこれらのセクションの例を以下に示します。

```
[loggers]
keys=root,log02,log03,log04,log05,log06,log07

[handlers]
keys=hand01,hand02,hand03,hand04,hand05,hand06,hand07,hand08,hand09

[formatters]
keys=form01,form02,form03,form04,form05,form06,form07,form08,form09
```

ルートロガーでは、レベルとハンドラのリストを指定しなければなりません。ルートロガーのセクションの例を以下に示します。

```
[logger_root]
level=NOTSET
handlers=hand01
```

level エントリは DEBUG, INFO, WARNING, ERROR, CRITICAL のうちのの一つか、NOTSET になります。ルートロガーの場合にのみ、NOTSET は全てのメッセージがログ記録されることを意味します。レベル値は logging パッケージの名前空間のコンテキストにおいて eval() されます。

handlers エントリはコンマで区切られたハンドラ名からなるリストで、[handlers] セクションになくてもなりません。また、これらの各ハンドラの名前に対応するセクションが設定ファイルに存在しなければなりません。

ルートロガー以外のロガーでは、いくつか追加の情報が必要になります。これは以下の例のように表されます。

```
[logger_parser]
level=DEBUG
handlers=hand01
propagate=1
qualname=compiler.parser
```

level および handlers エントリはルートロガーのエントリと同様に解釈されますが、非ルートロガーのレベルが NOTSET に指定された場合、ログ記録システムはロガー階層のより上位のロガーにロガーの実効レベルを問い合わせるところが違います。propagate エントリは、メッセージをロガー階層におけるこのロガーの上位のハンドラに伝播させることを示す 1 に設定されるか、メッセージを階層の上位に伝播しないことを示す 0 に設定されます。qualname エントリはロガーのチャンネル名を階層的に表したもので、例えばアプリケーションがこのロガーを取得する際に使う名前になります。

ハンドラの環境設定を指定しているセクションは以下の例のようになります。

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)
```

class エントリはハンドラのクラス(logging パッケージの名前空間において eval() で決定されます)を示します。level はロガーの場合と同じように解釈され、NOTSET は "全てを記録する(log everything)" と解釈されます。

formatter エントリはこのハンドラのフォーマッタに対するキー名を表します。空文字列の場合、デ

フォルトのフォーマッタ (logging.\_defaultFormatter) が使われます。名前が指定されている場合、その名前は [formatters] セクションになくてもならず、対応するセクションが設定ファイル中になければなりません。

args エントリは、logging パッケージの名前空間のコンテキストで eval() される際、ハンドラクラスのコンストラクタに対する引数からなるリストになります。典型的なエントリがどうやって作成されるかについては、対応するハンドラのコンストラクタが、以下の例を参照してください。

```
[handler_hand02]
class=FileHandler
level=DEBUG
formatter=form02
args=('python.log', 'w')

[handler_hand03]
class=handlers.SocketHandler
level=INFO
formatter=form03
args=('localhost', handlers.DEFAULT_TCP_LOGGING_PORT)

[handler_hand04]
class=handlers.DatagramHandler
level=WARN
formatter=form04
args=('localhost', handlers.DEFAULT_UDP_LOGGING_PORT)

[handler_hand05]
class=handlers.SysLogHandler
level=ERROR
formatter=form05
args=((('localhost', handlers.SYSLOG_UDP_PORT), handlers.SysLogHandler.LOG_USER))

[handler_hand06]
class=NTEventLogHandler
level=CRITICAL
formatter=form06
args=('Python Application', '', 'Application')

[handler_hand07]
class=SMTPHandler
level=WARN
formatter=form07
args=('localhost', 'from@abc', ['user1@abc', 'user2@xyz'], 'Logger Subject')

[handler_hand08]
class=MemoryHandler
level=NOTSET
formatter=form08
target=
args=(10, ERROR)

[handler_hand09]
class=HTTPHandler
level=NOTSET
formatter=form09
args=('localhost:9022', '/log', 'GET')
```

フォーマッタの環境設定を指定しているセクションは以下のような形式です。

```
[formatter_form01]
format=F1 %(asctime)s %(levelname)s %(message)s
datefmt=
```

format エントリは全体を書式化する文字列で、datefmt エントリは `strftime()` 互換の日付/時刻書式化文字列です。空文字列の場合、パッケージによって ISO8601 形式の日付/時刻に置き換えられ、日付書式化文字列 "ISO8601 形式ではミリ秒も指定しており、上の書式化文字列の結果にカンマで区切って追加されます。ISO8601 形式の時刻の例は 2003-01-23 00:29:50,411 です。

## 6.28.8 Using the logging package

### Basic example - log to a file

ログをファイルに記録する例を、以下に示します。この例では `Logger` インスタンスを作り、その後 `FileHandler` と `Formatter` を作ります。続いて `FileHandler` に `Formatter` を付加し、`Logger` に `FileHandler` を付加します。最後に、ロガーのデバッグ・レベルを設定します。

```
import logging
logger = logging.getLogger('myapp')
hdlr = logging.FileHandler('/var/tmp/myapp.log')
formatter = logging.Formatter('%(asctime)s %(levelname)s %(message)s')
hdlr.setFormatter(formatter)
logger.addHandler(hdlr)
logger.setLevel(logging.WARNING)
```

このロガーオブジェクトを使うと、エントリをログファイルに書き出せます。

```
logger.error('We have a problem')
logger.info('While this is just chatty')
```

書き出されたファイルの中には、次のように書かれているでしょう。

```
2003-07-08 16:49:45,896 ERROR We have a problem
```

info メッセージは、ファイルに書き出されません。setLevel メソッドで WARNING 以上のログだけを要求したので、info メッセージは捨てられたのです。

タイムスタンプは “year-month-day hour:minutes:seconds,milliseconds” 形式です。ミリ秒のフィールドは 3 桁ありますが、全てのシステムにおいて、この精度で時刻を得られるわけではありません。



# オプションのオペレーティングシステム サービス

この章で説明するモジュールでは、特定のオペレーティングシステムでだけ利用できるオペレーティングシステム機能へのインターフェースを提供します。このインターフェースは、おおむね UNIX や C のインターフェースにならってモデル化してありますが、他のシステム上（Windows や NT など）でも利用することがあります。次に概要を示します。

<code>signal</code>	非同期イベントにハンドラを設定します。
<code>socket</code>	低レベルネットワークインターフェース。
<code>select</code>	複数のストリームに対して I/O 処理の完了を待機します。
<code>thread</code>	1 つのインタープリタの中でのマルチスレッド制御
<code>threading</code>	高レベルスレッドインターフェース
<code>dummy_thread</code>	<code>thread</code> の代替モジュール。
<code>dummy_threading</code>	<code>threading</code> の代替モジュール。
<code>Queue</code>	同期キュークラス
<code>mmap</code>	UNIX と Windows のメモリマップファイルへのインターフェース
<code>anydbm</code>	DBM 形式のデータベースモジュールに対する汎用インタフェース。
<code>dbhash</code>	BSD データベースライブラリへの DBM 形式のインタフェース。
<code>whichdb</code>	どの DBM 形式のモジュールが与えられたデータベースを作ったかを推測する
<code>bsddb</code>	Berkeley DB ライブラリへのインタフェース
<code>dumbdbm</code>	単純な DBM インタフェースに対する可搬性のある実装。
<code>zlib</code>	<code>gzip</code> 互換の圧縮 / 解凍ルーチンへの低レベルインタフェース
<code>gzip</code>	ファイルオブジェクトを用いた <code>gzip</code> 圧縮および解凍のためのインタフェース
<code>bz2</code>	<code>bzip2</code> 互換の圧縮 / 解凍ルーチンへのインタフェース
<code>zipfile</code>	ZIP-フォーマットのアーカイブファイルを読み書きする
<code>tarfile</code>	tar-形式のアーカイブファイルを読み書きします。
<code>readline</code>	Python のための GNU readline サポート。
<code>rlcompleter</code>	GNU readline ライブラリ向けの Python 識別子補完

## 7.1 `signal` — 非同期イベントにハンドラを設定する

このモジュールでは Python でシグナルハンドラを使うための機構を提供します。シグナルとハンドラを扱う上では、一般的なルールがいくつかあります：

- 特定のシグナルに対するハンドラが一度設定されると、明示的にリセットしないかぎり設定されたままになります (Python は背後の実装系に関係なく BSD 形式のインタフェースをエミュレートします)。例外は `SIGCHLD` のハンドラで、この場合は背後の実装系の仕様に従います。

- クリティカルセクションから一時的にシグナルを“ブロック”することはできません。この機能をサポートしない UNIX 系システムも存在するためです。
- Python のシグナルハンドラは Python のユーザが望む限り非同期で呼び出されますが、呼び出されるのは Python インタプリタの“原子的な (atomic)” 命令実行単位の間です。従って、(巨大なサイズのテキストに対する正規表現の一致検索のような) 純粋に C 言語のレベルで実現されている時間のかかる処理中に到着したシグナルは、不定期間遅延する可能性があります。
- シグナルが I/O 操作中に到着すると、シグナルハンドラが処理を返した後に I/O 操作が例外を送出する可能性があります。これは背後にある UNIX システムが割り込みシステムコールにどういう意味付けをしているかに依存します。
- C 言語のシグナルハンドラは常に処理を返すので、SIGFPE や SIGSEGV のような同期エラーの捕捉はほとんど意味がありません。
- Python は標準でごく少数のシグナルハンドラをインストールしています: SIGPIPE は無視されます (従って、パイプやソケットに対する書き込みで生じたエラーは通常の Python 例外として報告されます) SIGINT は KeyboardInterrupt 例外に変換されます。これらはどれも上書きすることができます。
- シグナルとスレッドの両方を同じプログラムで使用する場合にはいくつか注意が必要です。シグナルとスレッドを同時に利用する上で基本的に注意すべきことは: 常に `signal()` 命令は主スレッド (main thread) の処理中で実行するということです。どのスレッドも `alarm()`、`getsignal()`、あるいは `pause()` を実行することができます; しかし、主スレッドだけが新たなシグナルハンドラを設定することができ、従ってシグナルを受け取ることができるのは主スレッドだけです (これは、背後のスレッド実装が個々のスレッドに対するシグナル送信をサポートしているかに関わらず、Python `signal` モジュールが強制している仕様です)。従って、シグナルをスレッド間通信の手段として使うことはできません。代わりにロック機構を使ってください。

以下に `signal` モジュールで定義されている変数を示します:

#### **SIG\_DFL**

二つある標準シグナル処理オプションのうちの一つです; 単にシグナルに対する標準の関数を実行します。例えば、ほとんどのシステムでは、SIGQUIT に対する標準の動作はコアダンプと終了で、SIGCLD に対する標準の動作は単にシグナルの無視です。

#### **SIG\_IGN**

もう一つの標準シグナル処理オプションで、単に受け取ったシグナルを無視します。

#### **SIG\***

全てのシグナル番号はシンボル定義されています。例えば、ハングアップシグナルは `signal.SIGHUP` で定義されています; 変数名は C 言語のプログラムで使われているのと同じ名前、`<signal.h>` にあります。‘`signal()`’ に関する UNIX マニュアルページでは、システムで定義されているシグナルを列挙しています (あるシステムではリストは `signal(2)` に、別のシステムでは `signal(7)` に列挙されています)。全てのシステムで同じシグナル名のセットを定義しているわけではないので注意してください; このモジュールでは、システムで定義されているシグナル名だけを定義しています。

#### **NSIG**

最も大きいシグナル番号に 1 を足した値です。

`signal` モジュールでは以下の関数を定義しています:

#### **alarm(*time*)**

*time* がゼロでない値の場合、この関数は *time* 秒後頃に SIGALRM をプロセスに送るように要求しま

す。それ以前にスケジュールしたアラームはキャンセルされます (常に一つのアラームしかスケジュールできません)。この場合、戻り値は以前に設定されたアラームシグナルが通知されるまであと何秒だったかを示す値です。 *time* がゼロの場合、アラームは一切スケジュールされず、現在スケジュールされているアラームがキャンセルされます。戻り値は以前にスケジュールされたアラームが通知される予定時刻までの残り時間です。戻り値がゼロの場合、現在アラームがスケジュールされていないことを示します。(UNIX マニュアルページ *alarm(2)* を参照してください)。利用可能: UNIX。

**getsignal**(*signalnum*)

シグナル *signalnum* に対する現在のシグナルハンドラを返します。戻り値は呼び出し可能な Python オブジェクトか、`signal.SIG_IGN`、`signal.SIG_DFL`、および `None` といった特殊な値のいずれかです。ここで `signal.SIG_IGN` は以前そのシグナルが無視されていたことを示し、`signal.SIG_DFL` は以前そのシグナルの標準の処理方法が使われていたことを示し、`None` はシグナルハンドラがまだ Python によってインストールされていないことを示します。

**pause**( )

シグナルを受け取るまでプロセスを一時停止します; その後、適切なハンドラが呼び出されます。戻り値はありません。Windows では利用できません。(UNIX マニュアルページ *signal(2)* を参照してください。)

**signal**(*signalnum*, *handler*)

シグナル *signalnum* に対するハンドラを関数 *handler* にします。*handler* は二つの引数 (下記参照) を取る呼び出し可能な Python オブジェクトにするか、`signal.SIG_IGN` あるいは `signal.SIG_DFL` といった特殊な値にすることができます。以前に使われていたシグナルハンドラが返されます (上記の `getsignal()` の記述を参照してください)。(UNIX マニュアルページ *signal(2)* を参照してください。)

複数スレッドの使用が有効な場合、この関数は主スレッドからのみ呼び出すことができます; 主スレッド以外のスレッドで呼び出そうとすると、例外 `ValueError` が送出されます。

*handler* は二つの引数: シグナル番号、および現在のスタックフレーム (`None` またはフレームオブジェクト; フレームオブジェクトについての記述はリファレンスマニュアルを参照してください)、とともに呼び出されます。

### 7.1.1 例

以下は最小限のプログラム例です。この例では `alarm()` を使って、ファイルを開く処理を待つのに費やす時間を制限します; これはそのファイルが電源の入れられていないシリアルデバイスを表している場合に有効で、通常こうした場合には `os.open()` は未定義の期間ハングアップしてしまいます。ここではファイルを開くまで 5 秒間のアラームを設定することで解決しています; ファイルを開く処理が長くかかりすぎると、アラームシグナルが送信され、ハンドラが例外を送出するようになっています。

```

import signal, os

pdef handler(signum, frame):
    print 'Signal handler called with signal', signum
    raise IOError, "Couldn't open device!"

# Set the signal handler and a 5-second alarm
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)

# This open() may hang indefinitely
fd = os.open('/dev/ttyS0', os.O_RDWR)

signal.alarm(0)          # Disable the alarm

```

## 7.2 socket — 低レベルネットワークインターフェース

このモジュールは、Python で BSD ソケット インターフェースを利用するために使用します。最近の UNIX システム、Windows, MacOS, BeOS, OS/2 など、多くのプラットフォームで利用可能です。

C 言語によるソケットプログラミングの基礎については、以下の資料を参照してください。 *An Introductory 4.3BSD Interprocess Communication Tutorial* (Stuart Sechrest), *An Advanced 4.3BSD Interprocess Communication Tutorial* (Samuel J. Leffler 他), *UNIX Programmer's Manual, Supplementary Documents 1* (PS1:7 章 PS1:8 章)。ソケットの詳細については、各プラットフォームのソケット関連システムコールに関するドキュメント (UNIX ではマニュアルページ、Windows では WinSock(または WinSock2) 仕様書) も参照してください。IPv6 対応の API については、RFC 2553 *Basic Socket Interface Extensions for IPv6* を参照してください。

Python インターフェースは、UNIX のソケット用システムコールとライブラリを、そのまま Python のオブジェクト指向スタイルに変換したものです。各種ソケット関連のシステムコールは、`socket()` 関数で生成するソケット オブジェクトのメソッドとして実装されています。メソッドのパラメータは C のインターフェースよりも多少高水準で、例えば `read()` や `write()` メソッドではファイルオブジェクトと同様、受信時のバッファ確保や送信時の出力サイズなどは自動的に処理されます。

ソケットのアドレスは以下のように指定します: 単一の文字列は、AF\_UNIX アドレスファミリを示します。(`host`, `port`) のペアは AF\_INET アドレスファミリを示し、`host` は 'daring.cwi.nl' のようなインターネットドメイン形式または '100.50.200.5' のような IPv4 アドレスを文字列で、`port` はポート番号を整数で指定します。AF\_INET6 アドレスファミリは (`host`, `port`, `flowinfo`, `scopeid`) の長さ 4 のタプルで示し、`flowinfo` と `scopeid` にはそれぞれ C の `struct sockaddr_in6` における `sin6_flowinfo` と `sin6_scope_id` の値を指定します。後方互換性のため、`socket` モジュールのメソッドでは `sin6_flowinfo` と `sin6_scope_id` を省略する事ができますが、`scopeid` を省略するとスコープを持った IPv6 アドレスの処理で問題が発生する場合があります。現在サポートされているアドレスファミリは以上です。ソケットオブジェクトで利用する事のできるアドレス形式は、ソケットオブジェクトの作成時に指定したアドレスファミリで決まります。

IPv4 アドレスのホストアドレスが空文字列の場合、INADDR\_ANY として処理されます。また、 '<broadcast>' の場合は INADDR\_BROADCAST として処理されます。IPv6 では後方互換性のためこの機能は用意されていないので、IPv6 をサポートする Python プログラムでは利用しないで下さい。

IPv4/v6 ソケットの `host` 部にホスト名を指定すると、処理結果が一定ではない場合があります。これは Python は DNS から取得したアドレスのうち最初のアドレスを使用するので、DNS の処理やホストの設定によって異なる IPv4/6 アドレスを取得する場合があるためです。常に同じ結果が必要であれば、`host` に数値のアドレスを指定してください。

エラー時には例外が発生します。引数型のエラーやメモリ不足の場合には通常の例外が発生し、ソケットやアドレス関連のエラーの場合は `socket.error` が発生します。

`setblocking()` メソッドで、非ブロッキングモードを使用することができます。また、より汎用的に `settimeout()` メソッドでタイムアウトを指定する事ができます。

`socket` モジュールでは、以下の定数と関数を提供しています。

#### **exception error**

この例外は、ソケット関連のエラーが発生した場合に送出されます。例外の値は障害の内容を示す文字列か、または `os.error` と同様な `(errno, string)` のペアとなります。オペレーティングシステムで定義されているエラーコードについては `errno` を参照してください。

#### **exception herror**

この例外は、C API の `gethostbyname_ex()` や `gethostbyaddr()` など、`h_errno` のようなアドレス関連のエラーが発生した場合に送出されます。

例外の値は `(h_errno, string)` のペアで、ライブラリの呼び出し結果を返します。`string` は C 関数 `hstrerror()` で取得した、`h_errno` の意味を示す文字列です。

#### **exception gaierror**

この例外は `getaddrinfo()` と `getnameinfo()` でアドレス関連のエラーが発生した場合に送出されます。

例外の値は `(error, string)` のペアで、ライブラリの呼び出し結果を返します。`string` は C 関数 `gai_strerror()` で取得した、`h_errno` の意味を示す文字列です。

#### **exception timeout**

この例外は、あらかじめ `settimeout()` を呼び出してタイムアウトを有効にしてあるソケットでタイムアウトが生じた際に送出されます。例外に付属する値は文字列で、その内容は現状では常に “timed out” となります。2.3 で追加された仕様です。

**AF\_UNIX**

**AF\_INET**

**AF\_INET6**

アドレス（およびプロトコル）ファミリを示す定数で、`socket()` の最初の引数に指定することができます。AF\_UNIX ファミリをサポートしないプラットフォームでは、AF\_UNIX は未定義となります。

**SOCK\_STREAM**

**SOCK\_DGRAM**

**SOCK\_RAW**

**SOCK\_RDM**

**SOCK\_SEQPACKET**

ソケットタイプを示す定数で、`socket()` の 2 番目の引数に指定することができます。（ほとんどの場合、SOCK\_STREAM と SOCK\_DGRAM 以外は必要ありません。）

**SO\_\***

**SOMAXCONN**

**MSG\_\***

**SOL\_\***

**IPPROTO\_\***

**IPPORT\_\***

**INADDR\_\***

**IP\_\***

**IPV6\_\***



EAI\_\*  
AI\_\*  
NI\_\*  
TCP\_\*

UNIX のソケット・IP プロトコルのドキュメントで定義されている各種定数。ソケットオブジェクトの `setsockopt()` や `getsockopt()` で使用します。ほとんどのシンボルは UNIX のヘッダファイルに従っています。一部のシンボルには、デフォルト値を定義してあります。

`has_ipv6`

現在のプラットフォームで IPv6 がサポートされているか否かを示す真偽値。2.3 で追加された仕様です。

`getaddrinfo(host, port[, family[, socktype[, proto[, flags]]])`

*host/port* 引数の指すアドレス情報を解決して、ソケット操作に必要な全ての引数が入った 5 要素のタプルを返します。*host* はドメイン名、IPv4/v6 アドレスの文字列、または `None` です。*port* は 'http' のようなサービス名文字列、ポート番号を表す数値、または `None` です。

これ以外の引数は省略可能で、指定する場合には数値でなければなりません。*host* と *port* に空文字列か `None` を指定すると C API に `NULL` を渡せます。`getaddrinfo()` 関数は以下の構造をとる 5 要素のタプルを返します:

*(family, socktype, proto, canonname, sockaddr)*

*family · socktype · proto* は、`socket()` 関数を呼び出す際に指定する値と同じ整数です。*canonname* は *host* の規準名を示す文字列です。`AI_CANONNAME` を指定した場合、数値による IPv4/v6 アドレスを返します。*sockaddr* は、ソケットアドレスを上述の形式で表すタプルです。この関数の使い方については、`httplib` モジュールなどのソースを参考にしてください。

2.2 で追加された仕様です。

`getfqdn([name])`

*name* の完全修飾ドメイン名を返します。*name* が空または省略された場合、ローカルホストを指定したとみなします。完全修飾ドメイン名の取得にはまず `gethostbyaddr()` でチェックし、次に可能であればエイリアスを調べ、名前にピリオドを含む最初の名前を値として返します。完全修飾ドメイン名を取得できない場合、ホスト名を返します。

2.0 で追加された仕様です。

`gethostbyname(hostname)`

ホスト名を '100.50.200.5' のような IPv4 形式のアドレスに変換します。ホスト名として IPv4 アドレスを指定した場合、その値は変換せずにそのまま返ります。`gethostbyname()` API へのより完全なインターフェースが必要であれば、`gethostbyname_ex()` を参照してください。`gethostbyname()` は、IPv6 名前解決をサポートしていません。IPv4/v6 のデュアルスタックをサポートする場合は `getaddrinfo()` を使用します。

`gethostbyname_ex(hostname)`

ホスト名から、IPv4 形式の各種アドレス情報を取得します。戻り値は *(hostname, aliaslist, ipaddrlist)* のタプルで、*hostname* は *ip\_address* で指定したホストの正式名、*aliaslist* は同じアドレスの別名のリスト (空の場合もある)、*ipaddrlist* は同じホスト上の同一インターフェースの IPv4 アドレスのリスト (ほとんどの場合は単一のアドレスのみ) を示します。`gethostbyname()` は、IPv6 名前解決をサポートしていません。IPv4/v6 のデュアルスタックをサポートする場合は `getaddrinfo()` を使用します。

`gethostname()`

Python インタプリタを現在実行中のマシンのホスト名を示す文字列を取得します。実行中マシンの IP アドレスが必要であれば、`gethostbyname(gethostname())` を使用してください。この処理



は実行中ホストのアドレス-ホスト名変換が可能であることを前提としていますが、常に変換可能であるとは限りません。注意: `gethostname()` は完全修飾ドメイン名を返すとは限りません。完全修飾ドメイン名が必要であれば、`gethostbyaddr(gethostname())` としてください(下記参照)。

**gethostbyaddr**(*ip\_address*)

(*hostname*, *aliaslist*, *ipaddrlist*) のタプルを返し、*hostname* は *ip\_address* で指定したホストの正式名、*aliaslist* は同じアドレスの別名のリスト(空の場合もある)、*ipaddrlist* は同じホスト上の同一インターフェースの IPv4 アドレスのリスト(ほとんどの場合は単一のアドレスのみ)を示します。完全修飾ドメイン名が必要であれば、`getfqdn()` を使用してください。`gethostbyaddr` は、IPv4/IPv6 の両方をサポートしています。

**getnameinfo**(*sockaddr*, *flags*)

ソケットアドレス *sockaddr* から、(*host*, *port*) のタプルを取得します。*flags* の設定に従い、*host* は完全修飾ドメイン名または数値形式アドレスとなります。同様に、*port* は文字列のポート名または数値のポート番号となります。2.2 で追加された仕様です。

**getprotobyname**(*protocolname*)

'icmp' のようなインターネットプロトコル名を、`socket()` の第三引数として指定する事ができる定数に変換します。これは主にソケットを "raw" モード (SOCK\_RAW) でオープンする場合には必要ですが、通常のソケットモードでは第三引数に 0 を指定するか省略すれば正しいプロトコルが自動的に選択されます。

**getservbyname**(*servicename*, *protocolname*)

インターネットサービス名とプロトコルから、そのサービスのポート番号を取得します。プロトコル名として、'tcp' が 'udp' を指定することができます。

**socket**(*family*, *type*[, *proto*])

アドレスファミリ、ソケットタイプ、プロトコル番号を指定してソケットを作成します。アドレスファミリには AF\_INET・AF\_INET6・AF\_UNIX を指定することができます。ソケットタイプには SOCK\_STREAM・SOCK\_DGRAM・または 'SOCK\_' の何れかを指定します。プロトコル番号には通常省略するか、または 0 を指定します。

**ssl**(*sock*[, *keyfile*, *certfile*])

ソケット *sock* による SSL 接続を初期化します。*keyfile* には、PEM フォーマットのプライベートキーファイル名を指定します。*certfile* には、PEM フォーマットの認証チェーンファイル名を指定します。処理が成功すると、新しい SSLObject が返ります。

警告: 証明書の認証は全く行いません。

**fromfd**(*fd*, *family*, *type*[, *proto*])

既存のファイルディスクリプタ(ファイルオブジェクトの `fileno()` で返る整数)から、ソケットオブジェクトを構築します。アドレスファミリとプロトコル番号は `socket()` と同様に指定します。ファイルディスクリプタはソケットを指していなければなりませんが、実際にソケットであるかどうかのチェックは行っていません。このため、ソケット以外のファイルディスクリプタを指定するとその後の処理が失敗する場合があります。この関数が必要な事はあまりありませんが、UNIX の inet デモンのようにソケットを標準入力や標準出力として使用するプログラムで使われます。この関数で使用するソケットは、ブロッキングモードと想定しています。利用可能:UNIX

**ntohl**(*x*)

32 ビット整数のバイトオーダーを、ネットワークバイトオーダーからホストバイトオーダーに変換します。ホストバイトオーダーとネットワークバイトオーダーが一致するマシンでは、この関数は何もしません。それ以外の場合は 4 バイトのスワップを行います。

**ntohs**(*x*)

16 ビット整数のバイトオーダーを、ネットワークバイトオーダーからホストバイトオーダーに変換します。

ホストバイトオーダーとネットワークバイトオーダーが一致するマシンでは、この関数は何もしません。それ以外の場合は2バイトのスワップを行います。

**htonl(*x*)**

32ビット整数のバイトオーダーを、ホストバイトオーダーからネットワークバイトオーダーに変換します。ホストバイトオーダーとネットワークバイトオーダーが一致するマシンでは、この関数は何もしません。それ以外の場合は4バイトのスワップを行います。

**htons(*x*)**

16ビット整数のバイトオーダーを、ホストバイトオーダーからネットワークバイトオーダーに変換します。ホストバイトオーダーとネットワークバイトオーダーが一致するマシンでは、この関数は何もしません。それ以外の場合は2バイトのスワップを行います。

**inet\_aton(*ip\_string*)**

ドット記法によるIPv4アドレス('123.45.67.89'など)を32ビットにパックしたバイナリ形式に変換し、長さ4の文字列として返します。この関数が返す値は、標準Cライブラリのstruct in\_addr型を使用する関数に渡すことができます。

IPv4アドレス文字列が不正であれば、socket.errorが発生します。このチェックは、この関数で使用しているCの実装inet\_aton()で行われます。

inet\_aton()は、IPv6をサポートしません。IPv4/v6のデュアルスタックをサポートする場合はgetnameinfo()を使用します。

**inet\_ntoa(*packed\_ip*)**

32ビットにパックしたバイナリ形式のIPv4アドレスを、ドット記法による文字列('123.45.67.89'など)に変換します。この関数が返す値は、標準Cライブラリのstruct in\_addr型を使用する関数に渡すことができます。

この関数に渡す文字列の長さが4バイト以外であれば、socket.errorが発生します。inet\_ntoa()は、IPv6をサポートしません。IPv4/v6のデュアルスタックをサポートする場合はgetnameinfo()を使用します。

**inet\_pton(*address\_family*, *ip\_string*)**

IPアドレスを、アドレスファミリ固有の文字列からパックしたバイナリ形式に変換します。inet\_pton()は、struct in\_addr型(inet\_aton()と同様)やstruct in6\_addrを使用するライブラリやネットワークプロトコルを呼び出す際に使用することができます。

現在サポートされている*address\_family*は、AF\_INETとAF\_INET6です。*ip\_string*に不正なIPアドレス文字列を指定すると、socket.errorが発生します。有効な*ip\_string*は、*address\_family*とinet\_pton()の実装によって異なります。

利用可能: UNIX (サポートしていないプラットフォームもあります) 2.3 で追加された仕様です。

**inet\_ntop(*address\_family*, *packed\_ip*)**

パックしたIPアドレス(数文字の文字列)を、'7.10.0.5'や'5aef:2b::8'などの標準的な、アドレスファミリ固有の文字列形式に変換します。inet\_ntop()は(inet\_ntoa()と同様に)struct in\_addr型やstruct in6\_addr型のオブジェクトを返すライブラリやネットワークプロトコル等で使用することができます。

現在サポートされている*address\_family*は、AF\_INETとAF\_INET6です。*packed\_ip*の長さが指定したアドレスファミリで適切な長さでなければ、ValueErrorが発生します。inet\_ntop()でエラーとなると、socket.errorが発生します。

利用可能: UNIX (サポートしていないプラットフォームもあります) 2.3 で追加された仕様です。

**getdefaulttimeout()**

新規に生成されたソケットオブジェクトの、デフォルトのタイムアウト値を浮動小数点形式の秒数で

返します。タイムアウトを使用しない場合には `None` を返します。最初に `socket` モジュールがインポートされた時の初期値は `None` です。

2.3 で追加された仕様です。

**`setdefaulttimeout(timeout)`**

新規に生成されたソケットオブジェクトの、デフォルトのタイムアウト値を浮動小数点形式の秒数で指定します。タイムアウトを使用しない場合には `None` を指定します。最初に `socket` モジュールがインポートされた時の初期値は `None` です。

2.3 で追加された仕様です。

**`SocketType`**

ソケットオブジェクトの型を示す型オブジェクト。 `type(socket(...))` と同じです。

参考資料:

`SocketServer` モジュール (11.15 節):

ネットワークサーバの開発を省力化するためのクラス群。

### 7.2.1 socket オブジェクト

ソケットオブジェクトは以下のメソッドを持ちます。 `makefile()` 以外のメソッドは、UNIX のソケット用システムコールに対応しています。

**`accept()`**

接続を受け付けます。ソケットはアドレスに `bind` 済みで、`listen` 中である必要があります。戻り値は `(conn, address)` のペアで、`conn` は接続を通じてデータの送受信を行うための新しいソケットオブジェクト、`address` は接続先でソケットに `bind` しているアドレスを示します。

**`bind(address)`**

ソケットを `address` に `bind` します。 `bind` 済みのソケットを再バインドする事はできません。 `address` のフォーマットはアドレスファミリによって異なります (前述)。注意: 本来、このメソッドは単一のタプルのみを引数として受け付けますが、以前は `AF_INET` アドレスを示す二つの値を指定する事ができました。これは本来の仕様ではなく、Python 2.0 以降では使用することはできません。

**`close()`**

ソケットをクローズします。以降、このソケットでは全ての操作が失敗します。リモート端点ではキューに溜まったデータがフラッシュされた後はそれ以上のデータを受信しません。ソケットはガベージコレクション時に自動的にクローズされます。

**`connect(address)`**

`address` で示されるリモートソケットに接続します。 `address` のフォーマットはアドレスファミリによって異なります (前述)。注意: 本来、このメソッドは単一のタプルのみを引数として受け付けますが、以前は `AF_INET` アドレスを示す二つの値を指定する事ができました。これは本来の仕様ではなく、Python 2.0 以降では使用することはできません。

**`connect_ex(address)`**

`connect(address)` と同様ですが、C 言語の `connect()` 関数の呼び出しでエラーが発生した場合には例外を送出せずにエラーを戻り値として返します。(これ以外の、“host not found,” 等のエラーの場合には例外が発生します。) 処理が正常に終了した場合には `0` を返し、エラー時には `errno` の値を返します。この関数は、非同期接続をサポートする場合などに使用することができます。注意: 本来、このメソッドは単一のタプルのみを引数として受け付けますが、以前は `AF_INET` アドレスを示す二つの値を指定する事ができました。これは本来の仕様ではなく、Python 2.0 以降では使用することはできません。

**`fileno()`**

ソケットのファイルディスクリプタを整数型で返します。ファイルディスクリプタは、`select.select()` などで使用します。

**getpeername()**

ソケットが接続しているリモートアドレスを返します。この関数は、リモート IPv4/v6 ソケットのポート番号を調べる場合などに使用します。*address* のフォーマットはアドレスファミリによって異なります (前述)。この関数をサポートしていないシステムも存在します。

**getsockname()**

ソケット自身のアドレスを返します。この関数は、IPv4/v6 ソケットのポート番号を調べる場合などに使用します。*address* のフォーマットはアドレスファミリによって異なります (前述)。

**getsockopt(level, optname[, buflen])**

ソケットに指定されたオプションを返します (UNIX のマニュアルページ *getsockopt(2)* を参照)。`SO_*` 等のシンボルは、このモジュールで定義しています。*buflen* を省略した場合、取得するオプションは整数とみなし、整数型の値を戻り値とします。*buflen* を指定した場合、長さ *buflen* のバッファでオプションを受け取り、このバッファを文字列として返します。このバッファは、呼び出し元プログラムで `struct` モジュール等を利用して内容を読み取ることができます。

**listen(backlog)**

ソケットを Listen し、接続を待ちます。引数 *backlog* には接続キューの最大の長さ (1 以上) を指定します。*backlog* の最大数はシステムに依存します (通常は 5)。

**makefile([mode[, bufsize]])**

ソケットに関連付けられたファイルオブジェクトを返します (ファイルオブジェクトについては 2.3.8 の “ファイルオブジェクト” を参照)。ファイルオブジェクトはソケットを `dup()` したファイルディスクリプタを使用しており、ソケットオブジェクトとファイルオブジェクトは別々にクローズしたりガベージコレクションで破棄したりする事ができます。ソケットはブロッキングモードでなければなりません。オプション引数の *mode* と *bufsize* には、`file()` 組み込み関数と同じ値を指定します。2.1 の “組み込み関数” を参照してください。

**recv(bufsize[, flags])**

ソケットからデータを受信し、文字列として返します。受信する最大バイト数は、*bufsize* で指定します。*flags* のデフォルト値は 0 です。値の意味については UNIX マニュアルページの *recv(2)* を参照してください。

**recvfrom(bufsize[, flags])**

ソケットからデータを受信し、結果をタプル (*string*, *address*) として返します。*string* は受信データの文字列で、*address* は送信元のアドレスを示します。オプション引数 *flags* の意味は、上記 *recv()* と同じです。*address* のフォーマットはアドレスファミリによって異なります (前述)。

**send(string[, flags])**

ソケットにデータを送信します。ソケットはリモートソケットに接続済みでなければなりません。オプション引数 *flags* の意味は、上記 *recv()* と同じです。戻り値として、送信したバイト数を返します。アプリケーションでは、必ず戻り値をチェックし、全てのデータが送られた事を確認する必要があります。データの一部だけが送信された場合、アプリケーションで残りのデータを再送信してください。

**sendall(string[, flags])**

ソケットにデータを送信します。ソケットはリモートソケットに接続済みでなければなりません。オプション引数 *flags* の意味は、上記 *recv()* と同じです。*send()* と異なり、このメソッドは *string* の全データを送信するか、エラーが発生するまで処理を継続します。正常終了の場合は `None` を返し、エラー発生時には例外が発生します。エラー発生時、送信されたバイト数を調べる事はできません。

**sendto(string[, flags], address)**

ソケットにデータを送信します。このメソッドでは接続先を *address* で指定するので、接続済みではいけません。オプション引数 *flags* の意味は、上記 `recv()` と同じです。戻り値として、送信したバイト数を返します。*address* のフォーマットはアドレスファミリによって異なります (前述)。

#### `setblocking(flag)`

ソケットのブロッキング・非ブロッキングモードを指定します。*flag* が 0 の場合は非ブロッキングモード、0 以外の場合はブロッキングモードとなります。全てのソケットは、初期状態ではブロッキングモードです。非ブロッキングモードでは、`recv()` メソッド呼び出し時に読み込みデータが無かったり `send()` メソッド呼び出し時にデータを処理する事ができないような場合に `error` 例外が発生します。しかし、ブロッキングモードでは呼び出しは処理が行われるまでブロックされます。`s.setblocking(0)` は `s.settimeout(0)` と、`s.setblocking(1)` は `s.settimeout(None)` とそれぞれ同じ意味を持ちます。

#### `settimeout(value)`

ソケットのブロッキング処理のタイムアウト値を指定します。*value* には、正の浮動小数点で秒数を指定するか、もしくは `None` を指定します。浮動小数点値を指定した場合、操作が完了する前に *value* で指定した秒数が経過すると `timeout` が発生します。タイムアウト値に `None` を指定すると、ソケットのタイムアウトを無効にします。`s.settimeout(0.0)` は `s.setblocking(0)` と、`s.settimeout(None)` は `s.setblocking(1)` とそれぞれ同じ意味を持ちます。2.3 で追加された仕様です。

#### `gettimeout()`

ソケットに指定されたタイムアウト値を取得します。タイムアウト値が設定されている場合には浮動小数点型で秒数が、設定されていなければ `None` が返ります。この値は、最後に呼び出された `setblocking()` または `settimeout()` によって設定されます。2.3 で追加された仕様です。

ソケットのブロッキングとタイムアウトについて:ソケットオブジェクトのモードは、ブロッキング・非ブロッキング・タイムアウトの何れかとなります。初期状態では常にブロッキングモードです。ブロッキングモードでは、処理が完了するまでブロックされます。非ブロッキングモードでは、処理を行う事ができなければ (不幸にもシステムによって異なる値の) エラーとなります。タイムアウトモードでは、ソケットに指定したタイムアウトまでに完了しなければ処理は失敗となります。`setblocking()` メソッドは、`settimeout()` の省略形式です。

内部的には、タイムアウトモードではソケットを非ブロッキングモードに設定します。ブロッキングとタイムアウトの設定は、ソケットと同じネットワーク端点へ接続するファイルディスクリプタにも反映されます。この結果、`makefile()` で作成したファイルオブジェクトはブロッキングモードでのみ使用することができます。これは非ブロッキングモードとタイムアウトモードでは、即座に完了しないファイル操作はエラーとなるためです。

#### `setsockopt(level, optname, value)`

ソケットのオプションを設定します (UNIX のマニュアルページ `setsockopt(2)` を参照)。`SO_*` 等のシンボルは、このモジュールで定義しています。*value* には、整数または文字列をバッファとして指定する事ができます。文字列を指定する場合、文字列には適切なビットを設定するようにします。(struct モジュールを利用すれば、C の構造体を文字列にエンコードする事ができます。)

#### `shutdown(how)`

接続の片方向、または両方向を切断します。*how* が 0 の場合、以降は受信を行えません。*how* が 1 の場合、以降は送信を行えません。*how* が 2 の場合、以降は送受信を行えません。

`read()` メソッドと `write()` メソッドは存在しませんので注意してください。代わりに *flags* を省略した `recv()` と `send()` を使うことができます。



## 7.2.2 SSL オブジェクト

SSL オブジェクトには、以下のメソッドがあります。

`write(s)`

文字列 *s* を SSL 接続で出力します。戻り値として、送信したバイト数を返します。

`read([n])`

SSL 接続からデータを受信します。*n* を指定した場合は指定したバイト数のデータを受信し、省略時は EOF まで読み込みます。戻り値として、受信したバイト列の文字列を返します。

## 7.2.3 例

以下は TCP/IP プロトコルの簡単なサンプルとして、受信したデータをクライアントにそのまま返送するサーバ(接続可能なクライアントは一件のみ)と、サーバに接続するクライアントの例を示します。サーバでは、`socket().bind().listen().accept()` を実行し(複数のクライアントからの接続を受け付ける場合、`accept()` を複数回呼び出します)、クライアントでは `socket()` と `connect()` だけ呼び出しています。サーバでは `send()/recv()` メソッドは `listen` 中のソケットで実行するのではなく、`accept()` で取得したソケットに対して実行している点にも注意してください。

次のクライアントとサーバは、IPv4 のみをサポートしています。

```
# Echo server program
import socket

HOST = ''                      # Symbolic name meaning the local host
PORT = 50007                   # Arbitrary non-privileged port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
print 'Connected by', addr
while 1:
    data = conn.recv(1024)
    if not data: break
    conn.send(data)
conn.close()

# Echo client program
import socket

HOST = 'daring.cwi.nl'         # The remote host
PORT = 50007                   # The same port as used by the server
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.send('Hello, world')
data = s.recv(1024)
s.close()
print 'Received', 'data'
```

The next two examples are identical to the above two, but support both IPv4 and IPv6. The server side will listen to the first address family available (it should listen to both instead). On most of IPv6-ready systems, IPv6 will take precedence and the server may not accept IPv4 traffic. The client side will try to connect to the all addresses returned as a result of the name resolution, and sends traffic to the first one connected successfully.

次のサンプルは上記のサンプルとほとんど同じですが、IPv4 と IPv6 の両方をサポートしています。サー



バでは、IPv4/v6 の両方ではなく、利用可能な最初のアドレスファミリーだけを listen しています。ほとんどの IPv6 対応システムでは IPv6 が先に現れるため、サーバは IPv4 には応答しません。クライアントでは名前解決の結果として取得したアドレスに順次接続を試み、最初に接続に成功したソケットにデータを送信しています。

```
# Echo server program
import socket
import sys

HOST = ''          # Symbolic name meaning the local host
PORT = 50007       # Arbitrary non-privileged port
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.SOCK_STREAM, 0, socket.AI_PASSIVE):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except socket.error, msg:
        s = None
        continue
    try:
        s.bind(sa)
        s.listen(1)
    except socket.error, msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print 'could not open socket'
    sys.exit(1)
conn, addr = s.accept()
print 'Connected by', addr
while 1:
    data = conn.recv(1024)
    if not data: break
    conn.send(data)
conn.close()
```

```

# Echo client program
import socket
import sys

HOST = 'daring.cwi.nl'      # The remote host
PORT = 50007                # The same port as used by the server
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.SOCK_STREAM):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
        except socket.error, msg:
        s = None
        continue
    try:
        s.connect(sa)
        except socket.error, msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print 'could not open socket'
    sys.exit(1)
s.send('Hello, world')
data = s.recv(1024)
s.close()
print 'Received', 'data'

```

## 7.3 select — I/O 処理の完了を待機する

このモジュールでは、ほとんどのオペレーティングシステムで利用可能な `select()` および `poll()` 関数へのアクセス機構を提供します。Windows の上ではソケットに対してしか動作しないので注意してください; その他のオペレーティングシステムでは、他のファイル形式でも (特に UNIX ではパイプにも) 動作します。通常のファイルに対して適用し、最後にファイルを読み出した時から内容が増えているかを決定するために使うことはできません。

このモジュールでは以下の内容を定義しています:

### **exception error**

エラーが発生したときに送出される例外です。エラーに付属する値は、`errno` からとったエラーコードを表す数値とそのエラーコードに対応する文字列からなるペアで、C 関数の `perror()` が出力するものと同様です。

### **poll()**

(全てのオペレーティングシステムでサポートされているわけではありません。) ポーリングオブジェクトを返します。このオブジェクトはファイル記述子を登録したり登録解除したりすることができ、ファイル記述子に対する I/O イベント発生をポーリングすることができます; ポーリングオブジェクトが提供しているメソッドについては下記の [7.3.1 節](#) を参照してください。

### **select(iwtd, owtd, ewtd[, timeout])**

UNIX の `select()` システムコールに対する直接的なインタフェースです。最初の 3 つの引数は '待機可能なオブジェクト' からなるリストです: ファイル記述子を表す整数値、または引数を持たず、整数を返すメソッド `fileno()` を持つオブジェクトです。待機可能なオブジェクトの 3 つのリストは

それぞれ入力、出力、そして‘例外状態’に対応します。いずれかに空のリストを指定してもかまいませんが、3 つ全てを空のリストにしてもよいかどうかはプラットフォームに依存します (UNIX では動作し、Windows では動作しないことが知られています)。オプションの *timeout* 引数にはタイムアウトまでの秒数を浮動小数点数型で指定します。 *timeout* 引数が省略された場合、関数は少なくとも一つのファイル記述子が何らかの準備完了状態になるまでブロックします。タイムアウト値ゼロは、ポーリングを行いブロックしないことを示します。

戻り値は準備完了状態のオブジェクトからなる 3 つのリストです: 従ってこのリストはそれぞれ関数の最初の 3 つの引数のサブセットになります。ファイル記述子のいずれも準備完了にならないままタイムアウトした場合、3 つの空のリストが返されます。

リストの中に含めることのできるオブジェクトは Python ファイルオブジェクト (すなわち `sys.stdin`, あるいは `open()` や `os.popen()` が返すオブジェクト)、`socket.socket()` が返すソケットオブジェクト です。 *wrapper* クラスを自分で定義することもできます。この場合、適切な (単なる乱数ではなく本当のファイル記述子を返す) `fileno()` メソッドを持つ必要があります注意: `select` は Windows のファイルオブジェクトを受理しませんが、ソケットは受理します。 Windows では、背後の `select()` 関数は WinSock ライブラリで提供されており、WinSock によって生成されたものではないファイル記述子を扱うことができないのです。

### 7.3.1 ポーリングオブジェクト

`poll()` システムコールはほとんどの UNIX システムでサポートされており、非常に多数のクライアントに同時にサービスを提供するようなネットワークサーバが高い拡張性を持てるようにしています。 `poll()` に高い拡張性があるのは、`select()` がビット対応表を構築し、対象ファイルの記述子に対応するビットを立て、その後全ての対応表の全てのビットを線形探索するのに対し、 `poll()` は対象のファイル記述子を列挙するだけでよいからです。 `select()` は  $O(\text{最大のファイル記述子番号})$  ののに対し、 `poll()` は  $O(\text{対象とするファイル記述子の数})$  で済みます。

`register(fd, eventmask)`

ファイル記述子をポーリングオブジェクトに登録します。これ以降の `poll()` メソッド呼び出しでは、そのファイル記述子に処理待ち中の I/O イベントがあるかどうかを監視します。 *fd* は整数か、整数値を返す `fileno()` メソッドを持つオブジェクトを取ります。ファイルオブジェクトも通常 `fileno()` を実装しているので、引数として使うことができます。

*eventmask* はオプションのビットマスクで、どのタイプの I/O イベントを監視したいかを記述します。この値は以下の表で述べる定数 `POLLIN`、`POLLPRI`、および `POLLOUT` の組み合わせにすることができます。ビットマスクを指定しない場合、標準の値が使われ、3 種のイベント全てに対して監視が行われます。

定数	意味
<code>POLLIN</code>	読み出せるデータの存在
<code>POLLPRI</code>	緊急の読み出しデータの存在
<code>POLLOUT</code>	書き出せるかどうか: 書き出し処理がブロックしないかどうか
<code>POLLERR</code>	何らかのエラー状態
<code>POLLHUP</code>	ハングアップ
<code>POLLNVAL</code>	無効な要求: 記述子が開かれていない

すでに登録済みのファイル記述子を登録してもエラーにはならず、一度だけ登録した場合と同じ効果になります。

`unregister(fd)`

ポーリングオブジェクトによって追跡中のファイル記述子を登録解除します。 `register()` メソッドと同様に、 *fd* は整数か、整数値を返す `fileno()` メソッドを持つオブジェクトを取ります。

登録されていないファイル記述子を登録解除しようとするとき `KeyError` 例外が送出されます。

`poll([timeout])`

登録されたファイル記述子に対してポーリングを行い、報告すべき I/O イベントまたはエラーの発生したファイル記述子に毎に 2 要素のタプル (`fd`, `event`) からなるリストを返します。リストは空になることもあります。`fd` はファイル記述子で、`event` は該当するファイル記述子について報告されたイベントを表すビットマスクです — 例えば `POLLIN` は入力待ちを示し、`POLLOUT` はファイル記述子に対する書き込みが可能を示す、などです。空のリストは呼び出しがタイムアウトしたか、報告すべきイベントがどのファイル記述子でも発生しなかったことを示します。`timeout` が与えられた場合、処理を戻すまで待機する時間の長さをミリ秒単位で指定します。`timeout` が省略されたり、負の値であったり、あるいは `None` の場合、そのポーリングオブジェクトが監視している何らかのイベントが発生するまでブロックします。

## 7.4 thread — マルチスレッドのコントロール

このモジュールはマルチスレッド (別名 軽量プロセス (*light-weight processes*) またはタスク (*tasks*)) に用いられる低レベルプリミティブを提供します — グローバルデータ空間を共有するマルチスレッドを制御します。同期のための単純なロック (別名 *mutexes* またはバイナリセマフォ (*binary semaphores*)) が提供されています。

このモジュールはオプションです。Windows, Linux, SGI IRIX, Solaris 2.x、そして同じような POSIX スレッド (別名 “pthread”) 実装のシステム上でサポートされます。`thread` を使用することのできないシステムでは、`dummy_thread` が用意されています。`dummy_thread` はこのモジュールと同じインターフェースを持ち、置き換えて使用することができます。

定数と関数は以下のように定義されています:

**exception error**

スレッド特有のエラーで送出されます。

**LockType**

これはロックオブジェクトのタイプです。

`start_new_thread(function, args[, kwargs])`

新しいスレッドを開始して、その ID を返します。スレッドは引数リスト `args` (タプルでなければなりません) の関数 `function` を実行します。オプション引数 `kwargs` はキーワード引数の辞書を指定します。関数が戻るとき、スレッドは黙って終了します。関数が未定義の例外でターミネートしたとき、スタックトレースが表示され、そしてスレッドが終了します (しかし他のスレッドは走り続けます)。

`interrupt_main()`

メインスレッドで `KeyboardInterrupt` を送出します。サブスレッドはこの関数を使ってメインスレッドに割り込みをかけることができます。2.3 で追加された仕様です。

`exit()`

`SystemExit` 例外を送出します。それが捕えられないときは、黙ってスレッドを終了させます。

`allocate_lock()`

新しいロックオブジェクトを返します。ロックのメソッドはこの後に記述されます。ロックは初期状態としてアンロック状態です。

`get_ident()`

現在のスレッドの ‘スレッド ID’ を返します。これは 0 でない整数です。この値は直接の意味を持っていません; 例えばスレッド特有のデータの辞書に索引をつけるためのような、マジッククッキーとして意図されています。スレッドが終了し、他のスレッドが作られたとき、スレッド ID は再利用さ

れるかもしれません。

ロックオブジェクトは次のようなメソッドを持っています:

`acquire([waitflag])`

オプションの引数なしで使用すると、このメソッドは他のスレッドがロックしているかどうかにかかわらずロックを獲得し、`None` を返します。ただし他のスレッドがすでにロックしている場合には解除されるまで待つからロックを獲得します (同時にロックを獲得できるスレッドはひとつだけであり、これこそがロックの存在理由です)。整数の引数 `waitflag` を指定すると、その値によって動作が変わります。引数が `0` のときは、待たずにすぐ獲得できる場合にだけロックを獲得します。`0` 以外の値を与えると、先の例と同様、ロックの状態にかかわらず獲得をおこないます。なお、引数を与えた場合、ロックを獲得すると `True`、できなかったときには `False` を返します。

`release()`

ロックを解放します。そのロックは既に獲得されたものでなければなりません、しかし同じスレッドによって獲得されたものである必要はありません。

`locked()`

ロックの状態を返します: 同じスレッドによって獲得されたものなら `True`、違うのなら `False` を返します。

#### Caveats:

- スレッドは割り込みと奇妙な相互作用をします: `KeyboardInterrupt` 例外は任意のスレッドによって受け取られます。(signal モジュールが利用可能なとき、割り込みは常にメインスレッドへ行きます。)
- `sys.exit()` を呼び出す、あるいは `SystemExit` 例外を送出することは、`exit()` を呼び出すことと同じです。
- I/O 待ちをブロックするかもしれない全ての組込み関数が、他のスレッドの走行を許すわけではありません。(ほとんどの一般的なもの (`time.sleep()`, `file.read()`, `select.select()`) は期待通りに働きます。)
- ロックの `acquire()` メソッドに割り込むことはできません— `KeyboardInterrupt` 例外は、ロックが獲得された後に発生します。
- メインスレッドが終了したとき、他のスレッドが生き残るかどうかは、システムが定義します。ネイティブスレッド実装を使う SGI IRIX では生き残ります。その他の多くのシステムでは、`try ... finally` 節を実行せずに殺されたり、デストラクタを実行せずに殺されたりします。
- メインスレッドが終了したとき、その通常のクリーンアップは行なわれず (`try ... finally` 節が尊重されることは除きます)、標準 I/O ファイルはフラッシュされません。

## 7.5 threading — 高レベルスレッドインターフェース

このモジュールは、低レベルな `thread` モジュールの上に、高レベルスレッドインターフェースを構築しています。

`thread` が存在しないために `threading` を利用できない場合、代わりに `refmodule[dummythreading]dummy_threading` を使用することができます。

関数とオブジェクトは次のように定義されています:

**activeCount()**

現在のアクティブな Thread オブジェクトの数を返します。この返された数は、enumerate() から返されるリストの長さと等しいです。現在のアクティブなスレッドの数を返す関数です。

**Condition()**

新しい条件変数オブジェクトを返すファクトリー関数です。条件変数は、一つ以上のスレッドが他のスレッドによって通知されるまで待つことを許します。

**currentThread()**

呼んだ関数のスレッドに相当する、現在の Thread オブジェクトを返します。その関数のスレッドが threading モジュールを使って作られていなければ、機能を制限したダミースレッドオブジェクトが返されます。

**enumerate()**

現在のすべてのアクティブ Thread オブジェクトのリストを返します。このリストは、デーモンスレッド(currentThread() によって作られたダミースレッドオブジェクト)と、メインスレッドを含みます。このリストは、ターミネートしたスレッドと、まだ開始していないスレッドを除きます。

**Event()**

新しいイベントオブジェクトを返すファクトリー関数です。イベントは、set() メソッドで True にセットでき、clear() メソッドで False にリセットできるフラグを扱います。wait() メソッドは、フラグが True になるまでブロックします。

**Lock()**

新しいプリミティブロックオブジェクトを返すファクトリー関数です。いったんスレッドがロックを獲得すると、それが解放されるまで次の獲得試行をブロックします; どのスレッドがそれを解放してもかまいません。

**RLock()**

新しい再入可能ロックオブジェクトを返すファクトリー関数です。再入可能ロックはそれを獲得したスレッドによって解放されなければなりません。いったんスレッドが再入可能ロックを獲得すると、同じスレッドはブロックされずにもう一度それを獲得できます; そのスレッドは獲得した回数だけ解放しなければいけません。

**Semaphore([value])**

新しいセマフォオブジェクトを返すファクトリー関数です。セマフォは、release() を呼び出した数から、acquire() を呼び出した数を引いて、初期値を足した数を反映するカウンターを扱います。acquire() メソッドは、カウンターが負にならずに返ることができるまで、ブロックします。もし value が与えられなかったら、デフォルトに 1 を使います。

**BoundedSemaphore([value])**

新しい有限セマフォオブジェクトを返すファクトリー関数です。有限セマフォは、現在値が初期値を超過することを阻止します。もし超過した場合、ValueError が投げられます。ほとんどの状況で、セマフォは限りある容量しかもたない資源を保護するために使用されます。セマフォがあまりにも何度も解放される場合、それはバグの表れです。もし value が与えられなかったら、デフォルトに 1 を使います。

**class Thread**

制御しているスレッドを表すクラスです。このクラスは制限のある範囲内で安全にサブクラス化できます。

**class Timer**

指定した時間が経過しか後に、関数を実行するスレッドです。

**settrace(func)**

threading モジュールで開始された全てのスレッドにトレース関数 を設定します。func は各スレ



ドの `run()` が呼び出される以前に `sys.settrace()` に渡されます。2.3 で追加された仕様です。

#### `setprofile(func)`

`threading` モジュールで開始された全てのスレッドにプロファイル関数 を設定します。*func* は各スレッドの `run()` が呼び出される以前に `sys.settrace()` に渡されます。2.3 で追加された仕様です。

オブジェクトの詳細なインターフェースを以下に説明します。

このモジュールのデザインは Java スレッドモデルを適当に使いました。しかし、Java はロックと条件変数をすべてのオブジェクトの基本的な振舞いにしましたが、Python ではそれらは独立したオブジェクトです。Python の `Thread` クラスは Java スレッドクラスの振舞いのサブセットを提供します; 現在、優先スレッド、スレッドグループはありません。スレッドの `destroy`, `stop`, `suspend`, `resume`, `interrupt` などではできません。Java スレッドクラスの `static` メソッドは、実装されたときに、モジュールレベルの関数にマップされています。

以下に記述された全てのメソッドはアトミックに実行されます。

### 7.5.1 Lock オブジェクト

プリミティブロックはロックされたとき、特定のスレッドによって所有されない同期プリミティブです。Python では現在、最も低レベルのプリミティブロックが利用可能であり、`thread` 拡張モジュールによって直接実装されています。

プリミティブロックは2つの状態 – “ロック” または “アンロック” – を持ちます。ロックはアンロック状態で作られます。ロックは2つの基本メソッド、`acquire()` と `release()` を持ちます。アンロック状態のとき、`acquire()` はロック状態へ変更し直ちにリターンします。ロック状態のとき、`acquire()` は、他のスレッドが `release()` を呼出しアンロック状態へ変更するまで、ブロックします。そして、`acquire()` 呼出しはロック状態へリセットしてリターンします。`release()` メソッドはロック状態のときに呼ぶべきです; それはアンロック状態へ変更すると直ちにリターンします。1つ以上のスレッドが、アンロック状態に変更されるのを待つため `acquire()` でブロックされているときに、`release()` でアンロック状態へリセットされると、1つのスレッドのみが先に進めます; 待っているスレッドのどれが進めるかは定義されておらず、実装によって異なることでしょう。

全てのメソッドはアトミックに実行されます。

#### `acquire([blocking = 1])`

ブロックまたは非ブロックのロックを獲得します。

引数なしで呼出すと、ロックがアンロックされるまでブロックして、そしてロック状態へセットしてリターンします。この場合戻り値はありません。

引数 *blocking* を `True` にセットして呼び出すと、引数なしで呼び出したときと同じことを行ない、`True` を返します。

引数 *blocking* を `False` にセットして呼び出すと、ブロックしません。引数のない呼び出しにおいてブロックされる状況ならば、直ちに `False` を返します; そうでない状況ならば、引数なしで呼び出したときと同じことを行ない、`True` を返します。

#### `release()`

ロックを解放します。

ロック状態のとき、アンロック状態へリセットして返ります。ロックがアンロック状態になるのを待っている他のスレッドが、ブロックされていたとき、厳密に1つだけが進むことを許可します。

ロックがアンロック状態のとき、このメソッドを呼び出してはいけません。

戻り値はありません。

### 7.5.2 RLock オブジェクト

再入可能ロックは同じスレッドによって複数回獲得される同期プリミティブです。それは内部に、プリミティブロックによって使われるロック/アンロック状態に加えて、“所有スレッド”と“再帰レベル”の概念を使います。ロック状態ではあるスレッドがロックを所有し、アンロック状態ではどのスレッドもそれを所有しません。

ロックすると、スレッドはその `acquire()` メソッドを呼びます。いったんスレッドがロックを所有すると返ります。アンロックすると、スレッドはその `release()` メソッドを呼びます。`acquire()/release()` 呼び出しペアはネストされるべきです。最後の `release()` だけは(最も外側のペアの `release()`)はロックをアンロック状態へリセットします。そして `acquire()` でブロックされた別のスレッドが進むことを許可します。

`acquire([blocking = 1])`

ブロックまたは非ブロックのロックを獲得します。

引数なしで呼び出されたとき: もしこのスレッドが既にロックを所有する場合、それによって再帰レベルをインクリメントし、直ちに返ります。一方、別のスレッドがロックを所有する場合、アンロックされるまでブロックします。いったんそのロックがアンロックされた(どのスレッドによっても所有されていない)とき、所有権を掴み、再帰レベルを1にセットし、返ります。1つ以上のスレッドが、そのロックがアンロックされるのを待ち、ブロックされているとき、ただ1つのスレッドがロックの所有権を掴むことができます。この場合、返り値はありません。

`blocking` 引数が `True` にセットされて呼び出されたとき、引数無しで呼び出したときと同じように動作し、`true` を返します。

`blocking` 引数が `False` にセットされて呼び出されたとき、ブロックしません。もし引数無しの呼び出しにおいてブロックされるような状況ならば、直ちに `False` を返します; そうでない状況ならば、引数無しの呼出しと同じように動作して、`True` を返します。

`release()`

ロックを解放し、再帰レベルをデクリメントします。デクリメントの後にそれが0になるならば、ロックをアンロックします(他のスレッドによって所有されていない場合)。そして他のスレッドがブロックされていたならば、それらのうちの1つに進むことを許可します。デクリメントの後も再帰レベルが非0であるならば、引続きロック状態であり呼出しスレッドによって所有されている。

呼出しスレッドがロックを所有しているとき、単にこのメソッドを呼出してください。アンロック状態のとき、このメソッドを呼出してはいけません。

返り値はありません。

### 7.5.3 Condition オブジェクト

条件変数 (condition variable) はつねにある種類のロックに関係しています; 明示的にロックを渡すこともできますし、デフォルトによって暗黙で作成させることもできます。(いくつかの条件変数が同じロックを共有しなければならない場合、そのロックを明示的に渡すことができます。)

条件変数は、関連するロックと対応するメソッドを呼出す `acquire()` と `release()` を持っています。加えて `wait()`, `notify()`, `notifyAll()` メソッドを持っています。呼出しスレッドがロックを獲得しているときにだけ、この3つのメソッドを呼出すようにしなければいけません。

`wait()` メソッドはロックを解放し、他のスレッドで同じ条件変数を `notify()` または `notifyAll()` によって起こされるまでブロックします。いったん起こされるとそれはロックを再獲得して戻ります。タイムアウトを指定することも可能です。

`notify()` メソッドは条件変数を待っているスレッドを 1 つ起こします。`notifyAll()` メソッドは条件変数を待っている全てのスレッドを起こします。

Note: `notify()` と `notifyAll()` はロックを解放しません。これは次のことを意味します; スレッドが起こされたとき `wait()` 呼出しから直ちに帰らず、`notify()` または `notifyAll()` を呼出したスレッドが最終的にロックの所有権を譲ることで戻ります。

Tip: 条件変数を使う典型的なプログラミングスタイルは、共有状態へ同期アクセスするためにロックを使います。状態の特定の変更に興味のあるスレッドは、望む状態になるまで `wait()` を繰り返し呼出します。同時に、待ちスレッドのうちの 1 つが望む状態に対して、状態を変更するスレッドが `notify()` または `notifyAll()` を呼出します。例えば、以下のコードはバッファ容量に限界のない場合における一般的な生産者-消費者問題です。

```
# Consume one item
cv.acquire()
while not an_item_is_available():
    cv.wait()
get_an_available_item()
cv.release()

# Produce one item
cv.acquire()
make_an_item_available()
cv.notify()
cv.release()
```

`notify()` と `notifyAll()` の選択は、その状態が 1 つだけあるいは複数の待ちスレッドに興味があるかによって考慮します。例えば典型的な生産者-消費者問題で、バッファに 1 つのアイテムを加えたとき、1 つだけ消費者スレッドを起こす必要があります。

`class Condition([lock])`

もし `lock` 引数を `None` 以外で与えるならば、`Lock` または `RLock` オブジェクトでなければならず、それは基礎ロックとして使われます。それ以外のとき、新しい `RLock` オブジェクトを作り基礎ロックとして使われます。

`acquire(*args)`

基礎ロックを獲得します。このメソッドは基礎ロックに関連するメソッドを呼出します。返回值はそのメソッドからのものです。

`release()`

基礎ロックを解放します。このメソッドは基礎ロックに関連するメソッドを呼出します。返回值はありません。

`wait([timeout])`

通知を受ける (`notify`) あるいはタイムアウトするまで待ちます。これは呼出しスレッドがロックを獲得しているときにだけ、呼出することができます。

このメソッドは基礎ロックを解放し、他のスレッドで同じ条件変数の `notify()` または `notifyAll()` 呼出しによって起こされるまで、あるいはタイムアウトするまでブロックします。いったん起こされたりタイムアウトしたなら、ロックを再獲得して戻ります。

`timeout` 引数が `None` 以外で与えられたとき、それはタイムアウトを秒 (またはその分数) で指定する浮動小数であるべきです。

基礎ロックが `RLock` の場合、それが複数回再帰的に獲得されたときにアンロックされないの `release()` メソッドを使っては解放されません。代わりに、`RLock` クラスの内部インターフェースが使うことで、実際には再帰的に複数回獲得されたときにアンロックします。そして他の内部イン

ターフェースはロックが再獲得されたときに、再帰レベルを元に戻すことで使用されます。

#### **notify()**

もし存在するなら、この条件を待っているスレッドを起こします。これは呼出しスレッドがロックを獲得しているときにだけ、呼出すようにしなければいけません。

もし待っているスレッドがあるのなら、このメソッドは条件変数を待っているスレッドのうちの1つを起こします。もし待っているスレッドがないのなら、なにもしません。

もし待っているスレッドがあるのなら、現在の実装ではまさしく1つのスレッドを起こします。しかしこの振舞いに依存するのは安全ではありません。将来、最適化された実装は1つ以上のスレッドを起こすかもしれません。

Note: 起こされたスレッドは実際にロックを再獲得できるまで wait() 呼出しから戻りません。notify() はロックを解放しないからです。

#### **notifyAll()**

この条件を待っているすべてのスレッドを起こします。このメソッドは notify() のように作用しますが、1つではなくすべての待ちスレッドを起こします。

### 7.5.4 Semaphore オブジェクト

これはコンピュータ科学の歴史の中で最も古い同期プリミティブの1つであり、オランダのコンピュータ科学者 Edsger W. Dijkstra によって発明されました (彼は acquire() と release() の代わりに P() と V() を使いました)。

セマフォは acquire() でデクリメントされ release() でインクリメントされる内部カウンタを管理します。カウンタが0より小さくなることは決してありません。カウンタが0であることを acquire() メソッドが見いだしたとき、他のスレッドが release() を呼び出すまでブロックされます。

#### **class Semaphore([value])**

オプション引数は内部カウンタの初期値を与えます; デフォルトは1です。

#### **acquire([blocking])**

セマフォを獲得します。

引数なしで呼出したとき: 内部カウンタが0より大きいならば、1だけデクリメントし直ちに帰ります。内部カウンタが0ならば、他のスレッドが release() を呼出しカウンタを0より大きくするまでブロックされます。複数の acquire() 呼出しがブロックされる場合、release() がそれらのうちのただ1つを起こすよう、適切な内部ロックを行ないます。その実装はランダムに1つを選び出すかもしれません。そのためブロックされたスレッドが起こされる順序に依存してはなりません。このケースの返り値はありません。

blocking を True で呼出したとき: 引数なしで呼出した場合と同じことを行ない、True を返します。

blocking を False で呼出したとき: ブロックされません。引数なし呼出しにおいてブロックされるような状況ならば、直ちに False を返します。そうでないなら引数なし呼出しと同じことを行ない、True を返します。

#### **release()**

セマフォを解放し、内部カウンタを1インクリメントします。カウンタが0で、他のスレッドが0より大きくなるのを待っているのなら、そのスレッドは起きます。

#### Semaphore の例

セマフォはよく有限なリソースを保護するために使われます。例えばデータベースサーバーなどです。リソースのサイズが固定されているすべての状況で、有限セマフォを使用すべきです。様々な労働者スレ

ドを産み出す前に、あなたのメインスレッドはセマフォを初期化すべきです:

```
maxconnections = 5
...
pool_sema = BoundedSemaphore(value=maxconnections)
```

いったん産み出せば、サーバーへ接続する必要が生じたときに、労働者スレッドはセマフォの `acquire` と `release` を呼出します:

```
pool_sema.acquire()
conn = connectdb()
... use connection ...
conn.close()
pool_sema.release()
```

有限セマフォの使用は、獲得した以上に解放したことを検出できないというプログラミングエラーを減らします。

### 7.5.5 Event オブジェクト

これはスレッド間で通信するための最も簡単なメカニズムの 1 つです: あるスレッドがイベントを発信し、他のスレッドがそれを待ちます。

イベントオブジェクトは、`set()` メソッドで `True` にセットし、`clear()` メソッドで `False` にリセットする、内部フラグを管理します。`wait()` メソッドはフラグが `True` になるまでブロックします。

**class Event()**

内部フラグを `False` に初期化します。

**isSet()**

内部フラグが `True` の場合、そして `True` の場合のみ、`True` を返します。

**set()**

内部フラグを `True` にセットします。それが `True` になるのを待っている全てのスレッドは起こされます。`wait()` を呼出したスレッドは、いったんフラグが `True` になると全てがブロックされなくなります。

**clear()**

内部フラグを `False` にリセットします。その後、`set()` が内部フラグを再び `True` にセットするために呼出されるまで、`wait()` を呼出したスレッドはブロックされるでしょう。

**wait([timeout])**

内部フラグが `True` になるまでブロックします。もし内部フラグが `True` だった場合、直ちに返ります。そうでなかった場合、他のスレッドが `set()` を呼出しフラグを `True` にセットするまで、あるいはオプション `timeout` が発生するまで、ブロックします。

`timeout` 引数が `None` 以外で与えられたとき、それはタイムアウトを秒 (またはその分数) で指定する浮動小数です。

### 7.5.6 Thread オブジェクト

このクラスは個別のスレッド中で実行される活動を表わします。その活動を決定する方法は 2 つあり、呼出し可能オブジェクトをコンストラクタへ渡す、またはサブクラスで `run()` メソッドをオーバーライドするということです。他のメソッド (コンストラクタを除く) はサブクラスでオーバーライドしてはなりません。



ん。言いかえれば、このクラスの `__init__()` と `run()` メソッドだけをオーバーライドしてください。

いったんスレッドオブジェクトが作成されれば、その活動をスレッドの `start()` メソッドを呼ぶことによりスタートさせなければなりません。これは、それぞれのスレッドの `run()` メソッドを起動します。

スレッドの活動が始まると、そのスレッドは‘生きていて (alive)’そして‘活動している (active)’と考えられます (これらの概念は、ほとんど同じであるが、しかし全く同じではない; それらの定義は故意に多少曖昧になっています)。それは `run()` メソッドが終了した時に、または `unhandled` 例外が送出された時に、生きること活動することをやめます。スレッドが生きていても `isAlive()` メソッドはテストします。

他のスレッドはスレッドの `join()` メソッドを呼ぶことができます。 `join()` メソッドを呼ばれたスレッドが終了するまで、呼出しスレッドをブロックします。

スレッドは名前を持っています。その名前はコンストラクタへ渡すことができ、 `setName()` メソッドでセットし、 `getName()` メソッドで得ることができます。

スレッドは“デーモンスレッド”としてフラグすることができます。このフラグの意味は、デーモンスレッドだけが残されたとき、Python プログラム全体が終了するということです。初期値は作成するスレッドから継承されます。そのフラグは `setDaemon()` メソッドでセットされ、 `isDaemon()` メソッドで得ることができます。

“メインスレッド”オブジェクトがあります。これは Python プログラムの最初のスレッドに相当します。それはデーモンスレッドではありません。

“ダミースレッドオブジェクト”を作成することができます。これは“エイリアンスレッド”に相当するスレッドオブジェクトです。スレッドモジュールの外部でスタートするスレッドで、直接 C コードで実装されているようなものです。ダミースレッドオブジェクトは機能が制限されています。それらは常に、生きていて (alive)、活動していて (active)、デーモン (daemon) であるが、 `join()` することはできません。エイリアンスレッドの終了は検知することが不可能であるので、それらは削除されません。

**class Thread**(*group=None, target=None, name=None, args=(), kwargs={}*)

このコンストラクタは常に引数とともに呼ばれるべきです。引数:

*group* は `None` でなければいけません。 `ThreadGroup` クラスが実装されたときのための、将来の拡張として予約されています。

*target* は `run()` メソッドによって呼出される、呼出し可能オブジェクトです。デフォルトは `None` であり、何も呼出さないことを意味します。

*name* はスレッドの名前です。デフォルトによって、ユニークな名前は “Thread-*N*” の形式で構築されます。 *N* は小さな 10 進数です。

*args* は *target* を呼出すときに渡される引数のタプルです。デフォルトは `()` です。

*kwargs* は *target* を呼出すときに渡される引数の辞書です。デフォルトは `{}` です。

サブクラスがコンストラクタをオーバーライドしたときは、スレッドが何かを始める前に、基底クラスのコンストラクタ (`Thread.__init__()`) を確実に呼出さなくてははいけません。

**start()**

スレッドの活動を始めます。

これはスレッドオブジェクトにつき一度だけ呼出すようにします。オブジェクトの `run()` メソッドが個別のスレッドの中で呼出されるよう準備します。

**run()**

スレッドの活動を表すメソッドです。

あなたは、このメソッドをサブクラスでオーバーライドします。標準の `run()` メソッドは、*target* としてオブジェクトのコンストラクタに渡された、呼出し可能オブジェクトを呼出します。もしあるのなら、*args* と *kwargs* 引数から得られたシーケンシャルな、あるいは辞書の引数を使います。



`join([timeout])`

スレッドが終了するまで待ちます。これは `join()` メソッドが呼ばれたスレッドが終了するまで、呼出しスレッドをブロックします。もしくは、`unhandled` 例外が送出される、あるいはオプションのタイムアウトが発生するまでブロックします。

`timeout` 引数が `None` 以外で与えられたとき、それはタイムアウトを秒 (またはその分数) で指定する浮動小数です。

スレッドは何度でも `join()` されることができます。

スレッドは自身に `join` することはできません。デッドロックを引き起こします。

スレッドがスタートする前に `join()` を試みることは間違っています。

`getName()`

スレッド名を返します。

`setName(name)`

スレッド名をセットします。

名前は識別のためだけに使われる文字列です。セマンティクスではありません。複数のスレッドに同じ名前を与えることができます。初期名はコンストラクタによってセットされます。

`isAlive()`

スレッドが活着ているかどうか返します。

多くの場合、スレッドは `start()` メソッドを呼出した瞬間から `run()` メソッドが終了するまで生きています。

`isDaemon()`

スレッドのデーモンフラグを返します。

`setDaemon(daemonic)`

スレッドのデーモンフラグを真偽値 `daemonic` でセットします。これは `start()` を呼出す前に使わなくてははいけません。

初期値は作成するスレッドから継承されます。

デーモンでないアクティブなスレッドが残ってないとき、Python プログラム全体が終了します。

### 7.5.7 Timer オブジェクト

このクラスは、タイマーで一定の時間が経過した後に実行すべき動作を表します。Timer は Thread のサブクラスであり、カスタムスレッドの例としての機能を持っています。

`start()` メソッドを呼出すことで、タイマーはスレッドとしてスタートします。`cancel()` メソッドを呼出すことで、(その動作が始まる前に) タイマーはストップすることが出来ます。タイマーは、ユーザーによって指定された間隔と正確に同じではないかもしれませんが、その動作を実行する前に待ちます。

例:

```
def hello():
    print "hello, world"

t = Timer(30.0, hello)
t.start() # after 30 seconds, "hello, world" will be printed
```

`class Timer(interval, function, args=[], kwargs={})`

`interval` 秒が過ぎた後に、引数 `args` とキーワード引数 `kwargs` によって `function` を実行するタイマーを作成します。

`cancel()`

タイマーをストップして、その動作の実行をキャンセルします。タイマーがまだその待ち状態にある場合は、単にこのように働きます。

## 7.6 `dummy_thread` — `thread` の代替モジュール

このモジュールは `thread` モジュールのインターフェースをそっくりまねるものです。 `thread` モジュールがサポートされていないプラットフォームで `import` することを意図して作られたものです。

使用例:

```
try:
    import thread as _thread
except ImportError:
    import dummy_thread as _thread
```

生成するスレッドが、他のブロックしたスレッドを待ち、デッドロック発生の可能性がある場合には、このモジュールを使わないようにしてください。ブロッキング I/O を使っている場合によく起きます。

## 7.7 `dummy_threading` — `threading` の代替モジュール

このモジュールは `threading` モジュールのインターフェースをそっくりまねるものです。 `threading` モジュールがサポートされていないプラットフォームで `import` することを意図して作られたものです。

使用例:

```
try:
    import threading as _threading
except ImportError:
    import dummy_threading as _threading
```

生成するスレッドが、他のブロックしたスレッドを待ち、デッドロック発生の可能性がある場合には、このモジュールを使わないようにしてください。ブロッキング I/O を使っている場合によく起きます。

## 7.8 `Queue` — 同期キュークラス

`Queue` モジュールは、多生産者-多消費者 FIFO キューを実装します。これは、複数のスレッドの間で情報を安全に交換しなければならないときのスレッドプログラミングで特に有益です。このモジュールの `Queue` クラスは、必要なすべてのロックセマンティクスを実装しています。これは Python のスレッドサポートの状況に依存します。

参考資料:

`bisect` モジュール (5.9 節):

`Queue` クラスを使った優先順位付きキューの例

`Queue` モジュールは以下のクラスと例外を定義します:

`class Queue(maxsize)`

クラスのコンストラクタです。 `maxsize` はキューに置くことのできる要素数の上限を設定する整数です。いったんこの大きさに達したら、挿入はキューの要素が消費されるまでブロックされます。もし `maxsize` が 0 以下であるならば、キューの大きさは無限です。

### **exception Empty**

空またはロックされている `Queue` オブジェクトで、非ブロックメソッドとして `get()`(または `get_nowait()`) が呼ばれたとき、送出される例外です。

### **exception Full**

満杯またはロックされている `Queue` オブジェクトで、非ブロックメソッドとして `put()`(または `put_nowait()`) が呼ばれたとき、送出される例外です。

## 7.8.1 キューオブジェクト

クラス `Queue` はキューオブジェクトを実装しており、以下のメソッドを持っています。このクラスは、他のキュー構造 (例えばスタック) を実装するために派生させられるますが、継承可能なインタフェースはここでは説明しません。詳しいことはソースコードを見てください。公開メソッドは次のものです:

#### **qsize()**

キューの大まかなサイズを返します。マルチスレッドセマンティクスにおいて、この値は信頼できません。

#### **empty()**

キューが空なら `True` を返し、そうでないなら `False` を返します。マルチスレッドセマンティクスにおいて、この値は信頼できません。

#### **full()**

キューが満杯なら `True` を返し、そうでないなら `False` を返します。マルチスレッドセマンティクスにおいて、この値は信頼できません。

#### **put(item[, block[, timeout]])**

`item` をキューに入れます。もしオプション引数 `block` が `True` で `timeout` が `None`(デフォルト) ならば、フリースロットが利用可能になるまでブロックします。`timeout` が正の値の場合、最大で `timeout` 秒間ブロックし、その時間内に空きスロットが利用可能にならなければ、例外 `Full` を送出します。他方 (`block` が `False`)、直ちにフリースロットが利用できるならば、キューにアイテムを置きます。できないならば、例外 `Full` を送出します (この場合 `timeout` は無視されます)。

2.3 で追加された仕様: the timeout parameter

#### **put\_nowait(item)**

`put(item, False)` と同じ意味です。

#### **get([block[, timeout]])**

キューからアイテムを取り除き、それを返します。もしオプション引数 `block` が `True` で `timeout` が `None`(デフォルト) ならば、アイテムが利用可能になるまでブロックします。もし `timeout` が正の値の場合、最大で `timeout` 秒間ブロックし、その時間内でアイテムが利用可能にならなければ、例外 `Empty` を送出します。他方 (`block` が `False`)、直ちにアイテムが利用できるならば、それを返します。できないならば、例外 `Empty` を送出します (この場合 `timeout` は無視されます)。

2.3 で追加された仕様: the timeout parameter

#### **get\_nowait()**

`get(False)` と同じ意味です。

## 7.9 mmap — メモリマップファイル

メモリにマップされたファイルオブジェクトは、文字列とファイルオブジェクトの両方のように振舞います。しかし通常の文字列オブジェクトとは異なり、これらは可変です。文字列が期待されるほとんどの場

所で mmap オブジェクトを利用できます。例えば、メモリマップファイルを探索するために re モジュールを使うことができます。それらは可変なので、`obj[index] = 'a'` のように文字を変換できますし、スライスを使うことで `obj[i1:i2] = '...'` のように部分文字列を変換することができます。現在のファイル位置をデータの始めとする読み込みや書き込み、ファイルの異なる位置へ `seek()` することもできます。

メモリマップファイルは UNIX 上と Windows 上とは異なる `mmap()` 関数によって作られます。いずれの場合も、開いたファイルのディスクリプタを、更新のために提供しなければなりません。すでに存在する Python ファイルオブジェクトをマップしたい場合は、`fileno` パラメータのための現在値を手に入れるために、`fileno()` メソッドを使用して下さい。そうでなければ、ファイル・ディスクリプタを直接返す `os.open()` 関数(呼び出すときにはまだファイルが閉じている必要があります)を使って、ファイルを開くことができます。

関数の UNIX バージョンと Windows バージョンのために、オプションのキーワード・パラメータとして `access` を指定することになるかもしれません。`access` は 3 つの値の内の 1 つを受け入れます。`ACCESS_READ` は読み込み専用、`ACCESS_WRITE` は書き込み可能、`ACCESS_COPY` はコピーした上での書き込みです。`access` は UNIX と Windows の両方で使用することができます。`access` が指定されない場合、Windows の `mmap` は書き込み可能マップを返します。3 つのアクセス型すべてに対する初期メモリ値は、指定されたファイルから得られます。`ACCESS_READ` を割り当てたメモリマップは `TypeError` 例外を送出します。`ACCESS_WRITE` を割り当てたメモリマップはメモリと元のファイルの両方に影響を与えます。`ACCESS_COPY` を割り当てたメモリマップはメモリに影響を与えますが、元のファイルを更新することはありません。

`mmap(fileno, length[, tagname[, access]])`

(Windows) バージョンはファイルハンドル `fileno` によって指定されたファイルから `length` バイトをマップして、`mmap` オブジェクトを返します。`length` が 0 の場合、マップの最大の長さは `mmap()` が呼ばれたときのファイルサイズになるでしょう。

`tagname` は、`None` 以外で指定された場合、マップのタグ名を与える文字列となります。Windows は同じファイルに対する様々なマップを持つことを可能にします。既存のタグの名前を指定すればそのタグがオープンされ、そうでなければこの名前の新しいタグが作成されます。もしこのパラメータを省略したり `None` を与えたりしたならば、マップは名前なしで作成されます。タグ・パラメータの使用の回避は、あなたのコードを UNIX と Windows の間で移植可能にしておくのを助けてくれるでしょう。

`mmap(fileno, length[, flags[, prot[, access]]])`

(UNIX) バージョンは、ファイル・ディスクリプタ `fileno` によって指定されたファイルから `length` バイトをマップし、`mmap` オブジェクトを返します。

`flags` はマップの種類を指定します。`MAP_PRIVATE` はプライベートな copy-on-write(書き込み時コピー)のマップを作成します。従って、`mmap` オブジェクトの内容への変更はこのプロセス内にのみ有効です。`MAP_SHARED` はファイルの同じ領域をマップする他のすべてのプロセスと共有されたマップを作成します。デフォルトは `MAP_SHARED` です。

`prot` が指定された場合、希望のメモリ保護を与えます。2 つの最も有用な値は、`PROT_READ` と `PROT_WRITE` です。これは、読み込み可能または書き込み可能を指定するものです。`prot` のデフォルトは `PROT_READ | PROT_WRITE` です。

`access` はオプションのキーワード・パラメータとして、`flags` と `prot` の代わりに指定してもかまいません。`flags, prot` と `access` の両方を指定することは間違っています。このパラメーターを使用法についての情報は、`access` の記述を参照してください。

メモリマップファイルオブジェクトは以下のメソッドをサポートしています:

`close()`

ファイルを閉じます。この呼出しの後にオブジェクトの他のメソッドの呼出すことは、例外の送出手引き起こすでしょう。

**find**(*string*[, *start*])

オブジェクト内で部分文字列 *string* が見つかった場所の最も小さいインデックスを返します。失敗したとき -1 を返します。 *start* は探索を始めた場所のインデックスで、デフォルトは 0 です。

**flush**([*offset*, *size*])

ファイルのメモリコピー内での変更をディスクへフラッシュします。この呼出しを使わなかった場合、オブジェクトが破壊される前に変更が書き込まれる保証はありません。もし *offset* と *size* が指定された場合、与えられたバイトの範囲の変更だけがディスクにフラッシュされます。指定されない場合、マップ全体がフラッシュされます。

**move**(*dest*, *src*, *count*)

オフセット *src* からインデックス *dest* へ *count* バイトだけコピーします。もし mmap が ACCESS\_READ で作成されていた場合、TypeError 例外を送出します。

**read**(*num*)

現在のファイル位置から *num* バイトの文字列を返します。ファイル位置は返したバイトの分だけ後ろの位置へ更新されます。

**read\_byte**()

現在のファイル位置から長さ 1 の文字列を返します。ファイル位置は 1 だけ進みます。

**readline**()

現在のファイル位置から次の新しい行までの、1 行を返します。

**resize**(*newsize*)

もし mmap が ACCESS\_READ または ACCESS\_COPY で作成されたならば、マップのリサイズは TypeError 例外を送出します。

**seek**(*pos*[, *whence*])

ファイルの現在位置をセットします。 *whence* 引数はオプションであり、デフォルトは 0(絶対位置) です。その他の値として、1(現在位置からの相対位置) と 2(ファイルの終わりからの相対位置) があります。

**size**()

ファイルの長さを返します。メモリマップ領域のサイズより大きいかもしれません。

**tell**()

ファイル・ポインタの現在位置を返します。

**write**(*string*)

メモリ内のファイル・ポインタの現在位置から *string* のバイト列を書き込みます。ファイル位置はバイト列が書き込まれた後の位置へ更新されます。もし mmap が ACCESS\_READ で作成されていた場合、書き込み時に TypeError 例外が送出されるでしょう。

**write\_byte**(*byte*)

メモリ内のファイル・ポインタの現在位置から単一文字の文字列 *byte* を書き込みます。ファイル位置は 1 だけ進みます。もし mmap が ACCESS\_READ で作成されていた場合、書き込み時に TypeError 例外が送出されるでしょう。

## 7.10 anydbm — DBM 形式のデータベースへの汎用アクセスインタフェース

anydbm は種々の DBM データベース — (bsddb を使う) dbhash、gdbm、および dbm — への汎用インタフェースです。これらのモジュールがどれもインストールされていない場合、dumbdbm モジュールの低速で単純な DBM 実装が使われます。

`open(filename[, flag[, mode]])`

データベースファイル *filename* を開き、対応するオブジェクトを返します。

データベースファイルがすでに存在する場合、`whichdb` モジュールを使ってファイルタイプが判定され、適切なモジュールが使われます; 既存のデータベースファイルが存在しなかった場合、上に挙げたモジュール中で最初にインポートすることができたものが使われます。

オプションの *flag* は既存のデータベースを読み込み専用で開く `'r'`、既存のデータベースを読み書き用に開く `'w'`、既存のデータベースが存在しない場合には新たに作成する `'c'`、および常に新たにデータベースを作成する `'n'` をとることができます。この引数が指定されない場合、標準の値は `'r'` になります。

オプションの *mode* 引数は、新たにデータベースを作成しなければならない場合に使われる UNIX のファイルモードです。標準の値は 8 進数の `0666` です (この値は現在有効な `umask` で修飾されます)。

#### exception error

サポートされているモジュールのどれかによって送出されうる例外が収められるタプルで、先頭の要素は `anydbm.error` になっています — `anydbm.error` が送出された場合、後者が使われます。

`open()` によって返されたオブジェクトは辞書とほとんど同じ同じ機能をサポートします; キーとそれに対応付けられた値を記憶し、引き出し、削除することができ、`has_key()` および `keys()` メソッドを使うことができます。キーおよび値は常に文字列です。

参考資料:

`dbhash` モジュール (7.11 節):

BSD db データベースインタフェース。

`dbm` モジュール (8.6 節):

標準の UNIX データベースインタフェース。

`dumbdbm` モジュール (7.14 節):

`dbm` インタフェースの移植性のある実装。

`gdbm` モジュール (8.7 節):

`dbm` インタフェースに基づいた GNU データベースインタフェース。

`shelve` モジュール (3.17 節):

Python `dbm` インタフェース上に構築された汎用オブジェクト永続化機構。

`whichdb` モジュール (7.12 節):

既存のデータベースがどの形式のデータベースか判定するユーティリティモジュール。

## 7.11 dbhash — BSD データベースライブラリへの DBM 形式のインタフェース

`dbhash` モジュールでは BSD db ライブラリを使ってデータベースを開くための関数を提供します。このモジュールは、DBM 形式のデータベースへのアクセスを提供する他の Python データベースモジュールのインタフェースをそのまま反映しています。`dbhash` を使うには `bsddb` モジュールが必要です。

このモジュールでは一つの例外と一つの関数を提供しています:

#### exception error

`KeyError` 以外のデータベースのエラーで送出されます。`bsddb.error` と同じ意味です。

`open(path[, flag[, mode]])`

データベース *db* を開き、データベースオブジェクトを返します。引数 *path* はデータベースファイルの名前です。



引数 *flag* は 'r' (標準の値)、'w'、'c' (データベースが存在しない場合には作成する)、あるいは 'n' (常に新たな空のデータベースを作成する)をとることができます。BSD db ライブラリがファイルロックをサポートするようなプラットフォームでは、ロックを使うよう示すために 'l' を追加することができます。

オプションの *mode* 引数は、新たにデータベースを作成しなければならないときにデータベースファイルに設定すべき UNIX ファイル権限ビットを表すために使われます; この値はプロセスの現在の *umask* 値でマスクされます。

参考資料:

anydbm モジュール (7.10 節):

dbm 形式のデータベースへの汎用インタフェース。

bsddb モジュール (7.13 節):

BSD db ライブラリへの低レベルインタフェース。

whichdb モジュール (7.12 節):

既存のデータベースがどの形式のデータベースか判定するユーティリティモジュール。

### 7.11.1 データベースオブジェクト

`open()` によって返されるデータベースオブジェクトは、全ての DBM 形式データベースやマップ型オブジェクトで共通のメソッドを提供します。それら標準のメソッドに加え、`dbhash` では以下のメソッドが利用可能です。

**first()**

このメソッドと `next()` メソッドを使って、データベースの全てのキー/値のペアにわたってループ処理を行えます。探索はデータベースの内部ハッシュ値の順番に行われ、キーの値に順に並んでいるとは限りません。このメソッドは最初のキーを返します。

**last()**

データベース探索における最後のキー/値を返します。逆順探索を開始する際に使うことができます; `previous()` を参照してください。

**next()**

データベースの順方向探索において、次のよりも後に来るキー/値のペアを返します。以下のコードはデータベース `db` について、キー全てを含むリストをメモリ上に生成することなく全てのキーを出力します。

```
print db.first()
for i in xrange(1, len(db)):
    print db.next()
```

**previous()**

データベースの逆方向探索において、手前に来るキー/値のペアを返します。`last()` と併せて、逆方向の探索に用いられます。

**sync()**

このメソッドはディスクにまだ書き込まれていないデータを全て書き込ませます。

## 7.12 whichdb — どの DBM モジュールがデータベースを作ったかを推測する

このモジュールに含まれる唯一の関数はあることを推測します。つまり、与えられたファイルを開くためには、利用可能なデータベースモジュール (dbm、gdbm、dbhash) のどれを用いるべきかということです。

`whichdb(filename)`

ファイルが読めないか存在しないために開くことが出来ない場合は `None`、ファイルの形式を推測できない場合は空の文字列 (`''`)、推測できる場合は必要なモジュール名 (`'dbm'`、`'gdbm'` など) を含む文字列を返します。

## 7.13 bsddb — Berkeley DB ライブラリへのインタフェース

bsddb モジュールは Berkeley DB ライブラリへのインタフェースを提供します。ユーザは適当な `open` 呼び出しを使うことで、ハッシュ、B-Tree、またはレコードに基づくデータベースファイルを生成することができます。bsddb オブジェクトは辞書と大体同じように振る舞います。しかし、キー及び値は文字列でなければならないので、他のオブジェクトをキーとして使ったり、他の種のオブジェクトを記録したい場合、それらのデータを何らかの方法で直列化しなければなりません。これには通常 `marshal.dumps` や `pickle.dumps` が使われます。

Python 2.3 以降の bsddb モジュールは、バージョン 3.1 以降の Berkeley DB ライブラリのみをサポートしています。(現時点では、3.1 から 4.1 までのバージョンでの動作を確認しています。)

参考資料:

<http://pybsddb.sourceforge.net/>

新しい Berkeley DB インターフェースのドキュメントがあります。新しいインターフェースは、Berkeley DB 3 と 4 で `sleepycat` が提供しているオブジェクト指向インターフェースとほぼ同じインターフェースとなっています。

<http://www.sleepycat.com/>

Sleepycat Software は、最新の Berkeley DB ライブラリを開発しています。

以下では、従来の bsddb モジュールと互換性のある、古いインターフェースを解説しています。現在の、Db と DbEnv によるオブジェクト指向的インターフェースについては上記 pybsddb の URL を参照してください。

警告: このインターフェースは旧式で、Python 2.3.x や以前のバージョンではスレッド安全ではありません。マルチスレッドからアクセスを行おうとすると、データの破壊やコアダンプ、デッドロックを引き起こすことがあります。マルチスレッドやマルチプロセスでデータベースにアクセスする必要がある場合には、上のリンク先にある、より新しい pybsddb インタフェースを使わねばなりません。

bsddb モジュールでは、適切な形式の Berkeley DB ファイルにアクセスするオブジェクトを生成する以下の関数を定義しています。各関数の最初の二つの引数は同じです。可搬性のために、ほとんどのインスタンスでは最初の二つの引数だけが使われているはずです。

`hashopen(filename[, flag[, mode[, bsize[, ffactor[, nelem[, cachesize[, hash[, lorder]]]]]]])`

`filename` と名づけられたハッシュ形式のファイルを開きます。`filename` に `None` を指定することで、ディスクに保存するつもりがないファイルを生成することもできます。オプションの `flag` には、ファイルを開くためのモードを指定します。このモードは `'r'` (読み出し専用、デフォルト)、`'w'` (読み書き)、`'c'` (読み書き - 必要ならファイルを生成)、または `'n'` (読み書き - ファイル長を 0 に切り詰め) にすることができます。他の引数はほとんど使われることはなく、下位レベルの `dbopen()` 関数に渡されるだけです。他の引数の使い方およびその解釈については Berkeley DB のドキュメントを読ん

で下さい。

**btopen**(*filename*[, *flag*[, *mode*[, *btflags*[, *cache\_size*[, *maxkeypage*[, *minkeypage*[, *psize*[, *lorder*]]]]]]]])  
*filename* と名づけられた B-Tree 形式のファイルを開きます。*filename* に `None` を指定することで、ディスクに保存するつもりがないファイルを生成することもできます。オプションの *flag* には、ファイルを開くためのモードを指定します。このモードは 'r' (読み出し専用、デフォルト)、'w' (読み書き)、'c' (読み書き - 必要ならファイルを生成)、または 'n' (読み書き - ファイル長を 0 に切り詰め)、にすることができます。他の引数はほとんど使われることはなく、下位レベルの `dbopen()` 関数に渡されるだけです。他の引数の使い方およびその解釈については Berkeley DB のドキュメントを読んで下さい。

**rnopen**(*filename*[, *flag*[, *mode*[, *rnflags*[, *cache\_size*[, *psize*[, *lorder*[, *reclen*[, *bval*[, *bfname*]]]]]]]])  
*filename* と名づけられた DB レコード形式のファイルを開きます。*filename* に `None` を指定することで、ディスクに保存するつもりがないファイルを生成することもできます。オプションの *flag* には、ファイルを開くためのモードを指定します。このモードは 'r' (読み出し専用、デフォルト)、'w' (読み書き)、'c' (読み書き - 必要ならファイルを生成)、または 'n' (読み書き - ファイル長を 0 に切り詰め)、にすることができます。他の引数はほとんど使われることはなく、下位レベルの `dbopen()` 関数に渡されるだけです。他の引数の使い方およびその解釈については Berkeley DB のドキュメントを読んで下さい。

参考資料:

dbhash モジュール (7.11 節):

bsddb への DBM 形式のインタフェース

注意: 2.3 以降の Unix 版 Python には、`bsddb185` モジュールが存在する場合があります。このモジュールは古い Berkeley DB 1.85 データベースライブラリを持つシステムをサポートするためだけに存在しています。新規に開発するコードでは、`bsddb185` を直接使用しないで下さい。

### 7.13.1 ハッシュ、BTree、およびレコードオブジェクト

インスタンス化したハッシュ、B-Tree、およびレコードオブジェクトは辞書型と同じメソッドをサポートするようになります。加えて、以下に列挙したメソッドもサポートします。2.3.1 で変更された仕様: マップ型メソッドを追加しました

**close()**

データベースの背後にあるファイルを閉じます。オブジェクトはアクセスできなくなります。これらのオブジェクトには `open` メソッドがないため、再度ファイルを開くためには、新たな `bsddb` モジュールを開く関数を呼び出さなくてはなりません。

**keys()**

DB ファイルに収められているキーからなるリストを返します。リスト内のキーの順番は決まっておらず、あてにはなりません。特に、異なるファイル形式の DB 間では返されるリストの順番が異なります。

**has\_key(key)**

引数 *key* が DB ファイルにキーとして含まれている場合 `1` を返します。

**set\_location(key)**

カーソルを *key* で示される要素に移動し、キー及び値からなるタプルを返します。(bopen を使って開かれる) B-Tree データベースでは、*key* が実際にはデータベース内に存在しなかった場合、カーソルは並び順が *key* の次に来るような要素を指し、その場所のキー及び値が返されます。他のデータベースでは、データベース中に *key* が見つからなかった場合 `KeyError` が送出されます。

**first()**

カーソルを DB ファイルの最初の要素に設定し、その要素を返します。B-Tree データベースの場合を除き、ファイル中のキーの順番は決まっています。

`next()`

カーソルを DB ファイルの次の要素に設定し、その要素を返します。B-Tree データベースの場合を除き、ファイル中のキーの順番は決まっています。

`previous()`

カーソルを DB ファイルの直前の要素に設定し、その要素を返します。B-Tree データベースの場合を除き、ファイル中のキーの順番は決まっています。(hashopen() で開かれるような) ハッシュ表データベースではサポートされていません。

`last()`

カーソルを DB ファイルの最後の要素に設定し、その要素を返します。ファイル中のキーの順番は決まっています。(hashopen() で開かれるような) ハッシュ表データベースではサポートされていません。

`sync()`

ディスク上のファイルをデータベースに同期させます。

以下はプログラム例です:

```
>>> import bsddb
>>> db = bsddb.btopen('/tmp/spam.db', 'c')
>>> for i in range(10): db['%d'%i] = '%d' % (i*i)
...
>>> db['3']
'9'
>>> db.keys()
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
>>> db.first()
('0', '0')
>>> db.next()
('1', '1')
>>> db.last()
('9', '81')
>>> db.set_location('2')
('2', '4')
>>> db.previous()
('1', '1')
>>> for k, v in db.iteritems():
...     print k, v
0 0
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
>>> '8' in db
True
>>> db.sync()
0
```

## 7.14 dumbdbm — 可搬性のある DBM 実装

注意: dumbdbm モジュールは、anydbm が安定なモジュールを他に見つけることができなかった際の最後の手段とされています。dumbdbm モジュールは速度を重視して書かれているわけではなく、他のデータベースモジュールのように重い使い方をするためのものではありません。

dumbdbm モジュールは永続性辞書に類似したインタフェースを提供し、全て Python で書かれています。gdbm や bsddb といったモジュールと異なり、外部ライブラリは必要ありません。他の永続性マップ型のように、キーおよび値は常に文字列でなければなりません。

このモジュールでは以下の内容を定義してします:

### exception error

I/O エラーのような dumbdbm 特有のエラーの際に送出されます。不正なキーを指定したときのような、一般的な対応付けエラーの際には `KeyError` が送出されます。

`open(filename[, flag[, mode]])`

dumbdbm データベースを開き、dumbdbm オブジェクトを返します。*filename* 引数はデータベースファイル名の雛型 (特定の拡張子をもたないもの) です。dumbdbm データベースが生成される際、`‘.dat’` および `‘.dir’` の拡張子を持ったファイルが生成されます。

オプションの *flag* 引数は現状では無視されます; データベースは常に更新のために開かれ、存在しない場合には新たに作成されます。

オプションの *mode* 引数は UNIX におけるファイルのモードで、データベースを作成する際に使われます。デフォルトでは 8 進コードの 0666 になっています (umask によって修正を受けます)。2.2 で変更された仕様: *mode* 引数は以前のバージョンでは無視されます

参考資料:

anydbm モジュール (7.10 節):

dbm 形式のデータベースに対する汎用インタフェース。

dbm モジュール (8.6 節):

DBM/NDBM ライブラリに対する同様のインタフェース。

gdbm モジュール (8.7 節):

GNU GDBM ライブラリに対する同様のインタフェース。

shelve モジュール (3.17 節):

非文字列データを記録する永続化モジュール。

whichdb モジュール (7.12 節):

既存のデータベースの形式を判定するために使われるユーティリティモジュール。

### 7.14.1 Dumbdbm オブジェクト

UserDict.DictMixin クラスで提供されているメソッドに加え、dumbdbm オブジェクトでは以下のメソッドを提供しています。

`sync()`

ディスク上の辞書とデータファイルを同期します。このメソッドは Shelve オブジェクトの `sync` メソッドから呼び出されます。

## 7.15 zlib — gzip 互換の圧縮

このモジュールでは、データ圧縮を必要とするアプリケーションが zlib ライブラリを使って圧縮および解凍を行えるようにします。zlib ライブラリ自体の Web ホームページは <http://www.gzip.org/zlib/> にある。2000 年 9 月の時点での最新バージョンは 1.1.3 です。可能ならこれ以降のバージョンを使うのがよいでしょう。以前のバージョンの zlib ライブラリにはこの Python モジュールと互換性のない部分があることが知られています。

このモジュールで利用可能な例外と関数を以下に示します:

### exception error

圧縮および解凍時のエラーによって送出される例外。

### adler32(*string*[, *value*])

*string* の Adler-32 チェックサムを計算します。(Adler-32 チェックサムは、おおむね CRC32 と同等の信頼性を持ちながらはるかに高速に計算することができます。) *value* が与えられていれば、*value* はチェックサム計算の初期値として使われます。それ以外の場合には固定のデフォルト値が使われます。この機能によって、複数の入力文字列を結合したデータ全体にわたり、通しのチェックサムを計算することができます。このアルゴリズムは暗号法論的には強力とはいえないので、認証やデジタル署名などに用いるべきではありません。このアルゴリズムはチェックサムアルゴリズムとして用いるために設計されたものなので、汎用的なハッシュアルゴリズムには向きません。

### compress(*string*[, *level*])

*string* で与えられた文字列を圧縮し、圧縮されたデータを含む文字列を返します。*level* は 1 から 9 までの整数をとる値で、圧縮のレベルを制御します。1 は最も高速で最小限の圧縮を行います。9 はもっとも低速になりますが最大限の圧縮を行います。デフォルトの値は 6 です。圧縮時に何らかのエラーが発生した場合、`error` 例外を送出します。

### compressobj([*level*])

一度にメモリ上に置くことができないようなデータストリームを圧縮するための圧縮オブジェクトを返します。*level* は 1 から 9 までの整数で、圧縮レベルを制御します。1 はもっとも高速で最小限の圧縮を、9 はもっとも低速になりますが最大限の圧縮を行います。デフォルトの値は 6 です。

### crc32(*string*[, *value*])

*string* の CRC (Cyclic Redundancy Check, 巡回符号方式) チェックサムを計算します。*value* が与えられていれば、チェックサム計算の初期値として使われます。与えられていなければデフォルトの初期値が使われます。*value* を与えることで、複数の入力文字列を結合したデータ全体にわたり、通しのチェックサムを計算することができます。このアルゴリズムは暗号法論的には強力ではなく、認証やデジタル署名に用いるべきではありません。アルゴリズムはチェックサムアルゴリズムとして設計されているので、汎用のハッシュアルゴリズムには向きません。

### decompress(*string*[, *wbits*[, *bufsize*]])

*string* 内のデータを解凍して、解凍されたデータを含む文字列を返します。*wbits* パラメタはウィンドウバッファの大きさを制御します。*bufsize* が与えられていれば、出力バッファの書記サイズとして使われます。解凍処理に何らかのエラーが生じた場合、`error` 例外を送出します。

*wbits* の絶対値は、データを圧縮する際に用いられるヒストリバッファのサイズ (ウィンドウサイズ) に対し、2 を底とする対数をとったものです。最近のほとんどのバージョンの zlib ライブラリを使っているなら、*wbits* の絶対値は 8 から 15 とするべきです。より大きな値はより良好な圧縮につながりますが、より多くのメモリを必要とします。デフォルトの値は 15 です。*wbits* の値が負の場合、標準的な **gzip** ヘッダを出力しません。これは zlib ライブラリの非公開仕様であり、**unzip** の圧縮ファイル形式に対する互換性のためのものです。

*bufsize* は解凍されたデータを保持するためのバッファサイズの初期値です。バッファの空きは必要に



応じて必要なだけ増加するので、なれば、必ずしも正確な値を指定する必要はありません。この値のチューニングでできることは、`malloc()` が呼ばれる回数を数回減らすことぐらいです。デフォルトのサイズは 16384 です。

**decompressobj**(*[wbits]*)

メモリ上に一度に展開できないようなデータストリームを解凍するために用いられる解凍オブジェクトを返します。*wbits* パラメタはウィンドウバッファのサイズを制御します。

圧縮オブジェクトは以下のメソッドをサポートします:

**compress**(*string*)

*string* を圧縮し、圧縮されたデータを含む文字列を返します。この文字列は少なくとも *string* に相当します。このデータは以前に呼んだ `compress()` が返した出力と結合することができます。入力の一部は以後の処理のために内部バッファに保存されることもあります。

**flush**(*[mode]*)

未処理の入力データが処理され、この未処理部分を圧縮したデータを含む文字列が返されます。*mode* は定数 `Z_SYNC_FLUSH`、`Z_FULL_FLUSH`、または `Z_FINISH` のいずれかをとり、デフォルト値は `Z_FINISH` です。`Z_SYNC_FLUSH` および `Z_FULL_FLUSH` ではこれ以後にもデータ文字列を圧縮できるモードで、解凍時の部分的なエラーリカバリを可能にします。一方、`Z_FINISH` は圧縮ストリームを閉じ、これ以後のデータの圧縮を禁止します。*mode* に `Z_FINISH` を設定して `flush()` メソッドを呼び出した後は、`compress()` メソッドを再び呼ぶべきではありません。唯一の現実的な操作はこのオブジェクトを削除することだけです。

解凍オブジェクトは以下のメソッドと 2 つの属性をサポートします:

**unused\_data**

圧縮データの末尾までのバイト列が入った文字列です。すなわち、この値は圧縮データの入っているバイト列の最後の文字までが読み出せるかぎり "" となります。入力文字列全てが圧縮データを含んでいた場合、この属性は ""、すなわち空文字列になります。

圧縮データ文字列がどこで終了しているかを決定する唯一の方法は、実際にそれを解凍することです。つまり、大きなファイルの一部に圧縮データが含まれているときに、その末端を調べるためには、データをファイルから読み出し、空でない文字列を後ろに続けて、`unused_data` が空文字列でなくなるまで、解凍オブジェクトの `decompress` メソッドに入力しつづけるしかありません。

**unconsumed\_tail**

解凍されたデータを収めるバッファの長さ制限を超えたために、最も最近の `decompress` 呼び出しで処理しきれなかったデータを含む文字列です。このデータはまだ `zlib` 側からは見えていないので、正しい解凍出力を得るには以降の `decompress` メソッド呼び出しに (場合によっては後続のデータが追加された) データを差し戻さなければなりません。

**decompress**(*string*)

*[max\_length]* *string* を解凍し、少なくとも *string* の一部分に対応する解凍されたデータを含む文字列を返します。このデータは以前に `decompress()` メソッドを呼んだ時に返された出力と結合することができます。入力データの一部分が以後の処理のために内部バッファに保存されることもあります。

オプションパラメタ *max\_length* が与えられると、返される解凍データの長さが *max\_length* 以下に制限されます。このことは入力した圧縮データの全てが処理されとは限らないことを意味し、処理されなかったデータは `unconsumed_tail` 属性に保存されます。解凍処理を継続したいならば、この保存されたデータを以降の `decompress()` 呼び出しに渡さなくてはなりません。*max\_length* が与えられなかった場合、全ての入力が解凍され、`unconsumed_tail` 属性は空文字列になります。

**flush**()

未処理の入力データを全て処理し、最終的に圧縮されなかった残りの出力文字列を返します。`flush()` を呼んだ後、`decompress()` を再度呼ぶべきではありません。このときできる唯一現実的な操作は

オブジェクトの削除だけです。

参考資料:

gzip モジュール (7.16 節):

Reading and writing **gzip**-format files.

<http://www.gzip.org/zlib/>

The zlib library home page.

## 7.16 gzip — gzip ファイルのサポート

zlib モジュールで提供されているデータ圧縮は、GNU の圧縮プログラム **gzip** のものと互換性があります。そこで、gzip モジュールでは、**gzip** 形式のファイルを読み書きするための `GzipFile` クラスを提供します。このクラスのオブジェクトは自動的にデータを圧縮または解凍するので、通常のファイルオブジェクトのように見えます。**gzip** や **gunzip** プログラムで解凍できる、**compress** や **pack** による他の形式の圧縮ファイルはこのモジュールではサポートされていないので注意してください。

このモジュールでは以下の項目を定義しています:

`class GzipFile([filename[, mode[, compresslevel[, fileobj]]]])`

`GzipFile` クラスのコンストラクタです。`GzipFile` オブジェクトは `readinto()` と `truncate()` メソッドを除くほとんどのファイルオブジェクトのメソッドをシミュレートします。少なくとも `fileobj` および `filename` は有効な値でなければなりません。

クラスの新しいインスタンスは、`fileobj` に基づいて作成されます。`fileobj` は通常のファイル、`StringIO` オブジェクト、そしてその他ファイルをシミュレートできるオブジェクトでかまいません。値はデフォルトでは `None` で、ファイルオブジェクトを生成するために `filename` を開きます。

**gzip** ファイルヘッダ中には、ファイルが解凍されたときの元のファイル名を収めることができますが、`fileobj` が `None` でない場合、引数 `filename` がファイル名として認識できる文字列であれば、`filename` はファイルヘッダに収めるためだけに使われます。そうでない場合（この値はデフォルトでは空文字列です）、元のファイル名はヘッダに収められません。

`mode` 引数は、ファイルを読み出すのか、書き込むのかによって、`'r'`、`'rb'`、`'a'`、`'ab'`、`'w'`、そして `'wb'`、のいずれかになります。`fileobj` のファイルモードが認識可能な場合、`mode` はデフォルトで `fileobj` のモードと同じになります。そうでない場合、デフォルトのモードは `'rb'` です。`'b'` フラグがついていなくても、ファイルがバイナリモードで開かれることを保証するために `'b'` フラグが追加されます。これはプラットフォーム間での移植性のためです。

`compresslevel` 引数は 1 から 9 までの整数で、圧縮のレベルを制御します。1 は最も高速で最小限の圧縮しか行いません。9 は最も低速ですが、最大限の圧縮を行います。デフォルトの値は 9 です。

圧縮したデータの後ろにさらに何か追記したい場合もあるので、`GzipFile` オブジェクトの `close()` メソッド呼び出しは `fileobj` をクローズしません。この機能によって、書き込みのためにオープンした `StringIO` オブジェクトを `fileobj` として渡し、(`GzipFile` を `close()` した後に) `StringIO` オブジェクトの `getvalue()` メソッドを使って書き込んだデータの入っているメモリバッファを取得することができます。

`open(filename[, mode[, compresslevel]])`

`GzipFile(filename, mode, compresslevel)` の短縮形です。引数 `filename` は必須です。デフォルトで `mode` は `'rb'` に、`compresslevel` は 9 に設定されています。

参考資料:

zlib モジュール (7.15 節):

**gzip** ファイル形式のサポートを行うために必要な基本ライブラリモジュール。

## 7.17 bz2 — bzip2 互換の圧縮ライブラリ

2.3 で追加された仕様です。

このモジュールでは bz2 圧縮ライブラリのためのわかりやすいインタフェースを提供します。モジュールでは完全なファイルインタフェース、データを一括して圧縮（解凍）する関数、データを逐次的に圧縮（解凍）するためのクラス型を実装しています。

bz2 モジュールで提供されている機能を以下にまとめます：

- BZ2File クラスは、`readline()`、`readlines()`、`writelines()`、`seek()`、等を含む完全なファイルインタフェースを実装します。
- BZ2File クラスは `seek()` をエミュレーションでサポートします。
- BZ2File クラスは広範囲の改行文字バリエーションをサポートします。
- BZ2File クラスはファイルオブジェクトで言うところの、先読みアルゴリズムを用いた行単位の反復処理機能を提供します。
- 逐次的圧縮（解凍）が BZ2Compressor および BZ2Decompressor クラスでサポートされています。
- 一括圧縮（解凍）が関数 `compress()` および `decompress()` でサポートされています。
- 個別のロックメカニズムによってスレッド安全性を持っています。
- 埋め込みドキュメントが完備しています。

### 7.17.1 ファイルの圧縮（解凍）

圧縮ファイルの操作は BZ2File クラスで提供されています。

```
class BZ2File(filename[, mode[, buffering[, compresslevel]]])
```

bz2 ファイルを開きます。ファイルのモードは 'r' または 'w' で、それぞれ読み出しと書き込みに対応します。書き出し用に開いた場合、ファイルが存在しないなら新しくファイルが作成され、そうでない場合ファイルは切り詰められます。buffering パラメタが与えられた場合、0 はバッファリングを行わないことを表し、それよりも大きい値はバッファサイズの指定値となります。デフォルトでは 0 です。compresslevel が与えられている場合、この値は 1 から 9 までの整数値でなければなりません。デフォルトの値は 9 です。ファイルへの入力に広範囲の改行文字バリエーションをサポートさせたい場合は 'U' をファイルモードに追加します。入力ファイルの行末はどれも、Python からは '\n' として見えます。また、また、開かれているファイルオブジェクトは newlines 属性を持ち、None（まだ改行文字を読み込んでいない時）、'\r'、'\n'、'\r\n' または全ての改行文字バリエーションを含むタプルになります。広範囲の改行文字サポートは読み込みだけで利用可能です。生成されるインスタンスは通常のファイルインスタンスと同様の反復操作をサポートします。

```
close()
```

ファイルを閉じます。オブジェクトのデータ属性 `closed` を真にします。閉じられたファイルは以後の入出力操作に用いることができません。close() 自体はエラーを引き起こすことなく何度も呼び出すことができます。

```
read([size])
```

最大で size バイトの解凍されたデータを読み出し、文字列として返します。size 引数が負であるか省略された場合、EOF にたどり着くまで読み込みます。

**readline**(*[size]*)

ファイルから次の 1 行を読み出し、改行文字も含めて文字列を返します。負でない *size* 値は、返される文字列の最大バイト長を制限します (その場合不完全な行が返されることもあります) EOF の時には空文字列が返されます。

**readlines**(*[size]*)

ファイルから読み取った行文字列の集合からなるリストを返します。オプション引数 *size* が与えられていれば、返される行文字列全体のバイト数長さの大体の上限が指定されます。

**xreadlines**()

前のバージョンとの互換性のために用意されています。BZ2File オブジェクトはかつて **xreadlines** モジュールで提供されていたパフォーマンス最適化を含んでいます。リリース 2.3 以降で撤廃された仕様です。このメソッドは **file** オブジェクトの同名のメソッドとの互換性のために用意されていますが、現在は推奨されないメソッドです。代わりに `for line in file` を使ってください。

**seek**(*offset*, *whence*)

ファイル位置を移動します。引数 *offset* はバイト数での値です。オプション引数 *whence* はデフォルトで 0 (ファイルの先頭からのオフセットであり、*offset* >= 0 となるはず) に設定されています。他の値は 1 (現在のファイル位置からの相対位置であり、正負どちらの値もとります)、および 2 (ファイルの終末端からの相対位置であり、通常負の値になりますが、多くのプラットフォームでファイルの終末端を越えて **seek** を行うことができます)。

bz2 ファイルの **seek** はエミュレーションであり、パラメタの設定によっては処理が非常に低速になるかもしれないので注意してください。

**tell**()

現在のファイル位置を整数 (long 整数になるかもしれませんが) で返します。

**write**(*data*)

ファイルに文字列 *data* を書き込みます。バッファリングのため、ディスク上のファイルに書き込まれたデータを反映させるには **close**() が必要になるかもしれないので注意してください。

**writelines**(*sequence\_of\_strings*)

複数の文字列からなる配列をファイルに書き込みます。それぞれの文字列を書き込む際にさらに改行文字が追加されることはありません。配列は文字列を反復して取り出すことができる任意のオブジェクトでもかまいません。この操作はそれぞれの文字列を **write**() を呼んで書き込むのと同等の操作です。

### 7.17.2 逐次的な圧縮 (解凍)

逐次的な圧縮および解凍は BZ2Compressor および BZ2Decompressor クラスを用いて行われます。

**class BZ2Compressor**(*[compresslevel]*)

新しい圧縮オブジェクトを作成します。このオブジェクトはデータを逐次的に圧縮できます。一括してデータを圧縮したいのなら、**compress**() 関数を代わりに使ってください。*compresslevel* パラメタを与える場合、この値は 1 and 9 の間の整数でなければなりません。デフォルトの値は 9 です。

**compress**(*data*)

圧縮オブジェクトに追加のデータを入力します。データが圧縮されたチャンクを生成することができた場合にはチャンクを返します。圧縮データの入力を終えた後は圧縮処理を終えるために **flush**() を呼んでください。内部バッファに残っている未処理のデータが返されます。

**flush**()

圧縮処理を終え、内部バッファに残されているデータを返します。メソッドを呼んだ後は、この圧縮オブジェクトを使うべきではありません。

**class BZ2Decompressor()**

新しい解凍オブジェクトを生成します。このオブジェクトは逐次的にデータを解凍できます。一括してデータを解凍したいのなら、`decompress()` 関数を代りに使ってください。

**decompress(data)**

解凍オブジェクトに追加のデータを入力します。可能な限り、データが解凍されたチャンクを生成することができた場合にはチャンクを返します。ストリームの末端に到達した後に解凍処理を行おうとした場合には、例外 `EOFError` が送出されます。ストリームの終末端の後ろに何らかのデータがあった場合には、解凍操作からは無視され、このデータはオブジェクトの `unused_data` 属性に収められます。

### 7.17.3 一括圧縮 (解凍)

一括での圧縮および解凍を行うための関数、`compress()` および `decompress()` が提供されています。

**compress(data[, compresslevel])**

`data` を一括して圧縮します。データを逐次的に圧縮したいのなら、`BZ2Compressor` を代りに使ってください。もし `compresslevel` パラメタを与えるなら、この値は 1 から 9 をとらなくてはなりません。デフォルトの値は 9 です。

**decompress(data)**

`data` を一括して解凍します。データを逐次的に解凍したいのなら、`BZ2Decompressor` を代りに使ってください。

## 7.18 zipfile — ZIP アーカイブの処理

1.6 で追加された仕様です。

ZIP は一般によく知られているアーカイブ (書庫化) および圧縮の標準ファイルフォーマットです。このモジュールでは ZIP 形式のファイルの作成、読み書き、ファイルの作成、書庫内のファイルリスト読み出しを行うためのツールを提供します。より高度な使い方でこのモジュールを利用したいのなら、*PKZIP Application Note* に定義されている ZIP ファイルフォーマットを理解することが必要になるでしょう。

このモジュールは現在のところ、コメントを追記した ZIP ファイルやマルチディスク ZIP ファイルを扱うことはできません。

このモジュールで利用できる属性を以下に示します:

**exception error**

不備のある ZIP ファイル操作の際に送出されるエラー

**class ZipFile**

ZIP ファイルの読み書きのためのクラスです。コンストラクタの詳細については、“*ZipFile* オブジェクト” (7.18.1 節) を参照してください。

**class PyZipFile**

Python ライブラリを含む ZIP アーカイブを生成するためのクラスです。

**class ZipInfo([filename[, date\_time]])**

アーカイブ中のメンバに関する情報を提供するために用いられるクラスです。このクラスのインスタンスは `ZipFile` オブジェクトの `getinfo()` および `infolist()` メソッドによって返されます。`zipfile` モジュールを利用するほとんどのユーザはこのオブジェクトを自ら生成する必要はなく、モジュールが生成して返すオブジェクトを利用するだけで良いでしょう。`filename` はアーカイブメンバの完全な名前で、`date_time` はファイルの最終更新時刻を記述する、6 つのフィールドからなるタプルでな



くてはなりません。各フィールドについては 7.18.3, “ZipInfo オブジェクト” 節を参照してください。

`is_zipfile(filename)`

`filename` が正しいマジックナンバをもつ ZIP ファイルのときに `True` を返し、そうでない場合 `False` を返します。このモジュールは現在のところ、コメントを追記した ZIP ファイルを扱うことができません。

`ZIP_STORED`

アーカイブメンバが圧縮されていないことを表す数値定数です。

`ZIP_DEFLATED`

通常の ZIP 圧縮手法を表す数値定数。ZIP 圧縮は `zlib` モジュールを必要とします。現在のところ他の圧縮手法はサポートされていません。

参考資料:

*PKZIP Application Note*

(<http://www.pkware.com/appnote.html>)

ZIP ファイル形式およびアルゴリズムを作成した Phil Katz によるドキュメント。

*Info-ZIP Home Page*

(<http://www.info-zip.org/pub/infozip/>)

Info-ZIP プロジェクトによる ZIP アーカイブプログラム及びプログラム開発ライブラリに関する情報。

### 7.18.1 ZipFile オブジェクト

`class ZipFile(file[, mode[, compression]])`

ZIP ファイルを開きます。`file` はファイルへのパス名 (文字列) またはファイルのように振舞うオブジェクトのどちらでもかまいません。`mode` パラメタは、既存のファイルを読むためには `'r'`、既存のファイルを切り詰めたり新しいファイルに書き込むためには `'w'`、追記を行うためには `'a'` でなくてはなりません。`mode` が `'a'` で `file` が既存の ZIP ファイルを参照している場合、追加するファイルは既存のファイル中の ZIP アーカイブに追加されます。`file` が ZIP を参照していない場合、新しい ZIP アーカイブが生成され、既存のファイルの末尾に追加されます。このことは、ある ZIP ファイルを他のファイル、例えば `'python.exe'` に

```
cat myzip.zip >> python.exe
```

として追加することができ、少なくとも WinZip がこのようなファイルを読めることを意味します。`compression` はアーカイブを書き出すときの ZIP 圧縮法で、`ZIP_STORED` または `ZIP_DEFLATED` でなくてはなりません。不正な値を指定すると `RuntimeError` が送出されます。また、`ZIP_DEFLATED` 定数が指定されているのに `zlib` を利用することができない場合、`RuntimeError` が送出されます。デフォルト値は `ZIP_STORED` です。

`close()`

アーカイブファイルを閉じます。`close()` はプログラムを終了する前に必ず呼び出さなければなりません。さもないとアーカイブ上の重要なレコードが書き込まれません。

`getinfo(name)`

アーカイブメンバ `name` に関する情報を持つ `ZipInfo` オブジェクトを返します。

`infolist()`

アーカイブに含まれる各メンバの `ZipInfo` オブジェクトからなるリストを返します。既存のアーカイブファイルを開いている場合、リストの順番は実際の ZIP ファイル中のメンバの順番と同じになります。



`namelist()`

アーカイブメンバの名前のリストを返します。

`printdir()`

アーカイブの目次を `sys.stdout` に出力します。

`read(name)`

アーカイブ中のファイルの大きさをバイト数で返します。アーカイブは読み込みまたは追記モードで開かれていなくてはなりません。

`testzip()`

アーカイブ中の全てのファイルを読み、CRC チェックサムが正常か調べます。アーカイブ中で不正なチェックサムをもつ最初のファイルの名前を返します。不正なファイルがなければ `None` を返します。

`write(filename[, arcname[, compress_type]])`

ファイル名 *filename* のファイルに、*arcname* のアーカイブ名 (デフォルトでは *filename* と同じになります) でアーカイブを書き出します。*compress\_type* が与えられていれば、コンストラクタを使って生成したときに与えた *compression* パラメタは上書きされます。アーカイブのモードは 'w' または 'a' でなくてはなりません。

`writestr(zinfo_or_arcname, bytes)`

文字列 *bytes* をアーカイブに書き込みます。*zinfo\_or\_arcname* はアーカイブ中で指定するファイル名か、または `ZipInfo` インスタンスを指定します。*zinfo\_or\_arcname* に `ZipInfo` インスタンスを指定する場合、*zinfo* インスタンスには少なくともファイル名、日付および時刻を指定しなければなりません。ファイル名を指定した場合、日付と時刻には現在の日付と時間が設定されます。アーカイブはモード 'w' または 'a' で開かれていなければなりません。

以下のデータ属性も利用することができます。

`debug`

使用するデバッグ出力レベル。この属性は 0 (デフォルト、何も出力しない) から 3 (最も多くデバッグ情報を出力する) までの値に設定することができます。デバッグ情報は `sys.stdout` に出力されます。

## 7.18.2 PyZipFile オブジェクト

`PyZipFile` コンストラクタは `ZipFile` コンストラクタと同じパラメタを必要とします。インスタンスは `ZipFile` のメソッドの他に、追加のメソッドを一つ持ちます。

`writepy(pathname[, basename])`

'\*.py' ファイルを探し、'\*.py' ファイルに対応するファイルをアーカイブに追加します。対応するファイルとは、もしあれば '\*.pyo' であり、そうでなければ '\*.pyc' で、必要に応じて '\*.py' からコンパイルします。もし *pathname* がファイルなら、ファイル名は '.py' で終わっていなければなりません。また、('\*.py' に対応する '\*.py[co]') ファイルはアーカイブのトップレベルに (パス情報なしで) 追加されます。もし *pathname* がディレクトリで、ディレクトリがパッケージディレクトリでないなら、全ての '\*.py[co]' ファイルはトップレベルに追加されます。もしディレクトリがパッケージディレクトリなら、全ての '\*.py[co]' ファイルはパッケージ名の名前をもつファイルパスの下に追加されます。サブディレクトリがパッケージディレクトリなら、それらは再帰的に追加されます *basename* はクラス内部での呼び出しに使用するためのものです。

`writepy()` メソッドは以下のようなファイル名を持ったアーカイブを生成します。

```

string.pyc          # トップレベル名
test/__init__.pyc   # パッケージディレクトリ
test/testall.pyc    # test.testall モジュール
test/bogus/__init__.pyc # サブパッケージディレクトリ
test/bogus/myfile.pyc # test.bogus.myfile サブモジュール

```

### 7.18.3 ZipInfo オブジェクト

ZipFile オブジェクトの `getinfo()` および `infolist()` メソッドは ZipInfo クラスのインスタンスを返します。それぞれのインスタンスオブジェクトは ZIP アーカイブの一個のメンバについての情報を保持しています。

インスタンスは以下の属性を持ちます:

**filename**

アーカイブ中のファイルの名前。

**date\_time**

アーカイブメンバの最終更新日時。この属性は 6 つの値からなるタプルです。:

Index	Value
0	西暦年
1	月 (1 から始まる)
2	日 (1 から始まる)
3	時 (0 から始まる)
4	分 (0 から始まる)
5	秒 (0 から始まる)

**compress\_type**

アーカイブメンバの圧縮形式。

**comment**

各アーカイブメンバに対するコメント。

**extra**

拡張フィールドデータ。この文字列データに含まれているデータの内部構成については、*PKZIP Application Note* でコメントされています。

**create\_system**

ZIP アーカイブを作成したシステムを記述する文字列。

**create\_version**

このアーカイブを作成した PKZIP のバージョン。

**extract\_version**

このアーカイブを展開する際に必要な PKZIP のバージョン。

**reserved**

予約領域。ゼロでなくてはなりません。

**flag\_bits**

ZIP フラグビット列。

**volume**

ファイルヘッダのボリュームナンバ。

**internal\_attr**

内部属性。

**external\_attr**  
外部ファイル属性。

**header\_offset**  
ファイルヘッダへのバイト数で表したオフセット。

**file\_offset**  
ファイルデータの開始点へのバイト数で表したオフセット。

**CRC**  
圧縮前のファイルの CRC-32 チェックサム。

**compress\_size**  
圧縮後のデータのサイズ。

**file\_size**  
圧縮前のファイルのサイズ。

## 7.19 tarfile — tar アーカイブファイルを読み書きする

2.3 で追加された仕様です。

tarfile モジュールは、tar アーカイブを読んで作成することができるようになります。いくつかの事実と外観：

- gzip と bzip2 で圧縮されたアーカイブを読み書きします。
- POSIX 1003.1-1990 準拠あるいは GNU tar 互換のアーカイブを作成します。
- GNU tar 拡張機能 長い名前、*longlink* および *sparse* を読みます。
- GNU tar 拡張機能を使って、無制限長さのパス名を保存します。
- ディレクトリ、普通のファイル、ハードリンク、シンボリックリンク、fifo、キャラクタデバイスおよびブロックデバイスを処理します。また、タイムスタンプ、アクセス許可およびオーナーのようなファイル情報の取得および保存が可能です。
- テープデバイスを取り扱うことができます。

`open([name[, mode[, fileobj[, bufsize]]]])`

パス名 *name* に TarFile オブジェクトを返します。TarFile オブジェクトに関する詳細な情報については、TarFile オブジェクト (セクション 7.19.1) を見て下さい。

*mode* has to be a string of the form 'filemode[:compression]', it defaults to 'r'. Here is a full list of mode combinations: *mode* は、' ファイルモード[:圧縮]' の形の文字列でなければならず、そのデフォルトは 'r' です。以下にモードの組み合わせの完全な一覧を示します。

mode	動作
'r'	透過な圧縮つきで読み込むためにオープンします (推奨)。
'r:'	圧縮なしで排他的に読み込むためにオープンします。
'r:gz'	gzip 圧縮で読み込むためにオープンします。
'r:bz2'	bzip2 圧縮で読み込むためにオープンします。
'a' または 'a:'	圧縮なしで追加するためにオープンします。
'w' または 'w:'	非圧縮で書き込むためにオープンします。
'w:gz'	gzip 圧縮で書き込むためにオープンします。
'w:bz2'	bzip2 圧縮で書き込むためにオープンします。

‘a:gz’ あるいは ‘a:bz2’ は可能ではないことに注意して下さい。もし *mode* が、ある (圧縮した) ファイルを読み込み用にオープンするのに、適していないなら、`ReadError` が発生します。これを防ぐには *mode* ‘r’ を使って下さい。もし圧縮メソッドがサポートされていないければ、`CompressionError` が発生します。

もし *fileobj* が指定されていれば、それは *name* でオープンされたファイルオブジェクトの代替として使うことができます。

特別な目的のために、*mode* の 2 番目のフォーマット: ‘ファイルモード|[圧縮]’ があります。open は、そのデータをブロックのストリームとして処理する `TarFile` オブジェクトを返します。ランダムシーキングはそのファイルにはなされません。もし *fileobj* が与えられていれば、それは、それぞれ `read()`、`write()` メソッドを持つ、任意のオブジェクトで構いません。*bufsize* は、ブロックサイズを指定し、デフォルトでは 20 \* 512 バイトです。例えば `sys.stdin` とソケットファイルオブジェクトやテープデバイスの組み合わせでは、これを変更して使って下さい。しかし、そのような `TarFile` オブジェクトには、ランダムにアクセスされることを許さないという制限があります、例 (セクション 7.19.3) を見て下さい。現在可能なモードは：

mode	動作
‘r ’	非圧縮 tar ブロックのストリームを読み込みにオープンします。
‘r gz’	gzip 圧縮 ストリームを読み込みにオープンします。
‘r bz2’	bzip2 圧縮 ストリームを読み込みにオープンします。
‘w ’	非圧縮 ストリームを書き込みにオープンします。
‘w gz’	gzip 圧縮 ストリームを書き込みにオープンします。
‘w bz2’	bzip2 圧縮 ストリームを書き込みにオープンします。

#### class `TarFile`

tar アーカイブを読んだり、書いたりするためのクラスです。このクラスを直接使わないで下さい、その代わりに `open()` を使った方が良いです。`TarFile` オブジェクト (セクション 7.19.1) を見て下さい。

#### `is_tarfile(name)`

もし *name* が tar アーカイブファイルであれば、`True` を返し、その `tarfile` モジュールで読むことができます。

#### class `TarFileCompat(filename[, mode[, compression]])`

zipfile-風なインターフェースを持つ tar アーカイブへの制限されたアクセスのためのクラスです。より詳細については `zipfile` のドキュメントに当たって下さい。*compression* は、以下の定数のどれかでなければなりません：

##### `TAR_PLAIN`

非圧縮 tar アーカイブのための定数。

##### `TAR_GZIPPED`

gzip 圧縮 tar アーカイブのための定数。

#### exception `TarError`

すべての `tarfile` 例外のための基本クラスです。

#### exception `ReadError`

tar アーカイブがオープンされた時、`tarfile` モジュールで操作できないか、あるいは何か無効であるとき発生します。

#### exception `CompressionError`

圧縮方法がサポートされていないか、あるいはデータを正しくデコードできない時に発生します。

#### exception `StreamError`

ストリーム風の `TarFile` オブジェクトで典型的な制限のために発生します。

#### exception `ExtractError`

`extract()` を使った時、もし `TarFile.errorlevel == 2` のフェータルでないエラーに対してだけ発生します。

参考資料:

zipfile モジュール (7.18 節):

zipfile 標準モジュールのドキュメント。

GNU tar マニュアル、標準セクション

([http://www.gnu.org/manual/tar/html\\_chapter/tar\\_8.html#SEC118](http://www.gnu.org/manual/tar/html_chapter/tar_8.html#SEC118))

GNU tar 拡張機能を含む、tar アーカイブファイルのためのドキュメント。

### 7.19.1 TarFile オブジェクト

`TarFile` オブジェクトは、tar アーカイブへのインターフェースを提供します。tar アーカイブは一連のブロックです。アーカイブメンバー (保存されたファイル) は、ヘッダーブロックとそれに続くデータブロックから構成されています。ある tar アーカイブにファイルを何回も保存することができます。各アーカイブメンバーは、`TarInfo` オブジェクトによって表わされます、詳細については *TarInfo* オブジェクト (セクション 7.19.2) を見て下さい。

`class TarFile([name[, mode[, fileobj]]])`

(非圧縮の) tar アーカイブ *name* をオープンします。 *mode* は、既存のアーカイブから読み込むには 'r'、既存のファイルにデータを追加するには 'a'、あるいは既存のファイルを上書きして新しいファイルを作成するには 'w' のどれかです。 *mode* のデフォルトは 'r' です。

もし *fileobj* が与えられていれば、それを使ってデータを読み書きします。もしそれが決定できれば、*mode* は *fileobj* のモードで上書きされます。

注意: *fileobj* は、`TarFile` をクローズする時は、クローズされません。

`open(...)`

代替コンストラクタです。モジュールレベルでの `open()` 関数は、実際はこのクラスメソッドへのショートカットです。詳細についてはセクション 7.19 を見て下さい。

`getmember(name)`

メンバー *name* に対する `TarInfo` オブジェクトを返します。もし *name* がアーカイブに見つからなければ、`KeyError` が発生します。

注意: もしメンバーがアーカイブに 1 つ以上あれば、その最後に出現するものが、最新のバージョンであるとみなされます。

`getmembers()`

`TarInfo` オブジェクトのリストとしてアーカイブのメンバーを返します。このリストはアーカイブ内のメンバーと同じ順番です。

`getnames()`

メンバーをその名前前のリストとして返します。これは `getmembers()` で返されるリストと同じ順番です。

`list(verbose=True)`

コンテンツの表を `sys.stdout` に印刷します。もし *verbose* が `False` であれば、メンバー名のみ印刷します。もしそれが `True` であれば、`"ls -l"`-風の出力が作成されます。

`next()`

`TarFile` が読み込み用にオープンされている時、アーカイブの次のメンバーを `TarInfo` オブジェクトとして返します。もしそれ以上利用可能なものがなければ、`None` を返します。

`extract(member[, path])`

メンバーをアーカイブから現在の作業ディレクトリに、そのフル名を使って、抽出します。そのファイル情報はできるだけ正確に抽出されます。*member* は、ファイル名でも `TarInfo` オブジェクトでも構いません。*path* を使って、異なるディレクトリを指定することができます。

**extractfile(*member*)**

アーカイブからメンバーをオブジェクトとして抽出します。*member* は、ファイル名あるいは `TarInfo` オブジェクトです。もし *member* が普通のファイルであれば、ファイル風のオブジェクトを返します。もし *member* がリンクであれば、ファイル風のオブジェクトをリンクのターゲットから構成します。もし *member* が上のどれでもなければ、`None` が返されます。

注意: ファイル風のオブジェクトは読み出し専用で以下のメソッドを提供します: `read()`, `readline()`, `readlines()`, `seek()`, `tell()`。

**add(*name* [, *arcname* [, *recursive=True* ]])**

ファイル *name* をアーカイブに追加します。*name* は、任意のファイルタイプ (ディレクトリ、`fifo`、シンボリックリンク等) です。もし *arcname* が与えられていれば、それはアーカイブ内のファイルの代替名を指定します。デフォルトではディレクトリは再帰的に追加されます。これは、*recursive* を `False` に設定することで避けることができます。

**addfile(*tarinfo* [, *fileobj* ])**

`TarInfo` オブジェクト *tarinfo* をアーカイブに追加します。もし *fileobj* が与えられていれば、`tarinfo.size` バイトがそれから読まれ、アーカイブに追加されます。`gettaringfo()` を使って `TarInfo` オブジェクトを作成することができます。

注意: Windows プラットフォームでは、*fileobj* は、ファイルサイズに関する問題を避けるために、常に、モード `'rb'` でオープンされるべきです。

**gettaringfo([*name* [, *arcname* [, *fileobj* ]]])**

`TarInfo` オブジェクトをファイル *name* あるいは (そのファイル記述子に `os.fstat()` を使って) ファイルオブジェクト *fileobj* のどちらか用に作成します。`TarInfo` の属性のいくつかは、`addfile()` を使って追加する前に修正することができます。*arcname* がもし与えられていれば、アーカイブ内のファイルの代替名を指定します。

**close()**

`TarFile` をクローズします。書き出しモードでは、完了ゼロブロックが 2 つ、アーカイブに追加されます。

**posix=True**

もし `True` なら、POSIX 1003.1-1990 準拠のアーカイブを作成します。GNU 拡張機能は使われません、というのは、それらは POSIX 標準の一部ではないからです。これはファイル名の長さを最大 256 に、linkname を 100 文字に制限します。もしパス名がこの制限を越えたら、`ValueError` が発生します。もし `False` であれば、GNU tar 互換アーカイブを作成します。それは POSIX 準拠ではありませんが、無制限長さのパス名を保管することができます。

**dereference=False**

もし `False` であれば、シンボリックリンクとハードリンクをアーカイブに追加します。もし `True` であれば、ターゲットファイルの内容をアーカイブに追加します。これは、リンクをサポートしないシステムには何も効果もありません。

**ignore\_zeros=False**

もし `False` であれば、空のブロックをアーカイブの終わりとして処理します。もし `True` であれば、空 (で無効な) ブロックは飛ばして、できるだけ多くのメンバーを取得しようとします。これは連結した、あるいは損傷したアーカイブでのみ役に立ちます。

**debug=0**

0 (デバッグメッセージなし) から 3 (すべてのデバッグメッセージあり) までを設定すべきです。その



メッセージは `sys.stdout` に書かれます。

**errorlevel=0**

もし 0 なら、`extract()` を使っている時、すべてのエラーが無視されます。それでも、デバッグングが有効である時は、それらは、デバッグ出力にエラーメッセージとして出力されます。もし 1 なら、すべてのフェータルエラーが、`OSError` あるいは `IOError` 例外として発生します。もし 2 なら、すべてのフェータルでないエラーが、`TarError` 例外として、やはり発生します。

### 7.19.2 TarInfo オブジェクト

`TarInfo` オブジェクトは `TarFile` の一つのメンバーを表します。ファイルに必要な (ファイルタイプ、ファイルサイズ、時刻、許可、所有者等のような) すべての属性を保存する他に、そのタイプを決定するのに役に立ついくつかのメソッドを提供します。これにはファイルのデータそのものは含まれません。

`TarInfo` オブジェクトは `TarFile` のメソッド `getmember()`、`getmembers()` および `gettarinfo()` によって返されます。

**class TarInfo([name])**

`TarInfo` オブジェクトを作成します。

**frombuf()**

`TarInfo` オブジェクトを文字列バッファから作成して返します。

**tobuf()**

`TarInfo` オブジェクトから文字列バッファを作成します。

`TarInfo` オブジェクトには以下の public なデータ属性があります：

**name**

アーカイブメンバーの名前。

**size**

バイト単位でのサイズ。

**mtime**

最終更新時刻。

**mode**

許可ビット。

**type**

ファイルタイプ。 *type* は普通、以下の定数のどれかです：`REGTYPE`、`AREGTYPE`、`LNKTYPE`、`SYMTYPE`、`DIRTYPE`、`FIFOTYPE`、`CONTTYTYPE`、`CHRTYPE`、`BLKTYPE`、`GNUTYPE_SPARSE`。`TarInfo` オブジェクトのタイプをもっと便利に決定するには、以降の `is_*`() メソッドを使って下さい。

**linkname**

ターゲットファイル名の名前で、これはタイプ `LNKTYPE` と `SYMTYPE` の `TarInfo` オブジェクトにだけ存在します。

**uid, gid**

このメンバーを本来保存した人のユーザとグループ ID。

**uname, gname**

ユーザとグループ名

`TarInfo` オブジェクトは便利な照会用のメソッドもいくつか提供しています：

**isfile()**

もし `Tarinfo` オブジェクトが普通のファイルであれば、`True` を返します。

`isreg()`

`isfile()` と同じです。

`isdir()`

もしそれがディレクトリであれば `True` を返します。

`issym()`

もしそれがシンボリックリンクであれば `True` を返します。

`islnk()`

もしそれがハードリンクであれば `True` を返します。

`ischr()`

もしそれがキャラクタデバイスであれば `True` を返します。

`isblk()`

もしそれがブロックデバイスであれば `True` を返します。

`isfifo()`

もしそれが FIFO であれば `True` を返します。

`isdev()`

もしそれが、キャラクタデバイス、ブロックデバイスあるいは FIFO のどれかであれば `True` を返します。

### 7.19.3 例

非圧縮 tar アーカイブをファイル名のリストから作成する方法：

```
import tarfile
tar = tarfile.open("sample.tar", "w")
for name in ["foo", "bar", "quux"]:
    tar.add(name)
tar.close()
```

gzip 圧縮 tar アーカイブを作成してメンバー情報のいくつかを表示する方法：

```
import tarfile
tar = tarfile.open("sample.tar.gz", "r:gz")
for tarinfo in tar:
    print tarinfo.name, " は 大きさが ", tarinfo.size, " バイトで ",
    if tarinfo.isreg():
        print " 普通のファイルです。 "
    elif tarinfo.isdir():
        print " ディレクトリです。 "
    else:
        print " ファイル・ディレクトリ以外のものです。 "
tar.close()
```

見せかけの情報を持つ tar アーカイブを作成する方法：

```

import tarfile
tar = tarfile.open("sample.tar.gz", "w:gz")
for name in namelist:
    tarinfo = tar.gettarinfo(name, "fakeproj-1.0/" + name)
    tarinfo.uid = 123
    tarinfo.gid = 456
    tarinfo.uname = "johndoe"
    tarinfo.gname = "fake"
    tar.addfile(tarinfo, file(name))
tar.close()

```

非圧縮 tar ストリームを `sys.stdin` から抽出する唯一の方法：

```

import sys
import tarfile
tar = tarfile.open(mode="r|", fileobj=sys.stdin)
for tarinfo in tar:
    tar.extract(tarinfo)
tar.close()

```

## 7.20 readline — GNU readline のインタフェース

readline モジュールでは、補完をしやすいしたり、ヒストリファイルを Python インタプリタから読み書きできるようにするためのいくつかの関数を定義しており、これらは直接使うことも `rlcompleter` モジュールを介して使うこともできます。

readline モジュールでは以下の関数を定義しています：

**parse\_and\_bind(*string*)**

readline 初期化ファイルの行を一行解釈して実行します。

**get\_line\_buffer()**

行編集バッファの現在の内容を返します。

**insert\_text(*string*)**

コマンドラインにテキストを挿入します。

**read\_init\_file(*[filename]*)**

readline 初期化ファイルを解釈します。標準のファイル名設定は最後に使われたファイル名です。

**read\_history\_file(*[filename]*)**

readline ヒストリファイルを読み出します。標準のファイル名設定は `“~/.history”` です。

**write\_history\_file(*[filename]*)**

readline ヒストリファイルを保存します。標準のファイル名設定は `“~/.history”` です。

**get\_history\_length()**

ヒストリファイルに必要な長さを返します。負の値はヒストリファイルのサイズに制限がないことを示します。

**set\_history\_length(*length*)**

ヒストリファイルに必要な長さを設定します。この値は `write_history_file()` がヒストリを保存する際にファイルを切り結めるために使います。負の値はヒストリファイルのサイズを制限しないことを示します。

**set\_startup\_hook(*[function]*)**

startup\_hook 関数を設定または除去します。function が指定されていれば、新たな startup\_hook 関数として用いられます; 省略されるか None になっていれば、現在インストールされているフック関数は除去されます。startup\_hook 関数は readline が最初のプロンプトを出力する直前に引数なしで呼び出されます。

**set\_pre\_input\_hook**( [function] )

pre\_input\_hook 関数を設定または除去します。function が指定されていれば、新たな pre\_input\_hook 関数として用いられます; 省略されるか None になっていれば、現在インストールされているフック関数は除去されます。pre\_input\_hook 関数は readline が最初のプロンプトを出力した後で、かつ readline が入力された文字を読み込み始める直前に引数なしで呼び出されます。

**set\_completer**( [function] )

completer 関数を設定または除去します。function が指定されていれば、新たな completer 関数として用いられます; 省略されるか None になっていれば、現在インストールされている completer 関数は除去されます。completer 関数は function(text, state) の形式で、関数が文字列でない値を返すまで state を 0, 1, 2, ..., にして呼び出します。この関数は text から始まる文字列の補完結果として可能性のあるものを返さなくてはなりません。

**get\_completer**( )

completer 関数を取得します。completer 関数が設定されていなければ None を返します。2.3 で追加された仕様です。

**get\_begidx**( )

readline タブ補完スコープの先頭のインデックスを取得します。

**get\_endidx**( )

readline タブ補完スコープの末尾のインデックスを取得します。

**set\_completer\_delims**( string )

タブ補完のための readline 単語区切り文字を設定します。

**get\_completer\_delims**( )

タブ補完のための readline 単語区切り文字を取得します。

**add\_history**( line )

1 行をヒストリバッファに追加し、最後に打ち込まれた行のようにします。

参考資料:

rlcompleter モジュール (7.21 節):

対話的プロンプトで Python 識別子を補完する機能。

### 7.20.1 例

以下の例では、ユーザのホームディレクトリにある '.pyhist' という名前のヒストリファイルを自動的に読み書きするために、readline モジュールによるヒストリの読み書き関数をどのように使うかを例示しています。以下のソースコードは通常、対話セッションの中で PYTHONSTARTUP ファイルから読み込まれ自動的に実行されることになります。

```

import os
histfile = os.path.join(os.environ["HOME"], ".pyhist")
try:
    readline.read_history_file(histfile)
except IOError:
    pass
import atexit
atexit.register(readline.write_history_file, histfile)
del os, histfile

```

## 7.21 rlcompleter — GNU readline 向け補完関数

rlcompleter モジュールでは Python の識別子やキーワードを定義した readline モジュール向けの補完関数を定義しています。

readline モジュールが UNIX 依存なのでこのモジュールも UNIX に依存しています。

rlcompleter モジュールは Completer クラスを定義しています。

使用例:

```

>>> import rlcompleter
>>> import readline
>>> readline.parse_and_bind("tab: complete")
>>> readline. <TAB PRESSED>
readline.__doc__          readline.get_line_buffer  readline.read_init_file
readline.__file__         readline.insert_text      readline.set_completer
readline.__name__         readline.parse_and_bind
>>> readline.

```

rlcompleter モジュールは Python の対話モードで利用する為にデザインされています。ユーザは以下の命令を初期化ファイル (環境変数 PYTHONSTARTUP によって定義されます) に書き込むことで、Tab キーによる補完を利用できます:

```

try:
    import readline
except ImportError:
    print "Module readline not available."
else:
    import rlcompleter
    readline.parse_and_bind("tab: complete")

```

### 7.21.1 Completer オブジェクト

Completer オブジェクトは以下のメソッドを持っています:

**complete**(*text*, *state*)

*text* の *state* 番目の補完候補を返します。

もし *text* がピリオド (‘.’) を含まない場合、\_\_main\_\_、\_\_builtin\_\_ で定義されている名前か、キーワード (keyword モジュールで定義されている) から補完されます。

ピリオドを含む名前の場合、副作用を出さずに名前を最後まで評価しようとします (関数を明示的に呼び出しはしませんが、\_\_getattr\_\_() を呼んでしまうことはあります) そして、dir() 関数で

マッチする語を見つけます。



# Unix 独特のサービス

本章に記述されたモジュールは、UNIX オペレーティングシステム、あるいはそれから変形した多くのものに特有する機能のためのインターフェイスを提供します。その概要を以下に述べます。

<b>posix</b>	最も一般的な POSIX システムコール群 (通常は <code>os</code> モジュールを介して利用されます)。
<b>pwd</b>	パスワードデータベースへのアクセスを提供する ( <code>getpwnam()</code> など)。
<b>grp</b>	グループデータベースへのアクセス ( <code>getgrnam()</code> およびその仲間)。
<b>crypt</b>	UNIX パスワードをチェックするための関数 <code>crypt()</code> 。
<b>dl</b>	共有オブジェクトの C 関数の呼び出し
<b>dbm</b>	ndbm を基にした基本的なデータベースインタフェースです。
<b>gdbm</b>	GNU による dbm の再実装。
<b>termios</b>	POSIX スタイルの端末制御。
<b>TERMIOS</b>	<code>termios</code> モジュールを利用する上で必要となるシンボル定数。
<b>tty</b>	一般的な端末制御操作のためのユーティリティ関数群。
<b>pty</b>	SGI と Linux 用の擬似端末を制御する
<b>fcntl</b>	<code>fcntl()</code> および <code>ioctl()</code> システムコール。
<b>pipes</b>	Python による UNIX シェルパイプラインへのインタフェース。
<b>posixfile</b>	ロック機構をサポートするファイル類似オブジェクト。
<b>resource</b>	現プロセスのリソース使用状態を提供するためのインタフェース。
<b>nis</b>	Sun の NIS (Yellow Pages) ライブラリへのインタフェース。
<b>syslog</b>	UNIX syslog ライブラリルーチン群へのインタフェース。
<b>commands</b>	外部コマンドを実行するためのユーティリティです。

## 8.1 posix — 最も一般的な POSIX システムコール群

このモジュールはオペレーティングシステムの機能のうち、C 言語標準および POSIX 標準 (UNIX インタフェースをほんの少し隠蔽した) で標準化されている機能に対するアクセス機構を提供します。

このモジュールを直接 `import` しないで下さい。その代わりに、移植性のあるインタフェースを提供している `os` をインポートしてください。UNIX では、`os` モジュールが提供するインタフェースは `posix` の内容を内包しています。非 UNIX オペレーティングシステムでは `posix` モジュールを使うことはできませんが、その部分的な機能セットは、たいてい `os` インタフェースを介して利用することができます。`os` は、一度 `import` してしまえば `posix` の代わりであることによるパフォーマンス上のペナルティは全くありません。その上、`os` は `os.environ` の内容が変更された際に自動的に `putenv()` を呼ぶなど、いくつかの追加機能を提供しています。

以下の説明は非常に簡潔なものです; 詳細については、UNIX マニュアルの (または POSIX) ドキュメントの) 対応する項目を参照してください。 *path* で呼ばれる引数は文字列で与えられたパス名を表します。

エラーは例外として報告されます; よくある例外は型エラーです。一方、システムコールから報告されたエラーは以下に述べるように `error` (標準例外 `OSError` と同義です) を送出します。

### 8.1.1 ラージファイルのサポート

いくつかのオペレーティングシステム (AIX, HPIX, Irix および Solaris が含まれます) は、`int` および `long` を 32 ビット値とする C プログラムモデルで 2Gb を超えるサイズのファイルのサポートを提供しています。このサポートは典型的には 64 ビット値のオフセット値と、そこからの相対サイズを定義することで実現しています。このようなファイルは時にラージファイル (*large files*) と呼ばれます。

Python では、`off_t` のサイズが `long` より大きく、かつ `long long` 型を利用することができて、少なくとも `off_t` 型と同じくらい大きなサイズである場合、ラージファイルのサポートが有効になります。この場合、ファイルのサイズ、オフセットおよび Python の通常整数型の範囲を超えるような値の表現には Python の長整数型が使われます。例えば、ラージファイルのサポートは Irix の最近のバージョンでは標準で有効ですが、Solaris 2.6 および 2.7 では、以下のようにする必要があります：

```
CFLAGS="`getconf LFS_CFLAGS`" OPT="-g -O2 $CFLAGS" \  
./configure
```

On large-file-capable Linux systems, this might work:

```
CFLAGS='-D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64' OPT="-g -O2 $CFLAGS" \  
./configure
```

### 8.1.2 モジュールの内容

`posix` では以下のデータ項目を定義しています：

#### `environ`

インタプリタが起動した時点の環境変数文字列を表現する辞書です。例えば、`environ['HOME']` はホームディレクトリのパス名で、C 言語の `getenv("HOME")` と等価です。

この辞書を変更しても、`execv()`、`popen()` または `system()` などに渡される環境変数文字列には影響しません；そうした環境を変更する必要がある場合、`environ` を `execve()` に渡すか、`system()` または `popen()` の命令文字列に変数の代入や `export` 文を追加してください。

注意: `os` モジュールでは、もう一つの `environ` 実装を提供しており、環境変数が変更された場合、その内容を更新するようになっています。`os.environ` を更新した場合、この辞書は古い内容を表示していることになってしまうので、このことにも注意してください。`posix` モジュール版を直接アクセスするよりも、`os` モジュール版を使う方が推奨されています。

このモジュールのその他の内容は `os` モジュールからのみのアクセスになっています；詳しい説明は `os` モジュールのドキュメントを参照してください。

## 8.2 `pwd` — パスワードデータベースへのアクセスを提供する

このモジュールは UNIX のユーザアカウントとパスワードのデータベースへのアクセスを提供します。全ての UNIX 系 OS で利用できます。

パスワードデータベースの各エントリはタプルのようなオブジェクトで提供され、それぞれの属性は `passwd` 構造体のメンバに対応しています (下の属性欄については、`<pwd.h>` を見てください)。

インデックス	属性	意味
0	pw_name	ログイン名
1	pw_passwd	暗号化されたパスワード (optional))
2	pw_uid	ユーザ ID(UID)
3	pw_gid	グループ ID(GID)
4	pw_gecos	実名またはコメント
5	pw_dir	ホームディレクトリ
6	pw_shell	シェル

UID と GID は整数で、それ以外は全て文字列です。検索したエントリが見つからないと `KeyError` が発生します。

注意: 伝統的な UNIX では、`pw_passwd` フィールドは DES 由来のアルゴリズムで暗号化されたパスワード (`crypt` モジュールをのぞいてください) が含まれています。しかし、近代的な UNIX 系 OS ではシャドウパスワードとよばれる仕組みを利用しています。この場合には `pw_passwd` フィールドにはアスタリスク ('\*') か、'x' という一文字だけが含まれており、暗号化されたパスワードは、一般には見えない `/etc/shadow` というファイルに入っています。

このモジュールでは以下のものが定義されています:

`getpwuid(uid)`

与えられた UID に対応するパスワードデータベースのエントリを返します。

`getpwnam(name)`

与えられたユーザ名に対応するパスワードデータベースのエントリを返します。

`getpwall()`

パスワードデータベースの全てのエントリを、任意の順番で並べたリストを返します。

参考資料:

`grp` モジュール (8.3 節):

このモジュールに似た、グループデータベースへのアクセスを提供するモジュール。

## 8.3 grp — グループデータベースへのアクセス

このモジュールでは UNIX グループ (group) データベースへのアクセス機構を提供します。全ての UNIX バージョンで利用可能です。

このモジュールはグループデータベースのエントリをタプルに似たオブジェクトとして報告されます。このオブジェクトの属性は `group` 構造体の各メンバ (以下の属性フィールド、`<pwd.h>` を参照) に対応します:

インデックス	属性	意味
0	gr_name	グループ名
1	gr_passwd	(暗号化された) グループパスワード; しばしば空文字列になります
2	gr_gid	数字のグループ ID
3	gr_mem	グループメンバの全てのユーザ名

`gid` は整数、名前およびパスワードは文字列、そしてメンバリストは文字列からなるリストです。(ほとんどのユーザは、パスワードデータベースで自分が入れているグループのメンバとしてグループデータベース内では明示的に列挙されていないので注意してください。完全なメンバ情報を取得するには両方のデータベースを調べてください。)

このモジュールでは以下の内容を定義しています:

`getgrgid(gid)`

与えられたグループ ID に対するグループデータベースエントリを返します。要求したエントリが見つからなかった場合、`KeyError` が送出されます。

`getgrnam(name)`

与えられたグループ名に対するグループデータベースエントリを返します。要求したエントリが見つからなかった場合、`KeyError` が送出されます。

`getgrall()`

全ての入手可能なグループエントリを返します。順番は決まっています。

参考資料:

`pwd` モジュール (8.2 節):

このモジュールと類似の、ユーザデータベースへのインタフェース。

## 8.4 `crypt` — UNIX パスワードをチェックするための関数

このモジュールは DES アルゴリズムに基づいた一方方向ハッシュ関数である `crypt(3)` ルーチンを実装しています。詳細については UNIX マニュアルページを参照してください。このモジュールは、Python スクリプトがユーザから入力されたパスワードを受理できるようにしたり、UNIX パスワードに (脆弱性検査のための) 辞書攻撃を試みるのに使えます。

`crypt(word, salt)`

`word` は通常はユーザのパスワードで、プロンプトやグラフィカルインタフェースからタイプ入力されます。`salt` は通常ランダムな 2 文字からなる文字列で、DES アルゴリズムに 4096 通りのうち 1 つの方法で外乱を与えるために使われます。`salt` に使う文字は集合「`./a-zA-Z0-9`」の要素でなければなりません。ハッシュされたパスワードを文字列として返します。パスワード文字列は `salt` と同じ文字集合に含まれる文字からなります (最初の 2 文字は `salt` 自体です)。

典型的な使用例のサンプルコード:

```
import crypt, getpass, pwd

def login():
    username = raw_input('Python login:')
    cryptpasswd = pwd.getpwnam(username)[1]
    if cryptpasswd:
        if cryptpasswd == 'x' or cryptpasswd == '*':
            raise "Sorry, currently no support for shadow passwords"
        cleartext = getpass.getpass()
        return crypt.crypt(cleartext, cryptpasswd[:2]) == cryptpasswd
    else:
        return 1
```

## 8.5 `dl` — 共有オブジェクトの C 関数の呼び出し

`dl` モジュールは `dlopen()` 関数へのインターフェースを定義します。これはダイナミックライブラリにハンドルするための UNIX プラットフォーム上の最も一般的なインターフェースです。そのライブラリの任意の関数を呼び出すプログラムを与えます。

注意: このモジュールは `sizeof(int) == sizeof(long) == sizeof(char *)` でなければ動きません。そうでなければ `import` するときに `SystemError` が送出されるでしょう。

dl モジュールは次の関数を定義します:

**open**(*name*[, *mode* = *RTLD\_LAZY* ])

共有オブジェクトファイルを開いて、ハンドルを返します。モードは遅延結合 (*RTLD\_LAZY*) または即時結合 (*RTLD\_NOW*) を表します。デフォルトは *RTLD\_LAZY* です。いくつかのシステムは *RTLD\_NOW* をサポートしていないことに注意してください。

返り値は *dlobject* です。

dl モジュールは次の定数を定義します:

**RTLD\_LAZY**

*open*() の引数として使います。

**RTLD\_NOW**

*open*() の引数として使います。即時結合をサポートしないシステムでは、この定数がモジュールに現われないことに注意してください。最大のポータビリティを求めるならば、システムが即時結合をサポートするかどうかを決定するために *hasattr*() を使用してください。

dl モジュールは次の例外を定義します:

**exception error**

動的なロードやリンクルーチンの内部でエラーが生じたときに送出される例外です。

例:

```
>>> import dl, time
>>> a=dl.open('/lib/libc.so.6')
>>> a.call('time'), time.time()
(929723914, 929723914.498)
```

この例は Debian GNU/Linux システム上で行なったもので、このモジュールの使用はたいてい悪い選択肢であるという事実のよい例です。

## 8.5.1 DI オブジェクト

*open*() によって返された DI オブジェクトは次のメソッドを持っています:

**close**()

メモリーを除く全てのリソースを解放します。

**sym**(*name*)

*name* という名前の関数が参照された共有オブジェクトに存在する場合、そのポインター (整数値) を返します。存在しない場合 *None* を返します。これは次のように使えます:

```
>>> if a.sym('time'):
...     a.call('time')
... else:
...     time.time()
```

(0 は *NULL* ポインターであるので、この関数は 0 でない数を返すだろうということに注意してください)

**call**(*name*[, *arg1*[, *arg2*... ]])

参照された共有オブジェクトの *name* という名前の関数を呼出します。引数は、Python 整数 (そのまま渡される)、Python 文字列 (ポインターが渡される)、*None* (*NULL* として渡される) のどれかでなければいけません。Python はその文字列が変化させられるのを好まないの、文字列は *const char\** として関数に渡されるべきであることに注意してください。

最大で 10 個の引数が渡すことができ、与えられない引数は `None` として扱われます。関数の返り値は `C long`(Python 整数である) です。

## 8.6 dbm — UNIX dbm のシンプルなインタフェース

`dbm` モジュールは `UNIX(n)dbm` インタフェースのライブラリを提供します。`dbm` オブジェクトは、キーと値が必ず文字列である以外は辞書オブジェクトのようなふるまいをします。`print` 文などで `dbm` インスタンスを出力してもキーと値は出力されません。また、`items()` と `values()` メソッドはサポートされません。

このモジュールは、BSD DB、GNU GDBM 互換インタフェースを持ったクラシックな `ndbm` インタフェースを使うことができます。UNIX 上のビルド時に `configure` スクリプトで適切なヘッダファイルが割り当てられます。

以下はこのモジュールの定義:

### exception error

I/O エラーのような `dbm` 特有のエラーが起きたときに上げられる値です。また、正しくないキーが与えられた場合に通常のマッピングエラーのような `KeyError` が上げられます。

### library

`ndbm` が使用している実装ライブラリ名です。

### open(filename[, flag[, mode]])

`dbm` データベースを開いて `dbm` オブジェクトを返します。引数 `filename` はデータベースのファイル名を指定します。(拡張子 `‘.dir’` や `‘.pag’` は付けません。また、BSD DB は拡張子 `‘.db’` がついたファイルが一つ作成されます。)

オプション引数 `flag` は次のような値を指定します:

Value	Meaning
<code>‘r’</code>	存在するデータベースを読み取り専用で開きます。(デフォルト)
<code>‘w’</code>	存在するデータベースを読み書き可能な状態で開きます。
<code>‘c’</code>	データベースを読み書き可能な状態で開きます。また、データベースが存在しない場合は新たに作成します。
<code>‘n’</code>	Always create a new, empty database, open for reading and writing
<code>‘n+’</code>	常に空のデータベースが作成され、読み書き可能な状態で開きます。

オプション引数 `mode` はデータベース作成時に使用される UNIX のファイルモードを指定します。デフォルトでは 8 進数の `0666` です

### 参考資料:

`anydbm` モジュール (7.10 節):

`dbm` スタイルの一般的なインタフェース

`gdbm` モジュール (8.7 節):

GNU GDBM ライブラリの類似したインタフェース

`whichdb` モジュール (7.12 節):

存在しているデータベースの形式を決めるためのユーティリティモジュール

## 8.7 gdbm — GNU による dbm の再実装

このモジュールは `dbm` モジュールによく似ていますが、`gdbm` を使っていくつかの追加機能を提供しています。`gdbm` と `dbm` では生成されるファイル形式に互換性がないので注意してください。



gdbm モジュールでは GNU DBM ライブラリへのインタフェースを提供します。gdbm オブジェクトはキーと値が常に文字列であることを除き、マップ型 (辞書型) と同じように動作します。gdbm オブジェクトに対して `print` を適用してもキーや値を印字することはなく、`items()` 及び `values()` メソッドはサポートされていません。

このモジュールでは以下の定数および関数を定義しています:

#### **exception error**

I/O エラーのような gdbm 特有のエラーで送出されます。誤ったキーの指定のように、一般的なマップ型のエラーに対しては `KeyError` が送出されます。

#### **open(filename, [flag, [mode]])**

gdbm データベースを開いて gdbm オブジェクトを返します。*filename* 引数はデータベースファイルの名前です。

オプションの *flag* としては、`'r'` (既存のデータベースを読み込み専用で開く — 標準の値です)、`'w'` (既存のデータベースを読み書き用に開く)、`'c'` (既存のデータベースが存在しない場合には新たに作成する)、または `'n'` (常に新たにデータベースを作成する)、をとることができます。

データベースをどのように開くかを制御するために、フラグに以下の文字を追加することができます:

- `'f'` — データベースを高速モードで開きます。このモードではデータベースへの書き込みはファイルシステムと同期されません。
- `'s'` — 同期モードで開きます。データベースへの変更はファイルに即座に書き込まれます。
- `'u'` — データベースをロックしません。

全てのバージョンの gdbm で全てのフラグが有効とは限りません。モジュール定数 `open_flags` はサポートされているフラグ文字からなる文字列です。無効なフラグが指定された場合、例外 `error` が送出されます。

オプションの *mode* 引数は、新たにデータベースを作成しなければならない場合に使われる UNIX のファイルモードです。標準の値は 8 進数の `0666` です。

辞書型形式のメソッドに加えて、gdbm オブジェクトには以下のメソッドがあります:

#### **firstkey()**

このメソッドと `next()` メソッドを使って、データベースの全てのキーにわたってループ処理を行うことができます。探索は gdbm の内部ハッシュ値の順番に行われ、キーの値に順に並んでいるとは限りません。このメソッドは最初のキーを返します。

#### **nextkey(key)**

データベースの順方向探索において、*key* よりも後に来るキーを返します。以下のコードはデータベース `db` について、キー全てを含むリストをメモリ上に生成することなく全てのキーを出力します:

```
k = db.firstkey()
while k != None:
    print k
    k = db.nextkey(k)
```

#### **reorganize()**

大量の削除を実行した後、gdbm ファイルの占めるスペースを削減したい場合、このルーチンはデータベースを再組織化します。この再組織化を使う以外に gdbm はデータベースファイルの大きさを短くすることはありません; そうでない場合、削除された部分のファイルスペースは保持され、新たな (キー、値の) ペアが追加される際に再利用されます。

`sync()`

データベースが高速モードで開かれていた場合、このメソッドはディスクにまだ書き込まれていないデータを全て書き込ませます。

参考資料:

`anydbm` モジュール (7.10 節):

`dbm` 形式のデータベースへの汎用インタフェース。

`whichdb` モジュール (7.12 節):

既存のデータベースがどの形式のデータベースか判定するユーティリティモジュール。

## 8.8 `termios` — POSIX スタイルの端末制御

このモジュールでは端末 I/O 制御のための POSIX 準拠の関数呼び出しインタフェースを提供します。これら呼び出しのための完全な記述については、POSIX または UNIX マニュアルページを参照してください。POSIX *termios* 形式の端末制御をサポートする UNIX のバージョンで (かつインストール時に指定した場合に) のみ利用可能です。

このモジュールの関数は全て、ファイル記述子 *fd* を最初の引数としてとります。この値は、`sys.stdin.fileno()` が返すような整数のファイル記述子でも、`sys.stdin` 自体のようなファイルオブジェクトでもかまいません。

このモジュールではまた、モジュールで提供されている関数を使う上で必要となる全ての定数を定義しています; これらの定数は C の対応する関数と同じ名前を持っています。これらの端末制御インタフェースを利用する上でのさらなる情報については、あなたのシステムのドキュメンテーションを参考にしてください。

このモジュールでは以下の関数を定義しています:

`tcgetattr(fd)`

ファイル記述子 *fd* の端末属性を含むリストを返します。その形式は: `[iflag, oflag, cflag, lflag, ispeed, ospeed, cc]` です。*cc* は端末特殊文字のリストです (それぞれ長さ 1 の文字列です。ただしインデクス `VMIN` および `VTIME` の内容は、それらのフィールドが定義されていた場合整数の値となります)。

端末設定フラグおよび端末速度の解釈、および配列 *cc* のインデクス検索は、`termios` で定義されているシンボル定数を使って行わなければなりません。

`tcsetattr(fd, when, attributes)`

ファイル記述子 *fd* の端末属性を *attributes* から取り出して設定します。*attributes* は `tcgetattr()` が返すようなリストです。引数 *when* は属性がいつ変更されるかを決定します: `TCSANOW` は即時変更を行い、`TCSAFLUSH` は現在キューされている出力を全て転送し、全てのキューされている入力を見捨てた後に変更を行います。

`tcsendbreak(fd, duration)`

ファイル記述子 *fd* にブレークを送信します。*duration* をゼロにすると、0.25–0.5 秒間のブレークを送信します; *duration* の値がゼロでない場合、その意味はシステム依存です。

`tcdrain(fd)`

ファイル記述子 *fd* に書き込まれた全ての出力が転送されるまで待ちます。

`tcflush(fd, queue)`

ファイル記述子 *fd* にキューされたデータを無視します。どのキューかは *queue* セレクタで指定します: `TCIFLUSH` は入力キュー、`TCOFLUSH` は出力キュー、`TCIOFLUSH` は両方のキューです。

`tcflow(fd, action)`

ファイル記述子 *fd* の入力または出力をサスペンドしたりレジュームしたりします。引数 *action* は出

力をサスペンドする TCOOFF、出力をレジュームする TCOON、入力をサスペンドする TCIOFF、入力をレジュームする TCION をとることができます。

参考資料:

tty モジュール (8.10 節):

一般的な端末制御操作のための便利な関数。

### 8.8.1 使用例

以下はエコーバックを切った状態でパスワード入力を促す関数です。ユーザの入力に関わらず以前の端末属性を正確に回復するために、二つの `tcgetattr()` と `try ... finally` 文によるテクニックが使われています:

```
def getpass(prompt = "Password: "):
    import termios, sys
    fd = sys.stdin.fileno()
    old = termios.tcgetattr(fd)
    new = termios.tcgetattr(fd)
    new[3] = new[3] & ~termios.ECHO          # lflags
    try:
        termios.tcsetattr(fd, termios.TCSADRAIN, new)
        passwd = raw_input(prompt)
    finally:
        termios.tcsetattr(fd, termios.TCSADRAIN, old)
    return passwd
```

## 8.9 TERMIO — termios モジュールで使われる定数

リリース 2.1 以降で撤廃された仕様です。必要な定数は `termios` から取り込んでください。

このモジュールは `termios` モジュールを使う上で必要となるシンボル定数を定義しています (上の節を参照してください)。これらの定数のリストは POSIX または UNIX のマニュアルページを参照してください。

## 8.10 tty — 端末制御のための関数群

`tty` モジュールは端末を `cbreak` および `raw` モードにするための関数を定義しています。

このモジュールは `termios` モジュールを必要とするため、UNIX でしか動作しません。

`tty` モジュールでは、以下の関数を定義しています:

**setraw(*fd*[, *when*])**

ファイル記述子 *fd* のモードを `raw` モードに変えます。*when* を省略すると標準の値は `TERMIO.TCAFLUSH` になり、`termios.tcsetattr()` に渡されます。

**setcbreak(*fd*[, *when*])**

ファイル記述子 *fd* のモードを `cbreak` モードに変えます。*when* を省略すると標準の値は `TERMIO.TCAFLUSH` になり、`termios.tcsetattr()` に渡されます。

参考資料:

`termios` モジュール (8.8 節):

低レベル端末制御インタフェース。

TERMIOS モジュール (8.9 節):

端末制御操作で便利な定数群。

## 8.11 pty — 擬似端末ユーティリティ

pty モジュールは擬似端末 (他のプロセスを実行してその制御をしている端末をプログラムで読み書きする) を制御する操作を定義しています。

擬似端末の制御はプラットフォームに強く依存するので、SGI と Linux 用のコードしか存在していません。(Linux 用のコードは他のプラットフォームでも動作するように作られていますがテストされていません。)

pty モジュールでは以下の関数を定義しています:

**fork()**

fork します。子プロセスの制御端末を擬似端末に接続します。戻り値は (*pid*, *fd*) です。子プロセスは *pid* として 0、*fd* として *invalid* をそれぞれ受けとります。親プロセスは *pid* として子プロセスの PID、*fd* として子プロセスの制御端末 (子プロセスの標準入出力に接続されている) のファイルディスクリプタを受けとります。

**openpty()**

新しい擬似端末のペアを開きます。利用できるなら `os.openpty()` を使い、利用できなければ SGI と一般的な UNIX システム用のエミュレーションコードを使います。マスター、スレーブそれぞれのためのファイルディスクリプタ、(*master*, *slave*) のタプルを返します。

**spawn(*argv*[, *master\_read*[, *stdin\_read*]])**

プロセスを生成して制御端末を現在のプロセスの標準入出力に接続します。これは制御端末を読もうとするプログラムをごまかすために利用されます。

*master\_read* と *stdin\_read* にはファイルディスクリプタから読み込む関数を指定してください。デフォルトでは呼ばれるたびに 1024 バイトずつ読み込むとします。

## 8.12 fcntl — fcntl() および ioctl() システムコール

このモジュールでは、ファイル記述子 (file descriptor) に基づいたファイル制御および I/O 制御を実現します。このモジュールは、UNIX のルーチンである `fcntl()` および `ioctl()` へのインタフェースです。

このモジュール内の全ての関数はファイル記述子 *fd* を最初の引数に取ります。この値は `sys.stdin.fileno()` が返すような整数のファイル記述子でも、`sys.stdin` 自体のような、純粹にファイル記述子だけを返す `fileno()` メソッドを提供しているファイルオブジェクトでもかまいません。

このモジュールでは以下の関数を定義しています:

**fcntl(*fd*, *op*[, *arg*])**

要求された操作をファイル記述子 *fd* (または `fileno()` メソッドを提供しているファイルオブジェクト) に対して実行します。操作は *op* で定義され、オペレーティングシステム依存です。これらの操作コードは `fcntl` モジュール内にもあります。引数 *arg* はオプションで、標準では整数値 0 です。この引数を与える場合、整数か文字列の値をとります。引数がないか整数値の場合、この関数の戻り値は C 言語の `fcntl()` を呼び出した際の整数の戻り値になります。引数が文字列の場合には、`struct.pack` で作られるようなバイナリの構造体を表します。バイナリデータはバッファにコピーされ、そのアドレスが C 言語の `fcntl()` 呼び出しに渡されます。呼び出しが成功した後に戻される値はバッファの内容で、文字列オブジェクトに変換されています。返される文字列は *arg* 引数と同じ長さになります。この値は 1024 バイトに制限されています。オペレーティングシステムからバッファに返される情報の長さが 1024 バイトよりも大きい場合、大抵はセグメンテーション違反となるか、

より不可思議なデータの破損を引き起こします。

`fcntl()` が失敗した場合、`IOError` が送出されます。

**`ioctl(fd, op, arg)`**

この関数は `fcntl()` 関数と同じですが、操作が通常ライブラリモジュール `termios` で定義されており、引数の扱いがより複雑であるところが異なります。

パラメタ `arg` は整数か、存在しない(整数 0 と等価なものとして扱われます) か、(通常の Python 文字列のような) 読み出し専用のバッファインタフェースをサポートするオブジェクトか、読み書きバッファインタフェースをサポートするオブジェクトです。

最後の型のオブジェクトを除き、動作は `fcntl()` 関数と同じです。

可変なバッファが渡された場合、動作は `mutate_flag` 引数の値で決定されます。

この値が偽の場合、バッファの可変性は無視され、動作は読み出しバッファの場合と同じになりますが、上で述べた 1024 バイトの制限は回避されます。従って、どれだけオペレーティングシステムが希望するより長いバッファを渡しても正しく動作します。

`mutate_flag` が真の場合、バッファは(実際には) 根底にある `ioctl()` システムコールに渡され、後者の戻り値が呼び出し側の Python に引き渡され、バッファの新たな内容は `ioctl` の動作を反映します。この説明はやや単純化されています。というのは、与えられたバッファが 1024 バイト長よりも短い場合、バッファはまず 1024 バイト長の静的なバッファにコピーされてから `ioctl` に渡され、その後引数で与えたバッファに戻しコピーされるからです。

`mutate_flag` が与えられなかった場合、2.3 ではこの値は偽となります。この仕様は今後のいくつかのバージョンを経た Python で変更される予定です: 2.4 では、`mutate_flag` を提供し忘れると警告が出されますが同じ動作を行い、2.5 ではデフォルトの値が真となるはずですが。

以下に例を示します:

```
>>> import array, fcntl, struct, termios, os
>>> os.getpgrp()
13341
>>> struct.unpack('h', fcntl.ioctl(0, termios.TIOCGPGRP, "  "))[0]
13341
>>> buf = array.array('h', [0])
>>> fcntl.ioctl(0, termios.TIOCGPGRP, buf, 1)
0
>>> buf
array('h', [13341])
```

**`flock(fd, op)`**

ファイル記述子 `fd` (`fileno()` メソッドを提供しているファイルオブジェクトも含む) に対してロック操作 `op` を実行します。詳細は UNIX マニュアルの `flock(3)` を参照してください(システムによっては、この関数は `fcntl()` を使ってエミュレーションされています)。

**`lockf(fd, operation, [len, [start, [whence]]])`**

本質的に `fcntl()` によるロッキングの呼び出しをラップしたものです。`fd` はロックまたはアンロックするファイルのファイル記述子で、`operation` は以下の値:

- `LOCK_UN` – アンロック
- `LOCK_SH` – 共有ロックを取得
- `LOCK_EX` – 排他的ロックを取得

のうちのいずれかになります。

*operation* が `LOCK_SH` または `LOCK_EX` の場合、`LOCK_NB` とビット OR にすることでロック取得時にブロックしないようにすることができます。`LOCK_NB` が使われ、ロックが取得できなかった場合、`IOError` が送出され、例外は *errno* 属性を持ち、その値は `EACCESS` または `EAGAIN` になります (オペレーティングシステムに依存します; 可搬性のため、両方の値をチェックしてください)。少なくともいくつかのシステムでは、ファイル記述子が参照しているファイルが書き込みのために開かれている場合、`LOCK_EX` だけしか使うことができません。

*length* はロックを行いたいバイト数、*start* はロック領域先頭の *whence* からの相対的なバイトオフセット、*whence* は `fileobj.seek()` と同じで、具体的には:

- 0 – ファイル先頭からの相対位置 (`SEEK_SET`)
- 1 – 現在のバッファ位置からの相対位置 (`SEEK_CUR`)
- 2 – ファイルの末尾からの相対位置 (`SEEK_END`)

*start* の標準の値は 0 で、ファイルの先頭から開始することを意味します。*whence* の標準の値も 0 です。

以下に (全ての SVR4 互換システムでの) 例を示します:

```
import struct, fcntl

file = open(...)
rv = fcntl(file, fcntl.F_SETFL, os.O_NDELAY)

lockdata = struct.pack('hhllhh', fcntl.F_WRLCK, 0, 0, 0, 0, 0)
rv = fcntl.fcntl(file, fcntl.F_SETLKW, lockdata)
```

最初の例では、戻り値 *rv* は整数値を保持しています; 二つ目の例では文字列値を保持しています。*lockdata* 変数の構造体レイアウトはシステム依存です — 従って `flock()` を呼ぶ方がベターです。

## 8.13 pipes — シェルパイプラインへのインタフェース

`pipes` モジュールでは、*'pipeline'* の概念 — あるファイルを別のファイルに変換する機構の直列接続 — を抽象化するためのクラスを定義しています。

このモジュールは `/bin/sh` コマンドラインを利用するため、`os.system()` および `os.popen()` のための POSIX 準拠のシェル、または互換のシェルが必要です。

`pipes` モジュールでは、以下のクラスを定義しています:

```
class Template()
```

パイプラインを抽象化したクラス。

使用例:

```
>>> import pipes
>>> t=pipes.Template()
>>> t.append('tr a-z A-Z', '---')
>>> f=t.open('/tmp/1', 'w')
>>> f.write('hello world')
>>> f.close()
>>> open('/tmp/1').read()
'HELLO WORLD'
```



### 8.13.1 テンプレートオブジェクト

テンプレートオブジェクトは以下のメソッドを持っています:

**reset()**

パイプラインテンプレートを初期状態に戻します。

**clone()**

元のパイプラインテンプレートと等価の新しいオブジェクトを返します。

**debug(flag)**

*flag* が真の場合、デバッグをオンにします。そうでない場合、デバッグをオフにします。デバッグがオンの時には、実行されるコマンドが印字され、より多くのメッセージを出力するようにするために、シェルに `set -x` 命令を与えます。

**append(cmd, kind)**

新たなアクションをパイプラインの末尾に追加します。*cmd* 変数は有効な bourne shell 命令でなければなりません。*kind* 変数は二つの文字からなります。

最初の文字は `'-'` (コマンドが標準入力からデータを読み出すことを意味します)、`'f'` (コマンドがコマンドライン上で与えたファイルからデータを読み出すことを意味します)、あるいは `'.'` (コマンドは入力を読まないことを意味します、従ってパイプラインの先頭になります)、のいずれかになります。

同様に、二つ目の文字は `'-'` (コマンドが標準出力に結果を書き込むことを意味します)、`'f'` (コマンドがコマンドライン上で指定したファイルに結果を書き込むことを意味します)、あるいは `'.'` (コマンドはファイルを書き込まないことを意味し、パイプラインの末尾になります)、のいずれかになります。

**prepend(cmd, kind)**

パイプラインの先頭に新しいアクションを追加します。引数の説明については `append()` を参照してください。

**open(file, mode)**

ファイル類似のオブジェクトを返します。このオブジェクトは *file* を開いていますが、パイプラインを通して読み書きするようになっています。*mode* には `'r'` または `'w'` のいずれか一つしか与えることができないので注意してください。

**copy(infile, outfile)**

パイプを通して *infile* を *outfile* にコピーします。

## 8.14 posixfile — ロック機構をサポートするファイル類似オブジェクト

リリース 1.5 以降で撤廃された仕様です。このモジュールが提供しているよりもうまく処理ができ、可搬性も高いロック操作が `fcntl.lockf()` で提供されています。

このモジュールでは、組み込みのファイルオブジェクトの上にいくつかの追加機能を実装しています。特に、このオブジェクトはファイルのロック機構、ファイルフラグへの操作、およびファイルオブジェクトを複製するための簡単なインタフェースを実装しています。オブジェクトは全ての標準ファイルオブジェクトのメソッドに加え、以下に述べるメソッドを持っています。このモジュールはファイルのロック機構に `fcntl.fcntl()` を用いるため、ある種の UNIX でしか動作しません。

`posixfile` オブジェクトをインスタンス化するには、`posixfile` モジュールの `open()` 関数を使います。生成されるオブジェクトは標準ファイルオブジェクトとだいたい同じルック&フィールです。

posixfile モジュールでは、以下の定数を定義しています:

**SEEK\_SET**

オフセットをファイルの先頭から計算します。

**SEEK\_CUR**

オフセットを現在のファイル位置から計算します。

**SEEK\_END**

オフセットをファイルの末尾から計算します。

posixfile モジュールでは以下の関数を定義しています:

**open**(*filename*[, *mode*[, *bufsize*]])

指定したファイル名とモードで新しい posixfile オブジェクトを作成します。*filename*、*mode* および *bufsize* 引数は組み込みの `open()` 関数と同じように解釈されます。

**fileopen**(*fileobject*)

指定した標準ファイルオブジェクトで新しい posixfile オブジェクトを作成します。作成されるオブジェクトは元のファイルオブジェクトと同じファイル名およびモードを持っています。

posixfile オブジェクトでは以下の追加メソッドを定義しています:

**lock**(*fmt*, [*len*[, *start*[, *whence*]]])

ファイルオブジェクトが参照しているファイルの指定部分にロックをかけます。指定の書式は下のテーブルで説明されています。*len* 引数にはロックする部分の長さを指定します。標準の値は 0 です。*start* にはロックする部分の先頭オフセットを指定し、その標準値は 0 です。*whence* 引数はオフセットをどこからの相対位置にするかを指定します。この値は定数 `SEEK_SET`、`SEEK_CUR`、または `SEEK_END` のいずれかになります。標準の値は `SEEK_SET` です。引数についてのより詳しい情報はシステムの `fcntl(2)` マニュアルページを参照してください。

**flags**( [*flags* ])

ファイルオブジェクトが参照しているファイルに指定したフラグを設定します。新しいフラグは特に指定しない限り以前のフラグと OR されます。指定書式は下のテーブルで説明されています。*flags* 引数なしの場合、現在のフラグを示す文字列が返されます(‘?’ 修飾子と同じです)。フラグについてのより詳しい情報はシステムの `fcntl(2)` マニュアルページを参照してください。

**dup**( )

ファイルオブジェクトと、背後のファイルポインタおよびファイル記述子を複製します。返されるオブジェクトは新たに開かれたファイルのように振舞います。

**dup2**(*fd*)

ファイルオブジェクトと、背後のファイルポインタおよびファイル記述子を複製します。新たなオブジェクトは指定したファイル記述子を持ちます。それ以外の点では、返されるオブジェクトは新たに開かれたファイルのように振舞います。

**file**( )

posixfile オブジェクトが参照している標準ファイルオブジェクトを返します。この関数は標準ファイルオブジェクトを使うよう強制している関数を使う場合に便利です。

全てのメソッドで、要求された操作が失敗した場合には `IOError` が送出されます。

`lock()` の書式指定文字には以下のような意味があります:

書式指定	意味
‘u’	指定領域のロックを解除します
‘r’	指定領域の読み出しロックを要求します
‘w’	指定領域の書き込みロックを要求します

これに加え、以下の修飾子を書式に追加できます:

修飾子	意味	注釈
' '	ロック操作が処理されるまで待ちます	
'?'	要求されたロックと衝突している第一のロックを返すか、衝突がない場合には <code>None</code> を返します。	(1)

注釈:

- (1) 返されるロックは `(mode, len, start, whence, pid)` の形式で、`mode` はロックの形式を表す文字 ('r' または 'w') です。この修飾子はロック要求の許可を行わせません; すなわち、問い合わせの目的にしか使えません。

`flags()` の書式指定文字には以下のような意味があります:

書式	意味
'a'	追記のみ (append only) フラグ
'c'	実行時クローズ (close on exec) フラグ
'n'	無遅延 (no delay) フラグ (非ブロック (non-blocking) フラグとも呼ばれます)
's'	同期 (synchronization) フラグ

これに加え、以下の修飾子を書式に追加できます:

修飾子	意味	注釈
'!'	指定したフラグを通常の 'オン' にせず 'オフ' にします	(1)
'='	フラグを標準の 'OR' 操作ではなく置換します。	(1)
'?'	設定されているフラグを表現する文字からなる文字列を返します。	(2)

注釈:

- (1) '!' および '=' 修飾子は互いに排他的の関係にあります。
- (2) この文字列が表すフラグは同じ呼び出しによってフラグが置き換えられた後のものです。

以下に例を示します:

```
import posixfile

file = posixfile.open('/tmp/test', 'w')
file.lock('w|')
...
file.lock('u')
file.close()
```

## 8.15 resource — リソース使用状態の情報

このモジュールでは、プログラムによって使用されているシステムリソースを計測したり制御するための基本的なメカニズムを提供します。

特定のシステムリソースを指定したり、現在のプロセスやその子プロセスのリソース使用情報を要求するためにはシンボル定数が使われます。

エラーを表すための例外が一つ定義されています:

## exception error

下に述べる関数は、背後にあるシステムコールが予期せず失敗した場合、このエラーを送出するかもしれません。

### 8.15.1 リソースの制限

リソースの使用は下に述べる `setrlimit()` 関数を使って制限することができます。各リソースは二つ組の制限値: ソフトリミット (soft limit)、およびハードリミット (hard limit)、で制御されます。ソフトリミットは現在の制限値で、時間とともにプロセスによって下げたり上げたりできます。ソフトリミットはハードリミットを超えることはできません。ハードリミットはソフトリミットよりも高い任意の値まで下げることができますが、上げることはできません。(スーパーユーザの有効な UID を持つプロセスのみがハードリミットを上げることができます。)

制限をかけるべく指定できるリソースはシステムに依存します。指定できるリソースは `getrlimit(2)` マニュアルページで解説されています。以下に列挙するリソースは背後のオペレーティングシステムがサポートする場合にサポートされています; オペレーティングシステム側で値を調べたり制御したりできないリソースは、そのプラットフォーム向けのこのモジュール内では定義されていません。

#### `getrlimit(resource)`

*resource* の現在のソフトおよびハードリミットを表すタプル (*soft*, *hard*) を返します。無効なリソースが指定された場合には `ValueError` が、背後のシステムコールが予期せず失敗した場合には `error` が送出されます。

#### `setrlimit(resource, limits)`

*resource* の新たな消費制限を設定します。*limits* 引数には、タプル (*soft*, *hard*) による二つの整数で、新たな制限を記述しなければなりません。現在指定可能な最大の制限を指定するために -1 を使うことができます。

無効なリソースが指定された場合、ソフトリミットの値がハードリミットの値を超えている場合、プロセスが (スーパーユーザの有効な UID を持っていない状態で) ハードリミットを引き上げようとした場合には `ValueError` が送出されます。背後のシステムコールが予期せず失敗した場合には `error` が送出される可能性もあります。

以下のシンボルは、後に述べる関数 `setrlimit()` および `getrlimit()` を使って消費量を制御することができるリソースを定義しています。これらのシンボルの値は、C プログラムで使われているシンボルと全く同じです。

`getrlimit(2)` の UNIX マニュアルページには、指定可能なリソースが列挙されています。全てのシステムで同じシンボルが使われているわけではなく、また同じリソースを表すために同じ値が使われているとも限らないので注意してください。このモジュールはプラットフォーム間の相違を隠蔽しようとはしていません — あるプラットフォームで定義されていないシンボルは、そのプラットフォーム向けの本モジュールでは利用することができません。

#### `RLIMIT_CORE`

現在のプロセスが生成できるコアファイルの最大 (バイト) サイズです。プロセスの全体イメージを入れるためにこの値より大きなサイズのコアファイルが要求された結果、部分的なコアファイルが生成される可能性があります。

#### `RLIMIT_CPU`

プロセッサが利用することができる最大プロセッサ時間 (秒) です。この制限を超えた場合、`SIGXCPU` シグナルがプロセスに送られます。(どのようにしてシグナルを捕捉したり、例えば開かれているファイルをディスクにフラッシュするといった有用な処理を行うかについての情報は、`signal` モジュールのドキュメントを参照してください)

#### RLIMIT\_FSIZE

プロセスが生成できるファイルの最大サイズです。マルチスレッドプロセスの場合、この値は主スレッドのスタックにのみ影響します。

#### RLIMIT\_DATA

プロセスのヒープの最大 (バイト) サイズです。

#### RLIMIT\_STACK

現在のプロセスのコールスタックの最大 (バイト) サイズです。

#### RLIMIT\_RSS

プロセスが取りうる最大 RAM 常駐ページサイズ (resident set size) です。

#### RLIMIT\_NPROC

現在のプロセスが生成できるプロセスの上限です。

#### RLIMIT\_NOFILE

現在のプロセスが開けるファイル記述子の上限です。

#### RLIMIT\_OFILE

RLIMIT\_NOFILE の BSD での名称です。

#### RLIMIT\_MEMLOCK

メモリ中でロックできる最大アドレス空間です。

#### RLIMIT\_VMEM

プロセスが占有できるマップメモリの最大領域です。

#### RLIMIT\_AS

アドレス空間でプロセスが占有できる最大領域 (バイト) です。

## 8.15.2 リソースの使用状態

以下の関数はリソース使用情報を取得するために使われます:

#### `getrusage(who)`

この関数は、*who* 引数で指定される、現プロセスおよびその子プロセスによって消費されているリソースを記述するオブジェクトを返します。*who* 引数は以下に記述される `RUSAGE_*` 定数のいずれかを使って指定します。

返される値の各フィールドはそれぞれ、個々のシステムリソースがどれくらい使用されているか、例えばユーザモードでの実行に費やされた時間やプロセスが主記憶からスワップアウトされた回数、を示しています。幾つかの値、例えばプロセスが使用しているメモリ量は、内部時計の最小単位に依存します。

以前のバージョンとの互換性のため、返される値は 16 要素からなるタプルとしてアクセスすることもできます。

戻り値のフィールド `ru_utime` および `ru_stime` は浮動小数点数で、それぞれユーザモードでの実行に費やされた時間、およびシステムモードでの実行に費やされた時間を表します。それ以外の値は整数です。これらの値に関する詳しい情報は `getrusage(2)` を調べてください。以下に簡単な概要を示します:

インデクス	フィールド名	リソース
0	ru_utime	ユーザモード実行時間 (float)
1	ru_stime	システムモード実行時間 (float)
2	ru_maxrss	最大常駐ページサイズ
3	ru_ixrss	共有メモリサイズ
4	ru_idrss	非共有メモリサイズ
5	ru_isrss	非共有スタックサイズ
6	ru_minflt	I/O を必要とするページフォールト数
7	ru_majflt	I/O を必要としないページフォールト数
8	ru_nswap	スワップアウト回数
9	ru_inblock	ブロック入力操作数
10	ru_oublock	ブロック出力操作数
11	ru_msgsnd	送信メッセージ数
12	ru_msgrcv	受信メッセージ数
13	ru_nsignals	受信シグナル数
14	ru_nvcsw	自発的な実行コンテキスト切り替え数
15	ru_nivcsw	非自発的な実行コンテキスト切り替え数

この関数は無効な *who* 引数を指定した場合には `ValueError` を送出します。また、異常が発生した場合には `error` 例外が送出される可能性があります。

2.3 で変更された仕様: 各値を返されたオブジェクトの属性としてアクセスできるようにしました

#### `getpagesize()`

システムページ内のバイト数を返します。(ハードウェアページサイズと同じとは限りません。) この関数はプロセスが使用しているメモリのバイト数を決定する上で有効です。 `getrusage()` が返すタプルの 3 つ目の要素はページ数で数えたメモリ使用量です; ページサイズを掛けるとバイト数になります。

以下の `RUSAGE_*` シンボルはどのプロセスの情報を提供させるかを指定するために関数 `getrusage()` に渡されます。

#### `RUSAGE_SELF`

`RUSAGE_SELF` はプロセス自体に属する情報を要求するために使われます。

#### `RUSAGE_CHILDREN`

`getrusage()` に渡すと呼び出し側プロセスの子プロセスのリソース情報を要求します。

#### `RUSAGE_BOTH`

`getrusage()` に渡すと現在のプロセスおよび子プロセスの両方が消費しているリソースを要求します。全てのシステムで利用可能なわけではありません。

## 8.16 nis — Sun の NIS (Yellow Pages) へのインタフェース

`nis` モジュールは複数のホストを集中管理する上で便利な NIS ライブラリを薄くラップします。

NIS は UNIX システム上にしかないので、このモジュールは UNIX でしか利用できません。

`nis` モジュールでは以下の関数を定義しています:

#### `match(key, mapname)`

`mapname` 中で `key` に一致するものを返すか、見つからない場合にはエラー (`nis.error`) を送出します。両方の引数とも文字列で、`key` は 8 ビットクリーンです。返される値は (`NULL` その他を含む可能性のある) 任意のバイト配列です。



*mapname* は他の名前の別名になっていないか最初にチェックされます。

**cat**(*mapname*)

*match(key, mapname) == value* となる *key* を *value* に対応付ける辞書を返します。辞書内のキーと値は共に任意のバイト列なので注意してください。

*mapname* は他の名前の別名になっていないか最初にチェックされます。

**maps**()

有効なマップのリストを返します。

*nis* モジュールは以下の例外を定義しています:

**exception error**

NIS 関数がエラーコードを返した場合に送出されます。

## 8.17 syslog — UNIX syslog ライブラリルーチン群

このモジュールでは UNIX syslog ライブラリルーチン群へのインタフェースを提供します。syslog の便宜レベルに関する詳細な記述は UNIX マニュアルページを参照してください。

このモジュールでは以下の関数を定義しています:

**syslog**( [*priority*, ] *message* )

文字列 *message* をシステムログ機構に送信します。末尾の改行文字は必要に応じて追加されます。各メッセージは *facility* および *level* からなる優先度でタグ付けされます。オプションの *priority* 引数はメッセージの優先度を定義します。標準の値は LOG\_INFO です。priority 中に、便宜レベルが (LOG\_INFO | LOG\_USER のように) 論理和を使ってコード化されていない場合、openlog() を呼び出した際の値が使われます。

**openlog**( *ident* [, *logopt* [, *facility* ] ] )

標準以外のログオプションは、syslog() の呼び出しに先立って openlog() でログファイルを開く際、明示的に設定することができます。標準の値は (通常) *ident* = 'syslog'、*logopt* = 0、*facility* = LOG\_USER です。*ident* 引数は全てのメッセージの先頭に付加する文字列です。オプションの *logopt* 引数はビットフィールドの値になります - とりうる組み合わせ値については以下を参照してください。オプションの *facility* 引数は、便宜レベルコードの設定が明示的になされていないメッセージに対する、標準の便宜レベルを設定します。

**closelog**()

ログファイルを閉じます。

**setlogmask**(*maskpri*)

優先度マスクを *maskpri* に設定し、以前のマスク値を返します。*maskpri* に設定されていない優先度レベルを持った syslog() の呼び出しは無視されます。標準では全ての優先度をログ出力します。関数 LOG\_MASK(*pri*) は個々の優先度 *pri* に対する優先度マスクを計算します。関数 LOG\_UPTO(*pri*) は優先度 *pri* までの全ての優先度を含むようなマスクを計算します。

このモジュールでは以下の定数を定義しています:

優先度 (高い優先度順): LOG\_EMERG、 LOG\_ALERT、 LOG\_CRIT、 LOG\_ERR、 LOG\_WARNING、 LOG\_NOTICE、 LOG\_INFO、 LOG\_DEBUG。

便宜レベル: LOG\_KERN、 LOG\_USER、 LOG\_MAIL、 LOG\_DAEMON、 LOG\_AUTH、 LOG\_LPR、 LOG\_NEWS、 LOG\_UUCP、 LOG\_CRON、 および LOG\_LOCAL0 から LOG\_LOCAL7。

ログオプション: <syslog.h> で定義されている場合、LOG\_PID、 LOG\_CONS、 LOG\_NDELAY、 LOG\_NOWAIT、 および LOG\_PERROR。

## 8.18 commands — コマンド実行ユーティリティ

`commands` は、システムへコマンド文字列を渡して実行する `os.popen()` のラッパー関数を含んでいるモジュールです。外部で実行したコマンドの結果や、その終了ステータスを扱います。

`commands` モジュールは以下の関数を定義しています。

**`getstatusoutput(cmd)`**

文字列 `cmd` を `os.popen()` を使いシェル上で実行し、タプル (`status`, `output`) を返します。実際には `{cmd ; }2>&1` と実行されるため、標準出力とエラー出力が混合されます。また、出力の最後の改行文字は取り除かれます。コマンドの終了ステータスは C 言語関数の `wait()` の規則に従って解釈することができます。

**`getoutput(cmd)`**

`getstatusoutput()` に似ていますが、終了ステータスは無視され、コマンドの出力のみを返します。

**`getstatus(file)`**

`'ls -ld file'` の出力を文字列で返します。この関数は `getoutput()` を使い、引数内のバックスラッシュ記号「\」とドル記号「\$」を適切にエスケープします。

例:

```
>>> import commands
>>> commands.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> commands.getstatusoutput('cat /bin/junk')
(256, 'cat: /bin/junk: No such file or directory')
>>> commands.getstatusoutput('/bin/junk')
(256, 'sh: /bin/junk: not found')
>>> commands.getoutput('ls /bin/ls')
'/bin/ls'
>>> commands.getstatus('/bin/ls')
'-rwxr-xr-x 1 root          13352 Oct 14  1994 /bin/ls'
```

## Python デバッガ

モジュール `pdb` は Python プログラム用の対話的ソースコードデバッガを定義します。(条件付き) ブレークポイントの設定やソース行レベルでのシングルステップ実行、スタックフレームのインスペクション、ソースコードリスティングおよびいかなるスタックフレームのコンテキストにおける任意の Python コードの評価をサポートしています。事後解析デバッグもサポートし、プログラムの制御下で呼び出すことができます。

デバッガは拡張可能です — 実際にはクラス `Pdb` として定義されています。現在これについてのドキュメントはありませんが、ソースを読めば簡単に理解できます。拡張インターフェースはモジュール `bdb` (ドキュメントなし) と `cmd` を使っています。

デバッガのプロンプトは `'(Pdb) '` です。デバッガに制御された状態でプログラムを実行するための典型的な使い方は:

```
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
> <string>(0)?()
(Pdb) continue
> <string>(1)?()
(Pdb) continue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

他のスクリプトをデバッグするために、`'pdb.py'` をスクリプトとして呼び出すこともできます。例えば:

```
python /usr/local/lib/python1.5/pdb.py myscript.py
```

クラッシュしたプログラムを調べるための典型的な使い方は:

```
>>> import pdb
>>> import mymodule
>>> mymodule.test()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "./mymodule.py", line 4, in test
    test2()
  File "./mymodule.py", line 3, in test2
    print spam
NameError: spam
>>> pdb.pm()
> ./mymodule.py(3)test2()
-> print spam
(Pdb)
```

モジュールは以下の関数を定義しています。それぞれが少しずつ違った方法でデバッガに入ります:

**run**(*statement*[, *globals*[, *locals*]])

デバッガに制御された状態で (文字列として与えられた)*statement* を実行します。デバッガプロンプトはあらゆるコードが実行される前に現れます。ブレークポイントを設定し、`'continue'` とタイプできます。あるいは、文を `'step'` や `'next'` を使って一つずつ実行することができます (これらのコマンドはすべて下で説明します)。オプションの *globals* と *locals* 引数はコードを実行する環境を指定します。デフォルトでは、モジュール `__main__` の辞書が使われます。(exec 文または `eval()` 組み込み関数の説明を参照してください。)

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

デバッガの制御もとで (文字列として与えられる)*expression* を評価します。`runeval()` がリターンしたとき、式の値を返します。その他の点では、この関数は `run()` を同様です。

**runcall**(*function*[, *argument*, ...])

*function*(関数またはメソッドオブジェクト、文字列ではありません) を与えられた引数とともに呼び出します。`runcall()` がリターンしたとき、関数呼び出しが返したものは何でも返します。デバッガプロンプトは関数に入るとすぐに現れます。

**set\_trace**()

スタックフレームを呼び出したところでデバッガに入ります。たとえコードが別の方法でデバッグされている最中でなくても (例えば、アサーションが失敗するとき)、これはプログラムの所定の場所でブレークポイントをハードコードするために役に立ちます。

**post\_mortem**(*traceback*)

与えられた *traceback* オブジェクトの事後解析デバッグングに入ります。

**pm**()

`sys.last_traceback` のトレースバックの事後解析デバッグングに入ります。

## 9.1 デバッガコマンド

デバッガは以下のコマンドを認識します。ほとんどのコマンドは一文字または二文字に省略することができます。例えば、`'h(elp)'` が意味するのは、ヘルプコマンドを入力するために `'h'` か `'help'` のどちらか一方を使うことができるということです (が、`'he'` や `'hel'` は使えず、また `'H'` や `'Help'`、`'HELP'` も使えません)。コマンドの引数は空白 (スペースまたはタブ) で区切られなければなりません。オプションの引数はコマンド構文の角括弧 (`'[]'`) の中に入れられなければなりません。角括弧をタイプしてはいけません。コマンド構文における選択肢は垂直バー (`'|'`) で区切られます。

空行を入力すると入力された直前のコマンドを繰り返します。例外: 直前のコマンドが `'list'` コマンドならば、次の 11 行がリストされます。

デバッガが認識しないコマンドは Python 文とみなして、デバッグしているプログラムのコンテキストにおいて実行されます。Python 文は感嘆符 (`'!'`) を前に付けることもできます。これはデバッグ中のプログラムを調査する強力な方法です。変数を変更したり関数を呼び出したりすることさえ可能です。このような文で例外が発生した場合には例外名がプリントされますが、デバッガの状態は変化しません。

複数のコマンドを  `';'`  で区切って一行で入力することができます。(一つだけの  `';'`  は使われません。なぜなら、Python パーサへ渡される行内の複数のコマンドのための分離記号だからです。) コマンドを分割するために何も知的なことはしていません。たとえ引用文字列の途中であっても、入力は最初の  `';'`  対で分割されます。

デバッガはエイリアスをサポートします。エイリアスはパラメータを持つことができ、調査中のコンテキストに対して人がある程度柔軟に対応できます。

ファイル  `'.pdbrc'`  はユーザのホームディレクトリか、またはカレントディレクトリにあります。それはまるでデバッガのプロンプトでタイプしたかのように読み込まれて実行されます。これは特にエイリアスのために便利です。両方のファイルが存在する場合、ホームディレクトリのものが最初に読まれ、そこに定義されているエイリアスはローカルファイルにより上書きされることがあります。

**h(elp)** [*command*] 引数なしでは、利用できるコマンドの一覧をプリントします。引数として *command* がある場合は、そのコマンドについてのヘルプをプリントします。`'help pdb'` は完全ドキュメンテーションファイルを表示します。環境変数 `PAGER` が定義されているならば、代わりにファイルはそのコマンドへパイプされます。*command* 引数が識別子でなければならないので、`'!'` コマンドについてのヘルプを得るためには `'help exec'` と入力しなければならない。

**w(here)** スタックの底にある最も新しいフレームと一緒にスタックトレースをプリントします。矢印はカレントフレームを指し、それがほとんどのコマンドのコンテキストを決定します。

**d(own)** (より新しいフレームに向かって) スタックトレース内でカレントフレームを一レベル下げます。

**u(p)** (より古いフレームに向かって) スタックトレース内でカレントフレームを一レベル上げます。

**b(reak)** [[*filename:*] *lineno* | *function* [, *condition* ]] *lineno* 引数がある場合は、現在のファイルのその場所にブレークポイントを設定します。*function* 引数がある場合は、その関数の中の最初の実行可能文にブレークポイントを設定します。別のファイル(まだロードされていないかもしれないもの)のブレークポイントを指定するために、行番号はファイル名とコロンをともに先頭に付けられます。ファイルは `sys.path` にそって検索されます。各ブレークポイントは番号を割り当てられ、その番号を他のすべてのブレークポイントコマンドが参照することに注意してください。

If a second argument is present, it is an expression which must evaluate to true before the breakpoint is honored. 二番目の引数がある場合は、それはブレークポイントが honor(???) される前に真と評価しなければならない式です。

引数なしの場合は、それぞれのブレークポイントに対して、そのブレークポイントに行き当たった回数、現在の通過カウント (ignore count) と、もしあれば関連条件を含めてすべてのブレークポイントをリストします。

**tbreak** [[*filename:*] *lineno* | *function* [, *condition* ]] 一時的なブレークポイントで、最初にそこに達したときに自動的に取り除かれます。引数は `break` と同じです。

**cl(ear)** [*bpnumber* [*bpnumber* ... ]] スペースで区切られたブレークポイントナンバーのリストを与えると、それらのブレークポイントを解除します。引数なしの場合は、すべてのブレークポイントを解除します (が、はじめに確認します)。

**disable** [*bpnumber* [*bpnumber* ...]] スペースで区切られたブレークポイントナンバーのリストとして与えられるブレークポイントを無効にします。ブレークポイントを無効にすると、プログラムの実行を止めることができなくなりますが、ブレークポイントの解除と違いブレークポイントのリストに残ったままになり、(再び)有効にすることができます。

**enable** [*bpnumber* [*bpnumber* ...]] 指定したブレークポイントを有効にします。

**ignore bpnumber** [*count*] 与えられたブレークポイントナンバーに通過カウントを設定します。count が省略されると、通過カウントは 0 に設定されます。通過カウントがゼロになったとき、ブレークポイントが機能する状態になります。ゼロでないときは、そのブレークポイントが無効にされず、どんな関連条件も真に評価されていて、ブレークポイントに来るたびに count が減らされます。

**condition bpnumber** [*condition*] condition はブレークポイントが honored(???) する前に真に評価しなければならない式です。condition がない場合は、どんな既存の条件も取り除かれます。すなわち、ブレークポイントに条件がありません。

**s(step)** 現在の行を実行し、最初に実行可能なものがあらわれたときに(呼び出された関数の)中か、現在の関数の次の行で) 停止します。

**n(ext)** 現在の関数の次の行に達するか、あるいは関数が返るまで実行を継続します。(‘next’ と ‘step’ の差は ‘step’ が呼び出された関数の内部で停止するのに対し、‘next’ は呼び出された関数を (ほぼ) 全速力で実行し、現在の関数内の次の行で停止するだけです。

**r(eturn)** 現在の関数が返るまで実行を継続します。

**c(ontinue)** ブレークポイントに出会うまで、実行を継続します。

**j(ump) lineno** 次に実行する行を指定します。最も底のフレーム中でのみ実行可能です。前に戻って実行したり、不要な部分をスキップして先の処理を実行する場合に使用します。

ジャンプには制限があり、例えば for ループの中には飛び込めませんし、finally 節の外にも飛び事ができません。

**l(ist)** [*first* [, *last*]] 現在のファイルのソースコードをリスト表示します。引数なしの場合は、現在の行の周囲を 11 行リストするか、または前のリストの続きを表示します。引数がある場合は、その行の周囲を 11 行表示します。引数が二つの場合は、与えられた範囲をリスト表示します。第二引数が第一引数より小さいときは、カウントと解釈されます。

**a(rgs)** 現在の関数の引数リストをプリントします。

**p expression** 現在のコンテキストにおいて *expression* を評価し、その値をプリントします。(注意: ‘print’ も使うことができますが、デバッガコマンドではありません — これは Python の print 文を実行します。)

**pp expression** pprint モジュールを使って例外の値が整形されることを除いて ‘p’ コマンドと同様です。

**alias** [*name* [*command*]] *name* という名前の *command* を実行するエイリアスを作成します。コマンドは引用符で囲まれてはいけません。入れ替え可能なパラメータは ‘%1’、‘%2’ など指し示され、さらに ‘%\*’ は全パラメータに置き換えられます。コマンドが与えられなければ、*name* に対する現在のエイリアスを表示します。引数が与えられなければ、すべてのエイリアスがリストされます。

エイリアスは入れ子になってもよく、pdb プロンプトで合法的にタイプできるどんなものでも含めることができます。内部 pdb コマンドをエイリアスによって上書きすることができます。そのとき、このようなコマンドはエイリアスが取り除かれるまで隠されます。エイリアス化はコマンド行の最初の語へ再帰的に適用されます。行の他のすべての語はそのままです。



例として、二つの便利なエイリアスがあります (特に `.pdbrc` ファイルに置かれたときに):

```
#Print instance variables (usage "pi classInst")
alias pi for k in %1.__dict__.keys(): print "%1.",k,"=",%1.__dict__[k]
#Print instance variables in self
alias ps pi self
```

`unalias name` 指定したエイリアスを削除します。

**[!]***statement* 現在のスタックフレームのコンテキストにおいて (一行の)*statement* を実行します。文の最初の語がデバッグコマンドと共通でない場合は、感嘆符を省略することができます。グローバル変数を設定するために、同じ行に `'global'` コマンドとともに代入コマンドの前に付けることができます。

```
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

`q(uit)` デバッグを終了します。実行しているプログラムは中断されます。

## 9.2 どのように動作しているか

いくつかの変更がインタプリタへ加えられました:

- `sys.settrace(func)` がグローバルトレース関数を設定します
- そこで、ローカルトレース関数を使うこともできます (後ろを参照)

トレース関数は三つの引数を持ちます: *frame*、*event* および *arg*。 *frame* は現在のスタックフレームです。 *event* は文字列です: `'call'`、`'line'`、`'return'` または `'exception'`。 *arg* はイベント型に依存します。

新しいローカルスコープに入ったときはいつでも、グローバルトレース関数が (`'call'` に設定された *event* とともに) 呼び出されます。そのスコープで用いられるローカルトレース関数への参照を返すか、またはスコープがトレースされるべきでないならば `None` を返します。

ローカルトレース関数はそれ自身への (あるいは、さらにそのスコープ内でさらにトレースを行うための他の関数への) 参照を返します。または、そのスコープにおけるトレースを停止させるために `None` を返します。

トレース関数としてインスタンスメソッドが受け入れられます (また、とても便利です)。

イベントは以下のような意味を持ちます:

`'call'` 関数が呼び出されます (または、他のコードブロックに入ります)。グローバルトレース関数が呼び出されます。 *arg* は `None` です。戻り値はローカルトレース関数を指定します。

`'line'` インタプリタがコードの新しい行を実行しようとしているところです (ときどき、一行に複数行イベントが存在します)。ローカルトレース関数が呼び出されます。 *arg* は `None` です。戻り値は新しいローカルトレース関数を指定します。

`'return'` 関数 (または、コードブロック) が返ろうとしているところです。ローカルトレース関数が呼び出されます。 *arg* は返るであろう値です。トレース関数の戻り値は無視されます。

**'exception'** 例外が生じています。ローカルトレース関数が呼び出されます。*arg* は三要素の (*exception*, *value*, *traceback*) です。戻り値は新しいローカルトレース関数を指定します。

例外が一連の呼び出し元を伝えられて行くときに、**'exception'** イベントは各レベルで生成されることに注意してください。

コードとフレームオブジェクトについてさらに情報を得るには、*Python Reference Manual* を参照してください。

# Python プロファイラ

Copyright © 1994, by InfoSeek Corporation, all rights reserved.

Written by James Roskind.<sup>1</sup>

Permission to use, copy, modify, and distribute this Python software and its associated documentation for any purpose (subject to the restriction in the following sentence) without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of InfoSeek not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. This permission is explicitly restricted to the copying and modification of the software to remain in Python, compiled Python, or other languages (such as C) wherein the modified or derived code is exclusively imported into a Python module.

INFOSEEK CORPORATION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL INFOSEEK CORPORATION BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

このプロファイラは私が Python プログラミングを始めてからわずか 3 週間後に書いたものです。その結果、稚拙なコードが出来上がってしまったのかもしれませんが、なにせ私はまだ初心者なのでそれもよくわかりません :-) コードはプロファイリングにふさわしいスピードを実現することに心血を注ぎました。しかし部分的な繰り返しを避けたため、かなり不格好になってしまったところがあります。改善のための意見があれば、ぜひ [jar@netscape.com](mailto:jar@netscape.com) までメールをください。サポートの 約束はできませんが... フィードバックへの感謝だけは確実にいたします。

## 10.1 プロファイラとは

プロファイラとは、プログラム実行時の様々な状態を得ることにより、その実行効率を調べるためのプログラムです。ここで解説するのは、`profile` と `pstats` モジュールが提供するプロファイラ機能についてです。このプロファイラはどの Python プログラムに対しても決定論的プロファイリングをおこないます。また、プロファイルの結果検証をす早くおこなえるよう、レポート生成用のツールも提供されています。

## 10.2 旧バージョンのプロファイラとの違い

(この節は歴史的資料としてのみ意味を持っています。ここで述べている旧バージョンのプロファイラとは Python 1.1. 以前のものを指しています)

<sup>1</sup> アップデートと  $\text{\LaTeX}$  への変換は Guido van Rossum によるもの。テキスト中に古いプロファイラのリファレンスも残してありますが、そのコードはもう含まれていません。

旧バージョンのプロファイリングモジュールとの大きな違いは、より少ないCPU 時間で、より多くの情報が得られるようになったことです。CPU 時間と情報量のトレードオフではなく、トレードアップを実現したのです。

主な内容は次の通りです。

バグ修正: ローカル・スタック・フレームの扱いに関する不具合を修正し、関数の実行時間を正しく計上するようにしました。

正確さの向上: プロファイラ自体の実行時間をユーザコード側に計上してしまうことがなくなりました。プラットフォーム毎のキャリブレーション (補正) をサポートし、プロファイラがプロファイル中にファイルの読み込みをおこなわないようにしました (当然その時間をユーザコードのものとして計上することなくなりました)。

スピードアップ: 2 つ以上の (たぶん 5 つ) の点を改善した結果、CPU の負荷が減りました。プロファイリング中はレポート生成用モジュール (pstats) を使う必要がないため、軽いプロファイラモジュールだけを常時ロードするようにしました。

再帰的な関数のサポート: 再帰エントリのカウントにより、再帰関数内で処理に費やされる時間が正確に計算されるようになりました。

レポート生成ユーザインターフェースの大幅な改善: 統計データを読み込む関数は任意の数のファイル名のリストを受け取り、独立した複数のプロファイル結果を合わせて総合的なレポートが作成できるようになりました。ソートの基準はキーワードで指定できるようになりました (4 つの整数オプションを除く)。レポートはどのプロファイル・ファイルが参照されたかと同様に、どの関数がプロファイルされたかを示すようになりました。そのほか出力形式は改善もおこなわれています。

## 10.3 インスタント・ユーザ・マニュアル

この節は“マニュアルなんか読みたいくない人”のために書かれています。ここではきわめて簡単な概要説明とアプリケーションのプロファイリングを手っとり早くおこなう方法だけを解説します。

main エントリにある関数 `foo()` をプロファイルしたいとき、モジュールに次の内容を追加します。

```
import profile
profile.run('foo()')
```

このように書くことで `foo()` を実行すると同時に一連の情報 (プロファイル) が表示されます。この方法はインタプリタ上で作業をしている場合、最も便利なやり方です。プロファイルの結果をファイルに残し、後で検証したいときは、`run()` の 2 番目の引数にファイル名を指定します。

```
import profile
profile.run('foo()', 'fooprof')
```

スクリプトファイル `'profile.py'` を使って、別のスクリプトをプロファイルすることも可能です。次のように実行します。

```
python /usr/local/lib/python1.5/profile.py myscript.py
```

プロファイル内容を確認するときは、`pstats` モジュールのメソッドを使用します。統計データの読み込みは次のようにします。

```
import pstats
p = pstats.Stats('fooprof')
```

Stats クラス (上記コードはこのクラスのインスタンスを生成するだけの内容です) は 'p' に読み込まれたデータを操作したり、表示するための各種メソッドを備えています。先に `profile.run()` を実行したとき表示された内容と同じものは、3 つのメソッド・コールにより実現できます。

```
p.strip_dirs().sort_stats(-1).print_stats()
```

最初のメソッドはモジュール名からファイル名の前に付いているパス部分を取り除きます。2 番目のメソッドはエントリをモジュール名/行番号/名前にもとづいてソートします (旧プロファイラとの構文上の互換性機能)。3 番目のメソッドで全ての統計情報を出力します。次のようなソート・メソッドも使えます。

```
p.sort_stats('name')
p.print_stats()
```

最初の行ではリストを関数名でソートしています。2 号目で情報を出力しています。さらに次の内容も試してください。

```
p.sort_stats('cumulative').print_stats(10)
```

このようにすると、関数が消費した累計時間でソートされ、さらにその上位 10 件だけを表示します。どのアルゴリズムが時間を多く消費しているのか知りたいときは、この方法が役に立つはずです。

ループで多くの時間を消費している関数はどれか調べたいときは、次のようにします。

```
p.sort_stats('time').print_stats(10)
```

上記は関数の実行で消費した時間でソートされ、上位 10 個の関数の情報が表示されます。次の内容も試してください。

```
p.sort_stats('file').print_stats('__init__')
```

このようにするとファイル名でソートされ、そのうちクラスの初期化メソッド (メソッド名 '`__init__`') に関する統計情報だけが表示されます。

```
p.sort_stats('time', 'cum').print_stats(.5, 'init')
```

上記は情報を時間 (time) をプライマリ・キー、累計時間 (cumulative time) をセカンダリ・キーにしてソートした後でさらに条件を絞って統計情報を出力します。'.5' は上位 50% だけの選択を意味し、さらにその中から文字列 `init` を含むものだけが表示されます。

どの関数がどの関数を呼び出しているのかを知りたいければ、次のようにします ('p' は最後に実行したときの状態でソートされています)。

```
p.print_callers(.5, 'init')
```

このようにすると、各関数ごとの呼出し側関数の一覧が得られます。

さらに詳しい機能を知りたいければマニュアルを読むか、次の関数の実行結果から内容を推察してください。

```
p.print_callees()
p.add('fooprof')
```

スクリプトとして起動した場合、`pstats` モジュールはプロファイルのダンプを読み込み、分析するための統計ブラウザとして動きます。シンプルな行指向のインターフェース (`cmd` を使って実装) とヘルプ機能を備えています。

## 10.4 決定論的プロファイリングとは何か？

決定論的プロファイリングとは、すべての関数呼出し、関数からのリターン、例外発生をモニターし、正確なタイミングを記録することで、イベント間の時間、つまりどの時間にユーザ・コードが実行されているのかを計測するやり方です。もう一方の統計学的プロファイリング(このモジュールでこの方法は採用していません)とは、有効なインストラクション・ポインタからランダムにサンプリングをおこない、プログラムのどこで時間が使われているかを推定する方法です。後者の方法は、オーバーヘッドが少いものの、プログラムのどこで多くの時間が使われているか、その相対的な示唆に留まります。

Python の場合、実行中必ずインタプリタが動作するため、決定論的プロファイリングをおこなうにあたり、計測用のコードは必須ではありません。Python は自動的に各イベントにフック (オプションとしてコールバック) を提供します。Python インタプリタの特性として、大きなオーバーヘッドを伴う傾向がありますが、一般的なアプリケーションに決定論的プロファイリングを用いると、プロセスのオーバーヘッドは少なくて済む傾向があります。結果的に決定論的プロファイリングは少ないコストで、Python プログラムの実行時間に関する統計を得られる方法となっているのです。

呼出し回数はコード中のバグ発見にも使用できます (とんでもない数の呼出しがおこなわれている部分)。インライン拡張の対象とすべき部分を見つけるためにも使えます (呼出し頻度の高い部分)。内部時間の統計は、注意深く最適化すべき“ホット・ループ”の発見にも役立ちます。累積時間の統計は、アルゴリズム選択に関連した高レベルのエラー検知に役立ちます。なお、このプロファイラは再帰的なアルゴリズム実装の累計時間を計ることが可能で、通常のループを使った実装と直接比較することもできるようになっています。

## 10.5 リファレンス・マニュアル

プロファイラのプライマリ・エントリ・ポイントはグローバル関数 `profile.run()` です。通常、プロファイル情報の作成に使われます。情報は `pstats.Stats` クラスのメソッドを使って整形や出力をおこないます。以下はすべての標準エントリポイントと関数の解説です。さらにいくつかのコードの詳細を知りたいければ、「プロファイラの拡張」を読んでください。派生クラスを使ってプロファイラを“改善”する方法やモジュールのソースコードの読み方が述べられています。

```
run(string[, filename[, ...]])
```

この関数はオプション引数として `exec` 文に渡すファイル名を指定できます。このルーチンは必ず最初の引数の `exec` を試み、実行結果からプロファイル情報を収集しようとします。ファイル名が指定されていないときは、各行の標準名 (standard name) 文字列 (ファイル名/行数/関数名) でソートされた、簡単なレポートが表示されます。以下はその出力例です。



```

main()
2706 function calls (2004 primitive calls) in 4.504 CPU seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      2    0.006    0.003    0.953    0.477 pobject.py:75(save_objects)
    43/3    0.533    0.012    0.749    0.250 pobject.py:99(evaluate)
...

```

最初の行はこのプロファイルが `profile.run('main()')` の呼び出しによって生成されたものであり、実行された文字列は `'main()'` であることを示しています。2 行目は 2706 回の関数呼出しがあったことを示しています。このうち 2004 回はプリミティブなものです。プリミティブ な呼出しとは、再帰によるものではない関数呼出しを指します。次の行 `Ordered by: standard name` は、一番右側の欄の文字列を使ってソートされたことを意味します。各カラムの見出しの意味は次の通りです。

**ncalls** 呼出し回数

**tottime** この関数が消費した時間の合計 (サブ関数呼出しの時間は除く)

**percall** `tottime` を `ncalls` で割った値

**cumtime** サブ関数を含む関数の (実行開始から終了までの) 消費時間の合計。この項目は再帰的な関数においても正確に計測されます。

**percall** `cumtime` をプリミティブな呼出し回数で割った値

**filename:lineno(function)** その関数のファイル名、行番号、関数名

(`'43/3'` など) 最初の欄に 2 つの数字が表示されている場合、最初の値は呼出し回数、2 番目はプリミティブな呼出しの回数を表しています。関数が再帰していない場合はどちらの回数も同じになるため、1 つの数値しか表示されません。

プロファイラ・データの分析は `pstats` モジュールを使っておこないます。

```
class Stats(filename[, ...])
```

このコンストラクタは `filename` で指定した (単一または複数の) ファイルから “統計情報オブジェクト” のインスタンスを生成します。Stats オブジェクトはレポートを出力するメソッドを通じて操作します。

上記コンストラクタで指定するファイルは、使用する Stats に対応したバージョンの `profile` で作成されたものでなければなりません。将来のバージョンのプロファイラとの互換性は保証されず、(旧バージョンのものなど) 他のプロファイラとの互換性もないことに注意してください。

複数のファイルを指定した場合、同一の関数の統計情報はすべて合算され、複数のプロセスで構成される全体をひとつのレポートで検証することが可能になります。既存の Stats オブジェクトに別のファイルの情報を追加するときは、`add()` メソッドを使用します。

## 10.5.1 Stats クラス

Stats には次のメソッドがあります。

```
strip_dirs()
```

このメソッドは Stats にファイル名の前に付いているすべてのパス情報を取り除かせるためのものです。出力の幅を 80 文字以内に収めたいときに重宝します。このメソッドはオブジェクトを変更するため、取り除いたパス情報は失われます。パス情報除去の操作後、オブジェクトが保持するデータエント

りは、オブジェクトの初期化、ロード直後と同じように“ランダムに”並んでいます。strip\_dirs() を実行した結果、2つの関数名が区別できない(両者が同じファイルの同じ行番号で同じ関数名となった)場合、一つのエントリに合算されられます。

**add**(filename[, ...])

Stats クラスのこのメソッドは、既存のプロファイリング・オブジェクトに情報を追加します。引数は対応するバージョンの profile.run() によって生成されたファイルの名前でなくてはなりません。関数の名前が区別できない(ファイル名、行番号、関数名が同じ)場合、一つの関数の統計情報として合算されます。

**dump\_stats**(filename)

Stats オブジェクトに読み込まれたデータを、ファイル名 filename のファイルに保存します。ファイルが存在しない場合新たに作成され、すでに存在する場合には上書きされます。このメソッドは profile.Profile クラスの同名のメソッドと等価です。2.3 で追加された仕様です。

**sort\_stats**(key[, ...])

このメソッドは Stats オブジェクトを指定した基準に従ってソートします。引数には通常ソートのキーにしたい項目を示す文字列を指定します(例: 'time' や 'name' など)。

2つ以上のキーが指定された場合、2つ目以降のキーは、それ以前のキーで同等となったデータエントリの再ソートに使われます。たとえば 'sort\_stats('name', 'file')' とした場合、まずすべてのエントリが関数名でソートされた後、同じ関数名で複数のエントリがあればファイル名でソートされるのです。

キー名には他のキーと判別可能である限り綴りを省略して名前を指定できます。現バージョンで定義されているキー名は以下の通りです。

正式名	内容
'calls'	呼び出し回数
'cumulative'	合計時間
'file'	ファイル名
'module'	モジュール名
'pcalls'	プリミティブな呼び出しの回数
'line'	行番号
'name'	関数名
'nfl'	関数名/ファイル名/行番号
'stdname'	標準名
'time'	内部時間

すべての統計情報のソート結果は降順(最も多く時間を消費したものが一番上に来る)となることに注意してください。ただし、関数名、ファイル名、行数に関しては昇順(アルファベット順)になります。'nfl' と 'stdname' はやや異なる点があります。標準名(standard name)とは表示欄の名前なのですが、埋め込まれた行番号の文字コード順でソートされます。たとえば、(ファイル名が同じで)3、20、40 という行番号のエントリがあった場合、20、30、40 の順に表示されます。一方 'nfl' は行番号を数値として比較します。結果的に、sort\_stats('nfl') は sort\_stats('name', 'file', 'line') と指定した場合と同じになります。

旧バージョンのプロファイラとの互換性のため、数値を引数に使った -1、0、1、2 の形式もサポートしています。それぞれ 'stdname'、'calls'、'time'、'cumulative' として処理されます。引数をこの旧形式で指定した場合、最初のキー(数値キー)だけが使われ、複数のキーを指定しても2番目以降は無視されます。

**reverse\_order**()

Stats クラスのこのメソッドは、オブジェクト内の情報のリストを逆順にソートします。これは旧

プロファイラとの互換性のために用意されています。現在は選択したキーに応じて昇順、降順が適切に選ばれるため、このメソッドの必要性はほとんどないはずです。

```
print_stats([restriction, ...])
```

Stats クラスのこのメソッドは、`profile.run()` の項で述べた プロファイルのレポートを出力します。

出力するデータの順序はオブジェクトに対し最後におこなった `sort_stats()` による操作にもとづいたものになります (`add()` と `strip_dirs()` による制限にも支配されます)。

引数は一覧に大きな制限を加えることになります。初期段階でリストはプロファイルした関数の完全な情報を持っています。制限の指定は (行数を指定する) 整数、(行のパーセンテージを指定する) 0.0 から 1.0 までの割合を指定する小数、(出力する standard name にマッチする) 正規表現のいずれかを使っておこないます。正規表現は Python 1.5b1 で導入された `re` モジュールで使える Perl スタイルのものです。複数の制限は指定された場合、それは指定の順に適用されます。たとえば次のようになります。

```
print_stats(.1, 'foo:')
```

上記の場合まず出力するリストは全体の 10% に制限され、さらにファイル名の一部に文字列 `.*foo:` を持つ関数だけが出力されます。

```
print_stats('foo:', .1)
```

こちらの例の場合、リストはまずファイル名に `.*foo:` を持つ関数だけに制限され、その中の最初の 10% だけが出力されます。

```
print_callers([restriction, ...])
```

Stats クラスのこのメソッドは、プロファイルのデータベースの中から何らかの関数呼び出しをおこなった関数すべてを出力します。出力の順序は `print_stats()` によって与えられるものと同じです。出力を制限する引数も同じです。呼出し側関数の後にパーレンで囲まれて表示される数値は呼出しが何回おこなわれたかを示すものです。続いてパーレンなしで表示される数値は、関数が消費した時間の合計です。

```
print_callees([restriction, ...])
```

Stats クラスのこのメソッドは指定した関数から呼出された関数のリストを出力します。呼出し側、呼出される側の方向は逆ですが、引数と出力の順序に関しては `print_callers()` と同じです。

```
ignore()
```

リリース 1.5.1 以降で撤廃された仕様です。現バージョンの Python の Python では不要です。<sup>2</sup>

## 10.6 制限事項

このプロファイラには 2 つの基本的な制限事項があります。ひとつは、Python インタプリタによる 呼び出し、リターン、例外発生 というイベントの連携を前提にしていることです。コンパイル済みの C コードはインタプリタの管理外で、プロファイラからは“見えません”。(組込み関数を含む) C のコードに費やされた時間は、その C コードを呼出した Python 関数のものとして計上されることになります。ただし C コードが Python のコードを呼び出す場合は、適切にプロファイルされます。

2 つ目の制限はタイミング情報の正確さに関するものです。決定論的プロファイラの正確さに関する根本的問題です。最も明白な制限は、(一般に)“クロック”は .001 秒の精度しかないということです。これ以上

<sup>2</sup> Python が None 以外の使われなかった結果を表示するときに使われたもので、旧バージョンとの互換性のためだけに定義されています。

の精度で計測することはできません。仮に十分な精度が得られたとしても、“エラー”が計測の平均値に影響を及ぼすことがあります。最初のエラーを取り除いたとしても、それがまた別のエラーを引き起こす原因となり...

もうひとつの問題として、イベントを検知してからプロファイラがその時刻を実際に取得するまでに“いくらかの時間がかかる”ことです。プロファイラが時刻を取得する(そしてその値を保存する)までの間に、ユーザコードがもう一度処理を実行したときにも、同様の遅延が発生します。結果的に多く呼び出される関数または多数の関数から呼び出される関数の情報にはこの種のエラーが蓄積する傾向にあります。

この種のエラーによる遅延の蓄積は一般にクロックの精度を越える(1クロック以下のタイミング)ところで起きていますが、一方でこの時間を累計可能ということが大きな意味を持っています。このプロファイラはプラットフォームごとに(平均値から)予想されるエラーによる遅延を補正する機能を備えています。プロファイラに補正を施すと(少くとも形式的には)正確さが増しますが、ときには数値が負の値になってしまうこともあります(呼出し回数が少く、確率の神があなたに意地悪をしたとき:-)。プロファイルの結果に負の値が出力されても驚かないでください。これは補正をおこなった場合にのみ現れることで、実際の計測結果は補正をおこなわない場合より、より正確なはずだからです。

## 10.7 キャリブレーション(補正)

プロファイラは イベントをハンドリングの際の time 関数呼出しおよびその値を保存するためのオーバーヘッドを補正するための、定数を持っています。デフォルトの値は0です。以下の手順で、プラットフォームに合った、より適切な定数が得られます(前節「制限事項」の説明を参照)。

```
import profile
pr = profile.Profile()
for i in range(5):
    print pr.calibrate(10000)
```

メソッドは引数として与えられた数だけ Python の呼出しをおこないます。呼出しは直接、プロファイラを使って呼出しの両方が実施され、それぞれの時間が計測されます。その結果、プロファイラのイベントに隠されたオーバーヘッドが計算され、その値は浮動小数として返されます。たとえば、800 MHz の Pentium で Windows 2000 を使用、Python の time.clock() をタイマとして使った場合、値はおよそ 12.5e-6 となります。

この手順で使用しているオブジェクトはほぼ一定の結果を返します。非常に早いコンピュータを使う場合、もしくはタイマの性能が貧弱な場合は一定の結果を得るために引数に 100000 や 1000000 といった大きな値を指定する必要があるかもしれません。

一定の結果が得られたら、それを使う方法には3通りあります。<sup>3</sup>

---

<sup>3</sup> Python 2.2 より前のバージョンではプロファイラのソースコードに補正值として埋め込まれた定数を直接編集する必要がありました。今でも同じことは可能ですが、その方法は説明しません。なぜなら、もうソースを編集する必要がないからです。

```

import profile

# 1. 算出した補正值 (your_computed_bias) をこれ以降生成する
#     Profile インスタンスに適用する。
profile.Profile.bias = your_computed_bias

# 2. 特定の Profile インスタンスに補正值を適用する。
pr = profile.Profile()
pr.bias = your_computed_bias

# 3. インスタンスのコンストラクタに補正值を指定する。
pr = profile.Profile(bias=your_computed_bias)

```

方法を選択したら、補正值は小さめに設定した方が良いでしょう。プロファイルの結果に負の値が表われる“確率が少なく”なるはずです。

## 10.8 拡張 — プロファイラの改善

profile モジュールの Profile クラスはプロファイラの機能を拡張するため、派生クラスの作成を前提に書かれています。しかしその方法を説明するには、Profile の内部動作について詳細な解説が必要となるため、ここでは述べません。もし拡張をおこないたいのであれば、profile モジュールのソースを注意深く読む必要があります。

プロファイラが時刻を取得する方法を変更したいだけなら (たとえば、通常の時間 (wall-clock) を使いたいとか、プロセスの経過時間を使いたい場合)、時刻取得用の関数を Profile クラスのコンストラクタに指定することができます。

```
pr = profile.Profile(your_time_func)
```

この結果生成されるプロファイラは時刻取得に your\_time\_func() を呼び出すようになります。このようなユーザ定義関数は単一の数値、あるいはその合計が (os.times() と同じように) 累計時間を示すリストを返すようになっていなければなりません。関数が 1 つの数値、あるいは長さ 2 の数値のリストを返すようになっていれば、非常に高速に処理が可能になります。

選択する時刻取得関数によって、プロファイラクラスを補正する必要があることに注意してください。多くのマシンにおいて、プロファイル時のオーバーヘッドを少なくする方法として、タイマはロング整数を返すのが最善です。os.times() は浮動小数のタプルを返すので おすすめできません。タイマをより正確なものに置き換えたいならば、派生クラスでそのディスパッチ・メソッドを適切なタイマ呼出しと適切な補正をおこなうように書き直す必要があります。

## 10.9 hotshot — ハイパフォーマンス・ロギング・プロファイラ

2.2 で追加された仕様です。

このモジュールは \_hotshot C モジュールへのより良いインターフェースを提供します。Hotshot は既存の profile に置き換わるものです。その大半が C で書かれているため、profile に比べパフォーマンス上の影響ははるかに少なく済みます。

```
class Profile(logfile[, lineevents=0[, linetimings=1]])
```

プロファイラ・オブジェクト。引数 logfile はプロファイル・データのログを保存するファイル名です。引数 lineevents はソースコードの 1 行ごとにイベントを発生させるか、関数の呼び出し/リターンのと

きだけ発生させるかを指定します。デフォルトの値は 0 (関数の呼び出し/ リターンするときだけログを残す) です。引数 *linetimings* は時間情報を記録するかどうかを指定します。デフォルトの値は 1 (時間情報を記録する) です。

### 10.9.1 プロファイル・オブジェクト

プロファイル・オブジェクトは以下のメソッドを持っています。

**addinfo**(*key*, *value*)

プロファイル出力の際、任意のラベル名を追加します。

**close**()

ログファイルを閉じ、プロファイラを終了します。

**fileno**()

プロファイラのログファイルのファイル・ディスクリプタを返します。

**run**(*cmd*)

スクリプト環境で `exec` 互換文字列のプロファイルをおこないます。 `__main__` モジュールのグローバル変数は、スクリプトのグローバル変数、ローカル変数の両方に使われます。

**runcall**(*func*, *\*args*, *\*\*keywords*)

単一の呼び出し可能オブジェクトのプロファイルをおこないます。位置依存引数やキーワード引数を追加して呼び出すオブジェクトに渡すこともできます。呼び出しの結果はそのまま返されます。例外が発生したときはプロファイリングが無効になり、例外をそのまま伝えるようになっています。

**runtx**(*cmd*, *globals*, *locals*)

指定した環境で `exec` 互換文字列の評価をおこないます。文字列のコンパイルはプロファイルを開始する前におこなわれます。

**start**()

プロファイラを開始します。

**stop**()

プロファイラを停止します。

### 10.9.2 hotshot データの利用

2.2 で追加された仕様です。

このモジュールは hotshot プロファイル・データを標準の `pstats` オブジェクトにロードします。

**load**(*filename*)

*filename* から hotshot データを読み込み、`pstats.Stats` クラスのインスタンスを返します。

参考資料:

`profile` モジュール (10.5 節):

`profile` モジュールの `Stats` クラス

### 10.9.3 使用例

これは Python の “ベンチマーク” `pystone` を使った例です。実行にはやや時間がかかり、巨大な出力ファイルを生成するので注意してください。



```
>>> import hotshot, hotshot.stats, test.pystone
>>> prof = hotshot.Profile("stones.prof")
>>> benchtime, stones = prof.runcall(test.pystone.pystones)
>>> prof.close()
>>> stats = hotshot.stats.load("stones.prof")
>>> stats.strip_dirs()
>>> stats.sort_stats('time', 'calls')
>>> stats.print_stats(20)
      850004 function calls in 10.090 CPU seconds

Ordered by: internal time, call count

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1      3.295      3.295    10.090    10.090 pystone.py:79(Proc0)
150000      1.315      0.000      1.315      0.000 pystone.py:203(Proc7)
 50000      1.313      0.000      1.463      0.000 pystone.py:229(Func2)
.
.
.
```

## 10.10 timeit — 小さなコード断片の実行時間計測

2.3 で追加された仕様です。

このモジュールは Python の小さなコード断片の時間を簡単に計測する手段を提供します。インターフェースはコマンドラインとメソッドとして呼び出し可能なものの両方を備えています。また、このモジュールは実行時間の計測にあたり陥りがちな落とし穴に対する様々な対策が取られています。詳しくは、O'Reilly の *Python Cookbook*、"Algorithms" の章にある Tim Peters が書いた解説を参照してください。

このモジュールには次のパブリック・クラスが定義されています。

```
class Timer ([stmt='pass' [, setup='pass' [, timer=<timer function> ]]])
```

小さなコード断片の実行時間計測をおこなうためのクラスです。

コンストラクタは引数として、時間計測の対象となる文、セットアップに使用する追加の文、タイマ関数を受け取ります。文のデフォルト値は両方とも 'pass' で、タイマ関数はプラットフォーム依存 (モジュールの doc string を参照) です。文には複数行の文字列リテラルを含まない限り、改行を入れることも可能です。

最初の文の実行時間を計測には `timeit()` メソッドを使用します。また `timeit()` を複数回呼び出し、その結果のリストを返す `repeat()` メソッドも用意されています。

```
print_exc ([file=None])
```

計測対象コードのトレースバックを出力するためのヘルパー。

利用例:

```
t = Timer(...)          # try/except の外側で
try:
    t.timeit(...)        # または t.repeat(...)
except:
    t.print_exc()
```

標準のトレースバックより優れた点は、コンパイルしたテンプレートのソース行が表示されることです。オプションの引数 `file` にはトレースバックの出力先を指定します。デフォルトは `sys.stderr` になっています。

```
repeat([repeat=3 [, number=1000000]])
```

timeit() を複数回呼び出します。

このメソッドは timeit() を複数回呼び出し、その結果をリストで返すユーティリティ関数です。最初の引数には timeit() を呼び出す回数を指定します。2 番目の引数は timeit() へ引数として渡す数値です。

注意:

結果のベクトルから平均値や標準偏差を計算して出力させたいと思うかもしれませんが、それはあまり意味がありません。多くの場合、最も低い値がそのマシンが与えられたコード断片を実行する場合の下限値です。結果のうち高めの値は、Python のスピードが一定しないために生じたものではなく、時刻取得の際他のプロセスと衝突がおこったため、正確さが損なわれた結果生じたものです。したがって、結果のうち min() だけが見るべき値となります。この点を押さえた上で、統計的な分析よりも常識的な判断で結果を見るようにしてください。

```
timeit([number=1000000])
```

メイン文の実行時間を *number* 回取得します。このメソッドはセットアップ文を 1 回だけ実行し、メイン文を指定回数実行するのにかかった秒数を浮動小数で返します。引数はループを何回実行するかで、デフォルト値は 100 万回です。メイン文、セットアップ文、タイマ関数はコンストラクタで指定されたものを使用します。

### 10.10.1 コマンドライン・インターフェース

コマンドラインからプログラムとして呼び出す場合は、次の書式を使います。

```
python timeit.py [-n N] [-r N] [-s S] [-t] [-c] [-h] [statement ...]
```

以下のオプションが使用できます。

**-n N/--number=N** 'statement' を何回実行するか

**-r N/--repeat=N** タイマを何回リピートするか (デフォルトは 3)

**-s S/--setup=S** 最初に 1 回だけ実行する文 (デフォルトは 'pass')

**-t/--time** time.time() を使用する (Windows を除くすべてのプラットフォームのデフォルト)

**-c/--clock** time.clock() を使用する (Windows のデフォルト)

**-v/--verbose** 時間計測の結果をそのまま詳細な数値でくり返し表示する

**-h/--help** 簡単な使い方を表示して終了する

文は複数行指定することもできます。その場合、各行は独立した文として引数に指定されたものとして処理します。クォートと行頭のスペースを使って、インデントした文を使うことも可能です。この複数行のオプションは -s においても同じ形式で指定可能です。

オプション **-n** でループの回数が指定されていない場合、10 回から始めて、所要時間が 0.2 秒になるまで回数を増やすことで適切なループ回数が自動計算されるようになっています。

デフォルトのタイマ関数はプラットフォーム依存です。Windows の場合、time.clock() はマイクロ秒の精度がありますが、time.time() は 1/60 秒の精度しかありません。一方 UNIX の場合、time.clock() でも 1/100 秒の精度があり、time.time() はもっと正確です。いずれのプラットフォームにおいても、デフォルトのタイマ関数は CPU 時間ではなく通常の時間を返します。つまり、同じコンピュータ上で別のプ

ロセスが動いている場合、タイミングの衝突する可能性があるということです。正確な時間を割り出すために最善の方法は、時間の取得を数回くり返しその中の最短の時間を採用することです。-r オプションはこれをおこなうもので、デフォルトのくり返し回数は3回になっています。多くの場合はデフォルトのままで充分でしょう。UNIX の場合 `time.clock()` を使って CPU 時間で測定することもできます。

注意: `pass` 文の実行による基本的なオーバーヘッドが存在することに注意してください。ここにあるコードはこの事実を隠そうとはしておらず、注意を払う必要があります。基本的なオーバーヘッドは引数なしでプログラムを起動することにより計測できます。

基本的なオーバーヘッドは Python のバージョンによって異なります。Python 2.3 とそれ以前の Python の公平な比較をおこなう場合、古い方の Python は -O オプションで起動し `SET_LINENO` 命令の実行時間が含まれないようにする必要があります。

## 10.10.2 使用例

以下に2つの使用例を記載します(ひとつはコマンドライン・インターフェースによるもの、もうひとつはモジュール・インターフェースによるものです)。内容はオブジェクトの属性の有無を調べるのに `hasattr()` を使った場合と `try/except` を使った場合の比較です。

```
% timeit.py 'try:' ' str.__nonzero__' 'except AttributeError:' ' pass'
100000 loops, best of 3: 15.7 usec per loop
% timeit.py 'if hasattr(str, "__nonzero__"): pass'
100000 loops, best of 3: 4.26 usec per loop
% timeit.py 'try:' ' int.__nonzero__' 'except AttributeError:' ' pass'
1000000 loops, best of 3: 1.43 usec per loop
% timeit.py 'if hasattr(int, "__nonzero__"): pass'
100000 loops, best of 3: 2.23 usec per loop
```

```

>>> import timeit
>>> s = """\
... try:
...     str.__nonzero__
... except AttributeError:
...     pass
... """
>>> t = timeit.Timer(stmt=s)
>>> print "%.2f usec/pass" % (1000000 * t.timeit(number=100000)/100000)
17.09 usec/pass
>>> s = """\
... if hasattr(str, '__nonzero__'): pass
... """
>>> t = timeit.Timer(stmt=s)
>>> print "%.2f usec/pass" % (1000000 * t.timeit(number=100000)/100000)
4.85 usec/pass
>>> s = """\
... try:
...     int.__nonzero__
... except AttributeError:
...     pass
... """
>>> t = timeit.Timer(stmt=s)
>>> print "%.2f usec/pass" % (1000000 * t.timeit(number=100000)/100000)
1.97 usec/pass
>>> s = """\
... if hasattr(int, '__nonzero__'): pass
... """
>>> t = timeit.Timer(stmt=s)
>>> print "%.2f usec/pass" % (1000000 * t.timeit(number=100000)/100000)
3.15 usec/pass

```

定義した関数に timeit モジュールがアクセスできるようにするために、import 文の入った setup 引数を渡すことができます:

```

def test():
    "Stupid test function"
    L = []
    for i in range(100):
        L.append(i)

if __name__ == '__main__':
    from timeit import Timer
    t = Timer("test()", "from __main__ import test")
    print t.timeit()

```

# インターネットプロトコルとその支援

この章で記述されるモジュールは、インターネットプロトコルと関連技術の支援を実装します。それらは全て Python で実装されています。これらのモジュールの大部分は、システム依存のモジュール `socket` が存在することが必要ですが、これは現在ではほとんどの一般的なプラットフォーム上でサポートされています。ここに概観を示します。

<code>webbrowser</code>	ウェブブラウザのための使い易いコントローラー
<code>cgi</code>	サーバ側で動作するスクリプトにおいてフォームの内容を解釈するために使われるゲートウェイ
<code>cgitb</code>	設定可能な、CGI スクリプトのトレースバック処理機構です。
<code>urllib</code>	URL による任意のネットワークリソースへのアクセス ( <code>socket</code> が必要です)。
<code>urllib2</code>	様々なプロトコルで URL を開くための拡張可能なライブラリ
<code>httpplib</code>	HTTP および HTTPS プロトコルのクライアント (ソケットを必要とします)。
<code>ftplib</code>	FTP プロトコルクライアント (ソケットを必要とします)。
<code>gopherlib</code>	<code>gopher</code> プロトコルのクライアント (ソケットを必要とします)。
<code>poplib</code>	POP3 プロトコルクライアント ( <code>sockets</code> を必要とする)
<code>imaplib</code>	IMAP4 protocol client (requires sockets).
<code>nntplib</code>	NNTP プロトコルクライアント (ソケットを必要とします)。
<code>smtplib</code>	SMTP プロトコル クライアント (ソケットが必要です)。
<code>telnetlib</code>	Telnet クライアントクラス
<code>urlparse</code>	URL を解析して構成要素にします。
<code>SocketServer</code>	ネットワークサーバ構築のためのフレームワーク。
<code>BaseHTTPServer</code>	基本的な機能を持つ HTTP サーバ ( <code>SimpleHTTPServer</code> および <code>CGIHTTPServer</code> の基底)
<code>SimpleHTTPServer</code>	このモジュールは HTTP サーバに基本的なリクエストハンドラを提供します。
<code>CGIHTTPServer</code>	CGI スクリプトの実行機能を持つ HTTP サーバのためのリクエスト処理機構を提供します。
<code>Cookie</code>	HTTP 状態管理 (cookies) のサポート。
<code>xmlrpclib</code>	XML-RPC client access.
<code>SimpleXMLRPCServer</code>	基本的な XML-RPC サーバの実装。
<code>DocXMLRPCServer</code>	セルフ-ドキュメンティング XML-RPC サーバの実装。
<code>asyncore</code>	非同期なソケット制御サービスのためのベースクラス
<code>asynchat</code>	非同期コマンド/レスポンスプロトコルの開発サポート

## 11.1 `webbrowser` — 便利なウェブブラウザコントローラー

`webbrowser` モジュールにはウェブベースのドキュメントを表示するための、とてもハイレベルなインターフェイスが定義されています。このコントローラーオブジェクトは使い易く、プラットフォーム非依存です。たいいていの環境では、このモジュールの `open()` を呼び出すだけで正しく動作します。

UNIX では、X11 上でグラフィカルなブラウザが選択されますが、グラフィカルなブラウザが利用できなかったり、X11 が利用できない場合はテキストモードのブラウザが使われます。もしテキストモードのブ

ブラウザが使われたら、ユーザがブラウザから抜け出すまでプロセスの呼び出しはブロックされます。

UNIX では、環境変数 BROWSER が存在するならプラットフォームのデフォルトであるブラウザのリストをオーバーライドし、コロンで区切られたリストの順にブラウザの起動を試みます。リストの中の値に %s が含まれていたら、テキストモードのブラウザのコマンドラインとして %s の代わりに URL が引数として解釈されます；もし %s が含まれなければ、起動するブラウザの名前として単純に解釈されます。

非 UNIX プラットフォームあるいは UNIX 上で X11 ブラウザが利用可能な場合、制御プロセスはユーザがブラウザを終了するのを待ちませんが、ディスプレイにブラウザのウィンドウを表示させたままにします。

以下の例外が定義されています：

#### exception Error

ブラウザのコントロールエラーが起こると発生する例外。

以下の関数が定義されています：

**open**(url[, new=0][, autoraise=1])

デフォルトのブラウザで url を表示します。new が true なら、可能であればブラウザの新しいウィンドウが開きます。autoraise が true なら、可能であればウィンドウが前面に表示されます（多くのウィンドウマネージャではこの変数の設定に関わらず、前面に表示されます）。

**open\_new**(url)

可能であれば、デフォルトブラウザの新しいウィンドウで url を開きますが、そうでない場合はブラウザのただ 1 つのウィンドウで url を開きます。

**get**([name])

ブラウザの種類 name のコントローラオブジェクトを返します。もし name が空文字列なら、呼び出した環境に適したデフォルトブラウザのコントローラを返します。

**register**(name, constructor[, instance])

ブラウザの種類 name を登録します。ブラウザの種類が登録されたら、get() でそのブラウザのコントローラを呼び出すことができます。instance が指定されなかったり、None なら、インスタスが必要な時には constructor がパラメータなしに呼び出されて作られます。instance が指定されたら、constructor は呼び出されないで、None でかまいません。

この登録は、変数 BROWSER を設定するか、get を空文字列でなく、宣言したハンドラの名前と一致する引数とともに呼び出すときだけ、役に立ちます。

いくつかの種類のブラウザがあらかじめ定義されています。このモジュールで定義されている、関数 get() に与えるブラウザの名前と、それぞれのコントローラクラスのインスタンスを以下の表に示します。

Type Name	Class Name	Notes
'mozilla'	Netscape('mozilla')	(1)
'netscape'	Netscape('netscape')	
'mosaic'	GenericBrowser('mosaic %s &')	
'kfm'	Konqueror()	
'grail'	Grail()	
'links'	GenericBrowser('links %s')	
'lynx'	GenericBrowser('lynx %s')	
'w3m'	GenericBrowser('w3m %s')	(2)
'windows-default'	WindowsDefault	
'internet-config'	InternetConfig	(3)

Notes:



- (1) “Konqueror” は UNIX の KDE デスクトップ環境のファイルマネージャで、KDE が動作している時にだけ意味を持ちます。何か信頼できる方法で KDE を検出するのがいいでしょう；変数 `KDEDIR` では十分ではありません。また、KDE 2 で `konqueror` コマンドを使うときにも、“kfm” が使われます—Konqueror を動作させるのに最も良い方法が実装によって選択されます。
- (2) Windows プラットフォームのみ；標準拡張モジュール `win32api` と `win32con` を必要とします。
- (3) MacOS プラットフォームのみ；*Macintosh Library Modules* マニュアルに解説されている標準 MacPython モジュール `ic` を必要とします。

### 11.1.1 ブラウザコントローラーオブジェクト

ブラウザコントローラーには 2 つのメソッドが定義されていて、モジュールレベルの便利な 2 つの関数に相当します：

`open(url[, new])`

このコントローラーでハンドルされたブラウザで `url` を表示します。`new` が `true` なら、可能であればブラウザの新しいウィンドウが開きます。

`open_new(url)`

可能であれば、このコントローラーでハンドルされたブラウザの新しいウィンドウで `url` を開きますが、そうでない場合はブラウザのただ 1 つのウィンドウで `url` を開きます。

## 11.2 cgi — CGI (ゲートウェイインタフェース規格) のサポート

ゲートウェイインタフェース規格 (CGI) に準拠したスクリプトをサポートするためのモジュールです。

このモジュールは Python で書かれた CGI スクリプトで使われる数多くのユーティリティを定義しています。

### 11.2.1 はじめに

CGI スクリプトは、HTTP サーバによって起動され、通常は HTML の `<FORM>` または `<ISINDEX>` エレメントを通じてユーザが入力した内容を処理するために使われます。

ほとんどの場合、CGI スクリプトはサーバ上の特殊な ‘cgi-bin’ ディレクトリ下に置かれます。HTTP サーバはまずリクエストの全ての情報 (クライアントのホスト名、リクエストされている URL、クエリ文字列、その他諸々) をスクリプトを動作させるシェルの環境変数に設定し、スクリプトを実行した後、スクリプトの出力をクライアントに送信します。

スクリプトの入力端もまたクライアントに接続されており、フォーム上のデータは時としてこの経路を通じて読み出されます；その他の場合、フォームデータは URL の一部分である “クエリ文字列” を介して渡されます。このモジュールでは、上記のケースの違いに注意しながら Python スクリプトに対しては単純なインタフェースを提供するためのものです。このモジュールではまた、スクリプトをデバッグするためのユーティリティを多数提供しています。また、最近ではフォームを経由したファイルのアップロードをサポートしています (ブラウザ側がサポートしていればです — Grail 0.3 および Netscape 2.0 はサポートしています。)

CGI スクリプトの出力は 2 つのセクションからなり、空行で分割されています。最初のセクションは複数のヘッダからなり、後続するデータがどのようなものかをクライアントに通知します。最小のヘッダセクションを生成するための Python のコードは以下のようなものです：

```
print "Content-Type: text/html"      # HTML is following
print                               # blank line, end of headers
```

二つ目のセクションは通常、ヘッダやインラインイメージ等の付属したテキストをうまくフォーマットして表示できるようにした HTML です。以下に単純な HTML を出力する Python コードを示します:

```
print "<TITLE>CGI script output</TITLE>"
print "<H1>This is my first CGI script</H1>"
print "Hello, world!"
```

### 11.2.2 cgi モジュールを使う

先頭には `import cgi` と書いてください。 `from cgi import *` と書いてはいけません — このモジュールは以前のバージョンとの互換性を持たせるために内部で呼び出すための名前を多数定義しており、それらがユーザの名前空間に存在する必要はありません。

新たにスクリプトを各なら、以下の行を付け加えることを検討してください:

```
import cgitb; cgitb.enable()
```

これによって、特別な例外処理が有効にされ、エラーが発生した際にブラウザ上に詳細な報告を出力するようになります。ユーザにスクリプトの内部を見せたくないのなら、以下のようにして報告をファイルに保存することもできます:

```
import cgitb; cgitb.enable(display=0, logdir="/tmp")
```

スクリプトを開発する際には、この機能はとても役に立ちます。 `cgitb` が生成する報告はバグを追跡するために掛かる時間を大きく減らせるような情報を提供してくれます。スクリプトをテストし終わり、正確に動作することを確認した後はいつでも `cgitb` の行を削除することができます。

入力されたフォームデータを取得するには、 `FieldStorage` クラスを使うのが最良の方法です。このモジュール定義されている他のクラスのほとんどは以前のバージョンとの互換性のためのものです。インスタンス生成は引数なしで必ず 1 度だけ行います。これにより、標準入力または環境変数からフォームの内容が読み出されます (どちらから読み出されるかは、CGI 標準に従って設定されている複数の環境変数の値によって決められます)。インスタンスが標準入力を使うかもしれないので、インスタンス生成を行うのは一度だけにしなければなりません。

`FieldStorage` のインスタンスは Python の辞書のようにインデックス参照することができ、標準の辞書に対するメソッド `has_key()` と `keys()` をサポートしています。組み込みの関数 `len()` もサポートされています。空の文字列を含むフォームのフィールドは無視され、辞書上にはありません; そういった値を保持するには、 `FieldStorage` のインスタンスを生成する時にオプションの `keep_blank_values` キーワード引数を `true` に設定してください。

例えば、以下のコード (Content-Type: ヘッダと空行はすでに出力された後とします) は `name` および `addr` フィールドが両方とも空の文字列に設定されていないか調べます:

```

form = cgi.FieldStorage()
if not (form.has_key("name") and form.has_key("addr")):
    print "<H1>Error</H1>"
    print "Please fill in the name and addr fields."
    return
print "<p>name:", form["name"].value
print "<p>addr:", form["addr"].value
...further form processing here...

```

ここで、`form[key]` で参照される各フィールドはそれ自体が `FieldStorage` (または `MiniFieldStorage`。フォームのエンコードによって変わります) のインスタンスです。インスタンスの属性 `value` フィールドの文字列値になります。`getvalue()` メソッドはこの文字列値を直接返します。`getvalue()` は2つめの引数にオプションの値を与えることができ、リクエストされたキーが存在しない場合に返されるデフォルトの値になります。

入力されたフォームデータが同じ名前をもった1個以上のフィールドを持つ場合、`form[key]` で得られるオブジェクトは `FieldStorage` や `MiniFieldStorage` のインスタンスではなく、それらインスタスからなるリストになります。同じく、この状況では、`form.getvalue(key)` は文字列を要素とするリストを返します。もしこうした状況が起こりうるとされる (HTML のフォームに同じ名前をもったフィールドが複数含まれている) なら、組み込み関数 `isinstance()` を使って、返された値が単一のインスタンスかインスタンスのリストかを調べてください。例えば、以下のコードは任意の数のユーザ名フィールドを結合し、コンマで分割された文字列にします:

```

value = form.getvalue("username", "")
if isinstance(value, list):
    # Multiple username fields specified
    usernames = ",".join(value)
else:
    # Single or no username field specified
    usernames = value

```

フィールドがアップロードされたファイルを表す場合、`value` 属性または `getvalue()` メソッドを介してアクセスされた値はファイルの内容を全て文字列としてメモリ上に読み込みます。これは望ましくない機能かもしれません。アップロードされたファイルがあるかどうかは `filename` 属性および `file` 属性のいずれかで調べることができます。その後、以下のようにして `file` 属性から落ちていてデータを読み出すことができます:

```

fileitem = form["userfile"]
if fileitem.file:
    # It's an uploaded file; count lines
    linecount = 0
    while 1:
        line = fileitem.file.readline()
        if not line: break
        linecount = linecount + 1

```

現在ドラフトとなっているファイルアップロードの標準仕様では、一つのフィールドから (再帰的な `multipart/*` エンコーディングを使うことで) 複数のファイルがアップロードされる可能性を受け入れています。この場合、アイテムは辞書形式の `FieldStorage` アイテムとなります。複数ファイルかどうかは `type` 属性が `multipart/form-data` (または `multipart/*` にマッチする他の MIME 型) になっているかどうか調べることで判別できます。この場合、トップレベルのフォームオブジェクトと同様にして再帰的に個別処理することができます。

フォームが“古い”形式で入力された場合 (クエリ文字列または application/x-www-form-urlencoded データの単一の部分の場合) アイテムは実際には MiniFieldStorage クラスのインスタンスになります。この場合、list、file、および filename 属性は常に None になります。

### 11.2.3 高レベルインタフェース

2.2 で追加された仕様です。

前節では CGI フォームデータを FieldStorage クラスを使って読み出す方法について解説しました。この節ではより分かりやすく直感的な方法でフォームデータを読み出せるようにするために追加されたより高レベルのインタフェースについて記述します。このインタフェースは前節で記述された技術を廃用化するものではありません — 例えば、前節の技術は依然としてファイルのアップロードを効率的に行う上で便利です。

このインタフェースは 2 つの単純なメソッドからなります。このメソッドを使うことで、一般的な方法でフォームデータを処理することができ、一つのフォーム名についてポストされた値が一つなのかそれ以上なのかについて心配する必要がなくなります。

前節では、一つのフォーム名に対して一つ以上の値がポストされるかもしれない場合には常に以下のようなコードを書くよう学びました:

```
item = form.getvalue("item")
if isinstance(item, list):
    # The user is requesting more than one item.
else:
    # The user is requesting only one item.
```

こういった状況は、例えば以下のように、同じ名前を持った複数のチェックボックスからなるグループがフォームに含まれているような場合によく起こります:

```
<input type="checkbox" name="item" value="1" />
<input type="checkbox" name="item" value="2" />
```

しかしながら、ほとんどの場合あるフォーム中で特定の名前を持ったコントロールはただ一つなので、その名前に関連付けられた値はただ一つしかないと期待され、かつ一つの値しか必要ありません。そこで、スクリプトには例えば以下のようなコードを書くでしょう:

```
user = form.getvalue("user").toupper()
```

このコードの問題点は、クライアント側がスクリプトに有効な入力を提供すると期待できないところにあります。例えば、もし好奇心旺盛なユーザがもう一つの ‘user=foo’ ペアをクエリ文字列に追加したら、getvalue(“user”) メソッドを呼び出したときに文字列ではなくリストが返されるため、このスクリプトはクラッシュするでしょう。リストに対して toupper() メソッドを呼び出すと、引数が有効でない (リスト型はその名前のメソッドを持っていない) ため、例外 AttributeError が送出されます。

従って、フォームデータに入力された値を読み出すための適切な方法は、常に取得した値が単一の値からなるか、あるいは複数の値からなるかを調べるコードを使うことになります。これでは煩わしく、また、より読みにくいスクリプトになってしまいます。

より便利なアプローチは、ここで述べる高レベルのインタフェースが提供する getfirst() および getlist() メソッドを使うことです。

```
getfirst(name[, default])
```

フォームフィールド *name* に関連付けられた値をつねに一つだけ返す軽量メソッドです。このメソッドは同じ名前で 1 つ以上の値がポストされた場合、最初の値だけを返します。フォームから値を受信する際の値が並べられる順番はブラウザ間で異なる可能性があり、特定の順番であるとは期待できないので注意してください。<sup>1</sup>

指定したフォームフィールドや値がない場合、このメソッドはオプションの引数 *default* を返します。このパラメタを指定しない場合、標準の値は `None` に設定されます。

`getlist(name)`

このメソッドはフォームフィールド *name* に関連付けられた値を常にリストで返します。*name* に指定したフォームフィールドや値が存在しない場合、このメソッドは空のリストを返します。値が一つだけ存在する場合、要素を一つだけ含むリストを返します。

これらのメソッドを使うことで、以下のようにナイスでコンパクトなコードを書くことができます：

```
import cgi
form = cgi.FieldStorage()
user = form.getfirst("user", "").toupper()    # This way it's safe.
for item in form.getlist("item"):
    do_something(item)
```

## 11.2.4 古いクラス群

これらのクラスは、`cgi` モジュールの以前のバージョンに入っており、以前のバージョンとの互換性のために現在もサポートされています。新しいアプリケーションでは `FieldStorage` クラスを使うべきです。

`SvFormContentDict` は単一の値しか持たないフォームデータの内容を辞書として記憶します；このクラスでは、各フィールド名はフォーム中に一度しか現れないと仮定しています。

`FormContentDict` は複数の値を持つフォームデータの内容を辞書として記憶します（フォーム要素は値のリストです）；フォームが同じ名前を持ったフィールドを複数含む際に便利です。

他のクラス（`FormContent`、`InterpFormContentDict`）は非常に古いアプリケーションとの後方互換性のために存在します。これらのクラスをいまだに使用して、個のモジュールの次のバージョンで消えてしまったら非常に不便な場合は私に連絡を下さい。

## 11.2.5 関数

より細かくコントロールしたり、このモジュールで実装されているアルゴリズムを他の状況で利用したい場合には、これらの関数が便利です。

`parse(fp[, keep_blank_values[, strict_parsing]])`

環境変数、またはファイルからクエリを解釈します（ファイルは標準で `sys.stdin` になります）。  
*keep\_blank\_values* および *strict\_parsing* パラメタはそのまま `parse_qs()` に渡されます。

`parse_qs(qs[, keep_blank_values[, strict_parsing]])`

文字列引数として渡されたクエリ文字列（`application/x-www-form-urlencoded` 型のデータ）を解釈します。解釈されたデータは辞書として返されます。辞書のキーは一意的なクエリ変数名で、値は各変数名に対する値からなるリストです。

オプションの引数 *keep\_blank\_values* は、URL エンコードされたクエリ中で値の入っていないものを空文字列と見なすかどうかを示すフラグです。値が真であれば、値の入っていないフィールドは空文

<sup>1</sup>最近のバージョンの HTML 仕様では、フィールドの値が提供される順番を取り決めてはいますが、ある HTTP リクエストがその取り決めに従ったブラウザから受信したのかどうか、そもそもブラウザから送信されたのかどうかの判別は退屈で間違いやすいので注意してください。



文字列のままになります。標準では偽で、値の入っていないフィールドは無視され、そのフィールドはクエリに含まれていないものとして扱われます。

オプションの引数 *strict\_pasing* はパース時のエラーをどう扱うかを定めるフラグです。値が偽なら（標準の設定です）、エラーは暗黙のうちに無視されます。値が真なら、`ValueError` 例外が送出されます。

辞書等をクエリ文字列に変換する場合は `urllib.urlencode()` 関数を使用してください。

`parse_qs1(qs[, keep_blank_values[, strict_parsing]])`

文字列引数として渡されたクエリ文字列 (`application/x-www-form-urlencoded` 型のデータ) を解釈します。解釈されたデータは名前と値のペアからなるリストです。

オプションの引数 *keep\_blank\_values* は、URL エンコードされたクエリ中で値の入っていないものを空文字列と見なすかどうかを示すフラグです。値が真であれば、値の入っていないフィールドは空文字列のままになります。標準では偽で、値の入っていないフィールドは無視され、そのフィールドはクエリに含まれていないものとして扱われます。

オプションの引数 *strict\_pasing* はパース時のエラーをどう扱うかを定めるフラグです。値が偽なら（標準の設定です）、エラーは暗黙のうちに無視されます。値が真なら、`ValueError` 例外が送出されます。

ペアのリストからクエリ文字列を生成する場合は、`urllib.urlencode()` 関数を使用します。

`parse_multipart(fp, pdict)`

(ファイル入力のための) `multipart/form-data` 型の入力を解釈します。引数は入力ファイルを示す *fp* と `Content-Type`: ヘッダ内の他のパラメタを含む辞書 *pdict* です。

`parse_qs()` と同じく辞書を返し、キーはフィールド名で、対応する値は各フィールドの値によるリストです。この関数は簡単に使うことができますが、数メガバイトのデータがアップロードされると考えられる場合にはあまり適していません — その場合、より柔軟性のある `FieldStorage` を代りに使ってください。

マルチパートデータがネストしている場合、各パートを解釈することはできないので注意してください — 代りに `FieldStorage` を使ってください。

`parse_header(string)`

(`Content-Type`: のような) MIME ヘッダを解釈し、ヘッダの主要値と各パラメタからなる辞書にします。

`test()`

メインプログラムから利用できる堅牢性テストを行う CGI スクリプトです。最小の HTTP ヘッダと HTML フォームからスクリプトに提供された全ての情報を書式化して出力します。

`print_environ()`

シェル変数を HTML に書式化して出力します。

`print_form(form)`

フォームを HTML に初期化して出力します。

`print_directory()`

現在のディレクトリを HTML に書式化して出力します。Format the current directory in HTML.

`print_environ_usage()`

意味のある (CGI の使う) 環境変数を HTML で出力します。

`escape(s[, quote])`

文字列 *s* 中の文字 '&'、'<'、および '>' を HTML で正しく表示できる文字配列に変換します。これらの文字が HTML に含まれるかもしれないようなテキストを出力する必要があるときに使ってください。オプションの引数 *quote* がの値が真であれば、二重引用符文字 ('"') も変換されます; この機能は、例えば `<A HREF="...">` といったような HTML の属性値を出力に含めるのに役立ちます。クオートされる値が単引用符か二重引用符、またはその両方を含む可能性がある場合は、代りに `xml.sax.saxutils.quoteattr()` 関数の利用を検討してください。



### 11.2.6 セキュリティへの配慮

重要なルールが一つあります: (関数 `os.system()` または `os.popen()`、またはその他の同様の機能によって) 外部プログラムを呼び出すなら、クライアントから受信した任意の文字列をシェルに渡していないことをよく確かめてください。これはよく知られているセキュリティホールであり、これによって Web のどこかにいる悪賢いハッカーが、だまされやすい CGI スクリプトに任意のシェルコマンドを実行させることができてしまいます。URL の一部やフィールド名でさえも信用してはいけません。CGI へのリクエストはあなたの作ったフォームから送信されるとは限らないからです！

安全な方法をとるためには、フォームから入力された文字をシェルに渡す場合、文字列が英数文字、ダッシュ、アンダースコア、およびピリオドだけを含むことを確認してください。

### 11.2.7 CGI スクリプトを UNIX システムにインストールする

あなたの使っている HTTP サーバのドキュメントを読んでください。そしてローカルシステムの管理者と一緒にどのディレクトリに CGI スクリプトをインストールすべきかを調べてください; 通常これはサーバのファイルシステムツリー内の 'cgi-bin' ディレクトリです。

あなたのスクリプトが "others" によって読み取り可能および実行可能であることを確認してください; UNIX ファイルモードは 8 進表記で 0755 です ('`chmod 0755 filename`' を使ってください)。スクリプトの最初の行の 1 カラム目が、#! で開始し、その後に Python インタプリタへのパス名が続いていることを確認してください。例えば:

```
#!/usr/local/bin/python
```

Python インタプリタが存在し、"others" によって実行可能であることを確かめてください。

あなたのスクリプトが読み書きする必要があるファイルが全て "others" によってそれぞれ読み出し可能、書き込み可能であることを確かめてください — それらのファイルモードは読み出し可能については 0644 で、書き込み可能については 0666 になるはずですが。これは、セキュリティ上の理由から、HTTP サーバがあなたのスクリプトを特権を全く持たないユーザ "nobody" の権限で実行するからです。この権限下では、誰でもが読める (書ける、実行できる) ファイルしか読む (書く、実行する) ことができません。スクリプト実行時のディレクトリや環境変数のセットもあなたがログインしたときの設定と異なります。特に、実行ファイルに対するシェルの検索パス (PATH) や Python のモジュール検索パス (PYTHONPATH) が何らかの値に設定されていることを期待してはいけません。

モジュールを Python の標準設定におけるモジュール検索パス上にないディレクトリからロードする必要がある場合、他のモジュールを取り込む前にスクリプト内で検索パスを変更することができます。例えば:

```
import sys
sys.path.insert(0, "/usr/home/joe/lib/python")
sys.path.insert(0, "/usr/local/lib/python")
```

(この方法では、最後に挿入されたディレクトリが最初に検索されます！)

非 UNIX システムにおける説明は変わるでしょう; あなたの使っている HTTP サーバのドキュメントを調べてください (普通は CGI スクリプトに関する節があります)。

### 11.2.8 CGI スクリプトをテストする

残念ながら、一般的に CGI スクリプトはコマンドラインから起動しようとしても動かず、コマンドラインから起動した場合には完璧に動作するスクリプトが、不思議なことにサーバからの起動では失敗すること

があります。しかし、スクリプトをコマンドラインから実行しなければならない理由が一つあります: もしスクリプトが文法エラーを含んでいれば、Python インタプリタはそのプログラムを全く実行しないため、HTTP サーバはほとんどの場合クライアントに謎めいたエラーを送信するからです。

スクリプトが構文エラーを含まないのにうまく動作しないなら、次の節に進み進むしかありません。

### 11.2.9 CGI スクリプトをデバッグする

何よりもまず、些細なインストール関連のエラーでないか確認してください— 上の CGI スクリプトのインストールに関する節を注意深く読むことで、大きく時間を節約できます。もしインストールの手続きを正しく理解しているか不安なら、このモジュールのファイル (‘cgi.py’) をコピーして、CGI スクリプトとしてインストールしてみてください。このファイルはスクリプトとして呼び出すと、スクリプトの実行環境とフォームの内容を HTML フォームに出力します。正しいモードなどをフォームに与えて、リクエストを送ってみてください。標準的な ‘cgi-bin’ ディレクトリにインストールされていれば、以下の形式でブラウザに URL を入力することで、リクエストを送ることができるはずです:

```
http://yourhostname/cgi-bin/cgi.py?name=Joe+Blow&addr=At+Home
```

もしタイプ 404 のエラーになるなら、サーバはスクリプトを見つけることができていません – おそらくあなたはスクリプトを別のディレクトリに入れる必要があるでしょう。他のエラーになるなら、先に進む前に解決しなければならないインストール上の問題があります。もしうまく書式化された実行環境の情報とフォーム内容 (この例では、各フィールドはフィールド名 “addr” に対して値 “At Home”、およびフィールド名 “name” に対して “Joe Blow”) になるなら、‘cgi.py’ スクリプトは正しくインストールされています。同じ操作をあなたの自作スクリプトに対して行えば、スクリプトをデバッグできるようになるはずです。次のステップでは cgi モジュールの test() 関数を呼び出すことになります: メインプログラムコードを以下の 1 行、

```
cgi.test()
```

と置き換えてください。この操作で ‘cgi.py’ ファイル自体をインストールした時と同じ結果を出力するはずです。

通常の Python スクリプトが unhandled exception 例外を送出する場合 (様々な理由: モジュール名のタイプミス、ファイルが開けなかった、など)、Python インタプリタはナイスなトレースバックを出力して終了します。Python インタプリタはあなたの CGI スクリプトが例外を送出した場合にも同様にするので、大抵はトレースバックが HTTP サーバのいずれかのログファイルに残るか、あるいはまったく無視されます。

幸運なことに、あなたが自作のスクリプトで 何らかの コードを実行できるようになったら、cgitb モジュールを使って簡単にトレースバックをブラウザに送信することができます。まだそうでないなら、以下の一行:

```
import cgitb; cgitb.enable()
```

をスクリプトの先頭に追加してください。そしてスクリプトを再度走らせます; 問題が発生すれば、クラッシュの原因を見出せるような詳細な報告を読むことができます。

cgitb モジュールのインポートに問題がありそうだと思うなら、(組み込みモジュールだけを使った) もっと堅牢なアプローチを取ることができます:

```
import sys
sys.stderr = sys.stdout
print "Content-Type: text/plain"
print
...your code here...
```

このコードは Python インタプリタがトレースバックを出力することに依存しています。出力のコンテンツ型はプレーンテキストに設定されており、全ての HTML 処理を無効にしています。スクリプトがうまく動作する場合、生の HTML コードがクライアントに表示されます。スクリプトが例外を送出する場合、最初の 2 行が出力された後、トレースバックが表示されます。HTML の解釈は行われないので、トレースバックを読むことができますでしょう。

### 11.2.10 よくある問題と解決法

- ほとんどの HTTP サーバはスクリプトの実行が完了するまで CGI からの出力をバッファします。このことは、スクリプトの実行中にクライアントが進捗状況報告を表示できないことを意味します。
- 上のインストールに関する説明を調べましょう。
- HTTP サーバのログファイルを調べましょう。(別のウィンドウで `'tail -f logfile'` を実行すると便利かもしれません！)
- 常に `'python script.py'` などとして、スクリプトが構文エラーでないか調べましょう。
- スクリプトに構文エラーがないなら、`'import cgitb; cgitb.enable()'` をスクリプトの先頭に追加してみましょう。
- 外部プログラムを起動するときには、スクリプトがそのプログラムを見つけられるようにしましょう。これは通常、絶対パス名を使うことを意味します — PATH は普通、あまり CGI スクリプトにとって便利でない値に設定されています。
- 外部のファイルを読み書きする際には、CGI スクリプトを動作させるときに使われる `userid` でファイルを読み書きできるようになっているか確認しましょう: `userid` は通常、Web サーバを動作させている `userid` か、Web サーバの `'suexec'` 機能で明示的に指定している `userid` になります。
- CGI スクリプトを `set-uid` モードにしてはいけません。これはほとんどのシステムで動作せず、セキュリティ上の信頼性也没有ありません。

## 11.3 cgitb — CGI スクリプトのトレースバック管理機構

2.2 で追加された仕様です。

`cgitb` モジュールでは、Python スクリプトのための特殊な例外処理を提供します。(実はこの説明は少し的外れです。このモジュールはもともと徹底的なトレースバック情報を CGI スクリプトで生成した HTML 内に表示するための設計されました。その後この情報を平文テキストでも表示できるように一般化されています。) このモジュールの有効化後に捕捉されない例外が生じた場合、詳細で書式化された報告が Web ブラウザに送信されます。この報告には各レベルにおけるソースコードの抜粋が示されたトレースバックと、現在動作している関数の引数やローカルな変数が収められており、問題のデバッグを助けます。オプションとして、この情報をブラウザに送信する代わりにファイルに保存することもできます。

この機能を有効化するためには、単に自作の CGI スクリプトの最初に以下の一行を追加します:

```
import cgitb; cgitb.enable()
```

`enable()` 関数のオプションは、報告をブラウザに表示するかどうかと、後で解析するためにファイルに報告をログ記録するかどうかを制御します。

```
enable([display[, logdir[, context[, format]]]])
```

この関数は、`sys.excepthook` を設定することで、インタプリタの標準の例外処理を `cgitb` モジュールに肩代わりさせるようにします。

オプションの引数 `display` は標準で 1 になっており、この値は 0 にしてトレースバックをブラウザに送らないように抑制することもできます。引数 `logdir` はログファイルを配置するディレクトリです。オプションの引数 `context` は、トレースバックの中で現在の行の周辺の何行を表示するかです; この値は標準で 5 です。オプションの引数 `format` が "html" の場合、出力は HTML に書式化されます。その他の値を指定すると平文テキストの出力を強制します。デフォルトの値は "html" です。

```
handler([info])
```

この関数は標準の設定 (ブラウザに報告を表示しますがファイルにはログを書き込みません) を使って例外を処理します。この関数は、例外を捕捉した際に `cgitb` を使って報告したい場合に使うことができます。オプションの `info` 引数は、例外の型、例外の値、トレースバックオブジェクトからなる 3 要素のタプルでなければなりません。これは `sys.exc_info()` によって返される値と全く同じです。 `info` 引数が与えられていない場合、現在の例外は `sys.exc_info()` から取得されます。

## 11.4 urllib — URL による任意のリソースへのアクセス

このモジュールはワールドワイドウェブ (World Wide Web) を介してデータを取り寄せるための高レベルのインタフェースを提供する。特に、関数 `urlopen()` は組み込み関数 `open()` と同様に動作し、ファイル名の代わりにファイルユニバーサルリソースロケータ (URL) を指定することができます。いくつかの制限があります — URL は読み出し専用でしか開けませんし、`seek` 操作を行うことはできません。

このモジュールでは、以下の public な関数を定義します。

```
urlopen(url[, data[, proxies]])
```

URL で表されるネットワーク上のオブジェクトを読み込み用に開きます。URL がスキーム識別子を持たないか、スキーム識別子が 'file:' である場合、ローカルシステムのファイルが (広範囲の改行サポートなしで) 開かれます。それ以外の場合はネットワーク上のどこかにあるサーバへのソケットを開きます。接続を作ることができないか、サーバがエラーコードを返した場合、例外 `IOError` が送出されます。全ての処理がうまくいけば、ファイル類似のオブジェクトが返されます。このオブジェクトは以下のメソッド: `read()`、`readline()`、`readlines()`、`fileno()`、`close()`、`info()` そして `geturl()` をサポートします。また、適切なイテレータプロトコルのサポートもしています。

`info()` および `geturl()` メソッドを除き、これらのメソッドはファイルオブジェクトと同じインタフェースを持っています — このマニュアルの [2.3.8](#) セクションを参照してください。(ですが、このオブジェクトは組み込みのファイルオブジェクトではないので、まれに真の組み込みファイルオブジェクトが必要な場所では使うことができません)

`info()` メソッドは開いた URL に関連付けられたメタ情報を含む `mimetypes.Message` クラスのインスタンスを返します。URL へのアクセスメソッドが HTTP である場合、メタ情報中のヘッダ情報はサーバが HTML ページを返すときに先頭に付加するヘッダ情報です (Content-Length および Content-Type を含みます)。アクセスメソッドが FTP の場合、ファイル取得リクエストにตอบสนองしてサーバがファイルの長さを返したときには (これは現在では普通になりましたが) Content-Length ヘッ

ダがメタ情報に含められます。Content-type ヘッダは MIME タイプが推測可能なときにメタ情報に含められます。アクセスメソッドがローカルファイルの場合、返されるヘッダ情報にはファイルの最終更新日時を表す Date エントリ、ファイルのサイズを示す Content-Length エントリ、そして推測されるファイル形式の Content-Type エントリが含まれます。mimetools モジュールを参照してください。

geturl() メソッドはページの実際の URL を返します。場合によっては、HTTP サーバはクライアントの要求を他の URL に振り向け (redirect、リダイレクト) します。関数 urlopen() はユーザに対してリダイレクトを透過的に行いますが、呼び出し側にとってクライアントがどの URL にリダイレクトされたかを知りたいときがあります。geturl() メソッドを使うと、このリダイレクトされた URL を取得できます。

url に 'http:' スキーム識別子を使う場合、data 引数を与えて POST 形式のリクエストを行うことができます (通常リクエストの形式は GET です)。引数 data は標準の application/x-www-form-urlencoded 形式でなければなりません; 以下の urlencode() 関数を参照してください。

urlopen() 関数は認証を必要としないプロキシ (proxy) に対して透過的に動作します。UNIX または Windows 環境では、Python を起動する前に、環境変数 http\_proxy、ftp\_proxy、および gopher\_proxy にそれぞれのプロキシサーバを指定する URL を設定してください。例えば ('%' はコマンドプロンプトです):

```
% http_proxy="http://www.someproxy.com:3128"
% export http_proxy
% python
...
```

Windows 環境では、プロキシを指定する環境変数が設定されていない場合、プロキシの設定値はレジストリの Internet Settings セクションから取得されます。

Macintosh 環境では、urlopen() は「インターネットの設定」(Internet Config) からプロキシ情報を取得します。

別の方法として、オプション引数 proxies を使って明示的にプロキシを設定することができます。この引数はスキーム名をプロキシの URL にマップする辞書型のオブジェクトでなくてはなりません。空の辞書を指定するとプロキシを使いません。None (デフォルトの値です) を指定すると、上で述べたように環境変数で指定されたプロキシ設定を使います。例えば:

```
# http://www.someproxy.com:3128 を http プロキシに使う
proxies = proxies={'http': 'http://www.someproxy.com:3128'}
filehandle = urllib.urlopen(some_url, proxies=proxies)
# プロキシを使わない
filehandle = urllib.urlopen(some_url, proxies={})
# 環境変数からプロキシを使う - 両方の表記とも同じ意味です。
filehandle = urllib.urlopen(some_url, proxies=None)
filehandle = urllib.urlopen(some_url)
```

(訳注: 上記と矛盾する内容です。おそらく旧バージョンのドキュメントです) 関数 urlopen() は明示的なプロキシ指定をサポートしていません。環境変数のプロキシ設定を上書きしたい場合には URLOpener を使うか、FancyURLOpener などのサブクラスを使ってください。

認証を必要とするプロキシは現在のところサポートされていません。これは実装上の制限 (implementation limitation) と考えています。

2.3 で変更された仕様: proxies のサポートを追加しました。

urlretrieve(url[, filename[, reporthook[, data]]])

URL で表されるネットワーク上のオブジェクトを、必要に応じてローカルなファイルにコピーしま



す。URL がローカルなファイルを指定していたり、オブジェクトのコピーが正しくキャッシュされていれば、そのオブジェクトはコピーされません。タプル (*filename*, *headers*) を返し、*filename* はローカルで見つかったオブジェクトに対するファイル名で、*headers* は `urlopen()` が返した (おそらくキャッシュされているリモートの) オブジェクトに `info()` を適用して得られるものになります。`urlopen()` と同じ例外を送出します。

2 つめの引数がある場合、オブジェクトのコピー先となるファイルの位置を指定します (もしなければ、ファイルの場所は一時ファイル (tmpfile) の置き場になり、名前は適当につけられます)。3 つめの引数がある場合、ネットワークとの接続が確立された際に一度呼び出され、以降データのブロックが読み出されるたびに呼び出されるフック関数 (hook function) を指定します。フック関数には 3 つの引数が渡されます; これまで転送されたブロック数のカウント、バイト単位で表されたブロックサイズ、ファイルの総サイズです。3 つ目のファイルの総サイズは、ファイル取得の際の応答時にファイルサイズを返さない古い FTP サーバでは -1 になります。

*url* が 'http:' スキーム識別子を使っていた場合、オプション引数 *data* を与えることで POST リクエストを行うよう指定することができます (通常リクエストの形式は GET です)。*data* 引数は標準の application/x-www-form-urlencoded 形式でなくてはなりません; 以下の `urlencode()` 関数を参照してください。

### `_urlopener`

パブリック関数 `urlopen()` および `urlretrieve()` は `FancyURLOpener` クラスのインスタンスを生成します。インスタンスは要求された動作に応じて使用されます。この機能をオーバーライドするために、プログラマは `URLOpener` または `FancyURLOpener` のサブクラスを作り、そのクラスから生成したインスタンスを変数 `urllib._urlopener` に代入した後、呼び出したい関数を呼ぶことができます。例えば、アプリケーションが `URLOpener` が定義しているのとは異なった User-Agent: ヘッダを指定したい場合があるかもしれません。この機能は以下のコードで実現できます:

```
import urllib

class AppURLOpener(urllib.FancyURLOpener):
    def __init__(self, *args):
        self.version = "App/1.7"
        urllib.FancyURLOpener.__init__(self, *args)

urllib._urlopener = AppURLOpener()
```

### `urlcleanup()`

以前の `urlretrieve()` で生成された可能性のあるキャッシュを消去します。

### `quote(string[, safe])`

*string* に含まれる特殊文字を '%xx' エスケープで置換 (`quote`) します。アルファベット、数字、および文字 '`_.-`' は `quote` 処理を行いません。オプションのパラメタ *safe* は `quote` 処理しない追加の文字を指定します — デフォルトの値は '/' です。

例: `quote('/~connolly/')` は `'/%7Econnolly/'` になります。

### `quote_plus(string[, safe])`

`quote()` と似ていますが、加えて空白文字をプラス記号 ("+") に置き換えます。これは HTML フォームの値を `quote` 処理する際に必要な機能です。もとの文字列におけるプラス記号は *safe* に含まれていない限りエスケープ置換されます。上と同様に、*safe* のデフォルトの値は '/' です。

### `unquote(string)`

'%xx' エスケープをエスケープが表す 1 文字に置き換えます。

例: `unquote('/~connolly/')` は `'/~connolly/'` になります。



`unquote_plus(string)`

`unquote()` と似ていますが、加えてプラス記号を空白文字に置き換えます。これは `quote` 処理された HTML フォームの値を元に戻すのに必要な機能です。

`urlencode(query[, doseq])`

マップ型オブジェクト、または 2 つの要素をもったタプルからなる配列を、“URL にエンコードされた (url-encoded)” に変換して、上述の `urlopen()` のオプション引数 *data* に適した形式にします。この関数はフォームのフィールド値でできた辞書を POST 型のリクエストに渡すときに便利です。返される文字列は *key=value* のペアを ‘&’ で区切った配列で、*key* と *value* の双方は上の `quote_plus()` で `quote` 処理されます。オプションのパラメタ *doseq* が与えられていて、その評価結果が真であった場合、配列 *doseq* の個々の要素について *key=value* のペアが生成されます。2 つの要素をもったタプルからなる配列が引数 *query* として使われた場合、各タプルの最初の値が *key* で、2 番目の値が *value* になります。このときエンコードされた文字列中のパラメタの順番は配列中のタプルの順番と同じになります。cgi モジュールでは、関数 `parse_qs()` および `parse_qsl()` を提供しており、クエリ文字列を解析して Python のデータ構造にするのに利用できます。

`pathname2url(path)`

ローカルシステムにおける記法で表されたパス名 *path* を、URL におけるパス部分の形式に変換します。この関数は完全な URL を生成するわけではありません。返される値は常に `quote()` を使って `quote` 処理されたものになります。

`url2pathname(path)`

URL のパスの部分 *path* をエンコードされた URL の形式からローカルシステムにおけるパス記法に変換します。この関数は *path* をデコードするために `unquote()` を使います。

`class URLOpener([proxies[, **x509]])`

URL をオープンし、読み出すためのクラスの基礎クラス (base class) です。‘http:’、‘ftp:’、‘gopher:’ または ‘file:’ 以外のスキームを使ったオブジェクトのオープンをサポートしたいのでないかぎり、`FancyURLOpener` を使おうと思うことになるでしょう。

デフォルトでは、`URLOpener` クラスは User-Agent: ヘッダとして ‘urllib/VVV’ を送信します。ここで VVV は `urllib` のバージョン番号です。アプリケーションで独自の User-Agent: ヘッダを送信したい場合は、`URLOpener` かまたは `FancyURLOpener` のサブクラスを作成し、`open()` メソッドを呼び出す前にインスタンス属性 *version* を適切な文字列値に設定することで行うことができます。

オプションのパラメタ *proxies* はスキーム名をプロキシの URL にマップする辞書でなくてはなりません。空の辞書はプロキシ機能を完全にオフにします。デフォルトの値は `None` で、この場合、`urlopen()` の定義で述べたように、プロキシを設定する環境変数が存在するならそれを使います。

追加のキーワードパラメタは *x509* に集められますが、これは ‘https:’ スキームによる認証に使われます。キーワード引数 *key\_file* および *cert\_file* がサポートされています。実際に ‘https:’ 形式の URL からリソースを取得するには両方の引数が必要です。

`class FancyURLOpener(...)`

`FancyURLOpener` は `URLOpener` のサブクラスで、以下の HTTP レスポンスコード: 301、302、303、307、および 401 を取り扱う機能を提供します。レスポンスコード 30x に対しては、Location: ヘッダを使って実際の URL を取得します。レスポンスコード 401 (認証が要求されていることを示す) に対しては、ベーシック認証 (basic HTTP authentication) が行われます。レスポンスコード 30x に対しては、最大で *maxtries* 属性に指定された数だけ再帰呼び出しを行うようになっています。この値はデフォルトで 10 です。

注意: RFC 2616 によると、POST 要求に対する 301 および 302 応答はユーザの承認無しに自動的にリダイレクトしてはなりません。実際は、これらの応答に対して自動リダイレクトを許すブラウザでは POST を GET に変更しており、`urllib` でもこの動作を再現します。

コンストラクタに与えるパラメタは `URLopener` と同じです。

注意: 基本的な HTTP 認証を行う際、`FancyURLopener` インスタンスは `prompt_user_passwd()` メソッドを呼び出します。このメソッドはデフォルトでは実行を制御している端末上で認証に必要な情報を要求するように実装されています。必要ならば、このクラスのサブクラスにおいてより適切な動作をサポートするために `prompt_user_passwd()` メソッドをオーバーライドしてもかまいません。

制限:

- 現在のところ、以下のプロトコルだけがサポートされています: HTTP、(バージョン 0.9 および 1.0)、Gopher (Gopher+ を除く)、FTP、およびローカルファイル。
- `urlretrieve()` のキャッシュ機能は、有効期限ヘッダ (Expiration time header) を正しく処理できるようにハックするための時間を取れるまで、無効にしてあります。
- ある URL がキャッシュにあるかどうか調べるような関数があればと思っています。。
- 後方互換性のため、URL がローカルシステム上のファイルを指しているように見えるにも関わらずファイルを開くことができなければ、URL は FTP プロトコルを使って再解釈されます。この機能は時として混乱を招くエラーメッセージを引き起こします。
- 関数 `urlopen()` および `urlretrieve()` は、ネットワーク接続が確立されるまでの間、一定でない長さの遅延を引き起こすことがあります。このことは、これらの関数を使ってインタラクティブな Web クライアントを構築するのはスレッドなしには難しいことを意味します。
- `urlopen()` または `urlretrieve()` が返すデータはサーバが返す生のデータです。このデータはバイナリデータ (例えば画像データ)、生テキスト (plain text)、または (例えば) HTML でもかまいません。HTTP プロトコルはリプライヘッダ (reply header) にデータのタイプに関する情報を返します。タイプは Content-Type: ヘッダを見ることで推測できます。  
Gopher プロトコルでは、データのタイプに関する情報は URL にエンコードされます; これを展開することは簡単ではありません。返されたデータが HTML であれば、`htmllib` を使ってパースすることができます。
- このモジュールは認証を必要とするプロキシをサポートしません。将来実装されるかもしれません。
- `urllib` モジュールは URL 文字列を解釈したり構築したりする (ドキュメント化されていない) ルーチンを含んでいますが、URL を操作するためのインタフェースとしては、`urlparse` モジュールをお勧めします。

### 11.4.1 URLopener オブジェクト

`URLopener` および `FancyURLopener` クラスのオブジェクトは以下の属性を持っています。

`open(fullurl[, data])`

適切なプロトコルを使って `fullurl` を開きます。このメソッドはキャッシュとプロキシ情報を設定し、その後適切な `open` メソッドを入力引数つきで呼び出します。認識できないスキームが与えられた場合、`open_unknown()` が呼び出されます。 `data` 引数は `urlopen()` の引数 `data` と同じ意味を持っています。

`open_unknown(fullurl[, data])`

オーバーライド可能な、未知のタイプの URL を開くためのインタフェースです。

`retrieve(url[, filename[, reporthook[, data]]])`

`url` のコンテンツを取得し、`filename` に書き込みます。返り値はタプルで、ローカルシステムにおけるファイル名と、応答ヘッダ (URL がリモートを指している場合) または `None` (URL がローカルを指している場合) からなります。呼び出し側の処理はその後 `filename` を開いて内容を読み出さなくてはなりません。`filename` が与えられており、かつ URL がローカルシステム上のファイルを示しているばあい、入力ファイル名が返されます。URL がローカルのファイルを示しておらず、かつ `filename` が与えられていない場合、ファイル名は入力 URL の最後のパス構成要素につけられた拡張子と同じ拡張子を `tempfile.mktemp()` につけたものになります。`reporthook` を与える場合、この変数は3つの数値パラメタを受け取る関数でなくてはなりません。この関数はデータの塊 (chunk) がネットワークから読み込まれるたびに呼び出されます。ローカルの URL を与えた場合 `reporthook` は無視されます。

`url` が 'http:' スキーム識別子を使っている場合、オプションの引数 `data` を与えて POST リクエストを行うよう指定できます (通常のリクエストの形式は GET です)。引数 `data` は標準の `application/x-www-form-urlencoded` 形式でなくてはなりません; 上の `urlencode()` を参照して下さい。

#### version

URL をオープンするオブジェクトのユーザエージェントを指定する変数です。`urllib` を特定のユーザエージェントであるとサーバに通知するには、サブクラスの中でこの値をクラス変数として値を設定するか、コンストラクタの中でベースクラスを呼び出す前に値を設定してください。

`FancyURLopener` クラスはオーバーライド可能な追加のメソッドを提供しており、適切な振る舞いをさせることができます:

`prompt_user_passwd(host, realm)`

指定されたセキュリティ領域 (security realm) 下にある与えられたホストにおいて、ユーザ認証に必要な情報を返すための関数です。この関数が返す値は (`user`, `password`)、からなるタプルでなくてはなりません。値はベーシック認証 (basic authentication) で使われます。

このクラスでの実装では、端末に情報を入力するようプロンプトを出します; ローカルの環境において適切な形に対話型モデルを使うには、このメソッドをオーバーライドしなければなりません。

### 11.4.2 使用例

以下は 'GET' メソッドを使ってパラメタを含む URL を取得するセッションの例です:

```
>>> import urllib
>>> params = urllib.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> f = urllib.urlopen("http://www.musi-cal.com/cgi-bin/query?%s" % params)
>>> print f.read()
```

以下は 'POST' メソッドを代わりに使った例です:

```
>>> import urllib
>>> params = urllib.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> f = urllib.urlopen("http://www.musi-cal.com/cgi-bin/query", params)
>>> print f.read()
```

以下の例では、環境変数による設定内容に対して上書きする形で HTTP プロキシを明示的に設定しています:

```
>>> import urllib
>>> proxies = {'http': 'http://proxy.example.com:8080/'}
>>> opener = urllib.FancyURLopener(proxies)
>>> f = opener.open("http://www.python.org")
>>> f.read()
```

以下の例では、環境変数による設定内容に対して上書きする形で、まったくプロキシを使わないよう設定しています:

```
>>> import urllib
>>> opener = urllib.FancyURLopener({})
>>> f = opener.open("http://www.python.org/")
>>> f.read()
```

## 11.5 urllib2 — URL を開くための拡張可能なライブラリ

urllib2 モジュールは基本的な認証、暗号化認証、リダイレクションその他の介在する複雑なアクセス環境において (大抵は HTTP で) URL を開くための関数とクラスを定義します。

urllib2 モジュールでは以下の関数を定義しています:

**urlopen**(*url*[, *data*])

URL *url* を開きます。*url* は文字列でも Request オブジェクトでもかまいません (現時点では、コードの中で引数が本当に Request クラスのインスタンスまたは Request のサブクラスのインスタンスであるかどうかチェックされます)。

*data* は文字列で、サーバに送信する追加のデータを指定します。HTTP リクエストは *data* をサポートする唯一のリクエスト形式ですが、ここでは *data* は例えば `urllib.urlencode()` が返すような `application/x-www-form-urlencoded` 形式でエンコードされたバッファでなくてはなりません。

この関数は以下の 2 つのメソッドを持つファイル類似のオブジェクトを返します:

- `geturl()` — 取得されたリソースの URL を返します。
- `info()` — 取得されたページのメタ情報を辞書形式のオブジェクトで返します。

エラーが発生した場合 `URLError` を送出します。

**install\_opener**(*opener*)

標準で URL を開くオブジェクトとして `OpenerDirector` のインスタンスをインストールします。このコードは引数が本当に `OpenerDirector` のインスタンスであるかどうかはチェックしないので、適切なインタフェースを持ったクラスは何でも動作します。

**build\_opener**( [*handler*, ... ] )

与えられた順番に URL ハンドラを連鎖させる `OpenerDirector` のインスタンスを返します。*handler* は `BaseHandler` または `BaseHandler` のサブクラスのインスタンスのどちらかです (どちらの場合も、コンストラクトは引数無しで呼び出せるようになっていなければなりません)。以下のクラス:

`ProxyHandler`, `UnknownHandler`, `HTTPHandler`, `HTTPDefaultErrorHandler`, `HTTPRedirectHandler`, `FTPHandler`, `FileHandler`

については、そのクラスのインスタンスか、そのサブクラスのインスタンスが *handler* に含まれていない限り、*handler* よりも先に連鎖します。

Python が SSL をサポートするように設定してインストールされている場合 (`socket.ssl()` が存在する場合)、`HTTPSHandler` も追加されます。

Python 2.3 からは、`BaseHandler` サブクラスでも `handler_order` メンバ変数を変更して、ハンドラリスト内での場所を変更できるようになりました。現在全てのハンドラの `handler_order` は 500 ですが、例外として `ProxyHandler` では 100 になっています。

状況に応じて、以下の例外が送出されます：

#### **exception URLError**

ハンドラが何らかの問題に遭遇した場合、この例外 (またはこの例外から導出された例外) を送出します。この例外は `IOError` のサブクラスです。

#### **exception HTTPError**

`URLError` のサブクラスです。このオブジェクトは例外でないファイル類似のオブジェクトとして返り値に使うことができます (`urlopen()` が返すのと同じものです)。この機能は、例えばサーバからの認証リクエストのように、変わった HTTP エラーを処理するのに役立ちます。

#### **exception GopherError**

`URLError` のサブクラスです。この例外は Gopher ハンドラによって送出されます。

以下のクラスが提供されています：

**class Request** (*url* [, *data* [, *headers* ]])

このクラスは URL リクエストを抽象化したものです。

*url* は有効な URL を指す文字列でなくてはなりません。 *data* の詳細については `add_data()` の記述を見てください。 *headers* は辞書でなくてはなりません。 この辞書は `add_header()` を辞書のキーおよび値を引数として呼び出した時と同じように扱われます。

**class OpenerDirector** ()

`OpenerDirector` クラスは、`BaseHandler` の連鎖的に呼び出して URL を開きます。このクラスはハンドラをどのように連鎖させるか、またどのようにエラーをリカバリするかを管理します。

**class BaseHandler** ()

このクラスはハンドラ連鎖に登録される全てのハンドラがベースとしているクラスです – このクラスでは登録のための単純なメカニズムだけを扱います。

**class HTTPDefaultErrorHandler** ()

HTTP エラー応答のための標準のハンドラを定義します; 全てのレスポンスに対して、例外 `HTTPError` を送出します。

**class HTTPRedirectHandler** ()

リダイレクションを扱うクラスです。

**class ProxyHandler** ([*proxies* ])

このクラスはプロキシを通過してリクエストを送らせます。引数 *proxies* を与える場合、プロトコル名からプロキシの URL へ対応付ける辞書でなくてはなりません。標準では、プロキシのリストを環境変数 `protocol_proxy` から読み出します。

**class HTTPPasswordMgr** ()

(*realm*, *uri*) -> (*user*, *password*) の対応付けデータベースを保持します。

**class HTTPPasswordMgrWithDefaultRealm** ()

(*realm*, *uri*) -> (*user*, *password*) の対応付けデータベースを保持します。レルム `None` はその他諸々のレルムを表し、他のレルムが該当しない場合に検索されます。

**class AbstractBasicAuthHandler** ([*password\_mgr* ])

このクラスは HTTP 認証を補助するための混ぜ込みクラス (mixin class) です。遠隔ホストとプロキシの両方に対応しています。 *password\_mgr* を与える場合、`HTTPPasswordMgr` と互換性がなければ



なりません; 互換性のためにサポートしなければならないインタフェースについての情報はセクション 11.5.6 を参照してください。

```
class HTTPBasicAuthHandler([password_mgr])
```

遠隔ホストとの間での認証を扱います。password\_mgr を与える場合、HTTPPasswordMgr と互換性がなければなりません; 互換性のためにサポートしなければならないインタフェースについての情報はセクション 11.5.6 を参照してください。

```
class ProxyBasicAuthHandler([password_mgr])
```

プロキシとの間での認証を扱います。password\_mgr を与える場合、HTTPPasswordMgr と互換性がなければなりません; 互換性のためにサポートしなければならないインタフェースについての情報はセクション 11.5.6 を参照してください。

```
class AbstractDigestAuthHandler([password_mgr])
```

このクラスは HTTP 認証を補助するための混ぜ込みクラス (mixin class) です。遠隔ホストとプロキシの両方に対応しています。password\_mgr を与える場合、HTTPPasswordMgr と互換性がなければなりません; 互換性のためにサポートしなければならないインタフェースについての情報はセクション 11.5.6 を参照してください。

```
class HTTPDigestAuthHandler([password_mgr])
```

遠隔ホストとの間での認証を扱います。password\_mgr を与える場合、HTTPPasswordMgr と互換性がなければなりません; 互換性のためにサポートしなければならないインタフェースについての情報はセクション 11.5.6 を参照してください。

```
class ProxyDigestAuthHandler([password_mgr])
```

プロキシとの間での認証を扱います。password\_mgr を与える場合、HTTPPasswordMgr と互換性がなければなりません; 互換性のためにサポートしなければならないインタフェースについての情報はセクション 11.5.6 を参照してください。

```
class HTTPHandler()
```

HTTP の URL を開きます。

```
class HTTPSHandler()
```

HTTPS の URL を開きます。

```
class FileHandler()
```

ローカルファイルを開きます。

```
class FTPHandler()
```

FTP の URL を開きます。

```
class CacheFTPHandler()
```

FTP の URL を開きます。遅延を最小限にするために、開かれている FTP 接続に対するキャッシュを保持します。

```
class GopherHandler()
```

gopher の URL を開きます。

```
class UnknownHandler()
```

その他諸々のためのクラスで、未知のプロトコルの URL を開きます。

### 11.5.1 Request オブジェクト

以下のメソッドは Request の全ての公開インタフェースを記述します。従ってサブクラスではこれら全てのメソッドをオーバーライドしなければなりません。

```
add_data(data)
```



Request のデータを *data* に設定します。この値は HTTP ハンドラ以外のハンドラでは無視されません。HTTP ハンドラでは、データは `application/x-www-form-urlencoded` でエンコードされたバッファでなくてはなりません。このメソッドを使うとリクエストの形式が GET POST に変更されます。

`get_method()`

HTTP リクエストメソッドを示す文字列を返します。このメソッドは HTTP リクエストだけに対して意味があり、現状では常に ("GET", "POST") のうちのいずれかの値をとります。

`has_data()`

インスタンスが `None` でないデータを持つかどうかを返します。

`get_data()`

インスタンスのデータを返します。

`add_header(key, val)`

リクエストに新たなヘッダを追加します。ヘッダは HTTP ハンドラ以外のハンドラでは無視されません。HTTP ハンドラでは、引数はサーバに送信されるヘッダのリストに追加されます。同じ名前を持つヘッダを 2 つ以上持つことはできず、*key* の衝突が生じた場合、後で追加したヘッダが前に追加したヘッダを上書きします。現時点では、この機能は HTTP の機能を損ねることはありません。というのは、複数回呼び出したときに意味を持つようなヘッダには、どれもただ一つのヘッダを使って同じ機能を果たすための (ヘッダ特有の) 方法があるからです。

`get_full_url()`

コンストラクタで与えられた URL を返します。

`get_type()`

URL のタイプ — いわゆるスキーム (scheme) — を返します。

`get_host()`

接続を行う先のホスト名を返します。

`get_selector()`

セクタ — サーバに送られる URL の一部分 — を返します。

`set_proxy(host, type)`

リクエストがプロキシサーバを経由するように準備します。*host* および *type* はインスタンスのものと設定と置き換えられます。インスタンスのセクタはコンストラクタに与えたもともとの URL になります。

## 11.5.2 OpenerDirector オブジェクト

OpenerDirector インスタンスは以下のメソッドを持っています:

`add_handler(handler)`

*handler* は `BaseHandler` のインスタンスでなければなりません。以下のメソッドを使った検索が行われ、URL を取り扱うことが可能なハンドラの連鎖が追加されます。

• `protocol_open()` — ハンドラが *protocol* の URL を開く方法を知っているかどうかを調べます。

• `protocol_error_type()` — ハンドラが *protocol* から返される *type* エラーの扱い方を知っているかどうかを調べます。

`close()`

循環参照を明示的にやめさせて、全てのハンドラを削除します。OpenerDirector は登録されているハンドラを参照する必要があるため、ハンドラは自分呼び出す OpenerDirector を参照するため、循環参照が生まれます。最近のバージョンの Python では循環参照を修正しますが、たまたま明示的に循環参照をやめさせた方がよいことがあります。

`open(url[, data])`

与えられた *url* (リクエストオブジェクトでも文字列でもかまいません) を開きます。オプションとして *data* を与えることができます。引数、返回值、および送出される例外は `urlopen()` と同じです (`urlopen()` の場合、標準でインストールされている `OpenerDirector` の `open()` メソッドを呼び出します)。

`error(proto[, arg[, ...]])`

与えられたプロトコルにおけるエラーを処理します。このメソッドは与えられたプロトコルにおける登録済みのエラーハンドラを (プロトコル固有の) 引数で呼び出します。HTTP プロトコルは特殊なケースで、特定のエラーハンドラを選び出すのに HTTP レスポンスコードを使います; ハンドラクラス `http_error_*()` メソッドを参照してください。

返回值および送出される例外は `urlopen()` と同じものです。

### 11.5.3 BaseHandler オブジェクト

`BaseHandler` オブジェクトは直接的に役に立つ 2 つのメソッドと、その他として導出クラスで使われることを想定したメソッドを提供します。以下は直接的に使うためのメソッドです:

`add_parent(director)`

親オブジェクトとして、`director` を追加します。

`close()`

全ての親オブジェクトを削除します。

以下のメンバおよびメソッドは `BaseHandler` から導出されたクラスでのみ使われます:

`parent`

有効な `OpenerDirector` です。この値は違うプロトコルを使って URL を開く場合やエラーを処理する際に使われます。

`default_open(req)`

このメソッドは `BaseHandler` では定義されていません。しかし、全ての URL をキャッチさせたいなら、サブクラスで定義する必要があります。

このメソッドが定義されていた場合、`OpenerDirector` から呼び出されます。このメソッドは `OpenerDirector` のメソッド `open()` が返す値について記述されているようなファイル類似のオブジェクトか、`None` を返さなくてはなりません。このメソッドが送出する例外は、真に例外的なことが起きない限り、`URLError` を送出しなければなりません (例えば、`MemoryError` を `URLError` をマッピングしてはいけません)。

このメソッドはプロトコル固有のオープンメソッドが呼び出される前に呼び出されます。

`protocol_open(req)`

このメソッドは `BaseHandler` では定義されていません。しかしプロトコルの指定された URL をキャッチしたいなら、サブクラスで定義する必要があります。

このメソッドが定義されていた場合、`OpenerDirector` から呼び出されます。戻り値は `default_open` と同じでなければなりません。

`unknown_open(req)`

このメソッドは `BaseHandler` では定義されていません。しかし URL を開くための特定のハンドラが登録されていないような URL をキャッチしたいなら、サブクラスで定義する必要があります。

このメソッドが定義されていた場合、`OpenerDirector` から呼び出されます。戻り値は `default_open` と同じでなければなりません。

`http_error_default(req, fp, code, msg, hdrs)`

このメソッドは `BaseHandler` では定義されていません。しかしその他の処理されなかった HTTP エラーを処理する機能をもたせたいなら、サブクラスで定義する必要があります。このメソッドはエラーに遭遇した `OpenerDirector` から自動的に呼び出されます。その他の状況では普通呼び出すべきではありません。

`req` は `Request` オブジェクトで、`fp` は HTTP エラー本体を読み出せるようなファイル類似のオブジェクトになります。`code` は 3 桁の 10 進数からなるエラーコードで、`msg` ユーザ向けのエラーコード解説です。`hdrs` はエラー応答のヘッダをマップしたオブジェクトです。

返される値および送出される例外は `urlopen()` と同じものでなければなりません。

`http_error_nnn(req, fp, code, msg, hdrs)`

`nnn` は 3 桁の 10 進数からなる HTTP エラーコードでなくてはなりません。このメソッドも `BaseHandler` では定義されていませんが、サブクラスのインスタンスで定義されていた場合、エラーコード `nnn` の HTTP エラーが発生した際に呼び出されます。

特定の HTTP エラーに対する処理を行うためには、このメソッドをサブクラスでオーバーライドする必要があります。

引数、返される値、および送出される例外は `http_error_default()` と同じものでなければなりません。

#### 11.5.4 HTTPRedirectHandler オブジェクト

注意: HTTP リダイレクトによっては、このモジュールのクライアントコード側での処理を必要とします。その場合、`HTTPError` が送出されます。様々なリダイレクトコードの厳密な意味に関する詳細は RFC 2616 を参照してください。

`redirect_request(req, fp, code, msg, hdrs)`

リダイレクトの通知に応じて、`Request` または `None` を返します。このメソッドは `http_error_30*()` メソッドにおいて、リダイレクトの通知をサーバから受信した際に、デフォルトの実装として呼び出されます。リダイレクトを起こす場合、新たな `Request` を生成して、`http_error_30*()` がリダイレクトを実行できるようにします。そうでない場合、他の `Handler` のいずれにもこの URL を処理させたくなければ `HTTPError` を送出し、リダイレクト処理を行うことはできないが他の `Handler` なら可能かもしれない場合には `None` を返します。

注意: このメソッドのデフォルトの実装は、RFC 2616 に厳密に従ったものではありません。RFC 2616 では、POST リクエストに対する 301 および 302 応答が、ユーザの承認なく自動的にリダイレクトされてはならないと述べています。現実には、ブラウザは POST を GET に変更することで、これらの応答に対して自動的にリダイレクトを行えるようにしています。デフォルトの実装でも、この挙動を再現しています。

`http_error_301(req, fp, code, msg, hdrs)`

`Location`: URL にリダイレクトします。このメソッドは HTTP における ‘moved permanently’ レスポンスを取得した際に親オブジェクトとなる `OpenerDirector` によって呼び出されます。

`http_error_302(req, fp, code, msg, hdrs)`

`http_error_301()` と同じですが、‘found’ レスポンスに対して呼び出されます。

`http_error_303(req, fp, code, msg, hdrs)`

`http_error_301()` と同じですが、‘see other’ レスポンスに対して呼び出されます。

`http_error_307(req, fp, code, msg, hdrs)`

`http_error_301()` と同じですが、‘temporary redirect’ レスポンスに対して呼び出されます。

### 11.5.5 ProxyHandler オブジェクト

`protocol_open(request)`

ProxyHandler は、コンストラクタで与えた辞書 *proxies* にプロキシが設定されているような *protocol* 全てについて、メソッド *protocol\_open()* を持つことになります。このメソッドは *request.set\_proxy()* を呼び出して、リクエストがプロキシを通過できるように修正します。その後連鎖するハンドラの中から次のハンドラを呼び出して実際にプロトコルを実行します。

### 11.5.6 HTTPPasswordMgr オブジェクト

以下のメソッドは HTTPPasswordMgr および HTTPPasswordMgrWithDefaultRealm オブジェクトで利用できます。

`add_password(realm, uri, user, passwd)`

*uri* は単一の URI でも複数の URI からなる配列でもかまいません。*realm*、*user* および *passwd* は文字列でなくてはなりません。このメソッドによって、*realm* と与えられた URI の上位 URI に対して (*user*, *passwd*) が認証トークンとして使われるようになります。

`find_user_password(realm, authuri)`

与えられたレルムおよび URI に対するユーザ名またはパスワードがあればそれを取得します。該当するユーザ名/パスワードが存在しない場合、このメソッドは (None, None) を返します。

HTTPPasswordMgrWithDefaultRealm オブジェクトでは、与えられた *realm* に対して該当するユーザ名/パスワードが存在しない場合、レルム None が検索されます。

### 11.5.7 AbstractBasicAuthHandler オブジェクト

`handle_authentication_request(authreq, host, req, headers)`

ユーザ名/パスワードを取得し、再度サーバへのリクエストを試みることで、サーバからの認証リクエストを処理します。*authreq* はリクエストにおいてレルムに関する情報が含まれているヘッダの名前、*host* は認証を行う対象のホスト名、*req* は (失敗した) Request オブジェクト、そして *headers* はエラーヘッダでなくてはなりません。

### 11.5.8 HTTPBasicAuthHandler オブジェクト

`http_error_401(req, fp, code, msg, hdrs)`

認証情報がある場合、認証情報付きで再度リクエストを試みます。

### 11.5.9 ProxyBasicAuthHandler オブジェクト

`http_error_407(req, fp, code, msg, hdrs)`

認証情報がある場合、認証情報付きで再度リクエストを試みます。

### 11.5.10 AbstractDigestAuthHandler オブジェクト

`handle_authentication_request(authreq, host, req, headers)`

*authreq* はリクエストにおいてレルムに関する情報が含まれているヘッダの名前、*host* は認証を行う対象のホスト名、*req* は (失敗した) Request オブジェクト、そして *headers* はエラーヘッダでなくてはなりません。

### 11.5.11 HTTPDigestAuthHandler オブジェクト

`http_error_401(req, fp, code, msg, hdrs)`

認証情報がある場合、認証情報付きで再度リクエストを試みます。

### 11.5.12 ProxyDigestAuthHandler オブジェクト

`http_error_407(req, fp, code, msg, hdrs)`

認証情報がある場合、認証情報付きで再度リクエストを試みます。

### 11.5.13 HTTPHandler オブジェクト

`http_open(req)`

HTTP リクエストを送ります。`req.has_data()` に応じて、GET または POST のどちらでも送ることができます。

### 11.5.14 HTTPSHandler オブジェクト

`https_open(req)`

HTTPS リクエストを送ります。`req.has_data()` に応じて、GET または POST のどちらでも送ることができます。

### 11.5.15 FileHandler オブジェクト

`file_open(req)`

ホスト名がない場合、またはホスト名が `'localhost'` の場合にファイルをローカルでオープンします。そうでない場合、プロトコルを `ftp` に切り替え、`parent` を使って再度オープンを試みます。

### 11.5.16 FTPHandler オブジェクト

`ftp_open(req)`

`req` で表されるファイルを FTP 越しにオープンします。ログインは常に空のユーザー名およびパスワードで行われます。

### 11.5.17 CacheFTPHandler オブジェクト

CacheFTPHandler オブジェクトは FTPHandler オブジェクトに以下のメソッドを追加したものです:

`setTimeout(t)`

接続のタイムアウトを *t* 秒に設定します。

`setMaxConns(m)`

キャッシュ付き接続の最大接続数を *m* に設定します。

### 11.5.18 GopherHandler オブジェクト

`gopher_open(req)`

`req` で表される gopher 上のリソースをオープンします。

### 11.5.19 UnknownHandler オブジェクト

`unknown_open()`

例外 `URLError` を送出します。

### 11.5.20 例

以下の例では、python.org のメインページを取得して、その最初の 100 バイト分を表示します：

```
>>> import urllib2
>>> f = urllib2.urlopen('http://www.python.org/')
>>> print f.read(100)
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<?xml-stylesheet href="/css/ht2html
```

今度は CGI の標準入力にデータストリームを送信し、CGI が返すデータを読み出します：

```
>>> import urllib2
>>> req = urllib2.Request(url='https://localhost/cgi-bin/test.cgi',
...                        data='This data is passed to stdin of the CGI')
>>> f = urllib2.urlopen(req)
>>> print f.read()
Got Data: "This data is passed to stdin of the CGI"
```

上の例で使われているサンプルの CGI は以下のようにになっています：

```
#!/usr/bin/env python
import sys
data = sys.stdin.read()
print 'Content-type: text-plain\n\nGot Data: "%s"' %
data
```

## 11.6 httplib — HTTP プロトコルクライアント

このモジュールでは HTTP および HTTPS プロトコルのクライアント側を実装しているクラスを定義しています。このモジュールは通常直接使われることはありません — モジュール `urllib` は HTTP や HTTPS を使う URL を扱うためにこのモジュールを使います。注意：HTTPS のサポートは `socket` モジュールが SSL をサポートするようにコンパイルされている場合のみ利用できます。

このモジュールで定義されている定数は以下の通りです：

**HTTP\_PORT**

HTTP プロトコルの標準のポート (通常は 80) です。

**HTTPS\_PORT**

HTTPS プロトコルの標準のポート (通常は 443) です。

このモジュールでは以下のクラスを提供しています：

**class HTTPConnection(*host*[, *port*])**

`HTTPConnection` インスタンスは、HTTP サーバとの一回のトランザクションを表現します。インスタンスの生成はホスト名とオプションのポート番号を与えて行います。ポート番号が与えられなかった場合、ホスト名文字列が `host:port` の形式であれば、ホスト名からポート番号を導き、そうで



ない場合には標準の HTTP ポート番号 (80) が使われます。例えば、以下の呼び出しは全て同じサーバの同じポートに接続するインスタンスを生成します:

```
>>> h1 = httpplib.HTTPConnection('www.cwi.nl')
>>> h2 = httpplib.HTTPConnection('www.cwi.nl:80')
>>> h3 = httpplib.HTTPConnection('www.cwi.nl', 80)
```

**class HTTPSSConnection**(*host*[, *port*, *key\_file*, *cert\_file*])

HTTPConnection のサブクラスで、セキュアなサーバと通信するために SSL を使います。標準のポート番号は 443 です。 *key\_file* には、秘密鍵を格納した PEM 形式ファイルのファイル名を指定します。 *cert\_file* には、PEM 形式の証明書チェーンファイルを指定します。

警告: この関数は証明書の検査を行いません!

必要に応じて以下の例外が送出されます:

**exception HTTPException**

このモジュールにおける他の例外クラスの基底クラスです。Exception のサブクラスです。

**exception NotConnected**

HTTPException サブクラスです。

**exception InvalidURL**

HTTPException のサブクラスで、ポート番号が与えられているが、その値が数字でなかったり空のオブジェクトの場合に送出されます。

**exception UnknownProtocol**

HTTPException のサブクラスです。

**exception UnknownTransferEncoding**

HTTPException のサブクラスです。

**exception IllegalKeywordArgument**

HTTPException のサブクラスです。

**exception UnimplementedFileMode**

HTTPException のサブクラスです。

**exception IncompleteRead**

HTTPException のサブクラスです。

**exception ImproperConnectionState**

HTTPException のサブクラスです。

**exception CannotSendRequest**

ImproperConnectionState のサブクラスです。

**exception CannotSendHeader**

ImproperConnectionState のサブクラスです。

**exception ResponseNotReady**

ImproperConnectionState のサブクラスです。

**exception BadStatusLine**

HTTPException のサブクラスです。サーバが理解できない HTTP 状態コードで応答した場合に送出されます。

### 11.6.1 HTTPConnection オブジェクト

HTTPConnection インスタンスは以下のメソッドを持ちます:

**request**(*method*, *url*[, *body*[, *headers*]])

このメソッドは、HTTP 要求メソッド *method* およびセクタ *url* を使って、要求をサーバに送ります。*body* 引数が存在する場合、ヘッダが終了した後に送信する文字列データでなければなりません。ヘッダの Content-Length は自動的に正しい値に設定されます。*headers* 引数は要求と同時に送信される拡張 HTTP ヘッダの内容からなるマップ型でなくてはなりません。

**getresponse**()

サーバに対して HTTP 要求を送り出した後に呼び出されなければなりません。要求に対する応答を取得します。HTTPResponse インスタンスを返します。

**set\_debuglevel**(*level*)

デバッグレベル (印字されるデバッグ出力の量) を設定します。標準のデバッグレベルは 0 で、デバッグ出力を全く印字しません。

**connect**()

オブジェクトを生成するときに指定したサーバに接続します。

**close**()

サーバへの接続を閉じます。

**send**(*data*)

サーバにデータを送ります。このメソッドは *endheaders*() が呼び出された直後で、かつ *getreply*() が呼び出される前に使わなければなりません。

**putrequest**(*request*, *selector*)

サーバへの接続が確立したら、最初にこのメソッドを呼び出さなくてはなりません。このメソッドは *request* 文字列、*selector* 文字列、そして HTTP バージョン (HTTP/1.1) からなる一行を送信します。

**putheader**(*header*, *argument*[, ... ])

RFC 822 形式のヘッダをサーバに送ります。この処理では、*header*、コロンとスペース、そして最初の引数からなる 1 行をサーバに送ります。追加の引数を与えられている場合、各々の行がタブと引数からなる連続した行が送られます。

**endheaders**()

サーバに食う行を送り、ヘッダ部が終了したことを通知します。

## 11.6.2 HTTPResponse オブジェクト

HTTPResponse インスタンスは以下のメソッドと属性を持ちます:

**read**([*amt*])

応答の本体全体か、*amt* バイトまで読み出して返します。

**getheader**(*name*[, *default*])

ヘッダ *name* の内容を取得して返すか、該当するヘッダがない場合には *default* を返します。

**msg**

応答ヘッダを含む *mimetools.Message* インスタンスです。

**version**

サーバが使用した HTTP プロトコルバージョンです。10 は HTTP/1.0 を、11 は HTTP/1.1 を表します。

**status**

サーバから返される状態コードです。

**reason**

サーバから返される応答の理由文です。

### 11.6.3 例

以下はどうやって ‘GET’ リクエストを行うかを示す例です:

```
>>> import httpplib
>>> conn = httpplib.HTTPConnection("www.python.org")
>>> conn.request("GET", "/index.html")
>>> r1 = conn.getresponse()
>>> print r1.status, r1.reason
200 OK
>>> data1 = r1.read()
>>> conn.request("GET", "/parrot.spam")
>>> r2 = conn.getresponse()
>>> print r2.status, r2.reason
404 Not Found
>>> data2 = r2.read()
>>> conn.close()
```

以下はどうやって ‘POST’ リクエストを行うかを示す例です:

```
>>> import httpplib, urllib
>>> params = urllib.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> headers = {"Content-type": "application/x-www-form-urlencoded",
...           "Accept": "text/plain"}
>>> conn = httpplib.HTTPConnection("musi-cal.mojam.com:80")
>>> conn.request("POST", "/cgi-bin/query", params, headers)
>>> response = conn.getresponse()
>>> print response.status, response.reason
200 OK
>>> data = response.read()
>>> conn.close()
```

## 11.7 ftplib — FTP プロトコルクライアント

このモジュールでは FTP クラスと、それに関連するいくつかの項目を定義しています。FTP クラスは、FTP プロトコルのクライアント側の機能を備えています。このクラスを使うと FTP のいろいろな機能の自動化、例えば他の FTP サーバのミラーリングといったことを実行する Python プログラムを書くことができます。また、urllib モジュールも FTP を使う URL を操作するのにこのクラスを使っています。FTP (File Transfer Protocol) についての詳しい情報は Internet RFC 959 を参照して下さい。

ftplib モジュールを使ったサンプルを以下に示します:

```
>>> from ftplib import FTP
>>> ftp = FTP('ftp.cwi.nl') # ホストのデフォルトポートへ接続
>>> ftp.login() # ユーザ名 anonymous、パスワード anonymous@
s@
>>> ftp.retrlines('LIST') # ディレクトリの内容をリストアップ
total 24418
drwxrwsr-x 5 ftp-usr pdmaint 1536 Mar 20 09:48 .
dr-xr-srwt 105 ftp-usr pdmaint 1536 Mar 21 14:32 ..
-rw-r--r-- 1 ftp-usr pdmaint 5305 Mar 20 09:48 INDEX
.
.
.
>>> ftp.retrbinary('RETR README', open('README', 'wb').write)
'226 Transfer complete.'
>>> ftp.quit()
```

このモジュールは以下の項目を定義しています：

**class FTP**(*[host[, user[, passwd[, acct]]]]*)

FTP クラスの新しいインスタンスを返します。*host* が与えられると、`connect(host)` メソッドが実行されます。*user* が与えられると、さらに `login(user, passwd, acct)` メソッドが実行されます（この *passwd* と *acct* は指定されなければデフォルトでは空文字列です）。

**all\_errors**

FTP インスタンスのメソッドの結果、FTP 接続で（プログラミングのエラーと考えられるメソッドの実行によって）発生する全ての例外（タプル形式）。この例外には以下の4つのエラーはもちろん、`socket.error` と `IOError` も含まれます。

**exception error\_reply**

サーバから想定外の応答があった時に発生する例外。

**exception error\_temp**

400–499 の範囲のエラー応答コードを受け取った時に発生する例外。

**exception error\_perm**

500–599 の範囲のエラー応答コードを受け取った時に発生する例外。

**exception error\_proto**

1–5 の数字で始まらない応答コードをサーバから受け取った時に発生する例外。

参考資料:

netrc モジュール ([12.18 節](#)):

‘netrc’ ファイルフォーマットのパーザ。‘netrc’ ファイルは、FTP クライアントがユーザにプロンプトを出す前に、ユーザ認証情報をロードするのによく使われます。

Python のソースディストリビューションの ‘Tools/scripts/ftplib.py’ ファイルは、FTP サイトあるいはその一部をミラーリングするスクリプトで、ftplib モジュールを使っています。このモジュールを適用した応用例として使うことができます。

### 11.7.1 FTP オブジェクト

いくつかのコマンドは2つのタイプについて実行します：1つはテキストファイルで、もう1つはバイナリファイルを扱います。これらのメソッドのテキストバージョンでは ‘lines’、バイナリバージョンでは ‘binary’ の語がメソッド名の終わりに付いています。

FTP インスタンスには以下のメソッドがあります：

**set\_debuglevel**(*level*)

インスタンスのデバッグレベルを設定します。この設定によってデバッグ時に出力される量を調節します。デフォルトは 0 で、何も出力されません。1 なら、一般的に 1 つのコマンドあたり 1 行の適当な量のデバッグ出力を行います。2 以上なら、コントロール接続で受信した各行を出力して、最大のデバッグ出力をします。

**connect**(*host*[, *port*])

指定されたホストとポートに接続します。ポート番号のデフォルト値は FTP プロトコルの仕様で定められた 21 です。他のポート番号を指定する必要はありません。この関数はひとつのインスタンスに対して一度だけ実行すべきです；インスタンスが作られた時にホスト名が与えられていたら、呼び出すべきではありません。これ以外の他の全てのメソッドは接続された後で実行可能となります。

**getwelcome**()

接続して最初にサーバから送られてくるウェルカムメッセージを返します。(このメッセージには、ユーザにとって適切な注意書きやヘルプ情報が含まれることがあります。)

**login**([*user*[, *passwd*[, *acct*]]])

与えられた *user* でログインします。*passwd* と *acct* のパラメータは省略可能で、デフォルトでは空文字列です。もし *user* が指定されないなら、デフォルトで 'anonymous' になります。もし *user* が 'anonymous' なら、デフォルトの *passwd* は 'anonymous@' になります。この function は各インスタンスについて一度だけ、接続が確立した後に呼び出さなければなりません；インスタンスが作られた時にホスト名とユーザ名が与えられていたら、このメソッドを実行すべきではありません。ほとんどの FTP コマンドはクライアントがログインした後に実行可能になります。

**abort**()

実行中のファイル転送を中止します。これはいつも機能するわけではありませんが、やってみる価値はあります。

**sendcmd**(*command*)

シンプルなコマンド文字列をサーバに送信して、受信した文字列を返します。

**voidcmd**(*command*)

シンプルなコマンド文字列をサーバに送信して、その応答を扱います。応答コードが 200–299 の範囲にあれば何も返しません。それ以外は例外を発生します。

**retrbinary**(*command*, *callback*[, *maxblocksize*[, *rest*]])

バイナリ転送モードでファイルを受信します。*command* は適切な 'RETR' コマンド: 'RETR *filename*' でなければなりません。関数 *callback* は、受信したデータブロックのそれぞれに対して、データブロックを 1 つの文字列の引数として呼び出されます。省略可能な引数 *maxblocksize* は、実際の転送を行うのに作られた低レベルのソケットオブジェクトから読み込む最大のチャンクサイズを指定します (これは *callback* に与えられるデータブロックの最大サイズにもなります)。妥当なデフォルト値が設定されます。*rest* は、**transfercmd**() メソッドと同じものです。

**retrlines**(*command*[, *callback*])

ASCII 転送モードでファイルとディレクトリの一覧を受信します。*command* は、適切な 'RETR' コマンド (**retrbinary**() を参照) あるいは 'LIST' コマンド (通常は文字列 'LIST') でなければなりません。関数 *callback* は末尾の CRLF を取り除いた各行に対して実行されます。デフォルトでは *callback* は `sys.stdout` に各行を印字します。

**set\_pasv**(*boolean*)

*boolean* が true なら “パッシブモード” をオンにし、そうでないならパッシブモードをオフにします。(Python 2.0 以前ではデフォルトでパッシブモードはオフにされていましたが、Python 2.1 以後ではデフォルトでオンになっています。)

**storbinary**(*command*, *file*[, *blocksize*])

バイナリ転送モードでファイルを転送します。*command* は適切な ‘STOR’ コマンド: "STOR *filename*" でなければなりません。*file* は開かれたファイルオブジェクトで、`read()` メソッドで EOF まで読み込まれ、ブロックサイズ *blocksize* でデータが転送されます。引数 *blocksize* のデフォルト値は 8192 です。2.1 で変更された仕様: *blocksize* のデフォルト値が追加されました

**storlines**(*command*, *file*)

ASCII 転送モードでファイルを転送します。*command* は適切な ‘STOR’ コマンドでなければなりません (`st_orbinary()` を参照)。 *file* は開かれたファイルオブジェクトで、`readline()` メソッドで EOF まで読み込まれ、各行がデータが転送されます。

**transfercmd**(*cmd*[, *rest*])

データ接続中に転送を初期化します。もし転送中なら、‘EPRT’ あるいは ‘PORT’ コマンドと、*cmd* で指定したコマンドを送信し、接続を続けます。サーバがパッシブなら、‘EPSV’ あるいは ‘PASV’ コマンドを送信して接続し、転送コマンドを開始します。どちらの場合も、接続のためのソケットを返します。

省略可能な *rest* が与えられたら、‘REST’ コマンドがサーバに送信され、*rest* を引数として与えます。*rest* は普通、要求したファイルのバイトオフセット値で、最初のバイトをとばして指定したオフセット値からファイルのバイト転送を再開するよう伝えます。しかし、RFC 959 では *rest* が印字可能な ASCII コード 33 から 126 の範囲の文字列からなることを要求していることに注意して下さい。したがって、`transfercmd()` メソッドは *rest* を文字列に変換しますが、文字列の内容についてチェックしません。もし ‘REST’ コマンドをサーバが認識しないなら、例外 `error_reply` が発生します。この例外が発生したら、引数 *rest* なしに `transfercmd()` を実行します。

**ntransfercmd**(*cmd*[, *rest*])

`transfercmd()` と同様ですが、データと予想されるサイズとのタプルを返します。もしサイズが計算できないなら、サイズの代わりに `None` が返されます。*cmd* と *rest* は `transfercmd()` のものと同じです。

**nlst**(*argument*[, ...])

‘NLST’ コマンドで返されるファイルのリストを返します。省略可能な *argument* は、リストアップするディレクトリです (デフォルトではサーバのカレントディレクトリです)。*‘NLST’* コマンドに非標準である複数の引数を渡すことができます。

**dir**(*argument*[, ...])

‘LIST’ コマンドで返されるディレクトリ内のリストを作り、標準出力へ出力します。省略可能な *argument* は、リストアップするディレクトリです (デフォルトではサーバのカレントディレクトリです)。*‘LIST’* コマンドに非標準である複数の引数を渡すことができます。もし最後の引数が関数なら、`retrlines()` のように *callback* として使われます; デフォルトでは `sys.stdout` に印字します。このメソッドは `None` を返します。

**rename**(*fromname*, *toname*)

サーバ上のファイルのファイル名 *fromname* を *toname* へ変更します。

**delete**(*filename*)

サーバからファイル *filename* を削除します。成功したら応答のテキストを返し、そうでないならパーミッションエラーでは `error_perm` を、他のエラーでは `error_reply` を返します。

**cwd**(*pathname*)

サーバのカレントディレクトリを設定します。

**mkd**(*pathname*)

サーバ上に新たにディレクトリを作ります。

**pwd**()

サーバ上のカレントディレクトリのパスを返します。



`rmrmdir(dirname)`

サーバ上のディレクトリ *dirname* を削除します。

`size(filename)`

サーバ上のファイル *filename* のサイズを尋ねます。成功したらファイルサイズが整数で返され、そうでないなら `None` が返されます。‘SIZE’ コマンドは標準化されていませんが、多くの普通のサーバで実装されていることに注意して下さい。

`quit()`

サーバに ‘QUIT’ コマンドを送信し、接続を閉じます。これは接続を閉じるのに“礼儀正しい”方法ですが、‘QUIT’ コマンドに反応してサーバの例外が発生するかもしれません。この例外は、`close()` メソッドによって FTP インスタンスに対するその後のコマンド使用が不可になっていることを示しています（下記参照）。

`close()`

接続を一方向的に閉じます。既に閉じた接続に対して実行すべきではありません（例えば `quit()` を呼び出して成功した後など）。この実行の後、FTP インスタンスはもう使用すべきではありません（`close()` あるいは `quit()` を呼び出した後で、`login()` メソッドをもう一度実行して再び接続を開くことはできません）。

## 11.8 gopherlib — gopher プロトコルのクライアント

このモジュールでは、gopher プロトコルのクライアント側について最小限の実装を提供しています。このモジュールは `urllib` モジュールで gopher プロトコルを使う URL を扱うために用いられます。

このモジュールでは以下の関数を定義しています：

`send_selector(selector, host[, port])`

*selector* 文字列を *host* および *port* (標準の値は 70 です) の gopher サーバに送信します。返信されたドキュメントデータを読み出すための、開かれた状態のファイルオブジェクトを返します。

`send_query(selector, query, host[, port])`

*selector* 文字列、および *query* 文字列を *host* および *port* (標準の値は 70 です) の gopher サーバに送信します。返信されたドキュメントデータを読み出すための、開かれた状態のファイルオブジェクトを返します。

gopher サーバから返されるデータは任意の形式であり、セレクタ (*selector*) 文字列の最初の文字に依存するので注意してください。データがテキスト (セレクタの最初の文字が ‘0’) の場合、各行は CRLF で終端され、データ全体は ‘.’ 一個だけからなる行で終端されます。‘.’ で始まる行の先頭は ‘.’ に置き換えられます。ディレクトリリスト (セレクタの最初の文字が ‘1’) の場合にも、同じプロトコルで転送されます。

## 11.9 poplib — POP3 プロトコルクライアント

このモジュールは、POP3 クラスを定義します。これは POP3 サーバへの接続と、RFC 1725 に定められたプロトコルを実装します。POP3 クラスは `minimal` と `optimal` という 2 つのコマンドセットをサポートします。

POP3 についての注意事項は、それが広くサポートされているにもかかわらず、既に時代遅れだということです。幾つも実装されている POP3 サーバーの品質は、貧弱なものが多数を占めています。もし、お使いのメールサーバーが IMAP をサポートしているなら、`imaplib` や IMAP4 が使えます。IMAP サーバーは、より良く実装されている傾向があります。

`poplib` モジュールでは、ひとつのクラスが提供されています。

`class POP3(host[, port])`

このクラスが、実際に POP3 プロトコルを実装します。インスタンスが初期化されるときに、コネクションが作成されます。`port` が省略されると、POP3 標準のポート (110) が使われます。

1 つの例外が、`poplib` モジュールのアトリビュートとして定義されています。

`exception error_proto`

例外は、すべてのエラーで発生します。例外の理由は文字列としてコンストラクタに渡されます。

参考資料:

`imaplib` モジュール (11.10 節):

The standard Python IMAP module.

*Frequently Asked Questions About Fetchmail*

(<http://www.tuxedo.org/~{esr}/fetchmail/fetchmail-FAQ.html>)

POP/IMAP クライアント **fetchmail** の FAQ。POP プロトコルをベースにしたアプリケーションを書くときに有用な、POP3 サーバの種類や RFC への適合度といった情報を収集しています。

### 11.9.1 POP3 オブジェクト

POP3 コマンドはすべて、それと同じ名前のメソッドとして lower-case で表現されます。そしてそのほとんどは、サーバからのレスポンスとなるテキストを返します。

POP3 クラスのインスタンスは以下のメソッドを持ちます。

`set_debuglevel(level)`

インスタンスのデバッグレベルを指定します。これはデバッグアウトプットの表示量をコントロールします。デフォルト値の 0 は、デバッグアウトプットを表示しません。値を 1 とすると、デバッグアウトプットの表示量を適当な量にします。これは大体、リクエストごと 1 行になります。値を 2 以上にすると、デバッグアウトプットの表示量を最大にします。コントロール中の接続で送受信される各行をログに出力します。

`getwelcome()`

POP3 サーバーから送られるグリーティングメッセージを返します。

`user(username)`

`user` コマンドを送出します。応答はパスワード要求を表示します。

`pass_(password)`

パスワードを送出します。応答は、メッセージ数とメールボックスのサイズを含みます。注: サーバー上のメールボックスは `quit()` が呼ばれるまでロックされます。

`apop(user, secret)`

POP3 サーバーにログオンするのに、よりセキュアな APOP 認証を使用します。

`rpop(user)`

POP3 サーバーにログオンするのに、(UNIX の `r`-コマンドと同様の) RPOP 認証を使用します。

`stat()`

メールボックスの状態を得ます。結果は 2 つの integer からなるタプルとなります。(message count, mailbox size).

`list([which])`

メッセージのリストを要求します。結果は以下のような形式で表されます。(response, ['mesg\_num octets', ...]) `which` が与えられると、それによりメッセージを指定します。

`retr(which)`

`which` 番のメッセージ全体を取り出し、そのメッセージに既読フラグを立てます。結果は (response,

[*'line'*, ...], *octets*) という形式で表されます。

**delete**(*which*)

*which* 番のメッセージに削除のためのフラグを立てます。ほとんどのサーバで、QUIT コマンドが実行されるまでは実際の削除は行われません (もっとも良く知られた例外は Eudora QPOP で、その配送メカニズムは RFC に違反しており、どんな切断状況でも削除操作を未解決にしています)。

**rset**()

メールボックスの削除マークすべてを取り消します。

**noop**()

何もしません。接続保持のために使われます。

**quit**()

Signoff: commit changes, unlock mailbox, drop connection. サインオフ: 変更をコミットし、メールボックスをアンロックして、接続を破棄します。

**top**(*which*, *howmuch*)

メッセージヘッダと *howmuch* で指定した行数のメッセージを、*which* で指定したメッセージ分取り出します。結果は以下のような形式となります。(response, [*'line'*, ...], *octets*).

このメソッドは POP3 の TOP コマンドを利用し、RETR コマンドのように、メッセージに既読フラグをセットしません。残念ながら、TOP コマンドは RFC では貧弱な仕様しか定義されておらず、しばしばノーブランドのサーバでは (その仕様が) 守られていません。このメソッドを信用してしまう前に、実際に使用する POP サーバでテストをしてください。

**uidl**([*which*])

(ユニーク ID による) メッセージダイジェストのリストを返します。*which* が設定されている場合、結果はユニーク ID を含みます。それは *'response mesgnum uid'* という形式のメッセージ、または (response, [*'mesgnum uid'*, ...], *octets*) という形式のリストとなります。

## 11.9.2 POP3 の例

これは (エラーチェックもない) 最も小さなサンプルで、メールボックスを開いて、すべてのメッセージを取り出し、プリントします。

```
import getpass, poplib

M = poplib.POP3('localhost')
M.user(getpass.getuser())
M.pass_(getpass.getpass())
numMessages = len(M.list()[1])
for i in range(numMessages):
    for j in M.retr(i+1)[1]:
        print j
```

モジュールの末尾に、より広い範囲の使用例となる test セクションがあります。

## 11.10 imaplib — IMAP4 プロトコルクライアント

このモジュールでは三つのクラス、IMAP4, IMAP4\_SSL と IMAP4\_stream を定義します。これらのクラスは IMAP4 サーバへの接続をカプセル化し、RFC 2060 に定義されている IMAP4rev1 クライアントプロトコルの大規模なサブセットを実装しています。このクラスは IMAP4 (RFC 1730) 準拠のサーバと後方互換性がありますが、*'STATUS'* コマンドは IMAP4 ではサポートされていないので注意してください。

imaplib モジュール内では三つのクラスを提供しており、IMAP4 は基底クラスとなります:

**class IMAP4**(*[host[, port]]*)

このクラスは実際の IMAP4 プロトコルを実装しています。インスタンスが初期化された際に接続が生成され、プロトコルバージョン (IMAP4 または IMAP4rev1) が決定されます。*host* が指定されていない場合、" (ローカルホスト) が用いられます。*port* が省略された場合、標準の IMAP4 ポート番号 (143) が用いられます。

例外は IMAP4 クラスの属性として定義されています:

**exception IMAP4.error**

何らかのエラー発生の際に送出される例外です。例外の理由は文字列としてコンストラクタに渡されます。

**exception IMAP4.abort**

IMAP4 サーバのエラーが生じると、この例外が送出されます。この例外は IMAP4.error のサブクラスです。通常、インスタンスを閉じ、新たなインスタンスを再び生成することで、この例外から復旧できます。

**exception IMAP4.readonly**

この例外は書き込み可能なメールボックスの状態がサーバによって変更された際に送出されます。この例外は IMAP4.error のサブクラスです。他の何らかのクライアントが現在書き込み権限を獲得しており、メールボックスを開きなおして書き込み権限を再獲得する必要があります。

このモジュールではもう一つ、安全 (secure) な接続を使ったサブクラスがあります:

**class IMAP4\_SSL**(*[host[, port[, keyfile[, certfile]]]]*)

IMAP4 から導出されたサブクラスで、SSL 暗号化ソケットを介して接続を行います (このクラスを利用するためには SSL サポート付きでコンパイルされた socket モジュールが必要です)。*host* が指定されていない場合、" (ローカルホスト) が用いられます。*port* が省略された場合、標準の IMAP4-over-SSL ポート番号 (993) が用いられます。*keyfile* および *certfile* もオプションです - これらは SSL 接続のための PEM 形式の秘密鍵 (private key) と認証チェーン (certificate chain) ファイルです。

さらにもう一つのサブクラスは、子プロセスで確立した接続を使用する場合に使用します。

**class IMAP4\_stream**(*command*)

IMAP4 から導出されたサブクラスで、*command* を `os.popen2()` に渡して作成される stdin/stdout ディスクリプタと接続します。2.3 で追加された仕様です。

以下のユーティリティ関数が定義されています:

**Internaldate2tuple**(*datestr*)

IMAP4 INTERNALDATE 文字列を標準世界時 (Coordinated Universal Time) に変換します。time モジュール形式のタプルを返します。

**Int2AP**(*num*)

整数を [A .. P] からなる文字集合を用いて表現した文字列に変換します。

**ParseFlags**(*flagstr*)

IMAP4 'FLAGS' 応答を個々のフラグからなるタプルに変換します。

**Time2Internaldate**(*date\_time*)

time モジュールタプルを IMAP4 'INTERNALDATE' 表現形式に変換します。文字列形式: "DD-Mmm-YYYY HH:MM:SS +HHMM" (二重引用符含む) を返します。

IMAP4 メッセージ番号は、メールボックスに対する変更が行われた後には変化します; 特に、'EXPUNGE' 命令はメッセージの削除を行いますが、残ったメッセージには再度番号を振りなおします。従って、メッセージ番号ではなく、UID 命令を使い、その UID を利用するよう強く勧めます。

モジュールの末尾に、より拡張的な使用例が収められたテストセクションがあります。

参考資料:

プロトコルに関する記述、およびプロトコルを実装したサーバのソースとバイナリは、全てワシントン大学の IMAP Information Center (<http://www.cac.washington.edu/imap/>) にあります。

### 11.10.1 IMAP4 オブジェクト

全ての IMAP4rev1 命令は、同じ名前のメソッドで表されており、大文字のものも小文字のものもあります。

命令に対する引数は全て文字列に変換されます。例外は 'AUTHENTICATE' の引数と 'APPEND' の最後の引数で、これは IMAP4 リテラルとして渡されます。必要に応じて (IMAP4 プロトコルが感知対象としている文字が文字列に入っており、かつ丸括弧か二重引用符で囲われていなかった場合) 文字列はクオートされます。しかし、'LOGIN' 命令の *password* 引数は常にクオートされます。文字列がクオートされないようにしたい (例えば 'STORE' 命令の *flags* 引数) 場合、文字列を丸括弧で囲ってください (例: `r'(\Deleted)'`)。

各命令はタプル: (*type*, [*data*, ...]) を返し、*type* は通常 'OK' または 'NO' です。*data* は命令に対する応答をテキストにしたものか、命令に対する実行結果です。各 *data* は文字列かタプルとなります。タプルの場合、最初の要素はレスポンスのヘッダで、次の要素にはデータが格納されます。(ie: 'literal' value)

IMAP4 のインスタンスは以下のメソッドを持っています:

**append**(*mailbox*, *flags*, *date\_time*, *message*)

指定された名前のメールボックスにメッセージを追加します。

**authenticate**(*func*)

認証命令です — 応答の処理が必要です。現在は実装されておらず、例外を送出します。

**check**()

サーバ上のメールボックスにチェックポイントを設定します。Checkpoint mailbox on server.

**close**()

現在選択されているメールボックスを閉じます。削除されたメッセージは書き込み可能メールボックスから除去されます。'LOGOUT' 前に実行することを勧めます。

**copy**(*message\_set*, *new\_mailbox*)

*message\_set* で指定したメッセージ群を *new\_mailbox* の末尾にコピーします。

**create**(*mailbox*)

*mailbox* と名づけられた新たなメールボックスを生成します。

**delete**(*mailbox*)

*mailbox* と名づけられた古いメールボックスを削除します。

**expunge**()

選択されたメールボックスから削除された要素を永久に除去します。各々の削除されたメッセージに対して、'EXPUNGE' 応答を生成します。返されるデータには 'EXPUNGE' メッセージ番号を受信した順番に並べたリストが入っています。

**fetch**(*message\_set*, *message\_parts*)

メッセージ (の一部) を取りよめます。*message\_parts* はメッセージパートの名前を表す文字列を丸括弧で囲ったもので、例えば: `"(UID BODY[TEXT])"` のようになります。返されるデータはメッセージパートのエンベロープ情報とデータからなるタプルです。

**getacl**(*mailbox*)

*mailbox* に対する 'ACL' を取得します。このメソッドは非標準ですが、'Cyrus' サーバでサポートされています。

**getquota**(*root*)

'quota' *root* により、リソース使用状況と制限値を取得します。このメソッドは RFC 2087 で定義さ



れている IMAP4 QUOTA 拡張の一部です。2.3 で追加された仕様です。

**getquotaroot** (*mailbox*)

*mailbox* に対して 'quota' *root* を実行した結果のリストを取得します。このメソッドは RFC 2087 で定義されている IMAP4 QUOTA 拡張の一部です。2.3 で追加された仕様です。

**list** ([*directory* [, *pattern*]])

*pattern* にマッチする *directory* メールボックス名を列挙します。*directory* の標準の設定値は最上レベルのメールフォルダで、*pattern* は標準の設定では全てにマッチします。返されるデータには 'LIST' 応答のリストが入っています。

**login** (*user*, *password*)

平文パスワードを使ってクライアントを照合します。*password* はクオートされます。

**login\_cram\_md5** (*user*, *password*)

パスワードの保護のため、クライアント認証時に 'CRAM-MD5' だけを使用します。これは、'CAPABILITY' レスポンスに 'AUTH=CRAM-MD5' が含まれる場合のみ有効です。2.3 で追加された仕様です。

+

**logout** ()

サーバへの接続を遮断します。サーバからの 'BYE' 応答を返します。

**lsub** ([*directory* [, *pattern*]])

購読しているメールボックス名のうち、ディレクトリ内でパターンにマッチするものを列挙します。*directory* の標準の設定値は最上レベルのメールフォルダで、*pattern* は標準の設定では全てにマッチします。返されるデータには返されるデータはメッセージパートエンベロープ情報とデータからなるタプルです。

**noop** ()

サーバに 'NOOP' を送信します。

**open** (*host*, *port*)

*host* 上の *port* に対するソケットを開きます。このメソッドで確立された接続オブジェクトは *read*、*readline*、*send*、および *shutdown* メソッドで使われます。このメソッドはオーバーライドすることができます。

**partial** (*message\_num*, *message\_part*, *start*, *length*)

メッセージの後略された部分を取り寄せます。返されるデータはメッセージパートエンベロープ情報とデータからなるタプルです。

**proxyauth** (*user*)

*user* として認証されたものとします。認証された管理者がユーザの代理としてメールボックスにアクセスする際に使用します。2.3 で追加された仕様です。

+

**read** (*size*)

遠隔のサーバから *size* バイト読み出します。このメソッドはオーバーライドすることができます。

**readline** ()

遠隔のサーバから一行読み出します。このメソッドはオーバーライドすることができます。

**recent** ()

サーバに更新を促します。新たなメッセージがない場合応答は *None* になり、そうでない場合 'RECENT' 応答の値になります。

**rename** (*oldmailbox*, *newmailbox*)

*oldmailbox* という名前のメールボックスを *newmailbox* に名称変更します。



**response**(*code*)

応答 *code* を受信していれば、そのデータを返し、そうでなければ `None` を返します。通常の形式 (usual type) ではなく指定したコードを返します。

**search**(*charset*, *criterion*[, ...])

条件に合致するメッセージをメールボックスから検索します。返されるデータには合致したメッセージ番号をスペースで分割したリストが入っています。*charset* は `None` でもよく、この場合にはサーバへの要求内に 'CHARSET' は指定されません。IMAP プロトコルは少なくとも一つの条件 (*criterion*) が指定されるよう要求しています; サーバがエラーを返した場合、例外が送出されます。

例:

```
# M is a connected IMAP4 instance...
msgnums = M.search(None, 'FROM', '"LDJ"')

# or:
msgnums = M.search(None, '(FROM "LDJ")')
```

**select**([*mailbox*[, *readonly*]])

メールボックスを選択します。返されるデータは *mailbox* 内のメッセージ数 ('EXISTS' 応答) です。標準の設定では *mailbox* は 'INBOX' です。*readonly* が設定された場合、メールボックスに対する変更はできません。

**send**(*data*)

遠隔のサーバに *data* を送信します。このメソッドはオーバーライドすることができます。

**setacl**(*mailbox*, *who*, *what*)

'ACL' を *mailbox* に設定します。このメソッドは非標準ですが、'Cyrus' サーバでサポートされています。

**setquota**(*root*, *limits*)

'quota' *root* のリソースを *limits* に設定します。このメソッドは RFC 2087 で定義されている IMAP4 QUOTA 拡張の一部です。2.3 で追加された仕様です。

**shutdown**()

`open` で確立された接続を閉じます。このメソッドはオーバーライドすることができます。

**socket**()

サーバへの接続に使われているソケットインスタンスを返します。

**sort**(*sort\_criteria*, *charset*, *search\_criterion*[, ...])

`sort` 命令は `search` に結果の並べ替え (sort) 機能をつけた変種です。返されるデータには、条件に合致するメッセージ番号をスペースで分割したリストが入っています。`sort` 命令は *search\_criterion* の前に二つの引数を持ちます; *sort\_criteria* のリストを丸括弧で囲ったものと、検索時の *charset* です。`search` と違って、検索時の *charset* は必須です。`uid sort` 命令もあり、`search` に対する `uid search` と同じように `sort` 命令に対応します。`sort` 命令はまず、*charset* 引数の指定に従って searching criteria の文字列を解釈し、メールボックスから与えられた検索条件に合致するメッセージを探します。次に、合致したメッセージの数を返します。

'IMAP4rev1' 拡張命令です。

**status**(*mailbox*, *names*)

*mailbox* の指定ステータス名の状態情報を要求します。

**store**(*message\_set*, *command*, *flag\_list*)

メールボックス内のメッセージ群のフラグ設定を変更します。

**subscribe**(*mailbox*)

新たなメールボックスを購読 (subscribe) します。

`uid(command, arg[, ...])`

`command` `args` を、メッセージ番号ではなく UID で指定されたメッセージ群に対して実行します。命令内容に応じた応答を返します。少なくとも一つの引数を与えなくてはなりません; 何も与えない場合、サーバはエラーを返し、例外が送出されます。

`unsubscribe(mailbox)`

古いメールボックスの購読を解除 (unsubscribe) します。

`xatom(name[, arg[, ...]])`

サーバから 'CAPABILITY' 応答で通知された単純な拡張命令を許容 (allow) します。

IMAP4\_SSL のインスタンスは追加のメソッドを一つだけ持ちます:

`ssl()`

サーバへの安全な接続に使われる SSLObject インスタンスを返します。

以下の属性が IMAP4 のインスタンス上で定義されています:

`PROTOCOL_VERSION`

サーバから返された 'CAPABILITY' 応答にある、サポートされている最新のプロトコルです。

`debug`

デバッグ出力を制御するための整数値です。初期値はモジュール変数 `Debug` から取られます。3 以上の値にすると各命令をトレースします。

## 11.10.2 IMAP4 の使用例

以下にメールボックスを開き、全てのメッセージを取得して印刷する最小の (エラーチェックをしない) 使用例を示します:

```
import getpass, imaplib

M = imaplib.IMAP4()
M.login(getpass.getuser(), getpass.getpass())
M.select()
typ, data = M.search(None, 'ALL')
for num in data[0].split():
    typ, data = M.fetch(num, '(RFC822)')
    print 'Message %s\n%s\n' % (num, data[0][1])
M.logout()
```

## 11.11 nntplib — NNTP プロトコルクライアント

このモジュールでは、クラス `NNTP` を定義しています。このクラスは NNTP プロトコルのクライアント側を実装しています。このモジュールを使えば、ニュースリーダや記事投稿プログラム、または自動的にニュース記事进行处理するプログラムを実装することができます。NNTP (Network News Transfer Protocol、ネットニュース転送プロトコル) の詳細については、インターネット RFC 977 を参照してください。

以下にこのモジュールの使い方の小さな例を二つ示します。ニュースグループに関する統計情報を列挙し、最新 10 件の記事を出力するには以下のようにします:

```

>>> s = NNTP('news.cwi.nl')
>>> resp, count, first, last, name = s.group('comp.lang.python')
>>> print 'Group', name, 'has', count, 'articles, range', first, 'to', last
Group comp.lang.python has 59 articles, range 3742 to 3803
>>> resp, subs = s.xhdr('subject', first + '-' + last)
>>> for id, sub in subs[-10:]: print id, sub
...
3792 Re: Removing elements from a list while iterating...
3793 Re: Who likes Info files?
3794 Emacs and doc strings
3795 a few questions about the Mac implementation
3796 Re: executable python scripts
3797 Re: executable python scripts
3798 Re: a few questions about the Mac implementation
3799 Re: PROPOSAL: A Generic Python Object Interface for Python C Modules
3802 Re: executable python scripts
3803 Re: \POSIX{} wait and SIGCHLD
>>> s.quit()
'205 news.cwi.nl closing connection.  Goodbye.'

```

ファイルから記事を投稿するには、以下のようにします (この例では記事番号は有効な番号を指定していると仮定しています):

```

>>> s = NNTP('news.cwi.nl')
>>> f = open('/tmp/article')
>>> s.post(f)
'240 Article posted successfully.'
>>> s.quit()
'205 news.cwi.nl closing connection.  Goodbye.'

```

このモジュール自体では以下の内容を定義しています:

**class NNTP**(*host* [, *port* [, *user* [, *password* [, *readermode* ]]]])

ホスト *host* 上で動作し、ポート番号 *port* で要求待ちをしている NNTP サーバとの接続を表現する新たな NNTP クラスのインスタンスを返します。標準の *port* は 119 です。オプションの *user* および *password* が与えられているか、または */.netrc* に適切な認証情報が指定されている場合、*'AUTHINFO USER'* および *'AUTHINFO PASS'* 命令を使ってサーバに対して身元証明および認証を行います。オプションのフラグ *readermode* が真の場合、認証の実行に先立って *'mode reader'* 命令が送信されます。reader モードは、ローカルマシン上の NNTP サーバに接続していて、*'group'* のような reader 特有の命令を呼び出したい場合に便利ことがあります。予期せず *NNTPPermanentError* に遭遇したなら、*readermode* を設定する必要があるかもしれません。readermode 標準で None です。

**class NNTPError**( )

標準の例外 *Exception* から導出されており、*nntplib* モジュールが送出する全ての例外の基底クラスです。

**class NNTPReplyError**( )

期待はずれの応答がサーバから返された場合に送出される例外です。以前のバージョンとの互換性のために、*error\_reply* はこのクラスと等価になっています。

**class NNTPTemporaryError**( )

エラーコードの範囲が 400-499 のエラーを受信した場合に送出される例外です。以前のバージョンとの互換性のために、*error\_temp* はこのクラスと等価になっています。

**class NNTPPermanentError**( )

エラーコードの範囲が 500-599 のエラーを受信した場合に送出される例外です。以前のバージョンと

の互換性のために、`error_perm` はこのクラスと等価になっています。

`class NNTPProtocolError()`

サーバから返される応答が 1-5 の範囲の数字で始まっていない場合に送出される例外です。以前のバージョンとの互換性のために、`error_proto` はこのクラスと等価になっています。

`class NNTPDataError()`

応答データ中に何らかのエラーが存在する場合に送出される例外です。以前のバージョンとの互換性のために、`error_data` はこのクラスと等価になっています。

### 11.11.1 NNTP オブジェクト

NNTP インスタンスは以下のメソッドを持っています。全てのメソッドにおける戻り値のタプルで最初の要素となる *response* は、サーバの応答です: この文字列は 3 桁の数字からなるコードで始まります。サーバの応答がエラーを示す場合、上記のいずれかの例外が送出されます。

`getwelcome()`

サーバに最初に接続した際に送信される応答中のウェルカムメッセージを返します。(このメッセージには時に、ユーザにとって重要な免責事項やヘルプ情報が入っています。)

`set_debuglevel(level)`

インスタンスのデバッグレベルを設定します。このメソッドは印字されるデバッグ出力の量を制御します。標準では 0 に設定されていて、これはデバッグ出力を全く印字しません。1 はそこそこの量、一般に NNTP 要求や応答あたり 1 行のデバッグ出力を生成します。値が 2 やそれ以上の場合、(メッセージテキストを含めて) NNTP 接続上で送受信された全ての内容を一行ごとにログ出力する、最大限のデバッグ出力を生成します。

`newgroups(date, time, [file])`

‘NEWSGROUPS’ 命令を送信します。*date* 引数は ‘*yymmdd*’ の形式を取り、日付を表します。*time* 引数は ‘*hhmmss*’ の形式をとり、時刻を表します。与えられた日付および時刻以後新たに出現したニュースグループ名のリストを *groups* として、(*response*, *groups*) を返します。*file* 引数が指定されている場合、‘NEWSGROUPS’ コマンドの出力結果はファイルに格納されます。*file* が文字列の場合、この文字列をファイル名としてファイルをオープンし、書き込み後にクローズします。*file* がファイルオブジェクトの場合、オブジェクトの `write()` メソッドを呼び出して出力結果を格納します。*file* が指定されている場合は戻り値として空のリストを返します。

`newnews(group, date, time, [file])`

‘NEWNEWS’ 命令を送信します。ここで、*group* はグループ名または ‘\*’ で、*date* および *time* は `newsgroups()` における引数と同じ意味を持ちます。( *response*, *articles* ) からなるペアを返し、*articles* は記事 ID のリストです。*file* 引数が指定されている場合、‘NEWNEWS’ コマンドの出力結果はファイルに格納されます。*file* が文字列の場合、この文字列をファイル名としてファイルをオープンし、書き込み後にクローズします。*file* がファイルオブジェクトの場合、オブジェクトの `write()` メソッドを呼び出して出力結果を格納します。*file* が指定されている場合は戻り値として空のリストを返します。

`list([file])`

‘LIST’ 命令を送信します。( *response*, *list* ) からなるペアを返します。*list* はタプルからなるリストです。各タプルは (*group*, *last*, *first*, *flag*) の形式をとり、*group* がグループ名、*last* および *first* はそれぞれ最新および最初の記事の記事番号 (を表す文字列)、そして *flag* は投稿が可能な場合には ‘y’、そうでない場合には ‘n’、グループがモデレート (moderated) されている場合には ‘m’ となります。(順番に注意してください: *last*, *first* の順です。) *file* 引数が指定されている場合、‘LIST’ コマンドの出力結果はファイルに格納されます。*file* が文字列の場合、この文字列をファイル名としてファ

イルをオープンし、書き込み後にクローズします。*file* がファイルオブジェクトの場合、オブジェクトの `write()` メソッドを呼び出して出力結果を格納します。*file* が指定されている場合は戻り値として空のリストを返します。

**group**(*name*)

‘GROUP’ 命令を送信します。*name* はグループ名です。タプル (*response*, *count*, *first*, *last*, *name*) を返します。*count* はグループ中の記事数 (の推定値) で、*first* はグループ中の最初の記事番号、*last* はグループ中の最新の記事番号、*name* はグループ名です。記事番号は文字列で返されます。

**help**( [*file*] )

‘HELP’ 命令を送信します。( *response*, *list* ) からなるペアを返します。*list* はヘルプ文字列からなるリストです。*file* 引数が指定されている場合、‘HELP’ コマンドの出力結果はファイルに格納されます。*file* が文字列の場合、この文字列をファイル名としてファイルをオープンし、書き込み後にクローズします。*file* がファイルオブジェクトの場合、オブジェクトの `write()` メソッドを呼び出して出力結果を格納します。*file* が指定されている場合は戻り値として空のリストを返します。

**stat**(*id*)

‘STAT’ 命令を送信します。*id* は (‘<’ と ‘>’ に囲まれた形式の) メッセージ ID か、(文字列の) 記事番号です。三つ組み (*response*, *number*, *id*) を返します。*number* は (文字列の) 記事番号で、*id* は (‘<’ と ‘>’ に囲まれた形式の) メッセージ ID です。

**next**()

‘NEXT’ 命令を送信します。`stat()` のような応答を返します。

**last**()

‘LAST’ 命令を送信します。`stat()` のような応答を返します。

**head**(*id*)

‘HEAD’ 命令を送信します。*id* は `stat()` におけるのと同じ意味を持ちます。( *response*, *number*, *id*, *list* ) からなるタプルを返します。最初の 3 要素は `stat()` と同じもので、*list* は記事のヘッダからなるリスト (まだ解析されておらず、末尾の改行が取り去られたヘッダ行のリスト) です。

**body**(*id*, [*file*] )

‘BODY’ 命令を送信します。*id* は `stat()` におけるのと同じ意味を持ちます。*file* 引数が与えられている場合、記事本体 (body) はファイルに保存されます。*file* が文字列の場合、このメソッドはその名前を持つファイルオブジェクトを開き、記事を書き込んで閉じます。*file* がファイルオブジェクトの場合、`write()` を呼び出して記事本体を記録します。`head()` のような戻り値を返します。*file* が与えられていた場合、返される *list* は空のリストになります。

**article**(*id*)

‘ARTICLE’ 命令を送信します。*id* は `stat()` におけるのと同じ意味を持ちます。`head()` のような戻り値を返します。

**slave**()

‘SLAVE’ 命令を送信します。サーバの *response* を返します。

**xhdr**(*header*, *string*, [*file*] )

‘XHDR’ 命令を送信します、この命令は RFC には定義されていませんが、一般に広まっている拡張です。*header* 引数は、例えば ‘subject’ といったヘッダキーワードです。*string* 引数は ‘first-last’ の形式でなければならず、ここで *first* および *last* は検索の対象とする記事範囲の最初と最後の記事番号です。( *response*, *list* ) のペアを返します。*list* は (*id*, *text*) のペアからなるリストで、*id* が (文字列で表した) 記事 ID、*text* がその記事のヘッダテキストです。*file* 引数が指定されている場合、‘XHDR’ コマンドの出力結果はファイルに格納されます。*file* が文字列の場合、この文字列をファイル名としてファイルをオープンし、書き込み後にクローズします。*file* がファイルオブジェクトの場合、オブジェクトの `write()` メソッドを呼び出して出力結果を格納します。*file* が指定されている場合は戻



り値として空のリストを返します。

**post**(*file*)

‘POST’ 命令を使って記事をポストします。*file* 引数は開かれているファイルオブジェクトで、その内容は `readline()` メソッドを使って EOF まで読み出されます。内容は必要なヘッダを含め、正しい形式のニュース記事でなければなりません。`post()` メソッドは ‘.’ で始まる行を自動的にエスケープします。

**ihave**(*id*, *file*)

‘IHAVE’ 命令を送信します。応答がエラーでない場合、*file* を `post()` と全く同じように扱います。

**date**()

タプル (*response*, *date*, *time*) を返します。このタプルには `newnews()` および `newgroups()` メソッドに合った形式の、現在の日付および時刻が入っています。これはオプションの NNTP 拡張なので、全てのサーバでサポートされているとは限りません。

**xgtitle**(*name*, [*file*])

‘XGTITLE’ 命令を処理し、(*response*, *list*) からなるペアを返します。*list* は (*name*, *title*) を含むタプルのリストです。*file* 引数が指定されている場合、‘XHDR’ コマンドの出力結果はファイルに格納されます。*file* が文字列の場合、この文字列をファイル名としてファイルをオープンし、書き込み後にクローズします。*file* がファイルオブジェクトの場合、オブジェクトの `write()` メソッドを呼び出して出力結果を格納します。*file* が指定されている場合は戻り値として空のリストを返します。これはオプションの NNTP 拡張なので、全てのサーバでサポートされているとは限りません。

**xover**(*start*, *end*, [*file*])

(*resp*, *list*) からなるペアを返します。*list* はタプルからなるリストで、各タプルは記事番号 *start* および *end* の間に区切られた記事です。各タプルは (*article number*, *subject*, *poster*, *date*, *id*, *references*, *size*, *lines*) の形式をとります。*file* 引数が指定されている場合、‘XHDR’ コマンドの出力結果はファイルに格納されます。*file* が文字列の場合、この文字列をファイル名としてファイルをオープンし、書き込み後にクローズします。*file* がファイルオブジェクトの場合、オブジェクトの `write()` メソッドを呼び出して出力結果を格納します。*file* が指定されている場合は戻り値として空のリストを返します。これはオプションの NNTP 拡張なので、全てのサーバでサポートされているとは限りません。

**xpath**(*id*)

(*resp*, *path*) からなるペアを返します。*path* はメッセージ ID が *id* である記事のディレクトリパスです。これはオプションの NNTP 拡張なので、全てのサーバでサポートされているとは限りません。

**quit**()

‘QUIT’ 命令を送信し、接続を閉じます。このメソッドを呼び出した後は、NNTP オブジェクトの他のいかなるメソッドも呼び出してはいけません。

## 11.12 smtpplib — SMTP プロトコル クライアント

`smtpplib` モジュールは、SMTP または ESMTP のリスナーデーモンを備えた任意のインターネット上のホストにメールを送るために使用することができる SMTP クライアント・セッション・オブジェクトを定義します。SMTP および ESMTP オペレーションの詳細は、RFC 821 (*Simple Mail Transfer Protocol*) や RFC 1869 (*SMTP Service Extensions*) を調べてください。

**class SMTP**( [*host*[, *port*[, *local\_hostname*]] )

SMTP インスタンスは SMTP コネクションをカプセル化し、SMTP と ESMTP の命令をサポートします。オプションである *host* と *port* を与えた場合は、SMTP クラスのインスタンスが作成されると同時に、`connect()` メソッドを呼び出し初期化されます。また、ホストから応答が無い場合は、



SMTPConnectError が上げられます。

普通に使う場合は、初期化と接続を行ってから、`sendmail()` と `quit()` メソッドを呼びます。使用例は先の方で記載しています。

このモジュールの例外には次のものがあります：

**exception SMTPException**

このモジュールの例外クラスのベースクラスです。

**exception SMTPServerDisconnected**

この例外はサーバが突然コネクションを切断するか、もしくは SMTP インスタンスを生成する前にコネクションを張ろうとした場合に上げられます。

**exception SMTPResponseException**

SMTP のエラーコードを含んだ例外のクラスです。これらの例外は SMTP サーバがエラーコードを返すときに生成されます。エラーコードは `smtp_code` 属性に格納されます。また、`smtp_error` 属性にはエラーメッセージが格納されます。

**exception SMTPSenderRefused**

送信者のアドレスが弾かれたときに上げられる例外です。全ての SMTPResponseException 例外に、SMTP サーバが弾いた 'sender' アドレスの文字列がセットされます。

**exception SMTPRecipientsRefused**

全ての受取人アドレスが弾かれたときに上げられる例外です。各受取人のエラーは属性 `recipients` によってアクセス可能で、SMTP.sendmail() が返す辞書と同じ並びの辞書になっています。

**exception SMTPDataError**

SMTP サーバが、メッセージのデータを受け入れることを拒絶した時に上げられる例外です。

**exception SMTPConnectError**

サーバへの接続時にエラーが発生した時に上げられる例外です。

**exception SMTPHeloError**

サーバが 'HELO' メッセージを弾いた時に上げられる例外です。

参考資料：

RFC 821, “*Simple Mail Transfer Protocol*”

SMTP のプロトコル定義です。このドキュメントでは SMTP のモデル、操作手順、プロトコルの詳細についてカバーしています。

RFC 1869, “*SMTP Service Extensions*”

SMTP に対する ESMTP 拡張の定義です。このドキュメントでは、新たな命令による SMTP の拡張、サーバによって提供される命令を動的に発見する機能のサポート、およびいくつかの追加命令定義について記述しています。

## 11.12.1 SMTP オブジェクト

SMTP クラスインスタンスは次のメソッドを提供します：

**set\_debuglevel(*level*)**

コネクション間でやりとりされるメッセージ出力のレベルをセットします。メッセージの冗長さは *level* に応じて決まります。

**connect([*host*[, *port*]])**

ホスト名とポート番号をもとに接続します。デフォルトは localhost の標準的な SMTP ポート (25 番) に接続します。もしホスト名の末尾がコロン (':') で、後に番号がついている場合は、「ホスト名:ポート番号」として扱われます。このメソッドはコンストラクタにホスト名及びポート番号が指定されて

いる場合、自動的に呼び出されます。

`docmd(cmd, [argstring])`

サーバへコマンド `cmd` を送信します。オプション引数 `argstring` はスペース文字でコマンドに連結します。戻り値は、整数値のレスポンスコードと、サーバからの応答の値をタプルで返します。(サーバからの応答が数行に渡る場合でも一つの大きな文字列で返します。)

通常、この命令を明示的に使う必要はありませんが、自分で拡張するする時に使用するとき役に立つかもしれません。

応答待ちのときに、サーバへのコネクションが失われると、`SMTPServerDisconnected` が上がります。

`helo([hostname])`

SMTP サーバに 'HELO' コマンドで身元を示します。デフォルトでは `hostname` 引数はローカルホストを指します。

通常は `sendmail()` が呼び出すため、これを明示的に呼び出す必要はありません。

`ehlo([hostname])`

'EHLO' を利用し、ESMTP サーバに身元を明かします。デフォルトでは `hostname` 引数はローカルホストを指します。

また、ESMTP オプションのために応答を調べたものは、`has_extn()` に備えて保存されます。

`has_extn()` をメールを送信する前に使わない限り、明示的にこのメソッドを呼び出す必要があるべきではなく、`sendmail()` が必要とした場合に呼ばれます。

`has_extn(name)`

`name` が拡張 SMTP サービス可能な場合には 1 を返し、そうでなければ 0 を返します。

`verify(address)`

'VRFY' を利用して SMTP サーバにアドレスの妥当性をチェックします。妥当である場合はコード 250 と完全な RFC 822 アドレス (人名) のタプルを返します。それ以外の場合は、400 以上のエラーコードとエラー文字列を返します。

注意: ほとんどのサイトはスパマーの裏をかくために SMTP の 'VRFY' は使用不可になっています。

`login(user, password)`

認証が必要な SMTP サーバにログインします。認証に使用する引数はユーザ名とパスワードです。まだセッションが無い場合は、'EHLO' または 'HELO' コマンドでセッションを作ります。ESMTP の場合は 'EHLO' が先に試されます。認証が成功した場合は通常このメソッドは戻りますが、例外が起こった場合は以下の例外が上がります:

`SMTPHelloError`サーバが 'HELO' に返答できなかった。

`SMTPAuthenticationError`サーバがユーザ名/パスワードでの認証に失敗した。

`SMTPError`どんな認証方法も見付からなかった。

`starttls([keyfile[, certfile]])`

TLS(Transport Layer Security) モードで SMTP コネクションを出し、全ての SMTP コマンドは暗号化されます。これは `ehlo()` をもう一度呼び出すときにするべきです。

`keyfile` と `certfile` が提供された場合に、`socket` モジュールの `ssl()` 関数が通るようになります。

`sendmail(from_addr, to_addrs, msg[, mail_options, rcpt_options])`

メールを送信します。必要な引数は RFC 822 の `from` アドレス文字列、RFC 822 の `to` アドレス文字列のリスト、メッセージ文字列です。送信側は 'MAIL FROM' コマンドで使用される `mail_options` の ESMTP オプション ('8bitmime' のような) のリストを得るかもしれません。

全ての 'RCPT' コマンドで使われるべき ESMTP オプション (例えば 'DSN' コマンド) は、*rcpt\_options* を通して利用することができます。(もし送信先別に ESMTP オプションを使う必要があれば、メッセージを送るために *mail*、*rcpt*、*data* といった下位レベルのメソッドを使う必要があります。)

注意: 配送エージェントは *from\_addr*、*to\_addrs* 引数を使い、メッセージのエンベロープを構成します。SMTP はメッセージヘッダを修正しません。

まだセッションが無い場合は、'EHLO' または 'HELO' コマンドでセッションを作ります。ESMTP の場合は 'EHLO' が先に試されます。また、サーバが ESMTP 対応ならば、メッセージサイズとそれぞれ指定されたオプションも渡します。(feature オプションがあればサーバの広告をセットします) 'EHLO' が失敗した場合は、ESMTP オプションの無い 'HELO' が試されます。

このメソッドはメールが受け入れられたときは普通に返りますが、そうでない場合は例外を投げます。このメソッドが例外を投げられなければ、誰かが送信したメールを得るべきです。また、例外を投げなかった場合は、拒絶された受取人ごとへの 1 つのエントリーと共に、辞書を返します。各エントリーは、サーバによって送られた SMTP エラーコードおよびエラーメッセージのタプルを含んでいます。

このメソッドは次の例外を上げることがあります:

**SMTPRecipientsRefused** 全ての受信を拒否され、誰にもメールが届けられませんでした。例外オブジェクトの *recipients* 属性は、受信拒否についての情報の入った辞書オブジェクトです。(辞書は少なくとも一つは受信されたときに似ています)。

**SMTPHeloError** サーバが 'HELP' に返答しませんでした。

**SMTPSenderRefused** サーバが *from\_addr* を弾きました。

**SMTPDataError** サーバが予期しないエラーコードを返しました。(受信拒否以外)

また、この他の注意として、例外が上がった後もコネクションは開いたままになっています。

**quit()**

SMTP セッションを終了し、コネクションを閉じます。

下位レベルのメソッドは標準 SMTP/ESMTP コマンド 'HELP'、'RSET'、'NOOP'、'MAIL'、'RCPT'、'DATA' に対応しています。通常これらは直接呼ぶ必要はなく、また、ドキュメント也没有ありません。詳細はモジュールのコードを調べてください。

## 11.12.2 SMTP 使用例

次の例は最低限必要なメールアドレス ('To' と 'From') を含んだメッセージを送信するものです。この例では RFC 822 ヘッダの加工もしていません。メッセージに含まれるヘッダは、メッセージに含まれる必要があり、特に、明確な 'To'、と 'From' アドレスはメッセージヘッダに含まれている必要があります。

```

import smtplib
import string

def prompt(prompt):
    return raw_input(prompt).strip()

fromaddr = prompt("From: ")
toaddrs = prompt("To: ").split()
print "Enter message, end with ^D (Unix) or ^Z (Windows):"

# Add the From: and To: headers at the start!
msg = ("From: %s\r\nTo: %s\r\n\r\n"
       % (fromaddr, ", ".join(toaddrs, ", ")))
while 1:
    try:
        line = raw_input()
    except EOFError:
        break
    if not line:
        break
    msg = msg + line

print "Message length is " + repr(len(msg))

server = smtplib.SMTP('localhost')
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()

```

## 11.13 telnetlib — Telnet クライアント

telnetlib モジュールでは、Telnet プロトコルを実装している Telnet クラスを提供します。Telnet プロトコルについての詳細は RFC 854 を参照してください。加えて、このモジュールでは Telnet プロトコルにおける制御文字 (下を参照してください) と、telnet オプションに対するシンボル定数を提供しています。telnet オプションに対するシンボル名は arpa/telnet.h の TELOPT\_ がない状態での定義に従います。伝統的に arpa/telnet.h に含まれていない telnet オプションのシンボル名については、このモジュールのソースコード自体を参照してください。

telnet コマンドのシンボル定数は、IAC、DONT、DO、WONT、WILL、SE (サブネゴシエーション終了)、NOP (何もしない)、DM (データマーク)、BRK (ブレーク)、IP (プロセス割り込み)、AO (出力中断)、AYT (応答確認)、EC (文字削除)、EL (行削除)、GA (進め)、SB (サブネゴシエーション開始) です。

```
class Telnet ([host[, port]])
```

Telnet は Telnet サーバへの接続を表現します。標準では、Telnet クラスのインスタンスは最初はサーバに接続していません; 接続を確立するには `open()` を使わなければなりません。別の方法として、コンストラクタにホスト名とオプションのポート番号を渡すことができます。この場合はコンストラクタの呼び出しが返る以前にサーバへの接続が確立されます。

すでに接続の開かれているインスタンスを再度開いてはいけません。

このクラスは多くの `read_*()` メソッドを持っています。これらのメソッドのいくつかは、接続の終端を示す文字を読み込んだ場合に `EOFError` を送出するので注意してください。例外を送出するのは、これらの関数が終端に到達しなくても空の文字列を返す可能性があるからです。詳しくは下記の個々の説明を参照してください。

参考資料:

RFC 854, “*Telnet プロトコル仕様 (Telnet Protocol Specification)*”  
Telnet プロトコルの定義。

### 11.13.1 Telnet オブジェクト

Telnet インスタンスは以下のメソッドを持っています:

`read_until(expected[, timeout])`

`expected` で指定された文字列を読み込むか、`timeout` で指定された秒数が経過するまで読み込みます。

与えられた文字列に一致する部分が見つからなかった場合、読み込むことができたものを返します。これは空の文字列になる可能性があります。接続が閉じられ、転送処理済みのデータが得られない場合には `EOFError` が送出されます。

`read_all()`

EOF に到達するまでの全てのデータを読み込みます; 接続が閉じられるまでブロックします。

`read_some()`

EOF に到達しない限り、少なくとも 1 バイトの転送処理済みデータを読み込みます。EOF に到達した場合は `''` を返します。すぐに読み出せるデータが存在しない場合にはブロックします。

`read_very_eager()`

I/O によるブロックを起こさずに読み出せる全てのデータを読み込みます (eager モード)。

接続が閉じられており、転送処理済みのデータとして読み出せるものがない場合には `EOFError` が送出されます。それ以外の場合で、単に読み出せるデータがない場合には `''` を返します。IAC シーケンス操作中でないかぎりブロックしません。

`read_eager()`

現在すぐに読み出せるデータを読み出します。

接続が閉じられており、転送処理済みのデータとして読み出せるものがない場合には `EOFError` が送出されます。それ以外の場合で、単に読み出せるデータがない場合には `''` を返します。IAC シーケンス操作中でないかぎりブロックしません。

`read_lazy()`

すでにキューに入っているデータを処理して返します (lazy モード)。

接続が閉じられており、読み出せるデータがない場合には `EOFError` を送出します。それ以外の場合で、転送処理済みのデータで読み出せるものがない場合には `''` を返します。IAC シーケンス操作中でないかぎりブロックしません。

`read_very_lazy()`

すでに処理済みキューに入っているデータを処理して返します (very lazy モード)。

接続が閉じられており、読み出せるデータがない場合には `EOFError` を送出します。それ以外の場合で、転送処理済みのデータで読み出せるものがない場合には `''` を返します。このメソッドは決してブロックしません。

`read_sb_data()`

SB/SE ペア (サブオプション開始 / 終了) の間に収集されたデータを返します。SE コマンドによって起動されたコールバック関数はこれらのデータにアクセスしなければなりません。

このメソッドは決してブロックしません。2.3 で追加された仕様です。

`open(host[, port])`

サーバホストに接続します。第二引数はオプションで、ポート番号を指定します。標準の値は通常の Telnet ポート番号 (23) です。

すでに接続しているインスタンスで再接続を試みてはいけません。

`msg(msg[, *args])`

デバッグレベルが  $> 0$  のとき、デバッグ用のメッセージを出力します。追加の引数が存在する場合、標準の文字列書式化演算子 `%` を使って `msg` 中の書式指定子に代入されます。

`set_debuglevel(debuglevel)`

デバッグレベルを設定します。`debuglevel` が大きくなるほど、(`sys.stdout` に) デバッグメッセージがたくさん出力されます。

`close()`

接続を閉じます。

`get_socket()`

内部的に使われているソケットオブジェクトです。

`fileno()`

内部的に使われているソケットオブジェクトのファイル記述子です。

`write(buffer)`

ソケットに文字列を書き込みます。このとき IAC 文字については 2 度送信します。接続がブロックした場合、書き込みがブロックする可能性があります。接続が閉じられた場合、`socket.error` が送出されるかもしれません。

`interact()`

非常に低機能の telnet クライアントをエミュレートする対話関数です。

`mt_interact()`

`interact()` のマルチスレッド版です。

`expect(list[, timeout])`

正規表現のリストのうちどれか一つにマッチするまでデータを読みます。

第一引数は正規表現のリストです。コンパイルされたもの (`re.RegexObject` のインスタンス) でも、コンパイルされていないもの (文字列) でもかまいません。オプションの第二引数はタイムアウトで、単位は秒です; 標準の値は無期限に設定されています。

3 つの要素からなるタプル: 最初にマッチした正規表現のインデクス; 返されたマッチオブジェクト; マッチ部分を含む、マッチするまでに読み込まれたテキストデータ、を返します。

ファイル終了子が見つかり、かつ何もテキストデータが読み込まれなかった場合、`EOFError` が送出されます。そうでない場合で何もマッチしなかった場合には `(-1, None, text)` が返されます。ここで `text` はこれまで受信したテキストデータです (タイムアウトが発生した場合には空の文字列になる場合もあります)。

正規表現の末尾が (`[.*]` のような) 貪欲マッチングになっている場合や、入力に対して 1 つ以上の正規表現がマッチする場合には、その結果は決定不能で、I/O のタイミングに依存するでしょう。

`set_option_negotiation_callback(callback)`

telnet オプションが入力フローから読み込まれるたびに、`callback` が (設定されていれば) 以下の引数形式: `callback(telnet socket, command (DO/DONT/WILL/WONT), option)` で呼び出されます。その後 telnet オプションに対しては `telnetlib` は何も行いません。

## 11.13.2 Telnet Example

典型的な使い方を表す単純な例を示します:



```

import getpass
import sys
import telnetlib

HOST = "localhost"
user = raw_input("Enter your remote account: ")
password = getpass.getpass()

tn = telnetlib.Telnet(HOST)

tn.read_until("login: ")
tn.write(user + "\n")
if password:
    tn.read_until("Password: ")
    tn.write(password + "\n")

tn.write("ls\n")
tn.write("exit\n")

print tn.read_all()

```

## 11.14 urlparse — URL を解析して構成要素にする

このモジュールでは URL (Uniform Resource Locator) 文字列をその構成要素 (アドレススキーム、ネットワーク上の位置、パスその他) に分解したり、構成要素を URL に組みなおしたり、“相対 URL (relative URL)” を指定した “基底 URL (base URL)” に基づいて絶対 URL に変換するための標準的なインタフェースを定義しています。

このモジュールは相対 URL のインターネット RFC に対応するように設計されました (そして RFC の初期ドラフトのバグを発見しました！)。

以下の関数が定義されています:

**urlparse**(*urlstring*[, *default\_scheme*[, *allow\_fragments*]])

URL を解釈して 6 つの構成要素にし、6 要素のタプル: (アドレススキーム、ネットワーク上の位置、パス、パラメタ、クエリ、フラグメント指定子) を返します。このタプルは URL の一般的な構造: *scheme*://*netloc*/*path*;*parameters*?*query*#*fragment* に対応しています。各タプル要素は文字列で、空の場合もあります。構成要素がさらに小さい要素に分解されることはありません (例えばネットワーク上の位置は単一の文字列になります)。また % によるエスケープは展開されません。上で示された区切り文字がタプルの各要素の一部として含まれることはありませんが、*path* 要素の先頭のスラッシュがある場合には例外です。

以下の例:

```
urlparse('http://www.cwi.nl:80/%7Eguido/Python.html')
```

では、タプル

```
('http', 'www.cwi.nl:80', '/%7Eguido/Python.html', '', '', '')
```

になります。

*default\_scheme* 引数が最低されている場合、標準のアドレススキームを表し、アドレススキームを指定していない URL 文字列に対してのみ使われます。この引数の標準の値は空文字列です。

`allow_fragments` 引数がゼロの場合、URL のアドレススキームがフラグメント指定をサポートしていても指定できなくなります。この引数の標準の値は 1 です。

`urlunparse(tuple)`

`urlparse()` が返すような形式のタプルから URL 文字列を構築します。解析された元の URL が、例えばクエリ内容が空の ? のような冗長な区切り文字を持っていた場合には、多少違いはあるが等価な URL になるかもしれません。(RFC のドラフトの時点では、これらは等価でした)。

`urlsplit(urlstring[, default_scheme[, allow_fragments]])`

`urlparse()` に似ていますが、URL から `params` を切り離しません。このメソッドは通常、URL の `path` 部分において、各セグメントにパラメタ指定をできるようにした最近の URL 構文 (RFC 2396 参照) の場合に、`urlparse()` の代わりに使われます。パスセグメントとパラメタを分割するためには分割用の関数が必要です。この関数は 5 要素のタプル: (アドレススキーム、ネットワーク上の位置、パス、クエリ、フラグメント指定子) を返します。2.2 で追加された仕様です。

`urlunsplit(tuple)`

`urlsplit()` が返すような形式のタプル中のエレメントを組み合わせて、文字列の完全な URL にします。2.2 で追加された仕様です。

`urljoin(base, url[, allow_fragments])`

“基底 URL” (`base`) と “相対 URL” (`url`) を組み合わせて、完全な URL (“絶対 URL”) を構成します。ぶっちゃけ、この関数は 基底 URL の要素、特にアドレススキーム、ネットワーク上の位置、およびパス (の一部) を使って、相対 URL にない要素を提供します。

以下の例:

```
urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
```

では、文字列

```
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

になります。

`allow_fragments` 引数は `urlparse()` における引数と同じ意味を持ちます。

`urldefrag(url)`

`url` がフラグメント指定子を含む場合、フラグメント指定子を持たないバージョンに修正された `url` と、別の文字列に分割されたフラグメント指定子を返します。`url` 中にフラグメント指定子がない場合、そのままの `url` と空文字列を返します。

参考資料:

RFC 1738, “*Uniform Resource Locators (URL)*”

この RFC では絶対 URL の形式的な文法と意味付けを仕様化しています。

RFC 1808, “*Relative Uniform Resource Locators*”

この RFC には絶対 URL と相対 URL を結合するための規則がボールドケースの取扱い方を決定する “異常な例” つきで収められています。

RFC 2396, “*Uniform Resource Identifiers (URI): Generic Syntax*”

この RFC では Uniform Resource Name (URN) と Uniform Resource Locator (URL) の両方に対する一般的な文法的要求事項を記述しています。

## 11.15 SocketServer — ネットワークサーバ構築のためのフレームワーク

SocketServer モジュールはネットワークサーバを実装するタスクを単純化します。

このモジュールには 4 つのサーバクラスがあります: TCPServer は、クライアントとサーバ間に継続的なデータ流路を提供する、インターネット TCP プロトコルを使います。UDPServer は、順序通りに到着しなかったり、転送中に喪失してしまってもかまわない情報の断続的なパケットである、データグラムを使います。UnixStreamServer および UnixDatagramServer クラスも同様ですが、UNIX ドメインソケットを使います; 従って非 UNIX プラットフォームでは利用できません。ネットワークプログラミングについての詳細は、W. Richard Steven 著 *UNIX Network Programming* や、Ralph Davis 著 *Win32 Network Programming* のような書籍を参照してください。

これらの 4 つのクラスは要求を同期的に (*synchronously*) 処理します; 各要求は次の要求を開始する前に完結していなければなりません。同期的な処理は、サーバで大量の計算を必要とする、あるいはクライアントが処理するには時間がかかりすぎるような大量のデータを返す、といった理由によってリクエストに長い時間がかかる状況には向いていません。こうした状況の解決方法は別のプロセスを生成するか、個々の要求を扱うスレッドを生成することです; ForkingMixIn および ThreadingMixIn 混合クラス (mix-in classes) を使えば、非同期的な動作をサポートできます。

サーバの作成にはいくつかのステップがあります。最初に、BaseRequestHandler クラスをサブクラス化して要求処理クラス (request handler class) を生成し、その `handle()` メソッドを上書きしなければなりません; このメソッドで入力される要求を処理します。次に、サーバクラスのうち一つをインスタンス化して、サーバのアドレスと要求処理クラスを渡さなければなりません。最後に、サーバオブジェクトの `handle_request()` または `serve_forever()` メソッドを呼び出して、単一または多数の要求を処理します。

ThreadingMixIn から継承してスレッドを利用した接続を行う場合、突発的な通信切断時の処理を明示的に指定する必要があります。ThreadingMixIn クラスには `daemon_threads` 属性があり、サーバがスレッドの終了を待ち合わせるかどうかを指定する事ができます。スレッドが独自の処理を行う場合は、このフラグを明示的に指定します。デフォルトは `False` で、Python は ThreadingMixIn クラスが起動した全てのスレッドが終了するまで実行し続けます。

サーバクラス群は使用するネットワークプロトコルに関わらず、同じ外部メソッドおよび属性を持ちます:

**`fileno()`**

サーバが要求待ちを行っているソケットのファイル記述子を整数で返します。この関数は一般的に、同じプロセス中の複数のサーバを監視できるようにするために、`select.select()` に渡されます。

**`handle_request()`**

単一の要求を処理します。この関数は以下のメソッド: `get_request()`、`verify_request()`、および `process_request()` を順番に呼び出します。ハンドラ中でユーザによって提供された `handle()` が例外を送出した場合、サーバの `handle_error()` メソッドが呼び出されます。

**`serve_forever()`**

無限個の要求を処理します。この関数は単に無限ループ内で `handle_request()` を呼び出します。

**`address_family`**

サーバのソケットが属しているプロトコルファミリです。取りえる値は `socket.AF_INET` および `socket.AF_UNIX` です。

**`RequestHandlerClass`**

ユーザが提供する要求処理クラスです; 要求ごとにこのクラスのインスタンスが生成されます。

**`server_address`**

サーバが要求待ちを行うアドレスです。アドレスの形式はプロトコルファミリによって異なります。詳細は `socket` モジュールを参照してください。インターネットプロトコルでは、この値は例えば `('127.0.0.1', 80)` のようにアドレスを与える文字列と整数のポート番号を含むタプルです。

#### `socket`

サーバが入力の要求待ちを行うためのソケットオブジェクトです。

サーバクラスは以下のクラス変数をサポートします:

#### `allow_reuse_address`

サーバがアドレスの再使用を許すかどうかを示す値です。この値は標準で `False` で、サブクラスで再使用ポリシーを変更するために設定することができます。

#### `request_queue_size`

要求待ち行列 (queue) のサイズです。単一の要求を処理するのに長時間かかる場合には、サーバが処理中に届いた要求は最大 `request_queue_size` 個まで待ち行列に置かれます。待ち行列が一杯になると、それ以降のクライアントからの要求は“接続拒否 (Connection denied)” エラーになります。標準の値は通常 5 ですが、この値はサブクラスで上書きすることができます。

#### `socket_type`

サーバが使うソケットの型です; 取りえる値は 2 つで、`socket.SOCK_STREAM` と `socket.SOCK_DGRAM` です。

`TCPServer` のような基底クラスのサブクラスで上書きできるサーバメソッドは多数あります; これらのメソッドはサーバオブジェクトの外部のユーザにとっては役にたたないものです。

#### `finish_request()`

`RequestHandlerClass` をインスタンス化し、`handle()` メソッドを呼び出して、実際に要求を処理します。

#### `get_request()`

ソケットから要求を受理して、クライアントとの通信に使われる新しいソケットオブジェクト、およびクライアントのアドレスからなる、2 要素のタプルを返します。

#### `handle_error(request, client_address)`

この関数は `RequestHandlerClass` の `handle()` メソッドが例外を送出した際に呼び出されます。標準の動作では標準出力ヘトレースバックを出力し、後続する要求を継続して処理します。

#### `process_request(request, client_address)`

`finish_request()` を呼び出して、`RequestHandlerClass` のインスタンスを生成します。必要なら、この関数から新たなプロセススレッドを生成して要求を処理することができます; その処理は `ForkingMixIn` または `ThreadingMixIn` クラスが行います。

#### `server_activate()`

サーバのコンストラクタによって呼び出され、サーバを活動状態にします。このメソッドは上書きできます。

#### `server_bind()`

サーバのコンストラクタによって呼び出され、適切なアドレスにソケットをバインドします。このメソッドは上書きできます。

#### `verify_request(request, client_address)`

ブール値を返さなければなりません; 値が真の場合には要求が処理され、偽の場合には要求は拒否されます。サーバへのアクセス制御を実装するためにこの関数を上書きすることができます。標準の実装では常に真を返します。

要求処理クラスでは、新たな `handle()` メソッドを定義しなくてはならず、また以下のメソッドのいずれかを上書きすることができます。各要求ごとに新たなインスタンスが生成されます。

**finish()**

`handle()` メソッドが呼び出された後、何らかの後始末を行うために呼び出されます。標準の実装では何も行いません。`setup()` または `handle()` が例外を送出した場合には、この関数は呼び出されません。

**handle()**

この関数では、クライアントからの要求を実現するために必要な全ての作業を行わなければなりません。この作業の上で、いくつかのインスタンス属性を利用することができます; クライアントからの要求は `self.request` です; クライアントのアドレスは `self.client_address` です; そしてサーバごとの情報にアクセスする場合には、サーバインスタンスを `self.server` で取得できます。

`self.request` の型はサービスがデータグラム型かストリーム型かで異なります。ストリーム型では、`self.request` はソケットオブジェクトです; データグラムサービスでは、`self.request` は文字列になります。しかし、この違いは要求処理配合クラスの `StreamRequestHandler` や `Data-gramRequestHandler` を使うことで隠蔽することができます。これらのクラスでは `setup()` および `finish()` メソッドを上書きしており、`self.rfile` および `self.wfile` 属性を提供しています。`self.rfile` および `self.wfile` は、要求データを取得したりクライアントにデータを返すために、それぞれ読み出し、書き込みを行うことができます。

**setup()**

`handle()` メソッドより前に呼び出され、何らかの必要な初期化処理を行います。標準の実装では何も行いません。

## 11.16 BaseHTTPServer — 基本的な機能を持つ HTTP サーバ

このモジュールでは、HTTP サーバ (Web サーバ) を実装するための二つのクラスを定義しています。通常、このモジュールが直接使用されることはなく、特定の機能を持つ Web サーバを構築するために使われます。`SimpleHTTPServer` および `CGIHTTPServer` モジュールを参照してください。

最初のクラス、`HTTPServer` は `SocketServer.TCPServer` のサブクラスです。`HTTPServer` は HTTP ソケットを生成してリクエスト待ち (listen) を行い、リクエストをハンドラに渡します。サーバを作成して動作させるためのコードは以下のようになります:

```
def run(server_class=BaseHTTPServer.HTTPServer,
        handler_class=BaseHTTPServer.BaseHTTPRequestHandler):
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    httpd.serve_forever()
```

**class HTTPServer (server\_address, RequestHandlerClass)**

このクラスは `TCPServer` 型のクラスの上に構築されており、サーバのアドレスをインスタンス変数 `server_name` および `server_port` に記憶します。サーバはハンドラからアクセス可能で、通常 `server` インスタンス変数でアクセスします。

**class BaseHTTPRequestHandler (request, client\_address, server)**

このクラスはサーバに到着したリクエストを処理します。このメソッド自体では、実際のリクエストに応答することはできません; (GET や POST のような) 各リクエストメソッドを処理するためにはサブクラス化しなければなりません。`BaseHTTPRequestHandler` では、サブクラスで使うためのクラスやインスタンス変数、メソッド群を数多く提供しています。

このハンドラはリクエストを解釈し、次いでリクエスト形式ごとに固有のメソッドを呼び出します。メソッド名はリクエストの名称から構成されます。例えば、リクエストメソッド 'SPAM' に対しては、



`do_SPAN()` メソッドが引数なしで呼び出されます。リクエストに関連する情報は全て、ハンドラのインスタンス変数に記憶されています。サブクラスでは `__init__()` メソッドを上書きしたり拡張したりする必要はありません。

`BaseHTTPRequestHandler` は以下のインスタンス変数を持っています:

**client\_address**

HTTP クライアントのアドレスを参照している、`(host, port)` の形式をとるタプルが入っています。

**command**

HTTP 命令 (リクエスト形式) が入っています。例えば `'GET'` です。

**path**

リクエストされたパスが入っています。

**request\_version**

リクエストのバージョン文字列が入っています。例えば `'HTTP/1.0'` です。

**headers**

`MessageClass` クラス変数で指定されたクラスのインスタンスを保持しています。このインスタンスは HTTP リクエストのヘッダを解釈し、管理しています。

**rfile**

入力ストリームが入っており、そのファイルポインタはオプション入力データ部の先頭を指しています。

**wfile**

クライアントに返送する応答を書き込むための出力ストリームが入っています。このストリームに書き込む際には、HTTP プロトコルに従った形式をとらなければなりません。

`BaseHTTPRequestHandler` は以下のクラス変数を持っています:

**server\_version**

サーバのソフトウェアバージョンを指定します。この値は上書きする必要が生じるかもしれません。書式は複数の文字列を空白で分割したもので、各文字列はソフトウェア名 [/バージョン] の形式をとります。例えば、`'BaseHTTP/0.2'` です。

**sys\_version**

Python 処理系のバージョンが、`version_string` メソッドや `server_version` クラス変数で利用可能な形式で入っています。例えば `'Python/1.4'` です。

**error\_message\_format**

クライアントに返すエラー応答を構築するための書式化文字列を指定します。この文字列は丸括弧で囲ったキー文字列で指定する形式を使うので、書式化の対象となる値は辞書でなければなりません。キー `code` は整数で、HTTP エラーコードを特定する数値です。`message` は文字列で、何が発生したかを表す (詳細な) エラーメッセージが入ります。`explain` はエラーコード番号の説明です。`message` および `explain` の標準の値は `response` クラス変数で見つけることができます。

**protocol\_version**

この値には応答に使われる HTTP プロトコルのバージョンを指定します。`'HTTP/1.1'` に設定されると、サーバは持続的 HTTP 接続を許可します; しかしその場合、サーバは全てのクライアントに対する応答に、正確な値を持つ `Content-Length` ヘッダを (`send_header()` を使って) 含めなければなりません。以前のバージョンとの互換性を保つため、標準の設定値は `'HTTP/1.0'` です。

**MessageClass**

HTTP ヘッダを解釈するための `rfc822.Message` 類似のクラスを指定します。通常この値が上書きされることはなく、標準の値 `mimetools.Message` になっています。

**responses**



この変数はエラーコードを表す整数を二つの要素をもつタプルに対応付けます。タプルには短いメッセージと長いメッセージが入っています。例えば、`{code: (shortmessage, longmessage)}` といったようになります。`shortmessage` は通常、エラー応答における `message` キーの値として使われ、`longmessage` は `explain` キーの値として使われます (`error_message_format` クラス変数を参照してください)。

`BaseHTTPRequestHandler` インスタンスは以下のメソッドを持っています:

**handle()**

`handle_one_request()` を一度だけ (持続的接続が有効になっている場合には複数回) 呼び出して、HTTP リクエストを処理します。このメソッドを上書きする必要はまったくありません; そうする代わりに適切な `do_*`() を実装してください。

**handle\_one\_request()**

このメソッドはリクエストを解釈し、適切な `do_*`() メソッドに転送します。このメソッドを上書きする必要はまったくありません。

**send\_error(*code*[, *message*])**

完全なエラー応答をクライアントに送信し、ログ記録します。`code` は数値型で、HTTP エラーコードを指定します。`message` はオプションで、より詳細なメッセージテキストです。完全なヘッダのセットが送信された後、`error_message_format` クラス変数を使って組み立てられたテキストが送られます。

**send\_response(*code*[, *message*])**

応答ヘッダを送信し、受理したリクエストをログ記録します。HTTP 応答行が送られた後、`Server` および `Date` ヘッダが送られます。これら二つのヘッダはそれぞれ `version_string()` および `date_time_string()` メソッドで取り出します。

**send\_header(*keyword*, *value*)**

出力ストリームに特定の MIME ヘッダを書き込みます。`keyword` はヘッダのキーワードを指定し、`value` にはその値を指定します。

**end\_headers()**

応答中の MIME ヘッダの終了を示す空行を送信します。

**log\_request([*code*[, *size*]])**

受理された (成功した) リクエストをログに記録します。`code` にはこの応答に関連付けられた HTTP コード番号を指定します。応答メッセージの大きさを知ることができる場合、`size` パラメタに渡すとよいでしょう。

**log\_error(...)**

リクエストを遂行できなかった際に、エラーをログに記録します。標準では、メッセージを `log_message()` に渡します。従って同じ引数 (`format` と追加の値) を取ります。

**log\_message(*format*, ...)**

任意のメッセージを `sys.stderr` にログ記録します。このメソッドは通常、カスタムのエラーログ記録機構を作成するために上書きされます。`format` 引数は標準の `printf` 形式の書式化文字列で、`log_message()` に渡された追加の引数は書式化の入力として適用されます。ログ記録される全てのメッセージには、クライアントのアドレスおよび現在の日付、時刻が先頭に付けられます。

**version\_string()**

サーバソフトウェアのバージョン文字列を返します。この文字列はクラス変数 `server_version` および `sys_version` を組み合わせたものです。

**date\_time\_string()**

メッセージヘッダ向けに書式化された、現在の日付および時刻を返します。

`log_data_time_string()`

ログ記録向けに書式化された、現在の日付および時刻を返します。

`address_string()`

ログ記録向けに書式化された、クライアントのアドレスを返します。このときクライアントの IP アドレスに対する名前解決を行います。

参考資料:

CGIHTTPServer モジュール (11.18 節):

CGI スクリプトをサポートするように拡張されたリクエストハンドラ。

SimpleHTTPServer モジュール (11.17 節):

ドキュメントルートの下にあるファイルに対する要求への応答のみに制限した基本リクエストハンドラ。

## 11.17 SimpleHTTPServer — 簡潔な HTTP リクエストハンドラ

SimpleHTTPServer モジュールはリクエストハンドラ (request-handler) クラスを定義しています。インタフェースは `BaseHTTPServer.BaseHTTPRequestHandler` と互換で、基底ディレクトリにあるファイルだけを提供します。

SimpleHTTPServer モジュールでは以下のクラスを定義しています:

**class SimpleHTTPRequestHandler** (*request, client\_address, server*)

このクラスは、現在のディレクトリ以下にあるファイルを、HTTP リクエストにおけるディレクトリ構造に直接対応付けて提供するために利用されます。

リクエストの解釈のような、多くの作業は基底クラス `BaseHTTPServer.BaseHTTPRequestHandler` で行われます。このクラスは関数 `do_GET()` および `do_HEAD()` を実装しています。

SimpleHTTPRequestHandler では以下のメンバ変数を定義しています:

**server\_version**

この値は `"SimpleHTTP/" + __version__` になります。`__version__` はこのモジュールで定義されている値です。

**extensions\_map**

拡張子を MIME 型指定子に対応付ける辞書です。標準の型指定は空文字列で表され、この値は `text/plain` と見なされます。対応付けは大小文字の区別をするので、小文字のキーのみを入れるべきです。

SimpleHTTPRequestHandler では以下のメソッドを定義しています:

**do\_HEAD()**

このメソッドは 'HEAD' 型のリクエスト処理を実行します: すなわち、GET リクエストの時に送信されるものと同じヘッダを送信します。送信される可能性のあるヘッダについての完全な説明は `do_GET()` メソッドを参照してください。

**do\_GET()**

リクエストを現在の作業ディレクトリからの相対的なパスとして解釈することで、リクエストをローカルシステム上のファイルと対応付けます。

リクエストがディレクトリに対応付けられた場合、出力は 403 応答であり、その後に説明 'Directory listing not supported' が続きます。要求されたファイルを開く際に何らかの `IOError` 例外が送出された場合、リクエストは 404、'File not found' エラーに対応づけられます。そうでない場合、コンテンツタイプが `extensions_map` 変数を用いて推測されます。

出力は 'Content-type:' と推測されたコンテンツタイプで、その後にヘッダの終了を示す空白行が続き、さらにその後にファイルの内容が続きます。このファイルは常にバイナリモードで開かれます。

使用例については関数 `test()` の実装を参照してください。

参考資料:

BaseHTTPServer モジュール (11.16 節):

Web サーバおよび要求ハンドラの基底クラス実装。

## 11.18 CGIHTTPServer — CGI 実行機能付き HTTP リクエスト処理機構

CGIHTTPServer モジュールでは、BaseHTTPServer.BaseHTTPRequestHandler 互換のインタフェースを持ち、SimpleHTTPServer.SimpleHTTPRequestHandler の動作を継承していますが CGI スクリプトを動作することもできる、HTTP 要求処理機構クラスを定義しています。

注意: このモジュールは CGI スクリプトを UNIX および Windows システム上で実行させることができます; Mac OS 上では、自分と同じプロセス内で Python スクリプトを実行することしかできないはずです。

CGIHTTPServer モジュールでは、以下のクラスを定義しています:

`class CGIHTTPRequestHandler(request, client_address, server)`

このクラスは、現在のディレクトリかその下のディレクトリにおいて、ファイルか CGI スクリプト出力を提供するために使われます。HTTP 階層構造からローカルなディレクトリ構造への対応付けは SimpleHTTPServer.SimpleHTTPRequestHandler と全く同じなので注意してください。

このクラスでは、ファイルが CGI スクリプトであると推測された場合、これをファイルとして提供する代わりにスクリプトを実行します。他の一般的なサーバ設定は特殊な拡張子を使って CGI スクリプトであることを示すのに対し、ディレクトリベースの CGI だけが使われます。

`do_GET()` および `do_HEAD()` 関数は、HTTP 要求が `cgi_directories` パス以下のどこかを指している場合、ファイルを提供するのではなく、CGI スクリプトを実行してその出力を提供するように変更されています。

CGIHTTPRequestHandler では以下のデータメンバを定義しています:

`cgi_directories`

この値は標準で `['/cgi-bin', '/htbin']` であり、CGI スクリプトを含んでいることを示すディレクトリを記述します。

CGIHTTPRequestHandler では以下のメソッドを定義しています:

`do_POST()`

このメソッドは、CGI スクリプトでのみ許されている 'POST' 型の HTTP 要求に対するサービスを行います。CGI でない url に対して POST を試みた場合、出力は Error 501, "Can only POST to CGI scripts" になります。

セキュリティ上の理由から、CGI スクリプトはユーザ nobody の UID で動作するので注意してください。CGI スクリプトが原因で発生した問題は、Error 403 に変換されます。

使用例については、`test()` 関数の実装を参照してください。

参考資料:

BaseHTTPServer モジュール (11.16 節):

Web サーバとリクエスト処理機構を実装した基底クラスです。

## 11.19 Cookie — HTTP の状態管理

Cookie モジュールは HTTP の状態管理機能である cookie の概念を抽象化、定義しているクラスです。単純な文字列のみで構成される cookie のほか、シリアル化可能なあらゆるデータ型でクッキーの値を保持するための機能も備えています。

このモジュールは元々 RFC 2109 と RFC 2068 に定義されている構文解析の規則を厳密に守っていました。しかし、MSIE 3.0x がこれらの RFC で定義された文字の規則に従っていないことが判明したため、結局、やや厳密さを欠く構文解析規則にせざるを得ませんでした。

**exception CookieError**

属性や Set-Cookie: ヘッダが正しくないなど、RFC 2109 に合致していないときに発生する例外です。

**class BaseCookie([input])**

このクラスはキーが文字列、値が Morsel インスタンスで構成される辞書風オブジェクトです。値に対するキーを設定するときは、値がキーと値を含む Morsel に変換されることに注意してください。

input が与えられたときは、そのまま load() メソッドへ渡されます。

**class SimpleCookie([input])**

このクラスは BaseCookie の派生クラスで、value\_decode() は与えられた値の正当性を確認するように、value\_encode() は str() で文字列化するようにそれぞれオーバーライドします。

**class SerialCookie([input])**

このクラスは BaseCookie の派生クラスで、value\_decode() と value\_encode() をそれぞれ pickle.loads() と pickle.dumps() を実行するようにオーバーライドします。

リリース 2.3 以降で撤廃された仕様です。このクラスを使ってはいけません! 信頼できない cookie のデータからピクルス化された値を読み込むことは、あなたのサーバ上で任意のコードを実行するためにピクルス化した文字列の作成が可能であることを意味し、重大なセキュリティホールとなります。

**class SmartCookie([input])**

このクラスは BaseCookie の派生クラスで、value\_decode() を、値がピクルス化されたデータとして正当なときは pickle.loads() を実行、そうでないときはその値自体を返すようにオーバーライドします。また value\_encode() を、値が文字列以外のときは pickle.dumps() を実行、文字列のときはその値自体を返すようにオーバーライドします。

リリース 2.3 以降で撤廃された仕様です。SerialCookie と同じセキュリティ上の注意が当てはまります。

関連して、さらなるセキュリティ上の注意があります。後方互換性のため、Cookie モジュールは Cookie というクラス名を SmartCookie のエイリアスとしてエクスポートしています。これはほぼ確実に誤った措置であり、将来のバージョンでは削除することが適当と思われます。アプリケーションにおいて SerialCookie クラスを使うべきでないのと同じ理由で Cookie クラスを使うべきではありません。

参考資料:

RFC 2109, “HTTP State Management Mechanism”

このモジュールが実装している HTTP の状態管理に関する規格です。

### 11.19.1 Cookie オブジェクト

**value\_decode(val)**

文字列表現を値にデコードして返します。戻り値の型はどのようなものでも許されます。このメソッドは BaseCookie において何も実行せず、オーバーライドされるためにだけ存在します。

**value\_encode(val)**

エンコードした値を返します。元の値はどのような型でもかまいませんが、戻り値は必ず文字列となります。このメソッドは `BaseCookie` において何も実行せず、オーバーライドされるためにだけ存在します。

通常 `value_encode()` と `value_decode()` はともに `value_decode` の処理内容から逆算した範囲に収まっていなければなりません。

`output([attrs[, header[, sep]]])`

HTTP ヘッダ形式の文字列表現を返します。`attrs` と `header` はそれぞれ `Morsel` の `output()` メソッドに送られます。`sep` はヘッダの連結に用いられる文字で、デフォルトは改行となっています。

`js_output([attrs])`

ブラウザが JavaScript をサポートしている場合、HTTP ヘッダを送信した場合と同様に動作する埋め込み可能な JavaScript snippet を返します。

`attrs` の意味は `output()` と同じです。

`load(rawdata)`

`rawdata` が文字列であれば、`HTTP_COOKIE` として処理し、その値を `Morsel` として追加します。辞書の場合は次と同様の処理をおこないます。

```
for k, v in rawdata.items():
    cookie[k] = v
```

## 11.19.2 Morsel オブジェクト

`class Morsel()`

RFC 2109 の属性をキーと値で保持する abstract クラスです。

`Morsel` は辞書風のオブジェクトで、キーは次のような RFC 2109 準拠の定数となっています。

- `expires`
- `path`
- `comment`
- `domain`
- `max-age`
- `secure`
- `version`

キーは大文字、小文字が区別されます。

**value**

クッキーの値。

**coded\_value**

実際に送信する形式にエンコードされた cookie の値。

**key**

cookie の名前。

`set(key, value, coded_value)`

メンバ `key`、`value`、`coded_value` に値をセットします。

`isReservedKey(K)`

`K` が `Morsel` のキーであるかどうかを判定します。

`output([attrs[, header]])`

Mosel を HTTP ヘッダ形式の文字列表現にして返します。*attrs* が与えられなければ、デフォルトですべての属性が含まれます。*attrs* を指定するときは属性をリストとして渡さなければなりません。*header* のデフォルトは "Set-Cookie:" です。

`js_output([attrs])`

ブラウザが JavaScript をサポートしている場合、HTTP ヘッダを送信した場合と同様に動作する埋め込み可能な JavaScript snippet を返します。

*attrs* の意味は `output()` と同じです。

`OutputString([attrs])`

Mosel の文字列表現を HTTP や JavaScript で囲まずに出力します。

*attrs* の意味は `output()` と同じです。

### 11.19.3 例

次の例は Cookie の使い方を示したものです。



```

>>> import Cookie
>>> C = Cookie.SimpleCookie()
>>> C = Cookie.SerialCookie()
>>> C = Cookie.SmartCookie()
>>> C["fig"] = "newton"
>>> C["sugar"] = "wafer"
>>> print C # generate HTTP headers
Set-Cookie: sugar=wafer;
Set-Cookie: fig=newton;
>>> print C.output() # same thing
Set-Cookie: sugar=wafer;
Set-Cookie: fig=newton;
>>> C = Cookie.SmartCookie()
>>> C["rocky"] = "road"
>>> C["rocky"]["path"] = "/cookie"
>>> print C.output(header="Cookie:")
Cookie: rocky=road; Path=/cookie;
>>> print C.output(attrs=[], header="Cookie:")
Cookie: rocky=road;
>>> C = Cookie.SmartCookie()
>>> C.load("chips=ahoy; vienna=finger") # load from a string (HTTP header)
>>> print C
Set-Cookie: vienna=finger;
Set-Cookie: chips=ahoy;
>>> C = Cookie.SmartCookie()
>>> C.load('keebler="E=everybody; L=\\"Loves\\"; fudge=\012;"')
>>> print C
Set-Cookie: keebler="E=everybody; L=\\"Loves\\"; fudge=\012;"
>>> C = Cookie.SmartCookie()
>>> C["oreo"] = "doublestuff"
>>> C["oreo"]["path"] = "/"
>>> print C
Set-Cookie: oreo=doublestuff; Path=/;
>>> C = Cookie.SmartCookie()
>>> C["twix"] = "none for you"
>>> C["twix"].value
'none for you'
>>> C = Cookie.SimpleCookie()
>>> C["number"] = 7 # equivalent to C["number"] = str(7)
>>> C["string"] = "seven"
>>> C["number"].value
'7'
>>> C["string"].value
'seven'
>>> print C
Set-Cookie: number=7;
Set-Cookie: string=seven;
>>> C = Cookie.SerialCookie()
>>> C["number"] = 7
>>> C["string"] = "seven"
>>> C["number"].value
7
>>> C["string"].value
'seven'
>>> print C
Set-Cookie: number="I7\012.";
Set-Cookie: string="S'seven'\012p1\012.";
>>> C = Cookie.SmartCookie()
>>> C["number"] = 7
>>> C["string"] = "seven"
>>> C["number"].value
7
>>> C["string"].value
'seven'
>>> print C
Set-Cookie: number="I7\012.";
Set-Cookie: string=seven;

```

## 11.20 xmlrpclib — XML-RPC クライアントアクセス

2.2 で追加された仕様です。

XML-RPC は XML を利用した遠隔手続き呼び出し (Remote Procedure Call) の一種で、HTTP をトランスポートとして使用します。XML-RPC ではクライアントはリモートサーバ (URI で指定されたサーバ) 上のメソッドをパラメータを指定して呼び出し、構造化されたデータを取得します。このモジュールは、XML-RPC クライアントの開発をサポートしており、Python オブジェクトに適合する転送用 XML の変換の全てを行います。

```
class ServerProxy(uri[, transport[, encoding[, verbose[, allow_none]]]])
```

ServerProxy は、リモートの XML-RPC サーバとの通信を管理するオブジェクトです。最初のパラメータは URI (Uniform Resource Indicator) で、通常はサーバの URL を指定します。2 番目のパラメータにはトランスポート・ファクトリを指定する事ができます。トランスポート・ファクトリを省略した場合、URL が https: ならモジュール内部の SafeTransport インスタンスを使用し、それ以外の場合にはモジュール内部の Transport インスタンスを使用します。オプションの 3 番目の引数はエンコード方法で、デフォルトでは UTF-8 です。オプションの 4 番目の引数はデバッグフラグです。allow\_none が真の場合、Python の定数 None は XML に翻訳されます; デフォルトの動作は None に対して TypeError を送出します。この仕様は XML-RPC 仕様でよく用いられている拡張ですが、全てのクライアントやサーバでサポートされているわけではありません; 詳細記述については <http://ontosys.com/xml-rpc/extensions.html> を参照してください。

HTTP 及び HTTPS 通信の両方で、http://user:pass@host:port/path のような HTTP 基本認証のための拡張 URL 構文をサポートしています。user:pass は base64 でエンコードして HTTP の 'Authorization' ヘッダとなり、XML-RPC メソッド呼び出し時に接続処理の一部としてリモートサーバに送信されます。リモートサーバが基本認証を要求する場合のみ、この機能を利用する必要があります。

生成されるインスタンスはリモートサーバへのプロキシオブジェクトで、RPC 呼び出しを行う為のメソッドを持ちます。リモートサーバがイントロスペクション API をサポートしている場合は、リモートサーバのサポートするメソッドを検索 (サービス検索) やサーバのメタデータの取得なども行えます。

ServerProxy インスタンスのメソッドは引数として Python の基礎型とオブジェクトを受け取り、戻り値として Python の基礎型かオブジェクトを返します。以下の型を XML に変換 (XML を通じてマーシャルする) する事ができます (特別な指定がない限り、逆変換でも同じ型として変換されます):

名前	意味
boolean	定数 True と False
整数	そのまま
浮動小数点	そのまま
文字列	そのまま
配列	変換可能な要素を含む Python シーケンス。戻り値はリスト。
構造体	Python の辞書。キーは文字列のみ。全ての値は変換可能でなくてはならない。
日付	エポックからの経過秒数。引数として指定する時は DateTime ラッパクラスのインスタンスを使用する
バイナリ	Binary ラッパクラスのインスタンス

上記の XML-RPC でサポートする全データ型を使用することができます。メソッド呼び出し時、XML-RPC サーバエラーが発生すると Fault インスタンスを送出し、HTTP/HTTPS トランスポート層でエラーが発生した場合には ProtocolError を送出します。Python 2.2 以降では組み込み型のサブクラスを作成する事ができますが、現在のところ xmlrpclib ではそのようなサブクラスのインスタンスをマーシャルすることはできません。

文字列を渡す場合、' < ' ' > ' ' & 'などのXMLで特殊な意味を持つ文字は自動的にエスケープされます。しかし、ASCII 値 0 ~ 31 の制御文字などのXMLで使用するのできない文字を使用することはできず、使用するとそのXML-RPC リクエストは well-formed なXMLとはなりません。そのような文字列を渡す必要がある場合は、後述の Binary ラッパクラスを使用してください。

Server は、上位互換性の為に ServerProxy の別名として残されています。新しいコードでは ServerProxy を使用してください。

参考資料:

XML-RPC HOWTO

(<http://xmlrpc-c.sourceforge.net/xmlrpc-howto/xmlrpc-howto.html>)

A good description of XML operation and client software in several languages. Contains pretty much everything an XML-RPC client developer needs to know.

XML-RPC-Hacks page

(<http://xmlrpc-c.sourceforge.net/hacks.php>)

Extensions for various open-source libraries to support introspection and multicall.

### 11.20.1 ServerProxy オブジェクト

ServerProxy インスタンスの各メソッドはそれぞれ XML-RPC サーバの遠隔手続き呼び出しに対応しており、メソッドが呼び出されると名前と引数をシグネチャとしてRPCを実行します(同じ名前のメソッドでも、異なる引数シグネチャによってオーバーロードされます)。RPC 実行後、変換された値を返すか、または Fault オブジェクトもしくは ProtocolError オブジェクトでエラーを通知します。

予約メンバ system から、XML イントロスペクション API の一般的なメソッドを利用する事ができます。

**system.listMethods()**

XML-RPC サーバがサポートするメソッド名 (system 以外) を格納する文字列のリストを返します。

**system.methodSignature (name)**

XML-RPC サーバで実装されているメソッドの名前を指定し、利用可能なシグネチャの配列を取得します。シグネチャは型のリストで、先頭の型は戻り値の型を示し、以降はパラメータの型を示します。

XML-RPC では複数のシグネチャ(オーバーロード)を使用することができるので、単独のシグネチャではなく、シグネチャのリストを返します。

シグネチャは、メソッドが使用する最上位のパラメータにのみ適用されます。例えばあるメソッドのパラメータが構造体の配列で戻り値が文字列の場合、シグネチャは単に"文字列, 配列" となります。パラメータが三つの整数で戻り値が文字列の場合は"文字列, 整数, 整数, 整数"となります。

メソッドにシグネチャが定義されていない場合、配列以外の値が返ります。Python では、この値は list 以外の値となります。

**system.methodHelp (name)**

XML-RPC サーバで実装されているメソッドの名前を指定し、そのメソッドを解説する文書文字列を取得します。文書文字列を取得できない場合は空文字列を返します。文書文字列には HTML マークアップが含まれます

イントロスペクション用のメソッドは、PHP・C・Microsoft .NET のサーバなどでサポートされています。UserLand Frontier の最近のバージョンでもイントロスペクションを部分的にサポートしています。Perl, Python, Java でのイントロスペクションサポートについては XML-RPC Hacks を参照してください。

## 11.20.2 Boolean オブジェクト

このクラスは全ての Python の値で初期化することができ、生成されるインスタンスは指定した値の真偽値によってのみ決まります。Boolean という名前から想像される通りに各種の Python 演算子を実装しており、`__cmp__()`、`__repr__()`、`__int__()`、`__nonzero__()` で定義される演算子を使用することができます。

以下のメソッドは、主に内部的にアンマーシャル時に使用されます:

**encode**(*out*)

出力ストリームオブジェクト *out* に、XML-RPC エンコーディングの Boolean 値を出力します。

## 11.20.3 DateTime オブジェクト

このクラスは、エポックからの秒数・時間タプル・ISO 8061 形式の時間/日付文字列の何れかで初期化することができます。

**decode**(*string*)

文字列をインスタンスの新しい時間を示す値として指定します。

**encode**(*out*)

出力ストリームオブジェクト *out* に、XML-RPC エンコーディングの DateTime 値を出力します。

また、`__cmp__`と`__repr__`で定義される演算子を使用することができます。

## 11.20.4 Binary オブジェクト

このクラスは、文字列 (NUL を含む) で初期化することができます。Binary の内容は、属性で参照します。

**data**

Binary インスタンスがカプセル化しているバイナリデータ。このデータは 8bit クリーンです。

以下のメソッドは、主に内部的にマーシャル/アンマーシャル時に使用されます:

**decode**(*string*)

指定された base64 文字列をデコードし、インスタンスのデータとします。

**encode**(*out*)

バイナリ値を base64 でエンコードし、出力ストリームオブジェクト *out* に出力します。

また、`__cmp__`で定義される演算子を使用することができます。

## 11.20.5 Fault オブジェクト

Fault オブジェクトは、XML-RPC の fault タグの内容をカプセル化しており、以下のメンバを持ちます:

**faultCode**

失敗のタイプを示す文字列。

**faultString**

失敗の診断メッセージを含む文字列。

## 11.20.6 ProtocolError オブジェクト

ProtocolError オブジェクトはトランスポート層で発生したエラー (URI で指定したサーバが見つからなかった場合に発生する 404 'not found' など) の内容を示し、以下のメンバを持ちます:

**url**

エラーの原因となった URI または URL。

**errcode**

エラーコード。

**errmsg**

エラーメッセージまたは診断文字列。

**headers**

エラーの原因となった HTTP/HTTPS リクエストを含む文字列。

### 11.20.7 補助関数

**boolean(*value*)**

Python の値を、XML-RPC の Boolean 定数 `True` または `False` に変換します。

**binary(*data*)**

Python 文字列をそのまま Binary オブジェクトに変換します。

### 11.20.8 クライアントのサンプル

```
# simple test program (from the XML-RPC specification)

# server = ServerProxy("http://localhost:8000") # local server
server = ServerProxy("http://betty.userland.com")

print server

try:
    print server.examples.getStateName(41)
except Error, v:
    print "ERROR", v
```

## 11.21 SimpleXMLRPCServer — 基本的な XML-RPC サーバー

SimpleXMLRPCServer モジュールは Python で記述された基本的な XML-RPC サーバーフレームワークを提供します。サーバーはスタンドアロンであるか、SimpleXMLRPCServer を使うか、CGIXMLRPCRequestHandler を使って CGI 環境に組み込まれるかの、いずれかです。

**class SimpleXMLRPCServer(*addr*[, *requestHandler*[, *logRequests*]])**

新しくサーバーインスタンスを作成します。引数 *requestHandler* には、リクエストハンドラーインスタンスのファクトリーを設定します。デフォルトは SimpleXMLRPCRequestHandler です。引数 *addr* と *requestHandler* は SocketServer.TCPServer のコンストラクターに引き渡されます。もし引数 *logRequests* が真 (true) であれば、(それがデフォルトですが、) リクエストはログに記録されます。偽 (false) である場合にはログは記録されません。このクラスは XML-RPC プロトコルで呼ばれる関数の登録のためのメソッドを提供します。

**class CGIXMLRPCRequestHandler()**

CGI 環境における XML-RPC リクエストハンドラーを、新たに作成します。2.3 で追加された仕様です。

**class SimpleXMLRPCRequestHandler()**

新しくリクエストハンドラーインスタンスを作成します。このリクエストハンドラーは POST リクエストを受け持ち、SimpleXMLRPCServer のコンストラクターの引数 *logRequests* に従ったログ出力を行います。

### 11.21.1 SimpleXMLRPCServer オブジェクト

SimpleXMLRPCServer クラスは SocketServer.TCPServer のサブクラスで、基本的なスタンドアロンの XML-RPC サーバーを作成する手段を提供します。

**register\_function**(*function*[, *name*])

XML-RPC リクエストに応じる関数を登録します。引数 *name* が与えられている場合はその値が、関数 *function* に関連付けられます。これが与えられない場合は *function.\_\_name\_\_* の値が用いられます。引数 *name* は通常の文字列でもユニコード文字列でも良く、Python で識別子として正しくない文字 ("." ピリオドなど) を含んでいても。

**register\_instance**(*instance*)

オブジェクトを登録し、そのオブジェクトの `register_function()` で登録されていないメソッドを公開します。もし、*instance* が `__dispatch__()` を定義していれば、`__dispatch__()` が、リクエストされたメソッド名とパラメータの組を引数として呼び出されます。そして、`__dispatch__()` の戻り値が結果としてクライアントに返されます。もし、*instance* が `__dispatch__()` を定義していなければ、リクエストされたメソッド名がそのインスタンスに定義されているメソッド名から探されます。リクエストされたメソッド名がピリオドを含む場合は、(訳注: 通常の Python でのピリオドの解釈と同様に) 階層的にオブジェクトを探索します。そして、そこで見つかったオブジェクトをリクエストから渡された引数で呼び出し、その戻り値をクライアントに返します。

**register\_introspection\_functions**()

XML-RPC のイントロスペクション関数、`system.listMethods`、`system.methodHelp`、`system.methodSignature` を登録します。2.3 で追加された仕様です。

**register\_multicall\_functions**()

XML-RPC における複数の要求を処理する関数 `system.multicall` を登録します。

以下に例を示します。

```
class MyFuncs:
    def div(self, x, y): return div(x,y)

server = SimpleXMLRPCServer(("localhost", 8000))
server.register_function(pow)
server.register_function(lambda x,y: x+y, 'add')
server.register_introspection_functions()
server.register_instance(MyFuncs())
server.serve_forever()
```

### 11.21.2 CGIXMLRPCRequestHandler

CGIXMLRPCRequestHandler クラスは、Python の CGI スクリプトに送られた XML-RPC リクエストを処理するときに使用できます

**register\_function**(*function*[, *name*])

XML-RPC リクエストに応じる関数を登録します。引数 *name* が与えられている場合はその値が、関数 *function* に関連付けられます。これが与えられない場合は *function.\_\_name\_\_* の値が用いられます。



す。引数 *name* は通常の文字列でもユニコード文字列でも良く、Python で識別子として正しくない文字 (". "ピリオドなど) を含んでもかまいません。

**register\_instance**(*instance*)

オブジェクトを登録し、そのオブジェクトの `register_function()` で登録されていないメソッドを公開します。もし、*instance* が `メソッド_dispatch()` を定義していれば、`_dispatch()` が、リクエストされたメソッド名とパラメータの組を引数として呼び出されます。そして、`_dispatch()` の戻り値が結果としてクライアントに返されます。もし、*instance* が `メソッド_dispatch()` を定義していなければ、リクエストされたメソッド名がそのインスタンスに定義されているメソッド名から探されます。リクエストされたメソッド名がピリオドを含む場合は、(訳注: 通常の Python でのピリオドの解釈と同様に) 階層的にオブジェクトを探索します。そして、そこで見つかったオブジェクトをリクエストから渡された引数で呼び出し、その戻り値をクライアントに返します。

**register\_introspection\_functions**()

XML-RPC のイントロスペクション関数、`system.listMethods`、`system.methodHelp`、`system.methodSignature` を登録します。

**register\_multicall\_functions**()

XML-RPC における複数の要求を処理する関数 `system.multicall` を登録します。

**handle\_request**( [*request\_text* = None ] )

XML-RPC リクエストを処理します。*request\_text* で渡されるのは、HTTP サーバーに提供された POST データです。何も渡されなければ標準入力からのデータが使われます。

以下に例を示します。

```
class MyFuncs:
    def div(self, x, y) : return div(x,y)

handler = CGIXMLRPCRequestHandler()
handler.register_function(pow)
handler.register_function(lambda x,y: x+y, 'add')
handler.register_introspection_functions()
handler.register_instance(MyFuncs())
handler.handle_request()
```

## 11.22 DocXMLRPCServer — セルフ-ドキュメンティング XML-RPC サーバ

2.3 で追加された仕様です。

`DocXMLRPCServer` モジュールは `SimpleXMLRPCServer` クラスを拡張し、HTTP GET リクエストに対し HTML ドキュメントを返します。サーバは `DocXMLRPCServer` を使ったスタンドアロン環境、`DocCGIXMLRPCRequestHandler` を使った CGI 環境の 2 つがあります。

**class DocXMLRPCServer**(*addr*[, *requestHandler*[, *logRequests*] ] )

当たなサーバ・インスタンスを生成します。各パラメータの内容は `SimpleXMLRPCServer.SimpleXMLRPCServer` のものと同じですが、*requestHandler* のデフォルトは `DocXMLRPCRequestHandler` になっています。

**class DocCGIXMLRPCRequestHandler**()

CGI 環境に XML-RPC リクエスト・ハンドラの新たなインスタンスを生成します。

**class DocXMLRPCRequestHandler**()

リクエスト・ハンドラの新たなインスタンスを生成します。このリクエスト・ハンドラは XML-RPC POST リクエスト、ドキュメントの GET、そして DocXMLRPCServer コンストラクタに与えられた *logRequests* パラメータ設定を優先するため、ロギングの変更をサポートします。

### 11.22.1 DocXMLRPCServer オブジェクト

DocXMLRPCServer は SimpleXMLRPCServer.SimpleXMLRPCServer の派生クラスで、セルフ-ドキュメンティングの手段と XML-RPC サーバ機能を提供します。HTTP POST リクエストは XML-RPC メソッドの呼び出しとして扱われます。HTTP GET リクエストは pydoc スタイルの HTML ドキュメント生成のリクエストとして扱われます。これはサーバが自分自身のドキュメントを Web ベースで提供可能であることを意味します。

`set_server_title(server_title)`

生成する HTML ドキュメントのタイトルをセットします。このタイトルは HTML の title 要素として使われます。

`set_server_name(server_name)`

生成する HTML ドキュメントの名前をセットします。この名前は HTML 冒頭の h1 要素に使われます。

`set_server_documentation(server_documentation)`

生成する HTML ドキュメントの本文をセットします。この本文はドキュメント中の名前の下にパラグラフとして出力されます。

### 11.22.2 DocCGIXMLRPCRequestHandler

DocCGIXMLRPCRequestHandler は SimpleXMLRPCServer.CGIXMLRPCRequestHandler の派生クラスで、セルフ-ドキュメンティングの手段と XML-RPC CGI スクリプト機能を提供します。HTTP POST リクエストは XML-RPC メソッドの呼び出しとして扱われます。HTTP GET リクエストは pydoc スタイルの HTML ドキュメント生成のリクエストとして扱われます。これはサーバが自分自身のドキュメントを Web ベースで提供可能であることを意味します。

`set_server_title(server_title)`

生成する HTML ドキュメントのタイトルをセットします。このタイトルは HTML の title 要素として使われます。

`set_server_name(server_name)`

生成する HTML ドキュメントの名前をセットします。この名前は HTML 冒頭の h1 要素に使われます。

`set_server_documentation(server_documentation)`

生成する HTML ドキュメントの本文をセットします。この本文はドキュメント中の名前の下にパラグラフとして出力されます。

## 11.23 asyncore — 非同期ソケットハンドラ

このモジュールは、非同期ソケットサービスのクライアント・サーバを開発するための基盤として使われます。

CPU が一つしかない場合、プログラムが“二つのことを同時に”実行する方法は二つしかありません。もっとも簡単で一般的なのはマルチスレッドを利用する方法ですが、これとはまったく異なるテクニックで、一つのスレッドだけでマルチスレッドと同じような効果を得られるテクニックがあります。このテク

ニックは I/O 処理が中心である場合にのみ有効で、CPU 負荷の高いプログラムでは効果が無く、この場合にはプリエンティブなスケジューリングが可能なスレッドが有効でしょう。しかし、多くの場合、ネットワークサーバでは CPU 負荷よりは IO 負荷が問題となります。

もし OS の I/O ライブラリがシステムコール `select()` をサポートしている場合（ほとんどの場合はサポートされている）、I/O 処理は“バックグラウンド”で実行し、その間に他の処理を実行すれば、複数の通信チャンネルを同時にこなすことができます。一見、この戦略は奇妙で複雑に思えるかもしれませんが、いろいろな面でマルチスレッドよりも理解しやすく、制御も容易です。`asyncore` は多くの複雑な問題を解決済みなもので、洗練され、パフォーマンスにも優れたネットワークサーバとクライアントを簡単に開発することができます。とくに、`asynchat` のような、対話型のアプリケーションやプロトコルには非常に有効でしょう。

基本的には、この二つのモジュールを使う場合は一つ以上のネットワークチャンネルを `asyncore.dispatcher` クラス、または `asynchat.async_chat` のインスタンスとして作成します。作成されたチャンネルはグローバルマップに登録され、`loop()` 関数で参照されます。`loop()` には、専用のマップを渡す事も可能です。

チャンネルを生成後、`loop()` を呼び出すとチャンネル処理が開始し、最後のチャンネル（非同期処理中にマップに追加されたチャンネルを含む）が閉じるまで続きます。

```
loop([timeout[, use_poll[, map]]])
```

ポーリングループを開始し、全てのオープン済みチャンネルがクローズされた場合のみ終了します。全ての引数はオプションです。引数 `timeout` は `select()` または `poll()` の引数 `timeout` として渡され、秒単位で指定します。デフォルト値は 30 秒です。引数 `use_poll` が真のとき、`select()` ではなく `poll()` が使われます。デフォルト値は `False` です。引数 `map` には、監視するチャンネルをアイテムとして格納した辞書を指定します。`map` が省略された場合、グローバルなマップが使用される。グローバルなマップは、チャンネルクラスの `__init__()` メソッドが呼び出されたときに自動的に更新されます。—この仕組みを利用するのであれば、チャンネルクラスの `__init__()` はオーバーライドするのではなく、拡張しなければなりません。

```
class dispatcher()
```

`dispatcher` クラスは、低レベルソケットオブジェクトの薄いラッパーです。便宜上、非同期ループから呼び出されるイベント処理メソッドを追加していますが、これ以外の点では、non-blocking なソケットと同様です。

`dispatcher` クラスには二つのクラス属性があり、パフォーマンス向上やメモリの削減のために更新する事ができます。

```
ac_in_buffer_size
```

非同期入力バッファのサイズ (デフォルト 4096)

```
ac_out_buffer_size
```

非同期出力バッファのサイズ (デフォルト 4096)

非同期ループ内で低レベルイベントが発生した場合、発生タイミングやコネクションの状態から特定の高レベルイベントへと置き換えることができます。例えばソケットを他のホストに接続する場合、最初の書き込み可能イベントが発生すれば接続が完了した事が分かります（この時点で、ソケットへの書き込みは成功すると考えられる）。このように判定できる高レベルイベントを以下に示します：

イベント	解説
<code>handle_connect()</code>	最初に <code>write</code> イベントが発生した時
<code>handle_close()</code>	読み込み可能なデータなしで <code>read</code> イベントが発生した時
<code>handle_accept()</code>	<code>listen</code> 中のソケットで <code>read</code> イベントが発生した時

非同期処理中、マップに登録されたチャンネルの `readable()` メソッドと `writable()` メソッドが呼び出され、`select()` か `poll()` で `read/write` イベントを検出するリストに登録するか否かを判

定します。

このようにして、チャンネルでは低レベルなソケットイベントの種類より多くの種類のイベントを検出する事ができます。以下にあげるイベントは、サブクラスでオーバーライドすることが可能です：

**handle\_read()**

非同期ループで、チャンネルのソケットの `read()` メソッドの呼び出しが成功した時に呼び出されます。

**handle\_write()**

非同期ループで、書き込み可能ソケットが実際に書き込み可能になった時に呼び出される。このメソッドは、パフォーマンスの向上のためバッファリングを行う場合などに利用できます。例：

```
def handle_write(self):
    sent = self.send(self.buffer)
    self.buffer = self.buffer[sent:]
```

**handle\_expt()**

out of band (OOB) データが検出された時に呼び出されます。OOB はあまりサポートされておらず、また滅多に使われないので、`handle_expt()` が呼び出されることはほとんどありません。

**handle\_connect()**

ソケットの接続が確立した時に呼び出されます。“welcome” バナーの送信、プロトコルネゴシエーションの初期化などを行います。

**handle\_close()**

ソケットが閉じた時に呼び出されます。

**handle\_error()**

捕捉されない例外が発生した時に呼び出されます。デフォルトでは、短縮したトレースバック情報が出力されます。

**handle\_accept()**

listen 中のチャンネルがリモートホストからの `connect()` で接続され、接続が確立した時に呼び出されます。

**readable()**

非同期ループ中に呼び出され、`read` イベントの監視リストに加えるか否かを決定します。デフォルトのメソッドでは `True` を返し、`read` イベントの発生を監視します。

**writable()**

非同期ループ中に呼び出され、`write` イベントの監視リストに加えるか否かを決定します。デフォルトのメソッドでは `True` を返し、`write` イベントの発生を監視します。

さらに、チャンネルにはソケットのメソッドとほぼ同じメソッドがあり、チャンネルはソケットのメソッドの多くを委譲・拡張しており、ソケットとほぼ同じメソッドを持っています。

**create\_socket(*family, type*)**

引数も含め、通常のソケット生成と同じ。socket モジュールを参照のこと。

**connect(*address*)**

通常のソケットオブジェクトと同様、*address* には一番目の値が接続先ホスト、2 番目の値がポート番号であるタプルを指定します。

**send(*data*)**

リモート側の端点に *data* を送出します。

**recv(*buffer\_size*)**

リモート側の端点より、最大 *buffer\_size* バイトのデータを読み込みます。長さ 0 の文字列が返ってきた場合、チャンネルはリモートから切断された事を示します。

**listen(backlog)**

ソケットへの接続を待つ。引数 *backlog* は、キューイングできるコネクションの最大数を指定します (1 以上)。最大数はシステムに依存でします (通常は 5)

**bind(address)**

ソケットを *address* にバインドします。ソケットはバインド済みであってはなりません。(*address* の形式は、アドレスファミリに依存します。socket モジュールを参照のこと。)

**accept()**

接続を受け入れます。ソケットはアドレスにバインド済みであり、listen() で接続待ち状態であればなりません。戻り値は (*conn*, *address*) のペアで、*conn* はデータの送受信を行うソケットオブジェクト、*address* は接続先ソケットがバインドされているアドレスです。

**close()**

ソケットをクローズします。以降の全ての操作は失敗します。リモート端点では、キューに溜まったデータ以外、これ以降のデータ受信は行えません。ソケットはガベージコレクト時に自動的にクローズされます。

### 11.23.1 `asyncore` の例：簡単な HTTP クライアント

基本的なサンプルとして、以下に非常に単純な HTTP クライアントを示します。この HTTP クライアントは `dispatcher` クラスでソケットを利用しています。

```
class http_client(asyncore.dispatcher):
    def __init__(self, host,path):
        asyncore.dispatcher.__init__(self)
        self.path = path
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.connect( (host, 80) )
        self.buffer = 'GET %s HTTP/1.0\r\n\r\n' % self.path

    def handle_connect(self):
        pass

    def handle_read(self):
        data = self.recv(8192)
        print data

    def writable(self):
        return (len(self.buffer) > 0)

    def handle_write(self):
        sent = self.send(self.buffer)
        self.buffer = self.buffer[sent:]
```

## 11.24 `asynchat` — 非同期ソケット コマンド/レスポンス ハンドラ

`asynchat` を使うと、`asyncore` を基盤とした非同期なサーバ・クライアントをより簡単に開発する事ができます。`asynchat` では、プロトコルの要素が任意の文字列で終了するか、または可変長の文字列であるようなプロトコルを容易に制御できるようになっています。`asynchat` は、抽象クラス `async_chat` を定義しており、`async_chat` を継承して `collect_incoming_data()` メソッドと `found_terminator()` メソッドを実装すれば使うことができます。`async_chat` と `asyncore` は同じ非同期ループを使用しており、`asyncore.dispatcher` も `asynchat.async_chat` も同じチャネルマップに登録する事がで



きます。通常、`asyncore.dispatcher` はサーバチャネルとして使用し、リクエストの受け付け時に `asyncchat.async_chat` オブジェクトを生成します。

`class async_chat()`

このクラスは、`asyncore.dispatcher` から継承した抽象クラスです。使用する際には `async_chat` のサブクラスを作成し、`collect_incoming_data()` と `found_terminator()` を定義しなければなりません。`asyncore.dispatcher` のメソッドを使用する事もできますが、メッセージ/レスポンス処理を中心に行う場合には使えないメソッドもあります。

`asyncore.dispatcher` と同様に、`async_chat` も `select()` 呼出し後のソケットの状態からイベントを生成します。ポーリングループ開始後、イベント処理フレームワークが自動的に `async_chat` のメソッドを呼び出しますので、プログラマが処理を記述する必要はありません。

`asyncore.dispatcher` と違い、`async_chat` ではプロデューサの first-in-first-out キュー (fifo) を作成する事ができます。プロデューサは `more()` メソッドを必ず持ち、このメソッドでチャネル上に送出するデータを返します。プロデューサが枯渇状態 (*i.e.* これ以上のデータを持たない状態) にある場合、`more()` は空文字列を返します。この時、`async_chat` は枯渇状態にあるプロデューサを fifo から除去し、次のプロデューサが存在すればそのプロデューサを使用します。fifo にプロデューサが存在しない場合、`handle_write()` は何もしません。リモート端点からの入力終了や重要な中断点を検出する場合は、`set_terminator()` に記述します。

`async_chat` のサブクラスでは、入力メソッド `collect_incoming_data()` と `found_terminator()` を定義し、チャネルが非同期に受信するデータを処理します。以下にメソッドの解説を示します。

`close_when_done()`

プロデューサ fifo のトップに `None` をプッシュします。このプロデューサがポップされると、チャネルがクローズします。

`collect_incoming_data(data)`

チャネルが受信した不定長のデータを `data` に指定して呼び出されます。このメソッドは必ずオーバーライドする必要があるため、デフォルトの実装では、`NotImplementedError` 例外を送出します。

`discard_buffers()`

非常用のメソッドで、全ての入出力バッファとプロデューサ fifo を廃棄します。

`found_terminator()`

入力データストリームが、`set_terminator` で指定した終了条件と一致した場合に呼び出されます。このメソッドは必ずオーバーライドする必要があるため、デフォルトの実装では、`NotImplementedError` 例外を送出します。入力データを参照する必要がある場合でも引数としては与えられないため、入力バッファをインスタンス属性として参照しなければなりません。

`get_terminator()`

現在のチャネルの終了条件を返します。

`handle_close()`

チャネル閉じた時に呼び出されます。デフォルトの実装では単にチャネルのソケットをクローズします。

`handle_read()`

チャネルの非同期ループで `read` イベントが発生した時に呼び出され、デフォルトの実装では、`set_terminator()` で設定された終了条件をチェックします。終了条件として、特定の文字列か受信文字数を指定する事ができます。終了条件が満たされている場合、`handle_read` は終了条件が成立するよりも前のデータを引数として `collect_incoming_data()` を呼び出し、その後 `found_terminator()` を呼び出します。



**handle\_write()**

アプリケーションがデータを出力する時に呼び出され、デフォルトの実装では `initiate_send()` を呼び出します。 `initiate_send()` では `refill_buffer()` を呼び出し、チャンネルのプロデューサ `fifo` からデータを取得します。

**push(data)**

`data` を持つ `simple_producer`(後述) オブジェクトを生成し、チャンネルの `producer_fifo` にプッシュして転送します。データをチャンネルに書き出すために必要なのはこれだけですが、データの暗号化やチャンク化などを行う場合には独自のプロデューサを使用する事もできます。

**push\_with\_producer(producer)**

指定したプロデューサオブジェクトをチャンネルの `fifo` に追加します。これより前に `push` されたプロデューサが全て枯渇した後、チャンネルはこのプロデューサから `more()` メソッドでデータを取得し、リモート端点に送信します。

**readable()**

`select()` ループでこのチャンネルの読み込み可能チェックを行う場合には、`True` を返します。

**refill\_buffer()**

`fifo` の先頭にあるプロデューサの `more()` メソッドを呼び出し、出力バッファを補充します。先頭のプロデューサが枯渇状態の場合には `fifo` からポップされ、その次のプロデューサがアクティブになります。アクティブなプロデューサが `None` になると、チャンネルはクローズされます。

**set\_terminator(term)**

チャンネルで検出する終了条件を設定します。 `term` は入力プロトコルデータの処理方式によって以下の3つの型の何れかを指定します。

term	説明
<i>string</i>	入力ストリーム中で <i>string</i> が検出された時、 <code>found_terminator()</code> を呼び出します。
<i>integer</i>	指定された文字数が読み込まれた時、 <code>found_terminator()</code> を呼び出します。
<i>None</i>	永久にデータを読み込みます。

終了条件が成立しても、その後に続くデータは、 `found_terminator()` の呼出し後に再びチャンネルを読み込めば取得する事ができます。

**writable()**

Should return `True` as long as items remain on the producer fifo, or the channel is connected and the channel's output buffer is non-empty.

プロデューサ `fifo` が空ではないか、チャンネルが接続中で出力バッファが空でない場合、`True` を返します。

## 11.24.1 asynchat - 補助クラスと関数

**class simple\_producer(data[, buffer\_size=512])**

`simple_producer` には、一連のデータと、オプションとしてバッファサイズを指定する事ができます。 `more()` が呼び出されると、その都度 `buffer_size` 以下の長さのデータを返します。

**more()**

プロデューサから取得した次のデータか、空文字列を返します。

**class fifo([list=None])**

各チャンネルは、アプリケーションからプッシュされ、まだチャンネルに書き出されていないデータを `fifo` に保管しています。 `fifo` では、必要なデータとプロデューサのリストを管理しています。引数 `list` には、プロデューサがチャンネルに出力するデータを指定する事ができます。

`is_empty()`

`fifo` が空のとき `True` を返します。

`first()`

`fifo` に `push()` されたアイテムのうち、最も古いアイテムを返します。

`push(data)`

データ (文字列またはプロデューサオブジェクト) をプロデューサ `fifo` に追加します。

`pop()`

`fifo` が空でなければ、`(True, first())` を返し、ポップされたアイテムを削除します。`fifo` が空であれば `(False, None)` を返します。

`asynchat` は、ネットワークとテキスト分析操作で使えるユーティリティ関数を提供しています。

`find_prefix_at_end(haystack, needle)`

文字列 `haystack` の末尾が `needle` の先頭と一致するかを調べ、一致した文字数を返します。

ex) `find_prefix_at_end("abc12`

`r", "`

`r")` は 1 を返します。

### 11.24.2 asynchat 使用例

以下のサンプルは、`async_chat` で HTTP リクエストを読み込む処理の一部です。Web サーバは、クライアントからの接続毎に `http_request_handler` オブジェクトを作成します。最初はチャンネルの終了条件に空行を指定して HTTP ヘッダの末尾までを検出し、その後ヘッダ読み込み済みを示すフラグを立てています。

ヘッダ読み込んだ後、リクエストの種類が POST であればデータが入力ストリームに流れるため、`Content-Length`:ヘッダの値を数値として終了条件に指定し、適切な長さのデータをチャンネルから読み込みます。

必要な入力データを全て入手したら、チャンネルの終了条件に `None` を指定して残りのデータを無視するようにしています。この後、`handle_request()` が呼び出されます。

```

class http_request_handler(asyncchat.async_chat):

    def __init__(self, conn, addr, sessions, log):
        asyncchat.async_chat.__init__(self, conn=conn)
        self.addr = addr
        self.sessions = sessions
        self.ibuffer = []
        self.obuffer = ""
        self.set_terminator("\r\n\r\n")
        self.reading_headers = True
        self.handling = False
        self.cgi_data = None
        self.log = log

    def collect_incoming_data(self, data):
        """Buffer the data"""
        self.ibuffer.append(data)

    def found_terminator(self):
        if self.reading_headers:
            self.reading_headers = False
            self.parse_headers("".join(self.ibuffer))
            self.ibuffer = []
            if self.op.upper() == "POST":
                clen = self.headers.getheader("content-length")
                self.set_terminator(int(clen))
            else:
                self.handling = True
                self.set_terminator(None)
                self.handle_request()
        elif not self.handling:
            self.set_terminator(None) # browsers sometimes over-send
            self.cgi_data = parse(self.headers, "".join(self.ibuffer))
            self.handling = True
            self.ibuffer = []
            self.handle_request()

```



# インターネット上のデータの操作

この章ではインターネット上で一般的に利用されているデータ形式の操作をサポートするモジュール群について記述します。

<code>formatter</code>	汎用の出力書式化機構およびデバイスインタフェース。
<code>email.Iterators</code>	メッセージオブジェクトツリーをたどる。
<code>mailcap</code>	mailcap ファイルの操作。
<code>mailbox</code>	様々なメールボックス形式の読み出し。
<code>mhlib</code>	Python から MH のメールボックスを操作します。
<code>mimetools</code>	MIME-スタイルのメッセージ本体を解析するためのツール。
<code>mimetypes</code>	ファイル名拡張子の MIME 型へのマッピング。
<code>MimeWriter</code>	汎用 MIME ファイルライター。
<code>mimify</code>	電子メールメッセージの MIME 化および非 MIME 化。
<code>multifile</code>	MIME データのような、個別の部分を含んだファイル群に対する読み出しのサポート。
<code>rfc822</code>	RFC 2822 形式のメールメッセージを解釈します。
<code>base64</code>	MIME の base64 形式のエンコード、デコードを行う。
<code>binascii</code>	バイナリと各種 ASCII コード化バイナリ表現との間の変換を行うツール群。
<code>binhex</code>	binhex4 形式ファイルのエンコードおよびデコード。
<code>quopri</code>	MIME quoted-printable 形式ファイルのエンコードおよびデコード。
<code>uu</code>	uuencode 形式のエンコードとデコードを行う。
<code>xdrlib</code>	外部データ表現 (XDR, External Data Representation) データのエンコードおよびデコード。
<code>netrc</code>	‘.netrc’ ファイル群の読み出し。
<code>robotparser</code>	‘robots.txt’ ファイルを読み出し、他の URL に対する取得可能性の質問に答えるクラス。
<code>csv</code>	デリミタで区切られた形式のファイルに対するテーブル状データ読み書き。

## 12.1 formatter — 汎用の出力書式化機構

このモジュールでは、二つのインタフェース定義を提供しており、それらの各インタフェースについて複数の実装を提供しています。`formatter` インタフェースは `htmllib` モジュールの `HTMLParser` クラスで使われており、`writer` インタフェースは `formatter` インタフェースを使う上で必要です。

`formatter` オブジェクトはある抽象化された書式イベントの流れを `writer` オブジェクト上の特定の出力イベントに変換します。`formatter` はいくつかのスタック構造を管理することで、`writer` オブジェクトの様々な属性を変更したり復元したりできるようにしています; このため、`writer` は相対的な変更や“元に戻す”操作を処理できなくてもかまいません。`writer` の特定のプロパティのうち、`formatter` オブジェクトを介して制御できるのは、水平方向の字揃え、フォント、そして左マージンの字下げです。任意の、非排他的なスタイル設定を `writer` に提供するためのメカニズムも提供されています。さらに、段落分割のように、可逆でない書式化イベントの機能を提供するインタフェースもあります。

`writer` オブジェクトはデバイスインタフェースをカプセル化します。ファイル形式のような抽象デバイス

も物理デバイス同様にサポートされています。ここで提供されている実装内容はすべて抽象デバイス上で動作します。デバイスインタフェースは `formatter` オブジェクトが管理しているプロパティを設定し、データを出力端に書き込めるようにします。

### 12.1.1 formatter インタフェース

`formatter` を作成するためのインタフェースは、インスタンス化しようとする個々の `formatter` クラスに依存します。以下で解説するのは、インスタンス化された全ての `formatter` がサポートしなければならないインタフェースです。

モジュールレベルではデータ要素を一つ定義しています:

**AS\_IS**

後に述べる `push_font()` メソッドでフォント指定をする時に使える値です。また、その他の `push_property()` メソッドの新しい値として使うことができます。

`AS_IS` の値をスタックに置くと、どのプロパティが変更されたかの追跡を行わずに、対応する `pop_property()` メソッドが呼び出されるようになります。

`formatter` インスタンスオブジェクトには以下の属性が定義されています:

**writer**

`formatter` とやり取りを行う `writer` インスタンスです。

**end\_paragraph(*blanklines*)**

開かれている段落があれば閉じ、次の段落との間に少なくとも *blanklines* が挿入されるようにします。

**add\_line\_break()**

強制改行挿入します。既に強制改行がある場合は挿入しません。論理的な段落は中断しません。

**add\_hor\_rule(*\*args, \*\*kw*)**

出力に水平罫線を挿入します。現在の段落に何らかのデータがある場合、強制改行が挿入されますが、論理的な段落は中断しません。引数とキーワードは `writer` の `send_line_break()` メソッドに渡されます。

**add\_flowng\_data(*data*)**

空白を折りたたんで書式化しなければならないデータを提供します。空白の折りたたみでは、直前や直後の `add_flowng_data` 呼び出しに入っている空白も考慮されます。このメソッドに渡されたデータは出力デバイスで行末の折り返し (word-wrap) されるものと想定されています。出力デバイスでの要求やフォント情報に応じて、`writer` オブジェクトでも何らかの行末折り返しが行われなければならないので注意してください。

**add\_literal\_data(*data*)**

変更を加えずに `writer` に渡さなければならないデータを提供します。改行およびタブを含む空白を *data* の値にしても問題ありません。

**add\_label\_data(*format, counter*)**

現在の左マージン位置の左側に配置されるラベルを挿入します。このラベルは箇条書き、数字つき箇条書きの書式を構築する際に使われます。*format* の値が文字列の場合、整数の値 *counter* の書式指定として解釈されます。

*format* の値が文字列の場合、整数の値をとる *counter* の書式化指定として解釈されます。書式化された文字列はラベルの値になります; *format* が文字列でない場合、ラベルの値として直接使われます。ラベルの値は `writer` の `send_label_data()` メソッドの唯一の引数として渡されます。非文字列のラベル値をどう解釈するかは関連付けられた `writer` に依存します。

書式化指定は文字列からなり、*counter* の値と合わせてラベルの値を算出するために使われます。書



式文字列の各文字はラベル値にコピーされます。このときいくつかの文字は counter 値を変換を指すものとして認識されます。特に、文字 ‘1’ はアラビア数字の counter 値を表し、‘A’ と ‘a’ はそれぞれ大文字および小文字のアルファベットによる counter 値を表し、‘I’ と ‘i’ はそれぞれ大文字および小文字のローマ数字による counter 値を表します。アルファベットおよびローマ数字への変換の際には、counter の値はゼロ以上である必要がありますので注意してください。

**flush\_softspace()**

以前の `add_flowring_data()` 呼び出しでバッファされている出力待ちの空白を、関連付けられている writer オブジェクトに送信します。このメソッドは writer オブジェクトに対するあらゆる直接操作の前に呼び出さなければなりません。

**push\_alignment(*align*)**

新たな字揃え (*alignment*) 設定を字揃えスタックの上にプッシュします。変更を行いたくない場合には `AS_IS` にすることができます。字揃え設定値が以前の設定から変更された場合、writer の `new_alignment()` メソッドが *align* の値と共に呼び出されます。

**pop\_alignment()**

以前の字揃え設定を復元します。

**push\_font((*size*, *italic*, *bold*, *teletype*))**

writer オブジェクトのフォントプロパティのうち、一部または全てを変更します。`AS_IS` に設定されていないプロパティは引数で渡された値に設定され、その他の値は現在の設定を維持します。writer の `new_font()` メソッドは完全に設定解決されたフォント指定で呼び出されます。

**pop\_font()**

以前のフォント設定を復元します。

**push\_margin(*margin*)**

左マージンのインデント数を一つ増やし、論理タグ *margin* を新たなインデントに関連付けます。マージンレベルの初期値は 0 です。変更された論理タグの値は真値とならなければなりません; `AS_IS` 以外の偽の値はマージンの変更としては不適切です。

**pop\_margin()**

以前のマージン設定を復元します。

**push\_style(\**styles*)**

任意のスタイル指定をスタックにプッシュします。全てのスタイルはスタイルスタックに順番にプッシュされます。`AS_IS` 値を含み、スタック全体を表すタプルは writer の `new_styles()` メソッドに渡されます。

**pop\_style([*n* = 1])**

`push_style()` に渡された最新 *n* 個のスタイル指定をポップします。`AS_IS` 値を含み、変更されたスタックを表すタプルは writer の `new_styles()` メソッドに渡されます。

**set\_spacing(*spacing*)**

writer の割り付けスタイル (*spacing style*) を設定します。

**assert\_line\_data([*flag* = 1])**

現在の段落にデータが予期せず追加されたことを formatter に知らせます。このメソッドは writer を直接操作した際に使わなければなりません。writer 操作の結果、出力の末尾が強制改行となった場合、オプションの *flag* 引数を偽に設定することができます。

### 12.1.2 formatter 実装

このモジュールでは、formatter オブジェクトに関して二つの実装を提供しています。ほとんどのアプリケーションではこれらのクラスを変更したりサブクラス化することなく使うことができます。

`class NullFormatter([writer])`

何も行わない formatter です。writer を省略すると、NullWriter インスタンスが生成されます。NullFormatter インスタンスは、writer のメソッドを全く呼び出しません。writer へのインタフェースを実装する場合にはこのクラスのインタフェースを継承する必要がありますが、実装を継承する必要は全くありません。

`class AbstractFormatter(writer)`

標準の formatter です。この formatter 実装は広範な writer で適用できることが実証されており、ほとんどの状況で直接使うことができます。高機能の WWW ブラウザを実装するために使われたこともあります。

### 12.1.3 writer インタフェース

writer を作成するためのインタフェースは、インスタンス化しようとする個々の writer クラスに依存します。以下で解説するのは、インスタンス化された全ての writer がサポートしなければならないインタフェースです。ほとんどのアプリケーションでは AbstractFormatter クラスを formatter として使うことができますが、通常 writer はアプリケーション側で与えなければならないので注意してください。

`flush()`

バッファに蓄積されている出力データやデバイス制御イベントをフラッシュします。

`new_alignment(align)`

字揃えのスタイルを設定します。align の値は任意のオブジェクトを取りえますが、慣習的な値は文字列または None で、None は writer の“好む”字揃えを使うことを表します。慣習的な align の値は 'left'、'center'、'right'、および 'justify' です。

`new_font(font)`

フォントスタイルを設定します。font は、デバイスの標準のフォントが使われることを示す None か、(size, italic, bold, teletype) の形式をとるタプルになります。size はフォントサイズを示す文字列になります; 特定の文字列やその解釈はアプリケーション側で定義します。italic、bold、および teletype といった値はブール値で、それらの属性を使うかどうかを指定します。

`new_margin(margin, level)`

マージンレベルを整数値 level に設定し、論理タグ (logical tag) を margin に設定します。論理タグの解釈は writer の判断に任されます; 論理タグの値に対する唯一の制限は level が非ゼロの値の際に偽であってはならないということです。

`new_spacing(spacing)`

割り付けスタイル (spacing style) を spacing に設定します。Set the spacing style to spacing.

`new_styles(styles)`

追加のスタイルを設定します。styles の値は任意の値からなるタプルです; AS\_IS 値は無視されます。styles タプルはアプリケーションや writer の実装上の都合により、集合としても、スタックとしても解釈され得ます。

`send_line_break()`

現在の行を改行します。

`send_paragraph(blankline)`

少なくとも blankline 空行分の間隔か、空行そのもので段落を分割します。blankline の値は整数になります。writer の実装では、改行を行う必要がある場合、このメソッドの呼び出しに先立って send\_line\_break() の呼び出しを受ける必要があります; このメソッドには段落の最後の行を閉じる機能は含まれておらず、段落間に垂直スペースを空ける役割しかありません。

`send_hor_rule(*args, **kw)`

水平罫線出力デバイスに表示します。このメソッドへの引数は全てアプリケーションおよび writer 特有のもので、注意して解釈する必要があります。このメソッドの実装では、すでに改行が `send_line_break()` によってなされているものと仮定しています。

`send_flowring_data(data)`

行端が折り返され、必要に応じて再割り付け解析を行った (re-flowed) 文字データを出力します。このメソッドを連続して呼び出す上では、writer は複数の空白文字は単一のスペース文字に縮約されていると仮定することがあります。

`send_literal_data(data)`

すでに表示用に書式化された文字データを出力します。これは通常、改行文字で表された改行を保存し、新たに改行を持ち込まないことを意味します。`send_formatted_data()` インタフェースと違って、データには改行やタブ文字が埋め込まれていてもかまいません。

`send_label_data(data)`

可能ならば、*data* を現在の左マージンの左側に設定します。*data* の値には制限がありません; 文字列でない値の扱い方はアプリケーションや writer に完全に依存します。このメソッドは行の先頭でのみ呼び出されます。

#### 12.1.4 writer 実装

このモジュールでは、3 種類の writer オブジェクトインタフェース実装を提供しています。ほとんどのアプリケーションでは、`NullWriter` から新しい writer クラスを導出する必要があるでしょう。

`class NullWriter()`

インタフェース定義だけを提供する writer クラスです; どのメソッドも何ら処理を行いません。このクラスは、メソッド実装をまったく継承する必要のない writer 全ての基底クラスになります。

`class AbstractWriter()`

この writer は formatter をデバッグするのに利用できますが、それ以外に利用できるほどのものではありません。各メソッドを呼び出すと、メソッド名と引数を標準出力に印字して呼び出されたことを示します。

`class DumbWriter([file[, maxcol = 72]])`

単純な writer クラスで *file* に渡されたファイルオブジェクトか *file* が省略された場合には標準出力に出力を書き込みます。出力は *maxcol* で指定されたカラム数で単純な行端折り返しが行われます。このクラスは連続した段落を再割り付けするのに適しています。

## 12.2 email — 電子メールと MIME 処理のためのパッケージ

2.2 で追加された仕様です。

`email` パッケージは電子メールのメッセージを管理するライブラリです。これには MIME やそれ以外の RFC 2822 ベースのメッセージ文書もふくまれます。このパッケージはいくつかの古い標準パッケージ、`rfc822`、`mimetools`、`multifile` などにふくまれていた機能のほとんどを持ち、くわえて標準ではなかった `mimecntl` などの機能もふくんでいます。このパッケージは、とくに電子メールのメッセージを SMTP (RFC 2821) サーバに送信するために作られているというわけではありません。それは `smtplib` モジュールの機能です。`email` パッケージは RFC 2822 に加えて、RFC 2045-RFC 2047 および RFC 2231 など MIME 関連の RFC をサポートしており、できるかぎり RFC に準拠することをめざしています。

`email` パッケージの一番の特徴は、電子メールの内部表現であるオブジェクトモデルと、電子メールメッセージの解析および生成とを分離していることです。`email` パッケージを使うアプリケーションは基本的にはオブジェクトを処理することができます。メッセージに子オブジェクトを追加したり、メッセー

ジから子オブジェクトを削除したり、内容を完全に並べかえたり、といったことができます。フラットなテキスト文書からオブジェクトモデルへの変換、またそこからフラットな文書へと戻す変換はそれぞれ別々の解析器 (パーサ) と生成器 (ジェネレータ) が担当しています。また、一般的な MIME オブジェクトタイプのいくつかについては手軽なサブクラスが存在しており、メッセージフィールド値を抽出したり解析したり、RFC 準拠の日付を生成したりなどのよくおこわれるタスクについてはいくつかの雑用ユーティリティもついています。

以下の節では email パッケージの機能を説明します。説明の順序は多くのアプリケーションで一般的な使用順序にもとづいています。まず、電子メールメッセージをファイルあるいはその他のソースからフラットなテキスト文書として読み込み、つぎにそのテキストを解析して電子メールのオブジェクト構造を作成し、その構造を操作して、最後にフラットなテキストに戻す、という順序になっています。

このオブジェクト構造は、まったくのゼロから作りだしたものであってもいっこうにかまいません。この場合も上と似たような作業順序になるでしょう。

またここには email パッケージが提供するすべてのクラスおよびモジュールに関する説明と、email パッケージを使っていくうえで遭遇するかもしれない例外クラス、いくつかの補助ユーティリティ、そして少々サンプルも含まれています。古い `mimelib` や前バージョンの email パッケージのユーザのために、現行バージョンとの違いと移植についての節も設けてあります。

参考資料:

`smtpplib` モジュール (11.12 節):

SMTP プロトコル クライアント

### 12.2.1 電子メールメッセージの表現

`Message` クラスは、email パッケージの中心となるクラスです。これは email オブジェクトモデルの基底クラスになっています。`Message` はヘッダフィールドを検索したりメッセージ本体にアクセスするための核となる機能を提供します。

概念的には、`Message` オブジェクトにはヘッダとペイロードが格納されています。ヘッダは、RFC 2822 形式のフィールド名およびフィールド値がコロンで区切られたものです。コロンはフィールド名またはフィールド値のどちらにも含まれません。

ヘッダは大文字小文字を区別した形式で保存されますが、ヘッダ名が一致するかどうかの検査は大文字小文字を区別せずにおこなうことができます。`Unix-From` ヘッダまたは `From_` ヘッダとして知られるエンベロープヘッダがひとつ存在することもあります。ペイロードは、単純なメッセージオブジェクトの場合は単なる文字列ですが、MIME コンテナ文書 (`multipart/*` または `message/rfc822` など) の場合は `Message` オブジェクトのリストになっています。

`Message` オブジェクトは、メッセージヘッダにアクセスするためのマップ (辞書) 形式のインタフェースと、ヘッダおよびペイロードの両方にアクセスするための明示的なインタフェースを提供します。これにはメッセージオブジェクトツリーからフラットなテキスト文書を生成したり、一般的に使われるヘッダのパラメータにアクセスしたり、またオブジェクトツリーを再帰的にたどったりするための便利なメソッドを含みます。

`Message` クラスのメソッドは以下のとおりです:

```
class Message ( )
```

    コンストラクタは引数を取りません。

```
as_string( [unixfrom] )
```

    メッセージ全体をフラットな文字列として返します。オプション `unixfrom` が `True` の場合、返される文字列にはエンベロープヘッダも含まれます。`unixfrom` のデフォルトは `False` です。

    このメソッドは手軽に利用する事ができますが、必ずしも期待通りにメッセージをフォーマットする

とは限りません。以下の例のように Generator のインスタンスを生成して `flatten()` メソッドを直接呼び出せばより柔軟な処理を行う事ができます。

```
from cStringIO import StringIO
from email.Generator import Generator
fp = StringIO()
g = Generator(fp, mangle_from_=False, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

`__str__()`

`as_string(unixfrom=True)` と同じです。

`is_multipart()`

メッセージのペイロードが子 Message オブジェクトからなるリストであれば True を返し、そうでなければ False を返します。 `is_multipart()` が False を返した場合は、ペイロードは文字列オブジェクトである必要があります。

`set_unixfrom(unixfrom)`

メッセージのエンベロープヘッダを `unixfrom` に設定します。これは文字列である必要があります。

`get_unixfrom()`

メッセージのエンベロープヘッダを返します。エンベロープヘッダが設定されていない場合は None が返されます。

`attach(payload)`

与えられた `payload` を現在のペイロードに追加します。この時点でのペイロードは None か、あるいは Message オブジェクトのリストである必要があります。このメソッドの実行後、ペイロードは必ず Message オブジェクトのリストになります。ペイロードにスカラーオブジェクト (文字列など) を格納したい場合は、かわりに `set_payload()` を使ってください。

`get_payload([i[, decode]])`

現在のペイロードへの参照を返します。これは `is_multipart()` が True の場合 Message オブジェクトのリストになり、`is_multipart()` が False の場合は文字列になります。ペイロードがリストの場合、リストを変更することはそのメッセージのペイロードを変更することになります。

オプション引数の `i` がある場合、`is_multipart()` が True ならば `get_payload()` はペイロード中で 0 から数えて `i` 番目の要素を返します。`i` が 0 より小さい場合、あるいはペイロードの個数以上の場合、`IndexError` が発生します。ペイロードが文字列 (つまり `is_multipart()` が False) にもかかわらず `i` が与えられたときは `TypeError` が発生します。

オプションの `decode` はそのペイロードが Content-Transfer-Encoding: ヘッダに従ってデコードされるべきかどうかを指示するフラグです。この値が True でメッセージが multipart でない場合、ペイロードはこのヘッダの値が 'quoted-printable' または 'base64' のときにかぎりデコードされます。これ以外のエンコーディングが使われている場合、Content-Transfer-Encoding: ヘッダがない場合、あるいは曖昧な base64 データが含まれる場合は、ペイロードはそのまま (デコードされずに) 返されます。もしメッセージが multipart で `decode` フラグが True の場合は None が返されます。`decode` のデフォルト値は False です。

`set_payload(payload[, charset])`

メッセージ全体のオブジェクトのペイロードを `payload` に設定します。ペイロードの形式をととのえるのは呼び出し側の責任です。オプションの `charset` はメッセージのデフォルト文字セットを設定します。詳しくは `set_charset()` を参照してください。

2.2.2 で変更された仕様: `charset` 引数の追加

`set_charset(charset)`



ペイロードの文字セットを *charset* に変更します。ここには `Charset` インスタンス (`email.Charset` 参照)、文字セット名をあらわす文字列、あるいは `None` のいずれかが指定できます。文字列を指定した場合、これは `Charset` インスタンスに変換されます。*charset* が `None` の場合、*charset* パラメータは `Content-Type`: ヘッダから除去されます。これ以外のものを文字セットとして指定した場合、`TypeError` が発生します。

ここでいうメッセージとは、`charset.input_charset` でエンコードされた `text/*` 形式のものを仮定しています。これは、もし必要とあらばプレーンテキスト形式を変換するさいに `charset.output_charset` のエンコードに変換されます。MIME ヘッダ (`MIME-Version`:, `Content-Type`:, `Content-Transfer-Encoding`:) は必要に応じて追加されます。

2.2.2 で追加された仕様です。

`get_charset()`

そのメッセージ中のペイロードの `Charset` インスタンスを返します。2.2.2 で追加された仕様です。

以下のメソッドは、メッセージの RFC 2822 ヘッダにアクセスするためのマップ (辞書) 形式のインタフェイスを実装したものです。これらのメソッドと、通常のマップ (辞書) 型はまったく同じ意味をもつわけではないことに注意してください。たとえば辞書型では、同じキーが複数あることは許されていませんが、ここでは同じメッセージヘッダが複数ある場合があります。また、辞書型では `keys()` で返されるキーの順序は保証されていませんが、`Message` オブジェクト内のヘッダはつねに元のメッセージ中に現れた順序、あるいはそのあとに追加された順序で返されます。削除され、その後ふたたび追加されたヘッダはリストの一番最後に現れます。

こういった意味のちがいは意図的なもので、最大の利便性をもつようにつくられています。

注意: どんな場合も、メッセージ中のエンベロープヘッダはこのマップ形式のインタフェイスには含まれません。

`__len__()`

複製されたものもふくめてヘッダ数の合計を返します。

`__contains__(name)`

メッセージオブジェクトが *name* という名前のフィールドを持っていれば `true` を返します。この検査では名前の大文字小文字は区別されません。*name* は最後にコロンをふくんでいてはいけません。このメソッドは以下のように `in` 演算子で使われます:

```
if 'message-id' in myMessage:
    print 'Message-ID:', myMessage['message-id']
```

`__getitem__(name)`

指定された名前のヘッダフィールドの値を返します。*name* は最後にコロンをふくんでいてはいけません。そのヘッダがない場合は `None` が返され、`KeyError` 例外は発生しません。

注意: 指定された名前のフィールドがメッセージのヘッダに 2 回以上現れている場合、どちらの値が返されるかは未定義です。ヘッダに存在するフィールドの値をすべて取り出したい場合は `get_all()` メソッドを使ってください。

`__setitem__(name, val)`

メッセージヘッダに *name* という名前の *val* という値をもつフィールドをあらたに追加します。このフィールドは現在メッセージに存在するフィールドのいちばん後に追加されます。

注意: このメソッドでは、すでに同一の名前で存在するフィールドは上書きされません。もしメッセージが名前 *name* をもつフィールドをひとつしか持たないにしたければ、最初にそれを除去してください。たとえば:



```
del msg['subject']
msg['subject'] = 'PythonPythonPython!'
```

**\_\_delitem\_\_**(*name*)

メッセージのヘッダから、*name* という名前をもつフィールドをすべて除去します。たとえこの名前をもつヘッダが存在していなくても例外は発生しません。

**has\_key**(*name*)

メッセージが *name* という名前をもつヘッダフィールドを持っていれば真を、そうでなければ偽を返します。

**keys**()

メッセージ中にあるすべてのヘッダのフィールド名のリストを返します。

**values**()

メッセージ中にあるすべてのフィールドの値のリストを返します。

**items**()

メッセージ中にあるすべてのヘッダのフィールド名とその値を 2-タプルのリストとして返します。

**get**(*name*[, *failobj*])

指定された名前をもつフィールドの値を返します。これは指定された名前がないときにオプション引数の *failobj* (デフォルトでは None) を返すことをのぞけば、**\_\_getitem\_\_**() と同じです。

役に立つメソッドをいくつか紹介します:

**get\_all**(*name*[, *failobj*])

*name* の名前をもつフィールドのすべての値からなるリストを返します。該当する名前のヘッダがメッセージ中に含まれていない場合は *failobj* (デフォルトでは None) が返されます。

**add\_header**(*\_name*, *\_value*, **\*\*\_params**)

拡張ヘッダ設定。このメソッドは **\_\_setitem\_\_**() と似ていますが、追加のヘッダ・パラメータをキーワード引数で指定できるところが違います。*\_name* に追加するヘッダフィールドを、*\_value* にそのヘッダの最初の値を渡します。

キーワード引数辞書 *\_params* の各項目ごとに、そのキーがパラメータ名として扱われ、キー名にふくまれるアンダースコアはハイフンに置換されます (なぜならハイフンは通常の Python 識別子としては使えないからです)。ふつう、パラメータの値が None 以外のときは、`key="value"` の形で追加されます。パラメータの値が None のときはキーのみが追加されます。

例を示しましょう:

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

こうするとヘッダには以下のように追加されます。

```
Content-Disposition: attachment; filename="bud.gif"
```

**replace\_header**(*\_name*, *\_value*)

ヘッダの置換。*\_name* と一致するヘッダで最初に見つかったものを置き換えます。このときヘッダの順序とフィールド名の太文字小文字は保存されます。一致するヘッダがない場合、`KeyError` が発生します。

2.2.2 で追加された仕様です。

**get\_content\_type**()

そのメッセージの content-type を返します。返された文字列は強制的に小文字で maintype/subtype の形式に変換されます。メッセージ中に Content-Type: ヘッダがない場合、デフォルトの content-type は

`get_default_type()` が返す値によって与えられます。RFC 2045 によればメッセージはつねにデフォルトの content-type をもっているため、`get_content_type()` はつねになんらかの値を返すはずです。

RFC 2045 はメッセージのデフォルト content-type を、それが multipart/digest コンテナに現れているとき以外は text/plain に規定しています。あるメッセージが multipart/digest コンテナ中にある場合、その content-type は message/rfc822 になります。もし Content-Type: ヘッダが適切でない content-type 書式だった場合、RFC 2045 はそのデフォルトを text/plain として扱うよう定めています。

2.2.2 で追加された仕様です。

`get_content_maintype()`

そのメッセージの主 content-type を返します。これは `get_content_type()` によって返される文字列の maintype 部分です。

2.2.2 で追加された仕様です。

`get_content_subtype()`

そのメッセージの副 content-type (sub content-type、subtype) を返します。これは `get_content_type()` によって返される文字列の subtype 部分です。

2.2.2 で追加された仕様です。

`get_default_type()`

デフォルトの content-type を返します。ほとんどのメッセージではデフォルトの content-type は text/plain ですが、メッセージが multipart/digest コンテナに含まれているときだけ例外的に message/rfc822 になります。

2.2.2 で追加された仕様です。

`set_default_type(ctype)`

デフォルトの content-type を設定します。ctype は text/plain あるいは message/rfc822 である必要がありますが、強制ではありません。デフォルトの content-type はヘッダの Content-Type: には格納されません。

2.2.2 で追加された仕様です。

`get_params([failobj[, header[, unquote]]])`

メッセージの Content-Type: パラメータをリストとして返します。返されるリストは キー/値の組からなる 2 要素タプルが連なったものであり、これらは '=' 記号で分離されています。 '=' の左側はキーになり、右側は値になります。パラメータ中に '=' がなかった場合、値の部分は空文字列になり、そうでなければその値は `get_param()` で説明されている形式になります。また、オプション引数 *unquote* が True (デフォルト) である場合、この値は *unquote* されます。

オプション引数 *failobj* は、Content-Type: ヘッダが存在しなかった場合に返すオブジェクトです。オプション引数 *header* には Content-Type: のかわりに検索すべきヘッダを指定します。

2.2.2 で変更された仕様: *unquote* が追加されました

`get_param(param[, failobj[, header[, unquote]]])`

メッセージの Content-Type: ヘッダ中のパラメータ *param* を文字列として返します。そのメッセージ中に Content-Type: ヘッダが存在しなかった場合、*failobj* (デフォルトは None) が返されます。

オプション引数 *header* が与えられた場合、Content-Type: のかわりにそのヘッダが使用されます。

パラメータのキー比較は常に大文字小文字を区別しません。返り値は文字列か 3 要素のタプルで、タプルになるのはパラメータが RFC 2231 エンコードされている場合です。3 要素タプルの場合、各要素の値は (CHARSET, LANGUAGE, VALUE) の形式になっています。CHARSET と LANGUAGE は None になることがあり、その場合 VALUE は us-ascii 文字セットでエンコードされているとみなさね

ばならないので注意してください。普段は `LANGUAGE` を無視できます。この関数を使うアプリケーションは 3 要素タプルが返された場合を想定している必要があります。この場合、Unicode 文字列にして返すなどの処理をおこなうことが考えられます:

```
param = msg.get_param('foo')
if isinstance(param, tuple):
    param = unicode(param[2], param[0] or 'us-ascii')
```

いずれの場合もパラメータの値は (文字列であれ 3 要素タプルの `VALUE` 項目であれ) つねに `unquote` されます。ただし、`unquote` が `False` に指定されている場合は `unquote` されません。

2.2.2 で変更された仕様: `unquote` 引数の追加、3 要素タプルが返り値になる可能性あり

```
set_param(param, value[, header[, requote[, charset[, language]]]])
```

Content-Type: ヘッダ中のパラメータを設定します。指定されたパラメータがヘッダ中にすでに存在する場合、その値は `value` に置き換えられます。Content-Type: ヘッダがまだこのメッセージ中に存在していない場合、RFC 2045 にしたがってこの値には `text/plain` が設定され、新しいパラメータ値が末尾に追加されます。

オプション引数 `header` が与えられた場合、Content-Type: のかわりにそのヘッダが使用されます。オプション引数 `unquote` が `False` でない限り、この値は `unquote` されます (デフォルトは `True`)。

オプション引数 `charset` が与えられると、そのパラメータは RFC 2231 に従ってエンコードされます。オプション引数 `language` は RFC 2231 の言語を指定しますが、デフォルトではこれは空文字列となります。 `charset` と `language` はどちらも文字列である必要があります。

2.2.2 で追加された仕様です。

```
del_param(param[, header[, requote]])
```

指定されたパラメータを Content-Type: ヘッダ中から完全にとりのぞきます。ヘッダはそのパラメータと値がない状態に書き換えられます。 `requote` が `False` でない限り (デフォルトでは `True` です)、すべての値は必要に応じて `quote` されます。オプション変数 `header` が与えられた場合、Content-Type: のかわりにそのヘッダが使用されます。

2.2.2 で追加された仕様です。

```
set_type(type[, header[, requote]])
```

Content-Type: ヘッダの maintype と subtype を設定します。 `type` は maintype/subtype という形の文字列でなければなりません。それ以外の場合は `ValueError` が発生します。

このメソッドは Content-Type: ヘッダを置き換えますが、すべてのパラメータはそのままにします。 `requote` が `False` の場合、これはすでに存在するヘッダを `quote` せず放置しますが、そうでない場合は自動的に `quote` します (デフォルト動作)。

オプション変数 `header` が与えられた場合、Content-Type: のかわりにそのヘッダが使用されます。Content-Type: ヘッダが設定される場合には、MIME-Version: ヘッダも同時に付加されます。

2.2.2 で追加された仕様です。

```
get_filename([failobj])
```

そのメッセージ中の Content-Disposition: ヘッダにある、`filename` パラメータの値を返します。目的のヘッダが欠けていたり、`filename` パラメータがない場合には `failobj` が返されます。返される文字列はつねに `Utils.unquote()` によって `unquote` されます。

```
get_boundary([failobj])
```

そのメッセージ中の Content-Type: ヘッダにある、`boundary` パラメータの値を返します。目的のヘッダが欠けていたり、`boundary` パラメータがない場合には `failobj` が返されます。返される文字列はつねに `Utils.unquote()` によって `unquote` されます。

`set_boundary(boundary)`

メッセージ中の Content-Type: ヘッダにある、boundary パラメータに値を設定します。`set_boundary()` は必要に応じて *boundary* を quote します。そのメッセージが Content-Type: ヘッダを含んでいない場合、`HeaderParseError` が発生します。

注意: このメソッドを使うのは、古い Content-Type: ヘッダを削除して新しい boundary をもったヘッダを `add_header()` で足すのとは少し違います。`set_boundary()` は一連のヘッダ中での Content-Type: ヘッダの位置を保つからです。しかし、これは元の Content-Type: ヘッダ中に存在していた連続する行の順番までは 保ちません。

`get_content_charset([failobj])`

そのメッセージ中の Content-Type: ヘッダにある、charset パラメータの値を返します。値はすべて小文字に変換されます。メッセージ中に Content-Type: がなかったり、このヘッダ中に boundary パラメータがない場合には *failobj* が返されます。

注意: これは `get_charset()` メソッドとは異なります。こちらのほうは文字列のかわりに、そのメッセージボディのデフォルトエンコーディングの Charset インスタンスを返します。

2.2.2 で追加された仕様です。

`get_charsets([failobj])`

メッセージ中に含まれる文字セットの名前をすべてリストにして返します。そのメッセージが multipart である場合、返されるリストの各要素がそれぞれの subpart のペイロードに対応します。それ以外の場合、これは長さ 1 のリストを返します。

リスト中の各要素は文字列であり、これは対応する subpart 中のそれぞれの Content-Type: ヘッダにある charset の値です。しかし、その subpart が Content-Type: をもっていないか、charset がないか、あるいは MIME maintype が text でないいずれかの場合には、リストの要素として *failobj* が返されます。

`walk()`

`walk()` メソッドは多目的のジェネレータで、これはあるメッセージオブジェクトツリー中のすべての part および subpart をわたり歩くのに使えます。順序は深さ優先です。おそらく典型的な用法は、`walk()` を for ループ中でのイテレータとして使うことでしょう。ループを一回まわるごとに、次の subpart が返されるのです。

以下の例は、multipart メッセージのすべての part において、その MIME タイプを表示していくものです。

```
>>> for part in msg.walk():
>>>     print part.get_content_type()
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
```

Message オブジェクトはオプションとして 2 つのインスタンス属性をとることができます。これはある MIME メッセージからプレーンテキストを生成するのに使うことができます。

`preamble`

MIME ドキュメントの形式では、ヘッダ直後にくる空行と最初の multipart 境界をあらわす文字列のあいだにいくらかのテキスト (訳注: preamble, 序文) を埋めこむことを許しています。このテキストは標準的な MIME の範疇からはみ出しているため、MIME 形式を認識するメールソフトからこれらは通常まったく見えません。しかしメッセージのテキストを生で見る場合、あるいはメッセージを

MIME 対応していないメールソフトで見える場合、このテキストは目に見えることになります。

*preamble* 属性は MIME ドキュメントに加えるこの最初の MIME 範囲外テキストを含んでいます。Parser があるテキストをヘッダ以降に発見したが、それはまだ最初の MIME 境界文字列が現れる前だった場合、パーザはそのテキストをメッセージの *preamble* 属性に格納します。Generator がある MIME メッセージからプレーンテキスト形式を生成するとき、これはそのテキストをヘッダと最初の MIME 境界の間に挿入します。詳細は `email.Parser` および `email.Generator` を参照してください。

注意: そのメッセージに *preamble* がない場合、*preamble* 属性には `None` が格納されます。

#### **epilogue**

*epilogue* 属性はメッセージの最後の MIME 境界文字列からメッセージ末尾までのテキストを含むもので、それ以外は *preamble* 属性と同じです。

注意: multipart メッセージからフラットなテキストを生成するとき、(標準的な Generator を使ったとして) もしそのメッセージに *epilogue* がない場合、最後の MIME 境界文字列のあとには改行文字が追加されません。もしそのメッセージオブジェクトが *epilogue* をもっており、それが改行文字から始まっていない場合、改行文字が MIME 境界文字列のあとに追加されます。これはややぶかっこうに見えますが、ほとんどの場合はこれでうまくいきます。要するに、もし multipart 境界のあとにかならず改行を入れるようにしたければ、*epilogue* に空文字列を入れておけばよいのです。

#### **推奨されないメソッド**

以下のメソッドは email バージョン 2 ではもはや時代遅れとなっており、推奨されません。万全を記すためここに記載しておきます。

##### **`add_payload(payload)`**

そのメッセージオブジェクトに存在するペイロードに *payload* を追加します。もしこのメソッドを呼ぶ前にそのメッセージのペイロードが `None` だった (つまり一度もセットされていない) 場合、呼び出し後のペイロードは引数 *payload* になります。

もしそのメッセージのペイロードがリストだった (つまり `is_multipart()` が 1 を返した) 場合、すでに存在するペイロードのリストの末尾に *payload* が付加されます。

ペイロードがこれ以外の形式だった場合、`add_payload()` は新しいペイロードを古いペイロードと *payload* からなるリストに自動的に変換します。が、これはそのドキュメントがすでに MIME multipart ドキュメントであるときだけです。この条件が満たされるのはそのメッセージの Content-Type: ヘッダの主形式 (maintype) が multipart であるときか、あるいはそのメッセージに Content-Type: がないときだけです。これ以外の場合、メソッドは `MultipartConversionError` を発生します。

リリース 2.2.2 以降で撤廃された仕様です。これのかわりに `attach()` メソッドを使ってください。

##### **`get_type([failobj])`**

そのメッセージの content-type を Content-Type: ヘッダから取得した maintype/subtype という形式の文字列で返します。取得した文字列は強制的に小文字に変換されます。

そのメッセージ中に Content-Type: ヘッダが存在しなかった場合、*failobj* (デフォルトは `None`) が返されます。

リリース 2.2.2 以降で撤廃された仕様です。これのかわりに `get_content_type()` メソッドを使ってください。

##### **`get_main_type([failobj])`**

そのメッセージの主 content-type を返します。これは `get_type()` によって返される文字列の maintype 部分です。*failobj* の働きは `get_type()` と同じです。



リリース 2.2.2 以降で撤廃された仕様です。これのかわりに `get_content_maintype()` メソッドを使ってください。

`get_subtype([failobj])`

そのメッセージの副 content-type (sub content-type、subtype) を返します。これは `get_type()` によって返される文字列の subtype 部分です。failobj の働きは `get_type()` と同じです。

リリース 2.2.2 以降で撤廃された仕様です。これのかわりに `get_content_subtype()` メソッドを使ってください。

## 12.2.2 電子メールメッセージを解析 (パース) する

メッセージオブジェクト構造体をつくるには 2 つの方法があります。ひとつはまったくのスクラッチから Message を生成して、これを `attach()` と `set_payload()` 呼び出しを介してつなげていく方法で、もうひとつは電子メールメッセージのフラットなテキスト表現を解析 (parse、パース) する方法です。

email パッケージでは、MIME 文書をふくむ、ほとんどの電子メールの文書構造に対応できる標準的なパーザ (解析器) を提供しています。このパーザに文字列あるいはファイルオブジェクトを渡せば、パーザはそのオブジェクト構造の基底となる (root の) Message インスタンスを返します。簡単な非 MIME メッセージであれば、この基底オブジェクトのペイロードはたんにメッセージのテキストを格納する文字列になるでしょう。MIME メッセージであれば、基底オブジェクトはその `is_multipart()` メソッドに対して True を返します。そして、その各 subpart に `get_payload()` メソッドおよび `walk()` メソッドを介してアクセスすることができます。

このパーザは、ある制限された方法で拡張できます。また、もちろん自分でご自分のパーザを完全に無から実装することもできます。email パッケージについているパーザと Message クラスの間に隠された秘密の関係はなにもありませんので、ご自分で実装されたパーザも、それが必要とするやりかたでメッセージオブジェクトツリーを作成することができます。

おもなパーザクラスは Parser です。これは電子メールのヘッダおよびペイロードを両方解析します。multipart メッセージの場合、これはそのコンテナのボディを再帰的に解析します。解析には 2 つのモードがサポートされており、ひとつは *strict* 解析で、これはふつういかなる RFC 非準拠なメッセージも受けつけません。もうひとつは *lax* 解析で、こちらはよく知られた MIME の書式上の問題に対処しようとしています。

email.Parser モジュールはまた、HeaderParser と呼ばれる 2 番目のクラスも提供しています。これはメッセージのヘッダのみを処理したい場合に使うことができ、ずっと高速な処理がおこなえます。なぜならこれはメッセージ本体を解析しようとはしないからです。かわりに、そのペイロードにはメッセージ本体の生の文字列が格納されます。HeaderParser クラスは Parser クラスと同じ API をもっています。

### Parser クラスの API

`class Parser([_class[, strict]])`

Parser クラスのコンストラクタです。オプション引数 `_class` をとることができ、これは呼び出し可能なオブジェクト (関数やクラス) でなければならず、メッセージ内コンポーネント (sub-message object) が作成される時はいつでもその“工場”として使用できなければなりません。デフォルトではこれは Message になっています (email.Message 参照)。この“工場”は引数なしで呼び出されます。

オプション引数である *strict* フラグは、strict 解析と lax 解析のどちらをおこなうかを指定します。MIME 終端文字列が欠けているとか、メッセージにその他の書式上の問題がある場合、Parser は MessageParseError を発生させます。しかし、lax 解析が有効になっている場合、Parser は崩れた書式からなんとかして有効なメッセージ構造を生成しようと試みます (これは MessageParseError が絶対に発生しないということではありません、ある種の異常な書式のメッセージはやはり解析でき



ません)。デフォルトでは、*strict* フラグは `False` になっています。これは通常 *lax* 解析がもっとも理にかなったふるまいを提供するためです。

2.2.2 で変更された仕様: *strict* フラグが追加されました

それ以外の Parser メソッドは以下のとおりです:

`parse(fp[, headersonly])`

ファイルなどストリーム形式<sup>1</sup>のオブジェクト *fp* からすべてのデータを読み込み、得られたテキストを解析して基底 (root) メッセージオブジェクト構造を返します。*fp* はストリーム形式のオブジェクトで `readline()` および `read()` 両方のメソッドをサポートしている必要があります。

*fp* に格納されているテキスト文字列は、一連の RFC 2822 形式のヘッダおよびヘッダ継続行 (header continuation lines) によって構成されている必要があります。オプションとして、最初にエンペローブヘッダが来ることもできます。ヘッダ部分はデータの終端か、ひとつの空行によって終了したとみなされます。ヘッダ部分に続くデータはメッセージ本体となります (MIME エンコードされた subpart を含んでいるかもしれません)。

オプション引数 *headersonly* はヘッダ部分を解析しただけで終了するか否かを指定します。デフォルトの値は `False` で、これはそのファイルの内容すべてを解析することを意味しています。

2.2.2 で変更された仕様: *headersonly* フラグが追加されました

`parsestr(text[, headersonly])`

メソッドに似ていますが、ファイルなどのストリーム形式のかわりに文字列を引数としてとるところが違います。文字列に対してこのメソッドを呼ぶことは、*text* を `StringIO` インスタンスとして作成して `parse()` を適用するのと同じです。

オプション引数 *headersonly* は `parse()` メソッドと同じです。

2.2.2 で変更された仕様: *headersonly* フラグが追加されました

ファイルや文字列からメッセージオブジェクト構造を作成するのはかなりよくおこなわれる作業なので、便宜上次のような 2 つの関数が提供されています。これらは `email` パッケージのトップレベルの名前空間で使用できます。

`message_from_string(s[, _class[, strict]])`

文字列からメッセージオブジェクト構造を作成し返します。これは `Parser().parsestr(s)` とまったく同じです。オプション引数 *\_class* および *strict* は `Parser` クラスのコンストラクタと同様に解釈されます。

2.2.2 で変更された仕様: *strict* フラグが追加されました

`message_from_file(fp[, _class[, strict]])`

Open されたファイルオブジェクトからメッセージオブジェクト構造を作成し返します。これは `Parser().parse(fp)` とまったく同じです。オプション引数 *\_class* および *strict* は `Parser` クラスのコンストラクタと同様に解釈されます。

2.2.2 で変更された仕様: *strict* フラグが追加されました

対話的な Python プロンプトでこの関数を使用するとすれば、このようになります:

```
>>> import email
>>> msg = email.message_from_string(myString)
```

---

<sup>1</sup>file-like object

## 追加事項

以下はテキスト解析の際に適用されるいくつかの規約です:

- ほとんどの 非 multipart 形式のメッセージは単一の文字列ペイロードをもつ単一のメッセージオブジェクトとして解析されます。このオブジェクトは `is_multipart()` に対して `False` を返します。このオブジェクトに対する `get_payload()` メソッドは文字列オブジェクトを返します。
- multipart 形式のメッセージはすべてメッセージ内コンポーネント (sub-message object) のリストとして解析されます。外側のコンテナメッセージオブジェクトは `is_multipart()` に対して `True` を返し、このオブジェクトに対する `get_payload()` メソッドは Message subpart のリストを返します。
- message/\* の Content-Type をもつほとんどのメッセージ (例: message/deliver-status や message/rfc822 など) もコンテナメッセージオブジェクトとして解析されますが、ペイロードのリストの長さは 1 になります。このオブジェクトは `is_multipart()` メソッドに対して `True` を返し、リスト内にあるひとつだけの要素がメッセージ内のコンポーネントオブジェクトになります。

### 12.2.3 MIME 文書を生成する

よくある作業のひとつは、メッセージオブジェクト構造からフラットな電子メールテキストを生成することです。この作業は `smtplib` や `nntplib` モジュールを使ってメッセージを送信したり、メッセージをコンソールに出力したりするときに必要になります。あるメッセージオブジェクト構造をとってきて、そこからフラットなテキスト文書を生成するのは `Generator` クラスの仕事です。

繰り返しになりますが、`email.Parser` モジュールと同じく、この機能は既存の `Generator` だけに限られるわけではありません。これらはご自身でゼロから作りあげることもできます。しかし、既存のジェネレータはほとんどの電子メールを標準に沿ったやり方で生成する方法を知っていますし、MIME メッセージも非 MIME メッセージも扱えます。さらにこれはフラットなテキストから `Parser` クラスを使ってメッセージ構造に変換し、それをまたフラットなテキストに戻しても、結果が冪等<sup>2</sup>になるよう設計されています。

`Generator` クラスで公開されているメソッドには、以下のようなものがあります:

```
class Generator(outfp[, mangle_from_[, maxheaderlen]])
```

`Generator` クラスのコンストラクタは `outfp` と呼ばれるストリーム形式<sup>3</sup>のオブジェクトひとつを引数にとります。`outfp` は `write()` メソッドをサポートし、Python 拡張 `print` 文の出力ファイルとして使えるようになっている必要があります。

オプション引数 `mangle_from_` はフラグで、`True` のときはメッセージ本体に現れる行頭のすべての 'From' という文字列の最初に '>' という文字を追加します。これは、このような行が Unix の mailbox 形式のエンベロープヘッダ区切り文字列として誤認識されるのを防ぐための、移植性ある唯一の方法です (詳しくは WHY THE CONTENT-LENGTH FORMAT IS BAD (なぜ Content-Length 形式が有害か) を参照してください)。デフォルトでは `mangle_from_` は `True` になっていますが、Unix の mailbox 形式ファイルに出力しないのならばこれは `False` に設定してもかまいません。

オプション引数 `maxheaderlen` は連続していないヘッダの最大長を指定します。ひとつのヘッダ行が `maxheaderlen` (これは文字数です、tab は空白 8 文字に展開されます) よりも長い場合、ヘッダは `email.Header` クラスで定義されているように途中で折り返され、間にはセミコロンが挿入されます。もしセミコロンが見つからない場合、そのヘッダは放置されます。ヘッダの折り返しを禁止するにはこの値にゼロを指定してください。デフォルトは 78 文字で、RFC 2822 で推奨されている (ですが強制ではありません) 値です。

<sup>2</sup>訳注: idempotent、その操作を何回くり返しても 1 回だけ行ったのと結果が同じになること。

<sup>3</sup>訳注: file-like object

これ以外のパブリックな Generator メソッドは以下のとおりです:

**flatten**(*msg*[, *unixfrom*])

*msg* を基点とするメッセージオブジェクト構造体の文字表現を出力します。出力先のファイルにはこの Generator インスタンスが作成されたときに指定されたものが使われます。各 subpart は深さ優先順序 (depth-first) で出力され、得られるテキストは適切に MIME エンコードされたものになっています。

オプション引数 *unixfrom* は、基点となるメッセージオブジェクトの最初の RFC 2822 ヘッダが現れる前に、エンベロープヘッダ区切り文字列を出力することを強制するフラグです。そのメッセージオブジェクトがエンベロープヘッダをもたない場合、標準的なエンベロープヘッダが自動的に作成されます。デフォルトではこの値は `False` に設定されており、エンベロープヘッダ区切り文字列は出力されません。

注意: 各 subpart に関しては、エンベロープヘッダは出力されません。

2.2.2 で追加された仕様です。

**clone**(*fp*)

この Generator インスタンスの独立したクローンを生成し返します。オプションはすべて同一になっています。

2.2.2 で追加された仕様です。

**write**(*s*)

文字列 *s* を既定のファイルに出力します。ここでいう出力先は Generator コンストラクタに渡した *outfp* のことをさします。この関数はただ単に拡張 `print` 文で使われる Generator インスタンスに対してファイル操作風の API を提供するためだけのものです。

ユーザの便宜をはかるため、メソッド `Message.as_string()` と `str(aMessage)` (つまり `Message.__str__()` のことです) をつかえばメッセージオブジェクトを特定の書式でフォーマットされた文字列に簡単に変換することができます。詳細は `email.Message` を参照してください。

`email.Generator` モジュールはひとつの派生クラスも提供しています。これは `DecodedGenerator` と呼ばれるもので、Generator 基底クラスと似ていますが、非 text 型の subpart を特定の書式でフォーマットされた表現形式で置きかえるところが違います。

**class DecodedGenerator**(*outfp*[, *mangle\_from\_*[, *maxheaderlen*[, *fnt*]])

このクラスは Generator から派生したもので、メッセージの subpart をすべて渡り歩きます。subpart の主形式が text だった場合、これはその subpart のペイロードをデコードして出力します。オプション引数 *mangle\_from\_* および *maxheaderlen* の意味は基底クラス Generator のそれと同じです。

Subpart の主形式が text ではない場合、オプション引数 *fnt* がそのメッセージペイロードのかわりのフォーマット文字列として使われます。*fnt* は '%(keyword)s' のような形式を展開し、以下のキーワードを認識します:

- type* – 非 text 型 subpart の MIME 形式
- maintype* – 非 text 型 subpart の MIME 主形式 (maintype)
- subtype* – 非 text 型 subpart の MIME 副形式 (subtype)
- filename* – 非 text 型 subpart のファイル名
- description* – 非 text 型 subpart につけられた説明文字列
- encoding* – 非 text 型 subpart の Content-transfer-encoding

*fnt* のデフォルト値は `None` です。こうすると以下の形式で出力します:

```
[Non-text %(type)s] part of message omitted, filename %(filename)s]
```

2.2.2 で追加された仕様です。

#### Deprecated methods

以下のメソッドは email バージョン 2 ではもはや時代遅れとなっており、推奨されません。万全を記すためここに記載しておきます。

`__call__(msg[, unixfrom])`

このメソッドは `flatten()` メソッドと同じです。

リリース 2.2.2 以降で撤廃された仕様です。これのかわりに `flatten()` メソッドを使ってください。

#### 12.2.4 電子メールおよび MIME オブジェクトをゼロから作成する

ふつう、メッセージオブジェクト構造はファイルまたは何がしかのテキストをパーザに通すことで得られます。パーザは与えられたテキストを解析し、基底となる `root` のメッセージオブジェクトを返します。しかし、完全なメッセージオブジェクト構造を何もないところから作成することもまた可能です。個別の `Message` を手で作成することさえできます。実際には、すでに存在するメッセージオブジェクト構造をとってきて、そこに新たな `Message` オブジェクトを追加したり、あるものを別のところへ移動させたりできます。これは MIME メッセージを切ったりおろしたりするために非常に便利なインターフェイスを提供します。

新しいメッセージオブジェクト構造は `Message` インスタンスを作成することにより作れます。ここに添付ファイルやその他適切なものをすべて手で加えてやればよいのです。MIME メッセージの場合、`email` パッケージはこれらを簡単におこなえるようにするためにいくつかの便利なサブクラスを提供しています。これらのサブクラスは `email` パッケージ内にある、そのクラスと同じ名前をもつモジュールから `import` してやります。たとえば:

```
import email.MIMEImage.MIMEImage
```

または以下のようにします:

```
from email.MIMEText import MIMEText
```

以下がそのサブクラスです:

```
class MIMEBase(_maintype, _subtype, **_params)
```

これはすべての MIME 用サブクラスの基底となるクラスです。とくに `MIMEBase` のインスタンスを直接作成することは (可能ではありますが) ふつうはしないでしょう。 `MIMEBase` は単により特化された MIME 用サブクラスのための便宜的な基底クラスとして提供されています。

`_maintype` は Content-Type: の主形式 (maintype) であり (text や image など)、`_subtype` は Content-Type: の副形式 (subtype) です (plain や gif など)。 `_params` は各パラメータのキーと値を格納した辞書であり、これは直接 `Message.add_header()` に渡されます。

`MIMEBase` クラスはつねに (`_maintype`、`_subtype`、および `_params` にもとづいた) Content-Type: ヘッダと、MIME-Version: ヘッダ (必ず 1.0 に設定される) を追加します。

```
class MIMENonMultipart()
```

`MIMEBase` のサブクラスで、これは multipart 形式でない MIME メッセージのための中間的な基底クラスです。このクラスのおもな目的は、通常 multipart 形式のメッセージに対してのみ意味をなす

attach() メソッドの使用をふせぐことです。もし attach() メソッドが呼ばれた場合、これは MultipartConversionError 例外が発生します。

2.2.2 で追加された仕様です。

```
class MIMEMultipart([subtype[, boundary[, _subparts[, _params]]]])
```

MIMEBase のサブクラスで、これは multipart 形式の MIME メッセージのための中間的な基底クラスです。オプション引数 `_subtype` はデフォルトでは `mixed` になっていますが、そのメッセージの副形式 (subtype) を指定するのに使うことができます。メッセージオブジェクトには `multipart/_subtype` という値をもつ Content-Type: ヘッダとともに、MIME-Version: ヘッダが追加されるでしょう。

オプション引数 `boundary` は multipart の境界文字列です。これが `None` の場合 (デフォルト)、境界は必要に応じて計算されます。

`_subparts` はそのペイロードの subpart の初期値からなるシーケンスです。このシーケンスはリストに変換できるようになっている必要があります。新しい subpart はつねに `Message.attach()` メソッドを使ってそのメッセージに追加できるようになっています。

Content-Type: ヘッダに対する追加のパラメータはキーワード引数 `_params` を介して取得あるいは設定されます。これはキーワード辞書になっています。

2.2.2 で追加された仕様です。

```
class MIMEAudio(_audiodata[, _subtype[, _encoder[, **_params]]])
```

MIMEAudio クラスは MIMENonMultipart のサブクラスで、主形式 (maintype) が `audio` の MIME オブジェクトを作成するのに使われます。`_audiodata` は実際の音声データを格納した文字列です。もしこのデータが標準の Python モジュール `sndhdr` によって認識できるものであれば、Content-Type: ヘッダの副形式 (subtype) は自動的に決定されます。そうでない場合はその画像の形式 (subtype) を `_subtype` で明示的に指定する必要があります。副形式が自動的に決定できず、`_subtype` の指定もない場合は、`TypeError` が発生します。

オプション引数 `_encoder` は呼び出し可能なオブジェクト (関数など) で、トランスポートのさいに画像の実際のエンコードをおこないます。このオブジェクトは MIMEAudio インスタンスの引数をひとつだけ取ることができます。この関数は、与えられたペイロードをエンコードされた形式に変換するのに `get_payload()` および `set_payload()` を使う必要があります。また、これは必要に応じて Content-Transfer-Encoding: あるいはそのメッセージに適した何らかのヘッダを追加する必要があります。デフォルトのエンコーディングは `base64` です。組み込みのエンコーダの詳細については `email.Encoders` を参照してください。

`_params` は MIMEBase コンストラクタに直接渡されます。

```
class MIMEImage(_imagedata[, _subtype[, _encoder[, **_params]]])
```

MIMEImage クラスは MIMENonMultipart のサブクラスで、主形式 (maintype) が `image` の MIME オブジェクトを作成するのに使われます。`_imagedata` は実際の画像データを格納した文字列です。もしこのデータが標準の Python モジュール `imgchr` によって認識できるものであれば、Content-Type: ヘッダの副形式 (subtype) は自動的に決定されます。そうでない場合はその画像の形式 (subtype) を `_subtype` で明示的に指定する必要があります。副形式が自動的に決定できず、`_subtype` の指定もない場合は、`TypeError` が発生します。

オプション引数 `_encoder` は呼び出し可能なオブジェクト (関数など) で、トランスポートのさいに画像の実際のエンコードをおこないます。このオブジェクトは MIMEImage インスタンスの引数をひとつだけ取ることができます。この関数は、与えられたペイロードをエンコードされた形式に変換するのに `get_payload()` および `set_payload()` を使う必要があります。また、これは必要に応じて Content-Transfer-Encoding: あるいはそのメッセージに適した何らかのヘッダを追加する必要があります。デフォルトのエンコーディングは `base64` です。組み込みのエンコーダの詳細については `email.Encoders` を参照してください。



`_params` は `MIMEBase` コンストラクタに直接渡されます。

```
class MIMEMessage(_msg[, _subtype])
```

`MIMEMessage` クラスは `MIMENonMultipart` のサブクラスで、主形式 (maintype) が `message` の `MIME` オブジェクトを作成するのに使われます。ペイロードとして使われるメッセージは `_msg` になります。これは `Message` クラス (あるいはそのサブクラス) のインスタンスでなければいけません。そうでない場合、この関数は `TypeError` を発生します。

オプション引数 `_subtype` はそのメッセージの副形式 (subtype) を設定します。デフォルトではこれは `rfc822` になっています。

```
class MIMEText(_text[, _subtype[, _charset[, _encoder]]])
```

`MIMEText` クラスは `MIMENonMultipart` のサブクラスで、主形式 (maintype) が `text` の `MIME` オブジェクトを作成するのに使われます。ペイロードの文字列は `_text` になります。`_subtype` には副形式 (subtype) を指定し、デフォルトは `plain` です。`_charset` はテキストの文字セットで、`MIMENonMultipart` コンストラクタに引数として渡されます。デフォルトではこの値は `us-ascii` になっています。テキストデータに対しては文字コードの推定やエンコードはまったく行われません。

リリース 2.2.2 以降で撤廃された仕様です。The `_encoding` 引数はもはや時代遅れとなっており、推奨されません。現在ではエンコードは `_charset` 引数にもとづき暗黙のうちに行われます。

## 12.2.5 国際化されたヘッダ

RFC 2822 は電子メールメッセージの形式を規定する基本規格です。これはほとんどの電子メールが ASCII 文字のみで構成されていたころ普及した RFC 822 標準から発展したものです。RFC 2822 は電子メールがすべて 7-bit ASCII 文字のみから構成されていると仮定して作られた仕様です。

もちろん、電子メールが世界的に普及するにつれ、この仕様は国際化されてきました。今では電子メールに言語依存の文字セットを使うことができます。基本規格では、まだ電子メールメッセージを 7-bit ASCII 文字のみを使って転送するよう要求していますので、多くの RFC でどうやって非 ASCII の電子メールを RFC 2822 準拠な形式にエンコードするかが記述されています。これらの RFC は以下のものを含みます: RFC 2045、RFC 2046、RFC 2047、および RFC 2231。email パッケージは、`email.Header` および `email.Charset` モジュールでこれらの規格をサポートしています。

ご自分の電子メールヘッダ、たとえば `Subject:` や `To:` などのフィールドに非 ASCII 文字を入れたい場合、`Header` クラスを使う必要があります。`Message` オブジェクトの該当フィールドに文字列ではなく、`Header` インスタンスを値として使うのです。たとえば:

```
>>> from email.Message import Message
>>> from email.Header import Header
>>> msg = Message()
>>> h = Header('p\xF6stal', 'iso-8859-1')
>>> msg['Subject'] = h
>>> print msg.as_string()
Subject: =?iso-8859-1?q?p=F6stal?=
```

`Subject:` フィールドに非 ASCII 文字をふくめていることに注目してください。ここでは、含めたいバイト列がエンコードされている文字セットを指定して `Header` インスタンスを作成することによって実現しています。のちにこの `Message` インスタンスからフラットなテキストを生成するさいに、この `Subject:` フィールドは RFC 2047 準拠の適切な形式にエンコードされます。MIME 機能のついているメーラなら、このヘッダに埋めこまれた ISO-8859-1 文字をただしく表示するでしょう。



2.2.2 で追加された仕様です。

以下は Header クラスの説明です:

```
class Header ([s[, charset[, maxlinelen[, header_name[, continuation_ws[, errors]]]]])
```

別の文字セットの文字列をふくむ MIME 準拠なヘッダを作成します。

オプション引数 *s* はヘッダの値の初期値です。これが None の場合 (デフォルト)、ヘッダの初期値は設定されません。この値はあとから `append()` メソッドを呼び出すことによって追加することができます。*s* はバイト文字列か、あるいは Unicode 文字列でもかまいません。この意味については `append()` の項を参照してください。

オプション引数 *charset* には 2 つの目的があります。ひとつは `append()` メソッドにおける *charset* 引数と同じものです。もうひとつの目的は、これ以降 *charset* 引数を省略した `append()` メソッド呼び出しすべてにおける、デフォルト文字セットを決定するものです。コンストラクタに *charset* が与えられない場合 (デフォルト)、初期値の *s* および以後の `append()` 呼び出しにおける文字セットとして `us-ascii` が使われます。

行の最大長は *maxlinelen* によって明示的に指定できます。最初の行を (Subject: などの *s* に含まれないフィールドヘッダの責任をとるため) 短く切りとる場合、*header\_name* にそのフィールド名を指定してください。*maxlinelen* のデフォルト値は 76 であり、*header\_name* のデフォルト値は None です。これはその最初の行を長い、切りとられたヘッダとして扱わないことを意味します。

オプション引数 *continuation\_ws* は RFC 2822 準拠の折り返し用余白文字で、ふつうこれは空白か、ハードウェアタブ文字 (hard tab) である必要があります。ここで指定された文字は複数にわたる行の行頭に挿入されます。

オプション引数 *errors* は、`append()` メソッドにそのまま渡されます。

```
append(s[, charset[, errors]])
```

この MIME ヘッダに文字列 *s* を追加します。

オプション引数 *charset* がもし与えられた場合、これは Charset インスタンス (`email.Charset` を参照) か、あるいは文字セットの名前でなければなりません。この場合は Charset インスタンスに変換されます。この値が None の場合 (デフォルト)、コンストラクタで与えられた *charset* が使われます。

*s* はバイト文字列か、Unicode 文字列です。これはバイト文字列 (`isinstance(s, str)` が真) の場合、*charset* はその文字列のエンコーディングであり、これが与えられた文字セットでうまくデコードできないときは `UnicodeError` が発生します。

いっぽう *s* が Unicode 文字列の場合、*charset* はその文字列の文字セットを決定するためのヒントとして使われます。この場合、RFC 2822 準拠のヘッダは RFC 2047 の規則をもちいて作成され、Unicode 文字列は以下の文字セットを (この優先順位で) 適用してエンコードされます: `us-ascii`、*charset* で与えられたヒント、それもなければ `utf-8`。最初の文字セットは `UnicodeError` をなるべくふせぐために使われます。

オプション引数 *errors* は `unicode()` 又は `ustr.encode()` の呼び出し時に使用し、デフォルト値は “strict” です。

```
encode([splitchars])
```

メッセージヘッダを RFC に沿ったやり方でエンコードします。おそらく長い行は折り返され、非 ASCII 部分は base64 または quoted-printable エンコーディングで包含されるでしょう。オプション引数 *splitchars* には長い ASCII 行を分割する文字の文字列を指定し、RFC 2822 の *highest level syntactic breaks* の大まかなサポートの為に使用します。この引数は RFC 2047 でエンコードされた行には影響しません。

Header クラスは、標準の演算子や組み込み関数をサポートするためのメソッドもいくつか提供しています。

`__str__()`

`Header.encode()` と同じです。 `str(aHeader)` などとすると有用でしょう。

`__unicode__()`

組み込みの `unicode()` 関数の補助です。ヘッダを Unicode 文字列として返します。

`__eq__(other)`

このメソッドは、ふたつの Header インスタンスどうしが等しいかどうか判定するのに使えます。

`__ne__(other)`

このメソッドは、ふたつの Header インスタンスどうしが異なっているかどうかを判定するのに使えます。

さらに、`email.Header` モジュールは以下のような便宜的な関数も提供しています。

`decode_header(header)`

文字セットを変換することなしに、メッセージのヘッダをデコードします。ヘッダの値は *header* に渡します。

この関数はヘッダのそれぞれのデコードされた部分ごとに、`(decoded_string, charset)` という形式の 2 要素タプルからなるリストを返します。*charset* はヘッダのエンコードされていない部分に対しては `None` を、それ以外の場合はエンコードされた文字列が指定している文字セットの名前を小文字からなる文字列で返します。

以下はこの使用例です:

```
>>> from email.Header import decode_header
>>> decode_header('=?iso-8859-1?q?p=F6stal?=' )
[('p\\xf6stal', 'iso-8859-1')]
```

`make_header(decoded_seq[, maxlenen[, header_name[, continuation_ws]]])`

`decode_header()` によって返される 2 要素タプルのリストから Header インスタンスを作成します。

`decode_header()` はヘッダの値をとってきて、`(decoded_string, charset)` という形式の 2 要素タプルからなるリストを返します。ここで *decoded\_string* はデコードされた文字列、*charset* はその文字セットです。

この関数はこれらのリストの項目から、Header インスタンスを返します。オプション引数 *maxlinelen*、*header\_name* および *continuation\_ws* は Header コンストラクタに与えるものと同じです。

## 12.2.6 文字セットの表現

このモジュールは文字セットを表現する `Charset` クラスと電子メールメッセージにふくまれる文字セット間の変換、および文字セットのレジストリとこのレジストリを操作するためのいくつかの便宜的なメソッドを提供します。`Charset` インスタンスは `email` パッケージ中にあるほかのいくつかのモジュールで使用されます。

2.2.2 で追加された仕様です。

`class Charset([input_charset])`

文字セットを email のプロパティに写像する。Map character sets to their email properties.

このクラスはある特定の文字セットに対し、電子メールに課される制約の情報を提供します。また、与えられた適用可能な codec をつかって、文字セット間の変換をおこなう便宜的なルーチンも提供

します。またこれは、ある文字セットが与えられたときに、その文字セットを電子メールメッセージのなかでどうやって RFC に準拠したやり方で使用するかに関する、できうるかぎりの情報も提供します。

文字セットによっては、それらの文字を電子メールのヘッダあるいはメッセージ本体で使う場合は quoted-printable 形式あるいは base64 形式でエンコードする必要があります。またある文字セットはむきだしのまま変換する必要があり、電子メールの中では使用できません。

以下ではオプション引数 `input_charset` について説明します。この値はつねに小文字に強制的に変換されます。そして文字セットの別名が正規化されたあと、この値は文字セットのレジストリ内を検索し、ヘッダのエンコーディングとメッセージ本体のエンコーディング、および出力時の変換に使われる codec をみつけるのに使われます。たとえば `input_charset` が `iso-8859-1` の場合、ヘッダおよびメッセージ本体は quoted-printable でエンコードされ、出力時の変換用 codec は必要ありません。もし `input_charset` が `eur-jp` ならば、ヘッダは base64 でエンコードされ、メッセージ本体はエンコードされませんが、出力されるテキストは `eur-jp` 文字セットから `iso-2022-jp` 文字セットに変換されます。

Charset インスタンスは以下のようなデータ属性をもっています:

#### `input_charset`

最初に指定される文字セットです。一般に通用している別名は、正式な電子メール用の名前に変換されます (たとえば、`latin_1` は `iso-8859-1` に変換されます)。デフォルトは 7-bit の `us-ascii` です。

#### `header_encoding`

この文字セットが電子メールヘッダに使われる前にエンコードされる必要がある場合、この属性は `Charset.QP` (quoted-printable エンコーディング)、`Charset.BASE64` (base64 エンコーディング)、あるいは最短の QP または BASE64 エンコーディングである `Charset.SHORTEST` に設定されます。そうでない場合、この値は `None` になります。

#### `body_encoding`

`header_encoding` と同じですが、この値はメッセージ本体のためのエンコーディングを記述します。これはヘッダ用のエンコーディングとは違うかもしれません。`body_encoding` では、`Charset.SHORTEST` を使うことはできません。

#### `output_charset`

文字セットによっては、電子メールのヘッダあるいはメッセージ本体に使う前にそれを変換する必要があります。もし `input_charset` がそれらの文字セットのどれかをさしていたら、この `output_charset` 属性はそれが出力時に変換される文字セットの名前をあらわしています。それ以外の場合、この値は `None` になります。

#### `input_codec`

`input_charset` を Unicode に変換するための Python 用 codec 名です。変換用の codec が必要ないときは、この値は `None` になります。

#### `output_codec`

Unicode を `output_charset` に変換するための Python 用 codec 名です。変換用の codec が必要ないときは、この値は `None` になります。この属性は `input_codec` と同じ値をもつことになるでしょう。

Charset インスタンスは、以下のメソッドも持っています:

#### `get_body_encoding()`

メッセージ本体のエンコードに使われる `content-transfer-encoding` の値を返します。

この値は使用しているエンコーディングの文字列 `'quoted-printable'` または `'base64'` か、あるいは関数のどちらかです。後者の場合、これはエンコードされる Message オブジェクトを単一の

引数として取るような関数である必要があります。この関数は変換後 Content-Transfer-Encoding: ヘッダ自体を、なんであれ適切な値に設定する必要があります。

このメソッドは *body\_encoding* が QP の場合 ‘quoted-printable’ を返し、*body\_encoding* が BASE64 の場合 ‘base64’ を返します。それ以外の場合は文字列 ‘7bit’ を返します。

**convert(*s*)**

文字列 *s* を *input\_codec* から *output\_codec* に変換します。

**toSplittable(*s*)**

おそらくマルチバイトの文字列を、安全に split できる形式に変換します。*s* には split する文字列を渡します。

これは *input\_codec* を使って文字列を Unicode にすることで、文字と文字の境界で (たとえそれがマルチバイト文字であっても) 安全に split できるようにします。

*input\_charset* の文字列 *s* をどうやって Unicode に変換すればいいかが不明な場合、このメソッドは与えられた文字列そのものを返します。

Unicode に変換できなかった文字は、Unicode 置換文字 (Unicode replacement character) ‘U+FFFD’ に置換されます。

**fromSplittable(*ustr*[, *to\_output*])**

split できる文字列をエンコードされた文字列に変換しなおします。*ustr* は “逆 split” するための Unicode 文字列です。

このメソッドでは、文字列を Unicode からべつのエンコード形式に変換するために適切な codec を使用します。与えられた文字列が Unicode ではなかった場合、あるいはそれをどうやって Unicode から変換するか不明だった場合は、与えられた文字列そのものが返されます。

Unicode から正しく変換できなかった文字については、適当な文字 (通常は ‘?’) に置き換えられます。

*to\_output* が True の場合 (デフォルト)、このメソッドは *output\_codec* をエンコードの形式として使用します。*to\_output* が False の場合、これは *input\_codec* を使用します。

**getOutputCharset()**

出力用の文字セットを返します。

これは *output\_charset* 属性が None でなければその値になります。それ以外の場合、この値は *input\_charset* と同じです。

**encodedHeaderLen()**

エンコードされたヘッダ文字列の長さを返します。これは quoted-printable エンコーディングあるいは base64 エンコーディングに対しても正しく計算されます。

**headerEncode(*s*[, *convert*])**

文字列 *s* をヘッダ用にエンコードします。

*convert* が True の場合、文字列は入力用文字セットから出力用文字セットに自動的に変換されます。これは行の長さ問題のあるマルチバイトの文字セットに対しては役に立ちません (マルチバイト文字はバイト境界ではなく、文字ごとの境界で split する必要があります)。これらの問題を扱うには、高水準のクラスである Header クラスを使ってください (email.Header を参照)。*convert* の値はデフォルトでは False です。

エンコーディングの形式 (base64 または quoted-printable) は、*header\_encoding* 属性に基づきます。

**bodyEncode(*s*[, *convert*])**

文字列 *s* をメッセージ本体用にエンコードします。

*convert* が True の場合 (デフォルト)、文字列は入力用文字セットから出力用文字セットに自動的に変換されます。headerEncode() とは異なり、メッセージ本体にはふつうバイト境界の問題やマ

ルチバイト文字セットの問題がないので、これはきわめて安全におこなえます。

エンコーディングの形式 (base64 または quoted-printable) は、*body\_encoding* 属性に基づきます。

Charset クラスには、標準的な演算と組み込み関数をサポートするいくつかのメソッドがあります。

`__str__()`

*input\_charset* を小文字に変換された文字列型として返します。`__repr__()` は、`__str__()` の別名となっています。

`__eq__(other)`

このメソッドは、2 つの Charset インスタンスが同じかどうかをチェックするのに使います。

`__ne__(other)`

このメソッドは、2 つの Charset インスタンスが異なるかどうかをチェックするのに使います。

また、`email.Charset` モジュールには、グローバルな文字セット、文字セットの別名 (エイリアス) および codec 用のレジストリに新しいエントリを追加する以下の関数もふくまれています:

`add_charset(charset[, header_enc[, body_enc[, output_charset]])`

文字の属性をグローバルなレジストリに追加します。

*charset* は入力用の文字セットで、その文字セットの正式名称を指定する必要があります。

オプション引数 *header\_enc* および *body\_enc* は quoted-printable エンコーディングをあらわす `Charset.QP` か、base64 エンコーディングをあらわす `Charset.BASE64`、最短の quoted-printable または base64 エンコーディングをあらわす `Charset.SHORTEST`、あるいはエンコーディングなしの `None` のどれかになります。`SHORTEST` が使えるのは *header\_enc* だけです。デフォルトの値はエンコーディングなしの `None` になっています。

オプション引数 *output\_charset* には出力用の文字セットが入ります。`Charset.convert()` が呼ばれたときの変換はまず入力用の文字セットを Unicode に変換し、それから出力用の文字セットに変換されます。デフォルトでは、出力は入力と同じ文字セットになっています。

*input\_charset* および *output\_charset* はこのモジュール中の文字セット-codec 対応表にある Unicode codec エントリである必要があります。モジュールがまだ対応していない codec を追加するには、`add_codec()` を使ってください。より詳しい情報については `codecs` モジュールの文書を参照してください。

グローバルな文字セット用のレジストリは、モジュールの global 辞書 `CHARSETS` 内に保持されています。

`add_alias(alias, canonical)`

文字セットの別名 (エイリアス) を追加します。*alias* はその別名で、たとえば `latin-1` のように指定します。*canonical* はその文字セットの正式名称で、たとえば `iso-8859-1` のように指定します。文字セットのグローバルな別名用レジストリは、モジュールの global 辞書 `ALIASES` 内に保持されています。

`add_codec(charset, codecname)`

与えられた文字セットの文字と Unicode との変換をおこなう codec を追加します。

*charset* はある文字セットの正式名称で、*codecname* は Python 用 codec の名前です。これは組み込み関数 `unicode()` の第 2 引数か、あるいは Unicode 文字列型の `encode()` メソッドに適した形式になっていなければなりません。

## 12.2.7 エンコーダ

何もないところから `Message` を作成するときしばしば必要になるのが、ペイロードをメールサーバに通すためにエンコードすることです。これはとくにバイナリデータを含んだ `image/*` や `text/*` タイプのメッセー



ジが必要です。

email パッケージでは、Encoders モジュールにおいていくつかの便宜的なエンコーディングをサポートしています。実際にはこれらのエンコーダは `MIMEImage` および `MIMEText` クラスのコンストラクタでデフォルトエンコーダとして使われています。すべてのエンコーディング関数は、エンコードするメッセージオブジェクトひとつだけを引数にとります。これらはふつうペイロードを取りだし、それをエンコードして、ペイロードをエンコードされたものにセットしなおします。これらはまた Content-Transfer-Encoding: ヘッダを適切な値に設定します。

提供されているエンコーディング関数は以下のとおりです：

`encode_quopri(msg)`

ペイロードを quoted-printable 形式にエンコードし、Content-Transfer-Encoding: ヘッダを quoted-printable<sup>4</sup> に設定します。これはそのペイロードのほとんどが通常の印刷可能な文字からなっているが、印刷不可能な文字がすこしだけあるときのエンコード方法として適しています。

`encode_base64(msg)`

ペイロードを base64 形式でエンコードし、Content-Transfer-Encoding: ヘッダを base64 に変更します。これはペイロード中のデータのほとんどが印刷不可能な文字である場合に適しています。quoted-printable 形式よりも結果としてはコンパクトなサイズになるからです。base64 形式の欠点は、これが人間にはまったく読めないテキストになってしまうことです。

`encode_7or8bit(msg)`

これは実際にはペイロードを変更はしませんが、ペイロードの形式に応じて Content-Transfer-Encoding: ヘッダを 7bit あるいは 8bit に適した形に設定します。

`encode_noop(msg)`

これは何もしないエンコーダです。Content-Transfer-Encoding: ヘッダを設定さえしません。

## 12.2.8 例外クラス

何も無いところから `Message` を作成するときしばしば必要になるのが、ペイロードをメールサーバに通すためにエンコードすることです。これはとくにバイナリデータを含んだ `image/*` や `text/*` タイプのメッセージが必要です。

email パッケージでは、Encoders モジュールにおいていくつかの便宜的なエンコーディングをサポートしています。実際にはこれらのエンコーダは `MIMEImage` および `MIMEText` クラスのコンストラクタでデフォルトエンコーダとして使われています。すべてのエンコーディング関数は、エンコードするメッセージオブジェクトひとつだけを引数にとります。これらはふつうペイロードを取りだし、それをエンコードして、ペイロードをエンコードされたものにセットしなおします。これらはまた Content-Transfer-Encoding: ヘッダを適切な値に設定します。

提供されているエンコーディング関数は以下のとおりです：

`encode_quopri(msg)`

ペイロードを quoted-printable 形式にエンコードし、Content-Transfer-Encoding: ヘッダを quoted-printable<sup>5</sup> に設定します。これはそのペイロードのほとんどが通常の印刷可能な文字からなっているが、印刷不可能な文字がすこしだけあるときのエンコード方法として適しています。

`encode_base64(msg)`

ペイロードを base64 形式でエンコードし、Content-Transfer-Encoding: ヘッダを base64 に変更します。これはペイロード中のデータのほとんどが印刷不可能な文字である場合に適しています。quoted-printable 形式よりも結果としてはコンパクトなサイズになるからです。base64 形式の欠点は、これが

<sup>4</sup>注意: `encode_quopri()` を使ってエンコードすると、データ中のタブ文字や空白文字もエンコードされます。

<sup>5</sup>注意: `encode_quopri()` を使ってエンコードすると、データ中のタブ文字や空白文字もエンコードされます。



人間にはまったく読めないテキストになってしまうことです。

`encode_7or8bit(msg)`

これは実際にはペイロードを変更はしませんが、ペイロードの形式に応じて Content-Transfer-Encoding: ヘッダを 7bit あるいは 8bit に適した形に設定します。

`encode_noop(msg)`

これは何もしないエンコーダです。Content-Transfer-Encoding: ヘッダを設定さえしません。

## 12.2.9 雑用ユーティリティ

email ではいくつかの便利なユーティリティを提供しています。

`quote(str)`

文字列 *str* 内のバックスラッシュを バックスラッシュ2つ に置換した新しい文字列を返します。また、ダブルクォートは バックスラッシュ + ダブルクォートに置換されます。

`unquote(str)`

文字列 *str* を 逆クォートした新しい文字列を返します。もし *str* の先頭あるいは末尾がダブルクォートだった場合、これらは単に切り落とされます。同様にもし *str* の先頭あるいは末尾が角ブラケット (<, >) だった場合も切り落とされます。

`parseaddr(address)`

アドレスをパースします。To: や Cc: のようなアドレスをふくんだフィールドの値を与えると、構成部分の実名と電子メールアドレスを取り出します。パースに成功した場合、これらの情報をタプル (realname, email\_address) にして返します。失敗した場合は 2 要素のタプル ("", "") を返します。

`formataddr(pair)`

`parseaddr()` の逆で、実名と電子メールアドレスからなる 2 要素のタプル (realname, email\_address) を引数にとり、To: あるいは Cc: ヘッダに適した形式の文字列を返します。タプル *pair* の第 1 要素が偽である場合、第 2 要素の値をそのまま返します。

`getaddresses(fieldvalues)`

このメソッドは 2 要素タプルのリストを `parseaddr()` と同じ形式で返します。*fieldvalues* はたとえば `Message.get_all()` が返すような、ヘッダのフィールド値からなるシーケンスです。以下はある電子メールメッセージからすべての受け取り人を得る一例です:

```
from email.Utils import getaddresses

tos = msg.get_all('to', [])
ccs = msg.get_all('cc', [])
resent_tos = msg.get_all('resent-to', [])
resent_ccs = msg.get_all('resent-cc', [])
all_recipients = getaddresses(tos + ccs + resent_tos + resent_ccs)
```

`parsedate(date)`

RFC 2822 に記された規則にもとづいて日付を解析します。しかし、メイラーによってはここで指定された規則に従っていないものがあり、そのような場合 `parsedate()` はなるべく正しい日付を推測しようとします。*date* は RFC 2822 形式の日付を保持している文字列で、"Mon, 20 Nov 1995 19:12:08 -0500" のような形をしています。日付の解析に成功した場合、`parsedate()` は関数 `time.mktime()` に直接渡せる形式の 9 要素からなるタプルを返し、失敗した場合は `None` を返します。返されるタプルの 6、7、8 番目のフィールドは有効ではないので注意してください。

`parsedate_tz(date)`

`parsedate()` と同様の機能を提供しますが、`None` または 10 要素のタプルを返すところが違いま

す。最初の 9 つの要素は `time.mktime()` に直接渡せる形式のものであり、最後の 10 番目の要素は、その日付の時間帯の UTC (グリニッジ標準時の公式な呼び名です) に対するオフセットです<sup>6</sup>。入力された文字列に時間帯が指定されていなかった場合、10 番目の要素には `None` が入ります。タプルの 6、7、8 番目のフィールドは有効ではないので注意してください。

**mktime\_tz**(*tuple*)

`parsedate_tz()` が返す 10 要素のタプルを UTC のタイムスタンプに変換します。与えられた時間帯が `None` である場合、時間帯として現地時間 (`localtime`) が仮定されます。マイナーな欠点: `mktime_tz()` はまず *tuple* の最初の 8 要素を `localtime` として変換し、つぎに時間帯の差を加味しています。夏時間を使っている場合には、これは通常の使用にはさしかえられないものの、わずかな誤差を生じるかもしれません。

**formatdate**([*timeval* [, *localtime* ]])

日付を RFC 2822 形式の文字列で返します。例:

```
Fri, 09 Nov 2001 01:08:47 -0000
```

オプションとして `float` 型の値をもつ引数 *timeval* が与えられた場合、これは `time.gmtime()` および `time.localtime()` に渡されます。それ以外の場合、現在の時刻が使われます。

オプション引数 *localtime* はフラグです。これが `True` の場合、この関数は *timeval* を解析したあと UTC のかわりに現地時間 (`localtime`) の時間帯をつかって変換します。おそらく夏時間も考慮に入れられるでしょう。デフォルトではこの値は `False` で、UTC が使われます。

**make\_msgid**([*idstring*])

RFC 2822 準拠形式の Message-ID: ヘッダに適した文字列を返します。オプション引数 *idstring* が文字列として与えられた場合、これはメッセージ ID の一意性を高めるのに利用されます。

**decode\_rfc2231**(*s*)

RFC 2231 に従って文字列 *s* をデコードします。

**encode\_rfc2231**(*s* [, *charset* [, *language* ]])

RFC 2231 に従って *s* をエンコードします。オプション引数 *charset* および *language* が与えられた場合、これらは文字セット名と言語名として使われます。もしこれらのどちらも与えられていない場合、*s* はそのまま返されます。*charset* は与えられているが *language* が与えられていない場合、文字列 *s* は *language* の空文字列を使ってエンコードされます。

**decode\_params**(*params*)

RFC 2231 に従ってパラメータのリストをデコードします。*params* は (`content-type`, `string-value`) のような形式の 2 要素からなるタプルです。

以下の関数はもはや時代遅れとなっており、推奨されません:

**dump\_address\_pair**(*pair*)

リリース 2.2.2 以降で撤廃された仕様です。これのかわりに `formataddr()` メソッドを使ってください。

**decode**(*s*)

リリース 2.2.2 以降で撤廃された仕様です。これのかわりに `Header.decode_header()` メソッドを使ってください。

**encode**(*s* [, *charset* [, *encoding* ]])

リリース 2.2.2 以降で撤廃された仕様です。これのかわりに `Header.encode()` メソッドを使ってください。

---

<sup>6</sup>注意: この時間帯のオフセット値は `time.timezone` の値と符号が逆です。これは `time.timezone` が POSIX 標準に準拠しているのに対して、こちらは RFC 2822 に準拠しているからです。

## 12.2.10 イテレータ

`Message.walk()` メソッドを使うと、簡単にメッセージオブジェクトツリー内を次から次へとたどる (iteration) ことができます。 `email.Iterators` モジュールはこのための高水準イテレータをいくつか提供します。

`body_line_iterator(msg[, decode])`

このイテレータは `msg` 中のすべてのサブパートに含まれるペイロードをすべて順にたどっていき、ペイロード内の文字列を 1 行ずつ返します。サブパートのヘッダはすべて無視され、Python 文字列でないペイロードからなるサブパートも無視されます。これは `readline()` を使って、ファイルからメッセージを (ヘッダだけとばして) フラットなテキストとして読むのにいくぶん似ているかもしれませんが。

オプション引数 `decode` は、`Message.get_payload()` にそのまま渡されます。

`typed_subpart_iterator(msg[, maintype[, subtype]])`

このイテレータは `msg` 中のすべてのサブパートをたどり、それらの中で指定された MIME 形式 `maintype` と `subtype` をもつようなパートのみを返します。

`subtype` は省略可能であることに注意してください。これが省略された場合、サブパートの MIME 形式は `maintype` のみがチェックされます。じつは `maintype` も省略可能で、その場合にはデフォルトは `text` です。

つまり、デフォルトでは `typed_subpart_iterator()` は MIME 形式 `text/*` をもつサブパートを順に返していくというわけです。

以下の関数は役に立つデバッグ用ツールとして追加されたもので、パッケージとして公式なサポートのあるインターフェイスではありません。

`_structure(msg[, fp[, level]])`

そのメッセージオブジェクト構造の content-type をインデントつきで表示します。たとえば:

```
>>> msg = email.message_from_file(somefile)
>>> _structure(msg)
multipart/mixed
  text/plain
  text/plain
  multipart/digest
    message/rfc822
      text/plain
    message/rfc822
      text/plain
    message/rfc822
      text/plain
    message/rfc822
      text/plain
    message/rfc822
      text/plain
    message/rfc822
      text/plain
  text/plain
```

オプション引数 `fp` は出力を渡すためのストリーム<sup>7</sup>オブジェクトです。これは Python の拡張 `print` 文が対応できるようになっている必要があります。 `level` は内部的に使用されます。

---

<sup>7</sup>訳注: 原文では file-like。

### 12.2.11 (Python 2.2.1 までの) email v1 とのちがい

email パッケージ バージョン 1 は、Python 2.2.1 リリースのときまでバンドルされていました。バージョン 2 は Python 2.3 リリース用に開発され、Python 2.2.2 にバックポートされたものです。またこれは distutils ベースの別個のパッケージとしても配布されています。email バージョン 2 はバージョン 1 とほとんどの点で下位互換ですが、以下のような違いがあります:

- email.Header モジュールおよび email.Charset モジュールが追加されています。
- Message インスタンスの Pickle 形式が変わりました。が、これは正式に定義されたことは一度もないので(そしてこれからも)、この変更は互換性の欠如とはみなされていません。ですがもしお使いのアプリケーションが Message インスタンスを pickle あるいは unpickle しているなら、現在 email バージョン 2 ではプライベート変数 `_charset` および `_default_type` を含むようになったということに注意してください。
- Message クラス中のいくつかのメソッドは推奨されなくなったか、あるいは呼び出し形式が変更になっています。また、多くの新しいメソッドが追加されています。詳しくは Message クラスの文書を参照してください。これらの変更は完全に下位互換になっているはずですが。
- message/rfc822 形式のコンテナは、見た目上のオブジェクト構造が変わりました。email バージョン 1 ではこの content type はスカラー形式のペイロードとして表現されていました。つまり、コンテナメッセージの `is_multipart()` は `false` を返し、`get_payload()` はリストオブジェクトではなく単一の Message インスタンスを直接返すようになっていたのです。

この構造はパッケージ中のほかの部分と整合がとれていなかったため、message/rfc822 形式のオブジェクト表現形式が変更されました。email バージョン 2 では、コンテナは `is_multipart()` に `True` を返します。また `get_payload()` はひとつの Message インスタンスを要素とするリストを返すようになりました。

注意: ここは下位互換が完全には成りたたなくなっている部分のひとつです。けれどもあらかじめ `get_payload()` が返すタイプをチェックするようになっていれば問題にはなりません。ただ message/rfc822 形式のコンテナを Message インスタンスにじかに `set_payload()` しないようにさえすればよいのです。

- Parser コンストラクタに `strict` 引数が追加され、`parse()` および `parsestr()` メソッドには `headersonly` 引数がつきました。`strict` フラグはまた `email.message_from_file()` と `email.message_from_string()` にも追加されています。
- `Generator.__call__()` はもはや推奨されなくなりました。かわりに `Generator.flatten()` を使ってください。また、Generator クラスには `clone()` メソッドが追加されています。
- email.Generator モジュールに DecodedGenerator クラスが加わりました。
- 中間的な基底クラスである MIMENonMultipart および MIMEMultipart がクラス階層の中に追加され、ほとんどの MIME 関係の派生クラスがこれを介するようになっています。
- MIMEText コンストラクタの `_encoder` 引数は推奨されなくなりました。いまやエンコーダは `_charset` 引数にもとづいて暗黙のうちに決定されます。
- email.Utils モジュールにおける以下の関数は推奨されなくなりました: `dump_address_pairs()`、`decode()`、および `encode()`。また、このモジュールには以下の関数が追加されています: `make_msgid()`、`decode_rfc2231()`、`encode_rfc2231()` そして `decode_params()`。
- Public ではない関数 `email.Iterators._structure()` が追加されました。

## 12.2.12 mimelib との違い

email パッケージはもともと mimelib と呼ばれる個別のライブラリからつくられたものです。その後変更が加えられ、メソッド名がより一貫したものになり、いくつかのメソッドやモジュールが加えられたりはずされたりしました。いくつかのメソッドでは、その意味も変更されています。しかしほとんどの部分において、mimelib パッケージで使うことのできた機能は、ときどきその方法が変わってはいるものの email パッケージでも使用可能です。mimelib パッケージと email パッケージの間の下位互換性はあまり優先はされませんでした。

以下では mimelib パッケージと email パッケージにおける違いを簡単に説明し、それに沿ってアプリケーションを移植するさいの指針を述べています。

おそらく 2 つのパッケージのもっとも明らかな違いは、パッケージ名が email に変更されたことです。さらにトップレベルのパッケージが以下のように変更されました:

- `messageFromString()` は `message_from_string()` に名前が変更されました。
- `messageFromFile()` は `message_from_file()` に名前が変更されました。

Message クラスでは、以下のような違いがあります:

- `asString()` メソッドは `as_string()` に名前が変更されました。
- `ismultipart()` メソッドは `is_multipart()` に名前が変更されました。
- `get_payload()` メソッドはオプション引数として *decode* をとるようになりました。
- `getall()` メソッドは `get_all()` に名前が変更されました。
- `addheader()` メソッドは `add_header()` に名前が変更されました。
- `gettype()` メソッドは `get_type()` に名前が変更されました。
- `getmaintype()` メソッドは `get_main_type()` に名前が変更されました。
- `getsubtype()` メソッドは `get_subtype()` に名前が変更されました。
- `getparams()` メソッドは `get_params()` に名前が変更されました。また、従来の `getparams()` は文字列のリストを返していましたが、`get_params()` は 2-タプルのリストを返すようになります。これはそのパラメータのキーと値の組が、`'='` 記号によって分離されたものです。
- `getparam()` メソッドは `get_param()`。
- `getcharsets()` メソッドは `get_charsets()` に名前が変更されました。
- `getfilename()` メソッドは `get_filename()` に名前が変更されました。
- `getboundary()` メソッドは `get_boundary()` に名前が変更されました。
- `setboundary()` メソッドは `set_boundary()` に名前が変更されました。
- `getdecodedpayload()` メソッドは廃止されました。これと同様の機能は `get_payload()` メソッドの *decode* フラグに 1 を渡すことで実現できます。
- `getpayloadastext()` メソッドは廃止されました。これと同様の機能は `email.Generator` モジュールの `DecodedGenerator` クラスによって提供されます。



- `getbodyastext()` メソッドは廃止されました。これと同様の機能は `email.Iterators` モジュールにある `typed_subpart_iterator()` を使ってイテレータを作ることにより実現できます。

`Parser` クラスは、その `public` なインターフェイスは変わっていませんが、これはより一層かしこくなって `message/delivery-status` 形式のメッセージを認識するようになりました。これは配送状態通知<sup>8</sup>において、各ヘッダブロックを表す独立した `Message` パートを含むひとつの `Message` インスタンスとして表現されます。

`Generator` クラスは、その `public` なインターフェイスは変わっていませんが、`email.Generator` モジュールに新しいクラスが加わりました。`DecodedGenerator` と呼ばれるこのクラスは以前 `Message.getpayloadastext()` メソッドで使われていた機能のほとんどを提供します。

また、以下のモジュールおよびクラスが変更されています：

- `MIMEBase` クラスのコンストラクタ引数 `_major` と `_minor` は、それぞれ `_maintype` と `_subtype` に変更されています。
- `Image` クラスおよびモジュールは `MIMEImage` に名前が変更されました。`_minor` 引数も `_subtype` に名前が変更されています。
- `Text` クラスおよびモジュールは `MIMEText` に名前が変更されました。`_minor` 引数も `_subtype` に名前が変更されています。
- `MessageRFC822` クラスおよびモジュールは `MIMEMessage` に名前が変更されました。注意：従来バージョンの `mimelib` では、このクラスおよびモジュールは `RFC822` という名前でしたが、これは大文字小文字を区別しないファイルシステムでは Python の標準ライブラリモジュール `rfc822` と名前がかち合っていました。

また、`MIMEMessage` クラスはいまや `message main type` をもつあらゆる種類の MIME メッセージを表現できるようになりました。これはオプション引数として、MIME subtype を指定する `_subtype` 引数をとることができるようになっています。デフォルトでは、`_subtype` は `rfc822` になります。

`mimelib` では、`address` および `date` モジュールでいくつかのユーティリティ関数が提供されていました。これらの関数はすべて `email.Utils` モジュールの中に移されています。

`MsgReader` クラスおよびモジュールは廃止されました。これにもっとも近い機能は `email.Iterators` モジュール中の `body_line_iterator()` 関数によって提供されています。

### 12.2.13 使用例

ここでは `email` パッケージを使って電子メールメッセージを読む・書く・送信するいくつかの例を紹介します。より複雑な MIME メッセージについても扱います。

最初に、テキスト形式の単純なメッセージを作成・送信する方法です：

```
# Import smtplib for the actual sending function
import smtplib

# Import the email modules we'll need
from email.MIMEText import MIMEText

# Open a plain text file for reading. For this example, assume that
# the text file contains only ASCII characters.
fp = open(textfile, 'rb')
# Create a text/plain message
```

<sup>8</sup>配送状態通知 (Delivery Status Notifications, DSN) は RFC 1894 によって定義されています。



```

msg = MIMEText(fp.read())
fp.close()

# me == the sender's email address
# you == the recipient's email address
msg['Subject'] = 'The contents of %s' % textfile
msg['From'] = me
msg['To'] = you

# Send the message via our own SMTP server, but don't include the
# envelope header.
s = smtplib.SMTP()
s.connect()
s.sendmail(me, [you], msg.as_string())
s.close()

```

つぎに、あるディレクトリ内にある何枚かの家族写真をひとつの MIME メッセージに収めて送信する例です:

```

# Import smtplib for the actual sending function
import smtplib

# Here are the email package modules we'll need
from email.MIMEImage import MIMEImage
from email.MIMEMultipart import MIMEMultipart

COMMASPACE = ', '

# Create the container (outer) email message.
msg = MIMEMultipart()
msg['Subject'] = 'Our family reunion'
# me == the sender's email address
# family = the list of all recipients' email addresses
msg['From'] = me
msg['To'] = COMMASPACE.join(family)
msg.preamble = 'Our family reunion'
# Guarantees the message ends in a newline
msg.epilogue = ''

# Assume we know that the image files are all in PNG format
for file in pngfiles:
    # Open the files in binary mode. Let the MIMEImage class automatically
    # guess the specific image type.
    fp = open(file, 'rb')
    img = MIMEImage(fp.read())
    fp.close()
    msg.attach(img)

# Send the email via our own SMTP server.
s = smtplib.SMTP()
s.connect()
s.sendmail(me, family, msg.as_string())
s.close()

```

つぎはあるディレクトリに含まれている内容全体をひとつの電子メールメッセージとして送信するやり方です<sup>9</sup>:

```

#!/usr/bin/env python

"""Send the contents of a directory as a MIME message.

Usage: dirmail [options] from to [to ...]*

```

---

<sup>9</sup>最初の思いつきと用例は Matthew Dixon Cowles のおかげです。

Options:

```
-h / --help
    Print this message and exit.
```

```
-d directory
--directory=directory
    Mail the contents of the specified directory, otherwise use the
    current directory. Only the regular files in the directory are sent,
    and we don't recurse to subdirectories.
```

'from' is the email address of the sender of the message.

'to' is the email address of the recipient of the message, and multiple recipients may be given.

The email is sent by forwarding to your local SMTP server, which then does the normal delivery process. Your local machine must be running an SMTP server.

```
"""
import sys
import os
import getopt
import smtplib
# For guessing MIME type based on file name extension
import mimetypes

from email import Encoders
from email.Message import Message
from email.MIMEAudio import MIMEAudio
from email.MIMEBase import MIMEBase
from email.MIMEMultipart import MIMEMultipart
from email.MIMEImage import MIMEImage
from email.MIMEText import MIMEText

COMMASPACE = ', '

def usage(code, msg=''):
    print >> sys.stderr, __doc__
    if msg:
        print >> sys.stderr, msg
    sys.exit(code)

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], 'hd:', ['help', 'directory='])
    except getopt.error, msg:
        usage(1, msg)

    dir = os.getcwd()
    for opt, arg in opts:
        if opt in ('-h', '--help'):
            usage(0)
        elif opt in ('-d', '--directory'):
            dir = arg

    if len(args) < 2:
        usage(1)

    sender = args[0]
    recips = args[1:]
```

```

# Create the enclosing (outer) message
outer = MIMEMultipart()
outer['Subject'] = 'Contents of directory %s' % os.path.abspath(dir)
outer['To'] = COMMASPACE.join(recips)
outer['From'] = sender
outer.preamble = 'You will not see this in a MIME-aware mail reader.\n'
# To guarantee the message ends with a newline
outer.epilogue = ''

for filename in os.listdir(dir):
    path = os.path.join(dir, filename)
    if not os.path.isfile(path):
        continue
    # Guess the content type based on the file's extension. Encoding
    # will be ignored, although we should check for simple things like
    # gzip'd or compressed files.
    ctype, encoding = mimetypes.guess_type(path)
    if ctype is None or encoding is not None:
        # No guess could be made, or the file is encoded (compressed), so
        # use a generic bag-of-bits type.
        ctype = 'application/octet-stream'
    maintype, subtype = ctype.split('/', 1)
    if maintype == 'text':
        fp = open(path)
        # Note: we should handle calculating the charset
        msg = MIMEText(fp.read(), _subtype=subtype)
        fp.close()
    elif maintype == 'image':
        fp = open(path, 'rb')
        msg = MIMEImage(fp.read(), _subtype=subtype)
        fp.close()
    elif maintype == 'audio':
        fp = open(path, 'rb')
        msg = MIMEAudio(fp.read(), _subtype=subtype)
        fp.close()
    else:
        fp = open(path, 'rb')
        msg = MIMEBase(maintype, subtype)
        msg.set_payload(fp.read())
        fp.close()
        # Encode the payload using Base64
        Encoders.encode_base64(msg)
    # Set the filename parameter
    msg.add_header('Content-Disposition', 'attachment', filename=filename)
    outer.attach(msg)

# Now send the message
s = smtplib.SMTP()
s.connect()
s.sendmail(sender, recips, outer.as_string())
s.close()

if __name__ == '__main__':
    main()

```

そして最後に、上のような MIME メッセージをどうやって展開してひとつのディレクトリ上の複数ファイルにするかを示します:

```

#!/usr/bin/env python

"""Unpack a MIME message into a directory of files.

```

Usage: unpackmail [options] msgfile

Options:

```
-h / --help
    Print this message and exit.

-d directory
--directory=directory
    Unpack the MIME message into the named directory, which will be
    created if it doesn't already exist.
```

msgfile is the path to the file containing the MIME message.

"""

```
import sys
import os
import getopt
import errno
import mimetypes
import email

def usage(code, msg=''):
    print >> sys.stderr, __doc__
    if msg:
        print >> sys.stderr, msg
    sys.exit(code)

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], 'hd:', ['help', 'directory='])
    except getopt.error, msg:
        usage(1, msg)

    dir = os.curdir
    for opt, arg in opts:
        if opt in ('-h', '--help'):
            usage(0)
        elif opt in ('-d', '--directory'):
            dir = arg

    try:
        msgfile = args[0]
    except IndexError:
        usage(1)

    try:
        os.mkdir(dir)
    except OSError, e:
        # Ignore directory exists error
        if e.errno <> errno.EEXIST: raise

    fp = open(msgfile)
    msg = email.message_from_file(fp)
    fp.close()

    counter = 1
    for part in msg.walk():
        # multipart/* are just containers
        if part.get_content_maintype() == 'multipart':
            continue
        # Applications should really sanitize the given filename so that an
```

```

# email message can't be used to overwrite important files
filename = part.get_filename()
if not filename:
    ext = mimetypes.guess_extension(part.get_type())
    if not ext:
        # Use a generic bag-of-bits extension
        ext = '.bin'
    filename = 'part-%03d%s' % (counter, ext)
    counter += 1
    fp = open(os.path.join(dir, filename), 'wb')
    fp.write(part.get_payload(decode=1))
    fp.close()

if __name__ == '__main__':
    main()

```

## 12.3 mailcap — mailcap ファイルの操作

mailcap ファイルは、メールリーダーや Web ブラウザのような MIME 対応のアプリケーションが、異なる MIME タイプのファイルにどのように反応するかを設定するために使われます (“mailcap” の名前は “mail capability” から取られました)。例えば、ある mailcap ファイルに ‘video/mpeg; xmpeg %s’ のような行が入っていたとします。ユーザが email メッセージや Web ドキュメント上でその MIME タイプ video/mpeg に遭遇すると、‘%s’ はファイル名 (通常テンポラリファイルに属するものになります) に置き換えられ、ファイルを閲覧するために xmpeg プログラムが自動的に起動されます。

mailcap の形式は RFC 1524, “A User Agent Configuration Mechanism For Multimedia Mail Format Information” で文書化されていますが、この文書はインターネット標準ではありません。しかしながら、mailcap ファイルはほとんどの UNIX システムでサポートされています。

**findmatch**(*caps*, *MIMEtype*[, *key*[, *filename*[, *plist*]]])

2 要素のタプルを返します; 最初の要素は文字列で、実行すべきコマンド (os.system() に渡されます) が入っています。二つめの要素は与えられた MIME タイプに対する mailcap エントリです。一致する MIME タイプが見つからなかった場合、(None, None) が返されます。

*key* は desired フィールドの値で、実行すべき動作のタイプを表現します; ほとんどの場合、単に MIME 形式のデータ本体を見たいと思うので、標準の値は ‘view’ になっています。与えられた MIME 型をもつ新たなデータ本体を作成した場合や、既存のデータ本体を置き換えたい場合には、‘view’ の他に ‘compose’ および ‘edit’ を取ることもできます。

これらフィールドの完全なリストについては RFC 1524 を参照してください。

*filename* はコマンドライン中で ‘%s’ に代入されるファイル名です; 標準の値は ‘/dev/null’ で、たいていこの値を使いたいわけではないはずです。従って、ファイル名を指定してこのフィールドを上書きする必要があるでしょう。

*plist* は名前付けされたパラメタのリストです; 標準の値は単なる空のリストです。リスト中の各エントリはパラメタ名を含む文字列、等号 (=)、およびパラメタの値でなければなりません。mailcap エントリには %*{foo}* といったような名前付きのパラメタを含めることができ、‘foo’ と名づけられたパラメタの値に置き換えられます。例えば、コマンドライン ‘showpartial %*{id}* %*{number}* %*{total}*’ が mailcap ファイルにあり、*plist* が [‘id=1’, ‘number=2’, ‘total=3’] に設定されていれば、コマンドラインは ‘showpartial 1 2 3’ になります。

mailcap ファイル中では、オプションの “test” フィールドを使って、(計算機アーキテクチャや、利用しているウィンドウシステムといった) 何らかの外部条件をテストするよう指定することができ

す。findmatch() はこれらの条件を自動的にチェックし、チェックが失敗したエントリを読み飛ばします。

**getcaps()**

MIME タイプを mailcap ファイルのエントリに対応付ける辞書を返します。この辞書は findmatch() 関数に渡されるべきものです。エントリは辞書のリストとして記憶されますが、この表現形式の詳細について知っておく必要はないでしょう。

mailcap 情報はシステム上で見つかった全ての mailcap ファイルから導出されます。ユーザ設定の mailcap ファイル '\$HOME/.mailcap' はシステムの mailcap ファイル '/etc/mailcap'、'/usr/etc/mailcap'、および '/usr/local/etc/mailcap' の内容を上書きします。

以下に使用例を示します:

```
>>> import mailcap
>>> d=mailcap.getcaps()
>>> mailcap.findmatch(d, 'video/mpeg', filename='/tmp/tmp1223')
('xmpeg /tmp/tmp1223', {'view': 'xmpeg %s'})
```

## 12.4 mailbox — 様々なメールボックス形式の読み出し

このモジュールでは (UNIX の) メールボックス内のメールに簡単かつ一般的な方法でアクセスできるようにするクラスを定義しています。

**class UnixMailbox**(fp[, factory])

全てのメッセージが単一のファイルに収められ、'From ' ('From\_' として知られています) 行によって分割されているような、旧来の UNIX 形式のメールボックスにアクセスします。ファイルオブジェクト fp はメールボックスファイルを指します。オプションの factory パラメタは新たなメッセージオブジェクトを生成するような呼び出し可能オブジェクトです。factory は、メールボックスオブジェクトに対して next() メソッドを実行した際に、単一の引数、fp を伴って呼び出されます。この引数の標準の値は rfc822.Message クラスです (rfc822 モジュール – および以下 – を参照してください)。

可搬性を最大限にするために、UNIX 形式のメールボックス内にあるメッセージは、正確に 'From ' (末尾の空白に注意してください) で始まる文字列が、直前の正しく二つの改行の後にくるような行で分割されます。現実的には広範なバリエーションがあるため、それ以外の From\_ 行について考慮すべきではないのですが、現在の実装では先頭の二つの改行をチェックしていません。これはほとんどのアプリケーションでうまく動作します。

UnixMailbox クラスでは、ほぼ正確に From\_ デリミタにマッチするような正規表現を用いることで、より厳密に From\_ 行のチェックを行うバージョンを実装しています。UnixMailbox ではデリミタ行が 'From namevartime' の行に分割されるものと考えます。可搬性を最大限にするためには、代わりに PortableUnixMailbox クラスを使ってください。このクラスは UnixMailbox と同じですが、個々のメッセージは 'From ' 行だけで分割されるものとみなします。

より詳細な情報については、*Configuring Netscape Mail on UNIX: Why the Content-Length Format is Bad* を参照してください。

**class PortableUnixMailbox**(fp[, factory])

厳密性の低い UnixMailbox のバージョンで、メッセージを分割する行は 'From ' のみであると見なします。実際に見られるメールボックスのバリエーションに対応するため、From 行における "name time" 部分は無視されます。メール処理ソフトウェアはメッセージ中の 'From ' で始まる行をクオー



トするため、この分割はうまく動作します。

```
class MmdfMailbox(fp[, factory])
```

全てのメッセージが単一のファイルに収められ、4つの control-A 文字によって分割されているような、MMDF 形式のメールボックスにアクセスします。ファイルオブジェクト *fp* はメールボックスファイルをさします。オプションの *factory* は `UnixMailbox` クラスにおけるのと同様です。

```
class MHMailbox(dirname[, factory])
```

数字で名前のつけられた別々のファイルに個々のメッセージを収めたディレクトリである、MH メールボックスにアクセスします。メールボックスディレクトリの名前は *dirname* で渡します。 *factory* は `UnixMailbox` クラスにおけるのと同様です。

```
class Maildir(dirname[, factory])
```

Qmail メールディレクトリにアクセスします。 *dirname* で指定された全ての新規および現在のメッセージにアクセスできます。 *factory* は `UnixMailbox` クラスにおけるのと同様です。

```
class BabylMailbox(fp[, factory])
```

MMDF メールボックスと似ている、Babyl メールボックスにアクセスします。Babyl 形式では、各メッセージは二つのヘッダからなるセット、*original* ヘッダおよび *visible* ヘッダを持っています。original ヘッダは '\*\*\* EOOH \*\*\*' (End-Of-Original-Headers) だけを含む行の前にあり、visible ヘッダは EOOH 行の後にあります。Babyl 互換のメールリーダーは visible ヘッダのみを表示し、BabylMailbox オブジェクトは visible ヘッダのみを含むようなメッセージを返します。メールメッセージは EOOH 行で始まり、'\037\014' だけを含む行で終わります。 *factory* は `UnixMailbox` クラスにおけるのと同様です。

rfc822 モジュールが撤廃されたことにより、email パッケージを使ってメールボックスからメッセージオブジェクトを生成するよう推奨されているので注意してください。(デフォルトの設定は以前のバージョンとの互換性のために変更されていません。)安全に移行を行うには、以下のちょっとしたコードを使います:

```
import email
import email.Errors
import mailbox

def msgfactory(fp):
    try:
        return email.message_from_file(fp)
    except email.Errors.MessageParseError:
        # Don't return None since that will
        # stop the mailbox iterator
        return ''

mbox = mailbox.UnixMailbox(fp, msgfactory)
```

上のラッパはメールボックス内にある不正な形式の MIME メッセージに対して防御性がある反面、メールボックスの `next()` メソッドが空文字列を渡す場合に備えなければなりません。逆に、もしメールボックス内には正しい形式の MIME メッセージしか入っていないと分かっているのなら、単に以下のようにします:

```
import email
import mailbox

mbox = mailbox.UnixMailbox(fp, email.message_from_file)
```

参考資料:

*mbox - file containing mail messages*

(<http://www.gmail.org/man/man5/mbox.html>)

伝統的な“mbox”メールボックス形式に関する記述です。

*maildir - directory for incoming mail messages*

(<http://www.gmail.org/man/man5/maildir.html>)

“maildir”メールボックス形式の記述です。

*Configuring Netscape Mail on UNIX: Why the Content-Length Format is Bad*

(<http://home.netscape.com/eng/mozilla/2.0/relnotes/demo/content-length.html>)

メールボックスファイルに記録されている Content-Length: ヘッダに依存した場合に発生する問題についての記述です。

### 12.4.1 Mailbox オブジェクト

メールボックスオブジェクトの実装はすべて反復可能なオブジェクトであり、外部に公開されているメソッドを一つもっています。このメソッドはメールボックスオブジェクトから生成されるイテレータによって使われ、直接利用することもできます。

**next()**

メールボックスオブジェクトのコンストラクタに渡された、オプションの *factory* 引数を使って、メールボックス中の次のメッセージを生成して返します。標準の設定では、*factory* は `rfc822.Message` オブジェクトです (`rfc822` モジュールを参照してください)。メールボックスの実装により、このオブジェクトの *fp* 属性は真のファイルオブジェクトかもしれないし、複数のメールメッセージが単一のファイルに収められているなどの場合に、メッセージ間の境界を注意深く扱うためにファイルオブジェクトをシミュレートするクラスのインスタンスであるかもしれません。次のメッセージがない場合、このメソッドは `None` を返します。

## 12.5 mhlb — MH のメールボックスへのアクセス機構

`mhlb` モジュールは MH フォルダおよびその内容に対する Python インタフェースを提供します。

このモジュールには、あるフォルダの集まりを表現する `MH`、単一のフォルダを表現する `Folder`、単一のメッセージを表現する `Message`、の 3 つのクラスが入っています。

**class MH([path[, profile]])**

`MH` は MH フォルダの集まりを表現します。

**class Folder(mh, name)**

`Folder` クラスは単一のフォルダとフォルダ内のメッセージ群を表現します。

**class Message(folder, number[, name])**

`Message` オブジェクトはフォルダ内の個々のメッセージを表現します。メッセージクラスは `mime-tools.Message` から導出されています。

### 12.5.1 MH オブジェクト

`MH` インスタンスは以下のメソッドを持っています:

**error(format[, ...])**

エラーメッセージを出力します – 上書きすることができます。

**getprofile(key)**

プロファイルエントリ (設定されていなければ `None`) を返します。

**getpath()**  
 メールボックスのパス名を返します。

**getcontext()**  
 現在のフォルダ名を返します。

**setcontext(name)**  
 現在のフォルダ名を設定します。

**listfolders()**  
 トップレベルフォルダのリストを返します。

**listallfolders()**  
 全てのフォルダを列挙します。

**listsubfolders(name)**  
 指定したフォルダの直下にあるサブフォルダのリストを返します。

**listallsubfolders(name)**  
 指定したフォルダの下にある全てのサブフォルダのリストを返します。

**makefolder(name)**  
 新しいフォルダを生成します。

**deletefolder(name)**  
 フォルダを削除します – サブフォルダが入っているはいけません。

**openfolder(name)**  
 新たな開かれたフォルダオブジェクトを返します。

## 12.5.2 Folder オブジェクト

Folder インスタンスは開かれたフォルダを表現し、以下のメソッドを持っています:

**error(format[, ...])**  
 エラーメッセージを出力します – 上書きすることができます。

**getfullname()**  
 フォルダの完全なパス名を返します。

**getsequencesfilename()**  
 フォルダ内のシーケンスファイルの完全なパス名を返します。

**getmessagefilename(n)**  
 フォルダ内のメッセージ *n* の完全なパス名を返します。

**listmessages()**  
 フォルダ内のメッセージの (番号の) リストを返します。

**getcurrent()**  
 現在のメッセージ番号を返します。

**setcurrent(n)**  
 現在のメッセージ番号を *n* に設定します。

**parsesequence(seq)**  
 msgs 文を解釈して、メッセージのリストにします。

**getlast()**  
 最新のメッセージを取得します。メッセージがフォルダにない場合には 0 を返します。

**setlast(n)**

最新のメッセージを設定します (内部使用のみ)。

**getsequences()**

フォルダ内のシーケンスからなる辞書を返します。シーケンス名がキーとして使われ、値はシーケンスに含まれるメッセージ番号のリストになります。

**putsequences(dict)**

フォルダ内のシーケンスからなる辞書 name: list を返します。

**removemessages(list)**

リスト中のメッセージをフォルダから削除します。

**refilemessages(list, tofolder)**

リスト中のメッセージを他のフォルダに移動します。

**movemessage(n, tofolder, ton)**

一つのメッセージを他のフォルダの指定先に移動します。

**copymessage(n, tofolder, ton)**

一つのメッセージを他のフォルダの指定先にコピーします。

### 12.5.3 Message オブジェクト

Message クラスは `mimertools.Message` のメソッドに加え、一つメソッドを持っています:

**openmessage(n)**

新たな開かれたメッセージオブジェクトを返します (ファイル記述子を一つ消費します)。

## 12.6 mimertools — MIME メッセージを解析するためのツール

リリース 2.3 以降で撤廃された仕様です。 `email` パッケージを `mimertools` モジュールより優先して使うべきです。このモジュールは、下位互換性維持のためにのみ存在しています。

このモジュールは、`rfc822` モジュールの `Message` クラスのサブクラスと、マルチパート MIME や符  
合化メッセージの操作に役に立つ多くのユーティリティ関数を定義しています。

これには以下の項目が定義されています:

**class Message(fp[, seekable])**

`Message` クラスの新しいインスタンスを返します。これは、`rfc822.Message` クラスのサブクラスで、いくつかの追加のメソッドがあります (以下を参照のこと)。`seekable` 引数は、`rfc822.Message` のものと同じ意味を持ちます。

**choose\_boundary()**

パートの境界として使うことができる見込みが高いユニークな文字列を返します。その文字列は、`'hostipaddr.uid.pid.timestamp.random'` の形をしています。

**decode(input, output, encoding)**

オープンしたファイルオブジェクト `input` から、許される MIME `encoding` を使って符号化されたデータを読んで、オープンされたファイルオブジェクト `output` に復号化されたデータを書きます。`encoding` に許される値は、`'base64'`、`'quoted-printable'`、`'uuencode'`、`'x-uuencode'`、`'uue'`、`'x-uue'`、`'7bit'`、および `'8bit'` です。`'7bit'` あるいは `'8bit'` で符号化されたメッセージを復号化しても何も効果がありません。入力が出力に単純にコピーされるだけです。

**encode(input, output, encoding)**

オープンしたファイルオブジェクト `input` からデータを読んで、それを許される MIME `encoding` を使って符号化して、オープンしたファイルオブジェクト `output` に書きます。`encoding` に許される値

は、`[methoddecode()]` のものと同じです。

**copyliteral**(*input*, *output*)

オープンしたファイル *input* から行を EOF まで読んで、それらをオープンしたファイル *output* に書きます。

**copybinary**(*input*, *output*)

オープンしたファイル *input* からブロックを EOF まで読んで、それらをオープンしたファイル *output* に書きます。ブロックの大きさは現在 8192 に固定されています。

参考資料:

email モジュール (12.2 節):

圧縮電子メール操作パッケージ ; `mimetools` モジュールに委譲。

`rfc822` モジュール (12.11 節):

`mimetools.Message` のベースクラスを提供する。

`multifile` モジュール (12.10 節):

MIME データのような、別個のパーツを含むファイルの読み込みをサポート。

<http://www.cs.uu.nl/wais/html/na-dir/mail/mime-faq/.html>

MIME でよく訊ねられる質問。MIME の概要に関しては、この文書の Part 1 の質問 1.1 への答えを見ること。

### 12.6.1 Message オブジェクトの追加メソッド

`Message` クラスは、`rfc822.Message` メソッドに加えて、以下のメソッドを定義しています :

**getplist()**

Content-Type: ヘッダのパラメータリストを返します。これは文字列のリストです。'key=value' の形のパラメータに対しては、*key* は小文字に変換されますが、*value* は変換されません。たとえば、もしメッセージに、ヘッダ 'Content-type: text/html; spam=1; Spam=2; Spam' が含まれていれば、`getplist()` は、Python リスト ['spam=1', 'spam=2', 'Spam'] を返すでしょう。

**getparam**(*name*)

与えられた *name* の ('name=value' の形に対して `getplist()` が返す) 第 1 パラメータの *value* を返します。もし *value* が、'<...>' あるいは '"..."' のように引用符で囲まれていれば、これらは除去されます。

**getencoding()**

Content-Transfer-Encoding: メッセージヘッダで指定された符号化方式を返します。もしそのようなヘッダが存在しなければ、'7bit' を返します。符号化方式文字列は小文字に変換されます。

**gettype()**

Content-Type: ヘッダで指定された ('type/subtype' の形での) メッセージ型を返します。もしそのようなヘッダが存在しなければ、'text/plain' を返します。型文字列は小文字に変換されます。

**getmaintype()**

Content-Type: ヘッダで指定された主要型を返します。もしそのようなヘッダが存在しなければ、'text' を返します。主要型文字列は小文字に変換されます。

**getsubtype()**

Content-Type:ヘッダで指定された下位型を返します。もしそのようなヘッダが存在しなければ、'plain' を返します。下位型文字列は小文字に変換されます。

## 12.7 mimetypes — ファイル名を MIME 型へマップする

mimetypes モジュールは、ファイル名あるいは URL と、ファイル名拡張子に関連付けられた MIME 型とを変換します。ファイル名から MIME 型へと、MIME 型からファイル名拡張子への変換が提供されます；後者の変換では符号化方式はサポートされていません。

このモジュールは、一つのクラスと多くの便利な関数を提供します。これらの関数がこのモジュールへの標準のインターフェースですが、アプリケーションによっては、そのクラスにも関係するかもしれません。

以下で説明されている関数は、このモジュールへの主要なインターフェースを提供します。たとえモジュールが初期化されていなくても、もしこれらの関数が、`init()` がセットアップする情報に依存していれば、これらの関数は、`init()` を呼びます。

`guess_type(filename[, strict])`

*filename* で与えられるファイル名あるいは URL に基づいて、ファイルの型を推定します。戻り値は、タプル (*type*, *encoding*) です、ここで *type* は、もし型が (拡張子がないあるいは未定義のため) 推定できない場合は、`None` を、あるいは、MIME content-type: ヘッダ に利用できる、`'type/subtype'` の形の文字列です。

*encoding* は、符号化方式がない場合は `None` を、あるいは、符号化に使われるプログラムの名前 (たとえば、`compress` あるいは `gzip`) です。符号化方式は Content-Encoding: ヘッダとして使うのに適しており、Content-Transfer-Encoding: ヘッダには適していません。マッピングはテーブルドリブンです。符号化方式のサフィックスは大/小文字を区別します；データ型サフィックスは、最初大/小文字を区別して試し、それから大/小文字を区別せずに試します。

省略可能な *strict* は、既知の MIME 型のリストとして認識されるものが、IANA として登録された 正式な型のみに限定されるかどうかを指定するフラグです。*strict* が `true` (デフォール) の時は、IANA 型のみがサポートされます；*strict* が `false` のときは、いくつかの追加の、非標準ではあるが、一般的に使用される MIME 型も認識されます。

`guess_all_extensions(type[, strict])`

*type* で与えられる MIME 型に基づいてファイルの拡張子を推定します。戻り値は、先頭のドット ('.') を含む、可能なファイル拡張子すべてを与える文字列のリストです。拡張子と特別なデータストリームとの関連付けは保証されませんが、`guess_type()` によって MIME 型 *type* とマップされます。

省略可能な *strict* は `guess_type()` 関数のものと同じ意味を持ちます。

`guess_extension(type[, strict])`

*type* で与えられる MIME 型に基づいてファイルの拡張子を推定します。戻り値は、先頭のドット ('.') を含む、ファイル拡張子を与える文字列のリストです。拡張子と特別なデータストリームとの関連付けは保証されませんが、`guess_type()` によって MIME 型 *type* とマップされます。もし *type* に対して拡張子が推定できない場合は、`None` が返されます。

省略可能な *strict* は `guess_type()` 関数のものと同じ意味を持ちます。

モジュールの動作を制御するために、いくつかの追加の関数とデータ項目が利用できます。

`init([files])`

内部のデータ構造を初期化します。もし *files* が与えられていれば、これはデフォルトの型のマップを増やすために使われる、一連のファイル名でなければなりません。もし省略されていれば、使われるファイル名は `knownfiles` から取られます。*file* あるいは `knownfiles` 内の各ファイル名は、それ以前に現れる名前より優先されます。繰り返し `init()` を呼び出すことは許されています。

`read_mime_types(filename)`

ファル *filename* で与えられた型のマップが、もしあればロードします。型のマップは、先頭の dot ('.') を含むファイル名拡張子を、`'type/subtype'` の形の文字列にマッピングする辞書として返されます。



もしファイル *filename* が存在しないか、読み込めなければ、`None` が返されます。

**add\_type**(*type*, *ext*[, *strict*])

mime 型 *type* からのマッピングを拡張子 *ext* に追加します。拡張子がすでに既知であれば、新しい型が古いものに置き替わります。その型がすでに既知であれば、その拡張子が、既知の拡張子のリストに追加されます。

*strict* がある時は、そのマッピングは正式な MIME 型に、そうでなければ、非標準の MIME 型に追加されます。

**inited**

グローバルなデータ構造が初期化されているかどうかを示すフラグ。これは `init()` により `true` に設定されます。

**knownfiles**

共通にインストールされた型マップファイル名のリスト。これらのファイルは、普通 `'mime.types'` という名前であり、パッケージごとに異なる場所にインストールされます。

**suffix\_map**

サフィックスをサフィックスにマップする辞書。これは、符号化方式と型が同一拡張子で示される符号化ファイルが認識できるように使用されます。例えば、`'tar.gz'` 拡張子は、符号化と型が別個に認識できるように `'tar.gz'` にマップされます。

**encodings\_map**

ファイル名拡張子を符号化方式型にマッピングする辞書

**types\_map**

ファイル名拡張子を MIME 型にマップする辞書

**common\_types**

ファイル名拡張子を非標準ではあるが、一般に使われている MIME 型にマップする辞書

`MimeTypes` クラスは、1 つ以上の MIME-型 データベースを必要とするアプリケーションに役に立つでしょう。

**class MimeTypes**( [*filenames* ] )

このクラスは、MIME-型データベースを表現します。デフォルトでは、このモジュールの他のものと同じデータベースへのアクセスを提供します。初期データベースは、このモジュールによって提供されるもののコピーで、追加の `'mime.types'`-形式のファイルを、`read()` あるいは `readfp()` メソッドを使って、データベースにロードすることで拡張されます。マッピング辞書も、もしデフォルトのデータが望むものでなければ、追加のデータをロードする前にクリアされます。

省略可能な *filenames* パラメータは、追加のファイルを、デフォルトデータベースの"トップに"ロードさせるのに使うことができます。

2.2 で追加された仕様です。

### 12.7.1 Mime 型 オブジェクト

`MimeTypes` インスタンスは、`mimetypes` モジュールのそれと非常によく似たインターフェースを提供します。

**suffix\_map**

サフィックスをサフィックスにマップする辞書。これは、符号化方式と型が同一拡張子で示されるような符号化ファイルが認識できるように使用されます。例えば、`'tar.gz'` 拡張子は、符号化方式と型が別個に認識できるように `'tar.gz'` に対応づけられます。これは、最初はモジュールで定義されたグローバルな `suffix_map` のコピーです。

#### **encodings\_map**

ファイル名拡張子を符号化型にマッピングする辞書。これは、最初はモジュールで定義されたグローバルな `encodings_map` のコピーです。

#### **types\_map**

ファイル名拡張子を MIME 型にマッピングする辞書。これは、最初はモジュールで定義されたグローバルな `types_map` のコピーです。

#### **common\_types**

ファイル名拡張子を非標準ではあるが、一般に使われている MIME 型にマップする辞書。これは、最初はモジュールで定義されたグローバルな `common_types` のコピーです。

#### **guess\_extension(*type* [, *strict*])**

`guess_extension()` 関数と同様に、オブジェクトの一部として保存されたテーブルを使用します。

#### **guess\_type(*url* [, *strict*])**

`guess_type()` 関数と同様に、オブジェクトの一部として保存されたテーブルを使用します。

#### **read(*path*)**

MIME 情報を、*path* という名のファイルからロードします。これはファイルを解析するのに `readfp()` を使用します。

#### **readfp(*file*)**

MIME 型情報を、オープンしたファイルからロードします。ファイルは、標準の 'mime.types' ファイルの形式でなければなりません。

## 12.8 MimeTypeWriter — 汎用 MIME ファイルライター

リリース 2.3 以降で撤廃された仕様です。 `email` パッケージを、`MimeTypeWriter` モジュールよりも優先して使用すべきです。このモジュールは、下位互換性維持のためだけに存在します。

このモジュールは、クラス `MimeTypeWriter` を定義します。この `MimeTypeWriter` クラスは、MIME マルチパートファイルを作成するための基本的なフォーマッタを実装します。これは出力ファイル内をあちこち移動することも、大量のバッファスペースを使うこともありません。あなたは、最終のファイルに現れるであろう順番に、パートを書かなければなりません。`MimeTypeWriter` は、あなたが追加するヘッダをバッファして、それらの順番を並び替えることができますようにします。

#### **class MimeTypeWriter(*fp*)**

`MimeTypeWriter` クラスの新しいインスタンスを返します。渡される唯一の引数 *fp* は、書くために使用するファイルオブジェクトです。`StringIO` オブジェクトを使うこともできることに注意して下さい。

### 12.8.1 MimeTypeWriter オブジェクト

`MimeTypeWriter` インスタンスには以下のメソッドがあります：

#### **addheader(*key*, *value* [, *prefix*])**

MIME メッセージに新しいヘッダ行を追加します。*key* は、そのヘッダの名前であり、そして *value* で、そのヘッダの値を明示的に与えます。省略可能な引数 *prefix* は、ヘッダが挿入される場所を決定します；'0' は最後に追加することを意味し、'1' は先頭への挿入です。デフォルトは最後に追加することです。

#### **flushheaders()**

今まで集められたヘッダすべてが書かれ(そして忘れられ)るようになります。これは、もし全く本体が

必要でない場合に役に立ちます。例えば、ヘッダのような情報を保管するために (誤って) 使用された、型 `message/rfc822` のサブパート用。

`startbody(ctype[, plist[, prefix]])`

メッセージの本体に書くのに使用できるファイルのようなオブジェクトを返します。コンテンツ-型は、与えられた `ctype` に設定され、省略可能なパラメータ `plist` は、コンテンツ-型定義のための追加のパラメータを与えます。`prefix` は、そのデフォルトが先頭への挿入以外は `addheader()` のように働きます。

`startmultipartbody(subtype[, boundary[, plist[, prefix]]])`

メッセージ本体を書くのに使うことができるファイルのようなオブジェクトを返します。更に、このメソッドはマルチパートのコードを初期化します。ここで、`subtype` が、そのマルチパートのサブタイプを、`boundary` がユーザ定義の境界仕様を、そして `plist` が、そのサブタイプ用の省略可能なパラメータを定義します。`prefix` は、`startbody()` のように働きます。サブパートは、`nextpart()` を使って作成するべきです。

`nextpart()`

マルチパートメッセージの個々のパートを表す、`MimeWriter` の新しいインスタンスを返します。これは、そのパートを書くのにも、また複雑なマルチパートを再帰的に作成するのにも使うことができます。メッセージは、`nextpart()` を使う前に、最初 `startmultipartbody()` で初期化しなければなりません。

`lastpart()`

これは、マルチパートメッセージの最後のパートを指定するのに使うことができ、マルチパートメッセージを書くときはいつでも使うべきです。

## 12.9 mimify — 電子メールメッセージの MIME 処理

リリース 2.3 以降で撤廃された仕様です。mimify モジュールを使うよりも `email` パッケージを使うべきです。このモジュールは以前のバージョンとの互換性のために保守されているにすぎません。

mimify モジュールでは電子メールメッセージから MIME へ、および MIME から電子メールメッセージへの変換を行うための二つの関数を定義しています。電子メールメッセージは単なるメッセージでも、MIME 形式でもかまいません。各パートは個別に扱われます。メッセージ (の一部) の MIME 化 (mimify) の際、7 ビット ASCII 文字を使って表現できない何らかの文字が含まれていた場合、メッセージの quoted-printable への符号化が伴います。メッセージが送信される前に編集しなければならない場合、MIME 化および非 MIME 化は特に便利です。典型的な使用法は以下のようになります:

```
unmimify message
edit message
mimify message
send message
```

モジュールでは以下のユーザから呼び出し可能な関数と、ユーザが設定可能な変数を定義しています:

`mimify(infile, outfile)`

`infile` を `outfile` にコピーします。その際、パートを quoted-printable に変換し、必要なら MIME メールヘッダを追加します。`infile` および `outfile` はファイルオブジェクト (実際には、`readline()` メソッドを持つ (`infile`) か、`write(outfile)` メソッドを持つあらゆるオブジェクト) か、ファイル名を指す文字列を指定することができます。`infile` および `outfile` が両方とも文字列の場合、同じ値にすることができます。

`unmimify(infile, outfile[, decode_base64])`

*infile* を *outfile* にコピーします。その際、全ての quoted-printable 化された部分を復号化します。*infile* および *outfile* はファイルオブジェクト (実際には、`readline()` メソッドを持つ (*infile*) か、`write` (*outfile*) メソッドを持つあらゆるオブジェクト) か、ファイル名を指す文字列を指定することができます。`decode_base64` 引数が与えられており、その値が真である場合、base64 符号で符号化されているパートも同様に復号化されます。

`mime_decode_header(line)`

*line* 内の符号化されたヘッダ行が復号化されたものを返します。ISO 8859-1 文字セット (Latin-1) だけをサポートします。

`mime_encode_header(line)`

*line* 内のヘッダ行が MIME 符号化されたものを返します。

#### MAXLEN

標準では、非 ASCII 文字 (8 ビット目がセットされている文字) を含むか、MAXLEN 文字 (標準の値は 200 です) よりも長い部分は quoted-printable 形式で符号化されます。

#### CHARSET

文字セットがメールヘッダで指定されていない場合指定しなければなりません。使われている文字セットを表す文字列は CHARSET に記憶されます。標準の値は ISO-8859-1 (Latin1 (latin-one) としても知られています)。

このモジュールはコマンドラインから利用することもできます。以下のような使用法:

```
mimify.py -e [-l length] [infile [outfile]]
mimify.py -d [-b] [infile [outfile]]
```

で、それぞれ符号化 (mimify) および復号化 (unmimify) を行います。標準の設定では *infile* は標準入力で、*putfile* は標準出力です。入出力に同じファイルを指定することもできます。

符号化の際に `-l` オプションを与えた場合、*length* で指定した長さより長い行があれば、その長さに含まれる部分が符号化されます。

復号化の際に `-b` オプションが与えられていれば、base64 パートも同様に復号化されます。

参考資料:

quopri モジュール (12.15 節):

MIME quoted-printable 形式ファイルのエンコードおよびデコード。

## 12.10 multifile — 個別の部分を含んだファイル群のサポート

MultiFile オブジェクトはテキストファイルを区分したものをファイル類似の入力オブジェクトとして扱うようにし、指定した区切り文字 (delimiter) パタンに遭遇した際に " が返されるようにします。このクラスの標準設定は MIME マルチパートメッセージを解釈する上で便利となるように設計されていますが、サブクラス化を行って幾つかのメソッドを上書きすることで、簡単に汎用目的に対応させることができます。

`class MultiFile(fp[, seekable])`

マルチファイル (multi-file) を生成します。このクラスは `open()` が返すファイルオブジェクトのような、MultiFile インスタンスが行データを取得するための入力となるオブジェクトを引数としてインスタンス化を行わなければなりません。

MultiFile は入力オブジェクトの `readline()`、`seek()`、および `tell()` メソッドしか参照せず、後者の二つのメソッドは個々の MIME パートにランダムアクセスしたい場合にのみ必要です。

MultiFile を seek できないストリームオブジェクトで使うには、オプションの *seekable* 引数の値を偽にしてください; これにより、入力オブジェクトの `seek()` および `tail()` メソッドを使わないようになります。

MultiFile の視点から見ると、テキストは三種類の行データ: データ、セクション分割子、終了マーカ、からなることを知っていることと約に立つでしょう。MultiFile は、多重入れ子構造になっている可能性のある、それぞれが独自のセクション分割子および終了マーカのパターンを持つメッセージパートをサポートするように設計されています。

参考資料:

email モジュール (12.2 節):

網羅的な電子メール操作パッケージ; multifile モジュールに取って代わります。

### 12.10.1 MultiFile オブジェクト

MultiFile インスタンスには以下のメソッドがあります:

**readline(*str*)**

一行データを読みます。その行が (セクション分割子や終了マーカや本物の EOF でない) データの場合、行データを返します。その行がもっとも最近スタックにプッシュされた境界パターンにマッチした場合、`"` を返し、マッチした内容が終了マーカかそうでないかによって `self.last` を 1 か 0 に設定します。行がその他のスタックされている境界パターンにマッチした場合、エラーが送出されます。背後のストリームオブジェクトにおけるファイルの終端に到達した場合、全ての境界がスタックから除去されていない限りこのメソッドは `Error` を送出します。

**readlines(*str*)**

このパートの残りの全ての行を文字列のリストとして返します。

**read()**

次のセクションまでの全ての行を読みます。読んだ内容を単一の (複数行にわたる) 文字列として返します。このメソッドには `size` 引数をとらないので注意してください!

**seek(*pos* [, *whence* ])**

ファイルを seek します。seek する際のインデクスは現在のセクションの開始位置からの相対位置になります。 *pos* および *whence* 引数はファイルの seek における引数と同じように解釈されます。

**tell()**

現在のセクションの先頭に対して相対的なファイル位置を返します。

**next()**

次のセクションまで行を読み飛ばします (すなわち、セクション分割子または終了マーカが消費されるまで行データを読みます)。次のセクションがあった場合には真を、終了マーカが発見された場合には偽を返します。最も最近スタックにプッシュされた境界パターンを最有効化します。

**is\_data(*str*)**

*str* がデータの場合に真を返し、セクション分割子の可能性がある場合には偽を返します。このメソッドは行の先頭が (全ての MIME 境界が持っている) `'--'` 以外になっているかを調べるように実装されていますが、導出クラスで上書きできるように宣言されています。

このテストは実際の境界テストにおいて高速性を保つために使われているので注意してください; このテストが常に `false` を返す場合、テストが失敗するのではなく、単に処理が遅くなるだけです。

**push(*str*)**

境界文字列をスタックにプッシュします。この境界文字列の適切に修飾されたバージョンが入力ファイル中に見つかった場合、セクション分割子または終了マーカであると解釈されます。それ以降の全てのデータ読み出しは、`pop()` を呼んで境界文字列を除去するか、`next()` を呼んで境界文字列を

再有効化しないかぎり、ファイル終端を示す空文字列を返します。

一つ以上の境界をプッシュすることは可能です。もっとも最近プッシュされた境界に遭遇すると EOF が返ります; その他の境界に遭遇するとエラーが送出されます。

`pop()`

セクション境界をポップします。この境界はもはや EOF として解釈されません。

`section_divider(str)`

境界をセクション分割子にします。標準では、このメソッドは(全ての MIME 境界が持っている) '---' を境界文字列の先頭に追加しますが、これは導出クラスで上書きできるように宣言されています。末尾の空白は無視されることから考えて、このメソッドでは LF や CR-LF を追加する必要はありません。

`end_marker(str)`

境界文字列を終了マーカ行にします。標準では、このメソッドは (MIME マルチパートデータのメッセージ終了マーカのように) '---' を境界文字列の先頭に追加し、かつ '---' を境界文字列の末尾に追加しますが、これは導出クラスで上書きできるように宣言されています。末尾の空白は無視されることから考えて、このメソッドでは LF や CR-LF を追加する必要はありません。

最後に、MultiFile インスタンスは二つの公開されたインスタンス変数を持っています:

`level`

現在のパートにおける入れ子の深さです。

`last`

最後に見つかったファイル終了イベントがメッセージ終了マーカであった場合に真となります。

## 12.10.2 MultiFile の例

```
import mimetools
import multifile
import StringIO

def extract_mime_part_matching(stream, mimetype):
    """Return the first element in a multipart MIME message on stream
    matching mimetype."""

    msg = mimetools.Message(stream)
    msgtype = msg.gettype()
    params = msg.getplist()

    data = StringIO.StringIO()
    if msgtype[:10] == "multipart/":

        file = multifile.MultiFile(stream)
        file.push(msg.getparam("boundary"))
        while file.next():
            submsg = mimetools.Message(file)
            try:
                data = StringIO.StringIO()
                mimetools.decode(file, data, submsg.getencoding())
            except ValueError:
                continue
            if submsg.gettype() == mimetype:
                break
        file.pop()
    return data.getvalue()
```



## 12.11 rfc822 — RFC 2822 準拠のメールヘッダ読み出し

リリース 2.3 以降で撤廃された仕様です。rfc822 モジュールを使うよりも email パッケージを使うべきです。このモジュールは以前のバージョンとの互換性のために保守されているにすぎません。

このモジュールでは、インターネット標準 RFC 2822<sup>10</sup>で定義されている“電子メールメッセージ”を表現するクラス、`Message` を定義しています。このメッセージはメッセージヘッダ群とメッセージボディの集まりからなります。このモジュールではまた、ヘルパークラス RFC 2822 アドレス群を解釈するための `AddressList` クラスを定義しています。RFC 2822 メッセージ固有の構文に関する情報は RFC を参照してください。

mailbox モジュールでは、多くのエンドユーザメールプログラムによって生成されるメールボックスを読み出すためのクラスを提供しています。

**class `Message`** (*file* [, *seekable* ])

`Message` インスタンスは入力オブジェクトをパラメタに与えてインスタンス化します。入力オブジェクトのメソッドのうち、`Message` が依存するのは `readline()` だけです; 通常ファイルオブジェクトは適格です。インスタンス化を行うと、入力オブジェクトからデリミタ行 (通常は空行 1 行) に到達するまでヘッダを読み出し、それらをインスタンス中に保持します。ヘッダの後のメッセージ本体は読み出しません。

このクラスは `readline()` メソッドをサポートする任意の入力オブジェクトを扱うことができます。入力オブジェクトが `seek` および `tell` できる場合、`rewindbody()` メソッドが動作します。また、不正な行データを入力ストリームにプッシュバックできます。入力オブジェクトが `seek` できない一方で、入力行をプッシュバックする `unread()` メソッドを持っている場合、`Message` は不正な行データにこのプッシュバックを使います。こうして、このクラスはバッファされているストリームから来るメッセージを解釈するのに使うことができます。

オプションの *seekable* 引数は、`lseek()` システムコールが動作しないと分かるまでは `tell()` がバッファされたデータを無視するような、ある種の `stdio` ライブラリで回避手段として提供されています。可搬性を最大にするために、`socket` オブジェクトによって生成されたファイルのような、`seek` できないオブジェクトを渡す際には、最初に `tell()` が呼び出されないようにするために *seekable* 引数をゼロに設定すべきです。

ファイルとして読み出された入力行データは CR-LF と単一の改行 (line feed) のどちらで終端されていてもかまいません; 行データを記憶する前に、終端の CR-LF は単一の改行と置き換えられます。

ヘッダに対するマッチは全て大小文字に依存しません。例えば、`m['From']`、`m['from']`、および `m['FROM']` は全て同じ結果になります。

**class `AddressList`** (*field*)

RFC 2833 アドレスをカンマで区切ったものとして解釈される単一の文字列パラメタを使って、`AddressList` ヘルパークラスをインスタンス化することができます。(パラメタ `None` は空のリストを表します。)

**`quote`** (*str*)

*str* 中のバックスラッシュが 2 つのバックスラッシュに置き換えられ、二重引用符がバックスラッシュ付きの二重引用符に置き換えられた、新たな文字列を返します。

**`unquote`** (*str*)

*str* の逆クオートされた 新たな文字列を返します。*str* が二重引用符で囲われていた場合、二重引用符を剥ぎ取ります。同様に、*str* が三角括弧で囲われていた場合にも剥ぎ取ります。

<sup>10</sup> このモジュールはもともと RFC 822 に適合していたので、そういう名前になっています。その後、RFC 2822 が RFC 822 に対する更新としてリリースされました。このモジュールは RFC 2822 適合であり、特に RFC 822 からの構文や意味付けに対する変更がなされています。

`parseaddr(address)`

To: や Cc: といった、アドレスが入っているフィールドの値 *address* を解析し、含まれている“実名 (realname)” 部分および“電子メールアドレス” 部分に分けます。それらの情報からなるタプルを返します。解析が失敗した場合には 2 要素のタプル (`None`, `None`) を返します。

`dump_address_pair(pair)`

`parseaddr()` の逆で、(*realname*, *email\_address*) 形式の 2 要素のタプルをとり、To: や Cc: ヘッダに適した文字列値を返します。*pair* の最初の要素が真値をとらない場合、二つ目の要素をそのまま返します。

`parsedate(date)`

RFC 2822 の規則に従っている日付を解析しようと試みます。しかしながら、メイラによっては RFC 2822 で指定されているような書式に従わないため、そのような場合には `parsedata()` は正しい日付を推測しようと試みます。*date* は 'Mon, 20 Nov 1995 19:12:08 -0500' のような RFC 2822 様式の日付を収めた文字列です。日付の解析に成功した場合、`parsedate()` は `time.mktime()` にそのまま渡すことができるような 9 要素のタプルを返します; そうでない場合には `None` を返します。結果のフィールド 6、7、および 8 は有用な情報ではありません。

`parsedate_tz(date)`

`parsedate()` と同じ機能を実現しますが、`None` または 10 要素のタプルを返します; 最初の 9 要素は `time.mktime()` に直接渡すことができるようなタプルで、10 番目の要素はその日のタイムゾーンにおける UTC (グリニッチ標準時の公式名称) からのオフセットです。(タイムゾーンオフセットの符号は、同じタイムゾーンにおける `time.timezone` 変数の符号と反転しています; 後者の変数が POSIX 標準に従っている一方、このモジュールは RFC 2822 に従っているからです。) 入力文字列がタイムゾーン情報を持たない場合、タプルの最後の要素は `None` になります。結果のフィールド 6、7、および 8 は有用な情報ではありません。

`mktime_tz(tuple)`

`parsedata_tz()` が返す 10 要素のタプルを UTC タイムスタンプに変換します。タプル内のタイムゾーン要素が `None` の場合、地域の時刻を表しているものと家庭します。些細な欠陥: この関数はまず最初の 8 要素を地域における時刻として変換し、次にタイムゾーンの違いに対する補償を行います; これにより、夏時間の切り替え日前後でちょっとしたエラーが生じるかもしれません。通常の利用に関しては心配ありません。

参考資料:

email モジュール (12.2 節):

網羅的な電子メール処理パッケージです; `rfc822` モジュールを代替します。

mailbox モジュール (12.4 節):

エンドユーザのメールプログラムによって生成される、様々な mailbox 形式を読み出すためのクラス群。

mimetools モジュール (12.6 節):

MIME エンコードされたメッセージを処理する `rfc822.Message` のサブクラス。

### 12.11.1 Message オブジェクト

Message インスタンスは以下のメソッドを持っています:

`rewindbody()`

メッセージ本体の先頭を seek します。このメソッドはファイルオブジェクトが seek 可能である場合にのみ動作します。

`isheader(line)`

ある行が正しい RFC 2822 ヘッダである場合、その行の正規化されたフィールド名 (インデックス指定の際に使われる辞書キー) を返します; そうでない場合 `None` を返します (解析をここで一度中断し、行データを入力ストリームに押し戻すことを意味します)。このメソッドをサブクラスで上書きすると便利なことがあります。

`islast(line)`

与えられた `line` が `Message` の区切りとなるデリミタであった場合に真を返します。このデリミタ行は消費され、ファイルオブジェクトの読み位置はその直後になります。標準ではこのメソッドは単にその行が空行かどうかをチェックしますが、サブクラスで上書きすることもできます。

`iscomment(line)`

与えられた行全体を無視し、単に読み飛ばすときに真を返します。標準では、これは控えメソッド (stub) であり、常に `False` を返しますが、サブクラスで上書きすることもできます。

`getallmatchingheaders(name)`

`name` に一致するヘッダからなる行のリストがあれば、それらを全て返します。各物理行は連続した行内容であるか否かに関わらず別々のリスト要素になります。`name` に一致するヘッダがない場合、空のリストを返します。

`getfirstmatchingheader(name)`

`name` に一致する最初のヘッダと、その行に連続する (複数) 行からなる行データのリストを返します。`name` に一致するヘッダがない場合 `None` を返します。

`getrawheader(name)`

`name` に一致する最初のヘッダにおけるコロン以降のテキストが入った単一の文字列を返します。このテキストには、先頭の空白、末尾の改行、また後続の行がある場合には途中の改行と空白が含まれます。`name` に一致するヘッダが存在しない場合には `None` を返します。

`getheader(name[, default])`

`getrawheader(name)` に似ていますが、先頭および末尾の空白を剥ぎ取ります。途中にある空白は剥ぎ取られません。オプションの `default` 引数は、`name` に一致するヘッダが存在しない場合に、別のデフォルト値を返すように指定するために使われます。

`get(name[, default])`

正規の辞書との互換性をより高めるための `getheader()` の別名 (alias) です。

`getaddr(name)`

`getheader(name)` が返した文字列を解析して、(`full name`, `email address`) からなるペアを返します。`name` に一致するヘッダが無い場合、(`None`, `None`) が返されます; そうでない場合、`full name` および `address` は (空文字列をとりうる) 文字列になります。

例: `m` に最初の `From:` ヘッダに文字列 '`jack@cwil.nl (Jack Jansen)`' が入っている場合、`m.getaddr('From')` はペア ('`Jack Jansen`', '`jack@cwil.nl`') になります。また、'`Jack Jansen <jack@cwil.nl>`' であっても、全く同じ結果になります。

`getaddrlist(name)`

`getaddr(list)` に似ていますが、複数のメールアドレスからなるリストが入ったヘッダ (例えば `To:` ヘッダ) を解析し、(`full name`, `email address`) のペアからなるリストを (たとえヘッダには一つしかアドレスが入っていなかったとしても) 返します。`name` に一致するヘッダが無かった場合、空のリストを返します。

指定された名前に一致する複数のヘッダが存在する場合 (例えば、複数の `Cc:` ヘッダが存在する場合)、全てのアドレスを解析します。指定されたヘッダが連続する行に収められている場合も解析されます。

`getdate(name)`

`getheader()` を使ってヘッダを取得して解析し、`time.mktime()` と互換な 9 要素のタプルにし

ます; フィールド 6、7、および 8 は有用な値ではないので注意して下さい。 *name* に一致するヘッダが存在しなかったり、ヘッダが解析不能であった場合、None を返します。

日付の解析は妖術のようなものであり、全てのヘッダが標準に従っているとは限りません。このメソッドは多くの発信源から集められた膨大な数の電子メールでテストされており、正しく動作することが分かっていますが、間違った結果を出力してしまう可能性はまだあります。

`getdate_tz(name)`

`getheader()` を使ってヘッダを取得して解析し、10 要素のタプルにします; 最初の 9 要素は `time.mktime()` と互換性のあるタプルを形成し、10 番目の要素はその日におけるタイムゾーンの UTC からのオフセットを与える数字になります。 `getdate()` と同様に、*name* に一致するヘッダがなかったり、解析不能であった場合、None を返します。

Message インスタンスはまた、限定的なマップ型のインタフェースを持っています。すなわち: *m[name]* は *m.getheader(name)* に似ていますが、一致するヘッダがない場合 `KeyError` を送出します; `len(m)`、`m.get(name[, default])`、`m.has_key(name)`、`m.keys()`、`m.values()` *m.items()*、および `m.setdefault(name[, default])` は期待通りに動作します。ただし `setdefault()` は標準の設定値として空文字列をとります。 Message インスタンスはまた、マップ型への書き込みを行えるインタフェース *m[name] = value* および `del m[name]` をサポートしています。 Message オブジェクトでは、`clear()`、`copy()`、`popitem()`、あるいは `update()` といったマップ型インタフェースのメソッドはサポートしていません。( `get()` および `setdefault()` のサポートは Python 2.2 でしか追加されていません。 )

最後に、Message インスタンスはいくつかの public なインスタンス変数を持っています:

**headers**

ヘッダ行のセット全体が、(`setitem` を呼び出して変更されない限り) 読み出された順番に入れられたリストです。各行は末尾の改行を含んでいます。ヘッダを終端する空行はリストに含まれません。

**fp**

インスタンス化の際に渡されたファイルまたはファイル類似オブジェクトです。この値はメッセージ本体を読み出すために使うことができます。

**unixfrom**

メッセージに UNIX 'From ' 行がある場合はその行、そうでなければ空文字列になります。この値は例えば mbox 形式のメールボックスファイルのような、あるコンテキスト中のメッセージを再生成するために必要です。

## 12.11.2 AddressList オブジェクト

AddressList インスタンスは以下のメソッドを持ちます:

`__len__()`

アドレスリスト中のアドレスの数を返します。

`__str__()`

アドレスリストの正規化 (canonicalize) された文字列表現を返します。アドレスはカンマで分割された "name" <host@domain> 形式になります。

`__add__(alist)`

二つの AddressList 被演算子中の双方に含まれるアドレスについて、重複を除いた (集合和の) 全てのアドレスを含む新たな AddressList インスタンスを返します。

`__iadd__(alist)`

`__add__()` のインプレース演算版です; AddressList インスタンスと右側値 *alist* との集合和を取り、その結果をインスタンス自体と置き換えます。

`__sub__(alist)`

左側値の `AddressList` インスタンスのアドレスのうち、右側値中に含まれていないもの全てを含む (集合差分の) 新たな `AddressList` インスタンスを返します。

`__isub__(alist)`

`__sub__()` のインプレース演算版で、`alist` にも含まれているアドレスを削除します。

最後に、`AddressList` インスタンスは `public` なインスタンス変数を一つ持ちます:

`addresslist`

アドレスあたり一つの文字列ペアで構成されるタプルからなるリストです。各メンバ中では、最初の要素は正規化された名前部分で、二つ目は実際の配送アドレス ('@' で分割されたユーザ名 とホスト・ドメインからなるペア) です。

## 12.12 base64 — MIME base64 形式データのエンコードおよびデコード

このモジュールは任意のバイナリ文字列を (e メールや HTTP の POST リクエストの一部としてで安全に送ることのできるテキスト文字列に変換する) base64 形式へエンコードおよびデコードする機能を提供します。エンコードの概要は RFC 1521 (*MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies*, section 5.2, “Base64 Content-Transfer-Encoding”) で定義されていて、MIME 形式の e メールやインターネットのさまざまな場面で利用されています。この形式は `uuencode` プログラムによる出力とは違うものです。たとえば、`'www.python.org'` は、`'d3d3LnB5dGhvb3R5bmc=\n'` とエンコードされます。

`decode(input, output)`

`input` の中身をデコードした結果を `output` に出力します。`input`、`output` ともにファイルオブジェクトか、ファイルオブジェクトと同じインターフェースを持ったオブジェクトである必要があります。`input` は `%codeinput.read()` が空文字列を返すまで読まれます。

`decodestring(s)`

文字列 `s` をデコードして結果のバイナリデータを返します。`s` には一行以上の base64 形式でエンコードされたデータが含まれている必要があります。

`encode(input, output)`

`input` の中身を base64 形式でエンコードした結果を `output` に出力します。`input`、`output` ともにファイルオブジェクトか、ファイルオブジェクトと同じインターフェースを持ったオブジェクトである必要があります。`input` は `%codeinput.read()` が空文字列を返すまで読まれます。`encode()` はエンコードされたデータと改行文字 ('`\n`') を出力します。

`encodestring(s)`

文字列 `s` (任意のバイナリデータを含むことができます) を base64 形式でエンコードした結果の (1 行以上の文字列) データを返します。`encodestring()` はエンコードされた一行以上のデータと改行文字 ('`\n`') を出力します。

参考資料:

`binascii` モジュール ([12.13 節](#)):

ASCII からバイナリへ、バイナリから ASCII への変換をサポートするモジュール。

RFC 1521, “MIME (*Multipurpose Internet Mail Extensions*) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies” Section 5.2, “Base64 Content-Transfer-Encoding,” provides the definition of the base64 encoding.



## 12.13 binascii — バイナリデータと ASCII データとの間での変換

binascii モジュールにはバイナリと ASCII コード化されたバイナリ表現との間の変換を行うための多数のメソッドが含まれています。通常、これらの関数を直接使う必要はなく、uu や binhex といった、ラッパ(wrapper) モジュールを使うことになるでしょう。このモジュールが独立して存在するのは、Python において大量のデータに対するビット操作が低速であるという理由からです。

binascii モジュールでは以下の関数を定義します:

**a2b\_uu**(*string*)

uuencode された 1 行のデータをバイナリに変換し、変換後のバイナリデータを返します。最後の行を除いて、通常 1 行には (バイナリデータで) 45 バイトが含まれます。入力データの先頭には空白文字が連続していてもかまいません。

**b2a\_uu**(*data*)

バイナリデータを uuencode して 1 行の ASCII 文字列に変換します。戻り値は変換後の 1 行の文字列で、改行を含みます。*data* の長さは 45 バイト以下でなければなりません。

**a2b\_base64**(*string*)

base64 でエンコードされたデータのブロックをバイナリに変換し、変換後のバイナリデータを返します。一度に 1 行以上のデータを与えてもかまいません。

**b2a\_base64**(*data*)

バイナリデータを base64 でエンコードして 1 行の ASCII 文字列に変換します。戻り値は変換後の 1 行の文字列で、改行文字を含みます。base64 標準を遵守するためには、*data* の長さは 57 バイト以下でなくてはなりません。

**a2b\_qp**(*string* [, *header*])

quoted-printable 形式のデータをバイナリに変換し、バイナリデータを返します。一度に 1 行以上のデータを渡すことができます。オプション引数 *header* が与えられており、かつその値が真であれば、アンダースコアは空白文字にデコードされます。

**b2a\_qp**(*data* [, *quotetabs*, *istext*, *header*])

バイナリデータを quoted-printable 形式でエンコードして 1 行から複数行の ASCII 文字列に変換します。変換後の文字列を返します。オプション引数 *quptetabs* が存在し、かつその値が真であれば、全てのタブおよび空白文字もエンコードされます。オプション引数 *header* が存在し、かつその値が真であれば、空白文字は RFC1522 に従ってアンダースコアにエンコードされます。オプション引数 *header* が存在し、かつその値が偽である場合、改行文字も同様にエンコードされます。そうでない場合、復帰 (linefeed) 文字の変換によってバイナリデータストリームが破損してしまうかもしれません。

**a2b\_hqx**(*string*)

binhex4 形式の ASCII 文字列データを RLE 展開を行わないでバイナリに変換します。文字列はバイナリのバイトデータを完全に含むような長さか、または (binhex4 データの最後の部分の場合) 余白のビットがゼロになっていなければなりません。

**rledecode\_hqx**(*data*)

*data* に対し、binhex4 標準に従って RLE 展開を行います。このアルゴリズムでは、あるバイトの後ろに 0x90 がきた場合、そのバイトの反復を指示しており、さらにその後ろに反復カウントが続きます。カウントが 0 の場合 0x90 自体を示します。このルーチンは入力データの末端における反復指定が不完全でないかぎり解凍されたデータを返しますが、不完全な場合、例外 *Incomplete* が送出されます。

**rlecode\_hqx**(*data*)

binhex4 方式の RLE 圧縮を *data* に対して行い、その結果を返します。



**b2a\_hqx**(*data*)

バイナリを hexbin4 エンコードして ASCII 文字列に変換し、変換後の文字列を返します。引数の *data* はすでに RLE エンコードされていなければならない、その長さは (最後のフラグメントを除いて) 3 で割り切れなければならない。

**crc\_hqx**(*data*, *crc*)

*data* の binhex4 CRC 値を計算します。初期値は *crc* で、計算結果を返します。

**crc32**(*data*[, *crc*])

32 ビットチェックサムである CRC-32 を *data* に対して計算します。初期値は *crc* です。これは ZIP ファイルのチェックサムと同じです。このアルゴリズムはチェックサムアルゴリズムとして設計されたもので、一般的なハッシュアルゴリズムには向きません。以下のようにして使います:

```
print binascii.crc32("hello world")
# Or, in two pieces:
crc = binascii.crc32("hello")
crc = binascii.crc32(" world", crc)
print crc
```

**b2a\_hex**(*data*)

**hexlify**(*data*)

バイナリデータ *data* の 16 進数表現を返します。*data* の各バイトは対応する 2 桁の 16 進数表現に変換されます。従って、変換結果の文字列は *data* の 2 倍の長さになります。

**a2b\_hex**(*hexstr*)

**unhexlify**(*hexstr*)

16 進数表記の文字列 *hexstr* の表すバイナリデータを返します。この関数は **b2a\_hex**() の逆です。*hexstr* は 16 進数字 (大文字でも小文字でもかまいません) を偶数個含んでいなければならない。そうでないばあい、例外 `TypeError` が送出されます。

**exception Error**

エラーが発生した際に送出される例外です。通常はプログラムのエラーです。

**exception Incomplete**

変換するデータが不完全な場合に送出される例外です。通常はプログラムのエラーではなく、多少追加読み込みを行って再度変換を試みることで対処できます。

参考資料:

base64 モジュール (12.12 節):

MIME 電子メールメッセージで使われる base64 エンコードのサポート。

binhex モジュール (12.14 節):

Macintosh で使われる binhex フォーマットのサポート。

uu モジュール (12.16 節):

UNIX で使われる UU エンコードのサポート。

quopri モジュール (12.15 節):

MIME 電子メールメッセージで使われる quoted-printable エンコードのサポート。

## 12.14 binhex — binhex4 形式ファイルのエンコードおよびデコード

このモジュールは binhex4 形式のファイルに対するエンコードやデコードを行います。binhex4 は Macintosh のファイルを ASCII で表現できるようにしたものです。Macintosh 上では、ファイルと finder 情報の両方のフォークがエンコード (またはデコード) されます。他のプラットフォームではデータフォークだけが処理

されます。

`binhex` モジュールでは以下の関数を定義しています:

**`binhex(input, output)`**

ファイル名 *input* のバイナリファイルをファイル名 *output* の `binhex` 形式ファイルに変換します。*output* パラメタはファイル名でも (`write()` および `close()` メソッドをサポートするような) ファイル様オブジェクトでもかまいません。

**`hexbin(input[, output])`**

`binhex` 形式のファイル *input* をデコードします。*input* はファイル名でも、`write()` および `close()` メソッドをサポートするようなファイル様オブジェクトでもかまいません。変換結果のファイルはファイル名 *output* になります。この引数が省略された場合、出力ファイルは `binhex` ファイルの中から復元されます。

以下の例外も定義されています:

**`exception Error`**

`binhex` 形式を使ってエンコードできなかった場合 (例えば、ファイル名が `filename` フィールドに収まらないくらい長かった場合など) や、入力が正しくエンコードされた `binhex` 形式のデータでなかった場合に出送される例外です。

参考資料:

`binascii` モジュール (12.13 節):

ASCII からバイナリ、およびバイナリから ASCII への変換をサポートするモジュール。

### 12.14.1 注記

別のより強力なエンコーダおよびデコーダへのインタフェースが存在します。詳しくはソースを参照してください。

非 Macintosh プラットフォームでテキストファイルをエンコードしたりデコードしたりする場合でも、Macintosh の改行文字変換 (行末をキャリッジリターンとする) が行われます。

このドキュメントを書いている時点では、`hexbin()` はいつも正しく動作するわけではないようです。

## 12.15 quopri — MIME quoted-printable 形式データのエンコードおよびデコード

このモジュールは RFC 1521: “MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies” で定義されている `quoted-printable` による伝送のエンコードおよびデコードを行います。`quoted-printable` エンコーディングは比較的印字不可能な文字の少ないデータのために設計されています; 画像ファイルを送るときのように印字不可能な文字がたくさんある場合には、`base64` モジュールで利用できる `base64` エンコーディングのほうがよりコンパクトになります。

**`decode(input, output[, header])`**

ファイル *input* の内容をデコードして、デコードされたバイナリデータを ファイル *output* に書き出します。*input* および *output* はファイルか、ファイルオブジェクトのインタフェースを真似たオブジェクトでなければなりません。*input* は `input.readline()` が空文字列を返すまで読みつづけられます。オプション引数 *header* が存在し、かつその値が真である場合、アンダースコアは空白文字にデコードされます。これは RFC 1522: “MIME (Multipurpose Internet Mail Extensions) Part Two: Message

Header Extensions for Non-ASCII Text”で記述されているところの“Q”-エンコードされたヘッダをデコードするのに使われます。

**encode**(*input*, *output*, *quotetabs*)

ファイル *input* の内容をエンコードして、quoted-printable 形式にエンコードされたデータをファイル *output* に書き出します。*input* および *output* はファイルか、ファイルオブジェクトのインタフェースを真似たオブジェクトでなければなりません。*input* は *input.readline()* が空文字列を返すまで読みつづけられます。*quotetabs* はデータ中に埋め込まれた空白文字やタブを変換するかどうか制御するフラグです; この値が真なら、それらの空白をエンコードします。偽ならエンコードせずそのままにしておきます。行末のスペースやタブは RFC 1521 に従って常に変換されるので注意してください。

**decodestring**(*s*[, *header*])

*decode()* に似ていますが、文字列を入力として受け取り、デコードされた文字列を返します。

**encodestring**(*s*[, *quotetabs*])

*encode()* に似ていますが、文字列を入力として受け取り、エンコードされた文字列を返します。*quotetabs* はオプション (デフォルトは 0 です) で、この値はそのまま *encode()* に渡されます。

参考資料:

mimify モジュール (12.9 節):

MIME メッセージを処理するための汎用ユーティリティ。

base64 モジュール (12.12 節):

MIME base64 形式データのエンコードおよびデコード

## 12.16 uu — uuencode 形式のエンコードとデコード

このモジュールではファイルを uuencode 形式 (任意のバイナリデータを ASCII 文字列に変換したもの) にエンコード、デコードする機能を提供します。引数としてファイルが仮定されている所では、ファイルのようなオブジェクトが利用できます。後方互換性のために、パス名を含む文字列も利用できるようにして、対応するファイルを開いて読み書きします。しかし、このインターフェースは利用しないでください。呼び出し側でファイルを開いて (Windows では 'rb' が 'wb' のモードで) 利用する方法が推奨されます。

このコードは Lance Ellinghouse によって提供され、Jack Jansen によって更新されました。

uu モジュールでは以下の関数を定義しています。

**encode**(*in\_file*, *out\_file*[, *name*[, *mode*]])

*in\_file* を *out\_file* にエンコードします。エンコードされたファイルには、デフォルトでデコード時に利用される *name* と *mode* を含んだヘッダがつきます。省略された場合には、*in\_file* から取得された名前が '-' という文字と、0666 がそれぞれデフォルト値として与えられます。

**decode**(*in\_file*[, *out\_file*[, *mode*]])

uuencode 形式でエンコードされた *in\_file* をデコードして *varout\_file* に書き出します。もし *out\_file* がパス名でかつファイルを作る必要があるときには、*mode* がパーミッションの設定に使われます。*out\_file* と *mode* のデフォルト値は *in\_file* のヘッダから取得されます。しかし、ヘッダで指定されたファイルが既に存在していた場合は、*uu.Error* が起きます。

**exception Error**()

*Exception* のサブクラスで、*uu.decode()* によって、さまざまな状況で起きる可能性があります。上で紹介された場合以外にも、ヘッダのフォーマットが間違っている場合や、入力ファイルが途中で区切れた場合にも起きます。

参考資料:

binascii モジュール (12.13 節):

ASCII からバイナリへ、バイナリから ASCII への変換をサポートするモジュール。

## 12.17 xdrlib — XDR データのエンコードおよびデコード

xdrlib モジュールは外部データ表現標準 (External Data Representation Standard) のサポートを実現します。この標準は 1987 年に Sun Microsystems, Inc. によって書かれ、RFC 1014 で定義されています。このモジュールでは RFC で記述されているほとんどのデータ型をサポートしています。

xdrlib モジュールでは 2 つのクラスが定義されています。一つは変数を XDR 表現にパックするためのクラスで、もう一方は XDR 表現からアンパックするためのものです。2 つの例外クラスが同様に定義されています。

**class Packer ( )**

Packer はデータを XDR 表現にパックするためのクラスです。Packer クラスのインスタンス生成は引数なしで行われます。

**class Unpacker ( data )**

Unpacker は Packer と対をなして、文字列バッファから XDR をアンパックするためのクラスです。入力バッファ data を引数に与えてインスタンスを生成します。

参考資料:

RFC 1014, “XDR: External Data Representation Standard”

この RFC が、かつてこのモジュールが最初に書かれたときに XDR 標準でであったデータのエンコード方法を定義していました。現在は RFC 1832 に更新されているようです。

RFC 1832, “XDR: External Data Representation Standard”

こちらが新しい方の RFC で、XDR の改訂版が定義されています。

### 12.17.1 Packer オブジェクト

Packer インスタンスには以下のメソッドがあります:

**get\_buffer ( )**

現在のパック処理用バッファを文字列で返します。

**reset ( )**

パック処理用バッファをリセットして、空文字にします。

一般的には、適切な pack\_type ( ) メソッドを使えば、一般に用いられているほとんどの XDR データをパックすることができます。各々のメソッドは一つの引数を取り、パックしたい値を与えます。単純なデータ型をパックするメソッドとして、以下のメソッド: pack\_uint ( )、pack\_int ( )、pack\_enum ( )、pack\_bool ( )、pack\_uhyper ( ) そして pack\_hyber ( ) がサポートされています。

**pack\_float ( value )**

単精度 (single-precision) の浮動小数点数 value をパックします。

**pack\_double ( value )**

倍精度 (double-precision) の浮動小数点数 value をパックします。

以下のメソッドは文字列、バイト列、不透明データ (opaque data) のパック処理をサポートします:

**pack\_fstring ( n, s )**

固定長の文字列、s をパックします。n は文字列の長さですが、この値自体はデータバッファにはパックされません。4 バイトのアラインメントを保証するために、文字列は必要に応じて null バイト列でパディングされます。

**pack\_fopaque ( n, data )**

`pack_fstring()` と同じく、固定長の不透明データストリームをパックします。

**`pack_string(s)`**

可変長の文字列 *s* をパックします。文字列の長さが最初に符号なし整数でパックされ、続いて `pack_fstring()` を使って文字列データがパックされます。

**`pack_opaque(data)`**

`pack_string()` と同じく、可変長の不透明データ文字列をパックします。

**`pack_bytes(bytes)`**

`pack_string()` と同じく、可変長のバイトストリームをパックします。

以下のメソッドはアレイやリストのパック処理をサポートします:

**`pack_list(list, pack_item)`**

一様な項目からなる *list* をパックします。このメソッドはサイズ不定、すなわち、全てのリスト内容を網羅するまでサイズが分からないリストに対して有用です。リストのすべての項目に対し、最初に符号無し整数 1 がパックされ、続いてリスト中のデータがパックされます。*pack\_item* は個々の項目をパックするために呼び出される関数です。リストの末端に到達すると、符号無し整数 0 がパックされます。

例えば、整数のリストをパックするには、コードは以下のようになるはずです:

```
import xdrlib
p = xdrlib.Packer()
p.pack_list([1, 2, 3], p.pack_int)
```

**`pack_farray(n, array, pack_item)`**

一様な項目からなる固定長のリスト (*array*) をパックします。*n* はリストの長さです。この値はデータバッファにパックされませんが、`len(array)` が *n* と等しくない場合、例外 `ValueError` が送出されます。上と同様に、*pack\_item* は個々の要素をパック処理するための関数です。

**`pack_array(list, pack_item)`**

一様な項目からなる可変長の *list* をパックします。まず、リストの長さが符号無し整数でパックされ、つづいて各要素が上の `pack_farray()` と同じやり方でパックされます。

## 12.17.2 Unpacker オブジェクト

Unpacker クラスは以下のメソッドを提供します:

**`reset(data)`**

文字列バッファを *data* でリセットします。

**`get_position()`**

データバッファ中の現在のアンパック処理位置を返します。

**`set_position(position)`**

データバッファ中のアンパック処理位置を *position* に設定します。`get_position()` および `set_position()` は注意して使わなければなりません。

**`get_buffer()`**

現在のアンパック処理用データバッファを文字列で返します。

**`done()`**

アンパック処理を終了させます。全てのデータがまだアンパックされていないければ、例外 `Error` が送出されます。

上のメソッドに加えて、Packer でパック処理できるデータ型はいずれも Unpacker でアンパック処理できます。アンパック処理メソッドは `unpack_type()` の形式をとり、引数を取りません。これらのメソッドはアンパックされたデータオブジェクトを返します。

**unpack\_float()**

単精度の浮動小数点数をアンパックします。

**unpack\_double()**

`unpack_float()` と同様に、倍精度の浮動小数点数をアンパックします。

上のメソッドに加えて、文字列、バイト列、不透明データをアンパックする以下のメソッドが提供されています:

**unpack\_fstring(*n*)**

固定長の文字列をアンパックして返します。*n* は予想される文字列の長さです。4 バイトのアラインメントを保証するために null バイトによるパディングが行われているものと仮定して処理を行います。

**unpack\_fopaque(*n*)**

`unpack_fstring()` と同様に、固定長の不透明データストリームをアンパックして返します。

**unpack\_string()**

可変長の文字列をアンパックして返します。最初に文字列の長さが符号無し整数としてアンパックされ、次に `unpack_fstring()` を使って文字列データがアンパックされます。

**unpack\_opaque()**

`unpack_string()` と同様に、可変長の不透明データ文字列をアンパックして返します。

**unpack\_bytes()**

`unpack_string()` と同様に、可変長のバイトストリームをアンパックして返します。

以下メソッドはアレイおよびリストのアンパック処理をサポートします。

**unpack\_list(*unpack\_item*)**

一様な項目からなるリストをアンパック処理してかえします。リストは一度に 1 要素ずつアンパック処理されます、まず符号無し整数によるフラグがアンパックされます。もしフラグが 1 なら、要素はアンパックされ、戻り値のリストに追加されます。フラグが 0 であれば、リストの終端を示します。*unpack\_item* は個々の項目をアンパック処理するために呼び出される関数です。

**unpack\_farray(*n*, *unpack\_item*)**

一様な項目からなる固定長のアレイをアンパックして (リストとして) 返します。*n* はバッファ内に存在すると期待されるリストの要素数です。上と同様に、*unpack\_item* は各要素をアンパックするために使われる関数です。

**unpack\_array(*unpack\_item*)**

一様な項目からなる可変長の *list* をアンパックして返します。まず、リストの長さが符号無し整数としてアンパックされ、続いて各要素が上の `unpack_farray()` のようにしてアンパック処理されます。

### 12.17.3 例外

このモジュールでの例外はクラスインスタンスとしてコードされています:

**exception Error**

ベースとなる例外クラスです。Error public なデータメンバとして `msg` を持ち、エラーの詳細が収められています。

**exception ConversionError**

Error から導出されたクラスです。インスタンス変数は塚されていません。



これらの例外を補足する方法を以下の例に示します:

```
import xdrlib
p = xdrlib.Packer()
try:
    p.pack_double(8.01)
except xdrlib.ConversionError, instance:
    print 'packing the double failed:', instance.msg
```

## 12.18 netrc — netrc ファイルの処理

1.5.2 で追加された仕様です。

netrc クラスは、UNIX **ftp** プログラムや他の FTP クライアントで用いられる netrc ファイル形式を解析し、カプセル化 (encapsulate) します。

**class netrc**(*[file]*)

netrc のインスタンスやサブクラスのインスタンスは netrc ファイルのデータをカプセル化します。初期化の際の引数が存在する場合、解析対象となるファイルの指定になります。引数がない場合、ユーザのホームディレクトリ下にある '.netrc' が読み出されます。解析エラーが発生した場合、ファイル名、行番号、解析を中断したトークンに関する情報の入った NetrcParseError を送出します。

**exception NetrcParseError**

ソースファイルのテキスト中で文法エラーに遭遇した場合に netrc クラスによって送出される例外です。この例外のインスタンスは 3 つのインスタンス変数を持っています: msg はテキストによるエラーの説明で、filename はソースファイルの名前、そして lineno はエラーが発見された行番号です。

### 12.18.1 netrc オブジェクト

netrc インスタンスは以下のメソッドを持っています:

**authenticators**(*host*)

*host* の認証情報として、三要素のタプル (*login*, *account*, *password*) を返します。与えられた *host* に対するエントリが netrc ファイルにない場合、'default' エントリに関連付けられたタプルが返されます。*host* に対応するエントリがなく、default エントリもない場合、None を返します。

**\_\_repr\_\_**()

クラスの持っているデータを netrc ファイルの書式に従った文字列で出力します。(コメントは無視され、エントリが並べ替えられる可能性があります。)

netrc のインスタンスは以下の公開されたインスタンス変数を持っています:

**hosts**

ホスト名を (*login*, *account*, *password*) からなるタプルに対応づけている辞書です。'default' エントリがある場合、その名前の擬似ホスト名として表現されます。

**macros**

マクロ名を文字列のリストに対応付けている辞書です。

+ 注意: Passwords are limited to a subset of the ASCII character set. + Versions of this module prior to 2.3 were extremely limited. Starting with + 2.3, all ASCII punctuation is allowed in passwords. However, note that + whitespace and non-printable characters are not allowed in passwords. This + is a limitation of the way the .netrc file is parsed and may be removed in + the future. 注意: 利用可能なパスワードの文字セットは、ASCII のサブ

セットのみです。2.3 より前のバージョンでは厳しく制限されていましたが、2.3 以降では ASCII の記号を使用することができます。しかし、空白文字と印刷不可文字を使用することはできません。この制限は .netrc ファイルの解析方法によるものであり、将来解除されます。

## 12.19 robotparser — robots.txt のためのパーザ

このモジュールでは単一のクラス、`RobotFileParser` を提供します。このクラスは、特定のユーザーエージェントが 'robots.txt' ファイルを公開している Web サイトのある URL を取得可能かどうかの質問に答えます。'robots.txt' ファイルの構造に関する詳細は <http://www.robotstxt.org/wc/norobots.html> を参照してください。

```
class RobotFileParser()
```

このクラスでは単一の 'robots.txt' ファイルを読み出し、解釈し、ファイルの内容に関する質問の回答を得るためのメソッドを定義しています。

```
set_url(url)
```

'robots.txt' ファイルを参照するための URL を設定します。

```
read()
```

'robots.txt' URL を読み出し、パーザに入力します。

```
parse(lines)
```

引数 *lines* の内容を解釈します。

```
can_fetch(useragent, url)
```

解釈された 'robots.txt' ファイル中に記載された規則に従ったとき、*useragent* が *url* を取得してもよい場合には `True` を返します。

```
mtime()
```

`robots.txt` ファイルを最後に取得した時刻を返します。この値は、定期的に新たな `robots.txt` をチェックする必要がある、長時間動作する Web スパイダープログラムを実装する際に便利です。

```
modified()
```

`robots.txt` ファイルを最後に取得した時刻を現在の時刻に設定します。

以下に `RobotFileParser` クラスの利用例を示します。

```
>>> import robotparser
>>> rp = robotparser.RobotFileParser()
>>> rp.set_url("http://www.musi-cal.com/robots.txt")
>>> rp.read()
>>> rp.can_fetch("*", "http://www.musi-cal.com/cgi-bin/search?city=San+Francisco")
False
>>> rp.can_fetch("*", "http://www.musi-cal.com/")
True
```

## 12.20 csv — CSV ファイルの読み書き

2.3 で追加された仕様です。

CSV (Comma Separated Values、カンマ区切り値列) と呼ばれる形式は、スプレッドシートやデータベース間でのデータのインポートやエクスポートにおける最も一般的な形式です。“CSV 標準”は存在しないた

め、CSV 形式はデータを読み書きする多くのアプリケーション上の操作に応じて定義されているにすぎません。標準がないということは、異なるアプリケーションによって生成されたり取り込まれたりするデータ間では、しばしば微妙な違いが発生するということを意味します。こうした違いのために、複数のデータ源から得られた CSV ファイルを処理する作業が鬱陶しいものになることがあります。とはいえ、デリミタ (delimiter) やクオート文字の相違はあっても、全体的な形式は十分似通っているため、こうしたデータを効率的に操作し、データの読み書きにおける細々としたことをプログラマから隠蔽するような単一のモジュールを書くことは可能です。

csv モジュールでは、CSV 形式で書かれたテーブル状のデータを読み書きするためのクラスを実装しています。このモジュールを使うことで、プログラマは Excel で使われている CSV 形式に関して詳しい知識をもっていなくても、“このデータを Excel で推奨されている形式で書いてください”とか、“データを Excel で作成されたこのファイルから読み出してください”と言うことができます。プログラマはまた、他のアプリケーションが解釈できる CSV 形式を記述したり、独自の特殊な目的をもった CSV 形式を定義することができます。

csv モジュールの reader および writer オブジェクトはシーケンス型を読み書きします。プログラマは DictReader や DictWriter クラスを使うことで、データを辞書形式で読み書きすることもできます。

注意: このバージョンの csv モジュールは Unicode 入力をサポートしていません。また、現在のところ、ASCII NUL 文字に関連したいくつかの問題があります。従って、安全を期すには、全ての入力を一般的には印字可能な ASCII にしなければなりません。これらの制限は将来取り去られることになっています。

参考資料:

PEP 305, “*CSV File API*”

Python へのこのモジュールの追加を提案している Python 改良案 (PEP: Python Enhancement Proposal)

## 12.20.1 モジュールの内容

csv モジュールでは以下の関数を定義しています:

**reader**(*csvfile*[, *dialect*=‘excel’[, *fntparam*]])

与えられた *csvfile* 内の行を反復処理するような reader オブジェクトを返します。*csvfile* はイテレータプロトコルをサポートし、next メソッドが呼ばれた際に常に文字列を返すような任意のオブジェクトにすることができます。*csvfile* がファイルオブジェクトの場合、ファイルオブジェクトの形式に違いがあるようなプラットフォームでは ‘b’ フラグを付けて開かなければなりません。オプションとして *dialect* パラメタを与えることができ、特定の CSV 表現形式 (dialect) 特有のパラメタの集合を定義するために使われます。*dialect* パラメタは Dialect クラスのサブクラスのインスタンスか、list\_dialects 関数が返す文字列の一つにすることができます。別のオプションである *fntparam* キーワード引数は、現在の表現形式における個々の書式パラメタを上書きするために与えることができます。表現形式および書式化パラメタの詳細については、12.20.2 節、“Dialect クラスと書式化パラメタ”を参照してください。

読み出されたデータは全て文字列として返されます。データ型の変換が自動的に行われることはありません。

**writer**(*csvfile*[, *dialect*=‘excel’[, *fntparam*]])

ユーザが与えたデータをデリミタで区切られた文字列に変換し、与えられたファイルオブジェクトにするための writer オブジェクトを返します。*csvfile* は write メソッドを持つ任意のオブジェクトでかまいません。*csvfile* がファイルオブジェクトの場合、ファイルオブジェクトの形式に違いがあるようなプラットフォームでは ‘b’ フラグを付けて開かなければなりません。オプションとして *dialect* パラメタを与えることができ、特定の CSV 表現形式 (dialect) 特有のパラメタの集合を定義するために

使われます。*dialect* パラメタは `Dialect` クラスのサブクラスのインスタンスか、`list_dialects` 関数が返す文字列の一つにすることができます。別のオプションである *fntparam* キーワード引数は、現在の表現形式における個々の書式パラメタを上書きするために与えることができます。表現形式および書式化パラメタの詳細については、[12.20.2 節](#)、“*Dialect* クラスと書式化パラメタ”を参照してください。DB API を実装するモジュールとのインタフェースを可能な限り容易にするために、`None` は空文字列として書き込まれます。この処理は可逆な変換ではありませんが、SQL で `NULL` データ値を CSV にダンプする処理を、`cursor.fetch*()` 呼び出しによって返されたデータを前処理することなく簡単に行うことができます。他の非文字データは、書き出される前に `str()` を使って文字列に変換されます。

`register_dialect(name, dialect)`

*dialect* を *name* と関連付けます。*dialect* は `csv.Dialect` のサブクラスでなければなりません。*name* は文字列か Unicode オブジェクトでなければなりません。

`unregister_dialect(name)`

*name* に関連づけられた表現形式を表現形式レジストリから削除します。*name* が表現形式名でない場合には `Error` を送出します。

`get_dialect(name)`

*name* に関連づけられた表現形式を返します。*name* が表現形式名でない場合には `Error` を送出します。

`list_dialects()`

登録されている全ての表現形式を返します。

CSV モジュールでは以下のクラスを定義しています:

`class DictReader(csvfile, fieldnames[, restkey=None[, restval=None[, dialect='excel'[, fntparam]]]])`

*fieldnames* パラメタで与えられたキーを読み出された情報に対応付ける他は正規の reader のように動作するオブジェクトを生成します。読み出された行が *fieldnames* の配列よりも多くのフィールドを持っていた場合、残りのフィールドデータは *restkey* の値をキーとする配列に追加されます。読み出された行が *fieldnames* の配列よりも少ないフィールドしか持たない場合、残りのキーはオプションの *restval* パラメタに指定された値を取ります。その他のパラメタは reader オブジェクトの場合と同様に解釈されます。

`class DictWriter(csvfile, fieldnames[, restval=""][, extrasaction='raise'[, dialect='excel'[, fntparam]]]])`

辞書を出力行に対応付ける他は正規の writer のように動作するオブジェクトを生成します。*fieldnames* パラメタには、辞書中の `writerow()` メソッドに渡される値がどの順番で *csvfile* に書き出されるかを指定します。オプションの *restval* パラメタは、*fieldnames* 内のキーが辞書中にない場合に書き出される値を指定します。`writerow()` メソッドに渡された辞書に、*fieldnames* 内には存在しないキーが入っている場合、オプションの *extraaction* パラメタでどのような動作を行うかを指定します。この値が `'raise'` に設定されている場合 `ValueError` が送出されます。`'ignore'` に設定されている場合、辞書の余分の値は無視されます。その他のパラメタは writer オブジェクトの場合と同様に解釈されます。

`class Dialect`

`Dialect` クラスはコンテナクラスで、基本的な用途としては、その属性を特定の reader や writer インスタンスのパラメタを定義するために用います。

`class Sniffer([sample=16384])`

`Sniffer` クラスは CSV ファイルの書式を推理するために用いられるクラスです。

`Sniffer` クラスではメソッドを一つ提供しています:

**sniff**(*fileobj*)

与えられた *sample* を解析し、発見されたパラメタを反映した `Dialect` サブクラスを返します。オプションの *delimiters* パラメタを与えた場合、有効なデリミタ文字を含んでいるはずの文字列として解釈されます。

**has\_header**(*sample*)

(CSV 形式と仮定される) サンプルテキストを解析して、最初の行がカラムヘッダの羅列のように推察される場合 `True` を返します。

`csv` モジュールでは以下の定数を定義しています:

**QUOTE\_ALL**

*writer* オブジェクトに対し、全てのフィールドをクオートするように指示します。

**QUOTE\_MINIMAL**

*writer* オブジェクトに対し、現在の *delimiter* を含むか、あるいは *quotechar* で始まるフィールドだけをクオートするように指示します。

**QUOTE\_NONNUMERIC**

*writer* オブジェクトに対し、全ての非数値フィールドをクオートするように指示します。

**QUOTE\_NONE**

*writer* オブジェクトに対し、フィールドを決してクオートしないように指示します。現在の *delimiter* が出力データ中に現れた場合、現在設定されている *escapechar* 文字が前に付けられます。`QUOTE_NONE` の効果下にある時には、1 文字からなる文字列 *escapechar* が定義されていないと、たとえ書き出されるデータ中に *delimiter* 文字が入っていてもエラーになります。

`csv` モジュールでは以下の例外を定義しています:

**exception Error**

全ての関数において、エラーが検出された際に送出される例外です。

## 12.20.2 Dialect クラスと書式化パラメタ

レコードに対する入出力形式の指定をより簡単にするために、特定の書式化パラメタは表現形式 (*dialect*) にまとめてグループ化されます。表現形式は `Dialect` クラスのサブクラスで、様々なクラス特有のメソッドと、`validate()` メソッドを一つ持っています。`reader` または `writer` オブジェクトを生成するとき、プログラマは文字列または `Dialect` クラスのサブクラスを表現形式パラメタとして渡さなければなりません。さらに、*dialect* パラメタの代りに、プログラマは上で定義されている属性と同じ名前を持つ個々の書式化パラメタを `Dialect` クラスに指定することができます。

`Dialect` は以下の属性をサポートしています:

**delimiter**

フィールド間を分割するのに用いられる 1 文字からなる文字列です。デフォルトでは `,` です。

**doublequote**

フィールド内に現れた *quotechar* のインスタンスで、クオートではないその文字自身でなければならない文字をどのようにクオートするかを制御します。`True` の場合、この文字は二重化されます。`False` の場合、*escapechar* は 1 文字からなる文字列でなければならない、*quotechar* の前に置かれます。デフォルトでは `True` です。

**escapechar**

*quoting* が `QUOTE_NONE` に設定されている場合に、*delimiter* をエスケープするために用いられる、1 文字からなる文字列です。デフォルトでは `None` です。

**lineterminator**

CSV ファイルの各行を終端する際に用いられる文字列です。デフォルトでは `'\r\n'` です。

#### **quotechar**

*delimiter* を含むか、*quotechar* から始まる要素をクオートする際に用いられる 1 文字からなる文字列です。デフォルトでは `'` です。

#### **quoting**

writer によってクオートがいつ生成されるかを制御します。QUOTE\_\* 定数のいずれか (12.20.1 節参照) をとることができ、デフォルトでは QUOTE\_MINIMAL です。

#### **skipinitialspace**

True の場合、*delimiter* の直後に続く空白は無視されます。デフォルトでは False です。

### 12.20.3 reader オブジェクト

reader オブジェクト (DictReader インスタンス、および reader() 関数によって返されたオブジェクト) は、以下の public なメソッドを持っています:

#### **next()**

reader の反復可能なオブジェクトから、現在の表現形式に基づいて次の行を解析して返します。

### 12.20.4 writer オブジェクト

writer オブジェクト (DictWriter インスタンス、および writer() 関数によって返されたオブジェクト) は、以下の public なメソッドを持っています:

#### **writerow(row)**

*row* パラメタを現在の表現形式に基づいて書式化し、writer のファイルオブジェクトに書き込みます。

#### **writerows(rows)**

*rows* パラメタ全てを現在の表現形式に基づいて書式化し、writer のファイルオブジェクトに書き込みます。

### 12.20.5 使用例

csv 読み出しの “Hello, world” プログラムは以下のようになります。

```
import csv
reader = csv.reader(file("some.csv"))
for row in reader:
    print row
```

上に対して、単純な書き込みのプログラム例は以下のようになります。

```
import csv
writer = csv.writer(file("some.csv", "w"))
for row in someiterable:
    writer.writerow(row)
```



# 構造化マークアップツール

Python は様々な構造化データマークアップ形式を扱うための、様々なモジュールをサポートしています。これらは標準化一般マークアップ言語 (SGML) およびハイパーテキストマークアップ言語 (HTML)、そして可拡張性マークアップ言語 (XML) を扱うためのいくつかのインタフェースからなります。

注意すべき重要な点として、`xml` パッケージは少なくとも一つの SAX に対応した XML パーザが利用可能でなければなりません。Python 2.3 からは Expat パーザが Python に取り込まれているので、`xml.parsers.expat` モジュールは常に利用できます。また、PyXML 追加パッケージについても知りたいと思うかもしれませんが; このパッケージは Python 用の拡張された XML ライブラリセットを提供します。

`xml.dom` および `xml.sax` パッケージのドキュメントは Python による DOM および SAX インタフェースへのバインディングに関する定義です。

<b>HTMLParser</b>	HTML と XHTML を扱えるシンプルなパーザ。
<b>sgmllib</b>	HTML を解析するのに必要な機能だけを備えた SGML パーザ。
<b>htmllib</b>	HTML 文書の解析器。
<b>htmlentitydefs</b>	HTML 一般エンティティの定義。
<b>xml.parsers.expat</b>	Expat による、検証を行わない XML パーザへのインタフェース
<b>xml.dom</b>	Python のための文書オブジェクトモデル API。
<b>xml.dom.minidom</b>	軽量な文書オブジェクトモデルの実装。
<b>xml.dom.pulldom</b>	SAX イベントからの部分的な DOM ツリー構築のサポート。
<b>xml.sax</b>	SAX2 基底クラスと有用な関数のパッケージ
<b>xml.sax.handler</b>	SAX イベント・ハンドラの基底クラス
<b>xml.sax.saxutils</b>	SAX とともに使う有用な関数とクラスです。
<b>xml.sax.xmlreader</b>	SAX 準拠の XML パーサが実装すべきインターフェースです。
<b>xmlilib</b>	XML ドキュメントのパーサ。

参考資料:

Python/XML ライブラリ

(<http://pyxml.sourceforge.net/>)

Python にバンドルされてくる `xml` パッケージへの拡張である PyXML パッケージのホームページです。

## 13.1 HTMLParser — HTML および XHTML のシンプルなパーザ

このモジュールでは `HTMLParser` クラスを定義します。このクラスは HTML (ハイパーテキスト記述言語、HyperText Mark-up Language) および XHTML で書式化されているテキストファイルを解釈するための基礎となります。`htmllib` にあるパーザと違って、このパーザは `sgmllib` の SGML パーザに基づいてはいません。

```
class HTMLParser( )
```

HTMLParser クラスは引数なしでインスタンス化します。

HTMLParser インスタンスに HTML データが入力されると、タグが開始したとき、及び終了したときに関数を呼び出します。HTMLParser クラスは、ユーザが行いたい動作を提供するために上書きできるようになっています。

htmlllib のパーザと違い、このパーザは終了タグが開始タグと一致しているか調べたり、外側のタグ要素が閉じるときに内側で明示的に閉じられていないタグ要素のタグ終了ハンドラを呼び出したりはしません。

HTMLParser インスタンスは以下のメソッドを提供します:

**reset()**

インスタンスをリセットします。未処理のデータは全て失われます。インスタンス化の際に非明示的に呼び出されます。

**feed(data)**

パーザにテキストを入力します。入力が完全なタグ要素で構成されている場合に限り処理が行われます; 不完全なデータであった場合、新たにデータが入力されるか、close() が呼び出されるまでバッファされます。

**close()**

全てのバッファされているデータについて、その後にファイル終了マークが続いているとみなして強制的に処理を行います。このメソッドは入力データの終端で行うべき追加処理を定義するために導出クラスで上書きすることができますが、再定義を行ったクラスでは常に、HTMLParser 基底クラスのメソッド close() を呼び出さなくてはなりません。

**getpos()**

現在の行番号およびオフセット値を返します。

**get\_starttag\_text()**

最も最近開かれた開始タグのテキスト部分を返します。このテキストは必ずしも元データを構造化する上で必須ではありませんが、“広く知られている (as deployed)” HTML を扱ったり、入力を最小限の変更で再生成 (属性間の空白をそのままにする、など) したりする場合に便利ことがあります。

**handle\_starttag(tag, attrs)**

このメソッドはタグの開始部分を処理するために呼び出されます。導出クラスで上書きするためのメソッドです; 基底クラスの実装では何も行いません。

tag 引数はタグの名前で、小文字に変換されています。attrs 引数は (name, value) のペアからなるリストで、タグの <> 括弧内にある属性が収められています。name は小文字に変換され、value 内のエンティティ参照は変換されます。二重引用符やバックスラッシュは変換しません。例えば、タグ <A HREF="http://www.cwi.nl/"> を処理する場合、このメソッドは 'handle\_starttag('a', [( 'href', 'http://www.cwi.nl/' )])' として呼び出されます。

**handle\_startendtag(tag, attrs)**

handle\_starttag() と似ていますが、パーザが XHTML 形式の空タグ (<a .../>) に遭遇した場合に呼び出されます。この特定の語彙情報 (lexical information) が必要な場合、このメソッドをサブクラスで上書きすることができます; 標準の実装では、単に handle\_starttag() および handle\_endtag() を呼ぶだけです。

**handle\_endtag(tag)**

このメソッドはあるタグ要素の終了タグを処理するために呼び出されます。導出クラスで上書きするためのメソッドです; 基底クラスの実装では何も行いません。tag 引数はタグの名前で、小文字に変換されています。

**handle\_data(data)**

このメソッドは、他のメソッドに当てはまらない任意のデータを処理するために呼び出されます。導出クラスで上書きするためのメソッドです; 基底クラスの実装では何も行いません。

`handle_charref(ref)`

このメソッドはタグ外の '`&#ref;`' 形式の文字参照 (character reference) を処理するために呼び出されます。`ref` には、先頭の '`&#`' および末尾の '`;`' は含まれません。導出クラスで上書きするためのメソッドです; 基底クラスの実装では何も行いません。

`handle_entityref(name)`

このメソッドはタグ外の '`&name;`' 形式の一般のエンティティ参照 (entity reference) `name` を処理するために呼び出されます。`name` には、先頭の '`&`' および末尾の '`;`' は含まれません。導出クラスで上書きするためのメソッドです; 基底クラスの実装では何も行いません。

`handle_comment(data)`

このメソッドはコメントに遭遇した場合に呼び出されます。`comment` 引数は文字列で、'`<!--`' and '`-->`' デリミタ間の、デリミタ自体を除いたテキストが収められています。例えば、コメント '`<!--text-->`' があると、このメソッドは引数 '`text`' で呼び出されます。導出クラスで上書きするためのメソッドです; 基底クラスの実装では何も行いません。

`handle_decl(decl)`

パーザが SGML 宣言を読み出した際に呼び出されるメソッドです。`decl` パラメタは `<!...>` 記述内の宣言内容全体になります。導出クラスで上書きするためのメソッドです; 基底クラスの実装では何も行いません。

`handle_pi(data)`

処理指令に遭遇した場合に呼び出されます。`data` には、処理指令全体が含まれ、例えば `<?proc color='red'>` という処理指令の場合、`handle_pi("proc color='red'")` のように呼び出されます。このメソッドは導出クラスで上書きするためのメソッドです; 基底クラスの実装では何も行いません。

注意: The `HTMLParser` クラスでは、処理指令に SGML の構文を使用します。末尾に '`?`' が XHTML の処理指令では、'`?`' が `data` に含まれます。

**exception `HTMLParseError`**

HTML の構文に沿わないパターンを発見したときに送出される例外です。HTML 構文法上の全てのエラーを発見できるわけではないので注意してください。

### 13.1.1 HTML パーザアプリケーションの例

基礎的な例として、`HTMLParser` クラスを使い、発見したタグを出力する、非常に基礎的な HTML パーザを以下に示します。

```
from HTMLParser import HTMLParser

class MyHTMLParser(HTMLParser):

    def handle_starttag(self, tag, attrs):
        print "Encountered the beginning of a %s tag" % tag

    def handle_endtag(self, tag):
        print "Encountered the end of a %s tag" % tag
```

## 13.2 sgmllib — 単純な SGML パーザ

このモジュールでは SGML (Standard Generalized Mark-up Language: 汎用マークアップ言語標準) で書式化されたテキストファイルを解析するための基礎として働く `SGMLParser` クラスを定義しています。実際には、このクラスは完全な SGML パーザを提供しているわけではありません— このクラスは HTML で用いられているような SGML だけを解析し、モジュール自体も `htmllib` モジュールの基礎にするためだけに存在しています。XHTML をサポートし、少し異なったインタフェースを提供しているもう一つの HTML パーザは、`HTMLParser` モジュールで使うことができます。

`class SGMLParser()`

`SGMLParser` クラスは引数無しでインスタンス化されます。このパーザは以下の構成を認識するようにハードコードされています:

- `<tag attr="value" ...>` と `</tag>` で表されるタグの開始部と終了部。
- `&#name;` 形式をとる文字の数値参照。
- `&name;` 形式をとるエンティティ参照。
- `<!--text-->` 形式をとる SGML コメント。末尾の `>` とその直前にある `--` の間にはスペース、タブ、改行を入れることができます。

`SGMLParser` インスタンスは以下のインタフェースメソッドを持っています:

`reset()`

インスタンスをリセットします。未処理のデータは全て失われます。このメソッドはインスタンス生成時に非明示的に呼び出されます。

`setnomoretags()`

タグの処理を停止します。以降の入力をリテラル入力 (CDATA) として扱います。(この機能は古い HTML タグ `<PAINTTEXT>` を実装できるようにするためだけに提供されています)

`setliteral()`

リテラルモード (CDATA モード) に移行します。

`feed(data)`

テキストをパーザに入力します。入力は完全なエレメントから成り立つ場合に限り処理されます; 不完全なデータは追加のデータが入力されるか、`close()` が呼び出されるまでバッファに蓄積されます。

`close()`

バッファに蓄積されている全てのデータについて、直後にファイル終了記号が来た時のようにして強制的に処理します。このメソッドは導出クラスで再定義して、入力の終了時に追加の処理を行うよう定義することができますが、このメソッドの再定義されたバージョンでは常に `close()` を呼び出さなければなりません。

`get_starttag_text()`

もっとも最近開かれた開始タグのテキストを返します。通常、構造化されたデータの処理をする上でこのメソッドは必要ありませんが、“広く知られている (as deployed)” HTML を扱ったり、入力を最小限の変更で再生成 (属性間の空白をそのままにする、など) したりする場合に便利ことがあります。

`handle_starttag(tag, method, attributes)`

このメソッドは `start_tag()` か `do_tag()` のどちらかのメソッドが定義されている開始タグを処理するために呼び出されます。`tag` 引数はタグの名前で、小文字に変換されています。`method` 引数は開始タグの意味解釈をサポートするために用いられるバインドされたメソッドです。`attributes` 引数は `(name, value)` のペアからなるリストで、タグの `<>` 括弧内にある属性が収められています。`name` は小文字に変換され、`value` 内の二重引用符とバックスラッシュも変換されます。例えば、タグ `<A`

HREF="http://www.cwi.nl/"> を処理する場合、このメソッドは `unknown_starttag('a', [( 'href', 'http://www.cwi.nl/' )])` として呼び出されます。基底クラスの実装では、単に `method` を単一の引数 `attributes` と共に呼び出します。

`handle_endtag(tag, method)`

このメソッドは `end_tag()` メソッドの定義されている終了タグを処理するために呼び出されます。`tag` 引数はタグの名前で、小文字に変換されており、`method` 引数は終了タグの意味解釈をサポートするために使われるバインドされたメソッドです。`end_tag()` メソッドが終了エレメントとして定義されていない場合、ハンドラは一切呼び出されません。基底クラスの実装では単に `method` を呼び出します。

`handle_data(data)`

このメソッドは何らかのデータを処理するために呼び出されます。導出クラスで上書きするためのメソッドです; 基底クラスの実装では何も行いません。

`handle_charref(ref)`

このメソッドは `&#ref;` 形式の文字参照 (character reference) を処理するために呼び出されます。基底クラスの実装では、`ref` は 0-255 の範囲の 10 進数でなければなりません。このメソッドは文字を ASCII に変換し、その文字を引数として `handle_data()` を呼び出します。`ref` が無効な値か、範囲を超えた値である場合、エラーを処理するために `unknown_charref(ref)` が呼び出されます。名前づけされた文字エンティティをサポートするためにはこのメソッドをサブクラスで上書きしなければなりません。

`handle_entityref(ref)`

このメソッドは `ref` を一般エンティティ参照として、`&ref;` 形式のエンティティ参照を処理するために呼び出されます。

このメソッドは、エンティティ名を対応する変換文字に対応付けているインスタンス (またはクラス) 変数である `entitydefs` 中で `ref` を探します。変換が見つかった場合、変換された文字を引数にして `handle_data()` を呼び出します; そうでない場合、`unknown_entityref(ref)` を呼び出します。標準では `entitydefs` は `&amp;`、`&apos;`、`&gt;`、`&lt;`、および `&quot;` の変換を定義しています。

`handle_comment(comment)`

このメソッドはコメントに遭遇した場合に呼び出されます。`comment` 引数は文字列で、`<!--` and `-->` デリミタ間の、デリミタ自体を除いたテキストが収められています。例えば、コメント `<!--text-->` があると、このメソッドは引数 `'text'` で呼び出されます。基底クラスの実装では何も行いません。

`handle_decl(data)`

パーザが SGML 宣言を読み出した際に呼び出されるメソッドです。実際には、DOCTYPE は HTML だけに見られる宣言ですが、パーザは宣言間の相違 (や誤った宣言) を判別しません。DOCTYPE の内部サブセット宣言はサポートされていません。`decl` パラメタは `<!...>` 記述内の宣言内容全体になります。基底クラスの実装では何も行いません。

`report_unbalanced(tag)`

個のメソッドは対応する開始エレメントのない終了タグが発見された時に呼び出されます。

`unknown_starttag(tag, attributes)`

未知の開始タグを処理するために呼び出されるメソッドです。導出クラスで上書きするためのメソッドです; 基底クラスの実装では何も行いません。

`unknown_endtag(tag)`

This method is called to process an unknown end tag. 未知の終了タグを処理するために呼び出されるメソッドです。導出クラスで上書きするためのメソッドです; 基底クラスの実装では何も行いません。



`unknown_charref(ref)`

このメソッドは解決不能な文字参照数値を処理するために呼び出されます。標準で何が処理可能かは `handle_charref()` を参照してください。導出クラスで上書きするためのメソッドです; 基底クラスの実装では何も行いません。

`unknown_entityref(ref)`

未知のエンティティ参照を処理するために呼び出されるメソッドです。導出クラスで上書きするためのメソッドです; 基底クラスの実装では何も行いません。

上に挙げたメソッドを上書きしたり拡張したりするのは別に、導出クラスでは以下の形式のメソッドを定義して、特定のタグを処理することもできます。入力ストリーム中のタグ名は大小文字の区別に依存しません; メソッド名中の *tag* は小文字でなければなりません:

`start_tag(attributes)`

このメソッドは開始タグ *tag* を処理するために呼び出されます。 `do_tag()` よりも高い優先順位があります。 *attributes* 引数は上の `handle_starttag()` で記述されているのと同じ意味です。

`do_tag(attributes)`

このメソッドは対応する終了タグのない開始タグ *tag* を処理するために呼び出されます。 *attributes* 引数は上の `handle_starttag()` で記述されているのと同じ意味です。

`end_tag()`

このメソッドは終了タグ *tag* を処理するために呼び出されます。

パーザは開始されたエレメントのうち、終了タグがまだ見つからないもののスタックを維持しているので注意してください。 `start_tag()` で処理されたタグだけがスタックにプッシュされます。 are pushed on this stack. Definition of an それらのタグに対する `end_tag()` メソッドの定義はオプションです。 `do_tag()` や `unknown_tag()` で処理されるタグについては、 `end_tag()` を定義してはいけません; 定義されていても使われることはありません。あるタグに対して `start_tag` および `do_tag()` メソッドの両方が存在する場合、 `start_tag()` が優先されます。

## 13.3 htmllib — HTML 文書の解析器

このモジュールでは、ハイパーテキスト記述言語 (HTML, HyperText Mark-up Language) 形式で書式化されたテキストファイルを解析するための基盤として役立つクラスを定義しています。このクラスは I/O と直接的には接続されません — このクラスにはメソッドを介して文字列形式の入力を提供する必要があり、出力を生成するには “フォーマッタ (formatter)” オブジェクトのメソッドを何度か呼び出さなくてはなりません。

HTMLParser クラスは、機能を追加するために他のクラスの基底クラスとして利用するように設計されており、ほとんどのメソッドが拡張したり上書きしたりできるようになっています。さらにこのクラスは sgmlib モジュールで定義されている SGMLParser クラスから導出されており、その機能を拡張しています。HTMLParser の実装は、RFC 1866 で解説されている HTML 2.0 記述言語をサポートします。formatter では 2 つのフォーマッタオブジェクト実装が提供されています; フォーマッタのインタフェースについての情報は formatter モジュールのドキュメントを参照してください。

以下は sgmlib.SGMLParser で定義されているインタフェースの概要です:

- インスタンスにデータを与えるためのインタフェースは `feed()` メソッドで、このメソッドは文字列を引数に取ります。このメソッドに一度に与えるテキストは必要に応じて多くも少なくもできます; というのは ‘`p.feed(a);p.feed(b)`’ は ‘`p.feed(a+b)`’ と同じ効果を持つからです。与えられたデータが完全な HTML タグ群を含む場合、それらのタグは即座に処理されます; 不完全な要素はバッファに保存されます。全ての未処理データを強制的に処理させるには、 `close()` メソッドを呼び出します。



例えば、ファイルの全内容を解析するには:

```
parser.feed(open('myfile.html').read())
parser.close()
```

のようにします。

- HTML タグに対して意味付けを定義するためのインタフェースはとても単純です: サブクラスを導出して、`start_tag()`、`end_tag()`、あるいは `do_tag()` といったメソッドを定義するだけです。パーザはこれらのメソッドを適切なタイミングで呼び出します: `start_tag` や `do_tag()` は `<tag ...>` の形式の開始タグに遭遇した時に呼び出されます; `end_tag()` は `<tag>` の形式の終了タグに遭遇した時に呼び出されます。 `<H1> ... </H1>` のように開始タグが終了タグと対応している必要がある場合、クラス中で `start_tag()` が定義されていなければなりません; `<P>` のように終了タグが必要ない場合、クラス中では `do_tag()` を定義しなければなりません。

このモジュールではクラスを一つだけ定義しています:

**class HTMLParser** (*formatter*)

基底となる HTML パーザクラスです。HTML 2.0 仕様 (RFC 1866) が要求している全てのエンティティ名をサポートしています。このクラスではまた、HTML 2.0 の全てのタグ要素と HTML 3.0 および 3.2 の多くのタグ要素に対するハンドラを定義しています。

参考資料:

`formatter` モジュール (12.1 節):

抽象化された書式イベントの流れを `writer` オブジェクト上の特定の出力イベントに変換するためのインターフェース。

`HTMLParser` モジュール (13.1 節):

HTML パーザのひとつです。やや低いレベルでしか入力を扱えませんが、XHTML を扱うことができるように設計されています。“広く知られている HTML (HTML as deployed)” では使われておらずかつ XHTML では正しくないと言われる SGML 構文のいくつかは実装されていません。

`htmlentitydefs` モジュール (13.4 節):

HTML 2.0 エンティティに対する置換テキストの定義。

`sgmllib` モジュール (13.2 節):

`HTMLParser` の基底クラス。

### 13.3.1 HTMLParser オブジェクト

タグメソッドに加えて、`HTMLParser` クラスではタグメソッドで利用するためのいくつかのメソッドとインスタンス変数を提供しています。

**formatter**

パーザに関連付けられているフォーマッタインスタンスです。

**nofill**

ブール値のフラグで、空白文字を縮約したくないときには真、縮約するときには偽にします。一般的には、この値を真にするのは、`<PRE>` 要素の中のテキストのように、文字列データが“書式化済みの (preformatted)” 場合だけです。標準の値は偽です。この値は `handle_data()` および `save_end()` の操作に影響します。

**anchor\_bgn** (*href, name, type*)

このメソッドはアンカー領域の先頭で呼び出されます。引数は `<A>` タグの属性で同じ名前を持つもの

に対応します。標準の実装では、ドキュメント内のハイパーリンク (<A> タグの HREF 属性) を列挙したリストを維持しています。ハイパーリンクのリストはデータ属性 `anchorlist` で手に入れることができます。

**anchor\_end()**

このメソッドはアンカー領域の末尾で呼び出されます。標準の実装では、テキストの注釈マーカを追加します。マーカは `anchor_bgn()` で作られたハイパーリンクリストのインデクス値です。

**handle\_image(source, alt[, ismap[, align[, width[, height]]]])**

このメソッドは画像を扱うために呼び出されます。標準の実装では、単に `handle_data()` に `alt` の値を渡すだけです。

**save\_bgn()**

文字列データをフォーマットオブジェクトに送らずにバッファに保存する操作を開始します。保存されたデータは `save_end()` で取得してください。 `save_bgn()` / `save_end()` のペアを入れ子構造にすることはできません。

**save\_end()**

文字列データのバッファリングを終了し、以前 `save_bgn()` を呼び出した時点から保存されている全てのデータを返します。 `nofill` フラグが偽の場合、空白文字は全てスペース文字一文字に置き換えられます。予め `save_bgn()` を呼ばないでこのメソッドを呼び出すと `TypeError` 例外が送出されます。

## 13.4 `htmlentitydefs` — HTML 一般エンティティの定義

このモジュールでは `entitydefs`、`codepoint2name`、`entitydefs` の三つの辞書を定義しています。 `entitydefs` は `htmllib` モジュールで `HTMLParser` クラスの `entitydefs` メンバを定義するために使われます。このモジュールでは XHTML 1.0 で定義された全てのエンティティを提供しており、Latin-1 キャラクタセット (ISO-8859-1) の簡単なテキスト置換を行う事ができます。

**entitydefs**

各 XHTML 1.0 エンティティ定義について、ISO Latin-1 における置換テキストへの対応付けを行っている辞書です。

**name2codepoint**

HTML のエンティティ名を Unicode のコードポイントに変換するための辞書です。2.3 で追加された仕様です。

**codepoint2name**

A dictionary that maps Unicode codepoints to HTML entity names. Unicode のコードポイントを HTML のエンティティ名に変換するための辞書です。2.3 で追加された仕様です。

## 13.5 `xml.parsers.expat` — Expat を使った高速な XML 解析

2.0 で追加された仕様です。

`xml.parsers.expat` モジュールは、検証 (validation) を行わない XML パーザ (parser, 解析器)、Expat への Python インタフェースです。モジュールは一つの拡張型 `xmlparser` を提供します。これは XML パーザの現在の状況を表します。一旦 `xmlparser` オブジェクトを生成すると、オブジェクトの様々な属性をハンドラ関数 (handler function) に設定できます。その後、XML 文書をパーザに入力すると、XML 文書の文字列とマークアップに応じてハンドラ関数が呼び出されます。

このモジュールでは、Expat パーザへのアクセスを提供するために `pyexpat` モジュールを使用します。

pyexpat モジュールの直接使用は撤廃されています。

このモジュールは、例外を一つと型オブジェクトを一つ提供しています。

#### **exception ExpatError**

Expat がエラーを報告したときに例外を送出します。Expat のエラーを解釈する上での詳細な情報は、[13.5.2](#) の “ExpatError Exceptions,” を参照してください。

#### **exception error**

ExpatError への別名です。

#### **XMLParserType**

ParserCreate() 関数から返された戻り値の型を示します。

xml.parsers.expat モジュールには以下の 2 つの関数が収められています:

#### **ErrorString(errno)**

与えられたエラー番号 *errno* を解説する文字列を返します。

#### **ParserCreate([encoding[, namespace\_separator]])**

新しい xmlparser オブジェクトを作成し、返します。*encoding* が指定されていた場合、XML データで使われている文字列のエンコード名でなければなりません。Expat は、Python のように多くのエンコードをサポートしておらず、またエンコーディングのレパートリを拡張することはできません; サポートするエンコードは、UTF-8, UTF-16, ISO-8859-1 (Latin1), ASCII です。*encoding* が指定されると、文書に対する明示的、非明示的なエンコード指定を上書き (override) します。

Expat はオプションで XML 名前空間の処理を行うことができます。これは引数 *namespace\_separator* に値を指定することで有効になります。この値は、1 文字の文字列でなければなりません; 文字列が誤った長さを持つ場合には ValueError が送出されます (None は値の省略と見なされます) 名前空間の処理が可能なとき、名前空間に属する要素と属性が展開されます。要素のハンドラである StartElementHandler と EndElementHandler に渡された要素名は、名前空間の URI、名前空間の区切り文字、要素名のローカル部を連結したものになります。名前空間の区切り文字が 0 バイト (chr(0)) の場合、名前空間の URI とローカル部は区切り文字なしで連結されます。

たとえば、*namespace\_separator* に空白文字 (' ') がセットされ、次のような文書が解析されるとします。

```
<?xml version="1.0"?>
<root xmlns      = "http://default-namespace.org/"
      xmlns:py   = "http://www.python.org/ns/">
  <py:elem1 />
  <elem2 xmlns="" />
</root>
```

StartElementHandler は各要素ごとに次のような文字列を受け取ります。

```
http://default-namespace.org/ root
http://www.python.org/ns/ elem1
elem2
```

#### **参考資料:**

*The Expat XML Parser*

(<http://www.libexpat.org/>)

Expat プロジェクトのホームページ

### 13.5.1 XMLParser Objects

xmlparser オブジェクトは以下のようなメソッドを持ちます。

**Parse**(*data*[, *isfinal*])

文字列 *data* の内容を解析し、解析されたデータを処理するための適切な関数を呼び出します。このメソッドを最後に呼び出す時は *isfinal* を真にしなければなりません。 *data* は空の文字列を取ることができます。

**ParseFile**(*file*)

*file* オブジェクトから読み込んだ XML データを解析します。 *file* には `read(nbytes)` メソッドのみが必要です。このメソッドはデータなくなった場合に空文字列を返さねばなりません。。

**SetBase**(*base*)

(XML) 宣言中のシステム識別子中の相対 URI を解決するための、基底 URI を設定します。相対識別子の解決はアプリケーションに任せられます: この値は関数 `ExternalEntityRefHandler` や `NotationDeclHandler`、`UnparsedEntityDeclHandler` に引数 *base* としてそのまま渡されます。

**GetBase**()

以前の `SetBase()` によって設定された基底 URI を文字列の形で返します。 `SetBase()` が呼ばれていないときには `None` を返します。

**GetInputContext**()

現在のイベントを発生させた入力データを文字列として返します。データはテキストの入っているエンティティが持っているエンコードになります。イベントハンドラがアクティブでないときに呼ばれると、戻り値は `None` となります。2.1 で追加された仕様です。

**ExternalEntityParserCreate**(*context*[, *encoding*])

親となるパーザで解析された内容が参照している、外部で解析されるエンティティを解析するために使える“子の”パーザを作成します。 *context* パラメータは、以下に記すように `ExternalEntityRefHandler()` ハンドラ関数に渡される文字列でなければなりません。子のパーザは `ordered_attributes`、`returns_unicode`、`specified_attributes` が現在のパーザの値に設定されて生成されます。

xmlparser オブジェクトは次のような属性を持ちます:

**buffer\_size**

`buffer_text` が真の時に使われるバッファのサイズです。この値は変更できません。2.3 で追加された仕様です。

**buffer\_text**

この値を真にすると、xmlparser オブジェクトが Expat から返されたもとの内容をバッファに保持するようになります。これにより可能なときに何度も `CharacterDataHandler()` を呼び出してしまうようなことを避けることができます。Expat は通常、文字列のデータを行末ごと大量に破棄するため、かなりパフォーマンスを改善できるはずですが。この属性はデフォルトでは偽で、いつでも変更可能です。2.3 で追加された仕様です。

**buffer\_used**

`buffer_text` が利用可能なとき、バッファに保持されたバイト数です。これらのバイトは UTF-8 でエンコードされたテキストを表します。この属性は `buffer_text` が偽の時には意味がありません。2.3 で追加された仕様です。

**ordered\_attributes**

この属性をゼロ以外の整数にすると、報告される (XML ノードの) 属性を辞書型ではなくリスト型にします。属性は文書のテキスト中の出現順で示されます。それぞれの属性は、2 つのリストのエント

り: 属性名とその値、が与えられます。(このモジュールの古いバージョンでも、同じフォーマットが使われています。) デフォルトでは、この属性はデフォルトでは偽となりますが、いつでも変更可能です。 2.1 で追加された仕様です。

#### **returns\_unicode**

この属性をゼロ以外の整数にすると、ハンドラ関数に Unicode 文字列が渡されます。 `returns_unicode` が 0 の時には、UTF-8 でエンコードされたデータを含む 8 ビット文字列がハンドラに渡されます。 1.6 で変更された仕様: 戻り値の型がいつでも変更できるように変更されたはず

#### **specified\_attributes**

ゼロ以外の整数にすると、パーザは文書のインスタンスで特定される属性だけを報告し、属性宣言から導出された属性は報告しないようになります。この属性が指定されたアプリケーションでは、XML プロセッサの振る舞いに関する標準に従うために必要とされる (文書型) 宣言によって、どのような付加情報が利用できるのかということについて特に注意を払わなければなりません。デフォルトで、この属性は偽となりますが、いつでも変更可能です。 2.1 で追加された仕様です。

以下の属性には、 `xmlparser` オブジェクトで最も最近に起きたエラーに関する値が入っており、また `Parse()` または `ParseFile()` メソッドが `xml.parsers.expat.ExpatError` 例外を送出した際にのみ正しい値となります。

#### **ErrorByteIndex**

エラーが発生したバイトのインデクスです。

#### **ErrorCode**

エラーを特定する数値によるコードです。この値は `ErrorString()` に渡したり、 `errors` オブジェクトで定義された内容と比較できます。

#### **ErrorColumnNumber**

エラーの発生したカラム番号です。

#### **ErrorLineNumber**

エラーの発生した行番号です。

以下に指定可能なハンドラのリストを示します。 `xmlparser` オブジェクト `o` にハンドラを指定するには、 `o.handlername = func` を使用します。 `handlername` は、以下のリストに挙げた値をとらねばならず、また `func` は正しい数の引数を受理する呼び出し可能なオブジェクトでなければなりません。引数は特に明記しない限り、すべて文字列となります。

#### **XmlDeclHandler** (*version, encoding, standalone*)

XML 宣言が解析された時に呼ばれます。XML 宣言とは、XML 勧告の適用バージョン (オプション)、文書テキストのエンコード、そしてオプションの “スタンドアロン” の宣言です。 `version` と `encoding` は `returns_unicode` 属性によって指示された型を示す文字列となり、 `standalone` は、文書がスタンドアロンであると宣言される場合には 1 に、文書がスタンドアロンでない場合には 0 に、スタンドアロン宣言を省略する場合には -1 になります。このハンドラは Expat のバージョン 1.95.0 以降のみ使用できます。 2.1 で追加された仕様です。

#### **StartDoctypeDeclHandler** (*doctypeName, systemId, publicId, has\_internal\_subset*)

Expat が文書型宣言 `<!DOCTYPE ...` を解析し始めたときに呼び出されます。 `doctypeName` は、与えられた値がそのまま Expat に提供されます。 `systemId` と `publicId` パラメタが指定されている場合、それぞれシステムと公開識別子を与えます。省略する時には `None` にします。文書が内部的な文書宣言のサブセット (internal document declaration subset) を持つか、サブセット自体の場合、 `has_internal_subset` は `true` になります。このハンドラには、Expat version 1.2 以上が必要です。

#### **EndDoctypeDeclHandler** ()

Expat が文書型宣言の解析を終えたときに呼び出されます。このハンドラには、Expat version 1.2 以



上が必要です。

**ElementDeclHandler**(*name, model*)

それぞれの要素型宣言ごとに呼び出されます。*name* は要素型の名前であり、*model* は内容モデル (content model) の表現です。

**AttlistDeclHandler**(*elname, attname, type, default, required*)

ひとつの要素型で宣言される属性ごとに呼び出されます。属性リストの宣言が3つの属性を宣言したとすると、このハンドラはひとつの属性に1度ずつ、3度呼び出されます。*elname* は要素名であり、これに対して宣言が適用され、*attname* が宣言された属性名となります。属性型は文字列で、*type* として渡されます; 取りえる値は、'CDATA', 'ID', 'IDREF', ... です。*default* は、属性が文書のインスタンスによって指定されていないときに使用されるデフォルト値を与えます。デフォルト値 (#IMPLIED values) が存在しないときには None を与えます。文書のインスタンスによって属性値が与えられる必要のあるときには *required* が true になります。このメソッドは Expat version 1.95.0 以上が必要です。

**StartElementHandler**(*name, attributes*)

要素の開始を処理するごとに呼び出されます。*name* は要素名を格納した文字列で、*attributes* はその値に属性名を対応付ける辞書型です。

**EndElementHandler**(*name*)

要素の終端を処理するごとに呼び出されます。

**ProcessingInstructionHandler**(*target, data*)

Called for every processing instruction. 処理命令を処理するごとに呼び出されます。

**CharacterDataHandler**(*data*)

文字データを処理するときに呼びだされます。このハンドラは通常の文字データ、CDATA セクション、無視できる空白文字列のために呼び出されます。これらを識別しなければならないアプリケーションは、要求された情報を収集するために StartCdataSectionHandler, EndCdataSectionHandler, and ElementDeclHandler コールバックメソッドを使用できます。

**UnparsedEntityDeclHandler**(*entityName, base, systemId, publicId, notationName*)

解析されていない (NDATA) エンティティ宣言を処理するために呼び出されます。このハンドラは Expat ライブラリのバージョン 1.2 のためだけに存在します; より最近のバージョンでは、代わりに EntityDeclHandler を使用してください (根底にある Expat ライブラリ内の関数は、撤廃されたものであると宣言されています)。

**EntityDeclHandler**(*entityName, is\_parameter\_entity, value, base, systemId, publicId, notationName*)

エンティティ宣言ごとに呼び出されます。パラメタと内部エンティティについて、*value* はエンティティ宣言の宣言済みの内容を与える文字列となります; 外部エンティティの時には None となります。解析済みエンティティの場合、*notationName* パラメタは None となり、解析されていないエンティティの時には記法 (notation) 名となります。*is\_parameter\_entity* は、エンティティがパラメタエンティティの場合真に、一般エンティティ (general entity) の場合には偽になります (ほとんどのアプリケーションでは、一般エンティティのことしか気にする必要がありません)。このハンドラは Expat ライブラリのバージョン 1.95.0 以降でのみ使用できます。2.1 で追加された仕様です。

**NotationDeclHandler**(*notationName, base, systemId, publicId*)

記法の宣言 (notation declaration) で呼び出されます。*notationName, base, systemId*, および *publicId* を与える場合、文字列にします。public な識別子が省略された場合、*publicId* は None になります。

**StartNamespaceDeclHandler**(*prefix, uri*)

要素が名前空間宣言を含んでいる場合に呼び出されます。名前空間宣言は、宣言が配置されている要素に対して StartElementHandler が呼び出される前に処理されます。

**EndNamespaceDeclHandler**(*prefix*)

名前空間宣言を含んでいたエレメントの終了タグに到達したときに呼び出されます。このハンドラ



は、要素に関する名前空間宣言ごとに、`StartNamespaceDeclHandler` とは逆の順番で一度だけ呼び出され、各名前空間宣言のスコープが開始されたことを示します。このハンドラは、要素が終了する際、対応する `EndElementHandler` が呼ばれた後に呼び出されます。

**CommentHandler**(*data*)

コメントで呼び出されます。*data* はコメントのテキストで、先頭の '`<!--`' と末尾の '`-->`' を除きます。

**StartCdataSectionHandler**()

CDATA セクションの開始時に呼び出されます。CDATA セクションの構文的な開始と終了位置を識別できるようにするには、このハンドラと `StartCdataSectionHandler` が必要です。

**EndCdataSectionHandler**()

CDATA セクションの終了時に呼び出されます。

**DefaultHandler**(*data*)

XML 文書中で、適用可能なハンドラが指定されていない文字すべてに対して呼び出されます。この文字とは、検出されたことが報告されるが、ハンドラは指定されていないようなコンストラクト (construct) の一部である文字を意味します。

**DefaultHandlerExpand**(*data*)

`DefaultHandler` と同じですが、内部エンティティの展開を禁止しません。エンティティ参照はデフォルトハンドラに渡されません。

**NotStandaloneHandler**()

XML 文書がスタンドアロンの文書として宣言されていない場合に呼び出されます。外部サブセットやパラメタエンティティへの参照が存在するが、XML 宣言が XML 宣言中で `standalone` 変数を `yes` に設定していない場合に起きます。このハンドラが 0 を返すと、パーザは `XML_ERROR_NOT_STANDALONE` を送出します。このハンドラが設定されていなければ、パーザは前述の事態で例外を送出しません。

**ExternalEntityRefHandler**(*context*, *base*, *systemId*, *publicId*)

外部エンティティの参照時に呼び出されます。*base* は現在の基底 (base) で、以前の `SetBase()` で設定された値になっています。*public*、および *system* の識別子である、*systemId* と *publicId* が指定されている場合、値は文字列です; *public* 識別子が指定されていない場合、*publicId* は `None` になります。*context* の値は不明瞭なものであり、以下に記述するようにしか使ってはなりません。

外部エンティティが解析されるようにするには、このハンドラを実装しなければなりません。このハンドラは、`ExternalEntityParserCreate(context)` を使って適切なコールバックを指定し、子パーザを生成して、エンティティを解析する役割を担います。このハンドラは整数を返さねばなりません; 0 を返した場合、パーザは `XML_ERROR_EXTERNAL_ENTITY_HANDLING` エラーを送出します。そうでないばあい、解析を継続します。

このハンドラが与えられておらず、`DefaultHandler` コールバックが指定されていれば、外部エンティティは `DefaultHandler` で報告されます。

### 13.5.2 ExpatError 例外

`ExpatError` 例外はいくつかの興味深い属性を備えています:

**code**

特定のエラーにおける `Expat` の内部エラー番号です。この値はこのモジュールの `errors` オブジェクトで定義されている定数のいずれかに一致します。2.1 で追加された仕様です。

**lineno**

エラーが検出された場所の行番号です。最初の行の番号は 1 です。2.1 で追加された仕様です。

**offset**

エラーが発生した場所の行内でのオフセットです。最初のカラムの番号は 0 です。2.1 で追加された仕様です。

### 13.5.3 例

以下のプログラムでは、与えられた引数を出力するだけの三つのハンドラを定義しています。

```
import xml.parsers.expat

# 3 handler functions
def start_element(name, attrs):
    print 'Start element:', name, attrs
def end_element(name):
    print 'End element:', name
def char_data(data):
    print 'Character data:', repr(data)

p = xml.parsers.expat.ParserCreate()

p.StartElementHandler = start_element
p.EndElementHandler = end_element
p.CharacterDataHandler = char_data

p.Parse("<?xml version='1.0'?>
<parent id='top'><child1 name='paul'>Text goes here</child1>
<child2 name='fred'>More text</child2>
</parent>")
```

このプログラムの出力は以下のようになります:

```
Start element: parent {'id': 'top'}
Start element: child1 {'name': 'paul'}
Character data: 'Text goes here'
End element: child1
Character data: '\n'
Start element: child2 {'name': 'fred'}
Character data: 'More text'
End element: child2
Character data: '\n'
End element: parent
```

### 13.5.4 内容モデルの記述

内容モデルは入れ子になったタプルを使って記述されています。各タプルには 4 つの値: 型、限定詞 (quantifier)、名前、そして子のタプル、が収められています。子のタプルは単に内容モデルを記述したものです。

最初の二つのフィールドの値は `xml.parsers.expat` モジュールの `model` オブジェクトで定義されている定数です。これらの定数は二つのグループ: モデル型 (model type) グループと限定子 (quantifier) グループ、に取りまとめられます。

以下にモデル型グループにおける定数を示します:

#### **XML\_CTYPE\_ANY**

モデル名で指定された要素は ANY の内容モデルを持つと宣言されます。

#### **XML\_CTYPE\_CHOICE**

指定されたエレメントはいくつかのオプションから選択できるようになっています; (A | B | C)

のような内容モデルで用いられます。

#### **XML\_CTYPE\_EMPTY**

EMPTY であると宣言されている要素はこのモデル型を持ちます。

#### **XML\_CTYPE\_MIXED**

#### **XML\_CTYPE\_NAME**

#### **XML\_CTYPE\_SEQ**

順々に続くようなモデルの系列を表すモデルがこのモデル型で表されます。(A, B, C) のようなモデルで用いられます。

限定子グループにおける定数を以下に示します:

#### **XML\_CQUANT\_NONE**

修飾子 (modifier) が指定されていません。従って A のように、厳密に一つだけです。

#### **XML\_CQUANT\_OPT**

このモデルはオプションです: A? のように、一つか全くないかです。

#### **XML\_CQUANT\_PLUS**

このモデルは (A+ のように) 一つかそれ以上あります。

#### **XML\_CQUANT\_REP**

このモデルは A\* のようにゼロ回以上あります。

### 13.5.5 Expat エラー定数

以下の定数は `xml.parsers.expat` モジュールにおける `errors` オブジェクトで提供されています。これらの定数は、エラーが発生した際に送出される `ExpatError` 例外オブジェクトのいくつかの属性を解釈する上で便利です。

`errors` オブジェクトは以下の属性を持ちます:

#### **XML\_ERROR\_ASYNC\_ENTITY**

#### **XML\_ERROR\_ATTRIBUTE\_EXTERNAL\_ENTITY\_REF**

属性値中のエンティティ参照が、内部エンティティではなく外部エンティティを参照しました。

#### **XML\_ERROR\_BAD\_CHAR\_REF**

文字参照が、XML では正しくない (illegal) 文字を参照しました (例えば 0 や ‘&#0;’ )。

#### **XML\_ERROR\_BINARY\_ENTITY\_REF**

エンティティ参照が、記法 (notation) つきで宣言されているエンティティを参照したため、解析できません。

#### **XML\_ERROR\_DUPLICATE\_ATTRIBUTE**

一つの属性が一つの開始タグ内に一度より多く使われています。

#### **XML\_ERROR\_INCORRECT\_ENCODING**

#### **XML\_ERROR\_INVALID\_TOKEN**

入力されたバイトが文字に適切に関連付けできない際に送出されます; 例えば、UTF-8 入力ストリームにおける NUL バイト (値 0) などです。

#### **XML\_ERROR\_JUNK\_AFTER\_DOC\_ELEMENT**

空白以外の何かがドキュメント要素の後にあります。

#### **XML\_ERROR\_MISPLACED\_XML\_PI**

入力データの先頭以外の場所に XML 定義が見つかりました。

#### **XML\_ERROR\_NO\_ELEMENTS**

このドキュメントには要素が入っていません (XML では全てのドキュメントは確実にトップレベルの要素を一つ持つよう要求しています)。

#### XML\_ERROR\_NO\_MEMORY

Expat が内部メモリを確保できませんでした。

#### XML\_ERROR\_PARAM\_ENTITY\_REF

パラメタエンティティが許可されていない場所で見つかりました。

#### XML\_ERROR\_PARTIAL\_CHAR

#### XML\_ERROR\_RECURSIVE\_ENTITY\_REF

エンティティ参照中に、同じエンティティへの別の参照が入っていました; おそらく違う名前で参照しているか、間接的に参照しています。

#### XML\_ERROR\_SYNTAX

何らかの仕様化されていない構文エラーに遭遇しました。

#### XML\_ERROR\_TAG\_MISMATCH

終了タグが最も内側で開かれている開始タグに一致しません。

#### XML\_ERROR\_UNCLOSED\_TOKEN

何らかの (開始タグのような) トークン が閉じられないまま、ストリームの終端や次のトークンに遭遇しました。

#### XML\_ERROR\_UNDEFINED\_ENTITY

定義されていないエンティティへの参照が行われました。

#### XML\_ERROR\_UNKNOWN\_ENCODING

ドキュメントのエンコードが Expat でサポートされていません。

## 13.6 xml.dom — 文書オブジェクトモデル (DOM) API

2.0 で追加された仕様です。

文書オブジェクトモデル、または“DOM”は、ワールドワイドウェブコンソーシアム (World Wide Web Consortium, W3C) による、XML ドキュメントにアクセスしたり変更を加えたりするための、プログラミング言語間共通の API です。DOM 実装によって、XML ドキュメントはツリー構造として表現されます。また、クライアントコード側でツリー構造をゼロから構築できるようになります。さらに、前述の構造に対して、よく知られたインタフェースをもつ一連のオブジェクトを通したアクセス手段も提供します。

DOM はランダムアクセスを行うアプリケーションで非常に有用です。SAX では、一度に閲覧することができるのはドキュメントのほんの一部分です。ある SAX 要素に注目している際には、別の要素をアクセスすることはできません。またテキストノードに注目しているときには、その中に入っている要素をアクセスすることができません。SAX によるアプリケーションを書くときには、プログラムがドキュメント内のどこを処理しているのかを追跡するよう、コードのどこかに記述する必要があります。SAX 自体がその作業を行ってくれることはありません。さらに、XML ドキュメントに対する先読み (look ahead) が必要だとすると不運なことになります。

アプリケーションによっては、ツリーにアクセスできなければイベント駆動モデルを実現できません。もちろん、何らかのツリーを SAX イベントに応じて自分で構築することもできるでしょうが、DOM ではそのようなコードを書かなくてもよくなります。DOM は XML データに対する標準的なツリー表現なのです。

文書オブジェクトモデルは、W3C によっていくつかの段階、W3C の用語で言えば“レベル (level)”で定義されています。Python においては、DOM API への対応付けは実質的には DOM レベル 2 勧告に基づいています。現在はドラフト形式でのみ入手できる レベル 3 仕様への対応付けは、Python XML 分科会 (Special Interest Group) により、PyXML パッケージの一部として開発中です。DOM レベル 3 サポートの現在の状

態についての情報は、PyXML パッケージに同梱されているドキュメントを参照してください。

DOM アプリケーションは、普通は XML を DOM に解析するところから始まります。どのようにして解析を行うかについては DOM レベル 1 では全くカバーしておらず、レベル 2 では限定的な改良だけが行われました: レベル 2 では Document を生成するメソッドを提供する DOMImplementation オブジェクトクラスがありますが、実装に依存しない方法で XML リーダ (reader)/パーザ (parser)/文書ビルダ (Document builder) にアクセスする方法はありません。また、既存の Document オブジェクトなしにこれらのメソッドにアクセスするような、よく定義された方法也没有ありません。Python では、各々の DOM 実装で getDOMImplementation() が定義されているはずで、DOM レベル 3 ではロード (Load)/ストア (Store) 仕様が追加され、リーダーのインタフェースにを定義していますが、Python 標準ライブラリではまだ利用することができません。

DOM 文書オブジェクトを生成したら、そのプロパティとメソッドを使って XML 文書の一部にアクセスできます。これらのプロパティは DOM 仕様で定義されています; 本リファレンスマニュアルでは、Python において DOM 仕様がどのように解釈されているかを記述しています。

W3C から提供されている仕様は、DOM API を Java、ECMAScript、および OMG IDL で定義しています。ここで定義されている Python での対応づけは、大部分がこの仕様の IDL 版に基づいていますが、厳密な準拠は必要とされていません (実装で IDL の厳密な対応付けをサポートするのは自由ですが)。API への対応付けに関する詳細な議論は 13.6.3、“適合性”を参照してください。

参考資料:

*Document Object Model (DOM) Level 2 Specification*

(<http://www.w3.org/TR/DOM-Level-2-Core/>)

Python DOM API が準拠している W3C 勧告。

*Document Object Model (DOM) Level 1 Specification*

(<http://www.w3.org/TR/REC-DOM-Level-1/>)

xml.dom.minidom でサポートされている W3C の DOM に関する勧告。

*PyXML*

(<http://pyxml.sourceforge.net>)

完全な機能をもった DOM 実装を必要とするユーザは PyXML パッケージを利用すべきです。

*CORBA Scripting with Python*

(<http://cgi.omg.org/cgi-bin/doc?orbos/99-08-02.pdf>)

このドキュメントでは OMG IDL から Python への対応付けを記述しています。

### 13.6.1 モジュールの内容

xml.dom には、以下の関数が収められています:

**registerDOMImplementation**(*name*, *factory*)

ファクトリ関数 (factory function) *factory* を名前 *name* で登録します。ファクトリ関数は DOMImplementation インタフェースを実装するオブジェクトを返さなければなりません。ファクトリ関数は毎回同じオブジェクトを返すこともでき、呼び出されるたびに、特定の実装 (例えば実装が何らかのカスタマイズをサポートしている場合) における、適切な新たなオブジェクトを返すこともできます。

**getDOMImplementation**( [*name* [, *features*] ] )

適切な DOM 実装を返します *name* は、よく知られた DOM 実装のモジュール名か、None になります。None でない場合、対応するモジュールを import して、import が成功した場合 DOMImplementation オブジェクトを返します。 *name* が与えられておらず、環境変数 PYTHON\_DOM が設定されていた場合、DOM 実装を見つけるのに環境変数が使われます。

*name* が与えられない場合、利用可能な実装を調べて、指定された機能 (feature) セットを持つもの

を探します。実装が見つからなければ `ImportError` を送出します。`features` のリストは (`feature`, `version`) のペアからなる配列で、利用可能な `DOMImplementation` オブジェクトの `hasFeature()` メソッドに渡されます。

いくつかの便利な定数も提供されています:

#### **EMPTY\_NAMESPACE**

DOM 内のノードに名前空間が何も関連づけられていないことを示すために使われる値です。この値は通常、ノードの `namespaceURI` の値として見つかったり、名前空間特有のメソッドに対する `namespaceURI` パラメタとして使われます。2.2 で追加された仕様です。

#### **XML\_NAMESPACE**

*Namspaces in XML* (4 節) で定義されている、予約済みプレフィクス (reserved prefix) `xml` に関連付けられた名前空間 URI です。2.2 で追加された仕様です。

#### **XMLNS\_NAMESPACE**

*Document Object Model (DOM) Level 2 Core Specification* (1.1.8 節) で定義されている、名前空間宣言への名前空間 URI です。2.2 で追加された仕様です。

#### **XHTML\_NAMESPACE**

*XHTML 1.0: The Extensible HyperText Markup Language* (3.1.1 節) で定義されている、XHTML 名前空間 URI です。2.2 で追加された仕様です。

加えて、`xml.dom` には基底となる `Node` クラスと DOM 例外クラスが収められています。このモジュールで提供されている `Node` クラスは DOM 仕様で定義されているメソッドや属性は何ら実装していません; これらは具体的な DOM 実装において提供しなければなりません。このモジュールの一部として提供されている `Node` クラスでは、具体的な `Node` オブジェクトの `nodeType` 属性として使う定数を提供しています; これらの定数は、DOM 仕様に適合するため、クラスではなくモジュールのレベルに配置されています。

## 13.6.2 DOM 内のオブジェクト

DOM について最も明確に限定しているドキュメントは W3C による DOM 仕様です。

DOM 属性は単純な文字列としてだけではなく、ノードとして操作されるかもしれないので注意してください。とはいえ、そうしなければならない場合はかなり稀なので、今のところ記述されていません。

インタフェース	節	目的
<code>DOMImplementation</code>	<a href="#">13.6.2</a>	根底にある実装へのインタフェース。
<code>Node</code>	<a href="#">13.6.2</a>	ドキュメント内の大部分のオブジェクトのに対する基底インタフェース。
<code>NodeList</code>	<a href="#">13.6.2</a>	ノードの配列に対するインタフェース。
<code>DocumentType</code>	<a href="#">13.6.2</a>	ドキュメントを処理するために必要な宣言についての情報。
<code>Document</code>	<a href="#">13.6.2</a>	ドキュメント全体を表現するオブジェクト。
<code>Element</code>	<a href="#">13.6.2</a>	ドキュメント階層内の要素ノード。
<code>Attr</code>	<a href="#">13.6.2</a>	階層ノード上の属性値。
<code>Comment</code>	<a href="#">13.6.2</a>	ソースドキュメント内のコメント表現。
<code>Text</code>	<a href="#">13.6.2</a>	ドキュメント内のテキスト記述を含むノード。
<code>ProcessingInstruction</code>	<a href="#">13.6.2</a>	処理命令 (processing instruction) 表現。

さらに追加の節として、Python で DOM を利用するために定義されている例外について記述しています。



## DOMImplementation オブジェクト

DOMImplementation インタフェースは、利用している DOM 実装において特定の機能が利用可能かどうかを決定するための方法をアプリケーションに提供します。DOM レベル 2 では、DOMImplementation を使って新たな Document オブジェクトや DocumentType オブジェクトを生成する機能も追加しています。

**hasFeature**(*feature, version*)

## Node オブジェクト

XML 文書の全ての構成要素は Node のサブクラスです。

### nodeType

ノード (node) の型を表現する整数値です。型に対応する以下のシンボル定数: ELEMENT\_NODE、ATTRIBUTE\_NODE、TEXT\_NODE、CDATA\_SECTION\_NODE、ENTITY\_NODE、PROCESSING\_INSTRUCTION\_NODE、COMMENT\_NODE、DOCUMENT\_NODE、DOCUMENT\_TYPE\_NODE、NOTATION\_NODE、が Node オブジェクトで定義されています。読み出し専用の属性です。

### parentNode

現在のノードの親ノードか、文書ノードの場合には None になります。この値は常に Node オブジェクトか None になります。Element ノードの場合、この値はルート要素 (root element) の場合を除き親要素 (parent element) となり、ルート要素の場合には Document オブジェクトとなります。Attr ノードの場合、この値は常に None となります。読み出し専用の属性です。

### attributes

属性オブジェクトの NamedNodeMap です。要素だけがこの属性に実際の値を持ちます; その他のオブジェクトでは、この属性を None にします。読み出し専用の属性です。

### previousSibling

このノードと同じ親ノードを持ち、直前にくるノードです。例えば、*self* 要素の開始タグの直前にくる終了タグを持つ要素です。もちろん、XML 文書は要素だけで構成されているだけではないので、直前にくる兄弟関係にある要素 (sibling) はテキストやコメント、その他になる可能性があります。このノードが親ノードにおける先頭の子ノードである場合、属性値は None になります。読み出し専用の属性です。

### nextSibling

このノードと同じ親ノードを持ち、直後にくるノードです。例えば、previousSibling も参照してください。このノードが親ノードにおける末尾頭の子ノードである場合、属性値は None になります。読み出し専用の属性です。

### childNodes

このノード内に収められているノードからなるリストです。読み出し専用の属性です。

### firstChild

このノードに子ノードがある場合、その先頭のノードです。そうでない場合 None になります。読み出し専用の属性です。

### lastChild

このノードに子ノードがある場合、その末尾のノードです。そうでない場合 None になります。読み出し専用の属性です。

### localName

tagName にコロンがあれば、コロン以降の部分に、なければ tagName 全体になります。値は文字列です。

### **prefix**

tagName のコロンがあれば、コロン以前の部分に、なければ空文字列になります。値は文字列か、None になります。

### **namespaceURI**

要素名に関連付けられた名前空間です。文字列か None になります。読み出し専用の属性です。

### **nodeName**

この属性はノード型ごとに異なる意味を持ちます; 詳しくは DOM 仕様を参照してください。この属性で得られることになる情報は、全てのノード型では tagName、属性では name プロパティといったように、常に他のプロパティで得ることができます。全てのノード型で、この属性の値は文字列か None になります。読み出し専用の属性です。

### **nodeValue**

この属性はノード型ごとに異なる意味を持ちます; 詳しくは DOM 仕様を参照してください。その序今日は nodeName と似ています。この属性の値は文字列か None になります。

### **hasAttributes()**

ノードが何らかの属性を持っている場合に真を返します。

### **hasChildNodes()**

ノードが何らかの子ノードを持っている場合に真を返します。

### **isSameNode(other)**

other がこのノードと同じノードを参照している場合に真を返します。このメソッドは、何らかのプロキシ (proxy) 機構を利用するような DOM 実装で特に便利です (一つ以上のオブジェクトが同じノードを参照するかもしれないからです)。

注意: このメソッドは DOM レベル 3 API で提案されており、まだ “ワーキングドラフト (working draft)” の段階です。しかし、このインタフェースだけは議論にはならないと考えられます。W3C による変更は必ずしも Python DOM インタフェースにおけるこのメソッドに影響するとは限りません (ただしこのメソッドに対する何らかの新たな W3C API もサポートされるかもしれません)。

### **appendChild(newChild)**

現在のノードの子ノードリストの末尾に新たな子ノードを追加し、newChild を返します。

### **insertBefore(newChild, refChild)**

新たな子ノードを既存の子ノードの前に挿入します。refChild は現在のノードの子ノードである場合に限られます; そうでない場合、ValueError が送出されます。newChild が返されます。

### **removeChild(oldChild)**

子ノードを削除します。oldChild はこのノードの子ノードでなければなりません。そうでない場合、ValueError が送出されます。成功した場合 oldChild が返されます。oldChild をそれ以降使わない場合、unlink() メソッドを呼び出さなければなりません。

### **replaceChild(newChild, oldChild)**

既存のノードと新たなノードを置き換えます。この操作は oldChild が現在のノードの子ノードである場合に限られます; そうでない場合、ValueError が送出されます。

### **normalize()**

一続きのテキスト全体を一個の Text インスタンスとして保存するために隣接するテキストノードを結合します。これにより、多くのアプリケーションで DOM ツリーからのテキスト処理が簡単になります。2.1 で追加された仕様です。

### **cloneNode(deep)**

このノードを複製 (clone) します。deep を設定すると、子ノードも同様に複製することを意味します。複製されたノードを返します。

## NodeList オブジェクト

NodeList は、ノードからなる配列を表現します。これらのオブジェクトは DOM コア勧告 (DOM Core recommendation) において、二通りに使われています: Element オブジェクトでは、子ノードのリストを提供するのに NodeList を利用します。また、このインタフェースにおける Node の `getElements-ByTagName()` および `getElementsByNameNS()` メソッドは、クエリに対する結果を表現するのに NodeList を利用します。

DOM レベル 2 勧告では、これらのオブジェクトに対し、メソッドと属性を一つずつ定義しています:

**item(*i*)**

配列に *i* 番目の要素がある場合にはその要素を、そうでない場合には None を返します。*i* はゼロよりも小さくはならず、配列の長さ以上であってはなりません。

**length**

配列中のノードの数です。

この他に、Python の DOM インタフェースでは、NodeList オブジェクトを Python の配列として使えるようにするサポートが追加されていることが必要です。NodeList の実装では、全て `__len__()` と `__getitem__()` をサポートしなければなりません; このサポートにより、for 文内で NodeList にわたる繰り返しと、組み込み関数 `len()` の適切なサポートができるようになります。

DOM 実装が文書の変更をサポートしている場合、NodeList の実装でも `__setitem__()` および `__delitem__()` メソッドをサポートしなければなりません。

## DocumentType オブジェクト

文書で宣言されている記法 (notation) やエンティティ (entity) に関する (外部サブセット (external subset) がパーザから利用でき、情報を提供できる場合にはそれも含めた) 情報は、DocumentType オブジェクトから手に入れることができます。文書の DocumentType は、Document オブジェクトの `doctype` 属性で入手することができます; 文書の DOCTYPE 宣言がない場合、文書の `doctype` 属性は、このインタフェースを持つインスタンスの代わりに None に設定されます。

DocumentType は Node を特殊化したもので、以下の属性を加えています:

**publicId**

文書型定義 (document type definition) の外部サブセットに対する公開識別子 (public identifier) です。文字列または None になります。

**systemId**

文書型定義 (document type definition) の外部サブセットに対するシステム識別子 (system identifier) です。文字列の URI または None になります。

**internalSubset**

ドキュメントの完全な内部サブセットを与える文字列です。サブセットを囲むブラケットは含みません。ドキュメントが内部サブセットを持たない場合、この値は None です。

**name**

DOCTYPE 宣言でルート要素の名前が与えられている場合、その値になります。

**entities**

外部エンティティの定義を与える NamedNodeMap です。複数回定義されているエンティティに対しては、最初の定義だけが提供されます (その他は XML 勧告での要求仕様によって無視されます)。パーザによって情報が提供されないか、エンティティが定義されていない場合には、この値は None になることがあります。

**notations**

記法の定義を与える `NamedNodeMap` です。複数回定義されている記法名に対しては、最初の定義だけが提供されます (その他は XML 勧告での要求仕様によって無視されます)。パーザによって情報が提供されないか、エンティティが定義されていない場合には、この値は `None` になることがあります。

## Document オブジェクト

`Document` は XML ドキュメント全体を表現し、その構成要素である要素、属性、処理命令、コメント等が入っています。`Document` は `Node` からプロパティを継承していることを思い出してください。

### `documentElement`

ドキュメントの唯一無二のルート要素です。

### `createElement(tagName)`

新たな要素ノードを生成して返します。要素は、生成された時点ではドキュメント内に挿入されません。`insertBefore()` や `appendChild()` のような他のメソッドの一つを使って明示的に挿入を行う必要があります。

### `createElementNS(namespaceURI, tagName)`

名前空間を伴う新たな要素ノードを生成して返します。`tagName` にはプレフィクス (prefix) があってもかまいません。要素は、生成された時点では文書内に挿入されません。`insertBefore()` や `appendChild()` のような他のメソッドの一つを使って明示的に挿入を行う必要があります。`appendChild()`。

### `createTextNode(data)`

パラメタで渡されたデータの入ったテキストノードを生成して返します。他の生成 (create) メソッドと同じく、このメソッドは生成されたノードをツリーに挿入しません。

### `createComment(data)`

パラメタで渡されたデータの入ったコメントノードを生成して返します。他の生成 (create) メソッドと同じく、このメソッドは生成されたノードをツリーに挿入しません。

### `createProcessingInstruction(target, data)`

パラメタで渡された `target` および `data` の入った処理命令ノードを生成して返します。他の生成 (create) メソッドと同じく、このメソッドは生成されたノードをツリーに挿入しません。

### `createAttribute(name)`

属性ノードを生成して返します。このメソッドは属性ノードを特定の要素に関連づけることはしません。新たに生成された属性インスタンスを使うには、適切な `Element` オブジェクトの `setAttributeNode()` を使わなければなりません。

### `createAttributeNS(namespaceURI, qualifiedName)`

名前空間を伴う新たな属性ノードを生成して返します。`tagName` にはプレフィクス (prefix) があってもかまいません。このメソッドは属性ノードを特定の要素に関連づけることはしません。新たに生成された属性インスタンスを使うには、適切な `Element` オブジェクトの `setAttributeNode()` を使わなければなりません。

### `getElementsByTagName(tagName)`

全ての下位要素 (直接の子要素、子要素の子要素、等) から、特定の要素型名を持つものを検索します。

### `getElementsByTagNameNS(namespaceURI, localName)`

全ての下位要素 (直接の子要素、子要素の子要素、等) から、特定の名前空間 URI とローカル名 (local name) を持つものを検索します。ローカル名は名前空間におけるプレフィクス以降の部分です。

## Element オブジェクト

Element は Node のサブクラスです。このため Node クラスの全ての属性を継承します。

### tagName

要素型名です。名前空間使用の文書では、要素型名中にコロンがあるかもしれません。値は文字列です。

### getElementsByTagName (tagName)

Document クラス内における同名のメソッドと同じです。

### getElementsByName (tagName)

Document クラス内における同名のメソッドと同じです。

### getAttribute (attname)

属性値を文字列で返します。

### getAttributeNode (attrname)

attrname で指定された属性の Attr ノードを返します。

### getAttributeNS (namespaceURI, localName)

指定した namespaceURI および localName を持つ属性値を文字列として返します。

### getAttributeNodeNS (namespaceURI, localName)

指定した namespaceURI および localName を持つ属性値をノードとして返します。

### removeAttribute (attname)

名前で指定された属性を削除します。該当する属性がなくても例外は送出されません。

### removeAttributeNode (oldAttr)

oldAttr が属性リストにある場合、削除して返します。oldAttr が存在しない場合、NotFoundError が送出されます。

### removeAttributeNS (namespaceURI, localName)

名前で指定された属性を削除します。このメソッドは qname ではなく localName を使うので注意してください。該当する属性がなくても例外は送出されません。

### setAttribute (attname, value)

文字列を使って属性値を設定します。

### setAttributeNode (newAttr)

新たな属性ノードを要素に追加します。name 属性が既存の属性に一致した場合、必要に応じて属性を置き換えます。置換が生じると、古い属性ノードが返されます。newAttr がすでに使われていれば、InuseAttributeErr が送出されます。

### setAttributeNodeNS (newAttr)

新たな属性ノードを要素に追加します。namespaceURI および localName 属性が既存の属性に一致した場合、必要に応じて属性を置き換えます。置換が生じると、古い属性ノードが返されます。newAttr がすでに使われていれば、InuseAttributeErr が送出されます。

### setAttributeNS (namespaceURI, qname, value)

指定された namespaceURI および qname で与えられた属性の値を文字列で設定します。qname は属性の完全な名前であり、この点が上記のメソッドと違うので注意してください。

## Attr オブジェクト

Attr は Node を継承しており、全ての属性を受け継いでいます。

### name

要素型名です。名前空間使用の文書では、要素型名中にコロンがあるかもしれません。

**localName**

名前にコロンがあればコロン以降の部分に、なければ名前全体になります。

**prefix**

名前にコロンがあればコロン以前の部分に、なければ空文字列になります。

## NamedNodeMap Objects

NamedNodeMap は Node を継承していません。

**length**

属性リストの長さです。

**item(index)**

特定のインデックスを持つ属性を返します。属性の並び方は任意ですが、DOM 文書が生成されている間は一定になります。各要素は属性ノードです。属性値はノードの value 属性で取得してください。

このクラスをよりマップ型の動作ができるようにする実験的なメソッドもあります。そうしたメソッドを使うこともできますし、Element オブジェクトに対して、標準化された `getAttribute*()` ファミリのメソッドを使うこともできます。

## Comment オブジェクト

Comment は XML 文書中のコメントを表現します。Comment は Node のサブクラスですが、子ノードを持つことはありません。

**data**

文字列によるコメントの内容です。この属性には、コメントの先頭にある `<!--` と末尾にある `-->` 間の全ての文字が入っていますが、`<!--` と `-->` 自体は含みません。

## Text オブジェクトおよび CDATASection オブジェクト

Text インタフェースは XML 文書内のテキストを表現します。パーザおよび DOM 実装が DOM の XML 拡張をサポートしている場合、CDATA でマークされた区域 (section) に入れられている部分テキストは CDATASection オブジェクトに記憶されます。これら二つのインタフェースは同一ののですが、`nodeType` 属性が異なります。

これらのインタフェースは Node インタフェースを拡張したものです。しかし子ノードを持つことはできません。

**data**

文字列によるテキストノードの内容です。

注意: CDATASection ノードの利用は、ノードが完全な CDATA マーク区域を表現するという意味ではなく、ノードの内容が CDATA 区域の一部であることを意味するだけです。単一の CDATA セクションは文書ツリー内で複数のノードとして表現されることがあります。二つの隣接する CDATASection ノードが、異なる CDATA マーク区域かどうかを決定する方法はありません。

## ProcessingInstruction オブジェクト

XML 文書内の処理命令を表現します; Node インタフェースを継承していますが、子ノードを持つことはできません。



#### **target**

最初の空白文字までの処理命令の内容です。読み出し専用の属性です。

#### **data**

最初の空白文字以降の処理命令の内容です。

#### **例外**

2.1 で追加された仕様です。

DOM レベル 2 勧告では、単一の例外 `DOMException` と、どの種のエラーが発生したかをアプリケーションが決定できるようにする多くの定数を定義しています。`DOMException` インスタンスは、特定の例外に関する適切な値を提供する `code` 属性を伴っています。

Python DOM インタフェースでは、上記の定数を提供していますが、同時に一連の例外を拡張して、DOM で定義されている各例外コードに対して特定の例外が存在するようにしています。DOM の実装では、適切な特定の例外を送出しなければならず、各例外は `code` 属性に対応する適切な値を伴わなければなりません。

#### **exception `DOMException`**

全ての特定の DOM 例外で使われている基底例外クラスです。この例外クラスは直接インスタンス化することができません。

#### **exception `DomstringSizeErr`**

指定された範囲のテキストが文字列に収まらない場合に送出されます。この例外は Python の DOM 実装で使われるかどうかは判っていませんが、Python で書かれていない DOM 実装から送出される場合があります。

#### **exception `HierarchyRequestErr`**

挿入できない型のノードを挿入しようと試みたときに送出されます。

#### **exception `IndexSizeErr`**

メソッドに与えたインデックスやサイズパラメタが負の値や許容範囲の値を超えた際に送出されます。

#### **exception `InuseAttributeErr`**

文書中にすでに存在する `Attr` ノードを挿入しようと試みた際に送出されます。

#### **exception `InvalidAccessErr`**

パラメタまたは操作が根底にあるオブジェクトでサポートされていない場合に送出されます。

#### **exception `InvalidCharacterErr`**

この例外は、文字列パラメタが、現在使われているコンテキストで XML 1.0 勧告によって許可されていない場合に送出されます。例えば、要素型に空白の入った `Element` ノードを生成しようとすると、このエラーが送出されます。

#### **exception `InvalidModificationErr`**

ノードの型を変更しようと試みた際に送出されます。

#### **exception `InvalidStateErr`**

利用できないオブジェクトや、もはや利用できなくなったオブジェクトを使おうと試みた際に送出されます。

#### **exception `NamespaceErr`**

*Namespaces in XML* に照らして許可されていない方法でオブジェクトを変更しようと試みた場合、この例外が送出されます。

#### **exception `NotFoundErr`**

参照しているコンテキスト中に目的のノードが存在しない場合に送出される例外です。例えば、

`NamedNodeMap.removeNamedItem()` は渡されたノードがノードマップ中に存在しない場合にこの例外を送出します。

**exception NotSupportedErr**

要求された方のオブジェクトや操作が実装でサポートされていない場合に送出されます。

**exception NoDataAllowedErr**

データ属性をサポートしないノードにデータを指定した際に送出されます。

**exception NoModificationAllowedErr**

オブジェクトに対して (読み出し専用ノードに対する修正のように) 許可されていない修正を行おうと試みた際に送出されます。

**exception SyntaxErr**

無効または不正な文字列が指定された際に送出されます。

**exception WrongDocumentErr**

ノードが現在属している文書と異なる文書に挿入され、かつある文書から別の文書へのノードの移行が実装でサポートされていない場合に送出されます。

DOM 勧告で定義されている例外コードは、以下のテーブルに従って上記の例外と対応付けられます:

定数	例外
DOMSTRING_SIZE_ERR	DomstringSizeErr
HIERARCHY_REQUEST_ERR	HierarchyRequestErr
INDEX_SIZE_ERR	IndexSizeErr
INUSE_ATTRIBUTE_ERR	InuseAttributeErr
INVALID_ACCESS_ERR	InvalidAccessErr
INVALID_CHARACTER_ERR	InvalidCharacterErr
INVALID_MODIFICATION_ERR	InvalidModificationErr
INVALID_STATE_ERR	InvalidStateErr
NAMESPACE_ERR	NamespaceErr
NOT_FOUND_ERR	NotFoundErr
NOT_SUPPORTED_ERR	NotSupportedErr
NO_DATA_ALLOWED_ERR	NoDataAllowedErr
NO_MODIFICATION_ALLOWED_ERR	NoModificationAllowedErr
SYNTAX_ERR	SyntaxErr
WRONG_DOCUMENT_ERR	WrongDocumentErr

### 13.6.3 適合性

この節では適合性に関する要求と、Python DOM API、W3C DOM 勧告、および OMG IDL の Python API への対応付けとの間の関係について述べます。

#### 型の対応付け

DOM 仕様で使われている基本的な IDL 型は、以下のテーブルに従って Python の型に対応付けられています。

IDL 型	Python 型
boolean	IntegerType (値 0 または 1) による
int	IntegerType
long int	IntegerType
unsigned int	IntegerType

さらに、勧告で定義されている DOMString は、Python 文字列または Unicode 文字列に対応付けられます。アプリケーションでは、DOM から文字列が返される際には常に Unicode を扱えなければなりません。

IDL の null 値は None に対応付けられており、API で null の使用が許されている場所では常に受理されるか、あるいは実装によって提供されるはずです。

## アクセサメソッド

OMG IDL から Python への対応付けは、IDL attribute 宣言へのアクセサ関数の定義を、Java による対応付けが行うのと同様に行います。

### IDL 宣言の対応付け

```
readonly attribute string someValue;
attribute string anotherValue;
```

は、三つのアクセサ関数: someValue に対する “get” メソッド (`_get_someValue()`)、そして anotherValue に対する “get” および “set” メソッド (`_get_anotherValue()` および `_set_anotherValue()`) を生み出します。とりわけ、対応付けでは、IDL 属性が通常の Python 属性としてアクセス可能であることは必須ではありません: `object.someValue` が動作することは必須ではなく、`AttributeError` を送出してもかまいません。

しかしながら、Python DOM API では、通常の属性アクセスが動作することが必須です。これは、Python IDL コンパイラによって生成された典型的なサロゲーションはまず動作することはなく、DOM オブジェクトが CORBA を解してアクセスされる場合には、クライアント上でラップオブジェクトが必要であることを意味します。CORBA DOM クライアントでは他にもいくつか考慮すべきことがある一方で、CORBA を介して DOM を使った経験を持つ実装者はこのことを問題視していません。readonly であると宣言された属性は、全ての DOM 実装で書き込みアクセスを制限しているとは限りません。

さらに、アクセサ関数は必須ではありません。アクセサ関数が提供された場合、Python IDL 対応付けによって定義された形式をとらなければなりませんが、属性は Python から直接アクセスすることができるので、それらのメソッドは必須ではないと考えられます。readonly であると宣言された属性に対しては、“set” アクセサを提供してはなりません。

## 13.7 xml.dom.minidom — 軽量な DOM 実装

2.0 で追加された仕様です。

`xml.dom.minidom` は、軽量な文書オブジェクトモデルインタフェースの実装です。この実装では、完全な DOM よりも単純で、かつ十分に小さくなるよう意図しています。

DOM アプリケーションは典型的に、XML を DOM に解析 (parse) することで開始します。`xml.dom.minidom` では、以下のような解析用の関数を介して行います:

```

from xml.dom.minidom import parse, parseString

dom1 = parse('c:\\temp\\mydata.xml') # parse an XML file by name

datasource = open('c:\\temp\\mydata.xml')
dom2 = parse(datasource) # parse an open file

dom3 = parseString('<myxml>Some data<empty/> some more data</myxml>')

```

`parse()` 関数はファイル名か、開かれたファイルオブジェクトを引数にとることができます。

**`parse(filename_or_file, parser)`**

与えられた入力から Document を返します。 *filename\_or\_file* はファイル名でもファイルオブジェクトでもかまいません。 *parser* を指定する場合、SAX2 パーザオブジェクトでなければなりません。この関数はパーザの文書ハンドラを変更し、名前空間サポートを有効にします; (エンティティリゾルバ (entity resolver) のような) 他のパーザ設定は前もっておこなわなければなりません。

XML データを文字列で持っている場合、`parseString()` を代わりに使うことができます:

**`parseString(string[, parser])`**

*string* を表現する Document を返します。このメソッドは文字列に対する StringIO オブジェクトを生成して、そのオブジェクトを `parse` に渡します。

これらの関数は両方とも、文書の内容を表現する Document オブジェクトを返します。

`parse()` や `parseString()` といった関数が行うのは、XML パーザを、何らかの SAX パーザからくる解析イベント (parse event) を受け取って DOM ツリーに変換できるような “DOM ビルダ (DOM builder)” に結合することです。関数は誤解を招くような名前になっているかもしれませんが、インタフェースについて学んでいるときには理解しやすいでしょう。文書の解析はこれらの関数が戻るより前に完結します; 要するに、これらの関数自体はパーザ実装を提供しないということです。

“DOM 実装” オブジェクトのメソッドを呼び出して Document を生成することもできます。このオブジェクトは、`xml.dom` パッケージ、または `xml.dom.minidom` モジュールの `getDOMImplementation()` 関数を呼び出して取得できます。`xml.dom.minidom` モジュールの実装を使うと、常に `minidom` 実装の Document インスタンスを返します。一方、`xml.dom` 版の関数では、別の実装によるインスタンスを返すかも知れません (PyXML package がインストールされているとそうなるでしょう)。Document を取得したら、DOM を構成するために子ノードを追加していくことができます:

```

from xml.dom.minidom import getDOMImplementation

impl = getDOMImplementation()

newdoc = impl.createDocument(None, "some_tag", None)
top_element = newdoc.documentElement
text = newdoc.createTextNode('Some textual content.')
top_element.appendChild(text)

```

DOM 文書オブジェクトを手にしたら、XML 文書のプロパティやメソッドを使って、文書の一部にアクセスすることができます。これらのプロパティは DOM 仕様で定義されています。文書オブジェクトの主要なプロパティは `documentElement` プロパティです。このプロパティは XML 文書の主要な要素: 他の全ての要素を保持する要素、を与えます。以下にプログラム例を示します:

```
dom3 = parseString("<myxml>Some data</myxml>")
assert dom3.documentElement.tagName == "myxml"
```

DOM を使い終えたら、後片付けを行わなければなりません。Python のバージョンによっては、循環的に互いを参照するオブジェクトに対するガベージコレクションをサポートしていないため、この操作が必要となります。この制限が全てのバージョンの Python から除去されるまでは、循環参照オブジェクトが消去されないものとしてコードを書くのが無難です。

DOM を片付けるには、`unlink()` メソッドを呼び出します:

```
dom1.unlink()
dom2.unlink()
dom3.unlink()
```

`unlink()` は、DOM API に対する `xml.dom.minidom` 特有の拡張です。ノードに対して `unlink()` を呼び出した後は、ノードとその下位ノードは本質的には無意味なものとなります。

参考資料:

*Document Object Model (DOM) Level 1 Specification*

(<http://www.w3.org/TR/REC-DOM-Level-1/>)

`xml.dom.minidom` でサポートされている DOM の W3C 勧告。

### 13.7.1 DOM オブジェクト

Python の DOM API 定義は `xml.dom` モジュールドキュメントの一部として与えられています。この節では、`xml.dom` の API と `xml.dom.minidom` との違いについて列挙します。

**`unlink()`**

DOM との内部的な参照を破壊して、循環参照ガベージコレクションを持たないバージョンの Python でもガベージコレクションされるようにします。循環参照ガベージコレクションが利用できても、このメソッドを使えば、大量のメモリをすぐに使えるようにできるため、必要なくなったらすぐにこのメソッドを DOM オブジェクトに対して呼ぶのが良い習慣です。このメソッドは Document オブジェクトに対してだけ呼び出せばよいのですが、あるノードの子ノードを放棄するために子ノードに対して呼び出してはかまいません。

**`writexml(writer)`**

XML を `writer` オブジェクトに書き込みます。 `writer` は、ファイルオブジェクトインタフェースの `write()` に該当するメソッドを持たなければなりません。

2.1 で追加された仕様: 美しい出力をサポートするため、新たなキーワード引数 `indent`、`addindent`、および `newl` が追加されました

2.3 で追加された仕様: Document ノードに対して、追加のキーワード引数 `encoding` を使って、XML ヘッダの `encoding` フィールドを指定できるようになりました

**`toxml([encoding])`**

DOM が表現している XML を文字列にして返します。

2.3 で追加された仕様: `encoding` 引数

引数がなければ、XML ヘッダは `encoding` を指定せず、文書内の全ての文字をデフォルトエンコード方式で表示できない場合、結果は Unicode 文字列となります。この文字列を UTF-8 以外のエンコード方式でエンコードするのは不正であり、なぜなら UTF-8 が XML のデフォルトエンコード方式だからです。

明示的な *encoding* 引数があると、結果は指定されたエンコード方式によるバイト文字列となります。引数を常に指定するよう推奨します。表現不可能なテキストデータの場合に `UnicodeError` が送出されるのを避けるため、*encoding* 引数は "utf-8" に指定するべきです。

`toprettyxml([indent[, newl]])`

美しく出力されたバージョンの文書を返します。*indent* はインデントを行うための文字で、デフォルトはタブです; *newl* には行末で出力される文字列を指定し、デフォルトは `n` です。

2.1 で追加された仕様です。

2.3 で追加された仕様: *encoding* 引数の追加; `toxml` を参照

以下の標準 DOM メソッドは、`xml.dom.minidom` では特別な注意をする必要があります:

`cloneNode(deep)`

このメソッドは Python 2.0 にパッケージされているバージョンの `xml.dom.minidom` にはありましたが、これには深刻な障害があります。以降のリリースでは修正されています。

### 13.7.2 DOM の例

以下のプログラム例は、かなり現実的な単純なプログラムの例です。特にこの例に関しては、DOM の柔軟性をあまり活用してはいません。

```
import xml.dom.minidom

document = """\
<slideshow>
<title>Demo slideshow</title>
<slide><title>Slide title</title>
<point>This is a demo</point>
<point>Of a program for processing slides</point>
</slide>

<slide><title>Another demo slide</title>
<point>It is important</point>
<point>To have more than</point>
<point>one slide</point>
</slide>
</slideshow>
"""

dom = xml.dom.minidom.parseString(document)

def getText(nodelist):
    rc = ""
    for node in nodelist:
        if node.nodeType == node.TEXT_NODE:
            rc = rc + node.data
    return rc

def handleSlideshow(slideshow):
    print "<html>"
    handleSlideshowTitle(slideshow.getElementsByTagName("title")[0])
    slides = slideshow.getElementsByTagName("slide")
    handleToc(slides)
    handleSlides(slides)
    print "</html>"

def handleSlides(slides):
    for slide in slides:
```



```

        handleSlide(slide)

def handleSlide(slide):
    handleSlideTitle(slide.getElementsByTagName("title")[0])
    handlePoints(slide.getElementsByTagName("point"))

def handleSlideshowTitle(title):
    print "<title>%s</title>" % getText(title.childNodes)

def handleSlideTitle(title):
    print "<h2>%s</h2>" % getText(title.childNodes)

def handlePoints(points):
    print "<ul>"
    for point in points:
        handlePoint(point)
    print "</ul>"

def handlePoint(point):
    print "<li>%s</li>" % getText(point.childNodes)

def handleToc(slides):
    for slide in slides:
        title = slide.getElementsByTagName("title")[0]
        print "<p>%s</p>" % getText(title.childNodes)

handleSlideshow(dom)

```

### 13.7.3 minidom と DOM 標準

xml.dom.minidom モジュールは、本質的には DOM 1.0 互換の DOM に、いくつかの DOM 2 機能 (主に名前空間機能) を追加したものです。

Python における DOM インタフェースは率直なものです。以下の対応付け規則が適用されます:

- インタフェースはインスタンスオブジェクトを介してアクセスされます。アプリケーション自身から、クラスをインスタンス化してはなりません; Document オブジェクト上で利用可能な生成関数 (creator function) を使わなければなりません。導出インタフェースでは基底インタフェースの全ての演算 (および属性) に加え、新たな演算をサポートします。
- 演算はメソッドとして使われます。DOM では in パラメタのみを使うので、引数は通常の順番 (左から右へ) で渡されます。オプション引数はありません。void 演算は None を返します。
- IDL 属性はインスタンス属性に対応付けられます。OMG IDL 言語における Python への対応付けとの互換性のために、属性 foo はアクセサメソッド \_get\_foo() および \_set\_foo() でもアクセスできます。readonly 属性は変更してはなりません; とはいえ、これは実行時には強制されません。
- short int、unsigned int、unsigned long long、および boolean 型は、全て Python 整数オブジェクトに対応付けられます。
- DOMString 型は Python 文字列型に対応付けられます。xml.dom.minidom ではバイト文字列 (byte string) および Unicode 文字列のどちらかに対応づけられますが、通常 Unicode 文字列を生成します。DOMString 型の値は、W3C の DOM 仕様で、IDL null 値になってもよいとされている場所では None になることもあります。
- const 宣言を行うと、(xml.dom.minidom.Node.PROCESSING\_INSTRUCTION\_NODE のように) 対応するスコープ内の変数に対応付けを行います; これらは変更してはなりません。

- `DOMException` は現状では `xml.dom.minidom` でサポートされていません。その代わり、`xml.dom.minidom` は、`TypeError` や `AttributeError` といった標準の Python 例外を使います。
- `NodeList` オブジェクトは Python の組み込みリスト型を使って実装されています。Python 2.2 からは、これらのオブジェクトは DOM 仕様で定義されたインタフェースを提供していますが、それ以前のバージョンの Python では、公式の API をサポートしていません。しかしながら、これらの API は W3C 勧告で定義されたインタフェースよりも “Python 的な” ものになっています。

以下のインタフェースは `xml.dom.minidom` では全く実装されていません:

- `DOMTimeStamp`
- `DocumentType` (added in Python 2.1)
- `DOMImplementation` (added in Python 2.1)
- `CharacterData`
- `CDATASection`
- `Notation`
- `Entity`
- `EntityReference`
- `DocumentFragment`

これらの大部分は、ほとんどの DOM のユーザにとって一般的な用途として有用とはならないような XML 文書内の情報を反映しています。

## 13.8 `xml.dom.pulldom` — 部分的な DOM ツリー構築のサポート

2.0 で追加された仕様です。

`xml.dom.pulldom` では、SAX イベントから、文書の文書オブジェクトモデル表現の選択された一部分だけを構築できるようにします。

```
class PullDOM([documentFactory])
    xml.sax.handler.ContentHandler 実装です ...
```

```
class DOMEventStream(stream, parser, bufsize)
    ...
```

```
class SAX2DOM([documentFactory])
    xml.sax.handler.ContentHandler 実装です ...
```

```
parse(stream_or_string[, parser[, bufsize]])
    ...
```

```
parseString(string[, parser])
    ...
```

```
default_bufsize
    parse() の bufsize パラメタのデフォルト値です。 2.1 で変更された仕様: この変数の値は parse() を呼び出す前に変更することができ、その場合新たな値が効果を持つようになります
```

### 13.8.1 DOMEventStream オブジェクト

```
getEvent()  
...  
expandNode(node)  
...  
reset()  
...
```

## 13.9 xml.sax — SAX2 パーサのサポート

2.0 で追加された仕様です。

xml.sax パッケージは Python 用の Simple API for XML (SAX) インターフェースを実装した数多くのモジュールを提供しています。またパッケージには SAX 例外と SAX API 利用者が頻繁に利用するであろう有用な関数群も含まれています。

その関数群は以下の通りです:

**make\_parser([*parser\_list*])**

SAX XMLReader オブジェクトを作成して返します。パーサには最初に見つかったものが使われます。*parser\_list* を指定する場合は、`create_parser()` 関数を含んでいるモジュール名のシーケンスを与える必要があります。*parser\_list* のモジュールはデフォルトのパーサのリストに優先して使用されます。

**parse(*filename\_or\_stream*, *handler*[, *error\_handler*])**

SAX パーサを作成してドキュメントをパースします。*filename\_or\_stream* として指定するドキュメントはファイル名、ファイル・オブジェクトのいずれでもかまいません。*handler* パラメータには SAX ContentHandler のインスタンスを指定します。*error\_handler* には SAX ErrorHandler のインスタンスを指定します。これが指定されていないときは、すべてのエラーで SAXParseException 例外が発生します。関数の戻り値はなく、すべての処理は *handler* に渡されます。

**parseString(*string*, *handler*[, *error\_handler*])**

`parse()` に似ていますが、こちらはパラメータ *string* で指定されたバッファをパースします。

典型的な SAX アプリケーションでは 3 種類のオブジェクト (リーダ、ハンドラ、入力元) が用いられます (ここで言うリーダとはパーサを指しています)。言い換えると、プログラムはまず入力元からバイト列、あるいは文字列を読み込み、一連のイベントを発生させます。発生したイベントはハンドラ・オブジェクトによって振り分けられます。さらに言い換えると、リーダがハンドラのメソッドを呼び出すわけです。つまり SAX アプリケーションには、リーダ・オブジェクト、(作成またはオープンされる) 入力元のオブジェクト、ハンドラ・オブジェクト、そしてこれら 3 つのオブジェクトを連携させることが必須なのです。前処理の最後の段階でリーダは入力をパースするために呼び出されます。パースの過程で入力データの構造、構文にもとづいたイベントにより、ハンドラ・オブジェクトのメソッドが呼び出されます。

これらのオブジェクトは (通常アプリケーション側でインスタンスを作成しない) インターフェースに相当するものです。Python はインターフェースという明確な概念を提供していないため、形としてはクラスが用いられています。しかし提供されるクラスを継承せずに、アプリケーション側で独自に実装することも可能です。InputSource、Locator、Attributes、AttributesNS、XMLReader の各インターフェースは xml.sax.xmlreader モジュールで定義されています。ハンドラ・インターフェースは xml.sax.handler で定義されています。しばしばアプリケーション側で直接インスタンスが作成される InputSource とハンドラ・クラスは利便性のため xml.sax にも含まれています。これらのインターフェースに関しては後に解説します。

このほかに `xml.sax` は次の例外クラスも提供しています。

**`exception SAXException(msg[, exception])`**

XML エラーと警告をカプセル化します。このクラスには XML パーサとアプリケーションで発生するエラーおよび警告の基本的な情報を持たせることができます。また機能追加や地域化のためにサブクラス化することも可能です。なお `ErrorHandler` で定義されているハンドラがこの例外のインスタンスを受け取ることに注意してください。実際に例外を発生させることは必須でなく、情報のコンテナとして利用されることもあるからです。

インスタンスを作成する際 `msg` はエラー内容を示す可読データにしてください。オプションの `exception` パラメータは `None` もしくはパース用コードで補足、渡って来る情報でなければなりません。

このクラスは SAX 例外の基底クラスになります。

**`exception SAXParseException(msg, exception, locator)`**

パースエラー時に発生する `SAXException` のサブクラスです。パースエラーに関する情報として、このクラスのインスタンスが `SAX ErrorHandler` インターフェースのメソッドに渡されます。このクラスは `SAXException` 同様 `SAX Locator` インターフェースもサポートしています。

**`exception SAXNotRecognizedException(msg[, exception])`**

`SAX XMLReader` が認識できない機能やプロパティに遭遇したとき発生させる `SAXException` のサブクラスです。`SAX アプリケーション`や拡張モジュールにおいて同様の目的にこのクラスを利用することもできます。

**`exception SAXNotSupportedException(msg[, exception])`**

`SAX XMLReader` が要求された機能をサポートしていないとき発生させる `SAXException` のサブクラスです。`SAX アプリケーション`や拡張モジュールにおいて同様の目的にこのクラスを利用することもできます。

参考資料:

*SAX: The Simple API for XML*

(<http://www.saxproject.org/>)

SAX API 定義に関し中心となっているサイトです。Java による実装とオンライン・ドキュメントが提供されています。実装と SAX API の歴史に関する情報のリンクも掲載されています。

`xml.sax.handler` モジュール (13.10 節):

アプリケーションが提供するオブジェクトのインターフェース定義

`xml.sax.saxutils` モジュール (13.11 節):

SAX アプリケーション向けの有用な関数群

`xml.sax.xmlreader` モジュール (13.12 節):

パーサが提供するオブジェクトのインターフェース定義

### 13.9.1 SAXException オブジェクト

`SAXException` 例外クラスは以下のメソッドをサポートしています。

**`getMessage()`**

エラー状態を示す可読メッセージを返します。

**`getException()`**

カプセル化した例外オブジェクトまたは `None` を返します。

## 13.10 xml.sax.handler — SAX ハンドラの基底クラス

2.0 で追加された仕様です。

SAX API はコンテンツ・ハンドラ、DTD ハンドラ、エラー・ハンドラ、エンティティ・リゾルバという 4 つのハンドラを規定しています。通常アプリケーション側で実装する必要があるのは、これらのハンドラが発生させるイベントのうち、処理したいものへのインターフェースだけです。インターフェースは 1 つのオブジェクトにまとめることも、複数のオブジェクトに分けることも可能です。ハンドラはすべてのメソッドがデフォルトで実装されるように、xml.sax で提供される基底クラスを継承しなくてはなりません。

### class ContentHandler

アプリケーションにとって最も重要なメインの SAX コールバック・インターフェースです。このインターフェースで発生するイベントの順序はドキュメント内の情報の順序を反映しています。

### class DTDHandler

DTD イベントのハンドラです。

未構文解析エンティティや属性など、パースに必要な DTD イベントの抽出だけをおこなうインターフェースです。

### class EntityResolver

エンティティ解決用の基本インターフェースです。このインターフェースを実装したオブジェクトを作成しパーサに登録することで、パーサはすべての外部エンティティを解決するメソッドを呼び出すようになります。

### class ErrorHandler

エラーや警告メッセージをアプリケーションに通知するためにパーサが使用するインターフェースです。このオブジェクトのメソッドが、エラーをただちに例外に変換するか、あるいは別の方法で処理するかの制御をしています。

これらのクラスに加え、xml.sax.handler は機能やプロパティ名のシンボル定数を提供しています。

### feature\_namespaces

値: "http://xml.org/sax/features/namespaces"

true: 名前空間の処理を有効にする。

false: オプションで名前空間の処理を無効にする (暗黙に namespace-prefixes も無効にする - デフォルト)。

アクセス: (パース時) リードオンリー; (パース時以外) 読み書き可

### feature\_namespace\_prefixes

値: "http://xml.org/sax/features/namespace-prefixes"

true: 名前空間宣言で用いられているオリジナルのプリフィックス名と属性を通知する。

false: 名前空間宣言で用いられている属性を通知しない。オプションでオリジナルのプリフィックス名も通知しない (デフォルト)。

アクセス: (パース時) リードオンリー; (パース時以外) 読み書き可

### feature\_string\_interning

値: "http://xml.org/sax/features/string-interning" true: すべての要素名、プリフィックス、属性、名前、名前空間、URI、ローカル名を組み込みの intern 関数を使ってシンボルに登録する。

false: 名前のすべてを必ずしもシンボルに登録しない (デフォルト)。

アクセス: (パース時) リードオンリー; (パース時以外) 読み書き可

### feature\_validation

値: "http://xml.org/sax/features/validation"

true: すべての妥当性検査エラーを通知する (external-general-entities と external-parameter-entities が暗

黙の前提になっている)。

false: 妥当性検査エラーを通知しない。

アクセス: (パース時) リードオンリー; (パース時以外) 読み書き可

#### **feature\_external\_ges**

値: "http://xml.org/sax/features/external-general-entities"

true: 外部一般 (テキスト) エンティティの取り込みをおこなう。

false: 外部一般エンティティを取り込まない。

アクセス: (パース時) リードオンリー; (パース時以外) 読み書き可

#### **feature\_external\_pes**

値: "http://xml.org/sax/features/external-parameter-entities"

true: 外部 DTD サブセットを含むすべての外部パラメータ・エンティティの取り込みをおこなう。

false: 外部パラメータ・エンティティおよび外部 DTD サブセットを取り込まない。

アクセス: (パース時) リードオンリー; (パース時以外) 読み書き可

#### **all\_features**

すべての機能の一覧。

#### **property\_lexical\_handler**

値: "http://xml.org/sax/properties/lexical-handler"

data type: xml.sax.sax2lib.LexicalHandler (Python 2 では未サポート)

description: コメントなど字句解析イベント用のオプション拡張ハンドラ。

アクセス: 読み書き可

#### **property\_declaration\_handler**

Value: "http://xml.org/sax/properties/declaration-handler"

data type: xml.sax.sax2lib.DeclHandler (Python 2 では未サポート)

description: ノーテーションや未解析エンティティをのぞく DTD 関連イベント用のオプション拡張ハンドラ。

access: read/write

#### **property\_dom\_node**

Value: "http://xml.org/sax/properties/dom-node"

data type: org.w3c.dom.Node (Python 2 では未サポート)

description: パース時は DOM イテレータにおけるカレント DOM ノード、非パース時はルート DOM ノードを指す。

アクセス: (パース時) リードオンリー; (パース時以外) 読み書き可

#### **property\_xml\_string**

値: "http://xml.org/sax/properties/xml-string"

データ型: 文字列

説明: カレント・イベントの元になったリテラル文字列

アクセス: リードオンリー

#### **all\_properties**

既知のプロパティ名の全リスト。

### 13.10.1 ContentHandler オブジェクト

ContentHandler はアプリケーション側でサブクラス化して利用することが前提になっています。パーサは入力ドキュメントのイベントにより、それぞれに対応する以下のメソッドを呼び出します。

**setDocumentLocator**(*locator*)



アプリケーションにドキュメント・イベントの発生位置を通知するためにパーサから呼び出されます。SAX パーサによるロケータの提供は強く推奨されています (必須ではありません)。もし提供する場合は、DocumentHandler インターフェースのどのメソッドよりも先にこのメソッドが呼び出されるようにしなければなりません。

アプリケーションはパーサがエラーを通知しない場合でもロケータによって、すべてのドキュメント関連イベントの終了位置を知ることが可能になります。典型的な利用方法としては、アプリケーション側でこの情報を使い独自のエラーを発生させること (文字コンテンツがアプリケーション側で決めた規則に沿っていない場合等) があげられます。しかしロケータが返す情報は検索エンジンなどで利用するものとしてはおそらく不十分でしょう。

ロケータが正しい情報を返すのは、インターフェースからイベントの呼出しが実行されている間だけです。それ以外のときは使用すべきではありません。

**startDocument()**

ドキュメントの開始通知を受け取ります。

SAX パーサはこのインターフェースや DTDHandler のどのメソッド (setDocumentLocator()) を除く) よりも先にこのメソッドを一度だけ呼び出します。

**endDocument()**

ドキュメントの終了通知を受け取ります。

SAX パーサはこのメソッドを一度だけ、パース過程の最後に呼び出します。パーサは (回復不能なエラーで) パース処理を中断するか、あるいは入力のために到達するまでこのメソッドを呼び出しません。

**startPrefixMapping(prefix, uri)**

プリフィックスと URI の名前空間の関連付けを開始します。

このイベントから返る情報は通常の名前空間処理では使われません。SAX XML リードは feature\_namespaces 機能が有効になっている場合 (デフォルト)、要素と属性名のプリフィックスを自動的に置換するようになっています。

しかしアプリケーション側でプリフィックスを文字データや属性値の中で扱う必要が生じることもあります。この場合プリフィックスの自動展開は保証されないため、必要に応じ startPrefixMapping() や endPrefixMapping() イベントからアプリケーションに提供される情報を用いてプリフィックスの展開をおこないます。

startPrefixMapping() と endPrefixMapping() イベントは相互に正しい入れ子関係になることが保証されていないので注意が必要です。すべての startPrefixMapping() は対応する startElement() の前に発生し、endPrefixMapping() イベントは対応する endElement() の後で発生しますが、その順序は保証されていません。

**endPrefixMapping(prefix)**

プリフィックスと URI の名前空間の関連付けを終了します。

詳しくは startPrefixMapping() を参照してください。このイベントは常に対応する endElement() の後で発生しますが、複数の endPrefixMapping() イベントの順序は特に保証されません。

**startElement(name, attrs)**

非名前空間モードで要素の開始を通知します。

name パラメータには要素型の raw XML 1.0 名を文字列として、attrs パラメータには要素の属性を保持する Attributes インターフェース オブジェクトをそれぞれ指定します。attrs として渡されたオブジェクトはパーサで再利用することも可能ですが、属性のコピーを保持するためにこれを参

照し続けるのは確実な方法ではありません。属性のコピーを保持したいときは *attrs* オブジェクトの `copy()` メソッドを用いてください。

**endElement**(*name*)

非名前空間モードで要素の終了を通知します。

*name* パラメータには `startElement()` イベント同様の要素型名を指定します。

**startElementNS**(*name*, *qname*, *attrs*)

名前空間モードで要素の開始を通知します。

*name* パラメータには要素型を (*uri*, *localname*) のタプルとして、*qname* パラメータにはソース・ドキュメントで用いられている raw XML 1.0 名、*attrs* には要素の属性を保持する `AttributesNS` インターフェースのインスタンスをそれぞれ指定します。要素に関連付けられた名前空間がないときは、*name* コンポーネントの *uri* が `None` になります。*attrs* として渡されたオブジェクトはパーサで再利用することも可能ですが、属性のコピーを保持するためにこれを参照し続けるのは確実な方法ではありません。属性のコピーを保持したいときは *attrs* オブジェクトの `copy()` メソッドを用いてください。

`feature_namespace_prefixes` 機能が有効になっていなければ、パーサで *qname* を `None` にセットすることも可能です。

**endElementNS**(*name*, *qname*)

非名前空間モードで要素の終了を通知します。

*name* パラメータには `startElementNS()` イベント同様の要素型を指定します。*qname* パラメータも同じです。

**characters**(*content*)

文字データの通知を受け取ります。

パーサは文字データのチャンクごとにこのメソッドを呼び出して通知します。SAX パーサは一連の文字データを単一のチャンクとして返す場合と複数のチャンクに分けて返す場合がありますが、ロケータの情報が正しく保たれるように、一つのイベントの文字データは常に同じ外部エンティティのものでなければなりません。

*content* はユニコード文字列、バイト文字列のどちらでもかまいませんが、`expat` リード・モジュールは常にユニコード文字列を生成するようになっています。

注意: Python XML SIG が提供していた初期 SAX 1 では、このメソッドにもっと JAVA 風のインターフェースが用いられています。しかし Python で採用されている大半のパーサでは古いインターフェースを有効に使うことができないため、よりシンプルなものに変更されました。古いコードを新しいインターフェースに変更するには、古い *offset* と *length* パラメータでスライスせずに、*content* を指定するようにしてください。

**ignorableWhitespace**()

要素コンテンツに含まれる無視可能な空白文字の通知を受け取ります。

妥当性検査をおこなうパーサは無視可能な空白文字 (W3C XML 1.0 勧告のセクション 2.10 参照) のチャンクごとに、このメソッドを使って通知しなければなりません。妥当性検査をしないパーサもコンテンツモデルの利用とパースが可能な場合、このメソッドを利用することが可能です。

SAX パーサは一連の空白文字を単一のチャンクとして返す場合と複数のチャンクに分けて返す場合がありますが、ロケータの情報が正しく保たれるように、一つのイベントの文字データは常に同じ外部エンティティのものでなければなりません。

**processingInstruction**(*target*, *data*)

処理命令の通知を受け取ります。

パーサは処理命令が見つかるたびにこのメソッドを呼び出します。処理命令はメインのドキュメント要素の前や後にも発生することがあるので注意してください。

SAX パーサがこのメソッドを使って XML 宣言 (XML 1.0 のセクション 2.8) やテキスト宣言 (XML 1.0 のセクション 4.3.1) の通知をすることはありません。

**skippedEntity**(*name*)

スキップしたエンティティの通知を受け取ります。

パーサはエンティティをスキップするたびにこのメソッドを呼び出します。妥当性検査をしないプロセッサは (外部 DTD サブセットで宣言されているなどの理由で) 宣言が見当たらないエンティティをスキップします。すべてのプロセッサは `feature_external_ges` および `feature_external_pes` 属性の値によっては外部エンティティをスキップすることがあります。

### 13.10.2 DTDHandler オブジェクト

DTDHandler インスタンスは以下のメソッドを提供します。

**notationDecl**(*name*, *publicId*, *systemId*)

表記法宣言イベントの通知を捕捉します。

**unparsedEntityDecl**(*name*, *publicId*, *systemId*, *ndata*)

未構文解析エンティティ宣言イベントの通知を受け取ります Handle an unparsed entity declaration event.

### 13.10.3 EntityResolver オブジェクト

**resolveEntity**(*publicId*, *systemId*)

エンティティのシステム識別子を解決し、文字列として読み込んだシステム識別子あるいは InputSource オブジェクトのいずれかを返します。デフォルトの実装では *systemId* を返します。

### 13.10.4 ErrorHandler オブジェクト

このインターフェースのオブジェクトは XMLReader からのエラーや警告の情報を受け取るために使われます。このインターフェースを実装したオブジェクトを作成し XMLReader に登録すると、パーサは警告やエラーの通知のためにそのオブジェクトのメソッドを呼び出すようになります。エラーには警告、回復可能エラー、回復不能エラーの3段階があります。すべてのメソッドは SAXParseException だけをパラメータとして受け取ります。受け取った例外オブジェクトを raise することで、エラーや警告は例外に変換されることもあります。

**error**(*exception*)

パーサが回復可能なエラーを検知すると呼び出されます。このメソッドが例外を raise しないとパーサは継続されますが、アプリケーション側ではエラー以降のドキュメント情報を期待していないこともあります。パーサが処理を継続した場合、入力ドキュメント内のほかのエラーを見つけることができます。

**fatalError**(*exception*)

パーサが回復不能なエラーを検知すると呼び出されます。このメソッドが return した後、すぐにパーサを停止することが求められています。

**warning**(*exception*)

パーサが軽微な警告情報をアプリケーションに通知するために呼び出されます。このメソッドが return した後もパーサを継続し、ドキュメント情報をアプリケーションに送り続けるよう求められています。このメソッドで例外を発生させた場合、パーサは中断されてしまいます。

## 13.11 xml.sax.saxutils — SAX ユーティリティ

2.0 で追加された仕様です。

モジュール `xml.sax.saxutils` には SAX アプリケーションの作成に役立つ多くの関数やクラスも含まれており、直接利用したり、基底クラスとして使うことができます。

`escape(data[, entities])`

文字列データ内の ‘&’、‘<’、‘>’ をエスケープします。

オプションの *entities* パラメータに辞書を渡すことで、そのほかの文字をエスケープさせることも可能です。辞書のキーと値はすべて文字列で、キーに指定された文字は対応する値に置換されます。

`unescape(data[, entities])`

エスケープされた文字列 ‘&amp;’、‘&lt;’、‘&gt;’ を元の文字に戻します。

オプションの *entities* パラメータに辞書を渡すことで、そのほかの文字をエスケープさせることも可能です。辞書のキーと値はすべて文字列で、キーに指定された文字は対応する値に置換されます。2.3 で追加された仕様です。

`quoteattr(data[, entities])`

`escape()` に似ていますが、*data* は属性値の作成に使われます。戻り値はクオート済みの *data* で、置換する文字の追加も可能です。`quoteattr()` はクオートすべき文字を *data* の文脈から判断し、クオートすべき文字を残さないように文字列をエンコードします。

*data* の中にシングルのクオート、ダブルのクオートがあれば、両方ともエンコードし、全体をダブルクオートで囲みます。戻り値の文字列はそのまま属性値として利用できます。:

```
>>> print "<element attr=%s>" % quoteattr("ab ' cd \" ef")
<element attr="ab ' cd &quot; ef">
```

この関数は参照具象構文を使って、HTML や SGML の属性値を生成するのに便利です。2.2 で追加された仕様です。

`class XMLGenerator([out[, encoding]])`

このクラスは `ContentHandler` インターフェースの実装で、SAX イベントを XML ドキュメントに書き戻します。つまり、`XMLGenerator` をコンテンツ・ハンドラとして用いると、パースしたオリジナル・ドキュメントの複製が作れるのです。*out* に指定するのはファイル風のオブジェクトで、デフォルトは `sys.stdout` です。*encoding* は出力ストリームのエンコーディングで、デフォルトは ‘iso-8859-1’ です。

`class XMLFilterBase(base)`

このクラスは `XMLReader` とクライアント・アプリケーションのイベント・ハンドラとの間に位置するものとして設計されています。デフォルトでは何もせず、ただリクエストをリーダに、イベントをハンドラに、それぞれ加工せず渡すだけです。しかし、サブクラスでメソッドをオーバーライドすると、イベント・ストリームやリクエストを加工してから渡すように変更可能です。

`prepare_input_source(source[, base])`

この関数は引き数に入力ソース、オプションとして URL を取り、読み取り可能な解決済み `InputSource` オブジェクトを返します。入力ソースは文字列、ファイル風オブジェクト、`InputSource` のいずれでも良く、この関数を使うことで、パーサは様々な *source* パラメータを `parse()` に渡すことが可能になります。

## 13.12 xml.sax.xmlreader — XML パーサのインターフェース

2.0 で追加された仕様です。

各 SAX パーサは Python モジュールとして XMLReader インターフェースを実装しており、関数 `create_parser()` を提供しています。この関数は新たなパーサ・オブジェクトを生成する際、`xml.sax.make_parser()` から引き数なしで呼び出されます。

**class XMLReader()**

SAX パーサが継承可能な基底クラスです。

**class IncrementalParser()**

入力ソースをパースする際、すべてを一気に処理しないで、途中でドキュメントのチャンクを取得したいことがあります。SAX リーダは通常、ファイル全体を一気に読み込まずチャンク単位で処理するのですが、全体の処理が終わるまで `parse()` は return しません。つまり、IncrementalParser インターフェースは `parse()` にこのような排他的挙動を望まないときに使われます。

パーサのインスタンスが作成されると、`feed` メソッドを通じてすぐに、データを受け入れられるようになります。`close` メソッドの呼出しでパースが終わると、パーサは新しいデータを受け入れられるように、`reset` メソッドを呼び出されなければなりません。

これらのメソッドをパース処理の途中で呼び出すことはできません。つまり、パースが実行された後で、パーサから return する前に呼び出す必要があるのです。

なお、SAX 2.0 ドライバを書く人のために、XMLReader インターフェースの `parse` メソッドがデフォルトで、IncrementalParser の `feed`、`close`、`reset` メソッドを使って実装されています。

**class Locator()**

SAX イベントとドキュメントの位置を関連付けるインターフェースです。locator オブジェクトは `DocumentHandler` メソッドを呼び出している間だけ正しい情報を返し、それ以外とのときに呼び出すと、予測できない結果が返ります。情報を取得できない場合、メソッドは `None` を返すこともあります。

**class InputSource([systemId])**

XMLReader がエンティティを読み込むために必要な情報をカプセル化します。

このクラスには公開識別子、システム識別子、(場合によっては文字エンコーディング情報を含む) バイト・ストリーム、そしてエンティティの文字ストリームなどの情報が含まれます。

アプリケーションは `XMLReader.parse()` メソッドに渡す引き数、または `EntityResolver.resolveEntity` の戻り値としてこのオブジェクトを作成します。

`InputSource` はアプリケーション側に属します。XMLReader はアプリケーションから渡された `InputSource` オブジェクトの変更を許していませんが、コピーを作り、それを変更することは可能です。

**class AttributesImpl(attrs)**

`Attributes` interface (13.12.5 参照) の実装です。辞書風のオブジェクトで、`startElement()` 内で要素の属性表示をおこないます。多くの辞書風オブジェクト操作に加え、ほかにもインターフェースに記述されているメソッドを、多数サポートしています。このクラスのオブジェクトはリーダによってインスタンスを作成しなければなりません。また、`attrs` は属性名と属性値を含む辞書風オブジェクトでなければなりません。

**class AttributesNSImpl(attrs, qnames)**

`AttributesImpl` を名前空間認識型に改良したクラスで、`startElementNS()` に渡されます。`AttributesImpl` の派生クラスですが、`namespaceURI` と `localname`、この2つのタプルを解釈します。さらに、元のドキュメントに出てくる修飾名を返す多くのメソッドを提供します。このクラス



は `AttributesNS` interface ( section [13.12.6](#) 参照) の実装です。

### 13.12.1 XMLReader オブジェクト

`XMLReader` は次のメソッドをサポートします。:

**parse**(*source*)

入力ソースを処理し、SAX イベントを発生させます。*source* オブジェクトにはシステム識別子(入力ソースを特定する文字列 – 一般にファイル名や URL)、ファイル風オブジェクト、または `InputSource` オブジェクトを指定できます。`parse()` から `return` された段階で、入力データの処理は完了、パーサ・オブジェクトは破棄ないしリセットされます。なお、現在の実装はバイト・ストリームのみをサポートしており、文字ストリームの処理は将来の課題になっています。

**getContentHandler**()

現在の `ContentHandler` を返します。

**setContentHandler**(*handler*)

現在の `ContentHandler` をセットします。`ContentHandler` がセットされていない場合、コンテンツ・イベントは破棄されます。

**getDTDHandler**()

現在の `DTDHandler` を返します。

**setDTDHandler**(*handler*)

現在の `DTDHandler` をセットします。`DTDHandler` がセットされていない場合、DTD イベントは破棄されます。

**getEntityResolver**()

現在の `EntityResolver` を返します。

**setEntityResolver**(*handler*)

現在の `EntityResolver` をセットします。`EntityResolver` がセットされていない場合、外部エンティティとして解決されるべきものが、システム識別子として解釈されてしまうため、該当するものがなければ結果的にエラーとなります。

**getErrorHandler**()

現在の `ErrorHandler` を返します。

**setErrorHandler**(*handler*)

現在のエラー・ハンドラをセットします。`ErrorHandler` がセットされていない場合、エラーは例外を発生し、警告が表示されます。

**setLocale**(*locale*)

アプリケーションにエラーや警告のロカール設定を許可します。

SAX パーサにとって、エラーや警告の地域化は必須ではありません。しかし、パーサは要求されたロカールをサポートしていない場合、SAX 例外を発生させなければなりません。アプリケーションはパースの途中でロカールを変更することもできます。

**getFeature**(*featurename*)

機能 *featurename* の現在の設定を返します。その機能が認識できないときは、`SAXNotRecognizedException` を発生させます。広く使われている機能名の一覧はモジュール `xml.sax.handler` に書かれています。

**setFeature**(*featurename*, *value*)

機能名 *featurename* に値 *value* をセットします。その機能が認識できないときは、`SAXNotRecognizedException` を発生させます。また、パーサが指定された機能や設定をサポートしていないと



きは、*SAXNotSupportedException* を発生させます。

**getProperty**(*propertyname*)

属性名 *propertyname* の現在の値を返します。その属性が認識できないときは、*SAXNotRecognizedException* を発生させます。広く使われている属性名の一覧はモジュール `xml.sax.handler` に書かれています。

**setProperty**(*propertyname*, *value*)

属性名 *propertyname* に値 *value* をセットします。その機能が認識できないときは、*SAXNotRecognizedException* を発生させます。また、パーサが指定された機能や設定をサポートしていないときは、*SAXNotSupportedException* is raised を発生させます。

### 13.12.2 IncrementalParser オブジェクト

*IncrementalParser* のインスタンスは次の追加メソッドを提供します。:

**feed**(*data*)

*data* のチャンクを処理します。

**close**()

ドキュメントの終わりを決定します。終わりに達した時点でドキュメントが整形形式であるかどうかを判別、ハンドラを起動後、パース時に使用した資源を解放します。

**reset**()

このメソッドは `close` が呼び出された後、次のドキュメントをパース可能にするため、パーサのリセットするのに呼び出されます。`close` 後、`reset` を呼び出さずに `parse` や `feed` を呼び出した場合の戻り値は未定義です。

### 13.12.3 Locator オブジェクト

*Locator* のインスタンスは次のメソッドを提供します。:

**getColumnNumber**()

現在のイベントが終了する列番号を返します。

**getLineNumber**()

現在のイベントが終了する行番号を返します。

**getPublicId**()

現在の文書イベントの公開識別子を返します。

**getSystemId**()

現在のイベントのシステム識別子を返します。

### 13.12.4 InputSource オブジェクト

**setPublicId**(*id*)

この *InputSource* の公開識別子をセットします。

**getPublicId**()

この *InputSource* の公開識別子を返します。

**setSystemId**(*id*)

この *InputSource* のシステム識別子をセットします。

**getSystemId**()

この *InputSource* のシステム識別子を返します。

**setEncoding**(*encoding*)

この `InputSource` の文字エンコーディングをセットします。

指定するエンコーディングは XML エンコーディング宣言として定義された文字列でなければなりません (セクション 4.3.3 の XML 勧告を参照)。

`InputSource` のエンコーディング属性は、`InputSource` がたとえ文字ストリームを含んでいたとしても、無視されます。

**getEncoding**()

この `InputSource` の文字エンコーディングを取得します。

**setByteStream**(*bytefile*)

この入力ソースのバイトストリーム (Python のファイル風オブジェクトですが、バイト列と文字の相互変換はサポートしません) を設定します。

なお、文字ストリームが指定されても SAX パーサは無視し、バイト・ストリームを使って指定された URI に接続しようとしています。

アプリケーション側でバイト・ストリームの文字エンコーディングを知っている場合は、`setEncoding` メソッドを使って指定する必要があります。

**getByteStream**()

この入力ソースのバイトストリームを取得します。

`getEncoding` メソッドは、このバイト・ストリームの文字エンコーディングを返します。認識できないときは `None` を返します。

**setCharacterStream**(*charfile*)

この入力ソースの文字ストリームをセットします (ストリームは Python 1.6 の Unicode-wrapped なファイル風オブジェクトで、ユニコード文字列への変換をサポートしていなければなりません)。

なお、文字ストリームが指定されても SAX パーサは無視、システム識別子とみなし、バイト・ストリームを使って URI に接続しようとしています。

**getCharacterStream**()

この入力ソースの文字ストリームを取得します。

### 13.12.5 The Attributes インターフェース

`Attributes` オブジェクトは `copy()`、`get()`、`has_key()`、`items()`、`keys()`、`values()` などを含む、マッピング・プロトコルの一部を実装したものです。さらに次のメソッドも提供されています。:

**getLength**()

属性の数を返す。

**getNames**()

属性の名前を返す。

**getType**(*name*)

属性名 *name* のタイプを返す。通常は `'CDATA'`。

**getValue**(*name*)

属性 *name* の値を返す。

### 13.12.6 AttributesNS インターフェース

このインターフェースは `Attributes interface` (セクション 13.12.5 参照) のサブタイプです。 `Attributes` インターフェースがサポートしているすべてのメソッドは `AttributesNS` オブジェクトでも利用可能です。

そのほか、次のメソッドがサポートされています。:

`getValueByQName (name)`

修飾名の値を返す。

`getNameByQName (name)`

修飾名 *name* に対応する (*namespace*, *localname*) のペアを返す。

`getQNameByName (name)`

(*namespace*, *localname*) のペアに対応する修飾名を返す。

`getQNames()`

すべての属性の修飾名を返す。

## 13.13 xmllib — XML ドキュメントのパーサ

リリース 2.0 以降で撤廃された仕様です。代わりに `xml.sax` を使ってください。新しい XML パッケージは XML 1.0 をフルにサポートしています。

1.5.2 で変更された仕様: 名前空間のサポートを追加

このモジュールには XML (Extensible Markup Language) 形式で記述されたテキストファイルのパーサに必要な基本機能を提供する `XMLParser` クラスが定義されています。

`class XMLParser()`

`XMLParser` `XMLParser` クラスのインスタンス生成は引数を指定せずにおこないます。<sup>1</sup>

このクラスは次のインターフェース・メソッドとインスタンス変数を提供しています。

**attributes**

要素名のマッピングへのマッピング・オブジェクトで、デフォルトは空の辞書です。このデフォルト変数はすべての `XMLParser` インスタンスで共有されるため、継承せずにオーバーライドする必要があります。なお、要素名のマッピングの方は、その要素に妥当な属性名とそのデフォルトの属性値をマッピングしており、デフォルト値がない場合は `None` になります。

**elements**

要素名からタプルへのマッピングです。タプルには要素の開始タグと終了タグをそれぞれ処理する関数、または `unknown_starttag()` や `unknown_endtag()` 呼出された場合に使用する `None` が含まれます。デフォルトは空の辞書になっています。このデフォルト変数はすべての `XMLParser` インスタンスで共有されるため、継承せずにオーバーライドする必要があります。

**entitydefs**

エンティティ名からその値へのマッピングです。デフォルトで `'lt'`、`'gt'`、`'amp'`、`'quot'`、and `'apos'` の定義がされています。

**reset()**

インスタンスをリセットします。同時に未処理のデータはすべて失われます。このメソッドはインスタンス生成時、暗黙裡に呼び出されます。

**setnomoretags()**

タグの処理を中止します。このメソッドの呼び出し以降、すべての入力はリテラル (CDATA) として扱われます。

---

<sup>1</sup>実際には、パーサに非標準的な形式のドキュメントを許容するための、いくつかのキーワード引数を指定できるようになっており、その内容は次の通りです。これらのデフォルト値はすべて 0 (false) ですが、最後のキーワード引数だけは 1 (true) になっています。 `accept_unquoted_attributes` (特定の属性がクオートされていなくても受け入れるようにする)、 `accept_missing_endtag_name` (`</>` のようなタグ名なしの終了タグを許容する)、 `map_case` (タグや属性名が大文字で書かれていても、小文字で解釈する)、 `accept_utf8` (UTF-8 文字列の入力を許容する。これは XML 標準規格で必須とされている内容ですが、Python は今のところ UTF-8 を適切に処理できないため、デフォルトでは無効になっています。)、 `translate_attribute_references` (属性値として使われている文字列や実体参照はそれ以上解釈しない。)

**setliteral()**

リテラルモード (CDATA mode) に入ります。このモードは最後に処理した開始タグに対応する終了タグが見つかった時点で自動的に終了します。

**feed(*data*)**

テキストをパーサに送ります。完全なタグで構成された部分までを処理し、不完全なタグは次のデータが来るか `close()` が呼び出されるまでバッファリングされます。

**close()**

現在バッファリング中のデータの前に end-of-file ファイルが来たものと強制的にみなします。このメソッドは派生クラスにおいて、入力終了時におこなう追加処理のために再定義されることもあります。再定義したメソッドの中で必ずこの `close()` を呼び出さなければなりません。

**translate\_references(*data*)**

*data* の中にある実体参照と文字参照をすべて変換し、変換後の文字列を返します。

**getnamespace()**

現在有効な名前空間の短縮名から名前空間 URI へのマッピングを返します。

**handle\_xml(*encoding, standalone*)**

このメソッドは '`<?xml ...?>`' タグを処理します。引数はタグ中のエンコーディングの値とスタンドアロン宣言です。エンコーディングとスタンドアロン属性はともにオプションです。デフォルトはそれぞれ `None` と文字列 '`no`' になります。

**handle\_doctype(*tag, pubid, syslit, data*)**

このメソッドは '`<!DOCTYPE ...>`' 宣言を処理します。引数にはルート要素のタグ名、公式公開識別子 (指定されていない場合は `None`)、システム識別子、および内部 DTD サブセットを処理せずそのまま文字列で指定します (指定されていない場合は `None`)。

**handle\_starttag(*tag, method, attributes*)**

このメソッドはインスタンス変数 `elements` にハンドラが定義されている開始タグを処理します。*tag* はタグ名で、*method* は開始タグの意味解析サポートに使われる関数 (メソッド) です。*attributes* は `<>` タグ内に記述されている属性名がキー、属性値を値として持つ辞書になります。文字およびエンティティの参照は解釈されます。たとえば、開始タグ `<A HREF="http://www.cwi.nl/">` に対するこのメソッドの呼出しは `handle_starttag('A', self.elements['A'][0], {'HREF': 'http://www.cwi.nl/'})` となります。基本的な実装は単純で、引数に *attributes* を使い、*method* を呼び出すだけのものです。

**handle\_endtag(*tag, method*)**

このメソッドはインスタンス変数 `elements` にハンドラが定義されている終了タグを処理します。*tag* はタグ名で、*method* は終了タグの意味解析サポートに使われる関数 (メソッド) です。たとえば、終了タグ `</A>` に対するこのメソッドの呼出しは `handle_endtag('A', self.elements['A'][1])` となります。基本的な実装は単純で、*method* を呼び出すだけのものです。

**handle\_data(*data*)**

これは任意のデータを処理するメソッドです。派生クラスでオーバーライドして使います。基底クラスの実装では何もおこないません。

**handle\_charref(*ref*)**

このメソッドは '`&#ref;`' 形式の文字参照を処理します。*ref* は 10 進数または頭に '`x`' を付けた 16 進数のどちらでもかまいません。基本実装において *ref* の値として許される範囲は 0 から 255 までに限定されています。この値が ASCII に変換された後、その文字列を引数にしてメソッド `handle_data()` が呼び出されます。*ref* の値が許容範囲外のときは、エラーハンドリングのために `unknown_charref(ref)` が呼び出されます。ASCII 範囲外の文字をサポートするには、サブクラスでこのメソッドをオーバーライドする必要があります。

#### `handle_comment (comment)`

コメントを検出すると、このメソッドが呼び出されます。引数 *comment* は、デリミタ '`<!--`' から '`-->`' の間にある文字列であり、デリミタ自体は含まれません。たとえばコメント '`<!--text-->`' を処理する場合、引数は '`text`' でこのメソッドが呼び出されます。このメソッドはデフォルトでは何もしません。

#### `handle_cdata (data)`

CDATA 要素を検出すると、このメソッドが呼び出されます。引数 *data* は、デリミタ '`<![CDATA[`' と '`]]>`' の間にある文字列であり、デリミタ自体は含まれません。たとえばエンティティ '`<![CDATA[text]]>`' を処理する場合、引数は '`text`' でこのメソッドが呼び出されます。このメソッドはデフォルトでは何もしないため、オーバーライドして使用します。

#### `handle_proc (name, data)`

処理命令 (PI) を検出すると、このメソッドが呼び出されます。引数 *name* は処理命令のターゲット、*data* は、処理命令ターゲットと終了デリミタの間にある文字列であり、デリミタ自体は含まれません。たとえば命令 '`<?XML text?>`' を処理する場合、引数は '`XML`' と '`text`' の2つでこのメソッドが呼び出されます。このメソッドはデフォルトでは何もしません。なおドキュメントの冒頭に現れる '`<?xml ..?>`' は `handle_xml()` で処理されます。

#### `handle_special (data)`

宣言を検出すると、このメソッドが呼び出されます。引数 *data* は、デリミタ '`<!`' と '`>`' の間にある文字列であり、デリミタ自体は含まれません。たとえばエンティティ宣言 '`<!ENTITY text>`' を処理する場合、引数は '`ENTITY text`' でこのメソッドが呼び出されます。このメソッドはデフォルトでは何もしないため、オーバーライドして使用します。なおドキュメントの冒頭に現れる '`<!DOCTYPE ...>`' は別途処理しなければなりません。

#### `syntax_error (message)`

シンタックスエラーが発生すると、このメソッドが呼び出されます。引数 *message* はエラー内容を知らせるテキストです。このメソッドはデフォルトで `RuntimeError` 例外を発生させます。メソッドをオーバーライドして、`return` するように変更することも可能です。なお、このメソッドが呼び出されるのは回復可能なエラーの場合だけです。回復不能なエラー発生したときは、`syntax_error()` を呼び出すことなく `RuntimeError` が発生します。

#### `unknown_starttag (tag, attributes)`

未知の開始タグを検出すると、このメソッドが呼び出されます。派生クラスでオーバーライドして使います。基底クラスの実装では何もおこないません。

#### `unknown_endtag (tag)`

未知の終了タグを検出すると、このメソッドが呼び出されます。派生クラスでオーバーライドして使います。基底クラスの実装では何もおこないません。

#### `unknown_charref (ref)`

解決できない文字参照を検出すると、このメソッドが呼び出されます。派生クラスでオーバーライドして使います。基底クラスの実装では何もおこないません。

#### `unknown_entityref (ref)`

解決できない実体参照を検出すると、このメソッドが呼び出されます。派生クラスでオーバーライドして使います。基底クラスの実装では `syntax_error()` でエラーを通知するようになっています。

参考資料:

Extensible Markup Language (XML) 1.0

(<http://www.w3.org/TR/REC-xml>)

World Wide Web Consortium (W3C) が送出した XML の仕様書で、XML のシンタックスと処理に必要な内容が定義されています。仕様書の翻訳など、そのほかの文献は <http://www.w3.org/XML/>

で参照できます。

*Python and XML Processing*

(<http://www.python.org/topics/xml/>)

Python XML トピックガイドは Python で XML を扱うための情報と XML 関連情報源へのリンクを数多く提供しています。

*SIG for XML Processing in Python*

(<http://www.python.org/sigs/xml-sig/>)

Python XML Special Interest Group は Python で XML を扱うための開発に多大な貢献をしています。

### 13.13.1 XML 名前空間

このモジュールは XML 名前空間勧告に定義された名前空間をサポートします。

XML 名前空間に定義されたタグ名と属性名は、名前空間 (名前空間を定義した URL) の後にスペース置き、それに続くタグ名または属性名として扱われます。たとえば、タグが `<html xmlns='http://www.w3.org/TR/REC-html40'>` の場合、タグ名は `'http://www.w3.org/TR/REC-html40 html'` として処理され、この要素の中にタグ `<html:a href='http://frob.com'>` が現れた場合、タグ名は `'http://www.w3.org/TR/REC-html40 a'` で、属性名は `'http://www.w3.org/TR/REC-html40 href'` となります。

古い XML 名前空間勧告ドラフトに基いたものも処理可能ですが、警告が発せられます。

参考資料:

*Namespaces in XML*

(<http://www.w3.org/TR/REC-xml-names/>)

この W3 コンソーシアム 勧告には、XML 名前空間の正確なシンタクスと処理に必要な内容について述べられています。



# マルチメディアサービス

この章で記述されているモジュールは、主にマルチメディアアプリケーションに役立つさまざまなアルゴリズムまたはインターフェイスを実装しています。これらのモジュールはインストール時の自由裁量に応じて利用できます。

以下に概要を示します。

<b>audioop</b>	生のオーディオデータの操作
<b>imageop</b>	生の画像データを操作する。
<b>aifc</b>	AIFF あるいは AIFC フォーマットのオーディオファイルの読み書き
<b>sunau</b>	Sun AU サウンドフォーマットへのインターフェイス
<b>wave</b>	WAV サウンドフォーマットへのインターフェイス
<b>chunk</b>	IFF チャンクデータの読み込み。
<b>colorsys</b>	RGB 他の色体系間の変換。
<b>rgbimg</b>	“SGI RGB” フォーマットの画像ファイルを読み書きします (ですが、このモジュールは SGI 特有のも
<b>imghdr</b>	ファイルやバイトストリームに含まれる画像の形式を決定する。
<b>sndhdr</b>	サウンドファイルの識別
<b>ossaudiodev</b>	OSS 互換オーディオデバイスへのアクセス。

## 14.1 audioop — 生のオーディオデータの操作

audioop モジュールにはサウンドデータに対する便利な操作が定義されています。

このモジュールは、Python 文字列で保存された、8、16、32 ビットサイズの符号付き整数型からなるサウンドデータを処理します。

このデータは `al` と `sunaudiodev` モジュールで使用されるのと同じフォーマットです。

特に断わらない限り数値項目は整数です。

このモジュールは `u-LAW` と `Intel/DVI ADPCM` エンコードをサポートしています。

複雑な操作のうちいくつかは 16 ビットのサンプルに対してのみ働きますが、それ以外は常にサンプルサイズ (バイト数) を操作のパラメータとして渡します。

このモジュールは以下の変数と関数を定義しています：

### exception error

この例外はサウンドサンプルの未知のバイト数など、全てのエラーに対して発生します。

### add(fragment1, fragment2, width)

パラメータとして渡された 2 つのサンプルの和のデータを返します。

`width` はサンプルサイズのバイト数で、1、2、4 のうちのどれかです。2 つのデータは同じサンプルサイズでなければなりません。

### adpcm2lin(adpcmfragment, width, state)

Intel/DVI ADPCM フォーマットのデータを linear フォーマットにデコードします。ADPCM フォーマットについての詳細は `lin2adpcm()` の説明を参照して下さい。 *width* で指定したサイズで (*sample*, *newstate*) のタプルを返します。

**adpcm32lin**(*adpcmfragment*, *width*, *state*)

3 ビットの新しい ADPCM フォーマットのデータをデコードします。詳しくは `lin2adpcm3()` を参照して下さい。

**avg**(*fragment*, *width*)

データ内の全サンプルの平均値を返します。

**avgpp**(*fragment*, *width*)

データ内の全サンプルの最大振幅の平均値を返します。フィルタリングされていないなら、このルーチンが役立つかどうか疑問です。

**bias**(*fragment*, *width*, *bias*)

もとのデータのサンプル一つ一つにバイアスを加えたデータを返します。

**cross**(*fragment*, *width*)

引数として渡されたデータ内のゼロ-クロスポイントの数を返します。

**findfactor**(*fragment*, *reference*)

`rms(add(fragment, mul(reference, -F)))` が最小となるような *F* を返します。つまり、*reference* に掛けると *fragment* にできるだけ一致するような値を返します。*fragment* と *reference* は両方とも 2 バイトのサンプルでなければなりません。

このルーチンの実行に要する時間は `len(fragment)` に比例します。

**findfit**(*fragment*, *reference*)

*reference* が *fragment* (より長いデータ) の一部に一致するか確かめます。これは (概念的には) *fragment* から断片を切り出して、`findfactor()` を使って一番ふさわしいものを計算し、結果を最小にすることで実現しています。*fragment* と *reference* は両方とも 2 バイトのサンプルでなければなりません。`(offset, factor)` のタプルを返します。*offset* は最適な一致箇所が始まる *fragment* のオフセット値 (整数) で、*factor* は `findfactor()` で返される数値 (浮動小数点数) です。

**findmax**(*fragment*, *length*)

*fragment* を長さ *length* のサンプル (バイトではありません!) にスライスして、最大レベルをもつサンプルを探します。つまり、`rms(fragment[i*2:(i+length)*2])` が最大となるような *i* を返します。データは 2 バイトのサンプルでなければなりません。

このルーチンの実行に要する時間は `len(fragment)` に比例します。

**getsample**(*fragment*, *width*, *index*)

データから *index* のサンプルの値を返します。

**lin2lin**(*fragment*, *width*, *newwidth*)

サンプルサイズを 1、2、4 バイトフォーマットの間で変換します。

**lin2adpcm**(*fragment*, *width*, *state*)

サンプルを 4 ビットの Intel/DVI ADPCM フォーマットにエンコードします。ADPCM フォーマットは、(さまざまな) あるステップで区切られたサンプルと次のサンプルとの差である 4 ビットの数値を利用したコード化スキームです。Intel/DVI ADPCM アルゴリズムは国際 MIDI 協会に採用されているので、標準になると言ってもいいでしょう。

*state* はエンコードの情報を含んだタプルです。`(adpcmfrag, newstate)` のタプルを返します。*newstate* は次に `lin2adpcm()` を呼び出す時に渡さなければなりません。最初の呼び出しでは *state* として `None` を渡します。*adpcmfrag* は ADPCM でコード化された、バイト当たり 2 つの 4 ビット値です。

**lin2adpcm3**(*fragment*, *width*, *state*)

これはサンプル当たり 3 ビットのみを使う新しい ADPCM フォーマットです。これは Intel/DVI ADPCM フォーマットと互換性がなく、出力は提供されていません（作者の怠慢によるものです）。使うとがっかりします。

**lin2ulaw**(*fragment*, *width*)

オーディオデータのサンプルを u-LAW フォーマットにエンコードし、Python 文字列として返します。u-LAW はオーディオエンコードフォーマットで、8 ビットのためのサンプルで約 14 ビットのダイナミックレンジが得られます。他でも使われていますが、Sun のオーディオハードウェアで使われています。

**minmax**(*fragment*, *width*)

サウンドデータ内の全サンプルの最小値と最大値からなるタプルを返します。

**max**(*fragment*, *width*)

データ内の全サンプルの絶対値の最大値を返します。

**maxpp**(*fragment*, *width*)

サウンドデータの最大振幅の最大値を返します。

**mul**(*fragment*, *width*, *factor*)

元のデータ内の全サンプルに浮動小数点数 *factor* を掛けたデータを返します。オーバーフローした分は無視します。

**ratecv**(*fragment*, *width*, *nchannels*, *inrate*, *outrate*, *state* [, *weightA* [, *weightB* ]])

入力したデータのフレームレートを変換します。

*state* は変換の情報を含むタプルです。(newfragment, newstate) を返します。newstate は次に ratecv() を呼び出す時に渡さなければなりません。

最初の呼び出しでは *state* として None を渡します。

引数 *weightA* と *weightB* はシンプルなデジタルフィルタのためのパラメータで、デフォルトではそれぞれ 1、0 です。

**reverse**(*fragment*, *width*)

データ内のサンプルの順序を逆転したデータを返します。

**rms**(*fragment*, *width*)

データの標準偏差を返します。つまり、

$$\sqrt{\frac{\sum S_i^2}{n}}$$

これはオーディオ信号の大きさを示します。

**tomono**(*fragment*, *width*, *lfactor*, *rfactor*)

ステレオデータをモノラルデータに変換します。2 つのチャンネルを足してモノラルの信号にする前に、左チャンネルは *lfactor* 倍、右チャンネルは *rfactor* 倍されます。

**tostereo**(*fragment*, *width*, *lfactor*, *rfactor*)

モノラルのデータからステレオのデータを作ります。左チャンネルのサンプルはモノラルのサンプルを *lfactor* 倍、右チャンネルは *rfactor* 倍して、ステレオデータのサンプルを算出します。

**ulaw2lin**(*fragment*, *width*)

u-LAW フォーマットのサウンドデータを linear フォーマットのサウンドデータにエンコードします。u-LAW フォーマットは常に 8 ビットのサンプルを使うので、*width* は出力データのサンプルサイズを示します。

mul() や max() はモノラルとステレオの区別をつけずに、全サンプルを等しく扱います。これが問題なら、初めにステレオデータを 2 つのモノラルデータに分割して、あとで結合するといいいでしょう。どうしたらいいか、例を挙げます。

```
def mul_stereo(sample, width, lfactor, rfactor):
    lsample = audioop.tomono(sample, width, 1, 0)
    rsample = audioop.tomono(sample, width, 0, 1)
    lsample = audioop.mul(lsample, width, lfactor)
    rsample = audioop.mul(rsample, width, rfactor)
    lsample = audioop.tostereo(lsample, width, 1, 0)
    rsample = audioop.tostereo(rsample, width, 0, 1)
    return audioop.add(lsample, rsample, width)
```

もしネットワークパケットを構築して ADPCM フォーマットを使って、プロトコルの説明なしにした  
いなら（つまりパケットの損失を我慢できるなら）、データだけでなく state も送信すべきです。注意し  
てほしいのは、（コーダーによって返される）最後の state でなく、*initial* の state（`lin2adpcm()` に渡す  
引数）をコーダーに渡すことです。もしバイナリで state を保存するのに `struct.struct()` を使いた  
いなら、最初の要素（最初の値）を 16 ビットで、2 番目の要素（前のサンプルとの差）を 8 ビットでコード  
化します。

ここの ADPCM コーダーは他の ADPCM コーダーに対しては試していません。それぞれの標準のものとの  
間で操作できない場合は、私が仕様を誤解していることも十分あり得ます。

`find*()` ルーチンは、一見、滑稽に見えるかもしれませんが。これらは最初、エコーを取り消すためのも  
のでした。これを合理的に速く実行するには、出力サンプルのレベルの一番大きい部分を取り出し、その  
場所の入力サンプル内での位置をみつけ、入力サンプルから出力サンプル全体を減算します：

```
def echocancel(outputdata, inputdata):
    pos = audioop.findmax(outputdata, 800)      # 1/10 秒
    out_test = outputdata[pos*2:]
    in_test = inputdata[pos*2:]
    ipos, factor = audioop.findfit(in_test, out_test)
    # Optional (for better cancellation):
    # factor = audioop.findfactor(in_test[ipos*2:ipos*2+len(out_test)],
    #                             out_test)
    prefill = '\0'*(pos+ipos)*2
    postfill = '\0'*(len(inputdata)-len(prefill)-len(outputdata))
    outputdata = prefill + audioop.mul(outputdata, 2, -factor) + postfill
    return audioop.add(inputdata, outputdata, 2)
```

## 14.2 imageop — 生の画像データを操作する

`imageop` モジュールは画像に関する便利な演算がふくまれています。Python 文字列に保存されている 8 ま  
たは 32 ビットのピクセルから構成される画像を操作します。これは `gl.rectwrite()` と `imgfile` モ  
ジュールが使用しているものと同じフォーマットです。

モジュールは次の変数と関数を定義しています：

### exception error

この例外はピクセル当りの未知のビット数などのすべてのエラーで発生させられます。

**crop**(*image*, *psize*, *width*, *height*, *x0*, *y0*, *x1*, *y1*)

*image* の選択された部分を返します。 *image* は *width* × *height* の大きさで、 *psize* バイトのピクセルか  
ら構成されなければなりません。 *x0*、 *y0*、 *x1* および *y1* は `gl.rectread()` パラメータと同様です。  
すなわち、境界は新画像に含まれます。新しい境界は画像の内部である必要はありません。旧画像の  
外側になるピクセルは値をゼロに設定されます。 *x0* が *varx1* より大きければ、新画像は鏡像反転され  
ます。 *y* 軸についても同じことが適用されます。

**scale**( *image*, *psize*, *width*, *height*, *newwidth*, *newheight* )

*image* を大きさ *newwidth* × *newheight* に伸縮させて返します。補間はいわれません。ばかばかしいほど単純なピクセルの複製と間引きを行い伸縮させます。そのため、コンピュータで作った画像やディザ処理された画像は伸縮した後見た目が良くありません。

**tovideo**( *image*, *psize*, *width*, *height* )

垂直ローパスフィルタ処理を画像全体に行います。それぞれの目標ピクセルを垂直に並んだ二つの元ピクセルから計算することで行います。このルーチンの主な用途としては、画像がインターレース走査のビデオ装置に表示された場合に極端なちらつきを抑えるために用います。そのため、この名前があります。

**grey2mono**( *image*, *width*, *height*, *threshold* )

全ピクセルを二値化することによって、深さ 8 ビットのグレースケール画像を深さ 1 ビットの画像へ変換します。処理後の画像は隙間なく詰め込まれ、おそらく **mono2grey**( ) の引数としてしか使い道がないでしょう。

**dither2mono**( *image*, *width*, *height* )

(ばかばかしいほど単純な) ディザ処理アルゴリズムを用いて、8 ビットグレースケール画像を 1 ビットのモノクロ画像に変換します。

**mono2grey**( *image*, *width*, *height*, *p0*, *p1* )

1 ビットモノクロが象画像を 8 ビットのグレースケールまたはカラー画像に変換します。入力で値ゼロの全てのピクセルは出力では値 *p0* を取り、値 0 の入力ピクセルは出力では値 *p1* を取ります。白黒のモノクロ画像をグレースケールへ変換するためには、値 0 と 255 をそれぞれ渡してください。

**grey2grey4**( *image*, *width*, *height* )

ディザ処理を行わずに、8 ビットグレースケール画像を 4 ビットグレースケール画像へ変換します。

**grey2grey2**( *image*, *width*, *height* )

ディザ処理を行わずに、8 ビットグレースケール画像を 2 ビットグレースケール画像に変換します。

**dither2grey2**( *image*, *width*, *height* )

ディザ処理を行い、8 ビットグレースケール画像を 2 ビットグレースケール画像へ変換します。**dither2mono**( ) については、ディザ処理アルゴリズムは現在とても単純です。

**grey42grey**( *image*, *width*, *height* )

4 ビットグレースケール画像を 8 ビットグレースケール画像へ変換します。

**grey22grey**( *image*, *width*, *height* )

2 ビットグレースケール画像を 8 ビットグレースケール画像へ変換します。

## 14.3 aifc — AIFF および AIFC ファイルの読み書き

このモジュールは AIFF と AIFF-C ファイルの読み書きをサポートします。AIFF ( Audio Interchange File Format ) はデジタルオーディオサンプルをファイルに保存するためのフォーマットです。AIFF-C は AIFF の新しいバージョンで、オーディオデータの圧縮に対応しています。

注意： 操作のいくつかは IRIX 上でのみ動作します；そういう操作では IRIX でのみ利用できる `c1` モジュールをインポートしようとして、`ImportError` を発生します。

オーディオファイルには、オーディオデータについて記述したパラメータがたくさん含まれています。サンプリングレートあるいはフレームレートは、1 秒あたりのオーディオサンプル数です。チャンネル数は、モノラル、ステレオ、4 チャンネルかどうかを示します。フレームはそれぞれ、チャンネルごとに一つのサンプルからなります。サンプルサイズは、一つのサンプルの大きさをバイト数で示したものです。したがって、一つのフレームは  $nchannels * samplesize$  バイトからなり、1 秒間では  $nchannels * samplesize * framerate$  バ

イトで構成されます。

例えば、CD 品質のオーディオは 2 バイト (16 ビット) のサンプルサイズを持っていて、2 チャンネル (ステレオ) であり、44,100 フレーム / 秒のフレームレートを持っています。そのため、フレームサイズは 4 バイト (2\*2) で、1 秒間では 2\*2\*44100 バイト (176,400 バイト) になります。

aifc モジュールは以下の関数を定義しています：

`open(file[, mode])`

AIFF あるいは AIFF-C ファイルを開き、後述するメソッドを持つインスタンスを返します。引数 *file* はファイルを示す文字列か、ファイルオブジェクトのいずれかです。*mode* は、読み込み用を開くときには 'r' か 'rb' のどちらかで、書き込み用を開くときには 'w' か 'wb' のどちらかでなければなりません。もし省略されたら、*file.mode* が存在すればそれが使用され、なければ 'rb' が使われます。書き込み用にこのメソッドを使用するときには、これから全部でどれだけのサンプル数を書き込むのか分からなかったり、`writeframesraw()` と `setnframes()` を使わないなら、ファイルオブジェクトはシーク可能でなければなりません。

ファイルが `open()` によって読み込み用に開かれたときに返されるオブジェクトには、以下のメソッドがあります：

`getnchannels()`

オーディオチャンネル数 (モノラルなら 1、ステレオなら 2) を返します。

`getsampwidth()`

サンプルサイズをバイト数で返します。

`getframerate()`

サンプリングレート (1 秒あたりのオーディオフレーム数) を返します。

`getnframes()`

ファイルの中のオーディオフレーム数を返します。

`getcomptype()`

オーディオファイルで使用されている圧縮形式を示す 4 文字の文字列を返します。AIFF ファイルでは 'NONE' が返されます。

`getcompname()`

オーディオファイルの圧縮形式を人に判読可能な形にしたものを返します。AIFF ファイルでは 'not compressed' が返されます。

`getparams()`

以上の全ての値を上順に並べたタプルを返します。

`getmarkers()`

オーディオファイルのマーカーのリストを返します。一つのマーカーは三つの要素のタプルです。要素の 1 番目はマーク ID (整数)、2 番目はマーク位置のフレーム数をデータの始めから数えた値 (整数)、3 番目はマークの名称 (文字列) です。

`getmark(id)`

与えられた *id* のマークの要素を `getmarkers()` で述べたタプルで返します。

`readframes(nframes)`

オーディオファイルの次の *nframes* 個のフレームを読み込んで返します。返されるデータは、全チャンネルの圧縮されていないサンプルをフレームごとに文字列にしたものです。

`rewind()`

読み込むポインタをデータの始めに巻き戻します。次に `readframes()` を使用すると、データの始めから読み込みます。

`setpos(pos)`



指定したフレーム数の位置にポインタを設定します。

**tell()**

現在のポインタのフレーム位置を返します。

**close()**

AIFF ファイルを閉じます。このメソッドを呼び出したあとでは、オブジェクトはもう使用できません。

ファイルが `open()` によって書き込み用に開かれたときに返されるオブジェクトには、`readframes()` と `setpos()` を除く上述の全てのメソッドがあります。さらに以下のメソッドが定義されています。`get*()` メソッドは、対応する `set*()` を呼び出したあとでのみ呼び出し可能です。最初に `writeframes()` あるいは `writeframesraw()` を呼び出す前に、フレーム数を除く全てのパラメータが設定されていなければなりません。

**aiff()**

AIFF ファイルを作ります。デフォルトでは AIFF-C ファイルが作られますが、ファイル名が `' .aiff '` で終わっていれば AIFF ファイルが作られます。

**aifc()**

AIFF-C ファイルを作ります。デフォルトでは AIFF-C ファイルが作られますが、ファイル名が `' .aiff '` で終わっていれば AIFF ファイルが作られます。

**setnchannels(*nchannels*)**

オーディオファイルのチャンネル数を設定します。

**setsampwidth(*width*)**

オーディオのサンプルサイズをバイト数で設定します。

**setframerate(*rate*)**

サンプリングレートを 1 秒あたりのフレーム数で設定します。

**setnframes(*nframes*)**

オーディオファイルに書き込まれるフレーム数を設定します。もしこのパラメータが設定されていなかったり正しくなかったら、ファイルはシークに対応していなければなりません。

**setcomptype(*type, name*)**

圧縮形式を設定します。もし設定しなければ、オーディオデータは圧縮されません。AIFF ファイルは圧縮できません。変数 `name` は圧縮形式を人に判読可能にしたもので、変数 `type` は 4 文字の文字列でなければなりません。現在のところ、以下の圧縮形式がサポートされています：NONE, ULAW, ALAW, G722。

**setparams(*nchannels, sampwidth, framerate, com ptype, compname*)**

上の全パラメータを一度に設定します。引数はそれぞれのパラメータからなるタプルです。つまり、`setparams()` の引数として、`getparams()` を呼び出した結果を使うことができます。

**setmark(*id, pos, name*)**

指定した ID (1 以上)、位置、名称でマークを加えます。このメソッドは、`close()` の前ならいつでも呼び出すことができます。

**tell()**

出力ファイルの現在の書き込み位置を返します。`setmark()` との組み合わせで使うと便利です。

**writeframes(*data*)**

出力ファイルにデータを書き込みます。このメソッドは、オーディオファイルのパラメータを設定したあとでのみ呼び出し可能です。

**writeframesraw(*data*)**

オーディオファイルのヘッダ情報が更新されないことを除いて、`writeframes()` と同じです。

**close()**

AIFF ファイルを閉じます。ファイルのヘッダ情報は、オーディオデータの実際のサイズを反映して更新されます。このメソッドを呼び出したあとでは、オブジェクトはもう使用できません。

## 14.4 sunau — Sun AU ファイルの読み書き

sunau モジュールは、Sun AU サウンドフォーマットへの便利なインターフェースを提供します。このモジュールは、aifc モジュールや wave モジュールと互換性のあるインターフェースを備えています。

オーディオファイルはヘッダとそれに続くデータから構成されます。ヘッダのフィールドは以下の通りです：

フィールド	内容
magic word	4 バイト文字列 ‘.snd’。
header size	info を含むヘッダのサイズをバイト数で示したもの。
data size	データの物理サイズをバイト数で示したもの。
encoding	オーディオサンプルのエンコード形式。
sample rate	サンプリングレート。
# of channels	サンプルのチャンネル数。
info	オーディオファイルについての説明を ASCII 文字列で示したもの（null バイトで埋められます）。

info フィールド以外の全てのヘッダフィールドは 4 バイトの大きさです。ヘッダフィールドは big-endian でエンコードされた、計 32 ビットの符合なし整数です。

sunau モジュールは以下の関数を定義しています：

**open**(*file*, *mode*)

*file* が文字列ならその名前のファイルを開き、そうでないならファイルのようにシーク可能なオブジェクトとして扱います。*mode* は以下のうちのいずれかです。

‘r’ 読み込みのみのモード。

‘w’ 書き込みのみのモード。

読み込み / 書き込み両方のモードで開くことはできないことに注意して下さい。

‘r’ の *mode* は AU\_read オブジェクトを返し、‘w’ と ‘wb’ の *mode* は AU\_write オブジェクトを返します。

**openfp**(*file*, *mode*)

open() と同義。後方互換性のために残されています。

sunau モジュールは以下の例外を定義しています：

**exception Error**

Sun AU の仕様や実装に対する不適切な操作により何か実行不可能となった時に発生するエラー。

sunau モジュールは以下のデータアイテムを定義しています：

**AUDIO\_FILE\_MAGIC**

big-endian で保存された正規の Sun AU ファイルは全てこの整数で始まります。これは文字列 ‘.snd’ を整数に変換したものです。

**AUDIO\_FILE\_ENCODING\_MULAW\_8**

**AUDIO\_FILE\_ENCODING\_LINEAR\_8**

**AUDIO\_FILE\_ENCODING\_LINEAR\_16**

**AUDIO\_FILE\_ENCODING\_LINEAR\_24**

AUDIO\_FILE\_ENCODING\_LINEAR\_32

AUDIO\_FILE\_ENCODING\_ALAW\_8

AU ヘッダの encoding フィールドの値で、このモジュールでサポートしているものです。

AUDIO\_FILE\_ENCODING\_FLOAT

AUDIO\_FILE\_ENCODING\_DOUBLE

AUDIO\_FILE\_ENCODING\_ADPCM\_G721

AUDIO\_FILE\_ENCODING\_ADPCM\_G722

AUDIO\_FILE\_ENCODING\_ADPCM\_G723\_3

AUDIO\_FILE\_ENCODING\_ADPCM\_G723\_5

AU ヘッダの encoding フィールドの値のうち既知のものとして追加されているものですが、このモジュールではサポートされていません。

#### 14.4.1 AU\_read オブジェクト

上述の `open()` によって返される AU\_read オブジェクトには、以下のメソッドがあります：

`close()`

ストリームを閉じ、このオブジェクトのインスタンスを使用できなくします。(これはオブジェクトのガベージコレクション時に自動的に呼び出されます。)

`getnchannels()`

オーディオチャンネル数 (モノラルなら 1、ステレオなら 2) を返します。

`getsampwidth()`

サンプルサイズをバイト数で返します。

`getframerate()`

サンプリングレートを返します。

`getnframes()`

オーディオフレーム数を返します。

`getcomptype()`

圧縮形式を返します。'ULAW'、'ALAW'、'NONE' がサポートされている形式です。

`getcompname()`

`getcomptype()` を人に判読可能な形にしたものです。上述の形式に対して、それぞれ 'CCITT G.711 u-law'、'CCITT G.711 A-law'、'not compressed' がサポートされています。

`getparams()`

`get*()` メソッドが返すのと同じ (*nchannels*, *sampwidth*, *framerate*, *nframes*, *comptype*, *compname*) のタプルを返します。

`readframes(n)`

*n* 個のオーディオフレームの値を読み込んで、バイトごとに文字に変換した文字列を返します。データは linear 形式で返されます。もし元のデータが u-LAW 形式なら、変換されます。

`rewind()`

ファイルのポインタをオーディオストリームの先頭に戻します。

以下の 2 つのメソッドは共通の “位置” を定義しています。“位置” は他の関数とは独立して実装されています。

`setpos(pos)`

ファイルのポインタを指定した位置に設定します。`tell()` で返される値を *pos* として使用しなければなりません。

**tell()**

ファイルの現在のポインタ位置を返します。返される値はファイルの実際の位置に対して何も操作はしません。

以下の2つのメソッドは `aifc` モジュールとの互換性のために定義されていますが、何も面白いことはありません。

**getmarkers()**

`None` を返します。

**getmark(*id*)**

エラーを発生します。

## 14.4.2 AU\_write オブジェクト

上述の `open()` によって返される `Wave_write` オブジェクトには、以下のメソッドがあります：

**setnchannels(*n*)**

チャンネル数を設定します。

**setsampwidth(*n*)**

サンプルサイズを (バイト数で) 設定します。

**setframerate(*n*)**

フレームレートを設定します。

**setnframes(*n*)**

フレーム数を設定します。あとからフレームが書き込まれるとフレーム数は変更されます。

**setcomptype(*type, name*)**

圧縮形式とその記述を設定します。'NONE' と 'ULAW' だけが、出力時にサポートされている形式です。

**setparams(*tuple*)**

*tuple* は (*nchannels, sampwidth, framerate, nframes, comptype, compname*) で、それぞれ `set*()` のメソッドの値にふさわしいものでなければなりません。全ての変数を設定します。

**tell()**

ファイルの中の現在位置を返します。`AU_read.tell()` と `AU_read.setpos()` メソッドでお断りしたことがこのメソッドにも当てはまります。

**writeframesraw(*data*)**

*nframes* の修正なしにオーディオフレームを書き込みます。

**writeframes(*data*)**

オーディオフレームを書き込んで *nframes* を修正します。

**close()**

*nframes* が正しいか確認して、ファイルを閉じます。このメソッドはオブジェクトの削除時に呼び出されます。

`writeframes()` や `writeframesraw()` メソッドを呼び出したあとで、どんなパラメータを設定しようとしても不正となることに注意して下さい。

## 14.5 wave — WAV ファイルの読み書き

`wave` モジュールは、WAV サウンドフォーマットへの便利なインターフェイスを提供するモジュールです。

このモジュールは圧縮 / 展開をサポートしていませんが、モノラル / ステレオには対応しています。

wave モジュールは、以下の関数と例外を定義しています。

**open**(*file*[, *mode*])

*file* が文字列ならその名前のファイルを開き、そうでないならファイルのようにシーク可能なオブジェクトとして扱います。*mode* は以下のうちのいずれかです。

‘r’, ‘rb’ 読み込みのみのモード。

‘w’, ‘wb’ 書き込みのみのモード。

WAV ファイルに対して読み込み / 書き込み両方のモードで開くことはできないことに注意して下さい。‘r’ と ‘rb’ の *mode* は Wave\_read オブジェクトを返し、‘w’ と ‘wb’ の *mode* は Wave\_write オブジェクトを返します。*mode* が省略されていて、ファイルのようなオブジェクトが *file* として渡されると、*file.mode* が *mode* のデフォルト値として使われます (必要であれば、さらにフラグ ‘b’ が付け加えられます)。

**openfp**(*file*, *mode*)

open() と同義。後方互換性のために残されています。

**exception Error**

WAV の仕様を犯したり、実装の欠陥に遭遇して何か実行不可能となった時に発生するエラー。

### 14.5.1 Wave\_read オブジェクト

open() によって返される Wave\_read オブジェクトには、以下のメソッドがあります：

**close**()

ストリームを閉じ、このオブジェクトのインスタンスを使用できなくします。これはオブジェクトのガベージコレクション時に自動的に呼び出されます。

**getnchannels**()

オーディオチャンネル数 (モノラルなら 1、ステレオなら 2) を返します。

**getsampwidth**()

サンプルサイズをバイト数で返します。

**getframerate**()

サンプリングレートを返します。

**getnframes**()

オーディオフレーム数を返します。

**getcomptype**()

圧縮形式を返します (‘NONE’ だけがサポートされている形式です)。

**getcompname**()

getcomptype() を人に判読可能な形にしたものです。通常、‘NONE’ に対して ‘not compressed’ が返されます。

**getparams**()

get\*() メソッドが返すのと同じ (*nchannels*, *sampwidth*, *framerate*, *nframes*, *comptype*, *compname*) のタプルを返します。

**readframes**(*n*)

現在のポインタから *n* 個のオーディオフレームの値を読み込んで、バイトごとに文字に変換して文字列を返します。

**rewind()**

ファイルのポインタをオーディオストリームの先頭に戻します。

以下の2つのメソッドは `aifc` モジュールとの互換性のために定義されていますが、何も面白いことはありません。

**getmarkers()**

`None` を返します。

**getmark(*id*)**

エラーを発生します。

以下の2つのメソッドは共通の“位置”を定義しています。“位置”は他の関数とは独立して実装されています。

**setpos(*pos*)**

ファイルのポインタを指定した位置に設定します。

**tell()**

ファイルの現在のポインタ位置を返します。

## 14.5.2 Wave\_write オブジェクト

`open()` によって返される `Wave_write` オブジェクトには、以下のメソッドがあります：

**close()**

*nframes* が正しいか確認して、ファイルを閉じます。このメソッドはオブジェクトの削除時に呼び出されます。

**setnchannels(*n*)**

チャンネル数を設定します。

**setsampwidth(*n*)**

サンプルサイズを *n* バイトに設定します。

**setframerate(*n*)**

サンプリングレートを *n* に設定します。

**setnframes(*n*)**

フレーム数を *n* に設定します。あとからフレームが書き込まれるとフレーム数は変更されます。

**setcomptype(*type, name*)**

圧縮形式とその記述を設定します。

**setparams(*tuple*)**

*tuple* は (*nchannels, sampwidth, framerate, nframes, comptype, compname*) で、それぞれ `set*()` のメソッドの値にふさわしいものでなければなりません。全ての変数を設定します。

**tell()**

ファイルの中の現在位置を返します。`Wave_read.tell()` と `Wave_read.setpos()` メソッドでお断りしたことがこのメソッドにも当てはまります。

**writeframesraw(*data*)**

*nframes* の修正なしにオーディオフレームを書き込みます。

**writeframes(*data*)**

オーディオフレームを書き込んで *nframes* を修正します。

`writeframes()` や `writeframesraw()` メソッドを呼び出したあとで、どんなパラメータを設定しようとしても不正となることに注意して下さい。そうすると `wave.Error` を発生します。



## 14.6 chunk — IFF チャンクデータの読み込み

このモジュールは EA IFF 85 チャンクを使用しているファイルの読み込みのためのインターフェースを提供します。<sup>1</sup> このフォーマットは少なくとも、Audio Interchange File Format (AIFF/AIFF-C) と Real Media File Format (RMFF) で使われています。WAVE オーディオファイルフォーマットも厳密に対応しているので、このモジュールで読み込みできます。チャンクは以下の構造を持っています：

Offset 値	長さ	内容
0	4	チャンク ID
4	4	big-endian で示したチャンクのサイズで、ヘッダは含みません
8	$n$	バイトデータで、 $n$ はこれより先のフィールドのサイズ
$8 + n$	0 or 1	$n$ が奇数ならチャンクの整頓のために埋められるバイト

ID はチャンクの種類を識別する 4 バイトの文字列です。

サイズフィールド（big-endian でエンコードされた 32 ビット値）は、8 バイトのヘッダを含まないチャンクデータのサイズを示します。

普通、IFF タイプのファイルは 1 個かそれ以上のチャンクからなります。このモジュールで定義される Chunk クラスの使い方として提案しているのは、それぞれのチャンクの始めにインスタンスを作り、終わりに達するまでそのインスタンスから読み取り、その後で新しいインスタンスを作るということです。ファイルの終わりで新しいインスタンスを作ろうとすると、EOFError の例外が発生して失敗します。

`class Chunk (file[, align, bigendian, inclheader])`

チャンクを表現するクラス。引数 *file* はファイルのようなオブジェクトであることが想定されています。このクラスのインスタンスは特別にそのように認められています。必要とされるメソッドは `read()` だけです。もし `seek()` と `tell()` メソッドが呼び出されて例外を発生させなかったら、これらのメソッドも動作します。これらのメソッドが呼び出されて例外を発生させても、オブジェクトを変化させないようになっています。

省略可能な引数 *align* が `true` なら、チャンクデータが偶数個で 2 バイトごとに整頓されていると想定します。もし *align* が `false` なら、チャンクデータが奇数個になっていると想定します。デフォルト値は `true` です。

もし省略可能な引数 *bigendian* が `false` なら、チャンクサイズは little-endian であると想定します。この引数の設定は WAVE オーディオファイルで必要です。デフォルト値は `true` です。

もし省略可能な引数 *inclheader* が `true` なら、チャンクのヘッダから得られるサイズはヘッダのサイズを含んでいると想定します。デフォルト値は `false` です。

Chunk オブジェクトには以下のメソッドが定義されています：

`getname()`

チャンクの名前 (ID) を返します。これはチャンクの始めの 4 バイトです。

`getsize()`

チャンクのサイズを返します。

`close()`

オブジェクトを閉じて、チャンクの終わりまで飛びます。これは元のファイル自体は閉じません。

残りの以下のメソッドは、`close()` メソッドを呼び出した後に呼び出すと例外 `IOError` を発生します。

`isatty()`

`False` を返します。

<sup>1</sup>“EA IFF 85” Standard for Interchange Format Files, Jerry Morrison, Electronic Arts, January 1985.

`seek(pos[, whence])`

チャンクの現在位置を設定します。引数 *whence* は省略可能で、デフォルト値は 0 (ファイルの絶対位置) です; 他に 1 (現在位置から相対的にシークします) と 2 (ファイルの末尾から相対的にシークします) の値を取ります。何も値は返しません。もし元のファイルがシークに対応していなければ、前方へのシークのみが可能です。

`tell()`

チャンク内の現在位置を返します。

`read([size])`

チャンクから最大で *size* バイト (*size* バイトを読み込むまで、少なくともチャンクの最後に行き着くまで) 読み込みます。もし引数 *size* が負か省略されたら、チャンクの最後まで全てのデータを読み込みます。バイト値は文字列のオブジェクトとして返されます。チャンクの最後に行き着いたら、空文字列を返します。

`skip()`

チャンクの最後まで飛びます。さらにチャンクの `read()` を呼び出すと、`"` が返されます。もしチャンクの内容に興味がないなら、このメソッドを呼び出してファイルポインタを次のチャンクの始めに設定します。

## 14.7 colorsys — 色体系間の変換

`colorsys` モジュールは、計算機のディスプレイモニタで使われている RGB (Red Green Blue) 色空間で表された色と、他の 3 種類の色座標系: YIQ, HLS (Hue Lightness Saturation: 色相、彩度、飽和) および HSV (Hue Saturation Value: 色相、彩度、明度) との間の双方向の色値変換を定義します。これらの色空間における色座標系は全て浮動小数点数で表されます。YIQ 空間では、Y 軸は 0 から 1 ですが、I および Q 軸は正の値も負の値もとります。他の色空間では、各軸は全て 0 から 1 の値をとります。

色空間に関するより詳細な情報は <http://www.inforamp.net/~%7epoynton/ColorFAQ.html> にあります。

`colorsys` モジュールでは、以下の関数が定義されています:

`rgb_to_yiq(r, g, b)`

RGB から YIQ に変換します。

`yiq_to_rgb(y, i, q)`

YIQ から RGB に変換します。

`rgb_to_hls(r, g, b)`

RGB から HLS に変換します。

`hls_to_rgb(h, l, s)`

HLS から RGB に変換します。

`rgb_to_hsv(r, g, b)`

RGB から HSV に変換します。

`hsv_to_rgb(h, s, v)`

HSV から RGB に変換します。

サンプルコード:

```
>>> import colorsys
>>> colorsys.rgb_to_hsv(.3, .4, .2)
(0.25, 0.5, 0.4)
>>> colorsys.hsv_to_rgb(0.25, 0.5, 0.4)
(0.3, 0.4, 0.2)
```

## 14.8 rgbimg — “SGI RGB” ファイルを読み書きする

rgbimg モジュールは Python プログラムが (‘.rgb’ としても知られる) SGI imglib 画像ファイルにアクセスできるようにします。モジュールは完全にはほど遠いが、場合によっては寿分といえる機能を持っているためとにかく提供されています。現在、カラーマップファイルがサポートされていません。

注意: このモジュールはデフォルトでは 32 ビットプラットフォームでのみ構築されます。他のプラットフォームで適切な動作をすることは期待できません。

モジュールは次の変数と関数を定義しています:

### exception error

例えば未サポートのファイル形式などのようなすべてのエラーに対してこの例外が発生します。

### sizeofimage(*file*)

この関数はタプル (*x*, *y*) を返します。ここで、*x* と *y* は画素単位の画像の大きさです。4 バイト RGBA ピクセル、3 バイト RGB ピクセル、および、1 バイトグレースケールピクセルが現在サポートされています。

### longimagedata(*file*)

この関数は特定ファイルの画像を読み込んでデコードし、Python 文字列としてそれを返します。その文字列は 4 バイト RGB ピクセルです。左下のピクセルが文字列の先頭です。例えば、このフォーマットは `gl.glRectwrite()` へ渡すために適しています。

### longstoimage(*data*, *x*, *y*, *z*, *file*)

この関数は *data* の RGBA データを画像ファイル *file* へ書き込みます。*x* と *y* は画像の大きさを表します。セーブされた画像が 1 バイトのグレースケールの場合は、*z* は 1 です。セーブされた画像が 3 バイトの RGB データの場合は 3 です。セーブされた画像が 4 バイトの RGBA データの場合は 4 です。入力画像は常にピクセル当たり 4 バイト含んでいます。`gl.glRectread()` が返すフォーマットがあります。

### ttob(*flag*)

この関数はグローバルなフラグを設定します。そのフラグは画像のスキャン行が下から上に向かって読み込みや書き込みが行われるのか (フラグは 0 で、SGI GL と互換性があります)、あるいは上から下に向かって読み込みや書き込みがお壊れるのか (フラグは 1 で、X と互換性があります) を定義します。デフォルトは、0 です。

## 14.9 imghdr — 画像の形式を決定する

imghdr モジュールはファイルやバイトストリームに含まれる画像の形式を決定します。

imghdr モジュールは次の関数を定義しています:

### what(*filename*[, *h*])

*filename* という名前のファイル内の画像データをテストし、画像形式を表す文字列を返します。オプションの *h* が与えられた場合は、*filename* は無視され、テストするバイトストリームを含んでいると

*h* は仮定されます。

以下に `what()` からの戻り値とともにリストするように、次の画像形式が認識されます:

Value	Image format
'rgb'	SGI ImgLib Files
'gif'	GIF 87a and 89a Files
'pbm'	Portable Bitmap Files
'pgm'	Portable Graymap Files
'ppm'	Portable Pixmap Files
'tiff'	TIFF Files
'rast'	Sun Raster Files
'xbm'	X Bitmap Files
'jpeg'	JPEG data in JFIF format
'bmp'	BMP files
'png'	Portable Network Graphics

この変数に追加することで、あなたは `imghdr` が認識できるファイル形式のリストを拡張できます:

#### tests

個別のテストを行う関数のリスト。それぞれの関数は二つの引数をとります: バイトストリームとオープンされたファイルのようにふるまうオブジェクト。`what()` がバイトストリームとともに呼び出されたときは、ファイルのようにふるまうオブジェクトは `None` でしょう。

テストが成功した場合は、テスト関数は画像形式を表す文字列を返すべきです。あるいは、失敗した場合は `None` を返すべきです。

例:

```
>>> import imghdr
>>> imghdr.what('/tmp/bass.gif')
'gif'
```

## 14.10 sndhdr — サウンドファイルの識別

`sndhdr` モジュールには、ファイルに保存されたサウンドデータの形式を識別するのに便利な関数が定義されています。どんな形式のサウンドデータがファイルに保存されているのか識別可能な場合、これらの関数は (*type*, *sampling\_rate*, *channels*, *frames*, *bits\_per\_sample*) のタプルを返します。*type* はデータの形式を示す文字列で、`'aifc'`、`'aiff'`、`'au'`、`'hcom'`、`'sndr'`、`'sndt'`、`'voc'`、`'wav'`、`'8svx'`、`'sb'`、`'ub'`、`'ul'` のうちの一つです。*sampling\_rate* は実際のサンプリングレート値で、未知の場合や読み取ることが出来なかった場合は 0 です。同様に、*channels* はチャンネル数で、識別できない場合や読み取ることが出来なかった場合は 0 です。*frames* はフレーム数で、識別できない場合は -1 です。タプルの最後の要素 *bits\_per\_sample* はサンプルサイズを示すビット数ですが、A-LAW なら `'A'`、u-LAW なら `'U'` です。

**what**(*filename*)

`whathdr()` を使って、ファイル *filename* に保存されたサウンドデータの形式を識別します。識別可能なら上記のタプルを返し、識別できない場合は `None` を返します。

**whathdr**(*filename*)

ファイルのヘッダ情報をもとに、保存されたサウンドデータの形式を識別します。ファイル名は *filename*

で渡されます。識別可能なら上記のタプルを返し、識別できない場合は `None` を返します。

## 14.11 ossaudiodev — OSS 互換オーディオデバイスへのアクセス

このモジュールを使うと OSS (Open Sound System) オーディオインターフェースにアクセスできます。OSS はオープンソースあるいは商用の Unix で広く利用可能で、Linux (カーネル 2.4 まで) と FreeBSD で標準のオーディオインターフェースとなっています。

参考資料:

オープンサウンドシステムプログラマーズガイド

(<http://www.opensound.com/pguide/oss.pdf>)

OSS C API のための公式ドキュメント

このモジュールでは OSS デバイスドライバで提供される多くの定数を定義しています; 定数のリストについては Linux か FreeBSD の '`<sys/soundcard.h>`' を参照してください。

ossaudiodev には以下の変数と関数が定義されています:

### exception error

この例外は特定のエラーを発生します。引数は何が誤っているかを示す文字列です。

(もし ossaudiodev が `open()`、`write()`、`ioctl()` などのシステムコールからエラーを受け取ったら `IOError` を発生します。ossaudiodev モジュールによって直接検出されたエラーは `os-saudiodev.error` になります。)

### open([device, ]mode)

オーディオデバイスを開いて、OSS オーディオデバイスオブジェクトを返します。このオブジェクトは `read()`、`write()`、`fileno()` などのようにファイルのような多くのメソッドをサポートしています。(しかし伝統的な Unix の `read/write` の構文と OSS オーディオデバイスのものとのあいだには微妙な違いがあります)。また、オーディオ特有の多くのメソッドがあります; メソッドの完全なリストについては下記を参照してください。

呼び出しの文法が普通と異なることに注意してください: 最初の引数が省略可能で、2 番目が必須です。これは ossaudiodev にとって替わられた古い `linuxaudiodev` との互換性のためという歴史的な産物です。

`device` は使用するオーディオデバイスファイルネームです。もしこれが指定されないなら、このモジュールは使うデバイスとして最初に環境変数 `AUDIODEV` を参照します。みつからなければ、'`/dev/dsp`' を参照します。

`mode` は、読み込みのみ (再生) のアクセスの '`r`'、書き込みのみ (録音) のアクセスの '`w`'、それら両方の '`rw`' のうちのどれか 1 つです。多くのサウンドカードでは一つのプロセスで同時にレコーダかプレーヤしか持てないので、操作が必要なときだけデバイスを開くのが良い考えです。また、いくつかのサウンドカードは半二重 (half-duplex) です: これらでは読み込みあるいは書き込みで開けますが、同時に両方では開けません。

### openmixer([device])

ミキサーデバイスを開いて、OSS ミキサーデバイスオブジェクトを返します。`device` は使用するミキサーデバイスのファイルネームです。もしこれが指定されないなら、このモジュールは使うデバイスとして最初に環境変数 `AUDIODEV` を参照します。みつからなければ、'`/dev/mixer`' を参照します。

### 14.11.1 オーディオデバイスオブジェクト

デバイスの設定

デバイスを設定するために、3つの関数を正しい順序で呼び出さなければなりません。

1. `setfmt()` で出力フォーマットを設定し、
2. `channels()` でチャンネル数を設定し、
3. `speed()` でサンプリングレートを設定します。

`open()` で返されるオーディオデバイスオブジェクトには以下のメソッドがあります:

**close()**

このメソッドはデバイスを明示的に閉じます。他にデバイスを参照しているものがあって、オブジェクトをすぐに閉じることができない場合に便利です。閉じられたオブジェクトを再び使うことはできません。

**fileno()**

デバイスに関連付けられたファイルディスクリプタを返します。

**read(size)**

オーディオ入力からサンプル数 *size* を読み込んで、Python 文字列として返します。この関数は必要なデータが得られるまでブロックします。

**write(data)**

Python 文字列の *data* をオーディオデバイスに書き込んで、書き込まれたバイト数を返します。もしオーディオデバイスがブロックモードで開かれたら、文字列全体が書き込まれます。もしデバイスが非ブロックモードで開かれたら、書き込まれないデータがあるかもしれません—`writeall()` を参照してください。

**writeall(data)**

Python 文字列の *data* 全体をオーディオデバイスに書き込みます。もしデバイスがブロックモードで開かれたら `write()` と同様に働きます; 非ブロックモードでは、デバイスにデータを与える前にデバイスが利用可能になるまで待ちます。書き込まれたデータの量は与えたデータと常に同じ量なので、`None` を返します。

シンプルな I/O コントロール:

**nonblock()**

デバイスを非ブロックモードにします。いったん非ブロックモードにしたら、ブロックモードに戻すことはできません。

I/O コントロールに失敗したら `IOError` が発生します。

**getfmts()**

サウンドカードでサポートされるオーディオ出力フォーマットのビットマスクを返します。典型的な Linux システムでは以下のフォーマットがあります:

フォーマット	説明
AFMT_MU_LAW	対数エンコーディング。これは '/dev/audio' のデフォルトのフォーマットで、Sun の .au ファイル形式。
AFMT_A_LAW	対数エンコーディング
AFMT_IMA_ADPCM	Interactive Multimedia Association で定められた 4:1 の圧縮フォーマット。
AFMT_U8	符号なし 8 ビットオーディオ。
AFMT_S16_LE	符号なし 16 ビットオーディオでリトルエンディアン (Intel プロセッサで使われるフォーマット)。
AFMT_S16_BE	符号なし 16 ビットオーディオでビッグエンディアン (68k、PowerPC、Sparc で使われるフォーマット)。
AFMT_S8	符号つき 8 ビットオーディオ。
AFMT_U16_LE	符号つき 16 ビットリトルエンディアンオーディオ
AFMT_U16_BE	符号つき 16 ビットビッグエンディアンオーディオ



たいていのシステムではこれらのフォーマットのうちのいくつかだけをサポートしています。多くのデバイスでは AFMT\_U8 だけをサポートしています; 現在使われている最も一般的なフォーマットは AFMT\_S16\_LE です。

**setfmt**(*format*)

現在のオーディオフォーマットを *format* にします—*format* については `getfmts()` のリストを参照してください。また、現在のオーディオフォーマットを返すのにも使えます—これは“オーディオフォーマット”として AFMT\_QUERY を渡すことでできます。デバイスに設定されたオーディオフォーマットが返されますが、設定するよう要求したフォーマットではないかもしれません。

**channels**(*num\_channels*)

出力チャンネル数を *num\_channels* に設定します。値 1 はモノラル、2 はステレオを示します。いくつかのデバイスでは 2 つより多いチャンネルを持つものもありますし、ハイエンドなデバイスではモノラルをサポートしないものもあります。デバイスに設定されたチャンネル数を返します。

**speed**(*samplerate*)

サンプリングレートを 1 秒あたり *samplerate* に設定し、実際に設定されたレートを返します。たいていのサウンドデバイスでは任意のサンプリングレートをサポートしていません。一般的なレートは以下の通りです:

8000—デフォルトのレート 11025—声の録音 22050 44100—オーディオ CD 品質 (2 チャンネルで 16 ビット/サンプル) 96000—DVD 品質

**sync**()

サウンドデバイスがバッファ内の全てのバイトを再生するまで待つて、それから制御が戻ります。これはサウンドデバイスが閉じられるときにも起こります。OSS の文書では `sync()` を使うよりもデバイスを単純に閉じて、再び開くことを推奨しています。

**reset**()

再生あるいは録音を中止して、デバイスをコマンド待ちの状態にします。OSS の文書では `reset()` を呼び出した後にデバイスを閉じて、再び開くことを推奨しています。

**post**()

簡単な `sync()` のように使われます。 `post()` の I/O コントロールによってオーディオデバイスにオーディオ出力にポーズがある—つまり、短いサウンドエフェクトの後や、ユーザの入力待ちの前、あるいはディスク I/O の前であることを伝えます。

便利なメソッド

**setparameters**(*samplerate,num\_channels,format,emulate*)

サウンドデバイスを一つのメソッドで初期化します。 *samplerate*、*channels*、*format* は `speed()`、`channels()`、`setfmt()` で述べたものでなければなりません。もし *emulate* が True なら、もっともマッチするフォーマットを見つけようとしますが、そうでないなら、指定したフォーマットをデバイスがサポートしなければ `ValueError` を発生します。デフォルトではサポートしないフォーマットに対して `ValueError` を発生します。

**bufsize**()

ハードウェアのバッファサイズをサンプル数で返します。

**obufcount**()

ハードウェアバッファにある、再生されるサンプル数を返します。

**obuffree**()

ブロックすることなしにハードウェアのキューに書き込んで再生できるサンプル数を返します。

## 14.11.2 ミキサーデバイスオブジェクト

ファイルに似たインターフェース

`close()`

このメソッドは、開かれたミキサーデバイスファイルを閉じます。ファイルを閉じた後でミキサーを使おうとすると、`IOError` を発生します。

`fileno()`

開いたミキサーデバイスファイルのファイルハンドルナンバーを返します。

ミキサーインターフェース

`controls()`

このメソッドは、利用可能なミキサーコントロールを指定するビットマスクを返します（“コントロール” はミックス可能な特定の“チャンネル” で、たとえば `SOUND_MIXER_PCM` あるいは `SOUND_MIXER_SYNTH` です）。このビットマスクは全ての利用可能なミキサーチャンネルのサブセットを示します—定数 `SOUND_MIXER_*` はモジュールレベルで定義されています。例えば、もし現在のミキサーオブジェクトが PCM ミキサーをサポートしているか調べるには、以下の Python コードを実行します：

```
mixer=ossaudiodev.openmixer()
if mixer.channels() & (1 << ossaudiodev.SOUND_MIXER_PCM):
    # PCM is supported
<code>
```

たいていの目的では、`SOUND_MIXER_VOLUME` (マスターボリューム) と `SOUND_MIXER_PCM` チャンネルで十分です—しかし、サウンドチャンネルを選ぶときに、ミキサーを使用するコードはフレキシブルでなければなりません。例えば、Gravis Ultrasound では `SOUND_MIXER_VOLUME` は存在しません。

`stereocontrols()`

ステレオミキサーチャンネルを示すビットマスクを返します。もしビットがセットされていたら対応するチャンネルはステレオで、もしセットされていないならそのチャンネルはモノラルか、ミキサーでサポートされていないかどうかです（そのどちらかを調べるには `channels()` を組み合わせて使います）。

ビットマスクのデータを得る例としては、関数 `channels()` のコードの例を参照してください。

`recontrols()`

録音に使用できるミキサーコントロールを特定するビットマスクを返します。ビットマスクから読み取る例としては、`controls()` のコードの例を参照してください。

`get(channel)`

与えられたミキサーのチャンネルのボリュームを返します。返される値は 2 要素のタプル (`left_volume, right_volume`) です。ボリュームの値は 0 (無音) から 100 (最大) で示されます。もしチャンネルがモノラルでも 2 要素のタプルが返されますが、2 つの要素の値は同じになります。

もし未知のチャンネルが指定されたら、`error` が発生します。

`set(channel, (left, right))`

与えられたミキサーチャンネルのボリュームを (`left, right`) に設定します。`left` と `right` は整数で、0 (無音) から 100 (最大) の間でなければなりません。成功したら、新しいボリューム値が 2 要素のタプルで返されます。あるサウンドカードではミキサーの分解能の限界のため、指定したボリューム値と厳密には同じにならないかもしれません。

もし不正なミキサーのチャンネルが指定されたら、`IOError` が発生します; 引数のフォーマットが正しくないなら `TypeError` が発生し、指定したボリューム値が範囲外なら `error` が発生します。

`get_recsrc()`

このメソッドは、現在録音のソースとして使われているチャンネルを示すビットマスクを返します。

`set_recsrc(bitmask)`

録音のソースを指定するのにこの関数を使います。成功したら、新たな録音のソースを示すビットマスクを返します; 不正なソースが指定されたら `IOError` が発生します。現在の録音のソースとしてマイク入力を設定する方法は以下の通りです:

```
mixer.setrecsrc (1 << ossaudiodev.SOUND_MIXER_MIC)
```



## 暗号関連のサービス

この章で記述されているモジュールでは、暗号の本質に関わる様々なアルゴリズムを実装しています。これらは必要に応じてインストールすることで使えます。概要を以下に示します:

**hmac** Python で実装された、メッセージ認証のための鍵付きハッシュ化 (HMAC: Keyed-Hashing for Message Authentication Code).

**md5** RSA's MD5 message digest algorithm.

**sha** NIST のセキュアハッシュアルゴリズム、SHA。

**mpz** 多倍精度における算術演算を実現する GNU MP ライブラリへのインタフェース。

**rotor** エニグマ暗号機のような暗号化と復号化。

あなたがハードコアなサイバーパンクなら、さらに A.M. Kuchling の書いた暗号化モジュールに興味を持つかもしれません。このパッケージでは組み込みの DES および IDEA 暗号を追加し、PGP 暗号化されたファイルの読み込みや復号化を行うためのモジュールなどを提供します。これらのモジュールは Python と一緒に配布されず、別に入手できます。詳細は <http://www.amk.ca/python/code/crypto.html> を見てください。

### 15.1 hmac — メッセージ認証のための鍵付きハッシュ化

2.2 で追加された仕様です。

このモジュールでは RFC 2104 で記述されている HMAC アルゴリズムを実装しています。

`new(key[, msg[, digestmod]])`

新たな hmac オブジェクトを返します。msg が存在すれば、メソッド呼び出し `update(msg)` を行います。digestmod は HMAC オブジェクトが使うダイジェストモジュールです。標準では md5 モジュールになっています。

HMAC オブジェクトは以下のメソッドを持っています:

`update(msg)`

hmac オブジェクトを文字列 msg で更新します。繰り返し呼び出しを行うと、それらの引数を全て結合した引数で単一の呼び出しをした際と同じに等価になります: すなわち `m.update(a); m.update(b)` は `m.update(a + b)` と等価です。

`digest()`

これまで `update()` メソッドに渡された文字列のダイジェスト値を返します。個の値は 16 バイトの文字列 (md5 の場合) か、20 バイトの文字列 (sha の場合) で、NULL バイトを含む非 ASCII 文字が含まれることがあります。

`hexdigest()`

`digest()` と似ていますが、ダイジェスト値が md5 のときで長さ 32 文字 (sha のときで 40 文字) の 16 進数字のみを含む文字列で返されます。この値は電子メールやその他の非バイナリ環境で値をやり取りする際に使うことができます。

`copy()`

hmac オブジェクトの (“クローン” の) コピーを返します。このコピーは最初の部分文字列が共通になっている文字列のダイジェスト値を効率よく計算するために使うことができます。

## 15.2 md5 — MD5 メッセージダイジェストアルゴリズム

このモジュールは RSA 社の MD5 メッセージダイジェスト アルゴリズムへのインタフェースを実装しています。(Internet RFC 1321 も参照してください)。利用方法は極めて単純です。まず md5 オブジェクトを `new()` を使って生成します。後は `update()` メソッドを使って、生成されたオブジェクトに任意の文字列データを入力します。オブジェクトに入力された文字列データ全体の *digest* (“fingerprint” として知られる強力な 128-bit チェックサム) は `digest()` を使っていつでも調べることができます。

例えば、文字列 ‘Nobody inspects the spammish repetition’ のダイジェストを得るためには以下のようにします:

```
>>> import md5
>>> m = md5.new()
>>> m.update("Nobody inspects")
>>> m.update(" the spammish repetition")
>>> m.digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
```

もっと詰めて書くと以下ようになります:

```
>>> md5.new("Nobody inspects the spammish repetition").digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
```

以下の値はモジュールの中で定数として与えられており、`new()` で返される md5 オブジェクトの属性としても与えられます:

### `digest_size`

返されるダイジェスト値のバイト数で表した長さ。常に 16 です。

md5 クラスオブジェクトは以下のメソッドをサポートします:

### `new([arg])`

新たな md5 オブジェクトを返します。もし *arg* が存在するなら、`update(arg)` を呼び出します。

### `md5([arg])`

下位互換性のために、`new()` の別名として提供されています。

md5 オブジェクトは以下のメソッドをサポートします:

### `update(arg)`

文字列 *arg* を入力として md5 オブジェクトを更新します。このメソッドを繰り返して呼び出す操作は、それぞれの呼び出し時の引数 *arg* を結合したデータを引数として一回の呼び出す操作と同等になります: つまり、`m.update(a); m.update(b)` は `m.update(a+b)` と同等です。

### `digest()`

これまで `update()` で与えてきた文字列入力のダイジェストを返します。返り値は 16 バイトの文字列で、null バイトを含む非 ASCII 文字が入っているかもしれません。

### `hexdigest()`

`digest()` に似ていますが、ダイジェストは長さ 32 の文字列になり、16 進表記文字しか含みません。この文字列は電子メールやその他のバイナリを受け付けない環境でダイジェストを安全にやりとりするために使うことができます。



**copy()**

md5 オブジェクトのコピー (“クローン”) を返します。冒頭の部分文字列が共通な複数の文字列のダイジェストを効率よく計算する際に使うことができます。

参考資料:

sha モジュール (15.3 節):

Secure Hash Algorithm (SHA) を実装した類似のモジュール。SHA アルゴリズムはより安全なハッシュアルゴリズムだと考えられています。

## 15.3 sha — SHA-1 メッセージダイジェストアルゴリズム

このモジュールは、SHA-1 として知られている、NIST の セキュアハッシュアルゴリズムへのインターフェースを実装しています。SHA-1 はオリジナルの SHA ハッシュアルゴリズムを改善したバージョンです。md5 モジュールと同じように使用します。: sha オブジェクトを生成するために `new()` を使い、`update()` メソッドを使って、このオブジェクトに任意の文字列を入力し、それまでに入力した文字列全体の *digest* をいつでも調べることができます。SHA-1 のダイジェストは MD5 の 128 bit とは異なり、160 bit です。

**new([string])**

新たな sha オブジェクトを返します。もし *string* が存在するなら、`update(string)` を呼び出します。

以下の値はモジュールの中で定数として与えられており、`new()` で返される sha オブジェクトの属性としても与えられます:

**blocksize**

ハッシュ関数に入力されるブロックのサイズ。このサイズは常に 1 です。このサイズは、任意の文字列をハッシュできるようにするために使われます。

**digest\_size**

返されるダイジェスト値をバイト数で表した長さ。常に 20 です。

sha オブジェクトには md5 オブジェクトと同じメソッドがあります。

**update(arg)**

文字列 *arg* を入力として sha オブジェクトを更新します。このメソッドを繰り返し呼び出す (操作は、それぞれの呼び出し時の引数を結合したデータを引数として一回の呼び出す操作と同等になります。つまり、`m.update(a)`; `m.update(b)` は `m.update(a+b)` と同等です。

**digest()**

これまで `update()` メソッド で与えてきた文字列のダイジェストを返します。戻り値は 20 バイトの文字列で、null バイトを含む非 ASCII 文字が入っているかもしれません。

**hexdigest()**

`digits()` と似ていますが、ダイジェストは長さ 40 の文字列になり、16 進表記数字しか含みません。電子メールやその他のバイナリを受け付けない環境で安全に値をやりとりするために使うことができます。

**copy()**

sha オブジェクトのコピー (“クローン”) を返します。冒頭の部分文字列が共通な複数の文字列のダイジェストを効率よく計算する際に使うことができます。

参考資料:

セキュアハッシュスタンダード

(<http://csrc.nist.gov/publications/fips/fips180-1/fip180-1.txt>)

セキュアハッシュアルゴリズムは NIST のドキュメント FIPS PUB 180-1 で定義されています。セキュアハッシュスタンダード, 1995 年 4 月出版。

ブレインテキスト (少なくとも一つの図が省略されています) と <http://csrc.nist.gov/publications/fips/fips180-1/fip180-1.pdf> にある PDF でオンラインから入手できます。

暗号ツールキット (セキュアハッシュ)  
(<http://csrc.nist.gov/encryption/tkhash.html>)

NIST からはられているセキュアハッシュに関するさまざまな情報へのリンク

## 15.4 mpz — GNU 多倍長整数

リリース 2.2 以降で撤廃された仕様です。この節の末尾にある参考文献を読んで下さい。同様の機能を提供するパッケージについての情報があります。このモジュールは Python 2.3 で除去されます。

このモジュールはオプションです。モジュールを含むように設定して Python を構築した際にのみ利用可能であり、GNU MP ソフトウェアがインストールされていることが必要です。

このモジュールは GNU MP ライブラリの一部の機能へのインタフェースを実装しています。GNU MP ライブラリは多倍精度の整数および有理数に対する算術演算ルーチンを定義しています。整数だけのインタフェース (`mpz_*()`) が提供されています。特に注釈のない限り、GNU MP ドキュメントに記述されている内容が適用されます。

有理数のサポート は Python で実装されています。例えば、Python ソースコード配布中の ‘`Demos/classes/Rat.py`’ で提供されている `Rat` モジュールを参照してください。

一般的に、`mpz` 型の数値は他の標準の Python 数値と同様に使うことができます。例えば、`+`、`*`、などといった組み込み演算子や、`abs()`、`int()` ...、`divmod()`、`pow()` といった、標準の組み込み関数を使うことができます。注意してください: ライブラリには `mpz_xor()` がなく、私にとっても必要ないので、ビット単位の `xor` 演算は `and`、`invert`、`or` を組み合わせて実装されています。

`mpz` 型の数値は `mpz()` を呼び出して生成します (厳密な解説については下を参照してください)。`mpz` 型の数値は以下のように印字されます: `mpz(value)`

**`mpz(value)`**

新しい `mpz` 型数値を生成します。`value` は通常の整数、長整数、その他の `mpz` 型数値、そして文字列にすることもできます。文字列の場合、引数は 256 進数の桁からなるアレイとして解釈され、最下桁が前で、値は正の数になります。下で記述されている `binary()` を参照してください。

**`MPZType`**

`mpz()` およびこのモジュール中の他のほとんどの関数が返すオブジェクトの型です。

個のモジュールでは多くの 外部 関数が定義されています。`mpz` 型でない引数はまず `mpz` 型の値に変換され、関数は `mpz` 型の数値を返します。

**`powm(base, exponent, modulus)`**

`pow(base, exponent) % modulus` を返します。`exponent == 0` の場合、`mpz(1)` を返します。C ライブラリ関数とは対照的に、この関数は負の指数を扱うことができます。

**`gcd(op1, op2)`**

`op1` および `op2` の最大公約数を返します。

**`gcdext(a, b)`**

`a*s + b*t == g == gcd(a, b)` であるようなタプル (`g, s, t`) を返します。

**`sqrt(op)`**

`op` の平方根を返します。結果はゼロの向きに丸められます。

**`sqrtrem(op)`**

`root*root + remainder == op` となるようなタプル (`root, remainder`) を返します。

`divm(numerator, denominator, modulus)`

$q * \text{denominator} \% \text{modulus} == \text{numerator}$  となる値  $q$  を返します。この関数は `gcdext()` を使って Python で実装することもできます。

`mpz` 数値型は一つのメソッドを持ちます:

`binary()`

`mpz` 型の数値をバイナリ文字列に変換します。数値は 256 進数の桁からなるアレイに記憶されます。最小の桁が先頭になります。

`mpz` 型の数値はゼロまたはそれ以上の値をもたなければなりません。そうでない場合 `ValueError` が送出されます。

参考資料:

*General Multiprecision Python*

(<http://gmpy.sourceforge.net/>)

このプロジェクトでは多倍精度の算術演算を Python で行えるようにする新たな数値型を構築中です。彼らの最初の試みも GNU MP ライブラリに基づいています。

*mxNumber — Extended Numeric Types for Python*

(<http://www.egenix.com/files/python/mxNumber.html>)

もう一つの GNU MP ライブラリに対するラッパです。Windows に移植された GNU MP ライブラリを含みます。

## 15.5 rotor — エニグマ暗号機のような暗号化と復号化

リリース 2.3 以降で撤廃された仕様です。この暗号化アルゴリズムは安全ではありません。

このモジュールは Lance Ellinghouse によって寄与されたロータによる暗号化アルゴリズムを実装しています。この設計は第二次世界大戦中メッセージを暗号化するために用いられたエニグマ暗号機からきたものです。ロータは単純に文字の入れ替えです。例えば 'A' をロータの起点とすると、このロータは 'A' を 'L' に、'B' を 'Z' に、'C' を 'G' に、等に関連付けるでしょう。暗号化を行うためには、複数の異なるロータを選び、ロータの起点を既知の位置に設定します; それらの初期位置が暗号化の鍵となります。文字を暗号化するには、原文の文字を最初のロータによって入れ替え、次にその結果を二つ目のロータで入れ替えます。そして全てのロータで入れ替え終わるまでこれを続けます。結果として得られた文字が暗号文になります。次に最終段ロータの起点位置を一つだけずらし、'A' を 'B' にします; 最終段ロータの起点位置が完全に一周したら、最終段から 2 段目のロータを一つだけずらし、同じ手続きを再帰的に繰り返します。別の言い方をすれば、一つの文字を暗号化した後は、ロータを車の距離計と同じ要領で一つ進めるということです。復号化の作業も同様です。ただし、置き換えを反転し、操作を逆に行います。

個のモジュールで利用可能な関数は以下のとおりです:

`newrotor(key[, numrotors])`

ロータオブジェクトを返します `key` はロータオブジェクトの暗号化キー文字列です; ヌル文字以外の任意のバイナリデータを含むことができます。暗号化キーはロータの置き換え順列をランダムに生成し、初期位置を決めるために使われます。 `numrotors` は返されるロータオブジェクト中におけるロータ置き換えの数です。省略される場合、標準の値として 6 が使われます。

ロータオブジェクトは以下のメソッドを持っています:

`setkey(key)`

ロータの暗号化キーを `key` にします、暗号化キーにヌル文字を入れてはいけません。

`encrypt(plaintext)`

ロータオブジェクトを初期状態に再初期化し、`plaintext` を暗号化して、暗号文を含む文字列を返しま

す。暗号文は常にもとの平文と同じ長さになります。

`encryptmore(plaintext)`

ロータオブジェクトを再初期化せずに *plaintext* を暗号化し、暗号文を含む文字列を返します。

`decrypt(ciphertext)`

ロータオブジェクトを初期状態に再初期化し、*ciphertext* を復号化して、平文を含む文字列を返します。平文は常に暗号文と同じ長さになります。

`decryptmore(ciphertext)`

ロータオブジェクトを再初期化せずに *ciphertext* を復号化し、平文を含む文字列を返します。

利用法の例:

```
>>> import rotor
>>> rt = rotor.newrotor('key', 12)
>>> rt.encrypt('bar')
'\xab4\xfb'
>>> rt.encryptmore('bar')
'\xef\xfd$'
>>> rt.encrypt('bar')
'\xab4\xfb'
>>> rt.decrypt('\xab4\xfb')
'bar'
>>> rt.decryptmore('\xef\xfd$')
'bar'
>>> rt.decrypt('\xef\xfd$')
'l(\xcd'
>>> del rt
```

このモジュールのコードはオリジナルのエニグマ暗号機の厳密なシミュレーションではありません; これはオリジナルのロータ暗号化手続きを異なる方法で実装しています。オリジナルのエニグマとの最も重要な違いは、エニグマでは異なるロータは実際には 5 から 6 個しか存在せず、一文字あたり 2 回ずつロータを適用していたことです; 暗号化キーはロータを暗号機に配置する順番でした。Python の `rotor` モジュールでは、与えられたキーはまず乱数生成器を初期化するために使います; 次にロータの順列およびその初期位置がランダムに決定されます。オリジナルの暗号機ではアルファベット文字だけを暗号化していたのに対し、このモジュールは 8 ビットの任意のバイナリデータを扱うことができ、またバイナリ出力を生成します。このモジュールはまた、任意の数のロータを扱うことができます。

オリジナルのエニグマ暗号は 1944 年に解読されました。ここで実装されているバージョンを解読するのは、おそらくそれよりかなり困難ですが、非常に高い技術を持ち、確信犯の攻撃者が暗号を解読するのは不可能ではありません。従って、NSA からあなたのファイルを守りたいと思うなら、このロータ暗号は全く安全とはいえませんが、行きずりの来訪者にあなたのファイルを覗き見る気力を無くさせるためにはおそらくちょうどよいでしょう。そして UNIX の `crypt` コマンドを使うよりもやや安全かもしれません。

# Tkを用いたグラフィカルユーザインターフェイス

Tk/Tcl は長らく Python の欠くことのできない一部となっていて、頑健でプラットフォームに依存しないウィンドウ・ツールキットです。Python プログラマが Tk/Tcl を利用するには、Tkinter モジュールとその拡張である Tix モジュールを使います。

Tkinter モジュールは Tcl/Tk の上に作られた軽量なオブジェクト指向レイヤです。Tkinter を使うために、Tcl コードを書く必要はありません。しかし、Tk ドキュメントと、時々 Tcl ドキュメントを調べる必要があります。Tkinter は Python クラスとして Tk ウィジェットを実装する一連のラッパーです。それに加えて、内部モジュール `_tkinter` は Python と Tcl に相互作用を許すスレッドセーフ・メカニズムを提供します。

Tk は Python にとってただ一つの GUI という訳ではありませんが、もっともよく使われているものです。Python 用の他の GUI ツールキットに関するこれ以上の情報については、16.6 章 “他のユーザインターフェイスモジュールとパッケージ” を参照してください。

<b>Tkinter</b>	グラフィカルユーザインターフェイスのための Tcl/Tk へのインターフェイス
<b>Tix</b>	Tkinter 用の Tk 拡張ウィジェット
<b>ScrolledText</b>	垂直スクロールバーを持つテキストウィジェット。
<b>turtle</b>	タートルグラフィックスのための環境。

## 16.1 Tkinter — Tcl/Tk への Python インターフェイス

Tkinter モジュール (“Tk interface”) は、Tk GUI ツールキットに対する標準の Python インターフェイスです。Windows と Macintosh システム上と同じく、ほとんどの UNIX プラットホームで Tk と Tkinter は利用できます。(Tk 自体は Python の一部ではありません。ActiveState によって保守されています。)

参考資料:

*Python Tkinter Resources*

(<http://www.python.org/topics/tkinter/>)

Python Tkinter Topic Guide は、Python から Tk の使うことについての情報と他の Tk に関する情報源へのリンクを非常にたくさん提供しています。

*An Introduction to Tkinter*

(<http://www.pythonware.com/library/an-introduction-to-tkinter.htm>)

Fredrik Lundh のオンライン・リファレンス資料

*Tkinter reference: a GUI for Python*

(<http://www.nmt.edu/tcc/help/pubs/lang.html>)

オンライン・リファレンス資料

*Tkinter for JPython*

(<http://jtkinter.sourceforge.net>)

Tkinter に対する Jython インターフェイス

*Python and Tkinter Programming*

(<http://www.amazon.com/exec/obidos/ASIN/1884777813>)

John Grayson による解説書 (ISBN 1-884777-81-3)。

### 16.1.1 Tkinter モジュール

ほとんどの場合、本当に必要となるのは Tkinter モジュールですが、それに加えて、たくさんの追加モジュールが利用できます。Tk インターフェイスは `_tkinter` という名前のバイナリモジュール内に存在します。このモジュールには Tk への低レベルのインターフェイスが含まれており、アプリケーションプログラマは決して直接使うべきではありません。たいてい共有ライブラリ (または DLL) ですが、時には Python インタプリタに静的にリンクされることもあります。

Tk インターフェイス・モジュールに加えて、Tkinter には多くの Python モジュールが含まれています。最も重要なモジュールが二つあり、Tkinter モジュール自身と `Tkconstants` と呼ばれるモジュールです。前者は自動的に後者をインポートします。従って、Tkinter を使うためには、一つのモジュールをインポートするだけで良いのです:

```
import Tkinter
```

あるいは、より多くの場合には:

```
from Tkinter import *
```

**class Tk**(*screenName=None, baseName=None, className='Tk'*)

Tk クラスは引数なしでインスタンス化されます。これは Tk のトップレベル・ウィジェットを作り出し、多くの場合にアプリケーションのメインウィンドウとなります。それぞれのインスタンスは、それ自身に対応する Tcl インタプリタを持っています。

Tk サポートを提供する他のモジュールには次のものが含まれています:

**ScrolledText** 垂直スクロールバー付きのテキストウィジェット。

**tkColorChooser** ユーザが色を選択するためのダイアログ。

**tkCommonDialog** ここにリストされている他のモジュール内で定義されるダイアログのためのベースクラス。

**tkFileDialog** ユーザがオープンまたはセーブするためのファイルを指定できるようにするコモンダイアログ。

**tkFont** フォントの扱いを補助するためのユーティリティ。

**tkMessageBox** 標準的な Tk ダイアログボックスへのアクセス。

**tkSimpleDialog** 基本的なダイアログと便利な機能。

**Tkdnd** Tkinter のためのドラッグアンドドロップのサポート。これは実験的なもので、Tk DND に置き換えられると使うことは推奨されなくなるはずです。

**turtle** Tk ウィンドウでのタートルグラフィックス。



### 16.1.2 Tkinter Life Preserver

この節は Tk または Tkinter のどちらかのすべてを網羅したチュートリアルを意図したものではありません。むしろ、システムについて学びはじめるための説明を提供する間に合わせのようなものです。

謝辞:

- Tkinter は Steen Lumholt と Guido van Rossum によって作成されました。
- Tk は John Ousterhout によって Berkeley に在籍中に作成されました。
- この Life Preserver は Virginia 大学の Matt Conway 他によって書かれました。
- html レンダリングおよび多くの編集は、Ken Manheimer によって FrameMaker バージョンから生成されました。
- Fredrik Lundh は現行の Tk 4.2 に合わせるためにクラスインターフェイスについての詳しい説明を書き、改訂しました。
- Mike Clarkson はドキュメンテーションを  $\text{\LaTeX}$  へ変換し、リファレンスマニュアルのユーザインターフェイスの章をコンパイルしました。

#### この節の使い方

この節は二つの部分からできています: 最初の半分は (大雑把に) 背景となる事柄を対象としています。二番目の半分はすぐ手に届くリファレンスとしてキーボードのところへ持って行くことができます。

“blah するにはどうしたら良いのだろうか” という形の問いに答えるとき、Tk そのもので “blah” する方法を調べてから、対応する Tkinter 呼び出しへ変換するのがもっとも良いということがよくあります。Python プログラマが Tk ドキュメンテーションを見ると、たいてい正しい Python コマンドの見当をつけることができます。これが意味するのは、Tkinter を使うには Tk についてほんの少しだけ知っていなければならないということです。このドキュメントはその役割を果たすことができていません。だから、我々ができるのは、既存の最も良いドキュメンテーションをあなたに示すことです。ここにはいくつかのヒントがあります:

- 著者たちは Tk の man ページのコピーを手に入れることを強く勧めています。特に、mann ディレクトリ内の man ページはもっとも役に立ちます。man3 man ページは Tk ライブラリに対する C インターフェイスについて説明しており、従って、スクリプト書きにとって特に役に立つ訳ではありません。
- Addison-Wesley は John Ousterhout が書いた *Tcl and the Tk Toolkit* (ISBN 0-201-63337-X) という名前の本を出版しており、初心者にとっては Tcl と Tk への良い入門書です。この本は網羅的なものではなく、細部の多くで man ページに沿ったものです。
- ‘Tkinter.py’ を見るのはほとんどの場合に最後の手段ですが、他に何も意味をなさないときには行ってみると良い場所であることもあるでしょう。

#### 参考資料:

ActiveState Tcl ホームページ

(<http://tcl.activestate.com/>)

Tk/Tcl の開発の大部分は ActiveState で行われています。

*Tcl and the Tk Toolkit*

(<http://www.amazon.com/exec/obidos/ASIN/020163337X>)

Tcl の発明者である John Ousterhout が書いた本。

百科事典のような Brent Welch の本。

## 簡単な Hello World プログラム

```
from Tkinter import *

class Application(Frame):
    def say_hi(self):
        print "hi there, everyone!"

    def createWidgets(self):
        self.QUIT = Button(self)
        self.QUIT["text"] = "QUIT"
        self.QUIT["fg"] = "red"
        self.QUIT["command"] = self.quit

        self.QUIT.pack({"side": "left"})

        self.hi_there = Button(self)
        self.hi_there["text"] = "Hello",
        self.hi_there["command"] = self.say_hi

        self.hi_there.pack({"side": "left"})

    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        self.createWidgets()

app = Application()
app.mainloop()
```

### 16.1.3 Tcl/Tk に (非常に) さっと目を通す

クラス階層は複雑に見えますが、実際の実務ではアプリケーションプログラマはほとんど常に階層のもっとも底辺にあるクラスを参照します。

注釈:

- これらのクラスは一つの名前空間に関数をまとめる目的で提供されます。それらは独立してインスタンス化されることを意図しています。
- Tk クラスはアプリケーション内で一度だけインスタンス化されるように意図されています。アプリケーションプログラマが明示的にインスタンス化する必要はなく、他のクラスがインスタンス化されたときにすぐシステムが作成します。
- Widget クラスはインスタンス化されることを意図しておらず、“本物の”ウィジェットを作るためにサブクラス化することだけを意図しています (C++では、これは‘抽象クラス’と呼ばれています)。

このリファレンス資料を使うために、Tk の短い進行を読む方法と Tk コマンドのいろいろな部分を識別する方法を知る必要があるということはまああるでしょう。(下に説明してあるものと Tkinter における同等なものについては、節 16.1.4を参照してください。)

Tk スクリプトは Tcl プログラムです。すべての Tcl プログラムのように、Tk スクリプトは空白で区切られたトークンを並べたものです。Tk ウィジェットはウィジェットのクラス、その設定を行うオプションと、それに役立つことをさせるアクションです。

Tk におけるウィジェットを作るには、コマンドは常に次のような形式です：

```
classCommand newPathname options
```

**classCommand** どの種類のウィジェットを作るかを表します (ボタン、ラベル、メニュー、...)

**newPathname** このウィジェットの新しい名前です。Tk のすべての名前はただ一つしか存在することが許されません。これを必ず実行するのを助けるために、ちょうどファイルシステム内のファイルのように、Tk のウィジェットにはパス名のある名前が付けられます。トップレベルウィジェットルートは、 (ピリオド) と呼ばれていて、子はさらにピリオドで区切られています。例として、`.myApp.controlPanel.okButton` はウィジェットの名前です。

**options** ウィジェットの見た目や場合によっては振る舞いを設定します。オプションはフラグと値のリスト形式で渡されます。unix シェルコマンドのフラグのように、フラグは ‘-’ の後に続きます。また、値は一語以上ある場合には引用符内に置かれます。

例として：

```
button .fred -fg red -text "hi there"
  ^         ^
  |         |
class      new
command   widget  (-opt val -opt val ...)
```

一旦作られたら、ウィジェットへのパス名は新しいコマンドとなります。この新しい *widget command* は、*action* を実行する新しいウィジェットを得るためのプログラムのハンドルです。C ではこれを `someAction(fred, someOptions)` と表し、C++ では `fred.someAction(someOptions)` と表すでしょう。そして、Tk では：

```
.fred someAction someOptions
```

とします。オブジェクト名 `.fred` がドットから始まっていることに注意してください。

あなたが期待するように、*someAction* に対する正当な値はウィジェットのクラスに依存します： `fred` がボタンなら `.fred disable` は機能しますが (`fred` はグレーになります)、`fred` がラベルなら機能しません (Tk ではラベルの無効化をサポートしていません)。

*someOptions* の正当な値はアクション依存です。 `disable` のようないくつかのアクションは引数を必要としませんが、テキストエントリボックスの `delete` コマンドのような他のものは、削除するテキストの範囲を指定するための引数を必要とします。

#### 16.1.4 基本的な Tk を Tkinter ヘマッピングする

Tk のクラスコマンドは、Tkinter のクラスコンストラクタに対応します。

```
button .fred          =====> fred = Button()
```

オブジェクトの親は作成時に与えられた新しい名前の中に暗に示されます。Tkinter では、親は明示的に指定されます。

```
button .panel.fred          =====> fred = Button(panel)
```

Tk の設定オプションは値に続くハイフン付きのタグのリストに入れて与えられます。Tkinter では、オプションはインスタンス・コンストラクタのキーワード引数と `configure` 呼び出しに対するキーワード引数として指定されるか、または既に作成されたインスタンスに対する辞書形式でのインスタンス・インデックスとして指定されます。節 16.1.6 を参照してください。

```
button .fred -fg red          =====> fred = Button(panel, fg = "red")
.fred configure -fg red       =====> fred["fg"] = red
OR ==> fred.config(fg = "red")
```

Tk では、ウィジェット上でアクションを実行するために、コマンドとしてウィジェット名を用います。そして、場合によっては引数 (オプション) を付けて、それにアクション名を続けます。Tkinter では、ウィジェットのアクションを呼び出すために、クラスインスタンスに対してメソッドを呼び出します。与えられたウィジェットが実行できるアクション (メソッド) は、Tkinter.py モジュールの中にリストされています。

```
.fred invoke                  =====> fred.invoke()
```

ウィジェットを packer (ジオメトリマネージャ) に渡すためには、オプションの引数付きで `pack` を呼び出します。Tkinter では、Pack クラスがこの機能をすべて持っており、`pack` コマンドのいろいろな形式がメソッドとして実装されています。Tkinter の全ウィジェットは Packer からサブクラス化されており、従って、すべての `pack` するメソッドを継承しています。Form ジオメトリマネージャに関する追加情報については、Tix モジュールのドキュメンテーションを参照してください。

```
pack .fred -side left         =====> fred.pack(side = "left")
```

### 16.1.5 Tk と Tkinter はどのように関係なのか

注意: これはグラフィック・イメージから導き出されました。このドキュメントの続きのバージョンで、そのイメージはより直接的に用いられるでしょう。

上から下に:

あなたのアプリはここです (Python) Python アプリケーションは Tkinter を呼び出します。

**Tkinter (Python モジュール)** この呼び出し (例えば、ボタンウィジェットの作成) は、Tkinter モジュール内で実装されています。それは Python で書かれています。この Python 関数はそのコマンドと引数を解析し、Python スクリプトの代わりにまるで Tk スクリプトから来たように見える形式に変換します。

**tkinter (C)** これらのコマンドと引数は *tkinter* 内の C 関数へ渡されます - 小文字であることに注意 - 拡張モジュール。

**Tk Widgets (C and Tcl)** Tk ライブラリを構成する C 関数を含む他の C モジュールを、この C 関数は呼び出すことができます。Tk は C といくらかの Tcl で実装されています。Tk ウィジェットの Tcl 部分は、ウィジェットへあるデフォルトの振る舞いをバインドするために使われます。そして、Python の

Tkinter モジュールがインポートされた時点で一度だけ実行されます (この段階をユーザが見たりすることはありません)。

**Tk (C)** Tk ウィジェットの Tk 部分は最終的なマッピングを実装します...

**Xlib (C)** スクリーン上にグラフィックスを描くための Xlib ライブラリ。

### 16.1.6 便利なりファレンス

#### オプションの設定

オプションは色やウィジェットの境界線幅のようなものを制御します。オプションを三通りの方法で設定することができます:

オブジェクト作成時にキーワード引数を使用 :

```
fred = Button(self, fg = "red", bg = "blue")
```

オブジェクト作成後にオプション名を辞書インデックスのように扱う :

```
fred["fg"] = "red"
fred["bg"] = "blue"
```

オブジェクト生成の後で複数の属性を更新するための `config()` メソッドを使う :

```
fred.config(fg = "red", bg = "blue")
```

あるオプションとその振る舞いの完全な説明には、問題のウィジェットの Tk の man ページを参照してください。

man ページには各ウィジェットの "STANDARD OPTIONS(標準オプション)" と "WIDGET SPECIFIC OPTIONS(ウィジェット固有のオプション)" が記載されていることに注意してください。前者は多くのウィジェットの共通のオプションのリストで、後者は特定のウィジェットに特有のオプションです。Standard Options は *options(3)* man ページに記述されます。

このドキュメントでは標準とウィジェット特有のオプションを区別しません。あるオプションはある種のウィジェットに適用できません。任意のウィジェットが特定のオプションに応答するかどうかは、ウィジェットのクラスに依存します。ボタンは `command` オプションを持っていますが、ラベルは持っていません。

任意のウィジェットによってサポートされるオプションは、ウィジェットの man ページに記載されています。または、`config()` メソッドを引数なしで呼び出すか、ウィジェットの `keys()` メソッドを呼び出すことで実行時に調べることができます。これらの呼び出しの戻り値は、キーがオプションの名前の文字列 (例えば、`'relief'`) で、値が 5-タプルである辞書です。

`bg` のようないくつかのオプションは長い名前を持つ共通オプションに対する同義語です (`bg` は "background" を短縮したものです)。`config()` メソッドに省略オプション名を渡すと、5-タプルではなく 2-タプルが返されます。返された 2-タプルには、同義語の名前と "本物の" オプション (例えば、(`'bg'`, `'background'`)) が含まれています。

インデックス	意味	例
0	オプション名	'relief'
1	データベース検索のためのオプション名	'relief'
2	データベース検索のためのオプションクラス	'Relief'
3	デフォルト値	'raised'
4	現在の値	'groove'

例:

```
>>> print fred.config()
{'relief' : ('relief', 'relief', 'Relief', 'raised', 'groove')}
```

もちろん、表示された辞書はすべての利用可能なオプションと値を含んでいます。これは例としての意味しかありません。

## Packer

packer は Tk のジオメトリマネジメント・メカニズムの一つです。 *the Packer class interface* も参照してください。

ジオメトリマネージャは、コンテナ - ウィジェットが共有するマスター (上位のウィジェット) - 内部のウィジェットの相対的な位置を指定するために用いられます。もっと扱いにくい *placer* (packer より一般的に使われておらず、ここで取り上げません) とは対照的に、packer は定性的な関係の指定を受け取ります - 上、~ の左、引き延ばしなど - そして、あなたのために厳密な座標位置が決まるようにすべてを調整します。

どんなマスターウィジェットでもその大きさは内部の "スレイブウィジェット" の大きさによって決まります。スレイブウィジェットが pack されるマスター内部で表示される場所を制御するために packer が用いられます。だいたいあなたが望んだ通りのレイアウトになるように、ウィジェットをフレーム内部へ、さらに、フレームは別のフレーム内部へ pack することがきます。さらに、一度 pack されると、追加の設定変更に合わせて並べ方は動的に調整されます。

ジオメトリマネージャによってウィジェットのジオメトリが確定されるまで、ウィジェットが表示されないことに注意してください。ジオメトリを確定するのを忘れるというのは初めのころによくやってしまう間違いで、そういうときにウィジェットが作られたのに何も表示されず驚くことになります。ウィジェットはジオメトリ確定された (例えば packer の pack() メソッドを適用された) 後のみ表示されます。

pack() メソッドを呼び出す時に、ウィジェットがそのコンテナ内部で表示される場所とメイン・アプリケーション・ウィンドウのサイズが変更されたときにどうするかを制御するためのキーワードオプション/値の組を指定することができます。いくつか例を示します:

```
fred.pack()                # デフォルトでは、side = "top"
fred.pack(side = "left")
fred.pack(expand = 1)
```

## Packer Options

packer とその取り得るオプションについてのより多くの情報を得るには、man ページと John Ousterhout の本の 183 ページを参照してください。

**anchor** アンカー型。 packer が区画内に各スレイブを配置する位置を示します。



**expand** ブール値、0 または 1。

**fill** 指定できる値: 'x'、'y'、'both'、'none'。

**ipadx**と**ipady** 距離 - スレイブウィジェットのそれぞれの面の内部パッドを指定します。

**padx**と**pady** 距離 - スレイブウィジェットのそれぞれの面の外部パッドを指定します。

**side** 指定できる値は: 'left'、'right'、'top'、'bottom'。

### ウィジェット変数を結びつける

(テキストエントリ・ウィジェットのような)いくつかのウィジェットの現在の値の設定は特別なオプションを使うことでアプリケーション変数に直接関連付けられています。これらのオプションとは `variable`、`textvariable`、`onvalue`、`offvalue` および `value` です。この関連付けは双方向に機能します: 変数が何らかの理由で変化した場合、それが関連付けられているウィジェットは新しい値を反映するように更新される。

不幸なことに、Tkinter の現在の実装では、任意の Python 変数をウィジェットに `variable` あるいは `textvariable` オプションを通して渡すことができません。これが機能するような変数のただ一つの種類は、Tkinter で定義されている `Variable` と呼ばれるクラスからサブクラス化された変数です。

既に定義された多くの便利な `Variable` のサブクラスがあります: `StringVar`、`IntVar`、`DoubleVar` および `BooleanVar`。このような変数の現在の値を読むためには、その `get()` メソッドを呼び出し下さい。また、その値を変更するには、`set()` メソッドを呼び出します。あなたがこのプロトコルに従えば、あなた側のこれ以上の介入が無くてもウィジェットは常に変数の値を追跡記録します。

例えば:

```
class App(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()

        self.entrythingy = Entry()
        self.entrythingy.pack()

        self.button.pack()
        # ここではアプリケーション変数
        self.contents = StringVar()
        # それをある値に設定します
        self.contents.set("this is a variable")
        # エントリウィジェットにこの変数を監視するように指示します
        self.entrythingy["textvariable"] = self.contents

        # そして、ここではユーザがリターンを押したときにコールバックを得ます。
        # ユーザがリターンを押したときに、アプリケーション変数の値を表示する
        # プログラムとなります。
        self.entrythingy.bind('<Key-Return>',
                               self.print_contents)

    def print_contents(self, event):
        print "hi. contents of entry is now ---->", \
              self.contents.get()
```

## ウィンドウマネージャ

Tk には、ウィンドウマネージャとやり取りするために、便利なコマンド `wm` があります。`wm` コマンドへのオプションを使うとタイトル、位置、アイコン・ビットマップなどのようなものをコントロールすることができます。Tkinter では、これらのコマンドが `Wm` クラスのメソッドとして実装されています。トップレベル・ウィジェットは `Wm` クラスからサブクラス化されるので、直接 `Wm` のメソッドを呼び出します。

与えられたウィジェットを含むトップレベル・ウィンドウを得るために、多くの場合ウィジェットのマスタへ問い合わせることができます。もちろん、ウィジェットがフレームの内部にパックされているならば、マスタはトップレベル・ウィンドウを意味しないでしょう。任意のウィジェットのトップレベル・ウィンドウを得るために、`_root()` メソッドを呼び出すことができます。このメソッドはこの関数が実装の一部であり、Tk 機能へのインターフェイスではないという事実を示す下線で始まっています。

以下に典型的な使い方の例をいくつか挙げます：

```
import Tkinter
class App(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()

# アプリケーションを作成します
myapp = App()

#
# ここは、ウィンドウマネージャ・クラスへのメソッド呼び出し
#
myapp.master.title("My Do-Nothing Application")
myapp.master.maxsize(1000, 400)

# プログラムを開始します
myapp.mainloop()
```

## Tk オプションデータ型

**anchor** 指定できる値はコンパスの方位です: "n"、"ne"、"e"、"se"、"s"、"sw"、"w"、"nw"、および"center"。

**bitmap** 八つの組み込み、名前付きビットマップ: 'error'、'gray25'、'gray50'、'hourglass'、'info'、'questhead'、'question'、'warning'。X ビットマップファイル名を指定するために、"@/usr/contrib/bitmap/gumby.bit"のような@を先頭に付けたファイルへの完全なパスを与えてください。

**boolean** 整数 0 または 1、あるいは、文字列 "yes" または "no" を渡すことができます。

**callback** これは引数を取らない Python 関数ならどれでも構いません。例えば：

```
def print_it():
    print "hi there"
fred["command"] = print_it
```

**color** 色は `rgb.txt` ファイルの X カラーの名前か、または RGB 値を表す文字列として与えられます。RGB 値を表す文字列は、4 ビット: "#RGB", 8 bit: "#RRGGBB", 12 bit: "#RRRGGBBBB", あるいは、16 bit

"#RRRRGGGGBBBB"の範囲を取ります。ここでは、R,G,B は適切な十六進数ならどんなものでも表します。詳細は、Ousterhout の本の 160 ページを参照してください。

**cursor** 'cursorfont.h' の標準 X カーソル名を、接頭語 XC\_無しで使うことができます。例えば、hand カーソル (XC\_hand2) を得るには、文字列 "hand2" を使ってください。あなた自身のビットマップとマスキュアファイルを指定することもできます。Ousterhout の本の 179 ページを参照してください。

**distance** スクリーン上の距離をピクセルか絶対距離のどちらかで指定できます。ピクセルは数として与えられ、絶対距離は文字列として与えられます。絶対距離を表す文字列は、単位を表す終了文字 (センチメートルには c、インチには i、ミリメートルには m、プリンタのポイントには p) を伴います。例えば、3.5 インチは "3.5i" と表現します。

**font** Tk は {courier 10 bold} のようなリストフォント名形式を使います。正の数のフォントサイズはポイント単位で表され、負の数のサイズはピクセル単位で表されます。

**geometry** これは 'widthxheight' 形式の文字列です。ここでは、ほとんどのウィジェットに対して幅と高さピクセル単位で (テキストを表示するウィジェットに対しては文字単位で) 表されます。例えば: fred["geometry"] = "200x100"。

**justify** 指定できる値は文字列です: "left"、"center"、"right"、and "fill"。

**region** これは空白で区切られた四つの要素をもつ文字列です。各要素は指定可能な距離です (以下を参照)。例えば: "2 3 4 5" と "3i 2i 4.5i 2i" と "3c 2c 4c 10.43c" は、すべて指定可能な範囲です。

**relief** ウィジェットのボーダのスタイルが何かを決めます。指定できる値は: "raised"、"sunken"、"flat"、"groove"、and "ridge"。

**scrollcommand** これはほとんど常にスクロールバー・ウィジェットの set() メソッドですが、一引数を取るどんなウィジェットメソッドでもあり得ます。例えば、Python ソース配布の 'Demo/tkinter/matt/canvas-with-scrollbars.py' ファイルを参照してください。

**wrap:** 次の中の一つでなければならない: "none"、"char"、あるいは "word"。

## バインドとイベント

ウィジェットコマンドからの bind メソッドによって、あるイベントを待つことと、そのイベント型が起きたときにコールバック関数を呼び出すことができるようになります。bind メソッドの形式は:

```
def bind(self, sequence, func, add='')
```

ここでは:

**sequence** は対象とするイベントの型を示す文字列です。(詳細については、bind の man ページと John Ousterhout の本の 201 ページを参照してください。)

**func** は一引数を取り、イベントが起きるときに呼び出される Python 関数です。イベント・インスタスが引数として渡されます。(このように実施される関数は、一般に *callbacks* として知られています。)

**add** はオプションで、' ' か '+' のどちらかです。空文字列を渡すことは、このイベントが関係する他のどんなバインドをもこのバインドが置き換えることを意味します。 '+' を使う仕方は、この関数がこのイベント型にバインドされる関数のリストに追加されることを意味しています。

例えば:

```
def turnRed(self, event):
    event.widget["activeforeground"] = "red"

self.button.bind("<Enter>", self.turnRed)
```

イベントのウィジェットフィールドが `turnRed()` コールバック内でどのようにアクセスされているかに注意してください。このフィールドは X イベントを捕らえるウィジェットを含んでいます。以下の表はあなたがアクセスできる他のイベントフィールドとそれらの Tk での表現方法の一覧です。Tk man ページを参照するときに役に立つでしょう。

Tk	Tkinter イベントフィールド	Tk	Tkinter イベントフィールド
--	-----	--	-----
%f	focus	%A	char
%h	height	%E	send_event
%k	keycode	%K	keysym
%s	state	%N	keysym_num
%t	time	%T	type
%w	width	%W	widget
%x	x	%X	x_root
%y	y	%Y	y_root

## index パラメータ

たくさんのウィジェットが渡される “index” パラメータを必要とします。これらはテキストウィジェットでの特定の場所や、エントリウィジェットでの特定の文字、あるいは、メニューウィジェットでの特定のメニュー項目を指定するために使われます。

エントリウィジェットのインデックス (インデックス、ビューインデックスなど) エントリウィジェットは表示されているテキスト内の文字位置を参照するオプションを持っています。テキストウィジェットにおけるこれらの特別な位置にアクセスするために、これらの Tkinter 関数を使うことができます:

**AtEnd()** テキストの最後の位置を参照します

**AtInsert()** テキストカーソルの位置を参照します

**AtSelFirst()** 選択されたテキストの先頭の位置を指します

**AtSelLast()** 選択されているテキストおよび最終的に選択されたテキストの末尾の位置を示します。

**At(x[,y])** ピクセル位置 *x, y*(テキストを一行だけ含むテキストエントリウィジェットの場合には *y* は使われない) の文字を参照します。

テキストウィジェットのインデックス テキストウィジェットに対するインデックス記法はとても機能が豊富で、Tk man ページでよく説明されています。

メニューのインデックス (**menu.invoke()**、**menu.entryconfig()** など) メニューに対するいくつかのオプションとメソッドは特定のメニュー項目を操作します。メニューインデックスはオプションまたはパラメータのために必要とされるときはいつでも、以下のものを渡すことができます:

- 頭から数えられ、0 で始まるウィジェットの数字の位置を指す整数。
- 文字列 'active'、現在カーソルがあるメニューの位置を指します。
- 最後のメニューを指す文字列 "last"。

- @6 のような@が前に来る整数。ここでは、整数がメニューの座標系における y ピクセル座標として解釈されます。
- 文字列 "none"、どんなメニューエントリもまったく指しておらず、ほとんどの場合、すべてのエントリの動作を停止させるために `menu.activate()` と一緒に使われます。そして、最後に、
- メニューの先頭から一番下までスキャンしたときに、メニューエントリのラベルに一致したパターンであるテキスト文字列。このインデックス型は他すべての後に考慮されることに注意してください。その代わりに、それは `last`、`active` または `none` とラベル付けされたメニュー項目への一致は上のリテラルとして解釈されることを意味します。

## 画像

Bitmap/Pixmap 画像を `Tkinter.Image` のサブクラスを使って作ることができます:

- `BitmapImage` は X11 ビットマップデータに対して使えます。
- `PhotoImage` は GIF と PPM/PGM カラービットマップに対して使えます。

画像のどちらの型でも `file` または `data` オプションを使って作られます (その上、他のオプションも利用できます)。

`image` オプションがウィジェットにサポートされる場所ならどこでも、画像オブジェクトを使うことができます (例えば、ラベル、ボタン、メニュー)。これらの場合では、Tk は画像への参照を保持しないでしょう。画像オブジェクトへの最後の Python の参照が削除されたときに、おまけに画像データが削除されます。そして、どこで画像が使われているようにも、Tk は空の箱を表示します。

## 16.2 Tix — Tk の拡張ウィジェット

Tix (Tk Interface Extension) モジュールは豊富な追加ウィジェットを提供します。標準 Tk ライブラリには多くの有用なウィジェットがありますが、完全では決してありません。Tix ライブラリは標準 Tk に欠けている一般的に必要なとされるウィジェットの大部分を提供します: `HList`、`ComboBox`、`Control` (別名 `SpinBox`) および各種のスクロール可能なウィジェット。Tix には、一般的に幅広い用途に役に立つたくさんのウィジェットも含まれています: `NoteBook`、`FileEntry`、`PanedWindow` など。それらは 40 以上あります。

これら全ての新しいウィジェットと使うと、より便利でより直感的なユーザインターフェイス作成し、あなたは新しい相互作用テクニックをアプリケーションに導入することができます。アプリケーションとユーザに特有の要求に合うように、大部分のアプリケーションウィジェットを選ぶことによって、アプリケーションを設計できます。

参考資料:

*Tix Homepage*

(<http://tix.sourceforge.net/>)

Tix のホームページ。ここには追加ドキュメントとダウンロードへのリンクがあります。

*Tix Man Pages*

(<http://tix.sourceforge.net/dist/current/man/>)

man ページと参考資料のオンライン版。

*Tix Programming Guide*

(<http://tix.sourceforge.net/dist/current/docs/tix-book/tix.book.html>)

プログラマ用参考資料のオンライン版。

(<http://tix.sourceforge.net/Tide/>)

Tix と Tkinter プログラムの開発のための Tix アプリケーション。Tide アプリケーションは Tk または Tkinter に基づいて動作します。また、リモートで Tix/Tk/Tkinter アプリケーションを変更やデバッグするためのインスペクタ **TixInspect** が含まれます。

### 16.2.1 Tix を使う

```
class Tix(screenName[, baseName[, className]])
```

たいていはアプリケーションのメインウィンドウを表す Tix のトップレベルウィジェット。それには Tcl インタープリタが付随します。

Tix モジュールのクラスは Tkinter モジュールのクラスをサブクラス化します。前者は後者をインポートします。だから、Tkinter と一緒に Tix を使うためにやらなければならないのは、モジュールを一つインポートすることだけです。一般的に、Tix をインポートし、トップレベルでの Tkinter.Tk の呼び出しを Tix.Tk に置き換えるだけでよいのです：

```
import Tix
from Tkconstants import *
root = Tix.Tk()
```

Tix を使うためには、通常 Tk ウィジェットのインストールと平行して、Tix ウィジェットをインストールしなければなりません。インストールをテストするために、次のことを試してください：

```
import Tix
root = Tix.Tk()
root.tk.eval('package require Tix')
```

これが失敗した場合は、先に進む前に解決しなければならない問題が Tk のインストールにあることになります。インストールされた Tix ライブラリを指定するためには環境変数 TIX\_LIBRARY を使ってください。Tk 動的オブジェクトライブラリ ('tk8183.dll' または 'libtk8183.so') を含むディレクトリと同じディレクトリに、動的オブジェクトライブラリ ('tix8183.dll' または 'libtix8183.so') があるかどうかを確かめてください。動的オブジェクトライブラリのあるディレクトリには、'pkgIndex.tcl' (大文字、小文字を区別します) という名前のファイルも含まれているべきで、それには次の一行が含まれます：

```
package ifneeded Tix 8.1 [list load "[file join $dir tix8183.dll]" Tix]
```

### 16.2.2 Tix ウィジェット

TixTix<http://tix.sourceforge.net/dist/current/man/html/TixCmd/TixIntro.htm> は 40 個以上のウィジェットクラスを Tkinter のレパートリーに導入します。標準配布の 'Demo/tix' ディレクトリには、Tix ウィジェットのデモがあります。

#### 基本ウィジェット

```
class Balloon()
```

ヘルプを提示するためにウィジェット上にポップアップする Balloon。ユーザがカーソルを Balloon ウィジェットが束縛されているウィジェット内部へ移動させたとき、説明のメッセージが付いた小さなポップアップウィンドウがスクリーン上に表示されます。



**class ButtonBox()**

ButtonBox ウィジェットは、Ok Cancel のためだけに普通は使われるようなボタンボックスを作成します。

**class ComboBox()**

ComboBox ウィジェットは MS Windows のコンボボックスコントロールに似ています。ユーザはエントリ・サブウィジェットでタイプするか、リストボックス・サブウィジェットから選択するかのどちらかで選択肢を選びます。

**class Control()**

Control ウィジェットは SpinBox ウィジェットとしても知られています。ユーザは二つの矢印ボタンを押すか、またはエントリに直接値を入力して値を調整します。新しい値をユーザが定義した上限と下限に対してチェックします。

**class LabelEntry()**

LabelEntry ウィジェットはエントリウィジェットとラベルを一つのメガウィジェットにまとめたものです。“記入形式”型のインターフェイスの作成を簡単に行うために使うことができます。

**class LabelFrame()**

LabelFrame ウィジェットはフレームウィジェットとラベルを一つのメガウィジェットにまとめたものです。LabelFrame ウィジェット内部にウィジェットを作成するためには、frame サブウィジェットに対して新しいウィジェットを作成し、それらを frame サブウィジェット内部で取り扱います。

**class Meter()**

Meter ウィジェットは実行に時間のかかるバックグラウンド・ジョブの進み具合を表示するために使用できます。

**class OptionMenu()**

OptionMenu はオプションのメニューボタンを作成します。

**class PopupMenu()**

PopupMenu ウィジェットは tk\_popup コマンドの代替品として使用できます。Tix PopupMenu ウィジェットの利点は、操作するためにより少ないアプリケーション・コードしか必要としないことです。

**class Select()**

Select ウィジェットはボタン・サブウィジェットのコンテナです。ユーザに対する選択オプションのラジオボックスまたはチェックボックス形式を提供するために利用することができます。

**class StdButtonBox()**

StdButtonBox ウィジェットは、Motif に似たダイアログボックスのための標準的なボタンのグループです。

## ファイルセクタ

**class DirList()**

DirList ウィジェットは、ディレクトリのリストビュー（その前のディレクトリとサブディレクトリ）を表示します。ユーザはリスト内の表示されたディレクトリの一つを選択したり、あるいは他のディレクトリへ変更したりできます。

**class DirTree()**

DirTree ウィジェットはディレクトリのツリービュー（その前のディレクトリとそのサブディレクトリ）を表示します。ユーザはリスト内に表示されたディレクトリの一つを選択したり、あるいは他のディレクトリに変更したりできます。

**class DirSelectDialog()**

DirSelectDialog ウィジェットは、ダイアログウィンドウにファイルシステム内のディレクトリを提示

します。望みのディレクトリを選択するために、ユーザはファイルシステムを介して操作するこのダイアログウィンドウを利用できます。

**class DirSelectBox()**

DirSelectBox は標準 Motif(TM) ディレクトリ選択ボックスに似ています。ユーザがディレクトリを選択するために一般的に使われます。DirSelectBox は主に最近 ComboBox ウィジェットに選択されたディレクトリを保存し、すばやく再選択できるようにします。

**class ExFileSelectBox()**

ExFileSelectBox ウィジェットは、たいてい tixExFileSelectDialog ウィジェット内に組み込まれます。ユーザがファイルを選択するのに便利なメソッドを提供します。ExFileSelectBox ウィジェットのスタイルは、MS Windows 3.1 の標準ファイルダイアログにとてもよく似ています。

**class FileSelectBox()**

FileSelectBox は標準的な Motif(TM) ファイル選択ボックスに似ています。ユーザがファイルを選択するために一般的に使われます。FileSelectBox は主に最近 ComboBox ウィジェットに選択されたファイルを保存し、素早く再選択できるようにします。

**class FileEntry()**

FileEntry ウィジェットはファイル名を入力するために使うことができます。ユーザは手でファイル名をタイプできます。その代わりに、ユーザはエントリの横に並んでいるボタンウィジェットを押すことができます。それはファイル選択ダイアログを表示します。

## ハイアラキカルリストボックス

**class HList()**

HList ウィジェットは階層構造をもつどんなデータ (例えば、ファイルシステムディレクトリツリー) でも表示するために使用できます。リストエントリは字下げされ、階層のそれぞれの場所に応じて分岐線で接続されます。

**class CheckList()**

CheckList ウィジェットは、ユーザが選ぶ項目のリストを表示します。CheckList は Tk のチェックリストやラジオボタンより多くの項目を扱うことができることを除いて、チェックボタンあるいはラジオボタンウィジェットと同じように動作します。

**class Tree()**

Tree ウィジェットは階層的なデータをツリー形式で表示するために使うことができます。ユーザはツリーの一部を開いたり閉じたりすることによって、ツリーの見えを調整できます。

## タビュラーリストボックス

**class TList()**

TList ウィジェットは、表形式でデータを表示するために使うことができます。TList ウィジェットのリスト・エントリは、Tk のリストボックス・ウィジェットのエントリに似ています。主な差は、(1) TList ウィジェットはリスト・エントリを二次元形式で表示でき、(2) リスト・エントリに対して複数の色やフォントだけでなく画像も使うことができるということです。

## 管理ウィジェット

**class PanedWindow()**

PanedWindow ウィジェットは、ユーザがいくつかのペインのサイズを対話的に操作できるようにします。ペインは垂直または水平のどちらかに配置されます。ユーザは二つのペインの間でリサイズ・ハ

ンドルをドラッグしてペインの大きさを変更します。

**class ListNoteBook()**

ListNoteBook ウィジェットは、TixNoteBook ウィジェットにとってもよく似ています。ノートのメタファを使って限られた空間をに多くのウィンドウを表示するために使われます。ノートはたくさんのページ(ウィンドウ)に分けられています。ある時には、これらのページの一つしか表示できません。ユーザはhlist サブウィジェットの中の望みのページの名前を選択することによって、これらのページを切り替えることができます。

**class Notebook()**

NoteBook ウィジェットは、ノートのメタファを多くのウィンドウを表示することができます。ノートはたくさんのページに分けられています。ある時には、これらのページの一つしか表示できません。ユーザはNoteBook ウィジェットの一番上にある目に見える“タブ”を選択することで、これらのページを切り替えることができます。

## 画像タイプ

Tix モジュールは次のものを追加します:

- 全ての Tix と Tkinter ウィジェットに対して XPM ファイルからカラー画像を作成する pixmap 機能。
- Compound 画像タイプは複数の水平方向の線から構成される画像を作成するために使うことができます。それぞれの線は左から右に並べられた一連のアイテム(テキスト、ビットマップ、画像あるいは空白)から作られます。例えば、Tk の Button ウィジェットの中にビットマップとテキスト文字列を同時に表示するために compound 画像は使われます。

## その他のウィジェット

**class InputOnly()**

InputOnly ウィジェットは、ユーザから入力を受け付けます。それは、bind コマンドを使って行われます (UNIX のみ)。

## ジオメトリマネージャを作る

加えて、Tix は次のものを提供することで Tkinter を補強します:

**class Form()**

Tk ウィジェットに対する接続ルールに基づいたジオメトリマネージャを作成 (Form) します。

## 16.2.3 Tix コマンド

**class tixCommand()**

tix コマンドは Tix の内部状態と Tix アプリケーション・コンテキストのいろいろな要素へのアクセスを提供します。これらのメソッドによって操作される情報の大部分は、特定のウィンドウというよりむしろアプリケーション全体がスクリーンあるいはディスプレイに関するものです。

現在の設定を見るための一般的な方法は、

```
import Tix
root = Tix.Tk()
print root.tix_configure()
```

`tix_configure([cnf,] **kw)`

Tix アプリケーション・コンテキストの設定オプションを問い合わせたり、変更したりします。オプションが指定されなければ、利用可能なオプションすべてのディクショナリを返します。オプションが値なしで指定された場合は、メソッドは指定されたオプションを説明するリストを返します(このリストはオプションが指定されていない場合に返される値に含まれている、指定されたオプションに対応するサブリストと同一です)。一つ以上のオプション-値のペアが指定された場合は、メソッドは与えられたオプションが与えられた値を持つように変更します。この場合は、メソッドは空文字列を返します。オプションは設定オプションのどれでも構いません。

`tix_cget(option)`

*option* によって与えられた設定オプションの現在の値を返します。オプションは設定オプションのどれでも構いません。

`tix_getbitmap(name)`

ビットマップディレクトリの一つの中の *name.xpm* または *name* という名前のビットマップファイルの場所を見つけ出します(`tix_addbitmapdir()` メソッドを参照してください)。`tix_getbitmap()` を使うことで、アプリケーションにビットマップファイルのパス名をハードコーディングすることを避けることができます。成功すれば、文字 '@' を先頭に付けたビットマップファイルの完全なパス名を返します。戻り値を Tk と Tix ウィジェットの `bitmap` オプションを設定するために使うことができます。

`tix_addbitmapdir(directory)`

Tix は `tix_getimage()` と `tix_getbitmap()` メソッドが画像ファイルを検索するディレクトリのリストを保持しています。標準ビットマップディレクトリは '\$TIX\_LIBRARY/bitmaps' です。`tix_addbitmapdir()` メソッドは *directory* をこのリストに追加します。そのメソッドを使うことによって、アプリケーションの画像ファイルを `tix_getimage()` または `tix_getbitmap()` メソッドを使って見つけることができます。

`tix_filedialog([dlgclass])`

このアプリケーションからの異なる呼び出しの間で共有される可能性があるファイル選択ダイアログを返します。最初に呼ばれた時に、このメソッドはファイル選択ダイアログ・ウィジェットを作成します。このダイアログはその後のすべての `tix_filedialog()` への呼び出しで返されます。オプションの *dlgclass* パラメータは、要求されているファイル選択ダイアログ・ウィジェットの型を指定するために文字列として渡されます。指定可能なオプションは `tix`、`FileSelectDialog` あるいは `tixExFileSelectDialog` です。

`tix_getimage(self, name)`

ビットマップディレクトリの一つの中の '*name.xpm*'、'*name.xbm*' または '*name.ppm*' という名前の画像ファイルの場所を見つけ出します(上の `tix_addbitmapdir()` メソッドを参照してください)。同じ名前(だが異なる拡張子)のファイルが一つ以上ある場合は、画像のタイプが X ディスプレイの深さに応じて選択されます。`xbm` 画像はモノクロディスプレイの場合に選択され、カラー画像はカラーディスプレイの場合に選択されます。`tix_getimage()` を使うことによって、アプリケーションに画像ファイルのパス名をハードコーディングすることを避けられます。成功すれば、このメソッドは新たに作成した画像の名前を返し、Tk と Tix ウィジェットの `image` オプションを設定するためにそれを使うことができます。

`tix_option_get(name)`

Tix のスキーム・メカニズムによって保持されているオプションを得ます。

`tix_resetoptions(newScheme, newFontSet[, newScmPrio])`

Tix アプリケーションのスキームとフォントセットを *newScheme* と *newFontSet* それぞれへと再設定します。これはこの呼び出し後に作成されたそれらのウィジェットだけに影響します。そのため、Tix

アプリケーションのどんなウィジェットを作成する前に `resetoptions` メソッドを呼び出すのが最も良いのです。

オプション・パラメータ `newScmPrio` を、Tix スキームによって設定される Tk オプションの優先度レベルを再設定するために与えることができます。

Tk が X オプションデータベースを扱う方法のため、Tix がインポートされ初期化された後に、カラースキームとフォントセットを `tix_config()` メソッドを使って再設定することができません。その代わりに、`tix_resetoptions()` メソッドを使わなければならないのです。

## 16.3 ScrolledText — スクロールするテキストウィジェット

`ScrolledText` モジュールは“正しい動作”をするように設定された垂直スクロールバーをもつ基本的なテキストウィジェットを実装する同じ名前のクラスを提供します。`ScrolledText` クラスを使うことは、テキストウィジェットとスクロールバーを直接設定するより簡単です。コンストラクタは `Tkinter.Text` クラスのものを同じです。

テキストウィジェットとスクロールバーは `Frame` の中に一緒に `pack` され、`Grid` と `Pack` ジオメトリマネージャのメソッドは `Frame` オブジェクトから得られます。これによって、もっとも標準的なジオメトリマネージャの振る舞いにするために、直接 `ScrolledText` ウィジェットを使えるようになります。

特定の制御が必要ならば、以下の属性が利用できます:

**frame**

テキストとスクロールバーウィジェットを取り囲むフレーム。

**vbar**

スクロールバーウィジェット。

## 16.4 turtle — Tkのためのタートルグラフィックス

`turtle` モジュールはオブジェクト指向と手続き指向の両方の方法でタートルグラフィックス・プリミティブを提供します。グラフィックスの基礎として `Tkinter` を使っているために、Tk をサポートした python のバージョンが必要です。

手続き型インターフェイスでは、関数のどれかが呼び出されたときに自動的に作られるペンとキャンバスを使います。

`turtle` モジュールは次の関数を定義しています:

**degrees()**

角度を計る単位を度にします。

**radians()**

角度を計る単位をラジアンにします。

**reset()**

スクリーンを消去し、ペンを中心に持って行き、変数をデフォルト値に設定します。

**clear()**

スクリーンを消去します。

**tracer(flag)**

トレースを on/off にします (フラグが真かどうかに応じて)。トレースとは、線に沿って矢印のアニメーションが付き、線がよりゆっくりと引かれることを意味します。

**forward**(*distance*)

*distance* ステップだけ前に進みます。

**backward**(*distance*)

*distance* ステップだけ後ろに進みます。

**left**(*angle*)

*angle* 単位だけ左に回ります。単位のデフォルトは度ですが、`degrees()` と `radians()` 関数を使って設定できます。

**right**(*angle*)

*angle* 単位だけ右に回ります。単位のデフォルトは度ですが、`degrees()` と `radians()` 関数を使って設定できます。

**up**()

ペンを上げます — 線を引くことを止めます。

**down**()

ペンを下げます — 移動したときに線を引きます。

**width**(*width*)

線幅を *width* に設定します。

**color**(*s*)

**color**((*r*, *g*, *b*))

**color**(*r*, *g*, *b*)

ペンの色を設定します。最初の形式では、色は文字列として Tk の色の仕様の通りに指定されます。二番目の形式は色を RGB 値 (それぞれは範囲 [0..1]) のタプルとして指定します。三番目の形式では、色は三つに別れたパラメータとして RGB 値 (それぞれは範囲 [0..1]) を与えて指定しています。

**write**(*text*[, *move*])

現在のペンの位置に *text* を書き込みます。*move* が真ならば、ペンはテキストの右下の角へ移動します。デフォルトでは、*move* は偽です。

**fill**(*flag*)

完全な仕様はかなり複雑ですが、推奨する使い方は: 塗りつぶしたい経路を描く前に `fill(1)` を呼び出し、経路を描き終えたときに `fill(0)` を呼び出します。

**circle**(*radius*[, *extent*])

半径 *radius*、中心がタートルの左 *radius* ユニットの円を描きます。*extent* は円のどの部分を描くかを決定します: 与えられなければ、デフォルトで完全な円になります。

*extent* が完全な円である場合は、弧の一つの端点は、現在のペンの位置です。*radius* が正の場合、弧は反時計回りに描かれます。そうでなければ、時計回りです。

**goto**(*x*, *y*)

**goto**((*x*, *y*))

座標 *x*, *y* へ移動します。座標は二つの別個の引数か、2-タプルのどちらかで指定することができます。

このモジュールは `from math import *` も実行します。従って、タートルグラフィックスのために役に立つ追加の定数と関数については、`math` モジュールのドキュメントを参照してください。

**demo**()

モジュールをちょっとばかり試しています。

**exception Error**

このモジュールによって捕捉されたあらゆるエラーに対して発生した例外。

例として、`demo()` 関数のコードを参照してください。



このモジュールは次のクラスを定義します:

```
class Pen( )
```

ペンを定義します。上記のすべての関数は与えられたペンのメソッドとして呼び出されます。このコンストラクタは線を描くキャンパスを自動的に作成します。

```
class RawPen( canvas )
```

キャンパス *canvas* に描くペンを定義します。これは“実際の”プログラムでグラフィックスを作成するためにモジュールを使いたい場合に役に立ちます。

### 16.4.1 Pen と RawPen オブジェクト

Pen と RawPen オブジェクトは、`demo()` がメソッドとしては除かれますが、与えられたペンを操作する上記のすべてのグローバル関数を持っています。

メソッドになって強力になっているメソッドは `degrees()` だけです。

```
degrees( [fullcircle] )
```

*fullcircle* はデフォルトで 360 です。たとえ *fullcircle* にラジアンで  $2\pi$ 、あるいは度で 400 を与えようと、これはペンがどんな角度単位でも取ることができるようにしています。

## 16.5 Idle

Idle は Tkinter GUI ツールキットをつかって作られた Python IDE です。

IDLE は次のような特徴があります:

- Tkinter GUI ツールキットを使って、100% ピュア Python でコーディングされています
- クロス-プラットフォーム: Windows と UNIX で動作します (Mac OS では、現在 Tcl/Tk に問題があります)
- 多段 Undo、Python 対応の色づけや他にもたくさんの機能 (例えば、自動的な字下げや呼び出し情報の表示) をもつマルチ-ウィンドウ・テキストエディタ
- Python シェルウィンドウ (別名、対話インタープリタ)
- デバッガ (完全ではありませんが、ブレークポイントの設定や値の表示、ステップ実行ができます)

### 16.5.1 メニュー

File メニュー

**New window** 新しい編集ウィンドウを作成します

**Open...** 既存のファイルをオープンします

**Open module...** 既存のモジュールをオープンします (sys.path を検索します)

**Class browser** 現在のファイルの中のクラスとモジュールを示します

**Path browser** sys.path ディレクトリ、モジュール、クラスおよびメソッドを示します

**Save** 現在のウィンドウを対応するファイルにセーブします (未セーブのウィンドウには、ウィンドウタイトルの前後に\*があります)

**Save As...** 現在のウィンドウを新しいファイルへセーブします。そのファイルが対応するファイルになります

**Save Copy As...** 現在のウィンドウを対応するファイルを変えずに異なるファイルにセーブします。

**Close** 現在のウィンドウを閉じます (未セーブの場合はセーブするか質問します)

**Exit** すべてのウィンドウを閉じて IDLE を終了します (未セーブの場合はセーブするか質問します)

## Edit メニュー

**Undo** 現在のウィンドウに対する最後の変更を Undo(取り消し) します (最大で 1000 個の変更)

**Redo** 現在のウィンドウに対する最後に undo された変更を Redo(再実行) します

**Cut** システムのクリップボードへ選択された部分をコピーします。それから選択された部分を削除します

**Copy** 選択された部分をシステムのクリップボードへコピーします

**Paste** システムのクリップボードをウィンドウへ挿入します

**Select All** 編集バッファの内容全体を選択します

**Find...** たくさんのオプションをもつ検索ダイアログボックスを開きます

**Find again** 最後の検索を繰り返します

**Find selection** 選択された文字列を検索します

**Find in Files...** 検索するファイルに対する検索ダイアログボックスを開きます

**Replace...** 検索と置換ダイアログボックスを開きます

**Go to line** 行番号を尋ね、その行を表示します

**Indent region** 選択された行を右へ空白 4 個分シフトします

**Dedent region** 選択された行を左へ空白 4 個分シフトします

**Comment out region** 選択された行の先頭に##を挿入します

**Uncomment region** 選択された行から先頭の#あるいは##を取り除きます

**Tabify region** 先頭の一続きの空白をタブに置き換えます

**Untabify region** すべてのタブを適切な数の空白に置き換えます

**Expand word** あなたがタイプした語を同じバッファの別の語に一致するように展開します。そして、異なる展開が得るために繰り返します

**Format Paragraph** 現在の空行で区切られた段落を再フォーマットします

**Import module** 現在のモジュールをインポートまたはリロードします

**Run script** 現在のファイルを\_\_main\_\_名前空間内で実行します

## Windows メニュー

**Zoom Height** ウィンドウを標準サイズ (24x80) と最大の高さの間で切り替えます

このメニューの残りはすべての開いたウィンドウの名前の一覧になっています。一つを選ぶとそれを最前面に持ってくることができます (必要ならばアイコン化をやめさせます)

**Debug メニュー** (Python シェルウィンドウ内のみ)

**Go to file/line** 挿入ポイントの周りからファイル名と行番号を探し、ファイルをオープンし、その行を表示します

**Open stack viewer** 最後の例外のスタックトレースバックを表示します

**Debugger toggle** デバッガの下、シェル内でコマンドを実行します

**JIT Stack viewer toggle** トレースバック上のスタックビューアをオープンします

## 16.5.2 基本的な編集とナビゲーション

- Backspace は左側を削除し、Del は右側を削除します
- 矢印キーと Page Up/Page Down はそれぞれ移動します
- Home/End は行の始め/終わりへ移動します
- C-Home/C-End はファイルの始め/終わりへ移動します
- C-B、C-P、C-A、C-E、C-D、C-L を含む、いくつかの Emacs バインディングも動作します

### 自動的な字下げ

ブロックの始まりの文の後、次の行は 4 つの空白 (Python Shell ウィンドウでは、一つのタブ) で字下げされます。あるキーワード (break、return など) の後では、次の行は字下げが解除 (dedent) されます。先頭の字下げでは、Backspace は 4 つの空白があれば削除します。Tab は 1-4 つの空白 (Python Shell ウィンドウでは一つのタブ) を挿入します。edit メニューの indent/dedent region コマンドも参照してください。

### Python Shell ウィンドウ

- C-C 実行中のコマンドを中断します
- C-D ファイル終端 (end-of-file) を送り、'»>' プロンプトでタイプしていた場合はウィンドウを閉じます
- Alt-p あなたがタイプしたことに一致する以前のコマンドを取り出します
- Alt-n 次を取り出します
- Return 以前のコマンドを取り出しているときは、そのコマンド
- Alt-/ (語を展開します) ここでも便利です

### 16.5.3 構文の色づけ

色づけはバックグラウンド“スレッド”で適用され、そのため時折色付けされないテキストが見えます。カラスキームを変えるには、‘config.txt’の[Colors]節を編集してください。

Python の構文の色: キーワード オレンジ

文字列 緑

コメント 赤

定義 青

シェルの色: コンソールの出力 茶色

stdout 青

stderr 暗い緑

stdin 黒

コマンドラインの使い方

```
idle.py [-c command] [-d] [-e] [-s] [-t title] [arg] ...
```

-c コマンド このコマンドを実行します  
-d デバッガを有効にします  
-e 編集モード、引数は編集するファイルです  
-s \$IDLESTARTUP または \$PYTHONSTARTUP を最初に実行します  
-t タイトル シェルウィンドウのタイトルを設定します

引数がある場合:

1. -e が使われる場合は、引数は編集のためにオープンされるファイルで、`sys.argv` は IDLE 自体へ渡される引数を反映します。
2. そうではなく、-c が使われる場合には、すべての引数が `sys.argv[1:...]` の中に置かれ、`sys.argv[0]` が ‘-c’ に設定されます。
3. そうではなく、-e でも -c でも使われない場合は、最初の引数は `sys.argv[1:...]` にある残りの引数とスクリプト名に設定される `sys.argv[0]` と一緒に実行されるスクリプトです。スクリプト名が ‘-’ のときは、実行されるスクリプトはありませんが、対話的な Python セッションが始まります。引数はまだ `sys.argv` にあり利用できます。

## 16.6 他のグラフィカルユーザインターフェイスパッケージ

Tkinter へ付け加えられるたくさんの拡張ウィジェットがあります。

Python メガウィジェット

(<http://pmw.sourceforge.net/>)

Tkinter モジュールを使い Python で高レベルの複合ウィジェットを構築するためのツールキットです。基本クラスとこの基礎の上に構築された柔軟で拡張可能なメガウィジェットから構成されています。これらのメガウィジェットはノートブック、コンボボックス、選択ウィジェット、ペインウィジェット、スクロールするウィジェット、ダイアログウィンドウなどを含みます。BLT に対する Pmw.Blt インターフェイスを持ち、busy、graph、stripchart、tabset および vector コマンドが利用できます。

Pmw の最初のアイディアは、Michael McLennan による Tk itcl 拡張 [incr Tk] と Mark Ulberts による [incr Widgets] から得ました。メガウィジェットのいくつかは itcl から Python へ直接変換したものです。[incr Widgets] が提供するウィジェットとほぼ同等のものを提供します。そして、Tix と同様にほぼ完成しています。しかしながら、ツリーを描くための Tix の高速な HList ウィジェットが欠けています。

*Tkinter3000 Widget Construction Kit (WCK)*

(<http://tkinter.effbot.org/>)

は、新しい Tkinter ウィジェットを、Python で書けるようにするライブラリです。WCK フレームワークは、ウィジェットの生成、設定、スクリーンの外観、イベント操作における、完全な制御を提供します。Tk/Tcl レイヤーを通してデータ転送する必要がなく、直接 Python のデータ構造を操作することができるので、WCK ウィジェットは非常に高速で軽量になり得ます。

Tk は Python に対する唯一の GUI ではありませんが、もっとも一般的に使われているものです。

*wxWindows*

(<http://www.wxwindows.org>)

Qt、Tk、Motif および GTK+ のもっとも魅力のある性質を一つのパッケージに結合した GUI ツールキットです。C++ で実装されています。wxWindows は二種類の UNIX 実装をサポートしています: GTK+ と Motif。Windows では、標準的な Microsoft Foundation Classes (MFC) の外観を持っています。なぜなら、Win32 ウィジェットを使っているからです。Tkinter に依存しない Python クラスブラウザがあります。

wxWindows は Tkinter よりさらにウィジェットが豊富で、そのヘルプシステム、洗練された HTML と画像ビューアおよび他の専門分野別のウィジェット、多数のドキュメントと印刷機能を持っています。

*PyQt*

PyQt は sip でラップされた Qt ツールキットへのバインディングです。Qt は UNIX、Windows および Mac OS X で利用できる大規模な C++ GUI ツールキットです。sip は Python クラスとして C++ ライブラリに対するバインディングを生成するためのツールキットで、特に Python 用に設計されています。オンライン・マニュアルは <http://www.opendocspublishing.com/pyqt/> (正誤表は <http://www.valdyas.org/python/book.html> にあります) で手に入ります。

*PyKDE*

(<http://www.riverbankcomputing.co.uk/pykde/index.php>)

PyKDE は sip でラップされた KDE デスクトップライブラリに対するインターフェイスです。KDE は UNIX コンピュータ用のデスクトップ環境です。グラフィカル・コンポーネントは Qt に基づいています。

*FXPy*

(<http://fxpy.sourceforge.net/>)

FOX GUI へのインターフェイスを提供する Python 拡張モジュールです。FOX は、グラフィカルユーザインターフェイスを簡単かつ効率良く開発するための C++ ベースのツールキットです。それは幅広く、成長しているコントロール・コレクションで、3D グラフィックスの操作のための OpenGL ウィジェットと同様に、ドラッグアンドドロップ、選択のような最新の機能を提供します。FOX はアイコン、画像およびステータスライン・ヘルプやツールチップのようなユーザにとって便利な機能も実装しています。

FOX はすでに大規模なコントロール・コレクションを提供していますが、単に既存のコントロールを使って望みの振る舞いを追加または再定義する派生クラスを作成することによってプログラマが簡単に追加コントロールと GUI 要素を構築できるようにするために、FOX は C++ を利用しています。

*PyGTK*

(<http://www.daa.com.au/~Tl\textasciitildejames/software/pygtk/>)

GTK ウィジェットセットのための一連のバインディングです。C のものより少しだけ高レベルなオブジェクト指向インターフェイスを提供します。普通は C API を使ってやらなければならない型キャストとリファレンス・カウントをすべて自動的に行います。GNOME に対しても、バインディングがあります。チュートリアルが手に入ります。



## 制限実行 (restricted execution)

警告: Python 2.3 では、既知の容易に修正できないセキュリティーホールのために、これらのモジュールは無効にされています。rexec や Bastion モジュールを使った古いコードを読むときに助けになるよう、モジュールのドキュメントだけは残されています。

制限実行 (*restricted execution*) とは、信頼できるコードと信頼できないコードを区別できるようにするための Python における基本的なフレームワークです。このフレームワークは、信頼できる Python コード (スーパーバイザ (*supervisor*)) が、パーミッションに制限のかけられた “拘束セル (padded cell)” を生成し、このセル中で信頼のおけないコードを実行するという概念に基づいています。信頼のおけないコードはこの拘束セルを破ることができず、信頼されたコードで提供され、管理されたインタフェースを介してのみ、傷つきやすいシステムリソースとやりとりすることができます。“制限実行” という用語は、“安全な Python (safe-Python)” を裏から支えるものです。というのは、真の安全を定義することは難しく、制限された環境を生成する方法によって決められるからです。制限された環境は入れ子にすることができ、このとき内側のセルはより縮小されることはあるが決して拡大されることのない特権を持ったサブセルを生成します。

Python の制限実行モデルの興味深い側面は、信頼されないコードに提供されるインタフェースが、信頼されるコードに提供されるそれらと同じ名前を持つということです。このため、制限された環境で動作するよう設計されたコードを書く上で特殊なインタフェースを学ぶ必要がありません。また、拘束セルの厳密な性質はスーパーバイザによって決められるため、アプリケーションによって異なる制限を課すことができます。例えば、信頼されないコードが指定したディレクトリ内の何らかのファイルを読み出すが決して書き込まないということが “安全” と考えられるかもしれません。この場合、スーパーバイザは組み込みの `open()` 関数について、`mode` パラメタが `'w'` の時に例外を送出するように再定義できます。また例えば、“安全” とは、`filename` パラメタに対して `chroot()` に似た操作を施して、ルートパスがファイルシステム上の何らかの安全な “砂場 (sandbox)” 領域に対する相対パスになるようにすることもかもしれません。この場合でも、信頼されないコードは依然として、もとの呼び出しインタフェースを持ったままの組み込みの `open()` 関数を制限環境中に見出します。ここでは、関数に対する意味付け (semantics) は同じですが、許可されないパラメタが使われようとしているとスーパーバイザが判断した場合には `IOError` が送出されます。

Python のランタイムシステムは、特定のコードブロックが制限実行モードかどうかを、グローバル変数の中の `__builtins__` オブジェクトの一意性をもとに判断します: オブジェクトが標準の `__builtin__` モジュール (の辞書) の場合、コードは非制限下にあるとみなされます。それ以外は制限下にあるとみなされます。

制限実行モードで動作する Python コードは、拘束セルから侵出しないように設計された数多くの制限に直面します。例えば、関数オブジェクト属性 `func_globals` や、クラスおよびインスタンスオブジェクトの属性 `__dict__` は利用できません。

二つのモジュールが、制限実行環境を立ち上げるためのフレームワークを提供しています:

**rexec**      基本的な制限実行フレームワーク。

**Bastion**    オブジェクトに対するアクセスの制限を提供する。

参考資料:

*Grail Home Page*

(<http://grail.sourceforge.net/>)

Python で書かれたインターネットブラウザ Grail です。Python で書かれたアプレットをサポートするために、上記のモジュールを使っています。Grail における Python 制限実行モードの利用に関する詳しい情報は、Web サイトで入手することができます。

## 17.1 rexec — 制限実行のフレームワーク

2.3 で変更された仕様: Disabled module

警告: このドキュメントは、`rexec` モジュールを使用している古いコードを読む際の参照用として残されています。

このモジュールには `RExec` クラスが含まれています。このクラスは、`r_eval()`、`r_execfile()`、`r_exec()` および `r_import()` メソッドをサポートし、これらは標準の Python 関数 `eval()`、`execfile()` および `exec` と `import` 文の制限されたバージョンです。この制限された環境で実行されるコードは、安全であると見なされたモジュールや関数だけにアクセスします；`RExec` をサブクラス化すれば、望むように能力を追加および削除できます。

警告: `rexec` モジュールは、下記のように動作するべく設計されていますが、注意深く書かれたコードなら利用できてしまうかもしれない、既知の脆弱性がいくつかあります。従って、“製品レベル”のセキュリティを要する状況では、`rexec` の動作をあてにするべきではありません。製品レベルのセキュリティを求めるなら、サブプロセスを介した実行や、あるいは処理するコードとデータの両方に対する非常に注意深い“浄化”が必要でしょう。上記の代わりに、`rexec` の既知の脆弱性に対するパッチ当ての手伝いも歓迎します。

注意: `RExec` クラスは、プログラムコードによるディスクファイルの読み書きや TCP/IP ソケットの利用といった、安全でない操作の実行を防ぐことができます。しかし、プログラムコードによる非常に大量のメモリや処理時間の消費に対して防御することはできません。

```
class RExec([hooks[, verbose]])
```

`RExec` クラスのインスタンスを返します。

`hooks` は、`RHooks` クラスあるいはそのサブクラスのインスタンスです。`hooks` が省略されているか `None` であれば、デフォルトの `RHooks` クラスがインスタンス化されます。`rexec` モジュールが(組み込みモジュールを含む)あるモジュールを探したり、あるモジュールのコードを読んだりする時は常に、`rexec` がじかにファイルシステムに出て行くことはありません。その代わりに、あらかじめ `RHooks` クラスに渡しておいたり、コンストラクタで生成された `RHooks` インスタンスのメソッドを呼び出します。

(実際には、`RExec` オブジェクトはこれら呼び出しません — 呼び出しは、`RExec` オブジェクトの一部であるモジュールローダオブジェクトによって行われます。これによって別のレベルの柔軟性が実現されます。この柔軟性は、制限された環境内で `import` 機構を変更する時に役に立ちます。)

代替の `RHooks` オブジェクトを提供することで、モジュールをインポートする際に行われるファイルシステムへのアクセスを制御することができます。このとき、各々のアクセスが行われる順番を制御する実際のアルゴリズムは変更されません。例えば、`RHooks` オブジェクトを置き換えて、ILU のようなある種の RPC メカニズムを介することで、全てのファイルシステムの要求をどこかにあるファイルサーバに渡すことができます。Grail のアプレットローダは、アプレットを URL からディレクトリ上に `import` する際にこの機構を使っています。

もし `verbose` が `true` であれば、追加のデバッグ出力が標準出力に送られます。

制限された環境で実行するコードも、やはり `sys.exit()` 関数を呼ぶことができることを知っておくことは大事なことです。制限されたコードがインタプリタから抜けだすことを許さないためには、いつで

も、制限されたコードが、SystemExit 例外をキャッチする try/except 文とともに実行するように、呼び出しを防御します。制限された環境から sys.exit() 関数を除去するだけでは不十分です – 制限されたコードは、やはり raise SystemExit を使うことができてしまいます。SystemExit を取り除くことも、合理的なオプションではありません；いくつかのライブラリコードはこれを使っていますし、これが利用できなくなると中断してしまうでしょう。

参考資料:

Grail のホームページ

(<http://grail.sourceforge.net/>)

Grail はすべて Python で書かれた Web ブラウザです。これは、rexec モジュールを、Python アプリットをサポートするのに使っていて、このモジュールの使用例として使うことができます。

### 17.1.1 RExec オブジェクト

RExec インスタンスは以下のメソッドをサポートします：

**r\_eval**(code)

code は、Python の式を含む文字列か、あるいはコンパイルされたコードオブジェクトのどちらかでなければなりません。そしてこれらは制限された環境の \_\_main\_\_ モジュールで評価されます。式あるいはコードオブジェクトの値が返されます。

**r\_exec**(code)

code は、1 行以上の Python コードを含む文字列か、コンパイルされたコードオブジェクトのどちらかでなければなりません。そしてこれらは、制限された環境の \_\_main\_\_ モジュールで実行されます。

**r\_execfile**(filename)

ファイル filename 内の Python コードを、制限された環境の \_\_main\_\_ モジュールで実行します。

名前が 's\_' で始まるメソッドは、'r\_' で始まる関数と同様ですが、そのコードは、標準 I/O ストリーム sys.stdin、sys.stderr および sys.stdout の制限されたバージョンへのアクセスが許されています。

**s\_eval**(code)

code は、Python 式を含む文字列でなければなりません。そして制限された環境で評価されます。

**s\_exec**(code)

code は、1 行以上の Python コードを含む文字列でなければなりません。そして制限された環境で実行されます。

**s\_execfile**(code)

ファイル filename に含まれた Python コードを制限された環境で実行します。

RExec オブジェクトは、制限された環境で実行されるコードによって暗黙のうちに呼ばれる、さまざまなメソッドもサポートしなければなりません。これらのメソッドをサブクラス内でオーバーライドすることによって、制限された環境が強制するポリシーを変更します。

**r\_import**(modulename[, globals[, locals[, fromlist]])

モジュール modulename をインポートし、もしそのモジュールが安全でないとみなされるなら、ImportError 例外を発生します。

**r\_open**(filename[, mode[, bufsize]])

open() が制限された環境で呼ばれるとき、呼ばれるメソッドです。引数は open() のものと同じであり、ファイルオブジェクト (あるいはファイルオブジェクトと互換性のあるクラスインスタンス) が返されます。RExec のデフォルトの動作は、任意のファイルを読み取り用にオープンすることを許可しますが、ファイルに書き込もうとすることは許しません。より制限の少ない r\_open() の実装については、以下の例を見て下さい。

`r_reload(module)`

モジュールオブジェクト *module* を再ロードして、それを再解析し再初期化します。

`r_unload(module)`

モジュールオブジェクト *module* をアンロードします (それを制限された環境の `sys.modules` 辞書から取りのぞきます)。

および制限された標準 I/O ストリームへのアクセスが可能な同等のもの：

`s_import(modulename[, globals[, locals[, fromlist]]])`

モジュール *modulename* をインポートし、もしそのモジュールが安全でないとみなされるなら、`ImportError` 例外を発生します。

`s_reload(module)`

モジュールオブジェクト *module* を再ロードして、それを再解析し再初期化します。

`s_unload(module)`

モジュールオブジェクト *module* をアンロードします。

### 17.1.2 制限された環境を定義する

`RExec` クラスには以下のクラス属性があります。それらは、`__init__()` メソッドが使います。それらを既存のインスタンス上で変更しても何の効果もありません；そうする代わりに、`RExec` のサブクラスを作成して、そのクラス定義でそれらに新しい値を割り当てます。そうすると、新しいクラスのインスタンスは、これらの新しい値を使用します。これらの属性のすべては、文字列のタプルです。

`nok_builtin_names`

制限された環境で実行するプログラムでは利用できないであろう、組み込み関数の名前を格納しています。`RExec` に対する値は、`('open', 'reload', '__import__')` です。(これは例外です。というのは、組み込み関数のほとんど大多数は無害だからです。この変数をオーバーライドしたいサブクラスは、基本クラスからの値から始めて、追加した許されない関数を連結していかなければなりません – 危険な関数が新しく Python に追加された時は、それらも、このモジュールに追加します。)

`ok_builtin_modules`

安全にインポートできる組み込みモジュールの名前を格納しています。`RExec` に対する値は、`('audioop', 'array', 'binascii', 'cmath', 'errno', 'imageop', 'marshal', 'math', 'md5', 'operator', 'parser', 'regex', 'rotor', 'select', 'sha', '_sre', 'strop', 'struct', 'time')` です。この変数をオーバーライドする場合も、同様な注意が適用されます – 基本クラスからの値を使って始めます。

`ok_path`

`import` が制限された環境で実行される時に検索されるディレクトリーを格納しています。`RExec` に対する値は、(モジュールがロードされた時は) 制限されないコードの `sys.path` と同一です。

`ok_posix_names`

制限された環境で実行するプログラムで利用できる、`os` モジュール内の関数の名前を格納しています。`RExec` に対する値は、`('error', 'fstat', 'listdir', 'lstat', 'readlink', 'stat', 'times', 'uname', 'getpid', 'getppid', 'getcwd', 'geteuid', 'getgid', 'getteuid', 'getegid')` です。

`ok_sys_names`

制限された環境で実行するプログラムで利用できる、`sys` モジュール内の関数名と変数名を格納しています。`RExec` に対する値は、`('ps1', 'ps2', 'copyright', 'version', 'platform', 'exit', 'maxint')` です。

`ok_file_types`

モジュールがロードすることを許されているファイルタイプを格納しています。各ファイルタイプは、`imp` モジュールで定義された整数定数です。意味のある値は、`PY_SOURCE`、`PY_COMPILED` および `C_EXTENSION` です。`RExec` に対する値は、`(C_EXTENSION, PY_SOURCE)` です。サブクラスで `PY_COMPILED` を追加することは推奨されません；攻撃者が、バイトコンパイルしたでっあげのファイル（`.pyc`）を、例えば、あなたの公開 FTP サーバの `/tmp` に書いたり、`/incoming` にアップロードしたりして、とにかくあなたのファイルシステム内に置くことで、制限された実行モードから抜け出ることができるかもしれないからです。

### 17.1.3 例

標準の `RExec` クラスよりも、若干、もっと緩めたポリシーを望んでいるとしましょう。例えば、もし `/tmp` 内のファイルへの書き込みを喜んで許すならば、`RExec` クラスを次のようにサブクラス化できます：

```
class TmpWriterRExec(rexec.RExec):
    def r_open(self, file, mode='r', buf=-1):
        if mode in ('r', 'rb'):
            pass
        elif mode in ('w', 'wb', 'a', 'ab'):
            # ファイル名をチェックします： /tmp/ で始まらなければなりません
            if file[:5] != '/tmp/':
                raise IOError, " /tmp 以外へは書き込みできません"
            elif (string.find(file, '/../') >= 0 or
                  file[:3] == '/../' or file[-3:] == '/../'):
                raise IOError, "ファイル名の '../' は禁じられています"
            else: raise IOError, "open() モードが正しくありません"
        return open(file, mode, buf)
```

上のコードは、完全に正しいファイル名でも、時には禁止する場合があることに注意して下さい；例えば、制限された環境でのコードでは、`/tmp/foo/../bar` というファイルはオープンできないかもしれません。これを修正するには、`r_open()` メソッドが、そのファイル名を `/tmp/bar` に単純化しなければなりません。そのためには、ファイル名を分割して、それにさまざまな操作を行う必要があります。セキュリティが重大な場合には、より複雑で、微妙なセキュリティホールを抱え込むかもしれない、一般性のあるコードよりも、制限が余りにあり過ぎるとしても単純なコードを書く方が、望ましいでしょう。

## 17.2 Bastion — オブジェクトに対するアクセスの制限

2.3 で変更された仕様: Disabled module

警告：このドキュメントは、Bastion モジュールを使用している古いコードを読む際の参照用として残されています。

辞書によると、バステイアン (bastion、要塞) とは、“防衛された領域や地点”、または“最後の砦と考えられているもの”であり、オブジェクトの特定の属性へのアクセスを禁じる方法を提供するこのモジュールにふさわしい名前です。制限モード下のプログラムに対して、あるオブジェクトにおける特定の安全な属性へのアクセスを許可し、かつその他の安全でない属性へのアクセスを拒否するには、要塞オブジェクトは常に `rexec` モジュールと共に使われなければなりません。

**Bastion**(*object*[, *filter*[, *name*[, *class*]]])

オブジェクト *object* を保護し、オブジェクトに対する要塞オブジェクトを返します。オブジェクトの属性に対するアクセスの試みは全て、*filter* 関数によって認可されなければなりません；アクセスが拒否された場合 `AttributeError` 例外が送出されます。

*filter* が存在する場合、この関数は属性名を含む文字列を受理し、その属性に対するアクセスが許可



される場合には真を返さなければなりません; *filter* が偽を返す場合、アクセスは拒否されます。標準のフィルタは、アンダースコア ('\_') で始まる全ての関数に対するアクセスを拒否します。 *name* の値が与えられた場合、要塞オブジェクトの文字列表現は '<Bastion for *name*>' になります; そうでない場合、 '`repr(object)`' が使われます。

*class* が存在する場合、 `BastionClass` のサブクラスでなくてはなりません; 詳細は '`bastion.py`' のコードを参照してください。稀に `BastionClass` の標準設定を上書きする必要ほとんどないはずです。

`class BastionClass(getfunc, name)`

実際に要塞オブジェクトを実装しているクラスです。このクラスは `Bastion()` によって使われる標準のクラスです。 *getfunc* 引数は関数で、唯一の引数である属性の名前を与えて呼び出した際、制限された実行環境に対して、開示すべき属性の値を返します。 *name* は `BastionClass` インスタンスの `repr()` を構築するために使われます。



# Python 言語サービス

Python には Python 言語を使って作業するときに役に立つモジュールがたくさん提供されています。これらのモジュールはトークンの切り出し、パース、構文解析、バイトコードのディスアセンブリおよびその他のさまざまな機能をサポートしています。

これらのモジュールには、次のものが含まれています:

<b>parser</b>	Python ソースコードに対する解析木へのアクセス。
<b>symbol</b>	Constants representing internal nodes of the parse tree.
<b>token</b>	Constants representing terminal nodes of the parse tree.
<b>keyword</b>	文字列が Python のキーワードか否かを調べます。
<b>tokenize</b>	Python ソースコードのための字句解析器。
<b>tabnanny</b>	ディレクトリツリー内の Python のソースファイルで問題となる空白を検出するツール。
<b>pyclbr</b>	Python クラスデスク립タの情報抽出サポート
<b>py_compile</b>	Python ソースファイルをバイトコードファイルへコンパイル。
<b>compileall</b>	ディレクトリに含まれる Python ソースファイルを、一括してバイトコンパイルします。
<b>dis</b>	Python バイトコードの逆アセンブラ。
<b>distutils</b>	現在インストールされている Python に追加するためのモジュール構築、および実際のインストールを

## 18.1 parser — Python 解析木にアクセスする

`parser` モジュールは Python の内部パーサとバイトコード・コンパイラへのインターフェイスを提供します。このインターフェイスの第一の目的は、Python コードから Python の式の解析木を編集したり、これから実行可能なコードを作成したりできるようにすることです。これは任意の Python コードの断片を文字列として構文解析や変更を行うより良い方法です。なぜなら、構文解析がアプリケーションを作成するコードと同じ方法で実行されるからです。その上、高速です。

このモジュールについて注意すべきことが少しあります。それは作成したデータ構造を利用するために重要なことです。この文書は Python コードの解析木を編集するためのチュートリアルではありませんが、`parser` モジュールを使った例をいくつか示しています。

もっとも重要なことは、内部パーサが処理する Python の文法についてよく理解しておく必要があるということです。言語の文法に関する完全な情報については、*Python 言語リファレンス*を参照してください。標準の Python ディストリビューションに含まれるファイル ‘Grammar/Grammar’ の中で定義されている文法仕様から、パーサ自身は作成されています。このモジュールが作成する AST オブジェクトの中に格納される解析木は、下で説明する `expr()` または `suite()` 関数によって作られるときに内部パーサから実際に出力されるものです。`sequence2ast()` が作る AST オブジェクトは忠実にこれらの構造をシミュレートしています。言語の形式文法が改訂されるために、“正しい”と考えられるシーケンスの値が Python のあるバージョンから別のバージョンで変化することがあるということに注意してください。しかし、Python のあるバージョンから別のバージョンへテキストのソースのままコードを移せば、目的のバージョンで正しい解析木を常に作成できます。ただし、インタプリタの古いバージョンへ移行する際に、最近の言語

コンストラクトをサポートしていないことがあるという制限だけがあります。ソースコードが常に前方互換性があるのに対して、一般的に解析木はあるバージョンから別のバージョンへの互換性がありません。

`ast2list()` または `ast2tuple()` から返されるシーケンスのそれぞれの要素は単純な形式です。文法の非終端要素を表すシーケンスは常に一より大きい長さを持ちます。最初の要素は文法の生成規則を識別する整数です。これらの整数は C ヘッドファイル `'Include/graminit.h'` と Python モジュール `symbol` の中の特定のシンボル名です。シーケンスに付け加えられている各要素は、入力文字列の中で認識されたままの形で生成規則の構成要素を表しています: これらは常に親と同じ形式を持つシーケンスです。この構造の注意すべき重要な側面は、`if_stmt` 中のキーワード `if` のような親ノードの型を識別するために使われるキーワードがいかなる特別な扱いもなくノードツリーに含まれているということです。例えば、`if` キーワードはタプル `(1, 'if')` と表されます。ここで、`1` は、ユーザが定義した変数名と関数名を含むすべての NAME トークンに対応する数値です。行番号情報が必要となき返される別の形式では、同じトークンが `(1, 'if', 12)` のように表されます。ここでは、`12` が終端記号の見つかった行番号を表しています。

終端要素は同じ方法で表現されますが、子の要素や識別されたソーステキストの追加は全くありません。上記の `if` キーワードの例が代表的なものです。終端記号のいろいろな型は、C ヘッドファイル `'Include/token.h'` と Python モジュール `token` で定義されています。

AST オブジェクトはこのモジュールの機能をサポートするために必要ありませんが、三つの目的から提供されています: アプリケーションが複雑な解析木を処理するコストを償却するため、Python のリストやタプル表現に比べてメモリ空間を保全する解析木表現を提供するため、解析木を操作する追加モジュールを C で作ることを簡単にするため。AST オブジェクトを使っていることを隠すために、簡単な“ラッパー”クラスを Python で作ることができます。

`parser` モジュールは二、三の別々の目的のために関数を定義しています。もっとも重要な目的は AST オブジェクトを作ることと、AST オブジェクトを解析木とコンパイルされたコードオブジェクトのような他の表現に変換することです。しかし、AST オブジェクトで表現された解析木の型を調べるために役に立つ関数もあります。

参考資料:

`symbol` モジュール (18.2 節):

解析木の内部ノードを表す便利な定数。

`token` モジュール (18.3 節):

便利な解析木の葉のノードを表す定数とノード値をテストするための関数。

### 18.1.1 AST オブジェクトを作成する

AST オブジェクトはソースコードあるいは解析木から作られます。AST オブジェクトをソースから作るときは、`'eval'` と `'exec'` 形式を作成するために別々の関数が使われます。

`expr(source)`

まるで `'compile(source, 'file.py', 'eval')` への入力であるかのように、`expr()` 関数はパラメータ `source` を構文解析します。解析が成功した場合は、AST オブジェクトは内部解析木表現を保持するために作成されます。そうでなければ、適切な例外を発生させます。

`suite(source)`

まるで `'compile(source, 'file.py', 'exec')` への入力であるかのように、`suite()` 関数はパラメータ `source` を構文解析します。解析が成功した場合は、AST オブジェクトは内部解析木表現を保持するために作成されます。そうでなければ、適切な例外を発生させます。

`sequence2ast(sequence)`

この関数はシーケンスとして表現された解析木を受け取り、可能ならば内部表現を作ります。木が Python の文法に合っていることと、すべてのノードが Python のホストバージョンで有効なノード型で

あることを確認した場合は、AST オブジェクトが内部表現から作成されて呼び出し側へ返されます。内部表現の作成に問題があるならば、あるいは木が正しいと確認できないならば、`ParserError` 例外が発生します。この方法で作られた AST オブジェクトが正しくコンパイルできると決めつけない方がよいでしょう。AST オブジェクトが `compileast()` へ渡されたとき、コンパイルによって送出された通常の例外がまだ発生するかもしれません。これは (`MemoryError` 例外のような) 構文に關係していない問題を示すのかもしれないし、`del f(0)` を解析した結果のようなコンストラクトが原因であるかもしれません。このようなコンストラクトは Python のパーサを逃れますが、バイトコードインタプリタによってチェックされます。

終端トークンを表すシーケンスは、`(1, 'name')` 形式の二つの要素のリストか、または `(1, 'name', 56)` 形式の三つの要素のリストです。三番目の要素が存在する場合は、有効な行番号だとみなされます。行番号が指定されるのは、入力木の終端記号の一部に対してです。

`tuple2ast(sequence)`

これは `sequence2ast()` と同じ関数です。このエントリポイントは後方互換性のために維持されています。

### 18.1.2 AST オブジェクトを変換する

作成するために使われた入力に關係なく、AST オブジェクトはリスト木またはタプル木として表される解析木へ変換されるか、または実行可能なオブジェクトへコンパイルされます。解析木は行番号情報を持って、あるいは持たずに抽出されます。

`ast2list(ast[, line_info])`

この関数は呼び出し側から `ast` に AST オブジェクトを受け取り、解析木と等価な Python のリストを返します。結果のリスト表現はインスペクションあるいはリスト形式の新しい解析木の作成に使うことができます。リスト表現を作るためにメモリが利用できる限り、この関数は失敗しません。解析木がインスペクションのためだけにつかわれるならば、メモリの消費量と断片化を減らすために `ast2tuple()` を代わりに使うべきです。リスト表現が必要とされるとき、この関数はタプル表現を取り出して入れ子のリストに変換するよりかなり高速です。

`line_info` が真ならば、トークンを表すリストの三番目の要素として行番号情報がすべての終端トークンに含まれます。与えられた行番号はトークンが終わる行を指定していることに注意してください。フラグが偽または省略された場合は、この情報は省かれます。

`ast2tuple(ast[, line_info])`

この関数は呼び出し側から `ast` に AST オブジェクトを受け取り、解析木と等価な Python のタプルを返します。リストの代わりにタプルを返す以外は、この関数は `ast2list()` と同じです。

`line_info` が真ならば、トークンを表すリストの三番目の要素として行番号情報がすべての終端トークンに含まれます。フラグが偽または省略された場合は、この情報は省かれます。

`compileast(ast[, filename = '<ast>'])`

`exec` 文の一部として使える、あるいは、組み込み `eval()` 関数への呼び出しとして使えるコードオブジェクトを生成するために、Python バイトコードコンパイラを AST オブジェクトに対して呼び出すことができます。この関数はコンパイラへのインターフェイスを提供し、`filename` パラメータで指定されるソースファイル名を使って、`ast` からパーサへ内部解析木を渡します。`filename` に与えられるデフォルト値は、ソースが AST オブジェクトだったことを示唆しています。

AST オブジェクトをコンパイルすることは、コンパイルに関する例外を引き起こすことになるかもしれません。例としては、`del f(0)` の解析木によって発生させられる `SyntaxError` があります：この文は Python の形式文法としては正しいと考えられますが、正しい言語コンストラクトではありません。この状況に対して発生する `SyntaxError` は、実際には Python バイトコンパイラによって

通常作り出されます。これが `parser` モジュールがこの時点で例外を発生できる理由です。解析木のインスペクションを行うことで、コンパイルが失敗するほとんどの原因をプログラムによって診断することができます。

### 18.1.3 AST オブジェクトに対する問い合わせ

AST が式または `suite` として作成されたかどうかをアプリケーションが決定できるようにする二つの関数が提供されています。これらの関数のどちらも、AST が `expr()` または `suite()` を通してソースコードから作られたかどうか、あるいは、`sequence2ast()` を通して解析木から作られたかどうかを決定できません。

**`isexpr(ast)`**

`ast` が `'eval'` 形式を表している場合に、この関数は真を返します。そうでなければ、偽を返します。これは役に立ちます。なぜならば、通常は既存の組み込み関数を使ってもコードオブジェクトに対してこの情報を問い合わせることができないからです。このどちらのようにも `compileast()` によって作成されたコードオブジェクトに問い合わせることはできませんし、そのコードオブジェクトは組み込み `compile()` 関数によって作成されたコードオブジェクトと同じであることに注意してください。

**`issuite(ast)`**

AST オブジェクトが (通常 “suite” として知られる) `'exec'` 形式を表しているかどうかを報告するという点で、この関数は `isexpr()` に酷似しています。追加の構文が将来サポートされるかもしれないので、この関数が `'not isexpr(ast)'` と等価であるとみなすのは安全ではありません。

### 18.1.4 例外とエラー処理

`parser` モジュールは例外を一つ定義していますが、Python ランタイム環境の他の部分が提供する別の組み込み例外を発生させることもあります。各関数が発生させる例外の情報については、それぞれ関数を参照してください。

**`exception ParserError`**

`parser` モジュール内部で障害が起きたときに発生する例外。普通の構文解析中に発生する組み込みの `SyntaxError` ではなく、一般的に妥当性確認が失敗した場合に引き起こされます。例外の引数としては、障害の理由を説明する文字列である場合と、`sequence2ast()` へ渡される解析木の中の障害を引き起こすシーケンスを含むタプルと説明用の文字列である場合があります。モジュール内の他の関数の呼び出しは単純な文字列値を検出すればよいのですが、`sequence2ast()` の呼び出しはどちらの例外の型も処理できる必要があります。

普通は構文解析とコンパイル処理によって発生する例外を、関数 `compileast()`、`expr()` および `suite()` が発生させることに注意してください。このような例外には組み込み例外 `MemoryError`、`OverflowError`、`SyntaxError` および `SystemError` が含まれます。こうした場合には、これらの例外が通常その例外に関係する全ての意味を伝えます。詳細については、各関数の説明を参照してください。

### 18.1.5 AST オブジェクト

AST オブジェクト間の順序と等値性の比較がサポートされています。(pickle モジュールを使った)AST オブジェクトのピクルス化もサポートされています。

**`ASTType`**

`expr()`、`suite()` と `sequence2ast()` が返すオブジェクトの型。

AST オブジェクトは次のメソッドを持っています:

```

compile([filename])
    compileast(ast, filename) と同じ。
isexpr()
    isexpr(ast) と同じ。
issuite()
    issuite(ast) と同じ。
tolist([line_info])
    ast2list(ast, line_info) と同じ。
totuple([line_info])
    ast2tuple(ast, line_info) と同じ。

```

### 18.1.6 例

parser モジュールを使うと、バイトコードが生成される前に Python のソースコードの解析木に演算を行えるようになります。また、モジュールは情報発見のために解析木のインスペクションを提供しています。例が二つあります。簡単な例では組み込み関数 `compile()` のエミュレーションを行っており、複雑な例では情報を得るための解析木の使い方を示しています。

#### `compile()` のエミュレーション

たくさんの有用な演算を構文解析とバイトコード生成の間に行うことができますが、もっとも単純な演算は何もしないことです。このため、parser モジュールを使って中間データ構造を作ることは次のコードと等価です。

```

>>> code = compile('a + 5', 'file.py', 'eval')
>>> a = 5
>>> eval(code)
10

```

parser モジュールを使った等価な演算はやや長くなりますが、AST オブジェクトとして中間内部解析木が維持されるようにします:

```

>>> import parser
>>> ast = parser.expr('a + 5')
>>> code = ast.compile('file.py')
>>> a = 5
>>> eval(code)
10

```

AST とコードオブジェクトの両方が必要なアプリケーションでは、このコードを簡単に利用できる関数にまとめることができます:



```

import parser

def load_suite(source_string):
    ast = parser.suite(source_string)
    return ast, ast.compile()

def load_expression(source_string):
    ast = parser.expr(source_string)
    return ast, ast.compile()

```

## 情報発見

あるアプリケーションでは解析木へ直接アクセスすることが役に立ちます。この節の残りでは、`import` を使って調査中のコードを実行中のインタプリタにロードする必要も無しに、解析木を使って docstrings に定義されたモジュールのドキュメンテーションへのアクセスを可能にする方法を示します。これは信頼性のないコードを解析するためにとても役に立ちます。

一般に、例は興味のある情報を引き出すために解析木をどのような方法でたどればよいかを示しています。二つの関数と一連のクラスが開発され、モジュールが提供する高レベルの関数とクラスの定義をプログラムから利用できるようになります。クラスは情報を解析木から引き出し、便利な意味レベルでその情報へアクセスできるようにします。一つの関数は単純な低レベルのパターンマッチング機能を提供し、もう一つの関数は呼び出し側の代わりにファイル操作を行うという点でクラスへの高レベルなインターフェイスです。ここで言及されていて Python のインストールに必要なすべてのソースファイルは、ディストリビューションの ‘Demo/parser/’ ディレクトリにあります。

Python の動的な性質によってプログラマは非常に大きな柔軟性を得ることができます。しかし、クラス、関数およびメソッドを定義するときには、ほとんどのモジュールがこれの限られた部分しか必要としません。この例では、考察される定義だけがコンテキストのトップレベルにおいて定義されるものです。例を挙げると、モジュールのゼロ列目に `def` 文によって定義される関数で、`if ... else` コンストラクトの枝の中に定義されていない関数（ある状況ではそうすることにもっともな理由があるのですが）。例で開発するコードによって、定義の入れ子を扱う予定です。

より上位レベルの抽出メソッドを作るために知る必要があるのは、解析木構造がどのようなものかということと、そのどの程度まで関心を持つ必要があるのかということです。Python はやや深い解析木を使いますので、たくさんの中間ノードがあります。Python が使う形式文法を読んで理解することは重要です。これは配布物に含まれるファイル ‘Grammar/Grammar’ に明記されています。docstrings を探すときに対象として最も単純な場合について考えてみてください: docstring の他に何も無いモジュール。（ファイル ‘docstring.py’ を参照してください。）

```

"""Some documentation.
"""

```

インタプリタを使って解析木を調べると、数と括弧が途方に暮れるほど多くて、ドキュメンテーションが入れ子になったタプルの深いところに埋まっていることがわかります。



```

>>> import parser
>>> import pprint
>>> ast = parser.suite(open('docstring.py').read())
>>> tup = ast.totuple()
>>> pprint.pprint(tup)
(257,
 (264,
  (265,
   (266,
    (267,
     (307,
      (287,
       (288,
        (289,
         (290,
          (292,
           (293,
            (294,
             (295,
              (296,
               (297,
                (298,
                 (299,
                  (300, (3, '""Some documentation.\n""')))))))))))))))
                (4, ''))),
 (4, ''),
 (0, ''))

```

木の各ノードの最初の要素にある数はノード型です。それらは文法の終端記号と非終端記号に直接に対応します。残念なことに、それらは内部表現の整数で表されていて、生成された Python の構造でもそのままになっています。しかし、`symbol` と `token` モジュールはノード型の記号名と整数からノード型の記号名へマッピングする辞書を提供します。

上に示した出力の中で、最も外側のタプルは四つの要素を含んでいます: 整数 257 と三つの付加的なタプル。ノード型 257 の記号名は `file_input` です。これらの各内部タプルは最初の要素として整数を含んでいます。これらの整数 264 と 4、0 は、ノード型 `stmt`、`NEWLINE`、`ENDMARKER` をそれぞれ表しています。これらの値はあなたが使っている Python のバージョンに応じて変化する可能性があることに注意してください。マッピングの詳細については、`'symbol.py'` と `'token.py'` を調べてください。もっとも外側のノードがファイルの内容ではなく入力ソースに主に関係していることはほとんど明らかで、差し当たり無視しても構いません。`stmt` ノードはさらに興味深いです。特に、すべての `docstrings` は、このノードが作られるのとまったく同じように作られ、違いがあるのは文字列自身だけである部分木にあります。同様の木の `docstring` と説明の対象である定義されたエンティティ(クラス、関数あるいはモジュール)の関係は、前述の構造を定義している木の内部における `docstring` 部分木の位置によって与えられます。

実際の `docstring` を木の変数要素を意味する何かと置き換えることによって、簡単なパターンマッチング方法で与えられたどんな部分木でも `docstrings` に対する一般的なパターンと同等かどうかを調べられるようになります。例では情報の抽出の実例を示しているので、`['variable_name']` という単純な変数表現を念頭において、リスト形式ではなくタプル形式の木を安全に要求できます。簡単な再帰関数でパターンマッチングを実装でき、その関数は真偽値と変数名から値へのマッピングの辞書を返します。(ファイル `'example.py'` を参照してください。)

```

from types import ListType, TupleType

def match(pattern, data, vars=None):
    if vars is None:
        vars = {}
    if type(pattern) is ListType:
        vars[pattern[0]] = data
        return 1, vars
    if type(pattern) is not TupleType:
        return (pattern == data), vars
    if len(data) != len(pattern):
        return 0, vars
    for pattern, data in map(None, pattern, data):
        same, vars = match(pattern, data, vars)
        if not same:
            break
    return same, vars

```

この構文の変数用の簡単な表現と記号のノード型を使うと、docstring 部分木の候補のパターンがとても読みやすくなります。(ファイル ‘example.py’ を参照してください。)

```

import symbol
import token

DOCSTRING_STMT_PATTERN = (
    symbol.stmt,
    (symbol.simple_stmt,
     (symbol.small_stmt,
      (symbol.expr_stmt,
       (symbol.testlist,
        (symbol.test,
         (symbol.and_test,
          (symbol.not_test,
           (symbol.comparison,
            (symbol.expr,
             (symbol.xor_expr,
              (symbol.and_expr,
               (symbol.shift_expr,
                (symbol.arith_expr,
                 (symbol.term,
                  (symbol.factor,
                   (symbol.power,
                    (symbol.atom,
                     (token.STRING, ['docstring']))
                    ))))))))))))))),
            (token.NEWLINE, ''))
            ))

```

このパターンと `match()` 関数を使うと、前に作った解析木からモジュールの docstring を簡単に抽出できます:

```

>>> found, vars = match(DOCSTRING_STMT_PATTERN, tup[1])
>>> found
1
>>> vars
{'docstring': '""Some documentation.\n""'}

```

特定のデータを期待された位置から抽出できると、次は情報を期待できる場所はどこかという疑問に答える

必要がでできます。docstring を扱う場合、答えはとても簡単です: docstring はコードブロック (file\_input または suite ノード型) の最初の stmt ノードです。モジュールは一つの file\_input ノードと、正確にはそれぞれが一つの suite ノードを含むクラスと関数の定義で構成されます。クラスと関数は (stmt, (compound\_stmt, (classdef, ... または (stmt, (compound\_stmt, (funcdef, ... で始まるコードブロックノードの部分木として簡単に識別されます。これらの部分木は match() によってマッチさせることができないことに注意してください。なぜなら、数を見捨てて複数の兄弟ノードにマッチすることをサポートしていないからです。この限界を超えるためにより念入りにつくったマッチング関数を使うことができますが、例としてはこれで充分です。

文が docstring かどうかを決定し、実際の文字列をその文から抽出する機能について考えると、ある作業にはモジュール全体の解析木を巡回してモジュールの各コンテキストにおいて定義される名前についての情報を抽出し、その名前と docstrings を結び付ける必要があります。この作業を行うコードは複雑ではありませんが、説明が必要です。

そのクラスへの公開インターフェイスは簡単で、おそらく幾分かより柔軟でしょう。モジュールのそれぞれの“主要な”ブロックは、問い合わせのための幾つかのメソッドを提供するオブジェクトと、少なくともそれが表す完全な解析木の部分木を受け取るコンストラクタによって記述されます。ModuleInfo コンストラクタはオプションの name パラメータを受け取ります。なぜなら、そうしないとモジュールの名前を決められないからです。

公開クラスには ClassInfo、FunctionInfo および ModuleInfo が含まれます。すべてのオブジェクトはメソッド get\_name()、get\_docstring()、get\_class\_names() および get\_class\_info() を提供します。ClassInfo オブジェクトは get\_method\_names() と get\_method\_info() をサポートしますが、他のクラスは get\_function\_names() と get\_function\_info() を提供しています。

公開クラスが表すコードブロックの形式のそれぞれにおいて、トップレベルで定義された関数が“メソッド”として参照されるという違いがクラスにはありますが、要求される情報のほとんどは同じ形式をしていて、同じ方法でアクセスされます。クラスの外側で定義される関数との実際の意味の違いを名前の付け方が違うことで反映しているため、実装はこの違いを保つ必要があります。そのため、公開クラスのほとんどの機能が共通の基底クラス SuiteInfoBase に実装されており、他の場所で提供される関数とメソッドの情報に対するアクセサを持っています。関数とメソッドの情報を表すクラスが一つだけであることに注意してください。これは要素の両方の型を定義するために def 文を使うことに似ています。

アクセサ関数のほとんどは SuiteInfoBase で宣言されていて、サブクラスでオーバーライドする必要はありません。より重要なこととしては、解析木からのほとんどの情報抽出が SuiteInfoBase コンストラクタに呼び出されるメソッドを通して行われるということがあります。平行して形式文法を読めば、ほとんどのクラスのコード例は明らかです。しかし、再帰的に新しい情報オブジェクトを作るメソッドはもっと調査が必要です。‘example.py’ の SuiteInfoBase 定義の関連する箇所を以下に示します:

```

class SuiteInfoBase:
    _docstring = ''
    _name = ''

    def __init__(self, tree = None):
        self._class_info = {}
        self._function_info = {}
        if tree:
            self._extract_info(tree)

    def _extract_info(self, tree):
        # extract docstring
        if len(tree) == 2:
            found, vars = match(DOCSTRING_STMT_PATTERN[1], tree[1])
        else:
            found, vars = match(DOCSTRING_STMT_PATTERN, tree[3])
        if found:
            self._docstring = eval(vars['docstring'])
        # discover inner definitions
        for node in tree[1:]:
            found, vars = match(COMPOUND_STMT_PATTERN, node)
            if found:
                cstmt = vars['compound']
                if cstmt[0] == symbol.funcdef:
                    name = cstmt[2][1]
                    self._function_info[name] = FunctionInfo(cstmt)
                elif cstmt[0] == symbol.classdef:
                    name = cstmt[2][1]
                    self._class_info[name] = ClassInfo(cstmt)

```

初期状態に初期化した後、コンストラクタは `_extract_info()` メソッドを呼び出します。このメソッドがこの例全体で行われる情報抽出の大部分を実行します。抽出には二つの別々の段階があります: 渡された解析木の `docstring` の位置の特定、解析木が表すコードブロック内の付加的な定義の発見。

最初の `if` テストは入れ子の suite が“短い形式”または“長い形式”かどうかを決定します。以下のコードブロックの定義のように、コードブロックが同じ行であるときに短い形式が使われます。

```

def square(x): "Square an argument."; return x ** 2

```

長い形式では字下げされたブロックを使い、入れ子になった定義を許しています:

```

def make_power(exp):
    "Make a function that raises an argument to the exponent 'exp'."
    def raiser(x, y=exp):
        return x ** y
    return raiser

```

短い形式が使われるとき、コードブロックは `docstring` を最初の `small_stmt` 要素として (ことによるとそれだけを) 持っています。このような `docstring` の抽出は少し異なり、より一般的な場合に使われる完全なパターンの一部だけを必要とします。実装されているように、`simple_stmt` ノードに `small_stmt` ノードが一つだけある場合には、`docstring` しかないことがあります。短い形式を使うほとんどの関数とメソッドが `docstring` を提供しないため、これで充分だと考えられます。`docstring` の抽出は前述の `match()` 関数を使って進み、`docstring` が `SuiteInfoBase` オブジェクトの属性として保存されます。

`docstring` を抽出した後、簡単な定義発見アルゴリズムを `suite` ノードの `stmt` ノードに対して実行します。短い形式の特別な場合はテストされません。短い形式では `stmt` ノードが存在しないため、アルゴ

リズムは黙って `simple_stmt` ノードを一つスキップします。正確に言えば、どんな入れ子になった定義も見ません。

コードブロックのそれぞれの文をクラス定義 (関数またはメソッドの定義、あるいは、何か他のもの) として分類します。定義文に対しては、定義された要素の名前が抽出され、コンストラクタに引数として渡される部分木の定義とともに定義に適した代理オブジェクトが作成されます。代理オブジェクトはインスタンス変数に保存され、適切なアクセサメソッドを使って名前から取り出されます。

公開クラスは `SuiteInfoBase` クラスが提供するアクセサより具体的で、必要とされるどんなアクセサでも提供します。しかし、実際の抽出アルゴリズムはコードブロックのすべての形式に対して共通のままです。高レベルの関数をソースファイルから完全な情報のセットを抽出するために使うことができます。(ファイル `'example.py'` を参照してください。)

```
def get_docs(fileName):
    import os
    import parser

    source = open(fileName).read()
    basename = os.path.basename(os.path.splitext(fileName)[0])
    ast = parser.suite(source)
    return ModuleInfo(ast.totuple(), basename)
```

これはモジュールのドキュメンテーションに対する使いやすいインターフェイスです。この例のコードで抽出されない情報が必要な場合は、機能を追加するための明確に定義されたところで、コードを拡張することができます。

## 18.2 `symbol` — Python 解析木と共に使われる定数

このモジュールは解析木の内部ノードの数値を表す定数を提供します。ほとんどの Python 定数とは違い、これらは小文字の名前を使います。言語の文法のコンテキストにおける名前の定義については、Python ディストリビューションのファイル `'Grammar/Grammar'` を参照してください。名前がマップする特定の数値は Python のバージョン間で変わります。

このモジュールには、データオブジェクトも一つ付け加えられています:

`sym_name`

ディクショナリはこのモジュールで定義されている定数の数値を名前の文字列へマップし、より人が読みやすいように解析木を表現します。

参考資料:

`parser` モジュール (18.1 節):

`parser` モジュールの二番目の例で、`symbol` モジュールの使い方を示しています。

## 18.3 `token` — Python 解析木と共に使われる定数

このモジュールは解析木の葉ノード (終端記号) の数値を表す定数を提供します。言語の文法のコンテキストにおける名前の定義については、Python ディストリビューションのファイル `'Grammar/Grammar'` を参照してください。名前がマップする特定の数値は、Python のバージョン間で変わります。

このモジュールは一つデータオブジェクトといくつかの関数も提供します。関数は Python の C ヘッドファイルの定義を反映します。

`tok_name`

辞書はこのモジュールで定義されている定数の数値を名前の文字列へマップし、より人が読みやすいように解析木を表現します。

`ISTERMINAL(x)`

終端トークンの値に対して真を返します。

`ISNONTERMINAL(x)`

非終端トークンの値に対して真を返します。

`ISEOF(x)`

$x$  が入力の終わりを示すマーカーならば、真を返します。

参考資料:

parser モジュール (18.1 節):

parser モジュールの二番目の例で、symbol モジュールの使い方を示しています。

## 18.4 keyword — Python キーワードチェック

このモジュールでは、Python プログラムで文字列がキーワードか否かをチェックする機能を提供します。

`iskeyword(s)`

$s$  が Python のキーワードであれば真を返します。

`kwlist`

インタプリタで定義している全てのキーワードのシーケンス。特定の `__future__` 宣言がなければ有効ではないキーワードでもこのリストには含まれます。

## 18.5 tokenize — Python ソースのためのトークナイザ

`tokenize` モジュールでは、Python で実装された Python ソースコードの字句解析器を提供します。さらに、このモジュールの字句解析器はコメントもトークンとして返します。このため、このモジュールはスクリーン上で表示する際の色付け機能 (colorizers) を含む “清書出力器 (pretty-printer)” を実装する上で便利です。

第一のエントリポイントはジェネレータです:

`generate_tokens(readline)`

`generate_tokens()` ジェネレータは一つの引数 `readline` を必要とします。この引数は呼び出し可能オブジェクトで、組み込みファイルオブジェクトにおける `readline()` メソッドと同じインタフェースを提供していなければなりません (2.3.8 節を参照してください)。この関数は呼び出しのたびに入力内の一行を文字列で返さなければなりません。

ジェネレータは 5 要素のタプルを返し、タプルは以下のメンバ: トークン型; トークン文字列; ソースコード中でトークンが始まる行と列を示す整数の 2 要素のタプル (`srow`, `scol`); ソースコード中でトークンが終わる行と列を示す整数の 2 要素のタプル (`erow`, `ecol`); そして、トークンが見つかった行、からなります。渡される行は論理行です; 連続する行は一行に含まれます。2.2 で追加された仕様です。

後方互換性のために古いエントリポイントが残されています:

`tokenize(readline[, tokeneater])`

`tokenize()` 関数は二つのパラメータを取ります: 一つは入力ストリームを表し、もう一つは `tokenize()` のための出力メカニズムを与えます。

最初のパラメータ、`readline` は、組み込みファイルオブジェクトの `readline()` メソッドと同じイ



インタフェイスを提供する呼び出し可能オブジェクトでなければなりません (2.3.8 節を参照)。この関数は呼び出しのたびに入力内の一行を文字列で返さなければなりません。

二番目のパラメータ *tokeneater* も呼び出し可能オブジェクトでなければなりません。この関数は各トークンに対して一度だけ呼び出され、`generate_tokens()` が生成するタプルに対応する 5 つの引数をとります。

`token` モジュールの全ての定数は `tokenize` でも公開されており、これに加え、以下の二つのトークン値が `tokenize()` の *tokeneater* 関数に渡される可能性があります:

#### COMMENT

コメントであることを表すために使われるトークン値です。

#### NL

終わりではない改行を表すために使われるトークン値。NEWLINE トークンは Python コードの論理行の終わりを表します。NL トークンはコードの論理行が複数の物理行にわたって続いているときに作られます。

## 18.6 tabnanny — あいまいなインデントの検出

差し当たり、このモジュールはスクリプトとして呼び出すことを意図しています。しかし、IDE 上にインポートして下で説明する関数 `check()` を使うことができます。

警告: このモジュールが提供する API を将来のリリースで変更する確率が高いです。このような変更は後方互換性がないかもしれません。

#### `check(file_or_dir)`

*file\_or\_dir* がディレクトリであってシンボリックリンクでないときに、*file\_or\_dir* という名前のディレクトリツリーを再帰的に下って行き、この通り道に沿ってすべての `.py` ファイルを変更します。*file\_or\_dir* が通常の Python ソースファイルの場合には、問題のある空白をチェックします。診断メッセージは `print` 文を使って標準出力に書き込まれます。

#### `verbose`

冗長なメッセージをプリントするかどうかを示すフラグ。スクリプトとして呼び出された場合は、`-v` オプションによって増加します。

#### `filename_only`

問題のある空白を含むファイルのファイル名のみをプリントするかどうかを示すフラグ。スクリプトとして呼び出された場合は、`-q` オプションによって真に設定されます。

#### `exception NannyNag`

あいまいなインデントを検出した場合に `tokeneater()` によって発生させられます。`check()` で捕捉され処理されます。

#### `tokeneater(type, token, start, end, line)`

この関数は関数 `tokenize.tokenize()` へのコールバックパラメータとして `check()` によって使われます。

#### 参考資料:

`tokenize` モジュール (18.5 節):

Python ソースコードの字句解析器。

## 18.7 pycclbr — Python クラスブラウザーサポート

この `pycclbr` はモジュールで定義されたクラス、メソッド、およびトップレベルの関数について、限られた量の情報を定義するのに使われます。このクラスによって提供される情報は、伝統的な 3 ペイン形式のクラスブラウザーを実装するのに十分なだけの量になります。情報はモジュールのインポートによらず、ソースコードから抽出します。このため、このモジュールは信用できないソースコードに対して利用しても安全です。この制限から、多くの標準モジュールやオプションの拡張モジュールを含む、Python で実装されていないモジュールに対して利用することはできません。

`readmodule(module[, path])`

モジュールを読み込み、辞書マッピングクラスを返し、クラス記述オブジェクトに名前をつけます。パラメタ `module` はモジュール名を表す文字列でなくてはなりません; パッケージ内のモジュール名でもかまいません。 `path` パラメタは配列型でなくてはならず、モジュールのソースコードがある場所を特定する際に `sys.path` の値に補完する形で使われます。

`readmodule_ex(module[, path])`

`readmodule()` に似ていますが、返される辞書は、クラス名からクラス記述オブジェクトへの対応付けに加えて、トップレベル関数から関数記述オブジェクトへの対応付けも行っています。さらに、読み出し対象のモジュールがパッケージの場合、返される辞書はキー `'__path__'` を持ち、その値はパッケージの検索パスが入ったリストになります。

### 18.7.1 クラス記述オブジェクト

クラス記述オブジェクトは、`readmodule()` や `readmodule()_ex` が返す辞書の値として使われており、以下のデータメンバを提供しています。

**module**

クラス記述オブジェクトが記述している対象のクラスを定義しているモジュールの名前です。

**name**

クラスの名前です。

**super**

クラス記述オブジェクトが記述しようとしている対象クラスの、直接の基底クラス群について記述しているクラス記述オブジェクトのリストです。スーパークラスとして挙げられているが `readmodule()` が見つけられなかったクラスは、クラス記述オブジェクトではなくクラス名としてリストに挙げられます。

**methods**

メソッド名を行番号に対応付ける辞書です。

**file**

クラスを定義している `class` 文が入っているファイルの名前です。

**lineno**

`file` で指定されたファイル内にある `class` 文の数です。

### 18.7.2 関数記述オブジェクト (Function Descriptor Object)

`readmodule_ex()` の返す辞書内でキーに対応する値として使われている関数記述オブジェクトは、以下のデータメンバを提供しています:

**module**

関数記述オブジェクトが記述している対象の関数を定義しているモジュールの名前です。

**name**

関数の名前です。

**file**

関数を定義してる `def` 文が入っているファイルの名前です。

**lineno**

`file` で指定されたファイル内にある `def` 文の数です。

## 18.8 `py_compile` — Python ソースファイルのコンパイル

`py_compile` モジュールには、ソースファイルからバイトコードファイルを作る関数と、モジュールのソースファイルがスクリプトとして呼び出される時に使用される関数が定義されています。

頻繁に必要なわけではありませんが、共有ライブラリとしてモジュールをインストールする場合や、特にソースコードのあるディレクトリにバイトコードのキャッシュファイルを書き込む権限がないユーザがいるときには、この関数は役に立ちます。

**exception `PyCompileError`**

ファイルをコンパイル中にエラーが発生すると、`PyCompileError` 例外が送出されます。

**`compile(file[, cfile[, dfile[, doraise]]])`**

ソースファイルをバイトコードにコンパイルして、バイトコードのキャッシュファイルに書き出します。ソースコードはファイル名 `file` で渡します。バイトコードはファイル `cfile` に書き込まれ、デフォルトでは `file + 'c'` (使用しているインタプリタで最適化が可能なら `'o'`) です。もし `dfile` が指定されたら、`file` の代わりにソースファイルの名前としてエラーメッセージの中で使われます。`doraise = True` の場合、コンパイル中にエラーが発生すると `PyCompileError` を送出します。`doraise = False` の場合 (デフォルト) はエラーメッセージは `sys.stderr` に出力し、例外は送出しません。

**`main([args])`**

いくつか複数のソースファイルをコンパイルします。`args` で (あるいは `args` で指定されなかったらコマンドラインで) 指定されたファイルをコンパイルし、できたバイトコードを通常の方法で保存します。この関数はソースファイルの存在するディレクトリを検索しません; 指定されたファイルをコンパイルするだけです。

このモジュールがスクリプトとして実行されると、`main()` がコマンドラインで指定されたファイルを全てコンパイルします。

参考資料:

`compileall` モジュール (18.9 節):

ディレクトリツリー内の Python ソースファイルを全てコンパイルするライブラリ。

## 18.9 `compileall` — Python ライブラリをバイトコンパイル

このモジュールは、指定したディレクトリに含まれる Python ソースをコンパイルする関数を定義しています。Python ライブラリをインストールする時、ソースファイルを事前にコンパイルしておく事により、ライブラリのインストール先ディレクトリに書き込み権限をもたないユーザでもキャッシュされたバイトコードファイルを利用する事ができるようになります。

このモジュールのソースコードは、Python ソースファイルをコンパイルするスクリプトとしても利用する事ができます。コンパイルするディレクトリは、`sys.path` で指定されたディレクトリ、またはコマンドラインで指定されたディレクトリとなります。

**`compile_dir(dir[, maxlevels[, ddir[, force[, rx[, quiet]]]])`**

*dir* で指定されたディレクトリを再帰的に下降し、見つかった `.py` を全てコンパイルします。*maxlevels* は、下降する最大の深さ（デフォルトは 10）を指定します。*ddir* には、エラーメッセージで使われるファイル名の、親ディレクトリ名を指定する事ができます。*force* が真の場合、モジュールはファイルの更新日付に関わりなく再コンパイルされます。

*rx* には、検索対象から除外するファイル名の正規表現を指定します。絶対パス名をこの正規表現で `search` し、一致した場合にはコンパイル対象から除外します。

*quiet* が真の場合、通常処理では標準出力に何も表示しません。

`compile_path([skip_curdir[, maxlevels[, force]])`

`sys.path` に含まれる、全ての `.py` ファイルをバイトコンパイルします。*skip\_curdir* が真（デフォルト）の時、カレントディレクトリは検索されません。*maxlevels* と *force* はデフォルトでは 0 で、`compile_dir()` に渡されます。

参考資料:

`py_compile` モジュール (18.8 節):

Byte-compile a single source file.

## 18.10 dis — Python バイトコードの逆アセンブラ

`dis` モジュールは Python バイトコードを逆アセンブルしてバイトコードの解析を助けます。Python アセンブラがないため、このモジュールが Python アセンブリ言語を定義しています。このモジュールが入力として受け取る Python バイトコードはファイル `Include/opcode.h` に定義されており、コンパイラとインタプリタが使用しています。

例: 関数 `myfunc` を考えると:

```
def myfunc(alist):
    return len(alist)
```

次のコマンドを `myfunc()` の逆アセンブリを得るために使うことができます:

```
>>> dis.dis(myfunc)
2          0 LOAD_GLOBAL           0 (len)
          3 LOAD_FAST             0 (alist)
          6 CALL_FUNCTION         1
          9 RETURN_VALUE
10         10 LOAD_CONST          0 (None)
13         13 RETURN_VALUE
```

(“2” は行番号です)。

`dis` モジュールは次の関数と定数を定義します:

`dis([bytestr])`

*bytestr* オブジェクトを逆アセンブルします。*bytestr* はモジュール、クラス、関数、あるいはコードオブジェクトのいずれかを示します。モジュールに対しては、すべての関数を逆アセンブルします。クラスに対しては、すべてのメソッドを逆アセンブルします。単一のコードシーケンスに対しては、バイトコード命令ごとに一行をプリントします。オブジェクトが与えられない場合は、最後のトレースバックを逆アセンブルします。

`disb([tb])`

トレースバックのスタックの先頭の関数を逆アセンブルします。None が渡された場合は最後のトレー

スタックを使います。例外を引き起こした命令が表示されます。

`disassemble(code[, lasti])`

コードオブジェクトを逆アセンブルします。*lasti* が与えられた場合は、最後の命令を示します。出力は次のようなカラムに分割されます:

1. 各行の最初の命令に対する行番号。
2. 現在の命令。‘-->’ として示されます。
3. ラベル付けされた命令。‘>>’ とともに表示されます。
4. the address of the instruction,
5. 命令のアドレス。
6. 演算コード名。
7. 演算パラメータ。
8. 括弧の中のパラメータのインタプリテーション。

パラメータインタプリテーションはローカルおよびグローバル変数名、定数値、分岐目標、そして比較演算子を認識します。

`disco(code[, lasti])`

`disassemble` の別名。よりタイプしやすく、以前の Python リリースと互換性があります。

`opname`

演算名。一連のバイトコードを使ってインデキシングできます。

`cmp_op`

すべての比較演算名。

`hasconst`

定数腹メータを持つ一連のバイトコード。

`hasfree`

自由変数にアクセスする一連のバイトコード。

`hasname`

名前によって属性にアクセスする一連のバイトコード。

`hasjrel`

相対ジャンプターゲットをもつ一連のバイトコード。

`hasjabs`

絶対ジャンプターゲットをもつ一連のバイトコード。

`haslocal`

ローカル変数にアクセスする一連のバイトコード。

`hascompare`

ブール演算の一連のバイトコード。

### 18.10.1 Python バイトコード命令

現在 Python コンパイラは次のバイトコード命令を生成します。

`STOP_CODE`

コンパイラに end-of-code(コードの終わり) を知らせます。インタプリタでは使われません。

`POP_TOP`

top-of-stack (TOS)(スタックの先頭) の項目を取り除きます。

**ROT\_TWO**

スタックの先頭から二つの項目を入れ替えます。

**ROT\_THREE**

スタックの二番目と三番目の項目の位置を一つ上げ、先頭を三番目へ下げます。

**ROT\_FOUR**

スタックの二番目、三番目および四番目の位置を一つ上げ、先頭を四番目に下げます。

**DUP\_TOP**

スタックの先頭に参照の複製を作ります。

一項演算はスタックの先頭を取り出して演算を適用し、結果をスタックへプッシュし戻します。

**UNARY\_POSITIVE**

TOS = +TOS を実行します。

**UNARY\_NEGATIVE**

TOS = -TOS を実行します。

**UNARY\_NOT**

TOS = not TOS を実行します。

**UNARY\_CONVERT**

TOS = 'TOS' を実行します。

**UNARY\_INVERT**

TOS = ~TOS を実行します。

**GET\_ITER**

TOS = iter(TOS) を実行します。

二項演算はスタックからスタックの先頭 (TOS) と先頭から二番目のスタック項目を取り除きます。演算を実行し、スタックへ結果をプッシュし戻します。

**BINARY\_POWER**

TOS = TOS1 \*\* TOS を実行します。

**BINARY\_MULTIPLY**

TOS = TOS1 \* TOS を実行します。

**BINARY\_DIVIDE**

from \_\_future\_\_ import division が有効でないとき、TOS = TOS1 / TOS を実行します。

**BINARY\_FLOOR\_DIVIDE**

TOS = TOS1 // TOS を実行します。

**BINARY\_TRUE\_DIVIDE**

from \_\_future\_\_ import division が有効でないとき、TOS = TOS1 / TOS を実行します。

**BINARY\_MODULO**

TOS = TOS1 % TOS を実行します。

**BINARY\_ADD**

TOS = TOS1 + TOS を実行します。

**BINARY\_SUBTRACT**

TOS = TOS1 - TOS を実行します。

**BINARY\_SUBSCR**

TOS = TOS1[TOS] を実行します。

**BINARY\_LSHIFT**

TOS = TOS1 << TOS を実行します。



**BINARY\_RSHIFT**

TOS = TOS1 >> TOS を実行します。

**BINARY\_AND**

TOS = TOS1 & TOS を実行します。

**BINARY\_XOR**

TOS = TOS1 ^ TOS を実行します。

**BINARY\_OR**

TOS = TOS1 | TOS を実行します。

インプレース演算は TOS と TOS1 を取り除いて結果をスタックへプッシュするという点で二項演算と似ています。しかし、TOS1 がインプレース演算をサポートしている場合には演算が直接 TOS1 に行われます。また、演算結果の TOS は元の TOS1 と同じオブジェクトになることが多いですが、常に同じというわけではありません。

**INPLACE\_POWER**

インプレースに TOS = TOS1 \*\* TOS を実行します。

**INPLACE\_MULTIPLY**

インプレースに TOS = TOS1 \* TOS を実行します。

**INPLACE\_DIVIDE**

from \_\_future\_\_ import division が有効でないとき、インプレースに TOS = TOS1 / TOS を実行します。

**INPLACE\_FLOOR\_DIVIDE**

インプレースに TOS = TOS1 // TOS を実行します。

**INPLACE\_TRUE\_DIVIDE**

from \_\_future\_\_ import division が有効でないとき、インプレースに TOS = TOS1 / TOS を実行します。

**INPLACE\_MODULO**

インプレースに TOS = TOS1 % TOS を実行します。

**INPLACE\_ADD**

インプレースに TOS = TOS1 + TOS を実行します。

**INPLACE\_SUBTRACT**

インプレースに TOS = TOS1 - TOS を実行します。

**INPLACE\_LSHIFT**

インプレースに TOS = TOS1 << TOS を実行します。

**INPLACE\_RSHIFT**

インプレースに TOS = TOS1 >> TOS を実行します。

**INPLACE\_AND**

インプレースに TOS = TOS1 & TOS を実行します。

**INPLACE\_XOR**

インプレースに TOS = TOS1 ^ TOS を実行します。

**INPLACE\_OR**

インプレースに TOS = TOS1 | TOS を実行します。

スライス演算は三つまでのパラメータを取ります。

**SLICE+0**

TOS = TOS[:] を実行します。

**SLICE+1**

TOS = TOS1[TOS:] を実行します。

**SLICE+2**

TOS = TOS1[:TOS] を実行します。

**SLICE+3**

TOS = TOS2[TOS1:TOS] を実行します。

スライス代入はさらに別のパラメータを必要とします。どんな文もそうであるように、スタックに何もプッシュしません。

**STORE\_SLICE+0**

TOS[:] = TOS1 を実行します。

**STORE\_SLICE+1**

TOS1[TOS:] = TOS2 を実行します。

**STORE\_SLICE+2**

TOS1[:TOS] = TOS2 を実行します。

**STORE\_SLICE+3**

TOS2[TOS1:TOS] = TOS3 を実行します。

**DELETE\_SLICE+0**

del TOS[:] を実行します。

**DELETE\_SLICE+1**

del TOS1[TOS:] を実行します。

**DELETE\_SLICE+2**

del TOS1[:TOS] を実行します。

**DELETE\_SLICE+3**

del TOS2[TOS1:TOS] を実行します。

**STORE\_SUBSCR**

TOS1[TOS] = TOS2 を実行します。

**DELETE\_SUBSCR**

del TOS1[TOS] を実行します。

その他の演算。

**PRINT\_EXPR**

対話モードのための式文を実行します。TOS はスタックから取り除かれプリントされます。非対話モードにおいては、式文は POP\_STACK で終了しています。

**PRINT\_ITEM**

sys.stdout に束縛されたファイル互換のオブジェクトへ TOS をプリントします。print 文に、各項目に対するこのような命令が一つあります。

**PRINT\_ITEM\_TO**

PRINT\_ITEM と似ていますが、TOS から二番目の項目を TOS にあるファイル互換オブジェクトへプリントします。これは拡張 print 文で使われます。

**PRINT\_NEWLINE**

sys.stdout へ改行をプリントします。これは print 文がコンマで終わっていない場合に print 文の最後の演算として生成されます。

**PRINT\_NEWLINE\_TO**

PRINT\_NEWLINE と似ていますが、TOS のファイル互換オブジェクトに改行をプリントします。こ

れは拡張 print 文で使われます。

#### **BREAK\_LOOP**

break 文があるためループを終了します。

#### **CONTINUE\_LOOP** *target*

continue 文があるためループを継続します。*target* はジャンプするアドレスです (アドレスは FOR\_ITER 命令であるべきです)。

#### **LOAD\_LOCALS**

現在のスコープのローカルな名前空間 (locals) への参照をスタックにプッシュします。これはクラス定義のためのコードで使われます: クラス本体が評価された後、locals はクラス定義へ渡されます。

#### **RETURN\_VALUE**

関数の呼び出し元へ TOS を返します。

#### **YIELD\_VALUE**

TOS をポップし、それをジェネレータから yield します。

#### **IMPORT\_STAR**

‘\_’ で始まっていないすべてのシンボルをモジュール TOS から直接ローカル名前空間へロードします。モジュールはすべての名前をロードした後にポップされます。この演算コードは from module import \*を実行します。

#### **EXEC\_STMT**

exec TOS2,TOS1,TOS を実行します。コンパイラは見つからないオプションのパラメータを None で埋めます。

#### **POP\_BLOCK**

ブロックスタックからブロックを一つ取り除きます。フレームごとにブロックのスタックがあり、ネストしたループ、try 文などを意味しています。

#### **END\_FINALLY**

finally 節を終わらせます。インタプリタは例外を再び発生させなければならないかどうか、あるいは、関数が返り外側の次のブロックに続くかどうかを思い出します。

#### **BUILD\_CLASS**

新しいクラスオブジェクトを作成します。TOS はメソッド辞書、TOS1 は基底クラスの名前のタプル、TOS2 はクラス名です。

次の演算コードはすべて引数を要求します。引数はより重要なバイトを下位にもつ 2 バイトです。

#### **STORE\_NAME** *namei*

name = TOS を実行します。*namei* はコードオブジェクトの属性 co\_names における *name* のインデックスです。コンパイラは可能ならば STORE\_LOCAL または STORE\_GLOBAL を使おうとします。

#### **DELETE\_NAME** *namei*

del name を実行します。ここで、*namei* はコードオブジェクトの co\_names 属性へのインデックスです。

#### **UNPACK\_SEQUENCE** *count*

TOS を *count* 個のへ個別の値に分け、右から左にスタックに置かれます。

#### **DUP\_TOPX** *count*

*count* 個の項目を同じ順番を保ちながら複製します。実装上の制限から、*count* は 1 から 5 の間 (5 を含む) でなければいけません。

#### **STORE\_ATTR** *namei*

TOS.name = TOS1 を実行します。ここで、*namei* は co\_names における名前のインデックスです。

**DELETE\_ATTR** *namei*

*co\_names* へのインデックスとして *namei* を使い、`del TOS.name` を実行します。

**STORE\_GLOBAL** *namei*

*STORE\_NAME* として機能しますが、グローバルとして名前を記憶します。

**DELETE\_GLOBAL** *namei*

*DELETE\_NAME* として機能しますが、グローバル名を削除します。

**LOAD\_CONST** *consti*

‘*co\_consts[consti]*’ をスタックにプッシュします。

**LOAD\_NAME** *namei*

‘*co\_names[namei]*’ に関連付けられた値をスタックにプッシュします。

**BUILD\_TUPLE** *count*

スタックから *count* 個の項目を消費するタプルを作り出し、できたタプルをスタックにプッシュします。

**BUILD\_LIST** *count*

*BUILD\_TUPLE* として機能しますが、リストを作り出します。

**BUILD\_MAP** *zero*

スタックに新しい空の辞書オブジェクトをプッシュします。引数は無視され、コンパイラによってゼロに設定されます。

**LOAD\_ATTR** *namei*

TOS を `getattr(TOS, co_names[namei])` と入れ替えます。

**COMPARE\_OP** *opname*

ブール演算を実行します。演算名は *cmp\_op[opname]* にあります。

**IMPORT\_NAME** *namei*

モジュール *co\_names[namei]* をインポートします。モジュールオブジェクトはスタックへプッシュされます。現在の名前空間は影響されません: 適切な `import` 文に対して、それに続く *STORE\_FAST* 命令が名前空間を変更します。

**IMPORT\_FROM** *namei*

属性 *co\_names[namei]* を TOS に見つかるモジュールからロードします。作成されたオブジェクトはスタックにプッシュされ、その後 *STORE\_FAST* 命令によって記憶されます。

**JUMP\_FORWARD** *delta*

バイトコードカウンタを *delta* だけ増加させます。

**JUMP\_IF\_TRUE** *delta*

TOS が真ならば、*delta* だけバイトコードカウンタを増加させます。TOS はスタックに残されます。

**JUMP\_IF\_FALSE** *delta*

TOS が偽ならば、*delta* だけバイトコードカウンタを増加させます。TOS は変更されません。

**JUMP\_ABSOLUTE** *target*

バイトコードカウンタを *target* に設定します。

**FOR\_ITER** *delta*

TOS はイテレータです。その `next()` メソッドを呼び出します。これが新しい値を作り出すならば、それを(その下にイテレータを残したまま)スタックにプッシュします。イテレータが尽きたことを示した場合は、TOS がポップされます。そして、バイトコードカウンタが *delta* だけ増やされます。

**LOAD\_GLOBAL** *namei*

グローバル名 *co\_names[namei]* をスタック上にロードします。

**SETUP\_LOOP** *delta*

ブロックスタックにループのためのブロックをプッシュします。ブロックは現在の命令から *delta* バイトの大きさを占めます。

**SETUP\_EXCEPT** *delta*

try-except 節から try ブロックをブロックスタックにプッシュします。*delta* は最初の except ブロックを指します。

**SETUP\_FINALLY** *delta*

try-except 節から try ブロックをブロックスタックにプッシュします。*delta* は finally ブロックを指します。

**LOAD\_FAST** *var\_num*

ローカルな `co_varnames[var_num]` への参照をスタックにプッシュします。

**STORE\_FAST** *var\_num*

TOS をローカルな `co_varnames[var_num]` の中に保存します。

**DELETE\_FAST** *var\_num*

ローカルな `co_varnames[var_num]` を削除します。

**LOAD\_CLOSURE** *i*

セルと自由変数記憶領域のスロット *i* に含まれるセルへの参照をプッシュします。*i* が `co_cellvars` の長さより小さければ、変数の名前は `co_cellvars[i]` です。そうでなければ、それは `co_freevars[i - len(co_cellvars)]` です。

**LOAD\_DEREF** *i*

セルと自由変数記憶領域のスロット *i* に含まれるセルをロードします。セルが持つオブジェクトへの参照をスタックにプッシュします。

**STORE\_DEREF** *i*

セルと自由変数記憶領域のスロット *i* に含まれるセルへ TOS を保存します。

**SET\_LINENO** *lineno*

このオペコードは廃止されました。

**RAISE\_VARARGS** *argc*

例外を発生させます。*argc* は raise 文へ与えるパラメータの数を 0 から 3 の範囲で示します。ハンドラは TOS2 としてトレースバック、TOS1 としてパラメータ、そして TOS として例外を見つけられます。

**CALL\_FUNCTION** *argc*

関数を呼び出します。*argc* の低位バイトは位置パラメータを示し、高位バイトはキーワードパラメータの数を示します。オペコードは最初にキーワードパラメータをスタック上に見つけます。それぞれのキーワード引数に対して、その値はキーの上にあります。スタック上のキーワードパラメータの下に位置パラメータはあり、先頭に最も右のパラメータがあります。スタック上のパラメータの下には、呼び出す関数オブジェクトがあります。

**MAKE\_FUNCTION** *argc*

新しい関数オブジェクトをスタックにプッシュします。TOS は関数に関連付けられたコードです。関数オブジェクトは TOS の下にある *argc* デフォルトパラメータをもつように定義されます。

**MAKE\_CLOSURE** *argc*

新しい関数オブジェクトを作り出し、その *func\_closure* スロットを設定し、それをスタックにプッシュします。TOS は関数に関連付けられたコードです。コードオブジェクトが *N* 個の自由変数を持っているならば、スタック上の次の *N* 個の項目はこれらの変数に対するセルです。関数はセルの前にある *argc* デフォルトパラメータも持っています。

**BUILD\_SLICE** *argc*

スライスオブジェクトをスタックにプッシュします。*argc* は 2 あるいは 3 でなければなりません。2 ならば `slice(TOS1, TOS)` がプッシュされます。3 ならば `slice(TOS2, TOS1, TOS)` がプッシュされます。これ以上の情報については、`slice()` 組み込み関数を参照してください。

#### EXTENDED\_ARG *ext*

大きすぎてデフォルトの二バイトに当てはめることができない引数をもつあらゆるオペコードの前に置かれます。*ext* は二つの追加バイトを保持し、その後のオペコードの引数と一緒に取られます。それらは四バイト引数を構成し、*ext* はその最上位バイトです。

#### CALL\_FUNCTION\_VAR *argc*

関数を呼び出します。*argc* は `CALL_FUNCTION` のように解釈実行されます。スタックの先頭の要素は変数引数リストを含んでおり、その後にキーワードと位置引数が続きます。

#### CALL\_FUNCTION\_KW *argc*

関数を呼び出します。*argc* は `CALL_FUNCTION` のように解釈実行されます。スタックの先頭の要素はキーワード引数辞書を含んでおり、その後に明示的なキーワードと位置引数が続きます。

#### CALL\_FUNCTION\_VAR\_KW *argc*

関数を呼び出します。*argc* は `CALL_FUNCTION` のように解釈実行されます。スタックの先頭の要素はキーワード引数辞書を含んでおり、その後に変数引数のタプルが続き、さらに明示的なキーワードと位置引数が続きます。

## 18.11 distutils — Python モジュールの構築とインストール

`distutils` パッケージは、現在インストールされている Python に追加するためのモジュール構築、および実際のインストールを支援します。新規のモジュールは 100%-pure Python でも、C で書かれた拡張モジュールでも、あるいは Python と C 両方のコードが入っているモジュールからなる Python パッケージでもかまいません。

このパッケージは、Python ドキュメンテーション パッケージに含まれているこれとは別の 2 つのドキュメントで詳しく説明されています。`distutils` の機能を使って新しいモジュールを配布する方法は、*Python* モジュールを配布する に書かれています。Python モジュールをインストールする方法は、モジュールの作者が `distutils` パッケージを使っている場合でもない場合でも、*Python* モジュールをインストールする に書かれています。

参考資料:

*Python* モジュールを配布する

([../dist/dist.html](#))

このマニュアルは Python モジュールの開発者およびパッケージ担当に向けたものです。ここでは、現在インストールされている Python に簡単に追加できる `distutils` ベースのパッケージをどうやって用意するかについて説明しています。

*Python* モジュールをインストールする

([../inst/inst.html](#))

現在インストールされている Python にモジュールを追加するための情報が書かれた“管理者”向けのマニュアルです。この文書を読むのに Python プログラマである必要はありません。



# Python コンパイラパッケージ

Python compiler パッケージは Python のソースコードを分析したり Python バイトコードを生成するためのツールです。compiler は Python のソースコードから抽象的な構文木を生成し、その構文木から Python バイトコードを生成するライブラリをそなえています。

compiler パッケージは、Python で書かれた Python ソースコードからバイトコードへの変換プログラムです。これは組み込みの構文解析器をつかい、そこで得られた具体的な構文木に対して標準的な parser モジュールを使用します。この構文木から抽象構文木 AST (Abstract Syntax Tree) が生成され、その後 Python バイトコードが得られます。

このパッケージの機能は、Python インタプリタに内蔵されている組み込みのコンパイラがすべて含んでいるものです。これはその機能と正確に同じものになるよう意図してつくられています。なぜ同じことをするコンパイラをもうひとつ作る必要があるのでしょうか？ このパッケージはいろいろな目的に使うことができるからです。これは組み込みのコンパイラよりも簡単に変更できますし、これが生成する AST は Python ソースコードを解析するのに有用です。

この章では compiler パッケージのいろいろなコンポーネントがどのように動作するのかを説明します。そのため説明はリファレンスマニュアル的なものと、チュートリアル的な要素がまざったものになっています。

以下のモジュールは compiler パッケージの一部です：

## 19.1 基本的なインターフェイス

このパッケージのトップレベルでは 4 つの関数が定義されています。compiler モジュールを import すると、これらの関数およびこのパッケージに含まれている一連のモジュールが使用可能になります。

**parse**(*buf*)

*buf* 中の Python ソースコードから得られた抽象構文木 AST を返します。ソースコード中にエラーがある場合、この関数は `SyntaxError` を発生させます。返り値は `compiler.ast.Module` インスタンスであり、この中に構文木が格納されています。

**parseFile**(*path*)

*path* で指定されたファイル中の Python ソースコードから得られた抽象構文木 AST を返します。これは `parse(open(path).read())` と等価な働きをします。

**walk**(*ast*, *visitor*[, *verbose*])

*ast* に格納された抽象構文木の各ノードを先行順序 (pre-order) でたどっていきます。各ノードごとに *visitor* インスタンスの該当するメソッドが呼ばれます。

**compile**(*source*, *filename*, *mode*, *flags=None*, *dont\_inherit=None*)

文字列 *source*、Python モジュール、文あるいは式を `exec` 文あるいは `eval()` 関数で実行可能なバイトコードオブジェクトにコンパイルします。この関数は組み込みの `compile()` 関数を置き換える

ものです。

*filename* は実行時のエラーメッセージに使用されます。

*mode* は、モジュールをコンパイルする場合は `'exec'`、(対話的に実行される) 単一の文をコンパイルする場合は `'single'`、式をコンパイルする場合には `'eval'` を渡します。

引数 *flags* および *dont\_inherit* は将来的に使用される文に影響しますが、いまのところはサポートされていません。

`compileFile(source)`

ファイル *source* をコンパイルし、`.pyc` ファイルを生成します。

`compiler` パッケージは以下のモジュールを含んでいます: `ast`、`consts`、`future`、`misc`、`pyassem`、`pycodegen`、`symbols`、`transformer`、そして `visitor`。

## 19.2 制限

`compiler` パッケージにはエラーチェックにいくつか問題が存在します。構文エラーはインタプリタの2つの別々のフェーズによって認識されます。ひとつはインタプリタのパーザによって認識されるもので、もうひとつはコンパイラによって認識されるものです。`compiler` パッケージはインタプリタのパーザに依存しているので、最初の段階のエラーチェックは労せずして実現できています。しかしその次の段階は、実装されてはいますが、その実装は不完全です。たとえば `compiler` パッケージは引数に同じ名前が2度以上出てきてもエラーを出しません: `def f(x, x): ...`

`compiler` の将来のバージョンでは、これらの問題は修正される予定です。

## 19.3 Python 抽象構文

`compiler.ast` モジュールは Python の抽象構文木 AST を定義します。AST では各ノードがそれぞれの構文要素をあらわします。木の根は `Module` オブジェクトです。

抽象構文木 AST は、パーズされた Python ソースコードに対する高水準のインターフェイスを提供します。Python インタプリタにおける `parser` モジュールとコンパイラは C で書かれおり、具体的な構文木を使っています。具体的な構文木は Python のパーザ中で使われている構文と密接に関連しています。ひとつの要素に単一のノードを割り当てる代わりに、ここでは Python の優先順位に従って、何層にもわたるネストしたノードがしばしば使われています。

抽象構文木 AST は、`compiler.transformer` (変換器) モジュールによって生成されます。`transformer` は組み込みの Python パーザに依存しており、これを使って具体的な構文木をまず生成します。つぎにそこから抽象構文木 AST を生成します。

`transformer` モジュールは、実験的な Python-to-C コンパイラ用に Greg Stein と Bill Tutt によって作られました。現行のバージョンではいくつかの修正と改良がなされていますが、抽象構文木 AST と `transformer` の基本的な構造は Stein と Tutt によるものです。

### 19.3.1 AST ノード

`compiler.ast` モジュールは、各ノードのタイプとその要素を記述したテキストファイルからつくられます。各ノードのタイプはクラスとして表現され、そのクラスは抽象基底クラス `compiler.ast.Node` を継承し子ノードの名前属性を定義しています。

`class Node()`

`Node` インスタンスはパーザジェネレータによって自動的に作成されます。ある特定の `Node` インス

タンスに対する推奨されるインターフェイスとは、子ノードにアクセスするために public な (訳注: 公開された) 属性を使うことです。public な属性は単一のノード、あるいは一連のノードのシーケンスに束縛されている (訳注: バインドされている) かもしれませんが、これは Node のタイプによって違います。たとえば Class ノードの bases 属性は基底クラスのノードのリストに束縛されており、doc 属性は単一のノードのみに束縛されている、といった具合です。

各 Node インスタンスは lineno 属性をもっており、これは None かもしれません。XXX どういったノードが使用可能な lineno をもっているかの規則は定かではない。

Node オブジェクトはすべて以下のメソッドをもっています:

`getChildren()`

子ノードと子オブジェクトを、これらが出てきた順で、平らなリスト形式にして返します。とくにノードの順序は、Python 文法中に現れるものと同じになっています。すべての子が Node インスタンスなわけではありません。たとえば関数名やクラス名といったものは、ただの文字列として表されます。

`getChildNodes()`

子ノードをこれらが出てきた順で平らなリスト形式にして返します。このメソッドは `getChildren()` に似ていますが、Node インスタンスしか返さないという点で異なります。

Node クラスの一般的な構造を説明するため、以下に 2 つの例を示します。while 文は以下のような文法規則により定義されています:

```
while_stmt:      "while" expression ":" suite
               ["else" ":" suite]
```

While ノードは 3 つの属性をもっています: test、body、および else\_ です。(ある属性にふさわしい名前が Python の予約語としてすでに使われているとき、その名前を属性名にすることはできません。そのため、ここでは名前が正規のものとして受けつけられるようにアンダースコアを後につけてあります、そのため else\_ は else のかわりです。)

if 文はもっとこみ入っています。なぜならこれはいくつもの条件判定を含む可能性があるからです。

```
if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
```

If ノードでは、tests および else\_ の 2 つだけの属性が定義されています。tests 属性には条件式とその後の動作のタプルがリスト形式で入っています。おのおのの if/elif 節ごとに 1 タプルです。各タプルの最初の要素は条件式で、2 番目の要素はもしその式が真ならば実行されるコードをふくんだ Stmt ノードになっています。

If の `getChildren()` メソッドは、子ノードの平らなリストを返します。if/elif 節が 3 つあって else 節がない場合なら、`getChildren()` は 6 要素のリストを返すでしょう: 最初の条件式、最初の Stmt、2 番目の条件式...といった具合です。

以下の表は `compiler.ast` で定義されている Node サブクラスと、それらのインスタンスに対して使用可能なパブリックな属性です。ほとんどの属性の値じたいは Node インスタンスか、インスタンスのリストです。この値がインスタンス型以外の場合、その型は備考の中で記されています。これら属性の順序は、`getChildren()` および `getChildNodes()` が返す順です。

ノードの型	属性	値
Add	left	左側の項
	right	右側の項

ノードの型	属性	値
And	nodes	項のリスト
AssAttr		代入先をあらわす属性
	expr	ドット (.) の左側の式
	attrname	属性名をあらわす文字列
	flags	XXX
AssList	nodes	代入先のリスト要素のリスト
AssName	name	代入先の名前
	flags	XXX
AssTuple	nodes	代入先のタプル要素のリスト
Assert	test	検査される条件式
	fail	AssertionError の値
Assign	nodes	代入先のリスト、代入記号 (=) ごとにひとつ
	expr	代入する値
AugAssign	node	
	op	
	expr	
Backquote	expr	
Bitand	nodes	
Bitor	nodes	
Bitxor	nodes	
Break		
CallFunc	node	呼ばれる側をあらわす式
	args	引数のリスト
	star_args	*-arg 拡張引数の値
	dstar_args	**arg 拡張引数の値
Class	name	クラス名をあらわす文字列
	bases	基底クラスのリスト
	doc	doc string、文字列あるいは None
	code	クラス文の本体
Compare	expr	
	ops	
Const	value	
Continue		
Dict	items	
Discard	expr	
Div	left	
	right	
Ellipsis		
Exec	expr	
	locals	
	globals	
For	assign	
	list	
	body	
	else_	

ノードの型	属性	値
From	modname names	
Function	name argnames defaults flags doc code	def で定義される名前をあらわす文字列 引数をあらわす文字列のリスト デフォルト値のリスト xxx doc string、文字列あるいは None 関数の本体
Getattr	expr attrname	
Global	names	
If	tests else_	
Import	names	
Invert	expr	
Keyword	name expr	
Lambda	argnames defaults flags code	
LeftShift	left right	
List	nodes	
ListComp	expr quals	
ListCompFor	assign list ifs	
ListCompIf	test	
Mod	left right	
Module	doc node	doc string、文字列あるいは None モジュール本体、Stmt インスタンス
Mul	left right	
Name	name	
Not	expr	
Or	nodes	
Pass		
Power	left right	
Print	nodes dest	
Printnl	nodes	

ノードの型	属性	値
	dest	
Raise	expr1 expr2 expr3	
Return	value	
RightShift	left right	
Slice	expr flags lower upper	
Sliceobj	nodes	文のリスト
Stmt	nodes	
Sub	left right	
Subscript	expr flags subs	
TryExcept	body handlers else_	
TryFinally	body final	
Tuple	nodes	
UnaryAdd	expr	
UnarySub	expr	
While	test body else_	
Yield	value	

### 19.3.2 代入ノード

代入をあらわすのに使われる一群のノードが存在します。ソースコードにおけるそれぞれの代入文は、抽象構文木 AST では単一のノード `Assign` になっています。nodes 属性は各代入の対象にたいするノードのリストです。これが必要なのは、たとえば `a = b = 2` のように代入が連鎖的に起こるためです。このリスト中における各 Node は、次のうちどれかのクラスになります: `AssAttr`、`AssList`、`AssName`、または `AssTuple`。

代入対象の各ノードには代入されるオブジェクトの種類が記録されています。`AssName` は `a = 1` などの単純な変数名、`AssAttr` は `a.x = 1` などの属性に対する代入、`AssList` および `AssTuple` はそれぞれ、`a`、`b`、`c = a_tuple` などのようなリストとタプルの展開をあらわします。

代入対象ノードはまた、そのノードが代入で使われるのか、それとも `del` 文で使われるのかをあらわす属性 `flags` も持っています。`AssName` は `del x` などのような `del` 文をあらわすのにも使われます。



ある式がいくつかの属性への参照をふくんでいるときは、代入あるいは `del` 文はただひとつだけの `AssAttr` ノードをもちます。最終的な属性への参照としてです。それ以外の属性への参照は `AssAttr` インスタンスの `expr` 属性にある `Getattr` ノードによってあらわされます。

### 19.3.3 サンプル

この節では、Python ソースコードに対する抽象構文木 AST のかんたんな例をいくつかご紹介します。これらの例では `parse()` 関数をどうやって使うか、AST の `repr` 表現はどんなふうになっているか、そしてある AST ノードの属性にアクセスするにはどうするかを説明します。

最初のモジュールでは単一の関数を定義しています。かりにこれは `'/tmp/doublelib.py'` に格納されていると仮定しましょう。

```
"""This is an example module.

This is the docstring.
"""

def double(x):
    "Return twice the argument"
    return x * 2
```

以下の対話的インタプリタのセッションでは、見やすさのため長い AST の `repr` を整形しなおしてあります。AST の `repr` では `qualify` されていないクラス名が使われています。 `repr` 表現からインスタンスを作成したい場合は、 `compiler.ast` モジュールからそれらのクラス名を `import` しなければなりません。

```
>>> import compiler
>>> mod = compiler.parseFile("/tmp/doublelib.py")
>>> mod
Module('This is an example module.\n\nThis is the docstring.\n',
      Stmt([Function('double', ['x'], [], 0, 'Return twice the argument',
                    Stmt([Return(Mul((Name('x'), Const(2))))]))]))
>>> from compiler.ast import *
>>> Module('This is an example module.\n\nThis is the docstring.\n',
...      Stmt([Function('double', ['x'], [], 0, 'Return twice the argument',
...                    Stmt([Return(Mul((Name('x'), Const(2))))]))]))
Module('This is an example module.\n\nThis is the docstring.\n',
      Stmt([Function('double', ['x'], [], 0, 'Return twice the argument',
                    Stmt([Return(Mul((Name('x'), Const(2))))]))]))
>>> mod.doc
'This is an example module.\n\nThis is the docstring.\n'
>>> for node in mod.node.nodes:
...     print node
...
Function('double', ['x'], [], 0, 'Return twice the argument',
      Stmt([Return(Mul((Name('x'), Const(2))))]))
>>> func = mod.node.nodes[0]
>>> func.code
Stmt([Return(Mul((Name('x'), Const(2))))])
```

## 19.4 Visitor を使って AST をわたり歩く

visitor パターンは ... `compiler` パッケージは、Python のイントロスペクション機能を利用して visitor のために必要な大部分のインフラを省略した、visitor パターンの変種を使っています。

visit されるクラスは、visitor を受け入れるようにプログラムされている必要はありません。visitor が必要なのはただそれがとくに興味あるクラスに対して visit メソッドを定義することだけです。それ以外はデフォルトの visit メソッドが処理します。

XXX The magic visit() method for visitors.

```
walk(tree, visitor[, verbose])
```

```
class ASTVisitor()
```

ASTVisitor は構文木を正しい順序でわたり歩くようにします。それぞれのノードはまず preorder() の呼び出しではじまります。各ノードに対して、これは 'visitNodeType' という名前のメソッドに対する preorder() 関数への visitor 引数をチェックします。ここで NodeType の部分はそのノードのクラス名です。たとえば While ノードなら、visitWhile() が呼ばれるわけです。もしそのメソッドが存在している場合、それはそのノードを第一引数として呼び出されます。

ある特定のノード型に対する visitor メソッドでは、その子ノードをどのようにわたり歩くかが制御できます。ASTVisitor は visitor に visit メソッドを追加することで、その visitor 引数を修正します。特定のノード型に対する visitor が存在しない場合、default() メソッドが呼び出されます。

ASTVisitor オブジェクトには以下のようなメソッドがあります:

XXX 追加の引数を記述

```
default(node[, ...])
```

```
dispatch(node[, ...])
```

```
preorder(tree, visitor)
```

## 19.5 バイトコード生成

バイトコード生成器はバイトコードを出力する visitor です。visit メソッドが呼ばれるたびにこれは emit() メソッドを呼び出し、バイトコードを出力します。基本的なバイトコード生成器はモジュール、クラス、および関数によって拡張できます。アセンブラがこれらの出力された命令を低レベルのバイトコードに変換します。これはコードオブジェクトからなる定数のリスト生成や、分岐のオフセット計算といった処理をおこないます。

# SGI IRIX 特有のサービス

この章で記述されているモジュールは、SGI の IRIX オペレーティングシステム (バージョン 4 と 5) 特有の機能へのインターフェイスを提供します。

<b>al</b>	SGI のオーディオ機能。
<b>AL</b>	al モジュールで使われる定数。
<b>cd</b>	Silicon Graphics システムの CD-ROM へのインターフェイス
<b>fl</b>	グラフィカルユーザーインターフェイスのための FORMS ライブラリ。
<b>FL</b>	fl モジュールで使用される定数。
<b>flp</b>	保存された FORMS デザインをロードする関数。
<b>fm</b>	SGI ワークステーションの <i>Font Manager</i> インターフェイス。
<b>gl</b>	Silicon Graphics の <i>Graphics Library</i> の関数。
<b>DEVICE</b>	gl モジュールで使われる定数。
<b>GL</b>	gl モジュールで使われる定数。
<b>imgfile</b>	SGI imglib ファイルのサポート。
<b>jpeg</b>	JPEG ファイルの読み書きを行います。

## 20.1 al — SGI のオーディオ機能

このモジュールを使うと、SGI Indy と Indigo ワークステーションのオーディオ装置にアクセスできます。詳しくは IRIX の man ページのセクション 3A を参照してください。ここに書かれた関数が何をするかを理解するには、man ページを読む必要があります！IRIX のリリース 4.0.5 より前のものでは使えない関数もあります。お使いのプラットフォームで特定の関数が使えるかどうか、マニュアルで確認してください。

このモジュールで定義された関数とメソッドは全て、名前に 'AL' の接頭辞を付けた C の関数と同義です。

C のヘッダーファイル `<audio.h>` のシンボル定数は標準モジュール `AL` に定義されています。下記を参照してください。

**警告：**オーディオライブラリの現在のバージョンは、不正な引数が渡されるとエラーステータスが返るのではなく、core を吐き出すことがあります。残念ながら、この現象が確実に起こる環境は述べられていないし、確認することは難しいので、Python インターフェイスでこの種の問題に対して防御することはできません。(一つの例は過大なキューサイズを特定することです — 上限については記載されていません。)

このモジュールには、以下の関数が定義されています：

`openport (name, direction[, config])`

引数 *name* と *direction* は文字列です。省略可能な引数 *config* は、`newconfig()` で返されるコンフィギュレーションオブジェクトです。返り値は *audio port object* です；オーディオポートオブジェクトのメソッドは下に書かれています。

`newconfig()`

返り値は新しい *audio configuration object* です；オーディオコンフィギュレーションオブジェクトのメソッドは下に書かれています。

`queryparams(device)`

引数 *device* は整数です。返り値は `ALqueryparams()` で返されるデータを含む整数のリストです。

`getparams(device, list)`

引数 *device* は整数です。引数 *list* は `queryparams()` で返されるようなリストです; `queryparams()` を適切に (!) 修正して使うことができます。

`setparams(device, list)`

引数 *device* は整数です。引数 *list* は `queryparams()` で返されるようなリストです。

### 20.1.1 コンフィギュレーションオブジェクト

`newconfig()` で返されるコンフィギュレーションオブジェクトには以下のメソッドがあります：

`getqueuesize()`

キューサイズを返します。

`setqueuesize(size)`

キューサイズを設定します。

`getwidth()`

サンプルサイズを返します。

`setwidth(width)`

サンプルサイズを設定します。

`getchannels()`

チャンネル数を返します。

`setchannels(nchannels)`

チャンネル数を設定します。

`getsampfmt()`

サンプルのフォーマットを返します。

`setsampfmt(sampfmt)`

サンプルのフォーマットを設定します。

`getfloatmax()`

浮動小数点数でサンプルデータの最大値を返します。

`setfloatmax(floatmax)`

浮動小数点数でサンプルデータの最大値を設定します。

### 20.1.2 ポートオブジェクト

`openport()` で返されるポートオブジェクトには以下のメソッドがあります：

`closeport()`

ポートを閉じます。

`getfd()`

ファイルディスクリプタを整数で返します。

`getfilled()`

バッファに存在するサンプルの数を返します。

`getfillable()`

バッファの空きに入れることのできるサンプルの数を返します。

`readsamps(nsamples)`

必要ならブロックして、キューから指定のサンプル数を読み込みます。生データを文字列として（例えば、サンプルサイズが 2 バイトならサンプル当たり 2 バイトが big-endian（high byte、low byte）で）返します。

`writesamps(samples)`

必要ならブロックして、キューにサンプルを書き込みます。サンプルは `readsamps()` で返される値のようにエンコードされていなければなりません。

`getfillpoint()`

‘fill point’ を返します。

`setfillpoint(fillpoint)`

‘fill point’ を設定します。

`getconfig()`

現在のポートのコンフィギュレーションを含んだコンフィギュレーションオブジェクトを返します。

`setconfig(config)`

コンフィギュレーションを引数に取り、そのコンフィギュレーションに設定します。

`getstatus(list)`

最後のエラーについてのステータスの情報を返します。

## 20.2 AL — al モジュールで使われる定数

このモジュールには、組み込みモジュール `al`（上記参照）を使用するのに必要とされるシンボリック定数が定義されています。定数の名前は C の include ファイル `<audioio.h>` で接頭辞 ‘AL\_’ を除いたものと同じです。

定義されている名前の完全なリストについてはモジュールのソースを参照してください。お勧めの使い方は以下の通りです：

```
import al
from AL import *
```

## 20.3 cd — SGI システムの CD-ROM へのアクセス

このモジュールは Silicon Graphics CD ライブラリへのインターフェースを提供します。Silicon Graphics システムだけで利用可能です。

ライブラリは以下のように使われます。

CD-ROM デバイスを `open()` で開き、`createparser()` で CD からデータをパースするためのパーザを作ります。`open()` で返されるオブジェクトは CD からデータを読み込むのに使われますが、CD-ROM デバイスのステータス情報や、CD の情報、たとえば目次などを得るのにも使われます。CD から得たデータはパーザに渡され、パーザはフレームをパースし、あらかじめ加えられたコールバック関数を呼び出します。

オーディオ CD はトラック *tracks* あるいはプログラム *programs*（同じ意味で、どちらかの用語が使われます）に分けられます。トラックはさらにインデックス *indices* に分けられます。オーディオ CD は、CD 上の各トラックのスタート位置を示す目次 *table of contents* を持っています。インデックス 0 は普通、トラックの始まりの前のポーズです。目次から得られるトラックのスタート位置は通常、インデックス 1 のスタート位置です。

CD 上の位置は 2 通りの方法で得ることができます。それはフレームナンバーと、分、秒、フレームの 3 つの値からなるタプルの 2 つです。ほとんどの関数は後者を使います。位置は CD の開始位置とトラックの開始位置の両方に相対的になります。

モジュール `cd` は、以下の関数と定数を定義しています：

**createparser()**

不透明なパーザオブジェクトを作って返します。パーザオブジェクトのメソッドは下に記載されています。

**msftoframe(*minutes, seconds, frames*)**

絶対的なタイムコードである (*minutes, seconds, frames*) の 3 つ組の表現を、相当する CD のフレームナンバーに変換します。

**open([*device* [, *mode* ]])**

CD-ROM デバイスを開きます。不透明なプレーヤーオブジェクトを返します；プレーヤーオブジェクトのメソッドは下に記載されています。デバイス *device* は SCSI デバイスファイルの名前で、例えば `"/dev/scsi/sc0d410"` あるいは `None` です。もし省略したり、`None` なら、ハードウェアが検索されて CD-ROM デバイスを割り当てます。*mode* は、省略しないなら `'r'` にすべきです。

このモジュールでは以下の変数を定義しています：

**exception error**

様々なエラーについて発生する例外です。

**DATASIZE**

オーディオデータの 1 フレームのサイズです。これは `audio` タイプのコールバックへ渡されるオーディオデータのサイズです。

**BLOCKSIZE**

オーディオデータが読み取られていないフレーム 1 つのサイズです。

以下の変数は `getstatus()` で返されるステータス情報です：

**READY**

オーディオ CD がロードされて、ドライブが操作可能であることを示します。

**NODISC**

ドライブに CD がロードされていないことを示します。

**CDROM**

ドライブに CD-ROM がロードされていることを示します。続いて `play` あるいは `read` の操作をする  
と、I/O エラーを返します。

**ERROR**

ディスクや目次を読み込もうとしているときに起こるエラー。

**PLAYING**

ドライブがオーディオ CD を CD プレーヤーモードでオーディオ端子から再生していることを示します。

**PAUSED**

ドライブが CD プレーヤーモードで、再生を一時停止していることを示します。

**STILL**

`PAUSED` と同じですが、古いモデル ( `non 3301` ) である Toshiba CD-ROM ドライブのものです。このドライブはもう SGI から出荷されていません。

**audio**

**pnum**



`index`  
`ptime`  
`atime`  
`catalog`  
`ident`  
`control`

これらは整数の定数で、パーザのいろいろなタイプのコールバックを示しています。コールバックは CD パーザオブジェクトの `addcallback()` で設定できます (下記参照)。

### 20.3.1 プレーヤーオブジェクト

プレーヤーオブジェクト (`open()` で返されます) には以下のメソッドがあります:

**`allowremoval()`**

CD-ROM ドライブのイジェクトボタンのロックを解除して、ユーザが CD キャディを排出するのを許可します。

**`bestreadsize()`**

メソッド `reada()` のパラメータ `num_frames` として最適の値を返します。最適値は CD-ROM ドライブからの連続したデータフローが許可される値が定義されます。

**`close()`**

プレーヤーオブジェクトと関連付けられたリソースを解放します。`close()` を呼び出したあとでは、そのオブジェクトに対するメソッドは使用できません。

**`eject()`**

CD-ROM ドライブからキャディを排出します。

**`getstatus()`**

CD-ROM ドライブの現在の状態に関する情報を返します。返される情報は以下の値からなるタプルです: `state`、`track`、`rtime`、`atime`、`ttime`、`first`、`last`、`scsi_audio`、`cur_block`。`rtime` は現在のトラックの初めからの相対的な時間; `atime` はディスクの初めからの相対的な時間; `ttime` はディスクの全時間です。それぞれの値の詳細については、マニュアルページ `CDgetstatus(3dm)` を参照してください。`state` の値は以下のうちのどれか一つです: `ERROR`、`NODISC`、`READY`、`PLAYING`、`PAUSED`、`STILL`、`CDROM`。

**`gettrackinfo(track)`**

特定のトラックについての情報を返します。返される情報は、トラックの開始時刻とトラックの時間の長さの二つの要素からなるタプルです。

**`msftoblock(min, sec, frame)`**

分、秒、フレームの 3 つからなる絶対的なタイムコードを、与えられた CD-ROM ドライブの相当する論理ブロック番号に変換します。時刻を比較するには `msftoblock()` よりも `msftoframe()` を使うべきです。論理ブロック番号は、CD-ROM ドライブによって必要とされるオフセット値が異なるため、フレームナンバーと異なります。

**`play(start, play)`**

CD-ROM ドライブのオーディオ CD の特定のトラックから再生を開始します。CD-ROM ドライブのヘッドフォン端子と (備えているなら) オーディオ端子から出力されます。ディスクの最後で再生は停止します。`start` は再生を開始する CD のトラックナンバーです; `play` が 0 なら、CD は最初の一時停止状態になります。その状態からメソッド `togglepause()` で再生を開始できます。

**`playabs(minutes, seconds, frames, play)`**

`play()` と似ていますが、開始位置をトラックナンバーの代わりに分、秒、フレームで与えます。

**`playtrack(start, play)`**

`play()` と似ていますが、トラックの終わりで再生を停止します。

**playtrackabs**(*track, minutes, seconds, frames, play*)

`play()` と似ていますが、指定した絶対的な時刻から再生を開始して、指定したトラックで終了します。

**preventremoval**()

CD-ROM ドライブのイジェクトボタンをロックして、ユーザが CD キャディを排出できないようにします。

**readda**(*num\_frames*)

CD-ROM ドライブにマウントされたオーディオ CD から、指定したフレーム数を読み込みます。オーディオフレームのデータを示す文字列を返します。この文字列はそのままパーザオブジェクトのメソッド `parseframe()` へ渡すことができます。

**seek**(*minutes, seconds, frames*)

CD-ROM から次にデジタルオーディオデータを読み込む開始位置のポインタを設定します。ポインタは *minutes*、*seconds*、*frames* で指定した絶対的なタイムコードの位置に設定されます。返される値はポインタが設定された論理ブロック番号です。

**seekblock**(*block*)

CD-ROM から次にデジタルオーディオデータを読み込む開始位置のポインタを設定します。ポインタは指定した論理ブロック番号に設定されます。返される値はポインタが設定された論理ブロック番号です。

**seektrack**(*track*)

CD-ROM から次にデジタルオーディオデータを読み込む開始位置のポインタを設定します。ポインタは指定したトラックに設定されます。返される値はポインタが設定された論理ブロック番号です。

**stop**()

現在実行中の再生を停止します。

**togglepause**()

再生中なら CD を一時停止し、一時停止中なら再生します。

### 20.3.2 パーザオブジェクト

パーザオブジェクト (`createparser()` で返されます) には以下のメソッドがあります：

**addcallback**(*type, func, arg*)

パーザにコールバックを加えます。デジタルオーディオストリームの 8 つの異なるデータタイプのためのコールバックをパーザは持っています。これらのタイプのための定数は `cd` モジュールのレベルで定義されています (上記参照)。コールバックは以下のように呼び出されます：`func(arg, type, data)`、ここで *arg* はユーザが与えた引数、*type* はコールバックの特定のタイプ、*data* はこの *type* のコールバックに渡されるデータです。データのタイプは以下のようにコールバックのタイプによって決まります：

Type	Value
audio	al.writesamps() へそのまま渡すことのできる文字列。
pnum	プログラム (トラック) ナンバーを示す整数。
index	インデックスナンバーを示す整数。
ptime	プログラムの時間を示す分、秒、フレームからなるタプル。
atime	絶対的な時刻を示す分、秒、フレームからなるタプル。
catalog	CD のカタログナンバーを示す 13 文字の文字列。
ident	録音の ISRC 識別番号を示す 12 文字の文字列。文字列は 2 文字の国別コード、3 文字の所有者コード、2 文字の年号、5 文字のシリアルナンバーからなります。
control	CD のサブコードデータのコントロールビットを示す整数。

**deleteparser()**

パーザを消去して、使用していたメモリを解放します。この呼び出しのあと、オブジェクトは使用できません。オブジェクトへの最後の参照が削除されると、自動的にこのメソッドが呼び出されます。

**parseframe(frame)**

readdata() などから返されたデジタルオーディオ CD のデータの 1 つあるいはそれ以上のフレームをパースします。データ内にどのようなサブコードがあるかを決定します。その前のフレームからサブコードが変化していたら、parseframe() は対応するタイプのコールバックを起動して、フレーム内のサブコードデータをコールバックに渡します。C の関数とは違って、1 つ以上のデジタルオーディオデータのフレームをこのメソッドに渡すことができます。

**removecallback(type)**

指定した type のコールバックを削除します。

**resetparser()**

サブコードを追跡しているパーザのフィールドをリセットして、初期状態にします。ディスクを交換したあと、resetparser() を呼び出さなければなりません。

## 20.4 fl — グラフィカルユーザーインターフェースのための FORMS ライブラリ

このモジュールは、Mark Overmars による FORMS Library へのインターフェースを提供します。FORMS ライブラリのソースは anonymous ftp 'ftp.cs.ruu.nl' の 'SGI/FORMS' ディレクトリから入手できます。最新のテストはバージョン 2.0b で行いました。

ほとんどの関数は接頭辞の 'fl\_' を取ると、対応する C の関数名になります。ライブラリで使われる定数は後述の FL モジュールで定義されています。

Python でこのオブジェクトを作る方法は C とは少し違っています: ライブラリに保持された '現在のフォーム' に新しい FORMS オブジェクトを加えるのではなく、フォームに FORMS オブジェクトを加えるには、フォームを示す Python オブジェクトのメソッドで全て行います。したがって、C の関数の fl\_addto\_form() と fl\_end\_form() に相当するものは Python にはありませんし、fl\_bgn\_form() に相当するものとしては fl.make\_form() を呼び出します。

用語のちょっとした混乱に注意してください: FORMS ではフォームの中に置くことができるボタン、スライダーなどに *object* の用語を使います。Python では全ての値が 'オブジェクト' です。FORMS への Python のインターフェースによって、2 つの新しいタイプの Python オブジェクト: フォームオブジェクト (フォーム全体を示します) と FORMS オブジェクト (ボタン、スライダーなどの一つひとつを示します) を作ります。おそらく、混乱するほどのことではありません。

FORMS への Python インターフェースに ‘フリーオブジェクト’ はありませんし、Python でオブジェクトクラスを書いて加える簡単な方法也没有ありません。しかし、GL イベントハンドルへの FORMS インターフェースが利用可能で、純粋な GL ウィンドウに FORMS を組み合わせることができます。

注意：fl をインポートすると、GL の関数 `foreground()` と FORMS のルーチン `fl_init()` を呼び出します。

#### 20.4.1 fl モジュールに定義されている関数

fl モジュールには以下の関数が定義されています。これらの関数の働きに関する詳しい情報については、FORMS ドキュメントで対応する C の関数の説明を参照してください。

**make\_form**(*type, width, height*)

与えられたタイプ、幅、高さでフォームを作ります。これは *form* オブジェクトを返します。このオブジェクトは後述のメソッドを持ちます。

**do\_forms**( )

標準の FORMS のメインループです。ユーザからの応答が必要な FORMS オブジェクトを示す Python オブジェクト、あるいは特別な値 `FL.EVENT` を返します。

**check\_forms**( )

FORMS イベントを確認します。do\_forms() が返すもの、あるいはユーザからの応答をすぐに必要とするイベントがないなら `None` を返します。

**set\_event\_call\_back**(*function*)

イベントのコールバック関数を設定します。

**set\_graphics\_mode**(*rgbmode, doublebuffering*)

グラフィックモードを設定します。

**get\_rgbmode**( )

現在の RGB モードを返します。これは C のグローバル変数 `fl_rgbmode` の値です。

**show\_message**(*str1, str2, str3*)

3 行のメッセージと OK ボタンのあるダイアログボックスを表示します。

**show\_question**(*str1, str2, str3*)

3 行のメッセージと YES、NO のボタンのあるダイアログボックスを表示します。ユーザによって YES が押されたら 1、NO が押されたら 0 を返します。

**show\_choice**(*str1, str2, str3, but1[, but2[, but3]]*)

3 行のメッセージと最大 3 つまでのボタンのあるダイアログボックスを表示します。ユーザによって押されたボタンの数値を返します（それぞれ 1、2、3）。

**show\_input**(*prompt, default*)

1 行のプロンプトメッセージと、ユーザが入力できるテキストフィールドを持つダイアログボックスを表示します。2 番目の引数はデフォルトで表示される入力文字列です。ユーザが入力した文字列が返されます。

**show\_file\_selector**(*message, directory, pattern, default*)

ファイル選択ダイアログを表示します。ユーザによって選択されたファイルの絶対パス、あるいはユーザが Cancel ボタンを押した場合は `None` を返します。

**get\_directory**( )

**get\_pattern**( )

**get\_filename**( )

これらの関数は最後にユーザが `show_file_selector()` で選択したディレクトリ、パターン、ファ

イル名 (パスの末尾のみ) を返します。

```
qdevice(dev)
unqdevice(dev)
isqueued(dev)
qtest()
qread()
qreset()
qenter(dev, val)
get_mouse()
tie(button, valuator1, valuator2)
```

これらの関数は対応する GL 関数への FORMS のインターフェースです。fl.do\_events() を使っていて、自分で何か GL イベントを操作したいときにこれらを使います。FORMS が扱うことのできない GL イベントが検出されたら fl.do\_forms() が特別の値 FL.EVENT を返すので、fl.qread() を呼び出して、キューからイベントを読み込むべきです。対応する GL の関数は使わないでください!

```
color()
mapcolor()
getmcolor()
```

FORMS ドキュメントにある fl\_color()、fl\_mapcolor()、fl\_getmcolor() の記述を参照してください。

## 20.4.2 フォームオブジェクト

フォームオブジェクト (上で述べた make\_form() で返されます) には下記のメソッドがあります。各メソッドは名前の接頭辞に 'fl\_' を付けた C の関数に対応します; また、最初の引数はフォームのポインタです; 説明は FORMS の公式文書を参照してください。

全ての add\_\*() メソッドは、FORMS オブジェクトを示す Python オブジェクトを返します。FORMS オブジェクトのメソッドを以下に記載します。ほとんどの FORMS オブジェクトは、そのオブジェクトの種類ごとに特有のメソッドもいくつか持っています。

```
show_form(placement, bordertype, name)
```

フォームを表示します。

```
hide_form()
```

フォームを隠します。

```
redraw_form()
```

フォームを再描画します。

```
set_form_position(x, y)
```

フォームの位置を設定します。

```
freeze_form()
```

フォームを固定します。

```
unfreeze_form()
```

固定したフォームの固定を解除します。

```
activate_form()
```

フォームをアクティベートします。

```
deactivate_form()
```

フォームをデアクティベートします。

**bgn\_group()**  
 新しいオブジェクトのグループを作ります；グループオブジェクトを返します。

**end\_group()**  
 現在のオブジェクトのグループを終了します。

**find\_first()**  
 フォームの中の最初のオブジェクトを見つけます。

**find\_last()**  
 フォームの中の最後のオブジェクトを見つけます。

**add\_box(*type, x, y, w, h, name*)**  
 フォームにボックスオブジェクトを加えます。特別な追加のメソッドはありません。

**add\_text(*type, x, y, w, h, name*)**  
 フォームにテキストオブジェクトを加えます。特別な追加のメソッドはありません。

**add\_clock(*type, x, y, w, h, name*)**  
 フォームにクロックオブジェクトを加えます。  
 メソッド：`get_clock()`。

**add\_button(*type, x, y, w, h, name*)**  
 フォームにボタンオブジェクトを加えます。  
 メソッド：`get_button()`、`set_button()`。

**add\_lightbutton(*type, x, y, w, h, name*)**  
 フォームにライトボタンオブジェクトを加えます。  
 メソッド：`get_button()`、`set_button()`。

**add\_roundbutton(*type, x, y, w, h, name*)**  
 フォームにラウンドボタンオブジェクトを加えます。  
 メソッド：`get_button()`、`set_button()`。

**add\_slider(*type, x, y, w, h, name*)**  
 フォームにスライダーオブジェクトを加えます。  
 メソッド：`set_slider_value()`、`get_slider_value()`、`set_slider_bounds()`、`get_slider_bounds()`、`set_slider_return()`、`set_slider_size()`、`set_slider_precision()`、`set_slider_step()`。

**add\_valslider(*type, x, y, w, h, name*)**  
 フォームにバリュースライダーオブジェクトを加えます。  
 メソッド：`set_slider_value()`、`get_slider_value()`、`set_slider_bounds()`、`get_slider_bounds()`、`set_slider_return()`、`set_slider_size()`、`set_slider_precision()`、`set_slider_step()`。

**add\_dial(*type, x, y, w, h, name*)**  
 フォームにダイアルオブジェクトを加えます。  
 メソッド：`set_dial_value()`、`get_dial_value()`、`set_dial_bounds()`、`get_dial_bounds()`。

**add\_positioner(*type, x, y, w, h, name*)**  
 フォームに2次元ポジショナーオブジェクトを加えます。  
 メソッド：`set_positioner_xvalue()`、`set_positioner_yvalue()`、`set_positioner_xbounds()`、`set_positioner_ybounds()`、`get_positioner_xvalue()`、`get_positioner_yvalue()`、`get_positioner_xbounds()`、`get_positioner_ybounds()`。



**add\_counter**(*type, x, y, w, h, name*)

フォームにカウンタオブジェクトを加えます。

メソッド: `set_counter_value()`、`get_counter_value()`、`set_counter_bounds()`、`set_counter_step()`、`set_counter_precision()`、`set_counter_return()`。

**add\_input**(*type, x, y, w, h, name*)

フォームにインプットオブジェクトを加えます。

メソッド: `set_input()`、`get_input()`、`set_input_color()`、`set_input_return()`。

**add\_menu**(*type, x, y, w, h, name*)

フォームにメニューオブジェクトを加えます。

メソッド: `set_menu()`、`get_menu()`、`addto_menu()`。

**add\_choice**(*type, x, y, w, h, name*)

フォームにチョイスオブジェクトを加えます。

メソッド: `set_choice()`、`get_choice()`、`clear_choice()`、`addto_choice()`、`replace_choice()`、`delete_choice()`、`get_choice_text()`、`set_choice_fontsize()`、`set_choice_fontstyle()`。

**add\_browser**(*type, x, y, w, h, name*)

フォームにブラウザオブジェクトを加えます。

メソッド: `set_browser_topline()`、`clear_browser()`、`add_browser_line()`、`addto_browser()`、`insert_browser_line()`、`delete_browser_line()`、`replace_browser_line()`、`get_browser_line()`、`load_browser()`、`get_browser_maxline()`、`select_browser_line()`、`deselect_browser_line()`、`deselect_browser()`、`isselected_browser_line()`、`get_browser()`、`set_browser_fontsize()`、`set_browser_fontstyle()`、`set_browser_specialkey()`。

**add\_timer**(*type, x, y, w, h, name*)

フォームにタイマーオブジェクトを加えます。

メソッド: `set_timer()`、`get_timer()`。

フォームオブジェクトには以下のデータ属性があります；FORMS ドキュメントを参照してください：

名称	C の型	意味
<code>window</code>	<code>int (read-only)</code>	GL ウィンドウの id
<code>w</code>	<code>float</code>	フォームの幅
<code>h</code>	<code>float</code>	フォームの高さ
<code>x</code>	<code>float</code>	フォーム左肩の x 座標
<code>y</code>	<code>float</code>	フォーム左肩の y 座標
<code>deactivated</code>	<code>int</code>	フォームがディアクティベートされているなら非ゼロ
<code>visible</code>	<code>int</code>	フォームが可視なら非ゼロ
<code>frozen</code>	<code>int</code>	フォームが固定されているなら非ゼロ
<code>doublebuf</code>	<code>int</code>	ダブルバッファリングがオンなら非ゼロ

### 20.4.3 FORMS オブジェクト

FORMS オブジェクトの種類ごとに特有のメソッドの他に、全ての FORMS オブジェクトは以下のメソッドも持っています：

**set\_call\_back**(*function, argument*)

オブジェクトのコールバック関数と引数を設定します。オブジェクトがユーザからの応答を必要とす

るときには、コールバック関数は2つの引数、オブジェクトとコールバックの引数とともに呼び出されます。(コールバック関数のないFORMS オブジェクトは、ユーザからの応答を必要とするときには `fl.do_forms()` あるいは `fl.check_forms()` によって返されます。) 引数なしにこのメソッドを呼び出すと、コールバック関数を削除します。

`delete_object()`

オブジェクトを削除します。

`show_object()`

オブジェクトを表示します。

`hide_object()`

オブジェクトを隠します。

`redraw_object()`

オブジェクトを再描画します。

`freeze_object()`

オブジェクトを固定します。

`unfreeze_object()`

固定したオブジェクトの固定を解除します。

FORMS オブジェクトには以下のデータ属性があります；FORMS ドキュメントを参照してください。

名称	C の型	意味
<code>objclass</code>	<code>int (read-only)</code>	オブジェクトクラス
<code>type</code>	<code>int (read-only)</code>	オブジェクトタイプ
<code>boxtype</code>	<code>int</code>	ボックスタイプ
<code>x</code>	<code>float</code>	左肩の x 座標
<code>y</code>	<code>float</code>	左肩の y 座標
<code>w</code>	<code>float</code>	幅
<code>h</code>	<code>float</code>	高さ
<code>col1</code>	<code>int</code>	第 1 の色
<code>col2</code>	<code>int</code>	第 2 の色
<code>align</code>	<code>int</code>	配置
<code>lcol</code>	<code>int</code>	ラベルの色
<code>lsize</code>	<code>float</code>	ラベルのフォントサイズ
<code>label</code>	<code>string</code>	ラベルの文字列
<code>lstyle</code>	<code>int</code>	ラベルのスタイル
<code>pushed</code>	<code>int (read-only)</code>	(FORMS ドキュメント参照)
<code>focus</code>	<code>int (read-only)</code>	(FORMS ドキュメント参照)
<code>belowmouse</code>	<code>int (read-only)</code>	(FORMS ドキュメント参照)
<code>frozen</code>	<code>int (read-only)</code>	(FORMS ドキュメント参照)
<code>active</code>	<code>int (read-only)</code>	(FORMS ドキュメント参照)
<code>input</code>	<code>int (read-only)</code>	(FORMS ドキュメント参照)
<code>visible</code>	<code>int (read-only)</code>	(FORMS ドキュメント参照)
<code>radio</code>	<code>int (read-only)</code>	(FORMS ドキュメント参照)
<code>automatic</code>	<code>int (read-only)</code>	(FORMS ドキュメント参照)

## 20.5 FL — fl モジュールで使用される定数

このモジュールには、組み込みモジュール `fl` を使うのに必要なシンボル定数が定義されています（上記参照）；これらは名前の接頭辞 ‘FL\_’ が省かれていることを除いて、C のヘッダファイル `<forms.h>` に定義されているものと同じです。定義されている名称の完全なリストについては、モジュールのソースをご覧ください。お勧めする使い方は以下の通りです：

```
import fl
from FL import *
```

## 20.6 flp — 保存された FORMS デザインをロードする関数

このモジュールには、FORMS ライブラリ（上記の `fl` モジュールを参照してください）とともに配布される ‘フォームデザイナー’（`fdesign`）プログラムで作られたフォームの定義を読み込む関数が定義されています。

詳しくは Python ライブラリソースのディレクトリの中の ‘flp.doc’ を参照してください。

XXX 完全な説明をここに書いて！

## 20.7 fm — Font Manager インターフェース

このモジュールは IRIS Font Manager ライブラリへのアクセスを提供します。Silicon Graphics マシン上だけで利用可能です。次も参照してください：4*Sight User's Guide*, section 1, chapter 5: “Using the IRIS Font Manager”。このモジュールは、まだ IRIS Font Manager への完全なインターフェースではありません。サポートされていない機能は次のものです：matrix operations; cache operations; character operations（代わりに string operations を使ってください）；font info のうちのいくつか；individual glyph metrics; printer matching。

以下の操作をサポートしています：

`init()`

関数を初期化します。`fminit()` を呼び出します。この関数は `fm` モジュールを最初にインポートすると自動的に呼び出されるので、普通、呼び出す必要はありません。

`findfont(fontname)`

フォントハンドルオブジェクトを返します。`fmfindfont(fontname)` を呼び出します。

`enumerate()`

利用可能なフォント名のリストを返します。この関数は `fmenumerate()` へのインターフェースです。

`prstr(string)`

現在のフォントを使って文字列をレンダリングします（下のフォントハンドルメソッド `setfont()` を参照）。`fmprstr(string)` を呼び出します。

`setpath(string)`

フォントの検索パスを設定します。`fmsetpath(string)` を呼び出します。（XXX 機能しない!?!）

`fontpath()`

現在のフォント検索パスを返します。

フォントハンドルオブジェクトは以下の操作をサポートします：

`scalefont(factor)`

このフォントを拡大／縮小したハンドルを返します。`fmscalefont(fh, factor)` を呼び出します。

`setfont()`

このフォントを現在のフォントに設定します。注意：フォントハンドルオブジェクトが削除されると、設定は告知なしに元に戻ります。`fmsetfont(fh)` を呼び出します。

`getfontname()`

このフォントの名前を返します。`fmgetfontname(fh)` を呼び出します。

`getcomment()`

このフォントに関連付けられたコメント文字列を返します。コメント文字列が何もなければ例外を返します。`fmgetcomment(fh)` を呼び出します。

`getfontinfo()`

このフォントに関連したデータを含むタプルを返します。これは `fmgetfontinfo()` へのインターフェースです。以下の数値を含むタプルを返します：*(printer\_matched、fixed\_width、xorig、yorig、xsize、ysize、height、nglyphs)*。

`getstrwidth(string)`

このフォントで *string* を描いたときの幅をピクセル数で返します。`fmgetstrwidth(fh, string)` を呼び出します。

## 20.8 gl — Graphics Library インターフェース

このモジュールは Silicon Graphics の *Graphics Library* へのアクセスを提供します。Silicon Graphics マシン上だけで利用可能です。

警告：GL ライブラリの不適切な呼び出しによっては、Python インタプリタがコアを吐き出すことがあります。特に、GL のほとんどの関数では最初のウィンドウを開く前に呼び出すのは安全ではありません。

このモジュールはとても大きいので、ここに全てを記述することはできませんが、以下の説明で出発点としては十分でしょう。C の関数のパラメータは、以下のような決まりに従って Python に翻訳されます：

- 全て (short、long、unsigned) の整数値 (int) は Python の整数に相当します。
- 全ての浮動小数点数と倍精度浮動小数点数は Python の浮動小数点数に相当します。たいていの場合、Python の整数も使えます。
- 全ての配列は Python の一次元のリストに相当します。たいていの場合、タプルも使えます。
- 全ての文字列と文字の引数は、Python の文字列に相当します。例えば、`winopen('Hi There!')` と `rotate(900, 'z')`。
- 配列である引数の長さを特定するためだけに使われる全て (short、long、unsigned) の整数値の引数あるいは戻り値は、無視されます。例えば、C の呼び出しで、

```
lmdef(deftype, index, np, props)
```

これは Python では、こうなります。

```
lmdef(deftype, index, props)
```

- 出力のための引数は、引数のリストから省略されています；代わりにこれらは関数の戻り値として渡されます。もし 1 つ以上の値が返されるのなら、戻り値はタプルです。もし C の関数が通常の戻り値

(先のルールによって省略されません)と、出力のための引数の両方を取るなら、返り値はタブルの最初に来ます。例: C の呼び出しで、

```
getmcolor(i, &red, &green, &blue)
```

これは Python ではこうなります。

```
red, green, blue = getmcolor(i)
```

以下の関数は一般的でないか、引数に特別な決まりを持っています:

**varray**(*argument*)

**v3d()** の呼び出しに相当しますが、それよりも速いです。*argument* は座標のリスト (あるいはタブル) です。各座標は (*x*, *y*, *z*) あるいは (*x*, *y*) のタブルでなければなりません。座標は 2 次元あるいは 3 次元が可能ですが、全て同次元でなければなりません。ですが、浮動小数点数と整数を混合して使えます。座標は (マニュアルページにあるように) 必要であれば *z* = 0.0 と仮定して、常に 3 次元の精密な座標に変換され、各座標について **v3d()** が呼び出されます。

**nvarray**()

**n3f** と **v3f** の呼び出しに相当しますが、それらよりも速いです。引数は法線と座標とのペアからなる配列 (リストあるいはタブル) です。各ペアは座標と、その座標からの法線とのタブルです。各座標と各法線は (*x*, *y*, *z*) からなるタブルでなければなりません。3 つの座標が渡されなければなりません。浮動小数点数と整数を混合して使えます。各ペアについて、法線に対して **n3f()** が呼び出され、座標に対して **v3f()** が呼び出されます。

**vnarray**()

**nvarray()** と似ていますが、各ペアは始めに座標を、2 番目に法線を持っています。

**nurbssurface**(*s\_k*, *t\_k*, *ctl*, *s\_ord*, *t\_ord*, *type*)

**nurbs** (非均一有理 B スプライン) 曲面を定義します。*ctl*[ ][ ] の次元は以下のように計算されます: [len(*s\_k*) - *s\_ord*], [len(*t\_k*) - *t\_ord*]。

**nurbscurve**(*knots*, *ctlpoints*, *order*, *type*)

**nurbs** (非均一有理 B スプライン) 曲線を定義します。*ctlpoints* の長さは、len(*knots*) - *order* です。

**pwlcurve**(*points*, *type*)

区分線形曲線 (piecewise-linear curve) を定義します。*points* は座標のリストです。*type* は **N\_ST** でなければなりません。

**pick**(*n*)

**select**(*n*)

これらの関数はただ一つの引数を取り、**pick/select** に使うバッファのサイズを設定します。

**endpick**()

**endselect**()

これらの関数は引数を取りません。**pick/select** に使われているバッファの大きさを示す整数のリストを返します。バッファがあふれているを検出するメソッドはありません。

小さいですが完全な Python の GL プログラムの例をここに挙げます:

```

import gl, GL, time

def main():
    gl.foreground()
    gl.perspective(500, 900, 500, 900)
    w = gl.winopen('CrissCross')
    gl.ortho2(0.0, 400.0, 0.0, 400.0)
    gl.color(GL.WHITE)
    gl.clear()
    gl.color(GL.RED)
    gl.bgnline()
    gl.v2f(0.0, 0.0)
    gl.v2f(400.0, 400.0)
    gl.endline()
    gl.bgnline()
    gl.v2f(400.0, 0.0)
    gl.v2f(0.0, 400.0)
    gl.endline()
    time.sleep(5)

main()

```

#### 参考資料:

*PyOpenGL: Python の OpenGL との結合*

(<http://pyopengl.sourceforge.net/>)

OpenGL へのインターフェースが利用できます；詳しくは **PyOpenGL** プロジェクト <http://pyopengl.sourceforge.net/> から情報を入手できます。これは、SGI のハードウェアが 1996 年頃より前である必要がないので、OpenGL の方が良い選択かもしれません。

## 20.9 DEVICE — gl モジュールで使われる定数

このモジュールには、Silicon Graphics の *Graphics Library* で使われる定数が定義されています。これらは C のプログラマーがヘッダーファイル<gl/device.h>の中から使っているものです。詳しくはモジュールのソースファイルをご覧ください。

### 20.10 GL — gl モジュールで使われる定数

このモジュールには Silicon Graphics の *Graphics Library* で使われる C のヘッダーファイル<gl/gl.h>の定数が定義されています。詳しくはモジュールのソースファイルをご覧ください。

## 20.11 imgfile — SGI imglib ファイルのサポート

`imgfile` モジュールは、Python プログラムが SGI `imglib` 画像ファイル（'.rgb' ファイルとしても知られています）にアクセスできるようにします。このモジュールは完全なものにはほど遠いですが、その機能はある状況で十分に立つものなので、ライブラリで提供されています。現在、カラーマップ形式のファイルはサポートされていません。

このモジュールでは以下の変数および関数が定義されています:

#### **exception error**

この例外は、サポートされていないファイル形式の場合のような全てのエラーで送出されます。



**getsizes**(*file*)

この関数はタプル (*x*, *y*, *z*) を返します。*x* および *y* は画像のサイズをピクセルで表したもので、*z* はピクセルあたりのバイト長です。3 バイトの RGB ピクセルと 1 バイトのグレイスケールピクセルのみが現在サポートされています。

**read**(*file*)

この関数は指定されたファイル上の画像を読み出して復号化し、Python 文字列として返します。この文字列は 1 バイトのグレイスケールピクセルか、4 バイトの RGBA ピクセルによるものです。左下のピクセルが文字列中の最初のピクセルになります。これは `gl.lrectwrite()` に渡すのに適した形式です。

**readscaled**(*file*, *x*, *y*, *filter*[, *blur*])

この関数は `read` と同じですが、*x* および *y* のサイズにスケールされた画像を返します。*filter* および *blur* パラメタが省略された場合、単にピクセルデータを捨てたり複製したりすることによってスケール操作が行われるので、処理結果は、特に計算機上で合成した画像の場合にはおよそ完璧とはいえないものになります。

そうする代わりに、スケール操作後に画像を平滑化するために用いるフィルタを指定することができます。サポートされているフィルタの形式は 'impulse'、'box'、'triangle'、'quadratic'、および 'gaussian' です。フィルタを指定する場合、*blur* はオプションのパラメタで、フィルタの不明瞭化度を指定します。標準の値は 1.0 です。

`readscaled()` は正しいアスペクト比をまったく維持しようとしないので、それはユーザの責任になります。

**ttob**(*flag*)

この関数は画像のスキャンラインの読み書きを下から上に向かって行う (フラグがゼロの場合で、SGI GL 互換です) か、上から下に向かって行う (フラグが 1 の場合で、X 互換です) かを決定する大域的なフラグを設定します。標準の値はゼロです。

**write**(*file*, *data*, *x*, *y*, *z*)

この関数は *data* 中の RGB またはグレイスケールのデータを画像ファイル *file* に書き込みます。*x* および *y* には画像のサイズを与え、*z* は 1 バイトグレイスケール画像の場合には 1 で、RGB 画像の場合には 3 (4 バイトの値として記憶され、下位 3 バイトが使われます) です。これらは `gl.lrectread()` が返すデータの形式です。

## 20.12 jpeg — JPEG ファイルの読み書きを行う

この jpeg モジュールは Independent JPEG Group (IJG) によって書かれた JPEG 圧縮及び展開アルゴリズムを提供します。JPEG 形式は写真等の画像圧縮で標準的に利用され、ISO 10918 で定義されています。JPEG、あるいは Independent JPEG Group ソフトウェアの詳細は、標準 JPEG、若しくは提供されるソフトウェアのドキュメントを参照してください。

JPEG ファイルを扱うポータブルなインタフェースは Fredrik Lundh による Python Imaging Library (PIL) があります。また、PIL の情報は <http://www.pythonware.com/products/pil/> で見つけることができます。

モジュール `jpeg` では、一つの例外といくつかの関数を定義しています。

**exception error**

関数 `compress()` または `decompress()` のエラーで上げられる例外です。

**compress**(*data*, *w*, *h*, *b*)

イメージファイルの幅 *w*、高さ *h*、1 ピクセルあたりのバイト数 *b* を引数として扱います。データは

SGI GL 順になっていて、最初のピクセルは左下端になります。また、これは `gl.lrectread()` が返す値をすぐに `compress()` にかけるためです。現在は、1 バイト若しくは 4 バイトのピクセルを取り扱うことができます、前者はグレースケール、後者は RGB カラーを扱います。`compress()` は、圧縮された JFIF 形式のイメージが含まれた文字列を返します。

#### **decompress (data)**

データは圧縮された JFIF 形式のイメージが含まれた文字列で、この関数はタプル (`data`, `width`, `height`, `bytesperpixel`) を返します。また、そのデータは `gl.lrectwrite()` を通過します。

#### **setoption (name, value)**

`compress()` と `decompress()` を呼ぶための様々なオプションをセットします。次のオプションが利用できます:

オプション	効果
'forcegray'	入力が RGB でも強制的にグレースケールを出力します。
'quality'	圧縮後イメージの品質を 0 から 100 の間の値で指定します (デフォルトは 75 です)。これは圧縮にのみ影響します。
'optimize'	ハフマンテーブルを最適化します。時間がかかりますが、高圧縮になります。これは圧縮にのみ影響します。
'smooth'	圧縮されていないイメージ上でインターブロックスムーシングを行います。低品質イメージに役立ちます。これは展開にのみ影響します。

#### 参考資料:

##### *JPEG Still Image Data Compression Standard*

The canonical reference for the JPEG image format, by Pennebaker and Mitchell.

*Information Technology - Digital Compression and Coding of Continuous-tone Still Images - Requirements and Guidelines*  
(<http://www.w3.org/Graphics/JPEG/itu-t81.pdf>)

The ISO standard for JPEG is also published as ITU T.81. This is available online in PDF form.

# SunOS 特有のサービス

この章では、SunOS オペレーティングシステム バージョン 4 及び 5(Solaris バージョン 2) に固有の機能を解説します。

## 21.1 sunaudiodev — Sun オーディオハードウェアへのアクセス

このモジュールを使うと、Sun のオーディオインターフェースにアクセスできます。Sun オーディオハードウェアは、1 秒あたり 8k のサンプリングレート、u-LAW フォーマットでオーディオデータを録音、再生できます。完全な説明文書はマニュアルページ *audio(7I)* にあります。

モジュール SUNAUDIODEV には、このモジュールで使われる定数が定義されています。

このモジュールには、以下の変数と関数が定義されています：

### exception error

この例外は、全てのエラーについて発生します。引数は誤りを説明する文字列です。

### open(mode)

この関数はオーディオデバイスを開き、Sun オーディオデバイスのオブジェクトを返します。こうすることで、オブジェクトが I/O に使用できるようになります。パラメータ *mode* は次のうちのいずれか一つで、録音のみには 'r'、再生のみには 'w'、録音と再生両方には 'rw'、コントロールデバイスへのアクセスには 'control' です。レコーダーやプレーヤーには同時に 1 つのプロセスしかアクセスが許されていないので、必要な動作についてだけデバイスをオープンするのがいい考えです。詳しくは *audio(7I)* を参照してください。マニュアルページにあるように、このモジュールは環境変数 AUDIODEV の中のベースオーディオデバイスファイルネームを初めに参照します。見つからない場合は '/dev/audio' を参照します。コントロールデバイスについては、ベースオーディオデバイスに "ctl" を加えて扱われます。

### 21.1.1 オーディオデバイスオブジェクト

オーディオデバイスオブジェクトは open() で返され、このオブジェクトには以下のメソッドが定義されています (control オブジェクトは除きます。これには getinfo()、setinfo()、fileno()、drain() だけが定義されています)：

#### close()

このメソッドはデバイスを明示的に閉じます。オブジェクトを削除しても、それを参照しているものがあって、すぐに閉じてくれない場合に便利です。閉じられたデバイスを使うことはできません。

#### fileno()

デバイスに関連づけられたファイルディスクリプタを返します。これは、後述の SIGPOLL の通知を組み立てるのに使われます。

#### drain()

このメソッドは全ての出力中のプロセスが終了するまで待って、それから制御が戻ります。このメソッドの呼び出しはそう必要ではありません：オブジェクトを削除すると自動的にオーディオデバイスを閉じて、暗黙のうちに吐き出します。

**flush()**

このメソッドは全ての出力中のものを捨て去ります。ユーザの停止命令に対する反応の遅れ（1秒までの音声のバッファリングによって起こります）を避けるのに使われます。

**getinfo()**

このメソッドは入出力のボリューム値などの情報を引き出して、オーディオステータスのオブジェクト形式で返します。このオブジェクトには何もメソッドはありませんが、現在のデバイスの状態を示す多くの属性が含まれます。属性の名称と意味は<sun/audioio.h>と *audio(7I)* に記載があります。メンバー名は相当する C のものとは少し違っています：ステータスオブジェクトは1つの構造体です。その中の構造体である `play` のメンバーには名前の初めに `'o_'` がついていて、`record` には `'i_'` がついています。そのため、C のメンバーである `play.sample_rate` は `o_sample_rate` として、`record.gain` は `i_gain` として参照され、`monitor_gain` はそのまま `monitor_gain` で参照されます。

**ibufcount()**

このメソッドは録音側でバッファリングされるサンプル数を返します。つまり、プログラムは同じ大きさのサンプルに対する `read()` の呼び出しをブロックしません。

**obufcount()**

このメソッドは再生側でバッファリングされるサンプル数を返します。残念ながら、この数値はブロックなしに書き込めるサンプル数を調べるのには使えません。というのは、カーネルの出力キューの長さは可変だからです。

**read(size)**

このメソッドはオーディオ入力から *size* のサイズのサンプルを読み込んで、Python の文字列として返します。この関数は必要なデータが得られるまで他の操作をブロックします。

**setinfo(status)**

このメソッドはオーディオデバイスのステータスパラメータを設定します。パラメータ *status* は `getinfo()` で返されたり、プログラムで変更されたオーディオステータスオブジェクトです。

**write(samples)**

パラメータとしてオーディオサンプルを Python 文字列を受け取り、再生します。もし十分なバッファの空きがあればすぐに制御が戻り、そうでないならブロックされます。

オーディオデバイスは SIGPOLL を介して様々なイベントの非同期通知に対応しています。Python でこれをどのようにしたらできるか、例を挙げます：

```
def handle_sigpoll(signum, frame):
    print 'I got a SIGPOLL update'

import fcntl, signal, STROPTS

signal.signal(signal.SIGPOLL, handle_sigpoll)
fcntl.ioctl(audio_obj.fileno(), STROPTS.I_SETSIG, STROPTS.S_MSG)
```

## 21.2 SUNAUDIODEV — sunaudiodev で使われる定数

これは `sunaudiodev` に付随するモジュールで、`MIN_GAIN`、`MAX_GAIN`、`SPEAKER` などの便利なシンボル定数を定義しています。定数の名前は C の include ファイル<sun/audioio.h>のものと同じで、初

めの文字列 'AUDIO\_' を除いたものです。





# MS Windows 特有のサービス

この章では、MS Windows プラットフォーム上でのみ利用可能なモジュール群について記述します。

- msvcrt** MS VC++実行時システムの雑多な有用ルーチン群。
- \_winreg** Windows レジストリを操作するためのルーチンおよびオブジェクト。
- winsound** Windows の音声再生機構へのアクセス。

## 22.1 msvcrt – MS VC++実行時システムの有用なルーチン群

このモジュールの関数は、Windows プラットフォームの便利な機能のいくつかに対するアクセス機構を提供しています。高レベルモジュールのいくつかは、提供するサービスを Windows で実装するために、これらの関数を使っています。例えば、getpass モジュールは関数 `getpass()` を実装するためにこのモジュールの関数を使います。

ここに挙げた関数の詳細なドキュメントについては、プラットフォーム API ドキュメントで見つけることができます。

### 22.1.1 ファイル操作関連

**locking**(*fd, mode, nbytes*)

C 言語による実行時システムにおけるファイル記述子 *fd* に基づいて、ファイルの一部にロックをかけます。ロックされるファイルの領域は、現在のファイル位置から *nbytes* バイトで、ファイルの末端まで延長することができます。*mode* は以下に列挙する `LK_*` のいずれか一つでなければなりません。一つのファイルの複数の領域を同時にロックすることは可能ですが、領域が重複してはなりません。接続する領域をまとめて指定することはできません; それらの領域は個別にロック解除しなければなりません。

**LK\_LOCK**

**LK\_RLCK**

指定されたバイト列にロックをかけます。指定領域がロックできなかった場合、プログラムは 1 秒後に再度ロックを試みます。10 回再試行した後もロックをかけられない場合、`IOError` が送出されます。

**LK\_NBLCK**

**LK\_NBRLCK**

指定されたバイト列にロックをかけます。指定領域がロックできなかった場合、`IOError` が送出されます。

**LK\_UNLCK**

指定されたバイト列のロックを解除します。指定領域はあらかじめロックされていなければなりません。

**setmode**(*fd, flags*)

ファイル記述子 *fd* に対して、行末文字の変換モードを設定します。テキストモードに設定するには、*flags* を `os.O_TEXT` にします; バイナリモードにするには `os.O_BINARY` にします。

`open_osfhandle(handle, flags)`

C 言語による実行時システムにおけるファイル記述子をファイルハンドル *handle* から生成します。*flags* パラメタは `os.O_APPEND`、`os.O_RDONLY`、および `os.O_TEXT` をビット単位で OR したものに なります。返されるファイル記述子は `os.fdopen()` でファイルオブジェクトを生成するために使うことができます。

`get_osfhandle(fd)`

ファイル記述子 *fd* のファイルハンドルを返します。*fd* が認識できない場合、`IOError` を送出します。

## 22.1.2 コンソール I/O 関連

`kbhit()`

読み出し待ちの打鍵イベントが存在する場合に真を返します。

`getch()`

打鍵を読み取り、読み出された文字を返します。コンソールには何もエコーバックされません。この関数呼び出しは読み出し可能な打鍵がない場合にはブロックしますが、文字を読み出せるようにするために Enter の打鍵を待つ必要はありません。打鍵されたキーが特殊機能キー (function key) である場合、この関数は `'\000'` または `'\xe0'` を返します; キーコードは次に関数を呼び出した際に返されます。この関数で Control-C の打鍵を読み出すことはできません。

`getche()`

`getch()` に似ていますが、打鍵した字が印字可能な文字の場合エコーバックされます。

`putch(char)`

キャラクタ *char* をバッファリングを行わないでコンソールに出力します。

`ungetch(char)`

キャラクタ *char* をコンソールバッファに“押し戻し (push back)”ます; これにより、押し戻された文字は `getch()` や `getche()` で次に読み出される文字になります。

## 22.1.3 その多の関数

`heapmin()`

`malloc()` されたヒープ領域を強制的に消去させて、未使用のメモリブロックをオペレーティングシステムに返します。この関数は Windows NT 上でのみ動作します。失敗した場合、`IOError` を送出します。

## 22.2 \_winreg – Windows レジストリへのアクセス

2.0 で追加された仕様です。

これらの関数は Windows レジストリ API を Python で使えるようにします。プログラマがレジストリハンドルのクローズを失念した場合でも、確実にハンドルがクローズされるようにするために、整数値をレジストリハンドルとして使う代わりにハンドルオブジェクトが使われます。

このモジュールは Windows レジストリ操作のための非常に低レベルのインタフェースを使えるようにします; 将来、より高レベルのレジストリ API インタフェースを提供するような、新たな winreg モジュールが作られるよう期待します。

このモジュールでは以下の関数を提供します:

#### **CloseKey**(*hkey*)

以前開かれたレジストリキーを閉じます。*hkey* 引数には以前開かれたレジストリキーを特定します。

このメソッド(または `handle.Close()`)を使って *hkey* が閉じられなかった場合、Python が *hkey* オブジェクトを破壊する際に閉じられるので注意してください。

#### **ConnectRegistry**(*computer\_name*, *key*)

他の計算機上にある既定のレジストリハンドル接続を確立し、ハンドルオブジェクト (*handle object*) を返します。

*computer\_name* はリモートコンピュータの名前で、`r"\\computername"` の形式をとります。None の場合、ローカルの計算機が使われます。

*key* は接続したい既定のハンドルです。

戻り値は開かれたキーのハンドルです。関数が失敗した場合、`EnvironmentError` 例外が送出されます。

#### **CreateKey**(*key*, *sub\_key*)

特定のキーを生成するか開き、ハンドルオブジェクトを返します。

*key* はすでに開かれたキーか、既定の `HKEY_*` 定数のうちの一つです。

*sub\_key* はこのメソッドが開く、または新規作成するキーの名前です。

*key* が既定のキーの一つなら、*sub\_key* は None でかまいません。この場合、返されるハンドルは関数に渡されたのと同じキーハンドルです。

キーがすでに存在する場合、この関数は既に存在するキーを開きます。

戻り値は開かれたキーのハンドルです。この関数が失敗した場合、`EnvironmentError` 例外が送出されます。

#### **DeleteKey**(*key*, *sub\_key*)

特定のキーを削除します。

*key* はすでに開かれたキーか、既定の `HKEY_*` 定数のうちの一つです。

*sub\_key* は文字列で、*key* パラメタによって特定されたキーのサブキーでなければなりません。この値は None であってはならず、キーはサブキーを持っていたはなりません。

このメソッドはサブキーをもつキーを削除することはできません。

このメソッドの実行が成功すると、キー全体が、その値すべてを含めて削除されます。このメソッドが失敗した場合、`EnvironmentError` 例外が送出されます。

#### **DeleteValue**(*key*, *value*)

レジストリキーから指定された名前付きの値を削除します。

*key* はすでに開かれたキーか、既定の `HKEY_*` 定数のうちのひとつでなければなりません。

*value* は削除したい値を指定するための文字列です。

#### **EnumKey**(*key*, *index*)

開かれているレジストリキーのサブキーを列挙し、文字列で返します。

*key* はすでに開かれたキーか、既定の `HKEY_*` 定数のうちのひとつでなければなりません。

*index* は整数値で、取得するキーのインデックスを特定します。

この関数は呼び出されるたびに一つのサブキーの名前を取得します。この関数は通常、これ以上サブキーがないことを示す `EnvironmentError` 例外が送出されるまで繰り返し呼び出されます。

#### **EnumValue**(*key*, *index*)

開かれているレジストリキーの値を列挙し、タプルで返します。

*key* はすでに開かれたキーか、既定の HKEY\_\* 定数のうちの一つでなければなりません。

*index* は整数値で、取得する値のインデックスを特定します。

この関数は呼び出されるたびに一つの値の名前を取得します。この関数は通常、これ以上値がないことを示す `EnvironmentError` 例外が送出されるまで繰り返し呼び出されます。

結果は 3 要素のタプルになります:

Index	Meaning
0	値の名前を特定する文字列
1	値のデータを保持するためのオブジェクトで、その型は背後のレジストリ型に依存します
2	値のデータ型を特定する整数です

#### **FlushKey(*key*)**

キーのすべての属性をレジストリに書き込みます。

*key* はすでに開かれたキーか、既定の HKEY\_\* 定数のうちの一つでなければなりません。

キーを変更するために `RegFlushKey` を呼ぶ必要はありません。レジストリの変更は怠惰なフラッシュ機構 (lazy flusher) を使ってフラッシュされます。また、システムの遮断時にもディスクにフラッシュされます。`CloseKey()` と違って、`FlushKey()` メソッドはレジストリに全てのデータを書き終えたときにのみ返ります。アプリケーションは、レジストリへの変更を絶対に確実にディスク上に反映させる必要がある場合にのみ、`FlushKey()` を呼ぶべきです。

`FlushKey()` を呼び出す必要があるかどうか分からない場合、おそらくその必要はありません。

#### **RegLoadKey(*key*, *sub\_key*, *file\_name*)**

指定されたキーの下にサブキーを生成し、サブキーに指定されたファイルのレジストリ情報を記録します。

*key* はすでに開かれたキーか、既定の HKEY\_\* 定数のうちの一つです。

*sub\_key* は記録先のサブキーを指定する文字列です。

*file\_name* はレジストリデータを読み出すためのファイル名です。このファイルは `SaveKey()` 関数で生成されたファイルでなくてはなりません。ファイル割り当てテーブル (FAT) ファイルシステム下では、ファイル名は拡張子を持っています。

この関数を呼び出しているプロセスが `SE_RESTORE_PRIVILEGE` 特権を持たない場合には `LoadKey()` は失敗します。この特権はファイル許可とは違うので注意してください - 詳細は Win32 ドキュメンテーションを参照してください。

*key* が `ConnectRegistry()` によって返されたハンドルの場合、*file\_name* に指定されたパスは遠隔計算機に対する相対パス名になります。

Win32 ドキュメンテーションでは、*key* は `HKEY_USER` または `HKEY_LOCAL_MACHINE` ツリー内になければならないとされています。これは正しいかもしれないし、そうでないかもしれません。

#### **OpenKey(*key*, *sub\_key* [, *res* = 0] [, *sam* = KEY\_READ])**

指定されたキーを開き、ハンドルオブジェクトを返します。

*key* はすでに開かれたキーか、既定の HKEY\_\* 定数のうちの一つです。

*sub\_key* は開きたいサブキーを特定する文字列です。

*res* 予約されている整数値で、ゼロでなくてはなりません。標準の値はゼロです。

*sam* は必要なキーへのセキュリティアクセスを記述する、アクセスマスクを指定する整数です。標準の値は `KEY_READ` です。

指定されたキーへの新しいハンドルが返されます。

この関数が失敗すると、`EnvironmentError` が送出されます。

#### `OpenKeyEx()`

`OpenKeyEx()` の機能は `OpenKey()` を標準の引数で使うことで提供されています。

#### `QueryInfoKey(key)`

キーに関数情報をタプルとして返します。

`key` はすでに開かれたキーか、既定の `HKEY_*` 定数のうちの一つです。

結果は以下の 3 要素からなるタプルです:

インデクス	意味
0	このキーが持つサブキーの数を表す整数。
1	このキーが持つ値の数を表す整数。
2	最後のキーの変更が (あれば) いつだったかを表す長整数で、1600 年 1 月 1 日からの 100 ナノ秒単位で数えたもの。

#### `QueryValue(key, sub_key)`

キーに対する、名前付けられていない値を文字列で取得します。

`key` はすでに開かれたキーか、既定の `HKEY_*` 定数のうちの一つです。

`sub_key` は値が関連付けられているサブキーの名前を保持する文字列です。この引数が `None` または空文字列の場合、この関数は `key` で特定されるキーに対して `SetValue()` メソッドで設定された値を取得します。

レジストリ中の値は名前、型、およびデータから構成されています。このメソッドはあるキーのデータ中で、名前 `NULL` をもつ最初の値を取得します。しかし背後の API 呼び出しは型情報を返しません。非常に、非常に、非常に不完全な実装です。この関数を使うべきではありません !!!

#### `QueryValueEx(key, value_name)`

開かれたレジストリキーに関連付けられている、指定した名前の値に対して、型およびデータを取得します。

`key` はすでに開かれたキーか、既定の `HKEY_*` 定数のうちの一つです。

`value_name` は要求する値を指定する文字列です。

結果は 2 つの要素からなるタプルです:

インデクス	意味
0	レジストリ要素の名前。
1	この値のレジストリ型を表す整数。

#### `SaveKey(key, file_name)`

指定されたキーと、そのサブキー全てを指定したファイルに保存します。

`key` はすでに開かれたキーか、既定の `HKEY_*` 定数のうちの一つです。

`file_name` はレジストリデータを保存するファイルの名前です、このファイルはすでに存在してはいけません。このファイル名が拡張子を含んでいる場合、`LoadKey()`、`ReplaceKey()` または `RestoreKey()` メソッドは、ファイル割り当てテーブル (FAT) 型ファイルシステムを使うことができません。

`key` が遠隔の計算機上にあるキーを表す場合、`file_name` で記述されているパスは遠隔の計算機に対して相対的なパスになります。このメソッドの呼び出し側は `SeBackupPrivilege` セキュリティ特権を保有していなければなりません。この特権はファイルパーミッションとは異なります - 詳細は Win32 ドキュメンテーションを参照してください。

この関数は `security_attributes` を `NULL` にして API に渡します。

**SetValue**(*key*, *sub\_key*, *type*, *value*)

値を指定したキーに関連付けます。

*key* はすでに開かれたキーか、既定の HKEY\_\* 定数のうちの一つです。

*sub\_key* は値が関連付けられているサブキーの名前を表す文字列です。

*type* はデータの型を指定する整数です。現状では、この値は REG\_SZ でなければならず、これは文字列だけがサポートされていることを示します。他のデータ型をサポートするには SetValueEx() を使ってください。

*value* は新たな値を指定する文字列です。

*sub\_key* 引数で指定されたキーが存在しなければ、SetValue 関数で生成されます。

値の長さは利用可能なメモリによって制限されます。(2048 バイト以上の) 長い値はファイルに保存して、そのファイル名を設定レジストリに保存するべきです。そうすればレジストリを効率的に動作させる役に立ちます。

*key* 引数に指定されたキーは KEY\_SET\_VALUE アクセスで開かれていなければなりません。

**SetValueEx**(*key*, *value\_name*, *reserved*, *type*, *value*)

開かれたレジストリキーの値フィールドにデータを記録します。

*key* はすでに開かれたキーか、既定の HKEY\_\* 定数のうちの一つです。

*sub\_key* は値が関連付けられているサブキーの名前を表す文字列です。

*type* はデータの型を指定する整数です。値はこのモジュールで定義されている以下の定数のうちの一つでなければなりません:

定数	意味
REG_BINARY	何らかの形式のバイナリデータ。
REG_DWORD	32 ビットの数。
REG_DWORD_LITTLE_ENDIAN	32 ビットのリトルエンディアン形式の数。
REG_DWORD_BIG_ENDIAN	32 ビットのビッグエンディアン形式の数。
REG_EXPAND_SZ	環境変数を参照している、ヌル文字で終端された文字列。('%PATH%')。
REG_LINK	Unicode のシンボリックリンク。
REG_MULTI_SZ	ヌル文字で終端された文字列からなり、二つのヌル文字で終端されている配列 (Python はこの終端の処理を自動的に行います)。
REG_NONE	定義されていない値の形式。
REG_RESOURCE_LIST	デバイスドライバリソースのリスト。
REG_SZ	ヌルで終端された文字列。

*reserved* は何もしません - API には常にゼロが渡されます。

*value* は新たな値を指定する文字列です。

このメソッドではまた、指定されたキーに対して、さらに別の値や型情報を設定することができます。*key* 引数で指定されたキーは KEY\_SET\_VALUE アクセスで開かれていなければなりません。

キーを開くには、CreateKeyEx() または OpenKey() メソッドを使ってください。

値の長さは利用可能なメモリによって制限されます。(2048 バイト以上の) 長い値はファイルに保存して、そのファイル名を設定レジストリに保存するべきです。そうすればレジストリを効率的に動作させる役に立ちます。



### 22.2.1 レジストリハンドルオブジェクト

このオブジェクトは Windows の HKEY オブジェクトをラップし、オブジェクトが破壊されたときに自動的にハンドルを閉じます。オブジェクトの `Close()` メソッドと `CloseKey()` 関数のどちらも、後始末がきちんと行われることを保証するために呼び出すことができます。

このモジュールのレジストリ関数は全て、これらのハンドルオブジェクトの一つを返します。

このモジュールのレジストリ関数でハンドルオブジェクトを受け取るものは全て整数も受け取りますが、ハンドルオブジェクトを利用することを推奨します。

ハンドルオブジェクトは `__nonzero__()` の意味構成を持ちます - すなわち、

```
if handle:
    print "Yes"
```

は、ハンドルが現在有効な (閉じられたり切り離されたりしていない) 場合には `Yes` となります。

ハンドルオブジェクトはまた、比較の意味構成もサポートしています。このため、背後の Windows ハンドル値が同じものを複数のハンドルオブジェクトが参照している場合、それらの比較は真になります。

ハンドルオブジェクトは (例えば組み込みの `int()` 関数を使って) 整数に変換することができます。この場合、背後の Windows ハンドル値が返されます、また、`Detach()` 目祖度を使って整数のハンドル値を返させると同時に、ハンドルオブジェクトから Windows ハンドルを切り離すこともできます。

**Close()**

背後の Windows ハンドルを閉じます。

ハンドルがすでに閉じられていてもエラーは送出されません。

**Detach()**

ハンドルオブジェクトから Windows ハンドルを切り離します。

切り離される以前にそのハンドルを保持していた整数 (または 64 ビット Windows の場合には長整数) オブジェクトが返されます。ハンドルがすでに切り離されていたり閉じられていたりした場合、ゼロが返されます。

この関数を呼び出した後、ハンドルは確実に無効化されますが、閉じられるわけではありません。背後の Win32 ハンドルがハンドルオブジェクトよりも長く維持される必要がある場合にはこの関数を呼び出すとよいでしょう。

## 22.3 winsound — Windows 用の音声再生インタフェース

1.5.2 で追加された仕様です。

`winsound` モジュールは Windows プラットフォーム上で提供されている基本的な音声再生機構へのアクセス手段を提供します。このモジュールでは 2 つの関数といくつかの定数が定義されています。

**Beep(*frequency*, *duration*)**

PC のスピーカを鳴らします。引数 *frequency* は鳴らす音の周波数の指定で、単位は Hz です。値は 37 から 32.767 でなくてはなりません。引数 *duration* は音を何ミリ秒鳴らすかの指定です。システムがスピーカを鳴らすことができない場合、例外 `RuntimeError` が送出されます。注意: Windows 95 および 98 では、Windows の関数 `Beep()` は存在しますが役に立ちません (この関数は引数を無視します)。これらのケースでは、Python はポートを直接操作して `Beep()` をシミュレートします (バージョン 2.1 で追加されました)。この機能が全てのシステムで動作するかどうかはわかりません。1.6 で追加された仕様です。

## PlaySound(*sound*, *flags*)

プラットフォームの API から関数 `PlaySound()` を呼び出します。引数 *sound* はファイル名、音声データの文字列、または `None` をとり得ます。*sound* の解釈は *flags* の値に依存します。この値は以下に述べる定数をビット単位で OR して組み合わせたものになります。システムのエラーが発生した場合、例外 `RuntimeError` が送出されます。

## MessageBeep([*type*=`MB_OK`])

根底にある `MessageBeep()` 関数をプラットフォームの API から呼び出します。この関数は音声をレジストリの指定に従って再生します。*type* 引数はどの音声を再生するかを指定します; とり得る値は `-1`、`MB_ICONASTERISK`、`MB_ICONEXCLAMATION`、`MB_ICONHAND`、`MB_ICONQUESTION`、および `MB_OK` で、全て以下に記述されています。値 `-1` は“単純なビープ音”を再生します; この値は他の場合で音声を再生することができなかった際の最終的な代替音です。2.3 で追加された仕様です。

## SND\_FILENAME

*sound* パラメタが WAV ファイル名であることを示します。`SND_ALIAS` と同時に使ってはいけません。

## SND\_ALIAS

引数 *sound* はレジストリにある音声データに関連付けられた名前であることを示します。指定した名前がレジストリ上にない場合、定数 `SND_NODEFAULT` が同時に指定されていない限り、システム標準の音声データが再生されます。標準の音声データが登録されていない場合、例外 `RuntimeError` が送出されます。`SND_FILENAME` と同時に使ってはいけません。

全ての Win32 システムは少なくとも以下の名前をサポートします; ほとんどのシステムでは他に多数あります:

<code>PlaySound()</code> <i>name</i>	対応するコントロールパネルでの音声名
<code>'SystemAsterisk'</code>	Asterisk
<code>'SystemExclamation'</code>	Exclamation
<code>'SystemExit'</code>	Exit Windows
<code>'SystemHand'</code>	Critical Stop
<code>'SystemQuestion'</code>	Question

例えば以下のように使います:

```
import winsound
# Play Windows exit sound.
winsound.PlaySound("SystemExit", winsound.SND_ALIAS)

# Probably play Windows default sound, if any is registered (because
# "*" probably isn't the registered name of any sound).
winsound.PlaySound("*", winsound.SND_ALIAS)
```

## SND\_LOOP

音声データを繰り返し再生します。システムがブロックしないようにするため、`SND_ASYNC` フラグを同時に使わなくてはなりません。`SND_MEMORY` と同時に使うことはできません。

## SND\_MEMORY

`PlaySound()` の引数 *sound* が文字列の形式をとった WAV ファイルのメモリ上のイメージであることを示します。注意: このモジュールはメモリ上のイメージを非同期に再生する機能をサポートしていません。従って、このフラグと `SND_ASYNC` を組み合わせると例外 `RuntimeError` が送出されます。

## SND\_PURGE

指定した音声の全てのインスタンスについて再生処理を停止します。

**SND\_ASYNC**

音声を非同期に再生するようにして、関数呼び出しを即座に返します。

**SND\_NODEFAULT**

指定した音声が見つからなかった場合にシステム標準の音声を鳴らさないようにします。

**SND\_NOSTOP**

現在鳴っている音声を中断させないようにします。

**SND\_NOWAIT**

サウンドドライバがビジー状態にある場合、関数がすぐ返るようにします。

**MB\_ICONASTERISK**

音声 SystemDefault を再生します。

**MB\_ICONEXCLAMATION**

音声 SystemExclamation を再生します。

**MB\_ICONHAND**

音声 SystemHand を再生します。

**MB\_ICONQUESTION**

音声 SystemQuestion を再生します。

**MB\_OK**

音声 SystemDefault を再生します。



# ドキュメント化されていないモジュール

現在ドキュメント化されていないが、ドキュメント化すべきモジュールを以下にざっと列挙します。どうぞこれらのドキュメントを寄稿してください！(電子メールで [docs@python.org](mailto:docs@python.org) に送ってください)。

この章のアイデアと元の文章内容は Fredrik Lundh のポストによるものです; この章の特定の内容は実際には改訂されてきています。

## A.1 フレームワーク

フレームワークは記述するのが難しくなりがちですが、そうする価値はあります。

**test** — 回帰テストフレームワークです。Python 本体の回帰テストのためのものですが、他の Python ライブラリに対しても有用です。単一のモジュールというよりはパッケージです。

## A.2 雑多な有用ユーティリティ

以下のいくつかは非常に古く、かつ / またはあまり頑健ではありません。“hmm.” マーク付きです。

**bdb** — 汎用の Python デバッガ基底クラスです (pdb で使われています)。

**ihooks** — import フックのサポートです (rexec のためのものです; 撤廃されるかもしれません)。

**platform** — このモジュールでは、プラットフォームを識別するためのデータを可能な限り多く取得しようとしします。これにより、関数 API を介して情報を取得できるようにします。コマンドラインから呼び出された場合、このモジュールはプラットフォーム情報を一つの文字列に繋げて `sys.stdout` に出力します。出力形式はファイル名の一部として使うことができます。2.3 で追加された仕様です。

**smtpd** — RFC 821 に適合するための最低限の要求を満たした SMTP デーモン実装です。

## A.3 プラットフォーム特有のモジュール

これらのモジュールは `os.path` モジュールを実装するために用いられていますが、ここで触れる内容を超えてドキュメントされていません。これらはもう少しドキュメント化する必要があります。

**ntpath** — Win32、Win64、WinCE、および OS/2 プラットフォームにおける `os.path` 実装です。

**posixpath** — POSIX における `os.path` 実装です。

**bsddb185** — まだ BerkeleyDB 1.85 を使用しているシステムで後方互換性を保つためのモジュール。通常、特定の BSD Unix ベースのシステムでのみ利用可能。直接使用しないで下さい。

## A.4 マルチメディア関連

**audiodev** — 音声データを再生するためのプラットフォーム非依存の API です。

**linuxaudiodev** — Linux 音声デバイスで音声データを再生します。Python 2.3 では **ossaudiodev** モジュールと置き換えられました。

**sunaudio** — Sun 音声データヘッダを解釈します (撤廃されるか、ツール/デモになるかもしれません)。

**toaiff** — "任意の" 音声ファイルを AIFF ファイルに変換します; おそらくツールかデモになるはずです。外部プログラム **sox** が必要です。

**ossaudiodev** — Open Sound System API を介して音声データを再生します。このモジュールは Linux、いくつかの BSD 系、およびいくつかの商用 UNIX プラットフォームで利用できます。

## A.5 撤廃されたもの

これらのモジュールは通常 `import` して利用できません; 利用できるようにするには作業を行わなければなりません。

Python で書かれたものは、標準ライブラリの一部としてインストールされている 'lib-old/' ディレクトリの中にインストールされます。利用するには、`PYTHONPATH` を使うなどして、'lib-old/' ディレクトリを `sys.path` に追加しなければなりません。

撤廃された拡張モジュールのうち C で書かれたものは、標準の設定ではビルドされません。UNIX でこれらのモジュールを有効にするには、ビルドツリー内の 'Modules/Setup' の適切な行のコメントアウトを外して、モジュールを静的リンクするなら Python をビルドしなおし、動的にロードされる拡張を使うなら共有オブジェクトをビルドしてインストールする必要があります。

**addpack** — パッケージへの別のアプローチです。組み込みのパッケージサポートを使ってください。

**cmp** — ファイル比較関数です。新しい **filecmp** を使ってください。

**cmpcache** — 古い **cmp** モジュールのキャッシュ化版です。 **filecmp** を使ってください。

**codehack** — 関数コードオブジェクトからか数名や行番号を抽出します (現在ではこれらは属性: `co.co_name`, `func.func_name`, `co.co_firstlineno` としてアクセスできます)。

**dircmp** — ディレクトリ間の差分 (diff) ツールを構築するためのクラスです (デモかツールになるかもしれません)。リリース 2.0 以降で撤廃された仕様です。 **filecmp** モジュールが **dircmp** に置き換わります。

**dump** — 変数を再構築する Python コードを出力します。

**fmt** — テキスト書式化処理の抽象化モジュールです (低速すぎます)。

**lockfile** — FCNTL ファイルロック機構へのラッパです (`fcntl.lockf()`/`flock()` を使ってください。; `fcntl` を参照してください)。

**newdir** — 新たな `dir()` 関数です。 (現在では標準の `dir()` が同じくらい改良されました)。

**Para** — **fmt** の補助モジュールです。

**poly** — 多項式です。



**regex** — Emacs 形式の正規表現サポートです; 古いコード内で未だに使われているかもしれません (拡張モジュールです)。ドキュメントは *Python 1.6 Documentation* を参照してください。

**regsub** — **regex** で利用するための、正規表現に基づいた文字列置換ユーティリティです (拡張モジュールです)。ドキュメントは *Python 1.6 Documentation* を参照してください。

**tb** — 局所変数のダンプを伴うトレースバック印字のためのモジュールです (`pdb.pm()` または `traceback` を使ってください)。

**timing** — 高い精度で経過時間を計測します (`time.clock()` を使ってください)。(拡張モジュールです。)

**tzparse** — タイムゾーン指定を解釈します (完成していません; 将来は消滅するモジュールで、TZ 環境変数が設定されていないと動作しません)。

**util** — 他のどこにも分類しようがない有用な関数群です。

**whatsound** — 音声ファイルを識別します; `sndhdr` を使ってください。

**zmod** — 数学的な“体”の性質を計算します。

以下のモジュールは撤廃されましたが、ツールやスクリプトとして復活しているようです:

**find** — ディレクトリツリー内からパターンに合致するファイルを探します。

**grep** — Python による **grep** 実装です。

**packmail** — 自己展開形式の UNIX シェルアーカイブを生成します。

以下のモジュールはこのマニュアルの以前のバージョンでドキュメントされていましたが、現在では撤廃されたものと考えられています。これらのドキュメントのソースは、まだドキュメントソースアーカイブから取得可能です。

**ni** — “パッケージ内の”モジュールを `import` します。今では基本的なパッケージのサポートは組み込みになっています。組み込みのサポートはこのモジュールで提供されていた内容に非常に近いものとなっています。

**rand** — 乱数生成器への古いインタフェースです。

**soundex** — 共有キーと同じような名前縮約アルゴリズムです。特定のアルゴリズムは公開されているもののアルゴリズムとも一致しないようです。(拡張モジュールです。)

## A.6 SGI 特有の拡張モジュール

以下は SGI 特有のモジュールで、現在のバージョンの SGI の実情が反映されていないかもしれません。

**cl** — SGI 圧縮ライブラリへのインタフェースです。

**sv** — SGI Indigo 上の “simple video” ボード (旧式のハードウェアです) へのインタフェースです。



## バグ報告

Python is a mature programming language which has established a reputation for stability. In order to maintain this reputation, the developers would like to know of any deficiencies you find in Python or its documentation.

Before submitting a report, you will be required to log into SourceForge; this will make it possible for the developers to contact you for additional information if needed. It is not possible to submit a bug report anonymously.

All bug reports should be submitted via the Python Bug Tracker on SourceForge ([http://sourceforge.net/bugs/?group\\_id=5470](http://sourceforge.net/bugs/?group_id=5470)). The bug tracker offers a Web form which allows pertinent information to be entered and submitted to the developers.

The first step in filing a report is to determine whether the problem has already been reported. The advantage in doing so, aside from saving the developers time, is that you learn what has been done to fix it; it may be that the problem has already been fixed for the next release, or additional information is needed (in which case you are welcome to provide it if you can!). To do this, search the bug database using the search box near the bottom of the page.

If the problem you're reporting is not already in the bug tracker, go back to the Python Bug Tracker ([http://sourceforge.net/bugs/?group\\_id=5470](http://sourceforge.net/bugs/?group_id=5470)). Select the "Submit a Bug" link at the top of the page to open the bug reporting form.

The submission form has a number of fields. The only fields that are required are the "Summary" and "Details" fields. For the summary, enter a *very* short description of the problem; less than ten words is good. In the Details field, describe the problem in detail, including what you expected to happen and what did happen. Be sure to include the version of Python you used, whether any extension modules were involved, and what hardware and software platform you were using (including version information as appropriate).

The only other field that you may want to set is the "Category" field, which allows you to place the bug report into a broad category (such as "Documentation" or "Library").

Each bug report will be assigned to a developer who will determine what needs to be done to correct the problem. You will receive an update each time action is taken on the bug.

### 参考資料:

*How to Report Bugs Effectively*

(<http://www-mice.cs.ucl.ac.uk/multimedia/software/documentation/ReportingBugs.html>)

Article which goes into some detail about how to create a useful bug report. This describes what kind of information is useful and why it is useful.

*Bug Writing Guidelines*

(<http://www.mozilla.org/quality/bug-writing-guidelines.html>)

Information about writing a good bug report. Some of this is specific to the Mozilla project, but describes general good practices.



# 歴史とライセンス

## C.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see <http://www.cwi.nl/>) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see <http://www.cnri.reston.va.us/>) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see <http://www.zope.com/>). In 2001, the Python Software Foundation (PSF, see <http://www.python.org/psf/>) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see <http://www.opensource.org/> for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.2	2.1.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2.1	2.2	2002	PSF	yes
2.2.2	2.2.1	2002	PSF	yes
2.2.3	2.2.2	2002-2003	PSF	yes
2.3	2.2.2	2002-2003	PSF	yes
2.3.1	2.3	2002-2003	PSF	yes
2.3.2	2.3.1	2003	PSF	yes

注意: GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike

the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

## C.2 Terms and conditions for accessing or otherwise using Python

### **PSF LICENSE AGREEMENT FOR PYTHON 2.3.3**

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.3.3 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.3.3 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2003 Python Software Foundation; All Rights Reserved" are retained in Python 2.3.3 alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.3.3 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.3.3.
4. PSF is making Python 2.3.3 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.3.3 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.3.3 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.3.3, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 2.3.3, Licensee agrees to be bound by the terms and conditions of this License Agreement.

### **BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0 BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1**

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").



2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an “AS IS” basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the “BeOpen Python” logos available at <http://www.pythonlabs.com/logos.html> may be used according to the permissions granted on that web page.
7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

#### **CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1**

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 (“CNRI”), and the Individual or Organization (“Licensee”) accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI’s License Agreement and CNRI’s notice of copyright, i.e., “Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved” are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI’s License Agreement, Licensee may substitute the following text (omitting the quotes): “Python 1.6.1 is made available subject to the terms and conditions in CNRI’s License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: <http://hdl.handle.net/1895.22/1013>.”
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an “AS IS” basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia’s conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By clicking on the “ACCEPT” button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

**CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2**

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

# 日本語訳について

## D.1 このドキュメントについて

この文書は、Python ドキュメント翻訳プロジェクトによる Python Library Reference Release 2.3.3 の日本語訳版です。日本語訳に対する質問や提案などがありましたら、Python ドキュメント翻訳プロジェクトのメーリングリスト

<http://www.python.jp/mailman/listinfo/python-doc-jp>

または、プロジェクトのバグ管理ページ

[http://sourceforge.jp/tracker/?atid=116\&group\\_id=11\&func=browse](http://sourceforge.jp/tracker/?atid=116\&group_id=11\&func=browse)

までご報告ください。

## D.2 翻訳者一覧 (敬称略)

Akihiro Takizawa, Aoki Nobuaki, Atsuo Ishimoto, G.Yoshida, Hiroyuki Yoshimura, Minami Masanori, Shinsei Nakano, Sumiya Sakoda, YASOZUMI Daisuke, Yasushi Iwata, Yasushi MASUDA, Hiroshi Ayukawa, ippei-at-mbd.nifty.com, sakito, umi-at-venus.dti.ne.jp, ふるかわとおる, 浦郷圭介, 梶山大輔, 根岸史郎, 山中裕史, 山本昇, 新山祐介, 森若和雄



# MODULE INDEX

## Symbols

`__builtin__`, 110  
`__future__`, 110  
`__main__`, 110  
`_winreg`, 774

## A

`aifc`, 663  
`AL`, 753  
`al`, 751  
`anydbm`, 399  
`array`, 193  
`asynchat`, 537  
`asyncore`, 534  
`atexit`, 60  
`audioop`, 659

## B

`base64`, 597  
`BaseHTTPServer`, 519  
`Bastion`, 717  
`binascii`, 598  
`binhex`, 599  
`bisect`, 189  
`bsddb`, 402  
`bz2`, 409

## C

`calendar`, 210  
`cd`, 753  
`cgi`, 467  
`CGIHTTPServer`, 523  
`cgilib`, 475  
`chunk`, 671  
`cmath`, 184  
`cmd`, 211  
`code`, 100

`codecs`, 143  
`codeop`, 102  
`colorsys`, 672  
`commands`, 444  
`compileall`, 733  
`compiler`, 743  
`compiler.ast`, 744  
`compiler.visitor`, 749  
`ConfigParser`, 205  
`Cookie`, 524  
`copy`, 91  
`copy_reg`, 88  
`cPickle`, 88  
`crypt`, 428  
`cStringIO`, 140  
`csv`, 606  
`curses`, 277  
`curses.ascii`, 295  
`curses.panel`, 298  
`curses.textpad`, 293  
`curses.wrapper`, 295

## D

`datetime`, 250  
`dbhash`, 400  
`dbm`, 430  
`DEVICE`, 766  
`difflib`, 131  
`dircache`, 242  
`dis`, 734  
`distutils`, 742  
`dl`, 428  
`doctest`, 158  
`DocXMLRPCServer`, 533  
`dumbdbm`, 405  
`dummy_thread`, 396  
`dummy_threading`, 396

## E

email, 547  
email.Charset, 564  
email.Encoders, 567, 568  
email.Generator, 558  
email.Header, 562  
email.Iterators, 571  
email.Message, 548  
email.Parser, 556  
email.Utils, 569  
encodings.idna, 152  
errno, 330  
exceptions, 36

## F

fcntl, 434  
filecmp, 246  
fileinput, 208  
FL, 763  
fl, 757  
flp, 763  
fm, 763  
fnmatch, 336  
formatter, 543  
fpectl, 59  
fpformat, 139  
ftplib, 493

## G

gc, 52  
gdbm, 430  
getopt, 299  
getpass, 276  
gettext, 345  
GL, 766  
gl, 764  
glob, 336  
gopherlib, 497  
grp, 427  
gzip, 408

## H

heapq, 190  
hmac, 681  
hotshot, 459  
hotshot.stats, 460  
htmlentitydefs, 618

htmllib, 616  
HTMLParser, 611  
httplib, 490

## I

imageop, 662  
imaplib, 499  
imgfile, 766  
imghdr, 673  
imp, 96  
inspect, 70  
itertools, 199

## J

jpeg, 767

## K

keyword, 730

## L

linecache, 76  
locale, 339  
logging, 354

## M

mailbox, 580  
mailcap, 579  
marshal, 92  
math, 182  
md5, 682  
mhlib, 582  
mimetools, 584  
mimetypes, 586  
MimeWriter, 588  
mimify, 589  
mmap, 397  
mpz, 684  
msvcrt, 773  
multifile, 590  
mutex, 276

## N

netrc, 605  
new, 107  
nis, 442  
nntplib, 504

## O

operator, 66



optparse, 301  
os, 220  
os.path, 239  
ossaudiodev, 675

## P

parser, 719  
pdb, 445  
pickle, 77  
pipes, 436  
pkgutil, 99  
popen2, 247  
poplib, 497  
posix, 425  
posixfile, 437  
pprint, 103  
profile, 454  
pstats, 455  
pty, 434  
pwd, 426  
py\_compile, 733  
pyclbr, 732  
pydoc, 157

## Q

Queue, 396  
quopri, 600

## R

random, 185  
re, 117  
readline, 421  
repr, 106  
resource, 439  
rexec, 714  
rfc822, 593  
rgbimg, 673  
rlcompleter, 423  
robotparser, 606  
rotor, 685

## S

sched, 274  
ScrolledText, 705  
select, 384  
sets, 196  
sgmllib, 614  
sha, 683

shelve, 89  
shlex, 214  
shutil, 337  
signal, 371  
SimpleHTTPServer, 522  
SimpleXMLRPCServer, 531  
site, 108  
smtplib, 508  
sndhdr, 674  
socket, 374  
SocketServer, 517  
stat, 242  
statcache, 244  
statvfs, 245  
string, 113  
StringIO, 140  
stringprep, 154  
struct, 128  
sunau, 666  
SUNAUDIODEV, 770  
sunaudiodev, 769  
symbol, 729  
sys, 46  
syslog, 443

## T

tabnanny, 731  
tarfile, 415  
telnetlib, 512  
tempfile, 327  
TERMIOS, 433  
termios, 432  
test, 178  
test.test\_support, 180  
textwrap, 141  
thread, 386  
threading, 387  
time, 269  
timeit, 461  
Tix, 699  
Tkinter, 687  
token, 729  
tokenize, 730  
traceback, 74  
tty, 433  
turtle, 705  
types, 61

## U

- unicodedata, 153
- unittest, 166
- urllib, 476
- urllib2, 482
- urlparse, 515
- user, 109
- UserDict, 64
- UserList, 64
- UserString, 65
- uu, 601

## W

- warnings, 93
- wave, 668
- weakref, 54
- webbrowser, 465
- whichdb, 402
- whrandom, 188
- winsound, 779

## X

- xdrlib, 602
- xml.dom, 626
- xml.dom.minidom, 637
- xml.dom.pulldom, 642
- xml.parsers.expat, 618
- xml.sax, 643
- xml.sax.handler, 645
- xml.sax.saxutils, 650
- xml.sax.xmlreader, 651
- xmllib, 655
- xmlrpclib, 528
- xreadlines, 209

## Z

- zipfile, 411
- zlib, 406

# INDEX

## Symbols

.ini  
    file, 205

.pdbrc  
    file, 447

.pythonrc.py  
    file, 109

==  
    演算子, 18  
    operator, 18

\_\_abs\_\_() (operator モジュール), 67

\_\_add\_\_() (AddressList のメソッド), 596

\_\_add\_\_() (operator モジュール), 67

\_\_and\_\_() (operator モジュール), 67

\_\_bases\_\_ (class の属性), 36

\_\_builtin\_\_ (built-in module), 110

\_\_call\_\_() (Generator のメソッド), 560

\_\_class\_\_ (instance の属性), 36

\_\_cmp\_\_() (instance method), 18

\_\_concat\_\_() (operator モジュール), 68

\_\_contains\_\_() (Message のメソッド), 550

\_\_contains\_\_() (operator モジュール), 68

\_\_copy\_\_() (copy protocol), 92

\_\_deepcopy\_\_() (copy protocol), 92

\_\_delitem\_\_() (Message のメソッド), 551

\_\_delitem\_\_() (operator モジュール), 68

\_\_delslice\_\_() (operator モジュール), 68

\_\_dict\_\_ (instance attribute), 82

\_\_dict\_\_ (object の属性), 36

\_\_displayhook\_\_ (sys のデータ), 46

\_\_div\_\_() (operator モジュール), 67

\_\_eq\_\_() (Charset のメソッド), 567

\_\_eq\_\_() (Header のメソッド), 564

\_\_eq\_\_() (operator モジュール), 66

\_\_excepthook\_\_ (sys のデータ), 46

\_\_floordiv\_\_() (operator モジュール), 67

\_\_future\_\_ (standard module), 110

\_\_ge\_\_() (operator モジュール), 66

\_\_getinitargs\_\_() (copy protocol), 82, 92

\_\_getitem\_\_() (Message のメソッド), 550

\_\_getitem\_\_() (operator モジュール), 68

\_\_getslice\_\_() (operator モジュール), 68

\_\_getstate\_\_() (copy protocol), 82, 92

\_\_gt\_\_() (operator モジュール), 66

\_\_iadd\_\_() (AddressList のメソッド), 596

\_\_import\_\_() (モジュール), 3

\_\_init\_\_() (NullTranslations のメソッド), 348

\_\_init\_\_() ( のメソッド), 359

\_\_init\_\_() (instance constructor), 82

\_\_inv\_\_() (operator モジュール), 67

\_\_invert\_\_() (operator モジュール), 67

\_\_isub\_\_() (AddressList のメソッド), 597

\_\_iter\_\_() (container のメソッド), 20

\_\_iter\_\_() (iterator のメソッド), 21

\_\_le\_\_() (operator モジュール), 66

\_\_len\_\_() (AddressList のメソッド), 596

\_\_len\_\_() (Message のメソッド), 550

\_\_lshift\_\_() (operator モジュール), 67

\_\_lt\_\_() (operator モジュール), 66

\_\_main\_\_ (built-in module), 110

\_\_members\_\_ (object の属性), 36

\_\_methods\_\_ (object の属性), 36

\_\_mod\_\_() (operator モジュール), 67

\_\_mul\_\_() (operator モジュール), 67

\_\_ne\_\_() (Header のメソッド), 564, 567

\_\_ne\_\_() (operator モジュール), 66

\_\_neg\_\_() (operator モジュール), 67

\_\_not\_\_() (operator モジュール), 66

\_\_or\_\_() (operator モジュール), 67

\_\_pos\_\_() (operator モジュール), 67

\_\_pow\_\_() (operator モジュール), 67

\_\_repeat\_\_() (operator モジュール), 68

\_\_repr\_\_() (netrc のメソッド), 605

[\\_\\_rshift\\_\\_\(\)](#) (operator モジュール), 68  
[\\_\\_setitem\\_\\_\(\)](#) (Message のメソッド), 550  
[\\_\\_setitem\\_\\_\(\)](#) (operator モジュール), 69  
[\\_\\_setslice\\_\\_\(\)](#) (operator モジュール), 69  
[\\_\\_setstate\\_\\_\(\)](#) (copy protocol), 82, 92  
[\\_\\_stderr\\_\\_](#) (sys のデータ), 51  
[\\_\\_stdin\\_\\_](#) (sys のデータ), 51  
[\\_\\_stdout\\_\\_](#) (sys のデータ), 51  
[\\_\\_str\\_\\_\(\)](#) (AddressList のメソッド), 596  
[\\_\\_str\\_\\_\(\)](#) (Charset のメソッド), 567  
[\\_\\_str\\_\\_\(\)](#) (Header のメソッド), 564  
[\\_\\_str\\_\\_\(\)](#) (Message のメソッド), 549  
[\\_\\_str\\_\\_\(\)](#) (date のメソッド), 255  
[\\_\\_str\\_\\_\(\)](#) (datetime のメソッド), 261  
[\\_\\_str\\_\\_\(\)](#) (time のメソッド), 262  
[\\_\\_sub\\_\\_\(\)](#) (AddressList のメソッド), 597  
[\\_\\_sub\\_\\_\(\)](#) (operator モジュール), 68  
[\\_\\_truediv\\_\\_\(\)](#) (operator モジュール), 68  
[\\_\\_unicode\\_\\_\(\)](#) (Header のメソッド), 564  
[\\_\\_xor\\_\\_\(\)](#) (operator モジュール), 68  
[\\_exit\(\)](#) (os モジュール), 233  
[\\_getframe\(\)](#) (sys モジュール), 48  
[\\_locale](#) (組み込みモジュール), 339  
[\\_parse\(\)](#) (NullTranslations のメソッド), 348  
[\\_structure\(\)](#) (email.Iterators モジュール), 571  
[\\_urlopener](#) (urllib のデータ), 478  
[\\_winreg](#) (extension module), 774  
[% formatting](#), 25  
[% interpolation](#), 25

## A

A-LAW, 665, 674  
[a2b\\_base64\(\)](#) (binascii モジュール), 598  
[a2b\\_hex\(\)](#) (binascii モジュール), 599  
[a2b\\_hqx\(\)](#) (binascii モジュール), 598  
[a2b\\_qp\(\)](#) (binascii モジュール), 598  
[a2b\\_uu\(\)](#) (binascii モジュール), 598  
[ABDAY\\_1 ... ABDAY\\_7](#) (locale のデータ), 342  
[ABMON\\_1 ... ABMON\\_12](#) (locale のデータ), 343  
[abort\(\)](#)  
     FTP のメソッド, 495  
     os モジュール, 232  
[above\(\)](#) ( のメソッド), 298  
[abs\(\)](#)  
     operator モジュール, 67

    モジュール, 4  
[abspath\(\)](#) (os.path モジュール), 239  
[AbstractBasicAuthHandler](#) (urllib2 のクラス), 483  
[AbstractDigestAuthHandler](#) (urllib2 のクラス), 484  
[AbstractFormatter](#) (formatter のクラス), 546  
[AbstractWriter](#) (formatter のクラス), 547  
[ac\\_in\\_buffer\\_size](#) (asyncore のデータ), 535  
[ac\\_out\\_buffer\\_size](#) (asyncore のデータ), 535  
[accept\(\)](#)  
     dispatcher のメソッド, 537  
     socket のメソッド, 379  
[accept2dyear](#) (time のデータ), 270  
[access\(\)](#) (os モジュール), 226  
[acos\(\)](#)  
     cmath モジュール, 184  
     math モジュール, 182  
[acosh\(\)](#) (cmath モジュール), 184  
[acquire\(\)](#)  
     Condition のメソッド, 391  
     Semaphore のメソッド, 392  
[acquire\(\)](#) (Timer のメソッド), 389, 390  
[acquire\(\)](#)  
     のメソッド, 359  
     lock のメソッド, 387  
[acquire\\_lock\(\)](#) (imp モジュール), 97  
[ACTIONS](#) ( の属性), 324  
[activate\\_form\(\)](#) (form のメソッド), 759  
[activeCount\(\)](#) (threading モジュール), 388  
[add\(\)](#)  
     audioop モジュール, 659  
     operator モジュール, 67  
     Stats のメソッド, 456  
     TarFile のメソッド, 418  
[add\\_alias\(\)](#) (email.Charset モジュール), 567  
[add\\_box\(\)](#) (form のメソッド), 760  
[add\\_browser\(\)](#) (form のメソッド), 761  
[add\\_button\(\)](#) (form のメソッド), 760  
[add\\_charset\(\)](#) (email.Charset モジュール), 567  
[add\\_choice\(\)](#) (form のメソッド), 761  
[add\\_clock\(\)](#) (form のメソッド), 760  
[add\\_codec\(\)](#) (email.Charset モジュール), 567  
[add\\_counter\(\)](#) (form のメソッド), 760  
[add\\_data\(\)](#) (Request のメソッド), 484  
[add\\_dial\(\)](#) (form のメソッド), 760

add\_fallback() (NullTranslations のメソッド), 348  
 add\_flow\_data() (formatter のメソッド), 544  
 add\_handler() (OpenerDirector のメソッド), 485  
 add\_header()  
     Message のメソッド, 551  
     Request のメソッド, 485  
 add\_history() (readline モジュール), 422  
 add\_hor\_rule() (formatter のメソッド), 544  
 add\_input() (form のメソッド), 761  
 add\_label\_data() (formatter のメソッド), 544  
 add\_lightbutton() (form のメソッド), 760  
 add\_line\_break() (formatter のメソッド), 544  
 add\_literal\_data() (formatter のメソッド), 544  
 add\_menu() (form のメソッド), 761  
 add\_parent() (BaseHandler のメソッド), 486  
 add\_password() (HTTPPasswordMgr のメソッド), 488  
 add\_payload() (Message のメソッド), 555  
 add\_positioner() (form のメソッド), 760  
 add\_roundbutton() (form のメソッド), 760  
 add\_section() (SafeConfigParser のメソッド), 206  
 add\_slider() (form のメソッド), 760  
 add\_text() (form のメソッド), 760  
 add\_timer() (form のメソッド), 761  
 add\_type() (mimetypes モジュール), 587  
 add\_valslider() (form のメソッド), 760  
 addcallback() (CD parser のメソッド), 756  
 addch() (window のメソッド), 283  
 addError() (TestResult のメソッド), 176  
 addFailure() (TestResult のメソッド), 176  
 addfile() (TarFile のメソッド), 418  
 addFilter() ( のメソッド), 358, 359  
 addHandler() ( のメソッド), 358  
 addheader() (MimeWriter のメソッド), 588  
 addinfo() (Profile のメソッド), 460  
 addLevelName() (logging モジュール), 357  
 addnstr() (window のメソッド), 283  
 address\_family (SocketServer のデータ), 517  
 address\_string() (BaseHTTPRequestHandler のメソッド), 522  
 AddressList (rfc822 のクラス), 593  
 addresslist (AddressList の属性), 597  
 addstr() (window のメソッド), 283  
 addSuccess() (TestResult のメソッド), 176  
 addTest() (TestSuite のメソッド), 175  
 addTests() (TestSuite のメソッド), 175  
 adler32() (zlib モジュール), 406  
 ADPCM, Intel/DVI, 659  
 adpcm2lin() (audioop モジュール), 659  
 adpcm32lin() (audioop モジュール), 660  
 AF\_INET (socket のデータ), 375  
 AF\_INET6 (socket のデータ), 375  
 AF\_UNIX (socket のデータ), 375  
 AI\_\* (socket のデータ), 376  
 aifc() (aifc のメソッド), 665  
 aifc (standard module), 663  
 AIFF, 663, 671  
 aiff() (aifc のメソッド), 665  
 AIFF-C, 663, 671  
 AL  
     standard module, 753  
     標準モジュール, 751  
 al (built-in module), 751  
 alarm() (signal モジュール), 372  
 all\_errors (ftplib のデータ), 494  
 all\_features (xml.sax.handler のデータ), 646  
 all\_properties (xml.sax.handler のデータ), 646  
 allocate\_lock() (thread モジュール), 386  
 allow\_reuse\_address (SocketServer のデータ), 518  
 allowremoval() (CD player のメソッド), 755  
 alt() (curses.ascii モジュール), 297  
 ALT\_DIGITS (locale のデータ), 343  
 altsep (os のデータ), 238  
 altzone (time のデータ), 270  
 anchor\_bgn() (HTMLParser のメソッド), 617  
 anchor\_end() (HTMLParser のメソッド), 618  
 and  
     演算子, 17, 18  
     operator, 17, 18  
 and\_() (operator モジュール), 67  
 annotate() (dircache モジュール), 242  
 anydbm (standard module), 399  
 api\_version (sys のデータ), 52  
 apop() (POP3 のメソッド), 498  
 append()  
     array のメソッド, 194  
     Header のメソッド, 563

- IMAP4\_stream のメソッド, 501
- list method, 28
- Template のメソッド, 437
- appendChild() (Node のメソッド), 630
- apply() (モジュール), 16
- arbitrary precision integers, 684
- aRepr (repr のデータ), 106
- argv (sys のデータ), 46
- arithmetic, 19
- ArithmeticError (exceptions の例外), 37
- array() (array モジュール), 193
- array (built-in module), 193
- arrays, 193
- ArrayType (array のデータ), 193
- article() (NNTPDataError のメソッド), 507
- AS\_IS (formatter のデータ), 544
- as\_string() (Message のメソッド), 548
- ascii() (curses.ascii モジュール), 297
- ascii\_letters (string のデータ), 113
- ascii\_lowercase (string のデータ), 113
- ascii\_uppercase (string のデータ), 113
- asctime() (time モジュール), 270
- asin()
  - cmath モジュール, 184
  - math モジュール, 182
- asinh() (cmath モジュール), 184
- assert
  - 実行文, 38
  - statement, 38
- assert\_() (TestCase のメソッド), 174
- assert\_line\_data() (formatter のメソッド), 545
- assertAlmostEqual() (TestCase のメソッド), 174
- assertEqual() (TestCase のメソッド), 174
- AssertionError (exceptions の例外), 38
- assertNotAlmostEqual() (TestCase のメソッド), 174
- assertNotEqual() (TestCase のメソッド), 174
- assertRaises() (TestCase のメソッド), 175
- assignment
  - extended slice, 28
  - slice, 28
  - subscript, 28
- ast2list() (parser モジュール), 721
- ast2tuple() (parser モジュール), 721
- astimezone() (datetime のメソッド), 259
- ASTType (parser のデータ), 722
- ASTVisitor (compiler.visitor のクラス), 750
- async\_chat (asynchat のクラス), 538
- asynchat (standard module), 537
- asyncore (built-in module), 534
- atan()
  - cmath モジュール, 184
  - math モジュール, 182
- atan2() (math モジュール), 182
- atanh() (cmath モジュール), 184
- atexit (standard module), 60
- atime (cd のデータ), 755
- atof()
  - locale モジュール, 341
  - string モジュール, 114
- atoi()
  - locale モジュール, 341
  - string モジュール, 114
- atol() (string モジュール), 114
- attach() (Message のメソッド), 549
- AttlistDeclHandler() (xmlparser のメソッド), 622
- AttributeError (exceptions の例外), 38
- attributes
  - Node の属性, 629
  - XMLParser の属性, 655
- AttributesImpl (xml.sax.xmlreader のクラス), 651
- AttributesNSImpl (xml.sax.xmlreader のクラス), 651
- attroff() (window のメソッド), 284
- attron() (window のメソッド), 284
- attrset() (window のメソッド), 284
- audio (cd のデータ), 754
- Audio Interchange File Format, 663, 671
- AUDIO\_FILE\_ENCODING\_ADPCM\_G721 (sunau のデータ), 667
- AUDIO\_FILE\_ENCODING\_ADPCM\_G722 (sunau のデータ), 667
- AUDIO\_FILE\_ENCODING\_ADPCM\_G723\_3 (sunau のデータ), 667
- AUDIO\_FILE\_ENCODING\_ADPCM\_G723\_5 (sunau のデータ), 667
- AUDIO\_FILE\_ENCODING\_ALAW\_8 (sunau のデータ), 667
- AUDIO\_FILE\_ENCODING\_DOUBLE (sunau のデータ), 667



AUDIO\_FILE\_ENCODING\_FLOAT (sunau のデータ), 667

AUDIO\_FILE\_ENCODING\_LINEAR\_16 (sunau のデータ), 666

AUDIO\_FILE\_ENCODING\_LINEAR\_24 (sunau のデータ), 666

AUDIO\_FILE\_ENCODING\_LINEAR\_32 (sunau のデータ), 667

AUDIO\_FILE\_ENCODING\_LINEAR\_8 (sunau のデータ), 666

AUDIO\_FILE\_ENCODING\_MULAW\_8 (sunau のデータ), 666

AUDIO\_FILE\_MAGIC (sunau のデータ), 666

AUDIODEV, 675

audioop (built-in module), 659

authenticate() (IMAP4\_stream のメソッド), 501

authenticators() (netrc のメソッド), 605

avg() (audioop モジュール), 660

avgpp() (audioop モジュール), 660

## B

b2a\_base64() (binascii モジュール), 598

b2a\_hex() (binascii モジュール), 599

b2a\_hqx() (binascii モジュール), 598

b2a\_qp() (binascii モジュール), 598

b2a\_uu() (binascii モジュール), 598

BabylMailbox (mailbox のクラス), 581

backslashreplace\_errors\_errors() (codecs モジュール), 144

backward() (turtle モジュール), 706

BadStatusLine (httplib の例外), 491

Balloon (Tix のクラス), 700

base64

- encoding, 597

base64 (standard module), 597

BaseCookie (Cookie のクラス), 524

BaseHandler (urllib2 のクラス), 483

BaseHTTPRequestHandler (BaseHTTPServer のクラス), 519

BaseHTTPServer (standard module), 519

basename() (os.path モジュール), 239

basestring() (モジュール), 4

basicConfig() (logging モジュール), 357

Bastion() (Bastion モジュール), 717

Bastion (standard module), 717

BastionClass (Bastion のクラス), 718

baudrate() (curses モジュール), 277

bdb (標準モジュール), 445

Beep() (winsound モジュール), 779

beep() (curses モジュール), 278

below() ( のメソッド), 298

Benchmarking, 461

benchmarking, 270

bestreadsize() (CD player のメソッド), 755

betavariate() (random モジュール), 187

bgn\_group() (form のメソッド), 760

bias() (audioop モジュール), 660

bidirectional() (unicodedata モジュール), 153

binary

- data, packing, 128

binary()

- mpz のメソッド, 685
- xmlrpclib モジュール, 531

binary semaphores, 386

binascii (built-in module), 598

bind()

- dispatcher のメソッド, 537
- socket のメソッド, 379

bind (widgets), 697

bindtextdomain() (gettext モジュール), 345

binhex() (binhex モジュール), 600

binhex

- standard module, 599
- 標準モジュール, 598

bisect() (bisect モジュール), 189

bisect (standard module), 189

bisect\_left() (bisect モジュール), 189

bisect\_right() (bisect モジュール), 189

bit-string

- operations, 20

bkgd() (window のメソッド), 284

bkgdset() (window のメソッド), 284

BLOCKSIZE (cd のデータ), 754

blocksize (sha のデータ), 683

body() (NNTPDataError のメソッド), 507

body\_encode() (Charset のメソッド), 566

body\_encoding (email.Charset のデータ), 565

body\_line\_iterator() (email.Iterators モジュール), 571

BOM (codecs のデータ), 145

BOM\_BE (codecs のデータ), 145

BOM\_LE (codecs のデータ), 145

BOM\_UTF16 (codecs のデータ), 145  
BOM\_UTF16\_BE (codecs のデータ), 145  
BOM\_UTF16\_LE (codecs のデータ), 145  
BOM\_UTF32 (codecs のデータ), 145  
BOM\_UTF32\_BE (codecs のデータ), 145  
BOM\_UTF32\_LE (codecs のデータ), 145  
BOM\_UTF8 (codecs のデータ), 145  
bool() (モジュール), 4  
Boolean  
    object, 19  
    オブジェクト, 19  
    operations, 17  
    type, 4  
    values, 36  
boolean() (xmlrpclib モジュール), 531  
BooleanType (types のデータ), 62  
border() (window のメソッド), 284  
bottom() ( のメソッド), 298  
bottom\_panel() (curses.panel モジュール), 298  
BoundedSemaphore() (threading モジュール), 388  
box() (window のメソッド), 284  
break\_long\_words (TextWrapper の属性), 143  
BROWSER, 466  
bsddb  
    extension module, 402  
    組み込みモジュール, 89  
BsdDbShelf (shelve のクラス), 90  
btopen() (bsddb モジュール), 403  
buffer  
    object, 21  
    オブジェクト, 21  
buffer()  
    モジュール, 16  
    組み込み関数, 21, 63  
buffer size, I/O, 8  
buffer\_info() (array のメソッド), 194  
buffer\_size (xmlparser の属性), 620  
buffer\_text (xmlparser の属性), 620  
buffer\_used (xmlparser の属性), 620  
BufferingHandler (logging のクラス), 364  
BufferType (types のデータ), 63  
bufsize() (audio device のメソッド), 677  
build\_opener() (urllib2 モジュール), 482  
built-in  
    constants, 3  
    exceptions, 3

functions, 3  
types, 3, 17

builtin\_module\_names (sys のデータ), 46  
BuiltinFunctionType (types のデータ), 63  
BuiltinMethodType (types のデータ), 63  
ButtonBox (Tix のクラス), 701  
byte-code  
    file, 96, 98, 733  
byteorder (sys のデータ), 46  
byteswap() (array のメソッド), 194  
bz2 (built-in module), 409  
BZ2Compressor (bz2 のクラス), 410  
BZ2Decompressor (bz2 のクラス), 411  
BZ2File (bz2 のクラス), 409

## C

C  
    language, 19, 20  
    structures, 128  
C\_BUILTIN (imp のデータ), 97  
C\_EXTENSION (imp のデータ), 97  
CacheFTPHandler (urllib2 のクラス), 484  
calcsite() (struct モジュール), 128  
calendar() (calendar モジュール), 211  
calendar (standard module), 210  
call() ( のメソッド), 429  
callable() (モジュール), 4  
CallableProxyType (weakref のデータ), 56  
can\_change\_color() (curses モジュール), 278  
can\_fetch() (RobotFileParser のメソッド), 606  
cancel()  
    scheduler のメソッド, 275  
    Timer のメソッド, 396  
CannotSendHeader (httplib の例外), 491  
CannotSendRequest (httplib の例外), 491  
capitalize()  
    string のメソッド, 23  
    string モジュール, 115  
capwords() (string モジュール), 115  
cat() (nis モジュール), 443  
catalog (cd のデータ), 755  
category() (unicodedata モジュール), 153  
cbreak() (curses モジュール), 278  
cd (built-in module), 753  
CDROM (cd のデータ), 754  
ceil()  
    in module math, 20

- math モジュール, 182
- center()
  - string のメソッド, 23
  - string モジュール, 116
- CGI
  - debugging, 474
  - exceptions, 475
  - protocol, 467
  - security, 473
  - tracebacks, 475
- cgi (standard module), 467
- cgi\_directories (CGIHTTPRequestHandler の属性), 523
- CGIHTTPRequestHandler (CGIHTTPServer のクラス), 523
- CGIHTTPServer
  - standard module, 523
  - 標準モジュール, 519
- cgitb (standard module), 475
- CGIXMLRPCRequestHandler (SimpleXMLRPCServer のクラス), 531
- chain() (itertools モジュール), 199
- chaining
  - comparisons, 18
- channels() (audio device のメソッド), 677
- CHAR\_MAX (locale のデータ), 342
- character, 153
- CharacterDataHandler() (xmlparser のメソッド), 622
- characters() (ContentHandler のメソッド), 648
- CHARSET (mimify のデータ), 590
- Charset (email.Charset のクラス), 564
- charset() (NullTranslations のメソッド), 348
- chdir() (os モジュール), 226
- check()
  - IMAP4\_stream のメソッド, 501
  - tabnanny モジュール, 731
- check\_forms() (fl モジュール), 758
- checkcache() (linecache モジュール), 76
- CheckList (Tix のクラス), 702
- checksum
  - Cyclic Redundancy Check, 406
  - MD5, 682
  - SHA, 683
- childerr (Popen4 の属性), 249
- childNodes (Node の属性), 629
- chmod() (os モジュール), 226
- choice()
  - random モジュール, 186
  - whrandom モジュール, 188
- choose\_boundary() (mimetools モジュール), 584
- chown() (os モジュール), 227
- chr() (モジュール), 4
- chroot() (os モジュール), 226
- Chunk (chunk のクラス), 671
- chunk (standard module), 671
- cipher
  - DES, 428, 681
  - Enigma, 685
  - IDEA, 681
- circle() (turtle モジュール), 706
- Class browser, 707
- classmethod() (モジュール), 4
- classobj() (new モジュール), 108
- ClassType (types のデータ), 63
- clear()
  - dictionary method, 29
  - Event のメソッド, 393
  - turtle モジュール, 705
  - window のメソッド, 284
- clear\_memo() (Pickler のメソッド), 80
- clearcache() (linecache モジュール), 76
- clearok() (window のメソッド), 284
- client\_address (BaseHTTPRequestHandler の属性), 520
- clock() (time モジュール), 270
- clone()
  - Generator のメソッド, 559
  - Template のメソッド, 437
- cloneNode()
  - のメソッド, 640
  - Node のメソッド, 630
- Close() (のメソッド), 779
- close()
  - のメソッド, 360, 398, 403
  - AU\_read のメソッド, 667
  - AU\_write のメソッド, 668
  - BaseHandler のメソッド, 486
  - BZ2File のメソッド, 409
  - CD player のメソッド, 755
  - Chunk のメソッド, 671
  - FileHandler のメソッド, 361

- FTP のメソッド, 497
- HTMLParser のメソッド, 612
- HTTPSConnection のメソッド, 492
- IMAP4\_stream のメソッド, 501
- IncrementalParser のメソッド, 653
- MemoryHandler のメソッド, 364
- NTEventLogHandler のメソッド, 363
- OpenerDirector のメソッド, 485
- Profile のメソッド, 460
- SGMLParser のメソッド, 614
- SocketHandler のメソッド, 361
- StringIO のメソッド, 140
- SysLogHandler のメソッド, 362
- TarFile のメソッド, 418
- Telnet のメソッド, 514
- Wave\_read のメソッド, 669
- Wave\_write のメソッド, 670
- XMLParser のメソッド, 656
- ZipFile のメソッド, 412 429
- close()
  - aifc のメソッド, 665
  - audio device のメソッド, 676, 769
  - dispatcher のメソッド, 537
  - file のメソッド, 31
  - fileinput モジュール, 209
  - mixer device のメソッド, 678
  - os モジュール, 223
  - socket のメソッド, 379
- close\_when\_done() (async\_chat のメソッド), 538
- closed (file の属性), 33
- CloseKey() (\_winreg モジュール), 775
- closelog() (syslog モジュール), 443
- closeport() (audio port のメソッド), 752
- clrrobot() (window のメソッド), 284
- clrtoeol() (window のメソッド), 285
- cmath (built-in module), 184
- Cmd (cmd のクラス), 211
- cmd
  - standard module, 211
  - 標準モジュール, 445
- cmdloop() (Cmd のメソッド), 212
- cmp()
  - filecmp モジュール, 246
  - モジュール, 5
  - 組み込み関数, 341
- cmp\_op (dis のデータ), 735
- cmpfiles() (filecmp モジュール), 246
- code
  - object, 34, 35, 92
  - オブジェクト, 34, 35, 92
- code() (new モジュール), 108
- code
  - ExpatriError の属性, 623
  - standard module, 100
- Codecs, 143
  - decode, 143
  - encode, 143
- codecs (standard module), 143
- coded\_value (Morsel の属性), 525
- codeop (standard module), 102
- codepoint2name (htmlentitydefs のデータ), 618
- CODESET (locale のデータ), 342
- CodeType (types のデータ), 63
- coerce() (モジュール), 16
- collect() (gc モジュール), 52
- collect\_incoming\_data() (async\_chat のメソッド), 538
- color()
  - fl モジュール, 759
  - turtle モジュール, 706
- color\_content() (curses モジュール), 278
- color\_pair() (curses モジュール), 278
- colorsys (standard module), 672
- COLUMNS, 283
- combine() (datetime のメソッド), 257
- combining() (unicodedata モジュール), 153
- ComboBox (Tix のクラス), 701
- command (BaseHTTPRequestHandler の属性), 520
- CommandCompiler (codeop のクラス), 103
- commands (standard module), 444
- COMMENT (tokenize のデータ), 731
- comment (ZipInfo の属性), 414
- commenters (shlex の属性), 216
- CommentHandler() (xmlparser のメソッド), 623
- common (dircmp の属性), 247
- Common Gateway Interface, 467
- common\_dirs (dircmp の属性), 247
- common\_files (dircmp の属性), 247
- common\_funny (dircmp の属性), 247
- common\_types (mimetypes のデータ), 587, 588
- commonprefix() (os.path モジュール), 239
- compare() (Differ のメソッド), 138

- comparing
  - objects, 18
- comparison
  - operator, 18
- comparisons
  - chaining, 18
- Compile (codeop のクラス), 103
- compile()
  - AST のメソッド, 723
  - compiler モジュール, 743
  - py\_compile モジュール, 733
  - re モジュール, 122
  - モジュール, 5
  - 組み込み関数, 35, 63
- compile\_command()
  - codeop モジュール, 102
  - code モジュール, 100
- compile\_dir() (compileall モジュール), 733
- compile\_path() (compileall モジュール), 734
- compileall (standard module), 733
- compileast() (parser モジュール), 721
- compileFile() (compiler モジュール), 744
- compiler (module), 743
- compiler.ast (module), 744
- compiler.visitor (module), 749
- complete() (Completer のメソッド), 423
- completedefault() (Cmd のメソッド), 213
- complex()
  - モジュール, 5
  - 組み込み関数, 19
- complex number
  - literals, 19
  - object, 19
  - オブジェクト, 19
- ComplexType (types のデータ), 62
- compress()
  - BZ2Compressor のメソッド, 410
  - bz2 モジュール, 411
  - Compress のメソッド, 407
  - jpeg モジュール, 767
  - zlib モジュール, 406
- compress\_size (ZipInfo の属性), 415
- compress\_type (ZipInfo の属性), 414
- CompressionError (tarfile の例外), 416
- compressobj() (zlib モジュール), 406
- COMSPEC, 237
- concat() (operator モジュール), 68
- concatenation
  - operation, 22
- Condition() (threading モジュール), 388
- Condition (threading のクラス), 391
- ConfigParser
  - ConfigParser のクラス, 205
  - standard module, 205
- configuration
  - file, 205
  - file, debugger, 447
  - file, path, 108
  - file, user, 109
- confstr() (os モジュール), 238
- confstr\_names (os のデータ), 238
- conjugate() (complex number method), 19
- connect()
  - dispatcher のメソッド, 536
  - FTP のメソッド, 495
  - HTTPConnection のメソッド, 492
  - SMTP のメソッド, 509
  - socket のメソッド, 379
- connect\_ex() (socket のメソッド), 379
- ConnectRegistry() (\_winreg モジュール), 775
- constants
  - built-in, 3
- constructor() (copy\_reg モジュール), 88
- container
  - iteration over, 20
- contains() (operator モジュール), 68
- content type
  - MIME, 586
- ContentHandler (xml.sax.handler のクラス), 645
- context\_diff() (difflib モジュール), 132
- Control (Tix のクラス), 701
- control (cd のデータ), 755
- controlnames (curses.ascii のデータ), 298
- controls() (mixer device のメソッド), 678
- ConversionError (xdrlib の例外), 604
- conversions
  - numeric, 20
- convert() (Charset のメソッド), 566
- Cookie (standard module), 524
- CookieError (Cookie の例外), 524
- Coordinated Universal Time, 269
- copy()

- hmac のメソッド, 681
- IMAP4\_stream のメソッド, 501
- md5 のメソッド, 683
- sha のメソッド, 683
- shutil モジュール, 337
- Template のメソッド, 437
- copy
  - standard module, 91
  - 標準モジュール, 88
- copy()
  - dictionary method, 29
  - in copy, 91
- copy2() (shutil モジュール), 337
- copy\_reg (standard module), 88
- copybinary() (mimetools モジュール), 585
- copyfile() (shutil モジュール), 337
- copyfileobj() (shutil モジュール), 337
- copying files, 337
- copyliteral() (mimetools モジュール), 585
- copymessage() (Folder のメソッド), 584
- copymode() (shutil モジュール), 337
- copyright (sys のデータ), 46
- copystat() (shutil モジュール), 337
- copytree() (shutil モジュール), 338
- cos()
  - cmath モジュール, 184
  - math モジュール, 182
- cosh()
  - cmath モジュール, 184
  - math モジュール, 182
- count()
  - array のメソッド, 194
  - itertools モジュール, 199
  - list method, 28
  - string のメソッド, 23
  - string モジュール, 115
- countOf() (operator モジュール), 68
- countTestCases() (TestCase のメソッド), 175
- cPickle
  - built-in module, 88
  - 組み込みモジュール, 88
- CPU time, 270
- CRC (ZipInfo の属性), 415
- crc32()
  - binascii モジュール, 599
  - zlib モジュール, 406
- crc\_hqx() (binascii モジュール), 599
- create() (IMAP4\_stream のメソッド), 501
- create\_socket() (dispatcher のメソッド), 536
- create\_system (ZipInfo の属性), 414
- create\_version (ZipInfo の属性), 414
- createAttribute() (Document のメソッド), 632
- createAttributeNS() (Document のメソッド), 632
- createComment() (Document のメソッド), 632
- createElement() (Document のメソッド), 632
- createElementNS() (Document のメソッド), 632
- CreateKey() (\_winreg モジュール), 775
- createLock() ( のメソッド), 359
- createparser() (cd モジュール), 754
- createProcessingInstruction() (Document のメソッド), 632
- createTextNode() (Document のメソッド), 632
- critical()
  - のメソッド, 358
  - logging モジュール, 356
- CRNCYSTR (locale のデータ), 343
- crop() (imageop モジュール), 662
- cross() (audioop モジュール), 660
- crypt() (crypt モジュール), 428
- crypt
  - built-in module, 428
  - 組み込みモジュール, 427
- crypt(3), 428
- cryptography, 681
- cStringIO (built-in module), 140
- csv, 606
- csv (standard module), 606
- ctermid() (os モジュール), 221
- ctime()
  - date のメソッド, 255
  - datetime のメソッド, 261
  - time モジュール, 271
- ctrl() (curses.ascii モジュール), 297
- cunifvariate() (random モジュール), 187
- curdir (os のデータ), 238
- currentThread() (threading モジュール), 388
- curs\_set() (curses モジュール), 278
- curses (standard module), 277
- curses.ascii (standard module), 295
- curses.panel (standard module), 298



`curses.textpad` (standard module), [293](#)  
`curses.wrapper` (standard module), [295](#)  
`cursyncup()` (window のメソッド), [285](#)  
`cwd()` (FTP のメソッド), [496](#)  
`cycle()` (itertools モジュール), [200](#)  
Cyclic Redundancy Check, [406](#)

## D

`D_FMT` (locale のデータ), [342](#)  
`D_T_FMT` (locale のデータ), [342](#)  
`data`  
    packing binary, [128](#)  
    tabular, [606](#)  
`data`  
    Binary の属性, [530](#)  
    Comment の属性, [634](#)  
    MutableString の属性, [66](#)  
    ProcessingInstruction の属性, [635](#)  
    Text の属性, [634](#)  
    UserDict の属性, [64](#)  
    UserList の属性, [65](#)  
`database`  
    Unicode, [153](#)  
`databases`, [405](#)  
`DatagramHandler` (logging のクラス), [362](#)  
`DATASIZE` (cd のデータ), [754](#)  
`date()`  
    datetime のメソッド, [259](#)  
    NNTPDataError のメソッド, [508](#)  
`date` (datetime のクラス), [250](#), [253](#)  
`date_time` (ZipInfo の属性), [414](#)  
`date_time_string()` (BaseHTTPRequestHandler のメソッド), [521](#)  
`datetime`  
    built-in module, [250](#)  
    datetime のクラス, [251](#), [256](#)  
`day`  
    date の属性, [254](#)  
    datetime の属性, [257](#)  
`DAY_1 ... DAY_7` (locale のデータ), [342](#)  
`daylight` (time のデータ), [271](#)  
Daylight Saving Time, [269](#)  
`DbfilenameShelf` (shelve のクラス), [90](#)  
`dbhash`  
    standard module, [400](#)  
    標準モジュール, [399](#)  
`dbm`

    built-in module, [430](#)  
    組み込みモジュール, [89](#), [399](#), [430](#)  
`deactivate_form()` (form のメソッド), [759](#)  
`debug()`  
    のメソッド, [358](#)  
    doctest モジュール, [163](#)  
    logging モジュール, [356](#)  
    Template のメソッド, [437](#)  
    TestCase のメソッド, [174](#)  
`debug`  
    IMAP4\_stream の属性, [504](#)  
    shlex の属性, [216](#)  
    ZipFile の属性, [413](#)  
`debug=0` (TarFile の属性), [418](#)  
`DEBUG_COLLECTABLE` (gc のデータ), [54](#)  
`DEBUG_INSTANCES` (gc のデータ), [54](#)  
`DEBUG_LEAK` (gc のデータ), [54](#)  
`DEBUG_OBJECTS` (gc のデータ), [54](#)  
`DEBUG_SAVEALL` (gc のデータ), [54](#)  
`DEBUG_STATS` (gc のデータ), [54](#)  
`DEBUG_UNCOLLECTABLE` (gc のデータ), [54](#)  
`debugger`, [51](#), [709](#)  
    configuration file, [447](#)  
`debugging`, [445](#)  
    CGI, [474](#)  
`decimal()` (unicodedata モジュール), [153](#)  
`decode`  
    Codecs, [143](#)  
`decode()`  
    のメソッド, [146](#)  
    base64 モジュール, [597](#)  
    Binary のメソッド, [530](#)  
    email.Utils モジュール, [570](#)  
    mimetools モジュール, [584](#)  
    quopri モジュール, [600](#)  
    ServerProxy のメソッド, [530](#)  
    string のメソッド, [23](#)  
    uu モジュール, [601](#)  
`decode_header()` (email.Header モジュール), [564](#)  
`decode_params()` (email.Utils モジュール), [570](#)  
`decode_rfc2231()` (email.Utils モジュール), [570](#)  
`DecodedGenerator` (email.Generator のクラス), [559](#)  
`decodestring()`  
    base64 モジュール, [597](#)

quopri モジュール, 601  
 decomposition() (unicodedata モジュール), 153  
 decompress()  
     BZ2Decompressor のメソッド, 411  
     bz2 モジュール, 411  
     Decompress のメソッド, 407  
     jpeg モジュール, 768  
     zlib モジュール, 406  
 decompressobj() (zlib モジュール), 407  
 decrypt() (rotor のメソッド), 686  
 decryptmore() (rotor のメソッド), 686  
 dedent() (textwrap モジュール), 141  
 deepcopy() (in copy), 91  
 def\_prog\_mode() (curses モジュール), 278  
 def\_shell\_mode() (curses モジュール), 278  
 default()  
     ASTVisitor のメソッド, 750  
     Cmd のメソッド, 213  
 default\_bufsize (xml.dom.pulldom のデータ), 642  
 default\_open() (BaseHandler のメソッド), 486  
 DefaultHandler() (xmlparser のメソッド), 623  
 DefaultHandlerExpand() (xmlparser のメソッド), 623  
 defaults() (SafeConfigParser のメソッド), 206  
 defaultTestLoader (unittest のデータ), 173  
 defaultTestResult() (TestCase のメソッド), 175  
 defpath (os のデータ), 239  
 degrees()  
     math モジュール, 182  
     RawPen のメソッド, 707  
     turtle モジュール, 705  
 del  
     実行文, 28, 29  
     statement, 28, 29  
 del\_param() (Message のメソッド), 553  
 delattr() (モジュール), 5  
 delay\_output() (curses モジュール), 278  
 delch() (window のメソッド), 285  
 dele() (POP3 のメソッド), 499  
 delete()  
     FTP のメソッド, 496  
     IMAP4\_stream のメソッド, 501  
     delete\_object() (FORMS object のメソッド), 762  
     deletefolder() (MH のメソッド), 583  
     DeleteKey() (\_winreg モジュール), 775  
     deleteln() (window のメソッド), 285  
     deletparser() (CD parser のメソッド), 757  
     DeleteValue() (\_winreg モジュール), 775  
     delimiter (Dialect の属性), 609  
     delitem() (operator モジュール), 68  
     delslice() (operator モジュール), 68  
     demo() (turtle モジュール), 706  
     DeprecationWarning (exceptions の例外), 41  
     dereference=False (TarFile の属性), 418  
     derwin() (window のメソッド), 285  
 DES  
     cipher, 428, 681  
 descriptor, file, 31  
 Detach() ( のメソッド), 779  
 deterministic profiling, 451  
 DEVICE (standard module), 766  
 dgettext() (gettext モジュール), 346  
 Dialect (csv のクラス), 608  
 dict() (モジュール), 5  
 dictionary  
     object, 29  
     オブジェクト, 29  
     type, operations on, 29  
 DictionaryType (types のデータ), 62  
 DictMixin (UserDict のクラス), 64  
 DictReader (csv のクラス), 608  
 DictType (types のデータ), 62  
 DictWriter (csv のクラス), 608  
 diff\_files (dircmp の属性), 247  
 Differ (difflib のクラス), 131, 137  
 difflib (standard module), 131  
 digest()  
     hmac のメソッド, 681  
     md5 のメソッド, 682  
     sha のメソッド, 683  
 digest\_size  
     md5 のデータ, 682  
     sha のデータ, 683  
 digit() (unicodedata モジュール), 153  
 digits (string のデータ), 113  
 dir()  
     FTP のメソッド, 496  
     モジュール, 6

dircache (standard module), [242](#)  
 dircmp (filecmp のクラス), [246](#)  
 directory  
     changing, [226](#)  
     creating, [228](#)  
     deleting, [228](#), [338](#)  
     site-packages, [108](#)  
     site-python, [108](#)  
     traversal, [231](#)  
     walking, [231](#)  
 DirList (Tix のクラス), [701](#)  
 dirname() (os.path モジュール), [239](#)  
 DirSelectBox (Tix のクラス), [702](#)  
 DirSelectDialog (Tix のクラス), [701](#)  
 DirTree (Tix のクラス), [701](#)  
 dis() (dis モジュール), [734](#)  
 dis (standard module), [734](#)  
 disable()  
     gc モジュール, [52](#)  
     logging モジュール, [356](#)  
 disassemble() (dis モジュール), [735](#)  
 discard\_buffers() (async\_chat のメソッド), [538](#)  
 disco() (dis モジュール), [735](#)  
 dispatch() (ASTVisitor のメソッド), [750](#)  
 dispatcher (asyncore のクラス), [535](#)  
 displayhook() (sys モジュール), [46](#)  
 distb() (dis モジュール), [734](#)  
 distutils (standard module), [742](#)  
 dither2grey2() (imageop モジュール), [663](#)  
 dither2mono() (imageop モジュール), [663](#)  
 div() (operator モジュール), [67](#)  
 division  
     integer, [19](#)  
     long integer, [19](#)  
 divm() (mpz モジュール), [685](#)  
 divmod() (モジュール), [6](#)  
 dl (extension module), [428](#)  
 dllhandle (sys のデータ), [46](#)  
 dngettext() (gettext モジュール), [346](#)  
 do\_command() (Textbox のメソッド), [294](#)  
 do\_forms() (fl モジュール), [758](#)  
 do\_GET() (SimpleHTTPRequestHandler のメソッド), [522](#)  
 do\_HEAD() (SimpleHTTPRequestHandler のメソッド), [522](#)  
 do\_POST() (CGIHTTPRequestHandler のメソッド), [523](#)  
 doc\_header (Cmd の属性), [213](#)  
 DocCGIXMLRPCRequestHandler (DocXMLRPCServer のクラス), [533](#)  
 docmd() (SMTP のメソッド), [510](#)  
 docstrings, [724](#)  
 doctest (standard module), [158](#)  
 DocTestSuite() (doctest モジュール), [163](#)  
 DOCTYPE declaration, [656](#)  
 documentation  
     generation, [157](#)  
     online, [157](#)  
 documentElement (Document の属性), [632](#)  
 DocXMLRPCRequestHandler (DocXMLRPCServer のクラス), [533](#)  
 DocXMLRPCServer  
     DocXMLRPCServer のクラス, [533](#)  
     standard module, [533](#)  
 DOMEventStream (xml.dom.pulldom のクラス), [642](#)  
 DOMException (xml.dom の例外), [635](#)  
 DomstringSizeErr (xml.dom の例外), [635](#)  
 done() (Unpacker のメソッド), [603](#)  
 doRollover() (RotatingFileHandler のメソッド), [361](#)  
 DOTALL (re のデータ), [122](#)  
 doublequote (Dialect の属性), [609](#)  
 doupdate() (curses モジュール), [278](#)  
 down() (turtle モジュール), [706](#)  
 drain() (audio device のメソッド), [769](#)  
 dropwhile() (itertools モジュール), [200](#)  
 dst()  
     datetime のメソッド, [260](#)  
     time のメソッド, [263](#)  
 DTDHandler (xml.sax.handler のクラス), [645](#)  
 dumbdbm  
     standard module, [405](#)  
     標準モジュール, [399](#)  
 DumbWriter (formatter のクラス), [547](#)  
 dummy\_thread (standard module), [396](#)  
 dummy\_threading (standard module), [396](#)  
 dump()  
     marshal モジュール, [93](#)  
     Pickler のメソッド, [80](#)  
     pickle モジュール, [79](#)  
 dump\_address\_pair() (email.Utils モジュー

ル), 570  
dump\_address\_pair() (rfc822 モジュール),  
594  
dump\_stats() (Stats のメソッド), 456  
dumps()  
    marshal モジュール, 93  
    pickle モジュール, 79  
dup()  
    os モジュール, 224  
    posixfile モジュール, 438  
dup2()  
    os モジュール, 224  
    posixfile モジュール, 438  
DuplicateSectionError (ConfigParser の例  
外), 205

## E

e  
    cmath のデータ, 185  
    math のデータ, 183  
E2BIG (errno のデータ), 330  
EACCES (errno のデータ), 330  
EADDRINUSE (errno のデータ), 334  
EADDRNOTAVAIL (errno のデータ), 334  
EADV (errno のデータ), 333  
EAFNOSUPPORT (errno のデータ), 334  
EAGAIN (errno のデータ), 330  
EAI\_\* (socket のデータ), 376  
EALREADY (errno のデータ), 335  
EBADF (errno のデータ), 332  
EBADF (errno のデータ), 330  
EBADFD (errno のデータ), 333  
EBADMSG (errno のデータ), 333  
EBADR (errno のデータ), 332  
EBADRQC (errno のデータ), 332  
EBADSLT (errno のデータ), 332  
EBFONT (errno のデータ), 333  
EBUSY (errno のデータ), 331  
ECHILD (errno のデータ), 330  
echo() (curses モジュール), 278  
echochar() (window のメソッド), 285  
ECHRNG (errno のデータ), 332  
ECOMM (errno のデータ), 333  
ECONNABORTED (errno のデータ), 335  
ECONNREFUSED (errno のデータ), 335  
ECONNRESET (errno のデータ), 335  
EDEADLK (errno のデータ), 331  
EDEADLOCK (errno のデータ), 333  
EDESTADDRREQ (errno のデータ), 334  
edit() (Textbox のメソッド), 294  
EDOM (errno のデータ), 331  
EDOTDOT (errno のデータ), 333  
EDQUOT (errno のデータ), 336  
EEXIST (errno のデータ), 331  
EFAULT (errno のデータ), 331  
EFBIG (errno のデータ), 331  
ehlo() (SMTP のメソッド), 510  
EHOSTDOWN (errno のデータ), 335  
EHOSTUNREACH (errno のデータ), 335  
EIDRM (errno のデータ), 332  
EILSEQ (errno のデータ), 334  
EINPROGRESS (errno のデータ), 335  
EINTR (errno のデータ), 330  
EINVAL (errno のデータ), 331  
EIO (errno のデータ), 330  
EISCONN (errno のデータ), 335  
EISDIR (errno のデータ), 331  
EISNAM (errno のデータ), 335  
eject() (CD player のメソッド), 755  
EL2HLT (errno のデータ), 332  
EL2NSYNC (errno のデータ), 332  
EL3HLT (errno のデータ), 332  
EL3RST (errno のデータ), 332  
ElementDeclHandler() (xmlparser のメソ  
ッド), 622  
elements (XMLParser の属性), 655  
ELIBACC (errno のデータ), 334  
ELIBBAD (errno のデータ), 334  
ELIBEXEC (errno のデータ), 334  
ELIBMAX (errno のデータ), 334  
ELIBSCN (errno のデータ), 334  
Ellinghouse, Lance, 601, 685  
Ellipsis (のデータ), 43  
EllipsisType (types のデータ), 63  
ELNRNG (errno のデータ), 332  
ELOOP (errno のデータ), 332  
email (standard module), 547  
email.Charset (standard module), 564  
email.Encoders (standard module), 567, 568  
email.Generator (standard module), 558  
email.Header (standard module), 562  
email.Iterators (standard module), 571  
email.Message (standard module), 548  
email.Parser (standard module), 556

email.Utils (standard module), [569](#)  
 EMFILE (errno のデータ), [331](#)  
 emit()  
     のメソッド, [360](#)  
     BufferingHandler のメソッド, [364](#)  
     DatagramHandler のメソッド, [362](#)  
     FileHandler のメソッド, [361](#)  
     HTTPHandler のメソッド, [365](#)  
     NTEventLogHandler のメソッド, [363](#)  
     RotatingFileHandler のメソッド, [361](#)  
     SMTPHandler のメソッド, [364](#)  
     SocketHandler のメソッド, [361](#)  
     StreamHandler のメソッド, [360](#)  
     SysLogHandler のメソッド, [362](#)  
 EMLINK (errno のデータ), [331](#)  
 Empty (Queue の例外), [397](#)  
 empty()  
     Queue のメソッド, [397](#)  
     scheduler のメソッド, [275](#)  
 EMPTY\_NAMESPACE (xml.dom のデータ), [628](#)  
 emptyline() (Cmd のメソッド), [213](#)  
 EMSGSIZE (errno のデータ), [334](#)  
 EMULTIHOP (errno のデータ), [333](#)  
 enable()  
     cgib モジュール, [476](#)  
     gc モジュール, [52](#)  
 ENAMETOOLONG (errno のデータ), [332](#)  
 ENAVAIL (errno のデータ), [335](#)  
 enclose() (window のメソッド), [285](#)  
 encode  
     Codecs, [143](#)  
 encode()  
     のメソッド, [146](#)  
     base64 モジュール, [597](#)  
     Binary のメソッド, [530](#)  
     email.Utils モジュール, [570](#)  
     Header のメソッド, [563](#)  
     mimetools モジュール, [584](#)  
     quopri モジュール, [601](#)  
     ServerProxy のメソッド, [530](#)  
     string のメソッド, [23](#)  
     uu モジュール, [601](#)  
 encode\_7or8bit() (email.Encoders モジュール), [568](#), [569](#)  
 encode\_base64() (email.Encoders モジュール), [568](#)  
 encode\_noop() (email.Encoders モジュール), [568](#), [569](#)  
 encode\_quopri() (email.Encoders モジュール), [568](#)  
 encode\_rfc2231() (email.Utils モジュール), [570](#)  
 encoded\_header\_len() (Charset のメソッド), [566](#)  
 EncodedFile() (codecs モジュール), [145](#)  
 encodePriority() (SysLogHandler のメソッド), [362](#)  
 encodestring()  
     base64 モジュール, [597](#)  
     quopri モジュール, [601](#)  
 encoding  
     base64, [597](#)  
     quoted-printable, [600](#)  
 encoding (file の属性), [33](#)  
 encodings.idna (standard module), [152](#)  
 encodings\_map (mimetypes のデータ), [587](#), [588](#)  
 encrypt() (rotor のメソッド), [685](#)  
 encryptmore() (rotor のメソッド), [686](#)  
 end() (MatchObject のメソッド), [126](#)  
 end\_group() (form のメソッド), [760](#)  
 end\_headers() (BaseHTTPRequestHandler のメソッド), [521](#)  
 end\_marker() (MultiFile のメソッド), [592](#)  
 end\_paragraph() (formatter のメソッド), [544](#)  
 EndCdataSectionHandler() (xmlparser のメソッド), [623](#)  
 EndDoctypeDeclHandler() (xmlparser のメソッド), [621](#)  
 endDocument() (ContentHandler のメソッド), [647](#)  
 endElement() (ContentHandler のメソッド), [648](#)  
 EndElementHandler() (xmlparser のメソッド), [622](#)  
 endElementNS() (ContentHandler のメソッド), [648](#)  
 endheaders() (HTTPConnection のメソッド), [492](#)  
 EndNamespaceDeclHandler() (xmlparser のメソッド), [622](#)  
 endpick() (gl モジュール), [765](#)  
 endpos (MatchObject の属性), [126](#)  
 endPrefixMapping() (ContentHandler のメソッド), [521](#)

- ツド), 647
- endselect() (gl モジュール), 765
- endswith() (string のメソッド), 23
- endwin() (curses モジュール), 279
- ENETDOWN (errno のデータ), 334
- ENETRESET (errno のデータ), 335
- ENETUNREACH (errno のデータ), 335
- ENFILE (errno のデータ), 331
- Enigma
  - cipher, 685
- ENOANO (errno のデータ), 332
- ENOBUFFS (errno のデータ), 335
- ENOCSI (errno のデータ), 332
- ENODATA (errno のデータ), 333
- ENODEV (errno のデータ), 331
- ENOENT (errno のデータ), 330
- ENOEXEC (errno のデータ), 330
- ENOLCK (errno のデータ), 332
- ENOLINK (errno のデータ), 333
- ENOMEM (errno のデータ), 330
- ENOMSG (errno のデータ), 332
- ENONET (errno のデータ), 333
- ENOPKG (errno のデータ), 333
- ENOPROTOOPT (errno のデータ), 334
- ENOSPC (errno のデータ), 331
- ENOSR (errno のデータ), 333
- ENOSTR (errno のデータ), 333
- ENOSYS (errno のデータ), 332
- ENOTBLK (errno のデータ), 331
- ENOTCONN (errno のデータ), 335
- ENOTDIR (errno のデータ), 331
- ENOTEMPTY (errno のデータ), 332
- ENOTNAM (errno のデータ), 335
- ENOTSOCK (errno のデータ), 334
- ENOTTY (errno のデータ), 331
- ENOTUNIQ (errno のデータ), 333
- enter() (scheduler のメソッド), 275
- enterabs() (scheduler のメソッド), 275
- entities (DocumentType の属性), 631
- ENTITY declaration, 657
- EntityDeclHandler() (xmlparser のメソッド), 622
- entitydefs
  - htmlentitydefs のデータ, 618
  - XMLParser の属性, 655
- EntityResolver (xml.sax.handler のクラス), 645
- enumerate()
  - fm モジュール, 763
  - threading モジュール, 388
  - モジュール, 6
- EnumKey() (\_winreg モジュール), 775
- EnumValue() (\_winreg モジュール), 775
- environ
  - os のデータ, 220
  - posix のデータ, 426
- environment variables
  - AUDIODEV, 675
  - BROWSER, 466
  - COLUMNS, 283
  - COMSPEC, 237
  - HOME, 109, 239
  - KDEDIR, 467
  - LANGUAGE, 345, 347
  - LANG, 339, 340, 345, 347
  - LC\_ALL, 345, 347
  - LC\_MESSAGES, 345, 347
  - LINES, 283
  - LNAME, 277
  - LOGNAME, 221, 277
  - PAGER, 447
  - PATH, 233, 235, 239, 473, 475
  - PYTHONPATH, 50, 473, 784
  - PYTHONSTARTUP, 109, 422, 423
  - PYTHON2K, 269, 270
  - PYTHON\_DOM, 627
  - TEMP, 329
  - TIX\_LIBRARY, 700
  - TMPDIR, 329
  - TMP, 329
  - TZ, 273, 274, 785
  - USERNAME, 277
  - USER, 277
  - Wimp\$ScrapDir, 329
  - ftp\_proxy, 477
  - gopher\_proxy, 477
  - http\_proxy, 477
  - setting, 221
- EnvironmentError (exceptions の例外), 37
- ENXIO (errno のデータ), 330
- 演算子
  - ==, 18
  - and, 17, 18
  - in, 18, 22



- is, 18
- is not, 18
- not, 18
- not in, 18, 22
- or, 17, 18
- eof (shlex の属性), 216
- EOFError (exceptions の例外), 38
- EOPNOTSUPP (errno のデータ), 334
- EOVERFLOW (errno のデータ), 333
- EPERM (errno のデータ), 330
- EPFNOSUPPORT (errno のデータ), 334
- epilogue (email.Message のデータ), 555
- EPIPE (errno のデータ), 331
- epoch, 269
- EPROTO (errno のデータ), 333
- EPROTONOSUPPORT (errno のデータ), 334
- EPROTOTYPE (errno のデータ), 334
- eq ( ) (operator モジュール), 66
- ERA (locale のデータ), 343
- ERA\_D\_FMT (locale のデータ), 343
- ERA\_D\_T\_FMT (locale のデータ), 343
- ERA\_YEAR (locale のデータ), 343
- ERANGE (errno のデータ), 331
- erase ( ) (window のメソッド), 285
- erasechar ( ) (curses モジュール), 279
- EREMCHG (errno のデータ), 333
- EREMOTE (errno のデータ), 333
- EREMOTEIO (errno のデータ), 335
- ERESTART (errno のデータ), 334
- EROFS (errno のデータ), 331
- ERR (curses のデータ), 289
- errcode (ServerProxy の属性), 531
- errmsg (ServerProxy の属性), 531
- errno
  - standard module, 330
  - 組み込みモジュール, 220, 375
- ERROR (cd のデータ), 754
- Error
  - binascii の例外, 599
  - binhex の例外, 600
  - csv の例外, 609
  - locale の例外, 339
  - shutil の例外, 338
  - sunau の例外, 666
  - turtle の例外, 706
  - uu の例外, 601
  - wave の例外, 669
  - webbrowser の例外, 466
  - xdrlib の例外, 604
- error ( )
  - のメソッド, 358
  - ErrorHandler のメソッド, 649
  - Folder のメソッド, 583
  - logging モジュール, 356
  - MH のメソッド, 582
  - OpenerDirector のメソッド, 486
- error
  - anydbm の例外, 400
  - audioop の例外, 659
  - cd の例外, 754
  - curses の例外, 277
  - dbhash の例外, 400
  - dbm の例外, 430
  - dl の例外, 429
  - dumbdbm の例外, 405
  - gdbm の例外, 431
  - getopt の例外, 300
  - imageop の例外, 662
  - imgfile の例外, 766
  - jpeg の例外, 767
  - nis の例外, 443
  - ossaudiodev の例外, 675
  - os の例外, 220
  - resource の例外, 440
  - re の例外, 124
  - rgbimg の例外, 673
  - select の例外, 384
  - socket の例外, 375
  - struct の例外, 128
  - sunaudiodev の例外, 769
  - thread の例外, 386
  - xml.parsers.expat の例外, 619
  - zipfile の例外, 411
  - zlib の例外, 406
- error\_leader ( ) (shlex のメソッド), 215
- error\_message\_format (BaseHTTPRequest-Handler の属性), 520
- error\_perm (ftplib の例外), 494
- error\_proto
  - ftplib の例外, 494
  - poplib の例外, 498
- error\_reply (ftplib の例外), 494
- error\_temp (ftplib の例外), 494
- ErrorByteIndex (xmlparser の属性), 621

- ErrorCode (xmlparser の属性), 621
- errorcode (errno のデータ), 330
- ErrorColumnNumber (xmlparser の属性), 621
- ErrorHandler (xml.sax.handler のクラス), 645
- errorlevel=0 (TarFile の属性), 419
- ErrorLineNumber (xmlparser の属性), 621
- Errors
  - logging, 354
- errors (TestResult の属性), 176
- ErrorString() (xml.parsers.expat モジュール), 619
- escape()
  - cgi モジュール, 472
  - re モジュール, 124
  - xml.sax.saxutils モジュール, 650
- escape (shlex の属性), 216
- escapechar (Dialect の属性), 609
- escapedquotes (shlex の属性), 216
- ESHUTDOWN (errno のデータ), 335
- ESOCKTNOSUPPORT (errno のデータ), 334
- ESPIPE (errno のデータ), 331
- ESRCH (errno のデータ), 330
- ESRMNT (errno のデータ), 333
- ESTALE (errno のデータ), 335
- ESTRPIPE (errno のデータ), 334
- ETIME (errno のデータ), 333
- ETIMEDOUT (errno のデータ), 335
- ETOOMANYREFS (errno のデータ), 335
- ETXTBSY (errno のデータ), 331
- EUCLEAN (errno のデータ), 335
- EUNATCH (errno のデータ), 332
- EUSERS (errno のデータ), 334
- eval()
  - モジュール, 7
  - 組み込み関数, 35, 105, 114 721
- Event() (threading モジュール), 388
- Event (threading のクラス), 393
- event scheduling, 274
- events (widgets), 697
- EWOLDBLOCK (errno のデータ), 332
- EX\_CANTCREAT (os のデータ), 234
- EX\_CONFIG (os のデータ), 234
- EX\_DATAERR (os のデータ), 233
- EX\_IOERR (os のデータ), 234
- EX\_NOHOST (os のデータ), 233
- EX\_NOINPUT (os のデータ), 233
- EX\_NOPERM (os のデータ), 234
- EX\_NOTFOUND (os のデータ), 234
- EX\_NOUSER (os のデータ), 233
- EX\_OK (os のデータ), 233
- EX\_OSERR (os のデータ), 234
- EX\_OSFILE (os のデータ), 234
- EX\_PROTOCOL (os のデータ), 234
- EX\_SOFTWARE (os のデータ), 234
- EX\_TEMPFAIL (os のデータ), 234
- EX\_UNAVAILABLE (os のデータ), 234
- EX\_USAGE (os のデータ), 233
- exc\_clear() (sys モジュール), 47
- exc\_info() (sys モジュール), 46
- exc\_traceback (sys のデータ), 47
- exc\_type (sys のデータ), 47
- exc\_value (sys のデータ), 47
- except
  - 実行文, 37
  - statement, 37
- excepthook()
  - in module sys, 476
  - sys モジュール, 46
- Exception (exceptions の例外), 37
- exception()
  - のメソッド, 358
  - logging モジュール, 356
- exceptions
  - built-in, 3
  - in CGI scripts, 475
- exceptions (standard module), 36
- EXDEV (errno のデータ), 331
- exec
  - 実行文, 35
  - statement, 35
- exec\_prefix (sys のデータ), 47
- execfile()
  - モジュール, 7
  - 組み込み関数, 109
- execl() (os モジュール), 232
- execle() (os モジュール), 232
- execlp() (os モジュール), 232
- execlpe() (os モジュール), 232
- executable (sys のデータ), 47
- execv() (os モジュール), 232
- execve() (os モジュール), 232
- execvp() (os モジュール), 233
- execvpe() (os モジュール), 233
- ExFileSelectBox (Tix のクラス), 702

EXFULL (errno のデータ), 332  
exists() (os.path モジュール), 239  
exit()  
    sys モジュール, 47  
    thread モジュール, 386  
exitfunc  
    in sys, 60  
    sys のデータ, 48  
exp()  
    cmath モジュール, 184  
    math モジュール, 182  
expand() (MatchObject のメソッド), 125  
expand\_tabs (TextWrapper の属性), 142  
expandNode() (DOMEventStream のメソッド), 643  
expandtabs()  
    string のメソッド, 23  
    string モジュール, 115  
expanduser() (os.path モジュール), 239  
expandvars() (os.path モジュール), 239  
Expat, 618  
ExpatError (xml.parsers.expat の例外), 619  
expect() (Telnet のメソッド), 514  
expovariate() (random モジュール), 187  
expr() (parser モジュール), 720  
expunge() (IMAP4\_stream のメソッド), 501  
extend()  
    array のメソッド, 194  
    list method, 28  
extend\_path() (pkgutil モジュール), 99  
extended slice  
    assignment, 28  
    operation, 22  
Extensible Markup Language, 655  
extensions\_map (SimpleHTTPRequestHandler の属性), 522  
External Data Representation, 78, 602  
external\_attr (ZipInfo の属性), 415  
ExternalEntityParserCreate() (xml-parser のメソッド), 620  
ExternalEntityRefHandler() (xmlparser のメソッド), 623  
extra (ZipInfo の属性), 414  
extract() (TarFile のメソッド), 417  
extract\_stack() (traceback モジュール), 75  
extract\_tb() (traceback モジュール), 75  
extract\_version (ZipInfo の属性), 414

ExtractError (tarfile の例外), 416  
extractfile() (TarFile のメソッド), 418  
extsep (os のデータ), 239

## F

F\_BAVAIL (statvfs のデータ), 245  
F\_BFREE (statvfs のデータ), 245  
F\_BLOCKS (statvfs のデータ), 245  
F\_BSIZE (statvfs のデータ), 245  
F\_FAVAIL (statvfs のデータ), 245  
F\_FFREE (statvfs のデータ), 245  
F\_FILES (statvfs のデータ), 245  
F\_FLAG (statvfs のデータ), 245  
F\_FRSIZE (statvfs のデータ), 245  
F\_NAMEMAX (statvfs のデータ), 246  
F\_OK (os のデータ), 226  
fabs() (math モジュール), 182  
fail() (TestCase のメソッド), 175  
failIf() (TestCase のメソッド), 175  
failIfAlmostEqual() (TestCase のメソッド), 175  
failIfEqual() (TestCase のメソッド), 174  
failUnless() (TestCase のメソッド), 174  
failUnlessAlmostEqual() (TestCase のメソッド), 174  
failUnlessEqual() (TestCase のメソッド), 174  
failUnlessRaises() (TestCase のメソッド), 175  
failureException (TestCase の属性), 175  
failures (TestResult の属性), 176  
False, 17, 36  
False  
    Built-in object, 17  
    のデータ, 42  
false, 17  
FancyURLopener (urllib のクラス), 479  
fatalError() (ErrorHandler のメソッド), 649  
faultCode (ServerProxy の属性), 530  
faultString (ServerProxy の属性), 530  
fcntl() (fcntl モジュール), 434  
fcntl  
    built-in module, 434  
    組み込みモジュール, 31  
fcntl() (in module fcntl), 437  
fdatasync() (os モジュール), 224  
fdopen() (os モジュール), 222

- feature\_external\_ges (xml.sax.handler のデータ), 646
- feature\_external\_pes (xml.sax.handler のデータ), 646
- feature\_namespace\_prefixes (xml.sax.handler のデータ), 645
- feature\_namespaces (xml.sax.handler のデータ), 645
- feature\_string\_interning (xml.sax.handler のデータ), 645
- feature\_validation (xml.sax.handler のデータ), 645
- feed()
  - HTMLParser のメソッド, 612
  - IncrementalParser のメソッド, 653
  - SGMLParser のメソッド, 614
  - XMLParser のメソッド, 656
- fetch() (IMAP4\_stream のメソッド), 501
- fifo (asynchat のクラス), 539
- file
  - .ini, 205
  - .pdbrc, 447
  - .pythonrc.py, 109
  - byte-code, 96, 98, 733
  - configuration, 205
  - copying, 337
  - debugger configuration, 447
  - large files, 426
  - mime.types, 587
  - object, 31
  - オブジェクト, 31
  - path configuration, 108
  - temporary, 327
  - user configuration, 109
- file()
  - posixfile モジュール, 438
  - モジュール, 7
  - 組み込み関数, 31
- file
  - class descriptor の属性, 732
  - function descriptor の属性, 733
- file control
  - UNIX, 434
- file descriptor, 31
- file name
  - temporary, 327
- file object
  - POSIX, 437
- file\_offset (ZipInfo の属性), 415
- file\_open() (FileHandler のメソッド), 489
- file\_size (ZipInfo の属性), 415
- filecmp (standard module), 246
- fileConfig() (logging モジュール), 367
- FileEntry (Tix のクラス), 702
- FileHandler
  - logging のクラス, 360
  - urllib2 のクラス, 484
- FileInput (fileinput のクラス), 209
- fileinput (standard module), 208
- filelineno() (fileinput モジュール), 209
- filename() (fileinput モジュール), 209
- filename (ZipInfo の属性), 414
- filename\_only (tabnanny のデータ), 731
- filenames
  - pathname expansion, 336
  - wildcard expansion, 336
- fileno()
  - audio device のメソッド, 676, 769
  - file のメソッド, 31
  - mixer device のメソッド, 678
  - Profile のメソッド, 460
  - socket のメソッド, 379
  - SocketServer モジュール, 517
  - Telnet のメソッド, 514
- fileopen() (posixfile モジュール), 438
- FileSelectBox (Tix のクラス), 702
- FileType (types のデータ), 63
- fill()
  - TextWrapper のメソッド, 143
  - textwrap モジュール, 141
  - turtle モジュール, 706
- Filter (logging のクラス), 366
- filter() (Filter のメソッド), 366
- filter() ( のメソッド), 358, 359
- filter()
  - curses モジュール, 279
  - fnmatch モジュール, 337
  - モジュール, 8
- filterwarnings() (warnings モジュール), 96
- find()
  - のメソッド, 399
  - gettext モジュール, 346
  - string のメソッド, 23
  - string モジュール, 115

`find_first()` (form のメソッド), 760  
`find_last()` (form のメソッド), 760  
`find_longest_match()` (SequenceMatcher のメソッド), 134  
`find_module()` (imp モジュール), 96  
`find_prefix_at_end()` (asynchat モジュール), 540  
`find_user_password()` (HTTPPasswordMgr のメソッド), 488  
`findall()`  
     RegexObject のメソッド, 125  
     re モジュール, 123  
`findCaller()` ( のメソッド), 359  
`findfactor()` (audioop モジュール), 660  
`findfile()` (test.test\_support モジュール), 181  
`findfit()` (audioop モジュール), 660  
`findfont()` (fm モジュール), 763  
`finditer()`  
     RegexObject のメソッド, 125  
     re モジュール, 123  
`findmatch()` (mailcap モジュール), 579  
`findmax()` (audioop モジュール), 660  
`finish()` (SocketServer モジュール), 518  
`finish_request()` (SocketServer モジュール), 518  
`first()`  
     のメソッド, 403  
     dbhash のメソッド, 401  
     fifo のメソッド, 540  
`firstChild` (Node の属性), 629  
`firstkey()` (gdbm モジュール), 431  
`firstweekday()` (calendar モジュール), 211  
`fix()` (fpformat モジュール), 139  
`fix_sentence_endings` (TextWrapper の属性), 142  
`FL` (standard module), 763  
`fl` (built-in module), 757  
`flag_bits` (ZipInfo の属性), 414  
`flags()` (posixfile モジュール), 438  
`flags` (RegexObject の属性), 125  
`flash()` (curses モジュール), 279  
`flatten()` (Generator のメソッド), 559  
`flattening`  
     objects, 77  
`float()`  
     モジュール, 8  
     組み込み関数, 19, 114  
`floating point`  
     literals, 19  
     object, 19  
     オブジェクト, 19  
`FloatingPointError`  
     exceptions の例外, 38  
     fpectl の例外, 59  
`FloatType` (types のデータ), 62  
`flock()` (fcntl モジュール), 435  
`floor()`  
     in module math, 20  
     math モジュール, 182  
`floordiv()` (operator モジュール), 67  
`flp` (standard module), 763  
`flush()`  
     のメソッド, 359, 399  
     audio device のメソッド, 770  
     BufferingHandler のメソッド, 364  
     BZ2Compressor のメソッド, 410  
     Compress のメソッド, 407  
     Decompress のメソッド, 407  
     file のメソッド, 31  
     MemoryHandler のメソッド, 364  
     StreamHandler のメソッド, 360  
     writer のメソッド, 546  
`flush_softspace()` (formatter のメソッド), 545  
`flushheaders()` (MimeWriter のメソッド), 588  
`flushinp()` (curses モジュール), 279  
`FlushKey()` (\_winreg モジュール), 776  
`fm` (built-in module), 763  
`fmod()` (math モジュール), 182  
`fnmatch()` (fnmatch モジュール), 337  
`fnmatch` (standard module), 336  
`fnmatchcase()` (fnmatch モジュール), 337  
`Folder` (mhlib のクラス), 582  
`Font Manager, IRIS`, 763  
`fontpath()` (fm モジュール), 763  
`forget()`  
     statcache モジュール, 245  
     test.test\_support モジュール, 181  
`forget_dir()` (statcache モジュール), 245  
`forget_except_prefix()` (statcache モジュール), 245  
`forget_prefix()` (statcache モジュール), 245  
`fork()`  
     os モジュール, 234

pty モジュール, 434  
forkpty() (os モジュール), 234  
Form (Tix のクラス), 703  
Formal Public Identifier, 656  
format()  
    のメソッド, 360  
    Formatter のメソッド, 365  
    locale モジュール, 341  
    PrettyPrinter のメソッド, 106  
format\_exception() (traceback モジュール), 75  
format\_exception\_only() (traceback モジュール), 75  
format\_list() (traceback モジュール), 75  
format\_stack() (traceback モジュール), 75  
format\_tb() (traceback モジュール), 75  
formataddr() (email.Utils モジュール), 569  
formatdate() (email.Utils モジュール), 570  
FormatException() (Formatter のメソッド), 366  
Formatter (logging のクラス), 365  
formatter  
    HTMLParser の属性, 617  
    standard module, 543  
    標準モジュール, 616  
formatTime() (Formatter のメソッド), 366  
formatting, string (%), 25  
formatwarning() (warnings モジュール), 95  
FORMS Library, 757  
forward() (turtle モジュール), 706  
found\_terminator() (async\_chat のメソッド), 538  
fp (AddressList の属性), 596  
fpathconf() (os モジュール), 224  
fpectl (extension module), 59  
fpformat (standard module), 139  
frame  
    object, 373  
    オブジェクト, 373  
frame (ScrolledText の属性), 705  
FrameType (types のデータ), 63  
freeze\_form() (form のメソッド), 759  
freeze\_object() (FORMS object のメソッド), 762  
frexp() (math モジュール), 182  
from\_splittable() (Charset のメソッド), 566  
frombuf() (TarInfo のメソッド), 419

fromchild (Popen4 の属性), 248  
fromfd() (socket モジュール), 377  
fromfile() (array のメソッド), 194  
fromkeys() (dictionary method), 29  
fromlist() (array のメソッド), 194  
fromordinal()  
    date のメソッド, 253  
    datetime のメソッド, 257  
fromstring() (array のメソッド), 194  
fromtimestamp()  
    date のメソッド, 253  
    datetime のメソッド, 256  
fromunicode() (array のメソッド), 194  
fromutc() (time のメソッド), 265  
fstat() (os モジュール), 224  
fstatvfs() (os モジュール), 224  
fsync() (os モジュール), 224  
FTP  
    ftplib (standard module), 493  
    protocol, 480, 493  
FTP (ftplib のクラス), 494  
ftp\_open() (FTPHandler のメソッド), 489  
ftp\_proxy, 477  
FTPHandler (urllib2 のクラス), 484  
ftplib (standard module), 493  
ftpmirror.py, 494  
ftruncate() (os モジュール), 224  
Full (Queue の例外), 397  
full() (Queue のメソッド), 397  
func\_code (function object attribute), 35  
function() (new モジュール), 108  
functions  
    built-in, 3  
FunctionTestCase (unittest のクラス), 173  
FunctionType (types のデータ), 63  
funny\_files (dircmp の属性), 247  
FutureWarning (exceptions の例外), 41

## G

G.722, 665  
gaierror (socket の例外), 375  
gammavariate() (random モジュール), 187  
garbage (gc のデータ), 53  
gather() (Textbox のメソッド), 294  
gauss() (random モジュール), 187  
gc (extension module), 52  
gcd() (mpz モジュール), 684



`gcdext()` (mpz モジュール), 684  
`gdbm`  
     built-in module, 430  
     組み込みモジュール, 89, 399  
`ge()` (operator モジュール), 66  
`generate_tokens()` (tokenize モジュール), 730  
`Generator` (email.Generator のクラス), 558  
`GeneratorType` (types のデータ), 63  
`get()`  
     AddressList のメソッド, 595  
     Message のメソッド, 551  
     Queue のメソッド, 397  
`get()` (SafeConfigParser のメソッド), 207, 208  
`get()`  
     dictionary method, 29  
     mixer device のメソッド, 678  
     webbrowser モジュール, 466  
`get_all()` (Message のメソッド), 551  
`get_begidx()` (readline モジュール), 422  
`get_body_encoding()` (Charset のメソッド), 565  
`get_boundary()` (Message のメソッド), 553  
`get_buffer()`  
     Packer のメソッド, 602  
     Unpacker のメソッド, 603  
`get_charset()` (Message のメソッド), 550  
`get_charsets()` (Message のメソッド), 554  
`get_close_matches()` (difflib モジュール), 132  
`get_completer()` (readline モジュール), 422  
`get_completer_delims()` (readline モジュール), 422  
`get_content_charset()` (Message のメソッド), 554  
`get_content_maintype()` (Message のメソッド), 552  
`get_content_subtype()` (Message のメソッド), 552  
`get_content_type()` (Message のメソッド), 551  
`get_data()` (Request のメソッド), 485  
`get_debug()` (gc モジュール), 53  
`get_default_type()` (Message のメソッド), 552  
`get_dialect()` (csv モジュール), 608  
`get_directory()` (fl モジュール), 758  
`get_endidx()` (readline モジュール), 422  
`get_filename()`  
     fl モジュール, 758  
     Message のメソッド, 553  
`get_full_url()` (Request のメソッド), 485  
`get_grouped_opcodes()` (SequenceMatcher のメソッド), 136  
`get_history_length()` (readline モジュール), 421  
`get_host()` (Request のメソッド), 485  
`get_ident()` (thread モジュール), 386  
`get_line_buffer()` (readline モジュール), 421  
`get_magic()` (imp モジュール), 96  
`get_main_type()` (Message のメソッド), 555  
`get_matching_blocks()` (SequenceMatcher のメソッド), 135  
`get_method()` (Request のメソッド), 485  
`get_mouse()` (fl モジュール), 759  
`get_nowait()` (Queue のメソッド), 397  
`get_objects()` (gc モジュール), 53  
`get_opcodes()` (SequenceMatcher のメソッド), 135  
`get_option()` ( のメソッド), 316  
`get_osfhandle()` (msvcrt モジュール), 774  
`get_output_charset()` (Charset のメソッド), 566  
`get_param()` (Message のメソッド), 552  
`get_params()` (Message のメソッド), 552  
`get_pattern()` (fl モジュール), 758  
`get_payload()` (Message のメソッド), 549  
`get_position()` (Unpacker のメソッド), 603  
`get_recsrc()` (mixer device のメソッド), 679  
`get_referents()` (gc モジュール), 53  
`get_referrers()` (gc モジュール), 53  
`get_request()` (SocketServer モジュール), 518  
`get_rgbmode()` (fl モジュール), 758  
`get_selector()` (Request のメソッド), 485  
`get_socket()` (Telnet のメソッド), 514  
`get_starttag_text()` (HTMLParser のメソッド), 612  
`get_starttag_text()` (SGMLParser のメソッド), 614  
`get_subtype()` (Message のメソッド), 556  
`get_suffixes()` (imp モジュール), 96  
`get_terminator()` (async\_chat のメソッド), 538  
`get_threshold()` (gc モジュール), 53

[get\\_token\(\)](#) (shlex のメソッド), [214](#)  
[get\\_type\(\)](#)  
     Message のメソッド, [555](#)  
     Request のメソッド, [485](#)  
[get\\_unixfrom\(\)](#) (Message のメソッド), [549](#)  
[getacl\(\)](#) (IMAP4\_stream のメソッド), [501](#)  
[getaddr\(\)](#) (AddressList のメソッド), [595](#)  
[getaddresses\(\)](#) (email.Utils モジュール), [569](#)  
[getaddrinfo\(\)](#) (socket モジュール), [376](#)  
[getaddrlist\(\)](#) (AddressList のメソッド), [595](#)  
[getallmatchingheaders\(\)](#) (AddressList のメソッド), [595](#)  
[getatime\(\)](#) (os.path モジュール), [240](#)  
[getattr\(\)](#) (モジュール), [8](#)  
[getAttribute\(\)](#) (Element のメソッド), [633](#)  
[getAttributeNode\(\)](#) (Element のメソッド), [633](#)  
[getAttributeNodeNS\(\)](#) (Element のメソッド), [633](#)  
[getAttributeNS\(\)](#) (Element のメソッド), [633](#)  
[GetBase\(\)](#) (xmlparser のメソッド), [620](#)  
[getbegyx\(\)](#) (window のメソッド), [285](#)  
[getboolean\(\)](#) (SafeConfigParser のメソッド), [207](#)  
[getByteStream\(\)](#) (InputSource のメソッド), [654](#)  
[getcaps\(\)](#) (mailcap モジュール), [580](#)  
[getch\(\)](#)  
     msvcrt モジュール, [774](#)  
     window のメソッド, [285](#)  
[getchannels\(\)](#) (audio configuration のメソッド), [752](#)  
[getCharacterStream\(\)](#) (InputSource のメソッド), [654](#)  
[getche\(\)](#) (msvcrt モジュール), [774](#)  
[getcheckinterval\(\)](#) (sys モジュール), [48](#)  
[getChildNodes\(\)](#) (Node のメソッド), [745](#)  
[getChildren\(\)](#) (Node のメソッド), [745](#)  
[getColumnNumber\(\)](#) (Locator のメソッド), [653](#)  
[getcomment\(\)](#) (fm モジュール), [764](#)  
[getcomments\(\)](#) (inspect モジュール), [74](#)  
[getcompname\(\)](#)  
     aifc のメソッド, [664](#)  
     AU\_read のメソッド, [667](#)  
     Wave\_read のメソッド, [669](#)  
[getcomptype\(\)](#)  
     aifc のメソッド, [664](#)  
     AU\_read のメソッド, [667](#)  
     Wave\_read のメソッド, [669](#)  
[getconfig\(\)](#) (audio port のメソッド), [753](#)  
[getContentHandler\(\)](#) (XMLReader のメソッド), [652](#)  
[getcontext\(\)](#) (MH のメソッド), [583](#)  
[getctime\(\)](#) (os.path モジュール), [240](#)  
[getcurrent\(\)](#) (Folder のメソッド), [583](#)  
[getcwd\(\)](#) (os モジュール), [226](#)  
[getdate\(\)](#) (AddressList のメソッド), [595](#)  
[getdate\\_tz\(\)](#) (AddressList のメソッド), [596](#)  
[getdecoder\(\)](#) (codecs モジュール), [144](#)  
[getdefaultencoding\(\)](#) (sys モジュール), [48](#)  
[getdefaultlocale\(\)](#) (locale モジュール), [340](#)  
[getdefaulttimeout\(\)](#) (socket モジュール), [378](#)  
[getdlopenflags\(\)](#) (sys モジュール), [48](#)  
[getdoc\(\)](#) (inspect モジュール), [74](#)  
[getDOMImplementation\(\)](#) (xml.dom モジュール), [627](#)  
[getDTDHandler\(\)](#) (XMLReader のメソッド), [652](#)  
[getEffectiveLevel\(\)](#) (のメソッド), [358](#)  
[getegid\(\)](#) (os モジュール), [221](#)  
[getElementsByTagName\(\)](#)  
     Document のメソッド, [632](#)  
     Element のメソッド, [633](#)  
[getElementsByTagNameNS\(\)](#)  
     Document のメソッド, [632](#)  
     Element のメソッド, [633](#)  
[getencoder\(\)](#) (codecs モジュール), [144](#)  
[getEncoding\(\)](#) (InputSource のメソッド), [654](#)  
[getencoding\(\)](#) (Message のメソッド), [585](#)  
[getEntityResolver\(\)](#) (XMLReader のメソッド), [652](#)  
[getenv\(\)](#) (os モジュール), [221](#)  
[getErrorHandler\(\)](#) (XMLReader のメソッド), [652](#)  
[geteuid\(\)](#) (os モジュール), [221](#)  
[getEvent\(\)](#) (DOMEventStream のメソッド), [643](#)  
[getEventCategory\(\)](#) (NTEventLogHandler のメソッド), [363](#)  
[getEventType\(\)](#) (NTEventLogHandler のメソッド), [363](#)  
[getException\(\)](#) (SAXException のメソッド), [644](#)

[getfd\(\)](#) (audio port のメソッド), [752](#)  
[getFeature\(\)](#) (XMLReader のメソッド), [652](#)  
[getFile\(\)](#) (inspect モジュール), [74](#)  
[getfilesystemencoding\(\)](#) (sys モジュール), [48](#)  
[getfillable\(\)](#) (audio port のメソッド), [752](#)  
[getfilled\(\)](#) (audio port のメソッド), [752](#)  
[getfillpoint\(\)](#) (audio port のメソッド), [753](#)  
[getfirst\(\)](#) (FieldStorage のメソッド), [470](#)  
[getfirstmatchingheader\(\)](#) (AddressList のメソッド), [595](#)  
[getfloat\(\)](#) (SafeConfigParser のメソッド), [207](#)  
[getfloatmax\(\)](#) (audio configuration のメソッド), [752](#)  
[getfmts\(\)](#) (audio device のメソッド), [676](#)  
[getfontinfo\(\)](#) (fm モジュール), [764](#)  
[getfontname\(\)](#) (fm モジュール), [764](#)  
[getfqdn\(\)](#) (socket モジュール), [376](#)  
[getframerate\(\)](#)  
     [aifc](#) のメソッド, [664](#)  
     [AU\\_read](#) のメソッド, [667](#)  
     [Wave\\_read](#) のメソッド, [669](#)  
[getfullname\(\)](#) (Folder のメソッド), [583](#)  
[getgid\(\)](#) (os モジュール), [221](#)  
[getgrall\(\)](#) (grp モジュール), [428](#)  
[getgrgid\(\)](#) (grp モジュール), [428](#)  
[getgrnam\(\)](#) (grp モジュール), [428](#)  
[getgroups\(\)](#) (os モジュール), [221](#)  
[getheader\(\)](#)  
     [AddressList](#) のメソッド, [595](#)  
     [HTTPSConnection](#) のメソッド, [492](#)  
[gethostbyaddr\(\)](#)  
     in module socket, [222](#)  
     socket モジュール, [377](#)  
[gethostbyname\(\)](#) (socket モジュール), [376](#)  
[gethostbyname\\_ex\(\)](#) (socket モジュール), [376](#)  
[gethostname\(\)](#)  
     in module socket, [222](#)  
     socket モジュール, [376](#)  
[getinfo\(\)](#)  
     audio device のメソッド, [770](#)  
     ZipFile のメソッド, [412](#)  
[GetInputContext\(\)](#) (xmlparser のメソッド), [620](#)  
[getint\(\)](#) (SafeConfigParser のメソッド), [207](#)  
[getitem\(\)](#) (operator モジュール), [68](#)  
[getkey\(\)](#) (window のメソッド), [285](#)  
[getlast\(\)](#) (Folder のメソッド), [583](#)  
[getLength\(\)](#) (Attributes のメソッド), [654](#)  
[getLevelName\(\)](#) (logging モジュール), [357](#)  
[getline\(\)](#) (linecache モジュール), [76](#)  
[getLineNumber\(\)](#) (Locator のメソッド), [653](#)  
[getlist\(\)](#) (FieldStorage のメソッド), [471](#)  
[getloadavg\(\)](#) (os モジュール), [238](#)  
[getlocale\(\)](#) (locale モジュール), [341](#)  
[getLogger\(\)](#) (logging モジュール), [356](#)  
[getlogin\(\)](#) (os モジュール), [221](#)  
[getmaintype\(\)](#) (Message のメソッド), [585](#)  
[getmark\(\)](#)  
     [aifc](#) のメソッド, [664](#)  
     [AU\\_read](#) のメソッド, [668](#)  
     [Wave\\_read](#) のメソッド, [670](#)  
[getmarkers\(\)](#)  
     [aifc](#) のメソッド, [664](#)  
     [AU\\_read](#) のメソッド, [668](#)  
     [Wave\\_read](#) のメソッド, [670](#)  
[getmaxyx\(\)](#) (window のメソッド), [285](#)  
[getmcolor\(\)](#) (fl モジュール), [759](#)  
[getmember\(\)](#) (TarFile のメソッド), [417](#)  
[getmembers\(\)](#)  
     inspect モジュール, [73](#)  
     TarFile のメソッド, [417](#)  
[getMessage\(\)](#) (SAXException のメソッド), [644](#)  
[getmessagefilename\(\)](#) (Folder のメソッド), [583](#)  
[getMessageID\(\)](#) (NTEventLogHandler のメソッド), [363](#)  
[getmodule\(\)](#) (inspect モジュール), [74](#)  
[getmoduleinfo\(\)](#) (inspect モジュール), [73](#)  
[getmodulename\(\)](#) (inspect モジュール), [73](#)  
[getmouse\(\)](#) (curses モジュール), [279](#)  
[getmtime\(\)](#) (os.path モジュール), [240](#)  
[getName\(\)](#) (Thread のメソッド), [395](#)  
[getname\(\)](#) (Chunk のメソッド), [671](#)  
[getNameByQName\(\)](#) (AttributesNS のメソッド), [655](#)  
[getnameinfo\(\)](#) (socket モジュール), [377](#)  
[getNames\(\)](#) (Attributes のメソッド), [654](#)  
[getnames\(\)](#) (TarFile のメソッド), [417](#)  
[getnamespace\(\)](#) (XMLParser のメソッド), [656](#)  
[getnchannels\(\)](#)  
     [aifc](#) のメソッド, [664](#)  
     [AU\\_read](#) のメソッド, [667](#)  
     [Wave\\_read](#) のメソッド, [669](#)

getnframes()   
     aifc のメソッド, 664   
     AU\_read のメソッド, 667   
     Wave\_read のメソッド, 669   
 getopt() (getopt モジュール), 299   
 getopt (standard module), 299   
 GetoptError (getopt の例外), 300   
 getoutput() (commands モジュール), 444   
 getpagesize() (resource モジュール), 442   
 getparam() (Message のメソッド), 585   
 getparams()   
     aifc のメソッド, 664   
     al モジュール, 752   
     AU\_read のメソッド, 667   
     Wave\_read のメソッド, 669   
 getparyx() (window のメソッド), 285   
 getpass() (getpass モジュール), 276   
 getpass (standard module), 276   
 getpath() (MH のメソッド), 583   
 getpeername() (socket のメソッド), 380   
 getpgrp() (os モジュール), 221   
 getpid() (os モジュール), 221   
 getplist() (Message のメソッド), 585   
 getpos() (HTMLParser のメソッド), 612   
 getppid() (os モジュール), 221   
 getpreferredencoding() (locale モジュール), 341   
 getprofile() (MH のメソッド), 582   
 getProperty() (XMLReader のメソッド), 653   
 getprotobyname() (socket モジュール), 377   
 getPublicId()   
     InputSource のメソッド, 653   
     Locator のメソッド, 653   
 getpwall() (pwd モジュール), 427   
 getpwnam() (pwd モジュール), 427   
 getpwuid() (pwd モジュール), 427   
 getQNameByName() (AttributesNS のメソッド), 655   
 getQNames() (AttributesNS のメソッド), 655   
 getqueuesize() (audio configuration のメソッド), 752   
 getquota() (IMAP4\_stream のメソッド), 501   
 getquotaroot() (IMAP4\_stream のメソッド), 502   
 getrawheader() (AddressList のメソッド), 595   
 getreader() (codecs モジュール), 144   
 getrecursionlimit() (sys モジュール), 48   
 getrefcount() (sys モジュール), 48   
 getresponse() (HTTPSConnection のメソッド), 492   
 getrlimit() (resource モジュール), 440   
 getrusage() (resource モジュール), 441   
 getsampfmt() (audio configuration のメソッド), 752   
 getsample() (audioop モジュール), 660   
 getsampwidth()   
     aifc のメソッド, 664   
     AU\_read のメソッド, 667   
     Wave\_read のメソッド, 669   
 getsequences() (Folder のメソッド), 584   
 getsequencesfilename() (Folder のメソッド), 583   
 getservbyname() (socket モジュール), 377   
 getsignal() (signal モジュール), 373   
 getsize()   
     Chunk のメソッド, 671   
     os.path モジュール, 240   
 getsizes() (imgfile モジュール), 767   
 getslice() (operator モジュール), 68   
 getsockname() (socket のメソッド), 380   
 getsockopt() (socket のメソッド), 380   
 getsourcefile() (inspect モジュール), 74   
 getstate() (random モジュール), 186   
 getstatus()   
     audio port のメソッド, 753   
     CD player のメソッド, 755   
     commands モジュール, 444   
 getstatusoutput() (commands モジュール), 444   
 getstr() (window のメソッド), 285   
 getstrwidth() (fm モジュール), 764   
 getSubject() (SMTPHandler のメソッド), 364   
 getsubtype() (Message のメソッド), 585   
 getSystemId()   
     InputSource のメソッド, 653   
     Locator のメソッド, 653   
 getsyx() (curses モジュール), 279   
 gettarinfo() (TarFile のメソッド), 418   
 gettempdir() (tempfile モジュール), 329   
 gettempprefix() (tempfile モジュール), 330   
 getTestCaseNames() (TestLoader のメソッド), 177   
 gettext()   
     gettext モジュール, 346

GNUTranslations のメソッド, 349  
NullTranslations のメソッド, 348  
gettext (standard module), 345  
gettimeout() (socket のメソッド), 381  
gettrackinfo() (CD player のメソッド), 755  
getType() (Attributes のメソッド), 654  
gettype() (Message のメソッド), 585  
getuid() (os モジュール), 221  
getuser() (getpass モジュール), 277  
getValue() (Attributes のメソッド), 654  
getvalue() (StringIO のメソッド), 140  
getValueByQName() (AttributesNS のメソッド), 655  
getweakrefcount() (weakref モジュール), 55  
getweakrefs() (weakref モジュール), 55  
getwelcome()  
    FTP のメソッド, 495  
    NNTPDataError のメソッド, 506  
    POP3 のメソッド, 498  
getwidth() (audio configuration のメソッド), 752  
getwin() (curses モジュール), 279  
getwindowsversion() (sys モジュール), 49  
getwriter() (codecs モジュール), 144  
getyx() (window のメソッド), 285  
GL (standard module), 766  
gl (built-in module), 764  
glob() (glob モジュール), 336  
glob  
    standard module, 336  
    標準モジュール, 336  
globals() (モジュール), 9  
gmtime() (time モジュール), 271  
GNOME, 350  
gnu\_getopt() (getopt モジュール), 299  
Gopher  
    protocol, 480, 497  
gopher\_open() (GopherHandler のメソッド), 489  
gopher\_proxy, 477  
GopherError (urllib2 の例外), 483  
GopherHandler (urllib2 のクラス), 484  
gopherlib (standard module), 497  
goto() (turtle モジュール), 706  
Graphical User Interface, 687  
Greenwich Mean Time, 269  
grey22grey() (imageop モジュール), 663

grey2grey2() (imageop モジュール), 663  
grey2grey4() (imageop モジュール), 663  
grey2mono() (imageop モジュール), 663  
grey42grey() (imageop モジュール), 663  
group()  
    MatchObject のメソッド, 125  
    NNTPDataError のメソッド, 507  
groupdict() (MatchObject のメソッド), 126  
groupindex (RegexObject の属性), 125  
groups() (MatchObject のメソッド), 126  
grp (built-in module), 427  
gt() (operator モジュール), 66  
guess\_all\_extensions() (mimetypes モジュール), 586  
guess\_extension()  
    MimeTypes のメソッド, 588  
    mimetypes モジュール, 586  
guess\_type()  
    MimeTypes のメソッド, 588  
    mimetypes モジュール, 586  
GUI, 687  
gzip (standard module), 408  
GzipFile (gzip のクラス), 408

## H

halfdelay() (curses モジュール), 280  
handle()  
    BaseHTTPRequestHandler のメソッド, 521  
    SocketServer モジュール, 519  
handle() (のメソッド), 359, 360  
handle\_accept() (dispatcher のメソッド), 536  
handle\_authentication\_request() (AbstractBasicAuthHandler のメソッド), 488  
handle\_authentication\_request() (AbstractDigestAuthHandler のメソッド), 488  
handle\_cdata() (XMLParser のメソッド), 657  
handle\_charref()  
    HTMLParser のメソッド, 613  
    SGMLParser のメソッド, 615  
    XMLParser のメソッド, 656  
handle\_close() (async\_chat のメソッド), 538  
handle\_close() (dispatcher のメソッド), 536  
handle\_comment()  
    HTMLParser のメソッド, 613  
    SGMLParser のメソッド, 615  
    XMLParser のメソッド, 656



[handle\\_connect\(\)](#) (dispatcher のメソッド), [536](#)  
[handle\\_data\(\)](#)  
     HTMLParser のメソッド, [612](#)  
     SGMLParser のメソッド, [615](#)  
     XMLParser のメソッド, [656](#)  
[handle\\_decl\(\)](#)  
     HTMLParser のメソッド, [613](#)  
     SGMLParser のメソッド, [615](#)  
[handle\\_doctype\(\)](#) (XMLParser のメソッド), [656](#)  
[handle\\_endtag\(\)](#)  
     HTMLParser のメソッド, [612](#)  
     SGMLParser のメソッド, [615](#)  
     XMLParser のメソッド, [656](#)  
[handle\\_entityref\(\)](#)  
     HTMLParser のメソッド, [613](#)  
     SGMLParser のメソッド, [615](#)  
[handle\\_error\(\)](#)  
     dispatcher のメソッド, [536](#)  
     SocketServer モジュール, [518](#)  
[handle\\_expt\(\)](#) (dispatcher のメソッド), [536](#)  
[handle\\_image\(\)](#) (HTMLParser のメソッド), [618](#)  
[handle\\_one\\_request\(\)](#) (BaseHTTPRequestHandler のメソッド), [521](#)  
[handle\\_pi\(\)](#) (HTMLParser のメソッド), [613](#)  
[handle\\_proc\(\)](#) (XMLParser のメソッド), [657](#)  
[handle\\_read\(\)](#) (async\_chat のメソッド), [538](#)  
[handle\\_read\(\)](#) (dispatcher のメソッド), [536](#)  
[handle\\_request\(\)](#) (SimpleXMLRPCRequestHandler のメソッド), [533](#)  
[handle\\_request\(\)](#) (SocketServer モジュール), [517](#)  
[handle\\_special\(\)](#) (XMLParser のメソッド), [657](#)  
[handle\\_startendtag\(\)](#) (HTMLParser のメソッド), [612](#)  
[handle\\_starttag\(\)](#)  
     HTMLParser のメソッド, [612](#)  
     SGMLParser のメソッド, [614](#)  
     XMLParser のメソッド, [656](#)  
[handle\\_write\(\)](#) (async\_chat のメソッド), [539](#)  
[handle\\_write\(\)](#) (dispatcher のメソッド), [536](#)  
[handle\\_xml\(\)](#) (XMLParser のメソッド), [656](#)  
[handleError\(\)](#)  
     のメソッド, [360](#)  
         SocketHandler のメソッド, [361](#)  
[handler\(\)](#) (cgith モジュール), [476](#)  
[has\\_colors\(\)](#) (curses モジュール), [279](#)  
[has\\_data\(\)](#) (Request のメソッド), [485](#)  
[has\\_extn\(\)](#) (SMTP のメソッド), [510](#)  
[has\\_header\(\)](#) (Sniffer のメソッド), [609](#)  
[has\\_ic\(\)](#) (curses モジュール), [279](#)  
[has\\_il\(\)](#) (curses モジュール), [279](#)  
[has\\_ipv6](#) (socket のデータ), [376](#)  
[has\\_key\(\)](#)  
     のメソッド, [403](#)  
     curses モジュール, [280](#)  
     dictionary method, [29](#)  
     Message のメソッド, [551](#)  
[has\\_option\(\)](#)  
     のメソッド, [316](#)  
     SafeConfigParser のメソッド, [206](#)  
[has\\_section\(\)](#) (SafeConfigParser のメソッド), [206](#)  
[hasattr\(\)](#) (モジュール), [9](#)  
[hasAttributes\(\)](#) (Node のメソッド), [630](#)  
[hasChildNodes\(\)](#) (Node のメソッド), [630](#)  
[hascompare](#) (dis のデータ), [735](#)  
[hasconst](#) (dis のデータ), [735](#)  
[hasFeature\(\)](#) (DOMImplementation のメソッド), [629](#)  
[hasfree](#) (dis のデータ), [735](#)  
[hash\(\)](#) (モジュール), [9](#)  
[hashopen\(\)](#) (bsddb モジュール), [402](#)  
[hasjabs](#) (dis のデータ), [735](#)  
[hasjrel](#) (dis のデータ), [735](#)  
[haslocal](#) (dis のデータ), [735](#)  
[hasname](#) (dis のデータ), [735](#)  
[have\\_unicode](#) (test.test\_support のデータ), [181](#)  
[head\(\)](#) (NNTPDataError のメソッド), [507](#)  
[Header](#) (email.Header のクラス), [563](#)  
[header\\_encode\(\)](#) (Charset のメソッド), [566](#)  
[header\\_encoding](#) (email.Charset のデータ), [565](#)  
[header\\_offset](#) (ZipInfo の属性), [415](#)  
[headers](#)  
     MIME, [467](#), [586](#)  
[headers](#)  
     AddressList の属性, [596](#)  
     BaseHTTPRequestHandler の属性, [520](#)  
     ServerProxy の属性, [531](#)  
[heapify\(\)](#) (heapq モジュール), [191](#)



heapmin() (msvcrt モジュール), 774  
 heappop() (heapq モジュール), 191  
 heappush() (heapq モジュール), 191  
 heapq (standard module), 190  
 heapreplace() (heapq モジュール), 191  
 hello() (SMTP のメソッド), 510  
 help  
     online, 157  
 help()  
     NNTPDataError のメソッド, 507  
     モジュール, 9  
 error (socket の例外), 375  
 hex() (モジュール), 9  
 hexadecimal  
     literals, 19  
 hexbin() (binhex モジュール), 600  
 hexdigest()  
     hmac のメソッド, 681  
     md5 のメソッド, 682  
     sha のメソッド, 683  
 hexdigits (string のデータ), 113  
 hexlify() (binascii モジュール), 599  
 hexversion (sys のデータ), 49  
 hidden() (のメソッド), 298  
 hide() (のメソッド), 298  
 hide\_form() (form のメソッド), 759  
 hide\_object() (FORMS object のメソッド), 762  
 HierarchyRequestErr (xml.dom の例外), 635  
 HIGHEST\_PROTOCOL (pickle のデータ), 79  
 hline() (window のメソッド), 285  
 HList (Tix のクラス), 702  
 hls\_to\_rgb() (colorsys モジュール), 672  
 hmac (standard module), 681  
 HOME, 109, 239  
 hosts (netrc の属性), 605  
 hotshot (standard module), 459  
 hotshot.stats (standard module), 460  
 hour  
     datetime の属性, 257  
     time の属性, 261  
 hsv\_to\_rgb() (colorsys モジュール), 672  
 HTML, 480, 611, 616  
 htmlentitydefs (standard module), 618  
 htmllib  
     standard module, 616  
     標準モジュール, 480  
 HTMLParseError (HTMLParser の例外), 613  
 HTMLParser  
     class in htmllib, 543  
     htmllib のクラス, 617  
     HTMLParser のクラス, 611  
     standard module, 611  
 htonl() (socket モジュール), 378  
 htons() (socket モジュール), 378  
 HTTP  
     httplib (standard module), 490  
     protocol, 467, 480, 490, 519  
 http\_error\_301() (HTTPRedirectHandler のメソッド), 487  
 http\_error\_302() (HTTPRedirectHandler のメソッド), 487  
 http\_error\_303() (HTTPRedirectHandler のメソッド), 487  
 http\_error\_307() (HTTPRedirectHandler のメソッド), 487  
 http\_error\_401() (HTTPBasicAuthHandler のメソッド), 488  
 http\_error\_401() (HTTPDigestAuthHandler のメソッド), 489  
 http\_error\_407() (ProxyBasicAuthHandler のメソッド), 488  
 http\_error\_407() (ProxyDigestAuthHandler のメソッド), 489  
 http\_error\_default() (BaseHandler のメソッド), 487  
 http\_open() (HTTPHandler のメソッド), 489  
 HTTP\_PORT (httplib のデータ), 490  
 http\_proxy, 477  
 HTTPBasicAuthHandler (urllib2 のクラス), 484  
 HTTPConnection (httplib のクラス), 490  
 httpd, 519  
 HTTPDefaultErrorHandler (urllib2 のクラス), 483  
 HTTPDigestAuthHandler (urllib2 のクラス), 484  
 HTTPError (urllib2 の例外), 483  
 HTTPException (httplib の例外), 491  
 HTTPHandler  
     logging のクラス, 365  
     urllib2 のクラス, 484  
 httplib (standard module), 490  
 HTTPPasswordMgr (urllib2 のクラス), 483

HTTPPasswordMgrWithDefaultRealm (urllib2 のクラス), 483  
HTTPRedirectHandler (urllib2 のクラス), 483  
https\_open() (HTTPSHandler のメソッド), 489  
HTTPS\_PORT (httplib のデータ), 490  
HTTPSConnection (httplib のクラス), 491  
HTTPServer (BaseHTTPServer のクラス), 519  
HTTPSHandler (urllib2 のクラス), 484  
hypertext, 616  
hypot() (math モジュール), 182  
**I**  
I (re のデータ), 122  
I/O control  
    buffering, 8, 223, 380  
    POSIX, 432, 433  
    tty, 432, 433  
    UNIX, 434  
ibufcount() (audio device のメソッド), 770  
id()  
    TestCase のメソッド, 175  
    モジュール, 9  
idcok() (window のメソッド), 285  
IDEA  
    cipher, 681  
ident (cd のデータ), 755  
identchars (Cmd の属性), 213  
Idle, 707  
idlok() (window のメソッド), 286  
IEEE-754, 59  
if  
    実行文, 17  
    statement, 17  
ifilter() (itertools モジュール), 200  
ifilterfalse() (itertools モジュール), 200  
ignorableWhitespace() (ContentHandler のメソッド), 648  
ignore() (Stats のメソッド), 457  
ignore\_errors() (codecs モジュール), 144  
ignore\_zeros=False (TarFile の属性), 418  
IGNORECASE (re のデータ), 122  
ihave() (NNTPDataError のメソッド), 508  
ihooks (標準モジュール), 3  
IllegalKeywordArgument (httplib の例外), 491  
imageop (built-in module), 662

imap() (itertools モジュール), 201  
IMAP4  
    protocol, 499  
IMAP4 (imaplib のクラス), 500  
IMAP4.abort (imaplib の例外), 500  
IMAP4.error (imaplib の例外), 500  
IMAP4.readonly (imaplib の例外), 500  
IMAP4\_SSL  
    protocol, 499  
IMAP4\_SSL (imaplib のクラス), 500  
IMAP4\_stream  
    protocol, 499  
IMAP4\_stream (imaplib のクラス), 500  
imaplib (standard module), 499  
imgfile (built-in module), 766  
imghdr (standard module), 673  
immedok() (window のメソッド), 286  
ImmutableSet (sets のクラス), 196  
imp  
    built-in module, 96  
    組み込みモジュール, 3  
import  
    実行文, 3, 96  
    statement, 3, 96  
Import module, 708  
ImportError (exceptions の例外), 38  
ImproperConnectionState (httplib の例外), 491  
**in**  
    演算子, 18, 22  
    operator, 18, 22  
in\_table\_a1() (stringprep モジュール), 155  
in\_table\_b1() (stringprep モジュール), 155  
in\_table\_c11() (stringprep モジュール), 155  
in\_table\_c11\_c12() (stringprep モジュール), 155  
in\_table\_c12() (stringprep モジュール), 155  
in\_table\_c21() (stringprep モジュール), 155  
in\_table\_c21\_c22() (stringprep モジュール), 155  
in\_table\_c22() (stringprep モジュール), 155  
in\_table\_c3() (stringprep モジュール), 155  
in\_table\_c4() (stringprep モジュール), 155  
in\_table\_c5() (stringprep モジュール), 155  
in\_table\_c6() (stringprep モジュール), 155  
in\_table\_c7() (stringprep モジュール), 155  
in\_table\_c8() (stringprep モジュール), 155

[in\\_table\\_c9\(\)](#) (stringprep モジュール), [155](#)  
[in\\_table\\_d1\(\)](#) (stringprep モジュール), [155](#)  
[in\\_table\\_d2\(\)](#) (stringprep モジュール), [155](#)  
[INADDR\\_\\*](#) (socket のデータ), [375](#)  
[inch\(\)](#) (window のメソッド), [286](#)  
[Incomplete](#) (binascii の例外), [599](#)  
[IncompleteRead](#) (httplib の例外), [491](#)  
[IncrementalParser](#) (xml.sax.xmlreader のクラス), [651](#)  
[indentation](#), [709](#)  
[Independent JPEG Group](#), [767](#)  
[index\(\)](#)  
     array のメソッド, [194](#)  
     string のメソッド, [23](#)  
     string モジュール, [115](#)  
[index](#) (cd のデータ), [754](#)  
[index\(\)](#) (list method), [28](#)  
[IndexError](#) (exceptions の例外), [38](#)  
[indexOf\(\)](#) (operator モジュール), [68](#)  
[IndexSizeErr](#) (xml.dom の例外), [635](#)  
[inet\\_aton\(\)](#) (socket モジュール), [378](#)  
[inet\\_ntoa\(\)](#) (socket モジュール), [378](#)  
[inet\\_ntop\(\)](#) (socket モジュール), [378](#)  
[inet\\_pton\(\)](#) (socket モジュール), [378](#)  
[infile](#) (shlex の属性), [216](#)  
[Infinity](#), [8](#), [114](#)  
[info\(\)](#)  
     のメソッド, [358](#)  
     logging モジュール, [356](#)  
     NullTranslations のメソッド, [348](#)  
[infolist\(\)](#) (ZipFile のメソッド), [412](#)  
[InfoSeek Corporation](#), [451](#)  
[ini file](#), [205](#)  
[init\(\)](#)  
     fm モジュール, [763](#)  
     mimetypes モジュール, [586](#)  
[init\\_builtin\(\)](#) (imp モジュール), [98](#)  
[init\\_color\(\)](#) (curses モジュール), [280](#)  
[init\\_frozen\(\)](#) (imp モジュール), [98](#)  
[init\\_pair\(\)](#) (curses モジュール), [280](#)  
[inited](#) (mimetypes のデータ), [587](#)  
[initial\\_indent](#) (TextWrapper の属性), [142](#)  
[initscr\(\)](#) (curses モジュール), [280](#)  
[input\(\)](#)  
     fileinput モジュール, [208](#)  
     モジュール, [9](#)  
     組み込み関数, [51](#)  
[input\\_charset](#) (email.Charset のデータ), [565](#)  
[input\\_codec](#) (email.Charset のデータ), [565](#)  
[InputOnly](#) (Tix のクラス), [703](#)  
[InputSource](#) (xml.sax.xmlreader のクラス), [651](#)  
[InputType](#) (cStringIO のデータ), [141](#)  
[insch\(\)](#) (window のメソッド), [286](#)  
[insdelln\(\)](#) (window のメソッド), [286](#)  
[insert\(\)](#)  
     array のメソッド, [194](#)  
     list method, [28](#)  
[insert\\_text\(\)](#) (readline モジュール), [421](#)  
[insertBefore\(\)](#) (Node のメソッド), [630](#)  
[insertln\(\)](#) (window のメソッド), [286](#)  
[insnstr\(\)](#) (window のメソッド), [286](#)  
[insort\(\)](#) (bisect モジュール), [189](#)  
[insort\\_left\(\)](#) (bisect モジュール), [189](#)  
[insort\\_right\(\)](#) (bisect モジュール), [189](#)  
[inspect](#) (standard module), [70](#)  
[insstr\(\)](#) (window のメソッド), [286](#)  
[install\(\)](#)  
     gettext モジュール, [347](#)  
     NullTranslations のメソッド, [348](#)  
[install\\_opener\(\)](#) (urllib2 モジュール), [482](#)  
[instance\(\)](#) (new モジュール), [107](#)  
[instancemethod\(\)](#) (new モジュール), [108](#)  
[InstanceType](#) (types のデータ), [63](#)  
[instr\(\)](#) (window のメソッド), [286](#)  
[instream](#) (shlex の属性), [216](#)  
[int\(\)](#)  
     モジュール, [9](#)  
     組み込み関数, [19](#)  
[Int2AP\(\)](#) (imaplib モジュール), [500](#)  
[integer](#)  
     arbitrary precision, [684](#)  
     division, [19](#)  
     division, long, [19](#)  
     literals, [19](#)  
     literals, long, [19](#)  
     object, [19](#)  
     オブジェクト, [19](#)  
     types, operations on, [20](#)  
[Integrated Developement Environment](#), [707](#)  
[Intel/DVI ADPCM](#), [659](#)  
[interact\(\)](#)  
     code モジュール, [100](#)  
     InteractiveConsole のメソッド, [101](#)  
     Telnet のメソッド, [514](#)

InteractiveConsole (code のクラス), 100

InteractiveInterpreter (code のクラス), 100

intern() (モジュール), 16

internal\_attr (ZipInfo の属性), 414

Internaldate2tuple() (imaplib モジュール), 500

internalSubset (DocumentType の属性), 631

Internet, 465

Internet Config, 477

interpolation, string (%), 25

InterpolationDepthError (ConfigParser の例外), 206

InterpolationError (ConfigParser の例外), 205

InterpolationMissingOptionError (ConfigParser の例外), 206

InterpolationSyntaxError (ConfigParser の例外), 206

interpreter prompts, 50

interrupt\_main() (thread モジュール), 386

intro (Cmd の属性), 213

IntType (types のデータ), 62

InuseAttributeErr (xml.dom の例外), 635

inv() (operator モジュール), 67

InvalidAccessErr (xml.dom の例外), 635

InvalidCharacterErr (xml.dom の例外), 635

InvalidModificationErr (xml.dom の例外), 635

InvalidStateErr (xml.dom の例外), 635

InvalidURL (httplib の例外), 491

invert() (operator モジュール), 67

ioctl() (fcntl モジュール), 435

IOError (exceptions の例外), 38

IP\_\* (socket のデータ), 375

IPPORT\_\* (socket のデータ), 375

IPPROTO\_\* (socket のデータ), 375

IPV6\_\* (socket のデータ), 375

IRIS Font Manager, 763

IRIX

- threads, 387

is

- 演算子, 18
- operator, 18

is not

- 演算子, 18
- operator, 18

is\_() (operator モジュール), 66

is\_builtin() (imp モジュール), 98

IS\_CHARACTER\_JUNK() (difflib モジュール), 134

is\_data() (MultiFile のメソッド), 591

is\_empty() (fifo のメソッド), 540

is\_frozen() (imp モジュール), 98

is\_jython (test.test\_support のデータ), 181

IS\_LINE\_JUNK() (difflib モジュール), 134

is\_linetouched() (window のメソッド), 286

is\_multipart() (Message のメソッド), 549

is\_not() (operator モジュール), 67

is\_resource\_enabled() (test.test\_support モジュール), 181

is\_tarfile() (tarfile モジュール), 416

is\_wintouched() (window のメソッド), 286

is\_zipfile() (zipfile モジュール), 412

isabs() (os.path モジュール), 240

isAlive() (Thread のメソッド), 395

isalnum()

- curses.ascii モジュール, 296
- string のメソッド, 23

isalpha()

- curses.ascii モジュール, 296
- string のメソッド, 23

isascii() (curses.ascii モジュール), 296

isatty()

- Chunk のメソッド, 671
- file のメソッド, 31
- os モジュール, 224

isblank() (curses.ascii モジュール), 297

isblk() (TarInfo のメソッド), 420

isbuiltin() (inspect モジュール), 73

isCallable() (operator モジュール), 69

ischr() (TarInfo のメソッド), 420

isclass() (inspect モジュール), 73

iscntrl() (curses.ascii モジュール), 297

iscode() (inspect モジュール), 73

iscomment() (AddressList のメソッド), 595

isctrl() (curses.ascii モジュール), 297

isDaemon() (Thread のメソッド), 395

isdatadescriptor() (inspect モジュール), 74

isdev() (TarInfo のメソッド), 420

isdigit()

- curses.ascii モジュール, 297
- string のメソッド, 23

isdir()

- os.path モジュール, 240
- TarInfo のメソッド, 420
- isEnabled() (gc モジュール), 52
- isEnabledFor() ( のメソッド), 358
- isendwin() (curses モジュール), 280
- ISEOF() (token モジュール), 730
- isexpr()
  - AST のメソッド, 723
  - parser モジュール, 722
- isfifo() (TarInfo のメソッド), 420
- isfile()
  - os.path モジュール, 240
  - TarInfo のメソッド, 419
- isfirstline() (fileinput モジュール), 209
- isframe() (inspect モジュール), 73
- isfunction() (inspect モジュール), 73
- isgraph() (curses.ascii モジュール), 297
- isheader() (AddressList のメソッド), 594
- isinstance() (モジュール), 10
- iskeyword() (keyword モジュール), 730
- islast() (AddressList のメソッド), 595
- isleap() (calendar モジュール), 211
- islice() (itertools モジュール), 201
- islink() (os.path モジュール), 240
- islnk() (TarInfo のメソッド), 420
- islower()
  - curses.ascii モジュール, 297
  - string のメソッド, 24
- isMappingType() (operator モジュール), 69
- ismeta() (curses.ascii モジュール), 297
- ismethod() (inspect モジュール), 73
- ismethoddescriptor() (inspect モジュール), 73
- ismodule() (inspect モジュール), 73
- ismount() (os.path モジュール), 240
- ISNONTERMINAL() (token モジュール), 730
- isNumberType() (operator モジュール), 69
- isocalendar()
  - date のメソッド, 255
  - datetime のメソッド, 260
- isoformat()
  - date のメソッド, 255
  - datetime のメソッド, 260
  - time のメソッド, 262
- isowekday()
  - date のメソッド, 255
  - datetime のメソッド, 260
- isprint() (curses.ascii モジュール), 297
- ispunct() (curses.ascii モジュール), 297
- isqueued() (fl モジュール), 759
- isreadable()
  - pprint モジュール, 105
  - PrettyPrinter のメソッド, 105
- isrecursive()
  - pprint モジュール, 105
  - PrettyPrinter のメソッド, 105
- isreg() (TarInfo のメソッド), 420
- isReservedKey() (Morsel のメソッド), 525
- isroutine() (inspect モジュール), 73
- isSameNode() (Node のメソッド), 630
- isSequenceType() (operator モジュール), 69
- isSet() (Event のメソッド), 393
- isspace()
  - curses.ascii モジュール, 297
  - string のメソッド, 24
- isstdin() (fileinput モジュール), 209
- issubclass() (モジュール), 10
- issuite()
  - AST のメソッド, 723
  - parser モジュール, 722
- issym() (TarInfo のメソッド), 420
- ISTERMINAL() (token モジュール), 730
- istitle() (string のメソッド), 24
- istraceback() (inspect モジュール), 73
- isupper()
  - curses.ascii モジュール, 297
  - string のメソッド, 24
- isxdigit() (curses.ascii モジュール), 297
- item()
  - NamedNodeMap のメソッド, 634
  - NodeList のメソッド, 631
- items() (Message のメソッド), 551
- items() (SafeConfigParser のメソッド), 207, 208
- items() (dictionary method), 29
- itemsizes (array の属性), 194
- iter() (モジュール), 10
- iterator protocol, 20
- iteritems() (dictionary method), 29
- iterkeys() (dictionary method), 29
- itertools (standard module), 199
- intervalues() (dictionary method), 29
- izip() (itertools モジュール), 201

## J

Jansen, Jack, [601](#)

JFIF, [768](#)

### 実行文

- [assert](#), [38](#)
- [del](#), [28](#), [29](#)
- [except](#), [37](#)
- [exec](#), [35](#)
- [if](#), [17](#)
- [import](#), [3](#), [96](#)
- [print](#), [17](#)
- [raise](#), [37](#)
- [try](#), [37](#)
- [while](#), [17](#)

[join\(\)](#)

- [os.path](#) モジュール, [240](#)
- [string](#) のメソッド, [24](#)
- [string](#) モジュール, [116](#)
- [Thread](#) のメソッド, [395](#)

[joinfields\(\)](#) ([string](#) モジュール), [116](#)

[jpeg](#) (built-in module), [767](#)

[js\\_output\(\)](#)

- [BaseCookie](#) のメソッド, [525](#)
- [Morsel](#) のメソッド, [526](#)

[jumpahead\(\)](#) ([random](#) モジュール), [186](#)

## K

[kbhit\(\)](#) ([msvcrt](#) モジュール), [774](#)

[KDEDIR](#), [467](#)

[key](#) ([Morsel](#) の属性), [525](#)

[KeyboardInterrupt](#) (exceptions の例外), [38](#)

[KeyError](#) (exceptions の例外), [38](#)

[keyname\(\)](#) ([curses](#) モジュール), [280](#)

[keypad\(\)](#) ([window](#) のメソッド), [286](#)

[keys\(\)](#)

- のメソッド, [403](#)
- dictionary method, [29](#)
- [Message](#) のメソッド, [551](#)

[keyword](#) (standard module), [730](#)

[kill\(\)](#) ([os](#) モジュール), [234](#)

[killchar\(\)](#) ([curses](#) モジュール), [280](#)

[killpg\(\)](#) ([os](#) モジュール), [234](#)

[knee](#) (モジュール), [99](#)

[knownfiles](#) ([mimetypes](#) のデータ), [587](#)

Kuchling, Andrew, [681](#)

[kwlist](#) ([keyword](#) のデータ), [730](#)

## L

[L](#) ([re](#) のデータ), [122](#)

[LabelEntry](#) ([Tix](#) のクラス), [701](#)

[LabelFrame](#) ([Tix](#) のクラス), [701](#)

[LambdaType](#) ([types](#) のデータ), [63](#)

[LANG](#), [339](#), [340](#), [345](#), [347](#)

[LANGUAGE](#), [345](#), [347](#)

[language](#)

- [C](#), [19](#), [20](#)

[large files](#), [426](#)

[last\(\)](#)

- のメソッド, [404](#)
- [dbhash](#) のメソッド, [401](#)
- [NNTPDataError](#) のメソッド, [507](#)

[last](#) ([MultiFile](#) の属性), [592](#)

[last\\_traceback](#) ([sys](#) のデータ), [49](#)

[last\\_type](#) ([sys](#) のデータ), [49](#)

[last\\_value](#) ([sys](#) のデータ), [49](#)

[lastChild](#) ([Node](#) の属性), [629](#)

[lastcmd](#) ([Cmd](#) の属性), [213](#)

[lastgroup](#) ([MatchObject](#) の属性), [127](#)

[lastindex](#) ([MatchObject](#) の属性), [127](#)

[lastpart\(\)](#) ([MimeWriter](#) のメソッド), [589](#)

[LC\\_ALL](#), [345](#), [347](#)

[LC\\_ALL](#) ([locale](#) のデータ), [342](#)

[LC\\_COLLATE](#) ([locale](#) のデータ), [342](#)

[LC\\_CTYPE](#) ([locale](#) のデータ), [341](#)

[LC\\_MESSAGES](#), [345](#), [347](#)

[LC\\_MESSAGES](#) ([locale](#) のデータ), [342](#)

[LC\\_MONETARY](#) ([locale](#) のデータ), [342](#)

[LC\\_NUMERIC](#) ([locale](#) のデータ), [342](#)

[LC\\_TIME](#) ([locale](#) のデータ), [342](#)

[ldexp\(\)](#) ([math](#) モジュール), [183](#)

[le\(\)](#) ([operator](#) モジュール), [66](#)

[leapdays\(\)](#) ([calendar](#) モジュール), [211](#)

[leaveok\(\)](#) ([window](#) のメソッド), [286](#)

[left\(\)](#) ([turtle](#) モジュール), [706](#)

[left\\_list](#) ([dircmp](#) の属性), [247](#)

[left\\_only](#) ([dircmp](#) の属性), [247](#)

[len\(\)](#)

- モジュール, [10](#)

- 組み込み関数, [22](#), [29](#)

[length](#)

- [NamedNodeMap](#) の属性, [634](#)

- [NodeList](#) の属性, [631](#)

[letters](#) ([string](#) のデータ), [113](#)



- level (MultiFile の属性), 592
- library (dbm のデータ), 430
- light-weight processes, 386
- lin2adpcm() (audioop モジュール), 660
- lin2adpcm3() (audioop モジュール), 660
- lin2lin() (audioop モジュール), 660
- lin2ulaw() (audioop モジュール), 661
- line-buffered I/O, 8
- linecache (standard module), 76
- lineno() (fileinput モジュール), 209
- lineno
  - class descriptor の属性, 732
  - ExpatError の属性, 623
  - function descriptor の属性, 733
  - shlex の属性, 216
- LINES, 283
- linesep (os のデータ), 239
- lineterminator (Dialect の属性), 609
- link() (os モジュール), 227
- linkname (TarInfo の属性), 419
- list
  - object, 21, 28
  - オブジェクト, 21, 28
  - type, operations on, 28
- list()
  - IMAP4\_stream のメソッド, 502
  - NNTPDataError のメソッド, 506
  - POP3 のメソッド, 498
  - TarFile のメソッド, 417
  - モジュール, 10
- list\_dialects() (csv モジュール), 608
- listallfolders() (MH のメソッド), 583
- listallsubfolders() (MH のメソッド), 583
- listdir()
  - dircache モジュール, 242
  - os モジュール, 227
- listen()
  - dispatcher のメソッド, 537
  - logging モジュール, 367
  - socket のメソッド, 380
- listfolders() (MH のメソッド), 583
- listmessages() (Folder のメソッド), 583
- ListNoteBook (Tix のクラス), 703
- listsubfolders() (MH のメソッド), 583
- ListType (types のデータ), 62
- literals
  - complex number, 19
  - floating point, 19
  - hexadecimal, 19
  - integer, 19
  - long integer, 19
  - numeric, 19
  - octal, 19
- ljust()
  - string のメソッド, 24
  - string モジュール, 116
- LK\_LOCK (msvcrt のデータ), 773
- LK\_NBLCK (msvcrt のデータ), 773
- LK\_NBRLCK (msvcrt のデータ), 773
- LK\_RLCK (msvcrt のデータ), 773
- LK\_UNLCK (msvcrt のデータ), 773
- LNAME, 277
- load()
  - BaseCookie のメソッド, 525
  - hotshot.stats モジュール, 460
  - marshal モジュール, 93
  - pickle モジュール, 79
  - Unpickler のメソッド, 81
- load\_compiled() (imp モジュール), 98
- load\_dynamic() (imp モジュール), 98
- load\_module() (imp モジュール), 96
- load\_source() (imp モジュール), 98
- loads()
  - marshal モジュール, 93
  - pickle モジュール, 79
- loadTestsFromModule() (TestLoader のメソッド), 177
- loadTestsFromName() (TestLoader のメソッド), 177
- loadTestsFromNames() (TestLoader のメソッド), 177
- loadTestsFromTestCase() (TestLoader のメソッド), 177
- LOCALE (re のデータ), 122
- locale (standard module), 339
- localeconv() (locale モジュール), 339
- localName
  - Attr の属性, 634
  - Node の属性, 629
- locals() (モジュール), 10
- localtime() (time モジュール), 271
- Locator (xml.sax.xmlreader のクラス), 651
- Lock() (threading モジュール), 388
- lock()

- mutex のメソッド, 276
- posixfile モジュール, 438
- lock\_held() (imp モジュール), 97
- locked() (lock のメソッド), 387
- lockf()
  - fcntl モジュール, 435
  - in module fcntl, 437
- locking() (msvrt モジュール), 773
- LockType (thread のデータ), 386
- log()
  - のメソッド, 358
  - cmath モジュール, 184
  - math モジュール, 183
- log10()
  - cmath モジュール, 184
  - math モジュール, 183
- log\_data\_time\_string() (BaseHTTPRequestHandler のメソッド), 521
- log\_error() (BaseHTTPRequestHandler のメソッド), 521
- log\_message() (BaseHTTPRequestHandler のメソッド), 521
- log\_request() (BaseHTTPRequestHandler のメソッド), 521
- logging
  - Errors, 354
- logging (standard module), 354
- login()
  - FTP のメソッド, 495
  - IMAP4\_stream のメソッド, 502
  - SMTP のメソッド, 510
- login\_cram\_md5() (IMAP4\_stream のメソッド), 502
- LOGNAME, 221, 277
- lognormvariate() (random モジュール), 187
- logout() (IMAP4\_stream のメソッド), 502
- LogRecord (logging のクラス), 366
- long
  - integer division, 19
  - integer literals, 19
- long()
  - モジュール, 10
  - 組み込み関数, 19, 115
- long integer
  - object, 19
  - オブジェクト, 19
- longimagedata() (rgbimg モジュール), 673

- longname() (curses モジュール), 280
- longstoimage() (rgbimg モジュール), 673
- LongType (types のデータ), 62
- lookup()
  - codecs モジュール, 143
  - unicodedata モジュール, 153
- lookup\_error() (codecs モジュール), 144
- LookupError (exceptions の例外), 37
- loop() (asyncore モジュール), 535
- lower()
  - string のメソッド, 24
  - string モジュール, 115
- lowercase (string のデータ), 113
- lseek() (os モジュール), 224
- lshift() (operator モジュール), 67
- lstat() (os モジュール), 227
- lstrip()
  - string のメソッド, 24
  - string モジュール, 116
- lsub() (IMAP4\_stream のメソッド), 502
- lt() (operator モジュール), 66
- Lundh, Fredrik, 767

## M

- M(re のデータ), 122
- macros (netrc の属性), 605
- mailbox
  - standard module, 580
  - 標準モジュール, 593
- mailcap (standard module), 579
- Maildir (mailbox のクラス), 581
- main()
  - py\_compile モジュール, 733
  - unittest モジュール, 173
- major() (os モジュール), 228
- make\_form() (fl モジュール), 758
- make\_header() (email.Header モジュール), 564
- make\_msgid() (email.Utils モジュール), 570
- make\_parser() (xml.sax モジュール), 643
- makedev() (os モジュール), 228
- makedirs() (os モジュール), 228
- makefile() (socket のメソッド), 380
- makefolder() (MH のメソッド), 583
- makeLogRecord() (logging モジュール), 357
- makePickle() (SocketHandler のメソッド), 362
- makeRecord() ( のメソッド), 359
- makeSocket()

- DatagramHandler のメソッド, [362](#)
- SocketHandler のメソッド, [362](#)
- maketrans() (string モジュール), [115](#)
- map() (モジュール), [10](#)
- map\_table\_b2() (stringprep モジュール), [155](#)
- map\_table\_b3() (stringprep モジュール), [155](#)
- mapcolor() (fl モジュール), [759](#)
- mapping
  - object, [29](#)
  - オブジェクト, [29](#)
  - types, operations on, [29](#)
- maps() (nis モジュール), [443](#)
- marshal (built-in module), [92](#)
- marshalling
  - objects, [77](#)
- masking
  - operations, [20](#)
- match()
  - nis モジュール, [442](#)
  - RegexObject のメソッド, [124](#)
  - re モジュール, [123](#)
- math
  - built-in module, [182](#)
  - 組み込みモジュール, [20](#), [185](#)
- max()
  - audioop モジュール, [661](#)
  - モジュール, [11](#)
  - 組み込み関数, [22](#)
- max
  - date の属性, [254](#)
  - datetime の属性, [257](#)
  - time の属性, [261](#)
  - timedelta の属性, [252](#)
- MAX\_INTERPOLATION\_DEPTH (ConfigParser のデータ), [206](#)
- maxdict (Repr の属性), [106](#)
- maxint (sys のデータ), [49](#)
- MAXLEN (mimify のデータ), [590](#)
- maxlevel (Repr の属性), [106](#)
- maxlist (Repr の属性), [106](#)
- maxlong (Repr の属性), [106](#)
- maxother (Repr の属性), [107](#)
- maxpp() (audioop モジュール), [661](#)
- maxstring (Repr の属性), [106](#)
- maxtuple (Repr の属性), [106](#)
- maxunicode (sys のデータ), [49](#)
- MAXYEAR (datetime のデータ), [250](#)
- MB\_ICONASTERISK (winsound のデータ), [781](#)
- MB\_ICONEXCLAMATION (winsound のデータ), [781](#)
- MB\_ICONHAND (winsound のデータ), [781](#)
- MB\_ICONQUESTION (winsound のデータ), [781](#)
- MB\_OK (winsound のデータ), [781](#)
- md5() (md5 モジュール), [682](#)
- md5 (built-in module), [682](#)
- MemoryError (exceptions の例外), [38](#)
- MemoryHandler (logging のクラス), [364](#)
- Message
  - email.Message のクラス, [548](#)
  - in module mimetools, [520](#)
  - mhlib のクラス, [582](#)
  - mimetools のクラス, [584](#)
  - rfc822 のクラス, [593](#)
- message digest, MD5, [682](#)
- message\_from\_file() (email.Parser モジュール), [557](#)
- message\_from\_string() (email.Parser モジュール), [557](#)
- MessageBeep() (winsound モジュール), [780](#)
- MessageClass (BaseHTTPRequestHandler の属性), [520](#)
- meta() (curses モジュール), [280](#)
- Meter (Tix のクラス), [701](#)
- method
  - object, [34](#)
  - オブジェクト, [34](#)
- methods (class descriptor の属性), [732](#)
- MethodType (types のデータ), [63](#)
- MH (mhlib のクラス), [582](#)
- mhlib (standard module), [582](#)
- MHMailbox (mailbox のクラス), [581](#)
- microsecond
  - datetime の属性, [258](#)
  - time の属性, [262](#)
- MIME
  - base64 encoding, [597](#)
  - content type, [586](#)
  - headers, [467](#), [586](#)
  - quoted-printable encoding, [600](#)
- mime\_decode\_header() (mimify モジュール), [590](#)
- mime\_encode\_header() (mimify モジュール), [590](#)
- MIMEAudio (email.Generator のクラス), [561](#)

MIMEBase (email.Generator のクラス), 560  
 MIMEImage (email.Generator のクラス), 561  
 MIMEMessage (email.Generator のクラス), 562  
 MIMEMultipart (email.Generator のクラス), 561  
 MIMENonMultipart (email.Generator のクラス), 560  
 MIMEText (email.Generator のクラス), 562  
 mimetools  
     standard module, 584  
     標準モジュール, 477  
 MimeTypes (mimetypes のクラス), 587  
 mimetypes (standard module), 586  
 MimeWriter  
     MimeWriter のクラス, 588  
     standard module, 588  
 mimify() (mimify モジュール), 589  
 mimify (standard module), 589  
 min()  
     モジュール, 11  
     組み込み関数, 22  
 min  
     date の属性, 253  
     datetime の属性, 257  
     time の属性, 261  
     timedelta の属性, 252  
 minmax() (audioop モジュール), 661  
 minor() (os モジュール), 228  
 minute  
     datetime の属性, 257  
     time の属性, 262  
 MINYEAR (datetime のデータ), 250  
 mirrored() (unicodedata モジュール), 153  
 misc\_header (Cmd の属性), 213  
 MissingSectionHeaderError (ConfigParser の例外), 206  
 mkd() (FTP のメソッド), 496  
 mkdir() (os モジュール), 228  
 mkdtemp() (tempfile モジュール), 328  
 mkfifo() (os モジュール), 227  
 mknod() (os モジュール), 227  
 mkstemp() (tempfile モジュール), 328  
 mktemp() (tempfile モジュール), 329  
 mktime() (time モジュール), 271  
 mktime\_tz()  
     email.Utils モジュール, 570  
     rfc822 モジュール, 594  
 mmap() (mmap モジュール), 398  
 mmap (built-in module), 397  
 MmdfMailbox (mailbox のクラス), 581  
 mod() (operator モジュール), 67  
 mode  
     file の属性, 33  
     TarInfo の属性, 419  
 modf() (math モジュール), 183  
 modified() (RobotFileParser のメソッド), 606  
 module  
     search path, 50, 76, 108  
 module() (new モジュール), 108  
 module  
     class descriptor の属性, 732  
     function descriptor の属性, 732  
 modules (sys のデータ), 49  
 ModuleType (types のデータ), 63  
 MON\_1 ... MON\_12 (locale のデータ), 343  
 mono2grey() (imageop モジュール), 663  
 month() (calendar モジュール), 211  
 month  
     date の属性, 254  
     datetime の属性, 257  
 monthcalendar() (calendar モジュール), 211  
 monthrange() (calendar モジュール), 211  
 more() (simple\_producer のメソッド), 539  
 Morsel (Cookie のクラス), 525  
 mouseinterval() (curses モジュール), 280  
 mousemask() (curses モジュール), 280  
 move()  
     のメソッド, 299, 399  
     shutil モジュール, 338  
     window のメソッド, 287  
 movemessage() (Folder のメソッド), 584  
 MP, GNU library, 684  
 mpz() (mpz モジュール), 684  
 mpz (built-in module), 684  
 MPZType (mpz のデータ), 684  
 msftoblock() (CD player のメソッド), 755  
 msftoframe() (cd モジュール), 754  
 msg() (Telnet のメソッド), 514  
 msg (httplib のデータ), 492  
 MSG\_\* (socket のデータ), 375  
 msvcrt (built-in module), 773  
 mt\_interact() (Telnet のメソッド), 514  
 mtime() (RobotFileParser のメソッド), 606  
 mtime (TarInfo の属性), 419

`mul()`  
    [audioop モジュール, 661](#)  
    [operator モジュール, 67](#)  
`MultiFile` (`multifile` のクラス), [590](#)  
`multifile` (standard module), [590](#)  
`MULTILINE` (`re` のデータ), [122](#)  
`mutable`  
    [sequence types, 28](#)  
    [sequence types, operations on, 28](#)  
`MutableString` (`UserString` のクラス), [65](#)  
`mutex`  
    [mutex のクラス, 276](#)  
    [standard module, 276](#)  
`mvderwin()` (`window` のメソッド), [287](#)  
`mvwin()` (`window` のメソッド), [287](#)

## N

`name()` (`unicodedata` モジュール), [153](#)  
`name`  
    [Attr の属性, 633](#)  
    [class descriptor の属性, 732](#)  
    [DocumentType の属性, 631](#)  
    [file の属性, 33](#)  
    [function descriptor の属性, 733](#)  
    [os のデータ, 220](#)  
    [TarInfo の属性, 419](#)  
`name2codepoint` (`htmlentitydefs` のデータ), [618](#)  
`NamedTemporaryFile()` (`tempfile` モジュール), [328](#)  
`NameError` (exceptions の例外), [39](#)  
`namelist()` (`ZipFile` のメソッド), [413](#)  
`nameprep()` (`encodings.idna` モジュール), [152](#)  
`NamespaceErr` (`xml.dom` の例外), [635](#)  
`namespaces`  
    [XML, 658](#)  
`namespaceURI` (`Node` の属性), [630](#)  
`NaN`, [8](#), [114](#)  
`NannyNag` (`tabnanny` の例外), [731](#)  
`napms()` (`curses` モジュール), [281](#)  
`National Security Agency`, [686](#)  
`ndiff()` (`difflib` モジュール), [132](#)  
`ne()` (`operator` モジュール), [66](#)  
`neg()` (`operator` モジュール), [67](#)  
`netrc`  
    [netrc のクラス, 605](#)  
    [standard module, 605](#)  
`NetrcParseError` (`netrc` の例外), [605](#)

`Network News Transfer Protocol`, [504](#)  
`new()`  
    [hmac モジュール, 681](#)  
    [md5 モジュール, 682](#)  
    [sha モジュール, 683](#)  
`new` (built-in module), [107](#)  
`new_alignment()` (`writer` のメソッド), [546](#)  
`new_font()` (`writer` のメソッド), [546](#)  
`new_margin()` (`writer` のメソッド), [546](#)  
`new_module()` (`imp` モジュール), [97](#)  
`new_panel()` (`curses.panel` モジュール), [298](#)  
`new_spacing()` (`writer` のメソッド), [546](#)  
`new_styles()` (`writer` のメソッド), [546](#)  
`newconfig()` (`al` モジュール), [751](#)  
`newgroups()` (`NNTPDataError` のメソッド), [506](#)  
`newlines` (`file` の属性), [33](#)  
`newnews()` (`NNTPDataError` のメソッド), [506](#)  
`newpad()` (`curses` モジュール), [281](#)  
`newrotor()` (`rotor` モジュール), [685](#)  
`newwin()` (`curses` モジュール), [281](#)  
`next()`  
    [のメソッド, 404](#)  
    [csv reader のメソッド, 610](#)  
    [dbhash のメソッド, 401](#)  
    [file のメソッド, 31](#)  
    [iterator のメソッド, 21](#)  
    [mailbox のメソッド, 582](#)  
    [MultiFile のメソッド, 591](#)  
    [NNTPDataError のメソッド, 507](#)  
    [TarFile のメソッド, 417](#)  
`nextfile()` (`fileinput` モジュール), [209](#)  
`nextkey()` (`gdbm` モジュール), [431](#)  
`nextpart()` (`MimeWriter` のメソッド), [589](#)  
`nextSibling` (`Node` の属性), [629](#)  
`ngettext()`  
    [gettext モジュール, 346](#)  
    [GNUTranslations のメソッド, 349](#)  
    [NullTranslations のメソッド, 348](#)  
`NI_*` (`socket` のデータ), [376](#)  
`nice()` (`os` モジュール), [235](#)  
`nis` (extension module), [442](#)  
`NIST`, [683](#)  
`NL` (`tokenize` のデータ), [731](#)  
`nl()` (`curses` モジュール), [281](#)  
`nl_langinfo()` (`locale` モジュール), [340](#)  
`nlst()` (`FTP` のメソッド), [496](#)  
`NNTP`

- protocol, [504](#)
- NNTP (nntplib のクラス), [505](#)
- NNTPDataError (nntplib のクラス), [506](#)
- NNTPError (nntplib のクラス), [505](#)
- nntplib (standard module), [504](#)
- NNTPPermanentError (nntplib のクラス), [505](#)
- NNTPProtocolError (nntplib のクラス), [506](#)
- NNTPReplyError (nntplib のクラス), [505](#)
- NNTPTemporaryError (nntplib のクラス), [505](#)
- nocbreak() (curses モジュール), [281](#)
- NoDataAllowedErr (xml.dom の例外), [636](#)
- Node (compiler.ast のクラス), [744](#)
- nodelay() (window のメソッド), [287](#)
- nodeName (Node の属性), [630](#)
- nodeType (Node の属性), [629](#)
- nodeValue (Node の属性), [630](#)
- NODISC (cd のデータ), [754](#)
- noecho() (curses モジュール), [281](#)
- NOEXPR (locale のデータ), [343](#)
- nofill (HTMLParser の属性), [617](#)
- nok\_builtin\_names (RExec の属性), [716](#)
- noload() (Unpickler のメソッド), [81](#)
- NoModificationAllowedErr (xml.dom の例外), [636](#)
- nonblock() (audio device のメソッド), [676](#)
- None
  - Built-in object, [17](#)
  - のデータ, [42](#)
- NoneType (types のデータ), [62](#)
- nonl() (curses モジュール), [281](#)
- noop()
  - IMAP4\_stream のメソッド, [502](#)
  - POP3 のメソッド, [499](#)
- NoOptionError (ConfigParser の例外), [205](#)
- noqiflush() (curses モジュール), [281](#)
- noraw() (curses モジュール), [281](#)
- normalize()
  - locale モジュール, [341](#)
  - Node のメソッド, [630](#)
  - unicodedata モジュール, [153](#)
- normalvariate() (random モジュール), [187](#)
- normcase() (os.path モジュール), [240](#)
- normpath() (os.path モジュール), [241](#)
- NoSectionError (ConfigParser の例外), [205](#)
- not
  - 演算子, [18](#)
  - operator, [18](#)

- not in
  - 演算子, [18, 22](#)
  - operator, [18, 22](#)
- not\_() (operator モジュール), [66](#)
- NotANumber (fpformat の例外), [140](#)
- notationDecl() (DTDHandler のメソッド), [649](#)
- NotationDeclHandler() (xmlparser のメソッド), [622](#)
- notations (DocumentType の属性), [631](#)
- NotConnected (httplib の例外), [491](#)
- NoteBook (Tix のクラス), [703](#)
- NotFoundErr (xml.dom の例外), [635](#)
- notify() (Condition のメソッド), [392](#)
- notifyAll() (Condition のメソッド), [392](#)
- notimeout() (window のメソッド), [287](#)
- NotImplemented (のデータ), [43](#)
- NotImplementedError (exceptions の例外), [39](#)
- NotStandaloneHandler() (xmlparser のメソッド), [623](#)
- NotSupportedErr (xml.dom の例外), [636](#)
- noutrefresh() (window のメソッド), [287](#)
- now(tz=None)() (datetime のメソッド), [256](#)
- NSA, [686](#)
- NSIG (signal のデータ), [372](#)
- NTEventLogHandler (logging のクラス), [363](#)
- ntohl() (socket モジュール), [377](#)
- ntohs() (socket モジュール), [377](#)
- ntransfercmd() (FTP のメソッド), [496](#)
- NullFormatter (formatter のクラス), [546](#)
- NullWriter (formatter のクラス), [547](#)
- numeric
  - conversions, [20](#)
  - literals, [19](#)
  - object, [18, 19](#)
  - オブジェクト, [19](#)
  - types, operations on, [19](#)
- numeric() (unicodedata モジュール), [153](#)
- Numerical Python, [14](#)
- nurbscurve() (gl モジュール), [765](#)
- nurbssurface() (gl モジュール), [765](#)
- nvarray() (gl モジュール), [765](#)

## O

- O\_APPEND (os のデータ), [226](#)
- O\_BINARY (os のデータ), [226](#)
- O\_CREAT (os のデータ), [226](#)



- O\_DSYNC (os のデータ), [226](#)
- O\_EXCL (os のデータ), [226](#)
- O\_NDELAY (os のデータ), [225](#)
- O\_NOCTTY (os のデータ), [226](#)
- O\_NONBLOCK (os のデータ), [226](#)
- O\_RDONLY (os のデータ), [225](#)
- O\_RDWR (os のデータ), [225](#)
- O\_RSYNC (os のデータ), [226](#)
- O\_SYNC (os のデータ), [226](#)
- O\_TRUNC (os のデータ), [226](#)
- O\_WRONLY (os のデータ), [225](#)
- object
  - Boolean, [19](#)
  - buffer, [21](#)
  - code, [34](#), [35](#), [92](#)
  - complex number, [19](#)
  - dictionary, [29](#)
  - file, [31](#)
  - floating point, [19](#)
  - frame, [373](#)
  - integer, [19](#)
  - list, [21](#), [28](#)
  - long integer, [19](#)
  - mapping, [29](#)
  - method, [34](#)
  - numeric, [18](#), [19](#)
  - sequence, [21](#)
  - socket, [374](#)
  - string, [21](#)
  - traceback, [47](#), [74](#)
  - tuple, [21](#)
  - type, [15](#)
  - Unicode, [21](#)
  - xrange, [21](#), [27](#)
- object() (モジュール), [11](#)
- objects
  - comparing, [18](#)
  - flattening, [77](#)
  - marshalling, [77](#)
  - persistent, [77](#)
  - pickling, [77](#)
  - serializing, [77](#)
- obufcount() (audio device のメソッド), [677](#), [770](#)
- obuf\_free() (audio device のメソッド), [677](#)
- オブジェクト
  - Boolean, [19](#)
  - buffer, [21](#)
  - code, [34](#), [35](#), [92](#)
  - complex number, [19](#)
  - dictionary, [29](#)
  - file, [31](#)
  - floating point, [19](#)
  - frame, [373](#)
  - integer, [19](#)
  - list, [21](#), [28](#)
  - long integer, [19](#)
  - mapping, [29](#)
  - method, [34](#)
  - numeric, [19](#)
  - sequence, [21](#)
  - socket, [374](#)
  - string, [21](#)
  - traceback, [47](#), [74](#)
  - tuple, [21](#)
  - type, [15](#)
  - Unicode, [21](#)
  - xrange, [21](#), [27](#)
- oct() (モジュール), [11](#)
- octal
  - literals, [19](#)
- octdigits (string のデータ), [114](#)
- offset (ExpatError の属性), [623](#)
- OK (curses のデータ), [289](#)
- ok\_builtin\_modules (RExec の属性), [716](#)
- ok\_file\_types (RExec の属性), [716](#)
- ok\_path (RExec の属性), [716](#)
- ok\_posix\_names (RExec の属性), [716](#)
- ok\_sys\_names (RExec の属性), [716](#)
- onecmd() (Cmd のメソッド), [212](#)
- open()
  - aifc モジュール, [664](#)
  - anydbm モジュール, [400](#)
  - cd モジュール, [754](#)
  - codecs モジュール, [145](#)
  - dbhash モジュール, [400](#)
  - dbm モジュール, [430](#)
  - dl モジュール, [429](#)
  - dumbdbm モジュール, [405](#)
  - gdbm モジュール, [431](#)
  - gzip モジュール, [408](#)
  - IMAP4\_stream のメソッド, [502](#)
  - OpenerDirector のメソッド, [486](#)
  - ossaudiodev モジュール, [675](#)

- os モジュール, 224
- posixfile モジュール, 438
- shelve モジュール, 89
- sunaudiodev モジュール, 769
- sunau モジュール, 666
- TarFile のメソッド, 417
- tarfile モジュール, 415
- Telnet のメソッド, 513
- Template のメソッド, 437
- URLopener のメソッド, 480
- wave モジュール, 669
- open() (webbrowser モジュール), 466, 467
- open() (モジュール), 11
- open\_new() (webbrowser モジュール), 466, 467
- open\_osfhandle() (msvcrt モジュール), 774
- open\_unknown() (URLopener のメソッド), 480
- opendir() (dircache モジュール), 242
- OpenerDirector (urllib2 のクラス), 483
- openfolder() (MH のメソッド), 583
- openfp()
  - sunau モジュール, 666
  - wave モジュール, 669
- OpenGL, 766
- OpenKey() (\_winreg モジュール), 776
- OpenKeyEx() (\_winreg モジュール), 777
- openlog() (syslog モジュール), 443
- openmessage() (Message のメソッド), 584
- openmixer() (ossaudiodev モジュール), 675
- openport() (al モジュール), 751
- openpty()
  - os モジュール, 225
  - pty モジュール, 434
- operation
  - concatenation, 22
  - extended slice, 22
  - repetition, 22
  - slice, 22
  - subscript, 22
- operations
  - bit-string, 20
  - Boolean, 17
  - masking, 20
  - shifting, 20
- operations on
  - dictionary type, 29
  - integer types, 20
  - list type, 28
  - mapping types, 29
  - mutable sequence types, 28
  - numeric types, 19
  - sequence types, 22, 28
- operator
  - ==, 18
  - and, 17, 18
  - comparison, 18
  - in, 18, 22
  - is, 18
  - is not, 18
  - not, 18
  - not in, 18, 22
  - or, 17, 18
- operator (built-in module), 66
- opname (dis のデータ), 735
- OptionMenu (Tix のクラス), 701
- options() (SafeConfigParser のメソッド), 206
- optionxform() (SafeConfigParser のメソッド), 207
- optparse (standard module), 301
- or
  - 演算子, 17, 18
  - operator, 17, 18
- or\_() (operator モジュール), 67
- ord() (モジュール), 11
- ordered\_attributes (xmlparser の属性), 620
- os
  - standard module, 220
  - 標準モジュール, 31, 425
- os.path (standard module), 239
- OSError (exceptions の例外), 39
- ossaudiodev (built-in module), 675
- output()
  - BaseCookie のメソッド, 525
  - Morsel のメソッド, 526
- output\_charset (email.Charset のデータ), 565
- output\_codec (email.Charset のデータ), 565
- OutputString() (Morsel のメソッド), 526
- OutputType (cStringIO のデータ), 141
- OverflowError (exceptions の例外), 39
- overlay() (window のメソッド), 287
- Overmars, Mark, 757
- overwrite() (window のメソッド), 287

## P

- P\_DETACH (os のデータ), 236

- P\_NOWAIT (os のデータ), 236
- P\_NOWAITO (os のデータ), 236
- P\_OVERLAY (os のデータ), 236
- P\_WAIT (os のデータ), 236
- pack() (struct モジュール), 128
- pack\_array() (Packer のメソッド), 603
- pack\_bytes() (Packer のメソッド), 603
- pack\_double() (Packer のメソッド), 602
- pack\_farray() (Packer のメソッド), 603
- pack\_float() (Packer のメソッド), 602
- pack\_fopaque() (Packer のメソッド), 602
- pack\_fstring() (Packer のメソッド), 602
- pack\_list() (Packer のメソッド), 603
- pack\_opaque() (Packer のメソッド), 603
- pack\_string() (Packer のメソッド), 603
- package, 108
- Packer (xdrlib のクラス), 602
- packing
  - binary data, 128
- packing (widgets), 694
- PAGER, 447
- pair\_content() (curses モジュール), 281
- pair\_number() (curses モジュール), 282
- PanedWindow (Tix のクラス), 702
- pardir (os のデータ), 238
- parent (BaseHandler の属性), 486
- parentNode (Node の属性), 629
- paretovariate() (random モジュール), 187
- Parse() (xmlparser のメソッド), 620
- parse()
  - cgi モジュール, 471
  - compiler モジュール, 743
  - Parser のメソッド, 557
  - RobotFileParser のメソッド, 606
  - xml.dom.minidom モジュール, 638
  - xml.dom.pulldom モジュール, 642
  - xml.sax モジュール, 643
  - XMLReader のメソッド, 652
- parse\_and\_bind() (readline モジュール), 421
- parse\_header() (cgi モジュール), 472
- parse\_multipart() (cgi モジュール), 472
- parse\_qs() (cgi モジュール), 471
- parse\_qs1() (cgi モジュール), 472
- parseaddr()
  - email.Utils モジュール, 569
  - rfc822 モジュール, 594
- parsedate()
  - email.Utils モジュール, 569
  - rfc822 モジュール, 594
- parsedate\_tz()
  - email.Utils モジュール, 569
  - rfc822 モジュール, 594
- ParseFile() (xmlparser のメソッド), 620
- parseFile() (compiler モジュール), 743
- ParseFlags() (imaplib モジュール), 500
- parseframe() (CD parser のメソッド), 757
- Parser (email.Parser のクラス), 556
- parser (built-in module), 719
- ParserCreate() (xml.parsers.expat モジュール), 619
- ParserError (parser の例外), 722
- parsesequence() (Folder のメソッド), 583
- parsestr() (Parser のメソッド), 557
- parseString()
  - xml.dom.minidom モジュール, 638
  - xml.dom.pulldom モジュール, 642
  - xml.sax モジュール, 643
- parsing
  - Python source code, 719
  - URL, 515
- ParsingError (ConfigParser の例外), 206
- partial() (IMAP4\_stream のメソッド), 502
- pass\_() (POP3 のメソッド), 498
- PATH, 233, 235, 239, 473, 475
- path
  - configuration file, 108
  - module search, 50, 76, 108
  - operations, 239
- path
  - BaseHTTPRequestHandler の属性, 520
  - os のデータ, 220
  - sys のデータ, 50
- Path browser, 707
- pathconf() (os モジュール), 228
- pathconf\_names (os のデータ), 228
- pathname2url() (urllib モジュール), 479
- pathsep (os のデータ), 239
- pattern (RegexObject の属性), 125
- pause() (signal モジュール), 373
- PAUSED (cd のデータ), 754
- Pdb (class in pdb), 445
- pdb (standard module), 445
- Pen (turtle のクラス), 707

PendingDeprecationWarning (exceptions の例外), 41

Performance, 461

persistence, 77

persistent

- objects, 77

pformat()

- pprint モジュール, 104
- PrettyPrinter のメソッド, 105

PGP, 681

pi

- cmath のデータ, 185
- math のデータ, 183

pick() (gl モジュール), 765

pickle() (copy\_reg モジュール), 88

pickle (standard module), 77

pickle (標準モジュール), 88, 89, 92

PickleError (pickle の例外), 80

Pickler (pickle のクラス), 80

pickling

- objects, 77

PicklingError (pickle の例外), 80

pid (Popen4 の属性), 249

PIL (the Python Imaging Library), 767

pipe() (os モジュール), 225

pipes (standard module), 436

PKG\_DIRECTORY (imp のデータ), 97

pkgutil (standard module), 99

platform (sys のデータ), 50

play() (CD player のメソッド), 755

playabs() (CD player のメソッド), 755

PLAYING (cd のデータ), 754

PlaySound() (winsound モジュール), 780

playtrack() (CD player のメソッド), 755

playtrackabs() (CD player のメソッド), 756

plock() (os モジュール), 235

pm() (pdb モジュール), 446

pnum (cd のデータ), 754

poll()

- のメソッド, 386
- Popen4 のメソッド, 248
- select モジュール, 384

pop()

- array のメソッド, 195
- dictionary method, 29
- fifo のメソッド, 540
- list method, 28

MultiFile のメソッド, 592

POP3

- protocol, 497

POP3 (poplib のクラス), 498

pop\_alignment() (formatter のメソッド), 545

pop\_font() (formatter のメソッド), 545

pop\_margin() (formatter のメソッド), 545

pop\_source() (shlex のメソッド), 215

pop\_style() (formatter のメソッド), 545

popen()

- in module os, 385
- os モジュール, 223

popen2()

- os モジュール, 223
- popen2 モジュール, 248

popen2 (standard module), 247

Popen3 (popen2 のクラス), 248

popen3()

- os モジュール, 223
- popen2 モジュール, 248

Popen4 (popen2 のクラス), 248

popen4()

- os モジュール, 223
- popen2 モジュール, 248

popitem() (dictionary method), 29

poplib (standard module), 497

PopupMenu (Tix のクラス), 701

PortableUnixMailbox (mailbox のクラス), 580

pos() (operator モジュール), 67

pos (MatchObject の属性), 126

POSIX

- file object, 437
- I/O control, 432, 433
- threads, 386

posix (built-in module), 425

posix=True (TarFile の属性), 418

posixfile (built-in module), 437

post()

- audio device のメソッド, 677
- NNTPDataError のメソッド, 508

post\_mortem() (pdb モジュール), 446

postcmd() (Cmd のメソッド), 213

postloop() (Cmd のメソッド), 213

pow()

- math モジュール, 183
- operator モジュール, 67

モジュール, 11  
 powm() (mpz モジュール), 684  
 pprint()  
     pprint モジュール, 104  
     PrettyPrinter のメソッド, 105  
 pprint (standard module), 103  
 prcal() (calendar モジュール), 211  
 preamble (email.Message のデータ), 554  
 precmd() (Cmd のメソッド), 213  
 prefix  
     Attr の属性, 634  
     Node の属性, 630  
     sys のデータ, 50  
 preloop() (Cmd のメソッド), 213  
 preorder() (ASTVisitor のメソッド), 750  
 prepare\_input\_source() (xml.sax.saxutils  
     モジュール), 650  
 prepend() (Template のメソッド), 437  
 Pretty Good Privacy, 681  
 PrettyPrinter (pprint のクラス), 104  
 preventremoval() (CD player のメソッド),  
     756  
 previous()  
     のメソッド, 404  
     dbhash のメソッド, 401  
 previousSibling (Node の属性), 629  
 print  
     実行文, 17  
     statement, 17  
 print\_callees() (Stats のメソッド), 457  
 print\_callers() (Stats のメソッド), 457  
 print\_directory() (cgi モジュール), 472  
 print\_environ() (cgi モジュール), 472  
 print\_environ\_usage() (cgi モジュール),  
     472  
 print\_exc()  
     Timer のメソッド, 461  
     traceback モジュール, 75  
 print\_exception() (traceback モジュール),  
     74  
 print\_form() (cgi モジュール), 472  
 print\_last() (traceback モジュール), 75  
 print\_stack() (traceback モジュール), 75  
 print\_stats() (Stats のメソッド), 457  
 print\_tb() (traceback モジュール), 74  
 printable (string のデータ), 114  
 printdir() (ZipFile のメソッド), 413  
 printf-style formatting, 25  
 Priority Queue, 190  
 prmonth() (calendar モジュール), 211  
 process  
     group, 221  
     id, 221  
     id of parent, 221  
     killing, 234  
     signalling, 234  
 process\_request() (SocketServer モジュー  
     ル), 518  
 processes, light-weight, 386  
 processingInstruction() (ContentHandler  
     のメソッド), 648  
 ProcessingInstructionHandler() (xml-  
     parser のメソッド), 622  
 processor time, 270  
 Profile (hotshot のクラス), 459  
 profile (standard module), 454  
 profile function, 389  
 profiler, 51  
 profiling, deterministic, 451  
 prompt (Cmd の属性), 213  
 prompt\_user\_passwd() (FancyURLopener の  
     メソッド), 481  
 prompts, interpreter, 50  
 propagate (logging のデータ), 357  
 property() (モジュール), 11  
 property\_declaration\_handler  
     (xml.sax.handler のデータ), 646  
 property\_dom\_node (xml.sax.handler のデー  
     タ), 646  
 property\_lexical\_handler  
     (xml.sax.handler のデータ), 646  
 property\_xml\_string (xml.sax.handler の  
     データ), 646  
 protocol  
     CGI, 467  
     FTP, 480, 493  
     Gopher, 480, 497  
     HTTP, 467, 480, 490, 519  
     IMAP4, 499  
     IMAP4\_SSL, 499  
     IMAP4\_stream, 499  
     iterator, 20  
     NNTP, 504  
     POP3, 497

SMTP, 508  
Telnet, 512  
PROTOCOL\_VERSION (IMAP4\_stream の属性), 504  
protocol\_version (BaseHTTPRequestHandler の属性), 520  
proxy() (weakref モジュール), 55  
proxyauth() (IMAP4\_stream のメソッド), 502  
ProxyBasicAuthHandler (urllib2 のクラス), 484  
ProxyDigestAuthHandler (urllib2 のクラス), 484  
ProxyHandler (urllib2 のクラス), 483  
ProxyType (weakref のデータ), 56  
ProxyTypes (weakref のデータ), 56  
prstr() (fm モジュール), 763  
ps1 (sys のデータ), 50  
ps2 (sys のデータ), 50  
pstats (standard module), 455  
pthreads, 386  
ptime (cd のデータ), 755  
pty  
    standard module, 434  
    標準モジュール, 225  
publicId (DocumentType の属性), 631  
PullDOM (xml.dom.pulldom のクラス), 642  
punctuation (string のデータ), 114  
push()  
    async\_chat のメソッド, 539  
    fifo のメソッド, 540  
    InteractiveConsole のメソッド, 102  
    MultiFile のメソッド, 591  
push\_alignment() (formatter のメソッド), 545  
push\_font() (formatter のメソッド), 545  
push\_margin() (formatter のメソッド), 545  
push\_source() (shlex のメソッド), 215  
push\_style() (formatter のメソッド), 545  
push\_token() (shlex のメソッド), 215  
push\_with\_producer() (async\_chat のメソッド), 539  
put() (Queue のメソッド), 397  
put\_nowait() (Queue のメソッド), 397  
putch() (msvcrt モジュール), 774  
putenv() (os モジュール), 221  
putheader() (HTTPSConnection のメソッド), 492  
putp() (curses モジュール), 282

putrequest() (HTTPSConnection のメソッド), 492  
putsequences() (Folder のメソッド), 584  
putwin() (window のメソッド), 287  
pwd() (FTP のメソッド), 496  
pwd  
    built-in module, 426  
    組み込みモジュール, 239  
pwlcurve() (gl モジュール), 765  
py\_compile (standard module), 733  
PY\_COMPILED (imp のデータ), 97  
PY\_FROZEN (imp のデータ), 98  
PY\_RESOURCE (imp のデータ), 97  
PY\_SOURCE (imp のデータ), 97  
pyclbr (standard module), 732  
PyCompileError (py\_compile の例外), 733  
pydoc (standard module), 157  
pyexpat (組み込みモジュール), 618  
PyOpenGL, 766  
Python Editor, 707  
Python Enhancement Proposals  
    PEP 0205, 56  
    PEP 236, 5  
    PEP 282, 357  
    PEP 305, 607  
Python Imaging Library, 767  
PYTHON\_DOM, 627  
PYTHONPATH, 50, 473, 784  
PYTHONSTARTUP, 109, 422, 423  
PYTHONY2K, 269, 270  
PyZipFile (zipfile のクラス), 411

## Q

qdevice() (fl モジュール), 759  
qenter() (fl モジュール), 759  
qiflush() (curses モジュール), 282  
qread() (fl モジュール), 759  
qreset() (fl モジュール), 759  
qsize() (Queue のメソッド), 397  
qtest() (fl モジュール), 759  
QueryInfoKey() (\_winreg モジュール), 777  
queryparams() (al モジュール), 752  
QueryValue() (\_winreg モジュール), 777  
QueryValueEx() (\_winreg モジュール), 777  
Queue  
    Queue のクラス, 396  
    standard module, 396



`quick_ratio()` (SequenceMatcher のメソッド),  
136

`quit()`

FTP のメソッド, 497

NNTPDataError のメソッド, 508

POP3 のメソッド, 499

SMTP のメソッド, 511

`quopri` (standard module), 600

`quote()`

email.Utils モジュール, 569

rfc822 モジュール, 593

urllib モジュール, 478

`QUOTE_ALL` (csv のデータ), 609

`QUOTE_MINIMAL` (csv のデータ), 609

`QUOTE_NONE` (csv のデータ), 609

`QUOTE_NONNUMERIC` (csv のデータ), 609

`quote_plus()` (urllib モジュール), 478

`quoteattr()` (xml.sax.saxutils モジュール), 650

`quotechar` (Dialect の属性), 610

`quoted-printable`

encoding, 600

`quotes` (shlex の属性), 216

`quoting` (Dialect の属性), 610

## R

`r_eval()` (RExec のメソッド), 715

`r_exec()` (RExec のメソッド), 715

`r_execfile()` (RExec のメソッド), 715

`r_import()` (RExec のメソッド), 715

`R_OK` (os のデータ), 226

`r_open()` (RExec のメソッド), 715

`r_reload()` (RExec のメソッド), 716

`r_unload()` (RExec のメソッド), 716

`radians()`

math モジュール, 183

turtle モジュール, 705

`RADIXCHAR` (locale のデータ), 343

`raise`

実行文, 37

statement, 37

`randint()`

random モジュール, 186

whrandom モジュール, 188

`random()`

random モジュール, 187

whrandom モジュール, 188

`random` (standard module), 185

`randrange()` (random モジュール), 186

`range()` (モジュール), 12

`Rat` (demo module), 684

`ratecv()` (audioop モジュール), 661

`ratio()` (SequenceMatcher のメソッド), 136

rational numbers, 684

`raw()` (curses モジュール), 282

`raw_input()`

InteractiveConsole のメソッド, 102

モジュール, 12

組み込み関数, 51

`RawConfigParser` (ConfigParser のクラス), 205

`RawPen` (turtle のクラス), 707

`re`

MatchObject の属性, 127

standard module, 117

標準モジュール, 27, 113, 336

`read()`

のメソッド, 382, 399

array のメソッド, 195

audio device のメソッド, 676, 770

BZ2File のメソッド, 409

Chunk のメソッド, 672

file のメソッド, 31

HTTPConnection のメソッド, 492

IMAP4\_stream のメソッド, 502

imgfile モジュール, 767

MimeTypes のメソッド, 588

MultiFile のメソッド, 591

os モジュール, 225

RobotFileParser のメソッド, 606

SafeConfigParser のメソッド, 206

StreamReader のメソッド, 148

ZipFile のメソッド, 413

`read_all()` (Telnet のメソッド), 513

`read_byte()` (のメソッド), 399

`read_eager()` (Telnet のメソッド), 513

`read_history_file()` (readline モジュール),  
421

`read_init_file()` (readline モジュール), 421

`read_lazy()` (Telnet のメソッド), 513

`read_mime_types()` (mimetypes モジュール),  
586

`read_sb_data()` (Telnet のメソッド), 513

`read_some()` (Telnet のメソッド), 513

`read_token()` (shlex のメソッド), 215

`read_until()` (Telnet のメソッド), 513

`read_very_eager()` (Telnet のメソッド), 513  
`read_very_lazy()` (Telnet のメソッド), 513  
`readable()`  
     `async_chat` のメソッド, 539  
     `dispatcher` のメソッド, 536  
`readadda()` (CD player のメソッド), 756  
`reader()` (csv モジュール), 607  
`ReadError` (tarfile の例外), 416  
`readfp()`  
     `MimeTypes` のメソッド, 588  
     `SafeConfigParser` のメソッド, 207  
`readframes()`  
     `aifc` のメソッド, 664  
     `AU_read` のメソッド, 667  
     `Wave_read` のメソッド, 669  
`readline()`  
     のメソッド, 399  
     `BZ2File` のメソッド, 410  
     `file` のメソッド, 32  
     `IMAP4_stream` のメソッド, 502  
     `MultiFile` のメソッド, 591  
     `StreamReader` のメソッド, 148  
`readline (built-in module)`, 421  
`readlines()`  
     `BZ2File` のメソッド, 410  
     `file` のメソッド, 32  
     `MultiFile` のメソッド, 591  
     `StreamReader` のメソッド, 148  
`readlink()` (os モジュール), 228  
`readmodule()` (pyclbr モジュール), 732  
`readmodule_ex()` (pyclbr モジュール), 732  
`readsamps()` (audio port のメソッド), 752  
`readscaled()` (imgfile モジュール), 767  
`READY` (cd のデータ), 754  
`Real Media File Format`, 671  
`real_quick_ratio()` (SequenceMatcher のメソッド), 136  
`realpath()` (os.path モジュール), 241  
`reason` (httplib のデータ), 492  
`recontrols()` (mixer device のメソッド), 678  
`recent()` (IMAP4\_stream のメソッド), 502  
`rectangle()` (curses.textpad モジュール), 293  
`recv()`  
     `dispatcher` のメソッド, 536  
     `socket` のメソッド, 380  
`recvfrom()` (socket のメソッド), 380  
`redirect`, 477  
`redirect_request()` (HTTPRedirectHandler のメソッド), 487  
`redraw_form()` (form のメソッド), 759  
`redraw_object()` (FORMS object のメソッド), 762  
`redrawln()` (window のメソッド), 287  
`redrawwin()` (window のメソッド), 287  
`reduce()` (モジュール), 12  
`ref()` (weakref モジュール), 55  
`ReferenceError`  
     exceptions の例外, 39  
     weakref の例外, 56  
`ReferenceType` (weakref のデータ), 56  
`refilemessages()` (Folder のメソッド), 584  
`refill_buffer()` (async\_chat のメソッド), 539  
`refresh()` (window のメソッド), 287  
`register()`  
     のメソッド, 385  
     `atexit` モジュール, 60  
     `codecs` モジュール, 143  
     `webbrowser` モジュール, 466  
`register_dialect()` (csv モジュール), 608  
`register_error()` (codecs モジュール), 144  
`register_function()` (SimpleXMLRPCRequestHandler のメソッド), 532  
`register_function()` (SimpleXMLRPCServer のメソッド), 532  
`register_instance()` (SimpleXMLRPCRequestHandler のメソッド), 533  
`register_instance()` (SimpleXMLRPCServer のメソッド), 532  
`register_introspection_functions()` (SimpleXMLRPCRequestHandler のメソッド), 532, 533  
`register_multicall_functions()` (SimpleXMLRPCRequestHandler のメソッド), 532, 533  
`registerDOMImplementation()` (xml.dom モジュール), 627  
`RegLoadKey()` (\_winreg モジュール), 776  
`relative`  
     URL, 515  
`release()`  
     `Condition` のメソッド, 391  
     `Semaphore` のメソッド, 392  
`release()` (Timer のメソッド), 389, 390

`release()`  
     のメソッド, 359  
     lock のメソッド, 387  
`release_lock()` (imp モジュール), 97  
`reload()`  
     モジュール, 12  
     組み込み関数, 49, 97, 99  
`remove()`  
     array のメソッド, 195  
     list method, 28  
     os モジュール, 228  
`remove_option()`  
     のメソッド, 317  
     SafeConfigParser のメソッド, 207  
`remove_section()` (SafeConfigParser のメソッド), 207  
`removeAttribute()` (Element のメソッド), 633  
`removeAttributeNode()` (Element のメソッド), 633  
`removeAttributeNS()` (Element のメソッド), 633  
`removecallback()` (CD parser のメソッド), 757  
`removeChild()` (Node のメソッド), 630  
`removedirs()` (os モジュール), 228  
`removeFilter()` ( のメソッド), 358, 359  
`removeHandler()` ( のメソッド), 359  
`removemessages()` (Folder のメソッド), 584  
`rename()`  
     FTP のメソッド, 496  
     IMAP4\_stream のメソッド, 502  
     os モジュール, 229  
`renames()` (os モジュール), 229  
`reorganize()` (gdbm モジュール), 431  
`repeat()`  
     itertools モジュール, 202  
     operator モジュール, 68  
     Timer のメソッド, 462  
repetition  
     operation, 22  
`replace()`  
     のメソッド, 299  
     date のメソッド, 254  
     datetime のメソッド, 259  
     string のメソッド, 24  
     string モジュール, 116  
     time のメソッド, 262  
`replace_errors()` (codecs モジュール), 144  
`replace_header()` (Message のメソッド), 551  
`replace_whitespace` (TextWrapper の属性), 142  
`replaceChild()` (Node のメソッド), 630  
`report()` (dircmp のメソッド), 246  
`report_full_closure()` (dircmp のメソッド), 247  
`report_partial_closure()` (dircmp のメソッド), 247  
`report_unbalanced()` (SGMLParser のメソッド), 615  
`Repr` (repr のクラス), 106  
`repr()`  
     Repr のメソッド, 107  
     repr モジュール, 106  
     モジュール, 13  
`repr` (standard module), 106  
`repr1()` (Repr のメソッド), 107  
`Request` (urllib2 のクラス), 483  
`request()` (HTTPSConnection のメソッド), 492  
`request_queue_size` (SocketServer のデータ), 518  
`request_version` (BaseHTTPRequestHandler の属性), 520  
`RequestHandlerClass` (SocketServer のデータ), 517  
`requires()` (test.test\_support モジュール), 181  
`reserved` (ZipInfo の属性), 414  
`reset()`  
     audio device のメソッド, 677  
     DOMEventStream のメソッド, 643  
     HTMLParser のメソッド, 612  
     IncrementalParser のメソッド, 653  
     Packer のメソッド, 602  
     SGMLParser のメソッド, 614  
     statcache モジュール, 245  
     StreamReader のメソッド, 148  
     StreamWriter のメソッド, 147  
     Template のメソッド, 437  
     turtle モジュール, 705  
     Unpacker のメソッド, 603  
     XMLParser のメソッド, 655  
`reset_prog_mode()` (curses モジュール), 282  
`reset_shell_mode()` (curses モジュール), 282  
`resetbuffer()` (InteractiveConsole のメソッド)

ド), 102

resetlocale() (locale モジュール), 341

resetparser() (CD parser のメソッド), 757

resetwarnings() (warnings モジュール), 96

resize() ( のメソッド), 399

resolution

- date の属性, 254
- datetime の属性, 257
- time の属性, 261
- timedelta の属性, 252

resolveEntity() (EntityResolver のメソッド), 649

resource (built-in module), 439

ResourceDenied (test.test\_support の例外), 181

response() (IMAP4\_stream のメソッド), 502

ResponseNotReady (httplib の例外), 491

responses (BaseHTTPRequestHandler の属性), 520

restore() (difflib モジュール), 133

retr() (POP3 のメソッド), 498

retrbinary() (FTP のメソッド), 495

retrieve() (URLopener のメソッド), 481

retrlines() (FTP のメソッド), 495

returns\_unicode (xmlparser の属性), 621

reverse()

- array のメソッド, 195
- audioop モジュール, 661
- list method, 28

reverse\_order() (Stats のメソッド), 456

rewind()

- aifc のメソッド, 664
- AU\_read のメソッド, 667
- Wave\_read のメソッド, 670

rewindbody() (AddressList のメソッド), 594

RExec (rexec のクラス), 714

rexec

- standard module, 714
- 標準モジュール, 3

RFC

- RFC 1014, 602
- RFC 1321, 682
- RFC 1521, 597, 600, 601
- RFC 1522, 600
- RFC 1524, 579
- RFC 1725, 497
- RFC 1730, 499
- RFC 1738, 516
- RFC 1766, 340, 341
- RFC 1808, 516
- RFC 1832, 602
- RFC 1866, 616, 617
- RFC 1869, 508, 509
- RFC 1894, 574
- RFC 2045, 547, 552, 553, 562
- RFC 2046, 562
- RFC 2047, 547, 562, 563
- RFC 2060, 499
- RFC 2068, 524
- RFC 2087, 501–503
- RFC 2104, 681
- RFC 2109, 524, 525
- RFC 2231, 547, 552, 553, 562, 570
- RFC 2396, 516
- RFC 2553, 374
- RFC 2616, 479, 487
- RFC 2821, 547
- RFC 2822, 272, 547, 548, 550, 557–559, 562, 563, 569, 570, 593–595
- RFC 2833, 593
- RFC 3454, 154
- RFC 3490, 152, 153
- RFC 3492, 152
- RFC 821, 508, 509, 783
- RFC 822, 205, 272, 349, 492, 510, 511, 562, 593
- RFC 854, 512, 513
- RFC 959, 493
- RFC 977, 504

rfc822

- standard module, 593
- 標準モジュール, 584

rfile (BaseHTTPRequestHandler の属性), 520

rfind()

- string のメソッド, 24
- string モジュール, 115

rgb\_to\_hls() (colorsys モジュール), 672

rgb\_to\_hsv() (colorsys モジュール), 672

rgb\_to\_yiq() (colorsys モジュール), 672

rgbimg (built-in module), 673

right() (turtle モジュール), 706

right\_list (dircmp の属性), 247

right\_only (dircmp の属性), 247

rindex()

- string のメソッド, 24

- string モジュール, 115
- rjust()
  - string のメソッド, 24
  - string モジュール, 116
- rlcompleter (standard module), 423
- rlencode\_hqx() (binascii モジュール), 598
- rldecode\_hqx() (binascii モジュール), 598
- RLIMIT\_AS (resource のデータ), 441
- RLIMIT\_CORE (resource のデータ), 440
- RLIMIT\_CPU (resource のデータ), 440
- RLIMIT\_DATA (resource のデータ), 441
- RLIMIT\_FSIZE (resource のデータ), 441
- RLIMIT\_MEMLOCK (resource のデータ), 441
- RLIMIT\_NOFILE (resource のデータ), 441
- RLIMIT\_NPROC (resource のデータ), 441
- RLIMIT\_OFILE (resource のデータ), 441
- RLIMIT\_RSS (resource のデータ), 441
- RLIMIT\_STACK (resource のデータ), 441
- RLIMIT\_VMEM (resource のデータ), 441
- RLock() (threading モジュール), 388
- rmd() (FTP のメソッド), 497
- rmdir() (os モジュール), 229
- RMFF, 671
- rms() (audioop モジュール), 661
- rmtree() (shutil モジュール), 338
- rnopen() (bsddb モジュール), 403
- RobotFileParser (robotparser のクラス), 606
- robotparser (standard module), 606
- robots.txt, 606
- RotatingFileHandler (logging のクラス), 361
- rotor (built-in module), 685
- round() (モジュール), 13
- rpop() (POP3 のメソッド), 498
- rset() (POP3 のメソッド), 499
- rshift() (operator モジュール), 68
- rstrip()
  - string のメソッド, 24
  - string モジュール, 116
- RTLD\_LAZY (dl のデータ), 429
- RTLD\_NOW (dl のデータ), 429
- ruler (Cmd の属性), 214
- run()
  - pdb モジュール, 446
  - Profile のメソッド, 460
  - profile モジュール, 454
  - scheduler のメソッド, 275

- TestCase のメソッド, 174
- TestSuite のメソッド, 176
- Thread のメソッド, 394
- Run script, 708
- run\_suite() (test.test\_support モジュール), 181
- run\_unittest() (test.test\_support モジュール), 181
- runcall()
  - pdb モジュール, 446
  - Profile のメソッド, 460
- runcode() (InteractiveConsole のメソッド), 101
- runctx() (Profile のメソッド), 460
- runeval() (pdb モジュール), 446
- runsource() (InteractiveConsole のメソッド), 101
- RuntimeError (exceptions の例外), 39
- RuntimeWarning (exceptions の例外), 41
- RUSAGE\_BOTH (resource のデータ), 442
- RUSAGE\_CHILDREN (resource のデータ), 442
- RUSAGE\_SELF (resource のデータ), 442

## S

- S (re のデータ), 122
- s\_eval() (RExec のメソッド), 715
- s\_exec() (RExec のメソッド), 715
- s\_execfile() (RExec のメソッド), 715
- S\_IFMT() (stat モジュール), 243
- S\_IMODE() (stat モジュール), 243
- s\_import() (RExec のメソッド), 716
- S\_ISBLK() (stat モジュール), 243
- S\_ISCHR() (stat モジュール), 243
- S\_ISDIR() (stat モジュール), 243
- S\_ISFIFO() (stat モジュール), 243
- S\_ISLNK() (stat モジュール), 243
- S\_ISREG() (stat モジュール), 243
- S\_ISSOCK() (stat モジュール), 243
- s\_reload() (RExec のメソッド), 716
- s\_unload() (RExec のメソッド), 716
- SafeConfigParser (ConfigParser のクラス), 205
- saferepr() (pprint モジュール), 105
- same\_files (dircmp の属性), 247
- samefile() (os.path モジュール), 241
- sameopenfile() (os.path モジュール), 241
- samestat() (os.path モジュール), 241
- sample() (random モジュール), 186
- save\_bgn() (HTMLParser のメソッド), 618

save\_end() (HTMLParser のメソッド), 618  
 SaveKey() (\_winreg モジュール), 777  
 SAX2DOM (xml.dom.pulldom のクラス), 642  
 SAXException (xml.sax の例外), 644  
 SAXNotRecognizedException (xml.sax の例外), 644  
 SAXNotSupportedException (xml.sax の例外), 644  
 SAXParseException (xml.sax の例外), 644  
 scale() (imageop モジュール), 663  
 scalefont() (fm モジュール), 763  
 scanf() (in module re), 127  
 sched (standard module), 274  
 scheduler (sched のクラス), 275  
 sci() (fpformat モジュール), 139  
 scroll() (window のメソッド), 288  
 ScrolledText (standard module), 705  
 scrollok() (window のメソッド), 288  
 search  
     path, module, 50, 76, 108  
 search()  
     IMAP4\_stream のメソッド, 503  
     RegexObject のメソッド, 125  
     re モジュール, 123  
 SEARCH\_ERROR (imp のデータ), 98  
 second  
     datetime の属性, 257  
     time の属性, 262  
 section\_divider() (MultiFile のメソッド), 592  
 sections() (SafeConfigParser のメソッド), 206  
 Secure Hash Algorithm, 683  
 security  
     CGI, 473  
 seed()  
     random モジュール, 186  
     whrandom のメソッド, 188  
     whrandom モジュール, 188  
 seek()  
     のメソッド, 399  
     BZ2File のメソッド, 410  
     CD player のメソッド, 756  
     Chunk のメソッド, 672  
     file のメソッド, 32  
     MultiFile のメソッド, 591  
 SEEK\_CUR (posixfile のデータ), 438  
 SEEK\_END (posixfile のデータ), 438  
 SEEK\_SET (posixfile のデータ), 438  
 seekblock() (CD player のメソッド), 756  
 seektrack() (CD player のメソッド), 756  
 Select (Tix のクラス), 701  
 select()  
     gl モジュール, 765  
     IMAP4\_stream のメソッド, 503  
     select モジュール, 384  
 select (built-in module), 384  
 Semaphore() (threading モジュール), 388  
 Semaphore (threading のクラス), 392  
 semaphores, binary, 386  
 send()  
     DatagramHandler のメソッド, 362  
     dispatcher のメソッド, 536  
     HTTPConnection のメソッド, 492  
     IMAP4\_stream のメソッド, 503  
     socket のメソッド, 380  
     SocketHandler のメソッド, 362  
 send\_error() (BaseHTTPRequestHandler のメソッド), 521  
 send\_flowling\_data() (writer のメソッド), 547  
 send\_header() (BaseHTTPRequestHandler のメソッド), 521  
 send\_hor\_rule() (writer のメソッド), 546  
 send\_label\_data() (writer のメソッド), 547  
 send\_line\_break() (writer のメソッド), 546  
 send\_literal\_data() (writer のメソッド), 547  
 send\_paragraph() (writer のメソッド), 546  
 send\_query() (gopherlib モジュール), 497  
 send\_response() (BaseHTTPRequestHandler のメソッド), 521  
 send\_selector() (gopherlib モジュール), 497  
 sendall() (socket のメソッド), 380  
 sendcmd() (FTP のメソッド), 495  
 sendmail() (SMTP のメソッド), 510  
 sendto() (socket のメソッド), 380  
 sep (os のデータ), 238  
 sequence  
     iteration, 20  
     object, 21  
     オブジェクト, 21  
     types, mutable, 28  
     types, operations on, 22, 28  
     types, operations on mutable, 28



sequence2ast() (parser モジュール), 720  
sequenceIncludes() (operator モジュール), 68  
SequenceMatcher (difflib のクラス), 131, 134  
SerialCookie (Cookie のクラス), 524  
serializing  
    objects, 77  
serve\_forever() (SocketServer モジュール), 517  
server  
    WWW, 467, 519  
server\_activate() (SocketServer モジュール), 518  
server\_address (SocketServer のデータ), 517  
server\_bind() (SocketServer モジュール), 518  
server\_version (BaseHTTPRequestHandler の属性), 520  
server\_version (SimpleHTTPRequestHandler の属性), 522  
ServerProxy (xmlrpclib のクラス), 528  
Set (sets のクラス), 196  
set()  
    Event のメソッド, 393  
    mixer device のメソッド, 678  
    Morsel のメソッド, 525  
    SafeConfigParser のメソッド, 207  
set\_boundary() (Message のメソッド), 554  
set\_call\_back() (FORMS object のメソッド), 761  
set\_charset() (Message のメソッド), 549  
set\_completer() (readline モジュール), 422  
set\_completer\_delims() (readline モジュール), 422  
set\_debug() (gc モジュール), 53  
set\_debuglevel()  
    FTP のメソッド, 494  
    HTTPSConnection のメソッド, 492  
    NNTPDataError のメソッド, 506  
    POP3 のメソッド, 498  
    SMTP のメソッド, 509  
    Telnet のメソッド, 514  
set\_default\_type() (Message のメソッド), 552  
set\_event\_call\_back() (fl モジュール), 758  
set\_form\_position() (form のメソッド), 759  
set\_graphics\_mode() (fl モジュール), 758  
set\_history\_length() (readline モジュール), 421  
set\_location() ( のメソッド), 403  
set\_option\_negotiation\_callback() (Telnet のメソッド), 514  
set\_param() (Message のメソッド), 553  
set\_pasv() (FTP のメソッド), 495  
set\_payload() (Message のメソッド), 549  
set\_position() (Unpacker のメソッド), 603  
set\_pre\_input\_hook() (readline モジュール), 422  
set\_proxy() (Request のメソッド), 485  
set\_recsrc() (mixer device のメソッド), 679  
set\_seq1() (SequenceMatcher のメソッド), 134  
set\_seq2() (SequenceMatcher のメソッド), 134  
set\_seqs() (SequenceMatcher のメソッド), 134  
set\_server\_documentation() (DocXMLRPCRequestHandler のメソッド), 534  
set\_server\_name() (DocXMLRPCRequestHandler のメソッド), 534  
set\_server\_title() (DocXMLRPCRequestHandler のメソッド), 534  
set\_spacing() (formatter のメソッド), 545  
set\_startup\_hook() (readline モジュール), 421  
set\_terminator() (async\_chat のメソッド), 539  
set\_threshold() (gc モジュール), 53  
set\_trace() (pdb モジュール), 446  
set\_type() (Message のメソッド), 553  
set\_unixfrom() (Message のメソッド), 549  
set\_url() (RobotFileParser のメソッド), 606  
set\_userptr() ( のメソッド), 299  
setacl() (IMAP4\_stream のメソッド), 503  
setattr() (モジュール), 13  
setAttribute() (Element のメソッド), 633  
setAttributeNode() (Element のメソッド), 633  
setAttributeNodeNS() (Element のメソッド), 633  
setAttributeNS() (Element のメソッド), 633  
SetBase() (xmlparser のメソッド), 620  
setblocking() (socket のメソッド), 381  
setByteStream() (InputSource のメソッド), 654  
setcbreak() (tty モジュール), 433  
setchannels() (audio configuration のメソッド), 752

[setCharacterStream\(\)](#) (InputStream のメソッド), [654](#)  
[setcheckinterval\(\)](#) (sys モジュール), [50](#)  
[setcomptype\(\)](#)  
     [aifc のメソッド](#), [665](#)  
     [AU\\_write のメソッド](#), [668](#)  
     [Wave\\_write のメソッド](#), [670](#)  
[setconfig\(\)](#) (audio port のメソッド), [753](#)  
[setContentHandler\(\)](#) (XMLReader のメソッド), [652](#)  
[setcontext\(\)](#) (MH のメソッド), [583](#)  
[setcurrent\(\)](#) (Folder のメソッド), [583](#)  
[setDaemon\(\)](#) (Thread のメソッド), [395](#)  
[setdefault\(\)](#) (dictionary method), [29](#)  
[setdefaultencoding\(\)](#) (sys モジュール), [50](#)  
[setdefaulttimeout\(\)](#) (socket モジュール), [379](#)  
[setdlopenflags\(\)](#) (sys モジュール), [50](#)  
[setDocumentLocator\(\)](#) (ContentHandler のメソッド), [646](#)  
[setDTDHandler\(\)](#) (XMLReader のメソッド), [652](#)  
[setegid\(\)](#) (os モジュール), [222](#)  
[setEncoding\(\)](#) (InputStream のメソッド), [654](#)  
[setEntityResolver\(\)](#) (XMLReader のメソッド), [652](#)  
[setErrorHandler\(\)](#) (XMLReader のメソッド), [652](#)  
[seteuid\(\)](#) (os モジュール), [222](#)  
[setFeature\(\)](#) (XMLReader のメソッド), [652](#)  
[setfillpoint\(\)](#) (audio port のメソッド), [753](#)  
[setfirstweekday\(\)](#) (calendar モジュール), [210](#)  
[setfloatmax\(\)](#) (audio configuration のメソッド), [752](#)  
[setfmt\(\)](#) (audio device のメソッド), [677](#)  
[setfont\(\)](#) (fm モジュール), [764](#)  
[setFormatter\(\)](#) ( のメソッド), [359](#)  
[setframerate\(\)](#)  
     [aifc のメソッド](#), [665](#)  
     [AU\\_write のメソッド](#), [668](#)  
     [Wave\\_write のメソッド](#), [670](#)  
[setgid\(\)](#) (os モジュール), [222](#)  
[setgroups\(\)](#) (os モジュール), [222](#)  
[setinfo\(\)](#) (audio device のメソッド), [770](#)  
[setitem\(\)](#) (operator モジュール), [69](#)  
[setkey\(\)](#) (rotor のメソッド), [685](#)  
[setlast\(\)](#) (Folder のメソッド), [583](#)  
[setLevel\(\)](#) ( のメソッド), [358](#), [359](#)  
[setliteral\(\)](#)  
     SGMLParser のメソッド, [614](#)  
     XMLParser のメソッド, [656](#)  
[setLocale\(\)](#) (XMLReader のメソッド), [652](#)  
[setlocale\(\)](#) (locale モジュール), [339](#)  
[setLoggerClass\(\)](#) (logging モジュール), [357](#)  
[setlogmask\(\)](#) (syslog モジュール), [443](#)  
[setmark\(\)](#) (aifc のメソッド), [665](#)  
[setMaxConns\(\)](#) (CacheFTPHandler のメソッド), [489](#)  
[setmode\(\)](#) (msvcrt モジュール), [773](#)  
[setName\(\)](#) (Thread のメソッド), [395](#)  
[setnchannels\(\)](#)  
     [aifc のメソッド](#), [665](#)  
     [AU\\_write のメソッド](#), [668](#)  
     [Wave\\_write のメソッド](#), [670](#)  
[setnframes\(\)](#)  
     [aifc のメソッド](#), [665](#)  
     [AU\\_write のメソッド](#), [668](#)  
     [Wave\\_write のメソッド](#), [670](#)  
[setnomoretags\(\)](#)  
     SGMLParser のメソッド, [614](#)  
     XMLParser のメソッド, [655](#)  
[setoption\(\)](#) (jpeg モジュール), [768](#)  
[setparameters\(\)](#) (audio device のメソッド), [677](#)  
[setparams\(\)](#)  
     [aifc のメソッド](#), [665](#)  
     al モジュール, [752](#)  
     [AU\\_write のメソッド](#), [668](#)  
     [Wave\\_write のメソッド](#), [670](#)  
[setpath\(\)](#) (fm モジュール), [763](#)  
[setpgid\(\)](#) (os モジュール), [222](#)  
[setpgrp\(\)](#) (os モジュール), [222](#)  
[setpos\(\)](#)  
     [aifc のメソッド](#), [664](#)  
     [AU\\_read のメソッド](#), [667](#)  
     [Wave\\_read のメソッド](#), [670](#)  
[setprofile\(\)](#)  
     sys モジュール, [50](#)  
     threading モジュール, [389](#)  
[setProperty\(\)](#) (XMLReader のメソッド), [653](#)  
[setPublicId\(\)](#) (InputStream のメソッド), [653](#)  
[setqueuesize\(\)](#) (audio configuration のメソッド), [752](#)

- setquota() (IMAP4\_stream のメソッド), 503
- setraw() (tty モジュール), 433
- setrecursionlimit() (sys モジュール), 51
- setregid() (os モジュール), 222
- setreuid() (os モジュール), 222
- setrlimit() (resource モジュール), 440
- sets (standard module), 196
- setsampfmt() (audio configuration のメソッド), 752
- setsampwidth()
  - aifc のメソッド, 665
  - AU\_write のメソッド, 668
  - Wave\_write のメソッド, 670
- setscrreg() (window のメソッド), 288
- setsid() (os モジュール), 222
- setslice() (operator モジュール), 69
- setsockopt() (socket のメソッド), 381
- setstate() (random モジュール), 186
- setSystemId() (InputSource のメソッド), 653
- setsysx() (curses モジュール), 282
- setTarget() (MemoryHandler のメソッド), 364
- setTimeout() (CacheFTPHandler のメソッド), 489
- settimeout() (socket のメソッド), 381
- settrace()
  - sys モジュール, 51
  - threading モジュール, 388
- setuid() (os モジュール), 222
- setUp() (TestCase のメソッド), 174
- setup() (SocketServer モジュール), 519
- setupterm() (curses モジュール), 282
- SetValue() (\_winreg モジュール), 778
- SetValueEx() (\_winreg モジュール), 778
- setwidth() (audio configuration のメソッド), 752
- SGML, 614
- sgmllib
  - standard module, 614
  - 標準モジュール, 616
- SGMLParser
  - in module sgmllib, 616
  - sgmllib のクラス, 614
- sha (built-in module), 683
- Shelf (shelve のクラス), 89
- shelve
  - standard module, 89
  - 標準モジュール, 92
- shifting
  - operations, 20
- shlex
  - shlex のクラス, 214
  - standard module, 214
- shortDescription() (TestCase のメソッド), 175
- shouldFlush()
  - BufferingHandler のメソッド, 364
  - MemoryHandler のメソッド, 364
- show() ( のメソッド), 299
- show\_choice() (fl モジュール), 758
- show\_file\_selector() (fl モジュール), 758
- show\_form() (form のメソッド), 759
- show\_input() (fl モジュール), 758
- show\_message() (fl モジュール), 758
- show\_object() (FORMS object のメソッド), 762
- show\_question() (fl モジュール), 758
- showsyntaxerror() (InteractiveConsole のメソッド), 101
- showtraceback() (InteractiveConsole のメソッド), 101
- showwarning() (warnings モジュール), 95
- shuffle() (random モジュール), 186
- shutdown()
  - IMAP4\_stream のメソッド, 503
  - logging モジュール, 357
  - socket のメソッド, 381
- shutil (standard module), 337
- SIG\* (signal のデータ), 372
- SIG\_DFL (signal のデータ), 372
- SIG\_IGN (signal のデータ), 372
- signal() (signal モジュール), 373
- signal
  - built-in module, 371
  - 組み込みモジュール, 387
- Simple Mail Transfer Protocol, 508
- simple\_producer (asynchat のクラス), 539
- SimpleCookie (Cookie のクラス), 524
- SimpleHTTPRequestHandler (SimpleHTTPServer のクラス), 522
- SimpleHTTPServer
  - standard module, 522
  - 標準モジュール, 519
- SimpleXMLRPCRequestHandler (SimpleXMLRPCServer のクラス), 531

- SimpleXMLRPCServer
  - SimpleXMLRPCServer のクラス, [531](#)
  - standard module, [531](#)
- sin()
  - cmath モジュール, [184](#)
  - math モジュール, [183](#)
- sinh()
  - cmath モジュール, [184](#)
  - math モジュール, [183](#)
- site (standard module), [108](#)
- site-packages
  - directory, [108](#)
- site-python
  - directory, [108](#)
- sitcustomize (モジュール), [109](#)
- size()
  - のメソッド, [399](#)
  - FTP のメソッド, [497](#)
- size (TarInfo の属性), [419](#)
- sizeofimage() (rgbimg モジュール), [673](#)
- skip() (Chunk のメソッド), [672](#)
- skipinitialspace (Dialect の属性), [610](#)
- skippedEntity() (ContentHandler のメソッド), [649](#)
- slave() (NNTPDataError のメソッド), [507](#)
- sleep() (time モジュール), [271](#)
- slice
  - assignment, [28](#)
  - operation, [22](#)
- slice()
  - モジュール, [13](#)
  - 組み込み関数, [63](#), [742](#)
- SliceType (types のデータ), [63](#)
- SmartCookie (Cookie のクラス), [524](#)
- SMTP
  - protocol, [508](#)
- SMTP (smtplib のクラス), [508](#)
- SMTPConnectError (smtplib の例外), [509](#)
- SMTPDataError (smtplib の例外), [509](#)
- SMTPException (smtplib の例外), [509](#)
- SMTPHandler (logging のクラス), [363](#)
- SMTPHeloError (smtplib の例外), [509](#)
- smtplib (standard module), [508](#)
- SMTPRecipientsRefused (smtplib の例外), [509](#)
- SMTPResponseException (smtplib の例外), [509](#)
- SMTPSenderRefused (smtplib の例外), [509](#)
- SMTPServerDisconnected (smtplib の例外), [509](#)
- SND\_ALIAS (winsound のデータ), [780](#)
- SND\_ASYNC (winsound のデータ), [781](#)
- SND\_FILENAME (winsound のデータ), [780](#)
- SND\_LOOP (winsound のデータ), [780](#)
- SND\_MEMORY (winsound のデータ), [780](#)
- SND\_NODEFAULT (winsound のデータ), [781](#)
- SND\_NOSTOP (winsound のデータ), [781](#)
- SND\_NOWAIT (winsound のデータ), [781](#)
- SND\_PURGE (winsound のデータ), [780](#)
- sndhdr (standard module), [674](#)
- sniff() (Sniffer のメソッド), [608](#)
- Sniffer (csv のクラス), [608](#)
- SO\_\* (socket のデータ), [375](#)
- SOCK\_DGRAM (socket のデータ), [375](#)
- SOCK\_RAW (socket のデータ), [375](#)
- SOCK\_RDM (socket のデータ), [375](#)
- SOCK\_SEQPACKET (socket のデータ), [375](#)
- SOCK\_STREAM (socket のデータ), [375](#)
- socket
  - object, [374](#)
  - オブジェクト, [374](#)
- socket()
  - IMAP4\_stream のメソッド, [503](#)
  - socket モジュール, [377](#)
- socket
  - built-in module, [374](#)
  - SocketServer のデータ, [518](#)
  - 組み込みモジュール, [31](#), [465](#)
- socket() (in module socket), [385](#)
- socket\_type (SocketServer のデータ), [518](#)
- SocketHandler (logging のクラス), [361](#)
- SocketServer (standard module), [517](#)
- SocketType (socket のデータ), [379](#)
- softspace (file の属性), [33](#)
- SOL\_\* (socket のデータ), [375](#)
- SOMAXCONN (socket のデータ), [375](#)
- sort()
  - IMAP4\_stream のメソッド, [503](#)
  - list method, [28](#)
- sort\_stats() (Stats のメソッド), [456](#)
- sortTestMethodsUsing (TestLoader の属性), [177](#)
- source (shlex の属性), [216](#)
- sourcehook() (shlex のメソッド), [215](#)

`span()` (`MatchObject` のメソッド), 126  
`spawn()` (`pty` モジュール), 434  
`spawnl()` (`os` モジュール), 235  
`spawnle()` (`os` モジュール), 235  
`spawnlp()` (`os` モジュール), 235  
`spawnlpe()` (`os` モジュール), 235  
`spawnv()` (`os` モジュール), 235  
`spawnve()` (`os` モジュール), 235  
`spawnvp()` (`os` モジュール), 235  
`spawnvpe()` (`os` モジュール), 235  
`specified_attributes` (`xmlparser` の属性), 621  
`speed()` (`audio device` のメソッド), 677  
`split()`  
     `os.path` モジュール, 241  
     `RegexObject` のメソッド, 125  
     `re` モジュール, 123  
     `shlex` モジュール, 214  
     `string` のメソッド, 24  
     `string` モジュール, 115  
`splitdrive()` (`os.path` モジュール), 241  
`splittext()` (`os.path` モジュール), 241  
`splitfields()` (`string` モジュール), 115  
`splitlines()` (`string` のメソッド), 25  
`sprintf-style formatting`, 25  
`sqrt()`  
     `cmath` モジュール, 184  
     `math` モジュール, 183  
     `mpz` モジュール, 684  
`sqrtrem()` (`mpz` モジュール), 684  
`ssl()`  
     `IMAP4_stream` のメソッド, 504  
     `socket` モジュール, 377  
`ST_ETIME` (`stat` のデータ), 243  
`ST_CTIME` (`stat` のデータ), 244  
`ST_DEV` (`stat` のデータ), 243  
`ST_GID` (`stat` のデータ), 243  
`ST_INO` (`stat` のデータ), 243  
`ST_MODE` (`stat` のデータ), 243  
`ST_MTIME` (`stat` のデータ), 244  
`ST_NLINK` (`stat` のデータ), 243  
`ST_SIZE` (`stat` のデータ), 243  
`ST_UID` (`stat` のデータ), 243  
`stack viewer`, 709  
`stackable`  
     `streams`, 143  
`StandardError` (`exceptions` の例外), 37  
`standend()` (`window` のメソッド), 288  
`standout()` (`window` のメソッド), 288  
`starmap()` (`itertools` モジュール), 202  
`start()`  
     `MatchObject` のメソッド, 126  
     `Profile` のメソッド, 460  
     `Thread` のメソッド, 394  
`start_color()` (`curses` モジュール), 282  
`start_new_thread()` (`thread` モジュール), 386  
`startbody()` (`MimeWriter` のメソッド), 589  
`StartCdataSectionHandler()` (`xmlparser` のメソッド), 623  
`StartDoctypeDeclHandler()` (`xmlparser` のメソッド), 621  
`startDocument()` (`ContentHandler` のメソッド), 647  
`startElement()` (`ContentHandler` のメソッド), 647  
`StartElementHandler()` (`xmlparser` のメソッド), 622  
`startElementNS()` (`ContentHandler` のメソッド), 648  
`startfile()` (`os` モジュール), 236  
`startmultipartbody()` (`MimeWriter` のメソッド), 589  
`StartNamespaceDeclHandler()` (`xmlparser` のメソッド), 622  
`startPrefixMapping()` (`ContentHandler` のメソッド), 647  
`startswith()` (`string` のメソッド), 25  
`startTest()` (`TestResult` のメソッド), 176  
`starttls()` (`SMTP` のメソッド), 510  
`stat()`  
     `NNTPDataError` のメソッド, 507  
     `os` モジュール, 229  
     `POP3` のメソッド, 498  
     `statcache` モジュール, 244  
`stat`  
     `standard module`, 242  
     標準モジュール, 229  
`stat_float_times()` (`os` モジュール), 230  
`statcache` (`standard module`), 244  
`statement`  
     `assert`, 38  
     `del`, 28, 29  
     `except`, 37  
     `exec`, 35

- if, 17
- import, 3, 96
- print, 17
- raise, 37
- try, 37
- while, 17
- staticmethod() (モジュール), 14
- Stats (pstats のクラス), 455
- status() (IMAP4\_stream のメソッド), 503
- status (httplib のデータ), 492
- statvfs() (os モジュール), 230
- statvfs
  - standard module, 245
  - 標準モジュール, 230
- StdButtonBox (Tix のクラス), 701
- stderr (sys のデータ), 51
- stdin (sys のデータ), 51
- stdout (sys のデータ), 51
- Stein, Greg, 744
- stereocontrols() (mixer device のメソッド), 678
- STILL (cd のデータ), 754
- stop()
  - CD player のメソッド, 756
  - Profile のメソッド, 460
  - TestResult のメソッド, 176
- StopIteration (exceptions の例外), 39
- stopListening() (logging モジュール), 367
- stopTest() (TestResult のメソッド), 176
- storbinary() (FTP のメソッド), 495
- store() (IMAP4\_stream のメソッド), 503
- STORE\_ACTIONS ( の属性), 324
- storlines() (FTP のメソッド), 496
- str()
  - locale モジュール, 341
  - モジュール, 14
- strcoll() (locale モジュール), 341
- StreamError (tarfile の例外), 416
- StreamHandler (logging のクラス), 360
- StreamReader (codecs のクラス), 148
- StreamReaderWriter (codecs のクラス), 149
- StreamRecoder (codecs のクラス), 149
- streams, 143
  - stackable, 143
- StreamWriter (codecs のクラス), 147
- strerror() (os モジュール), 222
- strftime()
  - date のメソッド, 255
  - datetime のメソッド, 261
  - time のメソッド, 262
  - time モジュール, 271
- strict\_errors() (codecs モジュール), 144
- string
  - documentation, 724
  - formatting, 25
  - interpolation, 25
  - object, 21
  - オブジェクト, 21
- string
  - MatchObject の属性, 127
  - standard module, 113
  - 標準モジュール, 27, 341, 344
- StringIO
  - standard module, 140
  - StringIO のクラス, 140
- stringprep (standard module), 154
- StringType (types のデータ), 62
- StringTypes (types のデータ), 63
- strip()
  - string のメソッド, 25
  - string モジュール, 116
- strip\_dirs() (Stats のメソッド), 455
- stripspaces (Textbox の属性), 294
- strptime() (time モジュール), 273
- struct (built-in module), 128
- struct (組み込みモジュール), 380, 381
- struct\_time (time のデータ), 273
- structures
  - C, 128
- strxfrm() (locale モジュール), 341
- sub()
  - operator モジュール, 68
  - RegexObject のメソッド, 125
  - re モジュール, 123
- subdirs (dircmp の属性), 247
- subn()
  - RegexObject のメソッド, 125
  - re モジュール, 124
- subpad() (window のメソッド), 288
- subscribe() (IMAP4\_stream のメソッド), 503
- subscript
  - assignment, 28
  - operation, 22



`subsequent_indent` (`TextWrapper` の属性), 142

`subwin()` (`window` のメソッド), 288

`suffix_map` (`mimetypes` のデータ), 587

`suite()` (`parser` モジュール), 720

`suiteClass` (`TestLoader` の属性), 178

`sum()` (モジュール), 14

`sunau` (standard module), 666

`SUNAUDIODEV`  
    standard module, 770  
    標準モジュール, 769

`sunaudiodev`  
    built-in module, 769  
    組み込みモジュール, 770

`super()` (モジュール), 14

`super` (class descriptor の属性), 732

`supports_unicode_filenames` (`os.path` のデータ), 242

`swapcase()`  
    string のメソッド, 25  
    string モジュール, 116

`sym()` ( のメソッド), 429

`sym_name` (symbol のデータ), 729

`symbol` (standard module), 729

symbol table, 3

`symlink()` (`os` モジュール), 230

`sync()` ( のメソッド), 404, 405

`sync()`  
    audio device のメソッド, 677  
    dbhash のメソッド, 401  
    gdbm モジュール, 432

`syncdown()` (`window` のメソッド), 288

`syncok()` (`window` のメソッド), 288

`syncup()` (`window` のメソッド), 288

`syntax_error()` (`XMLParser` のメソッド), 657

`SyntaxErr` (`xml.dom` の例外), 636

`SyntaxError` (exceptions の例外), 39

`SyntaxWarning` (exceptions の例外), 41

`sys` (built-in module), 46

`sys_version` (`BaseHTTPRequestHandler` の属性), 520

`sysconf()` (`os` モジュール), 238

`sysconf_names` (`os` のデータ), 238

`syslog()` (`syslog` モジュール), 443

`syslog` (built-in module), 443

`SysLogHandler` (logging のクラス), 362

`system()` (`os` モジュール), 236

`system.listMethods()` (`ServerProxy` のメソッド), 529

`system.methodHelp()` (`ServerProxy` のメソッド), 529

`system.methodSignature()` (`ServerProxy` のメソッド), 529

`SystemError` (exceptions の例外), 39

`SystemExit` (exceptions の例外), 40

`systemId` (`DocumentType` の属性), 631

## T

`T_FMT` (locale のデータ), 342

`T_FMT_AMPM` (locale のデータ), 342

`tabnanny` (standard module), 731

tabular  
    data, 606

`tagName` (`Element` の属性), 633

`takewhile()` (`itertools` モジュール), 202

`tan()`  
    cmath モジュール, 185  
    math モジュール, 183

`tanh()`  
    cmath モジュール, 185  
    math モジュール, 183

`TAR_GZIPPED` (tarfile のデータ), 416

`TAR_PLAIN` (tarfile のデータ), 416

`TarError` (tarfile の例外), 416

`TarFile` (tarfile のクラス), 416, 417

`tarfile` (standard module), 415

`TarFileCompat` (tarfile のクラス), 416

`target` (`ProcessingInstruction` の属性), 635

`TarInfo` (tarfile のクラス), 419

`tb_lineno()` (`traceback` モジュール), 75

`tcdrain()` (`termios` モジュール), 432

`tcflow()` (`termios` モジュール), 432

`tcflush()` (`termios` モジュール), 432

`tcgetattr()` (`termios` モジュール), 432

`tcgetpgrp()` (`os` モジュール), 225

`TCP_*` (`socket` のデータ), 376

`tcsendbreak()` (`termios` モジュール), 432

`tcsetattr()` (`termios` モジュール), 432

`tcsetpgrp()` (`os` モジュール), 225

`tearDown()` (`TestCase` のメソッド), 174

`tell()`  
    のメソッド, 399  
    aifc のメソッド, 665  
    AU\_read のメソッド, 668

- AU\_write のメソッド, 668
- BZ2File のメソッド, 410
- Chunk のメソッド, 672
- file のメソッド, 32
- MultiFile のメソッド, 591
- Wave\_read のメソッド, 670
- Wave\_write のメソッド, 670
- Telnet (telnetlib のクラス), 512
- telnetlib (standard module), 512
- TEMP, 329
- tempdir (tempfile のデータ), 329
- tempfile (standard module), 327
- Template (pipes のクラス), 436
- template (tempfile のデータ), 329
- tempnam() (os モジュール), 230
- temporary
  - file, 327
  - file name, 327
- TemporaryFile() (tempfile モジュール), 327
- termattrs() (curses モジュール), 282
- TERMIOS (standard module), 433
- termios
  - built-in module, 432
  - 組み込みモジュール, 433
- termname() (curses モジュール), 282
- test()
  - cgi モジュール, 472
  - mutex のメソッド, 276
- test (standard module), 178
- test.test\_support (standard module), 180
- testandset() (mutex のメソッド), 276
- TestCase (unittest のクラス), 173
- TestFailed (test.test\_support の例外), 181
- TESTFN (test.test\_support のデータ), 181
- TestLoader (unittest のクラス), 173
- testMethodPrefix (TestLoader の属性), 177
- testmod() (doctest モジュール), 163
- tests (imgchr のデータ), 674
- TestSkipped (test.test\_support の例外), 181
- testsource() (doctest モジュール), 163
- testsRun (TestResult の属性), 176
- TestSuite (unittest のクラス), 173
- testzip() (ZipFile のメソッド), 413
- Textbox (curses.textpad のクラス), 294
- textdomain() (gettext モジュール), 346
- TextTestRunner (unittest のクラス), 173
- textwrap (standard module), 141
- TextWrapper (textwrap のクラス), 142
- THOUSEP (locale のデータ), 343
- Thread (threading のクラス), 388, 394
- thread (built-in module), 386
- threading (standard module), 387
- threads
  - IRIX, 387
  - POSIX, 386
- tie() (fl モジュール), 759
- tigetflag() (curses モジュール), 282
- tigetnum() (curses モジュール), 282
- tigetstr() (curses モジュール), 282
- time()
  - datetime のメソッド, 259
  - time モジュール, 273
- time
  - built-in module, 269
  - datetime のクラス, 250, 261
- Time2Internaldate() (imaplib モジュール), 500
- timedelta (datetime のクラス), 251
- timegm() (calendar モジュール), 211
- timeit() (Timer のメソッド), 462
- timeit (standard module), 461
- timeout() (window のメソッド), 288
- timeout (socket の例外), 375
- Timer
  - threading のクラス, 388, 395
  - timeit のクラス, 461
- times() (os モジュール), 237
- timetuple()
  - date のメソッド, 255
  - datetime のメソッド, 260
- timetz() (datetime のメソッド), 259
- timezone (time のデータ), 273
- title() (string のメソッド), 25
- Tix, 699
- Tix
  - standard module, 699
  - Tix のクラス, 700
- tix\_addbitmapdir() (tixCommand のメソッド), 704
- tix\_cget() (tixCommand のメソッド), 704
- tix\_configure() (tixCommand のメソッド), 704
- tix\_filedialog() (tixCommand のメソッド), 704

tix\_getbitmap() (tixCommand のメソッド),  
     704  
 tix\_getimage() (tixCommand のメソッド),  
     704  
 TIX\_LIBRARY, 700  
 tix\_option\_get() (tixCommand のメソッド),  
     704  
 tix\_resetoptions() (tixCommand のメソッド), 704  
 tixCommand (Tix のクラス), 703  
 Tk, 687  
 Tk (Tkinter のクラス), 688  
 Tk Option Data Types, 696  
 Tkinter, 687  
 Tkinter (standard module), 687  
 TList (Tix のクラス), 702  
 TMP, 329  
 TMP\_MAX (os のデータ), 231  
 TMPDIR, 329  
 tmpfile() (os モジュール), 223  
 tmpnam() (os モジュール), 230  
 to\_splittable() (Charset のメソッド), 566  
 ToASCII() (encodings.idna モジュール), 153  
 tobuf() (TarInfo のメソッド), 419  
 tochild (Popen4 の属性), 248  
 today()  
     date のメソッド, 253  
     datetime のメソッド, 256  
 tofile() (array のメソッド), 195  
 togglepause() (CD player のメソッド), 756  
 tok\_name (token のデータ), 729  
 token  
     shlex の属性, 216  
     standard module, 729  
 tokeneater() (tabnanny モジュール), 731  
 tokenize() (tokenize モジュール), 730  
 tokenize (standard module), 730  
 tolist()  
     array のメソッド, 195  
     AST のメソッド, 723  
 tomono() (audioop モジュール), 661  
 toordinal()  
     date のメソッド, 255  
     datetime のメソッド, 260  
 top()  
     のメソッド, 299  
     POP3 のメソッド, 499  
 top\_panel() (curses.panel モジュール), 298  
 toprettyxml() (のメソッド), 640  
 tostereo() (audioop モジュール), 661  
 tostring() (array のメソッド), 195  
 totuple() (AST のメソッド), 723  
 touchline() (window のメソッド), 288  
 touchwin() (window のメソッド), 289  
 ToUnicode() (encodings.idna モジュール), 153  
 tounicode() (array のメソッド), 195  
 tovideo() (imageop モジュール), 663  
 toxml() (のメソッド), 639  
 tparm() (curses モジュール), 283  
 trace function, 388  
 traceback  
     object, 47, 74  
     オブジェクト, 47, 74  
 traceback (standard module), 74  
 tracebacklimit (sys のデータ), 51  
 tracebacks  
     in CGI scripts, 475  
 TracebackType (types のデータ), 63  
 tracer() (turtle モジュール), 705  
 transfercmd() (FTP のメソッド), 496  
 translate()  
     string のメソッド, 25  
     string モジュール, 116  
 translate\_references() (XMLParser のメソッド), 656  
 translation() (gettext モジュール), 347  
 Tree (Tix のクラス), 702  
 True, 17, 36  
 True (のデータ), 42  
 true, 17  
 truediv() (operator モジュール), 68  
 truncate() (file のメソッド), 32  
 truth  
     value, 17  
 truth() (operator モジュール), 66  
 try  
     実行文, 37  
     statement, 37  
 ttob()  
     imgfile モジュール, 767  
     rgbimg モジュール, 673  
 tty  
     I/O control, 432, 433  
 tty (standard module), 433

`ttynam()` (os モジュール), 225

`tuple`

object, 21

オブジェクト, 21

`tuple()` (モジュール), 14

`tuple2ast()` (parser モジュール), 721

`TupleType` (types のデータ), 62

`turnoff_sigfpe()` (fpectl モジュール), 59

`turnon_sigfpe()` (fpectl モジュール), 59

`turtle` (standard module), 705

Tutt, Bill, 744

`type`

Boolean, 4

object, 15

オブジェクト, 15

operations on dictionary, 29

operations on list, 28

`type()`

モジュール, 14

組み込み関数, 35, 62

`type` (TarInfo の属性), 419

`typeahead()` (curses モジュール), 283

`typecode` (array の属性), 193

`TYPED_ACTIONS` ( の属性), 324

`typed_subpart_iterator()` (email.Iterators  
モジュール), 571

`TypeError` (exceptions の例外), 40

`types`

built-in, 3, 17

mutable sequence, 28

operations on integer, 20

operations on mapping, 29

operations on mutable sequence, 28

operations on numeric, 19

operations on sequence, 22, 28

`types`

standard module, 61

標準モジュール, 15, 35

`types_map` (mimetypes のデータ), 587, 588

`TypeType` (types のデータ), 62

TZ, 273, 274, 785

`tzinfo`

datetime の属性, 258

datetime のクラス, 251

time の属性, 262

`tzname()` (datetime のメソッド), 260

`tzname()` (time のメソッド), 263, 264

`tzname` (time のデータ), 273

`tzset()` (time モジュール), 273

## U

U (re のデータ), 123

u-LAW, 659, 665, 674, 769

`ugettext()`

GNUTranslations のメソッド, 349

NullTranslations のメソッド, 348

`uid()` (IMAP4\_stream のメソッド), 504

`uid, gid` (TarInfo の属性), 419

`uidl()` (POP3 のメソッド), 499

`ulaw2lin()` (audioop モジュール), 661

`umask()` (os モジュール), 222

`uname()` (os モジュール), 222

`uname, gname` (TarInfo の属性), 419

`UnboundLocalError` (exceptions の例外), 40

`UnboundMethodType` (types のデータ), 63

unbuffered I/O, 8

UNC paths

and `os.makedirs()`, 228

`unconsumed_tail` ( の属性), 407

`unctrl()`

curses.ascii モジュール, 297

curses モジュール, 283

`undoc_header` (Cmd の属性), 213

`unescape()` (xml.sax.saxutils モジュール), 650

`unfreeze_form()` (form のメソッド), 759

`unfreeze_object()` (FORMS object のメソッ  
ド), 762

`ungetch()`

curses モジュール, 283

msvcrt モジュール, 774

`ungetmouse()` (curses モジュール), 283

`ungettext()`

GNUTranslations のメソッド, 349

NullTranslations のメソッド, 348

`unhexlify()` (binascii モジュール), 599

`unichr()` (モジュール), 15

UNICODE (re のデータ), 123

Unicode, 143, 153

database, 153

object, 21

オブジェクト, 21

`unicode()` (モジュール), 15

`unicodedata` (standard module), 153

`UnicodeDecodeError` (exceptions の例外), 40

UnicodeEncodeError (exceptions の例外), 40  
 UnicodeError (exceptions の例外), 40  
 UnicodeTranslateError (exceptions の例外), 40  
 UnicodeType (types のデータ), 62  
 unidata\_version (unicodedata のデータ), 154  
 unified\_diff() (difflib モジュール), 133  
 uniform()  
     random モジュール, 187  
     whrandom モジュール, 188  
 UnimplementedFileMode (httplib の例外), 491  
 unittest (standard module), 166  
 UNIX  
     file control, 434  
     I/O control, 434  
 unixfrom (AddressList の属性), 596  
 UnixMailbox (mailbox のクラス), 580  
 unknown\_charref()  
     SGMLParser のメソッド, 616  
     XMLParser のメソッド, 657  
 unknown\_endtag()  
     SGMLParser のメソッド, 615  
     XMLParser のメソッド, 657  
 unknown\_entityref()  
     SGMLParser のメソッド, 616  
     XMLParser のメソッド, 657  
 unknown\_open()  
     BaseHandler のメソッド, 486  
     UnknownHandler のメソッド, 490  
 unknown\_starttag()  
     SGMLParser のメソッド, 615  
     XMLParser のメソッド, 657  
 UnknownHandler (urllib2 のクラス), 484  
 UnknownProtocol (httplib の例外), 491  
 UnknownTransferEncoding (httplib の例外), 491  
 unlink()  
     のメソッド, 639  
     os モジュール, 231  
 unlock() (mutex のメソッド), 276  
 unmimify() (mimify モジュール), 589  
 unpack() (struct モジュール), 128  
 unpack\_array() (Unpacker のメソッド), 604  
 unpack\_bytes() (Unpacker のメソッド), 604  
 unpack\_double() (Unpacker のメソッド), 604  
 unpack\_farray() (Unpacker のメソッド), 604  
 unpack\_float() (Unpacker のメソッド), 604  
 unpack\_fopaque() (Unpacker のメソッド), 604  
 unpack\_fstring() (Unpacker のメソッド), 604  
 unpack\_list() (Unpacker のメソッド), 604  
 unpack\_opaque() (Unpacker のメソッド), 604  
 unpack\_string() (Unpacker のメソッド), 604  
 Unpacker (xdrlib のクラス), 602  
 unparsedEntityDecl() (DTDHandler のメソッド), 649  
 UnparsedEntityDeclHandler() (xmlparser のメソッド), 622  
 Unpickler (pickle のクラス), 81  
 UnpicklingError (pickle の例外), 80  
 unqdevice() (fl モジュール), 759  
 unquote()  
     email.Utils モジュール, 569  
     rfc822 モジュール, 593  
     urllib モジュール, 478  
 unquote\_plus() (urllib モジュール), 479  
 unregister() (のメソッド), 385  
 unregister\_dialect() (csv モジュール), 608  
 unsubscribe() (IMAP4\_stream のメソッド), 504  
 untouchwin() (window のメソッド), 289  
 unused\_data (の属性), 407  
 up() (turtle モジュール), 706  
 update()  
     dictionary method, 29  
     hmac のメソッド, 681  
     md5 のメソッド, 682  
     sha のメソッド, 683  
 update\_panels() (curses.panel モジュール), 298  
 upper()  
     string のメソッド, 25  
     string モジュール, 116  
 uppercase (string のデータ), 114  
 URL, 467, 476, 515, 519, 606  
     parsing, 515  
     relative, 515  
 url (ServerProxy の属性), 531  
 url2pathname() (urllib モジュール), 479  
 urlcleanup() (urllib モジュール), 478  
 urldefrag() (urlparse モジュール), 516  
 urlencode() (urllib モジュール), 479

URLError (urllib2 の例外), 483  
urljoin() (urlparse モジュール), 516  
urllib  
    standard module, 476  
    標準モジュール, 490  
urllib2 (standard module), 482  
urlopen()  
    urllib2 モジュール, 482  
    urllib モジュール, 476  
URLopener (urllib のクラス), 479  
urlparse() (urlparse モジュール), 515  
urlparse  
    standard module, 515  
    標準モジュール, 480  
urlretrieve() (urllib モジュール), 477  
urlsplit() (urlparse モジュール), 516  
urlunparse() (urlparse モジュール), 516  
urlunsplit() (urlparse モジュール), 516  
use\_env() (curses モジュール), 283  
use\_rawinput (Cmd の属性), 214  
USER, 277  
user  
    configuration file, 109  
    effective id, 221  
    id, 221  
    id, setting, 222  
user() (POP3 のメソッド), 498  
user (standard module), 109  
UserDict  
    standard module, 64  
    UserDict のクラス, 64  
UserList  
    standard module, 64  
    UserList のクラス, 64  
USERNAME, 277  
userptr() ( のメソッド), 299  
UserString  
    standard module, 65  
    UserString のクラス, 65  
UserWarning (exceptions の例外), 41  
UTC, 269  
utcfromtimestamp() (datetime のメソッド), 257  
utcnw() (datetime のメソッド), 256  
utcoffset() (datetime のメソッド), 259  
utcoffset() (time のメソッド), 262, 263  
utctimetuple() (datetime のメソッド), 260

utime() (os モジュール), 231  
uu  
    standard module, 601  
    標準モジュール, 598  
V  
value  
    truth, 17  
value (Morsel の属性), 525  
value\_decode() (BaseCookie のメソッド), 524  
value\_encode() (BaseCookie のメソッド), 524  
ValueError (exceptions の例外), 40  
values  
    Boolean, 36  
values()  
    dictionary method, 29  
    Message のメソッド, 551  
varray() (gl モジュール), 765  
vars() (モジュール), 15  
vbar (ScrolledText の属性), 705  
VERBOSE (re のデータ), 123  
verbose  
    tabnanny のデータ, 731  
    test.test\_support のデータ, 181  
verify() (SMTP のメソッド), 510  
verify\_request() (SocketServer モジュール), 518  
version  
    curses のデータ, 289  
    httplib のデータ, 492  
    sys のデータ, 51  
    URLopener の属性, 481  
version\_info (sys のデータ), 52  
version\_string() (BaseHTTPRequestHandler のメソッド), 521  
vline() (window のメソッド), 289  
vncarray() (gl モジュール), 765  
voidcmd() (FTP のメソッド), 495  
volume (ZipInfo の属性), 414  
vonmisesvariate() (random モジュール), 187

## W

W\_OK (os のデータ), 226  
wait()  
    Condition のメソッド, 391  
    Event のメソッド, 393  
    os モジュール, 237



Popen4 のメソッド, 248  
 waitpid() (os モジュール), 237  
 walk()  
     compiler.visitor モジュール, 750  
     compiler モジュール, 743  
     Message のメソッド, 554  
     os.path モジュール, 241  
     os モジュール, 231  
 warn() (warnings モジュール), 95  
 warn\_explicit() (warnings モジュール), 95  
 Warning (exceptions の例外), 41  
 warning()  
     のメソッド, 358  
     ErrorHandler のメソッド, 649  
     logging モジュール, 356  
 warnings, 93  
 warnings (standard module), 93  
 warnoptions (sys のデータ), 52  
 wasSuccessful() (TestResult のメソッド), 176  
 wave (standard module), 668  
 WeakKeyDictionary (weakref のクラス), 56  
 weakref (extension module), 54  
 WeakValueDictionary (weakref のクラス), 56  
 webbrowser (standard module), 465  
 weekday()  
     calendar モジュール, 211  
     date のメソッド, 255  
     datetime のメソッド, 260  
 weibullvariate() (random モジュール), 187  
 WEXITSTATUS() (os モジュール), 237  
 wfile (BaseHTTPRequestHandler の属性), 520  
 what()  
     imghdr モジュール, 673  
     sndhdr モジュール, 674  
 whathdr() (sndhdr モジュール), 674  
 whichdb() (whichdb モジュール), 402  
 whichdb (standard module), 402  
 while  
     実行文, 17  
     statement, 17  
 whitespace  
     shlex の属性, 216  
     string のデータ, 114  
 whitespace\_split (shlex の属性), 216  
 whrandom (standard module), 188  
 whseed() (random モジュール), 188  
 WichmannHill (random のクラス), 188  
 width() (turtle モジュール), 706  
 width (TextWrapper の属性), 142  
 WIFEXITED() (os モジュール), 237  
 WIFSIGNALED() (os モジュール), 237  
 WIFSTOPPED() (os モジュール), 237  
 Wimp\$ScrapDir, 329  
 window() ( のメソッド), 299  
 window manager (widgets), 696  
 Windows ini file, 205  
 WindowsError (exceptions の例外), 40  
 WinSock, 385  
 winsound (built-in module), 779  
 winver (sys のデータ), 52  
 WNOHANG (os のデータ), 237  
 wordchars (shlex の属性), 216  
 World Wide Web, 465, 476, 515, 606  
 wrap()  
     TextWrapper のメソッド, 143  
     textwrap モジュール, 141  
 wrapper() (curses.wrapper モジュール), 295  
 writable()  
     async\_chat のメソッド, 539  
     dispatcher のメソッド, 536  
 write()  
     のメソッド, 382, 399  
     array のメソッド, 195  
     audio device のメソッド, 676, 770  
     BZ2File のメソッド, 410  
     file のメソッド, 32  
     Generator のメソッド, 559  
     imgfile モジュール, 767  
     InteractiveConsole のメソッド, 101  
     os モジュール, 225  
     SafeConfigParser のメソッド, 207  
     StreamWriter のメソッド, 147  
     Telnet のメソッド, 514  
     turtle モジュール, 706  
     ZipFile のメソッド, 413  
 write\_byte() ( のメソッド), 399  
 write\_history\_file() (readline モジュール), 421  
 writeall() (audio device のメソッド), 676  
 writeframes()  
     aifc のメソッド, 665  
     AU\_write のメソッド, 668  
     Wave\_write のメソッド, 670  
 writeframesraw()

aifc のメソッド, 665  
AU\_write のメソッド, 668  
Wave\_write のメソッド, 670  
writelines()  
    BZ2File のメソッド, 410  
    file のメソッド, 32  
    StreamWriter のメソッド, 147  
writepy() (PyZipFile のメソッド), 413  
writer() (csv モジュール), 607  
writer(formatter の属性), 544  
writerow() (csv writer のメソッド), 610  
writerows() (csv writer のメソッド), 610  
writesamps() (audio port のメソッド), 753  
writestr() (ZipFile のメソッド), 413  
writexml() ( のメソッド), 639  
WrongDocumentErr (xml.dom の例外), 636  
WSTOPSIG() (os モジュール), 237  
WTERMSIG() (os モジュール), 237  
WWW, 465, 476, 515, 606  
    server, 467, 519  
  
**X**  
X(re のデータ), 123  
X\_OK (os のデータ), 226  
xatom() (IMAP4\_stream のメソッド), 504  
XDR, 78, 602  
xdrlib (standard module), 602  
xgtitle() (NNTPDataError のメソッド), 508  
xhdr() (NNTPDataError のメソッド), 507  
XHTML, 611  
XHTML\_NAMESPACE (xml.dom のデータ), 628  
XML, 655  
    namespaces, 658  
xml.dom (standard module), 626  
xml.dom.minidom (standard module), 637  
xml.dom.pulldom (standard module), 642  
xml.parsers.expat (standard module), 618  
xml.sax (standard module), 643  
xml.sax.handler (standard module), 645  
xml.sax.saxutils (standard module), 650  
xml.sax.xmlreader (standard module), 651  
XML\_NAMESPACE (xml.dom のデータ), 628  
xmlcharrefreplace\_errors\_errors()  
    (codecs モジュール), 144  
XmlDeclHandler() (xmlparser のメソッド),  
    621  
XMLFilterBase (xml.sax.saxutils のクラス), 650

XMLGenerator (xml.sax.saxutils のクラス), 650  
xmllib (standard module), 655  
XMLNS\_NAMESPACE (xml.dom のデータ), 628  
XMLParser (xmllib のクラス), 655  
XMLParserType (xml.parsers.expat のデータ),  
    619  
XMLReader (xml.sax.xmlreader のクラス), 651  
xmlrpclib (standard module), 528  
xor() (operator モジュール), 68  
xover() (NNTPDataError のメソッド), 508  
xpath() (NNTPDataError のメソッド), 508  
xrange  
    object, 21, 27  
    オブジェクト, 21, 27  
xrange()  
    モジュール, 16  
    組み込み関数, 21, 63  
XRangeType (types のデータ), 63  
xreadlines()  
    BZ2File のメソッド, 410  
    file のメソッド, 32  
    xreadlines モジュール, 210  
xreadlines (extension module), 209

## Y

Y2K, 269  
year  
    date の属性, 254  
    datetime の属性, 257  
Year 2000, 269  
Year 2038, 269  
YESEXPR (locale のデータ), 343  
yiq\_to\_rgb() (colorsys モジュール), 672

## Z

ZeroDivisionError (exceptions の例外), 40  
zfill()  
    string のメソッド, 25  
    string モジュール, 116  
zip() (モジュール), 16  
ZIP\_DEFLATED (zipfile のデータ), 412  
ZIP\_STORED (zipfile のデータ), 412  
ZipFile (zipfile のクラス), 411, 412  
zipfile (standard module), 411  
ZipInfo (zipfile のクラス), 411  
zlib (built-in module), 406