Python ライブラリリファレンス

リリース *2.5*

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

19th September, 2006

Python Software Foundation
Email: docs@python.org

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、プログラミング言語 Python の厳密な構文とセマンティクスについて説明していますが、Python とともに配付され，Python をすぐに活用する上で大いに役立つ標準ライブラリについては説明していません。このライブラリには、例えばファイル I/O のように， Python プログラマ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、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 pure Python で書かれたモジュールが入っています。これら数多くのモジュールには、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について説明しています (ライブラリモジュールの中には，プラットフォームでのサポートやコンパイル時の設定によって、使えたり使えなかったりするものがあります). 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で説明していなかったり，説明不足であるような多くの点についても説明しています。

このマニュアルでは、読者が Python 言語について基礎的な知識を持っていると仮定しています。形式ばらずに Python を学んでみたければ、*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、高度な文法とセマンティクス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 を組み込む方法が書かれています。

目次

一章

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ろいろあります。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！

二章

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。[1]

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いて、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。)[2]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*[, *level*]]]])

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(), [], -1)` になります; 文 'from spam.ham import eggs' は '`__import__('spam.ham', globals(), locals(), ['eggs'], -1)`' 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りませんが、'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:

[1] 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。
[2]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ので、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
```

level で絶対インポートを使うか相対インポートを使うかを指定します。デフォルトは -1 で、この値は絶対と相対の両インポートを試すことを示します。0 を指定すると、絶対インポートだけ行なう、という意味になります。*level* が正の値ならば、__import__ を呼び出すモジュールのディレクトリから幾つ上の親ディレクトリまで探索するか、を意味します。 2.5 で変更された仕様: level パラメータが追加されました 2.5 で変更された仕様: 引数のキーワードサポートが追加されました

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

all(*iterable*)

iterable の全ての要素が真ならば True を返します。以下のコードと等価です。

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

2.5 で追加された仕様です。

any(*iterable*)

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

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

2.5 で追加された仕様です。

basestring()

この抽象型は、 str および unicode のスーパクラスです。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が、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、以下の書きならわしを使います:

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

`@classmethod` は関数デコレータ形式です。詳しくは*../ref/ref.html*Python リファレンスマニュアルの 7 章にある関数定義についての説明を参照してください。

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。

もっとクラスメソッドについての情報が必要ならば、*Python* **リファレンスマニュアル**の 3 章にある標準型階層についてのドキュメントを繙いてください。 2.2 で追加された仕様です。 2.4 で変更された仕様: 関数デコレータ構文を追加しました

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()` を呼び出して評価できます。*filename* 引数にはコードの読み出し元のファイル名を指定します。コードをファイルから読み出したのでない場合には、それとわかるような値を渡します (一般的には `'<string>'` を使います)。引数 *kind* には、どの種類のコードをコンパイルするかを指定します。*string* が命令文の列からなる場合には `'exec'` 、単一の式からなる場合には `'eval'` 、単一の対話的な命令文からなる場合には`'single'` にします (最後のケースでは、式の評価結果が `None` 以外の場合に値を出力します)。

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'`) で表さねばなりません。また、入力行は少なくとも 1 つの改行文字で終端せねばなりません。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* のコンパイル時にどの future 文 (PEP 236 参照) の影響を及ぼすかを制御します。どちらも省略した場合 (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* が指定されていて、かつ *dont_inherit* が指定されていない (またはゼロ) の場合、上の場合に加えて *flags* に指定された future 文をいます。*dont_inherit* がゼロでない整数の場合、*flags* の値そのものを使い、この関数呼び出し周辺での future 文の効果は無視します。

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って複数の文を指定できます。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` 属性で得られます。

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ます。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か、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ければ、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シーケンス型か、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くてはなりません。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リストを返します。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リスト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。リスト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、リスト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、リスト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、リスト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。返されるリスト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 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 <= abs(*a* % *b*) < 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* はシーケンス型、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、(ゼロから始まる) 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*]])

文字列とオプションの引数 *globals*、*locals* をとります。*globals* を指定する場合には辞書でなくてはなりません。*locals* は任意のマップ型にできます。 2.4 で変更された仕様: 以前は *locals* も辞書でなければなりませんでした

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、'__builtins__' 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。[3]

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* は任意のマップ型に指定できます。 2.4 で変更された仕様: 以前は *locals* も辞書でなければなりませんでした *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*]])

`file` 型のコンストラクタです。詳しくは 3.9 節 “ファイルオブジェクト” を参照してください。コンストラクタの引数は後述の `open()` 組み込み関数と同じです。

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

2.2 で追加された仕様です。

`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* はシーケンスか、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。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。

[3] 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。

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。

frozenset([*iterable*])

frozenset オブジェクトを返します。要素は *iterable* から取得します。frozenset 型は、update メソッドを持たない代わりにハッシュ化でき、他の set 型の要素にしたり辞書型のキーにしたりできます。frozenset の要素自体は変更不能でなければなりません。集合 (set) 型の集合を表現するためには、内集合も frozenset オブジェクトでなければなりません。*iterable* を指定しない場合には空の集合 frozenset([]) を返します。 2.4 で追加された仕様です。

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。

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 True を、そうでない場合 False 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いてもです)。

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)。引数が与えられていない場合、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。引数が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 の式としても使える形式になります。 2.4 で変更された仕様: 以前は符号なしのリテラルしか返しませんでした

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 が送出されます。式を評価する際にエラーが生じた場合、他の例外も送出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` が送出されます。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 (他のシーケンス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 *clasinfo* 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、シーケンス型プロトコル (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()` を呼ぶ毎に *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。引数はシーケンス型 (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* はシーケンス、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。*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ず、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* 引数はどのようなシーケンス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...*][*key*])

単一の引数 *s* の場合、空でないシーケンス (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。

オプションの *key* 引数には `list.sort()` で使われるのと同じような 1 引数の順序付け関数を指定します。*key* を指定する場合はキーワード形式でなければなりません (たとえば '`max(a,b,c,key=func)`')。 2.5 で変更された仕様: オプションの *key* 引数が追加されました

min(*s*[, *args...*][*key*])

単一の引数 *s* の場合、空でないシーケンス (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。

オプションの *key* 引数には `list.sort()` で使われるのと同じような 1 引数の順序付け関数を指定します。*key* を指定する場合はキーワード形式でなければなりません (たとえば '`min(a,b,c,key=func)`')。 2.5 で変更された仕様: オプションの *key* 引数が追加されました

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 Python の式としても使える形式になります。 2.4 で変更された仕様: 以前は符号なしのリテラルしか返しませんでした

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

ファイルを開いて、3.9 節 "ファイルオブジェクト" に記述されている `file` 型のオブジェクトを返します。ファイルが開けなければ、`IOError` が送出されます。ファイルを開くときは `file` のコンストラクタを直接呼ばずに `open()` を使うのが望ましい方法です。

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。

最もよく使われる *mode* の値は、読み出しの '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) の 'w'、追記書き込みの '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) 。*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。) 他に *mode* に与えられる可能性のある値については後述します。

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。[4]

'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) 。バ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)。

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' 、Macintosh 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。

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

2.5 で変更された仕様: モード文字列の先頭についての制限が導入されました

ord(*c*)

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

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)。引数二つの `pow(x, y)` という形式は、冪乗演算子を使った *x****y* と等価です。

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

[4] 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。

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、三つめ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ず、*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 __init__(self): self._x = None
    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.")
```

doc がもし与えられたならばそれがプロパティ属性のドキュメント文字列になります。与えられない場合、プロパティは *fget* のドキュメント文字列 (がもしあれば) をコピーします。これにより、読み取り専用プロパティを `property()` をデコレータとして使って容易に作れるようになります。

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

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

のようにすると、`voltage()` が同じ名前の読み取り専用属性の “getter” になります。

2.2 で追加された仕様です。 2.5 で変更された仕様: *doc* が与えられない場合に *fget* のドキュメント文字列を使う

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 *proompt* 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、シーケンス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(labmda 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、計算の際にシーケンスの先頭に置かれます。また、シーケンス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。

reload(module) を実行すると、以下の処理が行われます:

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

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りません。

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ます:

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

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。通常の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。

reversed(*seq*)
要素を逆順に取り出すイテレータ (reverse iterator) を返します。*seq* はシーケンス型プロトコル (`__len__()` メソッド、および `0` から始まる整数を引数にとる`__getitem__()` メソッド) をサポートしていなければなりません。 2.4 で追加された仕様です。

round(*x*[, *n*])
x を小数点以下 *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)。

set([*iterable*])
集合を表現する `set` 型オブジェクトを返します。要素は *iterable* から取得します。要素は変更不能で

なければなりません。集合の集合を表現するには、内集合は `frozenset` オブジェクトでなければなりません。*iterable* を指定しない場合、新たな空の `set` 型オブジェクト、`set([])` を返します。 2.4 で追加された仕様です。

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ます。例えば、`setattr(`*x*`, '`*foobar*`', 123)` は *x*`.`*foobar* `= 123` と等価です。

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

iterable の要素をもとに、並べ替え済みの新たなリストを生成して返します。オプション引数 *cmp*、*key*、および *reverse* の意味は `list.sort()` メソッドと同じです。(3.6.4 節に説明があります。)

cmp は 2 つの引数 (iterable の要素) からなるカスタムの比較関数を指定します。これは始めの引数が 2 つ目の引数に比べて小さい、等しい、大きいかに応じて負数、ゼロ、正数を返します。'*cmp*=`lambda` *x*`,`*y*`: cmp(x.lower(), y.lower())`'

key は 1 つの引数からなる関数を指定します。これは個々のリストの要素から比較のキーを取り出すのに使われます。'*key*=`str.lower`'

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

一般的に、 *key* および *reverse* の変換プロセスは同等の *cmp* 関数を指定するより早く動作します。これは *key* および *reverse* がそれぞれの要素に一度だけ触れる間に、*cmp* はリストのそれぞれの要素に対して複数回呼ばれることによるものです。

2.4 で追加された仕様です。

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

`@staticmethod` は関数デコレータ形式です。詳しくは.*./ref/function.html*Python リファレンスマニュアルの 7 章にある関数定義についての説明を参照してください。

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。

もっと静的メソッドについての情報が必要ならば、*Python* **リファレンスマニュアル**の 3 章にある標準型階層についてのドキュメントを繙いてください。 2.2 で追加された仕様です。 2.4 で変更された仕様: 関数デコレータ構文を追加しました

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。文字列からなるシーケンスを結合する高速かつ正しい方法は″.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)
```

super は‘super(C, self).__getitem__(name)’のような明示的なドット表記の属性参照の一部として使われているので注意してください。これに伴って、super は‘super(C, self)[name]’のような文や演算子を使った非明示的な属性参照向けには定義されていないので注意してください。

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* はシーケンス、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。オブジェクトの型の検査には isinstance() 組み込み関数を使うことが推奨されます。

3 引数で呼び出された場合には type 関数は後述するようにコンストラクタとして働きます。

type(*name, bases, dict*)

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

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

2.2 で追加された仕様です。

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。引数の正当な範囲は Python がどのように構成されているかに依存しています— UCS2 ならば [0..0xFFFF] であり UCS4 ならば [0..0x10FFFF] であり、このどちらかです。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。[5]

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シーケンス型で、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。`ragne()` 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。

注意: `xrange()` はシンプルさと速度のために定義されている関数であり、その実現のために実装上の制限を課している場合があります。Python の C 実装では、全ての引数をネイティブの C long 型

[5] 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。

(Python の"short" 整数型) に制限しており、要素数がネイティブの C long 型の範囲内に収まるよう要求しています。

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

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シーケンスまたはイテレート可能オブジェクト中の *i* 番目の要素を含みます。返されるリストは引数のシーケンス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。引数が単一のシーケンスの場合、1 要素のタプルからなるリストを返します。引数を指定しない場合、空のリストを返します。 2.0 で追加された仕様です。

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

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* はシーケンス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。型変換が不可能である場合、`TypeError` を送出します。

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* 自体かコピーです。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、ユーザにも同様にするよう奨励しています。文字列による例外は Python 2.5 以降は DeprecationWarning 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 BaseException から導出された場合、関連値は例外インスタンスの args 属性中に置かれます。もし引数が一つならば (このようにすることが望まれますが)、その引数の値は message 属性に収められます。

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 クラスから導出するよう勧めます。BaseException からは導出しないで下さい。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 BaseException

全ての組み込み例外のルートクラスです。ユーザ定義例外を直接このクラスから導出することは意図していません (そういう場合は Exception を使ってください)。このクラスに対して str() や unicode() が呼ばれた場合、引数の文字列表現かまたは引数が無い時には空文字列が返されます。一つだけの引数が渡された場合、それが message 属性に格納されます。二つ以上の引数が渡された場合、message 属性は空文字列になります。こうした振舞いは message がなぜ例外が送出されたかを説明するメッセージを格納する場所だという事実を反映することを意図しています。例外に対してより多くのデータを紐付けたい場合は、インスタンスの任意の属性が利用できます。全ての引数は args にもタプルとして格納されるようになっていますが、この属性は廃止の方向に向かっていますのでできるだけ使わないようにする方がいいでしょう。 2.5 で追加された仕様です。

exception Exception

全ての組み込み例外のうち、システム終了でないもの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。 2.5 で変更された仕様: BaseException から導出するように変更されました

exception StandardError

StopIteration、SystemExit、KeyboardInterrupt 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 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

マップ型またはシーケンス型に使ったキーやインデ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。(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 GeneratorExit

ジェネレータの close() メソッドが呼び出されたときに送出されます。この例外は技術的にはエラーでないので StandardError ではなく Exception から導出されています。 2.5 で追加された仕様です。

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

シーケンスのインデクス指定がシーケンスの範囲を超えている場合に送出されます。(スライスのインデクスはシーケンス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。この例外は Exception を捕まえるコードに間違って捕まってインタプリタが終了するのを阻止されないように BaseException から導出されています。 2.5 で変更された仕様: BaseException から導出されるように変更されました

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に適用されます。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フローでは例外を送出するよりもむしろ、オーバ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ので、`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 `evel()` や `input()`、初期化スクリプトの読み込みや標準入力で (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 の作者か、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、`BaseException` 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。

この例外は `Exception` を捕まえるコードに間違って捕まえられないように、`StandardError` や

Exception からではなく BaseException から導出されています。これにより、この例外は着実に呼出し元の方に伝わっていってインタプリタを終了させます。2.5 で変更された仕様: BaseException から導出されるように変更されました。

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かった際に送出されます。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。winerrno 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。errno の値は winerror 値を対応する errno.h の値に対応付けたものです。

OSError のサブクラスです。 2.0 で追加された仕様です。 2.5 で変更された仕様: 以前のバージョンは GetLastError() のコードを errno に入れていました。

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。

exception ImportWarning
モジュールインポートの誤りと思われるものに対する警告の基底クラスです。 2.5 で追加された仕様です。

exception UnicodeWarning
ユニコードに関連した警告の基底クラスです。 2.5 で追加された仕様です。

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:

_hierarchy.txt

2.4 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に、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。

三章

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。**注意:** これまでの (リリース 2.2 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雛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。いまではこのような制限はなくなっています。

主要な組み込み型は数値型、シーケンス型、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、ほぼ全てのオブジェクトについて、比較、真値テスト、(`repr()` 関数や、わずかに異なる `str()` 関数による) 文字列への変換を行うことができます。オブジェクトが `print` によって書かれていると、後の方の文字列への変換が暗黙に行われます (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。)

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`。
- 空のシーケンス型。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。[1]

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)。

3.2 ブール演算 — and, or, not

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。:

[1] 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。

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

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。

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* <= *z* は *x* < *y* および *y* <= *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。

数値型間の比較か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すが任意のものです (従って要素の型が一様でないシーケンス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、シーケンス型でのみ ‘in’ および ‘not in’ がサポートされています (以下を参照)。

3.4 数値型 int, float, long, complex

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、ブール方は通常の整数型のサブタイプです。通常の整数 (単に **整数型** とも呼ばれます) 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 (sys.maxint は常に通常の整数の各プラットフォームにおける最大値にセットされており、最小値は -sys.maxint - 1 になります)。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 ‘11’ は 11 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。[2]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):

[2]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

演算	結果	注釈
x + *y*	*x* と *y* の和	
x - *y*	*x* と *y* の差	
x * *y*	*x* と *y* の積	
x / *y*	*x* と *y* の商	(1)
x // *y*	*x* と *y* の商 (を切り下げたもの)	(5)
x % *y*	*x* / *y* の剰余	(4)
-*x*	*x* の符号反転	
+*x*	*x* の符号不変	
abs(*x*)	*x* の絶対値または大きさ	
int(*x*)	*x* の通常整数への変換	(2)
long(*x*)	*x* の長整数への変換	(2)
float(*x*)	*x* の浮動小数点数への変換	
complex(*re*,*im*)	実数部 *re* 、虚数部 *im* の複素数。 *im* のデフォルト値はゼロ。	
c.conjugate()	複素数 *c* の共役複素数	
divmod(*x*, *y*)	(*x* // *y*, *x* % *y*) からなるペア	(3)
pow(*x*, *y*)	*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が行われるかもしれません; 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。

(5) 整数の除算とも呼ばれます。結果の値は整数ですが、整数型 (int) とは限りません。

3.4.1 整数型におけるビット列演算

通常および長整数型ではさらに、ビット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 | *y*	ビット単位の *x* と *y* の **論理和**	
x ^ *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 << *n*	*x* の *n* ビット左シフト	(1), (2)
x >> *n*	*x* の *n* ビット右シフト	(1), (3)
~*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。

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。後に詳しく述べるシーケンス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的に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。これらは一般的および特殊化されたシーケンス型、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。

3.6 シーケンス型 str, unicode, list, tuple, buffer, xrange

組み込み型には 6 つのシーケンス型があります: 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。

ほとんどのシーケンス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。[3]

以下のテーブルはシーケンス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* は同じ型のシーケンスです; *n*、*i* および *j* は整数です:

演算	結果	注釈
x in *s*	*s* のある要素 *x* と等しい場合 True、そうでない場合 False	(1)
x not in *s*	*s* のある要素が *x* と等しい場合 False、そうでない場合 True	(1)
s + *t*	*s* および *t* の結合	(6)
s * *n*, *n* * *s*	*s* の浅いコピー *n* 個からなる結合	(2)
s[*i*]	*s* の 0 から数えて *i* 番目の要素	(3)
s[*i*:*j*]	*s* の *i* 番目から *j* 番目までのスライス	(3), (4)
s[*i*:*j*:*k*]	*s* の *i* 番目から *j* 番目まで、*k* 毎のスライス	(3), (5)
len(*s*)	*s* の長さ	
min(*s*)	*s* の最小の要素	
max(*s*)	*s* 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* と同じ型の空のシーケンスを表します)。コピーは

[3] 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。

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], [3], [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: 上のコードで、`[[]]` は空のリストを要素として含んでいるリストですから、`[[]] * 3` の 3 つの要素の全てが、空のリスト（への参照）になります。`lists` のいずれかの要素を修正することでこの単一のリストが変更されます。以下のようにすると、異なる個別の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、インデクスは文字列の末端からの相対インデクスになります: `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クス *k* を持つ要素からなるシーケンス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* が省略されるか `None` だった場合、`0` を使います。*j* が省略されるか `None` だった場合、`len(`*s*`)` を使います。*i* が *j* 以上の場合、スライスは空のシーケンスになります。

(5) *s* の *i* 番目から *j* 番目まで *k* 毎のスライスは、$0 \leq n < \frac{j-i}{k}$ となるような、インデクス $x = i + n{\star}k$ を持つ要素からなるシーケンスとして定義されます。言い換えるとインデクスは `i`、`i+k`、`i+2*k`、`i+3*k` などであり、*j* に達したところ (しかし *j* は含みません) でストップします。*i* または *j* が `len(`*s*`)` より大きい場合、`len(`*s*`)` を使います。*i* または *j* を省略するか `None` だった場合、“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。*k* が `None` だった場合、`1` として扱われます。

(6) *s* と *t* の両者が文字列であるとき、CPython のような実装では、*s*=*s*+*t* や *s*+=*t* という書式で代入をするのに in-place optimization が働きます。このような時、最適化は二乗の実行時間の低減をもたらします。この最適化はバージョンや実装に依存します。実行効率が必要なコードでは、バージョンと実装が変わっても、直線的な連結の実行効率を保証する `str.join()` を使うのがより望ましいでしょう。2.4 で変更された仕様: 以前、文字列の連結は in-place で再帰されませんでした

3.6.1 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。

8 ビット文字列では、メソッドはロケール依存になります。

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

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 *fillchar* で指定された値（デフォルトではスペース）が使われます。 2.4 で変更された仕様: 引数 *fillchar* に対応

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 `UnicodeError` を送出します。他に利用できる値は `'ignore'` 、 `'replace'` および関数 `codecs.register_error` によって登録された名前です。これについてはセクション 4.8.1 節を参照してください。 2.2 で追加された仕様です。 2.3 で変更された仕様: その他のエラーハンドリングスキーマがサポートされました

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 `UnicodeError` を送出します。他に利用できる値は `'ignore'` 、 `'replace'` 、 `'xmlcharrefreplace'` 、 `'backslashreplace'` および関数 `codecs.register_error` によって登録された名前です。これについてはセクション 4.8.1 を参照してください。利用可能なエンコーディングの一覧は、セクション 4.8.3 を参照してください。

2.0 で追加された仕様です。 2.3 で変更された仕様: `'xmlcharrefreplace'` 、 `'backslashreplace'` およびその他のエラーハンドリングスキーマがサポートされました

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。*suffix* は見つけたい複数の接尾語のタプルでも構いません。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。

2.5 で変更された仕様: *suffix* でタプルを受け付けるようになりました

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。

8 ビット文字列では、メソッドはロケール依存になります。

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、そうでない場合はを返し

ます。

8 ビット文字列では、メソッドはロケール依存になります。

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。

8 ビット文字列では、メソッドはロケール依存になります。

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。

8 ビット文字列では、メソッドはロケール依存になります。

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。

8 ビット文字列では、メソッドはロケール依存になります。

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、例えば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。

8 ビット文字列では、メソッドはロケール依存になります。

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。

8 ビット文字列では、メソッドはロケール依存になります。

join(*seq*)
シーケンス *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*[, *fillchar*])
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 *fillchar* で指定された文字 (デフォルトではスペース) 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。
2.4 で変更された仕様: 引数 *fillchar* が追加されました

lower()
文字列をコピーし、小文字に変換して返します。

8 ビット文字列では、メソッドはロケール依存になります。

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

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

2.2.2 で変更された仕様: 引数 *chars* をサポートしました

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

の部分、区切り文字列そのもの、そして区切りの後ろの部分です。もし区切れなければ、タプルには元の文字列そのものとその後ろに二つの空文字列が入ります。 2.5 で追加された仕様です。

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*[, *fillchar*])

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 *fillchar* で指定された文字 (デフォルトではスペース）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。 2.4 で変更された仕様: 引数 *fillchar* が追加されました

rpartition(*sep*)

文字列を *sep* の最後の出現位置で区切り、3 要素のタプルを返します。タプルの内容は、区切りの前の部分、区切り文字列そのもの、そして区切りの後ろの部分です。もし区切れなければ、タプルには二つの空文字列とその後ろに元の文字列そのものが入ります。 2.5 で追加された仕様です。

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

sep を区切り文字とした、文字列中の単語のリストを返します。*maxsplit* が与えられた場合、最大で *maxsplit* 個になるように分割が行なわれます、**最も右側**（の単語）は 1 つになります。*sep* が指定されていない、あるいは `None` のとき、全ての空白文字が区切り文字となります。右から分割していくことを除けば、`rsplit()` は後ほど詳しく述べる `split()` と同様に振る舞います。 2.4 で追加された仕様です。

rstrip([*chars*])

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

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

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。(したがって返されるリストは *maxsplit*+1 の要素を持ちます）*maxsplit* が与えられていない場合、無制限に分割が行なわれます（全ての可能な分割が行なわれる)。連続した区切り文字はグループ化されず、空の文字列を区切っていると判断されます (例えば `'1,,2'.split(',')` は `['1', '', '2']` を返します)。引数 *sep* は複数の文字にもできます (例えば `'1, 2, 3'.split(', ')` は `['1', '2', '3']` を返します)。区切り文字を指定して空の文字列を分割すると、`[]` を返します。

sep が指定されていないか `None` が指定されている場合、異なる分割アルゴリズムが適用されます。最初に空白文字（スペース、タブ、改行 (newline)、復帰 (return)、改ページ (formfeed)) が文字列の両端から除去されます。次に任意の長さの空白文字列によって単語に分割されます。連続した空白の区切り文字は単一の区切り文字として扱われます（`'1 2 3'.split()` は `['1', '2', '3']` を

返します)。空の文字列や空白文字だけから成る文字列を分割する場合には空のリストを返します。

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。

8 ビット文字列では、メソッドはロケール依存になります。

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。*prefix* は複数の接頭語のタプルにしても構いません。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。

2.5 で変更された仕様: *prefix* でタプルを受け付けるようになりました

strip([*chars*])

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

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

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。

8 ビット文字列では、メソッドはロケール依存になります。

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*`)` よりも短い場合もと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。

3.6.2 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 (モジュロ) 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。[4] 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:

[4] 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。

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 (h 、 l 、または 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:

変換	意味	注釈
'd'	符号付き 10 進整数。	
'i'	符号付き 10 進整数。	
'o'	符号なし 8 進数。	(1)
'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)。	(3)
'E'	指数表記の浮動小数点数 (大文字)。	(3)
'f'	10 進浮動小数点数。	(3)
'F'	10 進浮動小数点数。	(3)
'g'	浮動小数点数。指数部が -4 以上または精度以下の場合には指数表記、それ以外の場合には 10 進表記。	(4)
'G'	浮動小数点数。指数部が -4 以上または精度以下の場合には指数表記、それ以外の場合には 10 進表記。	(4)
'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)。	(5)
's'	文字列 (python オブジェクトを str() で変換します)。	(6)
'%'	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) この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) この形式にした場合、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) この形式にした場合、変換結果には常に小数点が含まれ、それはその後ろに数字が続かない場合にも適用されます。

指定精度は小数点の後の桁数を決定し、そのデフォルトは 6 です。

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

指定精度は小数点の前後の有効桁数を決定し、そのデフォルトは 6 です。

(5) %r 変換は Python 2.0 で追加されました。

指定精度は最大文字数を決定します。

(6) 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。

指定精度は最大文字数を決定します。

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[5] 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。

3.6.3 XRange 型

xrange 型は値の変更不能なシーケンスで、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かか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。はっ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、インデクス検索、反復、 len() 関数のみをサポートしています。

3.6.4 変更可能なシーケンス型

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。他の変更可能なシーケンス型 (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は変更不可能なシーケンス型です: 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シーケンス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
s[*i*] = *x*	*s* の要素 *s* を *x* と入れ替えます	
s[*i*:*j*] = *t*	*s* の *i* から *j* 番目までのスライスをイテラブル *t* の内容に入れ替えます	
del *s*[*i*:*j*]	*s*[*i*:*j*] = [] と同じです	
s[*i*:*j*:*k*] = *t*	*s*[*i*:*j*:*k*] の要素を *t* と入れ替えます	(1
del *s*[*i*:*j*:*k*]	リストから *s*[*i*:*j*:*k*] の要素を削除します	
s.append(*x*)	*s*[len(*s*):len(*s*)] = [*x*] と同じです	(2
s.extend(*x*)	*s*[len(*s*):len(*s*)] = *x* と同じです	(3
s.count(*x*)	*s*[*i*] == *x* となる *i* の個数を返します	
s.index(*x*[, *i*[, *j*]])	*s*[*k*] == *x* かつ *i* <= *k* < *j* となる最小の *k* を返します。	(4
s.insert(*i*, *x*)	*i* >= 0 の場合の *s*[*i*:*i*] = [*x*] と同じです	(5
s.pop([*i*])	*x* = *s*[*i*]; del *s*[*i*]; return *x* と同じです	(6
s.remove(*x*)	del *s*[*s*.index(*x*)] と同じです	(4
s.reverse()	*s* の値の並びを反転します	(7
s.sort([*cmp*[, *key*[, *reverse*]]])	*s* の要素を並べ替えます	(7), (8), (

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。

[5] 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、際限なく長くて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。

(3) *x* は任意のイテラブル (繰り返し可能オブジェクト) にできます。

(4) *x* が *s* 中に見つからなかった場合 `ValueError` を送出します。　負のインデ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クスが渡された場合、スライスのインデクスと同じく、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。

(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、比較を制御するためにオプションの引数をとります。

cmp は 2 つの引数 (list items) からなるカスタムの比較関数を指定します。これは始めの引数が 2 つ目の引数に比べて小さい、等しい、大きいかに応じて負数、ゼロ、正数を返します。'*cmp*=`lambda x,y: cmp(x.lower(), y.lower())`'

key は 1 つの引数からなる関数を指定します。これは個々のリストの要素から比較のキーを取り出すのに使われます。'*key*=`str.lower`'

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

一般的に、 *key* および *reverse* の変換プロセスは同等の *cmp* 関数を指定するより早く動作します。これは *key* および *reverse* がそれぞれの要素に一度だけ触れる間に、*cmp* はリストのそれぞれの要素に対して複数回呼ばれることによるものです。

2.3 で変更された仕様: `None` を渡すのと、*cmp* を省略した場合とで、同等に扱うサポートを追加

2.4 で変更された仕様: *key* および *reverse* のサポートを追加

(9) Python2.3 以降、`sort()` メソッドは安定していることが保証されています。 ソートは等しいとされた要素の相対オーダーが変更されないことが保証されれば、安定しています — これは複合的なパス（例えば部署ごとにソートして、それを給与の等級）でソートを行なうのに役立ちます。

(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 以降 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。

3.7　set（集合）型 — `set`, `frozenset`

set オブジェクトは順序付けされていない変更不可能な値のコレクションです。よくある使い方には、メンバーシップのテスト、数列から重複を削除する、そして論理積、論理和、差集合、対称差など数学的演算の計算が含まれます。 2.4 で追加された仕様です。

他のコレクションと同様、 sets は *x* `in` *set*、`len(`*set*`)` および `for` *x* `in` *set* をサポートします。順序を持たないコレクションとして、sets は要素の位置と（要素の）挿入位置を保持しません。したがって、sets はインデックス、スライス、その他のシーケンス的な振る舞いをサポートしません。

set および frozenset という、2 つの組み込み set 型があります。set は変更可能な — add() や remove() のようなメソッドを使って内容を変更できます。変更可能なため、ハッシュ値を持たず、また辞書のキーや他の set の要素として用いることができません。frozenset 型は変更不能であり、ハッシュ化可能で — 一度作成されると内容を改変することができません。一方で辞書のキーや他の set の要素として用いることができます。

set および frozenset のインスタンスは、以下の演算を提供します。

Operation	Equivalent	Result
len(*s*)		set *s* の基数
x in *s*		*s* のメンバに *x* があるか調べる
x not in *s*		*s* のメンバに *x* がないか調べる
s.issubset(*t*)	*s* <= *t*	*t* に *s* の全ての要素が含まれるか調べる
s.issuperset(*t*)	*s* >= *t*	*s* に *t* の全ての要素が含まれるか調べる
s.union(*t*)	*s* | *t*	*s* と *t* に含まれるすべての要素を持った新しい set を作成
s.intersection(*t*)	*s* & *t*	*s* と *t* 共通に含まれる要素を持った新しい set を作成
s.difference(*t*)	*s* - *t*	*s* には含まれるが *t* には含まれない要素を持った新しい set を作成
s.symmetric_difference(*t*)	*s* ^ *t*	*s* と *t* のうち、両者には含まれない要素を持った新しい set を作成
s.copy()		*s* の浅いコピーを持った新しい set を作成

注意すべき点として、演算子ではないバージョンのメソッド union()、intersection()、+difference()、symmetric_difference()、issubset() および issuperset() はどの種類の iterable でも引数として受け入れます。対照的に、（それぞれのメソッドに）対応する演算子は引数に sets を要求します。これはより読みやすい set('abc').intersection('cbs') という構文を優先して set('abc') & 'cbs' というような、エラーになりがちな構文を除外します。

set と frozenset の両者とも、sets と sets の比較をサポートしています。もし、あるいは少なくともそれぞれの sets の全ての要素が他の sets に含まれている（それぞれの sets がもう片方のサブセットである）場合、2 つの sets は等しいと言えます。もし、あるいは少なくとも 1 つめの set が 2 つめの set の厳密なサブセットである（サブセットではあるが等しくない）場合、set は他の set より小さいと言えます。もし、あるいは少なくとも 1 つめの set が 2 つめの set の厳密なスーパーセットである（スーパーセットではあるが等しくない）場合、set は他の set より大きいと言えます。

set のインスタンスは frozenset のインスタンスと、そのメンバを基に比較されます。例えば 'set('abc') == frozenset('abc')' は True を返します。

サブセットと同一性の比較は完全な順序付け関数によって一般化されません。例えば、どのような共通部分も持たない 2 つの sets は、等しくもなく、互いのサブセットでもないので、以下のコードの **全て** に False を返します。*a*<*b*、 *a*==*b*、 *a*>*b*。それに応じて、sets は __cmp__ メソッドを実装していません。

sets が部分的な順序付け（サブセットの関係）しか定義していないことから、list.sort() メソッドの結果は不確定の sets のリストとなります。

set の要素は辞書のキーと同様に __hash__ と __eq__ の両方を定義していることが必要です。

set と frozenset のインスタンスを混在させたバイナリ演算は結果を 1 つめのオペランドの型で返します。例えば 'frozenset('ab') | set('bc')' は、frozenset のインスタンスを返します。

以下の表は set で可能なリスト操作です。これらの操作は変更不能な frozenset のインスタンスには適用されません。

Operation	Equivalent	Result
s.update(*t*)	*s* |= *t*	set *s* を *t* の要素を追加して更新します
s.intersection_update(*t*)	*s* &= *t*	set *s* を *s* と *t* の両方に属する要素だけ残すように更新します
s.difference_update(*t*)	*s* -= *t*	set *s* を *t* に属する要素を削除するように更新します
s.symmetric_difference_update(*t*)	*s* ^= *t*	set *s* を *s* か *t* に属するが両方には属さない要素を持つように更
s.add(*x*)		set *s* に要素 *x* を追加します
s.remove(*x*)		set *s* から要素 *x* を削除します。要素が存在しない場合は Key
s.discard(*x*)		set *s* に要素 *x* が存在していれば削除します
s.pop()		*s* から、任意の要素を返してその要素を削除します。空の場合
s.clear()		set *s* から全ての要素を削除します

注意すべき点として、演算子ではないバージョンのメソッド update()、intersection_update()、difference_update() および symmetric_difference_update() は、どんな iterable でも引数として受け入れます。

set 型のデザインは sets で学んだことに基づいています。

参考資料:

Comparison to the built-in set types

sets モジュールと組み込み set 型の違い

3.8 マップ型

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。対応付け自体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
`len(`*a*`)`	*a* 内の要素の数です	
a`[`*k*`]`	キー *k* を持つ *a* の要素です	(1), (1
a`[`*k*`] = `*v*	*a*`[`*k*`]` を *v* に設定します	
`del` *a*`[`*k*`]`	*a* から *a*`[`*k*`]` を削除します	(1)
a`.clear()`	a から全ての要素を削除します	
a`.copy()`	a の (浅い) コピーです	
k `in` *a*	*a* にキー *k* があれば `True` 、そうでなければ `False` です	(2)
k `not in` *a*	`not` *k* `in` *a* と同じです	(2)
a`.has_key(`*k*`)`	*k* `in` *a* と同じなので、新しく書くコードではその形を使ってください	
a`.items()`	*a* における (*key*, *value*) ペアのリストのコピーです	(3)
a`.keys()`	*a* におけるキーのリストのコピーです	(3)
a`.update(`[*b*]`)`	*b* によって key/value ペアを更新（上書き）	(9)
a`.fromkeys(`*seq*[`, `*value*]`)`	*seq* からキーを作り、値が *value* であるような、新しい辞書を作成します	(7)
a`.values()`	*a* における値のリストのコピーです	(3)
a`.get(`*k*[`, `*x*]`)`	もし *k* `in` *a* なら *a*`[`*k*`]`、そうでなければ *x* を返します	(4)
a`.setdefault(`*k*[`, `*x*]`)`	もし *k* `in` *a* なら *a*`[`*k*`]`、そうでなければ *x* (が与えられていた場合) を返します	(5)
a`.pop(`*k*[`, `*x*]`)`	もし *k* `in` *a* なら *a*`[`*k*`]` 、そうでなければ *x* を返して k を除去します	(8)
a`.popitem()`	任意の (*key*, *value*) ペアを除去して返します	(6)
a`.iteritems()`	(*key*, *value*) ペアにわたるイテレータを返します	(2), (3
a`.iterkeys()`	マップのキー列にわたるイテレータを返します	(2), (3
a`.itervalues()`	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) キーおよび値は任意の順序でリスト化されています。この順序はランダムではなく、Python の実装によって異なり、辞書の挿入、削除の履歴に依存します。`items()`、`keys()`、`values()`、`iteritems()`、`iterkeys()` および `itervalues()` が途中で辞書を変更せずに呼ばれた場合、リストも直接対応するでしょう。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。デフォルトで *x* は *None* です。

(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。もし辞書が空なら `popitem()` の呼び出しは `KeyError` の送出を引き起こします。

(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。

(9) `update()` はその他のマッピングオブジェクトや反復可能なキー/値のペア（タプルやその他 2 つの要素を持つ反復可能な要素）を受け入れます。キーワードとなる引数が指定されている場合、マッピングはそれらのキー/値のペアで更新されます。'`d.update(red=1, blue=2)`' 2.4 で変更された仕様: キー／値のペアでできたイテレーション可能オブジェクトを引数に取るようになりました。また、キーワード引数をとるようになりました。

(10) dict のサブクラスが `__missing__` メソッドを定義しているならば、キー *k* が無ければ *a*[*k*] は *k* を引数にそのメソッドを呼び出します。したがってキーが無いときに *a*[*k*] が結果を返すのも例外を送出するのも、`__missing__`(*k*) が結果を返すか例外を送出するかで決まります。他のどんなメソッドも演算も`__missing__`() を呼び出すことはありません。このような`__missing__` が定義されていなければ、`KeyError` が送出されます。`__missing__` はメソッドでなければならず、インスタンス変数では駄目です。例として `collections.defaultdict` を見てください。 2.5 で追加された仕様です。

3.9 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。[6] 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。

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

```
from __future__ import with_statement

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

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

```
f = open("hello.txt")
try:
    for line in f:
        print line
finally:
    f.close()
```

[6] `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。

注意: 全ての Python の “ファイル的” 型が `with` 文用のコンテキスト・マネージャとして使えるわけではありません。もし、全てのファイル的オブジェクトで動くようにコードを書きたいのならば、オブジェクトを直接使うのではなく `contextlib` にある `closing()` を使うと良いでしょう。詳細はセクション 26.5 を参照してください。

`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おすと、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（ですが、ファイルが不完全な行で終わっている場合は何も残らないかもしれません)。[7]引数 *size* が指定されていて負数でない場合、(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。

[7]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。

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で (`'b'` なしで) 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。

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

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*)

文字列からなるシーケンスをファイルに書き込みます。シーケンス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)

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 **configure** に指定された場合（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ず、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。

3.10 コンテキストマネージャ型

2.5 で追加された仕様です。

Python の `with` 文はコンテキストマネージャによって定義される実行時コンテキストの概念をサポートします。これは、ユーザ定義クラスが文の本体が実行される前に進入し文の終わりで脱出する実行時コンテキストを定義することを許す二つの別々のメソッドを使って実装されます。

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

__enter__`()`
: 実行時コンテキストに入り、このオブジェクトまたは他の実行時コンテキストに関連したオブジェク

トを返します。このメソッドが返す値はこのコンテキストマネージャを使う with 文の as 節の識別子に束縛されます。

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

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

__exit__(*exc_type, exc_val, exc_tb*)

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

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

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

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

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

これらのメソッドのために Python/C API の中の Python オブジェクトの型構造体に特別なスロットが作られたわけではないことに注意してください。これらのメソッドを定義したい拡張型については通常の Python からアクセスできるメソッドとして提供しなければなりません。実行時コンテキストを準備することに比べたら、一つのクラスの辞書引きは無視できるオーバーヘッドです。

3.11 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。

3.11.1 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、厳密にいえば、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すが、*m*.__dict__ = {} と書くことはできません)。 __dict__ を直接編集するのは推奨されません。

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/python2.5/os.pyc'> と書かれます。

3.11.2 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。

3.11.3 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。

より詳しい情報は *Python* **リファレンスマニュアル** を参照してください。

3.11.4 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) のインスタン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。

3.11.5 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。

3.11.6 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。

3.11.7 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。

3.11.8 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。

3.11.9 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。

3.11.10 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。

3.12 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。

四章

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。

string	一般的な文字列操作
re	Perl 風のシンタクスを用いた正規表現検索とマッチ操作。
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します。
difflib	オブジェクト同士の違いを計算する
StringIO	ファイルのように文字列を読み書きする。
cStringIO	StringIO を高速にしたものだが、サブクラス化はできない。
textwrap	テキストの折り返しと詰め込み
encodings.utf-8-sig	UTF-8 codec with BOM signature
unicodedata	Access the Unicode Database.
stringprep	RFC 3453 による文字列調製
fpformat	浮動小数点をフォーマットする汎用関数。

string オブジェクトのメソッドについては、3.6.1 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

string モジュールには便利な定数やクラスが数多く入っています。また、現在は文字列のメソッドとして利用できる、すでに撤廃された古い関数も入っています。正規表現に関する文字列操作の関数は re を参照してください。

4.1.1 文字列定数

このモジュールでは以下の定数を定義しています。

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、固定です。

ascii_uppercase
: 大文字 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'。この値はロケールに依存せず、固定です。

digits
: 文字列 '0123456789' です。

hexdigits
: 文字列 '0123456789abcdefABCDEF' です。

letters
後述の lowercase と uppercase を合わせた文字列です。具体的な値はロケールに依存しており、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です。ほとんどのシステムでは文字列 'abcdefghijklmnopqrstuvwxyz' です。この定義を変更してはなりません — 変更した場合の upper() と swapcase() に対する影響は定義されていません。具体的な値はロケールに依存しており、locale.setlocale() が呼ばれたときに更新されます。

octdigits
文字列 '01234567' です。

punctuation
'C' ロケールにおいて、句読点として扱われる ASCII 文字の文字列です。

printable
印刷可能な文字で構成される文字列です。digits、letters、punctuation および whitespace を組み合わせたものです。

uppercase
大文字として扱われる文字全てを含む文字列です。ほとんどのシステムでは 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' です。この定義を変更してはなりません—- 変更した場合の lower() と swapcase() に対する影響は定義されていません。具体的な値はロケールに依存しており、locale.setlocale() が呼ばれたときに更新されます。

whitespace
空白 (whitespace) として扱われる文字全てを含む文字列です。ほとんどのシステムでは、これはスペース (space)、タブ (tab)、改行 (linefeed)、復帰 (return)、改頁 (formfeed)、垂直タブ (vertical tab) です。この定義を変更してはなりません — 変更した場合の strip() と split() に対する影響は定義されていません。

4.1.2 テンプレート文字列

テンプレート (template) を使うと、PEP 292 で解説されているようにより簡潔に文字列置換 (string substitution) を行えるようになります。通常の '%' ベースの置換に代わって、テンプレートでは以下のような規則に従った '$' ベースの置換をサポートしています:

- '$$' はエスケープ文字です; '$' 一つに置換されます。
- '$identifier' は置換プレースホルダの指定で、 "identifier" というキーへの対応付けに相当します。デフォルトは、"identifier" の部分には Python の識別子が書かれていなければなりません。'$' の後に識別子に使えない文字が出現すると、そこでプレースホルダ名の指定が終わります。
- '${identifier}' は '$identifier' と同じです。プレースホルダ名の後ろに識別子として使える文字列が続いていて、それをプレースホルダ名の一部として扱いたくない場合、例えば"${noun}ification" のような場合に必要な書き方です。

上記以外の書き方で文字列中に '$' を使うと ValueError を送出します。

2.4 で追加された仕様です。

string モジュールでは、上記のような規則を実装した Template クラスを提供しています。Template のメソッドを以下に示します:

class Template(*template*)

コンストラクタはテンプレート文字列になる引数を一つだけ取ります。

substitute(*mapping*[, ***kws*])

テンプレート置換を行い、新たな文字列を生成して返します。*mapping* はテンプレート中のプレースホルダに対応するキーを持つような任意の辞書類似オブジェクトです。辞書を指定する代わりに、キーワード引数も指定でき、その場合にはキーワードをプレースホルダ名に対応させます。*mapping* と *kws* の両方が指定され、内容が重複した場合には、*kws* に指定したプレースホルダを優先します。

safe_substitute(*mapping*[, ***kws*])

`substitute()` と同じですが、プレースホルダに対応するものを *mapping* や *kws* から見つけられなかった場合に、`KeyError` 例外を送出する代わりにもとのプレースホルダがそのまま入ります。また、`substitute()` とは違い、規則外の書き方で '`$`' を使った場合でも、`ValueError` を送出せず単に '`$`' を返します。

その他の例外も発生しうる一方で、このメソッドが「安全 (safe)」と呼ばれているのは、置換操作が常に例外を送出する代わりに利用可能な文字列を返そうとしているからです。別の見方をすれば、`safe_substitute()` は区切り間違いによるぶら下がり (dangling delimiter) や波括弧の非対応、Python の識別子として無効なプレースホルダ名を含むような不正なテンプレートを何も警告せずに無視するため、安全とはいえないのです。

`Template` のインスタンスは、次のような public な属性を提供しています:

template

コンストラクタの引数 *template* に渡されたオブジェクトです。通常、この値を変更すべきではありませんが、読み込み専用アクセスを強制しているわけではありません。

Template の使い方の例を以下に示します:

```
>>> from string import Template
>>> s = Template('$who likes $what')
>>> s.substitute(who='tim', what='kung pao')
'tim likes kung pao'
>>> d = dict(who='tim')
>>> Template('Give $who $100').substitute(d)
Traceback (most recent call last):
[...]
ValueError: Invalid placeholder in string: line 1, col 10
>>> Template('$who likes $what').substitute(d)
Traceback (most recent call last):
[...]
KeyError: 'what'
>>> Template('$who likes $what').safe_substitute(d)
'tim likes $what'
```

さらに進んだ使い方: `Template` のサブクラスを導出して、プレースホルダの書式、区切り文字、テンプレート文字列の解釈に使われている正規表現全体をカスタマイズできます。こうした作業には、以下のクラス属性をオーバライドします:

- *delimiter* – プレースホルダの開始を示すリテラル文字列です。デフォルトの値は '`$`' です。実装系はこの文字列に対して必要に応じて `re.escape()` を呼び出すので、正規表現を表すような文字列にしては **なりません**。
- *idpattern* – 波括弧でくくらない形式のプレースホルダの表記パターンを示す正規表現です (波括弧は自動的に適切な場所に追加されます)。で尾フォルトの値は '`[_a-z][_a-z0-9]*`' という正規表現です。

他にも、クラス属性 *pattern* をオーバライドして、正規表現パターン全体を指定できます。オーバライドを行う場合、*pattern* の値は 4 つの名前つきキャプチャグループ (capturing group) を持った正規表現オブジェクトでなければなりません。これらのキャプチャグループは、上で説明した規則と、無効なプレースホルダに対する規則に対応しています:

- *escaped* – このグループはエスケープシーケンス、すなわちデフォルトパターンにおける '`$$`' に対応します。
- *named* – このグループは波括弧でくくらないプレースホルダ名に対応します; キャプチャグループに区切り文字を含めてはなりません。
- *braced* – このグループは波括弧でくくったプレースホルダ名に対応します; キャプチャグループに区切り文字を含めてはなりません。
- *invalid* – このグループはそのほかの区切り文字のパターン (通常は区切り文字一つ) に対応し、正規表現の末尾に出現せねばなりません。

4.1.3 文字列操作関数

以下の関数は文字列または Unicode オブジェクトを操作できます。これらの関数は文字列型のメソッドにはありません。

capwords(*s*)

`split()` を使って引数を単語に分割し、`capitalize()` を使ってそれぞれの単語の先頭の文字を大文字に変換し、 `join()` を使ってつなぎ合わせます。この置換処理は文字列中の連続する空白文字をスペース一つに置き換え、先頭と末尾の空白を削除するので注意してください。

maketrans(*from, to*)

`translate()` や `regex.compile()` に渡すのに適した変換テーブルを返します。このテーブルは、 *from* 内の各文字を *to* の同じ位置にある文字に対応付けます; *from* と *to* は同じ長さでなければなりません。

警告: `lowercase` と `uppercase` から取り出した文字列を引数に使ってはなりません; ロケールによっては、これらは同じ長さになりません。大文字小文字の変換には、常に `lower()` または `upper()` を使ってください。

4.1.4 撤廃された文字列関数

以下の一連の関数は、文字列型や Unicode 型のオブジェクトのメソッドとしても定義されています; 詳しくは "文字列型のメソッド" (3.6.1) を参照してください。ここに挙げた関数は Python 3.0 で削除されることはないはずですが、撤廃された関数とみなして下さい。このモジュールで定義されている関数は以下の通りです:

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()` を使ってください。

文字列を浮動小数点型の数値に変換します。文字列は Python における標準的なの浮動小数点リテラルの文法に従っていなければなりません。先頭に符号 ('+' または '-') が付くのは構いません。この関数に文字列を渡した場合は、組み込み関数 `float()` と同じように振舞います。

注意: 文字列を渡した場合、根底にある C ライブラリによって NaN や Infinity を返す場合があります。こうした値を返させるのがどんな文字列の集合であるかは、全て C ライブラリに依存しており、ライブラリによって異なると知られています。

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()` を使ってください。

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 桁またはそれ以上の数字からなっていなければなりません。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 です。 *base* が 0 の場合、(符号を剥ぎ取った後の) 文字列の先頭にある文字列に従ってデフォルトの基数を決定します。'`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`' が付いていても受け付けますが、必須ではありません。文字列を渡す場合、この関数は組み込み関数 `int()` と同じように振舞います。 (数値リテラルをより柔軟に解釈したい場合には、組み込み関数 `eval()` を使ってください。)

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()` を使ってください。

文字列 *s* を、*base* を基数とする長整数に変換します。文字列は 1 桁またはそれ以上の数字からなっていなければなりません。先頭に符号 ('+' または '−') が付くのは構いません。*base* は `atoi()` と同じ意味です。基数が 0 の場合を除き、文字列末尾に '`l`' や '`L`' を付けてはなりません。*base* を指定しないか、10 を指定して文字列を渡した場合には、この関数は組み込み関数 `long()` と同じように振舞います。

capitalize (*word*)

先頭文字だけ大文字にした *word* のコピーを返します。

expandtabs (*s*[, *tabsize*])

現在のカラムと指定タブ幅に従って文字列中のタブを展開し、一つまたはそれ以上のスペースに置き換えます。文字列中に改行が出現するたびにカラム番号は 0 にリセットされます。この関数は、他の非表示文字やエスケープシーケンスを解釈しません。タブ幅のデフォルトは 8 です。

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

s`[`*start*`:`*end*`]` の中で、部分文字列 *sub* が完全な形で入っている場所のうち、最初のものを *s* のインデクスで返します。見つからなかった場合は `-1` を返します。*start* と *end* のデフォルト値、および、負の値を指定した場合の解釈は文字列のスライスと同じです。

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` を送出します。

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` 送出します。

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

s`[`*start*`:`*end*`]` における、部分文字列 *sub* の (重複しない) 出現回数を返します。*start* と *end* のデフォルト値、および、負の値を指定した場合の解釈は文字列のスライスと同じです。

lower (*s*)

s のコピーを大文字を小文字に変換して返します。

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

文字列 *s* 内の単語からなるリストを返します。オプションの第二引数 *sep* を指定しないか、または `None` にした場合、空白文字 (スペース、タブ、改行、リターン、改頁) からなる任意の文字列で単語に区切ります。*sep* を `None` 以外の値に指定した場合、単語の分割に使う文字列の指定になります。戻り値のリストには、文字列中に分割文字列が重複せずに出現する回数より一つ多い要素が入るはずです。オプションの第三引数 *maxsplit* はデフォルトで 0 です。この値がゼロでない場合、最大でも *maxsplit* 回の分割しか行わず、リストの最後の要素は未分割の残りの文字列になります (従って、リ

スト中の要素数は最大でも *maxsplit*+1 です)。

空文字列に対する分割を行った場合の挙動は *sep* の値に依存します。*sep* を指定しないか None にした場合、結果は空のリストになります。 *sep* に文字列を指定した場合、空文字列一つの入ったリストになります。

rsplit(*s*[, *sep*[, *maxsplit*]])

s 中の単語からなるリストを *s* の末尾から検索して生成し返します。関数の返す語のリストは全ての点で split() の返すものと同じになります。ただし、オプションの第三引数 *maxsplit* をゼロでない値に指定した場合には必ずしも同じにはなりません。*maxsplit* がゼロでない場合には、最大で *maxsplit* 個の分割を **右端から** 行います - 未分割の残りの文字列はリストの最初の要素として返されます (従って、リスト中の要素数は最大でも *maxsplit*+1 です)。 2.4 で追加された仕様です。

splitfields(*s*[, *sep*[, *maxsplit*]])

この関数は split() と同じように振舞います。 (以前は split() は単一引数の場合にのみ使い、splitfields() は引数 2 つの場合でのみ使っていました)。

join(*words*[, *sep*])

単語のリストやタプルを間に *sep* を入れて連結します。*sep* のデフォルト値はスペース文字 1 つです。'string.join(string.split(*s*, *sep*), *sep*)' は常に *s* になります。

joinfields(*words*[, *sep*])

この関数は join() と同じふるまいをします (以前は、join() を使えるのは引数が 1 つの場合だけで、joinfields() は引数 2 つの場合だけでした)。文字列オブジェクトには joinfields() メソッドがないので注意してください。代わりに join() メソッドを使ってください。

lstrip(*s*[, *chars*])

文字列の先頭から文字を取り除いたコピーを生成して返します。*chars* を指定しない場合や None にした場合、先頭の空白を取り除きます。*chars* を None 以外の値にする場合、*chars* は文字列でなければなりません。 2.2.3 で変更された仕様: *chars* パラメタを追加しました。初期の 2.2 バージョンでは、*chars* パラメータを渡せませんでした

rstrip(*s*[, *chars*])

文字列の末尾から文字を取り除いたコピーを生成して返します。*chars* を指定しない場合や None にした場合、末尾の空白を取り除きます。*chars* を None 以外の値にする場合、*chars* は文字列でなければなりません。 2.2.3 で変更された仕様: *chars* パラメタを追加しました。初期の 2.2 バージョンでは、*chars* パラメータを渡せませんでした

strip(*s*[, *chars*])

文字列の先頭と末尾から文字を取り除いたコピーを生成して返します。*chars* を指定しない場合や None にした場合、先頭と末尾の空白を取り除きます。*chars* を None 以外に指定する場合、*chars* は文字列でなければなりません。 2.2.3 で変更された仕様: *chars* パラメタを追加しました。初期の 2.2 バージョンでは、*chars* パラメータを渡せませんでした

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* に入っている文字を削除し、*table* を使って文字変換を行って返します。*table* は 256 文字からなる文字列で、各文字はそのインデクスを序数とする文字に対する変換先の文字の指定になります。

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*)

文字列を指定した文字幅のフィールド中でそれぞれ左寄せ、右寄せ、中央寄せします。これらの関数は指定幅になるまで文字列 *s* の左側、右側、および両側のいずれかにスペースを追加して、少なくとも *width* 文字からなる文字列にして返します。文字列を切り詰めることはありません。

zfill(*s, width*)

数値を表現する文字列の左側に、指定の幅になるまでゼロを付加します。符号付きの数字も正しく処理します。

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* を指定した場合、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

このモジュールでは、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ています。正規表現のパターン文字列にはヌルバイトを含められませんが、*number* 記法を使えばヌルバイトを指定できます。パターンと検索対象文字列の両方について、 8 ビット文字列と Unicode 文字列を同じように扱えます。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、特殊文字の持つ特別な意味を呼び出さずにその特殊な文字を使えるようにするために、バックスラッシュ文字 ('\') を使います。こうしたバックスラッシュの使い方は、Python の文字列リテラルにおける同じバックスラッシュ文字と衝突を起こします。例えば、バックスラッシュ自体にマッチさせるには、パターン文字列として'\\\\' と書かなければなりません、というのも、正規表現は '\\' でなければならず、さらに正規な Python 文字列リテラルでは各々のバックスラッシュを '\\' と表現せねばならないからです。

正規表現パターンに Python の raw string 記法を使えばこの問題を解決できます。'r' を前置した文字列リテラル内ではバックスラッシュを特別扱いしません。従って、"\n" が改行一文字の入った文字列になるのに対して、r"\n" は '\' と 'n' という二つの文字の入った文字列になります。通常、 Python コード中では、パターンをこの raw string 記法を使って表現します。

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 Pyhon 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くわしく説明しています。

4.2.1 正規表現のシンタクス

正規表現 (すなわち RE) は、表現にマッチ (match) する文字列の集合を表しています。このモジュールの関数を使えば、ある文字列が指定の正規表現にマッチするか (または指定の正規表現がある文字列にマッチするか、つまりは同じことですが) を検査できます。

正規表現を連結すると新しい正規表現を作れます。*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、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* と *B* との間に境界条件がある場合や、番号付けされたグループ参照のような、優先度の低い演算を *A* や *B* が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、コンパイラの構築に関する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られます。'A'、'a'、あるいは '0' のようなほとんどの通常文字は最も簡単な正規表現になります。こうした文字は、単純にその文字自体にマッチします。通常の文字は連結できるので、「last」は文字列'last'とマッチします。(この節の以降の説明では、正規表現を引用符を使わずに「この表示スタイル: special style」で書き、マッチ対象の文字列は、'引用符で括って'書きます。)

'|' や '(' といったいくつかの文字は特殊文字です。特殊文字は通常の文字の種別を表したり、あるいは特殊文字の周辺にある通常の文字に対する解釈方法に影響します。

特殊文字を以下に示します:

'.' (ドット) デフォルトのモードでは改行以外の任意の文字にマッチします。DOTALL フラグが指定されていれば改行も含むすべての文字にマッチします。

'^' (キャレット) 文字列の先頭とマッチします。MULTILINE モードでは各改行の直後にマッチします。

**'$'** 文字列の末尾、あるいは文字列の末尾の改行の直前にマッチします。例えば、「foo」は 'foo' と 'foobar' の両方にマッチします。一方、正規表現「foo$」は 'foo' だけとマッチします。興味深いことに、'foo1\nfoo2\n' を「foo.$」で検索した場合、通常のモードでは 'foo2' だけにマッチし、MULTILINE モードでは 'foo1' にもマッチします。

'*' 直前にある RE に作用して、 RE を 0 回以上できるだけ多く繰り返したものにマッチさせるようにします。例えば「ab*」は 'a'、'ab'、あるいは'a' に任意個数の'b' を続けたものにマッチします。

'+' 直前にある RE に作用して、 RE を、1 回以上繰り返したものにマッチさせるようにします。例えば「ab+」は 'a' に一つ以上の 'b' が続いたものにマッチし、 'a' 単体にはマッチしません。

'?' 直前にある RE に作用して、 RE を 0 回か 1 回繰り返したものにマッチさせるようにします。例えば「ab?」は 'a' あるいは 'ab' にマッチします。

***?, +?, ??** '*'、'+'、'?' といった修飾子は、すべて**貪欲** *(greedy)* マッチ、すなわちできるだけ多くのテキストにマッチするようになっています。時にはこの動作が望ましくない場合もあります。例えば正規表現「<.*>」を'<H1>title</H1>'にマッチさせると、'<H1>'だけにマッチするのではなく全文字列にマッチしてしまいます。'?' を修飾子の後に追加すると、**非貪欲** *(non-greedy)* あるいは **最小一致** *(minimal)* のマッチになり、できるだけ **少ない**文字数のマッチになります。例えば上の式で「.*?」を使うと'<H1>'だけにマッチします。

{*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クスがどういう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。

(?(*id/name*)yes-pattern|no-pattern) グループに *id* が与えられている、もしくは *name* があるとき、「yes-pattern」とマッチします。存在しないときには「no-pattern」とマッチします。「|no-pattern」はオプションで省略できます。例えば「(<)?(\w+@\w+(?:\.\w+)+)(?(1)>)」は email アドレスとマッチする最低限のパターンです。これは '<user@host.com>' や 'user@host.com' にはマッチしますが、'<user@host.com' にはマッチしません。2.4 で追加された仕様です。

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るだけです。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、英数字であると見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 UNICODE フラグが指定されていない場合、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。UNICODE がある場合、Unicode 文字特性データベースで数字と分類されているものにマッチします。

\D UNICODE フラグが指定されていない場合、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。UNICODE がある場合、これは Unicode 文字特性データベースで数字とマーク付けされている文字以外にマッチします。

\s LOCALE と UNICODE フラグが指定されていない場合、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。

LOCALE がある場合、これはこの集合に加えて現在のロケールで空白と定義されている全てにマッチします。UNICODE が設定されると、これは「[\t\n\r\f\v]」と Unicode 文字特性データベースで空白と分類されている全てにマッチします。

\S LOCALE と UNICDOE がフラグが指定されていない場合、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。LOCALE がある場合、これはこの集合に無い文字と、現在のロケールで空白と定義されていない文字にマッチします。UNICODE が設定されていると、「[\t\n\r\f\v]」でない文字と、Unicode 文字特性データベースで空白とマーク付けされていないものにマッチします。

\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：

```
\a      \b      \f      \n
\r      \t      \v      \x
\\
```

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。文字列リテラルについて、8 進エスケープはほとんどの場合 3 桁長になります。

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拘らず、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' は先頭にない
re.compile("^a").search("\na", 1)        # 失敗； 'a' は先頭にない
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

このモジュールは幾つかの関数、定数、例外を定義します。この関数のいくつかはコンパイル済み正規表現向けの完全版のメソッドを簡略化したバージョンです。それなりのアプリケーションのほとんどで、コンパイルされた形式が用いられるのが普通です。

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」、「\s」と「\S」を、現在のロケールに従わさせます。

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 '$' は文字列の末尾および各行の末尾 (改行の直前) とマッチします。デフォールトでは、'^' は、文字列の先頭とだけマッチし、'$' 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」、「\B」、「\d」、「\D」、「\s」と「\S」を、Unicode 文字特性データベースに従わさせます。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.']
```

findall (*pattern, string*[, *flags*])

pattern の *string* へのマッチのうち、重複しない全てのマッチからなるリストを返します。パターン中に何らかのグループがある場合、グループのリストを返します。グループが複数定義されていた場合、タプルのリストになります。他のマッチの開始部分に接触しないかぎり、空のマッチも結果に含められます。 1.5.2 で追加された仕様です。 2.4 で変更された仕様: オプションの flags 引数を追加しました

finditer (*pattern, string*[, *flags*])

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。 2.4 で変更された仕様: Added the optional flags argument

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*\(\s*\):',
...         r'static PyObject*\npy_\1(void)\n{',
...         'def myfunc():')
'static PyObject*\npy_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が、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。もし文字列がパタンー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；デフォールトでは 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*[, *pos*[, *endpos*]])

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*[, *pos*[, *endpos*]])

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* はデフォールトでゼロです (マッチしたも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；それはデフォールトでは 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 **名前つきの**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；それはデフォールトでは 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、デフォールトでは (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*) と等しくなろ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 例外を発生します。

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* はデフォールトでゼロです。

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。例えば、「(a)b」、「((a)(b))」や「((ab))」といった表現が 'ab' に適用された場合、lastindex == 1 となり、同じ文字列に「(a)(b)」が適用された場合には lastindex == 2 となるでしょう。

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() に相当するものがありません。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。

scanf() トークン	正規表現
%c	「.」
%5c	「.{5}」
%d	「[-+]?\d+」
%e, %E, %f, %g	「[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?」
%i	「[-+]?(0[xX][\dA-Fa-f]+|0[0-7]*|\d+)」
%o	「0[0-7]*」
%s	「\S+」
%u	「\d+」
%x, %X	「0[xX][\dA-Fa-f]+」

```
/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/python2.5/re.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 calcsize (*fmt*) と一致しなければなりません)。

calcsize (*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。文字列をパックする際には、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際には、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 (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	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 (= 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、'@' であるとみなされます。

ネイティブのバイトオーダ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のようになります: 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ですが、後者のバイトサイズやバイトオーダ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に設定するか、'@' バイトオーダ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、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 'llh0l' 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 (5.6 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 (9.5 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。

実行時間: 基本的な Ratcliff-Obershelp アルゴリズムは、最悪の場合 3 乗、期待値でも 2 乗となります。`SequenceMatcher` オブジェクトは、最悪のケースで 4 乗、期待値はシーケンスの中の要素数に非常にややこしく依存しています。最良の場合は線形時間になります。

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ち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。

class `HtmlDiff`

このクラスは、二つのテキストを左右に並べて比較表示し、行間あるいは行内の変更点を強調表示するような HTML テーブル (またはテーブルの入った完全な HTML ファイル) を生成するために使います。テーブルは完全差分モード、コンテキスト差分モードのいずれでも生成できます。

このクラスのコンストラクタは以下のようになっています:

`__init__`(*tabsize*][, *wrapcolumn*][, *linejunk*][, *charjunk*])

`HtmlDiff` のインスタンスを初期化します。

tabsize はオプションのキーワード引数で、タブストップ幅を指定します。デフォルトは `8` です。

wrapcolumn はオプションのキーワード引数で、テキストを折り返すカラム幅を指定します。デフォルトは `None` で折り返しを行いません。

linejunk および *charjunk* はオプションのキーワード引数で、`ndiff()` (`HtmlDiff` はこの関数を使って左右のテキストの差分を HTML で生成します) に渡されます。それぞれの引数のデフォルト値および説明は `ndiff()` のドキュメントを参照してください。

以下のメソッドが public になっています:

`make_file`(*fromlines, tolines* [, *fromdesc*][, *todesc*][, *context*][, *numlines*])

fromlines と *tolines* (いずれも文字列のリスト) を比較し、行間または行内の変更点が強調表示された行差分の入った表を持つ完全な HTML ファイルを文字列で返します。

fromdesc および *todesc* はオプションのキーワード引数で、差分表示テーブルにおけるそれぞれ差分元、差分先ファイルのカラムのヘッダになる文字列を指定します (いずれもデフォルト値は空文字列です)。

context および *numlines* はともにオプションのキーワード引数です。*contest* を `True` にするとコンテキスト差分を表示し、デフォルトの `False` にすると完全なファイル差分を表示します。*numlines* のデフォルト値は `5` で、*context* が `True` の場合、 *numlines* は強調部分の前後にあるコンテキスト行の数を制御します。*context* が `False` の場合、*numlines* は "next" と書かれたハイパーリンクをたどった時に到達する場所が次の変更部分より何行前にあるかを制御します (値をゼロにした場合、"next" ハイパーリンクを辿ると変更部分の強調表示がブラウザの最上部に表示されるようになります)。

make_table(*fromlines, tolines* [, *fromdesc*][, *todesc*][, *context*][, *numlines*])

fromlines と *tolines* (いずれも文字列のリスト) を比較し、行間または行内の変更点が強調表示された行差分の入った完全な HTML テーブルを文字列で返します。

このメソッドの引数は、`make_file()` メソッドの引数と同じです。

'Tools/scripts/diff.py' はこのクラスへのコマンドラインフロントエンドで、使い方を学ぶ上で格好の例題が入っています。

2.4 で追加された仕様です。

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 のコンテクスト形式で返します。

コンテクスト形式は、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。コンテクスト（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 コンテクスト形式は、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。コンテクスト（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() で得た入力を処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 コンテクスト形式は、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）**
(http://www.ddj.com/documents/s=1103/ddj8807c/)

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. Dobb'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、None (デフォルトの値です) にするか、単一の引数をとる関数にせねばなりません。後者の場合、関数はシーケンスの要素を受け取り、要素が "junk" であり、無視すべきである場合に限り真をかえすようにせねばなりません。*isjunk* に None を渡すと、lambda x: 0 を渡したのと同じになります; すなわち、いかなる要素も無視しなくなります。例えば以下のような引数を渡すと、空白とタブ文字を無視して文字のシーケンスを比較します。

```
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づつ比較するために、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* >= *k'*, *i* <= *i'* が *i* == *i'*, *j* <= *j'* でも同様です。言い換えると、いくつも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*] 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、(len(*a*), 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。

もし (*i*, *j*, *n*) と (*i'*, *j'*, *n'*) がリストで並んでいる 3 つ組で、2 つ目が最後の 3 つ組でなければ、*i*+*n* != *i'* または *j*+*n* != *j'* です。言い換えると並んでいる 3 つ組は常に隣接していない同じブロックを表しています。 2.5 で変更された仕様: 隣接する 3 つ組は常に隣接しないブロックを表すと保証するようになりました

```
>>> 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* 行までのコンテク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、どこれ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 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 3.9)。

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。どちらの場合でも最初のファイル位置は 0 から始まります。

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 つを混ぜることには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。

使用例:

```
import StringIO

output = StringIO.StringIO()
output.write('First line.\n')
print >>output, 'Second line.'

# ファイルの内容を取り出す -- ここでは
# 'First line.\nSecond line.\n'
contents = output.getvalue()

# オブジェクトを閉じてメモリバッファを解放する --
# .getvalue() は例外を送出するようになる。
output.close()
```

4.6 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、プレイン 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。これらのオブジェクトは普段は見えません。トレースバックに StringI と StringO として表示されます。

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すに 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。

使用例:

```
import cStringIO

output = cStringIO.StringIO()
output.write('First line.\n')
print >>output, 'Second line.'

# ファイルの内容を取り出す -- ここでは
# 'First line.\nSecond line.\n'
contents = output.getvalue()

# オブジェクトを閉じてメモリバッファを解放する --
# 以降 .getvalue() は例外を送出するようになる。
output.close()
```

4.7 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、共通して現れる先頭の空白を削除します。

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は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。

タブとスペースはともにホワイトスペースとして扱われますが、同じではないことに注意してください: " hello" という行と "\thello" は、同じ先頭の空白文字をもっていないとみなされます。

(このふるまいは Python 2.5 で導入されました。古いバージョンではこのモジュールは不正にタブを展開して共通の先頭空白文字列を探していました)

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。

4.8 `codecs` — codec レジストリと基底クラス

このモジュールでは、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。codec レジストリは、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、以下の属性を持つ `CodecInfo` オブジェクトを返します。

- `name` エンコーディング名
- `encoder` 内部状態を持たないエンコード関数
- `decoder` 内部状態を持たないデコード関数
- `incrementalencoder` 漸増的エンコーダクラスまたはファクトリ関数
- `incrementaldecoder` 漸増的デコーダクラスまたはファクトリ関数
- `streamwriter` ストリームライタクラスまたはファクトリ関数
- `streamreader` ストリームリーダクラスまたはファクトリ関数

種々の関数やクラスが以下の引数をとります。

encoder と *decoder*: これらの引数は、Codec インスタンスの `encode()` と `decode()` (Codec Interface 参照) と同じインタフェースを持つ関数、またはメソッドでなければなりません。これらの関数・メソッドは内部状態を持たずに動作する (stateless mode) と想定されています。

incrementalencoder と *incrementaldecoder*: これらは以下のインタフェースを持つファクトリ関数でなければなりません。

`factory(`*errors*`='strict')`

ファクトリ関数は、それぞれ基底クラスの `IncrementalEncoder` や `IncrementalDecoder` が定義しているインタフェースを提供するオブジェクトを返さねばなりません。漸増的 codecs は内部状態を維持できます。

streamreader と *streamwriter*: これらの引数は、次のようなインタフェー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')`

ファクトリ関数は、基底クラスの `StreamWriter` や `StreamReader` が定義しているインタフェースを提供するオブジェクトを返さねばなりません。ストリーム codecs は内部状態を維持できます。

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処理名になります。

検索関数は、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、`None` を返さねばなりません。

`lookup`(*encoding*)

Python codec レジストリから codec 情報を探し、上で定義したような `CodecInfo` オブジェクトを返します。

エンコーディングの検索は、まずレジストリのキャッシュから行います。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。`CodecInfo` オブジェクトが一つも見つからなければ `LookupError` を送出します。見つかったら、その `CodecInfo` オブジェクトはキャッシュに保存され、呼び出し側に返されます。

さまざまな codec へのアクセスを簡便化するために、このモジュールは以下のような関数を提供しています。これらの関数は、 codec の検索に `lookup()` を使います。

`getencoder`(*encoding*)

encoding に指定した codec を検索し、エンコーダ関数を返します。

encoding が見つからなければ `LookupError` を送出します。

`getdecoder`(*encoding*)

encoding に指定した codec を検索し、デコーダ関数を返します。

encoding が見つからなければ `LookupError` を送出します。

`getincrementalencoder`(*encoding*)

encoding に指定した codec を検索し、漸増的エンコーダクラス、またはファクトリ関数を返します。

encoding が見つからない、もしくは codec が漸増的エンコーダをサポートしないとき `LookupError` を送出します。 2.5 で追加された仕様です。

`getincrementaldecoder`(*encoding*)

encoding に指定した codec を検索し、漸増的デコーダクラス、またはファクトリ関数を返します。

encoding が見つからない、もしくは codec が漸増的デコーダをサポートしないとき `LookupError` を送出します。 2.5 で追加された仕様です。

getreader(*encoding*)

encoding に指定した 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` を送出します。

getwriter(*encoding*)

encoding に指定した 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` を送出します。

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送出された場合、*errors* パラメタに *name* を指定していれば *error_handler* を呼び出すようになります。

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とエンコードを再開する場所の指定が入ったタプルを返すかしなければなりません。最後の場合、エンコーダは代替文字列をエンコードし、元の入力中の指定位置からエンコードを再開します。位置を負の値にすると、入力文字列の末端からの相対位置として扱われます。境界の外側にある位置を返した場合には `IndexError` が送出されます。

デコードと翻訳は同様に働きますが、エラー処理関数に渡されるのが `UnicodeDecodeError` か `UnicodeTranslateError` である点と、エラー処理関数の置換した内容が直接出力になる点が異なります。

lookup_error(*name*)

名前 *name* で登録済みの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` を送出します。

strict_errors(*exception*)

`strict` エラー処理の実装です。

replace_errors(*exception*)

`replace` エラー処理の実装です。

ignore_errors(*exception*)

`ignore` エラー処理の実装です。

xmlcharrefreplace_errors_errors(*exception*)

`xmlcharrefreplace` エラー処理の実装です。

backslashreplace_errors_errors(*exception*)

`backslashreplace` エラー処理の実装です。

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*]]])

mode でエンコードされたファイルを開き、透過的にエンコード・デコードを行うようにラップしたファイルオブジェクトを返します。

注意: ラップ版のファイルオブジェクトを操作する関数は、該当する codec が定義している形式のオブジェクトだけを受け付けます。多くの組み込み codec では Unicode オブジェクトです。関数の戻り値も codec に依存し、通常は Unicode オブジェクトです。

encoding にはファイルの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 を送出します。

buffering は、組み込み関数 open() と同じです。デフォルトでは行バッファリングです。

EncodedFile(*file, input*[, *output*[, *errors*]])

ラップしたファイルオブジェクトを返します。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、*input* に指定したエンコーディングに従って変換され、*output* に指定したエンコーディングを使って string 型に変換され、ファイルに書き込まれます。中間エンコーディングは指定された codecs に依存しますが、普通は Unicode です。

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 を送出します。

iterencode(*iterable, encoding*[, *errors*])

漸増的エンコーダを使って、*iterable* から供給される入力を反復的にエンコードします。この関数はジェネレータです。*errors* は (そして他のキーワード引数も同様に) 漸増的エンコーダにそのまま引き渡されます。 2.5 で追加された仕様です。

iterdecode(*iterable, encoding*[, *errors*])

漸増的デコーダを使って、*iterable* から供給される入力を反復的にデコードします。この関数はジェネレータです。*errors* は (そして他のキーワード引数も同様に) 漸増的デコーダにそのまま引き渡されます。 2.5 で追加された仕様です。

このモジュールは以下のような定数も定義しています。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 のバイトオーダマーカ (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に依存します。BOM は BOM_UTF16 の別名です。同様に BOM_LE は BOM_UTF16_LE の、BOM_BE は BOM_UTF16_BE の別名です。他は UTF-8 と UTF-32 エンコーディングの BOM を表します。

4.8.1 Codec 基底クラス

codecs モジュールでは、codec のインタフェースを定義する一連の基底クラスを用意して、Python 用 codec を簡単に自作できるようにしています。

Python で何らかの codec を使えるようにするには、状態なしエンコーダ、状態なしデコーダ、ストームリーダ、ストームライタの 4 つのインタフェースを定義せねばなりません。通常は、状態なしエンコー

ダとデコーダを再利用してストリームリーダとライタのファイル・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・デコーダのインタフェースを定義しています。

エラー処理の簡便化と標準化のため、encode() メソッドと decode() メソッドでは、*errors* 文字列引数を指定した場合に別のエラー処理を行うような仕組みを実装してもかまいません。全ての標準 Python codec では以下の文字列が定義され、実装されています。

Value	Meaning
'strict'	UnicodeError (または、そのサブクラス) を送出します – デフォルトの動作です。
'ignore'	その文字を無視し、次の文字から変換を再開します。
'replace'	適当な文字で置換します – Python の組み込み Unicode codec のデコード時には公式の U+FFI
'xmlcharrefreplace'	適切な XML 文字参照で置換します (エンコードのみ)
'backslashreplace'	バックスラッシュつきのエスケープシーケンスで置換します (エンコードのみ)

codecs がエラーハンドラとして受け入れる値は 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、内部状態を持たないエンコーダ／デコーダ関数のインタフェースを定義します。

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 に内部状態を保存してはなりません。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ような codecs には StreamCodec を使ってください。

エンコーダは長さが 0 の入力を処理できねばなりません。この場合、空のオブジェクトを出力オブジェクトとして返さねばなりません。

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なりません。バッファスロットを提供しているオブジェクトには Python 文字列オブジェクト、バッファオブジェクト、メモリマップファイルがあります。

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 インスタンスに内部状態を保存してはなりません。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ような codecs には StreamCodec を使ってください。

デコーダは長さが 0 の入力を処理できねばなりません。この場合、空のオブジェクトを出力オブジェクトとして返さねばなりません。

IncrementalEncoder クラスおよび IncrementalDecoder クラスはそれぞれ漸増的エンコーディングおよびデコーディングのための基本的なインタフェースを提供します。エンコーディング／デコーディ

ングは内部状態を持たないエンコーダ／デコーダを一度呼び出すことで行なわれるのではなく、漸増的エンコーダ／デコーダの encode/decode メソッドを複数回呼び出すことで行なわれます。漸増的エンコーダ／デコーダはメソッド呼び出しの間エンコーディング／デコーディング処理の進行を管理します。

encode/decode メソッド呼び出しの出力結果をまとめたものは、入力をひとまとめにして内部状態を持たないエンコーダ／デコーダでエンコード／デコードしたものと同じになります。

IncrementalEncoder オブジェクト

2.5 で追加された仕様です。

IncrementalEncoder クラスは入力を複数ステップでエンコードするのに使われます。全ての漸増的エンコーダが Python codec レジストリと互換性を持つために定義すべきメソッドとして、このクラスには以下のメソッドが定義されています。

class IncrementalEncoder(*[errors]*)

IncrementalEncoder インスタンスのコンストラクタ。

全ての漸増的エンコーダはこのコンストラクタインタフェースを提供しなければなりません。さらにキーワード引数を付け加えるのは構いませんが、Python codec レジストリで利用されるのはここで定義されているものだけです。

IncrementalEncoder は *errors* キーワード引数を提供して異なったエラー取扱方法を実装することもできます。あらかじめ定義されているパラメータは以下の通りです。

- 'strict' ValueError (またはそのサブクラス) を送出します。これがデフォルトです。
- 'ignore' 一文字無視して次に進みます。
- 'replace' 適当な代替文字で置き換えます。
- 'xmlcharrefreplace' 適切な XML 文字参照に置き換えます。
- 'backslashreplace' バックスラッシュ付きのエスケープシーケンスで置き換えます。

引数 *errors* は同名の属性に割り当てられます。属性に割り当てることで IncrementalEncoder オブジェクトが生きている間にエラー取扱戦略を違うものに切り替えることができるようになります。

errors 引数に許される値の集合は register_error() で拡張できます。

encode(*object*[, *final*])

object を (エンコーダの現在の状態を考慮に入れて) エンコードし、得られたエンコードされたオブジェクトを返します。encode 呼び出しがこれで最後という時には *final* は真でなければなりません (デフォルトは偽です)。

reset()

エンコーダを初期状態にリセットします。

IncrementalDecoder オブジェクト

IncrementalDecoder クラスは入力を複数ステップでデコードするのに使われます。全ての漸増的デコーダが Python codec レジストリと互換性を持つために定義すべきメソッドとして、このクラスには以下のメソッドが定義されています。

class IncrementalDecoder(*[errors]*)

IncrementalDecoder インスタンスのコンストラクタ。

全ての漸増的デコーダはこのコンストラクタインタフェースを提供しなければなりません。さらにキーワード引数を付け加えるのは構いませんが、Python codec レジストリで利用されるのはここで定義されているものだけです。

IncrementalDecoder は *errors* キーワード引数を提供して異なったエラー取扱方法を実装することもできます。あらかじめ定義されているパラメータは以下の通りです。

- 'strict' ValueError (またはそのサブクラス) を送出します。これがデフォルトです。
- 'ignore' 一文字無視して次に進みます。
- 'replace' 適当な代替文字で置き換えます。

引数 *errors* は同名の属性に割り当てられます。属性に割り当てることで IncrementalDecoder オブジェクトが生きている間にエラー取扱戦略を違うものに切り替えることができるようになります。

errors 引数に許される値の集合は register_error() で拡張できます。

decode (*object*[, *final*])

object を (デコーダの現在の状態を考慮に入れて) デコードし、得られたデコードされたオブジェクトを返します。decode 呼び出しがこれで最後という時には *final* は真でなければなりません (デフォルトは偽です)。もし *final* が真ならばデコーダは入力をデコードし切り全てのバッファをフラッシュしなければなりません。そうできない場合 (たとえば入力の最後に不完全なバイト列があるから)、デコーダは内部状態を持たない場合と同じようにエラーの取り扱いを開始しなければなりません (例外を送出するかもしれません)。

reset ()

デコーダを初期状態にリセットします。

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*])

StreamWriter インスタンスのコンストラクタです。

全てのストリームライタはコンストラクタとしてこのインタフェースを提供せねばなりません。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てい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類似の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。定義済みのパラメタを以下に示します。

- 'strict' ValueError (または、そのサブクラス) 送出します。デフォルトの動作です。
- 'ignore' 文字を無視して、次の文字から続けます。
- 'replace' 適切な置換文字で置換します。
- 'xmlcharrefreplace' 適切な XML 文字参照で置換します。
- '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します。

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と、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できます。

errors 引数が取りえる値の種類は register_error() で拡張できます。

write(*object*)

object の内容をエンコードしてストリームに書き出します。

writelines(*list*)

文字列からなるリストを連結して、(必要に応じて `write()` を何度も使って) ストリームに書き出します。

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てリセットします。

このメソッドが呼び出された場合、出力先データをきれいな状態にし、わざわざストリーム全体を再スキャンして状態を元に戻さなくても新しくデータを追加できるようにせねばなりません。

ここまでで挙げたメソッドの他にも、`StreamWriter` では背後にあるストリームの他の全てのメソッドや属性を継承せねばなりません。

StreamReader オブジェクト

`StreamReader` クラスは `Codec` のサブクラスで、以下のメソッドを定義しています。全てのストリームリーダは、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 **StreamReader**(*stream*[, *errors*])

`StreamReader` インスタンスのコンストラクタです。

全てのストリームリーダはコンストラクタとしてこのインタフェースを提供せねばなりません。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ている引数だけを使います。

stream は、(バイナリで) 読み出し可能なファイル類似のオブジェクトでなくてはなりません。

`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。定義済みのパラメタを以下に示します。

- `'strict'` `ValueError` (または、そのサブクラス) を送出します。デフォルトの処理です。
- `'ignore'` 文字を無視して、次の文字から続けます。
- `'replace'` 適切な置換文字で置換します。

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と、`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できます。

errors 引数が取りえる値の種類は `register_error()` で拡張できます。

read([*size*[, *chars*, [*firstline*]]])

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します。

chars はストリームから読み込む文字数です。`read()` は *chars* 以上の文字を返しませんが、それより少ない文字しか取得できない場合には *chars* 以下の文字を返します。

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、巨大なファイルの一括デコードを防ぐことにあります。

firstline は、1 行目さえ返せばその後の行でデコードエラーがあっても無視して十分だ、ということを示します。

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。 2.4 で変更された仕様: 引数 *chars* が追加されました。 2.4.2 で変更された仕様: 引数 *firstline* が追加されました。

readline(*[size[, keepends]]*)

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します。

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。

keepends が偽の場合には行末の改行が削除された行が返ります。

2.4 で変更された仕様: 引数 *keepends* が追加されました。

readlines(*[sizehint[, keepends]]*)

入力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。

keepends が真なら、改行は、codec のデコーダメソッドを使って実装され、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します。

ストリームの読み位置を再設定してはなら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にエラーから復帰できるようにするためのものです。

ここまでで挙げたメソッドの他にも、`StreamReader` では背後にあるストリームの他の全てのメソッドや属性を継承せねばなりません。

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します。 *stream* はファイル類似のオブジェクトです。*Reader* と *Writer* は、それぞれ `StreamReader` と `StreamWriter` インタフェースを提供するファクトリ関数かファクトリクラスでなければなりません。エラー処理は、ストリームリーダとライタ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フェースを継承します。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、透過に記録できます。

stream はファイル的オブジェクトでなくてはなりません。

encode と *decode* は Codec のインタフェースに忠実でなくてはならず、*Reader* と *Writer* は、それぞれ StreamReader と StreamWriter のインタフェースを提供するオブジェクトのファクトリ関数かクラスでなくてはなりません。

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・リーダやライタ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フェースを定義します。また、元のストリームのメソッドと属性も継承します。

4.8.2 エンコーディングと Unicode

Unicode 文字列は内部的にはコードポイントのシーケンスとして格納されます (正確に言えば Py_UNICODE 配列です)。Python がどのようにコンパイルされたか (デフォルトである**--enable-unicode=ucs2** かまたは **--enable-unicode=ucs4** のどちらか) によって、Py_UNICODE は 16 ビットまたは 32 ビットのデータ型です。Unicode オブジェクトが CPU とメモリの外で使われることになると、CPU のエンディアンやこれらの配列がバイト列としてどのように格納されるかが問題になってきます。Unicode オブジェクトをバイト列に変換することをエンコーディングと呼び、バイト列から Unicode オブジェクトを再生することをデコーディングと呼びます。どのようにこの変換を行うかには多くの異なった方法があります (これらの方法のこともエンコーディングと言います)。最も単純な方法はコードポイント 0-255 をバイト 0x0-0xff に写すことです。これは U+00FF より上のコードポイントを持つ Unicode オブジェクトはこの方法ではエンコードできないということを意味します (この方法を 'latin-1' とか 'iso-8859-1' と呼びます)。unicode.encode() は次のような UnicodeEncodeError を送出することになります: 'UnicodeEncodeError: 'latin-1' codec can't encode character u'\u1234' in position 3: ordinal not in range(256)'。

他のエンコーディングの一群 (charmap エンコーディングと呼ばれます) がありますが、Unicode コードポイントの別の部分集合とこれらがどのように 0x0-0xff のバイトに写されるかを選んだものです。これがどのように行なわれるかを知るには、単にたとえば 'encodings/cp1252.py' (主に Windows で使われるエンコーディングです) を開いてみてください。256 文字のひとつの文字列定数がありどの文字がどのバイト値に写されるかを示しています。

上に挙げた全てのエンコーディングは Unicode に定義された 65536(あるいは 1114111) あるコードポイント中 256 文字しかエンコードできません。全ての Unicode コードポイントを収める単純明快な方法は、それぞれのコードポイントを二つの引き続くバイトに収めるものです。二つの可能性があります。すなわちビッグエンディアンかリトルエンディアンか。これら二つのエンコーディングはそれぞれ UTF-16-BE あるいは UTF-16-LE と呼ばれます。欠点は、たとえば UTF-16-BE をリトルエンディアンの機械で使うときに、エンコーディングでもデコーディングでも常に二つのバイトを交換しなければならないことです。UTF-16 はこの問題を解消します。バイトはいつでも自然なエンディアンに従います。これらのバイトが異なるエンディアンの CPU で読まれる時は、結局交換しない訳にはいきません。UTF-16 のバイト列のエンディアンを検知できるようにするために、いわゆる BOM ("Byte Order Mark") があります。Unicode 文字で言うと U+FEFF です。この文字は全ての UTF-16 バイト列の先頭に付加されます。この文字のバイト位置を交換したもの (0xFFFE) は Unicode テキストに出現しないはずの違法な文字です。そこで、UTF-16 バイト列の一文字目が U+FFFE に見えたなら、デコーディングの際にバイトを交換しなければなりません。不幸なことに、Unicode 4.0 までは文字 U+FEFF には第二の目的 'ZERO WIDTH NO-BREAK SPACE' (幅を持たず単語が分割されるのを許さない文字) がありました。たとえばリガチャ(合字) アルゴリズムに対するヒントを与えるために使われることがあり得ます。Unicode 4.0 になって U+FEFF の 'ZERO WIDTH NO-BREAK SPACE' としての使用法は撤廃されました (U+2060 ('WORD JOINER') にこの役割を譲りました)。しか

しながら、Unicode ソフトウェアは依然として U+FEFF の二つの役割を扱えなければなりません。一つは BOM として、エンコードされたバイトの記憶装置上のレイアウトを決め、バイト列が Unicode 文字列にデコードされた暁には消え去るものという役割。もう一つは 'ZERO WIDTH NO-BREAK SPACE' として、通常の文字と同じようにデコードされる文字という役割です。

さらにもう一つ Unicode 文字全てをエンコードできるエンコーディングがあり、UTF-8 と呼ばれています。UTF-8 は 8 ビットエンコーディングで、したがって UTF-8 にはバイト順の問題はありません。UTF-8 バイト列の各バイトは二つのパートから成ります。二つはマーカ (上位数ビット) とペイロードです。マーカは 0 ビットから 6 ビットの 1 の列に 0 のビットが一つ続いたものです。Unicode 文字は次のようにエンコードされます (x はペイロードを表わし、連結されると一つの Unicode 文字を表わします):

範囲	エンコーディング
U-00000000 ... U-0000007F	0xxxxxxx
U-00000080 ... U-000007FF	110xxxxx 10xxxxxx
U-00000800 ... U-0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
U-00010000 ... U-001FFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
U-00200000 ... U-03FFFFFF	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
U-04000000 ... U-7FFFFFFF	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

Unicode 文字の最下位ビットとは最も右にある x のビットです。

UTF-8 は 8 ビットエンコーディングなので BOM は必要とせず、デコードされた Unicode 文字列中の U+FEFF は (たとえ最初の文字であったとしても) 'ZERO WIDTH NO-BREAK SPACE' として扱われます。

外部からの情報無しには、Unicode 文字列のエンコーディングにどのエンコーディングが使われたのか信頼できる形で決定することは不可能です。どの charmap エンコーディングもどんなランダムなバイト列でもデコードできます。しかし UTF-8 では、任意のバイト列が許される訳ではないような構造を持っているので、そのようなことは可能ではありません。UTF-8 エンコーディングであることを検知する信頼性を向上させるために、Microsoft は Notepad プログラム用に UTF-8 の変種 (Python 2.5 はで "utf-8-sig" と呼んでいます) を考案しました。まだ Unicode 文字がファイルに書き込まれない前に UTF-8 でエンコードした BOM (バイト列では 0xef, 0xbb, 0xbf のように見えます) を書き込んでしまいます。このようなバイト値で charmap エンコードされたファイルが始まることはほとんどあり得ない (たとえば iso-8859-1 では

LATIN SMALL LETTER I WITH DIAERESIS
RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
INVERTED QUESTION MARK

のようになる) ので、utf-8-sig エンコーディングがバイト列から正しく推測される確率を高めます。つまりここでは BOM はバイト列を生成する際のバイト順を決定できるように使われているのではなく、エンコーディングを推測する助けになる印として使われているのです。utf-8-sig codec はエンコーディングの際ファイルに最初の 3 文字として 0xef, 0xbb, 0xbf を書き込みます。デコーディングの際はファイルの先頭に現れたこれら 3 バイトはスキップします。

4.8.3 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し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	英語
big5	big5-tw, csbig5	繁体字中国語
big5hkscs	big5-hkscs, hkscs	繁体字中国語
cp037	IBM037, IBM039	英語
cp424	EBCDIC-CP-HE, IBM424	ヘブライ語
cp437	437, IBM437	英語
cp500	EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500	西ヨーロッパ
cp737		ギリシャ語
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
cp855	855, IBM855	ブルガリア、
cp856		ヘブライ語
cp857	857, IBM857	トルコ語
cp860	860, IBM860	ポルトガル語
cp861	861, CP-IS, IBM861	アイスランド
cp862	862, IBM862	ヘブライ語
cp863	863, IBM863	カナダ
cp864	IBM864	アラビア語
cp865	865, IBM865	デンマーク、
cp866	866, IBM866	ロシア語
cp869	869, CP-GR, IBM869	ギリシャ語
cp874		タイ語
cp875		ギリシャ語
cp932	932, ms932, mskanji, ms-kanji	日本語
cp949	949, ms949, uhc	韓国語
cp950	950, ms950	繁体字中国語
cp1006		Urdu
cp1026	ibm1026	トルコ語
cp1140	ibm1140	西ヨーロッパ
cp1250	windows-1250	中央および東
cp1251	windows-1251	ブルガリア、
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

Codec	別名	言語
cp1256	windows1256	アラビア
cp1257	windows-1257	バルト沿岸国
cp1258	windows-1258	ベトナム
euc_jp	eucjp, ujis, u-jis	日本語
euc_jis_2004	jisx0213, eucjis2004	日本語
euc_jisx0213	eucjisx0213	日本語
euc_kr	euckr, korean, ksc5601, ks_c-5601, ks_c-5601-1987, ksx1001, ks_x-1001	韓国語
gb2312	chinese, csiso58gb231280, euc-cn, euccn, eucgb2312-cn, gb2312-1980, gb2312-80, iso-ir-58	簡体字中国語
gbk	936, cp936, ms936	簡体字中国語
gb18030	gb18030-2000	簡体字中国語
hz	hzgb, hz-gb, hz-gb-2312	簡体字中国語
iso2022_jp	csiso2022jp, iso2022jp, iso-2022-jp	日本語
iso2022_jp_1	iso2022jp-1, iso-2022-jp-1	日本語
iso2022_jp_2	iso2022jp-2, iso-2022-jp-2	日本語, 韓国
iso2022_jp_2004	iso2022jp-2004, iso-2022-jp-2004	日本語
iso2022_jp_3	iso2022jp-3, iso-2022-jp-3	日本語
iso2022_jp_ext	iso2022jp-ext, iso-2022-jp-ext	日本語
iso2022_kr	csiso2022kr, iso2022kr, iso-2022-kr	韓国語
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
iso8859_3	iso-8859-3, latin3, L3	エスペラント
iso8859_4	iso-8859-4, latin4, L4	バルト沿岸国
iso8859_5	iso-8859-5, cyrillic	ブルガリア、
iso8859_6	iso-8859-6, arabic	アラビア語
iso8859_7	iso-8859-7, greek, greek8	ギリシャ語
iso8859_8	iso-8859-8, hebrew	ヘブライ語
iso8859_9	iso-8859-9, latin5, L5	トルコ語
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
johab	cp1361, ms1361	韓国語
koi8_r		ロシア語
koi8_u		ウクライナ
mac_cyrillic	maccyrillic	ブルガリア、
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
mac_roman	macroman	西ヨーロッパ
mac_turkish	macturkish	トルコ語
ptcp154	csptcp154, pt154, cp154, cyrillic-asian	カザフ
shift_jis	csshiftjis, shiftjis, sjis, s_jis	日本語
shift_jis_2004	shiftjis2004, sjis_2004, sjis2004	日本語
shift_jisx0213	shiftjisx0213, sjisx0213, s_jisx0213	日本語
utf_16	U16, utf16	全ての言語

Codec	別名	言語
utf_16_be	UTF-16BE	全ての言語 (
utf_16_le	UTF-16LE	全ての言語 (
utf_7	U7, unicode-1-1-utf-7	全ての言語
utf_8	U8, UTF, utf8	全ての言語
utf_8_sig		全ての言語

codec のいくつかは Python 特有のものなので、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 の中には 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せるという Python codec の性質を利用したものもあります。

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。
bz2_codec	bz2	byte string	被演算子を bz2 を使って圧縮します。
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 の実装です。 2.3 で追加された
mbcs	dbcs	Unicode string	Windows のみ: 被演算子を ANSI コードへ
palmos		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	Unicode 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。

4.8.4 `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、非 ASCII 文字の入ったドメイン名をサポートするためのプロトコルを定義しています。(“www.Alliancefrançaise.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、プロトコル中で任意の文字を使えないような全ての局面で用いられます。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、`httplib` や `ftplib` のようなモジュールでは Unicode ホスト名を受理します (`httplib` 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、国際化ドメイン名で大小文字を区別しないようにするとともに、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。

ToASCII (*label*)
: RFC 3490 仕様に従ってラベルを ASCII に変換します。`UseSTD3ASCIIRules` は偽であると仮定します。

ToUnicode (*label*)
: RFC 3490 仕様に従ってラベルを Unicode に変換します。

4.8.5 `encodings.utf_8_sig` — BOM 印付き UTF-8

2.5 で追加された仕様です。

このモジュールは UTF-8 codec の変種を実装します。この変種はエンコーディング時に UTF-8 でエンコードされた BOM を UTF-8 でエンコードされたバイト列の前に追加します。内部状態を持つエンコーダにとって、これは一度だけ (バイトストリームの最初の書き込み時) 行なわれます。デコーディングに際してはデータ開始の UTF-8 でエンコードされた BOM がもしあったらスキップします。

4.9 `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 4.1.0 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 4.1.0 (`http://www.unicode.org/Public/4.1.0/ucd/UCD.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。

east_asian_width(*unichr*)

unichr as string. ユニコード文字 *unichr* に割り当てられた east asian width を文字列で返します。 2.4 で追加された仕様です。

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。

ucd_3_2_0

これはモジュール全体と同じメソッドを具えたオブジェクトですが、Unicode データベースバージョン 3.2 を代わりに使っており、この特定のバージョンの Unicode データベースを必要とするアプリケーション (IDNA など) のためものです。

2.5 で追加された仕様です。

例:

```
>>> unicodedata.lookup('LEFT CURLY BRACKET')
u'{'
>>> unicodedata.name(u'/')
'SOLIDUS'
>>> unicodedata.decimal(u'9')
9
>>> unicodedata.decimal(u'a')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: not a decimal
>>> unicodedata.category(u'A')  # 'L'etter, 'u'ppercase
'Lu'
>>> unicodedata.bidirectional(u'\u0660') # 'A'rabic, 'N'umber
'AN'
```

4.10 stringprep — インターネットのための文字列調製

2.3 で追加された仕様です。

(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。

4.11 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'
```

五章

データ型

この章で解説されるモジュールは日付や時間、型が固定された配列、ヒープキュー、同期キュー、集合のような種々の特殊なデータ型を提供します。

この章で解説されるモジュールの完全な一覧は:

datetime	基本的な日付型および時間型。
calendar	UNIX の **cal** 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
collections	High-performance container datatypes
heapq	ヒープキュー (別名優先度キュー) アルゴリズム。
bisect	バイナリサーチ用の配列二分法アルゴリズム。
array	一様な型を持つ数値からなる効率のよいアレイ。
sets	ユニークな要素の集合の実装
sched	一般的な目的のためのイベントスケジューラ
mutex	排他制御のためのロックとキュー
Queue	同期キュークラス
weakref	弱参照と弱辞書のサポート。
UserDict	辞書オブジェクトのためのクラスラッパー。
UserList	リストオブジェクトのためのクラスラッパー。
UserString	文字列オブジェクトのためのクラスラッパー。
types	組み込み型の名前
new	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。
copy	浅いコピーおよび深いコピー操作。
pprint	Data pretty printer.
repr	大きさに制限のある別の `repr()` の実装。

5.1 `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、あり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 tsinfo 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 (5.2 節):
: 汎用のカレンダー関連関数。

time モジュール (14.2 節):
: 時刻へのアクセスと変換。

5.1.1 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オ暦 (Gregorian calender) 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のようになります:

```
object
    timedelta
    tzinfo
    time
    date
        datetime
```

5.1.2 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[, seconds[, microseconds[, milliseconds[, minutes[, hours[, weeks]]]]]]]*)

全ての引数がオプションで、デフォルト値は *0* 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ら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
days	両端値を含む -999999999 から 999999999 の間
seconds	両端値を含む 0 から 86399 の間
microseconds	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 timedelta オブジェクトを返します。(2)
−*t1*	timedelta(-*t1.days*, -*t1.seconds*, -*t1.microseconds*)、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。

5.1.3 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オ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、となっていきます。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 "予期的グレゴリオ (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*)

予期的グレゴリオ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 がオーバフローしない場合でも、-timedelta 単体がオーバ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(2002, 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()

予測的グレゴリオ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オ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 (グレゴリオ暦での) 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 年はグレゴリオ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 5.1.7 を参照してください。

5.1.4 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 は現在のグレゴリオ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*[, *minute*[, *second*[, *microsecond*[, *tzinfo*]]]]])
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*])

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*])

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オ暦序数に対応する 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。

strptime(*date_string, format*)

date_string に対応した datetime をかえします。*format* にしたがって構文解析されます。これは、datetime(*(time.strptime(date_string, format)[0:6])) と等価です。date_string と format が time.strptime() で構文解析できない場合や、この関数が 時刻タプルを返してこない場合には ValueError がおこります。

2.5 で追加された仕様です。

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。結果は入力の 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、結果は入力の 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 がオーバフローしない場合でも、-timedelta 単体がオーバ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 datatime 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 MAXUEAR 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()

予測的グレゴリオ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*])

日付と時刻を ISO 8601 形式、すなわち YYYY-MM-DDTHH:MM:SS.mmmmmm 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.mmmmmm+HH:MM となるか、microsecond が ゼロの場合には 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 5.1.7 を参照してください。

5.1.5 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*[, *minute*[, *second*[, *microsecond*[, *tzinfo*]]]])
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。 *tzinfo* のデフォルト値が None である以外のデフォルト値は *0* です。

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、utfoffset() (None を返した場合には 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.mmmmmm 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.mmmmmm+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 5.1.7 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。

5.1.6 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かは、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*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 を送出します。

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いなくてはなりません。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:

```
def dst(self):
    # a fixed-offset class:  doesn't account for DST
    return timedelta(0)
```

or

```
def dst(self):
    # 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) を使うとも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)
HOUR = 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っ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。

5.1.7 `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のようになります:

`%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 strftime() 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。

strftime() 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。

5.1.8 使用例

Datetime オブジェクトをフォーマットされた文字列から生成する

datetime クラスは直接フォーマットされた時刻文字列の構文解析をサポートしていません。time.strptime を使うことによって構文解析をし、返されるタプルから datetime オブジェクトを生成することができます。

```
>>> s = "2005-12-06T12:13:14"
>>> from datetime import datetime
>>> from time import strptime
>>> datetime(*strptime(s, "%Y-%m-%dT%H:%M:%S")[0:6])
datetime.datetime(2005, 12, 6, 12, 13, 14)
```

5.2 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とクラスのほとんどは 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。
(訳注: proleptic Gregorian 暦とはグレゴリオ暦制定 (1582 年) 以前についてもグレゴリオ暦で言い表す暦の方式のことで ISO 8601 などでも採用されています)

class Calendar ([*firstweekday*])

Calendar オブジェクトを作ります。*firstweekday* は整数で週の始まりの曜日を指定するものです。0 が月曜 (デフォルト)、6 なら日曜です。

Calendar オブジェクトは整形されるカレンダーのデータを準備するために使えるいくつかのメソッドを提供しています。しかし整形機能そのものは提供していません。それはサブクラスの仕事なのです。 2.5 で追加された仕様です。

Calendar インスタンスには以下のメソッドがあります。

iterweekdays (*weekday*)

曜日の数字を一週間分生成するイテレータを返します。イテレータから得られる最初の数字は firstweekday() が返す数字と同じになります。

itermonthdates (*year, month*)

year 年 *month* 月に対するイテレータを返します。このイテレータはその月の全ての日

(datetime.date オブジェクトとして) およびその前後の日で週に欠けが無いようにするのに必要な日を返します。

itermonthdays2 (*year, month*)
year 年 *month* 月に対する itermonthdates() と同じようなイテレータを返します。生成されるのは日付の数字と曜日を表す数字のタプルです。

itermonthdays (*year, month*)
year 年 *month* 月に対する itermonthdates() と同じようなイテレータを返します。生成されるのは日付の数字だけです。

monthdatescalendar (*year, month*)
year 年 *month* 月の週のリストを返します。週は全て七つの datetime.date オブジェクトからなるリストです。

monthdays2calendar (*year, month*)
year 年 *month* 月の週のリストを返します。週は全て七つの日付の数字と曜日を表す数字のタプルからなるリストです。

monthdayscalendar (*year, month*)
year 年 *month* 月の週のリストを返します。週は全て七つの日付の数字からなるリストです。

yeardatescalendar (*year*[, *width*])
指定された年のデータを整形に向く形で返します。返される値は月の並びのリストです。月の並びは最大で *width* ヶ月 (デフォルトは 3ヶ月) 分です。各月は 4 ないし 6 週からなり、各週は 1 ないし 7 日からなります。各日は datetime.date オブジェクトです。

yeardays2calendar (*year*[, *width*])
指定された年のデータを整形に向く形で返します (yeardatescalendar() と同様です)。週のリストの中が日付の数字と曜日の数字のタプルになります。月の範囲外の部分の日付はゼロです。

yeardayscalendar (*year*[, *width*])
指定された年のデータを整形に向く形で返します (yeardatescalendar() と同様です)。週のリストの中が日付の数字になります。月の範囲外の日付はゼロです。

class TextCalendar ([*firstweekday*])
このクラスはプレインテキストのカレンダーを生成するのに使えます。

2.5 で追加された仕様です。

TextCalendar インスタンスには以下のメソッドがあります。

formatmonth (*theyear, themonth*[, *w*[, *l*]])
ひと月分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。setfirstweekday() メソッドでセットされた週の最初の曜日に依存します。

prmonth (*theyear, themonth*[, *w*[, *l*]])
formatmonth() で返されるひと月分のカレンダーを出力します。

formatyear (*theyear*[, *w*[, *l*[, *c*[, *m*]]]])
m 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*, *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。setfirstweekday() メソッドでセットされた週の最初の曜日に依存します。カレンダーを出力できる最初の年はプラットフォームに依存します。

pryear (*theyear*[, *w*[, *l*[, *c*[, *m*]]]])
formatyear() で返される一年間のカレンダーを出力します。

class HTMLCalendar ([*firstweekday*])

このクラスは HTML のカレンダーを生成するのに使えます。

2.5 で追加された仕様です。

`HTMLCalendar` インスタンスには以下のメソッドがあります。

formatmonth(*theyear, themonth*[, *withyear*])

ひと月分のカレンダーを HTML のテーブルとして返します。*withyear* が真であればヘッダには年も含まれます。そうでなければ月の名前だけが使われます。

formatyear(*theyear*[, *width*])

一年分のカレンダーを HTML のテーブルとして返します。*width* の値 (デフォルトでは 3 です) は何ヶ月分を一行に収めるかを指定します。

formatyearpage(*theyear*[, *width*[, *css*[, *encoding*]]])

一年分のカレンダーを一つの完全な HTML ページとして返します。*width* の値 (デフォルトでは 3 です) は何ヶ月分を一行に収めるかを指定します。*css* は使われるカスケーディングスタイルシートの名前です。スタイルシートを使わないようにするために `None` を渡すこともできます。*encoding* には出力に使うエンコーディングを指定します (デフォルトではシステムデフォルトのエンコーディングです)。

class **LocaleTextCalendar**([*firstweekday*[, *locale*]])

この `TextCalendar` のサブクラスではコンストラクタにロケール名を渡すことができ、メソッドの返り値で月や曜日が指定されたロケールのものになります。このロケールがエンコーディングを含む場合には、月や曜日の入った文字列はユニコードとして返されます。 2.5 で追加された仕様です。

class **LocaleHTMLCalendar**([*firstweekday*[, *locale*]])

この `HTMLCalendar` のサブクラスではコンストラクタにロケール名を渡すことができ、メソッドの返り値で月や曜日が指定されたロケールのものになります。このロケールがエンコーディングを含む場合には、月や曜日の入った文字列はユニコードとして返されます。 2.5 で追加された仕様です。

単純なテキストのカレンダーに関して、このモジュールには以下のような関数が提供されています。

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 `True` を、そうでなければ `False` 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。

weekheader(*n*)

短縮された曜日名を含むヘッダを返します。*n* は各曜日を何文字で表すかを指定します。

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*]])
`TextCalendar` の `formatmonth` メソッドを利用して、ひと月分のカレンダーを複数行の文字列で返します。 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*]]])
`TextCalendar` の `formatyear` メソッドを利用して、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。 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。

`calendar` モジュールの以下のデータ属性を利用することができます:

day_name
現在のロケールでの曜日を表す配列です。

day_abbr
現在のロケールでの短縮された曜日を表す配列です。

month_name
現在のロケールでの月の名を表す配列です。この配列は通常の約束事に従って、1 月を数字の 1 で表しますので、長さが 13 ある代わりに `month_name[0]` が空文字列になります。

month_abbr
現在のロケールでの短縮された月の名を表す配列です。この配列は通常の約束事に従って、1 月を数字の 1 で表しますので、長さが 13 ある代わりに `month_name[0]` が空文字列になります。

参考資料:

`datetime` モジュール (5.1 節):
`time` モジュールと似た機能を持った日付と時間用のオブジェクト指向インタフェース。

`time` モジュール (14.2 節):
低レベルの時間に関連した関数群。

5.3 `collections` — 高性能なコンテナ・データ型

2.4 で追加された仕様です。

このモジュールでは高性能なコンテナ・データ型を実装しています。現在のところ、実装されている型は deque と defaultdict です。将来的に B-tree と ordere dictionary がふくまれるかもしれません。 2.5 で変更された仕様: defaultdict の追加

5.3.1 deque オブジェクト

deque([*iterable*])

iterable で与えられるデータから、新しい deque オブジェクトを (append() をつかって) 左→右に初期化し、返します。*iterable* が指定されない場合、新しい deque オブジェクトは空になります。

Deque とは、スタックとキューを一般化したものです (この名前は「デック」と発音され、これは「double-ended queue」の省略形です)。Deque はどちらの側からも append と pop が可能で、スレッドセーフでメモリ効率がよく、どちらの方向からもおよそ O(1) のパフォーマンスで実行できます。

list オブジェクトでも同様の操作を実現できますが、これは高速な固定長の操作に特化されており、内部のデータ表現形式のサイズと位置を両方変えるような 'pop(0)' and 'insert(0, v)' などの操作ではメモリ移動のために O(n) のコストを必要とします。 2.4 で追加された仕様です。

Deque オブジェクトは以下のようなメソッドをサポートしています:

append(*x*)

x を deque の右側につけ加えます。

appendleft(*x*)

x を deque の左側につけ加えます。

clear()

Deque からすべての要素を削除し、長さを 0 にします。

extend(*iterable*)

イテレータ化可能な引数 iterable から得られる要素を deque の右側に追加し拡張します。

extendleft(*iterable*)

イテレータ化可能な引数 iterable から得られる要素を deque の左側に追加し拡張します。注意: 左から追加した結果は、イテレータ引数の順序とは逆になります。

pop()

Deque の右側から要素をひとつ削除し、その要素を返します。要素がひとつも存在しない場合は IndexError を発生させます。

popleft()

Deque の左側から要素をひとつ削除し、その要素を返します。要素がひとつも存在しない場合は IndexError を発生させます。

remove(*value*)

最初に現れる value を削除します。要素がみつからないない場合は ValueError を発生させます。2.5 で追加された仕様です。

rotate(*n*)

Deque の要素を全体で *n* ステップだけ右にローテートします。*n* が負の値の場合は、左にローテートします。Deque をひとつ右にローテートすることは 'd.appendleft(d.pop())' と同じです。

上記の操作のほかにも、deque は次のような操作をサポートしています: イテレータ化、pickle、'len(d)'、'reversed(d)'、'copy.copy(d)'、 'copy.deepcopy(d)'、 in 演算子による包含検査、そして 'd[-1]' などの添え字による参照。

例:

```
>>> from collections import deque
>>> d = deque('ghi')                 # 3 つの要素からなる新しい deque をつくる。
>>> for elem in d:                   # deque の要素をひとつずつたどる。
...     print elem.upper()
G
H
I

>>> d.append('j')                    # 新しい要素を右側につけたす。
>>> d.appendleft('f')                # 新しい要素を左側につけたす。
>>> d                                # deque の表現形式。
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                          # いちばん右側の要素を削除し返す。
'j'
>>> d.popleft()                      # いちばん左側の要素を削除し返す。
'f'
>>> list(d)                          # deque の内容をリストにする。
['g', 'h', 'i']
>>> d[0]                             # いちばん左側の要素をのぞく。
'g'
>>> d[-1]                            # いちばん右側の要素をのぞく。
'i'

>>> list(reversed(d))                # deque の内容を逆順でリストにする。
['i', 'h', 'g']
>>> 'h' in d                         # deque を検索。
True
>>> d.extend('jkl')                  # 複数の要素を一度に追加する。
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1)                      # 右ローテート
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)                     # 左ローテート
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))               # 新しい deque を逆順でつくる。
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()                        # deque を空にする。
>>> d.pop()                          # 空の deque からは pop できない。
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')              # extendleft() は入力を逆順にする。
>>> d
deque(['c', 'b', 'a'])
```

5.3.2 レシピ

この節では deque をつかったさまざまなアプローチを紹介します。

rotate() メソッドのおかげで、 deque の一部を切り出したり削除したりできることになります。たとえば del d[n] の純粋な Python 実装では pop したい要素まで rotate() します :

```
def delete_nth(d, n):
    d.rotate(-n)
    d.popleft()
    d.rotate(n)
```

deque の切り出しを実装するのにも、同様のアプローチを使います。まず対象となる要素を rotate() によって deque の左端までもってきてから、popleft() をつかって古い要素を消します。そして、extend() で新しい要素を追加したのち、逆のローテートでもとに戻せばよいのです。

このアプローチをやや変えたものとして、Forth スタイルのスタック操作、つまり dup, drop, swap, over, pick, rot, および roll を実装するのも簡単です。

ラウンドロビンのタスクサーバは deque をつかって、popleft() で現在のタスクを選択し、入力ストリームが使い果たされなければ append() でタスクリストの戻してやることができます:

```
def roundrobin(*iterables):
    pending = deque(iter(i) for i in iterables)
    while pending:
        task = pending.popleft()
        try:
            yield task.next()
        except StopIteration:
            continue
        pending.append(task)

>>> for value in roundrobin('abc', 'd', 'efgh'):
...     print value

a
d
e
b
f
c
g
h
```

複数パスのデータ・リダクション アルゴリズムは、popleft() を複数回呼んで要素をとりだし、リダクション用の関数を適用してから append() で deque に戻してやることにより、簡潔かつ効率的に表現することができます。

たとえば入れ子状になったリストでバランスされた二進木をつくりたい場合、2 つの隣接するノードをひとつのリストにグループ化することになります:

```
def maketree(iterable):
    d = deque(iterable)
    while len(d) > 1:
        pair = [d.popleft(), d.popleft()]
        d.append(pair)
    return list(d)

>>> print maketree('abcdefgh')
[[[['a', 'b'], ['c', 'd']], [['e', 'f'], ['g', 'h']]]]
```

5.3.3 defaultdict オブジェクト

defaultdict([*default_factory*[, ...]])

新しいディクショナリ状のオブジェクトを返します。defaultdict は組込みの dict のサブクラスです。メソッドをオーバーライドし、書き込み可能なインスタンス変数を 1 つ追加している以外は dict クラスと同じです。同じ部分については以下では省略されています。

1 つめの引数は default_factory 属性の初期値です。デフォルトは None です。残りの引数はキーワード引数もふくめ、dict のコンストラクタにあたえられた場合と同様に扱われます。

2.5 で追加された仕様です。

defaultdict オブジェクトは標準の dict に加えて、以下のメソッドを実装しています:

__missing__(*key*)

もし default_factory 属性が None であれば、このメソッドは KeyError 例外を、*key* を引数として発生させます。

もし default_factory 属性が None でなければ、このメソッドは default_factory を引数なしで呼び出し、あたえられた *key* に対応するデフォルト値を作ります。そしてこの値を *key* に対応する値を辞書に登録して返ります。

もし default_factory の呼出が例外を発生させた場合には、変更せずそのまま例外を投げます。

このメソッドは dict クラスの __getitem__ メソッドで、キーが存在しなかった場合によびだされます。値を返すか例外を発生させるのどちらにしても、__getitem__からもそのまま値が返るか例外が発生します。

defaultdict オブジェクトは以下のインスタンス変数をサポートしています:

default_factory

この属性は __missing__ メソッドによって使われます。これは存在すればコンストラクタの第 1 引数によって初期化され、そうでなければ None になります。

defaultdict の使用例

list を default_factory とすることで、キー=値ペアのシーケンスをリストの辞書へ簡単にグループ化できます。

```
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
        d[k].append(v)

>>> d.items()
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

それぞれのキーが最初に登場したとき、マッピングにはまだ存在しません。そのためエントリは default_factory 関数が返す空の list を使って自動的に作成されます。list.append() 操作は新しいリストに紐付けられます。キーが再度出現下場合には、通常の参照動作が行われます (そのキーに対応するリストが返ります)。そして list.append() 操作で別の値をリストに追加します。このテクニックは dict.setdefault() を使った等価なものよりシンプルで速いです:

```
>>> d = {}
>>> for k, v in s:
d.setdefault(k, []).append(v)

>>> d.items()
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

default_factory を int にすると、defaultdict を (他の言語の bag や multiset のように) 要素の数え上げに便利に使うことができます:

```
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
        d[k] += 1

>>> d.items()
[('i', 4), ('p', 2), ('s', 4), ('m', 1)]
```

最初に文字が出現したときは、マッピングが存在しないので default_factory 関数が int() を呼んでデフォルトのカウント 0 を生成します。インクリメント操作が各文字を数え上げます。このテクニックは以下の dict.get() を使った等価なものよりシンプルで速いです:

```
>>> d = {}
>>> for k in s:
d[k] = d.get(k, 0) + 1

>>> d.items()
[('i', 4), ('p', 2), ('s', 4), ('m', 1)]
```

default_factory を set に設定することで、defaultdict をセットの辞書を作るために利用することができます:

```
>>> s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1), ('blue', 4)]
>>> d = defaultdict(set)
>>> for k, v in s:
        d[k].add(v)

>>> d.items()
[('blue', set([2, 4])), ('red', set([1, 3]))]
```

5.4 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*] <= *heap*[2**k*+1] かつ *heap*[*k*] <= *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 つの側面で異なっています: (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* をインプレース処理し、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は条件つき置換の一部として使われることに制限されています。

```
if item > heap[0]:
    item = heapreplace(heap, item)
```

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
>>>
```

このモジュールではさらに 2 つのヒープに基く汎用関数を提供します。

nlargest(*n, iterable*[, *key*])
: *iterable* で定義されるデータセットのうち、最大値から降順に *n* 個の値のリストを返します。(あたえられた場合)*key* は、引数を一つとる、*iterable* のそれぞれの要素から比較キーを生成する関数を指定します: '*key*=`str.lower`' 以下のコードと同等です: `sorted(iterable, key=key,`

reverse=True)[:n] 2.4 で追加された仕様です。 2.5 で変更された仕様: 省略可能な *key* 引数を追加

nsmallest(*n, iterable*[, *key*])

iterable で定義されるデータセットのうち、最小値から昇順に *n* 個の値のリストを返します。(あたえられた場合)*key* は、引数を一つとる、*iterable* のそれぞれの要素から比較キーを生成する関数を指定します: '*key*=str.lower' 以下のコードと同等です: sorted(iterable, key=key)[:n] 2.4 で追加された仕様です。 2.5 で変更された仕様: 省略可能な *key* 引数を追加

どちらの関数も *n* の値が小さな場合に最適な動作をします。大きな値の時には sorted() 関数の方が効率的です。さらに、n==1 の時には min() および max() 関数の方が効率的です。

5.4.1 理論

(説明は François 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*] <= a[2**k*+1] かつ a[*k*] <= a[2**k*+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:

```
                                   0

                  1                                2

          3               4                5               6

      7       8       9       10      11      12      13      14

  15 16   17 18   19 20   21 22   23 24   25 26   27 28   29 30
```

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 の場所に持っていき、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ところです。

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、たいていで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[1]。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" しまうために) 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。 :-)

5.5 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*]])

[1]現在使われているディスクバランス化アルゴリズムは、最近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！:-)

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。

5.5.1 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']
```

5.6 array — 効率のよい数値アレイ

このモジュールでは、基本的な値 (文字、整数、浮動小数点数) のアレイ (array、配列) を効率よく表現できるオブジェクト型を定義しています。アレイはシーケンス (sequence) 型であり、中に入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。値の実際のサイズは itemsize 属性から得られます。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' で表現されている要素に入る値は Python では長整数として表されます。

このモジュールでは次の型を定義しています:

array (*typecode*[, *initializer*])
要素のデータ型が *typecode* に限定される新しいアレイを返します。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、文字列または適当な型のイテレーション可能オブジェクトでなければなりません。

2.4 で変更された仕様: 以前はリストか文字列しか受け付けませんでした。 リストか文字列を渡した場合、新たに作成されたアレイの `fromlist()`、`fromstring()` あるいは `fromunicode()` メソッド (以下を参照して下さい) に渡され、初期値としてアレイに追加されます。それ以外の場合には、イテレーション可能オブジェクト *initializer* は新たに作成されたオブジェクトの `extend()` メソッドに渡されます。

ArrayType
`array` の別名です。撤廃されました。

アレイオブジェクトでは、インデクス指定、スライス、連結および反復といった、通常のシーケンスの演算をサポートしています。スライス代入を使うときは、代入値は同じ型コードのアレイオブジェクトでなければなりません。それ以外のオブジェクトを指定すると `TypeError` を送出します。アレイオブジェクトはバッファインタフェースを実装しており、バッファオブジェクトをサポートしている場所ならどこでも利用できます。

次のデータ要素やメソッドもサポートされています:

typecode
アレイを作るときに使う型コード文字です。

itemsize
アレイの要素 1 つの内部表現に使われるバイト長です。

append (*x*)
値 *x* の新たな要素をアレイの末尾に追加します。

buffer_info ()
アレイの内容を記憶するために使っているバッファの、現在のメモリアドレスと要素数の入ったタプル (*address*, *length*) を返します。バイト単位で表したメモリバッファの大きさは *array*`.buffer_info()[1] *` *array*`.itemsize` で計算できます。例えば `ioctl()` 操作のような、メモリアドレスを必要とする低レベルな (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。アレイ自体が存在し、長さを変えるような演算を適用しない限り、有効な値を返します。

注意: C や C++ で書いたコードからアレイオブジェクトを使う場合 (`buffer_info` の情報を使う意味のある唯一の方法です) は、アレイオブジェクトでサポートしているバッファインタフェー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。バッファインタフェースの説明は *Python/C API* **リファレンスマニュアル**にあります。

byteswap ()
アレイ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。このメソッドは大きさが 1、2、4 および 8 バイトの値にのみをサポートしています。他の型の値に使うと `RuntimeError` を送出します。異なるバイトオーダをもつ計算機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*)
シーケンス中の *x* の出現回数を返します。

extend(*iterable*)

iterable から要素を取り出し、アレイの末尾に要素を追加します。*iterable* が別のアレイ型である場合、二つのアレイは**全く**同じ型コードをでなければなりません。それ以外の場合には `TypeError` を送出します。*iterable* がアレイでない場合、アレイに値を追加できるような正しい型の要素からなるイテレーション可能オブジェクトでなければなりません。 2.4 で変更された仕様: 以前は他のアレイ型しか引数に指定できませんでした。

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で) *n* 個の要素を読み出し、アレイの末尾に要素を追加します。*n* 個の要素を読めなかったときは `EOFError` を送出しますが、それまでに読み出せた値はアレイに追加されています。*f* は本当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型では動作しません。

fromlist(*list*)

リストから要素を追加します。型に関するエラーが発生した場合にアレイが変更されないことを除き、'`for x in` *list*`: a.append(x)`' と同じです。

fromstring(*s*)

文字列から要素を追加します。文字列は、 (ファイルから `fromfile()` メソッドを使って値を読み込んだときのように) マシン依存のデータ形式で表された値の配列として解釈されます。

fromunicode(*s*)

指定した Unicode 文字列のデータを使ってアレイを拡張します。アレイの型コードは `'u'` でなければなりません。それ以外の場合には、`ValueError` を送出します。他の型のアレイに Unicode 型の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*)

アレイ中で *x* が出現するインデクスのうち最小の値 *i* を返します。

insert(*i, x*)

アレイ中の位置 *i* の前に値 *x* をもつ新しい要素を挿入します。*i* の値が負の場合、アレイの末尾からの相対位置として扱います。

pop([*i*])

アレイからインデクスが *i* の要素を取り除いて返します。オプションの引数はデフォルトで`-1` になっていて、最後の要素を取り除いて返すようになっています。

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で) *n* 個の要素を読み出し、アレイの末尾に要素を追加します。*n* 個の要素を読めなかったときは `EOFError` を送出しますが、それまでに読み出せた値はアレイに追加されています。*f* は本当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型では動作しません。

remove(*x*)

アレイ中の *x* のうち、最初に現れたものを取り除きます。

reverse()

アレイの要素の順番を逆にします。

tofile(*f*)

アレイのすべての要素をファイルオブジェクト *f* に (マシン依存のデータ形式そのままで) 書き込みます。

tolist()

アレイを同じ要素を持つ普通のリストに変換します。

tostring()
アレイをマシン依存のデータアレイに変換し、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列) を返します。

tounicode()
アレイを Unicode 文字列に変換します。アレイの型コードは 'u' でなければなりません。それ以外の場合には ValueError を送出します。他の型のアレイから Unicode 文字列を得るには、'array.tostring().decode(enc)' 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で) 書き込みます。

アレイオブジェクトを表示したり文字列に変換したりすると、array(*typecode*, *initializer*) という形式で表現されます。アレイが空の場合、*initializer* の表示を省略します。アレイが空でなければ、*typecode* が 'c' の場合には文字列に、それ以外の場合には数値のリストになります。関数 array() を from array import array で import している限り、変換後の文字列に逆クォーテーション (`) を用いると元のアレイオブジェクトと同じデータ型と値を持つアレイに逆変換できることが保証されています。文字列表現の例を以下に示します:

```
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のパックおよびアンパック。

xdrlib モジュール (9.5 節):
遠隔手続き呼び出しシステムで使われる外部データ表現仕様 (External Data Representation, XDR) のデータのパックおよびアンパック。

The Numerical Python Manual
(http://numpy.sourceforge.net/numdoc/HTML/numdoc.htm)
Numeric Python 拡張モジュール (NumPy) では、別の方法でシーケンス型を定義しています。Numerical Python に関する詳しい情報は 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。

5.7 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。このことから、集合の要素にするには辞書のキーと同様の要件を満たさなければなりません。具体的には、要素になるものには __eq__ と__hash__ が定義されているという条件です。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、イタ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 5.7.3 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、イタ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 5.7.3 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。

5.7.1 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
len(*s*)		集合 *s* の濃度 (cardinality)
x in *s*		*x* が *s* に帰属していれば真を返す
x not in *s*		*x* が *s* に帰属していなければ真を返す
s.issubset(*t*)	*s* <= *t*	*s* のすべての要素が *t* に帰属していれば真を返す
s.issuperset(*t*)	*s* >= *t*	*t* のすべての要素が *s* に帰属していれば真を返す
s.union(*t*)	*s* | *t*	*s* と *t* の両方の要素からなる新しい集合
s.intersection(*t*)	*s* & *t*	*s* と *t* で共通する要素からなる新しい集合
s.difference(*t*)	*s* - *t*	*s* にあるが *t* にない要素からなる新しい集合
s.symmetric_difference(*t*)	*s* ^ *t*	*s* と *t* のどちらか一方に属する要素からなる集合
s.copy()		*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() は任意のイテレート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。

部分集合比較やと等値比較では、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
hash(*s*)	*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
s.update(*t*)	*s* |= *t*	*t* を加えた要素からなる集合 *s* を返します
s.intersection_update(*t*)	*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
s.difference_update(*t*)	*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
s.symmetric_difference_update(*t*)	*s* ^= *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
s.add(*x*)		要素 *x* を集合 *s* に加えます
s.remove(*x*)		要素 *x* を集合 *s* から取り除きます; *x* がなければ KeyError を
s.discard(*x*)		要素 *x* が存在すれば、集合 *s* から取り除きます
s.pop()		*s* から要素を取り除き、それを返します; 集合が空なら KeyE
s.clear()		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 update()、intersection_update()、difference_update()、および symmetric_difference_update() は任意のイテレート可能オブジェク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。

もう一つ注意を述べますが、このモジュールでは union_update() が update() の別名として含まれています。このメソッドは後方互換性のために残されているものです。プログラマは組み込みの set() および frozenset() でサポートされている update() を選ぶべきです。

5.7.2 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'])
```

5.7.3 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。言いかえれば、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。

5.7.4 組み込み set 型との比較

組み込みの set および frozenset 型はこの sets で学んだことを生かして設計されています。主な違いは次の通りです。

- Set と ImmutableSet は set と frozenset に改名されました。
- BaseSet に相当するものはありません。代わりに isinstance(x, (set, frozenset)) を使って下さい。
- 組み込みのものに使われているハッシュアルゴリズムは、多くのデータ集合に対してずっと良い性能 (少ない衝突) を実現します。
- 組み込みのものはより空間効率良く pickle 化できます。
- 組み込みのものには union_update() メソッドがありません。代わりに同じ機能の update() メソッドを使って下さい。
- 組み込みのものには _repr(sorted=True) メソッドがありません。代わりに組み込み関数の repr() と sorted() を使って repr(sorted(s)) として下さい。
- 組み込みのものは変更不可能なものに自動で変換するプロトコルがありません。この機能は多くの人が困惑を覚えるわりに、コミュニティの誰からも実際的な使用例の報告がありませんでした。

5.8 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
```

5.8.1 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。

5.9 mutex — 排他制御

mutex モジュールでは、ロック (lock) の獲得と解除によって排他制御を可能にするクラスを定義しています。排他制御はスレッドやマルチタスクを使う上で便利かもしれませんが、このクラスがそうした機能を必要として (いたり、想定して) いるわけではありません。

mutex モジュールでは以下のクラスを定義しています:

class mutex()
新しい (ロックされてない) mutex を作ります。

mutex には 2 つの状態変数 — “ロック” ビット (locked bit) とキュー (queue) があります。mutex がロックされていなければ、キューは空です。それ以外の場合、キューは空になっているか、(*function*, *argument*) のペアが一つ以上入っています。このペアはロックを獲得しようと待機している関数 (またはメソッド) を表しています。キューが空でないときに mutex をロック解除すると、キューの先頭のエントリをキューから除去し、そのエントリのペアに基づいて *function*(*argument*) を呼び出します。これによって、先頭にあったエントリが新たなロックを獲得します。

当然のことながらマルチスレッドの制御には利用できません – というのも、lock() が、ロックを獲得したら関数を呼び出すという変なインタフェースだからです。

5.9.1 mutex オブジェクト

mutex には以下のメソッドがあります:

test()
mutex がロックされているかどうか調べます。

testandset()
「原子的 (Atomic)」な Test-and-Set 操作です。ロックがセットされていなければ獲得して True を返します。それ以外の場合には 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。mutex がロックされている場合、関数とその引数をキューに置きます。キューに置かれた *function* (*argument*) がいつ実行されるかについては unlock を参照してください。

unlock()
キューが空ならば mutex をロック解除します。そうでなければ、キューの最初の要素を実行します。

5.10 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。

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
空な 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
満杯な 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。

5.10.1 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。

キューに入れられたタスクが全て消費者スレッドに処理されたかどうかを追跡するために 2 つのメソッドが提供されます。

task_done()

過去にキューに入れられたタスクが完了した事を示します。キューの消費者スレッドに利用されます。タスクの取り出しに使われた、各 `get()` に対して、それに続く `task_done()` の呼び出しは、取り出したタスクに対する処理が完了した事をキューに教えます。

`join()` がブロックされていた場合、全 item が処理された (キューに `put()` された全ての item に対して `task_done()` が呼び出されたことを意味します) 時に復帰します。

キューにあるより item の個数よりも多く呼び出された場合、`ValueError` が送出されます。 2.5 で追加された仕様です。

join()

キューの中の全アイテムが処理される間でブロックします。

キューに item が追加される度に、未完了タスクカウントが増やされます。消費者スレッドが `task_done()` を呼び出して、item を受け取ってそれに対する処理が完了した事を知らせる度に、未完了タスクカウントが減らされます。未完了タスクカウントが 0 になったときに、join() のブロックが解除されます。 2.5 で追加された仕様です。

キューに入れたタスクが完了するのを待つ例:

```
def worker():
    while True:
        item = q.get()
        do_work(item)
        q.task_done()

q = Queue()
for i in range(num_worker_threads):
     t = Thread(target=worker)
     t.setDaemon(True)
     t.start()

for item in source():
    q.put(item)

q.join()       # 全タスクが完了するまでブロック
```

5.11 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 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、set および frozenset 型、ファイルオブジェクト、ジェネレータ、型オブジェクト、bsddb モジュールの DBcursor 型、ソケット型、array 型、deque 型、および正規表現パターンオブジェクトです。 2.4 で変更された仕様: ファイル、ソケット、array、および正規表現パターンのサポートを追加しました

list や dict など、いくつかの組み込み型は弱参照を直接サポートしませんが、以下のようにサブク

ラス化を行えばサポートを追加できます:

```
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)   # this object is weak referencable
```

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、5.11.3 節 “拡張型における弱参照” を読んでください。

class 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す。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* に None 以外の値を与えた場合、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。

2.4 で変更された仕様: 以前はファクトリでしたが、サブクラス化可能な型になりました。object 型から導出されています

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ーベ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。

5.11.1 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。

サブクラス化を行えば、ref オブジェクトの特殊なバージョンを作成できます。これは WeakValueDictionary の実装で使われており、マップ内の各エントリによるメモリのオーバヘッドを減らしています。こうした実装は、ある参照に追加情報を関連付けたい場合に便利ですし、リファレントを取り出すための呼び出し時に何らかの追加処理を行いたい場合にも使えます。

以下の例では、ref のサブクラスを使って、あるオブジェクトに追加情報を保存し、リファレントがアクセスされたときにその値に作用をできるようにするための方法を示しています:

```
import weakref

class ExtendedRef(weakref.ref):
    def __new__(cls, ob, callback=None, **annotations):
        weakref.ref.__new__(cls, ob, callback)
        self.__counter = 0

    def __init__(self, ob, callback=None, **annotations):
        super(ExtendedRef, self).__init__(ob, callback)
        for k, v in annotations:
            setattr(self, k, v)

    def __call__(self):
        """Return a pair containing the referent and the number of
        times the reference has been called.
        """
        ob = super(ExtendedRef, self)()
        if ob is not None:
            self.__counter += 1
            ob = (ob, self.__counter)
        return ob
```

5.11.2 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す。

```
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]
```

5.11.3 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。 */
}
```

5.12 UserDict — 辞書オブジェクトのためのクラスラッパー

このモジュールは 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、 すべての辞書メソッドを定義している mixin、DictMixin を定義しています。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。

このモジュールでは UserDict クラスを定義しています。これは辞書オブジェクトのラッパーとして動作します。これは dict(Python 2.2 から利用可能な機能です) によって置き換えられています。dict の導入以前に、UserDict クラスは辞書風のサブクラスをオーバライドや新メソッドの定義によって作成するために使われていました。

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。**注意:** 後方互換性のために、UserDict のインスタンスはイテレート可能ではありません。

class IterableUserDict ([*initialdata*])

UserDict のイテレーションをサポートするサブクラス (使用例: for key in myDict).

マッピングのメソッドと演算 (節 3.8 を参照) に加えて、UserDict、IterableUserDict 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。

5.13 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 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。

5.14 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 3.6.1、“文字列メソッド” を参照) に加えて、UserString 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。

5.15 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 listiterator 型のようなプロセス中に例外をふくまないので、‘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。

DictProxyType
`TypeType.__dict__` のような dict へのプロキシ型です。

NotImplementedType
`NotImplemented` の型です。

GetSetDescriptorType
`FrameType.f_locals` や `array.array.typecode` のような `PyGetSetDef` のある 拡張モジュールで定義されたオブジェクトの型です。この定数は上のような拡張型がない Python では定義されません。ポータブルなコードでは `hasattr(types, 'GetSetDescriptorType')` を使用してください。 2.5 で追加された仕様です。

MemberDescriptorType
`datetime.timedelta.days` のような `PyMemberDef` のある拡張モジュールで定義されたオブジェクトの型です。この定数は上のような拡張型がない Python では定義されません。ポータブルなコードでは `hasattr(types, 'MemberDescriptorType')` を使用してください。 2.5 で追加された仕様です。

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。

5.16 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。オブジェクトが利用される時にインタプリタをクラッシュさせるような引数を与えることもできてしまいます。

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*[, *closure*]]])
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 でなければならない。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。*closure* を与える場合は None または名前を *code*.co_freevars に束縛するセルオブジェクトのタプルである必要があります。

) が与えられていると、

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*[, *doc*])
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らない。省略可能な *doc* 引数はどんな型でもよい。

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。

5.17 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。

このモジュールでは、モジュール、メソッド、スタックトレース、スタックフレーム、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。このモジュールでは元のオブジェクトを変更せずに返すことで関数とクラスを (浅く または 深く)「コピー」します。これは pickle モジュールでの扱われかたと同じです。 2.5 で変更された仕様: 関数コピーの追加

クラスでは、pickle 化を制御するためのインタフェース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 (13.1 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。

5.18 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ー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。

2.5 で変更された仕様: 辞書は出力を計算する前にキーでソートされます。2.5 以前では、辞書は 1 行以上必要な場合にのみソートされていましたがドキュメントには書かれていませんでした。

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：

pformat(*object*[, *indent*[, *width*[, *depth*]]])

object をフォーマット化して文字列として返します。*indent*、*width* と、*depth* は `PrettyPrinter` コンストラクタにフォーマット指定引数として渡されます。 2.4 で変更された仕様: 引数 *indent*、 *width* と、*depth* が追加されました

pprint(*object*[, *stream*[, *indent*[, *width*[, *depth*]]]])

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ープリタ上で、求める値を `print` する代わりに使用できます。*indent*、*width* と、*depth* は `PrettyPrinter` コンストラクタにフォーマット指定引数として渡されます。

```
>>> 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']
```

2.4 で変更された仕様: 引数 *indent*、 *width* と、*depth* が追加されました

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/loca
l/lib/python1.5/test', '/usr/local/lib/python1.5/sunos5', '/usr/local/lib/python
1.5/sharedmodules', '/usr/local/lib/python1.5/tkinter']"
```

5.18.1 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。

5.19 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。

5.19.1 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
maxset
maxfrozenset
maxdeque
maxarray
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、maxarray は 5、その他に対しては 6 です。 2.4 で追加された仕様: maxset, maxfrozenset, set .

maxlong
長整数の表現の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。

5.19.2 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>'
```

六章

数値と数学モジュール

この章で解説されるモジュールは数値と数学関連の関数とデータ型を提供する。`math` と `cmath` はさまざまな浮動小数点数および複素数向け数学関数を含みます。速度より 10 進法での正確さに興味があるユーザには、`decimal` モジュールが真の 10 進表現をサポートしています。

この章で解説されるモジュールの一覧は:

math	数学関数 (`sin()` など)。
cmath	複素数のための数学関数です。
decimal	汎用 10 進数算術仕様 (General Decimal Arithmetic Specification) の実装。
random	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。
itertools	効率的なループ実行のためのイテレータ生成関数。
functools	高階関数と呼び出し可能オブジェクトの操作
operator	組み込み関数形式になっている全ての Python の標準演算子。

6.1 `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。[1]

このモジュールでは次の関数を提供しています。明示的な注記のない限り、戻り値は全て浮動小数点数になります。

以下は整数論および数表現にまつわる関数です:

ceil(*x*)
: *x* の天井値 (ceil)、すなわち *x* 以上の最も小さい整数を float 型で返します。

fabs(*x*)
: *x* の絶対値を返します。

floor(*x*)
: *x* の床値 (floor)、すなわち *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* という式は必ずしも同じ結果を返さないということに注意してください。C 標準の要求では、`fmod` は除算

[1] 訳注：例外が発生しないで、計算結果が返えってしまうと、計算結果がおかしい事から、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。

の結果が *x* と同じ符号になり、大きさが `abs(`*y*`)` より小さくなるような整数 *n* については `fmod(`*x*`, `*y*`)` が厳密に (数学的に、つまり限りなく高い精度で) *x* `-` *n*`*`*y* と等価であるよう求めています。Python の *x* `%` *y* は、*y* と同じ符号の結果を返し、浮動小数点の引数に対して厳密な解を出せないことがあります。例えば、`fmod(-1e-100, 1e100)` は `-1e-100` ですが、Python の `-1e-100 % 1e100` は `1e100-1e-100` になり、浮動小数点型で厳密に表現できず、ややこしいことに `1e100` に丸められます。このため、一般には浮動小数点の場合には関数 `fmod()`、整数の場合には *x* `%` *y* を使う方がよいでしょう。

frexp(*x*)
: *x* の仮数と指数を `(`*m*`, `*e*`)` のペアとして返します。*m* は float 型で、*e* は厳密に *x* `==` *m* `*` `2**`*e* であるような整数型です。*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` を返します。これは浮動小数点型の内部表現を可搬性を保ったまま "分解する (pick apart)" するためです。

ldexp(*x*, *i*)
: *x* `*` `(2**`*i*`)` 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。

`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 '出力用の引数' 経由で返したりはしません (Python には出力用の引数はありません)。

`ceil()`、`floor()`、および `modf()` 関数については、非常に大きな浮動小数点数が**全て** 整数そのものになるということに注意してください。通常、Python の浮動小数点型は 53 ビット以上の精度をもたない (プラットフォームにおける C double 型と同じ) ので、結果的に `abs(`*x*`) >= 2**52` であるような浮動小数点型 *x* は小数部分を持たなくなるのです。

以下は指数および対数関数です:

exp(*x*)
: `e**`*x* 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。

pow(*x*, *y*)
: *x*`**`*y* を返します。

sqrt(*x*)
: *x* の平方根を返します。

以下は三角関数です:

acos(*x*)
: *x* の逆余弦を返します。

asin(*x*)
: *x* の逆正弦を返します。

atan(*x*)
: *x* 逆正接を返します。

atan2(*y*, *x*)

atan(*y* / *x*) の逆正接をラジアンで返します。戻り値は -pi から pi の間になります。この角度は、極座標平面において原点から (*x*, *y*) へのベクトルが X 軸の正の方向となす角です。atan2() のポイントは、入力 *x*, *y* の両方の符号が既知であるために、位相角の正しい象限を計算できることにあります。例えば、atan(1) と atan2(1,1) はいずれも pi/4 ですが、atan2(-1, -1) は -3*pi/4 になります。

cos(*x*)
x の余弦を返します。

hypot(*x*, *y*)
ユークリッド距離 (sqrt(*x***x* + *y***y*)) を返します。

sin(*x*)
x の正弦を返します。

tan(*x*)
x の正接を返します。

以下は角度に関する関数です:

degrees(*x*)
角 *x* をラジアンから度数に変換します。

radians(*x*)
角 *x* を度数からラジアンに変換します。

以下は双曲線関数です:

cosh(*x*)
x の双曲線余弦を返します。

sinh(*x*)
x の双曲線正弦を返します。

tanh(*x*)
x の双曲線正接を返します。

このモジュールでは以下の 2 つの数学定数も定義しています:

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パでできています。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となっていません。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 (6.2 節):
これらの多くの関数の複素数版。

6.2 cmath — 複素数のための数学関数

このモジュールは常に利用できます。このモジュールでは、複素数を扱う数学関数へのアクセス手段を提供しています。

提供している関数を以下に示します:

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 ∞ 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 -∞ へと延びていて、上から連続しています。

acosh(*x*)
: *x* の逆双曲線余弦を返します。branch cut が一つあり、1 の左側に実数軸に沿って -∞ 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* の双曲線正弦を返します。2 つの brnch cut があり、±`1j` の左から ±-∞`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 ∞`j` までで右から連続、もう一方は -`1j` から下って -∞`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 ∞`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-∞`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* の逆双曲線正接を返します。2 つの branch cut があります: 一つは 1 から実数軸に沿って ∞ までで、上で連続です。もう一方は -1 から実数軸に沿って -∞ 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* の双曲線余弦を返します。

exp(*x*)
: 指数値 `e**`*x* を返します。

log[**, base**](*x*)
: *base* を底とする *x* の対数を返します。もし *base* が指定されていない場合には、*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 -∞ に延びており、上で連続しています。 2.4 で変更された仕様: 引数 *base* が追加されました。

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。

`math` と同じような関数が選ばれていますが、全く同じではないので注意してください。機能を二つのモジュールに分けているのは、複素数に興味がなかったり、もしかすると複素数とは何かすら知らないようなユーザがいるからです。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してほしいと考えます。また、`cmath` で定義されている関数は、たとえ結果が実数で表現可能な場合 (虚数部分がゼロの複素数) でも、常に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.

6.3 `decimal` — 10 進浮動小数点数の算術演算

2.4 で追加された仕様です。

`decimal` モジュールは 10 進の浮動小数点算術をサポートします。`decimal` には、 `float()` データ型に比べて、以下のような利点があります:

- 10 進数を正確に表現できます。`1.1` のような数は、2 進数の浮動小数点型では正しく表現できません。エンドユーザは普通、2 進数における `1.1` の近似値が `1.1000000000000001` だからといって、そのように表示してほしいとは考えないものです。

- 値の正確さは算術にも及びます。10 進の浮動小数点による計算では、'`0.1 + 0.1 + 0.1 - 0.3`' は厳密にゼロに等しくなります。2 進浮動小数点では `5.5511151231257827e-017` になってしまいます。ゼロに近い値とはいえ、この誤差は数値間の等価性テストの信頼性を阻害します。また、誤差が蓄積されることもあります。こうした理由から、数値間の等価性を厳しく保たねばならないようなアプリケーションを考えるなら、10 進数による数値表現が望ましいということになります。

- `decimal` モジュールでは、有効桁数の表記が取り入れられており、例えば '`1.30 + 1.20`' は `2.50` になります。すなわち、末尾のゼロは有効数字を示すために残されます。こうした仕様は通貨計算を行うアプリケーションでは慣例です。乗算の場合、「教科書的な」アプローチでは、乗算の被演算子すべての桁数を使います。例えば、'`1.3 * 1.2`' は `1.56` になり、'`1.30 * 1.20`' は `1.5600` になります。

- ハードウェアによる 2 進浮動小数点表現と違い、decimal モジュールでは計算精度をユーザが指定できます (デフォルトでは 28 桁です)。この桁数はほとんどの問題解決に十分な大きさです:

```
>>> getcontext().prec = 6
>>> Decimal(1) / Decimal(7)
Decimal("0.142857")
>>> getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal("0.1428571428571428571428571429")
```

- 2 進と 10 進の浮動小数点は、いずれも広く公開されている標準仕様のもとに実装されています。組み込みの浮動小数点型では、標準仕様で提唱されている機能のほんのささやかな部分を利用できるにすぎませんが、 decimal では標準仕様が要求している全ての機能を利用できます。必要に応じて、プログラマは値の丸めやシグナル処理を完全に制御できます。

このモジュールは、 10 進数型、算術コンテキスト (context for arithmetic)、そしてシグナル (signal) という三つの概念を中心に設計されています、

10 進数型は変更不可能な型です。この型には符号部、仮数部、そして指数部があります。有効桁数を残すために、仮数部の末尾にあるゼロの切り詰めは行われません。 decimal では、Infinity, -Infinity, および NaN といった特殊な値も定義されています。標準仕様では -0 と +0 も区別しています。

算術コンテキストとは、精度や値丸めの規則、指数部の制限を決めている環境です。この環境では、演算結果を表すためのフラグや、演算上発生した特定のシグナルを例外として扱うかどうかを決めるトラップイネーブラも定義しています。丸め規則には ROUND_CEILING, ROUND_DOWN, ROUND_FLOOR, ROUND_HALF_DOWN, ROUND_HALF_EVEN, ROUND_HALF_UP, および ROUND_UP があります。

シグナルとは、演算の過程で生じる例外的条件です。個々のシグナルは、アプリケーションそれぞれの要求に従って、無視されたり、単なる情報とみなされたり、例外として扱われたりします。decimal モジュールには、Clamped, InvalidOperation, DivisionByZero, Inexact, Rounded, Subnormal, Overflow, および Underflow といったシグナルがあります。

各シグナルには、フラグとトラップイネーブラがあります。演算上何らかのシグナルに遭遇すると、フラグはゼロからインクリメントされてゆきます。このとき、もしトラップイネーブラが 1 にセットされていれば、例外を送出します。フラグの値は膠着型 (sticky) なので、演算によるフラグの変化をモニタしたければ、予めフラグをリセットしておかねばなりません。

参考資料:

IBM による汎用 10 進演算仕様、*The General Decimal Arithmetic Specification*。

IEEE 標準化仕様 854-1987, *IEEE 854* **に関する非公式のテキスト**。

6.3.1 Quick-start Tutorial

普通、decimal を使うときには、モジュールを import し、現在の演算コンテキストを getcontext() で調べ、必要に応じて精度や丸めを設定し、演算エラーのトラップを有効にします:

```
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999999, Emax=999999999,
        capitals=1, flags=[], traps=[Overflow, InvalidOperation,
        DivisionByZero])

>>> getcontext().prec = 7       # 新たな精度を設定
```

Decimal のインスタンスは、整数、文字列またはタプルから生成できます。Decimal を float から生成したければ、まず文字列型に変換せねばなりません。そうすることで、変換方法の詳細を (representation error も含めて) 明示的に残せます。Decimal は “数値ではない (Not a Number)” を表す NaN や正負の Infinity (無限大)、-0 といった特殊な値も表現できます。

```
>>> Decimal(10)
Decimal("10")
>>> Decimal("3.14")
Decimal("3.14")
>>> Decimal((0, (3, 1, 4), -2))
Decimal("3.14")
>>> Decimal(str(2.0 ** 0.5))
Decimal("1.41421356237")
>>> Decimal("NaN")
Decimal("NaN")
>>> Decimal("-Infinity")
Decimal("-Infinity")
```

新たな Decimal 型数値の有効桁数は入力した数の桁数だけで決まります。演算コンテキストにおける精度や値丸めの設定が影響するのは算術操作の中だけです。

```
>>> getcontext().prec = 6
>>> Decimal('3.0')
Decimal("3.0")
>>> Decimal('3.1415926535')
Decimal("3.1415926535")
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal("5.85987")
>>> getcontext().rounding = ROUND_UP
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal("5.85988")
```

Decimal 型数値はほとんどの場面で Python の他の機能とうまくやりとりできます。Decimal 浮動小数点小劇場 (flying circus) を示しましょう:

```
>>> data = map(Decimal, '1.34 1.87 3.45 2.35 1.00 0.03 9.25'.split())
>>> max(data)
Decimal("9.25")
>>> min(data)
Decimal("0.03")
>>> sorted(data)
[Decimal("0.03"), Decimal("1.00"), Decimal("1.34"), Decimal("1.87"),
 Decimal("2.35"), Decimal("3.45"), Decimal("9.25")]
>>> sum(data)
Decimal("19.29")
>>> a,b,c = data[:3]
>>> str(a)
'1.34'
>>> float(a)
1.3400000000000001
>>> round(a, 1)      # round() は値をまず二進の浮動小数点数に変換します
1.3
>>> int(a)
1
>>> a * 5
Decimal("6.70")
>>> a * b
Decimal("2.5058")
>>> c % a
Decimal("0.77")
```

quantize() メソッドは位を固定して数値を丸めます。このメソッドは、計算結果を固定の桁数で丸めることがよくある、通貨を扱うアプリケーションで便利です:

```
>>> Decimal('7.325').quantize(Decimal('.01'), rounding=ROUND_DOWN)
Decimal("7.32")
>>> Decimal('7.325').quantize(Decimal('1.'), rounding=ROUND_UP)
Decimal("8")
```

前述のように、getcontext() 関数を使うと現在の演算コンテキストにアクセスでき、設定を変更できます。ほとんどのアプリケーションはこのアプローチで十分です。

より高度な作業を行う場合、Context() コンストラクタを使って別の演算コンテキストを作っておくと便利なことがあります。別の演算コンテキストをアクティブにしたければ、setcontext() を使います。

Decimal モジュールでは、標準仕様に従って、すぐ利用できる二つの標準コンテキスト、BasicContext および ExtendedContext を提供しています。後者はほとんどのトラップが有効になっており、とりわけデバッグの際に便利です:

```
>>> myothercontext = Context(prec=60, rounding=ROUND_HALF_DOWN)
>>> setcontext(myothercontext)
>>> Decimal(1) / Decimal(7)
Decimal("0.142857142857142857142857142857142857142857142857142857142857142857")

>>> ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999999, Emax=999999999,
        capitals=1, flags=[], traps=[])
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(7)
Decimal("0.142857143")
>>> Decimal(42) / Decimal(0)
Decimal("Infinity")

>>> setcontext(BasicContext)
>>> Decimal(42) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#143>", line 1, in -toplevel-
    Decimal(42) / Decimal(0)
DivisionByZero: x / 0
```

演算コンテキストには、演算中に遭遇した例外的状況をモニタするためのシグナルフラグがあります。フラグが一度セットされると、明示的にクリアするまで残り続けます。そのため、フラグのモニタを行いたいような演算の前には `clear_flags()` メソッドでフラグをクリアしておくのがベストです。

```
>>> setcontext(ExtendedContext)
>>> getcontext().clear_flags()
>>> Decimal(355) / Decimal(113)
Decimal("3.14159292")
>>> getcontext()
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999999, Emax=999999999,
        capitals=1, flags=[Inexact, Rounded], traps=[])
```

flags エントリから、`Pi` の有理数による近似値が丸められた (コンテキスト内で決められた精度を超えた桁数が捨てられた) ことと、計算結果が厳密でない (無視された桁の値に非ゼロのものがあった) ことがわかります。

コンテキストの `traps` フィールドに入っている辞書を使うと、個々のトラップをセットできます:

```
>>> Decimal(1) / Decimal(0)
Decimal("Infinity")
>>> getcontext().traps[DivisionByZero] = 1
>>> Decimal(1) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#112>", line 1, in -toplevel-
    Decimal(1) / Decimal(0)
DivisionByZero: x / 0
```

ほとんどのプログラムでは、開始時に一度だけ現在の演算コンテキストを修正します。また、多くのアプリケーションでは、データから `Decimal` への変換はループ内で一度だけキャストして行います。コンテキストを設定し、`Decimal` オブジェクトを生成できたら、ほとんどのプログラムは他の Python 数値型と全く変わらないかのように `Decimal` を操作できます。

6.3.2 Decimal オブジェクト

class Decimal(*[value [, context]]*)

value に基づいて新たな Decimal オブジェクトを構築します。

value は整数、文字列、タプル、および他の Decimal オブジェクトにできます。*value* を指定しない場合、Decimal("0") を返します。 *value* が文字列の場合、以下の 10 進数文字列の文法に従わねばなりません:

```
sign           ::=  '+' | '-'
digit          ::=  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
indicator      ::=  'e' | 'E'
digits         ::=  digit [digit]...
decimal-part   ::=  digits '.' [digits] | ['.'] digits
exponent-part  ::=  indicator [sign] digits
infinity       ::=  'Infinity' | 'Inf'
nan            ::=  'NaN' [digits] | 'sNaN' [digits]
numeric-value  ::=  decimal-part [exponent-part] | infinity
numeric-string ::=  [sign] numeric-value | [sign] nan
```

value を tuple にする場合、タプルは三つの要素を持ち、それぞれ符号 (正なら 0、負なら 1)、仮数部を表す数字のタプル、そして指数を表す整数でなければなりません。例えば、'Decimal((0, (1, 4, 1, 4), -3))' は Decimal("1.414") を返します。

context に指定した精度 (precision) は、オブジェクトが記憶する桁数には影響しません。桁数は *value* に指定した桁数だけから決定されます。例えば、演算コンテキストに指定された精度が 3 桁しかなくても、'Decimal("3.00000")' は 5 つのゼロを全て記憶します。

context 引数の目的は、*value* が正しくない形式の文字列であった場合に行う処理を決めることにあります; 演算コンテキストが InvalidOperation をトラップするようになっていれば、例外を送出します。それ以外の場合には、コンストラクタは値が NaN の Decimal を返します。

一度生成すると、Decimal オブジェクトは変更不能 (immutable) になります。

10 進浮動小数点オブジェクトは、float や int のような他の組み込み型と多くの点で似ています。通常の数学演算や特殊メソッドを適用できます。また、Decimal オブジェクトはコピーでき、pickle 化でき、print で出力でき、辞書のキーにでき、集合の要素にでき、比較、保存、他の型 (float や long) への型強制を行えます。

こうした標準的な数値型の特性の他に、10 進浮動小数点オブジェクトには様々な特殊メソッドがあります:

adjusted()

仮数部の先頭の一桁だけが残るように桁シフトを行い、そのときの指数部を返します: Decimal("321e+5").adjusted() なら 7 です。最上桁の小数点からの相対位置を調べる際に使います。

as_tuple()

数値を表現するためのタプル: '(sign, digittuple, exponent)' を返します。

compare(*other*[, *context*])

__cmp__() に似ていますが、Decimal インスタンスを返します。

```
a or b is a NaN ==> Decimal("NaN")
a < b           ==> Decimal("-1")
a == b          ==> Decimal("0")
a > b           ==> Decimal("1")
```

max(*other*[, *context*])
‘max(self, other)’と同じですが、値を返す前に現在のコンテキストに即した丸め規則を適用します。また、NaN に対して、(コンテキストでシグナルまたは黙認のどちらが設定されているかに応じて) シグナルを発行するか無視します。

min(*other*[, *context*])
‘min(self, other)’と同じですが、値を返す前に現在のコンテキストに即した丸め規則を適用します。また、NaN に対して、(コンテキストでシグナルまたは黙認のどちらが設定されているかに応じて) シグナルを発行するか無視します。

normalize([*context*])
数値を正規化 (normalize) して、右端に連続しているゼロを除去し、Decimal("0") と同じ結果はすべて Decimal("0e0") に変換します。同じクラスの値から基準表現を生成する際に用います。たとえば、 Decimal("32.100") と Decimal("0.321000e+2") の正規化は、いずれも同じ値 Decimal("32.1") になります。

quantize(*exp* [, *rounding*[, *context*[, *watchexp*]]])
指数部を *exp* と同じにします。値丸めの際、まず *rounding* があるか調べ、次に *context* を調べ、最後に現在のコンテキストの設定を用います。

watchexp が (default) に設定されている場合、処理結果の指数が Emax よりも大きい場合や Etiny よりも小さい場合にエラーを返します。

remainder_near(*other*[, *context*])
モジュロを計算し、正負のモジュロのうちゼロに近い値を返します。たとえば、‘Decimal(10).remainder_near(6)’ は Decimal("4") よりもゼロに近い値 Decimal("-2") を返します。

ゼロからの差が同じ場合には、 *self* と同じ符号を持った方を返します。

same_quantum(*other*[, *context*])
self と *other* が同じ指数を持っているか、あるいは双方とも NaN である場合に真を返します。

sqrt([*context*])
平方根を精度いっぱいまで求めます。

to_eng_string([*context*])
数値を工学で用いられる形式 (工学表記; enginnering notation) の文字列に変換します。

工学表記では指数は 3 の倍数になります。従って、最大で 3 桁までの数字が基数の小数部に現れます。たとえば、Decimal('123E+1') は Decimal("1.23E+3") に変換されます。

to_integral([*rounding*[, *context*]])
Inexact や Rounded といったシグナルを出さずに最近傍の整数に値を丸めます。*rounding* が指定されていれば適用されます; それ以外の場合、値丸めの方法は *context* の設定か現在のコンテキストの設定になります。

6.3.3 Context オブジェクト

コンテキスト (context) とは、算術演算における環境設定です。コンテキストは計算精度を決定し、値丸めの方法を設定し、シグナルのどれが例外になるかを決め、指数の範囲を制限しています。

多重スレッドで処理を行う場合には各スレッドごとに現在のコンテキストがあり、getcontext() や setcontext() といった関数でアクセスしたり設定変更できます:

getcontext()
アクティブなスレッドの現在のコンテキストを返します。

setcontext(*c*)

アクティブなスレッドのコンテキストを *c* に設定します。

Python 2.5 から、`with` 文と `localcontext()` 関数を使って実行するコンテキストを一時的に変更することもできるようになりました。

localcontext([*c*])

with 文の入口でアクティブなスレッドのコンテキストを *c* のコピーに設定し、with 文を抜ける時に元のコンテキストに復旧する、コンテキストマネージャを返します。コンテキストが指定されなければ、現在のコンテキストのコピーが使われます。 2.5 で追加された仕様です。

たとえば、以下のコードでは精度を 42 桁に設定し、計算を実行し、そして元のコンテキストに復帰します。

```
from __future__ import with_statement
from decimal import localcontext

with localcontext() as ctx:
    ctx.prec = 42   # 高精度の計算を実行
    s = calculate_something()
s = +s  # 最終的な結果をデフォルトの精度に丸める
```

新たなコンテキストは、以下で説明する `Context` コンストラクタを使って生成できます。その他にも、`decimal` モジュールでは作成済みのコンテキストを提供しています:

class BasicContext

General Decimal Arithmetic Specification で定義されている標準コンテキストの一つです。精度は 9 桁に設定されています。丸め規則は `ROUND_HALF_UP` です。すべての演算結果フラグはクリアされています。`Inexact`、`Rounded`、`Subnormal` を除く全ての演算エラートラップが有効 (例外として扱う) になっています。

多くのトラップが有効になっているので、デバッグの際に便利なコンテキストです。

class ExtendedContext

General Decimal Arithmetic Specification で定義されている標準コンテキストの一つです。精度は 9 桁に設定されています。丸め規則は `ROUND_HALF_EVEN` です。すべての演算結果フラグはクリアされています。トラップは全て無効 (演算中に一切例外を送出しない) になっています。

トラップが無効になっているので、エラーの伴う演算結果を `NaN` や `Infinity` にし、例外を送出しないようにしたいアプリケーションに向いたコンテキストです。このコンテキストを使うと、他の場合にはプログラムが停止してしまうような状況があっても実行を完了させられます。

class DefaultContext

`Context` コンストラクタが新たなコンテキストを作成するさいに雛形にするコンテキストです。このコンテキストのフィールド (精度の設定など) を変更すると、`Context` コンストラクタが生成する新たなコンテキストに影響を及ぼします。

このコンテキストは、主に多重スレッド環境で便利です。スレッドを開始する前に何らかのフィールドを変更しておくと、システム全体のデフォルト設定に効果を及ぼせます。スレッドを開始した後にフィールドを変更すると競合条件を抑制するためにスレッドを同期化せねばならないので推奨しません。

単一スレッドの環境では、このコンテキストを使わないよう薦めます。下で述べるように明示的にコンテキストを作成してください。

デフォルトの値は精度 28 桁、丸め規則 `ROUND_HALF_EVEN` で、トラップ `Overflow`、`InvalidOperation`、および `DivisionByZero` が有効になっています。

上に挙げた三つのコンテキストに加え、Context コンストラクタを使って新たなコンテキストを生成できます。

class Context(*prec=None, rounding=None, traps=None, flags=None, Emin=None, Emax=None, capitals=1*)

新たなコンテキストを生成します。あるフィールドが定義されていないか None であれば、DefaultContext からデフォルト値をコピーします。*flags* フィールドが設定されていいか None の場合には、全てのフラグがクリアされます。

prec フィールドは正の整数で、コンテキストにおける算術演算の計算精度を設定します。

rounding は、

- ROUND_CEILING (Infinity 寄りの値にする),
- ROUND_DOWN (ゼロ寄りの値にする),
- ROUND_FLOOR (-Infinity 寄りの値にする),
- ROUND_HALF_DOWN (最近値のうちゼロ寄りの値にする),
- ROUND_HALF_EVEN (最近値のうち偶数値を優先する),
- ROUND_HALF_UP (最近値のうちゼロから遠い値にする), または
- ROUND_UP (ゼロから遠い値にする)

のいずれかです。

traps および *flags* フィールドには、セットしたいシグナルを列挙します。一般的に、新たなコンテキストを作成するときにはトラップだけを設定し、フラグはクリアしておきます。

Emin および *Emax* フィールドには、指数範囲の外側値を整数で指定します。

capitals フィールドは 0 または 1 (デフォルト) にします。1 に設定すると、指数記号を大文字 E で出力します。それ以外の場合には Decimal('6.02e+23') のように e を使います。

Context クラスでは、いくつかの汎用のメソッドの他、現在のコンテキストで算術演算を直接行うためのメソッドを数多く定義しています。

clear_flags()

フラグを全て 0 にリセットします。

copy()

コンテキストの複製を返します。

create_decimal(*num*)

self をコンテキストとする新たな Decimal インスタンスを *num* から生成します。Decimal コンストラクタと違い、数値を変換する際にコンテキストの精度、値丸め方法、フラグ、トラップを適用します。

定数値はしばしばアプリケーションの要求よりも高い精度を持っているため、このメソッドが役に立ちます。また、値丸めを即座に行うため、例えば以下のように、入力値に値丸めを行わないために合計値にゼロの加算を追加するだけで結果が変わってしまうといった、現在の精度よりも細かい値の影響が紛れ込む問題を防げるという恩恵もあります。

```
>>> getcontext().prec = 3
>>> Decimal("3.4445") + Decimal("1.0023")
Decimal("4.45")
>>> Decimal("3.4445") + Decimal(0) + Decimal("1.0023")
Decimal("4.44")
```

Etiny()

'Emmin - prec + 1' に等しい値を返します。演算結果の劣化が起こる桁の最小値です。アンダーフローが起きた場合、指数は Etiny に設定されます。

Etop()

'Emax - prec + 1' に等しい値を返します。

Decimal を使った処理を行う場合、通常は Decimal インスタンスを生成して、算術演算を適用するというアプローチをとります。演算はアクティブなスレッドにおける現在のコンテキストの下で行われます。もう一つのアプローチは、コンテキストのメソッドを使った特定のコンテキスト下での計算です。コンテキストのメソッドは Decimal クラスのメソッドに似ているので、ここでは簡単な説明にとどめます。

abs(*x*)

x の絶対値を返します。

add(*x*, *y*)

x と *y* の加算を返します。

compare(*x*, *y*)

二つの値を数値として比較します。

__cmp__() に似ていますが、 以下のように Decimal インスタンスを返します:

```
a or b is a NaN ==> Decimal("NaN")
a < b           ==> Decimal("-1")
a == b          ==> Decimal("0")
a > b           ==> Decimal("1")
```

divide(*x*, *y*)

x を *y* で除算した値を返します。

divmod(*x*, *y*)

二つの数値間の除算を行い、結果の整数部を返します。

max(*x*, *y*)

二つの値を数値として比較し、大きいほうを返します。

数値上二つの値が等しい場合には、左側値を結果として返します。

min(*x*, *y*)

二つの値を数値として比較し、小さいほうを返します。

数値上二つの値が等しい場合には、左側値を結果として返します。

minus(*x*)

Python における単項の符号反転前置演算子 (unary prefix minus operator) に対応する演算です。

multiply(*x*, *y*)

x と *y* の積を返します。

normalize(*x*)

被演算子をもっとも単純な表記にします。

本質的には、plus 演算の結果から末尾のゼロを全て取り除いたものと同じです。

plus(*x*)

Python における単項の符号非反転前置演算子 (unary prefix plus operator) に対応する演算です。コンテキストにおける精度や値丸めを適用するので、等値 (identity) 演算とは **違います**。

power(*x*, *y*[, *modulo*])

'x ** y' を計算します。*modulo* が指定されていれば使います。

右被演算子は整数部が 9 桁以下で、小数部 (のある場合) は値丸め前に全てゼロになっていなければなりません。被演算子は正でも負でもゼロでもかまいません。右被演算子が負の場合には、左被演算子の逆数 (1 を左被演算子で割った値) を右被演算子の逆数でべき乗します。

中間演算でより高い計算精度が必要になり、その精度が実装の提供している精度を超えた場合、`InvalidOperation` エラーをシグナルします。

負のべき乗を行う際に 1 への除算でアンダーフローが起きても、その時点では演算を停止せず継続します。

quantize (*x, y*)

x に値丸めを適用し、指数を *y* にした値を返します。

他の演算と違い、量子化後の係数の長さが精度よりも大きい場合には `InvalidOperation` をシグナルします。このため、エラーが生じないかぎり、量子化後の指数は右側の被演算子の指数と等しくなることが保証されます。

また、結果が劣化していたり不正確な値であっても、`Underflow` をシグナルしないという点も他の演算と異なります。

remainder (*x, y*)

整数除算の剰余を返します。

剰余がゼロでない場合、符号は割られる数の符号と同じになります。

remainder_near (*x, y*)

モジュロを計算し、正負のモジュロのうちゼロに近い値を返します。たとえば、'`Decimal(10).remainder_near(6)`' は `Decimal("4")` よりもゼロに近い値 `Decimal("-2")` を返します。

ゼロからの差が同じ場合には、 *self* と同じ符号を持った方を返します。

same_quantum (*x, y*)

self と *other* が同じ指数を持っているか、あるいは双方とも `NaN` である場合に真を返します。

sqrt (*x*)

x の平方根を精度いっぱいまで求めます。

subtract (*x, y*)

x と *y* の間の差を返します。

to_eng_string ()

工学表記で文字列に変換します。

工学表記では指数は 3 の倍数になります。従って、最大で 3 桁までの数字が基数の小数部に現れます。たとえば、`Decimal('123E+1')` は `Decimal("1.23E+3")` に変換されます。

to_integral (*x*)

`Inexact` や `Rounded` といったシグナルを出さずに最近傍の整数に値を丸めます。

to_sci_string (*x*)

数値を科学表記で文字列に変換します。

6.3.4 シグナル

シグナルは、計算中に生じた様々なエラー条件を表現します。各々のシグナルは一つのコンテキストフラグと一つのトラップイネーブラに対応しています。

コンテキストフラグは、該当するエラー条件に遭遇するたびに加算されてゆきます。演算後にフラグを調べれば、演算に関する情報 (例えば計算が厳密だったかどうか) がわかります。フラグを調べたら、次の計算を始める前にフラグを全てクリアするようにしてください。

あるコンテキストのトラップイネーブラがあるシグナルに対してセットされている場合、該当するエラー条件が生じると Python の例外を送出します。例えば、DivisionByZero が設定されていると、エラー条件が生じた際に DivisionByZero 例外を送出します。

class Clamped

値の表現上の制限に沿わせるために指数部が変更されたことを通知します。

通常、クランプ (clamp) は、指数部がコンテキストにおける指数桁の制限値 Emin および Emax を越えたなった場合に発生します。可能な場合には、係数部にゼロを加えた表現に合わせて指数部を減らします。

class DecimalException

他のシグナルの基底クラスで、ArithmeticError のサブクラスです。

class DivisionByZero

有限値をゼロで除算したときのシグナルです。

除算やモジュロ除算、数を負の値で累乗した場合に起きることがあります。このシグナルをトラップしない場合、演算結果は Infinity または-Infinity になり、その符号は演算に使った入力に基づいて決まります。

class Inexact

値の丸めによって演算結果から厳密さが失われたことを通知します。

このシグナルは値丸め操作中にゼロでない桁を無視した際に生じます。演算結果は値丸め後の値です。シグナルのフラグやトラップは、演算結果の厳密さが失われたことを検出するために使えるだけです。

class InvalidOperation

無効な演算が実行されたことを通知します。

ユーザが有意な演算結果にならないような操作を要求したことを示します。このシグナルをトラップしない場合、NaN を返します。このシグナルの発生原因として考えられるのは、以下のような状況です:

```
Infinity - Infinity
0 * Infinity
Infinity / Infinity
x % 0
Infinity % x
x._rescale( non-integer )
sqrt(-x) and x > 0
0 ** 0
x ** (non-integer)
x ** Infinity
```

class Overflow

数値オーバフローを示すシグナルです。

このシグナルは、値丸めを行った後の指数部が Emax より大きいことを示します。シグナルをトラップしない場合、演算結果は値丸めのモードにより、表現可能な最大の数値になるように内側へ引き込んで丸めを行った値か、Infinity になるように外側に丸めた値のいずれかになります。いずれの場合も、Inexact および Rounded が同時にシグナルされます。

class Rounded

情報が全く失われていない場合も含み、値丸めが起きたときのシグナルです。

このシグナルは、値丸めによって桁がなくなると常に発生します。なくなった桁がゼロ (例えば 5.00

を丸めて 5.0 になった場合) であってもです。このシグナルをトラップしなければ、演算結果をそのまま返します。このシグナルは有効桁数の減少を検出する際に使います。

class Subnormal

値丸めを行う前に指数部が Emin より小さかったことを示すシグナルです。

演算結果が微小である場合 (指数が小さすぎる場合) に発生します。このシグナルをトラップしなければ、演算結果をそのまま返します。

class Underflow

演算結果が値丸めによってゼロになった場合に生じる数値アンダフローです。

演算結果が微小なため、値丸めによってゼロになった場合に発生します。Inexact および Subnormal シグナルも同時に発生します。

これらのシグナルの階層構造をまとめると、以下の表のようになります:

```
exceptions.ArithmeticError(exceptions.StandardError)
    DecimalException
        Clamped
        DivisionByZero(DecimalException, exceptions.ZeroDivisionError)
        Inexact
            Overflow(Inexact, Rounded)
            Underflow(Inexact, Rounded, Subnormal)
        InvalidOperation
        Rounded
        Subnormal
```

6.3.5 浮動小数点数に関する注意

精度を上げて丸め誤差を抑制する

10 進浮動小数点数を使うと、 10 進数表現による誤差を抑制できます (0.1 を正確に表現できるようになります); しかし、ゼロでない桁が一定の精度を越えている場合には、演算によっては依然として値丸めによる誤差を引き起こします。 Knuth は、十分でない計算精度の下で値丸めを伴う浮動小数点演算を行った結果、加算の結合則や分配則における恒等性が崩れてしまう例を二つ示しています:

```
# Examples from Seminumerical Algorithms, Section 4.2.2.
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 8

>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal("9.5111111")
>>> u + (v + w)
Decimal("10")

>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal("0.01")
>>> u * (v+w)
Decimal("0.0060000")
```

decimal モジュールでは、最下桁を失わないように十分に計算精度を広げることで、上で問題にしたような恒等性をとりもどせます:

```
>>> getcontext().prec = 20
>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal("9.51111111")
>>> u + (v + w)
Decimal("9.51111111")
>>>
>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal("0.0060000")
>>> u * (v+w)
Decimal("0.0060000")
```

特殊値

decimal モジュールの数体系では、NaN, sNaN, -Infinity , Infinity , および二つのゼロ、+0 と -0 といった特殊な値を提供しています。

無限大 (Infinity) は Decimal('Infinity') で直接構築できます。また、DivisionByZero をトラップせずにゼロで除算を行った場合にも出てきます。同様に、 Overflow シグナルをトラップしなければ、表現可能な最大の数値の制限を越えた値を丸めたときに出てきます。

無限大には符号があり (アフィン: affine であり)、算術演算に使用でき、非常に巨大で不確定の (indeterminate) 値として扱われます。例えば、無限大に何らかの定数を加算すると、演算結果は別の無限大になります。

演算によっては結果が不確定になるものがあり、NaN を返します。ただし、InvalidOperation シグナルをトラップするようになっていれば例外を送出します。

例えば、0/0 は NaN を返します。NaN は「非数値 (not a number)」を表します。このような NaN は暗黙のうちに生成され、一度生成されるとそれを他の計算にも流れてゆき、関係する個々の演算全てが個別の NaN を返すようになります。この挙動は、たまに入力値が欠けるような状況で一連の計算を行う際に便利です — 特定の計算に対しては無効な結果を示すフラグを立てつつ計算を進められるからです。

一方、NaN の変種である sNaN は関係する全ての演算で演算後にシグナルを送出します。sNaN は、無効な演算結果に対して特別な処理を行うために計算を停止する必要がある場合に便利です。

アンダフローの起きた計算は、符号付きのゼロ (signed zero) を返すことがあります。符号は、より高い精度で計算を行った結果の符号と同じになります。符号付きゼロの大きさはやはりゼロなので、正のゼロと負のゼロは等しいとみなされ、符号は単なる参考にすぎません。

二つの符号付きゼロが区別されているのに等価であることに加えて、異なる精度におけるゼロの表現はまちまちなのに、値は等価とみなされるということがあります。これに慣れるには多少時間がかかります。正規化浮動小数点表現に目が慣れてしまうと、以下の計算でゼロに等しい値が返っているとは即座に分かりません:

```
>>> 1 / Decimal('Infinity')
Decimal("0E-1000000026")
```

6.3.6 スレッドを使った処理

関数 getcontext() は、スレッド毎に別々の Context オブジェクトにアクセスします。別のスレッドコンテキストを持つということは、複数のスレッドが互いに影響を及ぼさずに (getcontext.prec=10

のような) 変更を適用できるということです。

同様に、setcontext() 関数は自動的に引数のコンテキストを現在のスレッドのコンテキストに設定します。

getcontext() を呼び出す前に setcontext() が呼び出されていなければ、現在のスレッドで使うための新たなコンテキストを生成するために getcontext() が自動的に呼び出されます。

新たなコンテキストは、*DefaultContext* と呼ばれる雛形からコピーされます。アプリケーションを通じて全てのスレッドに同じ値を使うようにデフォルトを設定したければ、*DefaultContext* オブジェクトを直接変更します。getcontext() を呼び出すスレッド間で競合条件が生じないようにするため、*DefaultContext* への変更はいかなるスレッドを開始するよりも**前に** 行わねばなりません。以下に例を示します:

```
# スレッドを立ち上げる前にアプリケーションにわたるデフォルトを設定
DefaultContext.prec = 12
DefaultContext.rounding = ROUND_DOWN
DefaultContext.traps = ExtendedContext.traps.copy()
DefaultContext.traps[InvalidOperation] = 1
setcontext(DefaultContext)

# その後でスレッドを開始
t1.start()
t2.start()
t3.start()
 . . .
```

6.3.7 レシピ

Decimal クラスの利用を実演している例をいくつか示します。これらはユーティリティ関数としても利用できます:

```
def moneyfmt(value, places=2, curr='', sep=',', dp='.',
             pos='', neg='-', trailneg=''):
    """Decimal を通貨表現の文字列に変換します。

    places:  小数点以下の値を表すのに必要な桁数
    curr:    符号の前に置く通貨記号 (オプションで、空でもかまいません)
    sep:     桁のグループ化に使う記号、オプションです (コンマ、ピリオド、
             スペース、または空)
    dp:      小数点 (コンマまたはピリオド)
             小数部がゼロの場合には空にできます。
    pos:     正数の符号オプション: '+', 空白または空文字列
    neg:     負数の符号オプション: '-', '(', 空白または空文字列
    trailneg:後置マイナス符号オプション:  '-', ')', 空白または空文字列

    >>> d = Decimal('-1234567.8901')
    >>> moneyfmt(d, curr='$')
    '-$1,234,567.89'
    >>> moneyfmt(d, places=0, sep='.', dp='', neg='', trailneg='-')
    '1.234.568-'
    >>> moneyfmt(d, curr='$', neg='(', trailneg=')')
    '($1,234,567.89)'
    >>> moneyfmt(Decimal(123456789), sep=' ')
    '123 456 789.00'
    >>> moneyfmt(Decimal('-0.02'), neg='<', trailneg='>')
    '<.02>'

    """
    q = Decimal((0, (1,), -places))    # 小数点以下 2 桁 --> '0.01'
    sign, digits, exp = value.quantize(q).as_tuple()
    assert exp == -places
    result = []
    digits = map(str, digits)
    build, next = result.append, digits.pop
    if sign:
        build(trailneg)
    for i in range(places):
        if digits:
            build(next())
        else:
            build('0')
    build(dp)
    i = 0
    while digits:
        build(next())
        i += 1
        if i == 3 and digits:
            i = 0
            build(sep)
    build(curr)
    if sign:
        build(neg)
    else:
        build(pos)
    result.reverse()
    return ''.join(result)

def pi():
    """現在の精度まで円周率を計算します。

    >>> print pi()
    3.141592653589793238462643383

    """
    getcontext().prec += 2  # 中間ステップのための余分の数字
    three = Decimal(3)      # 普通の float に対する "three=3.0" の代わり
    lasts, t, s, n, na, d, da = 0, three, 3, 1, 0, 0, 24
    while s != lasts:
        lasts = s
        n, na = n+na, na+8
```

6.3.8 Decimal FAQ

Q. decimal.Decimal('1234.5') などと打ち込むのは煩わしいのですが、対話式インタプリタを使う際にタイプ量を少なくする方法はありませんか?

A. コンストラクタを 1 文字に縮める人もいるようです。

```
>>> D = decimal.Decimal
>>> D('1.23') + D('3.45')
Decimal("4.68")
```

Q. 小数点以下 2 桁の固定小数点数のアプリケーションの中で、いくつかの入力が余計な桁を保持しているのでこれを丸めなければなりません。その他のものに余計な桁はなくそのまま使えます。どのメソッドを使うのがいいでしょうか?

A. quantize() メソッドで固定した桁に丸められます。Inexact トラップを設定しておけば、確認にも有用です。

```
>>> TWOPLACES = Decimal(10) ** -2       # Decimal('0.01') と同じ

>>> # 小数点以下 2 桁に丸める
>>> Decimal("3.214").quantize(TWOPLACES)
Decimal("3.21")

>>> # 小数点以下 2 桁を越える桁を保持していないことの確認
>>> Decimal("3.21").quantize(TWOPLACES, context=Context(traps=[Inexact]))
Decimal("3.21")

>>> Decimal("3.214").quantize(TWOPLACES, context=Context(traps=[Inexact]))
Traceback (most recent call last):
   ...
Inexact: Changed in rounding
```

Q. 正当な 2 桁の入力が得られたとして、その正当性をアプリケーション実行中も変わらず保ち続けるにはどうすればいいでしょうか?

A. 加減算のような演算は自動的に固定小数点を守ります。その他の乗除算などは小数点以下の桁を変えてしまいますので実行後は quantize() ステップが必要です。

Q. 一つの値に対して多くの表現方法があります。200 と 200.000 と 2E2 と .02E+4 は全て同じ値で違った精度の数です。これらをただ一つの正規化された値に変換することはできますか?

A. normalize() メソッドは全ての等しい値をただ一つの表現に直します。

```
>>> values = map(Decimal, '200 200.000 2E2 .02E+4'.split())
>>> [v.normalize() for v in values]
[Decimal("2E+2"), Decimal("2E+2"), Decimal("2E+2"), Decimal("2E+2")]
```

Q. ある種の 10 進数値はいつも指数表記で表示されます。指数表記以外の表示にする方法はありますか?

A. 値によっては、指数表記だけが有効桁数を表せる表記法なのです。たとえば、 5.0E+3 を 5000 と表してしまうと、値は変わりませんが元々の 2 桁という有効数字が反映されません。

Q. 普通の float を Decimal に変換できますか?

A. はい。どんな 2 進浮動小数点数も Decimal として正確に表現できます。正確な変換は直感的に考えたよりも多い桁になることもありますので、Inexact をトラップしたとすればそれはもっと精度を上げる必要性があることを示しています。

```
def floatToDecimal(f):
    "浮動小数点数を情報の欠落無く Decimal に変換します"

    # float で表された数を仮数 (0.5 <= abs(m) < 1.0) と指数に (正確に) 転
    # 換します。仮数を整数になるまで 2 倍し続けます。整数化した仮数と指数
    # を使って等価な Decimal を求めます。この手続きが正確に行なえなかっ
    # たら、精度を上げて再度同じことをします。

    mantissa, exponent = math.frexp(f)
    while mantissa != int(mantissa):
        mantissa *= 2.0
        exponent -= 1
    mantissa = int(mantissa)

    oldcontext = getcontext()
    setcontext(Context(traps=[Inexact]))
    try:
        while True:
            try:
               return mantissa * Decimal(2) ** exponent
            except Inexact:
                getcontext().prec += 1
    finally:
        setcontext(oldcontext)
```

Q. 上の `floatToDecimal()` はなぜモジュールに入っていないのですか?

A. 2 進と 10 進の浮動小数点数を混ぜるようにアドバイスするべきかどうか疑問があります。また、これを使うときには 2 進浮動小数点数の表示の問題を避けるように注意しなければなりません。

```
>>> floatToDecimal(1.1)
Decimal("1.100000000000000088817841970012523233890533447265625")
```

Q. 複雑な計算の中で、精度不足や丸めの異常で間違った結果になっていないことをどうやって保証すれば良いでしょうか?

A. decimal モジュールでは検算は容易です。一番良い方法は、大きめの精度や様々な丸めモードで再計算してみることです。大きく異なった結果が出てきたら、精度不足や丸めの問題や悪条件の入力、または数値計算的に不安定なアルゴリズムを示唆しています。

Q. コンテキストの精度は計算結果には適用されていますが入力には適用されていないようです。様々に異なる精度の入力値を混ぜて計算する時に注意すべきことはありますか?

A. はい。原則として入力値は正確であると見做しておりそれらの値を使った計算も同様です。結果だけが丸められます。入力の強みは “what you type is what you get” (打ち込んだ値が得られる値) という点にあります。入力が丸められないということを忘れていると結果が奇妙に見えるというのは弱点です。

```
>>> getcontext().prec = 3
>>> Decimal('3.104') + D('2.104')
Decimal("5.21")
>>> Decimal('3.104') + D('0.000') + D('2.104')
Decimal("5.20")
```

解決策は精度を上げるかまたは単項のプラス演算子を使って入力の丸めを強制することです。

```
>>> getcontext().prec = 3
>>> +Decimal('1.23456789')       # 単項のプラスで丸めを引き起こします
Decimal("1.23")
```

もしくは、入力を `Context.create_decimal()` を使って生成時に丸めてしまうこともできます。

```
>>> Context(prec=5, rounding=ROUND_DOWN).create_decimal('1.2345678')
Decimal("1.2345")
```

6.4 `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。シーケンス用には、シーケンス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。この関数は半開区間 [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列ができるだけ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()` をオーバライドしてください。オプションとして、新しいジェネレータは `getrandombits()` メソッドを提供できます — これにより `randrange()` メソッドが任意の、大きな範囲を超える集合を作成できるようになります。 2.4 で追加された仕様: `getrandombits()` メソッド

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。ただし、この Wichmann-Hill 生成器はもはや推奨することができないということに注意してください。現在の水準では生成される周期が短すぎ、また厳密な乱数性試験に合格しないことが知られています。こうした欠点を修正した最近の改良についてはページの最後に挙げた参考文献を参照してください。 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。

乱数の発生源をオペレーティングシステムが提供している場合、システム時刻の代わりにその発生源が使われます（詳細については `os.urandom()` 関数を参照）。 2.4 で変更された仕様: 通常、オペレーティングシステムのリソースは使われません

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

getrandbits(*k*)
乱数ビット *k* とともに Pyhon の `long` int を返します。このメソッドは MersenneTwister 生成器で提供されており、その他の乱数生成器でもオプションの API として提供されているかもしれません。このメソッドが使えるとき、`randrange()` メソッドは大きな範囲を扱えるようになります。 2.4 で追加された仕様です。

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 <= *N* <= *b* であるようなランダムな整数 *N* を返します。

シーケンス用の関数:

choice(*seq*)
空でないシーケンス *seq* からランダムに要素を返します。*seq* が空のときは、 `IndexError` が送出されます。

shuffle(*x*[, *random*])
シーケンス *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; このことは長いシーケンス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*)
母集団のシーケンスから選ばれた長さ *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()` オブジェクトを使いましょう。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 <= *N* <= *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 *alpha* > -1 および *beta* > -1 です。0 から 1 の値を返します。

expovariate(*lambd*)
指数分布です。*lambd* 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てます。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。

class **SystemRandom**([*seed*])

オペレーティングシステムの提供する発生源によって乱数を生成する `os.urandom()` 関数を使うクラスです。すべてのシステムで使えるメソッドではありません。ソフトウェアの状態に依存してはいけませんし、一連の操作は再現不能です。それに応じて、`seed()` と `jumpahead()` メソッドは何の影響も及ぼさず、無視されます。 `getstate()` と `setstate()` メソッドが呼び出されると、例外 `NotImplementedError` が送出されます。 2.4 で追加された仕様です。

基本使用例:

```
>>> random.random()         # Random float x, 0.0 <= x < 1.0
0.37444887175646646
>>> random.uniform(1, 10)   # Random float x, 1.0 <= x < 10.0
1.1800146073117523
>>> random.randint(1, 10)   # Integer from 1 to 10, endpoints included
7
>>> random.randrange(0, 101, 2)  # Even integer from 0 to 100
26
>>> random.choice('abcdefghij')  # Choose a random element
'c'

>>> items = [1, 2, 3, 4, 5, 6, 7]
>>> random.shuffle(items)
>>> items
[7, 3, 2, 5, 6, 4, 1]

>>> random.sample([1, 2, 3, 4, 5],  3)  # Choose 3 elements
[4, 1, 5]
```

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.

http://www.npl.co.uk/ssfm/download/abstracts.html#196

A modern variation of the Wichmann-Hill generator that greatly increases the period, and passes now-standard statistical tests that the original generator failed.

6.5 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て `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 で書いてもコンパイル言語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、高速になります。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*.

6.5.1 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か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
```

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

同じキーをもつような要素からなる *iterable* 中のグループに対して、キーとグループを返すようなイテレータを作成します。*key* は各要素に対するキー値を計算する関数です。キーを指定しない場合や `None` にした場合、*key* 関数のデフォルトは恒等関数になり要素をそのまま返します。通常、*iterable* は同じキー関数で並べ替え済みである必要があります。

返されるグループはそれ自体がイテレータで、`groupby()` と *iterable* を共有しています。もととなる *iterable* を共有しているため、`groupby` オブジェクトの要素取り出しを先に進めると、それ以前の要素であるグループは見えなくなってしまいます。従って、データが後で必要な場合にはリストの形で保存しておく必要があります:

```
groups = []
uniquekeys = []
for k, g in groupby(data, keyfunc):
    groups.append(list(g))      # Store group iterator as a list
    uniquekeys.append(k)
```

`groupby()` は以下のコードと等価です:

```
class groupby(object):
    def __init__(self, iterable, key=None):
        if key is None:
            key = lambda x: x
        self.keyfunc = key
        self.it = iter(iterable)
        self.tgtkey = self.currkey = self.currvalue = xrange(0)
    def __iter__(self):
        return self
    def next(self):
        while self.currkey == self.tgtkey:
            self.currvalue = self.it.next() # Exit on StopIteration
            self.currkey = self.keyfunc(self.currvalue)
        self.tgtkey = self.currkey
        return (self.currkey, self._grouper(self.tgtkey))
    def _grouper(self, tgtkey):
        while self.currkey == tgtkey:
            yield self.currvalue
            self.currvalue = self.it.next() # Exit on StopIteration
            self.currkey = self.keyfunc(self.currvalue)
```

2.4 で追加された仕様です。

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 funtion 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。シーケンス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)
    it = iter(xrange(s.start or 0, s.stop or sys.maxint, s.step or 1))
    nexti = it.next()
    for i, element in enumerate(iterable):
        if i == nexti:
            yield element
            nexti = it.next()
```

start が `None` ならば、繰返しは 0 から始まります。*step* が `None` ならば、ステップは 1 となります。2.5 で変更された仕様: *start* と *step* はデフォルト値として `None` を受け付けます。

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 = [it.next() for it in iterables]
        yield tuple(result)
```

2.4 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。

イテレート可能オブジェクトの左から右への評価順序は保証されることに注意して下さい。このことによって、データ列を長さ n のグループにまとめる常套句 '`izip(*[iter(s)]*n)`' が実現可能になります。長さ n のグループにまとめるのに中途半端なデータ列に対しては '`izip(*[chain(s, [None]*(n-1))]*n)`' のように、最後のタプルを埋める値をあらかじめ準備しておくことができます。

もう一つの注意は `izip()` が長さが不揃いの入力に対して呼ばれた時、`izip()` 終了の後引き続いて長い方のイテレート可能オブジェクトを呼び出した結果は保証の限りではないということです。可

能性として、残ったそれぞれのイテレート可能オブジェクトから値が一つ失われているかもしれないし失われていないかもしれません。これは次のようにして起こります。実行中にそれぞれのイテレート可能オブジェクトから一つずつ値を取り出しますが、その処理がいずれかのイテレート可能オブジェクトが空になることにより終了します。この時途中まで取り出された値たちは宙に浮きます (不完全なタプルとして送り出されることもなく、また次の `it.next()` のためにイテレート可能オブジェクトに押し戻すこともできません)。一般に、`izip()` を長さが不揃いな入力に使うのは、残され使われなかった長い方のイテレート可能オブジェクトの値を気にしない時だけにするべきです。

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 funtion 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
```

tee (*iterable*[, *n=2*])

一つの *iterable* から *n* 個の独立したイテレータを生成して返します。`n==2` の場合は、以下のコードと等価になります:

```
def tee(iterable):
    def gen(next, data={}, cnt=[0]):
        for i in count():
            if i == cnt[0]:
                item = data[i] = next()
                cnt[0] += 1
            else:
                item = data.pop(i)
            yield item
    it = iter(iterable)
    return (gen(it.next), gen(it.next))
```

一度 `tee()` でイテレータを分割すると、もとの *iterable* を他で使ってはならなくなるので注意してください; さもなければ、`tee` オブジェクトの知らない間に *iterable* が先の要素に進んでしまうことになります。

`tee` はかなり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)。一般には、一つのイテレータが他のイテレータよりも先にほとんどまたは全ての要素を消費するような場合には、`tee()` よりも `list()` を使った方が高速です。 2.4 で追加された仕様です。

6.5.2 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(1,5), repeat(3)):
...    print cube
...
1
8
27
64

>>> reportlines = ['EuroPython', 'Roster', '', 'alex', '', 'laura',
                   '', 'martin', '', 'walter', '', 'mark']
>>> for name in islice(reportlines, 3, None, 2):
...    print name.title()
...
Alex
Laura
Martin
Walter
Mark

# Show a dictionary sorted and grouped by value
>>> from operator import itemgetter
>>> d = dict(a=1, b=2, c=1, d=2, e=1, f=2, g=3)
>>> di = sorted(d.iteritems(), key=itemgetter(1))
>>> for k, g in groupby(di, key=itemgetter(1)):
...     print k, map(itemgetter(0), g)
...
1 ['a', 'c', 'e']
2 ['b', 'd', 'f']
3 ['g']

# Find runs of consecutive numbers using groupby.  The key to the solution
# is differencing with a range so that consecutive numbers all appear in
# same group.
>>> data = [ 1,  4,5,6, 10, 15,16,17,18, 22, 25,26,27,28]
>>> for k, g in groupby(enumerate(data), lambda (i,x):i-x):
...     print map(operator.itemgetter(1), g)
...
[1]
[4, 5, 6]
[10]
[15, 16, 17, 18]
[22]
[25, 26, 27, 28]
```

6.5.3 レシピ

この節では、既存の itertools をビルディングブロックとしてツールセットを拡張するためのレシピを示します。

iterable 全体を一度にメモリ上に置くよりも、要素を一つづつ処理する方がメモリ効率上の有利さを保てます。関数形式のままツールをリンクしてゆくと、コードのサイズを減らし、一時変数を減らす助けになります。インタプリタのオーバヘッドをもたらす for ループやジェネレータを使わずに、“ベクトル化された” ビルディングブロックを使うと、高速な処理を実現できます。

```
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=None):
    "Returns True if pred(x) is true for every element in the iterable"
    for elem in ifilterfalse(pred, seq):
        return False
    return True

def any(seq, pred=None):
    "Returns True if pred(x) is true for at least one element in the iterable"
    for elem in ifilter(pred, seq):
        return True
    return False

def no(seq, pred=None):
    "Returns True if pred(x) is false for every element in the iterable"
    return True not in imap(pred, seq)

def quantify(seq, pred=None):
    "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.

    Useful for emulating the behavior of the built-in map() function.
    """
    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 flatten(listOfLists):
    return list(chain(*listOfLists))

def repeatfunc(func, times=None, *args):
    """Repeat calls to func with specified arguments.

    Example:  repeatfunc(random.random)
    """
    if times is None:
        return starmap(func, repeat(args))
    else:
        return starmap(func, repeat(args, times))

def pairwise(iterable):
    "s -> (s0,s1), (s1,s2), (s2, s3), ..."
    a, b = tee(iterable)
    try:
        b.next()
```

6.6 functools — 高階関数と呼び出し可能オブジェクトの操作

2.5 で追加された仕様です。

モジュール functools は高階関数、つまり関数に対する関数、あるいは他の関数を返す関数、のためのものです。一般に、どんな呼び出し可能オブジェクトでもこのモジュールの目的には関数として扱えます。

モジュール functools では以下の関数を定義します。

partial (*func*[,**args*][, ***keywords*])

新しい partial オブジェクトを返します。このオブジェクトは呼び出されると位置引数 *args* とキーワード引数 *keywords* 付きで呼び出された *func* のように振る舞います。呼び出しに際してさらなる引数が渡された場合、それらは *args* に付け加えられます。追加のキーワード引数が渡された場合には、それらで *keywords* を拡張または上書きします。大雑把にいうと、次のコードと等価です。

```
def partial(func, *args, **keywords):
    def newfunc(*fargs, **fkeywords):
        newkeywords = keywords.copy()
        newkeywords.update(fkeywords)
        return func(*(args + fargs), **newkeywords)
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc
```

関数 partial は、関数の引数と/かキーワードの一部を「凍結」した部分適用として使われ、簡素化された引数形式をもった新たなオブジェクトを作り出します。例えば、partial を使って *base* 引数のデフォルトが 2 である int 関数のように振る舞う呼び出し可能オブジェクトを作ることができます。

```
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Convert base 2 string to an int.'
>>> basetwo('10010')
18
```

update_wrapper (*wrapper, wrapped*[, *assigned*][, *updated*])

wrapper 関数を wrapped 関数に見えるようにアップデートします。オプション引数はタプルで、元の関数のどの属性が wrapper 関数の一致する属性に直接書き込まれる (assigned) か、また wrapper 関数のどの属性が元の関数の対応する属性でアップデートされる (updated) か、を指定します。これらの引数のデフォルト値はモジュール定数 *WRAPPER_ASSIGNMENTS* (wrapper 関数に名前、モジュールそしてドキュメンテーション文字列を書き込みます) と *WRAPPER_UPDATES* (wrapper 関数のインスタンス辞書をアップデートします) です。

この関数は主に関数を包んで wrapper を返すデコレータ関数の中で使われるよう意図されています。もし wrapper 関数がアップデートされないとすると、返される関数のメタデータは元の関数の定義ではなく wrapper 関数の定義を反映してしまい、これは典型的に役立たずです。

wraps (*wrapped*[, *assigned*][, *updated*])

これはラッパ関数を定義するときに partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated) を関数デコレータとして呼び出す便宜関数です。

```
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwds):
...         print 'Calling decorated function'
...         return f(*args, **kwds)
...     return wrapper
...
>>> @my_decorator
... def example():
...     print 'Called example function'
...
>>> example()
Calling decorated function
Called example function
>>> example.__name__
'example'
```

このデコレータ・ファクトリーを使わなければ、上の例中の関数の名前は 'wrapper' となっているところです。

6.6.1 partial オブジェクト

partial オブジェクトは、partial() 関数によって作られる呼び出し可能オブジェクトです。オブジェクトには読み取り専用の属性が三つあります。

func
: 呼び出し可能オブジェクトまたは関数です。partial の呼び出しは新しい引数とキーワードと共に func に転送されます。

args
: 最左の位置引数で、partial オブジェクトの呼び出し時にその呼び出しの際の位置引数の前に追加されます。

keywords
: partial オブジェクトの呼び出し時に渡されるキーワード引数です。

partial オブジェクトは function オブジェクトのように呼び出し可能で、弱参照可能で、属性を持つことができます。重要な相違点もあります。例えば、__name__ と __doc__ 両属性は自動では作られません。また、クラス中で定義された partial オブジェクトはスタティックメソッドのように振る舞い、インスタンスの属性問い合わせの中で束縛メソッドに変換されません。

6.7 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、シーケンス操作、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* `<=` *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* `>=` *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、ブール代数値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。“真の” 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。

index(*a*)

__index__(*a*)

整数に変換された *a* を返します。 *a*.__index__() と同等です。 2.5 で追加された仕様です。

シーケンス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*)

シーケンス *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。演算対象が左右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* の要素を返します。

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*)

シーケンス *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 でインデクスが *b* から *c*-1 のスライス要素の値をシーケンス *v* に設定します。

多くの演算に「その場」バージョンがあります。以下の関数はそうした演算子の通常の文法に比べてより素朴な呼び出し方を提供します。たとえば、文 x += y は x = operator.iadd(x, y) と等価です。別の言い方をすると、z = operator.iadd(x, y) は複合文 z = x; z += y と等価です。

iadd(*a*, *b*)

__iadd__(*a*, *b*)

a = iadd(a, b) は a += b と等価です。 2.5 で追加された仕様です。

iand(*a, b*)

__iand__(*a, b*)

`a = iand(a, b)` は `a &= b` と等価です。 2.5 で追加された仕様です。

iconcat(*a, b*)

__iconcat__(*a, b*)

`a = iconcat(a, b)` は二つのシーケンス *a* と *b* に対し `a += b` と等価です。 2.5 で追加された仕様です。

idiv(*a, b*)

__idiv__(*a, b*)

`a = idiv(a, b)` は`__future__.division` が有効でないときに `a /= b` と等価です。 2.5 で追加された仕様です。

ifloordiv(*a, b*)

__ifloordiv__(*a, b*)

`a = ifloordiv(a, b)` は `a //= b` と等価です。 2.5 で追加された仕様です。

ilshift(*a, b*)

__ilshift__(*a, b*)

`a = ilshift(a, b)` は `a <<= b` と等価です。 2.5 で追加された仕様です。

imod(*a, b*)

__imod__(*a, b*)

`a = imod(a, b)` は `a %= b` と等価です。 2.5 で追加された仕様です。

imul(*a, b*)

__imul__(*a, b*)

`a = imul(a, b)` は `a *= b` と等価です。 2.5 で追加された仕様です。

ior(*a, b*)

__ior__(*a, b*)

`a = ior(a, b)` は `a |= b` と等価です。 2.5 で追加された仕様です。

ipow(*a, b*)

__ipow__(*a, b*)

`a = ipow(a, b)` は `a **= b` と等価です。 2.5 で追加された仕様です。

irepeat(*a, b*)

__irepeat__(*a, b*)

`a = irepeat(a, b)` は *a* がシーケンスで *b* が整数であるとき `a *= b` と等価です。 2.5 で追加された仕様です。

irshift(*a, b*)

__irshift__(*a, b*)

`a = irshift(a, b)` は `a >>= b` と等価です。 2.5 で追加された仕様です。

isub(*a, b*)

__isub__(*a, b*)

`a = isub(a, b)` は `a -= b` と等価です。 2.5 で追加された仕様です。

itruediv(*a, b*)

__itruediv__(*a, b*)

`a = itruediv(a, b)` は`__future__.division` が有効なときに `a /= b` と等価です。 2.5 で追加された仕様です。

ixor(*a, b*)

__ixor__(*a, b*)

`a = ixor(a, b)` は `a ^= b` と等価です。 2.5 で追加された仕様です。

`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。例えば以下のようになります:

```
>>> 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 `__getitem__` メソッドが定義された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 がシーケンス型プロトコルをサポートする場合に真を返します。シーケンス型メソッドを C で定義している全てのオブジェクトおよび `__getitem__` メソッドが定義された全てのインスタンス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シーケンス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 `255` 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)
```

`operator` モジュールはアトリビュートとアイテムの汎用的な検索のための道具も定義しています。`map()`, `sorted()`, `itertools.groupby()`, や関数を引数に取るその他の関数に対して高速にフィールドを抽出する際に引数として使うと便利です。

attrgetter(*attr*[, *args...*])

演算対象から *attr* を取得する呼び出し可能なオブジェクトを返します。二つ以上のアトリビュートを要求された場合には、アトリビュートのタプルを返します。'`f=attrgetter('name')`' とした後で、'`f(b)`' を呼び出すと '`b.name`' を返します。'`f=attrgetter('name', 'date')`' とした後で、'`f(b)`' を呼び出すと '`(b.name, b.date)`' を返します。 2.4 で追加された仕様です。 2.5 で変更された仕様: 複数のアトリビュートがサポートされました

itemgetter(*item*[, *args...*])

演算対象から *item* を取得する呼び出し可能なオブジェクトを返します。二つ以上のアイテムを要求された場合には、アイテムのタプルを返します。'`f=itemgetter(2)`' とした後で、'`f(b)`' を呼び出すと '`b[2]`' を返します。'`f=itemgetter(2,5,3)`' とした後で、'`f(b)`' を呼び出すと '`(b[2], b[5], b[3])`' を返します。 2.4 で追加された仕様です。 2.5 で変更された仕様: 複数のアトリビュートがサポートされました

例:

```
>>> from operator import itemgetter
>>> inventory = [('apple', 3), ('banana', 2), ('pear', 5), ('orange', 1)]
>>> getcount = itemgetter(1)
>>> map(getcount, inventory)
[3, 2, 5, 1]
>>> sorted(inventory, key=getcount)
[('orange', 1), ('banana', 2), ('apple', 3), ('pear', 5)]
```

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

七章

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.

email.iterators	メッセージオブジェクトツリーをたどる。
mailcap	mailcap ファイルの操作。
mailbox	様々な形式のメールボックス操作
mhlib	Python から MH のメイルボックスを操作します。
mimetools	MIME-スタイルのメッセージ本体を解析するためのツール。
mimetypes	ファイル名拡張子の MIME 型へのマッピング。
MimeWriter	汎用 MIME ファイルライター。
mimify	電子メイルメッセージの MIME 化および非 MIME 化。
multifile	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。
rfc822	RFC 2822 形式のメイルメッセージを解釈します。
base64	RFC 3548: Base16, Base32, Base64 テータの符号化
binhex	binhex4 形式ファイルのエンコードおよびデコード。
binascii	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。
quopri	MIME quoted-printable 形式ファイルのエンコードおよびデコード。
uu	uuencode 形式のエンコードとデコードを行う。

7.1 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)、 NNTP、 その他の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、nntplib モジュールなどの機能です。email パッケージは RFC 2822 に加えて、RFC 2045, RFC 2046, 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、最後にオブジェクトツリーを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、そして少々の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:

smtplib モジュール (18.13 節):
SMTP プロトコル クライアント

nntplib モジュール (18.12 節):
NNTP プロトコル クライアント

7.1.1 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、(email.message モジュールからインポートされる) 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) 形式のインタフェイスと、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イスを提供します。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とりません。

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。たとえば、これはデフォルトでは From で始まる行を変更してしまいます。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は 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 ではない場合、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ようにしたければ、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っています。*_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大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 をもっているので、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 はそれのデフォルトを 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。ほど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 LAGUAGE 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が、パラメータが RFC 2231 形式でエンコードされているかどうかを気にしないのであれば、`email.Utils.collapse_rfc2231_value()` に `get_param()` の返り値を渡して呼び出すことで、このパラメータをひとつにまとめることができます。この値がタプルならばこの関数は適切にデコードされた Unicode 文字列を返し、そうでない場合は unquote された元の文字列を返します。たとえば:

```
rawparam = msg.get_param('foo')
param = email.Utils.collapse_rfc2231_value(rawparam)
```

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 にしたがいこの値には 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。目的のヘッダに `filename` パラメータがない場合には `name` パラメータを探します。それも無い場合またはヘッダが無い場合には *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
```

2.5 で変更された仕様: 以前の非推奨メソッド get_type()、get_main_type()、get_subtype() は削除されました。

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ので、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。

2.5 で変更された仕様: `Generator` でファイル終端に改行を出力するため、epilogue に空文字列を設定する必要はなくなりました。

defects

defects 属性はメッセージを解析する途中で検出されたすべての問題点 (defect、障害) のリストを保持しています。解析中に発見されうる障害についてのより詳細な説明は `email.errors` を参照してください。

2.4 で追加された仕様です。

7.1.2 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、パーズ) 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。

実際には 2 つのパーザインターフェイスが使用可能です。ひとつは旧式の `Parser` API であり、もうひとつはインクリメンタルな `FeedParser` API です。旧式の `Parser` API はメッセージ全体のテキストが文字列としてすでにメモリ上にあるか、それがローカルなファイルシステム上に存在しているときには問題ありません。`FeedParser` はメッセージを読み込むときに、そのストリームが入力待ちのためにブロックされるような場合 (ソケットから email メッセージを読み込む時など) に、より有効です。`FeedParser` はインクリメンタルにメッセージを読み込み、解析します。パーザを close したときには根っこ (root) のオブジェクトのみが返されます[1]。

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

[1] Python 2.4 から導入された email パッケージ バージョン 3.0 では、旧式の `Parser` は `FeedParser` によって書き直されました。そのためパーザの意味論と得られる結果は 2 つのパーザで同一のものになります。

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。

FeedParser API

2.4 で追加された仕様です。

email.feedparser モジュールからインポートされる FeedParser は email メッセージをインクリメンタルに解析するのに向いた API を提供します。これは email メッセージのテキストを (ソケットなどの) 読み込みがブロックされる可能性のある情報源から入力するときに必要となります。もちろん FeedParser は文字列またはファイルにすべて格納されている email メッセージを解析するのにも使うことができますが、このような場合には旧式の Parser API のほうが便利かもしれません。これら 2 つのパーザ API の意味論と得られる結果は同一です。

FeedParser API は簡単です。まずインスタンスをつくり、それにテキストを (それ以上テキストが必要なくなるまで) 流しこみます。その後パーザを close すると根っこ (root) のメッセージオブジェクトが返されます。標準に従ったメッセージを解析する場合、FeedParser は非常に正確であり、標準に従っていないメッセージでもちゃんと動きます。そのさい、これはメッセージがどのように壊れていると認識されたかについての情報を残します。これはメッセージオブジェクトの *defects* 属性にリストとして現れ、メッセージ中に発見された問題が記録されます。パーザが検出できる障害 (defect) については email.errors モジュールを参照してください。

以下は FeedParser の API です:

class FeedParser([*_factory*])

FeedParser インスタンスを作成します。オプション引数 *_factory* には引数なしの callable を指定し、これはつねに新しいメッセージオブジェクトの作成が必要になったときに呼び出されます。デフォルトでは、これは email.message.Message クラスになっています。

feed(*data*)

FeedParser にデータを供給します。*data* は 1 行または複数行からなる文字列を渡します。渡される行は完結していなくてもよく、その場合 FeedParser は部分的な行を適切につなぎ合わせます。文字列中の各行は標準的な 3 種類の行末文字 (復帰 CR、改行 LF、または CR+LF) どれかの組み合わせでよく、これらが混在してもかまいません。

close()

FeedParser を close し、それまでに渡されたすべてのデータの解析を完了させ根っこ (root) のメッセージオブジェクトを返します。FeedParser を close したあとにさらにデータを feed した場合の挙動は未定義です。

Parser クラス API

email.parser モジュールからインポートされる Parser クラスは、メッセージを表すテキストが文字列またはファイルの形で完全に使用可能なときメッセージを解析するのに使われる API を提供します。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。

class Parser([*_class*])

Parser クラスのコンストラクタです。オプション引数 *_class* をとることができます。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) が作成されるときは常にそのファクトリクラスとして使用されます。デフォルトではこれは `Message` になっています (`email.message` 参照)。このファクトリクラスは引数なしで呼び出されます。

オプション引数 *strict* は無視されます。**リリース 2.4 以降で撤廃された仕様です。** `Parser` は Python 2.4 で新しく導入された `FeedParser` の後方互換性のための API ラッパで、**すべての** 解析が事実上 non-strict です。`Parser` コンストラクタに *strict* フラグを渡す必要はありません。

2.2.2 で変更された仕様: *strict* フラグが追加されました　2.4 で変更された仕様: *strict* フラグは推奨されなくなりました

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 [2] 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:

[2] file-like object

```
>>> import email
>>> msg = email.message_from_string(myString)
```

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 をもつほどんどのメッセージ (例: message/delivery-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。
- いくつかの標準的でないメッセージは、multipart の使い方に統一がとれていない場合があります。このようなメッセージは Content-Type: ヘッダに multipart を指定しているものの、その `is_multipart()` メソッドは `False` を返すことがあります。もしこのようなメッセージが `FeedParser` によって解析されると、その *defects* 属性のリスト中には `MultipartInvariantViolationDefect` クラスのインスタンスが現れます。詳しい情報については `email.errors` を参照してください。

7.1.3 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 [3] になるよう設計されています。

`email.generator` モジュールからインポートされる `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 [4] 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。

[3] 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。
[4] 訳注: file-like object

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 形式のエンペローブ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.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。

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、エンペローブ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ペローブヘッダをもたない場合、標準的なエンペローブ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、エンペローブ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、エンペローブ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っています。

class `DecodedGenerator` (*outfp*[, *mangle_from_*[, *maxheaderlen*[, *fmt*]]])

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 *fmt* 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。*fmt* 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

fmt 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。

2.5 で変更された仕様: 以前の非推奨メソッド __call__() は削除されました。

7.1.4 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。

以下がそのサブクラスです:

class MIMEBase(*_maintype, _subtype, **_params*)

Module: email.mime.base

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()

Module: email.mime.nonmultipart

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 例外を発生します。

2.2.2 で追加された仕様です。

class MIMEMultipart (*[subtype[, boundary[, _subparts[, _params]]]]*)

Module: email.mime.multipart

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 MIMEApplication (*_data[, _subtype[, _encoder[, **_params]]]*)

Module: email.mime.application

MIMENonMultipart のサブクラスである MIMEApplication クラスは MIME メッセージオブジェクトのメジャータイプ application を表します。*_data* は生のバイト列が入った文字列です。オプション引数 *_subtype* は MIME のサブタイプを設定します。サブタイプのデフォルトは octet-stream です。

オプション引数の*_encoder* は呼び出し可能なオブジェクト (関数など) で、データの転送に使う実際のエンコード処理を行います。この呼び出し可能なオブジェクトは引数を 1 つ取り、それは MIMEApplication のインスタンスです。ペイロードをエンコードされた形式に変更するために get_payload() と set_payload() を使い、必要に応じて Content-Transfer-Encoding:やその他のヘッダをメッセージオブジェクトに追加するべきです。デフォルトのエンコードは base64 です。組み込みのエンコーダの一覧は email.encoders モジュールを見てください。

_params は 基底クラスのコンストラクタにそのまま渡されます。 2.5 で追加された仕様です。

class MIMEAudio (*_audiodata[, _subtype[, _encoder[, **_params]]]*)

Module: email.mime.audio

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*]]])

Module: email.mime.image

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 imghdr 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*])

Module: email.mime.message

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*]])

Module: email.mime.text

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。

2.4 で変更された仕様: 以前、推奨されない引数であった *_encoding* は撤去されました。エンコーディングは *_charset* 引数をもとにして暗黙のうちに決定されます。

7.1.5 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 インスタンスを使うのです。 Header クラスは email.header モジュールからインポートしてください。 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() メソッドを呼びだ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。これればバイト文字列 (`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*[, *maxlinelen*[, *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。

7.1.6 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。

このクラスは email.charset モジュールから import してください。

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 euc-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 euc-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。

to_splittable(*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。

from_splittable(*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。

get_output_charset()
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。

encoded_header_len()
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。

header_encode(*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。

body_encode(*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。`header_encode()` 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。

7.1.7 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で必要です。

email パッケージでは、encoders モジュールにおいていく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 MIMEAudio および MIMEImage 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[5] 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。

7.1.8 例外および障害クラス

email.errors モジュールでは、以下の例外クラスが定義されています:

[5] 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。

exception MessageError()

これは email パッケージが発生しうるすべての例外の基底クラスです。これは標準の Exception クラスから派生しており、追加のメソッドはまったく定義されていません。

exception MessageParseError()

これは Parser クラスが発生しうる例外の基底クラスです。MessageError から派生しています。

exception HeaderParseError()

メッセージの RFC 2822 ヘッダを解析している途中にある条件でエラーがおこると発生します。これは MessageParseError から派生しています。この例外が起こる可能性があるのは Parser.parse() メソッドと Parser.parsestr() メソッドです。

この例外が発生するのはメッセージ中で最初の RFC 2822 ヘッダが現れたあとにエンベロープヘッダが見つかったとか、最初の RFC 2822 ヘッダが現れる前に前のヘッダからの継続行が見つかったとかいう状況を含みます。あるいはヘッダでも継続行でもない行がヘッダ中に見つかった場合でもこの例外が発生します。

exception BoundaryError()

メッセージの RFC 2822 ヘッダを解析している途中にある条件でエラーがおこると発生します。これは MessageParseError から派生しています。この例外が起こる可能性があるのは Parser.parse() メソッドと Parser.parsestr() メソッドです。

この例外が発生するのは、厳格なパーズ方式が用いられているときに、multipart/* 形式の開始あるいは終了の文字列が見つからなかった場合などです。

exception MultipartConversionError()

この例外は、Message オブジェクトに add_payload() メソッドを使ってペイロードを追加するとき、そのペイロードがすでに単一の値である (訳注: リストでない) にもかかわらず、そのメッセージの Content-Type: ヘッダのメインタイプがすでに設定されていて、それが multipart 以外になってしまっている場合にこの例外が発生します。MultipartConversionError は MessageError と組み込みの TypeError を両方継承しています。

Message.add_payload() はもはや推奨されないメソッドのため、この例外はふつうめったに発生しません。しかしこの例外は attach() メソッドが MIMENonMultipart から派生したクラスのインスタンス (例: MIMEImage など) に対して呼ばれたときにも発生することがあります。

以下は FeedParser がメッセージの解析中に検出する障害 (defect) の一覧です。注意: これらの障害は、問題が見つかったメッセージに追加されるため、たとえば multipart/alternative 内にあるネストしたメッセージが異常なヘッダをもっていた場合には、そのネストしたメッセージが障害を持っているが、その親メッセージには障害はないとみなされます。

すべての障害クラスは email.errors.MessageDefect のサブクラスですが、これは例外とは**違います**ので注意してください。

2.4 で追加された仕様: All the defect classes were added

- NoBoundaryInMultipartDefect – メッセージが multipart だと宣言されているのに、boundary パラメータがない。
- StartBoundaryNotFoundDefect – Content-Type: ヘッダで宣言された開始境界がない。
- FirstHeaderLineIsContinuationDefect – メッセージの最初のヘッダが継続行から始まっている。
- MisplacedEnvelopeHeaderDefect – ヘッダブロックの途中に “Unix From” ヘッダがある。
- MalformedHeaderDefect – コロンのないヘッダがある、あるいはそれ以外の異常なヘッダである。

- `MultipartInvariantViolationDefect` – メッセージが multipart だと宣言されているのに、サブパートが存在しない。注意: メッセージがこの障害を持っているとき、`is_multipart()` メソッドは たとえその content-type が multipart であっても false を返すことがあります。

7.1.9 雑用ユーティリティ

`email.utils` モジュール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、これらは単に切りおとされます。同様にもし *str* の先頭あるいは末尾が角ブラケット (<、>) だった場合も切りおとされます。

parseaddr(*address*)
アドレスをパーズします。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。パーズに成功した場合、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[6]。入

[6]注意: この時間帯のオフセット値は `time.timezone` の値と符合が逆です。これは `time.timezone` が POSIX 標準に準拠して

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つかえないものの、わずかな誤差を生じるかもしれません。

formatdate([*timeval*[, *localtime*][, *usegmt*]])

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。

オプション引数 *usegmt* が True のときは、タイムゾーンを表すのに数値の -0000 ではなく ascii 文字列である GMT が使われます。これは (HTTP などの) いくつかのプロトコルで必要です。この機能は *localtime* が False のときのみ適用されます。 2.4 で追加された仕様です。

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。

collapse_rfc2231_value(*value*[, *errors*[, *fallback_charset*]])

ヘッダのパラメータが RFC 2231 形式でエンコードされている場合、Message.get_param() は 3 要素からなるタプルを返すことがあります。ここには、そのパラメータの文字セット、言語、および値の順に格納されています。collapse_rfc2231_value() はこのパラメータをひとつの Unicode 文字列にまとめます。オプション引数 *errors* は built-in である unicode() 関数の引数 *errors* に渡されます。このデフォルト値は replace となっています。オプション引数 *fallback_charset* は、もし RFC 2231 ヘッダの使用している文字セットが Python の知っているものではなかった場合の非常用文字セットとして使われます。デフォルトでは、この値は us-ascii です。

便宜上、collapse_rfc2231_value() に渡された引数 *value* がタプルでない場合には、これは文字列である必要があります。その場合には unquote された文字列が返されます。

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。

いるのに対して、こちらは RFC 2822 に準拠しているからです。

2.4 で変更された仕様: `dump_address_pair()` 関数は撤去されました。かわりに `formataddr()` 関数を使ってください。

2.4 で変更された仕様: `decode()` 関数は撤去されました。かわりに `Header.decode_header()` メソッドを使ってください。

2.4 で変更された仕様: `encode()` 関数は撤去されました。かわりに `Header.encode()` メソッドを使ってください。

7.1.10 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。

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
    text/plain
```

オプション引数 *fp* は出力を渡すためのストリーム[7]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。

7.1.11 パッケージの履歴

このテーブルは email パッケージのリリース履歴を表しています。それぞれのバージョンと、それが同梱された Python のバージョンとの関連が示されています。このドキュメントでの、追加/変更されたバージョンの表記は email パッケージのバージョン**ではなく**、Python のバージョンです。このテーブルは Python の各バージョン間の email パッケージの互換性も示しています。

email バージョン	配布	互換
`1.x`	Python 2.2.0 to Python 2.2.1	**もうサポートされません**
`2.5`	Python 2.2.2+ and Python 2.3	Python 2.1 から 2.5
`3.0`	Python 2.4	Python 2.3 から 2.5
`4.0`	Python 2.5	Python 2.3 から 2.5

以下は `email` バージョン 4 と 3 の間のおもな差分です。

- 全モジュールが PEP 8 標準にあわせてリネームされました。たとえば、version 3 でのモジュール `email.Message` は version 4 では `email.message` になりました。
- 新しいサブパッケージ `email.mime` が追加され、version 3 の `email.MIME*` は、`email.mime` のサブパッケージにまとめられました。 たとえば、version 3 での `email.MIMEText` は、`email.mime.text` になりました。

  Python 2.6 **までは** *version 3* **の名前も有効です。**
- `email.mime.application` モジュールが追加されました。これは `MIMEApplication` クラスを含んでいます。
- version 3 で推奨されないとされた機能は削除されました。これらは `Generator.__call__()`、`Message.get_type()`、`Message.get_main_type()`、 `Message.get_subtype()` を含みます。
- RFC 2331 サポートの修正が追加されました。これは `Message.get_param()` などの関数の返り値を変更します。いくつかの環境では、3 つ組のタプルで返されていた値が 1 つの文字列で返されます (とくに、全ての拡張パラメータセグメントがエンコードされていなかった場合、予測されていた language や charset の指定がないと、返り値は単純な文字列になります)。過去の版では % デコードが エンコードされているセグメントおよびエンコードされていないセグメントに対して行われましたが、エンコードされたセグメントのみで行われるようになりました。

`email` バージョン 3 と バージョン 2 との違いは以下のようなものです:

- `FeedParser` クラスが新しく導入され、`Parser` クラスは `FeedParser` を使って実装されるようになりました。このパーザは non-strict なものであり、解析はベストエフォート方式でおこなわれ解析中に例外を発生させることはありません。解析中に発見された問題はそのメッセージの *defect* (障害) 属性に保存されます。
- バージョン 2 で `DeprecationWarning` を発生していた API はすべて撤去されました。以下のものが含まれています: `MIMEText` コンストラクタに渡す引数 *_encoder*、`Message.add_payload()` メソッド、`Utils.dump_address_pair()` 関数、そして `Utils.decode()` と `Utils.encode()` です。

[7]訳注: 原文では file-like。

- 新しく以下の関数が DeprecationWarning を発生するようになりました: Generator.__call__(),Message.get_type(),Message.get_main_type(),Message.get_subtype(),そして Parser クラスに対する *strict* 引数です。これらは email の将来のバージョンで撤去される予定です。
- Python 2.3 以前はサポートされなくなりました。

email バージョン 2 と バージョン 1 との違いは以下のようなものです:

- 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。
- 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。

7.1.12 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しょう。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っています。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[8]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しまっていました。

  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。

7.1.13 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.mime.text 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
```

[8]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.mime.image import MIMEImage
from email.mime.multipart 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'

# 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[9]:

```
#!/usr/bin/env python

"""Send the contents of a directory as a MIME message."""

import os
import sys
import smtplib
```

[9]最初の思いつきと用例は Matthew Dixon Cowles のおかげです。

```
# For guessing MIME type based on file name extension
import mimetypes

from optparse import OptionParser

from email import encoders
from email.message import Message
from email.mime.audio import MIMEAudio
from email.mime.base import MIMEBase
from email.mime.image import MIMEImage
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText

COMMASPACE = ', '


def main():
    parser = OptionParser(usage="""\
Send the contents of a directory as a MIME message.

Usage: %prog [options]

Unless the -o option is given, the email is sent by forwarding to your local
SMTP server, which then does the normal delivery process.  Your local machine
must be running an SMTP server.
""")
    parser.add_option('-d', '--directory',
                      type='string', action='store',
                      help="""Mail the contents of the specified directory,
                      otherwise use the current directory.  Only the regular
                      files in the directory are sent, and we don't recurse to
                      subdirectories.""")
    parser.add_option('-o', '--output',
                      type='string', action='store', metavar='FILE',
                      help="""Print the composed message to FILE instead of
                      sending the message to the SMTP server.""")
    parser.add_option('-s', '--sender',
                      type='string', action='store', metavar='SENDER',
                      help='The value of the From: header (required)')
    parser.add_option('-r', '--recipient',
                      type='string', action='append', metavar='RECIPIENT',
                      default=[], dest='recipients',
                      help='A To: header value (at least one required)')
    opts, args = parser.parse_args()
    if not opts.sender or not opts.recipients:
        parser.print_help()
        sys.exit(1)
    directory = opts.directory
    if not directory:
        directory = '.'
    # Create the enclosing (outer) message
    outer = MIMEMultipart()
    outer['Subject'] = 'Contents of directory %s' % os.path.abspath(directory)
    outer['To'] = COMMASPACE.join(opts.recipients)
    outer['From'] = opts.sender
    outer.preamble = 'You will not see this in a MIME-aware mail reader.\n'

    for filename in os.listdir(directory):
        path = os.path.join(directory, filename)
        if not os.path.isfile(path):
            continue
        # Guess the content type based on the file's extension.  Encoding
        # will be ignored, although we should check for simple things like
```

```
            # gzip'd or compressed files.
            ctype, encoding = mimetypes.guess_type(path)
            if ctype is None or encoding is not None:
                # No guess could be made, or the file is encoded (compressed), so
                # use a generic bag-of-bits type.
                ctype = 'application/octet-stream'
            maintype, subtype = ctype.split('/', 1)
            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 or store the message
        composed = outer.as_string()
        if opts.output:
            fp = open(opts.output, 'w')
            fp.write(composed)
            fp.close()
        else:
            s = smtplib.SMTP()
            s.connect()
            s.sendmail(opts.sender, opts.recipients, composed)
            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."""

    import os
    import sys
    import email
    import errno
    import mimetypes

    from optparse import OptionParser


    def main():
        parser = OptionParser(usage="""\
```

```
Unpack a MIME message into a directory of files.

Usage: %prog [options] msgfile
""")
    parser.add_option('-d', '--directory',
                      type='string', action='store',
                      help="""Unpack the MIME message into the named
                      directory, which will be created if it doesn't already
                      exist.""")
    opts, args = parser.parse_args()
    if not opts.directory:
        parser.print_help()
        sys.exit(1)

    try:
        msgfile = args[0]
    except IndexError:
        parser.print_help()
        sys.exit(1)

    try:
        os.mkdir(opts.directory)
    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(opts.directory, filename), 'wb')
        fp.write(part.get_payload(decode=True))
        fp.close()


if __name__ == '__main__':
    main()
```

7.2 mailcap — mailcap ファイルの操作

mailcap ファイルは、メイルリーダや 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つ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ます。`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'})
```

7.3 mailbox — 様々な形式のメールボックス操作

このモジュールでは二つのクラス Mailbox および Message をディスク上のメールボックスとそこに収められたメッセージへのアクセスと操作のために定義しています。Mailbox は辞書のようなキーからメッセージへの対応付けを提供しています。Message は email.Message モジュールの Message を拡張して形式ごとの状態と振る舞いを追加しています。サポートされるメールボックスの形式は Maildir, mbox, MH, Babyl, MMDF です。

参考資料:

email モジュール (7.1 節):
メッセージの表現と操作

7.3.1 Mailbox オブジェクト

class Mailbox
メールボックス。中を見られたり変更されたりします。

Mailbox のインタフェースは辞書風で、小さなキーがメッセージに対応します。キーは対象となる Mailbox インスタンスが発行するもので、そのインスタンスに対してのみ意味を持ちます。一つのキーは一つのメッセージにひも付けられ、その対応はメッセージが他のメッセージで置き換えられるような更新をされたあとも続きます。メッセージを Mailbox インスタンスに追加するには集合風のメソッド add() を使います。また削除は del 文または集合風の remove() や discard() を使って行ないます。

Mailbox インタフェースのセマンティクスと辞書のそれとは注意すべき違いがあります。メッセージは、要求されるたびに新しい表現 (典型的には Message インスタンス) が現在のメールボックスの状態に基づいて生成されます。同様に、メッセージが Mailbox インスタンスに追加される時も、渡されたメッセージ表現の内容がコピーされます。どちらの場合も Makebox インスタンスにメッセージ表現への参照は保たれません。

デフォルトの Mailbox イテレータはメッセージ表現ごとに繰り返すもので、辞書のイテレータのようにキーごとの繰り返しではありません。さらに、繰り返し中のメールボックスを変更することは安全であり整合的に定義されています。イテレータが作られた後にメールボックスに追加されたメッセージはそのイテレータからは見えません。そのイテレータが yield するまえにメールボックスから削除されたメッセージは黙ってスキップされますが、イテレータからのキーを使ったときにはそのキーに対応するメッセージが削除されているならば KeyError を受け取ることになります。

Mailbox 自体はインタフェースを定義し形式ごとのサブクラスに継承されるように意図されたもので、インスタンス化されることは想定されていません。インスタンス化したいならばサブクラスを代わりに使うべきです。

Mailbox インスタンスには次のメソッドがあります。

add(*message*)
メールボックスに *message* を追加し、それに割り当てられたキーを返します。

引数 *message* は Message インスタンス、email.Message.Message インスタンス、文字列、ファイル風オブジェクト (テキストモードで開かれていなければなりませんが) を使えます。*message* が適切な形式に特化した Message サブクラスのインスタンス (例えばメールボックスが mbox インスタンスのときの mboxMessage インスタンス) であれば、形式ごとの情報が利用されます。そうでなければ、形式ごとに必要な情報は適当なデフォルトが使われます。

remove(*key*)
__delitem__(*key*)
discard(*key*)

メールボックスから *key* に対応するメッセージを削除します。

対応するメッセージが無い場合、メソッドが `remove()` または `__delitem__()` として呼び出されている時は `KeyError` 例外が送出されます。しかし、`discard()` として呼び出されている場合は例外は発生しません。基づいているメールボックス形式が別のプロセスからの平行した変更をサポートしているならば、この `discard()` の振る舞いの方が好まれるかもしれません。

__setitem__(*key, message*)

key に対応するメッセージを *message* で置き換えます。*key* に対応しているメッセージが既に無くなっている場合 `KeyError` 例外が送出されます。

`add()` と同様に、引数の *message* には `Message` インスタンス、`email.Message.Message` インスタンス、文字列、ファイル風オブジェクト (テキストモードで開かれていなければなりませんが) を使えます。*message* が適切な形式に特化した `Message` サブクラスのインスタンス (例えばメールボックスが `mbox` インスタンスのときの `mboxMessage` インスタンス) であれば、形式ごとの情報が利用されます。そうでなければ、現在 *key* に対応するメッセージの形式ごとの情報が変更されずに残ります。

iterkeys()

keys()

`iterkeys()` として呼び出されると全てのキーについてのイテレータを返しますが、`keys()` として呼び出されるとキーのリストを返します。

itervalues()

__iter__()

values()

`itervalues()` または `__iter__()` として呼び出されると全てのメッセージの表現についてのイテレータを返しますが、`values()` として呼び出されるとその表現のリストを返します。メッセージは適切な形式ごとの `Message` サブクラスのインスタンスとして表現されるのが普通ですが、`Mailbox` インスタンスが初期化されるときに指定すればお好みのメッセージファクトリを使うこともできます。**注意:** `__iter__()` は辞書のそれのようにキーについてのイテレータではありません。

iteritems()

items()

(*key*, *message*) ペア、ただし *key* はキーで *message* はメッセージ表現、のイテレータ (`iteritems()` として呼び出された場合)、またはリスト (`items()` として呼び出された場合) を返します。メッセージは適切な形式ごとの `Message` サブクラスのインスタンスとして表現されるのが普通ですが、`Mailbox` インスタンスが初期化されるときに指定すればお好みのメッセージファクトリを使うこともできます。

get(*key*[, *default=None*])

__getitem__(*key*)

key に対応するメッセージの表現を返します。対応するメッセージが存在しない場合、`get()` として呼び出されたなら *default* を返しますが、`__getitem__()` として呼び出されたなら `KeyError` 例外が送出されます。メッセージは適切な形式ごとの `Message` サブクラスのインスタンスとして表現されるのが普通ですが、`Mailbox` インスタンスが初期化されるときに指定すればお好みのメッセージファクトリを使うこともできます。

get_message(*key*)

key に対応するメッセージの表現を形式ごとの `Message` サブクラスのインスタンスとして返します。もし対応するメッセージが存在しなければ `KeyError` 例外が送出されます。

get_string(*key*)

key に対応するメッセージの表現を文字列として返します。もし対応するメッセージが存在しなければ `KeyError` 例外が送出されます。

get_file(*key*)

key に対応するメッセージの表現をファイル風表現として返します。もし対応するメッセージが存在しなければ KeyError 例外が送出されます。ファイル風オブジェクトはバイナリモードで開かれているように振る舞います。このファイルは必要がなくなったら閉じなければなりません。

注意: 他の表現方法とは違い、ファイル風オブジェクトはそれを作り出した Mailbox インスタンスやそれが基づいているメールボックスと独立である必要がありません。より詳細な説明は各サブクラスごとにあります。

has_key(*key*)

__contains__(*key*)

key がメッセージに対応していれば True を、そうでなければ False を返します。

__len__()

メールボックス中のメッセージ数を返します。

clear()

メールボックスから全てのメッセージを削除します。

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

key に対応するメッセージの表現を返します。もし対応するメッセージが存在しなければ *default* が供給されていればその値を返し、そうでなければ KeyError 例外を送出します。メッセージは適切な形式ごとの Message サブクラスのインスタンスとして表現されるのが普通ですが、Mailbox インスタンスが初期化されるときに指定すればお好みのメッセージファクトリを使うこともできます。

popitem()

任意に選んだ (*key*, *message*) ペアを返します。ただしここで *key* はキーで *message* はメッセージ表現です。もしメールボックスが空ならば、KeyError 例外を送出します。メッセージは適切な形式ごとの Message サブクラスのインスタンスとして表現されるのが普通ですが、Mailbox インスタンスが初期化されるときに指定すればお好みのメッセージファクトリを使うこともできます。

update(*arg*)

引数 *arg* は *key* から *message* へのマッピングまたは (*key*, *message*) ペアのイテレート可能オブジェクトでなければなりません。メールボックスは、各 *key* と *message* のペアについて__setitem__()を使ったかのように *key* に対応するメッセージが *message* になるように更新されます。__setitem__()と同様に、*key* は既存のメールボックス中のメッセージに対応しているものでなければならず、そうでなければ KeyError が送出されます。ですから、一般的には *arg* に Mailbox インスタンスを渡すのは間違いです。**注意:** 辞書と違い、キーワオード引数はサポートされていません。

flush()

保留されている変更をファイルシステムに書き込みます。Mailbox のサブクラスによっては変更はいつも直ちにファイルに書き込まれこのメソッドは何もしないということもあります。

lock()

メールボックスの排他的アドバイザリロックを取得し、他のプロセスが変更しないようにします。ロックが取得できない場合 ExternalClashError が送出されます。ロック機構はメールボックス形式によって変わります。

unlock()

メールボックスのロックを、もしあれば、解放します。

close()

+Flush the mailbox, unlock it if necessary, and close any open files. For some +Mailbox subclasses, this method does nothing. メールボックスをフラッシュし、必要ならばアンロックし、開いているファイルを閉じます。Mailbox サブクラスによっては何もしないこともあります。

Maildir

class Maildir (*dirname*[, *factory=rfc822.Message*[, *create=True*]])

Maildir 形式のメールボックスのための Mailbox のサブクラス。パラメータ *factory* は呼び出し可能オブジェクトで (バイナリモードで開かれているかのように振る舞う) ファイル風メッセージ表現を受け付けて好みの表現を返すものです。*factory* が None ならば、MaildirMessage がデフォルトのメッセージ表現として使われます。*create* が True ならばメールボックスが存在しないときには作成します。

factory のデフォルトが rfc822.Message であったり、*path* ではなく *dirname* という名前であったりというのは歴史的理由によるものです。Maildir インスタンスが他の Mailbox サブクラスと同じように振る舞わせるためには、*factory* に None をセットしてください。

Maildir はディレクトリ型のメールボックス形式でメール転送エージェント qmail 用に発明され、現在では多くの他のプログラムでもサポートされているものです。Maildir メールボックス中のメッセージは共通のディレクトリ構造の下で個別のファイルに保存されます。このデザインにより、Maildir メールボックスは複数の無関係のプログラムからデータを失うことなくアクセスしたり変更したりできます。そのためロックは不要です。

Maildir メールボックスには三つのサブディレクトリ 'tmp', 'new', 'cur' があります。メッセージはまず 'tmp' サブディレクトリに瞬間的に作られた後、'new' サブディレクトリに移動されて配送を完了します。メールユーザエージェントが引き続いて 'cur' サブディレクトリにメッセージを移動しメッセージの状態についての情報をファイル名に追加される特別な"info"セクションに保存することができます。

Courier メール転送エージェントによって導入されたスタイルのフォルダもサポートされます。主たるメールボックスのサブディレクトリは '.' がファイル名の先頭であればフォルダと見なされます。フォルダ名は Maildir によって先頭の '.' を除いて表現されます。各フォルダはまた Maildir メールボックスですがさらにフォルダを含むことはできません。その代わり、論理的包含関係は例えば "Archived.2005.07" のような '.' を使ったレベル分けで表わされます。

注意: 本来の Maildir 仕様ではある種のメッセージのファイル名にコロン (':') を使う必要があります。しかしながら、オペレーティングシステムによってはこの文字をファイル名に含めることができないことがあります。そういった環境で Maildir のような形式を使いたい場合、代わりに使われる文字を指定する必要があります。感嘆符 ('!') を使うのが一般的な選択です。以下の例を見てください。

```
import mailbox
mailbox.Maildir.colon = '!'
```

colon 属性はインスタンスごとにセットしても構いません。

Maildir インスタンスには Mailbox の全てのメソッドに加え以下のメソッドもあります。

list_folders()

全てのフォルダ名のリストを返します。

get_folder(*folder*)

名前が *folder* であるフォルダを表わす Maildir インスタンスを返します。そのようなフォルダが存在しなければ NoSuchMailboxError 例外が送出されます。

add_folder(*folder*)

名前が *folder* であるフォルダを作り、それを表わす Maildir インスタンスを返します。

remove_folder(*folder*)

名前が *folder* であるフォルダを削除します。もしフォルダに一つでもメッセージが含まれていれば NotEmptyError 例外が送出されフォルダは削除されません。

clean()
過去 36 時間以内にアクセスされなかったメールボックス内の一時ファイルを削除します。Maildir 仕様はメールを読むプログラムはときどきこの作業をすべきだとしています。

`Maildir` で実装された `Mailbox` のいくつかのメソッドには特別な注意が必要です。

add(*message*)

__setitem__(*key, message*)

update(*arg*)
警告: これらのメソッドは一意的なファイル名をプロセス ID に基づいて生成します。複数のスレッドを使う場合は、同じメールボックスを同時に操作しないようにスレッド間で調整しておかないと検知されない名前の衝突が起こりメールボックスを壊すかもしれません。

flush()
Maildir メールボックスへの変更は即時に適用されるので、このメソッドは何もしません。

lock()

unlock()
Maildir メールボックスはロックをサポート (または要求) しないので、このメソッドは何もしません。

close()
`Maildir` インスタンスは開いたファイルを保持しませんしメールボックスはロックをサポートしませんので、このメソッドは何もしません。

get_file(*key*)
ホストのプラットフォームによっては、返されたファイルが開いている間元になったメッセージを変更したり削除したりできない場合があります。

参考資料:

qmail の maildir man ページ
Maildir 形式のオリジナルの仕様

Using maildir format
Maildir 形式の発明者による注意書き。更新された名前生成規則と "info" の解釈についても含まれます。

Courier の maildir man ページ
Maildir 形式のもう一つの仕様。フォルダをサポートする一般的な拡張について記述されています。

`mbox`

class mbox(*path*[, *factory=None*[, *create=True*]])
mbox 形式のメールボックスのための `Mailbox` のサブクラス。パラメータ *factory* は呼び出し可能オブジェクトで (バイナリモードで開かれているかのように振る舞う) ファイル風メッセージ表現を受け付けて好みの表現を返すものです。*factory* が `None` ならば、`mboxMessage` がデフォルトのメッセージ表現として使われます。*create* が `True` ならばメールボックスが存在しないときには作成します。

mbox 形式は UNIX システム上でメールを保存する古くからある形式です。mbox メールボックスでは全てのメッセージが一つのファイルに保存されておりそれぞれのメッセージは "From " という 5 文字で始まる行を先頭に付けられています。

mbox 形式には幾つかのバリエーションがあり、それぞれオリジナルの形式にあった欠点を克服すると主張しています。互換性のために、`mbox` はオリジナルの (時に *mboxo* と呼ばれる) 形式を実装しています。すなわち、Content-Length: ヘッダはもしあっても無視され、メッセージのボディにある行頭の "From " はメッセージを保存する際に">From " に変換されますが、この ">From " は読み出し時にも "From " に変換されません。

mbox で実装された Mailbox のいくつかのメソッドには特別な注意が必要です。

get_file(*key*)
: mbox インスタンスに対し flush() や close() を呼び出した後でファイルを使用すると予期しない結果を引き起こしたり例外が送出されたりすることがあります。

lock()

unlock()
: 3 種類のロック機構が使われます — ドットロッキングと、もし使用可能ならば flock() と lockf() システムコールです。

参考資料:

qmail の mbox man ページ
: mbox 形式の仕様および種々のバリエーション

tin の mbox man ページ
: もう一つの mbox 形式の仕様でロックについての詳細を含む

Configuring Netscape Mail on UNIX: Why The Content-Length Format is Bad
: バリエーションの一つではなくオリジナルの mbox を使う理由

"mbox" is a family of several mutually incompatible mailbox formats
: mbox バリエーションの歴史

MH

class MH(*path*[, *factory*=*None*[, *create*=*True*]])
: MH 形式のメールボックスのための Mailbox のサブクラス。パラメータ *factory* は呼び出し可能オブジェクトで (バイナリモードで開かれているかのように振る舞う) ファイル風メッセージ表現を受け付けて好みの表現を返すものです。*factory* が None ならば、MHMessage がデフォルトのメッセージ表現として使われます。*create* が True ならばメールボックスが存在しないときには作成します。

MH はディレクトリに基づいたメールボックス形式で MH Message Handling System というメールユーザエージェントのために発明されました。MH メールボックス中のそれぞれのメッセージは一つのファイルとして収められています。MH メールボックスにはメッセージの他に別の MH メールボックス (**フォルダ** と呼ばれます) を含んでもかまいません。フォルダは無限にネストできます。MH メールボックスにはもう一つ **シーケンス** という名前付きのリストでメッセージをサブフォルダに移動することなく論理的に分類するものがサポートされています。シーケンスは各フォルダの '.mh_sequences' というファイルで定義されます。

MH クラスは MH メールボックスを操作しますが、**mh** の動作の全てを模倣しようとはしていません。特に、**mh** が状態と設定を保存する 'context' や '.mh_profile' といったファイルは書き換えませんし影響も受けません。

MH インスタンスには Mailbox の全てのメソッドの他に次のメソッドがあります。

list_folders()
: 全てのフォルダの名前のリストを返します。

get_folder(*folder*)
: *folder* という名前のフォルダを表わす MH インスタンスを返します。もしフォルダが存在しなければ NoSuchMailboxError 例外が送出されます。

add_folder(*folder*)
: *folder* という名前のフォルダを作成し、それを表わす MH インスタンスを返します。

remove_folder(*folder*)
: *folder* という名前のフォルダを削除します。フォルダにメッセージが一つでも残っていれば、

NotEmptyError 例外が送出されフォルダは削除されません。

get_sequences()
シーケンス名をキーのリストに対応付ける辞書を返します。シーケンスが一つもなければ空の辞書を返します。

set_sequences(*sequences*)
メールボックス中のシーケンスを get_sequences() で返されるような名前とキーのリストを対応付ける辞書 *sequences* に基づいて再定義します。

pack()
番号付けの間隔を詰める必要に応じてメールボックス中のメッセージの名前を付け替えます。シーケンスのリストのエントリもそれに応じて更新されます。**注意:** 既に発行されたキーはこの操作によって無効になるのでそれ以降使ってはなりません。

MH で実装された Mailbox のいくつかのメソッドには特別な注意が必要です。

remove(*key*)
__delitem__(*key*)
discard(*key*)
これらのメソッドはメッセージを直ちに削除します。名前の前にコンマを付加してメッセージに削除の印を付けるという MH の規約は使いません。

lock()
unlock()
3 種類のロック機構が使われます — ドットロッキングと、もし使用可能ならば flock() と lockf() システムコールです。MH メールボックスに対するロックとは '.mh_sequences' のロックと、それが影響を与える操作中だけの個々のメッセージファイルに対するロックを意味します。

get_file(*key*)
ホストのプラットフォームによっては、返されたファイルが開いている間元になったメッセージを変更したり削除したりできない場合があります。

flush()
MH メールボックスへの変更は即時に適用されますのでこのメソッドは何もしません。

close()
MH インスタンスは開いたファイルを保持しませんのでこのメソッドは unlock と同じです。

参考資料:

nmh - Message Handling System
mh の改良版である **nmh** のホームページ

MH & nmh: Email for Users & Programmers
GPL ライセンスの **mh** および **nmh** の本で、このメールボックス形式についての情報があります

Babyl

class Babyl(*path*[, *factory=None*[, *create=True*]])
Babyl 形式のメールボックスのための Mailbox のサブクラス。パラメータ *factory* は呼び出し可能オブジェクトで (バイナリモードで開かれているかのように振る舞う) ファイル風メッセージ表現を受け付けて好みの表現を返すものです。*factory* が None ならば、BabylMessage がデフォルトのメッセージ表現として使われます。*create* が True ならばメールボックスが存在しないときには作成します。

Babyl は単一ファイルのメールボックス形式で Emacs に付属している Rmail メールユーザエージェントで使われているものです。メッセージの開始は Control-Underscore (‘\037’) および Control-L (‘\014’) の二文字を含む行で示されます。メッセージの終了は次のメッセージの開始または最後のメッセージの場合には Control-Underscore を含む行で示されます。

Babyl メールボックス中のメッセージには二つのヘッダのセット、オリジナルヘッダといわゆる可視ヘッダ、があります。可視ヘッダは典型的にはオリジナルヘッダの一部を分り易いように再整形したり短くしたりしたものです。Babyl メールボックス中のそれぞれのメッセージには **ラベル** というそのメッセージについての追加情報を記録する短い文字列のリストを伴い、メールボックス中に見出されるユーザが定義した全てのラベルのリストは Babyl オプションセクションに保持されます。

`Babyl` インスタンスには `Mailbox` の全てのメソッドの他に次のメソッドがあります。

`get_labels``()`
: メールボックスで使われているユーザが定義した全てのラベルのリストを返します。**注意:** メールボックスにどのようなラベルが存在するかを決めるのに、Babyl オプションセクション のリストを参考にせず、実際のメッセージを捜索しますが、Babyl セクションもメールボックスが変更されたときにはいつでも更新されます。

`Babyl` で実装された `Mailbox` のいくつかのメソッドには特別な注意が必要です。

`get_file``(`*key*`)`
: Babyl メールボックスにおいて、メッセージのヘッダはボディと繋がって格納されていません。ファイル風の表現を生成するために、ヘッダとボディが (`StringIO` モジュールの) ファイルと同じ API を持つ `StringIO` インスタンスに一緒にコピーされます。その結果、ファイル風オブジェクトは本当に元にしているメールボックスとは独立していますが、文字列表現と比べてメモリーを節約することにもなりません。

`lock``()`

`unlock``()`
: 3 種類のロック機構が使われます — ドットロッキングと、もし使用可能ならば `flock()` と `lockf()` システムコールです。

参考資料:

Format of Version 5 Babyl Files
: Babyl 形式の仕様

Reading Mail with Rmail
: Rmail のマニュアルで Babyl のセマンティクスについての情報も少しある

`MMDF`

class `MMDF` (*path*[, *factory=None*[, *create=True*]])
: MMDF 形式のメールボックスのための `Mailbox` のサブクラス。パラメータ *factory* は呼び出し可能オブジェクトで (バイナリモードで開かれているかのように振る舞う) ファイル風メッセージ表現を受け付けて好みの表現を返すものです。*factory* が `None` ならば、`BabylMessage` がデフォルトのメッセージ表現として使われます。*create* が `True` ならばメールボックスが存在しないときには作成します。

MMDF は単一ファイルのメールボックス形式で Multichannel Memorandum Distribution Facility というメール転送エージェント用に発明されたものです。各メッセージは mbox と同様の形式で収められますが、前後を 4 つの Control-A (‘\001’) を含む行で挟んであります。mbox 形式と同じようにそれぞれのメッセージの開始は "From " の 5 文字を含む行で示されますが、それ以外の場所での "From " は格納の際 ">From " には変えられません。それは追加されたメッセージ区切りによって新たなメッセージの開始と見間違うこ

とが避けられるからです。

MMDF で実装された Mailbox のいくつかのメソッドには特別な注意が必要です。

get_file(*key*)

MMDF インスタンスに対し flush() や close() を呼び出した後でファイルを使用すると予期しない結果を引き起こしたり例外が送出されたりすることがあります。

lock()

unlock()

3 種類のロック機構が使われます — ドットロッキングと、もし使用可能ならば flock() と lockf() システムコールです。

参考資料:

tin の mmdf man page

ニュースリーダ tin のドキュメント中の MMDF 形式仕様

MMDF

Multichannel Memorandum Distribution Facility についてのウィキペディアの記事

7.3.2 Message objects

class Message([*message*])

email.Message モジュールの Message のサブクラス。mailbox.Message のサブクラスはメールボックス形式ごとの状態と動作を追加します。

message が省略された場合、新しいインスタンスはデフォルトの空の状態で生成されます。*message* が email.Message.Message インスタンスならばその内容がコピーされます。さらに、*message* が Message インスタンスならば、形式固有の情報も可能な限り変換されます。*message* が文字列またはファイルならば、読まれ解析されるべき RFC 2822 準拠のメッセージを含んでいなければなりません

サブクラスにより提供される形式ごとの状態と動作は様々ですが、一般に或るメールボックスに固有のものでないプロパティだけがサポートされます (おそらくプロパティのセットはメールボックス形式ごとに固有でしょうが)。例えば、単一ファイルメールボックス形式におけるファイルオフセットやディレクトリ式メールボックス形式におけるファイル名は保持されません、というのもそれらは元々のメールボックスにしか適用できないからです。しかし、メッセージがユーザに読まれたかどうかあるいは重要だとマークされたかどうかという状態は保持されます、というのはそれらはメッセージ自体に適用されるからです。

Mailbox インスタンスを使って取得したメッセージを表現するのに Message インスタンスが使われなければいけないとは要求していません。ある種の状況では Message による表現を生成するのに必要な時間やメモリーが受け入れられないこともあります。そういった状況では Mailbox インスタンスは文字列やファイル風オブジェクトの表現も提供できますし、Mailbox インスタンスを初期化する際にメッセージファクトリーを指定することもできます。

MaildirMessage

class MaildirMessage([*message*])

Maildir 固有の動作をするメッセージ。引数 *message* は Message のコンストラクタと同じ意味を持ちます。

通常、メールユーザエージェントは 'new' サブディレクトリにある全てのメッセージをユーザが最初にメールボックスを開くか閉じるかした後で 'cur' サブディレクトリに移動し、メッセージが実際に読まれたかどうかを記録します。'cur' にある各メッセージには状態情報を保存するファイル名に付け加えられた"info"

セクションがあります。(メールリーダの中には "info" セクションを 'new' にあるメッセージに付けることもあります。) "info" セクションには二つの形式があります。一つは "2," の後に標準化されたフラグのリストを付けたもの (たとえば "2,FR")、もう一つは "1," の後にいわゆる実験的情報を付け加えるものです。Maildir の標準的なフラグは以下の通りです:

フラグ	意味	説明
D	ドラフト (Draft)	作成中
F	フラグ付き (Flagged)	重要とされたもの
P	通過 (Passed)	転送、再送またはバウンス
R	返答済み (Replied)	返答されたもの
S	既読 (Seen)	読んだもの
T	ごみ (Trashed)	削除予定とされたもの

`MaildirMessage` インスタンスは以下のメソッドを提供します。

`get_subdir`()
: "new" (メッセージが 'new' サブディレクトリに保存されるべき場合) または"cur" (メッセージが 'cur' サブディレクトリに保存されるべき場合) のどちらかを返します。**注意:** メッセージは通常メールボックスがアクセスされた後、メッセージが読まれたかどうかに関わらず 'new' から 'cur' に移動されます。メッセージ `msg` は `"S" not in msg.get_flags()` が `True` ならば読まれています。

`set_subdir`(*subdir*)
: メッセージが保存されるべきサブディレクトリをセットします。パラメータ *subdir* は "new" または "cur" のいずれかでなければなりません。

`get_flags`()
: 現在セットされているフラグを特定する文字列を返します。メッセージが標準 Maildir 形式に準拠しているならば、結果はアルファベット順に並べられたゼロまたは 1 回の 'D'、'F'、'P'、'R'、'S'、'T' をつなげたものです。空文字列が返されるのはフラグが一つもない場合、または"info" が実験的セマンティクスを使っている場合です。

`set_flags`(*flags*)
: *flags* で指定されたフラグをセットし、他のフラグは下ろします。

`add_flag`(*flag*)
: *flags* で指定されたフラグをセットしますが他のフラグは変えません。一度に二つ以上のフラグをセットすることは、*flag* に 2 文字以上の文字列を指定すればできます。現在の "info" はフラグの代わりに実験的情報を使っていても上書きされます。

`remove_flag`(*flag*)
: *flags* で指定されたフラグを下ろしますが他のフラグは変えません。一度に二つ以上のフラグを取り除くことは、*flag* に 2 文字以上の文字列を指定すればできます。"info" がフラグの代わりに実験的情報を使っている場合は現在の "info" は書き換えられません。

`get_date`()
: メッセージの配送日時をエポックからの秒数を表わす浮動小数点数で返します。

`set_date`(*date*)
: メッセージの配送日時を *date* にセットします。*date* はエポックからの秒数を表わす浮動小数点数です。

`get_info`()
: メッセージの "info" を含む文字列を返します。このメソッドは実験的 (即ちフラグのリストでない) "info" にアクセスし、また変更するのに役立ちます。

`set_info`(*info*)
: "info" に文字列 *info* をセットします。

MaildirMessage インスタンスが mboxMessage や MMDFMessage のインスタンスに基づいて生成されるとき、Status: および X-Status: ヘッダは省かれ以下の変換が行われます:

結果の状態	mboxMessage または MMDFMessage の状態
"cur" サブディレクトリ	O フラグ
F フラグ	F フラグ
R フラグ	A フラグ
S フラグ	R フラグ
T フラグ	D フラグ

MaildirMessage インスタンスが MHMessage インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	MHMessage の状態
"cur" サブディレクトリ	"unseen" シーケンス
"cur" サブディレクトリおよび S フラグ	"unseen" シーケンス無し
F フラグ	"flagged" シーケンス
R フラグ	"replied" シーケンス

MaildirMessage インスタンスが BabylMessage インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	BabylMessage の状態
"cur" サブディレクトリ	"unseen" ラベル
"cur" サブディレクトリおよび S フラグ	"unseen" ラベル無し
P フラグ	"forwarded" または "resent" ラベル
R フラグ	"answered" ラベル
T フラグ	"deleted" ラベル

mboxMessage

class mboxMessage ([*message*])

mbox 固有の動作をするメッセージ。引数 *message* は Message のコンストラクタと同じ意味を持ちます。

mbox メールボックス中のメッセージは単一ファイルにまとめて格納されています。送り主のエンベロープアドレスおよび配送日時は通常メッセージの開始を示す "From " から始まる行に記録されますが、正確なフォーマットに関しては mbox の実装ごとに大きな違いがあります。メッセージの状態を示すフラグ、たとえば読んだかどうかあるいは重要だとマークを付けられているかどうかといったようなもの、は典型的には Status: および X-Status: に収められます。

規定されている mbox メッセージのフラグは以下の通りです:

フラグ	意味	説明
R	既読 (Read)	読んだ
O	古い (Old)	以前に MUA に発見された
D	削除 (Deleted)	削除予定
F	フラグ付き (Flagged)	重要だとマークされた
A	返答済み (Answered)	返答した

"R" および "O" フラグは Status: ヘッダに記録され、"D"、"F"、"A" フラグは X-Status: ヘッダに記録されます。フラグとヘッダは通常記述された順番に出現します。

mboxMessage インスタンスは以下のメソッドを提供します:

get_from()
: mbox メールボックスのメッセージの開始を示す "From " 行を表わす文字列を返します。先頭の "From " および末尾の改行は含まれません。

set_from(*from_*[, *time_*=*None*])
: "From " 行を *from_* にセットします。*from_* は先頭の "From " や末尾の改行を含まない形で指定しなければなりません。利便性のために、*time_* を指定して適切に整形して *from_* に追加させることができます。*time_* を指定する場合、それは struct_time インスタンス、time.strftime() に渡すのに適したタプル、または True (この場合 time.gmtime() を使います) のいずれかでなければなりません。

get_flags()
: 現在セットされているフラグを特定する文字列を返します。メッセージが規定された形式に準拠しているならば、結果は次の順に並べられたゼロまたは 1 回の 'R'、'O'、'D'、'F'、'A' です。

set_flags(*flags*)
: *flags* で指定されたフラグをセットして、他のフラグは下ろします。*flags* は並べられたゼロまたは 1 回の 'R'、'O'、'D'、'F'、'A' です。

add_flag(*flag*)
: *flags* で指定されたフラグをセットしますが他のフラグは変えません。一度に二つ以上のフラグをセットすることは、*flag* に 2 文字以上の文字列を指定すればできます。

remove_flag(*flag*)
: *flags* で指定されたフラグを下ろしますが他のフラグは変えません。一度に二つ以上のフラグを取り除くことは、*flag* に 2 文字以上の文字列を指定すればできます。

mboxMessage インスタンスが MaildirMessage インスタンスに基づいて生成されるとき、MaildirMessage インスタンスの配送日時に基づいて"From " 行が作り出され、次の変換が行われます:

結果の状態	MaildirMessage の状態
R フラグ	S フラグ
O フラグ	"cur" サブディレクトリ
D フラグ	T フラグ
F フラグ	F フラグ
A フラグ	R フラグ

mboxMessage インスタンスが MHMessage インスタンスに基づいて生成されるとき、以下の変換が行われます。

結果の状態	MHMessage 状態
R フラグ および O フラグ	"unseen" シーケンス無し
O フラグ	"unseen" シーケンス
F フラグ	"flagged" シーケンス
A フラグ	"replied" シーケンス

mboxMessage インスタンスが BabylMessage インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	BabylMessage の状態
R フラグ および O フラグ	"unseen" ラベル無し
O フラグ	"unseen" ラベル
D フラグ	"deleted" ラベル
A フラグ	"answered" ラベル

mboxMessage インスタンスが MMDFMessage インスタンスに基づいて生成されるとき、"From " 行はコピーされ全てのフラグは直接対応します:

結果の状態	MMDFMessage の状態
R フラグ	R フラグ
O フラグ	O フラグ
D フラグ	D フラグ
F フラグ	F フラグ
A フラグ	A フラグ

MHMessage

class MHMessage ([*message*])

MH 固有の動作をするメッセージ。引数 *message* は Message のコンストラクタと同じ意味を持ちます。

MH メッセージは伝統的な意味あいにおいてマークやフラグをサポートしません。しかし、MH メッセージにはシーケンスがあり任意のメッセージを論理的にグループ分けできます。いくつかのメールソフト (標準の **mh** や **nmh** はそうではありませんが) は他の形式におけるフラグとほぼ同じようにシーケンスを使います。

シーケンス	説明
unseen	読んではいないが既に MUA に見つけられている
replied	返答した
flagged	重要だとマークされた

MHMessage インスタンスは以下のメソッドを提供します:

get_sequences ()

このメッセージを含むシーケンスの名前のリストを返す。

set_sequences (*sequences*)

このメッセージを含むシーケンスのリストをセットする。

add_sequence (*sequence*)

sequence をこのメッセージを含むシーケンスのリストに追加する。

remove_sequence (*sequence*)

sequence をこのメッセージを含むシーケンスのリストから除く。

MHMessage インスタンスが MaildirMessage インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	MaildirMessage の状態
"unseen" シーケンス	S フラグ無し
"replied" シーケンス	R フラグ
"flagged" シーケンス	F フラグ

MHMessage インスタンスが mboxMessage や MMDFMessage のインスタンスに基づいて生成されるとき、Status: および X-Status: ヘッダは省かれ以下の変換が行われます:

結果の状態	mboxMessage または MMDFMessage の状態
"unseen" シーケンス	R フラグ無し
"replied" シーケンス	A フラグ
"flagged" シーケンス	F フラグ

MHMessage インスタンスが BabylMessage インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	BabylMessage の状態
"unseen" シーケンス	"unseen" ラベル
"replied" シーケンス	"answered" ラベル

BabylMessage

class BabylMessage ([*message*])

Babyl 固有の動作をするメッセージ。引数 *message* は Message のコンストラクタと同じ意味を持ちます。

ある種のメッセージラベルは **アトリビュート** と呼ばれ、規約により特別な意味が与えられています。アトリビュートは以下の通りです:

ラベル	説明
unseen	読んでいないが既に MUA に見つかっている
deleted	削除予定
filed	他のファイルまたはメールボックスにコピーされた
answered	返答済み
forwarded	転送された
edited	ユーザによって変更された
resent	再送された

デフォルトでは Rmail は可視ヘッダのみ表示する。BabylMessage クラスはしかし、オリジナルヘッダをより完全だという理由で使います。可視ヘッダは望むならそのように指示してアクセスすることができます。

BabylMessage インスタンスは以下のメソッドを提供します:

get_labels()

メッセージに付いているラベルのリストを返します。

set_labels(*labels*)

メッセージに付いているラベルのリストを *labels* にセットします。

add_label(*label*)

メッセージに付いているラベルのリストに *label* を追加します。

remove_label(*label*)

メッセージに付いているラベルのリストから *label* を削除します。

get_visible()

ヘッダがメッセージの可視ヘッダでありボディが空であるような Message インスタンスを返します。

set_visible(*visible*)

メッセージの可視ヘッダを *visible* のヘッダと同じにセットします。引数 *visible* は Message インス

タンスまたは email.Message.Message インスタンス、文字列、ファイル風オブジェクト (テキストモードで開かれてなければなりません) のいずれかです。

update_visible()

BabylMessage インスタンスのオリジナルヘッダが変更されたとき、可視ヘッダは自動的に対応して変更されるわけではありません。このメソッドは可視ヘッダを以下のように更新します。対応するオリジナルヘッダのある可視ヘッダはオリジナルヘッダの値がセットされます。対応するオリジナルヘッダの無い可視ヘッダは除去されます。そして、オリジナルヘッダにあって可視ヘッダに無い Date:、From:、Reply-To:、To:、CC:、Subject: は可視ヘッダに追加されます。

BabylMessage インスタンスが MaildirMessage インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	MaildirMessage の状態
"unseen" ラベル	S フラグ無し
"deleted" ラベル	T フラグ
"answered" ラベル	R フラグ
"forwarded" ラベル	P フラグ

BabylMessage インスタンスが mboxMessage や MMDFMessage のインスタンスに基づいて生成されるとき、Status: および X-Status: ヘッダは省かれ以下の変換が行われます:

結果の状態	mboxMessage または MMDFMessage の状態
"unseen" ラベル	R フラグ無し
"deleted" ラベル	D フラグ
"answered" ラベル	A フラグ

BabylMessage インスタンスが MHMessage インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	MHMessage の状態
"unseen" ラベル	"unseen" シーケンス
"answered" ラベル	"replied" シーケンス

MMDFMessage

class MMDFMessage(*[message]*)

MMDF 固有の動作をするメッセージ。引数 *message* は Message のコンストラクタと同じ意味を持ちます。

mbox メールボックスのメッセージと同様に、MMDF メッセージは送り主のアドレスと配送日時が最初の "From " で始まる行に記録されています。同様に、メッセージの状態を示すフラグは通常 Status: および X-Status: ヘッダに収められています。

よく使われる MMDF メッセージのフラグは mbox メッセージのものと同一で以下の通りです:

フラグ	意味	説明
R	既読 (Read)	読んだ
O	古い (Old)	以前に MUA に発見された
D	削除 (Deleted)	削除予定
F	フラグ付き (Flagged)	重要だとマークされた
A	返答済み (Answered)	返答した

"R" および "O" フラグは Status: ヘッダに記録され、"D"、"F"、"A" フラグは X-Status: ヘッダに記録されます。フラグとヘッダは通常記述された順番に出現します。

MMDFMessage インスタンスは mboxMessage インスタンスと同一の以下のメソッドを提供します:

get_from()

MMDF メールボックスのメッセージの開始を示す "From " 行を表わす文字列を返します。先頭の "From " および末尾の改行は含まれません。

set_from(*from_*[, *time_*=*None*])

"From " 行を *from_* にセットします。*from_* は先頭の "From " や末尾の改行を含まない形で指定しなければなりません。利便性のために、*time_* を指定して適切に整形して *from_* に追加させることができます。*time_* を指定する場合、それは struct_time インスタンス、time.strftime() に渡すのに適したタプル、または True (この場合 time.gmtime() を使います) のいずれかでなければなりません。

get_flags()

現在セットされているフラグを特定する文字列を返します。メッセージが規定された形式に準拠しているならば、結果は次の順に並べられたゼロまたは 1 回の 'R'、'O'、'D'、'F'、'A' です。

set_flags(*flags*)

flags で指定されたフラグをセットして、他のフラグは下ろします。*flags* は並べられたゼロまたは 1 回の 'R'、'O'、'D'、'F'、'A' です。

add_flag(*flag*)

flags で指定されたフラグをセットしますが他のフラグは変えません。一度に二つ以上のフラグをセットすることは、*flag* に 2 文字以上の文字列を指定すればできます。

remove_flag(*flag*)

flags で指定されたフラグを下ろしますが他のフラグは変えません。一度に二つ以上のフラグを取り除くことは、*flag* に 2 文字以上の文字列を指定すればできます。

MMDFMessage インスタンスが MaildirMessage インスタンスに基づいて生成されるとき、MaildirMessage インスタンスの配送日時に基づいて"From " 行が作り出され、次の変換が行われます:

結果の状態	MaildirMessage の状態
R フラグ	S フラグ
O フラグ	"cur" サブディレクトリ
D フラグ	T フラグ
F フラグ	F フラグ
A フラグ	R フラグ

MMDFMessage インスタンスが MHMessage インスタンスに基づいて生成されるとき、以下の変換が行われます。

結果の状態	MHMessage 状態
R フラグ および O フラグ	"unseen" シーケンス無し
O フラグ	"unseen" シーケンス
F フラグ	"flagged" シーケンス
A フラグ	"replied" シーケンス

MMDFMessage インスタンスが BabylMessage インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	BabylMessage の状態
R フラグ および O フラグ	"unseen" ラベル無し
O フラグ	"unseen" ラベル
D フラグ	"deleted" ラベル
A フラグ	"answered" ラベル

MMDFMessage インスタンスが mboxMessage インスタンスに基づいて生成されるとき、"From " 行はコピーされ全てのフラグは直接対応します:

結果の状態	mboxMessage の状態
R フラグ	R フラグ
O フラグ	O フラグ
D フラグ	D フラグ
F フラグ	F フラグ
A フラグ	A フラグ

7.3.3 例外

mailbox モジュールでは以下の例外クラスが定義されています:

class Error()
他の全てのモジュール固有の例外の基底クラス。

class NoSuchMailboxError()
メールボックスがあると思っていたが見つからなかった場合に送出されます。これはたとえば Mailbox のサブクラスを存在しないパスでインスタンス化しようとしたとき (かつ *create* パラメータは False であった場合)、あるいは存在しないフォルダを開こうとした時などに発生します。

class NotEmptyError()
メールボックスが空であることを期待されているときに空でない場合、たとえばメッセージの残っているフォルダを削除しようとした時などに送出されます。

class ExternalClashError()
メールボックスに関係したある条件がプログラムの制御を外れてそれ以上作業を続けられなくなった場合、たとえば他のプログラムが既に保持しているロックを取得しようとして失敗したとき、あるいは一意的に生成されたファイル名が既に存在していた場合などに送出されます。

class FormatError()
ファイル中のデータが解析できない場合、たとえば MH インスタンスが壊れた '.mh_sequences' ファイルを読もうと試みた場合などに送出されます。

7.3.4 撤廃されたクラスとメソッド

古いバージョンの mailbox モジュールはメッセージの追加や削除といったメールボックスの変更をサポートしていませんでした。また形式ごとのメッセージプロパティを表現するクラスも提供していませんでした。後方互換性のために、古いメールボックスクラスもまだ使うことができますが、できるだけ新しいクラスを使うべきです。

古いメールボックスオブジェクトは繰り返しと一つの公開メソッドだけを提供していました:

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。

ほとんどの古いメールボックスクラスは現在のメールボックスクラスと違う名前ですが、Maildir だけは例外です。そのため、新しい方の Maildir クラスには next() メソッドが定義され、コンストラクタも他の新しいメールボックスクラスとは少し異なります。

古いメールボックスのクラスで名前が新しい対応物と同じでないものは以下の通りです:

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)。

注意: このモジュールの実装上の理由により、*fp* オブジェクトはバイナリモードで開くようにしてください。特に Windows 上では注意が必要です。

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 *name time*' 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 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。

古いメールボックスクラスを撤廃された rfc822 モジュールではなく、email モジュールと使いたいならば、以下のようにできます:

```
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)
```

一方、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)
```

7.3.5 例

メールボックス中の面白そうなメッセージのサブジェクトを全て印字する簡単な例:

```
import mailbox
for message in mailbox.mbox('~/mbox'):
    subject = message['subject']       # Could possibly be None.
    if subject and 'python' in subject.lower():
        print subject
```

Babyl メールボックスから MH メールボックスへ全てのメールをコピーし、変換可能な全ての形式固有の情報を変換する:

```
import mailbox
destination = mailbox.MH('~/Mail')
for message in mailbox.Babyl('~/RMAIL'):
    destination.add(MHMessage(message))
```

幾つかのメーリングリストのメールをソートする例。他のプログラムと平行して変更を加えることでメー

ルが破損したり、プログラムを中断することでメールを失ったり、はたまた半端なメッセージがメールボックス中にあることで途中で終了してしまう、といったことを避けるように注意深く扱っている:

```
import mailbox
import email.Errors
list_names = ('python-list', 'python-dev', 'python-bugs')
boxes = dict((name, mailbox.mbox('~/email/%s' % name)) for name in list_names)
inbox = mailbox.Maildir('~/Maildir', None)
for key in inbox.iterkeys():
    try:
        message = inbox[key]
    except email.Errors.MessageParseError:
        continue                # The message is malformed. Just leave it.
    for name in list_names:
        list_id = message['list-id']
        if list_id and name in list_id:
            box = boxes[name]
            box.lock()
            box.add(message)
            box.flush()         # Write copy to disk before removing original.
            box.unlock()
            inbox.discard(key)
            break               # Found destination, so stop looking.
for box in boxes.itervalues():
    box.close()
```

7.4 mhlib — MH のメイルボックスへのアクセス機構

mhlib 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 mimetools.Message から導出されています。

7.4.1 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 ()
メイ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てはいけません。

openfolder(*name*)
新たな開かれたフォルダオブジェクトを返します。

7.4.2 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。

7.4.3 Message オブジェクト

`Message` クラスは `mimetools.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)。

7.5 `mimetools` — 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 `mimetools` 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 (7.1 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 (7.10 節):

`mimetools.Message` のベースクラスを提供する。

`multifile` モジュール (7.9 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。

7.5.1 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。

7.6 `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、符合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、これはデフォー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、'.tgz' 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。デフォールトでは、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、もしデフォー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、デフォー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。

モジュールの使用例:

```
>>> import mimetypes
>>> mimetypes.init()
>>> mimetypes.knownfiles
['/etc/mime.types', '/etc/httpd/mime.types', ... ]
>>> mimetypes.suffix_map['.tgz']
'.tar.gz'
>>> mimetypes.encodings_map['.gz']
'gzip'
>>> mimetypes.types_map['.tgz']
'application/x-tar-gz'
```

7.6.1 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、'.tgz' 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。

7.7 `MimeWriter` — 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、`MimeWriter` 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 `MimeWriter` を定義します。この `MimeWriter` 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。`MimeWriter` 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るようにします。

class **MimeWriter**(*fp*)
`MimeWriter` 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。

7.7.1 MimeWriter オブジェクト

`MimeWriter` 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。デフォー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) るようにします。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。コンテント-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、コンテント-型定義のための追加のパラメータを与えます。 *prefix* は、そのデフォールトが先頭への挿入以外は addheader() でのように働きます。

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() でのように働きます。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。

7.8 mimify — 電子メイルメッセージの 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イルメッセージから MIME へ、および MIME から電子メイ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。電子メイ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 メイル

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 化されたパートを復号化します。*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
文字セットがメイ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 (7.14 節):
MIME quoted-printable 形式ファイルのエンコードおよびデコード。

7.9 `multifile` — 個別の部分を含んだファイル群のサポート

リリース 2.5 以降で撤廃された仕様です。 `multifile` モジュールよりも `email` パッケージを使うべきです。このモジュールは後方互換性のためだけに存在しています。

`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えるようにし、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。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 (7.1 節):

網羅的な電子メイル操作パッケージ; `multifile` モジュールに取って代わります。

7.9.1 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。*pis* 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。この境界文字列の修飾されたバージョンが入力行に見つかった場合、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 (どちらであるかは修飾に依存します。RFC 2045 を参照してください)。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。

7.9.2 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()
```

7.10 rfc822 — RFC 2822 準拠のメイルヘッダ読み出し

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 [10]で定義されている “電子メイルメッセージ” 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イルプログラムによって生成されるメイ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; 通常の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。

[10] 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。

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。

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 (7.1 節):
網羅的な電子メイ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 (7.3 節):
エンドユーザのメイ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 (7.5 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。

7.10.1 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@cwi.nl (Jack Jansen)' が入っている場合、m.getaddr('From') はペア ('Jack Jansen', 'jack@cwi.nl') になります。また、'Jack Jansen <jack@cwi.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、複数のメイ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 形式のメイ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。

7.10.2 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* との集合和をとり、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。

7.11 base64 — RFC 3548: Base16, Base32, Base64 テータの符号化

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、

'd3d3LnB5dGhvbi5vcmc=\n' とエンコードされます。

このモジュールは、 RFC 3548 で定められた仕様によるデータの符号化 (エンコード、encoding) および復元 (デコード、decoding) を提供します。この RFC 標準では Base16, Base32 および Base64 が定義されており、これはバイナリ文字列とテキスト文字列とをエンコードあるいはデコードするためのアルゴリズムです。変換されたテキスト文字列は email で確実に送信したり、URL の一部として使用したり、HTTP POST リクエストの一部に含めることができます。これらの符号化アルゴリズムは **uuencode** で使われているものとは別物です。

このモジュールでは 2 つのインターフェイスが提供されています。現代的なインターフェイスは、これら 3 種類のアルファベット集合を使った文字列オブジェクトのエンコードおよびデコードをすべてサポートします。一方、レガシーなインターフェイスは、文字列とともにファイル風のオブジェクトに対するエンコード / デコードを提供しますが、Base64 標準のアルファベット集合しか使いません。

現代的なインターフェイスは以下のものを提供します:

b64encode(*s*[, *altchars*])

Base64 をつかって、文字列を エンコード (符号化) します。

s はエンコードする文字列です。オプション引数 *altchars* は最低でも 2 の長さをもつ文字列で (これ以降の文字は無視されます)、これは + と / の代わりに使われる代替アルファベットを指定します。これにより、アプリケーションはたとえば URL やファイルシステムの影響をうけない Base64 文字列を生成することができます。デフォルトの値は None で、これは標準の Base64 アルファベット集合が使われることを意味します。

エンコードされた文字列が返されます。

b64decode(*s*[, *altchars*])

Base64 文字列をデコード (復元) します。

s にはデコードする文字列を渡します。オプション引数の *altchars* は最低でも 2 の長さをもつ文字列で (これ以降の文字は無視されます)、これは + と / の代わりに使われる代替アルファベットを指定します。

デコードされた文字列が返されます。*s* が正しくパディングされていなかったり、規定のアルファベット以外の文字が含まれていた場合には TypeError が発生します。

standard_b64encode(*s*)

標準の Base64 アルファベット集合をもちいて文字列 *s* をエンコード (符号化) します。

standard_b64decode(*s*)

標準の Base64 アルファベット集合をもちいて文字列 *s* をデコード (復元) します。

urlsafe_b64encode(*s*)

URL 用に安全なアルファベット集合をもちいて文字列 *s* をエンコード (符号化) します。これは、標準の Base64 アルファベット集合にある + のかわりに - を使い、/ のかわりに _ を使用します。

urlsafe_b64decode(*s*)

URL 用に安全なアルファベット集合をもちいて文字列 *s* をデコード (復元) します。これは、標準の Base64 アルファベット集合にある + のかわりに - を使い、/ のかわりに _ を使用します。

b32encode(*s*)

Base32 をつかって、文字列をエンコード (符号化) します。*s* にはエンコードする文字列を渡し、エンコードされた文字列が返されます。

b32decode(*s*[, *casefold*[, *map01*]])

Base32 をつかって、文字列をデコード (復元) します。

s にはエンコードする文字列を渡します。オプション引数 *casefold* は小文字のアルファベットを受けつけるかどうかを指定します。セキュリティ上の理由により、デフォルトではこれは False になっ

ています。

RFC 3548 は付加的なマッピングとして、数字の 0 (零) をアルファベットの O (オー) に、数字の 1 (壱) をアルファベットの I (アイ) または L (エル) に対応させることを許しています。オプション引数は *map01* は、`None` でないときは、数字の 1 をどの文字に対応づけるかを指定します (map01 が `None` でないとき、数字の 0 はつねにアルファベットの O (オー) に対応づけられます)。セキュリティ上の理由により、これはデフォルトでは `None` になっているため、0 および 1 は入力として許可されていません。

デコードされた文字列が返されます。*s* が正しくパディングされていなかったり、規定のアルファベット以外の文字が含まれていた場合には `TypeError` が発生します。

b16encode(*s*)

Base16 をつかって、文字列をエンコード (符号化) します。

s にはエンコードする文字列を渡し、エンコードされた文字列が返されます。

b16decode(*s*[, *casefold*])

Base16 をつかって、文字列をデコード (復元) します。

s にはエンコードする文字列を渡します。オプション引数 *casefold* は小文字のアルファベットを受けつけるかどうかを指定します。セキュリティ上の理由により、デフォルトではこれは `False` になっています。

デコードされた文字列が返されます。*s* が正しくパディングされていなかったり、規定のアルファベット以外の文字が含まれていた場合には `TypeError` が発生します。

レガシーなインターフェイスは以下のものを提供します:

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 *input*`.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 *input*`.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。

モジュールの使用例:

```
>>> import base64
>>> encoded = base64.b64encode('data to be encoded')
>>> encoded
'ZGF0YSB0byBiZSBlbmNvZGVk'
>>> data = base64.b64decode(encoded)
>>> data
'data to be encoded'
```

参考資料:

`binascii` モジュール (7.13 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 In*
Section 5.2, "Base64 Content-Transfer-Encoding," provides the definition of the base64 encoding.

7.12 `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送出される例外です。

参考資料:

`binascii` モジュール (7.13 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。

7.12.1 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。

7.13 `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`、`base64` 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。binascii モジュールは、高レベルなモジュールで利用される、高速な C で書かれた低レベル関数を提供しています。

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 *istext* が存在し、かつその値が真であれば、改行はエンコードされませんが、行末の空白文字は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る場合、空白文字は RFC1522 にしたが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ず、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りません。

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りません。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 (7.11 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 (7.12 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 (7.15 節):
UNIX で使われる UU エンコードのサポート。

`quopri` モジュール (7.14 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。

7.14 `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 円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 (7.8 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 (7.11 節):

MIME base64 形式データのエンコードおよびデコード

7.15 `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いて、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'` か`'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か`'-'` 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。

誤った実装の uuencoder による入力で、エラーから復旧できた場合、decode() は標準エラー出力に警告を表示するかもしれません。*quiet* を真にすることでこの警告を抑制することができます。

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 (7.13 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。

八章

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; 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。

HTMLParser	HTML と XHTML を扱えるシンプルなパーザ。
sgmllib	HTML を解析するのに必要な機能だけを備えた SGML パーザ。
htmllib	HTML 文書の解析器。
htmlentitydefs	HTML 一般エンティティの定義。
xml.parsers.expat	Expat による、検証を行わない XML パーザへのインタフェース
xml.dom	Python のための文書オブジェクトモデル API。
xml.dom.minidom	軽量な文書オブジェクトモデルの実装。
xml.dom.pulldom	SAX イベントからの部分的な DOM ツリー構築のサポート。
xml.sax	SAX2 基底クラスと有用な関数のパッケージ
xml.sax.handler	SAX イベント・ハンドラの基底クラス
xml.sax.saxutils	SAX とともに使う有用な関数とクラスです。
xml.sax.xmlreader	SAX 準拠の XML パーサが実装すべきインターフェースです。
xml.etree.ElementTree	Implementation of the ElementTree API.

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。

8.1 `HTMLParser` — HTML および XHTML のシンプルなパーザ

2.2 で追加された仕様です。

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。

htmllib 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。

例外も定義されています:

exception HTMLParseError

パーズ中にエラーに遭遇した場合に HTMLParser クラスが送出する例外です。この例外は三つの属性を提供しています: msg はエラーの内容を説明する簡単なメッセージ、lineno は壊れたマークアップ構造を検出した場所の行番号、offset は問題のマークアップ構造の行内での開始位置を示す文字数です。

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なことがあります。

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、'`-`' および '`-`' 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。

8.1.1 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
```

8.2 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。

例外が以下のように定義されます:

exception SGMLParseError

SGMLParser クラスで構文解析中にエラーに出逢うとこの例外が発生します。 2.1 で追加された仕様です。

SGMLParser インスタンスは以下の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 HTML タグ <PAINTEXT> 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なことがあります。

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ます。*value* 内の二重引用符とバックスラッシュも変換され、と同時に知られている文字参照および知られているエンティティ参照でセミコロンで終端されているものも変換されます (通常、エンティティ参照は任意の非英数文字で終端されてよいのですが、これを許すと非常に一般的な<A HREF="url?spam=1&eggs=2"> において eggs が正当なエンティティ参照であるようなケースを破綻させます)。

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。 2.5 で追加された仕様: 属性値中のエンティティおよび文字参照の扱い

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。基底クラスの実装は、convert_charref() を使って参照を文字列に変換します。もしそのメソッドが文字列を返せば handle_data() を呼び出します。そうでなければ、エラーを処理するために unknown_charref(*ref*) が呼び出されます。 2.5 で変更された仕様: ハードコードされた変換に代わり convert_charref() を使います

convert_charref(*ref*)

文字参照を文字列に変換するか、None を返します。*ref* は文字列として渡される参照です。基底クラスでは *ref* は 0-255 の範囲の十進数でなければなりません。そしてコードポイントをメソッド convert_codepoint() を使って変換します。もし *ref* が不正もしくは範囲外ならば、None を返します。このメソッドはデフォルト実装の handle_charref から、あるいは属性値パーザから呼び出されます。 2.5 で追加された仕様です。

convert_codepoint(*codepoint*)

コードポイントを str の値に変換します。もしそれが適切ならばエンコーディングをここで扱うこともできますが、sgmllib の残りの部分はこの問題に関知しません。 2.5 で追加された仕様です。

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、*ref* を convert_entityref() に渡して変換します。変換結果が返された場合、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&、 &apos、>、 <、および" の変換を定義しています。 2.5 で変更された仕様: ハードコードされた変換に代わり convert_entityref() を使います

convert_entityref(*ref*)
名前付きエンティティ参照を str の値に変換するか、または None を返します。変換結果は再パーズしません。 *ref* はエンティティの名前部分だけです。デフォルトの実装ではインスタンス (またはクラス) 変数の entitydefs というエンティティ名から対応する文字列へのマッピングから *ref* を探します。もし *ref* に対応する文字列が見つからなければメソッドは None を返します。このメソッドは handle_entityref() のデフォルト実装からおよび属性値パーザから呼び出されます。 2.5 で追加された仕様です。

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とは別に、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 start_*tag* メソッドが定義されていない開始タグ *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ってい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。

8.3 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 sgmllib 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 sgmllib.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 マークアップ文を含む場合、それらの文は即座に処理されます; 不完全なマークアップ構造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パーザクラスと例外を一つづつ定義しています:

class HTMLParser (*formatter*)
基底となる HTML パーザクラスです。XHTML 1.0 仕様 (http://www.w3.rog/TR/xhtml1) 勧告で要求されている全てのエンティティ名をサポートしています。

exception HTMLParseError
HTMLParser クラスがパーズ処理中にエラーに遭遇した場合に送出する例外です。 2.4 で追加された仕様です。

参考資料:

formatter モジュール (33.1 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 (8.1 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 では正しくないとされる SGML 構文のいくつかは実装されていません。

htmlentitydefs モジュール (8.4 節):
XHTML 1.0 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 (8.2 節):
HTMLParser の基底クラス。

8.3.1 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タ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。

8.4 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。

8.5 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、8.5.2 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

8.5.1 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もできます。

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が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されます: 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。

UseForeignDTD([*flag*])
flag の値をデフォルトの true にすると、Expat は代わりの DTD をロードするため、すべての引数に None を設定して ExternalEntityRefHandler を呼び出します。XML 文書が文書型定義を持っていなければ、 ExternalEntityRefHandler が呼び出しますが、StartDoctypeDeclHandler と EndDoctypeDeclHandler は呼び出されません。

flag に false を与えると、メソッドが前回呼ばれた時の true の設定が解除されますが、他には何も起こりません。

このメソッドは Parse() または ParseFile() メソッドが呼び出される前にだけ呼び出されます; これら 2 つのメソッドのどちらかが呼び出されたあとにメソッドが呼ばれると、code に定数 errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING が設定されて例外 ExpatError が送出されます。

2.3 で追加された仕様です。

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。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リ: 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 False 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。Python がユニコードサポートつきでビルドされている場合、この値はデフォルトで True です。 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です

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。

以下の属性は xmlparser オブジェクトがその時パースしている位置に関する値を保持しています。コールバックがパースイベントを報告している間、これらの値はイベントの生成した文字列の先頭の位置を指し示します。コールバックの外から参照された時には、(対応するコールバックであるかにかかわらず) 直前のパースイベントの位置を示します。 2.4 で追加された仕様です。

CurrentByteIndex
パーサへの入力の、現在のバイトインデックス。

CurrentColumnNumber
パーサへの入力の、現在のカラム番号。

CurrentLineNumber
パーサへの入力の、現在の行番号。

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 度づつ、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 entitiy) 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 `EndCdataSectionHandler` 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。

8.5.2 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。

8.5.3 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
```

8.5.4 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。

8.5.5 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
入力に不完全な文字が見つかりました。

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。

XML_ERROR_UNCLOSED_CDATA_SECTION
CDATA セクションが閉じられていません。

XML_ERROR_EXTERNAL_ENTITY_HANDLING

XML_ERROR_NOT_STANDALONE
XML 文書が “standalone” だと宣言されており `NotStandaloneHandler` が設定され `0` が返されているにもかかわらず、パーサは “standalone” ではないと判別しました。

XML_ERROR_UNEXPECTED_STATE

XML_ERROR_ENTITY_DECLARED_IN_PE

XML_ERROR_FEATURE_REQUIRES_XML_DTD
その操作を完了するには DTD のサポートが必要ですが、Expat が DTD のサポートをしない設定になっています。これは `xml.parsers.expat` モジュールの標準的なビルドでは報告されません。

XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING
パースが始まったあとで動作の変更が要求されました。これはパースが開始される前にのみ変更可能です。（現在のところ） `UseForeignDTD()` によってのみ送出されます。

XML_ERROR_UNBOUND_PREFIX
名前空間の処理を有効すると宣言されていないプレフィックスが見つかります。

XML_ERROR_UNDECLARING_PREFIX
XML 文書はプレフィックスに対応した名前空間宣言を削除しようとしました。

XML_ERROR_INCOMPLETE_PE
パラメータエンティティは不完全なマークアップを含んでいます。

XML_ERROR_XML_DECL
XML 文書中に要素がありません。

XML_ERROR_TEXT_DECL
外部エンティティ中のテキスト宣言にエラーがあります。

XML_ERROR_PUBLICID
パブリック ID 中に許可されていない文字があります。

XML_ERROR_SUSPENDED
要求された操作は一時停止されたパーサで行われていますが、許可されていない操作です。このエラーは追加の入力を行なおうとしている場合、もしくはパーサが停止しようとしている場合にも送出されます。

XML_ERROR_NOT_SUSPENDED
パーサを一時停止しようとしましたが、停止されませんでした。

XML_ERROR_ABORTED
Python アプリケーションには通知されません。

XML_ERROR_FINISHED
要求された操作で、パース対象となる入力が完了したと判断しましたが、入力は受理されませんでした。このエラーは追加の入力を行なおうとしている場合、もしくはパーサが停止しようとしている場合に送出されます。

XML_ERROR_SUSPEND_PE

8.6 `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もありません。Python では、各々の DOM 実装で `getDOMImplementation()` が定義されているはずです。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 8.6.3 、“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。

Python Language Mapping Specification
(http://www.omg.org/docs/formal/02-11-05.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。

8.6.1 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*) のペアからなるシーケンスで、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

Namespaces 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。

8.6.2 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
DOMImplementation	8.6.2	根底にある実装へのインタフェース。
Node	8.6.2	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。
NodeList	8.6.2	ノードの列に対するインタフェース。
DocumentType	8.6.2	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。
Document	8.6.2	ドキュメント全体を表現するオブジェクト。
Element	8.6.2	ドキュメント階層内の要素ノード。
Attr	8.6.2	階層ノード上の属性値。
Comment	8.6.2	ソースドキュメント内のコメント表現。
Text	8.6.2	ドキュメント内のテキスト記述を含むノード。
ProcessingInstruction	8.6.2	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*)
機能名 *feature* とバージョン番号 *version* で識別される機能（feature）が実装されていれば true を返します。

createDocument (*namespaceUri, qualifiedName, doctype*)
新たな (DOM のスーパークラスである)Document クラスのオブジェクトを返します。このクラスは *namespaceUri* と *qualifiedName* が設定された子クラス Element のオブジェクトを所有しています。*doctype* は createDocumentType() によって生成された DocumentType クラスのオブジェクト、または None である必要があります。Python DOM API では、子クラスである Element を作成しないことを示すために、はじめの 2 つの引数を None に設定することができます。

createDocumentType (*qualifiedName, publicId, systemId*)
新たな DocumentType クラスのオブジェクトを返します。このオブジェクトは *qualifiedName*、*publicId*、そして *systemId* 文字列をふくんでおり、XML 文書の形式情報を表現しています。

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。もし *refChild* が `None` なら、*newChild* を子ノードリストの最後に挿入します。

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、ノードからなるシーケンスを表現します。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 `getElementsByTagName()` および `getElementsByTagNameNS()` 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づつ定義しています:

item(*i*)
シーケンスに *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てはならず、シーケンスの長さ以上であってはなりません。

length

シーケンス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 のシーケンス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。

getElementsByTagNameNS(*tagName*)
Document クラス内における同名のメソッドと同じです。

hasAttribute(*name*)
指定要素に *name* で渡した名前の属性が存在していれば true を返します。

hasAttributeNS(*namespaceURI, localName*)

指定要素に *namespaceURI* と *localName* で指定した名前の属性が存在していれば true を返します。

getAttribute(*name*)
name で指定した属性の値を文字列として返します。もし、属性が存在しない、もしくは属性に値が設定されていない場合、空の文字列が返されます。

getAttributeNode(*attrname*)
attrname で指定された属性の `Attr` ノードを返します。

getAttributeNS(*namespaceURI, localName*)
namespaceURI と *localName* によって指定した属性の値を文字列として返します。もし、属性が存在しない、もしくは属性に値が設定されていない場合、空の文字列が返されます。

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(*name*)
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、`NotFoundErr` 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(*name, 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的な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ものですが、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 区域の一部であるという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
定義されてい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

8.6.3 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ほとんど同じように行います。

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パ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。

Python DOM API では、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。

この IDL での定義は W3C DOM API の全ての要件を実装しているわけではありません。例えば、一部のオブジェクトの概念や `getElementsByTagName()` が “live” であることなどです。Python DOM API はこれらの要件を実装することを強制しません。

8.7 `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しれません (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。

8.7.1 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もかまいません。

writexml(*writer*[,*indent*=""[,*addindent*=""[,*newl*=""]]])
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。*indent* パラメタには現在のノードのインデントを指定します。*addindent* パラメタには現在のノードの下にサブノードを追加する際のインデント増分を指定します。*newl* には、改行時に行末を終端する文字列を指定します。

2.1 で変更された仕様: 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 で変更された仕様: 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。

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。

2.3 で変更された仕様: *encoding* が追加されました

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 で変更された仕様: 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。

8.7.2 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)
```

8.7.3 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。

8.8 `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

8.8.1 DOMEventStream オブジェクト

`getEvent`()
...

`expandNode`(*node*)
...

`reset`()
...

8.9 `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 (8.10 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 (8.11 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 (8.12 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

8.9.1 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。

8.10 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 はコンテント・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.handler 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: 外部パラーメタ・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。

8.10.1 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充分でしょう。

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、あるいは入力の最後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 リーダは `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 パラー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 パラー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 リーダ・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(*whitespace*)
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。

8.10.2 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.

8.10.3 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。

8.10.4 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 しないとパースは継続されますが、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、すぐにパース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 した後もパースを継続し、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、パースは中断されてしまいます。

8.11 `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 '`&`'、'`<`'、'`>`' 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。戻り値はクォート済みの *data* で、置換する文字の追加も可能です。quoteattr() はクォートすべき文字を *data* の文脈から判断し、クォ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・クォート、ダブル・クォートがあれば、両方ともエンコードし、全体をダブルクォートで囲みます。戻り値の文字列はそのままで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 をコンテント・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。

8.12 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 (8.12.5 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 8.12.6 参照) の実装です。

8.12.1 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

テント・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。

8.12.2 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。

8.12.3 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。

8.12.4 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 に接続しようとします。

getCharacterStream()
この入力ソースの文字ストリームを取得します。

8.12.5 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。

8.12.6 `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 8.12.5 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。

8.13 `xml.etree.ElementTree` — ElementTree XML API

2.5 で追加された仕様です。

エレメント型は柔軟性のあるコンテナオブジェクトで、階層的データ構造をメモリーに格納するようにデザインされています。この型は言わばリストと辞書の間の子のようなものです。

各エレメントは関連する多くのプロパティを具えています:

- このエレメントがどういう種類のデータを表現しているかを同定する文字列であるタグ (別の言い方をすればエレメントの型)
- 幾つもの属性 (Python 辞書に収められます)
- テキスト文字列
- オプションの末尾文字列
- 幾つもの子エレメント (Python シーケンスに収められます)

エレメントのインスタンスを作るには、Element や SubElement といったファクトリー関数を使います。

`ElementTree` クラスはエレメントの構造を包み込み、それと XML を行き来するのに使えます。

この API の C 実装である `xml.etree.cElementTree` も使用可能です。

8.13.1 関数

`Comment`(*text*)

コメント・エレメントのファクトリーです。このファクトリー関数は XML コメントにシリアライズされる特別な要素を作ります。コメント文字列は、8-bit ASCII 文字列でも Unicode 文字列でも構いません。*text* はそのコメント文字列を含んだ文字列です。

戻り値:

コメントを表わすエレメントのインスタンス。

`dump`(*elem*)

エレメントの木もしくはエレメントの構造を sys.stdout に書き込みます。この関数はデバグ目的でだけ使用してください。

出力される形式の正確なところは実装依存です。このバージョンでは、通常の XML ファイルとして書き込まれます。

elem はエレメントの木もしくは個別のエレメントです。

`Element`(*tag*[, *attrib*][, ***extra*])

エレメントのファクトリー。この関数は標準エレメント・インタフェースを実装したオブジェクトを返します。このオブジェクトのクラスや型が正確に何であるかは実装に依存しますが、いつでもこのモジュールにある _ElementInterface クラスと互換性があります。

エレメント名、アトリビュート名およびアトリビュート値は 8-bit ASCII 文字列でも Unicode 文字列でも構いません。*tag* はエレメント名です。*attrib* はオプションの辞書で、エレメントのアトリビュートを含んでいます。*extra* は追加のアトリビュートで、キーワード引数として与えられたものです。

戻り値:

エレメント・インスタンス。

`fromstring`(*text*)

文字列定数で与えられた XML 断片を構文解析します。XML 関数と同じです。*text* は XML データを含んだ文字列です。

戻り値:

エレメント・インスタンス。

iselement(*element*)

オブジェクトが正当なエレメント・オブジェクトであるかをチェックします。*element* はエレメント・インスタンスです。

戻り値:

引数がエレメント・オブジェクトならば真値。

iterparse(*source*[, *events*])

XML 断片を構文解析してエレメントの木を漸増的に作っていき、その間進行状況をユーザーに報告します。*source* は XML データを含むファイル名またはファイル風オブジェクト。*events* は報告すべきイベントのリスト。省略された場合は “end” イベントだけが報告されます。

戻り値:

(イベント, エレメント) イテレータ。

parse(*source*[, *parser*])

XML 断片を構文解析してエレメントの木にしていきます。*source* は XML データを含むファイル名またはファイル風オブジェクト。*parser* はオプションの構文解析器インスタンスです。これが与えられない場合、標準の XMLTreeBuilder 構文解析器が使われます。

戻り値:

ElementTree インスタンス。

ProcessingInstruction(*target*[, *text*])

PI エレメントのファクトリー。このファクトリー関数は XML の処理命令 (processing instruction) としてシリアライズされる特別なエレメントを作ります。*target* は PI ターゲットを含んだ文字列です。*text* は与えられるならば PI コンテンツを含んだ文字列です。

戻り値:

PI を表わすエレメント・インスタンス。

SubElement(*parent, tag*[, *attrib*] [, ***extra*])

部分エレメントのファクトリー。この関数はエレメント・インスタンスを作り、それを既存のエレメントに追加します。

エレメント名、アトリビュート名およびアトリビュート値は 8-bit ASCII 文字列でも Unicode 文字列でも構いません。*parent* は親エレメントです。*tag* はエレメント名です。*attrib* はオプションの辞書で、エレメントのアトリビュートを含んでいます。*extra* は追加のアトリビュートで、キーワード引数として与えられたものです。

戻り値:

エレメント・インスタンス。

tostring(*element*[, *encoding*])

XML エレメントを全ての子エレメントを含めて表現する文字列を生成します。*element* はエレメント・インスタンス。*encoding* は出力エンコーディング (デフォルトは US-ASCII) です。

戻り値:

XML データを含んだエンコードされた文字列。

XML(*text*)

文字列定数で与えられた XML 断片を構文解析します。この関数は Python コードに「XML リテラル」を埋め込むのに使えます。*text* は XML データを含んだ文字列です。

戻り値:

エレメント・インスタンス。

XMLID(*text*)

文字列定数で与えられた XML 断片を構文解析し、エレメント ID からエレメントへのマッピングを

与える辞書も同時に返します。*text* は XML データを含んだ文字列です。

戻り値:

エレメント・インスタンスと辞書のタプル。

8.13.2 ElementTree オブジェクト

class ElementTree ([*element,*] [*file*])

ElementTree ラッパー・クラス。このクラスはエレメントの全階層を表現し、さらに標準 XML との相互変換を追加しています。

element は根エレメントです。木はもし *file* が与えられればその XML の内容により初期化されます。

_setroot (*element*)

この木の根エレメントを置き換えます。したがって現在の木の内容は破棄され、与えられたエレメントが代わりに使われます。注意して使ってください。*element* はエレメント・インスタンスです。

find (*path*)

子孫エレメントの中で与えられたタグを持つ最初のものを見つけます。getroot().find(path) と同じです。*path* は探したいエレメントです。

戻り値:

最初に条件に合ったエレメント、または見つからない時は None。

findall (*path*)

子孫エレメントの中で与えられたタグを持つものを全て見つけます。getroot().findall(path) と同じです。*path* は探したいエレメントです。

戻り値:

全ての条件に合ったエレメントのリストまたはイテレータで、セクション順です。

findtext (*path*[, *default*])

子孫エレメントの中で与えられたタグを持つ最初のもののテキストを見つけます。getroot().findtext(path) と同じです。*path* は探したい直接の子エレメントです。*default* はエレメントが見つからなかった場合に返される値です。

戻り値:

条件に合った最初のエレメントのテキスト、または見つからなかった場合にはデフォルト値。もしエレメントが見つかったもののテキストがなかった場合には、このメソッドは空文字列を返す、ということに気をつけてください。

getiterator ([*tag*])

根エレメントに対する木を巡るイテレータを作ります。イテレータは木の全てのエレメントに渡ってセクション順にループします。*tag* は探したいタグです (デフォルトでは全てのエレメントを返します)。

戻り値:

イテレータ。

getroot ()

この木の根エレメントを返します。

戻り値:

エレメント・インスタンス。

parse (*source*[, *parser*])

外部の XML 断片をこのエレメントの木に読み込みます。*source* は XML データを含むファイル名ま

たはファイル風オブジェクト。*parser* はオプションの構文解析器インスタンスです。これが与えられない場合、標準の XMLTreeBuilder 構文解析器が使われます。

戻り値:

断片の根エレメント。

write (*file*[, *encoding*])

エレメントの木をファイルに XML として書き込みます。*file* はファイル名またはファイル風オブジェクトで書き込み用に開かれたもの。*encoding* は出力エンコーディング (デフォルトは US-ASCII) です。

8.13.3 QName オブジェクト

class QName (*text_or_uri*[, *tag*])

QName ラッパー。このクラスは QName アトリビュート値をラップし、出力時に真っ当な名前空間の扱いを得るために使われます。*text_or_uri* は {uri}local という形式の QName 値を含む文字列、または tag 引数が与えられた場合には QName の URI 部分の文字列です。*tag* が与えられた場合、一つめの引数は URI と解釈され、この引数はローカル名と解釈されます。

戻り値:

QName を表わす不透明オブジェクト。

8.13.4 TreeBuilder オブジェクト

class TreeBuilder ([*element_factory*])

汎用のエレメント構造ビルダー。これは start、data、end のメソッド呼び出しの列を整形式のエレメント構造に変換します。このクラスを使うと、好みの XML 構文解析器、または他の XML に似た形式の構文解析器を使って、エレメント構造を作り出すことができます。*element_factory* が与えられた場合には新しいエレメント・インスタンスを作る際にこれを呼び出します。

close ()

構文解析器のバッファをフラッシュし、最上位の文書エレメントを返します。

戻り値:

エレメント・インスタンス。

data (*data*)

現在のエレメントにテキストを追加します。*data* は文字列です。8-bit ASCII 文字列もしくは Unicode 文字列でなければなりません。

end (*tag*)

現在のエレメントを閉じます。*tag* はエレメントの名前です。

戻り値:

閉じられたエレメント。

start (*tag, attrs*)

新しいエレメントを開きます。*tag* はエレメントの名前です。*attrs* はエレメントのアトリビュートを保持した辞書です。

戻り値:

開かれたエレメント。

8.13.5 XMLTreeBuilder オブジェクト

class `XMLTreeBuilder`(*[html,] [target]*)

XML ソースからエレメント構造を作るもので、expat 構文解析器に基づいています。*html* は前もって定義された HTML エンティティです。このオプションは現在の実装ではサポートされていません。*target* はターゲットとなるオブジェクトです。省略された場合、標準の TreeBuilder クラスのインスタンスが使われます。

`close`()

構文解析器にデータを供給するのを終わりにします。

戻り値:

エレメント構造。

`doctype`(*name, pubid, system*)

doctype 宣言を扱います。*name* は doctype の名前です。*pubid* は公開識別子です。*system* はシステム識別子です。

`feed`(*data*)

構文解析器にデータを供給します。

data はエンコードされたデータです。

九章

File Formats

The modules described in this chapter parse various miscellaneous file formats that aren't markup languages or are related to e-mail.

この章で説明されるモジュールは様々な (マークアップやでないものや E メールの) ファイルフォーマットを構文解析します。

csv	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。
ConfigParser	Configuration file parser.
robotparser	'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。
netrc	'.netrc' ファイル群の読み出し。
xdrlib	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。

9.1 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 UTF-8 または印字可能な ASCII にしなければなりません。これについては 9.1.5 節の例を参照してください。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)

9.1.1 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'*][, *fmtparam*])
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 — ファイルオブジェクトでもリストでも構いません。*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 *fmtparam* 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、 9.1.2 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。

2.5 で変更された仕様: パーサが複数行に亘るクオートされたフィールドに関して厳格になりました。以前は、クオートされたフィールドの中で終端の改行文字無しに行が終わった場合、返されるフィールドには改行が挿入されていましたが、この振る舞いはフールドの中に復帰文字を含むようなファイルを読むときに問題を起こしていました。そこでフィールドに改行文字を挿入せずに返すように改められました。この結果、フィールドに埋め込まれた改行文字が重要ならば、入力は改行文字を保存するような仕方で複数行に分割されなければなりません。

writer(*csvfile*[, *dialect='excel'*][, *fmtparam*])
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 *fmtparam* 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、 9.1.2 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*][, *fmtparam*])
dialect を *name* と関連付けます。*name* は文字列か Unicode オブジェクトでなければなりません。表現形式 (dialect) は Dialect のサブクラスを渡すか、またはキーワード引数 *fmtparam*、もしくは両方で指定できますが、キーワード引数の方が優先されます。表現形式と書式化パラメタについてより詳しいことは 9.1.2 節「Dialect クラスと書式化パラメタ」を参照してください。

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。

field_size_limit([*new_limit*])

パーサが許容する現在の最大フィールドサイズを返します。*new_limit* が渡されたときは、その値が新しい上限になります。 2.5 で追加された仕様です。

`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*=`None`,[, *restkey*=`None`[, *restval*=`None`[, *dialect*=`'excel'`[, **args*, ***kwds*]]]])

省略可能な *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。*fieldnames* パラメタが無い場合には、*csvfile* の最初の行の値がフィールド名として利用されます。読み出された行が *fieldnames* のシーケンス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* の値をキーとするシーケンスに追加されます。読み出された行が *fieldnames* のシーケンス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。その他の省略可能またはキーワード形式のパラメタはベースになっている `reader` のインスタンスに渡されます。

class DictWriter(*csvfile*, *fieldnames*[, *restval*=`""`[, *extrasaction*=`'raise'`[, *dialect*=`'excel'`[, **args*, ***kwds*]]]])

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ベースになっている `writer` のインスタンスに渡されます。

`DictReader` クラスとは違い、`DictWriter` の *fieldnames* パラメータは省略可能ではありません。Python の `dict` オブジェクトは整列されていないので、列が *csvfile* に書かれるべき順序を推定するための十分な情報はありません。

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 excel()

`excel` クラスは Excel で生成される CSV ファイルの通常のプロパティを定義します。

class excel_tab()

`excel` クラスは Excel で生成されるタブ分割ファイルの通常のプロパティを定義します。

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二つ提供しています:

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、*delimiter*、*quotechar* または *lineterminator* に含まれる任意の文字のような特別な文字を含む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。

reader に対しては、クオートされていない全てのフィールドを *float* 型に変換するよう指示します。

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。*escapechar* がセットされていない場合、エスケープが必要な文字に遭遇した writer は Error を送出します。

reader に対しては、クオート文字の特別扱いをしないように指示します。

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。

9.1.2 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 *quotechar* の前に置かれます。デフォルトでは True です。

出力においては、*doublequote* が False で *escapechar* がセットされていない場合、フールド内に *quotechar* が現れると Error が送出されます。

escapechar
writer が、*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、および、*doublequote* が False の場合に *quotechar* 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。読み込み時には *escapechar* はそれに引き続く文字の特別な意味を取り除きます。デフォルトでは `None` で、エスケープを行ないません。

lineterminator
`writer` が作り出す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。

注意: `reader` は `'\r'` または `'\n'` のどちらかを行末と認識するようにハードコードされており、*lineterminator* を無視します。この振る舞いは将来変更されるかもしれません。

quotechar
delimiter や *quotechar* といった特殊文字を含むか、改行文字を含むフィールド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
クオートがいつ writer によって生成されるか、また reader によって認識されるかを制御します。`QUOTE_*` 定数のいずれか (9.1.1 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。

9.1.3 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。

reader オブジェクトには以下の公開属性があります:

dialect
パーサで使われる表現形式の読み取り専用の記述です。

line_num
ソースイテレータから読んだ行数です。この数は返されるレコードの数とは、レコードが複数行に亘ることがあるので、一致しません。

9.1.4 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:

`Writer` オブジェクト (`writer()` で生成される `DictWriter` クラスのインスタンス）は、以下の公開メソッドを持っています。*row* には、`Writer` オブジェクトの場合には文字列か数値のシーケンスを指定し、`DictWriter` オブジェクトの場合はフィールド名をキーとして対応する文字列か数値を格納した辞書オブジェクトを指定します (数値は `str()` で変換されます)。 複素数を出力する場合、値をかっこで囲んで出力します。このため、CSV ファイルを読み込むアプリケーションで（そのアプリケーションが複素数をサポートしていたとしても）問題が発生する場合があります。

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 (上記 *row* のリスト) 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。

writer オブジェクトには以下の公開属性があります:

dialect
writer で使われる表現形式の読み取り専用の記述です。

9.1.5 使用例

最も簡単な CSV ファイル読み込みの例です:

```
import csv
reader = csv.reader(file("some.csv", "rb"))
for row in reader:
    print row
```

別の書式での読み込み:

```
import csv
reader = csv.reader(open("passwd", "rb"), delimiter=':', quoting=csv.QUOTE_NONE)
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", "wb"))
writer.writerows(someiterable)
```

新しい表現形式の登録:

```
import csv

csv.register_dialect('unixpwd', delimiter=':', quoting=csv.QUOTE_NONE)

reader = csv.reader(open("passwd", "rb"), 'unixpwd')
```

もう少し手の込んだ reader の使い方 — エラーを捉えてレポートします。

```
import csv, sys
filename = "some.csv"
reader = csv.reader(open(filename, "rb"))
try:
    for row in reader:
        print row
except csv.Error, e:
    sys.exit('file %s, line %d: %s' % (filename, reader.line_num, e))
```

このモジュールは文字列の解析は直接サポートしませんが、簡単にできます。

```
import csv
for row in csv.reader(['one,two,three']):
    print row
```

csv モジュールは直接は Unicode の読み書きをサポートしませんが、ASCII NUL 文字に関わる問題のために 8 ビットクリーンに書き込みます。ですから、NUL を使う UTF-16 のようなエンコーディングを避け

る限りエンコード・デコードを行なう関数やクラスを書くことができます。UTF-8 がお勧めです。

以下の unicode_csv_reader は Unicode の CSV データ (Unicode 文字列のリスト) を扱うための csv.reader をラップするジェネレータです。utf_8_encoder は一度に 1 文字列 (または行) ずつ Unicode 文字列を UTF-8 としてエンコードするジェネレータです。エンコードされた文字列は CSV reader により分解され、unicode_csv_reader が UTF-8 エンコードの分解された文字列をデコードして Unicode に戻します。

```
import csv

def unicode_csv_reader(unicode_csv_data, dialect=csv.excel, **kwargs):
    # csv.py doesn't do Unicode; encode temporarily as UTF-8:
    csv_reader = csv.reader(utf_8_encoder(unicode_csv_data),
                            dialect=dialect, **kwargs)
    for row in csv_reader:
        # decode UTF-8 back to Unicode, cell by cell:
        yield [unicode(cell, 'utf-8') for cell in row]

def utf_8_encoder(unicode_csv_data):
    for line in unicode_csv_data:
        yield line.encode('utf-8')
```

その他のエンコーディングには以下の UnicodeReader クラスと UnicodeWriter クラスが使えます。二つのクラスは *encoding* パラメータをコンストラクタで取り、本物の reader や writer に渡されるデータが UTF-8 でエンコードされていることを保証します。

```
import csv, codecs, cStringIO

class UTF8Recoder:
    """
    Iterator that reads an encoded stream and reencodes the input to UTF-8
    """
    def __init__(self, f, encoding):
        self.reader = codecs.getreader(encoding)(f)

    def __iter__(self):
        return self

    def next(self):
        return self.reader.next().encode("utf-8")

class UnicodeReader:
    """
    A CSV reader which will iterate over lines in the CSV file "f",
    which is encoded in the given encoding.
    """

    def __init__(self, f, dialect=csv.excel, encoding="utf-8", **kwds):
        f = UTF8Recoder(f, encoding)
        self.reader = csv.reader(f, dialect=dialect, **kwds)

    def next(self):
        row = self.reader.next()
        return [unicode(s, "utf-8") for s in row]

    def __iter__(self):
        return self

class UnicodeWriter:
    """
    A CSV writer which will write rows to CSV file "f",
    which is encoded in the given encoding.
    """

    def __init__(self, f, dialect=csv.excel, encoding="utf-8", **kwds):
        # Redirect output to a queue
        self.queue = cStringIO.StringIO()
        self.writer = csv.writer(self.queue, dialect=dialect, **kwds)
        self.stream = f
        self.encoder = codecs.getincrementalencoder(encoding)()

    def writerow(self, row):
        self.writer.writerow([s.encode("utf-8") for s in row])
        # Fetch UTF-8 output from the queue ...
        data = self.queue.getvalue()
        data = data.decode("utf-8")
        # ... and reencode it into the target encoding
        data = self.encoder.encode(data)
        # write to the target stream
        self.stream.write(data)
        # empty queue
        self.queue.truncate(0)

    def writerows(self, rows):
        for row in rows:
            self.writerow(row)
```

9.2 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 ';' ではじまる行は無視され，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。

置換で使われるすべてのオプション名は、ほかのオプション名への参照と同様に optionxform() メソッドを介して渡されます。たとえば、optionxform() のデフォルト実装 (これはオプション名を小文字に変換します) を使うと、値 'foo %(bar)s' および 'foo %(BAR)s' は同一になります。

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

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 (22.2 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.

9.2.1 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 True を返し、そうでなければ False 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をこころみ、うまく解析できたファイル名のリストを返します。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')])
```

2.4 で変更された仕様: うまく解析できたファイル名のリストを返す

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 を発生させます。RawConfigParser (あるいは *raw* パラメータをセットした ConfigParser) を文字列型でない値の **内部的な** 格納場所として使うことは可能ですが、すべての機能 (置換やファイルへの出力を含む) がサポートされるのは文字列を値として使った場合だけです。 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 True を、そうでない時は False 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。サブル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。

9.2.2 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。

9.2.3 SafeConfigParser オブジェクト

`SafeConfigParser` は `ConfigParser` と同様の拡張インターフェイスをもっていますが、以下のような機能が追加されています:

set(*section, option, value*)

もし与えられたセクションが存在している場合は、指定された値を与えられたオプションに設定します。そうでない場合は `NoSectionError` を発生させます。 *value* は文字列 (`str` または `unicode`) でなければならず、そうでない場合には `TypeError` が発生します。

2.4 で追加された仕様です。

9.3 `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
```

9.4 `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。

9.4.1 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: 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。

9.5 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して定義されています。

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 と対をなしていて、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当時に 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。

9.5.1 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とり、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_hyper() 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*)
一様の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。

9.5.2 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、引数をとりません。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 要素づ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、返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。

9.5.3 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
```

十章

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:

hashlib セキュアハッシュおよびメッセージダイジェストのアルゴリズム

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 Auth

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。

あなたがハードコアなサイファーパンクなら、 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。このパッケージは AES をはじめとする様々な暗号化アルゴリズムのモジュールを含みます。これらのモジュールは Python と一緒には配布されず、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。

10.1 hashlib — セキュアハッシュおよびメッセージダイジェスト

2.5 で追加された仕様です。

このモジュールは、セキュアハッシュやメッセージダイジェスト用のさまざまなアルゴリズムを実装したものです。FIPS のセキュアなハッシュアルゴリズムである SHA1、SHA224、SHA256、SHA384 および SHA512 (FIPS 180-2 で定義されているもの) だけでなく RSA の MD5 アルゴリズム (Internet RFC 1321 で定義されています) も実装しています。「セキュアなハッシュ」と「メッセージダイジェスト」はどちらも同じ意味です。古くからあるアルゴリズムは「メッセージダイジェスト」と呼ばれていますが、最近は「セキュアハッシュ」という用語が用いられています。

警告: 中には、ハッシュの衝突の脆弱性をかかえているアルゴリズムもあります。最後の FAQ をごらんください。

hash のそれぞれの型の名前をとったコンストラクタメソッドがひとつずつあります。返されるハッシュオブジェクトは、どれも同じシンプルなインターフェイスを持っています。たとえば `sha1()` を使用すると SHA1 ハッシュオブジェクトが作成されます。このオブジェクトの `update()` メソッドに、任意の文字列を渡すことができます。それまでに渡した文字列の *digest* を知りたければ、`digest()` メソッドあるいは `hexdigest()` メソッドを使用します。

このモジュールで常に使用できるハッシュアルゴリズムのコンストラクタは `md5()`、`sha1()`、`sha224()`、`sha256()`、`sha384()` および `sha512()` です。それ以外のアルゴリズムが使用できるかどうかは、Python が使用している OpenSSL ライブラリに依存します。

たとえば、`'Nobody inspects the spammish repetition'` という文字列のダイジェストを取得するには次のようにします。

```
>>> import hashlib
>>> m = hashlib.md5()
>>> 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'
```

もっと簡潔に書くと、このようになります。

```
>>> hashlib.sha224("Nobody inspects the spammish repetition").hexdigest()
'a4337bc45a8fc544c03f52dc550cd6e1e87021bc896588bd79e901e2'
```

汎用的なコンストラクタ `new()` も用意されています。このコンストラクタの最初のパラメータとして、使いたいアルゴリズムの名前を指定します。アルゴリズム名として指定できるのは、先ほど説明したアルゴリズムか OpenSSL ライブラリが提供するアルゴリズムとなります。しかし、アルゴリズム名のコンストラクタのほうが `new()` よりずっと高速なので、そちらを使うことをお勧めします。

`new()` に OpenSSL のアルゴリズムを指定する例です。

```
>>> h = hashlib.new('ripemd160')
>>> h.update("Nobody inspects the spammish repetition")
>>> h.hexdigest()
'cc4a5ce1b3df48aec5d22d1f16b894a0b894eccc'
```

コンストラクタが返すハッシュオブジェクトには、次のような定数属性が用意されています。

digest_size
: 作成されたダイジェストのバイト数。

ハッシュオブジェクトには次のようなメソッドがあります。

update(*arg*)
: ハッシュオブジェクトを文字列 *arg* で更新します。繰り返してコールするのは、すべての引数を連結して 1 回だけコールするのと同じ意味になります。つまり、`m.update(a); m.update(b)` と `m.update(a+b)` は同じ意味だということです。

digest()
: これまでに `update()` メソッドに渡した文字列のダイジェストを返します。これは `digest_size` バイトの文字列であり、非 ASCII 文字や null バイトを含むこともあります。

hexdigest()
: `digest()` と似ていますが、返される文字列は倍の長さとなり、16 進形式となります。これは、電子メールなどの非バイナリ環境で値を交換する場合に便利です。

copy()
: ハッシュオブジェクトのコピー ("クローン") を返します。これは、共通部分を持つ複数の文字列のダイジェストを効率的に計算するために使用します。

参考資料:

`hmac` モジュール (10.2 節):
: ハッシュを用いてメッセージ認証コードを生成するモジュールです。

`base64` モジュール (7.11 節):
: バイナリハッシュを非バイナリ環境用にエンコードするもうひとつの方法です。

http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

FIPS 180-2 のセキュアハッシュアルゴリズムについての説明。

http://www.cryptography.com/cnews/hash.html
Hash Collision FAQ。既知の問題を持つアルゴリズムとその使用上の注意点に関する情報があります。

10.2 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コンストラクタあるいはモジュールです。標準では hashlib.md5 コンストラクタになっています。**注意:** md5 ハッシュには既知の脆弱性がありますが、後方互換性を考慮してデフォルトのままにしています。使用するアプリケーションにあわせてよりよいものを選択してください。

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。これは digest_size バイトの文字列で、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、返される文字列は倍の長さとなり、16 進形式となります。これは、電子メールなどの非バイナリ環境で値を交換する場合に便利です。

copy()
hmac オブジェクトのコピー ("クローン") 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。

参考資料:

hashlib モジュール (10.1 節):
セキュアハッシュ関数を提供する python モジュールです。

10.3 md5 — MD5 メッセージダイジェストアルゴリズム

リリース 2.5 以降で撤廃された仕様です。 代わりにモジュール hashlib を使ってください。

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のようになります:

```
>>> 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 (10.4 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。

10.4 sha — SHA-1 メッセージダイジェストアルゴリズム

リリース 2.5 以降で撤廃された仕様です。 かわりにモジュール `hashlib` を使ってください。

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/fips180-2/fips180-2withchangenotice.pdf)
: 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 180-2 で定義されています。**セキュアハッシュスタンダード**, 2002 年 8 月出版。

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

十一章

ファイルとディレクトリへのアクセス

この章で説明されるモジュールはディスクのファイルやディレクトリを扱います。たとえば、ファイルの属性を読むためのモジュール、ファイルパスを移植可能な方式で操作する、テンポラリファイルを作成するためのモジュールです。この章の完全な一覧は:

os.path	共通のパス名操作。
fileinput	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 (その場で保存する機能
stat	`os.stat()`、`os.lstat()` および `os.fstat()` の返す内容を解釈するためのユーティリティ群。
statvfs	`os.statvfs()` の返す値を解釈するために使われる定数群。
filecmp	ファイル群を効率的に比較します。
tempfile	一時的なファイルやディレクトリを生成。
glob	UNIX シェル形式のパス名のパターン展開。
fnmatch	UNIX シェル形式のファイル名のパターンマッチ。
linecache	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。
shutil	コピーを含む高レベルなファイル操作。
dircache	キャッシュメカニズムを備えたディレクトリ一覧生成。

Python 組み込みのファイルオブジェクトについては 3.9 をごらんください。

参考資料:

`os` モジュール (14.1 節):

オペレーティングシステムのインタフェース、組み込みのファイルオブジェクトより低レベルでのファイル操作を含む。

11.1 `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* の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。壊れたシンボリックリンクについては `False` を返します。いくつかのプラットフォームでは、たとえ *path* が物理的に存在していたとしても、リクエストされたファイルに対する `os.stat()` の実行が許可されなければこの関数が `False` を返すことがあります。

lexists (*path*)
: *path* が存在するパスなら `True` を返す。壊れたシンボリックリンクについては `True` を返します。`os.lstat()` がない環境では `exists()` と同じです。 2.4 で追加された仕様です。

expanduser (*path*)
: UNIX では、与えられた引数の先頭のパス要素 '~' 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。先頭の '~' は、環境変数 HOME が設定されているならその値に置き換えられます。そうでなければ、現在のユーザのホームディレクトリを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探して置き換えます。先頭の '*~user*' については、直接パスワードディレクトリから探します。

  Windows では '~' だけがサポートされ、環境変数 HOME または HOMEDRIVE と HOMEPATH の組み合わせで置き換えられます。

  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。

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。

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*[, ...]])

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 (Windows ではドライブ名があればそれも含めて) 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、`A//B`、`A/./B`、`A/foo/../B` が全て `A/B` になるようにします。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。パスがシンボリックリンクを含んでいるかによって意味が変わることに注意してください。

realpath(*path*)

パスの中のシンボリックリンク (もしそれが当該オペレーティングシステムでサポートされていれば) を取り除いて、標準化したパスを返します。 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*)

ファイルディスクリプタ *fp1* と *fp2* が同じファイルを指していたら、`True` を返します。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。

splitunc(*path*)

パス名 *path* をペア（*unc*, *rest*）に分割します。ここで *unc* は (r'\\host\mount' のような)UNC マウントポイント、そして *rest* は (r'\path\file.ext' のような) パスの残りの部分です。ドライブ名を含むパスでは常に *unc* が空文字列になります。利用可能: Windows。

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。

11.2 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。

全てのファイルはデフォルトでテキストモードでオープンされます。しかし、input() や FileInput() をコールする際に *mode* パラメータを指定すれば、これをオーバーライドすることができます。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。

ファイルのオープン方法を制御するためのオープン時フックは、input() あるいは FileInput() の *openhook* パラメータで設定します。このフックは、ふたつの引数 *filename* と *mode* をとる関数でなければなりません。そしてその関数の返り値はオープンしたファイルオブジェクトとなります。このモジュールには、便利なフックが既に用意されています。

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*\[, *mode*\[, *openhook*\]\]\]\]\])

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。

2.5 で変更された仕様: パラメータ *mode* および *openhook* が追加されました

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。

fileno()

現在のファイルの "ファイルデスクリプタ" を整数値で返します。ファイルがオープンされていない場合 (最初の行の前、ファイルとファイルの間) は -1 を返します。 2.5 で追加された仕様です。

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しませんので、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しません。

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ることもできます。

class **FileInput**([*files*[, *inplace*[, *backup*[, *mode*[, *openhook*]]]]])

FileInput クラスはモジュールの関数に対応するメソッド filename()、fileno()、lineno()、fileline()、isfirstline()、isstdin()、nextfile() および 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。

mode を使用すると、open() に渡すファイルモードを指定することができます。これは 'r'、'rU'、'U' および 'rb' のうちのいずれかとなります。

openhook を指定する場合は、ふたつの引数 *filename* と *mode* をとる関数でなければなりません。この関数の返り値は、オープンしたファイルオブジェクトとなります。*inplace* と *openhook* を同時に使うことはできません。

2.5 で変更された仕様: パラメータ *mode* および *openhook* が追加されました

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、警告無しに置き替えられます)。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。

このモジュールには、次のふたつのオープン時フックが用意されています。

hook_compressed(*filename, mode*)

gzip や bzip2 で圧縮された (拡張子が'.gz' や'.bz2' の) ファイルを、gzip モジュールや bz2 モジュールを使って透過的にオープンします。ファイルの拡張子が'.gz' や'.bz2' でない場合は、通常通りファイルをオープンします (つまり、open() をコールする際に伸長を行いません)。

使用例: 'fi = fileinput.FileInput(openhook=fileinput.hook_compressed)'

2.5 で追加された仕様です。

hook_encoded(*encoding*)

各ファイルを codecs.open() でオープンするフックを返します。指定した *encoding* でファイルを読み込みます。

使用例: 'fi = fileinput.FileInput(openhook=fileinput.hook_encoded("iso-8859-1"))'

注意: このフックでは、指定した *encoding* によっては `FileInput` が Unicode 文字列を返す可能性があります。 2.5 で追加された仕様です。

11.3 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が、`stat()` システムコールを何度も呼び出してオーバ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
オペレーティングシステムから返される “ctime”。ある OS(UNIX など) では最後にメタデータが更新された時間となり、別の OS(Windows など) では作成時間となります (詳細については各プラットフォームのドキュメント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る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)
```

11.4 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。

11.5 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*])
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。

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ので注意してください。

cmpfiles (*dir1, dir2, common*[, *shallow*])
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* 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
```

11.5.1 `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。

11.6 `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、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と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*]]]])

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。

mkdtemp([*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:

- 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()

一時ファイルを生成する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。

11.7 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.expantion() 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。結果には (シェルと同じく) 壊れたシンボリックリンクも含まれます。

iglob (*pathname*)
全ての値を一度に格納することなく `glob()` と同じ値を生成するすイテレータを返します。2.5 で追加された仕様です。

iglob (*pathname*)
実際には一度に全てを格納せずに、`glob()` と同じ値を順に生成するイテレータを返します。2.5 で追加された仕様です。

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 (11.8 節):
シェル形式の (パスではない) ファイル名展開

11.8 `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。オペレーティングシステムが大文字、小文字を区別しない場合、比較を行う前に、両方のパラメタを全て大文字、または全て小文字に揃えます。オペレーティングシステムが標準でどうなっているかに関係なく、大小文字を区別して比較したい場合には、`fnmatchcase()` を代わりに使ってください。

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 (11.7 節):

UNIX シェル形式のパス展開。

11.9 `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*[, *module_globals*])

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。

zipfile やその他のファイルシステムでない import 元に対応するためまず *modules_globals* の PEP 302 `__loader__` をチェックし、そのあと `sys.path` を探索します。

2.5 で追加された仕様: パラメータ *module_globals* の追加

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([*filename*])

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れなくて、更新が必要なときにこの関数を使ってください。もし *filename* がなければ、全てのキャッシュエントリをチェックします。

サンプル:

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

11.10 `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。コピー先は書き込み可能である必要があります。そうでなければ IOError を発生します。もし *dst* が存在したら、置き換えられます。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* にはパス名を文字列で与えられます。

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 *dist* 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ーに既存のものは使えません。存在しない親ディレクトリも含めて作成されます。パーミッションと時刻は `copystat()` 関数でコピーされます。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 `Error` を起こします。

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 `Error` を起こす。

2.5 で変更された仕様: *dst* を作成する際に中間のディレクトリ作成が必要な場合、エラーを起こすのではなく作成する。ディレクトリのパーミッションと時刻を `copystat()` を利用してコピーする。

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.listdir()`、`os.remove()` または `os.rmdir()` 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純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。

11.10.1 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))
```

11.11 dircache — キャッシュされたディレクトリ一覧の生成

durcache 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:

reset()

ディレクトリキャッシュをリセットします。

listdir(*path*)

os.listdir() によって得た *path* のディレクトリ一覧を返します。*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']
```

十二章

データ圧縮とアーカイブ

この章で説明されるモジュールは zlib, gzip, bzip2 アルゴリズムによるデータの圧縮と、 ZIP, tar フォーマットのアーカイブ作成をサポートする。

zlib	**gzip** 互換の圧縮／解凍ルーチンへの低レベルインタフェース
gzip	ファイルオブジェクトを用いた **gzip** 圧縮および解凍のためのインタフェース
bz2	**bzip2** 互換の圧縮／解凍ルーチンへのインタフェース
zipfile	ZIP-フォーマットのアーカイブファイルを読み書きする
tarfile	tar-形式のアーカイブファイルを読み書きします。

12.1 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.zlib.net です。Python モジュールと zlib ライブラリの 1.1.3 より前のバージョンには互換性のない部分があることが知られています。1.1.3 にはセキュリティホールが存在しますので、1.1.4 以降のバージョンを利用することをお勧めします。

zlib の関数にはたくさんのオプションがあり、しばしば特定の順番で使う必要があります。このドキュメントでは順番のことについて全てを説明し尽くそうとはしていません。信頼できる情報が必要ならば http://www.zlib.net/manual.html にある zlib のマニュアルを参照するようにしてください。

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えいるので、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。

copy ()

圧縮オブジェクトのコピーを返します。これを使うと先頭部分が共通している複数のデータを効率的に圧縮することができます。 2.5 で追加された仕様です。

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分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る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` ([*length*])
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。

オプション引数 *length* は出力バッファの初期サイズを決めます。

`copy`()
解凍オブジェクトのコピーを返します。これを使うとデータストリームの途中にある解凍オブジェクトの状態を保存でき、未来のある時点で行なわれるストリームのランダムなシークをスピードアップするのに利用できます。 2.5 で追加された仕様です。

参考資料:

`gzip` モジュール (12.2 節):
Reading and writing **gzip**-format files.

`http://www.zlib.net`
zlib ライブラリホームページ

`http://www.zlib.net/manual.html`
zlib ライブラリの多くの関数の意味と使い方を解説したマニュアル

12.2 `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 (12.1 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。

12.3 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イテレーション機能を提供します。
- BZ2Compressor および BZ2Decompressor クラスでは逐次的圧縮（解凍）をサポートしています。
- compress() および decompress() では一括圧縮（解凍）を関数サポートしています。
- 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。

12.3.1 ファイルの圧縮（解凍）

BZ2File クラスは圧縮ファイルの操作機能を提供しています。

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作成し、そうでない場合ファイルを切り詰ます。*buffering* パラメタを与えた場合、0 はバッファリングなしを表し、それよりも大きい値はバッファサイズになります。デフォルトでは 0 です。圧縮レベル *compresslevel* を与える場合、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が利用できるのは読み込みだけです。BZ2File が生成す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イテレーション操作をサポートしています。

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。閉じたファイルはそれ以後入出力操作の対象にできません。close() 自体の呼び出しはエラーを引き起こすことなく何度も実行できます。

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* 引数を負の値にした場合や省略した場合、EOF にたどり着くまで読み出します。

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を返すこともあります)。 EOF の時には空文字列を返します。

readlines([*size*])

ファイルから読み取った各行の文字列からなるリストを返します。オプション引数 *size* を与えた場合、文字列リストの合計バイト長の大まかな上限の指定になります。

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。代りに for line in file を使ってください。

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

ファイルの読み書き位置を移動します。 引数 *offset* はバイト数で指定したオフセット値です。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、offset >= 0 になるはず)

です。他にとり得る値は 1 (現在のファイル位置からの相対位置で、正負どちらの値もとり得る)、および 2 (ファイルの終末端からの相対位置で、通常は負の値になるが、多くのプラットフォームではファイルの終末端を越えて seek できる) です。

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）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シーケンス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改行文字を追加することはありません。シーケンスはイテレーション処理で文字列を取り出せる任意のオブジェクトにできます。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じ操作です。

12.3.2 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 クラスを用いて行います。

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。圧縮データのチャンクを生成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を返します。

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。このメソッドの呼び出し以降は同じ圧縮オブジェクトを使ってはなりません。

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、解凍データのチャンクを生成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 を送出します。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、解凍処理はこのデータを無視し、オブジェクトの unused_data 属性に収めます。

12.3.3 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なら、`BZ2Decompressor` を代りに使ってください。

12.4 `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、追記、書庫内のファイル一覧の作成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。ZIP64 拡張を利用する ZIP ファイル (サイズが 4GB を超えるような ZIP ファイル) は扱えます。

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

exception LargeZipFile

ZIP ファイルが ZIP64 の機能を必要とするとき、その機能が有効にされていないと送出されるエラー

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**” (12.4.1 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 12.4.3, “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/business_and_developers/developer/appnote/)

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。

12.4.1 ZipFile オブジェクト

class ZipFile (*file*[, *mode*[, *compression*[, *allowZip64*]]])
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。*allowZip64* が True ならば 2GB より大きな ZIP ファイルの作成時に ZIP64 拡張を使用します。これが False ならば、zipfile モジュールは ZIP64 拡張が必要になる場面で例外を送出します。ZIP64 拡張はデフォルトでは無効にされていますが、これは UNIX の **zip**　および **unzip** (InfoZIP ユーティリティ) コマンドがこの拡張をサポートしていないからです。

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内容をバイト列にして返します。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 チェックサムとヘッダが正常か調べます。最初に見つかった不正な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*]])
filename に指定したファイル名を持つファイルを、アーカイブ名を *arcname* (デフォルトでは *filename*

と同じですがドライブレターと先頭にあるパスセパレータは取り除かれます) にしてアーカイブに収録します。*compress_type* を指定した場合、コンストラクタを使って新たなアーカイブエントリを生成した際に使った *compression* パラメタを上書きします。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。

注意: ZIP ファイル中のファイル名に関する公式なエンコーディング方式はありません。もしユニコードのファイル名が付けられているならば、それを write() に渡す前に望ましいエンコーディングでバイト列に変換しておいてください。WinZip は全てのファイル名を DOS Latin としても知られる CP437 で解釈します。

注意: アーカイブ名はアーカイブルートに対する相対的なものでなければなりません。言い換えると、アーカイブ名はパスセパレータで始まってはいけません。

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。

12.4.2 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
```

12.4.3 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。

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。

12.5 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します。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 12.5.1) を見て下さい。

mode は 'filemode[:compression]' の形式をとる文字列でなければなりません．デフォルトの値は 'r' です．以下に *mode* のとりうる組み合わせ全てを示します．

mode	動作
'r' または '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 番目の形式: ' ファイルモード|[圧縮]' があります。この形式を使うと，open が返すのはデータをブロックからなるストリームとして扱う TarFile オブジェクトになります．この場合，ファイルに対してランダムな seek を行えなくなります．*fileobj* を指定する

場合，`read()` および `write()` メソッドを持つ任意のオブジェクトにできます．*bufsize* にはブロックサイズを指定します．デフォルトは `20 * 512` バイトです。`sys.stdin`，ソケットファイルオブジェクト，テープデバイスと組み合わせる場合にはこの形式を使ってください．ただし，このような `TarFile` オブジェクトにはランダムアクセスを行えないという制限があります．**例** (セクション 12.5.3) を参照してください。現在可能なモードは：

モード	動作
`'r|*'`	tar ブロックの **ストリーム** を透過な読み込みにオープンします。
`'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ず，代わりに `open()` を使ってください．`TarFile` オブジェクト (12.5.1 節) を参照してください．

`is_tarfile`(*name*)

もし *name* が tar アーカイブファイルであり，`tarfile` モジュールで読み出せる場合に `True` を返します．

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。詳細は `zipfile` のドキュメントを参照してください．*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 (12.4 節):

`zipfile` 標準モジュールのドキュメント。

GNU tar **マニュアル**, **基本** *Tar* **形式**

(http://www.gnu.org/software/tar/manual/html_node/tar_134.html#SEC134)

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。

12.5.1 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 12.5.2) 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* のデフォールトは `'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 12.5 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"` に似た出力を生成します.

`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。

`extractall`(*[path[, members]]*)

全てのメンバーをアーカイブから現在の作業ディレクトリーまたは *path* に抽出します。オプションの *members* が与えられるときには、`getmembers()` で返されるリストの一部でなければなりません。所有者、変更時刻、許可のようなディレクトリー情報は全てのメンバーが抽出された後にセットされます。これは二つの問題を回避するためです。一つはディレクトリーの変更時刻はその中にファイルが作成されるたびにリセットされるということ。もう一つは、ディレクトリーに書き込み許可がなければその中のファイル抽出は失敗してしまうということです。 2.5 で追加された仕様です。

`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。

注意: extract() メソッドでは tar アーカイブにランダムアクセスすることが許されるので、これを使う場合には使用者自身が気をつけなければならない問題があります。上の extractall() の説明を参照してください。

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*]])

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。デフォー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。デフォルトは True です.

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。gettarinfo() 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。

gettarinfo([*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

この値が真なら、POSIX 1003.1-1990 準拠のアーカイブを作成します。GNU 拡張機能はは POSIX 標準の一部ではないため使いません．POSIX 準拠のアーカイブでは，ファイル名の長さは最大 256 ，リンク名の最大長は 100 文字に制限されており，ファイルの最大長は 8 ギガバイト以下です．ファイルがこれらの制限を超えた場合，ValueError を送出します．この値が偽の場合，GNU tar 互換のアーカイブを作成します．POSIX 仕様には準拠しませんが，上記の制約を受けずにファイルを保存できます． 2.4 で変更された仕様: *posix* のデフォルト値が False になりました

dereference

この値が偽の場合，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。真の場合，ターゲットファイルの内容をアーカイブに追加します。この値はリンクをサポートしないシステムには影響しません。

ignore_zeros

この値が偽の場合，空のブロックをアーカイブの終わりとして処理します。真の場合，空 (で無効な)

ブロックを飛ばして、できるだけ多くのメンバを取得しようとします。これはアーカイブを連結している場合やアーカイブが損傷している場合に役に立ちます。

debug=0

0(デバッグメッセージなし、デフォルト) から 3(すべてのデバッグメッセージあり) までの値に設定します．メッセージは sys.stderr に出力されます．

errorlevel=0

この値が 0 (デフォルトの値です) の場合，extract() 実行時の全てのエラーを無視します．ただし，デバッグが有効になっている場合には，デバッグ出力にエラーメッセージとして出力します．値を 1 にした場合，すべての**致命的な** エラーに対して OSError または IOError 例外を送出します．値を 2 にした場合、**致命的でない**エラーもまた，全て TarError 例外として送出します．

12.5.2 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 ([*posix*])

TarInfo オブジェクトから文字列バッファを作成します。*posix* 引数については TarFile の posix 属性の項を参照してください。この引数はデフォルトでは False です。

2.5 で追加された仕様: *posix* 引数

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

ファイルタイプです．*type* は普通、以下の定数: REGTYPE, AREGTYPE, LNKTYPE, SYMTYPE, DIRTYPE, FIFOTYPE, CONTTYPE, CHRTYPE, BLKTYPE, GNUTYPE_SPARSE のいずれかです．TarInfo オブジェクトのタイプをもっと便利に決定するには、下記の 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

ファイルメンバを保存した元のユーザのユーザ ID です.

gid
ファイルメンバを保存した元のユーザのグループ ID です.

uname
ファイルメンバを保存した元のユーザのユーザ名です.

gname
ファイルメンバを保存した元のユーザのグループ名です.

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()
Tarinfo オブジェクトが普通のファイルの場合に、True を返します。

isreg()
isfile() と同じです。

isdir()
ディレクトリの場合に True を返します。

issym()
シンボリックリンクの場合に True を返します。

islnk()
ハードリンクの場合に True を返します。

ischr()
キャラクタデバイスの場合に True を返します。

isblk()
ブロックデバイスの場合に True を返します。

isfifo()
FIFO の場合に True を返します。

isdev()
キャラクタデバイス、ブロックデバイスあるいは FIFO のいずれかの場合に True を返します。

12.5.3 例

tar アーカイブから現在のディレクトリーに全て抽出する方法：

```
import tarfile
tar = tarfile.open("sample.tar.gz")
tar.extractall()
tar.close()
```

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()
```

十三章

データの永続化

この章で解説されるモジュール群は Python データをディスクに永続的な形式で保存します。モジュール pickle と モジュール marshal は多くの Python データ型をバイト列に変換し、バイト列から再生成します。様々な DBM 関連モジュールはハッシュを基にした、文字列から他の文字列へのマップを保存するファイルフォーマット群をサポートします。モジュール bsddb はディスクベースの文字列から文字列へのマッピングを、ハッシュ、B-Tree, レコードを基にしたフォーマットで提供します。

この章で説明されるモジュールは:

pickle	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。
cPickle	pickle の高速バージョンですが、サブクラスはできません。
copy_reg	pickle サポート関数を登録する。
shelve	Python オブジェクトの永続化。
marshal	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。
anydbm	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。
whichdb	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
dbm	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。
gdbm	GNU による dbm の再実装。
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。
bsddb	Berkeley DB ライブラリへのインタフェース
dumbdbm	単純な DBM インタフェースに対する可搬性のある実装。
sqlite3	A DB-API 2.0 implementation using SQLite 3.x.

13.1 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)” [1] 、“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。

13.1.1 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

[1] marshal モジュールと間違えないように注意してください

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 化しようとすると 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。

- 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 (persisitence) 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。

13.1.2 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が持つ制限 (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 piclke 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: *protocol* パラメータが導入されました。

protocol version >= 1 を指定することで、少しだけ効率の高いバイナリ形式を選ぶことができます。

13.1.3 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。

注意: protocols >= 1 で作られた pickle ファイルは、常にバイナリモードでオープンするようにしてください。古い ASCII ベースの pickle プロトコル 0 では、矛盾しない限りにおいてテキストモードとバイナリモードのいずれも利用することができます。

プロトコル 0 で書かれたバイナリの pickle ファイルは、行ターミネータとして単独の改行 (LF) を含んでいて、ですのでこの形式をサポートしない、 Notepad や他のエディタで見たときに「おかしく」見えるかもしれません。

この pickle 化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(*obj, file*[, *protocol*])
: すでに開かれているファイルオブジェクト *file* に、*obj* 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*).dump(*obj*) 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: *protocol* パラメータが導入されました。

  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 (*obj*[, *protocol*])

obj 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: *protocol* パラメータが追加されました。

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[2]として、`Pickler` および `Unpickler` を提供しています:

class Pickler (*file*[, *protocol*])

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: *protocol* パラメータが導入されました。

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 (*obj*)

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 *obj* 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 *protocol* 引数の値に応じて、バイナリおよび ASCII 形式が使われます。

[2] `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 13.1.6 を参照してください。

clear_memo()

picller 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[3]。。

`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 13.1.5 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。

13.1.4 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

[3] **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) 最小限の変更を整列化すること、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。

- 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、集合および辞書
- 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 13.1.5 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。極端に再帰的なデータ構造を pickle 化しようとした場合には再帰の深さ制限を越えてしまうかもしれず、この場合には `RuntimeError` が送出されます。この制限は、`sys.setrecursionlimit()` で慎重に上げていくことは可能です。

(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と併せ、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[4] 。

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ければなら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でも、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よいかもしれません。

13.1.5 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 13.1.6 を参照してください。

[4] 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

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、旧スタイルクラスではメソッド __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 (例えば __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。

新スタイルクラスでは、プロトコル 2 で呼び出される__getnewargs__() を定義する事ができます。インスタンス生成時に内部的な不変条件が成立する必要があったり、(タプルや文字列のように) 型の__new__() メソッドに指定する引数によってメモリの割り当てを変更する必要がある場合には__getnewargs__() を定義してください。新スタイルクラス C のインスタンスは、次のように生成されます。

```
obj = C.__new__(C, *args)
```

ここで *args* は元のオブジェクトの__getnewargs__() メソッドを呼び出した時の戻り値となります。__getnewargs__() を定義していない場合、*args* は空のタプルとなります。

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[5]。__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 __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。[6]

> **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ます。メソッドはこの呼び出しに対して文字列またはタプルのどちらかを返さねばなりません。

文字列を返す場合、その文字列は通常通りに pickle 化されるグローバル変数の名前を指しています。__reduce__ の返す文字列は、モジュールにからみてオブジェクトのローカルな名前でなければなりません; pickle モジュールはモジュールの名前空間を検索して、オブジェクトの属するモジュールを決定します。

タプルを返す場合、タプルの要素数は 2 から 5 でなければなりません。オプションの要素は省略したり None を指定したりできます。各要素の意味づけは以下の通りです:

- 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

[5] 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 t 用いられます
[6] 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。

ト自体はその名前が pickle 化されます。

- オブジェクトの初期バージョンを生成するために呼び出される呼び出し可能オブジェクトです。この呼び出し可能オブジェクトへの引数はタプルの次の要素で与えられます。それ以降の要素では pickle 化されたデータを完全に再構築するために使われる付加的な状態情報が与えられます。

  逆 pickle 化の環境下では、このオブジェクトはクラスか、“安全なコンストラクタ (safe constructor, 下記参照)” として登録されていたり属性 `__safe_for_unpickling__` の値が真であるような呼び出し可能オブジェクトでなければなりません。そうでない場合、逆 pickle 化を行う環境で `UnpicklingError` が送出されます。通常通り、 callable は名前だけで pickle 化されるので注意してください。

- 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 2.5 で変更された仕様: 以前は、この引数には `None` もあり得ました。

- オプションとして、オブジェクトの状態。13.1.5 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。

- オプションとして、リスト中の連続する要素を返すイテレータ (シーケンスではありません)。このリストの要素は pickle 化され、`obj.append(`*item*`)` または `obj.extend(`*list_of_items*`)` のいずれかを使って追加されます。主にリストのサブクラスで用いられていますが、他のクラスでも、適切なシグネチャの `append()` や `extend()` を備えている限り利用できます。(`append()` と `extend()` のいずれを使うかは、どのバージョンの pickle プロトコルを使っているか、そして追加する要素の数で決まります。従って両方のメソッドをサポートしていなければなりません。)

-

- オプションとして、辞書中の連続する要素を返すイテレータ (シーケンスではありません)。このリストの要素は (*key*, *value*) という形式でなければなりません。要素は pickle 化され、`obj[`*key*`] =` *value* を使ってオブジェクトに格納されます。主に辞書のサブクラスで用いられていますが、他のクラスでも、`__setitem__` を備えている限り利用できます。

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。

`__reduce__` を実装する場合、プロトコルのバージョンを知っておくと便利なことがあります。これは `__reduce__` の代わりに `__reduce_ex__` を使って実現できます。`__reduce_ex__` が定義されている場合、 `__reduce__` よりも優先して呼び出されます (以前のバージョンとの互換性のために `__reduce__` を残しておいてもかまいません)。`__reduce_ex__` はプロトコルのバージョンを表す整数の引数を一つ伴って呼び出されます。

`object` クラスでは `__reduce__` と `__reduce_ex__` の両方を定義しています。とはいえ、サブクラスで `__reduce__` をオーバライドしており、`__reduce_ex__` をオーバライドしていない場合には、`__reduce_ex__` の実装がそれを検出して `__reduce__` を呼び出すようになっています。

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[7]。

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とり、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 化するだけです。永続化 id 文字列が返された場合、 piclkler 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:

[7] 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、persistend_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[8]。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。

13.1.6 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

[8]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[9]。

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ず、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す。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ません。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 属性を関数か 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。

13.1.7 例

いちばん単純には、dump() と load() を使用してください。自己参照リストが正しく pickle 化およびリストアされることに注目してください。

```
import pickle

data1 = {'a': [1, 2.0, 3, 4+6j],
         'b': ('string', u'Unicode string'),
         'c': None}

selfref_list = [1, 2, 3]
selfref_list.append(selfref_list)

output = open('data.pkl', 'wb')

# Pickle dictionary using protocol 0.
pickle.dump(data1, output)

# Pickle the list using the highest protocol available.
pickle.dump(selfref_list, output, -1)

output.close()
```

以下の例は pickle 化された結果のデータを読み込みます。pickle を含むデータを読み込む場合、ファイルはバイナリモードでオープンしなければいけません。これは ASCII 形式とバイナリ形式のどちらが使わ

[9] 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。

れているかは分からないからです。

```
import pprint, pickle

pkl_file = open('data.pkl', 'rb')

data1 = pickle.load(pkl_file)
pprint.pprint(data1)

data2 = pickle.load(pkl_file)
pprint.pprint(data2)

pkl_file.close()
```

より大きな例で、クラスを pickle 化する挙動を変更するやり方を示します。`TextReader` クラスはテキストファイルを開き、`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。

```
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になるでしょう:

```
>>> 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 (13.3 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 (13.4 節):
: オブジェクトのインデクス付きデータベース; pickle を使います。

copy モジュール (5.17 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 (13.5 節):
: 高いパフォーマンスを持つ組み込み型整列化機構。

13.2 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[10]。

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。

[10] 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。

13.3 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よ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。

13.4 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*]]])

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、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、値を整列化する際にはバージョン 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。

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てしまいます。

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。

もう一つ追加でサポートされるメソッドがあります。

sync()

シェルフが *writeback* を *True* にセットして開かれている場合に、キャッシュ中の全てのエントリを書き戻します。また容易にできるならば、キャッシュを空にしてディスク上の永続的な辞書を同期します。このメソッドはシェルフを `close()` によって閉じるとき自動的に呼び出されます。

13.4.1 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) はかなり小さくなければならず、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*]])

`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、値を整列化する際にはバージョン 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。

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*]])

`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なくてはなりません。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*、および *writeback* 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*]]])

`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*、および *writeback* 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。

13.4.2 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
klist = 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 (13.6 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 (13.11 節):
: BSD db データベースインタフェース。

dbhash モジュール (13.10 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 (13.8 節):
: 標準の UNIX データベースインタフェース。

dumbdbm モジュール (13.12 節):
: dbm インタフェースの移植性のある実装。

gdbm モジュール (13.9 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 (13.1 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 (13.2 節):
: pickle の高速版。

13.5 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 特有のものですが、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 (稀にしかないことですが) Python のバージョン間で変更される可能性があるからです。[11]

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:** `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なりません (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*[, *version*])

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。

2.4 で追加された仕様: `dump` で、データフォーマットを表す *version* 引数を利用するべきです。(下を参照)

load (*file*)

[11] 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。

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*[, *version*])

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。

2.4で追加された仕様: dumpで、データフォーマットを表す*version*引数を利用するべきです。(下を参照)

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。

これに加えて、以下の定数が定義されています:

version

モジュールが利用するバージョンを表します。バージョン0は歴史的なフォーマットです。バージョン1(Python 2.4で追加されました)は文字列の再利用をします。バージョン2 (Python 2.5で追加されました)は浮動小数点数にバイナリフォーマットを使用します。現在のバージョンは2です。 2.4で追加された仕様です。

13.6 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。

以下の例ではホスト名と対応するタイトルがいくつか登録し、データベースの内容を表示します:

```
import anydbm

# データベースを開く、必要なら作成する
db = anydbm.open('cache', 'c')

# いくつかの値を設定する
db['www.python.org'] = 'Python Website'
db['www.cnn.com'] = 'Cable News Network'

# 内容についてループ。
# .keys(), .values() のような他の辞書メソッドもつかえます。
for k, v in db.iteritems():
    print k, '\t', v

# 文字列でないキーまたは値は例外を
# おこします（ほとんどのばあい TypeError です)。
db['www.yahoo.com'] = 4

# 終了したら close します。
db.close()
```

参考資料:

dbhash モジュール (13.10 節):
: BSD db データベースインタフェース。

dbm モジュール (13.8 節):
: 標準の UNIX データベースインタフェース。

dumbdbm モジュール (13.12 節):
: dbm インタフェースの移植性のある実装。

gdbm モジュール (13.9 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 (13.4 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 (13.7 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。

13.7 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。

13.8 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 特有のエラーが起っ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
`'r'`	存在するデータベースを読取り専用で開きます。(デフォルト)
`'w'`	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。
`'c'`	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
`'n'`	Always create a new, empty database, open for reading and writing
`'n'`	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 (13.6 節):

`dbm` スタイルの一般的なインタフェース

`gdbm` モジュール (13.9 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 (13.7 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

13.9 `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いに書き込まれます。
- '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 (13.6 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 (13.7 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。

13.10 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 (13.6 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 (13.11 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 (13.7 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。

13.10.1 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。

13.11 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。

bsddb モジュールは、バージョン 3.3 から 4.4 までの間の Berkeley DB ライブラリを必要とします。

参考資料:

http://pybsddb.sourceforge.net/
Berkeley DB インターフェース bsddb.db 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、Berkeley DB ライブラリを開発しています。

より新しい DB である DBEnv や DBSequence オブジェクトのインターフェースも bsddb.db モジュールで使用できます。これは、上の URL で説明されている Sleepycat Berkeley DB C API によりマッチしています。bsddb.db API が提供する追加機能には、チューニングやトランザクション、ログ出力、マルチプロセス環境でのデータベースへの同時アクセスなどがあります。

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。Python 2.5 以降、このインターフェースはマルチスレッドに対応しています。マルチスレッドを使用する場合は bsddb.db API を推奨します。こちらのほうがスレッドをよりうまく制御できるからです。

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), 'w' (読み書き可能)、'c' (読み書き可能 - 必要ならファイルを生成 … これがデフォルトです) または 'n' (読み書き可能 - 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*[, *cachesize*[, *maxkeypage*[, *minkeypage*[, *pgsize*[, *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)、'w' (読み書き可能)、'c' (読み書き可能 - 必要ならファイルを生成 … これがデフォルトです)、または 'n' (読み書き可能 - 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*[, *cachesize*[, *pgsize*[, *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), 'w' (読み書き可能)、'c' (読み書き可能 - 必要ならファイルを生成 … これがデフォルトです)、または 'n' (読み書き可能 - 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。

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。

参考資料:

dbhash モジュール (13.10 節):
bsddb への DBM 形式のインタフェース

13.11.1 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: 辞書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 oepn 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せん。データベースが空の場合、このメソッドは bsddb.error を発生させます。

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せん。

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せん。(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せん。(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。データベースが空の場合、このメソッドは bsddb.error を発生させます。

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
```

13.12 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、dubmdbm 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 (13.6 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 (13.8 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 (13.9 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 (13.4 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 (13.7 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。

13.12.1 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。

13.13 sqlite3 — SQLite データベースに対する DB-API 2.0 インタフェース

2.5 で追加された仕様です。

SQLite は、別にサーバプロセスは必要とせずデータベースのアクセスに SQL 問い合わせ言語の非標準的な一種を使える軽量なディスク上のデータベースを提供する C ライブラリです。ある種のアプリケーションは内部データ保存に SQLite を使えます。また、SQLite を使ってアプリケーションのプロトタイプを作りその後そのコードを PostgreSQL や Oracle のような大規模データベースに移植するということも可能です。

pysqlite は Gerhard Häring によって書かれ、PEP 249 に記述された DB-API 2.0 仕様に準拠した SQL インタフェースを提供するものです。

このモジュールを使うには、最初にデータベースを表す Connection オブジェクトを作ります。ここではデータはファイル '/tmp/example' に格納されているものとします。

```
conn = sqlite3.connect('/tmp/example')
```

特別な名前である ':memory:' を使うと RAM 上にデータベースを作ることもできます。

Connection があれば、 Cursor オブジェクトを作りその execute() メソッドを呼んで SQL コマンドを実行することができます。

```
c = conn.cursor()

# Create table
c.execute('''create table stocks
(date text, trans text, symbol text,
 qty real, price real)''')

# Insert a row of data
c.execute("""insert into stocks
          values ('2006-01-05','BUY','RHAT',100,35.14)""")
```

たいてい、SQL 操作は Python 変数の値を使う必要があります。この時、クエリーを Python の文字列操作を使って構築することは、安全とは言えないので、すべきではありません。そのようなことをするとプログラムが SQL インジェクション攻撃に対し脆弱になりかねません。

代わりに、DB-API のパラメータ割り当てを使います。'?' を変数の値を使いたいところに埋めておきます。その上で、値のタプルをカーソルの execute() メソッドの第 2 引数として引き渡します。(他のデータベースモジュールでは変数の場所を示すのに '%s' や ':1' などの異なった表記を用いることがあります。) 例を示します。

```
# Never do this -- insecure!
symbol = 'IBM'
c.execute("... where symbol = '%s'" % symbol)

# Do this instead
t = (symbol,)
c.execute('select * from stocks where symbol=?', t)

# Larger example
for t in (('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
          ('2006-04-05', 'BUY', 'MSOFT', 1000, 72.00),
          ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
         ):
    c.execute('insert into stocks values (?,?,?,?,?)', t)
```

SELECT 文を実行した後データを取得する方法は 3 つありどれを使っても構いません。一つはカーソルをイテレータとして扱う、一つはカーソルの `fetchone()` メソッドを呼んで一致した内の一行を取得する、もう一つは `fetchall()` メソッドを呼んで一致した全ての行のリストとして受け取る、という 3 つです。

以下の例ではイテレータの形を使います。

```
>>> c = conn.cursor()
>>> c.execute('select * from stocks order by price')
>>> for row in c:
...     print row
...
(u'2006-01-05', u'BUY', u'RHAT', 100, 35.140000000000001)
(u'2006-03-28', u'BUY', u'IBM', 1000, 45.0)
(u'2006-04-06', u'SELL', u'IBM', 500, 53.0)
(u'2006-04-05', u'BUY', u'MSOFT', 1000, 72.0)
>>>
```

参考資料:

http://www.pysqlite.org
: pysqlite のウェブページ

http://www.sqlite.org
: SQLite のウェブページ。ここの文書ではサポートされる SQL 方言の文法と使えるデータ型を説明しています

PEP 249, "*Database API Specification 2.0*"
: Marc-André Lemburg により書かれた PEP

http://www.python.jp/doc/contrib/peps/pep-0249.txt
: 訳注: PEP 249 の日本語訳があります

13.13.1 モジュールの関数と定数

PARSE_DECLTYPES
: この定数は `connect` 関数の *detect_types* パラメータとして使われます。

  この定数を設定すると `sqlite3` モジュールは戻り値のカラムの宣言された型を読み取るようになります。意味を持つのは宣言の最初の単語です。すなわち、"integer primary key" においては "integer" が読み取られます。そしてそのコラムに対して、変換関数の辞書を探してその型に対して登録された関数を使うようにします。変換関数の名前は大文字と小文字を区別します!

PARSE_COLNAMES

この定数は connect 関数の *detect_types* パラメータとして使われます。

この定数を設定すると SQLite のインタフェースは戻り値のそれぞれのカラムの名前を読み取るようになります。文字列の中の [mytype] といった形の部分を探し、'mytype' がそのカラムの名前であると判断します。そして 'mytype' のエントリを変換関数辞書の中から見つけ、見つかった変換関数を値を返す際に用います。cursor.description で見つかるカラム名はその最初の単語だけです。すなわち、もし 'as "x [datetime]"' のようなものを SQL の中で使っていたとすると、読み取るのはカラム名の中の最初の空白までの全てですので、カラム名として使われるのは単純に "x" ということになります。

connect(*database*[, *timeout, isolation_level, detect_types, factory*])

ファイル *database* の SQLite データベースへの接続を開きます。":memory:" という名前を使うことでディスクの代わりに RAM 上のデータベースへの接続を開くこともできます。

データベースが複数の接続からアクセスされている状況で、その内の一つがデータベースに変更を加えたとき、SQLite データベースはそのトランザクションがコミットされるまでロックされます。*timeout* パラメータで、例外を送出するまで接続がロックが解除されるのをどれだけ待つかを決めます。デフォルトは 5.0 (5 秒) です。

isolation_level パラメータについては、13.13.2 節の Connection オブジェクトの isolation_level プロパティの説明を参照してください。

SQLite がネイティブにサポートするのは TEXT, INTEGER, FLOAT, BLOB および NULL 型だけです。もし他の型を使いたければ、その型のためのサポートを自分で追加しなければなりません。*detect_types* パラメータを、モジュールレベルの register_converter 関数で登録した自作の**変換関数**と一緒に使えば、簡単にできます。

パラメータ *detect_types* のデフォルトは 0 (つまりオフ、型検知無し) です。型検知を有効にするためには、PARSE_DECLTYPES と PARSE_COLNAMES の適当な組み合わせをこのパラメータにセットします。

デフォルトでは、sqlite3 モジュールは connect の呼び出しの際にモジュールの Connection クラスを使います。しかし、Connection クラスを継承したクラスを *factory* パラメータに渡して connect にそのクラスを使わせることもできます。詳しくはこのマニュアルの 13.13.4 節を参考にしてください。

sqlite3 モジュールは SQL 解析のオーバーヘッドを避けるために内部で文キャッシュを使っています。接続に対してキャッシュされる文の数を自分で指定したいならば、*cached_statements* パラメータに設定してください。現在の実装ではデフォルトでキャッシュされる SQL 文の数を 100 にしています。

register_converter(*typename, callable*)

データベースから得られるバイト列を希望する Python の型に変換する呼び出し可能オブジェクト (callable) を登録します。その呼び出し可能オブジェクトは型が *typename* である全てのデータベース上の値に対して呼び出されます。型検知がどのように働くかについては connect 関数の *detect_types* パラメータの説明も参照してください。注意が必要なのは *typename* はクエリの中の型名と大文字小文字も一致しなければならないということです。

register_adapter(*type, callable*)

自分が使いたい Python の型 *type* を SQLite がサポートしている型に変換する呼び出し可能オブジェクト (callable) を登録します。その呼び出し可能オブジェクト *callable* はただ一つの引数に Python の値を受け取り、int, long, float, (UTF-8 でエンコードされた) str, unicode または buffer のいずれかの型の値を返さなければなりません

complete_statement(*sql*)

もし文字列 *sql* がセミコロンで終端された一つ以上の完全な SQL 文であれば True を返します。判

定は SQL 文として文法的に正しいかではなく、閉じられていない文字列リテラルが無いことおよびセミコロンで終端されていることだけで行なわれます。

この関数は以下の例にあるような SQLite のシェルを作る際に使われます。

_statement.py

enable_callback_tracebacks (*flag*)
デフォルトでは、ユーザ定義の関数、集計関数、変換関数、認可コールバックなどはトレースバックを出力しません。デバッグの際にはこの関数を *flag* に `True` を指定して呼び出します。そうした後は先に述べたような関数のトレースバックが `sys.stderr` に出力されます。元に戻すには `False` を使います。

13.13.2 Connection オブジェクト

`Connection` のインスタンスには以下の属性とメソッドがあります:

isolation_level
現在の分離レベルを取得または設定します。None で自動コミットモードまたは"DEFERRED", "IMMEDIATE", "EXLUSIVE" のどれかです。より詳しい説明は 13.13.5 節「トランザクション制御」を参照してください。

cursor (*[cursorClass]*)
cursor メソッドはオプション引数 *CursorClass* を受け付けます。これを指定するならば、指定されたクラスは `sqlite3.Cursor` を継承したカーソルクラスでなければなりません。

execute (*sql, [parameters]*)
このメソッドは非標準のショートカットで、cursor メソッドを呼び出して中間的なカーソルオブジェクトを作り、そのカーソルの `execute` メソッドを与えられたパラメータと共に呼び出します。

executemany (*sql, [parameters]*)
このメソッドは非標準のショートカットで、cursor メソッドを呼び出して中間的なカーソルオブジェクトを作り、そのカーソルの `executemany` メソッドを与えられたパラメータと共に呼び出します。

executescript (*sql_script*)
このメソッドは非標準のショートカットで、cursor メソッドを呼び出して中間的なカーソルオブジェクトを作り、そのカーソルの `executescript` メソッドを与えられたパラメータと共に呼び出します。

create_function (*name, num_params, func*)
後から SQL 文中で *name* という名前の関数として使えるユーザ定義関数を作成します。*num_params* は関数が受け付ける引数の数、 *func* は SQL 関数として使われる Python の呼び出し可能オブジェクトです。

関数は SQLite でサポートされている任意の型を返すことができます。具体的には unicode, str, int, long, float, buffer および None です。

例:

```
import sqlite3
import md5

def md5sum(t):
    return md5.md5(t).hexdigest()

con = sqlite3.connect(":memory:")
con.create_function("md5", 1, md5sum)
cur = con.cursor()
cur.execute("select md5(?)", ("foo",))
print cur.fetchone()[0]
```

create_aggregate(*name, num_params, aggregate_class*)

ユーザ定義の集計関数を作成します。

集計クラスには パラメータ *num_params*　で指定される個数の引数を取る `step` メソッドおよび最終的な集計結果を返す `finalize` メソッドを実装しなければなりません。

`finalize` メソッドは SQLite でサポートされている任意の型を返すことができます。具体的には unicode, str, int, long, float, buffer および None です。

例:

```
import sqlite3

class MySum:
    def __init__(self):
        self.count = 0

    def step(self, value):
        self.count += value

    def finalize(self):
        return self.count

con = sqlite3.connect(":memory:")
con.create_aggregate("mysum", 1, MySum)
cur = con.cursor()
cur.execute("create table test(i)")
cur.execute("insert into test(i) values (1)")
cur.execute("insert into test(i) values (2)")
cur.execute("select mysum(i) from test")
print cur.fetchone()[0]
```

create_collation(*name, callable*)

name と *callable* で指定される照合順序を作成します。呼び出し可能オブジェクトには二つの文字列が渡されます。一つめのものが二つめのものより低く順序付けられるならば -1 を返し、等しければ 0 を返し、一つめのものが二つめのものより高く順序付けられるならば 1 を返すようにしなければなりません。この関数はソート (SQL での ORDER BY) をコントロールするもので、比較を行なうことは他の SQL 操作には影響を与えないことに注意しましょう。

また、呼び出し可能オブジェクトに渡される引数は Python のバイト文字列として渡されますが、それは通常 UTF-8 で符号化されたものになります。

以下の例は「間違った方法で」ソートする自作の照合順序です:

_reverse.py

照合順序を取り除くには `create_collation` を callable として None を渡して呼び出します:

```
con.create_collation("reverse", None)
```

interrupt()

このメソッドを別スレッドから呼び出して接続上で現在実行中であろうクエリを中断させられます。クエリが中断されると呼び出し元は例外を受け取ります。

set_authorizer(*authorizer_callback*)

このルーチンはコールバックを登録します。コールバックはデータベースのテーブルのカラムにアクセスしようとするたびに呼び出されます。コールバックはアクセスが許可されるならば `SQLITE_OK` を、SQL 文全体がエラーとともに中断されるべきならば `SQLITE_DENY` を、カラムが NULL 値と

して扱われるべきなら SQLITE_IGNORE を返さなければなりません。これらの定数は sqlite3 モジュールに用意されています。

コールバックの第一引数はどの種類の操作が許可されるかを決めます。第二第三引数には第一引数に依存して本当に使われる引数か None かが渡されます。第四引数はもし適用されるならばデータベースの名前 ("main", "temp", etc.) です。第五引数はアクセスを試みる要因となった最も内側のトリガまたはビューの名前、またはアクセスの試みが入力された SQL コードに直接起因するものならば None です。

第一引数に与えることができる値や、その第一引数によって決まる第二第三引数の意味については、SQLite の文書を参考にしてください。必要な定数は全て sqlite3 モジュールに用意されています。

row_factory

この属性を、カーソルとタプルの形での元の行のデータを受け取り最終的な行を表すオブジェクトを返す呼び出し可能オブジェクトに、変更することができます。これによって、より進んだ結果の返し方を実装することができます。例えば、カラムの名前で各データにアクセスできるようなオブジェクトを返したりできます。

例:

_factory.py

タプルを返すのでは物足りず、名前に基づいたカラムへのアクセスが行ないたい場合は、高度に最適化された sqlite3.Row 型を row_factory にセットすることを考えてはいかがでしょうか。Row クラスでは添字でも大文字小文字を無視した名前でもカラムにアクセスでき、しかもほとんどメモリーを浪費しません。おそらく、辞書を使うような独自実装のアプローチよりも、もしかすると db の行に基づいた解法よりも良いものかもしれません。

text_factory

この属性を使って TEXT データ型をどのオブジェクトで返すかを制御できます。デフォルトではこの属性は unicode に設定されており、sqlite3 モジュールは TEXT を Unicode オブジェクトで返します。もしバイト列で返したいならば、str に設定してください。

効率の問題を考えて、非 ASCII データに限って Unicode オブジェクトを返し、その他の場合にはバイト列を返す方法もあります。これを有効にしたければ、この属性を sqlite3.OptimizedUnicode に設定してください。

バイト列を受け取って望みの型のオブジェクトを返すような呼び出し可能オブジェクトを何でも設定して構いません。

以下の説明用のコード例を参照してください:

_factory.py

total_changes

データベース接続が開始されて以来の行の変更・挿入・削除がなされた行の総数を返します。

13.13.3 カーソルオブジェクト

Cursor のインスタンスはには以下の属性とメソッドがあります:

execute(*sql,* [*parameters*])

SQL 文を実行します。SQL 文はパラメータ化できます (すなわち SQL リテラルの代わりの場所確保文字 (placeholder) を入れておけます)。sqlite3 モジュールは 2 種類の場所確保記法をサポートします。一つは疑問符 (qmark スタイル)、もう一つは名前 (named スタイル) です。

まず最初の例は qmark スタイルのパラメータを使った書き方を示します:

_1.py

次の例は named スタイルの使い方です:

_2.py

`execute()` は一つの SQL 文しか実行しません。二つ以上の文を実行しようとすると、Warning を発生させます。複数の SQL 文を一つの呼び出しで実行したい場合は `executescript()` を使ってください。

executemany(*sql, seq_of_parameters*)

SQL 文 *sql* を *seq_of_parameters* の全てのパラメータシーケンスまたはマッピングに対して実行します。`sqlite3` モジュールでは、シーケンスの代わりにパラメータの組を作り出すイテレータ使うことが許されています。

_1.py

もう少し短いジェネレータを使った例です:

_2.py

executescript(*sql_script*)

これは非標準の便宜メソッドで、一度に複数の SQL 文を実行することができます。メソッドは最初に COMMIT 文を発行し、次いで引数として渡された SQL スクリプトを実行します。

sql_script はバイト文字列または Unicode 文字列です。

例:

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.executescript("""
    create table person(
        firstname,
        lastname,
        age
    );

    create table book(
        title,
        author,
        published
    );

    insert into book(title, author, published)
    values (
        'Dirk Gently''s Holistic Detective Agency',
        'Douglas Adams',
        1987
    );
    """)
```

rowcount

一応 `sqlite3` モジュールの `Cursor` クラスはこの属性を実装していますが、データベースエンジン自身の「影響を受けた行」/「選択された行」の決定方法は少し風変わりです。

`SELECT` 文では、全ての行を取得し終えるまで全部で何行になったか決められないので `rowcount` はいつでも None です。

DELETE 文では、条件を付けずに DELETE FROM table とすると SQLite は rowcount を 0 と報告します。

executemany では、変更数が rowcount に合計されます。

Python DB API 仕様で求められているように、rowcount 属性は「現在のカーソルがまだ executeXXX() を実行していない場合や、データベースインタフェースから最後に行った操作の結果行数を決定できない場合には、この属性は -1 となります」。

13.13.4 SQLite と Python の型

入門編

SQLite が最初からサポートしているのは次の型です: NULL, INTEGER, REAL, TEXT, BLOB。

したがって、次の Python の型は問題なく SQLite に送り込めます:

Python の型	SQLite の型
None	NULL
int	INTEGER
long	INTEGER
float	REAL
str (UTF8 エンコード)	TEXT
unicode	TEXT
buffer	BLOB

SQLite の型から Python の型へのデフォルトでの変換は以下の通りです:

SQLite の型	Python の型
NULL	None
INTEGER	int または long (サイズによる)
REAL	float
TEXT	text_factory に依存して決まるがデフォルトでは unicode
BLOB	buffer

sqlite3 モジュールの型システムは二つの方法で拡張できます。一つはオブジェクト適合 (adaptation) を通じて追加された Python の型を SQLite に格納することです。もう一つは変換関数 (converter) を通じて sqlite3 モジュールに SQLite の型を違った Python の型に変換させることです。

追加された Python の型を SQLite データベースに格納するために適合関数を使う

既に述べたように、SQLite が最初からサポートする型は限られたものだけです。それ以外の Python の型を SQLite で使うには、その型を sqlite3 モジュールがサポートしている型の一つに **適合** させなくてはなりません。サポートしている型というのは、NoneType, int, long, float, str, unicode, buffer です。

sqlite3 モジュールは PEP 246 に述べられているような Python オブジェクト適合を用います。使われるプロトコルは PrepareProtocol です。

sqlite3 モジュールで望みの Python の型をサポートされている型の一つに適合させる方法は二つあります。

オブジェクト自身で適合するようにする　自分でクラスを書いているならばこの方法が良いでしょう。次のようなクラスがあるとします:

```
class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y
```

さてこの点を SQLite の一つのカラムに収めたいと考えたとしましょう。最初にしなければならないのはサポートされている型の中から点を表現するのに使えるものを選ぶことです。ここでは単純に文字列を使うことにして、座標を区切るのにはセミコロンを使いましょう。次に必要なのはクラスに変換された値を返す `__conform__(self, protocol)` メソッドを追加することです。引数 *protocol* は `PrepareProtocol` になります。

_point_1.py

適合関数を登録する　もう一つの可能性は型を文字列表現に変換する関数を作り `register_adapter` でその関数を登録することです。

注意: 適合させる型/クラスは新形式クラスでなければなりません。すなわち、`object` を基底クラスの一つとしていなければなりません。

_point_2.py

`sqlite3` モジュールには二つの Python 標準型 `datetime.date` と `datetime.datetime` に対するデフォルト適合関数があります。いま `datetime.datetime` オブジェクトを ISO 表現でなく UNIX タイムスタンプとして格納したいとしましょう。

_datetime.py

SQLite の値を好きな Python 型に変換する

適合関数を書くことで好きな Python 型を SQLite に送り込めるようになりました。しかし、本当に使い物になるようにするには Python から SQLite さらに Python へという往還 (roundtrip) の変換ができる必要があります。

そこで変換関数 (converter) です。

`Point` クラスの例に戻りましょう。x, y 座標をセミコロンで区切った文字列として SQLite に格納したのでした。

まず、文字列を引数として取り `Point` オブジェクトをそれから構築する変換関数を定義します。

注意: 変換関数は SQLite に送り込んだデータ型に関係なく**常に**文字列を渡されます。

注意: 変換関数の名前を探す際、大文字と小文字は区別されます。

```
def convert_point(s):
    x, y = map(float, s.split(";"))
    return Point(x, y)
```

次に `sqlite3` モジュールにデータベースから取得したものが本当に点であることを教えなければなりません。二つの方法があります:

- 宣言された型を通じて暗黙的に
- カラム名を通じて明示的に

どちらの方法も 13.13.1 節 “モジュールの関数と定数” の中で説明されています。それぞれ PARSE_DECLTYPES 定数と PARSE_COLNAMES 定数の項目です。

以下の例で両方のアプローチを紹介します。

_point.py

デフォルトの適合関数と変換関数

datetime モジュールの date 型および datetime 型のためのデフォルト適合関数があります。これらの型は ISO 日付 / ISO タイムスタンプとして SQLite に送られます。

デフォルトの変換関数は datetime.date 用が "date" という名前で、datetime.datetime 用が "timestamp" という名前で登録されています。

これにより、多くの場合特別な細工無しに Python の日付 / タイムスタンプを使えます。適合関数の書式は実験的な SQLite の date/time 関数とも互換性があります。

以下の例でこのことを確かめます。

_datetime.py

13.13.5 トランザクション制御

デフォルトでは、sqlite3 モジュールはデータ変更言語 (DML) 文 (すなわち INSERT/UPDATE/DELETE/REPLACE) の前に暗黙のうちにトランザクションを開始し、非 DML、非クエリ文 (すなわち SELECT/INSERT/UPDATE/DELETE/REPLACE のいずれでもないもの) の前にトランザクションをコミットします。

ですから、もしトランザクション中に CREATE TABLE ..., VACUUM, PRAGMA といったコマンドを発行すると、sqlite3 モジュールはそのコマンドの実行前に暗黙のうちにコミットします。このようにする理由は二つあります。第一にこうしたコマンドのうちの幾つかはトランザクション中ではうまく動きません。第二に pysqlite はトランザクションの状態 (トランザクションが掛かっているかどうか) を追跡する必要があるからです。

pysqlite が暗黙のうちに実行する"BEGIN"文の種類 (またはそういうものを使わないこと) を connect 呼び出しの *isolation_level* パラメータを通じて、または接続の isolation_level プロパティを通じて、制御することができます。

もし**自動コミットモード**が使いたければ、isolation_level は None にしてください。

そうでなければデフォルトのまま"BEGIN"文を使い続けるか、SQLite がサポートする分離レベル DEFERRED, IMMEDIATE または EXCLUSIVE を設定してください。

sqlite モジュールがトランザクション状態を把握する必要があるので、SQL の中で OR ROLLBACK や ON CONFLICT ROLLBACK を使ってはなりません。その代わりに、IntegrityError を捕捉して接続の rollback メソッドを自分で呼び出すようにしてください。

13.13.6 pysqlite の効率的な使い方

ショートカットメソッドを使う

Connection オブジェクトの非標準的なメソッド execute, executemany, executescript を使うことで、(しばしば余計な) Cursor オブジェクトをわざわざ作り出さずに済むので、コードをより簡潔に書くことができます。Cursor オブジェクトは暗黙裡に生成されショートカットメソッドの戻り値として受け取ることができます。この方法を使えば、 SELECT 文を実行してその結果について反復することが、Connection オブジェクトに対する呼び出し一つで行なえます。

_methods.py

位置ではなく名前でカラムにアクセスする

sqlite3 モジュールの有用な機能の一つに、行生成関数として使われるための sqlite3.Row クラスがあります。

このクラスでラップされた行は、位置インデクス (タプルのような) でも大文字小文字を区別しない名前でもアクセスできます:

```
import sqlite3

con = sqlite3.connect("mydb")
con.row_factory = sqlite3.Row

cur = con.cursor()
cur.execute("select name_last, age from people")
for row in cur:
    assert row[0] == row["name_last"]
    assert row["name_last"] == row["nAmE_lAsT"]
    assert row[1] == row["age"]
    assert row[1] == row["AgE"]
```

十四章

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。

os	雑多なオペレーティングシステムインタフェース。
time	時刻データへのアクセスと変換
optparse	より便利で柔軟性に富んだ強力なコマンドライン解析ライブラリ
getopt	ポータブルなコマンドラインオプションのパーザ。長短の両方の形式をサポートします。
logging	PEP 282 に基づく Python 用のロギングモジュール。
getpass	ポータブルなパスワードとユーザー ID の検索
curses	可搬性のある端末操作を提供する curses ライブラリへのインタフェース.
curses.textpad	curses ウィンドウ内での Emacs ライクな入力編集機能。
curses.wrapper	curses プログラムのための端末設定ラッパ。
curses.ascii	ASCII 文字に関する定数および集合帰属関数。
curses.panel	curses 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。
platform	実行中プラットフォームからできるだけ多くの固有情報を取得する
errno	標準の errno システムシンボル。
ctypes	A foreign function library for Python.

14.1 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。

14.1.1 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。

このマップ型の内容は、os モジュールの最初の import の時点、通常は Python の起動時に 'site.py' が処理される中で取り込まれます。それ以後に変更された環境変数は os.environ を直接変更しない限り反映されません。

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: putenv() を直接呼び出しても os.environ の内容は変わらないので、os.environ を直接変更する方がベターです。**注意:** FreeBSD と Mac OS X を含むいつく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に変更を加えたコピー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。

プラットフォームが unsetenv() 関数をサポートしているならば、このマッピングからアイテムを取り除いて環境変数を取り消すことができます。unsetenv() は os.environ からアイテムが取り除かれた時に自動的に呼ばれます。

chdir(*path*)

getcwd()

これらの関数は、"ファイルとディレクトリ" (14.1.4 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 を含むいつく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(*euid*)
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、スーパユーザしか利用できません。利用できる環境: 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

getsid(*pid*)
システムコール `getsid()` 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。 2.4 で追加された仕様です。

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

unsetenv(*varname*)
varname という名前の環境変数を取り消します。このような環境の変化は `os.system()`、`popen()` または `fork()` と `execv()` で開始されるサブプロセスに影響を与えます。利用できる環境: ほとんどの UNIX 互換環境、Windows

`unsetenv()` がサポートされている時には `os.environ` のアイテムの削除が対応する `unsetenv()` の呼び出しに自動的に翻訳されます。しかし、`unsetenv()` の呼び出しは `os.environ` を更新しませんので、むしろ `os.environ` のアイテムを削除する方が好ましい方法です。

14.1.2 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 2.3 で変更された仕様: 引数 *mode* は、指定されるならば、'r'、 'w'、'a' のいずれかの文字で始まらなければなりません。そうでなければ `ValueError` が送出されます 2.5 で変更された仕様: UNIX では、引数 *mode* が 'a' で始まる時には *O_APPEND* フラグがファイル記述子に設定されます。(ほとんどのプラットフォームで `fdopen()` 実装が既に行なっていることです)

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'` か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: Macintosh、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: Macintosh、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。

また UNIX ではこれらの変種はいずれも *cmd* をシーケンスにできます。その場合、引数はシェルの介在なしに直接 (`os.spawnv()` のように) 渡されます。*cmd* が文字列の場合、引数は (`os.system()` のように) シェルに渡されます。

以下のメソッドは子プロセスから終了ステータス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 17.4.2) 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: Macintosh、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: Macintosh、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: Macintosh、UNIX、Windows 2.0 で追加された仕様です。

(*child_stdin*, *child_stdout*, および *child_stderr* は子プロセスの視点で名付けられているので注意してください。すなわち、*child_stdin* とは子プロセスの標準入力を意味します。)

この機能は `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。

14.1.3 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。

ファイル記述子とは現在のプロセスから開かれたファイルに対応する小さな整数です。例えば、標準入力のファイル記述子はいつでも 0 で、標準出力は 1、標準エラーは 2 です。その他にさらにプロセスから開かれたファイルには 3、4、5、などが割り振られます。「ファイル記述子」という名前は少し誤解を与えるものかもしれませんが、UNIX プラットフォームにおいて、ソケットやパイプもファイル記述子によって参照されます。

`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

注意: 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: Macintosh、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: Macintosh、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: Macintosh、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: Macintosh、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: Macintosh、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: Macintosh、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)。

注意: 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。ファイル記述子を「ファイルオブジェクト」でラップするには `fdopen()` を使ってください。

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: Macintosh、いくつかの UNIX 系システム

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: Macintosh、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。

注意: 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: Macintosh、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: Macintosh、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: Macintosh、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。

注意: 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。いくつかのアイテムは全てのプラットフォームで使えるわけではありません。何が使えるか、また何に使うのかといった説明は *open*(2) を参照してください。

O_RDONLY

O_WRONLY

O_RDWR

O_APPEND

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 を取れます。利用できる環境: Macintosh、 UNIX、Windows。

O_DSYNC

O_RSYNC

O_SYNC

O_NDELAY

O_NONBLOCK

O_NOCTTY

O_SHLOCK

O_EXLOCK

上のフラグと同様、`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 を取れます。利用できる環境: Macintosh、 UNIX。

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: Windows。

O_NOINHERIT

O_SHORT_LIVED

O_TEMPORARY

O_RANDOM

O_SEQUENTIAL

O_TEXT

`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: Windows

SEEK_SET

SEEK_CUR

SEEK_END

`lseek()` 関数のパラメータです。値はそれぞれ 0、1、2 です。利用できる環境: Windows、Macintosh、UNIX 2.5 で追加された仕様です。

14.1.4 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 `True` を、そうでない場合 `False` 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、UNIX、Windows

注意: `access()` を使ってユーザーが例えばファイルを開く権限を持っているか `open()` を使って実際にそうする前に調べることはセキュリティ・ホールを作り出してしまいます。というのは、調べる時点と開く時点の時間差を利用してそのユーザーがファイルを操作してしまうかもしれないからです。

注意: I/O 操作は `access()` が成功を思わせるときにも失敗することがありえます。特にネットワーク・ファイルシステムにおける操作が通常の POSIX 許可ビット・モデルをはみ出す意味論を備える場合にはそのようなことが起こりえます。

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。

getcwdu()

現在の作業ディレクトリを表現するユニコードオブジェクトを返します。利用できる環境: Macintosh、UNIX、Windows 2.3 で追加された仕様です。

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: Macintosh、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またはビット単位の OR で組み合わせた値を取り得ます:

- `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: Macintosh、 UNIX、 Windows。

注意: Windows でも `chmod()` はサポートされていますが、ファイルの読み込み専用フラグを (定数 `S_IWRITE` と `S_IREAD`、または対応する整数値を通して) 設定できるだけです。他のビットは全て無視されます。

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。いずれかの id を変更せずにおくには、その値として -1 をセットします。利用できる環境: Macintosh、 UNIX。

lchown (*path, uid, gid*)

Change the owner and group id of *path* to the numeric *uid* and gid. This function will not follow symbolic links. *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。この関数はシンボリックリンクをたどりません。利用できる環境: Macintosh、 UNIX。 2.3 で追加された仕様です。

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: Macintosh、 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てもリストには含められません。利用できる環境: Macintosh、 UNIX、 Windows。

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。

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: Macintosh、 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: Macintosh、 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 を読み出し用に開き、クライアントは書き込み用に開きます。`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 (*filename*[, *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、名前つきパイプ) 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を (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。(たいてい `stat` の `st_dev` フィールドか `st_rdev` フィールドです) 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。(たいてい `stat` の `st_dev` フィールドか `st_rdev` フィールドです) 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。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。**注意:** `makedirs()` は作り出すパス要素が *os.pardir* を含むと混乱することになります。 1.5.2 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 UNC パスを正しく扱えるようになりました

pathconf (*path, name*)

指定されたファイルに関係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: Macintosh、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。

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: Macintosh、 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: Macintosh、 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 が送出されます; 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出します; 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、removedirs() は *path* に現れる親ディレクトリをエラーが送出されるまで (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) 順に削除することを試みます。例えば、'os.removedirs('foo/bar/baz')' では最初にディレクトリ ''foo/bar/baz'' を削除し、次に ''foo/bar''、さらに ''foo'' をそれらが空ならば削除します。末端のディレクトリが削除できなかった場合には OSError が送出されます。 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。 1.5.2 で追加された仕様です。

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。

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 (プラットフォーム依存：UNIX では最終メタデータ変更時刻、Windows では作成時刻) となっています。

```
>>> import os
>>> statinfo = os.stat('somefile.txt')
>>> statinfo
(33188, 422511L, 769L, 1, 1032, 100, 926L, 1105022698,1105022732, 1105022732)
>>> statinfo.st_size
926L
>>>
```

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)、st_flags (ファイルに対するユーザー定義のフラグ) も利用可能なときがあります。

他の (FreeBSD のような) UNIX システムでは、以下の属性: st_gen (ファイル生成番号)、st_birthtime (ファイル生成時刻) も利用可能なときがあります (ただし root がそれらを使うことにした場合以外は値が入っていないでしょう)。

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_ftype (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。)

注意: st_atime, st_mtime, および st_ctime メンバの厳密な意味や精度はオペレーティングシステムやファイルシステムによって変わります。例えば、FAT や FAT32 ファイルシステムを使っている Windows システムでは、st_atime の精度は 1 日に過ぎません。詳しくはお使いのオペレーティングシステムのドキュメントを参照してください。

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 2.5 で変更された仕様: st_gen、 st_birthtime 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 True の場合、以後の stat() 呼び出しは浮動小数点を返し、False の場合には以後整数を返します。*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。

2.5 で変更された仕様: Python はデフォルトで浮動小数点数を返すようになりました。浮動小数点数のタイムスタンプではうまく動かないアプリケーションはこの機能を利用して昔ながらの振る舞いを取り戻すことができます。

タイムスタンプの精度 (すなわち最小の小数部分) はシステム依存です。システムによっては秒単位の精度しかサポートしません。そういったシステムでは小数部分は常に 0 です。

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_bsize`、`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、シーケンス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()` (第 14.1.2 節) を使うよう検討してください。利用できる環境: Macintosh、 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()` (第 14.1.2 節) 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; Micorosoft 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。*path* にディレクトリを指定できるかどうかは、オペレーティングシステムがディレクトリをファイルの一種として実装しているかどうかに依存します (例えば、Windows はそうではありません)。ここで設定した時刻の値は、オペレーティングシステムがアクセス時刻や更新時刻を記録する際の精度によっては、後で stat() 呼び出したときの値と同じにならないかも知れないので注意してください。stat() も参照してください。 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プルの後で (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 *dirpath* 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、OSError 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 属性として取得でき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:

```
# Delete everything reachable from the directory named in 'top',
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(os.path.join(root, name))
    for name in dirs:
        os.rmdir(os.path.join(root, name))
```

2.3 で追加された仕様です。

14.1.5 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 プログラマにとっては、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: Macintosh、 UNIX、 Windows。

`execl` (*path, arg0, arg1, ...*)

`execle` (*path, arg0, arg1, ..., env*)

`execlp` (*file, arg0, arg1, ...*)

`execlpe` (*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めるべきですが、これは強制ではありません。

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: Macintosh、 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行いません。利用できる環境: Macintosh、 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。**注意:** いくらかの違いがあって、これらの全てが全ての UNIX プラットフォームで使えるわけではありません。以下の定数は基礎にあるプラットフォームで定義されていれば定義されます。

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: Macintosh、 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使われたことを表す終了コード。利用できる環境: Macintosh、 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: Macintosh、 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: Macintosh、 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: Macintosh、 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: Macintosh、 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: Macintosh、 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: Macintosh、 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: Macintosh、 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: Macintosh、 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: Macintosh、 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: Macintosh、 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: Macintosh、 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: Macintosh、 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: Macintosh、 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: Macintosh、 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: Macintosh、 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: Macintosh、 UNIX。

forkpty()
子プロセスを fork します。このとき新しい擬似端末 (psheudo-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: Macintosh、 いくつかの UNIX 系。

kill(*pid, sig*)
プロセス *pid* にシグナル *sig* を送ります。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: Macintosh、 UNIX。

killpg(*pgid, sig*)
プロセスグループ *pgid* にシグナル *sig* を送ります。利用できる環境: Macintosh、 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: Macintosh、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: Macintosh、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 14.1.2 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くてはなりません。

`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: Macintosh、 UNIX、Windows。 1.6 で追加された仕様です。

P_WAIT

`spawn*()` 関数ファミリに対する *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: Macintosh、 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*[, *operation*])

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。

operation が指定されないかまたは `'open'` であるとき、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。

他の *operation* が与えられる場合、それはファイルに対して何がなされるべきかを表す “command verb” (コマンドを表す動詞) でなければなりません。Microsoft が文書化している動詞は、`'print'` と `'edit'` (ファイルに対して) および `'explore'` と `'find'` (ディレクトリに対して) です。

`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。 2.5 で追加された仕様: *operation* パラメータ

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: Macintosh、 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: Macintosh、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: Macintosh、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。

wait3([*options*])

`waitpid()` に似ていますが、プロセス id を引数に取らず、子プロセス id、終了ステータスインジケータ、リソース使用情報の 3 要素からなるタプルを返します。リソース使用情報の詳しい情報は `resource.getrusage()` を参照してください。*options* は `waitpid()` および `wait4()` と同様です。利用できる環境: UNIX。 2.5 で追加された仕様です。

wait4(*pid, options*)

`waitpid()` に似ていますが、子プロセス id、終了ステータスインジケータ、リソース使用情報の 3 要素からなるタプルを返します。リソース使用情報の詳しい情報は `resource.getrusage()` を参照してください。`wait4()` の引数は `waitpid()` に与えられるものと同じです。利用できる環境: UNIX。 2.5 で追加された仕様です。

WNOHANG

子プロセス状態がすぐに取得できなかった場合に直ちに終了するようにするための `waitpid()` のオプションです。この場合、関数は `(0, 0)` を返します。利用できる環境: Macintosh、UNIX。

WCONTINUED

このオプションによって子プロセスは前回状態が報告された後にジョブ制御による停止状態から実行を継続された場合に報告されるようになります。利用できる環境: ある種の UNIX システム。 2.3 で追加された仕様です。

WUNTRACED

このオプションによって子プロセスは停止されていながら停止されてから状態が報告されていない場合に報告されるようになります。利用できる環境: Macintosh、 UNIX。 2.3 で追加された仕様です。

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。

WCOREDUMP(*status*)

プロセスに対してコアダンプが生成されていた場合には `True` を、それ以外の場合は `False` を返します。利用できる環境: Macintosh、 UNIX。 2.3 で追加された仕様です。

WIFCONTINUED(*status*)

プロセスがジョブ制御による停止状態から実行を継続された (continue) 場合に `True` を、それ以外の場合は `False` を返します。利用できる環境: UNIX。 2.3 で追加された仕様です。

WIFSTOPPED(*status*)

プロセスが停止された (stop) 場合に `True` を、それ以外の場合は `False` 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 `True` を、それ以外の場合は `False` を返します。利用できる環境: Macintosh、 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 `True` を、それ以外の場合は `False` を返します。利用できる環境: Macintosh、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: Macintosh、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: Macintosh、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: Macintosh、UNIX

14.1.6 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` 辞書のキーとして与えられています。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: Macintosh、UNIX。

name に指定された設定値が定義されていない場合、`None` 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: Macintosh、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: Macintosh、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: Macintosh、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 '.' 、Mac OS 9 では':' 。os.path からも利用できます。

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 '..' 、Mac OS 9 では'::' 。os.path からも利用できます。

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、Mac OS 9 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 からも利用できます。

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 からも利用できます。

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 からも利用できます。 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 からも利用できます。

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 からも利用できます。

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 Mac OS 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。

devnull

ヌルデバイス (null device) のファイルパスです。例えば POSIX では'/dev/null'、Mac OS 9 では'Dev:Nul' です。この値は os.path からも利用できます。 2.4 で追加された仕様です。

14.1.7 雑多な関数

urandom(*n*)

暗号に関する用途に適した *n* バイトからなるランダムな文字列を返します。

この関数は OS 固有の乱数発生源からランダムなバイト列を生成して返します。この関数の返すデータは暗号を用いたアプリケーションで十分利用できる程度に予測不能ですが、実際のクオリティは OS の実装によって異なります。UNIX 系のシステムでは '/dev/urandom' への問い合わせを行い、Windows では CryptGenRandom を使います。乱数発生源が見つからない場合、NotImplementedError を送出します。 2.4 で追加された仕様です。

14.2 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 つの整数からなるシーケンスです。

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 `strftime()` 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 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* を指定しない、または None を指定した場合、time() が返す値を現在の時刻として使います。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 2.4 で変更された仕様: *secs* が None の場合に現在時刻を使うようになりました

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* を指定しない、または None を指定した場合、time() が返す値を現在の時刻として使います。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 2.4 で変更された仕様: *secs* が None の場合に現在時刻を使うようになりました

localtime([*secs*])

gmtime() に似ていますが、ローカルタイムに変換します。*secs* を指定しない、または None を指定した場合、time() が返す値を現在の時刻として使います。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。 2.4 で変更された仕様: *secs* が None の場合に現在時刻を使うようになりました

mktime(*t*)

localtime() の逆を行う関数です。引数は struct_time か完全な 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に、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。*t* のいずれかのフィールドが許容範囲外の数値であった場合、`ValueError` を送出します。 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。 2.4 で変更された仕様: *t* のフィールド値が許容範囲外の値の場合に `ValueError` を送出するようになりました 2.5 で変更された仕様: 0 は時刻値タプルのどこでも使用可能になりました。もし不正な値の場合には正常な値に修正されます。

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。	
`%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。	(1)
`%S`	秒を表す 10 進数 [00,61]。	(2)
`%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。	(3)
`%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。	(3)
`%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) `strptime()` 関数で使う場合、`%p` ディレクティブが出力結果の時刻フィールドに影響を及ぼすのは、時刻を解釈するために `%I` を使ったときのみです。

(2) 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。

(3) `strptime()` 関数で使う場合、`%U` および `%W` を計算に使うのは曜日と年を指定したときだけです。

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。[1]

[1] 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

```
>>> 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。

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 strftime() 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。欠落したデータについて、適切な値を推測できない場合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 を認識する時以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() が返す時刻値シーケンスのタイプです。 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います。

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 年になるだいぶ以前から 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。

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なっています:

std と 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の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:

J*n*ユリウス日 (Julian day) *n* (1 <= *n* <= 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 <= *n* <= 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。

M*m.n.dm* 月の第 *n* 週における *d* 番目の日 (0 <= *d* <= 6, 1 <= *n* <= 5, 1 <= *m* <= 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:

datetime モジュール (5.1 節):
: 日付と時刻に対する、よりオブジェクト指向のインタフェースです。

locale モジュール (21.2 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 (5.2 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。

14.3 optparse — より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、getopt よりも簡便で、柔軟性に富み、かつ強力なコマンドライン解析ライブラリです。optparse では、より明快なスタイルのコマンドライン解析手法、すなわち OptionParser のインスタンスを作成してオプションを追加してゆき、そのインスタンスでコマンドラインを解析するという手法をとっています。optparse を使うと、GNU/POSIX 構文でオプションを指定できるだけでなく、使用法やヘルプメッセージの生成も行えます。

optparse を使った簡単なスクリプト例を以下に示します:

```
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=True,
                  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例えば以下のような 「よくある使い方」 を実行できるようになります:

```
<yourscript> --file=outfile -q
```

コマンドライン解析の中で、optparse はユーザの指定したコマンドライン引数値に応じて parse_args() の返す options の属性値を設定してゆきます。parse_args() がコマンドライン解析から処理を戻したとき、options.filename は"outfile" に、options.verbose は False になっているはずです。optparse は長い形式と短い形式の両方のオプション表記をサポートしており、短い形式は結合して指定できます。また、様々な形でオプションに引数値を関連付けられます。従って、以下のコマンドラインは全て上の例と同じ意味になります:

```
<yourscript> -f outfile --quiet
<yourscript> --quiet --file outfile
<yourscript> -q -foutfile
<yourscript> -qfoutfile
```

さらに、ユーザが

```
<yourscript> -h
<yourscript> --help
```

のいずれかを実行すると、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
  -f FILE, --file=FILE  write report to FILE
  -q, --quiet           don't print status messages to stdout
```

yourscript の中身は実行時に決まります (通常は sys.argv[0] になります)。

14.3.1 Background

optparse は、素直で慣習に則ったコマンドラインインタフェースを備えたプログラムの作成を援助する目的で設計されました。その結果、UNIX で慣習的に使われているコマンドラインの構文や機能だけをサポートするに留まっています。こうした慣習に詳しくなければ、よく知っておくためにもこの節を読んでおきましょう。

Terminology

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する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」と読むべきでしょう。

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 における書法はハイフン ("-") 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られます。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りの語を指定する方法、例えば "--file" や "--dry-run" も提供しています。optparse は、これら二種類のオプション書法だけをサポートしています。

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うことはありません)
- 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"

上記のオプション書法は optparse ではサポートしておらず、今後もサポートする予定はありません。これは故意によるものです: 最初の三つはどの環境の標準でもなく、最後の一つは VMS や MS-DOS, そして Windows を対象にしているときにしか意味をなさないからです。

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。optparse では、オプション引数は以下のように別々の引数にできます:

```
-f foo
--file foo
```

また、一つの引数中にも入れられます:

```
-ffoo
--file=foo
```

通常、オプションは引数をとることもとらないこともあります。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。これは、あるオプションが引数が指定されている場合には引数をとり、そうでない場合には引数をもた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、議論の的となっています: 例えば、もし **-a** がオプション引数をとり、**-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？こうした曖昧さが存在するため、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 (required option)」という語は、英語では矛盾した言葉です。optparse では必須オプションの実装を妨げてはいませんが、とりたてて実装上役立つこともしていません。optparse で必須オプションを実装する方法は、optparse ソースコード配布物中の examples/required_1.py や examples/required_2.py を参照してください。

例えば、下記のような架空のコマンドラインを考えてみましょう:

```
prog -v --report /tmp/report.txt foo bar
```

"-v" と"--report" はどちらもオプションです。**--report** オプションが引数をとるとすれば、"/tmp/report.txt" はオプションの引数です。"foo"と"bar" は固定引数になります。

オプションとは何か

オプションはプログラムの実行を調整したり、カスタマイズしたりするための補助的な情報を与えるために使います。もっとはっきりいうと、オプションはあくまでもオプション (省略可能) であるということです。本来、プログラムはともかくもオプションなしでうまく実行できてしかるべきです。(UNIX や GNU ツールセットのプログラムをランダムにピックアップしてみてください。オプションを全く指定しなくてもちゃんと動くでしょう？例外は find, tar, dd くらいです—これらの例外は、オプション文法が標準的でなく、インタフェースが混乱を招くと酷評されてきた変種のはみ出しものなのです)

多くの人が自分のプログラムに「必須のオプション」を持たせたいと考えます。しかしよく考えてください。必須なら、それは**オプション *(省略可能)* ではないのです！**プログラムを正しく動作させるのに絶対的に必要な情報があるとすれば、そこには固定引数を割り当てるべきなのです。

良くできたコマンドラインインタフェース設計として、ファイルのコピーに使われる cp ユーティリティのことを考えてみましょう。ファイルのコピーでは、コピー先を指定せずにファイルをコピーするのは無意味な操作ですし、少なくとも一つのコピー元が必要です。従って、cp は引数無しで実行すると失敗します。とはいえ、cp はオプションを全く必要としない柔軟で便利なコマンドライン文法を備えています:

```
cp SOURCE DEST
cp SOURCE ... DEST-DIR
```

まだあります。ほとんどの cp の実装では、ファイルモードや変更時刻を変えずにコピーする、シンボリックリンクの追跡を行わない、すでにあるファイルを上書きする前にユーザに尋ねる、など、ファイルをコピーする方法をいじるための一連のオプションを実装しています。しかし、こうしたオプションは、一つのファイルを別の場所にコピーする、または複数のファイルを別のディレクトリにコピーするという、cp の中心的な処理を乱すことはないのです。

固定引数とは何か

固定引数とは、プログラムを動作させる上で絶対的に必要な情報となる引数です。

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ものです。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**方法** はさして問題にはなりません —ユーザはプログラムを正しく動作させられないうちに諦め、立ち去ってしまうからです。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 やその他の何であっても同じです: 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てしまうだけなのです。

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— そして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 でできた「環境設定」ダイアログ上のウィジェットであろうが、コマンドラインオプションであろうが関係ありません — より多くの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ますが、実装はより難解になるのです。高すぎる柔軟性はユーザを閉口させ、コードの維持をより難しくするのです。

14.3.2 Tutorial

optparse はとても柔軟で強力でありながら、ほとんどの場合には簡単に利用できます。この節では、optparse ベースのプログラムで広く使われているコードパターンについて述べます。

まず、OptionParser クラスを import しておかねばなりません。次に、プログラムの冒頭で OptionParser インスタンスを生成しておきます:

```
from optparse import OptionParser
[...]
parser = OptionParser()
```

これでオプションを定義できるようになりました。基本的な構文は以下の通りです:

```
parser.add_option(opt_str, ...,
                  attr=value, ...)
```

各オプションには、"-f" や"--file" のような一つまたは複数のオプション文字列と、パーザがコマンドライン上のオプションを見つけた際に、何を準備し、何を行うべきかを optparse に教えるためのオプション属性 (option attribute) がいくつか入ります。

通常、各オプションには短いオプション文字列と長いオプション文字列があります。例えば:

```
parser.add_option("-f", "--file", ...)
```

といった具合です。

オプション文字列は、(ゼロ文字の場合も含め) いくらでも短く、またいくらでも長くできます。ただしオプション文字列は少なくとも一つなければなりません。

add_option() に渡されたオプション文字列は、実際にはこの関数で定義したオプションに対するラベルになります。簡単のため、以後ではコマンドライン上で**オプションを見つける** という表現をしばしば使いますが、これは実際には optparse がコマンドライン上の**オプション文字列**を見つけ、対応づけされているオプションを捜し出す、という処理に相当します。

オプションを全て定義したら、optparse にコマンドラインを解析するように指示します:

```
(options, args) = parser.parse_args()
```

(お望みなら、parse_args() に自作の引数リストを渡してもかまいません。とはいえ、実際にはそうした必要はほとんどないでしょう: optionparser はデフォルトで sys.argv[1:] を使うからです。)

parse_args() は二つの値を返します:

- 全てのオプションに対する値の入ったオブジェクト options — 例えば、"--file" が単一の文字列引数をとる場合、options.file はユーザが指定したファイル名になります。オプションを指定しなかった場合には None になります。
- オプションの解析後に残った固定引数からなるリスト args。

このチュートリアルの節では、最も重要な四つのオプション属性: action, type, dest (destination), および help についてしか触れません。このうち最も重要なのは action です。

オプション・アクションを理解する

アクション (action) は optparse が コマンドライン上にあるオプションを見つけたときに何をすべきかを指示します。optparse には押し着せのアクションのセットがハードコードされています。新たなアクションの追加は上級者向けの話題であり、14.3.5 の「optparse の拡張」で触れます。ほとんどのアクションは、値を何らかの変数に記憶するよう optparse に指示します — 例えば、文字列をコマンドラインから取り出して、options の属性の中に入れる、といった具合にです。

オプション・アクションを指定しない場合、optparse のデフォルトの動作は store になります。

store アクション

もっとも良く使われるアクションは store です。このアクションは次の引数 (あるいは現在の引数の残りの部分) を取り出し、正しい型の値か確かめ、指定した保存先に保存するよう optparse に指示します。

例えば:

```
parser.add_option("-f", "--file",
                  action="store", type="string", dest="filename")
```

のように指定しておき、偽のコマンドラインを作成して optparse に解析させてみましょう:

```
args = ["-f", "foo.txt"]
(options, args) = parser.parse_args(args)
```

オプション文字列 "-f" を見つけると、optparse は次の引数である "foo.txt" を消費し、その値を options.filename に保存します。従って、この parse_args() 呼び出し後には options.filename は"foo.txt"になっています。

オプションの型として、optparse は他にも int や float をサポートしています。

整数の引数を想定したオプションの例を示します:

```
parser.add_option("-n", type="int", dest="num")
```

このオプションには長い形式のオプション文字列がないため、設定に問題がないということに注意してください。また、デフォルトのアクションは store なので、ここでは action を明示的に指定していません。

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くたにします: **-n42** (一つの引数のみ) は **-n 42** (二つの引数からなる) と等価になるので、

```
(options, args) = parser.parse_args(["-n42"])
print options.num
```

は "42" を出力します。

型を指定しない場合、 optparse は引数を string であると仮定します。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")
```

保存先 (destination) を指定しない場合、 optparse はデフォルト値としてオプション文字列から気のきいた名前を設定します: 最初に指定した長い形式のオプション文字列が"--foo-bar" であれば、デフォルトの保存先は foo_bar になります。長い形式のオプション文字列がなければ、optparse は最初に指定した短い形式のオプション文字列を探します: 例えば、"-f" に対する保存先は f になります。

optparse では、long や complex といった組み込み型も取り入れています。型の追加は 14.3.5 節の「optparse の拡張」で触れています。

ブール値 (フラグ) オプションの処理

フラグオプション—特定のオプションに対して真または偽の値の値を設定するオプション— はよく使われます。optparse では、二つのアクション、store_true および store_false をサポートしています。例えば、verbose というフラグを"-v" で有効にして、"-q" で無効にしたいとします:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```

ここでは二つのオプションに同じ保存先を指定していますが、全く問題ありません (下記のように、デフォルト値の設定を少し注意深く行わねばならないだけです)

`"-v"` をコマンドライン上に見つけると、`optparse` は `options.verbose` を `True` に設定します。`"-q"` を見つければ、`options.verbose` は `False` にセットされます。

その他のアクション

この他にも、`optparse` は以下のようなアクションをサポートしています:

`store_const` 定数値を保存します。

`append` オプションの引数を指定のリストに追加します。

`count` 指定のカウンタを 1 増やします。

`callback` 指定の関数を呼び出します。

これらのアクションについては、14.3.3 節の「リファレンスガイド」および 14.3.4 節の「オプション・コールバック」で触れます。

デフォルト値

上記の例は全て、何らかのコマンドラインオプションが見つかった時に何らかの変数 (保存先: destination) に値を設定していました。では、該当するオプションが見つからなかった場合には何が起きるのでしょうか？デフォルトは全く与えていないため、これらの値は全て `None` になります。たいていはこれで十分ですが、もっときちんと制御したい場合もあります。`optparse` では各保存先に対してデフォルト値を指定し、コマンドラインの解析前にデフォルト値が設定されるようにできます。

まず、 verbose/quiet の例について考えてみましょう。`optparse` に対して、`"-q"` がない限り `verbose` を `True` 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=True)
parser.add_option("-q", action="store_false", dest="verbose")
```

デフォルトの値は特定のオプションではなく **保存先** に対して適用されます。また、これら二つのオプションはたまたま同じ保存先を持っているにすぎないため、上のコードは下のコードと全く等価になります:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

下のような場合を考えてみましょう:

```
parser.add_option("-v", action="store_true", dest="verbose", default=False)
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

やはり `verbose` のデフォルト値は `True` になります; 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、最後に指定した値だからです。

デフォルト値をすっきりと指定するには、OptionParser の set_defaults() メソッドを使います。このメソッドは parse_args() を呼び出す前ならいつでも使えます:

```
parser.set_defaults(verbose=True)
parser.add_option(...)
(options, args) = parser.parse_args()
```

前の例と同様、あるオプションの値の保存先に対するデフォルトの値は最後に指定した値になります。コードを読みやすくするため、デフォルト値を設定するときには両方のやり方を混ぜるのではなく、片方だけを使うようにしましょう。

ヘルプの生成

optparse にはヘルプと使い方の説明 (usage text) を生成する機能があり、ユーザに優しいコマンドラインインタフェースを作成する上で役立ちます。やらなければならないのは、各オプションに対する help の値と、必要ならプログラム全体の使用法を説明する短いメッセージを与えることだけです。

ユーザフレンドリな (ドキュメント付きの) オプションを追加した OptionParser を以下に示します:

```
usage = "usage: %prog [options] arg1 arg2"
parser = OptionParser(usage=usage)
parser.add_option("-v", "--verbose",
                  action="store_true", dest="verbose", default=True,
                  help="make lots of noise [default]")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose",
                  help="be vewwy quiet (I'm hunting wabbits)")
parser.add_option("-f", "--filename",
                  metavar="FILE", help="write output to FILE"),
parser.add_option("-m", "--mode",
                  default="intermediate",
                  help="interaction mode: novice, intermediate, "
                       "or expert [default: %default]")
```

optparse がコマンドライン上で"-h" や"--help" を見つけた場合やユーザが parser.print_help() を呼び出した場合、この OptionParser は以下のようなメッセージ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)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]
```

(help オプションでヘルプを出力した場合、optparse は出力後にプログラムを終了します。)

optparse ができるだけうまくメッセージを生成するよう手助けするには、他にもまだまだやるべきことがあります:

- スクリプト自体の利用法を表すメッセージを定義します:

```
usage = "usage: %prog [options] arg1 arg2"
```

optparse は "%prog" を現在のプログラム名、すなわち os.path.basename(sys.argv[0]) と置き換えます。この文字列は詳細なオプションヘルプの前に展開され出力されます。

usage の文字列を指定しない場合、optparse は型どおりとはいえ気の効いたデフォルト値、"usage: %prog [options]" を使います。固定引数をとらないスクリプトの場合はこれで十分でしょう。

- 全てのオプションにヘルプ文字列を定義します。行の折り返しは気にしなくてかまいません — optparse は行の折り返しに気を配り、見栄えのよいヘルプ出力を生成します。
- オプションが値をとるということは自動的に生成されるヘルプメッセージの中で分かります。例えば、“mode” option の場合には:

  ```
  -m MODE, --mode=MODE
  ```

  のようになります。

  ここで “MODE” はメタ変数 (meta-variable) と呼ばれます: メタ変数は、ユーザが **-m/--mode** に対して指定するはずの引数を表します。デフォルトでは、optparse は保存先の変数名を大文字だけにしたものをメタ変数に使います。これは時として期待通りの結果になりません — 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テキストは:

  ```
  -f FILE, --filename=FILE
  ```

  のようになります。

  この機能の重要さは、単に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、手作業で書いたヘルプテキストの中でメタ変数として “FILE” を使っています。その結果、ユーザに対してやや堅苦しい表現の書法 “-f FILE” と、より平易に意味付けを説明した “write output to FILE” との間に対応があるというヒントを与えています。これは、エンドユーザにとってより明解で便利なヘルプテキストを作成する単純でありながら効果的な手法なのです。

- デフォルト値を持つオプションのヘルプ文字列には%default を入れられます — optparse は %default をデフォルト値の str() で置き換えます。該当するオプションにデフォルト値がない場合 (あるいはデフォルト値が None である場合) %default の展開結果は none になります。

バージョン番号の出力

optparse では、使用法メッセージと同様にプログラムのバージョン文字列を出力できます。OptionParser の version 引数に文字列を渡します:

```
parser = OptionParser(usage="%prog [-f] [-q]", version="%prog 1.0")
```

"%prog" は *usage* と同じような展開を受けます。その他にも version には何でも好きな内容を入れられます。version を指定した場合、optparse は自動的に"--version" オプションをパーザに渡します。コマンドライン中に"--version" が見つかると、optparse は version 文字列を展開して ("%prog" を置き換えて) 標準出力に出力し、プログラムを終了します。

例えば、 /usr/bin/foo という名前のスクリプトなら:

```
$ /usr/bin/foo --version
foo 1.0
```

のようになります。

optparse のエラー処理法

optparse を使う場合に気を付けねばならないエラーには、大きく分けてプログラマ側のエラーとユーザ側のエラーという二つの種類があります。プログラマ側のエラーの多くは、例えば不正なオプション文字列や定義されていないオプション属性の指定、あるいはオプション属性を指定し忘れるといった、誤った parser.add_option() 呼び出しによるものです。こうした誤りは通常通りに処理されます。すなわち、例外 (optparse.OptionError や TypeError) を送出して、プログラムをクラッシュさせます。もっと重要なのはユーザ側のエラーの処理です。というのも、ユーザの操作エラーというものはコードの安定性に関係なく起こるからです。optparse は、誤ったオプション引数の指定 (整数を引数にとるオプション **-n** に対して "-n4x" と指定してしまうなど) や、引数を指定し忘れた場合 (**-n** が何らかの引数をとるオプションであるのに、"-n" が引数の末尾に来ている場合) といった、ユーザによるエラーを自動的に検出します。また、アプリケーション側で定義されたエラー条件が起きた場合、parser.error() を呼び出してエラーを通知できます:

```
(options, args) = parser.parse_args()
[...]
if options.a and options.b:
    parser.error("options -a and -b are mutually exclusive")
```

いずれの場合にも optparse はエラーを同じやり方で処理します。すなわち、プログラムの使用法メッセージとエラーメッセージを標準エラー出力に出力して、終了ステータス 2 でプログラムを終了させます。

上に挙げた最初の例、すなわち整数を引数にとるオプションにユーザが "4x" を指定した場合を考えてみましょう:

```
$ /usr/bin/foo -n 4x
usage: foo [options]

foo: error: option -n: invalid integer value: '4x'
```

値を全く指定しない場合には、以下のようになります:

```
$ /usr/bin/foo -n
usage: foo [options]

foo: error: -n option requires an argument
```

optparse は、常にエラーを引き起こしたオプションについて説明の入ったエラーメッセージを生成するよう気を配ります; 従って、parser.error() をアプリケーションコードから呼び出す場合にも、同じようなメッセージになるようにしてください。

optparse のデフォルトのエラー処理動作が気に入らないのなら、OptionParser をサブクラス化して、exit() かつ/または error() をオーバライドする必要があります。

全てをつなぎ合わせる

optparse を使ったスクリプトは、通常以下のようになります:

```
from optparse import OptionParser
[...]
def main():
    usage = "usage: %prog [options] arg"
    parser = OptionParser(usage)
    parser.add_option("-f", "--file", dest="filename",
                      help="read data from FILENAME")
    parser.add_option("-v", "--verbose",
                      action="store_true", dest="verbose")
    parser.add_option("-q", "--quiet",
                      action="store_false", dest="verbose")
    [...]
    (options, args) = parser.parse_args()
    if len(args) != 1:
        parser.error("incorrect number of arguments")
    if options.verbose:
        print "reading %s..." % options.filename
    [...]

if __name__ == "__main__":
    main()
```

14.3.3 リファレンスガイド

Creating the parser

optparse を使う最初の一歩は OptionParser インスタンスを作ることです。

```
parser = OptionParser(...)
```

OptionParser のコンストラクタの引数はどれも必須ではありませんが、いくつものキーワード引数がオプションとして使えます。これらはキーワード引数として渡さなければなりません。すなわち、引数が宣言されている順番に頼ってはいけません。

usage (デフォルト: "%prog [options]") プログラムが間違った方法で実行されるかまたはヘルプオプションを付けて実行された場合に表示される使用法です。optparse は使用法の文字列を表示する際に %prog を os.path.basename(sys.argv[0]) (または prog キーワード引数が指定されていればその値) に展開します。使用法メッセージを抑制するためには特別な optparse.SUPPRESS_USAGE という値を指定します。

option_list (デフォルト: []) パーザに追加する Option オブジェクトのリストです。option_list の中のオプションは standard_option_list (OptionParser のサブクラスでセットされる可能性のあるクラス属性) の後に追加されますが、バージョンやヘルプのオプションよりは前になります。このオプションの使用は推奨されません。パーザを作成した後で、add_option() を使って追加してください。

option_class (デフォルト: optparse.Option) add_option() でパーザにオプションを追加するときに使用されるクラス。

version (デフォルト: None) ユーザがバージョンオプションを与えたときに表示されるバージョン文字列です。version に真の値を与えると、optparse は自動的に単独のオプ

ション文字列 "--version" とともにバージョンオプションを追加します。部分文字列 "%prog" は usage と同様に展開されます。

conflict_handler (デフォルト: "error") オプション文字列が衝突するようなオプションがパーザに追加されたときにどうするかを指定します。14.3.3 節「オプション間の衝突」を参照して下さい。

description (デフォルト: None) プログラムの概要を表す一段落のテキストです。optparse はユーザがヘルプを要求したときにこの概要を現在のターミナルの幅に合わせて整形し直して表示します (usage の後、オプションリストの前に表示されます)。

formatter (デフォルト: 新しい IndentedHelpFormatter) ヘルプテキストを表示する際に使われる optparse.HelpFormatter のインスタンスです。optparse はこの目的のためにすぐ使えるクラスを二つ提供しています。IndentedHelpFormatter と TitledHelpFormatter がそれです。

add_help_option (デフォルト: True) もし真ならば、optparse はパーザにヘルプオプションを (オプション文字列 "-h" と "--help" とともに) 追加します。

prog usage や version の中の "%prog" を展開するときに os.path.basename(sys.argv[0]) の代わりに使われる文字列です。

パーザへのオプション追加

パーザにオプションを加えていくにはいくつか方法があります。推奨するのは 14.3.2 節のチュートリアルで示したような OptionParser.add_option() を使う方法です。add_option() は以下の二つのうちいずれかの方法で呼び出せます:

- make_option() に (すなわち Option のコンストラクタに) 固定引数とキーワード引数の組み合わせを渡して、Option インスタンスを生成させます。
- (make_option() などが返す)Option インスタンスを渡します。

もう一つの方法は、あらかじめ作成しておいた Option インスタンスからなるリストを、以下のようにして OptionParser のコンストラクタに渡すというものです:

```
option_list = [
    make_option("-f", "--filename",
                action="store", type="string", dest="filename"),
    make_option("-q", "--quiet",
                action="store_false", dest="verbose"),
    ]
parser = OptionParser(option_list=option_list)
```

(make_option() は Option インスタンスを生成するファクトリ関数です; 現在のところ、個の関数は Option のコンストラクタの別名にすぎません。optparse の将来のバージョンでは、Option を複数のクラスに分割し、make_option() は適切なクラスを選んでインスタンスを生成するようになる予定です。従って、Option を直接インスタンス化しないでください。)

オプションの定義

各々の Option インスタンス、は**-f** や**--file** といった同義のコマンドラインオプションからなる集合を表現しています。一つの Option には任意の数のオプションを短い形式でも長い形式でも指定できます。ただし、少なくとも一つは指定せねばなりません。

正しい方法で Option インスタンスを生成するには、OptionParser の add_option() を使います:

```
parser.add_option(opt_str[, ...], attr=value, ...)
```

短い形式のオプション文字列を一つだけ持つようなオプションを生成するには:

```
parser.add_option("-f", attr=value, ...)
```

のようにします。

また、長い形式のオプション文字列を一つだけ持つようなオプションの定義は:

```
parser.add_option("--foo", attr=value, ...)
```

のようになります。

キーワード引数は新しい Option オブジェクトの属性を定義します。オプションの属性のうちでもっとも重要なのは action です。action は他のどの属性と関連があるか、そしてどの属性が必要かに大きく作用します。関係のないオプション属性を指定したり、必要な属性を指定し忘れたりすると、optparse は誤りを解説した OptionError 例外を送出します。

コマンドライン上にあるオプションが見つかったときの optparse の振舞いを決定しているのは **アクション** *(action)* です。optparse でハードコードされている標準的なアクションには以下のようなものがあります:

store オプションの引数を保存します (デフォルトの動作です)

store_const 定数を保存します

store_true 真 (True) を保存します

store_false 偽 (False) を保存します

append オプションの引数をリストに追加します

append_const 定数をリストに追加します

count カウンタを一つ増やします

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のドキュメントの入っ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い場合、デフォルトは store になります。このアクションでは、 type および dest オプション属性を指定せねばなりません。下記を参照してください。)

すでにお分かりのように、ほとんどのアクションはどこかに値を保存したり、値を更新したりします。この目的のために、optparse は常に特別なオブジェクトを作り出し、それは通常 options と呼ばれます (optparse.Values のインスタンスになっています)。オプションの引数 (や、その他の様々な値) は、dest (保存先: destination) オプション属性に従って、*options* の属性として保存されます。

例えば、

```
parser.parse_args()
```

を呼び出した場合、optparse はまず options オブジェクトを生成します:

```
options = Values()
```

パーザ中で以下のようなオプション

```
parser.add_option("-f", "--file", action="store", type="string", dest="filename")
```

が定義されていて、パーズしたコマンドラインに以下のいずれかが入っていた場合:

```
-ffoo
-f foo
--file=foo
--file foo
```

optparse はこのオプションを見つけて、

```
options.filename = "foo"
```

と同等の処理を行います。

type および dest オプション属性は action と同じくらい重要ですが、**全ての** オプションで意味をなすのは action だけなのです。

標準的なオプション・アクション

様々なオプション・アクションにはどれも互いに少しづつ異なった条件と作用があります。ほとんどのアクションに関連するオプション属性がいくつかあり、値を指定して optparse の挙動を操作できます; いくつかのアクションには必須の属性があり、必ず値を指定せねばなりません。

- store [relevant: type, dest, nargs, choices]

  オプションの後には必ず引数が続きます。引数は 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、複数の引数をコマンドラインから取り出します; 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。下記の 14.3.3 節「標準のオプション型」 を参照してください。

  choices を (文字列のリストかタプルで) 指定した場合、型のデフォルト値は “choice” になります。

  type を指定しない場合、デフォルトの値は string です。

  dest を指定しない場合、 optparse は保存先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" は foo_bar になります)。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保存先の変数名を導出します ("-f" は f になります)。

  例えば:

  ```
  parser.add_option("-f")
  parser.add_option("-p", type="float", nargs=3, dest="point")
  ```

  とすると、以下のようなコマンドライン:

```
-f foo.txt -p 1 -3.5 4 -fbar.txt
```

を解析した場合、optparse は

```
options.f = "foo.txt"
options.point = (1.0, -3.5, 4.0)
options.f = "bar.txt"
```

のように設定を行います。

- store_const [required: const; relevant: dest]

  値 cost を dest に保存します。

  例えば:

```
parser.add_option("-q", "--quiet",
                  action="store_const", const=0, dest="verbose")
parser.add_option("-v", "--verbose",
                  action="store_const", const=1, dest="verbose")
parser.add_option("--noisy",
                  action="store_const", const=2, dest="verbose")
```

  とします。

  "--noisy" が見つかると、 optparse は

```
options.verbose = 2
```

  のように設定を行います。

- store_true [relevant: dest]

  store_const の特殊なケースで、真 (True) を dest に保存します。

- store_false [relevant: dest]

  store_true と同じですが、偽 (False) を保存します。

  例:

```
parser.add_option("--clobber", action="store_true", dest="clobber")
parser.add_option("--no-clobber", action="store_false", dest="clobber")
```

- append [relevant: type, dest, nargs, choices]

  このオプションの後ろには必ず引数が続きます。引数は dest のリストに追加されます。dest のデフォルト値を指定しなかった場合、optparse がこのオプションを最初にみつけた時点で空のリストを自動的に生成します。nargs > 1 の場合、複数の引数をコマンドラインから取り出し、長さ nargs のタプルを生成して dest に追加します。

  type および dest のデフォルト値は store アクションと同じです。

  例:

```
parser.add_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:

```
options.tracks = []
options.tracks.append(int("3"))
```

と同等の処理を行います。

その後、"--tracks=4" が見つかると:

```
options.tracks.append(int("4"))
```

を実行します。

- append_const [required: const; relevant: dest]

  store_const と同様ですが、const の値は dest に追加 (append) されます。append の場合と同じように dest のデフォルトは None ですがこのオプションを最初にみつけた時点で空のリストを自動的に生成します。

- count [relevant: dest]

  dest に保存されている整数値をインクリメントします。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:

```
parser.add_option("-v", action="count", dest="verbosity")
```

  コマンドライン上で最初に "-v" が見つかると、optparse は:

```
options.verbosity = 0
options.verbosity += 1
```

  と同等の処理を行います。

  以後、"-v" が見つかるたびに、

```
options.verbosity += 1
```

  を実行します。

- callback [required: callback; relevant: type, nargs, callback_args, callback_kwargs]

  callback に指定された関数を次のように呼び出します。

```
func(option, opt_str, value, parser, *args, **kwargs)
```

  詳細は、14.3.4 節「オプション処理コールバック」を参照してください。

- help

  現在のオプションパーザ内の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した usage 文字列と、各オプションに渡した help 文字列から生成します。

オプションに help 文字列が指定されていなくても、オプションはヘルプメッセージ中に列挙されます。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 を使ってください。

optparse は全ての OptionParser に自動的に help オプションを追加するので、通常自分で生成する必要はありません。

例:

```
from optparse import OptionParser, SUPPRESS_HELP

parser = OptionParser()
parser.add_option("-h", "--help", action="help"),
parser.add_option("-v", action="store_true", dest="verbose",
                  help="Be moderately verbose")
parser.add_option("--file", dest="filename",
                  help="Input file to read data from"),
parser.add_option("--secret", help=SUPPRESS_HELP)
```

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 (sys.argv[0] は"foo.py" だとします):

```
usage: foo.py [options]

options:
  -h, --help        Show this help message and exit
  -v                Be moderately verbose
  --file=FILENAME   Input file to read data from
```

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

  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。help オプションと同様、optparse はこのオプションを必要に応じて自動的に追加するので、version オプションを作成することはほとんどないでしょう。

オプション属性

以下のオプション属性は parser.add_option() へのキーワード引数として渡すことができます。特定のオプションに無関係なオプション属性を渡した場合、または必須のオプションを渡しそこなった場合、optparse は OptionError を送出します。

- action (デフォルト: "store")

  このオプションがコマンドラインにあった場合に optparse に何をさせるかを決めます。取りうるオプションについては既に説明しました。

- type (デフォルト: "string")

  このオプションに与えられる引数の型 (たとえば "string" や"int") です。取りうるオプションの型については既に説明しました。

- `dest` (デフォルト: オプション文字列から)

  このオプションのアクションがある値をどこかに書いたり書き換えたりを意味する場合、これは `optparse` にその書く場所を教えます。詳しく言えば `dest` には `optparse` がコマンドラインを解析しながら組み立てる `options` オブジェクトの属性の名前を指定します。

- `default` (非推奨)

  コマンドラインに指定がなかったときにこのオプションの対象に使われる値です。使用は推奨されません。代わりに `parser.set_defaults()` を使ってください。

- `nargs` (デフォルト: 1)

  このオプションがあったときに幾つの `type` 型の引数が消費されるべきかを指定します。もし > 1 ならば、`optparse` は `dest` に値のタプルを格納します。

- `const`

  定数を格納する動作のための、その定数です。

- `choices`

  `"choice"` 型オプションに対してユーザがその中から選べる文字列のリストです。

- `callback`

  アクションが `"callback"` であるオプションに対し、このオプションがあったときに呼ばれる呼び出し可能オブジェクトです。`callable` に渡す引数の詳細については、14.3.4 節「オプション処理コールバック」を参照してください。

- `callback_args`, `callback_kwargs`

  `callback` に渡される標準的な 4 つのコールバック引数の後ろに追加する位置による引数またはキーワード引数です。

- `help`

  ユーザが `help` オプション (`"--help"` のような) を指定したときに表示される使用可能な全オプションのリストの中のこのオプションに関する説明文です。説明文を提供しておかなければ、オプションは説明文なしで表示されます。オプションを隠すには特殊な値 `SUPPRESS_HELP` を使います。

- `metavar` (デフォルト: オプション文字列から)

  説明文を表示する際にオプションの引数の身代わりになるものです。例は 14.3.2 節のチュートリアルを参照してください。

標準のオプション型

`optparse` には、*string* (文字列)、*int* (整数)、*long* (長整数)、 *choice* (選択肢)、 *float* (浮動小数点数) および *complex* (複素数) の 6 種類のオプション型があります。新たなオプションの型を追加したければ、14.3.5 節、「`optparse` の拡張」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や変換を一切受けません: コマンドライン上のテキストは保存先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 (`int` 型や `long` 型) は次のように読み取られます。

- 数が `0x` から始まるならば、16 進数として読み取られます
- 数が `0` から始まるならば、8 進数として読み取られます

- 数が 0b から始まるならば、2 進数として読み取られます
- それ以外の場合、数は 10 進数として読み取られます

変換は適切な底 (2, 8, 10, 16 のどれか) とともに int() または long() を呼び出すことで行なわれます。この変換が失敗した場合 optparse の処理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出力します。

float および complex のオプション引数は直接 float() や complex() で変換されます。エラーは同様の扱いです。

choice オプションは string オプションのサブタイプです。choice オプションの属性 (文字列からなるシーケンス) には、利用できるオプション引数のセットを指定します。optparse.check_choice() はユーザの指定したオプション引数とマスタリストを比較して、無効な文字列が指定された場合には OptionValueError を送出します。

引数の解析

OptionParser を作成してオプションを追加していく上で大事なポイントは、parse_args() メソッドの呼び出しです。

```
(options, args) = parser.parse_args(args=None, options=None)
```

ここで入力パラメータは

args 処理する引数のリスト (デフォルト: sys.argv[1:])

options オプション引数を格納するオブジェクト (デフォルト: 新しい optparse.Values のインスタンス)

であり、戻り値は

options options に渡されたものと同じオブジェクト、または optparse によって生成された optparse.Values インスタンス

args 全てのオプションの処理が終わった後で残った位置引数

です。

一番普通の使い方は一切キーワード引数を使わないというものです。options を指定した場合、それは繰り返される setattr() の呼び出し (大雑把に言うと保存される各オプション引数につき一回ずつ) で更新されていき、parse_args() で返されます。

parse_args() が引数リストでエラーに遭遇した場合、OptionParser の error() メソッドを適切なエンドユーザ向けのエラーメッセージとともに呼び出します。この呼び出しにより、最終的に終了ステータス 2 (伝統的な UNIX におけるコマンドラインエラーの終了ステータス) でプロセスを終了させることになります。

オプション解析器への問い合わせと操作

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なことがあります。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) OptionParser に ("-q" や "--verbose" のような) オプション opt_str がある場合、真を返します。

get_option(opt_str) オプション文字列 opt_str に対する Option インスタンスを返します。該当するオプションがなければ None を返します。

remove_option(opt_str) OptionParser に opt_str に対応するオプションがある場合、そのオプションを削除します。該当するオプションに他のオプション文字列が指定されていた場合、それらのオプション文字列は全て無効になります。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", ...)
```

(とりわけ、OptionParser から標準的なオプションを備えた自前のサブクラスを定義してしまった場合にはよく起きます。)

ユーザがオプションを追加するたびに、optparse は既存のオプションとの衝突がないかチェックします。何らかの衝突が見付かると、現在設定されている衝突処理メカニズムを呼び出します。衝突処理メカニズムはコンストラクタ中で呼び出せます:

```
parser = OptionParser(..., conflict_handler=handler)
```

個別にも呼び出せます:

```
parser.set_conflict_handler(handler)
```

衝突時の処理をおこなうハンドラ (handler) には、以下のものが利用できます:

error (デフォルトの設定) オプション間の衝突をプログラム上のエラーとみなし、OptionConflictError を送出します。

resolve オプション間の衝突をインテリジェントに解決します (下記参照)。

一例として、衝突をインテリジェントに解決する OptionParser を定義し、衝突を起こすようなオプションを追加してみましょう:

```
parser = OptionParser(conflict_handler="resolve")
parser.add_option("-n", "--dry-run", ..., help="do no harm")
parser.add_option("-n", "--noisy", ..., help="be noisy")
```

この時点で、optparse はすでに追加済のオプションがオプション文字列 "-n" を使っていることを検出します。conflict_handler が "resolve" なので、optparse は既に追加済のオプションリストの方から"-n" を除去して問題を解決します。従って、"-n" の除去されたオプションは"--dry-run" だけでしか有効にできなくなります。ユーザがヘルプ文字列を要求した場合、問題解決の結果を反映したメッセージが出力されます:

```
options:
  --dry-run     do no harm
  [...]
  -n, --noisy   be noisy
```

これまでに追加したオプション文字列を跡形もなく削り去り、ユーザがそのオプションをコマンドラインから起動する手段をなくせます。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どこにも表示されなくなります。例えば、現在の OptionParser の場合、以下の操作:

```
parser.add_option("--dry-run", ..., help="new dry-run option")
```

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ます。このため、optparse はオプションを消去してしまい、ヘルプテキスト:

```
options:
  [...]
  -n, --noisy   be noisy
  --dry-run     new dry-run option
```

だけが残ります。

クリーンアップ

OptionParser インスタンスはいくつかの循環参照を抱えています。このことは Python のガーベジコレクタにとって問題になるわけではありませんが、使い終わった OptionParser に対して destroy() を呼び出すことでこの循環参照を意図的に断ち切るという方法を選ぶこともできます。この方法は特に長時間実行するアプリケーションで OptionParser から大きなオブジェクトグラフが到達可能になっているような場合に有用です。

その他のメソッド

OptionParser にはその他にも幾つかの公開されたメソッドがあります:

- set_usage(usage)

  上で説明したコンストラクタの usage キーワード引数での規則に従った使用法の文字列をセットします。None を渡すとデフォルトの使用法文字列が使われるようになり、SUPPRESS_USAGE によって使用法メッセージを抑制できます。

- enable_interspersed_args(), disable_interspersed_args()

  位置引数をオプションと混ぜこぜにする GNU getopt のような扱いを有効化/無効化する (デフォルトでは有効)。たとえば、"-a" と "-b" はどちらも引数を取らない単純なオプションだとすると、optparse は通常つぎのような文法を受け入れます。

  ```
  prog -a arg1 -b arg2
  ```

  そして扱いは次のように指定した時と同じです。

```
prog -a -b arg1 arg2
```

この機能を無効化したい時は disable_interspersed_args() を呼び出してください。この呼び出しにより、伝統的な UNIX 文法に回帰し、オプションの解析は最初のオプションでない引数で止まるようになります。

- set_defaults(dest=value, ...)

  幾つかの保存先に対してデフォルト値をまとめてセットします。set_defaults() を使うのは複数のオプションにデフォルト値をセットする好ましいやり方です。というのも複数のオプションが同じ保存先を共有することがあり得るからです。たとえば幾つかの "mode" オプションが全て同じ保存先をセットするものだったとすると、どのオプションもデフォルトをセットすることができ、しかし最後に指定したものが勝ちます。

```
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced",
                  default="novice")    # 上書きされます
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice",
                  default="advanced")  # 上の設定を上書きします
```

  こうした混乱を避けるために set_defaults() を使います。

```
parser.set_defaults(mode="advanced")
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced")
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice")
```

14.3.4 オプション処理コールバック

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い場合、二つの選択肢があります: 一つは optparse の拡張、もう一つは callback オプションの定義です。optparse の拡張は汎用性に富んでいますが、単純なケースに対していささか大げさでもあります。大体は簡単なコールバックで事足りるでしょう。

callback オプションの定義は二つのステップからなります:

- callback アクションを使ってオプション自体を定義する。
- コールバックを書く。コールバックは少なくとも後で説明する 4 つの引数をとる関数 (またはメソッド) でなければなりません。

callback オプションの定義

callback オプションを最も簡単に定義するには、parser.add_option() メソッドを使います。action の他に指定しなければならない属性は callback、すなわちコールバックする関数自体です:

```
parser.add_option("-c", action="callback", callback=my_callback)
```

callback は関数 (または呼び出し可能オブジェクト) なので、callback オプションを定義する時にはあらかじめ my_callback() を定義しておかねばなりません。この単純なケースでは、optparse は **-c** が何らかの引数をとるかどうか判別できず、通常は**-c** が引数を伴わないことを意味します — 知りたいことはただ単に **-c** がコマンドライン上に現れたどうかだけです。とはいえ、場合によっては、自分のコールバック関数に任意の個数のコマンドライン引数を消費させたいこともあるでしょう。これがコールバック関数をトリッキーなものにしています; これについてはこの節の後の方で説明します。

optparse は常に四つの引数をコールバックに渡し、その他には callback_args および callback_kwargs で指定した追加引数しか渡しません。従って、最小のコールバック関数シグネチャは:

```
def my_callback(option, opt, value, parser):
```

のようになります。

コールバックの四つの引数については後で説明します。

callback オプションを定義する場合には、他にもいくつかオプション属性を指定できます:

type 他で使われているのと同じ意味です: store や append アクションの時と同じく、この属性は optparse に引数を一つ消費して、type に指定した型に変換させます。optparse は変換後の値をどこかに保存する代わりにコールバック関数に渡します。

nargs これも他で使われているのと同じ意味です: このオプションが指定されていて、かつ nargs > 1 である場合、 optparse は nargs 個の引数を消費します。このとき各引数は type 型に変換できねばなりません。変換後の値はタプルとしてコールバックに渡されます。

callback_args その他の固定引数からなるタプルで、コールバックに渡されます。

callback_kwargs その他のキーワード引数からなるタプルで、コールバックに渡されます。

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形式で呼び出されます:

```
func(option, opt_str, value, parser, *args, **kwargs)
```

ここで、

option 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_str 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。(長い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_str* は "--foobar" となります。)

value 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として指定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 (引数なしの) 場合、*value* は None になります。'nargs' > 1 であれば、value はは適切な型をもつ値のタプルになります。

parser 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。この変数が有用なのは、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からです:

parser.largs 現在放置されている引数、すなわち、すでに消費されたものの、オプションでもオプション引数でもない引数からなるリストです。parser.largs は自由に変更でき、たとえば引数を追加したりできます (このリストは args 、すなわち parse_args() の二つ目の戻り値になります)

parser.rargs 現在残っている引数、すなわち、 opt_str および value) があれば除き、それ以外の引数が残っているリストです。parser.rargs は自由に変更でき、例えばさらに引数を消費したりできます。

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 (optparse.OptionValues のインスタンス です。この値を使うと、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ため、グローバル変数や閉包 (closure) を台無しにしないので便利です。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もアクセスできます。

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。

コールバック中で例外を送出する

オプション自体か、あるいはその引数に問題があるばあい、コールバック関数は OptionValueError を送出せねばなりません。optparse はこの例外をとらえてプログラムを終了させ、ユーザが指定しておいたエラーメッセージを標準エラー出力に出力します。エラーメッセージは明確、簡潔かつ正確で、どのオプションに誤りがあるかを示さねばなりません。さもなければ、ユーザは自分の操作のどこに問題があるかを解決するのに苦労することになります。

コールバックの例 1: ありふれたコールバック

引数をとらず、発見したオプションを単に記録するだけのコールバックオプションの例を以下に示します:

```
def record_foo_seen(option, opt_str, value, parser):
    parser.saw_foo = True

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実現できます。

コールバックの例 2: オプションの順番をチェックする

もう少し面白みのある例を示します: この例では、"-b" を発見して、その後で"-a" がコマンドライン中に現れた場合にはエラーになります。

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use -a after -b")
    parser.values.a = 1
[...]
parser.add_option("-a", action="callback", callback=check_order)
parser.add_option("-b", action="store_true", dest="b")
```

コールバックの例 3: オプションの順番をチェックする (汎用的)

このコールバック (フラグを立てるが、"-b" が既に指定されていればエラーになる) を同様の複数のオプションに対して再利用したければ、もう少し作業する必要があります: エラーメッセージとセットされるフラグを一般化しなければなりません。

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use %s after -b" % opt_str)
    setattr(parser.values, option.dest, 1)
[...]
parser.add_option("-a", action="callback", callback=check_order, dest='a')
parser.add_option("-b", action="store_true", dest="b")
parser.add_option("-c", action="callback", callback=check_order, dest='c')
```

コールバックの例 4: 任意の条件をチェックする

もちろん、単に定義済みのオプションの値を調べるだけにとどまらず、コールバックには任意の条件を入れられます。例えば、満月でなければ呼び出してはならないオプションがあるとしましょう。やらなければならないことはこれだけです:

```
def check_moon(option, opt_str, value, parser):
    if is_moon_full():
        raise OptionValueError("%s option invalid when moon is full"
                               % opt_str)
    setattr(parser.values, option.dest, 1)
[...]
parser.add_option("--foo",
                  action="callback", callback=check_moon, dest="foo")
```

(`is_moon_full()` の定義は読者への課題としましょう。

コールバックの例 5: 固定引数

決まった数の引数をとるようなコールパックオプションを定義するなら、問題はやや興味深くなってきます。引数をとるようコールバックに指定するのは、`store` や `append` オプションの定義に似ています: `type` を定義していれば、そのオプションは引数を受け取ったときに該当する型に変換できねばなりません; さらに `nargs` を指定すれば、オプションは `nargs` 個の引数を受け取ります。

標準の `store` アクションをエミュレートする例を以下に示します:

```
def store_value(option, opt_str, value, parser):
    setattr(parser.values, option.dest, value)
[...]
parser.add_option("--foo",
                  action="callback", callback=store_value,
                  type="int", nargs=3, dest="foo")
```

`optparse` は 3 個の引数を受け取り、それらを整数に変換するところまで面倒をみてくれます; ユーザは単にそれを保存するだけです。 (他の処理もできます; いうまでもなく、この例にはコールバックは必要ありません)

コールバックの例 6: 可変個の引数

あるオプションに可変個の引数を持たせたいと考えているなら、問題はいささか手強くなってきます。この場合、optparse では該当する組み込みのオプション解析機能を提供していないので、自分でコールバックを書かねばなりません。さらに、optparse が普段処理している、伝統的な UNIX コマンドライン解析における難題を自分で解決せねばなりません。とりわけ、コールバック関数では引数が裸の"--" や "-" の場合における慣習的な処理規則:

- either "--" or "-" can be option arguments
- 裸の "--" (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し、"--"を無視します。
- 裸の"-" (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しますが、"-" は残します (parser.largs に追加します)。

を実装せねばなりません。

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 vararg_callback(option, opt_str, value, parser):
    assert value is None
    done = 0
    value = []
    rargs = parser.rargs
    while rargs:
        arg = rargs[0]

        # "--foo", "-a", "-fx", "--file=f" といった引数で停止。
        # "-3" や "-3.0" でも止まるので、オプションに数値が入る場合には
        # それを処理せねばならない。
        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、その引数は "-c" の引数ではなく次のオプションとして解釈される (そしておそらくエラーを引き起こす) 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。

14.3.5 optparse の拡張

optparse がコマンドラインオプションをどのように解釈するかを決める二つの重要な要素はそれぞれのオプションのアクションと型なので、拡張の方向は新しいアクションと型を追加することになると思います。

新しい型の追加

新しい型を追加するためには、optparse の Option クラスのサブクラスを自身で定義する必要があります。このクラスには optparse における型を定義する一対の属性があります。それは TYPES と TYPE_CHECKER です。

TYPES は型名のタプルです。新しく作るサブクラスでは、タプル TYPES は単純に標準的なもののを利用して定義すると良いでしょう。

TYPE_CHECKER は辞書で型名を型チェック関数に対応付けるものです。型チェック関数は以下のような引数をとります。

```
def check_mytype(option, opt, value)
```

ここで option は Option のインスタンスであり、opt はオプション文字列 (たとえば "-f") で、value は望みの型としてチェックされ変換されるべくコマンドラインで与えられる文字列です。check_mytype() は想定されている型 mytype のオブジェクトを返さなければなりません。型チェック関数から返される値は OptionParser.parse_args() で返される OptionValues インスタンスに収められるか、またはコールバックに value パラメータとして渡されます。

型チェック関数は何か問題に遭遇したら OptionValueError を送出しなければなりません。OptionValueError は文字列一つを引数に取り、それはそのまま OptionParser の error() メソッドに渡され、そこでプログラム名と文字列 "error:" が前置されてプロセスが終了する前に stderr に出力されます。

馬鹿馬鹿しい例ですが、Python スタイルの複素数を解析する complex オプション型を作ってみせることにします。(optparse 1.3 が複素数のサポートを組み込んでしまったため以前にも増して馬鹿らしくなりましたが、気にしないでください。)

最初に必要な import 文を書きます。

```
from copy import copy
from optparse import Option, OptionValueError
```

まずは型チェック関数を定義しなければなりません。これは後で (これから定義する Option のサブクラスの TYPE_CHECKER クラス属性の中で) 参照されることになります。

```
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 のサブクラスです。

```
class MyOption (Option):
    TYPES = Option.TYPES + ("complex",)
    TYPE_CHECKER = copy(Option.TYPE_CHECKER)
    TYPE_CHECKER["complex"] = check_complex
```

(もしここで Option.TYPE_CHECKER に copy() を適用しなければ、optparse の Option クラスの TYPE_CHECKER 属性をいじってしまうことになります。Python の常として、良いマナーと常識以外にそうすることを止めるものはありません。)

これだけです! もう新しいオプション型を使うスクリプトを他の optparse に基づいたスクリプトとまるで同じように書くことができます。ただし、 OptionParser に Option でなく MyOption を使うように指示しなければなければなりません。

```
parser = OptionParser(option_class=MyOption)
parser.add_option("-c", type="complex")
```

別のやり方として、オプションリストを構築して OptionParser に渡すという方法もあります。add_option() を上でやったように使わないならば、OptionParser にどのクラスを使うのか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)
```

新しいアクションの追加

新しいアクションの追加はもう少しトリッキーです。というのも optparse が使っている二つのアクションの分類を理解する必要があるからです。

“store” アクション optparse が値を現在の OptionValues の属性に格納することになるアクションです。この種類のオプションは Option のコンストラクタに dest 属性を与えることが要求されます。

“typed” アクション コマンドラインから引数を受け取り、それがある型であることが期待されているアクションです。もう少しはっきり言えば、その型に変換される文字列を受け取るものです。この種類のオプションは Option のコンストラクタに type 属性を与えることが要求されます。

この分類には重複する部分があります。デフォルトの “store” アクションには store、store_const、append、count などがありますが、デフォルトの “typed” オプションは store、append、callback の三つです。

アクションを追加する際に、以下の Option のクラス属性 (全て文字列のリストです) の中の少なくとも一つに付け加えることでそのアクションを分類する必要があります。

ACTIONS 全てのアクションは ACTIONS にリストされていなければなりません

STORE_ACTIONS “store” アクションはここにもリストされます

TYPED_ACTIONS “typed” アクションはここにもリストされます

ALWAYS_TYPED_ACTIONS 型を取るアクション (つまりそのオプションが値を取る) はここにもリストされます。このことの唯一の効果は optparse が、型の指定が無くアクションが ALWAYS_TYPED_ACTIONS のリストにあるオプションに、デフォルト型 string を割り当てるということだけです。

実際に新しいアクションを実装するには、Option の take_action() メソッドをオーバライドしてそのアクションを認識する場合分けを追加しなければなりません。

例えば、extend アクションというのを追加してみましょう。このアクションは標準的な append アクションと似ていますが、コマンドラインから一つだけ値を読み取って既存のリストに追加するのではなく、複数の値をコンマ区切りの文字列として読み取ってそれらで既存のリストを拡張します。すなわち、もし "--names" が string 型の extend オプションだとすると、次のコマンドライン

```
--names=foo,bar --names blah --names ding,dong
```

の結果は次のリストになります。

```
["foo", "bar", "blah", "ding", "dong"]
```

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",)
    ALWAYS_TYPED_ACTIONS = Option.ALWAYS_TYPED_ACTIONS + ("extend",)

    def take_action(self, action, dest, opt, value, values, parser):
        if action == "extend":
            lvalue = value.split(",")
            values.ensure_value(dest, []).extend(lvalue)
        else:
            Option.take_action(
                self, action, dest, opt, value, values, parser)
```

注意すべきは次のようなところです。

- extend はコマンドラインの値を予期していると同時にその値をどこかに格納しますので、STORE_ACTIONS と TYPED_ACTIONS の両方に入ります。
- optparse が extend アクションに string 型を割り当てるように extend アクションは ALWAYS_TYPED_ACTIONS にも入れてあります。
- MyOption.take_action() にはこの新しいアクション一つの扱いだけを実装してあり、他の標準的な optparse のアクションについては Option.take_action() に制御を戻すようにしてあります。
- values は optparse_parser.Values クラスのインスタンスであり、非常に有用な ensure_value() メソッドを提供しています。ensure_value() は本質的に安全弁付きの getattr() です。次のように呼び出します。

  ```
  values.ensure_value(attr, value)
  ```

  values に attr 属性が無いか None だった場合に、ensure_value() は最初に value をセットし、それから value を返します。この振る舞いは extend、append、count のように、データを変数に集積し、またその変数がある型 (最初の二つはリスト、最後のは整数) であると期待されるアクションを作るのにとても使い易いものです。ensure_value() を使えば、作ったアクションを使うスクリプトはオプションに保存先にデフォルト値をセットすることに煩わされずに済みます。デフォルトを None にしておけば ensure_value() がそれが必要になったときに適当な値を返してくれます。

14.4 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つけるためには、*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。

2.3 で追加された仕様です。

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のようになります:

```
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()
```

参考資料:

`optparse` モジュール (14.3 節):
よりオブジェクト指向的なコマンドラインオプションのパーズを提供します。

14.5 `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。

ログレベルの数値は以下の表のように与えられています。これらは基本的に自分でレベルを定義したい人のためのもので、定義するレベルを既存のレベルの間に位置づけるために具体的な値が必要になります。もし数値が他のレベルと同じだったら、既存の値は上書きされその名前は失われます。

レベル	数値
`CRITICAL`	50
`ERROR`	40
`WARNING`	30
`INFO`	20
`DEBUG`	10
`NOTSET`	0

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。
3. BaseRotatingHandler はログファイルをある時点で交替させるハンドラの基底クラスです。直接インスタンス化するためのクラスではありません。RotatingFileHandler や TimedRotatingFileHandler を使うようにしてください。
4. 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。
5. TimedRotatingFileHandler のインスタンスは、ログファイルを一定時間間隔ごとに交替しながら、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。
6. 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。
7. 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。
8. 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。
9. 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。
10. 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。
11. 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。
12. 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。*name* を指定する場合には、通常は *"a"*, *"a.b"*, あるいは *"a.b.c.d"* といったようなドット区切りの階層的な名前にします。名前の付け方はログ機能を使う開発者次第です。

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。

getLoggerClass()

標準の Logger クラスか、最後に setLoggerClass() に渡したクラスを返します。この関数は、新たに定義するクラス内で呼び出し、カスタマイズした Logger クラスのインストールを行うときに既に他のコードで適用したカスタマイズを取り消そうとしていないか確かめるのに使います。例えば以下のようにします:

```
class MyLogger(logging.getLoggerClass()):
    # ... override behaviour here
```

debug(*msg*[, **args*[, ***kwargs*]])

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* に文字列書式化演算子を使って取り込むための引数です。(これは、書式化文字列でキーワードを使い引数に辞書を渡すことができる、ということを意味します。)

キーワード引数 *kwargs* からは二つのキーワードが調べられます。一つめは *exc_info* で、この値の評価値が偽でない場合、例外情報をログメッセージに追加します。(sys.exc_info の返す形式の) 例外情報を表すタプルが与えられていれば、それをメッセージに使います。それ以外の場合には、sys.exc_info を呼び出して例外情報を取得します。

もう一つのキーワード引数は *extra* で、当該ログイベント用に作られた LogRecoed の __dict__ にユーザー定義属性を増やすのに使われる辞書を渡すのに用いられます。これらの属性は好きなように使えます。たとえば、ログメッセージの一部にすることもできます。以下の例を見てください:

```
FORMAT = "%(asctime)-15s %(clientip)s %(user)-8s %(message)s"
logging.basicConfig(format=FORMAT)
d = { 'clientip' : '192.168.0.1', 'user' : 'fbloggs' }
logging.warning("Protocol problem: %s", "connection reset", extra=d)
```

出力はこのようになります。

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs  Protocol problem: connection reset
```

extra で渡される辞書のキーはロギングシステムで使われているものとぶつからないようにしなければなりません。(どのキーがロギングシステムで使われているかについての詳細は Formatter のドキュメントを参照してください。)

これらの属性をログメッセージに使うことにしたなら、少し注意が必要です。上の例では、'clientip' と 'user' が LogRecord の属性辞書に含まれていることを期待した書式化文字列で Formatter はセットアップされています。これらの属性が欠けていると、書式化例外が発生してしまうためメッセージはログに残りません。したがってこの場合、常にこれらのキーがある *extra* 辞書を渡す必要があります。

このようなことは煩わしいかもしれませんが、この機能は限定された場面で使われるように意図しているものなのです。たとえば同じコードがいくつものコンテキストで実行されるマルチスレッドのサーバで、興味のある条件が現れるのがそのコンテキストに依存している (上の例で言えば、リモートのクライアント IP アドレスや認証されたユーザ名など)、というような場合です。そういった場面では、それ用の `Formatter` が特定の `Handler` と共に使われるというのはよくあることです。

2.5 で変更された仕様: *extra* が追加されました

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。

log(*level, msg*[, **args*[, ***kwargs*]])
レベル `level` のメッセージをルート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。

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を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。定義済みのレベルに対応する数値を指定した場合、レベルに対応した文字列表現を返します。そうでない場合、文字列 "Level %s" % lvl を返します。

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([***kwargs*])

デフォルトの 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。

2.4 で変更された仕様: 以前は basicConfig はキーワード引数をとりませんでした

以下のキーワード引数がサポートされます。

Format	説明
filename	StreamHandler ではなく指定された名前で FileHandler が作られます
filemode	filename が指定されているとき、ファイルモードを指定します (filemode が指定されない場合デフォル
format	指定された書式化文字列をハンドラで使います
datefmt	指定された日付/時刻の書式を使います
level	ルートロガーのレベルを指定されたものにします
stream	指定されたストリームを StreamHandler の初期化に使います。この引数は 'filename' と同時には使えな

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。

この logging パッケージのオリジナル

オリジナルの logging パッケージ。このサイトにあるバージョンのパッケージは、標準で logging パッケージを含まない Python 1.5.2 と 2.1.x、2.2.x でも使用できます

14.5.1 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 (これにより全てのメッセージについ

て、ロガーがルートロガーであれば処理される、そうでなくてロガーが非ルートロガーの場合には親ロガーに代行させる) に設定されます。ルートロガーは WARNING レベルで生成されることに注意してください。

「親ロガーに代行させる」という用語の意味は、もしロガーのレベルが NOTEST ならば、祖先ロガーの系列の中を NOTEST 以外のレベルの祖先を見つけるかルートに到達するまで辿っていく、ということです。

もし NOTEST 以外のレベルの祖先が見つかったなら、その祖先のレベルが祖先の探索を開始したロガーの実効レベルとして取り扱われ、ログイベントがどのように処理されるかを決めるのに使われます。

ルートに到達した場合、ルートのレベルが NOTEST ならば全てのメッセージは処理されます。そうでなければルートのレベルが実効レベルとして使われます。

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* に文字列書式化演算子を使って取り込むための引数です。(これは、書式化文字列でキーワードを使い引数に辞書を渡すことができる、ということを意味します。)

キーワード引数 *kwargs* からは二つのキーワードが調べられます。一つめは *exc_info* で、この値の評価値が偽でない場合、例外情報をログメッセージに追加します。(sys.exc_info の返す形式の) 例外情報を表すタプルが与えられていれば、それをメッセージに使います。それ以外の場合には、sys.exc_info を呼び出して例外情報を取得します。

もう一つのキーワード引数は *extra* で、当該ログイベント用に作られた LogRecoed の __dict__ にユーザー定義属性を増やすのに使われる辞書を渡すのに用いられます。これらの属性は好きなように使えます。たとえば、ログメッセージの一部にすることもできます。以下の例を見てください:

```
FORMAT = "%(asctime)-15s %(clientip)s %(user)-8s %(message)s"
logging.basicConfig(format=FORMAT)
d = { 'clientip' : '192.168.0.1', 'user' : 'fbloggs' }
logger = logging.getLogger("tcpserver")
logger.warning("Protocol problem: %s", "connection reset", extra=d)
```

出力はこのようになります。

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs  Protocol problem: connection reset
```

extra で渡される辞書のキーはロギングシステムで使われているものとぶつからないようにしなければなりません。(どのキーがロギングシステムで使われているかについての詳細は Formatter のドキュメントを参照してください。)

これらの属性をログメッセージに使うことにしたなら、少し注意が必要です。上の例では、'clientip' と 'user' が LogRecord の属性辞書に含まれていることを期待した書式化文字列で Formatter はセットアップされています。これらの属性が欠けていると、書式化例外が発生してしまうためメッセー

ジはログに残りません。したがってこの場合、常にこれらのキーがある *extra* 辞書を渡す必要があります。

このようなことは煩わしいかもしれませんが、この機能は限定された場面で使われるように意図しているものなのです。たとえば同じコードがいくつものコンテキストで実行されるマルチスレッドのサーバで、興味のある条件が現れるのがそのコンテキストに依存している (上の例で言えば、リモートのクライアント IP アドレスや認証されたユーザ名など)、というような場合です。そういった場面では、それ用の `Formatter` が特定の `Handler` と共に使われるというのはよくあることです。

2.5 で変更された仕様: *extra* が追加されました

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*]])
整数で表したレベル *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, func, extra*)

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ライドできるファクトリメソッドです。 2.5 で変更された仕様: *func* と *extra* が追加されました

14.5.2 基本的な使い方

2.4 で変更された仕様: 以前は basicConfig はキーワード引数をとりませんでした

logging パッケージには高い柔軟性があり、その設定にたじろぐこともあるでしょう。そこでこの節では、 logging パッケージを簡単に使う方法もあることを示します。

以下の最も単純な例では、コンソールにログを表示します:

```
import logging

logging.debug('A debug message')
logging.info('Some information')
logging.warning('A shot across the bows')
```

上のスクリプトを実行すると、以下のようなメッセージを目にするでしょう:

```
WARNING:root:A shot across the bows
```

ここではロガーを特定しなかったので、システムはルートロガーを使っています。デバッグメッセージや情報メッセージは表示されませんが、これはデフォルトのルートロガーが WARNING 以上の重要度を持つメッセージしか処理しないように設定されているからです。メッセージの書式もデフォルトの設定に従っています。出力先は sys.stderr で、これもデフォルトの設定です。重要度レベルやメッセージの形式、ログの出力先は、以下の例のように簡単に変更できます:

```
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s %(levelname)s %(message)s',
                    filename='/tmp/myapp.log',
                    filemode='w')
logging.debug('A debug message')
logging.info('Some information')
logging.warning('A shot across the bows')
```

ここでは、basicConfig() メソッドを使って、以下のような出力例になる (そして/tmp/myapp.log に書き込まれる) ように、デフォルト設定を変更しています:

```
2004-07-02 13:00:08,743 DEBUG A debug message
2004-07-02 13:00:08,743 INFO Some information
2004-07-02 13:00:08,743 WARNING A shot across the bows
```

今度は、重要度が DEBUG か、それ以上のメッセージが処理されました。メッセージの形式も変更され、出力はコンソールではなく特定のファイルに書き出されました。

出力の書式化には、通常の Python 文字列に対する初期化を使います - 3.6.2 節を参照してください。書式化文字列は、以下の指定子 (specifier) を常にとります。指定子の完全なリストについては Formatter の

ドキュメントを参照してください。

書式	説明
%(name)s	ロガーの名前 (ログチャネル) の名前です。
%(levelname)s	メッセージのログレベル ('DEBUG','INFO','WARNING','ERROR','CRITICAL') です。
%(asctime)s	LogRecord が生成された際の時刻を、人間が読み取れる形式にしたものです。デフォルトでは、"2
%(message)s	ログメッセージです。

以下のように、追加のキーワードパラメタ *datefmt* を渡すと日付や時刻の書式を変更できます:

```
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s %(levelname)-8s %(message)s',
                    datefmt='%a, %d %b %Y %H:%M:%S',
                    filename='/temp/myapp.log',
                    filemode='w')
logging.debug('A debug message')
logging.info('Some information')
logging.warning('A shot across the bows')
```

出力は以下のようになります:

```
Fri, 02 Jul 2004 13:06:18 DEBUG    A debug message
Fri, 02 Jul 2004 13:06:18 INFO     Some information
Fri, 02 Jul 2004 13:06:18 WARNING  A shot across the bows
```

日付を書式化する文字列は、strftime() の要求に従います - time モジュールを参照してください。

コンソールやファイルではなく、別個に作成しておいたファイル類似オブジェクトにログを出力したい場合には、basicConfig() に *stream* キーワード引数で渡します。*stream* と *filename* の両方の引数を指定した場合、*stream* は無視されるので注意してください。

状況に応じて変化する情報ももちろんログ出力できます。以下のように、単にメッセージを書式化文字列にして、その後ろに可変情報の引数を渡すだけです:

```
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s %(levelname)-8s %(message)s',
                    datefmt='%a, %d %b %Y %H:%M:%S',
                    filename='/temp/myapp.log',
                    filemode='w')
logging.error('Pack my box with %d dozen %s', 5, 'liquor jugs')
```

出力は以下のようになります:

```
Wed, 21 Jul 2004 15:35:16 ERROR    Pack my box with 5 dozen liquor jugs
```

14.5.3 複数の出力先にログを出力する

コンソールとファイルに、別々のメッセージ書式で、別々の状況に応じたログ出力を行わせたいとしましょう。例えばDEBUGよりも高いレベルのメッセージはファイルに記録し、INFO以上のレベルのメッセージはコンソールに出力したいという場合です。また、ファイルにはタイムスタンプを記録し、コンソールには出力しないとします。以下のようにすれば、こうした挙動を実現できます:

```
import logging

# set up logging to file - see previous section for more details
logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s %(name)-12s %(levelname)-8s %(message)s',
                    datefmt='%m-%d %H:%M',
                    filename='/temp/myapp.log',
                    filemode='w')
# define a Handler which writes INFO messages or higher to the sys.stderr
console = logging.StreamHandler()
console.setLevel(logging.INFO)
# set a format which is simpler for console use
formatter = logging.Formatter('%(name)-12s: %(levelname)-8s %(message)s')
# tell the handler to use this format
console.setFormatter(formatter)
# add the handler to the root logger
logging.getLogger('').addHandler(console)

# Now, we can log to the root logger, or any other logger. First the root...
logging.info('Jackdaws love my big sphinx of quartz.')

# Now, define a couple of other loggers which might represent areas in your
# application:

logger1 = logging.getLogger('myapp.area1')
logger2 = logging.getLogger('myapp.area2')

logger1.debug('Quick zephyrs blow, vexing daft Jim.')
logger1.info('How quickly daft jumping zebras vex.')
logger2.warning('Jail zesty vixen who grabbed pay from quack.')
logger2.error('The five boxing wizards jump quickly.')
```

このスクリプトを実行すると、コンソールには以下のように表示されるでしょう:

```
root        : INFO     Jackdaws love my big sphinx of quartz.
myapp.area1 : INFO     How quickly daft jumping zebras vex.
myapp.area2 : WARNING  Jail zesty vixen who grabbed pay from quack.
myapp.area2 : ERROR    The five boxing wizards jump quickly.
```

そして、ファイルは以下のようになるはずです:

```
10-22 22:19 root         INFO     Jackdaws love my big sphinx of quartz.
10-22 22:19 myapp.area1  DEBUG    Quick zephyrs blow, vexing daft Jim.
10-22 22:19 myapp.area1  INFO     How quickly daft jumping zebras vex.
10-22 22:19 myapp.area2  WARNING  Jail zesty vixen who grabbed pay from quack.
10-22 22:19 myapp.area2  ERROR    The five boxing wizards jump quickly.
```

ご覧のように、DEBUGメッセージはファイルだけに出力され、その他のメッセージは両方に出力されます。

この例題では、コンソールとファイルのハンドラだけを使っていますが、実際には任意の数のハンドラや組み合わせを使えます。

14.5.4 ログイベントをネットワーク越しに送受信する

ログイベントをネットワーク越しに送信し、受信端でそれを処理したいとしましょう。SocketHandler インスタンスを送信端のルートロガーに接続すれば、簡単に実現できます:

```
import logging, logging.handlers

rootLogger = logging.getLogger('')
rootLogger.setLevel(logging.DEBUG)
socketHandler = logging.handlers.SocketHandler('localhost',
                    logging.handlers.DEFAULT_TCP_LOGGING_PORT)
# don't bother with a formatter, since a socket handler sends the event as
# an unformatted pickle
rootLogger.addHandler(socketHandler)

# Now, we can log to the root logger, or any other logger. First the root...
logging.info('Jackdaws love my big sphinx of quartz.')

# Now, define a couple of other loggers which might represent areas in your
# application:

logger1 = logging.getLogger('myapp.area1')
logger2 = logging.getLogger('myapp.area2')

logger1.debug('Quick zephyrs blow, vexing daft Jim.')
logger1.info('How quickly daft jumping zebras vex.')
logger2.warning('Jail zesty vixen who grabbed pay from quack.')
logger2.error('The five boxing wizards jump quickly.')
```

受信端では、SocketServer モジュールを使って受信プログラムを作成しておきます。簡単な実用プログラムを以下に示します:

```
import cPickle
import logging
import logging.handlers
import SocketServer
import struct


class LogRecordStreamHandler(SocketServer.StreamRequestHandler):
    """Handler for a streaming logging request.

    This basically logs the record using whatever logging policy is
    configured locally.
    """

    def handle(self):
        """
        Handle multiple requests - each expected to be a 4-byte length,
        followed by the LogRecord in pickle format. Logs the record
        according to whatever policy is configured locally.
        """
        while 1:
            chunk = self.connection.recv(4)
            if len(chunk) < 4:
                break
            slen = struct.unpack(">L", chunk)[0]
            chunk = self.connection.recv(slen)
            while len(chunk) < slen:
                chunk = chunk + self.connection.recv(slen - len(chunk))
            obj = self.unPickle(chunk)
            record = logging.makeLogRecord(obj)
            self.handleLogRecord(record)

    def unPickle(self, data):
        return cPickle.loads(data)

    def handleLogRecord(self, record):
        # if a name is specified, we use the named logger rather than the one
        # implied by the record.
        if self.server.logname is not None:
            name = self.server.logname
        else:
            name = record.name
        logger = logging.getLogger(name)
        # N.B. EVERY record gets logged. This is because Logger.handle
        # is normally called AFTER logger-level filtering. If you want
        # to do filtering, do it at the client end to save wasting
        # cycles and network bandwidth!
        logger.handle(record)

class LogRecordSocketReceiver(SocketServer.ThreadingTCPServer):
    """simple TCP socket-based logging receiver suitable for testing.
    """

    allow_reuse_address = 1

    def __init__(self, host='localhost',
                 port=logging.handlers.DEFAULT_TCP_LOGGING_PORT,
                 handler=LogRecordStreamHandler):
        SocketServer.ThreadingTCPServer.__init__(self, (host, port), handler)
        self.abort = 0
        self.timeout = 1
        self.logname = None

    def serve_until_stopped(self):
        import select
        abort = 0
        while not abort:
            rd, wr, ex = select.select([self.socket.fileno()],
                                       [], [],
```

先にサーバを起動しておき、次にクライアントを起動します。クライアント側では、コンソールには何も出力されません; サーバ側では以下のようなメッセージを目にするはずです:

```
About to start TCP server...
   59 root            INFO     Jackdaws love my big sphinx of quartz.
   59 myapp.area1     DEBUG    Quick zephyrs blow, vexing daft Jim.
   69 myapp.area1     INFO     How quickly daft jumping zebras vex.
   69 myapp.area2     WARNING  Jail zesty vixen who grabbed pay from quack.
   69 myapp.area2     ERROR    The five boxing wizards jump quickly.
```

14.5.5 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(*record*)

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。*record* には、例外発生時に処理されていたレコードが入ります。

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、`logging` パッケージのコアにありますが、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、もっと正確にいえば、`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、`logging` パッケージのコアにありますが、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、`logging.handlers` モジュールの中にありますが、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 (*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`()

上述のような方法でロールオーバを行います。

`emit`(*record*)

上述のようなロールオーバを行いながら、レコードをファイルに出力します。

TimedRotatingFileHandler

`TimedRotatingFileHandler` クラスは、`logging.handlers` モジュールの中にありますが、特定の時間間隔でのログ交替をサポートしています。

class `TimedRotatingFileHandler` (*filename* [,*when* [,*interval* [,*backupCount*]]])

`TimedRotatingFileHandler` クラスの新たなインスタンスを返します。*filename* に指定したファイルを開き、ログ出力先のストリームとして使います。ログファイルの交替時には、ファイル名に拡張子 (suffix) をつけます。ログファイルの交替は *when* および *interval* の積に基づいて行います。

when は *interval* の単位を指定するために使います。使える値は下表の通りで、大小文字の区別を行いません:

値	*interval* の単位
S	秒
M	分
H	時間
D	日
W	曜日 (0=Monday)
midnight	深夜

backupCount がゼロでない場合、古いログファイルを保存する際にロギングシステムは拡張子を付けます。拡張子は日付と時間に基づいて、strftime の `%Y-%m-%d_%H-%M-%S` 形式かその前の方の一部分を、ロールオーバ間隔に依存した形で使います。保存されるファイル数は高々 *backupCount* 個で、それ以上のファイルがロールオーバされる時に作られるならば、一番古いものが削除されます。

doRollover()
上記の方法でロールオーバを行います。

emit(*record*)
`setRollover()` で解説した方法でロールオーバを行いながら、レコードをファイルに出力します。

SocketHandler

`SocketHandler` クラスは、`logging.handlers` モジュールの中にありますが、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、`logging.handlers` モジュールの中にありますが、`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 クラスは、logging.handlers モジュールの中にありますが、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 クラスは、logging.handlers モジュールの中にありますが、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` クラスは、`logging.handlers` モジュールの中にありますが、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ライドしてください。

MemoryHandler

`MemoryHandler` は、`logging.handlers` モジュールの中にありますが、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` クラスは、`logging.handlers` モジュールの中にありますが、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。*host* は特別なポートを使うことが必要な場合には、`host:port` の形式で使うこともできます。*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。

14.5.6 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、`'%(message)s'` 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 3.6.2 "String Formatting Operations" の章を参照してください。

現状では、 `LogRecord` の有用な属性は以下のようになっています:

Format	Description
%(name)s	ロガー (ログ記録チャネル) の名前
%(levelno)s	メッセージのログ記録レベルを表す数字 (DEBUG, INFO, WARNING, ERROR, CRITICAL)
%(levelname)s	メッセージのログ記録レベルを表す文字列 ("DEBUG", "INFO", "WARNING", "ERROR", "CRITICA
%(pathname)s	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)
%(filename)s	パス名中のファイル名部分
%(module)s	モジュール名 (ファイル名の名前部分)
%(funcName)s	ログ記録の呼び出しを含む関数の名前
%(lineno)d	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)
%(created)f	LogRecord が生成された時刻 (time.time() の返した値)
%(asctime)s	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:
%(msecs)d	LogRecord が生成された時刻の、ミリ秒部分
%(thread)d	スレッド ID (取得できる場合)
%(threadName)s	スレッド名 (取得できる場合)
%(process)d	プロセス ID (取得できる場合)
%(message)s	レコードが発信された際に処理された msg %args の結果

2.5 で変更された仕様: *funcName* が追加されました

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のようになります: *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。

14.5.7 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。

14.5.8 LogRecord オブジェクト

何かをログ記録する際には常に `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係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。

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。

`getMessage` ()

ユーザが供給した引数をメッセージに交ぜた後、この `LogRecord` インスタンスへのメッセージを返します。

14.5.9 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) なようになっています。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。

14.5.10 環境設定

環境設定のための関数

以下の関数で `logging` モジュールの環境設定をします。これらの関数は、`logging.config` にあります。これらの関数の使用はオプションです — `logging` モジュールはこれらの関数を使うか、(`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

ようにできます。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、適切な状況で 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。設定を送るには、まず設定ファイルを読み、それを 4 バイトからなる長さを struct.pack('>L', n) を使ってバイナリにパックしたものを前に付けたバイト列としてソケットに送ります。

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で指定しています。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はなりません。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ネル名を階層的に表したもの、すなわち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はならず、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か、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=handlers.NTEventLogHandler
level=CRITICAL
formatter=form06
args=('Python Application', '', 'Application')

[handler_hand07]
class=handlers.SMTPHandler
level=WARN
formatter=form07
args=('localhost', 'from@abc', ['user1@abc', 'user2@xyz'], 'Logger Subject')

[handler_hand08]
class=handlers.MemoryHandler
level=NOTSET
formatter=form08
target=
args=(10, ERROR)

[handler_hand09]
class=handlers.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=
class=logging.Formatter
```

`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。

`class` エントリはオプションです。`class` はフォーマッタのクラス名 (ドット区切りのモジュールとクラス名として) を示します。このオプションは `Formatter` のサブクラスをインスタンス化するのに有用です。`Formatter` のサブクラスは例外トレースバックを展開された形式または圧縮された形式で表現することができます。

14.6 `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*[, *stream*]])

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。UNIX ではプロンプトはファイルに似たオブジェクト *stream* へ出力されます。デフォルトは `sys.stdout` です (この引数は Windows では無視されます。)。

利用できるシステム: Macintosh, Unix, Windows 2.5 で変更された仕様: パラメータ *stream* の追加

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。

14.7 `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 (14.10 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 (14.11 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 (14.8 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 (14.9 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。

14.7.1 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たり、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たりしていました。

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、文字を一文字一文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。この手順のもた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、cud1、cuu1、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: BUTTON*n*_PRESSED, BUTTON*n*_RELEASED, BUTTON*n*_CLICKED, BUTTON*n*_DOUBLE_CLICKED, BUTTON*n*_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められています。

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められています。

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しまっている) 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る制限をうけず、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えます。(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()

newlime 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()

noquiflush 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 `1` 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()` を呼ぶのとと同じ効果です。*flag* が真か、引数が与えられていない場合、制御文字が読み出された最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 入力関数に一文字づつ渡されます。

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()` を呼んだ直後に呼ぶようにしておくとよいでしょう。

`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。入力が得られ、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。

use_default_colors()
この機能をサポートしている端末上で、色の値としてデフォルト値を使う設定をします。あなたのアプリケーションで透過性とサポートするためにこの関数を使ってください。デフォルトの色は 色番号-1 に割り当てられます。

この関数を呼んだ後、たとえば `init_pair(x, curses.COLOR_RED, -1)` は色ペア *x* を赤い前景色とデフォルトの背景色に初期化します。

14.7.2 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され (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されます。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
ls	左側	ACS_VLINE
rs	右側	ACS_VLINE
ts	上側	ACS_HLINE
bs	下側	ACS_HLINE
tl	左上の角	ACS_ULCORNER
tr	右上の角	ACS_URCORNER
bl	左下の角	ACS_LLCORNER
br	右下の角	ACS_LRCORNER

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。

clrtobot()

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(*[y, x]*)

(*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(*[y, x]*)

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、キーパッド上のキー等は 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 ([*y, x*])
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、キーバット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 ([*y, x*])
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 ([*y, x*])
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 行づつ下に移動します。

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で残されます。

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、数ミリ秒間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で残されます。

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に *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。

14.7.3 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
A_ALTCHARSET	代用文字 (alternate character) モード。
A_BLINK	点滅モード。
A_BOLD	太字モード。
A_DIM	低輝度モード。
A_NORMAL	通常の属性。
A_STANDOUT	強調モード。
A_UNDERLINE	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
KEY_MIN	最小のキー値
KEY_BREAK	ブレーク (Break, 信頼できません)
KEY_DOWN	下向き矢印 (Down-arrow)
KEY_UP	上向き矢印 (Up-arrow)
KEY_LEFT	左向き矢印 (Left-arrow)
KEY_RIGHT	右向き矢印 (Right-arrow)
KEY_HOME	ホームキー (Home, または上左矢印)
KEY_BACKSPACE	バックスペース (Backspace, 信頼できません)
KEY_F0	ファンクションキー 64 個までサポートされています。
KEY_F*n*	ファンクションキー *n* の値
KEY_DL	行削除 (Delete line)
KEY_IL	行挿入 (Insert line)
KEY_DC	文字削除 (Delete char)
KEY_IC	文字挿入、または文字挿入モードへ入る
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

キー定数	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)
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

キー定数	キー
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) が利用できると考えてよいでしょう; 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 コード	意味
ACS_CKBOARD	チェッカーボードパタン (点描)
ACS_DARROW	下向き矢印
ACS_DEGREE	度
ACS_DIAMOND	ダイアモンド
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_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

14.8 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。

14.8.1 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められるかどうかは 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	カーソルを上に移動します; 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められるかどうかは 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。

14.9 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() は例外を再送出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パ関数です。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。

14.10 `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' のニーモニック符号名からなる 33 要素の文字列によるシーケンスです。

14.11 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。

14.11.1 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。返されたパネルオブジェクトを参照しておく必要があることに注意してください。もし参照しなければ、パネルオブジェクトはガベージコレクションされてパネルスタックから削除されます。

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。

14.11.2 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。

14.12 platform — 実行中プラットフォームの固有情報を参照する

2.3 で追加された仕様です。

注意: プラットフォーム毎にアルファベット順に並べています。Linux については UNIX セクションを参照してください。

14.12.1 クロス プラットフォーム

architecture(*executable=sys.executable, bits='', linkage=''*)
executable で指定した実行可能ファイル（省略時は Python インタープリタのバイナリ）の各種アーキテクチャ情報を調べます。

戻り値はタプル (bits, linkage) で、アーキテクチャのビット数と実行可能ファイルのリンク形式を示します。どちらの値も文字列で返ります。

値が不明な場合は、パラメータで指定した値が返ります。*bits* を'' と指定した場合、ビット数として sizeof(pointer) が返ります。(Python のバージョンが 1.5.2 以下の場合は、サポートされているポインタサイズとして sizeof(long) を使用します。)

この関数は、システムの 'file' コマンドを使用します。'file' はほとんどの UNIX プラットフォームと一部の非 UNIX プラットフォームで利用可能ですが、'file' コマンドが利用できず、かつ *executable* が Python インタープリタでない場合には適切なデフォルト値が返ります。

machine()
'i386' のような、機種を返します。不明な場合は空文字列を返します。

node()
コンピュータのネットワーク名を返します。ネットワーク名は完全修飾名とは限りません。不明な場合は空文字列を返します。

platform(*aliased=0, terse=0*)
実行中プラットフォームを識別する文字列を返します。この文字列には、有益な情報をできるだけ多

く付加しています。

戻り値は機械で処理しやすい形式ではなく、**人間にとって読みやすい**形式となっています。異なったプラットフォームでは異なった戻り値となるようになっています。

aliased が真なら、システムの名称として一般的な名称ではなく、別名を使用して結果を返します。たとえば、SunOS は Solaris となります。この機能は `system_alias()` で実装されています。

terse が真なら、プラットフォームを特定するために最低限必要な情報だけを返します。

processor()
`'amdk6'` のような、（現実の）プロセッサ名を返します。

不明な場合は空文字列を返します。NetBSD のようにこの情報を提供しない、または `machine()` と同じ値しか返さないプラットフォームも多く存在しますので、注意してください。

python_build()
Python のビルド番号と日付を、(*buildno*, *builddate*) のタプルで返します。

python_compiler()
Python をコンパイルする際に使用したコンパイラを示す文字列を返します。

python_version()
Python のバージョンを、`'major.minor.patchlevel'` 形式の文字列で返します。

`sys.version` と異なり、patchlevel（デフォルトでは 0) も必ず含まれています。

python_version_tuple()
Python のバージョンを、文字列のタプル (*major*, *minor*, *patchlevel*) で返します。

`sys.version` と異なり、patchlevel（デフォルトでは `0`) も必ず含まれています。

release()
`'2.2.0'` や `'NT'` のような、システムのリリース情報を返します。不明な場合は空文字列を返します。

system()
`'Linux'`,`'Windows'`,`'Java'` のような、システム/OS 名を返します。不明な場合は空文字列を返します。

system_alias(*system, release, version*)
マーケティング目的で使われる一般的な別名に変換して (*system*, *release*, *version*) を返します。混乱を避けるために、情報を並べなおす場合があります。

version()
`'#3 on degas'` のような、システムのリリース情報を返します。不明な場合は空文字列を返します。

uname()
非常に可搬性の高い uname インターフェースで、実行中プラットフォームを示す情報を、文字列のタプル (*system*, *node*, *release*, *version*, *machine*, *processor*) で返します。

`os.uname()` と異なり、複数のプロセッサ名が候補としてタプルに追加される場合があります。

不明な項目は ″ となります。

14.12.2 Java プラットフォーム

java_ver(*release=”, vendor=”, vminfo=(”,”,”), osinfo=(”,”,”)*)
Jython 用のバージョンインターフェースで、タプル (*release*, *vendor*, *vminfo*, *osinfo*) を返します。*vminfo* はタプル (*vm_name*, *vm_release*, *vm_vendor*)、*osinfo* はタプル (*os_name*, *os_version*, *os_arch*) です。不明な項目は引数で指定した値（デフォルトは″）となります。

14.12.3 Windows プラットフォーム

win32_ver(*release=”, version=”, csd=”, ptype=”*)

Windows のレジストリからバージョン情報を取得し、バージョン番号/CSD レベル/OS タイプ（シングルプロセッサ又はマルチプロセッサ）をタプル（*version*, *csd*, *ptype*）で返します。

参考：*ptype* はシングルプロセッサの NT 上では'Uniprocessor Free'、マルチプロセッサでは'Multiprocessor Free' となります。*'Free'* がついている場合はデバッグ用のコードが含まれていないことを示し、*'Checked'* がついていれば引数や範囲のチェックなどのデバッグ用コードが含まれていることを示します。

注意: この関数は、Mark Hammond の win32all がインストールされた Win32 互換プラットフォームでのみ利用可能です。

Win95/98 固有

popen(*cmd, mode='r', bufsize=None*)

可搬性の高い popen() インターフェースで、可能なら win32pipe.popen() を使用します。win32pipe.popen() は Windows NT では利用可能ですが、Windows 9x ではハングしてしまいます。

14.12.4 Mac OS プラットフォーム

mac_ver(*release=”, versioninfo=(”,”,”), machine=”*)

Mac OS のバージョン情報を、タプル（*release*, *versioninfo*, *machine*）で返します。*versioninfo* は、タプル（*version*, *dev_stage*, *non_release_version*）です。

不明な項目は” となります。タプルの要素は全て文字列です。

この関数で使用している gestalt() API については、http://www.rgaros.nl/gestalt/ を参照してください。

14.12.5 UNIX プラットフォーム

dist(*distname=”, version=”, id=”, supported_dists=('SuSE','debian','redhat','mandrake')*)

OS ディストリビューション名の取得を試みます。戻り値はタプル（*distname*, *version*, *id*）で、不明な項目は引数で指定した値となります。

libc_ver(*executable=sys.executable, lib=”, version=”, chunksize=2048*)

executable で指定したファイル（省略時は Python インタープリタ）がリンクしている libc バージョンの取得を試みます。戻り値は文字列のタプル（*lib*, *version*）で、不明な項目は引数で指定した値となります。

この関数は、実行形式に追加されるシンボルの細かな違いによって、libc のバージョンを特定します。この違いは **gcc** でコンパイルされた実行可能ファイルでのみ有効だと思われます。

chunksize にはファイルから情報を取得するために読み込むバイト数を指定します。

14.13 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。

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)

EWOULDBLOCK
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)

EBADSLT
無効なスロットです (Invalid slot)

EDEADLOCK
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)

EBFONT
フォントファイル形式が間違っています (Bad font file format)

ENOSTR
ストリーム型でないデバイスです (Device not a stream)

ENODATA
利用可能なデータがありません (No data available)

ETIME
時間切れです (Timer expired)

ENOSR
streams リソースを使い切りました (Out of streams resources)

ENONET
計算機はネットワーク上にありません (Machine is not on the network)

ENOPKG
パッケージがインストールされていません (Package not installed)

EREMOTE
対象物は遠隔にあります (Object is remote)

ENOLINK
リンクが切られました (Link has been severed)

EADV
Advertise エラーです (Advertise error)

ESRMNT
Srmount エラーです (Srmount error)

ECOMM
送信時の通信エラーです (Communication error on send)

EPROTO
プロトコルエラーです (Protocol error)

EMULTIHOP
多重ホップを試みました (Multihop attempted)

EDOTDOT
RFS 特有のエラーです (RFS specific error)

EBADMSG
データメッセージではありません (Not a data message)

EOVERFLOW
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)

ENOTUNIQ
名前がネットワーク上で一意でありません (Name not unique on network)

EBADFD
ファイル記述子の状態が不正です (File descriptor in bad state)

EREMCHG
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)

ENOPROTOOPT
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
ディスククオータを超えました (Quota exceeded)

14.14 ctypes — Python のための外部関数ライブラリ。

2.5 で追加された仕様です。

ctypes は Python のための外部関数ライブラリです。このライブラリは C と互換性のあるデータ型を提供し、動的リンク/共有ライブラリ内の関数呼び出しを可能にします。動的リンク/共有ライブラリを純粋な Python でラップするために使うことができます。

14.14.1 ctypes チュートリアル

注意: このチュートリアルのコードサンプルは動作確認のために doctest を使います。コードサンプルの中には Linux、Windows、あるいは Mac OS X 上で異なる動作をするものがあるため、サンプルのコメントに doctest 命令を入れてあります。

注意: かなりのコードサンプルで ctypes の c_int 型を参照しています。32 ビットシステムにおいてこの型は c_long 型のエイリアスです。そのため、c_int 型を想定しているときに c_long が表示されたとしても、混乱しないようにしてください — 実際には同じ型なのです。

動的リンクライブラリをロードする

動的リンクライブラリをロードするために、ctypes は *cdll* をエクスポートします。Windows ではさらに *windll* と *oledll* オブジェクトもエクスポートします。

これらのオブジェクトの属性としてライブラリにアクセスすることでライブラリをロードします。*cdll* は標準 cdecl 呼び出し規約を用いて関数をエクスポートしているライブラリをロードします。それに対して、*windll* ライブラリは stdcall 呼び出し規約を用いる関数を呼び出します。*oledll* も stdcall 呼び出し規約を使いますが、関数が Windows HRESULT エラーコードを返すことを想定しています。このエラーコードは関数呼び出しが失敗したとき、WindowsError Python 例外を自動的に発生させるために使われます。

Windows 用の例ですが、msvcrt はほとんどの標準 C 関数が含まれている MS 標準 C ライブラリであり、cdecl 呼び出し規約を使うことに注意してください:

```
>>> from ctypes import *
>>> print windll.kernel32 # doctest: +WINDOWS
<WinDLL 'kernel32', handle ... at ...>
>>> print cdll.msvcrt # doctest: +WINDOWS
<CDLL 'msvcrt', handle ... at ...>
>>> libc = cdll.msvcrt # doctest: +WINDOWS
>>>
```

Windows ではいつもの'.dll' ファイル拡張子を自動的に追加します。

Linux ではライブラリをロードするために拡張子を**含む**ファイル名を指定する必要があるので、属性アクセスは動作しません。dll ローダーの LoadLibrary メソッドを使うか、コンストラクタを呼び出して CDLL のインスタンスを作ることでライブラリをロードするかのどちらかを行わなければなりません:

```
>>> cdll.LoadLibrary("libc.so.6") # doctest: +LINUX
<CDLL 'libc.so.6', handle ... at ...>
>>> libc = CDLL("libc.so.6")     # doctest: +LINUX
>>> libc                         # doctest: +LINUX
<CDLL 'libc.so.6', handle ... at ...>
>>>
```

ロードした dll から関数にアクセスする

dll オブジェクトの属性として関数にアクセスします:

```
>>> from ctypes import *
>>> libc.printf
<_FuncPtr object at 0x...>
>>> print windll.kernel32.GetModuleHandleA # doctest: +WINDOWS
<_FuncPtr object at 0x...>
>>> print windll.kernel32.MyOwnFunction # doctest: +WINDOWS
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "ctypes.py", line 239, in __getattr__
    func = _StdcallFuncPtr(name, self)
AttributeError: function 'MyOwnFunction' not found
>>>
```

`kernel32` や `user32` のような win32 システム dll は、多くの場合関数の UNICODE バージョンに加えて ANSI バージョンもエクスポートすることに注意してください。UNICODE バージョンは後ろに `W` が付いた名前でエクスポートされ、ANSI バージョンは `A` が付いた名前でエクスポートされます。与えられたモジュールの**モジュールハンドル**を返す win32 `GetModuleHandle` 関数は次のような C プロトタイプを持ちます。UNICODE バージョンが定義されているかどうかにより `GetModuleHandle` としてどちらか一つを公開するためにマクロが使われます:

```
/* ANSI version */
HMODULE GetModuleHandleA(LPCSTR lpModuleName);
/* UNICODE version */
HMODULE GetModuleHandleW(LPCWSTR lpModuleName);
```

windll は魔法を使ってどちらか一つを選ぼうとはしません。`GetModuleHandleA` もしくは `GetModuleHandleW` を明示的に指定して必要とするバージョンにアクセスし、通常の文字列かユニコード文字列を使ってそれぞれ呼び出さなければなりません。

時には、dll が関数を`"???2@YAPAXI@Z"`のような Python 識別子として有効でない名前でエクスポートすることがあります。このような場合に関数を取り出すには、`getattr` を使わなければなりません。

```
>>> getattr(cdll.msvcrt, "???2@YAPAXI@Z") # doctest: +WINDOWS
<_FuncPtr object at 0x...>
>>>
```

Windows では、名前ではなく序数によって関数をエクスポートする dll もあります。こうした関数には序数を使って dll オブジェクトにインデックス指定することでアクセスします:

```
>>> cdll.kernel32[1] # doctest: +WINDOWS
<_FuncPtr object at 0x...>
>>> cdll.kernel32[0] # doctest: +WINDOWS
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "ctypes.py", line 310, in __getitem__
    func = _StdcallFuncPtr(name, self)
AttributeError: function ordinal 0 not found
>>>
```

関数を呼び出す

これらの関数は他の Python 呼び出し可能オブジェクトと同じように呼び出すことができます。この例では `time()` 関数（UNIX エポックからのシステム時間を秒単位で返す）と、`GetModuleHandleA()` 関数（win32 モジュールハンドルを返す）を使います。

この例は両方の関数を NULL ポインタとともに呼び出します (`None` を NULL ポインタとして使う必要があります):

```
>>> print libc.time(None) # doctest: +SKIP
1150640792
>>> print hex(windll.kernel32.GetModuleHandleA(None)) # doctest: +WINDOWS
0x1d000000
>>>
```

`ctypes` は引数の数を間違えたり、あるいは呼び出し規約を間違えた関数呼び出しからあなたを守ろうとします。残念ながら、これは Windows でしか機能しません。関数が返った後にスタックを調べることでこれを行います。したがって、エラーは発生しますが、その関数は呼び出された**後です**:

```
>>> windll.kernel32.GetModuleHandleA() # doctest: +WINDOWS
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: Procedure probably called with not enough arguments (4 bytes missing)
>>> windll.kernel32.GetModuleHandleA(0, 0) # doctest: +WINDOWS
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: Procedure probably called with too many arguments (4 bytes in excess)
>>>
```

同じ例外が `cdecl` 呼び出し規約を使って `stdcall` 関数を呼び出したときに発生しますし、逆の場合も同様です。

```
>>> cdll.kernel32.GetModuleHandleA(None) # doctest: +WINDOWS
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: Procedure probably called with not enough arguments (4 bytes missing)
>>>

>>> windll.msvcrt.printf("spam") # doctest: +WINDOWS
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: Procedure probably called with too many arguments (4 bytes in excess)
>>>
```

正しい呼び出し規約を知るためには、呼び出したい関数についての C ヘッダファイルもしくはドキュメントを見なければなりません。

Windows では、関数が無効な引数とともに呼び出された場合の一般保護例外によるクラッシュを防ぐために、`ctypes` は win32 構造化例外処理を使います:

```
>>> windll.kernel32.GetModuleHandleA(32) # doctest: +WINDOWS
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
WindowsError: exception: access violation reading 0x00000020
>>>
```

しかし、ctypes を使って Python をクラッシュさせる方法は十分なほどあるので、よく注意すべきです。

None、整数、長整数、バイト文字列およびユニコード文字列だけが、こうした関数呼び出しにおいてパラメータとして直接使えるネイティブの Python オブジェクトです。None は C の NULL ポインタとして渡され、バイト文字列とユニコード文字列はそのデータを含むメモリブロックへのポインタ (char * または wchar_t *) として渡されます。Python 整数と Python 長整数はプラットホームのデフォルトの C int 型として渡され、その値は C int 型に合うようにマスクされます。

他のパラメータ型をもつ関数呼び出しに移る前に、ctypes データ型についてさらに学ぶ必要があります。

基本のデータ型

ctypes はたくさんの C と互換性のあるデータ型を定義しています :

ctypes の型	C の型	Python の型
c_char	char	1 文字の文字列
c_wchar	wchar_t	1 文字のユニコード文字列
c_byte	char	整数/長整数
c_ubyte	unsigned char	整数/長整数
c_short	short	整数/長整数
c_ushort	unsigned short	整数/長整数
c_int	int	整数/長整数
c_uint	unsigned int	整数/長整数
c_long	long	整数/長整数
c_ulong	unsigned long	整数/長整数
c_longlong	__int64 or long long	整数/長整数
c_ulonglong	unsigned __int64 or unsigned long long	整数/長整数
c_float	float	浮動小数点数
c_double	double	浮動小数点数
c_char_p	char * (NUL 終端)	文字列または None
c_wchar_p	wchar_t * (NUL 終端)	ユニコードまたは None
c_void_p	void *	整数/長整数または None

これら全ての型はその型を呼び出すことによって作成でき、オプションとして型と値が合っている初期化子を指定することができます:

```
>>> c_int()
c_long(0)
>>> c_char_p("Hello, World")
c_char_p('Hello, World')
>>> c_ushort(-3)
c_ushort(65533)
>>>
```

これらの型は変更可能であり、値を後で変更することもできます:

```
>>> i = c_int(42)
>>> print i
c_long(42)
>>> print i.value
42
>>> i.value = -99
>>> print i.value
-99
>>>
```

新しい値をポインタ型 c_char_p、c_wchar_p、および c_void_p のインスタンスへ代入すると、メモリブロックの**内容ではなく**指している**メモリ位置**が変わります、(もちろんできません。なぜなら、Python 文字列は変更不可能だからです):

```
>>> s = "Hello, World"
>>> c_s = c_char_p(s)
>>> print c_s
c_char_p('Hello, World')
>>> c_s.value = "Hi, there"
>>> print c_s
c_char_p('Hi, there')
>>> print s                 # 最初の文字列は変更されていない
Hello, World
>>>
```

しかし、変更可能なメモリを指すポインタであることを想定している関数へそれらを渡さないように注意すべきです。もし変更可能なメモリブロックが必要なら、ctypes には create_string_buffer 関数があり、いろいろな方法で作成することできます。現在のメモリブロックの内容は raw プロパティを使ってアクセス (あるいは変更) することができます。もし現在のメモリブロックに NUL 終端文字列としてアクセスしたいなら、value プロパティを使ってください:

```
>>> from ctypes import *
>>> p = create_string_buffer(3)       # 3 バイトのバッファを作成、NUL で初期化される
>>> print sizeof(p), repr(p.raw)
3 '\x00\x00\x00'
>>> p = create_string_buffer("Hello")       # NUL 終端文字列を含むバッファを作成
>>> print sizeof(p), repr(p.raw)
6 'Hello\x00'
>>> print repr(p.value)
'Hello'
>>> p = create_string_buffer("Hello", 10)  # 10 バイトのバッファを作成
>>> print sizeof(p), repr(p.raw)
10 'Hello\x00\x00\x00\x00\x00'
>>> p.value = "Hi"
>>> print sizeof(p), repr(p.raw)
10 'Hi\x00lo\x00\x00\x00\x00\x00'
>>>
```

create_string_buffer 関数は初期の ctypes リリースにあった c_string 関数だけでなく、(エイリアスとしてはまだ利用できる)c_buffer 関数をも置き換えるものです。C の型 wchar_t のユニコード文字を含む変更可能なメモリブロックを作成するには、create_unicode_buffer 関数を使ってください。

続・関数を呼び出す

printf は sys.stdout では**なく**、本物の標準出力チャンネルへプリントすることに注意してください。したがって、これらの例はコンソールプロンプトでのみ動作し、*IDLE* や *PythonWin* では動作しません:

```
>>> printf = libc.printf
>>> printf("Hello, %s\n", "World!")
Hello, World!
14
>>> printf("Hello, %S", u"World!")
Hello, World!
13
>>> printf("%d bottles of beer\n", 42)
42 bottles of beer
19
>>> printf("%f bottles of beer\n", 42.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ArgumentError: argument 2: exceptions.TypeError: Don't know how to convert parameter 2
>>>
```

前に述べたように、必要な C のデータ型へ変換できるようにするためには、整数、文字列およびユニコード文字列を除くすべての Python 型を対応する ctypes 型でラップしなければなりません:

```
>>> printf("An int %d, a double %f\n", 1234, c_double(3.14))
Integer 1234, double 3.1400001049
31
>>>
```

自作のデータ型とともに関数を呼び出す

自作のクラスのインスタンスを関数引数として使えるように、ctypes 引数変換をカスタマイズすることもできます。ctypes は_as_parameter_属性を探し出し、関数引数として使います。もちろん、整数、文字列もしくはユニコードの中の一つでなければなりません:

```
>>> class Bottles(object):
...     def __init__(self, number):
...         self._as_parameter_ = number
...
>>> bottles = Bottles(42)
>>> printf("%d bottles of beer\n", bottles)
42 bottles of beer
19
>>>
```

インスタンスのデータを_as_parameter_インスタンス変数の中に入れたくない場合には、そのデータを利用できるようにする property を定義することができます。

要求される引数の型を指定する (関数プロトタイプ)

argtypes 属性を設定することによって、DLL からエクスポートされている関数に要求される引数の型を指定することができます。

argtypes は C データ型のシーケンスでなければなりません (この場合 printf 関数はおそらく良い例

ではありません。なぜなら、引数の数が可変であり、フォーマット文字列に依存した異なる型のパラメータを取るからです。一方では、この機能の実験にはとても便利です):

```
>>> printf.argtypes = [c_char_p, c_char_p, c_int, c_double]
>>> printf("String 'Hi', Int %d, Double %f\n", "Hi", 10, 2.2)
String 'Hi', Int 10, Double 2.200000
37
>>>
```

(C の関数のプロトタイプのように) 書式を指定すると互換性のない引数型になるのを防ぎ、引数を有効な型へ変換しようとします:

```
>>> printf("%d %d %d", 1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ArgumentError: argument 2: exceptions.TypeError: wrong type
>>> printf("%s %d %f", "X", 2, 3)
X 2 3.00000012
12
>>>
```

関数呼び出しへ渡す自作のクラスを定義した場合には、argtypes シーケンスの中で使えるようにするために、そのクラスに from_param クラスメソッドを実装しなければなりません。from_param クラスメソッドは関数呼び出しへ渡された Python オブジェクトを受け取り、型チェックもしくはこのオブジェクトが受け入れ可能であると確かめるために必要なことはすべて行ってから、オブジェクト自身、_as_parameter_属性、あるいは、この場合に C 関数引数として渡したい何かの値を返さなければなりません。繰り返しになりますが、その返される結果は整数、文字列、ユニコード、ctypes インスタンス、あるいは_as_parameter_属性をもつものであるべきです。

戻り値の型

デフォルトでは、関数は C int を返すと仮定されます。他の戻り値の型を指定するには、関数オブジェクトの restype 属性に設定します。

さらに高度な例として、strchr 関数を使います。この関数は文字列ポインタと char を受け取り、文字列へのポインタを返します。

```
>>> strchr = libc.strchr
>>> strchr("abcdef", ord("d")) # doctest: +SKIP
8059983
>>> strchr.restype = c_char_p # c_char_p は文字列へのポインタ
>>> strchr("abcdef", ord("d"))
'def'
>>> print strchr("abcdef", ord("x"))
None
>>>
```

上の ord("x") 呼び出しを避けたいなら、argtypes 属性を設定することができます。二番目の引数が一文字の Python 文字列から C の char へ変換されます:

```
>>> strchr.restype = c_char_p
>>> strchr.argtypes = [c_char_p, c_char]
>>> strchr("abcdef", "d")
'def'
>>> strchr("abcdef", "def")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ArgumentError: argument 2: exceptions.TypeError: one character string expected
>>> print strchr("abcdef", "x")
None
>>> strchr("abcdef", "d")
'def'
>>>
```

外部関数が整数を返す場合は、restype 属性として呼び出し可能な Python オブジェクト (例えば、関数またはクラス) を使うこともできます。呼び出し可能オブジェクトは C 関数が返す integer とともに呼び出され、この呼び出しの結果は関数呼び出しの結果として使われるでしょう。これはエラーの戻り値をチェックして自動的に例外を発生させるために役に立ちます:

```
>>> GetModuleHandle = windll.kernel32.GetModuleHandleA # doctest: +WINDOWS
>>> def ValidHandle(value):
...     if value == 0:
...         raise WinError()
...     return value
...
>>>
>>> GetModuleHandle.restype = ValidHandle # doctest: +WINDOWS
>>> GetModuleHandle(None) # doctest: +WINDOWS
486539264
>>> GetModuleHandle("something silly") # doctest: +WINDOWS
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in ValidHandle
WindowsError: [Errno 126] The specified module could not be found.
>>>
```

WinError はエラーコードの文字列表現を得るために Windows の FormatMessage() api を呼び出し、例外を**返す**関数です。WinError はオプションでエラーコードパラメータを取ります。このパラメータが使われない場合は、エラーコードを取り出すために GetLastError() を呼び出します。

errcheck 属性によってもっと強力なエラーチェック機構を利用できることに注意してください。詳細はリファレンスマニュアルを参照してください。

ポインタを渡す (または、パラメータの参照渡し)

時には、C api 関数がパラメータのデータ型として**ポインタ**を想定していることがあります。おそらくパラメータと同一の場所に書き込むためか、もしくはそのデータが大きすぎて値渡しできない場合です。これは**パラメータの参照渡し**としても知られています。

ctypes は byref 関数をエクスポートしており、パラメータを参照渡しするために使用します。pointer 関数を使っても同じ効果が得られます。しかし、pointer は本当のポインタオブジェクトを構築するためより多くの処理を行うことから、Python 側でポインタオブジェクト自体を必要としないならば byref を使う方がより高速です:

```
>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer('\000' * 32)
>>> print i.value, f.value, repr(s.value)
0 0.0 ''
>>> libc.sscanf("1 3.14 Hello", "%d %f %s",
...             byref(i), byref(f), s)
3
>>> print i.value, f.value, repr(s.value)
1 3.1400001049 'Hello'
>>>
```

構造体と共用体

構造体と共用体は ctypes モジュールに定義されている Structure および Union ベースクラスから導出されなければなりません。それぞれのサブクラスは_fields_属性を定義する必要があります。_fields_は**フィールド名**と**フィールド型**を持つ *2* **要素タプル**のリストでなければなりません。

フィールド型は c_int か他の ctypes 型 (構造体、共用体、配列、ポインタ) から導出された ctypes 型である必要があります。

x と y という名前の二つの整数からなる簡単な POINT 構造体の例です。コンストラクタで構造体の初期化する方法の説明にもなっています:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = [("x", c_int),
...                 ("y", c_int)]
...
>>> point = POINT(10, 20)
>>> print point.x, point.y
10 20
>>> point = POINT(y=5)
>>> print point.x, point.y
0 5
>>> POINT(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: too many initializers
>>>
```

また、さらに複雑な構造体を構成することができます。Structure はそれ自体がフィールド型に構造体を使うことで他の構造体を内部に持つことができます。

upperleft と lowerright という名前の二つの POINT を持つ RECT 構造体です:

```
>>> class RECT(Structure):
...     _fields_ = [("upperleft", POINT),
...                 ("lowerright", POINT)]
...
>>> rc = RECT(point)
>>> print rc.upperleft.x, rc.upperleft.y
0 5
>>> print rc.lowerright.x, rc.lowerright.y
0 0
>>>
```

入れ子になった構造体はいくつかの方法を用いてコンストラクタで初期化することができます:

```
>>> r = RECT(POINT(1, 2), POINT(3, 4))
>>> r = RECT((1, 2), (3, 4))
```

フィールド記述子は**クラス**から取り出せます。デバッグするときに役に立つ情報を得ることができます:

```
>>> print POINT.x
<Field type=c_long, ofs=0, size=4>
>>> print POINT.y
<Field type=c_long, ofs=4, size=4>
>>>
```

構造体/共用体アライメントとバイトオーダー

デフォルトでは、Structure と Union のフィールドは C コンパイラが行うのと同じ方法でアライメントされています。サブクラスを定義するときに_pack_クラス属性を指定することでこの動作を変えることは可能です。このクラス属性には正の整数を設定する必要があり、フィールドの最大アライメントを指定します。これは MSVC で`#pragma pack(n)` が行っていること同じです。

`ctypes` は Structure と Union に対してネイティブのバイトオーダーを使います。ネイティブではないバイトオーダーの構造体を作成するには、BigEndianStructure、LittleEndianStructure、BigEndianUnion および LittleEndianUnion ベースクラスの中の一つを使います。これらのクラスにポインタフィールドを持たせることはできません。

構造体と共用体におけるビットフィールド

ビットフィールドを含む構造体と共用体を作ることができます。ビットフィールドは整数フィールドに対してのみ作ることができ、ビット幅は_fields_タプルの第三要素で指定します:

```
>>> class Int(Structure):
...     _fields_ = [("first_16", c_int, 16),
...                 ("second_16", c_int, 16)]
...
>>> print Int.first_16
<Field type=c_long, ofs=0:0, bits=16>
>>> print Int.second_16
<Field type=c_long, ofs=0:16, bits=16>
>>>
```

配列

Array はシーケンスであり、決まった数の同じ型のインスタンスを持ちます。

推奨されている配列の作成方法はデータ型に正の整数を掛けることです:

```
TenPointsArrayType = POINT * 10
```

ややわざとらしいデータ型の例になりますが、他のものに混ざって 4 個の POINT がある構造体です:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class MyStruct(Structure):
...     _fields_ = [("a", c_int),
...                 ("b", c_float),
...                 ("point_array", POINT * 4)]
>>>
>>> print len(MyStruct().point_array)
4
>>>
```

インスタンスはクラスを呼び出す通常の方法で作成します:

```
arr = TenPointsArrayType()
for pt in arr:
    print pt.x, pt.y
```

上記のコードは `0 0` という行が並んだものを表示します。配列の要素がゼロで初期化されているためです。

正しい型の初期化子を指定することもできます:

```
>>> from ctypes import *
>>> TenIntegers = c_int * 10
>>> ii = TenIntegers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> print ii
<c_long_Array_10 object at 0x...>
>>> for i in ii: print i,
...
1 2 3 4 5 6 7 8 9 10
>>>
```

ポインタ

ポインタのインスタンスは `ctypes` 型に対して `pointer` 関数を呼び出して作成します:

```
>>> from ctypes import *
>>> i = c_int(42)
>>> pi = pointer(i)
>>>
```

ポインタインスタンスはポインタが指すオブジェクト (上の例では `i`) を返す `contents` 属性を持ちます:

```
>>> pi.contents
c_long(42)
>>>
```

`ctypes` は OOR (original object return、元のオブジェクトを返すこと) ではないことに注意してください。属性を取り出す度に、新しい同等のオブジェクトを作成していいるのです:

```
>>> pi.contents is i
False
>>> pi.contents is pi.contents
False
>>>
```

別の c_int インスタンスがポインタの contents 属性に代入されると、これが記憶されているメモリ位置を指すポインタに変化します:

```
>>> i = c_int(99)
>>> pi.contents = i
>>> pi.contents
c_long(99)
>>>
```

ポインタインスタンスは整数でインデックス指定することもできます:

```
>>> pi[0]
99
>>>
```

整数インデックスへ代入するとポインタが指す値が変更されます:

```
>>> print i
c_long(99)
>>> pi[0] = 22
>>> print i
c_long(22)
>>>
```

0 ではないインデックスを使うこともできますが、C の場合と同じように自分が何をしているかを理解している必要があります。任意のメモリ位置にアクセスもしくは変更できるのです。一般的にこの機能を使うのは、C 関数からポインタを受け取り、そのポインタが単一の要素ではなく実際に配列を指していると**分かっている**場合だけです。

舞台裏では、pointer 関数は単にポインタインスタンスを作成するという以上のことを行っています。はじめにポインタ**型**を作成する必要があります。これは任意の ctypes 型を受け取る POINTER 関数を使って行われ、新しい型を返します:

```
>>> PI = POINTER(c_int)
>>> PI
<class 'ctypes.LP_c_long'>
>>> PI(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: expected c_long instead of int
>>> PI(c_int(42))
<ctypes.LP_c_long object at 0x...>
>>>
```

ポインタ型を引数なしで呼び出すと NULL ポインタを作成します。NULL ポインタは False ブール値を持っています:

```
>>> null_ptr = POINTER(c_int)()
>>> print bool(null_ptr)
False
>>>
```

ctypes はポインタの指す値を取り出すときに NULL かどうかを調べます (しかし、NULL でないポインタの指す値の取り出す行為は Python をクラッシュさせるでしょう):

```
>>> null_ptr[0]
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>

>>> null_ptr[0] = 1234
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>
```

型変換

たいていの場合、ctypes は厳密な型チェックを行います。これが意味するのは、関数の argtypes リスト内に、もしくは、構造体定義におけるメンバーフィールドの型として POINTER(c_int) がある場合、厳密に同じ型のインスタンスだけを受け取るということです。このルールには ctypes が他のオブジェクトを受け取る場合に例外がいくつかあります。例えば、ポインタ型の代わりに互換性のある配列インスタンスを渡すことができます。このように、POINTER(c_int) に対して、ctypes は c_int の配列を受け取ります:

```
>>> class Bar(Structure):
...     _fields_ = [("count", c_int), ("values", POINTER(c_int))]
...
>>> bar = Bar()
>>> bar.values = (c_int * 3)(1, 2, 3)
>>> bar.count = 3
>>> for i in range(bar.count):
...     print bar.values[i]
...
1
2
3
>>>
```

POINTER 型フィールドを NULL に設定するために、None を代入してもよい:

```
>>> bar.values = None
>>>
```

XXX list other conversions...

時には、非互換な型のインスタンスであることもあります。C では、ある型を他の型へキャストすることができます。ctypes は同じやり方で使える cast 関数を提供しています。上で定義した Bar 構造体は POINTER(c_int) ポインタまたは c_int 配列を values フィールドに対して受け取り、他の型のインスタンスは受け取りません:

```
>>> bar.values = (c_byte * 4)()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: incompatible types, c_byte_Array_4 instance instead of LP_c_long instance
>>>
```

このような場合には、cast 関数が便利です。

cast 関数は ctypes インスタンスを異なる ctypes データ型を指すポインタへキャストするために使えます。cast は二つのパラメータ、ある種のポインタかそのポインタへ変換できる ctypes オブジェクトと、ctypes ポインタ型を取ります。そして、第二引数のインスタンスを返します。このインスタンスは第一引数と同じメモリブロックを参照しています:

```
>>> a = (c_byte * 4)()
>>> cast(a, POINTER(c_int))
<ctypes.LP_c_long object at ...>
>>>
```

したがって、cast を Bar 構造体の values フィールドへ代入するために使うことができます:

```
>>> bar = Bar()
>>> bar.values = cast((c_byte * 4)(), POINTER(c_int))
>>> print bar.values[0]
0
>>>
```

不完全型

不完全型はメンバーがまだ指定されていない構造体、共用体もしくは配列です。C では、前方宣言により指定され、後で定義されます:

```
struct cell; /* 前方宣言 */

struct {
    char *name;
    struct cell *next;
} cell;
```

ctypes コードへの直接的な変換ではこうなるでしょう。しかし、動作しません:

```
>>> class cell(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("next", POINTER(cell))]
...
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in cell
NameError: name 'cell' is not defined
>>>
```

なぜなら、新しい class cell はクラス文自体の中では利用できないからです。ctypes では、cell クラスを定義して、_fields_属性をクラス文の後で設定することができます:

```
>>> from ctypes import *
>>> class cell(Structure):
...     pass
...
>>> cell._fields_ = [("name", c_char_p),
...                  ("next", POINTER(cell))]
>>>
```

試してみましょう。cell のインスタンスを二つ作り、互いに参照し合うようにします。最後に、つながったポインタを何度かたどります:

```
>>> c1 = cell()
>>> c1.name = "foo"
>>> c2 = cell()
>>> c2.name = "bar"
>>> c1.next = pointer(c2)
>>> c2.next = pointer(c1)
>>> p = c1
>>> for i in range(8):
...     print p.name,
...     p = p.next[0]
...
foo bar foo bar foo bar foo bar
>>>
```

コールバック関数

ctypes は C の呼び出し可能な関数ポインタを Python 呼び出し可能オブジェクトから作成できるようにします。これらは**コールバック関数**と呼ばれることがあります。

最初に、コールバック関数のためのクラスを作る必要があります。そのクラスには呼び出し規約、戻り値の型およびこの関数が受け取る引数の数と型についての情報があります。

CFUNCTYPE ファクトリ関数は通常の cdecl 呼び出し規約を用いてコールバック関数のための型を作成します。Windows では、WINFUNCTYPE ファクトリ関数が stdcall 呼び出し規約を用いてコールバック関数の型を作成します。

これらのファクトリ関数はともに最初の引数に戻り値の型、残りの引数としてコールバック関数が想定する引数の型を渡して呼び出されます。

標準 C ライブラリの qsort 関数を使う例を示します。これはコールバック関数の助けをかりて要素をソートするために使われます。qsort は整数の配列をソートするために使われます:

```
>>> IntArray5 = c_int * 5
>>> ia = IntArray5(5, 1, 7, 33, 99)
>>> qsort = libc.qsort
>>> qsort.restype = None
>>>
```

qsort はソートするデータを指すポインタ、データ配列の要素の数、要素の一つの大きさ、およびコールバック関数である比較関数へのポインタを引数に渡して呼び出さなければなりません。そして、コールバック関数は要素を指す二つのポインタを渡されて呼び出され、一番目が二番目より小さいなら負の数を、等しいならゼロを、それ以外なら正の数を返さなければなりません。

コールバック関数は整数へのポインタを受け取り、整数を返す必要があります。まず、コールバック関

数のための `type` を作成します:

```
>>> CMPFUNC = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>>
```

コールバック関数のはじめての実装なので、受け取った引数を単純に表示して、0 を返します (漸進型開発 (incremental development) です ;-):

```
>>> def py_cmp_func(a, b):
...     print "py_cmp_func", a, b
...     return 0
...
>>>
```

C の呼び出し可能なコールバック関数を作成します:

```
>>> cmp_func = CMPFUNC(py_cmp_func)
>>>
```

そうすると、準備完了です:

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func) # doctest: +WINDOWS
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
>>>
```

ポインタの中身にアクセスする方法がわかっているので、コールバック関数を再定義しましょう:

```
>>> def py_cmp_func(a, b):
...     print "py_cmp_func", a[0], b[0]
...     return 0
...
>>> cmp_func = CMPFUNC(py_cmp_func)
>>>
```

Windows での実行結果です:

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func) # doctest: +WINDOWS
py_cmp_func 7 1
py_cmp_func 33 1
py_cmp_func 99 1
py_cmp_func 5 1
py_cmp_func 7 5
py_cmp_func 33 5
py_cmp_func 99 5
py_cmp_func 7 99
py_cmp_func 33 99
py_cmp_func 7 33
>>>
```

linux ではソート関数がはるかに効率的に動作しており、実施する比較の数が少ないように見えるのが不思議です:

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func) # doctest: +LINUX
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 5 7
py_cmp_func 1 7
>>>
```

ええ、ほぼ完成です！最終段階は、実際に二つの要素を比較して実用的な結果を返すことです:

```
>>> def py_cmp_func(a, b):
...     print "py_cmp_func", a[0], b[0]
...     return a[0] - b[0]
...
>>>
```

Windows での最終的な実行結果です:

```
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func)) # doctest: +WINDOWS
py_cmp_func 33 7
py_cmp_func 99 33
py_cmp_func 5 99
py_cmp_func 1 99
py_cmp_func 33 7
py_cmp_func 1 33
py_cmp_func 5 33
py_cmp_func 5 7
py_cmp_func 1 7
py_cmp_func 5 1
>>>
```

Linux では:

```
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func)) # doctest: +LINUX
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

Windows の `qsort` 関数は linux バージョンより多く比較する必要があることがわかり、非常におもしろいですね！

簡単に確認できるように、今では配列はソートされています:

```
>>> for i in ia: print i,
...
1 5 7 33 99
>>>
```

コールバック関数についての重要な注意事項:

C コードから使われる限り、CFUNCTYPE オブジェクトへの参照を確実に保持してください。`ctypes` は保持しません。もしあなたがやらなければ、オブジェクトはゴミ集めされてしまい、コールバックしたときにあなたのプログラムをクラッシュさせるかもしれません。

dll からエクスポートされている値へアクセスする

dll は関数だけでなく変数をエクスポートしていることもあります。Python ライブラリにある例としては `Py_OptimizeFlag`、起動時の**-O** または**-OO** フラグに依存して、0, 1 または 2 が設定される整数があります。

`ctypes` は型の `in_dll` クラスメソッドを使ってこのように値にアクセスできます。*pythonapi* は Python C api へのアクセスできるようにするための予め定義されたシンボルです:

```
>>> opt_flag = c_int.in_dll(pythonapi, "Py_OptimizeFlag")
>>> print opt_flag
c_long(0)
>>>
```

インタープリタが**-O** を指定されて動き始めた場合、サンプルは `c_long(1)` を表示するでしょうし、**-OO** が指定されたならば `c_long(2)` を表示するでしょう。

ポインタの使い方を説明する拡張例では、Python がエクスポートする `PyImport_FrozenModules` ポインタにアクセスします。

Python ドキュメントからの引用すると: **このポインタはメンバーがすべて *NULL* またはゼロであるレコードを最後に持つ *“struct _frozen“* レコードの配列を指すように初期化されます。フローズン *(frozen)* モジュールがインポートされたとき、このテーブルから探索されます。サードパーティ製コードは動的に作成されたフローズンモジュールの集合を提供するためと、これにいたずらすることができます。**

これで、このポインタを操作することが役に立つことを証明できるでしょう。例の大きさを制限するために、このテーブルを `ctypes` を使って読む方法だけを示します:

```
>>> from ctypes import *
>>>
>>> class struct_frozen(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("code", POINTER(c_ubyte)),
...                 ("size", c_int)]
...
>>>
```

私たちは struct _frozen データ型を定義済みなので、このテーブルを指すポインタを得ることができます:

```
>>> FrozenTable = POINTER(struct_frozen)
>>> table = FrozenTable.in_dll(pythonapi, "PyImport_FrozenModules")
>>>
```

table が struct_frozen レコードの配列への pointer なので、その配列に対して反復処理を行えます。しかし、ループが確実に終了するようにする必要があります。なぜなら、ポインタに大きさの情報がないからです。遅かれ早かれ、アクセス違反か何かでクラッシュすることになるでしょう。NULL エントリに達したときはループを抜ける方が良い:

```
>>> for item in table:
...    print item.name, item.size
...    if item.name is None:
...        break
...
__hello__ 104
__phello__ -104
__phello__.spam 104
None 0
>>>
```

標準 Python はフローズンモジュールとフローズンパッケージ (負のサイズのメンバーで表されています) を持っているという事実はあまり知られておらず、テストにだけ使われています。例えば、import __hello__を試してみてください。

予期しないこと

ctypes には別のことを期待しているのに実際に起きる起きることは違うという場合があります。

次に示す例について考えてみてください:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class RECT(Structure):
...     _fields_ = ("a", POINT), ("b", POINT)
...
>>> p1 = POINT(1, 2)
>>> p2 = POINT(3, 4)
>>> rc = RECT(p1, p2)
>>> print rc.a.x, rc.a.y, rc.b.x, rc.b.y
1 2 3 4
>>> # now swap the two points
>>> rc.a, rc.b = rc.b, rc.a
>>> print rc.a.x, rc.a.y, rc.b.x, rc.b.y
3 4 3 4
>>>
```

うーん、最後の文に `3 4 1 2` と表示されることを期待していたはずです。何が起きたのでしょうか？上の行の `rc.a, rc.b = rc.b, rc.a` の各段階はこのようになります:

```
>>> temp0, temp1 = rc.b, rc.a
>>> rc.a = temp0
>>> rc.b = temp1
>>>
```

`temp0` と `temp1` は前記の `rc` オブジェクトの内部バッファでまだ使われているオブジェクトです。したがって、`rc.a = temp0` を実行すると `temp0` のバッファ内容が `rc` のバッファへコピーされます。さらに、これは `temp1` の内容を変更します。そのため、最後の代入 `rc.b = temp1` は、期待する結果にはならないのです。

Structure、Union および Array のサブオブジェクトを取り出しても、そのサブオブジェクトが**コピー**されるわけではなく、ルートオブジェクトの内部バッファにアクセスするラッパーオブジェクトを取り出すことを覚えておいてください。

期待とは違う振る舞いをする別の例はこれです:

```
>>> s = c_char_p()
>>> s.value = "abc def ghi"
>>> s.value
'abc def ghi'
>>> s.value is s.value
False
>>>
```

なぜ `False` と表示されるのでしょうか？ctypes インスタンスはメモリの内容にアクセスするいくつかの記述子付きメモリを含むオブジェクトです。メモリブロックに Python オブジェクトを保存してもオブジェクト自身が保存される訳ではなく、オブジェクトの `contents` が保存されます。その contents に再アクセスすると新しい Python オブジェクトがその度に作られます。

可変サイズのデータ型

`ctypes` は可変サイズの配列と構造体をサポートしています (バージョン 0.9.9.7 で追加されました)。

`resize` 関数は既存の ctypes オブジェクトのメモリバッファのサイズを変更したい場合に使えます。こ

の関数は第一引数にオブジェクト、第二引数に要求されたサイズをバイト単位で指定します。メモリブロックはオブジェクト型で指定される通常のメモリブロックより小さくすることはできません。これをやろうとすると、ValueError が発生します:

```
>>> short_array = (c_short * 4)()
>>> print sizeof(short_array)
8
>>> resize(short_array, 4)
Traceback (most recent call last):
    ...
ValueError: minimum size is 8
>>> resize(short_array, 32)
>>> sizeof(short_array)
32
>>> sizeof(type(short_array))
8
>>>
```

これはこれで上手くいっていますが、この配列の追加した要素へどうやってアクセスするのでしょうか？この型は要素の数が 4 個であるとまだ認識しているので、他の要素にアクセスするとエラーになります:

```
>>> short_array[:]
[0, 0, 0, 0]
>>> short_array[7]
Traceback (most recent call last):
    ...
IndexError: invalid index
>>>
```

ctypes で可変サイズのデータ型を使うもう一つの方法は、必要なサイズが分かった後に Python の動的性質を使って一つ一つデータ型を (再) 定義することです。

バグ、ToDo および実装されていないもの

列挙型は実装されていません。ベースクラスに c_int を使うことで簡単に実装できます。

long double は実装されていません。

14.14.2 ctypes リファレンス

共有ライブラリを見つける

コンパイルされる言語でプログラミングしている場合、共有ライブラリはプログラムをコンパイル/リンクしているときと、そのプログラムが動作しているときにアクセスされます。

ctypes ライブラリローダーはプログラムが動作しているときのように振る舞い、ランタイムローダーを直接呼び出すのに対し、find_library 関数の目的はコンパイラが行うのと似た方法でライブラリを探し出すことです (複数のバージョンの共有ライブラリがあるプラットホームでは、一番最近に見つかったものがロードされます)。

ctypes.util モジュールはロードするライブラリを決めるのに役立つ関数を提供します。

find_library(name)

ライブラリを見つけてパス名を返そうと試みます。*name* は *lib* のような接頭辞、.so、.dylib のような接尾辞、あるいは、バージョン番号が何も付いていないライブラリの名前です (これは posix リ

ンカのオプション**-l**) に使われている形式です)。もしライブラリが見つからなければ、None を返します。

厳密な機能はシステムに依存します。

Linux では、find_library はライブラリファイルを見つけるために外部プログラム (/sbin/ldcon?g、gcc および objdump) を実行しようとします。ライブラリファイルのファイル名を返します。いくつか例があります:

```
>>> from ctypes.util import find_library
>>> find_library("m")
'libm.so.6'
>>> find_library("c")
'libc.so.6'
>>> find_library("bz2")
'libbz2.so.1.0'
>>>
```

OS X では、find_library はライブラリの位置を探すために、予め定義された複数の命名方法とパスを試し、成功すればフルパスを返します:

```
>>> from ctypes.util import find_library
>>> find_library("c")
'/usr/lib/libc.dylib'
>>> find_library("m")
'/usr/lib/libm.dylib'
>>> find_library("bz2")
'/usr/lib/libbz2.dylib'
>>> find_library("AGL")
'/System/Library/Frameworks/AGL.framework/AGL'
>>>
```

Windows では、find_library はシステムの探索パスに沿って探し、フルパスを返します。しかし、予め定義された命名方法がないため、find_library("c") のような呼び出しは失敗し、None を返します。

もし ctypes を使って共有ライブラリをラップするなら、実行時にライブラリを探すために find_library を使う代わりに、開発時に共有ライブラリ名をを決めて、ラッパーモジュールにハードコードした方が良い**かもしれません**。

共有ライブラリをロードする

共有ライブラリを Python プロセスへロードする方法はいくつかあります。一つの方法は下記のクラスの一つをインスタンス化することです:

class CDLL (*name, mode=DEFAULT_MODE, handle=None*)

このクラスのインスタンスはロードされた共有ライブラリをあらわします。これらのライブラリの関数は標準 C 呼び出し規約を使用し、int を返すと仮定されます。

class OleDLL (*name, mode=DEFAULT_MODE, handle=None*)

Windows 用: このクラスのインスタンスはロードされた共有ライブラリをあらわします。これらのライブラリの関数は stdcall 呼び出し規約を使用し、windows 固有の HRESULT コードを返すと仮定されます。HRESULT 値には関数呼び出しが失敗したのか成功したのかを特定する情報とともに、補足のエラーコードが含まれます。戻り値が失敗を知らせたならば、WindowsError が自動的に発生します。

class WinDLL (*name, mode=DEFAULT_MODE, handle=None*)

Windows 用: このクラスのインスタンスはロードされた共有ライブラリをあらわします。これらのライブラリの関数は stdcall 呼び出し規約を使用し、デフォルトでは int を返すと仮定されます。

Windows CE では標準呼び出し規約だけが使われます。便宜上、このプラットホームでは、WinDLL と OleDLL が標準呼び出し規約を使用します。

これらのライブラリがエクスポートするどの関数でも呼び出す前に Python GIL は解放され、後でまた必要になります。

class PyDLL (*name, mode=DEFAULT_MODE, handle=None*)

Python GIL が関数呼び出しの間解放**されず**、関数実行の後に Python エラーフラグがチェックされるということを除けば、このクラスのインスタンスは CDLL インスタンスのように振る舞います。エラーフラグがセットされた場合、Python 例外が発生します。

要するに、これは Python C api 関数を直接呼び出すのに便利だというだけです。

これらすべてのクラスは少なくとも一つの引数、すなわちロードする共有ライブラリのパスを渡して呼び出すことでインスタンス化されます。すでにロード済みの共有ライブラリへのハンドルがあるなら、handle 名前付き引数として渡すことができます。土台となっているプラットホームの dlopen または LoadLibrary 関数がプロセスへライブラリをロードするために使われ、そのライブラリに対するハンドルを得ます。

mode パラメータはライブラリがどうやってロードされたかを特定するために使うことができます。詳細は、dlopen(3) マニュアルページを参考にしてください。Windows では *mode* は無視されます。

RTLD_GLOBAL

mode パラメータとして使うフラグ。このフラグが利用できないプラットホームでは、整数のゼロと定義されています。

RTLD_LOCAL

mode パラメータとして使うフラグ。これが利用できないプラットホームでは、*RTLD_GLOBAL* と同様です。

DEFAULT_MODE

共有ライブラリをロードするために使われるデフォルトモード。OSX 10.3 では *RTLD_GLOBAL* であり、そうでなければ *RTLD_LOCAL* と同じです。

これらのクラスのインスタンスには公開メソッドがありません。けれども、__getattr__と__getitem__は特別ははたらきをします。その共有ライブラリがエクスポートする関数に添字を使って属性としてアクセスできるのです。__getattr__と__getitem__のどちらもが結果をキャッシュし、そのため常に同じオブジェクトを返すことに注意してください。

次に述べる公開属性が利用できます。それらの名前はエクスポートされた関数名に衝突しないように下線で始まります:

_handle

ライブラリへのアクセスに用いられるシステムハンドル。

_name

コンストラクタに渡されたライブラリの名前。

共有ライブラリは (LibraryLoader クラスのインスタンスである) 前もって作られたオブジェクトの一つを使うことによってロードすることもできます。それらの LoadLibrary メソッドを呼び出すか、ローダーインスタンスの属性としてライブラリを取り出すかのどちらかによりロードします。

class LibraryLoader (*dlltype*)

共有ライブラリをロードするクラス。dlltype は CDLL、PyDLL、WinDLL もしくは OleDLL 型の一つであるべきです。

__getattr__は特別なはたらきをします: ライブラリローダーインスタンスの属性として共有ライ

ブラリにアクセスするとそれがロードされるということを可能にします。結果はキャッシュされます。そのため、繰り返し属性アクセスを行うといつも同じライブラリが返されます。

LoadLibrary(*name*)
共有ライブラリをプロセスへロードし、それを返します。このメソッドはライブラリの新しいインスタンスを常に返します。

これらの前もって作られたライブラリローダーを利用することができます:

cdll
CDLL インスタンスを作ります。

windll
Windows 用: WinDLL インスタンスを作ります。

oledll
Windows 用: OleDLL インスタンスを作ります。

pydll
PyDLL インスタンスを作ります。

C Python api に直接アクセスするために、すぐに使用できる Python 共有ライブラリオブジェクトが用意されています:

pythonapi
属性として Python C api 関数を公開する PyDLL のインスタンス。これらすべての関数は C int を返すと仮定されますが、もちろん常に正しいとは限りません。そのため、これらの関数を使うためには正しい restype 属性を代入しなければなりません。

外部関数

前節で説明した通り、外部関数はロードされた共有ライブラリの属性としてアクセスできます。デフォルトではこの方法で作成された関数オブジェクトはどんな数の引数でも受け取り、引数としてどんな ctypes データのインスタンスをも受け取り、そして、ライブラリローダーが指定したデフォルトの結果の値の型を返します。関数オブジェクトはプライベートクラスのインスタンスです:

class _FuncPtr
C の呼び出し可能外部関数のためのベースクラス。

外部関数のインスタンスも C 互換データ型です。それらは C の関数ポインタを表しています。

この振る舞いは外部関数オブジェクトの特別な属性に代入することによって、カスタマイズすることができます。

restype
外部関数の結果の型を指定するために ctypes 型を代入する。何も返さない関数を表す void に対しては None を使います。

ctypes 型ではない呼び出し可能な Python オブジェクトを代入することは可能です。このような場合、関数が C int を返すと仮定され、呼び出し可能オブジェクトはこの整数を引数に呼び出されます。さらに処理を行ったり、エラーチェックをしたりできるようにするためです。これの使用は推奨されません。より柔軟な後処理やエラーチェックのためには restype として ctypes 型を使い、errcheck 属性へ呼び出し可能オブジェクトを代入してください。

argtypes
関数が受け取る引数の型を指定するために ctypes 型のタプルを代入します。stdcall 呼び出し規約をつかう関数はこのタプルの長さと同じ数の引数で呼び出されます。その上、C 呼び出し規約をつかう関数は追加の不特定の引数も取ります。

外部関数が呼ばれたとき、それぞれの実引数は `argtypes` タプルの要素の `from_param` クラスメソッドへ渡されます。このメソッドは実引数を外部関数が受け取るオブジェクトに合わせて変えられるようにします。例えば、`argtypes` タプルの `c_char_p` 要素は、ctypes 変換規則にしたがって引数として渡されたユニコード文字列をバイト文字列へ変換するでしょう。

新: ctypes 型でない要素を argtypes に入れることができますが、個々の要素は引数として使える値 (整数、文字列、ctypes インスタンス) を返す `from_param` メソッドを持っていなければなりません。これにより関数パラメータとしてカスタムオブジェクトを適合するように変更できるアダプタが定義可能となります。

errcheck

Python 関数または他の呼び出し可能オブジェクトをこの属性に代入します。呼び出し可能オブジェクトは三つ以上の引数とともに呼び出されます。

callable(*result, func, arguments*)

`result` は外部関数が返すもので、`restype` 属性で指定されます。

`func` は外部関数オブジェクト自身で、これにより複数の関数の処理結果をチェックまたは後処理するために、同じ呼び出し可能オブジェクトを再利用できるよになります。

`arguments` は関数呼び出しに最初に渡されたパラメータが入ったタプルです。これにより使われた引数に基づた特別な振る舞いをさせることができるようになります。

この関数が返すオブジェクトは外部関数呼び出しから返された値でしょう。しかし、戻り値をチェックして、外部関数呼び出しが失敗しているなら例外を発生させることもできます。

exception ArgumentError()

この例外は外部関数呼び出しが渡された引数を変換できなかったときに発生します。

関数プロトタイプ

外部関数は関数プロトタイプをインスタンス化することによって作成されます。関数プロトタイプは C の関数プロトタイプと似ています。実装を定義せずに、関数 (戻り値、引数の型、呼び出し規約) を記述します。ファクトリ関数は関数に要求する戻り値の型と引数の型とともに呼び出されます。

CFUNCTYPE(*restype, *argtypes*)

返された関数プロトタイプは標準 C 呼び出し規約をつかう関数を作成します。関数は呼び出されている間 GIL を解放します。

WINFUNCTYPE(*restype, *argtypes*)

Windows 用: 返された関数プロトタイプは `stdcall` 呼び出し規約をつかう関数を作成します。ただし、`WINFUNCTYPE` が `CFUNCTYPE` と同じである Windows CE を除く。関数は呼び出されている間 GIL を解放します。

PYFUNCTYPE(*restype, *argtypes*)

返された関数プロトタイプは Python 呼び出し規約をつかう関数を作成します。関数は呼び出されている間 GIL を解放**しません**。

ファクトリ関数によって作られた関数プロトタイプは呼び出しのパラメータの型と数に依存した別の方法でインスタンス化することができます。

prototype(*address*)

指定されたアドレスの外部関数を返します。

prototype(*callable*)

Python の `callable` から C の呼び出し可能関数 (コールバック関数) を作成します。

prototype(*func_spec*[, *paramflags*])

共有ライブラリがエクスポートしている外部関数を返します。func_spec は 2 要素タプル (name_or_ordinal, library) でなければなりません。第一要素はエクスポートされた関数の名前である文字列、またはエクスポートされた関数の序数である小さい整数です。第二要素は共有ライブラリインスタンスです。

prototype(*vtbl_index, name*[, *paramflags*[, *iid*]])

COM メソッドを呼び出す外部関数を返します。vtbl_index は仮想関数テーブルのインデックスで、非負の小さい整数です。*name* は COM メソッドの名前です。*iid* はオプションのインターフェイス識別子へのポインタで、拡張されたエラー情報の提供のために使われます。

COM メソッドは特殊な呼び出し規約を用います。このメソッドは argtypes タプルに指定されたパラメータに加えて、第一引数として COM インターフェイスへのポインタを必要とします。

オプションの *paramflags* パラメータは上述した機能より多機能な外部関数ラッパーを作成します。

paramflags は argtypes と同じ長さのタプルでなければならない。

このタプルの個々の要素はパラメータについてのより詳細な情報を持ち、1、2 もしくは 3 要素を含むタプルでなければならない。

第一要素はパラメータについてのフラグを含んだ整数です。

1

入力パラメータを関数に指定します。

2

出力パラメータ。外部関数が値を書き込みます。

4

デフォルトで整数ゼロになる入力パラメータ。

オプションの第二要素はパラメータ名の文字列です。これが指定された場合は、外部関数を名前付きパラメータで呼び出すことができます。

オプションの第三要素はこのパラメータのデフォルト値です。

この例では、デフォルトパラメータと名前付き引数をサポートするために Windows MessageBoxA 関数をラップする方法を示します。windows ヘッダファイルの C の宣言はこれです:

```
WINUSERAPI int WINAPI
MessageBoxA(
    HWND hWnd ,
    LPCSTR lpText,
    LPCSTR lpCaption,
    UINT uType);
```

ctypes を使ってラップします:

```
>>> from ctypes import c_int, WINFUNCTYPE, windll
>>> from ctypes.wintypes import HWND, LPCSTR, UINT
>>> prototype = WINFUNCTYPE(c_int, HWND, LPCSTR, LPCSTR, UINT)
>>> paramflags = (1, "hwnd", 0), (1, "text", "Hi"), (1, "caption", None), (1, "flags", 0)
>>> MessageBox = prototype(("MessageBoxA", windll.user32), paramflags)
>>>
```

今は MessageBox 外部関数をこのような方法で呼び出すことができます:

```
>>> MessageBox()
>>> MessageBox(text="Spam, spam, spam")
>>> MessageBox(flags=2, text="foo bar")
>>>
```

二番目の例は出力パラメータについて説明します。win32 の GetWindowRect 関数は、指定されたウィンドウの大きさを呼び出し側が与える RECT 構造体へコピーすることで取り出します。C の宣言はこうです:

```
WINUSERAPI BOOL WINAPI
GetWindowRect(
     HWND hWnd,
     LPRECT lpRect);
```

ctypes を使ってラップします:

```
>>> from ctypes import POINTER, WINFUNCTYPE, windll, WinError
>>> from ctypes.wintypes import BOOL, HWND, RECT
>>> prototype = WINFUNCTYPE(BOOL, HWND, POINTER(RECT))
>>> paramflags = (1, "hwnd"), (2, "lprect")
>>> GetWindowRect = prototype(("GetWindowRect", windll.user32), paramflags)
>>>
```

もし単一の値もしくは一つより多い場合には出力パラメータ値が入ったタプルがあるならば、出力パラメータを持つ関数は自動的に出力パラメータ値を返すでしょう。そのため、今は GetWindowRect 関数は呼び出されたときに RECT インスタンスを返します。

さらに出力処理やエラーチェックを行うために、出力パラメータを errcheck プロトコルと組み合わせることができます。win32 GetWindowRect api 関数は成功したか失敗したかを知らせるために BOOL を返します。そのため、この関数はエラーチェックを行って、api 呼び出しが失敗した場合に例外を発生させることができます:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     return args
>>> GetWindowRect.errcheck = errcheck
>>>
```

errcheck 関数が変更なしに受け取った引数タプルを返したならば、ctypes は出力パラメータに対して通常の処理を続けます。RECT インスタンスの代わりに window 座標のタプルを返してほしいなら、関数のフィールドを取り出し、代わりにそれらを返すことができます。通常処理はもはや行われないでしょう:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     rc = args[1]
...     return rc.left, rc.top, rc.bottom, rc.right
>>>
>>> GetWindowRect.errcheck = errcheck
>>>
```

ユーティリティ関数

addressof(*obj*)
メモリバッファのアドレスを示す整数を返します。obj は ctypes 型のインスタンスでなければなりません。

alignment(*obj_or_type*)
ctypes 型のアライメントの必要条件を返します。obj_or_type は ctypes 型またはインスタンスでなければなりません。

byref(*obj*)
obj(ctypes 型のインスタンスでなければならない) への軽量ポインタを返します。返されるオブジェクトは外部関数呼び出しのパラメータとしてのみ使用できます。pointer(obj) と似たふるまいをしますが、作成が非常に速く行えます。

cast(*obj, type*)
この関数は C のキャスト演算子に似ています。obj と同じメモリブロックを指している type の新しいインスタンスを返します。type はポインタ型でなければならず、obj はポインタとして解釈できるオブジェクトでなければならない。

create_string_buffer(*init_or_size*[, *size*])
この関数は変更可能な文字バッファを作成します。返されるオブジェクトは c_char の ctypes 配列です。

init_or_size は配列のサイズを指定する整数もしくは配列要素を初期化するために使われる文字列である必要があります。

第一引数として文字列が指定された場合は、バッファが文字列の長さより一要素分大きく作られます。配列の最後の要素が NUL 終端文字であるためです。文字列の長さを使うべきでない場合は、配列のサイズを指定するために整数を第二引数として渡すことができます。

第一引数がユニコード文字列ならば、ctypes 変換規則にしたがい 8 ビット文字列へ変換されます。

create_unicode_buffer(*init_or_size*[, *size*])
この関数は変更可能なユニコード文字バッファを作成します。返されるオブジェクトは c_wchar の ctypes 配列です。

init_or_size は配列のサイズを指定する整数もしくは配列要素を初期化するために使われるユニコード文字列です。

第一引数としてユニコード文字列が指定された場合は、バッファが文字列の長さより一要素分大きく作られます。配列の最後の要素が NUL 終端文字であるためです。文字列の長さを使うべきでない場合は、配列のサイズを指定するために整数を第二引数として渡すことができます。

第一引数が 8 ビット文字列ならば、ctypes 変換規則にしたがいユニコード文字列へ変換されます。

DllCanUnloadNow()
Windows 用: この関数は ctypes をつかってインプロセス COM サーバーを実装できるようにするためのフックです。_ctypes 拡張 dll がエクスポートしている DllCanUnloadNow 関数から呼び出されます。

DllGetClassObject()
Windows 用: この関数は ctypes をつかってインプロセス COM サーバーを実装できるようにするためのフックです。_ctypes 拡張 dll がエクスポートしている DllGetClassObject 関数から呼び出されます。

FormatError([*code*])
Windows 用: エラーコードの説明文を返す。エラーコードが指定されない場合は、Windows api 関数 GetLastError を呼び出して、もっとも新しいエラーコードが使われます。

GetLastError()

Windows 用: 呼び出し側のスレッド内で Windows によって設定された最新のエラーコードを返します。

memmove(*dst, src, count*)

標準 C の memmove ライブラリ関数と同じもの: *count* バイトを `src` から *dst* へコピーします。*dst* と `src` はポインタへ変換可能な整数または ctypes インスタンスでなければなりません。

memset(*dst, c, count*)

標準 C の memset ライブラリ関数と同じもの: アドレス *dst* のメモリブロックを値 *c* を *count* バイト分書き込みます。*dst* はアドレスを指定する整数または ctypes インスタンスである必要があります。

POINTER(*type*)

このファクトリ関数は新しい ctypes ポインタ型を作成して返します。ポインタ型はキャッシュされ、内部で再利用されます。したがって、この関数を繰り返し呼び出してもコストは小さいです。型は ctypes 型でなければなりません。

pointer(*obj*)

この関数は `obj` を指す新しいポインタインスタンスを作成します。戻り値は POINTER(type(obj)) 型のオブジェクトです。

注意: 外部関数呼び出しへオブジェクトへのポインタを渡したいだけなら、はるかに高速な `byref(obj)` を使うべきです。

resize(*obj, size*)

この関数は obj の内部メモリバッファのサイズを変更します。obj は ctypes 型のインスタンスでなければなりません。バッファを sizeof(type(obj)) で与えられるオブジェクト型の本来のサイズより小さくすることはできませんが、バッファを拡大することはできます。

set_conversion_mode(*encoding, errors*)

この関数は 8 ビット文字列とユニコード文字列の間で変換するときに使われる規則を設定します。encoding は'`utf-8`'や'`mbcs`'のようなエンコーディングを指定する文字列でなければなりません。errors はエンコーディング/デコーディングエラーについてのエラー処理を指定する文字列でなければなりません。指定可能な値の例としては、"`strict`"、"`replace`"または"`ignore`"があります。

`set_conversion_mode` は以前の変換規則を含む 2 要素タプルです。windows では初期の変換規則は (`'mbcs', 'ignore'`) であり、他のシステムでは (`'ascii', 'strict'`) です。

sizeof(*obj_or_type*)

ctypes 型もしくはインスタンスのメモリバッファのサイズをバイト単位で返します。C の `sizeof()` 関数と同じ動作です。

string_at(*address*[, *size*])

この関数はメモリアドレス address から始まる文字列を返します。size が指定された場合はサイズとして使われます。指定されなければ、文字列がゼロ終端されていると仮定します。

WinError(*code=None, descr=None*)

Windows 用: この関数は ctypes の中でもおそらく最悪な名前がつけれたものです。WindowsError のインスタンスを作成します。*code* が指定されないならば、エラーコードを決めるために `GetLastError` が呼び出されます。`descr` が指定されないならば、`FormatError` がエラーの説明文を得るために呼び出されます。

wstring_at(*address*)

この関数はユニコード文字列としてメモリアドレス `address` から始まるワイドキャラクタ文字列を返します。`size` が指定されたならば、文字列の文字数として使われます。指定されなければ、文字列がゼロ終端されていると仮定します。

データ型

class _CData

この非公開クラスはすべての ctypes データ型の共通のベースクラスです。他のものに取り込まれることで、すべての ctypes 型インスタンスがは C 互換データを保持するメモリブロックを内部に持ちます。メモリブロックのアドレスを `addressof()` ヘルパー関数が返さします。別のインスタンス変数は`_objects` として公開されます。これはメモリブロックがポインタを含む場合に存続し続ける必要のある他の Python オブジェクトを含んでいます。

ctypes データ型の共通メソッド、すべてのクラスメソッドが存在します (正確には、メタクラスのメソッドです):

from_address(*address*)

このメソッドは address によって指定されたメモリを使用している ctypes 型のインスタンスを返します。address は整数でなければならない。

from_param(*obj*)

このメソッドは obj を ctypes 型に適合させます。その型が外部関数の `argtypes` タプルに存在する場合に、実際の外部関数呼び出しに使われるオブジェクトを与えて呼び出します。関数呼び出しパラメータとして使えるオブジェクトを返さなければなりません。

すべての ctypes データ型にはこのクラスメソッドのデフォルト実装が存在し、通常は `obj` がその型のインスタンスならそのままを返します。いくつかの型は他のオブジェクトも受け取ります。

in_dll(*library, name*)

このメソッドは共有ライブラリがエクスポートする ctypes 型のインスタンスを返します。*name* はデータをエクスポートしているシンボル名であり、*library* はロードされた共有ライブラリです。

ctypes データ型に共通のインスタンス変数:

_b_base_

時には ctypes データインスタンスは自信が含まれるメモリブロックを持たないことがあります。その代わり、ベースオブジェクトのメモリブロックの一部を共有します。_b_base_読み出し専用メンバーがメモリブロックを保有しているルート ctypes オブジェクトです。

_b_needsfree_

ctypes データインスタンスが確保したメモリブロック自体を保有している場合、この読み出し専用変数は真であり、それ以外では偽です。

_objects

このメンバーは `None` または Python オブジェクトが含まれる辞書であり、Python オブジェクトはメモリブロックの内容を有効に保つために、生き続けている必要があります。このオブジェクトはデバッギングのためにエクスポートされているだけです。この辞書の内容を決して変更してはいけません。

基本データ型

class _SimpleCData

この非公開クラスはすべての基本 ctypes データ型のベースクラスです。ここでこのクラスに触れたのは、基本 ctypes データ型の共通属性を含んでいるからです。`_SimpleCData` は`_CData` のサブクラスですので、そのメソッドと属性を継承しています。

インスタンスはただ一つの属性を持ちます:

value

この属性にはインスタンスの実際の値が入っています。それは整数とポインタ型に対しては整数、文字型に対しては一文字だけの文字列、文字ポインタ型に対しては Python 文字列またはユニコード文

字列です。

value 属性を ctypes インスタンスから取り出したとき、たいていは新しいオブジェクトがその都度返されます。ctypes は元のオブジェクトを戻すことは**しません**。常に新しいオブジェクトが作られます。同じことはすべての他の ctypes オブジェクトインスタンスに対しても当てはまります。

基本データ型は、外部関数呼び出しの結果として返されたときや、例えば構造体のフィールドメンバーや配列要素を取り出すときに、ネイティブの Python 型へ透過的に変換されます。言い換えると、外部関数が c_char_p の restype を持つ場合は、c_char_p インスタンスでは**なく**常に Python 文字列を受け取ることでしょう。

基本データ型のサブクラスはこの振る舞いを継承**しません**。したがって、外部関数の restype が c_void_p のサブクラスならば、関数呼び出しからこのサブクラスのインスタンスを受け取ります。もちろん、value 属性にアクセスしてポインタの値を得ることができます。

これらが基本データ型です:

class c_byte
C の signed char データ型を表し、小整数として値を解釈します。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。

class c_char
C char データ型を表し、単一の文字として値を解釈します。コンストラクタはオプションの文字列初期化子を受け取り、その文字列の長さちょうど一文字である必要があります。

class c_char_p
C char *データ型を表し、ゼロ終端文字列へのポインタでなければなりません。コンストラクタは整数のアドレスもしくは文字列を受け取ります。

class c_double
C double データ型を表します。コンストラクタはオプションの浮動小数点数初期化子を受け取ります。

class c_float
C float データ型を表します。コンストラクタはオプションの浮動小数点数初期化子を受け取ります。

class c_int
C signed int データ型を表します。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。sizeof(int) == sizeof(long) であるプラットホームでは、c_long の別名です。

class c_int8
C 8-bit signed int データ型を表します。たいていは、c_byte の別名です。

class c_int16
C 16-bit signed int データ型を表します。たいていは、c_short の別名です。

class c_int32
C 32-bit signed int データ型を表します。たいていは、c_int の別名です。

class c_int64
C 64-bit signed int データ型を表します。たいていは、c_longlong の別名です。

class c_long
C signed long データ型を表します。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。

class c_longlong
C signed long long データ型を表します。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。

class c_short

C signed short データ型を表します。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。

class c_size_t

C size_t データ型を表します。

class c_ubyte

C unsigned char データ型を表します。その値は小整数として解釈されます。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。

class c_uint

C unsigned int データ型を表します。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。sizeof(int) == sizeof(long) であるプラットホームでは、c_ulong の別名です。

class c_uint8

C 8-bit unsigned int データ型を表します。たいていは、c_ubyte の別名です。

class c_uint16

C 16-bit unsigned int データ型を表します。たいていは、c_ushort の別名です。

class c_uint32

C 32-bit unsigned int データ型を表します。たいていは、c_uint の別名です。

class c_uint64

C 64-bit unsigned int データ型を表します。たいていは、c_ulonglong の別名です。

class c_ulong

C unsigned long データ型を表します。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。

class c_ulonglong

C unsigned long long データ型を表します。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。

class c_ushort

C unsigned short データ型を表します。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。

class c_void_p

C void *データ型を表します。値は整数として表されます。コンストラクタはオプションの整数初期化子を受け取ります。

class c_wchar

C wchar_t データ型を表し、値はユニコード文字列の単一の文字として解釈されます。コンストラクタはオプションの文字列初期化子を受け取り、その文字列の長さはちょうど一文字である必要があります。

class c_wchar_p

C wchar_t *データ型を表し、ゼロ終端ワイド文字列へのポインタでなければなりません。コンストラクタは整数のアドレスもしくは文字列を受け取ります。

class c_bool

C bool データ型 (より正確には、C99 の_Bool) を表します。その値は True または False であり、コンストラクタはどんなオブジェクト (真値を持ちます) でも受け取ります。 2.6 で追加された仕様です。

class HRESULT

Windows 用: HRESULT 値を表し、関数またはメソッド呼び出しに対する成功またはエラーの情報を

含んでいます。

class py_object

C PyObject *データ型を表します。引数なしでこれを呼び出すと NULL PyObject *ポインタを作成します。

ctypes.wintypes モジュールは他の Windows 固有のデータ型を提供します。例えば、HWND、WPARAM または DWORD です。MSG や RECT のような有用な構造体も定義されています。

標準データ型

class Union (**args, **kw*)

ネイティブのバイトオーダーでの共用体のための抽象ベースクラス。

class BigEndianStructure (**args, **kw*)

ビックエンディアンバイトオーダーでの構造体のための抽象ベースクラス。

class LittleEndianStructure (**args, **kw*)

リトルエンディアンバイトオーダーでの構造体のための抽象ベースクラス。

ネイティブではないバイトオーダーを持つ構造体にポインタ型フィールドあるいはポインタ型フィールドを含む他のどんなデータ型をも入れることはできません。

class Structure (**args, **kw*)

ネイティブのバイトオーダーでの構造体のための抽象ベースクラス。

具象構造体型と具象共用体型はこれらの型の一つをサブクラス化することで作らなければなりません。少なくとも、_fields_クラス変数を定義する必要があります。ctypes は、属性に直接アクセスしてフィールドを読み書きできるようにする記述子を作成するでしょう。これらは、

fields

構造体のフィールドを定義するシーケンス。要素は 2 要素タプルか 3 要素タプルでなければなりません。第一要素はフィールドの名前です。第二要素はフィールドの型を指定します。それはどんな ctypes データ型でも構いません。

c_int のような整数型のために、オプションの第三要素を与えることができます。フィールドのビット幅を定義する正の小整数である必要があります。

一つの構造体と共用体の中で、フィールド名はただ一つである必要があります。これはチェックされません。名前が繰り返しでてきたときにアクセスできるのは一つのフィールドだけです。

Structure サブクラスを定義するクラス文の**後で**、_fields_クラス変数を定義することができます。これにより自身を直接または間接的に参照するデータ型を作成できるようになります:

```
class List(Structure):
    pass
List._fields_ = [("pnext", POINTER(List)),
                 ...
                ]
```

しかし、_fields_クラス変数はその型が最初に使われる (インスタンスが作成される、それに対して sizeof() が呼び出されるなど) より前に定義されていなければなりません。その後_fields_クラス変数へ代入すると AttributeError が発生します。

構造体および共用体サブクラスは位置引数と名前付き引数の両方を受け取ります。位置引数は_fields_定義中に現れたのと同じ順番でフィールドを初期化するために使われ、名前付き引数は対応する名前を使ってフィールドを初期化するために使われます。

構造体型のサブクラスを定義することができ、もしあるならサブクラス内で定義された_fields_に加えて、ベースクラスのフィールドも継承します。

pack

インスタンスの構造体フィールドのアライメントを上書きできるようにするオブションの小整数。_pack_は_fields_が代入されたときすでに定義されていなければならない。そうでなければ、何ら影響はありません。

anonymous

無名 (匿名) フィールドの名前が並べあげられたオプションのシーケンス。_fields_が代入されたとき、_anonymous_がすでに定義されていなければならない。そうでなければ、何ら影響はありません。

この変数に並べあげられたフィールドは構造体型もしくは共用体型フィールドである必要があります。構造体フィールドまたは共用体フィールドを作る必要なく、入れ子になったフィールドに直接アクセスできるようにするために、ctypes は構造体型の中に記述子を作成します。

型の例です (Windows):

```
class _U(Union):
    _fields_ = [("lptdesc", POINTER(TYPEDESC)),
                ("lpadesc", POINTER(ARRAYDESC)),
                ("hreftype", HREFTYPE)]

class TYPEDESC(Structure):
    _fields_ = [("u", _U),
                ("vt", VARTYPE)]

    _anonymous_ = ("u",)
```

TYPEDESC 構造体は COM データ型を表現しており、vt フィールドは共用体フィールドのどれが有効であるかを指定します。u フィールドは匿名フィールドとして定義されているため、TYPEDESC インスタンスから取り除かれてそのメンバーへ直接アクセスできます。td.lptdesc と td.u.lptdesc は同等ですが、前者がより高速です。なぜなら一時的な共用体インスタンスを作る必要がないためです:

```
td = TYPEDESC()
td.vt = VT_PTR
td.lptdesc = POINTER(some_type)
td.u.lptdesc = POINTER(some_type)
```

構造体のサブ-サブクラスを定義することができ、ベースクラスのフィールドを継承します。サブクラス定義に別の_fields_変数がある場合は、この中で指定されたフィールドはベースクラスのフィールドへ追加されます。

構造体と共用体のコンストラクタは位置引数とキーワード引数の両方を受け取ります。位置引数は_fields_の中に現れたのと同じ順番でメンバーフィールドを初期化するために使われます。コンストラクタのキーワード引数は属性代入として解釈され、そのため、同じ名前をもつ_fields_を初期化するか、_fields_に存在しない名前に対しては新しい属性を作ります。

配列とポインタ

未作成 - チュートリアルの節 14.14.1 ポインタと節 14.14.1 配列を参照してください。

十五章

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）でも利用できることがあります。次に概要を示します。

select	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。
thread	1 つのインタープリタの中でのマルチスレッド制御
threading	高水準のスレッドインタフェース
dummy_thread	`thread` の代替モジュール。
dummy_threading	`threading` の代替モジュール。
mmap	UNIX と Windows のメモリマップファイルへのインターフェース
readline	Python のための GNU readline サポート。
rlcompleter	GNU readline ライブラリ向けの Python 識別子補完

15.1 `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 15.1.1 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' からなるシーケンスです: 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 つのシー

ケンスはそれぞれ入力、出力、そして '例外状態' に対応します。いずれかに空のシーケンスを指定してもかまいませんが、3 つ全てを空のシーケンス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。

シーケンス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。

15.1.1 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(最大のファイル記述子番号) なのに対し、`poll()` は O(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
`POLLIN`	読み出せるデータの存在
`POLLPRI`	緊急の読み出しデータの存在
`POLLOUT`	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
`POLLERR`	何らかのエラー状態
`POLLHUP`	ハングアップ
`POLLNVAL`	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 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。

15.2 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。

stack_size([*size*])

新しいスレッドが作られる際に使われるスレッドのスタックサイズを返します。オプションの *size* 引数は次に作られるスレッドに対するスタックサイズを指定するものですが、0 (プラットフォームまたは設定されたデフォルト) または少なくとも 32,768 (32kB) であるような正の整数でなければなりません。もしスタックサイズの変更がサポートされていなければ ThreadError が送出されます。また指定されたスタックサイズが条件を満たしていなければ ValueError が送出されスタックサイズは変更されないままになります。32kB は今のところインタプリタ自体に十分なスタックスペースを保証するための値としてサポートされる最小のスタックサイズです。プラットフォームによってはスタックサイズの値に固有の制限が課されることもあります。たとえば 32kB より大きな最小スタックサイズを要求されたり、システムメモリサイズの倍数の割り当てを要求されるなどです - より詳しい情報はプラットフォームごとの文書で確認してください (4kB ページは一般的ですので、情報が見当たらないときには 4096 の倍数を指定しておくといいかもしれません)。利用可能: Windows, POSIX スレッドのあるシステム。 2.5 で追加された仕様です。

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って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、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が、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。

これらのメソッドに加えて、ロックオブジェクトは with 文を通じて以下の例のように使うこともできます。

```
from __future__ import with_statement
import thread

a_lock = thread.allocate_lock()

with a_lock:
    print "a_lock is locked while this executes"
```

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、それ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。

15.3 `threading` — 高水準のスレッドインタフェース

このモジュールでは、高水準のスレッドインタフェースをより低水準な `thread` モジュールの上に構築しています。

また、`thread` がないために `threading` を使えないような状況向けに `dummy_threading` を提供しています。

このモジュールでは以下のような関数とオブジェクトを定義しています:

`activeCount``()`
現在のアクティブな `Thread` オブジェクトの数を返します。この数は `enumerate()` の返すリストの長さと同じです。

`Condition``()`
新しい条件変数 (condition variable) オブジェクトを返すファクトリ関数です。条件変数を使うと、ある複数のスレッドを別のスレッドの通知があるまで待機させられます。

`currentThread``()`
関数を呼び出している処理のスレッドに対応する `Thread` オブジェクトを返します。関数を呼び出している処理のスレッドが `threading` モジュールで生成したものでない場合、限定的な機能しかもたないダミースレッドオブジェクトを返します。

`enumerate``()`
現在アクティブな `Thread` オブジェクト全てのリストを返します。リストには、デーモンスレッド (daemonic thread)、`currentThread()` の生成するダミースレッドオブジェクト、そして主スレッドが入ります。終了したスレッドとまだ開始していないスレッドは入りません。

`Event``()`
新たなイベントオブジェクトを返すファクトリ関数です。イベントは `set()` メソッドを使うと `True` に、`clear()` メソッドを使うと `False` にセットされるようなフラグを管理します。`wait()` メソッドは、全てのフラグが真になるまでブロックするようになっています。

class `local`
スレッドローカルデータ (thread-local data) を表現するためのクラスです。スレッドローカルデータとは、値が各スレッド固有になるようなデータです。スレッドローカルデータを管理するには、`local` (または `local` のサブクラス) のインスタンスを作成して、その属性に値を代入します:

```
mydata = threading.local()
mydata.x = 1
```

インスタンスの値はスレッドごとに違った値になります。

詳細と例題については、_threading_local モジュールのドキュメンテーション文字列を参照してください。

2.4 で追加された仕様です。

Lock()

新しいプリミティブロック (primitive lock) オブジェクトを返すファクトリ関数です。スレッドが一度プリミティブロックを獲得すると、それ以後のロック獲得の試みはロックが解放されるまでブロックします。どのスレッドでもロックを解放できます。

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 (semaphore) オブジェクトを返すファクトリ関数です。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き、初期値を足した値を表すカウンタを管理します。acquire() メソッドは、カウンタの値を負にせずに処理を戻せるまで必要ならば処理をブロックします。*value* を指定しない場合、デフォルトの値は 1 になります。

BoundedSemaphore([*value*])

新しい有限セマフォ (bounded semaphore) オブジェクトを返すファクトリ関数です。有限セマフォは、現在の値が初期値を超過しないようチェックを行います。超過を起こした場合、ValueError を送出します。たいていの場合、セマフォは限られた容量のリソースを保護するために使われるものです。従って、あまりにも頻繁なセマフォの解放はバグが生じているしるしです。*value* を指定しない場合、デフォルトの値は 1 になります。

class Thread

処理中の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

指定時間経過後に関数を実行するスレッドです。

settrace(*func*)

threading モジュールを使って開始した全てのスレッドにトレース関数 を設定します。*func* は各スレッドの run() を呼び出す前にスレッドの sys.settrace() に渡されます。 2.3 で追加された仕様です。

setprofile(*func*)

threading モジュールを使って開始し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() を呼び出す前にスレッドの sys.settrace() に渡されます。 2.3 で追加された仕様です。

stack_size([*size*])

新しいスレッドが作られる際に使われるスレッドのスタックサイズを返します。オプションの *size* 引数は次に作られるスレッドに対するスタックサイズを指定するものですが、0 (プラットフォームまたは設定されたデフォルト) または少なくとも 32,768 (32kB) であるような正の整数でなければなりません。もしスタックサイズの変更がサポートされていなければ ThreadError が送出されます。また指定されたスタックサイズが条件を満たしていなければ ValueError が送出されスタックサイズは変更されないままになります。32kB は今のところインタプリタ自体に十分なスタックスペースを保証するための値としてサポートされる最小のスタックサイズです。プラットフォームによってはスタックサイズの値に固有の制限が課されることもあります。たとえば 32kB より大きな最小スタック

サイズを要求されたり、システムメモリサイズの倍数の割り当てを要求されるなどです - より詳しい情報はプラットフォームごとの文書で確認してください (4kB ページは一般的ですので、情報が見当たらないときには 4096 の倍数を指定しておくといいかもしれません)。利用可能: Windows, POSIX スレッドのあるシステム。 2.5 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。

このモジュールのおおまかな設計は Java のスレッドモデルに基づいています。とはいえ、Java がロックと条件変数を全てのオブジェクトの基本的な挙動にしているのに対し、 Python ではこれらを別個のオブジェクトに分けています。Python の `Thread` クラスがサポートしているのは Java の Thread クラスの挙動のサブセットにすぎません; 現状では、優先度 (priority) やスレッドグループがなく、スレッドの破壊 (destroy)、中断 (stop)、一時停止 (suspend)、復帰 (resume)、割り込み (interrupt) は行えません。Java の Thread クラスにおける静的メソッドに対応する機能が実装されている場合には、、モジュールレベルの関数になっています。

以下に説明するメソッドは全て原子的 (atomic) に実行されます。

15.3.1 Lock オブジェクト

プリミティブロックとは、ロックが生じた際に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 では現在のところ拡張モジュール `thread` で直接実装されている最も低水準の同期プリミティブを使えます。

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” があります。このロックはアンロック状態で作成されます。ロックには基本となる二つのメソッド、`acquire()` と `release()` があります。ロックの状態がアンロックである場合、`acquire()` は状態をロックに変更して即座に処理を戻します。状態がロックの場合、`acquire()` は他のスレッドが `release()` を呼出してロックの状態をアンロックに変更するまでブロックします。その後、状態をロックに再度設定してから処理を戻します。`release()` メソッドを呼び出すのはロック状態のときでなければなりません; このメソッドはロックの状態をアンロックに変更し、即座に処理を戻します。複数のスレッドにおいて `acquire()` がアンロック状態への遷移を待っているためにブロックが起きている時に `release()` を呼び出してロックの状態をアンロックにすると、一つのスレッドだけが処理を進行できます。どのスレッドが処理を進行できるのかは定義されておらず、実装によって異なるかもしれません。

全てのメソッドは原子的に実行されます。

acquire([*blocking = 1*])

ブロックあり、またはブロックなしでロックを獲得します。

引数なしで呼び出した場合、ロックの状態がアンロックになるまでブロックし、その後状態をロックにセットして真値を返します。

引数 *blocking* の値を真にして呼び出した場合、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* の値を偽にして呼び出すとブロックしません。引数なしで呼び出した場合にブロックするような状況であった場合には直ちに偽を返します。それ以外の場合には、引数なしで呼び出したときと同じ処理を行い真を返します。

release()

ロックを解放します。

ロックの状態がロックのとき、状態をアンロックにリセットして処理を戻します。他のスレッドがロックがアンロック状態になるのを待ってブロックしている場合、ただ一つのスレッドだけが処理を継続できるようにします。

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なりません。

戻り値はありません。

15.3.2 RLock オブジェクト

再入可能ロック (reentrant lock) とは、同じスレッドが複数回獲得できるような同期プリミティブです。再入可能ロックの内部では、プリミティブロックの使うロック／アンロック状態に加え、“所有スレッド (owning thread)” と “再帰レベル (recursion level)” という概念を用いています。ロック状態では何らかのスレッドがロックを所有しており、アンロック状態ではいかなるスレッドもロックを所有していません。

スレッドがこのロックの状態をロックにするには、ロックの `acquire()` メソッドを呼び出します。このメソッドは、スレッドがロックを所有すると処理を戻します。ロックの状態をアンロックにするには `release()` メソッドを呼び出します。`acquire()`/`release()` からなるペアの呼び出しはネストできます; 最後に呼び出した `release()` (最も外側の呼び出しペア) だけが、ロックの状態をアンロックにリセットし、`acquire()` でブロック中の別のスレッドの処理を進行させられます。

acquire([*blocking = 1*])

ブロックあり、またはブロックなしでロックを獲得します。

引数なしで呼び出した場合: スレッドが既にロックを所有している場合、再帰レベルをインクリメントして即座に処理を戻します。それ以外の場合、他のスレッドがロックを所有していれば、そのロックの状態がアンロックになるまでブロックします。その後、ロックの状態がアンロックになる (いかなるスレッドもロックを所有しない状態になる) と、ロックの所有権を獲得し、再帰レベルを 1 にセットして処理を戻します。ロックの状態がアンロックになるのを待っているスレッドが複数ある場合、その中の一つだけがロックの所有権を獲得できます。この場合、戻り値はありません。

blocking 引数の値を真にした場合、引数なしで呼び出した場合と同じ処理を行って真を返します。

blocking 引数の値を偽にした場合、ブロックしません。引数なしで呼び出した場合にブロックするような状況であった場合には直ちに偽を返します。それ以外の場合には、引数なしで呼び出したときと同じ処理を行い真を返します。

release()

再帰レベルをデクリメントしてロックを解放します。デクリメント後に再帰レベルがゼロになった場合、ロックの状態をアンロック (いかなるスレッドにも所有されていない状態) にリセットし、ロックの状態がアンロックになるのを待ってブロックしているスレッドがある場合にはその中のただ一つだけが処理を進行できるようにします。デクリメント後も再帰レベルがゼロでない場合、ロックの状態はロックのままで、呼び出し手のスレッドに所有されたままになります。

呼び出し手のスレッドがロックを所有しているときにのみこのメソッドを呼び出してください。ロックの状態がアンロックの時にこのメソッドを呼び出してはなりません。

戻り値はありません。

15.3.3 Condition オブジェクト

条件変数 (condition variable) は常にある種のロックに関連付けられています; 条件変数に関連付けるロックは明示的に引き渡したり、デフォルトで生成させたりできます。(複数の条件変数で同じロックを共有するような場合には、引渡しによる関連付けが便利です。)

条件変数には、`acquire()` メソッドおよび `release()` があり、関連付けされているロックの対応するメソッドを呼び出すようになっています。また、`wait()`, `notify()`, `notifyAll()` といったメソッドがあります。これら三つのメソッドを呼び出せるのは、呼び出し手のスレッドがロックを獲得している時だけです。

wait() メソッドは現在のスレッドのロックを解放し、他のスレッドが同じ条件変数に対して notify() または notifyAll() を呼び出して現在のスレッドを起こすまでブロックします。一度起こされると、再度ロックを獲得して処理を戻します。wait() にはタイムアウトも設定できます。

notify() メソッドは条件変数待ちのスレッドを 1 つ起こします。notifyAll() メソッドは条件変数待ちの全てのスレッドを起こします。

注意: notify() と notifyAll() はロックを解放しません; 従って、スレッドが起こされたとき、wait() の呼び出しは即座に処理を戻すわけではなく、notify() または notifyAll() を呼び出したスレッドが最終的にロックの所有権を放棄したときに初めて処理を返すのです。

豆知識: 条件変数を使う典型的なプログラミングスタイルでは、何らかの共有された状態変数へのアクセスを同期させるためにロックを使います; 状態変数が特定の状態に変化したことを知りたいスレッドは、自分の望む状態になるまで繰り返し wait() を呼び出します。その一方で、状態変更を行うスレッドは、前者のスレッドが待ち望んでいる状態であるかもしれないような状態へ変更を行ったときに notify() や notifyAll() を呼び出します。例えば、以下のコードは無制限のバッファ容量のときの一般的な生産者-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() のどちらを使うかは、その状態の変化に興味を持っている待ちスレッドが一つだけなのか、あるいは複数なのかで考えます。例えば、典型的な生産者-消費者問題では、バッファに 1 つの要素を加えた場合には消費者スレッドを 1 つしか起こさなくてかまいません。

class Condition(*[lock]*)

lock を指定して、None の値にする場合、Lock または RLock オブジェクトでなければなりません。この場合、*lock* は根底にあるロックオブジェクトとして使われます。それ以外の場合には新しい RLock オブジェクトを生成して使います。

acquire(**args*)

根底にあるロックを獲得します。このメソッドは根底にあるロックの対応するメソッドを呼び出します。そのメソッドの戻り値を返します。

release()

根底にあるロックを解放します。このメソッドは根底にあるロックの対応するメソッドを呼び出します。戻り値はありません。

wait(*[timeout]*)

通知 (notify) を受けるか、タイムアウトするまで待機します。このメソッドを呼び出してよいのは、呼び出し手のスレッドがロックを獲得しているときだけです。

このメソッドは根底にあるロックを解放し、他のスレッドが同じ条件変数に対して notify() または notifyAll() を呼び出して現在のスレッドを起こすか、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。一度スレッドが起こされると、再度ロックを獲得して処理を戻します。

timeout 引数を指定して、None 以外の値にする場合、タイムアウトを秒 (または端数秒) を表す浮動

小数点数でなければなりません。

根底にあるロックが RLock である場合、release() メソッドではロックは解放されません。というのも、ロックが再帰的に複数回獲得されている場合には、release() によって実際にアンロックが行われないかもしれないからです。その代わり、 ロックが再帰的に複数回獲得されていても確実にアンロックを行える RLock クラスの内部インタフェースを使います。その後ロックを再獲得する時に、もう一つの内部インタフェースを使ってロックの再帰レベルを復帰します。

notify()

この条件変数を待っているスレッドがあれば、そのスレッドを起こします。このメソッドを呼び出してよいのは、呼び出し手のスレッドがロックを獲得しているときだけです。

何らかの待機中スレッドがある場合、そのスレッドの一つを起こします。待機中のスレッドがなければ何もしません。

現在の実装では、待機中のメソッドをただ一つだけ起こします。とはいえ、この挙動に依存するのは安全ではありません。将来、実装の最適化によって、複数のスレッドを起こすようになるかもしれないからです。

注意: 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ので、notify() 呼び出し手は明示的にロックを解放せねばなりません。

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() のように動作しますが、1 つではなくすべての待ちスレッドを起こします。

15.3.4 Semaphore オブジェクト

セマフォ (semaphore) は、計算機科学史上最も古い同期プリミティブの一つで、草創期のオランダ計算機科学者 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ような内部カウンタを管理します。カウンタは決してゼロより小さくはなりません; acquire() は、カウンタがゼロになっている場合、他のスレッドが release() を呼び出すまでブロックします。

class **Semaphore**([*value*])

オプションの引数には、内部カウンタの初期値を指定します。デフォルトは 1 です。

acquire([*blocking*])

セマフォを獲得します。

引数なしで呼び出した場合: acqure() 処理に入ったときに内部カウンタがゼロより大きければ、カウンタを 1 デクリメントして即座に処理を戻します。acqure() 処理に入ったときに内部カウンタがゼロの場合、他のスレッドが release() を呼び出してカウンタをゼロより大きくするまでブロックします。この処理は、適切なインターロック (interlock) を介して行い、複数の acquire() 呼び出しがブロックされた場合、release() が正確に一つだけを起こせるようにします。この実装はランダムに一つ選択するだけでもよいので、ブロックされたスレッドがどの起こされる順番に依存してはなりません。この場合、戻り値はありません。

blocking 引数の値を真にした場合、引数なしで呼び出した場合と同じ処理を行って真を返します。

blocking 引数の値を偽にした場合、ブロックしません。引数なしで呼び出した場合にブロックするような状況であった場合には直ちに偽を返します。それ以外の場合には、引数なしで呼び出したときと同じ処理を行い真を返します。

release()
内部カウンタを 1 インクリメントして、セマフォを解放します。release() 処理に入ったときにカウンタがゼロであり、カウンタの値がゼロより大きくなるのを待っている別のスレッドがあった場合、そのスレッドを起こします。

Semaphore の例

セマフォはしばしば、容量に限りのある資源、例えばデータベースサーバなどを保護するために使われます。リソースのサイズが固定の状況では、常に有限セマフォを使わねばなりません。主スレッドは、作業スレッドを立ち上げる前にセマフォを初期化します:

```
maxconnections = 5
...
pool_sema = BoundedSemaphore(value=maxconnections)
```

作業スレッドは、ひとたび立ち上がると、サーバへ接続する必要が生じたときにセマフォの acquire および release メソッドを呼び出します:

```
pool_sema.acquire()
conn = connectdb()
... use connection ...
conn.close()
pool_sema.release()
```

有限セマフォを使うと、セマフォを獲得回数以上に解放してしまうというプログラム上の間違いを見逃しにくくします。

15.3.5 Event オブジェクト

イベントは、あるスレッドがイベントを発信し、他のスレッドはそれを待つという、スレッド間で通信を行うための最も単純なメカニズムの一つです。

イベントオブジェクトは内部フラグを管理します。このフラグは set() メソッドで値を真に、clear() メソッドで値を偽にリセットします。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**()
内部フラグの初期値は偽です。

isSet()
内部フラグの値が真である場合かつその場合にのみ真を返します。

set()
内部フラグの値を真にセットします。フラグの値が真になるのを待っている全てのスレッドを起こします。一旦フラグが真になると、スレッドが wait() を呼び出しても全くブロックしなくなります。

clear()
内部フラグの値を偽にリセットします。以降は、set() を呼び出して再び内部フラグの値を真にセットするまで、wait() を呼出したスレッドはブロックするようになります。

wait([*timeout*])
内部フラグの値が真になるまでブロックします。wait() 処理に入った時点で内部フラグの値が真であれば、直ちに処理を戻します。そうでない場合、他のスレッドが set() を呼び出してフラグの値を真にセットするか、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 引数を指定して、None 以外の値にする場合、タイムアウトを秒 (または端数秒) を表す浮動小数点数でなければなりません。

15.3.6 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 (activity) を表現します。活動を決める方法はは 2 つあり、一つは呼出し可能オブジェクトをコンストラクタへ渡す方法、もう一つはサブクラスで run() メソッドをオーバライドする方法です。(コンストラクタを除く) その他のメソッドは一切サブクラスでオーバライドしてはなりません。言い換えるならば、このクラスの__init__() と run() メソッド**だけ**をオーバライドしてくださいということです。

ひとたびスレッドオブジェクトを生成すると、スレッドの start() メソッドを呼び出して活動を開始せねばなりません。start() メソッド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、スレッドは ' 生存中 (alive)' で、' 活動中 (active)' とみなされます (これら二つの概念はほとんど同じですが、全く同じというわけではありません; これら二つは意図的に曖昧に定義されているのです)。スレッドの活動は、通常終了、あるいは処理されない例外が送出されたことで run() メソッドが終了すると生存中でなくなり、かつ活動中でなくなります。isAlive() メソッドはスレッドが生存中であるかどうか調べます。

他のスレッドはスレッドの join() メソッドを呼び出せます。このメソッドは、join() を呼び出されたスレッドが終了するまで、メソッドの呼び出し手となるスレッドをブロックします。

スレッドには名前があります。名前はコンストラクタで渡したり、setName() メソッドで設定したり、getName() メソッドで取得したりできます。

スレッドには "デーモンスレッド (daemon thread)" であるというフラグを立てられます。このフラグには、残っているスレッドがデーモンスレッドだけになった時に Python プログラム全体を終了させるという意味があります。フラグの初期値はスレッドを生成する側のスレッドから継承します。フラグの値は setDaemon() メソッドで設定でき、isDaemon() メソッドで取得できます。

スレッドには "主スレッド (main thread)" オブジェクトがあります。主スレッドは Python プログラムを最初に制御していたスレッドです。主スレッドはデーモンスレッドではありません。

"ダミースレッド (dumm thread)" オブジェクトを作成できる場合があります。ダミースレッドは、 "外来スレッド (alien thread)" に相当するスレッドオブジェクトです。ダミースレッドは、C コードから直接生成されたスレッドのような、 threading モジュールの外で開始された処理スレッドです。ダミースレッドオブジェクトには限られた機能しかなく、常に生存中、活動中かつデーモンスレッドであるとみなされ、join() できません。また、外来スレッドの終了を検出するのは不可能なので、ダミースレッドは削除できません。

class Thread(*group=None, target=None, name=None, args=(), kwargs={}*)

コンストラクタは常にキーワード引数を使って呼び出さねばなりません。各引数は以下の通りです:

group は None にせねばなりません。将来 ThreadGroup クラスが実装されたときの拡張用に予約されている引数です。

target は run() メソッドによって起動される呼出し可能オブジェクトです。 デフォルトでは何も呼び出さないことを示す None になっています。

name はスレッドの名前です。デフォルトでは、 *N* を小さな 10 進数として、"Thread-*N*" という形式の一意な名前を生成します。

args は *target* を呼び出すときの引数タプルです。デフォルトは () です。

kwargs は *target* を呼び出すときのキーワード引数の辞書です。デフォルトは{}です。

サブクラスでコンストラクタをオーバライドした場合、必ず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__()`) を呼び出しておかなくてはなりません。

start()
スレッドの活動を開始します。

このメソッドは、スレッドオブジェクトあたり一度しか呼び出してはなりません。`start()` は、オブジェクトの `run()` メソッドが個別の処理スレッド中で呼び出されるように調整します。

run()
スレッドの活動をもたらすメソッドです。

このメソッドはサブクラスでオーバライドできます。標準の `run()` メソッドでは、オブジェクトのコンストラクタの *target* 引数に呼び出し可能オブジェクトを指定した場合、*args* および *kwargs* の引数列およびキーワード引数とともに呼び出します。

join([*timeout*])
スレッドが終了するまで待機します。このメソッドは、`join()` を呼び出されたスレッドが、正常終了あるいは処理されない例外によって終了するか、オプションのタイムアウトが発生するまで、メソッドの呼び出し手となるスレッドをブロックします。

timeout 引数を指定して、`None` 以外の値にする場合、タイムアウトを秒 (または端数秒) を表す浮動小数点数でなければなりません。`join()` はいつでも `None` を返すので、`isAlive()` を呼び出してタイムアウトしたかどうかを確認しなければなりません。

timeout が指定されないかまたは `None` であるときは、この操作はスレッドが終了するまでブロックします。

一つのスレッドに対して何度でも `join()` できます。

スレッドは自分自身を `join()` できません。デッドロックを引き起こすからです。

スレッドを開始するまえに `join()` を試みるのは誤りです。

getName()
スレッドの名前を返します。

setName(*name*)
スレッドの名前を設定します。

名前は識別のためだけに使われます。名前には機能上の意味づけ (semantics) はありません。複数のスレッドに同じ名前をつけてもかまいません。名前の初期値はコンストラクタで設定されます。

isAlive()
スレッドが生存中かどうかを返します。

大雑把な言い方をすると、スレッドは `start()` メソッドを呼び出した瞬間から `run()` メソッドが終了するまでの間生存しています。

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ブール値 *daemonic* に設定します。このメソッドは `start()` を呼び出す前に呼び出さねばなりません。

初期値は生成側のスレッドから継承されます。

デーモンでない活動中のスレッドが全てなくなると、Python プログラム全体が終了します。

15.3.7 Timer オブジェクト

このクラスは、一定時間経過後に実行される活動、すなわちタイマ活動を表現します。Timer は Thread のサブクラスであり、自作のスレッドを構築した一例でもあります。

タイマは start() メソッドを呼び出すとスレッドとして作動し始めします。(活動を開始する前に) cancel() メソッドを呼び出すと、タイマを停止できます。タイマが活動を実行するまでの待ち時間は、ユーザが指定した待ち時間と必ずしも厳密には一致しません。

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 秒後に *function* を引数 *args*、キーワード引数 *kwargs* つきで実行するようなタイマを生成します。

cancel()

タイマをストップして、その動作の実行をキャンセルします。このメソッドはタイマがまだ活動待ち状態にある場合にのみ動作します。

15.3.8 with 文でのロック・条件変数・セマフォの使い方

このモジュールのオブジェクトで acquire() と release() 両メソッドを具えているものは全て with 文のコンテキストマネージャとして使うことができます。acquire() メソッドが with 文のブロックに入るときに呼び出され、ブロック脱出時には release() メソッドが呼ばれます。

現在のところ、Lock、RLock、Condition、Semaphore、BoundedSemaphore を with 文のコンテキストマネージャとして使うことができます。以下の例を見てください。

```
from __future__ import with_statement
import threading

some_rlock = threading.RLock()

with some_rlock:
    print "some_rlock is locked while this executes"
```

15.4 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。

15.5 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。

15.6 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込みや書込み、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 上とでは異なる `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。 2.5 で変更された仕様: 無名メモリ (anonymous memory) をマップするためには fileno として-1 を渡して長さを与えてください

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* が現在のファイルサイズより大きな場合、ファイルサイズは *length* を含む大きさにまで拡張されます。*length* が 0 の場合、マップの最大の長さは Windows が空ファイルで例外を起こす (Windows では空のマップを作成することができません。) ことを除いては、mmap() が呼ばれたときのファイルサイズになります。

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。*length* が 0 の場合、そのマップの最大長が現在のファイルサイズになります。

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、読込み可能または書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。

メモリマップフィ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、例外の送出を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* は探索を始めたい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*)
マップと元ファイルのサイズを変更します。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。

15.7 readline — GNU readline のインタフェース

readline モジュールでは、補完をしやすくしたり、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います。このモジュールは直接使うことも rlcompleter モジュールを介して使うこともできます。このモジュールで利用される設定はインタプリタの対話プロンプトの振舞い、組み込みの raw_input() と input() 関数の振舞いに影響します。

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。

clear_history()
現在のヒストリをクリアします。 (注意:インストールされている GNU readline がサポートしていない場合、この関数は利用できません) 2.4 で追加された仕様です。

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詰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。

get_current_history_length()
現在のヒストリ行数を返します (この値は `get_history_length()` で取得する異なります。`get_history_length()` はヒストリファイルに書き出される最大行数を返します)。 2.3 で追加された仕様です。

get_history_item(*index*)
現在のヒストリから、*index* 番目の項目を返します。 2.3 で追加された仕様です。

remove_history_item(*pos*)
ヒストリから指定した位置にあるヒストリを削除します。 2.4 で追加された仕様です。

replace_history_item(*pos, line*)
指定した位置にあるヒストリを、指定した line で置き換えます。 2.4 で追加された仕様です。

redisplay()
画面の表示を、現在のヒストリ内容によって更新します。 2.3 で追加された仕様です。

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 (15.8 節):
対話的プロンプトで Python 識別子を補完する機能。

15.7.1 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
```

次の例では `code.InteractiveConsole` クラスを拡張し、ヒストリの保存・復旧をサポートします。

```
import code
import readline
import atexit
import os

class HistoryConsole(code.InteractiveConsole):
    def __init__(self, locals=None, filename="<console>",
                 histfile=os.path.expanduser("~/.console-history")):
        code.InteractiveConsole.__init__(self)
        self.init_history(histfile)

    def init_history(self, histfile):
        readline.parse_and_bind("tab: complete")
        if hasattr(readline, "read_history_file"):
            try:
                readline.read_history_file(histfile)
            except IOError:
                pass
            atexit.register(self.save_history, histfile)

    def save_history(self, histfile):
        readline.write_history_file(histfile)
```

15.8 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。

このモジュールが UNIX プラットフォームで import され、readline が利用できるときには、Completer クラスのインスタンスが自動的に作成され、complete メソッドが readline 補完に設定されます。

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")
```

readline のないプラットフォームでも、このモジュールで定義される Completer クラスは独自の目的に使えます。

15.8.1 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。

十六章

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。

posix	最も一般的な POSIX システムコール群 (通常は os モジュールを介して利用されます)。
pwd	パスワードデータベースへのアクセスを提供する (getpwnam() など)。
spwd	シャドウパスワードデータベース (getspnam() など
grp	グループデータベースへのアクセス (getgrnam() およびその仲間)。
crypt	UNIX パスワードをチェックするための関数 crypt()。
dl	共有オブジェクトの C 関数の呼び出し
termios	POSIX スタイルの端末制御。
tty	一般的な端末制御操作のためのユーティリティ関数群。
pty	SGI と Linux 用の擬似端末を制御する
fcntl	fcntl() および ioctl() システムコール。
pipes	Python による UNIX シェルパイプラインへのインタフェース。
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。
resource	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。
nis	Sun の NIS (Yellow Pages) ライブラリへのインタフェース。
syslog	UNIX syslog ライブラリルーチン群へのインタフェース。
commands	外部コマンドを実行するためのユーティリティです。

16.1 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。

16.1.1 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
```

16.1.2 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。

16.2 `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 (crypy モジュールをごらんください) 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。*pw_passwd* フィールドに有用な値が入っているかはシステムに依存します。利用可能なら、暗号化されたパスワードへのアクセスが必要なときには spwd モジュールを利用してください。

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 (16.4 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。

spwd モジュール (16.3 節):
このモジュールに似た、シャドウパスワードデータベースへのアクセスを提供するモジュール。

16.3 spwd — シャドウパスワードデータベース

2.5 で追加された仕様です。

このモジュールは UNIX のシャドウパスワードデータベースへのアクセスを提供します。様々な UNIX 環境で利用できます。

シャドウパスワードデータベースへアクセスできる権限が必要 (大抵の場合 root である必要があります) です。

シャドウパスワードデータベースのエントリはタプル状のオブジェクトで提供され、その属性は spwd 構造のメンバーに対応しています（以下を参照してください。<shadow.h>を参照):

Index	Attribute	Meaning
0	sp_nam	ログイン名
1	sp_pwd	暗号化されたパスワード
2	sp_lstchg	最終更新日
3	sp_min	パスワード変更が出来るようになるまでの最小日数
4	sp_max	パスワードを変更しなくても良い最大日数
5	sp_warn	パスワードが期限切れになる前に、期限切れが近づいている旨の警告をユーザに出しはじめる日数
6	sp_inact	パスワードが期限切れになってから、アカウントが inactive となり使用できなくなるまでの日数
7	sp_expire	1970-01-01 からアカウントが使用できなくなるまでの日数
8	sp_flag	将来のために予約

sp_nam と *sp_pwd* は文字列で、他は全て整数です。

エントリが見つからなかった時は KeyError が起きます。

このモジュールでは以下を定義しています:

getspnam(*name*)
与えられたユーザ名に対応するシャドウパスワードデータベースのエントリを返します。

getspall()
利用可能なシャドウパスワードデータベースの全エントリを任意の順番で返します。

参考資料:

grp モジュール (16.4 節):
このモジュールに似たグループデータベースへのインタフェース

pwd モジュール (16.2 節):
このモジュールに似た通常のパスワードデータベースへのインタフェース

16.4 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ら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せん。

参考資料:

pwd モジュール (16.2 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。

spwd モジュール (16.3 節):
このモジュールと類似の、シャドウパスワードデータベースへのインタフェース。

16.5 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。

このモジュールは実行環境の *crypt*(3) の実装に依存しています。そのため、現在の実装で利用可能な拡張を、このモジュールでもそのまま利用できます。

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).

いくつかの拡張された *crypt*(3) は異なる値と *salt* の長さを許しているので、パスワードをチェックする際には crypt されたパスワード文字列全体を *salt* として渡すよう勧めます。

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:')
    cryptedpasswd = pwd.getpwnam(username)[1]
    if cryptedpasswd:
        if cryptedpasswd == 'x' or cryptedpasswd == '*':
            raise "Sorry, currently no support for shadow passwords"
        cleartext = getpass.getpass()
        return crypt.crypt(cleartext, cryptedpasswd) == cryptedpasswd
    else:
        return 1
```

16.6 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

任意の関数を呼ぶプログラムを与えます。

警告: dl モジュールは Python の型システムとエラー処理をバイパスしています。もし間違って使用すれば、セグメンテーションフォルト、クラッシュ、その他の不正な動作を起こします。

注意: このモジュールは sizeof(int) == sizeof(long) == sizeof(char *) でなければ働きません。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。

16.6.1 Dl オブジェクト

open() によって返された Dl 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で、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。

16.7 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無視し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 (16.8 節):
一般的な端末制御操作のための便利な関数。

16.7.1 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
```

16.8 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 termios.TCSAFLUSH 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 termios.TCSAFLUSH になり、termios.tcsetattr() に渡されます。

参考資料:

termios モジュール (16.7 節):
低レベル端末制御インタフェース。

16.9 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。返り値は (*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 バイトずつ読み込もうとします。

16.10 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、純粋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。引数が無いか整数値の場合、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、オペレーティングシステムが希望するバッファ長までであれば正しく動作します。

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。

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,* [*length,* [*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, os

f = open(...)
rv = fcntl.fcntl(f, fcntl.F_SETFL, os.O_NDELAY)

lockdata = struct.pack('hhllhh', fcntl.F_WRLCK, 0, 0, 0, 0, 0)
rv = fcntl.fcntl(f, fcntl.F_SETLKW, lockdata)
```

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。

参考資料:

`os` モジュール (14.1 節):

もし `O_SHLOCK` と `O_EXLOCK` が `os` モジュールに存在する場合、`os.open()` 関数は `lockf()` や `flock()` 関数よりもよりプラットフォーム独立なロック機構を提供します。

16.11 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'
```

16.11.1 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。

16.12 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 None 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()
```

16.13 `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。

16.13.1 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*)

resouce 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。

16.13.2 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。

16.14 `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*[, *domain=default_domain*])

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。

2.5 で変更された仕様: *domain* 引数で参照する NIS ドメインをオーバーライドできます。設定されない場合にはデフォルトの NIS ドメインを参照します。

cat(*mapname*[, *domain*=*default_domain*])

`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。

2.5 で変更された仕様: *domain* 引数で参照する NIS ドメインをオーバーライドできます。設定されない場合にはデフォルトの NIS ドメインを参照します。

maps()

有効なマップのリストを返します。

get_default_domain()

システムのデフォルト NIS ドメインをかえします。 2.5 で追加された仕様です。

`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。

16.15 `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) *indent* = `'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。

16.16 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'
```

十七章

プロセス間通信とネットワーク

この章で解説されるモジュールは他のプロセスと通信するメカニズムを提供します。

いくつかのモジュール、たとえば signal や subprocess は同じマシン上での 2 つのプロセス間でだけ動作します。他のモジュールはネットワークプロトコルをサポートし、2 つかそれ以上のプロセスがマシンをまたいで通信するために利用できます。

この章で解説されるモジュールの一覧は:

subprocess	サブプロセス管理
socket	低レベルネットワークインターフェース。
signal	非同期イベントにハンドラを設定します。
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。
asyncore	非同期なソケット制御サービスのためのベースクラス
asynchat	非同期コマンド/レスポンスプロトコルの開発サポート

17.1 subprocess — サブプロセス管理

2.4 で追加された仕様です。

subprocess モジュールは、新しくプロセスを開始したり、それらの標準入出力/エラー出力に対してパイプで接続したり、それらの終了ステータスを取得したりします。このモジュールは以下のような古いいくつかのモジュールを置き換えることを目的としています:

```
os.system
os.spawn*
os.popen*
popen2.*
commands.*
```

これらのモジュールや関数の代わりに、subprocess モジュールをどのように使うかについては以下の節で説明します。

17.1.1 subprocess モジュールを使う

このモジュールでは Popen と呼ばれるクラスを定義しています:

class Popen(*args, bufsize=0, executable=None, stdin=None, stdout=None, stderr=None, preexec_fn=None, close_fds=False, shell=False, cwd=None, env=None, universal_newlines=False, startupinfo=None, creationflags=0*)

各引数の説明は以下のとおりです:

args は文字列か、あるいはプログラムへの引数のシーケンスである必要があります。実行するプログラムは通常 args シーケンスあるいは文字列の最初の要素ですが、executable 引数を使うことにより明

示的に指定することもできます。

UNIX で *shell=False* の場合 (デフォルト): この場合、Popen クラスは子プログラムを実行するのに `os.execvp()` を使います。*args* は通常シーケンスでなければなりません。文字列の場合はひとつだけの文字列要素 (= 実行するプログラム名) をもったシーケンスとして扱われます。

UNIX で *shell=True* の場合: args が文字列の場合、これはシェルを介して実行されるコマンドライン文字列を指定します。*args* がシーケンスの場合、その最初の要素はコマンドライン文字列となり、それ以降の要素はすべてシェルへの追加の引数として扱われます。

Windows の場合: `Popen` クラスは子プログラムを実行するのに文字列の扱える CreateProcess() を使います。*args* がシーケンスの場合、これは `list2cmdline` メソッドをつかってコマンドライン文字列に変換されます。注意: すべての MS Windows アプリケーションがコマンドライン引数を同じやりかたで解釈するとは限りません。`list2cmdline` は MS C ランタイムと同じやりかたで文字列を解釈するアプリケーション用に設計されています。

bufsize は、もしこれが与えられた場合、ビルトインの open() 関数の該当する引数と同じ意味をもちます: `0` はバッファされないことを意味し、`1` は行ごとにバッファされることを、それ以外の正の値は (ほぼ) その大きさのバッファが使われることを意味します。負の *bufsize* はシステムのデフォルト値が使われることを意味し、通常これはバッファがすべて有効となります。*bufsize* のデフォルト値は `0` (バッファされない) です。

executable 引数には実行するプログラムを指定します。これはほとんど必要ありません: ふつう、実行するプログラムは *args* 引数で指定されるからです。`shell=True` の場合、 *executable* 引数は使用するシェルを指定します。 UNIX では、デフォルトのシェルは '/bin/sh' です。Windows では、デフォルトのシェルは COMSPEC 環境変数で指定されます。

stdin、 *stdout* および *stderr* には、実行するプログラムの標準入力、標準出力、および標準エラー出力のファイルハンドルをそれぞれ指定します。とりうる値は `PIPE` 、既存のファイル記述子 (正の整数) 、既存のファイルオブジェクト、そして `None` です。`PIPE` を指定すると新しいパイプが子プロセスに向けて作られます。`None` を指定するとリダイレクトは起こりません。子プロセスのファイルハンドルはすべて親から受け継がれます。加えて、*stderr* を `STDOUT` にすると、アプリケーションの stderr からの出力は stdout と同じファイルハンドルに出力されます。

preexec_fn に callable オブジェクトが指定されている場合、このオブジェクトは子プロセスが起動されてから、プログラムが exec される直前に呼ばれます。(UNIX のみ)

close_fds が真の場合、子プロセスが実行される前に `0`、 `1` および `2` をのぞくすべてのファイル記述子が閉じられます。(UNIX のみ)

shell が `True` の場合、指定されたコマンドはシェルを介して実行されます。

cwd が `None` 以外の場合、子プロセスのカレントディレクトリが実行される前に *cwd* に変更されます。このディレクトリは実行ファイルを探す段階では考慮されませんので、プログラムのパスを *cwd* に対する相対パスで指定することはできない、ということに注意してください。

env が `None` 以外の場合、これは新しいプロセスでの環境変数を定義します。

universal_newlines が `True` の場合、stdout および stderr のファイルオブジェクトはテキストファイルとして open されますが、行の終端は UNIX 形式の行末 `'\n'` か、Macintosh 形式の行末 `'\r'` か、あるいは Windows 形式の行末`'\r\n'` のいずれも許されます。これらすべての外部表現は Python プログラムには `'\n'` として認識されます。**注意:** この機能は Python に universal newline がサポートされている場合 (デフォルト) にのみ有効です。また、 `stdout`, `stdin` および `stderr` のファイルオブジェクトの newlines 属性は communicate() メソッドでは更新されません。

startupinfo および *creationflags* が与えられた場合、これらは内部で呼びだされる CreateProcess() 関数に渡されます。これらはメインウインドウの形状や新しいプロセスの優先度などを指定することがで

きます。 (Windows のみ)

便利な関数

このモジュールは二つのショートカット関数も定義しています:

call(**popenargs, **kwargs*)
コマンドを指定された引数で実行し、そのコマンドが完了するのを待って、returncode 属性を返します。

この引数は Popen コンストラクタの引数と同じです。使用例:

```
retcode = call(["ls", "-l"])
```

check_call(**popenargs, **kwargs*)
コマンドを引数付きで実行します。コマンドが完了するのを待ちます。終了コードがゼロならば終わりますが、そうでなければ CalledProcessError 例外を送出します。CalledProcessError オブジェクトにはリターンコードが returncode 属性として収められています。

引数は Popen のコンストラクタと一緒です。使用例:

```
check_call(["ls", "-l"])
```

例外

子プロセス内で raise した例外は、新しいプログラムが実行される前であれば、親プロセスでも raise されます。さらに、この例外オブジェクトには child_traceback という属性が追加されており、これには子プロセスの視点からの traceback 情報が格納されています。

もっとも一般的に起こる例外は OSError です。これは、たとえば存在しないファイルを実行しようとしたときなどに発生します。アプリケーションは OSError 例外にはあらかじめ準備しておく必要があります。

不適当な引数で Popen が呼ばれた場合は、ValueError が発生します。

check_call() はもし呼び出されたプロセスがゼロでないリターンコードを返したならば CalledProcessError を送出します。

セキュリティ

ほかの popen 関数とは異なり、この実装は決して暗黙のうちに /bin/sh を実行しません。これはシェルのメタ文字をふくむすべての文字が安全に子プロセスに渡されるということを意味しています。

17.1.2 Popen オブジェクト

Popen クラスのインスタンスには、以下のようなメソッドがあります:

poll()
子プロセスが終了しているかどうかを検査します。returncode 属性を返します。

wait()
子プロセスが終了するまで待ちます。returncode 属性を返します。

communicate(*input=None*)
プロセスと通信します: end-of-file に到達するまでデータを stdin に送信し、stdout および stderr から

データを受信します。プロセスが終了するまで待ちます。オプション引数 *input* には子プロセスに送られる文字列か、あるいはデータを送らない場合は `None` を指定します。

communicate() はタプル (stdout, stderr) を返します。

注意: 受信したデータはメモリ中にバッファされます。そのため、返されるデータが大きいかあるいは制限がないような場合はこのメソッドを使うべきではありません。

以下の属性も利用できます:

stdin

stdin 引数が `PIPE` の場合、この属性には子プロセスの入力に使われるファイルオブジェクトになります。そうでない場合は `None` です。

stdout

stdout 引数が `PIPE` の場合、この属性には子プロセスの出力に使われるファイルオブジェクトになります。そうでない場合は `None` です。

stderr

stderr 引数が `PIPE` の場合、この属性には子プロセスのエラー出力に使われるファイルオブジェクトになります。そうでない場合は `None` です。

pid

子プロセスのプロセス ID が入ります。

returncode

子プロセスの終了ステータスが入ります。`None` はまだその子プロセスが終了していないことを示し、負の値 -N は子プロセスがシグナル N により中止させられたことを示します (UNIX のみ)。

17.1.3 古い関数を subprocess モジュールで置き換える

以下、この節では、"a ==> b" と書かれているものは a の代替として b が使えるということを表します。

注意: この節で紹介されている関数はすべて、実行するプログラムが見つからないときは (いくぶん) 静かに終了します。このモジュールは `OSError` 例外を発生させます。

以下の例では、 subprocess モジュールは "from subprocess import *" でインポートされたと仮定しています。

/bin/sh シェルのバッククォートを置き換える

```
output=`mycmd myarg`
==>
output = Popen(["mycmd", "myarg"], stdout=PIPE).communicate()[0]
```

シェルのパイプラインを置き換える

```
output=`dmesg | grep hda`
==>
p1 = Popen(["dmesg"], stdout=PIPE)
p2 = Popen(["grep", "hda"], stdin=p1.stdout, stdout=PIPE)
output = p2.communicate()[0]
```

os.system() を置き換える

```
sts = os.system("mycmd" + " myarg")
==>
p = Popen("mycmd" + " myarg", shell=True)
sts = os.waitpid(p.pid, 0)
```

注意:

- このプログラムは普通シェル経由で呼び出す必要はありません。
- 終了状態を見るよりも `returncode` 属性を見るほうが簡単です。

より現実的な例ではこうなるでしょう:

```
try:
    retcode = call("mycmd" + " myarg", shell=True)
    if retcode < 0:
        print >>sys.stderr, "子プロセスがシグナルによって中止されました", -retcode
    else:
        print >>sys.stderr, "子プロセスが終了コードを返しました", retcode
except OSError, e:
    print >>sys.stderr, "実行に失敗しました:", e
```

os.spawn* を置き換える

P_NOWAIT の例:

```
pid = os.spawnlp(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg")
==>
pid = Popen(["/bin/mycmd", "myarg"]).pid
```

P_WAIT の例:

```
retcode = os.spawnlp(os.P_WAIT, "/bin/mycmd", "mycmd", "myarg")
==>
retcode = call(["/bin/mycmd", "myarg"])
```

シーケンスを使った例:

```
os.spawnvp(os.P_NOWAIT, path, args)
==>
Popen([path] + args[1:])
```

環境変数を使った例:

```
os.spawnlpe(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg", env)
==>
Popen(["/bin/mycmd", "myarg"], env={"PATH": "/usr/bin"})
```

os.popen* を置き換える

```
pipe = os.popen(cmd, mode='r', bufsize)
==>
pipe = Popen(cmd, shell=True, bufsize=bufsize, stdout=PIPE).stdout
```

```
pipe = os.popen(cmd, mode='w', bufsize)
==>
pipe = Popen(cmd, shell=True, bufsize=bufsize, stdin=PIPE).stdin
```

```
(child_stdin, child_stdout) = os.popen2(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdin, child_stdout) = (p.stdin, p.stdout)
```

```
(child_stdin,
 child_stdout,
 child_stderr) = os.popen3(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=PIPE, close_fds=True)
(child_stdin,
 child_stdout,
 child_stderr) = (p.stdin, p.stdout, p.stderr)
```

```
(child_stdin, child_stdout_and_stderr) = os.popen4(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=STDOUT, close_fds=True)
(child_stdin, child_stdout_and_stderr) = (p.stdin, p.stdout)
```

popen2.* を置き換える

注意: popen2 に対するコマンド引数が文字列の場合、そのコマンドは /bin/sh 経由で実行されます。いっぽうこれがリストの場合、そのコマンドは直接実行されます。

```
(child_stdout, child_stdin) = popen2.popen2("somestring", bufsize, mode)
==>
p = Popen(["somestring"], shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

```
(child_stdout, child_stdin) = popen2.popen2(["mycmd", "myarg"], bufsize, mode)
==>
p = Popen(["mycmd", "myarg"], bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

popen2.Popen3 および popen2.Popen4 は基本的には subprocess.Popen と同様です。ただし、違う点は:

- subprocess.Popen は実行できなかった場合に例外を発生させます。
- *capturestderr* 引数は *stderr* 引数に代わりました。
- stdin=PIPE および stdout=PIPE を指定する必要があります。
- popen2 はデフォルトですべてのファイル記述子を閉じますが、subprocess.Popen では明示的に close_fds=True を指定する必要があります。

17.2 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。**注意:** いくつかの振る舞いはプラットフォームに依存します。これはオペレーティングシステムのソケット API を呼び出しているためです。

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ませんので、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。

2.5 で追加された仕様: AF_NETLINK ソケットが *pid*, *groups* のペアで表現されます

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() などで、*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。*error* の値は、このモジュールで定義される EAI_* 定数の何れかとなります。

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。`getattrinfo()` 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、`gethostname()` で返される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 インター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。省略可能なプロトコル名として、`'tcp'` か `'udp'` のどちらかを指定することができます。指定がなければどちらのプロトコルにもマッチします。

getservbyport (*port*[, *protocolname*])

インターネットポート番号とプロトコル名から、サービス名を取得します。省略可能なプロトコル名として、`'tcp'` か `'udp'` のどちらかを指定することができます。指定がなければどちらのプロトコルにもマッチします。

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`(デフォルト値)・`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`(デフォルト値)・`SOCK_DGRAM`・または他の 'SOCK_' 定数の何れかを指定します。プロトコル番号は通常省略するか、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。

socketpair ([*family*[, *type*[, *proto*]]])

指定されたアドレスファミリ、ソケットタイプ、プロトコル番号から、接続されたソケットのペアを作成します。アドレスファミリ、ソケットタイプ、プロトコル番号は `socket()` 関数と同様に指定します。デフォルトのアドレスファミリは、プラットフォームで定義されていれば `AF_UNIX`、そうでなければ `AF_INET` が使われます。

利用可能: UNIX. 2.4 で追加された仕様です。

fromfd (*fd, family, type*[, *proto*])

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) *fd* を複製して、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 デー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を、ネットワークバイトオーダからホストバイトオーダに変換します。ホストバイトオーダとネットワークバイトオーダ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を、ネットワークバイトオーダからホストバイトオーダに変換します。ホストバイトオーダとネットワークバイトオーダ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を、ホストバイトオーダからネットワークバイトオーダに変換します。ホストバイトオーダとネットワークバイトオーダ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を、ホストバイトオーダからネットワークバイトオーダに変換します。ホストバイトオーダとネットワークバイトオーダ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事ができます。

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事ができます。

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。タイプ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 (18.18 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。

17.2.1 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。

Windows ではこのメソッドで返された小整数をファイルディスクリプタを扱う箇所 (os.fdopen() など) で利用できません。UNIX にはこの制限はありません。

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 3.9 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。**注意:** ハードウェアおよびネットワークの現実に最大限マッチするように、*bufsize* の値は比較的小さい 2 の累乗、たとえば 4096、にすべきです。

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。

註: `connect()` はタイムアウト設定に従います。一般的に、`settimeout()` を `connect()` の前に呼ぶことをおすすめします。

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 `SHUT_RD` の場合、以降は受信を行えません。*how* が `SHUT_WR` の場合、以降は送信を行えません。*how* が `SHUT_RDWR` 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。

ソケットオブジェクトには以下の `socket` コンストラクタに渡された値に対応した (読み出し専用) 属性があります。

family
: ソケットファミリー。 2.5 で追加された仕様です。

type
: ソケットタイプ。 2.5 で追加された仕様です。

proto
: ソケットプロトコル。 2.5 で追加された仕様です。

17.2.2 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。

server ()
: サーバの証明書を特定するための ASN.1 識別名 (distinguished name) を含む文字列を返します。(下の例を見ると識別名がどう見えるものか判ります。)

issuer ()
: サーバの証明書の発行者を特定するための ASN.1 識別名 (distinguished name) を含む文字列を返します。

17.2.3 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()` だけを呼び出し

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', repr(data)
```

The next two examples are identical to the above two, but support both IPv4 and IPv6. The server side will listen to the first address family available (it should listen to both instead). On most of IPv6-ready systems, IPv6 will take precedence and the server may not accept IPv4 traffic. The client side will try to connect to the all addresses returned as a result of the name resolution, and sends traffic to the first one connected successfully.

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
    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', repr(data)
```

次の例では SSL サーバに接続し、サーバおよび発行者の識別名 (distinguished name) を表示し、いくらかのバイトを送り、レスポンスの一部を読みます:

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('www.verisign.com', 443))

ssl_sock = socket.ssl(s)

print repr(ssl_sock.server())
print repr(ssl_sock.issuer())

# Set a simple HTTP request -- use httplib in actual code.
ssl_sock.write("""GET / HTTP/1.0\r
Host: www.verisign.com\r\n\r\n""")

# Read a chunk of data.  Will not necessarily
# read all the data returned by the server.
data = ssl_sock.read()

# Note that you need to close the underlying socket, not the SSL object.
del ssl_sock
s.close()
```

執筆時点で、この SSL 実行例は次のような出力を表示しました (読み易いように改行は入れてあります):

```
'/C=US/ST=California/L=Mountain View/
 O=VeriSign, Inc./OU=Production Services/
 OU=Terms of use at www.verisign.com/rpa (c)00/
 CN=www.verisign.com'
'/O=VeriSign Trust Network/OU=VeriSign, Inc./
 OU=VeriSign International Server CA - Class 3/
 OU=www.verisign.com/CPS Incorp.by Ref. LIABILITY LTD.(c)97 VeriSign'
```

17.3 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 は標準でごく小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で、<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の標準型の階層 か、inspect モジュールの属性の説明を参照してください)、とともに呼び出されます。

17.3.1 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
```

17.4 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。

UNIX では *cmd* はシーケンスでもよく、その場合には (os.spawnv() のように) 引数はプログラムシェルを経由せず直接渡されます。*cmd* が文字列の場合、(os.system() のように) シェルに渡されます。

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は得ることができません。(os モジュールの関数から返されるタプルは popen2 モジュールの関数から返されるものとは違う順序です。)

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。

17.4.1 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; 14.1.5 節の `W*()` 関数ファミリ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。`Poepn4` 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出力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。

17.4.2 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 select() 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。

17.5 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プ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。

チャネルを生成後、`loop()` を呼び出すとチャネル処理が開始し、最後のチャネル（非同期処理中にマップに追加されたチャネルを含む）が閉じるまで継続します。

loop([*timeout*[, *use_poll*[, *map*[,*count*]]]])

ポーリングループを開始し、count回が過ぎるか、全てのオープン済みチャ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 *count* のデフォルト値はNoneで、ループは全てのチャネルがクローズされた場合のみ終了します。 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ます。

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、発生の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
`handle_connect()`	最初にwriteイベントが発生した時
`handle_close()`	読み込み可能なデータなしでreadイベントが発生した時
`handle_accept()`	listen中のソケットでread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ライドすることが可能です：

handle_read()
非同期ループで、チャネルのソケットの `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() で接続待ち状態でなければなりません。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。

17.5.1 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。

```
import asyncore, socket

class http_client(asyncore.dispatcher):

    def __init__(self, host, path):
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.connect( (host, 80) )
        self.buffer = 'GET %s HTTP/1.0\r\n\r\n' % path

    def handle_connect(self):
        pass

    def handle_close(self):
        self.close()

    def handle_read(self):
        print self.recv(8192)

    def writable(self):
        return (len(self.buffer) > 0)

    def handle_write(self):
        sent = self.send(self.buffer)
        self.buffer = self.buffer[sent:]

c = http_client('www.python.org', '/')

asyncore.loop()
```

17.6 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 asynchat.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の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ライドする必要があり、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ライドする必要があり、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、チャネルの producer_fifo にプッシュして転送します。データをチャ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、チャネルはクローズされます。

set_terminator(*term*)

チャ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
string	入力ストリーム中で string が検出された時、found_terminator() を呼び出します。
integer	指定された文字数が読み込まれた時、found_terminator() を呼び出します。
None	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。

17.6.1 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*])

各チャネルは、アプリケーションからプッシュされ、まだチャ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、プロデューサかチャ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("abc12r", "r")` は 1 を返します。

17.6.2 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(asynchat.async_chat):

    def __init__(self, conn, addr, sessions, log):
        asynchat.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()
```

十八章

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。

webbrowser	ウェウブブラウザーのための使い易いコントローラー
cgi	サーバ側で動作するスクリプトがフォームの内容を解釈するために使うゲートウェイイ
cgitb	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。
wsgiref.simple_server	WSGI ユーティリティとリファレンス実装
urllib	URL による任意のネットワークリソースへのアクセス (socket が必要です)。
urllib2	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
httplib	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) 。
ftplib	FTP プロトコルクライアント (ソケットを必要とします)。
gopherlib	gopher プロトコルのクライアント (ソケットを必要とします)。
poplib	POP3 プロトコルクライアント (sockets を必要とする)
imaplib	IMAP4 protocol client (requires sockets).
nntplib	NNTP プロトコルクライアント (ソケットを必要とします)。
smtplib	SMTP プロトコル クライアント (ソケットが必要です)。
smtpd	柔軟性のある SMTP サーバの実装
telnetlib	Telnet クライアントクラス
uuid	RFC 4122 に準拠した UUID オブジェクト（汎用一意識別子）
urlparse	URL を解析して構成要素にします。
SocketServer	ネットワークサーバ構築のためのフレームワーク。
BaseHTTPServer	基本的な機能を持つ HTTP サーバ (`SimpleHTTPServer` および `CGIHTTPServer` の
SimpleHTTPServer	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。
CGIHTTPServer	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
cookielib	HTTP クライアント用の Cookie 処理
Cookie	HTTP 状態管理 (cookies) のサポート。
xmlrpclib	XML-RPC client access.
SimpleXMLRPCServer	基本的な XML-RPC サーバーの実装。
DocXMLRPCServer	セルフ-ドキュメンティング XML-RPC サーバの実装。

18.1 `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ースが定義されています。たいていの環境では、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。

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、os.pathsep 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 上でリモート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。UNIX 上でリモートブラウザが利用可能でない場合、制御プロセスは新しいブラウザを立ち上げ、待ちます。

webbrowser スクリプトをこのモジュールのコマンドラインインタフェースとして使うことができます。スクリプトは引数に一つの URL を受け付けます。また次のオプション引数を受け付けます。 **-n** により可能ならば新しいブラウザウィンドウで指定された URL を開きます。一方、**-t** では新しいブラウザのページ (「タブ」) で開きます。当然ながらこれらのオプションは排他的です。

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 0 なら、*url* はブラウザの今までと同じウィンドウで開きます。*new* が 1 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。*new* が 2 なら、可能であればブラウザの新しいタブが開きます。*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)。 2.5 で変更された仕様: *new* を 2 にもできるようになりました

`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。

`open_new_tab` (*url*)

可能であれば、デフォルトブラウザの新しいページ (「タブ」) で *url* を開きますが、そうでない場合は `open_new` と同様に振る舞います。 2.5 で追加された仕様です。

`get` ([*name*])

ブラウザの種類 *name* のコントローラー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ーを返します。

`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ー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、インスタンスが必要な時には *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* は呼び出されないので、`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ー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'	Mozilla('mozilla')	
'firefox'	Mozilla('mozilla')	
'netscape'	Mozilla('netscape')	
'galeon'	Galeon('galeon')	
'epiphany'	Galeon('epiphany')	
'skipstone'	BackgroundBrowser('skipstone')	
'kfmclient'	Konqueror()	(1)
'konqueror'	Konqueror()	(1)
'kfm'	Konqueror()	(1)
'mosaic'	BackgroundBrowser('mosaic')	
'opera'	Opera()	
'grail'	Grail()	
'links'	GenericBrowser('links')	
'elinks'	Elinks('elinks')	
'lynx'	GenericBrowser('lynx')	
'w3m'	GenericBrowser('w3m')	
'windows-default'	WindowsDefault	(2)
'internet-config'	InternetConfig	(3)
'macosx'	MacOSX('default')	(4)

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。

(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。

(4) MacOS X プラットフォームのみ。

簡単な例を示します。

```
url = 'http://www.python.org'

# Open URL in a new tab, if a browser window is already open.
webbrowser.open_new_tab(url + '/doc')

# Open URL in new window, raising the window if possible.
webbrowser.open_new(url)
```

18.1.1 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*[, *autoraise=1*]])

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 1 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。*new* が 2 なら、可能であればブラウザの新しいページ (「タブ」) 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。open_new の別名。

open_new_tab(*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ページ (「タブ」) で *url* を開きますが、そうでない場合は open_new と同じです。 2.5 で追加された仕様です。

18.2 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。

このモジュールでは、 Python で CGI スクリプトを書く際に使える様々なユーティリティを定義しています。

18.2.1 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します。

ほとんどの場合、CGI スクリプトはサーバ上の特殊なディレクトリ 'cgi-bin' の下に置きます。HTTP サーバは、まずスクリプトを駆動するためのシェルの環境変数に、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クライアントに接続されていて、この経路を通じてフォームデータを読み込むこともあります。それ以外の場合には、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つつ、Python スクリプトに対しては単純なインタフェースを提供しています。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も多数提供しています。また、最近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)。

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"     # 以降のデータが HTML であることを示す行
print                               # ヘッダ部の終了を示す空行
```

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!"
```

18.2.2 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 *' と書いてはなりません — このモジュールでは、以前のバージョンとの互換性を持たせるため、内部で呼び出す名前を多数定義しており、それらをユーザの名前空間に存在させる必要はないからです。

新たにスクリプトを書く際には、以下の一行を付加するかどうか検討してください:

```
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レポート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、以下のようにしてレポートをファイルに保存できます:

```
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かかる時間を大幅に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ら、いつでも cgitb の行を削除できます。

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。このモジュールで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を読み出します (どちらから読み出すかは、複数の環境変数の値が CGI 標準に従ってどう設定されているかで決まります)。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クスを使って参照でき、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() もサポートしています。空の文字列を含むフォームのフィールドは無視され、辞書には入りません; 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 の内容は対応するフィールドの値で、文字列になります。getvalue() メソッドはこの文字列値を直接返します。getvalue() の 2 つめの引数にオプションの値を与えると、リクエストされたキーが存在しない場合に返すデフォルトの値になります。

入力されたフォームデータに同じ名前のフィールドが二つ以上あれば、'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、そうしたインスタンスのリス

トになります。この場合、'form.getvalue(*key*)' も同様に、文字列からなるリストを返します。もしこうした状況が起きうると思うなら (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のなら)、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どうか調べてください。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している場合、value 属性や getvalue() メソッドを使ってフィールドの値にアクセスすると、ファイルの内容を全て文字列としてメモリ上に読み込んでしまいます。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られます。その後、以下のようにして file 属性から落ち着いてデータを読み出せます:

```
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って) 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を調べれば判別できます。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単一の application/x-www-form-urlencoded データで入力された場合)、データ要素の実体は 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。

18.2.3 高水準インタフェース

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、フォームデータを分かりやすく直感的な方法で読み出せるようにするために追加された、より高水準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説明した技術を撤廃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。このメソッドを使えば、一般的な方法でフォームデータを処理でき、ある名前のフィールドに入力された値が一つなのかそれ以上なのかを心配する必要がなくなります。

前節では、一つのフィールド名に対して二つ以上の値が入力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入っているような場合によく起きます:

```
<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しかないので、その名前に関連付けられた値はただ一つしかないはずだと考えるでしょう。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").upper()
```

このコードの問題点は、クライアント側がスクリプトにとって常に有効な入力を提供するとは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'")` メソッドは文字列ではなくリストを返すため、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 `upper()` 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` を送出します。

従って、フォームデータの値を読み出しには、得られた値が単一の値なのか値のリストなのかを常に調べるコードを使うのが適切でした。これでは煩わしく、より読みにくいスクリプトになってしまいます。

ここで述べる高水準のインタフェースで提供している `getfirst()` や `getlist()` メソッドを使うと、もっと便利にアプローチできます。

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。同じ名前で 1 つ以上の値がポストされている場合、このメソッドは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の並び順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。[1]

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にして返します。*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にコードを書けます:

[1] 最近のバージョンの HTML 仕様ではフィールドの値を供給す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準拠したブラウザから受信したものかどうか、そもそもブラウザから送信されたも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。

```
import cgi
form = cgi.FieldStorage()
user = form.getfirst("user", "").upper()    # This way it's safe.
for item in form.getlist("item"):
    do_something(item)
```

18.2.4 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場合に便利です。

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っていて、こ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、作者まで連絡を下さい。

18.2.5 関数

より細かく CGI を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、以下の関数が便利です。

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。解釈されたデータを辞書として返します。辞書のキーは一意なクエリ変数名で、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を無視し、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います。

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決めるフラグです。値が偽なら (標準の設定です)、エラーは暗黙のうちに無視します。値が真なら ValueError 例外を送出します。

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_qsl (*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を無視し、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います。

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決めるフラグです。値が偽なら (標準の設定です)、エラーは暗黙のうちに無視します。値が真なら `ValueError` 例外を送出します。

ペアのリストからクエリ文字列を生成する場合には `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ます。辞書のキーはフィールド名で、対応する値は各フィールドの値でできたリストです。この関数は簡単に使えますが、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供給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列に変換します。それらの文字が中に入ってい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* の値が真であれば、二重引用符文字 ('`"`') も変換します; 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` の `quoteattr()` 関数を検討してください。

18.2.6 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てしまえます。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、フォームから入力された文字をシェルに渡す場合、文字列に入っているのが英数文字、ダッシュ、アンダースコア、およびピリオドだけかどうかを確認してください。

18.2.7 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しなければならないファイルが全て "others" によって読み出しや書き込み可能であることを確かめてください — 読み出し可能のファイルモードは `0644` で、書き込み可能のファイルモードは `0666` になるはずです。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) ファイルしか読み出し (書き込み、実行) 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と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)。

18.2.8 CGI スクリプトをテストする

残念ながら、 CGI スクリプトは普通、コマンドラインから起動しようとしても動きません。また、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てみ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、次の節に読み進むしかありません。

18.2.9 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めば時間を大いに節約できます。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、以下のような URL をブラウザに入力してリクエストを送信できるはずです:

```
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発見できないでいます – 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のでしょう。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") が綺麗にフォーマットされて表示されるなら、'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 スクリプトが例外を処理しきれずに送出した場合 (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振舞うので、トレースバックは大抵 HTTP サーバのいずれかのログファイルに残るかまったく無視されるかです。

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めます。

`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れます:

```
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。出力のコンテント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めるはずです。

18.2.10 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もありません。

18.3 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。

18.4 wsgiref — WSGI ユーティリティとリファレンス実装

2.5 で追加された仕様です。

Web Server Gateway Interface (WSGI) は、Web サーバソフトウェアと Python で記述された Web アプリケーションとの標準インターフェースです。標準インターフェースを持つことで、WSGI をサポートするアプリケーションを幾つもの異なる Web サーバで使うことが容易になります。

Web サーバとプログラミングフレームワークの作者だけが、WSGI デザインのあらゆる細部や特例などを知る必要があります。WSGI アプリケーションをインストールしたり、既存のフレームワークを使ったアプリケーションを記述するだけの皆さんは、全てについて理解する必要はありません。

wsgiref は WSGI 仕様のリファレンス実装で、これは Web サーバやフレームワークに WSGI サポートを加えるのに利用できます。これは WSGI 環境変数やレスポンスヘッダを操作するユーティリティ、WSGI サーバ実装時のベースクラス、WSGI アプリケーションを提供するデモ用 HTTP サーバ、それと WSGI サーバとアプリケーションの WSGI 仕様 (PEP 333) 準拠のバリデーションツールを提供します。

http://www.wsgi.org に、WSGI に関するさらなる情報と、チュートリアルやその他のリソースへのリンクがあります。

18.4.1 wsgiref.util – WSGI 環境のユーティリティ

このモジュールは WSGI 環境で使う様々なユーティリティ関数を提供します。WSGI 環境は PEP 333 で記述されているような HTTP リクエスト変数を含む辞書です。全ての *environ* パラメタを取る関数は WSGI 準拠の辞書を与えられることを期待しています；細かい仕様については PEP 333 を参照してください。

guess_scheme(*environ*)

wsgi.url_scheme が "http" か "https" かについて、*environ* 辞書の HTTPS 環境変数を調べることでその推測を返します。 戻り値は文字列 (string) です。

この関数は、CGI や FastCGI のような CGI に似たプロトコルをラップするゲートウェイを作成する場合に便利です。典型的には、それらのプロトコルを提供するサーバが SSL 経由でリクエストを受け取った場合には HTTPS 変数に値 "1" "yes"、または "on" を持つでしょう。ですので、この関数はそのような値が見つかった場合には "https" を返し、そうでなければ "http" を返します。

request_uri(*environ* [, *include_query=1*])

クエリ文字列をオプションで含むリクエスト URI 全体を、PEP 333 の "URL 再構築 (URL Reconstruction)" にあるアルゴリズムを使って返します。*include_query* が false の場合、クエリ文字列は結果となる文字列には含まれません。

application_uri(*environ*)

request_url に似ていて、PATH_INFO と QUERY_STRING 変数は無視されます。結果はリクエストによって指定されたアプリケーションオブジェクトのベース URI です。

shift_path_info(*environ*)

PATH_INFO から SCRIPT_NAME まで一つの名前をシフトしてその名前を返します。*environ* 辞書は**変更されます**；PATH_INFO や SCRIPT_NAME のオリジナルをそのまま残したい場合にはコピーを使ってください。

PATH_INFO にパスセグメントが何も残っていなければ、None が返されます。

典型的なこのルーチンの使い方はリクエスト URI のそれぞれの要素の処理で、例えばパスを一連の辞書のキーとして取り扱う場合です。このルーチンは、渡された環境を、ターゲット URL で示される別の WSGI アプリケーションの呼び出しに合うように調整します。例えば、/foo に WSGI アプリケーションがあったとして、そしてリクエスト URL パスが /foo/bar/baz で、/foo の WSGI アプリケーションが shift_path_info を呼んだ場合、これは "bar" 文字列を受け取り、環境は /foo/bar の WSGI アプリケーションへの受け渡しに適するように更新されます。つまり、SCRIPT_NAME は /foo から /foo/bar に変わって、PATH_INFO は /bar/baz から /baz に変化するのです。

PATH_INFO が単に "/" の場合、このルーチンは空の文字列を返し、SCRIPT_NAME の末尾にスラッシュを加えます、これはたとえ空のパスセグメントが通常は無視され、そして SCRIPT_NAME は通常スラッシュで終わる事が無かったとしてもです。これは意図的な振る舞いで、このルーチンでオブジェクト巡回 (object traversal) をした場合に/x で終わる URI と /x/ で終わるものをアプリケーションが識別できることを保証するためのものです。

setup_testing_defaults(*environ*)

テスト目的で、*environ* を自明なデフォルト値 (trivial defaults) で更新します。

このルーチンは WSGI に必要な様々なパラメタを追加し、それには HTTP_HOST、SERVER_NAME、SERVER_PORT、REQUEST_METHOD、SCRIPT_NAME、PATH_INFO、あとは PEP 333 で定義されている wsgi.* 変数群を含みます。これはデフォルト値のみを追加し、これらの変数の既存設定は一切置きかえません。

このルーチンは、ダミー環境をセットアップすることによって WSGI サーバとアプリケーションのユニットテストを容易にすることを意図しています。これは実際の WSGI サーバやアプリケーションで使うべきではありません。なぜならこのデータは偽物なのです！

上記の環境用関数に加えて、wsgiref.util モジュールも以下のようなその他のユーティリティを提供します：

is_hop_by_hop(*header_name*)

'header_name' が RFC 2616 で定義されている HTTP/1.1 の "Hop-by-Hop" ヘッダの場合に true を返します。

class **FileWrapper**(*filelike* [, *blksize=8192*])

ファイルライクオブジェクトをイテレータに変換するラッパです。結果のオブジェクトは __getitem__ と __iter__ 両方をサポートしますが、これは Python 2.1 と Jython の互換性のためです。オブジェクトがイテレートされる間、オプションの *blksize* パラメタがくり返し *filelike* オブジェクトの read() メソッドに渡されて受け渡す文字列を取得します。read() が空文字列を返した場合イテレーションは終了して、再開されることはありません。

filelike に close() メソッドがある場合、返されたオブジェクトも close() メソッドを持ち、これが呼ばれた場合には *filelike* オブジェクトの close() メソッドを呼び出します。

18.4.2 wsgiref.headers – WSGI レスポンスヘッダツール群

このモジュールは単一のクラス、Headers を提供し、WSGI レスポンスヘッダの操作をマップに似たインターフェースで便利にします。

class Headers(*headers*)

headers をラップするマップに似たオブジェクトを生成します。これは PEP 333 に定義されるようなヘッダの名前／値のタプルのリストです。新しい Headers オブジェクトに与えられた変更は、一緒に作成された *headers* リストを直接更新します。

Headers オブジェクトは典型的なマッピング操作をサポートし、これには__getitem__、get、__setitem__、setdefault、__delitem__、__contains__ と has_key を含みます。これらメソッドのそれぞれにおいて、キーはヘッダ名で（大文字小文字は区別しません）、値はそのヘッダ名に関連づけられた最初の値です。ヘッダを設定すると既存のヘッダ値は削除され、ラップされたヘッダのリストの末尾に新しい値が加えられます。既存のヘッダの順番は一般的に整えられていて、ラップされたリストの最後に新しいヘッダが追加されます。

辞書とは違って、Headers オブジェクトはラップされたヘッダリストに存在しないキーを取得または削除しようとした場合にもエラーを発生しません。単に、存在しないヘッダの取得は None を返し、存在しないヘッダの削除は何もしません。

Headers オブジェクトは keys()、values()、items() メソッドもサポートします。keys() と items() で返されるリストは、同じキーを一回以上含むことがあり、これは複数の値を持つヘッダの場合です。Header オブジェクトの len() は、その items() の長さと同じであり、ラップされたヘッダリストの長さと同じです。事実、items() メソッドは単にラップされたヘッダリストのコピーを返しているだけです。

Headers オブジェクトに対して str() を呼ぶと、HTTP レスポンスヘッダとして送信するのに適した形に整形された文字列を返します。それぞれのヘッダはコロンとスペースで区切られた値と共に一列に並んでいます。それぞれの行はキャリッジリターンとラインフィードで終了し、文字列は空行で終了しています。

これらのマッピングインターフェースと整形機能に加えて、Headers オブジェクトは複数の値を持つヘッダの取得と追加、MIME パラメタでヘッダを追加するための以下のようなメソッド群も持っています：

get_all(*name*)

指定されたヘッダの全ての値のリストを返します。

返されるリストは、元々のヘッダリストに現れる順、またはこのインスタンスに追加された順に並んでいて、複製を含む場合があります。削除されて加えられたフィールドは全てヘッダリストの末尾に付きます。ある名前のフィールドが何もなければ、空のリストが返ります。

add_header(*name, value, **_params*)

ヘッダ（複数の値かもしれません）を、キーワード引数を通じて指定するオプションの MIME パラメタと共に追加します。

name は追加するヘッダフィールドです。このヘッダフィールドに MIME パラメタを設定するためにキーワード引数を使うことができます。それぞれのパラメタは文字列か None でなければいけません。パラメタ中のアンダースコアはダッシュに変換されます、これはダッシュが Python の識別子としては不正なのですが、多くの MIME パラメタはダッシュを含むためです。パラメタ値が文字列の場合、これはヘッダ値のパラメタに name="value" の形で追加されます。これがもし None の場合、パラメタ名だけが追加されます。（これは値なしの MIME パラメタの場合に使われます。）使い方の例は：

```
h.add_header('content-disposition', 'attachment', filename='bud.gif')
```

上記はこのようなヘッダを追加します：

```
Content-Disposition: attachment; filename="bud.gif"
```

18.4.3 wsgiref.simple_server – シンプルな WSGI HTTP サーバ

このモジュールは WSGI アプリケーションを提供するシンプルな HTTP サーバです（BaseHTTPServer がベースです）。個々のサーバインスタンスは単一の WSGI アプリケーションを、特定のホストとポート上で提供します。もし一つのホストとポート上で複数のアプリケーションを提供したいならば、PATH_INFO をパースして個々のリクエストでどのアプリケーションを呼び出すか選択するような WSGI アプリケーションを作るべきです。（例えば、 wsgiref.util から shift_path_info() を利用します。）

make_server(*host, port, app* [, *server_class=WSGIServer* [, *handler_class=WSGIRequestHandler*]])

host と *port* 上で待機し、 *app* へのコネクションを受け付ける WSGI サーバを作成します。戻り値は与えられた *server_class* のインスタンスで、指定された *handler_class* を使ってリクエストを処理します。*app* は PEP 333 で定義されるところの WSGI アプリケーションでなければいけません。

使用例：

```
from wsgiref.simple_server import make_server, demo_app

httpd = make_server('', 8000, demo_app)
print "Serving HTTP on port 8000..."

# プロセスが死ぬまでリクエストに答える
httpd.serve_forever()

# 代替：1つのリクエストを受けて終了する
##httpd.handle_request()
```

demo_app(*environ, start_response*)

この関数は小規模ながら完全な WSGI アプリケーションで、"Hello world!" メッセージと、*environ* パラメタに提供されているキー／値のペアをを含むテキストページを返します。これは WSGI サーバ（wsgiref.simple_server のような）がシンプルな WSGI アプリケーションを正しく実行できるかを確かめるのに便利です。

class **WSGIServer**(*server_address, RequestHandlerClass*)

WSGIServer インスタンスを作成します。*server_address* は (host,port) のタプル、そして *RequesthandlerClass* はリクエストの処理に使われる BaseHTTPServer.BaseHTTPRequestHandler のサブクラスでなければいけません。

make_server が細かい調整をやってくれるので、通常はこのコンストラクタを呼ぶ必要はありません。

WSGIServer は BaseHTTPServer.HTTPServer のサブクラスですので、この全てのメソッド（serve_forever() や handle_request() のような）が利用できます。WSGIServer も以下のような WSGI 固有メソッドを提供します：

set_app(*application*)

呼び出し可能（callable）な *application* をリクエストを受け取る WSGI アプリケーションとして設定します。

get_app()

現在設定されている呼び出し可能（callable）アプリケーションを返します。

しかしながら、通常はこれらの追加されたメソッドを使う必要はありません。set_app() は普通は make_server() によって呼ばれ、get_app() は主にリクエストハンドラインスタンスの便宜上存在するからです。

class **WSGIRequestHandler**(*request, client_address, server*)

与えられた *request*（すなわちソケット）の HTTP ハンドラ、*client_address*（*host*,*port*）のタプル）、

server（WSGIServer インスタンス）の HTTP ハンドラを作成します。

このクラスのインスタンスを直接生成する必要はありません；これらは必要に応じて WSGIServer オブジェクトによって自動的に生成されます。しかしながら、このクラスをサブクラス化し、make_server() 関数に *handler_class* として与えることは可能でしょう。サブクラスにおいてオーバーライドする意味のありそうなものは：

get_environ()
: リクエストに対する WSGI 環境を含む辞書を返します。デフォルト実装では WSGIServer オブジェクトの base_environ 辞書属性のコンテンツをコピーし、それから HTTP リクエスト由来の様々なヘッダを追加しています。このメソッド呼び出し毎に、PEP 333 に指定されている関連する CGI 環境変数を全て含む新規の辞書を返さなければいけません。

get_stderr()
: wsgi.errors ストリームとして使われるオブジェクトを返します。デフォルト実装では単に sys.stderr を返します。

handle()
: HTTP リクエストを処理します。デフォルト実装では実際の WGI アプリケーションインターフェースを実装するのに wsgiref.handlers クラスを使ってハンドラインスタンスを作成します。

18.4.4 wsgiref.validate – WSGI 準拠チェッカ

WSGI アプリケーションのオブジェクト、フレームワーク、サーバ又はミドルウェアの作成時には、その新規のコードを wsgiref.validate を使って準拠の検証をすると便利です。このモジュールは WSGI サーバやゲートウェイと WSGI アプリケーションオブジェクト間の通信を検証する WSGI アプリケーションオブジェクトを作成する関数を提供し、双方のプロトコル準拠をチェックします。

このユーティリティは完全な PEP 333 準拠を保証するものでないことは注意してください；このモジュールでエラーが出ないことは必ずしもエラーが存在しないことを意味しません。しかしこのモジュールがエラーを出したならば、サーバかアプリケーションのどちらかが 100

このモジュールは lan Bicking の "Python Paste" ライブラリの paste.lint モジュールをベースにしています。

validator(*application*)
: *application* をラップし、新しい WSGI アプリケーションオブジェクトを返します。返されたアプリケーションは全てのリクエストを元々の *application* にフォワードし、*application* とそれを呼び出すサーバの両方が WSGI 仕様と RFC 2616 の両方に準拠しているかをチェックします。

  検出された非準拠は、投げられる AssertionError の中に入ります；しかし、このエラーがどう扱われるかはサーバ依存であることに注意してください。例えば、wsgiref.simple_server とその他 wsgiref.handlers ベースのサーバ（エラー処理メソッドが他のことをするようにオーバライドしていないもの）は単純にエラーが発生したというメッセージとトラックバックのダンプを sys.stderr やその他のエラーストリームに出力します。

  このラッパは warnings モジュールを使って出力を生成し、疑問の余地はあるが実際には PEP 333 で禁止はされていないかもしれない挙動を指摘します。これらは Python のコマンドラインオプションや warnings API で抑制されなければ、sys.stderr（たまたま同一のオブジェクトで無い限り wsgi.errors では**ない**）に書き出されます。

18.4.5 wsgiref.handlers – サーバ／ゲートウェイのベースクラス

このモジュールは WSGI サーバとゲートウェイ実装のベースハンドラクラスを提供します。これらのベースクラスは CGI ライクの環境を与えられれば入力、出力そしてエラーストリームと共に WSGI アプリケーションとの通信の大部分を処理します。

class CGIHandler()

sys.stdin、sys.stdout、stderr そして os.environ 経由での CGI ベースの呼び出しです。これは、もしあなたが WSGI アプリケーションを持っていて、これを CGI スクリプトとして実行したい場合に有用です。単に CGIHandler().run(app) を起動してください。app はあなたが起動したい WSGI アプリケーションオブジェクトです。

このクラスは BaseCGIHandler のサブクラスで、これは wsgi.run_once を true、wsgi.multithread を false、そして wsgi.multiprocess を true にセットし、常に sys と os を、必要な CGI ストリームと環境を取得するために使用します。

class BaseCGIHandler(*stdin, stdout, stderr, environ* [, *multithread=True* [, *multiprocess=False*]])

CGIHandler に似ていますが、sys と os モジュールを使う代わりに CGI 環境と I/O ストリームを明示的に指定します。*multithread* と *multiprocess* の値は、ハンドラインスタンスにより実行されるアプリケーションの wsgi.multithread と wsgi.multiprocess フラグの設定に使われます。

このクラスは SimpleHandler のサブクラスで、HTTP の "本サーバ" でないソフトウェアと使うことを意図しています。もしあなたが Status: ヘッダを HTTP ステータスを送信するのに使うような ゲートウェイプロトコルの実装（CGI、FastCGI、SCGI など）を書いているとして、おそらく SimpleHandler でなくこれをサブクラス化したいことでしょう。

class SimpleHandler(*stdin, stdout, stderr, environ* [,*multithread=True* [, *multiprocess=False*]])

BaseCGIHandler と似ていますが、HTTP の本サーバと使うためにデザインされています。もしあなたが HTTP サーバ実装を書いている場合、おそらく BaseCGIHandler でなくこれをサブクラス化したいことでしょう。

このクラスは BaseHandler のサブクラスです。これは __init__()、get_stdin()、get_stderr()、add_cgi_vars()、_write()、_flush() をオーバーライドして、コンストラクタから明示的に環境とストリームを設定するようにしています。与えられた環境とストリームは stdin、stdout、stderr それに environ 属性に保存されています。

class BaseHandler()

これは WSGI アプリケーションを実行するための抽象ベースクラスです。原理上は複数のリクエスト用に再利用可能なサブクラスを作成することができますが、それぞれのインスタンスは一つの HTTP リクエストを処理します。

BaseHandler インスタンスは外部からの利用にたった一つのメソッドを持ちます：

run(*app*)

指定された WSGI アプリケーション、*app* を実行します。

その他の全ての BaseHandler のメソッドはアプリケーション実行プロセスでこのメソッドから呼ばれます。ですので、主にそのプロセスのカスタマイズのために存在しています。

以下のメソッドはサブクラスでオーバーライドされなければいけません：

_write(*data*)

文字列の *data* をクライアントへの転送用にバッファします。このメソッドが実際にデータを転送しても OK です：下部システムが実際にそのような区別をしている場合に効率をより良くするために、BaseHandler は書き出しとフラッシュ操作を分けているからです。

_flush()
バッファされたデータをクライアントに強制的に転送します。このメソッドは何もしなくても OK です（すなわち、_write() が実際にデータを送る場合）。

get_stdin()
現在処理中のリクエストの wsgi.input としての利用に適当な入力ストリームオブジェクトを返します。

get_stderr()
現在処理中のリクエストの wsgi.errors としての利用に適当な出力ストリームオブジェクトを返します。

add_cgi_vars()
現在のリクエストの CGI 変数を environ 属性に追加します。

これらがオーバーライドするであろうメソッド及び属性です。しかしながら、このリストは単にサマリであり、オーバーライド可能な全てのメソッドは含んでいません。カスタマイズした BaseHandler サブクラスを作成しようとする前にドキュメント文字列 (docstrings) やソースコードでさらなる情報を調べてください。

WSGI 環境のカスタマイズのための属性とメソッド：

wsgi_multithread
wsgi.multithread 環境変数で使われる値。デフォルトは BaseHandler では true ですが、別のサブクラスではデフォルトで（またはコンストラクタによって設定されて）異なる値を持つことがあります。

wsgi_multiprocess
wsgi.multiprocess 環境変数で使われる値。デフォルトは BaseHandler では true ですが、別のサブクラスではデフォルトで（またはコンストラクタによって設定されて）異なる値を持つことがあります。

wsgi_run_once
wsgi.run_once 環境変数で使われる値。デフォルトは BaseHandler では false ですが、CGIHandler はデフォルトでこれを true に設定します。

os_environ
全てのリクエストの WSGI 環境に含まれるデフォルトの環境変数。デフォルトでは、wsgiref.handlers がインポートされた時点ではこれは os.environ のコピーですが、サブクラスはクラスまたはインスタンスレベルでそれら自身のものを作ることができます。デフォルト値は複数のクラスとインスタンスで共有されるため、この辞書は読み取り専用と考えるべきだという点に注意してください。

server_software
origin_server 属性が設定されている場合、この属性の値がデフォルトの SERVER_SOFTWARE WSGI 環境変数の設定や HTTP レスポンス中のデフォルトの Server: ヘッダの設定に使われます。これは（BaseCGIHandler や CGIHandler のような）HTTP オリジンサーバでないハンドラでは無視されます。

get_scheme()
現在のリクエストで使われている URL スキームを返します。デフォルト実装は wsgiref.util の guess_scheme() を使い、現在のリクエストの envion 変数に基づいてスキームが"http" か "https" かを推測します。

setup_environ()
environ 属性を、全てを導入済みの WSGI 環境に設定します。デフォルトの実装は、上記全てのメソッドと属性、加えて get_stdin()、get_stderr()、add_cgi_vars() メソッドと wsgi_file_wrapper 属性を利用します。これは、キーが存在せず、origin_

server 属性が true 値で server_software 属性も設定されている場合に SERVER_-SOFTWARE を挿入します。

例外処理のカスタマイズのためのメソッドと属性：

log_exception(*exc_info*)

exec_info タプルをサーバログに記録します。*exc_info* は（*type*, *value*, *traceback*）のタプルです。デフォルトの実装は単純にトレースバックをリクエストの wsgi.errors ストリームに書き出してフラッシュします。サブクラスはこのメソッドをオーバーライドしてフォーマットを変更したり出力先の変更、トレースバックを管理者にメールしたりその他適切と思われるいかなるアクションも取ることができます。

traceback_limit

デフォルトの log_exception() メソッドで出力されるトレースバック出力に含まれる最大のフレーム数です。None ならば、全てのフレームが含まれます。

error_output(*environ, start_response*)

このメソッドは、ユーザに対してエラーページを出力する WSGI アプリケーションです。これはクライアントにヘッダが送出される前にエラーが発生した場合にのみ呼び出されます。

このメソッドは sys.exc_info() を使って現在のエラー情報にアクセスでき、その情報はこれを呼ぶときに *start_response* に渡すべきです（PEP 333 の "Error Handling" セクションに記述があります）。

デフォルト実装は単に error_status、error_headers、そして error_body 属性を出力ページの生成に使います。サブクラスではこれをオーバーライドしてもっと動的なエラー出力をすることが出来ます。

しかし、セキュリティの観点からは診断をあらゆる老練ユーザに吐き出すことは推奨されないことに気をつけてください；理想的には、診断的な出力を有効にするには何らかの特別なことをする必要があるようにすべきで、これがデフォルト実装では何も含まれていない理由です。

error_status

エラーレスポンスで使われる HTTP ステータスです。これは PEP 333 で定義されているステータス文字列です；デフォルトは 500 コードとメッセージです。

error_headers

エラーレスポンスで使われる HTTP ヘッダです。これは PEP 333 で述べられているような、WSGI レスポンスヘッダ（（*name*, *value*）タプル）のリストであるべきです。デフォルトのリストはコンテントタイプを text/plain にセットしているだけです。

error_body

エラーレスポンスボディ。これは HTTP レスポンスのボディ文字列であるべきです。これはデフォルトではプレーンテキストで "A server error occurred. Please contact the administrator." です。

PEP 333 の "オプションのプラットフォーム固有のファイルハンドリング" 機能のためのメソッドと属性：

wsgi_file_wrapper

wsgi.file_wrapper ファクトリ、または None です。この属性のデフォルト値は wsgiref.util の FileWrapper クラスです。

sendfile()

オーバーライドしてプラットフォーム固有のファイル転送を実装します。このメソッドはアプリケーションの戻り値が wsgi_file_wrapper 属性で指定されたクラスのインスタンスの場合にのみ呼ばれます。これはファイルの転送が成功できた場合には true を返して、デフォルト

の転送コードが実行されないようにするべきです。このデフォルトの実装は単に false 値を返します。

その他のメソッドと属性：

origin_server
この属性はハンドラの `_write()` と `_flush()` が、特別に `Status:` ヘッダに HTTP ステータスを求めるような CGI 状のゲートウェイプロトコル経由でなく、クライアントと直接通信をするような場合には true 値に設定されているべきです。
この属性のデフォルト値は `BaseHandler` では true ですが、`BaseCGIHandler` と `CGIHandler` では false です。

http_version
`origin_server` が true の場合、この文字列属性はクライアントへのレスポンスセットの HTTP バージョンの設定に使われます。デフォルトは `"1.0"` です。

18.5 `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。いくつかの制限はあります — 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、イテレータプロトコルも正しくサポートしています。注意: `read()` の引数を省略または負の値を指定しても、データストリームの最後まで読みこむ訳ではありません。ソケットからすべてのストリームを読み込んだことを決定する一般的な方法は存在しません。

`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 3.9 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 `mimetools.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応答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 = {'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。

2.5 で変更された仕様: `'urlretrieve()'` は、予想 (これは *Content-Length* ヘッダにより通知されるサイズです) よりも取得できるデータ量が少ないことを検知した場合、`ContentTooShortError` を発生します。これは、例えば、ダウンロードが中断された場合などに発生します。

Content-Length は下限として扱われます: より多いデータがある場合、urlretrieve はそのデータを読みますが、より少ないデータしか取得できない場合、これは exception を発生します。

このような場合にもダウンロードされたデータを取得することは可能で、これは exception インスタンスの `content` 属性に保存されています。

Content-Length ヘッダが無い場合、urlretrieve はダウンロードされたデータのサイズをチェックできず、単にそれを返します。この場合は、ダウンロードは成功したと見なす必要があります。

`_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):
    version = "App/1.7"

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('/%7Econnolly/')` 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シーケンスを、"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 '&' で区切ったシーケンスで、*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、シーケンス *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シーケンスが引数 *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シーケンス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、サブクラス定義においてクラス属性 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:' スキームを使った際のクライアント認証に使われることがあります。キーワード引数 *key_file* および *cert_file* が SSL 鍵と証明書を設定するためにサポートされています; クライアント認証をするには両方が必要です。

URLopener オブジェクトは、サーバがエラーコードを返した時には IOError を発生します。

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 authintication) 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。

その他のレスポンスコードについては、http_error_default() が呼ばれます。これはサブクラ

スでエラーを適切に処理するようにオーバーライドすることができます。

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()` メソッドをオーバライドしてもかまいません。

exception `ContentTooShortError` (*msg*[, *content*])

この例外は `urlretrieve()` 関数が、ダウンロードされたデータの量が予期した量 (*Content-Length* ヘッダで与えられる) よりも少ないことを検知した際に発生します。`content` 属性には (恐らく途中までの) ダウンロードされたデータが格納されています。 2.5 で追加された仕様です。

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 (画像データ等) 、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。

  FTP プロトコルを扱うコードでは、ファイルとディレクトリを区別できません。このことから、アクセスできないファイルを指している URL からデータを読み出そうとすると、予期しない動作を引き起こす場合があります。 URL が / で終わっていれば、ディレクトリを指しているものとみなして、それに適した処理を行います。しかし、ファイルの読み出し操作が 550 エラー (URL が存在しないか、主にパーミッションの理由でアクセスできない) になった場合、URL がディレクトリを指していて、末尾の / を忘れたケースを処理するため、パスをディレクトリとして扱います。このために、パーミッションのためにアクセスできないファイルを fetch しようとすると、FTP コードはそのファイルを開こうとして 550 エラーに陥り、次にディレクトリ一覧を表示しようとするため、誤解を生むような結果を引き起こす可能性があるのです。よく調整された制御が必要なら、`ftplib` モジュールを使

うか、FancyURLOpener をサブクラス化するか、*_urlopener* を変更して目的に合わせるよう検討してください。

- 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。

18.5.1 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で対話型モデルを使うには、このメソッドをオーバライドしなければなりません。

18.5.2 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()
```

18.6 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、クッキー、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 。

data はサーバに送信する追加のデータを示す文字列か、そのようなデータが無ければ *None* を指定します。現時点で 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; *data* パラメタが指定が指定された場合、HTTP リクエストは GET でなく POST になります。 *data* は標準的な application/x-www-form-urlencoded 形式のバッファでなくてはなりません。 urllib.urlencode() 関数はマップ型か 2 タプルのシーケンスを取り、この形式の文字列を返します。

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。

どのハンドラもリクエストを処理しなかった場合には `None` を返すことがあるので注意してください (デフォルトでインストールされるグローバルハンドラの `OpenerDirector` は、`UnknownHandler` を使って上記の問題が起きないようにしています)。

`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`, `HTTPErrorProcessor`

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。

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*][, *origin_req_host*][, *unverifiable*])

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 はサーバに送信する追加のデータを示す文字列か、そのようなデータが無ければ *None* を指定します。現時点で 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; *data* パラメタが指定が指定された場合、HTTP リクエストは GET でなく POST になります。 *data* は標準的な application/x-www-form-urlencoded 形式のバッファでなくてはなりません。 `urllib.urlencode()` 関数はマップ型か 2 タプルのシーケンスを取り、この形式の文字列を返します。

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。

最後の二つの引数は、サードパーティの HTTP クッキーを正しく扱いたい場合にのみ関係してきます:

origin_req_host は、RFC 2965 で定義されている元のトランザクションにおけるリクエストホスト (request-host of the origin transaction) です。デフォルトの値は `cookielib.request_host(self)` です。この値は、ユーザによって開始された元々のリクエストにおけるホスト名や IP アドレスです。例えば、もしリクエストがある HTML ドキュメント内の画像を指していれば、この値は画像を含んでいるページへのリクエストにおけるリクエストホストになるはずです。

unverifiable は、RFC 2965 の定義において、該当するリクエストが証明不能 (unverifiable) であるかどうかを示します。デフォルトの値は False です。証明不能なリクエストとは、ユーザが受け入れの可否を選択できないような URL を持つリクエストのことです。例えば、リクエストが HTML ドキュメント中の画像であり、ユーザがこの画像を自動的に取得するかどうかを選択できない場合には、証明不能フラグは True になります。

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 `HTTPCookieProcessor`([*cookiejar*])
HTTP Cookie を扱うためのクラスです。

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 18.6.7 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 18.6.7 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 18.6.7 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 18.6.7 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 18.6.7 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 18.6.7 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。

18.6.1 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す。HTTP ハンドラでは、データはバイト文字列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 から 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' か 'POST' のいずれかの値を返します。

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す。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。

add_unredirected_header(*key, header*)
リダイレクトされたリクエストには追加されないヘッダを追加します。 2.4 で追加された仕様です。

has_header(*header*)
インスタンスが名前つきヘッダであるかどうかを (通常のヘッダと非リダイレクトヘッダの両方を調べて) 返します。 2.4 で追加された仕様です。

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()
セレクタ — 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との設定と置き換えられます。インスタンスのセレ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。

get_origin_req_host()
RFC 2965 の定義よる、始原トランザクションのリクエストホストを返します。`Request` コンストラクタのドキュメントを参照してください。

is_unverifiable()
リクエストが RFC 2965 の定義における証明不能リクエストであるかどうかを返します。`Request` コンストラクタのドキュメントを参照してください。

18.6.2 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 (HTTP エラーは特別扱いされているので注意してください)。

- *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。
- `http_error_`*type*`()` — ハンドラが HTTP エラーコード *type* の処理方法を知っていることを示すシグナルです。
- *protocol*`_error()` — ハンドラが (`http` でない) *protocol* のエラーを処理する方法を知ってい

ることを示すシグナルです。

•*protocol*_request() — ハンドラが *protocol* リクエストのプリプロセス方法を知っていることを示すシグナルです。

•*protocol*_response() — ハンドラが *protocol* リクエストのポストプロセス方法を知っていることを示すシグナルです。

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、返り値、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グローバルな 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; ハンドラクラスの http_error_*() メソッドを参照してください。

返り値および送出される例外は urlopen() と同じものです。

OpenerDirector オブジェクトは、以下の 3 つのステージに分けて URL を開きます:

各ステージで OpenerDirector オブジェクトのメソッドがどのような順で呼び出されるかは、ハンドラインスタンスの並び方で決まります。

1. *protocol*_request() 形式のメソッドを持つ全てのハンドラに対してそのメソッドを呼び出し、リクエストのプリプロセスを行います。

2. *protocol*_open() 形式のメソッドを持つハンドラを呼び出し、リクエストを処理します。このステージは、ハンドラが None でない値 (すなわちレスポンス) を返すか、例外 (通常は URLError) を送出した時点で終了します。例外は伝播 (propagate) できます。

   実際には、上のアルゴリズムではまず default_open という名前のメソッドを呼び出します。このメソッドが全て None を返す場合、同じアルゴリズムを繰り返して、今度は *protocol*_open() 形式のメソッドを試します。メソッドが全て None を返すと、さらに同じアルゴリズムを繰り返して unknown_open() を呼び出します。

   これらのメソッドの実装には、親となる OpenerDirector インスタンスの .open() や .error() といったメソッド呼び出しが入る場合があるので注意してください。

3. *protocol*_response() 形式のメソッドを持つ全てのハンドラに対してそのメソッドを呼び出し、リクエストのポストプロセスを行います。

18.6.3 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: **注意:** 慣習的に、*protocol*_request() や *protocol*_response() といったメソッドを定義しているサブクラスは *Processor と名づけ、その他は*Handler と名づけることになっています

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 をマップしてはいけません)。

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。

***protocol*_request**(*req*)

このメソッドは BaseHandler では**定義されていません** が、サブクラスで特定のプロトコルリクエ

ストのプリプロセスを行いたい場合には定義せねばなりません。

このメソッドが定義されていると、親となる OpenerDirector から呼び出されます。その際、*req* は Request オブジェクトになります。戻り値は Request オブジェクトでなければなりません。

***protocol*_response**(*req, response*)

このメソッドは BaseHandler では**定義されていません** が、サブクラスで特定のプロトコルリクエストのポストプロセスを行いたい場合には定義せねばなりません。

このメソッドが定義されていると、親となる OpenerDirector から呼び出されます。その際、*req* は Request オブジェクトになります。*response* は urlopen() の戻り値と同じインタフェースを実装したオブジェクトになります。戻り値もまた、urlopen() の戻り値と同じインタフェースを実装したオブジェクトでなければなりません。

18.6.4 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、他のどのハンドラにもこの 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ハンドラ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。

18.6.5 HTTPCookieProcessor オブジェクト

2.4 で追加された仕様です。

HTTPCookieProcessor インスタンスは属性をひとつだけ持ちます:

cookiejar

クッキーの入っている cookielib.CookieJar オブジェクトです。

18.6.6 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。

18.6.7 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 からなるシーケンスでもかまいません。*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。

18.6.8 AbstractBasicAuthHandler オブジェクト

http_error_auth_reqed(*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 URL とパスを指定します、*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。

host は、オーソリティ (例 "python.org") か、オーソリティコンポーネント を含む URL (例 "http://python.org") です。どちらの場合も、オーソリティはユーザ情報コンポーネントを含んではいけません (なので、"python.org" や "python.org:80" は正しく、"joe:password@python.org" は不正です)。

18.6.9 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。

18.6.10 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。

18.6.11 AbstractDigestAuthHandler オブジェクト

http_error_auth_reqed(*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。

18.6.12 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。

18.6.13 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。

18.6.14 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。

18.6.15 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。

18.6.16 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。

18.6.17 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ネームおよびパスワードで行われます。

18.6.18 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。

18.6.19 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。

18.6.20 UnknownHandler オブジェクト

unknown_open()
例外 `URLError` を送出します。

18.6.21 HTTPErrorProcessor オブジェクト

2.4 で追加された仕様です。

unknown_open()
HTTP エラーレスポンスを処理します。

エラーコード 200 の場合、レスポンスオブジェクトを即座に返します。

200 以外のエラーコードの場合、`OpenerDirector.error()` を介して *protocol*`_error_`*code*`()` メソッドに仕事を引き渡します。最終的にどのハンドラもエラーを処理しなかった場合、`urllib2.HTTPDefaultErrorHandler` が `HTTPError` を送出します。

18.6.22 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。この例は Python が SSL をサポートしている場合にのみ動作することに注意してください。

```
>>> 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
```

以下はベーシック HTTP 認証の例です:

```
import urllib2
# ベーシック HTTP 認証をサポートする OpenerDirector を作成する...
auth_handler = urllib2.HTTPBasicAuthHandler()
auth_handler.add_password('realm', 'host', 'username', 'password')
opener = urllib2.build_opener(auth_handler)
# ...urlopen から利用できるよう、グローバルにインストールする
urllib2.install_opener(opener)
urllib2.urlopen('http://www.example.com/login.html')
```

build_opener() はデフォルトで沢山のハンドラを提供しており、その中に ProxyHandler があります。デフォルトでは、ProxyHandler は<scheme>_proxy という環境変数を使います。ここで<scheme> は URL スキームです。例えば、 HTTP プロキシの URL を得るには、環境変数 http_proxy を読み出します。

この例では、デフォルトの ProxyHandler を置き換えてプログラム的に作成したプロキシ URL を使うようにし、ProxyBasicAuthHandler でプロキシ認証サポートを追加します。

```
proxy_handler = urllib2.ProxyHandler({'http': 'http://www.example.com:3128/'})
proxy_auth_handler = urllib2.HTTPBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'password')

opener = build_opener(proxy_handler, proxy_auth_handler)
# 今回は OpenerDirector をインストールするのではなく直接使います:
opener.open('http://www.example.com/login.html')
```

以下は HTTP ヘッダを追加する例です:

headers 引数を使って Request コンストラクタを呼び出す方法の他に、以下のようにできます:

```
import urllib2
req = urllib2.Request('http://www.example.com/')
req.add_header('Referer', 'http://www.python.org/')
r = urllib2.urlopen(req)
```

OpenerDirector は全ての Request に User-Agent: ヘッダを自動的に追加します。これを変更するには:

```
import urllib2
opener = urllib2.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
opener.open('http://www.example.com/')
```

のようにします。

また、Request が urlopen() (や OpenerDirector.open()) に渡される際には、いくつかの標準ヘッダ (Content-Length:, Content-Type: および Host:) も追加されることを忘れないでください。

18.7 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。通常、このモジュールは直接使いません — urllib モジュールが HTTP や HTTPS を使った URL を扱う上でこのモジュールを使います。**注意:** HTTPS のサポートは、SSL をサポートするように socket

モジュールをコンパイルした場合にのみ利用できます。

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。ポート番号を指定しなかった場合、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) を使います。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 = httplib.HTTPConnection('www.cwi.nl')
>>> h2 = httplib.HTTPConnection('www.cwi.nl:80')
>>> h3 = httplib.HTTPConnection('www.cwi.nl', 80)
```

class HTTPSConnection (*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 のサブクラスです。ポート番号を指定したものの、その値が数字でなかったり空のオブジェクトであった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。

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。

また、整数の状態コードについて以下の定数が定義されています:

Constant	Value	Definition
CONTINUE	100	HTTP/1.1, RFC 2616, Section 10.1.1
SWITCHING_PROTOCOLS	101	HTTP/1.1, RFC 2616, Section 10.1.2
PROCESSING	102	WEBDAV, RFC 2518, Section 10.1
OK	200	HTTP/1.1, RFC 2616, Section 10.2.1
CREATED	201	HTTP/1.1, RFC 2616, Section 10.2.2
ACCEPTED	202	HTTP/1.1, RFC 2616, Section 10.2.3
NON_AUTHORITATIVE_INFORMATION	203	HTTP/1.1, RFC 2616, Section 10.2.4
NO_CONTENT	204	HTTP/1.1, RFC 2616, Section 10.2.5
RESET_CONTENT	205	HTTP/1.1, RFC 2616, Section 10.2.6
PARTIAL_CONTENT	206	HTTP/1.1, RFC 2616, Section 10.2.7
MULTI_STATUS	207	WEBDAV RFC 2518, Section 10.2
IM_USED	226	Delta encoding in HTTP, RFC 3229, Section 10.4.1
MULTIPLE_CHOICES	300	HTTP/1.1, RFC 2616, Section 10.3.1
MOVED_PERMANENTLY	301	HTTP/1.1, RFC 2616, Section 10.3.2
FOUND	302	HTTP/1.1, RFC 2616, Section 10.3.3
SEE_OTHER	303	HTTP/1.1, RFC 2616, Section 10.3.4
NOT_MODIFIED	304	HTTP/1.1, RFC 2616, Section 10.3.5
USE_PROXY	305	HTTP/1.1, RFC 2616, Section 10.3.6
TEMPORARY_REDIRECT	307	HTTP/1.1, RFC 2616, Section 10.3.8
BAD_REQUEST	400	HTTP/1.1, RFC 2616, Section 10.4.1
UNAUTHORIZED	401	HTTP/1.1, RFC 2616, Section 10.4.2
PAYMENT_REQUIRED	402	HTTP/1.1, RFC 2616, Section 10.4.3
FORBIDDEN	403	HTTP/1.1, RFC 2616, Section 10.4.4
NOT_FOUND	404	HTTP/1.1, RFC 2616, Section 10.4.5
METHOD_NOT_ALLOWED	405	HTTP/1.1, RFC 2616, Section 10.4.6
NOT_ACCEPTABLE	406	HTTP/1.1, RFC 2616, Section 10.4.7
PROXY_AUTHENTICATION_REQUIRED	407	HTTP/1.1, RFC 2616, Section 10.4.8
REQUEST_TIMEOUT	408	HTTP/1.1, RFC 2616, Section 10.4.9
CONFLICT	409	HTTP/1.1, RFC 2616, Section 10.4.10
GONE	410	HTTP/1.1, RFC 2616, Section 10.4.11
LENGTH_REQUIRED	411	HTTP/1.1, RFC 2616, Section 10.4.12
PRECONDITION_FAILED	412	HTTP/1.1, RFC 2616, Section 10.4.13
REQUEST_ENTITY_TOO_LARGE	413	HTTP/1.1, RFC 2616, Section 10.4.14
REQUEST_URI_TOO_LONG	414	HTTP/1.1, RFC 2616, Section 10.4.15
UNSUPPORTED_MEDIA_TYPE	415	HTTP/1.1, RFC 2616, Section 10.4.16
REQUESTED_RANGE_NOT_SATISFIABLE	416	HTTP/1.1, RFC 2616, Section 10.4.17
EXPECTATION_FAILED	417	HTTP/1.1, RFC 2616, Section 10.4.18
UNPROCESSABLE_ENTITY	422	WEBDAV, RFC 2518, Section 10.3
LOCKED	423	WEBDAV RFC 2518, Section 10.4
FAILED_DEPENDENCY	424	WEBDAV, RFC 2518, Section 10.5
UPGRADE_REQUIRED	426	HTTP Upgrade to TLS, RFC 2817, Section 6
INTERNAL_SERVER_ERROR	500	HTTP/1.1, RFC 2616, Section 10.5.1
NOT_IMPLEMENTED	501	HTTP/1.1, RFC 2616, Section 10.5.2
BAD_GATEWAY	502	HTTP/1.1 RFC 2616, Section 10.5.3
SERVICE_UNAVAILABLE	503	HTTP/1.1, RFC 2616, Section 10.5.4
GATEWAY_TIMEOUT	504	HTTP/1.1 RFC 2616, Section 10.5.5
HTTP_VERSION_NOT_SUPPORTED	505	HTTP/1.1, RFC 2616, Section 10.5.6
INSUFFICIENT_STORAGE	507	WEBDAV, RFC 2518, Section 10.6
NOT_EXTENDED	510	An HTTP Extension Framework, RFC 2774, Section 7

responses
このディクショナリは、HTTP 1.1 ステータスコードを W3C の名前にマップしたものです。

たとえば `httplib.responses[httplib.NOT_FOUND]` は `'Not Found'` となります。 2.5 で追加された仕様です。

18.7.1 HTTPConnection オブジェクト

`HTTPConnection` インスタンスには以下のメソッドがあります:

request(*method, url*[, *body*[, *headers*]])
このメソッドは、 HTTP 要求メソッド *method* およびセレクタ *url* を使って、要求をサーバに送ります。*body* 引数を指定する場合、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。**注意:** すべての応答を読み込んでからでなければ新しい要求をサーバに送ることはできないことに注意しましょう。

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。

上で説明した `request()` メソッドを使うかわりに、以下の 4 つの関数を使用して要求をステップバイステップで送信することもできます。

putrequest(*request, selector*[, *skip_host*[, *skip_accept_encoding*]])
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。`Host:` や `Accept-Encoding:` ヘッダの自動送信を無効にしたい場合 (例えば別のコンテンツエンコーディングを受け入れたい場合) には、*skip_host* や *skip_accept_encoding* を偽でない値に設定してください。

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。追加の引数を指定した場合、継続して各行にタブ一つと引数の入った引数行が送信されます。

endheaders()
サーバに空行を送り、ヘッダ部が終了したことを通知します。

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。

18.7.2 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。

getheaders()
(header, value) のタプルからなるリストを返します。 2.4 で追加された仕様です。

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。

18.7.3 例

以下は 'GET' リクエストの送信方法を示した例です:

```
>>> import httplib
>>> conn = httplib.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 'POST' リクエストの送信方法を示した例です:

```
>>> import httplib, urllib
>>> params = urllib.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> headers = {"Content-type": "application/x-www-form-urlencoded",
...            "Accept": "text/plain"}
>>> conn = httplib.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()
```

18.8 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 anonyumou
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*, ăracct) 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 (9.4 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/ftpmi rror.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。

18.8.1 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ついています。

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めったにありません。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*[, *ac ct*]]])
: 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リストを受信します。*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_re ply 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。

rmd(*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）。

18.9 gopherlib — gopher プロトコルのクライアント

リリース 2.5 以降で撤廃された仕様です。 gopher プロトコルはあまり利用されていません。

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。

18.10 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 optinal という 2 つのコマンドセットをサポートします。モジュールは POP3_SSL クラスも提供します。このクラスは下位のプロトコルレイヤーに SSL を使った POP3 サーバへの接続を提供します。

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。

class POP3_SSL (*host*[, *port*[, *keyfile*[, *certfile*]]])
: POP3 クラスのサブクラスで、SSL でカプセル化されたソケットによる POP サーバへの接続を提供します。 *port* が指定されていない場合、POP3-over-SSL 標準の 995 番ポートが使われます。*keyfile* と *certfile* もオプションで - SSL 接続に使われる PEM フォーマットの秘密鍵と信頼された＃＃を含みます。

  2.4 で追加された仕様です。

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 (18.11 節):
: The standard Python IMAP module.

Frequently Asked Questions About Fetchmail
(http://www.catb.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。

18.10.1 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。これはデバッギン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、デバッギングアウトプットを表示しません。値を 1 とすると、デバッギン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、デバッギン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', ...]`, *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。

dele(*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ーでは（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 サーバー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。

`POP3_SSL` クラスのインスタンスは追加のメソッドを持ちません。このサブクラスのインターフェイスは親クラスと同じです。

18.10.2 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。

18.11 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イ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、メイ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) と認証チェイン (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、メイ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。

18.11.1 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んでください (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)

以下のコマンドにおける *message_set* オプションは、操作の対象となるひとつあるいは複数のメッセージを指す文字列です。単一のメッセージ番号 ('1') かメッセージ番号の範囲 ('2:4')、あるいは連続していないメッセージをカンマでつなげたもの ('1:3,6:9') となります。範囲指定でアスタリスクを使用すると、上限を無限とすることができます ('3:*')。

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イルボックスに *message* を追加します。

authenticate(*mechanism, authobject*)

認証命令です — 応答の処理が必要です。

mechanism は利用する認証メカニズムを与えます。認証メカニズムはインスタンス変数 `capabilities` の中に `AUTH=mechanism` という形式で現れる必要があります。

authobject は呼び出し可能なオブジェクトである必要があります。

```
data = authobject(response)
```

これはサーバで継続応答を処理するためによばれます。これは (おそらく) 暗号化されて、サーバへ送られた `data` を返します。もしクライアントが中断応答 '`*`' を送信した場合にはこれは `None` を返します。

check()
サーバ上のメイ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イ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イ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イルボックスを生成します。

delete(*mailbox*)
mailbox と名づけられた古いメイルボックスを削除します。

deleteacl(*mailbox, who*)
mailbox における who についての ACL を削除 (権限を削除) します。 2.4 で追加された仕様です。

expunge()
選択されたメイ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) を取りよせます。*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。

getannotation(*mailbox, entry, attribute*)
mailbox に対する '`ANNOTATION`' 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。 2.5 で追加された仕様です。

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* メイ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イルフォルダで、*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*]])
購読しているメイ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イルフォルダで、*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。

myrights(*mailbox*)
mailbox における自分の ACL を返します。(すなわち自分が mailbox で持っている権限を返します。) 2.4 で追加された仕様です。

namespace()
RFC2342 で定義される IMAP 名前空間を返します。 2.3 で追加された仕様です。

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イ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 という名前のメイルボックスを *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イルボックスから検索します。*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...
typ, msgnums = M.search(None, 'FROM', '"LDJ"')

# or:
typ, msgnums = M.search(None, '(FROM "LDJ")')
```

select([*mailbox*[, *readonly*]])

メイ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、メイ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。

setannotation(*mailbox, entry, attribute*[, ...])

'ANNOTATION' 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。 2.5 で追加された仕様です。

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_criterium* 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、メイ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*)

メイ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。*command* は RFC 2060 のセクション 6.4.6 で指定されているもので、"FLAGS", "+FLAGS", あるいは "-FLAGS" のいずれかとなります。オプションで末尾に ".SILENT" がつくこともあります。

たとえば、すべてのメッセージに削除フラグを設定するには次のようにします。

```
typ, data = M.search(None, 'ALL')
for num in data[0].split():
   M.store(num, '+FLAGS', '\\Deleted')
M.expunge()
```

subscribe(*mailbox*)
新たなメイルボックスを購読 (subscribe) します。

thread(*threading_algorithm, charset, search_criterion*[, ...])
thread コマンドは search にスレッドの概念を加えた変形版でス。返されるデータは空白で区切られたスレッドメンバのリストを含んでいます。

各スレッドメンバは 0 以上のメッセージ番号からなり、空白で区切られて おり、親子関係を示しています。

thread コマンドは *search_criterion* 引数の前に 2 つの引数を持っています。*threading_algorithm* と *charset* です。search コマンドとは違い、*charset* は必須です。search に対する uid search と同様に、 thread にも uid thread があります。

thread コマンドはまずメールボックス中のメッセージを、charset を用いた検索条件で検索します。その後マッチしたメッセージを指定されたスレッドアルゴリズムでスレッド化して返します.

これは 'IMAP4rev1' の拡張コマンドです。 2.4 で追加された仕様です。

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*)
古いメイ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。

18.11.2 IMAP4 の使用例

以下にメイルボックスを開き、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.close()
M.logout()
```

18.12 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を処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*] [, *usenetrc*]]]])

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て *usenetrc* が真 (デフォルト) の場合、'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なことがあります。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* のデフォルト値は None です。*usenetrc* のデフォルト値は True です。

2.4 で変更された仕様: *usenetrc* 引数を追加しました

exception 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。

exception 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。

exception 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。

exception 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。

exception 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。

exception 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。

18.12.1 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、'NEWGROUPS' 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 `newsgrups()` 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* はメッセージ 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。

descriptions (*grouppattern*)

'LIST NEWSGROUPS' 命令を送信します。*grouppattern* は RFC2980 の定義に従う wildmat 文字列です (実際には、 DOS や UNIX のシェルワイルドカード文字列と同じです)。(*response*, *list*) からなるペアを返し、*list* はタプル (*name*, *title*) リストになります。 2.4 で追加された仕様です。

description (*group*)

単一のグループ *group* から説明文字列を取り出します。('group' が実際には wildmat 文字列で) 複数のグループがマッチした場合、最初にマッチしたものを返します。何もマッチしなければ空文字列を返します。

このメソッドはサーバからの応答コードを省略します。応答コードが必要なら、`descriptions()` を使ってください。

2.4 で追加された仕様です。

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) 記事番号、*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。*id* は ('<' と '>' に囲まれた) メッセージ ID です。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。

RFC2980 では、"この拡張は撤廃すべきである" と主張しています。`descriptions()` または `description()` を使うようにしてください。

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、NTTP 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。

18.13 smtplib — SMTP プロトコル クライアント

`smtplib` 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イ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 の命令をサポートをします。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
サーバーが '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。

18.13.1 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に役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() が呼びだすため、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 サービスセットに含まれている場合には True を返し、そうでなければ False を返します。大小文字は区別されません。

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:

SMTPHeloError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() をもう一度呼びだ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 アドレス文字列または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戻りますが、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、例外を投げれなかった場合は、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、サーバーによって送られた 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、ドキュメントもありません。詳細はモジュールのコードを調べてください。

18.13.2 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()
```

18.14 smtpd — SMTP サーバ

このモジュールでは、 SMTP サーバを実装するためのクラスをいくつか提供しています。一つは何も行わない、オーバライドできる汎用のサーバで、その他の二つでは特定のメール送信ストラテジを提供しています。

18.14.1 SMTPServer オブジェクト

class SMTPServer(*localaddr, remoteaddr*)

新たな SMTPServer オブジェクトを作成します。このオブジェクトはローカルのアドレス *localaddr* に関連づけ (bind) されます。オブジェクトは *remoteaddr* を上流の SMTP リレー先にします。このクラスは asyncore.dispatcher を継承しており、インスタンス化時に自身を asyncore のイベントループに登録します。

process_message(*peer, mailfrom, rcpttos, data*)

このクラスでは NotImplementedError 例外を送出します。受信したメッセージを使って何か意味のある処理をしたい場合にはこのメソッドをオーバライドしてください。コンストラクタの *remoteaddr* に渡した値は_remoteaddr 属性で参照できます。*peer* はリモートホストのアドレスで、*mailfrom* はメッセージエンベロープの発信元 (envelope originator) 、*rcpttos* はメッセージエンベロープの受信対象、そして *data* は電子メールの内容が入った (RFC 2822 形式の) 文字列です。

18.14.2 DebuggingServer オブジェクト

class `DebuggingServer` (*localaddr, remoteaddr*)

新たなデバッグ用サーバを生成します。引数は `SMTPServer` と同じです。メッセージが届いても無視し、標準出力に出力します。

18.14.3 PureProxy オブジェクト

class `PureProxy` (*localaddr, remoteaddr*)

新たな単純プロキシ (pure proxy) サーバを生成します。引数は `SMTPServer` と同じです。全てのメッセージを *remoteaddr* にリレーします。このオブジェクトを動作させるとオープンリレーを作成してしまう可能性が多分にあります。注意してください。

18.14.4 MailmanProxy Objects

class `MailmanProxy` (*localaddr, remoteaddr*)

新たな単純プロキシサーバを生成します。引数は `SMTPServer` と同じです。全てのメッセージを *remoteaddr* にリレーしますが、ローカルの mailman の設定に *remoteaddr* がある場合には mailman を使って処理します。このオブジェクトを動作させるとオープンリレーを作成してしまう可能性が多分にあります。注意してください。

18.15 `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` に含められていない 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。

18.15.1 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全てを返します。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。

このメソッドはけっ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。

18.15.2 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()
```

18.16 uuid — RFC 4122 に準拠した UUID オブジェクト

2.5 で追加された仕様です。 このモジュールでは immutable（変更不能）な UUID オブジェクト（UUID クラス）と RFC 4122 の定めるバージョン 1、3、4、5 の UUID を生成するための uuid1()、uuid2()、uuid3()、uuid4()、uuid() が提供されています。

もしユニークな ID が必要なだけであれば、おそらく uuid1() か uuid4() をコールすれば良いでしょう。uuid1() はコンピュータのネットワークアドレスを含む UUID を生成するためにプライバシーを侵害するかもしれない点に注意してください。uuid4() はランダムな UUID を生成します。

class UUID(*[hex[, bytes[, bytes_le[, fields[, int[, version]]]]]]*)

32 桁の 16 進数文字列、*bytes* に 16 バイトの文字列、*bytes_le* 引数に 16 バイトのリトルエンディアンの文字列、*field* 引数に 6 つの整数のタプル（32 ビット *time_low*、16 ビット *time_mid*、16 ビット *time_hi_version*, 8 ビット *clock_seq_hi_variant*, 8 ビット *clock_seq_low*, 48 ビット *node*）、または *int* に一つの 128 ビット整数のいずれかから UUID を生成します。16 進数が与えられた時、波括弧、ハイフン、それと URN 接頭辞は無視されます。例えば、これらの表現は全て同じ UUID を払い出します。

```
UUID('{12345678-1234-5678-1234-567812345678}')
UUID('12345678123456781234567812345678')
UUID('urn:uuid:12345678-1234-5678-1234-567812345678')
UUID(bytes='\x12\x34\x56\x78'*4)
UUID(bytes_le='\x78\x56\x34\x12\x34\x12\x78\x56' +
              '\x12\x34\x56\x78\x12\x34\x56\x78')
UUID(fields=(0x12345678, 0x1234, 0x5678, 0x12, 0x34, 0x567812345678))
UUID(int=0x12345678123456781234567812345678)
```

hex、*bytes*、*bytes_le*、*fields*、または *int* のうち、どれかただ一つだけが与えられなければいけません。*version* 引数はオプションです；与えられた場合、結果の UUID は与えられた *hex*、*bytes*、*bytes_le*、*fields*、または *int* をオーバーライドして、RFC 4122 に準拠した variant と version ナンバーのセットを持つことになります。*bytes_le*, *fields*, or *int*.

UUID インスタンスは以下の読み取り専用属性を持ちます：

bytes

16 バイト文字列（バイトオーダーがビッグエンディアンの 6 つの整数フィールドを持つ）の UUID。

bytes_le

16 バイト文字列（*time_low*、*time_mid*、*time_hi_version* をリトルエンディアンで持つ）の UUID。

fields

UUID の 6 つの整数フィールドを持つタプルで、これは 6 つの個別の属性と 2 つの派生した属性としても取得可能です。

フィールド	意味
time_low	UUID の最初の 32 ビット
time_mid	UUID の次の 16 ビット
time_hi_version	UUID の次の 16 ビット
clock_seq_hi_variant	UUID の次の 8 ビット
clock_seq_low	UUID の次の 8 ビット
node	UUID の最後の 48 ビット
time	60 ビットのタイムスタンプ
clock_seq	14 ビットのシーケンス番号

hex

32 文字の 16 進数文字列での UUID。

int

128 ビット整数での UUID。

urn

RFC 4122 で規定される URN での UUID。

variant

UUID の内部レイアウトを決定する UUID の variant。これは整数の定数 The UUID variant, which determines the internal layout of the UUID. This will be one of the integer constants `RESERVED_NCS`、`RFC_4122`、 `RESERVED_MICROSOFT`、又は `RESERVED_FUTURE` のいずれかになります。

version

UUID の version 番号（1 から 5、variant が `RFC_4122` である場合だけ意味があります)。

The `uuid` モジュールには以下の関数があります：

getnode()

48 ビットの正の整数としてハードウェアアドレスを取得します。最初にこれを起動すると、別個のプログラムが立ち上がって非常に遅くなることがあります。もしハードウェアを取得する試みが全て失敗すると、ランダムな 48 ビットに RFC 4122 で推奨されているように 8 番目のビットを 1 に設定した数を使います。"ハードウェアアドレス" とはネットワークインターフェースの MAC アドレスを指し、複数のネットワークインターフェースを持つマシンの場合、それらのどれか一つの MAC アドレスが返るでしょう。

uuid1(*[node[, clock_seq]]*)

UUID をホスト ID、シーケンス番号、現在時刻から生成します。*node* が与えられなければ、`getnode()` がハードウェアアドレス取得のために使われます。*clock_seq* が与えられると、これはシーケンス番号として使われます；さもなくば 14 ビットのランダムなシーケンス番号が選ばれます。

uuid3(*namespace, name*)

UUID を名前空間識別子（これは UUID です）と名前（文字列です）の MD5 ハッシュから生成します。

uuid4()

ランダムな UUID を生成します。

uuid5(*namespace, name*)

名前空間識別子（これは UUID です）と名前（文字列です）の SHA-1 ハッシュから生成します。

`uuid` モジュールは `uuid3()` または `uuid5()` で利用するために次の名前空間識別子を定義しています。

NAMESPACE_DNS

この名前空間が指定された場合、*name* 文字列は完全修飾ドメイン名です。

NAMESPACE_URL

この名前空間が指定された場合、*name* 文字列は URL です。

NAMESPACE_OID

この名前空間が指定された場合、*name* 文字列は ISO OID です。

NAMESPACE_X500

この名前空間が指定された場合、*name* 文字列は X.500 DN の DER またはテキスト出力形式です。

The `uuid` モジュールは以下の定数を `variant` 属性が取りうる値として定義しています：

RESERVED_NCS

NCS 互換性のために予約されています。

RFC_4122

RFC 4122 で与えられた UUID レイアウトを指定します。

RESERVED_MICROSOFT

Microsoft の互換性のために予約されています。

RESERVED_FUTURE

将来のために予約されています。

参考資料:

RFC 4122, "*A Universally Unique IDentifier (UUID) URN Namespace*"

この仕様は UUID のための Uniform Resource Name 名前空間、UUID の内部フォーマットと UUID の生成方法を定義しています。

18.16.1 例

典型的な `uuid` モジュールの利用方法を示します：

```
>>> import uuid

# UUID をホスト ID と現在時刻に基づいて生成します
>>> uuid.uuid1()
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

# 名前空間 UUID と名前の MD5 ハッシュを使って UUID を生成します
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
UUID('6fa459ea-ee8a-3ca4-894e-db77e160355e')

# ランダムな UUID を作成します
>>> uuid.uuid4()
UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

# 名前空間 UUID と名前の SHA-1 ハッシュを使って UUID を生成します
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
UUID('886313e1-3b8a-5372-9b90-0c9aee199e5d')

# 16 進数文字列から UUID を生成します（波括弧とハイフンは無視されます）
>>> x = uuid.UUID('{00010203-0405-0607-0809-0a0b0c0d0e0f}')

# UUID を標準的な 16 進数の文字列に変換します
>>> str(x)
'00010203-0405-0607-0809-0a0b0c0d0e0f'

# 生の 16 バイトの UUID を取得します
>>> x.bytes
'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f'

# 16 バイトの文字列から UUID を生成します
>>> uuid.UUID(bytes=x.bytes)
UUID('00010203-0405-0607-0809-0a0b0c0d0e0f')
```

18.17 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！)。サポートされる URL スキームは以下の通りです: file, ftp, gopher, hdl, http, https, imap, mailto, mms, news, nntp, prospero, rsync, rtsp, rtspu, sftp, shttp, sip, sips, snews, svn, svn+ssh, telnet, wais。

2.5 で追加された仕様: sftp および sips スキームのサポートが追加されました

urlparse モジュールには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分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。たとえば以下のようになります。

```
>>> from urlparse import urlparse
>>> o = urlparse('http://www.cwi.nl:80/%7Eguido/Python.html')
>>> o
('http', 'www.cwi.nl:80', '/%7Eguido/Python.html', '', '', '')
>>> o.scheme
'http'
>>> o.port
80
>>> o.geturl()
'http://www.cwi.nl:80/%7Eguido/Python.html'
```

default_scheme 引数が指定されている場合、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 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 引数が偽の場合、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 `True` です。

戻り値は実際には tuple のサブクラスのインスタンスです。このクラスには以下の読み出し専用の便利な属性が追加されています。

属性	インデクス	値	指定されなかった場合の値
scheme	0	URL スキーム	空文字列
netloc	1	ネットワーク上の位置	空文字列
path	2	階層的パス	空文字列
params	3	最後のパス要素に対するパラメータ	空文字列
query	4	クエリ要素	空文字列
fragment	5	フラグメント指定子	空文字列
username		ユーザ名	`None`
password		パスワード	`None`
hostname		ホスト名 (小文字)	`None`
port		ポート番号を表わす整数 (もしあれば)	`None`

結果オブジェクトのより詳しい情報は 18.17.1 節 “`urlparse()` および `urlsplit()` の結果” を参照してください。

2.5 で変更された仕様: 戻り値に属性が追加されました

`urlunparse` (*parts*)

`urlparse()` が返すような形式のタプルから URL を構築します。*parts* 引数は任意の 6 要素イテラブルで構いません。解析された元の URL が、不要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。(例えばクエリ内容が空の ? のようなもので、RFC はこれらを等価だと述べています。)

`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) が必要な場合に、`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。

戻り値は実際には tuple のサブクラスのインスタンスです。このクラスには以下の読み出し専用の便利な属性が追加されています。

属性	インデクス	値	指定されなかった場合の値
scheme	0	URL スキーム	空文字列
netloc	1	ネットワーク上の位置	空文字列
path	2	階層的パス	空文字列
query	3	クエリ要素	空文字列
fragment	4	フラグメント指定子	空文字列
username		ユーザ名	None
password		パスワード	None
hostname		ホスト名 (小文字)	None
port		ポート番号を表わす整数 (もしあれば)	None

結果オブジェクトのより詳しい情報は 18.17.1 節 “urlparse() および urlsplit() の結果” を参照してください。

2.2 で追加された仕様です。 2.5 で変更された仕様: 戻り値に属性が追加されました

urlunsplit(*parts*)

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。*parts* 引数は任意の 5 要素イテラブルで構いません。解析された元の URL が、不要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。(例えばクエリ内容が空の ? のようなもので、RFC はこれらを等価だと述べています。) 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。以下の例のようになります。

```
>>> from urlparse import urljoin
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

allow_fragments 引数は urlparse() における引数と同じ意味とデフォルトを持ちます。

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ダ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。

18.17.1 urlparse() および urlsplit() の結果

urlparse() および urlsplit() から得られる結果オブジェクトはそれぞれ tuple 型のサブクラスです。これらのクラスはそれぞれの関数の説明の中で述べたような属性とともに、追加のメソッドを一つ提供し

ています。

geturl()

再結合された形で元の URL の文字列を返します。この文字列は元の URL とは次のような点で異なるかもしれません。スキームは常に小文字に正規化されます。また空の要素は省略されます。特に、空のパラメータ、クエリ、フラグメント識別子は取り除かれます。

このメソッドの結果は再び解析に回されたとしても不動点となります。

```
>>> import urlparse
>>> url = 'HTTP://www.Python.org/doc/#'

>>> r1 = urlparse.urlsplit(url)
>>> r1.geturl()
'http://www.Python.org/doc/'

>>> r2 = urlparse.urlsplit(r1.geturl())
>>> r2.geturl()
'http://www.Python.org/doc/'
```

2.5 で追加された仕様です。

以下のクラスが解析結果の実装を提供します。

class BaseResult

具体的な結果クラスたちの基底クラスです。このクラスがほとんどの属性の定義を与えます。しかし `geturl()` メソッドは提供しません。このクラスは `tuple` から派生していますが、`__init__()` や `__new__()` をオーバーライドしません。

class ParseResult(*scheme, netloc, path, params, query, fragment*)

`urlparse()` の結果のための具体クラスです。`__new__()` メソッドをオーバーライドして正しい個数の引数が引き渡されたことを確認するようにしています。

class SplitResult(*scheme, netloc, path, query, fragment*)

`urlsplit()` の結果のための具体クラスです。`__new__()` メソッドをオーバーライドして正しい個数の引数が引き渡されたことを確認するようにしています。

18.18 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` 配合クラス (mix-in

classes) を使えば、非同期的な動作をサポートできます。

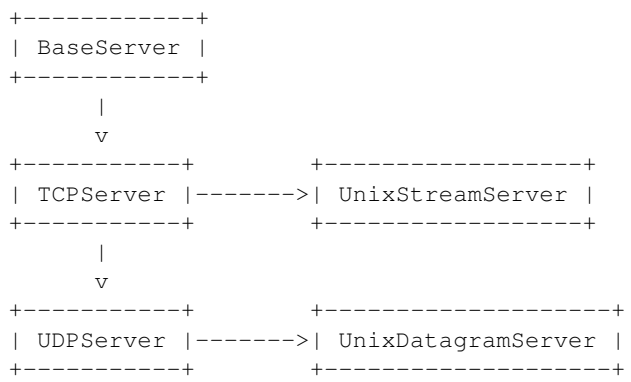
サーバの作成にはいくつかのステップがあります。最初に、BaseRequestHandler クラスをサブクラス化して要求処理クラス (request hander class) を生成し、その handle() メソッドを上書きしなければなりません; このメソッドで入力される要求を処理します。次に、サーバクラスのうち一つをインスタンス化して、サーバのアドレスと要求処理クラスを渡さなければなりません。最後に、サーバオブジェクトの handle_request() または serve_forever() メソッドを呼び出して、単一または多数の要求を処理します。

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:

18.18.1 サーバ生成に関するノート

継承図にある五つのクラスのうち四つは四種類の同期サーバを表わしています。



UnixDatagramServer は UDPServer から派生していて、UnixStreamServer からではないことに注意してください— IP と UNIX ストリームサーバの唯一の違いはアドレスファミリーでそれは両方の UNIX サーバクラスで単純に繰り返されています。

それぞれのタイプのサーバのフォークしたりスレッド実行したりするバージョンは ForkingMixIn および ThreadingMixIn ミクシン (mix-in) クラスを使って作ることができます。たとえば、スレッド実行する UDP サーバクラスは以下のようにして作られます。

```
class ThreadingUDPServer(ThreadingMixIn, UDPServer): pass
```

ミクシンクラスは UDPServer で定義されるメソッドをオーバライドするために、先に来なければなりません。様々なメンバ変数を設定することで元になるサーバ機構の振る舞いを変えられます。

サービスの実装には、BaseRequestHandler からクラスを派生させてその handle() メソッドを再定義しなければなりません。このようにすれば、サーバクラスと要求処理クラスを結合して様々なバージョンのサービスを実行することができます。要求処理クラスはデータグラムサービスかストリームサービスかで異なることでしょう。この違いは処理サブクラス StreamRequestHandler または DatagramRequestHandler を使うという形で隠蔽できます。

もちろん、まだ頭を使わなければなりません! たとえば、サービスがリクエストによっては書き換えられ

るようなメモリ上の状態を使うならば、フォークするサーバを使うのは馬鹿げています。というのも子プロセスでの書き換えは親プロセスで保存されている初期状態にも親プロセスから分配される各子プロセスの状態にも届かないからです。この場合、スレッド実行するサーバを使うことはできますが、共有データの一貫性を保つためにロックを使わなければならなくなるでしょう。

一方、全てのデータが外部に (たとえばファイルシステムに) 保存される HTTP サーバを作っているのだとすると、同期クラスではどうしても一つの要求が処理されている間サービスが「耳の聞こえない」状態を呈することになります — この状態はもしクライアントが要求した全てのデータをゆっくり受け取るととても長い時間続きかねません。こういう場合にはサーバをスレッド実行したりフォークすることが適切です。

ある場合には、要求の一部を同期的に処理する一方で、要求データに依って子プロセスをフォークして処理を終了させる、といった方法も適当かもしれません。こうした処理方法は同期サーバを使って要求処理クラスの `handle()` メソッドの中で自分でフォークするようにして実装することができます。

スレッドも `fork()` もサポートされない環境で (もしくはサービスにとってそれらがあまりに高価についたり不適切な場合に) 多数の同時要求を捌くもう一つのアプローチは、部分的に処理し終えた要求のテーブルを自分で管理し、次にどの要求に対処するか (または新しく入ってきた要求を扱うかどうか) を決めるのに `select()` を使う方法です。これは (もしスレッドやサブプロセスが使えなければ) 特にストリームサービスに対して重要で、そのようなサービスでは各クライアントが潜在的に長く接続し続けます。

18.18.2 Server オブジェクト

`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へ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か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。デフォルトではサーバのソケットを listen するだけです。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 True の場合には要求が処理され、False 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 True を返します。

18.18.3 RequestHandler オブジェクト

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。デフォルト実装では何もしません。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サブクラスの StreamRequestHandler や DatagramRequestHandler 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。

18.19 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_SPAM() 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* クラス変数でみ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 HTTP 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 HTTP 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([*timestamp*])
メッセージヘッダ向けに書式化された、*timestamp*(`time.time()` のフォーマットである必要があります) で与えられた日時を返します。もし *timestamp* が省略された場合には、現在の日時が使われます。

出力は `'Sun, 06 Nov 1994 08:49:37 GMT'` のようになります。 2.5 で追加された仕様: *timestamp* パラメータ

log_date_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 (18.21 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 (18.20 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。

18.20 `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 `application/octet-stream` 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、`index.html` または `index.htm` をこの順序でチェックします。もしファイルを発見できればその内容を、そうでなければディレクトリ一覧を `list_directory()` メソッドで生成して、返します。このメソッドは `os.listdir()` をディレクトリのスキャンに用いており、`listdir()` が失敗した場合には `404` 応答が返されます。

リクエストがファイルに対応付けられた場合、そのファイルを開いて内容を返します。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、コンテントタイプが *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トタイプで、その後にファイルサイズを示す

'Content-Lenght;' ヘッダと、ファイルの更新日時を示す 'Last-Modified:' ヘッダが続きます。

そして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。このファイルはコンテントタイプが text/ で始まっている場合はテキストモードで、そうでなければ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。

2.5 で追加された仕様: 'Last-Modified' ヘッダ

参考資料:

BaseHTTPServer モジュール (18.19 節):

Web サーバおよび要求ハンドラの基底クラス実装。

18.21 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。

注意: CGIHTTPRequestHandler クラスで実行される CGI スクリプトは HTTP コード 200 (スクリプトの出力が後に続く) を実行に先立って出力される (これがステータスコードになります) ため、リダイレクト (コード 302) を行なうことができません。

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 (18.19 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。

18.22 `cookielib` — HTTP クライアント用の Cookie 処理

2.4 で追加された仕様です。

`cookielib` モジュールは HTTP クッキーの自動処理をおこなうクラスを定義します。これは小さなデータの断片 – **クッキー** – を要求する web サイトにアクセスする際に有用です。クッキーとは web サーバの HTTP レスポンスによってクライアントのマシンに設定され、のちの HTTP リクエストをおこなうさいにサーバに返されるものです。

標準的な Netscape クッキープロトコルおよび RFC 2965 で定義されているプロトコルの両方を処理できます。RFC 2965 の処理はデフォルトではオフになっています。RFC 2109 のクッキーは Netscape クッキーとして解析され、のちに有効な 'ポリシー' に従って Netscape または RFC 2965 クッキーとして処理されます。但し、インターネット上の大多数のクッキーは Netscape クッキーです。`cookielib` はデファクトスタンダードの Netscape クッキープロトコル (これは元々 Netscape が策定した仕様とはかなり異なっています) に従うようになっており、RFC 2109 で導入された `max-age` や `port` などのクッキー属性にも注意を払います。 **注意:** Set-Cookie: や Set-Cookie2: ヘッダに現れる多種多様なパラメータの名前 (`domain` や `expires` など) は便宜上 **属性** と呼ばれますが、ここでは Python の属性と区別するため、かわりに **クッキー属性** と呼ぶことにします。

このモジュールは以下の例外を定義しています:

exception `LoadError`

この例外は `FileCookieJar` インスタンスがファイルからクッキーを読み込むのに失敗した場合に発生します。

以下のクラスが提供されています:

class `CookieJar` (*policy*=*None*)

policy は `CookiePolicy` インターフェイスを実装するオブジェクトです。

`CookieJar` クラスには HTTP クッキーを保管します。これは HTTP リクエストに応じてクッキーを取り出し、それを HTTP レスポンスの中で返します。必要に応じて、`CookieJar` インスタンスは保管されているクッキーを自動的に破棄します。このサブクラスは、クッキーをファイルやデータベースに格納したり取り出したりする操作をおこなう役割を負っています。

class `FileCookieJar` (*filename, delayload*=*None*, *policy*=*None*)

policy は `CookiePolicy` インターフェイスを実装するオブジェクトです。これ以外の引数については、該当する属性の説明を参照してください。

`FileCookieJar` はディスク上のファイルからのクッキーの読み込み、もしくは書き込みをサポートします。実際には、`load()` または `revert()` のどちらかのメソッドが呼ばれるまでクッキーは指定されたファイルからはロード**されません**。このクラスのサブクラスは 18.22.2 節で説明します。

class `CookiePolicy`()

このクラスは、あるクッキーをサーバから受け入れるべきか、そしてサーバに返すべきかを決定する役割を負っています。

class DefaultCookiePolicy(*blocked_domains=None, allowed_domains=None, netscape=True, rfc2965=False, rfc2109_as_netscape=None, hide_cookie2=False, strict_domain=False, strict_rfc2965_unverifiable=True, strict_ns_unverifiable=False, strict_ns_domain=DefaultCookiePolicy.DomainLiberal, strict_ns_set_initial_dollar=False, strict_ns_set_path=False*)

コンストラクタはキーワード引数しか取りません。*blocked_domains* はドメイン名からなるシーケンスで、ここからは決してクッキーを受けとらないし、このドメインにクッキーを返すこともありません。*allowed_domains* が None でない場合、これはこのドメインのみからクッキーを受けとり、返すという指定になります。これ以外の引数については CookiePolicy および DefaultCookiePolicy オブジェクトの説明をごらんください。

DefaultCookiePolicy は Netscape および RFC 2965 クッキーの標準的な許可 / 拒絶のルールを実装しています。デフォルトでは、RFC 2109 のクッキー (Set-Cookie: の version クッキー属性が 1 で受けとられるもの) は RFC 2965 のルールで扱われます。しかし、RFC 2965 処理が無効に設定されているか rfc2109_as_netscape が True の場合、RFC 2109 クッキーは CookieJar インスタンスによって Cookie のインスタンスの version 属性を 0 に設定する事で Netscape クッキーに「ダウングレード」されます。また DefaultCookiePolicy にはいくつかの細かいポリシー設定をおこなうパラメータが用意されています。

class Cookie()

このクラスは Netscape クッキー、RFC 2109 のクッキー、および RFC 2965 のクッキーを表現します。cookielib のユーザが自分で Cookie インスタンスを作成することは想定されていません。かわりに、必要に応じて CookieJar インスタンスの make_cookies() を呼ぶことになっています。

参考資料:

urllib2 モジュール (18.6 節):

クッキーの自動処理をおこない URL を開くモジュールです。

Cookie モジュール (18.23 節):

HTTP のクッキークラスで、基本的にはサーバサイドのコードで有用です。cookielib および Cookie モジュールは互いに依存してはいません。

http://wwwsearch.sf.net/ClientCookie/

このモジュールの拡張で、Windows 上の Microsoft Internet Explorer クッキーを読みこむクラスが含まれています。

http://www.netscape.com/newsref/std/cookie_spec.html

元祖 Netscape のクッキープロトコルの仕様です。今でもこれが主流のプロトコルですが、現在のメジャーなブラウザ (と cookielib) が実装している「Netscape クッキープロトコル」は cookie_spec.html で述べられているものとおおまかにしか似ていません。

RFC 2109, "*HTTP State Management Mechanism*"

RFC 2965 によって過去の遺物になりました。Set-Cookie: の version=1 で使います。

RFC 2965, "*HTTP State Management Mechanism*"

Netscape プロトコルのバグを修正したものです。Set-Cookie: のかわりに Set-Cookie2: を使いますが、普及してはいません。

http://kristol.org/cookie/errata.html

RFC 2965 に対する未完の正誤表です。

RFC 2964, "*Use of HTTP State Management*"

18.22.1 CookieJar および FileCookieJar オブジェクト

CookieJar オブジェクトは保管されている Cookie オブジェクトをひとつずつ取り出すための、イテレータ・プロトコルをサポートしています。

CookieJar は以下のようなメソッドを持っています:

add_cookie_header(*request*)

request に正しい Cookie: ヘッダを追加します。

ポリシーが許すようであれば (CookieJar の CookiePolicy インスタンスにある属性のうち、rfc2965 および hide_cookie2 がそれぞれ真と偽であるような場合)、必要に応じて Cookie2: ヘッダも追加されます。

request オブジェクト (通常は urllib2.Request インスタンス) は、urllib2 のドキュメントに記されているように、get_full_url(), get_host(), get_type(), unverifiable(), get_origin_req_host(), has_header(), get_header(), header_items() および add_unredirected_header() の各メソッドをサポートしている必要があります。

extract_cookies(*response, request*)

HTTP *response* からクッキーを取り出し、ポリシーによって許可されていればこれを CookieJar 内に保管します。

CookieJar は *response* 引数の中から許可されている Set-Cookie: および Set-Cookie2: ヘッダを探しだし、適切に (CookiePolicy.set_ok() メソッドの承認におうじて) クッキーを保管します。

response オブジェクト (通常は urllib2.urlopen() あるいはそれに類似する呼び出しによって得られます) は info() メソッドをサポートしている必要があります。これは getallmatchingheaders() メソッドのあるオブジェクト (通常は mimetools.Message インスタンス) を返すものです。

request オブジェクト (通常は urllib2.Request インスタンス) は urllib2 のドキュメントに記されているように、get_full_url(), get_host(), unverifiable() および get_origin_req_host() の各メソッドをサポートしている必要があります。この request はそのクッキーの保存が許可されているかを検査するとともに、クッキー属性のデフォルト値を設定するのに使われます。

set_policy(*policy*)

使用する CookiePolicy インスタンスを指定します。

make_cookies(*response, request*)

response オブジェクトから得られた Cookie オブジェクトからなるシーケンスを返します。

response および *request* 引数で要求されるインスタンスについては、extract_cookies の説明を参照してください。

set_cookie_if_ok(*cookie, request*)

ポリシーが許すのであれば、与えられた Cookie を設定します。

set_cookie(*cookie*)

与えられた Cookie を、それが設定されるべきかどうかのポリシーのチェックを行わずに設定します。

clear([*domain*[, *path*[, *name*]]])

いくつかのクッキーを消去します。

引数なしで呼ばれた場合は、すべてのクッキーを消去します。引数がひとつ与えられた場合、その *domain* に属するクッキーのみを消去します。ふたつの引数が与えられた場合、指定された *domain* と URL *path* に属するクッキーのみを消去します。引数が 3 つ与えられた場合、*domain*, *path* および *name* で指定されるクッキーが消去されます。

与えられた条件に一致するクッキーがない場合は KeyError を発生させます。

clear_session_cookies()
すべてのセッションクッキーを消去します。

保存されているクッキーのうち、discard 属性が真になっているものすべてを消去します (通常これは max-age または expires のどちらのクッキー属性もないか、あるいは明示的に discard クッキー属性が指定されているものです)。対話的なブラウザの場合、セッションの終了はふつうブラウザのウィンドウを閉じることに相当します。

注意: *ignore_discard* 引数に真を指定しないかぎり、save() メソッドはセッションクッキーは保存しません。

さらに FileCookieJar は以下のようなメソッドを実装しています:

save(*filename=None, ignore_discard=False, ignore_expires=False*)
クッキーをファイルに保存します。

この基底クラスは NotImplementedError を発生させます。サブクラスはこのメソッドを実装しないままにしておいてもかまいません。

filename はクッキーを保存するファイルの名前です。*filename* が指定されない場合、self.filename が使用されます (このデフォルト値は、それが存在する場合は、コンストラクタに渡されています)。self.filename も None の場合は ValueError が発生します。

ignore_discard: 破棄されるよう指示されていたクッキーでも保存します。*ignore_expires*: 期限の切れたクッキーでも保存します。

ここで指定されたファイルがもしすでに存在する場合は上書きされるため、以前にあったクッキーはすべて消去されます。保存したクッキーはあとで load() または revert() メソッドを使って復元することができます。

load(*filename=None, ignore_discard=False, ignore_expires=False*)
ファイルからクッキーを読み込みます。

それまでのクッキーは新しいものに上書きされない限り残ります。

ここでの引数の値は save() と同じです。

名前のついたファイルはこのクラスがわかるやり方で指定する必要があります。さもないと LoadError が発生します。さらに、例えばファイルが存在しないような時に IOError が発生する場合があります。 **注意:** (IOError を発行する)Python 2.4 との後方互換性のために、LoadError は IOError のサブクラスです。

revert(*filename=None, ignore_discard=False, ignore_expires=False*)
すべてのクッキーを破棄し、保存されているファイルから読み込み直します。

revert() は load() と同じ例外を発生させる事ができます。失敗した場合、オブジェクトの状態は変更されません。

FileCookieJar インスタンスは以下のような公開の属性をもっています:

filename
クッキーを保存するデフォルトのファイル名を指定します。この属性には代入することができます。

delayload
真であれば、クッキーを読み込むさいにディスクから遅延読み込み (lazy) します。この属性には代入することができません。この情報は単なるヒントであり、(ディスク上のクッキーが変わらない限りは) インスタンスのふるまいには影響を与えず、パフォーマンスのみに影響します。CookieJar オブジェクトはこの値を無視することもあります。標準ライブラリに含まれている FileCookieJar クラスで遅延読み込みをおこなうものはありません。

18.22.2 FileCookieJar のサブクラスと web ブラウザとの連携

クッキーの読み書きのために、以下の CookieJar サブクラスが提供されています。これ以外の CookieJar サブクラスは、Microsoft Internet Explorer ブラウザのクッキーを読みこむものも含め、http://wwwsearch.sf.net/ClientCookie/ から使用可能です。

class MozillaCookieJar (*filename, delayload=None, policy=None*)

Mozilla の cookies.txt ファイル形式 (この形式はまた Lynx と Netscape ブラウザによっても使われています) でディスクにクッキーを読み書きするための FileCookieJar です。**注意:** このクラスは RFC 2965 クッキーに関する情報を失います。また、より新しいか、標準でない port などのクッキー属性についての情報も失います。

警告: もしクッキーの損失や欠損が望ましくない場合は、クッキーを保存する前にバックアップを取っておくようにしてください (ファイルへの読み込み / 保存をくり返すと微妙な変化が生じる場合があります)。

また、 Mozilla の起動中にクッキーを保存すると、Mozilla によって内容が破壊されてしまうことにも注意してください。

class LWPCookieJar (*filename, delayload=None, policy=None*)

libwww-perl のライブラリである Set-Cookie3 ファイル形式でディスクにクッキーを読み書きするための FileCookieJar です。これはクッキーを人間に可読な形式で保存するのに向いています。

18.22.3 CookiePolicy オブジェクト

CookiePolicy インターフェイスを実装するオブジェクトは以下のようなメソッドを持っています:

set_ok (*cookie, request*)

クッキーがサーバから受け入れられるべきかどうかを表わす boolean 値を返します。

cookie は cookielib.Cookie インスタンスです。*request* は CookieJar.extract_cookies() の説明で定義されているインターフェイスを実装するオブジェクトです。

return_ok (*cookie, request*)

クッキーがサーバに返されるべきかどうかを表わす boolean 値を返します。

cookie は cookielib.Cookie インスタンスです。 *request* は CookieJar.add_cookie_header() の説明で定義されているインターフェイスを実装するオブジェクトです。

domain_return_ok (*domain, request*)

与えられたクッキーのドメインに対して、そこにクッキーを返すべきでない場合には false を返します。

このメソッドは高速化のためのものです。これにより、すべてのクッキーをある特定のドメインに対してチェックする (これには多数のファイル読みこみを伴なう場合があります) 必要がなくなります。domain_return_ok() および path_return_ok() の両方から true が返された場合、すべての決定は return_ok() に委ねられます。

もし、このクッキードメインに対して domain_return_ok() が true を返すと、つぎにそのクッキーのパス名に対して path_return_ok() が呼ばれます。そうでない場合、そのクッキードメインに対する path_return_ok() および return_ok() は決して呼ばれることはありません。path_return_ok() が true を返すと、return_ok() がその Cookie オブジェクト自身の全チェックのために呼ばれます。そうでない場合、そのクッキーパス名に対する return_ok() は決して呼ばれることはありません。

注意: domain_return_ok() は *request* ドメインだけではなく、すべての *cookie* ドメインに対して呼ばれます。たとえば request ドメインが"www.example.com" だった場合、この関数は

".example.com" および"www.example.com" の両方に対して呼ばれることがあります。同じことは path_return_ok() にもいえます。

request 引数は return_ok() で説明されているとおりです。

path_return_ok(*path, request*)
与えられたクッキーのパス名に対して、そこにクッキーを返すべきでない場合には false を返します。

domain_return_ok() の説明を参照してください。

上のメソッドの実装にくわえて、CookiePolicy インターフェイスの実装では以下の属性を設定する必要があります。これはどのプロトコルがどのように使われるべきかを示すもので、これらの属性にはすべて代入することが許されています。

netscape
Netscape プロトコルを実装していることを示します。

rfc2965
RFC 2965 プロトコルを実装していることを示します。

hide_cookie2
Cookie2: ヘッダをリクエストに含めないようにします (このヘッダが存在する場合、私たちは RFC 2965 クッキーを理解するということをサーバに示すことになります)。

もっとも有用な方法は、DefaultCookiePolicy をサブクラス化した CookiePolicy クラスを定義して、いくつか (あるいはすべて) のメソッドをオーバーライドすることでしょう。CookiePolicy 自体はどのようなクッキーも受け入れて設定を許可する「ポリシー無し」ポリシーとして使うこともできます (これが役に立つことはあまりありませんが)。

18.22.4 DefaultCookiePolicy オブジェクト

クッキーを受けつけ、またそれを返す際の標準的なルールを実装します。

RFC 2965 クッキーと Netscape クッキーの両方に対応しています。デフォルトでは、RFC 2965 の処理はオフになっています。

自分のポリシーを提供するいちばん簡単な方法は、このクラスを継承して、自分用の追加チェックの前にオーバーライドした元のメソッドを呼び出すことです:

```
import cookielib
class MyCookiePolicy(cookielib.DefaultCookiePolicy):
    def set_ok(self, cookie, request):
        if not cookielib.DefaultCookiePolicy.set_ok(self, cookie, request):
            return False
        if i_dont_want_to_store_this_cookie(cookie):
            return False
        return True
```

CookiePolicy インターフェイスを実装するのに必要な機能に加えて、このクラスではクッキーを受けとったり設定したりするドメインを許可したり拒絶したりできるようになっています。ほかにも、Netscape プロトコルのかなり緩い規則をややきつくするために、いくつかの厳密性のスイッチがついています (いくつかの良性クッキーをブロックする危険性もありますが)。

ドメインのブラックリスト機能やホワイトリスト機能も提供されています (デフォルトではオフになっています)。ブラックリストになく、(ホワイトリスト機能を使用している場合は) ホワイトリストにあるドメインのみがクッキーを設定したり返したりすることを許可されます。コンストラクタの引数 *blocked_domains*、および blocked_domains() と set_blocked_domains() メソッドを使ってください (*allowed_domains*

に関しても同様の対応する引数とメソッドがあります)。ホワイトリストを設定した場合は、それを None にすることでホワイトリスト機能をオフにすることができます。

ブラックリストあるいはホワイトリスト中にあるドメインのうち、ドット (.) で始まっていないものは、正確にそれと一致するドメインのクッキーにしか適用されません。たとえばブラックリスト中のエントリ ".example.com" は、".example.com" にはマッチしますが、".www.example.com" にはマッチしません。一方ドット (.) で始まっているドメインは、より特化されたドメインともマッチします。たとえば、".example.com" は、".www.example.com" と".www.coyote.example.com" の両方にマッチします (が、".example.com" 自身にはマッチしません)。IP アドレスは例外で、つねに正確に一致する必要があります。たとえば、かりに *blocked_domains* が ".192.168.1.2" と ".168.1.2" を含んでいたとして、192.168.1.2 はブロックされますが、193.168.1.2 はブロックされません。

DefaultCookiePolicy は以下のような追加メソッドを実装しています:

blocked_domains()
ブロックしているドメインのシーケンスを (タプルとして) 返します。

set_blocked_domains(*blocked_domains*)
ブロックするドメインを設定します。

is_blocked(*domain*)
domain がクッキーを授受しないブラックリストに載っているかどうかを返します。

allowed_domains()
None あるいは明示的に許可されているドメインを (タプルとして) 返します。

set_allowed_domains(*allowed_domains*)
許可するドメイン、あるいは None を設定します。

is_not_allowed(*domain*)
domain がクッキーを授受するホワイトリストに載っているかどうかを返します。

DefaultCookiePolicy インスタンスは以下の属性をもっています。これらはすべてコンストラクタから同じ名前の引数をつかって初期化することができ、代入してもかまいません。

rfc2109_as_netscape
True の場合、CookieJar のインスタンスに RFC 2109 クッキー (即ち Set-Cookie:ヘッダの Version cookie 属性の値が 1 のクッキー) を Netscape クッキーへ、Cookie インスタンスの version 属性を 0 に設定する事でダウングレードするように要求します。デフォルトの値は None であり、この場合 RFC 2109 クッキーは RFC 2965 処理が無効に設定されている場合に限りダウングレードされます。それ故に RFC 2109 クッキーはデフォルトではダウングレードされます。 2.5 で追加された仕様です。

一般的な厳密性のスイッチ:

strict_domain
サイトに、国別コードとトップレベルドメインだけからなるドメイン名 (.co.uk, .gov.uk, .co.nz など) を設定させないようにします。これは完璧からはほど遠い実装であり、いつもうまくいくとは限りません!

RFC 2965 プロトコルの厳密性に関するスイッチ:

strict_rfc2965_unverifiable
検証不可能なトランザクション (通常これはリダイレクトか、別のサイトがホスティングしているイメージの読み込み要求です) に関する RFC 2965 の規則に従います。この値が偽の場合、検証可能性を基準にしてクッキーがブロックされることは**決して**ありません。

Netscape プロトコルの厳密性に関するスイッチ:

strict_ns_unverifiable
検証不可能なトランザクションに関する RFC 2965 の規則を Netscape クッキーに対しても適用します。

strict_ns_domain
Netscape クッキーに対するドメインマッチングの規則をどの程度厳しくするかを指示するフラグです。とりうる値については下の説明を見てください。

strict_ns_set_initial_dollar
Set-Cookie: ヘッダで、'$' で始まる名前のクッキーを無視します。

strict_ns_set_path
要求した URI にパスがマッチしないクッキの設定を禁止します。

strict_ns_domain はいくつかのフラグの集合です。これはいくつかの値を or することで構成します (たとえば DomainStrictNoDots|DomainStrictNonDomain は両方のフラグが設定されていることになります)。

DomainStrictNoDots
クッキーを設定するさい、ホスト名のプレフィクスにドットが含まれるのを禁止します (例: www.foo.bar.com は .bar.com のクッキーを設定することはできません、なぜなら www.foo はドットを含んでいるからです)。

DomainStrictNonDomain
domain クッキー属性を明示的に指定していないクッキーは、そのクッキーを設定したドメインと同一のドメインだけに返されます (例: example.com からのクッキーに domain クッキー属性がない場合、そのクッキーが spam.example.com に返されることはありません)。

DomainRFC2965Match
クッキーを設定するさい、RFC 2965 の完全ドメインマッチングを要求します。

以下の属性は上記のフラグのうちもっともよく使われる組み合わせで、便宜をはかるために提供されています。

DomainLiberal
0 と同じです (つまり、上述の Netscape のドメイン厳密性フラグがすべてオフにされます)。

DomainStrict
DomainStrictNoDots|DomainStrictNonDomain と同じです。

18.22.5 Cookie オブジェクト

Cookie インスタンスは、さまざまなクッキーの標準で規定されている標準的なクッキー属性とおおまかに対応する Python 属性をもっています。しかしデフォルト値を決める複雑なやり方が存在しており、また max-age および expires クッキー属性は同じ値をもつことになっているので、また RFC 2109 クッキーは cookielib によって version 1 から version 0 (Netscape) クッキーへ' ダウングレード' される場合があるため、この対応は 1 対 1 ではありません。

CookiePolicy メソッド内でのごくわずかな例外を除けば、これらの属性に代入する必要はないはずです。このクラスは内部の一貫性を保つようにはしていないため、代入するのは自分のやっていることを理解している場合のみにしてください。

version
整数または None。 Netscape クッキーは バージョン 0 であり、RFC 2965 および RFC 2109 クッキーは バージョン 1 です。しかし、cookielib は RFC 2109 クッキーを Netscape クッキー (version が 0) に' ダウングレード' する場合がある事に注意して下さい。

name
クッキーの名前 (文字列)。

value

クッキーの値 (文字列)、あるいは None。

port
ポートあるいはポートの集合をあらわす文字列 (例: '80' または '80,8080')、あるいは None。

path
クッキーのパス名 (文字列、例:'/acme/rocket_launchers')。

secure
そのクッキーを返せるのが安全な接続のみならば真を返します。

expires
クッキーの期限が切れる日時をあわらす整数 (エポックから経過した秒数)、あるいは None。is_expired() も参照してください。

discard
これがセッションクッキーであれば真を返します。

comment
このクッキーの働きを説明する、サーバからのコメント文字列、あるいは None。

comment_url
このクッキーの働きを説明する、サーバからのコメントのリンク URL、あるいは None。

rfc2109
RFC 2109 クッキー (即ち Set-Cookie:ヘッダにあり、かつ Version cookie 属性の値が 1 のクッキー) の場合、True を返します。cookielib が RFC 2109 クッキーを Netscape クッキー (version が 0) に'ダウングレード' する場合があるので、この属性が提供されています。 2.5 で追加された仕様です。

port_specified
サーバがポート、あるいはポートの集合を (Set-Cookie: / Set-Cookie2: ヘッダ内で) 明示的に指定していれば真を返します。

domain_specified
サーバがドメインを明示的に指定していれば真を返します。

domain_initial_dot
サーバが明示的に指定したドメインが、ドット ('.') で始まっていれば真を返します。

クッキーは、オプションとして標準的でないクッキー属性を持つこともできます。これらは以下のメソッドでアクセスできます:

has_nonstandard_attr(*name*)
そのクッキーが指定された名前のクッキー属性をもっている場合には真を返します。

get_nonstandard_attr(*name, default=None*)
クッキーが指定された名前のクッキー属性をもっていれば、その値を返します。そうでない場合は *default* を返します。

set_nonstandard_attr(*name, value*)
指定された名前のクッキー属性を設定します。

Cookie クラスは以下のメソッドも定義しています:

is_expired([*now=None*])
サーバが指定した、クッキーの期限が切れるべき時が過ぎていれば真を返します。*now* が指定されているときは (エポックから経過した秒数です)、そのクッキーが指定された時間において期限切れになっているかどうかを判定します。

18.22.6 使用例

はじめに、もっとも一般的な cookielib の使用例をあげます:

```
import cookielib, urllib2
cj = cookielib.CookieJar()
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

以下の例では、 URL を開く際に Netscape や Mozilla または Lynx のクッキーを使う方法を示しています (クッキーファイルの位置は UNIX/Netscape の慣例にしたがうものと仮定しています):

```
import os, cookielib, urllib2
cj = cookielib.MozillaCookieJar()
cj.load(os.path.join(os.environ["HOME"], ".netscape/cookies.txt"))
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

つぎの例は DefaultCookiePolicy の使用例です。RFC 2965 クッキーをオンにし、Netscape クッキーを設定したり返したりするドメインに対してより厳密な規則を適用します。そしていくつかのドメインからクッキーを設定あるいは返還するのをブロックしています:

```
import urllib2
from cookielib import CookieJar, DefaultCookiePolicy
policy = DefaultCookiePolicy(
    rfc2965=True, strict_ns_domain=Policy.DomainStrict,
    blocked_domains=["ads.net", ".ads.net"])
cj = CookieJar(policy)
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

18.23 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 pickle 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 pickle 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 pickle 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:

cookielib モジュール (18.22 節):

Web **クライアント**向けの HTTP クッキー処理です。cookielib と Cookie は互いに独立しています。

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。

18.23.1 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'\r\n' (CRLF) となっています。 2.5 で変更された仕様: デフォルトのセパレータを '\n'　から、クッキーの使用にあわせた

output ([*attrs*[, *header*[, *sep*]]])

HTTP ヘッダ形式の文字列表現を返します。

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
```

18.23.2 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

キーの大小文字は区別されます。

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* を指定しない場合、デフォルトですべての属性を含めます。*attrs* を指定する場合，属性をリストで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。

18.23.3 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
```

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*[, *use_datetime*]]]]])

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。*use_datetime* フラグは datetime.datetime のオブジェクトとして日付/時刻を表現する時に使用し、デフォルトでは false に設定されています。datetime.datetime、datetime.date および datetime.time のオブジェクトを渡すことができます。datetime.date オブジェクトは時刻 “00:00:00” に変換されます。datetime.time オブジェクトは、今日の日付に変換されます。

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クシオブジェクトで、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 DataTime ラッパクラスまたは、datetime.da
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。`Error` をベースとする `Fault` と `ProtocolError` の両方が発生します。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 で使用すること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。

2.5 で変更された仕様: The *use_datetime* flag was added

参考資料:

XML-RPC HOWTO
(http://www.tldp.org/HOWTO/XML-RPC-HOWTO/index.html)
: 週種類のプログラミング言語で記述された XML の操作とクライアントソフトウェアの素晴らしい説明が掲載されています。XML-RPC クライアントの開発者が知っておくべきことがほとんど全て記載されています。

XML-RPC-Hacks page
(http://xmlrpc-c.sourceforge.net/hacks.php)
: イントロスペクションとマルチコールをサポートしているオープンソースの拡張ライブラリについて説明しています。

18.24.1 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。

18.24.2 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。

18.24.3 DateTime オブジェクト

このクラスは、エポックからの秒数、タプルで表現された時刻、ISO 8601 形式の時間/日付文字列、`datetime.datetime`、`datetime.date` または `datetime.time` のインスタンスの何れかで初期化することができます。

このクラスには以下のメソッドがあり、主にコードをマーシャル/アンマーシャルするための内部処理を行います。

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。

18.24.4 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。

18.24.5 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。

18.24.6 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。

18.24.7 MultiCall オブジェクト

2.4 で追加された仕様です。

遠隔のサーバに対する複数の呼び出しをひとつのリクエストにカプセル化する方法は、http://www.xmlrpc.com/discuss/msgReader%241208 で示されています。

class MultiCall (*server*)
巨大な (boxcar) メソッド呼び出しに使えるオブジェクトを作成します。*server* には最終的に呼び出しを行う対象を指定します。作成した MultiCall オブジェクトを使って呼び出しを行うと、即座に *None* を返し、呼び出したい手続き名とパラメタに保存するだけに留まります。オブジェクト自体を呼び出すと、それまでに保存しておいたすべての呼び出しを単一の `system.multicall` リクエストの形で伝送します。呼び出し結果はジェネレータになります。このジェネレータにわたってイテレーションを行うと、個々の呼び出し結果を返します。

以下にこのクラスの使い方を示します。

```
multicall = MultiCall(server_proxy)
multicall.add(2,3)
multicall.get_address("Guido")
add_result, address = multicall()
```

18.24.8 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。

dumps (*params*[, *methodname*[, *methodresponse*[, *encoding*[, *allow_none*]]]])

params を XML-RPC リクエストの形式に変換します。*methodresponse* が真の場合、XML-RPC レスポンスの形式に変換します。*params* に指定できるのは、引数からなるタプルか `Fault` 例外クラスのインスタンスです。*methodresponse* が真の場合、単一の値だけを返します。従って、*params* の長さも 1 でなければなりません。*encoding* を指定した場合、生成される XML のエンコード方式になります。デフォルトは UTF-8 です。Python の `None` は標準の XML-RPC には利用できません。`None` を使えるようにするには、*allow_none* を真にして、拡張機能つきにしてください。

loads(*data*[, *use_datetime*])

XML-RPC リクエストまたはレスポンスを（*params*, *methodname*）の形式をとる Python オブジェクトにします。*params* は引数のタプルです。*methodname* は文字列で、パケット中にメソッド名がない場合には `None` になります。例外条件を示す XML-RPC パケットの場合には、`Fault` 例外を送出します。*use_datetime* フラグは `datetime.datetime` のオブジェクトとして日付/時刻を表現する時に使用し、デフォルトでは false に設定されています。

もし、`datetime.date`、`datetime.time` のオブジェクトとともに XML-RPC を呼び出した場合は、内部で `DateTime` のオブジェクトに変換され、戻り値として `datetime.datetime` のオブジェクトのみが返されることに注意してください。

2.5 で変更された仕様: *use_datetime* フラグを追加

18.24.9 クライアントのサンプル

```
# simple test program (from the XML-RPC specification)
from xmlrpclib import ServerProxy, Error

# 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
```

XML-RPC サーバにプロキシを経由して接続する場合、カスタムトランスポートを定義する必要があります。以下に NoboNobo が作成した例を示します:

```
import xmlrpclib, httplib

class ProxiedTransport(xmlrpclib.Transport):
    def set_proxy(self, proxy):
        self.proxy = proxy
    def make_connection(self, host):
        self.realhost = host
h = httplib.HTTP(self.proxy)
return h
    def send_request(self, connection, handler, request_body):
        connection.putrequest("POST", 'http://%s%s' % (self.realhost, handler))
    def send_host(self, connection, host):
        connection.putheader('Host', self.realhost)

p = ProxiedTransport()
p.set_proxy('proxy-server:8080')
server = xmlrpclib.Server('http://time.xmlrpc.com/RPC2', transport=p)
print server.currentTime.getCurrentTime()
```

18.25 SimpleXMLRPCServer — 基本的な XML-RPC サーバー

2.2 で追加された仕様です。

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*[*allow_none*[, *encoding*]]]])
新しくサーバーインスタンスを作成します。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。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。引数 *allow_none* と *encoding* は xmlrpclib に引き継がれ、サーバーから返される XML-RPC レスポンスを制御します。 2.5 で変更された仕様: 引数 *allow_none* と *encoding* が追加されました

class CGIXMLRPCRequestHandler([*allow_none*[, *encoding*]])
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。引数 *allow_none* と *encoding* は xmlrpclib に引き継がれ、サーバーから返される XML-RPC レスポンスを制御します。 2.3 で追加された仕様です。 2.5 で変更された仕様: 引数 *allow_none* と *encoding* が追加されました

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。

18.25.1 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*[, *allow_dotted_names*])

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()` の返り値が結果としてクライアントに返されます。その API は `def _dispatch(self, method, params)` (注意: *params* は可変引数リストではありません) です。仕事をするために下位の関数を呼ぶ時には、その関数は `func(*params)` のように呼ばれます。`_dispatch()` の返り値はクライアントへ結果として返されます。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。

もしオプション引数 *allow_dotted_names* が真 (true) で、インスタンスがメソッド`_dispatch()` を定義していないとき、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、その返り値をクライアントに返します。

> **警告:** *allow_dotted_names* オプションを有効にすると、侵入者にあなたのモジュールのグローバル変数にアクセスすることを許し、あなたのコンピュータで任意のコードを実行することを許すことがあります。このオプションは安全な閉じたネットワークでのみお使い下さい。

2.3.5, 2.4.1 で変更された仕様: *allow_dotted_names* はセキュリティホールを塞ぐために追加されました。以前のバージョンは安全ではありません

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。

rpc_paths

この属性値は XML-RPC リクエストを受け付ける URL の正当なパス部分をリストするタプルでなければなりません。これ以外のパスへのリクエストは 404「そのようなページはありません」HTTP エラーになります。このタプルが空の場合は全てのパスが正当であると見なされます。デフォルト値は `('/', '/RPC2')` です。 2.5 で追加された仕様です。

以下に例を示します。

```
from SimpleXMLRPCServer import SimpleXMLRPCServer

# Create server
server = SimpleXMLRPCServer(("localhost", 8000))
server.register_introspection_functions()

# Register pow() function; this will use the value of
# pow.__name__ as the name, which is just 'pow'.
server.register_function(pow)

# Register a function under a different name
def adder_function(x,y):
    return x + y
server.register_function(adder_function, 'add')

# Register an instance; all the methods of the instance are
# published as XML-RPC methods (in this case, just 'div').
class MyFuncs:
    def div(self, x, y):
        return x // y

server.register_instance(MyFuncs())

# Run the server's main loop
server.serve_forever()
```

以下のクライアントコードは上のサーバーで使えるようになったメソッドを呼び出します:

```
import xmlrpclib

s = xmlrpclib.Server('http://localhost:8000')
print s.pow(2,3)  # Returns 2**3 = 8
print s.add(2,3)  # Returns 5
print s.div(5,2)  # Returns 5//2 = 2

# Print list of available methods
print s.system.listMethods()
```

18.25.2 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。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()` の返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、その返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 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()
```

18.26 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 XMR-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。

18.26.1 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。

18.26.2 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-RCP 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。

十九章

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。

audioop	生の音声データを操作する
imageop	生の画像データを操作する。
aifc	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
sunau	Sun AU サウンドフォーマットへのインターフェース
wave	WAV サウンドフォーマットへのインターフェイス
chunk	IFF チャンクデータの読み込み。
colorsys	RGB 他の色体系間の変換。
rgbimg	“SGI RGB” 形式の画像ファイルを読み書きします (とはいえ、このモジュールは SGI 特有のものでは
imghdr	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。
sndhdr	サウンドファイルの識別
ossaudiodev	OSS 互換オーディオデバイスへのアクセス。

19.1 audioop — 生の音声データを操作する

audioop モジュールは音声データを操作する関数を収録しています。このモジュールは、Python 文字列型中に入っている 8, 16, 32 ビットの符号付き整数でできた音声データ、すなわち al および sunaudiodev で使われているのと同じ形式の音声データを操作します。特に指定の無いかぎり、スカラ量を表す要素はすべて整数型になっています。

このモジュールは a-LAW、u-LAW そして 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サンプル幅が 16 ビットのデータに対してのみ働きますが、それ以外は常にサンプル幅を操作のパラメタとして (バイト単位で) 渡します。

このモジュールでは以下の変数と関数を定義しています：

exception error
この例外は、未知のサンプル当たりのバイト数を指定した時など、全般的なエラーに対して送出されます。

add (*fragment1, fragment2, width*)
パラメタに渡した 2 つのデータを加算した結果を返します。*width* はサンプル幅をバイトで表したもので、1、2、4 のうちいずれかです。2 つのデータは同じサンプル幅でなければなりません。

adpcm2lin (*adpcmfragment, width, state*)
Intel/DVI ADPCM 形式のデータをリニア (linear) 形式にデコードします。ADPCM 符号化方式の詳細については lin2adpcm() の説明を参照して下さい。(*sample*, *newstate*) からなるタプルを返し、

サンプルは *width* に指定した幅になります。

alaw2lin (*fragment, width*)
a-LAW 形式のデータをリニア (linear) 形式に変換します。a-LAW 形式は常に 8 ビットのサンプルを使用するので、ここでは *width* は単に出力データのサンプル幅となります。2.5 で追加された仕様です。

avg (*fragment, width*)
データ中の全サンプルの平均値を返します。

avgpp (*fragment, width*)
データ中の全サンプルの平均 peak-peak 振幅を返します。フィルタリングを行っていない場合、このルーチンの有用性は疑問です。

bias (*fragment, width, bias*)
元データの各サンプルにバイアスを加えたデータを返します。

cross (*fragment, width*)
引数に渡したデータ中のゼロ交差回数を返します。

findfactor (*fragment, reference*)
`rms(add(`*fragment*`, mul(`*reference*`, −`*F*`)))` を最小にするような係数 *F* 、すなわち、*reference* に乗算したときにもっとも *fragment* に近くなるような値を返します。*fragment* と *reference* のサンプル幅はいずれも 2 バイト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 を可能な限り *fragment* に一致させようとします (*fragment* は *reference* より長くなければなりません)。この処理は (概念的には) *fragment* からスライスをいくつか取り出し、それぞれについて `findfactor()` を使って最良な一致を計算し、誤差が最小の結果を選ぶことで実現します。*fragment* と *reference* のサンプル幅は両方とも 2 バイトでなければなりません。(*offset*, *factor*) からなるタプルを返します。*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()` の返す係数 (浮動小数点数) です。

findmax (*fragment, length*)
fragment から、長さが *length* サンプル (バイトではありません!) で最大のエネルギーを持つスライス、すなわち、`rms(fragment[i*2:(i+length)*2])` を最大にするようなスライスを探し、 *i* を返します。データのはサンプル幅は 2 バイト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*)
データ中の *index* サンプル目の値を返します。

lin2adpcm (*fragment, width, state*)
データを 4 ビットの Intel/DVI ADPCM 符号化方式に変換します。ADPCM 符号化方式とは適応符号化方式の一つで、あるサンプルと (可変の) ステップだけ離れたその次のサンプルとの差を 4 ビットの整数で表現する方式です。 Intel/DVI ADPCM アルゴリズムは IMA (国際 MIDI 協会) に採用されているので、おそらく標準になるはずです。

state はエンコーダの内部状態が入ったタプルです。エンコーダは (*adpcmfrag*, *newstate*) のタプルを返し、次に `lin2adpcm()` を呼び出す時に *newstate* を渡さねばなりません。最初に呼び出す時には *state* に `None` を渡してもかまいません。*adpcmfrag* は ADPCM で符号化されたデータで、バイト当たり 2 つの 4 ビット値がパックされています。

lin2alaw (*fragment, width*)
音声データの各サンプルを a-LAW 符号でエンコードし、Python 文字列として返します。a-LAW とは音声符号化方式の一つで、約 13 ビットに相当するダイナミックレンジをわずか 8 ビットで実現で

きます。 Sun やその他の音声ハードウェアで使われています。 2.5 で追加された仕様です。

lin2lin (*fragment, width, newwidth*)
サンプル幅を 1、2、4 バイト形式の間で変換します。

lin2ulaw (*fragment, width*)
音声データの各サンプルを u-LAW 符号でエンコードし、Python 文字列として返します。 u-LAW とは音声符号化方式の一つで、約 14 ビットに相当するダイナミックレンジをわずか 8 ビットで実現できます。 Sun やその他の音声ハードウェアで使われています。

minmax (*fragment, width*)
音声データ全サンプル中における最小値と最大値からなるタプルを返します。

max (*fragment, width*)
音声データ全サンプルの**絶対値** の最大値を返します。

maxpp (*fragment, width*)
音声データの最大 peak-peak 振幅を返します。

mul (*fragment, width, factor*)
元のデータ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。オーバフローが起きても例外を送出せず無視します。

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 は変換ルーチンの内部状態を入れたタプルです。変換ルーチンは (*newfragment*, *newstate*) を返し、次に `ratecv()` を呼び出す時には *newstate* を渡さなねばなりません。最初の呼び出しでは `None` を渡します。

引数 *weightA* と *weightB* は単純なデジタルフィルタのパラメタで、デフォルト値はそれぞれ `1` と `0` です。

reverse (*fragment, width*)
データ内のサンプルの順序を逆転し、変更されたデータを返します。

rms (*fragment, width*)
データの自乗平均根 (root-mean-square) 、すなわち

$$\sqrt{\frac{\sum {S_i}^2}{n}}$$

を返します。これはオーディオ信号の強度 (power) を測る一つの目安です。

tomono (*fragment, width, lfactor, rfactor*)
ステレオ音声データをモノラル音声データに変換します。左チャネルのデータに *lfactor*、右チャネルのデータに *rfactor* を掛けた後、二つのチャネルの値を加算して単一チャネルの信号を生成します。

tostereo (*fragment, width, lfactor, rfactor*)
モノラル音声データをステレオ音声データに変換します。ステレオ音声データの各サンプル対は、モノラル音声データの各サンプルをそれぞれ左チャネルは *lfactor* 倍、右チャネルは *rfactor* 倍して生成します。

ulaw2lin (*fragment, width*)
u-LAW で符号化されている音声データを線形に符号化された音声データに変換します。 u-LAW 符号化は常にサンプル当たり 8 ビットを使うため、*width* は出力音声データのサンプル幅にしか使われません。

`mul()` や `max()` といった操作はモノラルとステレオを区別しない、すなわち全てのデータを平等に扱うということに注意してください。この仕様が問題になるようなら、あらかじめステレオ音声データを二つ

のモノラル音声データに分割しておき、操作後に再度統合してください。そのような例を以下に示します:

```
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)
```

ADPCM エンコーダを使って音声データの入ったネットワークパケットを構築する際、自分のプロトコルを (パケットロスに耐えられるように) ステートレス (stateless) にしたいなら、データだけでなく状態変数 (state) も伝送せねばなりません。このとき、伝送するのはエンコード後状態 (エンコーダの返す値) ではなく、エンコーダの初期状態 (`lin2adpcm()` に渡した値) *initial* なので注意してください。`struct.struct()` を使って状態変数をバイナリ形式で保存したいなら、最初の要素 (予測値) は 16 ビットで、次の値 (デルタ係数: delta index) は 8 ビットで符号化できます。

このモジュールの ADPCM 符号のテストは自分自身に対してのみ行っており、他の ADPCM 符号との間では行っていません。作者が仕様を誤解している部分もあるかもしれず、それぞれの標準との間で相互運用できない場合もあり得ます。

`find*()` ルーチンは一見滑稽に見えるかもしれません。これらの関数の主な目的はエコー除去 (echo cancellation) にあります。エコー除去を十分高速に行うには、出力サンプル中から最も大きなエネルギーを持った部分を取り出し、この部分が入力サンプル中のどこにあるかを調べ、入力サンプルから出力サンプル自体を減算します:

```
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)
```

19.2 `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.lrectwrite()` 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.lrectread()` 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 *x1* 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。補間は行われません。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。

backward_compatible
0 にセットすると、このモジュールの関数は、リトルエンディアンのシステムで以前のバージョンと互換性のない方法でマルチバイトピクセル値を表現するようになります。このモジュールはもともと SGI 向けに書かれたのですが、SGI はビッグエンディアンのシステムであり、この変数を設定しても何の影響もありません。とはいえ、このコードはもともとどこでも動作するように考えて作られたわけではないので、バイトオーダに関する仮定が相互利用向けではありませんでした。この変数を 0 にすると、リトルエンディアンのシステムではバイトオーダを反転して、ビッグエンディアンと同じにします。

19.3 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 `cl` 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、読み込み用に開くときには`'r'` か`'rb'` のどちらかで、書き込み用に開くときには`'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の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。

19.4 `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。

19.4.1 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、何も面白いことはしません。

getmarkers()
: `None` を返します。

getmark(*id*)
: エラーを発生します。

19.4.2 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。

19.5 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。

19.5.1 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、何も面白いことはしません。

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。

19.5.2 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。

19.6 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。[1] 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。

[1] "EA IFF 85" Standard for Interchange Format Files, Jerry Morrison, Electronic Arts, January 1985.

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。

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。

19.7 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得ます。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.poynton.com/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)
```

19.8 rgbimg — “SGI RGB” ファイルを読み書きする

リリース 2.5 以降で撤廃された仕様です。 このモジュールはメンテナンスされておらず、使われてもいないようです。

rgbimg モジュールを使うと、Python プログラムから SGI imglib 画像ファイル (‘.rgb’ としても知られています) にアクセスできます。このモジュールは完全とはいえませんが、ちょっとした用途には十分な機能を持っているため提供されています。現在のところカラーマップファイルは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上でしか構築されません。他のシステムでは適切に動作しそうにないからです。

このモジュールでは以下の変数と関数を定義しています:

exception error
ファイル形式がサポートされていない場合など、全てのエラーに対して送出される例外です。

sizeofimage (*file*)
タプル (*x*, *y*) を返します。*x* と *y* は画像の大きさをピクセル単位で表した値です。現状では、 4 バイト RGBA ピクセル、3 バイト RGB ピクセル、および 1 バイトグレイスケールピクセル だけをサポートしています。

longimagedata (*file*)
指定したファイル上の画像を読み込んでデコードし、Python 文字列にして返します。文字列は 4 バイト RGB ピクセル形式です。左下のピクセルが文字列の先頭になります。この形式は、例えば gl.lrectwrite() に渡すといった用途に適しています。

longstoimage (*data, x, y, z, file*)
data の RGBA データを画像ファイル *file* に書き込みます。*x* と *y* は画像の大きさを表します。画像を 1 バイトの *z* はグレイスケールで保存する場合には 1 、3 バイトの RGB データの場合は 3 です、4 バイトの RGBA データの場合には 4 になります。入力データは常にピクセル当たり 4 バイトにせねばなりません。gl.lrectread() の返す形式と同じです。

ttob (*flag*)
画像のスキャンラインを下端から上端に向かって読み書きする (*flag* はゼロ、SGI GL 互換の方法) か、上端から下端に向かって読み書きする (*flag* は 1、 X 互換の方法) かを決めるグローバルなフラグです。デフォルト値はゼロです。

19.9 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 or Exif formats
'bmp'	BMP files
'png'	Portable Network Graphics

2.5 で追加された仕様: Exif の検出

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'
```

19.10 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。

19.11 ossaudiodev — OSS 互換オーディオデバイスへのアクセス

2.3 で追加された仕様です。

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 で広く利用でき、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です。

参考資料:

Open Sound System Programmer's Guide
(http://www.opensound.com/pguide/oss.pdf)

OSS C API の公式ドキュメント

このモジュールでは OSS デバイスドライバーが提供してい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 や FreeBSD の '<sys/soundcard.h>' を参照してください。

ossaudiodev では以下の変数と関数を定義しています:

exception error

何らかのエラーのときに送出される例外です。引数は何が誤っているかを示す文字列です。

(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場合には IOError を送出します。ossaudiodev が直接エラーを検出した場合には OSSAudioError になります。)

(以前のバージョンとの互換性のため、この例外クラスは ossaudiodev.error としても利用できます。)

open ([*device,*]*mode*)

オーディオデバイスを開き、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() といったファイル類似オブジェクトのメソッドを数多くサポートしています。(とはいえ、伝統的な UNIX の read/write における意味づけと OSS デバイスの read/write との間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。

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。見つからなければ '/dev/dsp' を参照します。

mode は読み出し専用アクセスの場合には 'r'、書き込み専用 (プレイバック) アクセスの場合には 'w'、読み書きアクセスの場合には 'rw' にします．多くのサウンドカードは一つのプロセスが一度にレコーダとプレーヤのどちらかしか開けないようにしているため，必要な操作に応じたデバイスだけを開くようにするのがよいでしょう。また，サウンドカードには半二重 (half-duplex) 方式のものが

あります: こうしたカードでは，デバイスを読み出しまたは書き込み用に開くことはできますが，両方同時には開けません.

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**引数は省略可能で、2 番目が必須です。これは ossaudiodev にとってかわられた古い linuxaudiodev との互換性のためという歴史的な産物です。

openmixer([*device*])
ミキサデバイスを開き、OSS ミキサデバイスオブジェクトを返します。*device* は使用するミキサデバイスのファイル名です。*device* を指定しない場合、モジュールはまず環境変数 AUDIODEV を参照して使用するデバイスを探します。見つからなければ、'/dev/mixer' を参照します。

19.11.1 オーディオデバイスオブジェクト

オーディオデバイスに読み書きできるようになるには、まず 3 つのメソッドを正しい順序で呼び出さねばなりません:

1. setfmt() で出力形式を設定し、
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。

この代わりに setparameters() メソッドを呼び出せば、三つのオーディオパラメタを一度で設定できます。setparameters() は便利ですが、多くの状況で柔軟性に欠けるでしょう。

open() の返すオーディオデバイスオブジェクトには以下のメソッドおよび (読み出し専用の) 属性があります:

close()
オーディオデバイスを明示的に閉じます。オーディオデバイスは、読み出しや書き込みが終了したら必ず閉じねばなりません。閉じたオブジェクトを再度開くことはできません。

fileno()
デバイスに関連付けられているファイル記述子を返します。

read(*size*)
オーディオ入力から *size* バイトを読みだし、 Python 文字列型にして返します。多くの UNIX デバイスドライバと違い、ブロックデバイスモード (デフォルト) の OSS オーディオデバイスでは、要求した量のデータ全体を取り込むまで read() がブロックします。

write(*data*)
Python 文字列 *data* の内容をオーディオデバイスに書き込み、書き込まれたバイト数を返します。オーディオデバイスがブロックモード (デフォルト) の場合、常に文字列データ全体を書き込みます (前述のように、これは通常の UNIX デバイスの振舞いとは異なります)。デバイスが非ブロックモードの場合、データの一部が書き込まれないことがあります — 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。オーディオデバイスがデータを受け取れるようになるまで待機し、書き込めるだけのデータを書き込むという操作を、*data* を全て書き込み終わるまで繰り返します。デバイスがブロックモード (デフォルト) の場合には、このメソッドは write() と同じです。writeall() が有用なのは非ブロックモードだけです。実際に書き込まれたデータの量と渡したデータの量は必ず同じになるので、戻り値はありません。

以下のメソッドの各々は ioctl() システムコール一つ一つに対応しています。対応関係ははっきりしています: 例えば、setfmt() は SNDCTL_DSP_SETFMT ioctl に対応していますし、sync() は SNDCTL_

DSP_SYNC に対応しています (このシンボル名は OSS のドキュメントを参照する時に助けになるでしょう)。根底にある ioctl() が失敗した場合、これらの関数は全て IOError を送出します。

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、ブロックモードは戻せません。

getfmts()
: サウンドカードがサポートしているオーディオ出力形式をビットマスクで返します。以下は OSS でサポートされているフォーマットの一部です。

フォーマット	説明
AFMT_MU_LAW	対数符号化 (Sun の .au 形式や '/dev/audio' で使われている形式)
AFMT_A_LAW	対数符号化
AFMT_IMA_ADPCM	Interactive Multimedia Association で定義されている 4:1 圧縮形式
AFMT_U8	符号なし 8 ビットオーディオ
AFMT_S16_LE	符号つき 16 ビットオーディオ、リトルエンディアンバイトオーダ (Intel プロセッサで使われ
AFMT_S16_BE	符号つき 16 ビットオーディオ、ビッグエンディアンバイトオーダ (68k、PowerPC、Sparc で
AFMT_S8	符号つき 8 ビットオーディオ
AFMT_U16_LE	符号なし 16 ビットリトルエンディアンオーディオ
AFMT_U16_BE	符号なし 16 ビットビッグエンディアンオーディオ

: オーディオ形式の完全なリストは OSS の文書をひもといてください。ただ、ほとんどのシステムは、こうした形式のサブセットしかサポートしていません。古めのデバイスの中には AFMT_U8 だけしかサポートしていないものがあります。現在使われている最も一般的な形式は AFMT_S16_LE です。

setfmt (*format*)
: 現在のオーディオ形式を *format* に設定しようと試みます — *format* については getfmts() のリストを参照してください。実際にデバイスに設定されたオーディオ形式を返します。要求通りの形式でないこともあります。AFMT_QUERY を渡すと現在デバイスに設定されているオーディオ形式を返します。

channels (*num_channels*)
: 出力チャネル数を *num_channels* に設定します。1 はモノラル、2 はステレオです。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* に設定しようと試み、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:

レート	説明
8000	/dev/audio のデフォルト
11025	会話音声の録音に使われるレート
22050	
44100	(サンプルあたり 16 ビットで 2 チャネルの場合) CD 品質のオーディオ
96000	(サンプル当たり 24 ビットの場合) DVD 品質のオーディオ

sync()
: サウンドデバイスがバッファ内の全てのデータを再生し終えるまで待機します。(デバイスを閉じると暗黙のうちに sync() が起こります) OSS のドキュメント上では、sync() を使うよりデバイスを一度閉じて開き直すよう勧めています。

reset()
再生あるいは録音を即座に中止して、デバイスをコマンドを受け取れる状態に戻します。OSS のドキュメントでは、reset() を呼び出した後に一度デバイスを閉じ、開き直すよう勧めています。

post()
ドライバに出力の一時停止 (pause) が起きそうであることを伝え、ドライバが一時停止をより賢く扱えるようにします。短いサウンドエフェクトを再生した直後やユーザ入力待ちの前、またディスク I/O 前などに使うことになるでしょう。

以下のメソッドは、複数の ioctl を組み合わせたり、ioctl と単純な計算を組み合わせたりした便宜用メソッドです。

setparameters (*format, nchannels, samplerate,* [*, strict=False*])
主要なオーディオパラメタ、サンプル形式、チャネル数、サンプルレートを一つのメソッド呼び出しで設定します。*format*、*nchannels* および *samplerate* には、それぞれ setfmt()、channels() および speed() と同じやり方で値を設定します。*strict* の値が真の場合、setparameters() は値が実際に要求通りにデバイスに設定されたかどうか調べ、違っていれば OSSAudioError を送出します。実際にデバイスドライバが設定したパラメタ値を表す (*format*, *nchannels*, *samplerate*) からなるタプルを返します (setfmt()、channels() および speed() の返す値と同じです)。

以下に例を示します:

```
(fmt, channels, rate) = dsp.setparameters(fmt, channels, rate)
```

is equivalent to

```
fmt = dsp.setfmt(fmt)
channels = dsp.channels(channels)
rate = dsp.rate(channels)
```

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上に残っていてまだ再生されていないサンプル数を返します。

obuffree()
ブロックを起こさずにハードウェアの再生キューに書き込めるサンプル数を返します。

オーディオデバイスオブジェクトは読み出し専用の属性もサポートしています:

closed
デバイスが閉じられたかどうかを示す真偽値です。

name
デバイスファイルの名前を含む文字列です。

mode
ファイルの I/O モードで、"r", "rw", "w" のどれかです。

19.11.2 ミキサデバイスオブジェクト

ミキサオブジェクトには、2 つのファイル類似メソッドがあります:

close()
すでに開かれているミキサ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を使おうとすると、IOError を送出します。

fileno()
開かれているミキサデバイスファイルのファイルハンドルナンバを返します。

以下はオーディオミキシング固有のメソッドです。

controls()
このメソッドは、利用可能なミキサコントロール (SOUND_MIXER_PCM や SOUND_MIXER_SYNTH のように、ミキシングを行えるチャネル) を指定するビットマスクを返します。このビットマスクは利用可能な全てのミキサコントロールのサブセットです — 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、もし現在のミキサオブジェクトが PCM ミキサ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:

```
if mixer.controls() & (1 << ossaudiodev.SOUND_MIXER_PCM):
    # PCM is supported
    ... code ...
```

ほとんどの用途には、SOUND_MIXER_VOLUME (マスタボリューム) と SOUND_MIXER_PCM コントロールがあれば十分でしょう — とはいえ、ミキサを使うコードを書くときには、コントロールを選ぶ時に柔軟性を持たせるべきです。例えば Gravis Ultrasound には SOUND_MIXER_VOLUME がありません。

stereocontrols()
ステレオミキサコントロールを示すビットマスクを返します。ビットが立っているコントロールはステレオであることを示し、立っていないコントロールはモノラルか、ミキサがサポートしていないコントロールである (どちらの理由かは controls() と組み合わせて使うことで判別できます) ことを示します。

ビットマスクから情報を得る例は関数 controls() のコード例を参照してください。

reccontrols()
録音に使用できるミキサコントロールを特定するビットマスクを返します。ビットマスクから情報を得る例は関数 controls() のコード例を参照してください。

get(*control*)
指定したミキサコントロールのボリュームを返します。2 要素のタプル (left_volume,right_volume) を返します。ボリュームの値は 0 (無音) から 100 (最大) で示されます。コントロールがモノラルでも 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。

不正なコントロールを指定した場合は OSSAudioError を送出します。また、サポートされていないコントロールを指定した場合には IOError を送出します。

set(*control, (left, right)*)
指定したミキサコントロールのボリュームを (left,right) に設定します。left と right は整数で、0 (無音) から 100 (最大) の間で指定せねばなりません。呼び出しに成功すると新しいボリューム値を 2 要素のタプルで返します。サウンドカードによっては、ミキサの分解能上の制限から、指定したボリュームと厳密に同じにはならない場合があります。

不正なコントロールを指定した場合や、指定したボリューム値が範囲外であった場合、IOError を送出します。

get_recsrc()
現在録音のソースに使われているコントロー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にはこの関数を使ってください。呼び出しに成功すると、新たな録音の (場合によっては複数の) ソースを示すビットマスクを返します; 不正なソースを指定すると IOError を送出します。現在の録音のソースとしてマイク入力を設定するには、以下のようにします:

```
mixer.setrecsrc (1 << ossaudiodev.SOUND_MIXER_MIC)
```

二十章

Tkを用いたグラフィカルユーザインターフェイス

Tk/Tcl は長きにわたり Python の不可欠な一部でありつづけています。Tk/Tcl は頑健でプラットホームに依存しないウィンドウ構築ツールキットであり、 Python プログラマは `Tkinter` モジュールやその拡張の `Tix` モジュールを使って利用できます。

`Tkinter` モジュールは、 Tcl/Tk 上に作られた軽量なオブジェクト指向のレイヤです。 `Tkinter` を使うために Tcl コードを書く必要はありませんが、 Tk のドキュメントや、場合によっては Tcl のドキュメントを調べる必要があるでしょう。`Tkinter` は Tk のウィジェットを Python のクラスとして実装しているラッパをまとめたものです。加えて、内部モジュール `_tkinter` では、 Python と Tcl がやり取りできるようなスレッド安全なメカニズムを提供しています。

Tk は Python にとって唯一の GUI というわけではありません。Python 用の他の GUI ツールキットに関する詳しい情報は、20.6 章、「他のユーザインタフェースモジュールとパッケージ」を参照してください。

`Tkinter`	グラフィカルユーザインタフェースを実現する Tcl/Tk へのインタフェース
`Tix`	Tkinter 用の Tk 拡張ウィジェット
`ScrolledText`	垂直スクロールバーを持つテキストウィジェット。
`turtle`	タートルグラフィックスのための環境。

20.1 `Tkinter` — Tcl/Tk への Python インタフェース

`Tkinter` モジュール (“Tk インタフェース”) は、Tk GUI ツールキットに対する標準の Python インタフェースです。Tk と `Tkinter` はほとんどの UNIX プラットフォームの他、 Windows や Macintosh システム上でも利用できます。(Tk 自体は Python の一部ではありません。 Tk は ActiveState で保守されています。)

参考資料:

Python Tkinter Resources
(`http://www.python.org/topics/tkinter/`)
Python Tkinter Topic Guide では、Tk を Python から利用する上での情報と、その他の Tk にまつわる情報源を数多く提供しています。

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です。

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です。

Tkinter for JPython
(`http://jtkinter.sourceforge.net`)

Jython から Tkinter へのインタフェースです。

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) です。

20.1.1 Tkinter モジュール

ほとんどの場合、本当に必要となるのは `Tkinter` モジュールだけですが、他にもいくつかの追加モジュールを利用できます。Tk インタフェース自体は `_tkinter` と言う名前のバイナリモジュール内にあります。このモジュールに入っているのは Tk への低水準のインタフェースであり、プリケーションプログラマが直接使ってはなりません。`_tkinter` は通常共有ライブラリ (や DLL) になっていますが、Python インタプリタに静的にリンクされていることもあります。

Tk インタフェースモジュールの他にも、`Tkinter` には Python モジュールが数多く入っています。最も重要なモジュールは、`Tkinter` 自体と `Tkconstants` と呼ばれるモジュールの二つです。前者は自動的に後者を import するので、以下のように一方のモジュールを import するだけで Tkinter を使えるようになります:

```
import Tkinter
```

あるいは、よく使うやり方で:

```
from Tkinter import *
```

のようにします。

class Tk (*screenName=None, baseName=None, className='Tk', useTk=1*)
`Tk` クラスは引数なしでインスタンス化します。これは Tk のトップレベルウィジェットを生成します。通常、トップレベルウィジェットはアプリケーションのメインウィンドウになります。それぞれのインスタンスごとに固有の Tcl インタプリタが関連づけられます。 2.4 で変更された仕様: *useTk* パラメタが追加されました

Tcl (*screenName=None, baseName=None, className='Tk', useTk=0*)
`Tcl` はファクトリ関数で、`Tk` クラスで生成するオブジェクトとよく似たオブジェクトを生成します。ただし Tk サブシステムを初期化しません。この関数は、余分なトップレベルウィンドウを作る必要がなかったり、(X サーバを持たない UNIX/Linux システムなどのように) 作成できない環境において Tcl インタプリタを駆動したい場合に便利です。`Tcl` で生成したオブジェクトに対して `loadtk` メソッドを呼び出せば、トップレベルウィンドウを作成 (して、Tk サブシステムを初期化) します。 2.4 で追加された仕様です。

Tk をサポートしているモジュールには、他にも以下のようなモジュールがあります:

ScrolledText 垂直スクロールバー付きのテキストウィジェットです。

tkColorChooser ユーザに色を選択させるためのダイアログです。

tkCommonDialog このリストの他のモジュールが定義しているダイアログの基底クラスです。

tkFileDialog ユーザが開きたいファイルや保存したいファイルを指定できるようにする共通のダイアログです。

tkFont フォントの扱いを補助するためのユーティリティです。

tkMessageBox 標準的な Tk のダイアログボックスにアクセスします。

tkSimpleDialog 基本的なダイアログと便宜関数 (convenience function) です。

Tkdnd `Tkinter` 用のドラッグアンドドロップのサポートです。実験的なサポートで、Tk DND に置き替わった時点で撤廃されるはずです。

turtle Tk ウィンドウ上でタートルグラフィックスを実現します。

20.1.2 Tkinter お助け手帳 (life preserver)

この節は、 Tk や Tkinter を全て網羅したチュートリアルを目指しているわけではありません。むしろ、Tkinter のシステムを学ぶ上での指針を示すための、その場しのぎ的なマニュアルです。

謝辞:

- Tkinter は Steen Lumholt と Guido van Rossum が作成しました。
- Tk は John Ousterhout が Berkeley の在籍中に作成しました。
- この Life Preserver は Virginia 大学の Matt Conway 他が書きました。
- html へのレンダリングやたくさんの編集は、Ken Manheimer が FrameMaker 版から行いました。
- Fredrik Lundh はクラスインタフェース詳細な説明を書いたり内容を改訂したりして、現行の Tk 4.2 に合うようにしました。
- Mike Clarkson はドキュメントを LaTeX 形式に変換し、リファレンスマニュアルのユーザインタフェー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で構成されています: 前半では、背景となることがらを (大雑把に) 網羅しています。後半は、キーボードの横に置けるような手軽なリファレンスになっています。

「ホゲホゲ (blah) するにはどうしたらよいですか」 という形の問いに答えようと思うなら、まず Tk で「ホゲホゲ」する方法を調べてから、このドキュメントに戻ってきてその方法に対応する `Tkinter` の関数呼び出しに変換するのが多くの場合最善の方法になります。Python プログラマが Tk ドキュメンテーションを見れば、たいてい正しい Python コマンドの見当をつけられます。従って、Tkinter を使うには Tk についてほんの少しだけ知っていればよいということになります。このドキュメントではその役割を果たせないので、次善の策として、すでにある最良のドキュメントについていくつかヒントを示しておくことにしましょう:

- Tk の man マニュアルのコピーを手に入れるよう強く勧めます。とりわけ最も役立つのは 'mann' ディレクトリ内にあるマニュアルです。`man3` のマニュアルページは Tk ライブラリに対する C インタフェースについての説明なので、スクリプト書きにとって取り立てて役に立つ内容ではありません。
- Addison-Wesley は John Ousterhout の書いた *Tcl and the Tk Toolkit* (ISBN 0-201-63337-X) という名前の本を出版しています。この本は初心者向けの Tcl と Tk の良い入門書です。内容は網羅的ではなく、詳細の多くは man マニュアル任せにしています。
- たいていの場合、'Tkinter.py' は参照先としては最後の地 (last resort) ですが、それ以外の手段で調べても分からない場合には救いの地 (good place) になるかもしれません。

参考資料:

ActiveState Tcl **ホームページ**
(http://tcl.activestate.com/)
Tk/Tcl の開発は ActiveState で大々的に行われています。

Tcl and the Tk Toolkit
(http://www.amazon.com/exec/obidos/ASIN/020163337X)
Tcl を考案した John Ousterhout による本です。

Practical Programming in Tcl and Tk
(http://www.amazon.com/exec/obidos/ASIN/0130220280)
Brent Welch の百科事典のような本です。

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()

root = Tk()
app = Application(master=root)
app.mainloop()
root.destroy()
```

20.1.3 Tcl/Tk を (本当に少しだけ) 見渡してみる

クラス階層は複雑に見えますが、実際にプログラムを書く際には、アプリケーションプログラマはほとんど常にクラス階層の最底辺にあるクラスしか参照しません。

注意:

- クラスのいくつかは、特定の関数を一つの名前空間下にまとめるために提供されています。こうしたクラスは個別にインスタンス化するためのものではありません。
- `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するようになっています。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ると常にシステムが作成します。

- `Widget` クラスもまた、インスタンス化して使うようにはなっていません。このクラスはサブクラス化して「実際の」ウィジェットを作成するためのものです。(C++ で言うところの、'抽象クラス (abstract class)' です)。

このリファレンス資料を活用するには、Tk の短いプログラムを読んだり、Tk コマンドの様々な側面を知っておく必要がままあるでしょう。(下の説明の `Tkinter` 版は、20.1.4 節を参照してください。)

Tk スクリプトは Tcl プログラムです。全ての Tcl プログラムに同じく、Tk スクリプトはトークンをスペースで区切って並べます。Tk ウィジェットとは、ウィジェットの**クラス**、ウィジェットの設定を行う**オプション**、そしてウィジェットに役立つことをさせる**アクション** をあわせたものに過ぎません。

Tk でウィジェットを作るには、常に次のような形式のコマンドを使います:

```
classCommand newPathname options
```

classCommand どの種類のウィジェット (ボタン、ラベル、メニュー、...) を作るかを表します。

newPathname 作成するウィジェットにつける新たな名前です。Tk 内の全ての名前は一意になっていなければなりません。一意性を持たせる助けとして、 Tk 内のウィジェットは、ファイルシステムにおけるファイルと同様、**パス名** *(pathname)* を使って名づけられます。トップレベルのウィジェット、すなわち **ルート** は `.` (ピリオド) という名前になり、その子ウィジェット階層もピリオドで区切ってゆきます。ウィジェットの名前は、例えば`.myApp.controlPanel.okButton` のようになります。

options ウィジェットの見た目を設定します。場合によってはウィジェットの挙動も設定します。オプションはフラグと値がリストになった形式をとります。UNIX のシェルコマンドのフラグと同じように、フラグの前には '-' がつき、複数の単語からなる値はクオートで囲まれます。

以下に例を示します:

```
button   .fred   -fg red -text "hi there"
   ^       ^     \_____________________/
   |       |                |
 class    new            options
command  widget  (-opt val -opt val ...)
```

ウィジェットを作成すると、ウィジェットへのパス名は新しいコマンドになります。この新たな *widget command* は、プログラマが新たに作成したウィジェットに *action* を実行させる際のハンドル (handle) になります。C では someAction(fred, someOptions) と表し、C++では fred.someAction(someOptions) と表すでしょう。Tk では:

```
.fred someAction someOptions
```

のようにします。オブジェクト名 `.fred` はドットから始まっているので注意してください。

読者の想像の通り、*someAction* に指定できる値はウィジェットのクラスに依存しています: fred がボタンなら `.fred disable` はうまくいきます (fred はグレーになります) が、fred がラベルならうまくいきません (Tk ではラベルの無効化をサポートしていないからです)。

someOptions に指定できる値はアクションの内容に依存しています。`disable` のようなアクションは引

数を必要としませんが、テキストエントリボックスの delete コマンドのようなアクションにはテキストを削除する範囲を指定するための引数が必要になります。

20.1.4 基本的な Tk プログラムと Tkinter との対応関係

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ています。

```
button .fred                =====>  fred = Button()
```

オブジェクトの親 (master) は、オブジェクトの作成時に指定した新たな名前から非明示的に決定されます。Tkinter では親を明示的に指定します。

```
button .panel.fred          =====>  fred = Button(panel)
```

Tk の設定オプションは、ハイフンをつけたタグと値の組からなるリストで指定します。Tkinter では、オプションはキーワード引数にしてインスタンスのコンストラクタに指定したり、config にキーワード引数を指定して呼び出したり、インデクス指定を使ってインスタンスに代入したりして設定します。オプションの設定については 20.1.6 節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 でウィジェットにアクションを実行させるには、ウィジェット名をコマンドにして、その後にアクション名を続け、必要に応じて引数 (オプション) を続けます。 Tkinter では、クラスインスタンスのメソッドを呼び出して、ウィジェットのアクションを呼び出します。あるウィジェットがどんなアクション (メソッド) を実行できるかは、Tkinter.py モジュール内にリストされています。

```
.fred invoke                =====>  fred.invoke()
```

Tk でウィジェットを packer (ジオメトリマネジャ) に渡すには、pack コマンドをオプション引数付きで呼び出します。Tkinter では Pack クラスがこの機能すべてを握っていて、 様々な pack の形式がメソッドとして実装されています。Tkinter のウィジェットは全て Packer からサブクラス化されているため、pack 操作にまつわる全てのメソッドを継承しています。Form ジオメトリマネジャに関する詳しい情報については Tix モジュールのドキュメントを参照してください。

```
pack .fred -side left       =====>  fred.pack(side = "left")
```

20.1.5 Tk と Tkinter はどのように関わっているのか

注意: 以下の構図は図版をもとに書き下ろしたものです。このドキュメントの今後のバージョンでは、図版をもっと直接的に利用する予定です。

上から下に、呼び出しの階層構造を説明してゆきます:

あなたのアプリケーション (Python) まず、 Python アプリケーションが Tkinter を呼び出します。

Tkinter (Python モジュール) 上記の呼び出し (例えば、ボタンウィジェットの作成) は、*Tkinter* モジュー

ル内で実現されており、Python で書かれています。この Python で書かれた関数は、コマンドと引数を解析して変換し、あたかもコマンドが Python スクリプトではなく Tk スクリプトから来たようにみせかけます。

tkinter (C) 上記のコマンドと引数は *tkinter* (小文字です。注意してください) 拡張モジュール内の C 関数に渡されます 。

Tk Widgets (C and Tcl) 上記の C 関数は、Tk ライブラリを構成する C 関数の入った別の C モジュールへの呼び出しを行えるようになっています。Tk は C と Tcl を少し使って実装されています。Tk ウィジェットの Tcl 部分は、ウィジェットのデフォルト動作をバインドするために使われ、Python で書かれた `Tkinter` モジュールが import される時点で一度だけ実行されます。(ユーザがこの過程を目にすることはありません。

Tk (C) Tk ウィジェットの Tk 部分で実装されている最終的な対応付け操作によって...

Xlib (C) Xlib ライブラリがスクリーン上にグラフィックスを描きます。

20.1.6 簡単なリファレンス

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などを制御します。オプションの設定には三通りの方法があります:

オブジェクトを作成する時にキーワード引数を使う :

```
fred = Button(self, fg = "red", bg = "blue")
```

オブジェクトを作成した後、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"
```

オブジェクトを生成した後、config() メソッドを使って複数の属性を更新する :

```
fred.config(fg = "red", bg = "blue")
```

オプションとその振る舞いに関する詳細な説明は、該当するウィジェットの Tk の man マニュアルを参照してください。

man マニュアルには、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)" がリストされていることに注意しましょう。前者は多くのウィジェットに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。標準オプションの説明は man マニュアルの *options*(3) にあります。

このドキュメントでは、標準オプションとウィジェット固有のオプションを区別していません。オプションによっては、ある種のウィジェットに適用できません。あるウィジェットがあるオプションに対応しているかどうかは、ウィジェットのクラスによります。例えばボタンには `command` オプションがありますが、ラベルにはありません。

あるウィジェットがどんなオプションをサポートしているかは、ウィジェットの man マニュアルにリストされています。また、実行時にウィジェットの `config()` メソッドを引数なしで呼び出したり、`keys()` メソッドを呼び出したりして問い合わせることもできます。メソッド呼び出しを行うと辞書型の値を返します。この辞書は、オプションの名前がキー (例えば `'relief'`) になっていて、値が 5 要素のタプルになっています。

`bg` のように、いくつかのオプションはより長い名前を持つ共通のオプションに対する同義語になっています (`bg` は "background" を短縮したものです)。短縮形のオプション名を `config()` に渡すと、5 要素ではなく 2 要素のタプルを返します。このタプルには、同義語の名前と 「本当の」オプション名が入っています (例えば `('bg', 'background')`)。

インデックス	意味	例
0	オプション名	`'relief'`
1	データベース検索用のオプション名	`'relief'`
2	データベース検索用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、実際に出力される辞書には利用可能なオプションが全て表示されます。上の表示例は単なる例にすぎません。

Packer

packer は Tk のジオメトリ管理メカニズムの一つです。

ジオメトリマネジャは、複数のウィジェットの位置を、それぞれのウィジェットを含むコンテナ - 共通の**マスタ** *(master)* からの相対で指定するために使います。やや扱いにくい *placer* (あまり使われないのでここでは取り上げません) と違い、packer は定性的な関係を表す指定子 - **上** *(above)*、**〜の左** *(to the left of)*、**引き延ばし** *(filling)* など - を受け取り、厳密な配置座標の決定を全て行ってくれます。

どんな**マスタ** ウィジェットでも、大きさは内部の "スレイブ (slave) ウィジェット" の大きさで決まります。packer は、スレイブウィジェットを pack 先のマスタウィジェット中のどこに配置するかを制御するために使われます。望みのレイアウトを達成するには、ウィジェットをフレームにパックし、そのフレームをまた別のフレームにパックできます。さらに、一度パックを行うと、それ以後の設定変更に合わせて動的に並べ方を調整します。

ジオメトリマネジャがウィジェットのジオメトリを確定するまで、ウィジェットは表示されないので注意してください。初心者のころにはよくジオメトリの確定を忘れてしまい、ウィジェットを生成し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、(例えば packer の `pack()` メソッドを適用して) ジオメトリを確定した後で初めて表示されます。

pack() メソッドは、キーワード引数つきで呼び出せます。キーワード引数は、ウィジェットをコンテナ内のどこに表示するか、メインのアプリケーションウィンドウをリサイズしたときにウィジェットがどう振舞うかを制御します。以下に例を示します:

```
fred.pack()                         # デフォルトでは、side = "top"
fred.pack(side = "left")
fred.pack(expand = 1)
```

Packer のオプション

packer と packer の取りえるオプションについての詳細は、man マニュアルや John Ousterhout の本の 183 ページを参照してください。

anchor アンカーの型です。 packer が区画内に各スレイブを配置する位置を示します。

expand ブール値で、`0` または `1` になります。

fill 指定できる値は `'x'`、`'y'`、`'both'`、`'none'` です。

ipadxとipady スレイブウィジェットの各側面の内側に行うパディング幅を表す長さを指定します。

padxとpady スレイブウィジェットの各側面の外側に行うパディング幅を表す長さを指定します。

side 指定できる値は `'left'`,`'right'`,`'top'`,`'bottom'` です。

ウィジェット変数を関連付ける

ウィジェットによっては、(テキスト入力ウィジェットのように) 特殊なオプションを使って、現在設定されている値をアプリケーション内の変数に直接関連付けできます。このようなオプションには `variable`、`textvariable`、`onvalue`、`offvalue` および `value` があります。この関連付けは双方向に働きます: 変数の値が何らかの理由で変更されると、関連付けされているウィジェットも更新され、新しい値を反映します。

残念ながら、現在の `Tkinter` の実装では、`variable` や `textvariable` オプションでは任意の Python の値をウィジェットに渡せません。この関連付け機能がうまく働くのは、`Tkinter` モジュール内で Variable というクラスからサブクラス化されている変数によるオプションだけです。

Variable には、`StringVar`、`IntVar`、`DoubleVar` および `BooleanVar` といった便利なサブクラスがすでにすでに数多く定義されています。こうした変数の現在の値を読み出したければ、`get()` メソッドを呼び出します。また、値を変更したければ `set()` メソッドを呼び出します。このプロトコルに従っている限り、それ以上なにも手を加えなくてもウィジェットは常に現在値に追従します。

例えば:

```
class App(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()

        self.entrythingy = Entry()
        self.entrythingy.pack()

        # アプリケーション変数です
        self.contents = StringVar()
        # 変数の値を設定します
        self.contents.set("this is a variable")
        # エントリウィジェットに変数の値を監視させます
        self.entrythingy["textvariable"] = self.contents

        # ユーザがリターンキーを押した時にコールバックを呼び出させます
        # これで、このプログラムは、ユーザがリターンキーを押すと
        # アプリケーション変数の値を出力するようになります。
        self.entrythingy.bind('<Key-Return>',
                              self.print_contents)

    def print_contents(self, event):
        print "hi. contents of entry is now ---->", \
              self.contents.get()
```

ウィンドウマネジャ

Tk には、ウィンドウマネジャとやり取りするための wm というユーティリティコマンドがあります。wm コマンドにオプションを指定すると、タイトルや配置、アイコンビットマップなどを操作できます。Tkinter では、こうしたコマンドは 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ているので、Wm のメソッドを直接呼び出せます。

あるウィジェットの入っているトップレベルウィンドウを取得したい場合、大抵は単にウィジェットのマスタを参照するだけですみます。とはいえ、ウィジェットがフレーム内にパックされている場合、マスタはトップレベルウィンドウではありません。任意のウィジェットの入っているトップレベルウィンドウを知りたければ _root() メソッドを呼び出してください。このメソッドはアンダースコアがついていますが、これはこの関数が Tkinter の実装の一部であり、Tk の機能に対するインタフェースではないことを示しています。

以下に典型的な使い方の例をいくつか挙げます:

```
from Tkinter import *
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()

#
# ウィンドウマネジャクラスのメソッドを呼び出します。
#
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" "#RRRGGGBBB", 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ク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 '*width*x*height*' 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・インスタン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/Pixelmap 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ところならどこでも、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ようとも、Tk は空の箱を表示します。

20.2 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フェース作成し、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。

Tix Development Applications
(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グするためのインスペクタ **TixInspect** が含まれます。

20.2.1 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]
```

20.2.2 Tix ウィジェット

Tix 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” 型のインタフェー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。

ファイルセレクタ

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ジャを作成 (Form) します。

20.2.3 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か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 と言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。

20.3 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。

]

20.4 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。

setup(***kwargs*)
: メインウインドウの大きさと位置を設定します。キーワードは：

  - width: ピクセル数かスクリーンに対する割合での大きさ。デフォルトはスクリーンの 50% です。
  - height: ピクセル数かスクリーンに対する割合での大きさ。デフォルトはスクリーンの 50% です。
  - startx: スクリーン左端からのピクセル数での開始位置。None はデフォルト値で、スクリーンの水平方向にセンタリングします。
  - starty: スクリーン左端からのピクセル数での開始位置。None はデフォルト値で、スクリーンの垂直方向にセンタリングします。

  例：

```
# デフォルトのジオメトリを利用： スクリーンの 50% x 50%、センタリング。
setup()

# ウインドウを 200x200 ピクセル、スクリーンの左上。
setup (width=200, height=200, startx=0, starty=0)

# ウインドウをスクリーンの 75% x 50% にして、センタリング。
setup(width=.75, height=0.5, startx=None, starty=None)
```

title(*title_str*)

ウインドウのタイトルを *title* に設定します。

done()

Tk のメインループに入ります。ウインドウは、クローズされるか、プロセスが kill されるまで表示され続けます。

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。

speed(*speed*)

タートルのスピードを設定します。*speed* パラメータに適切な値は'fastest'（ウェイト無し）、'fast'（5ms のウェイト）、'normal'（10ms のウェイト）、'slow'（15ms のウェイト）、それと 'slowest'（20ms のウェイト）です。 2.5 で追加された仕様です。

delay(*delay*)

タートルのスピードを *delay* に設定します。これは ms で与えます。 2.5 で追加された仕様です。

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。

begin_fill()
タートルを塗りつぶしモードにします。後には、対応する end_fill() 呼び出しが続かなければいけません。さもないと、これは無視されてしまいます。 2.5 で追加された仕様です。

end_fill()
塗りつぶしモードを終了し、図形を塗りつぶします； `fill(0)` と等価です。End filling mode, and fill the shape; equivalent to `fill(0)`. 2.5 で追加された仕様です。

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。

towards(*x, y*)
タートルの位置から点 *x*、*y* までの線の角度を返します。この座標は二つの別々の引数、2 タプルまたは別のペンオブジェクトとして指定できます。 2.5 で追加された仕様です。

heading()
タートルの現在の向きを返します。 2.3 で追加された仕様です。

setheading(*angle*)
タートルの向きを *angle* に設定します。 2.3 で追加された仕様です。

position()
タートルの現在の位置を `(x,y)` のペアで返します。 2.3 で追加された仕様です。

setx(*x*)
タートルの x 座標を *x* に設定します。 2.3 で追加された仕様です。

sety(*y*)
タートルの y 座標を *y* に設定します。Set the y coordinate of the turtle to *y*. 2.3 で追加された仕様です。

window_width()
キャンバスウインドウの幅を返します。 2.3 で追加された仕様です。

window_height()
キャンバスウインドウの高さを返します。 2.3 で追加された仕様です。

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バスを自動的に作成します。

class `Turtle`()

ペンを定義します。これは基本的に `Pen()` と同義です; `Turtle` は、`Pen` の空の派生クラスです。

class `RawPen`(*canvas*)

キャンバス *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。

20.4.1 Turtle、Pen と RawPen オブジェクト

モジュールで利用可能なグローバル関数の大部分は `Turtle`、`Pen` や `RawPen` のメソッドとしても利用可能で、これは特定のペンの状態にだけ影響します。

メソッドとして強力になっているメソッドは `degrees()` だけで、これは 1 回転相当の単位数を指定できるオプション引数を取ります。

`degrees`([*fullcircle*])

fullcircle はデフォルトで 360 です。たとえ *fullcircle* にラジアンで 2*π、あるいは度で 400 を与えようとも、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。

20.5 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-プラットホーム: 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)

20.5.1 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

20.5.2 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

20.5.3 構文の色づけ

色づけはバックグランド “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。カラー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。

20.6 他のグラフィカルユーザインタフェースパッケージ

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 インタフェースを持ち、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 Ulferts 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。

他にも Python で使える GUI パッケージがあります。

wxPython
(http://www.wxpython.org)

wxPython はクロスプラットフォームの Python 用 GUI ツールキットで、人気のある wxWidgets C++ ツールキットに基づいて作られています。このツールキットは Windows, Mac OS X および UNIX システムのアプリケーションに、それぞれのプラットフォームのネイティブなウィジェットを可能ならば利用して (UNIX 系のシステムでは GTK+)、ネイティブなルック&フィールを提供します。多彩なウィジェットの他に、オンラインドキュメントや場面に応じたヘルプ、印刷、HTML 表示、低級デバイスコンテキスト描画、ドラッグ&ドロップ、システムクリップボードへのアクセス、XML に基づいたリソースフォーマット、さらにユーザ寄贈のモジュールからなる成長し続けているライブラリ等々を wxPython は提供しています。wxWidget も wxPython もどちらのプロジェクトも活発に開発が続けられ改良が進められており、活動的で親切なユーザと開発者のコミュニティがあります。

wxPython in Action
(http://www.amazon.com/exec/obidos/ASIN/1932394621)

Noel Rappin と Robin Dunn による wxPython の本。

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フェースです。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 へのインタフェースを提供する Python 拡張モジュールです。FOX は、グラフィカルユーザインタフェー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/~james/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フェースを提供します。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。

二十一章

国際化

この章ので解説されるモジュールはプログラムのメッセージで使用される言語を選択する、または出力を地域の習慣に従って変更するメカニズムを提供して言語や地域に依存しないソフトの開発を支援します。

この章で解説されるモジュールの一覧は:

gettext 多言語対応に関する国際化サービス。

locale 国際化サービス。

21.1 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。

21.1.1 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。[1]

bind_textdomain_codeset (*domain*[, *codeset*])

domain を *codeset* に結び付けて、gettext() ファミリの関数が返す文字列のエンコード方式を変更します。*codeset* を省略すると、現在結び付けられているコードセットを返します。

[1] 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、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。

2.4 で追加された仕様です。

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)。

lgettext(*message*)
`gettext()` と同じですが、`bind_textdomain_codeset()` で特にエンコードを指定しない限り、翻訳結果を優先システムエンコーディング (preferred system encoding) で返します。

2.4 で追加された仕様です。

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。

ldgettext(*message*)
`dgettext()` と同じですが、`bind_textdomain_codeset()` で特にエンコードを指定しない限り、翻訳結果を優先システムエンコーディング (preferred system encoding) で返します。

2.4 で追加された仕様です。

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。

lngettext(*message*)
`ngettext()` と同じですが、`bind_textdomain_codeset()` で特にエンコードを指定しない限り、翻訳結果を優先システムエンコーディング (preferred system encoding) で返します。

2.4 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。

ldngettext(*message*)
`dngettext()` と同じですが、`bind_textdomain_codeset()` で特にエンコードを指定しない限り、翻訳結果を優先システムエンコーディング (preferred system encoding) で返します。

2.4 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.')
```

21.1.2 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。この "翻訳" クラスのインスタンスも、組み込み名前空間に関数 `_()` として組みこみ (install) 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、*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。[2]

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*[, *codeset*]]]]])

`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。*codeset* を指定した場合、翻訳文字列のエンコードに使う文字セットを変更します。

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と見なされ、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。

[2] 上の `bindtextdomain()` に関する脚注を参照してください。

2.4 で変更された仕様: *codeset* パラメタを追加しました

install(*domain*[, *localedir*[, *unicode* [, *codeset*[, *names*]]]])

`translation()` に *domain*、 *localedir*、および *codeset* を渡してできる関数 _ を Python の組み込み名前空間に組み込みます。*unicode* フラグは `translation()` の返す翻訳オブジェクトの `install` メソッドに渡されます。

names パラメタについては、翻訳オブジェクトの `install` メソッドの説明を参照ください。

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。

2.4 で変更された仕様: *codeset* パラメタを追加しました 2.5 で変更された仕様: *names* パラメタを追加しました

`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:

__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) ようになっています。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。

lgettext(*message*)

代替オブジェクトが設定されている場合、`lgettext()` 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。 2.4 で追加された仕様です。

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。

lngettext (*singular, plural, n*)
代替オブジェクトが設定されている場合、lngettext() 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。 2.4 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 ()
“protected” の_info 変数を返します。

charset ()
“protected” の_charset 変数を返します。

output_charset ()
翻訳メッセージとして返す文字列のエンコードを決める、“protected” の_output_charset 変数を返します。

2.4 で追加された仕様です。

set_output_charset (*charset*)
翻訳メッセージとして返す文字列のエンコードを決める、“protected” の変数 _output_charset を変更します。

2.4 で追加された仕様です。

install ([*unicode* [, *names*]])
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。

names パラメタには、_() 以外に組み込みの名前空間にインストールしたい関数名のシーケンスを指定します。サポートしている名前は'gettext' (*unicode* フラグの設定に応じて self.gettext() あるいは self.ugettext() のいずれかに対応します)、'ngettext' (*unicode* フラグの設定に応じて self.ngettext() あるいは self.ungettext() のいずれかに対応します)、'lgettext' および 'lngettext' 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。

2.5 で変更された仕様: *names* パラメタを追加しました

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) % {'num': 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, localedir)
_ = 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。

21.1.3 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 [3]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**[4]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単一のモジュールを国際化するのか、それともアプリケーション全体を国際化するのかによります。次のふたつのセクションで、それぞれについて説明します。

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

[3] 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://po-utils.progiciels-bpi.ca/ で入手できます。

[4] **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。

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ようにします:

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.lgettext
```

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('myapplication', languages=['en'])
lang2 = gettext.translation('myapplication', languages=['fr'])
lang3 = gettext.translation('myapplication', languages=['de'])

# start by using language1
lang1.install()

# ... time goes by, user selects language 2
lang2.install()

# ... more time goes by, user selects language 3
lang3.install()
```

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トです:

```
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。

こうした状況を処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 [5]、_() 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。

gettext() vs. lgettext()

Python 2.4 からは、 lgettext() ファミリが導入されました。この関数の目的は、現行の GNU gettext 実装によりよく準拠した別の関数を提供することにあります。翻訳メッセージファイル中で使われているのと同じコードセットを使って文字列をエンコードして返す gettext() と違い、これらの関数は locale.getpreferredencoding() の返す優先システムエンコーディングを使って翻訳メッセージ文字列をエンコードして返します。また、Python 2.4 では、翻訳メッセージ文字列で使われているコードセットを明示的に選べるようにする関数が新たに導入されていることにも注意してください。コードセットを明示的に設定すると、lgettext() でさえ、指定したコードセットで翻訳メッセージ文字列を返します。これは GNU gettext 実装が期待している仕様と同じです。

21.1.4 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

[5] 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。

21.2 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:

カテゴリ	キー名	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。
	'p_cs_precedes/n_cs_precedes'	通貨記号が値の前につくかどうかです (それぞれ正の値、負の値を表します)。
	'p_sep_by_space/n_sep_by_space'	通貨記号と値との間にスペースを入れるかどうかです (それぞれ正の値、負の値を表します)。
	'mon_decimal_point'	金額表示の際に使われる小数点です。
	'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。
	'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。
	'p_sign_posn/n_sign_posn'	符号の位置です (それぞれ正の値と負の値を表します)。以下を参照ください。

数値形式の値に CHAR_MAX を設定すると、そのロケールでは値が指定されていないことを表します。

'p_sign_posn' および 'n_sing_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。
CHAR_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* を含むシーケンスで返します。*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シーケンスを順序付けて並べる際に使うことができます。

format(*format, val*[, *grouping*[, *monetary*]])

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。

monetary が真なら、桁区切り記号やグループ化文字列を用いて変換を行います。

この関数や、1 文字の指定子でしか動作しないことに注意しましょう。フォーマット文字列を使う場合は `format_string()` を使用します。

2.5 で変更された仕様: *monetary* パラメータが追加されました

format_string (*format, val*[, *grouping*])

`format %val` 形式のフォーマット指定子を、現在のロケール設定を考慮したうえで処理します。

2.5 で追加された仕様です。

currency (*val*[, *symbol*[, *grouping*[, *international*]]])

数値 *val* を、現在の `LC_MONETARY` の設定にあわせてフォーマットします。

symbol が真の場合は、返される文字列に通貨記号が含まれるようになります。これはデフォルトの設定です。*grouping* が真の場合 (これはデフォルトではありません) は、値をグループ化します。*international* が真の場合 (これはデフォルトではありません) は、国際的な通貨記号を使用します。

この関数は 'C' ロケールでは動作しないことに注意しましょう。まず最初に `setlocale()` でロケールを設定する必要があります。

2.5 で追加された仕様です。

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.getlocale(locale.LC_ALL) # get current locale
>>> locale.setlocale(locale.LC_ALL, 'de_DE') # use German locale; name might vary with platf
>>> 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
```

21.2.1 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。

21.2.2 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)。

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 _locate モジュールにアクセスできないようにしてください。

21.2.3 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、locake 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()、bindtextdomain()、および bind_textdomain_codeset() 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 dcgettext() 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。

二十二章

プログラムのフレームワーク

この章で解説されるモジュールはあなたのプログラムの大枠を規定するフレームワークです。現状では、ここで解説されるモジュールは全てコマンドラインインタフェースを書くためのものです。

この章の完全な一覧は:

cmd 行指向のコマンドインタープリタを構築

shlex UNIX シェル類似の言語に対する単純な字句解析。

22.1 cmd — 行指向のコマンドインタープリタのサポート

Cmd クラスでは、行指向のコマンドインター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、こうしたインター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、行指向のインタープリタ・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。

オプション引数 *stdin* と *stdout* には、Cmd またはその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* が使用されます。

2.3 で変更された仕様: 引数 *stdin* と *stdout* を追加

22.1.1 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)。

このメソッドは postcmd() メソッドが真を返したときに return します。postcmd() に対する *stop* 引数は、このコマンドが対応する do_*() メソッドからの返り値です。

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。コマンド *str* に対応する do_*() メソッドがある場合、そのメソッドの返り値が返されます。そうでない場合は default() メソッドからの返り値が返されます。

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、インタープリタが自動的に **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。)

22.2 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。

注意: モジュール shlex は今のところユニコード入力をサポートしていません。

22.2.1 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*])
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`(デフォルト値) 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*[, *infile*[, *posix*]]])
`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、文字列でなくてはいけません（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。22.2.2 節を参照のこと。

参考資料:

`ConfigParser` モジュール (9.2 節):
Windows ‘.ini’ ファイルに似た設定ファイルのパーザ。

22.2.2 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オ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が取り出され、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

きと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なことがあります。

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、入力は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なことがあります。

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。

22.2.3 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

二十三章

開発ツール

この章で紹介されるモジュールはソフトウェアを書くことを支援します。たとえば、pydoc モジュールはモジュールの内容からドキュメントを生成します。doctest と unittest モジュールで自動的に実行して予想通りの出力が生成されるか確認するユニットテストを書くことができます。

この章で解説されるモジュールの完全な一覧は:

pydoc	ドキュメント生成とオンラインヘルプシステム
doctest	対話モードを使った使用例の内容を検証するためのフレームワーク。
unittest	単体テストフレームワーク
test.test_support	Python 回帰テストのサポート

23.1 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) 含む場合になります)、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、お好きな 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。

コアモジュールのドキュメントは http://www.python.org/doc/current/lib/ にあると仮定されています。これは、ライブラリリファレンスマニュアルを置いている異なる URL かローカルディレクトリを 環境変数 PYTHONDOCS に設定することでオーバーライドすることができます。

23.2 doctest — 対話モードを使った使用例の内容をテストする

`doctest` モジュールは、対話的 Python セッションのように見えるテキストを探し出し、セッションの内容を実行して、そこに書かれている通りに振舞うかを調べます。 `doctest` は以下のような用途によく使われています:

- モジュールの docstring (ドキュメンテーション文字列) 中にある対話モードでの使用例全てが書かれている通りに動作するかを検証することで、docstring の内容が最新のものになるよう保ちます。
- テストファイルやテストオブジェクト中の対話モードにおける使用例が期待通りに動作するかを検証することで、回帰テストを実現します。
- 入出力例をふんだんに使ったパッケージのチュートリアルドキュメントを書けます。入出力例と解説文のどちらに注目するかによって、ドキュメントは「読めるテスト」にも「実行できるドキュメント」にもなります。

以下にちょっとした、それでいて完全な例を示します:

`doctest` モジュールは、モジュールの docstring から、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。

```
"""
This is the "example" module.

The example module supplies one function, factorial().  For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    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)
    265252859812191058636308480000000L
    >>> factorial(30L)
    265252859812191058636308480000000L
    >>> 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)
    265252859812191058636308480000000L

    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """
```

```
        import math
        if not n >= 0:
            raise ValueError("n must be >= 0")
        if math.floor(n) != n:
            raise ValueError("n must be exact integer")
        if n+1 == n:  # catch a value like 1e300
            raise OverflowError("n too large")
        result = 1
        factor = 2
        while factor <= n:
            result *= factor
            factor += 1
        return result

    def _test():
        import doctest
        doctest.testmod()

    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。スクリプトに **-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
Trying:
    factorial(5)
Expecting:
    120
ok
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
```

といった具合で、最後には:

```
Trying:
    factorial(1e100)
Expecting:
    Traceback (most recent call last):
        ...
    OverflowError: n too large
ok
1 items had no tests:
    __main__._test
2 items passed all tests:
   1 tests in __main__
   8 tests in __main__.factorial
9 tests in 3 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。詳細な事柄は後続の各節で全て説明しています。doctest の例は、標準の Python テストスイートやライブラリ中に沢山あります。標準のテストファイル 'Lib/test/test_doctest.py' には、特に便利な例題があります。

'doctest.py' 内の docstring 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。

23.2.1 簡単な利用法: docstring 中の例題をチェックする

doctest を試す簡単な方法、(とはいえ、いつもそうする必要はないのですが) は、各モジュール M の最後を、以下:

```
def _test():
    import doctest, M
    doctest.testmod()

if __name__ == "__main__":
    _test()
```

のようにして締めくくるやりかたです。

こうすると、doctest は M 中の docstring を検査します。モジュールをスクリプトとして実行すると、docstring 中の例題が実行され、検証されます:

```
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、最後に '`***Test Failed***` *N* `failures.`' という行を出力します。ここで、*N* は失敗した例題の数です。

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=True を testmod() に渡せば、詳細報告 (verbose) モードを強制できます。また、verbose=False にすれば禁止できます。どちらの場合にも、testmod() は sys.argv 上のスイッチを調べません。(従って、**-v** をつけても効果はありません)。

testmod() の詳しい情報は 23.2.7 節を参照してください。

23.2.2 簡単な利用法: テキストファイル中の例題をチェックする

doctest のもう一つの簡単な用途は、テキストファイル中にある対話操作の例に対するテストです。これには testfile() 関数を使います:

```
import doctest
doctest.testfile("example.txt")
```

この短いスクリプトは、'example.txt' というファイルの中に入っている対話モードの Python 操作例全てを実行して、その内容を検証します。ファイルの内容は一つの巨大な docstring であるかのように扱われます; ファイルが Python プログラムでなくてもよいのです！ 例えば、'example.txt' には以下のような内容が入っているかもしれません:

```
The ``example`` module
======================

Using ``factorial``
-------------------
This is an example text file in reStructuredText format.  First import
``factorial`` from the ``example`` module:

    >>> from example import factorial

Now use it:

    >>> factorial(6)
    120
```

doctest.testfile("example.txt") を実行すると、このドキュメント内のエラーを見つけ出します:

```
File "./example.txt", line 14, in example.txt
Failed example:
    factorial(6)
Expected:
    120
Got:
    720
```

testmod() と同じく、testfile() は例題が失敗しない限り何も表示しません。例題が失敗すると、失敗した例題とその原因が (場合によっては複数) testmod() と同じ書式で標準出力に書き出されます。

デフォルトでは、testfile() は自分自身を呼び出したモジュールのあるディレクトリを探します。その他の場所にあるファイルを見に行くように testfile() に指示するためのオプション引数についての説明は 23.2.7 節を参照してください。

testmod() と同様、コマンドラインオプション **-v** またはオプションのキーワード引数 *verbose* を使う

と、testfile() の冗長度を設定できます。

testfile() の詳細は 23.2.7 節を参照してください。

23.2.3 doctest のからくり

この節では、doctest のからくり: どの docstring を見に行くのか、どうやって対話操作例を見つけ出すのか、どんな実行コンテキストを使うのか、例外をどう扱うか、上記の振る舞いを制御するためにどのようなオプションフラグを使うか、について詳しく吟味します。こうした情報は、doctest に対応した例題を書くために必要な知識です; 書いた例題に対して実際に doctest を実行する上で必要な情報については後続の節を参照してください。

どのドキュメンテーション文字列が検証されるのか?

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。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も検索され、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。

2.4 で変更された仕様: "プライベート名" の概念は撤廃されたため、今後はドキュメントにしません

23.2.4 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。とはいえ、doctest は特定の Python シェルの振る舞いを正確にエミュレーションしようとするわけではありません。ハードタブは全て 8 カラムのタブストップを使ってスペースに展開されます。従って、タブがそのように表現されると考えておかないととまずいことになります: その場合は、ハードタブを使わないか、自前で DocTestParser クラスを書いてください。

2.4 で変更された仕様: 新たにタブをスペースに展開するようになりました; 以前のバージョンはハードタブを保存しようとしていたので、混乱させるようなテスト結果になってしまっていました

```
>>> # 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。

細かな注意:

- 出力結果例には、全て空白の行が入っていてはなりません。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。もし予想出力結果の内容に空白行が入っている場合には、空白行が入るべき場所全てに<BLANKLINE> を入れてください。 2.4 で変更された仕様: <BLANKLINE> を追加しました; 以前のバージョンでは、空白行の入った予想出力結果を扱う方法がありませんでした
- 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、raw docstring を使ってバックスラッシュを入力どおりに扱わせるようにせねばなりません:

  ```
  >>> def f(x):
  ...     r'''Backslashes in a raw docstring: m\n'''
  >>> print f.__doc__
  Backslashes in a raw docstring: m\n
  ```

  こうしなければ、バックスラッシュは文字列の一部として解釈されてしまいますう。例えば、上の例の "\" は改行文字として認識されてしまうでしょう。こうする代わりに、(raw docstring を使わずに) doctest 版の中ではバックスラッシュを全て二重にしてもかまいません:

  ```
  >>> def f(x):
  ...     '''Backslashes in a raw docstring: m\\n'''
  >>> print f.__doc__
  Backslashes in a raw docstring: m\n
  ```

- 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、例題の開始部分にあたる'»> ' 行の先頭にある空白文字列と同じだけはぎとられます。

23.2.5 実行コンテキストとは何か?

デフォルトでは、doctest はテストを行うべき docstring を見つけるたびに M のグローバル名前空間の**浅いコピー**を使い、テストの実行によってモジュールの実際のグローバル名前空間を変更しないようにし、かつ M 内で行ったテストが痕跡を残して偶発的に別のテストを誤って動作させないようにしています。従って、例題中では M 内のトップレベルで定義されたすべての名前と、docstring 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を自由に使えます。個々の例題は他の docstring 中で定義された名前を参照できません。

testmod() や testfile() に globs=your_dict を渡し、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。

23.2.6 例外はどう扱えばよいのですか?

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: 単にトレースバックを貼り付けてください。[1] トレースバックには、頻繁に変更されがちな情報が入っている (例えばファイルパスや行番号など) ものなので、受け入れるべきテスト結果に柔軟性を持たせようと doctest が苦労している部分の一つです。

簡単な例を示しましょう:

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: list.remove(x): x not in list
>>>
```

この doctest は ValueError が送出され、かつ詳細情報に 'list.remove(x): x not in list' が入っている場合にのみ成功します。

例外が発生したときの予想出力はトレースバックヘッダから始まっていなければなりません。トレースバックの形式は以下の二通りの行のいずれかでよく、例題の最初の行と同じインデントでなければりません:

```
Traceback (most recent call last):
Traceback (innermost last):
```

トレースバックヘッダの後ろにトレースバックスタックを続けてもかまいませんが、doctest はその内容を無視します。普通はトレースバックスタックを無視するか、対話セッションからそのままコピーしてきます。

トレースバックスタックの後ろにはもっとも有意義な部分、例外の型と詳細情報の入った行があります。通常、この行はトレースバックの末尾にあるのですが、例外が複数行の詳細情報を持っている場合、複数の行にわたることもあります:

[1] 予想出力結果と例外の両方を含んだ例はサポートされていません。一方の終わりと他方の始まりを見分けようとするのはエラーの元になりがちですし、解りにくいテストになってしまいます。

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: multi
    line
detail
```

上の例では、最後の 3 行 (ValueError から始まる行) における例外の型と詳細情報だけが比較され、それ以外の部分は無視されます。

例外を扱うコツは、例題をドキュメントとして読む上で明らかに価値のある情報でない限り、トレースバックスタックは無視する、ということです。従って、先ほどの例は以下のように書くべきでしょう:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
    ...
ValueError: multi
    line
detail
```

トレースバックの扱いは非常に特殊なので注意してください。特に、上の書き直した例題では、'...' の扱いが doctest の ELLIPSIS オプションによって変わります。この例での省略記号は何かの省略を表しているかもしれませんし、コンマや数字が 3 個 (または 300 個かもしれませんし、Monty Python のスキットをインデントして書き写したものかもしれません。

以下の詳細はずっと覚えておく必要はないのですが、一度目を通しておいてください:

- doctest は予想出力の出所が print 文なのか例外なのかを推測できません。従って、例えば予想出力が 'ValueError: 42 is prime' であるような例題は、ValueError が実際に送出された場合と、万が一予想出力と同じ文字列を print した場合の両方でパスしてしまいます。現実的には、通常の出力がトレースバックヘッダから始まることはないので、さしたる問題にはなりません。

- トレースバックスタック (がある場合) の各行は、例題の最初の行よりも深くインデントされているか、**または** 英数文字以外で始まっていなければなりません。トレースバックヘッダ以後に現れる行のうち、インデントが等しく英数文字で始まる最初の行は例外の詳細情報が書かれた行とみなされるからです。もちろん、通常のトレースバックでは全く正しく動作します。

- doctest のオプション IGNORE_EXCEPTION_DETAIL を指定した場合、最も左端のコロン以後の内容が無視されます。

- 対話シェルでは、SyntaxError の場合にトレースバックヘッダを無視することがあります。しかし doctest にとっては、例外を例外でないものと区別するためにトレースバックヘッダが必要です。そこで、トレースバックヘッダを省略するような SyntaxError をテストする必要があるというごく稀なケースでは、例題に自分で作ったトレースバックヘッダを追加する必要があるでしょう。

- SyntaxError の場合、 Python は構文エラーの起きた場所を^ マーカで表示します:

```
>>> 1 1
  File "<stdin>", line 1
    1 1
      ^
SyntaxError: invalid syntax
```

例外の型と詳細情報の前にエラー位置を示す行がくるため、 doctest はこの行を調べません。例えば、以下の例では、間違った場所に^ マーカを入れてもパスしてしまいます:

```
>>> 1 1
Traceback (most recent call last):
  File "<stdin>", line 1
    1 1
    ^
SyntaxError: invalid syntax
```

2.4 で変更された仕様: 複数行からなる例外の詳細情報を扱えるようにし、doctest オプション IGNORE_EXCEPTION_DETAIL を追加しました

オプションフラグとディレクティブ

doctest では、その挙動の様々な側面をたくさんのオプションフラグで制御しています。各フラグのシンボル名はモジュールの定数として提供されており、論理和で組み合わせて様々な関数に渡せるようになっています。シンボル名は doctest のディレクティブ (directive, 下記参照) としても使えます。

最初に説明するオプション群は、 テストのセマンティクスを決めます。すなわち、実際にテストを実行したときの出力と例題中の予想出力とが一致しているかどうかを doctest がどうやって判断するかを制御します:

DONT_ACCEPT_TRUE_FOR_1

デフォルトでは、予想出力ブロックに単に 1 だけが入っており、実際の出力ブロックに 1 または True だけが入っていた場合、これらの出力は一致しているとみなされます。0 と False の場合も同様です。DONT_ACCEPT_TRUE_FOR_1 を指定すると、こうした値の読み替えを行いません。デフォルトの挙動で読み替えを行うのは、最近の Python で多くの関数の戻り値型が整数型からブール型に変更されたことに対応するためです; 読み替えを行う場合、"通常の整数" の出力を予想出力とするような doctest も動作します。このオプションはそのうち無くなるでしょうが、ここ数年はそのままでしょう。

DONT_ACCEPT_BLANKLINE

デフォルトでは、予想出力ブロックに <BLANKLINE> だけの入った行がある場合、その行は実際の出力における空行に一致するようになります。完全な空行を入れてしまうと予想出力がそこで終わっているとみなされてしまうため、空行を予想出力に入れたい場合にはこの方法を使わねばなりません。DONT_ACCEPT_BLANKLINE を指定すると、<BLANKLINE> の読み替えを行わなくなります。

NORMALIZE_WHITESPACE

このフラグを指定すると、空白 (空白と改行文字) の列は互いに等価であるとみなします。予想出力における任意の空白列は実際の出力における任意の空白と一致します。デフォルトでは、空白は厳密に一致せねばなりません。NORMALIZE_WHITESPACE は、予想出力の内容が非常に長いために、ソースコード中でその内容を複数行に折り返して書きたい場合に特に便利です。

ELLIPSIS

このフラグを指定すると、予想出力中の省略記号マーカ (...) を実際の出力中の任意の部分文字列に一致させられます。部分文字列は行境界にわたるものや空文字列を含みます。従って、このフラグを使うのは単純な内容を対象にする場合にとどめましょう。複雑な使い方をすると、正規表現に「.*」を使ったときのように"あらら、省略部分をマッチがえてる (match too much) ！" と驚くことになりかねません。

IGNORE_EXCEPTION_DETAIL

このフラグを指定すると、予想される実行結果に例外が入るような例題で、予想通りの型の例外が送出された場合に、例外の詳細情報が一致していなくてもテストをパスさせます。例えば、予想出力が

'ValueError: 42' であるような例題は、実際に送出された例外が 'ValueError: 3*14' でもパスしますが、TypeError が送出されるといった場合にはパスしません。

ELLIPSIS を使っても同様のことができ、IGNORE_EXCEPTION_DETAIL は リリース 2.4 以前の Python を使う人がほとんどいなくなった時期を見計らって撤廃するかもしれないので気をつけてください。それまでは、IGNORE_EXCEPTION_DETAIL は 2.4 以前の Python で例外の詳細については気にせずテストをパスさせるように doctest を書くための唯一の明確な方法です。例えば、

```
>>> (1, 2)[3] = 'moo' #doctest: +IGNORE_EXCEPTION_DETAIL
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

にすると、 Python 2.4 と Python 2.3 の両方でテストをパスさせられます。というのは、例外の詳細情報は 2.4 で変更され、 "doesn't" から "does not" と書くようになったからです。

SKIP

このフラグを指定すると、例題は一切実行されません。こうした機能は doctest の実行例がドキュメントとテストを兼ねていて、ドキュメントのためには含めておかなければならないけれどチェックされなくても良い、というような文脈で役に立ちます。例えば、実行例の出力がランダムであるとか、テスト機構には手が届かない資源に依存している場合などです。

SKIP フラグは一時的に例題を"コメントアウト"するのにも使えます。

COMPARISON_FLAGS

上記の比較フラグ全ての論理和をとったビットマスクです。

二つ目のオプション群は、テストの失敗を報告する方法を制御します:

REPORT_UDIFF

このオプションを指定すると、複数行にわたる予想出力や実際の出力を、一元化 (unified) diff を使って表示します。

REPORT_CDIFF

このオプションを指定すると、複数行にわたる予想出力や実際の出力を、コンテキスト diff を使って表示します。

REPORT_NDIFF

このオプションを指定すると、予想出力と実際の出力との間の差分をよく知られている 'ndiff.py' ユーティリティと同じアルゴリズムを使っている difflib.Differ で分析します。これは、行単位の差分と同じように行内の差分にマーカをつけられるようにする唯一の手段です。例えば、予想出力のある行に数字の 1 が入っていて、実際の出力には l が入っている場合、不一致のおきているカラム位置を示すキャレットの入った行が一行挿入されます。

REPORT_ONLY_FIRST_FAILURE

このオプションを指定すると、各 doctest で最初にエラーの起きた例題だけを表示し、それ以後の例題の出力を抑制します。これにより、正しく書かれた例題が、それ以前の例題の失敗によっておかしくなってしまった場合に、doctest がそれを報告しないようになります。とはいえ、最初に失敗を引き起こした例題とは関係なく誤って書かれた例題の報告も抑制してしまいます。REPORT_ONLY_FIRST_FAILURE を指定した場合、例題がどこかで失敗しても、それ以後の例題を続けて実行し、失敗したテストの総数を報告します; 出力が抑制されるだけです。

REPORTING_FLAGS

上記のエラー報告に関するフラグ全ての論理和をとったビットマスクです。

「doctest ディレクティブ」を使うと、個々の例題に対してオプションフラグの設定を変更できます。doctest ディレクティブは特殊な Python コメント文として表現され、例題のソースコードの後に続けます:

```
directive               ::=  "#" "doctest:" directive_options
directive_options       ::=  directive_option ("," directive_option)*
directive_option        ::=  on_or_off directive_option_name
on_or_off               ::=  "+" | "-"
directive_option_name   ::=  "DONT_ACCEPT_BLANKLINE" | "NORMALIZE_WHITESPACE" | ...
```

+ や- とディレクティブオプション名の間に空白を入れてはなりません。ディレクティブオプション名は上で説明したオプションフラグ名のいずれかです。

ある例題の doctest ディレクティブは、その例題だけの doctest の振る舞いを変えます。ある特定の挙動を有効にしたければ + を、無効にしたければ - を使います。

例えば、以下のテストはパスします:

```
>>> print range(20) #doctest: +NORMALIZE_WHITESPACE
[0,   1,  2,  3,  4,  5,  6,  7,  8,  9,
10,  11, 12, 13, 14, 15, 16, 17, 18, 19]
```

ディレクティブがない場合、実際の出力には一桁の数字の間に二つスペースが入っていないこと、実際の出力は 1 行になることから、テストはパスしないはずです。別のディレクティブを使って、このテストをパスさせることもできます:

```
>>> print range(20) # doctest:+ELLIPSIS
[0, 1, ..., 18, 19]
```

複数のディレクティブは、一つの物理行の中にコンマで区切って指定できます:

```
>>> print range(20) # doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE
[0,    1, ...,   18,    19]
```

一つの例題中で複数のディレクティブコメントを使った場合、それらは組み合わされます:

```
>>> print range(20) # doctest: +ELLIPSIS
...                 # doctest: +NORMALIZE_WHITESPACE
[0,    1, ...,   18,    19]
```

前の例題で示したように、'...' の後ろにディレクティブだけの入った行を例題のうしろに追加して書けます。この書きかたは、例題が長すぎるためにディレクティブを同じ行に入れると収まりが悪い場合に便利です:

```
>>> print range(5) + range(10,20) + range(30,40) + range(50,60)
... # doctest: +ELLIPSIS
[0, ..., 4, 10, ..., 19, 30, ..., 39, 50, ..., 59]
```

デフォルトでは全てのオプションが無効になっており、ディレクティブは特定の例題だけに影響を及ぼすので、通常意味があるのは有効にするためのオプション (+ のついたディレクティブ) だけです。とはいえ、doctest を実行する関数はオプションフラグを指定してデフォルトとは異なった挙動を実現できるので、そのような場合には - を使った無効化オプションも意味を持ちます。

2.4 で変更された仕様: 定数 DONT_ACCEPT_BLANKLINE, NORMALIZE_WHITESPACE, ELLIPSIS, IGNORE_EXCEPTION_DETAIL, REPORT_UDIFF, REPORT_CDIFF, REPORT_NDIFF, REPORT_-

ONLY_FIRST_FAILURE, COMPARISON_FLAGS, REPORTING_FLAGS を追加しました。予想出力中の <BLANKLINE> がデフォルトで実際の出力中の空行にマッチするようになりました。また、doctest ディレクティブが追加されました 2.5 で変更された仕様: 定数 SKIP が追加されました

新たなオプションフラグ名を登録する方法もありますが、doctest の内部をサブクラスで拡張しない限り、意味はないでしょう:

register_optionflag(*name*)

名前 *name* の新たなオプションフラグを作成し、作成されたフラグの整数値を返します。register_optionflag() は OutputChecker や DocTestRunner をサブクラス化して、その中で新たに作成したオプションをサポートさせる際に使います。register_optionflag は以下のような定形文で呼び出さねばなりません:

```
MY_FLAG = register_optionflag('MY_FLAG')
```

2.4 で追加された仕様です。

注意

doctest では、予想出力に対する厳密な一致を厳しく求めています。一致しない文字が一文字でもあると、テストは失敗してしまいます。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"}
```

は失敗するかもしれないのです! 回避するには

```
>>> foo() == {"Hermione": "hippogryph", "Harry": "broomstick"}
True
```

とするのが一つのやり方です。別のやり方は、

```
>>> d = foo().items()
>>> d.sort()
>>> d
[('Harry', 'broomstick'), ('Hermione', 'hippogryph')]
```

です。

他にもありますが、自分で考えてみてください。

以下のように、オブジェクトアドレスを埋め込むような結果を print するのもよくありません:

```
>>> id(1.0) # certain to fail some of the time
7948648
>>> class C: pass
>>> C()   # the default repr() for instances embeds an address
<__main__.C instance at 0x00AC18F0>
```

ELLIPSIS ディレクティブを使うと、上のような例をうまく解決できます:

```
>>> C() #doctest: +ELLIPSIS
<__main__.C instance at 0x...>
```

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。というのも、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、この点では、C ライブラリはプラットフォーム間で非常に大きく異なっているからです。

```
>>> 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どのプラットフォームでもうまく動作するので、私はこの形式の数値を生成するように doctest の例題を工夫しています:

```
>>> 3./4  # utterly safe
0.75
```

このように、単分数 (simple fraction) を使えば、人間にとっても理解しやすくよいドキュメントになります。

23.2.7 基本 API

関数 `testmod()` および `testfile()` は、基本的なほとんどの用途に十分な doctest インタフェースを提供しています。これら二つの関数についてもっとくだけた説明を読みたければ、23.2.1 節および 23.2.2 節を参照してください。

`testfile`(*filename*[, *module_relative*][, *name*][, *package*][, *globs*][, *verbose*][, *report*][, *optionflags*][, *extraglobs*][, *raise_on_error*][, *parser*][, *encoding*])

filename 以外の引数は全てオプションで、キーワード引数形式で指定せねばなりません。

filename に指定したファイル内にある例題をテストします。'(*failure_count*, *test_count*)' を返します。

オプション引数の *module_relative* は、ファイル名をどのように解釈するかを指定します:

- *module_relative* が `True` (デフォルト) の場合、*filename* は OS に依存しないモジュールの相対パスになります。デフォルトでは、このパスは関数 `testfile` を呼び出しているモジュールからの相対パスになります; ただし、*package* 引数を指定した場合には、パッケージからの相対になります。OS への依存性を除くため、*filename* ではパスを分割する文字に / を使わねばならず、絶対パスにしてはなりません (パス文字列を / で始めてはなりません)。
- *module_relative* が `False` の場合、*filename* は OS 依存のパスを示します。パスは絶対パスでも相対パスでもかまいません; 相対パスにした場合、現在の作業ディレクトリを基準に解決します。

オプション引数 *name* には、テストの名前を指定します; デフォルトの場合や `None` を指定した場合、`os.path.basename`(*filename*) になります。

オプション引数 *package* には、Python パッケージを指定するか、モジュール相対のファイル名の場合には相対の基準ディレクトリとなる Python パッケージの名前を指定します。パッケージを指定しない倍、関数を呼び出しているモジュールのディレクトリを相対の基準ディレクトリとして使います。*module_relative* を `False` に指定している場合、*package* を指定するとエラーになります。

オプション引数 *globs* には辞書を指定します。この辞書は、例題を実行する際のグローバル変数として用いられます。doctest はこの辞書の浅いコピーを生成するので、例題は白紙の状態からスタートします。デフォルトの場合や `None` を指定した場合、新たな空の辞書になります。

オプション引数 *extraglobs* には辞書を指定します。この辞書は、例題を実行する際にグローバル変数にマージされます。マージは `dict.update()` のように振舞います: *globs* と *extraglobs* との間に同じキー値がある場合、両者を合わせた辞書中には *extraglobs* の方の値が入ります。この仕様は、パラメタ付きで doctest を実行するという、やや進んだ機能です。例えば、一般的な名前を使って基底クラス向けに doctest を書いておき、その後で辞書で一般的な名前からテストしたいサブクラスへの対応付けを行う辞書を *extraglobs* に渡して、様々なサブクラスをテストできます。

オプション引数 *verbose* が真の場合、様々な情報を出力します。偽の場合にはテストの失敗だけを報告します。デフォルトの場合や `None` を指定した場合、`sys.argv` に `-v` を指定しない限りこの値は真になりません。

オプション引数 *report* が真の場合、テストの最後にサマリを出力します。それ以外の場合には何も出力しません。verbose モードの場合、サマリには詳細な情報を出力しますが、そうでない場合にはサマリはとても簡潔になります (実際には、全てのテストが成功した場合には何も出力しません)。

オプション引数 *optionflags* は、各オプションフラグの論理和をとった値を指定します。23.2.6 節を参照してください。

オプション引数 *raise_on_error* の値はデフォルトでは偽です。真にすると、最初のテスト失敗や予期しない例外が起きたときに例外を送出します。このオプションを使うと、失敗の原因を検死デバッグ (post-mortem debug) できます。デフォルトの動作では、例題の実行を継続します。

オプション引数 *parser* には、`DocTestParser` (またはそのサブクラス) を指定します。このクラスはファイルから例題を抽出するために使われます。デフォルトでは通常のパーザ (`DocTestParser()`) です。

オプション引数 *encoding* にはファイルをユニコードに変換する際に使われるエンコーディングを指定します。

2.4 で追加された仕様です。

2.5 で変更された仕様: *encoding* パラメタが追加されました

testmod([*m*][, *name*][, *globs*][, *verbose*][, *report*][, *optionflags*][, *extraglobs*][, *raise_on_error*][, *exclude_empty*])

引数は全てオプションで、*m* 以外の引数はキーワード引数として指定せねばなりません。

モジュール *m* (*m* を指定しないか `None` にした場合には `__main__`) から到達可能な関数およびクラスの docstring 内にある例題をテストします。*m*`.__doc__` 内の例題からテストを開始します。

また、辞書 *m*`.__test__` が存在し、`None` でない場合、この辞書から到達できる例題もテストします。*m*`.__test__` は、(文字列の) 名前から関数、クラスおよび文字列への対応付けを行っています。関数およびクラスの場合には、その docstring 内から例題を検索します。文字列の場合には、docstring と同じようにして例題の検索を直接実行します。

モジュール *m* に属するオブジェクトにつけられた docstrings のみを検索します。

'(*failure_count*, *test_count*)' を返します。

オプション引数 *name* には、モジュールの名前を指定します。デフォルトの場合や `None` を指定した場合には、*m*`.__name__` を使います。

オプション引数 *exclude_empty* はデフォルトでは偽になっています。この値を真にすると、doctest を持たないオブジェクトを考慮から外します。デフォルトの設定は依存のバージョンとの互換性を考えたハックであり、`doctest.master.summarize()` と `testmod()` を合わせて利用しているよう

なコードでも、テスト例題を持たないオブジェクトから出力を得るようにしています。新たに追加された DocTestFinder のコンストラクタの *exclude_empty* はデフォルトで真になります。

オプション引数 *extraglobs*, *verbose*, *report*, *optionflags*, *raise_on_error*, および *globs* は上で説明した testfile() の引数と同じです。ただし、*globs* のデフォルト値は *m*.__dict__ になります。

2.3 で変更された仕様: *optionflags* パラメタを追加しました

2.4 で変更された仕様: *extraglobs*, *raise_on_error* および *exclude_empty* パラメタを追加しました

2.5 で変更された仕様: オプション引数 *isprivate* は、2.4 では非推奨でしたが、廃止されました

単一のオブジェクトに関連付けられた doctest を実行するための関数もあります。この関数は以前のバージョンとの互換性のために提供されています。この関数を撤廃する予定はありませんが、役に立つことはほとんどありません:

run_docstring_examples(*f, globs*[, *verbose*][, *name*][, *compileflags*][, *optionflags*])

オブジェクト *f* に関連付けられた例題をテストします。*f* はモジュール、関数、またはクラスオブジェクトです。

引数 *globs* に辞書を指定すると、その浅いコピーを実行コンテキストに使います。

オプション引数 *name* はテスト失敗時のメッセージに使われます。デフォルトの値は NoName です。

オプション引数 *verbose* の値を真にすると、テストが失敗しなくても出力を生成します。デフォルトでは、例題のテストに失敗したときのみ出力を生成します。

オプション引数 *compileflags* には、例題を実行するときに Python バイトコードコンパイラが使うフラグを指定します。デフォルトの場合や None を指定した場合、フラグは *globs* 内にある future 機能セットに対応したものになります。

オプション引数 *optionflags* は、上で述べた testfile() と同様の働きをします。

23.2.8 単位テスト API

doctest 化したモジュールのコレクションが増えるにつれ、全ての doctest をシステマティックに実行したいと思うようになるはずです。 Python 2.4 以前の doctest には Tester というほとんどドキュメント化されていないクラスがあり、複数のモジュールの doctest を統合する初歩的な手段を提供していました。Tester は非力であり、実際のところ、もっときちんとした Python のテストフレームワークが unittest モジュールで構築されており、複数のソースコードからのテストを統合する柔軟な方法を提供しています。そこで Python 2.4 では doctest の Tester クラスを撤廃し、モジュールや doctest の入ったテキストファイルから unittest テストスイートを作成できるような二つの関数を doctest 側で提供するようにしました。こうしたテストスイートは、unittest のテストランナを使って実行できます:

```
import unittest
import doctest
import my_module_with_doctests, and_another

suite = unittest.TestSuite()
for mod in my_module_with_doctests, and_another:
    suite.addTest(doctest.DocTestSuite(mod))
runner = unittest.TextTestRunner()
runner.run(suite)
```

doctest の入ったテキストファイルやモジュールから unittest.TestSuite インスタンスを生成するための主な関数は二つあります:

DocFileSuite(*[module_relative][, package][, setUp][, tearDown][, globs][, optionflags][, parser][, encoding]*)

単一または複数のテキストファイルに入っている doctest 形式のテストを、`unittest.TestSuite` インスタンスに変換します。

この関数の返す `unittest.TestSuite` インスタンスは、unittest フレームワークで動作させ、各ファイルの例題を対話的に実行するためのものです。ファイル内の何らかの例題の実行に失敗すると、この関数で生成した単位テストは失敗し、該当するテストの入っているファイルの名前と、(場合によりだいたいの) 行番号の入った `failureException` 例外を送出します。

関数には、テストを行いたい一つまたは複数のファイルへのパスを (文字列で) 渡します。

`DocFileSuite` には、キーワード引数でオプションを指定できます:

オプション引数 *module_relative* は *paths* に指定したファイル名をどのように解釈するかを指定します:

- *module_relative* が `True` (デフォルト) の場合、*filename* は OS に依存しないモジュールの相対パスになります。デフォルトでは、このパスは関数 `testfile` を呼び出しているモジュールからの相対パスになります; ただし、*package* 引数を指定した場合には、パッケージからの相対になります。OS への依存性を除くため、*filename* ではパスを分割する文字に / を使わねばならず、絶対パスにしてはなりません (パス文字列を / で始めてはなりません)。
- *module_relative* が `False` の場合、*filename* は OS 依存のパスを示します。パスは絶対パスでも相対パスでもかまいません; 相対パスにした場合、現在の作業ディレクトリを基準に解決します。

オプション引数 *package* には、 Python パッケージを指定するか、モジュール相対のファイル名の場合には相対の基準ディレクトリとなる Python パッケージの名前を指定します。パッケージを指定しない倍、関数を呼び出しているモジュールのディレクトリを相対の基準ディレクトリとして使います。*module_relative* を `False` に指定している場合、*package* を指定するとエラーになります。

オプション引数 *setUp* には、テストスイートのセットアップに使う関数を指定します。この関数は、各ファイルのテストを実行する前に呼び出されます。 *setUp* 関数は `DocTest` オブジェクトに引き渡されます。*setUp* は *globs* 属性を介してテストのグローバル変数にアクセスできます。

オプション引数 *tearDown* には、テストを解体 (tear-down) するための関数を指定します。この関数は、各ファイルのテストの実行を終了するたびに呼び出されます。*tearDown* 関数は `DocTest` オブジェクトに引き渡されます。*tearDown* はは *globs* 属性を介してテストのグローバル変数にアクセスできます。

オプション引数 *globs* は辞書で、テストのグローバル変数の初期値が入ります。この辞書は各テストごとに新たにコピーして使われます。デフォルトでは *globs* は空の新たな辞書です。

オプション引数 *optionflags* には、テストを実行する際にデフォルトで適用される doctest オプションを OR で結合して指定します。23.2.6 節を参照してください。結果レポートに関するオプションの指定する上手いやり方は下記の `set_unittest_reportflags()` の説明を参照してください。

オプション引数 *parser* には、ファイルからテストを抽出するために使う `DocTestParser` (またはサブクラス) を指定します。デフォルトは通常のパーザ (`DocTestParser()`) です。

オプション引数 *encoding* にはファイルをユニコードに変換する際に使われるエンコーディングを指定します。

2.4 で追加された仕様です。

2.5 で変更された仕様: グローバル変数 `__file__` が追加され `DocFileSuite()` を使ってテキストファイルから読み込まれた doctest に提供されます

2.5 で変更された仕様: *encoding* パラメタが追加されました

DocTestSuite([*module*][, *globs*][, *extraglobs*][, *test_finder*][, *setUp*][, *tearDown*][, *checker*])

doctest のテストを unittest.TestSuite に変換します。

この関数の返す unittest.TestSuite インスタンスは、unittest フレームワークで動作させ、モジュール内の各 doctest を実行するためのものです。何らかの doctest の実行に失敗すると、この関数で生成した単位テストは失敗し、該当するテストの入っているファイルの名前と、(場合によりだいたいの) 行番号の入った failureException 例外を送出します。

オプション引数 *module* には、テストしたいモジュールの名前を指定します。*module* にはモジュールオブジェクトまたは (ドット表記の) モジュール名を指定できます。*module* を指定しない場合、この関数を呼び出しているモジュールになります。

オプション引数 *globs* は辞書で、テストのグローバル変数の初期値が入ります。この辞書は各テストごとに新たにコピーして使われます。デフォルトでは *glob* は空の新たな辞書です。

オプション引数 *extraglobs* には追加のグローバル変数セットを指定します。この変数セットは *globs* に統合されます。デフォルトでは、追加のグローバル変数はありません。

オプション引数 *test_finder* は、モジュールから doctest を抽出するための DocTestFinder オブジェクト (またはその代用となるオブジェクト) です。

オプション引数 *setUp*、*tearDown*、および *optionflags* は上の DocFileSuite() と同じです。

2.3 で追加された仕様です。

2.4 で変更された仕様: *globs*, *extraglobs*, *test_finder*, *setUp*, *tearDown*, および *optionflags* パラメタを追加しました。また、この関数は doctest の検索に testmod() と同じテクニックを使うようになりました

DocTestSuite() は水面下では doctest.DocTestCase インスタンスから unittest.TestSuite を作成しており、DocTestCase は unittest.TestCase のサブクラスになっています。DocTestCase についてはここでは説明しません (これは内部実装上の詳細だからです) が、そのコードを調べてみれば、unittest の組み込みの詳細に関する疑問を解決できるはずです。

同様に、DocFileSuite() は doctest.DocFileCase インスタンスから unittest.TestSuite を作成し、DocFileCase は DocTestCase のサブクラスになっています。これにははっきりとした訳があります: doctest 関数を自分で実行する場合、オプションフラグを doctest 関数に渡すことで、doctest のオプションを直接操作できます。しかしながら、unittest フレームワークを書いている場合には、いつどのようにテストを動作させるかを unittest が完全に制御してしまいます。フレームワークの作者はたいてい、doctest のレポートオプションを (コマンドラインオプションで指定するなどして) 操作したいと考えますが、unittest を介して doctest のテストランナにオプションを渡す方法は存在しないのです。

このため、doctest では、以下の関数を使って、unittest サポートに特化したレポートフラグ表記方法もサポートしています:

set_unittest_reportflags(*flags*)

doctest のレポートフラグをセットします。

引数 *flags* にはオプションフラグを OR で結合して渡します。23.2.6 節を参照してください。「レポートフラグ」しか使えません。

この関数で設定した内容はモジュール全体にわたる物であり、関数呼び出し以後に unittest モジュールから実行される全ての doctest に影響します: DocTestCase の runTest() メソッドは、DocTestCase インスタンスが作成された際に、現在のテストケースに指定されたオプションフラグを見に行きます。レポートフラグが指定されていない場合 (通常の場合で、望ましいケースです)、doctest の unittest レポートフラグが OR で結合され、doctest を実行するために作成される

DocTestRunner インスタンスに渡されます。DocTestCase インスタンスを構築する際に何らかのレポートフラグが指定されていた場合、doctest の unittest レポートフラグは無視されます。

この関数は、関数を呼び出す前に有効になっていた unittest レポートフラグの値を返します。

2.4 で追加された仕様です。

23.2.9 拡張 API

基本 API は、 doctest を使いやすくするための簡単なラッパであり、柔軟性があってほとんどのユーザの必要を満たしています; とはいえ、もっとテストをきめ細かに制御したい場合や、 doctest の機能を拡張したい場合、拡張 API (advanced API) を使わねばなりません。

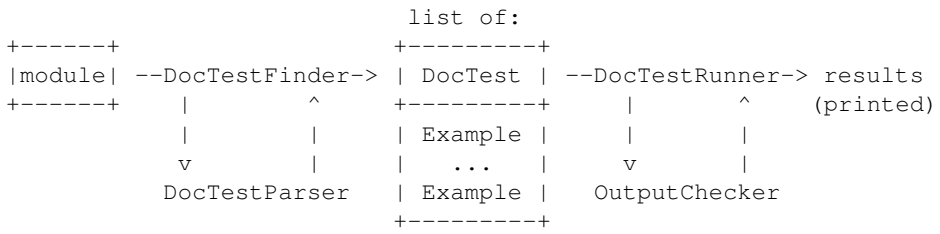
拡張 API は、doctest ケースから抽出した対話モードでの例題を記憶するための二つのコンテナクラスを中心に構成されています:

- Example: 一つの Python 実行文と、その予想出力をペアにしたもの。
- DocTest: Example の集まり。通常一つの docstring やテキストファイルから抽出されます。

その他に、 doctest の例題を検索、パーズ、実行、チェックするための処理クラスが以下のように定義されています:

- DocTestFinder: 与えられたモジュールから全ての docstring を検索し、対話モードでの例題が入った各 docstring から DocTestParser を使って DocTest を生成します。
- DocTestParser: (オブジェクトにつけられた docstring のような) 文字列から DocTest オブジェクトを生成します。
- DocTestRunner: DocTest 内の例題を実行し、OutputChecker を使って出力を検証します。
- OutputChecker: doctest 例題から実際に出力された結果を予想出力と比較し、両者が一致するか判別します。

これらの処理クラスの関係を図にまとめると、以下のようになります:



DocTest オブジェクト

class DocTest(*examples, globs, name, filename, lineno, docstring*)

単一の名前空間内で実行される doctest 例題の集まりです。コンストラクタの引数は DocTest インスタンス中の同名のメンバ変数の初期化に使われます。 2.4 で追加された仕様です。

DocTest では、以下のメンバ変数を定義しています。これらの変数はコンストラクタで初期化されます。直接変更してはなりません。

examples
対話モードにおける例題それぞれをエンコードしていて、テストで実行される、Example オブジェクトからなるリストです。

globs
例題を実行する名前空間 (いわゆるグローバル変数) です。このメンバは、名前から値への対応付けを行っている辞書です。例題が名前空間に対して (新たな変数をバインドするなど) 何らかの変更を行った場合、globs への反映はテストの実行後に起こります。

name
DocTest を識別する名前の文字列です。通常、この値はテストを取り出したオブジェクトかファイルの名前になります。

filename
DocTest を取り出したファイルの名前です; ファイル名が未知の場合や DocTest をファイルから取り出したのでない場合には None になります。

lineno
filename 中で DocTest のテスト例題が始まっている行の行番号です。行番号は、ファイルの先頭をゼロとして数えます。

docstring
テストを取り出した docstring 自体を現す文字列です。docstring 文字列を得られない場合や、文字列からテスト例題を取り出したのでない場合には None になります。

Example オブジェクト

class Example (*source, want*[, *exc_msg*][, *lineno*][, *indent*][, *options*])
ひとつの Python 文と、それに対する予想出力からなる、単一の対話的モードの例題です。コンストラクタの引数は Example インスタンス中の同名のメンバ変数の初期化に使われます。 2.4 で追加された仕様です。

Example では、以下のメンバ変数を定義しています。これらの変数はコンストラクタで初期化されます。直接変更してはなりません。

source
例題のソースコードが入った文字列です。ソースコードは単一の Python で、末尾は常に改行です。コンストラクタは必要に応じて改行を追加します。

want
例題のソースコードを実行した際の予想出力 (標準出力と、例外が生じた場合にはトレースバック) です。want の末尾は、予想出力が全くない場合を除いて常に改行になります。予想出力がない場合には空文字列になります。コンストラクタは必要に応じて改行を追加します。

exc_msg
例題が例外を生成すると予想される場合の例外メッセージです。例外を送出しない場合には None です。この例外メッセージは、traceback.format_exception_only() の戻り値と比較されます。 値が None でない限り、exc_msg は改行で終わっていなければなりません; コンストラクタは必要に応じて改行を追加します。

lineno
この例題の入っている文字列中における、例題の実行文のある行のの行番号です。行番号は文字列の先頭をゼロとして数えます。

indent
例題の入っている文字列のインデント、すなわち例題の最初のプロンプトより前にある空白文字の数

です。

options
オプションフラグを True または False に対応付けている辞書です。例題に対するデフォルトオプションを上書きするために用いられます。この辞書に入っていないオプションフラグはデフォルトの状態 (DocTestrunner の optionflags の内容) のままになります。

DocTestFinder オブジェクト

class DocTestFinder([*verbose*][, *parser*][, *recurse*][, *exclude_empty*])
与えられたオブジェクトについて、その docstring か、そのオブジェクトに入っているオブジェクトの docstring から DocTest を抽出する処理クラスです。現在のところ、モジュール、関数、クラス、メソッド、静的メソッド、クラスメソッド、プロパティから DocTest を抽出できます。

オプション引数 *verbose* を使うと、抽出処理の対象となるオブジェクトを表示できます。デフォルトは False (出力をおこなわない) です。

オプション引数 *parser* には、 docstring から DocTest を抽出するのに使う DocTestParser オブジェクト (またはその代替となるオブジェクト) を指定します。

オプション引数 *recurse* が偽の場合、DocTestFinder.find() は与えられたオブジェクトだけを調べ、そのオブジェクトに入っている他のオブジェクトを調べません。

オプション引数 *exclude_empty* が偽の場合、DocTestFinder.find() は空の docstring を持つオブジェクトもテスト対象に含めます。

2.4 で追加された仕様です。

DocTestFinder では以下のメソッドを定義しています:

find(*obj*[, *name*][, *module*][, *globs*][, *extraglobs*])
obj または *obj* 内に入っているオブジェクトの docstring 中で定義されている DocTest のリストを返します。

オプション引数 *name* には、オブジェクトの名前を指定します。この名前は、関数が返す DocTest の名前になります。*name* を指定しない場合、*obj*.__name__ を使います。

オプションのパラメタ *module* は、指定したオブジェクトを収めているモジュールを指定します。*module* を指定しないか、None を指定した場合には、正しいモジュールを自動的に決定しようと試みます。オブジェクトのモジュールは以下のような役割を果たします:

- *globs* を指定していない場合、オブジェクトのモジュールはデフォルトの名前空間になります。
- 他のモジュールから import されたオブジェクトに対して DocTestFinder が DocTest を抽出するのを避けるために使います (*module* 由来でないオブジェクトを無視します)。
- オブジェクトの入っているファイル名を調べるために使います。
- オブジェクトがファイル内の何行目にあるかを調べる手助けにします。

module が False の場合には、モジュールの検索を試みません。これは正確さを欠くような使い方で、通常 doctest 自体のテストにしかつかいません。*module* が False の場合、 または *module* が None で自動的に的確なモジュールを見つけ出せない場合には、全てのオブジェクトは (non-existent) モジュールに属するとみなされ、そのオブジェクト内の全てのオブジェクトに対して (再帰的に) doctest の検索をおこないます。

各 DocTest のグローバル変数は、*globs* と *extraglobs* を合わせたもの (*extraglobs* 内のバインドが *globs* 内のバインドを上書きする) になります。各々の DocTest に対して、グローバル変数を表す辞

書の新たな浅いコピーを生成します。*globs* を指定しない場合に使われるのデフォルト値は、モジュールを指定していればそのモジュールの *__dict__* になり、指定していなければ `{}` になります。*extraglobs* を指定しない場合、デフォルトの値は `{}` になります。

DocTestParser オブジェクト

class `DocTestParser`()
対話モードの例題を文字列から抽出し、それを使って `DocTest` オブジェクトを生成するために使われる処理クラスです。 2.4 で追加された仕様です。

`DocTestParser` では以下のメソッドを定義しています:

`get_doctest`(*string, globs, name, filename, lineno*)
指定した文字列から全ての doctest 例題を抽出し、`DocTest` オブジェクト内に集めます。

globs, *name*, *filename*, および *lineno* は新たに作成される `DocTest` オブジェクトの属性になります。詳しくは `DocTest` のドキュメントを参照してください。

`get_examples`(*string*[, *name*])
指定した文字列から全ての doctest 例題を抽出し、`Example` オブジェクトからなるリストにして返します。各 `Example` の行番号はゼロから数えます。オプション引数 *name* はこの文字列につける名前で、エラーメッセージにしか使われません。

`parse`(*string*[, *name*])
指定した文字列を、例題とその間のテキストに分割し、例題を `Example` オブジェクトに変換し、`Example` と文字列からなるリストにして返します。各 `Example` の行番号はゼロから数えます。オプション引数 *name* はこの文字列につける名前で、エラーメッセージにしか使われません。

DocTestRunner オブジェクト

class `DocTestRunner`([*checker*][, *verbose*][, *optionflags*])
`DocTest` 内の対話モード例題を実行し、検証する際に用いられる処理クラスです。

予想出力と実際の出力との比較は `OutputChecker` で行います。比較は様々なオプションフラグを使ってカスタマイズできます; 詳しくは 23.2.6 を参照してください。オプションフラグでは不十分な場合、コンストラクタに `OutputChecker` のサブクラスを渡して比較方法をカスタマイズできます。

テストランナの表示出力の制御には二つの方法があります。一つ目は、`TestRunner.run()` に出力用の関数を渡すというものです。この関数は、表示すべき文字列を引数にして呼び出されます。デフォルトは `sys.stdout.write` です。出力を取り込んで処理するだけでは不十分な場合、`DocTestRunner` をサブクラス化し、`report_start`, `report_success`, `report_unexpected_exception`, および `report_failure` をオーバライドすればカスタマイズできます。

オプションのキーワード引数 *checker* には、`OutputChecker` オブジェクト (またはその代用品) を指定します。このオブジェクトは doctest 例題の予想出力と実際の出力との比較を行う際に使われます。

オプションのキーワード引数 *verbose* は、`DocTestRunner` の出すメッセージの冗長性を制御します。*verbose* が `True` の場合、各例題を実行するつど、その例題についての情報を出力します。*verbose* が `False` の場合、テストの失敗だけを出力します。*verbose* を指定しない場合や `None` を指定した場合、コマンドラインスイッチ **-v** を使った場合にのみ *verbose* 出力を適用します。

オプションのキーワード引数 *optionflags* を使うと、テストランナが予想出力と実際の出力を比較する方法や、テストの失敗を表示する方法を制御できます。詳しくは 23.2.6 節を参照してください。

2.4 で追加された仕様です。

`DocTestRunner` では、以下のメソッドを定義しています:

report_start(*out, test, example*)
テストランナが例題を処理しようとしているときにレポートを出力します。DocTestRunner の出力をサブクラスでカスタマイズできるようにするためのメソッドです。直接呼び出してはなりません。

example は処理する例題です。*test* は *example* の入っているテストです。*out* は出力用の関数で、DocTestRunner.run() に渡されます。

report_success(*out, test, example, got*)
与えられた例題が正しく動作したことを報告します。このメソッドは DocTestRunner のサブクラスで出力をカスタマイズできるようにするために提供されています; 直接呼び出してはなりません。

example は処理する例題です。*got* は例題から実際に得られた出力です。*test* は *example* の入っているテストです。*out* は出力用の関数で、DocTestRunner.run() に渡されます。

report_failure(*out, test, example, got*)
与えられた例題が正しく動作しなかったことを報告します。このメソッドは DocTestRunner のサブクラスで出力をカスタマイズできるようにするために提供されています; 直接呼び出してはなりません。

example は処理する例題です。*got* は例題から実際に得られた出力です。*test* は *example* の入っているテストです。*out* は出力用の関数で、DocTestRunner.run() に渡されます。

report_unexpected_exception(*out, test, example, exc_info*)
与えられた例題が予想とは違う例外を送出したことを報告します。このメソッドは DocTestRunner のサブクラスで出力をカスタマイズできるようにするために提供されています; 直接呼び出してはなりません。

example は処理する例題です。*exc_info* には予期せず送出された例外の情報を入れたタプル (sys.exc_info() の返す内容) になります。*test* は *example* の入っているテストです。*out* は出力用の関数で、DocTestRunner.run() に渡されます。

run(*test*[, *compileflags*][, *out*][, *clear_globs*])
test 内の例題 (DocTest オブジェクト) を実行し、その結果を出力用の関数 *out* を使って表示します。

例題は名前空間 test.globs の下で実行されます。*clear_globs* が真 (デフォルト) の場合、名前空間はテストの実行後に消去され、ガベージコレクションをうながします。テストの実行完了後にその内容を調べたければ、*clear_globs* を False にしてください。

compileflags には、例題を実行する際に Python コンパイラに適用するフラグセットを指定します。*compileflags* を指定しない場合、デフォルト値は *globs* で適用されている future-import フラグセットになります。

各例題の出力は DocTestRunner の出力チェッカで検査され、その結果は DocTestRunner.report_*. メソッドで書式化されます。

summarize([*verbose*])
この DocTestRunner が実行した全てのテストケースのサマリを出力し、タプル '(*failure_count*, *test_count*)' を返します。

オプションの *verbose* 引数を使うと、どのくらいサマリを詳しくするかを制御できます。冗長度を指定しない場合、DocTestRunner 自体の冗長度を使います。

OutputChecker オブジェクト

class OutputChecker()
doctest 例題を実際に実行したときの出力が予想出力と一致するかどうかをチェックするために使われるクラスです。 OutputChecker では、与えられた二つの出力を比較して、一致する場合には真を

返す `check_output` と、二つの出力間の違いを説明する文字列を返す `output_difference` の、二つのメソッドがあります。 2.4 で追加された仕様です。

`OutputChecker` では以下のメソッドを定義しています:

`check_output` (*want, got, optionflags*)
例題から実際に得られた出力 (*got*) と、予想出力 (*want*) が一致する場合にのみ `True` を返します。二つの文字列が全く同一の場合には常に一致するとみなしますが、テストランナの使っているオプションフラグにより、厳密には同じ内容になっていなくても一致するとみなす場合もあります。オプションフラグについての詳しい情報は 23.2.6 節を参照してください。

`output_difference` (*want, got, optionflags*)
与えられた例題の予想出力 (*want*) と、実際に得られた出力 (*got*) の間の差異を解説している文字列を返します。*optionflags* は *want* と *got* を比較する際に使われるオプションフラグのセットです。

23.2.10 デバッグ

`doctest` では、doctest 例題をデバッグするメカニズムをいくつか提供しています:

- doctest を実行可能な Python プログラムに変換し、 Python デバッガ `pdb` で実行できるようにするための関数がいくつかあります。
- `DocTestRunner` のサブクラス `DebugRunner` クラスがあります。このクラスは、最初に失敗した例題に対して例外を送出します。例外には例題に関する情報が入っています。この情報は例題の検視デバッグに利用できます。
- `DocTestSuite()` の生成する `unittest` テストケースは、`debug()` メソッドをサポートしています。`debug()` は `unittest.TestCase` で定義されています。
- `pdb.set_trace()` を doctest 例題の中で呼び出しておけば、その行が実行されたときに Python デバッガが組み込まれます。デバッガを組み込んだあとは、変数の現在の値などを調べられます。たとえば、以下のようなモジュールレベルの docstring の入ったファイル 'a.py' があるとします:

```
"""
>>> def f(x):
...     g(x*2)
>>> def g(x):
...     print x+3
...     import pdb; pdb.set_trace()
>>> f(3)
9
"""
```

対話セッションは以下のようになるでしょう:

```
>>> import a, doctest
>>> doctest.testmod(a)
--Return--
> <doctest a[1]>(3)g()->None
-> import pdb; pdb.set_trace()
(Pdb) list
  1     def g(x):
  2         print x+3
  3  ->     import pdb; pdb.set_trace()
[EOF]
(Pdb) print x
6
(Pdb) step
--Return--
> <doctest a[0]>(2)f()->None
-> g(x*2)
(Pdb) list
  1     def f(x):
  2  ->     g(x*2)
[EOF]
(Pdb) print x
3
(Pdb) step
--Return--
> <doctest a[2]>(1)?()->None
-> f(3)
(Pdb) cont
(0, 3)
>>>
```

2.4 で変更された仕様: `pdb.set_trace()` を doctest の中で有効に使えるようになりました

以下は、doctest を Python コードに変換して、できたコードをデバッガ下で実行できるようにするための関数です:

script_from_examples(*s*)

例題の入ったテキストをスクリプトに変換します。

引数 *s* は doctest 例題の入った文字列です。この文字列は Python スクリプトに変換され、その中では *s* の doctest 例題が通常のコードに、それ以外は Python のコメント文になります。生成したスクリプトを文字列で返します。例えば、

```
import doctest
print doctest.script_from_examples(r"""
    Set x and y to 1 and 2.
    >>> x, y = 1, 2

    Print their sum:
    >>> print x+y
    3
""")
```

は、

```
# Set x and y to 1 and 2.
x, y = 1, 2
#
# Print their sum:
print x+y
# Expected:
## 3
```

になります。

この関数は他の関数 (下記参照) から使われていまるが、対話セッションを Python スクリプトに変換したいような場合にも便利でしょう。

2.4 で追加された仕様です。

testsource(*module, name*)

あるオブジェクトの doctest をスクリプトに変換します。

引数 *module* はモジュールオブジェクトか、対象の doctest を持つオブジェクトの入ったモジュールのドット表記名です。引数 *name* は対象の doctest を持つオブジェクトの (モジュール内の) 名前です。対象オブジェクトの docstring を上の `script_from_examples()` で説明した方法で Python スクリプトに変換してできた文字列を返しますます。例えば、'a.py' モジュールのトップレベルに関数 `f()` がある場合、以下のコード

```
import a, doctest
print doctest.testsource(a, "a.f")
```

を実行すると、`f()` の docstring から doctest をコードに変換し、それ以外をコメントにしたスクリプトを出力します。

2.3 で追加された仕様です。

debug(*module, name*[, *pm*])

オブジェクトの持つ doctest をデバッグします。

module および *name* 引数は上の `testsource()` と同じです。指定したオブジェクトの docstring から合成された Python スクリプトは一時ファイルに書き出され、その後 Python デバッガ `pdb` の制御下で実行されます。

ローカルおよびグローバルの実行コンテキストには、*module*`.__dict__` の浅いコピーが使われます。

オプション引数 *pm* は、検死デバッグを行うかどうかを指定します。*pm* が真の場合、スクリプトファイルは直接実行され、スクリプトが送出した例外が処理されないまま終了した場合にのみデバッガが立ち入ります。その場合、`pdb.post_mortem()` によって検死デバッグを起動し、処理されなかった例外から得られたトレースバックオブジェクトを渡します。*pm* を指定しないか値を偽にした場合、`pdb.run()` に適切な `execfile()` 呼び出しを渡して、最初からデバッガの下でスクリプトを実行します。

2.3 で追加された仕様です。

2.4 で変更された仕様: 引数 *pm* を追加しました

debug_src(*src*[, *pm*][, *globs*])

文字列中の doctest をデバッグします。

上の `debug()` に似ていますが、doctest の入った文字列は *src* 引数で直接指定します。

オプション引数 *pm* は上の `debug()` と同じ意味です。

オプション引数 *globs* には、ローカルおよびグローバルな実行コンテキストの両方に使われる辞書を指定します。*globs* を指定しない場合や `None` にした場合、空の辞書を使います。辞書を指定した場合、実際の実行コンテキストには浅いコピーが使われます。

2.4 で追加された仕様です。

`DebugRunner` クラス自体や `DebugRunner` クラスが送出する特殊な例外は、テストフレームワークの作者にとって非常に興味のあるところで The `DebugRunner` class, and the special exceptions it may raise, are of most interest to testing framework authors, and will only be sketched here.

詳しくはソースコード、とりわけ `DebugRunner` の docstring (それ自体 doctest ですよ!) を参照してください。

class `DebugRunner` (*[checker][, verbose][, optionflags]*)

テストの失敗に遭遇するとすぐに例外を送出するようになっている `DocTestRunner` のサブクラスです。予期しない例外が生じると、`UnexpectedException` 例外を送出します。この例外には、テスト、例題、もともと送出された例外が入っています。予想出力と実際出力が一致しないために失敗した場合には、`DocTestFailure` 例外を送出します。この例外には、テスト、例題、実際の出力が入っています。

コンストラクタのパラメタやメソッドについては、23.2.9 節の `DocTestRunner` のドキュメントを参照してください。

`DebugRunner` インスタンスの送出する例外には以下の二つがあります:

exception `DocTestFailure` (*test, example, got*)

doctest 例題の実際の出力が予想出力と一致しなかったことを示すために `DocTestRunner` が送出する例外です。コンストラクタの引数は、インスタンスの同名のメンバ変数を初期化するために使われます。

`DocTestFailure` では以下のメンバ変数を定義しています:

`test`

例題が失敗した時に実行されていた `DocTest` オブジェクトです。

`example`

失敗した `Example` オブジェクトです。

`got`

例題の実際の出力です。

exception `UnexpectedException` (*test, example, exc_info*)

doctest 例題が予期しない例外を送出したことを示すために `DocTestRunner` が送出する例外です。コンストラクタの引数は、インスタンスの同名のメンバ変数を初期化するために使われます。

`UnexpectedException` では以下のメンバ変数を定義しています:

`test`

例題が失敗した時に実行されていた `DocTest` オブジェクトです。

`example`

失敗した `Example` オブジェクトです。

`exc_info`

予期しない例外についての情報の入ったタプルで、`sys.exc_info()` が返すのと同じものです。

23.2.11 提言

冒頭でも触れたように、`doctest` は、

1. docstring 内の例題をチェックする、

2. 回帰テストを行う、

3. 実行可能なドキュメント/読めるテストの実現、

という三つの主な用途を持つようになりました。これらの用途にはそれぞれ違った要求があるので、区別して考えるのが重要です。特に、 docstring を曖昧なテストケースに埋もれさせてしまうとドキュメントとしては最悪です。

docstring の例は注意深く作成してください。 doctest の作成にはコツがあり、きちんと学ぶ必要があります — 最初はすんなりできないでしょう。例題は、ドキュメントに紛れ無しの価値を与えます。よい例がたくさんの言葉に値することは多々あります。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という形で、きっと見返りを得るでしょう。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。

説明テキストの作成をけちらなければ、`doctest` は回帰テストの優れたツールにもなり得ます。説明文と例題を交互に記述していけば、実際に何をどうしてテストしているのかもっと簡単に把握できるようになるでしょう。もちろん、コードベースのテストに詳しくコメントを入れるのも手ですが、そんなことをするプログラマはほとんどいません。多くの人々が、`doctest` のアプローチをとった方がきれいにテストを書けると気づいています。おそらく、これは単にコード中にコメントを書くのが少し面倒だからという理由でしょう。私はもう少しうがった見方もしています: doctest ベースのテストを書くときの自然な態度は、自分のソフトウェアのよい点を説明しようとして、例題を使って説明しようとするときの態度そのものだからだ、という理由です。それゆえに、テストファイルは自然と単純な機能の解説から始め、論理的により複雑で境界条件的なケースに進むような形になります。結果的に、一見ランダムに見えるような個別の機能をテストしている個別の関数の集まりではなく、首尾一貫した説明ができるようになるのです。`doctest` によるテストの作成は全く別の取り組み方であり、テストと説明の区別をなくして、全く違う結果を生み出すのです。

回帰テストは特定のオブジェクトやファイルにまとめておくのがよいでしょう。回帰テストの組み方にはいくつか選択肢があります:

- テストケースを対話モードの例題にして入れたテキストファイルを書き、`testifle()` や `DocFileSuite()` を使ってそのファイルをテストします。この方法をお勧めします。最初から doctest を使うようにしている新たなプロジェクトでは、この方法が一番簡単です。

- `_regrtest_`*`topic`* という名前の関数を定義します。この関数には、あるトピックに対応するテストケースの入った docstring が一つだけ入っています。この関数はモジュールと同じファイルの中にも置けますし、別のテストファイルに分けてもかまいません。

- 回帰テストのトピックをテストケースの入った docstring に対応付けた辞書`__test__` 辞書を定義します。

23.2.12 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。

23.3 `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 は時に “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。

unittest 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。

このようなことを実現するために unittest 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。テストケースを作成する場合は、unittest が提供する TestCase クラスを基底クラスとして利用することができます。

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。

unittest 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 unittest 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、FunctionTestCase 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。unittest 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:

doctest モジュール (23.2 節):

Another test-support module with a very different flavor.

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.

23.3.1 基礎的な例

unittest モジュールには、テストの開発や実行の為の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.assert_(element in self.seq)

    def testsample(self):
        self.assertRaises(ValueError, random.sample, self.seq, 20)
        for element in random.sample(self.seq, 5):
            self.assert_(element in 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 assertEqual() 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.TestLoader().loadTestsFromTestCase(TestSequenceFunctions)
unittest.TextTestRunner(verbosity=2).run(suite)
```

変更後のスクリプトをインター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。

23.3.2 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。unittest 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、TestCase のサブクラスは runTest() をオーバライドし、必要なテスト処理を記述するだけで簡単に書くことができます:

```
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def runTest(self):
        widget = Widget('The widget')
        self.assertEqual(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。テストが失敗すると例外が送出され、unittest はテスト結果を *failure* とします。その他の例外は *error* となります。これによりどこに問題があるかが判ります。*failure* は間違った結果 (6 になるはず

が 5 だった) で発生します。*error* は間違ったコード (たとえば間違った関数呼び出しによる `TypeError`) で発生します。

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()` が成功したかどうかに従って `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、`unittest` 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')
```

unittest 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:

```
def suite():
    tests = ['testDefaultSize', 'testResize']

    return unittest.TestSuite(map(WidgetTestCase, tests))
```

一般的には、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ので、unittest ではテストスイートを作成して個々のテストで満たすプロセスを自動化するのに使う TestLoader を用意しています。たとえば、

```
suite = unittest.TestLoader().loadTestsFromTestCase(WidgetTestCase)
```

はWidgetTestCase.testDefaultSize()とWidgetTestCase.testResizeを走らせるテストスイートを作成します。TestLoaderは自動的にテストメソッドを識別するのに'test'というメソッド名の接頭辞を使います。

いろいろなテストケースが実行される順序は、テスト関数名を組み込み関数cmp()でソートして決定されます。

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('test_widget.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。

23.3.3 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unittestで実行しようとするため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、unittestでは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も行なえます。

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)
```

既存のテストスイートからの移行を容易にするため、unittest は AssertionError の送出でテストの失敗を示すような書き方もサポートしています。しかしながら、TestCase.fail*() および TestCase.assert*() メソッドを使って明確に書くことが推奨されています。unittest の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。

注意: FunctionTestCase を使って既存のテストを unittest ベースのテスト体系に変換することができますが、この方法は推奨されません。時間を掛けて TestCase のサブクラスに書き直した方が将来的なテストのリファクタリングが限りなく易しくなります。

23.3.4 クラスと関数

class TestCase ([*methodName*])

TestCase クラスのインスタンスは、unittest の世界における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。

それぞれの TestCase クラスのインスタンスはただ一つのテストメソッド、*methodName* という名のメソッドを実行します。既に次のような例を扱ったことを憶えているでしょうか。

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('testDefaultSize'))
    suite.addTest(WidgetTestCase('testResize'))
    return suite
```

ここでは、それぞれが一つずつのテストを実行するような WidgetTestCase の二つのインスタンスを作成しています。

methodName のデフォルトは'runTest' です。

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ためのインタフェースを備えています。TestSuite インスタンスを実行することはスイートの繰り返しを使って個々のテストを実行することと同じです。

引数 *tests* が与えられるならば、それはテストケースに亘る繰り返し可能オブジェクトまたは内部でスイートを組み立てるための他のテストスイートでなければなりません。後からテストケースやスイートをコレクションに付け加えるためのメソッドも提供されています。

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。このクラスは与えられたモジュールまたは TestCase のサブクラスの中から全てのテストをロードできます。

class TestResult()
このクラスはどのテストが成功しどのテストが失敗したかの情報を集積するのに使います。

defaultTestLoader
TestLoader のインスタンスで、共用することが目的です。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*]]])
実行結果を標準エラーに出力する、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、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。一番簡単なこの関数の使い方は、以下の行をテストスクリプトの最後に置くことです。

```
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 doctest に基づいたテストコードから 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。

23.3.5 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 (defaultTestCase() メソッドを呼んで) 生成して使用しますが run() の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

ガ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 *secondexpr* 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 *secondexpr* 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* で与えた小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。指定小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くなかった場合、テストは失敗し、*msg* で指定した説明か、None を返します。

assertNotAlmostEqual(*first, second*[, *places*[, *msg*]])

failIfAlmostEqual(*first, second*[, *places*[, *msg*]])

first と *second* を *places* で与えた小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。指定小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。

defaultTestResult()
このテストケースクラスで使われるテスト結果クラスのインスタンスを (もし run() メソッドに他の結果インスタンスが提供されないならば) 返します。

TestCase インスタンスに対しては、いつも TestResult のインスタンスですので、TestCase のサブクラスでは必要に応じてこのメソッドをオーバライドしてください。

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。

23.3.6 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*)
イテラブル *tests* 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。

このメソッドは *test* 上のイテレーションをしながらそれぞれの要素に addTest() を呼び出すのと等価です。

TestSuite クラスは TestCase と以下のメソッドを共有します:

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。

debug()
このスイートに関連づけられたテストを結果を収集せずに実行します。これによりテストで送出された例外は呼び出し元に伝わるようになり、デバッガの下でのテスト実行をサポートできるようになります。

countTestCases()
このテストオブジェクトによって表現されるテストの数を返します。これには個別のテストと下位のスイートも含まれます。

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。

23.3.7 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。

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
: 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 2 要素タプルからなるリスト。それぞれのタプルは予想外の例外を送出したテストに対応します。 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
: 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 2 要素タプルからなるリスト。それぞれのタプルは TestCase.fail*() や TestCase.assert*() メソッドを使って見つけ出した失敗に対応します。 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
: これまでに実行したテストの総数。

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。

stop()
: このメソッドを呼び出して TestResult の shouldStop 属性に True をセットすることで、実行中のテストは中断しなければならないというシグナルを送ることができます。TestRunner オブジェクトはこのフラグを尊重してそれ以上のテストを実行することなく復帰しなければなりません。

  たとえばこの機能は、ユーザのキーボード割り込みを受け取って TextTestRunner クラスがテストフレームワークを停止させるのに使えます。TestRunner の実装を提供する対話的なツールでも同じように使用することができます。

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。

  デフォルトの実装では単純にインスタンスの testRun カウンタをインクリメントします。

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。

  デフォルトの実装では何もしません。

addError(*test, err*)
: テスト *test* 実行中に、想定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。

  デフォルトの実装ではインスタンスの errors 属性に (*test*, *err*) を追加します。

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。

  デフォルトの実装ではインスタンスの failures 属性に (*test*, *err*) を追加します。

addSuccess(*test*)
: テストケース *test* が成功した場合に呼び出されます。

  デフォルトの実装では何もしません。

23.3.8 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 unittest.defaultTestLoader を共用インスタンスとして使用することができます。ただ、サブクラスや別のインスタンスを活用すると設定可能なプロパティをカスタマイズすることもできます。

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、テストケースクラス内のメソッド、TestSuite インスタンスまたは 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。このチェックはここで挙げた順番に行なわれます。すなわち、候補テストケースクラス内のメソッドは「呼び出し可能オブジェクト」としてではなく「テストケースクラス内のメソッド」として拾い出されます。

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* 中の名前で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。*testCaseClass* は TestCase のサブクラスでなければなりません。

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。

この値は getTestCaseNames() と全ての loadTestsFrom*() メソッドに影響を与えます。

sortTestMethodsUsing

getTestCaseNames() および全ての loadTestsFrom*() メソッド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。ソートを行なわないようにこの属性に None を指定することもできます。

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。

この値は全ての loadTestsFrom*() メソッドに影響を与えます。

23.4 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.regtest 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 (23.3 節):

PyUnit 回帰テストを書く。

doctest モジュール (23.2 節):

ドキュメンテーション文字列に埋め込まれたテスト。

23.4.1 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を、'test_' で始め、テスト対象となるモジュール名で終えることです。テストモジュール中のテストメソッドは名前を 'test_' で始めて、そのメソッドが何をテストしているかという説明で終えます。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必要です。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 `regrtest.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え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。

- テストがオペレーティングシステムの特定の状況に依存する場合、テストを開始する前に状況を確認してください。
- 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、て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

23.4.2 `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。

23.4.3 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を全て削除します。

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。

二十四章

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。事後解析デバッギングもサポートし、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せます。例えば:

```
python -m pdb myscript.py
```

スクリプトとして pdb を起動すると、デバッグ中のプログラムが異常終了した時に pdb が自動的に検死デバッグモードに入ります。検死デバッグ後 (またはプログラムの正常終了後) には、pdb はプログラムを再起動します。自動再起動を行った場合、 pdb の状態 (ブレークポイントなど) はそのまま維持されるので、たいていの場合、プログラム終了時にデバッガも終了させるよりも便利なはずです。 2.4 で追加された仕様: 検死デバッグ後の再起動機能が追加されました

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。それぞれが少しづ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' を使って一つづ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ギングに入ります。

pm()
: sys.last_traceback のトレースバックの事後解析デバッギングに入ります。

24.1 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。

第二引数を指定する場合、その値は式で、その評価値が真でなければブレークポイントは有効になりません。

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取り上げられる前に真と評価され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、ブレークポイントは無条件になります。

commands [*bpnumber*] ブレークポイントナンバー *bpnumber* にコマンドのリストを指定します。コマンドそのものはその後の行に続けます。'end' だけからなる行を入力することでコマンド群の終わりを示します。例を挙げます:

```
(Pdb) commands 1
(com) print some_variable
(com) end
(Pdb)
```

ブレークポイントからコマンドを取り除くには、commands のあとに end だけを続けます。つまり、コマンドを一つも指定しないようにします。

bpnumber 引数が指定されない場合、最後にセットされたブレークポイントを参照することになります。

ブレークポイントコマンドはプログラムを走らせ直すのに使えます。ただ continue コマンドや step、その他実行を再開するコマンドを使えば良いのです。

実行を再開するコマンド (現在のところ continue, step, next, return, jump, quit とそれらの省略形) によって、コマンドリストは終了するものと見なされます (コマンドにすぐ end が続いているかのように)。というのも実行を再開すれば (それが単純な next や step であっても) 別のブレークポイントに到達するかもしれないからです。そのブレークポイントにさらにコマンドリストがあれば、どちらのリストを実行すべきか状況が曖昧になります。

コマンドリストの中で 'silent' コマンドを使うと、ブレークポイントで停止したという通常のメッセージはプリントされません。この振る舞いは特定のメッセージを出して実行を続けるようなブレークポイントでは望ましいものでしょう。他のコマンドが何も画面出力をしなければ、そのブレークポイントに到達したというサインを見ないことになります。

2.5 で追加された仕様です。

s(tep) 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(ont(inue)) 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` 節の外にも飛ぶ事ができません。

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。引数が一つある場合は、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いては**いけません**。入れ替え可能なパラメータは '`%1`'、'`%2`' などで指し示され、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ガ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) デバッガを終了します。実行しているプログラムは中断されます。

24.2 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、 *frame*、*event* および *arg* を持ちます。*frame* は現在のスタックフレームです。*event* は文字列で、'call'、'line'、'return'、'exception'、'c_call'、'c_return' または'c_exception' です。*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。

'c_call' 拡張モジュールまたは組み込みの C 関数が呼び出されようとしています。*arg* は C 関数オブジェクトです。

'c_return' C 関数が処理を戻しました。*arg* は None です。

'c_exception' C 関数が例外を送出しました。*arg* は None です。

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。

二十五章

Python プロファイラ

執筆者 James Roskind [1]

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 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。

25.1 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。

Python 標準ライブラリは 3 つの異なるプロファイラを提供します。

1. `profile` はピュア Python モジュールで、このあとすぐ説明します。Copyright © 1994, by InfoSeek Corporation. 2.4 で変更された仕様: 組み込み関数やメソッド呼出しに費やされる時間も報告するようになりました

2. `cProfile` は C で書かれたモジュールで、少ないオーバーヘッドにより長く実行されるプログラム

[1] アップデートと LaTeX への変換は Guido van Rossum によるもの。さらに Python 2.5 の新しい `cProfile` モジュールの文書を統合するアップデートは Armin Rigo による。

のプロファイルに向きます。Brett Rosen と Ted Czotter が提供した lsprof に基づいています。 2.5 で追加された仕様です。

3. hotshot は C モジュールでプロファイル中のオーバーヘッドを極力小さくすることに焦点を絞っており、その代わりに後処理時間の長さというつけを払います。 2.5 で変更された仕様: 以前より意味のある結果が得られているはずです。かつては時間計測の中核部分に致命的なバグがありました

profile と cProfile の両モジュールは同じインタフェースを提供しているので、ほぼ取り替え可能です。 cProfile はずっと小さなオーバーヘッドで動きますが、まだ同じぐらいテストされたとは言えず、全てのシステムで使えるとは限らないでしょう。cProfile は実際には _lsprof 内部モジュールに被せられた互換性レイヤです。hotshot モジュールは特別な使い道のために取っておいてあります。

25.2 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 cProfile
cProfile.run('foo()')
```

(お使いのシステムで cProfile が使えないときは代わりに profile を使って下さい)

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 cProfile
cProfile.run('foo()', 'fooprof')
```

ファイル ‘cProfile.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 -m cProfile myscript.py
```

‘cProfile.py’ はオプションとしてコマンドライン引数を 2 つ受け取ります。

```
cProfile.py [-o output_file] [-s sort_order]
```

-s は標準出力 (つまり、 **-o** が与えられなかった場合) にのみ有効です。利用可能なソートの値は、 Stats のドキュメントをご覧ください。

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 cProfile.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。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フェース (cmd を使って実装) とヘルプ機能を備えています。

25.3 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。

25.4 リファレンス・マニュアル – profile と cProfile

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() (または cProfile.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 (*command*[, *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。

```
         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 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** な呼び出しとは、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び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び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。

`runctx` (*command, globals, locals*[, *filename*])

この関数は `run()` に似ていますが、*command* 文字列用にグローバル辞書とローカル辞書の引数を追加しています。

プロファイラ・データの分析は `Stats` クラスを使っておこないます。

注意: `Stats` クラスは `pstats` モジュールで定義されています。

class `Stats` (*filename*[, *stream=sys.stdout*[, ...]])

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。また別の出力ストリームをキーワード引数 `stream` で指定できます。

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` または `cProfile` 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**保証されておらず**、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。

2.5 で変更された仕様: *stream* 引数が追加されました

25.4.1 `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リは、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されます。

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() または cProfile.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 クラスおよび cProfile.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'	累積時間
'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。

後方互換性のため、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。引数をこの旧スタイルで指定した場合、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。デフォルトでは選択したキーに応じて昇順、降順が適切に選ばれることに注意してください。

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。各呼出し側関数についてそれぞれ一行ずつ表示されます。フォーマットは統計を作り出したプロファイラごとに微妙に異なります。

- profile を使った場合、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、便宜上右側の関数が消費した累積時間を繰り返したものです。
- cProfile を使った場合、各呼出し側関数は 3 つの数字の後に来ます。その 3 つとは、呼出しが何回おこなわれたか、呼出しの結果現在の関数内で費やされた合計時間および累積時間です。

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。

25.5 制限事項

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。これ以上の精度で

計測することはできません。仮に充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ます。

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。

この問題はオーバーヘッドの小さい `cProfile` よりも `profile` においてより重要です。そのため、`profile` 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。

25.6 キャリブレーション (補正)

`profile` のプロファイラは 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に、各イベントハンドリング時間から定数を引きます。デフォルトでこの定数の値は 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。[2]

[2] 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値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. 算出した補正値 (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値を適用する。
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値を指定する。
pr = profile.Profile(bias=your_computed_bias)
```

方法を選択したら、補正値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。

25.7 拡張 — プロファイラの改善

profile モジュールおよび cProfile 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、使用する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。

profile.Profile your_time_func() 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。

cProfile.Profile your_time_func() は単一の数値を返さなければなりません。もしこれが整数を返す関数ならば、2 番目の引数に時間単位当たりの実際の持続時間を指定してクラスのコンスタラクタを呼び出すことができます。たとえば、your_integer_time_func() が 1000 分の 1 秒単位で計測した時間を返すとすると、Profile インスタンスを次のように生成することができます。

```
pr = profile.Profile(your_integer_time_func, 0.001)
```

cProfile.Profile クラスはキャリブレーションができないので、自前のタイマ関数は注意を払って使う必要があり、またそれは可能な限り速くなければなりません。自前のタイマ関数で最高の結果

を得るには、_lsprof 内部モジュールの C ソースファイルにハードコードする必要があるかもしれません。

25.8 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がはるかに少なく済みます。

hotshot は C モジュールでプロファイル中のオーバーヘッドを極力小さくすることに焦点を絞っており、その代わりに後処理時間の長さというつけを払います。通常の使用法についてはこのモジュールではなく cProfile を使うことを推奨します。hotshot は保守されておらず、将来的には標準ライブラリから外されるかもしれません。

2.5 で変更された仕様: 以前より意味のある結果が得られているはずです。かつては時間計測の中核部分に致命的なバグがありました

hotshot プロファイラはまだスレッド環境ではうまく動作しません。測定したいコード上でプロファイラを実行するためにスレッドを使わない版のスクリプトを使う方法が有用です。

class Profile (*logfile*[, *lineevents*[, *linetimings*]])

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/ リターンの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。

25.8.1 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。

runctx (*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。

25.8.2 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 (25.4 節):
profile モジュールの Stats クラス

25.8.3 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)
 .
 .
 .
```

25.9 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の指定で、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。

注意: デフォルトでは、timeit() は時間計測中、一時的にガーベッジコレクションを切ります。このアプローチの利点は、個別の測定結果を比較しやすくなることです。不利な点は、GC が測定している関数のパフォーマンスの重要な一部かもしれないということです。そうした場合、*setup* 文字列の最初の文で GC を再度有効にすることができます。例えば:

```
timeit.Timer('for i in xrange(10): oct(i)', 'gc.enable()').timeit()
```

25.9.1 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。

25.9.2 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()
```

— Python-Docs-2.4/./lib/libtrace.tex 1970-01-01 09:00:00.000000000 +0900 +++ Python-Docs-2.5/./lib/libtrace.tex 2006-05-03 11:04:40.000000000 +0900 @@ -0,1 +1,1 @@

25.10 trace — Python ステートメント実行のトレースと追跡

trace モジュールはプログラム実行のトレースを可能にし, generate ステートメントのカバレッジリストを注釈付きで生成して、呼び出し元/呼び出し先の関連やプログラム実行中に実行された関数のリストを出力します。これは別個のプログラム中またはコマンドラインから利用することができます。

25.10.1 コマンドラインからの利用

trace モジュールはコマンドラインから起動することができます。これは次のように単純です。

```
python -m trace --count somefile.py ...
```

これで、'somefile.py' の実行中に import された Python モジュールの注釈付きリストが生成されます。

以下のコマンドライン引数がサポートされています：

--trace, -t 実行されるままに行を表示します。

--count, -c プログラム完了時に、それぞれのステートメントが何回実行されたかを示す注釈付きリストのファイルを生成します。

--report, -r **--count** と **--file** 引数を使った、過去のプログラム実行結果から注釈付きリストのファイルを生成します。

--no-report, -R 注釈付きリストを生成しません。これは **--count** を何度か走らせてから最後に単一の注釈付きリストを生成するような場合に便利です。

--listfuncs, -l プログラム実行の際に実行された関数を列挙します。

--trackcalls, -T プログラム実行によって明らかになった呼び出しの関連を生成します。

--file, -f カウント (count) を含む（べき）ファイルに名前をつけます。

--coverdir, -C 中に注釈付きリストのファイルを保存するディレクトリを指定します。

--missing, -m 注釈付きリストの生成時に、実行されなかった行に '»»»' の印を付けます。

--summary, -s **--count** または **--report** の利用時に、処理されたファイルそれぞれの簡潔なサマリを標準出力 (stdout) に書き出します。

--ignore-module 指定されたモジュールとそのサブモジュールを（パッケージだった場合に）無視します。複数回指定できます。

--ignore-dir 指定されたディレクトリとサブディレクトリ中のモジュールとパッケージを全て無視します。複数回指定できます。

25.10.2 プログラミングインターフェース

class Trace(*[count=1[, trace=1[, countfuncs=0[, countcallers=0[, ignoremods=()[, ignoredirs=()[, infile=None[, outfile=None]]]]]]]]*)

文 (statement) や式 (expression) の実行をトレースするオブジェクトを作成します。全てのパラメタがオプションです。*count* は行数を数えます。*trace* は行実行のトレースを行います。*countfuncs* は実行中に呼ばれた関数を列挙します。*countcallers* は呼び出しの関連の追跡を行います。*ignoremods* は無

視するモジュールやパッケージのリストです。*ignoredirs* は無視するパッケージやモジュールを含むディレクトリのリストです。*infile* は保存された集計 (count) 情報を読むファイルです。*outfile* は更新された集計 (count) 情報を書き出すファイルです。

run(*cmd*)

cmd を、Trace オブジェクトのコントロール下で現在のトレースパラメタのもとに実行します。

runctx(*cmd*[, *globals=None*[, *locals=None*]])

cmd を、Trace オブジェクトのコントロール下で現在のトレースパラメタのもと、定義されたグローバルおよびローカル環境で実行します。定義しない場合、*globals* と *locals* はデフォルトで空の辞書となります。

runfunc(*func, *args, **kwds*)

与えられた引数の *func* を、Trace オブジェクトのコントロール下で現在のトレースパラメタのもとに呼び出します。

これはこのモジュールの使い方を示す簡単な例です：

```
import sys
import trace

# Trace オブジェクトを、無視するもの、トレースや行カウントのいずれか
# または両方を行うか否かを指定して作成します。
tracer = trace.Trace(
    ignoredirs=[sys.prefix, sys.exec_prefix],
    trace=0,
    count=1)

# 与えられたトレーサを使って、コマンドを実行します。
tracer.run('main()')

# 出力先を /tmp としてレポートを作成します。
r = tracer.results()
r.write_results(show_missing=True, coverdir="/tmp")
```

二十六章

Python ランタイム サービス

この章では、Python インタープリタや 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:

sys	システムパラメータと関数へのアクセス
__builtin__	組み込み名前空間を提供するモジュール
__main__	トップレベルスクリプトが実行される環境。
warnings	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。
contextlib	with-構文 コンテキストのためのユーティリティ。
atexit	後始末関数の登録と実行。
traceback	スタックトレースの表示や取り出し。
__future__	Future ステートメントの定義
gc	循環検出ガベージコレクタのインターフェース。
inspect	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。
site	サイト固有のモジュールを参照する標準の方法。
user	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
fpectl	浮動小数点例外処理の制御。

26.1 sys — システムパラメータと関数

このモジュールでは、インタープリタで使用・管理している変数や、インター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。

subversion

3 つ組 (repo, branch, version) で Python インタプリタの Subversion 情報を表します。*repo* はリポジトリの名前で、'CPython'。*branch* は 'trunk'、'branches/name' または'tags/name' のいずれかの形式の文字列です。*version* はもしインタプリタが Subversion のチェックアウトからビルドされたものならば svnversion の出力であり、リビジョン番号 (範囲) とローカルでの変更がある場合には最後に 'M' が付きます。ツリーがエクスポートされたもの (または svnversion が取得できない) で、branch がタグならば Include/patchlevel.h のリビジョンになります。それ以外の場合

には None です。 2.5 で追加された仕様です。

builtin_module_names
コンパイル時に Python インター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 インタープリタの著作権を表示する文字列。

_current_frames()
各スレッドの識別子を関数が呼ばれた時点のそのスレッドでアクティブになっている一番上のスタックフレームに結びつける辞書を返します。モジュール traceback の関数を使えばそのように与えられたフレームのコールスタックを構築できます。

この関数はデッドロックをデバッグするのに非常に有効です。デッドロック状態のスレッドの協調動作を必要としませんし、そういったスレッドのコールスタックはデッドロックである限り凍り付いたままです。デッドロックにないスレッドのフレームについては、そのフレームを調べるコードを呼んだ時にはそのスレッドの現在の実行状況とは関係ないところを指し示しているかもしれません。

この関数は外部に見せない特別な目的でのみ使われるべきです。 2.5 で追加された仕様です。

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、インター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 `exctype, 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/python`*version*`/config'` に、共有ライブラリは `exec_prefix + '/lib/python`*version*`/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、インター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:** プログラムがシグナルで kill された場合，Python 内部で致命的なエラーが発生した場合，os._exit() が呼び出された場合には，終了処理関数は呼び出されません。**リリース 2.4 以降で撤廃された仕様です。** atexit を使ってください.

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 nl_langinfo(CODESET) が返すユーザの設定となります。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できるので変換の必要はありません。getfilesystemencoding() は'mbcs' を返しますが、これはある Unicode 文字列をバイト文字列に明示的に変換して、ファイル名として使うと同じファイルを指すようにしたい場合に、アプリケーションが使わねばならないエンコーディングです。

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() からも参照されるため、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 インター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:

Constant	Platform
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) ごとにかならず増加します。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ー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 24 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 -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/python`*version*`'` 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/python`*version*`'` 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 25 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、プロファイラ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 インタープリタの、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。24.2 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。**注意:** `settrace()` 関数は，デバッガ，プロファイラ，カバレッジツール等で使うためだけのものです．この関数の挙動は言語定義よりも実装プラットフォームの分野の問題で，全ての Python 実装で利用できるとは限りません．

`settscdump` (*on_flag*)

on_flag が真の場合，Pentium タイムスタンプカウンタを使った VM 計測結果のダンプ出力を有効にします．*on_flag* をオフにするとダンプ出力を無効化します．この関数は Python を**--with-tsc** つきでコンパイルしたときにのみ利用できます．ダンプの内容を理解したければ， Python ソースコード中の 'Python/ceval.c' を読んでください． 2.4 で追加された仕様です。

`stdin`
`stdout`
`stderr`

インタープリタの標準入力・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。インター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 インター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
使用中のインタープリタの 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 (26.11 節):
This describes how to use .pth files to extend sys.path.

26.2 __builtin__ — 組み込みオブジェクト

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 章, "組み込みオブジェクト" も参照してください。

このモジュールは通常ほとんどのアプリケーションにおいて直接名指しでアクセスされることはありませんが、組み込みの名前と同じ名前のオブジェクトを提供しつつ組み込みのその名前も必要であるようなモジュールにおいて有用です。たとえば、open() という関数を組み込みの open() をラップして実装したいというモジュールがあったとすると、このモジュールは次のように直接的に使われます。

```
import __builtin__

def open(path):
    f = __builtin__.open(path, 'r')
    return UpperCaser(f)

class UpperCaser:
    '''Wrapper around a file that converts output to upper-case.'''

    def __init__(self, f):
        self._f = f

    def read(self, count=-1):
        return self._f.read(count).upper()

    # ...
```

実装の詳細に属することですが、ほとんどのモジュールでは__builtins__ ('s' に注意) がグローバルの一部として使えるようになっています。__builtins__ の内容は通常このモジュールそのものか、またはこのモジュールの__dict__ 属性です。実装の詳細部分ということで、異なる Python の実装の下ではこのようになっていないかもしれません。

26.3 __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()
```

26.4 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 プログラマは、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 言語のプログラマは 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) と動作からなるシーケンスです。`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。

26.4.1 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
`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
`PendingDeprecationWarning`	将来その機能が廃用化されることを示す警告カテゴリの基底クラスです (デフォルト
`ImportWarning`	モジュールのインポート処理中に引き起こされる警告カテゴリの基底クラスです (デ
`UnicodeWarning`	Unicode に関係した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。

26.4.2 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)。

デフォルトでは無視される警告を **-Wd** をインタプリタに渡すことで有効にすることができます。このオプションは通常はデフォルトで無視されるようなものを含む全ての警告のデフォルトでの扱いを有効化します。このような振る舞いは開発中のパッケージをインポートする問題をデバッグする時に ImportWarning を有効化するために使えます。ImportWarning は次のような Python コードを使って明示的に有効化することもできます。

```
warnings.simplefilter('default', ImportWarning)
```

26.4.3 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 でラッパ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*[, *module_globals*]]])

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。

module_globals は、もし与えられるならば、警告が発せられるコードが使っているグローバル名前空間でなければなりません。(この引数は zipfile やその他の非ファイルシステムのインポート元の中にあるモジュールのソースを表示することをサポートするためのもので、Python 2.5 で追加されました。)

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リストに挿入します。二つのエントリが特定の警告に合致した場合、リストの先頭に近い方のエントリが後方にあるエントリに優先します。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。

simplefilter(*action*[, *category*[, *lineno*[, *append*]]])

単純なエントリを警告フィルタのリストに挿入します。引数の意味は filterwarnings() と同じですが、この関数により挿入されるフィルタはカテゴリと行番号が一致していれば全てのモジュールの全てのメッセージに合致しますので、正規表現は必要ありません。

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 simplefilter() 呼び出しによるものを含め、filterwarnings の呼び出しによる影響はすべて無効化されます。

26.5 contextlib — with-構文 コンテキストのためのユーティリティ。

2.5 で追加された仕様です。

このモジュールは with 文を必要とする一般的なタスクのためのユーティリティを提供します。

用意されている関数:

contextmanager(*func*)

この関数はデコレータであり、with 文コンテキストマネージャのためのファクトリ関数の定義に利用できます。ファクトリ関数を定義するために、クラスあるいは別の__enter__() と__exit__

() メソッドを作る必要はありません。

簡単な例（実際に HTML を生成する方法としてはお勧めできません！）:

```
from __future__ import with_statement
from contextlib import contextmanager

@contextmanager
def tag(name):
    print "<%s>" % name
    yield
    print "</%s>" % name

>>> with tag("h1"):
...    print "foo"
...
<h1>
foo
</h1>
```

デコレートされた関数は呼び出されたときにジェネレータ-イテレータを返します。このイテレータは値をちょうど一つ yield しなければなりません。with 文の as 節が存在するなら、その値が as 節のターゲットへ束縛されることになります。

ジェネレータが yield するところで、with 文のネストされたブロックが実行されます。ジェネレータはブロックから出た後に再開されます。ブロック内で処理されない例外が発生した場合は、yield が起きた場所でジェネレータ内部へ再送出されます。このように、（もしあれば）エラーを捕捉したり、後片付け処理を確実に実行したりするために、try...except...finally 文を使うことができます。単に例外のログをとるためだけに、もしくは（完全に例外を抑えてしまうのではなく）あるアクションを実行するだけに例外を捕まえるなら、ジェネレータはその例外を再送出しなければなりません。そうしないと、ジェネレータコンテキストマネージャは例外が処理された with 文を指しており、その with 文のすぐ後につづく文から実行を再開します。

nested(*mgr1*[, *mgr2*[, ...]])

複数のコンテキストマネージャを一つのネストされたコンテキストマネージャへ結合します。

このようなコードは:

```
from contextlib import nested

with nested(A, B, C) as (X, Y, Z):
    do_something()
```

これと同等です:

```
with A as X:
    with B as Y:
        with C as Z:
            do_something()
```

ネストされたコンテキストマネージャの一つの__exit__() メソッドに止めるべき例外がある場合は、残りの外側のコンテキストマネージャすべてに例外情報が渡されないということに注意してください。同じように、ネストされたマネージャの一つの__exit__() メソッドが例外を送出したならば、どんな以前の例外状態も失われ、新しい例外が残りすべての外側にあるコンテキストマネージャの__exit__() メソッドに渡されます。一般的に__exit__() メソッドが例外を送出することは避けるべきであり、特に渡された例外を再送出すべきではありません。

closing(*thing*)

ブロックの完了時に *thing* を閉じるコンテキストマネージャを返します。これは基本的に以下と等価です:

```
from contextlib import contextmanager

@contextmanager
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()
```

そして、明確に `page` を閉じる必要なしに、このように書くことができます:

```
from __future__ import with_statement
from contextlib import closing
import codecs

with closing(urllib.urlopen('http://www.python.org')) as page:
    for line in page:
        print line
```

たとえエラーが発生したとしても、`with` ブロックを出るときに `page.close()` が呼ばれます。

参考資料:

PEP 0343, "*The "with" statement*"

仕様、背景、および、Python `with` 文の例。

26.6 atexit — 終了ハンドラ

2.0 で追加された仕様です。

`atexit` モジュールでは、後始末関数を登録するための関数を一つだけ定義しています。この関数を使って登録した後始末関数は、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した関数は呼び出されません。

このモジュールは、`sys.exitfunc` 変数の提供している機能の代用となるインタフェースです。

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、このモジュールは正しく動作しないでしょう。特に、他のコア Python モジュールでは、プログラマの意図を知らなくても `atexit` を自由に使えます。`sys.exitfunc` を使っている人は、代わりに `atexit` を使うコードに変換してください。`sys.exitfunc` を設定するコードを変換するには、`atexit` を import し、`sys.exitfunc` へ束縛されていた関数を登録するのが最も簡単です。

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()` が呼び出されるとき、あるいは、メインモジュールの実行が完了したときに、登録された全ての関数を、最後に登録されたものから順に呼び出します。通常、より低レベルのモジュールはより高レベルのモジュールより前に import されるので、後で後始末が行われるという仮定に基づいています。

終了ハンドラの実行中に例外が発生すると、(SystemExit 以外の場合は) トレースバックを表示して、例外の情報を保存します。全ての終了ハンドラに動作するチャンスを与えた後に、最後に送出された例外を再送出します。

参考資料:

readline モジュール (15.7 節):

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

26.6.1 atexit 例

次の簡単な例では、あるモジュールを import した時にカウンタを初期化しておき、プログラムが終了する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カウンタが更新されるように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() に指定した固定引数とキーワードパラメ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')
```

26.7 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ト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。)

format_exc([*limit*])

これは、print_exc(*limit*) に似ていますが、ファイルに出力するかわりに文字列を返します。 2.4 で追加された仕様です。

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() が返すタプ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。

26.7.1 トレースバックの例

この簡単な例では基本的な read-eval-print ループを実装います。それは標準的な Python の対話インター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。インター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)
```

26.8 __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 _Feature インスタンスの compilier_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。

26.9 gc — ガベージコレクタ インターフェース

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 Pyhon 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。これは gc.DEBUG_SAVEALL を含んでいることに注意しましょう。ガベージとして検出されたオブジェクトは、インスペクション用に gc.garbage に保存されます。

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([*generation*])
引数を指定しない場合は、全ての検出を行います。オプションの引数 *generation* は、どの世代を検出するかを (0 から 2 までの) 整数値で指定します。無効な世代番号を指定した場合は `ValueError` が発生します。検出した到達不可オブジェクトの数を返します。

2.5 で変更された仕様: オプションの引数 *generation* が追加されました

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_count()
現在の検出数を、(*count0*, *count1*, *count2*) のタプルで返します。 2.5 で追加された仕様です。

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。

`get_referrers()` から返されるオブジェクトは作りかけや利用できない状態である場合があるので、利用する際には注意が必要です。`get_referrers()` をデバッグ以外の目的で利用するのは避けてください。

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。[1]

__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 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、garbade 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。)

[1] 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。)

26.10 inspect — 使用中オブジェクトの情報を取得する

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・インター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。

26.10.1 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
	func_name	(__name__と同じ)
traceback	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
	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
frame	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
	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
code	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
	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ときの名前
	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` で定義している整定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。

getmodulename(*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、インター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 getset やメンバです。後者のふたつは C で定義されており、個々の型に特有のテストも行います。そのため、Python の実装よりもより確実確実です。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。

isgetsetdescriptor(*object*)
オブジェクトが getset デスクリプタの場合に真を返します。

getset とは `PyGetSetDef` 構造体を用いて拡張モジュールで定義されている属性のことです。Python の実装の場合はそのような型はないので、このメソッドは常に `False` を返します。 2.5 で追加された仕様です。

ismemberdescriptor(*object*)
オブジェクトがメンバデスクリプタの場合に真を返します。

メンバデスクリプタとは `PyMemberDef` 構造体を用いて拡張モジュールで定義されている属性のことです。Python の実装の場合はそのような型はないので、このメソッドは常に `False` を返します。 2.5 で追加された仕様です。

26.10.2 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のモジュール、クラス、関数の場合には、`TypeError` 例外が発生します。

getsourcelines(*object*)
オブジェクトのソース行のリストと開始行番号を返します。引数にはモジュール・クラス・メソッド・関数・トレースバック・フレーム・コードオブジェクトを指定する事ができます。戻り値は指定したオブジェクトに対応するソースコードのソース行リストと元のソースファイル上での開始行となります。ソースコードを取得できない場合は `IOError` が発生します。

getsource(*object*)
オブジェクトのソースコードを返します。引数にはモジュール・クラス・メソッド・関数・トレース

バック・フレーム・コードオブジェクトを指定する事ができます。ソースコードは単一の文字列で返します。ソースコードを取得できない場合は `IOError` が発生します。

26.10.3 クラスと関数

getclasstree (*classes*[, *unique*])

リストで指定したクラスの継承関係から、ネストしたリストを作成します。ネストしたリストには、直前の要素から派生したクラスが格納されます。各要素は長さ 2 のタプルで、クラスと基底クラスのタプルを格納しています。*unique* が真の場合、各クラスは戻り値のリスト内に一つだけしか格納されません。真でなければ、多重継承を利用したクラスとその派生クラスは複数回格納される場合があります。

getargspec (*func*)

関数の引数名とデフォルト値を取得します。戻り値は長さ 4 のタプルで、次の値を返します: (*args*, *varargs*, *varkw*, *defaults*)。*args* は引数名のリストです（ネストしたリストが格納される場合があります）。*varargs* と *varkw* は*引数と**引数の名前で、引数がなければ `None` となります。*defaults* は引数のデフォルト値のタプルか、デフォルト値がない場合は `None` です。このタプルに *n* 個の要素があれば、各要素は *args* の後ろから *n* 個分の引数のデフォルト値となります。

getargvalues (*frame*)

指定したフレームに渡された引数の情報を取得します。戻り値は長さ 4 のタプルで、次の値を返します: (*args*, *varargs*, *varkw*, *locals*)。*args* は引数名のリストです（ネストしたリストが格納される場合があります）。*varargs* と *varkw* は*引数と**引数の名前で、引数がなければ `None` となります。*locals* は指定したフレームのローカル変数の辞書です。

formatargspec (*args*[, *varargs, varkw, defaults, formatarg, formatvarargs, formatvarkw, formatvalue, join*])

`getargspec()` で取得した 4 つの値を読みやすく整形します。format* 引数はオプションで、名前と値を文字列に変換する整形関数を指定する事ができます。

formatargvalues (*args*[, *varargs, varkw, locals, formatarg, formatvarargs, formatvarkw, formatvalue, join*])

`getargvalues()` で取得した 4 つの値を読みやすく整形します。format* 引数はオプションで、名前と値を文字列に変換する整形関数を指定する事ができます。

getmro (*cls*)

cls クラスの基底クラス（*cls* 自身も含む）を、メソッドの優先順位順に並べたタプルを返します。結果のリスト内で各クラスは一度だけ格納されます。メソッドの優先順位はクラスの型によって異なります。非常に特殊なユーザ定義のメタクラスを使用していない限り、*cls* が戻り値の先頭要素となります。

26.10.4 インタープリタ スタック

以下の関数には、戻り値として “フレームレコード” を返す関数があります。“ フレームレコード” は長さ 6 のタプルで、以下の値を格納しています:フレームオブジェクト・ファイル名・実行中の行番号・関数名・コンテキストのソース行のリスト・ソース行リストの実行中行のインデックス。

警告:
フレームレコードの最初の要素などのフレームオブジェクトへの参照を保存すると、循環参照になってしまう場合があります。循環参照ができると、Python の循環参照検出機能を有効にしていたとしても関連するオブジェクトが参照しているすべてのオブジェクトが解放されにくくなり、明示的に参照を削除しないとメモリ消費量が増大する恐れがあります。
参照の削除を Python の循環参照検出機能にまかせる事もできますが、`finally` 節で循環参照を解除すれば確実にフレーム（とそのローカル変数）は削除されます。また、循環参照検出機能は Python のコンパイルオプションや `gc.  disable()` で無効とされている場合がありますので注意が必要です。例：

```
def handle_stackframe_without_leak():
    frame = inspect.currentframe()
    try:
        # do something with the frame
    finally:
        del frame
```

以下の関数でオプション引数 *context* には、戻り値のソース行リストに何行分のソースを含めるかを指定します。ソース行リストには、実行中の行を中心として指定された行数分のリストを返します。

getframeinfo (*frame*[, *context*])
フレーム又はトレースバックオブジェクトの情報を取得します。フレームレコードの先頭要素を除いた、長さ 5 のタプルを返します。

getouterframes (*frame*[, *context*])
指定したフレームと、その外側の全フレームのフレームレコードを返します。外側のフレームとは *frame* が生成されるまでのすべての関数呼び出しを示します。戻り値のリストの先頭は *frame* のフレームレコードで、末尾の要素は *frame* のスタックにあるもっとも外側のフレームのフレームレコードとなります。

getinnerframes (*traceback*[, *context*])
指定したフレームと、その内側の全フレームのフレームレコードを返します。内のフレームとは *frame* から続く一連の関数呼び出しを示します。戻り値のリストの先頭は *traceback* のフレームレコードで、末尾の要素は例外が発生した位置を示します。

currentframe ()
呼び出し元のフレームオブジェクトを返します。

stack ([*context*])
呼び出し元スタックのフレームレコードのリストを返します。最初の要素は呼び出し元のフレームレコードで、末尾の要素はスタックにあるもっとも外側のフレームのフレームレコードとなります。

trace ([*context*])
実行中のフレームと処理中の例外が発生したフレームの間のフレームレコードのリストを返します。最初の要素は呼び出し元のフレームレコードで、末尾の要素は例外が発生した位置を示します。

26.11 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。自動インポートはインタプリタの**-S** オプションで禁止できます。

このモジュールをインポートすることで、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、まず空文字列を使い、次に 'lib/site-packages'(Windows) または 'lib/python2.5/site-packages'、そして 'lib/site-python' (UNIX と Macintosh) 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、上の 4 つのディレクトリのひとつにあります。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 2.5 ライブラリは '/usr/local/lib/python2.5' 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/python2.5/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 へこの順番んで追加されます:

```
/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。

26.12 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

verbose = bool(getattr(user, "spam_verbose", 0))
```

(ユーザが spam_verbose をファイル '.pythonrc.py' 内で定義していない時に getattr() の 3 引数形式は使われます。)

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てしまいます。

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 (26.11 節):
サイト毎のカスタマイズを行う機構

26.13 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ホームで動作します。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。

26.13.1 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
```

26.13.2 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。

二十七章

カスタム Python インタプリタ

この章で解説されるモジュールで Python の対話インタプリタに似たインタフェースを書くことができます。もし Python そのもの以外に何か特殊な機能をサポートした Python インタプリタを作りたければ、code モジュールを参照してください。(codeop モジュールはより低レベルで、不完全 (かもしれない) Python コード断片のコンパイルをサポートするために使われます。)

この章で解説されるモジュールの完全な一覧は:

code 対話的 Python インタプリタのための基底クラス。

codeop (完全ではないかもしれない)Python コードをコンパイルする。

27.1 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 をつかっ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。

27.1.1 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。あるいくつかのことが起きる可能性があります:

- 入力はが正しくない。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。伝わることが許されている 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() メソッドによて書き込まれます。

write(*data*)

文字列を標準エラーストリーム (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。

27.1.2 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にエミュレートします。オプションの banner 引数は最初のやりと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。

27.2 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。

二十八章

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。

28.1 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はいますが、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、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を呼び出しません — 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 アプレ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。

28.1.1 REexec オブジェクト

`REexec` 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。

`REexec` 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。

28.1.2 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', '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', 'getuid', 'getgid', 'geteuid', '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ちあげのファイル ('.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。

28.1.3 例

標準の RExec クラスよりも、若干、もっと緩めたポリシ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。

28.2 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、バスティアン (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。

二十九章

モジュールのインポート

この章で解説されるモジュールは他の Python モジュールをインポートする新しい方法と、インポート処理をカスタマイズするためのフックを提供します。

この章で解説されるモジュールの完全な一覧は:

imp	import 文の実装へアクセスする。
zipimport	Python モジュール を ZIP アーカイブから import する機能のサポート
pkgutil	パッケージの拡張をサポートするユーティリティです。
modulefinder	スクリプト中で使われているモジュールを検索します。
runpy	Python モジュールの位置特定とスクリプトとしての実行

29.1 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ホームでは、常に False を返します。

スレッドのあるプラットホームでは、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ホ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ホ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 Mac OS 9 リソースとして発見された。この値は Mac OS 9 以前の 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と、`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に、例外を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 'init*name*()' 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。

class NullImporter(*path_string*)

`NullImporter` 型は PEP 302 インポートフックで、何もモジュールが見つからなかったときの非ディレクトリパス文字列を処理します。この型を既存のディレクトリや空文字列に対してコールすると `ImportError` が発生します。それ以外の場合は `NullImporter` のインスタンスが返されます。

Python は、ディレクトリでなく `sys.path_hooks` のどのパスフックでも処理されていないすべてのパスエントリに対して、この型のインスタンスを `sys.path_importer_cache` に追加します。このインスタンスが持つメソッドは次のひとつです。

find_module(*fullname* [, *path*])

このメソッドは常に `None` を返し、要求されたモジュールが見つからなかったことを表します。

2.5 で追加された仕様です。

29.1.1 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。

29.2 `zipimport` — Zip アーカイブからモジュールを import する

2.3 で追加された仕様です。

このモジュールは， Python モジュール ('*.py'，'*.py[co]' やパッケージを ZIP 形式のアーカイブから import できるようにします。通常，`zipimport` を明示的に使う必要はありません; 組み込みの `import` は，`sys.path` の要素が ZIP アーカイブへのパスを指している場合にこのモジュールを自動的に使います。

普通，`sys.path` はディレクトリ名の文字列からなるリストです。このモジュールを使うと，`sys.path` の要素に ZIP ファイルアーカイブを示す文字列を使えるようになります。ZIP アーカイブにはサブディレクトリ構造を含めることができ，パッケージの import をサポートさせしたり，アーカイブ内のパスを指定してサブディレクトリ下から import を行わせたりできます。例えば，'/tmp/example.zip/lib/' のように指定すると，アーカイブ中の 'lib/' サブディレクトリ下だけから import を行います。

ZIP アーカイブ内にはどんなファイルを置いてもかまいませんが，import できるのは '.py' および '.py[co]' だけです。動的モジュール ('.pyd'，'.so') の ZIP import は行えません。アーカイブ内に '.py' ファイルしか入っていない場合，Python がアーカイブを変更して， '.py' ファイルに対応する '.pyc' や '.pyo' ファイルを追加したりはしません。つまり，ZIP アーカイブ中に '.pyc' が入っていない場合， import はやや低速になるかもしれないので注意してください。

ZIP アーカイブからロードしたモジュールに対して組み込み関数 `reload()` を呼び出すと失敗します; `reload()` が必要になるということは，実行時に ZIP ファイルが置き換えられてしまうことになり，あまり起こりそうにない状況だからです。

このモジュールで使える属性を以下に示します:

exception `ZipImporterError`

zipimporter オブジェクトが送出する例外です。`ImportError` のサブクラスなので，`ImportError` としても捕捉できます。

class `zipimporter`

ZIP ファイルを import するためのクラスです。コンストラクタの詳細は “*zipimporter* **オブジェクト**” (29.2.1 節) を参照してください。

参考資料:

PKZIP Application Note

(http://www.pkware.com/appnote.html)

ZIP ファイル形式の作者であり，ZIP で使われているアルゴリズムの作者でもある Phil Katz による，ZIP ファイル形式についてのドキュメントです。

PEP 0273, “*Import Modules from Zip Archives*”

このモジュールの実装も行った、James C. Ahlstrom による PEP です。Python 2.3 は PEP 273 の仕様に従っていますが、 Just van Rossum の書いた import フックによる実装を使っています。 import フックは PEP 302 で解説されています。

PEP 0302, “*New Import Hooks*”

このモジュールを動作させる助けになっている import フックの追加を提案している PEP です。

29.2.1 zipimporter オブジェクト

class zipimporter(*archivepath*)

新たな zipimporter インスタンスを生成します。*archivepath* は ZIP ファイルへのパスでなければなりません。 *archivepath* が有効な ZIP アーカイブを指していない場合、ZipImportError を送出します。

find_module(*fullname*[, *path*])

fullname に指定したモジュールを検索します。*fullname* は完全指定の (ドット表記の) モジュール名でなければなりません。モジュールが見つかった場合には zipimporter インスタンス自体を返し、そうでない場合には None を返します。*path* 引数は無視されます — この引数は importer プロトコルとの互換性を保つためのものです。

get_code(*fullname*)

fullname に指定したモジュールのコードオブジェクトを返します。モジュールがない場合には ZipImportError を送出します。

get_data(*pathname*)

pathname に関連付けられたデータを返します。該当するファイルが見つからなかった場合には IOError を送出します。

get_source(*fullname*)

fullname に指定したモジュールのソースコードを返します。モジュールが見つからなかった場合には ZipImportError を送出します。モジュールは存在するが、ソースコードがない場合には None を返します。

is_package(*fullname*)

fullname で指定されたモジュールがパッケージの場合に True を返します。モジュールが見つからなかった場合には ZipImportError を送出します。

load_module(*fullname*)

fullname に指定したモジュールをロードします。*fullname* は完全指定の (ドット表記の) モジュール名でなくてはなりません。import 済みのモジュールを返します。モジュールがない場合には ZipImportError を送出します。

29.2.2 使用例

モジュールを ZIP アーカイブから import する例を以下に示します - zipimport モジュールが明示的に使われていないことに注意してください。

```
$ unzip -l /tmp/example.zip
Archive:  /tmp/example.zip
  Length     Date   Time    Name
 --------    ----   ----    ----
     8467  11-26-02 22:30   jwzthreading.py
 --------                   -------
     8467                   1 file
$ ./python
Python 2.3 (#1, Aug 1 2003, 19:54:32)
>>> import sys
>>> sys.path.insert(0, '/tmp/example.zip')  # パス先頭に .zip ファイル追加
>>> import jwzthreading
>>> jwzthreading.__file__
'/tmp/example.zip/jwzthreading.py'
```

29.3 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。

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。

29.4 modulefinder — スクリプト中で使われているモジュールを検索する

2.3 で追加された仕様です。

このモジュールでは、スクリプト中で import されているモジュールセットを調べるために使える ModuleFinder クラスを提供しています。modulefinder.py はまた、Python スクリプトのファイル名を引数に指定してスクリプトとして実行し、import されているモジュールのレポートを出力させることもできます。

AddPackagePath(*pkg_name, path*)
pkg_name という名前のパッケージの在り処が *path* であることを記録します。

ReplacePackage(*oldname, newname*)
実際にはパッケージ内で *oldname* という名前になっているモジュールを *newname* という名前で指定できるようにします。この関数の主な用途は、_xmlplus パッケージが xml パッケージに置き換わっている場合の処理でしょう。

class ModuleFinder([*path=None, debug=0, excludes=[], replace_paths=[]*])
このクラスでは run_script() および report() メソッドを提供しています。これらのメソッドは何らかのスクリプト中で import されているモジュールの集合を調べます。*path* はモジュールを検索する先のディレクトリ名からなるリストです。*path* を指定しない場合、sys.path を使います。*debug* にはデバッグレベルを設定します; 値を大きくすると、実行している内容を表すデバッグメッセージを出力します。*excludes* は検索から除外するモジュール名です。*replace_paths* には、モジュールパス内で置き換えられるパスをタプル (*oldpath*, *newpath*) からなるリストで指定します。

report()
スクリプトで import しているモジュールと、そのパスからなるリストを列挙したレポートを標準出力に出力します。モジュールを見つけられなかったり、モジュールがないように見える場合にも報告します。

run_script(*pathname*)
pathname に指定したファイルの内容を解析します。ファイルには Python コードが入っていなければなりません。

29.5 runpy — Python モジュールの位置特定と実行

2.5 で追加された仕様です。

runpy モジュールは Python のモジュールをインポートせずにその位置を特定したり実行したりするのに使われます。その主な目的はファイルシステムではなく Python のモジュール名前空間を使って位置を特定したスクリプトの実行を可能にする **-m** コマンドラインスイッチを実装することです。

スクリプトとして実行されると、このモジュールは効率よく以下の操作をします。

```
del sys.argv[0]  # Remove the runpy module from the arguments
run_module(sys.argv[0], run_name="__main__", alter_sys=True)
```

runpy モジュールでは一つの関数だけ提供します。

run_module(*mod_name*[, *init_globals*] [, *run_name*][, *alter_sys*])
指定されたモジュールのコードを実行し、実行後のモジュールグローバル辞書を返します。モジュールのコードはまず標準インポート機構 (詳細は PEP 302 を参照) を使ってモジュールの位置を特定さ

れ、まっさらなモジュール名前空間で実行されます。

オプションの辞書型引数 *init_globals* はコードを実行する前にグローバル辞書に前もって必要な設定しておくのに使われます。与えられた辞書は変更されません。その辞書の中に以下に挙げる特別なグローバル変数が定義されていたとしても、それらの定義は `run_module` 関数によってオーバーライドされます。

特別なグローバル変数 `__name__`、`__file__`、`__loader__`、`__builtins__` はモジュールコードが実行される前にグローバル辞書にセットされます。

`__name__` は、もしオプション引数 *run_name* が与えられていればその値が、そうでなければ *mod_name* 引数の値がセットされます。

`__loader__` はモジュールのコードを取得するのに使われる PEP 302 のモジュールローダがセットされます (このローダは標準のインポート機構に対するラッパーかもしれません)。

`__file__` はモジュールローダにより与えられた名前がセットされます。もしローダがファイル名情報を取得可能にしなければ、この変数の値は `None` になります。

`__builtins__` は自動的に `__builtin__` モジュールのトップレベル名前空間への参照で初期化されます。

引数 *alter_sys* が与えられて `True` に評価されるならば、`sys.argv[0]` は `__file__` の値で更新され `sys.modules[__name__]` は実行されるモジュールの一時的モジュールオブジェクトで更新されます。`sys.argv[0]` と `sys.modules[__name__]` はどちらも関数が処理を戻す前にもとの値に復旧します。

この `sys` に対する操作はスレッドセーフではないということに注意してください。他のスレッドは部分的に初期化されたモジュールを見たり、入れ替えられた引数リストを見たりするかもしれません。この関数をスレッド化されたコードから起動するときは `sys` モジュールには手を触れないことが推奨されます。

参考資料:

PEP 338, "*Executing modules as scripts*"

Nick Coghlan によって書かれ実装された PEP

三十章

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:

parser	Python ソースコードに対する解析木へのアクセス。
symbol	Constants representing internal nodes of the parse tree.
token	Constants representing terminal nodes of the parse tree.
keyword	文字列が Python のキーワードか否かを調べます。
tokenize	Python ソースコードのための字句解析器。
tabnanny	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。
pyclbr	Python クラスデスクリプタの情報抽出サポート
py_compile	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。
compileall	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。
dis	Python バイトコードの逆アセンブラ。
pickletools	pickle プロトコルと pickle マシン opcode に関する詳しいコメントと、有用な関数がいくつかが入って
distutils	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

30.1 `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、インター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 ヘッダファイル '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` の中のキーワード `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。行番号情報が必要なときに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 ヘッダファイル '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 (30.2 節):
: 解析木の内部ノードを表す便利な定数。

`token` モジュール (30.3 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。

30.1.1 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` 例外を発生します。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) 構文に関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ー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。

30.1.2 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関係なく、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ルグラムによって診断することができます。

30.1.3 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。

30.1.4 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だけですが、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。

30.1.5 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。

30.1.6 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ー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い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、それ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.
"""
```

インター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

必要がでてきます。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、数を無視し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発見しません。

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。

30.2 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 (30.1 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。

30.3 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の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 ヘッダ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 (30.1 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。

30.4 `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
インター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。

30.5 `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 (3.9 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 (*srow*, *scol*); 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められます。 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

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 (3.9 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。もしくは、*readline* を呼び出し可能オブジェクトで `StopIteration` を送出することで補完を知らせるものとすることもできます。 2.5 で変更された仕様: `StopIteration` サポートの追加

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。

もう一つの関数がトークン化プロセスを逆転するために提供されています。これは、スクリプトを字句解析し、トークンのストリームに変更を加え、変更されたスクリプトを書き戻すようなツールを作成する際に便利です。

untokenize(*iterable*)

トークンの列を Python ソースコードに変換します。*iterable* は少なくとも二つの要素、トークン型およびトークン文字列、からなるシーケンスを返します。その他のシーケンスの要素は無視されます。

再構築されたスクリプトは一つの文字列として返されます。得られる結果はもう一度字句解析すると入力と一致することが保証されるので、変換がロスレスでありラウンドトリップできることは間違いありません。この保証はトークン型およびトークン文字列に対してのものでトークン間のスペース (コラム位置) のようなものは変わることがあり得ます。 2.5 で追加された仕様です。

スクリプト書き換えの例で、浮動小数点数リテラルを Decimal オブジェクトに変換します:

```
def decistmt(s):
    """Substitute Decimals for floats in a string of statements.

    >>> from decimal import Decimal
    >>> s = 'print +21.3e-5*-.1234/81.7'
    >>> decistmt(s)
    "print +Decimal ('21.3e-5')*-Decimal ('.1234')/Decimal ('81.7')"

    >>> exec(s)
    -3.21716034272e-007
    >>> exec(decistmt(s))
    -3.217160342717258261933904529E-7

    """
    result = []
    g = generate_tokens(StringIO(s).readline)   # tokenize the string
    for toknum, tokval, _, _, _  in g:
        if toknum == NUMBER and '.' in tokval:  # replace NUMBER tokens
            result.extend([
                (NAME, 'Decimal'),
                (OP, '('),
                (STRING, repr(tokval)),
                (OP, ')')
            ])
        else:
            result.append((toknum, tokval))
    return untokenize(result)
```

30.6 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 (30.5 節):
Python ソースコードの字句解析器。

30.7 `pyclbr` — Python クラスブラウザーサポート

この `pyclbr` 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* パラメタはシーケンス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。

30.7.1 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。

30.7.2 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。

30.8 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。

頻繁に必要となる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ー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* が 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* が 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 (30.9 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。

30.9 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、Pyton 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用されるファイル名の、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式を指定します。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。

'Lib/' ディレクトリ以下にある全ての '.py' ファイルを強制的にリコンパイルするには、以下のようにします:

```
import compileall

compileall.compile_dir('Lib/', force=True)

# .svn ディレクトリにあるファイルをのぞいて同じことをするにはこのようにします。
import re
compileall.compile_dir('Lib/', rx=re.compile('/[.]svn'), force=True)
```

参考資料:

`py_compile` モジュール (30.8 節):

Byte-compile a single source file.

30.10 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
```

(“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(*bytesource*)

bytesource オブジェクトを逆アセンブルします。*bytesource* 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。

distb(*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バックを使います。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。

パラメータインタープリテーションはローカルおよびグルーバル変数名、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。

opmap

バイトコードからオペレーション名へのマッピング辞書。

cmp_op

すべての比較演算名。

hasconst

定数パラ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。

30.10.1 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。

NOP
なにもしないコード。バイトコードオプティマイザでプレースホルダとして使われます。Do nothing code. Used as a placeholder by the bytecode optimizer.

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)。

LIST_APPEND

`list.append(TOS1, TOS)` を呼びます。 リスト内包表記を実装するために使われます。

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、グル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*
グルーバル名 `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、その後ろのオペコードの引数と一緒になって取られます。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。

HAVE_ARGUMENT

これはオペコードではありません。引数をとらないオペコード< `HAVE_ARGUMENT` と、とるオペコー

ド >= HAVE_ARGUMENT を分割する行です。

30.11 pickletools — pickle 開発者のためのツール群

2.3 で追加された仕様です。

このモジュールには、pickle モジュールの詳細に関わる様々な定数や実装に関する長大なコメント、そして pickle 化されたデータを解析する上で有用な関数をいくつか定義しています。このモジュールの内容は pickle および cPickle の実装に関わっている Python core 開発者にとって有用なものです; 普通の pickle 利用者にとっては、pickletools モジュールはおそらく関係ないものでしょう。

dis(*pickle*[, *out=None, memo=None, indentlevel=4*])

pickle をシンボル分解 (symbolic disassembly) した内容をファイル類似オブジェクト *out* (デフォルトでは sys.stdout) に出力します。*pickle* は文字列にもファイル類似オブジェクトにもできます。*memo* は Python 辞書型で、pickle のメモに使われます。同じ pickler の生成した複数の pickle 間にわたってシンボル分解を行う場合に使われます。ストリーム中で MARK opcode で表される継続レベル (successive level) は *indentlevel* に指定したスペース分インデントされます。

genops(*pickle*)

pickle 内の全ての opcode を取り出すイテレータを返します。このイテレータは (*opcode*, *arg*, *pos*) の三つ組みからなる配列を返します。*opcode* は OpcodeInfo クラスのインスタンスのクラスです。*arg* は *opcode* の引数としてデコードされた Python オブジェクトの値です。*pos* は *opcode* の場所を表す値です。*pickle* は文字列でもファイル類似オブジェクトでもかまいません。

30.12 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。このドキュメントには distutils を拡張する方法も含まれています。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いない場合でも、*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。

三十一章

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:

31.1 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`。

31.2 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ープリタの 2 つの別々のフェーズによって認識されます。ひとつはインター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ー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 度以上出てきてい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。

31.3 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も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。

31.3.1 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っています。

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		
Decorators	nodes	関数のデコレータ表現のリスト
Dict	items	
Discard	expr	
Div	left	
	right	
Ellipsis		
Expression	node	
Exec	expr	
	locals	
	globals	
FloorDiv	left	
	right	

ノードの型	属性	値
For	assign	
	list	
	body	
	else_	
From	modname	
	names	
Function	decorators	Decorators か None
	name	def で定義される名前をあらわす文字列
	argnames	引数をあわらす文字列のリスト
	defaults	デフォルト値のリスト
	flags	xxx
	doc	doc string、文字列あるいは None
	code	関数の本体
GenExpr	code	
GenExprFor	assign	
	iter	
	ifs	
GenExprIf	test	
GenExprInner	expr	
	quals	
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	doc string、文字列あるいは None

ノードの型	属性	値
	node	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	
	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_	
With	expr	
	vars	
	body	
Yield	value	

31.3.2 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。

31.3.3 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(None, '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(None, '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(None, '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(None, '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))))])
```

31.4 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*)

31.5 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。

三十二章

抽象構文木

2.5 で追加された仕様です。

_ast モジュールは、Python アプリケーションで Python の抽象構文木を処理しやすくするものです。Python コンパイラは、現在は構文木への読み込みアクセス機能しか提供していません。つまり、アプリケーションでできるのは Python ソースコードから構文木を作成することだけであり、(それを修正したりした) 構文木からバイトコードを作成することはできないということです。抽象構文そのものは、Python のリリースごとに変化する可能性があります。このモジュールを使用すると、現在の文法をプログラム上で知る助けになるでしょう。

抽象構文木を作成するには、組み込み関数 compile のフラグとして _ast.PyCF_ONLY_AST を渡します。その結果は、_ast.AST を継承したクラスのオブジェクトのツリーとなります。

実際のクラスは Parser/Python.asdl ファイルから派生したものです。これは後ほど示します。抽象構文の左辺のシンボルに対してそれぞれクラスが定義されています (たとえば _ast.stmt や _ast.expr)。また、右辺の各コンストラクタに対してもそれぞれクラスが定義されています。これらのクラスは左辺のツリーのクラスを継承しています。たとえば _ast.BinOp は _ast.expr を継承しています。production rules with alternatives (aka "sums") の場合、左辺は抽象クラスとなります。特定のコンストラクタノードのインスタンスのみが作成されます。

各具象クラスは属性 _fields を持っており、すべての子ノードの名前をそこに保持しています。

具象クラスのインスタンスは、各子ノードに対してそれぞれひとつの属性を持っています。この属性は、文法で定義された型となります。たとえば_ast.BinOp のインスタンスは left という属性を持っており、その型は _ast.expr です。_ast.expr や _ast.stmt のサブクラスのインスタンスにはさらに lineno や col_offset といった属性があります。lineno はソーステキスト上の行番号 (1 から数え始めるので、最初の行の行番号は 1 となります)、そして col_offset はノードが生成した最初のトークンの utf8 バイトオフセットとなります。utf8 オフセットが記録される理由は、パーサが内部で utf8 を使用するからです。

これらの属性が、文法上オプションであると (クエスチョンマークを用いて) マークされている場合は、その値が None となることもあります。属性のが複数の値をとりうる場合 (アスタリスクでマークされている場合) は、値は Python のリストで表されます。

32.1 抽象文法 (Abstract Grammar)

このモジュールでは文字列定数 __version__ を定義しています。これは、以下に示すファイルの subversion リビジョン番号です。

抽象文法は、現在次のように定義されています。

三十三章

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。

formatter 汎用の出力書式化機構およびデバイスインタフェース。

33.1 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。

33.1.1 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_flowing_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_flowing_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字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るので注意してください。

flush_softspace()
以前の `add_flowing_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。

33.1.2 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。

33.1.3 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*)
水平罫線を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 特有のものなので、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_flowing_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。

33.1.4 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。

三十四章

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。

al SGI のオーディオ機能。

AL al モジュールで使われる定数。

cd Silicon Graphics システムの CD-ROM へのインターフェース

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。

FL fl モジュールで使用される定数。

flp 保存された FORMS デザインをロードする関数。

fm SGI ワークステーションの *Font Manager* インターフェース。

gl Silicon Graphics の *Graphics Library* の関数。

DEVICE gl モジュールで使われる定数。

GL gl モジュールで使われる定数。

imgfile SGI imglib ファイルのサポート。

jpeg JPEG ファイルの読み書きを行います。

34.1 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イ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 インターフェー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。

34.1.1 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。

34.1.2 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イタスの情報を返します。

34.2 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 *
```

34.3 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/sc0d4l0'` あるいは `None` です。もし省略したり、`None` なら、ハードウエアが検索されて 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）。

34.3.1 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 `readda()` 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違うため、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。

34.3.2 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*)

readda() 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。データ内にどういう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。

34.4 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もありません。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。

34.4.1 `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。

34.4.2 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()

フォームをディ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
window	int (read-only)	GL ウィンドウの id
w	float	フォームの幅
h	float	フォームの高さ
x	float	フォーム左肩の x 座標
y	float	フォーム左肩の y 座標
deactivated	int	フォームがディアクティベートされているなら非ゼロ
visible	int	フォームが可視なら非ゼロ
frozen	int	フォームが固定されているなら非ゼロ
doublebuf	int	ダブルバッファリングがオンなら非ゼロ

34.4.3 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
`objclass`	int (read-only)	オブジェクトクラス
`type`	int (read-only)	オブジェクトタイプ
`boxtype`	int	ボックスタイプ
`x`	float	左肩の x 座標
`y`	float	左肩の y 座標
`w`	float	幅
`h`	float	高さ
`col1`	int	第 1 の色
`col2`	int	第 2 の色
`align`	int	配置
`lcol`	int	ラベルの色
`lsize`	float	ラベルのフォントサイズ
`label`	string	ラベルの文字列
`lstyle`	int	ラベルのスタイル
`pushed`	int (read-only)	（FORMS ドキュメント参照）
`focus`	int (read-only)	（FORMS ドキュメント参照）
`belowmouse`	int (read-only)	（FORMS ドキュメント参照）
`frozen`	int (read-only)	（FORMS ドキュメント参照）
`active`	int (read-only)	（FORMS ドキュメント参照）
`input`	int (read-only)	（FORMS ドキュメント参照）
`visible`	int (read-only)	（FORMS ドキュメント参照）
`radio`	int (read-only)	（FORMS ドキュメント参照）
`automatic`	int (read-only)	（FORMS ドキュメント参照）

34.5 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 *
```

34.6 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！

34.7 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：*4Sight 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：(*printermatched*、*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。

34.8 `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 インター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返り値は、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返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、返り値はタプルです。もし C の関数が通常の返り値

（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、返り値はタプルの最初に来ます。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（あるいはタプル）です。各座標は (*x*, *y*, *z*) あるいは (*x*, *y*) のタプ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シーケンス（リストあるいはタプル）です。各ペアは座標と、その座標からの法線とのタプルです。各座標と各法線は (*x*, *y*, *z*) からなるタプ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の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.prefposition(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。

34.9 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。

34.10 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。

34.11 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役に立つものなので、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。

34.12 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 JEPG 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.

三十五章

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 5(Solaris バージョン 2) に固有の機能を解説します。

35.1 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。

35.1.1 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)
```

35.2 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。

三十六章

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。

msilib Creation of Microsoft Installer files, and CAB files.

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。

36.1 msilib — Microsoft インストーラーファイルの読み書き

2.5 で追加された仕様です。

msilib モジュールは Microdoft インストーラー (.msi) の作成を支援します。このファイルはしばしば埋め込まれた「キャビネット」ファイル (.cab) を含むので、CAB ファイル作成用の API も暴露します。現在のところ .cab ファイルの読み出しはサポートしていませんが、.msi データベースの読み出しサポートは可能です。

このパッケージの目的は .msi ファイルにある全てのテーブルへの完全なアクセスの提供なので、提供されているものは正直に言って低レベルな API です。このパッケージの二つの主要な応用は distutils の bdist_msi コマンドと、Python インストーラーパッケージそれ自体 (と言いつつ現在は別バージョンの msilib を使っているのですが) です。

パッケージの内容は大きく四つのパートに分けられます。低レベル CAB ルーチン、低レベル MSI ルーチン、少し高レベルの MSI ルーチン、標準的なテーブル構造、の四つです。

FCICreate(*cabname, files*)

新しい CAB ファイルを *cabname* という名前で作ります。*files* はタプルのリストで、それぞれのタプルがディスク上のファイル名と CAB ファイルで付けられるファイル名とからなるものでなければなりません。

ファイルはリストに現れた順番で CAB ファイルに追加されます。全てのファイルは MSZIP 圧縮アルゴリズムを使って一つの CAB ファイルに追加されます。

MSI 作成の様々なステップに対する Python コールバックは現在暴露されていません。

UUIDCreate()

新しい一意識別子の文字列表現を返します。この関数は Windows API の関数 UuidCreate と UuidToString をラップしたものです。

OpenDatabase(*path, persist*)

MsiOpenDatabase を呼び出して新しいデータベースオブジェクトを返します。*path* は MSI ファイルのファイル名です。*persist* は五つの定数 MSIDBOPEN_CREATEDIRECT, MSIDBOPEN_CREATE, MSIDBOPEN_DIRECT, MSIDBOPEN_READONLY, MSIDBOPEN_TRANSACT のどれか一つで、フラグ MSIDBOPEN_PATCHFILE を含めても構いません。これらのフラグの意味は Microsoft のドキュメントを参照してください。フラグに依って既存のデータベースを開いたり新しいのを作ったりします。

CreateRecord(*count*)

MSICreateRecord を呼び出して新しいレコードオブジェクトを返します。*count* はレコードのフィールドの数です。

init_database(*name, schema, ProductName, ProductCode, ProductVersion, Manufacturer*)

name という名前の新しいデータベースを作り、*schema* で初期化し、プロパティ *ProductName*, *ProductCode*, *ProductVersion*, *Manufacturer* をセットして、返します

schema は tables と _Validation_records という属性をもったモジュールオブジェクトでなければなりません。典型的には、msilib.schema を使うべきです。

データベースはこの関数から返された時点でスキーマとバリデーションレコードだけが収められています。

add_data(*database, records*)

全ての *records* を *database* に追加します。*records* はタプルのリストで、それぞれのタプルにはテーブルのスキーマに従ったレコードの全てのフィールドを含んでいるものでなければなりません。オプションのフィールドには None を渡すことができます。

フィールドの値には、整数・長整数・文字列・Binary クラスのインスタンスが使えます。

class **Binary**(*filename*)

Binary テーブル中のエントリーを表わします。add_data を使ってこのクラスのオブジェクトを挿入するときには *filename* という名前のファイルをテーブルに読み込みます。

add_tables(*database, module*)

module の全てのテーブルの内容を *database* に追加します。*module* は *tables* という内容が追加されるべき全てのテーブルのリストと、テーブルごとに一つある実際の内容を持っている属性とを含んでいなければなりません。

この関数は典型的にシーケンステーブルをインストールするのに使われます。

add_stream(*database, name, path*)

database の _Stream テーブルに、ファイル *path* を *name* というストリーム名で追加します。

gen_uuid()

新しい UUID を、 MSI が通常要求する形式 (すなわち、中括弧に入れ、16 進数は大文字) で返します。

参考資料:

FCICreateFile
(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/devnotes/winprog/fcicreate.asp)

UuidCreate
(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/rpc/rpc/uuidcreate.asp)

UuidToString
(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/rpc/rpc/uuidtostring.asp)

36.1.1 データベースオブジェクト

OpenView(*sql*)

MSIDatabaseOpenView を呼び出してビューオブジェクトを返します。*sql* は実行される SQL 命令です。

Commit()

MSIDatabaseCommit を呼び出して現在のトランザクションで保留されている変更をコミットします。

GetSummaryInformation(*count*)

MsiGetSummaryInformation を呼び出して新しいサマリー情報オブジェクトを返します。*count* は更新された値の最大数です。

参考資料:

MSIOpenView
(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/msiopenview.asp)

MSIDatabaseCommit
(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/msidatabasecommit.asp)

MSIGetSummaryInformation
(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/msigetsummaryinformati

36.1.2 ビューオブジェクト

Execute(*[params=None]*)

MSIViewExecute を通してビューに対する SQL 問い合わせを実行します。*params* はオプションのレコードでクエリ中のパラメータトークンの実際の値を与えるものです。

GetColumnInfo(*kind*)

MsiViewGetColumnInfo の呼び出しを通してビューのカラムを説明するレコードを返します。*kind* は MSICOLINFO_NAMES または MSICOLINFO_TYPES です。

Fetch()

MsiViewFetch の呼び出しを通してクエリの結果レコードを返します。

Modify(*kind, data*)

MsiViewModify を呼び出してビューを変更します。*kind* は MSIMODIFY_SEEK, MSIMODIFY_REFRESH, MSIMODIFY_INSERT, MSIMODIFY_UPDATE, MSIMODIFY_ASSIGN, MSIMODIFY_REPLACE, MSIMODIFY_MERGE, MSIMODIFY_DELETE, MSIMODIFY_INSERT_TEMPORARY, MSIMODIFY_VALIDATE, MSIMODIFY_VALIDATE_NEW, MSIMODIFY_VALIDATE_FIELD, MSIMODIFY_VALIDATE_DELETE のいずれかです。

data は新しいデータを表わすレコードでなければなりません。

Close()

MsiViewClose を通してビューを閉じます。

参考資料:

MsiViewExecute
(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/msiviewexecute.asp)

MSIViewGetColumnInfo
(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/msiviewgetcolumninfo.a

MsiViewFetch
(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/msiviewfetch.asp)

MsiViewModify
(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/msiviewmodify.asp)

MsiViewClose
(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/msiviewclose.asp)

36.1.3 サマリー情報オブジェクト

GetProperty (*field*)

MsiSummaryInfoGetProperty を通してサマリーのプロパティを返します。*field* はプロパティ名で、定数 PID_CODEPAGE, PID_TITLE, PID_SUBJECT, PID_AUTHOR, PID_KEYWORDS, PID_COMMENTS, PID_TEMPLATE, PID_LASTAUTHOR, PID_REVNUMBER, PID_LASTPRINTED, PID_CREATE_DTM, PID_LASTSAVE_DTM, PID_PAGECOUNT, PID_WORDCOUNT, PID_CHARCOUNT, PID_APPNAME, PID_SECURITY のいずれかです。

GetPropertyCount ()

MsiSummaryInfoGetPropertyCount を通してサマリープロパティの個数を返します。

SetProperty (*field, value*)

MsiSummaryInfoSetProperty を通してプロパティをセットします。*field* は GetProperty におけるものと同じ値をとります。*value* はプロパティの新しい値です。許される値の型は整数と文字列です。

Persist ()

MsiSummaryInfoPersist を使って変更されたプロパティをサマリー情報ストリームに書き込みます。

参考資料:

MsiSummaryInfoGetProperty
(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/msisummaryinfogetprope

MsiSummaryInfoGetPropertyCount
(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/msisummaryinfogetprope

MsiSummaryInfoSetProperty
(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/msisummaryinfosetprope

MsiSummaryInfoPersist
(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/msisummaryinfopersist.

36.1.4 レコードオブジェクト

GetFieldCount ()

MsiRecordGetFieldCount を通してレコードのフィールド数を返します。

SetString (*field, value*)

MsiRecordSetString を通して *field* を *value* にセットします。*field* は整数、*value* は文字列でなければなりません。

SetStream (*field, value*)

MsiRecordSetStream を通して *field* を *value* という名のファイルの内容にセットします。*field* は整数、*value* は文字列でなければなりません。

SetInteger (*field, value*)

MsiRecordSetInteger を通して *field* を *value* にセットします。*field* も *value* も整数でなければなりません。

ClearData ()

MsiRecordClearData を通してレコードの全てのフィールドを 0 にセットします。

参考資料:

MsiRecordGetFieldCount

(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/msirecordgetfieldcount

MsiRecordSetString

(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/msirecordsetstring.asp

MsiRecordSetStream

(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/msirecordsetstream.asp

MsiRecordSetInteger

(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/msirecordsetinteger.as

MsiRecordClear

(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/msirecordclear.asp)

36.1.5 エラー

全ての MSI 関数のラッパーは `MsiError` を送出します。例外の内部の文字列がより詳細な情報を含んでいます。

36.1.6 CAB オブジェクト

class CAB (*name*)

`CAB` クラスは CAB ファイルを表わすものです。MSI 構築中、ファイルは `Files` テーブルと CAB ファイルとに同時に追加されます。そして、全てのファイルを追加し終えたら、CAB ファイルは書き込まれることが可能になり、MSI ファイルに追加されます。

name は MSI ファイル中の CAB ファイルの名前です。

append (*full, logical*)

パス名 *full* のファイルを CAB ファイルに *logical* という名で追加します。*logical* という名が既に存在したならば、新しいファイル名が作られます。

ファイルの CAB ファイル中のインデクスと新しいファイル名を返します。

append (*database*)

CAB ファイルを作り、MSI ファイルにストリームとして追加し、`Media` テーブルに送り込み、作ったファイルはディスクから削除します。

36.1.7 ディレクトリオブジェクト

class Directory (*database, cab, basedir, physical, logical, default, component,* [*componentflags*])

新しいディレクトリを Directory テーブルに作成します。ディレクトリには各時点で現在のコンポーネントがあり、それは `start_component` を使って明ら様に作成されたかまたは最初にファイルが追加された際に暗黙裡に作成されたものです。ファイルは現在のコンポーネントと cab ファイルに追加されます。ディレクトリを作成するには親ディレクトリオブジェクト (`None` でも可)、物理的ディレクトリへのパス、論理的ディレクトリ名を指定する必要があります。*default* はディレクトリテーブルの DefaultDir スロットを指定します。*componentflags* は新しいコンポーネントが得るデフォルトのフラグを指定します。

start_component ([*component*[, *feature*[, *flags*[, *keyfile*[, *uuid*]]]]])

エントリを Component テーブルに追加し、このコンポーネントをこのディレクトリの現在のコンポーネントにします。もしコンポーネント名が与えられなければディレクトリ名が使われます。*feature* が与えられなければ、ディレクトリのデフォルトフラグが使われます。*keyfile* が与えられなければ、Component テーブルの KeyPath は null のままになります。

add_file (*file*[, *src*[, *version*[, *language*]]])
ファイルをディレクトリの現在のコンポーネントに追加します。このとき現在のコンポーネントがなければ新しいものを開始します。デフォルトではソースとファイルテーブルのファイル名は同じになります。*src* ファイルが与えられたならば、それば現在のディレクトリから相対的に解釈されます。オプションで *version* と *language* を File テーブルのエントリ用に指定することができます。

glob (*pattern*[, *exclude*])
現在のコンポーネントに glob パターンで指定されたファイルのリストを追加します。個々のファイルを *exclude* リストで除外することができます。

remove_pyc ()
アンインストールの際に `.pyc`/`.pyo` を削除します。

参考資料:

Directory Table
(http://msdn.microsoft.com/library/en-us/msi/setup/directory_table.asp)

File Table
(http://msdn.microsoft.com/library/en-us/msi/setup/file_table.asp)

Component Table
(http://msdn.microsoft.com/library/en-us/msi/setup/component_table.asp)

FeatureComponents Table
(http://msdn.microsoft.com/library/en-us/msi/setup/featurecomponents_table.asp)

36.1.8 フィーチャー

class **Feature** (*database, id, title, desc, display*[, *level=1*[, *parent*[*{*], *directory*[, *attributes=0*]]])
id, *parent.id*, *title*, *desc*, *display*, *level*, *directory*, *attributes* の値を使って、新しいレコードを `Feature` テーブルに追加します。出来上がったフィーチャーオブジェクトは `Directory` の `start_component` メソッドに渡すことができます。

set_current ()
このフィーチャーを `msilib` の現在のフィーチャーにします。フィーチャーが明ら様に指定されない限り、新しいコンポーネントが自動的にデフォルトのフィーチャーに追加されます。

参考資料:

Feature Table
(http://msdn.microsoft.com/library/en-us/msi/setup/feature_table.asp)

36.1.9 GUI クラス

`msilib` モジュールは MSI データベースの中の GUI テーブルをラップする幾つかのクラスを提供しています。しかしながら、標準で提供されるユーザーインタフェースはありません。インストールする Python パッケージに対するユーザーインタフェース付きの MSI ファイルを作成するには `bdist_msi` を使ってください。

class **Control** (*dlg, name*)
ダイアログコントロールの基底クラス。*dlg* はコントロールの属するダイアログオブジェクト、*name* はコントロールの名前です。

event (*event, argument*[, *condition* = *"1"*[, *ordering*]])
このコントロールの `ControlEvent` テーブルにエントリを作ります。

mapping (*event, attribute*)

このコントロールの EventMapping テーブルにエントリを作ります。

condition (*action, condition*)

このコントロールの ControlCondition テーブルにエントリを作ります。

class RadioButtonGroup (*dlg, name, property*)

name という名前のラジオボタンコントロールを作成します。*property* はラジオボタンが選ばれたときにセットされるインストーラープロパティです。

add (*name, x, y, width, height, text* [, *value*])

グループに *name* という名前で、座標 *x*, *y* に大きさが *width*, *height* で *text* というラベルの付いたラジオボタンを追加します。*value* が省略された場合、デフォルトは *name* になります。

class Dialog (*db, name, x, y, w, h, attr, title, first, default, cancel*)

新しい Dialog オブジェクトを返します。Dialog テーブルの中に指定された座標、ダイアログ属性、タイトル、最初とデフォルトとキャンセルコントロールの名前を持ったエントリが作られます。

control (*name, type, x, y, width, height, attributes, property, text, control_next, help*)

新しい Control オブジェクトを返します。Control テーブルに指定されたパラメータのエントリが作られます。

これは汎用のメソッドで、特定の型に対しては特化したメソッドが提供されています。

text (*name, x, y, width, height, attributes, text*)

Text コントロールを追加して返します。

bitmap (*name, x, y, width, height, text*)

Bitmap コントロールを追加して返します。

line (*name, x, y, width, height*)

Line コントロールを追加して返します。

pushbutton (*name, x, y, width, height, attributes, text, next_control*)

PushButton コントロールを追加して返します。

radiogroup (*name, x, y, width, height, attributes, property, text, next_control*)

RadioButtonGroup コントロールを追加して返します。

checkbox (*name, x, y, width, height, attributes, property, text, next_control*)

CheckBox コントロールを追加して返します。

参考資料:

Dialog Table

(http://msdn.microsoft.com/library/en-us/msi/setup/dialog_table.asp)

Control Table

(http://msdn.microsoft.com/library/en-us/msi/setup/control_table.asp)

Control Types

(http://msdn.microsoft.com/library/en-us/msi/setup/controls.asp)

ControlCondition Table

(http://msdn.microsoft.com/library/en-us/msi/setup/controlcondition_table.asp)

ControlEvent Table

(http://msdn.microsoft.com/library/en-us/msi/setup/controlevent_table.asp)

EventMapping Table

(http://msdn.microsoft.com/library/en-us/msi/setup/eventmapping_table.asp)

RadioButton Table

(http://msdn.microsoft.com/library/en-us/msi/setup/radiobutton_table.asp)

36.1.10 事前に計算されたテーブル

msilib はスキーマとテーブル定義だけから成るサブパッケージをいくつか提供しています。現在のところ、これらの定義は MSI バージョン 2.0 に基づいています。

schema
: これは MSI 2.0 用の標準 MSI スキーマで、テーブル定義のリストを提供する *tables* 変数と、MSI バリデーション用のデータを提供する *_Validation_records* 変数があります。

sequence
: このモジュールは標準シーケンステーブルのテーブル内容を含んでいます。*AdminExecuteSequence*, *AdminUISequence*, *AdvtExecuteSequence*, *InstallExecuteSequence*, *InstallUISequence* が含まれています。

text
: このモジュールは標準的なインストーラーのアクションのための UIText および ActionText テーブルの定義を含んでいます。

36.2 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。

36.2.1 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。連接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 回再試行した後で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。

36.2.2 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。

36.2.3 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。

36.3 _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。

このメソッドを使って (または `handle.Close()` によって) *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てはなりません。

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つ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_*` 定数のうちの一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_*` 定数のうちの一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てはなりません。

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、*fileName* 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いてはいけません。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。

36.3.1 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理するものは全て整数も受理しますが、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()` メソッド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。

36.4 `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いくつかの関数と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:

PlaySound() ***name***	**対応するコントロールパネルでの音声名**
'SystemAsterisk'	Asterisk
'SystemExclamation'	Exclamation
'SystemExit'	Exit Windows
'SystemHand'	Critical Stop
'SystemQuestion'	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。

付録 A

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 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。

ドキュメント化されていないフレームワークはありません。

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)。

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。

これらの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。

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。)

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。

付録 B

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). 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 on the left side 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). 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.

付録 C

歴史とライセンス

C.1 Python の歴史

Python は 1990 年代の始め、オランダにある Stichting Mathematisch Centrum (CWI, http://www.cwi.nl/ 参照) で Guido van Rossum によって ABC と呼ばれる言語の後継言語として生み出されました。その後多くの人々が Python に貢献していますが、Guido は今日でも Python 製作者の先頭に立っています。

1995 年、 Guido は米国ヴァージニア州レストンにある Corporation for National Reserch Initiatives (CNRI, http://www.cnri.reston.va.us/ 参照) で Python の開発に携わり、いくつかのバージョンをリリースしました。

2000 年 3 月、 Guido と Python のコア開発チームは BeOpen.com に移り、 BeOpen PythonLabs チームを結成しました。同年 10 月、PythonLabs チームは Digital Creations (現在の Zope Corporation, http://www.zope.com/ 参照) に移りました。 そして 2001 年、Python に関する知的財産を保有するための非営利組織 Python Software Foundation (PSF、 http://www.python.org/psf/ 参照) を立ち上げました。このとき Zope Corporation は PSF の賛助会員になりました。

Python のリリースは全てオープンソース (オープンソースの定義は http://www.opensource.org/ を参照してください) です。歴史的にみて、ごく一部を除くほとんどの Python リリースは GPL 互換になっています; 各リリースについては下表にまとめてあります。

リリース	ベース	年	権利	GPL 互換
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
2.3.3	2.3.2	2003	PSF	yes
2.3.4	2.3.3	2004	PSF	yes
2.3.5	2.3.4	2005	PSF	yes
2.4	2.3	2004	PSF	yes
2.4.1	2.4	2005	PSF	yes
2.4.2	2.4.1	2005	PSF	yes
2.4.3	2.4.2	2006	PSF	yes
2.5	2.4	2006	PSF	yes

注意: 「GPL 互換」という表現は、Python が GPL で配布されているという意味ではありません。Python のライセンスは全て、 GPL と違い、変更したバージョンを配布する際に変更をオープンソースにしなくてもかまいません。GPL 互換のライセンスの下では、GPL でリリースされている他のソフトウェアと Python を組み合わせられますが、それ以外のライセンスではそうではありません。

Guido の指示の下、これらのリリースを可能にしてくださった多くのボランティアのみなさんに感謝します。

C.2 Terms and conditions for accessing or otherwise using Python

PSF LICENSE AGREEMENT FOR PYTHON 2.5

1. This LICENSE AGREEMENT is between the Python Software Foundation (“PSF”), and the Individual or Organization (“Licensee”) accessing and otherwise using Python 2.5 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 2.5 alone or in any derivative version, provided, however, that PSF’s License Agreement and PSF’s notice of copyright, i.e., “Copyright © 2001-2006 Python Software Foundation; All Rights Reserved” are retained in Python 2.5 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 2.5 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 2.5.

4. PSF is making Python 2.5 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 2.5 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.5 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.5, 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 2.5, 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

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

C.3.1 Mersenne Twister

The `_random` module includes code based on a download from `http://www.math.keio.ac.jp/~matumoto/MT2002/emt19937ar.html`. The following are the verbatim comments from the original code:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.keio.ac.jp/matumoto/emt.html
email: matumoto@math.keio.ac.jp
```

C.3.2 Sockets

The `socket` module uses the functions, `getaddrinfo`, and `getnameinfo`, which are coded in separate source files from the WIDE Project, http://www.wide.ad.jp/about/index.html.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
GAI_ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR GAI_ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON GAI_ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN GAI_ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Floating point exception control

The source for the `fpectl` module includes the following notice:

C.3.4 MD5 message digest algorithm

The source code for the `md5` module contains the following notice:

```
  Copyright (C) 1999, 2002 Aladdin Enterprises.  All rights reserved.

  This software is provided 'as-is', without any express or implied
  warranty.  In no event will the authors be held liable for any damages
  arising from the use of this software.

  Permission is granted to anyone to use this software for any purpose,
  including commercial applications, and to alter it and redistribute it
  freely, subject to the following restrictions:

  1. The origin of this software must not be misrepresented; you must not
     claim that you wrote the original software. If you use this software
     in a product, an acknowledgment in the product documentation would be
     appreciated but is not required.
  2. Altered source versions must be plainly marked as such, and must not be
     misrepresented as being the original software.
  3. This notice may not be removed or altered from any source distribution.

  L. Peter Deutsch
  ghost@aladdin.com

  Independent implementation of MD5 (RFC 1321).

  This code implements the MD5 Algorithm defined in RFC 1321, whose
  text is available at
http://www.ietf.org/rfc/rfc1321.txt
  The code is derived from the text of the RFC, including the test suite
  (section A.5) but excluding the rest of Appendix A.  It does not include
  any code or documentation that is identified in the RFC as being
  copyrighted.

  The original and principal author of md5.h is L. Peter Deutsch
  <ghost@aladdin.com>.  Other authors are noted in the change history
  that follows (in reverse chronological order):

  2002-04-13 lpd Removed support for non-ANSI compilers; removed
references to Ghostscript; clarified derivation from RFC 1321;
now handles byte order either statically or dynamically.
  1999-11-04 lpd Edited comments slightly for automatic TOC extraction.
  1999-10-18 lpd Fixed typo in header comment (ansi2knr rather than md5);
added conditionalization for C++ compilation from Martin
Purschke <purschke@bnl.gov>.
  1999-05-03 lpd Original version.
```

C.3.5 Asynchronous socket services

The `asynchat` and `asyncore` modules contain the following notice:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.6 Cookie management

The `Cookie` module contains the following notice:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.7 Profiling

The `profile` and `pstats` modules contain the following notice:

```
Copyright 1994, by InfoSeek Corporation, all rights reserved.
Written by James Roskind

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.
```

C.3.8 Execution tracing

The `trace` module contains the following notice:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.9 UUencode and UUdecode functions

The `uu` module contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with python standard
```

C.3.10 XML Remote Procedure Calls

The `xmlrpclib` module contains the following notice:

The XML-RPC client interface is

付録 D

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

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

D.3 2.4 差分翻訳者一覧 (敬称略)

Akihiro Takizawa, G.Yoshida, Yasushi MASUDA, 新山祐介, 森若和雄

D.4 2.5 差分翻訳者一覧 (敬称略)

Kazuo Moriwaka, TAKAGI Masahiro, MATSUI Tetsushi, Minami Masanori, Hiroshi Okagawa, hkurosawa, Naoki INADA, Keisuke Urago, Shinya Okano, Toshiyuki Kawanishi

モジュール索引

索引