

---

# Python チュートリアル

リリース 2.3.4

Guido van Rossum

Fred L. Drake, Jr., editor

日本語訳: Python ドキュメント翻訳プロジェクト

平成 17 年 3 月 29 日

**PythonLabs**

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

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

Copyright © 2000 BeOpen.com. All rights reserved.

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

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

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

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

## 概要

Python は簡単に学ぶことができ、それでいて強力な言語の一つです。Python には高レベルなデータ構造が効率的に実装されており、オブジェクト指向プログラミングに対しても、単純でありながら効果的なアプローチをしています。洗練された文法とデータ型を動的に決定する機能は、そのインタプリタの特性とあいまって Python を理想的なスクリプトプログラミング言語にするとともに、多くのプラットフォームにおける幅広い対象領域において迅速にアプリケーションを開発できるようにしています。

Python インタプリタ自体と拡張可能な標準ライブラリは、ソースコード、または多くの主要な計算機環境向けのバイナリ形式として、いずれも Python Web サイト <http://www.python.org/> から無料で入手でき、かつ無料で配布することができます。このサイトではまた、無料で手に入るたくさんのサードパーティ製 Python モジュール、プログラム、ツール類、追加のドキュメントについて、その配布物やポイントが置かれています。

Python インタプリタは C 言語や C++ 言語 (あるいは C 言語から呼び出すことができるその他の言語) で実装された新たな関数やデータ構造を組み込んで拡張することが簡単にできます。Python はまた、カスタマイズ可能なアプリケーションを作るための拡張機能記述言語としてぴったりです。

このチュートリアルでは、Python の言語仕様と仕組みについて、その基本的な概念と機能をざっと紹介します。例として挙げた題について自分の手で体験するには手元に Python インタプリタがあると便利ですが、どの題もそれ自体で完結した話になっているので、このチュートリアルをオフラインで読むことも可能です。

標準のオブジェクトやモジュールの記述については、ドキュメント「Python ライブラリリファレンス (*Python Library Reference*)」を参照してください。C 言語や C++ 言語で拡張モジュールを書くなら、「Python インタプリタの拡張と埋め込み (*Extending and Embedding the Python Interpreter*)」、および「Python/C API リファレンス (*Python/C API Reference*)」を参照してください。Python について広く深くカバーしている書籍もいくつかあります。

このチュートリアルは網羅的な内容ではありません。また、個別の機能について全てをカバーしているわけでもありません。通常使われている機能すら全てカバーされていません。その代わり、このチュートリアルでは Python の特筆すべき機能をたくさん紹介して、この言語の持ち味やスタイルについて好印象を持ってもらうつもりです。このチュートリアルを読んだ後には、読者のみなさんは Python のモジュールやプログラムを読み書きできるようになり、「Python ライブラリリファレンス (*Python Library Reference*)」に記述されているさまざまな Python ライブラリモジュールについて学ぶ準備ができていることでしょう。



# 目次

第 1 章	Python への意欲を高める	1
1.1	ここからどこへ	2
第 2 章	Python インタプリタを使う	3
2.1	インタプリタを起動する	3
2.2	インタプリタとその環境	4
第 3 章	形式ばらない Python の紹介	7
3.1	Python を電卓として使う	7
3.2	プログラミングへの第一歩	17
第 4 章	その他の制御フローツール	19
4.1	if 文	19
4.2	for 文	19
4.3	range() 関数	20
4.4	break 文と continue 文とループの else 節	20
4.5	pass 文	21
4.6	関数を定義する	21
4.7	関数定義についてもう少し	23
第 5 章	データ構造	29
5.1	リスト型についてもう少し	29
5.2	del 文	33
5.3	タプルと配列	33
5.4	辞書	34
5.5	ループのテクニック	35
5.6	条件についてもう少し	36
5.7	配列とその他の型の比較	36
第 6 章	モジュール	39
6.1	モジュールについてもう少し	40
6.2	標準モジュール	42
6.3	dir() 関数	42
6.4	パッケージ	44
第 7 章	入力と出力	49
7.1	ファンシーな出力の書式化	49

7.2	ファイルを読み書きする	52
<b>第 8 章</b>	<b>エラーと例外</b>	<b>55</b>
8.1	構文エラー	55
8.2	例外	55
8.3	例外を処理する	56
8.4	例外を送出する	58
8.5	ユーザ定義の例外	59
8.6	後片付け動作を定義する	60
<b>第 9 章</b>	<b>クラス</b>	<b>63</b>
9.1	用語について一言	63
9.2	Python のスコープと名前空間	64
9.3	クラス初見	65
9.4	いろいろな注意点	68
9.5	継承	69
9.6	プライベート変数	71
9.7	残りのはしばし	71
9.8	イテレータ (iterator)	73
9.9	ジェネレータ (generator)	74
<b>第 10 章</b>	<b>標準ライブラリの簡単なツアー</b>	<b>77</b>
10.1	オペレーティングシステムへのインタフェース	77
10.2	ファイルのワイルドカード表記	77
10.3	コマンドライン引数	78
10.4	エラー出力のリダイレクトとプログラムの終了	78
10.5	文字列のパターンマッチング	78
10.6	数学	78
10.7	インターネットへのアクセス	79
10.8	日付と時刻	79
10.9	データ圧縮	80
10.10	パフォーマンスの計測	80
10.11	品質管理	81
10.12	一揃いあつらえ済み	81
<b>第 11 章</b>	<b>さあ何を？</b>	<b>83</b>
<b>付 録 A</b>	<b>対話入力編集とヒストリ置換</b>	<b>85</b>
A.1	行編集	85
A.2	ヒストリ置換	85
A.3	キー割り当て	86
A.4	解説	87
<b>付 録 B</b>	<b>浮動小数点演算、その問題と制限</b>	<b>89</b>
B.1	表現エラー	91
<b>付 録 C</b>	<b>歴史とライセンス</b>	<b>95</b>
C.1	History of the software	95
C.2	Terms and conditions for accessing or otherwise using Python	96

付録 D 日本語訳について	99
D.1 このドキュメントについて . . . . .	99
D.2 翻訳者一覧 (敬称略) . . . . .	99
付録 E 用語集	101
索引	105





# Python への意欲を高める

巨大なシェルスクリプトを書いたことがあるなら、「もう一つ機能を追加したいのだけれど、プログラムはすでに遅くなりすぎているし、巨大すぎるし、複雑すぎる...」、あるいは「追加したい機能にはシステムコールか、C 言語からしかアクセスできない別の関数が必要だ...」といった気持ちを理解できるでしょう。

身近な課題というのはたいてい、スクリプトを C 言語で書き直すほど大したことではありません; そしてその課題にはおそらく可変長の文字列や (ファイル名をソートしたリストのような) 他のデータ型が必要で、これはシェルスクリプトで実装するのは簡単だが C 言語でやるのは大仕事になるし、もしかするとあなた自身はそれほど C 言語に詳しくはないかもしれません。

別の状況として、「おそらくいくつかの C 言語のライブラリを使って仕事をしなければならないのだけれど、C 言語でお決まりの、プログラム作成 / コンパイル / テスト / 再コンパイルのサイクルは時間がかかりすぎる。もっと早くプログラムを開発したい」、あるいは、「機能拡張のための言語を使えるようなプログラムを書いたのだけれど、新しく言語を設計して、その言語のためのインタプリタを実装して、テストして、自分のアプリケーションと連動させるまではやりたくない」といったこともあるでしょう。

これらのケースでは、Python はまさにあなたにぴったりの言語です。Python は使い方は単純ですが、シェルスクリプトよりも多くのデータ構造を扱うことができ、大規模なプログラムを行うためのサポートを持った本物のプログラム言語です。一方で、Python は C 言語よりもはるかに多くのエラーチェック機構を提供しています。また、超高水準 (*very-high-level*) のプログラム言語で、柔軟性のある配列や辞書といった高水準データ型が組み込まれています。これらの型を効率よく処理するための実装を C 言語でやろうとすると数日がかかりになってしまうでしょう。比較的汎用性の高いデータ型を持つため、Python は *Awk* や *Perl* さえはるかにしのぐ広い領域の問題に適用することができます。その上、Python でやれば、多くの仕事は上述の言語と少なくとも同じくらいに簡単にこなせます。

Python ではプログラムをモジュールに分割して他の Python プログラムで再利用することができます。Python には膨大な標準モジュールが付属していて、プログラムを作る上での基盤として — あるいは Python プログラミングを学ぶために — 利用することができます。組み込みモジュールではまた、ファイル I/O、システムコール、ソケットといった機能や、Tk のようなグラフィカルユーザインタフェースツールキットを使うためのインタフェースさえも提供しています。

Python はインタプリタ言語です。このため、コンパイルやリンクが必要ないので、プログラムを開発する際にかなりの時間を節約できます。インタプリタは対話的に利用することもできます。対話的インタプリタの利用によって、この言語の様々な機能について実験してみたり、やっつけ仕事のプログラムを書いたり、ボトムアップでプログラムを開発する際に関数をテストしたりといったことが簡単にできます。便利な電卓にもなります。

Python では、とてもコンパクトで読みやすいプログラムを書くことができます。Python で書かれたプログラムは大抵、同じ機能を提供する C 言語や C++ 言語のプログラムよりもはるかに短くなります。これには以下のようないくつかの理由があります:

- 高レベルのデータ型によって、複雑な操作を一つの実行文で表現することができます;
- 実行文のグループ化はグループの開始や終了の括弧を使う代わりにインデントで行うことができます;

- 変数や引数の宣言が不要になります。

Python は 拡張することが出来ます: C 言語でプログラムを書く方法を知っているなら、新たな組み込み関数やモジュールをインタプリタに追加することは簡単です。これによって、処理速度を決定的に左右する操作を最大速度で動作するように実現したり、(ベンダ特有のグラフィクスライブラリのように) バイナリ形式でしか手に入らないライブラリを Python にリンクしたりできます。その気になれば、Python インタプリタを C で書かれたアプリケーションにリンクして、アプリケーションに対する拡張言語や命令言語として使うこともできます。

ところで、この言語は BBC のショー番組、“モンティパイソンの空飛ぶサーカス (Monty Python's Flying Circus)” から取ったもので、気味の悪い爬虫類とは関係ありません。このドキュメントにあるモンティパイソンの寸劇を参照するのは、よいどころか、むしろ奨励されています！

## 1.1 ここからどこへ

さて、皆さんはもう Python にとてもそそられて、もうちょっと詳しく調べてみたくなったはずです。プログラミング言語を習得する最良の方法は使ってみることですから、ここでやってみましょう。

次の章では、まずインタプリタを使うための機微を説明します。これはさして面白みのない情報なのですが、後に説明する例題を試してみる上で不可欠なことです。

チュートリアルに残りの部分では、Python プログラム言語と実行システムの様々な機能を例題を交えて紹介します。単純な式、実行文、データ型から始めて、関数とモジュールを経て、最後には例外処理やユーザ定義クラスといったやや高度な概念にも触れます。

# Python インタプリタを使う

## 2.1 インタプリタを起動する

Python が使える計算機なら、インタプリタはたいてい `/usr/local/bin/python` にインストールされています; UNIX シェルのサーチパスに `/usr/local/bin` を入れれば、シェルで

```
python
```

とコマンドを入力すれば使えるようになります。インストールする際にどのディレクトリに Python インタプリタを入れるかをオプションで指定できるので、インタプリタは他のディレクトリにあるかもしれませんが; 身近な Python の導師 (guru) か、システム管理者に聞いてみてください。(例えば、その他の場所として `/usr/local/python` が一般的です。)

ファイル終端文字 (UNIX では Control-D、DOS や Windows では Control-Z) を一次プロンプト (primary prompt) に入力すると、インタプリタは終了状態ゼロで終了します。もしこの操作がうまく働かないなら、コマンド: `import sys; sys.exit()` を入力することでインタプリタを終了することができます。

通常、インタプリタの行編集機能は、あまり洗練されたものではありません。UNIX システムでは、インタプリタをインストールした誰かが GNU readline ライブラリのサポートを有効にしていれば、洗練された対話的行編集やヒストリ機能が追加されます。コマンドライン編集機能がサポートされているかを最も手っ取り早く調べる方法は、おそらく最初に表示された Python プロンプトに Control-P を入力してみることでしょう。ピープ音が鳴るなら、コマンドライン編集機能があります; 編集キーについての解説は付録 A を参照してください。何も起こらないように見えるか、`^P` がエコーバックされるなら、コマンドライン編集機能は利用できません; 現在編集の行から文字を削除するにはバックスペースを使うしかありません。

インタプリタはさながら UNIX シェルのように働きます: 標準入力端末に接続された状態で呼び出されると、コマンドを対話的に読み込んで実行します; ファイル名を引数にしたり、標準入力からファイルを入力すると、インタプリタはファイルから スクリプト を読み込んで実行します。

インタプリタを起動する第二の方法は `python -c command [arg] ...` です。この形式では、シェルの `-c` オプションと同じように、`command` に指定した文を実行します。Python 文はしばしばスペースその他のシェルにとって特殊な意味をもつ文字を含むので、`command` 全体を二重引用符を囲っておくのがベストです。

`'python file'` と `'python <file'` の違いに注意してください。後者の場合、`input()` や `raw_input()` を呼び出した時のようにプログラム自体から入力が必要されると、その入力はファイルから調達されます。プログラムの実行が開始される前にファイルはすでにパーザによってファイルの終端まで読み込まれているので、入力はすぐにファイル終端に到達します。前者の場合 (大抵はこちらの方が望ましい動作です)、入力には Python インタプリタの標準入力に接続された何らかのファイルまたはデバイスが充てられます。

スクリプトファイルが使われた場合、スクリプトを走らせて、そのまま対話モードに入れると便利なこ

とがあります。これは `-i` をスクリプトの前に追加することで実現できます。(前の段落で述べたのと同じ理由から、スクリプトが標準入力から読まれた場合にはこのオプションはうまく働きません。)

### 2.1.1 引数の受け渡し

インタプリタがスクリプト名と付加な引数を受け取ると、それらは変数 `sys.argv` としてスクリプトに渡されます。これは文字列からなるリストになります。リストの長さは少なくとも 1 です; スクリプト名も引数も与えられなければ、`sys.argv[0]` は空の文字列になります。スクリプト名として `'-'` (標準入力を意味します) を与えると、`sys.argv[0]` は `'-'` に設定されます。`-c command` を使うと、`sys.argv[0]` は `'-c'` に設定されます。`-c command` より後ろにあるオプションは、Python インタプリタがオプションを処理する際には使われませんが、`command` から扱えるように `sys.argv` には残されます。

### 2.1.2 対話モード

命令文を `tty` から読み取っているときには、インタプリタは対話モード (*interactive mode*) であるといえます。このモードでは、インタプリタは一次プロンプト (*primary prompt*) を表示して次のコマンドを入力するよう促します。一次プロンプトは普通、三つの大なり記号 (`>>>`) です; 継続する行 (*continuation line*) に対しては、インタプリタは二次プロンプト (*secondary prompt*) を表示します。これはデフォルトでは三つのドット (`...`) です。インタプリタは、最初のプロンプトを出す前にバージョン番号と著作権表示から始まる歓迎のメッセージを出力します。

```
python
Python 1.5.2b2 (#1, Feb 28 1999, 00:02:06) [GCC 2.8.1] on sunos5
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>>
```

継続する行は、複数の行から構成される構文を入力するときに必要です。例として、以下の `if` 文を見てください。

```
>>> the_world_is_flat = 1
>>> if the_world_is_flat:
...     print "Be careful not to fall off!"
...
Be careful not to fall off!
```

## 2.2 インタプリタとその環境

### 2.2.1 エラー処理

エラーが発生すると、インタプリタはエラーメッセージとスタックトレース (*stack trace*) を出力します。対話モードにいるときは、インタプリタは一次プロンプトに戻ります; 入力ファイルからきているときには、インタプリタはスタックトレースを出力した後、非ゼロの終了状態で終了します。(try 文の `except` 節で処理された例外は、ここでいうエラーにはあたりません。) いくつかのエラーは無条件に致命的であり、非ゼロの終了状態となるプログラムの終了を引き起こします; これにはインタプリタ内部の矛盾やある種のメモリ枯渇が当てはまります。エラーメッセージは全て標準エラー出力ストリームに書き込まれます; これに対して、実行した命令からの通常出力される内容は標準出力に書き込まれます。

割り込み文字 (*interrupt character*、普通は `Control-C` か `DEL`) を一次または二次プロンプトに対して打鍵

すると、入力を取り消されて一次プロンプトに戻ります。<sup>1</sup> コマンドの実行中に割り込み文字を打鍵すると `KeyboardInterrupt` 例外が送出されます。この例外は `try` 文で処理できます。

## 2.2.2 実行可能な Python スクリプト

BSD 風の UNIX システムでは、Python スクリプトはシェルスクリプトのようにして直接実行可能にできます。これを行うには、以下の行

```
#!/usr/bin/env python
```

(ここではインタプリタがユーザの `PATH` 上にあると仮定しています) をスクリプトの先頭に置き、スクリプトファイルに実行可能モードを与えます。`#!` はファイルの最初の 2 文字でなければなりません。プラットフォームによっては、この最初の行を終端する改行文字が Mac OS 形式 (`'\r'`) や Windows 形式 (`'\r\n'`) ではなく、UNIX 形式でなければならないことがあります。ハッシュまたはポンド文字、すなわち `#` は、Python ではコメントを書き始めるために使われているので注意してください。

`chmod` コマンドを使えば、スクリプトに実行モード (または実行権限) を与えることができます:

```
$ chmod +x myscript.py
```

## 2.2.3 ソースコードの文字コード方式 (encoding)

ASCII 形式でない文字コード化方式 (エンコーディング: encoding) を Python ソースコードファイル中使用することができます。最良の方法は、`#!` 行の直後に一行かそれ以上の特殊なコメントを挿入して、ソースファイルのエンコードを指定するということです:

```
# -*- coding: iso-8859-1 -*-
```

このように宣言しておく、ソースファイル中の全ての文字は `iso-8859-1` でエンコードされているものとして扱われ、Unicode 文字列リテラルを指定したエンコードで直接記述することができます。利用可能なエンコードのリストは *Python* ライブラリリファレンスの `codecs` の節にあります。

利用しているエディタがファイルを UTF-8 バイト整列記号 (通称 BOM: Byte Order Mark) 付きの UTF-8 で保存できる場合、エンコード宣言の代わりに使うことができます。IDLE は `Options/General/Default Source Encoding/UTF-8` が設定されている場合、UTF-8 でエンコードされたファイルの識別機能をサポートします。UTF-8 シグネチャは (2.2 以前の) 古い Python リリースでは解釈されず、オペレーティングシステムは `#!` ファイルを判別できなくなるので注意してください。

UTF-8 を (シグネチャやエンコード宣言を行って) 使うと、世界中のほとんどの言語で使われている文字を文字列リテラルやコメントの中に同時に使うことができます。識別子に対する非 ASCII 文字の使用はサポートされていません。全ての文字を正しく表示できるようにするには、使っているエディタがファイルを UTF-8 であると認識することができなければならず、かつファイル内で使われている全ての文字をサポートするようなフォントを使わなければなりません。

---

<sup>1</sup> GNU Readline パッケージに関する問題のせいで妨げられることがあります。

## 2.2.4 対話モード用の起動時実行ファイル

Python を対話的に使うときには、インタプリタが起動する度に実行される何らかの標準的なコマンドがあると便利がよくあります。これを行うには、PYTHONSTARTUP と呼ばれる環境変数を、インタプリタ起動時に実行されるコマンドが入ったファイル名に設定します。この機能は UNIX シェルの `‘.profile’` に似ています。

このファイルは対話セッションのときだけ読み出されます。Python がコマンドをスクリプトから読み出しているときや、`‘/dev/tty’` がコマンドの入力元として明示的に指定されている (この場合対話的セッションのように動作します) わけではない場合にはこのファイルは読み出されません。ファイル内のコマンドは、対話的コマンドが実行される名前空間と同じ名前空間内で実行されます。このため、ファイル内で定義されていたり import されたオブジェクトは、限定子をつけなくても対話セッション内で使うことができます。また、このファイル内で `sys.ps1` や `sys.ps2` を変更して、プロンプトを変更することもできます。

もし現在のディレクトリから追加的なスタートアップファイルを読み出したいのなら、グローバルのスタートアップファイルの中で `‘if os.path.isfile(‘.pythonrc.py’): execfile(‘.pythonrc.py’)’` のようなコードのプログラムを書くことができます。スクリプト中でスタートアップファイルを使いたいのなら、以下のようにしてスクリプト中で明示的に実行しなければなりません:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    execfile(filename)
```



## 形式ばらない Python の紹介

以下の例では、入力と出力は(‘>>> ’や‘... ’)といったプロンプトがあるかないかで区別します: 例どおりに実行するなら、プロンプトが表示されているときに、例中のプロンプトよりも後ろの内容全てをタイプ入力しなければなりません; プロンプトが先頭でない行はインタプリタからの出力です<sup>1</sup>。

例中には二次プロンプトだけが表示されている行がありますが、これは空行を入力しなければならないことを意味するので注意してください; 空行の入力は複数の行からなる命令の終わりをインタプリタに教えるために使われます。

このマニュアルにある例の多くは、対話プロンプトで入力されるものでもコメントを含んでいます。Python におけるコメント文はハッシュ文字 ‘#’ で始まり、物理行の終わりまで続きます。コメントは行の先頭にも、空白やコードの後にも書くことができますが、文字列リテラル (string literal) の内部に置くことはできません。文字列リテラル中のハッシュ文字はただのハッシュ文字です。

例:

```
# これは1 番目のコメント
SPAM = 1                # そしてこれは2 番目のコメント
                        # ... そしてこれは3 番目!
STRING = "# これはコメントではありません。"
```

### 3.1 Python を電卓として使う

それでは、簡単な Python コマンドをいくつか試しましょう。インタプリタを起動して、一次プロンプト、‘>>> ’ が現れるのを待ちます。(そう長くはかからないはずです)

#### 3.1.1 数

インタプリタは単純な電卓のように動作します: 式をタイプ入力することができ、その結果が書き出されます。式の文法は素直なものです: 演算子 +, -, \*, / は (Pascal や C といった) 他のほとんどの言語と同じように動作します。括弧をグループ化に使うこともできます。例えば:

<sup>1</sup> 入力と出力を区別するために異なるフォントを使おうと思うのですが、それに必要な LaTeX の hack 作業に必要な量が、今のところ私の能力を超えています。

```

>>> 2+2
4
>>> # これはコメント
... 2+2
4
>>> 2+2 # そしてこれはコードと同じ行にあるコメント
4
>>> (50-5*6)/4
5
>>> # 整数の除算は floor (実数の解を越えない最大の整数) を返す:
... 7/3
2
>>> 7/-3
-3

```

C と同じく、等号 ('=') を使って変数に値を代入します。代入された値は書き出されません:

```

>>> width = 20
>>> height = 5*9
>>> width * height
900

```

複数の変数に同時に値を代入することができます:

```

>>> x = y = z = 0 # x と y と z をゼロにする
>>> x
0
>>> y
0
>>> z
0

```

浮動小数点は完全にサポートしています; 被演算子の型が混合されているときには、演算子は整数の被演算子を浮動小数点型に変換します。

```

>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5

```

複素数もサポートされています。虚数は接尾辞 'j' または 'J' を付けて書き表します。ゼロでない実数部をもつ複素数は '*(real+imagj)*' のように書き表すか、'*complex(real, imag)*' 関数で生成できます。

```

>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)

```



複素数は、常に実部と虚部に相当する二つの浮動小数点数で表されます。複素数  $z$  からそれぞれの部分を取り出すには、`z.real` と `z.imag` を使います。

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

数値を浮動小数点数や整数へに変換する関数 (`float()`, `int()`, `long()`) は複素数に対しては動作しません — 複素数を実数に変換する方法には、ただ一つの正解というものがないからです。絶対値 (magnitude) を (浮動小数点数として) 得るには `abs(z)` を使い、実部を得るには `z.real` を使ってください。

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a) # sqrt(a.real**2 + a.imag**2)
5.0
>>>
```

対話モードでは、最後に印字された式は変数 `_` に代入されます。このことを利用すると、Python を電卓として使うときに、計算を連続して行う作業が多少楽になります。以下に例を示します：

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
>>>
```

ユーザはこの変数を読み取り専用の値として扱うべきです。この変数に明示的な代入を行ってはいけません — そんなことをすれば、この組み込み変数と同じ名前で、元の組み込み変数の不思議な動作を覆い隠してしまうような、別個のローカルな変数が生成されてしまいます。

### 3.1.2 文字列

数のほかに、Python は文字列も操作できます。文字列はいくつもの方法で表現できます。文字列はシングルまたはダブルのクォートで囲みます。

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

文字列リテラルはいくつかの方法で複数行にまたがって記述することができます。継続行を使うことができ、これには行の末尾の文字をバックスラッシュにします。こうすることで、次の行が現在の行と論理的に継続していることを示します:

```
hello = "This is a rather long string containing\n\
several lines of text just as you would do in C.\n\
    Note that whitespace at the beginning of the line is\
significant."

print hello
```

`\n` を使って文字列に改行位置を埋め込まなくてはならないことに注意してください; 末尾のバックスラッシュの後ろにある改行文字は無視されます。従って、上の例は以下のような出力を行います:

```
This is a rather long string containing
several lines of text just as you would do in C.
    Note that whitespace at the beginning of the line is significant.
```

一方、文字列リテラルを“raw”文字列にすると、`\n` のようなエスケープシーケンスは改行に変換されません。逆に、行末のバックスラッシュやソースコード中の改行文字が文字列データに含まれます。つまり、以下の例:

```
hello = r"This is a rather long string containing\n\
several lines of text much as you would do in C."

print hello
```

は、以下のような出力を行います:

```
This is a rather long string containing\n\
several lines of text much as you would do in C.
```

また、対になった三重クォート `"""` または `'''` で文字列を囲むこともできます。三重クォートを使っているときには、行末をエスケープする必要はありません、しかし、行末の改行文字も文字列に含まれることになります。

```

print """
Usage: thingy [OPTIONS]
    -h                Display this usage message
    -H hostname       Hostname to connect to
"""

```

は以下のような出力を行います:

```

Usage: thingy [OPTIONS]
    -h                Display this usage message
    -H hostname       Hostname to connect to

```

インタプリタは、文字列演算の結果を、タイプ入力する時のと同じ方法で出力します: 文字列はクオート文字で囲い、クオート文字自体やその他の奇妙な文字は、正しい文字が表示されるようにするためにバックスラッシュでエスケープします。文字列がシングルクオートを含み、かつダブルクオートを含まない場合には、全体をダブルクオートで囲います。そうでない場合にはシングルクオートで囲みます。(後で述べる `print` を使って、クオートやエスケープのない文字列を書くことができます。)

文字列は `+` 演算子で連結させる (くっつけて一つにする) ことができ、`*` 演算子で反復させることができます。

```

>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'

```

互いに隣あった二つの文字列リテラルは自動的に連結されます: 例えば、上記の最初の行は `'word = 'Help' 'A''` と書くこともできました; この機能は二つともリテラルの場合にのみ働くもので、任意の文字列表現で使うことができるわけではありません。

```

>>> 'str' 'ing'                # <- これは ok
'string'
>>> 'str'.strip() + 'ing'      # <- これは ok
'string'
>>> 'str'.strip() 'ing'        # <- これはダメ
File "<stdin>", line 1, in ?
    'str'.strip() 'ing'
                ^
SyntaxError: invalid syntax

```

文字列は添字表記 (インデックス表記) することができます; C 言語と同じく、文字列の最初の文字の添字 (インデックス) は 0 となります。独立した文字型というものはありません; 単一の文字は、単にサイズが 1 の文字列です。Icon 言語と同じく、部分文字列をスライス表記: コロンで区切られた二つのインデックスで指定することができます。

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
```

スライスのインデックスには便利なデフォルト値があります; 最初のインデックスを省略すると、0 と見なされます。第 2 のインデックスを省略すると、スライスしようとする文字列のサイズとみなされます。

```
>>> word[2:]      # 先頭の 2 文字以外全て
'He'
>>> word[:-2]     # 末尾の 2 文字を除くすべて
'lpA'
```

C 言語の文字列と違い、Python の文字列は変更できません。インデックス指定された文字列中のある位置に代入を行おうとするとエラーになります:

```
>>> word[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> word[:1] = 'Splat'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support slice assignment
```

一方、要素どうしを組み合わせた新たな文字列の生成は、簡単で効率的です:

```
>>> 'x' + word[1:]
'xelpA'
>>> 'Splat' + word[4]
'SplatA'
```

スライス演算には便利な不変式があります: `s[:i] + s[i:]` は `s` に等しくなります。

```
>>> word[:2] + word[2:]
'HelpA'
>>> word[:3] + word[3:]
'HelpA'
```

スライス表記に行儀の悪いインデックス指定をしても、値はたしなみよく処理されます: インデックスが大きすぎる場合は文字列のサイズと置き換えられます。スライスの下境界 (文字列の左端) よりも小さいインデックス値を上境界 (文字列の右端) に指定すると、空文字列が返されます。

```
>>> word[1:100]
'elpA'
>>> word[10:]
''
>>> word[2:1]
''
```

インデックスを負の数にして、右から数えることもできます。例えば:

```
>>> word[-1]      # 最後の文字
'A'
>>> word[-2]      # 最後から二つめの文字
'p'
>>> word[-2:]      # 最後の2文字
'pA'
>>> word[:-2]      # 最後の2文字を除くすべて
'Hel'
```

-0 は 0 と全く同じなので、右から数えることができません。注意してください!

```
>>> word[-0]      # (-0 は 0 に等しい)
'H'
```

負で、かつ範囲外のインデックスをスライス表記で行うと、インデックスは切り詰められます。しかし、単一の要素を指定する (スライスでない) インデックス指定でこれを行ってはいけません:

```
>>> word[-100:]
'HelpA'
>>> word[-10]      # エラー
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

スライスの働きかたをおぼえる最も良い方法は、インデックスが文字と文字のあいだ (*between*) を指しており、最初の文字の左端が 0 になっていると考えることです。そうすると、 $n$  文字からなる文字列中の最後の文字の右端はインデックス  $n$  となります。例えば:

```
+---+---+---+---+---+
| H | e | l | p | A |
+---+---+---+---+
0   1   2   3   4   5
-5  -4  -3  -2  -1
```

といった具合です。

数が記された行のうち、最初の方の行は、文字列中のインデックス 0...5 の位置を表します; 次の行は、対応する負のインデックスを表しています。 $i$  から  $j$  までのスライスは、それぞれ  $i, j$  とラベル付けされたけられた端点間のすべての文字からなります。

非負のインデックス対の場合、スライスされた配列の長さは、スライスの両端のインデックスが境界内にあるかぎり、インデックス間の差になります。例えば、`word[1:3]` の長さは 2 になります。

組み込み関数 `len()` は文字列の長さ (length) を返します。

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

参考資料:

配列型

([../lib/typeseq.html](http://lib/typeseq.html))

次節で記述されている文字列および Unicode 文字列は配列型 の例であり、配列型でサポートされている共通の操作をサポートしています。

#### 文字列メソッド

([../lib/string-methods.html](#))

文字列や Unicode 文字列では、基本的な変換や検索を行うための数多くのメソッドをサポートしています。

#### 文字列フォーマット操作

([../lib/typesseq-strings.html](#))

文字列や Unicode 文字列が % 演算子の左被演算子である場合に呼び出されるフォーマット操作については、ここで詳しく記述されています。

### 3.1.3 Unicode 文字列

Python 2.0 から、プログラマはテキスト・データを格納するための新しいデータ型、Unicode オブジェクトを利用できるようになりました。Unicode オブジェクトを使うと、Unicode データ (<http://www.unicode.org/> 参照) を記憶したり、操作したりできます。また、Unicode オブジェクトは既存の文字列オブジェクトとよく統合していて、必要に応じた自動変換機能を提供しています。

Unicode には、古今のテキストで使われているあらゆる書き文字のあらゆる文字について、対応付けを行うための一つの序数を規定しているという利点があります。これまでは、書き文字のために利用可能な序数は 256 個しかなく、テキストは書き文字の対応付けを行っているコードページに束縛されているのが通常でした。このことは、とりわけソフトウェアの国際化 (通常 'i18n' — 'i' + 18 文字 + 'n' の意) に対して大きな混乱をもたらしました。Unicode では、すべての書き文字に対して単一のコードページを定義することで、これらの問題を解決しています。

Python では、Unicode 文字列の作成は通常の文字列を作成するのと同じように単純なものです:

```
>>> u'Hello World !'
u'Hello World !'
```

クオートの前にある小文字の 'u' は、Unicode 文字列を生成することになっていることを示します。文字列に特殊な文字を含めたければ、Python の *Unicode-Escape* エンコーディングを使って行えます。以下はその方法を示しています:

```
>>> u'Hello\u0020World !'
u'Hello World !'
```

エスケープシーケンス `\u0020` は、序数の値 `0x0020` を持つ Unicode 文字 (スペース文字) を、指定場所に挿入することを示します。

他の文字は、それぞれの序数値をそのまま Unicode の序数値に用いて解釈されます。多くの西洋諸国で使われている標準 Latin-1 エンコーディングのリテラル文字列があれば、Unicode の下位 256 文字が Latin-1 の 256 文字と同じになっていて便利だと思うことでしょう。

上級者のために、通常の文字列の場合と同じく raw モードもあります。これには、文字列を開始するクオート文字の前に 'ur' を付けて、Python に *Raw-Unicode-Escape* エンコーディングを使わせなければなりません。このモードでは、上記の `\uXXXX` の変換は機能、小文字の 'u' の前に奇数個のバックスラッシュがあるときにだけ適用されます。

```
>>> ur'Hello\u0020World !'
u'Hello World !'
>>> ur'Hello\\u0020World !'
u'Hello\\\u0020World !'
```

raw モードは、正規表現を記述する時のように、沢山のバックスラッシュを入力しなければならないときとても役に立ちます。

これら標準のエンコーディングとは別に、Python では、既知の文字エンコーディングに基づいて Unicode 文字列を生成する一連の手段を提供しています。

組み込み関数 `unicode()` は、登録されているすべての Unicode codecs (COder: エンコーダ と DECoder デコーダ) へのアクセス機能を提供します。codecs が変換できるエンコーディングには、よく知られているものとして *Latin-1*, *ASCII*, *UTF-8* および *UTF-16* があります。後者の二つは可変長のエンコードで、各 Unicode 文字を 1 バイトまたはそれ以上のバイト列に保存します。デフォルトのエンコーディングは通常 ASCII に設定されています。ASCII では 0 から 127 の範囲の文字だけを通過させ、それ以外の文字は受理せずエラーを出します。Unicode 文字列を印字したり、ファイルに書き出したり、`str()` で変換すると、デフォルトのエンコーディングを使った変換が行われます。

```
>>> u"abc"
u'abc'
>>> u"あいう"
u'\x82\xa0\x82\xa2\x82\xa4'
>>> str(u"あいう")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-5:
ordinal not in range(128)
```

特定のエンコーディングを使って Unicode 文字列を 8 ビットの文字列に変換するために、Unicode オブジェクトでは `encode()` メソッドを提供しています。このメソッドは単一の引数としてエンコーディングの名前をとります。エンコーディング名には小文字の使用が推奨されています。

```
>>> u"あいう".encode('utf-8')
'\xc2\x82\xc2\xa0\xc2\x82\xc2\xa2\xc2\x82\xc2\xa4'
```

特定のエンコーディングで書かれているデータがあり、そこから Unicode 文字列を生成したいなら、`unicode()` を使い、第 2 引数にエンコーディング名を指定します。

```
unicode('\xc2\x82\xc2\xa0\xc2\x82\xc2\xa2\xc2\x82\xc2\xa4', 'utf-8')
u'\x82\xa0\x82\xa2\x82\xa4'
```

### 3.1.4 リスト

Python は数多くの複合 (*compound*) データ型を備えており、別々の値を一まとめにするために使えます。最も汎用的なデータ型はリスト (*list*) で、コンマで区切られた値からなるリストを各カッコで囲んだものとして書き表されます。リストの要素をすべて同じ型にする必要はありません。

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

文字列のインデクスと同じく、リストのインデクスは0から開始します。また、スライス、連結なども行えます:

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boe!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boe!']
```

変化不可能 (*immutable*) な文字列型と違い、リストは個々の要素を変更することができます。

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

スライスに代入することもできます。スライスの代入を行って、リストのサイズを変更することさえできます。

```
>>> # いくつかの項目を置換する:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # いくつかの項目を除去する:
... a[0:2] = []
>>> a
[123, 1234]
>>> # いくつかの項目を挿入する:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> a[:0] = a      # それ自身 (のコピー) を先頭に挿入する
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
```

組込み関数 `len()` はリストにも適用できます。

```
>>> len(a)
8
```

リストを入れ子にする (ほかのリストを含むリストを造る) ことも可能です。例えば、



```

>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('extra')      # 5.1 節を参照
>>> p
[1, [2, 3, 'extra'], 4]
>>> q
[2, 3, 'extra']

```

最後の例では、`p[1]` と `q` が実際には同一のオブジェクトを参照していることに注意してください！ オブジェクトの意味付け (*semantics*) については、後ほど触れることにします。

## 3.2 プログラミングへの第一歩

もちろん、2 たす 2 よりももっと複雑な仕事にも Python を使うことができます。*Fibonacci* 級数列の先頭の部分列は次のようにして書くことができます：

```

>>> # Fibonacci 級数:
... # 二つの要素の和が次の要素を定義する
... a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
1
1
2
3
5
8

```

上の例では、いくつか新しい機能を取り入れています。

- 最初の行には 複数同時の代入 (*multiple assignment*) が入っています: 変数 `a` と `b` は、それぞれ同時に新しい値 0 と 1 になっています。この代入は最後の行でも再度使われており、代入が行われる前に右辺の式がまず評価されます。右辺の式は左から右へと順番に評価されます。
- `while` は、条件 (ここでは `b < 10`) が真である限り実行を繰り返し (ループし) ます。Python では、C 言語と同様に、ゼロでない整数値は真となり、ゼロは偽です。条件式は文字列値やリスト値、実際には任意の配列型でもかまいません。例中で使われている条件テストは単なる比較です。標準的な比較演算子は C 言語と同様です: すなわち、`<` (より小さい)、`>` (より大きい)、`==` (等しい)、`<=` (より小さいか等しい)、`>=` (より大きい等しい)、および `!=` (等しくない)、です。
- ループの本体 (*body*) はインデント (*indent*, 字下げ) されています: インデントは Python において実行文をグループにまとめる方法です。Python は (いまだに!) 賢い入力行編集機能を提供していないので、インデントされた各行を入力するにはタブや (複数個の) スペースを使わなければなりません。実際には、Python へのより複雑な入力を準備するにはテキストエディタを使うことになるでしょう; ほとんどのテキストエディタは自動インデント機能を持っています。複合文を対話的に入力するときに

は、(パーザはいつ最後の行を入力したのか推し量ることができないので) 入力の完了を示すために最後に空行を続けなければなりません。基本ブロックの各行は同じだけインデントされていなければならないので注意してください。

- `print` は指定した (1 つまたは複数の) 式の値を書き出します。 `print` は、(電卓の例でしたように) 単に値を出力したい式を書くのとは、複数の式や文字列を扱う方法が違います。文字列は引用符無しで出力され、複数の要素の間にはスペースが挿入されるので、以下のように出力をうまく書式化できます。

```
>>> i = 256*256
>>> print 'The value of i is', i
The value of i is 65536
```

末尾にコンマを入れると、出力を行った後に改行されません:

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

インタプリタは、最後に入力した行がまだ完全な文になっていない場合、改行をはさんで次のプロンプトを出力することに注意してください。

## その他の制御フローツール

今しがた紹介した `while` 文の他に、Python では他の言語でおなじみの普通の制御フロー文を備えていますが、これらには多少ひねりを加えています。

### 4.1 `if` 文

おそらく最もおなじみの文型は `if` 文でしょう。例えば、

```
>>> x = int(raw_input("Please enter an integer: "))
>>> if x < 0:
...     x = 0
...     print 'Negative changed to zero'
... elif x == 0:
...     print 'Zero'
... elif x == 1:
...     print 'Single'
... else:
...     print 'More'
...
...

```

`elif` 部はゼロ個またはそれ以上にできます。`else` 部を付けることもできます。キーワード '`elif`' は '`else if`' を短くしたもので、過剰なインデントを避けるのに役立ちます。一連の `if... elif... elif...` は、他の言語における `switch` 文や `case` 文の代用となります。

### 4.2 `for` 文

Python の `for` 文は、読者が C 言語や Pascal 言語で使いなれているかもしれない `for` 文とは少し違います。(Pascal のように) 常に算術型の数列にわたる反復を行ったり、(C のように) 繰返しステップと停止条件を両方ともユーザが定義できるようにするのは違い、Python の `for` 文は、任意の配列型 (リストまたは文字列) にわたって反復を行います。反復の順番は配列中に要素が現れる順番です。(for example というしゃれではないけれど) 例えば、

```
>>> # いくつかの文字列の長さを測る:
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print x, len(x)
...
cat 3
window 6
defenestrate 12

```

反復操作の対象になっている配列をループの中で書き換える操作 (リストのような、変更可能 (mutable) な配列型でおきます) は、安全ではありません。もし反復処理を行う対象とするリスト型を変更したいのなら、(対象の要素を複製するなどして) コピーに対して反復を行わなければなりません。この操作にはスライス表記を使うと特に便利です:

```
>>> for x in a[:]: # リスト全体のスライス・コピーを作る
...     if len(x) > 6: a.insert(0, x)
...
>>> a
['defenestrate', 'cat', 'window', 'defenestrate']
```

### 4.3 range() 関数

数列にわたって反復を行う必要がある場合、組み込み関数 `range()` が便利です。この関数は算術型の数列が入ったリストを生成します。

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

指定した終端値は生成されるリストには入りません。`range(10)` は 10 個の値からなるリストを生成し、ちょうど長さ 10 の配列における各項目のインデクスとなります。`range` を別の数から開始したり、他の増加量 (負の増加量でさえも; 増加量は時に ‘ステップ (step)’ と呼ばれることもあります) を指定することもできます:

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

ある配列にわたってインデクスで反復を行うには、`range()` と `len()` を次のように組み合わせます:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Mary
1 had
2 a
3 little
4 lamb
```

### 4.4 break 文と continue 文とループの else 節

`break` 文は、C 言語と同じく、最も内側の `for` または `while` ループを中断します。

`continue` 文は、これもまた C 言語から借りてきたものですが、ループを次の反復処理に飛ばします。

ループ文は `else` 節を持つことができます; `else` 節は、(`for` で) 反復処理対象のリストを使い切ってループが終了したとき、または (`while` で) 条件が偽になったときに実行されますが、`break` 文でループ

が終了したときは実行されません。この動作を、素数を探す下記のループを例にとって示します:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'equals', x, '*', n/x
...             break
...         else:
...             # 因数が見つからずにループが終了
...             print n, 'is a prime number'
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

## 4.5 pass 文

pass 文は何もしません。pass は、文を書くことが構文上要求されているが、プログラム上何の動作もする必要がない時に使われます。

```
>>> while True:
...     pass # キーボード割り込み (keyboard interrupt) を busy-wait で待つ
...
```

## 4.6 関数を定義する

フィボナッチ数列 (Fibonacci series) を任意の上限値まで書き出すような関数を作成できます:

```
>>> def fib(n):    # n までのフィボナッチ級数を出力する
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...
>>> # 今しがた定義した関数を呼び出す:
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

def は関数の定義 (*definition*) を導くキーワードです。def の後には、関数名と仮引数 (formal parameter) を丸括弧で囲んだリストを続けなければなりません。関数の実体を校正する実行文は次の行から始め、インデントされていなければなりません。関数の本体の記述する文の最初の行は文字列リテラルにすることもできます; その場合、文字列は関数のドキュメンテーション文字列 (documentation string)、または *docstring* です。

ドキュメンテーション文字列を使ったツールには、オンライン文書や印刷文書を自動的に生成したり、ユーザが対話的にコードを閲覧できるようにするものがあります; 自分が書くコードにドキュメンテーション

ン文字列を入れるのはよい習慣です。書く癖をつけるようにしてください。

関数を実行 (*execution*) すると、関数のローカル変数のために使われる新たなシンボルテーブル (*symbol table*) が取り込まれます。もっと正確にいうと、関数内で変数への代入を行うと、その値はすべてこのローカルなシンボルテーブルに記憶されます; 一方、変数の参照を行うと、まずローカルなシンボルテーブルが検索され、その後グローバルなシンボルテーブルを調べ、最後に組み込みの名前テーブルを調べます。従って、関数の中では、グローバルな変数を参照することはできますが、直接値を代入することは (*global* 文で名前を挙げておかない限り) できません。

関数を呼び出す際の実際のパラメタ (引数) は、関数が呼び出されるときに関数のローカルなシンボルテーブル内に取り込まれます; そうすることで、引数は 値渡し (*call by value*) で関数に渡されることになります (ここでの 値 (*value*) とは常にオブジェクトへの参照 (*reference*) をいい、オブジェクトの値そのものではありません)<sup>1</sup>。ある関数がほかの関数を呼び出すときには、新たな呼び出しのためにローカルなシンボルテーブルが新たに作成されます。

関数の定義を行うと、関数名は現在のシンボルテーブル内に取り入れられます。関数名の値は、インタプリタからはユーザ定義関数 (*user-defined function*) として認識される型を持ちます。この値は別の名前に代入して、その名前を後に関数として使うこともできます。これは一般的な名前変更のメカニズムとして働きます。

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
```

`fib` は関数ではなく手続き (*procedure*) だと異論があるかもしれませんがね。Python では C 言語と同様、手続きはただの関数で、値を返さないに過ぎません。技術的に言えば、実際には手続きもややつまらない値ですが値を返しています。この値は `None` と呼ばれます (これは組み込みの名前です)。`None` だけを書き出そうとすると、インタプリタは通常出力を抑制します。本当に出力したいのなら、以下のようにと見ることができます:

```
>>> print fib(0)
None
```

フィボナッチ数列の数からなるリストを出力する代わりに、値を返すような関数を書くのは簡単です:

```
>>> def fib2(n): # n までのフィボナッチ級数を返す
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while b < n:
...         result.append(b)    # 下記参照
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # 関数を呼び出す
>>> f100                # 結果を出力する
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

例によって、この例は Python の新しい機能を示しています:

---

<sup>1</sup> 実際には、オブジェクトへの参照渡し (*call by object reference*) と書けばよいのかもしれませんが。というのは、変更可能なオブジェクトが渡されると、関数の呼び出し側は、呼び出された側の関数がオブジェクトに (リストに値が挿入されるといった) 何らかの変更に出くわすことになるからです。

- `return` 文では、関数から一つ値を返します。`return` の引数となる式がない場合、`None` が返ります。手続きが終了したときにも `None` が返ります。
- 文 `result.append(b)` では、リストオブジェクト `result` のメソッド (*method*) を呼び出しています。メソッドとは、オブジェクトに ‘属している’ 関数のことで、`obj` を何らかのオブジェクト (式であっても構いません)、`methodname` をそのオブジェクトで定義されているメソッド名とすると、`obj.methodname` と書き表されます。異なる型は異なるメソッドを定義しています。異なる型のメソッドで同じ名前のメソッドを持つことができ、あいまいさを生じることはありません。(自前のオブジェクト型とメソッドを定義することもできます。これには、後でこのチュートリアルで述べるクラス (*class*) を使います。) 例で示されているメソッド `append()` は、リストオブジェクトで定義されています; このメソッドはリストの末尾に新たな要素を追加します。この例での `append()` は `‘result = result + [b]’` と等価ですが、より効率的です。

## 4.7 関数定義についてもう少し

可変個の引数を伴う関数も定義できます。引数の定義方法には 3 つの形式があり、それらを組み合わせられます。

### 4.7.1 デフォルトの引数値

もっとも便利なのは、一つ以上の引数に対してデフォルトの値を指定する形式です。この形式を使うと、定義されている引数より少ない個数の引数で呼び出せる関数を作成します:

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'): return True
        if ok in ('n', 'no', 'nop', 'nope'): return False
        retries = retries - 1
        if retries < 0: raise IOError, 'refusenik user'
    print complaint
```

この関数は、`ask_ok('Do you really want to quit?')` のようにも、`ask_ok('OK to overwrite the file?', 2)` のようにも呼び出せます。上の例では、予約語 `in` も紹介しています。`in` は、配列中に特定の値が存在するかどうかの判定結果を返します。

デフォルト値は、関数が定義された時点で、関数を定義している側のスコープ (scope) で評価されるので、

```
i = 5

def f(arg=i):
    print arg

i = 6
f()
```

は 5 を出力します。

**重要な警告:** デフォルト値は 1 度だけしか評価されません。デフォルト値がリストや辞書のような変更可能なオブジェクトの時にはその影響がでます。例えば以下の関数は、後に続く関数呼び出しで関数に渡されている引数を累積します:

```
def f(a, L=[]):
    L.append(a)
    return L

print f(1)
print f(2)
print f(3)
```

このコードは、

```
[1]
[1, 2]
[1, 2, 3]
```

を出力します。

後続の関数呼び出しでデフォルト値を共有したくなければ、代わりに以下のように関数を書くことができます:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

## 4.7.2 キーワード引数

関数を `'keyword = value'` という形式のキーワード引数を使って呼び出すこともできます。例えば、以下の関数:

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "Volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

は、以下のいずれの方法でも呼び出せます:

```
parrot(1000)
parrot(action = 'VOOOOOM', voltage = 1000000)
parrot('a thousand', state = 'pushing up the daisies')
parrot('a million', 'bereft of life', 'jump')
```

しかし、以下の呼び出しはすべて不正なものです:

```
parrot() # 必要な引数がない
parrot(voltage=5.0, 'dead') # キーワード引数の後に非キーワード引数がある
parrot(110, voltage=220) # 引数に対して値が重複している
parrot(actor='John Cleese') # 未知のキーワードを使用している
```

一般に、引数リストでは、固定引数 (positional argument) の後ろにキーワード引数を置かねばならず、キー



ワードは仮引数名から選ばなければなりません。仮引数がデフォルト値を持っているかどうかは重要ではありません。引数はいずれも一つ以上の値を受け取りません — 同じ関数呼び出しの中では、固定引数に対応づけられた仮引数名をキーワードとして使うことはできません。この制限のために実行が失敗する例を以下に示します。

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

仮引数の最後に *\*\*name* の形式のものと、仮引数に対応しないすべてのキーワード引数が入った辞書を受け取ります。*\*\*name* は *\*name* の形式をとる、仮引数のリストを超えた固定引数の入ったタプルを受け取る引数 (次の節で述べます) と組み合わせることができます。(*\*name* は *\*\*name* より前になければなりません)。例えば、ある関数の定義を以下:

```
def cheeseshop(kind, *arguments, **keywords):
    print "-- Do you have any", kind, '?'
    print "-- I'm sorry, we're all out of", kind
    for arg in arguments: print arg
    print '-'*40
    keys = keywords.keys()
    keys.sort()
    for kw in keys: print kw, ': ', keywords[kw]
```

のようになると、呼び出しは以下:

```
cheeseshop('Limburger', "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           client='John Cleese',
           shopkeeper='Michael Palin',
           sketch='Cheese Shop Sketch')
```

のようになり、もちろん以下のように出力されます:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
-----
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

キーワード引数名のリストに対して `sort()` を呼び出した後に `keywords` 辞書の内容を出力していることに注意してください; `sort()` が呼び出されていないと、引数が出力される順番は不確定となります。

### 4.7.3 任意引数リスト

最後に、最も使うことの少ない選択肢として、関数が任意の個数の引数で呼び出せるよう指定する方法があります。これらの引数はタプルにくるまれます。可変個の引数の前に、ゼロ個かそれ以上の引数があっ

でも構いません。

```
def fprintf(file, format, *args):
    file.write(format % args)
```

#### 4.7.4 引数リストのアンパック

引数がすでにリストやタプルになっていて、個別な固定引数を要求する関数呼び出しに渡すためにアンパックする必要がある場合には、逆の状況が起こります。例えば、組み込み関数 `range()` は引数 *start* と *stop* を別に与える必要があります。個別に引数を与えることができない場合、関数呼び出しを `*` 演算子を使って書き、リストやタプルから引数をアンパックします：

```
>>> range(3, 6)                # 個別の引数を使った通常の呼び出し
[3, 4, 5]
>>> args = [3, 6]
>>> range(*args)               # リストからアンパックされた引数での呼び出し
[3, 4, 5]
```

#### 4.7.5 ラムダ形式

多くの人の要望により、関数型プログラミング言語や Lisp によくあるいくつかの機能が Python に加えられました。キーワード `lambda` を使うと、名前のない小さな関数を生成できます。例えば `'lambda a, b: a+b'` は、二つの引数の和を返す関数です。ラムダ形式 (lambda form) は、関数オブジェクトが要求されている場所にならどこでも使うことができます。ラムダ形式は、構文上単一の式に制限されています。意味付け的には、ラムダ形式はただ通常に関数に構文的な糖衣をかぶせたものに過ぎません。入れ子構造になった関数定義と同様、ラムダ形式もそれを取り囲むスコープから変数を参照することができます。

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

#### 4.7.6 ドキュメンテーション文字列

ドキュメンテーション文字列については、その内容と書式に関する慣習ができてつつあります。

最初の行は、常に対象物の目的を短く簡潔にまとめたものでなくてはなりません。簡潔に書くために、対象物の名前や型を明示する必要はありません。名前や型は他の方法でも得られるからです (名前がたまたま関数の演算内容を記述する動詞である場合は例外です)。最初の行は大文字で始まり、ピリオドで終わっていないければなりません。

ドキュメンテーション文字列中にさらに記述すべき行がある場合、二行目は空行にし、まとめの行と残りの記述部分を視覚的に分離します。つづく行は一つまたはそれ以上の段落で、対象物の呼び出し規約や副作用について記述します。

Python のパーザは複数行にわたる Python 文字列リテラルからインデントを剥ぎ取らないので、ドキュメ

ントを処理するツールでは必要に応じてインデントを剥ぎ取らなければなりません。この処理は以下の規約に従って行います。最初の行の 後にある 空行でない最初の行が、ドキュメント全体のインデントの量を決めます。(最初の行は通常、文字列を開始するクオートに隣り合っているため、インデントが文字列リテラル中に現れないためです。) このインデント量と “等価な” 空白が、文字列のすべての行頭から剥ぎ取られます。インデントの量が少ない行を書いてはならないのですが、もしそういう行があると、先頭の空白すべてが剥ぎ取られます。インデントの空白の大きさが等しいかどうかは、タブ文字を (通常は 8 文字のスペースとして) 展開した後に調べられます。

以下に複数行のドキュメンテーション文字列の例を示します:

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
...
>>> print my_function.__doc__
Do nothing, but document it.

    No, really, it doesn't do anything.
```



# データ構造

この章では、すでに学んだことについてより詳しく説明するとともに、いくつか新しいことを追加します。

## 5.1 リスト型についてもう少し

リストデータ型には、他にもいくつかメソッドがあります。リストオブジェクトのすべてのメソッドを以下に示します:

**append(*x*)**

リストの末尾に要素を一つ追加します。 `a[len(a):] = [x]` と等価です。

**extend(*L*)**

指定したリスト中のすべての要素を対象のリストに追加し、リストを拡張します。 `a[len(a):] = L` と等価です。

**insert(*i*, *x*)**

指定した位置に要素を挿入します。第 1 引数は、リストのインデクスで、そのインデクスを持つ要素の直前に挿入が行われます。従って、 `a.insert(0, x)` はリストの先頭に挿入を行います。また `a.insert(len(a), x)` は `a.append(x)` と等価です。

**remove(*x*)**

リスト中で、値 *x* を持つ最初の要素を削除します。該当する項目がなければエラーとなります。

**pop([*i*])**

リスト中の指定された位置にある要素をリストから削除して、その要素を返します。インデクスが指定されなければ、 `a.pop()` はリストの末尾の要素を返します。この場合も要素は削除されます。(メソッドの用法 (signature) で *i* の両側にある角括弧は、この引数がオプションであることを表しているだけなので、角括弧を入力する必要はありません。この表記法は *Python Library Reference* の中で頻繁に見ることになるでしょう。)

**index(*x*)**

リスト中で、値 *x* を持つ最初の要素のインデクスを返します。該当する項目がなければエラーとなります。

**count(*x*)**

リストでの *x* の出現回数を返します。

**sort()**

リストの項目を、インプレース演算 (in place、元のデータを演算結果で置き換えるやりかた) でソートします。

**reverse()**

リストの要素を、インプレース演算で逆順にします。

以下にリストのメソッドをほぼ全て使った例を示します:

```

>>> a = [66.6, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.6), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.6, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.6, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.6]
>>> a.sort()
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]

```

### 5.1.1 リストをスタックとして使う

リスト型のメソッドのおかげで、簡単にリストをスタックとして使えます。スタックでは、最後に追加された要素が最初に取り出されます (“last-in, first-out”)。スタックの一番上に要素を追加するには `append()` を使います。スタックの一番上から要素を取り出すには `pop()` をインデックスを指定せずに使います。例えば以下のようにします:

```

>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]

```

### 5.1.2 リストをキューとして使う

リストをキュー (queue) として手軽に使うこともできます。キューでは、最初に追加された要素が最初に取り出されます (“first-in, first-out”)。キューの末尾に項目を追加するには `append()` を使います。キューの先頭から項目を取り出すにはインデックスに 0 を指定して `pop()` を使います。例えば以下のようにします:

```
>>> queue = ["Eric", "John", "Michael"]
>>> queue.append("Terry")           # Terry が到着 (arrive) する
>>> queue.append("Graham")         # Graham が到着する
>>> queue.pop(0)
'Eric'
>>> queue.pop(0)
'John'
>>> queue
['Michael', 'Terry', 'Graham']
```

### 5.1.3 実用的なプログラミングツール

組み込み関数には、リストで使うと非常に便利なものが三つあります: `filter()`、`map()`、`reduce()` です。

‘`filter(function, sequence)`’ は、配列 *sequence* 中の要素 *item* から、`function(item)` が真となるような要素からなる配列 (可能ならば *sequence* と同じ型の) 配列を返します。例えば、いくつかの素数を計算するには以下のようにします:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

‘`map(function, sequence)`’ は、配列 *sequence* の各要素 *item* に対して `function(item)` を呼び出し、その戻り値からなるリストを返します。例えば、三乗された値の列を計算するには以下のようにします:

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]

>>> seq = range(8)
>>> def add(x, y): return x+y
...
>>> map(add, seq, seq)
[0, 2, 4, 6, 8, 10, 12, 14]
```

‘`reduce(func, sequence)`’ は単一の値を返します。この値は 2 つの引数をとる関数 *func* を配列 *sequence* の最初の二つの要素を引数として呼び出し、次にその結果と配列の次の要素を引数にとり、以降これを繰り返すことで構成します。例えば、1 から 10 までの数の総和を計算するには以下のようにします:

```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
55
```

配列中にただ一つしか要素がなければ、その値自体が返されます; 配列が空なら、例外が送出されます。

3 つめの引数をわたして、初期値を指定することもできます。この場合、空の配列を渡すと初期値が返されます。それ以外の場合には、まず初期値と配列中の最初の要素に対して関数が適用され、次いでその結果と配列の次の要素に対して適用され、以降これが繰り返されます。例えば以下ようになります:

```

>>> def sum(seq):
...     def add(x,y): return x+y
...     return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0

```

(2.3 以降では) 実際には、上の例のように `sum()` を定義しないでください: 数値の合計は広く必要とされている操作なので、すでに組み込み関数 `sum(sequence)` が提供されており、上の例と全く同様に動作します。2.3 で追加された仕様です。

#### 5.1.4 リストの内包表記

リストの内包表記 (list comprehension) は、リストの生成を `map()` や `filter()` や `lambda` の使用に頼らずに行うための簡潔な方法を提供しています。結果として得られるリストの定義は、しばしば上記の構文を使ってリストを生成するよりも明快になります。各々のリストの内包表記は、式、続いて `for` 節、そしてその後ろに続くゼロ個かそれ以上の `for` 節または `if` 節からなります。式をタプルで評価したいなら、丸括弧で囲わなければなりません。

```

>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
>>> [x, x**2 for x in vec] # エラー - タプルには丸かっこが必要
File "<stdin>", line 1, in ?
    [x, x**2 for x in vec]
        ^
SyntaxError: invalid syntax
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]

```

リストの内包表記は `map()` よりもはるかに柔軟性があり、一つ以上の引数を持つ関数や入れ子になった関数でも利用できます:

```

>>> [str(round(355/113.0, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']

```



## 5.2 del 文

指定された値の要素をリストから削除する代わりに、インデックスで指定する方法があります: それが `del` 文です。この文はリストからスライスを除去することもできます (以前はスライスに空のリストを代入して行っていました)。例えば以下のようにします:

```
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.6, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.6, 1234.5]
```

`del` は変数全体の削除にも使えます:

```
>>> del a
```

この文の後で名前 `a` を参照すると、(別の値を `a` に代入するまで) エラーになります。 `del` の別の用途についてはまた後で取り上げます。

## 5.3 タプルと配列

リストや文字列には、インデックスやスライスを使った演算のように、数多くの共通の性質があることを見てきました。これらは配列 (*sequence*) データ型の二つの例です。Python はまだ進歩の課程にある言語なので、他の配列データ型が追加されるかもしれません。標準の配列型はもう一つあります: タプル (*tuple*) 型です。

タプルはコンマで区切られたいくつかの値からなります。例えば以下のように書きます:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # タプルを入れ子にしてもよい
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

ご覧のように、タプルは常に丸括弧で囲われています。これは、入れ子になったタプルが正しく解釈されるようにするためです; 入力の際には丸括弧なしでもかまいませんが、結局 (タプルがより大きな式の一部分の場合) たいいてい必要となります。

タプルの用途はたくさんあります。例えば、 $(x, y)$  座標対、データベースから取り出した従業員レコードなどです。タプルは文字列と同じく、変更不能です: タプルの個々の要素に代入を行うことはできません (スライスと連結を使って同じ効果を実現することはできますが)。リストのような変更可能なオブジェクトの入ったタプルを作成することもできます。

問題は 0 個または 1 個の項目からなるタプルの構築です: これらの操作を行うため、構文には特別な細工がされています。空のタプルは空の丸括弧ペアで構築できます; 一つの要素を持つタプルは、値の後ろに

コンマを続ける (単一の値を丸括弧で囲むだけでは不十分です) ことで構築できます。美しくはないけれども、効果的です。例えば以下のようにします:

```
>>> empty = ()
>>> singleton = 'hello',      # <-- 末尾のコンマに注目
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

文 `t = 12345, 54321, 'hello!'` はタプルのパック (*tuple packing*) の例です: 値 `12345`、`54321`、および `'hello!'` が一つのタプルにパックされます。逆の演算も可能です:

```
>>> x, y, z = t
```

この操作は、配列のアンパック (*sequence unpacking*) とでも呼ぶべきものです。配列のアンパックでは、左辺に列挙されている変数が、右辺の配列の長さと同じであることが要求されます。複数同時の代入が実はタプルのパックと配列のアンパックを組み合わせたものに過ぎないことに注意してください!

この操作にはわずかな非対称性があります: 複数の値をパックすると常にタプルが生成されますが、アンパックはどの配列にも働きます。

## 5.4 辞書

もう一つ、有用な型が Python に組み込まれています。それは辞書 (*dictionary*) です。辞書は他の言語にも“連想記憶 (*associated memory*)” や “連想配列 (*associative array*)” として存在することがあります。ある範囲の数でインデクス化されている配列と異なり、辞書はキー (*key*) でインデクス化されています。このキーは何らかの変更不能な型になります; 文字列、数値、およびタプルは常にキーにすることができます; ただし、タプルに何らかの変更可能なオブジェクトが含まれている場合にはキーに使うことはできません。リストをキーとして使うことはできません。これは、リストの `append()` や `extend()` メソッドを使ったリ、またスライスやインデクス指定の代入を行うと、インプレースで変更することができるためです。

辞書は順序付けのされていないキー (*key*): 値 (*value*) のペアからなり、キーが (辞書の中で) 一意でなければならない、と考えると最もよいでしょう。波括弧 (*brace*) のペア: `{}` は空の辞書を生成します。カンマで区切られた `key: value` のペアを波括弧ペアの間に入れると、辞書の初期値となる `key: value` が追加されます; この表現方法は出力時に辞書が書き出されるのと同じ方法です。

辞書での主な操作は、ある値を何らかのキーを付けて記憶することと、キーを指定して値を取り出すことです。 `del` で `key: value` のペアを削除することもできます。すでに使われているキーを使って値を記憶すると、以前そのキーに関連づけられていた値は忘れ去られてしまいます。存在しないキーを使って値を取り出そうとするとエラーになります。

辞書オブジェクトの `keys()` メソッドは、辞書で使われている全てのキーからなるリストをランダムな順番で返します (リストをソートしたいなら、このキーのリストに `sort()` を使ってください)。ある単一のキーが辞書にあるかどうか調べるには、辞書の `has_key()` メソッドを使います。

以下に、辞書を使った小さな例を示します:

```

>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.has_key('guido')
1

```

`dict()` コンストラクタは、キーと値のペアをタプルにしたものからなるリストを使って直接辞書を作成します。キーと値のペアがあるパターンをなしているなら、リストの内包表現を使えばキーと値のリストをコンパクトに指定できます。

```

>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> dict([(x, x**2) for x in vec])      # リスト内包表現を利用
{2: 4, 4: 16, 6: 36}

```

## 5.5 ループのテクニック

辞書の内容にわたってループを行う際、`iteritems()` メソッドを使うと、キーとそれに対応する値を同時に取り出せます。

```

>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...     print k, v
...
gallahad the pure
robin the brave

```

配列にわたるループを行う際、`enumerate()` 関数を使うと、要素のインデクスと要素を同時に取り出すことができます。

```

>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print i, v
...
0 tic
1 tac
2 toe

```

二つまたはそれ以上の配列型を同時にループするために、関数 `zip()` を使って各要素をひと組みにすることができます。

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print 'What is your %s? It is %s.' % (q, a)
...
What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.
```

## 5.6 条件についてもう少し

前述の `while` や `if` 文 で使った条件 (condition) には、値の比較だけでなく、他の演算子も使うことができます、

比較演算子 `in` および `not in` は、ある値がある配列中に存在するか (または存在しないか) どうかを調べます。演算子 `is` および `is not` は、二つのオブジェクトが実際に同じオブジェクトであるかどうかを調べます; この比較は、リストのような変更可能なオブジェクトにだけ意味があります。全ての比較演算子は同じ優先順位を持っており、ともに数値演算子よりも低い優先順位となります。

比較は連鎖 (chain) させることができます。例えば、`a < b == c` は、`a` が `b` より小さく、かつ `b` と `c` が等しいかどうか、をテストします。

比較演算をブール演算子 `and` や `or` で組み合わせることができ、比較演算 (あるいは何らかのブール式) の結果を `not` で否定 (negate) することもできます。これらは全て比較演算子よりもさらに低い優先順位を持ちます。ブール演算子の中では、`not` が最も高い順位を持ち、`or` が最も低くなります。これにより、`A and not B or C` と `(A and (not B)) or C` は等価になります。もちろん、丸括弧を使って望みの組み合わせを表現できます。

ブール演算子 `and` と `or` は、いわゆる短絡 (*short-circuit*) 演算子です: これらの演算子の引数は左から右へと順に評価され、結果が確定した時点で評価を止めます。例えば、`A` と `C` は真で `B` が偽のとき、`A and B and C` は式 `C` を評価しません。一般に、短絡演算子の戻り値をブール値ではなくて一般的な値として用いると、値は最後に評価された引数になります。

比較や他のブール式の結果を変数に代入することもできます。例えば、

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Python では、C 言語と違って、式の内部で代入を行えないので注意してください。C 言語のプログラムは不満を呈するかもしれませんが、この仕様は、C 言語プログラムで遭遇する、式の中で `==` のつもりで `=` とタイプしてしまうといったありふれた問題を回避します。

## 5.7 配列とその他の型の比較

配列オブジェクトは同じ配列型の他のオブジェクトと比較できます。比較には 辞書的な (*lexicographical*) 順序が用いられます: まず、最初の二つの要素を比較し、その値が等しくなければその時点で比較結果が決まります。等しければ次の二つの要素を比較し、以降配列の要素が尽きるまで続けます。比較しようとする二つの要素がいずれも同じ配列型であれば、その配列間での辞書比較を再帰的に行います。二つの配列の全ての要素の比較結果が等しくなれば、配列は等しいとみなされます。片方の配列がもう一方の先頭部

分にあたる部分配列ならば、短い方の配列が小さい(劣位の)配列とみなされます。文字列に対する辞書的な順序づけには、個々の文字ごとに ASCII 順序を用います。以下に、同じ型のオブジェクトを持つ配列間での比較を行った例を示します:

```
(1, 2, 3) < (1, 2, 4)
[1, 2, 3] < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4) < (1, 2, 4)
(1, 2) < (1, 2, -1)
(1, 2, 3) == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)
```

違う型のオブジェクト間の比較は認められていることに注意してください。比較結果は決定性がありますが、その決め方は、型は型の名前で順番づけられる、という恣意的なものです。従って、リスト (list) 型は常に文字列 (string) 型よりも小さく、文字列型は常にタプル (tuple) よりも小さい、といった具合になります。型混合の数値の比較は、数値そのものに従って比較されるので、例えば 0 は 0.0 と等しい、という結果になります。<sup>1</sup>

---

<sup>1</sup> 異なる型のオブジェクトを比較するための規則を今後にわたって当てにしていけません; Python 言語の将来のバージョンでは変更されるかもしれません。



# モジュール

Python インタプリタを終了させ、再び起動すると、これまでに行ってきた定義 (関数や変数) は失われています。ですから、より長いプログラムを書きたいなら、テキストエディタを使ってインタプリタへの入力を用意しておき、手作業の代わりにファイルを入力に使うことで動作させるとよいでしょう。この作業を スクリプト (*script*) の作成と言います; プログラムが長くなるにつれ、メンテナンスを楽にするために、スクリプトをいくつかのファイルに分割したくなるかもしれません。また、いくつかのプログラムで書いてきた便利な関数について、その定義をコピーすることなく個々のプログラムで使いたいと思うかもしれません。

こういった要求をサポートするために、Python では定義をファイルに書いておき、スクリプトの中やインタプリタの対話インスタンス上で使う方法があります。このファイルを モジュール (*module*) と呼びます; モジュールにある定義は、他のモジュールや *main* モジュール (実行のトップレベルや電卓モードでアクセスできる変数の集まりを指します) に *import* (取り込み) することができます。

モジュールは Python の定義や文が入ったファイルです。ファイル名はモジュール名に接尾語 `'.py'` がついたものになります。モジュールの中では、(文字列の) モジュール名をグローバル変数 `__name__` で取得できます。例えば、お気に入りのテキストエディタを使って、現在のディレクトリに以下の内容のファイル `'fibonacci.py'` を作成してみましょう:

```
# フィボナッチ数列モジュール

def fib(n):      # n まで加算されるフィボナッチ級数を印字
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n):     # n まで加算されるフィボナッチ級数を返す
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

次に Python インタプリタに入り、モジュールを以下のコマンドで `import` しましょう。

```
>>> import fibo
```

この操作では、`fibo` で定義された関数の名前を直接現在のシンボルテーブルに入力することはありません; 単にモジュール名 `fibo` だけをシンボルテーブルに入れます。関数にはモジュール名を使ってアクセスします:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

関数を度々使うのなら、ローカルな名前に代入できます:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

## 6.1 モジュールについてもうすこし

モジュールには、関数定義に加えて実行文を入れることができます。これらの実行文はモジュールを初期化するためのものです。これらの実行文は、モジュールがどこかで最初に import された時にだけ実行されます。<sup>1</sup>

各々のモジュールは、自前のプライベートなシンボルテーブルを持っていて、モジュールで定義されている関数はこのテーブルをグローバルなシンボルテーブルとして使います。したがって、モジュールの作者は、ユーザのグローバル変数と偶然的な衝突が起こる心配をせずに、グローバルな変数をモジュールで使うことができます。一方、自分が行っている操作をきちんと理解していれば、モジュール内の関数を参照するのと同じ表記法 `modname.itemname` で、モジュールのグローバル変数をいじることもできます。

モジュールが他のモジュールを import することもできます。import 文は全てモジュールの先頭に (さらに言えばスクリプトでも) 置きますが、これは慣習であって必須ではありません。import されたモジュール名は import を行っているモジュールのグローバルなシンボルテーブルに置かれます。

import 文には、あるモジュール内の名前を、import を実行しているモジュールのシンボルテーブル内に直接取り込むという変型があります。例えば:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

この操作は、import の対象となるモジュール名をローカルなシンボルテーブル内に取り入れることはありません (従って上の例では、`fibo` は定義されません)。

モジュールで定義されている名前を全て import するという変型もあります:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

上の操作は、アンダースコア (`_`) で開始する名前以外の全ての名前を import します。

---

<sup>1</sup>実際には、関数定義も‘実行’される‘文’です; モジュールを実行すると、関数名はモジュールのグローバルなシンボルテーブルに入力されます。



### 6.1.1 モジュール検索パス

spam という名前のモジュールが import されると、インタプリタは 'spam.py' という名前のファイルを現在のディレクトリ内で探し、次に環境変数 PYTHONPATH に指定されているディレクトリのリストから探します。PYTHONPATH はシェル変数 PATH と同じ構文、すなわちディレクトリ名を並べたものです。PYTHONPATH が設定されていないか、探しているファイルが見つからなかった場合は、検索対象をインストール方法に依存するデフォルトのパスにして続けます; UNIX では、このパスは通常 './usr/local/lib/python' です。

実際には、モジュールは変数 `sys.path` で指定されたディレクトリのリストから検索されます。`sys.path` は、入力とするスクリプトの入ったディレクトリ (現在のディレクトリ)、PYTHONPATH、およびインストール方法依存のデフォルト値を使って初期化されます。Python プログラマは、自分の行っている操作を理解しているなら、この変数を使ってモジュール検索パスを修正したり置き換えたりすることができます。起動しようとするスクリプトの入ったディレクトリが検索パス上にあるため、スクリプトが標準モジュールと同じ名前をもたないようにすることが重要です。さもなければ、Python が標準モジュールを import するときにスクリプトをモジュールとして import しようと試みてしまうので注意してください。このような誤りを犯すと、通常はエラーになります。詳しくは 6.2 節、“標準モジュール”を参照してください。

### 6.1.2 “コンパイル”された Python ファイル

たくさんの標準モジュールを使うような短いプログラムで重要となる起動時間の高速化を行うために、'spam.py' が見つかったディレクトリに 'spam.pyc' という名前のファイルがあった場合には、このファイルをモジュール spam の“バイトコンパイルされた”バージョンであると仮定します。'spam.pyc' を生成するのに使われたバージョンの 'spam.py' のファイル修正時刻が 'spam.pyc' に記録されており、この値が一致しなければ 'spam.pyc' ファイルは無視されます。

通常、'spam.pyc' ファイルを生成するために何かをする必要はありません。'spam.py' が無事コンパイルされると、常にコンパイルされたバージョンを 'spam.pyc' へ書き出すよう試みます。この試みが失敗してもエラーにはなりません; 何らかの理由でファイルが完全に書き出されなかった場合、作成された 'spam.pyc' は無効であるとみなされ、それ以後無視されます。'spam.pyc' ファイルの内容はプラットフォームに依存しないので、Python のモジュールのディレクトリは異なるアーキテクチャのマシン間で共有することができます。

エキスパートへの助言:

- Python インタプリタを `-O` フラグ付きで起動すると、最適化 (optimize) されたコードが生成されて 'pyc' ファイルに保存されます。最適化機構は今のところあまり役に立っていません; 最適化機構は `assert` 文と `SET_LINENO` 命令を除去しているだけです。 `-O` を使うと、すべてのバイトコードが最適化されます; `.pyc` ファイルは無視され、`.py` ファイルは最適化されたバイトコードにコンパイルされます。
- 二つの `-O` フラグ (`-OO`) を Python インタプリタへ渡すと、バイトコードコンパイラは、まれにプログラムが正しく動作しなくなるかもしれないような最適化を実行します。現状では、ただ `__doc__` 文字列をバイトコードから除去して、よりコンパクトな 'pyc' ファイルにするだけです。この文字列が利用できることをあてにしているプログラムがあるかもしれないので、自分の行っている操作が何かわかっているときにだけこのオプションを使うべきです。
- 'pyc' ファイルや 'pyo' ファイルから読み出されたとしても、プログラムは何ら高速に動作するわけではありません。'pyc' ファイルや 'pyo' ファイルで高速化されるのは、読み込まれるときの速度だけです。

- スクリプトの名前をコマンドラインで指定して実行した場合、そのスクリプトのバイトコードが `.pyc` や `.pyo` に書き出されることはありません。従って、スクリプトのほとんどのコードをモジュールに移し、そのモジュールを `import` する小さなブートストラップスクリプトを作れば、スクリプトの起動時間を短縮できることがあります。`.pyc` または `.pyo` ファイルの名前を直接コマンドラインに指定することもできます。
- 一つのモジュールについて、ファイル `spam.py` のない `spam.pyc` (`-O` を使ったときは `spam.pyo`) があってもかまいません。この仕様は、Python コードでできたライブラリをリバースエンジニアリングがやや困難な形式で配布するために使えます。
- `compileall` は、`.pyc` ファイル (または `-O` を使ったときは `.pyo` ファイル) をディレクトリ内の全てのモジュールに対して生成することができます。

## 6.2 標準モジュール

Python には標準モジュールのライブラリが付属しています。ライブラリは独立したドキュメント *Python ライブラリリファレンス (Python Library Reference)* (以降“ライブラリリファレンス”) で記述されています。モジュールによってはインタプリタに組み込まれたものがあります; インタプリタに組み込まれているモジュールが提供しているのは、言語の中核の部分ではありませんが、効率化のためや、システムコールのようなオペレーティングシステムの根本機能へのアクセス手段を提供するための操作です。これらのモジュールのセットは設定時に選択可能で、また根底にあるプラットフォームにも依存します。例えば、`amoeba` モジュールは、Amoeba の根本機能を何らかの形でサポートしているようなシステムでのみ提供されます。とりわけ、注目に値するモジュールが一つあります: `sys` はどの Python インタプリタにも組み込まれています。変数 `sys.ps1` と `sys.ps2` は、それぞれ一次プロンプトと二次プロンプトとして使われる文字列を定義しています:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print 'Yuck!'
Yuck!
C>
```

これらの二つの変数は、インタプリタが対話モードにあるときだけ定義されています。

変数 `sys.path` は文字列からなるリストで、インタプリタがモジュールを検索するときのパスを決定します。`sys.path` は環境変数 `PYTHONPATH` から得たデフォルトパスに、`PYTHONPATH` が設定されていなければ組み込みのデフォルト値に設定されます。標準的なリスト操作で変更することができます:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

## 6.3 `dir()` 関数

組込み関数 `dir()` は、あるモジュールがどんな名前を定義しているか調べるために使われます。`dir()` はソートされた文字列のリストを返します:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__',
 '__stdin__', '__stdout__', '__getframe__', 'api_version', 'argv',
 'builtin_module_names', 'byteorder', 'callstats', 'copyright',
 'displayhook', 'exc_clear', 'exc_info', 'exc_type', 'excepthook',
 'exec_prefix', 'executable', 'exit', 'getdefaultencoding', 'getdlopenflags',
 'getrecursionlimit', 'getrefcount', 'hexversion', 'maxint', 'maxunicode',
 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache',
 'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout',
 'version', 'version_info', 'warnoptions']
```

引数がなければ、`dir()` は現在定義している名前を列挙します。

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo, sys
>>> fib = fibo.fib
>>> dir()
['__name__', 'a', 'fib', 'fibo', 'sys']
```

変数、モジュール、関数、その他の、すべての種類の名前をリストすることに注意してください。

`dir()` は、組込みの関数や変数の名前はリストしません。これらの名前からなるリストが必要なら、標準モジュール `__builtin__` で定義されています:

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError',
 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
 'Exception', 'False', 'FloatingPointError', 'IOError', 'ImportError',
 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
 'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError', 'OverflowWarning',
 'PendingDeprecationWarning', 'ReferenceError',
 'RuntimeError', 'RuntimeWarning', 'StandardError', 'StopIteration',
 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError',
 'True', 'TypeError', 'UnboundLocalError', 'UnicodeError', 'UserWarning',
 'ValueError', 'Warning', 'ZeroDivisionError', '__debug__', '__doc__',
 '__import__', '__name__', 'abs', 'apply', 'bool', 'buffer',
 'callable', 'chr', 'classmethod', 'cmp', 'coerce', 'compile', 'complex',
 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod',
 'enumerate', 'eval', 'execfile', 'exit', 'file', 'filter', 'float',
 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id',
 'input', 'int', 'intern', 'isinstance', 'issubclass', 'iter',
 'len', 'license', 'list', 'locals', 'long', 'map', 'max', 'min',
 'object', 'oct', 'open', 'ord', 'pow', 'property', 'quit',
 'range', 'raw_input', 'reduce', 'reload', 'repr', 'round',
 'setattr', 'slice', 'staticmethod', 'str', 'string', 'sum', 'super',
 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

## 6.4 パッケージ

パッケージ (package) は、Python のモジュール名前空間を“ドット付きモジュール名 (dotted module names)”を使って構造化する手段です。例えば、モジュール名 `A.B` は、‘`A`’ というパッケージのサブモジュール ‘`B`’ を表します。ちょうど、モジュールを利用すると、別々のモジュールの著者が互いのグローバル変数名について心配しなくても済むようになるのと同じように、ドット付きモジュール名を利用すると、NumPy や Python Imaging Library のように複数モジュールからなるパッケージの著者が、互いのモジュール名について心配しなくても済むようになります。

音声ファイルや音声データを一様に扱うためのモジュールのコレクション (“パッケージ”) を設計したいと仮定しましょう。音声ファイルには多くの異なった形式がある (通常は拡張子、例えば ‘`.wav`’, ‘`.aiff`’, ‘`.au`’ など認識されます) ので、様々なファイル形式間で変換を行うためのモジュールからなる、次第に増えていくモジュールのコレクションを作成したりメンテナンスしたりする必要があるかもしれません。また、音声データに対して実行したい様々な独自の操作 (ミキシング、エコーの追加、イコライザ関数の適用、人工的なステレオ効果の作成など) があるかもしれません。そうすると、こうした操作を実行するモジュールを果てしなく書くことになるでしょう。以下に (階層的なファイルシステムで表現した) パッケージの構造案を示します:

Sound/	トップレベルのパッケージ
__init__.py	サウンドパッケージを初期化する
Formats/	ファイルフォーマット変換用の下位パッケージ
__init__.py	
wavread.py	
wavwrite.py	
aiffread.py	
aiffwrite.py	
auread.py	
auwrite.py	
...	
Effects/	サウンド効果用の下位パッケージ
__init__.py	
echo.py	
surround.py	
reverse.py	
...	
Filters/	フィルタ用の下位パッケージ
__init__.py	
equalizer.py	
vocoder.py	
karaoke.py	
...	

パッケージを `import` する際、Python は `sys.path` 上のディレクトリを検索して、トップレベルのパッケージのに入ったサブディレクトリを探します。

あるディレクトリを、パッケージが入ったディレクトリとして Python に扱わせるには、ファイル ‘`__init__.py`’ が必要です: このファイルを置かなければならないのは、‘`string`’ のようなよくある名前のディレクトリにより、モジュール検索パスの後の方で見つかる正しいモジュールが意図せず隠蔽されてしまうのを防ぐためです。最も簡単なケースでは ‘`__init__.py`’ はただの空ファイルで構いませんが、‘`__init__.py`’ ではパッケージのための初期化コードを実行したり、後述の `__all__` 変数を設定してもかまいません。

パッケージのユーザは、個々のモジュールをパッケージから `import` することができます。例えば:

```
import Sound.Effects.echo
```

この操作はサブモジュール `Sound.Effects.echo` をロードします。このモジュールは、以下のように完全な名前でも参照しなければなりません:

```
Sound.Effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

サブモジュールを `import` するもう一つの方法を示します:

```
from Sound.Effects import echo
```

これもサブモジュール `echo` をロードし、`echo` をパッケージ名を表す接頭辞なしで利用できるようにします。従って以下のように用いることができます:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

さらにもう一つのバリエーションとして、必要な関数や変数を直接 `import` する方法があります:

```
from Sound.Effects.echo import echofilter
```

この操作も同様にサブモジュール `echo` をロードしますが、`echofilter` を直接利用できるようにします。

```
echofilter(input, output, delay=0.7, atten=4)
```

`from package import item` を使う場合、`item` はパッケージ `package` のサブモジュール (またはサブパッケージ) でもかまいませんし、関数やクラス、変数のような、`package` で定義されている別の名前でもかまわないことに注意してください。`import` 文はまず、`item` がパッケージ内で定義されているかどうか調べます; 定義されていないければ、`item` はモジュール名であると仮定して、モジュールをロードしようと試みます。もしモジュールが見つからなければ、`ImportError` が送出されます。

反対に、`import item.subitem.subsubitem` のような構文を使った場合、最後の `subsubitem` を除く各要素はパッケージでなければなりません; 最後の要素はモジュールかパッケージにできますが、一つ前の要素で定義されているクラスや関数や変数にはできません。

### 6.4.1 パッケージから `*` を `import` する

それでは、ユーザが `from Sound.Effects import *` と書いたら、どうなるのでしょうか? 理想的には、何らかの方法でファイルシステムが調べられ、そのパッケージにどんなサブモジュールがあるかを調べ上げ、全てを `import` する、という処理を望むことでしょう。残念ながら、この操作は Mac や Windows のプラットフォームではうまく動作しません。これらのプラットフォームでは、ファイルシステムはファイル名の大小文字の区別について正しい情報をもっているとは限らないからです! こうしたプラットフォームでは、ファイル `'ECHO.PY'` をモジュール `echo` として `import` すべきか、`Echo` とすべきかが分かる確かな方法がないからです (例えば、Windows 95 はすべてのファイル名の最初の文字を大文字にして表示するという困った慣習があります)。また、DOS の 8+3 のファイル名制限のせいで、長いモジュール名に関して別の奇妙な問題が追加されています。

唯一の解決策は、パッケージの作者にパッケージの索引を明示的に提供させるというものです。`import` 文は次の規約を使います: パッケージの `'__init__.py'` コードに `__all__` という名前のリストが定義されていれば、`from package import *` が現れたときに `import` するリストとして使います。新たなパッケー



ジがリリースされるときにリストを最新の状態に更新するのはパッケージの作者の責任となります。自分のパッケージから `*` を `import` するという使い方に同意できなければ、パッケージの作者は `'__init__.py'` をサポートしないことにしてもかまいません。例えば、ファイル `Sounds/Effects/__init__.py` には、次のようなコードを入れてもよいかもしれません:

```
__all__ = ["echo", "surround", "reverse"]
```

このコードは、`from Sound.Effects import *` とすると、`Sound` パッケージから指定された 3 つのサブモジュールが `import` されることになっている、ということを意味します。

もしも `__all__` が定義されていなければ、実行文 `from Sound.Effects import *` は、パッケージ `Sound.Effects` の全てのサブモジュールを現在の名前空間の中へ `import` しません; この文は単に (場合によっては初期化コード `'__init__.py'` を実行して) パッケージ `Sound.Effects` が `import` されたということを確認し、そのパッケージで定義されている名前を全て `import` するだけです。 `import` される名前には、`'__init__.py'` で定義された名前 (と、明示的にロードされたサブモジュール) が含まれます。パッケージのサブモジュールで、以前の `import` 文で明示的にロードされたものも含まれます。以下のコードを考えてください:

```
import Sound.Effects.echo
import Sound.Effects.surround
from Sound.Effects import *
```

上の例では、`echo` と `surround` モジュールが現在の名前空間に `import` されます。これらのモジュールは `from...import` 文が実行された際に `Sound.Effects` 内で定義されているからです (この機構は `__all__` が定義されているときにも働きます)。

一般的には、モジュールやパッケージから `*` を `import` するというやり方には賛同できません。というのは、この操作を行うとしばしば可読性に乏しいコードになるからです。しかし、対話セッションでキータイプの量を減らすために使うのは構わないでしょう。それに、特定のモジュールでは、特定のパターンに従った名前のみを公開 (`export`) するように設計されています。

`from package import specific_submodule` を使っても何も問題はないことに留意してください! 実際この表記法は、`import` を行うモジュールが他のパッケージかと同じ名前を持つサブモジュールを使わなければならない場合を除いて推奨される方式です。

## 6.4.2 パッケージ内での参照

サブモジュール同士で互いに参照を行う必要がしばしば起こります。例えば、`surround` モジュールは `echo` モジュールを使うかもしれません。実際には、このような参照はよくあることなので、`import` 文を実行すると、まず最初に `import` 文の入っているパッケージを検索し、その後になって標準のモジュール検索パスを見に行きます。こうして、`surround` モジュールは単に `import echo` や `from echo import echofilter` を使うことができます。 `import` されたモジュールが現在のパッケージ (現在のモジュールをサブモジュールにしているパッケージ) 内に見つからなかった場合、`import` 文は指定した名前のトップレベルのモジュールを検索します。

パッケージが (前述の例の `Sound` パッケージのように) サブパッケージの集まりに構造化されている場合、兄弟関係にあるパッケージを短縮された記法で参照する方法は存在しません - サブパッケージの完全な名前を使わなければなりません。例えば、モジュール `Sound.Filters.vocoder` で `Sound.Effects` パッケージの `echo` モジュールを使いたいとすると、`from Sound.Effects import echo` を使うことはできます。

### 6.4.3 複数ディレクトリ中のパッケージ

パッケージのサポートする特殊な属性には、もう一つ `__path__` があります。この属性は、パッケージの `'__init__.py'` 中のコードが実行されるよりも前に、`'__init__.py'` の収められているディレクトリ名の入ったリストになるよう初期化されます。この変数は変更することができます; 変更を加えると、以降そのパッケージに入っているモジュールやサブパッケージの検索に影響します。

この機能はほとんど必要にはならないのですが、パッケージ内に見つかるモジュールのセットを拡張するために使うことができます。





# 入力と出力

プログラムの出力をもたらす方法はいくつかあります; データは人間が可読な形で出力することも、将来使うためにファイルに書くこともできます。この章では、こうした出力のいくつかの可能性について議論します。

## 7.1 ファンシーな出力の書式化

これまでのところ、値を出力する二つの方法: 式でできた文 (*expression statement*) と `print` 文が出てきました。(第三はファイルオブジェクトの `write()` を使う方法です; 標準出力を表すファイルは `sys.stdout` で参照できます。詳細はライブラリリファレンスを参照してください。)

出力を書式化する際に、単に値をスペースで区切って出力するよりももっときめ細かな制御をしたいと思うことがしばしばあるでしょう。出力を書式化するには二つの方法があります; 第一の方法は、全ての文字列を自分で処理するというものです; 文字列のスライスや結合といった操作を使えば、思い通りのレイアウトを作成することができます。標準モジュール `string` には、文字列を指定されたカラム幅にそろえるための便利な操作がいくつかあります; これらの操作については、後で簡単に説明します。第二の方法は `%` 演算子を使い、文字列を演算子の左引数 (left argument) として使う方法です。`%` 演算子は、左引数を `sprintf()` のような形式で解釈して右引数に適用し、その書式化操作で得られた文字列を返します。

もちろん、一つ問題があります。値をどうやって文字列に変換したらいいのでしょうか? 幸運なことに、Python には値を文字列に変換する方法があります: 値を `repr()` か `str()` 関数に渡してください。逆クオート (`"`) は `repr()` と等しい操作ですが、利用はお勧めしません。

`str()` 関数は、値を表現するときにかなり人間にとって可読なものにするためのものです。一方、`repr()` はインタプリタで読めるような表現にする (あるいは、等価な値を表現するための構文がない場合には `SyntaxError` を送出させる) ためのものです。人間が利用するための特別な表現をもたないオブジェクトでは、`str()` は `repr()` と同じ値を返します。数値や、リストや辞書といった構造体のような多くの値は、どちらの関数でも同じ表現になります。文字列と浮動小数点は特別で、二つの別個の表現となります。

下にいくつか例を挙げます:

```

>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(0.1)
'0.1'
>>> repr(0.1)
'0.10000000000000001'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print s
The value of x is 32.5, and y is 40000...
>>> # 文字列への repr() はクォートとバックスラッシュが付加される:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print hellos
'hello, world\n'
>>> # repr() の引数は Python オブジェクトの場合もある:
... repr((x, y, ('spam', 'eggs'))))
"(32.5, 40000, ('spam', 'eggs'))"
>>> # 逆クォートは対話セッションで便利である:
... 'x, y, ('spam', 'eggs')'
"(32.5, 40000, ('spam', 'eggs'))"

```

以下に 2 乗と 3 乗の値からなる表を書く二つの方法を示します:

```

>>> for x in range(1, 11):
...     print repr(x).rjust(2), repr(x*x).rjust(3),
...     # 上の行の末尾のコンマに注意
...     print repr(x*x*x).rjust(4)
...
1   1   1
2   4   8
3   9  27
4  16  64
5  25 125
6  36 216
7  49 343
8  64 512
9  81 729
10 100 1000
>>> for x in range(1,11):
...     print '%2d %3d %4d' % (x, x*x, x*x*x)
...
1   1   1
2   4   8
3   9  27
4  16  64
5  25 125
6  36 216
7  49 343
8  64 512
9  81 729
10 100 1000

```

(各カラムの間のスペース一個は print の働きで追加されていることに注意してください: print は引数間に常に空白を追加します)

この例では、メソッド `rjust()` を実際に利用しています。`rjust()` は文字列を指定された幅のフィールド内に右詰めに入るように、左に空白を追加します。同様のメソッドとして、`ljust()` と `center()` があります。これらのメソッドは何か出力を行うわけではなく、ただ新しい文字列を返します。入力文字列が長すぎる場合、文字列を切り詰めることはせず、ただ値をそのまま返します; この仕様のために、カラムのレイアウトが滅茶苦茶になるかもしれませんが、嘘の値が代わりに書き出されるよりはましです。(本当に切り詰めを行いたいのなら、全てのカラムに '`ljust(x, n)[0:n]`' のようにスライス表記を加えることもできます。)

もう一つのメソッド、`zfill()` は、数値文字列の左側をゼロ詰めします。このメソッドは正と負の符号を正しく扱います:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

% 演算子を使う場合は以下ようになります:

```
>>> import math
>>> print 'The value of PI is approximately %5.3f.' % math.pi
The value of PI is approximately 3.142.
```

文字列の中に複数の書式がある場合には、以下の例のように、右側の被演算子にタプルを渡す必要があります:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print '%-10s ==> %10d' % (name, phone)
...
Jack          ==>      4098
Dcab          ==>      7678
Sjoerd        ==>      4127
```

ほとんどの書式化は C 言語と同じように動作し、正しい型を渡す必要があります; しかし、正しい型を渡さなかった場合にはコアダンプではなく例外の送出になります。書式 `%s` はもっと寛大です: 対応する引数が文字列オブジェクトでなければ、組み込み関数 `str()` を使って文字列に変換してくれます。また、数値表現の桁幅や精度を別個の (整数の) 引数として渡せるよう、`*` がサポートされています。C 言語の書式 `%n` と `%p` はサポートされていません。

もしも長い書式化文字列があり、それを分割したくない場合には、変数を引数の位置ではなく、変数の名前前で参照できるとよいでしょう。以下の形式 `%(name)format` を使えば可能になります:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print 'Jack: %(Jack)d; Sjoerd: %(Sjoerd)d; Dcab: %(Dcab)d' % table
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

全てのローカルな変数が入った辞書を返す、新たに紹介する組み込み関数 `vars()` と組み合わせると特に便利です。

## 7.2 ファイルを読み書きする

`open()` はファイルオブジェクトを返します。 `open()` は、`'open(filename, mode)'` のように二つの引数を伴って呼び出されることがほとんどです。

```
>>> f = open('/tmp/workfile', 'w')
>>> print f
<open file '/tmp/workfile', mode 'w' at 80a0960>
```

最初の引数はファイル名の入った文字列です。二つめの引数もまた文字列で、ファイルをどのように使うかを示す数個の文字が入っています。 *mode* は、ファイルが読み出し専用なら `'r'`、書き込み専用 (同名の既存のファイルがあれば消去されます) なら `'w'` とします。 `'a'` はファイルを追記用に開きます; ファイルに書き込まれた内容は自動的にファイルの終端に追加されます。 `'r+'` はファイルを読み書き両用に開きます。 *mode* 引数はオプションです; 省略された場合には `'r'` であると仮定します。

Windows や Macintosh では、*mode* に `'b'` を追加するとファイルをバイナリモードで開きます。したがって、`'rb'`、`'wb'`、`'r+b'` といったモードがあります。Windows はテキストファイルとバイナリファイルを区別しています; テキストファイルでは、読み書きの際に行末文字が自動的に少し変更されます。この舞台裏でのファイルデータ変更は、ASCII でできたテキストファイルでは差し支えないものですが、JPEG や `'EXE'` ファイルのようなバイナリデータは破損してしまうことになるでしょう。こうしたファイルを読み書きする際にはバイナリモードを使うよう十分注意してください。(Macintosh では、テキストモードに対する厳密な意味付けは、根底にある使用中の C 言語ライブラリに依存するので注意してください。)

### 7.2.1 ファイルオブジェクトのメソッド

この節の以降の例は、`f` というファイルオブジェクトが既に生成されているものと仮定します。

ファイルの内容を読み出すには、`f.read(size)` を呼び出します。このメソッドはある量のデータを読み出して、文字列として返します。 *size* はオプションの数値引数です。 *size* が省略されたり負の数であった場合、ファイルの内容全てを読み出して返します; ただし、ファイルがマシンのメモリの二倍の大きさもある場合にはどうなるかわかりません。 *size* が負でない数ならば、最大で *size* バイトを読み出して返します。ファイルの終端にすでに達していた場合、`f.read()` は空の文字列 (`""`) を返します。

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

`f.readline()` はファイルから 1 行だけを読み取ります; 改行文字 (`\n`) は読み出された文字列の終端に残ります。改行が省略されるのは、ファイルが改行で終わっていない場合の最終行のみです。これは、戻り値があいまいでないようにするためです; `f.readline()` が空の文字列を返したら、ファイルの終端に達したことが分かります。一方、空行は `'\n'`、すなわち改行 1 文字だけからなる文字列で表現されます。

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

`f.readlines()` は、ファイルに入っているデータの全ての行からなるリストを返します。オプション

のパラメタ *sizehint* が指定されていれば、ファイルから指定されたバイト数を読み出し、さらに一行を完成させるのに必要なだけを読み出して、読み出された行からなるリストを返します。このメソッドは巨大なファイルを行単位で効率的に読み出すためによく使われます。未完成の行が返されることはありません。

```
>>> f.readlines()
['This is the first line of the file.\n', 'Second line of the file\n']
```

`f.write(string)` は、*string* の内容をファイルに書き込み、`None` を返します。

```
>>> f.write('This is a test\n')
```

文字列以外をファイルに書き込みたければ、まず文字列に変換しておかねばなりません：

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
```

`f.tell()` は、ファイルオブジェクトが指しているあるファイル中の位置を示す整数を、ファイルの先頭からのバイト数で図った値で返します。ファイルオブジェクトの位置を変更するには、`'f.seek(offset, from_what)'` を使います。ファイル位置は基準点 (reference point) にオフセット値 *offset* を足して計算されます；参照点は *from\_what* 引数で選びます。*from\_what* の値が 0 ならばファイルの先頭から測り、1 ならば現在のファイル位置を使い、2 ならばファイルの終端を参照点として使います。*from\_what* は省略することができ、デフォルトの値は 0、すなわち参照点としてファイルの先頭を使います。

```
>>> f = open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)          # ファイルの第 6 バイトへ行く
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # 終端から前へ第 3 バイトへ行く
>>> f.read(1)
'd'
```

ファイルが用済みになったら、`f.close()` を呼び出してファイルを閉じ、ファイルを開くために取られていたシステム資源を解放します。`f.close()` を呼び出した後、そのファイルオブジェクトを使おうとすると自動的に失敗します。

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

ファイルオブジェクトには、他にも `isatty()` や `truncate()` といった、あまり使われないメソッドがあります；ファイルオブジェクトについての完全なガイドは、ライブラリリファレンスを参照してください。

## 7.2.2 pickle モジュール

文字列をファイルに読み書きするのは簡単にできます。数値でもほんのわずかに苦労するくらいです。というのは、`read()` は文字列だけを返すので、`'123'` のような文字列を受け取って、その数値 `123` を返す `int()` のような関数に対して文字列を渡してやらなければならないからです。ところが、リストや辞書、クラスのインスタンスのように、もっと複雑なデータ型を保存したいなら、事態はもっと複雑になります。

複雑なデータ型を保存するためのコードを利用者に毎回毎回書かせてデバッグさせる代わりに、Python では `pickle` という標準モジュールを用意しています。`pickle` は驚くべきモジュールで、ほとんどどんな Python オブジェクトも (ある形式の Python コードでさえも!) 受け取って文字列表現へ変換できます。この変換過程は *pickling* (ピクルス (漬物) 化、以降 `pickle` 化) と呼ばれます。文字列表現からオブジェクトを再構成する操作は *unpickling* (逆 `pickle` 化) と呼びます。`pickle` 化や逆 `unpickle` 化の間、オブジェクトを表現する文字列はファイルやデータに保存したり、ネットワーク接続を介して離れたマシンに送信したりできます。

オブジェクト `x` と、書込み用に開かれているファイルオブジェクト `f` があると仮定すると、オブジェクトを `pickle` 化する最も簡単な方法は、たった一行のコードしか必要ありません:

```
pickle.dump(x, f)
```

逆 `pickle` 化して再びオブジェクトに戻すには、`f` を読取り用に開かれているファイル・オブジェクトと仮定して:

```
x = pickle.load(f)
```

とします。

(逆 `pickle` 化にはいくつか変型があり、たくさんのオブジェクトを `pickle` 化したり、`pickle` 化されたデータをファイルに書きたくないときに使われます。完全なドキュメントについては、ライブラリリファレンスの `pickle` を調べてください。)

`pickle` は、Python のオブジェクトを保存できるようにし、他のプログラムや、同じプログラムが将来起動されたときに再利用できるようにする標準の方法です; 技術的な用語でいうと *persistent* (永続性) オブジェクトです。`pickle` はとても広範に使われているので、Python 拡張モジュールの多くの作者は、行列のような新たなデータ型が正しく `pickle` 化/`unpickle` 化できるよう気をつけています。

## エラーと例外

これまでエラーメッセージについては簡単に触れるだけでしたが、チュートリアル中の例を自分で試していたら、実際にいくつかのエラーメッセージを見ていることでしょう。エラーには (少なくとも) 二つのはっきり異なる種類があります: それは 構文エラー (*syntax error*) と例外 (*exception*) です。

### 8.1 構文エラー

構文エラーは構文解析エラー (*parsing error*) としても知られており、まだ Python を学習中なら、おそらくもっともよく受け取る種の文句でしょう:

```
>>> while True print 'Hello world'
      File "<stdin>", line 1, in ?
        while True print 'Hello world'
                        ^
SyntaxError: invalid syntax
```

パーサは違反の起きている行を繰り返し、小さな ‘矢印’ を表示して、違反の起きている行中でエラーが検出された最初の位置を示します。エラーは矢印の直前のトークンで引き起こされています (または、少なくともそこで検出されています)。上述の例の中では、エラーは `print` で検出されています。コロンの (‘:’) がその前に無いからです。入力がスクリプトから来ている場合は、どこを見ればよいか分かるようにファイル名と行番号が出力されます。

### 8.2 例外

たとえ文や式が構文的に正しくても、実行しようとしたときにエラーが発生するかもしれません。実行中に検出されたエラーは 例外 (*exception*) と呼ばれ、常に致命的とは限りません: Python プログラムで例外をどのように扱うかは、すぐに習得することでしょう。ほとんどの例外はプログラムで処理されず、以下に示されるようなメッセージになります:



```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

エラーメッセージの最終行は何が起こったかを示しています。例外は様々な例外型 (type) で起こり、その型がエラーメッセージの一部として出力されます: 上の例での型は `ZeroDivisionError`, `NameError`, `TypeError` です。例外型として出力される文字列は、発生した組み込み例外の名前です。これは全ての組み込み例外について成り立ちますが、ユーザ定義の例外では (成り立つようにするのは有意義な慣習ですが) 必ずしも成り立ちません。標準例外の名前は組み込みの識別子です (予約語ではありません)。

残りの行は例外の詳細で、その解釈は例外の型に依存します; これらの行の意味するところは例外の型に依存します。

エラーメッセージの先頭部分では、例外が発生した実行コンテキスト (context) を、スタックのバックトレース (stack backtrace) の形式で示しています。一般には、この部分にはソースコード行をリストしたバックトレースが表示されます; しかし、標準入力から読み取られた行については表示しません。

*Python* ライブラリリファレンスには、組み込み例外とその意味がリストされています。

## 8.3 例外を処理する

例外を選別して処理するようなプログラムを書くことができます。以下の例を見てください。この例では、有効な文字列が入力されるまでユーザに入力を促しますが、ユーザがプログラムに (Control-C か、またはオペレーティングシステムがサポートしている何らかのキーを使って) 割り込みをかけてプログラムを中断させることができるようにしています; ユーザが生成した割り込みは、`KeyboardInterrupt` 例外が送出されることで通知されるということに注意してください。

```
>>> while True:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops! That was no valid number. Try again..."
... 
```

`try` 文は下記のように動作します。

- まず、`try` 節 (*try clause*) (キーワード `try` と `except` のあいだの文) が実行されます。
- 何も例外が発生しなければ、`except` 節 をスキップして `try` 文の実行を終えます。
- `try` 節内の実行中に例外が発生すると、その節の残りは飛ばされます。次に、例外型が `except` キーワードの後に指定されている例外に一致する場合、`try` 節の残りはスキップして、`except` 節が実行された後、`try` 節の後の文に実行が継続されます。



- もしも `except` 節で指定された例外と一致しない例外が発生すると、その例外は `try` 文の外側に渡されます。例外に対するハンドラ (handler、処理部) がどこにもなければ、処理されない例外 (*unhandled exception*) となり、上記に示したようなメッセージを出して実行を停止します。

一つの `try` 文に複数の `except` 節を設けて、さまざまな例外に対するハンドラを指定することができます。同時に一つ以上のハンドラが実行されることはありません。ハンドラは対応する `try` 節内で発生した例外だけを処理し、同じ `try` 節内の別の例外ハンドラで起きた例外は処理しません。`except` 節には複数の例外を丸括弧で囲ったリストにして渡すことができます。例えば以下のようにします:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

最後の `except` 節では例外名を省いて、ワイルドカード (wildcard、総称記号) にすることができます。ワイルドカードの `except` 節は非常に注意して使ってください。というのは、ワイルドカードは通常のプログラムエラーをたやすく隠してしまうからです! ワイルドカードの `except` 節はエラーメッセージを出力した後に例外を再送出する (関数やメソッドの呼び出し側が同様にして例外を処理できるようにする) 用途にも使えます:

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError, (errno, strerror):
    print "I/O error(%s): %s" % (errno, strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

`try...except` 文には、オプションで `else` 節 (*else clause*) を設けることができます。`else` 節を設ける場合、全ての `except` 節よりも後ろに置かねばなりません。`except` 節は、`try` 節で全く例外が送出されなかったときに実行されるコードを書くのに役立ちます。例えば以下のようにします:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
```

追加のコードを追加するのは `try` 節の後ろよりも `else` 節の方がよいでしょう。なぜなら、そうすることで `try...except` 文で保護したいコードから送出されたものの以外の例外を偶然に捕捉してしまうという事態を避けられるからです。

例外が発生するとき、例外に関連付けられた値を持つことができます。この値は例外の例外の引数 (*argument*) としても知られています。引数の有無と引数の型がどうなっているかは例外の型に依存します。

`except` 節では、例外名 (または例外名リスト) の後に変数を指定することができます。この変数は例外インスタンスに結び付けられており、`instance.args` に例外インスタンス生成時の引数が入っています。

例外インスタンスには `__getitem__` および `__str__` が定義されており、`.args` を参照しなくても引数に直接アクセスしたり印字したりできるように利便性が図られています。

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception, inst:
...     print type(inst)          # 例外インスタンス
...     print inst.args          # .args に記憶されている引数
...     print inst               # __str__ で引数を直接出力できる
...     x, y = inst              # __getitem__ で引数を直接アンパックできる
...     print 'x =', x
...     print 'y =', y
...
<type 'instance'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

処理されない例外の場合、例外が引数を持っていれば、メッセージの最後の（‘詳細説明の’）部分に出力されます。

例外ハンドラは、try 節でじかに発生した例外を処理するだけでなく、その try 節から呼び出された関数の内部で発生した例外も処理します（間接的に呼ばれていてもです）。例えば：

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError, detail:
...     print 'Handling run-time error:', detail
...
Handling run-time error: integer division or modulo
```

## 8.4 例外を送出する

`raise` 文を使うと、プログラマは指定した例外を強制的に送出させられます。例えば：

```
>>> raise NameError, 'HiThere'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HiThere
```

`raise` の第一引数には、ひき起こすべき例外を指定します。オプションの第二引数では例外の引数を指定します。

例外が発生したかどうかは判定したいが、その処理を行おうとは思っていない場合、単純な形式の `raise` 文を使って例外を再送出させることができます：

```
>>> try:
...     raise NameError, 'HiThere'
... except NameError:
...     print 'An exception flew by!'
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```

## 8.5 ユーザ定義の例外

プログラム上で新しい例外クラスを作成することで、独自の例外を指定することができます。例外は、典型的に `Exception` クラスから、直接または間接的に導出したものです。例えば:

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError, e:
...     print 'My exception occurred, value:', e.value
...
My exception occurred, value: 4
>>> raise MyError, 'oops!'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

例外クラスでは、他のクラスができることなら何でも定義することができますが、通常は単純なものにしておきます。たいていは、いくつかの属性だけを提供し、例外が発生したときにハンドラがエラーに関する情報を取り出せるようにする程度にとどめます。複数の別個の例外を送出するようなモジュールを作成する際には、そのモジュールで定義されている例外の基底クラスを作成するのが一般的なならわしです:

```

class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not
    allowed.

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition is not allowed
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message

```

ほとんどの例外は、標準の例外の名前付けと同様に、“Error,”で終わる名前で定義されています。

多くの標準モジュールでは、モジュールで定義されている関数内で発生する可能性のあるエラーを報告させるために、独自の例外を定義しています。クラスについての詳細な情報は [9 章](#)、“クラス”で提供されています。

## 8.6 後片付け動作を定義する

`try` 文にはもう一つオプションの節があります。この節はクリーンアップ動作を定義するためのもので、どんな状況でも必ず実行されます。例えば:

```

>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Goodbye, world!'
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
KeyboardInterrupt

```

*finally* 節 (*finally clause*) は、`try` 節で例外が発生したかどうかに関係なく実行されます。例外が発生した時は、`finally` 節を実行した後、その例外を再送出します。`finally` 節はまた、`try` 文から `break` 文や `return` 文経由で抜ける際にも、“抜ける途中で”実行されます。

`finally` 節中のコードは (ファイルやネットワーク接続のような) 外部の資源について、資源の利用が成功

したかどうかに関係なく解放を行うのに便利です。

`try` 文には、一個以上の `except` 節か、または一個の `finally` 節を持たねばなりませんが、両方持つことはできません。



# クラス

Python では、最小限の構文と意味付けを使ってクラス (class) のメカニズムを言語に追加しています。Python のクラスは、C++ と Modula-3 で見られるクラスメカニズムを混合したものです。モジュールがそうであるように、Python におけるクラスでは、クラス定義とユーザとの間に絶対的な障壁をおかず、ユーザが礼儀正しく、“定義に首を突っ込む” ことはないとあてにしています。とはいえ、クラスにおける最も重要な機能はそのままに、完全な力を持っています: クラスの継承 (inheritance) メカニズムでは、複数の基底クラスを持つことができ、導出されたクラスでは基底クラスの任意のメソッドをオーバーライド (override, 上書き) することができます。メソッドでは、基底クラスのメソッドを同じ名前で呼び出すことができます。オブジェクトには任意のプライベートなデータを入れることができます。

C++ の用語では、全てのクラスメンバ (データメンバも含む) は *public* (公開されたデータ) であり、メンバ関数はすべて仮想関数 (*virtual*) です。特別なコンストラクタ (constructor、生成関数) やデストラクタ (destructor、破壊関数) はありません。Modula-3 にあるような、オブジェクトのメンバをメソッドから参照するために短縮した記法を使うことはできません: メソッド関数の宣言では、オブジェクト自体をあらわす明示的な第一の引数を伴います。オブジェクトはメソッド呼び出しの際に非明示的に渡されます。Smalltalk にあるように、クラス自体もオブジェクトです。ただし広義のオブジェクトですが: Python では全てのデータ型はオブジェクトです。このことが、import や名前変更といった操作の意味付けを提供しています。しかし、C++ や Modula-3 と違って、ユーザが組込みの型を基底クラスにして拡張することはできません。また、C++ と同じで、Modula-3 とは違い、特別な構文を伴うほとんどの組み込み演算子 (算術演算子 (arithmetic operator) や添字表記) はクラスインスタンスで使うために再定義することができます。

## 9.1 用語について一言

クラスに関して広範に受け入れられている用語定義がないので、Smalltalk と C++ の用語を場合に応じて使っていくことにします。(オブジェクト指向における意味付けの方法は C++ よりも Modula-3 のほうが Python に近いので Modula-3 の用語を使いたいのですが、ほとんどの読者はそれを耳にしたことがないと思います。)

また、オブジェクト指向派の読者にとって用語上の落とし穴があることを、警告しておかねばなりません: Python では、“オブジェクト” という言葉は必ずしもクラスのインスタンスを意味しません。C++ や Modula-3 と同じく、そして Smalltalk とは違い、Python ではすべての型がクラスで定義されているとは限りません。整数やリストのような基本的な組込み型はクラスではなく、ファイルのような幾分珍しい型ですらクラスではありません。とはいえ、Python の型はすべて、オブジェクトという言葉を使って記述するのが最適な、ちょっとした共通の意味付けを共有しています。

オブジェクトには個性があり、同一のオブジェクトに (複数のスコープの) 複数の名前を割り当てることができます。この機能は他の言語では別名 (alias) づけとして知られています。Python を一見しただけでは、別名づけの重要性は分からないことが多く、変更不能の基本型 (数値、文字列、タプル) を扱うときには無視して差し支えありません。しかしながら、別名付けには、リストや辞書、またプログラムの外部にある実体 (ファイル、ウィンドウ、など) を表現するためのほとんどの型が入った Python コードで意味付け

を行う上で (意図的な!) 効果があります。別名付けはいくつかの点でポインタのように振舞うので、通常はプログラムに利するように使われます。例えば、オブジェクトの受け渡しは、実装上はポインタが渡されるだけなのでコストの低い操作になります; また、関数があるオブジェクトを引数として渡されたとき、関数の呼び出し側からオブジェクトに対する変更を見ることができます — これにより、Pascal にあるような二つの引数渡し機構をもつ必要をなくしています。

## 9.2 Python のスコープと名前空間

クラスを紹介する前に、Python のスコープ規則についてあることを話しておかなければなりません。クラス定義はある巧みなトリックを名前空間に施すので、何が起きているのかを完全に理解するには、スコープと名前空間がどのように動作するかを理解する必要があります。ちなみに、この問題に関する知識は全ての Python プログラマにとって有用です。

まず定義から始めましょう。

名前空間 (*namespace*) とは、名前からオブジェクトへの対応付け (mapping) です。ほとんどの名前空間は、現状では Python の辞書として実装されていますが、そのことは通常は (パフォーマンス以外では) 目立つことはないし、将来は変更されるかもしれません。名前空間の例には、組み込み名の集合 (`abs()` 等の関数や組み込み例外名)、モジュールないのグローバルな名前; 関数を呼び出したときのローカルな名前、があります。その意味では、オブジェクトの属性からなる集合もまた、名前空間を形成します。名前空間について知っておくべき重要なことは、異なった名前空間にある名前の間には全く関係がないということです; 例えば、二つの別々のモジュールの両方で関数 “maximize” という関数を定義することができ、定義自体は混同されることはありません — モジュールのユーザは名前の前にモジュール名をつけなければなりません。

ところで、属性という言葉は、ドットに続く名前すべてに対して使っています — 例えば式 `z.real` で、`real` はオブジェクト `z` の属性です。厳密に言えば、モジュール内の名前に対する参照は属性の参照です: 式 `modname.funcname` では、`modname` はあるモジュールオブジェクトで、`funcname` はその属性です。この場合には、たまたまモジュールの属性とモジュール内のグローバルな名前の間にはこの場合はたまたま、モジュールの属性とモジュールで定義されているグローバル名の間には、直接的な対応付けがされます: これらの名前は同じ名前空間を共有しているのです!<sup>1</sup>

属性は読取り専用にも、書き込み専用にもできます。後者の場合、属性に代入することができます。モジュール属性は書き込み可能です: `modname.the_answer = 42` と書くことができます。書き込み可能な属性は、`del` 文で削除することもできます。例えば、`del modname.the_answer` は、`modname` で指定されたオブジェクトから属性 `the_answer` を除去します。

名前空間は様々な時点で作成され、その寿命も様々です。組み込みの名前が入った名前空間は Python インタプリタが起動するときに作成され、決して削除されることはありません。モジュールのグローバルな名前空間は、モジュール定義が読み込まれたときに作成されます; 通常、モジュールの名前空間は、インタプリタが終了するまで残ります。インタプリタのトップレベルで実行された文は、スクリプトファイルから読み出されたものでも対話的に読み出されたものでも、`__main__` という名前のモジュールの一部分であるとみなされるので、独自の名前空間を持つことになります。(組み込みの名前は実際にはモジュール内に存在します; そのモジュールは `__builtin__` と呼ばれています。)

関数のローカルな名前空間は、関数が呼び出されたときに作成され、関数から戻ったときや、関数内で例外が送出され、かつ関数内で処理されなかった場合に削除されます。(実際には、忘れられる、と言ったほうが起きていることをよく表しています。) もちろん、再帰呼出しのときには、各々の呼び出しで各自のローカルな名前空間があります。

スコープ (*scope*) とは、ある名前空間が直接アクセスできる (directly accessible) ような、Python プログ

<sup>1</sup> 例外が一つあります。モジュールオブジェクトには、秘密の読取り専用の属性 `__dict__` があり、モジュールの名前空間を実装するために使われている辞書を返します; `__dict__` という名前は属性ですが、グローバルな名前ではありません。この属性を利用すると名前空間の実装に対する抽象化を侵すことになるので、プログラムを検死するデバッガのような用途に限るべきです。



ラムのテキスト上の領域です。“直接アクセス可能”とは、限定なし (unqualified) である名前を参照した際に、その名前空間から名前を見つけようと試みることを意味します。

スコープは静的に決定されますが、動的に使用されます。実行中はいつでも、直接名前空間にアクセス可能な、少なくとも三つの入れ子になったスコープがあります: 最初に検索される最も内側のスコープには、ローカルな名前が入っています; あるいは、最も内側のスコープを囲んでいる関数群のスコープで、最も近傍のスコープから検索を始めます; 中間のスコープが次に検索され、このスコープには現在のモジュールのグローバルな名前が入っています; (最後に検索される) 最も外側のスコープは、組み込みの名前が入った名前空間です。

名前がグローバルであると宣言されている場合、その名前に対する参照や代入は全て、モジュールのグローバルな名前が入った中間のスコープに対して直接行われます。そうでない場合、最も内側のスコープより外側にある変数は全て読み出し専用となります。

通常、ローカルスコープは(プログラムテキスト上の) 現在の関数のローカルな名前を参照します。関数の外側では、ローカルスコープはグローバルな名前空間と同じ名前空間: モジュールの名前空間を参照します。クラスを定義すると、ローカルスコープの中にもう一つ名前空間が置かれます。

スコープはテキスト上で決定されていると理解することが重要です: モジュール内で定義される関数のグローバルなスコープは、関数がどこから呼び出されても、どんな別名をつけて呼び出されても、そのモジュールの名前空間になります。反対に、実際の名前の検索は実行時に動的に行われます — とはいえ、言語の定義は、“コンパイル”時の静的な名前解決の方向に進化しているので、動的な名前解決に頼ってはいけません! (事実、ローカルな変数は既に静的に決定されています。)

Python 特有の癖として、代入を行うと名前がいつも最も内側のスコープに入るというものがあります。代入はデータのコピーを行いません — 単に名前をオブジェクトに結びつける (bind) だけです。オブジェクトの削除でも同じです: ‘del x’ は、x をローカルスコープが参照している名前空間から削除します。実際、新たな名前を導入する操作は全てローカルスコープを用います: とりわけ、import 文や関数定義は、モジュールや関数の名前をローカルスコープに結び付けます。(global 文を使えば、特定の変数がグローバルスコープにあることを示せます。)

## 9.3 クラス初見

クラスでは、新しい構文を少しと、三つの新たなオブジェクト型、そして新たな意味付けをいくつか取り入れています。

### 9.3.1 クラス定義の構文

クラス定義の最も単純な形式は、以下のようになります:

```
class ClassName:
    <文-1>
    .
    .
    .
    <文-N>
```

関数定義 (def 文) と同様、クラス定義が効果をもつにはまず実行しなければなりません。(クラス定義を if 文の分岐先や関数内部に置くことも、考え方としてはありえます。)

実際には、クラス定義の内側にある文は、通常は関数定義になりますが、他の文を書くこともでき、それがそれが役に立つこともあります — これについては後で述べます。クラス内の関数定義は通常、メソッド

ドの呼び出し規約で決められた独特の形式の引数リストを持ちます — これについても後で述べます。

クラス定義に入ると、新たな名前空間が作成され、ローカルな名前空間として使われます — 従って、ローカルな変数に対する全ての代入はこの新たな名前空間に名要ります。特に、関数定義を行うと、新たな関数の名前はこの名前空間に結び付けられます。

クラス定義から普通に (定義の終端に到達して) 抜けると、クラスオブジェクト (*class object*) が生成されます。クラスオブジェクトは、基本的にはクラス定義で作成された名前空間の内容をくるむラップ (wrapper) です; クラスオブジェクトについては次の節で詳しく学ぶことにします。(クラス定義に入る前に有効だった) 元のローカルスコープが復帰し、生成されたクラスオブジェクトは復帰したローカルスコープにクラス定義のヘッダで指定した名前 (上の例では `ClassName`) で結び付けられます。

### 9.3.2 クラスオブジェクト

クラス・オブジェクトでは2種類の演算: 属性参照とインスタンス生成をサポートしています。

属性参照 (*attribute reference*) は、Python におけるすべての属性参照で使われている標準的な構文、`obj.name` を使います。クラスオブジェクトが生成された際にクラスの名前空間にあった名前すべてが有効な属性名です。従って、以下のようなクラス定義:

```
class MyClass:
    "A simple example class"
    i = 12345
    def f(self):
        return 'hello world'
```

では、`MyClass.i` と `MyClass.f` は妥当な属性参照であり、それぞれ整数とメソッド・オブジェクトを返します。クラス属性に代入を行うこともできます。従って、`MyClass.i` の値を代入して変更できます。`__doc__` も有効な属性で、そのクラスに属している docstring、この場合は "A simple example class" を返します。

クラスの インスタンス生成 (*instantiation*) には関数のような表記法を使います。クラスオブジェクトのことを、単にクラスの新しいインスタンスを返すパラメタを持たない関数かのように扱います。例えば (上記のクラスでいえば):

```
x = MyClass()
```

は、クラスの新しいインスタンス (*instance*) を生成し、そのオブジェクトをローカル変数 `x` へ代入します。

インスタンス生成操作 (クラスオブジェクトの “呼出し”) を行うと、空のオブジェクト (empty object) を生成します。多くのクラスは、オブジェクトを作成する際に、既知の初期状態になってほしいと望んでいます。従って、クラスでは `__init__()` という名前の特別なメソッド定義することができます。例えば以下のようにします:

```
def __init__(self):
    self.data = []
```

クラスが `__init__()` メソッドを定義している場合、クラスのインスタンスを生成すると、新しく生成されたクラスインスタンスに対して自動的に `__init__()` を呼び出します。従って、この例では、新たな初期済みのインスタンスを以下のようにして得ることができます:

```
x = MyClass()
```

もちろん、より大きな柔軟性を持たせるために、`__init__()` メソッドに複数の引数をもたせることができます。その場合、クラスのインスタンス生成操作に渡された引数は `__init__()` に渡されます。例えば以下のように:

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

### 9.3.3 インスタンスオブジェクト

ところで、インスタンスオブジェクトを使うと何ができるのでしょうか？ インスタンスオブジェクトが理解できる唯一の操作は、属性の参照です。有効な属性の名前には二種類あります。

一つ目の属性の種類は データ属性 (*data attribute*) と呼ぶことにしましょう。これは、これは Smalltalk の “インスタンス変数” (instance variable) や C++ の “データメンバ” (data member) に相当します。データ属性を宣言する必要はありません; ローカルな変数と同様に、これらの属性は最初に代入された時点で湧き出てきます。例えば、上で生成した `MyClass` のインスタンス `x` に対して、以下のコード断片を実行すると、値 16 を印字し、`x` の痕跡は残りません。

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print x.counter
del x.counter
```

インスタンスオブジェクトが理解できる属性の参照の二つ目はメソッド (*method*) です。メソッドとは、オブジェクトに “属している” 関数のことです。(Python では、メソッドという用語はクラスインスタンスだけのものではありません: オブジェクト型にもメソッドを持つことができます。例えば、リストオブジェクトには、`append`, `insert`, `remove`, `sort` などといったメソッドがあります。とはいえ、以下では特に明記しない限り、クラスのインスタンスオブジェクトのメソッドだけを意味するものとして使うことにします。)

インスタンスオブジェクトで有効なメソッドは、そのクラスによって決まります。定義により、クラスの全ての属性は、(ユーザが定義した) 関数オブジェクトインスタンスオブジェクトの妥当なメソッド名は、そのクラスによって決まります。定義により、(利用者定義の) 関数オブジェクトになっているクラス属性は全て、そのインスタンスの対応するメソッドとなります。従って、例では、`MyClass.f` は関数なので、`x.f` はメソッドの参照として有効です。しかし、`MyClass.i` は関数ではないので、`x.i` はメソッドの参照として有効ではありません。`x.f` は `MyClass.f` と同じものではありません — 関数オブジェクトではなく、メソッドオブジェクト (*method object*) です。

### 9.3.4 メソッドオブジェクト

普通、メソッドはその場ですぐに呼び出されます:

```
x.f()
```

私たちの例では、上のコードは文字列 'hello world' を返すでしょう。しかしながら、必ずしもメソッドを正しい方法で呼び出さなければならないわけではありません: `x.f` はメソッドオブジェクトであり、どこかに記憶しておいて後で呼び出すことができます。例えば以下のコード:

```
xf = x.f
while True:
    print xf()
```

は、'hello world' を時が終わるまで印字し続けるでしょう。

メソッドが呼び出されるときには実際には何が起きているのでしょうか? `f` の関数定義では引数の一つ指定していたにもかかわらず、上記では `x.f()` が引数なしで呼び出されたことに気付いているかもしれませんね。引数はどうなったのでしょうか? たしか、引数が必要な関数を引数無しで呼び出すと、Python が例外を送出するはずですが — たとえその引数が実際には使われなくても...

実際、もう答は想像できているかもしれませんね: メソッドについて特別なこととして、オブジェクトが関数の第 1 引数として渡される、ということがあります。我々の例では、`x.f()` という呼び出しは、`MyClass.f(x)` と厳密に等価なものです。一般に、 $n$  個の引数リストもったメソッドの呼出しは、そのメソッドのオブジェクトを最初の引数の前に挿入した引数リストでメソッドに対応する関数を呼び出すことと等価です。

もしもまだメソッドの働きかたを理解できなければ、一度実装を見てみると事情がよく分かるかもしれません。データ属性ではないインスタンス属性が参照された時は、そのクラスが検索されます。その名前が有効なクラス属性を表している関数オブジェクトなら、インスタンスオブジェクトと見つかった関数オブジェクト (へのポインタ) を抽象オブジェクト: すなわちメソッドオブジェクトにパック (pack) して作成します。メソッドオブジェクトは、引数リストを伴って呼び出される際に再度アンパック (unpack) され、新たな引数リストがインスタンスオブジェクトとオリジナルの引数リストから新たな引数リストが構成され、新たな引数リストを使って関数オブジェクトを呼び出します。

## 9.4 いろいろな注意点

データ属性は同じ名前のメソッド属性を上書きしてしまいます; 大規模なプログラムでみつけにくいバグを引き起こすことがあるこの偶然的な名前の衝突を避けるには、衝突の可能性を最小限にするような規約を使うのが賢明です。可能な規約としては、メソッド名を大文字で始める、データ属性名の先頭に短い一意的な文字列 (あるいはただの下線) をつける、またメソッドには動詞、データ属性には名詞を用いる、などがあります。

データ属性は、メソッドから参照できると同時に、通常のオブジェクトのユーザ (“クライアント”) から参照できます。言い換えると、クラスは純粋な抽象データ型として使うことができません。実際、Python では、データ隠蔽を補強するための機構はなにもありません — データの隠蔽はすべて規約に基づいています。(逆に、C 言語で書かれた Python の実装では実装の詳細を完全に隠蔽し、必要に応じてオブジェクトへのアクセスを制御できます; この機構は C 言語で書かれた Python 拡張で使うことができます)

クライアントはデータ属性を注意深く扱うべきです — クライアントは、メソッドを使うことで維持しているデータ属性の不変性を踏みにじり、台無しにするかもしれません。クライアントは、名前の衝突が回避されている限り、メソッドの有効性に影響を及ぼすことなくインスタンスに独自の属性を追加することができる、ということに注意してください — ここでも、名前付けの規約は頭痛の種を無くしてくれます。

データ属性を (またはその他のメソッドも!) メソッドの中で参照するための短縮された記法はありませ

ん。私は、この仕様が実際にメソッドの可読性を高めていると考えています: あるメソッドを眺めているときにローカルな変数とインスタンス変数を混同する可能性はまったくありません。

慣習として、メソッドの最初の引数を、しばしば `self` と呼びます。この名前付けは単なる慣行でしかありません: `self` という名前は、Python では何ら特殊な意味を持ちません。(とはいえ、この慣行に従わないと、コードは他の Python プログラマにとってやや読みにくいものとなります。また、クラスブラウザ (*class browser*) プログラムがこの慣行をあてにして書かれていないとも限りません。)

クラス属性である関数オブジェクトはいずれも、そのクラスのインスタンスのためのメソッドを定義しています。関数定義は、テキスト上ではクラス定義の中に入っていなければならないわけではありません: 関数オブジェクトをクラスのローカルな変数の中に代入するのも OK です。例えば以下のコードのようにします:

```
# クラスの外側で定義された関数
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1
    def g(self):
        return 'hello world'
    h = g
```

これで、`f`、`g`、および `h` は、すべて `C` の属性であり関数オブジェクトを参照しています。従って、これら全ては、`C` のインスタンスのメソッドとなります — `h` は `g` と全く等価です。これを実践しても、大抵は単にプログラムの読者に混乱をもたらすだけなので注意してください。

メソッドは、`self` 引数のメソッド属性を使って、他のメソッドを呼び出すことができます:

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

メソッドは、通常、関数と同じようにして、グローバルな名前を参照してもかまいません。あるメソッドに関連付けられたグローバルなスコープは、クラス定義の入っているモジュールになります。(クラス自体はグローバルなスコープとして用いられることはありません!) メソッドでグローバルなデータを使う良い理由はほとんどありませんが、グローバルなスコープを使う合法的な使い方は多々あります: 一つ挙げると、メソッド内では、グローバルなスコープに `import` された関数やモジュールや、その中で定義された関数やクラスを使うことができます。通常、メソッドの入っているクラス自体はグローバルなスコープ内で定義されています。次の章では、メソッドが自分のクラスを参照する理由として正当なものを見てみましょう!

## 9.5 継承

言うまでもなく、継承の概念をサポートしない言語機能は“クラス”と呼ぶに値しません。導出クラス (*derived class*) を定義する構文は以下ようになります:



```
class DerivedClassName(BaseClassName):
    <文-1>
    .
    .
    .
    <文-N>
```

基底クラス (base class) の名前 `BaseClassName` は、派生クラス定義の入っているスコープで定義されていなければなりません。基底クラス名のかわりに式を入れることもできます。これは以下のように、

```
class DerivedClassName(modname.BaseClassName):
```

基底クラスが別モジュールで定義されているとき便利です。

導出クラス定義の実行は、基底クラスの場合と同じように進められます。クラスオブジェクトが構築される時、基底クラスが記憶されます。記憶された基底クラスは、属性参照を解決するために使われます: 要求された属性がクラスに見つからなかった場合、基底クラスから検索されます。この規則は、基底クラスが他の何らかのクラスから導出されたものであった場合、再帰的に適用されます。

導出クラスのインスタンス化では、特別なことは何もありません: `DerivedClassName()` はクラスの新たなインスタンスを生成します。メソッドの参照は以下のようにして解決されます: まず対応するクラス属性が検索されます。検索は、必要に応じ、基底クラス連鎖を下って行われ、検索の結果として何らかの関数オブジェクトがもたらされた場合、メソッド参照は有効なものとなります。

導出クラスは基底クラスのメソッドを上書き (override) してもかまいません。メソッドは同じオブジェクトの別のメソッドを呼び出す際に何ら特殊な権限を持ちません。このため、ある基底クラスのメソッドが、同じ基底クラスで定義されているもう一つのメソッド呼び出しを行っている場合、実際には導出クラスで上書きされた何らかのメソッドが呼び出されることになるかもしれません。(C++ プログラマへ: Python では、すべてのメソッドは事実上 virtual です。)

導出クラスで上書きしているメソッドでは、実際は単に基底クラスの同名のメソッドを置き換えるだけではなく、拡張を行いたいかもしれません。基底クラスのメソッドを直接呼び出す簡単な方法があります: 単に `'BaseClassName.methodname(self, arguments)'` を呼び出すだけです。この仕様は、場合によってはクライアントでも役に立ちます。(この呼び出し方が動作するのは、基底クラスがグローバルなスコープ内で定義されているか、直接 import されている場合だけなので注意してください。)

### 9.5.1 多重継承

Python では、限られた形式の多重継承 (multiple inheritance) もサポートしています。複数の基底クラスをもつクラス定義は以下ようになります:

```
class DerivedClassName(Base1, Base2, Base3):
    <文-1>
    .
    .
    .
    <文-N>
```

多重継承への意味付けを説明する上で必要な唯一の規則は、クラス属性の参照を行うときに用いられる名前解決の規則 (resolution rule) です。解決規則は深さ優先 (depth-first)、左から右へ (left-to-right) となっています。従って、ある属性が `DerivedClassName` で見つからなければ `Base1` で検索され、次に `Base1` の基底クラスで (再帰的に) 検索されます。それでも見つからなければはじめて `Base2` で検索される、と

いった具合です。

(人によっては、幅優先 (breadth first) — Base2 と Base3 を検索してから Base1 の基底クラスで検索する — のほうが自然のように見えます。しかしながら、幅優先の検索では、Base1 の特定の属性のうち、実際に定義されているのが Base1 なのか、その基底クラスなのかを知らなければ、Base2 の属性との名前衝突がどんな結果をもたらすのか分からないことになります。深さ優先規則では、Base1 の直接の属性と継承された属性とを区別しません。)

Python では偶然的な名前の衝突を慣習に頼って回避しているので、見境なく多重継承の使用すると、メンテナンスの悪夢に陥ることは明らかです。多重継承に関するよく知られた問題は、二つのクラスから導出されたクラスがたまたま共通の基底クラスを持つ場合です。この場合になにが起こるかを結論することは簡単です (インスタンスは共通の基底クラスで使われている “インスタンス変数” の単一のコピーを持つことになります) が、この意味付けが何の役に立つのかは明らかではありません。

## 9.6 プライベート変数

クラスプライベート (class-private) の識別子に関して限定的なサポートがなされています。\_\_spam (先頭に二個以上の下線文字、末尾に高々一個の下線文字) という形式の識別子、テキスト上では `__classname__spam` へと置換されるようになりました。ここで `classname` は、現在のクラス名から先頭の下線文字をはぎとった名前になります。このような難号化 (mangle) は、識別子の文法的な位置にかかわらず行われるので、クラスプライベートなインスタンス変数やクラス変数、メソッド、そしてグローバル変数を定義するための利用できます。また、このクラスにとってプライベートなインスタンス変数を他のクラスのインスタンスに格納するために使うことさえできます。難号化した名前が 255 文字より長くなるときは、切り詰めが起こるかもしれません。クラスの外側や、クラス名が下線文字だけからできているときには、難号化加工は起こりません。

名前の難号化は、クラスにおいて、“プライベートな” インスタンス変数やメソッドを定義する際に、導出クラスで定義されるインスタンス変数を気にしたり、クラスの外側のコードからインスタンス変数をいじりまわすことがないように簡単に定義できるようにするためのものです。難号化の規則は主に不慮の事故を防ぐためのものだということに注意してください; 確信犯的な方法で、プライベートとされている変数にアクセスしたり変更することは依然として可能なのです。デバッガのような特殊な状況では、この仕様は便利です。そのため、この抜け穴は塞がれていません。(些細なバグ: 基底クラスと同じ名前のクラスを導出すると、基底クラスのプライベート変数を使えるようになります。)

`exec` や `eval()` や `evalfile()` へ渡されたコードでは、呼出し元のクラス名を現在のクラスと見なさないことに注意してください; この仕様は `global` 文の効果と似ており、その効果もまた同様に、バイトコンパイルされたコードに制限されています。同じ制約が `getattr()` と `setattr()` と `delattr()` にも適用されます。また、`__dict__` を直接参照するときにも適用されます。

## 9.7 残りののはし

Pascal の “レコード (record)” や、C 言語の “構造体 (struct)” のような、名前付きのデータ要素を一まとめにするデータ型があると便利なのがたまにあります。空のクラス定義を使うとうまくできます:

```

class Employee:
    pass

john = Employee() # 空の従業員レコードを作る

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000

```

ある特定の抽象データ型を要求する Python コードの断片には、そのデータ型のメソッドをエミュレーションするクラスを代わりに渡すことができます。例えば、ファイルオブジェクトから何らかのデータを書式化する関数がある場合、`read()` と `readline()` を持つクラスを定義して、ファイルではなく文字列バッファからデータを書式するようにしておき、引数として渡すことができます。

インスタンスメソッドオブジェクトにもまた、属性があります: `m.im_self` はメソッドの属しているインスタンスオブジェクトで、`m.im_func` はメソッドに対応する関数オブジェクトです。

### 9.7.1 例外はクラスであってもよい

ユーザ定義の例外をクラスとして識別することもできます。このメカニズムを使って、拡張可能な階層化された例外を作成することができます。

新しく二つの (意味付け的な) 形式の `raise` 文ができました:

```

raise Class, instance

raise instance

```

第一の形式では、`instance` は `Class` またはその導出クラスのインスタンスでなければなりません。第二の形式は以下の表記:

```

raise instance.__class__, instance

```

の短縮された記法です。

`except` 節には、文字列オブジェクトだけでなくクラスを並べることができます。`except` 節のクラスは、同じクラスか基底クラスの例外のときに互換 (compatible) となります (逆方向では成り立ちません — 導出クラスの例外がリストされている `except` 節は基底クラスの例外と互換ではありません)。例えば、次のコードは、`B`, `C`, `D` を順序通りに出力します:



```

class B:
    pass
class C(B):
    pass
class D(C):
    pass

for c in [B, C, D]:
    try:
        raise c()
    except D:
        print "D"
    except C:
        print "C"
    except B:
        print "B"

```

except 節が逆に並んでいた場合（‘except B’ が最初にくる場合）、B, B, B と出力されるはずだったことに注意してください — 最初に一致した except 節が駆動されるのです。

処理されないクラスの例外に対してエラーメッセージが出力されるとき、まずクラス名が出力され、続いてコロン、スペース、最後に組み込み関数 `str()` を使って文字列に変換したインスタンスが出力されます。

## 9.8 イテレータ (iterator)

すでに気づいているでしょうが、`for` 文を使うとほとんどのコンテナオブジェクトにわたってループを行います：

```

for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
    print char
for line in open("myfile.txt"):
    print line

```

こうしたアクセス方法は明確で、簡潔で、かつ便利なものです。イテレータの使用は Python 全体に普及していて、統一性をもたらしています。背後では、`for` 文はコンテナオブジェクトの `iter()` を呼び出しています。この関数は `next()` メソッドの定義されたイテレータオブジェクトを返します。`next()` メソッドは一度コンテナ内の要素に一度に一つずつアクセスします。コンテナ内にアクセスすべき要素がなくなると、`next()` は `StopIteration` 例外を送出し、`for` ループを終了させます。実際にどのように動作するかを以下の例に示します：

```

>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()

Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    it.next()
StopIteration

```

イテレータプロトコルの背後にあるメカニズムを一度目にすれば、自作のクラスにイテレータとしての振る舞いを追加するのは簡単です。\_\_iter\_\_() メソッドを定義して、next() メソッドを持つオブジェクトを返すようにしてください。クラス自体で next() を定義している場合、\_\_iter\_\_() では単に self を返すようにできます:

```

>>> class Reverse:
    "Iterator for looping over a sequence backwards"
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

>>> for char in Reverse('spam'):
    print char

m
a
p
s

```

## 9.9 ジェネレータ (generator)

ジェネレータは、イテレータを作成するための簡潔で強力なツールです。ジェネレータは通常の変数のように書かれますが、何らかのデータを返すときには yield 文を使います。next() が呼び出されるたびに、ジェネレータは以前に中断した処理を再開します (ジェネレータは、全てのデータ値と最後にどの文が実行されたかを記憶しています)。以下の例を見れば、ジェネレータがとても簡単に作成できることがわかります:

```
>>> def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]

>>> for char in reverse('golf'):
    print char

f
l
o
g
```

ジェネレータを使ってできることは、前節で記述したクラスに基づいたイテレータを使えばできます。ジェネレータを使うとコンパクトに記述できるのは、`__iter__()` と `next()` メソッドが自動的に作成されるからです。

ジェネレータのもう一つの重要な機能は、呼び出しごとにローカル変数と実行状態が自動的に保存されるということです。これにより、`self.index` や `self.data` といったクラス変数を使ったアプローチよりも簡単に関数を書くことができるようになります。

メソッドを自動生成したりプログラムの実行状態を自動保存するほかに、ジェネレータは終了時に自動的に `StopIteration` を送出します。これらの機能を組み合わせると、通常の間数を書くのに比べ、全く苦労することなく簡単にイテレータを生成することができます。



## 標準ライブラリの簡単なツアー

### 10.1 オペレーティングシステムへのインタフェース

os モジュールは、オペレーティングシステムと対話するための何ダースもの関数を提供しています:

```
>>> import os
>>> os.system('time 0:02')
0
>>> os.getcwd()          # 現在の作業ディレクトリを返す
'C:\\Python24'
>>> os.chdir('/server/accesslogs')
```

‘from os import \*’ではなく、‘import os’形式を使うようにしてください。そうすることで、動作が大きく異なる組み込み関数 open() が os.open() で隠蔽されるのを避けられます。

組み込み関数 dir() および help() は、os のような大規模なモジュールで作業をするときに、対話的な操作上の助けになります:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

ファイルやディレクトリの日常的な管理作業のために、より簡単に使える高レベルインタフェースが shutil モジュールで提供されています:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
>>> shutil.move('/build/executables', 'installdir')
```

### 10.2 ファイルのワイルドカード表記

glob モジュールでは、ディレクトリのワイルドカード検索からファイルのリストを生成するための関数を提供しています:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

## 10.3 コマンドライン引数

広く使われているユーティリティスクリプトでは、しばしばコマンドライン引数の処理を呼び出します。これらの引数は `sys` モジュールの `argv` 属性にリストとして記憶されます。例えば、以下の出力は、`'python demo.py one two three'` をコマンドライン上で起動した際に得られるものです：

```
>>> import sys
>>> print sys.argv
['demo.py', 'one', 'two', 'three']
```

`getopt` モジュールは、`sys.argv` を UNIX の `getopt()` 関数の慣習に従って処理します。より強力で柔軟性のあるコマンドライン処理機能は、`optparse` モジュールで提供されています。

## 10.4 エラー出力のリダイレクトとプログラムの終了

`sys` モジュールには、`stdin`、`stdout`、および `stderr` を表す属性値も存在します。後者の `stderr` は、警告やエラーメッセージを出力して、`stdout` がリダイレクトされた場合でもそれらが読めるようにする上で便利です：

```
>>> sys.stderr.write('Warning, log file not found starting a new one')
Warning, log file not found starting a new one
```

`'sys.exit()'` は、スクリプトを終了させるもっとも直接的な方法です。

## 10.5 文字列のパターンマッチング

`re` モジュールでは、より高度な文字列処理のための正規表現 (regular expression) を提供しています。正規表現は複雑な一致検索や操作に対して簡潔で最適化された解決策を与えます：

```
>>> import re
>>> re.findall(r'\b[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

最小限の機能だけが必要ななら、読みやすくデバッグしやすい文字列メソッドの方がお勧めです：

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

## 10.6 数学

`math` モジュールでは、根底にある浮動小数点演算のための C 言語ライブラリ関数にアクセスする手段を提供しています：

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

random モジュールでは、乱数に基づいた要素選択のためのツールを提供しています:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(xrange(100), 10)    # 要素を戻さないサンプリング
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # ランダムな浮動小数点数
0.17970987693706186
>>> random.randrange(6)    # range(6) からランダムに選ばれた整数
4
```

## 10.7 インターネットへのアクセス

インターネットにアクセスしたり、インターネットプロトコルを処理したりするための数多くのモジュールがあります。その中でも最も単純な二つのモジュールは、URL を指定してデータを取得するための `urllib2` と、メールを送信するための `smtplib` です:

```
>>> import urllib2
>>> for line in urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
...     if 'EST' in line:    # look for Eastern Standard Time
...         print line

<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@tmp.org', 'jceasar@tmp.org',
... """To: jceasar@tmp.org
From: soothsayer@tmp.org

Beware the Ides of March.
""")
>>> server.quit()
```

## 10.8 日付と時刻

`datetime` モジュールは、日付や時刻を操作するためのクラスを、単純な方法と複雑な方法の両方で供給しています。日付や時刻に対する算術がサポートされている一方、実装では出力の書式化や操作のための効率的なデータメンバ抽出に重点を置いています。このモジュールでは、タイムゾーンに対応したオブジェクトもサポートしています。

```
# dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y or %d%b %Y is a %A on the %d day of %B")
'12-02-03 or 02Dec 2003 is a Tuesday on the 02 day of December'

# dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

## 10.9 データ圧縮

データの書庫化や圧縮で広く使われている形式については、zlib、gzip、bz2、zipfile、およびtarfile といったモジュールで直接サポートしています。

```
>>> import zlib
>>> s = 'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
'witch which has which witches wrist watch'
>>> zlib.crc32(t)
-1438085031
```

## 10.10 パフォーマンスの計測

Python ユーザの中には、同じ問題を異なったアプローチで解いた際の相対的なパフォーマンスについて知りたいという深遠な興味を抱いている人がいます。Python では、そういった疑問に即座に答える計測ツールを提供しています。

例えば、引数の入れ替え操作に対して、伝統的なアプローチの代わりにタプルのパックやアンパックを使ってみたい気持ちになるかもしれません。timeit モジュールを使うと、伝統的なアプローチのほうが高速であることがすぐに実証されます：

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.60864915603680925
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.8625194857439773
```

timeit では高い粒度レベルを提供しているのに対し、profile や pstats モジュールではより大きなコードブロックにおいて律速となる部分を判定するためのツールを提供しています。



## 10.11 品質管理

高い品質のソフトウェアを開発するための一つのアプローチは、全ての関数に対して開発と同時にテストを書き、開発の過程で頻繁にテストを走らせるというものです。

doctest モジュールでは、モジュールを検索して、プログラムの docstring に埋め込まれたテストの評価を行うためのツールを提供しています。テストの作り方は単純で、典型的な呼び出し例とその結果を docstring にカット&ペーストするというものです。この作業は、ユーザに使用例を与えるという意味でドキュメントの情報を増やすと同時に、ドキュメントに書かれている内容が正しいかどうか doctest モジュールが確認できるようにしています：

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print average([20, 30, 70])
    40.0
    """
    return sum(values, 0.0) / len(values)

import doctest
doctest.testmod() # automatically validate the embedded tests
```

unittest モジュールは doctest モジュールほど気楽に使えるものではありませんが、より網羅的なテストセットを別のファイルで管理することができます：

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        self.assertRaises(ZeroDivisionError, average, [])
        self.assertRaises(TypeError, average, 20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

## 10.12 一揃いあつらえ済み

Python には“一揃いあつらえ済み (batteries included)” 哲学があります。この哲学は、洗練され、安定した機能を持つ Python の膨大なパッケージ群に如実に表れています。例えば：

\* The xmlrpclib および SimpleXMLRPCServer モジュールは、遠隔手続き呼び出し (remote procedure call) を全くたいしたことのない作業に変えてしまいます。名前とは違い、XML を扱うための直接的な知識は必要ありません。

\* The email パッケージは、MIME やその他の RFC 2822 に基づくメッセージ文書を含む電子メールメッセージを管理するためのライブラリです。実際にメッセージを送信したり受信したりする smtplib や poplib と違って、email パッケージには (添付文書を含む) 複雑なメッセージ構造の構築やデコードを行ったり、インターネット標準のエンコードやヘッダプロトコルの実装を行ったりするための完全なツールセットを備えています。

\* xml.dom および xml.sax パッケージでは、一般的なデータ交換形式である XML を解析するための頑健なサポートを提供しています。同様に、csv モジュールでは、広く用いられているデータベース形式

のデータを直接読み書きする機能をサポートしています。これらのモジュールやパッケージは併用することで、Python アプリケーションと他のツール群との間でのデータ交換を劇的に簡単化します。

\* 国際化に関する機能は、`gettext`、`locale`、および `codecs` パッケージといったモジュール群でサポートされています。

## さあ何を？

このチュートリアルを読んだことで、おそらく Python を使ってみようという関心はますます強くなったことでしょう — 現実世界の問題を解決するために、Python を適用してみたくなったはずです。さて、それでは何をしたらよいのでしょうか？

*Python* ライブラリリファレンスを読むか、少なくともざっと読み流すとよいでしょう。ライブラリリファレンスでは、データ型、関数、そして Python プログラムを書く上で大いに時間の節約となるモジュールについて、(簡潔ではあるけれども) 完全なリファレンスマニュアルを提供しています。標準的な Python の配布物には、C 言語と Python の両方のコードがたくさん収められています; UNIX メールボックスを読み出したり、HTTP を介してドキュメントを取得したり、乱数を生成したり、コマンドラインオプションを解析したり、CGI プログラムを書いたり、データを圧縮したり、その他にももっとたくさんのモジュールがあります; ライブラリリファレンスにざっと目を通すだけでも、Python で何ができるかアイデアを得ることができますでしょう。

主要な Python Web サイトは <http://www.python.org/> です; このサイトには、コード、ドキュメント、そして Web のあちこちの Python に関連したページへのポインタがあります。この Web サイトは世界のあちこちのさまざまな場所、例えばヨーロッパ、日本、オーストラリアなどでミラーされています。地理的な位置によっては、メインのサイトよりミラーのほうが速いかもしれません。非公式なサイトには <http://starship.python.net/> があります。ここには、一群の Python 関連の個人的ホームページがあります; 多くの人がここにダウンロード可能なソフトウェアを置いています。

Python Package Index (PyPI). にあるサードパーティ製モジュールリポジトリに行けば、さらに多くのユーザによって作成された Python モジュールを見つけることができます。

Python に関する質問をしたり、問題を報告するために、ニュースグループ `comp.lang.python` に投稿したり、`python-list@python.org` のメーリングリストに送信することができます。ニュースグループとメーリングリストは相互接続されているので、どちらかにポストされたメッセージは自動的にもう一方にも転送されます。一日に約 120 通程度、質問を (そしてその解凍)、新機能の提案、新たなモジュールのアナウンスが投稿されています。投稿する前に、必ずよく出される質問 (Frequently Asked Questions, FAQ ともいいます) のリストを確認するか、Python ソースコード配布物の ‘Misc/’ ディレクトリを探すようにしてください。メーリングリストのアーカイブは <http://www.python.org/pipermail/> で入手することができます。FAQ では、何度も繰り返し現れる質問の多くに答えています。読者の抱えている問題に対する解答がすでに入っているかもしれません。



## 対話入力編集とヒストリ置換

あるバージョンの Python インタプリタでは、Korn シェルや GNU Bash シェルに見られる機能に似た、現在の入力行に対する編集機能やヒストリ置換機能をサポートしています。この機能は *GNU Readline* ライブラリを使って実装されています。このライブラリは Emacs スタイルと vi スタイルの編集をサポートしています。ライブラリには独自のドキュメントがあり、ここでそれを繰り返すつもりはありません; とはいえ、基本について簡単に解説することにします。ここで述べる対話的な編集とヒストリについては、UNIX 版と CygWin 版のインタプリタでオプションとして利用することができます。

この章では、Mark Hammond の PythonWin パッケージや、Python とともに配布される Tk ベースの環境である IDLE にある編集機能については解説しません。NT 上の DOS ボックスやその他の DOS および Windows 類で働くコマンド行ヒストリ呼出しもまた別のものです。

### A.1 行編集

入力行の編集がサポートされている場合、インタプリタが一次または二次プロンプトを出力している際にはいつでも有効になっています。現在の行は、慣例的な Emacs 制御文字を使って編集することができます。そのうち最も重要なものとして、以下のようなキーがあります: C-A (Control-A) はカーソルを行の先頭へ移動させます。C-E は末尾へ移動させます。C-B は逆方向へ一つ移動させます。C-F は順方向へ移動させます。Backspace は逆方向に向かって文字を消します。C-D は順方向に向かって消します。C-K は順方向に向かって行の残りを kill し (消し) ます、C-Y は最後に kill された文字列を再び yank し (取り出し) ます。C-underscore 最後の変更を元に戻します; これは、繰り返してどんどんさかのぼることができます。

### A.2 ヒストリ置換

ヒストリ置換は次のように働きます。入力された行のうち、空行でない実行された行はすべてヒストリバッファに保存されます。そして、プロンプトが呈示されるときには、ヒストリバッファの最も下の新たな行に移動します。C-P はヒストリバッファの中を一行だけ上に移動し (戻し) ます。C-N は 1 行だけ下に移動します。ヒストリバッファのどの行も編集することができます。行が編集されると、それを示すためにプロンプトの前にアスタリスクが表示されます<sup>1</sup>。Return キーを押すと現在行がインタプリタへ渡されます。C-R はインクリメンタルな逆方向サーチ (reverse search) を開始し、C-S は順方向サーチ (forward search) を開始します。

<sup>1</sup> 訳注: これはデフォルト設定の Readline では現れません。set mark-modified-lines on という行を `~/inputrc` または環境変数 INPUTRC が指定するファイルに置くことによって現れるようになります。

## A.3 キー割り当て

Readline ライブラリのキー割り当て (key binding) やその他のパラメタは、`~/inputrc` という初期化ファイル<sup>2</sup>にコマンドを置くことでカスタマイズできます。キー割り当ての形式は

```
key-name: function-name
```

または

```
"string": function-name
```

で、オプションの設定方法は

```
set option-name value
```

です。例えば、以下のように設定します:

```
# vi スタイルの編集を選択する:
set editing-mode vi

# 一行だけを使って編集する:
set horizontal-scroll-mode On

# いくつかのキーを再束縛する:
Meta-h: backward-kill-word
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

Python では、Tab に対するデフォルトの割り当ては TAB の挿入です。Readline のデフォルトであるファイル名補完関数ではないので注意してください。もし、どうしても Readline のデフォルトを割り当てたいのなら、`~/inputrc` に

```
Tab: complete
```

を入れれば設定を上書きすることができます。(もちろん、Tab を使って補完を行うのに慣れている場合、この設定を行うとインデントされた継続行を入力しにくくなります。)

変数名とモジュール名の自動的な補完がオプションとして利用できます。補完をインタプリタの対話モードで有効にするには、以下の設定をスタートアップファイルに追加します:<sup>3</sup>

```
import rlcompleter, readline
readline.parse_and_bind('tab: complete')
```

この設定は、Tab キーを補完関数に束縛します。従って、Tab キーを二回たたくと補完候補が示されます; 補完機能は Python の文の名前、現在のローカル変数、および利用可能なモジュール名を検索します。string.a のようなドットで区切られた式については、最後の `'.'` までの式を評価し、結果として得られたオブジェクトの属性から補完候補を示します。\_\_getattr\_\_() メソッドを持ったオブジェクトが式に含まれている場合、\_\_getattr\_\_() がアプリケーション定義のコードを実行するかもしれないので注意

<sup>2</sup> 訳注: このファイル名は環境変数 INPUTRC がもしあればその指定が優先されます。

<sup>3</sup> Python は、対話インタプリタを開始する時に PYTHONSTARTUP 環境変数が指定するファイルの内容を実行します。

してください。

より良くできたスタートアップファイルは以下例のようになります。この例では、作成した名前が不要になると削除されるので気をつけてください; これは、スタートアップファイルが対話コマンドと同じ名前空間で実行されているので、不要な名前を除去して対話環境に副作用を生まないようにするためです。importされたモジュールのうち、osのようなインタプリタのほとんどのセッションで必要なものについては、残しておくくと便利に思うかもしれません。

```
# Add auto-completion and a stored history file of commands to your Python
# interactive interpreter. Requires Python 2.0+, readline. Autocomplete is
# bound to the Esc key by default (you can change it - see readline docs).
#
# Store the file in ~/.pystartup, and set an environment variable to point
# to it, e.g. "export PYTHONSTARTUP=/max/home/itamar/.pystartup" in bash.
#
# Note that PYTHONSTARTUP does *not* expand "~", so you have to put in the
# full path to your home directory.

import atexit
import os
import readline
import rlcompleter

historyPath = os.path.expanduser("~/pyhistory")

def save_history(historyPath=historyPath):
    import readline
    readline.write_history_file(historyPath)

if os.path.exists(historyPath):
    readline.read_history_file(historyPath)

atexit.register(save_history)
del os, atexit, readline, rlcompleter, save_history, historyPath
```

## A.4 解説

この機能は、初期の版のインタプリタに比べれば大きな進歩です; とはいえ、まだいくつかの要望が残されています: 例えば、行を継続するときに正しいインデントが呈示されたら快適でしょう (パーサは次の行でインデントトークンが必要かどうかを知っています)。補完機構がインタプリタのシンボルテーブルを使ってもよいかもしれません。カッコやクォートなどの対応をチェックする (あるいは指示する) コマンドも有用でしょう。





## 浮動小数点演算、その問題と制限

浮動小数点数は、計算機ハードウェアの中では、基数を 2 とする (2 進法の) 分数として表現されています。例えば、小数

0.125

は、 $1/10 + 2/100 + 5/1000$  という値を持ちますが、これと同様に、2 進法の分数

0.001

は  $0/2 + 0/4 + 1/8$  という値になります。これら二つの分数は同じ値を持っていますが、ただ一つ、最初の分数は基数 10 で記述されており、二番目の分数は基数 2 で記述されていることが違います。

残念なことに、ほとんどの小数は 2 進法の分数として正確に表わすことができません。その結果、一般に、入力した 10 進の浮動小数点数は、2 進法の浮動小数点数で近似された後、実際にマシンに記憶されます。

最初は基数 10 を使うと問題を簡単に理解できます。分数  $1/3$  を考えてみましょう。分数  $1/3$  は、基数 10 の分数として、以下のように近似することができます:

0.3

さらに正確な近似は、

0.33

です。さらに正確に近似すると、

0.333

となり、以後同様です。何個桁数を増やして書こうが、結果は決して厳密な  $1/3$  にはなりません。しかし、少しずつ正確な近似にはなっていくでしょう。

同様に、基数を 2 とした表現で何桁使おうとも、10 進数の 0.1 は基数を 2 とした分数で正確に表現することはできません。基数 2 では、 $1/10$  は循環小数 (repeating fraction)

0.0001100110011001100110011001100110011001100110011001100110011...

となります。どこか有限の桁で止めると、近似値を得ることになります。これこそが、以下のような事態:

```
>>> 0.1
0.100000000000000001
```

に出くわす理由です。

今日では、ほとんどのマシンでは、0.1 を Python のプロンプトから入力すると上のような結果を目にします。そうならないかもしれませんが、これはハードウェアが浮動小数点数を記憶するのに用いているビット数がマシンによって異なり、Python は単にマシンに 2 進で記憶されている、真の 10 進の値を近似した値を、それに 10 進で近似して出力するだけだからです。ほとんどのマシンでは、Python が 0.1 を記憶するために 2 進近似した真の値を 10 進で表すと、以下のような出力

```
>>> 0.1
0.10000000000000000055511151231257827021181583404541015625
```

になるでしょう！ Python プロンプトは、文字列表現を得るために何に対しても `repr()` を (非明示的に) 使います。浮動小数点数の場合、`repr(float)` は真の 10 進値を有効数字 17 桁で丸め、以下のような表示

```
0.100000000000000001
```

を行います。

`repr(float)` が有効数字 17 桁の値を生成するのは、この値が (ほとんどのマシン上で)、全ての有限の浮動小数点数  $x$  について `eval(repr(x)) == x` が成り立つのに十分で、かつ有効数字 16 桁に丸めると成り立たないからです。

これは 2 進法の浮動小数点の性質です: Python のバグでも、ソースコードのバグでもなく、浮動小数点演算を扱えるハードウェア上の、すべての言語で同じ類の現象が発生します (ただし、言語によっては、デフォルトのモードや全ての出力モードでその差を表示しない かもしれません)。

Python の組み込みの `str()` 関数は有効数字 12 桁しか生成しません。このため、この関数を代わりに使いたいと思うかもしれません。この関数は `eval(str(x))` としたときに  $x$  を再現しないことが多いですが、出力を目で見るには好ましいかもしれません:

```
>>> print str(0.1)
0.1
```

現実という考えからは、上の表示は錯覚であると気づくのは重要なことです: マシン内の値は厳密に 1/10 ではなく、単に真のマシン内の表示される値を丸めているだけなのです。

まだ驚くべきことがあります。例えば、以下

```
>>> 0.1
0.100000000000000001
```

を見て、`round()` 関数を使って桁を切り捨て、期待する 1 桁にしたい誘惑にかられたとします。しかし、結果は依然同じ値です:

```
>>> round(0.1, 1)
0.100000000000000001
```

問題は、"0.1" を表すために記憶されている 2 進表現の浮動小数点数の値は、すでに 1/10 に対する最良

の近似になっており、値を再度丸めようとしてもこれ以上ましにはならないということです: すでに値は、`round()` で得られる値になっているというわけです。

もう一つの重要なことは、0.1 が正確に 1/10 ではないため、0.1 どうしを 10 回加算すると厳密に 1.0 にはならないこともある、ということです:

```
>>> sum = 0.0
>>> for i in range(10):
...     sum += 0.1
...
>>> sum
0.9999999999999999
```

2 進の浮動小数点数に対する算術演算は、このような意外性をたくさん持っています。"0.1" に関する問題は、以下の "表現エラー" の章で詳細に説明します。2 進法の浮動小数点演算にともなうその他のよく知られた意外な事象に関しては *The Perils of Floating Point* を参照してください。

究極的にいうと、“容易な答えはありません”。ですが、浮動小数点数のことを過度に警戒しないでください! Python の float 型操作におけるエラーは浮動小数点処理ハードウェアから受けついたものであり、ほとんどのマシン上では一つの演算あたり高々  $2^{53}$  分の 1 です。この誤差はほとんどの作業で相当以上のものですが、浮動小数点演算は 10 進の演算ではなく、浮動小数点の演算を新たに行うと、新たな丸め誤差の影響を受けることを心にとどめておいてください。

異常なケースが存在する一方で、普段の浮動小数点演算の利用では、単に最終的な結果の値を必要な 10 進の桁数に丸めて表示するのなら、最終的には期待通りの結果を得ることになるでしょう。こうした操作は普通 `str()` で事足りますし、よりきめ細かな制御をしたければ、Python の % 書式化演算子についての議論を参照してください: %g、%f、および %e といった書式化コードでは、浮動小数点数を表示用に丸めるための柔軟性のある、簡単な手段を提供しています。

## B.1 表現エラー

この章では、“0.1” の例について詳細に説明し、このようなケースに対してどのようにすれば正確な分析を自分で行えるかを示します。ここでは、2 進法表現の浮動小数点数についての基礎的な知識があるものとして話を進めます。

表現エラー は、いくつかの (実際にはほとんどの) 10 進の小数が 2 進法 (基数 2) の分数として表現できないことに関係しています。これは Python (あるいは Perl、C、C++、Java、Fortran、およびその他多く) が期待通りの正確な 10 進数を表示できない主要な理由です:

```
>>> 0.1
0.10000000000000001
```

なぜこうなるのでしょうか? 1/10 は 2 進法の分数で厳密に表現することができません。今日 (2000 年 11 月) のマシンは、ほとんどすべて IEEE-754 浮動小数点演算を使用しており、ほとんどすべてのプラットフォームでは Python の浮動小数点を IEEE-754 における "倍精度 (double precision)" に対応付けます。754 の double には 53 ビットの精度を持つ数が入るので、計算機に入力を行おうとすると、可能な限り 0.1 を最も近い値の分数に変換し、 $J/2^{**N}$  の形式にしようと努力します。J はちょうど 53 ビットの精度の整数です。

$$1 / 10 \sim J / (2^{**N})$$

を書き直すと、

```
J ~= 2**N / 10
```

となります。

$J$  は厳密に 53 ビットの精度を持っている ( $\geq 2^{52}$  だが  $< 2^{53}$ ) を思い出すと、 $N$  として最適な値は 56 になります:

```
>>> 2L**52
4503599627370496L
>>> 2L**53
9007199254740992L
>>> 2L**56/10
7205759403792793L
```

すなわち、56 は  $J$  をちょうど 53 ビットの精度のままに保つ  $N$  の唯一の値です。 $J$  の取りえる値はその商を丸めたものです:

```
>>> q, r = divmod(2L**56, 10)
>>> r
6L
```

残りは 10 の半分以上なので、最良の近似は丸め値を一つ増やした (round up) ものになります:

```
>>> q+1
7205759403792794L
```

従って、754 倍精度における  $1/10$  の取りえる最良の近似は  $2^{56}$  以上の値、もしくは

```
7205759403792794 / 72057594037927936
```

となります。丸め値を 1 増やしたので、この値は実際には  $1/10$  より少し小さいことに注意してください; 丸め値を 1 増やさない場合、商は  $1/10$  よりもわずかに小さくなります。しかし、どちらにしろ厳密に  $1/10$  ではありません!

つまり、計算機は  $1/10$  を“理解する”ことは決してありません: 計算機が理解できるのは、上記のような厳密な分数であり、754 の倍精度浮動小数点数で得られるもっともよい近似は:

```
>>> .1 * 2L**56
7205759403792794.0
```

となります。

この分数に  $10^{30}$  を掛ければ、有効数字 30 桁の十進数の (切り詰められた) 値を見ることができます:

```
>>> 7205759403792794L * 10L**30 / 2L**56
10000000000000000005551115123125L
```

これは、計算機に記憶されている正確な数値が、10 進数値 0.100000000000000005551115123125 とほぼ等しいことを意味します。有効数字 17 桁に丸めることで、Python が表示する 0.10000000000000001 にな

ります (そう、得られるかぎり最良の入力あるいは出力値の変換を C ライブラリで行うような、754 に適合するプラットフォームにおいてです — 読者の計算機ではそうではないかもしれません！)



# 歴史とライセンス

## C.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see <http://www.cwi.nl/>) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see <http://www.cnri.reston.va.us/>) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see <http://www.zope.com/>). In 2001, the Python Software Foundation (PSF, see <http://www.python.org/psf/>) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see <http://www.opensource.org/> for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.2	2.1.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2.1	2.2	2002	PSF	yes
2.2.2	2.2.1	2002	PSF	yes
2.2.3	2.2.2	2002-2003	PSF	yes
2.3	2.2.2	2002-2003	PSF	yes
2.3.1	2.3	2002-2003	PSF	yes
2.3.3	2.3.2	2004	PSF	yes
2.3.4	2.3.3	2004	PSF	yes
2.3.5	2.3.4	2005	PSF	yes

注意: GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

## C.2 Terms and conditions for accessing or otherwise using Python

### **PSF LICENSE AGREEMENT FOR PYTHON 2.3.4**

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.3.4 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.3.4 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2003 Python Software Foundation; All Rights Reserved" are retained in Python 2.3.4 alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.3.4 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.3.4.
4. PSF is making Python 2.3.4 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.3.4 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.3.4 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.3.4, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 2.3.4, Licensee agrees to be bound by the terms and conditions of this License Agreement.

### **BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0**

#### **BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1**

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").



2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an “AS IS” basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the “BeOpen Python” logos available at <http://www.pythonlabs.com/logos.html> may be used according to the permissions granted on that web page.
7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

#### **CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1**

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 (“CNRI”), and the Individual or Organization (“Licensee”) accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI’s License Agreement and CNRI’s notice of copyright, i.e., “Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved” are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI’s License Agreement, Licensee may substitute the following text (omitting the quotes): “Python 1.6.1 is made available subject to the terms and conditions in CNRI’s License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: <http://hdl.handle.net/1895.22/1013>.”
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an “AS IS” basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia’s conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By clicking on the “ACCEPT” button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

**CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2**

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

# 日本語訳について

## D.1 このドキュメントについて

この文書は、Python ドキュメント翻訳プロジェクトによる Python Tutorial の日本語訳版です。日本語訳に対する質問や提案などがありましたら、Python ドキュメント翻訳プロジェクトのメーリングリスト

<http://www.python.jp/mailman/listinfo/python-doc-jp>

または、プロジェクトのバグ管理ページ

[http://sourceforge.jp/tracker/?atid=116\&group\\_id=11\&func=browse](http://sourceforge.jp/tracker/?atid=116\&group_id=11\&func=browse)

までご報告ください。

## D.2 翻訳者一覧 (敬称略)

1.3 和訳: SUZUKI Hisao <suzuki at acm.org> (August 27, 1995)

1.4 和訳: SUZUKI Hisao <suzuki at acm.org> (April 20, 1997)

1.5.1 和訳: SUZUKI Hisao <suzuki at acm.org> (August 23, 1998)

2.1 和訳: SUZUKI Hisao <suzuki at acm.org> (June 10, 2001)

2.2.3 差分和訳: sakito <sakito at s2.xrea.com> (August 10, 2003)

2.2.3 敬体訳: ymasuda <y.masuda at acm.org> (September 5, 2003)

2.3 差分和訳: sakito <sakito at s2.xrea.com> (August 30, 2003)

2.3 敬体訳: ymasuda <y.masuda at acm.org> (November 30, 2003)

2.3.3 差分訳: ymasuda <y.masuda at acm.org> (Devember 25, 2003)



## 用語集

>>> 典型的な対話シェルのプロンプトです。インタプリタ上ですぐ実行できるようなプログラムコード例の中によく書かれています。

... 典型的な対話シェルのプロンプトです。インデントされたコードブロック内のコードの入力を促す際に出力されます。

`__future__` 互換性のない新たな機能を現在のインタプリタで有効にするためにプログラマが利用できる擬似モジュールです。例えば、式  $11/4$  は現状では 2 になります。この式を実行しているモジュールで

```
from __future__ import division
```

を行って 真の除算操作 (*true division*) を有効にすると、式  $11/4$  は 2.75 になります。実際に `__future__` モジュールを import してその変数を評価すれば、新たな機能が初めて追加されたのがいつで、いつデフォルトの機能になる予定かわかります：

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

`__slots__` 新形式のクラス内で、インスタンス属性の記憶に必要な領域をあらかじめ定義しておき、それとひきかえにインスタンス辞書を排除してメモリの節約を行うための宣言です。これはよく使われるテクニックですが、正しく動作させるには少々手際を要するので、例えばメモリが死活問題となるようなアプリケーション内にインスタンスが大量に存在するといった稀なケースを除き、使わないのがベストです。

**BDFL** 慈悲ぶかき独裁者 (Benevolent Dictator For Life) の略です。Python の作者、Guido van Rossum のことです。

**EAFP** 「認可をとるより許しを請う方が容易 (easier to ask for forgiveness than permission、マーフィーの法則)」の略です。Python で広く使われているコーディングスタイルでは、通常は有効なキーや属性が存在するものと仮定し、その仮定が誤っていた場合に例外を捕捉します。この簡潔で手早く書けるコーディングスタイルには、`try` 文および `except` 文がたくさんあるのが特徴です。このテクニックは、C のような言語でよく使われている *LBYL* スタイルと対照的なものです。

**GIL** グローバルインタプリタロック (*global interpreter lock*) を参照してください。

**IDLE** Python の組み込み開発環境 (Integrated DeveLopment Environment) です。IDLE は Python の標準的な配布物についてくる基本的な機能のエディタとインタプリタ環境です。初心者に向いている点として、IDLE はよく洗練され、複数プラットフォームで動作する GUI アプリケーションを実装したい人むけの明解なコード例にもなっています。

**LBYL** 「ころばぬ先の杖」(look before you leap) の略です。このコーディングスタイルでは、呼び出しや検索を行う前に、明示的に前提条件 (pre-condition) 判定を行います。*EAFP* アプローチと対照的で、`if` 文がたくさん使われるのが特徴的です。

**Python3000** テレパシーインタフェースを持ち、後方互換性をもたなくてもよいとされている、架空の Python リリースのことです。

**Python の悟り (Zen of Python)** Python を理解し利用する上での導きとなる、Python の設計原則と哲学をリストにしたものです。対話プロンプトで “`import this`” とすると、リストを読めます。

**イテレータ (iterator)** 一連のデータ列 (stream) を表現するオブジェクトです。イテレータの `next()` メソッドを繰り返し呼び出すと、データ列中の要素を一つずつ返します。後続のデータがなくなると、データの代わりに `StopIteration` 例外を送出します。現時点では、イテレータオブジェクトが全てのオブジェクトを出し尽くすと、それ以降は `next()` を何度呼んでも `StopIteration` を送じます。イテレータは、イテレータオブジェクト自体を返す `__iter__()` メソッドを実装しなければならず、そのため全てのイテレータは他の反復可能オブジェクトを受け取れるほとんどの場所で反復可能で利用できます。著しい例外は複数の反復を行うようなコードです。(list のような) コンテナオブジェクトでは、`iter()` 関数にオブジェクトを渡したり、`for` ループ内で使うたびに、新たな未使用のイテレータを生成します。このイテレータをさらに別の場所でイテレータとして使おうとすると、二段階目のイテレータは使用済みの同じイテレータオブジェクトを返すため、空のコンテナのように見えます。

**インタプリタ形式の (interpreted) Python** はインタプリタ形式の言語であり、コンパイラ言語の対極に位置します。ここでのインタプリタ言語とは、ソースコードのファイルを、まず実行可能形式にしてから実行させるといった操作なしに、直接実行できることを意味します。インタプリタ形式の言語は通常、コンパイラ形式の言語よりも開発 / デバッグのサイクルを持ちます。対話的 (*interactive*) も参照してください。

**型強制 (coercion)** データをある型から別の型に変換することです。例えば、`int(3.15)` は浮動小数点数を整数に型強制し、3 にします。ほとんどの数学的演算には、引数を共通の型に型強制するような規則があります。例えば、加算 `3+4.5` は、整数 3 を 4.5 に加算する前に浮動小数点数に型強制し、その結果は浮動小数点数 7.5 となります。

**旧形式クラス (classic class) object** を継承していないクラス全てを指します。新形式クラス (*new-style class*) も参照してください。

**グローバルインタプリタロック (global interpreter lock)** Python の実行スレッド間で使われているロックで、一度に一つのスレッドだけが動作するよう保証しています。このロックによって、同時に同じメモリにアクセスする二つのプロセスは存在しないと保証されているので、Python を単純な構造にできるのです。インタプリタ全体にロックをかけると、多重プロセッサ計算機における並列性の恩恵と引き換えにインタプリタの多重スレッド化を簡単に行えます。かつて “スレッド自由な (free-threaded)” インタプリタを作ろうと努力したことがありましたが、広く使われている単一プロセッサの場合にはパフォーマンスが低下するという事態に悩まされました。

**ジェネレータ (generator)** イテレータを返す関数です。`return` の代わりに `yield` が使われている他は、通常関数と同じに見えます。ジェネレータ関数は一つまたはそれ以上の `for` ループや `while` ループを含んでおり、ループの呼び出し側に要素を返す (`yield`) ようになっています。ジェネレータが返すイテレータを使って関数を実行すると、関数は `yield` キーワードで (値を返して) 一旦停止し、`next()` を呼んで次の要素を要求するたびに実行を再開します。

辞書 (dictionary) 任意のキーを値に対応付ける連想配列です。dict の使い方は list に似ていますが、ゼロから始まる整数ではなく、`__hash__()` 関数を実装している全てのオブジェクトをキーにできます。Perl ではハッシュ (hash) と呼ばれています。

新形式クラス (new-style class) object から継承したクラス全てを指します。これには list や dict のような全ての組み込み型が含まれます。`__slots__`、デスクリプタ、プロパティ、`__getattr__()`、クラスメソッド、静的メソッドといった、Python の新しい精緻な機能を使えるのは新形式のクラスだけです。

整数除算 (integer division) 剰余を考慮しない数学的除算です。例えば、式  $11/4$  は現状では 2 になりますが、浮動小数点数の除算では 2.75 を返します。切り捨て除算 (floor division) とも呼ばれます。二つの整数間で除算を行うと、結果は (端数切捨て関数が適用されて) 常に整数になります。しかし、被演算子の一方が (float のような) 別の数値型の場合、演算の結果は共通の型に型強制されます (型強制 (coercion) 参照)。例えば、浮動小数点数で整数を除算すると結果は浮動小数点になり、場合によっては端数部分を伴います。// 演算子を / の代わりに使うと、整数除算を強制できます。`__future__` も参照してください。

対話的 (interactive) Python には対話的インタプリタがあり、何かを試してその結果を直接見られます。python を引数なしで起動 (場合によってはコンピュータのメインメニューから選んで起動) してください。対話的インタプリタは新しいアイデアを試したり、モジュールやパッケージの中を調べてみたりする (`help(x)` を思い出してください) ための強力な方法です。

デスクリプタ (descriptor) メソッド `__get__()`、`__set__()`、あるいは `__delete__()` が定義されている新形式のオブジェクトです。あるクラス属性がデスクリプタである場合、その属性を検索すると、そのデスクリプタ固有に束縛されている動作を呼び出します。通常、 $a.b$  と書くと、 $a$  のクラス辞書内でオブジェクト  $b$  を検索しますが、 $b$  がデスクリプタの場合には、デスクリプタで定義されたメソッドを呼び出します。デスクリプタの理解は、Python を深く理解する上で鍵となります。というのは、デスクリプタこそが、関数、メソッド、プロパティ、クラスメソッド、静的メソッド、そしてスーパクラスの参照といった多くの機能の基盤だからです。

名前空間 (namespace) 変数を記憶している場所です。名前空間は辞書を用いて実装しています。名前空間には、ローカル、グローバル、組み込み名前空間、そして (メソッド内の) オブジェクトのネストされた名前空間があります。例えば、関数 `__builtin__.open()` および `os.open()` は名前空間で区別します。名前空間はまた、ある関数をどのモジュールが実装しているかをはっきりさせることで、可読性やメンテナンス性を与えます。例えば、`random.seed()`、あるいは `itertools.izip()` と書くことで、これらの関数がそれぞれ random モジュールや itertools モジュールで実装されていることがはっきりします。

ネストされたスコープ (nested scope) ある文が何らかの定義に囲われているとき、定義の外で使われている変数をその文から参照できる機能です。例えば、ある関数が別の関数の中で定義されている場合、内側の関数は外側の関数中の変数を参照できます。ネストされたスコープは変数の参照だけができ、変数の代入はできないので注意してください。変数の代入は、常に最も内側のスコープにある変数に対する書き込みになります。対照的に、ローカル変数を使うと、常に最も内側のスコープで値を読み書きします。同様に、グローバル変数を使うと、グローバル名前空間の値を読み書きします。

バイトコード (byte code) インタプリタ中の Python プログラムを表す内部表現です。バイトコードはまた、.pyc や .pyo ファイルにキャッシュされ、同じファイルを二度目に実行した際により高速に実行できるようにします (ソースコードはバイトコードにコンパイルされて保存されます)。このバイトコードは、各々のバイトコードに対応するサブルーチンを呼び出すような “仮想計算機 (virtual machine)” で動作する “中間言語 (intermediate language)” といえます。



**配列 (sequence)** 反復可能なオブジェクト (*iterable*) 反復可能なオブジェクト (*iterable*) は、特殊なメソッド `__getitem__()` および `__len__()` を介して整数インデックスを使った効率的な要素アクセスをサポートします。組み込み配列型には、`list`、`str`、`tuple`、および `unicode` があります。`dict` は `__getitem__()` と `__len__()` もサポートしますが、検索の際に任意の変更不能 (*immutable*) なキーを使うため、配列というよりもむしろマップ (*mapping*) とみなされているので注意してください。

**反復可能オブジェクト (iterable)** コンテナオブジェクトで、コンテナ内のメンバを一つずつ返せるようになっているものです。反復可能オブジェクトの例には、(`list`、`str`、および `tuple` といった) 全ての配列型や、`dict` や `file` といった非配列型、あるいは `__iter__()` や `__getitem__()` メソッドを実装したクラスのインスタンスが含まれます。反復可能オブジェクトは `for` ループ内やその他多くの配列が必要となる状況 (`zip()`、`map()`、...) で利用できます。反復可能オブジェクトを組み込み関数 `iter()` の引数として渡すと、オブジェクトに対するイテレータを返します。このイテレータは一連の値を引き渡す際に便利です。反復可能オブジェクトを使う際には、通常 `iter()` を呼んだり、イテレータオブジェクトを自分で扱う必要はありません。`for` 文ではこの操作を自動的に行い、無名の変数を作成して、ループの間イテレータを記憶します。イテレータ (*iterator*)、配列 (*sequence*)、およびジェネレータ (*generator*) も参照してください。

**変更可能なオブジェクト (mutable)** 変更可能なオブジェクトは、`id()` を変えることなく値を変更できます。変更不能 (*immutable*) も参照してください。

**変更不能なオブジェクト (immutable)** 固定の値を持ったオブジェクトです。変更不能なオブジェクトには、数値、文字列、およびタプル (そしてその他) があります。これらのオブジェクトは値を変えられません。別の値を記憶させる際には、新たなオブジェクトを作成しなければなりません。変更不能なオブジェクトは、固定値のハッシュ値が必要となる状況で重要な役割を果たします。辞書におけるキーがその例です。

**マップ (mapping)** 特殊メソッド `__getitem__()` を使って、任意のキーに対する検索をサポートする (`dict` のような) コンテナオブジェクトです。

**メタクラス (metaclass)** 何らかのクラスを生成するクラスです。クラス定義を行うと、クラス名、クラス辞書、そして基底クラスからなるリストが生成されます。メタクラスはこれらの三つを引数として取り、メタクラスを生成する働きをします。ほとんどのオブジェクト指向プログラム言語には、メタクラスに対するデフォルトの実装があります。Python の特徴的な点は、自作のメタクラスを作成できることです。ほとんどのユーザにとって、このツールはまったく必要のないものですが、必要さえあれば、メタクラスは強力でエレガントな問題解決手段になります。メタクラスは、属性へのアクセスをログに記録したり、システムをスレッド安全にしたり、オブジェクトの生成の追跡したり、単集合を実装したり、そしてその他多くのタスクで使われてきています。

**リストの内包表記 (list comprehension)** 配列内の全ての要素、あるいはサブセットを処理してその結果からなるリストを返させるための、コンパクトなやりかたです。`result = ["0x%02x" % x for x in range(256) if x % 2 == 0]` とすると、0 から 255 までの偶数を 16 進数表記 (0x..) した文字列からなるリストを生成します。`if` 節はオプションです。`if` 節がない場合、上のケースでは `range(256)` の全ての要素が処理されます。



# 索引

## Symbols

..., [101](#)  
»>, [101](#)  
\_\_builtin\_\_ (組み込みモジュール), [43](#)  
\_\_future\_\_, [101](#)  
\_\_slots\_\_, [101](#)

## A

append() (list のメソッド), [29](#)

## B

バイトコード, [103](#)  
BDFL, [101](#)  
byte code, [103](#)

## C

classic class, [102](#)  
coercion, [102](#)  
compileall (標準モジュール), [42](#)  
count() (list のメソッド), [29](#)

## D

descriptor, [103](#)  
デスクリプタ, [103](#)  
dictionary, [102](#)  
docstrings, [21](#), [26](#)  
documentation strings, [21](#), [26](#)

## E

EAFP, [101](#)  
environment variables  
    INPUTRC, [85](#), [86](#)  
    PATH, [5](#), [41](#)  
    PYTHONPATH, [41](#), [42](#)  
    PYTHONSTARTUP, [6](#), [86](#)  
extend() (list のメソッド), [29](#)

## F

file  
    object, [52](#)  
    オブジェクト, [52](#)  
for  
    実行文, [19](#)  
    statement, [19](#)

## G

generator, [102](#)  
GIL, [101](#)  
global interpreter lock, [102](#)  
グローバルインタプリタロック, [102](#)

## H

配列, [103](#)  
反復可能, [104](#)  
変更不能, [104](#)  
変更可能, [104](#)

## I

IDLE, [101](#)  
immutable, [104](#)  
index() (list のメソッド), [29](#)  
INPUTRC, [85](#), [86](#)  
insert() (list のメソッド), [29](#)  
インタプリタ形式の, [102](#)  
integer division, [103](#)  
interactive, [103](#)  
interpreted, [102](#)  
iterable, [104](#)  
iterator, [102](#)  
イテレータ, [102](#)

## J

ジェネレータ, [102](#)  
実行文

`for`, [19](#)  
辞書, [102](#)

## K

型強制, [102](#)  
旧形式クラス, [102](#)

## L

LBYL, [101](#)  
list comprehension, [104](#)  
リストの内包表記, [104](#)

## M

マップ, [104](#)  
mapping, [104](#)  
metaclass, [104](#)  
メタクラス, [104](#)  
method  
    object, [67](#)  
    オブジェクト, [67](#)  
module  
    search path, [41](#)  
mutable, [104](#)

## N

名前空間, [103](#)  
namespace, [103](#)  
nested scope, [103](#)  
ネストされたスコープ, [103](#)  
new-style class, [103](#)

## O

object  
    file, [52](#)  
    method, [67](#)  
オブジェクト  
    file, [52](#)  
    method, [67](#)  
`open()` (組み込み関数), [52](#)

## P

PATH, [5](#), [41](#)  
path  
    module search, [41](#)  
pickle (標準モジュール), [54](#)  
`pop()` (list のメソッド), [29](#)  
Python3000, [102](#)  
Python の悟り, [102](#)

PYTHONPATH, [41](#), [42](#)  
PYTHONSTARTUP, [6](#), [86](#)

## R

`readline` (組み込みモジュール), [86](#)  
`remove()` (list のメソッド), [29](#)  
`reverse()` (list のメソッド), [29](#)  
`rlcompleter` (標準モジュール), [86](#)

## S

search  
    path, module, [41](#)  
整数除算, [103](#)  
sequence, [103](#)  
新形式クラス, [103](#)  
`sort()` (list のメソッド), [29](#)  
statement  
    for, [19](#)  
`string` (標準モジュール), [49](#)  
strings, documentation, [21](#), [26](#)  
`sys` (標準モジュール), [42](#)

## T

対話的, [103](#)

## U

`unicode()` (組み込み関数), [15](#)

## Z

Zen of Python, [102](#)