Python Tutorial 和訳

Release 2.1

Guido van Rossum
Fred L. Drake, Jr., editor

April 15, 2001

PythonLabs
E-mail: python-docs@python.org

[0]和訳: SUZUKI Hisao <suzuki@acm.org> June 10, 2001

BEOPEN.COM TERMS AND CONDITIONS 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 OPEN SOURCE GPL-COMPATIBLE LICENSE AGREEMENT

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.

CWI PERMISSIONS STATEMENT AND DISCLAIMER

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.

Abstract

Python (パイソン) は学びやすく強力なプログラミング言語である。Python は効率的な高水準のデータ構造と，オブジェクト指向プログラミングへの単純だが効果的なアプローチをそなえている。Python のエレガントな構文と動的な型付けは，そのインタープリタ的な性質とともに，Python を大多数のプラットフォーム上の多くの分野でスクリプティングとラピッド・アプリケーション開発のための理想的な言語にしている。

Python インタープリタとその広範な標準ライブラリは，すべての主要なプラットフォームに対し，Python web サイト http://www.python.org からソースまたはバイナリ形式でフリーに入手可能であり，そしてフリーに配布可能である。このサイトにはまた，多くのフリーなサード・パーティの Python モジュール，プログラムおよびツールと付加的な文書類のディストリビューションおよびポインタがある。

Python インタープリタは，C または C++ (または C から呼出し可能な他の言語) で実装された新しい関数とデータ型で容易に拡張される。Python はカスタマイズ可能なアプリケーションのための拡張言語としても適している。

このチュートリアルは読者に Python の言語とシステムの基本的な概念と特徴を非形式的に紹介する。Python インタープリタを手もとにおいて実際に試してみることは助けになるが，すべての例題は自己完結的だから，オフラインでこのチュートリアルを読むこともできる。

標準のオブジェクトとモジュールの記述については *Python Library Reference* ドキュメントを見られたい。*Python Reference Manual* はより形式的な言語の定義を与えている。C または C++ で拡張を書くには *Extending and Embedding the Python Interpreter* と *Python/C API Reference* の各マニュアルを読まれたい。Python を詳細にカバーしている本も何冊かある。

このチュートリアルは包括的であろうとも，あらゆる個々の機能をカバーしようともしていない。それどころか普通に使われる各機能をカバーしようとすらしていない。そのかわり，これは Python の最も注目に値する機能の多くを紹介し，言語の香りとスタイルについての十分な理解を君に与えるだろう。これを読み終えた後，君は Python のモジュールとプログラムを読んだり書いたりできるようになっているだろう。そして *Python Library Reference* に記述されたさまざまな Python ライブラリ・モジュールについてさらに学ぶ準備ができているだろう。

CONTENTS

CHAPTER

ONE

君をその気にさせるために

もしも大規模なシェル・スクリプトを書いたことがあるならば，多分こんな感覚を君は知っているだろう。もう一つ機能を加えたいのに，もう既にとても遅く，とても大きく，とても複雑になってしまっている。あるいは，C からしかアクセスできないシステム・コールか何かの関数をその機能が必要としている…。たいてい手もとにあるその問題は，可変長の文字列か (ファイル名のソート済みリストのような) 何かのデータ型を必要としているため，シェルでは簡単なのに C での実装は大仕事になったり，あるいは君が C にあまり慣れていなかったりで，スクリプトを C で書き直していたのでは深刻すぎて引き合わない。

もう一つの場合として，あるいは君はいくつかの C ライブラリを使って作業しなくてはならないが，通常の C の書込み/コンパイル/テスト/再コンパイルのサイクルでは時間がかかりすぎるかもしれない。ソフトウェアをもっと素早く開発する必要があるというわけだ。あるいはことによると君は拡張言語を使うことのできるプログラムを書いているが，一つの言語を設計し，そのインタープリタを書いてデバッグし，そしてそれを君のアプリケーションにくくりつける，ということまでは，やりたくないかもしれない。

そんな場合，Python が君にふさわしい言語かもしれない。Python は簡単に使えるが，本物のプログラミング言語 (real programming language) であり，大規模プログラムのためにシェルよりもずっと多くの構造とサポートを提供している。他方，Python はまた C よりもずっと多くのエラー検査を提供し，そして，**超高水準言語** *(very-high-level language)* として，C で効率良く実装しようとすると何日もかかってしまうフレキシブルな配列や辞書などの高水準データ型を組込みにしている。その一般性の高いデータ型のおかげで Python は *Awk* あるいは *Perl* と比べてさえずっと広い問題領域に適用可能だが，それでもなお多くのことがそれらの言語と少なくとも同じくらいに簡単だ。

Python はプログラムを，別の Python プログラムで再利用できるように，モジュールに分割することを可能にしている。Python には大規模な標準モジュールのコレクションが付いて来る。君はこれを自分のプログラムの基礎として — または Python でのプログラミングの学習を始めるための例題として — 利用することができる。ファイル I/O, システム・コール，ソケット，さらには Tk のような GUI ツールキットへのインタフェース等を用意する組込みモジュールもある。

Python はインタープリタによる言語である。コンパイルもリンクも不要だから，君はかなりの時間をプログラム開発のあいだ節約できる。インタープリタは対話的に使用できるから，たやすく言語の機能を試したり，使い捨てプログラムを書いたり，ボトムアップにプログラムを開発している途中で関数をテストしたりできる。インタープリタはまた手頃な卓上計算機にもなる。

Python 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を可能にしている。Python で書かれたプログラムは，等価な C や C++ のプログラムよりも典型的にはかなり短い。なぜなら，

- 高水準なデータ型は，複雑な演算を一つの文で表現することを可能にしている。
- 文のグループ分けは，begin/end ブラケットのかわりに段付け (indentation) でなされる。
- 変数や引数の宣言が不要である。

Python は**拡張可能** *(extensible)* である。つまり，もし君が C でプログラムを作る方法を知っているならば，インタープリタに新しい組込みの関数やモジュールを追加して，クリティカルな演算を最大の速度で実行したり，バイナリ形式でしか入手できない (ベンダ固有のグラフィクス・ライブラリなどの) ライブラリに Python プログラムをリンクさせることはたやすい。君がその気になれば，C で書かれたアプリケーションの中に Python インタープリタをリンクして，Python をアプリケーションの拡張言語ないしコマンド言語として使うこともできる。

なお，この言語の名前は BBC のショー “Monty Python’s Flying Circus” にちなんだものであり，不快な爬虫類とは無関係である。Monty Python のスキットをドキュメンテーションの中で引用することは差し支えないどころではない。それは推奨されている!

1.1 ここからどこへ

さて君たちは皆 Python にエキサイトして，もっと詳しく調べてみたくなったことだろう。言語を習う最良の方法はそれを使うことだから，ここではそうすることを勧める。

次の章ではインタープリタを使う操作手順を説明する。これはかなりお約束な情報だが，あとで示される例を試してみるためには欠かせない。

チュートリアルの残りでは Python の言語とシステムのさまざまな機能を例を通して紹介する。簡単な式と文とデータ型から始め，関数とモジュールを経て，最後に例外や利用者定義クラスのような高度な概念に触れる。

CHAPTER

TWO

Python インタープリタを使う

2.1 インタープリタを起動する

Python インタープリタは，それが利用可能になっているマシンには，普通 '/usr/local/bin/python' としてインストールされている。だから，君の UNIX シェルの検索パスに '/usr/local/bin' を入れれば，シェルにコマンド

```
python
```

を打鍵することでインタープリタを開始できる。インタープリタをどのディレクトリに置くかはインストール時のオプションだから，ほかの場所もあり得る。君のまわりの Python のグルーかシステム管理者に問い合わせられたい (たとえば '/usr/local/python' がよくある候補地である)。

一次プロンプトで end-of-file 文字 (UNIX では `Control-D`，DOS や Windows では `Control-Z`) を打鍵すると，インタープリタは exit ステータス 0 で終了する。もしこれが働かないときは，'`import sys; sys.exit()`' というコマンドを打鍵することによってインタープリタを終了できる。

インタープリタの行編集機能は普通あまり洗練されていない。UNIX では，インタープリタをインストールした人が GNU readline ライブラリのサポートを有効にしているかもしれない。これはより高度な対話的編集およびヒストリ機能を追加する。コマンド行編集がサポートされているかどうかを調べるには，最初に現れた Python プロンプトに対して `Control-P` を打鍵してみるのが，おそらく最も早い。もしベルが鳴れば，コマンド行編集が可能である。キーの手ほどきについては付録 A を見られたい。もしも何も起こらないようにみえたり，あるいは `^P` がエコーされたなら，コマンド行編集は利用可能になっていない。単にバックスペースを使って現在行から文字を消せるだけだろう。

インタープリタはいくぶん UNIX シェルのように動作する。つまり，標準入力を tty デバイスに接続した状態で呼び出すとコマンドを対話的に読み取って実行し，ファイル名を引数にしたりファイルを標準入力にして呼び出すとそのファイルから**スクリプト**を読み取って実行する。

インタープリタを開始するもう一つの方法は '**python -c** *command* `[arg] ...`' である。これはシェルの **-c** オプションと同じように*command* の文を実行する。Python の文はしばしばスペースなどの，シェルにとって特別な文字を含むから，*command* 全体をダブル・クォートで囲むのがベストだ。

'`python file`' と '`python <file`' には一つの違いがあることに注意しよう。後者では `input()` や `raw_input()` の呼出しなどのプログラムからの入力要求に対して *file* が充てられる。このファイルはプログラムが実行を開始する前に既にパーサによって最後まで読み取られているから，プログラムは即座に end-of-file に出会うことになる。前者では (君が普通期待するように) 入力要求に対して何であれ Python インタープリタの標準入力に接続されたファイルまたはデバイスが充てられる。

スクリプト・ファイルを使うとき，スクリプトを走らせた後そのまま対話モードに入れると便利な場合がある。そうするにはスクリプトの前に **-i** を渡せばよい。(スクリプトが標準入力から読み取られるときは，前の段落の説明と同じ理由で，これは働かない)。

2.1.1 引数渡し

インタープリタに渡されたスクリプト名とそれ以降の付加的な引数は，変数 `sys.argv` としてスクリプトへ渡される。これは文字列のリストである。その長さは少なくとも 1 である。スクリプトも引数も与えられないとき，`sys.argv[0]` は空文字列になる。スクリプト名が (標準入力を意味する) `'-'` として与えられたとき，`sys.argv[0]` は `'-'` にセットされる。**-c** *command* が使われたとき，`sys.argv[0]` は `'-c'` にセットされる。**-c** *command* より後にあるオプションは，Python インタープリタのオプション処理によって消費されずに`sys.argv` に残されるから，*command* で処理することができる。

2.1.2 対話モード

コマンドが tty から読み取られるとき，インタープリタは**対話モード** *(interactive mode)* にあると言われる。このモードではインタープリタは次のコマンドを**一次プロンプト** *(primary prompt)* で催促する。一次プロンプトは普通，三つの大ナリ記号 (‘`>>> `’) である。継続行 (continuation line) に対しては**二次プロンプト** *(secondary prompt)* で催促する。これはデフォルトでは三つのドット (‘`... `’) である。インタープリタは，そのバージョン番号と著作権表示を言明する welcome メッセージを印字してから，最初のプロンプトを印字する。たとえば，

```
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
>>>
```

継続行は，複数の行から構成される構文を入力するとき必要とされる。例としてこの `if` 文を見てみよう。

```
>>> 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 インタープリタとその環境

2.2.1 エラー処理

エラーが起こると，インタープリタはエラー・メッセージとスタック・トレースを印字する。対話モードのときはそれから一次プロンプトに戻る。入力がファイルから来ているときはスタック・トレースを印字してから非零 exit ステータスで終了する。(`try` 文の `except` 節で処理された例外は，ここで言うところのエラーではない)。エラーには無条件に致命的 (fatal) であって非零 exit をひき起こすものがある。内部的な矛盾状態とある種のメモリ枯渇がこれにあてはまる。エラー・メッセージはすべて標準エラー・ストリームへ書かれる。実行コマンドからの通常の出力は標準出力へ書かれる。

中断 (interrupt) 文字 (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が取り消され，一次プロンプトへ戻る[1]。コマンドの実行中に中断文字を打鍵すると `KeyboardInterrupt` 例外がひき起こされる。この例外は `try` 文で処理できる。

2.2.2 実行可能な Python スクリプト

BSD の流れをくむ UNIX システムでは，(インタープリタが利用者の PATH 上にあると仮定して)

[1] GNU Readline パッケージに関する問題がこれを妨げることがある。

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

という行を Python スクリプトの先頭におき，そしてそのファイルに実行可能モードを与えることによって，シェル・スクリプトのように Python スクリプトを直接実行可能にできる。‘#!’ はファイルの最初の 2 文字でなければならない。ハッシュ (ないしポンド) 文字 ‘#’ は Python ではコメントを開始するために使われていることに注意されたい。

2.2.3 対話スタートアップ・ファイル

Python を対話的に使うとき，インタープリタを起動するたびに何がしかの標準的なコマンドを実行させるようにするとしばしば手頃で便利である。そうするには，君が望むスタートアップ時のコマンドを内容とするファイルの名前を，PYTHONSTARTUP という名前の環境変数にセットすればよい。これは UNIX シェルの ‘.profile’ の機能に似ている。

このファイルが読まれるのは対話セッションだけであり，Python がスクリプトからコマンドを読み取るときや，‘/dev/tty’ がコマンドのソースとして陽に与えられたとき (このときは，そのこと以外では対話セッションのように振舞う) には読まれない。このファイルは対話コマンドが実行されるのと同じ名前空間で実行されるから，これが定義したり輸入したオブジェクトは，限定子無しで対話セッションに使える。このファイルでプロンプト `sys.ps1` と `sys.ps2` を変更してもよい。

もしも君が何か付加的なスタートアップ・ファイルをカレント・ディレクトリから読みたいならば，グローバルなスタートアップ・ファイルの中で君がそのようにプログラムすればよい (たとえば ‘`if os.path.isfile('.pythonrc.py'): execfile('.pythonrc.py')`’ など)。もしスクリプトの中でスタートアップ・ファイルを使いたいならば，君はスクリクプトの中で陽にそうしなくてはならない。たとえば，

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

CHAPTER

THREE

Pythonへの非形式的な手ほどき

以下の例では，入力と出力をプロンプト (‘>>> ’ と ‘... ’) の有無で区別する。だから，例を再現するには，君はなんであれプロンプトより後にあるものを，プロンプトが現れているときに打鍵しなくてはならない。プロンプトで始まらない行はインタープリタからの出力である。例の中で二次プロンプトだけの行は，君が空行を打鍵しなければならないことを意味することに注意しよう。これは複数の行からなるコマンドを終わらせるために使われる。

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さえ，コメントを含んでいる。Python のコメントはハッシュ文字 ‘#’ で始まり，物理行の終わりまで続く。コメントは行の先頭にも，空白やコードの後にも現れてよいが，文字列リテラル (string literal) の内部に現れることはできない。文字列リテラル内部のハッシュ文字はただのハッシュ文字である。

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ない。"
```

3.1 Python を電卓として使う

簡単な Python コマンドを試してみよう。インタープリタをスタートさせ，一次プロンプト ‘>>> ’ が現れるのを待とう。(長くはかからないはずだ)

3.1.1 数

インタープリタは単なる電卓のように振舞う。つまり，君がインタープリタに式を打鍵すると，インタープリタがその値を書く。式の構文は素直だ。演算子 +, -, *, / は (Pascal や C など) ほかの大多数の言語と同じように働く。かっこはグループ化のために使うことができる。たとえば,

```
>>> 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 (‘=’) は変数へ値を代入するために使われる。代入値は書かれない。

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

いくつもの変数へ一つの値を同時に代入することもできる。

```
>>> x = y = z = 0  # x と y と z を零にする
>>> x
0
>>> y
0
>>> z
0
```

浮動小数点数もフルにサポートされている。混合型のオペランドをもつ演算子は，整数オペランドを浮動小数点数に変換する。

```
>>> 4 * 2.5 / 3.3
3.0303030303
>>> 7.0 / 2
3.5
```

複素数もサポートされている。虚数は接尾辞 ‘`j`’ または ‘`J`’ を付けて書かれる。非零の実部をもつ複素数は ‘(*real*+*imag*`j`)’ と書かれるか，または ‘`complex(`*real*`, `*imag*`)`’ 関数で造られ得る。

```
>>> 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)
```

複素数はいつも実部と虚部の二つの浮動小数点数として表現される。複素数 z から実部と虚部を取り出すには z.real と z.imag を使う。

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

浮動小数点数や整数への変換関数 (float(), int(), long()) は複素数には働かない — 複素数を実数に変換する唯一の正しい方法というものはない。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 を使う。

```
>>> a=1.5+0.5j
>>> float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use e.g. abs(z)
>>> a.real
1.5
>>> abs(a)
1.58113883008
```

対話モードでは，最後に印字された式が変数 _ に代入される。これにより，Python を電卓として使う時，連続した計算が多少楽になる。たとえば，

```
>>> tax = 17.5 / 100
>>> price = 3.50
>>> price * tax
0.61249999999999993
>>> price + _
4.1124999999999998
>>> round(_, 2)
4.1100000000000003
```

利用者はこの変数を読取り専用として扱うべきである。これへ値を陽に代入してはならない — そんなことをすれば，同じ名前をもった別個のローカル変数が造られ，魔法の振舞をもった組込み変数が隠されてしまう。

3.1.2 文字列

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

```
>>> '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もの方法で複数行にまたがることができる。改行はバックスラッシュでエスケープすることができる。たとえば[1],

```
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.\n"
print hello
```

これは下記を印字する。

```
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.
```

または，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る。三重クォートを使ったときは，行末をエスケープする必要はないが，それら行末も文字列に含まれることになる。

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

は下記の出力を生成する。

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

インタープリタは文字列演算の結果を，文字列が入力のために打鍵されるのと同じ形式で印字する。つまり，正確な値を表示すべく，クォートで囲み，クォート自身やその他の奇妙な文字をバックスラッシュでエスケープする。もし文字列がシングル・クォートを含み，ダブル・クォートを含まないならば，文字列はダブル・クォートで囲まれるが，そうでなければシングル・クォートで囲まれる。(後述の `print` 文を使えば，クォートやエスケープなしに文字列を書くことができる)

文字列は + 演算子で連結させる (くっつけ合わす) ことができ，* 演算子で反復させることができる。

[1] 訳注: 訳せば「これは複数の行を C 言語の場合と全く同じやりかたで含んでいるかなり長い文字列です。行の先頭にある空白が意味を持っていることに注意しなさい。」

```
>>> 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る。したがって，上記の最初の行は ‘`word = 'Help' 'A'`’ と書くこともできた。ただし，これは二つのリテラルにだけ働くのであって，任意の文字列式に働くわけではない。

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

文字列は添字付け可能であり，C と同じく，文字列の最初の文字の添字 (subscript ないし index) は 0 である。独立した文字型というものはなく，文字 (character) はサイズ 1 の文字列 (string) にすぎない。Icon と同じく，部分文字列は**スライス記法** *(slice notation)*，つまりコロンで分けた二つの添字で指定できる。

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

C の文字列と異なり，Python の文字列は変更できない。文字列内の添字付けられた位置への代入はエラーになる。

```
>>> 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
```

しかし，組み合わせた内容をもつ新しい文字列を造ることは簡単で効率的である。

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

スライス添字には便利なデフォルト値がある。第 1 の添字を省略すると，0 と見なされる。第 2 の添字を省略すると，スライスされる文字列のサイズと見なされる。

```
>>> word[:2]    # 最初の 2 文字
'He'
>>> word[2:]    # 最初の 2 文字を除くすべて
'lpA'
```

スライス演算には便利な不変式がある: `s[:i] + s[i:]` は `s` に等しい。

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

外道なスライス添字は，たしなみよく取り扱われる。すなわち，大きすぎる添字は文字列のサイズに置き換えられ，上限が下限より小さいときは空文字列が返される。

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

添字は負の数でもよい。そのときは右から数える。たとえば,

```
>>> 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'
```

だが -0 は 0 と実際のところ同一であり，したがってこのときは右から数えるわけではないことに注意しよう!

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

範囲外の負数のスライス添字は切り捨てられる。しかし，これを単一要素の (スライスでない) 添字で試みてはいけない。

```
>>> word[-100:]
'HelpA'
>>> word[-10]    # エラー
Traceback (most recent call last):
  File "<stdin>", line 1
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，添字が文字と文字の**あいだ** *(between)* を指していると考え，先頭文字の左端を 0 と数える方法である。このとき，*n* 文字の文字列の最後の文字の右端が添字 *n* に

なる。たとえば,

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

第 1 の行の数は文字列の添字 0...5 の位置を示し，第 2 の行は対応する負の添字を示す。*i* から *j* までのスライスは，それぞれ *i, j* と位置付けられた端と端のあいだの文字すべてからなる。

スライスの添字がともに非負で，ともに文字列の範囲内にあるならば，スライスの長さはその添字の差に等しい。たとえば `word[1:3]` の長さは 2 である。

組込み関数 `len()` は文字列の長さ (length) を返す。

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

3.1.3 Unicode 文字列

Python 2.0 以降，テキスト・データを格納するための新しいデータ型がプログラマに利用可能になった。それが Unicode オブジェクトである。これは Unicode データ (http://www.unicode.org 参照) を格納し，操作するために使用できるとともに，必要なところで自動変換を用意して，既存の文字列オブジェクトと上手く統合している。

Unicode には，現代および古代のテキストで使われたあらゆるスクリプトのあらゆる文字にそれぞれ一つの順序数を与える，という利点がある。以前は，スクリプト文字に対し高々 256 個の順序数しかなく，テキストは典型的には順序数をスクリプト文字へ写像するどれか 1 個のコードページに束縛されていた。このことはとりわけソフトウェアの国際化 (internationalization，普通 'i18n' と書かれる — 'i' + 18 文字 + 'n') に関して非常に大きな混乱をもたらした。Unicode はこれらの問題を，すべてのスクリプトに対して唯一のコードページを定義することで解決する。

Python で Unicode 文字列を造ることは，通常の文字列を造ることと同じくらい簡単だ。

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

クォートの前にある小文字の 'u' が Unicode 文字列を造ることを示す。もし文字列の中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そうすることができる。下記の例はその方法を表している。

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

エスケープ列 `\u0020` は順序数の値 0x0020 の Unicode 文字 (スペース文字) を与えられた位置に挿入することを示す。

他の文字はそれぞれの順序数の値をそのまま Unicode の順序数として使うことによって解釈される。もしも君のもっているリテラル文字列が多くの西洋諸国で使われている標準 Latin-1 エンコーディングならば，Unicode の下位 256 文字が Latin-1 の 256 文字と同じであることを，君はきっと便利だと思うだろう。

エキスパート向けに，通常の文字列の場合と同じく raw モードもある。文字列のはじめのクォートに 'ur' を接頭して，Python に *Raw-Unicode-Escape* エンコーディングを使わせる。これは小文字の 'u' の前に奇数

個の逆スラッシュがあるときだけ上記の `\uXXXX` 変換を適用する。

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

raw モードは，たとえば正規表現を書くときなど，君が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最も役に立つ。

これら標準のエンコーディングに加えて，Python は，既知のエンコーディングに基づいて Unicode 文字列を造る一連の手段を提供している。

組込み関数 `unicode()` はすべての登録されている Unicode codec (COder and DECoder, 符号化器および復号化器) へのアクセスを提供する。これらの codec が変換することのできるエンコーディングのうち，よく知られているものには，*Latin-1, ASCII, UTF-8* および *UTF-16* がある。後者二つは可変長のエンコーディングであり，各 Unicode 文字を 1 バイトまたは複数バイトに格納する。デフォルト・エンコーディングは通常は ASCII にセットされている。これは 0 から 127 までの範囲の文字を通過させ，他の文字はすべて受理せずエラーとする。Unicode 文字列が印字されたり，ファイルに書き込まれたり，`str()` で変換されるときには，このデフォルト・エンコーディングを使った変換が行われる。

```
»> u"abc"
u'abc'
»> str(u"abc")
'abc'
»> u"äöü"
u'\xe4\xf6\xfc'
»> str(u"äöü")
Traceback (most recent call last):
File "<stdin>", line 1, in ?
UnicodeError:  ASCII encoding error:  ordinal not in range(128)
```

Unicode 文字列を，特定のエンコーディングを使った 8 ビット文字列へ変換するために，Unicode オブジェクトは `encode()` メソッドを用意している。これは 1 引数をとる。引数はエンコーディング名である。エンコーディング名には小文字を使うのが望ましい。

```
»> u"äöö".encode('utf-8')
'\xc3\xa4\xc3\xb6\xc3\xbc'
```

もしも特定のエンコーディングで書かれているデータから，それに対応する Unicode 文字列を生成したいならば，`unicode()` 関数を，第 2 引数にエンコーディング名を与えて使えばよい。

```
>>> unicode('\xc3\xa4\xc3\xb6\xc3\xbc', 'utf-8')
u'\xe4\xf6\xfc'
```

3.1.4 リスト

Python は，別々の値を一緒にしてグループ化するために使う**複合** *(compound)* データ型を数多く備えている。その最も汎用的なものが**リスト** *(list)* である。リストは，かぎかっこに囲まれ，コンマで分けられた値 (項目) の並びとして書かれる。リストの各項目は必ずしもすべて同じ型でなくてもよい。

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

文字列の添字と同じく，リストの添字は 0 から始まる。スライスや連結ができることなども文字列と同様である。

```
>>> 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)* な文字列とは異なり，リストは個々の要素を変更することが可能である。

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

スライスへの代入も可能である。そしてこれはそのリストのサイズを変更させることさえできる。

```
>>> # 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る。

```
>>> 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) ことも可能である。たとえば,

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')     # 5.1 節を見よ
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

最後の例で `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よう! **オブジェクトの意味論**についてはまた後で立ち戻ることにする。

3.2 プログラミングへの第一歩

もちろん，私たちは Python を 2 足す 2 よりも複雑な仕事のために使うことができる。たとえば，私たちは *Fibonacci* 級数の最初の部分列を次のように書くことができる。

```
>>> # 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
```

この例は，いくつかの新しい機能を導入している。

- 最初の行は**多重代入** *(multiple assignment)* からなっている。変数 `a` と `b` は同時に新しい値 0 と 1 を得ている。最後の行にも多重代入が使われているが，それを見て分かるとおり，代入が行われるのは右辺の複数の式がすべて評価されてからである。右辺の複数の式は，左から右へと評価される。
- `while` ループは条件 (ここでは `b < 10`) が真である限り実行される。Python では，C と同じく，非零整数値は真であり，零は偽である。条件は文字列値やリスト値であってもよく，実際どんな列であってもよい。その場合，非零の長さならば真であり，空列ならば偽である。例で用いられたテストは単純な比較である。標準の比較演算子は C と同じように書かれる。すなわち，< (less than), > (greater than), == (equal to), <= (less than or equal to), >= (greater than or equal to) および != (not equal to) である。
- ループの**本体** *(body)* は**段付け** *(indentation)* されている。段付けは Python が複数の文をグループ化する方法である。Python は (いまだに!) インテリジェントな入力行編集機能を備えていないから，行を段付けするために君がいちいちタブやスペースを打鍵しなくてはならない。実際には君は Python への複雑な入力をテキスト・エディタで用意するだろうし，大多数のテキスト・エディタには自動段付け機能がある。対話的に複合文を入力する時には，その終わりを示すために空行を続けなくてはならない (なぜなら Python のパーサは，いつ君が最後の行を打鍵したのかを推測できないからだ)。一つの基本ブロック内の各行は，同じ量だけ段付けされなくてはならないことに注意しよう。

- `print` 文は与えられた (１個または複数の) 式の値を書く。これは，(前に電卓の例で私たちがしたように) 書きたい式をただ書いただけの場合とは，複数の式と文字列を処理する方法で異なる。文字列はクォートなしに印字され，項目間にはスペースが挿入されるから，君はこのようにナイスに書式化できる。

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

末尾のコンマは，出力後の改行を防ぐ。

```
>>> 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
```

最後の行が完了していないとき，インタープリタは一つ改行を書き込んでから次のプロンプトを印字することに注意しよう。

CHAPTER

FOUR

もっと制御フローの道具を

今しがた紹介した `while` 文のほかに，Python は他の言語でおなじみの普通の制御フロー文を，多少のひねりを伴いつつ備えている。

4.1 `if` 文

おそらく最もおなじみの文型は `if` 文である。たとえば,

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

零個以上の `elif` 部があってもよい。`else` 部を付けてもよい。キーワード '`elif`' は 'else if' をつづめたものであり，過剰な段付けを避けるのに役立つ。`if` … `elif` … `elif` … 列は，他の言語で見られる *switch* 文や *case* 文の代用品である。

4.2 `for` 文

Python の `for` 文は，君が C や Pascal で慣れてきたかもしれないものとは少しばかり違う。(Pascal のように) いつも数の等差数列上で繰返しをしたり，(C のように) 繰返しのステップと停止条件の両方の定義を利用者に任せたりするのではない。むしろ Python の `for` 文は，任意の列 (たとえばリストや文字列) の各項目に対し，それらが列に現れる順序で，繰返しをする。たとえば (なんのしゃれのつもりもないが),

```
>>> # 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
```

ループ内で繰返しの対象にしている列を書き換えることは安全ではない (このことは変化可能 (mutable) な

列型，すなわち，リストにだけ起こり得る)。もしも，たとえば特定の項目を二重化するなど，繰返しの対象にしているリストを書き換える必要があるならば，君はコピーに対して繰返しをしなくてはならない。スライス記法はこれを特に手軽なものにしている。たとえば，

```
>>> 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

もし数列上で繰返しをする必要があるなら，組込み関数 range() が手頃だ。これは等差数列からなるリストを生成する。たとえば，

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

与えた終端は，生成されるリストには決して含まれない。range(10) は 10 個の値からなるリストを生成し，その 10 個の値はそれぞれ，ちょうど長さ 10 の列の各項目に対応する正当な添字になっている。range を別の数から開始させることや，様々な増分 (負でもよい — これは 'ステップ' とも呼ばれる) を指定することも可能である。

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

列の添字の上で繰返しをするには，range() と len() を次のように組み合わせる。

```
>>> 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 と同じく，それを取り囲む最小の for または while ループの外へ脱出する。

continue 文は，これもまた C から借りてきたものであり，ループの次の繰返しへと続ける。

ループ文には else 節があってもよい。これは (for で) リストが尽きてループが停止したとき，または (while で) 条件が偽になったときに実行されるが，break 文でループが終了したときは実行されない。このことを，素数を探す下記のループで例示する。

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...            print n, 'equals', x, '*', n/x
...            break
...     else:
...          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 文は何もしない。これは，文が構文上要求されているが，プログラムが何の動作も要求していないときに使われる。たとえば，

```
>>> while 1:
...       pass # keyboard interrupt を busy-wait する
...
```

4.6 関数を定義する

私たちは Fibonacci 級数を任意の限界まで書く関数を造ることができる。

```
>>> def fib(n):    # n までのフィボナッチ級数を書く
...     "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
```

キーワード def は関数の**定義** *(definition)* を導入する。その後に関数の名前と，かっこで囲んだ仮引数の並びが必ず続く，関数の本体を構成する文は，その次の行から，必ず段付けされて始まる。関数本体の最初の文は文字列リテラルでもよい。この文字列リテラルは関数のドキュメンテーション文字列 (documentation string)，つまり *docstring* となる。

docstring を使って，オンライン文書や印刷文書を自動生成したり，利用者が対話的にコードを閲覧できるようにするツールがある。君が書くコードに docstring を入れることは良い習慣だから，それを習いにするよう努めよう。

関数の**実行** *(execution)* は，その関数のローカル変数のために使われる新しい記号表 (symbol table) を導入する。より正確にいえば，関数内のすべての変数代入は，値をローカル記号表へ格納する。しかるに変数参照は，まずローカル記号表を調べ，次にグローバル記号表を調べ，そしてそれから組込み名の表を調べ

る。したがって，関数の内部では，グローバル変数を参照することはできても，直接それへ値を代入することは (`global` 文で名前を挙げない限り) できない。

関数呼出しでの実引数は，呼出しの時，その関数のローカル記号表の中へ導入される。このように引数は**値渡し** *(call by value)* で渡される (ここで**値** *(value)* とは常にオブジェクト**参照** (object *reference*) である。オブジェクトの値 (value of the object) ではない)[1]。関数がほかの関数を呼び出すとき，新しいローカル記号表がその呼出しのために造られる。

関数定義は関数名を現在の記号表の中へ導入する。関数名の値は，インタープリタが利用者定義関数として認識する型をもつ。この値をほかの名前へ代入してよい。代入するとその名前も関数として使うことができる。これは一般的な改名 (renaming) のからくりとして働く。

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

君は `fib` が関数 (function) でなく手続き (procedure) だと異議を唱えるかもしれない。Python では，C と同じく，手続きとは値を返さない関数にすぎない。ただし，技術的に言えば，かなりつまらない値とはいえ，手続きも値を返している。その値を `None` という (これは組込み名である)。もしも書く値が `None` だけならば，インタープリタは通常それを書くことを差し控える。もしも君が本当にそれを見たいならば，このようにしてそれを見ることができる。

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

Fibonacci 級数を印字するかわりにその数列を返す関数を書くことは簡単だ。

```
>>> 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)    # 下記を見よ
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # 関数を呼び出す
>>> f100                # 結果を書く
[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る。

- `return` 文は関数から一つの値を返す。式の引数がない `return` は `None` を返す。手続きの終端から落っこちたときも `None` を返す。
- 文 `result.append(b)` は，リスト・オブジェクト `result` の**メソッド** *(method)* を呼び出す。メソッドとはオブジェクトに「属している」関数であり，`obj.methodname` として指名される。ここで `obj` はなんらかのオブジェクト (これは式であってもよい) であり，`methodname` はそのオブジェクトの型によって定義されるメソッドの名前である。定義されるメソッドは型によって様々である。別々の型のメソッドならば，あいまいさを生ずることなく同じ名前にできる。(このチュートリアルで後から論ずるように，**クラス** *(class)* を使えば，君独自のオブジェクト型とメソッドを定義することが可能だ)。例に示したメソッド `append()` は，リスト・オブジェクトに対して定義されており，

[1] 実際には，**オブジェクト参照渡し** *(call by object reference)* と書くのが適当だっただろう。なぜなら，変化可能 (mutable) なオブジェクトを渡したとき，呼び出した側は，なんであれ呼び出された側がそれに施す変更 (たとえばリストへの項目の挿入) を見ることになるからだ。

リストの終端に新しい要素を追加する。この例では ‘`result = result + [b]`’ と等価だが，より効率的である。

4.7 もっと関数の定義について

可変個数の引数をとる関数も定義できる。三つの形式があり，それらは組み合わせ可能である。

4.7.1 デフォルト引数値

その最も使える形式は，１個または複数の引数に対するデフォルト値の指定である。これは定義された個数よりも少ない引数で呼び出せる関数を造る。たとえば，

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while 1:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'): return 1
        if ok in ('n', 'no', 'nop', 'nope'): return 0
        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る。

デフォルト値は，関数定義の時点で，その関数を**定義する**スコープの中で評価されるから，たとえば，

```
i = 5
def f(arg = i): print arg
i = 6
f()
```

は `5` を印字する。

重要な警告: デフォルト値はたった１回だけ評価される。これが差になるのは，デフォルトがリストや辞書のような変化可能オブジェクトのときである。たとえば，下記の関数は次々の呼出しで渡される引数を累積する。

```
def f(a, l = []):
    l.append(a)
    return l
print f(1)
print f(2)
print f(3)
```

これはこう印字される。

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

もしも次の呼出しとデフォルトを共有したくないならば，かわりにこのように関数を書けばよい。

```
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してもよい。たとえば下記の関数

```
def parrot(voltage, state='a stiff', action='voom', 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, "!"
```

は下記のどの方法で呼び出してもよい。

```
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，下記の呼出しはすべて不正である。

```
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')  # 未知のキーワードである
```

一般に，実引数並びでは，位置引数 (positional argument) は，どのキーワード引数 (keyword argument) よりも前でなければならず，キーワードは仮引数の名前から選ばなければならない。仮引数にデフォルト値があるかどうかは重要ではない。どの引数も値を重複して受けとってはならない — 位置引数に対応する仮引数名を，同じ呼出しの中でキーワードとして使うことはできない。ここにある例は，この制約により失敗している。

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: keyword parameter redefined
```

最後の仮引数が ***name* という形式のとき，それは仮引数に対応しないキーワードのキーワード引数すべてからなる辞書を受けとる。これを (次の節で述べる) **name* という形式の仮引数と組み合わせてもよい。**name* は仮引数並びを超えた位置引数からなるタプルを受けとる。(**name* は ***name* より前に出現しなくてはならない)。たとえば，もし次のような関数を定義したならば，

```
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
    for kw in keywords.keys(): print kw, ':', keywords[kw]
```

それはこのように呼び出せる。

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

そしてもちろんこう印字される。

```
-- 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
```

4.7.3 任意引数並び

最後に，最も使われないオプションとして，任意個数の引数で呼び出せる関数の指定がある。これらの引数は１個のタプルにくるまれる。可変個数引数より前に，零個以上の通常の引数が出現してもよい。

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

4.7.4 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と Lisp によく見られるいくつかの機能が Python に加えられた。キーワード `lambda` を使って，名前のない小さな関数を造ることができる。たとえば '`lambda a, b: a+b`' は二つの引数の和を返す関数である。ラムダ形式 (lambda form) は，関数オブジェクトが求められるところならどこに使ってもよい。ラムダ形式は構文上，単一の式に制限されている。ラムダ形式は意味論上，通常の関数定義に対する構文上の糖衣にすぎない。入れ子にした関数定義と同じく，ラムダ形式は，それを取り囲むスコープからの変数を参照できないが，この問題はデフォルト引数値を賢く使うことで克服できる。たとえば，

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

4.7.5 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の内容と書式について規約ができつつある。

第１行はつねに，対象の目的の，短く簡明なまとめとする。簡潔さを求め，対象の名前や型をわざわざ述べることはしない。これらは他の方法で得られるからだ (ただし，名前がたまたま関数の演算を記述する動詞である場合を除く)。この行は大文字で始めてピリオドで終える。

ドキュメンテーション文字列が複数行からなるときは，第２行を空行にして，まとめと，残りの記述を視覚的に分ける。その次の行から，対象の呼出し規約や副作用などを記述する１個または複数の段落を書く。

Python のパーサは複数行にわたる Python の文字列リテラルから段付けをはぎとることはしないから，もし望むなら，ドキュメンテーションを処理するツールが段付けをはぎとらなければならない。これは次の規約に従って行う。文字列の第１行よりも**後**の最初の非空行が，ドキュメンテーション文字列全体に対する段付けの量を決定する (一般に第１行は，文字列を開始するクォートにとなりあっており，その段付けは文字列リテラルにおいて明らかではないから，使うことはできない)。それから，この段付けと “等価な” 量の空白を，その文字列のすべての行の先頭からはぎとる。それより少なく段付けされた行は現れてはならないが，もし現れたときは，その先頭の空白をすべてはぎとる。空白の等価性は，タブを展開した後にテストする (通常は 8 文字スペース換算とする)。

ここに複数行 docstring の例がある。

```
>>> 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.
```

CHAPTER

FIVE

データ構造

この章は君が既に学んだことのいくつかを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する。

5.1 もっとリストについて

リストというデータ型には，もういくつかのメソッドがある。ここにリスト・オブジェクトのすべてのメソッドを挙げる。

append(x) 項目をリストの末尾に追加する。`a[len(a):] = [x]` と等価である。

extend(L) 与えられたリストの項目すべての追加によってリストを拡張する。`a[len(a):] = L` と等価である。

insert(i, x) 与えられた位置に項目を挿入する。第１引数を添字としてその前に項目を挿入する。したがって `a.insert(0, x)` はリストの先頭に挿入する。そして `a.insert(len(a), x)` は `a.append(x)` と等価である。

remove(x) x を値とする最初の項目をリストから削除する。該当する項目がなければエラーである。

pop([i]) 与えられた位置にある項目をリストから削除し，その項目を戻り値とする。もし添字が指定されていなければ，`a.pop()` はリスト末尾の項目をリストから削除し，その項目を戻り値とする。

index(x) x を値とする最初の項目の，リストでの添字を返す。該当する項目がなければエラーである。

count(x) リストでの x の出現回数を返す。

sort() リストの項目を，その場で (in place)，ソートする。

reverse() リストの要素を，その場で，逆順にする。

リストのメソッドの大半を使った例:

```
>>> 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・メソッドは，リストをスタック (stack) として使うことを非常に容易にしている。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る (“last-in, first-out”)。スタックのトップに項目を追加するには `append()` を使う。スタックのトップから項目を取り出すには `pop()` を添字指定なしで使う。例えば，

```
>>> 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) として使うことも手軽にできる。キューでは，最初に追加された要素が最初に取り出される (“first-in, first-out”)。キューの末尾に項目を追加するには `append()` を使う。キューの先頭から項目を取り出すには `pop()` を `0` を添字として使う。例えば，

```
>>> 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 関数型プログラミングの道具

リストで使うと非常に便利な三つの組込み関数がある。それは `filter()` と `map()` と `reduce()` である。

'`filter(`関数, 列`)`' は，列から 関数(項目) が真である項目を選んで，それらからなる (なるべく列と同じ型の) 列を返す。たとえば，いくつかの素数を計算するには,

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

'`map(`関数, 列`)`' は，列の各項目について 関数(項目) を呼び出し，その戻り値からなるリストを返す。たとえば，いくつかの３乗数を計算するには,

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

２個以上の列を渡してもよい。関数は列と同じ個数の引数をとる必要がある。各列からの対応する項目 (ある列がほかよりも短いときは，項目のかわりに None) を引数として関数を呼び出す。関数のかわりに None を渡すと，引数をそのまま返す関数に置き換えられる。

これら二つの特別な場合を組み合わせると，'`map(None, `*list1*`, `*list2*`)`' はリストのペアをペアのリストにする手軽な方法であることが分かる。たとえば,

```
>>> seq = range(8)
>>> def square(x): return x*x
...
>>> map(None, seq, map(square, seq))
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49)]
```

'`reduce(`関数, 列`)`' は，２引数の関数を，まず列の最初の２項目について呼び出し，それからその結果と列の次の項目について呼び出し，等々として構成された単一の値を返す。たとえば 1 から 10 までの数の総和を計算するには,

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

列に項目が 1 個だけならば，その値が返される。列が空ならば，例外がひき起こされる。

第 3 引数を渡して開始値を指定してもよい。この場合，列が空ならば開始値が返され，そうでなければ，関数を，まず開始値と列の先頭項目に適用し，それからその結果と次の項目に適用し，等々となる。たとえば，

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

5.1.4 リストの内包表記

リストの内包表記 (list comprehension) は，`map()` や `filter()` や `lambda` の使用に頼らずにリストを造る簡明な方法を用意している。結果として得られるリスト定義は，それらの構文要素を使って組み立てたリストよりも，しばしば明快になる傾向がある。リストの内包表記の内容は，1 個の式の後に 1 個の `for` 節が続き，そしてその後に零個以上の `for` 節または `if` 節が続いたものである。結果は，その式を，後に続く `for` 節と `if` 節の文脈で評価して得られる値のリストになる。タプルへと評価される式を書きたければ，丸かっこで囲まなければならない。

```
>>> 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]
[[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
    [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]
```

5.2 `del` 文

リストから項目を，値ではなく添字で指定して，削除する方法がある。それが `del` 文である。この文はリストからのスライスの削除にも使える (それを私たちは以前はスライスへの空リストの代入でおこなった)。

たとえば,

```
>>> 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` は変数そのものの削除にも使える。

```
>>> del a
```

名前 `a` を参照することは，これ以降 (少なくともほかの値をそれに代入するまでは) エラーになる。`del` のほかの使用方法については，また後でみることにしよう。

5.3 タプルと列

リストと文字列には多くの共通の性質 (たとえば添字付けやスライス演算など) があることを私たちは見てきた。これらは**列** *(sequence)* データ型の二つの例である。Python は進化途上の言語だから，ほかの列データ型が追加されることもあるかもしれない。標準的な列データ型はもう一つある。それが**タプル** *(tuple)* である。

タプルはコンマで区切られたいくつかの値からなる。たとえば,

```
>>> 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))
```

御覧のように，入れ子のタプルを正しく解釈できるようにするため，出力時にはタプルはいつも丸かっこで囲まれる。入力時には丸かっこで囲んでも，囲まなくてもよいが，(もしもタプルがより大きな式の一部ならば) どのみち丸かっこが必要になる場合がしばしばである。

タプルには，(x, y) 座標対や，データベースからの従業員レコードなど，多くの用途がある。タプルは，文字列と同じく，変化不可能 (immutable) である。つまり，タプルの個々の項目へ代入することはできない (ただし，君はスライスと連結を使って同様の効果の多くを模倣できる)。リストなどの，変化可能 (mutable) なオブジェクトを含んだタプルを造ることもできる。

問題は 0 個または 1 個の項目からなるタプルの構築だが，これを解決するため，構文に特別な小細工がある。空タプルは，空の丸かっこのペアで構築される。１項目のタプルは，１個の値の後にコンマを続けることで構築される (単一の値を丸かっこで囲むことでは出来ない)。美しくないが，効果的だ。たとえば,

```
>>> 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・パッキング** *(tuple packing)* の例である。ここでは値 `12345` と `54321` と `'hello!'` が一つのタプルにパックされる。逆の演算も可能である。たとえば,

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

これは，いみじくも，**列アンパッキング** *(sequence unpacking)* と呼ばれている。列アンパッキングは，左辺の変数並びの要素数が，列の長さと同じであることを要求する。多重代入が実はタプル・パッキングと列アンパッキングの組み合わせにすぎないことに注意しよう!

ここにはわずかな非対称性がある。すなわち，複数個の値のパッキングはいつもタプルを造るが，アンパッキングはどんな列にも働く。

5.4 辞書

Python に組み込まれているもう一つの便利なデータ型が**辞書** *(dictionary)* である。辞書は，ほかの言語では “連想記憶 (associative memory)” とか “連想配列 (associative array)” として見出されることがある。ある範囲の数によって添字付けられる列とちがい，辞書は**キー** *(key)* によって添字付けられる。キーはどんな変化不可能型 (immutable type) でもよい。文字列と数はいつでもキーになり得る。文字列か数かタプルだけからなるタプルならば，それもキーとして使ってよい。しかし，もしもタプルがなんであれ変化可能 (mutable) なオブジェクトを直接または間接に含んでいるならば，それをキーとして使うことはできない。リストは，スライス代入や添字付け代入はもちろん `append()` や `extend()` のメソッドを使って，その場で改変 (modify in place) できるから，キーとしては使えない。

辞書とは **キー***:* **値** のペアからなる順序付けられていない集合であり，各キーは (一つの辞書の中で) 一意的であることが要求されている，と考えるのが最も良い。波かっこのペア `{}` は空の辞書を造る。コンマで区切った キー:値 のペアの並びを波かっこの中に置くと，初期値として キー:値 のペアがその辞書に加わる。これは出力時に辞書が書かれる方法でもある。

辞書についての主な演算は，なんらかのキーとともに値を格納することと，キーを与えてその値を取り出すことである。キー:値 のペアを `del` で削除することもできる。もしも既に使用中であるキーを使って格納すると，そのキーに結合していた古い値は忘れ去られる。存在しないキーを使って値を取り出すことはエラーである。

辞書オブジェクトの `keys()` メソッドは，その辞書で使われているすべてのキーを順不同に並べたリストを返す (もしソートされたリストが欲しければ，キーのリストに `sort()` を適用すればよい)。ある単一のキーが辞書にあるかどうか調べるには，その辞書の `has_key()` メソッドを使う。

ここにあるのは辞書を用いた小さな例である。

```
>>> 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
```

5.5 もっと条件について

前述の `while` 文や `if` 文で使われる条件 (condition) を，比較 (comparison) 以外の演算子で構成することもできる。

比較演算子 `in` と `not in` は，値が列に出現するか (出現しないか) どうかを調べる。演算子 `is` と `is not` は，二つのオブジェクトが本当に同じオブジェクトかどうか比べるが，このことはリストのような変化可能オブジェクトにとってだけ重要である。すべての比較演算子は同じ優先順位であり，そしてそれはすべての数値演算子の順位よりも低い。

比較は連鎖 (chain) することができる。たとえば `a < b == c` は，`a` が `b` より小さく，かつ `b` が `c` に等しいかどうかをテストする。

複数の比較を論理演算子 `and` と `or` で組み合わせてもよいし，比較の (またはなんであれ論理式の) 結果を `not` で否定してもよい。これらはすべて，比較演算子よりもさらに低い優先順位である。その中では `not` の優先順位が最高であり，`or` が最低である。したがって `A and not B or C` は `(A and (not B)) or C` と等価である。もちろん，丸かっこを使えば，望みの組み合わせを表現できる。

論理演算子 `and` と `or` はいわゆる**短絡** *(shortcut)* 演算子である。つまり，引数が左から右へ評価され，結果が決定したらすぐに評価が止まる。たとえば，もしも `A` と `C` が真だが `B` が偽ならば，`A and B and C` は式 `C` を評価しない。一般に，短絡演算子の戻り値は，それが論理値としてではなく一般の値として使われるときは，最後に評価された引数である。

比較やその他の論理式の結果を変数に代入することもできる。たとえば，

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

Python では，C とちがい，代入が式の内部に出現できないことに注意しよう。C プログラマはこのことで不平を言うかもしれないが，これは C プログラムの中で遭遇するありふれた種類の問題，すなわち，式の中で == のつもりで = とタイプしてしまうことを回避する。

5.6 列およびその他の型の比較

列オブジェクトは，同じ列型の他のオブジェクトと比較してよい。比較には**辞書式** *(lexicographical)* 順序が用いられる。つまり，まず最初の項目どうしが比較され，そしてもしそれらが異なっていればそれで比較結果が決まる。もしそれらが等しければ，その次の項目どうしが比較される。以下，これが列が尽きるまで繰り返される。もしも比較される項目どうし自身が同じ型の列ならば，辞書式比較が再帰的に行われる。

もし二つの列の項目がすべて等しければ，二つの列は等しいと見なされる。もし一方の列が他方の列の先頭部分列ならば，短い列の方が小さいと見なされる。文字列についての辞書式順序は，個々の文字について ASCII 順序を用いる。下記は同じ型の列どうしの比較の例である。

```
(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)
```

異なった型のオブジェクトの比較が合法であることに注意しよう。その結果は決定性をもつが恣意的である。つまり，型はその名前によって順序付けられる。たとえば，リスト (list) はつねに文字列 (string) よりも小さく，文字列はつねにタプル (tuple) よりも小さい。混合数値型はその数値にしたがって比較される。したがってたとえば 0 は 0.0 と等しい[1]。

[1] 異なった型のオブジェクトを比較する規則に頼るべきではない。それらは言語の将来のバージョンでは変更されるかもしれない。

CHAPTER

SIX

モジュール

君が Python インタープリタを終了させ，そして再び起動したとき，君がしてきた定義 (関数と変数) は失われている。だから，もし君が何かもっと長いプログラムを書きたいならば，じかに入力するのではなく，テキスト・エディタを使ってインタープリタへの入力を用意し，そして入力としてそのファイルを使ってインタープリタを走らせるのが良い。これは**スクリプト** *(script)* の作成として知られている。君のプログラムが長くなるにつれ，メンテナンスをしやすくするため，プログラムを複数のファイルに分割したくなるかもしれない。あるいは，いくつかのプログラムで書いてきた便利な関数を，その定義をいちいち各プログラムのなかに書き写すことなく使いたくなるかもしれない。

これをサポートするため，Python には，いくつかの定義を一つのファイルに置き，そしてそれらをスクリプトや対話的に起動したインタープリタの中で使う方法がある。そのようなファイルを**モジュール** *(module)* と呼ぶ。モジュールからの定義は，ほかのモジュールの中へ，または**メイン**・モジュール (トップ・レベルで実行されるスクリプトの中や電卓モードの中で，君がアクセスできる変数の集まり) の中へ**輸入** *(import)* することができる。

モジュールは Python の定義と文からなるファイルである。そのファイル名はモジュール名に接尾辞 '.py' を付けたものである。モジュール内では，そのモジュールの (文字列としての) 名前をグローバル変数 `__name__` の値として得ることができる。例として，君の好きなテキスト・エディタを使って，現在のディレクトリに以下の内容で 'fibo.py' という名前のファイルを作成しよう。

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

次に Python インタープリタに入り，このモジュールを以下のコマンドで輸入しよう。

```
>>> import fibo
```

これは `fibo` の中で定義されている各関数の名前をじかに現在の記号表に入れているわけではない。モジュール名 `fibo` だけを現在の記号表に入れている。関数へはモジュール名を使ってアクセスできる。

```
>>> 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'
```

もしも関数をたびたび使うつもりならば，それをローカル名に代入してもよい。

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

6.1 もっとモジュールについて

モジュールには関数定義だけでなく実行文を含めてもよい。これらの文の目的はモジュールの初期化である。それらはモジュールがどこであれ**最初**に輸入されたときだけ実行される[1]。

モジュールは各自，プライベートな記号表を持っており，モジュールの中で定義されたすべての関数はそれをグローバル記号表として使用する。したがって，モジュールの著者は，利用者のグローバル変数と偶然かちあうかもしれないという心配をすることなく，モジュールの中でグローバル変数を使うことができる。他方，君は，もしも自分が何をしているのかをわきまえているならば，モジュールの関数への参照と同じ記法 `modname.itemname` を使って，モジュールのグローバル変数に触ってもよい。

モジュールはほかのモジュールを輸入できる。慣習であって要求ではないが，すべての `import` 文をモジュールの先頭に置くべきである (これに関してはスクリプトでも同様である)。輸入されるモジュールの名前が，輸入をするモジュールのグローバル記号表に置かれる。

`import` 文には，モジュールからの名前を，輸入をするモジュールの記号表の中へじかに輸入するという変種がある。たとえば，

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

これは輸出元のモジュールの名前を，ローカルな記号表に導入することはしない (だから，この例では `fibo` は定義されない)。

モジュールが定義する名前をすべて輸入するという変種さえある。

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

これは，下線 (_) で始まる名前を除くすべての名前を輸入する。

6.1.1 モジュール検索パス

`spam` という名前のモジュールを輸入するとき，インタープリタは 'spam.py' という名前のファイルを，まず現在のディレクトリから，そして次に環境変数 PYTHONPATH で指定されたディレクトリの並びから検索する。この環境変数はシェル変数 PATH と同じ構文，つまりディレクトリ名の並びである。PYTHONPATH

[1] 実際には関数定義もまた「実行」される「文」であり，その実行は関数名をモジュールのグローバル記号表に入れる。

がセットされていないとき，またはファイルがそこから見つからないとき，検索はインストールの仕方に依存するデフォルトのパスで続行される。UNIX では，これは普通 '.:/usr/locall/lib/python' である。

実際には，モジュールは，変数 `sys.path` が与えるディレクトリの並びから検索される。この変数は入力スクリプトの所在するディレクトリ (または現在のディレクトリ)，PYTHONPATH，およびインストールの仕方に依存するデフォルト値で初期化される。これにより，自分が何をしているのかをわきまえている Python プログラムは，モジュール検索パスを修正または置換することができる。後述の標準モジュールの節を参照されたい。

6.1.2 "コンパイル" された Python ファイル

沢山の標準モジュールを使う短いプログラムにとって重要な起動時間の高速化として，もしも '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ならば，それはモジュール `spam` の "バイト=コンパイル" 済みのバージョンであると仮定される。'spam.pyc' を造るため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，もしもこれが不一致ならば 'spam.pyc' ファイルは無視される。

通常，君は 'spam.pyc' ファイルを造るために何もする必要はない。'spam.py' が無事コンパイルされたときはいつでも，コンパイルされたバージョンを 'spam.pyc' へ書くことが試みられる。この試みが失敗してもエラーではない。なんらかの理由でファイルが完全には書けなかったとき，結果として残された 'spam.pyc' ファイルはその後，無効であると認識され，したがって無視されることになる。'spam.pyc' ファイルの内容はプラットフォームに依存しないから，Python モジュールのディレクトリを，別々のアーキテクチャのマシン間で共有することができる。

エキスパートへの助言:

- Python インタープリタが **-O** フラグ付きで起動されたときは，最適化 (optimize) されたコードが生成され，'.pyo' ファイルに格納される。オプティマイザは今のところあまり役に立たない。それはただ `assert` 文と `SET_LINENO` 命令を除去するだけである。**-O** が使われたときは，**すべて**のバイトコードが最適化される。したがって `.pyc` ファイルは無視され，`.py` ファイルが最適化バイトコードへとコンパイルされる。
- 二つの **-O** フラグ (**-OO**) を Python インタープリタへ渡すと，バイトコード・コンパイラは，あるまれな場合にプログラムを機能不全にするかもしれない最適化を実行する。今のところは，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 '.pyo' ファイルにするだけである。この文字列が利用可能であることに頼っているプログラムがあるかもしれないから，君は，自分が何をしているのかわきまえているときだけ，このオプションを使うべきである。
- プログラムが '.pyc' ファイルや '.pyo' ファイルから読まれたとしても，'.py' ファイルから読まれたときと比べ，なんら速く走るわけではない。'.pyc' ファイルや '.pyo' ファイルで高速化されるのは，それらがロードされる速度だけである。
- スクリプトを，コマンド行で名前を与えて走らせたときは，そのスクリプトのバイトコードが '.pyc' ファイルや '.pyo' ファイルに書かれることはない。したがって，スクリプトの起動時間を短縮するには，そのコードの大部分をモジュールへ移し，そしてそのモジュールを輸入する小さなブートストラップ・スクリプトにするとよいかもしれない。'.pyc' または '.pyo' ファイルをじかにコマンド行で指名することも可能である。
- 同じモジュールについて，ファイル 'spam.py' がなくて 'spam.pyc' (または **-O** が使われるときは 'spam.pyo') というファイルだけがあってもよい。このことは Python コードのライブラリを，リバース・エンジニアリングがかなり困難な形式で，配布するために利用できる。
- モジュール `compileall` は，あるディレクトリのすべてのモジュールについて，'.pyc' ファイル (または **-O** が使われたときは '.pyo' ファイル) を造ることができる。

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おり，別のドキュメント *Python Library Reference* (これ以降 "Library Reference") で記述されている。モジュールにはインタープリタの中に組み込まれているもの

もある。それらは言語の核の部分ではないが，効率のために，またはシステム・コールなどのオペレーティング・システムのプリミティブへのアクセスを提供するために組み込まれている。このようなモジュールのセットはコンフィグレーション・オプションである。たとえば amoeba モジュールは，なんであれ Amoeba プリミティブをサポートしているシステムでだけ用意される。いくらかの注目に値する 1 個のモジュールがある。それは sys である。これはどの Python インタープリタにも組み込まれている。変数 sys.ps1 と sys.ps2 は，一次と二次のプロンプトとして使われる文字列を定義する。

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

これら二つの変数はインタープリタが対話モードにあるときだけ定義される。

変数 sys.path は文字列のリストであり，モジュールを探すためのインタープリタの検索パスを決定する。これは環境変数 PYTHONPATH から，またはそれがセットされていなければ組込みのデフォルト値から，取られたデフォルトのパスに初期化される。君はこれを標準的なリスト演算を使って改変できる。たとえば,

```
>>> 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の名前を定義しているかを調べるために使われる。これは文字列のソートされたリストを返す。たとえば,

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__name__', 'argv', 'builtin_module_names', 'copyright', 'exit',
'maxint', 'modules', 'path', 'ps1', 'ps2', 'setprofile', 'settrace',
'stderr', 'stdin', 'stdout', 'version']
```

引数がないとき，dir() は君が現在定義している名前をリストする。

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

これは変数やモジュールや関数など，すべての種類の名前をリストすることに注意しよう。

dir() は組込みの関数や変数の名前はリストしない。もしもそれらのリストが欲しいならば，それらは標準モジュール __builtin__ で定義されている。

```
>>> import __builtin__
>>> dir(__builtin__)
['AccessError', 'AttributeError', 'ConflictError', 'EOFError', 'IOError',
'ImportError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
'MemoryError', 'NameError', 'None', 'OverflowError', 'RuntimeError',
'SyntaxError', 'SystemError', 'SystemExit', 'TypeError', 'ValueError',
'ZeroDivisionError', '__name__', 'abs', 'apply', 'chr', 'cmp', 'coerce',
'compile', 'dir', 'divmod', 'eval', 'execfile', 'filter', 'float',
'getattr', 'hasattr', 'hash', 'hex', 'id', 'input', 'int', 'len', 'long',
'map', 'max', 'min', 'oct', 'open', 'ord', 'pow', 'range', 'raw_input',
'reduce', 'reload', 'repr', 'round', 'setattr', 'str', 'type', 'xrange']
```

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ある。たとえば，モジュール名 A.B は，‘A’ というパッケージの ‘B’ という下位モジュールを表す。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 のような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る。

サウンド・ファイルとサウンド・データの統一的な処理のためのモジュール群 (“パッケージ”) を，君が設計しようとしていると仮定しよう。サウンド・ファイルのフォーマットには多くの種類がある (普通それらは ‘.wav’, ‘.aiff’, ‘.au’ など接尾辞で認識される) から，君は次第に増大するさまざまなファイル・フォーマット間の変換用モジュール群を作成し維持する必要があるかもしれない。君がサウンド・データに施そうとする演算にも (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) 多くの種類があるだろうから，さらに君はこれらの演算を施す一連のモジュールを果てしなく書くことになるだろう。君のパッケージの構造は (階層的なファイルシステムで表現すると) こんな感じになるだろう。

```
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
              ...
```

ファイル ‘__init__.py’ は，Python にそのディレクトリをパッケージを収めているものとして取り扱わせるために必要である。これは ‘string’ のようなよくある名前のディレクトリが，モジュール検索パスの後のほうで出現する正当なモジュールを，意図せず隠蔽してしまうことを防止する。最も簡単には ‘__init__.py’

はただの空ファイルでよいが，そのファイルで，パッケージに対する初期化コードを実行したり，後述する `__all__` 変数をセットしてもよい。

パッケージの利用者は個々のモジュールをパッケージから輸入できる。たとえば，

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

これは下位モジュール `Sound.Effects.echo` をロードする。それはフル・ネームで参照しなければならない。たとえば，

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

下位モジュールを輸入するもう一つの方法は，

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

これも下位モジュール `echo` をロードする。そしてこれは `echo` をパッケージ接頭辞なしで利用可能にするから，次のように使用できる。

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

さらにもう一つのバリエーションは，欲しい関数や変数をじかに輸入する方法である。

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

やはり，これも下位モジュール `echo` をロードするが，これはその関数 echofilter をじかに利用可能にする。

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

`from` *package* `import` *item* を使うとき，項目 *item* は，パッケージ *package* の下位モジュール (または下位パッケージ) でもよいし，*package* で定義される他の名前 (関数やクラスや変数などの名前) でもよいことに注意しよう。`import` 文は，*item* が *package* で定義されているかどうかをまず調べ，もしも定義されていなければ，モジュールだと仮定してロードを試みる。もしそれを見つけることができなければ，`ImportError` がひき起こされる。

対照的に，`import` *item.subitem.subsubitem* のような構文を使ったときは，最後の項目を除くどの項目もパッケージでなければならない。最後の項目は，モジュールまたはパッケージであり得るが，直前の項目の中で定義されるクラスや関数や変数であってはならない。

6.4.1 パッケージから * を輸入する

ところで利用者が `from Sound.Effects import *` と書いたら，何が起こるだろうか? 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下位モジュールがあるかが見つけ出され，そしてそれらすべてが輸入される，ということを期待したいところだろう。あいにく，この操作は Mac と Windows のプラットフォームではうまく働かない。それらのファイルシステムは必ずしもファイル名の大文字小文字について正確な情報を持っているわけではないからだ! これらのプラットフォームには，ファイル 'ECHO.PY' をモジュール `echo` として，`Echo` として，それとも `ECHO` として輸入すべきかどうかを知る確かな方法がない。(たとえば Windows 95 には，すべてのファイル名を，最初の文字を大文字にして表示するという困ったクセがある)。DOS の 8+3 のファイル名制限は，別の興味深い問題を長いモジュール名に対して追加している。

唯一の解決策は，パッケージの著者がパッケージの索引を陽に用意するという方法である。import 文は次の規約を使う。すなわち，もしもパッケージの ‘__init__.py’ コードが `__all__` という名前のリストを定義しているならば，そのリストは，`from` *package* `import *` に出会ったときに輸入すべきモジュール名のリストであると解釈される。新しいバージョンのパッケージをリリースする際にこのリストを最新に保つことは，パッケージの著者の責任である。パッケージの著者は，もしも自分のパッケージから * を輸入するという利用法を認めないならば，そのリストをサポートしないことにしてもよい。例として，ファイル `Sounds/Effects/__init__.py` は次のコードを収めているかもしれない。

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

これは `from Sound.Effects import *` が Sound パッケージのうち名前を挙げられた三つの下位モジュールを輸入するだろうということを意味する。

もしも `__all__` が定義されていなければ，文 `from Sound.Effects import *` は，どの下位モジュールもパッケージ `Sound.Effects` から現在の名前空間の中へ輸入**しない**。それはただ (何とかしてその初期化コード ‘__init__.py’ を実行して) パッケージ `Sound.Effects` が輸入されたということを確立し，そのパッケージで定義されている名前を何であれ輸入するだけである。これには ‘__init__.py’ で定義された名前 (とそこで陽にロードされた下位モジュール) が含まれる。また，以前の import 文で陽にロードされていたそのパッケージの下位モジュールも含まれる。たとえば，

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

この例では，echo モジュールと surround モジュールが，現在の名前空間に輸入される。なぜなら，それらは，`from...import` 文が実行される時，`Sound.Effects` パッケージの中で定義されているからである。(これは `__all__` が定義されているときにも働く)。

モジュールやパッケージから * を輸入するというやり方は，しばしば読みにくいコードをもたらすため，一般的には，まゆをひそめるものであることに注意しよう。しかし，対話セッションで打鍵量を節約するために利用することは OK だし，モジュールによっては特定のパターンに従った名前だけを輸出するように設計されているものもある。

「`from` パッケージ `import` 特定の下位モジュール」を使うことに何も不適切さはない，ということに留意しよう！　実際，これは推奨される記法である。ただし，輸入をするモジュールが，別々のパッケージからの同じ名前の下位モジュールを使わなくてはならないときは，この限りではない。

6.4.2　パッケージ内での参照

下位モジュールどうしが互いに参照しあう必要がしばしばある。たとえば，`surround` モジュールが `echo` モジュールを使うかもしれない。実際，このような参照はとてもよくあることなので，`import` 文は，標準のモジュール検索パスを見る前に，まず取り囲んでいるパッケージを見る。こうして，surround モジュールは単純に `import echo` とか `from echo import echofilter` を使うことができる。もしも輸入されるモジュールが現在のパッケージ (現在のモジュールを，下位モジュールとしているパッケージ) に見つからなければ，`import` 文は，与えられた名前をもつトップ=レベルのモジュールを探す。

(例の中の Sound パッケージのように) パッケージが下位パッケージへと構造化されているとき，兄弟パッケージの下位モジュールを参照する短縮記法はない — 下位パッケージのフル・ネームを使わなければならない。たとえば，もしもモジュール `Sound.Filters.vocoder` が，`Sound.Effects` パッケージの `echo` モジュールを使う必要があるならば，`from Sound.Effects import echo` を使うことはできる。

CHAPTER

SEVEN

入力と出力

プログラムの出力を現す方法はいくつもある。データは，人間可読な形式で印字することも，あるいは将来の使用のためにファイルに書き込むこともできる。この章はいくつかの可能性を議論する。

7.1　よりファンシーな出力の書式化

これまでのところ私たちは値を書く二つの方法，**式文** *(expression statement)* と `print` 文を見てきた。(もう一つの方法はファイル・オブジェクトの `write()` メソッドを使う方法である。標準出力ファイルは `sys.stdout` として参照できる。これに関する詳細については Library Reference を見られたい)

しばしば，君は出力の書式化に対し，単純に値をスペースで区切って印字する以上の制御が欲しくなることだろう。出力を書式化する方法は二つある。第一の方法は，君自身で文字列の処理をすべて行う方法である。文字列のスライスと連結の演算を使えば，想像し得る限りのレイアウトを造りだせる。標準モジュール `string` には，与えられた桁幅に文字列をそろえる便利な演算がある。これらを簡単に説明する。第二の方法は，`%` 演算子を，文字列を左項として使う方法である。`%` 演算子は左項を `sprintf()` スタイルの書式文字列のように解釈して右項に適用し，その書式化演算から得られた文字列を返す。

もちろん，一つの問題が残っている。どうやって値を文字列へ変換するのだろうか?　幸い Python にはどんな値でも文字列へ変換する方法がある。値を `repr()` 関数に渡すか，あるいはただ逆クォート (‘‘) ではさんで書けばよい。いくつか例を挙げる。

```
>>> x = 10 * 3.14
>>> y = 200*200
>>> s = 'The value of x is ' + `x` + ', and y is ' + `y` + '...'
>>> print s
The value of x is 31.4, and y is 40000...
>>> #  逆クォートは数以外の型にも働く:
... p = [x, y]
>>> ps = repr(p)
>>> ps
'[31.400000000000002, 40000]'
>>> # 文字列を変換するとクォートとバックスラッシュが付加される:
... hello = 'hello, world\n'
>>> hellos = `hello`
>>> print hellos
'hello, world\n'
>>> # 逆クォートの引数はタプルでもよい:
... `x, y, ('spam', 'eggs')`
"(31.400000000000002, 40000, ('spam', 'eggs'))"
```

下記は２乗と３乗の表を書く二つの方法である。

```
>>> import string
>>> for x in range(1, 11):
...     print string.rjust(‘x‘, 2), string.rjust(‘x*x‘, 3),
...     # 上の行の末尾のコンマに注意
...     print string.rjust(‘x*x*x‘, 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
```

(print が印字しているとき，コラム間に 1 個ずつスペースが加えられたことに注意しよう。print はつねに引数間にスペースを加える)

この例は関数 string.rjust() の使い方を示している。この関数は，文字列の左側にスペースを詰めることによって，文字列を指定された幅のフィールドに右ぞろえ (right-justify) する。似たような関数に string.ljust() と string.center() がある。これらの関数は何も書かず，ただ新しい文字列を返す。もしも入力文字列が長すぎるときは，切り詰めずにそのまま返す。これは君のコラム・レイアウトをメチャクチャにするが，普通は，切り詰めてウソの値を書くよりもましだ (もし本当に切り詰めたいのならば，‘string.ljust(x, n)[0:n]’) のようにつねにスライス演算を加えればよい)。

もう一つの関数 string.zfill() は，数値文字列の左側をゼロで詰める。これは正と負の符号を正しく扱う。

```
>>> import string
>>> string.zfill(’12’, 5)
’00012’
>>> string.zfill(’-3.14’, 7)
’-003.14’
>>> string.zfill(’3.14159265359’, 5)
’3.14159265359’
```

% 演算子を使うとこのようになる。

```
>>> 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ときは，右オペランドとしてタプルを渡す。たとえば，

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

ほとんどの書式は正確に C と同じように働く。君は書式に正しい型を渡す必要がある — しかし，そうしなかったからといって例外を得るだけで，コア・ダンプはしない。書式 `%s` はもっと寛大だ。つまり，もしも対応する引数が文字列オブジェクトでなければ，組込み関数 `str()` を使って文字列へと変換する。`*` を使って幅や精度を別個の (整数) 引数として渡すことはサポートされている。C の書式 `%n` と `%p` はサポートされていない。

もしも本当に長い書式文字列があって，それを分割したくないとき，書式化すべき変数を位置ではなく名前で参照できたらナイスだろう。`%(name)format` という形式を用いる C 書式の拡張を使うことによって，それが可能である。たとえば，

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

これは，すべてのローカル変数を収めた辞書を返す新しい組込み関数 `vars()` と組み合わせると特に便利である。

7.2 ファイルを読み書きする

`open()` はファイル・オブジェクトを返す。`open()` は普通，'`open(`*filename*`, `*mode*`)`' のように 2 引数で使われる。

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

第 1 引数はファイル名からなる文字列である。第 2 引数も文字列であり，ファイルの使われ方を記述する少数の文字からなる。*mode* はファイルが読まれるだけならば `'r'` でよい。書かれるだけならば `'w'` でよい (同名の既存ファイルは消去される)。`'a'` はファイルを追加書込みのために開くから，そのファイルに書かれるデータは自動的に終端に追加される。`'r+'` はファイルを読み書き両方のために開く。*mode* 引数は省略可能であり，省略されたときは `'r'` が仮定される。

Windows と Macintosh では，*mode* に `'b'` を追加すると，ファイルをバイナリ・モードで開く。したがって `'rb'`, `'wb'`, `'r+b'` のようなモードもある。Windows はテキスト・ファイルとバイナリ・ファイルを区別しており，テキスト・ファイルの行末文字は，データが読み書きされるとき，自動的にわずかばかり改変される。ファイル・データへのこの暗黙裏の改変は，ASCII テキスト・ファイルには差しつかえないが，JPEG や '.EXE' ファイルのようなバイナリ・データをダメにしてしまう。このようなファイルを読み書きするときは，バイナリ・モードを使うように十分気を付けられたい。(Macintosh でのテキスト・モードの正確な意味論は，使用される基底の C ライブラリに依存することに注意されたい)

7.2.1 ファイル・オブジェクトのメソッド

この節のこれ以降の例は，`f` というファイル・オブジェクトが既に造られていることを仮定する。

ファイルの内容を読み取るには，`f.read(`*size*`)` を呼び出す。これは，ある量のデータを読み取り，それを文字列として返す。*size* は省略可能な数値引数である。*size* が省略されるかまたは負数ならば，ファイルの

全内容が読み取られて返される — もしもファイルが君のマシンのメモリの 2 倍の大きさでも，それは君の問題だ。*size* が非負数ならば，高々 *size* バイトだけ読み取られて返される。もし既にファイルの終端に達しているならば，`f.read()` は空文字列 (`""`) を返す。

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

`f.readline()` はファイルから 1 行だけ読み取る。このとき，改行文字 (`\n`) が文字列の終端に残される。改行文字がないのは，ファイルが改行で終わらないときの，ファイルの最終行だけである。これは戻り値からあいまいさをなくす。つまり，もしも `f.readline()` が空文字列を返したならば，ファイルの終端に達している。しかるに空行は `'\n'` — つまり 1 個の改行だけからなる文字列 — で表現される。

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

`f.readlines()` は，ファイルの中のデータからなる行すべてを含んだリストを返す。もしも省略可能な引数 *sizehint* が与えられたならば，ファイルからそれだけのバイト数を読み取り，そしてさらに 1 行を完成させるに十分なだけ読み取って，それらから行のリストを作って返す。これはしばしば，大きなファイル全体をメモリにロードすることなく，ファイルを行ごとに効率よく読み取るために使われる。未完成の行が返されることはない。

```
>>> 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` を返す。

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

`f.tell()` はファイルの中でのファイル・オブジェクトの現在の位置を与える整数を返す。これはファイルの先頭からバイトで測った数である。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*`)`’ を使う。位置は，参照点に *offset* を足して計算される。参照点は *from_what* 引数によって選択される。*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う。*from_what* は省略可能である。省略時の値は 0 であり，ファイルの先頭を参照点として使う。

```
>>> f=open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)     # ファイルの第 5 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'
```

ファイルでの用が済んだら，`f.close()` を呼び出してファイルを閉じ，開いていたファイルに取られていたシステム資源を解放する。`f.close()` を呼び出した後は，そのファイル・オブジェクトを使おうとしても自動的に失敗することになる。

```
>>> 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，それほど頻繁には使われないが，`isatty()` や `truncate()` など，もういくつかのメソッドがある。ファイル・オブジェクトの完全なガイドについては Library Reference を調べられたい。

7.2.2 `pickle` モジュール

文字列はたやすくファイルから読み書きできる。数は少々骨が折れる。なぜなら `read()` メソッドが文字列しか返さないから，`'123'` のような文字列を受け取ってその数値 123 を返す `string.atoi()` のような関数にその文字列を渡してやらなければならないからである。しかし，リストや辞書やクラス・インスタンスのようなもっと複雑なデータ型をセーブしようと思ったら，ことはもっと複雑になる。

Python は，複雑なデータ型をセーブするコードを利用者に毎回毎回書かせてデバッグさせるようなことはしない。そのかわりに `pickle` という標準モジュールを用意している。これはほとんどどんな Python オブジェクトをも (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るという驚嘆すべきモジュールである。この変換過程は *pickling* (漬け物化) と呼ばれる。文字列表現からのオブジェクトの再構成は *unpickling* と呼ばれる。pickling と unpickling のあいだ，その文字列表現を，ファイルやデータに格納したり，ネットワーク接続を越えて遠くのマシンに送ったりできる。

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` があると仮定して，オブジェクトを pickling する最も簡単な方法はたった 1 行のコードで済む。

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

オブジェクトを再び unpickling するには，`f` が読取り用に開かれているファイル・オブジェクトであると仮定して，

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

(これにはいくつか別の方法がある。それらは多数のオブジェクトを pickling したり，pickling したデータをファイルへ書き込みたくないときに使われる。`pickle` の完全なドキュメンテーションについては Library Reference を調べられたい)

`pickle` は，ほかのプログラムや将来起動したときの自プログラムから再利用できるように格納しておける Python オブジェクトを作るための，標準的な方法である。これを表す技術用語は**永続**オブジェクト (persistent object) である。`pickle` はとても広く使われているから，Python 拡張を書く多くの著者は，行列などの新しいデータ型が正しく pickling/unpickling できるように気をつけている。

CHAPTER

EIGHT

エラーと例外

今までエラー・メッセージについてはせいぜい言及するだけだったが，もしも君が例を試してきたならば，おそらくそのいくつかを見ていることだろう。エラーは (少なくとも) 二つに大別できる。それは**構文エラー** *(syntax error)* と**例外** *(exception)* である。

8.1 構文エラー

構文エラーは構文解析エラー (parsing error) としても知られる。これはもしかすると君が Python を学んでいるあいだに受ける苦情のうち最もありふれた種類のものかもしれない。

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

パーサは構文違反の行を繰り返し，その行の中でエラーが検出された最初の位置を指す小さな「矢印」を表示する。エラーは矢印の**直前**のトークンでひき起こされている (か，少なくともそこで検出されている)。この例ではエラーはキーワード `print` で検出されている。これはコロン (':') がその前に無いからである。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印字される。

8.2 例外

文や式が構文的に正しくても，それを実行しようとした時にエラーが起こるかもしれない。実行中に検出されるエラーは**例外**と呼ばれる。例外は無条件に致命的なわけではない。君はすぐ後で Python プログラム内で例外を処理する方法を学ぶことになる。とはいっても，たいていの例外はプログラムで処理されず，ここに示すようなエラー・メッセージに終わる。

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1
ZeroDivisionError: integer division or modulo
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1
NameError: spam
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1
TypeError: illegal argument type for built-in operation
```

エラー・メッセージの最後の行は，何が起こったかを示している。例外はさまざまな型 (type) で到来し，その型はメッセージの一部として印字される。例で挙げられている型は `ZeroDivisionError`, `NameError`, `TypeError` である。例外型として印字される文字列は，発生した例外を表す組込み名の名前である。これはすべての組込み例外について成り立つが，利用者定義の例外については必ずしも成り立たない (しかし，有用な慣習である)。標準例外名は組込み識別子である (予約語ではない)。

行の残りは詳細説明であり，その解釈は例外型に依存する。その意味は例外型によって決まる。

それより前の部分のエラー・メッセージは，例外が起こった文脈を，スタック・バックトレース (stack backtrace) の形式で示している。一般にこれはソース行をリストするスタック・バックトレースからなるが，標準入力から読み取られた行は表示されない。

Python Library Reference は組込み例外とその意味をリストしている。

8.3 例外を処理する

例外を選んで処理するプログラムを書くことができる。次の例を見よう。これは正しく整数が入れられるまで利用者に入力を求める。ただし，利用者が (`Control-C` または何であれオペレーティング・システムがサポートしているキーを使って) プログラムを中断することもできる。利用者が発生させた中断は，`KeyboardInterrupt` 例外のひき起こしによって合図されることに注意しよう。

```
>>> while 1:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops! That was no valid number.  Try again..."
...
```

`try` 文は下記のように働く。

- まず *try* **節** (*try clause* つまりキーワード `try` と `except` のあいだの文) が実行される。
- もしも何も例外が発生しなければ，*except* **節**をスキップして `try` 文の実行を終える。
- もしも例外が try 節の実行中に発生すれば，その節の残りはスキップされる。それから，もしも例外型が，`except` キーワードの後に名前が挙げられている例外とマッチするならば，try 節の残りをスキップしてその except 節が実行され，そしてそれから `try` 文の後へと実行が継続される。
- もしも except 節で名前が挙げられている例外とマッチしない例外が発生したならば，その例外は `try` 文の外側へ渡される。もしもハンドラがどこにも見つからなければ，それは**処理されない例外** *(unhandled exception)* であり，上記に示したようなメッセージとともに実行が停止する。

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してもよい。たかだか一つのハンドラが実行される。ハンドラは，対応する try 節の中で発生した例外を処理するだけであり，

同じ `try` 文のほかのハンドラで発生した例外は処理しない。一つの except 節で，複数の例外を，丸かっこで囲んだ並びとして指定してもよい。たとえば，

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

最後の except 節は例外名を省いてもよい。これはワイルドカードとして働く。これは極度に注意して使おう。なぜなら，こうすると本物のプログラミング・エラーがたやすく隠されてしまうからだ! これは，エラー・メッセージを印字して，そしてそれから (呼出し元が同様に例外を処理できるように) 例外を再発生させることにも使える。たとえば，

```
import string, sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(string.strip(s))
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 *else* **節** *(else clause)* を設けてもよい。もし設けるとすれば，すべての except 節の次でなければならない。これは，try 節が例外をひき起こさなかったときに実行しなければならないコードとして役立つ。たとえば，

```
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()
```

`else` 節を使うほうが，`try` 節にコードを追加するよりも良い。なぜなら，`try` ... `except` 文で保護されているコードによってひき起こされたのではない例外をたまたま捕獲してしまう，ということを避けられるからだ。

例外が発生した時，その例外には一つの値，例外の**引数** *(argument)* としても知られる値が，結合しているかもしれない。引数の有無とその型は，例外型によって決まる。引数をもつ例外型のために，except 節は，引数値を受けとる変数を，例外の名前 (またはその並び) の後に，下記のように指定してもよい。

```
>>> try:
...     spam()
... except NameError, x:
...     print 'name', x, 'undefined'
...
name spam undefined
```

もしも例外に引数があれば，それは，処理されない例外に対するメッセージの最後の部分 ('詳細説明') として印字される。

例外ハンドラは，try 節でじかに発生した例外だけではなく，その try 節から (間接的にであれ) 呼び出された関数の内部で発生した例外も処理する。たとえば,

```
>>> 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 例外をひき起こす

raise 文は，プログラマが，指定した例外を強制的に発生させることを可能にしている。たとえば,

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

raise の第 1 引数は，ひき起こすべき例外を指名する。省略可能な第 2 引数は，例外の引数を指定する。

8.5 利用者定義の例外

プログラムは，文字列を変数へ代入したり，新しい例外クラスを造ることによって，それ独自の例外を指名してよい。たとえば,

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

モジュールが定義する関数の中で発生するかもしれないエラーを報告するために，多くの標準モジュールがこれを使っている。

クラスについてのもっと多くの情報は第 9 章 “クラス” にある。

8.6 後片付け動作を定義する

`try` 文にはもう一つ，いかなる状況下でも実行しなくてはならない後片付け動作 (clean-up action) の定義を目的とする選択可能な節がある。たとえば，

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Goodbye, world!'
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2
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る。例外が発生した時は，finally 節を実行した後，再びその例外がひき起こされる。`try` 文から `break` 文や `return` 文経由で脱出する時にも，そこを “出る途中で” finally 節が実行される。

`try` 文には，１個以上の except 節か，または１個の finally 節が必要だが，両方あってはならない。

CHAPTER

NINE

クラス

Python のクラスのからくりは，最小限の新しい構文と意味論をもって言語にクラスを追加している。それは C++ と Modula-3 に見られるクラスのからくりの混合である。モジュールがそうであるように，Python のクラスは定義と利用者のあいだに絶対的な障壁を置くことはせず，むしろ “定義内部に乱入する” ことはしないという利用者の礼儀正しさに頼っている。それでも，クラスの最も重要な機能は，強力さを損なうことなく保たれている。すなわち，クラス継承のからくりは多重の基底クラスを許しており，派生クラスはその基底クラスのどのメソッドも上書きする (override) ことができ，メソッド呼出しは基底クラスの同名のメソッドを呼び出すことができる。オブジェクトは任意の量の私有データを持つことができる。

C++ の用語でいえば，クラス・メンバは (データ・メンバも含め) すべて**公開** *(public)* であり，メンバ関数はすべて**仮想** *(virtual)* である。特別なコンストラクタやデストラクタはない。Modula-3 と同じく，オブジェクトのメンバをそのメソッドから参照するための短縮記法はない。すなわち，メソッド関数はそのオブジェクトをあらわす第１引数を陽に伴って宣言される。この第１引数は呼出しによって自動的に供給される。言葉の広い意味においてではあるが，Smalltalk と同じく，クラスはそれ自身がオブジェクトである。さらにいえば，Python では，すべてのデータ型がオブジェクトである。このことが輸入 (importing) と改名 (renaming) の意味論を規定している。しかし，C++ や Modula-3 と全く同じく，組込み型を基底クラスとして使って利用者が拡張することはできない。また，C++ と同じく，Modula-3 とは違って，(算術演算子や添字付けなど) 特別な構文をもつ組込み演算子の大多数をクラス・インスタンスのために再定義することができる。

9.1 用語について一言

クラスについて語るための普遍的に受け入れられている用語がないので，私は Smalltalk と C++ の用語を場合に応じて使っていくことにする (オブジェクト指向の意味論は C++ よりも Modula-3 のほうが Python に近いから，Modula-3 の用語を使いたかったが，ほとんどの読者はそれを耳にしたことがないだろうと思う)。

私はまた，オブジェクト指向派の読者にとって用語上の落し穴があることを，君に警告しなくてならない。すなわち，“オブジェクト” という言葉は Python では必ずしもクラスのインスタンスを意味しない。C++ や Modula-3 と同じく，Smalltalk とは違って，Python ではすべての型がクラスだというわけではない。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ない。それでも，Python の**すべての**型は，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論を共有している。

オブジェクトは個体として存在しており，(別々のスコープで) 別々の名前が同一のオブジェクトへと束縛され得る。これは，ほかの言語では別名 (aliasing) として知られている。そのありがたみは普通は Python をひとめ見ただけでは分からないし，変化不可能な基本型 (数，文字列，タプル) を扱うときには無視しても差し支えない。しかし，別名には，変化可能オブジェクト — リストや，辞書や，プログラム外部の実体 (ファイルやウィンドウなど) を表現するほとんどの型 — に関係する Python コードの意味論に関して，ある (意図的な!) 効果がある。これは普通はプログラムのためになるように使われる。なぜなら別名は，ある意味，ポインタのように振舞うからである。たとえば，オブジェクトの受渡しは，実装ではポインタが渡されるだけだから安価である。そして，もし関数が引数として渡されたオブジェクトを改変したならば，呼出した側からその変化が見られることになる — これは Pascal にあるような二つの異なった引数渡しのからくりの必要性をなくしている。

9.2 Python のスコープと名前空間

クラスを紹介する前に，私はまず Python のスコープ規則について君になにがしか話さなくてはならない。クラス定義はある巧みなトリックを名前空間に対して演じるから，何が起こっているのかを完全に理解するには，スコープと名前空間の働きかたを知る必要がある。ついでながら，この件についての知識はあらゆる上級 Python プログラマの役に立つ。

まず定義から始めよう。

名前空間 *(namespace)* とは，名前からオブジェクトへの写像 (mapping) である。ほとんどの名前空間は今のところ Python の辞書として実装されているが，そのことは通常どのみち (性能を除き) 重要ではないし，将来は変更されるかもしれない。名前空間の例としては，組込み名の集合 (`abs()` 等の関数や組込み例外名)，１モジュールにおけるグローバル名の集合，１関数呼出しにおけるローカル名の集合がある。ある意味では１オブジェクトの属性 (attribute) の集合も１名前空間を形成している。名前空間について知っておくべき重要なことは，別々の名前空間にある名前どうしは絶対的に無関係だ，ということだ。たとえば，二つの別々のモジュールが，混同することなく，ともに “maximize” という関数を定義してもよい — モジュールの利用者はそれにモジュール名を接頭しなくてはならない。

ところで，私は**属性**という言葉を，なんであれドットに続く名前に対して使っている — たとえば式 `z.real` で，`real` はオブジェクト `z` の属性である。厳密にいえば，モジュールの中にある名前を参照することは，属性参照である。つまり，式 `modname.funcname` で，`modname` は１モジュール・オブジェクトであり，`funcname` はその１属性である。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あいだに，素直な写像がある。すなわち，それらは同じ名前空間を共有している![1]

属性は読取り専用かもしれないし書込み可能かもしれない。後者の場合，属性への代入が可能である。モジュール属性は書込み可能である。たとえば，君は ‘`modname.the_answer = 42`’ と書くことができる。書込み可能属性は，たとえば ‘`del modname.the_answer`’ のように，`del` 文で削除することもできる，

名前空間が造られる時 (moment) と寿命 (lifetime) はさまざまである。組込み名を収めている名前空間は，Python インタープリタが起動する時に造られ，そして決して削除されない。モジュールのためのグローバル名前空間は，モジュール定義が読み込まれた時に造られ，そして通常これもインタープリタが終わるまで存続する。インタープリタのトップ=レベル呼出しによって実行される文は，スクリプト・ファイルから読まれたにせよ，対話的に読まれたにせよ，`__main__` という１モジュールの一部であると見なされる。だから，それらにもそれら自身のグローバル名前空間がある。(組込み名も，実際には `__builtin__` という１モジュールの中に住んでいる)

関数のためのローカル名前空間は，関数が呼び出された時に造られ，そして関数から戻るか，またはその関数内で処理されない例外をひき起した時に削除される (実際には，忘れられる，と言ったほうが実態に近い)。もちろん，再帰呼出しのときも，それぞれの呼出しごとにローカル名前空間がある。

スコープ *(scope)* とは，ある名前空間を直接アクセス可能な，Python プログラムの字面上の領域 (textual region) である。ここで，ある名前空間を “直接アクセス可能” (directly accessible) とは，接頭辞なしに名前を参照したとき，その名前空間で名前の検索が試みられることを意味する。

スコープは静的に決定されるが，動的に使用される。実行中はいつでも，きっかり３個の入れ子になった名前空間が使用される (つまり，きっかり３個の名前空間が直接アクセス可能である)。すなわち，最初に検索される最内スコープがローカル名を収め，次に検索される中間スコープが現在のモジュールのグローバル名を収め，そして (最後に検索される) 最外スコープが組込み名を収めている名前空間である。

普通，ローカル・スコープは現在の関数—字面的に—のローカル名を参照する。関数の外側では，ローカル・スコープはグローバル・スコープと同じ名前空間，すなわちモジュールの名前空間を参照する。クラス定義はローカル・スコープの中にもう一つの名前空間を設ける。

重要なのは，スコープは字面で決定される，ということをはっきり理解することだ。たとえば，あるモジュールの中で定義された関数のグローバル・スコープは，たとえその関数がどこから，またはどんな別名によって，呼び出されようとも，そのモジュールの名前空間である。他方，実際の名前検索は，実行時に，動的に行われる — しかし，言語の定義は，“コンパイル” 時の，静的な名前解決へ向けて進化しているから，動的な名前解決に頼ってはならない! (事実，ローカル変数は既に静的に決定されている)

[1] 一つ例外がある。モジュール・オブジェクトには，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す秘密の読取り専用属性 `__dict__` がある。`__dict__` という名前は属性だがグローバル名ではない。明らかに，これの利用は，名前空間の実装の抽象化を侵しており，したがって検死デバッガのようなものに限られるべきである。

Python の特別なクセとして，代入はいつも最内スコープに入る。代入はデータをコピーしない — ただ名前をオブジェクトへ束縛する。同じことが削除にもあてはまる。たとえば，文 'del x' は，x の束縛を，ローカル・スコープが参照する名前空間から削除する。実際，新しい名前を導入する演算はすべてローカル・スコープを用いる。とりわけ，import 文と関数定義は，モジュールや関数名を，ローカル・スコープで束縛する。(global 文を使えば，ある変数がグローバル・スコープに住んでいることを指示できる)

9.3 クラス初見

クラスはちょっとした新しい構文，三つの新しいオブジェクト型，そしていくつかの新しい意味論を導入している。

9.3.1 クラス定義の構文

最も簡単な形式のクラス定義はこんな形をしている。

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

クラス定義は，関数定義 (def 文) と同じく，実行されて初めて効果がある。(if 文の分岐や関数内部にクラス定義を置くことも，考えられる限りではあり得る)

実際には，クラス定義内部の文は普通は関数定義だろう。しかし，ほかの文であってもよいし，それが役に立つこともある — これについては後述しよう。クラス内部の関数定義は通常は，メソッドの呼出し規約に従った，ある特定の形式の引数並びをもつ — これもまた後述する。

クラス定義へ進入する時，新しい名前空間が造られ，ローカル・スコープとして使われる — したがって，ローカル変数への代入はすべてこの新しい名前空間へ入る。とりわけ，関数定義は新しい関数の名前をここで束縛する。

クラス定義から (終端を経由して) 正常に退出する時，**クラス・オブジェクト** *(class object)* が造られる。これは基本的には，クラス定義が造った名前空間の内容をくるんだラッパー (wrapper) である。クラス・オブジェクトについては次の節でさらに学ぶことにしよう。元々のローカル・スコープ (クラス定義へ進入する直前まで有効だったスコープ) が復帰し，そしてこのスコープにおいて，クラス・オブジェクトが，クラス定義ヘッダで与えられた名前 (上の例でいえば ClassName) へ束縛される。

9.3.2 クラス・オブジェクト

クラス・オブジェクトは 2 種類の演算 — 属性参照とインスタンス生成 — をサポートしている。

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 を使う。妥当な属性名は，クラス・オブジェクトが造られた時にクラスの名前空間にあった名前すべてである。だから，もしも次のようなクラス定義ならば，

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

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・オブジェクトを返す。クラス属性へ代入してもよい。だから君は MyClass.i の値を代入によって変えられる。__doc__ も妥当

な属性であり，そのクラスに属している docstring，この場合は `"A simple example class"` を返す。

クラスの**インスタンス生成** *(instantiation)* は関数記法を使う。クラス・オブジェクトのことを，クラスの新しいインスタンスを返す無引数関数だと思い込めばよい。たとえば (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` へ代入する。

インスタンス生成演算 (クラス・オブジェクトの “呼出し”) は，空のオブジェクト (empty object) を造る。多くのクラスでは既知の初期状態にオブジェクトを造ることが望ましい。したがってクラスは `__init__()` という名前の特別なメソッドを，このように定義してよい。

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

クラスが `__init__()` メソッドを定義しているとき，クラスのインスタンス生成は，新しく造られたクラス・インスタンスに対して自動的に `__init__()` を呼び出す。だからこの例では，新しい，初期化済みのインスタンスが下記によって得られる。

```
x = MyClass()
```

もちろん，より大きな柔軟性を求めて `__init__()` メソッドに複数の引数をもたせてもよい。その場合，クラスのインスタンス生成演算子に与えられた引数が `__init__()` へ渡される。たとえば,

```
>>> 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

ところで私たちはインスタンス・オブジェクトで何ができるのだろうか? インスタンス・オブジェクトが理解する唯一の演算は属性参照である。妥当な属性名は 2 種類ある。

第 1 の種類を**データ属性** *(data attribute)* と呼ぶことにする。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) にあたる。データ属性を宣言する必要はない。ローカル変数と同じく，これらは最初に代入された時点で突然存在し始める。たとえば，上記で造った `MyClass` のインスタンス `x` に対し，下記のコード片は，痕跡を残すことなく，値 `16` を印字するだろう。

```
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する第 2 の種類の属性参照は**メソッド** *(method)* である。メソッドとは，オブジェクトに “属している” 関数 (a function that “belongs to” an object) である。(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ない。ほかのオブジェクト型にもメソッドはあり得る。

たとえば，リスト・オブジェクトには append, insert, remove, sort などのメソッドがある。しかし以下では，特に明記しない限り，メソッドという用語をクラス・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する)

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る。定義により，(利用者定義の) 関数オブジェクトであるクラス属性すべてが，そのインスタンスの対応するメソッドを定義する。私たちの例でいえば，`MyClass.f` が関数だから `x.f` は妥当なメソッド参照である。しかし，`MyClass.i` は関数ではないから `x.i` は妥当なメソッド参照ではない。しかし `x.f` は `MyClass.f` と同じものではない — それは関数オブジェクトではなく，**メソッド・オブジェクト** *(method object)* である。

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る。たとえば，

```
x.f()
```

私たちの例では，これは文字列 `'hello world'` を返すだろう。しかし，必ずしもメソッドをすぐに呼び出す必要はない。`x.f` はメソッド・オブジェクトであり，どこかに格納しておいて後から呼び出すこともできる。たとえば，

```
xf = x.f
while 1:
    print xf()
```

は ‘`hello world`’ を時の終わりまで印字し続けるだろう。

メソッドが呼び出される時，正確には何が起こるのだろうか? `f` の関数定義は 1 個の引数を指定していたのにもかかわらず，上記で `x.f()` が無引数で呼び出されたことに君は気付いているかもしれない。引数はどうしたのだろうか? たしか，引数が必要な関数を無引数で呼び出すと，Python が例外をひき起こすはずなのに — たとえその引数が実際には使われなくても…。

実際，君はもう答を解き当てているもしれない。メソッドについて特別なことは，オブジェクトが関数の第 1 引数として渡される，ということだ。私たちの例では，呼出し `x.f()` は `MyClass.f(x)` と正確に等価である。一般に，*n* 引数の並びをもったメソッド呼出しは，対応する関数を，そのメソッドのオブジェクトを第 1 引数の前に挿入して造った引数並びとともに呼び出すことと等価である。

もしもまだメソッドの働きかたを理解できないならば，実装を見てみると事情がよく分かるかもしれない。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る。もしもその名前が，関数オブジェクトである妥当なクラス属性を表しているならば，メソッド・オブジェクトが造られる。その方法は，インスタンス・オブジェクトと，今しがた見つけた関数オブジェクト (のそれぞれを指すポインタ) を，一つの抽象オブジェクトにパック (pack) する，という方法である。この抽象オブジェクトがすなわちメソッド・オブジェクトである。メソッド・オブジェクトが引数並びとともに呼び出された時は，それが再びアンパック (unpack) されて，インスタンス・オブジェクトと元々の引数並びから新しい引数並びが構築される。そして関数オブジェクトがこの新しい引数並びとともに呼び出される。

9.4 いろいろな注意点

[これらはおそらくもっと注意深く配置すべきだろう…]

データ属性は同じ名前のメソッド属性を上書き (override) する。予期しない名前の衝突は，大規模なプログラムにおいて見つけにくいバグの原因となり得る。これを避けるため，衝突の機会を最小にするなんらかの種類の規約，たとえば，メソッド名を大文字で始める，データ属性名に短い一意的な文字列 (あるいはただの下線) を接頭する，またはメソッドには動詞をデータ属性には名詞を用いる，などの規約を用いるのが賢明である。

データ属性は，メソッドからだけではなく，オブジェクトの一般利用者 (“クライアント”) からも同様に参

照できる。言い換えれば，クラスを使って純粋な抽象データ型を実装することはできない。実際，Python にはデータ隠蔽の強制的な順守を可能にするものは何もない — データ隠蔽はすべて規約にもとづいている。(他方，C で書かれた Python 実装は，実装上の詳細を完全に隠すこと，そしてもし必要ならばオブジェクトへのアクセスを制御することができる。C で書かれた Python への拡張はこのことを利用してよい)

クライアントはデータ属性を注意して使うべきである — クライアントは，データ属性を踏みにじることによって，メソッドが維持している不変式を台無しにすることもできる。名前の衝突が回避されている限り，クライアントは，メソッドの妥当性に影響を与えることなく独自のデータ属性をインスタンス・オブジェクトに追加してもよいことに注意しよう — ここでもやはり，名前付けの規約は頭痛の種をなくすのに役立つ。

メソッド内部からデータ属性を (または，ほかのメソッドを!) 参照するための短縮記法はない。私の見るところ，これは実際にはメソッドの可読性を改善している。メソッドに目を通している時に，ローカル変数とインスタンス変数を混同する機会は皆無である。

慣習上，メソッドの第 1 引数はしばしば `self` と呼ばれる。これは単なる規約以外の何ものでもない。すなわち，`self` という名前は Python にとって断じてなんら特別な意味はない。(しかし，この規約に従わなければ，君のコードはほかの Python プログラマにとって読みにくいものになるかもしれない。それに，この規約にもとづいた**クラス・ブラウザ** *(class browser)* プログラムが書かれないとも限らない)

クラス属性である関数オブジェクトはどれも，そのクラスのインスタンスのためのメソッドを定義する。関数定義が字面の上でクラス定義に取り囲まれている必要はない。関数オブジェクトをクラスのローカル変数へ代入するだけでも ok だ。たとえば，

```
# 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
```

このとき `f` と `g` と `h` は，すべて関数オブジェクトを参照するクラス `C` の属性であり，したがってすべて `C` のインスタンスのメソッドである — `h` は `g` と正確に等価である。ただし，これを実践しても，普通はプログラムの読者を混乱させるのに役立つだけだ。

メソッドは，`self` 引数のメソッド属性を使うことによって，ほかのメソッドを呼び出せる。たとえば，

```
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，通常の関数と同じようにグローバル名を参照できる。メソッドに結合されているグローバル・スコープは，クラス定義を収めているモジュールである。(クラスそれ自身がグローバル・スコープとして使われることはない!)　メソッドでのグローバル・データの利用に賛成する良い理由に出会うことは滅多にないが，それでもなおグローバル・スコープには多くの正当な用途がある。たとえば，グローバル・スコープへ輸入された関数とモジュールが，同スコープで定義される関数やクラスからはもちろん，メソッドからも利用可能になる。普通は，メソッドを収めているクラスもそれ自身このグローバル・スコープで定義される。次の節では，メソッドが自分のクラスを参照しようとする良い理由を見てみよう!

9.5 継承

いうまでもなく，継承 (inheritance) をサポートしないような言語機能は，“クラス” という名前に値しない。派生クラス (derived class) を定義する構文は次のような形をしている。

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

基底クラス (base class) の名前 `BaseClassName` は，派生クラス定義を収めているスコープで定義されていなければならない。基底クラス名のかわりに式でもよい。これは基底クラスが別モジュールで定義されているとき役に立つ。たとえば，

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

派生クラス定義の実行は，基底クラスのときと同じように進行する。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る。これは属性参照を解決するために使われる。すなわち，もしも要求された属性がクラスに見つからなければ，基底クラスが検索される。もしも基底クラスそれ自身がほかのクラスから派生しているならば，この規則が再帰的に適用される。

派生クラスのインスタンス生成について特別なことは何もない— `DerivedClassName()` がクラスの新しいインスタンスを造る。メソッド参照は次のように解決される— 必要に応じて基底クラスの連鎖を下りつつ，対応するクラス属性が検索され，そしてその結果が関数オブジェクトならば，そのメソッド参照は妥当である。

派生クラスは基底クラスのメソッドを上書きしてもよい。メソッドが同じオブジェクトの別メソッドを呼び出す時に何も特権はないから，基底クラスのメソッドが同じ基底クラスで定義された別メソッドを呼び出す時，実際にはそれを上書きした派生クラスのメソッドを呼び出すことになるかもしれない。(C++ プログラマへ: Python ではすべてのメソッドが事実上 `virtual` である)

派生クラスで上書きしているメソッドは，実際には，単純に同名の基底クラス・メソッドに取って代わりたいのではなく，むしろそれを拡張したいのかもしれない。基底クラス・メソッドをじかに呼び出す簡単な方法がある。‘`BaseClassName.methodname(self, 引数)`’ を呼び出せばよい。これは時にはクライアントにも役に立つ。(これが働くのは基底クラスがじかにグローバル・スコープで定義または輸入されているときだけであることに注意しよう)

9.5.1 多重継承

Python は限られた形式の多重継承 (multiple inheritance) もサポートしている。多重の基底クラスを伴ったクラス定義は次のような形をしている。

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

意味論を説明するために必要な唯一の規則は，クラス属性参照に使われる解決規則 (resolution rule) である。これは深さ優先 (depth-first)，左から右へ (left-to-right) である。したがって，もしも属性が `DerivedClassName` に見つからなければ，`Base1` で検索され，それから (再帰的に) `Base1` の基底クラスで検索される。そしてそこに見つからないときに限り `Base2` で検索される，等々である。

(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る。しかし，もしもそうすると，`Base1` のある 1 属性が実際に `Base1` で定義されているのかそれともその基底クラスのどれかで定義されているのかを知っていない限り，君はそれと `Base2` の属性との名前衝突がどんな結果をもたらすのか結論できないことになる。深さ優先規則は `Base1` の直接の属性と継承された属性との差異を皆無にする)

Python は予期しない名前の衝突を規約に頼って回避しているから，多重継承を見境なく使うと，メンテナンスの悪夢になることは明らかである。多重継承にともなう有名な問題に，たまたま共通の基底クラスをもつ 2 個のクラスから派生させた 1 個のクラス，というものがある。この場合，何が起こるのかを結論することは簡単だが (インスタンスは，共通の基底クラスが使用する “インスタンス変数” つまりデータ属性を 1 組だけもつことになる)，この意味論が何か役に立つかどうかは明らかではない。

9.6 プライベート変数

クラス=プライベートな識別子のための限られたサポートがある。`__spam` (先頭に 2 個以上の下線文字，末尾に高々 1 個の下線文字) という形式の識別子はどれも今や字面上で `_classname__spam` へと置換される。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である。この符号化加工 (mangling) は，識別子の構文上の位置に関係なく行われるから，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，さらにはグローバル変数を定義するために利用できるほか，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**ほかの**クラスのインスタンスに格納するためにさえ利用できる。符号化加工した名前が 255 文字より長くなるときは，切り詰めが起こるかもしれない。クラスの外側では，またはクラス名が下線文字だけからできているときは，なんら符号化加工は起こらない。

名前の符号化加工は，派生クラスで定義されるインスタンス変数について心配する必要のない，あるいはクラスの外側のコードがインスタンス変数をいじりまわすことのない，そんな “プライベート” インスタンス変数およびメソッドを定義する簡便な方法を，クラスに与えることを意図している。この符号化加工の規則は主に不慮の事故を避けるために設計されていることに注意しよう。プライベートと見なされている変数を参照または改変することは，もし本当にそうしたければ，依然として可能である，このことは，たとえばデバッガにとっては有用でさえあり得る。そしてそれが，なぜこの抜け穴が閉ざされていないかの一つの理由である。(ささいなバグ: 基底クラスと同名のクラスを派生させると，基底クラスのプライベート変数の使用が可能になる)

`exec` や `eval()` や `evalfile()` へ渡されたコードは，呼出し元のクラスの classname を現在のクラスだとは見なさないことに注意しよう。これは `global` 文の効果と似ている — その効果もやはり，一緒にバイト=コンパイルされたコードに限定されている。同じ制約が `getattr()` と `setattr()` と `delattr()` にも，もちろん `__dict__` をじかに参照するときにも，適用される。

ここにあるのは，独自の `__getattr__` メソッドと `__setattr__` メソッドを実装して，すべての属性を一つのプライベート変数に格納するクラスの例である。これは，この機能が追加される前に入手可能だったバージョンを含め，すべてのバージョンの Python で動作する。

```
class VirtualAttributes:
    __vdict = None
    __vdict_name = locals().keys()[0]

    def __init__(self):
        self.__dict__[self.__vdict_name] = {}

    def __getattr__(self, name):
        return self.__vdict[name]

    def __setattr__(self, name, value):
        self.__vdict[name] = value
```

9.7 残りのはしばし

ときには Pascal の “record” や C の “struct” のように，一組の名前付きデータ項目を一つにまとめるデータ型があると便利である。空のクラス定義はこれをナイスに実現する。たとえば，

```
class Employee:
    pass

john = Employee() # 空の従業員レコードを造る

# 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期待する Python コード片に対し，しばしば，そのデータ型のメソッドをエミュレートするクラスを代用として渡すことができる。たとえば，ファイル・オブジェクトからデータを読んで書式化する関数に対し，君は，代用として文字列バッファからデータを得るメソッド `read()` と `readline()` を備えたクラスを定義して，それを引数として渡すことができる。

インスタンス・メソッド・オブジェクトにも属性がある。すなわち，`m.im_self` はメソッドのインスタンス・オブジェクトであり，`m.im_func` はメソッドに対応する関数オブジェクトである。

9.7.1 例外はクラスであってもよい

利用者定義の例外はもはや文字列オブジェクトに限られない— 例外はクラスによっても識別できる。このからくりを使えば，拡張可能な例外の階層構造を造ることができる。

raise 文に対し (意味論的に) 新しく妥当になった二つの形式がある。

```
raise Class, instance

raise instance
```

第一の形式では，`instance` は `Class` またはその派生クラスのインスタンスでなければならない。第二の形式は次の短縮記法である。

```
raise instance.__class__, instance
```

except 節には文字列オブジェクトだけでなくクラスを並べてもよい。except 節のクラスが例外と適合するのは，例外がそれと同じクラスまたはその派生クラスの場合である (しかし逆方向には適合しない — 派生クラスを並べた except 節は基底クラスとは適合しない)。たとえば，次のコードは B, C, D をこの順序で印字する。

```
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"
```

もしも except 節が逆に並んでいたら (‘`except B`’ が最初だったら)，B, B, B と印字されたはずだったことに注意しよう — 最初にマッチした except 節が駆動される。

処理されない例外に対してエラー・メッセージが印字される時，もしその例外がクラスならば，クラス名に続けてコロンとスペースが印字され，そして最後に，組込み関数 `str()` を使って文字列へと変換されたインスタンスが印字される。

CHAPTER

TEN

さあ何を？

このチュートリアルを読んで，Python を使うことへの君の関心はより強くなったことだろう — 君が抱えている現実世界の問題に是非とも Python を適用したくなったはずだ。さあ，これから君は何をしたらよいだろうか?

君は *Python Library Reference* を読むか，あるいはせめてそのページをパラパラとめくるとよい。これは Python プログラムを書く時間を大いに節約するのに役立つ型と関数とモジュールについての (ぶっきらぼうだが) 完全な参照資料を与えている。標準的な Python ディストリビューションは，C と Python の両方において**沢山の**コードを含んでいる。UNIX メールボックスを読んだり，HTTP 経由で文書を取り出したり，乱数を発生させたり，コマンド行オプションをパースしたり，CGI プログラムを書いたり，データを圧縮したり，その他多くのことをするモジュールがある。ざっとでも Library Reference に目を通すことは，何が利用可能なのかについての概念を君に与えるだろう。

主要な Python Web サイトは http://www.python.org/ だ。ここにはコード，文書類，そして Web のあちこちの Python 関連のページへのポインタがある。この web サイトは世界のあちこちのさまざまな場所，たとえばヨーロッパ，日本，オーストラリアなどでミラーされている。君の地理的な位置によっては，メイン・サイトよりミラーのほうが速いかもしれない。より非公式なサイトは http://starship.python.net/ だ。ここには一群の Python 関連の個人的ホーム・ページがある。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る。

Python に関する質問と問題の報告については，ニュースグループ comp.lang.python に投稿するか，python-list@python.org のメーリング・リストに送ればよい。この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。だから，一方に投稿されたメッセージは自動的に他方へ転送される。質問 (と回答), 新機能の提案，新モジュールのアナウンスなど，一日に約 120 通の投稿がある。投稿の前には，必ず http://www.python.org/doc/FAQ.html にある Frequently Asked Questions (つまり FAQ) のリストを確認するか，Python ソース・ディストリビューションの ‘Misc/’ ディレクトリを探すようにしよう。メーリング・リストのアーカイブは http://www.python.org/pipermail/ で入手可能だ。FAQ は何度も繰り返し現れる質問の多くに答えている。そこには君の問題に対する回答も既にあるかもしれない。

APPENDIX

A

対話入力編集とヒストリ置換

Python インタープリタの版によっては，Korn シェルや GNU Bash シェルと似た現在入力行の編集とヒストリ置換をサポートしている。これは *GNU Readline* ライブラリを使って実装されている。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る。このライブラリにはそれ自身のドキュメンテーションがあり，ここでそれを繰り返そうとは思わない。しかし基本は簡単に説明できる。ここで述べる対話編集とヒストリは UNIX 版と CygWin 版のインタープリタでオプションとして利用できる[1]。

この章は，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**しない**。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だ。

A.1　行編集

入力行編集は，もしサポートされているならば，インタープリタが一次または二次プロンプトを印字した時いつでも有効になっている。現在行を，通常の Emacs 制御文字を使って編集できる。そのうち最も重要なものを挙げる。`C-A` (Control-A) はカーソルを行の先頭へ移動させる，`C-E` は末尾へ移動させる，`C-B` は逆方向へ一つ移動させる，`C-F` は順方向へ移動させる。Backspace は逆方向に向かって文字を消す，`C-D` は順方向に向かって消す。`C-K` は順方向に向かって行の残りを kill する (消す)，`C-Y` は最後に kill された文字列を再び yank する (取り出す)。`C-underscore` は君がした最後の変更を元に戻す — これは繰り返して，どんどんさかのぼることができる。

A.2　ヒストリ置換

ヒストリ置換は次のように働く。出された非空の入力行はすべてヒストリ・バッファにセーブされる。新しいプロンプトが与えられたとき，君はこのバッファの底にある新しい行に位置している。`C-P` はヒストリ・バッファの中を 1 行だけ上に移動する (戻る)。`C-N` は 1 行だけ下に移動する。ヒストリ・バッファのどの行も編集できる。行が修正されたことを表すためプロンプトの前にアステリスクが現れる[2]。`Return` キーを押すと現在行がインタープリタへ渡される。`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) を開始する。

A.3　キー束縛

Readline ライブラリのキー束縛 (key binding) やその他のパラメタは，“~/.inputrc’ という初期化ファイル[3]にコマンドを置くことによってカスタマイズできる。キー束縛の形式は

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

[1]訳注: BeOS R5 版の Python 2.1 でも利用できるようです。

[2]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。

[3]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。

または

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

であり，オプションの設定方法は

```
set option-name value
```

である。たとえば，

```
# 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
```

Tab に対する Python でのデフォルト束縛は，TAB の挿入であって，Readline のデフォルトであるファイル名補完関数ではないことに注意しよう。もし，どうしてもと言うならば，君の ‘~/.inputrc’ に

```
Tab: complete
```

を入れることによって，これをくつがえすことができる。(もちろん，こうすると段付けした継続行を打鍵しにくくなるのだが)

変数名とモジュール名の自動的な補完が，オプションで利用できる。それをインタープリタの対話モードで有効にするには，下記を君のスタートアップ・ファイルへ追加する[4]。

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

これは TAB キーを補完関数に束縛するから，TAB キーを 2 回たたくと補完候補が示される。この補完は Python の文の名前と，現在のローカル変数と，利用可能なモジュール名を検索する。`string.a` のようなドット付き式については，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す。もし `__getattr__()` メソッド付きのオブジェクトがその式の一部ならば，これ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ことに注意しよう。

A.4 解説

この機能は，初期の版のインタープリタに比べれば大きな前進である。しかし，いくつかの要望が残されている。正しい段付けが継続行で提示されたら快適だろう (パーサは次に段付けトークンが必要かどうかを知っている)。補完機構がインタープリタの記号表を使ってもよいかもしれない。かっこやクォートの対応をチェックする (さらには提示もする) コマンドも有用だろう。

[4]君が対話インタープリタを開始する時，Python は PYTHONSTARTUP 環境変数が指定するファイルの内容を実行する。