Python 簡易資料

2018/7/16 改訂
by NIDE, N. (nide@ics.nara-wu.ac.jp)

目 次

1 参考書籍&オンラインドキュメント . . . 1
2 Python プログラムの実行 . . . 1
3 文字コードの指定 . . . 3
4 データ型 . . . 4
5 基本構文 . . . 13
6 関数定義 . . . 19
7 実行時の引数 . . . 21
8 ファイル . . . 22
9 クラス . . . 24
10 モジュール . . . 30
11 番外: Tkinter . . . 32

1 参考書籍&オンラインドキュメント

書籍は「初めての Python」(Mark Lutz, 夏目大訳、O'Reilly) がとりあえず原典 (買うなら最新のものを)。ただし非常に分厚く、「初めての」という名前とは裏腹に、入門者対象の本では必ずしもない。むしろ Python のことなら何でも載っている本として使える。

初心者用書籍のおすすめは「みんなの Python」(柴田淳, ソフトバンククリエイティブ) か。

オンラインドキュメントは、`https://docs.python.jp/2/` に日本語版のものが各種ある (Python 3.X 用のものは `https://docs.python.jp/3/`)。その中には「チュートリアル」もあり、入門用に便利 (これは書籍の「Python チュートリアル」(Guido van Rossum, 鴨澤眞夫訳, O'Reilly) ともほぼ同等の内容と思われる)。また、組み込みの関数やメソッドにどんなものがあるか調べたいときなどは、同ページの「ライブラリリファレンス」や「言語リファレンス」が有用。

Python は、バージョンが 2.X から 3.X に変わる段階で結構大規模な仕様変更があった (print が文でなく関数に、文字列フォーマッティングの変更、文字列とバイト列の分離、など)。この資料の初稿を書いていた当時は、まだ 2.X (主に 2.6, 2.7 など) の方が広く使われていたため、この資料も原則的に **Python 2.X について**述べ、必要に応じて **3.X との違い**を述べる形であるが、近年は主流が 3.X に交代しつつある。

2 Python プログラムの実行

Python はスクリプト言語で、AWK, Perl, Ruby などの言語と同様の特徴や文化を持つ。他の言語にない Python 独特の特徴は、「if や for などの構文を、begin～end や "{" "}" などではなく**インデント** (字下げ) で表現する」という点 (5 節)。

Python プログラムは大きく分けて以下の 3 つの実行方法を持つ。

1. コマンド行にプログラムを直接与えて Python インタプリタを実行 (いわゆる「ワンライナー」)
2. ファイルにプログラム (スクリプト) を書き、Python インタプリタにそのファイルを与えて実行
3. 2. と同様だが、スクリプトファイルの先頭行に「`#!`」に続けて Python の絶対パスを書き、スクリプトファイルをコマンドにする

また、これらに加えて Python には

4. Python インタプリタを起動し、対話的にプログラムを実行

という実行方式もある。

それぞれについて以下で述べる。

2.1 ワンライナー

「python -c 'Pythonプログラム'↵」で Python プログラムを実行できる。プログラムが短い場合に便利。

```
$ python -c 'print "Hello"'↵          (「print "Hello"」の部分が Python プログラム)
Hello
```

python コマンドには他にもオプションがあるが (ただし多分あまり使わない。man コマンド man python 参照)、それらを指定する場合は、-c オプションは他のオプションよりも後でなければならない。

Python 3.0 以後は、print が関数になるため、print("Hello") のように print の引数は**括弧で囲まなくてはならない**。次節以降の例でも同様であることに注意。なお Python 2 以前でも、print の引数を括弧で囲んでも特に支障はない。

2.2 スクリプトファイルを作成して実行

以下のファイルを hello.py という名前で作成。

```
print "Hello"
print 1+2
```

そして以下のように、「python ファイル名↵」で実行。

```
$ python hello.py↵
Hello
3
```

2.3 スクリプトファイルをコマンドにする

以下のファイルを hello という名前で作成。

```
#!/usr/bin/python
print "Hello"
print 1+2
```

そして

```
$ chmod a+x hello↵
```

として実行可能にすると、hello がコマンドになる (この操作は 1 度だけでよい)。これを

```
$ ./hello↵
Hello
3
```

のようにして実行。

先頭の「#!/usr/bin/python」の行は、システムによってどう書けばよいかが異なる。

```
$ which python↵
/usr/bin/python
```

として出力された python コマンドのフルパスを、「#!」に続けて書く。

次節以降では原則的にこの実行方法をとるものとして説明する (2.4 節で述べる**対話的実行**の場合を除く)。

2.3.1 PATH を通す

上の hello ファイルを、環境変数 PATH で指定されたディレクトリに移動させておけば、「./hello↵」でなく「hello↵」で実行できるようになる。環境変数 PATH の設定は、(シェルが bash の場合) ホームディレクトリの .bashrc ファイ

ルに書く。次の例では、chmod a+x hello は既に済んでいるものとする。

```
$ emacs ~/.bashrc↲
…末尾に「export PATH=~/bin:"$PATH"」と追加…
$ cat ~/.bashrc↲
  ⋮
export PATH=~/bin:"$PATH"
$ mv hello ~/bin↲      (ここで hash -r あるいは rehash コマンドが必要な場合もある (基本的には不要))
$ hello↲
Hello
3
```

2.4 対話的実行

python コマンドを無引数で起動すると「>>>」が表示され、そこへ Python の文や式を入力するとその結果を表示する。キーボードから EOF (**Ctrl-D**、ただし DOS や Windows の場合は Ctrl-Z) あるいは「exit()」を入力すると終了する。

```
$ python↲
Python 2.7.9 (default, Aug 13 2016, 16:41:35)
  ⋮
>>> 1+2↲
3
>>> (Ctrl-D)
$
```

以後本資料では、「>>>」で始まっている部分はこの**対話的実行**による入力を意味する。

2.5 Emacs の Python モード

Python モードがインストールされ、しかも正しく設定されている Emacs の場合、

1. ファイル名が「.py」で終わっている
2. 先頭行が「#!/usr/bin/python」で始まっている
   (「#!/usr/local/bin/python」とか「#!/usr/bin/env python」などでも可)

の**いずれか**を満たすファイルを編集しようとすると、自動的に **Python モード**になる (モードラインに「(Python)」と出るのでわかる)。Python モードになっていると、構文が色分けで表示されたり、Tab キーで自動的にインデントできたりして楽。また、キーボードから ESC x python-mode↲ としても Python モードになる。

.emacs ファイルに、以下のように**書き足し**ておくと、Python モードではリターンキーで自動的に次の行をインデントするようになる (Python モード以外には影響なし)。インデントを調整したい場合は **Tab キー**を何度か押せばよい。

```
(add-hook 'python-mode-hook
      (lambda ()
        (define-key python-mode-map "\C-m" 'newline-and-indent)))
```

3 文字コードの指定

Python は国際化されており、日本語ほか多言語の文字を扱えるが、ASCII 文字以外の文字を直接スクリプトに書く場合は、スクリプトがどの**文字コード**で記述されているかを、注釈の形式で書いておかねばならない。例えば

```
#!/usr/bin/python
print "あ"
```

というプログラムを実行すると

```
  File "a.py", line 2
SyntaxError: Non-ASCII character '\xa4' in file a.py on line 2, but no encoding declared;
see http://python.org/dev/peps/pep-0263/ for details
```

のようにエラーとなる (以前の Python では、警告は出るものの実行はされていた)。このプログラムが日本語 EUC で記述されているなら

```
#!/usr/bin/python
# coding: euc-jp
print "あ"
```

のように書いておけば、エラーせず正常に実行できる (文字コードが UTF-8 なら「# coding: utf-8」と書く[1])。

この「# coding: euc-jp」の行を「エンコード宣言」と呼ぶ。エンコード宣言は「#!」行の直後 (つまり 2 行目。ただし「#!」行がない場合は先頭行) に書かなければならない。

ただし、「# coding: euc-jp」と書くより、「# -*- coding: euc-jp -*-」と書く方が (開発に Emacs を使うなら) より望ましい。Emacs がこの行を認識して文字コードを把握し、(Python モードでの) 文字列の構文認識を正しく行ってくれるため。

以後本資料では、たとえプログラム中に日本語文字が全くなくても、「# -*- coding: euc-jp -*-」の行を書くことに統一する。

なお、Python では、「#」から行末までが**注釈** (注釈開始の「#」は行頭になくてもよい)。エンコード宣言は、注釈の特別な場合ということになる。

3.1 注釈に注意

たとえ**注釈の中**であっても、ASCII 文字以外の文字が出てくる場合は、エンコード宣言で文字コードを指定しておかねばならない。例えば

```
#!/usr/bin/python
# これは注釈です
print "Hello"
```

というプログラム (注釈は日本語 EUC で書かれているとする) を実行すると、注釈以外に非 ASCII 文字は全くないにもかかわらず、上記と同様にエラーとなってしまう。が、

```
#!/usr/bin/python
# -*- coding: euc-jp -*-
# これは注釈です
print "Hello"
```

ならエラーにならず実行できる。

4 データ型

Python には、基本的なデータ型として、

- 数 (多倍長整数 · 実数 · 複素数)
- 真偽値 (True と False)
- 文字列
- リスト (配列に相当するもの。サイズも含めて変更可能)
- タプル (配列に相当するもの。変更不能)

[1] Python3 からは、文字コードが UTF-8 である場合、エンコード宣言は不要になる。

- 辞書 (配列に似ているが、任意のデータを添字にできる)

などがある。リストやタプルや辞書は入れ子にできる。

変数 (Python では「名前」と呼ぶ) には型はないので、どの変数にもどの型のデータでも代入できる。変数名の規則は C などと同様 (英字か「_」で始まり英数字と「_」からなる[2]。if などの予約語は除く)。また、変数の**宣言は不要**。さらに、関数定義 (6 節) の中で使われる変数は原則として自動的に**ローカル変数**になる。

また、**オブジェクト指向プログラミング**の機能があるので、クラスを定義 (9 節) することによって**新たなデータ型**を作ることもできる。

4.1 数に対する演算

```
$ python↵
Python 2.7.9 (default, Aug 13 2016, 16:41:35)
  ⋮
>>> 2**100↵
1267650600228229401496703205376L      (** は累乗。無限多倍長整数が使える)
>>> 7.0/5↵
1.3999999999999999
>>> print 7.0/5↵
1.4                                   (対話的実行では、print 文を使うと出力が少し見やすくなる)
>>> 7/5↵                              (整数除算。ただし Python3 では 7.0/5 と同じになる)
1
>>> 7//5↵                             (こちらは Python2 でも 3 でも整数除算)
1
>>> a = 3↵                            (変数の使用)
>>> a + 5↵
8
>>> import math↵                      (数学関数を使うのに必要)
>>> math.sqrt(2)↵
1.4142135623730951
```

`import math` のところは、数学関数を扱う「math モジュール」の読み込み (インポート)。import については 10 節参照。

他に**複素数** (数学関数を使う場合は cmath モジュール要)、分数 (fractions モジュール要) なども使える。複素数を使う場合、虚数単位は「`j`」と表記する (`i` ではない)。

```
>>> (1-2j)*(3+4j)↵
(11-2j)                                 (複素数演算。(1 − 2i)(3 + 4i) = 11 − 2i の計算)
>>> import cmath↵
>>> cmath.sqrt(2j)↵
(1+1j)                                  (√2i = 1 + i の計算)
>>> cmath.sqrt(-1)↵
1j                                      (√−1 = i の計算)
>>> import math↵
>>> math.sqrt(-1)↵
Traceback (most recent call last):      (math モジュールは実数用のため
  File "<stdin>", line 1, in <module>    math モジュールの sqrt では −1 の平方根は計算できない)
ValueError: math domain error
>>> from fractions import Fraction↵     (モジュール読み込みのもう 1 つの方法、10 節参照)
>>> Fraction(1,2)+Fraction(1,3)↵
Fraction(5, 6)                          (1/2 + 1/3 = 5/6 の計算)
```

さらには、任意精度の小数 (decimal モジュールまたは mpmath モジュールなどが必要) なども利用可能 (略)。

[2] Python 3 以降では漢字なども変数名に使える。

4.2 真偽値

「真偽値」に属する値は True と False のみ。and, or, not などの演算ができる。数と混ぜて演算することもでき、その場合は True は整数 1、False は整数 0 として扱われる。

```
>>> True and not (True or False)↵
False
>>> True + 4↵
5
```

4.3 文字列に対する演算

```
>>> print 'a\nb\'c\\d'↵                  (文字列は「'」と「"」のどちらでも囲める (扱いは同じ))
a
b'c\d
>>> print r'a\nb\'c\\d'↵                 (raw 文字列; r を付けると「\」がエスケープ記号では
a\nb\'c\\d                                なく「\」そのものとして扱われる)
>>> a = '''abc↵                          (ブロック文字列; 複数行の文字列を「\n」を使わずに書ける)
... def   ↵                              (対話モードで前の行の続きを入力する必要がある場合は
... ghi'''↵                               「...」が行頭に表示される)
>>> a↵
'abc\ndef\nghi'
>>> print a↵
abc
def
ghi
>>> 'ab' + 'cde' * 4↵                    (文字列の連接や反復)
'abcdecdecdecde'
>>> a = 'abcde!'↵
>>> a[1:3]↵                              (部分文字列。「:」の前後の省略も可能)
'bc'
>>> a[1:-2]↵                             (負数を指定すると後ろから数える)
'bcd'
>>> len(a)↵
6
>>> 'cd' in a↵                           (部分文字列として含まれているか)
True
>>> 'bd' in a↵
False
>>> a.index('cd')↵                  (文字列メソッドの 1 つ; 部分文字列として何文字目から始まるか)
2                                        (先頭を 0 と数えるので)
>>> a.index('bd')↵
Traceback (most recent call last):       (index メソッドの代わりに find メソッドを使うと、
  File "<stdin>", line 1, in <module>     部分文字列が見つからない場合にエラーにならずに
ValueError: substring not found           −1 を返す)
>>> a.upper()↵                           (大文字化。他に小文字化などのメソッドもある)
'ABCDE!'
>>> a↵
'abcde!'                                 (元の文字列は変更されない)
>>> 'ab,c,de'.split(',')↵                (これも文字列メソッドの 1 つ; 区切り文字で分割して
['ab', 'c', 'de']                        リストに)
>>> ' a   bc d '.split()↵                (引数がなければ空白で分割。
['a', 'bc', 'd']                          ' a   bc d '.split(' ') とは異なる)
>>> a = 'xabyabz'↵
>>> a.replace('ab','#')↵                 (文字列の置き換え)
'x#y#z'
```

```
>>> a.replace('ab','#',1)↵                (先頭の n 個だけ置き換え)
'x#yabz'
>>> a↵
'xabyabz'                                 (元の文字列は変更されない)
>>> a = '1.3'↵
>>> float(a)↵                             (実数への変換。整数への変換には int() 関数を使う)
1.3
>>> '%d and %.3f and %s' % (1, 2.3, 'XYZ')↵      (文字列フォーマッティング[3])
'1 and 2.300 and XYZ'
>>> 'あ'↵                                 (バイト列)
'\xa4\xa2'
>>> u'あ'↵                                (Unicode 文字列[4])
u'\u3042'
```

組み込みの機能の中には、メソッドとして提供されるものとそうでないものがある。おおむね、その型に特有の操作 (文字列の split など) はメソッド、いくつかの型に共通な操作 (例えば len や `'abcde'[1:3]` など。これらは 4.4 節のリストやタプルにも適用可) はメソッド以外の構文 (関数など) として表現される (例外もあるが)。

[備考] Unicode 文字列をプログラム中で使い、これを出力する場合は注意がいる。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-
print u'あ'
```

というプログラムを実行する際、普通に実行すると正常に実行できるが、標準出力をリダイレクトすると

```
Traceback (most recent call last):
  File "pytest", line 3, in <module>
    print u'あ'
UnicodeEncodeError: 'ascii' codec can't encode character u'\u3042' in position 0:
ordinal not in range(128)
```

とエラーになる。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-
print u'あ'.encode('euc-jp')
```

なら、標準出力をリダイレクトしてもエラーにならない。なお、Unicode 文字列でない文字列ではこの問題は生じない。
Python 3.0 ではこの問題はなくなる (そもそも「`u'あ'`」という書き方がなくなる)。

4.3.1 正規表現

使える正規表現自体は他の言語 (特に Perl) とほぼ同じ (詳しくは `https://docs.python.jp/2/library/re.html` の 7.2.1「正規表現のシンタクス」に書かれている)。Python では、正規表現処理を行うには

1. まず、正規表現の文字列を「**正規表現オブジェクト**」に変換
2. 正規表現オブジェクトを使って、マッチさせたい文字列とのマッチに関する情報を取り出す

という手順をとる。

```
>>> import re↵                        (正規表現処理を行うのに必要)
>>> r = re.compile('ab*c')↵           (r に、正規表現 ab*c を表す「正規表現オブジェクト」が入る)
>>> m = r.search('pqrabbbcxyz')↵      (文字列 'pqrabbbcxyz' と r を照合し、結果を m へ)
>>> m.start()↵                        (マッチした部分の先頭の位置を取り出す)
3
>>> m.end()↵                          (マッチした部分の末尾の位置を取り出す)
8
>>> m.group()↵                        (マッチした部分の文字列を取り出す)
'abbbc'
```

[3] Python 2.6 からは `'{0} and {1:.3f} and {2:s}'.format(1, 2.3, 'XYZ')` のように、文字列の format メソッドを使う新しい書き方が導入された。Python 3.1 では完全にこれに置き換わる予定であったが、実際には「%」を使う従来の書き方も Python 3.6 の時点でまだ残っている。

[4] Python 3.0 では単に `'あ'` と書けばこちらの意味になり、`u'あ'` という書き方はなくなる。

上の例で、もし r.search の引数の文字列が r で表される正規表現にマッチしなかった場合、m には「偽」を表す値が入るので、**if 文** (5.1 節) を使って、マッチしていたかどうかの**判定**ができる[5](下記のサンプルプログラムは 6 節に出てくる**関数定義**を使っている)。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-
import re

# 第1引数の正規表現が第2引数の文字列にマッチするかのテスト、
# およびマッチした文字列の取り出し・出力を行う関数を定義
def match_test(re_obj, string):
    m = re_obj.search(string)   # このmは自動的にローカル変数になる
    if m:
        print 'マッチした部分: %s' % m.group()
    else:
        print 'マッチしません'

r = re.compile('ab*c')
match_test(r, 'pqrabbbcxyz')
match_test(r, 'pqrxyz')
```

このプログラムは

```
マッチした部分: abbbc
マッチしません
```

と出力する。

なお、正規表現には「\」が含まれることが多い (上の例はそうではないが) ので、raw 文字列 (4.3 節) で書かれることが多い (上の例では re.compile(r'ab*c') のように)。raw 文字列で書くと、正規表現内の「\」を「\\」と書かなくてよいためである (例えば「re.compile('*')」でなく「re.compile(r'*')」と書ける)。

[***正規表現との直接照合***] re.compile を使わず、正規表現を直接文字列と照合することもできる。

```
>>> import re↵
>>> m = re.search('ab*c', 'pqrabbbcxyz')↵
```

(文字列 'pqrabbbcxyz' と正規表現 'ab*c' を照合し、結果を m へ。以後は先の例と同じ処理ができる)

しかし、同一の正規表現を繰り返し使う場合は、re.compile を使う方が効率がよい。

[***複数のマッチ処理***] 文字列の中に正規表現にマッチする部分が複数あり、そのそれぞれを取り出して処理したい場合は、正規表現オブジェクトの finditer メソッドと、for 文 (5.3 節) による繰り返しを組み合わせて、以下のようにする。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-
import re

r = re.compile('ab*c')
for m in r.finditer('pqrabbbcxyzabbcjkl'):  # for文(繰り返し構文)を利用
    print '%d文字目と%d文字目の間の「%s」にマッチ' % (  # 行が継続
        m.start(), m.end(), m.group())
```

このプログラムは以下のように出力する。

```
3文字目と8文字目の間の「abbbc」にマッチ
11文字目と15文字目の間の「abbc」にマッチ
```

[5] 5.1 節で述べるように、Python プログラムはインデント (字下げ) を正しく行わないと動かない。インデントも含めて資料通りにプログラムを入力されたい。

なお、Python では「( )」「{ }」「[ ]」の中で改行した場合は**継続行**として扱われる (上の例のプログラムの最後の行がそれ)[6]。

[*置換・分割*] 正規表現を使う操作としては、単なるマッチ検査の他、「マッチした部分を他の文字列に**置換**」「マッチした部分で文字列を**分割**」が代表的。

```
>>> import re↵
>>> r = re.compile('ab*c')↵
>>> r.split('pqrabbcxyzabbbcjkl')↵          (分割。文字列の split メソッドと違って、
['pqr', 'xyz', 'jkl']                          分割される文字列が引数)
>>> r.sub('!', 'pqrabbcxyzabbbcjkl')↵       (置換)
'pqr!xyz!jkl'
>>> s = 'pqrabbcxyzabbbcjkl'↵
>>> r.sub('!', s, 1)↵                       (先頭の n 個だけ置換)
'pqr!xyzabbbcjkl'
>>> s↵
'pqrabbcxyzabbbcjkl'                          (引数の文字列そのものは変わらない)
```

4.4 リスト・タプルとそれに対する演算

4.4.1 リスト

Python での配列に相当するもの。

```
>>> a = [1, 'abc', 2]↵                      (要素の型は混ざってもよい)
>>> a↵
[1, 'abc', 2]
>>> b = a↵
>>> b↵
[1, 'abc', 2]
>>> a[1]↵                                   (要素の参照)
'abc'
>>> a[1] = 'def'↵                           (要素の入れ換えも可)
>>> a↵
[1, 'def', 2]
>>> b↵
[1, 'def', 2]                                 (a と b は同じリストを参照している)
>>> a[3] = 'def'↵
Traceback (most recent call last):            (存在しない要素の書き換えは不可)
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>> a += ['xyz', [4, 5]]↵                   (要素の追加は可能)
>>> a↵
[1, 'def', 2, 'xyz', [4, 5]]                  (入れ子のリストになった)
>>> b↵
[1, 'def', 2, 'xyz', [4, 5]]                  (a と b はまだ同じリストを参照している)
>>> a = a + ['pqr', [6, 7]]↵
>>> a↵
[1, 'def', 2, 'xyz', [4, 5], 'pqr', [6, 7]]
>>> b↵
[1, 'def', 2, 'xyz', [4, 5]]                  (a と b は違うリストになった)
>>> len(a)↵
7
>>> a[-2]↵
'pqr'                                         (負の添字を指定すると後ろから数える)
```

[6]継続行ではインデントは無視されるので、自由にインデントしてかまわない。

```
>>> 'xyz' in a ↵
True
>>> a.index('xyz') ↵
3
>>> a[3:5] ↵                                    (スライス。「:」の前後の省略も可能)
['xyz', [4, 5]]
>>> a[3:5]=[7, 8, 'ijk'] ↵
>>> a ↵
[1, 'def', 2, 7, 8, 'ijk', 'pqr', [6, 7]]
>>> a[7][0] ↵
6
>>> a.pop() ↵                                   (リストメソッドの例)
[6, 7]
>>> a ↵
[1, 'def', 2, 7, 8, 'ijk', 'pqr']
>>> a.append(3) ↵
>>> a ↵
[1, 'def', 2, 7, 8, 'ijk', 'pqr', 3]
>>> a.reverse() ↵
>>> a ↵
[3, 'pqr', 'ijk', 8, 7, 2, 'def', 1]
>>> sorted(a) ↵
[1, 2, 3, 7, 8, 'def', 'ijk', 'pqr']
>>> a ↵
[3, 'pqr, 'ijk', 8, 7, 2, 'def', 1]             (a の値は変わっていない)
>>> a.sort() ↵
>>> a ↵
[1, 2, 3, 7, 8, 'def', 'ijk', 'pqr']            (a の値が変わる)
>>> max([5, 7, 1]) ↵                            (他に min 関数もある)
7
```

Python のリストは、Lisp 言語や Prolog 言語のリストと違って linked list として実装されてはいない。その結果として、どの要素にも定数時間でアクセスできる (Lisp や Prolog のリストは、後ろの方の要素ほどアクセスに時間がかかる)。反面、前の方に要素を**加除**するのは、後ろの方に要素を加除するより時間がかかる。

巨大な配列を用いた数値計算には、numpy モジュールの使用が適している (この資料では述べない)。

sort メソッドはタプルにはない。一方、sorted 関数はタプル (4.4.2 節) にも使える (が、結果はリストになる)。in 演算子、index メソッド、len · max · min 関数もタプルにも使える。これらのうち、sort, sorted, max, min など大小比較を伴う関数やメソッドでは、文字列と数を混ぜて比較させると、文字列の方が大きいという判定になる (Python3 ではそのような比較はエラーになる)。なお、これらの関数やメソッドでは、(要素の比較前に自分で決めた関数を要素に適用させることによって) 比較の順序を変えることもできる (本資料では略)。

4.4.2 タプル

リストに似るが、要素の書き換えができない。

```
>>> a = (1, 'abc', (2, 'pqr'), [3, 'xyz']) ↵
>>> a ↵
(1, 'abc', (2, 'pqr'), [3, 'xyz'])
>>> b = (3,) ↵                  (要素 1 つのタプルはこう記述する; 「b = (3)」ではだめ)
>>> b ↵
(3,)
>>> len(a) ↵
4
>>> a[1] ↵                      (a(1) ではないことに注意)
'abc'
```

```
>>> a[1] = 'def'↵
Traceback (most recent call last):        (タプルはリストと違って要素の書き換えはできない)
  File "<stdin>", line 1, in <module>
TypeError: object does not support item assignment
>>> a = a[0:1]+('def',)+a[2:]↵            (どうしても要素を書き換えたければタプルを作り直す)
>>> a↵
(1, 'def', (2, 'pqr'), [3, 'xyz'])
>>> a[3][0]=4↵                            (でもこれはできる; 書き換えているのはリストの要素だから)
>>> a↵
(1, 'def', (2, 'pqr'), [4, 'xyz'])
>>> list(a)↵                              (タプルをリストに変換)
[1, 'def', (2, 'pqr'), [4, 'xyz']]        (逆の変換は tuple 関数で可)
```

Python のデータは「**不変性オブジェクト**」と「**可変性オブジェクト**」に分類できる。中身の書き換えができるのが後者。タプルや数や文字列などは不変性、リストや辞書などは可変性。

[*代入の左辺のタプルやリスト*] Python では、代入式の左辺をタプルやリストとすることによって、一度に複数の変数に代入したりできる。

```
>>> (a, b) = (1, 2)↵                (実は右辺は [1, 2] でも OK)
>>> a + b↵
3
```

また、タプルを囲む「( )」が省略されることがままある。

```
>>> a, b = 1, 2↵
>>> a + b↵
3
```

[*map 関数・filter 関数*] リストにもタプルにも使える (ただし、map 関数をタプルに適用した結果はリストになる)。map 関数は Lisp 言語でいう mapcar 関数と同じ働き。filter 関数は、指定した条件に合う要素だけ取り出す働きをする。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-
def nijo(x): # nijo関数の定義
    return x*x
def even(x): # even関数の定義
    return x % 2 == 0    # xが偶数ならTrue、でなければFalseが返る

print map(nijo, [1, 3, 5])
print filter(even, [4, 7, 2])
```

このプログラムは

```
[1, 9, 25]
[4, 2]
```

と出力する (このプログラムも関数定義を使っている)。なお、同じことは「ラムダ記法」というものを使うと

```
#!/usr/bin/python
# -*- coding: euc-jp -*-
print map(lambda x: x*x, [1, 3, 5])
print filter(lambda x: x % 2 == 0, [4, 7, 2])
```

と短く書ける (関数の本体が 1 つの式で書けるなら)。

Python 3.0 からは、map 関数はリストではなく「イテレータ」というもの (5.3 節) を返すようになるため、上の例はそのままではリストを出力しなくなる。リストとして出力したければ、map 関数の返り値 (イテレータ) を list 関数 (リストへの変換) に渡して `print(list(map(nijo, [1, 3, 5])))` や `print(list(filter(lambda x: x % 2 == 0, [4,`

7, 2]))) のようにする。
map 関数の戻り値をそのままループ構文で使うような場合は、Python 3 のように、map 関数がイテレータを返す方が効率が良くなる利点がある。

4.5 辞書とそれに対する演算

辞書とは、'添字' として任意の**不変性**オブジェクト (4.4.2 節) を取れる配列のようなもの。Java や Perl などにもある[7]。リストやタプルと違って、要素に順番はない。
辞書の '添字' のことを「キー」(鍵) という。

```
>>> a = {}↵                              (空の辞書を作る)
>>> a↵
{}
>>> a[2] = 'def'; a['abc'] = [3,5]↵      (要素の追加)
>>> a↵
{2: 'def', 'abc': [3, 5]}
>>> b = {(3,5): 9, 1: (2,4)}↵            (最初から要素のある辞書を作る)
>>> b↵
{1: (2, 4), (3, 5): 9}                   (要素の順序は保存されない)
>>> a[2] = b↵                            (要素の書き換え)
>>> a↵
{2: {1: (2, 4), (3, 5): 9}, 'abc': [3, 5]}
>>> a[2]↵
{1: (2, 4), (3, 5): 9}                   (要素の取り出し)
>>> a[3]↵                                (そのキーに対する要素がないとエラー)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 3
>>> 3 in a↵                              (a にキー 3 に対する要素があるか)
False
>>> a.get(2, 'pqr')↵                     (要素の取り出し、ただしデフォルト値つき)
{1: (2, 4), (3, 5): 9}
>>> a.get(3, 'pqr')↵                     (指定したキーの要素がなければデフォルト値が返される)
'pqr'
>>> del a['abc']↵
>>> a↵
{2: {1: (2, 4), (3, 5): 9}}
```

4.5.1 集合

集合型は、Python 2.4 以降で利用可能になった。

```
>>> a = set([3,1,2,3])↵          (Python 2.7 以降では a = {3,1,2,3}と書けるようになった[8])
>>> a↵
set([1, 2, 3])                   (集合なので要素の重複はない)
>>> b = set([2,3,1])↵
>>> a == b↵
True
>>> a.add(9)↵
>>> a↵
set([1, 2, 3, 9])
```

集合型の要素になれるのは不変性オブジェクトだけ。

[7] Java や Perl など他言語のものは「ハッシュ」や「連想配列」とも呼ばれる。
[8] 例外として、空の集合は Python 2.7 以降でも {} とは書けず (空の辞書と区別がつかなくなるため)、 a = set() のようにしないと作れない。

5 基本構文

構文を**インデント** (字下げ) で表現するのが Python の特徴。

5.1 if 文

```
#!/usr/bin/python
# -*- coding: euc-jp -*-

import sys  # sys.stderrやsys.exit()などを使うため
import math
# 2つまとめて import sys, math とも書ける

l = raw_input('a = ? ')      # 文字列の1行入力。Python 3ではinput関数を使う
a = float(l)                 # 文字列から実数へ変換
if a >= 0:
    b = math.sqrt(a)
else:
    print >>sys.stderr, 'a < 0'     # 標準エラー出力へのprintはこう書く
    sys.exit(1)              # exit関数。指定の戻り値でプログラムを終了させる
print b
```

慣れないうちは、条件式や else などの後ろの「:」を付け忘れやすい。

raw_input 関数は、入力が 1 文字もないうちに入力が終了 (EOF) すると EOFError というエラーを起こし、プログラムが終了する。これを防ぐには、このエラーを捕捉 (5.4.1 節) する必要がある。

Python 3 では raw_input 関数は input 関数という名に変わっている。なお、Python 2 にも input という名の関数があるが、これは働きが異なる。

Python 3 で print 関数の出力先を変えるには、`print('a < 0', file=sys.stderr)` のようにする。

「sys.」の付かない単なる exit という関数もあるが、これは対話的実行を終了させるのに使う (2.4 節)。プログラム中でそのプログラムを終了させるには、sys.exit の方を使うべきである。なお、sys.exit の引数に文字列を与えると、その文字列 (と改行) を標準エラー出力に出力して、戻り値 1 でプログラムを終了するという機能もある (従って上のプログラムの `else:` の後 2 行はまとめて「`sys.exit('a < 0')`」とも書ける)。

条件式のところには、「真」を表す定数 True や「偽」を表す定数 False も書ける。また、条件式の値が数値の 0、空文字列、定数 None、空のリスト · タプル · 辞書 · 集合のいずれかになった場合は、「偽」の扱いになる。

[*インデントによる構文*] 構文は**インデントで決まる**ので、インデントは必ず正しく行わねばならない。例えば上のプログラムの `else:` 以降の部分を

```
else:
    print >>sys.stderr, 'a < 0'
sys.exit(1)
```

のようにすると、`sys.exit(1)` の部分は **if 文の外**ということになり、a が 0 以上であっても戻り値 1 でプログラムが終了してしまう。また、if の直後が

```
if a >= 0:
b = math.sqrt(a)
```

のようにインデントなしになっていると、**構文エラー**となる。

この理由から、「条件が成り立ったときに何もしたくない場合」には、

```
if a >= 0:
else:
    …
```

と書くことはできない (「`if a >= 0:`」の直後にはインデントされた行が必要なため)。このような場合は、「何もしない文」である `pass` を使って

```
if a >= 0:
    pass
else:
    …
```

のように書く必要がある (あるいは 5.1.1 節で述べる not を使って if not a >= 0: と書く)。

5.1.1 elif、および条件式の複合

elif も使える。また、条件式中の「かつ」「または」「否定」には、C のような「&&」など**ではなく**「and」「or」「not」を使う (4.2 節に出てきた and, or, not である。Python ではこれらを真偽値間の演算のオペレータとして使うため)。and や or 演算子には、C の && や || と同様、「**短絡評価**」が行われるという性質がある。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-

def if_test(x):
    if 3 <= x and x <= 5:   # 実はPythonでは「if 3 <= x <= 5:」とも書ける
        print '3以上5以下'
    elif x < 2 or not isinstance(x, int): #「isinstance(x, int)」はxの型が整数か否かの判定
        print '2未満か整数でない'
    else:
        print 'その他'

if_test(4)
if_test(2.7)
if_test(6)
```

このプログラムもまた、関数定義 (6 節) を使っている。このプログラムの実行結果は次のようになる。

```
3以上5以下
2未満か整数でない
その他
```

5.2 while 文

```
#!/usr/bin/python
# -*- coding: euc-jp -*-

def while_test(x):
    while x > 0:
        print x,     # 後ろが「,」で終わっていると改行せずに出力
        x -= 1       # 「--」や「++」はPythonにはない
    print 'Boom!'

while_test(3)
```

このプログラムは「3 2 1 Boom!」と出力。

print 文の最後を「,」で終えると、改行しないだけでなく、その次に print で出力する際に、その前に空白が 1 つ自動的に出力されることにも注意 (ただし、次の出力が行頭から始まる場合を除く)。出力が「321Boom!」とならないのはそのためである。

Python 3.0 では、print x, で改行せずに出力する手が使えない。Python 3.0 では print(x, end = '') とする必要がある。ただしこうすると出力は「321Boom!」となり、間に空白が入らない。print(x, ' ', end = '') あるいは print(x, end = ' ') のようにして明示的に空白を入れてやる必要がある。

ループ内で使える、C 言語と同様の break や continue がある。加えて Python の while 文には、**else 部**も付けられる (繰り返しが終わった後に実行される。ただしループを break で抜けた場合は実行**されない**)。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-

def while_test(x):
    while x > 0:
        print x,
        if x == 1.5: break
        if x == 1: x += 3; continue
        # if文の本体が短い場合はこのような書き方も可(while, for, 関数定義などでも)
        x -= 2
    else:
        print 'Boom!'   # breakで抜けるとここは実行されない

while_test(6)
while_test(5)
while_test(5.5)
```

実行結果

```
6 4 2 Boom!
5 3 1 4 2 Boom!
5.5 3.5 1.5
```

ループを抜けるだけでなくプログラムの実行そのものを終了させたい場合は、5.1 節に出てきた sys.exit を使えばよい。

5.3 for 文

C 言語と同様の for 文は Python にはない (while 文で代用する)。Python にある for 文は、「リストやタプルなどの**各要素に対し**繰り返す」というもので、シェルスクリプトの for 文に似る。(awk や Perl などには両方の for 文がある)

```
#!/usr/bin/python
# -*- coding: euc-jp -*-

for x in [4, 1, 9]: # リストの各要素を1つずつ変数xに代入しながら繰り返す
    print x,
print ''        # 空文字列をprintすることにより、ここで1回改行を出力

h = {'c' : 10, 'mn' : 50, 'a' : 1, 'b' : -1}
for k in h: # 辞書の各キーを1つずつ変数kに代入しながら繰り返す
    print 'key: %s value: %d' % (k, h[k])
```

このプログラムは以下の出力をする。辞書に対して for 文を使う場合は、キーは**順不同**で取り出されることに注意 (リストやタプルの場合は、**先頭の要素から**順に取り出される)。

```
4 1 9
key: a value: 1
key: c value: 10
key: mn value: 50
key: b value: -1
```

なお、while 文と同様の break, continue, else は for 文にもある。

for 文とは関係ないが、4.3.1 節に述べたように、括弧類の中では自由に改行できる。プログラム中に長いリストなどを書く場合は、このことを利用して例えば

```
for x in [
    4,
    1,
    9,
]:
```

のように書く方が見やすいことがある (また、リスト･タプル･辞書の最後の直前に 1 つ余分に「,」があってもよいことになっているので、この例ではそのことも使っている)。

[*enumerate* 関数] リストなどの各要素を取り出すだけでなく、それが「**何番目の要素か**」も考えながら処理したい場合は、enumerate 関数を用いて次のようにする。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-

a = ['wa', 'ha', 'fu', 'hyo']
for x in enumerate(a):
    print '%d: %s' % (x[0], x[1])    # print '%d: %s' % x でもOK
    if x[0] == 2: break
```

実行結果

```
0: wa
1: ha
2: fu
```

この場合、変数 x に「a の何番目の要素か」を表す整数と a の実際の要素からなる 2 要素のタプルを毎回代入しながら繰り返す。a の要素だけ取り出したければ、x[1] を取ればよい。あるいは、上の例の for 文の箇所を

```
for (n, x) in enumerate(a):      # nに「何番目か」、xに要素が入る
    print '%d: %s' % (n, x)
    if n == 2: break
```

のように書くこともできる (実行結果は同じ)。

enumerate 関数は、「イテレータ」というものを返す関数の例である (enumerate 関数がイテレータなのではなくて、enumerate(a) の呼び出しで返される値がイテレータ)。イテレータとは、リストやタプルのような配列に似たデータではないが、「for 変数 in イテレータ:」と書けば、ループの毎回の繰り返しのたびに、変数に代入する値を何らかの規則で生成してくれる、特別なデータである (とここでは説明しておく)。enumerate の他には、4.3.1 節に登場した finditer はイテレータを返すメソッドの例。また、8 節に登場する「ファイルオブジェクト」も、イテレータとしても扱える。

[*xrange* 関数] Python で、あらかじめ決められた回数だけループしたい場合によく使われるのは、xrange 関数 (Python3 では range。なお、Python2 の range は違う意味の関数) である。「for 変数 in xrange(n)」(n は非負整数) とすると、変数に 0 から $n-1$ までの整数を順番に代入しながらループする[9](ので、n 回ループすることになる。なお、xrange には 2 引数や 3 引数で呼ぶ使い方もある)。次のプログラムは「0 1 2 3 4」と出力する。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-

for x in xrange(5):
    print x,
print ''        # 空文字列をprintすることにより、最後に1回改行を出力
```

5.4 例外処理

Python では、ファイル (8 節) がオープンできないなど何らかのエラーが起きた場合、そのままでは、その時点で問答無用で**プログラムが終了**してしまう。例えば

```
#!/usr/bin/python
# -*- coding: euc-jp -*-
a = open('xyz')      # xyzというファイルは存在しないとする
print 'Ok'
```

[9] xrange(n) が返す値はイテレータではないが (イテレータとはいくつかの点で異なる)、繰り返しのたびに変数に代入する値を生成するという点ではイテレータと似た点がある。

というプログラムを実行すると

```
Traceback (most recent call last):
  File "zzz", line 3, in <module>
    a = open('xyz')
IOError: [Errno 2] No such file or directory: 'xyz'
```

open (ファイルのオープン、8 節参照) の時点でプログラムはエラー終了しており、後続の「print 'Ok'」の部分は実行されていない。

5.4.1 try～except

エラーが起きた場合に何らかの処理をしてプログラムの実行を継続したい場合は、**例外処理**を書かねばならない。Python には、try～except という例外処理用の構文がある。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-
import sys

fname = 'xyz'
try:
    a = open(fname)     # xyzというファイルは存在しないとする
except IOError as err:      # Python 2.5までは「as」でなく「,」と書く
    print >>sys.stderr, err
else:
    print '正常にオープンできた'
print 'Ok'
```

実行結果

```
[Errno 2] No such file or directory: 'xyz'
Ok
```

今度はエラーが起きてもプログラムは続行する。

基本的には「except エラー名 as 値」のように記述。「エラー名」には捕捉したいエラーの種別を書く。代表的なエラーには、ここで使った IOError の他、ArithmeticError (0 で割るなどの算術エラーで発生) や、OSError (os モジュールの各種関数で発生) がある。どの操作でどのエラーが起き得るかは、マニュアルなどであらかじめ調べておく。なお、エラーが (expect に指定しなかったものも含め) 全く起きなかった場合は、else 部 (省略可) が実行される。

「as」の次の「値」のところには変数名 (上の例では「err」) を書いておく (「as 値」を省略することも可能)。エラーが捕捉された場合、この変数には、捕捉されたエラーに関するいろいろな情報が入っており、タプルのように添字を指定してそれらの情報を個別に取り出したりすることもできる (ただし、どの添字の要素がどんな情報であるかはエラーの種類によって異なる) のだが、とりあえず上の例のようにそのまま print で出力すれば、エラーの情報を知らせるメッセージ (上の例では「[Errno 2] No such file or directory: 'xyz'」) として使える。次は ArithmeticError を発生させる例。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-
import sys

try:
    a = 1 / 0
except ArithmeticError as err:
    print >>sys.stderr, err
print 'Ok'
```

実行結果

```
integer division or modulo by zero
Ok
```

実際には、0 での割り算で発生するエラーは ZeroDivisionError というエラーなのだが、上記のように「except ArithmeticError …」と書いてもこのエラーを捕捉できる (「except ZeroDivisionError …」でもできる)。この理由は、ArithmeticError が ZeroDivisionError の上位のエラー (ZeroDivisionError を含む、より広い範囲のエラー) であるためである。

複数のエラーを 1 つの except で捕捉したい場合は、「except (IOError, OSError) as err:」のようにエラー種別のタプルを書く。また、1 つの try に対し複数の except を書くこともできる。

5.4.2 try～finally

「例外処理」とはちょっと違うが、「**後始末**」を行うための構文 try～finally もある。下の例は、ファイルの 1 行目だけを出力するプログラムを、わざと try～finally を使って書いたもの。なおこの例では、8 節に出てくる「ファイルオブジェクト」に関する知識を使っている。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-
import sys

def test(fname):
    try:
        f = open(fname)
    except IOError:     # as以後を省略した例
        print >>sys.stderr, 'Cannot open', fname
        sys.exit(1)
    try:
        s = f.readline()    # オープンされたファイルfから1行読む
        s = s.rstrip('\n')  # readline関数は読んだ行の末尾の改行を除去しない
                            # ので、末尾の改行を除去したければこうする
        print s             # sの末尾の改行を除去せず「print s,」とするのでも可
        return
    finally:
        f.close()
        print >>sys.stderr, fname, 'is closed'

test('abc')     # ファイルabcは存在するとする
```

こうすると、try～finally の try 部から抜ける際には必ず finally 部を実行する (たとえ try 部で例外が発生したり、return で関数から抜けたりしても) ので、ファイルのクローズ忘れを防げる[10]。実行すると

```
…ファイルabcの1行目のみ…
abc is closed
```

のような出力を得る。

try～expect につく else と、try～finally の違いは、else 部は例外が発生しなかった場合のみ実行されるが、finally 部は例外が発生したか否かに関わらず必ず実行される点である。

Python 2.5 以後は 1 つの try に except (else 含む) と finally を**両方**付けることができる (finally の方を後に書く)。

5.4.3 raise

raise 文で、「わざとエラーを発生させる」こともできる。

[10] もっとも、現時点の Python の標準的な処理系の実装は、ファイルオブジェクトへの参照が失われたらそのファイルを自動的にクローズするようになっているので、明示的にクローズしなくても必ずしも支障は起きない。しかし、これは Python の言語仕様として保証されたことではないのだそうであり、Python の公式ドキュメントでは「ファイルは必ず閉じてください」としている。ファイルの利用終了後もプログラムが継続する場合は、ファイルを明示的にクローズする方がよいだろう。
ただし、ファイルの利用終了とともにプログラムも終了する場合は、(Python プログラムに限らず) 明示的にファイルをクローズしなくても、よっぽどできの悪い OS でない限り、OS がファイルをクローズしてくれるはずである (組み込みシステムなどの場合にはそうでないこともありうるが)。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-
import sys

try:
    raise ArithmeticError('fake arithmetic error')
except ArithmeticError as err:
    print >>sys.stderr, err
print 'Ok'
```

実行結果

```
fake arithmetic error
Ok
```

ここでは既存の種類のエラー (`ArithmeticError`) を発生させたが、raise 文は多くの場合、自分で新たな種別のエラーを定義 (方法略) して、それを自作モジュールの中で発生させる目的で使う。

6 関数定義

関数定義には def 文を使う。

[*ローカル変数*] 関数定義の中で使われる変数 (仮引数も含む) は、原則として自動的に**ローカル変数**になる (global 宣言によりグローバルにすることもできる)。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-

x = 1; y = 2

def yonjo(x):
    y = x*x
    return y*y

print yonjo(3),      # 出力後に改行しない
print x, y
```

実行結果は「`81 1 2`」となり、yonjo 関数の呼び出し後、関数定義の外の x と y は変わっていない。

[*参照渡し*] 関数の引数に**リストなど**を渡す場合、それは「**参照渡し**」になる。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-

def test(x, y):      # この関数は値を返さない
    x = 3; y[0] = 4

a = 2; b = [5, 6]
test(a, b)
print a, b
```

このプログラムの出力は「`2 [4, 6]`」となり、a の値は変わらないが、b の第 0 要素は変わっている。

[*返り値の型*] 関数の返り値の型は自由である (上のように、値を返さない関数も書ける)。特に、タプルなどを返すことによって、事実上複数の値を返すこともできる。また、場合によって返り値の型が違うこともある。
次のプログラムは「`3 (-3, 3)`」と出力する。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-

def f(x):
    if x >= 0:
        return x
    else:
        return (x, -x)

print f(3), f(-3)
```

[*他の構文の中での関数定義*] 関数定義は (C 言語と違って) 文の 1 種なので、if 文や他の関数定義の中などに入れられる。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-

def f(x):
    global g
    if x >= 0:
        def g(x):
            return x*x
    else:
        def g(x):
            return x*x*x

f(-3)
print g(2)
```

このプログラムでは、関数 g の定義としては 2 つのうち下の方のものが使われるため、出力は「8」となる。

この場合、`global g` 宣言がないと、関数 g は関数 f の中の**ローカル関数**になり、f の外から呼べなくなる (もちろん、その方がありがたいこともある。関数ローカルな関数を作りたいことはよくあることである)

6.1 データとしての関数

Python では関数もデータ (**オブジェクト**) の 1 種である。例えば、次のようにリストや辞書などに入れて使える。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-

def f1(x):
    return x*x

def f2(x):
    return x*x*x

def f3(x):
    y = x*x
    return y*y

fh = {'nijo': f1, 'sanjo': f2, 'yonjo': f3}

print fh['yonjo'](3),    # 改行しない

fn = fh['sanjo']
print fn(2)
```

実行結果は「81 8」となる。この例では fn 関数は f2 関数と同じものになっていることに注意。

6.2 キーワード引数

Python では関数への引数として「**キーワード引数**」というものが使える。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-

def f(height = 170, weight = 60, bust = 72, seating_height = 120):
    print 'H=%dcm W=%dkg B=%dcm S=%dcm' % (
        height, weight, bust, seating_height)

f()
f(seating_height = 115, weight = 62)
```

実行結果

```
H=170cm W=60kg B=72cm S=120cm
H=170cm W=62kg B=72cm S=115cm
```

キーワード引数とそうでない引数の混用も可能 (略)。また、この他には可変長引数も使えるが、これも略。

6.3 ドキュメント文字列

関数定義の本体の最初 (注釈除く) に文字列を書くと、それは「その関数の説明」として扱われる。

```
def yonjo(x):
    '引数の4乗を返す関数'   # これがドキュメント文字列
    y = x*x
    return y*y
```

ドキュメント文字列は、実行時には特に使われないが、「関数名.__doc__」で取り出すことや、Python のヘルプシステムでの利用 (本資料では略) などができる。ドキュメント文字列は複数行にわたることが多いので、ブロック文字列 (4.3 節) で書かれることが多い。

クラス定義 (9 節) にもドキュメント文字列を書ける。

7 実行時の引数

Python プログラムの実行時に引数を与えると、それは **sys.argv** というリストに入る (import sys が必要)。C 言語と同様、sys.argv[0] はプログラムファイルの名前で、プログラムの引数はそれ以降の要素。

```
$ cat fuhyo↵
#!/usr/bin/python
# -*- coding: euc-jp -*-
import sys
print sys.argv

$ ./fuhyo foo bar baz↵
['./fuhyo', 'foo', 'bar', 'baz']
```

sys.argv は文字列のリストなので、その他の型 (数値など) として使いたければ型変換をしなければならない (文字列から整数や実数への変換は 4.3 節に出てきた)。

```
$ cat fuhyo↵
#!/usr/bin/python
```

```
# -*- coding: euc-jp -*-
import sys
print int(sys.argv[0]) + 2
```

```
$ ./fuhyo 3⏎
5
```

8 ファイル

ファイルをオープンするには、基本的には組み込み関数 open を「open(ファイル名, モード)」のようにして使う。モードとして指定できる文字列には「'r'」「'w'」「'a'」「'r+'」「'w+'」「'a+'」などがあり、それぞれ C 言語の fopen 関数と意味は同じ。モードを**省略**すると、「'r'」を指定したのと同じになる。

open 関数は、「**ファイルオブジェクト**」を返す。ファイルを読み書きするには、このオブジェクトの**メソッド** read, readline, readlines, write などを呼ぶ。特に、一定バイト数読むには read、一行読むには readline、ファイル全体を行単位で一括して読んでリストにするには readlines を呼ぶ。クローズするにはファイルオブジェクトの close メソッドを呼ぶ。

次の例は、引数を 2 つ取り、第 1 引数のファイルの各行を読み出してそれを「,」で分割し、その第 1·3·4 欄目 (先頭を 0 と数えると第 0, 2, 3 欄目になる) をやはり「,」で区切りながら第 2 引数のファイルに書き出す、というもの[11]。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-
import sys

# 引数の個数が2個でないとエラー終了
if len(sys.argv) != 3:
    print >>sys.stderr, 'Usage: %s src dest' % sys.argv[0]
    sys.exit(1)
# 第1・2引数を変数infname, outfnameに格納
(infname, outfname) = sys.argv[1:3]

# ファイルをオープン
try:
    inf = open(infname)
    outf = open(outfname, 'w')
except IOError as err:
    print >> sys.stderr, err
    sys.exit(1)

while True:
    # infから1行読む(行末の改行は除去されない)
    str = inf.readline()
    # ファイルの終わりに達しているときはreadline()は空文字列を返すので
    # その場合はループ脱出
    if str == '': break     # if not str: break でもOK(空文字列は偽扱いなので)

    # strの末尾の改行を除去
    str = str.rstrip('\n')

    # strを「,」で分割し、リストとしてaryに入れる
    ary = str.split(',')
    # 空のリストを用意
    newary = []
    for i in [0,2,3]:
        if i < len(ary): # aryの第0, 2, 3要素が存在するなら
```

[11] このプログラムではファイルのクローズを明示的に行っているが、注釈 10 でも述べたように、基本的にはプログラムの終了時にはファイルもクローズされるはずなので、このプログラムの場合はあえて明示的にクローズしなくても支障は生じない。

```
                # それをnewaryに追加
                newary.append(ary[i])
        # newaryの要素を「,」で連結し、改行をつけてoutfに出力
        outf.write('%s\n' % ','.join(newary))
    # 最後にファイルをクローズ
    inf.close()
    outf.close()
```

このプログラムの名前を csvpick とし、ファイル abc に以下の内容

```
abc,def,ghi,jkl,mno
あ,い,う
12,3,45,678,90
```

が入っているとする。

```
$ csvpick abc def↵
```

を実行すると、ファイル def の内容は

```
abc,ghi,jkl
あ,う
12,45,678
```

となるだろう。

ファイルオブジェクトに対する各種メソッド (write, readline など) は、標準入出力や標準エラー出力 (sys.stdin, sys.stdout, sys.stderr) にももちろん使える。特に、raw_input (5.1 節) と同じようなことを sys.stdin.readline で行ったり (後者では行末の改行が削除されないので注意)、print の代用に sys.stdout.write を使ったりできる。

ただし、write で出力する場合、print と違って引数は文字列 1 つしか取れない。それ以外を出力したい場合は、文字列フォーマッティング (4.3 節) を用いて文字列に変換してから出力する必要がある。また、write は print と違って出力の最後に勝手に**改行**を付加しない。

逆に、write の代用に「`print >>出力先`」の形を使うこともできる。この書き方は、出力先として sys.stderr だけでなく、open でオープンしたファイルも指定できる。例えば上のプログラムの `outf.write(···)` のところを、`print >>outf, ','.join(newary)` とも書ける。この場合、print なので出力の最後で自動的に改行される。

ちなみに、os モジュールには、UNIX の**システムコール** open や close などをそのまま提供している os.open, os.close などの関数もある。これらは上記のものとは異なる。

[*イテレータとしてのファイルオブジェクト*] 5.3 節で述べたように、(読み出しオープンした) ファイルオブジェクトは「イテレータ」として扱うこともできる—すなわち **for 文**の繰り返し対象として「`for 変数 in ファイルオブジェクト:`」の形で書ける。その場合、変数にはファイルから 1 行ずつ読んだ文字列が代入される。よって、例えば上のプログラムの中 (注釈を除去してある) の

```
while True:
    str = inf.readline()
    if str == '': break
```

のところを

```
for str in inf:     # ファイルオブジェクトinfをイテレータとして使用
```

に変えても同じ動作をする。ただしこの方法が使えるのはあくまで、読んだ**各行**に対し**繰り返し処理**を行う場合のみ。

8.1 with 文

プログラム内でファイルを一時的に利用した後にクローズするのを忘れることを防ぐには、5.4.2 節に出てきた try～finally を使う方法の他に、with 文を使う方法もある (Python 2.6 以降で利用可。2.5 でも with_statement モジュールを import すれば利用可)。

with 文は、何らかのデータを利用する前あるいは後に、あらかじめ決められた定形の前･後処理を必ず行わせたい場

合に使える構文。ファイルオブジェクトに対して使うと、「ファイルの利用終了後に必ずクローズする」という処理が行われる。例えば、5.4.2 節のプログラムの try ～ finally の部分と同じことは、with 文では

```
with f:
    s = f.readline()    # オープンされたファイルfから1行読む
    s = s.rstrip('\n')  # 読んだ行の末尾の改行を除去
    print s
print >>sys.stderr, fname, 'is closed'
```

のように書ける (最後の print 文は with 文の外)。この with 文を抜けるときに f は自動的にクローズされるので、f を明示的にクローズする必要がない。

あるいは、5.4.2 節のプログラム全体を以下のように書き直すこともできる (try でエラーがキャッチされる範囲が with 文全体に変わることを除き、実行結果は 5.4.2 節のプログラムと同じ)。この使い方は、with 文でオープンから自動クローズまでやってしまえるので、プログラム中の短い間でファイルを利用したい場合に便利。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-
import sys

def test(fname):
    try:
        with open(fname) as f:
            s = f.readline()    # オープンされたファイルfから1行読む
            s = s.rstrip('\n')  # 読んだ行の末尾の改行を除去
            print s
    except IOError:
        print >>sys.stderr, 'Cannot open', fname
        sys.exit(1)
    print >>sys.stderr, fname, 'is closed'

test('abc')     # ファイルabcは存在するとする
```

9　クラス

クラスの定義は「`class クラス名(親クラス名, 親クラス名 ...):`」で始める。Python では多重継承 (親クラスを複数持つ) が使える (ただし、多重継承は**極力避ける**方がよい)。

親クラスがない場合、単に「`class クラス名:`」でもいいが、これは「旧スタイルクラス」と呼ばれ、少し制限がある (子クラスから自分のクラスのメソッドを呼べなくなる)。「`class クラス名(object):`」とすれば (最上位のクラスである「object」を親として宣言する)、「**新スタイルクラス**」というものになる (Python 3.0 からは前者の書き方でも「新スタイル」になる。あるいは、Python 2 であっても、クラスの定義より前に「`__metaclass__ = type`」という行を書いておけば、前者の書き方でも「新スタイル」になる)。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-

class xy_coord(object):
    def __init__(self, x, y): # コンストラクタ
        self.x = x
        self.y = y
    def vec_add(self, v): # メソッドの定義
        self.x += v.x
        self.y += v.y
    def midpoint(self, v):
        self.vec_add(v)
        self.x /= 2
```

```
        self.y /= 2

a = xy_coord(3, 8) # インスタンスの生成。Javaと違って「new」とは書かない
print a.x, a.y
b = xy_coord(1, 2)
a.midpoint(b) # メソッドの呼び出し
print a.x, a.y
```

実行結果

```
3 8
2 5
```

クラス内のメソッド (コンストラクタを含む) 定義の**第 1 引数**には必ず、そのメソッドを実行するインスタンス (コンストラクタの場合は、そのコンストラクタによって作成されたばかりのインスタンス) が**自動的に**渡される。これを受ける第 1 引数変数の名前は、慣習的に「`self`」とする。ただし、**呼び出す側**では第 1 引数にインスタンスは書かない (よって、呼び出す側での引数の個数はメソッド定義の引数の個数より **1 つ少なくなる**)。なお、コンストラクタメソッドの名前は「`__init__`」と決められている。

クラス内のメソッド定義の中で、そのクラス内の変数 (Java でいうフィールド) やメソッドにアクセスしたい場合は「`self.`」を付ける。それらにクラス外からアクセスする場合は「`インスタンス名.`」を付ける (ただし、**インスタンス変数**と**クラス変数**の区別についての詳細は 9.1 節参照)。

Python の場合、Java とは違って、1 つのクラスに引数の異なる同名のメソッド (コンストラクタを含む) を複数定義すること (メソッドのオーバーロード) は**できない** (同名のメソッドの定義を複数書くと、後のものが先のものを上書きする)。どうしてもそれをしたければ、メソッドは 1 つだけ書き、そのメソッドを可変長引数にしたり、メソッド定義の中で引数の型を判定 (5.1.1 節に出た isinstance 関数を使う) したりする。

また、Python ではクラス定義は**実行文**なので、クラス定義を入れ子 (クラス定義中に内部クラス定義を書く) にしたり、if 文などの中に入れたりできる。

9.1 クラス変数とインスタンス変数

Python では、クラス定義の中 (メソッドの外) で代入された変数は**クラス変数**になる (グローバル変数を除く)。また、クラス変数には「`クラス名.変数名`」でもアクセスできる。

インスタンス変数を作るには、メソッドの中で「`self.変数名`」に値を代入するか、あるいはクラス外で「`インスタンス名.変数名`」に値を代入する。

同じ「`self.変数名`」あるいは「`インスタンス名.変数名`」という記法でも、その名のインスタンス変数が作られる (代入される) までは、それはクラス変数を指すので注意。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-
class a:
    n = 1           # クラス変数への代入
    def f(self):
        self.n = 4  # インスタンス変数への代入
    def g(self):
        a.n = 5     # クラス変数への代入

p = a()
q = a()
r = a()
print a.n, p.n, q.n, r.n    # この時点では4つともクラス変数
p.n = 2                     # ここでp.nはインスタンス変数になる
print a.n, p.n, q.n, r.n
a.n = 3                     # これはクラス変数の値の変更
```

```
print a.n, p.n, q.n, r.n
q.f()                       # ここでq.nもインスタンス変数になる
print a.n, p.n, q.n, r.n
r.g()                       # クラス変数の値変更
print a.n, p.n, q.n, r.n
```

実行結果

```
1 1 1 1
1 2 1 1
3 2 3 3
3 2 4 3
5 2 4 5
```

上の例では、値が 1 か 3 か 5 になっているのがクラス変数、2 か 4 になっているのがインスタンス変数である。

この例のメソッド g の定義で、「a.n = 5」のようにクラス名を直接使うと、クラスの名前が a から他のものに変わった場合に、そこも**書き換えねばならない**。これに代えて

```
self.__class__.n = 5
```

とすれば、クラス名の変更に合わせて書き換える必要がなくなる。「インスタンス.__class__」でそのインスタンスの属するクラスを取り出せるため。

9.2 クラスメソッド

Python では、クラス内に普通に書いたメソッドはインスタンスメソッドになる。**クラスメソッド**を作りたい場合は、メソッド定義の直前の行に「@classmethod」と書く[12]。この場合、メソッドの第 1 引数にはクラスが自動的に渡される。呼び出す側では第 1 引数にクラスは書かない。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-
class a:
    x = 1
    @classmethod
    def f(cls, x): # これはクラスメソッド
        cls.x = x # クラス変数xに引数xを代入
    def g(self, x): # こちらはインスタンスメソッド
        self.x = x

b = a()
print a.x, b.x
b.f(2)  # クラス変数a.xが変更される
print a.x, b.x
a.f(3)  # これもクラス変数a.xが変更される
print a.x, b.x
b.g(4)  # ここでインスタンス変数b.xができ、値が4になる
print a.x, b.x
```

実行結果

```
1 1
2 2
3 3
3 4
```

[12] これに代えて、メソッド定義の直後に「f = classmethod(f)」と書いても同じ効果を持つ (classmethod は Python の組み込み関数で、クラスメソッドを作り出す関数)。これの省略形が「@classmethod」である。

9.3 アクセス制限

Python のクラス内のメソッドや変数に対するアクセス制限は、Java の private · protected · public ほどきめ細かく設けられてはいない。基本的には以下の規則で**アクセス制限**が決まる。

1. クラス内の変数名やメソッド名が「__」で始まり「__」で終わる場合、それは Python システムにとって特別な変数またはメソッドとして使われることが多く、普通の変数やメソッドにそういう名前を付けることは好ましくない (その例として、コンストラクタである `__init__()` や、Java の toString にほぼ相当する `__str__()`、Java の equals にほぼ相当する `__eq__()` などがある)。
2. 上記以外で、クラス内の変数名やメソッド名が「__」で始まる場合、その変数やメソッドはクラス外からは**アクセスできない**[13](クラス変数 · メソッドであってもインスタンス変数 · メソッドであっても)。
3. 上記以外で、クラス内の変数名やメソッド名が「_」で始まる場合、その変数やメソッドは特にクラス外からアクセスできないわけではないが、プログラマ側の一般的マナーとしてクラス外からはアクセスしないことになっている。

9 節の例を、インスタンス変数 x や y を隠蔽するように変えるには、以下のようにする。この例のように、外からアクセスする必要のない変数 (やメソッド) は外部からは隠蔽すべきである。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-

class xy_coord(object):
    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    def get_x(self): return self.__x
    def get_y(self): return self.__y

    def vec_add(self, v):
        self.__x += v.get_x()
        self.__y += v.get_y()
    def midpoint(self, v):
        self.vec_add(v)
        self.__x /= 2
        self.__y /= 2

a = xy_coord(3, 8)
print a.get_x(), a.get_y() # a.__xやa.__yには外からアクセスできない
b = xy_coord(1, 2)
a.midpoint(b)
print a.get_x(), a.get_y()
```

9.4 継承

親クラスを持つクラス (子クラス) では、自動的に親クラスの変数やメソッドを使える (**継承**)。ただし、子クラス側で同名のメソッドを定義すると、親クラスのものは (super を使って明示的に呼ばない限り) 隠される (オーバーライド)。なお、Java と違い、コンストラクタも継承される。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-
```

[13]実際には、特別なルールを知っていてそれを使えばアクセスはできるが、わざわざそうしない限りアクセスはできないので、実質的に十分なアクセス制限にはなっている (と Python コミュニティの人々は考えている)。

```
class a(object):
    x = 10
    def __init__(self, x):
        self.x = x
    def f(self, x):
        return x+self.x
    def g(self, x):
        return x/self.x
    def h(self, x):
        return x*self.x
    @classmethod
    def i(cls, x):
        return x-cls.x
class b(a): # クラスbはaの子クラス
    def g(self, x): # 親クラスのg()をオーバーライド
        return x % self.x
    def h(self, x):
        return super(self.__class__,self).h(x)+1
        # 親クラスのインスタンスメソッドの呼び出し方
    @classmethod
    def i(cls, x):
        return super(cls,cls).i(x)+1
        # 親クラスのクラスメソッドの呼び出し方

c = b(2) # インスタンス変数c.xが2になる
print c.f(3), c.g(4), c.h(5), b.i(6)
```

実行結果は「`5 0 11 -3`」となる。

Python3.0 からは、super() 関数の引数は省略可能になる。

Python には、Java のインタフェースや抽象クラスにあたるものは提供されていない。ただし、**抽象クラス**にほぼ相当することを以下のように実現することはできる (残念ながら未実装は実行時にしか検出できない)。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-
class a(object):
    def f(self, x): # 子クラスでの実装を要求
        raise NotImplementedError
        # NotImplementedErrorは「実装されていない」を表すPythonの
        # 組み込みエラーで、これを発生させる
class b(a): # aの子クラス
    def f(self, x):
        return x*x
c = b() # ここが c = a() だと下の行でNotImplementedErrorが発生する
print c.f(3)
```

9.5 ちょっと (だけ) 大きな例題

次の例は、命題論理の論理式の真偽値を、解釈関数 (真理値割り当て) を与えて求めるもの。formula クラスが論理式を表す抽象クラスで、その子クラスが原子論理式 (命題記号のみからなる論理式) や $\neg\phi$, $\phi \wedge \psi$ などそれぞれの形の論理式を表す。論理式を表すクラスには、引数として解釈関数を受け取って、その論理式の真偽値を返すメソッド truth_value が用意されている。解釈関数は、命題記号 (命題変数) をキーとし True, False を値とする辞書で表現する。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-
```

```
# 命題論理の論理式の真偽値を、解釈関数を与えて求める

class formula(object): # 論理式を表す抽象クラス
    def truth_value(self, interpret):
        raise NotImplementedError
    # interpretには命題変数名をキー、真偽値を値とする辞書を渡す

class atomic_formula(formula): # 原始論理式(i.e. 命題変数だけ)
    def __init__(self, varname): # 引数は命題変数名
        self.__varname = varname
    def truth_value(self, interpret):
        return interpret[self.__varname]
        # interpretにvarnameの解釈が与えられていなかったら
        # ここでエラーが発生する

class not_formula(formula): # ¬φの形
    def __init__(self, subformula):
        self.__subformula = subformula
    def truth_value(self, interpret):
        return not self.__subformula.truth_value(interpret)

class and_formula(formula): # φ∧ψの形
    def __init__(self, subformula1, subformula2):
        self.__subformula1 = subformula1
        self.__subformula2 = subformula2
    def truth_value(self, interpret):
        return self.__subformula1.truth_value(interpret) and \
            self.__subformula2.truth_value(interpret)

class or_formula(formula): # φ∨ψの形
    def __init__(self, subformula1, subformula2):
        self.__subformula1 = subformula1
        self.__subformula2 = subformula2
    def truth_value(self, interpret):
        return self.__subformula1.truth_value(interpret) or \
            self.__subformula2.truth_value(interpret)

class imp_formula(formula): # φ⊃ψの形
    def __init__(self, subformula1, subformula2):
        self.__subformula1 = subformula1
        self.__subformula2 = subformula2
    def truth_value(self, interpret):
        return not self.__subformula1.truth_value(interpret) or \
            self.__subformula2.truth_value(interpret)

i = {'p': True, 'q': False} # 解釈関数 pが真、qが偽
f = or_formula(
    and_formula(
        not_formula(atomic_formula('p')),
        atomic_formula('q')
    ),
    and_formula(
        atomic_formula('p'),
        not_formula(atomic_formula('q'))
    )
) # fには論理式 (¬p∧q)∨(p∧¬q) を表すインスタンスが代入される
```

```
print f.truth_value(i) # Trueが出力される
```

注: 行末の「\」は**継続行**を表す (Python では基本的に文は行で区切られるため、一文が次の行に跨る場合は「\」で継続せねばならない)。ただし**括弧類の中**は、4.3.1 節にも述べた通り「\」なしでも行が継続する。上記の例の、変数 f への代入はそれを用いて、関数の引数が長くなる場合の書き方を工夫したものである。

10　モジュール

大規模開発になると、一まとまりの機能を「**モジュール**」とし、各モジュールの他からの独立性を高めることが求められる。

例えば、9.3 節の xy_coord クラスを、coord という名前のモジュールとして独立させたいとする。

まず、以下のような名前の coord.py ファイルを作る。このファイルはもはや単独の実行には供用しないので、先頭の「`#!`」行は不要、実行可能属性も付けなくてよい。

```
# -*- coding: euc-jp -*-

class xy_coord(object):
    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    def get_x(self): return self.__x
    def get_y(self): return self.__y

    def vec_add(self, v):
        self.__x += v.get_x()
        self.__y += v.get_y()
    def midpoint(self, v):
        self.vec_add(v)
        self.__x /= 2
        self.__y /= 2

if __name__ == '__main__':
    # テスト用コード; このファイルを単独で python coord.py として実行した
    # 場合に限りここが実行される。このファイルがモジュールとして呼び出さ
    # れた場合ここは実行されない。テスト用コードを特に必要としない場合は
    # ここは書かなくてよい
    a = xy_coord(3, 8)
    print a.get_x(), a.get_y()
    b = xy_coord(1, 2)
    a.midpoint(b)
    print a.get_x(), a.get_y()
```

次に、coord.py ファイルをモジュールとして使うプログラム、pymodtest を作る。coord モジュールを使うには「`import coord`」が必要。このようにしてモジュール (自作でもあらかじめ用意されているものでも) を読み込める。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-

import coord

a = coord.xy_coord(3, 8) # coordモジュールのクラスはこうやって呼び出す
print a.get_x(), a.get_y()
b = coord.xy_coord(1, 2)
a.midpoint(b)
```

```
print a.get_x(), a.get_y()
```

さて、この場合 coord.py ファイルはどこに置けばよいか、というと、以下のどこかに置けばよい (**この順に探される**)。

1. pymodtest ファイルと同じディレクトリ (つまり、メインの Python プログラムのファイルと同じディレクトリ)
2. 環境変数 PYTHONPATH に挙げられているディレクトリ (「:」区切り)
3. Python の標準のモジュールが納められているディレクトリ (`/usr/lib/python2.6` (`2.6` の部分は Python のバージョン番号) など多数。Python のインストール時に決まる)。

ただし、このうち 3. はシステムの全ユーザで共用するので、私用のモジュールは置くべきではない。私用のモジュールは 1., 2. のどちらかに置くとよいだろう。ちなみに、このリスト (1., 2., 3. の全て) は Python の変数 sys.path にも収められている。

そこで、例えば PYTHONPATH に /home/nide/lib/python (自分が書き込みできるディレクトリを選ぶ) を設定しておいて、coord.py をそこに置き、

```
$ ./pymodtest↵
```
(pymodtest がカレントディレクトリにあるとして)

とすると、無事実行できて

```
3 8
2 5
```

という出力を得る。

ちなみに、モジュールにはクラス定義だけではなく、(クラス定義を伴わない) 関数定義や変数代入を置くこともできる。

[*from* 型インポート] 「`import coord`」でなく「`from coord import *`」と書くと、coord モジュールのクラスや関数を、いちいち `coord.` を付けずに呼べるようになる (これは、自分で作ったモジュールだけでなく、標準のモジュール、例えば sys や math など、でも同様。4.1 節にも登場した)。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-

from coord import *

a = xy_coord(3, 8) # coordモジュールのクラスを呼び出す
print a.get_x(), a.get_y()
b = xy_coord(1, 2)
a.midpoint(b)
print a.get_x(), a.get_y()
```

しかし、これに頼り**すぎる**のは好ましくない。異なるモジュール間での、クラスや関数の名前の衝突が起きやすくなるからである。

10.1 .pyc ファイル

Python があるモジュールの .py ファイルを見つけて読み込む場合、自動的にそれをコンパイルして、 .pyc ファイルを作る (そのディレクトリあるいはファイルに書き込み権限がない場合を除く)。コンパイルといっても機械語にするわけではなく、次回に Python インタプリタが読み込みやすい (高速に読める) ような内部形式へのコンパイルである。なお、.py ファイルがコンパイルされるのは**モジュールとして読み込まれた場合だけ**で、そうでない場合 (メインのプログラムの Python ファイル) はコンパイルされない。

一度 .pyc ファイルが作られると、次回からは .py ファイルは読まれず、.pyc ファイルの方だけが読まれる (ただし、.py ファイルが更新されて .pyc ファイルより新しくなっている場合は、再度 .py ファイルの方が読まれてコンパイルされる)。

例えば 10 節の例では、一度 Python が /home/nide/lib/python/coord.py を見つけて読み込むと、コンパイルされて/home/

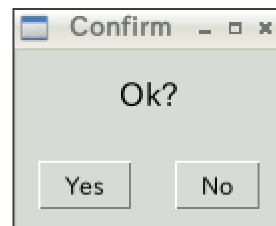
nide/lib/python/coord.pyc が作られる。

なお、Python3 の場合は、モジュールのファイルがあるディレクトリに `__pycache__` というディレクトリが作られ、そこにコンパイルされた .pyc ファイルが書き込まれる。

11 番外: Tkinter

Python にはあらかじめいろいろなモジュールが提供されている。その 1 つとして、Python から Tk の機能を使う Tkinter というモジュールがあり[14]、これを用いれば Python で **GUI** が書ける。

```
#!/usr/bin/python
# -*- coding: euc-jp -*-

# Tkinterを使用 (sysも使用)
import Tkinter, sys

# メッセージを出力してexit
def msg_exit(msg, val):
    print >>sys.stderr, msg
    sys.exit(val)

# ルートフレームを生成
root = Tkinter.Tk()
root.title('Confirm')

# メッセージウィジェットを生成
msg = Tkinter.Message(
    root,
    text = 'Ok?',
    justify = 'center',
    width = 160,
    font = '-*-fixed-medium-r-normal--16-*'
)
msg.pack(pady = 10)
# Yes/Noのボタンウィジェット生成
yes = Tkinter.Button(
    root,
    text = 'Yes',
    command = lambda: msg_exit('Ok', 0)
)
yes.pack(padx = 10, pady = 10, side = 'left')
no = Tkinter.Button(
    root,
    text = 'No',
    command = lambda: msg_exit('NG', 1)
)
no.pack(padx = 10, side = 'right')

# 実行開始
root.mainloop()
```



[14] Python 3 ではモジュール名は小文字の tkinter になるので注意。