Python3 簡易資料

2023/7/18 訂補
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 4
4 データ型 5
5 基本構文 18
6 関数定義 28
7 コマンドライン引数 31
8 ファイル 32
9 オブジェクト指向プログラミング 36
10 モジュール 44
11 既存モジュールの利用 46
12 その他の機能 51

[注意] 本資料には一部、奈良女子大学 G 棟教育用計算機システム (以下「**G 棟システム**」と呼称) 固有の事情を念頭に述べている箇所がある。その場合は、その旨特記する。

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) が古くから知られ定評がある。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。いずれも、買うなら最新のものを買おう。また、近年は他にも良書が出つつある。

オンラインドキュメントは、`https://docs.python.jp/3/` に日本語版のものが各種ある[1]。ここは Python の**オフィシャルサイト**の一部なので、第 3 者の書いたネット上の情報よりは、基本的にはこちらを情報源とするのがよいだろう。同ページから辿れるものの中には「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。

2 Python プログラムの実行

Python はスクリプト言語で、AWK, Perl, Ruby などの言語と同様に、「簡潔な記述でテキスト処理を始め多彩な処理が書ける」「処理系がインタプリタであるためプログラムをすぐ実行でき、開発サイクルが速い」「多数の OS や環境で動作する」などの特徴や、それゆえの「実行時の効率よりも開発の速さ・容易さを優先する場合や、環境に依存せずに動くソフトの開発などに多用される」といった文化を持つ。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 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)。長らく 2 と 3 の両バージョンが並存した後、2020 年 4 月に Python 2 のサポートが終了したが、今後も Python 2 で書かれた過去のプログラムを読んだり保守したりする需要は当分残るものと思われる。この資料では原則的に **Python 3.X について**述べ、必要に応じて **2.X との違い**を述べる。

以降、UNIX 系 OS を前提として説明する。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」)

[1] Python 2 用のものは `https://docs.python.org/ja/2.7/`。ただし、Python 2 のサポート終了に伴い、そのうちなくなるかもしれない。

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。なお、現状で世の中には、「**python**」コマンドで Python 3.X が起動する環境と、「**python3**」コマンドで Python 3.X が起動する環境 (その場合、「python」コマンドでは Python 2.X が起動することが多い) のどちらもあるが、本資料では後者を前提として述べる。G 棟システムではどちらでも Python 3.X が起動する。

2.1 ワンライナー

「`python3 -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。

```
$ python3 -c 'print("Hello")'↵                    (「print("Hello")」の部分が Python プログラム)
Hello
```

python3 コマンド (あるいは python コマンド) には、`-c` 以外にも多数のオプションがあるが (man コマンド `man python3` あるいは `man python` 参照)、本資料では述べない。なお、`-c` オプションと他のオプションを両方指定する場合、Python プログラムの部分は `-c` の直後になければならない。

> **Python 2.X** では print の後ろを括弧で囲む必要がなく、`print "Hello"` のようにすればよい。これは、Python 2.X での print は「print 文」という文 (に使われる予約語) であったのが、Python 3.X では print が関数に変わったことによる。これにより、引数を括弧で囲むか否か以外にも、print の書き方にいろいろ差異が出る。Python 3.X だけを使うなら特に気にしなくてもよいが、Python 2.X で書かれたプログラムを 3.X へ移行するにあたっては注意すべき点の 1 つである。

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 World")
print(1+2*3, 4-5)
```

そして以下のように、「`python3 ファイル名`↵」で実行。print 関数は複数の引数をとると、それらを空白で区切りながら出力する。

```
$ python3 hello.py↵
Hello World
7 -1
```

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/python3
print("Hello World")
print(1+2*3, 4-5)
```

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)。これを次のようにして実行。

```
$ ./hello↵
Hello World
7 -1
```

先頭の「#!/usr/bin/python3」の行は、システムによってどう書けばよいかが異なる。

```
$ which python3↵
/usr/bin/python3
```

として出力された python3 コマンドのフルパスを、「#!」に続けて書く[2]。

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

2.3 節で作成した hello ファイルを、環境変数 PATH で指定されたディレクトリのうちのどこか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。

```
$ mkdir -p ~/bin↵          (ホームディレクトリ下に bin ディレクトリが既にあればこの操作は不要)
$ emacs ~/.bashrc↵
  ⋮ (末尾に「export PATH=~/bin:"$PATH"」と追加。ただし G 棟システムではこの操作は不要)
$ cat ~/.bashrc↵           (末尾に上記の行が追加されたことの確認)
  ⋮
export PATH=~/bin:"$PATH"  (ここで、ログインし直すことで設定を有効にする)
$ 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 World
7 -1
```

2.4 対話的実行

python3 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) または「exit()」を入力すると終了する。

```
$ python3↵
Python 3.6.8 (default, Apr 16 2020, 01:36:27)
  ⋮
>>> 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

Emacs (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もの) の場合、以下の 3 条件

1. ファイル名が「.py」で終わっている
2. 先頭行が「#!」で始まり「/python3」や「/bin/env python3」などで終わっている (最後は「python3」でなく「python」でも可)。例えば先頭行が「#!/usr/bin/python3」や「#!/usr/bin/env python3」であるなど
3. **末尾**付近に「# Local variables:」「# mode: python」「# End:」の 3 行がこの順に連続して書かれている

のうちの**少なくとも 1 つ**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)」あるいは「(Python ElDoc)」と出るのでわかる)。Emacs が 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。

[2] もう 1 つの方法として、先頭の行を #!/usr/bin/env python3 と書くことも可能。こうすると python3 コマンドのフルパスによらず、共通の記述で動作する。

また、上記を満たさなくても、キーボードから ESC x python-mode ↵ でも Python モードになる。特に、新規ファイルで名前が .py で終わらない場合、自動では Python モードにならないので、この操作で Python モードにしよう。全部打ち込まなくても、ESC x py TAB で「python-」まで補完、続いて mo TAB で「python-mode」まで補完できる[3]。

なお、環境によっては、上記 2. で先頭行の最後が「python3」でなく「python」でないと Python モードに自動的に入らない場合がある。そのときは、.emacs ファイルに

```
(add-to-list 'interpreter-mode-alist '("python3" . python-mode))
```

という 1 行を**書き足し**ておけば、先頭行の最後が「python3」でも Python モードに自動で入るようになる。

また、.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 ↵ 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)))
```

G 棟システムでは、これら 2 つの設定はいずれも**既に済んでいる**ので、ユーザが改めて行う必要はない。

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、UTF-8 以外の文字コードでスクリプトを書く場合 (スクリプト内に ASCII 文字しかない場合を除く) 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/python3
print("あ")
```

というプログラムを、日本語 EUC で書いて実行すると、以下のようにエラーが起きる。

```
SyntaxError: Non-UTF-8 code starting with '\xa4' in file ./a on line 2, but no
encoding declared; see http://python.org/dev/peps/pep-0263/ for details
```

しかし以下のように書けば、エラーせず正常に実行できる。「# coding: euc-jp」の行を「**エンコード宣言**」と呼ぶ[4]。

```
#!/usr/bin/python3
# coding: euc-jp
print("あ")
```

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、開発に **Emacs** を使うなら、「# coding: euc-jp」と書くより「# -*- coding: euc-jp -*-」と書く方が**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**、文字コードとして**日本語 EUC** を使うものとし、また、書き忘れを防ぐため、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**である (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。

[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、(文字コードが UTF-8 でない限り) 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/python3
print("Hello")  # ここに注釈があります
```

[3]補完のされ方はシステムによっても異なる。環境によっては ESC x py TAB で一気に「python-mode」まで補完される場合や、それ以外の場合もあるかも知れない。

[4]Python 2.X では、文字コードが UTF-8 であっても、スクリプト中に非 ASCII 文字があればエンコード宣言が必要であった。

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、本節冒頭の例と同様のエラーが起きてしまう。エラーを避けるには以下のようにしなければならない。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
print("Hello")  # ここに注釈があります
```

3.1 入出力の文字コード

プログラムが文字列を入出力する際の文字コードは、スクリプトの文字コードとは**関係なく**、ロケール (locale, 言語環境) で指定された文字コードが使われる[5]。例えば、日本語 EUC で書かれた

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
print("あ")
```

というスクリプトを、環境変数 LANG を指定して「`LANG=ja_JP.utf8 スクリプト名`↵」のように実行すると、日本語 EUC ではなく UTF-8 での「あ」が出力される (環境変数 LC_ALL にも影響を受け、こちらの方が優先される)。オープンされたファイル (8 節) に対して入出力する場合でも同様。

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。4.1 節)
- 真偽値 (True と False。4.2 節)
- 文字列 (4.3 節)
- リスト (配列に相当するもの。サイズも含めて変更可能。4.4.1 節)
- タプル (配列に相当するもの。変更不能。4.4.2 節)
- 辞書 (配列に似ているが、任意のデータを添字にできる。4.4.3 節)

などがある。リストやタプルや辞書は互いに入れ子にできる。

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[6]。ただし if などの予約語は変数名には使えない[7])。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 数と、数に対する演算

```
$ python3↵
Python 3.6.8 (default, Apr 16 2020, 01:36:27)
  ⋮
>>> 2**100↵
1267650600228229401496703205376        (** は累乗。無限多倍長整数が使える)
>>> 7/5↵
1.4                                    (実数の除算。被演算数が整数でも答は実数。ただし Python 2.X では、整数同士の除算は整数となるので、7/5 は 1 となる)
>>> 7//5↵
1                                      (整数除算。被演算数が実数でも、答は切り捨てで整数値にされるので、7.0//5.0 は 1.0 になる。Python 2.X でも使用可)
>>> a = 3; b = a + 1↵                  (変数の使用。「;」で複数文を 1 行に書くこともできる)
>>> a += 6 + b * (2 + 3)↵              (a に 6 + 4 × 5 を足すので、a は 29 に。演算子の優先順位は他の
```

[5]従って、バイナリデータを入出力する場合は、文字列とは別の「バイト列」(4.3.2 節) を使わねばならないことになる。
[6]Python 3.X では漢字なども変数名に使える。が、個人的にお勧めはしない。
[7]予約語の一覧は `https://docs.python.org/ja/3/reference/lexical_analysis.html#keywords` にある。他に、`list` や `max`. `len` など組み込み関数と同じ名前も、変数名には使わない方がよい。組み込み関数の一覧は `https://docs.python.org/ja/3/library/functions.html` に。

```
>>> a↵
29
>>> import math↵
>>> math.sqrt(2)↵
1.4142135623730951
```

多くの言語と同様、*, /, % が +, - より優先、** はそれより優先で右結合、など。ちなみに ++ や -- などは Python にはない)
(数学関数を使うのに必要。四則や累乗しか使わないなら不要)
(他にも sin, log, pi, gamma, erf など[8]多数の数学関数のメソッドあり)

import math は、**数学関数**を扱う「math **モジュール**」の読み込み (**インポート**)。モジュール (10·11 節) をインポート (10.1 節) してから、そのモジュールが提供する**メソッド** (例えば math モジュールなら sqrt メソッドなど) を呼び出す。

他に**複素数** (複素数に対する数学関数を使う場合のみ 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)
>>> import cmath↵
>>> cmath.sqrt(2j)↵
(1+1j)
>>> cmath.sqrt(-1)↵
1j
>>> import math↵
>>> math.sqrt(-1)↵
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
>>> from fractions import Fraction↵
>>> Fraction(1,2)+Fraction(1,3)↵
Fraction(5, 6)
```

(複素数演算。$(1-2i)(3+4i)=11-2i$ の計算)

($\sqrt{2i}=1+i$ の計算)

($\sqrt{-1}=i$ の計算)

(math モジュールは実数用のため、math モジュールの sqrt メソッドでは -1 の平方根は計算できない)

(モジュール読み込みのもう 1 つの方法、10.1.1 節参照)

($1/2+1/3=5/6$ 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)。

random モジュールを読み込めば**疑似乱数**の発生ができる。

```
>>> import random↵
>>> random.randrange(10)↵
6
>>> random.random()↵
0.5408358177987949
>>> random.choice(['q','w','e','r','t'])↵
'w'
>>> random.normalvariate(0, 1)↵
-0.2883307262352452
```

(10 未満の自然数の一様乱数を 1 つ生成。乱数なのでもちろん結果は場合によって異なる)
(0 以上 1 未満の実数の一様乱数を 1 つ生成)
(リスト (4.4.1 節) の要素をランダムに選ぶメソッドなども用意されている)
(平均 0、標準偏差 1 の正規分布をする乱数を 1 つ生成[9]。他にも種々の分布での乱数生成機能がある)

4.1.1 ビット演算

ビット演算は**整数**にだけ適用できる。ビット毎の NOT (~)、左右へのビットシフト (「<<」と「>>」)、AND (&)、XOR (^)、OR (|) といった演算子がある。優先順位は後のものほど低く (ビットシフト 2 つは同じ優先順位)、また ~ 以外は算術演算子 (+, *など) より優先順位が低い。なおビット演算では、負の整数は「無限桁の 2 進数の 2 の補数表現」と理解すればよい (例えば ~-4 は、-4 が無限桁の 2 の補数表現で $\cdots 1100$、そのビット毎 NOT なので、$\cdots 0011$ つまり 3)。

```
>>> 8 | 3 & 6↵
10
>>> ~-4↵
3↵
>>> 3 << 1 + 1↵
12
```

(8 と 3&6(= 2) のビット毎 OR で 10)

(上述の通りで 3)

(+ の方が優先順位が高いので、3 を 2 ビット左にシフト)

[8] gamma (ガンマ関数) と erf (誤差関数) は Python 2.7 と 3.2 以降で使用可。
[9] マルチスレッド (11.3 節) で使うのでなければ、gauss メソッドのほうが少し高速。

4.2 真偽値と、真偽値に対する演算

比較演算などが返す値が「真偽値」である。「真偽値」に属する値は True と False のみ。真偽値に対する and, or, not などを表す**論理演算子**は、C のような「`&&`」「`||`」「`!`」では**なく**、「`and`」「`or`」「`not`」であるので注意。また、これら論理演算子は、4.1.1 節に出た整数に対するビット演算子 (「`&`」「`|`」「`~`」) とは**異なる**ことにも注意が必要[10]。なお、数値に対する演算 (四則など) は真偽値にも適用でき、その場合 True は整数 1、False は整数 0 として扱われる。

```
>>> 1 == 2↵
False
>>> 5 > 4 > 3↵                          (Python ではこれは 5 > 4 and 4 > 3 と同じ[11])
True
>>> not (3 != 3 or True) and False↵     (優先順位は not, and, or の順。従ってこの式は真の not と偽の
False                                    and なので偽。もし括弧がなければ、真と偽の or で真となる)
>>> True + 4↵
5                                       (数値演算を適用すると True は整数 1 として扱われる)
```

実は、演算子 and, or, not は真偽値 (True, False) 以外の型のデータにも適用できる。真偽値に適用した場合も含め、結果は以下のようになる。

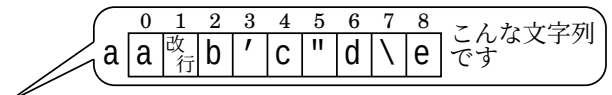
1. 式$_1$ `and` 式$_2$ の値は、式$_1$ の値が「偽として扱われる値」であれば式$_1$ の値になり (その場合式$_2$ の値は計算されない)、でなければ式$_2$ の値になる
2. 式$_1$ `or` 式$_2$ の値は、式$_1$ の値が「偽として扱われる値」**以外**ならば式$_1$ の値になり (その場合式$_2$ の値は計算されない)、でなければ式$_2$ の値になる
3. `not` 式$_1$ の結果は、式$_1$ の値が「偽として扱われる値」であれば True、でなければ False になる

ここで「**偽として扱われる値**」には False の他に、(本節以降に出てくるものも含め) 数値の 0、空文字列、空のリストやタプルや辞書や集合、特別な定数 **None** などが該当する (5 節に出てくる if 文や while 文などの条件部として使われた場合も、これらのデータは偽として扱われる)。

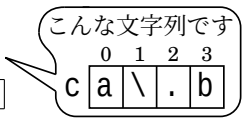
```
>>> 1+1 and 0.2+0.3↵                   (1 + 1 = 2 が「偽として扱われる値」でないので、上記 1. により
0.5                                      結果は 0.5)
>>> {} or 3↵                           (空の辞書 {} (4.4.3 節) は「偽として扱われる値」なので、上記 2.
3                                        により結果は 3)
>>> 0.0 and 1/0↵                       (0.0 は「偽として扱われる値」なので、上記 1. により結果は 0.0)
0.0
```

またこのことは、`and` や `or` 演算子については、C などの `&&` や `||` 演算子と同様、「**短絡評価**」が行われるということでもある[12]。上の最後の例では、`1/0` の部分は計算されないので、0 で割ることによるエラーは起きない。

4.3 文字列に対する演算




```
>>> a = 'a\nb\'c"d\\e'↵                 (文字列は「'」と「"」のどちらでも囲める。この例では
>>> b = "a\nb'c\"d\\e"↵                  a も b も「a 改行 b ' c " d \ e」という 9 文字からなる
>>> a == b↵                              文字列で、両者は等しい)
True
>>> a↵
'a\nb\'c"d\\e'                          (Python のデータ表記としての文字列 a の値)
>>> print(a)↵
a                                       (print 関数で文字列 a を出力した結果)[13]
b'c"d\e
>>> c = r'a\.b'↵                        (raw 文字列; r を付けると「\」がエスケープ記号ではなく
>>> c == 'a\\.b'↵                       「\」そのものとして扱われる)
```

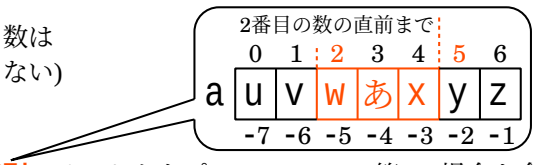


[10] 特に、真偽値に対する論理演算に、誤ってビット演算子を使わないよう注意。
[11] 厳密に言うと、式$_1$ > 式$_2$ > 式$_3$ のような場合に、式$_2$ が 2 度評価されないという違いがある。
[12] ビット演算子の「`&`」や「`|`」にはこの性質はない。

```
True                                    (表記が異なっても文字列としては等しい)
>>> d = '''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)
>>> d == 'abc\ndef\nghi'↵
True                                    (表記が異なっても文字列としては等しい)
```

raw 文字列もブロック文字列も「"」でも囲める (ブロック文字列の場合は「"""」) し、ブロック文字列の前に r を付ければ「raw 文字列のブロック文字列」という表記法も使える。

```
>>> a = 'ab' "c" 'de'↵                  (文字列定数をいくつか続けて書くと、それらをつなげた
>>> a↵                                  1 つの文字列と同じ扱いになる。空白を挟んでも可)
'abcde'
>>> a + 'xyz' * 3↵                      (文字列の連接や反復。文字列定数同士の場合以外は
'abcdexyzxyzxyz'                        「+」演算子をはさまないと連接はできない)
>>> 'abce' < 'abd'↵                     (文字列の比較。辞書順で後に来る方が大。文字同士の比較は
True                                    文字コードの大小で行われる)
>>> 'abc' < 2↵
Traceback (most recent call last):      (文字列と数は
  File "<stdin>", line 1, in <module>   比較できない)
TypeError: unorderable types: str() < int()
>>> a = 'uvwあxyz'↵
>>> a[2:5]↵                             (部分文字列。リストやタプル (4.4.1, 4.4.2 節) の場合も含めて
'wあx'                                  「スライス」とも呼ぶ。日本語文字も 1 文字として扱う[14])
>>> a[2:-1]↵                            (負数を指定すると後ろから数える。ただし -0 を指定すると
'wあxy'                                 0 と同じことになって、負数の指定にはならないので注意)
>>> a[2:]↵                              (「:」の前や後ろの省略も可能)
'wあxyz'
>>> a[2]↵                               (「[ ]」の中が 1 つのみだと 1 文字だけの文字列を取り出す)
'w'
>>> a[1:6:2]↵                           (「:」が 2 つの場合。a[1], a[3], ··· の順に a[6] の
'vあy'                                  直前までを (3 番目の数 − 1) 個おきに取り出す)
>>> a[6:1:-2]↵                          (3 番目の数が負の場合、a[6], a[4], ··· の順に a[1] の
'zxw'                                   直前までを −(3 番目の数 − 1) 個おきに取り出す)
>>> a[::-1]↵                            (特に、こうすると文字列の反転ができる。リストなども
'zyxあwvu'                              同様にして反転可)
>>> len(a)↵                             (長さを求める。組み込み関数の 1 つ。リストなどにも
7                                       使える)
>>> 'wあx' in a↵                        (部分文字列として含まれているか)
True                                    (a の値は 'uvwあxyz' なので含まれている)
>>> 'wx' not in a↵                      (in 演算子の否定はこう書ける。not 'wx' in a でも可[15])
True                                    (含まれていないので not で反転して True)
>>> a.index('wあx')↵                    (文字列メソッドの 1 つ; 部分文字列として何文字目から
2                                       (先頭を 0 と数えるので)                始まるか)
>>> a.index('wx')↵
Traceback (most recent call last):      (部分文字列として含まれていない場合エラー。なお、index
  File "<stdin>", line 1, in <module>   によく似た find というメソッドもあり、そちらは部分文字
ValueError: substring not found         列として含まれていない場合エラーにならずに −1 を返す)
>>> a.upper()↵                          (大文字化。他に小文字化 (lower) などのメソッドもある)
'UVWあXYZ'
>>> a↵
'uvwあxyz'                              (元の文字列は変更されない)
```



[13] ここで、「print(a)」と「a」の出力の違いに注意。Python の対話的実行では基本的に、入力された式の評価 (計算) 結果が出力されるのだが、print 関数は値を返さないので、「print(a)」と入力した場合はその式の評価結果は出力されない。「a」改行「b'c"d\e」と出ているのは式の評価結果ではなく print 関数の副作用 (値を返す以外の作用のこと) であり、その結果改行文字は改行として出力されるし、出力の前後にわざわざクォート文字は出ない。それに対し「a」と入力すると、a という式の評価結果、すなわち a の値が (Python のデータ表記の形で) 出力される。それは (文字列なので) クォート文字で囲まれるし、改行文字は「\n」と表記されるなど、print 関数による出力とは違いがある。

```
>>> '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)
>>> ','.join(['ab', 'c', 'de'])↵         (join の逆。指定した文字列をはさみながら、引数のリストや
'ab,c,de'                                タプルの要素の文字列を連接)
>>> a = 'xabyabzwy'↵
>>> a.replace('ab', '#')↵                (文字列の置き換え)
'x#y#zwy'
>>> a.replace('ab', '#', 1)↵             (先頭の n 個だけ置き換え)
'x#yabzwy'
>>> a.rstrip('wxy')↵                     (文字列の末尾から、引数に含まれる文字を削除。先頭から
'xabyabz'                                削除する strip メソッドもある)
>>> a↵
'xabyabzwy'                              (replace や rstrip メソッドでは元の文字列は変更されない)
>>> float('1.3')↵                        (実数への変換。整数への変換は int() 関数でできる)
1.3
>>> ord('A')↵                            (文字から文字コードへの変換。引数は 1 文字のみ)
65
>>> chr(65)↵                             (文字コードから文字への変換。ord の逆)
'A'
>>> '%d and others' % 10↵                (文字列フォーマッティングその 1)
'10 and others'
>>> '%d and %.3f and %s' % (10, 2.3, 'XYZ')↵   (フォーマットするもの (% の右) が複数ある場合は
'10 and 2.300 and XYZ'                          タプル (括弧で囲む。4.4.2 節) にせねばならない)
>>> '{0} and {2:.3f} and {1:s}'.format(10, 'XYZ', 2.3)↵   (文字列フォーマッティングその 2)[16]
'10 and 2.300 and XYZ'
```

「%」演算子は、数値に対しては除算の剰余、文字列に対しては**文字列フォーマッティング**として働くので注意。

組み込みの機能の中には、**メソッド** (データの後ろに「.」とメソッド名およびその引数を書いて呼び出す。例えば a が文字列の入った変数のときに a.index('cd') など) 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 やスライスなど。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)。

> 文字コードに関する注意点: Python 3.X にとって文字列とは、Unicode のコードポイントの列である (https://docs.python.org/ja/3/library/stdtypes.html#textseq)。従って、プログラムを記述する文字コードや、実行時の環境の文字コード (入出力に用いられる文字コード) の設定の如何によらず、文字列同士の比較や、ord や chr による文字列と文字コードとの変換は必ず Unicode のコードポイントで行われる (特に、ASCII 文字に対しては ASCII コードで行われることになる)。例えば '哀' < '唖' という式の値は、Unicode コードポイントでの比較により常に True となる。たとえプログラムが日本語 EUC で書かれていても (そして日本語 EUC の文字コード順では「唖」の方が先であるにも関わらず) 結果は変わらない。正規表現 (4.3.1 節) の表 1 中の、「[文字$_1$-文字$_2$]」という表記での文字同士の比較でも同様。

注釈 7 でも述べたが、組み込み関数の名前を誤って**変数として使わない**よう注意。例えば間違って len = 3 とすると、以後 len を組み込み関数として**使えなくなる**。モジュールの名前にも同様な注意が必要。

4.3.1 正規表現

正規表現とは、文字列のパターンを表す記法。「.」で任意の 1 文字、「*」で 0 回以上任意回の繰り返し、などを表す。文字列の中に、指定したパターンにマッチする部分があるかの検索などが行える。

正規表現処理には re モジュールを使う。使える正規表現は、他の言語 (特に Perl) とほぼ同じ。詳しくは https://docs.python.jp/3/library/re.html 中の「正規表現のシンタックス」を参照[17])。表 1 はその抜粋。

[14] Python 2.X では、Unicode 文字列というものを使わない限り文字列はバイト単位で扱われ、日本語文字 1 字は (日本語 EUC の場合) 2 文字として扱われていた。

[15] ただし、'wx' not in a の方が読みやすさの点で望ましいとされている。

[16] その 2 は Python 2.6 以後使用可。代わってその 1 が Python 3.1 以後廃止予定だったが、利用者が多いためか今では廃止の予定はなくなった模様。なお、Python 3.6 からは文字列フォーマッティングの手段として新たに「**f 文字列**」というものも使えるが、本資料では略。

[17] 現在のところ、POSIX の文字クラス ([[:upper:]] で英大文字を表すなど) は使えない。re でなく regex という別なモジュール (https://pypi.org/project/regex/) を使えば、POSIX の文字クラスも利用できる。なお、regex モジュールは別途インストールしないと使えない場合がある。

表 1: Python で利用可能な正規表現 (抜粋)

正規表現	それにマッチする文字列	例	それにマッチする文字列
特殊文字 (「*」「|」など) 以外の 1 文字	その文字自身	a	a
例えば英字や数字などがこの「特殊文字以外」に含まれる。			
\n, \t など	C 言語と同じ	\t	タブ文字
[文字並び]	[ ] 内の文字のいずれか	[abc]	a, b, c
[文字$_1$-文字$_2$]	文字コード順で文字$_1$ と文字$_2$ の間にある 1 文字	[a-e]	a, b, c, d, e
[^···]	「···」の部分に当てはまらない文字	[^a-ex]	a, b, c, d, e, x 以外の 1 文字
. (ピリオド 1 つ)	改行以外の任意の 1 文字◉	.	改行以外の任意の 1 文字
\特殊文字	その特殊文字自身	\.	.
\d	数字 1 文字	\d	0, 1, ···, 9
\s	空白類 1 文字	\s	空白 · タブ · 改行など 1 文字
\w	“単語” を構成する文字 1 字	\w	(ASCII 文字では) 英数字 · _
\D, \S, \W など	\d, \s, \w などにマッチしない文字	\D	数字以外の 1 文字
$\alpha\beta$ (α, β は正規表現とする。以下も同様)	α にマッチするものと β にマッチするものとを連接▲した文字列	ab	ab
$\alpha\|\beta$	α か β にマッチするもの	ab|cde	ab, cde
α*	α にマッチするものを 0 回以上つなげたもの	ab*c	ac, abc, abbc, abbbc, ···
α+	α にマッチするものを 1 回以上つなげたもの	ab+c	abc, abbc, abbbc, ···
α?	α にマッチするもの 1 回または 0 回	ab?c	ac, abc
(α)	α にマッチするのと同じ文字列。優先順位の変更に括弧を使う	a(n|pq)*	a, an, apq, ann, anpq, apqn, apqpq, annn, annpq, ···
\n (n は数)	左から n 番目の括弧で囲まれた正規表現にマッチした文字列と同じもの	(a|bc)\1	aa, bcbc
^	文字列の先頭	^a	文字列の先頭にある a
$	文字列の末尾	a$	文字列の末尾にある a

同表中の**連接** (▲) は、結合優先順位が選択 (「$\alpha|\beta$」の「|」) よりも高く、+ や * や ? よりも低い。また、「.」(表中の◉) は通常は改行にマッチしないが、「(?s:···)」の中では「.」が改行にもマッチする、というように、正規表現の意味を少し変える記法がいくつかある (英字の大文字小文字を区別せずにマッチさせる記法などもある)。本資料では略。

「\n」「\d」などの記法は、「[···]」の中でも同じ意味を持つ。

Python は国際化されているため、表中の「\d」「\w」などにマッチするのは ASCII 文字だけではない。例えば「\d」は Unicode の Fullwidth Digit (全角数字) にもマッチするし、「\w」は句読点などを除くほとんどの日本語文字 (仮名 · 漢字など) にもマッチする。これらを ASCII 文字にしかマッチさせなくする方法もあるが、これも略。

Python では、正規表現処理を行うには、典型的には以下の手順をとる。マッチ対象の文字列を S、正規表現パターンの文字列を R とすると

1. まず、re.compile() を使って正規表現パターン R を「**正規表現オブジェクト**」という特別なデータに変換
2. その正規表現オブジェクトの search メソッドに、引数としてマッチ対象の文字列 S を与える。これにより、S が正規表現パターン R と照合される
3. その結果、正規表現 R に**マッチする部分**が文字列 S の中にあれば「**マッチオブジェクト**」という特別なデータ、なければ特別な定数 None(4.2 節) が返る。前者の場合、マッチオブジェクトの group メソッドなどを使って「マッチしたのは文字列 S 中のどの部分か」などの情報を取り出せる (None には group などのメソッドは使えない)。

```
>>> import re↵                          (正規表現処理を行うのに re モジュールが必要)
>>> r = re.compile('ab*\\.c')↵          (正規表現パターンの文字列 ab*\.c を「正規表現オブジェクト」
```

```
                                          に変換し、変数 r に入れる)
>>> m = r.search('pqrabbb.cxyz')↵     (マッチ対象文字列 pqrabbb.cxyz を正規表現 ab*\.c と照合。
                                          この場合は正規表現にマッチする部分 abbb.c が文字列内に
>>> m↵                                  あるので、「マッチオブジェクト」が返され、それが m に入る)
<re.Match object; span=(3, 9), match='abbb.c'>   (左記を直接打ち込んでの入力はできない)
>>> m.group()↵                        (マッチオブジェクトの group メソッドは、マッチした部分の
'abbb.c'                                  文字列を取り出す)
>>> 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)
9
>>> m1 = r.search('pqrabbbkcxyz')↵    (今度はマッチする部分がないので m1 には None が入る)
>>> m1↵                               (変数に None が入っている場合は出力しても何も出ない)
>>> print(m1)↵                        (ただし print で出力した場合は別)
None
>>> m1.group()
Traceback (most recent call last):    (None に対しては group などのメソッドは呼べない)
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'group'
```

なお、正規表現パターンには「\」が含まれることが多い (上の例もそうである) ので、**raw 文字列** (4.3 節) で書けば、いちいち「\」を 2 重にする必要がなく、正規表現を読みやすく、書きやすくなる。上の例でも、`re.compile('ab*\\.c')` を raw 文字列を使って `re.compile(r'ab*\.c')` に書き直せば、意味は同じだが正規表現として読みやすい。以後も、当資料では正規表現パターンには (たとえ「\」が含まれていなくても) raw 文字列を使うことに統一する。

先の手順の 3. で、「マッチしたかどうか」を **if 文** (5.2 節に登場) などで判別したい場合は、定数 None が「**偽**」として扱われる (4.2 節) という性質 (そして「マッチオブジェクト」は常に「**真**」として扱われる) を使う。なお、下記のサンプルプログラム[18]は、同じ処理の繰り返しがあるため、6 節に出てくる**関数定義**を使っている。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
import re

# 第1引数に正規表現オブジェクト、第2引数に文字列を与え、正規表現が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、regex_test関数を定義
def regex_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:       # mが真ならmにはマッチオブジェクトが入っている(つまり、マッチしている)
        print('「%s」の中の「%s」にマッチ' % (string, m.group()))
    else:       # mが偽ならmにはNoneが入っている(つまり、マッチしていない)
        print('「%s」の中にはマッチする部分なし' % string) # Noneにはgroup()などは使えない

r = re.compile(r'ab*\.c')
regex_test(r, 'pqrabbb.cxyz')
regex_test(r, 'pqrabbbkcxyz')
```

このプログラムは、regex_text 関数の第 2 引数の文字列中に、正規表現 ab*\.c にマッチする部分があるか探し、

```
「pqrabbb.cxyz」の中の「abbb.c」にマッチ
「pqrabbbkcxyz」の中にはマッチする部分なし
```

のように出力する。

search メソッドは「対象文字列**の中**に、正規表現にマッチする部分があるか」を判定する。これに対し、match というメソッドを使うと「対象文字列の**先頭**から始まる部分に、正規表現にマッチする部分があるか」の判定を行う。ただし同じことは、正規表現パターンの先頭に「^」を置いて search メソッドを使っても実現できる。

[18] 5.2 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 (ただし注釈は、エンコード宣言を除き入力しなくてもよいが)。

[*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**ある場合、search メソッドでは**最初**にマッチする部分しか取り出せない。マッチする部分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 メソッドを用いる。例えば、このメソッドと for 文 (5.4 節) による繰り返しを組み合わせた以下のプログラム

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
import re

r = re.compile(r'ab*\.c')
for m in r.finditer('pqrabbb.cxyzabb.cjkl'):     # 毎回異なるマッチオブジェクトがmに入る
    print('%d文字目と%d文字目の間の「%s」にマッチ' % (m.start(), m.end(), m.group()))
```

は、finditer メソッドの引数の文字列中で、正規表現 `ab*\.c` にマッチする部分を全て探し、以下のように出力する[19]。

```
3文字目と9文字目の間の「abbb.c」にマッチ
12文字目と17文字目の間の「abb.c」にマッチ
```

[*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(r'ab*\.c')↵
>>> r.split('pqrabb.cxyzabbb.cjkl')↵        (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('!', 'pqrabb.cxyzabbb.cjkl')↵     (置換。文字列の replace メソッドとは引数が異なる)
'pqr!xyz!jkl'
>>> s = 'pqrabb.cxyzabbb.cjkl'↵
>>> r.sub('!', s, 1)↵                       (先頭の n 個だけ置換)
'pqr!xyzabbb.cjkl'
>>> r.sub(r'M\g<0>N', s)↵[20]               (置き換え文字列中の「\g<0>」は、正規表現にマッチ
'pqrMabb.cNxyzMabbb.cNjkl'                   した部分に置き換えられる。マッチしたもの全体では
>>> s↵                                      なく一部に置き換える方法もあるが、略)
'pqrabb.cxyzabbb.cjkl'                       (引数の文字列そのものは変わらない)
```

置換を行う sub メソッドは、第 1 引数 (マッチした部分を何に置き換えるか) に、文字列ではなく「マッチオブジェクトを受け取り、何に置き換えるかを文字列で返す関数」を指定することもできる (略)。

[*正規表現パターン文字列との直接照合*] 正規表現を、`re.compile` で正規表現オブジェクトに変換せずに、マッチ対象の文字列と直接照合してマッチオブジェクト (または None) を得ることもできる。本節冒頭の例で言えば

```
>>> r = re.compile(r'ab*\.c')↵
>>> m = r.search('pqrabbb.cxyz')↵
```

の 2 行の代わりに

```
>>> m = re.search(r'ab*\.c', 'pqrabbb.cxyz')↵     (文字列 'pqrabbb.cxyz' と正規表現 'ab*\.c'
                                                   を照合し、結果のマッチオブジェクトを m へ)
```

のようにすることもでき、以後は本節冒頭の例と同じ処理ができる。このとき、正規表現オブジェクトは内部でのみ生成されて使い捨てられる。search 以外のメソッド (finditer, split, sub など) についても、同様のことができる。

しかし、同一の正規表現を何度も使う場合は、`re.compile` を使って一度だけ正規表現オブジェクトに変換し、それを繰り返し使う方が効率が良くなることが期待される[21]。

[19] ただし、マッチする部分同士が重なるようなマッチは行われない。例えば finditer メソッドに文字列 `0ababbabbba0` と正規表現 `ab+a` を与えると、文字列中の `aba` と `abbba` の部分にマッチし、`abba` の部分にはマッチしない。

[20] ここでの第 1 引数 `r'M\g<0>N'` の先頭の `r` は、raw 文字列の `r` である。`=` の左が変数 r であることとは**関係ない**ので、念のため。文字列中にバックスラッシュがあるので、見やすさのために raw 文字列で表記しているのである。

[21] 実際には、最近に利用されたパターンの正規表現オブジェクトは内部で保存 (キャッシュ) されて再利用されるので、正規表現を少数しか使わないプログラムなら `re.compile` を使わなくても効率は落ちない。

4.3.2 バイト列

Python 3.X には文字列の他に「バイト列」というデータ型があり、バイナリデータを扱う用途 (バイナリファイルの読み書きなど) に利用できる。

```
>>> 'あ'.encode('euc-jp')↵                (文字列から encode メソッドでバイト列に変換可)
b'\xa4\xa2'                               (バイト列はこう表記する)
>>> b'\xa4\xa2'↵                          (バイト列の直接指定)
b'\xa4\xa2'
>>> b'\xa4\xa2'.decode('euc-jp')↵         (バイト列から文字列へ変換)
'あ'
>>> b'\xa4\xa2' + 'あ'↵
Traceback (most recent call last):        (バイト列と文字列は混ぜて演算できない)
  File "<stdin>", line 1, in <module>
TypeError: can't concat bytes to str
>>> print(b'\xa4\xa2')↵
b'\xa4\xa2'                               (バイト列を print するとこうなる)
```

バイト列を print すると、上のように Python のバイト列表記で出力される。バイトの列として入出力したい場合は 8.5 節を参照。

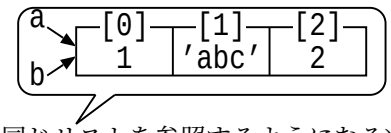
> Python 2.X には「バイト列」はなく (「バイト配列」はあるが)、代わりに文字列に (通常の)「文字列」と「Unicode 文字列」の区別があり、通常の文字列がバイト列の役割を兼ねる。Python 2.X では Unicode 文字列は `u'あ'` のように表す (Python 3.X では `u'あ'` のように書いても普通の文字列と同じ扱い)。なお、やや細かいが、Python 2.X で Unicode 文字列を print で出力する際、端末以外に出力するには `print u'あ'.encode('euc-jp')` のようにエンコーディングを指定しなければならないという注意点がある。
>
> 正規表現マッチング (4.3.1 節) はバイト列に対しても行える。その際は、正規表現パターンとマッチ対象の両方をバイト列で与えなければならない (いずれか一方だけをバイト列にすることはできない)。

4.4 リスト・タプルなどとそれに対する演算

4.4.1 リスト

他言語での「**配列**」にほぼ相当するもの。

```
>>> 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 と b は同じリストを参照するようになる)
>>> a[1]↵                                 (要素の参照。他の言語と同様、[ ] の中を
'abc'                                     「添字」と呼ぶ)
>>> 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ため)
>>> a[3] = 'def'↵
Traceback (most recent call last):        (存在しない要素の値を取り出したり書き換えたり
  File "<stdin>", line 1, in <module>      は不可)
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 + [[8, 9], 'pqr']↵              (リスト同士やタプル (4.4.2 節) 同士は + で連接可)
```



```
>>> a↵                                          (a への再代入が行われたので、これで a と b は
[1, 'def', 2, 'xyz', [4, 5], [8, 9], 'pqr']     違うリストになる。+= との違いに注意)
>>> b↵
[1, 'def', 2, 'xyz', [4, 5]]
>>> b * 2↵                                      (リストやタプルの反復も可)
[1, 'def', 2, 'xyz', [4, 5], 1, 'def', 2, 'xyz', [4, 5]]
>>> len(a)↵
7
>>> a[-2]↵
[8, 9]                                          (負の添字を指定すると後ろから数える)
>>> 'xyz' in a↵                                 (指定した要素があるか)
True
>>> a.index('xyz')↵                             (何番目にあるか)
3
>>> a[3:5]↵                                     (スライス。「:」の前後の省略も可能)
['xyz', [4, 5]]
>>> a[3:5] = [7, 6, 'ijk']↵                     (リストの場合はスライスへの代入も可)
>>> a↵
[1, 'def', 2, 7, 6, 'ijk', [8, 9], 'pqr']
>>> a[1:6:2]↵                                   (文字列の場合と同様、n 個おきに取り出す
['def', 7, 'ijk']                                スライスもある)
>>> a[6][0]↵                                    (リストの要素のさらに要素)
8
>>> a.pop()↵                                    (リストメソッドの例。最後の要素を削除し、
'pqr'↵                                          それを返す)
>>> a↵
[1, 'def', 2, 7, 6, 'ijk', [8, 9]]
>>> a.append(3)↵                                (要素を 1 つだけ追加するのはこれでもできる)
>>> a↵
[1, 'def', 2, 7, 6, 'ijk', [8, 9], 3]
>>> del a[5]↵                                   (要素の削除)
>>> a↵
[1, 'def', 2, 7, 6, [8, 9], 3]
>>> a.reverse()↵                                (要素を逆順にする。a の値が変わる)
>>> a↵
[3, [8, 9], 6, 7, 2, 'def', 1]
>>> a = [3, 1, 7.2, 5]
>>> sorted(a)↵                                  (ソートした結果を求める)
[1, 3, 5, 7.2]
>>> a↵
[3, 1, 7.2, 5]                                  (a の値は変わっていない)
>>> a.sort()↵                                   (ソートし、a の内容を直接書き換える)
>>> a↵
[1, 3, 5, 7.2]                                  (a の値が変わる)
>>> sorted([3, 1, 'a'])↵
Traceback (most recent call last):              (数と文字列のような比較不能な要素が混じった
  File "<stdin>", line 1, in <module>            リストはソートできない)
TypeError: unorderable types: str() < int()
>>> sorted(['foo', 'bar', 'bazzzz'])↵
['bar', 'bazzzz', 'foo']                        (文字列の大小比較は辞書順なので、辞書順でソート)
>>> ['a', 'c'] < ['a', 'd', 'b'] < ['b']↵       (リスト同士の比較は、要素同士の比較による辞書順)
True
>>> ['a', 'b'] < [1, 2]↵
Traceback (most recent call last):              (要素同士の比較が行えない場合はエラーになる)
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() < int()
```

```
>>> sorted([[3], [1,7], [6,5,4], [1,2]])↵
[[1, 2], [1, 7], [3], [6, 5, 4]]
>>> max([5, 7, 1])↵
7
>>> sum([5, 7, 1])↵
13
```

(要素がリストなので
リスト同士の比較で小さい順にソート)
(組み込み関数の 1 つ。他に min 関数もある)

(これも組み込み関数。数を要素とするリストに
使える)

Lisp 言語や Prolog 言語のリストは「連結リスト」というデータ構造で実装されているため、後ろの方の要素ほどアクセスに時間がかかるが、Python のリストはそうではなく、どの要素にも定数時間でアクセスできる。反面、要素の**加除**は、後ろの方に対してよりも、前の方に対して行うほうが時間を要する。

> 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が、特に機械学習などでは多用される。行列の乗算などが簡潔に書け、しかも Python の for ループを使うよりはるかに高速になる)。

リストに対する関数やメソッドのうち、sort, sorted, max, min など、要素の大小比較を伴うものは、文字列と数のような比較不能な要素が混じったリストに適用 (上の例の `sorted([3, 1, 'a'])` のように) するとエラーになる (Python 2.X ではそのような場合でも謎の順序により比較が行われ、エラーにはならない)。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[22]。

sort メソッドはタプル (4.4.2 節) にはない。一方、sorted 関数はタプルにも使える (が、結果はリストになる)。in 演算子、index メソッド、len · max · min · sum 関数もタプルにも使える。

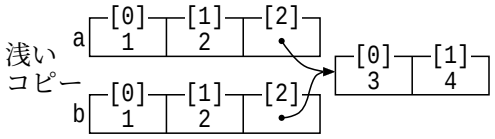
[*リストのコピーについて*] 本節冒頭の例の `b = a` のところから先の振る舞いでわかるように、リストを他の変数に代入しても、その変数は代入されたものと**同じリスト**を指すだけで、リストの**コピーはされない**。その結果、変数 a のリストの要素を書き換えると変数 b のリストの要素も書き換わる、ということが起きる。

リストをコピーしたければ、copy メソッドを使って `b = a.copy()` とするか、あるいはスライスを使って `b = a[:]` のようにすればよい。ただし、各要素のさらに中まではコピーされないので注意 (これを「**浅いコピー**」という)。

> `a = [1,2,[3,4]]; b = a[:]; a[1] = 6; a[2][0] = 5` とした後、`print(b)` としてみればそのことがわかる。b[1] は書き換わっていないが、b[2][0] は**書き換わっている**。a と b は異なるリストだが、a[2] と b[2] は同じリストだからである。

各要素のさらに中までコピー (「**深いコピー**」という) する方法もあるが、本資料では略。

辞書 (4.4.3 節) などについても同様の注意が必要である。



[*等しいかどうかの判定演算子*] Python には「**等しいかどうか**」を判定する演算子として「==」と「is」の 2 つがある。「==」は「値が等しいかどうか」、「is」は「同じデータかどうか」を判定する。例えば

```
>>> a = [1, 2, 3]; b = a↵
>>> print(a == b, a is b)↵
True True                    (a と b は同じリストである)
>>> b = a[:]↵                (スライスによってコピーを作るので、b は a とは違うリストになる)
>>> print(a == b, a is b)↵
True False                   (a と b は値は等しいが、同じリストではない)
```

のようになる。値の等しさを判定する == の方が、計算機中で 2 つのデータか同じものを指しているかどうかだけ判定する is より、わずかに時間はかかる。

特に、あるデータ (例えば変数 a の値) が **None** であるかどうかを判定したい場合、`a == None` よりも `a is None` の方がよい。None は Python にとって「シングルトン」(実行中の Python プログラム内に必ず 1 つしかない存在) であるため、この場合は a の値が「None と同一のデータであるかどうか」だけ判断すれば、すなわち `a is None` と書けばよく、その方が `a == None` より効率もよいためである[23]。

> もっとも、「a が None かどうか」ではなく「a が偽かどうか」を判定すれば十分な場合の方が多く、そのときは単に「`if not a:`」と書けばよい。4.3.1 節中の例の regex_text 関数内で if 文による判定を行っている箇所も同様で、「変数

[22] 本資料では略。`https://docs.python.org/ja/3/howto/sorting.html#key-functions` を参照。

[23] もう 1 つ理由がある。変数 a の値が、あるクラス (9 節) のインスタンスである場合、== 演算の結果はクラスの `__eq__()` メソッド (9.5 節) の定義により変更できるため、`a == None` の結果をわざと True にしてしまうこともできるのである。`a is None` ならそういうことがない。

m が真かどうか」を判定すれば十分なので、m を None と比較するのではなく「`if m:`」と書いている。

逆に、**リスト**や**文字列**などのように、値が等しくても同一のデータとは限らないものについて、**値が等しい**かどうかの判定をしたい場合は、`is` ではなく `==` を使わねばならない。

ちなみに、`is` の否定の演算子として `is not` というものがある。例えば「`a is b`」の否定は、「`not a is b`」とも「`a is not b`」とも書けるが、後者の方が少し読みやすい。
これに対し、`==` の否定の演算子は `!=` であり、`== not` ではない。例えば例えば「`a == b`」の否定は、「`not a == b`」とも「`a != b`」とも書けるが「`a == not b`」とは書けない。

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'])↵   (「[ ]」の代わりに「( )」で囲むとタプルになる)
>>> 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
>>> b↵                                          だめ。単に「3」という式を括弧でくくっているのと
(3,)                                           区別がつかないため)
>>> len(a)↵
4
>>> a[1]↵                                      (要素の取り出し。a(1) ではないことに注意。添字を
'abc'                                          囲むのはタプルでも [ ] を使う)
>>> 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: 'tuple' 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
>>> a↵                                          要素だから)
(1, 'def', (2, 'pqr'), [4, 'xyz'])
>>> list(a)↵                                   (タプルをリストに変換する組み込み関数)
[1, 'def', (2, 'pqr'), [4, 'xyz']]             (逆の変換は tuple 関数で可)
>>> ('a', 'c') < ('a', 'd', 'b') < ('b',)↵
True                                           (タプル同士の比較はリストと同様に行われる)
>>> ('a', 'c') < ['a', 'd', 'b']↵
Traceback (most recent call last):             (タプルとリストは比較できない)
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: tuple() < list()
```

Python のデータは「**不変性** (イミュータブル) オブジェクト」と「**可変性** (ミュータブル) 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 (4.4.3 節) などは可変性。

また、Python のデータは「**ハッシュ可能**オブジェクト」とそれ以外にも分類できる (`https://docs.python.jp/3/glossary.html#term-hashable`)。タプルや数や文字列などは「ハッシュ可能オブジェクト」でもある (リストや辞書はそうでない)。辞書のキーや、集合 (4.4.4 節) の要素になれるのはハッシュ可能オブジェクトのみ。

不変性なデータは大抵ハッシュ可能でもあるが、逆にハッシュ可能であっても不変性でないデータもある (例えばクラス (9 節) のインスタンスは、特別なことをしない限りそうなる)。

[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、左辺は [a, b] 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。

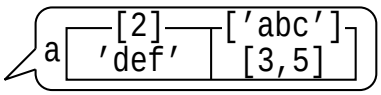
```
>>> a, b = 1, 2↵                    (2 つの変数に 2 つの値を代入するのはこう書ける)
>>> a + b↵
3
```

特に、要素が 1 個のタプルを作る場合、`b = (3,)` の代わりに `b = 3,` のように書ける。また、2 つの変数の値を入れ替えることが、`x, y = y, x` のようにしてできる。

4.4.3 辞書とそれに対する演算

辞書とは、“添字” として任意の**ハッシュ可能**オブジェクト (数、文字列、タプルなど。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[24]。リストやタプルと違って、要素に**順番**はない (ただし Python 3.7 から[25]は、要素を挿入した順に順番がつく)。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]}            (要素の順番はこの通りに表示されるとは限らない。以降
>>> b = {(3,5): 9, 1: (2,4)}↵        (最初から要素のある辞書を作る)          でも同様)
>>> b↵
{(3, 5): 9, 1: (2, 4)}
>>> a[2] = b↵                        (要素の書き換え)
>>> a↵
{2: {(3, 5): 9, 1: (2, 4)}, 'abc': [3, 5]}
>>> a[2]↵
{(3, 5): 9, 1: (2, 4)}               (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)
{(3, 5): 9, 1: (2, 4)}
>>> 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'
>>> a.update({'xyz': 1.5, 2: [6,7]})↵ (複数の要素の追加や書き換えを一度に)
>>> a↵
{2: [6, 7], 'abc': [3, 5], 'xyz': 1.5}
>>> del a['abc']↵                    (要素の削除)
>>> a↵
{2: [6, 7], 'xyz': 1.5}
>>> list(a.items())↵                 (リストへの変換。キーと値のタプルのリストになる)
 [(2, [6, 7]), ('xyz', 1.5)]
```



4.4.4 集合

集合型は、Python 2.4 以降で利用可能。集合型の要素になれるのはハッシュ可能オブジェクト (4.4.2 節) だけ。

[24] Java や Perl など他言語のものは「マップ」や「連想配列」とも、あるいは実装手段に由来して「ハッシュ」とも呼ばれる。
[25] 言語仕様としては 3.7 からだが、標準の処理系の動作としては 3.6 からそうなっている。なお、3.7 より前の Python では、順番を保存する辞書として OrderedDict というものが使える。これは 3.7 以降も残っており、一部の動作が普通の辞書と異なる。

```
>>> a = {7,2,11,7}↵
>>> a↵
{2, 7, 11}
>>> b = {11,7,2}↵
>>> a == b↵
True
>>> a.add(9); a.remove(11)↵
>>> a↵
{2, 7, 9}
>>> a | {-1,7}↵
{-1, 2, 7, 9}
>>> a↵
{2, 7, 9}
>>> a |= {-1,7}↵
>>> a↵
{-1, 2, 7, 9}
>>> a = set()↵
>>> a↵
set()
>>> {2,3} <= {1,2,3}↵
True
```

(Python 2.7 より前はこう書けず、`a = set([7,2,11,7])` と書く必要があった。それ以降でも set 関数でリストなどから集合を作ることは可能)
(集合なので要素の重複はない。また要素の**順番**もなく、この通りの順番で表示されるとは限らない。以降でも同様)
↳辞書と違って Python 3.7 以降でも要素に順番はない
(集合としてはこの 2 つはどちらも $\{2, 7, 11\}$ で等しい)
(要素の加除)

(a と集合 $\{-1, 7\}$ との和集合。他に `&` で共通集合、 `-` で差集合)

(a の値は変わっていない)

(`a.update({-1,7})` でも同じことができる)

(a の値が変わる)
(空の集合だけはこのように書かねばならず、`a = {}` とは書けない。空の辞書と区別がつかなくなるため)

(集合同士の大小比較は包含関係。`<=, >=, <, >` がそれぞれ $\subseteq, \supseteq, \subset, \supset$ に対応)

5 基本構文

if · for · while 文や関数 · クラス定義などの構文を、**インデント** (字下げ) で表現するのが Python の特徴。

5.1 文と継続行

後述の if · for 文などの、構造を持つ文を除き、基本的に文は行末で終わる。従って、例えば以下のプログラム

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
a = 1 + 2 + 3
print(a)
```

の中の「`a = ···`」の行を、以下のように複数行に分けて書くことは (C 言語などとは違って) **できない**。

```
a = 1 + 2
 + 3
```

ただし、行が「`\`」で終わった場合は、その次の行は「**継続行**」と呼ばれ、**前の行の続き**と見なされる。従って

```
a = 1 + 2 \
 + 3    # この行が「継続行」です
```

と書くことはできる。このとき、次の行が継続行であることを表す「`\`」の後ろには、何も書いてはならない (**空白**もあってはだめ。注釈 (3 節) も書けない)。

5.2 if 文

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
import sys      # sys.stderrやsys.exit()などを使うため
import math     # math.sqrt()を使うため。import sys, math のようにまとめてインポートも可能
```

```
l = input('何か非負実数を入れて下さい: ')        # 文字列の1行入力
a = float(l)                    # 文字列から実数へ変換

if a >= 0:
    b = math.sqrt(a)            # 平方根を計算
else:
    print('入力が負です', file = sys.stderr)     # 標準エラー出力へのprintはこう書く
    sys.exit(1)                 # sys.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 (上の例での「`a >= 0`」) や else などの後ろに「`:`」を付け忘れやすい。

このプログラムは、ユーザが入力した実数の**平方根**を出力するものである。入力が負だった場合のメッセージは、**標準エラー出力**に出している。

> ただし、input 関数は、入力が 1 文字もないうちに入力が終了 (EOF) するとエラーが起き、プログラムが終了する。これを防ぐには、このエラーで発生する EOFError という例外を捕捉する「例外処理」を書く必要がある (詳しくは 5.5.1 節。あるいは、input 関数の代わりに 8.1 節のように sys.stdin.readline を使う)。なお、Python 2.X にも input という名の関数があるが、これは働きが異なる (ここでの input 関数に相当する Python 2.X での関数は raw_input という名である)。
>
> Python 2.X で print の出力先を変えるには、`print >>sys.stderr, '入力が負です'` 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、これは対話的実行の終了 (2.4 節) に使う (また、sys. の付かない quit という関数も同じ役割)。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。

if 文の else 部 (「`else:`」を含んだそれ以降) は省略も可能。また、if 文の条件式のところには、「真」を表す定数 True や「偽」を表す定数 False を直接書く場合もある。さらに、4.2 節で述べたように、数値の 0 や空文字列や空リストや None など一部の値は、if 文の条件式としては「偽」扱いされる (5.3 節の while 文などでも同様)。

ちなみに (if 文と関係ないが)、sys.exit 関数には「引数に数でなく文字列を与えると、その文字列 (と改行) を**標準エラー出力**に出力し、**戻り値 1** でプログラムを終了する」という機能もある。エラーを理由に終了する場合、この機能は便利 (戻り値が 1 でよいなら)。例えば上のプログラムの `else:` の後 2 行はまとめて「`sys.exit('入力が負です')`」とも書ける。

> もう 1 つ、これも if 文とは関係ないが、「式$_1$ `if` 条件式 `else` 式$_2$」と書くと「条件式が真なら式$_1$、偽なら式$_2$」の意味の式になる (C 言語の「条件式 ? 式$_1$: 式$_2$」に相当)。if 文の代用にたまに使う。

[*インデントによる構文*] Python では、文の構造は**インデントで決まる**。例えば上のプログラムでは、`if a >= 0:` の後続のインデントされている部分が、条件が真のとき実行される範囲で、`else:` の後続のインデントされている部分が、条件が偽のとき実行される範囲、というように定まる。

従って、インデントは必ず**正しく**行わねばならない。例えば、先のプログラムの `else:` 以降の部分を

```
else:
    print('入力が負です', file = sys.stderr)
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、if 文の条件の真偽に関わらずここが実行されて、戻り値 1 でプログラムが終了してしまうため、後続の `print(b)` は決して**実行されない**。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:
    …
```

とは書けない (「`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 5.2.1 節に出てきた not を使って if not a >= 0: と書く)。

インデントにタブ文字と空白文字のいずれを使うかには注意がいる。行によって、インデントをタブで行うか空白で行うかが違っている場合、Python 2.X では、エディタ画面上で見えるインデント幅 (タブが空白 8 字分の幅と仮定した場合の) が同じでさえあれば、同じインデント量と見なされていた。しかし Python 3.X では、そのような場合は「TabError: inconsistent use of tabs and spaces in indentation」あるいは「IndentationError: unindent does not match any outer indentation level」というエラーになってしまう。
よって、インデントはタブ文字と空白文字のどちらか一方だけで行う方が (一般には空白文字だけで行う方が) よいだろう。Emacs が Python モード (2.5 節) になっている場合、Tab キーを押しても自動的に空白文字でインデントしてくれる。また、上記のエラーが発生してしまった場合は、インデントにタブと空白が混在していることを疑おう。

なお、例外として継続行 (5.1 節) は自由にインデントしてよい。例えば 5.1 節に出た

```
a = 1 + 2 \
 + 3    # この行が「継続行」です
```

という場合の「+ 3」の行は、どれだけインデントしても (しなくても) エラーにはならない。しかし、「a = ···」の行 (継続する前の行) は継続行ではないので、インデント量を正しくしなければならない。

5.2.1 elif、および条件式の複合

elif も使える。また、4.2 節にも述べたように、条件の「かつ」「または」「否定」には、C のような「&&」などではなく、and, or, not を使うので注意。

下記のプログラムは、ユーザが入力した整数が例えば 130 なら「0未満または100以上、かつ偶数」を、35 なら「24，56，98のいずれか、あるいは10進表記で3を含む」を出力し、11 ならどちらでもないので「その他」を出力する。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-

l = input('何か整数を入れて下さい: ')
x = int(l)              # 文字列から整数への変換

if not 0 <= x <= 99 and x % 2 == 0:
    print('0未満または100以上、かつ偶数')
elif x in (24, 56, 98) or '3' in str(x):        # str()で文字列に変換
    print('24, 56, 98のいずれか、あるいは10進表記で3を含む')
else:
    print('その他')
```

5.3 while 文

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-

x = 3
while x > 0:
    print(x, end = '')  # 改行を伴わない出力
    x -= 1      # 「--」や「++」はPythonにはない
print('Boom!')
```

このプログラムは「321Boom!」と出力する。

(while 文と関係ない話だが) print 関数は、標準では最後に自動的に改行を出力するが、print 関数の引数に「end = ''」を指定することでそれを抑制できる。このプログラムではそれを使って、改行しない出力を行っている。「end =」の

後ろには空文字列に限らず任意の文字列も指定でき、例えば上の例の 1 つ目の print 関数を「`print(x, end = ' ')`」に変更すれば、出力は「`3 2 1 Boom!`」になる。

> Python 2.X では、`print x,` のように print 文を「`,`」で終えれば最後の自動改行を抑制できる。この場合、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 (ただし、次の出力が行頭から始まる場合を除く)。

while 文や for 文 (5.4 節) のループ内では、C 言語と同様、break や continue が使える。加えて Python の while 文や for 文には、**else 部**も付けられる (繰り返しが終わった後に 1 回実行される。ただしループを break で抜けた場合は実行**されない**)。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-

l = input('何か整数を入れて下さい: ')
x = int(l)

while x > 0:
    print(x, end = ' ')
    if x == 2: break                # if文の本体が短い場合はこのような書き方も可能
    if x == 1: x += 5; continue     # (while, for, 関数定義などでも)。ただし好ましくはない
    x -= 3
else:
    print('Boom!', end = '')        # breakで抜けるとここは実行されない
print('')           # 空文字列をprintすることにより、ここで改行を出力
```

このプログラムは、ユーザが入力した整数が例えば 9 なら「9␣6␣3␣Boom!」を、7 なら「7␣4␣1␣6␣3␣Boom!」を、8 なら (else 部が実行されないため) 「8␣5␣2␣」を出力する。

> 行末に空白が入る箇所があるので、空白がどこに入るかを明示するため、ここでは空白を「␣」で示した。今後も、同様の場合はそうすることがある。

なお、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、sys.exit (5.2 節) を使えばよい。

5.4 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, Perl, Java などには両方の for 文がある)。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-

for x in (4, 1, 3): # タプルの各要素を1つずつ変数xに代入しながら繰り返す
    print(x, end = ' ')
print('')
```

このプログラムは「4␣1␣3␣」と出力する。

> for 文とは関係ないが、**括弧類** (「`( )`」「`{ }`」「`[ ]`」) **の中**では自由に改行でき、改行した場合、その次の行は自動的に「**継続行**」(5.1 節) と見なされる (次の行を継続行にする「`\`」を書く必要がない)。また、この場合も**継続行では**インデントは自由に行える。プログラム中にタプルやリストなどを書く場合に、このことを利用して例えば
>
> ```
> for x in ( # 以下の各要素について繰り返す
>     4,
>     1,
>     3,
> ):
> ```
>
> のように書くことができる (「`4,`」以降、括弧を閉じるまでの行が全て継続行と見なされている)。タプルやリストの要素が多かったり、各要素が長かったりする場合には、この方が見やすく、後での編集も楽なので、こう書くことはよくある (また、「`\`」によらない継続行の場合は、上の例のように継続行の前の行の後ろにも注釈を書ける)。
> さらに、リスト・タプル・辞書を表記する際に最後の「`)`」「`]`」「`}`」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、上記ではそのことも使っている。そうしておくと、後でプログラムを改造してタプルやリストの末尾に要素を追加することになった際に、「,」の書き落としでエラーになることを防ぎやすい。

辞書に対して for 文を使うと、辞書のキーを 1 つずつ取り出す。このとき、辞書の要素には順番はないため、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、必ず**先頭の要素から**順に取り出される)。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-

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が、出力の行の順番はこの通りとは限らない (ただし Python 3.7 からは[26]、辞書の要素には挿入した順に順番がつく (4.4.3 節) ので、出力の行の順番は必ずこの通りになる)。

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

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

5.4.1 イテレータ

「for 変数 in」の次には、リストやタプルのような配列状のデータだけではなく、「**イテレータ**」と呼ばれるデータも書ける。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、いわば“データ生成工場”のようなものである (とここでは説明しておく)。また、イテレータやリスト・タプルなど、for 文の繰り返しの対象にできるデータを「**イテラブル**」と総称する。

ここでは、よく用いられるイテレータのいくつかについて述べる。

[*enumerate 関数*] for 文で、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、例えば以下のようにする。

```
#!/usr/bin/python3
# -*- 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:                   # 添字が2なら
        break                       # ループ終了
```

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。実行結果はこうなる。

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

enumerate は、イテレータ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、enumerate 関数の**返す値**がイテレータ。以後に紹介するものについても全般的に同様)。また、enumerate 関数の引数には、リストやタプルなどだけではなく、イテラブルを渡すことができる。

上の例の for 文の箇所を

[26]注釈 25 で述べたように、実質的には Python 3.6 からそうなっている。

```
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)。このように、for 文の for の直後は変数 1 つであるとは限らない。

[*map 関数*・*filter 関数*] リストやタプルなどの各要素に対し、特定の関数を適用した結果を生成するイテレータを返すのが map 関数、特定の条件を満たすものだけを生成するイテレータを返すのが filter 関数である。この両関数は、第 1 引数に関数、第 2 引数にイテラブルをとる。従ってこれらは関数定義 (6 節で説明、ただし初出は 4.3.1 節) とともに使うことが多く、次の例のプログラムもそうである。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
def nijo(x): # nijo関数の定義
    return x*x      # xの2乗を返す
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

for i in map(nijo, [1, 3, 5]):          # リストの各要素の2乗に対し繰り返す
    print('A:%d' % i, end = ' ')
for i in filter(even, [24, 37, 52]):    # リストの各要素のうち偶数であるものに対し繰り返す
    print('B:%d' % i, end = ' ')
print('')
```

このプログラムの出力は以下のようになる。

```
A:1␣A:9␣A:25␣B:24␣B:52␣
```

なお、このように関数定義の本体が 1 つの式を返すだけの場合、関数定義の代わりに「ラムダ式」(6.2 節) というものを使って以下のようにも書ける。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
for i in map(lambda x: x*x, [1, 3, 5]):         # lambda x: x*x が
    print('A:%d' % i, end = ' ')                # 「xを与えてx*xを返す関数」を表す
for i in filter(lambda x: x % 2 == 0, [24, 37, 52]):    # lambda x: x % 2 == 0 が
    print('B:%d' % i, end = ' ')                # 「xを与えてx%2 == 0を返す関数」を表す
print('')
```

map 関数や filter 関数の処理結果を、for 文の繰り返しで使うのではなくリストやタプルとして得たい場合は、「イテラブルを **list 関数**の引数に渡すと、そのイテラブルの要素あるいは生成したデータをリストにして返す」という性質[27](tuple 関数、set 関数についても同様) を使う。すなわち、map 関数や filter 関数の返り値を list 関数の引数にすればよい。

```
>>> list(map(lambda x: x*x, [1, 3, 5]))↵
[1, 9, 25]
>>> list(filter(lambda x: x % 2 == 0, [24, 37, 52]))↵
[24, 52]
```

なお、Python 2.X では、map 関数や filter 関数自体がイテレータでなくリストを返す。

list 関数と map や filter の組み合わせと同様の効果が得られる「リストの**内包表記**」というものもある。例のみ示す。

```
>>> [x*x for x in [1, 3, 5]]↵
[1, 9, 25]
>>> [x for x in [24, 37, 52] if x % 2 == 0]↵
```

[27] ちなみに、4.4.3 に出た、`list(辞書.items())` で辞書をリストに変換する例も、実はこの性質を使っていた。辞書の items メソッドが「その辞書のキーと値のペアのタプルを順に生成するイテレータ」を返すので、それを list 関数でリストにしていたのである。

```
[24, 52]
```

[*range 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が range 関数 (Python 2.X では xrange 関数。なお、Python 2.X での range はイテレータでなくリストを返す、別の関数)。

`range(`n`)` (ここで n は非負整数) は、0 から $n-1$ までの整数を順番に生成するイテレータを返す[28]。従って、「`for` 変数 `in range(`n`)`」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 n 回ループすることになる。

次のプログラムは「0␣1␣2␣3␣4␣」と出力する。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
for x in range(5):  # xが0から4までの間繰り返す
    print(x, end = ' ')
print('')
```

なお、range には 2 引数や 3 引数で呼ぶ使い方もある。例えば `range(3, 11, 2)` は 3 から 9 まで (11 は含まない) の奇数を順に生成する。

[*zip 関数*] zip 関数は、いくつかのイテラブルを引数に取り、それらの生成する値 (あるいは要素) を組にしたタプルを 1 回ずつ返すイテレータを作る。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
for i in zip([3.2, 2.4, 5.1, 6.9], ('q', 'w', 'a'), range(8)):
    print(i)
```

このプログラムは、zip 関数が 3 つのイテラブルを引数に取り、それらが生成する値を 1 つずつ組にして

```
(3.2, 'q', 0)
(2.4, 'w', 1)
(5.1, 'a', 2)
```

と出力する (3 つのイテラブルのうち `('q', 'w', 'a')` が一番先に要素が尽きるので、zip 関数が生成したイテレータによる繰り返しもそこで止まる)。

2 引数の zip 関数が返すイテレータ[29]を dict 関数に渡すと、辞書 (4.4.3 節) を作る用途に使える。

```
>>> dict(zip(['z', 'x', 'c'], [7, 2, 5]))↵
{'z': 7, 'x': 2, 'c': 5}
```

[*その他*] 以上の他、例えば 4.3.1 節に出た finditer もイテレータを返すメソッドの 1 つ。また、8 節に登場する「ファイルオブジェクト」も、8.2 節のようにイテレータとしても扱える。12.1 節には「yield 文」によるイテレータも登場。

[*itertools モジュール*] itertools モジュール (`https://docs.python.org/ja/3/library/itertools.html`) は、イテレータを生成する便利な関数を多数含む。ここには 2 例だけ挙げる (list 関数と併用してリストを生成している)。

```
>>> import itertools↵
>>> list(itertools.product((0, 1), repeat = 3))↵      (直積の生成)
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)]
>>> list(itertools.permutations((4, 9, 2)))↵          (順列の生成)
[(4, 9, 2), (4, 2, 9), (9, 4, 2), (9, 2, 4), (2, 4, 9), (2, 9, 4)]
```

5.5 エラーと例外処理

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 (対話的実行 (2.4 節) の場合を除く)。例えば

[28] range 関数が返すものは、Python の「イテレータオブジェクト」とは厳密には異なるのだが、for 文の繰り返し対象として使われたときに、変数に代入される値を順に生成してくれるという点では、イテレータに近い存在と考えてよいだろう。Python の公式ドキュメントでも、range 関数は「整数のイテレータ」を返すとしている (`https://docs.python.jp/3/reference/compound_stmts.html#for`)。

[29] zip 関数が返すイテレータでなくても、2 つのもののペアからなるイテラブル一般でもよい。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
a = open('xyz')    # ファイルxyzをオープン
print('Ok')
```

というプログラムを実行する (ただし、xyz というファイルは存在しないとする) と、

```
Traceback (most recent call last):
  File "./xxx", line 3, in <module>
    a = open('xyz')    # ファイルxyzをオープン
FileNotFoundError: [Errno 2] No such file or directory: 'xyz'
```

open 関数 (ファイルのオープン、8 節参照) を呼んだ時点で、ファイルがないためエラーが起きるので、ここでプログラムは終了し、後続の「`print('Ok')`」の部分は**実行されていない**。

5.5.1 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、そのエラーにより発生する「**例外**」を捕捉する「**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/python3
# -*- coding: euc-jp -*-
import sys

try:
    a = open('xyz')    # ファイルxyzをオープン
except FileNotFoundError as err:
    print('エラー: %s' % err, file = sys.stderr)
print('Ok')
```

実行結果はこうなる。今度はエラーが起きてもプログラムは続行し、「`print('Ok')`」の部分は実行される。

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

基本的には「`except 例外名 as 変数名`」のように記述する (「`as 変数名`」は省略も可能。Python 2.5 以前では「`as`」でなく「`,`」)。「例外名」には捕捉したい例外の種別を書く。上の例では、open 関数で起きる「オープンしようとしたファイルがない」というエラーで発生する `FileNotFoundError` という例外 (5.5 節のプログラムの実行で出る**エラーメッセージ**の最後の行で分かる) を書き、これを捕捉している。

なお、try～except 構文の中に入れる部分は**最小限**にすべきであり、捕捉するエラーに関係ない処理を try～except 構文の中に入れるべきではない。上の例では、「`print('Ok')`」を except の中に入れるのは**よくない**。

例外が捕捉された場合、「`as`」の次の変数 (上の例では「`err`」だが、変数名は何でもよい) には、捕捉された例外に関する様々な情報を保持する特別なデータ (「**例外オブジェクト**」) が入る。例外オブジェクトにはいろいろな利用方法があるが、とりあえず、上の例のように例外オブジェクト自体があたかも文字列であるかのように `%s` で書式化 (あるいはそのまま print で出力) すれば、どのような例外であるかの情報を知らせるメッセージを保持した文字列として使える[30](上の例での「`[Errno 2] No such file or directory: 'xyz'`」がそれ)。

しかし、`FileNotFoundError` は「open 関数でオープンしようとしたファイルが**存在しない**」というエラーで発生する例外であり、open 関数で発生しうる例外は**これだけではない**。例えば、xyz というファイルではなくディレクトリが存在する場合、上のプログラムを実行すると「ディレクトリをオープンしようとする」という別なエラーが起きるため、下記のように `IsADirectoryError` という別な例外が発生し、上のプログラムではこれは捕捉できない。

```
Traceback (most recent call last):
```

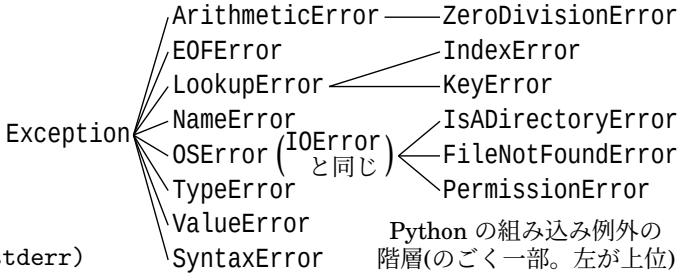
[30] これは、`%s` での書式化では Java でいう toString にあたる機構が働くことで起こる。なので、基本的に整数や実数などどんなデータでも `%s` で書式化が可能である。ただし、例えば実数を出力する場合、`%s` では小数第何位まで出力するかなどのコントロールはできない。

```
  File "./xxx", line 6, in <module>
    a = open('xyz')    # ファイルxyzをオープン
IsADirectoryError: [Errno 21] Is a directory: 'xyz'
```

open 関数で発生する例外を**全て捕捉**したい場合は、下記のように、捕捉する例外として OSError (Python 3.2 までは IOError)[31] を指定すればよい (このことは、https://docs.python.jp/3/library から「組み込み関数」→ open() とたどればわかる)。この例外は、FileNotFoundError や IsADirectoryError などの、open で起きうる例外を全て包含する、より “上位” の例外である (例外の**階層**については、https://docs.python.jp/3/library/exceptions.html#exception-hierarchy でわかる)。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
import sys

try:        # ファイルxyzをオープン
    a = open('xyz')
except OSError as err:
    print('エラー: %s' % err, file = sys.stderr)
print('Ok')
```



Python の組み込み例外の階層(のごく一部。左が上位)

代表的な例外としては、ここに登場したものの他、EOFError (5.2 節に登場)、ZeroDivisionError (0 での割り算で発生。上位に ArithmeticError あり)、IndexError (リストの添字が範囲外のとき発生。上位に LookupError あり)、TypeError (数と文字列を比較したなど、データ型に関する誤りで発生)、ValueError (math.sqrt の引数が負であるなど、組み込み関数の引数の誤りで発生) などがある。どの操作でどの例外が起き得るかは、Python のオンラインドキュメント (1 節) などであらかじめ調べておく。

なお、try ～ except 文には else 部を付けることもでき、その場合、例外が (except に指定しなかったものも含め) 全く起きなかったら else 部が実行される。次は ArithmeticError を捕捉した上で、else 部を付けた例である。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
import sys

try:
    a = 1 / 0   # ここでArithmeticErrorが発生
except ArithmeticError as err:
    print(err, file = sys.stderr)
else:
    print('Error not occured', file = sys.stderr)
print('Ok')
```

出力は

```
division by zero
Ok
```

となるが、上の例の「1 / 0」を「1 / 1」に変えると、例外が発生しないため、出力は以下のようになる。

```
Error not occured
Ok
```

複数の種類の例外を 1 つの except で捕捉したい場合は、例えば「except (ArithmeticError, ValueError) as err:」のように例外の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。

5.5.2 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、abc というファイルの 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

[31] Python 3.3 からは IOError は OSError の別名で、両者は同じものになっている。

イルオブジェクト」に関する知識を使っている (関数定義も使っている)。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
import sys

def test(fname):
    try:
        f = open(fname)     # ファイルをオープンする
    except OSError:         # as以後を省略した例
        sys.exit('Cannot open %s' % fname)
    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, end = '')」とするのでも可
        return
    finally:
        f.close()
        print(fname, 'is closed', file = sys.stderr)

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[32](ただし、ファイルのクローズ忘れを防ぐ目的であれば、8.3 節の with 文の方がより便利である)。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**付けることもできる (finally の方を後に書く)。

5.5.3 raise による例外発生

raise 文で、「わざと例外を発生させる」こともできる。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
import sys

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

この例の実行結果は以下のようになる。

```
fake arithmetic error
```

[32] もっとも、現時点の Python の標準的な処理系の実装では、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 (いつかは) 自動的にクローズするようになっているので、ファイルを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、このことは Python の言語仕様として保証されたことではなく、従ってこのことに依存したプログラムを書くべきではない。Python の公式ドキュメントでも、ファイルは必ずクローズするようにとしている。よって、ファイルは必ず明示的にクローズするか、あるいは try～finally や with などで自動クローズするようにしよう。

ただし例外として、ファイルの利用終了後**間もなく**プログラムも終了するような場合は、明示的にクローズしなくてもよいだろう。プログラムが終了する際は、OS が自動的にファイルをクローズするはず (一般的な OS なら) だからである。

```
Ok
```

ここでは既存の種類の例外 (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例外を定義 (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。

6.1 関数定義の例

[*ローカル変数*] 関数定義の中で変数に値を代入すると、その変数は自動的に**ローカル変数**になる (ただし、global 宣言によりグローバル変数にすることもできる)。関数定義の**仮引数**も、「関数定義の中で (その関数が呼び出されることにより) 値を代入されている変数」なので、ローカル変数となる。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-

x, y = 1, 2          # 2つの変数に値をそれぞれ代入

def yonjo(x):        # 引数の4乗を返す関数。xは仮引数なのでローカル変数
    y = x*x          # yに代入したので、yもローカル変数になる
    return y*y       # return文で値を返す

z = yonjo(3)
print(z, 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。

ただし、関数定義の中に現れる変数であっても、その変数への代入 (仮引数としての代入も含めて) が関数内に**一切書かれず**に、値を使うだけの変数であれば、その変数の値は関数定義の外のものが使われる。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
def f(x):
    return b * x
b = 2
print(f(3))
```

この場合、関数 f の定義の中には b への代入が書かれておらず、b は f の仮引数でもないので、b の値は関数 f の定義の外のものが使われる。関数 f が呼び出される時点では b には 2 が代入されているので、出力は「6」である。

[*返り値の型*] 関数は値を返すことも、返さないこともできる[33]。値を返すには return 文を使う (先の例に既に出てきた)。値を返す場合の返り値の型は自由であり、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ったり、場合によって値を返したり返さなかったりする関数も書ける。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-

def f(x): # 場合によって異なる型の結果を返す関数
    if isinstance(x, str):       # xが文字列なら
        return x + x[::-1]       # xとそれを反転したものをつなげた文字列を返す
    elif isinstance(x, (int, float, complex)):  # xが整数か実数か複素数なら
        return (x, -x)                          # xと-xからなるタプルを返す
    # どちらでもなかったら何も返さない
```

[33] 正確に言うと Python では、関数が値を返さないように見える場合、実は 4.2 節に出てきた特別な定数 None を返しているのだが、「値を返さない」関数のように考えておいてかまわない。組み込み関数の print なども、そのような関数の例である。

```
print(f(3), f('abc'))
```

このプログラムは「(3, -3) abccba」と出力する。**isinstance 関数**でデータの型の判定ができることを使っている。

> **数であるかどうか**の判定は、整数・実数・複素数 (4.1 節) などを包含する「数」を表す型である numbers.Number を使い、isinstance(データ, numbers.Number) とする方がもっと簡単に済む。ただし import numbers が必要。

[*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/python3
# -*- coding: euc-jp -*-

def test(x, y):     # この関数は(return文がないので)値を返さない
    x, y[0] = 3, 4  # xとy[0]にそれぞれ3と4を代入

a, b = 2, [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。

[*他の構文の中での関数定義*] Python では関数定義は (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。次の例では、f の引数が負のため、関数 g の定義は引数の 3 乗を返すものになり、出力は「8」となる。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-

def f(x):   # 引数の値によって異なる関数をgとして定義する関数
    global g    # gをグローバル変数(この場合はグローバル関数)にする
    if x >= 0:
        def g(x): # 引数の2乗を返す
            return x*x
    else:
        def g(x): # 引数の3乗を返す
            return x*x*x
f(-3)
print(g(2))
```

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)

6.2 データとしての関数

Python では関数もデータの 1 種 (**関数オブジェクト**) 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/python3
# -*- 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
fmap = {'nijo': f1, 'sanjo': f2, 'yonjo': f3}
```

```
fn = fmap['yonjo']
print(fn(3), fmap['sanjo'](2))      # 3の4乗と2の3乗で「81 8」を出力
```

実行結果は「81 8」となる。この例では fn 関数は f3 関数と同じものになっていることに注意。

この性質を使うと、関数を引数に取ったり、関数を返したりする関数も書ける。例えば、以下の例の pow_composite 関数は、関数 f (引数を 1 つ取るもの) を第 1 引数に、数 n を第 2 引数に渡すと、引数 x に対し $(f(x))^n$ を返す関数を作ってそれを返すものである。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-

def pow_composite(f, n):
    def powf(x): # f(x)のn乗を返す関数を定義して…
        return f(x)**n
    return powf  # 定義した関数を返す
def plus1(x):
    return x+1
plus1_cube = pow_composite(plus1, 3)  # plus1_cubeは引数xに対し、(x+1)の3乗を返す関数になる
print(plus1_cube(3))    # (3+1)の3乗で64を出力
```

この例では、pow_composite 関数の第 1 引数には、引数を 1 つ取る関数しか渡せないが、6.3 節に出てくる可変長引数関数の書き方を使えば、任意個の引数をとる関数を渡すようにも書ける。なお、powf 関数は pow_composite 関数内にローカルであり、他の場所から powf の名前で呼ぶことはできない。

他にも、5.4.1 節の map, filter 関数は関数を引数に取る関数の例である。また、関数を返す式も存在し、5.4.1 節に登場した「ラムダ式」がその例である。例えば

```
def nijo(x):
    return x * x
```

と書くのと、「`nijo = lambda x: x*x`」と書くのとは同じ意味である (Python では「関数定義」とは、関数と同名の変数に関数オブジェクトを代入することなので、ラムダ式が返した関数を変数 nijo に代入することで関数定義と同じことができる)。このように 1 つの式の計算結果を返すだけの関数の場合、関数定義をラムダ式で代用できる。

6.3 キーワード引数・可変長引数

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/python3
# -*- 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)
```

実行結果は次のようになる。呼び出し時に指定しない引数は、関数定義で与えられたデフォルトの値が使われていることに注意。

```
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 (本資料では略)。

さらに、Python では引数の個数が可変の関数 (**可変長引数**関数) も定義できる。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
```

```
def f(x, y, *z):
    print(x, y, z)

f(3, 4, 5, 6, 7)
```

実行結果は以下の通り。x と y に第 1·2 引数、z には残りの引数のタプルが入っている。

```
3 4 (5, 6, 7)
```

キーワード引数と可変長引数との混用や、キーワード引数を任意個取ることも可能 (本資料では略)。

上記で可変長引数関数の定義の仮引数に出てきた「*」は通称「アスタリスク演算子」などと呼ばれるもの。関数の引数の中では、イテラブル (5.4.1 節) の前に「アスタリスク演算子」を置くのと、そのイテラブルの要素を 1 つ 1 つ書くのとが同等になる。従って上の例では、要素が 5, 6, 7 となるイテラブルの 1 種であるタプルが f の引数 z に代入されている。関数の定義だけでなく呼び出すときも同じで、例えば 6 引数の関数 g が定義されているとき、`g(8, *range(4), 7)` という呼び出しは `g(8, 0, 1, 2, 3, 7)` と同等になる。

関数の引数の中で、イテラブルでなく辞書に対して同様の効果を持つ「**」という演算子もある。キーワード引数を任意個取る関数の定義には、これを用いる。

6.4 ドキュメント文字列

関数定義の本体の最初 (注釈除く) に文字列定数を書くと、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 (上の例はそうではないが)。

クラス定義 (9 節) にもドキュメント文字列を書ける。

7 コマンドライン引数

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 言語の main 関数の第 2 引数 (通常 argv という名前にする) 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。例えば

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
import sys
print(sys.argv)
```

というプログラム (プログラム名が fuhyo とする) を「`./fuhyo foo bar baz`↵」として実行すると、以下を出力する。

```
['./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)。例えば以下のプログラム

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
import sys
print(int(sys.argv[1]) + 2)      # sys.argv[1]をintに変換してからでないと、2を足せない
```

を「`./このプログラム␣3`↵」として実行すると、出力は「5」である。

7.1 オプション引数の解析

プログラムに**オプション引数** (UNIX の ls コマンドの「-l」引数のように、「-」で始まる、コマンドの働きを少し変えるための引数) を設けたいことがある。その場合、まずコマンドライン引数のうちオプション引数を分別し、どのようなオプション引数が指定されたのか特定する作業が必要になる。

その目的には、getopt モジュール[34]の getopt メソッドが使える。このメソッドは、機能も使い方も、C 言語の getopt() 関数や、シェルの getopts コマンドとほぼ同等であり、コマンドライン引数を与えて、『「-」＋一文字』の形のオプション引数、あるいは同じ形で**引数を取るオプション引数** (例えば tail コマンドの「-n」オプションは「-n 行数」のように、引数として行数をとる形で使われる) を解析することができる。

getopt メソッドの第 1 引数には、解析する引数のリスト (典型的には sys.argv[1:]) を与える (これを以下「引数リスト」という)。また、第 2 引数にはオプション文字 (オプション引数の「-」の次の文字) を連ねた文字列を与え、このとき、引数を取るオプションの場合は、オプション文字の後ろに「:」をつける。例えば、プログラムに -a, -b の 2 種類のオプション引数と、引数をとる -c, -d の 2 種類のオプション引数を設けたい場合、第 2 引数は文字列 abc:d: とする。

そうして getopt メソッドを呼ぶと、引数リストの先頭からオプション引数の解析が行われ、2 つの値 (からなるタプル) が返る。1 つ目の値は、引数リスト内に指定されていたオプション引数と、そのオプションが取る引数 (取らなければ空文字列) のペア (タプル) からなるリスト。2 つ目の値は、引数リストのうちオプション引数として解析されなかった残りの引数のリスト (引数リストの末尾) である。オプション解析でエラーが起きた場合 (第 2 引数に指定していないオプションを発見したなど) は、getopt.GetoptError という例外を発生する。なお、引数リスト内では、「-」の次に複数のオプション文字をまとめて指定することもできる。また、引数リスト内に (引数を取るオプションの引数以外で)「--」があれば、(その「--」は除去された上で) そこから先はオプション引数と見なされない。例を示す。

```
>>> import getopt↵
>>> opts, args = getopt.getopt(['-c', 'pqr', '-a', '-dijk', 'xyz', 'fgh'], 'abc:d:')↵
>>> print(opts, args)↵
[('-c', 'pqr'), ('-a', ''), ('-d', 'ijk')] ['xyz', 'fgh']
>>> opts, args = getopt.getopt('-badpqr -- -c xyz fgh'.split(), 'abc:d:')↵
>>> print(opts, args)↵                              (「--」以後はオプション解析されていない)
[('-b', ''), ('-a', ''), ('-d', 'pqr')] ['-c', 'xyz', 'fgh']
>>> opts, args = getopt.getopt(['-e', 'xyz'], 'abc:d:')↵
    ··· エラー発生 ···  (中略)
getopt.GetoptError: option -e not recognized
```

getopt メソッドの実際のプログラムでの**使用例**は、8.1.2 節のプログラムで示す。

getopt メソッドには、「長いオプション引数」(「--long」など、「--」の次に 1 文字以上が続く形のオプション) を解析する機能もあるが、本資料では省く。また、同メソッドの代わりに gnu_getopt というメソッドを使うと、引数リストの後ろからもオプション引数を解析する (Linux の cat, ls など多くのコマンドのオプション解析方式と同じ) ようになる。

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 (つまり読み出しオープン) になる。なお、Python 3.X では読み書きは基本的に (バイト列ではなく) 文字列として行われる (バイト列を読み書きする場合、すなわちバイナリファイルの読み書きについては 8.5 節参照)。

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、一定文字数読むには read[35]、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。また、ファイルオブジェクトの flush メソッドは入出力バッファのフラッシュ(C の fflush() 関数に相当する動作) を行う。

[34]使い方がやや異なる、argparse モジュールというものもある。

[35]read メソッドは、引数を指定すればその指定した文字数分だけファイルから読み込むが、引数なしで呼べば、ファイル全体を読み込んで 1 つの文字列として返す用途にも使える。

8.1 ファイルオープンの例

ファイルから読むだけの例は 5.5.2 節に既に出てきたが、次の例は、ファイルへの書き出しも行う例である。ファイル abc から 1 行ずつ読みながら、各行頭に「×行目:␣」を付加してファイル def に書く。ただし、ファイルのオープン時のエラーチェックや、最後のファイルのクローズは省いた[36]。実行結果は省略。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-

f = open('abc')          # モードを省略したので読み出しオープン
g = open('def', 'w')     # 上書きオープン
i = 0
while True:
    s = f.readline()     # 行末の改行は削除されない
    if s == '':
        break
    i += 1
    g.write('%d行目: %s' % (i, s))  # write()は出力末尾で勝手に改行しない
```

なお、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。それらは、ここに出てきたものとは異なる (本資料では略)。

標準入出力や標準エラー出力 (sys.stdin, sys.stdout, sys.stderr) も、ファイルオブジェクトの一種である。従って、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**にも使える。

特に、input (5.2 節) 関数と同じようなことを sys.stdin.readline で行ったり、print の代用に sys.stdout.write を使ったりできる。ただし、input や print と、readline や write には違いがあることに注意。具体的には以下のような差異がある。

- input は行末の改行を勝手に削除するが、readline は削除しない
- 入力を読み終わった後で呼ぶと、input はエラーを起こす (5.2 節) のに対し、readline は空文字列を返す
- print と異なり、write で出力する場合は引数は**文字列 1 つ**しか取れない (それ以外を write で出力したい場合は、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)
- print は (デフォルトでは) 出力の最後に勝手に**改行**を付加するが、write は改行を付加しない

逆に、write の代用に「`print(···, file = ファイルオブジェクト)`」の形を使うこともできる。特に、出力先が sys.stderr である場合のこの書き方の例は既に 5.2 節に出てきているが、他の例として、例えば上のプログラムの `g.write('%d行目: %s' % (i, s))` のところを `print('%d行目: %s' % (i, s), file = g, end = '')` とも書ける。この場合、print なので `end = ''` がないと出力の最後に自動的に改行を付加されてしまう。

8.1.1 ファイル全体を一気に読むことについて

次のプログラムは、8.1 節のプログラムと同じ動作をする。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-

f = open('abc')
g = open('def', 'w')
l = f.readlines()   # ファイルabc全体を行単位で読み込んでリストにする
i = 0
for s in l:         # リストの各要素、つまりファイルabcの各行についてループ
    i += 1
    g.write('%d行目: %s' % (i, s))
```

しかし違いとして、このプログラムは readlines() メソッドでファイル abc の内容**全体**を一気にメモリに読み込んでしまう。ファイルが**巨大な場合**、これはシステムに負荷をかけることになる。今回の処理の場合、こうしなくても 1 行ずつの読み書きで**同じことができる**ので、このプログラムはシステムに**無駄な**負荷をかけていることになり、望ましくない。

ファイル全体を読み込むメソッドを使う場合、それが**本当に必要か**考えて行うようにしよう。

[36]注釈 32 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、このようにオープンしたファイルの読み書きを終えてすぐ終了するような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。8.1.2 節のプログラムでも同様。

8.1.2 もう少し複雑な (ファイル読み書きの) 例

次の例は、プログラムが少し長いので本文には載せず、この PDF ファイルの「**添付ファイル**」に収めてある。

そのプログラムを閲覧するには、まず、この PDF の添付ファイル samples.tar.gz を何らかの方法で取り出す。添付ファイルを取り出す機能が PDF ビューアに備わって[37]いればそれを使えばよいし、UNIX 系 OS なら **PDFtk** というコマンドラインツールを使って「`pdftk このPDFファイル unpack_files`↵」としても取り出せる。

続いてそれを、「`tar zxvf samples.tar.gz`↵」とするかまたは何らかのアーカイブ展開ツールで展開すると、samples というディレクトリができる。そのディレクトリ内の comma_rw (2.3 節の実行方法を前提とするため、ファイル名に拡張子 .py がない) が、ここで取り上げるサンプルプログラムである。なお、今回はファイルのオープン時のエラーチェックや、ファイルのクローズも行っている。また、7.1 節で述べた**オプション引数解析**のサンプルも兼ねている。

このプログラムは引数を 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 1·2 欄目をやはり「,」で区切りながら第 2 引数のファイルに書き出す。ただし以下の 2 種類のオプション引数がある。

- `-a` を指定すると第 2 引数へのファイルの書き出しが追加書きになる (なければ上書き)
- 引数つきのオプション`-c` により、第何欄を第 2 引数のファイルに書き出すかを変更できる (`-c` は複数指定もでき、例えば `-c 2,5,3 -c 1,7` とすると第 2, 5, 3, 1, 7 欄をこの順に書き出す)

comma_rw をカレントディレクトリに置いたとして、例えばファイル abc に以下の 3 行の内容

```
ghi,jkl,mno,pqr,stu,vwx
あ,い,う
12,3,45,678,90
```

が入っているとすると、「`$ ./comma_rw abc def`↵」によって、ファイル def には以下の内容 (abc の各行の第 1·2 欄) が書き込まれ、

```
ghi,jkl
あ,い
12,3
```

それに**引き続いて**「`./comma_rw -a -c5,3 abc def`↵」とすると、ファイル def には**追加書きで**以下の内容 (abc の各行の第 5·3 欄) が書き込まれる (試してみよ)。プログラムの詳細については、当該プログラムの注釈を参照のこと。

```
stu,mno
,う
90,45
```

8.2 イテレータとしてのファイルオブジェクト

5.4.1 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」としても扱える——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:`」の形で使える。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 8.1 節に出てきたプログラムを次のように書き換えても、元と同じ動作をする[38]。ただしこの方法が使えるのはあくまで、**行単位**で読んで繰り返し処理を行う場合のみ。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-

f = open('abc')
g = open('def', 'w')
i = 0
for s in f:      # ファイルオブジェクトfをイテレータとして使用
    i += 1
    g.write('%d行目: %s' % (i, s))
```

[37] PDF Studio Viewer や Foxit PDF Reader など、著名な複数の PDF ビューアがその機能を備える。ただし Acrobat Reader の同機能は「添付ファイルを開くことができません」と言われて役に立たない。また、UNIX で PDF ビューアとして (も) よく使われる Evince には、同機能はない模様。

[38] しかもイテレータなので、8.1.1 節のプログラムと違ってファイル全体を一気に読み込みはせず、メモリ消費は 8.1 節のプログラムと変わらない。

8.3 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、5.5.2 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。Python 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、「with 文から抜ける際に、そのファイルを必ずクローズする」という処理が行われる。例えば、5.5.2 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(fname, 'is closed', file = sys.stderr)
```

のように書ける (最後の print 文は、f をクローズした後の処理なので 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。

上の例はオープン済みのファイルに対し with 文を使ったが、with 文でファイルのオープンもまとめて行うこともできる。例えば、5.5.2 節のプログラム全体を以下のように書き直せる (try～except で例外が捕捉される範囲が with 文全体に変わる[39]ことを除き、実行結果は 5.5.2 節のプログラムと同じ)。この使い方は、ファイルのオープンから自動クローズまでを全部 with 文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/python3
# -*- 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 OSError:
        sys.exit('Cannot open %s' % fname)
    print(fname, 'is closed', file = sys.stderr)

test('abc')     # ファイルabcは存在するとする
```

8.4 文字コードを指定しての読み書き

3.1 節の末尾にも述べたように、オープンしたファイルに対して入出力 (文字列を読み書き) する場合、文字コードはロケールで (環境変数 LANG や LC_ALL で) 決まる。プログラムの記述に使われている (プログラム内のエンコード宣言に書かれている) 文字コードとは無関係である。

特定の文字コードで読み書きしたい場合は、ファイルをオープンする際に文字コードを指定すればよい。例えば

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-

f = open('abc', 'w', encoding = 'utf-8')
print('あ', file = f)
```

というプログラムは、環境変数 LANG や LC_ALL の値の如何に関わらず、ファイル abc に UTF-8 で「あ」を (改行つ

[39]ただしこれでは、エラー捕捉の対象となる「ファイルのオープン」という操作とは関係ない処理が try～except の中に入ってしまい、5.5.1 節で述べた「捕捉するエラーに関係ない処理を try～except の中に入れるべきではない」という方針に反してしまう。従って、このように with 文を try～except の中で使うのは、オープンしたファイルに対する操作が簡潔なもので済む場合に限るのがよく、それ以外の場合はファイルを (エラー捕捉つきで) オープンした後で with 文を使うか、あるいは 5.5.2 節のように try～finally を使うやり方の方がよいだろう。

きで) 書き出す。

8.5 バイナリファイルの読み書き

バイナリファイルの読み書きは、通常のファイルオブジェクトではできない。「バイナリファイル用のファイルオブジェクト」を用意し、それに対して (Python 3.X での) バイト列 (4.3.2 節) を読み書きする必要がある。

「バイナリファイル用のファイルオブジェクト」を得るには、通常のファイルオブジェクトの後ろに「.buffer」を付ける (例えば「sys.stdout.buffer」のように) か、あるいは open 関数 (8 節) の第 2 引数 (モード) の文字列に「b」を付加 (例えば「open(ファイル名, 'rb')」のように) する。読み書き自体は、通常と同様、read や write などのメソッドでできる (input や print はバイナリファイル用のファイルオブジェクトに対しては使えない)。なお、普通の文字列をバイナリファイル用のファイルオブジェクトに読み書きしたり、バイト列を通常のファイルオブジェクトに読み書きしたりはできない。また、バイト列の読み書きには文字コードは当然関係ない。

下記のプログラムは、ファイル abc (がバイナリファイルであっても) をファイル def にコピーする。ファイルがオープンできなかった場合のエラー処理などは省いている。説明のため、f は普通のファイルオブジェクト (f.buffer をバイナリファイル用ファイルオブジェクトとして使用)、g はバイナリファイル用ファイルオブジェクトとしたが、f をバイナリファイル用ファイルオブジェクトにしたり (その場合 f.buffer ではなく f から read する)、g を普通のファイルオブジェクトにしたり (その場合、g ではなく g.buffer に対して write する) してももちろんかまわない。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
import sys

f = open('abc', 'r')    # 読み出しに使うファイルオブジェクト(普通の)
g = open('def', 'wb')   # 書き出しに使うバイナリファイル用ファイルオブジェクト

while True:
    s = f.buffer.read(1024) # f.bufferからバイト列を最大1024バイト分読む
    if not s:               # 空のバイト列は「偽」扱いになる
        break
    g.write(s)              # バイト列をgに書き出す
f.close()
g.close()
```

9 オブジェクト指向プログラミング

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。親クラスが**ない**場合、単に「class クラス名:」と書く (ただし Python 2.X では「class クラス名(object):」とする。最上位のクラスである「object」を親として宣言していることになる)。なお、Java と異なり、クラス名はプログラムを格納するファイル名と一致する必要はなく、また 1 つのファイルに多数のクラスを定義してもかまわない。

クラスの**インスタンス**を生成するには「new クラス名(引数···)」ではなく、単に「クラス名(引数···)」と書く (クラスと同名の関数を呼べば、値としてインスタンスが返るわけである[40])。

クラスやメソッドの名前の規則は、変数名のそれと同じである。

> ただし、大規模なプログラムや、他人と共用する (共同開発、あるいは公開して他人に使わせてあげるなど) プログラムなどの場合は、https://pep8-ja.readthedocs.io/ja/latest/#section-20 で提案されている「クラス名は『英大文字 1 字+英小文字』からなる単語をいくつかつなげたもの (例えば「LogicalFormula」のように)、メソッド名は英小文字からなる単語かそれらを「_」でつなげたもの (例えば「calculate_truth_value」のように)」という規約に従うのがよいだろう[41]。それ以外の変数名や関数名などについても同様のことが言える。

Python では Java と異なり、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 (継承がどの親クラスから行われているのか、わかりにくくなるためである)。

[40] int, float, str, list などの関数も、実はクラスのインスタンスを返す、クラスと同名の関数なのである。
[41] この資料はその点をあまりまじめに考えずに書いたので、それらの規約には従っていない箇所があることをお断りしておく。

9.1 クラス定義の例

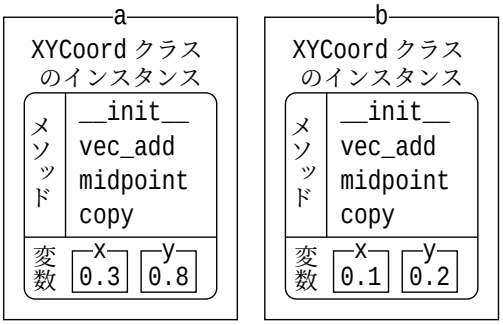
クラスの構文もインデントで決まるので、インデントが終わるまでがクラス定義の範囲である。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-

class XYCoord:  # xy座標を表すクラス
    def __init__(self, x, y):   # コンストラクタ
        self.x, self.y = x, y   # クラス内の変数(インスタンス変数)x, yに引数のx, yを代入
    # メソッドの定義
    def vec_add(self, v):   # 他のXYCoordのインスタンスを引数にとり、
        self.x += v.x       # そのx, y座標を自分のx, y座標に加算
        self.y += v.y
    def midpoint(self, v):  # 他のXYCoordのインスタンスを引数にとり、
        self.vec_add(v)     # その点と自分との中点を新たな自分の座標にする
        mag = .5                # ローカル変数の使用
        self.x *= mag
        self.y *= mag
    def copy(self):         # 自分と同じ座標を持つ新たなXYCoordのインスタンスを作って返す
        return XYCoord(self.x, self.y)

a = XYCoord(0.3, 0.8)   # インスタンスの生成
print(a.x, a.y)
b = XYCoord(0.1, 0.2)
c = a.copy()            # メソッドの呼び出し
a.midpoint(b)           # メソッドの呼び出し
print(a.x, a.y)
print(c.x, c.y)
```

bへの代入が済んだ時点の変数の状況



実行結果を示す[42]。

```
0.3 0.8
0.2 0.5
0.3 0.8
```

クラス内で定義されるメソッド (コンストラクタを含む) 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 引数の変数名は、慣習的に「`self`」とする。ただし、メソッドを**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、メソッド定義側の引数の個数は、呼び出す側での引数の個数より **1 つ多くなる**)。通常、メソッドの引数の個数がいくつか、という場合は、**呼び出す側**の視点で数える (上の例ではコンストラクタは 2 引数、midpoint() は 1 引数、copy() は 0 引数、など)。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。なお、そうやってアクセスできる変数やメソッドは、基本的には**インスタンス変数**および**インスタンスメソッド**である (例外的にクラス変数になる場合あり、9.2.1 節参照)。クラス変数やクラスメソッドを作りたい場合は 9.2.1 節参照。

なお、メソッド定義は関数定義 (6 節) の一種なので、メソッド定義の中で (`self.` を付けずに) 代入された変数は、(global 宣言がない限り) **ローカル変数**になる。上の例の midpoint メソッドの変数 mag がその例 (もちろん、こんな変数を使わずに単に `self.x /= 2` のようにしてもよいのだが)。

Python の場合、Java とは違って、1 つのクラスに引数の個数や型が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 (6.3 節) にしたり、メソッド定義の中で引数の型を判定 (6.1 節に出た isinstance 関数を使う) して分岐したりする。

[42] ただし、実数演算の誤差のために、これとぴったり一致する結果にはならない可能性がある。

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。

> クラスに存在する属性 (メソッドや変数) を概観するには、dir 関数が使える。dir 関数は引数にクラスやそのインスタンスを取り、そのクラスやインスタンスに今存在する属性の名前 (暗黙に存在するものを含む) をリストで返してくれる。自分で定義したクラスだけでなく、既存のクラス (int, str などを含む) にも使える (例えば対話モードで「`dir('a')`↵」などとしてみよう。文字列は str クラスのインスタンスなので、str クラスの属性が一覧できる)。

9.2 クラス変数 · クラスメソッド

9.2.1 クラス変数とインスタンス変数

Python では、クラス定義の中で、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。

これに対し、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**すれば、そればインスタンス変数になる。

しかし、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。

以下は、クラス変数とインスタンス変数の区別を例示するための、実用性は全くないプログラム例である。

```
#!/usr/bin/python3
# -*- 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つともクラス変数であり同じもので、値は1
p.n = 2                     # ここでp.nはインスタンス変数になり、値は2になる
print(a.n, p.n, q.n, r.n)   # p.nだけ2、他は1
a.n = 3                     # クラス変数a.nの値を3に変更
print(a.n, p.n, q.n, r.n)   # p.nだけ2、他は3
q.f()                       # ここでq.nもインスタンス変数になり、値は4になる
print(a.n, p.n, q.n, r.n)   # p.nは2、q.nは4、それ以外は3
r.g()                       # クラス変数a.nの値を5に変更
print(a.n, p.n, q.n, r.n)   # p.nは2、q.nは4、それ以外は5
```

実行結果はこうなる。この例では、値が偶数になっているのがインスタンス変数、それ以外がクラス変数である。

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

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。

9.2.2 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**[43]を作りたい場合は、メソッド定義の直前の行に「`@classmethod`」と書く[44]。この場合、メソッドの第 1 引数にはこのメソッドを実行するクラス (つまりこのメソッドを定義しているクラス) が自動的に渡される。この引数の変数名は慣習的に「`cls`」とする。呼び出す側では第 1 引数にクラスは書かない。よって、クラスメソッドでもインスタンスメソッドと同様、メソッド定義側での引数の個数は、呼び出す側での引数の個数より 1 つ多くなる。また、クラスメソッド内で、そのクラスのクラスメソッドやクラス変数にアクセスするには「`cls.`」を付ける。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
class a:
    x = 1
    @classmethod
    def f(cls, x):      # これはクラスメソッド。引数clsにはaが渡される
        cls.x = x           # クラス変数xに引数xを代入
    def g(self, x):     # @classmethodの直後にないのでインスタンスメソッド
        self.x = x

b = a()
print(a.x, b.x) # この時点では2つともクラス変数であり同じもので、値は1
b.f(2)          # クラスメソッドfの呼び出し。クラス変数a.xの値が2になる
print(a.x, b.x) # 両方とも2
a.f(3)          # これもクラスメソッドfの呼び出し。クラス変数a.xの値が3になる
print(a.x, b.x) # 両方とも3
b.g(4)          # インスタンスメソッドgの呼び出し。インスタンス変数b.xができ、値が4になる
print(a.x, b.x) # b.xは4になっているが、a.xは3のまま
```

実行結果を示す。この例では、値が 4 になっているのがインスタンス変数、それ以外がクラス変数である。

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

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 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 (その例として、コンストラクタである `__init__()` や、9.5 節に紹介する `__str__()`、 `__eq__()` などがある)、普通の変数やメソッドにそういう名前を**付けるべきではない**。
2. 1. 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**[45](インスタンス変数 · メソッドであってもクラス変数 · メソッドであっても)。

[43] Python では「クラスメソッド」とは別に「**静的** (スタティック) **メソッド**」(本資料では略) というものも存在し、両者は**異なる**ものだが、Java では「クラスメソッド」と「静的メソッド」は同じものを指すようである。ただし、「クラスメソッド」と「静的メソッド」のどちらも「メソッドの実体がインスタンス毎に作られない」「インスタンスを作らなくても呼べる」「インスタンス変数にアクセスできない」などの点は共通。

[44] 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 (`f` は定義したメソッド。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。

[45] この機能は、そのような変数 · メソッドの名前を、クラス外からは違う名前に見せることで実現されている。例えば 9.3 節のプログラムの `XYCoord` クラスの変数 `__x` は、クラス外からは `_XYCoord__x` という名前に見えるので、外からは `__x` という名ではアクセスできない。変更後の名前を知っていてその名前を使えば、外からアクセスできてしまうのだが、わざわざそうしない限り外からのアクセスは防げるので、これでも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)。

なお、この例でクラス外から `__x` という名の変数を作ることはできる。例えば `a.__x` に `a` の外から値を代入することはできてしまう。しかしこれは、`a` の中の `__x` に見える変数 (外からは `a._XYCoord__x` に見える) とは**別物**である。`a` の中の `__x` に見える変数に外からアクセスしているわけではない。

3. 1. 2. 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。

9.1 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**外部からは隠蔽する**のが望ましい。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-

class XYCoord:  # xy座標を表すクラス
    def __init__(self, x, y):
        self.__x, self.__y = x, y
    def get_x(self): # xのゲッター
        return self.__x
    def get_y(self): # yのゲッター
        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
    def copy(self):
        return XYCoord(self.__x, self.__y)

a = XYCoord(0.3, 0.8)
print(a.get_x(), a.get_y())     # print(a.__x, a.__y)とはできない
b = XYCoord(0.1, 0.2)
c = a.copy()
a.midpoint(b)
print(a.get_x(), a.get_y())
print(c.get_x(), c.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、親クラスのものは隠される (**オーバーライド**)。このとき、親クラスのメソッドを明示的に呼ぶには、「super().」を付けて呼ぶ。なお、Java と違い、コンストラクタも子クラスに継承される。

次の例では、クラス変数 x、インスタンス変数 x、メソッドの引数 x がそれぞれ別な変数であることに注意。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-

class a:
    x = 'Sample'    # クラス変数xが'Sample'になる
    def __init__(self, x):
        self.x = x  # インスタンス変数xに引数のxを代入
    def add(self, x):
        return self.x + x   # インスタンス変数xと引数xの和
    def hi(self, x):
        return ['Hello', x] # 'Hello'と引数xのリスト
```

```
    def no(self):
        return '%04d' % self.x  # インスタンス変数xを'%04d'で書式化
    @classmethod
    def title(cls, x):
        return cls.x + x    # クラス変数xと引数xの文字列連接
class b(a): # クラスbはaの子クラス
    def hi(self, x): # 親クラスのhi()をオーバーライド
        return ['Hi', x]    # 'Hi'と引数xのリスト
    def no(self): # 親クラスのno()をオーバーライド
        return 'No.%s' % super().no()   # 親クラスのインスタンスメソッドno()が呼ばれる
    def mul(self, x): # 子クラス独自のメソッド
        return self.x * x   # インスタンス変数xと引数xの積
    @classmethod
    def title(cls, x): # 親クラスのクラスメソッドtitle()をオーバーライド
        return 'Inherited' + super().title(x)   # 親クラスのクラスメソッドtitle()が呼ばれる

c = b(2)    # aのコンストラクタがbに継承されるので、インスタンス変数c.xが2になる
print(c.add(3), c.hi('there'), c.no(), c.mul(7), b.title('A'))
```

出力は「`5 ['Hi', 'there'] No.0002 14 InheritedSampleA`」となる。最後の行の add メソッドは、クラス b がクラス a から継承したものが使われている。no メソッドはクラス b でオーバーライドしているためクラス b のものが使われるが、その中で b の親クラス a の no メソッドを呼んでいることに注意。クラス b のクラスメソッド title の中でも、親クラス a の title を呼んでいる。

> Python 2.X では、super 関数には引数を与えなければならない。上記の例と同じ動作にするには、インスタンスメソッド (上記の h) 内では super 関数には self.__class__ と self の 2 つを、クラスメソッド (上記の i) 内では super 関数には cls と cls の 2 つ (i 自身の第 1 引数が cls であるとして) を、引数として与える。

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[46](残念ながら未実装は実行時にしか検出できない)。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
class a:
    def f(self, x): # 子クラスでの実装を要求
        raise NotImplementedError       # 例外NotImplementedErrorを発生させる
        # NotImplementedErrorは「実装されていない」を表すPythonの組み込み例外
class b(a): # aの子クラス
    def f(self, x):
        return x*x
c = b()         # もしクラスbにメソッドfが実装されていなければ
print(c.f(3))   # この行でNotImplementedErrorが発生する
```

9.5 特殊メソッド

クラス内に定義しておくと特別な意味を持つメソッドが、コンストラクタ `__init__()` 以外にもいくつかある。それらは「`__`」で始まり「`__`」で終わるメソッド名を持っている。ここではそれらのうち、Java での toString にほぼ相当する `__str__()` と、Java での equals にほぼ相当する `__eq__()` について紹介する。

`__str__()` メソッドは、引数は self だけで、文字列を返すように定義する。このメソッドの返り値は、そのオブジェクトを print() でそのまま出力したり、`%s` で書式化したりしたときに、どのような文字列になるかを決める。

`__eq__()` メソッドは、(self 以外に) 引数を 1 つとり、真偽値を返すように定義する。このメソッドは、オブジェクト自身を `==` や `!=` で他のオブジェクトと比較しようとすると、その相手のオブジェクトを引数として自動的に呼ばれ、その返り値が「等しい」かどうかの判定結果を決める[47]。ちなみに、このメソッドが定義されていなければ、自作クラ

[46] abc というモジュールを使う手段もある (「abc」の名は Abstract Base Class に由来する)。

[47] ちなみに Python 3.X では、`__eq__()` メソッドを定義すると `!=` による比較も自動的にその影響を受けるが、Python 2.X では、`__eq__()` があるだけでは `!=` による比較には影響しない。なお、Python 2.X でも 3.X でも、`__ne__()` メソッドがあれば `!=` による比較はその影響を受ける。

スのオブジェクトは、自分自身 (自分と同一のオブジェクト) とだけ「等しい」という判定になる。
次は、これらのメソッドを 9.3 節のプログラムのクラス定義に追加した例である。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-

class XYCoord:  # xy座標を表すクラス
    def __init__(self, x, y):
        self.__x, self.__y = x, y
    def get_x(self): # xのゲッター
        return self.__x
    def get_y(self): # yのゲッター
        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
    def copy(self):
        return XYCoord(self.__x, self.__y)

    def __str__(self):
        return 'coord (%g, %g)' % (self.__x, self.__y)
    def __eq__(self, v): # x, y座標が同じなら「同じ」と判定
        return self.__x == v.get_x() and self.__y == v.get_y()

a = XYCoord(0.3, 0.8)
print(a)    # __str__()の呼び出し結果が出力される
b = XYCoord(0.1, 0.2)
c = a.copy()
a.midpoint(b)
print(a)
print(c)
d = XYCoord(0.2, 0.5)
if a == d:  # a.__eq__(d)が真を返せば「等しい」、さもなければ「等しくない」と判定される
    print('aとdは同じ点です')
else:
    print('aとdは異なる点です')
```

出力は以下のようになる。a と d の x 座標および y 座標が同じため、a == d は真と判定されている[48]。

```
coord (0.3, 0.8)
coord (0.2, 0.5)
coord (0.3, 0.8)
aとdは同じ点です
```

XYCoord クラスに __eq__() が定義されていないと、a == d の判定は偽となる (同一のオブジェクトでないと「等しい」と判定されないため) ので、出力の最後は「aとdは異なる点です」になる。

[48]ただし、実数演算の誤差のために、本来等しいはずの点が「等しくない」と判定されることはありうる。例えば、本節の例で a の (最初の) 座標が (1.3, 1.8)、b の座標が (1.1, 1.2)、d の座標が (1.2, 1.5) の場合、a (最後の時点での) と d は同じ点のはずだが、演算誤差のため異なる点と判定されてしまう (処理系にもよる)。これは実数を == で比較する限り避けられないことであり、Python の __eq__() メソッドの問題ではない。

9.6 (少しだけ) 大きな例題

次の例も、少し長いため本文には載せず、この PDF の添付ファイルに収めてある。8.1.2 節と同様の方法で取り出した samples ディレクトリ内の、bool_alg というファイルがここでのサンプルである。

このプログラムは、**ブール代数**の論理式を表すクラスを定義したものである。Formula クラスが**論理式**を表すクラス (9.4 節に紹介した “抽象クラス” 的なクラス) で、その子クラスが、「0 や 1」「論理変数だけ」「$\overline{\phi}$ の形」「$\phi \cdot \psi$ の形」「$\phi + \psi$ の形」などといったそれぞれの形の論理式を表す[49]。各クラスには、その論理式に出現する論理変数の集合を返すメソッド vars と、その論理式の真偽値 (0 または 1) を、各論理変数の値を与えて求めるメソッド truth_value が定義されている。上位のクラスで定義されたメソッドは、子クラスにも継承されることに注意。

プログラム中では、クラス定義の後、論理式 $AB + \overline{C} \cdot 1$ を表すインスタンスを作成して変数 f に代入し、その論理式に出現する論理変数の集合 ($\{A, B, C\}$ ということになる) を求めた後、$A = 1, B = 0, C = 0$ を与えてこの論理式の真偽値 (1 である) を求めている。従って、bool_alg を実行すると、出力はこうなる。

```
{'A', 'B', 'C'}     (ただし順番は異なることがある)
1
```

なお、このサンプルプログラムの AndFormula および OrFormula クラスにおいて、インスタンス変数 _subformula1 および _subformula2 の名が「_」1 つで始まっている。その理由は、これらの変数をクラス外からはアクセスさせたくないが、これらのクラスのコンストラクタが親クラス BinOpFormula から継承されており、そちらとの間ではこれらの変数を共通にアクセスできなければならないので、変数名を「__」で始める (9.3 節) わけにはいかないためである (Java でいう protected 相当のアクセス制限を意図しているわけである)。NotFormula クラスの _subformula についても同様。

> このサンプルプログラムでは、「\」による**継続行** (5.1 節) を使っている。また、変数 f への代入は、「**括弧類の中**では自由に改行でき、改行の後は自動的に継続行になる」こと (5.4 節) 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。
>
> なお、プログラム中で not 演算子の結果を int 関数で整数に変換しているのに and や or 演算子の結果をそうしていない理由は、4.2 節で述べたように、not 演算子の結果は True か False になり、一方でこのプログラムは真偽値 (truth_value メソッドの返す値) を 0 か 1 として出したいからである。and や or 演算子は、0 や 1 に対して適用すると結果も 0 か 1 なので、改めて整数に変換する必要がないのである。

[*ちょっとした機能追加*] bool_alg プログラムの Formula クラスに、「この論理式は恒真か否か」を判定する valid メソッドを加えてみよう。ここで、論理式が「**恒真**である」とは、それに出現する論理変数にどんな代入をしても、その論理式の真偽値が「真」(= 1) になることをいう。例えば論理式 $\overline{A} + A$ は、A に何を代入しても値が 1 なので、恒真である。

恒真かどうかの判定方法は、ここでは「その論理式に出現する論理変数へのあらゆる代入について、その論理式の真偽値を計算してみて、全ての場合について 1 だったら恒真」という、**効率は良くないが単純**なものを使うことにしよう。するとまず、論理変数の集合を与えて、その変数に対するあらゆる代入を辞書の形で生成することが必要になるが、これは 5.4.1 節に登場した itertools を zip と組み合わせれば次のようにできる。

```
>>> import itertools↵
>>> a = {'A', 'B', 'C'}↵
>>> len(a)↵
3
>>> list(itertools.product((0, 1), repeat = len(a)))↵
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)]
>>> for i in itertools.product((0, 1), repeat = len(a)):↵
...   print(dict(zip(a, i)))↵                    (先頭に空白が 1 つ以上必要)
... ↵                                            (再度↵を押す)
{'A': 0, 'B': 0, 'C': 0}                         (変数の順番は異なることがあるが、支障ない)
{'A': 0, 'B': 0, 'C': 1}
{'A': 0, 'B': 1, 'C': 0}
{'A': 0, 'B': 1, 'C': 1}
{'A': 1, 'B': 0, 'C': 0}
{'A': 1, 'B': 0, 'C': 1}
{'A': 1, 'B': 1, 'C': 0}
```

[49] 論理否定の「¯」をテキストで書けないため、プログラム (の注釈) 中では $\overline{A}$ を「~A」のように記述している。

```
{'A': 1, 'B': 1, 'C': 1}                    (変数に対する全ての代入が辞書の形で生成された)
```

この考え方を用いて、valid メソッドの実装を実際に追加したものを、samples ディレクトリの bool_alg2 ファイルとして用意してある (bool_alg との違いは `diff -u bool_alg bool_alg2`↵ でわかる)。末尾に、論理式 $\overline{A}+A\overline{B}+B\overline{C}+ABC$ のインスタンスを変数 f に代入してそれが恒真かどうかを求める例が追加されており、実行すると出力には (bool_alg の出力に加えて)「fは恒真です」と出る。

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。

10.1　モジュール作成の例

例えば、9.3 節の XYCoord クラスをモジュールとして独立させ、他のプログラムから呼び出して使えるようにしたいとする。

まず、モジュール名を決めよう。モジュール名の規約は**変数名と同じ** (4 節) である (例えば、2a や a–b というモジュール名は使えない)。ただし、**英小文字と、単語を区切る「_」だけ**からなる名前が**推奨**されている (例えば、a2 や aB というモジュール名は、使うことはできるが推奨されない)。ここでは coord というモジュール名とする。

そして、モジュール名.py という名前 (ここでは coord.py) で、以下のような内容のファイルを作る。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**。ただし、モジュールのファイルとそれを呼び出す側のプログラムとでは、ファイルの文字コードは独立なので[50]、呼び出す側のプログラムとは別に、モジュール側にも**エンコード宣言**は (ファイルの文字コードが UTF-8 である場合を除き) 必要である。

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

class XYCoord:
    def __init__(self, x, y):
        self.__x, self.__y = x, 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
    def copy(self):
        return XYCoord(self.__x, self.__y)

if __name__ == '__main__':
    # テスト用コード; このファイルを単独で python3 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 = XYCoord(0.3, 0.8)
    print(a.get_x(), a.get_y())      # 0.3 0.8 と出力
    b = XYCoord(0.1, 0.2)
```

[50] そうでないと、異なる文字コードで書かれている複数のプログラムから、共通のモジュールを呼び出すことができなくなる。

```
c = a.copy()
a.midpoint(b)
print(a.get_x(), a.get_y())     # 0.2 0.5 と出力
print(c.get_x(), c.get_y())     # 0.3 0.8 と出力
```

なお、coord.py の中の「`if __name__ == '__main__':`」以降の部分は、注釈にもあるように、このファイルが単独で実行された場合のみ実行される部分[51]で、主にモジュールを単独で**動作テスト**するためのコードがここに書かれる。

次に、coord.py ファイルをモジュールとして**呼び出して使う側**のプログラム (名前が pymodtest だとする) 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/python3
# -*- coding: euc-jp -*-
import coord

a = coord.XYCoord(1.3, 1.8)     # 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.XYCoord(1.1, 1.2)
c = a.copy()
a.midpoint(b)
print(a.get_x(), a.get_y())
print(c.get_x(), c.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/python3.6` (`3.6` 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、例えば環境変数 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 (あるいは単に coord.py をカレントディレクトリに置き)、そして

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

とすると、無事実行できて以下の出力を得る[52]。

```
1.3 1.8
1.2 1.5
1.3 1.8
```

ちなみに、モジュールにはクラス定義だけでなく、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。

10.1.1 from 型式のインポート

「`import coord`」でなく「`from coord import XYCoord`」と書くと、coord モジュール内の XYCoord という名前のクラスや関数 (今回の例ではクラス) 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 `from fractions import Fraction` という形で登場した)。これを from 型式のインポートと呼ぶ。

[51] 特別な変数 `__name__` に、単独実行の場合は文字列 `'__main__'` が、モジュールとして呼ばれた場合はモジュール名が入るので、それを用いて条件分岐している。

[52] ここでも、実数演算の誤差のために、これとぴったり一致する結果にはならない可能性がある。

特に、「from coord import *」と書くと、coord モジュール内の全てのクラスや関数を、coord. を付けずに呼べるようになる。しかし、これの**乱用は好ましくない**。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。

今回 10.1 節で作成した coord モジュールには、import で取り込めるものは XYCoord クラス 1 つしかないので、from coord import XYCoord と書いても from coord import * と書いても結果的には同じである。下記では後者を使ってみた。しかし、一般的な応用では、取り込むものを明示的に指定する方がよいだろう。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
from coord import *

a = XYCoord(1.3, 1.8)   # coordモジュールのXYCoordクラスを呼び出す
print(a.get_x(), a.get_y())
b = XYCoord(1.1, 1.2)
a.midpoint(b)
print(a.get_x(), a.get_y())
```

10.2 .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と、自動的にそれがコンパイルされ、コンパイル結果は __pycache__ ディレクトリ (なければ作られる) の ××.*version*.pyc というファイルに書き込まれる (そのファイルに書き込み権限がなかったり、 __pycache__ ディレクトリが作れなかったりする場合を除く)。ここで *version* の部分は Python のバージョンを表す文字列で、例えば Python 3.6 の標準の処理系であれば cpython-36 のようになる。

例えば 10.1 節の例で、coord.py を/home/nide/lib/python に置いたとして、それを一度 Python がモジュールとして読み込むと、コンパイルされて /home/nide/lib/python/__pycache__/coord.cpython-36.pyc というファイルが作られる。

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 インタプリタが高速に読めるような内部形式へのコンパイルである (ので、実行自体は速くならない)。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)。

> なお、Python 2.X の場合は .pyc ファイルの命名規則が異なり、例えば /home/nide/lib/python/coord.py がコンパイルされると /home/nide/lib/python/coord.pyc が作られる (.py ファイルと同じディレクトリに .pyc ファイルが作られる)。

11 既存モジュールの利用

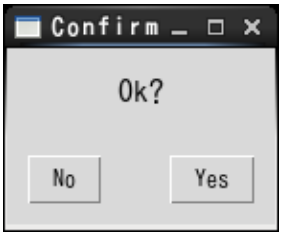
Python にはあらかじめ様々なモジュールが提供されている。これまでにも sys や math などの標準モジュールを利用してきたが、その他にも拡張機能を提供する有用な既存モジュールが多数あり、ここではそのうち数例を紹介する。

11.1 tkinter

Python から Tk の機能を使う tkinter というモジュールがあり[53]、これを使えば Python で **GUI** が書ける。ここでは例だけ挙げる。下記は、No ボタンを押せば戻り値 1 で、Yes ボダンを押せば戻り値 0 で終了するプログラムである。

```
#!/usr/bin/python3
# -*- 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):
```



[53]Python 2.X ではモジュール名は Tkinter である。

```
        print(msg, file = sys.stderr)
        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 = 'right')
    no = tkinter.Button(
        root,
        text = 'No',
        command = lambda: msg_exit('NG', 1)
    )
    no.pack(padx = 10, side = 'left')

    root.mainloop()      # 実行開始
```

なお、GUI ではなくグラフの描画が目的なら、matplotlib という別なモジュールが便利。また、画像処理 (処理した画像の表示も含む) には OpenCV の Python バインディング (モジュール名は cv2) が利用できる。

11.2 シグナル処理 (signal モジュール)

端末から Ctrl-C キーを押した場合に、プログラムが停止するのでなく、何か別の動作を行うようにしたいことがある。

端末で (かつフォアグラウンドで) 実行中のプログラム (プロセス) に対し、Ctrl-C キーを押すと、プロセスに対し SIGINT という「**シグナル**」が送られる。シグナル SIGINT を受け取ったときのプロセスのデフォルトの動作は「プログラムの強制終了」であり[54]、従って通常は Ctrl-C によりプログラムは停止するのである。そこで、Ctrl-C を押したときの動作を変更するには、SIGINT を受けたときのプログラムの動作を変更することになる。

シグナル (SIGINT に限らず) を受けたときのプログラムの動作の変更は、Python では signal モジュールで行う。これは、UNIX 系 OS の signal (あるいは sigaction) システムコールに相当する機能を提供するモジュールである。

このモジュールは以下のように使う。

1. まず、シグナルを受けたときに呼び出される関数 (**シグナルハンドラ**) を用意する。この関数は **2 引数**で、第 1 引数には呼び出されるにあたって受けたシグナル、第 2 引数には実行中のプログラムに関する情報が収められた「フレームオブジェクト」というものが入って呼ばれる。2 引数とも、必ずしも使わなくてもよい。
2. そして signal.signal メソッドを、第 1 引数に捕捉するシグナル (signal モジュールの定数。例えば SIGINT を捕捉するなら signal.SIGINT) を、第 2 引数にシグナルハンドラを渡して呼ぶ。なお第 2 引数には、シグナルハン

[54]ただし Python プログラムの場合は、SIGINT を受け取るとデフォルトでは KeyboardInterrupt という例外が発生し、これによりプログラムが停止する。従って、プログラム中で KeyboardInterrupt を捕捉 (5.5.1 節) することによっても、Ctrl-C を押したときの動作を変更できる。

ドラの他に、特別な定数 signal.SIG_DFL (そのシグナルを受けたときの動作を動作をデフォルトに戻す) または signal.SIG_IGN (そのシグナルを無視させる) も指定可。

次の例は、実行すると 0.5 秒毎に「ok n」(n は数) と表示し続けるが、その n が最初は 5 で、Ctrl-C を 1 回押すたびに n が 4, 3, $\cdots$ のように 1 ずつ減り、n が 0 以下になるとプログラムが終了する、というものである。

数 n と、それを減らす処理とをセットで扱うために、クラス Counter を定義し、Counter のインスタンス変数 __count に n を保持させ、それを減らすメソッド countdown を用意しておく。そして Counter のインスタンス counter を作り、それの countdown メソッドを呼ぶ関数を、SIGINT に対するシグナルハンドラにしている。

この例では、シグナルハンドラは渡された 2 つの引数 (sig と frame) を使っていない。また、この例では signal.signal メソッドで動作を変更しているシグナルは SIGINT だけなので、シグナルハンドラが受け取る第 1 引数 sig は結果的に常に signal.SIGINT ということになる。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
import signal
import time     # time.sleep()を使うため

class Counter:
    def __init__(self, start):
        self.__count = start
    def countdown(self):
        self.__count -= 1
    def get_count(self):
        return self.__count

counter = Counter(5)
def sig_handler(sig, frame):    # シグナルハンドラ
    counter.countdown()
signal.signal(signal.SIGINT, sig_handler)   # SIGINTに対するシグナルハンドラのセット
while True:
    count = counter.get_count()
    print('ok %d' % count)  # 「ok 数」と出力
    if count <= 0:
        break
    time.sleep(.5)          # 0.5秒待つ。時間待ちはこのようにしてできる
```

11.3 スレッド (threading モジュール)

マルチスレッドとは、プログラムの一部を他の部分とは同時並行で動かせる機能をいい。並行して動く各部分を**スレッド**と呼ぶ。スレッドは、複数が同時並行で動く点でプロセスと似ているが、プロセスの方は 1 プロセスが 1 つのプログラムを表すのに対し、スレッドは 1 つのプログラム (プロセス) の中で複数動作できる点が異なる。

Python での**スレッド生成**には threading モジュールを使う。このモジュールを直接呼び出すのではなく、このモジュールにある threading.Thread というクラスの子クラス (名前は何でもいい) を定義し、そのクラスの run メソッドに、スレッドでやらせたい処理を書く。そして、そのクラスのインスタンスを生成し、そのインスタンスの start メソッド (run メソッドではない) を呼べばスレッドが生成される。また、コンストラクタの上書き (その場合、子クラスのコンストラクタは必ず親のコンストラクタを呼ぶことが必要) により、スレッドの生成の際に引数を渡すこともできる。

変数の値などは原則的にスレッド間で共有なので、注意を要する。ただし、スレッドローカルな変数も作れる (略)。

下記の例は、メインのプログラム (主スレッド) から 2 つのスレッドを生成する。2 つのスレッドは主スレッドとは同時並行で動くので、このプログラム全体の動作は時系列的に図 1 の左のようになり、実行には 2.2 秒かかって出力は

```
ok0 0 4
ok0 0 [5, 'Hello']
ok1 0 4
ok1 0 [5, 'Hello']
```

```
ok
ok2 1 4
ok2 1 [5, 'Hello']
```

のようになる。「4」を含むのが最初に生成されたスレッドの、「5」を含むのが 2 番目に生成されたスレッドの出力で、「ok」だけのものが主スレッドの出力である。主スレッドで変数 n の値を変えると、その他のスレッドでも n の値が変わっている。また、主スレッドは、プログラムの終わりに到達しても、他のスレッドが終了するまでは待たされる。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
import threading
import time     # time.sleep()を使うため

class ThreadSample(threading.Thread):
    # lck = threading.Lock()    # ロックオブジェクトを取得しクラス変数に入れる
    def __init__(self, arg):    # 引数を取るコンストラクタを定義
        super().__init__()      # 親のコンストラクタを呼ぶことが必要
        self.arg = arg
    def run(self):
        global n    # クラス外の変数nの値を利用
        # self.lck.acquire()    # ロックの獲得
        print("ok0", n, self.arg)
        time.sleep(.9)
        print("ok1", n, self.arg)
        # self.lck.release()    # ロックの解放
        time.sleep(.7)
        print("ok2", n, self.arg)

n = 0
time.sleep(.3)
ThreadSample(4).start()     # 最初のスレッドを生成
time.sleep(.3)
ThreadSample([5, "Hello"]).start()  # 2番目のスレッドを生成
time.sleep(1.1)
n = 1
print("ok")
```

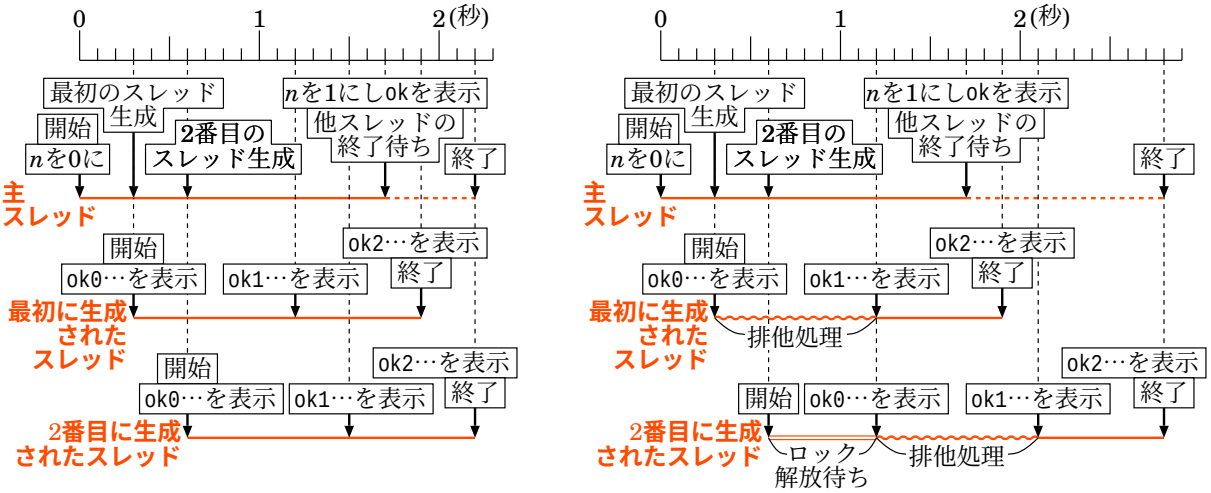


図 1: スレッドを用いたプログラムの動作の時系列の例 (左: 排他処理なし、右: 排他処理あり)

11.3.1 排他処理

スレッド間で**排他処理**を行う[55]には、まず threading モジュールの Lock メソッドを呼んで「ロックオブジェクト」を取得する。ロックオブジェクトの acquire メソッドによりロックを獲得、release メソッドでロックの解放ができる。共有ロックも可能だが、本資料では省く。

11.3 節の例のプログラムは、「#」で始まっている (コメントアウトされている) 行の先頭の「#」を外す (**アンコメント**する) と、排他処理を行う例に変わる (ロックオブジェクトをスレッド間で共有するために、クラス変数に格納している)。その場合、2 番目のスレッドは最初のスレッドがロックを解放するのを待ってからロックを取得するため、動作の時系列は図 1 の右のようになり、実行には 2.8 秒かかって出力は以下のようになる。

```
ok0 0 4
ok1 0 4
ok0 0 [5, 'Hello']
ok
ok2 1 4
ok1 1 [5, 'Hello']
ok2 1 [5, 'Hello']
```

11.4 TCP による通信

socket モジュールを使うと、**ネットワーク通信**を行うプログラムが書ける。ここでは IPv4 上の TCP を利用して通信を行うサーバとクライアントの例を示す。このプログラムもやや長いので本文には載せず、この PDF の添付ファイルに収めてある。8.1.2 節と同様の方法で取り出した samples ディレクトリ内の、tcp_comm というファイルがそれである。

これは「クライアントがサーバに 1 行文字列を送るたびに、それを逆順にした文字列をサーバがクライアントに送り返す」というもので、実行方法はプログラム内の「`if __name__ == '__main__':`」以降のところに注釈で記載してある。例えば、tcp_comm をカレントディレクトリに置いたとして、「`./tcp_comm server 51200`↵」でサーバを実行し、同じマシン (の別端末) で「`./tcp_comm client localhost 51200`↵」でクライアントを実行すると、クライアントからの 1 行入力を逆順にしてサーバが送り返してくる。クライアント側は入力を EOF (Ctrl-D) で終えれば終了する。サーバ側は Ctrl-C で終了する。なお、ネットワーク接続が拒否された場合などのエラー処理は省かれている。

> ここで、51200 は通信に利用する TCP のポート番号である。IANA によると 49152 番以上は「動的·私用ポート」とされているので、ここではその中から 51200 番を使った。クライアントの引数 localhost を他のホスト名に変えれば**他のホスト**のサーバとも通信はできる (が、そのためには指定したポート番号の TCP 通信が途中でフィルタされていないことが必要で、ネットワーク管理の知識が必要な可能性がある)。

Python でネットワーク通信を行う方法は、他の言語 (特に C) と特段の差はない。**サーバ**側は以下の手順を踏む。

1. `soc = socket.socket()` でソケットを生成する。変数名は soc である必要はない。引数を特に指定しなければ、IPv4 上の TCP による通信になる
2. `soc.bind(`*address*`)` により、1. で生成したソケット soc を、引数 *address* に指定した「アドレス」に割り当てる。引数 *address* は、IPv4 の場合はホスト名 (文字列) とポート番号 (整数) のタプル。ホスト名は、接続元を特に限定しない場合は空文字列でよく、その場合 `soc.bind(('', ポート番号))` として呼ぶことになる
3. `soc.listen(`n`)` で、ソケット soc を「接続待ちソケット」にする。引数 n は 0 以上の整数で、接続待ちキューの最大長 (同時に接続可能なクライアントの最大個数ではない) の指定だが、Python 3.5 以降では省略可能であり、省略するとデフォルトの妥当な値が選択されるので、省略する方がよい (ただし、サンプルプログラムでは指定している)
4. `(conn, addr) = soc.accept()` とすると、クライアントからの接続要求を受け付け、接続が来たら、conn に接続済みの新たなソケット、addr には接続元の「アドレス」(2. で `soc.bind` の引数に指定したのと同じ形式のデータ) が入る。変数名は conn や addr でなくてもよい。addr の方は接続元がどこであるかの特定が必要な場合に使うだけで、通信そのものには必要ない

[55] この話はあくまで、同一プロセス内の複数スレッド間での排他処理の話である。異なるプロセスとの間での排他処理は、当然ながらこの方法ではできない。

5. 以後は接続済みソケット conn を使って、クライアントとの間でバイナリデータ (Python のバイト列 (4.3.2 節)) の通信を行う。`conn.recv(4096)` でクライアント側から最大 4096 バイト受信、`conn.sendall(バイト列)` あるいは `conn.send(バイト列)` でクライアントにバイト列を送信できる
6. `conn.close()` で接続済みソケットをクローズする。これで、4. で接続してきたクライアントとの接続を切断
7. 4. ～6. を必要なだけ繰り返し
8. `soc.close()` で接続待ちソケットをクローズ

また、同時に**複数のクライアント**からの接続を受け付けたいならば、上記の 4. ～6. を**マルチスレッド** (11.3 節) で複数同時並行で行う (サンプルプログラムでは、それを行うために thread_call という関数[56]を用意している)。そうしないと、1 つのクライアントと通信している間は他のクライアントからの接続を受け付けることができず、他のクライアントは待たされることになる。

クライアントの方は以下の手順を踏む。

1. サーバの場合の 1. と同じく、`soc = socket.socket()` でソケットを生成する
2. `soc.connect(`*address*`)` でサーバに接続。引数 *address* はサーバの場合の 2. と同じく、ホスト名とポート番号のタプルで、「ホスト名」には接続先のホスト名あるいは IP アドレス (の文字列、例えば「`'127.0.0.1'`」) を指定する (従って、`soc.connect(('127.0.0.1', 51200))` などとすることになる)
3. soc に対して、サーバの場合の 5. と同様に recv や send などのメソッドを使って、サーバとの間でバイト列の通信を行う (すなわち、`soc.recv(4096)` や `soc.send(バイト列)` などとする)
4. `soc.close()` でソケットをクローズ

以上の過程では、サーバとクライアント間のやり取りは、基本的にはソケットを介して**バイナリデータ**で行う。しかし、Python の socket クラスは makefile() というメソッドを提供しており、このメソッドは、ソケットに対する読み書きを行うファイルオブジェクト (8 節) を返す。そのファイルオブジェクトのメソッド (readline() や write() など) を使うことにより、通信相手とのやり取りを、あたかもファイルへの読み書きのように、(Python の) 文字列で行うことができる (ただし、makefile() の引数を `'rwb'` のようにすれば、バイナリファイル用のファイルオブジェクト (8.5 節) を得るので、バイナリでのやり取りも可)。この場合、接続を終えるには、makefile() メソッドで得たファイルオブジェクトを close した後で、ソケット自体も close すればよい。

そこで、サンプルプログラムではこの方法を用いて、あえて文字列ベースでしかも行単位でのやり取りを行う例を示している。具体的には、サンプルプログラムでは SocketFileWrapper というクラスを用意して通信用のソケットをラップしており、この中で上述の makefile() メソッドを用いてファイルオブジェクトを作り出し、それに対して行単位で読み書きするメソッドや、ファイルオブジェクトとソケットを close して通信を終えるメソッドを提供している。

なお、サンプルプログラムはデータを文字列としてやりとりするため、サーバとクライアントとでロケールは (文字コードの設定は) 一致させておく必要がある。

> サンプルプログラムでは、サーバを Ctrl-C で終了する際に、Python の KeyboardInterrupt 例外の発生によるメッセージが出るのを防ぐために、シグナル処理 (11.2 節) も使用している (プログラム中の exit_when_kbdint 関数)。

12　その他の機能

多様な用途でのプログラミングにあたって、使う機会が多いと思われる、あるいは知っておくと便利と思われる、さまざまな機能のうちのいくつかについて、ここで紹介する。

12.1　yield 文によるイテレータ

関数定義の本体の中に「yield 文」(「`yield 式`」という形の文) が書かれていると、その関数は「ジェネレータ関数」となり、その関数の返す値はイテレータ (5.4.1 節) の一種 (ジェネレータイテレータ) となる。そのイテレータは、ジェ

[56] この関数は、サンプルプログラム中では ch_loop 関数に対してしか使っていないが、可変長引数やキーワード引数も含めどのような引数を取る関数に対しても使えるように作ってある。

ネレータ関数が実行中に「yield 式」に行き当たるたびに、その「式」の値を生成する。例を挙げる。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
def f():
    yield 7
    yield 4
for i in f():
    print(i, end = ' ')
print(list(f()))
```

このプログラムでは、関数 f は「7 と 4 をこの順に生成するイテレータ」を返す (f が 2 度呼ばれるので、このイテレータも 2 回生成される)。従って出力は「7 4 [7, 4]」となる。yield 文を使うことで、コルーチンに似たことが書ける。

なお、たとえ yield 文が (if False: の中にあるなどで) 実際には全く実行されなくても、yield 文が本体に書かれてさえいればその関数はジェネレータ関数になる[57]。また、下記のプログラム

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
import time     # time.sleep()を使うため
def f():
    while True:
        yield 7
        yield 4
        yield 5
for i in f():
    print(i)
    time.sleep(.1) # 0.1秒待つ
```

の f が返すイテレータは、無限に値を生成し続ける (その結果、このプログラムは「7」「4」「5」を永遠に繰り返し出力し続ける。無限ループになるので Ctrl-C で強制終了させよう)。

次は、yield を使って、itertools (5.4.1 節) の permutations メソッドと同様のもの (ただし引数はリストとし、生成する順列もリストになる) を定義した例である。

```
#!/usr/bin/python3
# -*- coding: euc-jp -*-
def my_perm(l): # 引数はリストとし、その全ての順列を生成するジェネレータ
    n = len(l)
    if n == 0: # 引数が空リストなら、空リストだけを生成
        yield []
    else:
        for pos in range(n):
            element = l[pos] # リストの中のどこか1箇所の要素
            rest_list = l[:pos] + l[pos+1:] # その要素を除いたリスト
            # rest_listの順列の先頭にelementをつなげたものを生成
            for rest_perm in my_perm(rest_list):
                yield [element] + rest_perm

for i in my_perm([4,2,9]):
    print(i)
```

このプログラムは「[4, 2, 9]」「[4, 9, 2]」··· のように、リスト「[4, 2, 9]」の全ての順列を出力する。

12.2 長い Python プログラムと -c オプション

本節の内容は、Python だけでプログラムを書く場合には不要だが、シェルスクリプトとの併用でプログラムを開発する場合などでは有用な場面があるかもしれない。

[57] この場合、値を 1 度も生成しないイテレータを返すジェネレータ関数ということになる。

シェルスクリプトの中で、他のスクリプト言語を呼び出したい場合が往々にしてある。特に、シェルスクリプト内で、2.1 節で述べた python3 コマンドの -c オプションによって Python プログラムを動かしたいことはありうる。しかし、動かしたい Python プログラムが複数行にわたる場合、シェルスクリプトの中でのインデントが問題となる。例えば

```
import sys
for x in sys.argv[1:]:
    print('引数に「%s」が指定されています' % x)
print('この先にもまだまだ長いPythonプログラムが続きます…')
```

という Python プログラム (日本語 EUC で書かれているとする) を、シェルスクリプト中で -c オプションを使って動かそうとする (引数をいくつかとるものと想定する) と、-c の次の引数として上記プログラムを以下のように

```
#!/bin/sh
LANG=ja_JP.eucJP python3 -c 'import sys
for x in sys.argv[1:]:
    print("引数に「%s」が指定されています" % x)
print("この先にもまだまだ長いPythonプログラムが続きます…")
' 引数1 引数2
echo この先さらに長いシェルスクリプトが続きます…
```

そのままクォートの中に埋め込むことになり、シェルスクリプトの他の部分に対しインデントすることができない。

本題からはそれるが、ここで先頭に「LANG=ja_JP.eucJP」がある理由は、python3 コマンドの -c オプションを使う場合、Python プログラムの文字コードはエンコード宣言 (3 節) ではなく、環境変数 LANG や LC_ALL で決まる[58] ことによる。そのため、ここでは Python プログラム内にエンコード宣言は書かず、LANG に文字コードを指定の上で python3 を起動している (またそのため、シェルスクリプトを起動した環境の LANG の値にかかわらず、Python プログラムからの出力は日本語 EUC となる)。

加えて、Python プログラムを「' '」で囲んでいるため、それとかち合わないよう、プログラム内では文字列を囲むのに「' '」でなく「" "」を使うなどの工夫も行っている。

しかし、これでは Python プログラムの部分を、シェルスクリプトの他の部分と見た目で区別しにくくなる。シェルスクリプト全体にとっては、Python プログラムの中身は一種のローカルブロックなので、その部分とシェルスクリプトの他の部分との区別が、見た目にわかりやすくできることが望まれる。

そこで、そのための対応策として、Python プログラム全体を if True: の中に入れてしまうという手が使える。そうすれば、Python プログラムをインデントできるので、例えば、先のシェルスクリプトは以下のように書き直せる。

```
#!/bin/sh
LANG=ja_JP.eucJP python3 -c 'if True:
    import sys
    for x in sys.argv[1:]:
        print("引数に「%s」が指定されています" % x)
    print("この先にもまだまだPythonプログラムが続きます…")
' 引数1 引数2
echo この先さらに長いシェルスクリプトが続きます…
```

このスクリプトでは、Python プログラムの部分がインデントできたことによって、シェルスクリプトの他の部分との区別が見た目にわかりやすくなっている。

12.3 モジュールとメインプログラムを単一ファイル化

本節の話は Python での開発に必ずしも必要ではないが、開発したプログラムの管理 (特に配布) に使うことができ、特にこの方法で配布されるソフトウェアが実際に存在することから、知っておくと有用な場面があるかもしれない。

Python でのプログラム開発において、モジュール (10 節) を作ると、コマンドラインから直接実行するメインの Python プログラムと、そこから呼び出されるモジュールが分離されるので、Python プログラムのファイルが複数になってしまう。しかし、標準の Python インタプリタには、実行ファイルとして ZIP ファイルを指定されると、その中に格納されている Python プログラムを内部で自動的に展開して実行する機能が備わっている[59]ので、以下のようにすれば、メ

[58] そのように Python のドキュメントに明記してある箇所は見当たらないが、標準の処理系ではそうなるようである。
[59] Java での JAR ファイルの実行と類似の機能と言える。

インプログラムとモジュールを、Python インタプリタが直接実行可能な **1 つのファイル**にまとめることができる。

10.1 節の例で説明する。まず、直接実行するメインプログラム (10.1 節の例では pymodtest という名前であった) を、__main__.py という名前に変更する (モジュールのファイル名はそのままでよい)。

次に、プログラムで使う全てのファイル (__main__.py と、そこから呼び出されるモジュール) を、1 つの **ZIP ファイル**にする。UNIX 系 OS の場合は zip コマンドなどを使えばよいだろう。今回の例では

```
$ zip prog.zip __main__.py coord.py↵
```
(prog.zip を作成し、__main__.py と coord.py を格納)

とする。作成する ZIP ファイルの名前は、prog.zip でなくても適当な名前でよい。

こうしてできた ZIP ファイルは、`python3 prog.zip`↵のように直接 Python プログラムとして実行できる。また、UNIX 系 OS の場合は、そのファイルの**先頭**に、2.3 節で述べた「`#!`」で始まる行を付加すれば、コマンドとして実行できる実行可能ファイルにもなる。それには以下のようにすればよい (作成する実行ファイルの名前が coordtest であるとする。また、prog.zip は先程作った zip ファイルの名前)。

```
$ echo '#!/usr/bin/python3' > coordtest↵
$ cat prog.zip >> coordtest↵
$ chmod a+x coordtest↵
```
(python3 コマンドのフルパスが /usr/bin/python3 の場合)

これで coordtest が**実行可能ファイル**になり、`./coordtest`↵で実行できる。もちろん、__main__.py や coord.py などがなくてもこのファイル単独で動作する。動作に必要なファイルが 1 つで済むので、配布などにも便利である。