1. やる気を高めよう

コンピュータを使って様々な作業をしていたら、自動化したい作業が出てくるでしょう。たとえば、たくさんのテキストファイルで検索-置換操作を行いたい、大量の写真ファイルを込み入ったやりかたでリネームまたは整理したいといったものです。ひょっとすると、小さなカスタムデータベースや、何かに特化したGUIアプリケーション、シンプルなゲームを作りたいかもしれません。

もしあなたがプロのソフト開発者なら、C/C++/Java ライブラリを扱う必要があって、通常の write/compile/test/re-compile サイクルが遅すぎると感じるかもしれません。ひょっとするとそのようなライブラリのテストスイートを書いていて、テスト用のコードを書くのにうんざりしているかもしれません。拡張言語を使えるプログラムを書いていて、アプリケーションのために新しい言語一式の設計と実装をしたくないと思っているかもしれません。

Pythonはそんなあなたのための言語です。

それらの作業の幾つかは、Unix シェルスクリプトや Windows バッチファイルで書くこともできますが、シェルスクリプトはファイル操作やテキストデータの操作には向いているもののGUIアプリケーションやゲームにはむいていません。C/C++/Java プログラムを書くこともできますが、最初の試し書きにすらかなりの時間がかかってしまいます。Pythonは簡単に利用でき、Windows、Mac OS X、そして Unix オペレーティングシステムで動作し、あなたの作業を素早く行う助けになるでしょう。

Pyhon は簡単に利用できますが、本物のプログラミング言語であり、シェルスクリプトやバッチファイルで提供されるよりもたくさんの、大規模プログラム向けの構造やサポートを提供しています。一方、Python は C よりたくさんのエラーチェックを提供しており、超高級言語*(very-high-level language)* であり、可変長配列や辞書などの高級な型を組込みで持っています。そのような型は一般的なため、Python は Awk や Perl が扱うものより (多くの場合、少なくともそれらの言語と同じくらい簡単に)大規模な問題に利用できます。

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

Python はインタプリタ言語です。コンパイルやリンクが必要ないので、プログラムを開発する際にかなりの時間を節約できます。インタプリタは対話的な使い方もできます。インタプリタは対話的にも使えるので、言語の様々な機能について実験してみたり、書き捨てのプログラムを書いたり、ボトムアップでプログラムを開発する際に関数をテストしたりといったことが簡単にできます。便利な電卓にもなります。

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

- 高レベルのデータ型によって、複雑な操作を一つの実行文で表現できます。
- 実行文のグループ化を、グループの開始や終了の括弧ではなくインデントで行えます。
- 変数や引数の宣言が不要です。

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

ところで、この言語は BBC のショー番組、」モンティパイソンの空飛ぶサーカス (Monty Python's Flying Circus)」 から取ったもので、爬虫類とは関係ありません。このドキュメントでは、モンティパイソンの寸劇への参照が許可されているだけでなく、むしろ推奨されています！

さて、皆さんはもう Python にワクワクして、もうちょっと詳しく調べてみたくなったはずです。プログラミング言語を習得する最良の方法は使ってみることですから、このチュートリアルではみなさんが読んだ内容を Python インタプリタで試してみることをおすすめします。

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

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

2. Python インタプリタを使う

2.1. インタプリタを起動する

The Python interpreter is usually installed as /usr/local/bin/python3.5 on those machines where it is available; putting /usr/local/bin in your Unix shell's search path makes it possible to start it by typing the command:

```
python3.5
```

[1] どのディレクトリに Python インタプリタをインストールするかはインストール時に選択できるので、インタプリタは他のディレクトリにあるかもしれません; 身近な Python に詳しい人か、システム管理者に聞いてみてください。 (例えば、その他の場所としては /usr/local/python が一般的です。)

On Windows machines, the Python installation is usually placed in C:¥Python35, though you can change this when you're running the installer. To add this directory to your path, you can type the following command into the command prompt in a DOS box:

```
set path=%path%;C:¥python35
```

ファイル終端文字 (Unixでは Control-D 、DOS や Windows では Control-Z) を一次プロンプト (訳注: 『>>>』 のこと) に入力すると、インタプリタが終了ステータス 0 で終了します。もしこの操作がうまく働かないなら、コマンド: quit() と入力すればインタプリタを終了できます。

readline をサポートしているシステム上では、対話的行編集やヒストリ置換、コード補完のインタプリタの行編集機能が利用できます。コマンドライン編集機能がサポートされているかを最も手っ取り早く調べる方法は、おそらく最初に表示された Python プロンプトに Control-P を入力してみることでしょう。ビープ音が鳴るなら、コマンドライン編集機能があります。編集キーについての解説は付録 対話入力編集と履歴置換 を参照してください。何も起こらないように見えるか、 ^P がエコーバックされるなら、コマンドライン編集機能は利用できません。この場合、現在編集中の行から文字を削除するにはバックスペースを使うしかありません。

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

インタプリタを起動する第二の方法は python -c command [arg] ... です。この形式では、シェルの -c オプションと同じように、 *command* に指定した文を実行します。 Python 文には、スペースなどのシェルにとって特殊な意味をもつ文字がしばしば含まれるので、 *command* 全体をシングルクォート(訳注: ')で囲っておいたほうが良いでしょう。

Python のモジュールには、スクリプトとしても便利に使えるものがあります。 python -mmodule [arg] ... のように起動すると、 *module* のソースファイルを、フルパスを指定して起動したかのように実行できます。

スクリプトファイルを使用する場合、スクリプトを走らせて、そのまま対話モードに入れると便利なことがあります。これには -i をスクリプトの前に追加します。

全てのコマンドラインオプションは コマンドラインと環境 で説明されています。

2.1.1. 引数の受け渡し

スクリプト名と引数を指定してインタプリタを起動した場合、スクリプト名やスクリプト名以後に指定した引数は、文字列のリストに変換されて sys モジュールの argv 変数に格納されます。 import sys することでこのリストにアクセスすることができます。 sys.argv には少なくとも一つ要素が入っています。スクリプト名も引数も指定しなければ sys.argv[0] は空の文字列になります。スクリプト名の代わりに '-' (標準入力を意味します) を指定すると、 sys.argv[0] は '-' になります。 -c *command* を使うと、 sys.argv[0] は '-c' になります。 -m *module* を使った場合、 sys.argv[0] はモジュールのフルパスになります。 -c*command* や -m *module* の後ろにオプションを指定した場合、 Python インタプリタ自体はこれらの引数を処理せず、 sys.argv を介して *command* や *module* から扱えるようになります。

2.1.2. 対話モード

インタプリタが命令を端末 (tty) やコマンドプロンプトから読み取っている場合、インタプリタは 対話モード *(interactive mode)* で動作しているといいます。このモードでは、インタプリタは 一次プロンプト *(primary prompt)* を表示して、ユーザにコマンドを入力するよう促します。一次プロンプトは普通、三つの「大なり記号」 (>>>) です。一つの入力が次の行まで続く (行継続: continuation line を行う) 場合、インタプリタは 二次プロンプト *(secondary prompt)* を表示します。二次プロンプトは、デフォルトでは三つのドット (...) です。インタプリタは、最初のプロンプトを出す前にバージョン番号と著作権表示から始まる起動メッセージを出力します:

```
$ python3.5
Python 3.5 (default, Sep 16 2015, 09:25:04)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

行継続は、例えば以下の if 文のように、複数の行からなる構文を入力するときに必要です:

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

対話モードについての詳細は 対話モード を参照してください。

2.2. インタプリタとその環境

2.2.1. ソースコードの文字コード

デフォルトでは、Python のソースコードは UTF-8 でエンコードされているものとして扱われます。UTF-8 では、世界中のほとんどの言語の文字を同時に文字列リテラルや識別子やコメント中に書くことができます。— ただし、標準ライブラリは識別子に ASCII 文字のみを利用していて、その他のポータブルなコードもその慣習に従うべきです。それらの文字を正しく表示するためには、エディターはそのファイルが UTF-8 である事を識別して、そのファイルに含まれている文字を全てサポートしたフォントを使わなければなりません。

It is also possible to specify a different encoding for source files. In order to do this, put one more special comment line right after the #! line to define the source file encoding:

```
# -*- coding: encoding -*-
```

With that declaration, everything in the source file will be treated as having the encoding *encoding* instead of UTF-8. The list of possible encodings can be found in the Python Library Reference, in the section on `codecs`.

For example, if your editor of choice does not support UTF-8 encoded files and insists on using some other encoding, say Windows-1252, you can write:

```
# -*- coding: cp-1252 -*-
```

and still use all characters in the Windows-1252 character set in the source files. The special encoding comment must be in the *first or second* line within the file.

3. 形式ばらない Python の紹介

以下の例では、入力と出力は (>>> や ...) といったプロンプトの有無で区別します: 例を実際に試してみるためには、プロンプトが表示されているときに、例中のプロンプトから後ろの内容全てを入力しなければなりません; プロンプトが先頭にない行はインタプリタからの出力です。例中には2つ目のプロンプトだけが表示されている行がありますが、これは空行を入力しなければならないことを意味しています; 空行の入力は複数の行からなる命令の終わりをインタプリタに教えるために使われます。

このマニュアルにある例の多くは、対話プロンプトで入力されるものでもコメントを含んでいます。 Python におけるコメント文はハッシュ文字 # で始まり、物理行の終わりまで続きます。コメントは行の先頭にも、空白やコードの後にも書くことができますが、文字列リテラルの内部に置くことはできません。文字列リテラル中のハッシュ文字はただのハッシュ文字です。コメントはコードを明快にするためのものであり、Pythonはコメントを解釈しません。なので、コードサンプルを実際に入力して試して見るときは、コメントをを省いても大丈夫です。

いくつかの例です:

```
# this is the first comment
spam = 1  # and this is the second comment
          # ... and now a third!
text = "# This is not a comment because it's inside quotes."
```

3.1. Python を電卓として使う

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

3.1.1. 数

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

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # division always returns a floating point number
1.6
```

整数 (例えば、 2 、 4 、 20) は int 型を持ち、小数部を持つ数 (例えば、 5.0 、 1.6) は float 型を持ちます。数値型については後のチュートリアルでさらに見ていきます。

除算 (/) は常に浮動小数点数を返します。 floor division を行い、(小数部を切り捨てて) 整数を返すには、 // 演算子を使うことができます; 剰余を計算するには % を使うことができます:

```
>>> 17 / 3  # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3  # floor division discards the fractional part
5
>>> 17 % 3  # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2  # result * divisor + remainder
17
```

Python では、冪乗を計算するのに ** 演算子が使えます [1]:

```
>>> 5 ** 2  # 5 squared
25
>>> 2 ** 7  # 2 to the power of 7
128
```

等号 (=) は変数に値を代入するときに使います。代入を行っても、次のプロンプトの手前には結果は出力されません:

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

変数が 「定義」 されて (つまり値が代入されて) いない場合、その変数を使おうとするとエラーが発生します:

```
>>> n  # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

浮動小数点を完全にサポートしています。演算対象の値(オペランド)の型が統一されていない場合、演算子は整数のオペランドを浮動小数点型に変換します:

```
>>> 4 * 3.75 - 1
14.0
```

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

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
```

```
>>> round(_, 2)
113.06
```

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

int と float に加え、 Python は Decimal や Fraction のような他の数値型をサポートしています。 Python はビルトインで 複素数 もサポートし、 j もしくは J 接尾辞を使って虚部を示します (例。 3+5j)。

3.1.2. 文字列型 (string)

数のほかに、Python は文字列の操作ができ、文字列はいくつもの方法で表現できます。文字列は単引用符 ('...') もしくは二重引用符 ("...") で囲み、結果はどちらも同じ文字列になります。[2] ¥ は引用符をエスケープするのに使います:

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> 'doesn¥'t'  # use ¥' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "¥"Yes,¥" he said."
'"Yes," he said.'
>>> '"Isn¥'t," she said.'
'"Isn¥'t," she said.'
```

対話的インタプリタでは、出力文字列は引用符に囲まれ、特殊文字はバックスラッシュでエスケープされます。これは入力とは違って見える (囲っている引用符が変わる) こともありますが、その 2 つの文字列は同じ文字列です。文字列が単引用符を含み二重引用符を含まない場合、二重引用符で囲われ、それ以外の場合は単引用符で囲われます。 print() 関数は、囲っている引用符を落とし、エスケープした特殊文字を出力することで、より読み易い出力を作成します:

```
>>> '"Isn¥'t," she said.'
'"Isn¥'t," she said.'
>>> print('"Isn¥'t," she said.')
"Isn't," she said.
>>> s = 'First line.¥nSecond line.'  # ¥n means newline
>>> s  # without print(), ¥n is included in the output
'First line.¥nSecond line.'
>>> print(s)  # with print(), ¥n produces a new line
First line.
Second line.
```

特殊文字として解釈させるために文字の前に ¥ を付けることをしたくない場合は、最初の引用符の前に r を付けた *raw strings* が使えます:

```
>>> print('C:¥some¥name')  # here ¥n means newline!
C:¥some
ame
>>> print(r'C:¥some¥name')  # note the r before the quote
C:¥some¥name
```

文字列リテラルは複数行にまたがって書けます。1 つの方法は三連引用符 ("""...""" や '''...''') を使うことです。行末は自動的に文字列に含まれますが、行末に ¥ を付けることで含めないようにすることもできます。次の例:

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

は次のような出力になります (最初の改行文字は含まれていないことに注意してください):

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

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

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

連続して並んでいる複数の 文字列リテラル (つまり、引用符に囲われた文字列) は自動的に連結されます。

```
>>> 'Py' 'thon'
'Python'
```

これは 2 つのリテラルどうしに対してのみ働き、変数や式には働きません:

```
>>> prefix = 'Py'
>>> prefix 'thon'  # can't concatenate a variable and a string literal
  ...
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
  ...
SyntaxError: invalid syntax
```

変数どうしや変数とリテラルを連結したい場合は、+ を使ってください:

```
>>> prefix + 'thon'
'Python'
```

この機能は特に、長い文字列を改行したいときに役に立ちます:

```
>>> text = ('Put several strings within parentheses '
...         'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

文字列は インデクス表記 (添字表記) することができ、最初の文字のインデクスは 0 になります。文字列型と区別された文字型はありません; 文字というのは単なる長さが 1 の文字列です:

```
>>> word = 'Python'
>>> word[0]  # character in position 0
'P'
>>> word[5]  # character in position 5
'n'
```

インデクスは負の数を指定しても良く、そのときは右から数えていきます:

```
>>> word[-1]  # last character
'n'
>>> word[-2]  # second-last character
'o'
>>> word[-6]
'P'
```

-0 は 0 と等しいので、負のインデクスは -1 から始まることに注意してください。

インデクス表記に加え、スライス もサポートされています。インデクス表記は個々の文字を取得するのに使いますが、スライス を使うと部分文字列を取得することができます:

```
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
```

開始インデクスは常に含まれ、終了インデクスは常に含まれないことに注意してください。なので s[:i] + s[i:] が常に s と等しい、ということになります:

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

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

```
>>> word[:2]   # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:]   # characters from position 4 (included) to the end
'on'
>>> word[-2:]  # characters from the second-last (included) to the end
'on'
```

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

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

1行目の数字は文字列の 0 から 6 までのインデクスの位置を示しています; 2行目は対応する負のインデクスを示しています。*i* から *j* までのスライスは、それぞれ *i* と付いた境界から *j*と付いた境界までの全ての文字から成っています。

非負のインデクス対の場合、スライスされたシーケンスの長さは、スライスの両端のインデクスが範囲内にあるかぎり、インデクス間の差になります。例えば、 word[1:3] の長さは 2 になります。

大き過ぎるインデクスを使おうとするとエラーが発生します:

```
>>> word[42]  # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

しかし、スライスで範囲外のインデクスを使ったときは上手く対応して扱ってくれます:

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

Python の文字列は変更できません – つまり不変 (immutable) なのです。従って、文字列のインデクスで指定したある場所に代入を行うとエラーが発生します:

```
>>> word[0] = 'J'
  ...
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
  ...
TypeError: 'str' object does not support item assignment
```

元の文字列と別の文字列が必要な場合は、新しく文字列を作成してください:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

組込み関数 len() は文字列の長さ (length) を返します:

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

参考

テキストシーケンス型 — str

文字列は シーケンス型 の例であり、シーケンス型でサポートされている共通の操作をサポートしています。

文字列メソッド

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

書式指定文字列の文法

str.format() を使った文字列のフォーマットについての情報があります。

printf 形式の文字列書式化

文字列が % 演算子の左オペランドである場合に呼び出される古いフォーマット操作について、詳しく記述されています。

3.1.3. リスト型 (list)

Python 多くの 複合 *(compound)* データ型を備えており、複数の値をまとめるのに使われます。最も汎用性が高いのは リスト *(list)* で、コンマ区切りの値 (要素) の並びを角括弧で囲んだものとして書き表されます。リストは異なる型の要素を含むこともありますが、通常は全ての要素は同じ型を持ちます。

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

文字列 (や他の全てのビルトインのシーケンス (sequence) 型) のように、リストはインデクス表記やスライス表記ができます:

```
>>> squares[0]  # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

全てのスライス操作は指定された要素を含む新しいリストを返します。これは、次のスライスはリストの新しい (浅い) コピーを返すことを意味します:

```
>>> squares[:]
[1, 4, 9, 16, 25]
```

リストは文字列の連結に似た操作もサポートしています:

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

不変 (immutable) な文字列とは違って、リストは可変 (mutable) 型、つまり含んでいる要素を取り替えることができます:

```
>>> cubes = [1, 8, 27, 65, 125]  # something's wrong here
>>> 4 ** 3  # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

append() を使って、リストの末尾に新しい要素を追加することもできます (このメソッドについては後で詳しく見ていきます):

```
>>> cubes.append(216)  # add the cube of 6
>>> cubes.append(7 ** 3)  # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

スライスに代入することもできます。スライスの代入を行って、リストのサイズを変更したり、完全に消すことさえできます:

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

組込み関数 len() はリストにも適用できます:

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

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

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
```

```
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3.2. プログラミングへの第一歩

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

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
...     print(b)
...     a, b = b, a+b
...
1
1
2
3
5
8
```

上の例では、いくつか新しい機能を取り入れています。

- 最初の行には 複数同時の代入 *(multiple assignment)* が入っています: 変数 a と b は、それぞれ同時に新しい値 0 と 1 になっています。この代入は最後の行でも再度使われており、代入が行われる前に右辺の式がまず評価されます。右辺の式は左から右へと順番に評価されます。

- `while` は、条件 (ここでは `b < 10`) が真である限り実行を繰り返し (ループし) ます。Python では、C 言語と同様に、ゼロでない整数値は真となり、ゼロは偽です。条件式は文字列値やリスト値、実際には任意のシーケンス型でもかまいません。 1つ以上の長さのシーケンスは真で、空のシーケンスは偽になります。例中で使われている条件テストはシンプルな比較です。標準的な比較演算子は C 言語と同様です: すなわち、 `<` (より小さい)、 `>` (より大きい)、 `==` (等しい)、 `<=` (より小さいか等しい)、 `>=` (より大きいか等しい)、および `!=` (等しくない)、です。

- ループの 本体 *(body)* は インデント *(indent,* 字下げ*)* されています: インデントは Python において実行文をグループにまとめる方法です。対話的プロンプトでは、インデントされた各行を入力するにはタブや (複数個の) スペースを使わなければなりません。実際には、Python へのより複雑な入力を準備するにはテキストエディタを使うことになるでしょう; ほとんどのテキストエディタは自動インデント機能を持っています。複合文を対話的に入力するときには、(パーザはいつ最後の行を入力したのか推し量ることができないので) 入力の完了を示すために最後に空行を続けなければなりません。基本ブロックの各行は同じだけインデントされていなければならないので注意してください。

- `print()` 関数は与えられた引数の値を書き出します。これは (前に電卓の例でやったような) 単に出力したい式を書くのとは、複数の引数や浮動小数点量や文字列に対する扱い方が違います。文字列は引用符無しで出力され、要素の間に空白が挿入されて、このように出力の書式が整えられます:

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

キーワード引数 *end* を使うと、出力の末尾に改行文字を出力しないようにしたり、別の文字列を末尾に出力したりできます:

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

4. その他の制御フローツール

先ほど紹介した while 文の他にも、 Python は他の言語でおなじみの制御フロー文を備えていますが、これらには多少ひねりを加えてあります。

4.1. if 文

おそらく最もおなじみの文型は if 文でしょう。例えば:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

ゼロ個以上の elif 部を使うことができ、 else 部を付けることもできます。キーワード 『elif』 は 『else if』 を短くしたもので、過剰なインデントを避けるのに役立ちます。一連の if ... elif ... elif ... は、他の言語における switch 文や case 文の代用となります。

4.2. for 文

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

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

ループ内部でイテレートしているシーケンスを修正する必要があれば (例えば選択されたアイテムを複製するために)、最初にコピーを作ることをお勧めします。シーケンスに対するイテレーションは暗黙にコピーを作りません。スライス記法はこれを特に便利にします:

```
>>> for w in words[:]:  # Loop over a slice copy of the entire list.
...     if len(w) > 6:
...         words.insert(0, w)
...
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

4.3. range() 関数

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

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

指定した終端値は生成されるシーケンスには入りません。range(10) は 10 個の値を生成し、長さ 10 のシーケンスにおける各項目のインデクスとなります。range を別の数から開始したり、他の増加量 (負でも; 増加量は時に 『ステップ(step)』 と呼ばれることもあります) を指定することもできます:

```
range(5, 10)
   5 through 9

range(0, 10, 3)
   0, 3, 6, 9

range(-10, -100, -30)
  -10, -40, -70
```

あるシーケンスにわたってインデクスで反復を行うには、 range() と len() を次のように組み合わせられます:

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

しかし、多くの場合は enumerate() 関数を使う方が便利です。 ループのテクニック を参照してください。

range を直接出力すると変なことになります:

```
>>> print(range(10))
range(0, 10)
```

range() が返すオブジェクトは、いろいろな点でリストであるかのように振る舞いますが、本当はリストではありません。これは、イテレートした時に望んだ数列の連続した要素を返すオブジェクトです。しかし実際にリストを作るわけではないので、スペースの節約になります。

このようなオブジェクトは イテラブル *(iterable)* と呼ばれます。これらは関数やコンストラクタのターゲットとして、あるだけの項目を逐次与えるのに適しています。 for 文がそのような イテレータ であることはすでに見てきました。関数 list() もまた一つの例です。これはイテラブルからリストを生成します:

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

後ほど、イテラブルを返したりイテラブルを引数として取る関数をもっと見ていきます。

4.4. break 文と continue 文とループの else 節

The break statement, like in C, breaks out of the innermost enclosing for or while loop.

ループ文は else 節を持つことができます。これは、 (for で) 反復処理対象のリストを使い切ってループが終了したとき、または (while で) 条件が偽になったときに実行されますが、 break 文でループが終了したときは実行されません。この動作を、素数を探す下記のループを例にとって示します:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(そう、これは正しいコードです。よく見てください: else 節は if 文 **ではなく** 、 for ループに属しています。)

ループの else 句は、 if 文の else よりも try 文の else に似ています。 try 文の else 句は例外が発生しなかった時に実行され、ループの else 句は break されなかった場合に実行されます。 try 文と例外についての詳細は 例外を処理する を参照してください。

continue 文も C 言語から借りてきたもので、ループの次のイテレーションを実行します:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Found an even number", num)
...         continue
...     print("Found a number", num)
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

4.5. pass 文

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

```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

これは最小のクラスを作るときによく使われる方法です:

```
>>> class MyEmptyClass:
...     pass
...
```

pass が使われるもう1つの場所は、新しいコードを書いている時の関数や条件文の中身です。こうすることで、具体的なコードを書かないで抽象的なレベルで考えることができます。 pass は何もすることなく無視されます:

```
>>> def initlog(*args):
...     pass   # Remember to implement this!
...
```

4.6. 関数を定義する

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

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
```

```
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

def は関数の 定義 *(definition)* を導くキーワードです。 def の後には、関数名と仮引数を丸括弧で囲んだリストを続けなければなりません。関数の実体を構成する実行文は次の行から始め、インデントされていなければなりません。

関数の本体の記述する文の最初の行は文字列リテラルにすることもできます。その場合、この文字列は関数のドキュメンテーション文字列 (documentation string)、または *docstring* と呼ばれます。 (docstring については ドキュメンテーション文字列 でさらに扱っています。) ドキュメンテーション文字列を使ったツールには、オンライン文書や印刷文書を自動的に生成したり、ユーザが対話的にコードから直接閲覧できるようにするものがあります。自分が書くコードにドキュメンテーション文字列を入れるのはよい習慣です。書く癖をつけてください。

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

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

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

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

他の言語出身の人からは、 fib は値を返さないので関数ではなく手続き (procedure) だと異論があるかもしれませんね。技術的に言えば、実際には return 文を持たない関数もややつまらない値ですが値を返しています。この値は None と呼ばれます (これは組み込みの名前です)。 None だけを書き出そうとすると、インタプリタは通常出力を抑制します。本当に出力したいのなら、以下のように print() を使うと見ることができます:

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

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

```
>>> def fib2(n):  # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

この例は Python の新しい機能を示しています:

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

4.7. 関数定義についてもう少し

可変個の引数を伴う関数を定義することもできます。引数の定義方法には 3 つの形式があり、それらを組み合わせることができます。

4.7.1. デフォルトの引数値

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

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

この関数はいくつかの方法で呼び出せます:

- 必須の引数のみ与える: ask_ok('Do you really want to quit?')
- 一つのオプション引数を与える: ask_ok('OK to overwrite the file?', 2)
- 全ての引数を与える: ask_ok('OK to overwrite the file?', 2, 'Come on, only yesor no!')

この例では in キーワードが導入されています。このキーワードはシーケンスが特定の値を含んでいるかどうか調べるのに使われます。

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

```
i = 5

def f(arg=i):
    print(arg)

i = 6
f()
```

は 5 を出力します。

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

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

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

このコードは、以下を出力します

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

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

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

4.7.2. キーワード引数

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

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

は、必須引数 (voltage) とオプション引数 (state、action、type) を受け付けます。この関数は以下のいずれかの方法で呼び出せます:

```
parrot(1000)                                          # 1 positional argument
parrot(voltage=1000)                                  # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')             # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)             # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump')         # 3 positional arguments
parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1 keyword
```

が、以下の呼び出しは不適切です:

```
parrot()                     # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument after a keyword argument
parrot(110, voltage=220)     # duplicate value for the same argument
parrot(actor='John Cleese')  # unknown keyword argument
```

関数の呼び出しにおいて、キーワード引数は位置指定引数の後でなければなりません。渡されるキーワード引数は全て、関数で受け付けられる引数のいずれかに対応していなければならず (例えば、actor はこの parrot 関数の引数として適切ではありません)、順序は重要ではありません。これはオプションでない引数でも同様です (例えば、parrot(voltage=1000) も適切です)。いかなる引数も値を複数回は受け取れません。この制限により失敗する例は:

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

仮引数の最後に **name の形式のものがあると、それまでの仮引数に対応したものを除くすべてのキーワード引数が入った辞書 (マッピング型 — dict を参照) を受け取ります。 **nameは *name の形式をとる、仮引数のリストを超えた位置指定引数の入ったタプルを受け取る引数 (次の節で述べます) と組み合わせることができます。 (*name は **name より前になければなりません)。例えば、ある関数の定義を以下のようにすると:

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

```
    keys = sorted(keywords.keys())
    for kw in keys:
        print(kw, ":", keywords[kw])
```

呼び出しは以下のようになり:

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

もちろん以下のように出力されます:

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

Note that the list of keyword argument names is created by sorting the result of the keywords dictionary's `keys()` method before printing its contents; if this is not done, the order in which the arguments are printed is undefined.

4.7.3. 任意引数リスト

最後に、最も使うことの少ない選択肢として、関数が任意の個数の引数で呼び出せるよう指定する方法があります。これらの引数はタプル (タプルとシーケンス を参照) に格納されます。可変個の引数の前に、ゼロ個かそれ以上の引数があっても構いません。

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

通常このような *可変* 引数は、関数に渡される入力引数の残りを全て掬い取るために、仮引数リストの最後に置かれます。*args 引数の後にある仮引数は 『キーワード専用』 引数で、位置指定引数ではなくキーワード引数としてのみ使えます。

```
>>> def concat(*args, sep="/"):
...     return sep.join(args)
...
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```

4.7.4. 引数リストのアンパック

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

```
>>> list(range(3, 6))            # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))            # call with arguments unpacked from a list
[3, 4, 5]
```

同じやりかたで、** オペレータを使って辞書でもキーワード引数を渡すことができます:

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' dem
```

4.7.5. ラムダ式

キーワード `lambda` を使うと、名前のない小さな関数を生成できます。例えば `lambda a, b:a+b` は、二つの引数の和を返す関数です。ラムダ式の関数は、関数オブジェクトが要求されている場所にならどこでも使うことができます。ラムダ式は、構文上単一の式に制限されています。意味付け的には、ラムダ形式は単に通常の関数定義に構文的な糖衣をかぶせたものに過ぎません。入れ子構造になった関数定義と同様、ラムダ式もそれを取り囲むスコープから変数を参照することができます:

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

上記の例は、関数を返すところでラムダ式を使っています。もう1つの例では、ちょっとした関数を引数として渡すのに使っています:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

4.7.6. ドキュメンテーション文字列

ドキュメンテーション文字列については、その内容と書式に関する慣習をいくつか挙げます。

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

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

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

以下に複数行のドキュメンテーション文字列の例を示します:

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

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

4.7.7. 関数のアノテーション

関数のアノテーション はユーザ定義関数で使用される型についての完全にオプションなメタデータ情報です (詳細は **PEP 484** を参照してください)。

アノテーションは関数の __annotations__ 属性に辞書として格納され、関数の他の部分には何も影響がありません。パラメータアノテーションは、パラメータ名の後にコロンを続けることによって定義され、その後にアノテーションの値として評価される式が置かれます。戻り値アノテーションは、パラメータリストと def ステートメントの終わりを表すコロンの間に置かれたリテラル -> によって定義され、その後に式が続きます。次の例は位置引数とキーワード引数、そして戻り値アノテーションを持っています:

```
>>> def f(ham: str, eggs: str = 'eggs') -> str:
...     print("Annotations:", f.__annotations__)
...     print("Arguments:", ham, eggs)
...     return ham + ' and ' + eggs
...
>>> f('spam')
Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}
Arguments: spam eggs
'spam and eggs'
```

4.8. 間奏曲: コーディングスタイル

これからより長くより複雑な Python のコードを書いていくので、そろそろ コーディングスタイル について語っても良い頃です。ほとんどの言語は様々なスタイルで書け (もっと簡潔に言えば フォーマットでき)、スタイルによって読み易さが異なります。他人にとって読み易いコードにしようとするのはどんなときでも良い考えであり、良いコーディングスタイルを採用することが非常に強力な助けになります。

Python には、ほとんどのプロジェクトが守っているスタイルガイドとして **PEP 8** があります。それは非常に読み易く目に優しいコーディングスタイルを推奨しています。全ての Python 開発者はある時点でそれを読むべきです。ここに最も重要な点を抜き出しておきます:

- インデントには空白 4 つを使い、タブは使わないこと。

  空白 4 つは (深くネストできる) 小さいインデントと (読み易い) 大きいインデントのちょうど中間に当たります。タブは混乱させるので、使わずにおくのが良いです。

- ソースコードの幅が 79 文字を越えないように行を折り返すこと。

  こうすることで小さいディスプレイを使っているユーザも読み易くなり、大きなディスプレイではソースコードファイルを並べることもできるようになります。

- 関数やクラスや関数内の大きめのコードブロックの区切りに空行を使うこと。

- 可能なら、コメントは行に独立で書くこと。

- docstring を使うこと。

- 演算子の前後とコンマの後には空白を入れ、括弧類のすぐ内側には空白を入れないこと: `a = f(1, 2) + g(3, 4)`。

- クラスや関数に一貫性のある名前を付けること。慣習では CamelCase をクラス名に使い、 lower_case_with_underscores を関数名やメソッド名に使います。常に self をメソッドの第 1 引数の名前 (クラスやメソッドについては クラス初見 を見よ) として使うこと。

- あなたのコードを世界中で使ってもらうつもりなら、風変りなエンコーディングは使わないこと。どんな場合でも、Python のデフォルト UTF-8 またはプレーン ASCII が最も上手くいきます。

- 同様に、ほんの少しでも他の言語を話す人がコードを読んだりメンテナンスする可能性があるのであれば、非 ASCII 文字も識別子に使うべきではありません。

5. データ構造

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

5.1. リスト型についてもう少し

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

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

list.**extend**(*iterable*)
イテラブルのすべての要素を対象のリストに追加し、リストを拡張します。a[len(a):]= iterable と等価です。

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

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

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

list.**clear**()
リスト中の全ての要素を削除します。del a[:] と等価です。

list.**index**(*x*[, *start*[, *end*]])
リスト中で、値 *x* を持つ最初の要素の位置をゼロから始まる添字で返します。 該当する要素がなければ、ValueError を送出します。

任意の引数である *start* と *end* はスライス記法として解釈され、リストの探索範囲を指定できます。返される添字は、*start* 引数からの相対位置ではなく、リスト全体の先頭からの位置になります。

list.**count**(*x*)
リストでの *x* の出現回数を返します。

list.**sort**(*key=None*, *reverse=False*)
リストの項目を、インプレース演算 (in place、元のデータを演算結果で置き換えるやりかた) でソートします。引数はソート方法のカスタマイズに使えます。 sorted() の説明を参照してください。

list.**reverse**()
リストの要素を、インプレース演算で逆順にします。

list.**copy**()
リストの浅い (shallow) コピーを返します。a[:] と等価です。

以下にリストのメソッドをほぼ全て使った例を示します:

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # Find next banana starting a position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```

insert, remove, sort などのリストを操作するメソッドの戻り値が表示されていないことに気が付いたかもしれません。これらのメソッドは None を返しています。[1] これは Python の変更可能なデータ構造全てについての設計上の原則となっています。

5.1.1. リストをスタックとして使う

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

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

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

5.1.2. リストをキューとして使う

リストをキュー (queue) として使うことも可能です。この場合、最初に追加した要素を最初に取り出します (「first-in, first-out」)。しかし、リストでは効率的にこの目的を達成することが出来ません。追加（append）や取り出し（pop）をリストの末尾に対して行うと速いのですが、挿入（insert）や取り出し（pop）をリストの先頭に対して行うと遅くなってしまいます（他の要素をひとつずつずらす必要があるからです）。

キューの実装には、 `collections.deque` を使うと良いでしょう。このクラスは良く設計されていて、高速な追加（append）と取り出し（pop）を両端に対して実現しています。例えば以下のようにします:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3. リストの内包表記

リスト内包表記はリストを生成する簡潔な手段を提供しています。主な利用場面は、あるシーケンスや iterable (イテレート可能オブジェクト) のそれぞれの要素に対してある操作を行った結果を要素にしたリストを作ったり、ある条件を満たす要素だけからなる部分シーケンスを作成することです。

例えば、次のような平方のリストを作りたいとします:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

これはループが終了した後にも存在する x という名前の変数を作る (または上書きする) ことに注意してください。以下のようにして平方のリストをいかなる副作用もなく計算することができます:

```
squares = list(map(lambda x: x**2, range(10)))
```

もしくは、以下でも同じです:

```
squares = [x**2 for x in range(10)]
```

これはより簡潔で読みやすいです。

リスト内包表記は、括弧の中の 式、 `for` 句、そして0個以上の `for` か `if` 句で構成されます。リスト内包表記の実行結果は、 `for` と `if` 句のコンテキスト中で式を評価した結果からなる新しいリストです。例えば、次のリスト内包表記は2つのリストの要素から、違うもの同士をペアにします:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

これは次のコードと等価です:

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

`for` と `if` 文が両方のコードで同じ順序になっていることに注目してください。

式がタプルの場合 (例: 上の例で式が (x, y) の場合) は、タプルに円括弧が必要です。

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
```

```
  File "<stdin>", line 1, in <module>
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

リスト内包表記の式には、複雑な式や関数呼び出しのネストができます:

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

5.1.4. ネストしたリストの内包表記

リスト内包表記中の最初の式は任意の式なので、そこに他のリスト内包表記を書くこともできます。

次の、長さ4のリスト3つからなる、3x4 の matrix について考えます:

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

次のリスト内包表記は、matrix の行と列を入れ替えます:

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

前の節で見たように、ネストしたリスト内包表記は、続く `for` のコンテキストの中で評価されます。なので、この例は次のコードと等価です:

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

これをもう一度変換すると、次のコードと等価になります:

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

実際には複雑な流れの式よりも組み込み関数を使う方が良いです。この場合 `zip()` 関数が良い仕事をしてくれるでしょう:

```
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

この行にあるアスタリスクの詳細については 引数リストのアンパック を参照してください。

5.2. del 文

リストから要素を削除する際、値を指定する代わりにインデックスを指定する方法があります。それが `del` 文です。これは `pop()` メソッドと違い、値を返しません。 `del` 文はリストからスライスを除去したり、リスト全体を削除することもできます(以前はスライスに空のリストを代入して行っていました)。例えば以下のようにします:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` は変数全体の削除にも使えます:

```
>>> del a
```

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

5.3. タプルとシーケンス

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

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

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
```

```
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

ご覧のとおり、タプルの表示には常に丸括弧がついていて、タプルのネストが正しく解釈されるようになっています。タプルを書くときは必ずしも丸括弧で囲まなくてもいいですが、(タプルが大きな式の一部だった場合は) 丸括弧が必要な場合もあります。タプルの要素を代入することはできません。しかし、タプルにリストのような変更可能型を含めることはできます。

タプルはリストと似ていますが、たいてい異なる場面と異なる目的で利用されます。タプルは不変型 (immutable) で、複数の型の要素からなることもあり、要素はアンパック(この節の後半に出てきます)操作やインデックス (あるいは namedtuples の場合は属性)でアクセスすることが多いです。一方、リストは変更可能 (mutable) で、要素はたいてい同じ型のオブジェクトであり、たいていイテレートによってアクセスします。

問題は 0 個または 1 個の項目からなるタプルの構築です。これらの操作を行うため、構文には特別な細工がされています。空のタプルは空の丸括弧ペアで構築できます。一つの要素を持つタプルは、値の後ろにコンマを続ける (単一の値を丸括弧で囲むだけでは不十分です) ことで構築できます。美しくはないけれども、効果的です。例えば以下のようにします:

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

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

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

この操作は、シーケンスのアンパック *(sequence unpacking)* とでも呼ぶべきもので、右辺には全てのシーケンス型を使うことができます。シーケンスのアンパックでは、等号の左辺に列挙されている変数が、右辺のシーケンスの長さと同じ数だけあることが要求されます。複数同時の代入が実はタプルのパックとシーケンスのアンパックを組み合わせたものに過ぎないことに注意してください。

5.4. 集合型

Python には、 集合 *(set)* を扱うためのデータ型もあります。集合とは、重複する要素をもたない、順序づけられていない要素の集まりです。 Set オブジェクトは、和 (union)、積 (intersection)、差 (difference)、対称差 (symmetric difference)といった数学的な演算もサポートしています。

中括弧、または `set()` 関数は set を生成するために使用することができます。注: 空集合を作成するためには `set()` を使用しなければなりません (`{}` ではなく)。後者は空の辞書を作成します。辞書は次のセクションで議論するデータ構造です。

簡単なデモンストレーションを示します:

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in either a or b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

リスト内包 と同様に、 set 内包もサポートされています:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

5.5. 辞書型 (dictionary)

もう一つ、有用な型が Python に組み込まれています。それは 辞書 *(dictionary)* (マッピング型 — dict を参照)です。辞書は他の言語にも 「連想記憶 (associated memory)」 や 「連想配列 (associative array)」 という名前で存在することがあります。ある範囲の数でインデクス化されているシーケンスと異なり、辞書は キー *(key)* でインデクス化されています。このキーは何らかの変更不能な型になります。文字列、数値は常にキーにすることができます。タプルは、文字列、数値、その他のタプルのみを含む場合はキーにすることができます。直接、あるいは間接的に変

更可能なオブジェクトを含むタプルはキーにできません。リストをキーとして使うことはできません。これは、リストにスライスやインデクス指定の代入を行ったり、 append() や extend() のようなメソッドを使うと、インプレースで変更することができるためです。

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

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

辞書オブジェクトに list(d.keys()) を実行すると、辞書で使われている全てのキーからなるリストを適当な順番で返します (ソートされたリストが欲しい場合は、代わりに sorted(d.keys()) を使ってください)。 [2] ある単一のキーが辞書にあるかどうか調べるには、 in キーワードを使います。

以下に、辞書を使った簡単な例を示します:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> list(tel.keys())
['irv', 'guido', 'jack']
>>> sorted(tel.keys())
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

dict() コンストラクタは、キーと値のペアのタプルを含むリストから辞書を生成します:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

さらに、辞書内包表現を使って、任意のキーと値のペアから辞書を作れます:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

キーが単純な文字列の場合、キーワード引数を使って定義する方が単純な場合もあります:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

5.6. ループのテクニック

辞書に対してループを行う際、 items() メソッドを使うと、キーとそれに対応する値を同時に取り出せます。

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

シーケンスにわたるループを行う際、 enumerate() 関数を使うと、要素のインデックスと要素を同時に取り出すことができます。

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

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

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

シーケンスを逆方向に渡ってループするには、まずシーケンスの範囲を順方向に指定し、次いで関数 reversed() を呼び出します。

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

シーケンスをソートされた順序でループするには、 sorted() 関数を使います。この関数は元の配列を変更せず、ソート済みの新たな配列を返します。

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```

ときどきループ内でリストを変更したい誘惑に駆られるでしょうが、代わりに新しいリストを作ってしまうほうがより簡単で安全なことが、ままあります

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```

5.7. 条件についてもう少し

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

比較演算子 in および not in は、ある値があるシーケンス中に存在するか (または存在しないか) どうかを調べます。演算子 is および is not は、二つのオブジェクトが実際に同じオブジェクトであるかどうかを調べます。この比較は、リストのような変更可能なオブジェクトにだけ意味があります。全ての比較演算子は同じ優先順位を持っており、ともに数値演算子よりも低い優先順位となります。(訳注: is は、 is None のように、シングルトンの変更不能オブジェクトとの比較に用いる場合もあります。(「変更可能なオブジェクトにだけ意味があります」の部分を削除することを Doc-SIG に提案中。))

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

ブール演算子 and や or で比較演算を組み合わせることができます。そして、比較演算 (あるいは何らかのブール式) の結果の否定は not でとれます。これらの演算子は全て、比較演算子よりも低い優先順位になっています。 A and not B or C と (A and (not B)) or C が等価になるように、ブール演算子の中で、 not の優先順位が最も高く、 or が最も低くなっています。もちろん、丸括弧を使えば望みの組み合わせを表現できます。

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

比較や他のブール式の結果を変数に代入することもできます。例えば、

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

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

5.8. シーケンスとその他の型の比較

シーケンスオブジェクトは同じシーケンス型の他のオブジェクトと比較できます。比較には 辞書的な *(lexicographical)* 順序が用いられます。まず、最初の二つの要素を比較し、その値が等しくなければその時点で比較結果が決まります。等しければ次の二つの要素を比較し、以降シーケンスの要素が尽きるまで続けます。比較しようとする二つの要素がいずれも同じシーケンス型であれば、そのシーケンス間での辞書比較を再帰的に行います。二つのシーケンスの全ての要素の比較結果が等しくなれば、シーケンスは等しいとみなされます。片方のシーケンスがもう一方の先頭部分にあたる部分シーケンスならば、短い方のシーケンスが小さいシーケンスとみなされます。文字列に対する辞書的な順序づけには、個々の文字を並べるのに Unicode コードポイントの数を用います。以下に、同じ型のオブジェクトを持つシーケンス間での比較を行った例を示します:

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

違う型のオブジェクト同士を < や > で比較することも、それらのオブジェクトが適切な比較メソッドを提供しているのであれば許可されます。例えば、異なる数値型同士の比較では、その数値によって比較が行われます。例えば、 0 と 0.0 は等価です。一方、適切な比較順序がない場合は、インタープリターは TypeError 例外を発生させます。

6. モジュール (module)

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

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

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

```
# Fibonacci numbers module

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

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

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

```
>>> import fibo
```

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

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

関数を度々使うのなら、ローカルな名前に代入できます:

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

6.1. モジュールについてもうすこし

モジュールには、関数定義に加えて実行文を入れることができます。これらの実行文はモジュールを初期化するためのものです。これらの実行文は、インポート文の中で 最初に モジュール名が見つかったときにだけ実行されます。[1] (ファイルがスクリプトとして実行される場合も実行されます。)

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

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

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

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

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

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

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

この書き方ではアンダースコア (_) で始まるものを除いてすべての名前をインポートします。殆どの場面で、Python プログラマーはこの書き方を使いません。未知の名前がインタープリターに読み込まれ、定義済みの名前を上書きしてしまう可能性があるからです。

一般的には、モジュールやパッケージから * を import するというやり方には賛同できません。というのは、この操作を行うとしばしば可読性に乏しいコードになるからです。しかし、対話セッションでキータイプの量を減らすために使うのは構わないでしょう。

注釈 実行効率上の理由で、各モジュールはインタープリタの 1 セッションごとに 1 回だけ import されます。従って、モジュールを修正した場合には、インタープリタを再起動させなければなりません – もしくは、その場で手直ししてテストしたいモジュールが 1 つだった場合には、例えば `import importlib; importlib.reload(modulename)` のように `importlib.reload()` を使ってください。

6.1.1. モジュールをスクリプトとして実行する

Python モジュールを

```
python fibo.py <arguments>
```

と実行すると、 `__name__` に `__main__` が設定されている点を除いて import したときと同じようにモジュール内のコードが実行されます。つまりモジュールの末尾に:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

このコードを追加することで、このファイルが import できるモジュールであると同時にスクリプトとしても使えるようになります。なぜならモジュールが 「main」 ファイルとして起動されたときだけ、コマンドラインを解釈するコードが実行されるからです:

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

モジュールが import された場合は、そのコードは実行されません:

```
>>> import fibo
>>>
```

この方法はモジュールに便利なユーザインターフェースを提供したり、テストのために (スクリプトをモジュールとして起動しテストスイートを実行して) 使われます。

6.1.2. モジュール検索パス

`spam` という名前のモジュールをインポートするとき、インタープリターはまずその名前のビルトインモジュールを探します。見つからなかった場合は、 `spam.py` という名前のファイルを `sys.path` にあるディレクトリのリストから探します。 `sys.path` は以下の場所に初期化されます:

- 入力されたスクリプトのあるディレクトリ (あるいはファイルが指定されなかったときはカレントディレクトリ)。
- `PYTHONPATH` (ディレクトリ名のリスト。シェル変数の `PATH` と同じ構文)。
- インストールごとのデフォルト。

注釈 シンボリックリンクをサポートするファイルシステム上では、入力されたスクリプトのあるディレクトリはシンボリックリンクをたどった後に計算されます。言い換えるとシンボリックリンクを含むディレクトリはモジュール検索パスに追加 **されません**。

初期化された後、 Python プログラムは `sys.path` を修正することができます。スクリプトファイルを含むディレクトリが検索パスの先頭、標準ライブラリパスよりも前に追加されます。なので、ライブラリのディレクトリにあるファイルよりも、そのディレクトリにある同じ名前のスクリプトが優先してインポートされます。これは、標準ライブラリを意図して置き換えているのでない限りは間違いのもとです。より詳しい情報は 標準モジュール を参照してください。

6.1.3. 「コンパイル」 された Python ファイル

モジュールの読み込みを高速化するため、 Python はコンパイル済みの各モジュールを `__pycache__` ディレクトリの `module.version.pyc` ファイルとしてキャッシュします。ここで version はコンパイルされたファイルのフォーマットを表すもので、一般的には Python のバージョン番号です。例えば、 CPython のリリース 3.3 の、コンパイル済みの spam.py は `__pycache__/spam.cpython-33.pyc` としてキャッシュされるでしょう。この命名の慣習により、Python の異なる複数のリリースやバージョンのコンパイル済みモジュールが共存できます。

Python はソースの変更日時をコンパイル済みのものと比較し、コンパイル済みのものが最新でなくなり再コンパイルが必要になっていないかを確認します。これは完全に自動で処理されます。また、コンパイル済みモジュールはプラットフォーム非依存なため、アーキテクチャの異なるシステム間で同一のライブラリを共有することもできます。

Python は2つの場合にキャッシュのチェックを行いません。ひとつは、コマンドラインから直接モジュールが読み込まれた場合で、常に再コンパイルされ、結果を保存することはありません。2つめは、ソース・モジュールのない場合で、キャッシュの確認を行いません。ソースのない (コンパイル済みのもののみの) 配布をサポートするには、コンパイル済みモジュールはソース・ディレクトリになくてはならず、ソース・ディレクトリにソース・モジュールがあってはいけません。

エキスパート向けのTips:

- コンパイル済みモジュールのサイズを小さくするために、Python コマンドに `-O` または `-OO` スイッチを使うことができます。`-O` スイッチは assert ステートメントを除去し、`-OO` スイッチは assert ステートメントと `__doc__` 文字列を除去します。いくつかのプログラムはこれらの除去されるものに依存している可能性があるため、自分が何をしているかを理解しているときに限ってこれらのオプションを使うべきです。」最適化」 されたモジュールは `opt-` タグを持ち、通常のコンパイル済みモジュールよりサイズが小さくなります。将来のリリースでは最適化の影響が変わる可能性があります。
- `.pyc` ファイルや `.pyo` ファイルから読み出されたとしても、プログラムは `.py` ファイルから読み出されたときより何ら高速に動作するわけではありません。`.pyc` ファイルで高速化されるのは、読み込みにかかる時間だけです。
- `compileall` モジュールを使ってディレクトリ内の全てのモジュールに対して .pyc ファイルを作ることができます。
- フローチャートなど、この処理に関する詳細が **PEP 3147** に記載されています。

6.2. 標準モジュール

Python は標準モジュールライブラリを同梱していて、別の Python ライブラリリファレンスというドキュメントで解説しています。幾つかのモジュールは言語のコアにはアクセスしないものの、効率や、システムコールなどOSの機能を利用するために、インタープリター内部にビルトインされています。そういったモジュールセットはまたプラットフォームに依存した構成オプションです。例えば、 winreg モジュールは Windows システムでのみ提供されています。 1つ注目に値するモジュールとして、 sys モジュールは、全ての Python インタープリターにビルトインされています。 sys.ps1 と sys.ps2 という変数は一次プロンプトと二次プロンプトに表示する文字列を定義しています:

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

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

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

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

6.3. dir() 関数

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

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__loader__', '__name__',
 '__package__', '__stderr__', '__stdin__', '__stdout__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_getframe',
 '_home', '_mercurial', '_xoptions', 'abiflags', 'api_version', 'argv',
 'base_exec_prefix', 'base_prefix', 'builtin_module_names', 'byteorder',
 'call_tracing', 'callstats', 'copyright', 'displayhook',
 'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix',
 'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
 'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
 'getrefcount', 'getsizeof', 'getswitchinterval', 'gettotalrefcount',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path',
 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
 'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionlimit',
 'setswitchinterval', 'settrace', 'stderr', 'stdin', 'stdout',
 'thread_info', 'version', 'version_info', 'warnoptions']
```

引数がなければ、 dir() は現在定義している名前を列挙します:

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

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

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

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
 'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
 'ValueError', 'Warning', 'ZeroDivisionError', '_', '__build_class__',
 '__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs',
 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable',
 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',
 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr',
 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass',
 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview',
 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property',
 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars',
 'zip']
```

6.4. パッケージ

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

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

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

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

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

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

```
import sound.effects.echo
```

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

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

サブモジュールを import するもう一つの方法を示します:

```
from sound.effects import echo
```

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

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

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

```
from sound.effects.echo import echofilter
```

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

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

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

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

6.4.1. パッケージから * を import する

それでは、ユーザが from sound.effects import * と書いたら、どうなるのでしょうか？理想的には、何らかの方法でファイルシステムが調べられ、そのパッケージにどんなサブモジュールがあるかを調べ上げ、全てを import する、という処理を望むことでしょう。これには長い時間がかかってしまうこともありますし、あるサブモジュールを import することで、そのモジュールが明示的に import されたときのみ発生して欲しい副作用が起きてしまうかもしれません。

唯一の解決策は、パッケージの作者にパッケージの索引を明示的に提供させる というものです。 import 文の使う規約は、パッケージの __init__.py コードに __all__ という名前のリスト

が定義されていれば、 from package import * が現れたときに import すべきモジュール名のリストとして使う、というものです。 パッケージの新バージョンがリリースされるときにリストを最新の状態に更新するのは パッケージの作者の責任となります。 自分のパッケージから * を import するという使い方が考えられないならば、 パッケージの作者はこの使い方をサポートしないことにしてもかまいません。 例えば、ファイル sound/effects/__init__.py には、次のような コードを入れてもよいかもしれません:

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

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

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

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

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

特定のモジュールでは import * を使ったときに、特定のパターンに従った名前のみを公開 (export) するように設計されてはいますが、それでもやはり製品のコードでは良いことではないと考えます。

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

6.4.2. パッケージ内参照

パッケージが (前述の例の sound パッケージのように) サブパッケージの集まりに構造化されている場合、絶対 import を使って兄弟関係にあるパッケージを参照できます。例えば、モジュール sound.filters.vocoder で sound.effects パッケージの echo モジュールを使いたいとすると、 from sound.effects import echo を使うことができます。

また、明示的な相対importを from module import name の形式の import 文で利用できます。この明示的な相対 import では、先頭のドットで現在および親パッケージを指定します。 surround モジュールの例では、以下のように記述できます:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

相対 import は現在のモジュール名をベースにすることに注意してください。メインモジュールの名前は常に "__main__" なので、Python アプリケーションのメインモジュールとして利用されることを意図しているモジュールでは絶対 import を利用するべきです。

6.4.3. 複数ディレクトリ中のパッケージ

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

この機能はほとんど必要にはならないのですが、パッケージ内存在するモジュール群を拡張するために使うことができます。

7. 入力と出力

プログラムの出力方法にはいくつかの種類があります。 データを人間が読める形で出力することもあれば、将来使うためにファイルに書くこともあります。 この章では、こうした幾つかの出力の方法について話します。

7.1. 出力を見やすくフォーマットする

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

Often you'll want more control over the formatting of your output than simply printing space-separated values. There are two ways to format your output; the first way is to do all the string handling yourself; using string slicing and concatenation operations you can create any layout you can imagine. The string type has some methods that perform useful operations for padding strings to a given column width; these will be discussed shortly. The second way is to use the `str.format()` method.

`string` モジュールの `Template` クラスも、文字列中の値を置換する別の方法を提供しています。

もちろん、一つ問題があります。値をどうやって文字列に変換したらいいのでしょうか？幸運なことに、Python には値を文字列に変換する方法があります。値を `repr()` か `str()` 関数に渡してください。

`str()` 関数は値の人間に読める表現を返すためのもので、 `repr()` 関数はインタープリタに読める (あるいは同値となる構文がない場合は必ず `SyntaxError` になるような) 表現を返すためのものです。人間が読むのに適した特定の表現を持たないオブジェクトにおいては、 `str()` は `repr()` と同じ値を返します。数値や、リストや辞書を始めとするデータ構造など、多くの値がどちらの関数に対しても同じ表現を返します。一方、文字列は、2つの異なる表現を持っています。

例:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
... hello = 'hello, world¥n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world¥n'
>>> # The argument to repr() may be any Python object:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

以下に 2 乗と 3 乗の値からなる表を書く二つの方法を示します:

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Note use of 'end' on previous line
...     print(repr(x*x*x).rjust(4))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000

>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

(最初の例で、各カラムの間のスペース一個は `print()` により追加されていることに注意してください。 `print()` は引数間に常に空白を追加します。)

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

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

```
>>> '12'.zfill(5)
'00012'
```

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

str.format() メソッドの基本的な使い方は次のようなものです:

```
>>> print('We are the {} who say "{}!"'.format('knights', 'Ni'))
We are the knights who say "Ni!"
```

括弧とその中の文字(これをフォーマットフィールドと呼びます)は、 str.format() メソッドに渡されたオブジェクトに置換されます。括弧の中の数字は str.format() メソッドに渡されたオブジェクトの位置を表すのに使えます。

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

str.format() メソッドにキーワード引数が渡された場合、その値はキーワード引数の名前によって参照されます。

```
>>> print('This {food} is {adjective}.'.format(
...       food='spam', adjective='absolutely horrible'))
This spam is absolutely horrible.
```

順序引数とキーワード引数を組み合わせて使うこともできます:

```
>>> print('The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred',
                                                       other='Georg'))
The story of Bill, Manfred, and Georg.
```

ascii() を適応する '!a' や str() を適応する '!s' や repr() を適応する '!r' を使って、値をフォーマットする前に変換することができます:

```
>>> contents = 'eels'
>>> print('My hovercraft is full of {}.'.format(contents))
My hovercraft is full of eels.
>>> print('My hovercraft is full of {!r}.'.format(contents))
My hovercraft is full of 'eels'.
```

オプションの ':' とフォーマット指定子を、フィールド名の後ろに付けることができます。フォーマット指定子によって値がどうフォーマットされるかを制御することができます。次の例では、円周率πを、小数点以下3桁で丸めてフォーマットしています。

```
>>> import math
>>> print('The value of PI is approximately {0:.3f}.'.format(math.pi))
The value of PI is approximately 3.142.
```

':' の後ろに整数をつけると、そのフィールドの最低の文字幅を指定できます。この機能は綺麗なテーブルを作るのに便利です。

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print('{0:10} ==> {1:10d}'.format(name, phone))
...
Jack       ==>       4098
Dcab       ==>       7678
Sjoerd     ==>       4127
```

もしも長い書式文字列があり、それを分割したくない場合には、変数を引数の位置ではなく変数の名前で参照できるとよいでしょう。これは、辞書を引数に渡して、角括弧 '[]' を使って辞書のキーを参照することで可能です

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
...       'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

table を 『**』 記法を使ってキーワード引数として渡す方法もあります。

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

全てのローカルな変数が入った辞書を返す組み込み関数 vars() と組み合わせると特に便利です。

str.format() による文字列書式設定の完全な解説は、 書式指定文字列の文法 を参照してください。

7.1.1. 古い文字列書式設定方法

% 演算子を使って文字列書式設定をする方法もあります。これは、演算子の左側の sprintf() スタイルのフォーマット文字列に、演算子の右側の値を適用し、その結果の文字列を返します。例えば:

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

より詳しい情報は printf 形式の文字列書式化 にあります。

7.2. ファイルを読み書きする

open() は file object を返します。大抵、 open(filename, mode) のように二つの引数を伴って呼び出されます。

```
>>> f = open('workfile', 'w')
```

最初の引数はファイル名の入った文字列です。二つめの引数も文字列で、ファイルをどのように使うかを示す数個の文字が入っています。 *mode* は、ファイルが読み出し専用なら 'r'、書き込

み専用 (同名の既存のファイルがあれば消去されます) なら 'w' とします。 'a' はファイルを追記用に開きます。ファイルに書き込まれた内容は自動的にファイルの終端に追加されます。 'r+' はファイルを読み書き両用に開きます。 *mode* 引数は省略可能で、省略された場合には 'r' であると仮定します。

通常、ファイルはテキストモード (*text mode*) で開かれ、特定のエンコーディングでエンコードされたファイルに対して文字列を読み書きします。エンコーディングが指定されなければ、デフォルトはプラットフォーム依存です (`open()` を参照してください)。モードに 'b' をつけるとファイルをバイナリモード (*binary mode*) で開き、 byte オブジェクトを読み書きします。テキストファイル以外のすべてのファイルはバイナリモードを利用するべきです。

テキストモードの読み取りでは、プラットフォーム固有の行末記号 (Unix では ¥n 、Windows では ¥r¥n) をただの ¥n に変換するのがデフォルトの動作です。テキストモードの書き込みでは、 ¥n が出てくる箇所をプラットフォーム固有の行末記号に戻すのがデフォルトの動作です。この裏で行われるファイルデータの変換はテキストファイルには上手く働きますが、 JPEG ファイルや EXE ファイルのようなバイナリデータを破壊する恐れがあります。そのようなファイルを読み書きする場合には注意して、バイナリモードを使うようにしてください。

It is good practice to use the `with` keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point. Using `with` is also much shorter than writing equivalent `try-finally` blocks:

```
>>> with open('workfile') as f:
...     read_data = f.read()
>>> f.closed
True
```

If you're not using the `with` keyword, then you should call `f.close()` to close the file and immediately free up any system resources used by it. If you don't explicitly close a file, Python's garbage collector will eventually destroy the object and close the open file for you, but the file may stay open for a while. Another risk is that different Python implementations will do this clean-up at different times.

After a file object is closed, either by a `with` statement or by calling `f.close()`, attempts to use the file object will automatically fail.

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

7.2.1. ファイルオブジェクトのメソッド

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

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

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

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

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

ファイルから複数行を読み取るには、ファイルオブジェクトに対してループを書く方法があります。この方法はメモリを効率的に使え、高速で、簡潔なコードになります:

```
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
Second line of the file
```

ファイルのすべての行をリスト形式で読み取りたいなら、 `list(f)` や `f.readlines()` を使うこともできます。

`f.write(string)` は、 *string* の内容をファイルに書き込み、書き込まれた文字数を返します。

```
>>> f.write('This is a test¥n')
15
```

オブジェクトの他の型は、書き込む前に変換しなければなりません – 文字列 (テキストモード) と bytes オブジェクト (バイナリーモード) のいずれかです:

```
>>> value = ('the answer', 42)
>>> s = str(value)  # convert the tuple to string
>>> f.write(s)
18
```

`f.tell()` は、ファイルオブジェクトのファイル中における現在の位置を示す整数を返します。ファイル中の現在の位置は、バイナリモードではファイルの先頭からのバイト数で、テキストモードでは不明瞭な値で表されます。

ファイルオブジェクトの位置を変更するには、f.seek(offset, from_what) を使います。ファイル位置は基準点 (reference point) にオフセット値 *offset* を足して計算されます。参照点は *from_what* 引数で選びます。*from_what* の値が 0 ならばファイルの 先頭から測り、1 ならば現在のファイル位置を使い、2 ならばファイルの終端を参照点として使います。*from_what*は省略することができ、デフォルトの値は 0、すなわち参照点としてファイルの先頭を使います。

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

テキストファイル (mode 文字列に b を付けなかった場合) では、ファイルの先頭からの相対位置に対するシークだけが許可されています (例外として、seek(0, 2) でファイルの末尾へのシークは可能です)。また、唯一の有効な *offset* 値は f.tell() から返された値か、0 のいずれかです。それ以外の *offset* 値は未定義の振る舞いを引き起こします。

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

7.2.2. json による構造化されたデータの保存

文字列は簡単にファイルに書き込んだり、ファイルから読み込んだりすることができます。数値の場合には少し努力が必要です。というのも、read() メソッドは文字列しか返さないため、int() のような関数にその文字列を渡して、たとえば文字列 '123' のような文字列を、数値 123 に変換しなくてはならないからです。もっと複雑なデータ型、例えば入れ子になったリストや辞書の場合、手作業でのパースやシリアライズは困難になります。

ユーザが毎回コードを書いたりデバッグしたりして複雑なデータ型をファイルに保存するかわりに、Python では一般的なデータ交換形式である JSON (JavaScript Object Notation) を使うことができます。この標準モジュール json は、Python のデータ 階層を取り、文字列表現に変換します。この処理は シリアライズ *(serializing)* と呼ばれます。文字列表現からデータを再構築することは、デシリアライズ *(deserializing)* と呼ばれます。シリアライズされてからデシリアライズされるまでの間に、オブジェクトの文字列表現はファイルやデータの形で保存したり、ネットワークを通じて離れたマシンに送ったりすることができます。

> **注釈** JSON 形式は現代的なアプリケーションでデータをやりとりする際によく使われます。多くのプログラマーが既に JSON に詳しいため、JSON はデータの相互交換をする場合の良い選択肢です。

オブジェクト x があり、その JSON 形式の文字列表現を見るには、単純な1行のコードを書くだけです:

```
>>> import json
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

dumps() に似た関数に、dump() があり、こちらは単純にオブジェクトを text file にシリアライズします。f が書き込み用に開かれた text file だとすると、次のように書くことができます:

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

逆にデシリアライズするには、f が読み込み用に開かれた text file だとすると、次のようになります:

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

このような単純なシリアライズをする手法は、リストや辞書を扱うことはできますが、任意のクラス・インスタンスを JSON にシリアライズするにはもう少し努力しなくてはなりません。json モジュールのリファレンスにこれについての解説があります。

8. エラーと例外

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

8.1. 構文エラー

構文エラーは構文解析エラー (parsing error) としても知られており、Python を勉強している間に最もよく遭遇する問題の一つでしょう:

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

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

8.2. 例外

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

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

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

残りの行は例外の詳細で、その例外の型と何が起きたかに依存します。

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

組み込み例外 には、組み込み例外とその意味がリストされています。

8.3. 例外を処理する

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

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

`try` 文は下記のように動作します。

- まず、 *try* 節 *(try clause)* (キーワード `try` と `except` の間の文) が実行されます。
- 何も例外が発生しなければ、 *except* 節 をスキップして `try` 文の実行を終えます。
- try 節内の実行中に例外が発生すると、その節の残りは飛ばされます。次に、例外型が `except` キーワードの後に指定されている例外に一致する場合、except 節が実行された後、 `try` 文の後ろへ実行が継続されます。
- もしも except 節で指定された例外と一致しない例外が発生すると、その例外は `try` 文の外側に渡されます。例外に対するハンドラ (handler、処理部) がどこにもなければ、 処理されない例外 *(unhandled exception)* となり、上記に示したようなメッセージを出して実行を停止します。

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

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

except 節のクラスは、例外と同じクラスか基底クラスのときに互換 (compatible)となります。 (逆方向では成り立ちません — 派生クラスの例外がリストされている except 節は基底クラスの例外と互換ではありません)。例えば、次のコードは、 B, C, D を順序通りに出力します:

```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

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

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

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

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

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

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

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

例外が発生するとき、例外は関連付けられた値を持つことができます。この値は例外の 引数 *(argument)* とも呼ばれます。引数の有無および引数の型は、例外の型に依存します。

except 節では、例外名の後に変数を指定することができます。この変数は例外インスタンスに結び付けられており、 instance.args に例外インスタンス生成時の引数が入っています。例外インスタンスには __str__() が定義されており、 .args を参照しなくても引数を直接印字できるように利便性が図られています。必要なら、例外を送出する前にインスタンス化して、任意の属性を追加できます。

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception instance
...     print(inst.args)     # arguments stored in .args
...     print(inst)          # __str__ allows args to be printed directly,
...                          # but may be overridden in exception subclasses
...     x, y = inst.args     # unpack args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

例外が引数を持っていれば、それらは処理されない例外のメッセージの最後の部分 (「詳細説明」) に出力されます。

例外ハンドラは、try 節の直接内側で発生した例外を処理するだけではなく、その try 節から (たとえ間接的にでも) 呼び出された関数の内部で発生した例外も処理します。例えば:

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
...
Handling run-time error: division by zero
```

8.4. 例外を送出する

raise 文を使って、特定の例外を発生させることができます。例えば:

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

raise の唯一の引数は送出される例外を指し示します。 これは例外インスタンスか例外クラス (Exception を継承したクラス) でなければなりません。 例外クラスが渡された場合は、引数無しのコンストラクタが呼び出され、暗黙的にインスタンス化されます:

```
raise ValueError  # shorthand for 'raise ValueError()'
```

例外が発生したかどうかを判定したいだけで、その例外を処理するつもりがなければ、単純な形式の raise 文を使って例外を再送出させることができます:

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

8.5. ユーザー定義例外

プログラム上で新しい例外クラスを作成することで、独自の例外を指定することができます (Python のクラスについては クラス 参照)。例外は、典型的に Exception クラスから、直接または間接的に派生したものです。

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

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

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

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

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

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

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

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

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

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

8.6. クリーンアップ動作を定義する

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

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

finally 節 *(finally clause)* は、例外が発生したかどうかに関わらず、 try 文を抜ける前に常に実行されます。 try 節の中で例外が発生して、 except 節で処理されていない場合、または except 節か else 節で例外が発生した場合は、 finally 節を実行した後、その例外を再送出します。 finally 節はまた、 try 節から break 文や continue 文、 return 文経由で抜ける際にも、「抜ける途中で」 実行されます。より複雑な例です:

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
```

```
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

見てわかるとおり、 `finally` 節はどの場合にも実行されています。文字列を割り算することで発生した `TypeError` は `except` 節で処理されていないので、 `finally` 節実行後に再度送出されています。

実世界のアプリケーションでは、 `finally` 節は(ファイルやネットワーク接続などの)外部リソースを、利用が成功したかどうかにかかわらず解放するために便利です。

8.7. 定義済みクリーンアップ処理

オブジェクトのなかには、その利用の成否にかかわらず、不要になった際に実行される標準的なクリーンアップ処理が定義されているものがあります。以下の、ファイルをオープンして内容を画面に表示する例をみてください。

```
for line in open("myfile.txt"):
    print(line, end="")
```

このコードの問題点は、コードの実行が終わった後に不定の時間ファイルを開いたままでいることです。これは単純なスクリプトでは問題になりませんが、大きなアプリケーションでは問題になりえます。 `with` 文はファイルのようなオブジェクトが常に、即座に正しくクリーンアップされることを保証します。

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

この文が実行されたあとで、たとえ行の処理中に問題があったとしても、ファイル *f* は常に close されます。ファイルなどの、定義済みクリーンアップ処理を持つオブジェクトについては、それぞれのドキュメントで示されます。

10. 標準ライブラリミニツアー

10.1. OSへのインタフェース

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

```
>>> import os
>>> os.getcwd()      # Return the current working directory
'C:¥¥Python35'
>>> os.chdir('/server/accesslogs')   # Change current working directory
>>> os.system('mkdir today')   # Run the command mkdir in the system shell
0
```

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

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

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

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

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

10.2. ファイルのワイルドカード表記

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

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

10.3. コマンドライン引数

一般的なユーティリティスクリプトでは、よくコマンドライン引数を扱う必要があります。コマンドライン引数は sys モジュールの *argv* 属性にリストとして保存されています。例えば、以下の出力は、 python demo.py one two three とコマンドライン上で起動した時に得られるものです:

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

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

10.4. エラー出力のリダイレクトとプログラムの終了

sys モジュールには、 *stdin*, *stdout*, *stderr* を表す属性も存在します。 *stderr* は、警告やエラーメッセージを出力して、 *stdout* がリダイレクトされた場合でも読めるようにするために便利です:

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

sys.exit() は、スクリプトを終了させるもっとも直接的な方法です。

10.5. 文字列のパターンマッチング

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

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

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

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

10.6. 数学

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

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

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

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)   # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float
0.17970987693706186
>>> random.randrange(6)    # random integer chosen from range(6)
4
```

statistics モジュールは数値データの基礎的な統計的特性（平均、中央値、分散等）を計算します:

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

SciPy プロジェクト <https://scipy.org> は数値処理のための多くのモジュールを提供しています。

10.7. インターネットへのアクセス

インターネットにアクセスしたりインターネットプロトコルを処理したりするための多くのモジュールがあります。最も単純な2つのモジュールは、 URL からデータを取得するための urllib.request と、メールを送るための smtplib です:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl') as response:
...     for line in response:
...         line = line.decode('utf-8')  # Decoding the binary data to text.
...         if 'EST' in line or 'EDT' in line:  # look for Eastern Time
...             print(line)

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

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
...     """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(2つ目の例は localhost でメールサーバーが動いている必要があることに注意してください。)

10.8. 日付と時刻

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

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

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

10.9. データ圧縮

一般的なデータアーカイブと圧縮形式は、以下のようなモジュールによって直接的にサポートされます: zlib, gzip, bz2, lzma, zipfile, tarfile。

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

10.10. パフォーマンスの計測

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

例えば、引数の入れ替え操作に対して、伝統的なアプローチの代わりにタプルのパックやアンパックを使ってみたいと思うかもしれません。 timeit モジュールを使えば、パフォーマンスがほんの少し良いことがすぐに分かります:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

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

10.11. 品質管理

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

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

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

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

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

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

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

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

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

10.12. バッテリー同梱

Python には 「バッテリー同梱 (batteries included)」 哲学があります。この哲学は、洗練され、安定した機能を持つ Python の膨大なパッケージ群に如実に表れています。例えば:

- xmlrpc.client および xmlrpc.server モジュールは、遠隔手続き呼び出し (remote procedure call) を全く大したことのない作業に変えてしまいます。モジュール名とは違い、XML を扱うための直接的な知識は必要ありません。
- email パッケージは、MIME やその他の RFC 2822 に基づくメッセージ文書を含む電子メールメッセージを管理するためのライブラリです。実際にメッセージを送信したり受信したりする smtplib や poplib と違って、email パッケージには (添付文書を含む) 複雑なメッセージ構造の構築やデコードを行ったり、インターネット標準のエンコードやヘッダプロトコルの実装を行ったりするための完全なツールセットを備えています。
- json パッケージはこの一般的なデータ交換形式のパースをロバストにサポートしています。csv モジュールはデータベースや表計算で一般的にサポートされている CSV ファイルを直接読み書きするのをサポートしています。xml.etree.ElementTree、xml.dom ならびに xml.sax パッケージは XML の処理をサポートしています。総合すると、これらのモジュールによって Python アプリケーションと他のツールの間でとても簡単にデータを受け渡すことが出来ます。
- sqlite3 モジュールは SQLite データベースライブラリのラッパです。若干非標準の SQL シンタックスを用いて更新や接続出来る永続的なデータベースを提供します。
- 国際化に関する機能は、 gettext, locale, codecs パッケージといったモジュール群でサポートされています。

12. 仮想環境とパッケージ

12.1. はじめに

Python アプリケーションはよく標準ライブラリ以外のパッケージやモジュールを利用します。またアプリケーションがあるバグ修正を必要としていたり、過去のバージョンのインターフェイスに依存しているために、ライブラリの特定のバージョンを必要とすることもあります。

そのため、1つのインストールされたPythonが全てのアプリケーションの要求に対応することは不可能です。もしアプリケーションAがあるモジュールのバージョン 1.0 を要求していて、別のアプリケーションBが同じモジュールのバージョン 2.0 を要求している場合、2つの要求は衝突していて、1.0 と 2.0 のどちらかのバージョンをインストールしても片方のアプリケーションが動きません。

The solution for this problem is to create a virtual environment (often shortened to 「virtualenv」), a self-contained directory tree that contains a Python installation for a particular version of Python, plus a number of additional packages.

Different applications can then use different virtual environments. To resolve the earlier example of conflicting requirements, application A can have its own virtual environment with version 1.0 installed while application B has another virtualenv with version 2.0. If application B requires a library be upgraded to version 3.0, this will not affect application A's environment.

12.2. 仮想環境の作成

The script used to create and manage virtual environments is called **pyvenv**. **pyvenv** will usually install the most recent version of Python that you have available; the script is also installed with a version number, so if you have multiple versions of Python on your system you can select a specific Python version by running `pyvenv-3.4` or whichever version you want.

To create a virtualenv, decide upon a directory where you want to place it and run **pyvenv** with the directory path:

```
pyvenv tutorial-env
```

これは `tutorial-env` ディレクトリがなければ作成して、その中に Python インタプリタ、標準ライブラリ、その他関連するファイルを含むサブディレクトリを作ります。

Once you've created a virtual environment, you need to activate it.

Windows の場合:

```
tutorial-env/Scripts/activate
```

Unix や Mac OS の場合:

```
source tutorial-env/bin/activate
```

(このスクリプトは bash shell で書かれています。 **csh** や **fish** を利用している場合、代わりに利用できる `activate.csh` と `activate.fish` スクリプトがあります。)

Activating the virtualenv will change your shell's prompt to show what virtualenv you're using, and modify the environment so that running `python` will get you that particular version and installation of Python. For example:

```
-> source ~/envs/tutorial-env/bin/activate
(tutorial-env) -> python
Python 3.4.3+ (3.4:c7b9645a6f35+, May 22 2015, 09:31:25)
  ...
>>> import sys
>>> sys.path
['', '/usr/local/lib/python34.zip', ...,
'~/envs/tutorial-env/lib/python3.4/site-packages']
>>>
```

12.3. pip を使ったパッケージ管理

Once you've activated a virtual environment, you can install, upgrade, and remove packages using a program called **pip**. By default `pip` will install packages from the Python Package Index, <https://pypi.python.org/pypi>. You can browse the Python Package Index by going to it in your web browser, or you can use `pip`'s limited search feature:

```
(tutorial-env) -> pip search astronomy
skyfield               - Elegant astronomy for Python
gary                   - Galactic astronomy and gravitational dynamics.
novas                  - The United States Naval Observatory NOVAS astronomy library
astroobs               - Provides astronomy ephemeris to plan telescope observations
PyAstronomy            - A collection of astronomy related tools for Python.
...
```

`pip` は 「search」, 「install」, 「uninstall」, 「freeze」 など、いくつかのサブコマンドを持っています。 (`pip` の完全なドキュメントは Python モジュールのインストール ガイドを参照してください。)

You can install the latest version of a package by specifying a package's name:

```
-> pip install novas
Collecting novas
  Downloading novas-3.1.1.3.tar.gz (136kB)
Installing collected packages: novas
  Running setup.py install for novas
Successfully installed novas-3.1.1.3
```

You can also install a specific version of a package by giving the package name followed by == and the version number:

```
-> pip install requests==2.6.0
Collecting requests==2.6.0
  Using cached requests-2.6.0-py2.py3-none-any.whl
Installing collected packages: requests
Successfully installed requests-2.6.0
```

If you re-run this command, pip will notice that the requested version is already installed and do nothing. You can supply a different version number to get that version, or you can run pipinstall --upgrade to upgrade the package to the latest version:

```
-> pip install --upgrade requests
Collecting requests
Installing collected packages: requests
  Found existing installation: requests 2.6.0
    Uninstalling requests-2.6.0:
      Successfully uninstalled requests-2.6.0
Successfully installed requests-2.7.0
```

pip uninstall コマンドに削除するパッケージ名を1つ以上指定します。

pip show will display information about a particular package:

```
(tutorial-env) -> pip show requests
---
Metadata-Version: 2.0
Name: requests
Version: 2.7.0
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.com
License: Apache 2.0
Location: /Users/akuchling/envs/tutorial-env/lib/python3.4/site-packages
Requires:
```

pip list will display all of the packages installed in the virtual environment:

```
(tutorial-env) -> pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

pip freeze will produce a similar list of the installed packages, but the output uses the format that pip install expects. A common convention is to put this list in a requirements.txt file:

```
(tutorial-env) -> pip freeze > requirements.txt
(tutorial-env) -> cat requirements.txt
novas==3.1.1.3
numpy==1.9.2
requests==2.7.0
```

The requirements.txt can then be committed to version control and shipped as part of an application. Users can then install all the necessary packages with install -r:

```
-> pip install -r requirements.txt
Collecting novas==3.1.1.3 (from -r requirements.txt (line 1))
  ...
Collecting numpy==1.9.2 (from -r requirements.txt (line 2))
  ...
Collecting requests==2.7.0 (from -r requirements.txt (line 3))
  ...
Installing collected packages: novas, numpy, requests
  Running setup.py install for novas
Successfully installed novas-3.1.1.3 numpy-1.9.2 requests-2.7.0
```

pip にはたくさんのオプションがあります。 pip の完全なドキュメントは Python モジュールのインストール を参照してください。パッケージを作成してそれを Python Package Index で公開したい場合、 Python モジュールの配布 ガイドを参照してください。