Python Tutorial

リリース *3.13.0a5*

Guido van Rossum and the Python development team

4 月 09, 2024

目次

索引 199

Python は強力で、学びやすいプログラミング言語です。効率的な高レベルデータ構造と、シンプルで効果的なオブジェクト指向プログラミング機構を備えています。Python は、洗練された文法・動的なデータ型付け・インタープリタであることなどから、スクリプティングや高速アプリケーション開発 (Rapid Application Development: RAD) に理想的なプログラミング言語となっています。

Python Web サイト (https://www.python.org) は、Python インタープリタと標準ライブラリのソースコードと、主要プラットフォームごとにコンパイル済みのバイナリファイルを無料で配布しています。また、Python ウェブサイトには、無料のサードパーティモジュールやプログラム、ツール、ドキュメントなども紹介しています。

Python インタプリタは、簡単に C/C++ 言語などで実装された関数やデータ型を組み込み、拡張できます。また、アプリケーションのカスタマイズを行う、拡張言語としても適しています。

このチュートリアルは、Python 言語の基本的な概念と機能を、形式ばらずに紹介します。読むだけではなく、Python インタープリタで実際にサンプルを実行すると理解が深まりますが、サンプルはそれぞれ独立していますので、ただ読むだけでも良いでしょう。

標準オブジェクトやモジュールの詳細は、library-index を参照してください。また、正式な言語定義は、reference-index にあります。C 言語や C++ 言語で拡張モジュールを書くなら、extending-index や c-api-index を参照してください。Python の解説書も販売されています。

このチュートリアルは、Python 全体を対象とした、包括的な解説書ではありません。よく使われる機能に限っても、全ては紹介していません。その代わり、このチュートリアルでは、Python のもっとも特徴的な機能を中心に紹介して、この言語の持ち味や、スタイルを感じられるようにしています。このチュートリアルを読み終えると、Python のモジュールやプログラムを読み書きできるようになっているでしょう。また、library-index のさまざまな Python ライブラリモジュールを、詳しく調べられるようになっているはずです。

用語集 にも目を通しておくと良いでしょう。

第

ONE

やる気を高めよう

コンピュータを使っていろいろな作業をしていると、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ファイル名を整理して変更したり、などです。ちょっとした専用のデータベースや、何か専用の 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。

あなたがプロのソフト開発者として、C/C++/Java ライブラリを扱う必要があるけども、通常の編集/コンパイル/テスト/再コンパイルのサイクルを遅すぎると感じているかもしれません。上記ライブラリのためのテストを書くことにうんざりしているかもしれません。または、拡張言語を持つアプリケーションを書いているなら、そのために新しい言語一式の設計と実装をしたくないでしょう。

Python はそんなあなたのための言語です。

そういった処理は、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。しかし、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ますが、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、最初の試し書きだけでもかなりの時間がかかってしまいます。Python はもっと簡単に利用でき、Windows、macOS、そして Unix オペレーティングシステムで動作し、あなたの仕事をすばやく片付ける助けになるでしょう。

Python 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よりも多くの機構があり、大きなプログラムの開発にも適しています。一方では、Python は C よりたくさんのエラーチェックを実行時に行っており、また可変長配列や辞書などの高級な型を組込みで持つ **超高級言語** *(very-high-level language)* です。Python は Awk や Perl などよりも汎用的なデータ型を備えており、より多くの領域で利用できます。また、Python はこれらの言語と比べても、少なくとも同じぐらいには簡単です。

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。標準モジュールには、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ーフェースツールキットを使うためのインターフェイ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の必要がないので、プログラムを開発する際にかなりの時間を節約できます。インタプリタは対話的にも使えるので、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、同じ機能の 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 には **拡張性** があります: C 言語でプログラムを書く方法を知っているなら、簡単に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、いちばん時間のかかる処理を高速化したり、ベンダ特有のグラフィクスライブラリなどの、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、まずインタプリタの使い方を説明します。これはわかりきった内容かもしれませんが、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。

第

TWO

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/python3.13` 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:

```
python3.13
```

[*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 where you have installed Python from the Microsoft Store, the `python3.13` command will be available. If you have the py.exe launcher installed, you can use the `py` command. See setting-envvars for other ways to launch Python.

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。

GNU Readline ライブラリ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 Unix シェルと同じように使えます。標準入力が端末に接続された状態では、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指定するか、`python3 < filename` のように標準入力ファイルとし

[*1] Unix では、Python 3.x インタープリタの実行ファイルはデフォルトでは `python` という名前ではインストールされません。同時にインストールされた Python 2.x 実行ファイルと衝突させないためです。

て指定すると、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 `python -c command [arg] ...` のように起動する方法もあります。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 -m module [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、スクリプトの実行が完了した後、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` をスクリプト名の前に追加します。

全てのコマンドラインオプションは using-on-general 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` とすることでこのリストにアクセスできます。`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。Python インタープリタは、`-c` *command* や `-m` *module* の後ろに指定したオプションは無視します。無視された引数は、`sys.argv` を使って *command* や *module* から参照できます。

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。継続行では、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:

```
$ python3.13
Python 3.13 (default, April 4 2023, 09:25:04)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

継続行は、複数行の構文を入力するときに使います。例えば、`if` 文は継続行を使用します

```
>>> 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、識別子、コメントなどに書けます。--- 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。

デフォルトエンコーディング以外のエンコーディングを使用するには、ファイルの **先頭** の行に特別なコメントを追加しなければなりません。書式は以下の通りです:

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

encoding には、Python が `codecs` でサポートしている有効なエンコーディングを指定します。

例えば、Windows-1252 エンコーディングを使用するには、ソースコードファイルの先頭行は下記のようにします:

```
# -*- coding: cp1252 -*-
```

ソースコードが *UNIX "shebang" 行* で始まる場合には、**先頭行** のルールは当てはまりません。この場合には、エンコーディングの宣言はファイルの 2 行目に追加します。例えば以下のようになります:

```
#!/usr/bin/env python3
# -*- coding: cp1252 -*-
```

脚注

第

THREE

形式ばらない PYTHON の紹介

以下のサンプルでは、入力と出力はプロンプト (»> や ...) の有無で区別します: 例を実際に試す場合は、プロンプトが表示されているときに、サンプル中のプロンプトから後ろの内容全てを入力します。

このマニュアルにあるサンプルの多くは、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、サンプルコードを実際に入力して試して見るときは、コメント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てみましょう。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、簡単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 +、-、*、/ によって算術演算を行うことができ、丸括弧 (()) をグループ化に使うことができます。例えば:

```
>>> 2 + 2
4
```

(次のページに続く)

(前のページからの続き)

```
>>> 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 型であり、小数部を持つ数 (例えば、5.0 、1.6) は float 型です。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。// 演算子は **整数除算** を行い、整数値を返します; 剰余は、% で求めます。:

```
>>> 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  # floored quotient * 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、結果は出力されず、次の入力プロンプトが表示されます。:

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

変数が ” 定義” されていない (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):
```

(次のページに続く)

[*1] ** は - より優先順位が高いので、-3**2 は -(3**2) と解釈され、計算結果は -9 になります。これを避けて 9 を得たければ、(-3)**2 と書きます。

(前のページからの続き)

```
  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) に複数の型が入り混じっている場合、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
```

この変数には読取りだけを行い、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、同じ名前で別のローカル変数が生成され、元の特別な動作をする組み込み変数を覆い隠してしておかしなことになってしまうかもしれません。

int と float に加え、Python は Decimal や Fraction などの他の数値型もサポートしています。複素数 も組み込み型としてサポートしており、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 テキスト

Python は数値だけでなくテキスト (いわゆる「文字列」である str 型によって表現されます) を扱うことができます。! のような文字や rabbit のような単語、Paris のような名前、Got your back. ような文もすべて文字列です。文字列はシングルクォート ('...') またはダブルクォート ("...") で囲み、どちらを使っても違いはありません[*2] 。

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> "Paris rabbit got your back :)! Yay!"  # double quotes
'Paris rabbit got your back :)! Yay!'
>>> '1975'  # digits and numerals enclosed in quotes are also strings
'1975'
```

[*2] 他の言語と違って、\n のような特殊文字は、単引用符 ('...') と二重引用符 ("...") で同じ意味を持ちます。両者の唯一の違いは、単引用符で囲われた箇所では " をエスケープする必要がない (ただし \' はエスケープする必要がある) ことで、逆もまた同様です。

クォートの中でクォートを使いたい場合、\ を前に付け加えることで「エスケープ」をする必要があります。もしくは、文字列で使用したいクォートとは別の方のクォートで囲むこともできます。

```
>>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

Python シェルでは、文字列を定義するときと文字列が出力されるときでは見え方が異なることがあります。`print()` 関数を使うと、両端のクォートがなくなり、エスケープされた文字や特殊文字が表示されるため、より読みやすい形で出力できます。

```
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print(), special characters are included in the string
'First line.\nSecond line.'
>>> print(s)  # with print(), special characters are interpreted, so \n produces new line
First line.
Second line.
```

\ に続く文字を特殊文字として解釈されたくない場合は、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
```

raw 文字列には微妙な面があります: raw 文字列は奇数個の “\“ 文字では終了できません。詳細と解決方法については the FAQ entry を参照してください。

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。改行文字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'
```

この機能は、長い文字列を改行したいときにとても役に立ちます:

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

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
  File "<stdin>", line 1
    prefix 'thon'
           ^^^^^^
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
  File "<stdin>", line 1
    ('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'
```

文字列は **インデックス** (添字) を指定して文字を取得できます。最初の文字のインデックスは 0 になります。文字を表す、専用のデータ型は用意されていません; 文字とは、単に長さが 1 の文字列です:

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

(次のページに続く)

(前のページからの続き)

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

インデックスには、負の値も指定できます。この場合、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 と区別できないので、負のインデックスは -1 から始まります。

インデックス表記に加え、**スライス** もサポートされています。インデッ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'
```

スライスのインデックスには、便利なデフォルト値があります; 最初のインデックスを省略すると、0 と見なされます。二番め のインデックスを省略すると、スライス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'
```

開始値は常に含まれ、終了値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:]` は常に `s` と等しくなります:

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

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

```
 +---+---+---+---+---+---+
 | 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 までのインデッ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ッ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。

正のインデックスの場合、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ックスが範囲内にあるかぎり、インデックス間の差になります。例えば、`word[1:3]` の長さは 2 になります。

大き過ぎるインデッ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ッ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** です。従って、文字列のインデッ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'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
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:

textseq 文
: 字列は代表的な **シーケンス型** で、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。

string-methods 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。

f-strings 式
: の埋め込みをサポートした文字列リテラル

formatstrings
: `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。

old-string-formatting 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 は多くの **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、通常は同じ型の要素のみを持ちます。

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

文字列 (や他の全ての組み込みの **シーケンス** 型) のように、リストはインデックスやスライスができます:

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

リストは、リストの連結など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 な文字列とは違って、リストは **可変** 型ですので、要素を入れ替えられます:

```
>>> 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]
```

`list.append()` を使って、リストの末尾に新しい要素を追加できます (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]
```

単純な代入では Python は決してデータをコピーしません。リストを変数に代入すると、その変数は **既存のリスト** を参照します。ある変数を通してリストに任意の変更を行うと、そのリストを参照したすべての他の変数を通して確認できます。:

```
>>> rgb = ["Red", "Green", "Blue"]
>>> rgba = rgb
>>> id(rgb) == id(rgba)  # they reference the same object
True
>>> rgba.append("Alph")
>>> rgb
["Red", "Green", "Blue", "Alph"]
```

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。例えば、次のスライスは、リストの **浅いコピー** を返します。:

```
>>> correct_rgba = rgba[:]
>>> correct_rgba[-1] = "Alpha"
>>> correct_rgba
["Red", "Green", "Blue", "Alpha"]
>>> rgba
["Red", "Green", "Blue", "Alph"]
```

スライスには、代入もできます。スライスの代入で、リストのサイズを変更したり、全てを削除したりもできます:

```
>>> 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
```

(次のページに続く)

(前のページからの続き)

```
['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()` はリストにも使えます:

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

リストを入れ子 (ほかのリストを含むリストを造る) にできます。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課題にも Python を使えます。例えば、Fibonacci series の先頭の部分列は次のように書けます:

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

(次のページに続く)

(前のページからの続き)

```
3
5
8
```

上の例では、いくつか新しい機能を使用しています。

- 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、最後の行でも使われています。代入文では、まず右辺の式がすべて評価され、次に代入が行われます。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 " a < 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。条件式には、文字列値やリスト値なども使えます。それ以外のシーケンスも、条件式として使用できます。長さが 1 以上のシーケンスは真で、空のシーケンスは偽になります。サンプル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。実用的には、もっと複雑な処理を入力する場合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、入力完了のしるしとして最後に空行を入力します。これは、パーザはどれが最後の行を入力なのか、判断できないためです。基本的なブロック内では、全ての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。`print()` 関数では、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 a < 1000:
  ...     print(a, end=',')
  ...     a, b = b, a+b
  ...
  0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
  ```

脚注

第

FOUR

その他の制御フローツール

前章で紹介した `while` 文の他にも、Python にはいくつか制御フローツールがあり、本章で説明します。

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。

いくつかの定数と同じ値かを比較する場合や、特定の型や属性かを確認する場合には、`match` 文が便利です。詳細は *match* **文** を参照してください。

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
```

コレクションオブジェクトの値を反復処理をしているときに、そのコレクションオブジェクトを変更するコードは理解するのが面倒になり得ます。そうするよりも、コレクションオブジェクトのコピーに対して反復処理をするか、新しいコレクションオブジェクトを作成する方が通常は理解しやすいです:

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

# Strategy:  Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# Strategy:  Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

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
```

(次のページに続く)

(前のページからの続き)

```
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:

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(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:

```
>>> 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** と呼ばれます。これらは関数や構成物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 文がそのような構成物であることはすでに見てきており、イテラブルを受け取る関数の例には sum() があります:

```
>>> sum(range(4))  # 0 + 1 + 2 + 3
6
```

この後には、イテラブルを返したりイテラブルを引数で受け取るいくつかの関数が出てきます。データ構造 では、list() についてより詳しく説明します。

4.4 break 文と continue 文とループの else 節

break 文は、その *break* 文を内包している最も内側にある for 文または while 文から抜け出すことができます。

for 文と while 文では else 節を書くことができます。

for 文の場合、else 節はループ処理の最後の回が実行されたあとに実行されます。

while 文の場合は、ループ条件が偽となったあとに実行されます。

どちらのループ文でも、break によってループ処理が終了したときは else 節は**実行されません**。

その例として、素数を探索する for 文を以下に示します:

```
>>> 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 an odd number", num)
...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd 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 のもう 1 つの使い道は、新しいコードを書いているときの関数や条件文の仮置きの本体としてです。こうすることで、より抽象的なレベルで考え続けられます。pass は何事も無く無視されます

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

4.6 match 文

match 文は 1 つの式を指定し、その値と次に続く 1 つ以上の case ブロックに指定されたパターンを比較します。この機能は C や Java、JavaScript(や他の多数の言語) の switch 文と表面的には似ていますが、Rust や Haskell のパターンマッチングにより似ています。最初にマッチしたパターンのみが実行され、コンポーネント (シーケンスの要素やオブジェクトの属性) から値を取り出して変数に代入することもできます。

最も単純な形式は、対象の値に対して 1 つ以上のリテラルです:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

最後のブロックについて: 変数名 _ は **ワイルドカード** の働きをし、マッチに絶対失敗しません。マッチするケースがない場合は、そのブロックも実行されません。

複数のリテラルを | ("or") を使用して組み合わせて 1 つのパターンにできます:

```
case 401 | 403 | 404:
    return "Not allowed"
```

パターンはアンパック代入ができ、変数に結びつけられます:

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

このコードは注意して見てください！ 最初のパターンには 2 つのリテラルがあり、上で示したリテラルパターンの拡張と考えることができます。しかし次の 2 つのパターンはリテラルと変数の組み合わせのため、対象 (point) から値を取り出して変数に **結びつけ** ます。4 番目のパターンは 2 つの値を取り込みます。これは、アンパック代入 (x, y) = point と概念的に似ています。

データを構造化するためにクラスを使っている場合は、クラス名の後ろにコンストラクターのように引数のリストを指定できます。属性の値は変数に取り込まれます。

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Somewhere else")
        case _:
            print("Not a point")
```

いくつかの組み込みクラスでは位置引数が使用でき、属性の順番を提供します (例: データクラス)。クラスの `__match_args__` 特殊属性によって、パターンの中で属性の明確な位置を定義することもできます。("x", "y") が設定された場合、以下のすべてのパターンは等価です (すべて属性 `y` が `var` 変数に結びつけられます):

```
Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)
```

おすすめのパターンの読み方は、パターンが、代入文の左辺に配置するものを拡張した形式であるとみなすことです。これにより、どの変数になにが代入されるかが分かります。単独の名前（上記の `var` など）だけがマッチ文で値が代入されます。ドット付きの名前（`foo.bar` など)、属性名（上記の `x=`、`y=` など)、クラス名（名前の後ろの "(...)" によって判別される。上記の `Point` など）には値は代入されません。

パターンはいくらでも入れ子 (ネスト) にすることができます。例えば、`__match_args__` を追加した Point クラスのリストに対して次のようにマッチを行うことができます:

```
class Point:
    __match_args__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x
        self.y = y

match points:
    case []:
        print("No points")
```

(次のページに続く)

(前のページからの続き)

```
    case [Point(0, 0)]:
        print("The origin")
    case [Point(x, y)]:
        print(f"Single point {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two on the Y axis at {y1}, {y2}")
    case _:
        print("Something else")
```

パターンに `if` 節を追加できます。これは ” ガード” と呼ばれます。ガードが false の場合、`match` は次の case ブロックの処理に移動します。ガードを評価する前に値が取り込まれることに注意してください:

```
match point:
    case Point(x, y) if x == y:
        print(f"Y=X at {x}")
    case Point(x, y):
        print(f"Not on the diagonal")
```

この文のその他のいくつか重要な特徴:

- アンパック代入のように、タプルとリストのパターンでは正確に同じ意味で、任意のシーケンスと一致します。重要な例外として、イテレーターや文字列ではマッチしません。
- シーケンスパターンは拡張アンパックをサポート: `[x, y, *rest]` と `(x, y, *rest)` はアンパック代入として同じように動作します。`*` のあとの変数名は `_` でもよく、そのため `(x, y, *_)` は最低でも 2 つのアイテムを持つシーケンスにマッチし、残りのアイテムは変数に結びつけられません。
- マッピングパターン: `{"bandwidth": b, "latency": l}` は辞書から `"bandwidth"` と `"latency"` の値を取り込みます。シーケンスパターンとは異なり、それ以外のキーは無視されます。アンパッキングのような `**rest` もサポートされています（しかし、`**_` は冗長なため禁止されています)。
- サブパターンでは `as` キーワードを使用して値を取り込みます:

  ```
  case (Point(x1, y1), Point(x2, y2) as p2): ...
  ```

  この例では入力から 2 番目の要素を `p2` として取り込みます（入力が 2 つのポイントのシーケンスである場合）
- ほとんどのリテラルは同一性を比較しますが、シングルトンの `True`、`False`、`None` では識別値を比較します。
- パターンには名前を付けた定数が使用できます。値を取り込む変数としてと解釈することを防ぐために、ドット付きの変数名にする必要があります。

```
from enum import Enum
class Color(Enum):
    RED = 'red'
    GREEN = 'green'
    BLUE = 'blue'

color = Color(input("Enter your choice of 'red', 'blue' or 'green': "))

match color:
    case Color.RED:
        print("I see red!")
    case Color.GREEN:
        print("Grass is green")
    case Color.BLUE:
        print("I'm feeling the blues :(")
```

より詳細な説明と追加の例は **PEP 636** にチュートリアル形式で記述してあります。

4.7 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 (グローバル変数が `global` 文で指定されていたり、外側の関数の変数が `nonlocal` 文で指定されていない限り) グローバル変数や外側の関数の変数に直接値を代入できませんが、参照することはできます。

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、実引数は **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や、自身を再帰的に呼び出すときには、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、関数名は関数オブジェクトとともに現在のシンボルテーブル内に取り入れられます。インタープリタはその名前が指すオブジェクトをユーザ定義関数 (user-defined function) として認識します。他の名前も同じ関数オブジェクトを指すことができ、またその関数にアクセスするために使用することができます:

```
>>> 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
```

(次のページに続く)

[*1] 実のところ、**オブジェクトへの参照渡し** *(call by object reference)* という言ったほうがより正確です。というのは、変更可能なオブジェクトが渡されると、呼び出された側の関数がオブジェクトに行った変更 (例えばリストに挿入された要素) はすべて、関数の呼び出し側にも反映されるからです。

(前のページからの続き)

```
...     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。
- result.append(a) という文で、result オブジェクトの*メソッド* が呼び出されます。メソッドとは、オブジェクトに「属する」関数のことであり、obj.methodname と表されます。ここで、obj は何らかのオブジェクト (式の場合もあります) であり、methodname はそのオブジェクトの型で定義されたメソッド名です。色々な型がそれぞれ独自のメソッドを定義しています。異なる型が同じ名前のメソッドを持つことも可能であり、どちらの方のものであるかという曖昧さは生まれません。(*クラス*を使って独自の型やメソッドを自分で定義することもできます。参照: クラス) 例にある append() メソッドは、リストオブジェクトに対して定義されているもので、リストの末尾に新しい要素を追加します。この例では result = result + [a] と等価ですが、計算効率の上ではベターです。

4.8 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。

4.8.1 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:
        reply = input(prompt)
        if reply in {'y', 'ye', 'yes'}:
            return True
        if reply 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 yes or 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
```

4.8.2 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 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 (typesmapping 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引数の入った **タプル** を受け取る引数 (次の小節で述べます) と組み合わせられます。(*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)
    for kw in keywords:
        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.
----------------------------------------
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

なお、複数のキーワード引数を与えた場合に、それらが出力される順序は、関数呼び出しで与えられた順序と同じになります。

4.8.3 特殊なパラメータ

デフォルトでは、引数は位置またはキーワードによる明示で Python 関数に渡されます。可読性とパフォーマンスのために、その引数が位置、位置またはキーワード、キーワードのどれで渡されるかを開発者が判定するのに関数定義だけを見ればよいように、引数の渡され方を制限することには意味があります。

関数定義は次のようになります:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
        |             |                  |
        |        Positional or keyword   |
        |                                - Keyword only
         -- Positional only
```

ここで、/ と * はオプションです。使用された場合、これらの記号は、引数が関数に渡される方法、すなわち、位置専用、位置またはキーワード、キーワード専用、といった引数の種類を示します。キーワード引数は、名前付き引数とも呼ばれます。

位置またはキーワード引数

関数定義に / も * もない場合は、引数は位置またはキーワードで関数に渡されます。

位置専用引数

これをもう少し詳しく見てみると、特定の引数を **位置専用** と印を付けられます。**位置専用** の場合、引数の順序が重要であり、キーワードで引数を渡せません。位置専用引数は /（スラッシュ）の前に配置されます。/ は、位置専用引数を残りの引数から論理的に分離するために使用されます。関数定義に / がない場合、位置専用引数はありません。

/ の後の引数は、**位置またはキーワード** 、もしくは、**キーワード専用** です。

キーワード専用引数

引数をキーワード引数で渡す必要があることを示す **キーワード専用** として引数をマークするには、引数リストの最初の **キーワード専用** 引数の直前に * を配置します。

関数の例

/ および * といったマーカーに注意を払って、次の関数定義の例を見てください:

```
>>> def standard_arg(arg):
...     print(arg)
...
>>> def pos_only_arg(arg, /):
...     print(arg)
...
>>> def kwd_only_arg(*, arg):
...     print(arg)
...
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...     print(pos_only, standard, kwd_only)
```

最も馴染みのある形式の最初の関数定義 standard_arg は、呼び出し規約に制限を設けておらず、引数は位置またはキーワードで渡されます:

```
>>> standard_arg(2)
2
```

(次のページに続く)

(前のページからの続き)

```
>>> standard_arg(arg=2)
2
```

2 番目の関数の pos_only_arg は、/ が関数定義にあるので、引数は位置専用になります:

```
>>> pos_only_arg(1)
1

>>> pos_only_arg(arg=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pos_only_arg() got some positional-only arguments passed as keyword arguments: 'arg'
```

3 番目の関数 kwd_only_args は、関数定義に * があるので、引数はキーワード専用になります:

```
>>> kwd_only_arg(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: kwd_only_arg() takes 0 positional arguments but 1 was given

>>> kwd_only_arg(arg=3)
3
```

そして最後の関数は 3 つの引数の種類を一つの関数定義の中で使用しています:

```
>>> combined_example(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() takes 2 positional arguments but 3 were given

>>> combined_example(1, 2, kwd_only=3)
1 2 3

>>> combined_example(1, standard=2, kwd_only=3)
1 2 3

>>> combined_example(pos_only=1, standard=2, kwd_only=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() got some positional-only arguments passed as keyword arguments: 'pos_
↪only'
```

最後に、位置引数 name と name をキーとして持つ **kwds の間に潜在的な衝突がある関数定義を考えてみましょう。

```
def foo(name, **kwds):
    return 'name' in kwds
```

キーワードに 'name' を入れても、先頭の引数と同じになってしまうため、この関数が True を返すような呼び出しの方法はありません。例えば、次のようになってしまいます:

```
>>> foo(1, **{'name': 2})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() got multiple values for argument 'name'
>>>
```

しかし位置専用を示す / を使用すれば可能になります。name は位置引数として、そして 'name' はキーワード引数のキーワードとして認識されるからです:

```
>>> def foo(name, /, **kwds):
...     return 'name' in kwds
...
>>> foo(1, **{'name': 2})
True
```

言い換えると、位置専用引数であれば、その名前を **kwds の中で使用しても、曖昧にならないということです。

要約

使用例で、関数定義でどの種類の引数を使うかべきかがわかると思います:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
```

ガイドとしては、

- もし引数の名前をユーザーに知らせる必要がないなら、位置専用引数を使用しましょう。これは引数の名前がユーザーにとって意味がなく、関数が呼ばれたときの引数の順序が問題であり、または、位置引数と任意のキーワードを使用する必要がある場合に便利です。
- 引数の名前に意味があり、それにより関数の定義がより明らかになる、または、ユーザーが引数の順番に縛られることを避けたほうがいいと考えるのなら、キーワード専用引数を使用しましょう。
- API の場合、将来引数の名前が変更された場合に API の変更ができなくなることを防ぐために、位置専用引数を使用しましょう。

4.8.4 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、位置引数ではなくキーワード引数としてのみ使えることを意味します。

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

4.8.5 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、個別な位置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' demised !
```

4.8.6 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')]
```

4.8.7 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.
```

4.8.8 関数のアノテーション

関数アノテーション 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 3107** と **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影響しません。パラメータアノテーションは、パラメータ名のあとのコロンと式で定義され、その式の評価結果がアノテーションとなります。戻り値アノテーションは、パラメータリストと `def` 文の終わりを表すコロンの間で、`->` の後ろに式を書くことで定義されます。次の例では、必須引数と任意引数、戻り値にアノテーションが付与されています:

```
>>> 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'
```

4.9 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 `UpperCamelCase` をクラス名に使い、`lowercase_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。

脚注

第

FIVE

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 x と等しい値を持つ最初の要素を削除します。該当する要素がなければ `ValueError` が送出されます。

`list.pop(`$\left[i\right]$`)`

指定された位置の要素をリストから取り除き、それを返します。インデックスが指定されていない場合、`a.pop()` はリスト末尾の要素を取り除いて返します。リストが空であるか、インデックスがリストの範囲外の場合は、`IndexError` を送出します。

`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 *x* と等しい値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 が送出されます。

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 at 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

[*1] 他の言語では変更可能なオブジェクトを返して、d->insert("a")->remove("b")->sort(); のようなメソッドチェインを許してい

設計上の原則となっています。

気がつくかもしれないもう一つのことは、すべてのデータをソートまたは比較できるわけではないということです。例えば、整数は文字列と比較できず、*None* は他の型と比較できないため、`[None, 'hello', 10]` はソートされません。また、定義された順序関係を持たないタイプもあります。たとえば、`3+4j < 5+7j` は有効な比較ではありません。

5.1.1 リストをスタックとして使う

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

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

5.1.2 リストをキューとして使う

リストをキュー (queue) として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:

る場合もあります。

```
>>> 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
    [x, x**2 for x in range(6)]
     ^^^^^^^
SyntaxError: did you forget parentheses around the comprehension target?
>>> # 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、内側のリスト内包表記は、続く `for` のコンテキストの中で評価されます。そのため、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 (typesseq 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
```

(次のページに続く)

(前のページからの続き)

```
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** 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 **可変** 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 a or b or both
{'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)* (typesmapping 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ックス化されているシーケンスと異なり、辞書は **キー** *(key)* でインデッ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ッ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 **キー** *(key)*: **値** *(value)* のペアの集合であり、キーが (辞書の中で) 一意でなければならない、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対し `list(d)` 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キーが挿入された順番で返します (ソートされたリストが欲しい場合は、代わりに `sorted(d)` を使ってください)。ある単一のキーが辞書にあるかどうか調べるには、`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
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['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, 'guido': 4127, 'jack': 4098}
```

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, 'guido': 4127, 'jack': 4098}
```

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.
```

(次のページに続く)

(前のページからの続き)

```
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 i in sorted(basket):
...     print(i)
...
apple
apple
banana
orange
orange
pear
```

シーケンスに set() を使うと、重複要素が除去されます。シーケンスに set() を使った上で、sorted() を使うという組み合わせ方は、順番が整列されているシーケンスで、同一要素に 1 度のみループでアクセスする慣用的な方法です。

```
>>> 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、ある値があるコンテナーの中に存在するか (または存在しないか) どうかを確認するメンバーシップテストです。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。全ての比較演算子は同じ優先順位を持っており、ともに数値演算子よりも低い優先順位となります。

比較は連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 とは異なり、式の中での代入は セイウチ演算子 := を使用して明示的に行う必要があることに注意してください。これにより、== が意図されていたところに = を入力してしまうという、C プログラムで発生する一般的なクラスの問題を回避できます。

5.8 シーケンスとその他の型の比較

概して、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、個々の文字ごとに ASCII 順序を用います。以下に、同じ型のオブジェクトを持つ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。

脚注

第

SIX

モジュール

Python インタプリタを終了させ、再び起動すると、これまでに行ってきた定義 (関数や変数) は失われています。ですから、より長いプログラムを書きたいなら、テキストエディタを使ってインタプリタへの入力を用意しておき、手作業の代わりにファイルを入力に使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 a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        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 *namespace* (詳細は *Python* **のスコープと名前空間**を参照してください) に追加することはありません。単にモジュール名 fibo だけを名前空間に追加します。関数にはモジュール名を使ってアクセスします:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 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)
0 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名前空間を持っていて、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名前空間として使います。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。

モジュールは他のモジュールをインポートできます。import 文はモジュール（さらに言えばスクリプトでも）の先頭に置きますが、これは慣習であって必須ではありません。インポートされたモジュール名は、モジュールのトップレベル（関数やクラスの外）に書いてあれば、モジュールのグローバルな名前空間に置かれます。

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名前空間内に直接取り込むという変型があります。例えば:

[*1] 実際には、関数定義も ' 実行' される ' 文' です。モジュールレベルの関数定義を実行すると、関数名はモジュールのグローバルな名前空間に追加されます。

```
>>> from fibo import fib, fib2
>>> fib(500)
0 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名前空間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)
0 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。

モジュール名の後に as が続いていた場合は、as の後ろの名前を直接、インポートされたモジュールが束縛します。

```
>>> import fibo as fib
>>> fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

これは実質的には import fibo と同じ方法でモジュールをインポートしていて、唯一の違いはインポートしたモジュールが fib という名前で取り扱えるようになっていることです。

このインポート方法は from が付いていても使え、同じ効果が得られます:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

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
0 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、インタープリタはまずその名前の組み込みモジュールを探します。モジュール名の一覧は sys.builtin_module_names にあります。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)。
- インストール方法に依存したデフォルト（慣例として site-packages ディレクトリーが含まれ、site モジュールによって処理される)。

詳細は sys-path-init を参照してください。

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。
- この処理に関する詳細は、判定のフローチャートを含めて、**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']
```

(次のページに続く)

(前のページからの続き)

```
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__',
 '__interactivehook__', '__loader__', '__name__', '__package__', '__spec__',
 '__stderr__', '__stdin__', '__stdout__', '__unraisablehook__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_framework',
 '_getframe', '_git', '_home', '_xoptions', 'abiflags', 'addaudithook',
 'api_version', 'argv', 'audit', 'base_exec_prefix', 'base_prefix',
 'breakpointhook', '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', 'get_asyncgen_hooks', 'get_coroutine_origin_tracking_depth',
 'getallocatedblocks', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencodeerrors', 'getfilesystemencoding', 'getprofile',
 'getrecursionlimit', 'getrefcount', 'getsizeof', 'getswitchinterval',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'is_finalizing', 'last_traceback', 'last_type', 'last_value',
 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks',
 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2', 'pycache_prefix',
 'set_asyncgen_hooks', 'set_coroutine_origin_tracking_depth', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace', 'stderr',
 'stdin', 'stdout', 'thread_info', 'unraisablehook', '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',
```

(次のページに続く)

(前のページからの続き)

```
'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 Pillow 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
```

(次のページに続く)

(前のページからの続き)

```
              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。

ファイルを含むディレクトリをパッケージとして Python に扱わせるには、ファイル `__init__.py` が必要です (より高度な機能 *namespace package* を使う場合を除く)。これにより、`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、パッケージ名なしで利用可能にします。なのでこう使えます:

```
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
```

また、これもサブモジュール echo をロードしますが、echofilter() 関数を直接利用可能にします:

```
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"]
```

これが意味するのは、from sound.effects import * とすると、名前の挙がった 3 つのサブモジュールが sound.effects パッケージから import される、ということです。

注意点として、サブモジュールはローカルな名前で隠されることがあります。例として、reverse 関数を sound/effects/__init__.py に足すと、from sound.effects import * では echo と surround の 2 サブモジュールだけ import されます。reverse サブモジュールは import **されません** 。それはローカルに定義された reverse 関数で隠されたので:

```
__all__ = [
    "echo",      # refers to the 'echo.py' file
    "surround",  # refers to the 'surround.py' file
    "reverse",   # !!! refers to the 'reverse' function now !!!
]

def reverse(msg: str):  # <-- this name shadows the 'reverse.py' submodule
    return msg[::-1]    #     in the case of a 'from sound.effects import *'
```

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` モジュールが必要な場合、`from sound.effects import echo` が使えます。

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。

脚注

第

SEVEN

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。)

単に空白区切りで値を並べただけの出力よりも、フォーマットを制御したいと思うことはよくあることでしょう。出力をフォーマットする方法はいくつかあります。

- **フォーマット済み文字列リテラル** を使うには、開き引用符や三重の開き引用符の前に `f` あるいは `F` を付けて文字列を始めます。この文字列の内側では、文字 `{` と文字 `}` の間に Python の式が書け、その式から変数やリテラル値が参照できます。

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

- 文字列の `str.format()` メソッドは、もう少し手間がかかります。ここでも `{` と `}` を使って変数に代入する場所の印を付けて、細かいフォーマットの指示を出せますが、フォーマットされる対象の情報を与える必要があります。

```
>>> yes_votes = 42_572_654
>>> no_votes = 43_132_495
>>> percentage = yes_votes / (yes_votes + no_votes)
>>> '{:-9} YES votes  {:2.2%}'.format(yes_votes, percentage)
' 42572654 YES votes  49.67%'
```

- 最後に、文字列のスライス操作や結合操作を使い、全ての文字列を自分で処理し、思い通りのレイアウトを作成できます。文字列型には、文字列の間隔を調整して指定されたカラム幅に揃えるのに便利な操作を行うメソッドがいくつかあります。

凝った出力である必要は無いけれど、デバッグ目的で変数をすばやく表示したいときは、repr() 関数か str() 関数でどんな値も文字列に変換できます。

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。

いくつかの例です:

```
>>> 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'))"
```

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。$x のようなプレースホルダーを使い、その箇所と辞書にある値を置き換えますが、使えるフォーマット方式はとても少ないです。

7.1.1 フォーマット済み文字列リテラル

フォーマット済み文字リテラル (短くして f-string とも呼びます) では、文字列の頭に `f` か `F` を付け、式を `{expression}` と書くことで、Python の式の値を文字列の中に入れ込めます。

オプションのフォーマット指定子を式の後ろに付けられます。このフォーマット指定子によって値のフォーマット方式を制御できます。次の例では、円周率π を小数点以下 3 桁に丸めてフォーマットしています:

```
>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}.')
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小の文字幅を指定できます。この機能は縦を揃えるのに便利です。

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

他の修飾子は、フォーマットする前に値を変換するのに使えます。`'!a'` は `ascii()` を、`'!s'` は `str()` を、`'!r'` は `repr()` を適用します:

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

`=` 指定子を使用すると式を展開して、式、イコール記、式を評価した文字列表現、の形式で表示されます。

```
>>> bugs = 'roaches'
>>> count = 13
>>> area = 'living room'
>>> print(f'Debugging {bugs=} {count=} {area=}')
Debugging bugs='roaches' count=13 area='living room'
```

`=` 指定子の詳細については self-documenting expressions を参照してください。フォーマットについての仕様は formatspec のリファレンスガイドを参照してください。

7.1.2 文字列の format() メソッド

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.
```

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
```

辞書 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。

例として、下のコード行は与えられた整数とその 2 乗と 3 乗がきちんと揃った列を生成します:

```
>>> 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
```

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

7.1.3 文字列の手作業でのフォーマット

次は 2 乗と 3 乗の値からなる同じ表を手作業でフォーマットしたものです:

```
>>> 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
```

(各カラムの間のスペース一個分は print() の動作で追加されていることに注意してください。print() は常に引数間に空白を追加します。)

文字列オブジェクトの str.rjust() メソッドは、指定された幅のフィールド内に文字列が右寄せで入るように左側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、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'
```

7.1.4 古い文字列書式設定方法

% 演算子 (剰余) は文字列のフォーマットでも使えます。'string' % values という文字列が与えられた場合、string の中の % 部分はゼロあるいは values の余りの要素に置換えられます。この操作は文字列補間として知られています。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 old-string-formatting 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, encoding=None) のように 2 つの位置引数と 1 つのキーワード引数を伴って呼び出されます。

```
>>> f = open('workfile', 'w', encoding="utf-8")
```

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)。UTF-8 は現在の事実上の標準のため、異なるエンコーディングを指定したい場合以外は encoding="utf-8" の指定がおすすめです。モードに 'b' をつけるとファイルを *binary mode* で開きます。バイナリーモードでのデータは bytes オブジェクトで読み書きします。ファイルをバイナリーモードで開くときは **エンコーディング** は指定できません。

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。

ファイルオブジェクトを扱うときに with キーワードを使うのは良い習慣です。その利点は、処理中に例外が発生しても必ず最後にファイルをちゃんと閉じることです。with を使うと、同じことを try-finally ブロックを使って書くよりずっと簡潔に書けます:

```
>>> with open('workfile', encoding="utf-8") as f:
...     read_data = f.read()

>>> # We can check that the file has been automatically closed.
>>> f.closed
True
```

もし with キーワードを使用しない場合は、ファイルを閉じ、このファイルのために利用されたシステムのリソースを直ちに解放するために f.close() を呼び出してください。

警告: f.write() を with キーワードや f.close() を使わずに呼び出した場合、プログラムが正常に終了した場合でも、f.write() の実引数がディスクに完全に **書き込まれないことがあります** 。

with 文や f.close() の呼び出しによって閉じられた後にファイルオブジェクトを使おうとするとそこで処理が失敗します。:

```
>>> 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 (テキストモードの場合) *size* 文字、(バイナリモードの場合) *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, whence)` 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 *whence* 引数で選びます。*whence* 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。*whence* 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読み書きできます。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
>>> x = [1, 'simple', 'list']
>>> json.dumps(x)
'[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 *binary file* か *text file* だとすると、次のようになります:

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

注釈: JSON ファイルは必ず UTF-8 でエンコードします。JSON ファイルを *text file* として読み込み、書き込みで開くときには、encoding="utf-8" を指定します。

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。

参考:

pickle - pickle モジュール

JSON とは対照的に、*pickle* は任意の複雑な Python オブジェクトをシリアライズ可能なプロトコルです。しかし、Python に特有のプロトコルで、他の言語で記述されたアプリケーションと通信するのには使えません。さらに、デフォルトでは安全でなく、信頼できない送信元から送られてきた、スキルのある攻撃者によって生成された pickle データをデシリアライズすると、攻撃者により任意のコードが実行されてしまいます。

第

EIGHT

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トークンを示します。エラーは指し示された **前の** トークンが存在しないために発生する可能性があります。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):
```

(次のページに続く)

(前のページからの続き)

```
  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 only concatenate str (not "int") to str
```

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。

bltin-exceptions 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/except ブロック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、エラーメッセージを出して実行を停止します。

一つの `try` 文には複数の *except* **節** が付けられ、別々の例外に対するハンドラを指定できます。多くとも一つのハンドラしか実行されません。ハンドラは対応する *try* **節** 内で発生した例外だけを処理し、同じ `try` 節内の別の例外ハンドラで起きた例外は処理しません。*except* **節** では丸括弧で囲ったタプルという形で複数の例外を指定できます。例えば次のようにします:

```
... 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。

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** *(arguments)* 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 節は例外名の後に変数を指定できます。その変数は例外インスタンスに紐付けられ、一般的には引数を保持する args 属性を持ちます。利便性のため、組み込み例外型には __str__() が定義されており、明示的に .args を参照せずとも すべての引数を表示できます。

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception type
...     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
```

例外の __str__() 出力は、処理されない例外のメッセージ末尾 (' 詳細') として表示されます。

BaseException はすべての例外に共通する基底クラスです。そのサブクラスの一つである Exception は、致命的でない例外すべての基底クラスです。Exception のサブクラスではない例外は、一般的にプログラムが終了することを示すために使われているため処理されません。例えば、sys.exit() によって送出される SystemExit や、ユーザーがプログラムを中断させたいときに送出される KeyboardInterrupt があります。

Exception はほぼすべての例外を捕捉するワイルドカードとして使えます。しかし良い例外処理の手法とは、処理対象の例外の型をできる限り詳細に書き、予期しない例外はそのまま伝わるようにすることです。

Exception に対する最も一般的な例外処理のパターンでは、例外を表示あるいはログ出力してから再度送出します (呼び出し側でも例外を処理できるようにします):

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error:", err)
except ValueError:
    print("Could not convert data to an integer.")
except Exception as err:
    print(f"Unexpected {err=}, {type(err)=}")
    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 OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

追加のコードを付け加えるのは try 節よりも else 節の方がよいでしょう。なぜなら、そうすることで try ... except 文で保護されたコードから送出されたもの以外の例外を過って捕捉してしまうという事態を避けられるからです。

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 (BaseException を継承したクラス、たとえば Exception やそのサブクラス) 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 例外の連鎖

except 節の中で未処理の例外が発生した場合、その未処理の例外は処理された例外のエラーメッセージに含まれます:

```
>>> try:
...     open("database.sqlite")
... except OSError:
...     raise RuntimeError("unable to handle error")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'database.sqlite'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: unable to handle error
```

ある例外が他の例外から直接影響されていることを示すために、raise 文にオプションの from 句を指定します:

```
# exc must be exception instance or None.
raise RuntimeError from exc
```

これは例外を変換するときに便利です。例えば:

```
>>> def func():
...     raise ConnectionError
...
>>> try:
...     func()
... except ConnectionError as exc:
...     raise RuntimeError('Failed to open database') from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in func
ConnectionError

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Failed to open database
```

また、自動的な例外の連鎖を無効にするには `from None` を指定します。

```
>>> try:
...     open('database.sqlite')
... except OSError:
...     raise RuntimeError from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError
```

例外の連鎖の仕組みに関して、詳しくは bltin-exceptions を参照してください。

8.6 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。

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。

8.7 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!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
```

もし finally 節がある場合、try 文が終わる前の最後の処理を、finally 節が実行します。try 文が例外を発生させるか否かに関わらず、finally 節は実行されます。以下では、例外が発生するという更に複雑なケースを議論します:

- もし try 文の実行中に例外が発生したら、その例外は except 節によって処理されるでしょう。もしその例外が except 節によって処理されなければ、finally 節が実行された後に、その例外が再送出されます。
- except 節または else 節の実行中に例外が発生することがあり得ます。その場合も、finally 節が実行された後に例外が再送出されます。
- finally 節で break、continue または return 文が実行された場合、例外は再送出されません。
- もし try 文が break 文、continue 文または return 文のいずれかに達すると、その break 文、continue 文または return 文の実行の直前に finally 節が実行されます。
- もし finally 節が return 文を含む場合、返される値は try 節の return 文ではなく、finally 節の return 文によるものになります。

例えば:

```
>>> def bool_return():
...     try:
...         return True
...     finally:
...         return False
...
>>> bool_return()
False
```

より複雑な例:

```
>>> 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。文字列で割り算を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。

8.8 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。

8.9 複数の関連しない例外の送出と処理

いくつか発生した例外の報告が必要な状況があります。並列処理のフレームワークでは、複数のタスクが平行して失敗することがたびたびあります。他にも最初の例外を送出するよりも、処理を継続して複数の例外を集約した方が望ましいユースケースもあります。

組み込みの ExceptionGroup は例外インスタンスのリストをまとめ、同時に送出できるようにします。ExceptionGroup も例外なので、他の例外と同じように捕捉できます。:

```
>>> def f():
...     excs = [OSError('error 1'), SystemError('error 2')]
...     raise ExceptionGroup('there were problems', excs)
...
>>> f()
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 1, in <module>
  |   File "<stdin>", line 3, in f
  | ExceptionGroup: there were problems
  +-+---------------- 1 ----------------
    | OSError: error 1
    +---------------- 2 ----------------
    | SystemError: error 2
    +------------------------------------
>>> try:
...     f()
... except Exception as e:
...     print(f'caught {type(e)}: e')
...
caught <class 'ExceptionGroup'>: e
>>>
```

except の代わりに except* を使用すると、グループの中にある特定の型に一致した例外だけを選択して処理できます。以下の例では、ネストした例外グループに対して各 except* 節で特定の型の例外を取り出し、それ以外の例外は他の節に伝えられ、最終的に例外が送出されます。:

```
>>> def f():
...     raise ExceptionGroup(
...         "group1",
...         [
...             OSError(1),
...             SystemError(2),
```

(次のページに続く)

(前のページからの続き)

```
...             ExceptionGroup(
...                 "group2",
...                 [
...                     OSError(3),
...                     RecursionError(4)
...                 ]
...             )
...         ]
...     )
...
>>> try:
...     f()
... except* OSError as e:
...     print("There were OSErrors")
... except* SystemError as e:
...     print("There were SystemErrors")
...
There were OSErrors
There were SystemErrors
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  |   File "<stdin>", line 2, in f
  | ExceptionGroup: group1
  +-+---------------- 1 ----------------
    | ExceptionGroup: group2
    +-+---------------- 1 ----------------
      | RecursionError: 4
      +------------------------------------
>>>
```

例外グループの中に含める例外は、型ではなくインスタンスである必要があることに注意してください。これは一般的に、以下のパターンようにプログラムで送出された複数の例外を捕捉することが多いためです。:

```
>>> excs = []
... for test in tests:
...     try:
...         test.run()
...     except Exception as e:
...         excs.append(e)
...
>>> if excs:
...    raise ExceptionGroup("Test Failures", excs)
...
```

8.10 ノートによって例外を充実させる

例外を送出するために生成するときに、通常は発生したエラーを説明する情報で初期化されます。例外を受け取ったあとに情報を追加すると便利な場合があります。この目的のために、例外は add_note(note) メソッドを持ちます。このメソッドは文字列を受け取り、例外のノートのリストに追加します。標準のトレースバックでは例外の後に、全てのノートが追加した順番に出力されます。:

```
>>> try:
...     raise TypeError('bad type')
... except Exception as e:
...     e.add_note('Add some information')
...     e.add_note('Add some more information')
...     raise
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: bad type
Add some information
Add some more information
>>>
```

たとえば、複数の例外を 1 つの例外グループにまとめるときに、各エラーのコンテキスト情報を追加したい場合があります。以下のグループ内の各例外が持つノートは、エラーがいつ発生したかを示しています。:

```
>>> def f():
...     raise OSError('operation failed')
...
>>> excs = []
>>> for i in range(3):
...     try:
...         f()
...     except Exception as e:
...         e.add_note(f'Happened in Iteration {i+1}')
...         excs.append(e)
...
>>> raise ExceptionGroup('We have some problems', excs)
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 1, in <module>
  | ExceptionGroup: We have some problems (3 sub-exceptions)
  +-+---------------- 1 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
    |   File "<stdin>", line 2, in f
    | OSError: operation failed
    | Happened in Iteration 1
    +---------------- 2 ----------------
    | Traceback (most recent call last):
```

(次のページに続く)

(前のページからの続き)

```
    |   File "<stdin>", line 3, in <module>
    |   File "<stdin>", line 2, in f
    | OSError: operation failed
    | Happened in Iteration 2
    +---------------- 3 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
    |   File "<stdin>", line 2, in f
    | OSError: operation failed
    | Happened in Iteration 3
    +------------------------------------
>>>
```

第

NINE

クラス

クラスはデータと機能を組み合わせる方法を提供します。新規にクラスを作成することで、新しいオブジェクトの **型** を作成し、その型を持つ新しい **インスタンス** が作れます。クラスのそれぞれのインスタンスは自身の状態を保持する属性を持てます。クラスのインスタンスは、その状態を変更するための (そのクラスが定義する) メソッドも持てます。

Python は、他のプログラミング言語と比較して、最小限の構文と意味付けを使ってクラスを言語に追加しています。Python のクラスは、C++ と Modula-3 のクラスメカニズムを混ぜたものです。Python のクラス機構はオブジェクト指向プログラミングの標準的な機能を全て提供しています。クラスの継承メカニズムは、複数の基底クラスを持つことができ、派生クラスで基底クラスの任意のメソッドをオーバライドすることができます。メソッドでは、基底クラスのメソッドを同じ名前で呼び出すことができます。オブジェクトには任意の種類と数のデータを格納することができます。モジュールと同じく、クラス機構も Python の動的な性質に従うように設計されています。クラスは実行時に生成され、生成後に変更することができます。

C++ の用語で言えば、通常のクラスメンバ (データメンバも含む) は (**プライベート変数** に書かれている例外を除いて) *public* であり、メンバ関数はすべて **仮想関数** *(virtual)* です。Modula-3 にあるような、オブジェクトのメンバをメソッドから参照するための短縮した記法は使えません: メソッド関数の宣言では、オブジェクト自体を表す第一引数を明示しなければなりません。第一引数のオブジェクトはメソッド呼び出しの際に暗黙の引数として渡されます。Smalltalk に似て、クラスはそれ自体がオブジェクトです。そのため、import や名前変更といった操作が可能です。C++ や Modula-3 と違って、ユーザーは組込み型を基底クラスにして拡張を行えます。また、C++ とは同じで Modula-3 とは違う点として、特別な構文を伴うほとんどの組み込み演算子 (算術演算子 (arithmetic operator) や添字表記) はクラスインスタンスで使うために再定義できます。

(クラスに関して普遍的な用語定義がないので、Smalltalk と C++ の用語を場合に応じて使っていくことにします。C++ よりも Modula-3 の方がオブジェクト指向の意味論が Python に近いので、Modula-3 の用語を使いたいのですが、ほとんどの読者は Modula-3 についてしらないでしょうから。)

9.1 名前とオブジェクトについて

オブジェクトには個体性があり、同一のオブジェクトに (複数のスコープから) 複数の名前を割り当てることができます。この機能は他の言語では別名づけ (alias) として知られています。Python を一見しただけでは、別名づけの重要性は分からないことが多く、変更不能な基本型 (数値、文字列、タプル) を扱うときには無視して差し支えありません。しかしながら、別名付けは、リストや辞書や他の多くの型など、変更可能な型を扱う Python コード上で驚くべき効果があります。別名付けはいくつかの点でポインタのように振舞い、このことは通常はプログラムに利するように使われます。例えば、オブジェクトの受け渡しは、実装上はポインタが渡されるだけなのでコストの低い操作になります。また、関数があるオブジェクトを引数として渡されたとき、関数の呼び出し側からオブジェクトに対する変更を見ることができます --- これにより、Pascal にあるような二つの引数渡し機構をもつ必要をなくしています。

9.2 Python のスコープと名前空間

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

まず定義から始めましょう。

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

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

属性は読取り専用にも、書込み可能にもできます。書込み可能であれば、属性に代入することができます。モジュール属性は書込み可能で、`modname.the_answer = 42` と書くことができます。書込み可能な属性は、`del`

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

文で削除することもできます。例えば、`del modname.the_answer` は、`modname` で指定されたオブジェクトから属性 `the_answer` を除去します。

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

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

スコープ *(scope)* とは、ある名前空間が直接アクセスできるような、Python プログラムのテキスト上の領域です。" 直接アクセス可能" とは、修飾なしに (訳注: `spam.egg` ではなく単に `egg` のように) 名前を参照した際に、その名前空間から名前を見つけようと試みることを意味します。

スコープは静的に決定されますが、動的に使用されます。実行中はいつでも、直接名前空間にアクセス可能な、3 つまたは 4 つの入れ子になったスコープがあります:

- 最初に探される、最も内側のスコープは、ローカルな名前を持っています。
- 外側の (enclosing) 関数のスコープは、近いほうから順に探され、ローカルでもグローバルでもない名前を持っています。
- 次のスコープは、現在のモジュールのグローバルな名前を持っています。
- 一番外側の (最後に検索される) スコープはビルトイン名を持っています。

名前が global と宣言されている場合、その名前に対する参照や代入は全て、モジュールのグローバルな名前の入った最後から 2 番目のスコープに対して直接行われます。最内スコープの外側にある変数に再束縛するには、`nonlocal` 文が使えます。nonlocal と宣言されなかった変数は、全て読み出し専用となります (そのような変数に対する書き込みは、単に **新しい** ローカル変数をもっとも内側のスコープで作成し、外部のスコープの値は変化しません)。

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

スコープはテキスト上で決定されていると理解することが重要です。モジュール内で定義される関数のグローバルなスコープは、関数がどこから呼び出されても、どんな別名をつけて呼び出されても、そのモジュールの名前空間になります。反対に、実際の名前の検索は実行時に動的に行われます --- とはいえ、言語の定義は、" コンパイ

ル” 時の静的な名前解決の方向に進化しているので、動的な名前解決に頼ってはいけません！ (事実、ローカルな変数は既に静的に決定されています。)

Python の特徴として、global や nonlocal 文が有効でない場合は、名前に対する参照は常に最も内側のスコープに対して有効になります。代入はデータをコピーしません。オブジェクトを名前に束縛するだけです。削除も同様で、del x は、ローカルスコープの名前空間から x に対する拘束を取り除きます。つまるところ、新しい名前を与えるようなすべての操作は、ローカルスコープを使って行われます。import 文、関数の定義は、モジュールや関数名をローカルスコープの名前に拘束します。

global 文を使うと、特定の変数がグローバルスコープに存在し、そこで再束縛されることを指示できます。nonlocal 文は、特定の変数が外側のスコープに存在し、そこで再束縛されることを指示します。

9.2.1 スコープと名前空間の例

異なるスコープと名前空間がどのように参照されるか、また global および nonlocal が変数の束縛にどう影響するか、この例で実演します:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

このコード例の出力は:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

このとおり、(デフォルトの) **ローカルな** 代入は *scope_test* 上の *spam* への束縛を変更しませんでした。nonlocal 代入は *scope_test* 上の *spam* への束縛を変更し、global 代入はモジュールレベルの束縛を変更しました。

またここから、global 代入の前には *spam* に何も束縛されていなかったことも分かります。

9.3 クラス初見

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

9.3.1 クラス定義の構文

クラス定義の最も単純な形式は、次のようになります:

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

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

実際には、クラス定義の内側にある文は、通常は関数定義になりますが、他の文を書くこともでき、それが役に立つこともあります --- これについては後で述べます。クラス内の関数定義は通常、メソッドの呼び出し規約で決められた独特の形式の引数リストを持ちます --- これについても後で述べます。

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

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

9.3.2 クラスオブジェクト

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

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

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

`MyClass.i` と `MyClass.f` は有効な属性参照であり、それぞれ整数と関数オブジェクトを返します。クラス属性も代入できるため、代入により `MyClass.i` の値を変えられます。`__doc__` も有効な属性で、そのクラスに属しているドキュメンテーション文字列 (docstring) 、この場合は `"A simple example class"` を返します。

クラスの **インスタンス化** *(instantiation)* には関数記法を使います。クラスオブジェクトのことを、クラスの新しいインスタンスを返す、引数のない関数のように扱ってください。上記クラスで例示すると:

```
x = MyClass()
```

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

インスタンス化操作 (クラスオブジェクトの ” 呼出し”) では、空のオブジェクトが作られます。多くのクラスでは、特定の初期状態にカスタマイズされたオブジェクトを作りたいです。そのために、クラスには `__init__()` という名前の特殊メソッドを定義できます。例えば次のようにします:

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

クラスが `__init__()` メソッドを定義している場合、クラスをインスタンス化すると、新しく作られたクラスインスタンスに対して自動的に `__init__()` を呼び出します。従って この例では、新たな初期済みインスタンスを次のようにして得られます:

```
x = MyClass()
```

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

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
```

(次のページに続く)

(前のページからの続き)

```
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3 インスタンスオブジェクト

ところで、インスタンスオブジェクトを使うと何ができるのでしょうか？ インスタンスオブジェクトが理解できる唯一の操作は、属性の参照です。有効な属性名には (データ属性およびメソッドの) 二種類あります。

データ属性 *(data attribute)* は、Smalltalk の ” インスタンス変数” や C++ の ” データメンバ” に相当します。データ属性を宣言する必要はありません。ローカル変数と同様に、これらの属性は最初に代入された時点で湧き出てきます。例えば、上で生成した `MyClass` のインスタンス `x` に対して、次のコードを実行すると、跡を残さず値 16 を印字します:

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

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

インスタンスオブジェクトで有効なメソッド名は、そのクラスによります。定義により、クラスの全ての関数オブジェクトである属性がインスタンスオブジェクトの妥当なメソッド名に決まります。従って、例では、`MyClass.f` は関数なので、`x.f` はメソッドの参照として有効です。しかし、`MyClass.i` は関数ではないので、`x.i` はメソッドの参照として有効ではありません。`x.f` は `MyClass.f` と同じものではありません --- 関数オブジェクトではなく、**メソッドオブジェクト** *(method object)* です。

9.3.4 メソッドオブジェクト

普通、メソッドはバインドされた直後に呼び出されます:

```
x.f()
```

この MyClass の例では、文字列 'hello world' が返されます。しかし、必ずしもすぐメソッドを呼び出さなければならないわけではありません。x.f はメソッドオブジェクトであり、どこかに入れておいて後で呼び出すことができます。例えば次のコードは:

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

hello world を時が終わるまで印字し続けるでしょう。

メソッドが呼び出されるときには実際には何が起きているのでしょうか？ f() の関数定義では引数を一つ指定していたにもかかわらず、上の例では x.f() が引数なしで呼び出されています。引数はどうなったのでしょうか？たしか、引数が必要な関数を引数無しで呼び出すと、Python が例外を送出するはずです --- たとえその引数が実際には使われなくても…。

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

一般的に、メソッドは以下のように動作します。インスタンスの非データ属性が参照されたときは、そのインスタンスのクラスが検索されます。その名前が有効なクラス属性を表している関数オブジェクトなら、インスタンスオブジェクトと関数オブジェクトの両方への参照がメソッドオブジェクトにパックされます。メソッドオブジェクトが引数リストと共に呼び出されるとき、インスタンスオブジェクトと渡された引数リストから新しい引数リストを作成して、元の関数オブジェクトを新しい引数リストで呼び出します。

9.3.5 クラスとインスタンス変数

一般的に、インスタンス変数はそれぞれのインスタンスについて固有のデータのためのもので、クラス変数はそのクラスのすべてのインスタンスによって共有される属性やメソッドのためのものです:

```
class Dog:

    kind = 'canine'         # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance
```

(次のページに続く)

(前のページからの続き)

```
>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
>>> d.name                  # unique to d
'Fido'
>>> e.name                  # unique to e
'Buddy'
```

名前とオブジェクトについて で議論したように、共有データはリストや辞書のような *mutable* オブジェクトが関与すると驚くべき効果を持ち得ます。例えば、以下のコードの *tricks* リストはクラス変数として使われるべきではありません、なぜならたった一つのリストがすべての *Dog* インスタンスによって共有されることになり得るからです:

```
class Dog:

    tricks = []             # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

このクラスの正しい設計ではインスタンス変数を代わりに使用するべきです:

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
```

(次のページに続く)

(前のページからの続き)

```
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

9.4 いろいろな注意点

インスタンスとクラスの両方で同じ属性名が使用されている場合、属性検索はインスタンスが優先されます。

```
>>> class Warehouse:
...     purpose = 'storage'
...     region = 'west'
...
>>> w1 = Warehouse()
>>> print(w1.purpose, w1.region)
storage west
>>> w2 = Warehouse()
>>> w2.region = 'east'
>>> print(w2.purpose, w2.region)
storage east
```

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

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

メソッドの中から、データ属性を (または別のメソッドも！) 参照するための短縮された記法はありません。私は、この仕様がメソッドの可読性を高めていると感じています。あるメソッドを眺めているときにローカルな変数とインスタンス変数をはっきり区別できるからです。

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

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

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g
```

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

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

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

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

メソッドは、通常の関数と同じようにしてグローバルな名前を参照します。あるメソッドに関するグローバルスコープは、その定義を含むモジュールです。(クラスはグローバルなスコープとして用いられることはありません。) メソッドでグローバルなデータを使う良い理由はほとんどありませんが、グローバルなスコープを使うべき場面は多々あります。一つ挙げると、メソッド内から、グローバルなスコープに import された関数やモジュールや、そのモジュール中で定義された関数やクラスを使うことができます。通常、メソッドの入っているクラス自体はグローバルなスコープ内で定義されています。次の節では、メソッドが自分のクラスを参照する理由として正当なものを見てみましょう。

個々の値はオブジェクトなので、**クラス** (**型** とも言います) を持っています。それは object.__class__ に保持されています。

9.5 継承

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

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

基底クラス (base class) の名前 BaseClassName は、派生クラス定義の入っているスコープからアクセスできる名前空間で定義されていなければなりません。基底クラス名のかわりに任意の式を入れることもできます。これは次の例のように、基底クラスが別モジュールで定義されているときに便利なことがあります:

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

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

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

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

派生クラスで上書きしているメソッドでは、基底クラスの同名のメソッドを置き換えるのではなく、拡張したいのかもしれません。基底クラスのメソッドを直接呼び出す簡単な方法があります。単に BaseClassName.methodname(self, arguments) を呼び出すだけです。この仕様は、場合によってはクライアントでも役に立ちます。(この呼び出し方が動作するのは、基底クラスがグローバルスコープの BaseClassName という名前でアクセスできるときだけです。)

Python には継承に関係する 2 つの組み込み関数があります:

- isinstance() を使うとインスタンスの型が調べられます。isinstance(obj, int) は obj.__class__ が int や int の派生クラスの場合に限り True になります。

- issubclass() を使うとクラスの継承関係が調べられます。bool は int のサブクラスなので issubclass(bool, int) は True です。しかし、float は int のサブクラスではないので issubclass(float, int) は False です。

9.5.1 多重継承

Python では、多重継承 (multiple inheritance) の形式もサポートしています。複数の基底クラスをもつクラス定義は次のようになります:

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

ほとんどのシンプルな多重継承において、親クラスから継承される属性の検索は、深さ優先で、左から右に、そして継承の階層の中で同じクラスが複数出てくる（訳注: ダイアモンド継承と呼ばれます）場合に 2 度探索をしない、と考えることができます。なので、ある属性が DerivedClassName で見つからない場合、まず Base1 から検索され、そして（再帰的に）Base1 の基底クラスから検索され、それでも見つからなかった場合は Base2 から検索される、といった具合になります。

実際には、それよりもう少しだけ複雑です。協調的な super() の呼び出しのためにメソッドの解決順序は動的に変更されます。このアプローチは他の多重継承のある言語で call-next-method として知られており、単一継承しかない言語の super 呼び出しよりも強力です。

多重継承の全ての場合に 1 つかそれ以上のダイヤモンド継承 (少なくとも 1 つの祖先クラスに対し最も下のクラスから到達する経路が複数ある状態) があるので、動的順序付けが必要です。例えば、全ての新形式のクラスは object を継承しているので、どの多重継承でも object へ到達するための道は複数存在します。基底クラスが複数回アクセスされないようにするために、動的アルゴリズムで検索順序を直列化し、各クラスで指定されている祖先クラスどうしの左から右への順序は崩さず、各祖先クラスを一度だけ呼び出し、かつ単調になる (つまり祖先クラスの検索順序に影響を与えずにクラスをサブクラス化できる) ようにします。まとめると、これらの特徴のおかげで信頼性と拡張性のある多重継承したクラスを設計することができるのです。さらに詳細を知りたければ、https://www.python.org/download/releases/2.3/mro/ を見てください。

9.6 プライベート変数

オブジェクトの中からしかアクセス出来ない ” プライベート” インスタンス変数は、Python にはありません。しかし、ほとんどの Python コードが従っている慣習があります。アンダースコアで始まる名前 (例えば `_spam`) は、(関数であれメソッドであれデータメンバであれ) 非 public な API として扱います。これらは、予告なく変更されるかもしれない実装の詳細として扱われるべきです。

クラスのプライベートメンバについて適切なユースケース (特にサブクラスで定義された名前との衝突を避ける場合) があるので、名前マングリング (*name mangling*) と呼ばれる、限定されたサポート機構があります。`__spam` (先頭に二個以上の下線文字、末尾に一個以下の下線文字) という形式の識別子は、`_classname__spam` へとテキスト置換されるようになりました。ここで `classname` は、現在のクラス名から先頭の下線文字をはぎとった名前になります。このような難号化 (mangle) は、識別子の文法的な位置にかかわらず行われるので、クラス定義内に現れた識別子全てに対して実行されます。

名前マングリングは、サブクラスが内部のメソッド呼び出しを壊さずにメソッドをオーバーライドするのに便利です。例えば:

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

上の例は、もし仮に `MappingSubclass` に `__update` 識別子を実装したとしてもきちんと動きます。その理由は、`Mapping` クラスではその識別子を `_Mapping__update` に、`MappingSubclass` クラスでは `_MappingSubclass__update` にそれぞれ置き換えるからです。

難号化の規則は主に不慮の事故を防ぐためのものだということに注意してください; 確信犯的な方法で、プライベートとされている変数にアクセスしたり変更することは依然として可能なのです。デバッガのような特殊な状況では、この仕様は便利ですらあります。

`exec()` や `eval()` へ渡されたコードでは、呼出し元のクラス名を現在のクラスと見なさないことに注意してください。この仕様は `global` 文の効果と似ており、その効果もまた同様に、バイトコンパイルされたコードに制限さ

れています。同じ制約が getattr() と setattr() と delattr() にも適用されます。また、__dict__ を直接参照するときにも適用されます。

9.7 残りのはしばし

Pascal の ” レコード (record)” や、C 言語の ” 構造体 (struct)” のような、名前つきのデータ要素を一まとめにするデータ型があると便利なことがあります。慣用的な手法として、この目的のために dataclasses を使用します。

```
from dataclasses import dataclass

@dataclass
class Employee:
    name: str
    dept: str
    salary: int
```

```
>>> john = Employee('john', 'computer lab', 1000)
>>> john.dept
'computer lab'
>>> john.salary
1000
```

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

Instance method objects にも属性があります。m.__self__ はメソッド m() の属しているインスタンスオブジェクトで、m.__func__ はそのメソッドに対応する function object です。

9.8 イテレータ (iterator)

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

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
```

(次のページに続く)

(前のページからの続き)

```
for char in "123":
    print(char)
for line in open("myfile.txt"):
    print(line, end='')
```

こういう要素へのアクセス方法は明確で簡潔で使い易いものです。イテレータの活用は Python へ広く行き渡り、統一感を持たせています。裏では for 文はコンテナオブジェクトに対して iter() 関数を呼んでいます。関数は、コンテナの中の要素に 1 つずつアクセスする __next__() メソッドが定義されているイテレータオブジェクトを返します。これ以上要素が無い場合は、__next__() メソッドは StopIteration 例外を送出し、その通知を受け for ループは終了します。組み込みの next() 関数を使って __next__() メソッドを直接呼ぶこともできます; この例は関数がどう働くのかを示しています:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<str_iterator object at 0x10c90e650>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    next(it)
StopIteration
```

イテレータプロトコルの裏にある仕組みを観察していれば、自作のクラスにイテレータとしての振舞いを追加するのは簡単です。__next__() メソッドを持つオブジェクトを返す __iter__() メソッドを定義するのです。クラスが __next__() メソッドを定義している場合、__iter__() メソッドは単に self を返すことも可能です:

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

9.9 ジェネレータ (generator)

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

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

```
>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

ジェネレータでできることは、前の節で解説したクラスを使ったイテレータでも実現できます。ジェネレータの定義がコンパクトになるのは __iter__() メソッドと __next__() メソッドが自動で作成されるからです。

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

メソッドを自動生成したりプログラムの実行状態を自動保存するほかに、ジェネレータは終了時に自動的に StopIteration を送出します。これらの機能を組み合わせると、通常の関数を書くのと同じ労力で、簡単にイテレータを生成できます。

9.10 ジェネレータ式

単純なジェネレータなら式として簡潔にコーディングできます。その式はリスト内包表記に似た構文を使いますが、角括弧ではなく丸括弧で囲います。ジェネレータ式は、関数の中でジェネレータをすぐに使いたいような状況のために用意されています。ジェネレータ式は完全なジェネレータの定義よりコンパクトですが、ちょっと融通の効かないところがあります。同じ内容を返すリスト内包表記よりはメモリに優しいことが多いという利点があります。

例:

```
>>> sum(i*i for i in range(10))                 # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # dot product
260

>>> unique_words = set(word for line in page  for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```

脚注

第

TEN

標準ライブラリミニツアー

10.1 OS へのインターフェース

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:\\Python313'
>>> 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ーフェースが 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。例として以下の demo.py ファイルを見てみましょう:

```
# File demo.py
import sys
print(sys.argv)
```

以下は、コマンドライン上で python demo.py one two three と実行した時の出力です:

```
['demo.py', 'one', 'two', 'three']
```

argparse モジュールは、コマンドライン引数を処理するための更に洗練された仕組みを提供します。次のスクリプトは 1 つ以上のファイル名を抽出し、オプションで行数を表示します。

```
import argparse

parser = argparse.ArgumentParser(
    prog='top',
    description='Show top lines from each file')
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

コマンドラインで python top.py --lines=5 alpha.txt beta.txt を実行すると、上のスクリプトは args.lines を 5 、args.filenames を ['alpha.txt', 'beta.txt'] に設定します。

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
```

(次のページに続く)

(前のページからの続き)

```
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float from the interval [0.0, 1.0)
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://worldtimeapi.org/api/timezone/etc/UTC.txt') as response:
...     for line in response:
...         line = line.decode()             # Convert bytes to a str
...         if line.startswith('datetime'):
...             print(line.rstrip())         # Remove trailing newline
...
datetime: 2022-01-01T01:36:47.689215+00:00

>>> 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ーです。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。

第

ELEVEN

標準ライブラリミニツアー --- その 2

ツアーの第 2 部では、プロフェッショナルプログラミングを支えるもっと高度なモジュールをカバーします。ここで挙げるモジュールは、小さなスクリプトの開発ではほとんど使いません。

11.1 出力のフォーマット

reprlib モジュールは、大きなコンテナや、深くネストしたコンテナを省略して表示するバージョンの repr() を提供しています:

```
>>> import reprlib
>>> reprlib.repr(set('supercalifragilisticexpialidocious'))
"{'a', 'c', 'd', 'e', 'f', 'g', ...}"
```

pprint モジュールは、組み込み型やユーザ定義型をわかりやすく表示するための洗練された制御手段を提供しています。表示結果が複数行にわたる場合は、”pretty printer” と呼ばれるものが改行やインデントを追加して、データ構造がより明確になるように印字します:

```
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta',
...     'yellow'], 'blue']]]
...
>>> pprint.pprint(t, width=30)
[[[['black', 'cyan'],
   'white',
   ['green', 'red']],
  [['magenta', 'yellow'],
   'blue']]]
```

textwrap モジュールは、段落で構成された文章を、指定したスクリーン幅にぴったり収まるように調整します:

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
```

(次のページに続く)

(前のページからの続き)

```
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
...
>>> print(textwrap.fill(doc, width=40))
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

locale モジュールは、文化により異なるデータ表現形式のデータベースにアクセスします。locale の format() 関数の grouping 属性を使えば、数値を適切な桁区切り文字によりグループ化された形式に変換できます:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()          # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format_string("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'],
...                      conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
```

11.2 文字列テンプレート

string モジュールには、柔軟で、エンドユーザが簡単に編集できる簡単な構文を備えた Template クラスが入っています。このクラスを使うと、ユーザがアプリケーションを修正することなしにアプリケーションの出力をカスタマイズできるようになります。

テンプレートでは、$ と有効な Python 識別子名 (英数字とアンダースコア) からなるプレースホルダ名を使います。プレースホルダの周りを {} で囲えば、プレースホルダの後ろにスペースを挟まず、英数文字を続けることができます。$$ のようにすると、$ 自体をエスケープできます:

```
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

substitute() メソッドは、プレースホルダに相当する値が辞書やキーワード引数にない場合に KeyError を送出します。メールマージ機能のようなアプリケーションの場合、ユーザが入力するデータは不完全なことがあるので、欠落したデータがあるとプレースホルダをそのままにして出力する safe_substitute() メソッドを使う方が適切かもしれません:

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  ...
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

区切り文字はデフォルトは $ ですが、Template のサブクラスを派生すると変更することができます。例えば、画像ブラウザ用に一括で名前を変更するユーティリティを作っていたとして、現在の日付や画像のシーケンス番号、ファイル形式といったプレースホルダにパーセント記号を使うことにしたら、次のようになります:

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
...
>>> fmt = input('Enter rename style (%d-date %n-seqnum %f-format):  ')
Enter rename style (%d-date %n-seqnum %f-format):  Ashley_%n%f

>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print('{0} --> {1}'.format(filename, newname))

img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

テンプレートのもう一つの用途は、複数ある出力フォーマットからのプログラムロジックの分離です。これにより、XML ファイル用、プレーンテキストのレポート用、HTML の web レポート用のテンプレートに、同じプログラムロジックから値を埋め込むことができます。

11.3 バイナリデータレコードの操作

struct モジュールでは、様々な長さのバイナリレコード形式を操作する pack() や unpack() といった関数を提供しています。以下の例では、zipfile モジュールを使わずに、ZIP ファイルのヘッダ情報を巡回する方法を示しています。"H" と "I" というパック符号は、それぞれ 2 バイトと 4 バイトの符号無し 整数を表しています。"<" は、そのパック符号が standard サイズであり、バイトオーダーがリトルエンディアンであることを示しています:

```
import struct

with open('myfile.zip', 'rb') as f:
    data = f.read()

start = 0
for i in range(3):                      # show the first 3 file headers
    start += 14
    fields = struct.unpack('<IIIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size)

    start += extra_size + comp_size     # skip to the next header
```

11.4 マルチスレッディング

スレッド処理 (threading) とは、順序的な依存関係にない複数のタスクを分割するテクニックです。スレッドは、ユーザの入力を受け付けつつ、背後で別のタスクを動かすようなアプリケーションの応答性を高めます。同じような使用例として、I/O を別のスレッドの計算処理と並列して動作させるというものがあります。

以下のコードでは、高水準のモジュール threading でメインのプログラムを動かしながら背後で別のタスクを動作させられるようにする方法を示しています:

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')
```

(次のページに続く)

(前のページからの続き)

```
background.join()    # Wait for the background task to finish
print('Main program waited until background was done.')
```

マルチスレッドアプリケーションを作る上で最も難しい問題は、データやリソースを共有するスレッド間の調整 (coordination) です。この問題を解決するため、threading モジュールではロックやイベント、状態変数、セマフォといった数々の同期プリミティブを提供しています。

こうしたツールは強力な一方、ちょっとした設計上の欠陥で再現困難な問題を引き起こすことがあります。したがって、タスク間調整では queue モジュールを使って他の複数のスレッドからのリクエストを一つのスレッドに送り込み、一つのリソースへのアクセスをできるだけ一つのスレッドに集中させるほうが良いでしょう。スレッド間の通信や調整に Queue オブジェクトを使うと、設計が容易になり、可読性が高まり、信頼性が増します。

11.5 ログ記録

logging モジュールでは、数多くの機能をそなえた柔軟性のあるログ記録システムを提供しています。最も簡単な使い方では、ログメッセージをファイルや sys.stderr に送信します:

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

これは以下の出力を生成します:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

デフォルトでは、info() と debug() による出力は抑制され、出力は標準エラーに送信されます。選択可能な送信先には、email、データグラム、ソケット、HTTP サーバへの送信などがあります。新たにフィルタを作成すると、DEBUG、INFO、WARNING、ERROR、CRITICAL といったメッセージのプライオリティによって異なる送信先を選択することができます。

ログ記録システムは Python から直接設定することもできますし、アプリケーションを変更しなくてもカスタマイズできるよう、ユーザが編集可能な設定ファイルによって設定することもできます。

11.6 弱参照

Python は自動的にメモリを管理します (ほとんどのオブジェクトは参照カウント方式で管理し、ガベージコレクション で循環参照を除去します)。オブジェクトに対する最後の参照がなくなってしばらくするとメモリは解放されます。

このようなアプローチはほとんどのアプリケーションでうまく動作しますが、中にはオブジェクトをどこか別の場所で利用している間だけ追跡しておきたい場合もあります。残念ながら、オブジェクトを追跡するだけでオブジェクトに対する恒久的な参照を作ることになってしまいます。weakref モジュールでは、オブジェクトへの参照を作らずに追跡するためのツールを提供しています。弱参照オブジェクトが不要になると、弱参照 (weakref) テーブルから自動的に除去され、コールバック関数がトリガされます。弱参照を使う典型的な応用例には、作成コストの大きいオブジェクトのキャッシュがあります:

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return str(self.value)
...
>>> a = A(10)                   # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a            # does not create a reference
>>> d['primary']                # fetch the object if it is still alive
10
>>> del a                       # remove the one reference
>>> gc.collect()                # run garbage collection right away
0
>>> d['primary']                # entry was automatically removed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']                # entry was automatically removed
  File "C:/python313/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7 リスト操作のためのツール

多くのデータ構造は、組み込みリスト型を使った実装で事足ります。とはいえ、時には組み込みリストとは違うパフォーマンス上のトレードオフを持つような実装が必要になることもあります。

array (配列) モジュールでは、array() オブジェクトを提供しています。配列はリストに似ていますが、同じ形式のデータだけが保存でき、よりコンパクトに保存されます。以下の例では、通常 1 要素あたり 16 バイトを必要とする Python 整数型のリストの 代りに、2 バイトの符号無しの 2 進数 (タイプコード "H") の配列を使ってい

ます:

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

collections モジュールでは、deque() オブジェクトを提供しています。リスト型に似ていますが、データの追加と左端からの取り出しが速く、その一方で中間にある値の参照は遅くなります。こうしたオブジェクトはキューや木構造の幅優先探索の実装に向いています:

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
>>> print("Handling", d.popleft())
Handling task1
```

```
unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
```

リストの代わりの実装以外にも、標準ライブラリにはソート済みのリストを操作するための関数を備えた bisect のようなツールも提供しています:

```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

heapq モジュールは、通常のリストでヒープを実装するための関数を提供しています。ヒープでは、最も低い値をもつエントリがつねにゼロの位置に配置されます。ヒープは、毎回リストをソートすることなく、最小の値をもつ要素に繰り返しアクセスするようなアプリケーションで便利です:

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)                      # rearrange the list into heap order
>>> heappush(data, -5)                 # add a new entry
>>> [heappop(data) for i in range(3)]  # fetch the three smallest entries
[-5, 0, 1]
```

11.8 10 進浮動小数演算

decimal モジュールでは、10 進浮動小数の算術演算をサポートする Decimal データ型を提供しています。組み込みの 2 進浮動小数の実装である float に比べて、このクラスがとりわけ便利なのは、以下の場合です

- 財務アプリケーションやその他の正確な 10 進表記が必要なアプリケーション、
- 精度の制御、
- 法的または規制上の理由に基づく値丸めの制御、
- 有効桁数の追跡が必要になる場合
- ユーザが手計算の結果と同じ演算結果を期待するようなアプリケーション。

例えば、70 セントの電話代にかかる 5% の税金を計算しようとすると、10 進の浮動小数点値と 2 進の浮動小数点値では違う結果になってしまいます。計算結果を四捨五入してセント単位にしようとすると、以下のように違いがはっきり現れます:

```
>>> from decimal import *
>>> round(Decimal('0.70') * Decimal('1.05'), 2)
Decimal('0.74')
>>> round(.70 * 1.05, 2)
0.73
```

上の例で、Decimal を使った計算では、末尾桁のゼロが保存されており、有効数字 2 桁の被乗数から自動的に有効数字を 4 桁と判断しています。Decimal は手計算と 同じ方法で計算を行い、2 進浮動小数が 10 進小数成分を正確に表現できないことに よって起きる問題を回避しています。

Decimal クラスは厳密な値を表現できるため、2 進浮動小数点数では 期待通りに計算できないような剰余の計算や等値テストも実現できます:

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 1.0
False
```

decimal モジュールを使うと、必要なだけの精度で算術演算を行えます:

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```

第

TWELVE

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。

この問題の解決策は 仮想環境 を作ることです。仮想環境とは、特定のバージョンの Python と幾つかの追加パッケージを含んだ Python インストールを構成するディレクトリです。

別のアプリケーションはそれぞれ別の仮想環境を使うことができます。先の例にあった要求の衝突を解決する場合、アプリケーション A が固有の仮想環境を持ってそこにライブラリのバージョン 1.0 をインストールし、アプリケーション B が持つ別の仮想環境にライブラリのバージョン 2.0 をインストールすることができます。そしてアプリケーション B がライブラリのバージョンを 3.0 に更新することを要求する場合も、アプリケーション A に影響しません。

12.2 仮想環境の作成

仮想環境の作成と管理を行うためのモジュールが `venv` です。`venv` は通常利用可能なもっとも新しいバージョンの Python をインストールします。複数のバージョンの Python がインストールされている場合、`python3` のように利用したいバージョンを指定して実行することで Python バージョンを選択できます。

仮想環境を作るには、仮想環境を置くディレクトリを決めて、そのディレクトリのパスを指定して、`venv` をスクリプトとして実行します:

```
python -m venv 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、その他関連するファイルのコピーを含むサブディレクトリを作ります。

仮想環境の一般的なディレクトリの場所は .venv です。この名前は、通常はシェルで隠されているため、ディレクトリが存在する理由を説明する名前を付けても、邪魔にはなりません。また、一部のツールでサポートされている .env 環境変数定義ファイルによるクラッシュも防止します。

仮想環境を作ったら、それを有効化する必要があります。

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。)

仮想環境を有効化すると、シェルのプロンプトに利用中の仮想環境が表示されるようになり、python を実行するとその仮想環境の Python を実行するようになります:

```
$ source ~/envs/tutorial-env/bin/activate
(tutorial-env) $ python
Python 3.5.1 (default, May  6 2016, 10:59:36)
  ...
>>> import sys
>>> sys.path
['', '/usr/local/lib/python35.zip', ...,
'~/envs/tutorial-env/lib/python3.5/site-packages']
>>>
```

仮想環境を無効化するには、ターミナルに:

```
deactivate
```

と入力します。

12.3 pip を使ったパッケージ管理

pip と呼ばれるプログラムでパッケージをインストール、アップグレード、削除することができます。デフォルトでは pip は Python Package Index からパッケージをインストールします。ブラウザを使って Python Package Index を閲覧することができます。

pip は "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 installing-index ガイドを参照してください。)

パッケージ名を指定してそのパッケージの最新版をインストールすることができます:

```
(tutorial-env) $ python -m 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
```

パッケージ名のあとに == とバージョン番号を付けることで、特定のバージョンのパッケージをインストールすることもできます:

```
(tutorial-env) $ python -m 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
```

同じコマンドを再び実行した場合、pip は要求されたバージョンがインストール済みだと表示して何もしません。別のバージョン番号を指定すればそのバージョンをインストールしますし、python -m pip install --upgrade を実行すればそのパッケージを最新版に更新します:

```
(tutorial-env) $ python -m 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
```

python -m 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。

python -m pip show は指定されたパッケージの情報を表示します:

```
(tutorial-env) $ python -m pip show requests
---
Metadata-Version: 2.0
```

(次のページに続く)

(前のページからの続き)

```
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:
```

`python -m pip list` は仮想環境にインストールされた全てのパッケージを表示します:

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

`python -m pip freeze` はインストールされたパッケージ一覧を、`python -m pip install` が解釈するフォーマットで生成します。一般的な慣習として、このリストを `requirements.txt` というファイルに保存します:

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

`requirements.txt` をバージョン管理システムにコミットして、アプリケーションの一部として配布することができます。ユーザーは必要なパッケージを `install -r` でインストールできます:

```
(tutorial-env) $ python -m 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 installing-index 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 packaging user guide を参照してください。

第

THIRTEEN

さあ何を？

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

このチュートリアルは Python のドキュメンテーションセットの一部です。セットの中の他のドキュメンテーションをいくつか紹介します:

- library-index:

  このマニュアルをざっと眺めておくと便利です。このマニュアルは型、関数、標準ライブラリのモジュールについての完全なリファレンスです。標準的な Python 配布物は **たくさんの** 追加コードを含んでいます。Unix メールボックスの読み込み、HTTP によるドキュメント取得、乱数の生成、コマンドラインオプションの構文解析、データ圧縮やその他たくさんのタスクのためのモジュールがあります。ライブラリリファレンスをざっと見ることで、何が利用できるかのイメージをつかむことができます。

- installing-index は、他の Python ユーザによって書かれた追加モジュールをどうやってインストールするかを説明しています。

- reference-index: Python の文法とセマンティクスを詳しく説明しています。読むのは大変ですが、言語の完全なガイドとして有用です。

さらなる Python に関するリソース:

- https://www.python.org: 有名な Python の Web サイト。コードやドキュメント、Python に関する Web ページへのポインターを含んでいます。

- https://docs.python.org: Python ドキュメントへの素早いアクセスを提供します。

- https://pypi.org: Python パッケージインデックス、以前は Cheese Shop[*1] という愛称でも呼ばれていました。これは、ユーザ作成のダウンロードできる Python モジュールの索引です。コードのリリースをしたら、ここに登録することで他の人が見つけられます。

[*1] "Cheese Shop" は Monty Python のスケッチです: お客さんがチーズ屋に入ったけれど、彼が求めたチーズはどれも無いと clerk が言った。

- https://code.activestate.com/recipes/langs/python/: Python クックブックはコード例、モジュール、実用的なスクリプトの巨大なコレクションです。主要なものは同名の本 Python Cookbook (O'Reilly & Associates, ISBN 0-596-00797-3) に収録されています。
- https://pyvideo.org は学会やユーザグループの会合から Python 関連のビデオのリンクを集めています。
- https://scipy.org: Scientific Python プロジェクトは配列の高速な計算・操作モジュールに加え、線形代数、フーリエ変換、非線形ソルバー、乱数分布、統計分析などの多くのパッケージを提供しています。

Python 関連の質問や問題の報告については、ニュースグループ *comp.lang.python* に投稿するか、またはメーリングリスト python-list@python.org に送ることができます。ニュースグループとメーリングリストはゲートウェイされます。したがって、片方に投稿されたメッセージは、もう片方へ自動的に転送されます。質問 (と回答)、新しい機能の提案、新しいモジュールの発表などで、1 日に数百通の投稿があります。メーリングリストのアーカイブは https://mail.python.org/pipermail/ で利用可能です。

投稿の前に、必ず よくある質問 (FAQ とも呼ばれます) のリストをチェックしてください。FAQ は繰り返し取り上げられる多くの質問に答えています。あなたの問題に対する解決が既に含まれているかもしれません。

脚注

第

FOURTEEN

対話入力編集と履歴置換

いくつかのバージョンの Python インタプリタでは、Korn シェルや GNU Bash シェルに見られる機能に似た、現在の入力行に対する編集機能や履歴置換機能をサポートしています。この機能は様々な編集スタイルをサポートしている、GNU Readline ライブラリを使って実装されています。このライブラリには独自のドキュメントがあり、ここでそれを繰り返すつもりはありません。

14.1 タブ補完と履歴編集

変数とモジュール名の補完はインタプリタの起動時に 自動的に有効化されます 。従って `Tab` キーは補完機能を呼び出し、Python の文の名前、現在のローカル変数、および利用可能なモジュール名を検索します。`string.a` のようなドットで区切られた式については、最後の `'.'` までの式を評価し、結果として得られたオブジェクトの属性から補完候補を示します。`__getattr__()` メソッドを持ったオブジェクトが式に含まれている場合、`__getattr__()` がアプリケーション定義のコードを実行するかもしれないので注意してください。デフォルトの設定ではあなたのユーザーディレクトリの `.python_history` という名前のファイルに履歴を保存します。履歴は次回の対話的なインタプリタのセッションで再び利用することができます。

14.2 対話的インタープリタの代替

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

より優れた対話的インタープリタの代替の一つに IPython があります。このインタープリタは、様々なところで使われていて、タブ補完、オブジェクト探索や先進的な履歴管理といった機能を持っています。他のアプリケーションにカスタマイズされたり、組込まれこともあります。別の優れたインタラクティブ環境としては bpython があります。

第

FIFTEEN

浮動小数点演算、その問題と制限

浮動小数点数はコンピューターのハードウェア上は 2 進数 (binary) の分数で表されます。たとえば、**10 進数** の分数では `0.625` は 6/10 + 2/100 + 5/1000 という値を持ち、**2 進数** の分数では `0.101` は 1/2 + 0/4 + 1/8 という値を持ちます。この 2 つの分数はまったく同じ値を持ち、唯一異なる点は 1 つ目が 10 進数の分数で書かれており、2 つ目は 2 進数の分数で書かれているということです。

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

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

```
0.3
```

さらに正確な近似は、

```
0.33
```

さらに正確な近似は、

```
0.333
```

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

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

```
0.0001100110011001100110011001100110011001100110011001100110011...
```

どこか有限の桁で止めると、近似値を得ることになります。近年の殆どのコンピュータでは float 型は、最上位ビットから数えて最初の 53 ビットを分子、2 の冪乗を分母とした、二進小数で近似されます。1/10 の場合は、二進小数は `3602879701896397 / 2 ** 55` となります。これは、1/10 に近いですが、厳密に同じ値ではありません。

値が表示される方法のために、ほとんどのユーザは、近似に気づきません。Python はマシンに格納されている二進近似値の 10 進小数での近似値を表示するので、格納されている値が元の 10 進小数の近似値でしか無いことを忘れがちです。ほとんどのマシンで、もし Python が 2 進数で近似された 0.1 の近似値をそのまま 10 進数で表示していたら、その結果は次のようになったでしょう:

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

これは、ほとんどの人が必要と感じるよりも多すぎる桁数です。なので、Python は丸めた値を表示することで、桁数を扱いやすい範囲にとどめます:

```
>>> 1 / 10
0.1
```

表示された結果が正確に 1/10 であるように見えたとしても、実際に格納されている値は最も近く表現できる二進小数であるということだけは覚えておいてください。

幾つかの異なる 10 進数の値が、同じ 2 進有理数の近似値を共有しています。例えば、`0.1` と `0.10000000000000001` と `0.1000000000000000055511151231257827021181583404541015625` はどれも `3602879701896397 / 2 ** 55` に近似されます。同じ近似値を共有しているので、どの 10 進数の値も `eval(repr(x)) == x` という条件を満たしたまま同じように表示されます。

昔の Python は、プロンプトと `repr()` ビルトイン関数は 17 桁の有効数字を持つ `0.10000000000000001` のような 10 進数の値を選んで表示していました。Python 3.1 からは、ほとんどの場面で `0.1` のような最も短い桁数の 10 進数の値を選ぶようになりました。

この動作は 2 進数の浮動小数点にとってはごく自然なものです。これは Python のバグではありませんし、あなたのコードのバグでもありません。ハードウェアの浮動小数点演算をサポートしている全ての言語で同じ種類の問題を見つけることができます (いくつかの言語ではデフォルトの、あるいはどの出力モードを選んでも、この差を **表示** しないかもしれませんが)。

よりよい出力のために、文字列フォーマットを利用して有効桁数を制限した 10 進数表現を得ることができます:

```
>>> format(math.pi, '.12g')  # give 12 significant digits
'3.14159265359'

>>> format(math.pi, '.2f')   # give 2 digits after the point
'3.14'

>>> repr(math.pi)
'3.141592653589793'
```

これが、実際のコンピューター上の値の **表示** を丸めているだけの、いわば錯覚だということを認識しておいてください。

もう一つの錯覚を紹介します。例えば、0.1 が正確には 1/10 ではないために、それを 3 回足した値もまた正確に

は 0.3 ではありません:

```
>>> 0.1 + 0.1 + 0.1 == 0.3
False
```

0.1 はこれ以上 1/10 に近くなることができない値で、0.3 もまた 3/10 に一番近い値なので、round() 関数を使って計算前に丸めを行なっても意味がありません:

```
>>> round(0.1, 1) + round(0.1, 1) + round(0.1, 1) == round(0.3, 1)
False
```

数値を意図した正確な値に近づけることはできませんが、math.isclose() 関数は不正確な値を比べるのに便利です:

```
>>> math.isclose(0.1 + 0.1 + 0.1, 0.3)
True
```

あるいは、round() 関数を粗い近似値比較に使うこともできます:

```
>>> round(math.pi, ndigits=2) == round(22 / 7, ndigits=2)
True
```

このように 2 進数の浮動小数点の演算には多くの驚きがあります。「0.1」の問題について詳しい説明は、「表現エラー」セクションで行います。2 進数の浮動小数点の仕組みと、実際によく遭遇する問題各種についての分かりやすい概要は、Examples of Floating Point Problems を参照してください。その他よくある驚きの より詳細な説明は The Perils of Floating Point も参照してください。

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

異常なケースが存在する一方で、普段の浮動小数点演算の利用では、単に最終的な結果の値を必要な 10 進の桁数に丸めて表示するのなら、最終的には期待通りの結果を得ることになるでしょう。たいては str() で十分ですが、きめ細かな制御をしたければ、formatstrings にある str.format() メソッドのフォーマット仕様を参照してください。

正確な 10 進数表現が必要となるような場合には、decimal モジュールを利用してみてください。このモジュールは会計アプリケーションや高精度の計算が求められるアプリケーションに適した、10 進数の計算を実装しています。

別の正確な計算方法として、fractions モジュールが有理数に基づく計算を実装しています (1/3 のような数を正確に表すことができます)。

あなたが浮動小数点演算のヘビーユーザーなら、SciPy プロジェクトが提供している NumPy パッケージやその他の数学用パッケージを調べてみるべきです。<https://scipy.org> を参照してください。

Python は **本当に** float の正確な値が必要なレアケースに対応するためのツールを提供しています。`float.as_integer_ratio()` メソッドは float の値を有理数として表現します:

```
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

この分数は正確なので、元の値を完全に復元することができます:

```
>>> x == 3537115888337719 / 1125899906842624
True
```

`float.hex()` メソッドは float の値を 16 進数で表現します。この値もコンピューターが持っている正確な値を表現できます:

```
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

この正確な 16 進数表現はもとの float 値を正確に復元するために使うことができます:

```
>>> x == float.fromhex('0x1.921f9f01b866ep+1')
True
```

この 16 進数表現は正確なので、値を (プラットフォームにも依存せず) バージョンの異なる Python 間でやり取りしたり、他のこのフォーマットをサポートした言語 (Java や C99 など) と正確にやり取りするのに利用することができます。

別の便利なツールとして、合計処理における精度のロスを緩和してくれる `sum()` 関数があります。これは累計加算中の丸めに拡張精度を使います。これにより、誤差が最終的な合計値に影響を与えるまで蓄積されなくなり、結果が改善されます:

```
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 1.0
False
>>> sum([0.1] * 10) == 1.0
True
```

`math.fsum()` はさらに進んで、累計の加算時に「失われた桁」をすべて追跡し、結果の丸めは一度だけです。これは `sum()` より遅いものの、大きな入力がほとんど相殺され、最終的な合計がゼロに近くなるような珍しいケースでは、より正確です:

```
>>> arr = [-0.10430216751806065, -266310978.67179024, 143401161448607.16,
...        -143401161400469.7, 266262841.31058735, -0.003244936839808227]
```

(次のページに続く)

(前のページからの続き)

```
>>> float(sum(map(Fraction, arr)))   # Exact summation with single rounding
8.042173697819788e-13
>>> math.fsum(arr)                   # Single rounding
8.042173697819788e-13
>>> sum(arr)                         # Multiple roundings in extended precision
8.042178034628478e-13
>>> total = 0.0
>>> for x in arr:
...     total += x                   # Multiple roundings in standard precision
...
>>> total                            # Straight addition has no correct digits!
-0.0051575902860057365
```

15.1 表現エラー

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

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

なぜそうなるのでしょう？ 1/10 は 2 進法の小数で厳密に表現できません。少なくとも 2000 年以降、ほぼすべてのマシンは IEEE 754 2 進数の浮動小数点演算を用いており、ほぼすべてのプラットフォームでは Python の浮動小数点を IEEE 754 binary64 ” 倍精度 (double precision)” 値に対応付けます。IEEE 754 binary64 値は 53 ビットの精度を持つため、計算機に入力を行おうとすると、可能な限り 0.1 を最も近い値の分数に変換し、$J/2^{**}N$ の形式にしようと努力します。J はちょうど 53 ビットの精度の整数です。

```
1 / 10 ~= J / (2**N)
```

を書き直すと

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

となります。J は厳密に 53 ビットの精度を持っている (`>= 2**52` だが `< 2**53`) ことを思い出すと、N として最適な値は 56 になります:

```
>>> 2**52 <=  2**56 // 10  < 2**53
True
```

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

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

剰余が 10 の半分以上なので、最良の近似は切り上げて丸めたものになります。:

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

従って、IEEE 754 の倍精度における 1/10 の取りえる最良の近似は:

```
7205759403792794 / 2 ** 56
```

分子と分母を 2 で割って分数を小さくします:

```
3602879701896397 / 2 ** 55
```

丸めたときに切り上げたので、この値は実際には 1/10 より少し大きいことに注目してください。もし切り捨てをした場合は、商は 1/10 よりもわずかに小さくなります。どちらにしろ **厳密な** 1/10 ではありません！

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

```
>>> 0.1 * 2 ** 55
3602879701896397.0
```

この分数に 10**55 を掛ければ、55 桁の十進数の値を見ることができます:

```
>>> 3602879701896397 * 10 ** 55 // 2 ** 55
1000000000000000055511151231257827021181583404541015625
```

これは、計算機が記憶している正確な数値が、10 進数値 0.1000000000000000055511151231257827021181583404541015625 にほぼ等しいということです。多くの言語 (古いバージョンの Python を含む) では、完全な 10 進値を表示するのではなく、結果を有効数字 17 桁に丸めます:

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

fractions モジュールと decimal モジュールを使うとこれらの計算を簡単に行えます:

```
>>> from decimal import Decimal
>>> from fractions import Fraction

>>> Fraction.from_float(0.1)
Fraction(3602879701896397, 36028797018963968)
```

(次のページに続く)

(前のページからの続き)

```
>>> (0.1).as_integer_ratio()
(3602879701896397, 36028797018963968)

>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')

>>> format(Decimal.from_float(0.1), '.17')
'0.10000000000000001'
```

第

SIXTEEN

付録

16.1 対話モード

16.1.1 エラー処理

エラーが発生すると、インタープリタはエラーメッセージとスタックトレースを表示します。対話モードでは、それから元のプロンプトに戻ります; ファイルから実行した場合は、スタックトレースを表示してゼロ以外の終了ステータスで終了します（try 文の except 節で処理された例外は、ここでいうエラーにはあたりません。）一部のエラーは無条件に致命的であり、ゼロ以外の終了ステータスで終了します; これは内部の不整合やある種のメモリ不足の場合に適用されます。エラーメッセージは全て標準エラー出力に書き込まれます; これに対して、通常は実行した命令から出力される内容は標準出力に書き込まれます。

割り込み文字 (interrupt character、普通は Control-C か Delete) を一次または二次プロンプトに対してタイプすると、入力が取り消されて一次プロンプトに戻ります。[*1] コマンドの実行中に割り込み文字をタイプすると KeyboardInterrupt 例外が送出されます。この例外は try 文で処理できます。

16.1.2 実行可能な Python スクリプト

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

```
#!/usr/bin/env python3.5
```

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

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

chmod コマンドを使えば、スクリプトに実行モードや実行権限を与えることができます。

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

Windows では、” 実行モード” のような概念はありません。Python のインストーラーは自動的に .py ファイルを python.exe に関連付けるので、Python ファイルをダブルクリックするとそれをスクリプトとして実行します。.pyw 拡張子も (訳注: pythonw.exe に) 関連付けられ、通常コンソールウィンドウを抑制して実行します。

16.1.3 対話モード用の起動時実行ファイル

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

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

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

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    with open(filename) as fobj:
        startup_file = fobj.read()
    exec(startup_file)
```

16.1.4 カスタマイズ用モジュール

Python はユーザーが Python をカスタマイズするための 2 つのフック、sitecustomize と usercustomize を提供しています。これがどのように動作しているかを知るには、まずはユーザーの site-packages ディレクトリの場所を見つける必要があります。Python を起動して次のコードを実行してください:

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python3.5/site-packages'
```

usercustomize.py をそのディレクトリに作成して、そこでやりたいことをすべて書くことができます。このファイルは自動インポートを無効にする -s オプションを使わない限り、全ての Python の起動時に実行されます。

sitecustomize モジュールも同じように動作しますが、一般的にコンピューターの管理者によって、グローバルの site-packages ディレクトリに作成され、usercustomize より先にインポートされます。詳細は site モジュールのドキュメントを参照してください。

脚注

付録

A

用語集

>>> インタラクティブシェルにおけるデフォルトの Python プロンプトです。インタプリタでインタラクティブに実行されるコード例でよく出てきます。

... 次のものが考えられます:

- インタラクティブシェルにおいて、インデントされたコードブロック、対応する左右の区切り文字の組 (丸括弧、角括弧、波括弧、三重引用符) の内側、デコレーターの後に、コードを入力する際に表示されるデフォルトの Python プロンプトです。
- 組み込みの定数 `Ellipsis` 。

abstract base class
: (抽象基底クラス) 抽象基底クラスは *duck-typing* を補完するもので、`hasattr()` などの別のテクニックでは不恰好であったり微妙に誤る (例えば magic methods の場合) 場合にインターフェースを定義する方法を提供します。ABC は仮想 (virtual) サブクラスを導入します。これは親クラスから継承しませんが、それでも `isinstance()` や `issubclass()` に認識されます; `abc` モジュールのドキュメントを参照してください。Python には、多くの組み込み ABC が同梱されています。その対象は、(`collections.abc` モジュールで) データ構造、(`numbers` モジュールで) 数、(`io` モジュールで) ストリーム、(`importlib.abc` モジュールで) インポートファインダ及びローダーです。`abc` モジュールを利用して独自の ABC を作成できます。

annotation
: (アノテーション) 変数、クラス属性、関数のパラメータや返り値に関係するラベルです。慣例により *type hint* として使われています。

  ローカル変数のアノテーションは実行時にはアクセスできませんが、グローバル変数、クラス属性、関数のアノテーションはそれぞれモジュール、クラス、関数の `__annotations__` 特殊属性に保持されています。

  機能の説明がある *variable annotation*, *function annotation*, **PEP 484**, **PEP 526** を参照してください。また、アノテーションを利用するベストプラクティスとして annotations-howto も参照してください。

引数 (argument)

(実引数) 関数を呼び出す際に、**関数** (または **メソッド**) に渡す値です。実引数には 2 種類あります:

- **キーワード引数**: 関数呼び出しの際に引数の前に識別子がついたもの (例: `name=`) や、`**` に続けた辞書の中の値として渡された引数。例えば、次の `complex()` の呼び出しでは、`3` と `5` がキーワード引数です:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- **位置引数**: キーワード引数以外の引数。位置引数は引数リストの先頭に書くことができ、また `*` に続けた *iterable* の要素として渡すことができます。例えば、次の例では `3` と `5` は両方共位置引数です:

```
complex(3, 5)
complex(*(3, 5))
```

実引数は関数の実体において名前付きのローカル変数に割り当てられます。割り当てを行う規則については calls を参照してください。シンタックスにおいて実引数を表すためにあらゆる式を使うことが出来ます。評価された値はローカル変数に割り当てられます。

仮引数 、FAQ の 実引数と仮引数の違いは何ですか? 、**PEP 362** を参照してください。

asynchronous context manager

(非同期コンテキストマネージャ) `__aenter__()` と `__aexit__()` メソッドを定義することで `async with` 文内の環境を管理するオブジェクトです。**PEP 492** で導入されました。

asynchronous generator

(非同期ジェネレータ) *asynchronous generator iterator* を返す関数です。`async def` で定義されたコルーチン関数に似ていますが、`yield` 式を持つ点で異なります。`yield` 式は `async for` ループで使用できる値の並びを生成するのに使用されます。

通常は非同期ジェネレータ関数を指しますが、文脈によっては **非同期ジェネレータイテレータ** を指す場合があります。意図された意味が明らかでない場合、明瞭化のために完全な単語を使用します。

非同期ジェネレータ関数には、`async for` 文や `async with` 文だけでなく `await` 式もあることがあります。

asynchronous generator iterator

(非同期ジェネレータイテレータ) *asynchronous generator* 関数で生成されるオブジェクトです。

これは *asynchronous iterator* で、`__anext__()` メソッドを使って呼ばれると awaitable オブジェクトを返します。この awaitable オブジェクトは、次の `yield` 式まで非同期ジェネレータ関数の本体を実行します。

各 `yield` では一時的に処理を中断し、その場の実行状態 (ローカル変数や保留中の try 文を含む) を記憶します。**非同期ジェネレータイテレータ** が `__anext__()` で返された他の awaitable で実際に再開する時

には、その中断箇所が選ばれます。**PEP 492** および **PEP 525** を参照してください。

asynchronous iterable
: (非同期イテラブル) `async for` 文の中で使用できるオブジェクトです。自身の `__aiter__()` メソッドから *asynchronous iterator* を返さなければなりません。**PEP 492** で導入されました。

asynchronous iterator
: (非同期イテレータ) `__aiter__()` と `__anext__()` メソッドを実装したオブジェクトです。`__anext__()` は *awaitable* オブジェクトを返さなければなりません。`async for` は `StopAsyncIteration` 例外を送出するまで、非同期イテレータの `__anext__()` メソッドが返す awaitable を解決します。**PEP 492** で導入されました。

属性
: (属性) オブジェクトに関連付けられ、ドット表記式によって名前で通常参照される値です。例えば、オブジェクト *o* が属性 *a* を持っているとき、その属性は *o.a* で参照されます。

  オブジェクトには、identifiers で定義される識別子ではない名前の属性を与えることができます。たとえば `setattr()` を使い、オブジェクトがそれを許可している場合に行えます。このような属性はドット表記式ではアクセスできず、代わりに `getattr()` を使って取る必要があります。

awaitable
: (待機可能) `await` 式で使用することが出来るオブジェクトです。*coroutine* か、`__await__()` メソッドがあるオブジェクトです。**PEP 492** を参照してください。

BDFL
: 慈悲深き終身独裁者 (Benevolent Dictator For Life) の略です。Python の作者、Guido van Rossum のことです。

binary file
: (バイナリファイル) *bytes-like* **オブジェクト** の読み込みおよび書き込みができる **ファイルオブジェクト** です。バイナリファイルの例は、バイナリモード (`'rb'`, `'wb'` or `'rb+'`) で開かれたファイル、`sys.stdin.buffer`、`sys.stdout.buffer`、`io.BytesIO` や `gzip.GzipFile`. のインスタンスです。

  `str` オブジェクトの読み書きができるファイルオブジェクトについては、*text file* も参照してください。

borrowed reference
: In Python's C API, a borrowed reference is a reference to an object, where the code using the object does not own the reference. It becomes a dangling pointer if the object is destroyed. For example, a garbage collection can remove the last *strong reference* to the object and so destroy it.

  Calling `Py_INCREF()` on the *borrowed reference* is recommended to convert it to a *strong reference* in-place, except when the object cannot be destroyed before the last usage of the borrowed reference. The `Py_NewRef()` function can be used to create a new *strong reference*.

bytes-like object
: bufferobjects をサポートしていて、C 言語の意味で **連続した** バッファーを提供可能なオブジェクト。

bytes, bytearray, array.array や、多くの一般的な memoryview オブジェクトがこれに当たります。bytes-like オブジェクトは、データ圧縮、バイナリファイルへの保存、ソケットを経由した送信など、バイナリデータを要求するいろいろな操作に利用することができます。

幾つかの操作ではバイナリデータを変更する必要があります。その操作のドキュメントではよく ” 読み書き可能な bytes-like オブジェクト” に言及しています。変更可能なバッファーオブジェクトには、bytearray と bytearray の memoryview などが含まれます。また、他の幾つかの操作では不変なオブジェクト内のバイナリデータ (” 読み出し専用の bytes-like オブジェクト”) を必要します。それには bytes と bytes の memoryview オブジェクトが含まれます。

bytecode

(バイトコード) Python のソースコードは、Python プログラムの CPython インタプリタの内部表現であるバイトコードへとコンパイルされます。バイトコードは .pyc ファイルにキャッシュされ、同じファイルが二度目に実行されるときはより高速になります (ソースコードからバイトコードへの再度のコンパイルは回避されます)。この ” 中間言語 (intermediate language)” は、各々のバイトコードに対応する機械語を実行する **仮想マシン** で動作するといえます。重要な注意として、バイトコードは異なる Python 仮想マシン間で動作することや、Python リリース間で安定であることは期待されていません。

バイトコードの命令一覧は dis モジュール にあります。

callable

A callable is an object that can be called, possibly with a set of arguments (see *argument*), with the following syntax:

```
callable(argument1, argument2, argumentN)
```

A *function*, and by extension a *method*, is a callable. An instance of a class that implements the `__call__()` method is also a callable.

callback

(コールバック) 将来のある時点で実行されるために引数として渡される関数

クラス

(クラス) ユーザー定義オブジェクトを作成するためのテンプレートです。クラス定義は普通、そのクラスのインスタンス上の操作をするメソッドの定義を含みます。

class variable

(クラス変数) クラス上に定義され、クラスレベルで (つまり、クラスのインスタンス上ではなしに) 変更されることを目的としている変数です。

complex number

(複素数) よく知られている実数系を拡張したもので、すべての数は実部と虚部の和として表されます。虚数は虚数単位 (-1 の平方根) に実数を掛けたもので、一般に数学では i と書かれ、工学では j と書かれます。Python は複素数に組み込みで対応し、後者の表記を取っています。虚部は末尾に j をつけて書きま

す。例えば 3+1j です。math モジュールの複素数版を利用するには、cmath を使います。複素数の使用はかなり高度な数学の機能です。必要性を感じなければ、ほぼ間違いなく無視してしまってよいでしょう。

context manager
: An object which controls the environment seen in a with statement by defining __enter__() and __exit__() methods. See **PEP 343**.

context variable
: (コンテキスト変数) コンテキストに依存して異なる値を持つ変数。これは、ある変数の値が各々の実行スレッドで異なり得るスレッドローカルストレージに似ています。しかしコンテキスト変数では、1 つの実行スレッドにいくつかのコンテキストがあり得、コンテキスト変数の主な用途は並列な非同期タスクの変数の追跡です。contextvars を参照してください。

contiguous
: (隣接、連続) バッファが厳密に *C*-**連続** または *Fortran* **連続** である場合に、そのバッファは連続しているとみなせます。ゼロ次元バッファは C 連続であり Fortran 連続です。一次元の配列では、その要素は必ずメモリ上で隣接するように配置され、添字がゼロから始まり増えていく順序で並びます。多次元の C-連続な配列では、メモリアドレス順に要素を巡る際には最後の添え字が最初に変わるのに対し、Fortran 連続な配列では最初の添え字が最初に動きます。

コルーチン
: (コルーチン) コルーチンはサブルーチンのより一般的な形式です。サブルーチンには決められた地点から入り、別の決められた地点から出ます。コルーチンには多くの様々な地点から入る、出る、再開することができます。コルーチンは async def 文で実装できます。**PEP 492** を参照してください。

coroutine function
: (コルーチン関数) *coroutine* オブジェクトを返す関数です。コルーチン関数は async def 文で実装され、await、async for、および async with キーワードを持つことが出来ます。これらは **PEP 492** で導入されました。

CPython
: python.org で配布されている、Python プログラミング言語の標準的な実装です。”CPython” という単語は、この実装を Jython や IronPython といった他の実装と区別する必要が有る場合に利用されます。

decorator
: (デコレータ) 別の関数を返す関数で、通常、@wrapper 構文で関数変換として適用されます。デコレータの一般的な利用例は、classmethod() と staticmethod() です。

  デコレータの文法はシンタックスシュガーです。次の 2 つの関数定義は意味的に同じものです:

```
def f(arg):
    ...
f = staticmethod(f)

@staticmethod
```

(次のページに続く)

(前のページからの続き)

```
def f(arg):
    ...
```

同じ概念がクラスにも存在しますが、あまり使われません。デコレータについて詳しくは、関数定義 および クラス定義 のドキュメントを参照してください。

descriptor

Any object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

デスクリプタのメソッドに関しての詳細は、descriptors や Descriptor How To Guide を参照してください。

dictionary

An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

dictionary comprehension

(辞書内包表記) iterable 内の全てあるいは一部の要素を処理して、その結果からなる辞書を返すコンパクトな書き方です。`results = {n: n ** 2 for n in range(10)}` とすると、キー `n` を値 `n ** 2` に対応付ける辞書を生成します。comprehensions を参照してください。

dictionary view

(辞書ビュー) `dict.keys()`、`dict.values()`、`dict.items()` が返すオブジェクトです。辞書の項目の動的なビューを提供します。すなわち、辞書が変更されるとビューはそれを反映します。辞書ビューを強制的に完全なリストにするには `list(dictview)` を使用してください。dict-views を参照してください。

docstring

A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

duck-typing

あるオブジェクトが正しいインターフェースを持っているかを決定するのにオブジェクトの型を見ないプログラミングスタイルです。代わりに、単純にオブジェクトのメソッドや属性が呼ばれたり使われたりします。(「アヒルのように見えて、アヒルのように鳴けば、それはアヒルである。」) インターフェースを型より重視することで、上手くデザインされたコードは、ポリモーフィックな代替を許して柔軟性を向上させます。ダックタイピングは `type()` や `isinstance()` による判定を避けます。(ただし、ダックタイピングを

抽象基底クラス で補完することもできます。) その代わり、典型的に `hasattr()` 判定や *EAFP* プログラミングを利用します。

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

expression
: (式) 何かの値と評価される、一まとまりの構文 (a piece of syntax) です。言い換えると、式とはリテラル、名前、属性アクセス、演算子や関数呼び出しなど、値を返す式の要素の積み重ねです。他の多くの言語と違い、Python では言語の全ての構成要素が式というわけではありません。`while` のように、式としては使えない 文 もあります。代入も式ではなく文です。

extension module
: (拡張モジュール) C や C++ で書かれたモジュールで、Python の C API を利用して Python コアやユーザーコードとやりとりします。

f-string
: `'f'` や `'F'` が先頭に付いた文字列リテラルは "f-string" と呼ばれ、これは フォーマット済み文字列リテラル の短縮形の名称です。**PEP 498** も参照してください。

file object
: An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called *file-like objects* or *streams*.

  ファイルオブジェクトには実際には 3 種類あります: 生の バイナリーファイル、バッファされた バイナリーファイル、そして テキストファイル です。インターフェイスは `io` モジュールで定義されています。ファイルオブジェクトを作る標準的な方法は `open()` 関数を使うことです。

file-like object
: *file object* と同義です。

filesystem encoding and error handler
: Encoding and error handler used by Python to decode bytes from the operating system and encode Unicode to the operating system.

  ファイルシステムのエンコーディングでは、すべてが 128 バイト以下に正常にデコードされることが保証されなくてはなりません。ファイルシステムのエンコーディングでこれが保証されなかった場合は、API 関数が `UnicodeError` を送出することがあります。

The `sys.getfilesystemencoding()` and `sys.getfilesystemencodeerrors()` functions can be used to get the filesystem encoding and error handler.

The *filesystem encoding and error handler* are configured at Python startup by the `PyConfig_Read()` function: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

See also the *locale encoding*.

finder
: (ファインダ) インポートされているモジュールの *loader* の発見を試行するオブジェクトです。

  Python 3.3 以降では 2 種類のファインダがあります。`sys.meta_path` で使用される *meta path finder* と、`sys.path_hooks` で使用される *path entry finder* です。

  詳細については **PEP 302**、**PEP 420** および **PEP 451** を参照してください。

floor division
: (切り捨て除算) 一番近い整数に切り捨てる数学的除算。切り捨て除算演算子は `//` です。例えば、`11 // 4` は `2` になり、それとは対称に浮動小数点数の真の除算では `2.75` が 返ってきます。`(-11) // 4` は `-2.75` を **小さい方に** 丸める (訳注: 負の無限大への丸めを行う) ので `-3` になることに注意してください。**PEP 238** を参照してください。

関数
: (関数) 呼び出し側に値を返す一連の文のことです。関数には 0 以上の **実引数** を渡すことが出来ます。実体の実行時に引数を使用することが出来ます。**仮引数**、**メソッド**、function を参照してください。

function annotation
: (関数アノテーション) 関数のパラメータや返り値の *annotation* です。

  関数アノテーションは、通常は **型ヒント** のために使われます: 例えば、この関数は 2 つの `int` 型の引数を取ると期待され、また `int` 型の返り値を持つと期待されています。

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

  関数アノテーションの文法は function の節で解説されています。

  機能の説明がある *variable annotation*, **PEP 484**, を参照してください。また、アノテーションを利用するベストプラクティスとして annotations-howto も参照してください。

__future__
: A future statement, `from __future__ import <feature>`, directs the compiler to compile the current module using syntax or semantics that will become standard in a future release of Python. The `__future__` module documents the possible values of *feature*. By importing this module and evaluating its variables, you can see when a new feature was first added to the language and when it will (or did) become the default:

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

garbage collection

(ガベージコレクション) これ以降使われることのないメモリを解放する処理です。Python は、参照カウントと、循環参照を検出し破壊する循環ガベージコレクタを使ってガベージコレクションを行います。ガベージコレクタは `gc` モジュールを使って操作できます。

ジェネレータ

(ジェネレータ) *generator iterator* を返す関数です。通常の関数に似ていますが、`yield` 式を持つ点で異なります。`yield` 式は、for ループで使用できたり、`next()` 関数で値を 1 つずつ取り出したりできる、値の並びを生成するのに使用されます。

通常はジェネレータ関数を指しますが、文脈によっては **ジェネレータイテレータ** を指す場合があります。意図された意味が明らかでない場合、明瞭化のために完全な単語を使用します。

generator iterator

(ジェネレータイテレータ) *generator* 関数で生成されるオブジェクトです。

`yield` のたびに局所実行状態 (局所変数や未処理の try 文などを含む) を記憶して、処理は一時的に中断されます。**ジェネレータイテレータ** が再開されると、中断した位置を取得します (通常の関数が実行のたびに新しい状態から開始するのと対照的です)。

generator expression

An *expression* that returns an *iterator*. It looks like a normal expression followed by a `for` clause defining a loop variable, range, and an optional `if` clause. The combined expression generates values for an enclosing function:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

generic function

(ジェネリック関数) 異なる型に対し同じ操作をする関数群から構成される関数です。呼び出し時にどの実装を用いるかはディスパッチアルゴリズムにより決定されます。

single dispatch、`functools.singledispatch()` デコレータ、**PEP 443** を参照してください。

generic type

A *type* that can be parameterized; typically a container class such as `list` or `dict`. Used for *type hints* and *annotations*.

For more details, see generic alias types, **PEP 483**, **PEP 484**, **PEP 585**, and the `typing` module.

GIL

global interpreter lock を参照してください。

global interpreter lock

(グローバルインタプリタロック) *CPython* インタプリタが利用している、一度に Python の **バイトコード** を実行するスレッドは一つだけであることを保証する仕組みです。これにより (`dict` などの重要な組み込み型を含む) オブジェクトモデルが同時アクセスに対して暗黙的に安全になるので、CPython の実装がシンプルになります。インタプリタ全体をロックすることで、マルチプロセッサマシンが生じる並列化のコストと引き換えに、インタプリタを簡単にマルチスレッド化できるようになります。

ただし、標準あるいは外部のいくつかの拡張モジュールは、圧縮やハッシュ計算などの計算の重い処理をするときに GIL を解除するように設計されています。また、I/O 処理をする場合 GIL は常に解除されます。

過去に "自由なマルチスレッド化" したインタプリタ (供用されるデータを細かい粒度でロックする) が開発されましたが、一般的なシングルプロセッサの場合のパフォーマンスが悪かったので成功しませんでした。このパフォーマンスの問題を克服しようとすると、実装がより複雑になり保守コストが増加すると考えられています。

hash-based pyc

(ハッシュベース pyc ファイル) 正当性を判別するために、対応するソースファイルの最終更新時刻ではなくハッシュ値を使用するバイトコードのキャッシュファイルです。pyc-invalidation を参照してください。

hashable

An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equal must have the same hash value.

ハッシュ可能なオブジェクトは辞書のキーや集合のメンバーとして使えます。辞書や集合のデータ構造は内部でハッシュ値を使っているからです。

Python のイミュータブルな組み込みオブジェクトは、ほとんどがハッシュ可能です。(リストや辞書のような) ミュータブルなコンテナはハッシュ不可能です。(タプルや frozenset のような) イミュータブルなコンテナは、要素がハッシュ可能であるときのみハッシュ可能です。ユーザー定義のクラスのインスタンスであるようなオブジェクトはデフォルトでハッシュ可能です。それらは全て (自身を除いて) 比較結果は非等価であり、ハッシュ値は `id()` より得られます。

IDLE

Python の統合開発環境 (Integrated DeveLopment Environment) 及び学習環境 (Learning Environment) です。idle は Python の標準的な配布に同梱されている基本的な機能のエディタとインタプリタ環境です。

immortal

If an object is immortal, its reference count is never modified, and therefore it is never deallocated.

Built-in strings and singletons are immortal objects. For example, `True` and `None` singletons are immmortal.

See PEP 683 – Immortal Objects, Using a Fixed Refcount for more information.

immutable

(イミュータブル) 固定の値を持ったオブジェクトです。イミュータブルなオブジェクトには、数値、文字列、およびタプルなどがあります。これらのオブジェクトは値を変えられません。別の値を記憶させる際には、新たなオブジェクトを作成しなければなりません。イミュータブルなオブジェクトは、固定のハッシュ値が必要となる状況で重要な役割を果たします。辞書のキーがその例です。

import path
: *path based finder* が import するモジュールを検索する場所 (または *path entry*) のリスト。import 中、このリストは通常 `sys.path` から来ますが、サブパッケージの場合は親パッケージの `__path__` 属性からも来ます。

importing
: あるモジュールの Python コードが別のモジュールの Python コードで使えるようにする処理です。

importer
: モジュールを探してロードするオブジェクト。*finder* と *loader* のどちらでもあるオブジェクト。

interactive
: (対話的) Python には対話的インタプリタがあり、文や式をインタプリタのプロンプトに入力すると即座に実行されて結果を見ることができます。`python` と何も引数を与えずに実行してください。(コンピュータのメインメニューから Python の対話的インタプリタを起動できるかもしれません。) 対話的インタプリタは、新しいアイデアを試してみたり、モジュールやパッケージの中を覗いてみる (`help(x)` を覚えておいてください) のに非常に便利なツールです。

interpreted
: Python はインタプリタ形式の言語であり、コンパイラ言語の対極に位置します。(バイトコードコンパイラがあるために、この区別は曖昧ですが。) ここでのインタプリタ言語とは、ソースコードのファイルを、まず実行可能形式にしてから実行させるといった操作なしに、直接実行できることを意味します。インタプリタ形式の言語は通常、コンパイラ形式の言語よりも開発／デバッグのサイクルは短いものの、プログラムの実行は一般に遅いです。**対話的** も参照してください。

interpreter shutdown
: Python インタープリターはシャットダウンを要請された時に、モジュールやすべてのクリティカルな内部構造をなどの、すべての確保したリソースを段階的に開放する、特別なフェーズに入ります。このフェーズは **ガベージコレクタ** を複数回呼び出します。これによりユーザー定義のデストラクターや weakref コールバックが呼び出されることがあります。シャットダウンフェーズ中に実行されるコードは、それが依存するリソースがすでに機能しない (よくある例はライブラリーモジュールや warning 機構です) ために様々な例外に直面します。

  インタープリタがシャットダウンする主な理由は `__main__` モジュールや実行されていたスクリプトの実行が終了したことです。

iterable
: An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict`, *file objects*, and

objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements *sequence* semantics.

Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ...). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

iterator

An object representing a stream of data. Repeated calls to the iterator's `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `__next__()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

詳細な情報は typeiter にあります。

CPython 実装の詳細: CPython does not consistently apply the requirement that an iterator define `__iter__()`.

key function

(キー関数) キー関数、あるいは照合関数とは、ソートや順序比較のための値を返す呼び出し可能オブジェクト (callable) です。例えば、`locale.strxfrm()` をキー関数に使えば、ロケール依存のソートの慣習にのっとったソートキーを返します。

Python の多くのツールはキー関数を受け取り要素の並び順やグループ化を管理します。`min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()`, `itertools.groupby()` 等があります。

キー関数を作る方法はいくつかあります。例えば `str.lower()` メソッドを大文字小文字を区別しないソートを行うキー関数として使うことが出来ます。あるいは、`lambda r: (r[0], r[2])` のような `lambda` 式からキー関数を作ることができます。また、`operator.attrgetter()`, `operator.itemgetter()`, `operator.methodcaller()` の 3 つのキー関数コンストラクタがあります。キー関数の作り方と使い方の例は Sorting HOW TO を参照してください。

keyword argument 実

引数 を参照してください。

lambda

(ラムダ) 無名のインライン関数で、関数が呼び出されたときに評価される 1 つの **式** を含みます。ラムダ関数を作る構文は `lambda [parameters]: expression` です。

LBYL

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

マルチスレッド化された環境では、LBYL アプローチは ” 見る” 過程と ” 飛ぶ” 過程の競合状態を引き起こすリスクがあります。例えば、`if key in mapping: return mapping[key]` というコードは、判定の後、別のスレッドが探索の前に *mapping* から *key* を取り除くと失敗します。この問題は、ロックするか EAFP アプローチを使うことで解決できます。

list

A built-in Python *sequence*. Despite its name it is more akin to an array in other languages than to a linked list since access to elements is $O(1)$.

list comprehension

(リスト内包表記) シーケンス中の全てあるいは一部の要素を処理して、その結果からなるリストを返す、コンパクトな方法です。`result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` とすると、0 から 255 までの偶数を 16 進数表記 (0x..) した文字列からなるリストを生成します。`if` 節はオプションです。`if` 節がない場合、`range(256)` の全ての要素が処理されます。

loader

モジュールをロードするオブジェクト。`load_module()` という名前のメソッドを定義していなければなりません。ローダーは一般的に *finder* から返されます。詳細は **PEP 302** を、*abstract base class* については `importlib.abc.Loader` を参照してください。

ロケールエンコーディング

On Unix, it is the encoding of the LC_CTYPE locale. It can be set with `locale.setlocale(locale.LC_CTYPE, new_locale)`.

On Windows, it is the ANSI code page (ex: `"cp1252"`).

On Android and VxWorks, Python uses `"utf-8"` as the locale encoding.

`locale.getencoding()` can be used to get the locale encoding.

See also the *filesystem encoding and error handler*.

magic method

special method のくだけた同義語です。

mapping

(マッピング) 任意のキー探索をサポートしていて、`collections.abc.Mapping` か `collections.abc.`

MutableMapping の 抽象基底クラス で指定されたメソッドを実装しているコンテナオブジェクトです。例えば、dict, collections.defaultdict, collections.OrderedDict, collections.Counter などです。

meta path finder
: sys.meta_path を検索して得られた *finder*. meta path finder は *path entry finder* と関係はありますが、別物です。

  meta path finder が実装するメソッドについては importlib.abc.MetaPathFinder を参照してください。

metaclass
: (メタクラス) クラスのクラスです。クラス定義は、クラス名、クラスの辞書と、基底クラスのリストを作ります。メタクラスは、それら 3 つを引数として受け取り、クラスを作る責任を負います。ほとんどのオブジェクト指向言語は (訳注:メタクラスの) デフォルトの実装を提供しています。Python が特別なのはカスタムのメタクラスを作成できる点です。ほとんどのユーザーに取って、メタクラスは全く必要のないものです。しかし、一部の場面では、メタクラスは強力でエレガントな方法を提供します。たとえば属性アクセスのログを取ったり、スレッドセーフ性を追加したり、オブジェクトの生成を追跡したり、シングルトンを実装するなど、多くの場面で利用されます。

  詳細は metaclasses を参照してください。

メソッド
: (メソッド) クラス本体の中で定義された関数。そのクラスのインスタンスの属性として呼び出された場合、メソッドはインスタンスオブジェクトを第一 **引数** として受け取ります (この第一引数は通常 self と呼ばれます)。**関数** と **ネストされたスコープ** も参照してください。

method resolution order
: (メソッド解決順序) 探索中に基底クラスが構成要素を検索される順番です。2.3 以降の Python インタープリタが使用するアルゴリズムの詳細については The Python 2.3 Method Resolution Order を参照してください。

module
: (モジュール) Python コードの組織単位としてはたらくオブジェクトです。モジュールは任意の Python オブジェクトを含む名前空間を持ちます。モジュールは *importing* の処理によって Python に読み込まれます。

  パッケージ を参照してください。

module spec
: モジュールをロードするのに使われるインポート関連の情報を含む名前空間です。importlib.machinery.ModuleSpec のインスタンスです。

MRO
: *method resolution order* を参照してください。

mutable

(ミュータブル) ミュータブルなオブジェクトは、id() を変えることなく値を変更できます。イミュータブル) も参照してください。

named tuple ”

名前付きタプル” という用語は、タプルを継承していて、インデックスが付く要素に対し属性を使ってのアクセスもできる任意の型やクラスに応用されています。その型やクラスは他の機能も持っていることもあります。

time.localtime() や os.stat() の返り値を含むいくつかの組み込み型は名前付きタプルです。他の例は sys.float_info です:

```
>>> sys.float_info[1]                   # indexed access
1024
>>> sys.float_info.max_exp              # named field access
1024
>>> isinstance(sys.float_info, tuple)   # kind of tuple
True
```

Some named tuples are built-in types (such as the above examples). Alternatively, a named tuple can be created from a regular class definition that inherits from tuple and that defines named fields. Such a class can be written by hand, or it can be created by inheriting typing.NamedTuple, or with the factory function collections.namedtuple(). The latter techniques also add some extra methods that may not be found in hand-written or built-in named tuples.

namespace

(名前空間) 変数が格納される場所です。名前空間は辞書として実装されます。名前空間にはオブジェクトの (メソッドの) 入れ子になったものだけでなく、局所的なもの、大域的なもの、そして組み込みのものがあります。名前空間は名前の衝突を防ぐことによってモジュール性をサポートする。例えば関数 builtins.open と os.open() は名前空間で区別されています。また、どのモジュールが関数を実装しているか明示することによって名前空間は可読性と保守性を支援します。例えば、random.seed() や itertools.islice() と書くと、それぞれモジュール random や itertools で実装されていることが明らかです。

namespace package

(名前空間パッケージ) サブパッケージのコンテナとしてのみ提供される **PEP 420** で定義された *package* です。名前空間パッケージは物理的な表現を持たないことができ、__init__.py ファイルを持たないため、*regular package* とは異なります。

module を参照してください。

nested scope

(ネストされたスコープ) 外側で定義されている変数を参照する機能です。例えば、ある関数が別の関数の中で定義されている場合、内側の関数は外側の関数中の変数を参照できます。ネストされたスコープはデフォルトでは変数の参照だけができ、変数の代入はできないので注意してください。ローカル変数は、最も

内側のスコープで変数を読み書きします。同様に、グローバル変数を使うとグローバル名前空間の値を読み書きします。`nonlocal` で外側の変数に書き込めます。

new-style class
: Old name for the flavor of classes now used for all class objects. In earlier Python versions, only new-style classes could use Python's newer, versatile features like `__slots__`, descriptors, properties, `__getattribute__()`, class methods, and static methods.

object
: (オブジェクト) 状態 (属性や値) と定義された振る舞い (メソッド) をもつ全てのデータ。もしくは、全ての **新スタイルクラス** の究極の基底クラスのこと。

package
: (パッケージ) サブモジュールや再帰的にサブパッケージを含むことの出来る *module* のことです。専門的には、パッケージは `__path__` 属性を持つ Python オブジェクトです。

  regular package と *namespace package* を参照してください。

parameter
: (仮引数) 名前付の実体で **関数** (や **メソッド**) の定義において関数が受ける **実引数** を指定します。仮引数には 5 種類あります:

  - **位置またはキーワード**: **位置** であるいは **キーワード引数** として渡すことができる引数を指定します。これはたとえば以下の *foo* や *bar* のように、デフォルトの仮引数の種類です:

    ```
    def func(foo, bar=None): ...
    ```

  - 位置専用: 位置によってのみ与えられる引数を指定します。位置専用の引数は 関数定義の引数のリストの中でそれらの後ろに `/` を含めることで定義できます。例えば下記の *posonly1* と *posonly2* は位置専用引数になります:

    ```
    def func(posonly1, posonly2, /, positional_or_keyword): ...
    ```

  - **キーワード専用**: キーワードによってのみ与えられる引数を指定します。キーワード専用の引数を定義できる場所は、例えば以下の *kw_only1* や *kw_only2* のように、関数定義の仮引数リストに含めた可変長位置引数または裸の `*` の後です:

    ```
    def func(arg, *, kw_only1, kw_only2): ...
    ```

  - **可変長位置**: (他の仮引数で既に受けられた任意の位置引数に加えて) 任意の個数の位置引数が与えられることを指定します。このような仮引数は、以下の *args* のように仮引数名の前に `*` をつけることで定義できます:

    ```
    def func(*args, **kwargs): ...
    ```

- **可変長キーワード**: (他の仮引数で既に受けられた任意のキーワード引数に加えて) 任意の個数のキーワード引数が与えられることを指定します。このような仮引数は、上の例の *kwargs* のように仮引数名の前に ** をつけることで定義できます。

仮引数はオプションと必須の引数のどちらも指定でき、オプションの引数にはデフォルト値も指定できます。

仮引数 、FAQ の 実引数と仮引数の違いは何ですか? 、`inspect.Parameter` クラス、function セクション、**PEP 362** を参照してください。

path entry
: *path based finder* が import するモジュールを探す *import path* 上の 1 つの場所です。

path entry finder
: `sys.path_hooks` にある callable (つまり *path entry hook*) が返した *finder* です。与えられた *path entry* にあるモジュールを見つける方法を知っています。

パスエントリーファインダが実装するメソッドについては `importlib.abc.PathEntryFinder` を参照してください。

path entry hook
: A callable on the `sys.path_hooks` list which returns a *path entry finder* if it knows how to find modules on a specific *path entry*.

path based finder
: デフォルトの *meta path finder* の 1 つは、モジュールの *import path* を検索します。

path-like object
: (path-like オブジェクト) ファイルシステムパスを表します。path-like オブジェクトは、パスを表す `str` オブジェクトや `bytes` オブジェクト、または `os.PathLike` プロトコルを実装したオブジェクトのどれかです。`os.PathLike` プロトコルをサポートしているオブジェクトは `os.fspath()` を呼び出すことで `str` または `bytes` のファイルシステムパスに変換できます。`os.fsdecode()` と `os.fsencode()` はそれぞれ `str` あるいは `bytes` になるのを保証するのに使えます。**PEP 519** で導入されました。

PEP
: Python Enhancement Proposal。PEP は、Python コミュニティに対して情報を提供する、あるいは Python の新機能やその過程や環境について記述する設計文書です。PEP は、機能についての簡潔な技術的仕様と提案する機能の論拠 (理論) を伝えるべきです。

PEP は、新機能の提案にかかる、コミュニティによる問題提起の集積と Python になされる設計決断の文書化のための最上位の機構となることを意図しています。PEP の著者にはコミュニティ内の合意形成を行うこと、反対意見を文書化することの責務があります。

PEP 1 を参照してください。

portion

PEP 420 で定義されている、namespace package に属する、複数のファイルが (zip ファイルに格納されている場合もある) 1 つのディレクトリに格納されたもの。

位置引数 (positional argument) 実

引数 を参照してください。

provisional API

(暫定 API) 標準ライブラリの後方互換性保証から計画的に除外されたものです。そのようなインターフェースへの大きな変更は、暫定であるとされている間は期待されていませんが、コア開発者によって必要とみなされれば、後方非互換な変更 (インターフェースの削除まで含まれる) が行われえます。このような変更はむやみに行われるものではありません -- これは API を組み込む前には見落とされていた重大な欠陥が露呈したときにのみ行われます。

暫定 API についても、後方互換性のない変更は「最終手段」とみなされています。問題点が判明した場合でも後方互換な解決策を探すべきです。

このプロセスにより、標準ライブラリは問題となるデザインエラーに長い間閉じ込められることなく、時代を超えて進化を続けられます。詳細は PEP 411 を参照してください。

provisional package

provisional API を参照してください。

Python 3000

Python 3.x リリースラインのニックネームです。(Python 3 が遠い将来の話だった頃に作られた言葉です。) "Py3k" と略されることもあります。

Pythonic 他

の言語で一般的な考え方で書かれたコードではなく、Python の特に一般的なイディオムに従った考え方やコード片。例えば、Python の一般的なイディオムでは `for` 文を使ってイテラブルのすべての要素に渡ってループします。他の多くの言語にはこの仕組みはないので、Python に慣れていない人は代わりに数値のカウンターを使うかもしれません:

```
for i in range(len(food)):
    print(food[i])
```

これに対し、きれいな Pythonic な方法は:

```
for piece in food:
    print(piece)
```

qualified name

(修飾名) モジュールのグローバルスコープから、そのモジュールで定義されたクラス、関数、メソッドへの、" パス" を表すドット名表記です。PEP 3155 で定義されています。トップレベルの関数やクラスでは、修飾名はオブジェクトの名前と同じです:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

モジュールへの参照で使われると、**完全修飾名** *(fully qualified name)* はすべての親パッケージを含む全体のドット名表記、例えば `email.mime.text` を意味します:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count

The number of references to an object. When the reference count of an object drops to zero, it is deallocated. Some objects are *immortal* and have reference counts that are never modified, and therefore the objects are never deallocated. Reference counting is generally not visible to Python code, but it is a key element of the *CPython* implementation. Programmers can call the `sys.getrefcount()` function to return the reference count for a particular object.

regular package 伝

統的な、`__init__.py` ファイルを含むディレクトリとしての *package*。

namespace package を参照してください。

___slots___ ク

ラス内での宣言で、インスタンス属性の領域をあらかじめ定義しておき、インスタンス辞書を排除することで、メモリを節約します。これはよく使われるテクニックですが、正しく扱うには少しトリッキーなので、稀なケース、例えばメモリが死活問題となるアプリケーションでインスタンスが大量に存在する、といったときを除き、使わないのがベストです。

sequence

An *iterable* which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `__len__()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `bytes`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary *immutable* keys rather than integers.

The `collections.abc.Sequence` abstract base class defines a much richer interface that goes

beyond just `__getitem__()` and `__len__()`, adding `count()`, `index()`, `__contains__()`, and `__reversed__()`. Types that implement this expanded interface can be registered explicitly using `register()`. For more documentation on sequence methods generally, see Common Sequence Operations.

set comprehension
: (集合内包表記) iterable 内の全てあるいは一部の要素を処理して、その結果からなる集合を返すコンパクトな書き方です。`results = {c for c in 'abracadabra' if c not in 'abc'}` とすると、`{'r', 'd'}` という文字列の辞書を生成します。comprehensions を参照してください。

single dispatch
: *generic function* の一種で実装は一つの引数の型により選択されます。

slice
: (スライス) 一般に **シーケンス** の一部を含むオブジェクト。スライスは、添字表記 `[]` で与えられた複数の数の間にコロンを書くことで作られます。例えば、`variable_name[1:3:5]` です。角括弧 (添字) 記号は `slice` オブジェクトを内部で利用しています。

soft deprecated
: A soft deprecation can be used when using an API which should no longer be used to write new code, but it remains safe to continue using it in existing code. The API remains documented and tested, but will not be developed further (no enhancement).

  The main difference between a "soft" and a (regular) "hard" deprecation is that the soft deprecation does not imply scheduling the removal of the deprecated API.

  Another difference is that a soft deprecation does not issue a warning.

  See PEP 387: Soft Deprecation.

special method
: (特殊メソッド) ある型に特定の操作、例えば加算をするために Python から暗黙に呼び出されるメソッド。この種類のメソッドは、メソッド名の最初と最後にアンダースコア 2 つがついています。特殊メソッドについては specialnames で解説されています。

statement
: (文) 文はスイート (コードの" ブロック") に不可欠な要素です。文は **式** かキーワードから構成されるもののどちらかです。後者には `if`、`while`、`for` があります。

static type checker
: An external tool that reads Python code and analyzes it, looking for issues such as incorrect types. See also *type hints* and the `typing` module.

strong reference
: In Python's C API, a strong reference is a reference to an object which is owned by the code holding the

reference. The strong reference is taken by calling `Py_INCREF()` when the reference is created and released with `Py_DECREF()` when the reference is deleted.

The `Py_NewRef()` function can be used to create a strong reference to an object. Usually, the `Py_DECREF()` function must be called on the strong reference before exiting the scope of the strong reference, to avoid leaking one reference.

See also *borrowed reference*.

text encoding

A string in Python is a sequence of Unicode code points (in range `U+0000--U+10FFFF`). To store or transfer a string, it needs to be serialized as a sequence of bytes.

Serializing a string into a sequence of bytes is known as "encoding", and recreating the string from the sequence of bytes is known as "decoding".

There are a variety of different text serialization codecs, which are collectively referred to as "text encodings".

text file

(テキストファイル) `str` オブジェクトを読み書きできる *file object* です。しばしば、テキストファイルは実際にバイト指向のデータストリームにアクセスし、**テキストエンコーディング** を自動的に行います。テキストファイルの例は、`sys.stdin`, `sys.stdout`, `io.StringIO` インスタンスなどをテキストモード (`'r'` or `'w'`) で開いたファイルです。

bytes-like **オブジェクト** を読み書きできるファイルオブジェクトについては、**バイナリファイル** も参照してください。

triple-quoted string

(三重クォート文字列) 3 つの連続したクォート記号 (") かアポストロフィー (') で囲まれた文字列。通常の (一重) クォート文字列に比べて表現できる文字列に違いはありませんが、幾つかの理由で有用です。1 つか 2 つの連続したクォート記号をエスケープ無しに書くことができますし、行継続文字 (\) を使わなくても複数行にまたがることができるので、ドキュメンテーション文字列を書く時に特に便利です。

type

(型) Python オブジェクトの型はオブジェクトがどのようなものかを決めます。あらゆるオブジェクトは型を持っています。オブジェクトの型は `__class__` 属性でアクセスしたり、`type(obj)` で取得したり出来ます。

type alias

(型エイリアス) 型の別名で、型を識別子に代入して作成します。

型エイリアスは **型ヒント** を単純化するのに有用です。例えば:

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

これは次のようにより読みやすくできます:

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

機能の説明がある typing と **PEP 484** を参照してください。

type hint

(型ヒント) 変数、クラス属性、関数のパラメータや返り値の期待される型を指定する *annotation* です。

Type hints are optional and are not enforced by Python but they are useful to *static type checkers*. They can also aid IDEs with code completion and refactoring.

グローバル変数、クラス属性、関数で、ローカル変数でないものの型ヒントは typing.get_type_hints() で取得できます。

機能の説明がある typing と **PEP 484** を参照してください。

universal newlines テ

キストストリームの解釈法の一つで、以下のすべてを行末と認識します: Unix の行末規定 '\n'、Windows の規定 '\r\n'、古い Macintosh の規定 '\r'。利用法について詳しくは、**PEP 278** と **PEP 3116** 、さらに bytes.splitlines() も参照してください。

variable annotation

(変数アノテーション) 変数あるいはクラス属性の *annotation* 。

変数あるいはクラス属性に注釈を付けたときは、代入部分は任意です:

```
class C:
    field: 'annotation'
```

変数アノテーションは通常は **型ヒント** のために使われます: 例えば、この変数は int の値を取ることを期待されています:

```
count: int = 0
```

変数アノテーションの構文については annassign 節で解説しています。

機能の説明がある *function annotation*, **PEP 484** , **PEP 526** を参照してください。また、アノテーションを利用するベストプラクティスとして annotations-howto も参照してください。

virtual environment
: (仮想環境) 協調的に切り離された実行環境です。これにより Python ユーザとアプリケーションは同じシステム上で動いている他の Python アプリケーションの挙動に干渉することなく Python パッケージのインストールと更新を行うことができます。

  venv を参照してください。

virtual machine
: (仮想マシン) 完全にソフトウェアにより定義されたコンピュータ。Python の仮想マシンは、バイトコードコンパイラが出力した **バイトコード** を実行します。

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

付録

B

このドキュメントについて

このドキュメントは、Python のドキュメントを主要な目的として作られた ドキュメントプロセッサの Sphinx を利用して、reStructuredText 形式のソースから生成されました。

ドキュメントとそのツール群の開発は、Python 自身と同様に完全にボランティアの努力です。もしあなたが貢献したいなら、どのようにすればよいかについて reporting-bugs ページをご覧下さい。新しいボランティアはいつでも歓迎です!（訳注: 日本語訳の問題については、GitHub 上の Issue Tracker で報告をお願いします。）

多大な感謝を:

- Fred L. Drake, Jr., オリジナルの Python ドキュメントツールセットの作成者で、ドキュメントの多くを書きました。
- Docutils プロジェクト reStructuredText と Docutils ツールセットを作成しました。
- Fredrik Lundh の Alternative Python Reference プロジェクトから Sphinx は多くのアイデアを得ました。

B.1 Python ドキュメント 貢献者

多くの方々が Python 言語、Python 標準ライブラリ、そして Python ドキュメンテーションに貢献してくれています。ソース配布物の Misc/ACKS に、それら貢献してくれた人々を部分的にではありますがリストアップしてあります。

Python コミュニティからの情報提供と貢献がなければこの素晴らしいドキュメンテーションは生まれませんでした -- ありがとう!

付録

C

歴史とライセンス

C.1 Python の歴史

Python は 1990 年代の始め、オランダにある Stichting Mathematisch Centrum (CWI, https://www.cwi.nl/ 参照) で Guido van Rossum によって ABC と呼ばれる言語の後継言語として生み出されました。その後多くの人々が Python に貢献していますが、Guido は今日でも Python 製作者の先頭に立っています。

1995 年、Guido は米国ヴァージニア州レストンにある Corporation for National Reserch Initiatives (CNRI, https://www.cnri.reston.va.us/ 参照) で Python の開発に携わり、いくつかのバージョンをリリースしました。

2000 年 3 月、Guido と Python のコア開発チームは BeOpen.com に移り、BeOpen PythonLabs チームを結成しました。同年 10 月、PythonLabs チームは Digital Creations (現在の Zope Corporation, https://www.zope.org/ 参照) に移りました。そして 2001 年、Python に関する知的財産を保有するための非営利組織 Python Software Foundation (PSF、https://www.python.org/psf/ 参照) を立ち上げました。このとき Zope Corporation は PSF の賛助会員になりました。

Python のリリースは全てオープンソース (オープンソースの定義は https://opensource.org/ を参照してください) です。歴史的にみて、ごく一部を除くほとんどの Python リリースは GPL 互換になっています; 各リリースについては下表にまとめてあります。

リリース	ベース	西暦年	権利	GPL 互換
0.9.0 - 1.2	n/a	1991-1995	CWI	yes
1.3 - 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2 以降	2.1.1	2001-現在	PSF	yes

注釈: 「GPL 互換」という表現は、Python が GPL で配布されているという意味ではありません。Python のライセンスは全て、GPL と違い、変更したバージョンを配布する際に変更をオープンソースにしなくてもかまいません。GPL 互換のライセンスの下では、GPL でリリースされている他のソフトウェアと Python を組み合わせられますが、それ以外のライセンスではそうではありません。

Guido の指示の下、これらのリリースを可能にしてくださった多くのボランティアのみなさんに感謝します。

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

Python software and documentation are licensed under the *PSF License Agreement*.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Agreement and the *Zero-Clause BSD license*.

Some software incorporated into Python is under different licenses. The licenses are listed with code falling under that license. See *Licenses and Acknowledgements for Incorporated Software* for an incomplete list of these licenses.

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.13.0a5

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 3.13.0a5 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 3.13.0a5 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2024 Python Software Foundation; All Rights Reserved" are retained in Python 3.13.0a5 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 3.13.0a5 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 3.13.0a5.

4. PSF is making Python 3.13.0a5 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 3.13.0a5 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.13.0a5 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.13.0a5, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 3.13.0a5, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

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

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
```

(次のページに続く)

(前のページからの続き)

```
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

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

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

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

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0a5 DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
```

(次のページに続く)

(前のページからの続き)

```
INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

C.3.1 Mersenne Twister

The `_random` C extension underlying the `random` module includes code based on a download from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
```

(次のページに続く)

(前のページからの続き)

```
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 ソケット

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, https://www.wide.ad.jp/.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Asynchronous socket services

The `test.support.asynchat` and `test.support.asyncore` modules contain the following notice:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

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

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

C.3.4 Cookie management

The `http.cookies` module contains the following notice:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

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

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
```

(次のページに続く)

(前のページからの続き)

```
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 Execution tracing

The `trace` module contains the following notice:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 UUencode and UUdecode functions

The `uu` codec contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
```

(次のページに続く)

(前のページからの続き)

```
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 XML Remote Procedure Calls

The `xmlrpc.client` module contains the following notice:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

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

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
```

(次のページに続く)

(前のページからの続き)

```
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

The `test.test_epoll` module contains the following notice:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Select kqueue

`select` モジュールは kqueue インターフェースについての次の告知を含んでいます:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
```

(次のページに続く)

(前のページからの続き)

```
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

The file `Python/pyhash.c` contains Marek Majkowski' implementation of Dan Bernstein's SipHash24 algorithm. It contains the following note:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod と dtoa

The file `Python/dtoa.c`, which supplies C functions dtoa and strtod for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.12 OpenSSL

The modules `hashlib`, `posix` and `ssl` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here. For the OpenSSL 3.0 release, and later releases derived from that, the Apache License v2 applies:

```
                                 Apache License
                           Version 2.0, January 2004
                        https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
```

(次のページに続く)

(前のページからの続き)

```
the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all
other entities that control, are controlled by, or are under common
control with that entity. For the purposes of this definition,
"control" means (i) the power, direct or indirect, to cause the
direction or management of such entity, whether by contract or
otherwise, or (ii) ownership of fifty percent (50%) or more of the
outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity
exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications,
including but not limited to software source code, documentation
source, and configuration files.

"Object" form shall mean any form resulting from mechanical
transformation or translation of a Source form, including but
not limited to compiled object code, generated documentation,
and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or
Object form, made available under the License, as indicated by a
copyright notice that is included in or attached to the work
(an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object
form, that is based on (or derived from) the Work and for which the
editorial revisions, annotations, elaborations, or other modifications
represent, as a whole, an original work of authorship. For the purposes
of this License, Derivative Works shall not include works that remain
separable from, or merely link (or bind by name) to the interfaces of,
the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including
the original version of the Work and any modifications or additions
to that Work or Derivative Works thereof, that is intentionally
submitted to Licensor for inclusion in the Work by the copyright owner
or by an individual or Legal Entity authorized to submit on behalf of
the copyright owner. For the purposes of this definition, "submitted"
means any form of electronic, verbal, or written communication sent
to the Licensor or its representatives, including but not limited to
communication on electronic mailing lists, source code control systems,
and issue tracking systems that are managed by, or on behalf of, the
Licensor for the purpose of discussing and improving the Work, but
excluding communication that is conspicuously marked or otherwise
```

(次のページに続く)

(前のページからの続き)

```
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
```

(次のページに続く)

(前のページからの続き)

```
      distribution, then any Derivative Works that You distribute must
      include a readable copy of the attribution notices contained
      within such NOTICE file, excluding those notices that do not
      pertain to any part of the Derivative Works, in at least one
      of the following places: within a NOTICE text file distributed
      as part of the Derivative Works; within the Source form or
      documentation, if provided along with the Derivative Works; or,
      within a display generated by the Derivative Works, if and
      wherever such third-party notices normally appear. The contents
      of the NOTICE file are for informational purposes only and
      do not modify the License. You may add Your own attribution
      notices within Derivative Works that You distribute, alongside
      or as an addendum to the NOTICE text from the Work, provided
      that such additional attribution notices cannot be construed
      as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
```

(次のページに続く)

(前のページからの続き)

```
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

C.3.13 expat

The `pyexpat` extension is built using an included copy of the expat sources unless the build is configured `--with-system-expat`:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
```

(次のページに続く)

(前のページからの続き)

```
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

The `_ctypes` C extension underlying the `ctypes` module is built using an included copy of the libffi sources unless the build is configured `--with-system-libffi`:

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 zlib

The `zlib` extension is built using an included copy of the zlib sources if the zlib version found on the system is too old to be used for the build:

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.
```

(次のページに続く)

(前のページからの続き)

```
Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

The implementation of the hash table used by the `tracemalloc` で使用しているハッシュテーブルの実装は、cfuhash プロジェクトのものに基づきます:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
```

(次のページに続く)

(前のページからの続き)

```
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

The `_decimal` C extension underlying the `decimal` module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec`:

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.18 W3C C14N test suite

The C14N 2.0 test suite in the `test` package (`Lib/test/xmltestdata/c14n-20/`) was retrieved from the W3C website at https://www.w3.org/TR/xml-c14n2-testcases/ and is distributed under the 3-clause BSD license:

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.19 mimalloc

MIT License

Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the ”Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions

of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

C.3.20 asyncio

Parts of the `asyncio` module are incorporated from uvloop 0.16, which is distributed under the MIT license:

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.21 Global Unbounded Sequences (GUS)

The file `Python/qsbr.c` is adapted from FreeBSD's "Global Unbounded Sequences" safe memory reclamation scheme in subr_smr.c. The file is distributed under the 2-Clause BSD License:

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
```

(次のページに続く)

(前のページからの続き)

```
are met:
1. Redistributions of source code must retain the above copyright
   notice unmodified, this list of conditions, and the following
   disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

付録

D

COPYRIGHT

ライセンスおよび許諾に関する完全な情報は、歴史とライセンス を参照してください。

索引

アルファベット以外

A

B

C

D

E

F

G

H

I

J