Python Tutorial

發🄵 *3.7.14*

Guido van Rossum
and the Python development team

9 月 09, 2022

Python Software Foundation
Email: docs@python.org

Contents

Python 是一種易學、功能🄵大的程式語言。它有高效能的高階資料結構，也有簡單但有效的方法去實現物件導向程式設計。Python 優雅的語法和動態型🄵，結合其直譯特性，使它成🄵🄵多領域和大多數平臺上，撰寫🄵本和快速開發應用程式的理想語言。

使用者可以自由且免費地從 Python 官網上 (https://www.python.org/) 取得各大平台上用的 Python 直譯器和標准函式庫，下載其源碼或二進位形式執行檔，同時，也可以將其自由地散🄵。另外，Python 官網也提供了許多自由且免費的第三方 Python 模組、程式與工具、以及額外🄵明文件，有興趣的使用者，可在官網上找到相關的發行版本與連結網址。

使用 C 或 C++（或其他可被 C 呼叫的程式語言），可以很容易在 Python 直譯器🄵新增功能函式及資料型🄵。同時，對可讓使用者自🄵功能的應用程式來🄵，Python 也適合作🄵其擴充用界面語言 (extension language)。

這份教學將簡介 Python 語言與系統的基本概念及功能。除了🄵讀之外、實際用 Python 直譯器寫程式跑範例，將有助於學習。但如果只用讀的，也是可行的學習方式，因🄵所有範例的🄵容皆獨立且完整。

若想了解 Python 標准物件和模組的描述，請參🄵 library-index。在 reference-index 中，您可以學到 Python 語言更正規的定義。想用 C 或 C++ 寫延伸套件 (extensions) 的讀者，請🄵讀 extending-index 和 c-api-index 。此外，市面上也能找到更深入的 Python 學習書。

這份教學中，我們不會介紹每一個功能，甚至，也不打算介紹完每一個常用功能。取而代之，我們的重心將放在介紹 Python 中最值得一提的那些功能，幫助您了解 Python 語言的特色與風格。讀完教學後，您將有能力🄵讀和撰寫 Python 模組與程式，也做好進一步學習 library-index 中各類型的 Python 函式庫模組的准備。

Glossary 頁面也值得細讀。

CHAPTER 1

淺嘗滋味

如果你經常在電腦上工作，最終總能發現有些工作你會想要自動化。舉例來[F]，你會想在很多文字檔案[F]做相同的搜尋取代，或者是用個[F]雜的規則重新命名或整理一群照片。也有可能你想寫個自己的小資料庫，一個專門的 GUI 應用程式，或一個小[F]戲。

如果你是一個職業軟體開發者，你可能要操作數個 C/C++/Java 程式庫，[F]覺得平常寫程式碼、編譯、測試、再編譯的流程太慢；有可能你正[F]了一個程式庫撰寫一套測試集，但發現寫測試單調乏味；也有可能你正在開發一個能使用某一語言擴充的程式，但[F]不想要[F]了這程式特[F]設計一個全新的擴充語言。

在上述的例子中，Python 正是你合適的語言。

也許你可以[F]了某些任務而寫個 Unix shell [F]本或者 Windows 批次檔來處理，但 shell [F]本最適合於搬動檔案或更動文字[F]容，而不適於圖形應用程式或[F]戲。你可以[F]此寫個 C/C++/Java 程式，但僅僅是完成個草稿也需要很長的開發時間。相較而言，Python 更易於使用，[F]能在 Windows、Mac OSX、Unix 作業系統上執行，且能更快速地幫助你完成工作。

Python 即便易用也是個貨真價實的程式語言。它提供比 shell [F]本、批次檔更多樣的程式架構與更多的支援。另一方面，Python 提供比 C 更豐富的錯語檢查。相較於 C ，Python 作[F]一個「非常高階的程式語言」，它[F]建了高階的資料型[F]如彈性的數列與字典。因[F]這些多用途的資料型[F]，Python 適用解[F]比 Awk （甚至是 Perl）能處理的更多問題上。至少在許多事情中，使用 Python 處理起來跟其他語言是同樣容易的。

Python 允許你把程式切割成許多模組 (module) [F]將他們重覆運用至其他 Python 程式中。Python 自帶了一個很大集合的標準模組，它們能做[F]你程式的基礎 --- 或把它們當作一開始學寫 Python 程式的範例。有些模組提供了如檔案 I/O、系統呼叫、socket 的功能，甚至提供了 Tk 等圖形介面工具庫 (GUI toolkit) 的介面。

Python 是個直譯式語言，因[F]不需要編譯與連結，能[F]你在開發過程中省下可觀的時間。它的直譯器能互動地使用，因此能很方便地實驗每個語言的功能、寫些用完即[F]的程式、幫助測試一些從細部開始開發的函式。它也是個好用的計算機。

Python 讓程式寫得精簡[F]易讀。用 Python 實作的程式長度往往遠比用 C、C++、Java 實作的短。這有以下幾個原因：

- Python 高階的資料型[F]能在一陳述句 (statement) 中表達很[F]雜的操作；
- 陳述句的段落以縮排[F]區格而非括號；
- 不需要宣告變數和引數；

Python 是可擴充的：如果你會寫 C 程式，那[F]要加個新的[F]建函式或模組到直譯器中是很容易的。無論是[F]了用最快速的執行速度完成一些關鍵的操作，或是讓 Python 連結到一些僅以二進元形式 (binary

form) 釋出的程式庫（例如特定供應商的繪圖程式庫）。如果你想更多這樣的結合，你其實也可以把 Python 直譯器連結到用 C 寫的應用程式，🄵在該應用程式中使用 Python 寫擴充或者作🄵下達指令的語言。

順帶一提，這個語言是以 BBC 的戲劇《Monty Python's Flying Circus》命名，與爬蟲類完全🄵有關🄵。在🄵明文件中引用他們的喜劇不但🄵問題，這甚至是個被鼓勵的行🄵!

如果你現在已經躍躍欲試，你會想了解 Python 更多細節，而學習語言的最好方式就是直接使用它。接下來這個教學就將帶領你，一邊🄵讀，一邊將所學用在 Python 直譯器中玩耍。

在下個章節中，將會解🄵如何使用該直譯器。這也許只是個普通的資訊，但你必須試著操作接下來呈現的範例。

接下來的教學🄵，將會透過許多範例介紹 Python 語言與其系統的諸多特色。一開始是簡單的表示句 (expression)、陳述句 (statement) 和資料型🄵 (data type)；接著是函式 (function) 與模組 (module)；最後會接觸一些較進階的主題如例外狀🄵 (exception) 與使用者自定義類🄵 (class)。

CHAPTER 2

使用 Python 直譯器

2.1 □動直譯器

The Python interpreter is usually installed as `/usr/local/bin/python3.7` 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:

```
python3.7
```

能□動 Python[1]。因□直譯器存放的目□是個安裝選項，其他的目□也是有可能的；請洽談在地的 Python 達人或者系統管理員。（例如：`/usr/local/python` 是個很常見的另類存放路徑。）

On Windows machines where you have installed Python from the Microsoft Store, the `python3.7` 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.

在主提示符輸入一個 end-of-file 字元（在 Unix 上□ `Control-D`；在 Windows 上□ `Control-Z`）會使得直譯器以零退出狀□ (zero exit status) 離開。如果上述的做法□有效，也可以輸入指令 `quit()` 離開直譯器。

The interpreter's line-editing features include interactive editing, history substitution and code completion on systems that support the GNU Readline library. Perhaps the quickest check to see whether command line editing is supported is typing `Control-P` to the first Python prompt you get. If it beeps, you have command line editing; see Appendix *Interactive Input Editing and History Substitution* for an introduction to the keys. If nothing appears to happen, or if `^P` is echoed, command line editing isn't available; you'll only be able to use backspace to remove characters from the current line.

這個直譯器使用起來像是 Unix shell：如果它被呼叫時連結至一個 tty 裝置，它會互動式地讀取□執行指令；如果被呼叫時給定檔名□引數或者使用 stdin 傳入檔案□容，它會將這個檔案視□□**本**來□讀。

另一個起動直譯器的方式□ `python -c command [arg] ...`，它會執行在 *command* □的指令（們），行□如同 shell 的 `-c` 選項。因□ Python 的指令包含空白等 shell 用到的特殊字元，通常建議用單引號把 *command* 包起來。

有些 Python 模組使用上如□本般一樣方便。透過 `python -m module [arg] ...` 可以執行 *module* 模組的原始碼，就如同直接傳入那個模組的完整路徑一樣的行□。

當要執行一個□本檔時，有時候會希望在□本結束時進入互動模式。此時可在執行□本的指令加入 `-i`。

[1] 在 Unix 中，Python 3.x 直譯器預設安裝不會以 `python` 作□執行檔名稱，以避免與現有的 Python 2.x 執行檔名稱衝突。

所有指令可用的參數都詳記在 using-on-general。

2.1.1 傳遞引數

當直擇器收到🄵本的名稱和額外的引數後，他們會轉🄵🄵由字串所組成的 list（串列）🄵指派給 `sys` 模組的 `argv` 變數。你可以執行 `import sys` 取得這個串列。這個串列的長度至少🄴一；當🄵有給任何🄵本名稱和引數時，`sys.argv[0]` 🄵空字串。當🄵本名🄵 `'-'`（指標准輸入）時，`sys.argv[0]` 🄵 `'-'`。當使用 `-c` *command* 時，`sys.argv[0]` 🄵 `'-c'`。當使用 `-m` *module* 時，`sys.argv[0]` 🄵該模組存在的完整路徑。其余非 `-c` *command* 或 `-m` *module* 的選項不會被 Python 直譯器吸收掉，而是留在 `sys.argv` 變數中給後續的 command 或 module 使用。

2.1.2 互動模式

When commands are read from a tty, the interpreter is said to be in *interactive mode*. In this mode it prompts for the next command with the *primary prompt*, usually three greater-than signs (>>>); for continuation lines it prompts with the *secondary prompt*, by default three dots (`...`). The interpreter prints a welcome message stating its version number and a copyright notice before printing the first prompt:

```
$ python3.7
Python 3.7 (default, Sep 16 2015, 09:25:04)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

接續多行的情🄵出現在需要多行才能建立完整指令時。舉例來🄵，像是 `if` 🄵述：

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

更多有關互動模式的使用，請見*互動模式*。

2.2 直譯器與它的環境

2.2.1 原始碼的字元編碼 (encoding)

預設 Python 原始碼檔案的字元編碼使用 UTF-8。在這個編碼中，世界上多數語言的文字可以同時被使用在字串🄵容、識🄵名 (identifier) 及🄵解中 --- 雖然在標准函式庫中只使用 ASCII 字元作🄵識🄵名，這也是個任何 portable 程式碼需遵守的慣例。如果要正確地顯示所有字元，您的編輯器需要能🄵認識檔案🄵 UTF-8，🄵且需要能顯示檔案中所有字元的字型。

To declare an encoding other than the default one, a special comment line should be added as the *first* line of the file. The syntax is as follows:

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

where *encoding* is one of the valid `codecs` supported by Python.

For example, to declare that Windows-1252 encoding is to be used, the first line of your source code file should be:

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

One exception to the *first line* rule is when the source code starts with a *UNIX "shebang" line*. In this case, the encoding declaration should be added as the second line of the file. For example:

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

🄵解

CHAPTER 3

一個非正式的 Python 簡介

在下面的例子中，輸入與輸出的區F在於有無提示符（prompt，»> 和...）：如果要重做範例，你必須在提示符出現的時候，輸入提示符後方的所有F容；那些非提示符開始的文字行是直譯器的輸出。注意到在範例中，若出現單行只有次提示符時，代表該行你必須直接F行；這被使用在多行指令結束輸入時。

在本手F中的許多範例中，即便他們F互動式地輸入，仍然包含F解。Python 中的F解 (comments) 由 hash 字元 # 開始一直到該行結束。F解可以從該行之首、空白後、或程式碼之後開始，但不會出現在字串之中。hash 字元在字串之中時仍視F一 hash 字元。因FF解只是用來F明程式而不會被 Python 解讀，在練習範例時不一定要輸入。

一些範例如下：

```
# 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 把 Python 當作計算機使用

讓我們來試試一些簡單的 Python 指令。F動直譯器F等待第一個主提示符 >>> 出現。（應該不會等太久）

3.1.1 數字 (Number)

直譯器如同一台簡單的計算機：你可以輸入一個 expression（運算式），它會寫出該式的值。Expression 的語法很使用：運算子 +、−、* 和 / 的行F如同大多數的程式語言（例如：Pascal 或 C）；括號（）可以用來分群。例如：

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

整數數字（即 2、4、20）🄵 int 型態，數字有小數點部份的（即 5.0、1.6）🄵 float 型態。我們將在之後的教學中看到更多數字相關的型態。

除法 (/) 永遠回傳一個 float。如果要做*floor division* 🄵拿到整數的結果（即去除所有小數點的部份），你可以使用 // 運算子；計算余數可以使用 %：

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

在 Python 中，計算🄵次 (powers) 可以使用 ** 運算子[1]：

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

等於符號 (=) 可以用於🄵變數賦值。賦值完之後，在下個指示符前🄵不會顯示任何結果：

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

如果一個變數未被「定義 (defined)」（即變數未被賦值），試著使用它時會出現一個錯誤：

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

浮點數的運算有完善的支援，運算子 (operator) 遇上混合的運算元 (operand) 時會把整數的運算元轉🄵🄵浮點數：

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

在互動式模式中，最後一個印出的運算式的結果會被指派至變數 _ 中。這表示當你把 Python 當作桌上計算機使用者，要接續計算變得容易許多：

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

這個變數應該被使用者視🄵只能讀取。不應該明確地🄵它賦值 --- 你可以創一個獨立但名稱相同的本地變數來覆蓋掉預設變數和它的神奇行🄵。

除了 int 和 float，Python 還支援了其他的數字型態，包含 Decimal 和 Fraction。Python 亦🄵建支援🄵數 (complex numbers)，🄵使用 j 和 J 後綴來指定🄵數的部份（即 3+5j）。

[1] 因🄵 ** 擁有較 - 高的優先次序，-3**2 會被解釋🄵 -(3**2) 🄵得到 -9。如果要避免這樣的優先順序以得到 9，你可以使用 (-3)**2。

3.1.2 字串 (String)

除了數字之外，Python 也可以操作字串，而表達字串有數種方式。它們可以用包含在單引號 ('...') 或雙引號 ("...") 之中，兩者會得到相同的結果[2]。使用 \ 跳F出現於字串中的引號：

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

在互動式的直譯器中，輸出的字串會被引號包圍且特殊符號會使用反斜F (\) 跳F。雖然這有時會讓它看起來跟輸入的字串不相同（包圍用的引號可能會改變），輸入和輸出兩字串實F相同。一般來F，字串包含單引號而F有雙引號時，會使用雙引號包圍字串。函式 print() 會F生更易讀的輸出，它會去除掉包圍的引號，F且直接印出被跳F的字元和特殊字元：

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

如果你不希望字元前出現 \ 就被當成特殊字元時，可以改使用 *raw string*，在第一個包圍引號前加上 r：

```
>>> 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
```

字串值可以跨越數行。其中一方式是使用三個重覆引號："""..."""或'''...'''。此時F行會被自動加入字串值中，但也可以在F行前加入 \ 來取消這個行F。在以下的例子中：

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

會F生以下的輸出（注意第一個F行FF有被包含進字串值中）：

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

字串可以使用 + 運算子連接 (concatenate)，F用 * 重覆該字串的F容：

[2] 不像其他語言，特殊符號如 \n 在單 ('...') 和雙 ("...") 括號中有相同的意思。兩種刮號的唯一差F，在於使用單刮號時，不需要跳F (escape) "（但需要跳F \'），反之亦同。

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

兩個以上相鄰的字串值（*string literal*，即被引號包圍的字串）會被自動連接起來：

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

當你想要分段一個非常長的字串時，兩相鄰字串值自動連接的特性十分有用：

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

但這特性只限於兩相鄰的字串值間，而非兩相鄰變數或表達式：

```
>>> 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
```

如果要連接變數們或一個變數與一個字串值，使用 +：

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

字串可以被「索引 *indexed*」(下標，即 subscripted)，第一個字元的索引值🄵 0。🄵有獨立表示字元的型🄵；一個字元就是一個大小🄵 1 的字串：

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

索引值可以是負的，此時改成從右開始計數：

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

注意到因🄵 -0 等同於 0，負的索引值由 -1 開始。

除了索引外，字串亦支援「切片 *slicing*」。索引用來拿到單獨的字元，而切片則可以讓你拿到子字串 (substring)：

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

注意到起點永遠被包含，而結尾永遠不被包含。這確保了 `s[:i] + s[i:]` 永遠等於 `s`：

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

切片索引 (slice indices) 有很常用的預設值，省略起點索引值時預設🄵 0，而省略第二個索引值時預設整個字串被包含在 slice 中：

```
>>> 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'
```

這🄵有個簡單記住 slice 是如何運作的方式。想像 slice 的索引值指著字元們之間，其中第一個字元的左側邊緣由 0 計數。則 *n* 個字元的字串中最後一個字元的右側邊緣會有索引值 *n*，例如：

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

第一行數字給定字串索引值🄵 0...6 的位置；第二行則標示了負索引值的位置。由 *i* 至 *j* 的 slice 包含了標示 *i* 和 *j* 邊緣間的所有字元。

對非負數的索引值而言，一個 slice 的長度等於其索引值之差，如果索引值落在字串邊界🄵。例如，`word[1:3]` 的長度是 2。

嘗試使用一個過大的索引值會造成錯誤：

```
>>> 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
```

然而，超出範圍的索引值在 slice 中會被妥善的處理：

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

Python 字串無法被改變 --- 它們是*immutable*。因此，嘗試對字串中某個索引位置賦值會🄵生錯誤：

```
>>> 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
```

如果你需要一個不一樣的字串，你必須建立一個新的：

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

🄵建的函式 `len()` 回傳一個字串的長度：

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

也參考：

textseq 字串是 *sequence* 型🄵的範例之一，🄵支援該型🄵常用的操作。

string-methods 字串支援非常多種基本轉🄵和搜尋的方法。

f-strings 包含有表示式的字串值。

formatstrings 關於透過 `str.format()` 字串格式化 (string formatting) 的資訊。

old-string-formatting 在字串🄵 `%` 的左運算元時，將觸發舊的字串格式化操作，更多的細節在本連結中介紹。

3.1.3 List（串列）

Python 理解數種🄵合型資料型🄵，用來組合不同的數值。當中最多樣變化的型🄵🄵 *list*，可以寫成一系列以逗號分隔的數值（稱之元素，即 item），包含在方括號之中。List 可以包合不同型🄵的元素，但通常這些元素會有相同的型🄵：

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

Like strings (and all other built-in *sequence* types), lists can be indexed and sliced:

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

所有 slice 操作都會回傳一個新的 list 包含要求的元素。這意謂著以下這個 slice 🄵🄵了原本 list（淺🄵🄵，即 shallow copy）：

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

List 對支援如接合 (concatenation) 等操作：

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

不同於字串是*immutable*，list 是*mutable* 型🄵，即改變 list 的🄵容是可能的：

```
>>> 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]
```

你也可以在 list 的最後加入新元素，透過使用 `append()` 方法 (method)（我們稍後會看到更多方法的🄵明）：

```
>>> 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]
```

也可以對 slice 賦值，這能改變 list 的大小，甚至是清空一個 list：

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

🄵建的函式 `len()` 亦可以作用在 list 上：

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

也可以嵌套多層 list （建立 list 包含其他 list），例如：

```
>>> 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 初探程式設計的前幾步

當然，我們可以用 Python 來處理比 2 加 2 更🄵雜的工作。例如，我們可以印出費氏數列的首幾項序列：

```
>>> # 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
3
5
8
```

這例子引入了許多新的特性。

- 第一行出現了多重賦值：變數 a 與 b 同時得到了新的值 0 與 1。在最後一行同樣的賦值再被使用了一次，示範了等號的右項運算 (expression) 會先被計算 (evaluate)，賦值再發生。右項的運算式由左至右依序被計算。
- while 𝔽圈只要它的條件𝔽真（此範例：a < 10），將會一直重覆執行。在 Python 中如同 C 語言，任何非零的整數值𝔽真 (true)；零𝔽假 (false)。條件可以是字串、list、甚至是任何序列型𝔽；任何非零長度的序列𝔽真，空的序列即𝔽假。本例子使用的條件是個簡單的比較。標準的比較運算子 (comparison operators) 使用如同 C 語言一樣的符號：<（小於）、>（大於）、==（等於）、<=（小於等於）、>=（大於等於）以及 !=（不等於）。
- 𝔽圈的主體會縮排：縮排在 Python 中用來關連一群陳述式。在互動式提示符中，你必須在𝔽圈𝔽的每一行一開始鍵入 tab 或者（數個）空白來維持縮排。實務上，你會先在文字編輯器中准備好比較𝔽雜的輸入；多數編輯器都有自動縮排的功能。當一個𝔽合陳述式以互動地方式輸入，必須在結束時多加一行空行來代表結束（因𝔽語法解析器無法判斷你何時輸入𝔽合陳述的最後一行）。注意在一個縮排段落𝔽的縮排方式與數量必須維持一致。
- print() 函式印出它接收到引數（們）的值。不同於先前僅我們寫下想要的運算（像是先前的計算機範例），它可以處理𝔽數個引數、浮點數數值和字串。印出的字串將不帶有引號，𝔽且不同項目間會插入一個空白，因此可以讓你容易格式化輸出，例如：

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

關鍵字引數 *end* 可以被用來避免額外的𝔽行符加入到輸出中，或者以不同的字串結束輸出：

```
>>> 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,
```

𝔽解

CHAPTER 4

深入了解流程控制

Besides the `while` statement just introduced, Python uses the usual flow control statements known from other languages, with some twists.

4.1 `if` Statements

或許最常見的陳述式種類就是 `if` 了。舉例來🄴:

```
>>> 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
```

There can be zero or more `elif` parts, and the `else` part is optional. The keyword 'elif' is short for 'else if', and is useful to avoid excessive indentation. An `if` ... `elif` ... `elif` ... sequence is a substitute for the `switch` or `case` statements found in other languages.

4.2 `for` Statements

The `for` statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python's `for` statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example (no pun intended):

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

如果你在🄵圈中需要修改一個你正在🄵代的序列（例如重🄵一些選擇的元素），那🄵會建議你先建立一個序列的拷貝。🄵代序列🄵不暗示建立新的拷貝。此時 slice 語法就讓這件事十分容易完成：

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

在 `for w in words:` 的情🄵，這個例子會試著重覆不斷地插入 `defenestrate`，🄵生一個無限長的 list。

4.3 `range()` 函式

如果你需要🄵代一個數列的話，使用🄵建 `range()` 函式就很方便。它可以生成一等差級數：

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

給定的結束值永遠不會出現在生成的序列中；`range(10)` 生成的 10 個數值，即對應存取一個長度🄵 10 的序列🄵每一個元素的索引值。也可以讓 range 從其他數值計數，或者給定不同的級距（甚至🄵負；有時稱之🄵 step）：

```
range(5, 10)
   5, 6, 7, 8, 9

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

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

欲🄵代一個序列的索引值，你可以搭配使用 `range()` 和 `len()` 如下：

```
>>> 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
```

然而，在多數的情🄵，使用 `enumerate()` 函式將更🄵方便，詳見🄵圈技巧。

如果直接印出一個 range 則會出現奇怪的輸出：

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

在很多情🄵下，由 `range()` 回傳的物件的行🄵如同一個 list，但實際上它🄵不是。它是一個物件在你🄵代時會回傳想要的序列的連續元素，🄵不會真正建出這個序列的 list，以節省空間。

我們稱這樣的物件🄵 *iterable*（可🄵代的），意即能作🄵函式、陳述式中能一直獲取連續元素直到用盡的部件。我們已經看過 `for` 陳述式可做🄵如此的 *iterator*（🄵代器）。`list()` 函式🄵另一個例子，他可以自 iterable（可🄵代物件）建立 list：

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

待會我們可以看到更多函式回傳 iterable 和接受 iterable 🄵引數。

4.4 `break` and `continue` Statements, and `else` Clauses on Loops

`break` 陳述，如同 C 語言，終止包含其最🄵部的 `for` 或 `while` 🄵圈。

Loop statements may have an `else` clause; it is executed when the loop terminates through exhaustion of the list (with `for`) or when the condition becomes false (with `while`), but not when the loop is terminated by a `break` statement. This is exemplified by the following loop, which searches for prime numbers:

```
>>> 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
```

（🄵錯，這是正確的程式碼。請看仔細：`else` 段落屬於 `for` 🄵圈，🄵非 `if` 陳述。）

When used with a loop, the `else` clause has more in common with the `else` clause of a `try` statement than it does that of `if` statements: a `try` statement's `else` clause runs when no exception occurs, and a loop's `else` clause runs when no `break` occurs. For more on the `try` statement and exceptions, see 處理例外.

continue 陳述，亦承襲於 C 語言，讓所屬的🄵圈繼續執行下個🄵代：

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

4.5 pass Statements

pass 陳述不執行任何動作。它用在語法上需要一個陳述但不需要執行任何動作的時候。例如：

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

這經常用於定義一個最簡單的類🄵：

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

Another place pass can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The pass is silently ignored:

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

4.6 定義函式 (function)

我們可以建立一個函式來🄵生費式數列到任何一個上界：

```
>>> 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
```

關鍵字 def 帶入一個函式的定義。它之後必須連著該函式的名稱和置於括號之中的參數。自下一行起，所有縮排的陳述成🄵該函式的主體。

一個函式的第一個陳述可以是一個字串值；此情🄵該字串值被視🄵該函式的🄵明文件字串，即 *docstring*。（關於 docstring 的細節請參見🄵明文件字串段落。）有些工具可以使用 docstring 來自動🄵生🄵上或

可列印的文件，或讓使用者能自由地自原始碼中🄵覽文件。在原始碼中加入 docstring 是個好慣例，應該養成這樣的習慣。

The *execution* of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables and variables of enclosing functions cannot be directly assigned a value within a function (unless, for global variables, named in a `global` statement, or, for variables of enclosing functions, named in a `nonlocal` statement), although they may be referenced.

在一個函式被呼叫的時候，實際傳入的參數（引數）會被加入至該函數的區域符號表。因此，引數傳入的方式🄵傳值呼叫 *(call by value)*（這🄵傳遞的「值」永遠是一個物件的參照（*reference*），而不是該物件的值）。[1] 當一個函式呼叫🄵的函式時，在被呼叫的函式中會建立一個新的區域符號表。

一個函式定義會把該函式名稱加入至當前的符號表。該函式名稱的值帶有一個型🄵，🄵被直譯器辨識🄵使用者自定函式（user-defined function）。該值可以被賦予給🄵的變數名，而該變數也可以被當作函式使用。這即是常見的重新命名方式：

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

如果你是來自🄵的語言，你可能不同意 `fib` 是個函式，而是個程序 (procedure)，因🄵它🄵🄵有回傳值。實際上，即使一個函式缺少一個 `return` 陳述，它亦有一個固定的回傳值。這個值🄵 `None`（它是一個🄵建名稱）。在直譯器中單獨使用 `None` 時，通常不會被顯示。你可以使用 `print()` 來看到它：

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

如果要寫一個函式回傳費式數列的 list 而不是直接印出它，這也很容易：

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

這個例子一樣示範了一些新的 Python 特性：

- The `return` statement returns with a value from a function. `return` without an expression argument returns `None`. Falling off the end of a function also returns `None`.
- `result.append(a)` 陳述呼叫了一個 list 物件的 `result` *method*（*方法*）。method 🄵「屬於」一個物件的函式，命名規則🄵 `obj.methodname`，其中 `obj` 🄵某個物件（亦可🄵一表達式），而 `methodname` 🄵該 method 的名稱，🄵由該物件的型🄵所定義。不同的型🄵代表不同的 method。不同型🄵的 method 可以擁有一樣的名稱而不會讓 Python 混淆。（你可以使用 *class* 定義自己的物件型🄵和 method，見*Classes*）這🄵 `append()` method 定義在 list 物件中；它會加入一個新的元素在該 list 的末端。這個例子等同於 `result = result + [a]`，但更有效率。

[1] Actually, *call by object reference* would be a better description, since if a mutable object is passed, the caller will see any changes the callee makes to it (items inserted into a list).

4.7 More on Defining Functions

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

4.7.1 Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

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

This function can be called in several ways:

- giving only the mandatory argument: `ask_ok('Do you really want to quit?')`
- giving one of the optional arguments: `ask_ok('OK to overwrite the file?', 2)`
- or even giving all arguments: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

This example also introduces the `in` keyword. This tests whether or not a sequence contains a certain value.

The default values are evaluated at the point of function definition in the *defining* scope, so that

```
i = 5

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

i = 6
f()
```

will print `5`.

Important warning: The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:

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

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

This will print

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

If you don't want the default to be shared between subsequent calls, you can write the function like this instead:

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

4.7.2 Keyword Arguments

Functions can also be called using *keyword arguments* of the form `kwarg=value`. For instance, the following function:

```
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, "!")
```

accepts one required argument (`voltage`) and three optional arguments (`state`, `action`, and `type`). This function can be called in any of the following ways:

```
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
```

but all the following calls would be invalid:

```
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
```

In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function (e.g. `actor` is not a valid argument for the `parrot` function), and their order is not important. This also includes non-optional arguments (e.g. `parrot(voltage=1000)` is valid too). No argument may receive a value more than once. Here's an example that fails due to this restriction:

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

When a final formal parameter of the form `**name` is present, it receives a dictionary (see typesmapping) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form `*name` (described in the next subsection) which receives a *tuple* containing the positional arguments beyond the formal parameter list. (`*name` must occur before `**name`.) For example, if we define a function like this:

```
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)
```

(下页继续)

(繼續上一頁)

```
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

It could be called like this:

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

and of course it would print:

```
-- 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
```

Note that the order in which the keyword arguments are printed is guaranteed to match the order in which they were provided in the function call.

4.7.3 Arbitrary Argument Lists

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple (see *Tuples 和序列 (Sequences)*). Before the variable number of arguments, zero or more normal arguments may occur.

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

Normally, these `variadic` arguments will be last in the list of formal parameters, because they scoop up all remaining input arguments that are passed to the function. Any formal parameters which occur after the `*args` parameter are 'keyword-only' arguments, meaning that they can only be used as keywords rather than positional arguments.

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

4.7.4 Unpacking Argument Lists

The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments. For instance, the built-in `range()` function expects separate *start* and *stop* arguments. If they are not available separately, write the function call with the `*` operator to unpack the arguments out of a list or tuple:

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

(下页继续)

(繼續上一頁)

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

In the same fashion, dictionaries can deliver keyword arguments with the ** operator:

```
>>> 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 !
```

4.7.5 Lambda Expressions

Small anonymous functions can be created with the `lambda` keyword. This function returns the sum of its two arguments: `lambda a, b: a+b`. Lambda functions can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. Like nested function definitions, lambda functions can reference variables from the containing scope:

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

The above example uses a lambda expression to return a function. Another use is to pass a small function as an argument:

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

4.7.6 🄵明文件字串

Here are some conventions about the content and formatting of documentation strings.

The first line should always be a short, concise summary of the object's purpose. For brevity, it should not explicitly state the object's name or type, since these are available by other means (except if the name happens to be a verb describing a function's operation). This line should begin with a capital letter and end with a period.

If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description. The following lines should be one or more paragraphs describing the object's calling conventions, its side effects, etc.

The Python parser does not strip indentation from multi-line string literals in Python, so tools that process documentation have to strip indentation if desired. This is done using the following convention. The first non-blank line *after* the first line of the string determines the amount of indentation for the entire documentation string. (We can't use the first line since it is generally adjacent to the string's opening quotes so its indentation is not apparent in the string literal.) Whitespace "equivalent" to this indentation is then stripped from the start of all lines of the string. Lines that are indented less should not occur, but if they occur all their leading whitespace should be stripped. Equivalence of whitespace should be tested after expansion of tabs (to 8 spaces, normally).

Here is an example of a multi-line docstring:

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

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

4.7.7 Function Annotations

Function annotations are completely optional metadata information about the types used by user-defined functions (see **PEP 3107** and **PEP 484** for more information).

Annotations are stored in the `__annotations__` attribute of the function as a dictionary and have no effect on any other part of the function. Parameter annotations are defined by a colon after the parameter name, followed by an expression evaluating to the value of the annotation. Return annotations are defined by a literal `->`, followed by an expression, between the parameter list and the colon denoting the end of the `def` statement. The following example has a positional argument, a keyword argument, and the return value annotated:

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

4.8 Intermezzo: Coding Style

Now that you are about to write longer, more complex pieces of Python, it is a good time to talk about *coding style*. Most languages can be written (or more concise, *formatted*) in different styles; some are more readable than others. Making it easy for others to read your code is always a good idea, and adopting a nice coding style helps tremendously for that.

For Python, **PEP 8** has emerged as the style guide that most projects adhere to; it promotes a very readable and eye-pleasing coding style. Every Python developer should read it at some point; here are the most important points extracted for you:

- Use 4-space indentation, and no tabs.

  4 spaces are a good compromise between small indentation (allows greater nesting depth) and large indentation (easier to read). Tabs introduce confusion, and are best left out.

- Wrap lines so that they don't exceed 79 characters.

  This helps users with small displays and makes it possible to have several code files side-by-side on larger displays.

- Use blank lines to separate functions and classes, and larger blocks of code inside functions.

- When possible, put comments on a line of their own.

- Use docstrings.

- Use spaces around operators and after commas, but not directly inside bracketing constructs: `a = f(1, 2) + g(3, 4)`.
- Name your classes and functions consistently; the convention is to use `UpperCamelCase` for classes and `lowercase_with_underscores` for functions and methods. Always use `self` as the name for the first method argument (see *A First Look at Classes* for more on classes and methods).
- Don't use fancy encodings if your code is meant to be used in international environments. Python's default, UTF-8, or even plain ASCII work best in any case.
- Likewise, don't use non-ASCII characters in identifiers if there is only the slightest chance people speaking a different language will read or maintain the code.

🄵解

CHAPTER 5

資料結構

這個章節將會更深入的介紹一些你已經學過的東西的細節上，F且加入一些你還F有接觸過的部分。

5.1 進一步了解 List（串列）

List（串列）這個資料型態，具有更多操作的方法。下面條列了所有關於 list 的物件方法：

`list.`**`append`**`(`*x*`)`
將一個新的項目加到 list 的尾端。等同於 `a[len(a):] = [x]`。

`list.`**`extend`**`(`*iterable*`)`
將 iterable（可列舉物件）接到 list 的尾端。等同於 `a[len(a):] = iterable`。

`list.`**`insert`**`(`*i*, *x*`)`
將一個項目插入至 list 中給定的位置。第一個引數F插入處前元素的索引值，所以 `a.insert(0, x)` 會插入F list 首位，而 `a.insert(len(a), x)` 則相當於 `a.append(x)`。

`list.`**`remove`**`(`*x*`)`
Remove the first item from the list whose value is equal to *x*. It raises a `ValueError` if there is no such item.

`list.`**`pop`**`(`[*i*]`)`
移除 list 中給定位置的項目，F回傳它。如果F有指定位置，`a.pop()` 將會移除 list 中最後的項目F回傳它。（在 *i* 周圍的方括號代表這個參數是選用的，F不代表你應該在該位置輸入方括號。你將會常常在 Python 函式庫參考指南中看見這個表示法）

`list.`**`clear`**`()`
F除 list 中所有項目。這等同於 `del a[:]`。

`list.`**`index`**`(`*x*[, *start*[, *end*]]`)`
回傳 list 中第一個值等於 *x* 的項目之索引值（從零開始的索引）。若 list 中無此項目，則F出 `ValueError` 錯誤。

引數 *start* 和 *end* 的定義跟在 slice 表示法中相同，搜尋的動作被這兩個引數限定在 list 中特定的子序列。但要注意的是，回傳的索引值是從 list 的開頭開始算，而不是從 *start* 開始算。

`list.`**`count`**`(`*x*`)`
回傳數值F *x* 在 list 中所出現的次數。

`list.`**`sort`**`(`*key=None*, *reverse=False*`)`
將 list 中的項目排序。（有參數可以使用來進行客F化的排序，請參考 `sorted()` 部分的解釋）

`list.reverse()`
將 list 中的項目前後順序反過來。

`list.copy()`
回傳一個淺🄵🄵 (shallow copy) 的 list 。等同於 `a[:]`。

以下是一個使用到許多 list 物件方法的例子:

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

你可能會注意到一些方法，像是 `insert` 、`remove` 或者是 `sort` ，🄵不會印出回傳的值，事實上，他們回傳預設值 `None`[1]。這是一個用於 Python 中所有可變資料結構的設計法則。

5.1.1 將 List 作🄵 Stack（堆🄵）使用

List 的操作方法使得它非常簡單可以用來實作 stack（堆🄵）。Stack 🄵一個遵守最後加入元素最先被取回（後進先出，"last-in, first-out"）規則的資料結構。你可以使用方法 `append()` 將一個項目放到堆🄵的頂層。而使用方法 `pop()` 且不給定索引值去取得堆🄵最上面的項目。舉例而言:

```
>>> 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]
```

[1] 其他語言可能可以回傳可變的物件🄵允許方法串連，例如 `d->insert("a")->remove("b")->sort();` 。

5.1.2 將 List 作🄵 Queue（🄵列）使用

我們也可以將 list 當作 queue（🄵列）使用，即最先加入元素最先被取回（先進先出，"first-in, first-out"）的資料結構。然而，list 在這種使用方式下效率較差。使用 append 和 pop 來加入和取出尾端的元素較快，而使用 insert 和 pop 來插入和取出頭端的元素較慢（因🄵其他元素都需要挪動一格）。

如果要實作 queue，請使用 collections.deque，其被設計成能快速的從頭尾兩端加入和取出。例如：

```
>>> 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 List Comprehensions（串列綜合運算）

List Comprehension（串列綜合運算）讓你可以用簡潔的方法創建 list。常見的應用是基於一個 list 或 iterable（可列舉物件），將每一個元素經過某個運算的結果串接起來成🄵一個新的 list 。或是創建一個 list 的子序列，其每一個元素皆滿足一個特定的條件。

舉例來🄵，假設我們要創建一個「平方的 list」：

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

注意這是創建（或🄵寫）一個變數叫 x 其在🄵圈結束後仍然存在。我們可以這樣🄵生平方串列而不造成任何 side effects（副作用）：

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

或與此相等的：

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

這樣更簡潔和易讀。

A list comprehension consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses. The result will be a new list resulting from evaluating the expression in the context of the for and if clauses which follow it. For example, this listcomp combines the elements of two lists if they are not equal:

```
>>> [(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)]
```

而這和下者相同：

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

(下页继续)

(繼續上一頁)

```
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

注意 for 和 if 在這兩段程式🄵的順序是相同的。

如果 expression 是一個 tuple（例如上面例子中的 (x, y)），它必須加上小括弧：

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1, in <module>
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List comprehensions 可以含有🄵雜的 expression 和巢狀的函式呼叫：

```
>>> 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 巢狀的 List Comprehensions

最初放在 list comprehesion 中的 expression 可以是任何形式的 expression，包括再寫一個 list comprehension。

考慮以下表示 3x4 矩陣的範例，使用 list 包含 3 個長度🄵 4 的 list：

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

以下的 list comprehesion 會將矩陣的行與列作轉置：

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

如同我們在上一節看到的，此巢狀的 list comprehension 🄵一個 list comprehension 在 for 之前先被計算，接著再作一次 list comprehension，所以，這個例子和下者相同：

```
>>> 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]]
```

因此，也和下者相同：

```
>>> 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]]
```

在實際運用上，我們傾向於使用🄵建函式 (built-in functions) 而不是🄵雜的流程控制陳述式。在這個例子中，使用 `zip()` 函式會非常有效率：

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

關於星號的更多細節，請參考*Unpacking Argument Lists* 。

5.2 The `del` statement

There is a way to remove an item from a list given its index instead of its value: the `del` statement. This differs from the `pop()` method which returns a value. The `del` statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

```
>>> 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` 也可以用來🄵除變數：

```
>>> del a
```

🄵除之後，對 `a` 的參照將會造成錯誤（至少在另一個值又被指派到它之前）。我們將在後面看到更多關於 `del` 的其他用法。

5.3 Tuples 和序列 (Sequences)

我們看到 lists 和 strings 有許多共同的特性，像是索引操作 (indexing) 以及切片操作 (slicing) 。他們是 序列資料結構中的兩個例子（請參考 typesseq ）。由於 Python 是個持續發展中的語言，未來可能還會有其他的序列資料結構加入。接著要介紹是下一個標准序列資料結構：*tuple* 。

一個 tuple 是由若干個值藉由逗號區隔而組成，例如：

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

如同我們看到的，被輸出的 tuple 總是以括號包著，如此巢狀的 tuple 才能被正確的直譯 (interpret)；他們可以是以被括號包著或不被包著的方式當作輸入，雖然括號的使用常常是有其必要的（譬如此 tuple 是一個較大的陳述式的一部分）。指派東西給 tuple 中個🄴的項是不行的，但是可以在 tuple 中放入含有可變項的物件，譬如 list 。

雖然 tuple 和 list 看起來很類似，但是他們通常用在不同的情🄴與不同目的。tuple 是*immutable* （不可變的），通常儲存🄴質的序列元素，🄴可經由拆解 (unpacking) （請參考本節後段）或索引 (indexing) 來存取（或者在使用 `namedtuples` 的時候藉由屬性 (attribute) 來存取）。list 是*mutable* （可變的），其元素通常是同質的且可藉由迭代整個串列來存取。

一個特🄴的議題是，關於創建一個含有 0 個或 1 個項目的 tuple：語法上會🄴納一些奇怪的用法。空的 tuple 藉由一對空括號來創建；含有一個項目的 tuple 經由一個值加上一個逗點來創建（不需要用括號把一個單一的值包住）。醜，但有效率。例如：

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

陳述式 `t = 12345, 54321, 'hello!'` 就是一個 *tuple packing* 的例子：`12345` , `54321` 和 `'hello!'` 一起被放進 tuple 🄴。反向操作也可以：

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

這個正是我們所🄴序列拆解 (*sequence unpacking*) ，可運用在任何位在等號右邊的序列。序列拆解要求等號左邊的變數數量必須與等號右邊的序列中的元素數量相同。注意，多重指派就只是 tuple packing 和序列拆解的結合而已。

5.4 集合 (Sets)

Python 也包含了一種用在集合 (*sets*) 的資料結構。一個 set 是一組無序且𠀀有重𠀀的元素。基本的使用方式包括了成員測試和消除重𠀀項。Set 物件也支援聯集，交集，差集和互斥等數學操作。

大括號或 `set()` 函式都可以用來創建 set 。注意：要創建一個空的 set ，我們必須使用 `set()` 而不是 `{}` ；後者會創建一個空的 dictionary ，一種我們將在下一節討論的資料結構。

這𠀀是一個簡單的演示：

```
>>> 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'}
```

和*list comprehensions* 類似，也有 set comprehensions （集合綜合運算）：

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

5.5 字典（Dictionary）

下一個常用的 python 𠀀建資料結構𠀀 *dictionary* （請參考 typesmapping ）。Dictionary 有時被稱𠀀" 關聯記憶體" (associative memories) 或" 關聯矩陣" (associative arrays) 。不像序列是由一個範圍𠀀的數字當作索引，dictionary 是由 *key* （鍵）來當索引，key 可以是任何不可變的型態；字串和數字都可以當作 key 。Tuple 也可以當作 key 如果他們只含有字串、數字或 tuple；若一個 tuple 直接或間接地含有任何可變的物件，它就不能當作 key 。我們無法使用 list 當作 key ，因𠀀 list 可以經由索引操作、切片操作或是方法像是 `append()` 和 `extend()` 來修改。

思考 dict 最好的方式是把它想成是一組鍵值對 (*key: value* pair) 的集合，其中 key 在同一個 dictionary（字典）𠀀必須是獨一無二的。使用一對大括號可創建一個空的字典：`{}`。將一串由逗號分隔的鍵值對置於大括號則可初始化字典。這同樣也是字典輸出時的格式。

Dict 主要的操作𠀀藉由鍵來儲存一個值𠀀且可藉由該鍵來取出該值。也可以使用 `del` 來𠀀除鍵值對。如果我們使用用過的鍵來儲存，該鍵所對應的較舊的值會被覆蓋。使用不存在的鍵來取出值會造成錯誤。

對字典使用 `list(d)` 會得到一個包含該字典所有鍵（key）的 list，其排列順序𠀀插入時的順序。（若想要排序，則使用 `sorted(d)` 代替即可）。如果想確認一個鍵是否已存在於字典中，可使用關鍵字 `in` 。

這是個使用一個字典的簡單範例：

```
>>> 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
```

函式 `dict()` 可直接透過一串鍵值對序列來創建 dict：

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

此外，dict comprehensions 也可以透過鍵與值的陳述式來創建 dict：

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

當鍵是簡單的字串時，使用關鍵字引數 (keyword arguments) 有時會較🄵簡潔：

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

5.6 🄵圈技巧

當對 dict 作🄵圈時，鍵以及其對應的值可以藉由使用 `items()` 方法來同時取得：

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

當對序列作🄵圈時，位置索引及其對應的值可以藉由使用 `enumerate()` 函式來同時取得：

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

要同時對兩個以上的序列作🄵圈，可以將其以成對的方式放入 `zip()` 函式：

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
```

(下页继续)

(繼續上一頁)

```
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

要對序列作反向的🄵圈，首先先寫出正向的序列，在對其使用 `reversed()` 函式：

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

要以🄵圈對序列作排序，使用 `sorted()` 函式會得到一個新的經排序過的 list ，但不會改變原本的序列：

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

有時我們會想要以🄵圈來改變的一個 list，但是，通常創建一個新的 list 會更簡單且安全：

```
>>> 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 更多條件式主題

使用在 `while` 和 `if` 🄵的陳述式可以包含任何運算子，而不是只有比較運算子 (comparisons) 。

比較運算子 `in` 和 `not in` 檢查一個數值是否存在（不存在）於一個序列中。運算子 `is` 和 `not is` 比較兩個物件是否真的是相同的物件；這對可變物件例如 list 來🄵很重要。所有的比較運算子優先度都相同且都低於數值運算子。

比較運算是可以串連在一起的。例如，`a < b == c` 就是在測試 `a` 是否小於 `b` 和 `b` 是否等於 `c` 。

比較運算可以結合布林運算子 `and` 和 `or` ，且一個比較運算的結果（或任何其他布林表達式）可以加上 `not` 來否定。這些運算子的優先度都比比較運算子還低，其中，`not` 的優先度最高，`or` 的優先度最低，因此 `A and not B or C` 等同於 `(A and (not B)) or C` 。一如往常，括號可以用來表示任何想要的組合。

布林運算子 `and` 和 `or` 也被稱🄵短路 (*short-circuit*) 運算子：會將其引數從左至右進行運算，當結果出現時即結束運算。例如，若 `A` 和 `C` 🄵真但 `B` 🄵假，則 `A and B and C` 的運算🄵不會執行到 `C` 。當運算結果被當成一般數值而非布林值時，短路運算子的回傳值🄵最後被運算的引數。

將一個比較運算或其他布林表達式的結果指派給一個變數是可以的。例如：

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

注意，Python 不像 C 語言，在表達式🄴進行指派是不行的。C 語言的程式設計師可能會抱怨這件事，但這樣做避免了在 C 語言🄵常見的一種問題：想要打 == 🄵在表達式🄵輸入 = 。

5.8 序列和其他資料結構之比較

序列物件可以拿來和其他相同型態的物件做比較。這種比較使用詞典式順序 (*lexicographical* ordering)：首先比較各自最前面的那項，若不相同，便可🄴定結果，若相同，則比較下丨項，以此類推，直到其中一個序列完全用完。如果被拿出來比較的兩項本身又是相同的序列型態，則詞典式順序的比較會遞🄴處理。如果兩個序列所有的項都相等，則此兩個序列被認🄴是相等的。如果其中一個序列是另一個的子序列，則較短的那個序列🄴較小的序列。字串的詞典式順序使用 Unicode 的碼位 (code point) 編號來排序個🄴字元。以下是一些相同序列型態的比較：

```
(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)
```

注意，若使用 < 或 > 來比較不同型態的物件是合法的，表示物件擁有適當的比較方法。例如，混合型數值比較是根據它們數字的值來做比較，所以 0 等於 0.0，等等。否則直譯器會選擇🄵出一個 `TypeError` 錯誤而不是提供一個任意的排序。

🄵解

CHAPTER 6

模組

如果從 Python 直譯器離開後又再次進入，之前（幫函式或變數）做的定義都會消失。因此，想要寫一些比較長的程式時，你最好使用編輯器來准備要輸入給直譯器的🄴容，🄵且用該檔案來運行它。這就是一個🄵本（*script*）。隨著你的程式越變越長，你可能會想要把它分開成幾個檔案，讓它比較好維護。你可能也會想用一個你之前已經在其他程式寫好的函式，但不想要🄵🄵該函式的原始定義到所有使用它的程式🄵。

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, use your favorite text editor to create a file called `fibo.py` in the current directory with the following contents:

```
# 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
```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

This does not enter the names of the functions defined in `fibo` directly in the current symbol table; it only enters the module name `fibo` there. Using the module name you can access the functions:

```
>>> 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'
```

If you intend to use a function often you can assign it to a local name:

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

6.1 More on Modules

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the *first* time the module name is encountered in an import statement.[1] (They are also run if the file is executed as a script.)

Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions, `modname.itemname`.

Modules can import other modules. It is customary but not required to place all `import` statements at the beginning of a module (or script, for that matter). The imported module names are placed in the importing module's global symbol table.

There is a variant of the `import` statement that imports names from a module directly into the importing module's symbol table. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, `fibo` is not defined).

There is even a variant to import all names that a module defines:

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

This imports all names except those beginning with an underscore (_). In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

Note that in general the practice of importing `*` from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

If the module name is followed by `as`, then the name following `as` is bound directly to the imported module.

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

[1] In fact function definitions are also 'statements' that are 'executed'; the execution of a module-level function definition enters the function name in the module's global symbol table.

This is effectively importing the module in the same way that `import fibo` will do, with the only difference of it being available as `fib`.

It can also be used when utilising `from` with similar effects:

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

備🄵: For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter -- or, if it's just one module you want to test interactively, use `importlib.reload()`, e.g. `import importlib; importlib.reload(modulename)`.

6.1.1 Executing modules as scripts

When you run a Python module with

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

the code in the module will be executed, just as if you imported it, but with the `__name__` set to `"__main__"`. That means that by adding this code at the end of your module:

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

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the "main" file:

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

If the module is imported, the code is not run:

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

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

6.1.2 The Module Search Path

When a module named `spam` is imported, the interpreter first searches for a built-in module with that name. If not found, it then searches for a file named `spam.py` in a list of directories given by the variable `sys.path`. `sys.path` is initialized from these locations:

- The directory containing the input script (or the current directory when no file is specified).
- `PYTHONPATH` (a list of directory names, with the same syntax as the shell variable `PATH`).
- The installation-dependent default.

備🄵: On file systems which support symlinks, the directory containing the input script is calculated after the symlink is followed. In other words the directory containing the symlink is **not** added to the module search path.

After initialization, Python programs can modify `sys.path`. The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path. This means that scripts in that directory will

be loaded instead of modules of the same name in the library directory. This is an error unless the replacement is intended. See section *Standard Modules* for more information.

6.1.3 ”Compiled” Python files

To speed up loading modules, Python caches the compiled version of each module in the `__pycache__` directory under the name `module.version.pyc`, where the version encodes the format of the compiled file; it generally contains the Python version number. For example, in CPython release 3.3 the compiled version of spam.py would be cached as `__pycache__/spam.cpython-33.pyc`. This naming convention allows compiled modules from different releases and different versions of Python to coexist.

Python checks the modification date of the source against the compiled version to see if it's out of date and needs to be recompiled. This is a completely automatic process. Also, the compiled modules are platform-independent, so the same library can be shared among systems with different architectures.

Python does not check the cache in two circumstances. First, it always recompiles and does not store the result for the module that's loaded directly from the command line. Second, it does not check the cache if there is no source module. To support a non-source (compiled only) distribution, the compiled module must be in the source directory, and there must not be a source module.

Some tips for experts:

- You can use the `-O` or `-OO` switches on the Python command to reduce the size of a compiled module. The `-O` switch removes assert statements, the `-OO` switch removes both assert statements and __doc__ strings. Since some programs may rely on having these available, you should only use this option if you know what you're doing. ”Optimized” modules have an `opt-` tag and are usually smaller. Future releases may change the effects of optimization.
- A program doesn't run any faster when it is read from a `.pyc` file than when it is read from a `.py` file; the only thing that's faster about `.pyc` files is the speed with which they are loaded.
- The module `compileall` can create .pyc files for all modules in a directory.
- There is more detail on this process, including a flow chart of the decisions, in **PEP 3147**.

6.2 Standard Modules

Python comes with a library of standard modules, described in a separate document, the Python Library Reference (”Library Reference” hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform. For example, the `winreg` module is only provided on Windows systems. One particular module deserves some attention: `sys`, which is built into every Python interpreter. The variables `sys.ps1` and `sys.ps2` define the strings used as primary and secondary prompts:

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

These two variables are only defined if the interpreter is in interactive mode.

The variable `sys.path` is a list of strings that determines the interpreter's search path for modules. It is initialized to a default path taken from the environment variable `PYTHONPATH`, or from a built-in default if `PYTHONPATH` is not set. You can modify it using standard list operations:

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

6.3 The `dir()` Function

The built-in function `dir()` is used to find out which names a module defines. It returns a sorted list of strings:

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

Without arguments, `dir()` lists the names you have defined currently:

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

Note that it lists all types of names: variables, modules, functions, etc.

`dir()` does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module `builtins`:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
 'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
```

(下页继续)

(繼續上一頁)

```
 '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 Packages

Packages are a way of structuring Python's module namespace by using "dotted module names". For example, the module name `A.B` designates a submodule named `B` in a package named `A`. Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or Pillow from having to worry about each other's module names.

Suppose you want to design a collection of modules (a "package") for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: `.wav`, `.aiff`, `.au`), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect), so in addition you will be writing a never-ending stream of modules to perform these operations. Here's a possible structure for your package (expressed in terms of a hierarchical filesystem):

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

When importing the package, Python searches through the directories on `sys.path` looking for the package subdirectory.

The `__init__.py` files are required to make Python treat directories containing the file as packages. This prevents

directories with a common name, such as `string`, unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.

Users of the package can import individual modules from the package, for example:

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

This loads the submodule `sound.effects.echo`. It must be referenced with its full name.

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

An alternative way of importing the submodule is:

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

This also loads the submodule `echo`, and makes it available without its package prefix, so it can be used as follows:

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

Yet another variation is to import the desired function or variable directly:

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

Again, this loads the submodule `echo`, but this makes its function `echofilter()` directly available:

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

Note that when using `from package import item`, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable. The `import` statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an `ImportError` exception is raised.

Contrarily, when using syntax like `import item.subitem.subsubitem`, each item except for the last must be a package; the last item can be a module or a package but can't be a class or function or variable defined in the previous item.

6.4.1 Importing * From a Package

Now what happens when the user writes `from sound.effects import *`? Ideally, one would hope that this somehow goes out to the filesystem, finds which submodules are present in the package, and imports them all. This could take a long time and importing sub-modules might have unwanted side-effects that should only happen when the sub-module is explicitly imported.

The only solution is for the package author to provide an explicit index of the package. The `import` statement uses the following convention: if a package's `__init__.py` code defines a list named `__all__`, it is taken to be the list of module names that should be imported when `from package import *` is encountered. It is up to the package author to keep this list up-to-date when a new version of the package is released. Package authors may also decide not to support it, if they don't see a use for importing * from their package. For example, the file `sound/effects/__init__.py` could contain the following code:

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

This would mean that `from sound.effects import *` would import the three named submodules of the `sound` package.

If `__all__` is not defined, the statement `from sound.effects import *` does *not* import all submodules from the package `sound.effects` into the current namespace; it only ensures that the package `sound.effects` has been imported (possibly running any initialization code in `__init__.py`) and then imports whatever names are defined in the package. This includes any names defined (and submodules explicitly loaded) by `__init__.py`. It also includes any submodules of the package that were explicitly loaded by previous `import` statements. Consider this code:

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

In this example, the `echo` and `surround` modules are imported in the current namespace because they are defined in the `sound.effects` package when the `from...import` statement is executed. (This also works when `__all__` is defined.)

Although certain modules are designed to export only names that follow certain patterns when you use `import *`, it is still considered bad practice in production code.

Remember, there is nothing wrong with using `from package import specific_submodule`! In fact, this is the recommended notation unless the importing module needs to use submodules with the same name from different packages.

6.4.2 Intra-package References

When packages are structured into subpackages (as with the `sound` package in the example), you can use absolute imports to refer to submodules of siblings packages. For example, if the module `sound.filters.vocoder` needs to use the `echo` module in the `sound.effects` package, it can use `from sound.effects import echo`.

You can also write relative imports, with the `from module import name` form of import statement. These imports use leading dots to indicate the current and parent packages involved in the relative import. From the `surround` module for example, you might use:

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

Note that relative imports are based on the name of the current module. Since the name of the main module is always `"__main__"`, modules intended for use as the main module of a Python application must always use absolute imports.

6.4.3 Packages in Multiple Directories

Packages support one more special attribute, `__path__`. This is initialized to be a list containing the name of the directory holding the package's `__init__.py` before the code in that file is executed. This variable can be modified; doing so affects future searches for modules and subpackages contained in the package.

While this feature is not often needed, it can be used to extend the set of modules found in a package.

🄵解

CHAPTER 7

輸入和輸出

有數種方式可以顯示程式的輸出；資料可以以人類易讀的形式印出，或是寫入檔案以供未來所使用。這章節會討論幾種不同的方式。

7.1 更華麗的輸出格式

目前🄴止我們已經學過兩種寫值的方式：表示式陳述 *(expression statements)* 與 `print()` 函式。(第三種方法是使用檔案物件的 `write()` 方法；標准輸出的檔案是使用 `sys.stdout` 來達成的。詳細的資訊請參考對應的函示庫🄴明。)

通常你會想要對輸出格式有更多地控制；而不是僅列印出以空格隔開的值。以下是幾種格式化輸出的方式。

- To use *formatted string literals*, begin a string with `f` or `F` before the opening quotation mark or triple quotation mark. Inside this string, you can write a Python expression between `{` and `}` characters that can refer to variables or literal values.

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

- The `str.format()` method of strings requires more manual effort. You'll still use `{` and `}` to mark where a variable will be substituted and can provide detailed formatting directives, but you'll also need to provide the information to be formatted.

```
>>> 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%'
```

- Finally, you can do all the string handling yourself by using string slicing and concatenation operations to create any layout you can imagine. The string type has some methods that perform useful operations for padding strings to a given column width.

When you don't need fancy output but just want a quick display of some variables for debugging purposes, you can convert any value to a string with the `repr()` or `str()` functions.

The `str()` function is meant to return representations of values which are fairly human-readable, while `repr()` is meant to generate representations which can be read by the interpreter (or will force a `SyntaxError` if there is no equivalent syntax). For objects which don't have a particular representation for human consumption, `str()` will return the same value as `repr()`. Many values, such as numbers or structures like lists and dictionaries, have the same representation using either function. Strings, in particular, have two distinct representations.

一些範例:

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

The `string` module contains a `Template` class that offers yet another way to substitute values into strings, using placeholders like `$x` and replacing them with values from a dictionary, but offers much less control of the formatting.

7.1.1 格式化的字串文本 (Formatted String Literals)

Formatted string literals (also called f-strings for short) let you include the value of Python expressions inside a string by prefixing the string with `f` or `F` and writing expressions as `{expression}`.

An optional format specifier can follow the expression. This allows greater control over how the value is formatted. The following example rounds pi to three places after the decimal:

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

Passing an integer after the `':'` will cause that field to be a minimum number of characters wide. This is useful for making columns line up.

```
>>> 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
```

Other modifiers can be used to convert the value before it is formatted. `'!a'` applies `ascii()`, `'!s'` applies `str()`, and `'!r'` applies `repr()`:

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

(下页继续)

(繼續上一頁)

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

For a reference on these format specifications, see the reference guide for the formatspec.

7.1.2 The String format() Method

Basic usage of the `str.format()` method looks like this:

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

The brackets and characters within them (called format fields) are replaced with the objects passed into the `str.format()` method. A number in the brackets can be used to refer to the position of the object passed into the `str.format()` method.

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

If keyword arguments are used in the `str.format()` method, their values are referred to by using the name of the argument.

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

Positional and keyword arguments can be arbitrarily combined:

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

If you have a really long format string that you don't want to split up, it would be nice if you could reference the variables to be formatted by name instead of by position. This can be done by simply passing the dict and using square brackets `'[]'` to access the keys.

```
>>> 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
```

This could also be done by passing the table as keyword arguments with the '**' notation.

```
>>> 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
```

This is particularly useful in combination with the built-in function `vars()`, which returns a dictionary containing all local variables.

As an example, the following lines produce a tidily-aligned set of columns giving integers and their squares and cubes:

```
>>> 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
```

(下页继续)

(繼續上一頁)

```
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

For a complete overview of string formatting with `str.format()`, see formatstrings.

7.1.3 Manual String Formatting

Here's the same table of squares and cubes, formatted manually:

```
>>> 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
```

(Note that the one space between each column was added by the way `print()` works: it always adds spaces between its arguments.)

The `str.rjust()` method of string objects right-justifies a string in a field of a given width by padding it with spaces on the left. There are similar methods `str.ljust()` and `str.center()`. These methods do not write anything, they just return a new string. If the input string is too long, they don't truncate it, but return it unchanged; this will mess up your column lay-out but that's usually better than the alternative, which would be lying about a value. (If you really want truncation you can always add a slice operation, as in `x.ljust(n)[:n]`.)

There is another method, `str.zfill()`, which pads a numeric string on the left with zeros. It understands about plus and minus signs:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

7.1.4 Old string formatting

The % operator (modulo) can also be used for string formatting. Given `'string' % values`, instances of `%` in `string` are replaced with zero or more elements of `values`. This operation is commonly known as string interpolation. For example:

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

More information can be found in the old-string-formatting section.

7.2 Reading and Writing Files

`open()` returns a *file object*, and is most commonly used with two arguments: `open(filename, mode)`.

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

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used. *mode* can be `'r'` when the file will only be read, `'w'` for only writing (an existing file with the same name will be erased), and `'a'` opens the file for appending; any data written to the file is automatically added to the end. `'r+'` opens the file for both reading and writing. The *mode* argument is optional; `'r'` will be assumed if it's omitted.

Normally, files are opened in *text mode*, that means, you read and write strings from and to the file, which are encoded in a specific encoding. If encoding is not specified, the default is platform dependent (see `open()`). `'b'` appended to the mode opens the file in *binary mode*: now the data is read and written in the form of bytes objects. This mode should be used for all files that don't contain text.

In text mode, the default when reading is to convert platform-specific line endings (`\n` on Unix, `\r\n` on Windows) to just `\n`. When writing in text mode, the default is to convert occurrences of `\n` back to platform-specific line endings. This behind-the-scenes modification to file data is fine for text files, but will corrupt binary data like that in `JPEG` or `EXE` files. Be very careful to use binary mode when reading and writing such files.

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

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

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

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

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

7.2.1 Methods of File Objects

The rest of the examples in this section will assume that a file object called `f` has already been created.

To read a file's contents, call `f.read(size)`, which reads some quantity of data and returns it as a string (in text mode) or bytes object (in binary mode). *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned; it's your problem if the file is twice as large as your machine's memory. Otherwise, at most *size* characters (in text mode) or *size* bytes (in binary mode) are read and returned. If the end of the file has been reached, `f.read()` will return an empty string (`''`).

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

`f.readline()` reads a single line from the file; a newline character (`\n`) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline. This makes the return value unambiguous; if `f.readline()` returns an empty string, the end of the file has been reached, while a blank line is represented by `'\n'`, a string containing only a single newline.

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

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

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

If you want to read all the lines of a file in a list you can also use `list(f)` or `f.readlines()`.

`f.write(string)` writes the contents of *string* to the file, returning the number of characters written.

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

Other types of objects need to be converted -- either to a string (in text mode) or a bytes object (in binary mode) -- before writing them:

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

`f.tell()` returns an integer giving the file object's current position in the file represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode.

To change the file object's position, use `f.seek(offset, whence)`. The position is computed from adding *offset* to a reference point; the reference point is selected by the *whence* argument. A *whence* value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. *whence* can be omitted and defaults to 0, using the beginning of the file as the reference point.

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # Go to the 6th byte in the file
```

(下页继续)

(繼續上一頁)

```
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

In text files (those opened without a `b` in the mode string), only seeks relative to the beginning of the file are allowed (the exception being seeking to the very file end with `seek(0, 2)`) and the only valid *offset* values are those returned from the `f.tell()`, or zero. Any other *offset* value produces undefined behaviour.

File objects have some additional methods, such as `isatty()` and `truncate()` which are less frequently used; consult the Library Reference for a complete guide to file objects.

7.2.2 Saving structured data with `json`

Strings can easily be written to and read from a file. Numbers take a bit more effort, since the `read()` method only returns strings, which will have to be passed to a function like `int()`, which takes a string like `'123'` and returns its numeric value 123. When you want to save more complex data types like nested lists and dictionaries, parsing and serializing by hand becomes complicated.

Rather than having users constantly writing and debugging code to save complicated data types to files, Python allows you to use the popular data interchange format called JSON (JavaScript Object Notation). The standard module called `json` can take Python data hierarchies, and convert them to string representations; this process is called *serializing*. Reconstructing the data from the string representation is called *deserializing*. Between serializing and deserializing, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine.

備🄵: The JSON format is commonly used by modern applications to allow for data exchange. Many programmers are already familiar with it, which makes it a good choice for interoperability.

If you have an object `x`, you can view its JSON string representation with a simple line of code:

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

Another variant of the `dumps()` function, called `dump()`, simply serializes the object to a *text file*. So if `f` is a *text file* object opened for writing, we can do this:

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

To decode the object again, if `f` is a *text file* object which has been opened for reading:

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

This simple serialization technique can handle lists and dictionaries, but serializing arbitrary class instances in JSON requires a bit of extra effort. The reference for the `json` module contains an explanation of this.

也參考:

`pickle` - the pickle module

Contrary to *JSON*, *pickle* is a protocol which allows the serialization of arbitrarily complex Python objects. As such, it is specific to Python and cannot be used to communicate with applications written in other languages. It is also insecure by default: deserializing pickle data coming from an untrusted source can execute arbitrary code, if the data was crafted by a skilled attacker.

CHAPTER 8

錯誤和例外

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

8.1 語法錯誤

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

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

The parser repeats the offending line and displays a little 'arrow' pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token *preceding* the arrow: in the example, the error is detected at the function `print()`, since a colon (`':'`) is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

8.2 例外

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

(下页继续)

(繼續上一頁)

```
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are `ZeroDivisionError`, `NameError` and `TypeError`. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception happened, in the form of a stack traceback. In general it contains a stack traceback listing source lines; however, it will not display lines read from standard input.

bltin-exceptions lists the built-in exceptions and their meanings.

8.3 處理例外

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using `Control-C` or whatever the operating system supports); note that a user-generated interruption is signalled by raising the `KeyboardInterrupt` exception.

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

The `try` statement works as follows.

- First, the *try clause* (the statement(s) between the `try` and `except` keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the `try` statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the `except` keyword, the except clause is executed, and then execution continues after the `try` statement.
- If an exception occurs which does not match the exception named in the except clause, it is passed on to outer `try` statements; if no handler is found, it is an *unhandled exception* and execution stops with a message as shown above.

A `try` statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same `try` statement. An except clause may name multiple exceptions as a parenthesized tuple, for example:

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

A class in an `except` clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around --- an except clause listing a derived class is not compatible with a base class). For example, the following code will print B, C, D in that order:

```
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")
```

Note that if the except clauses were reversed (with `except B` first), it would have printed B, B, B --- the first matching except clause is triggered.

The last except clause may omit the exception name(s), to serve as a wildcard. Use this with extreme caution, since it is easy to mask a real programming error in this way! It can also be used to print an error message and then re-raise the exception (allowing a caller to handle the exception as well):

```
import sys

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

The `try` ... `except` statement has an optional *else clause*, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception. For example:

```
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()
```

The use of the `else` clause is better than adding additional code to the `try` clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try` ... `except` statement.

When an exception occurs, it may have an associated value, also known as the exception's *argument*. The presence and type of the argument depend on the exception type.

The except clause may specify a variable after the exception name. The variable is bound to an exception instance with the arguments stored in `instance.args`. For convenience, the exception instance defines `__str__()` so the arguments can be printed directly without having to reference `.args`. One may also instantiate an exception first before raising it and add any attributes to it as desired.

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

If an exception has arguments, they are printed as the last part ('detail') of the message for unhandled exceptions.

Exception handlers don't just handle exceptions if they occur immediately in the try clause, but also if they occur inside functions that are called (even indirectly) in the try clause. For example:

```
>>> 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 Raising Exceptions

The `raise` statement allows the programmer to force a specified exception to occur. For example:

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

The sole argument to `raise` indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from `Exception`). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

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

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the `raise` statement allows you to re-raise the exception:

```
>>> 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 User-defined Exceptions

Programs may name their own exceptions by creating a new exception class (see *Classes* for more about Python classes). Exceptions should typically be derived from the `Exception` class, either directly or indirectly.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception. When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions:

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

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

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

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

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

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

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

Most exceptions are defined with names that end in ”Error”, similar to the naming of the standard exceptions.

Many standard modules define their own exceptions to report errors that may occur in functions they define. More information on classes is presented in chapter *Classes*.

8.6 Defining Clean-up Actions

The `try` statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

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

If a `finally` clause is present, the `finally` clause will execute as the last task before the `try` statement completes. The `finally` clause runs whether or not the `try` statement produces an exception. The following points discuss more complex cases when an exception occurs:

- If an exception occurs during execution of the `try` clause, the exception may be handled by an `except` clause. If the exception is not handled by an `except` clause, the exception is re-raised after the `finally` clause has been executed.
- An exception could occur during execution of an `except` or `else` clause. Again, the exception is re-raised after the `finally` clause has been executed.
- If the `try` statement reaches a `break`, `continue` or `return` statement, the `finally` clause will execute just prior to the `break`, `continue` or `return` statement's execution.
- If a `finally` clause includes a `return` statement, the returned value will be the one from the `finally` clause's `return` statement, not the value from the `try` clause's `return` statement.

For example:

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

A more complicated example:

```
>>> 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'
```

As you can see, the `finally` clause is executed in any event. The `TypeError` raised by dividing two strings is not handled by the `except` clause and therefore re-raised after the `finally` clause has been executed.

In real world applications, the `finally` clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

8.7 Predefined Clean-up Actions

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen.

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

The problem with this code is that it leaves the file open for an indeterminate amount of time after this part of the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The `with` statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

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

After the statement is executed, the file *f* is always closed, even if a problem was encountered while processing the lines. Objects which, like files, provide predefined clean-up actions will indicate this in their documentation.

CHAPTER 9

Classes

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are *public* (except see below *Private Variables*), and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)

9.1 A Word About Names and Objects

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change --- this eliminates the need for two different argument passing mechanisms as in Pascal.

9.2 Python Scopes and Namespaces

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let's begin with some definitions.

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as `abs()`, and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion --- users of the modules must prefix it with the module name.

By the way, I use the word *attribute* for any name following a dot --- for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace![1]

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A *scope* is a textual region of a Python program where a namespace is directly accessible. "Directly accessible" here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are at least three nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contains non-local, but also non-global names
- the next-to-last scope contains the current module's global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the middle scope containing the module's global names. To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used; if not declared nonlocal, those variables are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

[1] Except for one thing. Module objects have a secret read-only attribute called `__dict__` which returns the dictionary used to implement the module's namespace; the name `__dict__` is an attribute but not a global name. Obviously, using this violates the abstraction of namespace implementation, and should be restricted to things like post-mortem debuggers.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time --- however, the language definition is evolving towards static name resolution, at "compile" time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that -- if no `global` or `nonlocal` statement is in effect -- assignments to names always go into the innermost scope. Assignments do not copy data --- they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope.

The `global` statement can be used to indicate that particular variables live in the global scope and should be rebound there; the `nonlocal` statement indicates that particular variables live in an enclosing scope and should be rebound there.

9.2.1 Scopes and Namespaces Example

This is an example demonstrating how to reference the different scopes and namespaces, and how `global` and `nonlocal` affect variable binding:

```
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)
```

The output of the example code is:

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

Note how the *local* assignment (which is default) didn't change *scope_test*'s binding of *spam*. The `nonlocal` assignment changed *scope_test*'s binding of *spam*, and the `global` assignment changed the module-level binding.

You can also see that there was no previous binding for *spam* before the `global` assignment.

9.3 A First Look at Classes

Classes introduce a little bit of new syntax, three new object types, and some new semantics.

9.3.1 Class Definition Syntax

The simplest form of class definition looks like this:

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

Class definitions, like function definitions (`def` statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an `if` statement, or inside a function.)

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful --- we'll come back to this later. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods --- again, this is explained later.

When a class definition is entered, a new namespace is created, and used as the local scope --- thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

When a class definition is left normally (via the end), a *class object* is created. This is basically a wrapper around the contents of the namespace created by the class definition; we'll learn more about class objects in the next section. The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (`ClassName` in the example).

9.3.2 Class Objects

Class objects support two kinds of operations: attribute references and instantiation.

Attribute references use the standard syntax used for all attribute references in Python: `obj.name`. Valid attribute names are all the names that were in the class's namespace when the class object was created. So, if the class definition looked like this:

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

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

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment. `__doc__` is also a valid attribute, returning the docstring belonging to the class: `"A simple example class"`.

Class *instantiation* uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

```
x = MyClass()
```

creates a new *instance* of the class and assigns this object to the local variable `x`.

The instantiation operation ("calling" a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named `__init__()`, like this:

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

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly-created class instance. So in this example, a new, initialized instance can be obtained by:

```
x = MyClass()
```

Of course, the `__init__()` method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to `__init__()`. For example,

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3 Instance Objects

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names: data attributes and methods.

data attributes correspond to ”instance variables” in Smalltalk, and to ”data members” in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if `x` is the instance of `MyClass` created above, the following piece of code will print the value `16`, without leaving a trace:

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

The other kind of instance attribute reference is a *method*. A method is a function that ”belongs to” an object. (In Python, the term method is not unique to class instances: other object types can have methods as well. For example, list objects have methods called append, insert, remove, sort, and so on. However, in the following discussion, we'll use the term method exclusively to mean methods of class instance objects, unless explicitly stated otherwise.)

Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, `x.f` is a valid method reference, since `MyClass.f` is a function, but `x.i` is not, since `MyClass.i` is not. But `x.f` is not the same thing as `MyClass.f` --- it is a *method object*, not a function object.

9.3.4 Method Objects

Usually, a method is called right after it is bound:

```
x.f()
```

In the `MyClass` example, this will return the string `'hello world'`. However, it is not necessary to call a method right away: `x.f` is a method object, and can be stored away and called at a later time. For example:

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

will continue to print `hello world` until the end of time.

What exactly happens when a method is called? You may have noticed that `x.f()` was called without an argument above, even though the function definition for `f()` specified an argument. What happened to the argument? Surely Python raises an exception when a function that requires an argument is called without any --- even if the argument isn't actually used...

Actually, you may have guessed the answer: the special thing about methods is that the instance object is passed as the first argument of the function. In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`. In general, calling a method with a list of *n* arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's instance object before the first argument.

If you still don't understand how methods work, a look at the implementation can perhaps clarify matters. When a non-data attribute of an instance is referenced, the instance's class is searched. If the name denotes a valid class attribute that is a function object, a method object is created by packing (pointers to) the instance object and the function object just found together in an abstract object: this is the method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.

9.3.5 Class and Instance Variables

Generally speaking, instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class:

```
class Dog:

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

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

>>> 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'
```

As discussed in *A Word About Names and Objects*, shared data can have possibly surprising effects with involving *mutable* objects such as lists and dictionaries. For example, the *tricks* list in the following code should not be used as a class variable because just a single list would be shared by all *Dog* instances:

```
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')
```

(下页继续)

(繼續上一頁)

```
>>> d.tricks                 # unexpectedly shared by all dogs
['roll over', 'play dead']
```

Correct design of the class should use an instance variable instead:

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

9.4 Random Remarks

Data attributes override method attributes with the same name; to avoid accidental name conflicts, which may cause hard-to-find bugs in large programs, it is wise to use some kind of convention that minimizes the chance of conflicts. Possible conventions include capitalizing method names, prefixing data attribute names with a small unique string (perhaps just an underscore), or using verbs for methods and nouns for data attributes.

Data attributes may be referenced by methods as well as by ordinary users ("clients") of an object. In other words, classes are not usable to implement pure abstract data types. In fact, nothing in Python makes it possible to enforce data hiding --- it is all based upon convention. (On the other hand, the Python implementation, written in C, can completely hide implementation details and control access to an object if necessary; this can be used by extensions to Python written in C.)

Clients should use data attributes with care --- clients may mess up invariants maintained by the methods by stamping on their data attributes. Note that clients may add data attributes of their own to an instance object without affecting the validity of the methods, as long as name conflicts are avoided --- again, a naming convention can save a lot of headaches here.

There is no shorthand for referencing data attributes (or other methods!) from within methods. I find that this actually increases the readability of methods: there is no chance of confusing local variables and instance variables when glancing through a method.

Often, the first argument of a method is called `self`. This is nothing more than a convention: the name `self` has absolutely no special meaning to Python. Note, however, that by not following the convention your code may be less readable to other Python programmers, and it is also conceivable that a *class browser* program might be written that relies upon such a convention.

Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also ok. For example:

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

class C:
```

(下页继续)

(繼續上一頁)

```
    f = f1

    def g(self):
        return 'hello world'

    h = g
```

Now `f`, `g` and `h` are all attributes of class `C` that refer to function objects, and consequently they are all methods of instances of `C` --- `h` being exactly equivalent to `g`. Note that this practice usually only serves to confuse the reader of a program.

Methods may call other methods by using method attributes of the `self` argument:

```
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)
```

Methods may reference global names in the same way as ordinary functions. The global scope associated with a method is the module containing its definition. (A class is never used as a global scope.) While one rarely encounters a good reason for using global data in a method, there are many legitimate uses of the global scope: for one thing, functions and modules imported into the global scope can be used by methods, as well as functions and classes defined in it. Usually, the class containing the method is itself defined in this global scope, and in the next section we'll find some good reasons why a method would want to reference its own class.

Each value is an object, and therefore has a *class* (also called its *type*). It is stored as `object.__class__`.

9.5 Inheritance

Of course, a language feature would not be worthy of the name "class" without supporting inheritance. The syntax for a derived class definition looks like this:

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

The name `BaseClassName` must be defined in a scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

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

Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class.

There's nothing special about instantiation of derived classes: `DerivedClassName()` creates a new instance of the class. Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object.

Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively `virtual`.)

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call `BaseClassName.methodname(self, arguments)`. This is occasionally useful to clients as well. (Note that this only works if the base class is accessible as `BaseClassName` in the global scope.)

Python has two built-in functions that work with inheritance:

- Use `isinstance()` to check an instance's type: `isinstance(obj, int)` will be `True` only if `obj.__class__` is `int` or some class derived from `int`.
- Use `issubclass()` to check class inheritance: `issubclass(bool, int)` is `True` since `bool` is a subclass of `int`. However, `issubclass(float, int)` is `False` since `float` is not a subclass of `int`.

9.5.1 Multiple Inheritance

Python supports a form of multiple inheritance as well. A class definition with multiple base classes looks like this:

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

For most purposes, in the simplest cases, you can think of the search for attributes inherited from a parent class as depth-first, left-to-right, not searching twice in the same class where there is an overlap in the hierarchy. Thus, if an attribute is not found in `DerivedClassName`, it is searched for in `Base1`, then (recursively) in the base classes of `Base1`, and if it was not found there, it was searched for in `Base2`, and so on.

In fact, it is slightly more complex than that; the method resolution order changes dynamically to support cooperative calls to `super()`. This approach is known in some other multiple-inheritance languages as call-next-method and is more powerful than the super call found in single-inheritance languages.

Dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all classes inherit from `object`, so any case of multiple inheritance provides more than one path to reach `object`. To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance. For more detail, see https://www.python.org/download/releases/2.3/mro/.

9.6 Private Variables

"Private" instance variables that cannot be accessed except from inside an object don't exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. `_spam`) should be treated as a non-public part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.

Since there is a valid use-case for class-private members (namely to avoid name clashes of names with names defined by subclasses), there is limited support for such a mechanism, called *name mangling*. Any identifier of the form `__spam` (at least two leading underscores, at most one trailing underscore) is textually replaced with `_classname__spam`, where `classname` is the current class name with leading underscore(s) stripped. This

mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls. For example:

```
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)
```

The above example would work even if `MappingSubclass` were to introduce a `__update` identifier since it is replaced with `_Mapping__update` in the `Mapping` class and `_MappingSubclass__update` in the `MappingSubclass` class respectively.

Note that the mangling rules are designed mostly to avoid accidents; it still is possible to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger.

Notice that code passed to `exec()` or `eval()` does not consider the classname of the invoking class to be the current class; this is similar to the effect of the `global` statement, the effect of which is likewise restricted to code that is byte-compiled together. The same restriction applies to `getattr()`, `setattr()` and `delattr()`, as well as when referencing `__dict__` directly.

9.7 Odds and Ends

Sometimes it is useful to have a data type similar to the Pascal ”record” or C ”struct”, bundling together a few named data items. An empty class definition will do nicely:

```
class Employee:
    pass

john = Employee()  # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

A piece of Python code that expects a particular abstract data type can often be passed a class that emulates the methods of that data type instead. For instance, if you have a function that formats some data from a file object, you can define a class with methods `read()` and `readline()` that get the data from a string buffer instead, and pass it as an argument.

Instance method objects have attributes, too: `m.__self__` is the instance object with the method `m()`, and `m.__func__` is the function object corresponding to the method.

9.8 Iterators

By now you have probably noticed that most container objects can be looped over using a `for` statement:

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

This style of access is clear, concise, and convenient. The use of iterators pervades and unifies Python. Behind the scenes, the `for` statement calls `iter()` on the container object. The function returns an iterator object that defines the method `__next__()` which accesses elements in the container one at a time. When there are no more elements, `__next__()` raises a `StopIteration` exception which tells the `for` loop to terminate. You can call the `__next__()` method using the `next()` built-in function; this example shows how it all works:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> 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
```

Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes. Define an `__iter__()` method which returns an object with a `__next__()` method. If the class defines `__next__()`, then `__iter__()` can just return `self`:

```
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)
```

(下页继续)

(繼續上一頁)

```
...
m
a
p
s
```

9.9 Generators

*Generator*s are a simple and powerful tool for creating iterators. They are written like regular functions but use the `yield` statement whenever they want to return data. Each time `next()` is called on it, the generator resumes where it left off (it remembers all the data values and which statement was last executed). An example shows that generators can be trivially easy to create:

```
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
```

Anything that can be done with generators can also be done with class-based iterators as described in the previous section. What makes generators so compact is that the `__iter__()` and `__next__()` methods are created automatically.

Another key feature is that the local variables and execution state are automatically saved between calls. This made the function easier to write and much more clear than an approach using instance variables like `self.index` and `self.data`.

In addition to automatic method creation and saving program state, when generators terminate, they automatically raise `StopIteration`. In combination, these features make it easy to create iterators with no more effort than writing a regular function.

9.10 Generator Expressions

Some simple generators can be coded succinctly as expressions using a syntax similar to list comprehensions but with parentheses instead of square brackets. These expressions are designed for situations where the generator is used right away by an enclosing function. Generator expressions are more compact but less versatile than full generator definitions and tend to be more memory friendly than equivalent list comprehensions.

Examples:

```
>>> 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

>>> from math import pi, sin
>>> sine_table = {x: sin(x*pi/180) for x in range(0, 91)}
```

(下页继续)

(繼續上一頁)

```
>>> 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']
```

🄵解

CHAPTER 10

Python 標准函式庫概覽

10.1 作業系統介面

os 模組提供了數十個與作業系統溝通的函式:

```
>>> import os
>>> os.getcwd()      # Return the current working directory
'C:\\Python37'
>>> os.chdir('/server/accesslogs')   # Change current working directory
>>> os.system('mkdir today')   # Run the command mkdir in the system shell
0
```

務必使用 import os 而非 from os import *。這將避免因系統不同而實作有差🄵的 os.open() 覆蓋🄵建函式 open()。

在使用 os 諸如此類大型模組時搭配🄵建函式 dir() 和 help() 是非常有用的:

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

對於日常檔案和目🄵管理任務，shutil 模組提供了更容易使用的高階介面:

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

10.2 檔案之萬用字元

The `glob` module provides a function for making file lists from directory wildcard searches:

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

10.3 命令列引數

通用工具🄵本常需要處理命令列引數。這些引數會以串列形式存放在 `sys` 模組的 *argv* 此變數中。例如在命令列執行 `python demo.py one two three` 會有以下輸出結果:

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

The `argparse` module provides a more sophisticated mechanism to process command line arguments. The following script extracts one or more filenames and an optional number of lines to be displayed:

```
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)
```

When run at the command line with `python top.py --lines=5 alpha.txt beta.txt`, the script sets `args.lines` to `5` and `args.filenames` to `['alpha.txt', 'beta.txt']`.

10.4 錯誤輸出重新導向與程式終止

`sys` 模組也有 *stdin*，*stdout*，和 *stderr* 等變數。即使當 *stdout* 被重新導向時，後者 *stderr* 可輸出發送警告和錯誤訊息。

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

終止🄵本最直接的方式就是利用 `sys.exit()`。

10.5 字串樣式比對

`re` 模組提供正規表示式 (regular expression) 做進階的字串處理。當要處理🄵雜的比對以及操作時，正規表示式是簡潔且經過最佳化的解🄵方案。

```
>>> 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'
```

當只需要簡單的字串操作時，因🄵可讀性以及方便除錯，字串本身的方法是比較建議的。

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

10.6 數學相關

math 模組提供了 C 函式庫中底層的浮點數運算的函式。

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

random 模組提供了隨機選擇的工具。

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

statistics 模組提供了替數值資料計算基本統計量（包括平均、中位數、變🄴量數等）的功能。

```
>>> 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 專案 <https://scipy.org> 上也有許多數值計算相關的模組。

10.7 網路存取

Python 中有許多存取網路以及處理網路協定。最簡單的兩個例子包括 urllib.request 模組可以從網址抓取資料以及 smtplib 可以用來寄郵件。：

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

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

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
```

(下页继续)

(繼續上一頁)

```
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(注意第二個例子中需要在本地端執行一個郵件伺服器)

10.8 日期與時間

datetime 模組中有許多類🄵供以操作日期以及時間，從簡單從🄵雜都有。模組支援日期與時間的運算，而實作的重點是有效率的成員🄵取以達到輸出格式化以及操作。模組也提供支援時區🄵算的類🄵。

```
>>> # 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 資料壓縮

常見的解壓縮以及壓縮格式都有直接支援。包括：zlib，gzip，bz2，lzma，zipfile 以及 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 效能量測

有一些 Python 使用者很想了解同個問題的不同實作方法的效能差🄵。Python 提供評估了效能差🄵的工具。

舉例來🄵，有人可能會試著用 tuple 的打包機制來交🄵引數代替傳統的方式。timeit 模組可以迅速地展示效能的進步。

```
>>> 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
```

相對於 timeit 模組提供這🄵細的粒度，profile 模組以及 pstats 模組則提供了一些在大型的程式碼識🄵關鍵臨界區間（Critical Section）的工具。

10.11 品質控管

達到高品質軟體的一個方法當開發時對每個函式寫測試以及在開發過程中要不斷的跑這些測試。

doctest 模組提供了一個工具，掃描模組🄵根據程式中🄵嵌的文件字符串執行測試。測試構造如同簡單的將它的輸出結果剪下🄵貼上到文件字符串中。通過用🄵提供的例子，它🄵化了文件，允許 doctest 模塊組認代碼的結果是否與文件一致:

```
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 模組不像 doctest 模組這般容易，但是它提供了更完整的測試集🄵且可以整合在不同的檔案間。

```
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 標准模組庫

”batteries included” 是 Python 設計哲學。它的好處是可以透過這些套件使用🄵雜與🄵大的功能。例如:

- 使用 `xmlrpc.client` 和 `xmlrpc.server` 模組實現遠端控制看似變更🄴容易。使用前也不需要先了解相關知識或是掌握 XML 的技能就能直接透過名稱使用模組。
- 函式庫 `email` 套件用來管理 MIME 和其他 **RFC 2822** 相關電子郵件訊息的文件。相🄵於其他電子郵件套件 `smtplib` 和 `poplib` 這些實際用來發送與接收訊息，擁有更完整的工具設置提供建置與解析🄵雜訊息的結構（包含附件檔案）和實現網路傳送之間的解碼與標頭協定。
- 函式庫 `json` 套件提供 JSON 資料解析🄵大的交🄵格式。`csv` 模組則提供直接讀寫以逗號分隔值的檔案格式，支援一般資料庫與電子表格。`xml.etree.ElementTree` ， `xml.dom` 與 `xml.sax` 套件則支援 XML 流程。綜觀所有，這些模組和套件都簡化了 Python 應用程式與其他工具之間的資料交🄵。
- `sqllite3` 套件作🄴包覆 SQLite 資料庫的函式庫，提供一個一致性的資料庫用來更新與操作使用些微非標准的 SQL 語法。
- 有數種支援國際化模組 `gettext`，`locale`，和 `codecs` 等套件。

CHAPTER 11

Brief Tour of the Standard Library --- Part II

第二部分涵蓋更多支援專業程式設計所需要的進階模組。這些模組很少出現在小🄵本中。

11.1 Output Formatting

The `reprlib` module provides a version of `repr()` customized for abbreviated displays of large or deeply nested containers:

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

The `pprint` module offers more sophisticated control over printing both built-in and user defined objects in a way that is readable by the interpreter. When the result is longer than one line, the "pretty printer" adds line breaks and indentation to more clearly reveal data structure:

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

The `textwrap` module formats paragraphs of text to fit a given screen width:

```
>>> import textwrap
>>> doc = """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."""
...
>>> print(textwrap.fill(doc, width=40))
The wrap() method is just like fill()
except that it returns a list of strings
```

(下页继续)

(繼續上一頁)

```
instead of one big string with newlines
to separate the wrapped lines.
```

The `locale` module accesses a database of culture specific data formats. The grouping attribute of locale's format function provides a direct way of formatting numbers with group separators:

```
>>> 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("%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 Templating

The `string` module includes a versatile `Template` class with a simplified syntax suitable for editing by end-users. This allows users to customize their applications without having to alter the application.

The format uses placeholder names formed by `$` with valid Python identifiers (alphanumeric characters and underscores). Surrounding the placeholder with braces allows it to be followed by more alphanumeric letters with no intervening spaces. Writing `$$` creates a single escaped `$`:

```
>>> 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.'
```

The `substitute()` method raises a `KeyError` when a placeholder is not supplied in a dictionary or a keyword argument. For mail-merge style applications, user supplied data may be incomplete and the `safe_substitute()` method may be more appropriate --- it will leave placeholders unchanged if data is missing:

```
>>> 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.'
```

Template subclasses can specify a custom delimiter. For example, a batch renaming utility for a photo browser may elect to use percent signs for placeholders such as the current date, image sequence number, or file format:

```
>>> 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)
```

(下页继续)

(繼續上一頁)

```
...     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
```

Another application for templating is separating program logic from the details of multiple output formats. This makes it possible to substitute custom templates for XML files, plain text reports, and HTML web reports.

11.3 Working with Binary Data Record Layouts

The `struct` module provides `pack()` and `unpack()` functions for working with variable length binary record formats. The following example shows how to loop through header information in a ZIP file without using the `zipfile` module. Pack codes `"H"` and `"I"` represent two and four byte unsigned numbers respectively. The `"<"` indicates that they are standard size and in little-endian byte order:

```
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 Multi-threading

Threading is a technique for decoupling tasks which are not sequentially dependent. Threads can be used to improve the responsiveness of applications that accept user input while other tasks run in the background. A related use case is running I/O in parallel with computations in another thread.

The following code shows how the high level `threading` module can run tasks in background while the main program continues to run:

```
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)
```

(下页继续)

(繼續上一頁)

```
        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.')

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

The principal challenge of multi-threaded applications is coordinating threads that share data or other resources. To that end, the threading module provides a number of synchronization primitives including locks, events, condition variables, and semaphores.

While those tools are powerful, minor design errors can result in problems that are difficult to reproduce. So, the preferred approach to task coordination is to concentrate all access to a resource in a single thread and then use the `queue` module to feed that thread with requests from other threads. Applications using `Queue` objects for inter-thread communication and coordination are easier to design, more readable, and more reliable.

11.5 Logging

The `logging` module offers a full featured and flexible logging system. At its simplest, log messages are sent to a file or to `sys.stderr`:

```
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')
```

This produces the following output:

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

By default, informational and debugging messages are suppressed and the output is sent to standard error. Other output options include routing messages through email, datagrams, sockets, or to an HTTP Server. New filters can select different routing based on message priority: `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `CRITICAL`.

The logging system can be configured directly from Python or can be loaded from a user editable configuration file for customized logging without altering the application.

11.6 Weak References

Python does automatic memory management (reference counting for most objects and *garbage collection* to eliminate cycles). The memory is freed shortly after the last reference to it has been eliminated.

This approach works fine for most applications but occasionally there is a need to track objects only as long as they are being used by something else. Unfortunately, just tracking them creates a reference that makes them permanent. The `weakref` module provides tools for tracking objects without creating a reference. When the object is no longer needed, it is automatically removed from a weakref table and a callback is triggered for weakref objects. Typical applications include caching objects that are expensive to create:

```
>>> 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:/python37/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7 Tools for Working with Lists

Many data structure needs can be met with the built-in list type. However, sometimes there is a need for alternative implementations with different performance trade-offs.

The `array` module provides an `array()` object that is like a list that stores only homogeneous data and stores it more compactly. The following example shows an array of numbers stored as two byte unsigned binary numbers (typecode `"H"`) rather than the usual 16 bytes per entry for regular lists of Python int objects:

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

The `collections` module provides a `deque()` object that is like a list with faster appends and pops from the left side but slower lookups in the middle. These objects are well suited for implementing queues and breadth first tree searches:

```
>>> 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)
```

In addition to alternative list implementations, the library also offers other tools such as the `bisect` module with functions for manipulating sorted lists:

```
>>> 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')]
```

The heapq module provides functions for implementing heaps based on regular lists. The lowest valued entry is always kept at position zero. This is useful for applications which repeatedly access the smallest element but do not want to run a full list sort:

```
>>> 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 Decimal Floating Point Arithmetic

The decimal module offers a Decimal datatype for decimal floating point arithmetic. Compared to the built-in float implementation of binary floating point, the class is especially helpful for

- financial applications and other uses which require exact decimal representation,
- control over precision,
- control over rounding to meet legal or regulatory requirements,
- tracking of significant decimal places, or
- applications where the user expects the results to match calculations done by hand.

For example, calculating a 5% tax on a 70 cent phone charge gives different results in decimal floating point and binary floating point. The difference becomes significant if the results are rounded to the nearest cent:

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

The Decimal result keeps a trailing zero, automatically inferring four place significance from multiplicands with two place significance. Decimal reproduces mathematics as done by hand and avoids issues that can arise when binary floating point cannot exactly represent decimal quantities.

Exact representation enables the Decimal class to perform modulo calculations and equality tests that are unsuitable for binary floating point:

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

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

The decimal module provides arithmetic with as much precision as needed:

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

CHAPTER 12

🄵擬環境與套件

12.1 簡介

Python 應用程式通常會用到不在標准函式庫的套件和模組。應用程式有時候會需要某個特定版本的函式庫，因🄴這個應用程式可能需要某個特殊的臭蟲修正，或是這個應用程式是根據該函式庫特定版本的介面所撰寫。

這意味著不太可能安裝一套 Python 就可以滿足所有應用程式的要求。如果應用程式 A 需要一個特定的模組的 1.0 版，但另外一個應用程式 B 需要 2.0 版，那🄴這整個需求不管安裝 1.0 或是 2.0 都會衝突，以致於應用程式無法使用。

解🄵方案是創建一個🄵擬環境（*virtual environment*），這是一個獨立的資料夾，🄵且🄴面裝好了特定版本的 Python，以及一系列相關的套件。

不同的應用程式可以使用不同的🄵擬環境。以前述中需要被解🄵的例子中，應用程式 A 能🄵擁有它自己的🄵擬環境，🄵且是裝好 1.0 版，然而應用程式 B 則可以是用另外一個有 2.0 版的🄵擬環境。要是應用程式 B 需要某個函式庫被升級到 3.0 版，這🄴不會影響到應用程式 A 的環境。

12.2 建立🄵擬環境

用來建立與管理🄵擬環境的模組叫做 `venv`。`venv` 通常會安裝你能🄴取得的最新版本的 Python。要是你的系統有不同版本的 Python，你可以透過 `python3` 這個指令選擇特定或是任意版本的 Python。

在建立🄵擬環境的時候，在你🄵定要放該🄵擬環境的資料夾之後，在 script 中執行 `venv` 模組🄵且給定資料夾 path：

```
python3 -m venv tutorial-env
```

如果 `tutorial-env` 不存在的話，這會建立 `tutorial-env` 資料夾，🄵且也會在🄵面建立一個有 Python 直譯器的🄴本、標准函式庫、以及不同的支援檔案的資料夾。

一旦你建立了一個🄵擬環境，你可以🄵動他。

在 Windows 系統中，使用：

```
tutorial-env\Scripts\activate.bat
```

在 Unix 或 MacOS 系統，使用：

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

(這段程式碼適用於 bash shell。如果你是用 **csh** 或者 **fish** shell，應當使用替代的 `activate.csh` 與 `activate.fish` 🄵本。)

🄵動🄵擬環境會改變你的 shell 提示字元來顯示你正在使用的🄵擬環境，🄵且修改環境以讓你在執行 `python` 的時候可以得到特定的 Python 版本，例如🄵:

```
$ 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']
>>>
```

12.3 用 pip 管理套件

你可以使用一個叫做 **pip** 的程式來安裝、升級和移除套件。`pip` 預設會從 Python Package Index <https://pypi.org> 安裝套件。你可以透過你的🄵覽器🄵覽 Python Package Index，或是使用 `pip` 的限定搜索功能:

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

`pip` 有好幾個子指令: "search"、"install"、"uninstall"、"freeze" 等等。(這可以參考 installing-index 🄵明書來取得 `pip` 的完整文件🄵明。)

你可以透過指定套件名字來安裝最新版本的套件:

```
(tutorial-env) $ 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
```

你也可以透過在套件名稱之後接上 == 和版號來指定特定版本:

```
(tutorial-env) $ 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
```

要是你重新執行此指令，`pip` 會知道該版本已經安裝過，然後什🄵也不做。你可以提供不同的版本號碼來取得該版本，或是可以執行 `pip install --upgrade` 來把套件升級到最新的版本:

```
(tutorial-env) $ pip install --upgrade requests
Collecting requests
Installing collected packages: requests
```

(下页继续)

(繼續上一頁)

```
  Found existing installation: requests 2.6.0
    Uninstalling requests-2.6.0:
      Successfully uninstalled requests-2.6.0
Successfully installed requests-2.7.0
```

`pip uninstall` 後面接一個或是多個套件名稱可以從🄵擬環境中移除套件。

`pip show` 可以顯示一個特定套件的資訊：

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

`pip list` 會顯示🄵擬環境中所有已經安裝的套件：

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

`pip freeze` 可以🄵🄵一整個已經安裝的套件清單，但是輸出使用 `pip install` 可以讀懂的格式。一個常見的慣例是放這整個清單到一個叫做 `requirements.txt` 的檔案：

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

`requirements.txt` 可以提交到版本控制，🄵且作🄵釋出應用程式的一部分。使用者可以透過 `install -r` 安裝對應的的套件：

```
(tutorial-env) $ 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` 還有更多功能。可以參考 installing-index 🄵明書來取得完整的 `pip` 參考資料。當你撰寫了一個套件🄵且想要讓它可以在 Python Package Index 上可以取得的話，可以參考 distributing-index 🄵明。

CHAPTER 13

現在可以來學習些什[F]?

[F]讀本教學可能增[F]您對於使用 Python 的興趣—您應該非常渴望使用 Python 來解[F]在現實生活中所遭遇的問題。該從哪[F]學習更多呢?

本教學是 Python 文件中的一部分。這份文件集[F]頭的其他文件包含:

- library-index:

  你該好好的[F]覽這份手[F],它提供了完整的(但簡潔)參考素材像是型[F]、函式與標准函式庫[F]的模組。標准的 Python 發行版本會包含大量的附加程式碼。有些模組可以讀取 Unix 信箱、通過 HTTP 來檢索文件、[F]生亂數、分析命令列選項、編寫 CGI 程式、壓縮資料、及許多其他任務。[F]覽函式庫參考手[F]可以讓你了解有哪些模組可以用。

- installing-index:[F]明與解釋如何安裝其他 Python 使用者所編寫的模組。

- reference-index:Python 語法以及語意的詳細[F]明。這份文件[F]讀起來會有些吃力,但作[F]一個語言本身的完整指南是非常有用的。

更多 Python 的資源:

- https://www.python.org:Python 的主要網站。它包含程式碼、文件以及連結到 Python 相關聯的網頁。網站[F]鏡像的[F]設置於世界各地,像是歐洲、日本以及澳大利亞;鏡像網站也許會比主網站來得更快,不過具體速度則還是取[F]於你所在的地理位置。

- https://docs.python.org:快速訪問 Python 的文件。

- https://pypi.org: The Python Package Index, previously also nicknamed the Cheese Shop[1], is an index of user-created Python modules that are available for download. Once you begin releasing code, you can register it here so that others can find it.

- https://code.activestate.com/recipes/langs/python/:Python Cookbook 是一個相當大的程式碼範例集,大量的模組以及有用的[F]本。一些值得注意與特[F]貢獻則被收集在一本名[F] Python Cookbook (O'Reilly & Associates, ISBN 0-596-00797-3.) 的書籍中。

- http://www.pyvideo.org 從研討會與使用者群組聚會[F]所收集與 Python 相關的影片連結。

- https://scipy.org:The Scientific Python 專案是一個包含用於高速陣列運算與操作的模組,以及用於如[F]性代數、傅利葉變[F]、非[F]性求解器、隨機數分[F]、統計分析等一系列的套件。

對於 Python 相關的疑問與問題回報,您可以張貼到新聞群組 `comp.lang.python`,或將它們寄至 python-list@python.org 的郵寄清單(mailing list)。新聞群組和郵寄清單是個閘道,因此張貼到其中的郵件

[1] "Cheese Shop" is a Monty Python's sketch: a customer enters a cheese shop, but whatever cheese he asks for, the clerk says it's missing.

都將自動轉發給另一個。每天會有數以百計的F容，詢問（和回答）問題、建議新功能與發F新的模組。郵寄清單會存檔在 https://mail.python.org/pipermail/。

在張貼之前，請先確認問題是否在常見問題（也被稱 FAQ）這個清單F。FAQ 會回答出現很多次的問題及解答，有很多問題甚至已經包含解F問題的方法。

CHAPTER 14

Interactive Input Editing and History Substitution

Some versions of the Python interpreter support editing of the current input line and history substitution, similar to facilities found in the Korn shell and the GNU Bash shell. This is implemented using the GNU Readline library, which supports various styles of editing. This library has its own documentation which we won't duplicate here.

14.1 Tab Completion and History Editing

Completion of variable and module names is automatically enabled at interpreter startup so that the `Tab` key invokes the completion function; it looks at Python statement names, the current local variables, and the available module names. For dotted expressions such as `string.a`, it will evaluate the expression up to the final `'.'` and then suggest completions from the attributes of the resulting object. Note that this may execute application-defined code if an object with a `__getattr__()` method is part of the expression. The default configuration also saves your history into a file named `.python_history` in your user directory. The history will be available again during the next interactive interpreter session.

14.2 Alternatives to the Interactive Interpreter

This facility is an enormous step forward compared to earlier versions of the interpreter; however, some wishes are left: It would be nice if the proper indentation were suggested on continuation lines (the parser knows if an indent token is required next). The completion mechanism might use the interpreter's symbol table. A command to check (or even suggest) matching parentheses, quotes, etc., would also be useful.

One alternative enhanced interactive interpreter that has been around for quite some time is IPython, which features tab completion, object exploration and advanced history management. It can also be thoroughly customized and embedded into other applications. Another similar enhanced interactive environment is bpython.

CHAPTER 15

浮點數運算：問題與限制

在計算機架構中，浮點數透過二進位小數表示。例如🄵，在十進位小數中：

```
0.125
```

可被分🄵 1/10 + 2/100 + 5/1000，同樣的道理，二進位小數：

```
0.001
```

可被分🄵 0/2 + 0/4 + 1/8。這兩個小數有相同的數值，而唯一真正的不同在於前者以十進位表示，後者以二進位表示。

不幸的是，大多數十進位小數無法精准地以二進位小數表示。一般的結果🄵，您輸入的十進位浮點數由實際存在計算機中的二進位浮點數近似。

在十進位中，這個問題更容易被理解。以分數 1/3 🄵例，您可以將其近似🄵十進位小數：

```
0.3
```

或者，更好的近似：

```
0.33
```

或者，更好的近似：

```
0.333
```

依此類推，不論你使用多少位數表示小數，最後的結果都無法精准地表示 1/3，但你還是能越來越精准地表示 1/3。

同樣的道理，不論你願意以多少位數表示二進位小數，十進位小數 0.1 都無法被二進位小數精准地表達。在二進位小數中，1/10 會是一個無限循環小數：

```
0.0001100110011001100110011001100110011001100110011001100110011...
```

只要您停在任何有限的位數，您就只會得到近似值。而現在大多數的計算機中，浮點數是透過二進位分數近似的，其中分子從最高有效位元使開始用 53 個位元表示，分母則是以二🄵底的指數。在 1/10 的例子中，二進位分數🄵 `3602879701896397 / 2 ** 55`，而這樣的表示十分地接近，但不完全等同於 1/10 的真正數值。

由於數值顯示的方式，很多使用者𝔽𝔽有發現數值是個近似值。Python 只會印出一個十進位近似值，其近似了儲存在計算機中的二進位近似值的十進位數值。在大多數的計算機中，如果 Python 真的會印出完整的十進位數值，其表示儲存在計算機中的 0.1 的二進位近似值，它將顯示𝔽：

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

這比一般人感到有用的位數還多，所以 Python 將位數保持在可以接受的範圍，只顯示舍入後的數值：

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

一定要記住，雖然印出的數字看起來是精准的 1/10，但真正儲存的數值是能表示的二進位分數中，最接近精准數值的數。

有趣的是，有許多不同的十進位數，共用同一個最接近的二進位近似小數。例如𝔽：數字 `0.1` 和 `0.10000000000000001` 和 `0.1000000000000000055511151231257827021181583404541015625`,都由 `3602879701896397 / 2 ** 55` 近似。由於這三個數值共用同一個近似值，任何一個數值都可以被顯示，同時保持 `eval(repr(x)) == x`。

歷史上，Python 的提示字元 (prompt) 與𝔽建的 `repr()` 函式會選擇上段𝔽明中有 17 個有效位元的數：`0.10000000000000001`。從 Python 3.1 版開始，Python（在大部分的系統上）可以選擇其中最短的數𝔽簡單地顯示𝔽 `0.1`。

注意，這是二進位浮點數理所當然的特性，𝔽不是 Python 的錯誤 (bug)，更不是您程式碼的錯誤。只要有程式語言支持硬體的浮點數運算，您將會看到同樣的事情出現在其中（雖然某些程式語言預設不顯示差𝔽，或者預設全部輸出）。

𝔽求更優雅的輸出，您可能想要使用字串的格式化 (string formatting) 𝔽生限定的有效位數：

```
>>> 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'
```

要了解一件很重要的事，在真正意義上，浮點數的表示是一種幻覺：你基本上在舍入真正機器數值所展示的值。

這種幻覺可能會𝔽生下一個幻覺。舉例來𝔽，因𝔽 0.1 不是真正的 1/10，把三個 0.1 的值相加，也不會𝔽生精准的 0.3：

```
>>> .1 + .1 + .1 == .3
False
```

同時，因𝔽 0.1 不能再更接近精准的 1/10，還有 0.3 不能再更接近精准的 3/10，預先用 `round()` 函式舍入𝔽不會有幫助：

```
>>> round(.1, 1) + round(.1, 1) + round(.1, 1) == round(.3, 1)
False
```

雖然數字不會再更接近他們的精准數值，但 `round()` 函式可以對事後的舍入有所幫助，如此一來，不精確的數值就變得可以互相比較：

```
>>> round(.1 + .1 + .1, 10) == round(.3, 10)
True
```

二進位浮點數架構擁有很多這樣的驚喜。底下的「表示法錯誤」章節，詳細的解釋了「0.1」的問題。如果想要其他常見驚喜更完整的描述，可以參考 The Perils of Floating Point（浮點數的風險）。

As that says near the end, ”there are no easy answers.” Still, don’t be unduly wary of floating-point! The errors in Python float operations are inherited from the floating-point hardware, and on most machines are on the order of no more than 1 part in 2**53 per operation. That’s more than adequate for most tasks, but you do need to keep in mind that it’s not decimal arithmetic and that every float operation can suffer a new rounding error.

While pathological cases do exist, for most casual use of floating-point arithmetic you’ll see the result you expect in the end if you simply round the display of your final results to the number of decimal digits you expect. `str()` usually suffices, and for finer control see the `str.format()` method’s format specifiers in formatstrings.

For use cases which require exact decimal representation, try using the `decimal` module which implements decimal arithmetic suitable for accounting applications and high-precision applications.

Another form of exact arithmetic is supported by the `fractions` module which implements arithmetic based on rational numbers (so the numbers like 1/3 can be represented exactly).

If you are a heavy user of floating point operations you should take a look at the Numerical Python package and many other packages for mathematical and statistical operations supplied by the SciPy project. See <https://scipy.org>.

Python provides tools that may help on those rare occasions when you really *do* want to know the exact value of a float. The `float.as_integer_ratio()` method expresses the value of a float as a fraction:

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

Since the ratio is exact, it can be used to losslessly recreate the original value:

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

The `float.hex()` method expresses a float in hexadecimal (base 16), again giving the exact value stored by your computer:

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

This precise hexadecimal representation can be used to reconstruct the float value exactly:

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

Since the representation is exact, it is useful for reliably porting values across different versions of Python (platform independence) and exchanging data with other languages that support the same format (such as Java and C99).

Another helpful tool is the `math.fsum()` function which helps mitigate loss-of-precision during summation. It tracks ”lost digits” as values are added onto a running total. That can make a difference in overall accuracy so that the errors do not accumulate to the point where they affect the final total:

```
>>> sum([0.1] * 10) == 1.0
False
>>> math.fsum([0.1] * 10) == 1.0
True
```

15.1 Representation Error

This section explains the ”0.1” example in detail, and shows how you can perform an exact analysis of cases like this yourself. Basic familiarity with binary floating-point representation is assumed.

Representation error refers to the fact that some (most, actually) decimal fractions cannot be represented exactly as binary (base 2) fractions. This is the chief reason why Python (or Perl, C, C++, Java, Fortran, and many others) often won't display the exact decimal number you expect.

Why is that? 1/10 is not exactly representable as a binary fraction. Almost all machines today (November 2000) use IEEE-754 floating point arithmetic, and almost all platforms map Python floats to IEEE-754 ”double precision”. 754 doubles contain 53 bits of precision, so on input the computer strives to convert 0.1 to the closest fraction it can of the form *J*/2***N** where *J* is an integer containing exactly 53 bits. Rewriting

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

as

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

and recalling that *J* has exactly 53 bits (is `>= 2**52` but `< 2**53`), the best value for *N* is 56:

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

That is, 56 is the only value for *N* that leaves *J* with exactly 53 bits. The best possible value for *J* is then that quotient rounded:

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

Since the remainder is more than half of 10, the best approximation is obtained by rounding up:

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

Therefore the best possible approximation to 1/10 in 754 double precision is:

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

Dividing both the numerator and denominator by two reduces the fraction to:

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

Note that since we rounded up, this is actually a little bit larger than 1/10; if we had not rounded up, the quotient would have been a little bit smaller than 1/10. But in no case can it be *exactly* 1/10!

So the computer never ”sees” 1/10: what it sees is the exact fraction given above, the best 754 double approximation it can get:

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

If we multiply that fraction by 10**55, we can see the value out to 55 decimal digits:

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

meaning that the exact number stored in the computer is equal to the decimal value 0.1000000000000000055511151231257827021181583404541015625. Instead of displaying the full decimal value, many languages (including older versions of Python), round the result to 17 significant digits:

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

The `fractions` and `decimal` modules make these calculations easy:

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

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

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

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

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

CHAPTER 16

附F

16.1 互動模式

16.1.1 Error Handling

When an error occurs, the interpreter prints an error message and a stack trace. In interactive mode, it then returns to the primary prompt; when input came from a file, it exits with a nonzero exit status after printing the stack trace. (Exceptions handled by an `except` clause in a `try` statement are not errors in this context.) Some errors are unconditionally fatal and cause an exit with a nonzero exit; this applies to internal inconsistencies and some cases of running out of memory. All error messages are written to the standard error stream; normal output from executed commands is written to standard output.

Typing the interrupt character (usually `Control-C` or `Delete`) to the primary or secondary prompt cancels the input and returns to the primary prompt.[1] Typing an interrupt while a command is executing raises the `KeyboardInterrupt` exception, which may be handled by a `try` statement.

16.1.2 Executable Python Scripts

On BSD'ish Unix systems, Python scripts can be made directly executable, like shell scripts, by putting the line

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

(assuming that the interpreter is on the user's `PATH`) at the beginning of the script and giving the file an executable mode. The `#!` must be the first two characters of the file. On some platforms, this first line must end with a Unix-style line ending (`'\n'`), not a Windows (`'\r\n'`) line ending. Note that the hash, or pound, character, `'#'`, is used to start a comment in Python.

The script can be given an executable mode, or permission, using the **`chmod`** command.

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

On Windows systems, there is no notion of an "executable mode". The Python installer automatically associates `.py` files with `python.exe` so that a double-click on a Python file will run it as a script. The extension can also be `.pyw`, in that case, the console window that normally appears is suppressed.

[1] A problem with the GNU Readline package may prevent this.

16.1.3 The Interactive Startup File

When you use Python interactively, it is frequently handy to have some standard commands executed every time the interpreter is started. You can do this by setting an environment variable named `PYTHONSTARTUP` to the name of a file containing your start-up commands. This is similar to the `.profile` feature of the Unix shells.

This file is only read in interactive sessions, not when Python reads commands from a script, and not when `/dev/tty` is given as the explicit source of commands (which otherwise behaves like an interactive session). It is executed in the same namespace where interactive commands are executed, so that objects that it defines or imports can be used without qualification in the interactive session. You can also change the prompts `sys.ps1` and `sys.ps2` in this file.

If you want to read an additional start-up file from the current directory, you can program this in the global start-up file using code like `if os.path.isfile('.pythonrc.py'): exec(open('.pythonrc.py').read())`. If you want to use the startup file in a script, you must do this explicitly in the script:

```
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 The Customization Modules

Python provides two hooks to let you customize it: `sitecustomize` and `usercustomize`. To see how it works, you need first to find the location of your user site-packages directory. Start Python and run this code:

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

Now you can create a file named `usercustomize.py` in that directory and put anything you want in it. It will affect every invocation of Python, unless it is started with the `-s` option to disable the automatic import.

`sitecustomize` works in the same way, but is typically created by an administrator of the computer in the global site-packages directory, and is imported before `usercustomize`. See the documentation of the `site` module for more details.

🄵解

APPENDIX A

Glossary

>>> The default Python prompt of the interactive shell. Often seen for code examples which can be executed interactively in the interpreter.

... The default Python prompt of the interactive shell when entering the code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.

2to3 A tool that tries to convert Python 2.x code to Python 3.x code by handling most of the incompatibilities which can be detected by parsing the source and traversing the parse tree.

2to3 is available in the standard library as `lib2to3`; a standalone entry point is provided as `Tools/scripts/2to3`. See 2to3-reference.

abstract base class Abstract base classes complement *duck-typing* by providing a way to define interfaces when other techniques like `hasattr()` would be clumsy or subtly wrong (for example with magic methods). ABCs introduce virtual subclasses, which are classes that don't inherit from a class but are still recognized by `isinstance()` and `issubclass()`; see the `abc` module documentation. Python comes with many built-in ABCs for data structures (in the `collections.abc` module), numbers (in the `numbers` module), streams (in the `io` module), import finders and loaders (in the `importlib.abc` module). You can create your own ABCs with the `abc` module.

annotation A label associated with a variable, a class attribute or a function parameter or return value, used by convention as a *type hint*.

Annotations of local variables cannot be accessed at runtime, but annotations of global variables, class attributes, and functions are stored in the `__annotations__` special attribute of modules, classes, and functions, respectively.

See *variable annotation*, *function annotation*, **PEP 484** and **PEP 526**, which describe this functionality.

argument A value passed to a *function* (or *method*) when calling the function. There are two kinds of argument:

- *keyword argument*: an argument preceded by an identifier (e.g. `name=`) in a function call or passed as a value in a dictionary preceded by `**`. For example, `3` and `5` are both keyword arguments in the following calls to `complex()`:

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

- *positional argument*: an argument that is not a keyword argument. Positional arguments can appear at the beginning of an argument list and/or be passed as elements of an *iterable* preceded by `*`. For example, `3`

and `5` are both positional arguments in the following calls:

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

Arguments are assigned to the named local variables in a function body. See the calls section for the rules governing this assignment. Syntactically, any expression can be used to represent an argument; the evaluated value is assigned to the local variable.

See also the *parameter* glossary entry, the FAQ question on the difference between arguments and parameters, and **PEP 362**.

asynchronous context manager An object which controls the environment seen in an `async with` statement by defining `__aenter__()` and `__aexit__()` methods. Introduced by **PEP 492**.

asynchronous generator A function which returns an *asynchronous generator iterator*. It looks like a coroutine function defined with `async def` except that it contains `yield` expressions for producing a series of values usable in an `async for` loop.

Usually refers to an asynchronous generator function, but may refer to an *asynchronous generator iterator* in some contexts. In cases where the intended meaning isn't clear, using the full terms avoids ambiguity.

An asynchronous generator function may contain `await` expressions as well as `async for`, and `async with` statements.

asynchronous generator iterator An object created by a *asynchronous generator* function.

This is an *asynchronous iterator* which when called using the `__anext__()` method returns an awaitable object which will execute the body of the asynchronous generator function until the next `yield` expression.

Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the *asynchronous generator iterator* effectively resumes with another awaitable returned by `__anext__()`, it picks up where it left off. See **PEP 492** and **PEP 525**.

asynchronous iterable An object, that can be used in an `async for` statement. Must return an *asynchronous iterator* from its `__aiter__()` method. Introduced by **PEP 492**.

asynchronous iterator An object that implements the `__aiter__()` and `__anext__()` methods. `__anext__` must return an *awaitable* object. `async for` resolves the awaitables returned by an asynchronous iterator's `__anext__()` method until it raises a `StopAsyncIteration` exception. Introduced by **PEP 492**.

attribute A value associated with an object which is referenced by name using dotted expressions. For example, if an object *o* has an attribute *a* it would be referenced as *o.a*.

awaitable An object that can be used in an `await` expression. Can be a *coroutine* or an object with an `__await__()` method. See also **PEP 492**.

BDFL Benevolent Dictator For Life, a.k.a. Guido van Rossum, Python's creator.

binary file A *file object* able to read and write *bytes-like objects*. Examples of binary files are files opened in binary mode (`'rb'`, `'wb'` or `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer`, and instances of `io.BytesIO` and `gzip.GzipFile`.

See also *text file* for a file object able to read and write `str` objects.

bytes-like object An object that supports the bufferobjects and can export a C-*contiguous* buffer. This includes all `bytes`, `bytearray`, and `array.array` objects, as well as many common `memoryview` objects. Bytes-like objects can be used for various operations that work with binary data; these include compression, saving to a binary file, and sending over a socket.

Some operations need the binary data to be mutable. The documentation often refers to these as "read-write bytes-like objects". Example mutable buffer objects include `bytearray` and a `memoryview` of a `bytearray`. Other operations require the binary data to be stored in immutable objects ("read-only bytes-like objects"); examples of these include `bytes` and a `memoryview` of a `bytes` object.

bytecode Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter. The bytecode is also cached in `.pyc` files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This "intermediate language" is said to run on a *virtual machine* that executes the machine code corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.

A list of bytecode instructions can be found in the documentation for the dis module.

class A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class.

class variable A variable defined in a class and intended to be modified only at class level (i.e., not in an instance of the class).

coercion The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer `3`, but in `3+4.5`, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a `TypeError`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.

complex number An extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part. Imaginary numbers are real multiples of the imaginary unit (the square root of `-1`), often written `i` in mathematics or `j` in engineering. Python has built-in support for complex numbers, which are written with this latter notation; the imaginary part is written with a `j` suffix, e.g., `3+1j`. To get access to complex equivalents of the `math` module, use `cmath`. Use of complex numbers is a fairly advanced mathematical feature. If you're not aware of a need for them, it's almost certain you can safely ignore them.

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

context variable A variable which can have different values depending on its context. This is similar to Thread-Local Storage in which each execution thread may have a different value for a variable. However, with context variables, there may be several contexts in one execution thread and the main usage for context variables is to keep track of variables in concurrent asynchronous tasks. See `contextvars`.

contiguous A buffer is considered contiguous exactly if it is either *C-contiguous* or *Fortran contiguous*. Zero-dimensional buffers are C and Fortran contiguous. In one-dimensional arrays, the items must be laid out in memory next to each other, in order of increasing indexes starting from zero. In multidimensional C-contiguous arrays, the last index varies the fastest when visiting items in order of memory address. However, in Fortran contiguous arrays, the first index varies the fastest.

coroutine Coroutines are a more generalized form of subroutines. Subroutines are entered at one point and exited at another point. Coroutines can be entered, exited, and resumed at many different points. They can be implemented with the `async def` statement. See also **PEP 492**.

coroutine function A function which returns a *coroutine* object. A coroutine function may be defined with the `async def` statement, and may contain `await`, `async for`, and `async with` keywords. These were introduced by **PEP 492**.

CPython The canonical implementation of the Python programming language, as distributed on python.org. The term "CPython" is used when necessary to distinguish this implementation from others such as Jython or IronPython.

decorator A function returning another function, usually applied as a function transformation using the `@wrapper` syntax. Common examples for decorators are `classmethod()` and `staticmethod()`.

The decorator syntax is merely syntactic sugar, the following two function definitions are semantically equivalent:

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

(下页继续)

(繼續上一頁)

```
@staticmethod
def f(...):
    ...
```

The same concept exists for classes, but is less commonly used there. See the documentation for function definitions and class definitions for more about decorators.

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.

For more information about descriptors' methods, see descriptors.

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 view The objects returned from `dict.keys()`, `dict.values()`, and `dict.items()` are called dictionary views. They provide a dynamic view on the dictionary' s entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use `list(dictview)`. See dict-views.

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 A programming style which does not look at an object's type to determine if it has the right interface; instead, the method or attribute is simply called or used ("If it looks like a duck and quacks like a duck, it must be a duck.") By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using `type()` or `isinstance()`. (Note, however, that duck-typing can be complemented with *abstract base classes*.) Instead, it typically employs `hasattr()` tests or *EAFP* programming.

EAFP Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many `try` and `except` statements. The technique contrasts with the *LBYL* style common to many other languages such as C.

expression A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also *statement*s which cannot be used as expressions, such as `while`. Assignments are also statements, not expressions.

extension module A module written in C or C++, using Python's C API to interact with the core and with user code.

f-string String literals prefixed with `'f'` or `'F'` are commonly called "f-strings" which is short for formatted string literals. See also **PEP 498**.

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*.

There are actually three categories of file objects: raw *binary files*, buffered *binary files* and *text files*. Their interfaces are defined in the `io` module. The canonical way to create a file object is by using the `open()` function.

file-like object A synonym for *file object*.

finder An object that tries to find the *loader* for a module that is being imported.

Since Python 3.3, there are two types of finder: *meta path finders* for use with `sys.meta_path`, and *path entry finders* for use with `sys.path_hooks`.

See **PEP 302**, **PEP 420** and **PEP 451** for much more detail.

floor division Mathematical division that rounds down to nearest integer. The floor division operator is `//`. For example, the expression `11 // 4` evaluates to `2` in contrast to the `2.75` returned by float true division. Note that `(-11) // 4` is `-3` because that is `-2.75` rounded *downward*. See **PEP 238**.

function A series of statements which returns some value to a caller. It can also be passed zero or more *arguments* which may be used in the execution of the body. See also *parameter*, *method*, and the function section.

function annotation An *annotation* of a function parameter or return value.

Function annotations are usually used for *type hints*: for example, this function is expected to take two `int` arguments and is also expected to have an `int` return value:

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

Function annotation syntax is explained in section function.

See *variable annotation* and **PEP 484**, which describe this functionality.

__future__ A pseudo-module which programmers can use to enable new language features which are not compatible with the current interpreter.

By importing the `__future__` module and evaluating its variables, you can see when a new feature was first added to the language and when it becomes the default:

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

garbage collection The process of freeing memory when it is not used anymore. Python performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles. The garbage collector can be controlled using the `gc` module.

generator A function which returns a *generator iterator*. It looks like a normal function except that it contains `yield` expressions for producing a series of values usable in a for-loop or that can be retrieved one at a time with the `next()` function.

Usually refers to a generator function, but may refer to a *generator iterator* in some contexts. In cases where the intended meaning isn't clear, using the full terms avoids ambiguity.

generator iterator An object created by a *generator* function.

Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the *generator iterator* resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

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 A function composed of multiple functions implementing the same operation for different types. Which implementation should be used during a call is determined by the dispatch algorithm.

See also the *single dispatch* glossary entry, the `functools.singledispatch()` decorator, and **PEP 443**.

GIL See *global interpreter lock*.

global interpreter lock The mechanism used by the *CPython* interpreter to assure that only one thread executes Python *bytecode* at a time. This simplifies the CPython implementation by making the object model (including critical built-in types such as `dict`) implicitly safe against concurrent access. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines.

However, some extension modules, either standard or third-party, are designed so as to release the GIL when doing computationally-intensive tasks such as compression or hashing. Also, the GIL is always released when doing I/O.

Past efforts to create a "free-threaded" interpreter (one which locks shared data at a much finer granularity) have not been successful because performance suffered in the common single-processor case. It is believed that overcoming this performance issue would make the implementation much more complicated and therefore costlier to maintain.

hash-based pyc A bytecode cache file that uses the hash rather than the last-modified time of the corresponding source file to determine its validity. See pyc-invalidation.

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.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

Most of Python's immutable built-in objects are hashable; mutable containers (such as lists or dictionaries) are not; immutable containers (such as tuples and frozensets) are only hashable if their elements are hashable. Objects which are instances of user-defined classes are hashable by default. They all compare unequal (except with themselves), and their hash value is derived from their `id()`.

IDLE An Integrated Development Environment for Python. IDLE is a basic editor and interpreter environment which ships with the standard distribution of Python.

immutable An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.

import path A list of locations (or *path entries*) that are searched by the *path based finder* for modules to import. During import, this list of locations usually comes from `sys.path`, but for subpackages it may also come from the parent package's `__path__` attribute.

importing The process by which Python code in one module is made available to Python code in another module.

importer An object that both finds and loads a module; both a *finder* and *loader* object.

interactive Python has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch `python` with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`).

interpreted Python is an interpreted language, as opposed to a compiled one, though the distinction can be blurry because of the presence of the bytecode compiler. This means that source files can be run directly without explicitly creating an executable which is then run. Interpreted languages typically have a shorter development/debug cycle than compiled ones, though their programs generally also run more slowly. See also *interactive*.

interpreter shutdown When asked to shut down, the Python interpreter enters a special phase where it gradually releases all allocated resources, such as modules and various critical internal structures. It also makes several calls to the *garbage collector*. This can trigger the execution of code in user-defined destructors or weakref callbacks. Code executed during the shutdown phase can encounter various exceptions as the resources it relies on may not function anymore (common examples are library modules or the warnings machinery).

The main reason for interpreter shutdown is that the `__main__` module or the script being run has finished executing.

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.

More information can be found in typeiter.

key function A key function or collation function is a callable that returns a value used for sorting or ordering. For example, locale.strxfrm() is used to produce a sort key that is aware of locale specific sort conventions.

A number of tools in Python accept key functions to control how elements are ordered or grouped. They include min(), max(), sorted(), list.sort(), heapq.merge(), heapq.nsmallest(), heapq.nlargest(), and itertools.groupby().

There are several ways to create a key function. For example. the str.lower() method can serve as a key function for case insensitive sorts. Alternatively, a key function can be built from a lambda expression such as lambda r: (r[0], r[2]). Also, the operator module provides three key function constructors: attrgetter(), itemgetter(), and methodcaller(). See the Sorting HOW TO for examples of how to create and use key functions.

keyword argument See *argument*.

lambda An anonymous inline function consisting of a single *expression* which is evaluated when the function is called. The syntax to create a lambda function is lambda [parameters]: expression

LBYL Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the *EAFP* approach and is characterized by the presence of many if statements.

In a multi-threaded environment, the LBYL approach can risk introducing a race condition between "the looking" and "the leaping". For example, the code, if key in mapping: return mapping[key] can fail if another thread removes *key* from *mapping* after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach.

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 A compact way to process all or part of the elements in a sequence and return a list with the results. result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0] generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The if clause is optional. If omitted, all elements in range(256) are processed.

loader An object that loads a module. It must define a method named load_module(). A loader is typically returned by a *finder*. See **PEP 302** for details and importlib.abc.Loader for an *abstract base class*.

magic method An informal synonym for *special method*.

mapping A container object that supports arbitrary key lookups and implements the methods specified in the Mapping or MutableMapping abstract base classes. Examples include dict, collections.defaultdict, collections.OrderedDict and collections.Counter.

meta path finder A *finder* returned by a search of `sys.meta_path`. Meta path finders are related to, but different from *path entry finders*.

See `importlib.abc.MetaPathFinder` for the methods that meta path finders implement.

metaclass The class of a class. Class definitions create a class name, a class dictionary, and a list of base classes. The metaclass is responsible for taking those three arguments and creating the class. Most object oriented programming languages provide a default implementation. What makes Python special is that it is possible to create custom metaclasses. Most users never need this tool, but when the need arises, metaclasses can provide powerful, elegant solutions. They have been used for logging attribute access, adding thread-safety, tracking object creation, implementing singletons, and many other tasks.

More information can be found in metaclasses.

method A function which is defined inside a class body. If called as an attribute of an instance of that class, the method will get the instance object as its first *argument* (which is usually called `self`). See *function* and *nested scope*.

method resolution order Method Resolution Order is the order in which base classes are searched for a member during lookup. See The Python 2.3 Method Resolution Order for details of the algorithm used by the Python interpreter since the 2.3 release.

module An object that serves as an organizational unit of Python code. Modules have a namespace containing arbitrary Python objects. Modules are loaded into Python by the process of *importing*.

See also *package*.

module spec A namespace containing the import-related information used to load a module. An instance of `importlib.machinery.ModuleSpec`.

MRO See *method resolution order*.

mutable Mutable objects can change their value but keep their `id()`. See also *immutable*.

named tuple The term "named tuple" applies to any type or class that inherits from tuple and whose indexable elements are also accessible using named attributes. The type or class may have other features as well.

Several built-in types are named tuples, including the values returned by `time.localtime()` and `os.stat()`. Another example is `sys.float_info`:

```
>>> 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 with the factory function `collections.namedtuple()`. The latter technique also adds some extra methods that may not be found in hand-written or built-in named tuples.

namespace The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `builtins.open` and `os.open()` are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.islice()` makes it clear that those functions are implemented by the `random` and `itertools` modules, respectively.

namespace package A **PEP 420** *package* which serves only as a container for subpackages. Namespace packages may have no physical representation, and specifically are not like a *regular package* because they have no `__init__.py` file.

See also *module*.

nested scope The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes by default work only for

reference and not for assignment. Local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace. The `nonlocal` allows writing to outer scopes.

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 Any data with state (attributes or value) and defined behavior (methods). Also the ultimate base class of any *new-style class*.

package A Python *module* which can contain submodules or recursively, subpackages. Technically, a package is a Python module with an `__path__` attribute.

See also *regular package* and *namespace package*.

parameter A named entity in a *function* (or method) definition that specifies an *argument* (or in some cases, arguments) that the function can accept. There are five kinds of parameter:

- *positional-or-keyword*: specifies an argument that can be passed either *positionally* or as a *keyword argument*. This is the default kind of parameter, for example *foo* and *bar* in the following:

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

- *positional-only*: specifies an argument that can be supplied only by position. Python has no syntax for defining positional-only parameters. However, some built-in functions have positional-only parameters (e.g. `abs()`).
- *keyword-only*: specifies an argument that can be supplied only by keyword. Keyword-only parameters can be defined by including a single var-positional parameter or bare `*` in the parameter list of the function definition before them, for example *kw_only1* and *kw_only2* in the following:

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

- *var-positional*: specifies that an arbitrary sequence of positional arguments can be provided (in addition to any positional arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `*`, for example *args* in the following:

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

- *var-keyword*: specifies that arbitrarily many keyword arguments can be provided (in addition to any keyword arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `**`, for example *kwargs* in the example above.

Parameters can specify both optional and required arguments, as well as default values for some optional arguments.

See also the *argument* glossary entry, the FAQ question on the difference between arguments and parameters, the `inspect.Parameter` class, the function section, and **PEP 362**.

path entry A single location on the *import path* which the *path based finder* consults to find modules for importing.

path entry finder A *finder* returned by a callable on `sys.path_hooks` (i.e. a *path entry hook*) which knows how to locate modules given a *path entry*.

See `importlib.abc.PathEntryFinder` for the methods that path entry finders implement.

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

path based finder One of the default *meta path finders* which searches an *import path* for modules.

path-like object An object representing a file system path. A path-like object is either a `str` or `bytes` object representing a path, or an object implementing the `os.PathLike` protocol. An object that supports the `os.PathLike` protocol can be converted to a `str` or `bytes` file system path by calling the `os.fspath()` function; `os.fsdecode()` and `os.fsencode()` can be used to guarantee a `str` or `bytes` result instead, respectively. Introduced by **PEP 519**.

PEP Python Enhancement Proposal. A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment. PEPs should provide a concise technical specification and a rationale for proposed features.

PEPs are intended to be the primary mechanisms for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python. The PEP author is responsible for building consensus within the community and documenting dissenting opinions.

See **PEP 1**.

portion A set of files in a single directory (possibly stored in a zip file) that contribute to a namespace package, as defined in **PEP 420**.

positional argument See *argument*.

provisional API A provisional API is one which has been deliberately excluded from the standard library's backwards compatibility guarantees. While major changes to such interfaces are not expected, as long as they are marked provisional, backwards incompatible changes (up to and including removal of the interface) may occur if deemed necessary by core developers. Such changes will not be made gratuitously -- they will occur only if serious fundamental flaws are uncovered that were missed prior to the inclusion of the API.

Even for provisional APIs, backwards incompatible changes are seen as a "solution of last resort" - every attempt will still be made to find a backwards compatible resolution to any identified problems.

This process allows the standard library to continue to evolve over time, without locking in problematic design errors for extended periods of time. See **PEP 411** for more details.

provisional package See *provisional API*.

Python 3000 Nickname for the Python 3.x release line (coined long ago when the release of version 3 was something in the distant future.) This is also abbreviated "Py3k".

Pythonic An idea or piece of code which closely follows the most common idioms of the Python language, rather than implementing code using concepts common to other languages. For example, a common idiom in Python is to loop over all elements of an iterable using a `for` statement. Many other languages don't have this type of construct, so people unfamiliar with Python sometimes use a numerical counter instead:

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

As opposed to the cleaner, Pythonic method:

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

qualified name A dotted name showing the "path" from a module's global scope to a class, function or method defined in that module, as defined in **PEP 3155**. For top-level functions and classes, the qualified name is the same as the object's name:

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

When used to refer to modules, the *fully qualified name* means the entire dotted path to the module, including any parent packages, e.g. `email.mime.text`:

```
>>> 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. Reference counting is generally not visible to Python code, but it is a key element of the *CPython* implementation. The `sys` module defines a `getrefcount()` function that programmers can call to return the reference count for a particular object.

regular package A traditional *package*, such as a directory containing an `__init__.py` file.

See also *namespace package*.

__slots__ A declaration inside a class that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

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()`.

single dispatch A form of *generic function* dispatch where the implementation is chosen based on the type of a single argument.

slice An object usually containing a portion of a *sequence*. A slice is created using the subscript notation, `[]` with colons between numbers when several are given, such as in `variable_name[1:3:5]`. The bracket (subscript) notation uses `slice` objects internally.

special method A method that is called implicitly by Python to execute a certain operation on a type, such as addition. Such methods have names starting and ending with double underscores. Special methods are documented in specialnames.

statement A statement is part of a suite (a "block" of code). A statement is either an *expression* or one of several constructs with a keyword, such as `if`, `while` or `for`.

text encoding A codec which encodes Unicode strings to bytes.

text file A *file object* able to read and write `str` objects. Often, a text file actually accesses a byte-oriented datastream and handles the *text encoding* automatically. Examples of text files are files opened in text mode (`'r'` or `'w'`), `sys.stdin`, `sys.stdout`, and instances of `io.StringIO`.

See also *binary file* for a file object able to read and write *bytes-like objects*.

triple-quoted string A string which is bound by three instances of either a quotation mark (") or an apostrophe ('). While they don't provide any functionality not available with single-quoted strings, they are useful for a number of reasons. They allow you to include unescaped single and double quotes within a string and they can span multiple lines without the use of the continuation character, making them especially useful when writing docstrings.

type The type of a Python object determines what kind of object it is; every object has a type. An object's type is accessible as its `__class__` attribute or can be retrieved with `type(obj)`.

type alias A synonym for a type, created by assigning the type to an identifier.

Type aliases are useful for simplifying *type hints*. For example:

```
from typing import List, Tuple

```

(下页继续)

(繼續上一頁)

```
def remove_gray_shades(
        colors: List[Tuple[int, int, int]]) -> List[Tuple[int, int, int]]:
    pass
```

could be made more readable like this:

```
from typing import List, Tuple

Color = Tuple[int, int, int]

def remove_gray_shades(colors: List[Color]) -> List[Color]:
    pass
```

See `typing` and **PEP 484**, which describe this functionality.

type hint An *annotation* that specifies the expected type for a variable, a class attribute, or a function parameter or return value.

Type hints are optional and are not enforced by Python but they are useful to static type analysis tools, and aid IDEs with code completion and refactoring.

Type hints of global variables, class attributes, and functions, but not local variables, can be accessed using `typing.get_type_hints()`.

See `typing` and **PEP 484**, which describe this functionality.

universal newlines A manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention `'\n'`, the Windows convention `'\r\n'`, and the old Macintosh convention `'\r'`. See **PEP 278** and **PEP 3116**, as well as `bytes.splitlines()` for an additional use.

variable annotation An *annotation* of a variable or a class attribute.

When annotating a variable or a class attribute, assignment is optional:

```
class C:
    field: 'annotation'
```

Variable annotations are usually used for *type hints*: for example this variable is expected to take `int` values:

```
count: int = 0
```

Variable annotation syntax is explained in section annassign.

See *function annotation*, **PEP 484** and **PEP 526**, which describe this functionality.

virtual environment A cooperatively isolated runtime environment that allows Python users and applications to install and upgrade Python distribution packages without interfering with the behaviour of other Python applications running on the same system.

See also `venv`.

virtual machine A computer defined entirely in software. Python's virtual machine executes the *bytecode* emitted by the bytecode compiler.

Zen of Python Listing of Python design principles and philosophies that are helpful in understanding and using the language. The listing can be found by typing "`import this`" at the interactive prompt.

APPENDIX B

關於這些🄵明文件

這些🄵明文件是透過 Sphinx（一個專🄵 Python 🄵明文件所撰寫的文件處理器）將使用 reStructuredText 撰寫的原始檔轉🄵而成。

如同 Python 自身，透過自願者的努力下🄵出文件與封裝後自動化執行工具。若想要回報臭蟲，請見 reporting-bugs 頁面，🄵含相關資訊。我們永遠歡迎新的自願者加入！

致謝：

- Fred L. Drake, Jr.，原始 Python 文件工具集的創造者以及一大部份🄵容的作者。
- 創造 reStructuredText 和 Docutils 工具組的 Docutils 專案；
- Fredrik Lundh 先生，Sphinx 從他的 Alternative Python Reference 計劃中獲得許多的好主意。

B.1 Python 文件的貢獻者們

許多人都曾🄵 Python 這門語言、Python 標准函式庫和 Python 🄵明文件貢獻過。Python 所發🄵的原始碼中含有部份貢獻者的清單，請見 Misc/ACKS 。

正因🄵 Python 社群的撰寫與貢獻才有這份這🄵棒的🄵明文件 -- 感謝所有貢獻過的人們！

APPENDIX C

歷史與授權

C.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see https://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see https://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see https://www.zope.org/). In 2001, the Python Software Foundation (PSF, see https://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see https://opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 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 and above	2.1.1	2001-now	PSF	yes

備🄵: GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

C.2 Terms and conditions for accessing or otherwise using Python

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.7.14

```
1. This LICENSE AGREEMENT is between the Python Software Foundation␣
↪("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise␣
↪using Python
   3.7.14 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 3.7.14 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's␣
↪notice of
   copyright, i.e., "Copyright © 2001-2022 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 3.7.14 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 3.7.14 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
   3.7.14.

4. PSF is making Python 3.7.14 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 3.7.14 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.7.14
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A␣
↪RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.7.14, 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 3.7.14, 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
   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.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` 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
```

(下页继续)

(繼續上一頁)

```
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 Sockets

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, http://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 `asynchat` and `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
```

(下页继续)

(繼續上一頁)

```
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
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

```

(下页继续)

(繼續上一頁)

```
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` module contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       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 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
```

(下页继续)

(繼續上一頁)

```
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-
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_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

The `select` module contains the following notice for the kqueue interface:

```
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
   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.
```

(下页继续)

(繼續上一頁)

```
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 and 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 http://www.netlib.org/fp/. 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.
 *
```

(下页继续)

(繼續上一頁)

```
* 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`, `ssl`, `crypt` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and Mac OS X installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here:

```
LICENSE ISSUES
==============

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.

OpenSSL License
---------------

  /* ====================================================================
   * Copyright (c) 1998-2008 The OpenSSL 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. All advertising materials mentioning features or use of this
   *    software must display the following acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
   *
   * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
   *    endorse or promote products derived from this software without
   *    prior written permission. For written permission, please contact
   *    openssl-core@openssl.org.
   *
   * 5. Products derived from this software may not be called "OpenSSL"
   *    nor may "OpenSSL" appear in their names without prior written
   *    permission of the OpenSSL Project.
   *
   * 6. Redistributions of any form whatsoever must retain the following
   *    acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
   *
```

(下页继续)

(繼續上一頁)

```
    * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
    * EXPRESSED 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 OpenSSL PROJECT OR
    * ITS 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.
    * ====================================================================
    *
    * This product includes cryptographic software written by Eric Young
    * (eay@cryptsoft.com).  This product includes software written by Tim
    * Hudson (tjh@cryptsoft.com).
    *
    */

Original SSLeay License
-----------------------

   /* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
    * All rights reserved.
    *
    * This package is an SSL implementation written
    * by Eric Young (eay@cryptsoft.com).
    * The implementation was written so as to conform with Netscapes SSL.
    *
    * This library is free for commercial and non-commercial use as long as
    * the following conditions are aheared to.  The following conditions
    * apply to all code found in this distribution, be it the RC4, RSA,
    * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
    * included with this distribution is covered by the same copyright terms
    * except that the holder is Tim Hudson (tjh@cryptsoft.com).
    *
    * Copyright remains Eric Young's, and as such any Copyright notices in
    * the code are not to be removed.
    * If this package is used in a product, Eric Young should be given attribution
    * as the author of the parts of the library used.
    * This can be in the form of a textual message at program startup or
    * in documentation (online or textual) provided with the package.
    *
    * 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 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. All advertising materials mentioning features or use of this software
    *    must display the following acknowledgement:
    *    "This product includes cryptographic software written by
    *     Eric Young (eay@cryptsoft.com)"
    *    The word 'cryptographic' can be left out if the rouines from the library
    *    being used are not cryptographic related :-).
    * 4. If you include any Windows specific code (or a derivative thereof) from
    *    the apps directory (application code) you must include an acknowledgement:
    *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
```

(下页继续)

(繼續上一頁)

```
    *
    * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``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.
    *
    * The licence and distribution terms for any publically available version or
    * derivative of this code cannot be changed.  i.e. this code cannot simply be
    * copied and put under another distribution licence
    * [including the GNU Public Licence.]
    */
```

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
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` extension 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,
```

(下页继续)

(繼續上一頁)

```
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.

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` is based on the cfuhash project:

```
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:
```

(下页继续)

(繼續上一頁)

```
  * 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
"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` module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec`:

```
Copyright (c) 2008-2016 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.
```

APPENDIX D

版權宣告

Python 和這些文件是:

完整的授權條款資訊請參見歷史與授權。

索引

Q

R

S

T

U

V

W

Z