Python Tutorial

發🄵 *3.9.6*

Guido van Rossum
and the Python development team

8 月 30, 2021

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。

术语对照表 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准模組，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能在一陳述式 (statement) 中表達很F雜的操作；
- 陳述式的段落以縮排F區隔而非括號；
- 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位形式 (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，F在該應用程式中使用 Python 寫擴充或者作F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F有關F。在F明文件中引用他們的喜劇不但F問題，這甚至是個被鼓勵的行F!

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F讀，一邊將所學用在 Python 直譯器中玩耍。

在下個章節中，將會解F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F，將會透過許多範例介紹 Python 語言與其系統的諸多特色。一開始是簡單的運算式 (expression)、陳述式 (statement) 和資料型F (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 (exception) 與使用者自定義類F (class)。

CHAPTER 2

使用 Python 直譯器

2.1 Ｆ動直譯器

Python 直譯器一般安裝在 /usr/local/bin/python3.9 路徑下；將 /usr/local/bin 加入 Unix shell 的搜索路徑，輸入以下指令就可以Ｆ動 Python：

```
python3.9
```

能Ｆ動 Python[1]。因Ｆ直譯器存放的目Ｆ是個安裝選項，其他的目Ｆ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。）

Windows 系統中，從 Microsoft Store 安裝 Python 後，就可以使用 python3.9 命令了。如果安裝了 py.exe launcher，則可以使用 py 命令。請參Ｆ附Ｆ：setting-envvars，了解其他Ｆ動 Python 的方式。

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

直譯器的指令列編輯功能有很多，在支援 GNU Readline 函式庫的系統上包含：互動編輯、歷史取代、指令補完等功能。最快檢查有無支援指令列編輯的方法Ｆ：在第一個 Python 提示符後輸入 Control-P，如果出現ＦＦ聲，就代表有支援；見附Ｆ互動式輸入編輯和歷史記Ｆ替Ｆ介紹相關的快速鍵。如果什Ｆ事都Ｆ有發生，或者出現一個 ^P，就代表ＦＦ有指令列編輯功能；此時只能使用 backspace 去除該行的字元。

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Ｆ執行指令；如果被呼叫時給定檔名Ｆ引數或者使用 stdin 傳入檔案Ｆ容，它會將這個檔案視ＦＦ本來Ｆ讀。

另一個Ｆ動直譯器的方式Ｆ python -c command [arg] ...，它會執行在 *command* Ｆ的指令（們），行Ｆ如同 shell 的 -c 選項。因Ｆ 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Ｆ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Ｆ。

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

所有指令可用的參數都詳記在 using-on-general。

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

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

在終端 (tty) 輸入□執行指令時，直譯器在*互動模式*（*interactive mode*）中運行。在這種模式中，會顯示*主提示符*，提示輸入下一條指令，主提示符通常用三個大於號（>>>）表示；輸入連續行時，顯示*次要提示符*，預設是三個點（...）。進入直譯器時，首先顯示歡迎訊息、版本訊息、版權聲明，然後才是提示符：

```
$ python3.9
Python 3.9 (default, June 4 2019, 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。

如果不使用預設編碼，則要聲明檔案的編碼，檔案的第一行要寫成特殊□解。語法如下：

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

其中，*encoding* 可以是 Python 支援的任意一種 codecs。

比如，聲明使用 Windows-1252 編碼，源碼檔案要寫成：

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

第一行的規則也有一種例外情□，在源碼以*UNIX "shebang" line* 行開頭時。此時，編碼聲明要寫在檔案的第二行。例如：

```
#!/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🄵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🄵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🄵行；這被使用在多行指令結束輸入時。

在本手🄵中的許多範例中，即便他們🄵互動式地輸入，仍然包含🄵解。Python 中的🄵解 (comments) 由 hash 字元 # 開始一直到該行結束。🄵解可以從該行之首、空白後、或程式碼之後開始，但不會出現在字串文本 (string literal) 之中。hash 字元在字串文本之中時仍視🄵一 hash 字元。因🄵🄵解只是用來🄵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。🄵動直譯器🄵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🄵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
```

(下页继续)

(繼續上一頁)

```
>>> (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  # floored quotient * 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
```

(下页继续)

[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。

(繼續上一頁)

```
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`）。

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🄴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🄴 (\) 跳🄴。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🄵相同。一般來🄵，字串包含單引號而🄵有雙引號時，會使用雙引號包圍字串。函式 `print()` 會🄵生更易讀的輸出，它會去除掉包圍的引號，🄵且直接印出被跳🄴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
```

[2] 不像其他語言，特殊符號如 `\n` 在單 (`'...'`) 和雙 (`"..."`) 引號中有相同的意思。兩種引號的唯一差🄵，在於使用單引號時，不需要跳🄵 `"`（但必須跳🄴 `\'`），反之亦同。

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

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

會🄵生以下的輸出（注意第一個🄵行🄵🄵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)，🄵用 * 重覆該字串的🄵容：

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

兩個以上相鄰的字串文本（*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
```

(下页继续)

(繼續上一頁)

```
'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'
```

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'
```

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'
```

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 method（方法）。

f-strings 包含有運算式的字串文本。

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

old-string-formatting 在字串🄵 `%` 運算子的左運算元時，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]
```

如同字串（以及其他🄵建的*sequence* 型🄵），list 可以被索引和切片 (slice)：

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

所有切片操作都會回傳一個新的 list ，包含要求的元素。這意謂著以下這個切片回傳了原本 list 的 淺🄵🄵：

```
>>> 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
```

(下页继续)

(繼續上一頁)

```
['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` 🄵圈只要它的條件🄵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🄴真 (true)；零🄵假 (false)。條件可以是字串、list、甚至是任何序列型🄴；任何非零長度的序列🄵真，空的序列即🄵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）。

- 🄵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🄵圈🄵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🄵雜的輸入；多數編輯器都有自動縮排的功能。當一個🄵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🄵語法解析器無法判斷你何時輸入🄵合陳述的最後一行）。注意在一個縮排段落🄵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🄵數個引數、浮點數數值和字串。印出的字串將不帶有引號，🄵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🄵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,
```

🄵解

CHAPTER 4

深入了解流程控制

除了剛才介紹的 while，Python 擁有在其他程式語言中常用的流程控制語法，🄵有一些不一樣的改變。

4.1 if 陳述式

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
```

在陳述式中，可以🄵有或有許多個 elif 🄵述，且 else 🄵述🄵不是必要的。關鍵字 elif 是「else if」的縮寫，用來避免過多的縮排。一個 if ... elif ... elif ... 序列可以用來替代其他程式語言中的 switch 或 case 陳述式。

4.2 for 陳述式

在 Python 中的 for 陳述式有點不同於在 C 或 Pascal 中的慣用方式。相較於只能🄴代 (iterate) 一個等差數列（如 Pascal），或給予使用者定義🄴代步驟與終止條件（如 C），Python 的 for 陳述式🄴代任何序列（list 或者字串）的元素，順序與它們出現在序列中的順序相同。例如（無意雙關）：

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

在🄴代一個集合的同時修改該集合的🄴容，很難獲取想要的結果。比較直觀的替代方式，是🄴代該集合的副本，或建立一個新的集合：

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

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

4.3 range() 函式

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

```
>>> 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🄴每一個項目的索引值。也可以讓 range 從其他數值開始計數，或者給定不同的公差（甚至🄵負；有時稱之🄵 step）：

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

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

>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

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：

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

在很多情🄵下，由 range() 回傳的物件表現得像是一個 list（串列）一樣，但實際上它🄵不是。它是一個在🄵代時能🄵回傳所要求的序列中所有項目的物件，但它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🄵代物件），意即能作🄵函式及架構中可以一直獲取項目直到取盡的對象。我們已經了解 for 陳述式就是如此的架構，另一個使用 iterable 的函式範例是 sum()：

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

下文將介紹更多返回可迭代对象或把可迭代对象当作参数的函数。在資料結構 这一章节中，我们将讨论有关 list() 的更多细节。

4.4 🄵圈🄵的 break 和 continue 陳述式及 else 子句

break 陳述式，如同 C 語言，終止包含它的最🄵部 for 或 while 🄵圈。

🄵圈陳述式可以帶有一個 else 子句。當🄵圈用盡所有的 iterable （在 for 中）或條件🄵假（在 while 中）時，這個子句會被執行；但🄵圈被 break 陳述式終止時則不會執行。底下尋找質數的🄵圈即示範了這個行🄵：

```
>>> 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 子句屬於 for 🄵圈，🄵非 if 陳述式。)

當 else 子句用於🄵圈時，相較於搭配 if 陳述式使用，它的行🄵與 try 陳述式中的 else 子句更🄵相似：try 陳述式的 else 子句在🄵有發生例外 (exception) 時執行，而🄵圈的 else 子句在🄵有任何 break 發生時執行。更多有關 try 陳述式和例外的介紹，見處理例外。

continue 陳述式，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 an odd number", num)
...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

4.5 pass 陳述式

pass 陳述式不執行任何動作。它可用在語法上需要一個陳述式但程式不需要執行任何動作的時候。例如：

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

這經常用於建立簡單的 class（類🄵）：

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

pass 亦可作🄵一個函式或條件判斷主體的預留位置，在你撰寫新的程式碼時讓你保持在更抽象的思維層次。pass 會直接被忽略：

```
>>> 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=' ')
```

(下页继续)

(繼續上一頁)

```
...         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 介紹一個函式的定義。它之後必須連著該函式的名稱和置於括號之中的一串參數。自下一行起，所有縮排的陳述式成F該函式的主體。

一個函式的第一個陳述式可以是一個字串文本；該字串文本被視F該函式的F明文件字串，即 *docstring*。（關於 docstring 的細節請參見F明文件字串 *(Documentation Strings)*段落。）有些工具可以使用 docstring 來自動F生F上或可列印的文件，或讓使用者能以互動的方式在原始碼中F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。

函式執行時會建立一個新的符號表 (symbol table) 來儲存該函式F的區域變數 (local variable)。更精確地F，所有在函式F的變數賦值都會把該值儲存在一個區域符號表。然而，在引用一個變數時，會先從區域符號表開始搜尋，其次F外層函式的區域符號表，其次F全域符號表 (global symbol table)，最後F所有F建的名稱。因此，在函式中，全域變數及外層函式變數雖然可以被引用，但無法被直接賦值（除非全域變數是在 global 陳述式中被定義，或外層函式變數在 nonlocal 陳述式中被定義）。

在一個函式被呼叫的時候，實際傳入的參數（引數）會被加入至該函式的區域符號表。因此，引數傳入的方式F傳值呼叫 *(call by value)*（這F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F的函式或遞F呼叫它自己時，在被呼叫的函式中會建立一個新的區域符號表。

函式定義時，會把該函式名稱加入至當前的符號表。函式名稱的值帶有一個型F，F被直譯器辨識F使用者自定函式 (user-defined function)。該值可以被指定給F的變數名，使該變數名也可以被當作函式使用。這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F的語言，你可能不同意 fib 是個函式，而是個程序 (procedure)，因F它FF有回傳值。實際上，即使一個函式缺少一個 return 陳述式，它亦有一個固定的回傳值。這個值稱F None（它是一個F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
...
```

(下页继续)

[1] 實際上，傳址呼叫 *(call by object reference)* 的F法可能較F貼切。因F，若傳遞的是一個可變物件時，呼叫者將看得見被呼叫者對物件做出的任何改變（例如被插入 list F的新項目）。

(繼續上一頁)

```
>>> 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：

- `return` 陳述式會讓一個函式回傳一個值。單獨使用 `return` 不外加一個運算式作🄵引數時會回傳 `None`。一個函式執行到結束也會回傳 `None`。
- `result.append(a)` 陳述式呼叫了一個 list 物件 `result` 的 *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🄵一運算式），而 `methodname` 🄵該 method 的名稱，🄵由該物件的型🄵所定義。不同的型🄵定義不同的 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*（類🄵）定義自己的物件型🄵和 method，見类）範例中的 `append()` method 定義在 list 物件中；它會在該 list 的末端加入一個新的元素。這個例子等同於 `result = result + [a]`，但更有效率。

4.7 深入了解函式定義

定義函式時使用的引數 (argument) 數量是可變的。總共有三種可以組合使用的形式。

4.7.1 預設引數值

🄵一個或多個引數指定預設值是很有用的方式。函式建立後，可以用比定義時更少的引數呼叫該函式。例如：

```
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)
```

該函式可以用以下幾種方式被呼叫：

- 只給必要引數：`ask_ok('Do you really want to quit?')`
- 給予一個選擇性引數：`ask_ok('OK to overwrite the file?', 2)`
- 給予所有引數：`ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

此例也使用了關鍵字 `in`，用於測試序列中是否包含某個特定值。

預設值是在函式定義當下，於定義時的作用域中求值，所以：

```
i = 5

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

(下页继续)

(繼續上一頁)

```
i = 6
f()
```

將會輸出 5。

重要警告：預設值只求值一次。當預設值🄴可變物件，例如 list、dictionary（字典）或許多類🄴實例時，會🄵生不同的結果。例如，以下函式於後續呼叫時會累積曾經傳遞的引數：

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

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

將會輸出：

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

如果不想在後續呼叫之間共用預設值，應以如下方式編寫函式：

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

4.7.2 關鍵字引數

函式也可以使用關鍵字引數，以 kwarg=value 的形式呼叫。舉例來🄵，以下函式：

```
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, "!")
```

接受一個必要引數 (voltage) 和三個選擇性引數 (state，action，和 type)。該函式可用下列任一方式呼叫：

```
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
```

但以下呼叫方式都無效：

```
parrot()                     # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument after a keyword argument
```

(下页继续)

(繼續上一頁)

```
parrot(110, voltage=220)     # duplicate value for the same argument
parrot(actor='John Cleese')  # unknown keyword argument
```

函式呼叫時，關鍵字引數 (keyword argument) 必須在位置引數 (positional argument) 後面。所有傳遞的關鍵字引數都必須匹配一個可被函式接受的引數（`actor` 不是 `parrot` 函式的有效引數），而關鍵字引數的順序🄵不重要。此規則也包括必要引數，(`parrot(voltage=1000)` 也有效)。一個引數不可多次被賦值，下面就是一個因此限制而無效的例子：

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

當最後一個參數🄵 `**name` 形式時，它接收一個 dictionary（字典，詳見 typesmapping），該字典包含所有可對應形式參數以外的關鍵字引數。`**name` 可以與 `*name` 參數（下一小節介紹）組合使用，`*name` 接收一個*tuple*，該 tuple 包含一般參數以外的位置引數（`*name` 必須出現在 `**name` 前面）。例如，若我們定義這樣的函式：

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

它可以被如此呼叫：

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

輸出結果如下：

```
-- 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
```

注意，關鍵字引數的輸出順序與呼叫函式時被提供的順序必定一致。

4.7.3 特殊參數

在預設情🄵，引數能以位置或明確地以關鍵字傳遞給 Python 函式。🄵了程式的可讀性及效能，限制引數的傳遞方式是合理的，如此，開發者只需查看函式定義，即可確定各項目是按位置，按位置或關鍵字，還是按關鍵字傳遞。

函式定義可能如以下樣式：

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

/ 和 * 是選擇性的。這些符號若被使用，是表明引數被傳遞給函式的參數種類：僅限位置、位置或關鍵字、僅限關鍵字。關鍵字參數也稱🄵附名參數 (named parameters)。

位置或關鍵字引數 (Positional-or-Keyword Arguments)

若函式定義中未使用 / 和 * 時，引數可以按位置或關鍵字傳遞給函式。

僅限位置參數 (Positional-Only Parameters)

此處再詳述一些細節，特定參數可以標記🄵僅限位置。若參數🄵僅限位置時，它們的順序很重要，且這些參數不能用關鍵字傳遞。僅限位置參數必須放在 /（斜🄵）之前。/ 用於在邏輯上分開僅限位置參數與其余參數。如果函式定義中🄵有 /，則表示🄵有任何僅限位置參數。

/ 後面的參數可以是位置或關鍵字或僅限關鍵字參數。

僅限關鍵字引數 (Keyword-Only Arguments)

要把參數標記🄵僅限關鍵字，表明參數必須以關鍵字引數傳遞，必須在引數列表中第一個僅限關鍵字參數前放上 *。

函式范例

請看以下的函式定義範例，注意 / 和 * 記號：

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

第一個函式定義 standard_arg 是我們最熟悉的形式，對呼叫方式🄵有任何限制，可以按位置或關鍵字傳遞引數：

```
>>> standard_arg(2)
2

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

第二個函式 pos_only_arg 的函式定義中有 /，因此僅限使用位置參數：

```
>>> pos_only_arg(1)
1

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

第三個函式 kwd_only_args 的函式定義透過 * 表明僅限關鍵字引數：

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

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

最後一個函式在同一個函式定義中，使用了全部三種呼叫方式：

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

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

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

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

最後，請看這個函式定義，如果 **kwds 🄵有 name 這個鍵，可能與位置引數 name 🄵生🄵在衝突：

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

呼叫該函式不可能回傳 True，因🄵關鍵字 'name' 永遠是連結在第一個參數。例如：

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

使用 / （僅限位置引數）後，就可以了。函式定義會允許 name 當作位置引數，而 'name' 也可以當作關鍵字引數中的鍵：

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

🄵句話🄵，僅限位置參數的名稱可以在 **kwds 中使用，而不🄴生歧義。

回顧

此用例🄴定哪些參數可以用於函式定義：

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

🄵明：

- 如果不想讓使用者使用參數名稱，請使用僅限位置。當參數名稱🄴有實際意義時，若你想控制引數在函式呼叫的排列順序，或同時使用位置參數和任意關鍵字時，這種方式很有用。
- 當參數名稱有意義，且明確的名稱可讓函式定義更易理解，或是你不希望使用者依賴引數被傳遞時的位置時，請使用僅限關鍵字。
- 對於應用程式介面 (API)，使用僅限位置，以防止未來參數名稱被修改時造成 API 的中斷性變更。

4.7.4 任意引數列表 (Arbitrary Argument Lists)

最後，有個較不常用的選項，是規定函式被呼叫時，可以使用任意數量的引數。這些引數會被包裝進一個 tuple 中（詳見*Tuples 和序列 (Sequences)*）。在可變數量的引數之前，可能有零個或多個普通引數：

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

通常，這些 variadic（可變的）引數會出現在參數列表的最末端，這樣它們就可以把所有傳遞給函式的剩余輸入引數都撈起來。出現在 *args 參數後面的任何參數必須是「僅限關鍵字」引數，意即它們只能作🄵關鍵字引數，而不能用作位置引數。

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

4.7.5 拆解引數列表（Unpacking Argument Lists）

當引數們已經存在一個 list 或 tuple 🄵，但🄵了滿足一個需要個🄵位置引數的函式呼叫，而去拆解它們時，情🄵就剛好相反。例如，🄵建的 `range()` 函式要求分開的 *start* 和 *stop* 引數。如果這些引數不是分開的，則要在呼叫函式時，用 `*` 運算子把引數們從 list 或 tuple 中拆解出來：

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

同樣地，dictionary（字典）可以用 `**` 運算子傳遞關鍵字引數：

```
>>> 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 !
```

4.7.6 Lambda 運算式

`lambda` 關鍵字用於建立小巧的匿名函式。`lambda a, b: a+b` 函式返回兩個引數的和。Lambda 函式可用於任何需要函數物件的地方。在語法上，它們被限定只能是單一運算式。在語義上，它就是一個普通函式定義的語法糖 (syntactic sugar)。與巢狀函式定義一樣，lambda 函式可以從包含它的作用域中引用變數：

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

上面的例子用 lambda 運算式回傳了一個函式。另外的用法是傳遞一個小函式當作引數：

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

4.7.7 🄵明文件字串 (Documentation Strings)

以下是關於🄵明文件字串🄵容和格式的慣例。

第一行都是一段關於此物件目的之簡短摘要。🄵保持簡潔，不應在這🄵明確地陳述物件的名稱或型🄵，因🄵有其他方法可以達到相同目的（除非該名稱剛好是一個描述函式運算的動詞）。這一行應以大寫字母開頭，以句號結尾。

文件字串🄵多行時，第二行應🄵空白行，在視覺上將摘要與其余描述分開。後面幾行可包含一或多個段落，描述此物件的呼叫慣例、副作用等。

Python 剖析器 (parser) 不會去除 Python 中多行字串的縮排，因此，處理🄵明文件的工具應在必要時去除縮排。這項操作遵循以下慣例：在字串第一行之後的第一個非空白行🄵定了整個🄵明文件字串的縮排量（不能用第一行的縮排，因🄵它通常與字串的開頭引號們相鄰，其縮排在字串文本中🄵不明顯），然後，所有字串行開頭處與此縮排量「等價」的空白字元會被去除。不應出現比上述縮進量更少的字串行，但若真的出現了，這些行的全部前導空白字元都應被去除。展開 tab 鍵後（通常🄵八個空格），應測試空白字元量是否等價。

下面是多行🄵明字串的一個範例：

```
>>> 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.
```

4.7.8 函式🄵釋 (Function Annotations)

函式🄵釋是選擇性的元資料（metadata）資訊，描述使用者定義函式所使用的型🄵（更多資訊詳見 **PEP 3107** 和 **PEP 484**）。

🄵釋以 dictionary（字典）的形式存放在函式的 `__annotations__` 屬性中，且不會影響函式的任何其他部分。參數🄵釋的定義方式是在參數名稱後加一個冒號，冒號後面跟著一個對🄵釋求值的運算式。回傳🄵釋的定義方式是在參數列表和 `def` 陳述式結尾的冒號中間，用一個 `->` 文字接著一個運算式。以下範例🄵釋了一個必要引數、一個選擇性引數，以及回傳值：

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

4.8 間奏曲：程式碼風格 (Coding Style)

現在你即將要寫更長、更🄵雜的 Python 程式，是時候討論一下*編碼樣式*了。大多數語言都能以不同的樣式被書寫（或更精確地🄵，*被格式化*），而有些樣式比其他的更具可讀性。能讓其他人輕松🄴讀你的程式碼永遠是一個好主意，而使用優良的編碼樣式對此有極大的幫助。

對於 Python，大多數的專案都遵循 **PEP 8** 的樣式指南；它推行的編碼樣式相當可讀且賞心🄴目。每個 Python 開發者都應該花點時間研讀；這🄵是該指南的核心重點：

- 用 4 個空格縮排，不要用 tab 鍵。

  4 個空格是小縮排（容許更大的巢套深度）和大縮排（較易🄴讀）之間的折衷方案。Tab 鍵會造成混亂，最好🄴用。

- 🄵行，使一行不超過 79 個字元。

  🄵行能讓使用小顯示器的使用者方便🄴讀，也可以在較大顯示器上🄵排陳列多個程式碼檔案。

- 用空行分隔函式和 class（類🄵），及函式🄵較大塊的程式碼。

- 如果可以，把🄵解放在單獨一行。

- 使用🄵明字串。

- 運算子前後、逗號後要加空格，但不要直接放在括號🄵側：`a = f(1, 2) + g(3, 4)`。

- Class 和函式的命名樣式要一致；按慣例，命名 class 用 `UpperCamelCase`（駝峰式大小寫），命名函式與 method 用 `lowercase_with_underscores`（小寫加底🄵）。永遠用 `self` 作🄵 method 第一個引數的名稱（關於 class 和 method，詳見*初探类*）。

- 若程式碼是🄵了用於國際環境時，不要用花俏的編碼。Python 預設的 UTF-8 或甚至普通的 ASCII，就可以勝任各種情🄵。

- 同樣地，若不同語言使用者🄵讀或維護程式碼的可能性微乎其微，就不要在命名時使用非 ASCII 字元。

🄵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 物件的 method：

`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F代物件）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)` 會插入在 list 首位，而 `a.insert(len(a), x)` 則相當於 `a.append(x)`。

`list.`**`remove`**`(`*x*`)`
F除 list 中第一個值等於 *x* 的元素。若 list 中無此元素則會觸發 `ValueError`。

`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 *x* 在 list 中所出現的次數。

`list.sort`(*, *key=None*, *reverse=False*)
將 list 中的項目排序。（可使用引數來進行客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FF (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`，FF有印出回傳值，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。

另外你可能也會發現，不是所有資料都可以被排序或比較。例如，`[None, 'hello', 10]` 就不可排序，因F整數不能與字串比較，而 *None* 不能與其他型F比較。有些型F根本就F有被定義彼此之間的大小順序，例如，`3+4j < 5+7j` 就是一個無效的比較。

5.1.1 將 List 作F Stack（堆F）使用

List 的操作方法使得它非常簡單可以用來實作 stack（堆F）。Stack F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F的頂層。而使用方法 `pop()` 且不給定索引值去取得堆F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
```

(下页继续)

[1] 其他語言可以回傳變更後的物件，這就允許 method 的串連，例如 `d->insert("a")->remove("b")->sort();`。

(繼續上一頁)

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

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序列或 iterable（可🄵代物件），將每一個元素經過某個運算的結果串接起來成🄵新的 list，或是創建一個子序列，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（或覆寫）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。

一個 list comprehension 的組成，是在一對方括號F，放入一個 expression（運算式）、一個 `for` 子句、再接著零個或多個 `for` 或 `if` 子句。結果會是一個新的 list，F容是在後面的 `for` 和 `if` 子句情境下，對前面運算式求值的結果。例如，這個 list comprehension 組合了兩個 list 中彼此相F的元素：

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

而它就等於：

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

注意 `for` 和 `if` 在這兩段程式F的順序是相同的。

如果 expression 是一個 tuple（例如上面例子中的 `(x, y)`），它必須加上括號：

```
>>> 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F雜的 expression 和巢狀的函式：

```
>>> 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

在 list comprehesion 中開頭的 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 會依據後面的 `for` 環境被求值，所以這個例子就等於：

```
>>> 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]]
```

而它也和這一段相同：

```
>>> 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*拆解引數列表（Unpacking Argument Lists）*。

5.2 del 陳述式

有一個方法可以藉由索引而不是值來🄵除 list 中的項目：del 陳述式。這和 pop() method 傳回一個值不同，del 陳述式可以用來🄵除 list 中的片段或者清空整個 list（我們之前藉由指派一個空的 list 給想🄵除的片段來完成這件事）。例如：

```
>>> 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🄵除整個變數：

```
>>> 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 list 和字串 (string) 有許多共同的特性，像是索引操作 (indexing) 以及切片操作 (slicing) 。他們是序列資料類型中的兩個例子（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類型加入。接著要介紹是下一個標准序列資料類型：*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 是一個較大的運算式的一部分）。指派東西給 tuple 中的個🄵項目是不行的，但是可以建立含有可變物件（譬如 list）的 tuple。

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🄵質的元素序列，🄵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🄵代整個 list 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（用括號把一個單一的值包住是不🄵的）。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 *set*（集合）的資料類型。一個 set 是一組無序且🄵有重🄵的元素。基本的使用方式包括了成員測試和消除重🄵元素。Set 物件也支援聯集、交集、差集和互斥等數學運算。

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，你必須使用 `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🄵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
```

(下页继续)

(繼續上一頁)

```
{'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 🄵建資料類型🄵 *dictionary*（請參考 typesmapping）。Dictionary 有時被稱🄵「關聯記憶體」(associative memories) 或「關聯陣列」(associative arrays)。不像序列是由一個範圍🄵的數字當作索引，dictionary 是由鍵 (key) 來當索引，鍵可以是任何不可變的類型；字串和數字都可以當作鍵。Tuple 也可以當作鍵，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，它就不能當作鍵。你無法使用 list 當作鍵，因🄵 list 可以經由索引指派 (index assignment)、切片指派 (slice assignment) 或是像 `append()` 和 `extend()` 等 method 被修改。

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

Dictionary 主要的操作🄵藉由鍵來儲存一個值🄵且可藉由該鍵來取出該值。也可以使用 `del` 來🄵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 dictionary 使用 `list(d)` 會得到一個包含該字典所有鍵的 list，其排列順序🄵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 dictionary 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 dictionary：

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

此外，dict comprehensions 也可以透過任意鍵與值的運算式來創建 dictionary：

```
>>> {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 dictionary 作🄵圈時，鍵以及其對應的值可以藉由使用 `items()` method 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):
...     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，再對其使用 `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 i in sorted(basket):
...     print(i)
...
apple
apple
banana
orange
orange
pear
```

對序列使用 `set()` 可去除重🄵元素。對序列使用 `sorted()` 加上 `set()`，則是對經過排序後的非重🄵元素跑🄴圈的慣用方法：

```
>>> 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 深入了解條件 (Condition)

使用在 `while` 和 `if` 陳述式的條件句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。所有比較運算子的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（或任何其他布林運算式）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值而非布林值時，短路運算子的回傳值🄴最後被求值的引數。

將一個比較運算或其他布林運算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
```

(下页继续)

(繼續上一頁)

```
>>> non_null
'Trondheim'
```

注意，Python 與 C 語言不一樣，在運算式🄵進行指派必須外顯地使用海象運算子 :=。這樣做避免了在 C 語言🄵常見的一種問題：想要打 == 🄵在運算式🄵輸入 =。

5.8 序列和其他資料類型之比較

序列物件通常可以拿來和其他相同類型的物件做比較。這種比較使用*詞典式 (lexicographical)* 順序：首先比較各自最前面的那項，若不相同，便可🄵定結果；若相同，則比較下一項，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類型，則詞典式比較會遞🄵地執行。如果兩個序列所有的項目都相等，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類型的比較：

```
(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 > 來比較不同類型的物件是合法的，表示物件擁有適當的比較方法。例如，混合的數值類型是根據它們數值來做比較，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🄵。

🄵了支援這一點，Python 有一種方法可以將定義放入檔案中，🄵在互動模式下的直譯器中使用它們。這種檔案稱🄵模組 *(module)*；模組中的定義可以被 *import* 到其他模組中，或是被 *import* 至主 *(main)* 模組（在最頂層執行的🄵本，以及互動模式下，所使用的變數集合）。

模組是指包含 Python 定義和語句的檔案，檔案名稱是模組名稱加上 `.py`。在模組中，模組的名稱（作🄵字串）會是全域變數 `__name__` 的值。例如，用您喜歡的文字編輯器在資料夾中創一個名🄵 `fibo.py` 的檔案，🄵容如下：

```
# 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
```

現在進入 Python 直譯器🄵用以下指令 import 這個模組：

```
>>> import fibo
```

這🄵不會將 fibo 中定義的函式名稱直接輸入當前的符號表中；它只會輸入 fibo 的模組名稱。使用此模組名稱，就可以存取函式：

```
>>> 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'
```

如果您打算經常使用其中某個函式，可以將其指定至區域變數：

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

6.1 深入了解模組

模組可以包含可執行的陳述式以及函式的定義。這些陳述式是作🄵模組的初始化，它們只會在第一次被 import 時才會執行。[1]（如果檔案被當成🄵本執行，也會執行它們）。

每個模組都有它自己的私有符號表，模組🄵定義的函式會把該模組的私有符號表當成全域符號表使用。因此，模組的作者可以在模組中使用全域變數，而不必擔心和使用者的全域變數發生意外的名稱衝突。另一方面，如果你知道自己在做什🄵，你可以用這個方式取用模組的全域變數，以和引用函式一樣的寫法，modname.itemname。

在一個模組中可以 import 其他模組。把所有的 import 陳述式放在模組（就這邊來🄵，🄵本也是一樣）的最開頭是個慣例，但🄵🄵有🄵制。被 import 的模組名稱放置在原模組的全域符號表中。

import 陳述式有另一種變形寫法，可以直接將名稱從欲 import 的模組，直接 import 至原模組的符號表中。例如：

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

在 import 之後的名稱會被導入，但定義該函式的整個模組名稱🄵不會被引入在區域符號表中（因此，示例中的 fibo 未被定義）。

甚至還有另一種變形寫法，可以 import 模組定義的所有名稱：

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

這個寫法會 import 模組中所有的名稱，除了使用底🄵（_）開頭的名稱。大多數情🄵下，Python 程式設計師不大使用這個功能，因🄵它在直譯器中引入了一組未知的名稱，🄵且可能覆蓋了某些您已經定義的🄵容。

請注意，一般情🄵下🄵不建議從模組或套件中 import * 的做法，因🄵它通常會導致可讀性較差的程式碼。但若是使用它來在互動模式中節省打字時間，則是可以接受的。

如果模組名稱後面出現 as，則 as 之後的名稱將直接和被 import 模組綁定在一起。

[1] 實際上，函式定義也是「被執行」的「陳述式」；在執行模組階層的函式定義時，會輸入函式名稱到模組的全域符號表。

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

這個 import 方式和 `import fibo` 實質上是一樣的，唯一的差□是現在要用 `fib` 使用模組。

在使用 `from` 時也可以用同樣的方式獲得類似的效果：

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

備□： 出於效率原因，每個模組在每個直譯器 session 中僅會被 import 一次。因此，如果您更改了模組，則必須重□直譯器——或者，如果只是一個想要在互動模式下測試的模組，可以使用 `importlib.reload()`。例如：`import importlib; importlib.reload(modulename)`。

6.1.1 把模組當作□本執行

當使用以下□容運行 Python 模組時：

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

如同使用 import 指令，模組中的程式碼會被執行，但 `__name__` 被設□ `"__main__"`。這意味著，透過在模組的末尾添加以下程式碼：

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

你可以將檔案作□□本也同時可以作□被 import 的模組，因□解析 (parse) 命令列的程式碼只會在當模組是「主」檔案時，才會執行：

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

如果此模組是被 import 的，則該段程式碼不會被執行：

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

這通常是用來□模組提供方便的使用者介面，或者用於測試目的（執行測試套件時，以□本的方式執行模組）。

6.1.2 模組的搜尋路徑

Import 一個名🄵 `spam` 的模組時，直譯器首先會搜尋具有該名稱的🄵建模組。如果找不到，接下來會在變數 `sys.path` 所給定的資料夾清單之中，搜尋一個名🄵 `spam.py` 的檔案。`sys.path` 從這些位置開始進行初始化：

- 輸入🄵本所位在的資料夾（如未指定檔案時，則是當前資料夾）。
- `PYTHONPATH`（一連串和 shell 變數 `PATH` 的語法相同的資料夾名稱）。
- 安裝相關的預設值。

備🄵： 在支援符號連結 (symlink) 的檔案系統中，輸入🄵本的所在資料夾是在跟隨符號連結之後才被計算的。🄵言之，包含符號連結的資料夾🄵🄵**有**增加到模組的搜尋路徑中。

初始化之後，Python 程式可以修改 `sys.path`。執行中🄵本的所在資料夾會在搜尋路徑的開頭，在標准函式庫路徑之前。這代表該資料夾中的🄵本會優先被載入，而不是函式庫資料夾中相同名稱的模組。除非是有意要做這樣的替🄵，否則這是一個錯誤。請參見標準模組以🄵解更多資訊。

6.1.3 「編譯」Python 檔案

🄵了加快載入模組的速度，Python 將每個模組的編譯版本暫存在 `__pycache__` 資料夾下，🄵命名🄵 `module.`*`version`*`.pyc`，這🄵的 version 是編譯後的檔案的格式名稱，且名稱通常會包含 Python 的版本編號。例如，在 CPython 3.3 中，spam.py 的編譯版本將被暫存🄵 `__pycache__/spam.cpython-33.pyc`。此命名准則可以讓來自不同版本的編譯模組和 Python 的不同版本同時共存。

Python 根據原始碼最後修改的日期，檢查編譯版本是否過期而需要重新編譯。這是一個完全自動的過程。另外，編譯後的模組獨立於平台，因此不同架構的作業系統之間可以共用同一函式庫。

Python 在兩種情🄵下不檢查快取 (cache)。首先，它總是重新編譯且不儲存直接從命令列載入的模組的結果。第二，如果🄵有源模組，則不會檢查快取。要支援非源模組（僅編譯）的發布，編譯後的模組必須位於原始資料夾中，🄵且不能有源模組。

一些給專家的秘訣：

- 可以在 Python 指令上使用開關參數 (switch) `-O` 或 `-OO` 來🄵小已編譯模組的大小。指令參數 `-O` 🄵除 assert（斷言）陳述式，`-OO` 同時🄵除 assert 陳述式和 `__doc__` 字串。由於有些程式可能依賴於上述這些🄵容，因此只有在您知道自己在做什🄵時，才應使用此參數。「已優化」模組有 `opt-` 標記，且通常較小。未來的版本可能會改變優化的效果。
- 讀取 `.pyc` 檔案時，程式的執行速度🄵不會比讀取 `.py` 檔案快。唯一比較快的地方是載入的速度。
- 模組 `compileall` 可以🄵資料夾中的所有模組創建.pyc 檔。
- 更多的細節，包括🄵策流程圖，請參考**PEP 3147**。

6.2 標準模組

Python 附帶了一個標准模組庫，詳細的介紹在另一份文件，稱🄵「Python 函式庫參考手🄵」（簡稱🄵「函式庫參考手🄵」）。有些模組是直譯器中🄵建的；它們使一些不屬於語言核心但依然🄵建的運算得以存取，其目的是🄵了提高效率，或提供作業系統基本操作（例如系統呼叫）。這些模組的集合是一個組態選項，它們取🄵於底層平台。例如：`winreg` 模組僅供 Windows 使用。值得注意的模組是 `sys`，它被🄵建在每個 Python 直譯器中。變數 `sys.ps1` 和 `sys.ps2` 則用來定義主、次提示字元的字串：

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

只有直譯器在互動模式時，才需要定義這兩個變數。

變數 `sys.path` 是一個字串 list，它🄵定直譯器的模組搜尋路徑。它的初始值🄵環境變數 `PYTHONPATH` 中提取的預設路徑，或是當 `PYTHONPATH` 未設定時，從🄵建預設值提取。你可以用標准的 list 操作修改該變數：

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

6.3 dir() 函式

🄵建函式 `dir()` 用於找出模組定義的所有名稱。它回傳一個排序後的字串 list：

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__',
 '__interactivehook__', '__loader__', '__name__', '__package__', '__spec__',
 '__stderr__', '__stdin__', '__stdout__', '__unraisablehook__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_framework',
 '_getframe', '_git', '_home', '_xoptions', 'abiflags', 'addaudithook',
 'api_version', 'argv', 'audit', 'base_exec_prefix', 'base_prefix',
 'breakpointhook', 'builtin_module_names', 'byteorder', 'call_tracing',
 'callstats', 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_info',
 'excepthook', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info',
 'float_repr_style', 'get_asyncgen_hooks', 'get_coroutine_origin_tracking_depth',
 'getallocatedblocks', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencodeerrors', 'getfilesystemencoding', 'getprofile',
 'getrecursionlimit', 'getrefcount', 'getsizeof', 'getswitchinterval',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'is_finalizing', 'last_traceback', 'last_type', 'last_value',
 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks',
 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2', 'pycache_prefix',
 'set_asyncgen_hooks', 'set_coroutine_origin_tracking_depth', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace', 'stderr',
```

(下页继续)

(繼續上一頁)

```
 'stdin', 'stdout', 'thread_info', 'unraisablehook', 'version', 'version_info',
 'warnoptions']
```

🄵有給引數時，dir() 列出目前已定義的名稱：

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

請注意，它列出所有類型的名稱：變數、模組、函式等。

dir() 不會列出🄵建函式和變數的名稱。如果你想要列出它們，它們被定義在標準模組 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',
 '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 套件 (Package)

套件是一種使用「點分隔模組名稱」組織 Python 模組命名空間的方法。例如，模組名稱 A.B 表示套件 A 中名□ B 的子模組。正如模組使用時，不同模組的作者不需擔心與其他模組的全域變數名稱重□，點分隔模組名稱的使用，也讓多模組套件（像 NumPy 或 Pillow）的作者們不須擔心其他套件的模組名稱。

假設你想設計一個能統一處理音訊檔案及音訊數據的模組集（「套件」）。因□音訊檔案有很多的不同的格式（通常以它們的副檔名來辨識，例如：.wav，.aiff，.au），因此，□了不同檔案格式之間的轉□，你會需要建立和維護一個不斷增長的模組集合。□了要達成對音訊數據的許多不同作業（例如，音訊混合、增加回聲、套用等化器功能、創造人工立體音效），你將編寫一系列無止盡的模組來執行這些作業。以下是你的套件可能的架構（以階層式檔案系統的方式表示）：

```
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
              ...
```

Import 套件時，Python 會搜尋 sys.path □的目□，尋找套件的子目□。

目□中必須含有 __init__.py 檔案，才會被 Pyhon 當成套件；這樣可以避免一些以常用名稱命名（例如 string）的目□，無意中隱藏了較晚出現在模組搜尋路徑中的有效模組。在最簡單的情□，__init__.py 可以只是一個空白檔案；但它也可以執行套件的初始化程式碼，或設置 __all__ 變數，之後會詳述。

套件使用者可以從套件中 import 個□模組，例如：

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

這樣就載入了子模組 sound.effects.echo。引用時必須用它的全名：

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

另一種 import 子模組的方法是：

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

這段程式碼一樣可以載入子模組 echo，□且不加套件前綴也可以使用，因此能以如下方式使用：

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

另一種變化是直接 import 所需的函式或變數：

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

同樣地，這樣也會載入子模組 echo，但它的函式 echofilter() 就可以直接使用：

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

請注意，使用 from package import item 時，item 可以是套件的子模組（或子套件），也可以是套件中被定義的名稱，像是函式、class（類🄵）或變數。import 陳述式首先測試套件中有🄵有定義該 item；如果🄵有，則會假設它是模組，🄵嘗試載入。如果還是找不到 item，則會引發 ImportError 例外。

相反地，使用 import item.subitem.subsubitem 語法時，除了最後一項之外，每一項都必須是套件；最後一項可以是模組或套件，但不能是前一項中定義的 class、函式或變數。

6.4.1 從套件中 import *

當使用者寫下 from sound.effects import * 時，會發生什🄵事？理想情🄵下，我們可能希望程式碼會去檔案系統，尋找套件中存在的子模組，🄵將它們全部 import。這會花費較長的時間，且 import 子模組的過程可能會有不必要的副作用，這些副作用只有在明確地 import 子模組時才會發生。

唯一的解法是由套件作者🄵套件提供明確的索引。import 陳述式使用以下慣例：如果套件的 __init__.py 程式碼有定義一個名🄵 __all__ 的 list，若遇到 from package import * 的時候，它就會是要被 import 的模組名稱。發布套件的新版本時，套件作者可自行🄵定是否更新此 list。如果套件作者認🄵🄵有人會從他的套件中 import *，他也可能會🄵定不支援這個 list。舉例來🄵，sound/effects/__init__.py 檔案可包含以下程式碼：

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

意思是，from sound.effects import * 將會 import sound 套件中，這三個被提名的子模組。

如果 __all__ 🄵有被定義，from sound.effects import * 陳述式🄵不會把 sound.effects 套件中所有子模組都 import 到當前的命名空間；它只保證 sound.effects 套件有被 import（可能會運行 __init__.py 中的初始化程式碼），然後 import 套件中被定義的全部名稱。這包含 __init__.py 定義（以及被明確載入的子模組）的任何名稱。它也包括任何之前被 import 陳述式明確載入的套件子模組。請看以下程式碼：

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

此例中，當 from...import 陳述式被執行時，echo 和 surround 模組被 import 進當前的命名空間，因🄵它們是在 sound.effects 套件🄵定義的。（當 __all__ 有被定義時，這規則也有效。）

雖然，有些特定模組的設計，讓你使用 import * 時，該模組只會輸出遵循特定樣式的名稱，但在正式環境 (production) 的程式碼中這仍然被視🄵是不良習慣。

記住，使用 from package import specific_submodule 不會有任何問題！實際上，這是推薦用法，除非 import 的模組需要用到的子模組和其他套件的子模組同名。

6.4.2 套件🄵引用

當套件的結構🄵多個子套件的組合時（如同範例中的 sound 套件），可以使用「🄵對 (absolute) import」，引用同層套件中的子模組。例如，要在 sound.filters.vocoder 模組中使用 sound.effects 中的 echo 模組時，可以用 from sound.effects import echo。

你也可以用 from module import name 的 import 陳述式，編寫「相對 (relative) import」。這些 import 使用前導句號指示相對 import 中的當前套件和母套件。例如，在 surround 模組中，你可以使用：

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

請注意，相對 import 的運作是以目前的模組名稱🄵依據。因🄵主模組的名稱永遠是 "__main__"，所以如果一個模組預期被用作 Python 應用程式的主模組，那它必須永遠使用🄵對 import。

6.4.3 多目🄵中的套件

套件也支援一個特殊屬性 __path__。它在初始化時是一個 list，包含該套件的 __init__.py 檔案所在的目🄵名稱，初始化時機是在這個檔案的程式碼被執行之前。這個變數可以被修改，但這樣做會影響將來對套件🄵的模組和子套件的搜尋。

雖然這個特色不太常被需要，但它可用於擴充套件中的模組集合。

🄵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：運算式陳述 *(expression statements)* 與 `print()` 函式。（第三種方法是使用檔案物件的 `write()` 方法；標準輸出的檔案是使用 `sys.stdout` 來達成的。詳細的資訊請參考對應的函式庫🅴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。

- 要使用格式化字串文本 *(formatted string literals)*，需在字串開始前的引號或連續三個引號前加上 `f` 或 `F`。你可以在這個字串中使用 `{` 與 `}` 包夾 Python 的運算式，引用變數或其他字面值 (literal values)。

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

- 字串的 `str.format()` method 需要更多手動操作。你還是可以用 `{` 和 `}` 標示欲替代變數的位置，且可給予詳細的格式指令，但你也需提供要被格式化的資訊。

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

- 最後，你還可以自己用字串切片 (slicing) 和串接 (concatenation) 操作，完成所有的字串處理，建立任何你能想像的排版格式。字串型🅴有一些 method，能以給定的欄寬填補字串，這些運算也很有用。

如果你不需要華麗的輸出，只想快速顯示變數以進行除錯，可以用 `repr()` 或 `str()` 函式把任何的值轉🅵🅴字串。

str() 函式的用意是回傳一個人類易讀的表示法，而 repr() 的用意是🄵生直譯器可讀取的表示法（如果🄵有等效的語法，則造成 SyntaxError）。如果物件🄵有人類易讀的特定表示法，str() 會回傳與 repr() 相同的值。有許多的值，像是數字，或 list 及 dictionary 等結構，使用這兩個函式會有相同的表示法。而字串，則較🄵特🄵，有兩種不同的表示法。

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'))"
```

string 模組包含一個 Template class（類🄵），提供了將值替代🄵字串的另一種方法。該方法使用 $x 🄵位符號，🄵以 dictionary 的值進行取代，但對格式的控制明顯較少。

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

格式化的字串文本（簡稱🄵 f-字串），透過在字串加入前綴 f 或 F，🄵將運算式編寫🄵 {expression}，讓你可以在字串🄵加入 Python 運算式的值。

格式🄵明符 (format specifier) 是選擇性的，寫在運算式後面，可以更好地控制值的格式化方式。以下範例將 pi 舍入到小數點後三位：

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

在 ':' 後傳遞一個整數，可以設定該欄位至少🄵幾個字元寬，常用於將每一欄對齊。

```
>>> 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
```

還有一些修飾符號可以在格式化前先將值轉🄵過。'!a' 會套用 ascii()，'!s' 會套用 str()，'!r' 會套用 repr()：

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

若要參考這些格式化字串的規格，詳見 formatspec 參考指南。

7.1.2 字串的 format() method

`str.format()` method 的基本用法如下：

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

大括號及其🄵的字元（稱🄵格式欄位）會被取代🄵傳遞給 `str.format()` method 的物件。大括號中的數字表示該物件在傳遞給 `str.format()` method 時所在的位置。

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

如果在 `str.format()` method 中使用關鍵字引數，可以使用引數名稱去引用它們的值。

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

位置引數和關鍵字引數可以任意組合：

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

如果你有一個不想分割的長格式化字串，比較好的方式是按名稱而不是按位置來引用變數。這項操作可以透過傳遞字典 (dict)，🄵用方括號 `'[]'` 使用鍵 (key) 來輕松完成。

```
>>> 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
```

用'**' 符號，把 table 當作關鍵字引數來傳遞，也有一樣的結果。

```
>>> 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
```

與🄵建函式 `vars()` 組合使用時，這種方式特🄵實用。該函式可以回傳一個包含所有區域變數的 dictionary。

例如，下面的程式碼🄵生一組排列整齊的欄，列出整數及其平方與立方：

```
>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
```

(下页继续)

(繼續上一頁)

```
 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
```

關於使用 `str.format()` 進行字串格式化的完整概述，請見 formatstrings。

7.1.3 手動格式化字串

下面是以手動格式化完成的同一個平方及立方的表：

```
>>> 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
```

（請注意，使用 `print()` 讓每欄之間加入一個空格的方法：這種方法總是在其引數間加入空格。）

字串物件的 `str.rjust()` method 透過在左側填補空格，使字串以給定的欄寬進行靠右對齊。類似的 method 還有 `str.ljust()` 和 `str.center()`。這些 method 不寫入任何🄴容，只回傳一個新字串，如果輸入的字串太長，它們不會截斷字串，而是不做任何改變地回傳；雖然這樣會弄亂欄的編排，但這通常還是比另一種情🄵好，那種情🄴會讓值變得不正確。（如果你真的想截斷字串，可以加入像 `x.ljust(n)[:n]` 這樣的切片運算。）

另一種 method 是 `str.zfill()`，可在數值字串的左邊填補零，且能識🄴正負號：

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

7.1.4 格式化字串的舊方法

% 運算子（modulo，模數）也可用於字串格式化。在 `'string' % values` 中，`string` 中所有的 `%` 會被 `values` 的零個或多個元素所取代。此運算常被稱🄴字串插值 (string interpolation)。例如：

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

更多資訊請見 old-string-formatting 小節。

7.2 讀寫檔案

`open()` 回傳一個*file object*，而它最常使用的兩個引數是：`open(filename, mode)`。

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

第一個引數是一個包含檔案名稱的字串。第二個引數是另一個字串，包含了描述檔案使用方式的幾個字元。*mode* 🄸 `'r'` 時，表示以唯讀模式開🄵檔案；🄸 `'w'` 時，表示以唯寫模式開🄴檔案（已存在的同名檔案會被抹除）；🄸 `'a'` 時，以附加🄵容🄵目的開🄵檔案，任何寫入檔案的資料會自動被加入到檔案的結尾。`'r+'` 可以開🄵檔案🄵進行讀取和寫入。*mode* 引數是選擇性的，若省略時會預設🄸 `'r'`。

通常，檔案以 *text mode* 開🄴，意即，從檔案中讀取或寫入字串時，都以特定編碼方式進行編碼。如未指定編碼方式，則預設值會取🄵於系統平台（見 `open()`）。在 mode 加上 `'b'` 會以 *binary mode* 開🄵檔案：此時，資料以位元組串物件 (bytes object) 的形式被讀寫。此模式應該使用於所有不含文字的檔案。

在文字模式 (text mode) 下，讀取時會預設把平台特定的行尾符號（Unix 上🄵 `\n`，Windows 上🄵 `\r\n`）轉🄵🄵 `\n`。在文字模式下寫入時，預設會把 `\n` 出現之處轉🄵回平台特定的行尾符號。這種在幕後對檔案資料的修改方式對文字檔案來🄵🄵有問題，但會🄵壞像是 JPEG 或 EXE 檔案中的二進制資料。在讀寫此類檔案時，注意一定要使用二進制模式。

在處理檔案物件時，使用 `with` 關鍵字是個好習慣。優點是，當它的套件結束後，即使在某個時刻引發了例外，檔案仍會正確地被關閉。使用 `with` 也比寫等效的 `try-finally` 區塊，來得簡短許多：

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

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

如果你🄵有使用 `with` 關鍵字，則應呼叫 `f.close()` 關閉檔案，可以立即釋放被它所使用的系統資源。

> **警告：** 呼叫 `f.write()` 時，若未使用 `with` 關鍵字或呼叫 `f.close()`，即使程式成功退出，也**可能**導致 `f.write()` 的引數🄵有被完全寫入硬碟。

不論是透過 `with` 陳述式，或呼叫 `f.close()` 關閉一個檔案物件之後，嘗試使用該檔案物件將會自動失效。

```
>>> 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 檔案物件的 method

本節其余的範例皆假設一個名🄵 `f` 的檔案物件已被建立。

要讀取檔案的🄵容，可呼叫 `f.read(size)`，它可讀取一部份的資料，🄵以字串（文字模式）或位元組串物件（二進制模式）形式回傳。*size* 是個選擇性的數字引數。當 *size* 被省略或🄵負數時，檔案的全部🄵容會被讀取🄵回傳；如果檔案是機器記憶體容量的兩倍大時，這會是你的問題。否則，最多只有等同於 *size* 數量的字元（文字模式）或 *size* 數量的位元組串（二進制模式）會被讀取及回傳。如果之前已經到達檔案的末端，`f.read()` 會回傳空字串（`''`）。

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

`f.readline()` 從檔案中讀取單獨一行；🄵行字元（`\n`）會被留在字串的結尾，只有當檔案末端不是🄵行字元時，它才會在檔案的最後一行被省略。這種方式讓回傳值清晰明確；只要 `f.readline()` 回傳一個空字串，就表示已經到達了檔案末端，而空白行的表示法是 `'\n'`，也就是只含一個🄵行字元的字串。

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

想從檔案中讀取多行時，可以對檔案物件進行🄵圈。這種方法能有效地使用記憶體、快速，且程式碼簡潔：

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

如果你想把一個檔案的所有行讀進一個 list 🄵，可以用 `list(f)` 或 `f.readlines()`。

`f.write(string)` 把 *string* 的🄵容寫入檔案，🄵回傳寫入的字元數。

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

寫入其他類型的物件之前，要先把它們轉🄵🄵字串（文字模式）或位元組串物件（二進制模式）：

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

`f.tell()` 回傳一個整數，它給出檔案物件在檔案中的當前位置，在二進制模式下表示🄵檔案開始至今的位元組數，在文字模式下表示🄵一個意義不明的數字。

使用 `f.seek(offset, whence)` 可以改變檔案物件的位置。位置計算方法是從一個參考點增加 *offset* 的偏移量；參考點則由引數 *whence* 來選擇。當 *whence* 值🄵 0 時，表示使用檔案開頭，1 表示使用當前的檔案位置，2 表示使用檔案末端作🄵參考點。*whence* 可省略，其預設值🄵 0，即以檔案開頭作🄵參考點。

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
```

(下页继续)

(繼續上一頁)

```
>>> f.seek(5)      # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

在文字檔案（開🄴時模式字串未加入 b 的檔案）中，只允許以檔案開頭🄵參考點進行尋找（但 `seek(0, 2)` 尋找檔案最末端是例外），且只有從 `f.tell()` 回傳的值，或是 0，才是有效的 *offset* 值。其他任何 *offset* 值都會🄵生未定義的行🄴。

檔案物件還有一些附加的 method，像是較不常使用的 `isatty()` 和 `truncate()`；檔案物件的完整指南詳見程式庫參考手🄵。

7.2.2 使用 `json` 儲存結構化資料

字串可以簡單地從檔案中被寫入和讀取。數字則稍嫌麻煩，因🄵 `read()` method 只回傳字串，這些字串必須傳遞給像 `int()` 這樣的函式，它接受 `'123'` 這樣的字串，🄵回傳數值 123。當你想儲存像是巢狀 list 和 dictionary（字典）等🄵雜的資料類型時，手動剖析 (parsing) 和序列化 (serializing) 就變得🄵雜。

相較於讓使用者不斷地編寫和除錯程式碼才能把🄵雜的資料類型儲存到檔案，Python 支援一個普及的資料交🄵格式，稱🄵 JSON (JavaScript Object Notation)。標準模組 `json` 可接收 Python 資料階層，🄵將它們轉🄵🄵字串表示法；這個過程稱🄵 *serializing*（序列化）。從字串表示法中重建資料則稱🄵 *deserializing*（反序列化）。在序列化和反序列化之間，表示物件的字串可以被儲存在檔案或資料中，或通過網路連接發送到遠端的機器。

備🄵： JSON 格式經常地使用於現代應用程式的資料交🄵。許多程序設計師早已對它耳熟能詳，使它成🄵提升互操作性 (interoperability) 的好選擇。

如果你有一個物件 `x`，只需一行簡單的程式碼即可檢視它的 JSON 字串表示法：

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

`dumps()` 函式有一個變體，稱🄵 `dump()`，它單純地將物件序列化🄵*text file*。因此，如果 `f` 是一個🄵了寫入而開🄵的*text file* 物件，我們可以這樣做：

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

若 `f` 是一個已開🄵、可讀取的*text file* 物件，要再次解碼物件的話：

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

這種簡單的序列化技術可以處理 list 和 dictionary，但要在 JSON 中序列化任意的 class（類🄵）實例，則需要一些額外的工作。`json` 模組的參考資料包含對此的🄵明。

也參考：

`pickle` - pickle 模組

與*JSON* 不同，*pickle* 是一種允許對任意的🄵雜 Python 物件進行序列化的協定。因此，它🄵 Python 所特有，不能用於與其他語言編寫的應用程式溝通。在預設情🄵，它也是不安全的：如果資料是由手段高明的攻擊者精心設計，將這段來自於不受信任來源的 pickle 資料反序列化，可以執行任意的程式碼。

CHAPTER 8

錯誤和例外

到目前F止還F有提到錯誤訊息，但如果你嘗試運行範例，你可能會發現一些錯誤訊息。常見的（至少）兩種不同的錯誤類FF：*語法錯誤 (syntax error)* 和*例外 (exception)*。

8.1 語法錯誤 (Syntax Error)

語法錯誤又稱剖析錯誤 (parsing error)，它或許是學習 Python 的過程最常聽見的抱怨：

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

剖析器 (parser) 會重F犯錯的那一行，F用一個小「箭頭」指向該行檢測到的第一個錯誤點。錯誤是由箭頭之前的標記 (token) 導致的（或至少是在這F檢測到的）：此例中，錯誤是在 `print()` 函式中被檢測到，因F在它前面少了一個冒號（`':'`）。檔案名稱和行號會被印出來，所以如果訊息是來自F本時，就可以知道去哪F找問題。

8.2 例外 (Exception)

即使一段陳述式或運算式使用了正確的語法，嘗試執行時仍可能導致錯誤。執行時檢測到的錯誤稱F*例外*，例外不一定都很嚴重：你很快就能學會在 Python 程式中如何處理它們。不過大多數的例外不會被程式處理，F且會顯示如下的錯誤訊息：

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

(下页继续)

(繼續上一頁)

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

錯誤訊息的最後一行指示發生了什F事。例外有不同的類型，而類型名稱會作F訊息的一部份被印出。範例中的例外類型F：ZeroDivisionError、NameError 和 TypeError。作F例外類型被印出的字串，就是發生的F建例外 (built-in exception) 的名稱。所有的F建例外都是如此運作，但對於使用者自定的例外則不一定需要遵守（雖然這是一個有用的慣例）。標准例外名稱是F建的識F字 (identifier)，不是保留關鍵字 (reserved keyword)。

此行其余部分，根據例外的類型及導致例外的原因，F明例外的細節。

錯誤訊息的開頭，用堆F回溯 (stack traceback) 的形式顯示發生例外的語境。一般來F，它含有一個列出源程式碼行 (source line) 的堆F回溯；但它不會顯示從標准輸入中讀取的程式碼。

bltin-exceptions 章節列出F建的例外及它們的意義。

8.3 處理例外

編寫程式處理選定的例外是可行的。以下範例會要求使用者輸入F容，直到有效的整數被輸入F止，但它允許使用者中斷程式（使用 Control-C 或作業系統支援的指令）；請注意，由使用者F生的程式中斷會引發 KeyboardInterrupt 例外信號。

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

try 陳述式運作方式如下。

- 首先，執行 *try* 子句（try 和 except 關鍵字之間的陳述式）。
- 如果F有發生例外，則 *except* 子句會被跳過，try 陳述式執行完畢。
- 如果執行 try 子句時發生了例外，則該子句中剩下的部分會被跳過。如果例外的類型與 except 關鍵字後面的例外名稱相符，則 except 子句被執行，然後，繼續執行 try 陳述式之後的程式碼。
- 如果發生的例外未符合 except 子句中的例外名稱，則將其傳遞到外層的 try 陳述式；如果仍無法找到處理者，則它是一個*未處理例外 (unhandled exception)*，執行將停止，F顯示如上所示的訊息。

try 陳述式可以有不只一個 except 子句，F不同的例外指定處理者，而最多只有一個處理者會被執行。處理者只處理對應的 try 子句中發生的例外，而不會處理同一 try 陳述式F其他處理者F的例外。一個 except 子句可以用一組括號F的 tuple 列舉多個例外，例如：

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

一個在 except 子句中的 class（類F）和一個例外是可相容的，只要它與例外是同一個 class 或是F其 base class（基底類F）；反之則無法成立——列出 derived class（衍生類F）的 except 子句F不能與 base class 相容。例如，以下程式碼會依序印出 B、C、D：

```
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")
```

請注意，如果 except 子句的順序被反轉（把 `except B` 放到第一個），則會印出 B、B、B ——第一個符合的 except 子句會被觸發。

最後一個 except 子句可以省略例外名稱，以統一處理所有其他例外。但使用上要極其小心，因🄵這種方式容易遮蔽真正的程式設計錯誤！它也可用於印出錯誤訊息，然後重新引發例外（也讓呼叫者可以處理該例外）：

```
import sys

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

`try` ... `except` 陳述式有一個選擇性的 *else* 子句，使用時，該子句必須放在所有 except 子句之後。如果一段程式碼必須被執行，但 try 子句又🄵有引發例外時，這個子句很有用。例如：

```
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()
```

使用 `else` 子句比向 `try` 子句添加額外的程式碼要好，因🄵這可以避免意外地捕獲不是由 `try` ... `except` 陳述式保護的程式碼所引發的例外。

當例外發生時，它可能有一個相關的值，也就是例外的*引數*。此引數的存在與否及它的類型，是取🄵於例外的類型。

`except` 子句可以在例外名稱後面指定一個變數。這個變數被綁定到一個例外實例 (instance)，其引數儲存在 `instance.args` 中。🄵了方便，例外實例會定義 `__str__()`，因此引數可以直接被印出而無須引用

.args。你也可以在引發例外前就先建立一個例外實例，🄵隨心所欲地🄵它加入任何屬性。

```
>>> 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
```

如果一個例外有引數，則它們會被印在未處理例外的訊息的最後一部分（「細節」）。

例外的處理者不僅處理 try 子句🄵立即發生的例外，還處理 try 子句🄵（即使是間接地）呼叫的函式🄵部發生的例外。例如：

```
>>> 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 引發例外

raise 陳述式可讓程式設計師🄵制引發指定的例外。例如：

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

raise 唯一的引數就是要引發的例外。該引數必須是一個例外實例或例外 class（衍生自 Exception 的 class）。如果一個例外 class 被傳遞，它會不含引數地呼叫它的建構函式 (constructor)，使它被自動建立實例 (implicitly instantiated)：

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

如果你只想判斷是否引發了例外，但🄵不打算處理它，則可以使用簡單的 raise 陳述式來重新引發該例外：

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print('An exception flew by!')
```

(下页继续)

(繼續上一頁)

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

8.5 例外鏈接 (Exception Chaining)

raise 陳述式容許一個選擇性的 from，它透過被引發例外中的 __cause__ 屬性的設定，來🄴用例外鏈接。例如：

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

要變🄴例外時，這種方式很有用。例如：

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

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

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

當例外是在一個 except 或 finally 段落的🄴部被引發時，例外鏈接會自動發生。要禁止例外鏈接，可以使用慣用語 from None：

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

更多關於鏈接機制的資訊，詳見 bltin-exceptions。

8.6 使用者自定的例外

程式可以通過建立新的例外 class 來命名自己的例外（深入了解 Python class，詳見类）。不論是直接還是間接地，例外通常應該從 `Exception` class 衍生出來。

例外 class 可被定義來做任何其他 class 能🄵做的事，但通常會讓它維持簡單，只提供一些屬性，讓關於錯誤的資訊可被例外的處理者抽取出來。在建立一個可能引發多種不同錯誤的模組時，常見做法是🄵該模組定義的例外建立一個 base class，以及多個 subclass，其能🄵不同錯誤情🄵建立特定的例外 class：

```
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
```

大多數的例外定義，都會以「Error」作🄵名稱結尾，類似於標準例外的命名。

許多標准模組會定義它們自己的例外，以報告在其定義的函式中發生的錯誤。更多有關 class 的資訊，詳見类章節。

8.7 定義清理動作

`try` 陳述式有另一個選擇性子句，用於定義在所有情🄵下都必須被執行的清理動作。例如：

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
```

(下页继续)

(繼續上一頁)

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

如果 finally 子句存在，則 finally 子句會是 try 陳述式結束前執行的最後一項任務。不論 try 陳述式是否🄵生例外，都會執行 finally 子句。以下幾點將探討例外發生時，比較🄴雜的情🄵：

- 若一個例外發生於 try 子句的執行過程，則該例外會被某個 except 子句處理。如果該例外🄵有被 except 子句處理，它會在 finally 子句執行後被重新引發。
- 一個例外可能發生於 except 或 else 子句的執行過程。同樣地，該例外會在 finally 子句執行後被重新引發。
- 如果 finally 子句執行 break、continue 或 return 陳述式，則例外不會被重新引發。
- 如果 try 陳述式遇到 break、continue 或 return 陳述式，則 finally 子句會在執行 break、continue 或 return 陳述式之前先執行。
- 如果 finally 子句中包含 return 陳述式，則回傳值會是來自 finally 子句的 return 陳述式的回傳值，而不是來自 try 子句的 return 陳述式的回傳值。

例如：

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

另一個比較🄴雜的範例：

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

如你所見，finally 子句在任何情🄵下都會被執行。兩個字串相除所引發的 TypeError 🄵有被 except 子句處理，因此會在 finally 子句執行後被重新引發。

在真實應用程式中，finally 子句對於釋放外部資源（例如檔案或網路連🄵）很有用，無論該資源的使用是否成功。

8.8 預定義的清理動作

某些物件定義了在物件不再被需要時的標准清理動作，無論使用該物件的作業是成功或失敗。請看以下範例，它嘗試開🄵一個檔案，🄵印出檔案🄵容至螢幕。

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

這段程式碼的問題在於，執行完該程式碼後，它讓檔案在一段不確定的時間🄵處於開🄵狀態。在簡單🄵本中這不是問題，但對於較大的應用程式來🄵可能會是個問題。with 陳述式讓物件（例如檔案）在被使用時，能保證它們總是及時、正確地被清理。

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

陳述式執行完畢後，就算是在處理🄵容時遇到問題，檔案 *f* 總是會被關閉。和檔案一樣，提供預定義清理動作的物件會在🄵明文件中表明這一點。

CHAPTER 9

类

类把数据与功能绑定在一起。创建新类就是创建新的对象 *类型*，从而创建该类型的新 *实例*。类实例具有多种保持自身状态的属性。类实例还支持（由类定义的）修改自身状态的方法。

和其他编程语言相比，Python 只用了很少的新语法和语义就加入了类。Python 的类是 C++ 和 Modula-3 中类机制的结合体，而且支持所有面向对象编程（OOP）的标准特性：类继承机制支持多个基类，派生类可以覆盖基类的任何方法，类的方法可以调用基类中相同名称的方法。对象可以包含任意数量和类型的数据。和模块一样，类也拥有 Python 天然的动态特性：在运行时创建，创建后也可以修改。

在 C++ 术语中，通常类成员（包括数据成员）是 *public* (例外见下文私有变量)，所有成员函数都是 *virtual*。与在 Modula-3 中一样，没有用于从对象的方法中引用对象成员的简写：方法函数在声明时，有一个显示的参数代表本对象，该参数由调用隐式提供。与 Smalltalk 一样，类本身也是对象。这为导入和重命名提供了语义。与 C++ 和 Modula-3 不同，内置类型可以用作基类，供用户扩展。此外，与 C++ 一样，大多数具有特殊语法（算术运算符，下标等）的内置运算符都可以为类实例而重新定义。

（由于缺乏关于类的公认术语，我会偶尔使用 Smalltalk、C++ 的术语，我还会使用 Modula-3 的术语，因为它的面向对象语义比 C++ 更接近 Python，但估计没几个读者听说过这门语言。）

9.1 名称和对象

对象之间相互独立，多个名称（在多个作用域内）可以绑定到同一个对象。其他语言称之为别名。Python 初学者通常不容易理解这个概念，处理数字、字符串、元组等不可变基本类型时，可以不必理会。但是，对涉及可变对象，如列表、字典等大多数其他类型的 Python 代码的语义，别名可能会产生意料之外的效果。这样做，通常是为了让程序受益，因为别名在某些方面就像指针。例如，传递对象的代价很小，因为实现只传递一个指针；如果函数修改了作为参数传递的对象，调用者就可以看到更改 --- 无需 Pascal 用两个不同参数的传递机制。

9.2 Python 作用域和命名空间

在介绍类之前，我首先要告诉你一些 Python 的作用域规则。类定义对命名空间有一些巧妙的技巧，你需要知道作用域和命名空间如何工作才能完全理解正在发生的事情。顺便说一下，关于这个主题的知识对任何高级 Python 程序员都很有用。

让我们从一些定义开始。

namespace（命名空间）是一个从名字到对象的映射。当前大部分命名空间都由 Python 字典实现，但一般情况下基本不会去关注它们（除了要面对性能问题时），而且也有可能在将来更改。下面是几个命名空间的例子：存放内置函数的集合（包含 `abs()` 这样的函数，和内建的异常等）；模块中的全局名称；函数调用中的局部名称。从某种意义上说，对象的属性集合也是一种命名空间的形式。关于命名空间的重要一点是，不同命名空间中的名称之间绝对没有关系；例如，两个不同的模块都可以定义一个 `maximize` 函数而不会产生混淆 --- 模块的用户必须在其前面加上模块名称。

顺便说明一下，我把任何跟在一个点号之后的名称都称为 *属性* --- 例如，在表达式 `z.real` 中，`real` 是对象 `z` 的一个属性。按严格的说法，对模块中名称的引用属于属性引用：在表达式 `modname.funcname` 中，`modname` 是一个模块对象而 `funcname` 是它的一个属性。在此情况下在模块的属性和模块中定义的全局名称之间正好存在一个直观的映射：它们共享相同的命名空间![1]

属性可以是只读或者可写的。如果为后者，那么对属性的赋值是可行的。模块属性是可写的，你可以写 `modname.the_answer = 42`。可写的属性同样可以用 `del` 语句删除。例如，`del modname.the_answer` 将会从名为 `modname` 的对象中移除 `the_answer` 属性。

命名空间在不同时刻被创建，拥有不同的生存期。包含内置名称的命名空间是在 Python 解释器启动时创建的，永远不会被删除。模块的全局命名空间在模块定义被读入时创建；通常，模块命名空间也会持续到解释器退出。被解释器的顶层调用执行的语句，从一个脚本文件读取或交互式地读取，被认为是 `__main__` 模块调用的一部分，因此它们拥有自己的全局命名空间。（内置名称实际上也存在于一个模块中；这个模块被称作 `builtins`。）

一个函数的本地命名空间在这个函数被调用时创建，并在函数返回或抛出一个不在函数内部处理的错误时被删除。（事实上，比起描述到底发生了什么，忘掉它更好。）当然，每次递归调用都会有它自己的本地命名空间。

一个 *作用域* 是一个命名空间可直接访问的 Python 程序的文本区域。这里的“可直接访问”意味着对名称的非限定引用会尝试在命名空间中查找名称。

虽然作用域是静态地确定的，但它们会被动态地使用。在执行期间的任何时刻，会有 3 或 4 个命名空间可被直接访问的嵌套作用域:

- 最先搜索的最内部作用域包含局部名称
- 从最近的封闭作用域开始搜索的任何封闭函数的作用域包含非局部名称，也包括非全局名称
- 倒数第二个作用域包含当前模块的全局名称
- 最外面的作用域（最后搜索）是包含内置名称的命名空间

如果一个名称被声明为全局变量，则所有引用和赋值将直接指向包含该模块的全局名称的中间作用域。要重新绑定在最内层作用域以外找到的变量，可以使用 `nonlocal` 语句声明为非本地变量。如果没有被声明为非本地变量，这些变量将是只读的（尝试写入这样的变量只会在最内层作用域中创建一个 *新的* 局部变量，而同名的外部变量保持不变）。

通常，当前局部作用域将（按字面文本）引用当前函数的局部名称。在函数以外，局部作用域将引用与全局作用域相一致的命名空间：模块的命名空间。类定义将在局部命名空间内再放置另一个命名空间。

重要的是应该意识到作用域是按字面文本来确定的：在一个模块内定义的函数的全局作用域就是该模块的命名空间，无论该函数从什么地方或以什么别名被调用。另一方面，实际的名称搜索是在运行时动态完成的

[1] 存在一个例外。模块对象有一个秘密的只读属性 `__dict__`，它返回用于实现模块命名空间的字典；`__dict__` 是属性但不是全局名称。显然，使用这个将违反命名空间实现的抽象，应当仅被用于事后调试器之类的场合。

--- 但是，Python 正在朝着“编译时静态名称解析”的方向发展，因此不要过于依赖动态名称解析！（事实上，局部变量已经是被静态确定了。）

Python 的一个特殊规定是这样的 -- 如果不存在生效的 `global` 或 `nonlocal` 语句 -- 则对名称的赋值总是会进入最内层作用域。赋值不会复制数据 --- 它们只是将名称绑定到对象。删除也是如此：语句 `del x` 会从局部作用域所引用的命名空间中移除对 `x` 的绑定。事实上，所有引入新名称的操作都是使用局部作用域：特别地，`import` 语句和函数定义会在局部作用域中绑定模块或函数名称。

`global` 语句可被用来表明特定变量生存于全局作用域并且应当在其中被重新绑定；`nonlocal` 语句表明特定变量生存于外层作用域中并且应当在其中被重新绑定。

9.2.1 作用域和命名空间示例

这个例子演示了如何引用不同作用域和名称空间，以及 `global` 和 `nonlocal` 会如何影响变量绑定:

```
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)
```

示例代码的输出是:

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

请注意 *局部*赋值（这是默认状态）不会改变 *scope_test* 对 *spam* 的绑定。`nonlocal` 赋值会改变 *scope_test* 对 *spam* 的绑定，而 `global` 赋值会改变模块层级的绑定。

您还可以发现在 `global` 赋值之前没有 *spam* 的绑定。

9.3 初探类

类引入了一些新语法，三种新对象类型和一些新语义。

9.3.1 类定义语法

最简单的类定义看起来像这样:

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

类定义与函数定义 (def 语句) 一样必须被执行才会起作用。(你可以尝试将类定义放在 if 语句的一个分支或是函数的内部。)

在实践中，类定义内的语句通常都是函数定义，但也允许有其他语句，有时还很有用 --- 我们会稍后再回来说明这个问题。在类内部的函数定义通常具有一种特别形式的参数列表，这是方法调用的约定规范所指明的 --- 这个问题也将在稍后再说明。

当进入类定义时，将创建一个新的命名空间，并将其用作局部作用域 --- 因此，所有对局部变量的赋值都是在这个新命名空间之内。特别的，函数定义会绑定到这里的新函数名称。

当（从结尾处）正常离开类定义时，将创建一个 *类对象*。这基本上是一个包围在类定义所创建命名空间内容周围的包装器；我们将在下一节了解有关类对象的更多信息。原始的（在进入类定义之前起作用的）局部作用域将重新生效，类对象将在这里被绑定到类定义头所给出的类名称 (在这个示例中为 ClassName)。

9.3.2 类对象

类对象支持两种操作：属性引用和实例化。

属性引用 使用 Python 中所有属性引用所使用的标准语法: obj.name。有效的属性名称是类对象被创建时存在于类命名空间中的所有名称。因此，如果类定义是这样的:

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

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

那么 MyClass.i 和 MyClass.f 就是有效的属性引用，将分别返回一个整数和一个函数对象。类属性也可以被赋值，因此可以通过赋值来更改 MyClass.i 的值。__doc__ 也是一个有效的属性，将返回所属类的文档字符串: "A simple example class"。

类的 *实例化* 使用函数表示法。可以把类对象视为是返回该类的一个新实例的不带参数的函数。举例来说（假设使用上述的类）:

```
x = MyClass()
```

创建类的新 *实例* 并将此对象分配给局部变量 x。

实例化操作（“调用”类对象）会创建一个空对象。许多类喜欢创建带有特定初始状态的自定义实例。为此类定义可能包含一个名为 __init__() 的特殊方法，就像这样:

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

当一个类定义了 __init__() 方法时，类的实例化操作会自动为新创建的类实例发起调用 __init__()。因此在这个示例中，可以通过以下语句获得一个经初始化的新实例:

```
x = MyClass()
```

当然，__init__() 方法还可以有额外参数以实现更高灵活性。在这种情况下，提供给类实例化运算符的参数将被传递给 __init__()。例如，:

```
>>> 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 实例对象

现在我们能用实例对象做什么？实例对象所能理解的唯一操作是属性引用。有两种有效的属性名称：数据属性和方法。

*数据属性*对应于 Smalltalk 中的“实例变量”，以及 C++ 中的“数据成员”。数据属性不需要声明；像局部变量一样，它们将在第一次被赋值时产生。例如，如果 x 是上面创建的 MyClass 的实例，则以下代码段将打印数值 16，且不保留任何追踪信息:

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

另一类实例属性引用称为*方法*。方法是“从属于”对象的函数。(在 Python 中，方法这个术语并不是类实例所特有的：其他对象也可以有方法。例如，列表对象具有 append, insert, remove, sort 等方法。然而，在以下讨论中，我们使用方法一词将专指类实例对象的方法，除非另外显式地说明。)

实例对象的有效方法名称依赖于其所属的类。根据定义，一个类中所有是函数对象的属性都是定义了其实例的相应方法。因此在我们的示例中，x.f 是有效的方法引用，因为 MyClass.f 是一个函数，而 x.i 不是方法，因为 MyClass.i 不是函数。但是 x.f 与 MyClass.f 并不是一回事 --- 它是一个*方法对象*，不是函数对象。

9.3.4 方法对象

通常，方法在绑定后立即被调用:

```
x.f()
```

在 MyClass 示例中，这将返回字符串 'hello world'。但是，立即调用一个方法并不是必须的: x.f 是一个方法对象，它可以被保存起来以后再调用。例如:

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

将持续打印 `hello world`，直到结束。

当一个方法被调用时到底发生了什么？你可能已经注意到上面调用 `x.f()` 时并没有带参数，虽然 `f()` 的函数定义指定了一个参数。这个参数发生了什么事？当不带参数地调用一个需要参数的函数时 Python 肯定会引发异常 --- 即使参数实际未被使用...

实际上，你可能已经猜到了答案：方法的特殊之处就在于实例对象会作为函数的第一个参数被传入。在我们的示例中，调用 `x.f()` 其实就相当于 `MyClass.f(x)`。总之，调用一个具有 *n* 个参数的方法就相当于调用再多一个参数的对应函数，这个参数值为方法所属实例对象，位置在其他参数之前。

如果你仍然无法理解方法的运作原理，那么查看实现细节可能会弄清楚问题。当一个实例的非数据属性被引用时，将搜索实例所属的类。如果被引用的属性名称表示一个有效的类属性中的函数对象，会通过打包（指向）查找到的实例对象和函数对象到一个抽象对象的方式来创建方法对象：这个抽象对象就是方法对象。当附带参数列表调用方法对象时，将基于实例对象和参数列表构建一个新的参数列表，并使用这个新参数列表调用相应的函数对象。

9.3.5 类和实例变量

一般来说，实例变量用于每个实例的唯一数据，而类变量用于类的所有实例共享的属性和方法:

```
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'
```

正如名称和对象 中已讨论过的，共享数据可能在涉及*mutable* 对象例如列表和字典的时候导致令人惊讶的结果。例如以下代码中的 *tricks* 列表不应该被用作类变量，因为所有的 *Dog* 实例将只共享一个单独的列表:

```
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')
```

(下页继续)

(繼續上一頁)

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

正确的类设计应该使用实例变量:

```
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 补充说明

如果同样的属性名称同时出现在实例和类中，则属性查找会优先选择实例:

```
>>> class Warehouse:
        purpose = 'storage'
        region = 'west'

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

数据属性可以被方法以及一个对象的普通用户（“客户端”）所引用。换句话说，类不能用于实现纯抽象数据类型。实际上，在 Python 中没有任何东西能强制隐藏数据 --- 它是完全基于约定的。（而在另一方面，用 C 语言编写的 Python 实现则可以完全隐藏实现细节，并在必要时控制对象的访问；此特性可以通过用 C 编写 Python 扩展来使用。）

客户端应当谨慎地使用数据属性 --- 客户端可能通过直接操作数据属性的方式破坏由方法所维护的固定变量。请注意客户端可以向一个实例对象添加他们自己的数据属性而不会影响方法的可用性，只要保证避免名称冲突 --- 再次提醒，在此使用命名约定可以省去许多令人头痛的麻烦。

在方法内部引用数据属性（或其他方法！）并没有简便方式。我发现这实际上提升了方法的可读性：当浏览一个方法代码时，不会存在混淆局部变量和实例变量的机会。

方法的第一个参数常常被命名为 `self`。这也不过就是一个约定: `self` 这一名称在 Python 中绝对没有特殊含义。但是要注意，不遵循此约定会使得你的代码对其他 Python 程序员来说缺乏可读性，而且也可以想像一个类浏览器程序的编写可能会依赖于这样的约定。

任何一个作为类属性的函数都为该类的实例定义了一个相应方法。函数定义的文本并非必须包含于类定义之内：将一个函数对象赋值给一个局部变量也是可以的。例如:

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

class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g
```

现在 `f`, `g` 和 `h` 都是 `C` 类的引用函数对象的属性，因而它们就都是 `C` 的实例的方法 --- 其中 `h` 完全等同于 `g`。但请注意，本示例的做法通常只会令程序的阅读者感到迷惑。

方法可以通过使用 `self` 参数的方法属性调用其他方法:

```
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)
```

方法可以通过与普通函数相同的方式引用全局名称。与方法相关联的全局作用域就是包含其定义的模块。(类永远不会被作为全局作用域。) 虽然我们很少会有充分的理由在方法中使用全局作用域，但全局作用域存在许多合理的使用场景：举个例子，导入到全局作用域的函数和模块可以被方法所使用，在其中定义的函数和类也一样。通常，包含该方法的类本身是在全局作用域中定义的，而在下一节中我们将会发现为何方法需要引用其所属类的很好的理由。

每个值都是一个对象，因此具有 *类* (也称为 *类型*)，并存储为 `object.__class__` 。

9.5 继承

当然，如果不支持继承，语言特性就不值得称为“类”。派生类定义的语法如下所示:

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

名称 `BaseClassName` 必须定义于包含派生类定义的作用域中。也允许用其他任意表达式代替基类名称所在的位置。这有时也可能会用得上，例如，当基类定义在另一个模块中的时候:

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

派生类定义的执行过程与基类相同。当构造类对象时，基类会被记住。此信息将被用来解析属性引用：如果请求的属性在类中找不到，搜索将转往基类中进行查找。如果基类本身也派生自其他某个类，则此规则将被递归地应用。

派生类的实例化没有任何特殊之处：`DerivedClassName()` 会创建该类的一个新实例。方法引用将按以下方式解析：搜索相应的类属性，如有必要将按基类继承链逐步向下查找，如果产生了一个函数对象则方法引用就生效。

派生类可能会重写其基类的方法。因为方法在调用同一对象的其他方法时没有特殊权限，所以调用同一基类中定义的另一方法的基类方法最终可能会调用覆盖它的派生类的方法。（对 C++ 程序员的提示：Python 中所有的方法实际上都是 `virtual` 方法。）

在派生类中的重载方法实际上可能想要扩展而非简单地替换同名的基类方法。有一种方式可以简单地直接调用基类方法：即调用 `BaseClassName.methodname(self, arguments)`。有时这对客户端来说也是有用的。（请注意仅当此基类可在全局作用域中以 `BaseClassName` 的名称被访问时方可使用此方式。）

Python 有两个内置函数可被用于继承机制：

- 使用 `isinstance()` 来检查一个实例的类型：`isinstance(obj, int)` 仅会在 `obj.__class__` 为 `int` 或某个派生自 `int` 的类时为 `True`。
- 使用 `issubclass()` 来检查类的继承关系：`issubclass(bool, int)` 为 `True`，因为 `bool` 是 `int` 的子类。但是，`issubclass(float, int)` 为 `False`，因为 `float` 不是 `int` 的子类。

9.5.1 多重继承

Python 也支持一种多重继承。带有多个基类的类定义语句如下所示：

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

对于多数应用来说，在最简单的情况下，你可以认为搜索从父类所继承属性的操作是深度优先、从左至右的，当层次结构中存在重叠时不会在同一个类中搜索两次。因此，如果某一属性在 `DerivedClassName` 中未找到，则会到 `Base1` 中搜索它，然后（递归地）到 `Base1` 的基类中搜索，如果在那里未找到，再到 `Base2` 中搜索，依此类推。

真实情况比这个更复杂一些；方法解析顺序会动态改变以支持对 `super()` 的协同调用。这种方式在某些其他多重继承型语言中被称为后续方法调用，它比单继承型语言中的 super 调用更强大。

动态改变顺序是有必要的，因为所有多重继承的情况都会显示出一个或更多的菱形关联（即至少有一个父类可通过多条路径被最底层类所访问）。例如，所有类都是继承自 `object`，因此任何多重继承的情况都提供了一条以上的路径可以通向 `object`。为了确保基类不会被访问一次以上，动态算法会用一种特殊方式将搜索顺序线性化，保留每个类所指定的从左至右的顺序，只调用每个父类一次，并且保持单调（即一个类可以被子类化而不影响其父类的优先顺序）。总而言之，这些特性使得设计具有多重继承的可靠且可扩展的类成为可能。要了解更多细节，请参阅 https://www.python.org/download/releases/2.3/mro/。

9.6 私有变量

那种仅限从一个对象内部访问的“私有”实例变量在 Python 中并不存在。但是，大多数 Python 代码都遵循这样一个约定：带有一个下划线的名称 (例如 _spam) 应该被当作是 API 的非公有部分 (无论它是函数、方法或是数据成员)。这应当被视为一个实现细节，可能不经通知即加以改变。

由于存在对于类私有成员的有效使用场景（例如避免名称与子类所定义的名称相冲突），因此存在对此种机制的有限支持，称为 *名称改写*。任何形式为 __spam 的标识符（至少带有两个前缀下划线，至多一个后缀下划线）的文本将被替换为 _classname__spam，其中 classname 为去除了前缀下划线的当前类名称。这种改写不考虑标识符的句法位置，只要它出现在类定义内部就会进行。

名称改写有助于让子类重载方法而不破坏类内方法调用。例如:

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

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

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

class MappingSubclass(Mapping):

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

上面的示例即使在 MappingSubclass 引入了一个 __update 标识符的情况下也不会出错，因为它会在 Mapping 类中被替换为 _Mapping__update 而在 MappingSubclass 类中被替换为 _MappingSubclass__update。

请注意，改写规则的设计主要是为了避免意外冲突；访问或修改被视为私有的变量仍然是可能的。这在特殊情况下甚至会很有用，例如在调试器中。

请注意传递给 exec() 或 eval() 的代码不会将发起调用类的类名视作当前类；这类似于 global 语句的效果，因此这种效果仅限于同时经过字节码编译的代码。同样的限制也适用于 getattr(), setattr() 和 delattr()，以及对于 __dict__ 的直接引用。

9.7 杂项说明

有时会需要使用类似于 Pascal 的“record”或 C 的“struct”这样的数据类型，将一些命名数据项捆绑在一起。这种情况适合定义一个空类:

```
class Employee:
    pass

john = Employee()  # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
```

(下页继续)

(繼續上一頁)

```
john.dept = 'computer lab'
john.salary = 1000
```

一段需要特定抽象数据类型的 Python 代码往往可以被传入一个模拟了该数据类型的方法的类作为替代。例如，如果你有一个基于文件对象来格式化某些数据的函数，你可以定义一个带有 read() 和 readline() 方法从字符串缓存获取数据的类，并将其作为参数传入。

实例方法对象也具有属性: m.__self__ 就是带有 m() 方法的实例对象，而 m.__func__ 则是该方法所对应的函数对象。

9.8 迭代器

到目前为止，您可能已经注意到大多数容器对象都可以使用 for 语句:

```
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='')
```

这种访问风格清晰、简洁又方便。迭代器的使用非常普遍并使得 Python 成为一个统一的整体。在幕后，for 语句会在容器对象上调用 iter()。该函数返回一个定义了 __next__() 方法的迭代器对象，此方法将逐一访问容器中的元素。当元素用尽时，__next__() 将引发 StopIteration 异常来通知终止 for 循环。你可以使用 next() 内置函数来调用 __next__() 方法；这个例子显示了它的运作方式:

```
>>> 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
```

看过迭代器协议的幕后机制，给你的类添加迭代器行为就很容易了。定义一个 __iter__() 方法来返回一个带有 __next__() 方法的对象。如果类已定义了 __next__()，则 __iter__() 可以简单地返回 self:

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

(下页继续)

(繼續上一頁)

```
    def __iter__(self):
        return self

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

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

9.9 生成器

生成器 是一个用于创建迭代器的简单而强大的工具。它们的写法类似于标准的函数，但当它们要返回数据时会使用 `yield` 语句。每次在生成器上调用 `next()` 时，它会从上次离开的位置恢复执行（它会记住上次执行语句时的所有数据值）。一个显示如何非常容易地创建生成器的示例如下:

```
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
```

可以用生成器来完成的操作同样可以用前一节所描述的基于类的迭代器来完成。但生成器的写法更为紧凑，因为它会自动创建 `__iter__()` 和 `__next__()` 方法。

另一个关键特性在于局部变量和执行状态会在每次调用之间自动保存。这使得该函数相比使用 `self.index` 和 `self.data` 这种实例变量的方式更易编写且更为清晰。

除了会自动创建方法和保存程序状态，当生成器终结时，它们还会自动引发 `StopIteration`。这些特性结合在一起，使得创建迭代器能与编写常规函数一样容易。

9.10 生成器表达式

某些简单的生成器可以写成简洁的表达式代码，所用语法类似列表推导式，但外层为圆括号而非方括号。这种表达式被设计用于生成器将立即被外层函数所使用的情况。生成器表达式相比完整的生成器更紧凑但较不灵活，相比等效的列表推导式则更为节省内存。

例如:

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

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

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

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

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

🄵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:\\Python39'
>>> 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有實作差🄵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 (File Wildcards)

glob 模組提供了一函式可以從目🄵萬用字元中搜尋🄵🄵生檔案列表：

```
>>> 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 list（串列）形式存放在 sys 模組的 *argv* 屬性中。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']
```

argparse 模組提供了一種更🄵雜的機制來處理命令列引數。以下🄵本可🄵取一個或多個檔案名稱，🄵可選擇要顯示的行數：

```
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)
```

當 `python top.py --lines=5 alpha.txt beta.txt` 在命令列執行時，該🄵本會將 `args.lines` 設🄵 5，🄵將 `args.filenames` 設🄵 `['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* 等屬性。即使當 *stdout* 被重新導向時，後者 *stderr* 可輸出警告和錯誤訊息：

```
>>> 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 method 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
... 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` 模組提供許多 class 可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的模組。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 提供了評估效能差🄵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🄵時間使用上關鍵的區塊 (time critical section) 的工具。

10.11 品質控管

達到高品質軟體的一個方法，是在開發時對每個函式寫測試，以及在開發過程中要不斷地跑這些測試。

doctest 模組提供了一個工具，掃描模組🄵根據程式中🄵嵌的文件字串執行測試。撰寫測試如同簡單地將它的呼叫及輸出結果剪下🄵貼上到文件字串中。透過提供範例給使用者，它🄵化了🄵明文件，🄵允許 doctest 模組確認程式碼的結果與🄵明文件一致：

```
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，但是它讓你可以在另外一個檔案🄵撰寫更完整的測試集：

```
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 標準模組庫

”Batteries included” 是 Python 的設計哲學。這個理念可以透過使用它的大型套件，感受□雜與□大的功能，來得到印證。例如：

- 使用 `xmlrpc.client` 和 `xmlrpc.server` 模組使實作遠端程序呼叫變得更□容易。即使模組名稱□有 XML，使用者□不需要直接操作 XML 檔案或事先具備相關知識。
- 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 `smtplib` 和 `poplib` 這些實際用來發送與接收訊息的模組，email 套件擁有更完整的工具集，可用於建立與解析□雜訊息結構（包含附件檔案）以及實作編碼與標頭協定。
- `json` 套件對 JSON 資料交□格式的解析，提供□大的支援。`csv` 模組則提供直接讀寫 CSV（以逗號分隔值的檔案格式，通常資料庫和電子表格都有支援）。`xml.etree.ElementTree`、`xml.dom` 與 `xml.sax` 套件則支援 XML 的處理。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` 模組是 SQLite 資料庫函式庫的一層包裝，提供一個具持久性的資料庫，可以使用稍微非標准的 SQL 語法來對它進行更新與存取。
- 有數種支援國際化的模組，包括 `gettext`、`locale` 和 `codecs` 等套件。

CHAPTER 11

标准库简介——第二部分

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 格式化输出

reprlib 模块提供了一个定制化版本的 repr() 函数，用于缩略显示大型或深层嵌套的容器对象:

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

pprint 模块提供了更加复杂的打印控制，其输出的内置对象和用户自定义对象能够被解释器直接读取。当输出结果过长而需要折行时，“美化输出机制”会添加换行符和缩进，以更清楚地展示数据结构:

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

textwrap 模块能够格式化文本段落，以适应给定的屏幕宽度:

```
>>> 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))
```

(下页继续)

(繼續上一頁)

```
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

locale 模块处理与特定地域文化相关的数据格式。locale 模块的 format 函数包含一个 grouping 属性，可直接将数字格式化为带有组分隔符的样式:

```
>>> 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 模板

string 模块包含一个通用的 Template 类，具有适用于最终用户的简化语法。它允许用户在不更改应用逻辑的情况下定制自己的应用。

上述格式化操作是通过占位符实现的，占位符由 $ 加上合法的 Python 标识符（只能包含字母、数字和下划线）构成。一旦使用花括号将占位符括起来，就可以在后面直接跟上更多的字母和数字而无需空格分割。$$ 将被转义成单个字符 $:

```
>>> 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.'
```

如果在字典或关键字参数中未提供某个占位符的值，那么 substitute() 方法将抛出 KeyError。对于邮件合并类型的应用，用户提供的数据有可能是不完整的，此时使用 safe_substitute() 方法更加合适——如果数据缺失，它会直接将占位符原样保留。

```
>>> 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 的子类可以自定义分隔符。例如，以下是某个照片浏览器的批量重命名功能，采用了百分号作为日期、照片序号和照片格式的占位符:

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
```

(下页继续)

(繼續上一頁)

```
>>> fmt = input('Enter rename style (%d-date %n-seqnum %f-format):  ')
Enter rename style (%d-date %n-seqnum %f-format):  Ashley_%n%f

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

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

模板的另一个应用是将程序逻辑与多样的格式化输出细节分离开来。这使得对 XML 文件、纯文本报表和 HTML 网络报表使用自定义模板成为可能。

11.3 使用二进制数据记录格式

`struct` 模块提供了 `pack()` 和 `unpack()` 函数，用于处理不定长度的二进制记录格式。下面的例子展示了在不使用 `zipfile` 模块的情况下，如何循环遍历一个 ZIP 文件的所有头信息。Pack 代码 `"H"` 和 `"I"` 分别代表两字节和四字节无符号整数。`"<"` 代表它们是标准尺寸的小端字节序:

```
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 多线程

线程是一种对于非顺序依赖的多个任务进行解耦的技术。多线程可以提高应用的响应效率，当接收用户输入的同时，保持其他任务在后台运行。一个有关的应用场景是，将 I/O 和计算运行在两个并行的线程中。

以下代码展示了高阶的 `threading` 模块如何在后台运行任务，且不影响主程序的继续运行:

```
import threading, zipfile

```

(下页继续)

(繼續上一頁)

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

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

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

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

多线程应用面临的主要挑战是，相互协调的多个线程之间需要共享数据或其他资源。为此，threading 模块提供了多个同步操作原语，包括线程锁、事件、条件变量和信号量。

尽管这些工具非常强大，但微小的设计错误却可以导致一些难以复现的问题。因此，实现多任务协作的首选方法是将所有对资源的请求集中到一个线程中，然后使用 queue 模块向该线程供应来自其他线程的请求。应用程序使用 Queue 对象进行线程间通信和协调，更易于设计，更易读，更可靠。

11.5 日志

logging 模块提供功能齐全且灵活的日志记录系统。在最简单的情况下，日志消息被发送到文件或 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')
```

这会产生以下输出:

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

默认情况下，informational 和 debugging 消息被压制，输出会发送到标准错误流。其他输出选项包括将消息转发到电子邮件，数据报，套接字或 HTTP 服务器。新的过滤器可以根据消息优先级选择不同的路由方式: DEBUG，INFO，WARNING，ERROR，和 CRITICAL。

日志系统可以直接从 Python 配置，也可以从用户配置文件加载，以便自定义日志记录而无需更改应用程序。

11.6 弱引用

Python 会自动进行内存管理（对大多数对象进行引用计数并使用*garbage collection* 来清除循环引用）。当某个对象的最后一个引用被移除后不久就会释放其所占用的内存。

此方式对大多数应用来说都适用，但偶尔也必须在对象持续被其他对象所使用时跟踪它们。不幸的是，跟踪它们将创建一个会令其永久化的引用。`weakref` 模块提供的工具可以不必创建引用就能跟踪对象。当对象不再需要时，它将自动从一个弱引用表中被移除，并为弱引用对象触发一个回调。典型应用包括对创建开销较大的对象进行缓存:

```
>>> 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:/python39/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7 用于操作列表的工具

许多对于数据结构的需求可以通过内置列表类型来满足。但是，有时也会需要具有不同效费比的替代实现。

`array` 模块提供了一种 `array()` 对象，它类似于列表，但只能存储类型一致的数据且存储密集更高。下面的例子演示了一个以两个字节为存储单元的无符号二进制数值的数组 (类型码为 `"H"`)，而对于普通列表来说，每个条目存储为标准 Python 的 int 对象通常要占用 16 个字节:

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

`collections` 模块提供了一种 `deque()` 对象，它类似于列表，但从左端添加和弹出的速度较快，而在中间查找的速度较慢。此种对象适用于实现队列和广度优先树搜索:

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

(下页继续)

(繼續上一頁)

```
>>> 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)
```

在替代的列表实现以外，标准库也提供了其他工具，例如 bisect 模块具有用于操作有序列表的函数:

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

heapq 模块提供了基于常规列表来实现堆的函数。最小值的条目总是保持在位置零。这对于需要重复访问最小元素而不希望运行完整列表排序的应用来说非常有用:

```
>>> 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 十进制浮点运算

decimal 模块提供了一种 Decimal 数据类型用于十进制浮点运算。相比内置的 float 二进制浮点实现，该类特别适用于

- 财务应用和其他需要精确十进制表示的用途，
- 控制精度，
- 控制四舍五入以满足法律或监管要求，
- 跟踪有效小数位，或
- 用户期望结果与手工完成的计算相匹配的应用程序。

例如，使用十进制浮点和二进制浮点数计算 70 美分手机和 5% 税的总费用，会产生的不同结果。如果结果四舍五入到最接近的分数差异会更大:

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

Decimal 表示的结果会保留尾部的零，并根据具有两个有效位的被乘数自动推出四个有效位。Decimal 可以模拟手工运算来避免当二进制浮点数无法精确表示十进制数时会导致的问题。

精确表示特性使得 Decimal 类能够执行对于二进制浮点数来说不适用的模运算和相等性检测:

```
>>> 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
```

decimal 模块提供了运算所需要的足够精度:

```
>>> 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，以🄵本 (script) 執行 `venv` 模組🄵且給定資料夾路徑：

```
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。

🄵擬環境的常用資料夾位置是 .venv。這個名稱通常會使該資料夾在你的 shell 中保持隱藏，因此這樣命名既可以解釋資料夾存在的原因，也不會造成任何困擾。它還能防止與某些工具所支援的 .env 環境變數定義檔案發生衝突。

一旦你建立了一個🄵擬環境，你可以🄵動它。

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。你可以透過你的網頁🄵覽器🄵覽 Python Package Index。

pip 有好幾個子指令：”install”、”uninstall”、”freeze” 等等。（可以參考 installing-index 指南，來取得 pip 的完整🄵明文件。）

你可以透過指定套件名字來安裝最新版本的套件：

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

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

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

pip uninstall 後面接一個或是多個套件名稱可以從F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F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FF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，F且作F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) $ python -m pip install -r requirements.txt
Collecting novas==3.1.1.3 (from -r requirements.txt (line 1))
  ...
Collecting numpy==1.9.2 (from -r requirements.txt (line 2))
  ...
Collecting requests==2.7.0 (from -r requirements.txt (line 3))
  ...
Installing collected packages: novas, numpy, requests
  Running setup.py install for novas
Successfully installed novas-3.1.1.3 numpy-1.9.2 requests-2.7.0
```

pip 還有更多功能。可以參考 installing-index 指南，來取得完整的 pip F明文件。當你撰寫了一個套件F且

想要讓它在 Python Package Index 上可以取得的話，可以參考 distributing-index 指南。

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（但簡潔）的參考素材像是型[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：Python 套件索引 (Python Package Index)，之前也被[F]稱[F] Cheese Shop[1]，[F]總了使用者開發 Python 模組的索引，[F]提供模組能[F]被下載。一旦開始發[F]相關程式碼，你可以將開發的作品[F][F]到這[F][F]且讓其他人找到。

- https://code.activestate.com/recipes/langs/python/：Python Cookbook 是一個相當大的程式碼集，包含程式碼範例、較大的模組以及有用的[F]本。特[F]值得注意的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]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。

[1] 「Cheese Shop（起司店）」是 Monty Python 的一個短劇：一位顧客進入一家起司店，但無論他要哪種起司，店員都[F][F]有貨。

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都將自動轉發給另一個。每天會有數以百計的🄵容，詢問（和回答）問題、建議新功能與發🄵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🄵。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🄵問題的方法。

🄵解

CHAPTER 14

互動式輸入編輯和歷史記🄵替🄵

有些版本的 Python 直譯器支援當前輸入🄵容的編輯和歷史記🄵的替🄵 (history substitution)，類似在 Korn shell 和 GNU Bash shell 中的功能。這個功能是用 GNU Readline 函式庫來實作，它支援多種編輯的風格。這個函式庫有它自己的🄵明文件，在這🄵我們就不重🄵了。

14.1 Tab 鍵自動完成 (Tab Completion) 和歷史記🄵編輯 (History Editing)

在直譯器🄵動的時候，變數和模組名稱的自動完成功能會被自動🄵用，所以可以用 `Tab` 鍵來呼叫自動完成函式；它會查看 Python 的陳述式名稱、當前區域變數名稱和可用模組名稱。對於像是 `string.a` 的點分隔運算式 (dotted expression)，它會對最後一個 `'.'` 之前的運算式求值，然後根據求值結果物件的屬性，給予自動完成的建議。請注意，如果一個物件有 `__getattr__()` method（方法），同時又是該運算式的一部份，這樣可能會執行應用程式自定義的程式碼。預設設定也會把你的指令歷史記🄵儲存在你的使用者資料夾🄵，一個名🄵 `.python_history` 的檔案中。在下一次🄵動互動式直譯器時，這些歷史記🄵依然可以被使用。

14.2 互動式直譯器的替代方案

與早期版本的直譯器相比，上述功能的出現的確是一個巨大的進步；但還是有一些願望🄴有被實現：如果能在每次🄵行時給予適當的縮排建議（剖析器 (parser) 知道下一行是否需要縮排標記 (indent token)），那就更棒了。自動完成機制可能會使用直譯器的符號表。若有一個命令能🄵檢查（或甚至建議）括號、引號和其他符號的匹配，那也會很有用。

有一個功能增🄵的互動式直譯器替代方案，已經存在一段時間，稱🄵 IPython，它具有 Tab 鍵自動完成、物件探索和進階歷史記🄵管理等特色。它也可以完全客🄵化🄵被嵌入到其他應用程式中。另一個類似的增🄵型互動式環境，稱🄵 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：

```
0.0001100110011001100110011001100110011001100110011...
```

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□□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□：

```
>>> 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□：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□建的 `repr()` 函式會選擇上段□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□簡單地顯示□ `0.1`。

注意，這是二進位浮點數理所當然的特性，□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□，或者預設全部輸出）。

□求更優雅的輸出，您可能想要使用字串的格式化 (string formatting) □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□生下一個幻覺。舉例來□，因□ 0.1 不是真正的 1/10，把三個 0.1 的值相加，也不會□生精准的 0.3：

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

同時，因□ 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□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）。

正如那篇文章的结尾所言，“对此问题并无简单的答案。”但是也不必过于担心浮点数的问题！Python 浮点运算中的错误是从浮点运算硬件继承而来，而在大多数机器上每次浮点运算得到的 2**53 数码位都会被作为 1 个整体来处理。这对大多数任务来说都已足够，但你确实需要记住它并非十进制算术，且每次浮点运算都可能会导致新的舍入错误。

虽然病态的情况确实存在，但对于大多数正常的浮点运算使用来说，你只需简单地将最终显示的结果舍入为你期望的十进制数值即可得到你期望的结果。str() 通常已足够，对于更精度的控制可参看 formatstrings 中 str.format() 方法的格式描述符。

对于需要精确十进制表示的使用场景，请尝试使用 decimal 模块，该模块实现了适合会计应用和高精度应用的十进制运算。

另一种形式的精确运算由 fractions 模块提供支持，该模块实现了基于有理数的算术运算（因此可以精确表示像 1/3 这样的数值）。

如果你是浮点运算的重度用户，你应该看一下数值运算 Python 包 NumPy 以及由 SciPy 项目所提供的许多其它数学和统计运算包。参见 <https://scipy.org>。

Python 也提供了一些工具，可以在你真的 想要知道一个浮点数精确值的少数情况下提供帮助。例如 float.as_integer_ratio() 方法会将浮点数表示为一个分数：

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

由于这是一个精确的比值，它可以被用来无损地重建原始值：

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

float.hex() 方法会以十六进制（以 16 为基数）来表示浮点数，同样能给出保存在你的计算机中的精确值：

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

这种精确的十六进制表示法可被用来精确地重建浮点值：

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

由于这种表示法是精确的，它适用于跨越不同版本（平台无关）的 Python 移植数值，以及与支持相同格式的其他语言（例如 Java 和 C99）交换数据.

另一个有用的工具是 math.fsum() 函数，它有助于减少求和过程中的精度损失。它会在数值被添加到总计值的时候跟踪“丢失的位”。这可以很好地保持总计值的精确度，使得错误不会积累到能影响结果总数的程度：

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

15.1 表示性错误

本小节将详细解释"0.1" 的例子，并说明你可以怎样亲自对此类情况进行精确分析。假定前提是已基本熟悉二进制浮点表示法。

*表示性错误*是指某些（其实是大多数）十进制小数无法以二进制（以 2 为基数的计数制）精确表示这一事实造成的错误。这就是为什么 Python（或者 Perl、C、C++、Java、Fortran 以及许多其他语言）经常不会显示你所期待的精确十进制数值的主要原因。

为什么会这样？1/10 是无法用二进制小数精确表示的。目前（2000 年 11 月）几乎所有使用 IEEE-754 浮点运算标准的机器以及几乎所有系统平台都会将 Python 浮点数映射为 IEEE-754 "双精度类型"。754 双精度类型包含 53 位精度，因此在输入时，计算会尽量将 0.1 转换为以 *J*/2***N* 形式所能表示的最接近分数，其中 *J* 为恰好包含 53 个二进制位的整数。重新将

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

写为

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

并且由于 *J* 恰好有 53 位 (即 `>= 2**52` 但 `< 2**53`)，*N* 的最佳值为 56:

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

也就是说，56 是唯一的 *N* 值能令 *J* 恰好有 53 位。这样 *J* 的最佳可能值就是经过舍入的商:

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

由于余数超过 10 的一半，最佳近似值可通过四舍五入获得:

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

这样在 754 双精度下 1/10 的最佳近似值为:

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

分子和分母都除以二则结果小数为:

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

请注意由于我们做了向上舍入，这个结果实际上略大于 1/10；如果我们没有向上舍入，则商将会略小于 1/10。但无论如何它都不会是 *精确的* 1/10!

因此计算永远不会"看到" 1/10：它实际看到的就是上面所给出的小数，它所能达到的最佳 754 双精度近似值:

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

如果我们将该小数乘以 10**55，我们可以看到该值输出为 55 位的十进制数:

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

这意味着存储在计算机中的确切数值等于十进制数值 0.1000000000000000055511151231257827021181583404541015625。许多语言（包括较旧版本的 Python）都不会显示这个完整的十进制数值，而是将结果舍入为 17 位有效数字:

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

fractions 和 decimal 模块可令进行此类计算更加容易:

```
>>> 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□

16.1 互動模式

16.1.1 錯誤處理

當一個錯誤發生時，直譯器會印出一個錯誤訊息和堆□回溯。在互動模式下，它將返回主提示字元；當輸入來自檔案時，它在印出堆□回溯後以非零退出狀態退出。(由 except 子句在 try 陳述式中處理的例外在這種情□下不是錯誤。) 有些錯誤是無條件嚴重的，會導致非零退出；這適用於□部不一致和一些記憶體耗盡的情□。所有的錯誤訊息都被寫入標准錯誤輸出；被執行指令的正常輸出被寫入標准輸出。

向主提示字元或次提示字元輸入中斷字元（通常是 Control-C 或 Delete ）會取消輸入□返回到主提示字元。[1] 在指令執行過程中輸入一個中斷，會引發 KeyboardInterrupt 例外，但可以通過 try 陳述式來處理。

16.1.2 可執行的 Python □本

在類 BSD 的 Unix 系統上，Python □本可以直接執行，就像 shell □本一樣，通過放置以下這行：

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

（假設直譯器在用□的 PATH 上）在□本的開頭□給檔案一個可執行模式。#! 必須是檔案的前兩個字元。在某些平台上，第一行必須以 Unix 樣式的□行 (' \n') 結尾，而不是 Windows (' \r\n') □行。請注意，井號 '#' 用於在 Python 中開始□解。

可以使用 **chmod** 指令□□本賦予可執行模式或權限。

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

在 Windows 系統上，□有「可執行模式」的概念。Python 安裝程式會自動將 .py 檔案與 python.exe 聯□起來，這樣雙擊 Python 檔案就會作□□本運行。副檔名也可以是 .pyw，在這種情□下，通常會出現的控制台視窗會被隱藏。

[1] GNU Readline 套件的一個問題可能會阻止這一點。

16.1.3 互動式F動檔案

當你互動式地使用 Python 時，每次F動直譯器時執行一些標准指令是非常方便的。你可以通過設置一個名F `PYTHONSTARTUP` 的環境變數來實現，該變數是一個包含F動指令的檔案名。它的功能類似 Unix shell 的 `.profile` 。

這個檔案只在互動模式中被讀取，當 Python 從F本中讀取指令時，此檔案不會被讀取，當:file: */dev/tty* 作F指令的明確來源時也不會（否則表現得像互動模式）。它在執行互動式指令的同一命名空間中執行，因此它所定義或 import 的物件可以在互動模式中不加限定地使用。你也可以在這個檔案中改變 `sys.ps1` 和 `sys.ps2` 等提示字元。

如果你想從當前目F中讀取一個額外的F動檔案，你可以在全域F動檔案中使用類似 `if os.path.isfile('.pythonrc.py'): exec(open('.pythonrc.py ').read()` 的程式碼設定這個行F。如果你想在一個F本中使用F動檔案，你必須在F本中明確地這樣做:

```
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 客F化模組

Python 提供了兩個F子（hook）讓你可以將它客F化: `sitecustomize` 和 `usercustomize` 。要看它是如何運作的，你首先需要找到你的 site-packages 的位置。F動 Python F運行這段程式碼:

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

現在，您可以在該目F中創建一個名F `usercustomize.py` 的檔案，F將您想要的任何F容放入其中。它會影響 Python 的每次呼叫，除非它以 `-s` 選項F動以禁用自動 import 。

`sitecustomize` 的運作方式相同，但通常是由電腦的管理員在全域 site-packages 目F下創建，F在 `usercustomize` 之前 import 。更多細節請參F `site` 模組的文件。

F解

APPENDIX A

术语对照表

>>> 交互式终端中默认的 Python 提示符。往往会显示于能以交互方式在解释器里执行的样例代码之前。

... 具有以下含义：

- 交互式终端中输入特殊代码行时默认的 Python 提示符，包括：缩进的代码块，成对的分隔符之内（圆括号、方括号、花括号或三重引号），或是指定一个装饰器之后。
- `Ellipsis` 内置常量。

2to3 把 Python 2.x 代码转换为 Python 3.x 代码的工具，通过解析源码，遍历解析树，处理绝大多数检测到的不兼容问题。

2to3 包含在标准库中，模块名为 `lib2to3`；提供了独立入口点 `Tools/scripts/2to3`。详见 2to3-reference。

abstract base class -- 抽象基类 抽象基类简称 ABC，是对*duck-typing* 的补充，它提供了一种定义接口的新方式，相比之下其他技巧例如 `hasattr()` 显得过于笨拙或有微妙错误（例如使用 魔术方法）。ABC 引入了虚拟子类，这种类并非继承自其他类，但却仍能被 `isinstance()` 和 `issubclass()` 所认可；详见 `abc` 模块文档。Python 自带许多内置的 ABC 用于实现数据结构（在 `collections.abc` 模块中）、数字（在 `numbers` 模块中）、流（在 `io` 模块中）、导入查找器和加载器（在 `importlib.abc` 模块中）。你可以使用 `abc` 模块来创建自己的 ABC。

annotation -- 注解 关联到某个变量、类属性、函数形参或返回值的标签，被约定作为*type hint* 来使用。

局部变量的标注在运行时不可访问，但全局变量、类属性和函数的标注会分别存放模块、类和函数的 `__annotations__` 特殊属性中。

参见*variable annotation*、*function annotation*、**PEP 484** 和 **PEP 526**，对此功能均有介绍。

argument -- 参数 在调用函数时传给*function* （或*method* ）的值。参数分为两种：

- *关键字参数*: 在函数调用中前面带有标识符（例如 `name=`）或者作为包含在前面带有 `**` 的字典里的值传入。举例来说，`3` 和 `5` 在以下对 `complex()` 的调用中均属于关键字参数:

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

- *位置参数*: 不属于关键字参数的参数。位置参数可出现于参数列表的开头以及/或者作为前面带有 * 的*iterable* 里的元素被传入。举例来说，3 和 5 在以下调用中均属于位置参数:

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

参数会被赋值给函数体中对应的局部变量。有关赋值规则参见 calls 一节。根据语法，任何表达式都可用来表示一个参数；最终算出的值会被赋给对应的局部变量。

另参见*parameter* 术语表条目，常见问题中 参数与形参的区别以及 **PEP 362**。

asynchronous context manager -- 异步上下文管理器 此种对象通过定义 `__aenter__()` 和 `__aexit__()` 方法来对 `async with` 语句中的环境进行控制。由 **PEP 492** 引入。

asynchronous generator -- 异步生成器 返回值为*asynchronous generator iterator* 的函数。它与使用 `async def` 定义的协程函数很相似，不同之处在于它包含 `yield` 表达式以产生一系列可在 `async for` 循环中使用的值。

此术语通常是指异步生成器函数，但在某些情况下则可能是指 异步生成器迭代器。如果需要清楚表达具体含义，请使用全称以避免歧义。

一个异步生成器函数可能包含 `await` 表达式或者 `async for` 以及 `async with` 语句。

asynchronous generator iterator -- 异步生成器迭代器 *asynchronous generator* 函数所创建的对象。

此对象属于*asynchronous iterator*，当使用 `__anext__()` 方法调用时会返回一个可等待对象来执行异步生成器函数的代码直到下一个 `yield` 表达式。

每个 `yield` 会临时暂停处理，记住当前位置执行状态 (包括局部变量和挂起的 try 语句)。当该 异步生成器迭代器与其他 `__anext__()` 返回的可等待对象有效恢复时，它会从离开位置继续执行。参见 **PEP 492** 和 **PEP 525**。

asynchronous iterable -- 异步可迭代对象 可在 `async for` 语句中被使用的对象。必须通过它的 `__aiter__()` 方法返回一个*asynchronous iterator*。由 **PEP 492** 引入。

asynchronous iterator -- 异步迭代器 实现了 `__aiter__()` 和 `__anext__()` 方法的对象。`__anext__` 必须返回一个*awaitable* 对象。`async for` 会处理异步迭代器的 `__anext__()` 方法所返回的可等待对象，直到其引发一个 `StopAsyncIteration` 异常。由 **PEP 492** 引入。

attribute -- 属性 关联到一个对象的值，可以使用点号表达式通过其名称来引用。例如，如果一个对象 *o* 具有一个属性 *a*，就可以用 *o.a* 来引用它。

awaitable -- 可等待对象 能在 `await` 表达式中使用的对象。可以是*coroutine* 或是具有 `__await__()` 方法的对象。参见 **PEP 492**。

BDFL “终身仁慈独裁者”的英文缩写，即 Guido van Rossum，Python 的创造者。

binary file -- 二进制文件 *file object* 能够读写字节类对象。二进制文件的例子包括以二进制模式（`'rb'`, `'wb'` or `'rb+'`）打开的文件、`sys.stdin.buffer`、`sys.stdout.buffer` 以及 `io.BytesIO` 和 `gzip.GzipFile` 的实例。

另请参见*text file* 了解能够读写 `str` 对象的文件对象。

bytes-like object -- 字节类对象 支持 bufferobjects 并且能导出 C-*contiguous* 缓冲的对象。这包括所有 `bytes`、`bytearray` 和 `array.array` 对象，以及许多普通 `memoryview` 对象。字节类对象可在多种二进制数据操作中使用；这些操作包括压缩、保存为二进制文件以及通过套接字发送等。

某些操作需要可变的二进制数据。这种对象在文档中常被称为 “可读写字节类对象”。可变缓冲对象的例子包括 `bytearray` 以及 `bytearray` 的 `memoryview`。其他操作要求二进制数据存放于不可变对象 (” 只读字节类对象”)；这种对象的例子包括 `bytes` 以及 `bytes` 对象的 `memoryview`。

bytecode -- 字节码 Python 源代码会被编译为字节码，即 CPython 解释器中表示 Python 程序的内部代码。字节码还会缓存在 `.pyc` 文件中，这样第二次执行同一文件时速度更快（可以免去将源码重新编译为字

节码）。这种" 中间语言" 运行在根据字节码执行相应机器码的*virtual machine* 之上。请注意不同 Python 虚拟机上的字节码不一定通用，也不一定能在不同 Python 版本上兼容。

字节码指令列表可以在 dis 模块的文档中查看。

callback -- 回调 一个作为参数被传入以用以在未来的某个时刻被调用的子例程函数。

class -- 类 用来创建用户定义对象的模板。类定义通常包含对该类的实例进行操作的方法定义。

class variable -- 类变量 在类中定义的变量，并且仅限在类的层级上修改 (而不是在类的实例中修改)。

coercion -- 强制类型转换 在包含两个相同类型参数的操作中，一种类型的实例隐式地转换为另一种类型。例如，`int(3.15)` 是将原浮点数转换为整型数 `3`，但在 `3+4.5` 中，参数的类型不一致（一个是 int, 一个是 float），两者必须转换为相同类型才能相加，否则将引发 `TypeError`。如果没有强制类型转换机制，程序员必须将所有可兼容参数归一化为相同类型，例如要写成 `float(3)+4.5` 而不是 `3+4.5`。

complex number -- 复数 对普通实数系统的扩展，其中所有数字都被表示为一个实部和一个虚部的和。虚数是虚数单位（`-1` 的平方根）的实倍数，通常在数学中写为 `i`，在工程学中写为 `j`。Python 内置了对复数的支持，采用工程学标记方式；虚部带有一个 `j` 后缀，例如 `3+1j`。如果需要 `math` 模块内对象的对应复数版本，请使用 `cmath`，复数的使用是一个比较高级的数学特性。如果你感觉没有必要，忽略它们也几乎不会有任何问题。

context manager -- 上下文管理器 在 `with` 语句中使用，通过定义 `__enter__()` 和 `__exit__()` 方法来控制环境状态的对象。参见 **PEP 343**。

context variable -- 上下文变量 一种根据其所属的上下文可以具有不同的值的变量。这类似于在线程局部存储中每个执行线程可以具有不同的变量值。不过，对于上下文变量来说，一个执行线程中可能会有多个上下文，而上下文变量的主要用途是对并发异步任务中变量进行追踪。参见 `contextvars`。

contiguous -- 连续 一个缓冲如果是 *C 连续*或 *Fortran 连续*就会被认为是连续的。零维缓冲是 C 和 Fortran 连续的。在一维数组中，所有条目必须在内存中彼此相邻地排列，采用从零开始的递增索引顺序。在多维 C-连续数组中，当按内存地址排列时用最后一个索引访问条目时速度最快。但是在 Fortran 连续数组中则是用第一个索引最快。

coroutine -- 协程 协程是子例程的更一般形式。子例程可以在某一点进入并在另一点退出。协程则可以在许多不同的点上进入、退出和恢复。它们可通过 `async def` 语句来实现。参见 **PEP 492**。

coroutine function -- 协程函数 返回一个*coroutine* 对象的函数。协程函数可通过 `async def` 语句来定义，并可能包含 `await`、`async for` 和 `async with` 关键字。这些特性是由 **PEP 492** 引入的。

CPython Python 编程语言的规范实现，在 python.org 上发布。"CPython" 一词用于在必要时将此实现与其他实现例如 Jython 或 IronPython 相区别。

decorator -- 装饰器 返回值为另一个函数的函数，通常使用 `@wrapper` 语法形式来进行函数变换。装饰器的常见例子包括 `classmethod()` 和 `staticmethod()`。

装饰器语法只是一种语法糖，以下两个函数定义在语义上完全等价:

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

@staticmethod
def f(...):
    ...
```

同的样概念也适用于类，但通常较少这样使用。有关装饰器的详情可参见 函数定义和 类定义的文档。

descriptor -- 描述器 任何定义了 `__get__()`, `__set__()` 或 `__delete__()` 方法的对象。当一个类属性为描述器时，它的特殊绑定行为就会在属性查找时被触发。通常情况下，使用 *a.b* 来获取、设置或删除一个属性时会在 *a* 的类字典中查找名称为 *b* 的对象，但如果 *b* 是一个描述器，则会调用对应的描述器

方法。理解描述器的概念是更深层次理解 Python 的关键，因为这是许多重要特性的基础，包括函数、方法、属性、类方法、静态方法以及对超类的引用等等。

有关描述器的方法的更多信息，请参阅 descriptors 或 描述器使用指南。

dictionary -- 字典 一个关联数组，其中的任意键都映射到相应的值。键可以是任何具有 `__hash__()` 和 `__eq__()` 方法的对象。在 Perl 语言中称为 hash。

dictionary comprehension -- 字典推导式 处理一个可迭代对象中的所有或部分元素并返回结果字典的一种紧凑写法。`results = {n: n ** 2 for n in range(10)}` 将生成一个由键 `n` 到值 `n ** 2` 的映射构成的字典。参见 comprehensions。

dictionary view -- 字典视图 从 `dict.keys()`, `dict.values()` 和 `dict.items()` 返回的对象被称为字典视图。它们提供了字典条目的一个动态视图，这意味着当字典改变时，视图也会相应改变。要将字典视图强制转换为真正的列表，可使用 `list(dictview)`。参见 dict-views。

docstring -- 文档字符串 作为类、函数或模块之内的第一个表达式出现的字符串字面值。它在代码执行时会被忽略，但会被解释器识别并放入所在类、函数或模块的 `__doc__` 属性中。由于它可用于代码内省，因此是对象存放文档的规范位置。

duck-typing -- 鸭子类型 指一种编程风格，它并不依靠查找对象类型来确定其是否具有正确的接口，而是直接调用或使用其方法或属性（“看起来像鸭子，叫起来也像鸭子，那么肯定就是鸭子。”）由于强调接口而非特定类型，设计良好的代码可通过允许多态替代来提升灵活性。鸭子类型避免使用 `type()` 或 `isinstance()` 检测。(但要注意鸭子类型可以使用抽象基类 作为补充。) 而往往会采用 `hasattr()` 检测或是*EAFP* 编程。

EAFP “求原谅比求许可更容易”的英文缩写。这种 Python 常用代码编写风格会假定所需的键或属性存在，并在假定错误时捕获异常。这种简洁快速风格的特点就是大量运用 `try` 和 `except` 语句。于其相对的则是所谓*LBYL* 风格，常见于 C 等许多其他语言。

expression -- 表达式 可以求出某个值的语法单元。换句话说，一个表达式就是表达元素例如字面值、名称、属性访问、运算符或函数调用的汇总，它们最终都会返回一个值。与许多其他语言不同，并非所有语言构件都是表达式。还存在不能被用作表达式的*statement*，例如 `while`。赋值也是属于语句而非表达式。

extension module -- 扩展模块 以 C 或 C++ 编写的模块，使用 Python 的 C API 来与语言核心以及用户代码进行交互。

f-string -- f-字符串 带有 `'f'` 或 `'F'` 前缀的字符串字面值通常被称为“f-字符串”即 格式化字符串字面值的简写。参见 **PEP 498**。

file object -- 文件对象 对外提供面向文件 API 以使用下层资源的对象（带有 `read()` 或 `write()` 这样的方法）。根据其创建方式的不同，文件对象可以处理对真实磁盘文件，对其他类型存储，或是对通讯设备的访问（例如标准输入/输出、内存缓冲区、套接字、管道等等）。文件对象也被称为 *文件类对象* 或 *流*。

实际上共有三种类别的文件对象: 原始二进制文件, 缓冲二进制文件 以及文本文件。它们的接口定义均在 `io` 模块中。创建文件对象的规范方式是使用 `open()` 函数。

file-like object -- 文件类对象 *file object* 的同义词。

finder -- 查找器 一种会尝试查找被导入模块的*loader* 的对象。

从 Python 3.3 起存在两种类型的查找器: 元路径查找器 配合 `sys.meta_path` 使用，以及*path entry finders* 配合 `sys.path_hooks` 使用。

更多详情可参见 **PEP 302**, **PEP 420** 和 **PEP 451**。

floor division -- 向下取整除法 向下舍入到最接近的整数的数学除法。向下取整除法的运算符是 `//` 。例如，表达式 `11 // 4` 的计算结果是 `2` ，而与之相反的是浮点数的真正除法返回 `2.75` 。注意 `(-11) // 4` 会返回 `-3` 因为这是 `-2.75` 向下舍入得到的结果。见 **PEP 238** 。

function -- 函数 可以向调用者返回某个值的一组语句。还可以向其传入零个或多个参数 并在函数体执行中被使用。另见*parameter*, *method* 和 function 等节。

function annotation -- 函数标注 即针对函数形参或返回值的*annotation* 。

函数标注通常用于类型提示：例如以下函数预期接受两个 int 参数并预期返回一个 int 值:

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

函数标注语法的详解见 function 一节。

请参看*variable annotation* 和 **PEP 484** 对此功能的描述。

__future__ future 语句, `from __future__ import <feature>` 指示编译器使用将在未来的 Python 发布版中成为标准的语法和语义来编译当前模块。__future__ 模块文档记录了可能的 *feature* 取值。通过导入此模块并对其变量求值，你可以看到每项新特性在何时被首次加入到该语言中以及它将（或已）在何时成为默认:

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

garbage collection -- 垃圾回收 释放不再被使用的内存空间的过程。Python 是通过引用计数和一个能够检测和打破循环引用的循环垃圾回收器来执行垃圾回收的。可以使用 gc 模块来控制垃圾回收器。

generator -- 生成器 返回一个*generator iterator* 的函数。它看起来很像普通函数，不同点在于其包含 yield 表达式以便产生一系列值供给 for-循环使用或是通过 next() 函数逐一获取。

通常是指生成器函数，但在某些情况下也可能是指 生成器迭代器。如果需要清楚表达具体含义，请使用全称以避免歧义。

generator iterator -- 生成器迭代器 *generator* 函数所创建的对象。

每个 yield 会临时暂停处理，记住当前位置执行状态（包括局部变量和挂起的 try 语句）。当该 生成器迭代器恢复时，它会从离开位置继续执行（这与每次调用都从新开始的普通函数差别很大）。

generator expression -- 生成器表达式 返回一个迭代器的表达式。它看起来很像普通表达式后面带有定义了一个循环变量、范围的 for 子句，以及一个可选的 if 子句。以下复合表达式会为外层函数生成一系列值:

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

generic function -- 泛型函数 为不同的类型实现相同操作的多个函数所组成的函数。在调用时会由调度算法来确定应该使用哪个实现。

另请参见*single dispatch* 术语表条目、functools.singledispatch() 装饰器以及 **PEP 443**。

generic type -- 泛型类型 可以被形参化的*type*；通常为容器类型例如 list。可用于类型提示 和标注。

请参阅 **PEP 483** 来了解详情，以及 typing 或 泛型别名类型来了解其用法。

GIL 参见*global interpreter lock*。

global interpreter lock -- 全局解释器锁 *CPython* 解释器所采用的一种机制，它确保同一时刻只有一个线程在执行 Python *bytecode*。此机制通过设置对象模型（包括 dict 等重要内置类型）针对并发访问的隐式安全简化了 CPython 实现。给整个解释器加锁使得解释器多线程运行更方便，其代价则是牺牲了在多处理器上的并行性。

不过，某些标准库或第三方库的扩展模块被设计为在执行计算密集型任务如压缩或哈希时释放 GIL。此外，在执行 I/O 操作时也总是会释放 GIL。

创建一个（以更精细粒度来锁定共享数据的）“自由线程”解释器的努力从未获得成功，因为这会牺牲在普通单处理器情况下的性能。据信克服这种性能问题的措施将导致实现变得更复杂，从而更难以维护。

hash-based pyc -- 基于哈希的 pyc 使用对应源文件的哈希值而非最后修改时间来确定其有效性的字节码缓存文件。参见 pyc-invalidation。

hashable -- 可哈希 一个对象的哈希值如果在其生命周期内绝不改变，就被称为 *可哈希*（它需要具有 `__hash__()` 方法），并可以同其他对象进行比较（它需要具有 `__eq__()` 方法）。可哈希对象必须具有相同的哈希值比较结果才会相同。

可哈希性使得对象能够作为字典键或集合成员使用，因为这些数据结构要在内部使用哈希值。

大多数 Python 中的不可变内置对象都是可哈希的；可变容器（例如列表或字典）都不可哈希；不可变容器（例如元组和 frozenset）仅当它们的元素均为可哈希时才是可哈希的。用户定义类的实例对象默认是可哈希的。它们在比较时一定不相同（除非是与自己比较），它们的哈希值的生成是基于它们的 `id()`。

IDLE Python 的 IDE，“集成开发与学习环境”的英文缩写。是 Python 标准发行版附带的基本编辑器和解释器环境。

immutable -- 不可变 具有固定值的对象。不可变对象包括数字、字符串和元组。这样的对象不能被改变。如果必须存储一个不同的值，则必须创建新的对象。它们在需要常量哈希值的地方起着重要作用，例如作为字典中的键。

import path -- 导入路径 由多个位置（或路径条目）组成的列表，会被模块的*path based finder* 用来查找导入目标。在导入时，此位置列表通常来自 `sys.path`，但对次级包来说也可能来自上级包的 `__path__` 属性。

importing -- 导入 令一个模块中的 Python 代码能为另一个模块中的 Python 代码所使用的过程。

importer -- 导入器 查找并加载模块的对象；此对象既属于*finder* 又属于*loader*。

interactive -- 交互 Python 带有一个交互式解释器，即你可以在解释器提示符后输入语句和表达式，立即执行并查看其结果。只需不带参数地启动 `python` 命令（也可以在你的计算机开始菜单中选择相应菜单项）。在测试新想法或检验模块和包的时候用这种方式会非常方便（请记得使用 `help(x)`）。

interpreted -- 解释型 Python 一是种解释型语言，与之相对的是编译型语言，虽然两者的区别由于字节码编译器的存在而会有所模糊。这意味着源文件可以直接运行而不必显式地创建可执行文件再运行。解释型语言通常具有比编译型语言更短的开发/调试周期，但是其程序往往运行得更慢。参见*interactive*。

interpreter shutdown -- 解释器关闭 当被要求关闭时，Python 解释器将进入一个特殊运行阶段并逐步释放所有已分配资源，例如模块和各种关键内部结构等。它还会多次调用垃圾回收器。这会触发用户定义析构器或弱引用回调中的代码执行。在关闭阶段执行的代码可能会遇到各种异常，因为其所依赖的资源已不再有效（常见的例子有库模块或警告机制等）。

解释器需要关闭的主要原因有 `__main__` 模块或所运行的脚本已完成执行。

iterable -- 可迭代对象 能够逐一返回其成员项的对象。可迭代对象的例子包括所有序列类型 (例如 `list`, `str` 和 `tuple`) 以及某些非序列类型例如 `dict`, 文件对象 以及定义了 `__iter__()` 方法或是实现了序列 语义的 `__getitem__()` 方法的任意自定义类对象。

可迭代对象被可用于 `for` 循环以及许多其他需要一个序列的地方（`zip()`、`map()` ...）。当一个可迭代对象作为参数传给内置函数 `iter()` 时，它会返回该对象的迭代器。这种迭代器适用于对值集合的一次性遍历。在使用可迭代对象时，你通常不需要调用 `iter()` 或者自己处理迭代器对象。`for` 语句会为你自动处理那些操作，创建一个临时的未命名变量用来在循环期间保存迭代器。参见*iterator*、*sequence* 以及*generator*。

iterator -- 迭代器 用来表示一连串数据流的对象。重复调用迭代器的 `__next__()` 方法（或将其传给内置函数 `next()`）将逐个返回流中的项。当没有数据可用时则将引发 `StopIteration` 异常。到这时迭代器对象中的数据项已耗尽，继续调用其 `__next__()` 方法只会再次引发 `StopIteration` 异常。迭代器必须具有 `__iter__()` 方法用来返回该迭代器对象自身，因此迭代器必定也是可迭代对象，可被用于其他可迭代对象适用的大部分场合。一个显著的例外是那些会多次重复访问迭代项的代码。容器对象（例如 `list`）在你每次向其传入 `iter()` 函数或是在 `for` 循环中使用它时都会产生一个新的迭

代器。如果在此情况下你尝试用迭代器则会返回在之前迭代过程中被耗尽的同一迭代器对象，使其看起来就像是一个空容器。

更多信息可查看 typeiter。

key function -- 键函数 键函数或称整理函数，是能够返回用于排序或排位的值的可调用对象。例如，`locale.strxfrm()` 可用于生成一个符合特定区域排序约定的排序键。

Python 中有许多工具都允许用键函数来控制元素的排位或分组方式。其中包括 `min()`，`max()`，`sorted()`，`list.sort()`，`heapq.merge()`，`heapq.nsmallest()`，`heapq.nlargest()` 以及 `itertools.groupby()`。

要创建一个键函数有多种方式。例如，`str.lower()` 方法可以用作忽略大小写排序的键函数。另外，键函数也可通过 `lambda` 表达式来创建，例如 `lambda r: (r[0], r[2])`。还有 `operator` 模块提供了三个键函数构造器：`attrgetter()`、`itemgetter()` 和 `methodcaller()`。请查看 如何排序 一节以获取创建和使用键函数的示例。

keyword argument -- 关键字参数 参见*argument*。

lambda 由一个单独*expression* 构成的匿名内联函数，表达式会在调用时被求值。创建 lambda 函数的句法为 `lambda [parameters]: expression`

LBYL “先查看后跳跃”的英文缩写。这种代码编写风格会在进行调用或查找之前显式地检查前提条件。此风格与*EAFP* 方式恰成对比，其特点是大量使用 `if` 语句。

在多线程环境中，LBYL 方式会导致“查看”和“跳跃”之间发生条件竞争风险。例如，以下代码 `if key in mapping: return mapping[key]` 可能由于在检查操作之后其他线程从 *mapping* 中移除了 *key* 而出错。这种问题可通过加锁或使用 EAFP 方式来解决。

list -- 列表 Python 内置的一种*sequence*。虽然名为列表，但更类似于其他语言中的数组而非链接列表，因为访问元素的时间复杂度为 O(1)。

list comprehension -- 列表推导式 处理一个序列中的所有或部分元素并返回结果列表的一种紧凑写法。`result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` 将生成一个 0 到 255 范围内的十六进制偶数对应字符串（0x..）的列表。其中 `if` 子句是可选的，如果省略则 `range(256)` 中的所有元素都会被处理。

loader -- 加载器 负责加载模块的对象。它必须定义名为 `load_module()` 的方法。加载器通常由一个*finder* 返回。详情参见 **PEP 302**，对于*abstract base class* 可参见 `importlib.abc.Loader`。

magic method -- 魔术方法 *special method* 的非正式同义词。

mapping -- 映射 一种支持任意键查找并实现了 `Mapping` 或 `MutableMapping` 抽象基类中所规定方法的容器对象。此类对象的例子包括 `dict`，`collections.defaultdict`，`collections.OrderedDict` 以及 `collections.Counter`。

meta path finder -- 元路径查找器 `sys.meta_path` 的搜索所返回的*finder*。元路径查找器与*path entry finders* 存在关联但并不相同。

请查看 `importlib.abc.MetaPathFinder` 了解元路径查找器所实现的方法。

metaclass -- 元类 一种用于创建类的类。类定义包含类名、类字典和基类列表。元类负责接受上述三个参数并创建相应的类。大部分面向对象的编程语言都会提供一个默认实现。Python 的特别之处在于可以创建自定义元类。大部分用户永远不需要这个工具，但当需要出现时，元类可提供强大而优雅的解决方案。它们已被用于记录属性访问日志、添加线程安全性、跟踪对象创建、实现单例，以及其他许多任务。

更多详情参见 metaclasses。

method -- 方法 在类内部定义的函数。如果作为该类的实例的一个属性来调用，方法将会获取实例对象作为其第一个*argument* (通常命名为 `self`)。参见*function* 和*nested scope*。

method resolution order -- 方法解析顺序 方法解析顺序就是在查找成员时搜索全部基类所用的先后顺序。请查看 Python 2.3 方法解析顺序 了解自 2.3 版起 Python 解析器所用相关算法的详情。

module -- 模块 此对象是 Python 代码的一种组织单位。各模块具有独立的命名空间，可包含任意 Python 对象。模块可通过*importing* 操作被加载到 Python 中。

另见*package*。

module spec -- 模块规格 一个命名空间，其中包含用于加载模块的相关导入信息。是 `importlib.machinery.ModuleSpec` 的实例。

MRO 参见*method resolution order*。

mutable -- 可变 可变对象可以在其 `id()` 保持固定的情况下改变其取值。另请参见*immutable*。

named tuple -- 具名元组 术语“具名元组”可用于任何继承自元组，并且其中的可索引元素还能使用名称属性来访问的类型或类。这样的类型或类还可能拥有其他特性。

有些内置类型属于具名元组，包括 `time.localtime()` 和 `os.stat()` 的返回值。另一个例子是 `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
```

有些具名元组是内置类型(例如上面的例子)。此外，具名元组还可通过常规类定义从 `tuple` 继承并定义名称字段的方式来创建。这样的类可以手工编写，或者使用工厂函数 `collections.namedtuple()` 创建。后一种方式还会添加一些手工编写或内置具名元组所没有的额外方法。

namespace -- 命名空间 命名空间是存放变量的场所。命名空间有局部、全局和内置的，还有对象中的嵌套命名空间（在方法之内）。命名空间通过防止命名冲突来支持模块化。例如，函数 `builtins.open` 与 `os.open()` 可通过各自的命名空间来区分。命名空间还通过明确哪个模块实现那个函数来帮助提高可读性和可维护性。例如，`random.seed()` 或 `itertools.islice()` 这种写法明确了这些函数是由 `random` 与 `itertools` 模块分别实现的。

namespace package -- 命名空间包 **PEP 420** 所引入的一种仅被用作子包的容器的*package*，命名空间包可以没有实体表示物，其描述方式与*regular package* 不同，因为它们没有 `__init__.py` 文件。

另可参见*module*。

nested scope -- 嵌套作用域 在一个定义范围内引用变量的能力。例如，在另一函数之内定义的函数可以引用前者的变量。请注意嵌套作用域默认只对引用有效而对赋值无效。局部变量的读写都受限于最内层作用域。类似的，全局变量的读写则作用于全局命名空间。通过 `nonlocal` 关键字可允许写入外层作用域。

new-style class -- 新式类 对于目前已被应于所有类对象的类形式的旧称谓。在早先的 Python 版本中，只有新式类能够使用 Python 新增的更灵活特性，例如 `__slots__`、描述符、特征属性、`__getattribute__()`、类方法和静态方法等。

object -- 对象 任何具有状态（属性或值）以及预定义行为（方法）的数据。object 也是任何*new-style class* 的最顶层基类名。

package -- 包 一种可包含子模块或递归地包含子包的 Python *module*。从技术上说，包是带有 `__path__` 属性的 Python 模块。

另参见*regular package* 和*namespace package*。

parameter -- 形参 *function*（或方法）定义中的命名实体，它指定函数可以接受的一个*argument*（或在某些情况下，多个实参）。有五种形参：

- *positional-or-keyword*：位置或关键字，指定一个可以作为位置参数 传入也可以作为关键字参数 传入的实参。这是默认的形参类型，例如下面的 *foo* 和 *bar*:

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

- *positional-only*：仅限位置，指定一个只能通过位置传入的参数。仅限位置形参可通过在函数定义的形参列表中它们之后包含一个 / 字符来定义，例如下面的 *posonly1* 和 *posonly2*:

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

- *keyword-only*：仅限关键字，指定一个只能通过关键字传入的参数。仅限关键字形参可通过在函数定义的形参列表中包含单个可变位置形参或者在多个可变位置形参之前放一个 * 来定义，例如下面的 *kw_only1* 和 *kw_only2*:

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

- *var-positional*：可变位置，指定可以提供由一个任意数量的位置参数构成的序列（附加在其他形参已接受的位置参数之后）。这种形参可通过在形参名称前加缀 * 来定义，例如下面的 *args*:

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

- *var-keyword*：可变关键字，指定可以提供任意数量的关键字参数（附加在其他形参已接受的关键字参数之后）。这种形参可通过在形参名称前加缀 ** 来定义，例如上面的 *kwargs*。

形参可以同时指定可选和必选参数，也可以为某些可选参数指定默认值。

另参见*argument* 术语表条目、参数与形参的区别中的常见问题、`inspect.Parameter` 类、function 一节以及 **PEP 362**。

path entry -- 路径入口 *import path* 中的一个单独位置，会被*path based finder* 用来查找要导入的模块。

path entry finder -- 路径入口查找器 任一可调用对象使用 `sys.path_hooks`（即*path entry hook*）返回的*finder*，此种对象能通过*path entry* 来定位模块。

请参看 `importlib.abc.PathEntryFinder` 以了解路径入口查找器所实现的各个方法。

path entry hook -- 路径入口钩子 一种可调用对象，在知道如何查找特定*path entry* 中的模块的情况下能够使用 `sys.path_hook` 列表返回一个*path entry finder*。

path based finder -- 基于路径的查找器 默认的一种元路径查找器，可在一个*import path* 中查找模块。

path-like object -- 路径类对象 代表一个文件系统路径的对象。类路径对象可以是一个表示路径的 `str` 或者 `bytes` 对象，还可以是一个实现了 `os.PathLike` 协议的对象。一个支持 `os.PathLike` 协议的对象可通过调用 `os.fspath()` 函数转换为 `str` 或者 `bytes` 类型的文件系统路径；`os.fsdecode()` 和 `os.fsencode()` 可被分别用来确保获得 `str` 或 `bytes` 类型的结果。此对象是由 **PEP 519** 引入的。

PEP “Python 增强提议”的英文缩写。一个 PEP 就是一份设计文档，用来向 Python 社区提供信息，或描述一个 Python 的新增特性及其进度或环境。PEP 应当提供精确的技术规格和所提议特性的原理说明。

PEP 应被作为提出主要新特性建议、收集社区对特定问题反馈以及为必须加入 Python 的设计决策编写文档的首选机制。PEP 的作者有责任在社区内部建立共识，并应将不同意见也记入文档。

参见 **PEP 1**。

portion -- 部分 构成一个命名空间包的单个目录内文件集合（也可能存放于一个 zip 文件内），具体定义见 **PEP 420**。

positional argument -- 位置参数 参见*argument*。

provisional API -- 暂定 API 暂定 API 是指被有意排除在标准库的向后兼容性保证之外的应用编程接口。虽然此类接口通常不会再有重大改变，但只要其被标记为暂定，就可能在核心开发者确定有必要的情况下进行向后不兼容的更改（甚至包括移除该接口）。此种更改并不会随意进行 -- 仅在 API 被加入之前未考虑到的严重基础性缺陷被发现时才可能会这样做。

即便是对暂定 API 来说，向后不兼容的更改也会被视为“最后的解决方案”——任何问题被确认时都会尽可能先尝试找到一种向后兼容的解决方案。

这种处理过程允许标准库持续不断地演进，不至于被有问题的长期性设计缺陷所困。详情见 **PEP 411**。

provisional package -- 暂定包 参见*provisional API*。

Python 3000 Python 3.x 发布路线的昵称（这个名字在版本 3 的发布还遥遥无期的时候就已出现了）。有时也被缩写为“Py3k”。

Pythonic 指一个思路或一段代码紧密遵循了 Python 语言最常用的风格和理念，而不是使用其他语言中通用的概念来实现代码。例如，Python 的常用风格是使用 `for` 语句循环来遍历一个可迭代对象中的所有元素。许多其他语言没有这样的结构，因此不熟悉 Python 的人有时会选择使用一个数字计数器:

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

而相应的更简洁更 Pythonic 的方法是这样的:

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

qualified name -- 限定名称 一个以点号分隔的名称，显示从模块的全局作用域到该模块中定义的某个类、函数或方法的“路径”，相关定义见 **PEP 3155**。对于最高层级的函数和类，限定名称与对象名称一致:

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

当被用于引用模块时，*完整限定名称*意为标示该模块的以点号分隔的整个路径，其中包含其所有的父包，例如 `email.mime.text`:

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

reference count -- 引用计数 对特定对象的引用的数量。当一个对象的引用计数降为零时，所分配资源将被释放。引用计数对 Python 代码来说通常是不可见的，但它是*CPython* 实现的一个关键元素。`sys` 模块定义了一个 `getrefcount()` 函数，程序员可调用它来返回特定对象的引用计数。

regular package -- 常规包 传统型的*package*，例如包含有一个 `__init__.py` 文件的目录。

另参见*namespace package*。

__slots__ 一种写在类内部的声明，通过预先声明实例属性等对象并移除实例字典来节省内存。虽然这种技巧很流行，但想要用好却并不容易，最好是只保留在少数情况下采用，例如极耗内存的应用程序，并且其中包含大量实例。

sequence -- 序列 一种*iterable*，它支持通过 `__getitem__()` 特殊方法来使用整数索引进行高效的元素访问，并定义了一个返回序列长度的 `__len__()` 方法。内置的序列类型有 `list`、`str`、`tuple` 和 `bytes`。注意虽然 `dict` 也支持 `__getitem__()` 和 `__len__()`，但它被认为属于映射而非序列，因为它查找时使用任意的*immutable* 键而非整数。

collections.abc.Sequence 抽象基类定义了一个更丰富的接口，它在 __getitem__() 和 __len__() 之外又添加了 count(), index(), __contains__() 和 __reversed__()。实现此扩展接口的类型可以使用 register() 来显式地注册。

set comprehension -- 集合推导式 处理一个可迭代对象中的所有或部分元素并返回结果集合的一种紧凑写法。results = {c for c in 'abracadabra' if c not in 'abc'} 将生成字符串集合 {'r', 'd'}。参见 comprehensions。

single dispatch -- 单分派 一种*generic function* 分派形式，其实现是基于单个参数的类型来选择的。

slice -- 切片 通常只包含了特定*sequence* 的一部分的对象。切片是通过使用下标标记来创建的，在 [] 中给出几个以冒号分隔的数字，例如 variable_name[1:3:5]。方括号（下标）标记在内部使用 slice 对象。

special method -- 特殊方法 一种由 Python 隐式调用的方法，用来对某个类型执行特定操作例如相加等等。这种方法的名称的首尾都为双下划线。特殊方法的文档参见 specialnames。

statement -- 语句 语句是程序段（一个代码“块”）的组成单位。一条语句可以是一个*expression* 或某个带有关键字的结构，例如 if、while 或 for。

text encoding -- 文本编码 用于将 Unicode 字符串编码为字节串的编码器。

text file -- 文本文件 一种能够读写 str 对象的*file object*。通常一个文本文件实际是访问一个面向字节的数据流并自动处理*text encoding*。文本文件的例子包括以文本模式（'r' 或 'w'）打开的文件、sys.stdin、sys.stdout 以及 io.StringIO 的实例。

另请参看*binary file* 了解能够读写字节类对象 的文件对象。

triple-quoted string -- 三引号字符串 首尾各带三个连续双引号（"）或者单引号（'）的字符串。它们在功能上与首尾各用一个引号标注的字符串没有什么不同，但是有多种用处。它们允许你在字符串内包含未经转义的单引号和双引号，并且可以跨越多行而无需使用连接符，在编写文档字符串时特别好用。

type -- 类型 类型决定一个 Python 对象属于什么种类；每个对象都具有一种类型。要知道对象的类型，可以访问它的 __class__ 属性，或是通过 type(obj) 来获取。

type alias -- 类型别名 一个类型的同义词，创建方式是把类型赋值给特定的标识符。

类型别名的作用是简化类型提示。例如:

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

可以这样提高可读性:

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

参见 typing 和 **PEP 484**，其中有对此功能的详细描述。

type hint -- 类型提示 *annotation* 为变量、类属性、函数的形参或返回值指定预期的类型。

类型提示属于可选项，Python 不要求提供，但其可对静态类型分析工具起作用，并可协助 IDE 实现代码补全与重构。

全局变量、类属性和函数的类型提示可以使用 typing.get_type_hints() 来访问，但局部变量则不可以。

参见 typing 和 **PEP 484**，其中有对此功能的详细描述。

universal newlines -- 通用换行 一种解读文本流的方式，将以下所有符号都识别为行结束标志：Unix 的行结束约定 `'\n'`、Windows 的约定 `'\r\n'` 以及旧版 Macintosh 的约定 `'\r'`。参见 **PEP 278** 和 **PEP 3116** 和 `bytes.splitlines()` 了解更多用法说明。

variable annotation -- 变量注解 对变量或类属性的*annotation*。

在标注变量或类属性时，还可选择为其赋值:

```
class C:
    field: 'annotation'
```

变量标注通常被用作类型提示：例如以下变量预期接受 `int` 类型的值:

```
count: int = 0
```

变量标注语法的详细解释见 annassign 一节。

请参看*function annotation*、**PEP 484** 和 **PEP 526**，其中对此功能有详细描述。

virtual environment -- 虚拟环境 一种采用协作式隔离的运行时环境，允许 Python 用户和应用程序在安装和升级 Python 分发包时不会干扰到同一系统上运行的其他 Python 应用程序的行为。

另参见 `venv`。

virtual machine -- 虚拟机 一台完全通过软件定义的计算机。Python 虚拟机可执行字节码编译器所生成的*bytecode*。

Zen of Python -- Python 之禅 列出 Python 设计的原则与哲学，有助于理解与使用这种语言。查看其具体内容可在交互模式提示符中输入”`import this`”。

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 该软件的历史

Python 由荷兰数学和计算机科学研究学会（CWI，见 https://www.cwi.nl/ ）的 Guido van Rossum 于 1990 年代初设计，作为一门叫做 ABC 的语言的替代品。尽管 Python 包含了许多来自其他人的贡献，Guido 仍是其主要作者。

1995 年，Guido 在弗吉尼亚州的国家创新研究公司（CNRI，见 https://www.cnri.reston.va.us/ ）继续他在 Python 上的工作，并在那里发布了该软件的多个版本。

2000 年五月，Guido 和 Python 核心开发团队转到 BeOpen.com 并组建了 BeOpen PythonLabs 团队。同年十月，PythonLabs 团队转到 Digital Creations (现为 Zope Corporation；见 https://www.zope.org/)。2001 年，Python 软件基金会 (PSF，见 https://www.python.org/psf/) 成立，这是一个专为拥有 Python 相关知识产权而创建的非营利组织。Zope Corporation 现在是 PSF 的赞助成员。

所有的 Python 版本都是开源的（有关开源的定义参阅 https://opensource.org/ ）。历史上，绝大多数 Python 版本是 GPL 兼容的；下表总结了各个版本情况。

发布版本	源自	年份	所有者	GPL 兼容？
0.9.0 至 1.2	n/a	1991-1995	CWI	是
1.3 至 1.5.2	1.2	1995-1999	CNRI	是
1.6	1.5.2	2000	CNRI	否
2.0	1.6	2000	BeOpen.com	否
1.6.1	1.6	2001	CNRI	否
2.1	2.0+1.6.1	2001	PSF	否
2.0.1	2.0+1.6.1	2001	PSF	是
2.1.1	2.1+2.0.1	2001	PSF	是
2.1.2	2.1.1	2002	PSF	是
2.1.3	2.1.2	2002	PSF	是
2.2 及更高	2.1.1	2001 至今	PSF	是

備🄴: GPL 兼容并不意味着 Python 在 GPL 下发布。与 GPL 不同，所有 Python 许可证都允许您分发修改后

的版本，而无需开源所做的更改。GPL 兼容的许可证使得 Python 可以与其它在 GPL 下发布的软件结合使用；但其它的许可证则不行。

感谢众多在 Guido 指导下工作的外部志愿者，使得这些发布成为可能。

C.2 获取或以其他方式使用 Python 的条款和条件

Python 软件和文档的使用许可均基于*PSF 许可协议*。

从 Python 3.8.6 开始，文档中的示例、操作指导和其他代码采用的是 PSF 许可协议和*零条款 BSD 许可* 的双重使用许可。

某些包含在 Python 中的软件是基于不同的许可。这些许可会与相应许可之下的代码一同列出。有关这些许可的不完整列表请参阅*被收录软件的许可证与鸣谢*。

C.2.1 用于 PYTHON 3.9.6 的 PSF 许可协议

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),␣
↪and
   the Individual or Organization ("Licensee") accessing and otherwise using␣
↪Python
   3.9.6 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 3.9.6 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice␣
↪of
   copyright, i.e., "Copyright © 2001-2021 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 3.9.6 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 3.9.6 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
   3.9.6.

4. PSF is making Python 3.9.6 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 3.9.6 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.9.6
```

```
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.9.6, 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 3.9.6, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 用于 PYTHON 2.0 的 BEOPEN.COM 许可协议

BEOPEN PYTHON 开源许可协议第 1 版

```
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
```

(下页继续)

(繼續上一頁)

```
   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 用于 PYTHON 1.6.1 的 CNRI 许可协议

```
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
```

(下页继续)

(繼續上一頁)

```
   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 用于 PYTHON 0.9.0 至 1.2 的 CWI 许可协议

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.9.6 DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
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 被收录软件的许可证与鸣谢

本节是 Python 发行版中收录的第三方软件的许可和致谢清单，该清单是不完整且不断增长的。

C.3.1 Mersenne Twister

_random 模块包含基于 http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html 下载的代码。以下是原始代码的完整注释（声明）：

```
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
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 套接字

socket 模块使用 getaddrinfo() 和 getnameinfo() 函数，这些函数源代码在 WIDE 项目（http://www.wide.ad.jp/）的单独源文件中。

```
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 异步套接字服务

asynchat 和 asyncore 模块包含以下声明:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.4 Cookie 管理

http.cookies 模块包含以下声明:

```
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 执行追踪

trace 模块包含以下声明:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 UUencode 与 UUdecode 函数

uu 模块包含以下声明:

```
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 远程过程调用

xmlrpc.client 模块包含以下声明:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
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
```

(下页继续)

(繼續上一頁)

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

`test_epoll` 模块包含以下声明:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Select kqueue

`select` 模块关于 kqueue 的接口包含以下声明:

```
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.

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
```

(下页继续)

(繼續上一頁)

```
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

Python/pyhash.c 文件包含 Marek Majkowski' 对 Dan Bernstein 的 SipHash24 算法的实现。它包含以下声明:

```
<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 和 dtoa

Python/dtoa.c 文件提供了 C 语言的 dtoa 和 strtod 函数，用于将 C 语言的双精度型和字符串进行转换，由 David M. Gay 的同名文件派生而来，该文件当前可从 http://www.netlib.org/fp/ 下载。2009 年 3 月 16 日检索到的原始文件包含以下版权和许可声明:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.12 OpenSSL

如果操作系统可用，则 hashlib, posix, ssl, crypt 模块使用 OpenSSL 库来提高性能。此外，适用于 Python 的 Windows 和 Mac OS X 安装程序可能包括 OpenSSL 库的拷贝，所以在此处也列出了 OpenSSL 许可证的拷贝:

```
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/)"
   *
   * 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,
```

(下页继续)

(繼續上一頁)

```
    * 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

除非使用 `--with-system-expat` 配置了构建，否则 `pyexpat` 扩展都是用包含 expat 源的拷贝构建的:

```
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

除非使用 `--with-system-libffi` 配置了构建，否则 `_ctypes` 扩展都是包含 libffi 源的拷贝构建的:

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 zlib

如果系统上找到的 zlib 版本太旧而无法用于构建，则使用包含 zlib 源代码的拷贝来构建 `zlib` 扩展:

```
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

tracemalloc 使用的哈希表的实现基于 cfuhash 项目:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"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

除非使用 --with-system-libmpdec 配置了构建，否则 _decimal 模块都是用包含 libmpdec 库的拷贝构建的。

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
```

(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.3.18 W3C C14N 测试套件

`test` 包中的 C14N 2.0 测试集 (`Lib/test/xmltestdata/c14n-20/`) 提取自 W3C 网站 https://www.w3.org/TR/xml-c14n2-testcases/ 并根据 3 条款版 BSD 许可证发行:

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

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

非字母

A

B

C

D

P

Q

R

S

T

U

V

W

Z