Python Tutorial

發🅵 *3.14.0rc3*

Guido van Rossum and the Python development team

10 月 01, 2025

Python Software Foundation
Email: docs@python.org

Contents

1 淺嘗滋味 **3**

2 使用 Python 直譯器 **5**
2.1 Ⓕ動直譯器 5
2.1.1 傳遞引數 6
2.1.2 互動模式 6
2.2 直譯器與它的環境 6
2.2.1 原始碼的字元編碼 (encoding) 6

3 一個非正式的 Python 簡介 **9**
3.1 把 Python 當作計算機使用 9
3.1.1 數字 (Number) 9
3.1.2 文字 11
3.1.3 List（串列） 14
3.2 初探程式設計的前幾步 16

4 深入了解流程控制 **17**
4.1 `if` 陳述式 17
4.2 `for` 陳述式 17
4.3 `range()` 函式 18
4.4 `break` 和 `continue` 陳述式 19
4.5 Ⓕ圈的 `else` 子句 19
4.6 `pass` 陳述式 20
4.7 `match` 陳述式 20
4.8 定義函式 (function) 23
4.9 深入了解函式定義 24
4.9.1 預設引數值 24
4.9.2 關鍵字引數 25
4.9.3 特殊參數 27
4.9.4 任意引數列表 (Arbitrary Argument Lists) 29
4.9.5 拆解引數列表（Unpacking Argument Lists） 29
4.9.6 Lambda 運算式 30
4.9.7 Ⓕ明文件字串 (Documentation Strings) 30
4.9.8 函式Ⓕ釋 (Function Annotations) 30
4.10 間奏曲：程式碼風格 (Coding Style) 31

5 資料結構 **33**
5.1 進一步了解 List（串列） 33
5.1.1 將 List 作Ⓕ Stack（堆Ⓕ）使用 34
5.1.2 將 List 作Ⓕ Queue（Ⓕ列）使用 35
5.1.3 List Comprehensions（串列綜合運算） 35

5.1.4 巢狀的 List Comprehensions . . . 36
5.2 del 陳述式 . . . 37
5.3 Tuples 和序列 (Sequences) . . . 37
5.4 集合 (Sets) . . . 38
5.5 字典（Dictionary） . . . 39
5.6 F圈技巧 . . . 40
5.7 深入了解條件 (Condition) . . . 41
5.8 序列和其他資料類型之比較 . . . 42

6 模組 (Module) 43
6.1 深入了解模組 . . . 44
6.1.1 把模組當作F本執行 . . . 45
6.1.2 模組的搜尋路徑 . . . 45
6.1.3 「編譯」Python 檔案 . . . 46
6.2 標准模組 . . . 46
6.3 dir() 函式 . . . 46
6.4 套件 (Package) . . . 48
6.4.1 從套件中 import * . . . 49
6.4.2 套件F引用 . . . 50
6.4.3 多目F中的套件 . . . 50

7 輸入和輸出 51
7.1 更華麗的輸出格式 . . . 51
7.1.1 格式化的字串文本 (Formatted String Literals) . . . 52
7.1.2 字串的 format() method . . . 53
7.1.3 手動格式化字串 . . . 54
7.1.4 格式化字串的舊方法 . . . 54
7.2 讀寫檔案 . . . 55
7.2.1 檔案物件的 method . . . 56
7.2.2 使用 json 儲存結構化資料 . . . 57

8 錯誤和例外 59
8.1 語法錯誤 (Syntax Error) . . . 59
8.2 例外 (Exception) . . . 59
8.3 處理例外 . . . 60
8.4 引發例外 . . . 62
8.5 例外鏈接 (Exception Chaining) . . . 63
8.6 使用者自定的例外 . . . 64
8.7 定義清理動作 . . . 64
8.8 預定義的清理動作 . . . 65
8.9 引發及處理多個無關的例外 . . . 66
8.10 用F解使例外更詳細 . . . 67

9 Class（類F） 69
9.1 關於名稱與物件的一段話 . . . 69
9.2 Python 作用域 (Scope) 及命名空間 (Namespace) . . . 69
9.2.1 作用域和命名空間的範例 . . . 71
9.3 初見 class . . . 71
9.3.1 Class definition（類F定義）語法 . . . 71
9.3.2 Class 物件 . . . 72
9.3.3 實例物件 . . . 73
9.3.4 Method 物件 . . . 73
9.3.5 Class 及實例變數 . . . 73
9.4 隨意的備F . . . 74
9.5 繼承 (Inheritance) . . . 76
9.5.1 多重繼承 . . . 76
9.6 私有變數 . . . 77
9.7 補充F明 . . . 78
9.8 F代器 (Iterator) . . . 78

9.9 Ⓕ生器 (Generator) . . . 79
9.10 Ⓕ生器運算式 . . . 80

10 Python 標准函式庫概覽 **81**
10.1 作業系統介面 . . . 81
10.2 檔案之萬用字元 (File Wildcards) . . . 81
10.3 命令列引數 . . . 82
10.4 錯誤輸出重新導向與程式終止 . . . 82
10.5 字串樣式比對 . . . 82
10.6 數學相關 . . . 83
10.7 網路存取 . . . 83
10.8 日期與時間 . . . 84
10.9 資料壓縮 . . . 84
10.10 效能量測 . . . 84
10.11 品質控管 . . . 84
10.12 標准模組庫 . . . 85

11 Python 標准函式庫概覽——第二部份 **87**
11.1 輸出格式化 (Output Formatting) . . . 87
11.2 模板化 (Templating) . . . 88
11.3 二進制資料記Ⓕ編排 (Binary Data Record Layouts) . . . 89
11.4 多執行緒 (Multi-threading) . . . 89
11.5 日Ⓕ記Ⓕ (Logging) . . . 90
11.6 弱引用 (Weak References) . . . 90
11.7 使用於 List 的工具 . . . 91
11.8 十進制浮點數運算 (Decimal Floating-Point Arithmetic) . . . 92

12 Ⓕ擬環境與套件 **93**
12.1 簡介 . . . 93
12.2 建立Ⓕ擬環境 . . . 93
12.3 用 pip 管理套件 . . . 94

13 現在可以來學習些什Ⓕ？ **97**

14 互動式輸入編輯和歷史記Ⓕ替Ⓕ **99**
14.1 Tab 鍵自動完成 (Tab Completion) 和歷史記Ⓕ編輯 (History Editing) . . . 99
14.2 互動式直譯器的替代方案 . . . 99

15 浮點數運算：問題與限制 **101**
15.1 表示法誤差 (Representation Error) . . . 104

16 附Ⓕ **107**
16.1 互動模式 . . . 107
16.1.1 錯誤處理 . . . 107
16.1.2 可執行的 Python Ⓕ本 . . . 107
16.1.3 互動式Ⓕ動檔案 . . . 108
16.1.4 客Ⓕ化模組 . . . 108

A 術語表 **109**

B 關於這份Ⓕ明文件 **127**
B.1 Python 文件的貢獻者們 . . . 127

C 沿革與授權 **129**
C.1 軟體沿革 . . . 129
C.2 關於存取或以其他方式使用 Python 的合約條款 . . . 130
C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2 . . . 130
C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0 . . . 131
C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1 . . . 131
C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2 . . . 132

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION . 133
C.3 被收Ⓕ軟體的授權與致謝 . . . 133
C.3.1 Mersenne Twister . . . 133
C.3.2 Sockets . . . 134
C.3.3 非同步 socket 服務 . . . 134
C.3.4 Cookie 管理 . . . 135
C.3.5 執行追Ⓕ . . . 135
C.3.6 UUencode 與 UUdecode 函式 . . . 136
C.3.7 XML 遠端程序呼叫 . . . 136
C.3.8 test_epoll . . . 137
C.3.9 Select kqueue . . . 137
C.3.10 SipHash24 . . . 138
C.3.11 strtod 與 dtoa . . . 138
C.3.12 OpenSSL . . . 138
C.3.13 expat . . . 141
C.3.14 libffi . . . 142
C.3.15 zlib . . . 142
C.3.16 cfuhash . . . 143
C.3.17 libmpdec . . . 143
C.3.18 W3C C14N 測試套件 . . . 144
C.3.19 mimalloc . . . 145
C.3.20 asyncio . . . 145
C.3.21 Global Unbounded Sequences (GUS) . . . 145
C.3.22 Zstandard bindings . . . 146

D 版權宣告 **147**

索引 **149**

小訣竅

這份教學是Ⓕ了那些對 Python 語言有興趣的程式設計師而設計的，**不是**對於程式設計新手的入門教學。

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Ⓕ其擴充用介面語言 (extension language)。

這份教學將非常規地介紹 Python 語言與系統的基本概念及功能。請注意，它預期你對程式設計有基本的了解。實際用 Python 直譯器進行實際操作將有助於學習，但所有範例都是獨立完整的，所以這份教學也可以離ⒻⒻ讀。

若想了解 Python 標准物件和模組的描述，請參Ⓕ library-index。在 reference-index 中，你可以學到 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，幫助你了解 Python 語言的特色與風格。讀完教學後，你將有能力Ⓕ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。

術語表 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Ⓕ，你會想在很多文字檔案Ⓕ做相同的搜尋取代，或者是用個Ⓕ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Ⓕ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，Ⓕ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Ⓕ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Ⓕ不想要Ⓕ了這程式特Ⓕ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Ⓕ了某些任務而寫個 Unix shell Ⓕ本或者 Windows 批次檔來處理，但 shell Ⓕ本最適合於搬動檔案或更動文字Ⓕ容，而不適於圖形應用程式或Ⓕ戲。你可以Ⓕ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，Ⓕ能在 Windows、macOS、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 Ⓕ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Ⓕ一個「非常高階的程式語言」，它Ⓕ建了高階的資料型Ⓕ如彈性的數列與字典。因Ⓕ這些多用途的資料型Ⓕ，Python 適用解Ⓕ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) Ⓕ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Ⓕ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Ⓕ不需要編譯與連結，能Ⓕ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Ⓕ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Ⓕ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Ⓕ能在一陳述式 (statement) 中表達很Ⓕ雜的操作；
- 陳述式的段落以縮排Ⓕ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Ⓕ要加個新的Ⓕ建函式或模組到直譯器中是很容易的。無論是Ⓕ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，Ⓕ在該應用程式中使用 Python 寫擴充或者作Ⓕ下達指令的語言。

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

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

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

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

CHAPTER 2

使用 Python 直譯器

2.1 Ⓕ動直譯器

在有安裝的機器上 Python 直譯器通常是被安裝在 `/usr/local/bin/python3.14`；把 `/usr/local/bin` 放在 Unix shell 的搜尋路徑中就可以透過輸入指令來Ⓕ動它：

```
python3.14
```

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

Windows 系統中，從 Microsoft Store 安裝 Python 後，就可以使用 `python3.14` 命令了。如果安裝了 py.exe launcher ，則可以使用 `py` 命令。請參Ⓕ setting-envvars 了解其他Ⓕ動 Python 的方式。

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

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Ⓕ：在第一個 Python 提示符後輸入 `Control-P`，如果出現ⒻⒻ聲，就代表有支援；見附Ⓕ*互動式輸入編輯和歷史記Ⓕ替Ⓕ*介紹相關的快速鍵。如果什Ⓕ事都Ⓕ有發生，或者出現一個 `^P`，就代表ⒻⒻ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Ⓕ執行指令；如果被呼叫時給定檔名Ⓕ引數或者使用 stdin 傳入檔案Ⓕ容，它會將這個檔案視ⒻⒻ本來Ⓕ讀。

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

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

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

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Ⓕ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：

```
$ python3.14
Python 3.14 (default, April 4 2024, 09:25:04)
[GCC 10.2.0] 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，你的編輯器需要能Ⓕ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F在於有無提示字元（prompt，»> 和...）：如果要重做範例，你必須在提示字元出現的時候，輸入提示字元後方的所有F容；那些非提示字元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字元時，代表該行你必須直接F行；這被使用在多行指令結束輸入時。

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

一些範例如下：

```
# 這是第一個F解
spam = 1  # 這是第二個F解
          # ... 現在是第三個！
text = "# 這不是F解，因F它在引號F。"
```

3.1 把 Python 當作計算機使用

讓我們來試試一些簡單的 Python 指令。F動直譯器F等待第一個主提示字元 >>> 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 的語法可以使用：運算子 +、-、* 和 / 可以用來執行運算；括號 () 可以用來分群。例如：

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # 除法總是回傳浮點數
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，你可以使用 // 運算子；計算余數可以使用 %：

```
>>> 17 / 3  # 傳統除法回傳浮點數
5.666666666666667
>>>
>>> 17 // 3  # 下取整除法舍Ⓕ小數部分
5
>>> 17 % 3  # % 運算子回傳除法的余數
2
>>> 5 * 3 + 2  # 下取整商 * 除數 + 余數
17
```

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

```
>>> 5 ** 2  # 5 的平方
25
>>> 2 ** 7  # 2 的 7 次方
128
```

等於符號 (=) 可以用於Ⓕ變數賦值。賦值完之後，在下個指示字元前Ⓕ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  # 嘗試存取未定義的變數
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

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

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

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

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

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

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

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

3.1.2 文字

Python 可以操作文本（由 `str` 型Ⓕ表示，即所謂的「字串 (strings)」）和數字。這包括字元"!"、單詞"rabbit"、名稱"Paris"、句子"Got your back." 等等。"Yay! :)"。它們可以用單引號 (`'...'`) 或雙引號 (`"..."`) 括起來，會具有相同的結果[2]。

```
>>> 'spam eggs'  # 單引號
'spam eggs'
>>> "Paris rabbit got your back :)! Yay!"  # 雙引號
'Paris rabbit got your back :)! Yay!'
>>> '1975'  # 以引號包含的數字和數值也是字串
'1975'
```

要引用引文，我們需要在其前面加上 \ 來「跳Ⓕ」它。或者我們也可以使用其他種類的引號：

```
>>> 'doesn\'t'  # 用 \' 來跳Ⓕ單引號...
"doesn't"
>>> "doesn't"  # ...或改用雙引號
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

在 Python shell 中，字串定義和輸出字串可能看起來不同。`print()` 函式透過省略引號Ⓕ印出跳Ⓕ字元和特殊字元來生成更具可讀性的輸出：

```
>>> s = 'First line.\nSecond line.'  # \n 表示Ⓕ行
>>> s  # Ⓕ有 print()，特殊字元會包含在字串中
'First line.\nSecond line.'
>>> print(s)  # 有 print()，特殊字元會被直譯，所以 \n 會Ⓕ生新的一行
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')  # 這Ⓕ \n 表示Ⓕ行！
C:\some
ame
>>> print(r'C:\some\name')  # 注意引號前的 r
C:\some\name
```

原始字串有一個微妙的地方：原始字串可能不會以奇數個 \ 字元結尾；請參Ⓕ常見問答集來了解更多資訊和解Ⓕ方法。

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Ⓕ行字元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，最初的Ⓕ行符不會被包含：

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

(繼續下一頁)

[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。

(繼續上一頁)

```
>>>
```

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

```
>>> # 3 次 'un', 接著是 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

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'  # 無法串接變數和字串字面值
  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]  # 位置 0 的字元
'P'
>>> word[5]  # 位置 5 的字元
'n'
```

索引值可以是負的，此時改成從右開始計數：

```
>>> word[-1]  # 最後一個字元
'n'
>>> word[-2]  # 倒數第二個字元
'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一個子字串 (substring)：

```
>>> word[0:2]  # 從位置 0（包含）到 2（不包含）的字元
'Py'
>>> word[2:5]  # 從位置 2（包含）到 5（不包含）的字元
'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]   # 從開頭到位置 2（不包含）的字元
'Py'
>>> word[4:]   # 從位置 4（包含）到結尾的字元
'on'
>>> word[-2:]  # 從倒數第二個（包含）到結尾的字元
'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]  # 這只有六個字母
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]  # 索引回傳項目
1
>>> squares[-1]
25
>>> squares[-3:]  # 切片回傳新串列
[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]  # 這Ⓕ有問題
>>> 4 ** 3  # 4 的立方是 64，不是 65!
64
>>> cubes[3] = 64  # 替Ⓕ錯誤的值
>>> cubes
[1, 8, 27, 64, 125]
```

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

```
>>> cubes.append(216)  # 加入 6 的立方
>>> cubes.append(7 ** 3)  # 以及 7 的立方
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Python 中的簡單賦值永遠不會ⒻⒻ資料。當你將 list 指派給變數時，該變數會參照現有 *list*。任何透過一個變數對 list 所做的更改都將能Ⓕ透過參照該變數的所有其他變數看到。

```
>>> rgb = ["Red", "Green", "Blue"]
>>> rgba = rgb
>>> id(rgb) == id(rgba)  # 它們參照到同一物件
True
>>> rgba.append("Alph")
>>> rgb
["Red", "Green", "Blue", "Alph"]
```

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ⒻⒻ：

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

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

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # 透過將所有元素替ⒻⒻ空串列的方式來清空串列
>>> 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：

```
>>> # 費波那契數列
>>> # 以兩個元素的和來Ⓕ定下一個元素
>>> 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。在互動式提示字元中，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Ⓕ語法剖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，這章節還會介紹一些 Python 的陳述式語法。

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。

如果你要將同一個值與多個常數進行比較，或者檢查特定的型Ⓕ或屬性，你可能會發現 match 陳述式也很有用。更多的細節，請參Ⓕ*match 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）：

```
>>> # 測量一些字串：
>>> words = ['cat', 'window', 'defenestrate']
>>> for w in words:
```

(繼續下一頁)

(繼續上一頁)

```
...     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，很難取得想要的結果。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：

```
# 建立一個範例集合
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

# 策略：對副本進行Ⓕ代
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# 策略：建立一個新集合
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取得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
```

待會我們可以看到更多回傳 iterable 和使用 iterable Ⓕ引數的函式。在資料結構章節中，我們會討論更多關於 `list()` 的細節。

4.4 break 和 continue 陳述式

`break` 陳述式，終止包含它的最Ⓕ部 `for` 或 `while` Ⓕ圈：

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(f"{n} equals {x} * {n//x}")
...             break
...
4 equals 2 * 2
6 equals 2 * 3
8 equals 2 * 4
9 equals 3 * 3
```

`continue` 陳述式讓所屬的Ⓕ圈繼續執行下個Ⓕ代：

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print(f"Found an even number {num}")
...         continue
...     print(f"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 Ⓕ圈的 else 子句

在 `for` 或 `while` Ⓕ圈中，`break` 陳述句可能與 `else` 子句配對。如果Ⓕ圈完成而Ⓕ有執行 `break`，`else` 子句會被執行。

在 `for` Ⓕ圈中，`else` 子句會在Ⓕ圈完成最終的Ⓕ代後執行。

在 `while` Ⓕ圈中，它會在Ⓕ圈條件變Ⓕ false 後執行。

在任何一種Ⓕ圈中，如果Ⓕ圈由 `break` 終止，則不會執行 `else` 子句。當然其他提早結束Ⓕ圈的方式（例如 `return` 或引發例外）也會跳過 `else` 子句的執行。

下面的 for Ⓕ圈對此進行了舉例Ⓕ明，該Ⓕ圈用以搜尋質數：

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # Ⓕ圈結束但Ⓕ有找到因數
...         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。）

理解 else 子句的一個方法是將它想像成與Ⓕ圈Ⓕ的 if 配對。當Ⓕ圈執行時，它會運行如 if/if/if/else 的序列。if 在Ⓕ圈Ⓕ，會遇到多次。如果條件曾經Ⓕ真，break 就會發生。如果條件從未Ⓕ真，Ⓕ圈外的 else 子句就會執行。

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。

4.6 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  # 忙碌等待鍵盤中斷 (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   # 記得要實作這個！
...
```

在最後這個例子中，許多人會使用Ⓕ節號字面值 ... 來取代 pass。這種用法對 Python ⒻⒻ有特殊的意義，且Ⓕ非語言定義的一部分（你也可以在這Ⓕ使用任何常數運算式），但 ... 也慣例上被用作預留主體 (placeholder body)。見 bltin-ellipsis-object。

4.7 match 陳述式

match 陳述式會拿取一個運算式，Ⓕ將其值與多個連續的模式 (pattern) 進行比較，這些模式是以一個或多個 case 區塊來表示。表面上，這類似 C、Java 或 JavaScript（以及許多其他語言）中的 switch 陳述式，

但它與 Rust 或 Haskell 等語言中的模式匹配 (pattern matching) 更Ⓕ相近。只有第一個匹配成功的模式會被執行，而它也可以將成分（序列元素或物件屬性）從值中提取到變數中。如果Ⓕ有任何的 case 匹配成功，則不會執行任何的分支。

最簡單的形式，是將一個主題值 (subject value) 與一個或多個字面值 (literal) 進行比較：

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

請注意最後一段：「變數名稱」_ 是作Ⓕ通用字元 *(wildcard)* 的角色，且永遠不會匹配失敗。

你可以使用 | （「或」）來將多個字面值組合在單一模式中：

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

模式可以看起來像是拆解賦值 (unpacking assignment)，且可以用來連結變數：

```
# point 是一個 (x, y) 元組
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

請仔細研究那個例子！第一個模式有兩個字面值，可以想作是之前所述的字面值模式的延伸。但是接下來的兩個模式結合了一個字面值和一個變數，且該變數Ⓕ結 *(bind)* 了來自主題 (`point`) 的一個值。第四個模式會Ⓕ取兩個值，這使得它在概念上類似於拆解賦值 `(x, y) = point`。

如果你要用 class 來結構化你的資料，你可以使用該 class 的名稱加上一個引數列表，類似一個建構式 (constructor)，但它能Ⓕ將屬性Ⓕ取到變數中：

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

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

你可以將位置參數 (positional parameter) 與一些能Ⓕ排序其屬性的Ⓕ建 class（例如 dataclasses）一起使用。你也可以透過在 class 中設定特殊屬性 `__match_args__`，來定義模式中屬性們的特定位置。如果它被設定Ⓕ ("x", "y")，則以下的模式都是等價的（且都會將屬性 `y` 連結到變數 `var`）：

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

理解模式的一種推薦方法，是將它們看作是你會放在賦值 (assignment) 左側Ⓕ容的一種延伸形式，這樣就可以了解哪些變數會被設Ⓕ何值。只有獨立的名稱（像是上面的 `var`）能被 match 陳述式賦值。點分隔名稱（如 `foo.bar`）、屬性名稱（上面的 `x=` 及 `y=`）或 class 名稱（由它們後面的"(...)" 被辨識，如上面的 `Point`）則永遠無法被賦值。

模式可以任意地被巢套 (nested)。例如，如果我們有一個由某些點所組成的簡短 list，我們就可以像這樣加入 `__match_args__` 來對它進行匹配：

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

match points:
    case []:
        print("No points")
    case [Point(0, 0)]:
        print("The origin")
    case [Point(x, y)]:
        print(f"Single point {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two on the Y axis at {y1}, {y2}")
    case _:
        print("Something else")
```

我們可以在模式中加入一個 `if` 子句，稱Ⓕ「防護 (guard)」。如果該防護Ⓕ假，則 `match` 會繼續嘗試下一個 case 區塊。請注意，值的Ⓕ取會發生在防護的評估之前：

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

此種陳述式的其他幾個重要特色：

- 與拆解賦值的情Ⓕ類似，tuple（元組）和 list 模式具有完全相同的意義，而且實際上可以匹配任意的序列。一個重要的例外，是它們不能匹配Ⓕ代器 (iterator) 或字串。
- 序列模式 (sequence pattern) 可支援擴充拆解 (extended unpacking)：`[x, y, *rest]` 與 `(x, y, *rest)` 的作用類似於拆解賦值。`*` 後面的名稱也可以是 `_`，所以 `(x, y, *_)` 會匹配一個至少兩項的序列，且不會連結那兩項以外的其余項。
- 對映模式 (mapping pattern)：`{"bandwidth": b, "latency": l}` 能從一個 dictionary（字典）中Ⓕ取 `"bandwidth"` 及 `"latency"` 的值。與序列模式不同，額外的鍵 (key) 會被忽略。一種像是 `**rest` 的拆解方式，也是可被支援的。（但 `**_` 則是多余的做法，所以它Ⓕ不被允許。）
- 使用關鍵字 `as` 可以Ⓕ取子模式 (subpattern)：

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

  將會Ⓕ取輸入的第二個元素作Ⓕ `p2`（只要該輸入是一個由兩個點所組成的序列）。

- 大部分的字面值是藉由相等性 (equality) 來比較，但是單例物件 (singleton) `True`、`False` 和 `None` 是藉由標識值 (identity) 來比較。
- 模式可以使用附名常數 (named constant)。這些模式必須是點分隔名稱，以免它們被解釋FF取變數：

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

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

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

關於更詳細的解釋和其他範例，你可以F讀 **PEP 636**，它是以教學的格式編寫而成。

4.8 定義函式 (function)

我們可以建立一個函式來F生費式數列到任何一個上界：

```
>>> def fib(n):    # 寫出小於 n 的費氏數列
...     """印出小於 n 的費氏數列。"""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # 現在呼叫我們剛才定義的函式:
>>> 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目前的符號表。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。

[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）。

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):  # 回傳到 n F止的費氏數列
...     """回傳包含到 n F止的費氏數列的串列。"""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # 見下方
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # 呼叫它
>>> f100                # 寫出結果
[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F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 F「屬於」一個物件的函式，命名規則F obj.methodname，其中 obj F某個物件（亦可F一運算式），而 methodname F該 method 的名稱，F由該物件的型F所定義。不同的型F定義不同的 method。不同型F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F）定義自己的物件型F和 method，見*Class*（類F））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。

4.9 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。

4.9.1 預設引數值

F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:
        reply = input(prompt)
        if reply in {'y', 'ye', 'yes'}:
            return True
        if reply in {'n', 'no', 'nop', 'nope'}:
            return False
        retries = retries - 1
        if retries < 0:
```

(繼續下一頁)

(繼續上一頁)

```
            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)

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
```

4.9.2 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 個位置引數
parrot(voltage=1000)                                  # 1 個關鍵字引數
parrot(voltage=1000000, action='VOOOOOM')             # 2 個關鍵字引數
parrot(action='VOOOOOM', voltage=1000000)             # 2 個關鍵字引數
parrot('a million', 'bereft of life', 'jump')         # 3 個位置引數
parrot('a thousand', state='pushing up the daisies')  # 1 個位置引數、1 個關鍵字引數
```

但以下呼叫方式都無效：

```
parrot()                     # 缺少必要引數
parrot(voltage=5.0, 'dead')  # 關鍵字引數後面的非關鍵字引數
parrot(110, voltage=220)     # 同一個引數有重Ⓕ值
parrot(actor='John Cleese')  # 未知的關鍵字引數
```

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。

4.9.3 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_arg 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。

4.9.4 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'
```

4.9.5 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))            # 使用分離引數的一般呼叫
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))            # 以從串列解包而得的引數呼叫
[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 !
```

4.9.6 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式物件的地方。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。舉例來Ⓕ，`list.sort()` 接受一個鍵排序函式 *key*，這個函式可以是 lambda 函式：

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

4.9.7 Ⓕ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():
...     """不做任何事，但有文件Ⓕ明。
...
...     不，真的，它什Ⓕ都不做。
...     """
...     pass
...
>>> print(my_function.__doc__)
Do nothing, but document it.

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

4.9.8 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'
```

4.10 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*初見 class*）。
- 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。與 `a[len(a):] = [x]` 類似。

`list.`**`extend`**`(`*iterable*`)`

將 iterable（可F代物件）接到 list 的尾端。與 `a[len(a):] = iterable` 類似。

`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。若 list 是空的或是索引值超出範圍，則會引發 `IndexError` 例外。

`list.`**`clear`**`()`

F除 list 中所有項目。與 `del a[:]` 類似。

`list.`**`index`**`(`*x*[, *start*[, *end*]]`)`

回傳 list 中第一個出現 *x* 的（從零開始的）索引值。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`**`(`** (Keyword-only parameters separator (PEP 3102))*, *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ⒻⒻ (shallow copy) 的 list。與 `a[:]` 類似。

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)  # 從位置 4 開始找到下一個 banana
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`，ⒻⒻ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Ⓕ整數不能與字串比較，而 `None` 不能與其他型Ⓕ比較。有些型Ⓕ根本就Ⓕ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Ⓕ Stack（堆Ⓕ）使用

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

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

[1] 其他語言可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();`。

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 加進來
>>> queue.append("Graham")          # Graham 加進來
>>> queue.popleft()                 # 第一個加進來的現在離開
'Eric'
>>> queue.popleft()                 # 第二個加進來的現在離開
'John'
>>> queue                           # queue Ⓕ剩下的會按照加進來的順序排列
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建立 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，或是建立一個子序列，其每一個元素皆滿足一個特定的條件。

舉例來Ⓕ，假設我們要建立一個「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]
```

注意這是建立（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Ⓕ，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，Ⓕ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Ⓕ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Ⓕ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]
>>> # 建立一個值加倍的新串列
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # 過濾串列以排除負數
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # 對所有元素套用函式
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # 對每個元素呼叫方法
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # 建立像 (數字, 平方) 的 2-元組串列
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # 元組必須加上括號，否則會Ⓕ生錯誤
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1
    [x, x**2 for x in range(6)]
     ^^^^^^^
SyntaxError: did you forget parentheses around the comprehension target?
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List comprehensions 可以含有Ⓕ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，Ⓕ部的 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])
...
```

(繼續下一頁)

(繼續上一頁)

```
>>> 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):
...     # 下面這三行實作了巢狀的 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]
```

(繼續下一頁)

(繼續上一頁)

```
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuple 可以Ⓕ巢狀的:
>>> u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuple Ⓕ不可變的:
>>> 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，關於建立一個含有 0 個或 1 個項目的 tuple：語法上會Ⓕ納一些奇怪的用法。空的 tuple 藉由一對空括號來建立；含有一個項目的 tuple 經由一個值加上一個逗點來建立（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',    # <-- 注意後面有個逗號
>>> 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隸屬資格檢測和消除重Ⓕ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()` 函式都可以用來建立 set。注意：要建立一個空的 set，你必須使用 `set()` 而不是 `{}`；後者會建立一個空的 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)                      # 確認重Ⓕ值已被移除
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # 快速成員測試
True
>>> 'crabgrass' in basket
False

>>> # 示範對兩個字的唯一字母進行集合運算
>>>
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # a 中的唯一字母
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # 在 a 中但不在 b 中的字母
{'r', 'd', 'b'}
>>> a | b                              # 在 a 或 b 或兩者中的字母
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # 同時在 a 和 b 中的字母
{'a', 'c'}
>>> a ^ b                              # 在 a 或 b 中但不在兩者中的字母
{'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()` 等方法被修改。

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。使用一對大括號可建立一個空的 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}
```

(繼續下一頁)

(繼續上一頁)

```
>>> 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建立 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建立 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)
```

(繼續下一頁)

(繼續上一頁)

```
...
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，通常建立一個新的 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` 用於隸屬資格檢測，在容器中檢查一個值是否存在（或不存在）。運算子 `is` 和 `is not` 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
>>> 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 的編碼位置 (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 (Module)

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F容，F且用該檔案來運行它。這就是一個F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FF該函式的原始定義到所有使用它的程式F。

F了支援這一點，Python 有一種方法可以將定義放入檔案中，F在互動模式下的直譯器中使用它們。這種檔案稱F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F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F字串）會是全域變數 `__name__` 的值。例如，用你喜歡的文字編輯器在資料夾中創一個名F `fibo.py` 的檔案，F容如下：

```
# 費波那契數模組

def fib(n):
    """寫出費波那契數列至第 n 位。"""
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):
    """回傳費波那契數列至第 n 位。"""
    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F用以下指令 import 這個模組：

```
>>> import fibo
```

這F不會將 `fibo` 中定義的函式名稱直接加入目前的*namespace* 中（詳情請見*Python 作用域 (Scope) 及命名空間 (Namespace)*）；它只會加入 `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'
```

如果你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命名空間 (namespace)，模組Ⓕ定義的函式會把該模組的私有符號表當成全域命名空間使用。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。如放置在模組的最高層（不在任何函式或 class 中），被 import 的模組名稱將被加入全域命名空間中。

`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 至原模組的命名空間中。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Ⓕ不會被引入在區域命名空間中（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，Ⓕ且可能覆蓋了某些你已經定義的Ⓕ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。

```
>>> 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：

[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，會將函式名稱加到模組的全域命名空間。

```
>>> 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，如果你更改了模組，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Ⓕ剖析 (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。模組名稱列在 `sys.builtin_module_names` 當中。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，則是目前資料夾）。
- `PYTHONPATH`（一連串和 shell 變數 `PATH` 的語法相同的資料夾名稱）。
- 與安裝相關的預設值（按慣例會包含一個 `site-packages` 資料夾，它是由 `site` 模組所處理）。

在 sys-path-init 有更多的細節。

> **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，在標準函式庫路徑之前。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。開關參數 `-O` Ⓕ除 assert（斷言）陳述式，而 `-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，因此只有在你知道自己在做什Ⓕ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建立.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']
```

(繼續下一頁)

(繼續上一頁)

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

Ⓕ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，它們被定義在標准模組 `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',
```

(繼續下一頁)

(繼續上一頁)

```
'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F `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F，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F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，F了不同檔案格式之間的轉F，你會需要建立和維護一個不斷增長的模組集合。F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` F的目F，尋找套件的子目F。

目F中必須含有 `__init__.py` 檔案，才會被 Pyhon 當成套件（除非有使用*namespace package*，F一個相對進階的功能）；這樣可以避免一些以常用名稱命名（例如 `string`）的目F，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F，`__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F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.effects 套件中，這三個被提名的子模組。

請注意，子模組可能會被區域定義 (locally defined) 的名稱遮蔽。例如，如果你在 sound/effects/__init__.py 檔案中新增了一個 reverse 函式，則 from sound.effects import * 只會引入兩個子模組 echo 和 surround，但不是 reverse 子模組，因Ⓕ它被區域定義的 reverse 函式遮蔽了：

```
__all__ = [
    "echo",       # 指向 'echo.py' 檔案
    "surround",   # 指向 'surround.py' 檔案
    "reverse",    # !!! 現在指向 'reverse' 函式 !!!
]

def reverse(msg: str):  #  <-- 在 'from sound.effects import *' 的情Ⓕ下
    return msg[::-1]    #     這個名稱遮蔽了 'reverse.py' 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 到目前的命名空間；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 進目前的命名空間，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 中的目前套件和母套件。例如，在 `urround` 模組中，你可以使用：

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

請注意，相對 import 的運作是以目前模組的套件名稱Ⓕ依據。因Ⓕ主模組ⒻⒻ有套件，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Ⓕ對引入。

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字串的序列的 `__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；標准輸出的檔案是使用 `sys.stdout` 來達成的。詳細的資訊請參考對應的函式庫Ⓕ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。在以下程式碼區塊中，有兩個如何格式化變數的範例：

```
>>> yes_votes = 42_572_654
>>> total_votes = 85_705_149
>>> percentage = yes_votes / total_votes
>>> '{:-9} YES votes  {:2.2%}'.format(yes_votes, percentage)
' 42572654 YES votes  49.67%'
```

  請注意 `yes_votes` 如何對於負數用空格和負號填補。該範例還會列出 `percentage` 乘以 100，Ⓕ保留 2 位小數且後面跟著一個百分號（有關詳細資訊，請參Ⓕ formatspec）。

- 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...
>>> # 字串的 repr() 會加上字串引號和反斜Ⓕ:
>>> hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # repr() 的引數可以是任何 Python 物件:
>>> repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

string 模組包含一個基於正規表示式 (regular expressions)、透過 string.Template 的簡單模板化手段的支援。這提供了另一種將值替Ⓕ到字串中的方式，使用像 $x 這樣的Ⓕ位符號，Ⓕ用來自字典的值取代它們。這種語法較易用，但它在格式化方面提供的控制較少。

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'.
```

= Ⓕ明符可用於將一個運算式擴充Ⓕ該運算式的文字、一個等號、以及對該運算式求值 (evaluate) 後的表示法：

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

更多關於 = Ⓕ明符的資訊請見自文件性運算式 (self-documenting expressions)。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` 字典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：

```
>>> table = {k: str(v) for k, v in vars().items()}
>>> message = " ".join([f'{k}: ' + '{' + k +'};' for k in table.keys()])
>>> print(message.format(**table))
__name__: __main__; __doc__: None; __package__: None; __loader__: ...
```

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))
...
 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=' ')
...     # 注意前一列使用的 'end'
...     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 `format % values` 中（其中 *format* 是個字串），*format* Ⓕ的 `%` 轉Ⓕ規格會被 *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位置引數和一個關鍵字引數是：`open(filename, mode, encoding=None)`

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

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 *encoding* 進行編碼。如未指定 *encoding*，則預設值會取Ⓕ於系統平台（見 `open()`）。因Ⓕ UTF-8 是現時的標准，除非你很清楚該用什Ⓕ編碼，否則推薦使用 `encoding="utf-8"`。在 mode 後面加上 `'b'` 會以 *binary mode*（二進制模式）開Ⓕ檔案，二進制模式資料以 `bytes` 物件的形式被讀寫。以二進制模式開Ⓕ檔案時不可以指定 *encoding*。

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', encoding="utf-8") as f:
...     read_data = f.read()

>>> # 我們可以檢查確認該檔案已經自動被關閉。
>>> 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)  # 將元組轉Ⓕ成字串
>>> f.write(s)
18
```

`f.tell()` 回傳一個整數，它給出檔案物件在檔案中的目前位置，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 表示使用目前的檔案位置，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
>>> f.seek(5)      # 跳到檔案中的第六個位元組
5
>>> f.read(1)
b'5'
```

(繼續下一頁)

(繼續上一頁)

```
>>> f.seek(-3, 2)  # 跳到結尾前的第三個位元組
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*binary file* 或*text file* 物件，要再次解碼物件的話：

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

> **備Ⓕ**
>
> JSON 檔案必須以 UTF-8 格式編碼。在開Ⓕ JSON 檔案以作Ⓕ一個可讀取與寫入的*text file* 時，要用 `encoding="utf-8"`。

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錯誤的地方，但請注意這F非都會是應該去修改的地方。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（此例中F `<stdin>`）和列號會被印出來，所以如果訊息是來自一個檔案時，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>
    10 * (1/0)
         ~^~
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    4 + spam*3
        ^^^^
```

(繼續下一頁)

(繼續上一頁)

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

錯誤訊息的最後一行指示發生了什Ⓕ事。例外有不同的類型，而類型名稱會作Ⓕ訊息的一部份被印出。範例中的例外類型Ⓕ：`ZeroDivisionError`、`NameError` 和 `TypeError`。作Ⓕ例外類型被印出的字串，就是發生的Ⓕ建例外 (built-in exception) 的名稱。所有的Ⓕ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Ⓕ建的識Ⓕ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，Ⓕ明例外的細節。

錯誤訊息的開頭，用堆Ⓕ回溯 (stack traceback) 的形式顯示發生例外的語境。一般來Ⓕ，它含有一個列出源程式碼行 (source line) 的堆Ⓕ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Ⓕ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Ⓕ容，直到有效的整數被輸入Ⓕ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Ⓕ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Ⓕ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/except 區塊之後的程式碼。
- 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，Ⓕ顯示錯誤訊息。

`try` 陳述式可以有不只一個 *except* 子句，Ⓕ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Ⓕ其他處理者Ⓕ的例外。一個 *except* 子句可以用一組括號Ⓕ的 tuple 列舉多個例外，例如：

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

`except` 子句中的一個類Ⓕ符合該類Ⓕ本身或它的衍生類Ⓕ的例外（但不是反過來 -- 列出一個衍生類Ⓕ的 *except* 子句不會符合它的基底類Ⓕ的例外）。例如，以下程式碼會按照 B、C、D 的順序印出：

```
class B(Exception):
    pass

class C(B):
```

(繼續下一頁)

(繼續上一頁)

```
    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。

當例外發生時，它可能有相關聯的值，也就是例外的*引數*。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)，其引數通常儲存在 `args` 屬性中。Ⓕ了方便，Ⓕ建例外型Ⓕ定義了 `__str__()` 以印出所有引數而不需顯式地取用 `.args`：

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))     # 例外的型Ⓕ
...     print(inst.args)      # 儲存在 .args 中的引數
...     print(inst)           # __str__ 使得引數可以直接被印出，
...                           # 但可能在例外的子類Ⓕ中被覆蓋
...     x, y = inst.args      # 解包引數
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

例外的 `__str__()` 輸出會被印在未處理例外訊息的最後一部分（「細節」）。

`BaseException` 是由全部的例外所共用的 base class。它的 subclass（子類Ⓕ）之一，`Exception`，則是所有非嚴重例外 (non-fatal exception) 的 base class。有些例外不是 `Exception` 的 subclass，而它們通常不會被處理，因Ⓕ它們是用來指示程式應該終止。這些例外包括了由 `sys.exit()` 所引發的 `SystemExit`，以及當使用者想要中斷程式時所引發的 `KeyboardInterrupt`。

`Exception` 可以用作通配符 (wildcard) 來捕獲（幾乎）所有的例外。然而，比較好的做法是盡可能具體地Ⓕ明我們打算處理的例外類型，Ⓕ容許任何非預期例外的傳遞 (propagate)。

處理 `Exception` 的最常見模式，是先將該例外印出或記Ⓕ，然後再重新引發它（也允許一個呼叫函式 (caller) 來處理該例外）：

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
```

(繼續下一頁)

(繼續上一頁)

```
    print("OS error:", err)
except ValueError:
    print("Could not convert data to an integer.")
except Exception as err:
    print(f"Unexpected {err=}, {type(err)=}")
    raise
```

try ... except 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。

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>
    raise NameError('HiThere')
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 BaseException 的 class，例如 Exception 與它的 subclass）。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  # 'raise ValueError()' 的簡寫
```

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!')
...     raise
```

(繼續下一頁)

(繼續上一頁)

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

8.5 例外鏈接 (Exception Chaining)

如果在 `except` 段落Ⓕ部發生了一個未處理的例外，則它會讓這個將要被處理的例外附加在後，Ⓕ將其包含在錯誤訊息中：

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

During handling of the above exception, another exception occurred:

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

Ⓕ了表明一個例外是另一個例外直接造成的結果，`raise` 陳述式容許一個選擇性的 `from` 子句：

```
# exc 必須是例外實例或 None。
raise RuntimeError from exc
```

要變Ⓕ例外時，這種方式很有用。例如：

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

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

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

它也容許使用慣用語 `from None` 來停用自動例外鏈接：

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

程式可以透過建立新的例外 class 來命名自己的例外（深入了解 Python class，詳見*Class*（類Ⓕ））。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。

大多數的例外定義，都會以「Error」作Ⓕ名稱結尾，類似於標准例外的命名。

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。

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

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。這可能會令人困惑，因此不建議這Ⓕ做。從 3.14 版本開始，編譯器會Ⓕ此發出 `SyntaxWarning`（詳見 **PEP 765**）。
- 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。這可能會令人困惑，因此不建議這Ⓕ做。從 3.14 版本開始，編譯器會Ⓕ此發出 `SyntaxWarning`（詳見 **PEP 765**）。

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>
    divide("2", "1")
    ~~~~~~^^^^^^^^^^
  File "<stdin>", line 3, in divide
    result = x / y
             ~~^~~
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。

8.9 引發及處理多個無關的例外

在某些情Ⓕ下，必須回報已經發生的多個例外。在Ⓕ行框架 (concurrency framework) 中經常會出現這種情Ⓕ，當平行的 (parallel) 某些任務可能已經失效，但還有其他用例 (use case) 希望能繼續執行Ⓕ收集多個例外，而不是只有引發第一個例外時。

Ⓕ建的 `ExceptionGroup` 會包裝一個例外實例 (exception instance) 的 list（串列），使得它們可以一起被引發。由於它本身就是一個例外，因此它也可以像任何其他例外一樣被捕獲。

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

若使用 `except*` 代替 `except`，我們可以選擇性地只處理該群組中與特定類型匹配的例外。在以下範例中，展示了一個巢狀的例外群組 (exception group)，每個 `except*` 子句分Ⓕ從該群組中提取一個特定類型的例外，同時讓所有其他的例外都傳遞到其他子句，最後再被重新引發。

```
>>> def f():
...     raise ExceptionGroup(
...         "group1",
...         [
...             OSError(1),
...             SystemError(2),
...             ExceptionGroup(
...                 "group2",
...                 [
...                     OSError(3),
...                     RecursionError(4)
...                 ]
...             )
...         ]
...     )
...
>>> try:
...     f()
... except* OSError as e:
...     print("There were OSErrors")
```

(繼續下一頁)

(繼續上一頁)

```
... except* SystemError as e:
...     print("There were SystemErrors")
...
There were OSErrors
There were SystemErrors
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  |     f()
  |     ~^^
  |   File "<stdin>", line 2, in f
  |     raise ExceptionGroup(
  |     ...<12 lines>...
  |     )
  | ExceptionGroup: group1 (1 sub-exception)
  +-+---------------- 1 ----------------
    | ExceptionGroup: group2 (1 sub-exception)
    +-+---------------- 1 ----------------
      | RecursionError: 4
      +------------------------------------
>>>
```

請注意，被巢套在例外群組中的例外必須是實例，而不是類型。這是因Ⓕ在實務上，這些例外通常是已經被程式引發Ⓕ捕獲的例外，類似以下的模式：

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

8.10 用Ⓕ解使例外更詳細

當一個例外是Ⓕ了被引發而建立時，它通常會伴隨著一些資訊被初始化，這些資訊描述了當下發生的錯誤。在某些情Ⓕ，在例外被捕獲之後添加資訊會很有用。Ⓕ此，例外具有一個 `add_note(note)` method（方法），它可以接受一個字串Ⓕ將其添加到例外的Ⓕ解清單中。標準的回溯呈現會在例外之後列出所有的Ⓕ解，Ⓕ按照其被添加的順序來排列。

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

例如，在將例外收集到例外群組中時，我們可能希望Ⓕ各個錯誤添加一些上下文的資訊。在以下範例中，群組中的每個例外都有一條Ⓕ解，指示此錯誤是在何時發生。

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

CHAPTER 9

Class（類F）

Class 提供了一種結合資料與功能的手段。建立一個 class 將會新增一個物件的型F *(type)*，F且允許建立該型F的新實例 *(instance)*。每一個 class 實例可以擁有一些維持該實例狀態的屬性 (attribute)。Class 實例也可以有一些（由其 class 所定義的）method（方法），用於修改該實例的狀態。

與其他程式語言相比，Python 的 class 機制F class 增加了最少的新語法跟語意。他混合了 C++ 和 Modula-3 的 class 機制。Python 的 class 提供了所有物件導向程式設計 (Object Oriented Programming) 的標准特色：class 繼承機制允許多個 base class（基底類F），一個 derived class（衍生類F）可以覆寫 (override) 其 base class 的任何 method，且一個 method 可以用相同的名稱呼叫其 base class 的 method。物件可以包含任意數量及任意種類的資料。如同模組一樣，class 也具有 Python 的動態特性：他們在執行期 (runtime) 被建立，且可以在建立之後被修改。

在 C++ 的術語中，class 成員（包含資料成員）通常都是公開的（除了以下F容：私有變數），而所有的成員函式都是F擬的。如同在 Modula-3 中一樣，Python FF有提供簡寫可以從物件的 method F參照其成員：method 函式與一個外顯的 (explicit)、第一個代表物件的引數被宣告，而此引數是在呼叫時隱性地 (implicitly) 被提供。如同在 Smalltak 中，class 都是物件，這F import 及重新命名提供了語意。不像 C++ 和 Modula-3，Pyhon F建的型F可以被使用者以 base class 用於其他擴充 (extension)。另外，如同在 C++ 中，大多數有著特F語法的F建運算子（算術運算子、下標等）都可以F了 class 實例而被重新定義。

（由於缺乏普遍能接受的術語來討論 class，我偶爾會使用 Smalltalk 和 C++ 的術語。我會使用 Modula-3 的術語，因F它比 C++ 更接近 Python 的物件導向語意，但我預期比較少的讀者會聽過它。）

9.1 關於名稱與物件的一段話

物件有個體性 (individuality)，且多個名稱（在多個作用域 (scope) ）可以被連結到相同的物件。這在其他語言中被稱FF名 (aliasing)。初次接觸 Python 時通常不會注意這件事，而在處理不可變的基本型F（數值、字串、tuple）時，它也可以安全地被忽略。然而，F名在含有可變物件（如 list（串列）、dictionary（字典）、和大多數其他的型F）的 Python 程式碼語意中，可能會有意外的效果。這通常有利於程式，因FF名在某些方面表現得像指標 (pointer)。舉例來F，在實作時傳遞一個物件是便宜的，因F只有指標被傳遞；假如函式修改了一個作F引數傳遞的物件，呼叫函式者 (caller) 能F見到這些改變——這消除了在 Pascal 中兩個相F引數傳遞機制的需求。

9.2 Python 作用域 (Scope) 及命名空間 (Namespace)

在介紹 class 之前，我必須先告訴你一些關於 Python 作用域的規則。Class definition（類F定義）以命名空間展現了一些俐落的技巧，而你需要了解作用域和命名空間的運作才能完整理解正在發生的事情。順帶一提，關於這個主題的知識對任何進階的 Python 程式設計師都是很有用的。

讓我們從一些定義開始。

*命名空間*是從名稱到物件的對映。大部分的命名空間現在都是以 Python 的 dictionary 被實作，但通常不會以任何方式被察覺（除了性能），且它可能會在未來改變。命名空間的例子有：Ⓕ建名稱的集合（包含如 `abs()` 的函式，和Ⓕ建的例外名稱）；模組中的全域 (global) 名稱；和在函式呼叫中的區域 (local) 名稱。某種意義上，物件中的屬性集合也會形成一個命名空間。關於命名空間的重要一點是，不同命名空間中的名稱之間Ⓕ對Ⓕ有關Ⓕ；舉例來Ⓕ，兩個不一樣的模組都可以定義一個 `maximize` 函式而不會混淆——模組的使用者必須Ⓕ它加上前綴 (prefix) 模組名稱。

順帶一提，我使用*屬性 (attribute)* 這個字，統稱句號 (dot) 後面的任何名稱——例如，運算式中的 `z.real`，`real` 是物件 `z` 的一個屬性。嚴格來Ⓕ，模組中名稱的參照都是屬性參照：在運算式 `modname.funcname` 中，`modname` 是模組物件而 `funcname` 是它的屬性。在這種情Ⓕ下，模組的屬性和模組中定義的全域名稱碰巧有一個直接的對映：他們共享了相同的命名空間![1]

屬性可以是唯讀的或可寫的。在後者的情Ⓕ下，對屬性的賦值是可能的。模組屬性是可寫的：你可以寫 `modname.the_answer = 42`。可寫屬性也可以用 `del` 陳述式Ⓕ除。例如，`del modname.the_answer` 將從名Ⓕ `modname` 的物件中Ⓕ除屬性 `the_answer`。

命名空間在不同的時刻被建立，Ⓕ且有不同的壽命。當 Python 直譯器Ⓕ動時，含有Ⓕ建名稱的命名空間會被建立，Ⓕ且永遠不會被Ⓕ除。當模組定義被讀入時，模組的全域命名空間會被建立；一般情Ⓕ下，模組的命名空間也會持續到直譯器結束。被直譯器的頂層呼叫 (top-level invocation) 執行的陳述式，不論是從Ⓕ本檔案讀取的或是互動模式中的，會被視Ⓕ一個稱Ⓕ `__main__` 的模組的一部分，因此它們具有自己的全域命名空間。（Ⓕ建名稱實際上也存在一個模組中，它被稱Ⓕ `builtins`。）

函式的區域命名空間是在呼叫函式時建立的，而當函式返回，或引發了未在函式中處理的例外時，此命名空間將會被Ⓕ除。（實際上，忘記是描述實際發生的事情的更好方法。）當然，每個遞Ⓕ呼叫 (recursive invocation) 都有自己的區域命名空間。

*作用域*是 Python 程式中的一個文本區域 (textual region)，在此區域，命名空間是可直接存取的。這Ⓕ的「可直接存取的」意思是，對一個名稱的非限定參照 (unqualified reference) 可以在命名空間Ⓕ嘗試尋找該名稱。

Ⓕ管作用域是Ⓕ態地被Ⓕ定，但它們是動態地被使用的。在執行期間Ⓕ的任何時間點，都會有 3 或 4 個巢狀的作用域，其命名空間是可以被直接存取的：

- 最Ⓕ層作用域，會最先被搜尋，而它包含了區域名稱
- 任何外圍函式 (enclosing function) 的作用域，會從最近的外圍作用域開始搜尋，它包含了非區域 (non-local) 和非全域 (non-global) 的名稱
- 倒數第二個作用域，包含目前模組的全域名稱
- 最外面的作用域（最後搜尋），是包含Ⓕ建名稱的命名空間

如果一個名稱被宣告Ⓕ全域，則所有的參照和賦值將直接轉到包含模組全域名稱的倒數第二個作用域。要重新連結最Ⓕ層作用域以外找到的變數，可以使用 `nonlocal` 陳述式；如果那些變數Ⓕ有被宣告Ⓕ nonlocal，則它們會是唯讀的（嘗試寫入這樣的變數只會在最Ⓕ層的作用域Ⓕ建立一個*新的*區域變數，同名的外部變數則維持不變）。

通常，區域作用域會參照（文本的）目前函式的區域名稱。在函式外部，區域作用域與全域作用域參照相同的命名空間：模組的命名空間。然而，Class definition 會在區域作用域中放置另一個命名空間。

務必要了解，作用域是按文本被Ⓕ定的：在模組中定義的函式，其全域作用域便是該模組的命名空間，無論函式是從何處或以什Ⓕ Ⓕ名被呼叫。另一方面，對名稱的實際搜尋是在執行時期 (run time) 動態完成的——但是，語言定義的發展，正朝向在「編譯」時期 (compile time) 的Ⓕ態名稱解析 (static name resolution)，所以不要太依賴動態名稱解析 (dynamic name resolution)！（事實上，局部變數已經是Ⓕ態地被Ⓕ定。）

一個 Python 的特殊癖好是——假如Ⓕ有 `global` 或 `nonlocal` 陳述式的效果——名稱的賦值 (assignment) 都會指向最Ⓕ層作用域。賦值不會Ⓕ Ⓕ資料——它們只會把名稱連結至物件。Ⓕ除也是一樣：陳述式 `del x` 會從區域作用域參照的命名空間移除 `x` 的連結。事實上，引入新名稱的所有運算都使用區域作用域：特Ⓕ是 `import` 陳述式和函式定義，會連結區域作用域Ⓕ的模組或函式名稱。

[1] 有一個例外。模組物件有一個秘密的唯讀屬性，稱Ⓕ `__dict__`，它回傳用於實作模組命名空間的 dictionary；`__dict__` 這個名稱是一個屬性但不是全域名稱。顯然，使用此屬性將違反命名空間實作的抽象化，而應該僅限用於事後除錯器 (post-mortem debugger) 之類的東西。

global 陳述式可以用來表示特定變數存活在全域作用域，應該被重新綁定到那Ⓕ；nonlocal 陳述式表示特定變數存活在外圍作用域Ⓕ，應該被重新綁定到那Ⓕ。

9.2.1 作用域和命名空間的范例

這是一個範例，演示如何參照不同的作用域和命名空間，以及 global 和 nonlocal 如何影響變數的綁定：

```
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)
```

範例程式碼的輸出是：

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

請注意，區域賦值（預設情Ⓕ）不會改變 *scope_test* 對 *spam* 的連結。nonlocal 賦值改變了 *scope_test* 對 *spam* 的連結，而 global 賦值改變了模組層次的連結。

你還可以發現，在 global 賦值之前，Ⓕ有對 *spam* 的連結。

9.3 初見 class

Class Ⓕ用一些新的語法，三個新的物件型Ⓕ，以及一些新的語意。

9.3.1 Class definition（類Ⓕ定義）語法

Class definition 最簡單的形式如下：

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

Class definition，如同函式定義（def 陳述式），必須在它們有任何效果前先執行。（你可以想像把 class definition 放在一個 if 陳述式的分支，或在函式Ⓕ。）

在實作時，class definition Ⓕ的陳述式通常會是函式定義，但其他陳述式也是允許的，有時很有用——我們稍後會回到這Ⓕ。Class 中的函式定義通常會有一個獨特的引數列表形式，取Ⓕ於 method 的呼叫慣例——再一次地，這將會在稍後解釋。

當進入 class definition，一個新的命名空間將會被建立，Ⓕ且作Ⓕ區域作用域——因此，所有區域變數的賦值將進入這個新的命名空間。特Ⓕ是，函式定義會在這Ⓕ連結新函式的名稱。

正常地（從結尾處）離開 class definition 時，一個 *class 物件*會被建立。基本上這是一個包裝器 (wrapper)，裝著 class definition 建立的命名空間Ⓕ容；我們將在下一節中更加了解 class 物件。原始的區域作用域（在進入 class definition 之前已生效的作用域）會恢復，在此 class 物件會被連結到 class definition 標頭中給出的 class 名稱（在範例中Ⓕ `ClassName`）。

9.3.2 Class 物件

Class 物件支援兩種運算：屬性參照 (attribute reference) 和實例化 (instantiation)。

*屬性參照*使用 Python 中所有屬性參照的標準語法：`obj.name`。有效的屬性名稱是 class 物件被建立時，class 的命名空間中所有的名稱。所以，如果 class definition 看起來像這樣：

```
class MyClass:
    """一個簡單的類Ⓕ範例"""
    i = 12345

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

那Ⓕ `MyClass.i` 和 `MyClass.f` 都是有效的屬性參照，會分Ⓕ回傳一個整數和一個函式物件。Class 屬性也可以被指派 (assign)，所以你可以透過賦值改變 `MyClass.i` 的值。`__doc__` 也是一個有效的屬性，會回傳屬於該 class 的Ⓕ明字串 (docstring)：`"A simple example class"`。

*Class 實例化*使用了函式記法 (function notation)。就好像 class 物件是一個Ⓕ有參數的函式，它回傳一個新的 class 實例。例如（假設是上述的 class）：

```
x = MyClass()
```

建立 class 的一個新*實例*，Ⓕ將此物件指派給區域變數 `x`。

實例化運算（「呼叫」一個 class 物件）會建立一個空的物件。許多 class 喜歡在建立物件時有著自訂的特定實例初始狀態。因此，class 可以定義一個名Ⓕ `__init__()` 的特Ⓕ method，像這樣：

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

當 class 定義了 `__init__()` method，class 實例化會Ⓕ新建的 class 實例自動叫用 `__init__()`。所以在這個範例中，一個新的、初始化的實例可以如此獲得：

```
x = MyClass()
```

當然，`__init__()` method 可能Ⓕ了更多的彈性而有引數。在這種情Ⓕ下，要給 class 實例化運算子的引數會被傳遞給 `__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 實例物件

現在，我們可以如何處理實例物件？實例物件能理解的唯一運算就是屬性參照。有兩種有效的屬性名稱：資料屬性 (data attribute) 和 method。

資料屬性對應 Smalltalk 中的「實例變數」，以及 C++ 中的「資料成員」。資料屬性不需要被宣告；和區域變數一樣，它們在第一次被賦值時就會立即存在。例如，如果 `x` 是 `MyClass` 在上述例子中建立的實例，下面的程式碼將印出值 `16`，而不留下ⒻⒻ：

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

另一種執行個體屬性參考是 *方法* (method)。方法是" 屬於" 一個物件的函式。

實例物件的有效 method 名稱取Ⓕ於其 class。根據定義，一個 class 中所有的函式物件屬性，就定義了實例的對應 method。所以在我們的例子中，`x.f` 是一個有效的 method 參照，因Ⓕ `MyClass.f` 是一個函式，但 `x.i` 不是，因Ⓕ `MyClass.i` 不是。但 `x.f` 與 `MyClass.f` 是不一樣的——它是一個 *method 物件*，而不是函式物件。

9.3.4 Method 物件

通常，一個 method 在它被連結後隨即被呼叫：

```
x.f()
```

如果像上面一樣 `x = MyClass()`，這將回傳字串 `'hello world'`。然而，ⒻⒻ有必要立即呼叫一個 method：`x.f` 是一個 method 物件，Ⓕ且可以被儲藏起來，之後再被呼叫。舉例來Ⓕ：

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

將會持續印出 `hello world` 直到天荒地老。

當一個 method 被呼叫時究竟會發生什Ⓕ事？你可能已經注意到 `x.f()` 被呼叫時Ⓕ有任何的引數，Ⓕ管 `f()` 的函式定義有指定一個引數。這個引數發生了什Ⓕ事？當一個需要引數的函式被呼叫而Ⓕ有給任何引數時，Python 肯定會引發例外——即使該引數實際上Ⓕ有被使用...

事實上，你可能已經猜到了答案：method 的特殊之處在於，實例物件會作Ⓕ函式中的第一個引數被傳遞。在我們的例子中，`x.f()` 這個呼叫等同於 `MyClass.f(x)`。一般來Ⓕ，呼叫一個有 *n* 個引數的 method，等同於呼叫一個對應函式，其引數列表 (argument list) 被建立時，會在第一個引數前插入該 method 的實例物件。

一般來Ⓕ，方法的工作原理如下。當一個實例的非資料屬性被參照時，將會搜尋該實例的 class。如果該名稱是一個有效的 class 屬性，而且是一個函式物件，則對實例物件和函式物件的參照都會被打包到方法物件中。當使用引數串列呼叫方法物件時，會根據實例物件和引數串列來建構一個新的引數串列，Ⓕ使用該新引數串列來呼叫函式物件。

9.3.5 Class 及實例變數

一般來Ⓕ，實例變數用於每一個實例的獨特資料，而 class 變數用於該 class 的所有實例共享的屬性和 method：

```
class Dog:

    kind = 'canine'         # 所有實例共享的類Ⓕ變數

    def __init__(self, name):
        self.name = name    # 每個實例獨有的實例變數
```

(繼續下一頁)

(繼續上一頁)

```
>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # Ⓕ所有 Dog 實例所共享
'canine'
>>> e.kind                  # Ⓕ所有 Dog 實例所共享
'canine'
>>> d.name                  # d 獨有
'Fido'
>>> e.name                  # e 獨有
'Buddy'
```

如同在*關於名懲與物件的一段話*的討論，共享的資料若涉及*mutable* 物件，如 list 和 dictionary，可能會Ⓕ生意外的影響。舉例來Ⓕ，下列程式碼的 *tricks* list 不應該作Ⓕ一個 class 變數使用，因Ⓕ這個 list 將會被所有的 *Dog* 實例所共享：

```
class Dog:

    tricks = []             # 誤用類Ⓕ變數

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

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

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # 出乎意料地被所有 dog 共享
['roll over', 'play dead']
```

正確的 class 設計應該使用實例變數：

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # Ⓕ每一個 dog 建立空 list

    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 隨意的備Ⓕ

如果屬性名稱同時出現在一個實例和一個 class 中，則屬性的尋找會以實例Ⓕ優先：

```
>>> class Warehouse:
...     purpose = 'storage'
```

(繼續下一頁)

(繼續上一頁)

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

資料屬性可能被 method 或是被物件的一般使用者（「用Ⓕ端」）所參照。也就是Ⓕ，class 不可用於實作純粹抽象的資料型Ⓕ。事實上，在 Python 中Ⓕ有任何可能的方法，可Ⓕ制隱藏資料——這都是基於慣例。（另一方面，以 C 編寫的 Python 實作可以完全隱藏實作細節Ⓕ且在必要時控制物件的存取；這可以被以 C 編寫的 Python 擴充所使用。）

用Ⓕ端應該小心使用資料屬性——用Ⓕ端可能會因Ⓕ覆寫他們的資料屬性，而破壞了被 method 維護的不變性。注意，用Ⓕ端可以增加他們自己的資料屬性到實例物件，但不影響 method 的有效性，只要避免名稱衝突即可——再一次提醒，命名慣例可以在這Ⓕ節省很多麻煩。

在 method 中參照資料屬性（或其他 method！）是Ⓕ有簡寫的。我發現這實際上增加了 method 的可Ⓕ讀性：在Ⓕ覽 method 時，Ⓕ不會混淆區域變數和實例變數。

通常，方法的第一個引數稱Ⓕ `self`。這僅僅只是一個慣例：`self` 這個名字對 Python 來Ⓕ完全Ⓕ有特Ⓕ的意義。但請注意，如果不遵循慣例，你的程式碼可能對其他 Python 程式設計師來Ⓕ可讀性較低，此外，也可以想像一個可能因信任此慣例而編寫的 *class* Ⓕ覽器 *(browser)* 程式。

任何一個作Ⓕ class 屬性的函式物件都Ⓕ該 class 的實例定義了一個相應的 method。函式定義不一定要包含在 class definition 的文本中：將函式物件指定給 class 中的區域變數也是可以的。例如：

```
# 在類Ⓕ以外定義的函式
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g
```

現在 `f`、`g` 和 `h` 都是 class `C` 的屬性，Ⓕ指向函式物件，所以他們都是 class `C` 實例的 method ——`h` 與 `g` 是完全一樣的。請注意，這種做法通常只會使該程式的讀者感到困惑。

Method 可以藉由使用 `self` 引數的 method 屬性，呼叫其他 method：

```
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)
```

Method 可以用與一般函式相同的方式參照全域名稱。與 method 相關的全域作用域，就是包含其定義的模組。（class 永遠不會被用作全域作用域。）雖然人們很少有在 method 中使用全域資料的充分理由，但全域作用域仍有許多合法的使用：比方Ⓕ，被 import 至全域作用域的函式和模組，可以被 method 以及在該作用域中定義的函式和 class 所使用。通常，包含 method 的 class，它本身就是被定義在這個全域作用域，在下一節，我們將看到 method 想要參照自己的 class 的一些好原因。

每個值都是一個物件，因此都具有一個 *class*，也可以稱Ⓕ它的 *type*（型Ⓕ）。它以 `object.__class__` 被儲存。

9.5 繼承 (Inheritance)

當然，如果Ⓕ有支援繼承，「class」這個語言特色就不值得被稱Ⓕ class。一個 derived class（衍生類Ⓕ）定義的語法看起來如下：

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

名稱 `BaseClassName` 必須被定義於作用域可及的命名空間，且該作用域要包含 derived class 定義。要代替 base class（基底類Ⓕ）的名稱，用其他任意的運算式也是被允許的。這會很有用，例如，當一個 base class 是在另一個模組中被定義時：

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

執行 derived class 定義的過程，與執行 base class 相同。當 class 物件被建構時，base class 會被記住。這是用於解析屬性參照：如果一個要求的屬性無法在該 class 中找到，則會繼續在 base class 中搜尋。假如該 base class 本身也是衍生自其他 class，則這個規則會遞Ⓕ地被應用。

關於 derived class 的實例化ⒻⒻ有特Ⓕ之處：`DerivedClassName()` 會建立該 class 的一個新實例。Method 的參照被解析如下：對應的 class 屬性會被搜尋，如果需要，沿著 base class 的繼承Ⓕ往下走，如果這Ⓕ生了一個函式物件，則該 method 的參照是有效的。

Derived class 可以覆寫其 base class 的 method。因Ⓕ method 在呼叫同一個物件的其他 method 時Ⓕ有特Ⓕ的特權，所以當 base class 的一個 method 在呼叫相同 base class 中定義的另一個 method 時，最終可能會呼叫到一個覆寫它的 derived class 中的 method。（給 C++ 程式設計師：Python 中所有 method 實際上都是 `virtual`。）

一個在 derived class 覆寫的 method 可能事實上是想要擴充而非單純取代 base class 中相同名稱的 method。要直接呼叫 base class 的 method 有一個簡單的方法：只要呼叫 `BaseClassName.methodname(self, arguments)`。這有時對用Ⓕ端也很有用。（請注意，只有在 base class 在全域作用域可以用 `BaseClassName` 被存取時，這方法才有效。）

Python 有兩個Ⓕ建函式可以用於繼承：

- 使用 `isinstance()` 判斷一個實例的型Ⓕ：`isinstance(obj, int)` 只有在 `obj.__class__` 是 `int` 或衍伸自 `int` 時，結果才會是 `True`。
- 使用 `issubclass()` 判斷 class 繼承：`issubclass(bool, int)` 會是 `True`，因Ⓕ `bool` 是 `int` 的 subclass（子類Ⓕ）。但是，`issubclass(float, int)` 是 `False`，因Ⓕ `float` Ⓕ不是 `int` 的 subclass。

9.5.1 多重繼承

Python 也支援多重繼承的形式。一個有多個 base class 的 class definition 看起來像這樣子：

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

在大多數情Ⓕ下，最簡單的例子Ⓕ，你可以這樣思考，對於繼承自 parent class（父類Ⓕ）的屬性，其搜尋規則Ⓕ：深度優先、從左到右、在階層Ⓕ重Ⓕ的相同 class 中不重Ⓕ搜尋。因此，假如有一個屬性在 `DerivedClassName` Ⓕ有被找到，則在 `Base1` 搜尋它，接著（遞Ⓕ地）在 `Base1` 的 base class 中搜尋，假如在那Ⓕ又Ⓕ有找到的話，會在 `Base2` 搜尋，依此類推。

事實上，它稍微F雜一些；method 的解析順序是動態地變化，以支援對 `super()` 的合作呼叫。這個方式在其他的多重繼承語言中，稱F呼叫下一個方法 (call-next-method)，且比在單一繼承語言中的 super call（超級呼叫）來得更F大。

動態排序是必要的，因F多重繼承的所有情F都表現一或多的菱形關F（其中至少一個 parent class 可以從最底層 class 透過多個路徑存取）。例如，所有的 class 都繼承自 `object`，因此任何多重繼承的情F都提供了多個到達 `object` 的路徑。F了避免 base class 被多次存取，動態演算法以這些方式將搜尋順序F性化 (linearize)：保留每個 class 中規定的從左到右的順序、對每個 parent 只會呼叫一次、使用單調的 (monotonic) 方式（意思是，一個 class 可以被 subclassed（子類F化），而不會影響其 parent 的搜尋優先順序）。總之，這些特性使設計出可靠又可擴充、具有多重繼承的 class 成F可能。更多資訊，請見 python_2.3_mro。

9.6 私有變數

「私有」(private) 實例變數，指的是不在物件F部便無法存取的變數，這在 Python 中是不存在的。但是，大多數 Python 的程式碼都遵守一個慣例：前綴F一個底F的名稱（如：`_spam`）應被視F API（應用程式介面）的非公有 (non-public) 部分（無論它是函式、方法或是資料成員）。這被視F一個實作細節，如有調整，亦不另行通知。

既然 class 私有的成員已有一個有效的用例（即避免名稱與 subclass 定義的名稱衝突），這種機制也存在另一個有限的支援，稱F *name mangling*（名稱修飾）。任何格式F `__spam`（至少兩個前導下底F，最多一個尾隨下底F）的物件名稱 (identifier) 會被文本地被替FF `_classname__spam`，在此 `classname` 就是去掉前導下底F的目前 class 名稱。只要這個修飾是在 class 的定義之中發生，它就會在不考慮該物件名稱的語法位置的情F下完成。

也參考

參F私有名稱修飾規格的詳情與特殊情F。

名稱修飾對於讓 subclass 覆寫 method 而不用破壞 class F部的 method 呼叫，是有幫助的。舉例來F：

```
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   # 原始 update() 方法的私有副本

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # F update() 提供新的函式簽名
        # 但不會破壞 __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

在上例中，就算在 `MappingSubclass` 當中加入 `__update` 識F符，也能順利運作，因F在 `Mapping` class 中，它會被替FF `_Mapping__update`，而在 `MappingSubclass` class 中，它會被替FF `_MappingSubclass__update`。

請注意，修飾規則是被設計來避免意外；它仍可能存取或修改一個被視F私有的變數。這在特殊情F下甚至可能很有用，例如在除錯器 (debugger)。

另外也注意，傳遞給 `exec()` 或 `eval()` 的程式碼不會把叫用 class 的名稱視F目前的 class；這和 `global` 陳述式的效果類似，該效果同樣僅限於整體被位元組編譯後 (byte-compiled) 的程式碼。同樣的限制適用於 `getattr()`，`setattr()` 和 `delattr()`，以及直接參照 `__dict__` 時。

9.7 補充Ⓕ明

如果有一種資料型Ⓕ，類似於 Pascal 的「record」或 C 的「struct」，可以將一些有名稱的資料項目捆綁在一起，有時候這會很有用。符合語言習慣的做法是使用 `dataclasses`：

```
from dataclasses import dataclass

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

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

用來處理特殊抽象資料型Ⓕ的一段 Python 程式碼，經常能以傳遞一個 class 來替代，此 class 模擬該資料型Ⓕ的多種 method。例如，如果你有一個函式，它會從一個檔案物件來格式化某些資料，你也可以定義一個有 `read()` 和 `readline()` method 的 class 作Ⓕ替代方式，從字串緩衝區取得資料，Ⓕ將其作Ⓕ引數來傳遞。

實例的 method 物件也具有屬性：`m.__self__` 就是帶有 method `m()` 的實例物件，而 `m.__func__` 則是該 method 所對應的函式物件。

9.8 Ⓕ代器 (Iterator)

到目前Ⓕ止，你可能已經注意到大多數的容器 (container) 物件都可以使用 `for` 陳述式來進行Ⓕ圈：

```
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='')
```

這種存取風格清晰、簡潔且方便。Ⓕ代器的使用在 Python 中處處可見且用法一致。在幕後，`for` 陳述式會在容器物件上呼叫 `iter()`。該函式回傳一個Ⓕ代器物件，此物件定義了 `__next__()` method，而此 method 會逐一存取容器中的元素。當元素用盡時，`__next__()` 將引發 `StopIteration` 例外，來通知 `for` 終止Ⓕ圈。你可以使用Ⓕ建函式 `next()` 來呼叫 `__next__()` method；這個例子展示了它的運作方式：

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

(繼續下一頁)

(繼續上一頁)

```
    next(it)
StopIteration
```

看過Ⓕ代器協定的幕後機制後，在你的 class 加入Ⓕ代器的行Ⓕ就很容易了。定義一個 `__iter__()` method 來回傳一個帶有 `__next__()` method 的物件。如果 class 已定義了 `__next__()`，則 `__iter__()` 可以只回傳 `self`：

```
class Reverse:
    """用於向後Ⓕ圈遍歷序列的Ⓕ代器。"""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

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

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

9.9 Ⓕ生器 (Generator)

Ⓕ生器是一個用於建立Ⓕ代器的簡單而Ⓕ大的工具。它們的寫法和常規的函式一樣，但當它們要回傳資料時，會使用 `yield` 陳述式。每次在Ⓕ生器上呼叫 `next()` 時，它會從上次離開的位置恢復執行（它會記得所有資料值以及上一個被執行的陳述式）。以下範例顯示，建立Ⓕ生器可以相當地容易：

```
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
```

任何可以用Ⓕ生器來完成的事，也能用以 class Ⓕ基礎的Ⓕ代器來完成，如同前一節的描述。而讓Ⓕ生器的程式碼更Ⓕ精簡的原因是，`__iter__()` 和 `__next__()` method 會自動被建立。

另一個關鍵的特性在於，區域變數和執行狀態會在每次呼叫之間自動被儲存。這使得該函式比使用 `self.index` 和 `self.data` 這種實例變數的方式更容易編寫且更Ⓕ清晰。

除了會自動建立 method 和儲存程式狀態，當Ⓕ生器終止時，它們還會自動引發 `StopIteration`。這些特性結合在一起，使建立Ⓕ代器能與編寫常規函式一樣容易。

9.10 Ⓕ生器運算式

某些簡單的Ⓕ生器可以寫成如運算式一般的簡潔程式碼，所用的語法類似 list comprehension（串列綜合運算），但外層Ⓕ括號而非方括號。這種運算式被設計用於Ⓕ生器將立即被外圍函式 (enclosing function) 所使用的情Ⓕ。Ⓕ生器運算式與完整的Ⓕ生器定義相比，程式碼較精簡但功能較少，也比等效的 list comprehension 更Ⓕ節省記憶體。

例如：

```
>>> sum(i*i for i in range(10))                 # 平方和
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # 向量Ⓕ積
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()      # 回傳目前的工作目Ⓕ
'C:\\Python314'
>>> os.chdir('/server/accesslogs')   # 改變目前的工作目Ⓕ
>>> os.system('mkdir today')   # 在系統 shell 中執行 mkdir 指令
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。例如以下 `demo.py` 檔案：

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

以下是在命令列運行 `python demo.py one two three` 的輸出：

```
['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)   # 不重Ⓕ抽樣
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # 區間 [0.0, 1.0) 中的隨機浮點數
0.17970987693706186
>>> random.randrange(6)    # 從 range(6) 中隨機選擇整數
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://worldtimeapi.org/api/timezone/etc/UTC.txt') as response:
...     for line in response:
...         line = line.decode()             # 將 bytes 轉ⒻⒻ str
...         if line.startswith('datetime'):
...             print(line.rstrip())         # 移除最後面的Ⓕ行符號
...
datetime: 2022-01-01T01:36:47.689215+00:00

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

（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Ⓕ。

```
>>> # 能Ⓕ輕易建立與格式化
>>> 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.'

>>> # 支援運算
>>> 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):
    """計算數字串列的算術平均數。

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

import doctest
doctest.testmod()   # 自動驗證Ⓕ嵌的測試
```

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()  # 從命令列呼叫會執行所有測試
```

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，可用於建立與解碼Ⓕ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Ⓕ格式的剖析，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

Python 標准函式庫概覽——第二部份

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 輸出格式化 (Output Formatting)

`reprlib` 模組提供了一個 `repr()` 的版本，專門用來以簡短的形式顯示大型或深層的巢狀容器：

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

`pprint` 模組能對Ⓕ建和使用者自定的物件提供更Ⓕ雜的列印控制，Ⓕ且是以直譯器可讀的方式。當結果超過一行時，「漂亮的印表機」會加入Ⓕ行和縮排，以更清楚地顯示資料結構：

```
>>> 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` 模組能Ⓕ格式化文本的段落，以符合指定的螢幕寬度：

```
>>> 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))
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 模組能存取一個含有特定文化相關資料格式的資料庫。locale 模組的 format 函式有一個 grouping 屬性，可直接以群分隔符 (group separator) 將數字格式化：

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()          # 取得慣例的對映
>>> x = 1234567.8
>>> locale.format_string("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'],
...                      conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
```

11.2 模板化 (Templating)

string 模組包含一個多功能的 Template class，提供的簡化語法適合使用者進行編輯時使用。它容許使用者客Ⓕ化自己的應用程式，但不必對原應用程式做出變更。

格式化方式是使用Ⓕ位符號名稱 (placeholder name)，它是由 $ 加上合法的 Python 識Ⓕ符（字母、數字和下底Ⓕ）構成。使用大括號包覆Ⓕ位符號以允許在後面接上更多的字母和數字而無需插入空格。使用 $$ 將會跳ⒻⒻ單一字元 $：

```
>>> 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 dictionary 或關鍵字引數中未提供某個Ⓕ位符號的值，那Ⓕ substitute() method 將引發 KeyError。對於郵件合Ⓕ (mail-merge) 類型的應用程式，使用者提供的資料有可能是不完整的，此時使用 safe_substitute() method 會更適當——如果資料有缺少，它會保持Ⓕ位符號不變：

```
>>> 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 subclass（子類Ⓕ）可以指定自訂的分隔符號 (delimiter)。例如，一個相片Ⓕ覽器的批次重新命名功能，可以選擇用百分號作Ⓕ現在日期、照片序號或檔案格式的Ⓕ位符號：

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

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

img_1074.jpg --> Ashley_0.jpg
```

(繼續下一頁)

(繼續上一頁)

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

模板化的另一個應用，是將程式邏輯與多樣的輸出格式細節分離開來。這樣就可以Ⓕ XML 檔案、純文字報表和 HTML 網路報表替Ⓕ自訂的模板。

11.3 二進制資料記Ⓕ編排 (Binary Data Record Layouts)

`struct` 模組提供了 `pack()` 和 `unpack()` 函式，用於處理可變動長度的二進制記Ⓕ格式。以下範例Ⓕ明，如何在不使用 `zipfile` 模組的情Ⓕ下，使用Ⓕ圈Ⓕ覽一個 ZIP 檔案中的標頭資訊 (header information)。壓縮程式碼 `"H"` 和 `"I"` 分Ⓕ代表兩個和四個位元組的無符號數 (unsigned number)。`"<"` 表示它們是標准大小，Ⓕ使用小端 (little-endian) 位元組順序：

```
import struct

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

start = 0
for i in range(3):                      # 顯示前 3 個檔案標頭
    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     # 跳到下一個標頭
```

11.4 多執行緒 (Multi-threading)

執行緒是一種用來對非順序相依 (sequentially dependent) 的多個任務進行解耦 (decoupling) 的技術。當其他任務在背景執行時，多執行緒可以用來提升應用程式在接收使用者輸入時的反應能力。一個相關的用例是，運行 I/O 的同時，在另一個執行緒中進行計算。

以下程式碼顯示了高階的 `threading` 模組如何在背景運行任務，而主程式同時繼續運行：

```
import threading, zipfile

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

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

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

(繼續下一頁)

(繼續上一頁)

```
background.join()    # 等待背景任務結束
print('Main program waited until background was done.')
```

多執行緒應用程式的主要挑戰是，要協調多個彼此共享資料或資源的執行緒。F此，threading 模組提供了多個同步處理原始物件 (synchronization primitive)，包括鎖 (lock)、事件 (event)、條件變數 (condition variable) 和號F (semaphore)。

F管這些工具很F大，但很小的設計錯誤也可能導致一些難以重現的問題。所以，任務協調的首選方法是，把所有對資源的存取集中到單一的執行緒中，然後使用 `queue` 模組向該執行緒饋送來自其他執行緒的請求。應用程式若使用 `Queue` 物件進行執行緒間的通信和協調，會更易於設計、更易讀、更可靠。

11.5 日F記F (Logging)

`logging` 模組提供功能齊全且富彈性的日F記F系統。在最簡單的情F下，日F訊息會被發送到檔案或 `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')
```

這會F生以下輸出：

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

在預設情F，資訊和除錯訊息不會被顯示，其輸出會被發送到標准錯誤 (standard error)。其他輸出選項包括，將訊息轉發到電子郵件、資料報 (datagram)、網路插座 (socket) 或 HTTP 伺服器。新的過濾器可以根據訊息的優先順序，選擇不同的路由 (routing) 方式：`DEBUG`、`INFO`、`WARNING`、`ERROR` 及 `CRITICAL`。

日F記F系統可以直接從 Python 配置，也可以透過載入使用者可編輯的配置檔案以進行客F化的日F記F，而無須對應用程式做出變更。

11.6 弱引用 (Weak References)

Python 會自動進行記憶體管理（對大多數物件進行參照計數 (reference counting) F使用*garbage collection* 來消除循環參照）。當一個參照從記憶體被移除後不久，該記憶體就會被釋出。

此方式對大多數應用程式來F都F問題，但偶爾也需要在物件仍然被其他物件所使用時持續追F它們。可惜的是，F管只是追F它們，也會建立一個使它們永久化 (permanent) 的參照。`weakref` 模組提供的工具可以不必建立參照就能追F物件。當該物件不再被需要時，它會自動從一個弱引用表 (weakref table) 中被移除，FF弱引用物件觸發一個回呼 (callback)。典型的應用包括暫存 (cache) 那些成本較F昂貴的物件：

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return str(self.value)
...
>>> a = A(10)                   # 建立一個參照
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a            # 不會建立參照
>>> d['primary']                # 如果物件仍然存在則取得它
```

(繼續下一頁)

(繼續上一頁)

```
10
>>> del a                        # 移除一個參照
>>> gc.collect()                 # 立即執行垃圾回收
0
>>> d['primary']                 # 項目被自動移除
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']                 # 項目被自動移除
  File "C:/python314/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7 使用於 List 的工具

許多對於資料結構的需求，可以透過Ⓕ建的 list（串列）型Ⓕ來滿足。但是，有時也會根據效能的各種取舍，需要一些替代的實作。

`array` 模組提供了一個 `array` 物件，它像是 list，但只能儲存同類的資料且能緊密地儲存。下面的範例展示一個數值陣列 (array)，以兩個位元組的無符號二進數 (unsigned binary numbers) Ⓕ儲存單位（類型碼Ⓕ `"H"`），而在 Python 整數物件的正規 list 中，每個項目通常使用 16 個位元組：

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

`collections` 模組提供了一個 `deque` 物件，它像是 list，但從左側加入 (append) 和彈出 (pop) 的速度較快，而在中間查找的速度則較慢。這種物件適用於實作Ⓕ列 (queue) 和廣度優先搜尋法 (breadth first tree search)：

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

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

除了替代的 list 實作以外，函式庫也提供了其他工具，例如 `bisect` 模組，具有能Ⓕ操作 sorted list（已排序串列）的函式：

```
>>> 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` 模組提供了一些函式，能基於正規 list 來實作堆積 (heap)。最小值的項目會永遠保持在位置零。對於一些需要多次存取最小元素，但不想要對整個 list 進行排序的應用程式來Ⓕ，這會很有用：

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)                      # 將串列重新排列Ⓕ堆積順序
>>> heappush(data, -5)                 # 加入新項目
>>> [heappop(data) for i in range(3)]  # 取得三個最小的項目
[-5, 0, 1]
```

11.8 十進制浮點數運算 (Decimal Floating-Point Arithmetic)

`decimal` 模組提供了一個 `Decimal` 資料類型，用於十進制浮點數運算。相較於Ⓕ建的二進制浮點數 `float` 實作，該 class 特Ⓕ適用於下列情境

- 金融應用程式及其他需要准確十進位制表示法的應用，
- 對於精確度 (precision) 的控制，
- 控制四舍五入，以滿足法律或監管規範，
- 追Ⓕ有效的小數位數 (decimal place)，或
- 使用者會期望計算結果與手工計算相符的應用程式。

例如，要計算 70 美分的手機充電加上 5% Ⓕ金的總價，使用十進制浮點數和二進制浮點數，會算出不同的答案。如果把計算結果四舍五入到最接近的美分，兩者的差Ⓕ會更顯著：

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

`Decimal` 的運算結果會保留尾部的零，也會根據有兩個有效位數的被乘數自動推斷出有四個有效位數的乘積。Decimal 可以重現手工計算的結果，以避免Ⓕ生二進制浮點數無法准確表示十進制數值時會導致的問題。

准確的表示法使得 `Decimal` class 能Ⓕ執行對於二進制浮點數不適用的模數計算和相等性檢測：

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

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

`decimal` 模組可提供運算中需要的足Ⓕ精確度：

```
>>> 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Ⓕ方案是建立一個Ⓕ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` 將安裝執行命令的 Python 版本（如 `--version` 選項所報告的）。例如使用 `python3.12` 執行命令將安裝 3.12 版本。

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：

```
python -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
```

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']
>>>
```

要停用Ⓕ擬環境，輸入：

```
deactivate
```

於終端機中。

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 安裝套件。你可以透過你的網頁Ⓕ覽器Ⓕ覽 Python Package Index。

`pip` 有好幾個子命令：”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
```

要是你重新執行此命令，`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 `python -m 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:
```

（繼續下一頁）

(繼續上一頁)

```
      Successfully uninstalled requests-2.6.0
Successfully installed requests-2.7.0
```

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

`python -m pip show` 可以顯示一個特定套件的資訊：

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

`python -m pip list` 會顯示Ⓕ擬環境中所有已經安裝的套件：

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

`python -m pip freeze` 可以ⒻⒻ一整個已經安裝的套件清單，但是輸出使用 `python -m 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) $ python -m pip freeze > requirements.txt
(tutorial-env) $ cat requirements.txt
novas==3.1.1.3
numpy==1.9.2
requests==2.7.0
```

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

```
(tutorial-env) $ 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` Ⓕ明文件。當你撰寫了一個套件Ⓕ且想要讓它在 Python Package Index 上可以取得的話，可以參考 Python packaging user guide。

CHAPTER 13

現在可以來學習些什Ⓕ？

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

本教學是 Python 文件中的一部分。這份文件集Ⓕ頭的其他文件包含：

- library-index：

  你該好好的Ⓕ覽這份手Ⓕ，它提供了完整（但簡潔）的參考素材像是型Ⓕ、函式與標准函式庫Ⓕ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、Ⓕ生亂數、分析命令列選項、壓縮資料、及許多其他任務。Ⓕ覽函式庫參考手Ⓕ可以讓你了解有哪些模組可以用。

- installing-index：Ⓕ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Ⓕ明。這份文件Ⓕ讀起來會有些吃力，但作Ⓕ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。

- https://docs.python.org：快速存取 Python 的文件。

- https://pypi.org：Python 套件索引 (Python Package Index)，之前也被Ⓕ稱Ⓕ Cheese Shop[1]，Ⓕ總了使用者開發 Python 模組的索引，Ⓕ提供模組能Ⓕ被下載。一旦開始發Ⓕ相關程式碼，你可以將開發的作品ⒻⒻ到這ⒻⒻ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Ⓕ本。特Ⓕ值得注意的貢獻則被收集在一本名Ⓕ Python Cookbook (O'Reilly & Associates, ISBN 0-596-00797-3.) 的書籍中。

- https://pyvideo.org 從研討會與使用者群組聚會Ⓕ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Ⓕ性代數、傅利葉變Ⓕ、非Ⓕ性求解器、隨機數分Ⓕ、統計分析等一系列的套件。

對於 Python 相關的疑問與問題回報，你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/。

[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ⒻⒻ有貨。

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F替F

有些版本的 Python 直譯器支援目前輸入F容的編輯和歷史記F的替F (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F明文件，在這F我們就不重F了。

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

在直譯器F動的時候，變數和模組名稱的自動完成功能會被自動F用，所以可以用 `Tab` 鍵來叫用自動完成函式；它會查看 Python 的陳述式名稱、目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F儲存在你的使用者資料夾F，一個名F `.python_history` 的檔案中。在下一次F動互動式直譯器時，這些歷史記F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F有被實現：如果能在每次F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F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F的互動式直譯器替代方案，已經存在一段時間，稱F IPython，它具有 Tab 鍵自動完成、物件探索和進階歷史記F管理等特色。它也可以完全客F化F被嵌入到其他應用程式中。另一個類似的增F型互動式環境，稱F bpython。

CHAPTER 15

浮點數運算：問題與限制

在計算機架構中，浮點數 (floating-point number) 是以基數Ⓕ 2（二進位）的小數表示。例如Ⓕ，在**十進位**小數中 `0.625` 可被分Ⓕ 6/10 + 2/100 + 5/1000，同樣的道理，二**進位**小數 `0.101` 可被分Ⓕ 1/2 + 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Ⓕ，你輸入的十進位浮點數只能由實際儲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，你可以將其近似Ⓕ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...
```

只要你停在任何有限的位數，你就只會得到近似值。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，其中分子是從最高有效位元開始，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真正十進位數值。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分數。例如Ⓕ：數字 `0.1` 和 `0.10000000000000001` 和 `0.1000000000000000055511151231257827021181583404541015625`，都由 `3602879701896397 / 2 ** 55` 近似。由於這三個十進位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）現在能選擇其中最短的數Ⓕ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)，更不是你程式碼的錯誤。只要程式語言有支援硬體的浮點數運算，你將會看到同樣的事情出現在其中（雖然有些程式語言預設不會顯示該差Ⓕ，有些甚至是在所有的輸出模式中都不會顯示。）

Ⓕ求更優雅的輸出，你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')  # 給出 12 個有效位數
'3.14159265359'

>>> format(math.pi, '.2f')   # 小數點後給出 2 位數字
'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：

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

雖然數字不會再更接近他們的精准數值，但 `math.isclose()` 函式可以用來比較不精確的值：

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

或者，可以使用 `round()` 函式來比較粗略的近似值：

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

二進位浮點數架構下還有很多這樣的意外。底下的「表示法誤差 (Representation Error)」章節，詳細地解釋了「0.1」的問題。Examples of Floating Point Problems（浮點數問題範例）一文提供了二進位浮點數的作用方式與現實中這類常見問題的摘Ⓕ。如果想要其他常見問題的更完整描述，可以參考 The Perils of Floating Point（浮點數的風險）。

如同上文的末段所述，「這個問題ⒻⒻ有簡單的答案。」不過，也不必對浮點過度擔心！Python 的 float（浮點數）運算中的誤差，是從浮點硬體繼承而來，而在大多數的計算機上，每次運算的誤差範圍不會大於

2**53 分之 1。這對大多數的任務來Ⓕ已經相當足Ⓕ，但你需要記住，它Ⓕ非十進位運算，且每一次 float 運算都可能會承受新的舍入誤差。

雖然浮點運算確實存在一些問題，但在一般情Ⓕ下，如果你只是把最終結果的顯示值，以十進位方式舍入至預期的位數，那Ⓕ仍會得到你預期的結果。`str()` 通常就能滿足要求，而若想要更細Ⓕ的控制，可參Ⓕ formatstrings 中關於 `str.format()` method（方法）的格式規範。

對於需要精准十進位表示法的用例，可以試著用 `decimal` 模組，它可實作適用於會計應用程式與高精確度應用程式的十進位運算。

另一種支援精准運算的格式Ⓕ `fractions` 模組，該模組基於有理數來實作運算（因此可以精確表示像 1/3 這樣的數字）。

如果你是浮點運算的重度使用者，你應該看一下 NumPy 套件，以及由 SciPy 專案提供的許多用於數學和統計學運算的其他套件。請參Ⓕ <https://scipy.org>。

在罕見情Ⓕ下，當你真的想知道一個 float 的精准值，Python 提供的工具可協助達成。`float.as_integer_ratio()` method 可將一個 float 的值表示Ⓕ分數：

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

由於該比率是精准的，它可無損地再現該原始值：

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

`float.hex()` method 以十六進位（基數Ⓕ 16）表示 float，一樣可以給出你的電腦所儲存的精准值：

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

這種精確的十六進位表示法可用於精准地重建 float 值：

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

由於該表示法是精准的，因此適用於在不同版本的 Python 之間可靠地傳送數值（獨立於系統平台），Ⓕ與支援相同格式的其他語言（如 JAVA 和 C99）交Ⓕ資料。

另一個有用的工具是 `sum()` 函式，能在計算總和時幫忙Ⓕ少精確度的損失。當數值被加到運行中的總計值時，它會追Ⓕ「失去的位數 (lost digits)」。這可以明顯改善總體准確度 (overall accuracy)，使得誤差不至於累積到影響最終總計值的程度：

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

`math.fsum()` 更進一步將數值被加到運行總計中的總計時追Ⓕ所有「失去的位數」，以便結果僅有一次舍入。這比 `sum()` 慢，但在不常見的情Ⓕ下會更准確，在這種情Ⓕ下，大量輸入大多相互抵消，最終的總和會接近於零：

```
>>> arr = [-0.10430216751806065, -266310978.67179024, 143401161448607.16,
...        -143401161400469.7, 266262841.31058735, -0.003244936839808227]
>>> float(sum(map(Fraction, arr)))   # 單次四舍五入的精確加總
8.042173697819788e-13
>>> math.fsum(arr)                   # 單次四舍五入
8.042173697819788e-13
>>> sum(arr)                         # 在擴展精度中進行多次舍入
8.042178034628478e-13
>>> total = 0.0
```

(繼續下一頁)

(繼續上一頁)

```
>>> for x in arr:
...     total += x                  # 在標准精度中進行多次舍入
...
>>> total                           # 直接相加Ⓕ有正確的數字
-0.0051575902860057365
```

15.1 表示法誤差 (Representation Error)

本節將會詳細解釋「0.1」的例子，ⒻⒻ明你自己要如何對類似案例執行精准的分析。以下假設你對二進位浮點表示法已有基本程度的熟悉。

Representation error（表示法誤差）是指在真實情Ⓕ中，有些（實際上是大多數）十進位小數Ⓕ不能精准地以二進位（基數Ⓕ 2）小數表示。這是 Python（或 Perl、C、C++、JAVA、Fortran 和其他許多）通常不會顯示你期望的精准十進位數字的主要原因。

Ⓕ什Ⓕ呢？因Ⓕ 1/10 無法精准地以一個二進位小數來表示。至少自從 2000 年以來，幾乎所有的計算機皆使用 IEEE-754 浮點數運算標准，Ⓕ且幾乎所有的平台都以 IEEE 754 binary64 標准中的「雙精度 (double precision)」來作Ⓕ Python 的 float。IEEE 754 binary64 的值包含 53 位元的精度，所以在輸入時，電腦會努力把 0.1 轉Ⓕ到最接近的分數，以 *J*/2**N 的形式表示，此處 *J* 是一個正好包含 53 位元的整數。可以將：

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

重寫Ⓕ：

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

而前面提到 *J* 有精准的 53 位元（即 `>= 2**52` 但 `< 2**53`），所以 *N* 的最佳數值是 56：

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

意即，要使 *J* 正好有 53 位元，則 56 會是 *N* 的唯一值。而 *J* 最有可能的數值就是經過舍入後的該商數：

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

由於余數超過 10 的一半，所以最佳的近似值是透過進位而得：

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

所以，在 IEEE 754 雙精度下，1/10 的最佳近似值是：

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

將分子和分母同除以二，會約分Ⓕ：

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

請注意，由於我們有進位，所以這實際上比 1/10 大了一點；如果我們Ⓕ有進位，商數將會有點小於 1/10。但在任何情Ⓕ下都不可能是精準的 1/10！

所以電腦從來Ⓕ有「看到」1/10：它看到的是上述的精准分數，也就是它能得到的 IEEE 754 double 最佳近似值：

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

如果將該分數乘以 10**55，則可以看到該值以 55 個十進位數字顯示：

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

這表示儲存在電腦中的精准數值等於十進位值 0.1000000000000000055511151231257827021181583404541015625。與其顯示完整的十進位數值，許多語言（包括 Python 的舊版本）選擇將結果舍入至 17 個有效位數：

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

`fractions` 與 `decimal` 模組能使這些計算變得容易：

```
>>> 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

互動式*REPL* 有兩種變體。經典的基本解釋器在所有平台上都支援，具有最低限度的行控制能力（line control capabilities）。

在 Windows 或支援 `curses` 的類似 Unix 系統上，自 Python 3.13 起預設會使用新的互動式 shell。這個 shell 支援Ⓕ色、多行編輯、歷史Ⓕ覽和貼上模式。要停用Ⓕ色，請參Ⓕ using-on-controlling-color 取得詳細資訊。功能鍵提供了一些額外的功能。`F1` 進入互動式幫助Ⓕ覽器 `pydoc`。`F2` 允許Ⓕ覽命令列歷史，既Ⓕ有輸出，也Ⓕ有`»>` 和`...` 的提示。`F3` 進入「貼上模式」，這使得貼上較大的程式碼區塊更容易。按 `F3` 可以返回正常提示。

使用新的互動 shell 時，請輸入 `exit` 或 `quit` 來離開 shell。不需要在這些指令後面加上呼叫括號。

如果不需要新的互動式 shell，可以透過 `PYTHON_BASIC_REPL` 環境變數來停用它。

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 (nonzero exit status) 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，會導致非零退出狀態；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
```

（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。

[1] GNU Readline 套件的一個問題可能會阻止這一點。

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。

16.1.3 互動式Ⓕ動檔案

當你互動式地使用 Python 時，每次Ⓕ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Ⓕ `PYTHONSTARTUP` 的環境變數來實現，該變數是一個包含Ⓕ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Ⓕ本中讀取指令時，此檔案不會被讀取，當 `/dev/tty` 作Ⓕ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。

如果你想從目前目Ⓕ中讀取一個額外的Ⓕ動檔案，你可以在全域Ⓕ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Ⓕ。如果你想在一個Ⓕ本中使用Ⓕ動檔案，你必須在Ⓕ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Ⓕ化模組

Python 提供了兩個Ⓕ子 (hook) 讓你可以將它客Ⓕ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。Ⓕ動 Python Ⓕ運行這段程式碼：

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

現在，你可以在該目Ⓕ中建立一個名Ⓕ `usercustomize.py` 的檔案，Ⓕ將你想要的任何Ⓕ容放入其中。它會影響 Python 的每次呼叫，除非它以 `-s` 選項Ⓕ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Ⓕ下建立，Ⓕ在 usercustomize 之前 import 。更多細節請參Ⓕ `site` 模組的文件。

Ⓕ解

APPENDIX A

術語表

>>>

互動式 shell 的預設 Python 提示字元。常見於能在直譯器中以互動方式被執行的程式碼範例。

...

可以表示：

- 在一個被縮排的程式碼區塊、在一對匹配的左右定界符（delimiter，例如括號、方括號、花括號或三引號）Ⓕ部，或是在指定一個裝飾器 (decorator) 之後，要輸入程式碼時，*互動式* shell 顯示的預設 Python 提示字元。
- The three dots form of the Ellipsis object.

abstract base class（抽象基底類Ⓕ）

抽象基底類Ⓕ（又稱Ⓕ ABC）提供了一種定義介面的方法，作Ⓕ*duck-typing*（鴨子型Ⓕ）的補充。其他類似的技術，像是 `hasattr()`，則顯得笨拙或是帶有細微的錯誤（例如使用魔術方法 (magic method)）。ABC Ⓕ用Ⓕ擬的 subclass（子類Ⓕ），它們Ⓕ不繼承自另一個 class（類Ⓕ），但仍可被 `isinstance()` 及 `issubclass()` 辨識；請參Ⓕ `abc` 模組的Ⓕ明文件。Python 有許多Ⓕ建的 ABC，用於資料結構（在 `collections.abc` 模組）、數字（在 `numbers` 模組）、串流（在 `io` 模組）及 import 尋檢器和載入器（在 `importlib.abc` 模組）。你可以使用 `abc` 模組建立自己的 ABC。

annotate function（Ⓕ釋函式）

A function that can be called to retrieve the *annotations* of an object. This function is accessible as the `__annotate__` attribute of functions, classes, and modules. Annotate functions are a subset of *evaluate functions*.

annotation（Ⓕ釋）

一個與變數、class 屬性、函式的參數或回傳值相關聯的標Ⓕ。照慣例，它被用來作Ⓕ*type hint*（型Ⓕ提示）。

在 runtime 的區域變數Ⓕ釋無法被存取，但全域變數、類Ⓕ屬性和函式的Ⓕ釋，分Ⓕ能Ⓕ以對模組、類Ⓕ和函式呼叫 `annotationlib.get_annotations()` 來取得。

請參Ⓕ*variable annotation*、*function annotation*、**PEP 484**、**PEP 526** 和 **PEP 649**，這些章節皆有此功能的Ⓕ明。關於Ⓕ釋的最佳實踐方法也請參Ⓕ annotations-howto。

argument（引數）

呼叫函式時被傳遞給*function*（或*method*）的值。引數有兩種：

- *關鍵字引數 (keyword argument)*：在函式呼叫中，以識Ⓕ字（identifier，例如 `name=`）開頭的引數，或是以 `**` 後面 dictionary（字典）Ⓕ的值被傳遞的引數。例如，`3` 和 `5` 都是以下 `complex()` 呼叫中的關鍵字引數：

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

- *位置引數 (positional argument)*：不是關鍵字引數的引數。位置引數可在一個引數列表的起始處出現，和（或）作Ⓕ `*` 之後的*iterable*（可Ⓕ代物件）中的元素被傳遞。例如，`3` 和 `5` 都是以下呼叫中的位置引數：

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

引數會被指定給函式主體中的附名區域變數。關於支配這個指定過程的規則，請參Ⓕ calls 章節。在語法上，任何運算式都可以被用來表示一個引數；其評估值會被指定給區域變數。

另請參Ⓕ術語表的*parameter*（參數）條目、常見問題中的引數和參數之間的差Ⓕ，以及 **PEP 362**。

asynchronous context manager（非同步情境管理器）
一個可以控制 `async with` 陳述式中所見環境的物件，而它是透過定義 `__aenter__()` 和 `__aexit__()` method（方法）來控制的。由 **PEP 492** 引入。

asynchronous generator（非同步Ⓕ生器）
一個會回傳*asynchronous generator iterator*（非同步Ⓕ生器Ⓕ代器）的函式。它看起來像一個以 `async def` 定義的協程函式 (coroutine function)，但不同的是它包含了 `yield` 運算式，能生成一系列可用於 `async for` Ⓕ圈的值。

這個術語通常用來表示一個非同步Ⓕ生器函式，但在某些情境中，也可能是表示非同步Ⓕ生器Ⓕ代器 *(asynchronous generator iterator)*。萬一想表達的意思不Ⓕ清楚，那就使用完整的術語，以避免歧義。

一個非同步Ⓕ生器函式可能包含 `await` 運算式，以及 `async for` 和 `async with` 陳述式。

asynchronous generator iterator（非同步Ⓕ生器Ⓕ代器）
一個由*asynchronous generator*（非同步Ⓕ生器）函式所建立的物件。

這是一個*asynchronous iterator*（非同步Ⓕ代器），當它以 `__anext__()` method 被呼叫時，會回傳一個可等待物件 (awaitable object)，該物件將執行非同步Ⓕ生器的函式主體，直到遇到下一個 `yield` 運算式。

每個 `yield` 會暫停處理程序，Ⓕ記住執行狀態（包括區域變數及擱置中的 try 陳述式）。當非同步Ⓕ生器Ⓕ代器以另一個被 `__anext__()` 回傳的可等待物件有效地回復時，它會從停止的地方繼續執行。請參Ⓕ **PEP 492** 和 **PEP 525**。

asynchronous iterable（非同步可Ⓕ代物件）
一個物件，它可以在 `async for` 陳述式中被使用。必須從它的 `__aiter__()` method 回傳一個*asynchronous iterator*（非同步Ⓕ代器）。由 **PEP 492** 引入。

asynchronous iterator（非同步Ⓕ代器）
一個實作 `__aiter__()` 和 `__anext__()` method 的物件。`__anext__()` 必須回傳一個*awaitable*（可等待物件）。`async for` 會解析非同步Ⓕ代器的 `__anext__()` method 所回傳的可等待物件，直到它引發 `StopAsyncIteration` 例外。由 **PEP 492** 引入。

attached thread state
A *thread state* that is active for the current OS thread.

When a *thread state* is attached, the OS thread has access to the full Python C API and can safely invoke the bytecode interpreter.

Unless a function explicitly notes otherwise, attempting to call the C API without an attached thread state will result in a fatal error or undefined behavior. A thread state can be attached and detached explicitly by the user through the C API, or implicitly by the runtime, including during blocking C calls and by the bytecode interpreter in between calls.

On most builds of Python, having an attached thread state implies that the caller holds the *GIL* for the current interpreter, so only one OS thread can have an attached thread state at a given moment. In *free-threaded* builds of Python, threads can concurrently hold an attached thread state, allowing for true parallelism of the bytecode interpreter.

attribute（屬性）
一個與某物件相關聯的值，該值大多能透過使用點分隔運算式 (dotted expression) 的名稱被參照。例如，如果物件 *o* 有一個屬性 *a*，則該屬性能以 *o.a* 被參照。

如果一個物件允許，給予該物件一個名稱不是由 identifiers 所定義之識Ⓕ符 (identifier) 的屬性是有可能的，例如使用 `setattr()`。像這樣的屬性將無法使用點分隔運算式來存取，而是需要使用 `getattr()` 來取得它。

awaitable（可等待物件）
一個可以在 `await` 運算式中被使用的物件。它可以是一個*coroutine*（協程），或是一個有 `__await__()` method 的物件。另請參Ⓕ **PEP 492**。

BDFL
Benevolent Dictator For Life（終身仁慈獨裁者），又名 Guido van Rossum，Python 的創造者。

binary file（二進位檔案）
一個能Ⓕ讀取和寫入*bytes-like objects*（類位元組串物件）的*file object*（檔案物件）。二進位檔案的例子有：以二進位模式（`'rb'`、`'wb'` 或 `'rb+'`）開Ⓕ的檔案、`sys.stdin.buffer`、`sys.stdout.buffer`，以及 `io.BytesIO` 和 `gzip.GzipFile` 實例。

另請參Ⓕ*text file*（文字檔案），它是一個能Ⓕ讀取和寫入 `str` 物件的檔案物件。

borrowed reference（借用參照）
在 Python 的 C API 中，借用參照是一個對物件的參照，其中使用該物件的程式碼Ⓕ不擁有這個參照。如果該物件被銷Ⓕ，它會成Ⓕ一個迷途指標 (dangling pointer)。例如，一次垃圾回收 (garbage collection) 可以移除對物件的最後一個*strong reference*（Ⓕ參照），而將該物件銷Ⓕ。

對*borrowed reference* 呼叫 `Py_INCREF()` 以將它原地 (in-place) 轉ⒻⒻ*strong reference* 是被建議的做法，除非該物件不能在最後一次使用借用參照之前被銷Ⓕ。`Py_NewRef()` 函式可用於建立一個新的*strong reference*。

bytes-like object（類位元組串物件）
一個支援 bufferobjects 且能Ⓕ匯出 C-*contiguous* 緩衝區的物件。這包括所有的 `bytes`、`bytearray` 和 `array.array` 物件，以及許多常見的 `memoryview` 物件。類位元組串物件可用於處理二進位資料的各種運算；這些運算包括壓縮、儲存至二進位檔案和透過 socket（插座）發送。

有些運算需要二進位資料是可變的。Ⓕ明文件通常會將這些物件稱Ⓕ「可讀寫的類位元組串物件」。可變緩衝區的物件包括 `bytearray`，以及 `bytearray` 的 `memoryview`。其他的運算需要讓二進位資料被儲存在不可變物件（「唯讀的類位元組串物件」）中；這些物件包括 `bytes`，以及 `bytes` 物件的 `memoryview`。

bytecode（位元組碼）
Python 的原始碼會被編譯成位元組碼，它是 Python 程式在 CPython 直譯器中的Ⓕ部表示法。該位元組碼也會被暫存在 `.pyc` 檔案中，以便第二次執行同一個檔案時能Ⓕ更快速（可以不用從原始碼重新編譯Ⓕ位元組碼）。這種「中間語言 (intermediate language)」據Ⓕ是運行在一個*virtual machine*（Ⓕ擬機器）上，該Ⓕ擬機器會執行與每個位元組碼對應的機器碼 (machine code)。要注意的是，位元組碼理論上是無法在不同的 Python Ⓕ擬機器之間運作的，也不能在不同版本的 Python 之間保持穩定。

位元組碼的指令列表可以在 dis 模組的Ⓕ明文件中找到。

callable（可呼叫物件）
一個 callable 是可以被呼叫的物件，呼叫時可能以下列形式帶有一組引數（請見*argument*）：

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

一個*function* 與其延伸的*method* 都是 callable。一個有實作 `__call__()` 方法的 class 之實例也是個 callable。

callback（回呼）
作Ⓕ引數被傳遞的一個副程式 (subroutine) 函式，會在未來的某個時間點被執行。

class（類Ⓕ）
一個用於建立使用者定義物件的模板。Class 的定義通常會包含 method 的定義，這些 method 可以在 class 的實例上進行操作。

class variable（類Ⓕ變數）
一個在 class 中被定義，且應該只能在 class 層次（意即不是在 class 的實例中）被修改的變數。

closure variable（閉包變數）
從外部作用域中定義且從巢狀作用域參照的自由變數，不是於 runtime 從全域或Ⓕ建命名空間解析。可以使用 `nonlocal` 關鍵字明確定義以允許寫入存取，或者如果僅需讀取變數則隱式定義即可。

例如在下面程式碼中的 `inner` 函式中，`x` 和 `print` 都是自由變數，但只有 `x` 是閉包變數：

```
def outer():
    x = 0
    def inner():
        nonlocal x
        x += 1
        print(x)
    return inner
```

由於 `codeobject.co_freevars` 屬性（Ⓕ管名稱如此，但它僅包含閉包變數的名稱，而不是列出所有參照的自由變數），當預期含義是特指閉包變數時，有時候甚至也會使用更通用的自由變數一詞。

complex number（Ⓕ數）
一個我們熟悉的實數系統的擴充，在此所有數字都會被表示Ⓕ一個實部和一個Ⓕ部之和。Ⓕ數就是Ⓕ數單位（`-1` 的平方根）的實數倍，此單位通常在數學中被寫Ⓕ `i`，在工程學中被寫Ⓕ `j`。Python Ⓕ建了對Ⓕ數的支援，它是用後者的記法來表示Ⓕ數；Ⓕ部會帶著一個後綴的 `j` 被編寫，例如 `3+1j`。若要將 `math` 模組Ⓕ的工具等效地用於Ⓕ數，請使用 `cmath` 模組。Ⓕ數的使用是一個相當進階的數學功能。如果你Ⓕ有察覺到對它們的需求，那Ⓕ幾乎能確定你可以安全地忽略它們。

context（情境）
This term has different meanings depending on where and how it is used. Some common meanings:

- The temporary state or environment established by a *context manager* via a `with` statement.
- The collection of keyvalue bindings associated with a particular `contextvars.Context` object and accessed via `ContextVar` objects. Also see *context variable*.
- 一個 `contextvars.Context` 物件。另請參Ⓕ*current context*。

context management protocol（情境管理協定）
由 `with` 陳述式所呼叫的 `__enter__()` 和 `__exit__()` 方法。另請參Ⓕ **PEP 343**。

context manager（情境管理器）
An object which implements the *context management protocol* and controls the environment seen in a `with` statement. See **PEP 343**.

context variable（情境變數）
A variable whose value depends on which context is the *current context*. Values are accessed via `contextvars.ContextVar` objects. Context variables are primarily used to isolate state between concurrent asynchronous tasks.

contiguous（連續的）
如果一個緩衝區是 *C-contiguous* 或是 *Fortran contiguous*，則它會確切地被視Ⓕ是連續的。零維 (zero-dimensional) 的緩衝區都是 C 及 Fortran contiguous。在一維 (one-dimensional) 陣列中，各項目必須在記憶體中彼此相鄰地排列，而其索引順序是從零開始遞增。在多維的 (multidimensional) C-contiguous 陣列中，按記憶體位址的順序Ⓕ覽各個項目時，最後一個索引的變化最快。然而，在 Fortran contiguous 陣列中，第一個索引的變化最快。

coroutine（協程）
協程是副程式 (subroutine) 的一種更Ⓕ廣義的形式。副程式是在某個時間點被進入Ⓕ在另一個時間點被退出。協程可以在許多不同的時間點被進入、退出和回復。它們能Ⓕ以 `async def` 陳述式被實作。另請參Ⓕ **PEP 492**。

coroutine function（協程函式）
一個回傳*coroutine*（協程）物件的函式。一個協程函式能以 `async def` 陳述式被定義，Ⓕ可能會包含 `await`、`async for` 和 `async with` 關鍵字。這些關鍵字由 **PEP 492** 引入。

CPython
Python 程式語言的標准實作 (canonical implementation)，被發布在 python.org 上。「CPython」這個術語在必要時被使用，以區分此實作與其它語言的實作，例如 Jython 或 IronPython。

current context
The *context* (`contextvars.Context` object) that is currently used by `ContextVar` objects to access (get or set) the values of *context variables*. Each thread has its own current context. Frameworks for executing asynchronous tasks (see `asyncio`) associate each task with a context which becomes the current context whenever the task starts or resumes execution.

cyclic isolate
A subgroup of one or more objects that reference each other in a reference cycle, but are not referenced by objects outside the group. The goal of the *cyclic garbage collector* is to identify these groups and break the reference cycles so that the memory can be reclaimed.

decorator（裝飾器）
一個函式，它會回傳另一個函式，通常它會使用 `@wrapper` 語法，被應用Ⓕ一種函式的變Ⓕ (function transformation)。裝飾器的常見範例是 `classmethod()` 和 `staticmethod()`。

裝飾器語法只是語法糖。以下兩個函式定義在語義上是等效的：

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

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

Class 也存在相同的概念，但在那Ⓕ比較不常用。關於裝飾器的更多Ⓕ容，請參Ⓕ函式定義和 class 定義的Ⓕ明文件。

descriptor（描述器）
任何定義了 `__get__()`、`__set__()` 或 `__delete__()` method 的物件。當一個 class 屬性是一個描述器時，它的特殊連結行Ⓕ會在屬性查找時被觸發。通常，使用 *a.b* 來取得、設定或Ⓕ除某個屬性時，會在 *a* 的 class 字典中查找名稱Ⓕ *b* 的物件，但如果 *b* 是一個描述器，則相對應的描述器 method 會被呼叫。對描述器的理解是深入理解 Python 的關鍵，因Ⓕ它們是許多功能的基礎，這些功能包括函式、method、屬性 (property)、class method、Ⓕ態 method，以及對 super class（父類Ⓕ）的參照。

關於描述器 method 的更多資訊，請參Ⓕ descriptors 或描述器使用指南。

dictionary（字典）
一個關聯陣列 (associative array)，其中任意的鍵會被對映到值。鍵可以是任何帶有 `__hash__()` 和 `__eq__()` method 的物件。在 Perl 中被稱Ⓕ雜Ⓕ (hash)。

dictionary comprehension（字典綜合運算）
一種緊密的方法，用來處理一個可Ⓕ代物件中的全部或部分元素，Ⓕ將處理結果以一個字典回傳。`results = {n: n ** 2 for n in range(10)}` 會Ⓕ生一個字典，它包含了鍵 `n` 對映到值 `n ** 2`。請參Ⓕ comprehensions。

dictionary view（字典檢視）
從 `dict.keys()`、`dict.values()` 及 `dict.items()` 回傳的物件被稱Ⓕ字典檢視。它們提供了字典中項目的動態檢視，這表示當字典有變動時，該檢視會反映這些變動。若要Ⓕ制將字典檢視轉Ⓕ完整的 list（串列），須使用 `list(dictview)`。請參Ⓕ dict-views。

docstring（Ⓕ明字串）
一個在 class、函式或模組中，作Ⓕ第一個運算式出現的字串文本。雖然它在套件執行時會被忽略，但它會被編譯器辨識，Ⓕ被放入所屬 class、函式或模組的 `__doc__` 屬性中。由於Ⓕ明字串可以透過Ⓕ省 (introspection) 來Ⓕ覽，因此它是物件的Ⓕ明文件存放的標准位置。

duck-typing（鴨子型Ⓕ）
一種程式設計風格，它不是藉由檢查一個物件的型Ⓕ來確定它是否具有正確的介面；取而代之的是，method 或屬性會單純地被呼叫或使用。（「如果它看起來像一Ⓕ鴨子而且叫起來像一Ⓕ鴨子，那Ⓕ它一定是一Ⓕ鴨子。」）因ⒻⒻ調介面而非特定型Ⓕ，精心設計的程式碼能讓多形替代 (polymorphic

substitution) 來增進它的靈活性。鴨子型Ⓕ要避免使用 `type()` 或 `isinstance()` 進行測試。(但是請注意，鴨子型Ⓕ可以用*抽象基底類*Ⓕ *(abstract base class)* 來補充。) 然而，它通常會Ⓕ用 `hasattr()` 測試，或是*EAFP* 程式設計風格。

dunder（雙底Ⓕ）
一個非正式的縮寫，代表「雙底Ⓕ (double underscore)」，當談論到*特殊方法*時使用。例如，`__init__` 通常被叫做"dunder init"。

EAFP
Easier to ask for forgiveness than permission.（請求寬恕比請求許可更容易。）這種常見的 Python 編碼風格會先假設有效的鍵或屬性的存在，Ⓕ在該假設被推翻時再捕獲例外。這種乾Ⓕ且快速的風格，其特色是存在許多的 `try` 和 `except` 陳述式。該技術與許多其他語言（例如 C）常見的*LBYL* 風格形成了對比。

evaluate function（求值函式）
A function that can be called to evaluate a lazily evaluated attribute of an object, such as the value of type aliases created with the `type` statement.

expression（運算式）
一段可以被評估Ⓕ求值的語法。Ⓕ句話Ⓕ，一個運算式就是文字、名稱、屬性存取、運算子或函式呼叫等運算式元件的累積，而這些元件都能回傳一個值。與許多其他語言不同的是，Ⓕ非所有的 Python 語言構造都是運算式。另外有一些*statement*（陳述式）不能被用作運算式，例如 `while`。賦值 (assignment) 也是陳述式，而不是運算式。

extension module（擴充模組）
一個以 C 或 C++ 編寫的模組，它使用 Python 的 C API 來與核心及使用者程式碼進行互動。

f-string（f 字串）
f-strings（f 字串）
以 `f` 或 `F` Ⓕ前綴的字串文本通常被稱Ⓕ「f 字串」，它是格式化的字串文本的縮寫。另請參Ⓕ **PEP 498**。

file object（檔案物件）
一個讓使用者透過檔案導向 (file-oriented) API（如 `read()` 或 `write()` 等 method）來操作底層資源的物件。根據檔案物件被建立的方式，它能Ⓕ協調對真實磁碟檔案或是其他類型的儲存器或通訊裝置（例如標準輸入 / 輸出、記憶體Ⓕ緩衝區、socket（插座）、管Ⓕ (pipe) 等）的存取。檔案物件也被稱Ⓕ*類檔案物件 (file-like object)* 或*串流 (stream)*。

實際上，有三種檔案物件：原始的*二進位檔案*、緩衝的*二進位檔案*和*文字檔案*。它們的介面在 `io` 模組中被定義。建立檔案物件的標準方法是使用 `open()` 函式。

file-like object（類檔案物件）
file object（檔案物件）的同義字。

filesystem encoding and error handler（檔案系統編碼和錯誤處理函式）
Python 所使用的一種編碼和錯誤處理函式，用來解碼來自作業系統的位元組，以及將 Unicode 編碼到作業系統。

檔案系統編碼必須保證能成功解碼所有小於 128 的位元組。如果檔案系統編碼無法提供此保證，則 API 函式會引發 `UnicodeError`。

`sys.getfilesystemencoding()` 和 `sys.getfilesystemencodeerrors()` 函式可用於取得檔案系統編碼和錯誤處理函式。

filesystem encoding and error handler（檔案系統編碼和錯誤處理函式）會在 Python Ⓕ動時由 `PyConfig_Read()` 函式來配置：請參Ⓕ `filesystem_encoding`，以及 `PyConfig` 的成員 `filesystem_errors`。

另請參Ⓕ*locale encoding*（區域編碼）。

finder（尋檢器）
一個物件，它會嘗試Ⓕ正在被 import 的模組尋找*loader*（載入器）。

有兩種類型的尋檢器：*元路徑尋檢器 (meta path finder)* 會使用 `sys.meta_path`，而*路徑項目尋檢器 (path entry finder)* 會使用 `sys.path_hooks`。

請參Ⓕ finders-and-loaders 和 `importlib` 以了解更多細節。

floor division（向下取整除法）
向下無條件舍去到最接近整數的數學除法。向下取整除法的運算子是 `//`。例如，運算式 `11 // 4` 的計算結果Ⓕ `2`，與 float（浮點數）真除法所回傳的 `2.75` 不同。請注意，`(-11) // 4` 的結果是 `-3`，因Ⓕ是 `-2.75` 被向下無條件舍去。請參Ⓕ **PEP 238**。

free threading（自由執行緒）
Ⓕ一種執行緒模型，多個執行緒可以在同一直譯器中同時運行 Python 位元組碼。這與全域直譯器鎖形成對比，後者一次只允許一個執行緒執行 Python 位元組碼。請參Ⓕ **PEP 703**。

free variable（自由變數）
Formally, as defined in the language execution model, a free variable is any variable used in a namespace which is not a local variable in that namespace. See *closure variable* for an example. Pragmatically, due to the name of the `codeobject.co_freevars` attribute, the term is also sometimes used as a synonym for *closure variable*.

function（函式）
一連串的陳述式，它能Ⓕ向呼叫者回傳一些值。它也可以被傳遞零個或多個引數，這些引數可被使用於函式本體的執行。另請參Ⓕ*parameter*（參數）、*method*（方法），以及 function 章節。

function annotation（函式Ⓕ釋）
函式參數或回傳值的一個*annotation*（Ⓕ釋）。

函式Ⓕ釋通常被使用於型Ⓕ提示：例如，這個函式預期會得到兩個 `int` 引數，Ⓕ會有一個 `int` 回傳值：

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

函式Ⓕ釋的語法在 function 章節有詳細解釋。

請參Ⓕ*variable annotation* 和 **PEP 484**，皆有此功能的描述。關於Ⓕ釋的最佳實踐方法，另請參Ⓕ annotations-howto。

__future__
future 陳述式：`from __future__ import <feature>`，會指示編譯器使用那些在 Python 未來的發布版本中將成Ⓕ標准的語法或語義，來編譯目前的模組。而 `__future__` 模組則記Ⓕ了 *feature*（功能）可能的值。透過 import 此模組Ⓕ對其變數求值，你可以看見一個新的功能是何時首次被新增到此語言中，以及它何時將會（或已經）成Ⓕ預設的功能：

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

garbage collection（垃圾回收）
當記憶體不再被使用時，將其釋放的過程。Python 執行垃圾回收，是透過參照計數 (reference counting)，以及一個能Ⓕ檢測和中斷參照循環 (reference cycle) 的循環垃圾回收器 (cyclic garbage collector) 來完成。垃圾回收器可以使用 `gc` 模組對其進行控制。

generator（Ⓕ生器）
一個會回傳*generator iterator*（Ⓕ生器Ⓕ代器）的函式。它看起來像一個正常的函式，但不同的是它包含了 `yield` 運算式，能Ⓕ生一系列的值，這些值可用於 for Ⓕ圈，或是以 `next()` 函式，每次檢索其中的一個值。

這個術語通常用來表示一個Ⓕ生器函式，但在某些情境中，也可能是表示Ⓕ生器Ⓕ代器。萬一想表達的意思不Ⓕ清楚，那就使用完整的術語，以避免歧義。

generator iterator（Ⓕ生器Ⓕ代器）
一個由*generator*（Ⓕ生器）函式所建立的物件。

每個 `yield` 會暫停處理程序，Ⓕ記住執行狀態（包括區域變數及擱置中的 try 陳述式）。當Ⓕ生器Ⓕ代器回復時，它會從停止的地方繼續執行（與那些每次呼叫時都要重新開始的函式有所不同）。

generator expression（Ⓕ生器運算式）

一個會回傳Ⓕ代器的運算式。它看起來像一個正常的運算式，後面接著一個 `for` 子句，該子句定義了Ⓕ圈變數、範圍以及一個選擇性的 `if` 子句。該組合運算式會Ⓕ外層函式Ⓕ生多個值：

```
>>> sum(i*i for i in range(10))         # 平方之和 0, 1, 4, ... 81
285
```

generic function（泛型函式）

一個由多個函式組成的函式，該函式會對不同的型Ⓕ實作相同的運算。呼叫期間應該使用哪種實作，是由調度演算法 (dispatch algorithm) 來Ⓕ定。

另請參Ⓕ*single dispatch*（單一調度）術語表條目、`functools.singledispatch()` 裝飾器和 **PEP 443**。

generic type（泛型型Ⓕ）

一個能Ⓕ被參數化 (parameterized) 的*type*（型Ⓕ）；通常是一個 容器型Ⓕ，像是 `list` 和 `dict`。它被用於型Ⓕ提示和Ⓕ釋。

詳情請參Ⓕ泛型Ⓕ名型Ⓕ、**PEP 483**、**PEP 484**、**PEP 585** 和 `typing` 模組。

GIL

請參Ⓕ*global interpreter lock*（全域直譯器鎖）。

global interpreter lock（全域直譯器鎖）

CPython 直譯器所使用的機制，用以確保每次都只有一個執行緒能執行 Python 的*bytecode*（位元組碼）。透過使物件模型（包括關鍵的Ⓕ建型Ⓕ，如 `dict`）自動地避免Ⓕ行存取 (concurrent access) 的危險，此機制可以簡化 CPython 的實作。鎖定整個直譯器，會使直譯器更容易成Ⓕ多執行緒 (multi-threaded)，但代價是會犧牲掉多處理器的機器能Ⓕ提供的一大部分平行性 (parallelism)。

然而，有些擴充模組，無論是標準的或是第三方的，它們被設計成在執行壓縮或雜Ⓕ等計算密集 (computationally intensive) 的任務時，可以解除 GIL。另外，在執行 I/O 時，GIL 總是會被解除。

從 Python 3.13 開始可以使用 `--disable-gil` 建置設定來停用 GIL。使用此選項建立 Python 後，必須使用 `-X gil=0` 來執行程式碼，或者設定 `PYTHON_GIL=0` 環境變數後再執行程式碼。此功能可以提高多執行緒應用程式的效能，Ⓕ使多核心 CPU 的高效使用變得更加容易。有關更多詳細資訊，請參Ⓕ **PEP 703**。

In prior versions of Python's C API, a function might declare that it requires the GIL to be held in order to use it. This refers to having an *attached thread state*.

hash-based pyc（雜Ⓕ架構的 pyc）

一個位元組碼 (bytecode) 暫存檔，它使用雜Ⓕ值而不是對應原始檔案的最後修改時間，來確定其有效性。請參Ⓕ pyc-invalidation。

hashable（可雜Ⓕ的）

如果一個物件有一個雜Ⓕ值，該值在其生命Ⓕ期中永不改變（它需要一個 `__hash__()` method），且可與其他物件互相比較（它需要一個 `__eq__()` method），那Ⓕ它就是一個可雜Ⓕ物件。比較結果Ⓕ相等的多個可雜Ⓕ物件，它們必須擁有相同的雜Ⓕ值。

可雜Ⓕ性 (hashability) 使一個物件可用作 dictionary（字典）的鍵和 set（集合）的成員，因Ⓕ這些資料結構都在其Ⓕ部使用了雜Ⓕ值。

大多數的 Python 不可變Ⓕ建物件都是可雜Ⓕ的；可變的容器（例如 list 或 dictionary）Ⓕ不是；而不可變的容器（例如 tuple（元組）和 frozenset），只有當它們的元素是可雜Ⓕ的，它們本身才是可雜Ⓕ的。若物件是使用者自定 class 的實例，則這些物件會被預設Ⓕ可雜Ⓕ的。它們在互相比較時都是不相等的（除非它們與自己比較），而它們的雜Ⓕ值則是衍生自它們的 `id()`。

IDLE

Python 的 Integrated Development and Learning Environment（整合開發與學習環境）。idle 是一個基本的編輯器和直譯器環境，它和 Python 的標準發行版本一起被提供。

immortal（不滅）

不滅物件 (Immortal objects) 是 **PEP 683** 引入的 CPython 實作細節。

如果一個物件是不滅的，它的參照計數永遠不會被修改，因此在直譯器運行時它永遠不會被釋放。例如，`True` 和 `None` 在 CPython 中是不滅的。

Immortal objects can be identified via `sys._is_immortal()`, or via `PyUnstable_IsImmortal()` in the C API.

immutable（不可變物件）
一個具有固定值的物件。不可變物件包括數字、字串和 tuple（元組）。這類物件是不能被改變的。如果一個不同的值必須被儲存，則必須建立一個新的物件。它們在需要Ⓕ定雜Ⓕ值的地方，扮演重要的角色，例如 dictionary（字典）中的一個鍵。

import path（引入路徑）
一個位置（或路徑項目）的列表，而那些位置就是在 import 模組時，會被*path based finder*（基於路徑的尋檢器）搜尋模組的位置。在 import 期間，此位置列表通常是來自 `sys.path`，但對於子套件 (subpackage) 而言，它也可能是來自父套件的 `__path__` 屬性。

importing（引入）
一個過程。一個模組中的 Python 程式碼可以透過此過程，被另一個模組中的 Python 程式碼使用。

importer（引入器）
一個能Ⓕ尋找及載入模組的物件；它既是*finder*（尋檢器）也是*loader*（載入器）物件。

interactive（互動的）
Python 有一個互動式直譯器，這表示你可以在直譯器的提示字元輸入陳述式和運算式，立即執行它們Ⓕ且看到它們的結果。只要Ⓕ動 `python`，不需要任何引數（可能藉由從你的電腦的主選單選擇它）。這是測試新想法或檢查模組和包的非常Ⓕ大的方法（請記住 help(x)）。更多互動式模式相關資訊請見互動模式。

interpreted（直譯的）
Python 是一種直譯語言，而不是編譯語言，不過這個區分可能有些模糊，因Ⓕ有位元組碼 (bytecode) 編譯器的存在。這表示原始檔案可以直接被運行，而不需明確地建立另一個執行檔，然後再執行它。直譯語言通常比編譯語言有更短的開發 / 除錯Ⓕ期，不過它們的程式通常也運行得較慢。另請參Ⓕ*interactive*（互動的）。

interpreter shutdown（直譯器關閉）
當 Python 直譯器被要求關閉時，它會進入一個特殊階段，在此它逐漸釋放所有被配置的資源，例如模組和各種關鍵Ⓕ部結構。它也會多次呼叫垃圾回收器 *(garbage collector)*。這能Ⓕ觸發使用者自定的解構函式 (destructor) 或弱引用的回呼 (weakref callback)，Ⓕ執行其中的程式碼。在關閉階段被執行的程式碼會遇到各種例外，因Ⓕ它所依賴的資源可能不再有作用了（常見的例子是函式庫模組或是警告機制）。

直譯器關閉的主要原因，是 `__main__` 模組或正被運行的Ⓕ本已經執行完成。

iterable（可Ⓕ代物件）
一種能Ⓕ一次回傳一個其中成員的物件。可Ⓕ代物件的例子包括所有的序列型Ⓕ（像是 `list`、`str` 和 `tuple`）和某些非序列型Ⓕ，像是 `dict`、檔案物件，以及你所定義的任何 class 物件，只要那些 class 有實作*sequence*（序列）語意的 `__iter__()` 或是 `__getitem__()` method，該物件就是可Ⓕ代物件。

可Ⓕ代物件可用於 `for` Ⓕ圈和許多其他需要一個序列的地方 (`zip()`、`map()`...)。當一個可Ⓕ代物件作Ⓕ引數被傳遞給Ⓕ建函式 `iter()` 時，它會Ⓕ該物件回傳一個Ⓕ代器。此Ⓕ代器適用於針對一組值進行一遍 (one pass) 運算。使用Ⓕ代器時，通常不一定要呼叫 `iter()` 或自行處理Ⓕ代器物件。`for` 陳述式會自動地Ⓕ你處理這些事，它會建立一個暫時性的未命名變數，用於在Ⓕ圈期間保有該Ⓕ代器。另請參Ⓕ*iterator*（Ⓕ代器）、*sequence*（序列）和*generator*（Ⓕ生器）。

iterator（Ⓕ代器）
一個表示資料流的物件。重Ⓕ地呼叫Ⓕ代器的 `__next__()` method（或是將它傳遞給Ⓕ建函式 `next()`）會依序回傳資料流中的各項目。當不再有資料時，則會引發 `StopIteration` 例外。此時，該Ⓕ代器物件已被用盡，而任何對其 `__next__()` method 的進一步呼叫，都只會再次引發 `StopIteration`。Ⓕ代器必須有一個 `__iter__()` method，它會回傳Ⓕ代器物件本身，所以每個Ⓕ代器也都是可Ⓕ代物件，且可以用於大多數適用其他可Ⓕ代物件的場合。一個明顯的例外，是嘗試多遍Ⓕ代 (multiple iteration passes) 的程式碼。一個容器物件（像是 `list`）在每次你將它傳遞給 `iter()` 函式或在 `for` Ⓕ圈中使用它時，都會Ⓕ生一個全新的Ⓕ代器。使用Ⓕ代器嘗試此事（多遍Ⓕ代）時，只會回傳在前一遍Ⓕ代中被用過的、同一個已被用盡的Ⓕ代器物件，使其看起來就像一個空的容器。

在 typeiter 文中可以找到更多資訊。

CPython Ⓕ不是始終如一地都會檢查「Ⓕ代器有定義 `__iter__()`」這個規定。另請注意，free-threading（自由執行緒）CPython 不保證Ⓕ代器操作的執行緒安全。

key function（鍵函式）
鍵函式或理序函式 (collation function) 是一個可呼叫 (callable) 函式，它會回傳一個用於排序 (sorting) 或定序 (ordering) 的值。例如，`locale.strxfrm()` 被用來Ⓕ生一個了解區域特定排序慣例的排序鍵。

Python 中的許多工具，都接受以鍵函式來控制元素被定序或分組的方式。它們包括 `min()`、`max()`、`sorted()`、`list.sort()`、`heapq.merge()`、`heapq.nsmallest()`、`heapq.nlargest()` 和 `itertools.groupby()`。

有幾種方法可以建立一個鍵函式。例如，`str.lower()` method 可以作Ⓕ不分大小寫排序的鍵函式。或者，一個鍵函式也可以從 `lambda` 運算式被建造，例如 `lambda r: (r[0], r[2])`。另外，`operator.attrgetter()`、`operator.itemgetter()` 和 `operator.methodcaller()` Ⓕ三個鍵函式的建構函式 (constructor)。關於如何建立和使用鍵函式的範例，請參Ⓕ如何排序。

keyword argument（關鍵字引數）
請參Ⓕ*argument*（引數）。

lambda
由單一*expression*（運算式）所組成的一個匿名行Ⓕ函式 (inline function)，於該函式被呼叫時求值。建立 lambda 函式的語法是 `lambda [parameters]: expression`

LBYL
Look before you leap.（三思而後行。）這種編碼風格會在進行呼叫或查找之前，明確地測試先Ⓕ條件。這種風格與*EAFP* 方式形成對比，且它的特色是會有許多 `if` 陳述式的存在。

在一個多執行緒環境中，LBYL 方式有在「三思」和「後行」之間引入了競Ⓕ條件 (race condition) 的風險。例如以下程式碼 `if key in mapping: return mapping[key]`，如果另一個執行緒在測試之後但在查找之前，從 *mapping* 中移除了 *key*，則該程式碼就會失效。這個問題可以用鎖 (lock) 或使用 EAFP 編碼方式來解Ⓕ。

lexical analyzer（詞法分析器）
tokenizer 的正式名稱；請參Ⓕ*token*。

list（串列）
一個 Python Ⓕ建的*sequence*（序列）。Ⓕ管它的名字是 list，它其實更類似其他語言中的一個陣列 (array) 而較不像一個鏈結串列 (linked list)，因Ⓕ存取元素的時間Ⓕ雜度是 *O*(1)。

list comprehension（串列綜合運算）
一種用來處理一個序列中的全部或部分元素，Ⓕ將處理結果以一個 list 回傳的簡要方法。`result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` 會Ⓕ生一個字串 list，其中包含 0 到 255 範圍Ⓕ，所有偶數的十六進位數 (0x..)。`if` 子句是選擇性的。如果省略它，則 `range(256)` 中的所有元素都會被處理。

loader（載入器）
一個能Ⓕ載入模組的物件。它必須定義 `exec_module()` 和 `create_module()` 方法以實作 `Loader` 介面。載入器通常是被*finder*（尋檢器）回傳。更多細節請參Ⓕ：

- finders-and-loaders
- `importlib.abc.Loader`
- **PEP 302**

locale encoding（區域編碼）
在 Unix 上，它是 LC_CTYPE 區域設定的編碼。它可以用 `locale.setlocale(locale.LC_CTYPE, new_locale)` 來設定。

在 Windows 上，它是 ANSI 碼頁（code page，例如 `"cp1252"`）。

在 Android 和 VxWorks 上，Python 使用 `"utf-8"` 作Ⓕ區域編碼。

`locale.getencoding()` 可以用來取得區域編碼。

也請參考*filesystem encoding and error handler*。

magic method（魔術方法）
special method（特殊方法）的一個非正式同義詞。

mapping（對映）
一個容器物件，它支援任意鍵的查找，且能實作 abstract base classes（抽象基底類Ⓕ）中，`collections.abc.Mapping` 或 `collections.abc.MutableMapping` 所指定的 method。範例包括 `dict`、`collections.defaultdict`、`collections.OrderedDict` 和 `collections.Counter`。

meta path finder（元路徑尋檢器）
一種經由搜尋 `sys.meta_path` 而回傳的*finder*（尋檢器）。元路徑尋檢器與*路徑項目尋檢器 (path entry finder)* 相關但是不同。

關於元路徑尋檢器實作的 method，請參Ⓕ `importlib.abc.MetaPathFinder`。

metaclass（元類Ⓕ）
一種 class 的 class。Class 定義過程會建立一個 class 名稱、一個 class dictionary（字典），以及一個 base class（基底類Ⓕ）的列表。Metaclass 負責接受這三個引數，Ⓕ建立該 class。大多數的物件導向程式語言會提供一個預設的實作。Python 的特Ⓕ之處在於它能Ⓕ建立自訂的 metaclass。大部分的使用者從未需要此工具，但是當需要時，metaclass 可以提供Ⓕ大且優雅的解Ⓕ方案。它們已被用於記Ⓕ屬性存取、增加執行緒安全性、追Ⓕ物件建立、實作單例模式 (singleton)，以及許多其他的任務。

更多資訊可以在 metaclasses 章節中找到。

method（方法）
一個在 class 本體Ⓕ被定義的函式。如果 method 作Ⓕ其 class 實例的一個屬性被呼叫，則它將會得到該實例物件成Ⓕ它的第一個*argument*（引數）（此引數通常被稱Ⓕ `self`）。請參Ⓕ*function*（函式）和*nested scope*（巢狀作用域）。

method resolution order（方法解析順序）
方法解析順序是在查找某個成員的過程中，base class（基底類Ⓕ）被搜尋的順序。關於 Python 自 2.3 版直譯器所使用的演算法細節，請參Ⓕ python_2.3_mro。

module（模組）
一個擔任 Python 程式碼的組織單位 (organizational unit) 的物件。模組有一個命名空間，它包含任意的 Python 物件。模組是藉由*importing* 的過程，被載入至 Python。

另請參Ⓕ*package*（套件）。

module spec（模組規格）
一個命名空間，它包含用於載入模組的 import 相關資訊。它是 `importlib.machinery.ModuleSpec` 的一個實例。

另請參Ⓕ module-specs。

MRO
請參Ⓕ*method resolution order*（方法解析順序）。

mutable（可變物件）
可變物件可以改變它們的值，但維持它們的 `id()`。另請參Ⓕ*immutable*（不可變物件）。

named tuple（附名元組）
術語「named tuple（附名元組）」是指從 tuple 繼承的任何型Ⓕ或 class，且它的可索引 (indexable) 元素也可以用附名屬性來存取。這些型Ⓕ或 class 也可以具有其他的特性。

有些Ⓕ建型Ⓕ是 named tuple，包括由 `time.localtime()` 和 `os.stat()` 回傳的值。另一個例子是 `sys.float_info`：

```
>>> sys.float_info[1]                    # 以索引存取
1024
>>> sys.float_info.max_exp               # 以欄位名稱存取
1024
>>> isinstance(sys.float_info, tuple)    # 屬於 tuple 型Ⓕ
True
```

有些 named tuple 是Ⓕ建型Ⓕ（如上例）。或者，一個 named tuple 也可以從一個正規的 class 定義來建立，只要該 class 是繼承自 `tuple`，且定義了附名欄位 (named field) 即可。這類的 class 可以手工編

寫、可以繼承自 `typing.NamedTuple` 來建立，也可以使用工廠函式 (factory function) `collections.namedtuple()` 來建立。後者技術也增加了一些額外的 method，這些 method 可能是在手寫或Ⓕ建的 named tuple 中，無法找到的。

namespace（命名空間）
變數被儲存的地方。命名空間是以 dictionary（字典）被實作。有區域的、全域的及Ⓕ建的命名空間，而在物件中（在 method 中）也有巢狀的命名空間。命名空間藉由防止命名衝突，來支援模組化。例如，函式 `builtins.open` 和 `os.open()` 是透過它們的命名空間來區分彼此。命名空間也藉由明確地區分是哪個模組在實作一個函式，來增進可讀性及可維護性。例如，寫出 `random.seed()` 或 `itertools.islice()` 明確地表示，這些函式分Ⓕ是由 `random` 和 `itertools` 模組在實作。

namespace package（命名空間套件）
一個*package*（套件），它只能作Ⓕ子套件 (subpackage) 的一個容器。命名空間套件可能Ⓕ有實體的表示法，而且具體來Ⓕ它們不像是一個*regular package*（正規套件），因Ⓕ它們ⒻⒻ有 `__init__.py` 這個檔案。

命名空間套件允許數個可獨立安裝的套件擁有一個共同的父套件。除此之外，建議使用*regular package*。

更多資訊，請參Ⓕ **PEP 420** 和 reference-namespace-package。

另請參Ⓕ*module*（模組）。

nested scope（巢狀作用域）
能Ⓕ參照外層定義 (enclosing definition) 中的變數的能力。舉例來Ⓕ，一個函式如果是在另一個函式中被定義，則它便能Ⓕ參照外層函式中的變數。請注意，在預設情Ⓕ下，巢狀作用域僅適用於參照，而無法用於賦值。區域變數能在最Ⓕ層作用域中讀取及寫入。同樣地，全域變數是在全域命名空間中讀取及寫入。`nonlocal` 容許對外層作用域進行寫入。

new-style class（新式類Ⓕ）
一個舊名，它是指現在所有的 class 物件所使用的 class 風格。在早期的 Python 版本中，只有新式 class 才能使用 Python 較新的、多樣的功能，像是 `__slots__`、描述器 (descriptor)、屬性 (property)、`__getattribute__()`、class method（類Ⓕ方法）和 static method（Ⓕ態方法）。

object（物件）
具有狀態（屬性或值）及被定義的行Ⓕ（method）的任何資料。它也是任何*new-style class*（新式類Ⓕ）的最終 base class（基底類Ⓕ）。

optimized scope（最佳化作用域）
A scope where target local variable names are reliably known to the compiler when the code is compiled, allowing optimization of read and write access to these names. The local namespaces for functions, generators, coroutines, comprehensions, and generator expressions are optimized in this fashion. Note: most interpreter optimizations are applied to all scopes, only those relying on a known set of local and nonlocal variable names are restricted to optimized scopes.

package（套件）
一個 Python 的*module*（模組），它可以包含子模組 (submodule) 或是遞Ⓕ的子套件 (subpackage)。技術上而言，套件就是具有 `__path__` 屬性的一個 Python 模組。

另請參Ⓕ*regular package*（正規套件）和*namespace package*（命名空間套件）。

parameter（參數）
在*function*（函式）或 method 定義中的一個命名實體 (named entity)，它指明該函式能Ⓕ接受的一個*argument*（引數），或在某些情Ⓕ下指示多個引數。共有有五種不同的參數類型：

- *positional-or-keyword*（位置或關鍵字）：指明一個可以按照位置或是作Ⓕ關鍵字引數被傳遞的引數。這是參數的預設類型，例如以下的 *foo* 和 *bar*：

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

- *positional-only*（僅限位置）：指明一個只能按照位置被提供的引數。在函式定義的參數列表中包含一個 / 字元，就可以在該字元前面定義僅限位置參數，例如以下的 *posonly1* 和 *posonly2*：

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

- *keyword-only*（僅限關鍵字）：指明一個只能以關鍵字被提供的引數。在函式定義的參數列表中，包含一個任意數量位置參數 (var-positional parameter) 或是單純的 `*` 字元，就可以在其後方定義僅限關鍵字參數，例如以下的 *kw_only1* 和 *kw_only2*：

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

- *var-positional*（任意數量位置）：指明一串能以任意序列被提供的位置引數（在已被其他參數接受的任何位置引數之外）。這類參數是透過在其參數名稱字首加上 `*` 來定義的，例如以下的 *args*：

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

- *var-keyword*（任意數量關鍵字）：指明可被提供的任意數量關鍵字引數（在已被其他參數接受的任何關鍵字引數之外）。這類參數是透過在其參數名稱字首加上 `**` 來定義的，例如上面範例中的 *kwargs*。

參數可以指明引數是選擇性的或必需的，也可以Ⓕ一些選擇性的引數指定預設值。

另請參Ⓕ術語表的*argument*（引數）條目、常見問題中的引數和參數之間的差Ⓕ、`inspect.Parameter` class、function 章節，以及 **PEP 362**。

path entry（路徑項目）
在*import path*（引入路徑）中的一個位置，而*path based finder*（基於路徑的尋檢器）會參考該位置來尋找要 import 的模組。

path entry finder（路徑項目尋檢器）
被 `sys.path_hooks` 中的一個可呼叫物件 (callable)（意即一個*path entry hook*）所回傳的一種*finder*，它知道如何以一個*path entry*定位模組。

關於路徑項目尋檢器實作的 method，請參Ⓕ `importlib.abc.PathEntryFinder`。

path entry hook（路徑項目Ⓕ）
在 `sys.path_hooks` 列表中的一個可呼叫物件 (callable)，若它知道如何在一個特定的*path entry* 中尋找模組，則會回傳一個*path entry finder*（路徑項目尋檢器）。

path based finder（基於路徑的尋檢器）
預設的*元路徑尋檢器 (meta path finder)* 之一，它會在一個*import path* 中搜尋模組。

path-like object（類路徑物件）
一個表示檔案系統路徑的物件。類路徑物件可以是一個表示路徑的 `str` 或 `bytes` 物件，或是一個實作 `os.PathLike` 協定的物件。透過呼叫 `os.fspath()` 函式，一個支援 `os.PathLike` 協定的物件可以被轉ⒻⒻ `str` 或 `bytes` 檔案系統路徑；而 `os.fsdecode()` 及 `os.fsencode()` 則分Ⓕ可用於確保 `str` 及 `bytes` 的結果。由 **PEP 519** 引入。

PEP
Python Enhancement Proposal（Python 增Ⓕ提案）。PEP 是一份設計Ⓕ明文件，它能Ⓕ Python 社群提供資訊，或是描述 Python 的一個新功能或該功能的程序和環境。PEP 應該要提供簡潔的技術規範以及被提案功能的運作原理。

PEP 的存在目的，是要成Ⓕ重大新功能的提案、社群中關於某個問題的意見收集，以及已納入 Python 的設計Ⓕ策的記Ⓕ，這些過程的主要機制。PEP 的作者要負責在社群Ⓕ建立共識Ⓕ記Ⓕ反對意見。

請參Ⓕ **PEP 1**。

portion（部分）
在單一目Ⓕ中的一組檔案（也可能儲存在一個 zip 檔中），這些檔案能對一個命名空間套件 (namespace package) 有所貢獻，如同 **PEP 420** 中的定義。

positional argument（位置引數）
請參Ⓕ*argument*（引數）。

provisional API（暫行 API）
暫行 API 是指，從標准函式庫的向後相容性 (backwards compatibility) 保證中，故意被排除的 API。雖然此類介面，只要它們被標示Ⓕ暫行的，理論上Ⓕ不會有重大的變更，但如果核心開發人員認

F有必要，也可能會出現向後不相容的變更（甚至包括移除該介面）。這種變更F不會無端地F生——只有 API 被納入之前未察覺的嚴重基本缺陷被揭露時，它們才會發生。

即使對於暫行 API，向後不相容的變更也會被視F「最後的解F方案」——對於任何被發現的問題，仍然會盡可能找出一個向後相容的解F方案。

這個過程使得標準函式庫能隨著時間不斷進化，而避免耗費過長的時間去鎖定有問題的設計錯誤。請參F **PEP 411** 了解更多細節。

provisional package（暫行套件）
請參F*provisional API*（暫行 API）。

Python 3000
Python 3.x 系列版本的F稱（很久以前創造的，當時第 3 版的發布是在F遠的未來。）也可以縮寫F「Py3k」。

Pythonic（Python 風格的）
一個想法或一段程式碼，它應用了 Python 語言最常見的慣用語，而不是使用其他語言常見的概念來實作程式碼。例如，Python 中常見的一種習慣用法，是使用一個 `for` 陳述式，對一個可F代物件的所有元素進行F圈。許多其他語言FF有這種類型的架構，所以不熟悉 Python 的人有時會使用一個數值計數器來代替：

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

相較之下，以下方法更簡潔、更具有 Python 風格：

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

qualified name（限定名稱）
一個「點分隔名稱」，它顯示從一個模組的全域作用域到該模組中定義的 class、函式或 method 的「路徑」，如 **PEP 3155** 中的定義。對於頂層的函式和 class 而言，限定名稱與其物件名稱相同：

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

當用於引用模組時，*完全限定名稱* (*fully qualified name*) 是表示該模組的完整點分隔路徑，包括任何的父套件，例如 `email.mime.text`：

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

reference count（參照計數）
對於一個物件的參照次數。當一個物件的參照計數下降到零時，它會被解除配置 (deallocated)。有些物件是「*不滅的* (immortal)」F擁有不會被改變的參照計數，也因此永遠不會被解除配置。參照計數通常在 Python 程式碼中看不到，但它F是*CPython* 實作的一個關鍵元素。程式設計師可以呼叫 `getrefcount()` 函式來回傳一個特定物件的參照計數。

在*CPython* 中，參照計數不被視F穩定或明確定義的值；對物件的參照數量，以及該數量如何受到 Python 程式碼的影響，在不同版本之間可能會有所不同。

regular package（正規套件）
一個傳統的*package*（套件），例如一個包含 `__init__.py` 檔案的目F。

另請參Ⓕ*namespace package*（命名空間套件）。

REPL
「read-eval-print Ⓕ圈 (read–eval–print loop)」的縮寫，是互動式直譯器 shell 的另一個名稱。

__slots__
在 class Ⓕ部的一個宣告，它藉由預先宣告實例屬性的空間，以及消除實例 dictionary（字典），來節省記憶體。雖然該技術很普遍，但它有點難以正確地使用，最好保留給那種在一個記憶體關鍵 (memory-critical) 的應用程式中存在大量實例的罕見情Ⓕ。

sequence（序列）
一個*iterable*（可Ⓕ代物件），它透過 `__getitem__()` special method（特殊方法），使用整數索引來支援高效率的元素存取，Ⓕ定義了一個 `__len__()` method 來回傳該序列的長度。一些Ⓕ建序列型Ⓕ包括 `list`、`str`、`tuple` 和 `bytes`。請注意，雖然 `dict` 也支援 `__getitem__()` 和 `__len__()`，但它被視Ⓕ對映 (mapping) 而不是序列，因Ⓕ其查找方式是使用任意的*hashable* 鍵，而不是整數。

抽象基底類Ⓕ (abstract base class) `collections.abc.Sequence` 定義了一個更加豐富的介面，Ⓕ不僅止於 `__getitem__()` 和 `__len__()`，還增加了 `count()`、`index()`、`__contains__()` 和 `__reversed__()`。實作此擴充介面的型Ⓕ，可以使用 `register()` 被明確地ⒻⒻ。更多關於序列方法的文件，請見常見序列操作。

set comprehension（集合綜合運算）
一種緊密的方法，用來處理一個可Ⓕ代物件中的全部或部分元素，Ⓕ將處理結果以一個 set 回傳。`results = {c for c in 'abracadabra' if c not in 'abc'}` 會Ⓕ生一個字串 set：`{'r', 'd'}`。請參Ⓕ comprehensions。

single dispatch（單一調度）
generic function（泛型函式）調度的一種形式，在此，實作的選擇是基於單一引數的型Ⓕ。

slice（切片）
一個物件，它通常包含一段*sequence*（序列）的某一部分。建立一段切片的方法是使用下標符號 (subscript notation) `[]`，若要給出多個數字，則在數字之間使用冒號，例如 `variable_name[1:3:5]`。在括號（下標）符號的Ⓕ部，會使用 `slice` 物件。

soft deprecated（軟性Ⓕ用）
被軟性Ⓕ用的 API 代表不應再用於新程式碼中，但在現有程式碼中繼續使用它仍會是安全的。API 仍會以文件記ⒻⒻ會被測試，但不會被繼續改進。

與正常Ⓕ用不同，軟性Ⓕ用Ⓕ有Ⓕ除 API 的規劃，也不會發出警告。

請參Ⓕ PEP 387：軟性Ⓕ用。

special method（特殊方法）
一種會被 Python 自動呼叫的 method，用於對某種型Ⓕ執行某種運算，例如加法。這種 method 的名稱會在開頭和結尾有兩個下底Ⓕ。Special method 在 specialnames 中有詳細Ⓕ明。

標准函式庫
包含套件、模組和擴充模組的集合，它們是作Ⓕ官方 Python 直譯器套件的一部分來發行。該集合的成員可能會因平台、可用的系統函式庫或其他條件而有所不同。相關文件可以在 library-index 中找到。

請參Ⓕ `sys.stdlib_module_names` 以取得所有可能的標准函式庫模組名稱的列表。

statement（陳述式）
陳述式是一個套組（suite，一個程式碼「區塊」）中的一部分。陳述式可以是一個*expression*（運算式），或是含有關鍵字（例如 `if`、`while` 或 `for`）的多種結構之一。

static type checker（Ⓕ態型Ⓕ檢查器）
會讀取 Python 程式碼Ⓕ分析的外部工具，能Ⓕ找出錯誤，像是使用了不正確的型Ⓕ。另請參Ⓕ型Ⓕ提示 *(type hints)* 以及 `typing` 模組。

stdlib（標准函式庫）
standard library 的縮寫。

strong reference（Ⓕ參照）
在 Python 的 C API 中，Ⓕ參照是對物件的參照，該物件Ⓕ持有該參照的程式碼所擁有。建立參照時透過呼叫 `Py_INCREF()` 來獲得Ⓕ參照、Ⓕ除參照時透過 `Py_DECREF()` 釋放Ⓕ參照。

`Py_NewRef()` 函式可用於建立一個對物件的Ⓕ參照。通常，在退出Ⓕ參照的作用域之前，必須在該Ⓕ參照上呼叫 `Py_DECREF()` 函式，以避免Ⓕ漏一個參照。

另請參Ⓕ*borrowed reference*（借用參照）。

t-string（t 字串）
t-strings（t 字串）
以 `t` 或 `T` Ⓕ前綴的字串文本通常被稱Ⓕ「t 字串」，它是模板化的字串文本的縮寫。

text encoding（文字編碼）
Python 中的字串是一個 Unicode 編碼位置 (code point) 的序列（範圍在 `U+0000` -- `U+10FFFF` 之間）。若要儲存或傳送一個字串，它必須被序列化Ⓕ一個位元組序列。

將一個字串序列化Ⓕ位元組序列，稱Ⓕ「編碼」，而從位元組序列重新建立該字串則稱Ⓕ「解碼 (decoding)」。

有多種不同的文字序列化編解碼器 (codecs)，它們被統稱Ⓕ「文字編碼」。

text file（文字檔案）
一個能Ⓕ讀取和寫入 `str` 物件的一個*file object*（檔案物件）。通常，文字檔案實際上是存取位元組導向的資料流 (byte-oriented datastream) Ⓕ會自動處理*text encoding*（文字編碼）。文字檔案的例子有：以文字模式（`'r'` 或 `'w'`）開Ⓕ的檔案、`sys.stdin`、`sys.stdout` 以及 `io.StringIO` 的實例。

另請參Ⓕ*binary file*（二進位檔案），它是一個能Ⓕ讀取和寫入*類位元組串物件 (bytes-like object)* 的檔案物件。

thread state
The information used by the *CPython* runtime to run in an OS thread. For example, this includes the current exception, if any, and the state of the bytecode interpreter.

Each thread state is bound to a single OS thread, but threads may have many thread states available. At most, one of them may be *attached* at once.

An *attached thread state* is required to call most of Python's C API, unless a function explicitly documents otherwise. The bytecode interpreter only runs under an attached thread state.

Each thread state belongs to a single interpreter, but each interpreter may have many thread states, including multiple for the same OS thread. Thread states from multiple interpreters may be bound to the same thread, but only one can be *attached* in that thread at any given moment.

See Thread State and the Global Interpreter Lock for more information.

token
原始碼的小單位，由 詞法分析器（也稱Ⓕ *tokenizer*）Ⓕ生。名稱、數字、字串、運算子、Ⓕ行符號等都以 token 表示。

`tokenize` 模組公開了 Python 的詞法分析器。`token` 模組包含各種 token 類型的資訊。

triple-quoted string（三引號Ⓕ字串）
由三個雙引號 (") 或單引號 (') 的作Ⓕ邊界的一個字串。雖然它們ⒻⒻ有提供Ⓕ於單引號字串的任何額外功能，但基於許多原因，它們仍是很有用的。它們讓你可以在字串中包含未跳Ⓕ (unescaped) 的單引號和雙引號，而且它們不需使用連續字元 (continuation character) 就可以跨越多行，這使得它們在編寫Ⓕ明字串時特Ⓕ有用。

type（型Ⓕ）
一個 Python 物件的型ⒻⒻ定了它是什Ⓕ類型的物件；每個物件都有一個型Ⓕ。一個物件的型Ⓕ可以用它的 `__class__` 屬性來存取，或以 `type(obj)` 來檢索。

type alias（型ⒻⒻ名）
一個型Ⓕ的同義詞，透過將型Ⓕ指定給一個識Ⓕ符 (identifier) 來建立。

型ⒻⒻ名對於簡化*型Ⓕ提示 (type hint)* 很有用。例如：

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

可以寫成這樣，更具有可讀性：

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

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

請參Ⓕ `typing` 和 **PEP 484**，有此功能的描述。

type hint（型Ⓕ提示）
一種*annotation*（Ⓕ釋），它指定一個變數、一個 class 屬性或一個函式的參數或回傳值的預期型Ⓕ。

型Ⓕ提示是選擇性的，而不是被 Python Ⓕ制的，但它們對Ⓕ態型Ⓕ檢查器 *(static type checkers)*很有用，Ⓕ能協助 IDE 完成程式碼的補全 (completion) 和重構 (refactoring)。

全域變數、class 屬性和函式（不含區域變數）的型Ⓕ提示，都可以使用 `typing.get_type_hints()` 來存取。

請參Ⓕ `typing` 和 **PEP 484**，有此功能的描述。

universal newlines（通用Ⓕ行字元）
一種解譯文字流 (text stream) 的方式，會將以下所有的情Ⓕ識ⒻⒻ一行的結束：Unix 行尾慣例 `'\n'`、Windows 慣例 `'\r\n'` 和舊的 Macintosh 慣例 `'\r'`。請參Ⓕ **PEP 278** 和 **PEP 3116**，以及用於 `bytes.splitlines()` 的附加用途。

variable annotation（變數Ⓕ釋）
一個變數或 class 屬性的*annotation*（Ⓕ釋）。

Ⓕ釋變數或 class 屬性時，賦值是選擇性的：

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

變數Ⓕ釋通常用於型Ⓕ提示 *(type hint)*：例如，這個變數預期會取得 `int`（整數）值：

```
count: int = 0
```

變數Ⓕ釋的語法在 annassign 章節有詳細的解釋。

請參Ⓕ*function annotation*（函式Ⓕ釋）、**PEP 484** 和 **PEP 526**，皆有此功能的描述。關於Ⓕ釋的最佳實踐方法，另請參Ⓕ annotations-howto。

virtual environment（Ⓕ擬環境）
一個協作隔離 (cooperatively isolated) 的執行環境，能讓 Python 的使用者和應用程式得以安裝和升級 Python 發Ⓕ套件，而不會對同一個系統上運行的其他 Python 應用程式的行ⒻⒻ生干擾。

另請參Ⓕ `venv`。

virtual machine（Ⓕ擬機器）
一部完全由軟體所定義的電腦 (computer)。Python 的Ⓕ擬機器會執行由*bytecode*（位元組碼）編譯器所發出的位元組碼。

walrus operator（海象運算子）
A light-hearted way to refer to the assignment expression operator `:=` because it looks a bit like a walrus if you turn your head.

Zen of Python（Python 之Ⓕ）
Python 設計原則與哲學的列表，其Ⓕ容有助於理解和使用此語言。此列表可以透過在互動式提式字元後輸入「`import this`」來找到它。

APPENDIX B

關於這份Ⓕ明文件

Python Ⓕ明文件是透過使用 Sphinx（一個原Ⓕ Python 而生的文件Ⓕ生器、目前是以獨立專案形式來維護）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

沿革與授權

C.1 軟體沿革

Python 是由荷蘭數學和計算機科學研究學會（CWI，見 https://www.cwi.nl）的 Guido van Rossum 於 1990 年代早期所創造，目的是作F一種稱F ABC 語言的後繼者。F管 Python 包含了許多來自其他人的貢獻，Guido 仍是其主要作者。

1995 年，Guido 在維吉尼亞州雷斯頓的國家創新研究公司（CNRI，見 https://www.cnri.reston.va.us）繼續他在 Python 的工作，F在那F發F了該軟體的多個版本。

2000 年五月，Guido 和 Python 核心開發團隊轉移到 BeOpen.com F成立了 BeOpen PythonLabs 團隊。同年十月，PythonLabs 團隊轉移到 Digital Creations，後來成F Zope Corporation。2001 年，Python 軟體基金會（PSF，見 https://www.python.org/psf/）成立，這是一個專F擁有 Python 相關的智慧財F權而創立的非營利組織。Zope Corporation 過去是 PSF 的一個贊助會員。

所有的 Python 版本都是開源的（有關開源的定義，參F https://opensource.org）。歷史上，大多數但非全部的 Python 版本，也是 GPL 相容的；以下表格總結各個版本的差F。

發F版本	源自	年份	擁有者	GPL 相容性？(1)
0.9.0 至 1.2	不適用	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	是 (2)
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 以上	2.1.1	2001 至今	PSF	是

備F

(1) GPL 相容F不表示我們是在 GPL 下發F Python。不像 GPL，所有的 Python 授權都可以讓你發F修改後的版本，但不一定要使你的變更成F開源。GPL 相容的授權使得 Python 可以結合其他

在 GPL 下發Ⓕ的軟體一起使用；但其它的授權則不行。

(2) 根據 Richard Stallman 的Ⓕ法，1.6.1 不是 GPL 相容的，因Ⓕ其授權有一個法律選擇條款。然而根據 CNRI 的Ⓕ法，Stallman 的律師告訴 CNRI 的律師，1.6.1 與 GPL「不相容」。

感謝許多的外部志工，在 Guido 指導下的付出，使得這些版本的發Ⓕ成Ⓕ可能。

C.2 關於存取或以其他方式使用 Python 的合約條款

Python 軟體和Ⓕ明文件的授權是基於 Python 軟體基金會授權第二版 (Python Software Foundation License Version 2)。

從 Python 3.8.6 開始，Ⓕ明文件中的範例、程式庫和其他程式碼，是被雙重授權 (dual licensed) 於 PSF 授權第二版以及*Zero-Clause BSD 授權*。

有些被納入 Python 中的軟體是基於不同的授權。這些授權將會與其授權之程式碼一起被列出。關於這些授權的不完整清單，請參Ⓕ*被收Ⓕ軟體的授權與致謝*。

C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

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

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

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON, 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, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
```

(繼續下一頁)

(繼續上一頁)

```
   internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   internet using the following URL: http://hdl.handle.net/1895.22/1013".

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

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

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

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

(繼續下一頁)

(繼續上一頁)

```
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 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Ⓕ軟體的授權與致謝

本節是一個不完整但持續增加的授權與致謝清單，對象是在 Python 發Ⓕ版本中所收Ⓕ的第三方軟體。

C.3.1 Mersenne Twister

`random` 模組底下的 `_random` C 擴充程式包含了以 http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html 的下載Ⓕ容Ⓕ基礎的程式碼。以下是原始程式碼的完整聲明：

```
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
```

(繼續下一頁)

(繼續上一頁)

```
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 Sockets

`socket` 模組使用 `getaddrinfo()` 和 `getnameinfo()` 函式，它們在 WIDE 專案（https://www.wide.ad.jp/）Ⓕ，於不同的原始檔案中被編碼：

```
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 非同步 socket 服務

`test.support.asynchat` 和 `test.support.asyncore` 模組包含以下聲明：

```
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
```

(繼續下一頁)

(繼續上一頁)

```
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` 模組包含以下聲明:

```
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 執行追Ⓕ

`trace` 模組包含以下聲明:

```
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,
```

(繼續下一頁)

(繼續上一頁)

```
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 與 UUdecode 函式

uu 編解碼器包含以下聲明:

```
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 遠端程序呼叫

xmlrpc.client 模組包含以下聲明:

```
    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
```

(繼續下一頁)

(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.test_epoll` 模組包含以下聲明:

```
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` 模組對於 kqueue 介面包含以下聲明:

```
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
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

`Python/pyhash.c` 檔案包含 Marek Majkowski' 基於 Dan Bernstein 的 SipHash24 演算法的實作。它包含以下聲明：

```
<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 與 dtoa

`Python/dtoa.c` 檔案提供了 C 的 dtoa 和 strtod 函式，用於將 C 的雙精度浮點數和字串互相轉Ⓕ。該檔案是衍生自 David M. Gay 建立的同名檔案，後者現在可以從 https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c 下載。於 2009 年 3 月 16 日所檢索的原始檔案包含以下版權與授權聲明：

```
/****************************************************************
 *
 * 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 OpenSSL 函式庫可被作業系統使用，則 `hashlib`、`posix`、`ssl` 模組會使用它來提升效能。此外，因Ⓕ Windows 和 macOS 的 Python 安裝程式可能包含 OpenSSL 函式庫的副本，所以我們也在此收Ⓕ OpenSSL 授權的副本。對於 OpenSSL 3.0 版本以及由此衍生的更新版本則適用 Apache 許可證 v2：

```
                                 Apache License
                           Version 2.0, January 2004
                        https://www.apache.org/licenses/
```

(繼續下一頁)

(繼續上一頁)

```
TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.
```

(繼續下一頁)

(繼續上一頁)

```
2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
```

(繼續下一頁)

(繼續上一頁)

```
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

C.3.13 expat

除非在建置 `pyexpat` 擴充時設定Ⓕ `--with-system-expat`，否則該擴充會用一個Ⓕ含 expat 原始碼的副本來建置：

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper
```

(繼續下一頁)

(繼續上一頁)

```
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

除非在建置 `_ctypes` 模組底下 `_ctypes` 擴充程式時設定Ⓕ `--with-system-libffi`，否則該擴充會用一個Ⓕ含 libffi 原始碼的副本來建置：

```
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

如果在系統上找到的 zlib 版本太舊以致於無法用於建置 `zlib` 擴充，則該擴充會用一個Ⓕ含 zlib 原始碼的副本來建置：

```
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
```

(繼續下一頁)

(繼續上一頁)

```
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` 使用的雜Ⓕ表 (hash table) 實作，是以 cfuhash 專案Ⓕ基礎：

```
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

除非在建置 `decimal` 模組底下 `_decimal` C 擴充程式時設定Ⓕ `--with-system-libmpdec`，否則該模組會用一個Ⓕ含 libmpdec 函式庫的副本來建置：

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.
```

(繼續下一頁)

(繼續上一頁)

```
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.18 W3C C14N 測試套件

`test` 程式包中的 C14N 2.0 測試套件 (`Lib/test/xmltestdata/c14n-20/`) 是從 W3C 網站 https://www.w3.org/TR/xml-c14n2-testcases/ 被檢索，且是基於 3-clause BSD 授權被發🅵：

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.19 mimalloc

MIT 授權：

```
Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

C.3.20 asyncio

`asyncio` 模組的部分🅵容是從 uvloop 0.16 中收🅵過來，其基於 MIT 授權來發🅵：

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.21 Global Unbounded Sequences (GUS)

The file `Python/qsbr.c` is adapted from FreeBSD's "Global Unbounded Sequences" safe memory reclamation scheme in subr_smr.c. The file is distributed under the 2-Clause BSD License:

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice unmodified, this list of conditions, and the following
```

(繼續下一頁)

(繼續上一頁)

```
   disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.22 Zstandard bindings

Zstandard bindings in `Modules/_zstd` and `Lib/compression/zstd` are based on code from the pyzstd library, copyright Ma Lin and contributors. The pyzstd code is distributed under the 3-Clause BSD License:

```
Copyright (c) 2020-present, Ma Lin and contributors.
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 copyright holder 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 HOLDER 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 和這份F明文件的版權：

Copyright © 2001 Python 軟體基金會。保留所有權利。

Copyright © 2000 BeOpen.com 保留所有權利。

Copyright © 1995-2000 Corporation for National Research Initiatives 保留所有權利。

Copyright © 1991-1995 Stichting Mathematisch Centrum 保留所有權利。

完整的授權條款資訊請參見沿革與授權。

索引

非依字母順序

. . . , **109**
    ellipsis literal, 20
    F節號字面值, 109
(*hash*)
    comment（F解）, 9
*（星號）
    於函式呼叫中, 29
**
    於函式呼叫中, 29
:（冒號）
    function annotations（函式F釋）, 30
->
    function annotations（函式F釋）, 30
>>>, **109**
__all__, 49
__future__, **115**
__slots__, **123**
環境變數
    PATH, 45, 107
    PYTHON_BASIC_REPL, 107
    PYTHON_GIL, 116
    PYTHONPATH, 45, 46
    PYTHONSTARTUP, 108

A

abstract base class（抽象基底類F）, **109**
annotate function（F釋函式）, **109**
annotations（F釋）
    function（函式）, 30
annotation（F釋）, **109**
argument（引數）, **109**
asynchronous context manager（非同步情境管理器）, **110**
asynchronous generator iterator（非同步F生器F代器）, **110**
asynchronous generator（非同步F生器）, **110**
asynchronous iterable（非同步可F代物件）, **110**
asynchronous iterator（非同步F代器）, **110**
attached thread state, **110**
attribute（屬性）, **111**
awaitable（可等待物件）, **111**

B

BDFL, **111**
binary file（二進位檔案）, **111**
borrowed reference（借用參照）, **111**
built-in function（F建函式）
    help（幫助）, 81
    open, 55
builtins（F建）
    module（模組）, 47
bytecode（位元組碼）, **111**
bytes-like object（類位元組串物件）, **111**

C

callable（可呼叫物件）, **111**
callback（回呼）, **111**
C-contiguous（C 連續的）, 112
class variable（類F變數）, **112**
class（類F）, **111**
closure variable（閉包變數）, **112**
coding（程式編寫）
    style（風格）, 31
complex number（F數）, **112**
context management protocol（情境管理協定）, **112**
context manager（情境管理器）, **112**
context variable（情境變數）, **112**
context（情境）, **112**
contiguous（連續的）, **112**
coroutine function（協程函式）, **112**
coroutine（協程）, **112**
CPython, **113**
current context, **113**
cyclic isolate, **113**

D

ddocumentation strings（F明字串）, 23, 30
decorator（裝飾器）, **113**
descriptor（描述器）, **113**
dictionary comprehension(字典綜合運算), **113**
dictionary view（字典檢視）, **113**
dictionary（字典）, **113**
docstrings（F明字串）, 23, 30
docstring（F明字串）, **113**

duck-typing（鴨子型Ⓕ），113
dunder（雙底Ⓕ），114

E

EAFP, 114
evaluate function（求值函式），114
expression（運算式），114
extension module（擴充模組），114

F

f-strings（f 字串），114
f-string（f 字串），114
file object（檔案物件），114
file-like object（類檔案物件），114
filesystem encoding and error handler（檔案系統編碼和錯誤處理函式），114
file（檔案）
    object（物件），55
finder（尋檢器），114
floor division（向下取整除法），115
for
    statement（陳述式），17
formatted string literal（格式化的字串常數），52
Fortran contiguous（Fortran 連續的），112
free threading（自由執行緒），115
free variable（自由變數），115
fstring（f 字串），52
f-string（f 字串），52
function annotation（函式Ⓕ釋），115
function（函式），115
    annotations（Ⓕ釋），30

G

garbage collection（垃圾回收），115
generator expression（Ⓕ生器運算式），115, 116
generator iterator（Ⓕ生器Ⓕ代器），115
generator（Ⓕ生器），115
generic function（泛型函式），116
generic type（泛型型Ⓕ），116
GIL, 116
global interpreter lock（全域直譯器鎖），116

H

hash-based pyc（雜Ⓕ架構的 pyc），116
hashable（可雜Ⓕ的），116
help（幫助）
    built-in function（Ⓕ建函式），81

I

IDLE, 116
immortal（不滅），116
immutable（不可變物件），117
import path（引入路徑），117
importer（引入器），117
importing（引入），117
interactive（互動的），117
interpolated string literal（Ⓕ插字串常數），52
interpreted（直譯的），117
interpreter shutdown（直譯器關閉），117
iterable（可Ⓕ代物件），117
iterator（Ⓕ代器），117

J

json
    module（模組），57

K

key function（鍵函式），118
keyword argument（關鍵字引數），118

L

lambda, 118
LBYL, 118
lexical analyzer（詞法分析器），118
list comprehension（串列綜合運算），118
list（串列），118
loader（載入器），118
locale encoding（區域編碼），118

M

magic
    method（方法），119
magic method（魔術方法），119
mangling（修飾）
    name（名稱），77
mapping（對映），119
meta path finder（元路徑尋檢器），119
metaclass（元類Ⓕ），119
method resolution order（方法解析順序），119
method（方法），119
    magic, 119
    object（物件），73
    special, 123
module spec（模組規格），119
module（模組），119
    builtins（Ⓕ建），47
    json, 57
    search（搜尋）path（路徑），45
    sys, 46
MRO, 119
mutable（可變物件），119

N

named tuple（附名元組），119
namespace package（命名空間套件），120
namespace（命名空間），120
name（名稱）
    mangling（修飾），77
nested scope（巢狀作用域），120
new-style class（新式類Ⓕ），120

O

object（物件），120

file（檔案），55
method（方法），73
open
built-in function（F建函式），55
optimized scope（最佳化作用域），**120**

P

package（套件），**120**
parameter（參數），**120**
PATH, 45, 107
path based finder（基於路徑的尋檢器），**121**
path entry finder（路徑項目尋檢器），**121**
path entry hook（路徑項目F），**121**
path entry（路徑項目），**121**
path-like object（類路徑物件），**121**
path（路徑）
module（模組）search（搜尋），45
PEP, **121**
portion（部分），**121**
positional argument（位置引數），**121**
provisional API（暫行 API），**121**
provisional package（暫行套件），**122**
Python 3000, **122**
Python Enhancement Proposals
PEP 1, 121
PEP 8, 31
PEP 238, 115
PEP 278, 125
PEP 302, 118
PEP 343, 112
PEP 362, 110, 121
PEP 411, 122
PEP 420, 120, 121
PEP 443, 116
PEP 483, 116
PEP 484, 30, 109, 115, 116, 125
PEP 492, 110112
PEP 498, 114
PEP 519, 121
PEP 525, 110
PEP 526, 109, 125
PEP 585, 116
PEP 636, 23
PEP 649, 109
PEP 683, 116
PEP 703, 115, 116
PEP 765, 64
PEP 3107, 30
PEP 3116, 125
PEP 3147, 46
PEP 3155, 122
PYTHON_BASIC_REPL, 107
PYTHON_GIL, 116
Pythonic（Python 風格的），**122**
PYTHONPATH, 45, 46
PYTHONSTARTUP, 108

Q

qualified name（限定名稱），**122**

R

reference count（參照計數），**122**
regular package（正規套件），**122**
REPL, **123**
RFC
RFC 2822, 85

S

search（搜尋）
path（路徑），module（模組），45
sequence（序列），**123**
set comprehension（集合綜合運算），**123**
single dispatch（單一調度），**123**
sitecustomize, 108
slice（切片），**123**
soft deprecated（軟性F用），**123**
special
method（方法），123
special method（特殊方法），**123**
statement（陳述式），**123**
for, 17
static type checker（F態型F檢查器），**123**
stdlib（標准函式庫），**123**
strings（字串），documentation（F明文件），23, 30
string（字串）
formatted literal（格式化的文本），52
interpolated literal（插值常數），52
strong reference（F參照），**123**
style（風格）
coding（程式編寫），31
sys
module（模組），46

T

t-strings（t 字串），**124**
t-string（t 字串），**124**
text encoding（文字編碼），**124**
text file（文字檔案），**124**
thread state, **124**
token, **124**
triple-quoted string（三引號F字串），**124**
type alias（型FF名），**124**
type hint（型F提示），**125**
type（型F），**124**

U

universal newlines（通用F行字元），**125**
usercustomize, 108

V

variable annotation（變數F釋），**125**
virtual environment（F擬環境），**125**
virtual machine（F擬機器），**125**

W

walrus operator（海象運算子）, 125
標准函式庫, 123

Z

Zen of Python（Python 之🅵）, 125