Python Frequently Asked Questions

發□ *3.13.2*

Guido van Rossum and the Python development team

3 月 03, 2025

Python Software Foundation
Email: docs@python.org

Contents

1 一般的 Python 常見問答集 **1**

1.1 一般資訊 1

1.1.1 什🄵是 Python? 1

1.1.2 什🄵是 Python 軟體基金會? 1

1.1.3 使用 Python 時有任何版權限制嗎? 1

1.1.4 當初🄵什🄵 Python 會被創造出來? 2

1.1.5 什🄵是 Python 擅長的事情? 2

1.1.6 Python 的版本編號系統是如何運作的? 2

1.1.7 我要如何得到 Python 的原始碼🄵本? 3

1.1.8 我要如何取得 Python 的🄵明文件? 3

1.1.9 我從來🄵有寫過程式，有🄵有 Python 的教學? 3

1.1.10 有🄵有 Python 專屬的新聞群組或郵件討論群? 3

1.1.11 如何取得 Python 的 beta 測試版本? 3

1.1.12 如何提交 Python 的錯誤報告和修補程式? 3

1.1.13 是否有關於 Python 的任何已出版文章可供參考? 4

1.1.14 有🄵有關於 Python 的書? 4

1.1.15 www.python.org 的真實位置在哪🄵? 4

1.1.16 🄵什🄵要取名🄵 Python? 4

1.1.17 我需要喜歡「Monty Python 的飛行馬戲團」嗎? 4

1.2 在真實世界中的 Python 4

1.2.1 Python 的穩定性如何? 4

1.2.2 有多少人在使用 Python? 4

1.2.3 有🄵有任何重要的專案使用 Python 完成開發? 4

1.2.4 Python 未來預期會有哪些新的開發? 5

1.2.5 對 Python 提出不相容的變更建議是否適當? 5

1.2.6 Python 對於入門的程式設計師而言是否🄵好的語言? 5

2 程式開發常見問答集 **7**

2.1 常見問題 7

2.1.1 是否有可以使用在程式碼階段, 具有中斷點, 步驟執行等功能的除錯器? 7

2.1.2 有🄵有工具能🄵幫忙找 bug 或執行🄵態分析? 7

2.1.3 如何從 Python 🄵本建立獨立的二進位檔案? 8

2.1.4 Python 程式碼是否有編碼標准或風格指南? 8

2.2 核心語言 8

2.2.1 🄵什🄵當變數有值時，我仍得到錯誤訊息 UnboundLocalError? 8

2.2.2 Python 的區域變數和全域變數有什🄵規則? 9

2.2.3 🄵什🄵以不同的值在🄵圈中定義的 lambda 都回傳相同的結果? 9

2.2.4 如何跨模組共享全域變數? 10

2.2.5 在模組中使用 import 的「最佳實踐」有哪些? 10

2.2.6 Why are default values shared between objects? 11

2.2.7 How can I pass optional or keyword parameters from one function to another? 12

2.2.8 引數 (arguments) 和參數 (parameters) 有什🄵區🄵? 12
2.2.9 🄵什🄵更改 list 'y' 也會更改 list 'x'? 12
2.2.10 如何編寫帶有輸出參數的函式（透過傳參照呼叫 (call by reference)）? 13
2.2.11 How do you make a higher order function in Python? 14
2.2.12 如何在 Python 中🄵🄵物件? 15
2.2.13 如何找到物件的方法或屬性? 15
2.2.14 我的程式碼如何發現物件的名稱? 15
2.2.15 逗號運算子的優先級是什🄵? 16
2.2.16 是否有等效於 C 的"?:" 三元運算子? 16
2.2.17 Is it possible to write obfuscated one-liners in Python? 16
2.2.18 函式參數串列中的斜🄵 (/) 是什🄵意思? 17
2.3 數字和字串 17
2.3.1 如何指定十六進位和八進位整數? 17
2.3.2 🄵什🄵 -22 // 10 回傳 -3? 18
2.3.3 How do I get int literal attribute instead of SyntaxError? 18
2.3.4 如何將字串轉🄵🄵數字? 18
2.3.5 如何將數字轉🄵🄵字串? 18
2.3.6 如何原地修改字串? 19
2.3.7 如何使用字串呼叫函式/方法? 19
2.3.8 是否有與 Perl 的 chomp() 等效的方法，能用於從字串中🄵除後綴的🄵行符號? 20
2.3.9 是否有 scanf() 或 sscanf() 的等效方法? 20
2.3.10 'UnicodeDecodeError' 或'UnicodeEncodeErro' 錯誤是什🄵意思? 20
2.3.11 Can I end a raw string with an odd number of backslashes? 20
2.4 Performance 21
2.4.1 我的程式太慢了。我該如何加快速度? 21
2.4.2 What is the most efficient way to concatenate many strings together? 22
2.5 Sequences (Tuples/Lists) 22
2.5.1 How do I convert between tuples and lists? 22
2.5.2 What's a negative index? 22
2.5.3 How do I iterate over a sequence in reverse order? 22
2.5.4 How do you remove duplicates from a list? 23
2.5.5 How do you remove multiple items from a list 23
2.5.6 How do you make an array in Python? 23
2.5.7 How do I create a multidimensional list? 23
2.5.8 How do I apply a method or function to a sequence of objects? 24
2.5.9 Why does a_tuple[i] += ['item'] raise an exception when the addition works? 24
2.5.10 我想做一個🄵雜的排序：你能用 Python 做一個 Schwartzian 變🄵嗎? 25
2.5.11 如何根據另一個串列中的值對一個串列進行排序? 26
2.6 物件 26
2.6.1 什🄵是類🄵 (class)? 26
2.6.2 什🄵是方法 (method)? 26
2.6.3 什🄵是 self? 26
2.6.4 How do I check if an object is an instance of a given class or of a subclass of it? 26
2.6.5 What is delegation? 27
2.6.6 How do I call a method defined in a base class from a derived class that extends it? 28
2.6.7 How can I organize my code to make it easier to change the base class? 28
2.6.8 How do I create static class data and static class methods? 29
2.6.9 How can I overload constructors (or methods) in Python? 29
2.6.10 我嘗試使用 __spam，但收到有關 _SomeClassName__spam 的錯誤。 30
2.6.11 我的類🄵定義了 __del__ 但是當我🄵除物件時它🄵有被呼叫。 30
2.6.12 How do I get a list of all instances of a given class? 31
2.6.13 🄵什🄵 `id()` 的結果看起來不唯一? 31
2.6.14 我什🄵時候可以依靠 *is* 運算子進行識🄵性測試? 31
2.6.15 子類🄵如何控制不可變實例中存儲的資料? 32
2.6.16 如何快取方法呼叫? 33
2.7 模組 34
2.7.1 如何建立.pyc 檔案? 34
2.7.2 如何找到當前模組名稱? 34

2.7.3 要怎樣才能擁有相互引入的模組？ 35
2.7.4 __import__('x.y.z') 回傳 <module 'x'>，那我怎□得到 z？ 35
2.7.5 當我編輯需要引入的模組□重新引入它時，更動□有反應出來。□什□會這樣？ . 36

3 設計和歷史常見問答集 37
3.1 □什□ Python 使用縮排將陳述式進行分組？ 37
3.2 □什□我會從簡單的數學運算得到奇怪的結果？ 37
3.3 □何浮點數運算如此不精確？ 37
3.4 □什□ Python 字串不可變動？ 38
3.5 □何「self」在方法 (method) 定義和呼叫時一定要明確使用？ 38
3.6 □何我不能在運算式 (expression) 中使用指派運算？ 39
3.7 □何 Python 對於一些功能實作使用方法（像是 list.index()），另一些使用函式（像是 len(list)）？ 39
3.8 □何 join() 是字串方法而非串列 (list) 或元組 (tuple) 方法？ 39
3.9 例外處理有多快？ 39
3.10 □什□ Python □□有 switch 或 case 陳述式？ 40
3.11 □何不能在直譯器上模擬執行緒，而要使用作業系統的特定實作方式？ 40
3.12 □何 lambda 運算式不能包含陳述式？ 40
3.13 Python 可以被編譯成機器語言、C 語言或其他種語言嗎？ 41
3.14 Python 如何管理記憶體？ 41
3.15 □何 CPython 不使用更多傳統的垃圾回收機制？ 41
3.16 當 CPython 結束時，□何所有的記憶體不會被釋放？ 41
3.17 □何要把元組 (tuple) 和串列 (list) 分成兩個資料型態？ 42
3.18 串列 (list) 在 CPython 中是怎□實作的？ 42
3.19 字典 (dictionaries) 在 CPython 中是怎□實作的？ 42
3.20 □何字典的鍵一定是不可變的？ 42
3.21 □何 list.sort() 不是回傳排序過的串列？ 43
3.22 如何在 Python 中指定和□制使用一個介面規範 (interface spec)？ 43
3.23 □何□有 goto 語法？ 44
3.24 □何純字串 (r-string) 不能以反斜□結尾？ 44
3.25 □何 Python □有屬性賦值的 with 陳述式？ 44
3.26 □何□生器 (generator) 不支援 with 陳述式？ 45
3.27 □何 if、while、def、class 陳述式□需要冒號？ 45
3.28 □何 Python 允許在串列和元組末端加上逗號？ 46

4 函式庫和擴充功能的常見問題 47
4.1 常見函式問題 47
4.1.1 我如何找到執行任務 X 的模組或應用程式？ 47
4.1.2 哪□可以找到 math.py (socket.py, regex.py, 等...) 來源檔案？ 47
4.1.3 我如何使 Python script 執行在 Unix？ 47
4.1.4 是否有適用於 Python 的 curses/termcap 套件？ 48
4.1.5 Python 中是否有等同於 C 的 onexit() 的函式？ 48
4.1.6 □什□我的信號處理程式不起作用？ 48
4.2 常見課題 48
4.2.1 如何測試 Python 程式或元件？ 48
4.2.2 How do I create documentation from doc strings? 49
4.2.3 How do I get a single keypress at a time? 49
4.3 執行緒 49
4.3.1 如何使用執行緒編寫程式？ 49
4.3.2 我的執行緒似乎都□有運行：□什□？ 49
4.3.3 How do I parcel out work among a bunch of worker threads? 50
4.3.4 What kinds of global value mutation are thread-safe? 51
4.3.5 不能擺□全局直譯器鎖嗎？ 51
4.4 輸入與輸出 52
4.4.1 如何□除檔案？（以及其他檔案問題...） 52
4.4.2 如何□□檔案？ 52
4.4.3 如何讀取（或寫入）二進位制資料？ 52
4.4.4 I can't seem to use os.read() on a pipe created with os.popen(); why? 53
4.4.5 如何存取序列 (RS232) 連接埠？ 53

4.4.6 🄵什🄵關閉 sys.stdout (stdin, stderr) 🄵🄵有真正關閉它? . . . 53
4.5 網路 (Network)/網際網路 (Internet) 程式 . . . 53
4.5.1 Python 有哪些 WWW 工具? . . . 53
4.5.2 我應該使用什🄵模組來輔助🄵生 HTML? . . . 54
4.5.3 如何從 Python 🄵本發送郵件? . . . 54
4.5.4 How do I avoid blocking in the connect() method of a socket? . . . 54
4.6 資料庫 . . . 55
4.6.1 Python 中是否有任何資料庫套件的介面? . . . 55
4.6.2 How do you implement persistent objects in Python? . . . 55
4.7 數學和數值 . . . 55
4.7.1 如何在 Python 中生成隨機數? . . . 55

5 擴充/嵌入常見問題集 57
5.1 我可以在 C 中建立自己的函式嗎? . . . 57
5.2 我可以在 C++ 中建立自己的函式嗎? . . . 57
5.3 寫 C 很難;還有其他選擇嗎? . . . 57
5.4 如何從 C 執行任意 Python 陳述式? . . . 57
5.5 How can I evaluate an arbitrary Python expression from C? . . . 58
5.6 如何從 Python 物件中提取 C 值? . . . 58
5.7 如何使用 Py_BuildValue() 建立任意長度的元組? . . . 58
5.8 如何從 C 呼叫物件的方法? . . . 58
5.9 我如何捕捉 PyErr_Print() 的輸出(或任何印出到 stdout/stderr 的東西)? . . . 58
5.10 如何從 C 存取用 Python 編寫的模組? . . . 59
5.11 How do I interface to C++ objects from Python? . . . 59
5.12 我使用安裝檔案新增了一個模組,但 make 失敗了;🄵什🄵? . . . 59
5.13 如何🄵擴充套件除錯? . . . 60
5.14 我想在我的 Linux 系統上編譯一個 Python 模組,但是缺少一些檔案。🄵什🄵? . . . 60
5.15 如何從「無效輸入」區分出「不完整輸入」? . . . 60
5.16 如何找到未定義的 g++ 符號 __builtin_new 或 __pure_virtual? . . . 60
5.17 Can I create an object class with some methods implemented in C and others in Python (e.g. through inheritance)? . . . 60

6 在 Windows 使用 Python 的常見問答集 61
6.1 如何在 Windows 作業系統🄵運行 Python 程式? . . . 61
6.2 如何使 Python 🄵本可以執行? . . . 62
6.3 🄵什🄵 Python 有時需要這🄵長的時間才能開始? . . . 62
6.4 如何從 Python 🄵本🄵作可執行檔? . . . 62
6.5 `*.pyd` 檔是否與 DLL 相同? . . . 62
6.6 如何將 Python 嵌入 Windows 應用程式中? . . . 63
6.7 如何防止編輯器在我的 Python 原始碼中插入 tab? . . . 64
6.8 如何在不阻塞的情🄵下檢查 keypress? . . . 64
6.9 如何解🄵遺漏 api-ms-win-crt-runtime-l1-1-0.dll 的錯誤? . . . 64

7 圖形使用者介面常見問答集 65
7.1 圖形使用者介面 (GUI) 的常見問題 . . . 65
7.2 Python 有哪些 GUI 套件? . . . 65
7.3 Tkinter 的問答 . . . 65
7.3.1 如何凍結 Tkinter 應用程式? . . . 65
7.3.2 是否可以在等待 I/O 時處理 Tk 事件? . . . 65
7.3.3 我無法讓鍵🄵結 (key binding) 在 Tkinter 中作用:🄵什🄵? . . . 65

8 「🄵什🄵 Python 被安裝在我的機器上?」常見問答集 67
8.1 什🄵是 Python? . . . 67
8.2 🄵什🄵 Python 被安裝在我的機器上? . . . 67
8.3 我能🄵自行🄵除 Python 嗎? . . . 67

A 術語表 69

B 關於這份🄵明文件 85

B.1 Python 文件的貢獻者們 85

C 沿革與授權 **87**

C.1 軟體沿革 87

C.2 關於存取或以其他方式使用 Python 的合約條款 88

C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2 88

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0 89

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1 89

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2 90

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION . 91

C.3 被收🄵軟體的授權與致謝 91

C.3.1 Mersenne Twister 91

C.3.2 Sockets 92

C.3.3 非同步 socket 服務 92

C.3.4 Cookie 管理 93

C.3.5 執行追🄵 93

C.3.6 UUencode 與 UUdecode 函式 94

C.3.7 XML 遠端程序呼叫 94

C.3.8 test_epoll 95

C.3.9 Select kqueue 95

C.3.10 SipHash24 96

C.3.11 strtod 與 dtoa 96

C.3.12 OpenSSL 96

C.3.13 expat 99

C.3.14 libffi 100

C.3.15 zlib 100

C.3.16 cfuhash 101

C.3.17 libmpdec 101

C.3.18 W3C C14N 測試套件 102

C.3.19 mimalloc 103

C.3.20 asyncio 103

C.3.21 Global Unbounded Sequences (GUS) 103

D 版權宣告 **105**

索引 **107**

CHAPTER 1

一般的 Python 常見問答集

1.1 一般資訊

1.1.1 什🄵是 Python？

Python 是一種直譯的、互動的、物件導向的程式設計語言。它結合了模組、例外、動態型🄵 (dynamic typing)、非常高階的動態資料型🄵，以及 class（類🄵）。它能支援物件導向程式設計之外的多種程式設計典範，例如程序式 (procedural) 和函式語言 (functional) 程式設計。Python 結合了卓越的功能與非常清晰的語法。它有許多系統呼叫和函式庫的介面，以及各種視窗系統的介面，🄵且在 C 或 C++ 中可以擴充。它還可以作🄵一種擴充語言，使用於需要可程式化介面 (programmable interface) 的應用程式。最後，Python 是可🄵的 (portable)：它能運行在許多 Unix 的變體上，包括 Linux 和 macOS，也能運行在 Windows 上。

要尋找更多🄵容，請從 tutorial-index 開始。Python 初學者指南可連結到其他介紹式教學以及學習 Python 的資源。

1.1.2 什🄵是 Python 軟體基金會？

Python 軟體基金會 (Python Software Foundation) 是一個獨立的非營利性組織，它擁有 Python 2.1 版與之後各版本的版權。PSF 的使命在於推展 Python 程式設計語言相關的開放原始碼技術，以及宣傳 Python 的使用。PSF 首頁的網址是 https://www.python.org/psf/。

在美國捐款給 PSF 是免🄵的。如果你使用了 Python 且發現它很有用，請至 PSF 捐款頁面🄵它做出貢獻。

1.1.3 使用 Python 時有任何版權限制嗎？

你可以對原始碼做任何你想做的事情，只要你保留版權，🄵且在你🄵作的任何關於 Python 的🄵明文件中顯示這些版權即可。如果你遵守版權規則，就可以將 Python 用於商業用途，以原始碼或二進制形式（修改或未修改）銷售 Python 的🄵本，或者以某種形式銷售🄵含 Python 的🄵品。當然，我們仍然會想要知道所有的 Python 商業用途。

請參🄵 授權頁面，查詢更深入的🄵明和 PSF 授權全文的連結。

Python 標🄵是🄵🄵商標，在某些情🄵下需要許可才能使用它。請參🄵商標使用政策以取得更多資訊。

1.1.4 當初🄵什🄵 Python 會被創造出來？

以下是由 Guido van Rossum 所撰寫，關於這一切如何開始的非常簡短的摘要：

> 我在 CWI 的 ABC 小組中擁有實作直譯語言方面的豐富經驗，而透過與該小組的合作，我學到了很多關於語言設計的知識。這是許多 Python 功能的起源，包括使用縮排進行陳述式分組以及納入非常高階的資料型🄵（🄵管在 Python 中的細節都已經不同）。
>
> 我對 ABC 語言有一些牢騷，但我也喜歡它的許多功能。想要擴充 ABC 語言（或其實作）來去除我的抱怨是不可能的。事實上，缺乏可擴充性就是它最大的問題之一。我有一些使用 Modula-2+ 的經驗，也與 Modula-3 的設計者交談過，🄵🄵讀了 Modula-3 的報告。Modula-3 就是用於例外及另外一些 Python 功能的語法和語義的起源。
>
> 我當時正在 CWI 的 Amoeba 分散式作業系統小組工作。我們需要一種比編寫 C 程式或 Bourne shell 🄵本更好的方法來進行系統管理，因🄵 Amoeba 有自己的系統呼叫介面，而它無法簡單地從 Bourne shell 進行存取。我在 Amoeba 中處理錯誤的經驗，使我深切地意識到例外作🄵程式設計語言功能的重要性。
>
> 我突然想到，一種具有類似 ABC 的語法但可以存取 Amoeba 系統呼叫的🄵本語言將能滿足該需求。我了解編寫 Amoeba 專用語言是愚蠢的，所以我🄵定，我需要一種可以廣泛擴充的語言。
>
> 在 1989 年的聖誕節假期，我有很多自由時間，所以我🄵定來嘗試一下。在接下來的一年🄵，雖然我大部分時間仍然在🄵此而努力，但 Python 在 Amoeba 專案中的使用得到了越來越多的成功，且同事們的回饋也使我🄵它增加了許多早期的改進。
>
> 在 1991 年 2 月，經過一年多的發展，我🄵定將它發表到 USENET。其他的記🄵都在 `Misc/HISTORY` 檔案中。

1.1.5 什🄵是 Python 擅長的事情？

Python 是一種高階的、用途廣泛的程式設計語言，可以用來解🄵許多不同類型的問題。

這個語言提供了一個大型的標准函式庫，涵蓋了字串處理（正規表示式、Unicode、檔案之間的差🄵計算）、網際網路協定（HTTP、FTP、SMTP、XML-RPC、POP、IMAP）、軟體工程（單元測試、日🄵記🄵、效能分析、剖析 Python 程式碼）以及作業系統介面（系統呼叫、檔案系統、TCP/IP 插座 (socket)）等領域。請查看 library-index 的目🄵，以了解可用的函式。此外，還有各式各樣的第三方擴充。請查詢 Python 套件索引 (Python Package Index) 來尋找你有興趣的套件。

1.1.6 Python 的版本編號系統是如何運作的？

Python 各版本會被編號🄵"A.B.C" 或"A.B"：

- *A* 🄵主要版本編號 -- 它只會在語言中有真正重大的變更時才會增加。
- *B* 🄵次要版本編號 -- 只有在影響範圍較小的變更出現時增加。
- *C* 🄵微小版本編號—- 會在每個錯誤修正發布 (bugfix release) 增加。

🄵非所有的發布版本都是錯誤修正發布版本。在一個新功能發布版本的准備階段，會發布一系列開發版本，標示🄵 alpha、beta 或候選發布版本 (release candidate)。Alpha 是介面尚未最終化的早期發布版本；看到兩個 alpha 發布版本之間的介面變更🄵不會令人意外。Beta 則更🄵穩定，保留了現有的介面，但可能會增加新的模組，而候選發布版本會被凍結，除了需要修正關鍵錯誤之外，不會再進行任何變更。

Alpha、beta 和候選發布版本都有一個額外的後綴：

- Alpha 版本的後綴是"aN"，其中 *N* 是某個較小的數字。
- Beta 版本的後綴是"bN"，其中 *N* 是某個較小的數字。
- 候選發布版本的後綴是"rcN"，其中 *N* 是某個較小的數字。

🄵句話🄵，所有標記🄵 *2.0aN* 的版本都在標記🄵 *2.0bN* 的版本之前，而 *2.0bN* 版本都在標記🄵 *2.0rcN* 的版本之前，而它們都是在 2.0 版之前。

你還可以找到帶有「+」後綴的版本編號，例如「2.2+」。這些是未發布的版本，直接從 CPython 的開發儲存庫被建置。實際上，在每一次的最終次要版本發布完成之後，版本編號將會被增加到下一個次要版本，□成□「a0」版，例如「2.4a0」。

See the Developer's Guide for more information about the development cycle, and **PEP 387** to learn more about Python's backward compatibility policy. See also the documentation for `sys.version`, `sys.hexversion`, and `sys.version_info`.

1.1.7 我要如何得到 Python 的原始碼□本？

最新的 Python 原始碼發行版永遠可以從 python.org 取得，在 https://www.python.org/downloads/。最新的開發中原始碼可以在 https://github.com/python/cpython/ 取得。

原始碼發行版是一個以 gzip 壓縮的 tar 檔，它包含完整的 C 原始碼、Sphinx 格式的□明文件、Python 函式庫模組、範例程式，以及幾個好用的可自由發行軟體。該原始碼在大多數 UNIX 平台上，都是可以立即編譯及運行的。

關於取得和編譯原始碼的詳細資訊，請參□ Python 開發人員指南中的"Getting Started" 段落。

1.1.8 我要如何取得 Python 的□明文件？

Python 目前穩定版本的標准□明文件可在 https://docs.python.org/3/ 找到。PDF、純文字和可下載的 HTML 版本也可在 https://docs.python.org/3/download.html 找到。

□明文件是以 reStructuredText 格式編寫，□由 Sphinx □明文件工具處理。□明文件的 reStructuredText 原始碼是 Python 原始碼發行版的一部分。

1.1.9 我從來□有寫過程式，有□有 Python 的教學？

有許多可用的教學和書籍。標准□明文件包括 tutorial-index。

要尋找 Python 程式設計初學者的資訊，包括教學資源列表，請參□初學者指南。

1.1.10 有□有 Python 專屬的新聞群組或郵件討論群？

有一個新聞群組 (newsgroup)，*comp.lang.python*，也有一個郵件討論群 (mailing list)，python-list。新聞群組和郵件討論群是彼此相通的——如果你能□讀新聞，則無需加入郵件討論群。*comp.lang.python* 的流量很高，每天會收到數百篇文章，而 Usenet 的讀者通常較能□處理這樣的文章數量。

新的軟體發布版本及事件的通知，可以在 comp.lang.python.announce 中找到，這是一個低流量的精選討論群，每天收到大約五篇文章。它也能從 python-announce 郵件討論群的頁面中訂□。

關於其他郵件討論群和新聞群組的更多資訊，可以在 https://www.python.org/community/lists/ 中找到。

1.1.11 如何取得 Python 的 beta 測試版本？

Alpha 和 beta 發布版本可以從 https://www.python.org/downloads/ 取得。所有的發布版本都會在 comp.lang.python 和 comp.lang.python.announce 新聞群組上宣布，也會在 Python 首頁 https://www.python.org/ 中宣布；RSS 新聞摘要也是可使用的。

你也可以藉由 Git 來存取 Python 的開發版本。更多詳細資訊，請參□ Python 開發人員指南。

1.1.12 如何提交 Python 的錯誤報告和修補程式？

要回報一個錯誤 (bug) 或提交一個修補程式 (patch)，請使用於 https://github.com/python/cpython/issues 的問題追□系統。

關於如何開發 Python 的更多資訊，請參□ Python 開發人員指南。

1.1.13 是否有關於 Python 的任何已出版文章可供參考？

也許最好是引用你最喜歡的關於 Python 的書。

最早討論 Python 的文章是在 1991 年寫的，但現在來看已經過時了。

> Guido van Rossum 和 Jelke de Boer，「使用 Python 程式設計語言互動式測試遠端伺服器」，CWI 季刊，第 4 卷，第 4 期（1991 年 12 月），阿姆斯特丹，第 283–303 頁。

1.1.14 有□有關於 Python 的書？

有，很多書已經出版，也有更多正在出版中的書。請參□ python.org 的 wiki 在 https://wiki.python.org/moin/PythonBooks 頁面中的書目清單。

你也可以在網路書店搜尋關鍵字「Python」，□過濾掉 Monty Python 的結果；或者可以搜尋「Python」和「語言」。

1.1.15 www.python.org 的真實位置在哪□？

Python 專案的基礎建設遍□世界各地，由 Python 基礎建設團隊管理。詳細資訊在此。

1.1.16 □什□要取名□ Python？

當 Guido van Rossum 開始實作 Python 時，他也正在□讀 1970 年代 BBC 喜劇節目「Monty Python 的飛行馬戲團」的出版劇本。Van Rossum 認□他需要一個簡短、獨特且略帶神秘的名字，因此他□定將該語言稱□ Python。

1.1.17 我需要喜歡「Monty Python 的飛行馬戲團」嗎？

不需要，但它有幫助。:)

1.2 在真實世界中的 Python

1.2.1 Python 的穩定性如何？

非常穩定。自從 1991 年開始，大約每隔 6 到 18 個月都會發布新的穩定版本，而且這看起來會繼續進行。從 3.9 版開始，Python 每隔 12 個月將會釋出一個新功能發行版本 (**PEP 602**)。

開發人員會釋出針對先前版本的錯誤修正發布版本，因此現有發布版本的穩定性會逐漸提高。錯誤修正發布版本是由版本編號的第三個部分表示（例如 3.5.3，3.6.2），這些版本會被用於改善穩定性；在錯誤修正發布版本中，只會包含針對已知問題的修正，□且會保證介面在一系列的錯誤修正發布版本中維持不變。

The latest stable releases can always be found on the Python download page. Python 3.x is the recommended version and supported by most widely used libraries. Python 2.x **is not maintained anymore**.

1.2.2 有多少人在使用 Python？

可能有幾百萬個使用者，但實際的數量是難以確定的。

Python 是可以免費下載的，所以不會有銷售數據，而且它可以從許多不同的網站取得，□與許多 Linux 發行版套裝在一起，所以下載次數的統計也無法反映完整的情□。

comp.lang.python 新聞群組非常活躍，但□非所有 Python 使用者都會在該群組發表文章或甚至□讀它。

1.2.3 有□有任何重要的專案使用 Python 完成開發？

要查看使用 Python 的專案清單，請參□ https://www.python.org/about/success。藉由查詢過去的 Python 會議記□可以看見來自許多不同公司和組織的貢獻。

備受矚目的 Python 專案包括 Mailman 郵件討論群管理員和 Zope 應用程式伺服器。有一些 Linux 發行版，最著名的是 Red Hat，已經用 Python 編寫了部分或全部的安裝程式及系統管理軟體。□部使用 Python 的公司包括 Google、Yahoo 和 Lucasfilm Ltd。

1.2.4 Python 未來預期會有哪些新的開發？

請至 https://peps.python.org/ 參□ Python 增□提案 (Python Enhancement Proposal, PEP)。PEP 是用來描述一項被建議的 Python 新功能的設計文件，它提供了簡潔的技術規範及基本原理。請尋找一篇名□「Python X.Y Release Schedule（發布時程表）」的 PEP，其中 X.Y 是一個尚未公開發布的版本。

新的開發會在 python-dev 郵件討論群中討論。

1.2.5 對 Python 提出不相容的變更建議是否適當？

一般來□，不適當。全世界已經有數百萬行 Python 程式碼，因此在語言中的任何變更，若會使現有程式的一小部分成□無效，它都是不被允許的。即使你可以提供轉□程式，仍然會有需要更新全部□明文件的問題；市面上已經有很多介紹 Python 的書，而我們不想一下子就把它們都變□無效。

如果一項功能必須被變更，那□一定要提供逐步升級的路徑。**PEP 5** 描述了要引進反向不相容 (backward-incompatible) 的變更，同時也要對使用者的擾亂最小化，所需遵循的程序。

1.2.6 Python 對於入門的程式設計師而言是否□好的語言？

是的。

學生們仍然普遍地會從一種程序語言和□態型□語言 (statically typed language) 開始入門，這些語言像是 Pascal、C，或是 C++ 或 Java 的某個子集。透過學習 Python 作□他們的第一個語言，學生們可能會學得更好。Python 具有非常簡單且一致的語法和一個大型的標准函式庫，最重要的是，在入門程式設計課程中使用 Python 可以讓學生專注於重要的程式設計技巧，例如問題的分解和資料型□的設計。使用 Python，可以快速地向學生介紹基本觀念，例如□圈和程序。他們甚至可能在第一堂課中就學到使用者自訂的物件。

對於以前從未進行過程式設計的學生來□，使用□態型□語言似乎是不自然的。它使學生必須掌握額外的□雜性，□□慢了課程的節奏。學生們正在試圖學著像電腦一樣思考、分解問題、設計一致的介面，□封裝資料。雖然從長遠來看，學習使用□態型□語言很重要，但在學生的第一堂程式設計課程中，它不一定是最好的課程主題。

Python 的許多其他面向使它成□一種很好的第一語言。像 Java 一樣，Python 有一個大型的標准函式庫，因此學生可以在課程的早期就被指派程式設計的專案，且這些專案能□做一些事情。指派的□容不會□限於標准的四功能計算機和平衡檢驗程式。透過使用標准函式庫，學生可以在學習程式設計基礎知識的同時，獲得處理真實應用程式的滿足感。使用標准函式庫還可以教導學生程式碼再使用 (code reuse) 的課題。像是 PyGame 等第三方模組也有助於延伸學生的學習領域。

Python 的互動式直譯器使學生能□在程式設計時測試語言的功能。他們可以開著一個運行直譯器的視窗，同時在另一個視窗中輸入他們的程式原始碼。如果他們不記得 list（串列）的 method（方法），他們可以像這樣做：

```
>>> L = []
>>> dir(L)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__',
'__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',
'__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']
>>> [d for d in dir(L) if '__' not in d]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
↪ 'sort']
```

(繼續下一頁)

(繼續上一頁)

```
>>> help(L.append)
Help on built-in function append:

append(...)
    L.append(object) -> None -- append object to end

>>> L.append(1)
>>> L
[1]
```

有了直譯器，當學生正在程式設計時，🄵明文件永遠都不會遠離他們。

Python 也有很好的 IDE。IDLE 是 Python 的一個跨平臺 IDE，它以 Python 編寫🄵使用 Tkinter。Emacs 使用者會很高興知道 Emacs 有一個非常好的 Python 模式。這些程式設計環境全部都能提供語法突顯 (syntax highlighting)、自動縮排，以及在編寫程式時存取互動式直譯器。要查看 Python 編輯環境的完整清單，請參🄵 Python wiki。

如果你想討論 Python 在教育領域中的使用，你可能會有興趣加入 edu-sig 郵件討論群。

CHAPTER 2

程式開發常見問答集

2.1 常見問題

2.1.1 是否有可以使用在程式碼階段, 具有中斷點, 步驟執行等功能的除錯器？

有的。

下面描述了幾個 Python 除錯器，🄵建函式 `breakpoint()` 允許你進入其中任何一個。

pdb 模組是一個簡單但足🄵的 Python 控制台模式除錯器。它是標準 Python 函式庫的一部分，🄵記🄵在函式庫參考手🄵中。你也可以參考 pdb 的程式碼作🄵範例來編寫自己的除錯器。

IDLE 交互式開發環境，它是標準 Python 發行版的一部分（通常作🄵 Tools/scripts/idle3 提供），包括一個圖形除錯器。

PythonWin 是一個 Python IDE，它包含一個基於 pdb 的 GUI 除錯器。PythonWin 除錯器🄵斷點著色🄵具有許多很酷的功能，例如除錯非 PythonWin 程式。PythonWin 作🄵 pywin32 專案的一部分和作🄵 ActivePython 的一部分發🄵。

Eric 是一個基於 PyQt 和 Scintilla 編輯元件所建構的 IDE。

trepan3k 是一個類似 gdb 的除錯器。

Visual Studio Code 是一個整合了版本控制軟體與除錯工具的 IDE。

有數個商業化 Python 整合化開發工具包含圖形除錯功能。這些包含：

- Wing IDE
- Komodo IDE
- PyCharm

2.1.2 有🄵有工具能🄵幫忙找 bug 或執行🄵態分析？

有的。

Pylint 和 Pyflakes 進行基本檢查以幫助你🄵早抓出錯誤。

🄵態型🄵檢查器，例如 Mypy、Pyre 和 Pytype 可以檢查 Python 原始碼中的型🄵提示。

2.1.3 如何從 Python 🄵本建立獨立的二進位檔案？

You don't need the ability to compile Python to C code if all you want is a stand-alone program that users can download and run without having to install the Python distribution first. There are a number of tools that determine the set of modules required by a program and bind these modules together with a Python binary to produce a single executable.

One is to use the freeze tool, which is included in the Python source tree as Tools/freeze. It converts Python byte code to C arrays; with a C compiler you can embed all your modules into a new program, which is then linked with the standard Python modules.

It works by scanning your source recursively for import statements (in both forms) and looking for the modules in the standard Python path as well as in the source directory (for built-in modules). It then turns the bytecode for modules written in Python into C code (array initializers that can be turned into code objects using the marshal module) and creates a custom-made config file that only contains those built-in modules which are actually used in the program. It then compiles the generated C code and links it with the rest of the Python interpreter to form a self-contained binary which acts exactly like your script.

以下套件可以幫助建立 console 和 GUI 可執行檔案：

- Nuitka（跨平台）
- PyInstaller（跨平台）
- PyOxidizer（跨平台）
- cx_Freeze（跨平台）
- py2app（僅限 macOS）
- py2exe（僅限 Windows）

2.1.4 Python 程式碼是否有編碼標準或風格指南？

是的。標准函式庫模組所需的編碼風格稱🄵 **PEP 8**。

2.2 核心語言

2.2.1 🄵什🄵當變數有值時，我仍得到錯誤訊息 UnboundLocalError？

It can be a surprise to get the `UnboundLocalError` in previously working code when it is modified by adding an assignment statement somewhere in the body of a function.

這段程式碼:

```
>>> x = 10
>>> def bar():
...     print(x)
...
>>> bar()
10
```

可以執行, 但是這段程式:

```
>>> x = 10
>>> def foo():
...     print(x)
...     x += 1
```

導致 `UnboundLocalError`：

```
>>> foo()
Traceback (most recent call last):
```

(繼續下一頁)

(繼續上一頁)

```
  ...
UnboundLocalError: local variable 'x' referenced before assignment
```

This is because when you make an assignment to a variable in a scope, that variable becomes local to that scope and shadows any similarly named variable in the outer scope. Since the last statement in foo assigns a new value to `x`, the compiler recognizes it as a local variable. Consequently when the earlier `print(x)` attempts to print the uninitialized local variable and an error results.

在上面的範例中，你可以透過將其聲明🄵全域變數來存取外部範圍變數：

```
>>> x = 10
>>> def foobar():
...     global x
...     print(x)
...     x += 1
...
>>> foobar()
10
```

This explicit declaration is required in order to remind you that (unlike the superficially analogous situation with class and instance variables) you are actually modifying the value of the variable in the outer scope:

```
>>> print(x)
11
```

你可以使用 `nonlocal` 關鍵字在巢狀作用域🄵做類似的事情：

```
>>> def foo():
...    x = 10
...    def bar():
...        nonlocal x
...        print(x)
...        x += 1
...    bar()
...    print(x)
...
>>> foo()
10
11
```

2.2.2 Python 的區域變數和全域變數有什🄵規則？

In Python, variables that are only referenced inside a function are implicitly global. If a variable is assigned a value anywhere within the function's body, it's assumed to be a local unless explicitly declared as global.

Though a bit surprising at first, a moment's consideration explains this. On one hand, requiring `global` for assigned variables provides a bar against unintended side-effects. On the other hand, if `global` was required for all global references, you'd be using `global` all the time. You'd have to declare as global every reference to a built-in function or to a component of an imported module. This clutter would defeat the usefulness of the `global` declaration for identifying side-effects.

2.2.3 🄵什🄵以不同的值在🄵圈中定義的 lambda 都回傳相同的結果？

假設你使用 for 🄵圈來定義幾個不同的 lambda（甚至是普通函式），例如：

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda: x**2)
```

這會提供一個包含五個計算 `x**2` 的 lambda 串列。你可能會預期在呼叫它時，它們會分🄵回傳 `0`、`1`、`4`、`9` 和 `16`，然而當你實際嘗試你會發現它們都回傳 `16`：

```
>>> squares[2]()
16
>>> squares[4]()
16
```

發生這種情🄵是因🄵 x 不是 lambda 的局部變數，而是在外部作用域中定義的，且是在呼叫 lambda 時才會存取它，🄵非於定義時就會存取。在🄵圈結束時，x 的值🄵 4，因此所有函式都回傳 4**2，即🄵 16。你還可以透過更改 x 的值來驗證這一點，🄵查看 lambda 運算式的結果如何變化：

```
>>> x = 8
>>> squares[2]()
64
```

🄵了避免這種情🄵，你需要將值保存在 lambda 的局部變數中，這樣它們就不會依賴於全域 x 的值：

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda n=x: n**2)
```

Here, n=x creates a new variable n local to the lambda and computed when the lambda is defined so that it has the same value that x had at that point in the loop. This means that the value of n will be 0 in the first lambda, 1 in the second, 2 in the third, and so on. Therefore each lambda will now return the correct result:

```
>>> squares[2]()
4
>>> squares[4]()
16
```

請注意，此行🄵🄵非 lambda 所特有，也適用於常規函式。

2.2.4 如何跨模組共享全域變數？

The canonical way to share information across modules within a single program is to create a special module (often called config or cfg). Just import the config module in all modules of your application; the module then becomes available as a global name. Because there is only one instance of each module, any changes made to the module object get reflected everywhere. For example:

config.py：

```
x = 0   # 'x' 配置設定的預設值
```

mod.py：

```
import config
config.x = 1
```

main.py：

```
import config
import mod
print(config.x)
```

請注意，出於同樣的原因，使用模組也是實作單例設計模式的基礎。

2.2.5 在模組中使用 import 的「最佳實踐」有哪些？

In general, don't use from modulename import *. Doing so clutters the importer's namespace, and makes it much harder for linters to detect undefined names.

Import modules at the top of a file. Doing so makes it clear what other modules your code requires and avoids questions of whether the module name is in scope. Using one import per line makes it easy to add and delete module imports, but using multiple imports per line uses less screen space.

It's good practice if you import modules in the following order:

1. 標准函式庫模組——例如 `sys`、`os`、`argparse`、`re`
2. third-party library modules (anything installed in Python's site-packages directory) -- e.g. `dateutil`, `requests`, `PIL.Image`
3. locally developed modules

It is sometimes necessary to move imports to a function or class to avoid problems with circular imports. Gordon McMillan says:

> Circular imports are fine where both modules use the "import <module>" form of import. They fail when the 2nd module wants to grab a name out of the first ("from module import name") and the import is at the top level. That's because names in the 1st are not yet available, because the first module is busy importing the 2nd.

In this case, if the second module is only used in one function, then the import can easily be moved into that function. By the time the import is called, the first module will have finished initializing, and the second module can do its import.

It may also be necessary to move imports out of the top level of code if some of the modules are platform-specific. In that case, it may not even be possible to import all of the modules at the top of the file. In this case, importing the correct modules in the corresponding platform-specific code is a good option.

Only move imports into a local scope, such as inside a function definition, if it's necessary to solve a problem such as avoiding a circular import or are trying to reduce the initialization time of a module. This technique is especially helpful if many of the imports are unnecessary depending on how the program executes. You may also want to move imports into a function if the modules are only ever used in that function. Note that loading a module the first time may be expensive because of the one time initialization of the module, but loading a module multiple times is virtually free, costing only a couple of dictionary lookups. Even if the module name has gone out of scope, the module is probably available in `sys.modules`.

2.2.6 Why are default values shared between objects?

This type of bug commonly bites neophyte programmers. Consider this function:

```
def foo(mydict={}):  # Danger: shared reference to one dict for all calls
    ... compute something ...
    mydict[key] = value
    return mydict
```

The first time you call this function, `mydict` contains a single item. The second time, `mydict` contains two items because when `foo()` begins executing, `mydict` starts out with an item already in it.

It is often expected that a function call creates new objects for default values. This is not what happens. Default values are created exactly once, when the function is defined. If that object is changed, like the dictionary in this example, subsequent calls to the function will refer to this changed object.

By definition, immutable objects such as numbers, strings, tuples, and `None`, are safe from change. Changes to mutable objects such as dictionaries, lists, and class instances can lead to confusion.

Because of this feature, it is good programming practice to not use mutable objects as default values. Instead, use `None` as the default value and inside the function, check if the parameter is `None` and create a new list/dictionary/whatever if it is. For example, don't write:

```
def foo(mydict={}):
    ...
```

但是：

```
def foo(mydict=None):
    if mydict is None:
        mydict = {}  # F區域命名空間建立一個新字典
```

This feature can be useful. When you have a function that's time-consuming to compute, a common technique is to cache the parameters and the resulting value of each call to the function, and return the cached value if the same value is requested again. This is called "memoizing", and can be implemented like this:

```
# Callers can only provide two parameters and optionally pass _cache by keyword
def expensive(arg1, arg2, *, _cache={}):
    if (arg1, arg2) in _cache:
        return _cache[(arg1, arg2)]

    # Calculate the value
    result = ... expensive computation ...
    _cache[(arg1, arg2)] = result           # Store result in the cache
    return result
```

你可以使用包含字典的全域變數而不是預設值；這取🄵於喜好。

2.2.7 How can I pass optional or keyword parameters from one function to another?

Collect the arguments using the `*` and `**` specifiers in the function's parameter list; this gives you the positional arguments as a tuple and the keyword arguments as a dictionary. You can then pass these arguments when calling another function by using `*` and `**`:

```
def f(x, *args, **kwargs):
    ...
    kwargs['width'] = '14.3c'
    ...
    g(x, *args, **kwargs)
```

2.2.8 引數 (arguments) 和參數 (parameters) 有什🄵區🄵？

參數由出現在函式定義中的名稱定義，而引數是呼叫函式時實際傳遞給函式的值。參數定義函式可以接受的引數種類。例如，給定以下函式定義：

```
def func(foo, bar=None, **kwargs):
    pass
```

foo、*bar* 和 *kwargs* 是 `func` 的參數。然而當呼叫 `func` 時，例如：

```
func(42, bar=314, extra=somevar)
```

`42`、`314` 和 `somevar` 是引數。

2.2.9 🄵什🄵更改 list 'y' 也會更改 list 'x'？

如果你寫了像這樣的程式碼：

```
>>> x = []
>>> y = x
>>> y.append(10)
>>> y
[10]
>>> x
[10]
```

你可能想知道🄵什🄵將一個元素附加到 `y` 時也會改變 `x`。

🄵生這個結果的原因有兩個：

1) Variables are simply names that refer to objects. Doing `y = x` doesn't create a copy of the list -- it creates a new variable `y` that refers to the same object `x` refers to. This means that there is only one object (the list), and both `x` and `y` refer to it.

2) list 是*mutable*，這意味著你可以變更它們的□容。

在呼叫 `append()` 之後，可變物件的□容從 `[]` 變成了 `[10]`。由於這兩個變數都參照同一個物件，因此使用任一名稱都可以存取修改後的值 `[10]`。

如果我們改□賦予一個不可變物件給 `x`：

```
>>> x = 5  # 整數□不可變的
>>> y = x
>>> x = x + 1  # 5 不可變，在這邊會建立一個新物件
>>> x
6
>>> y
5
```

we can see that in this case `x` and `y` are not equal anymore. This is because integers are *immutable*, and when we do `x = x + 1` we are not mutating the int `5` by incrementing its value; instead, we are creating a new object (the int `6`) and assigning it to `x` (that is, changing which object `x` refers to). After this assignment we have two objects (the ints `6` and `5`) and two variables that refer to them (`x` now refers to `6` but `y` still refers to `5`).

Some operations (for example `y.append(10)` and `y.sort()`) mutate the object, whereas superficially similar operations (for example `y = y + [10]` and `sorted(y)`) create a new object. In general in Python (and in all cases in the standard library) a method that mutates an object will return `None` to help avoid getting the two types of operations confused. So if you mistakenly write `y.sort()` thinking it will give you a sorted copy of `y`, you'll instead end up with `None`, which will likely cause your program to generate an easily diagnosed error.

However, there is one class of operations where the same operation sometimes has different behaviors with different types: the augmented assignment operators. For example, `+=` mutates lists but not tuples or ints (`a_list += [1, 2, 3]` is equivalent to `a_list.extend([1, 2, 3])` and mutates `a_list`, whereas `some_tuple += (1, 2, 3)` and `some_int += 1` create new objects).

□句話□：

- If we have a mutable object (`list`, `dict`, `set`, etc.), we can use some specific operations to mutate it and all the variables that refer to it will see the change.
- If we have an immutable object (`str`, `int`, `tuple`, etc.), all the variables that refer to it will always see the same value, but operations that transform that value into a new value always return a new object.

If you want to know if two variables refer to the same object or not, you can use the `is` operator, or the built-in function `id()`.

2.2.10 如何編寫帶有輸出參數的函式（透過傳參照呼叫 (call by reference)）？

請記住，在 Python 中引數是透過賦值傳遞的。由於賦值只是建立對物件的參照，因此呼叫者和被呼叫者的引數名稱之間□有□名，因此本身□有傳參照呼叫。你可以透過多種方式實作所需的效果。

1) 透過回傳結果的元組：

```
>>> def func1(a, b):
...     a = 'new-value'        # a 和 b □區域名稱
...     b = b + 1              # 賦值到新物件
...     return a, b            # 回傳新值
...
>>> x, y = 'old-value', 99
>>> func1(x, y)
('new-value', 100)
```

這幾乎都會是最清楚的方案。

2) By using global variables. This isn't thread-safe, and is not recommended.

3) 透過傳遞一個可變的（可於原地 (in-place) 改變的）物件：

```
>>> def func2(a):
...     a[0] = 'new-value'     # 'a' 參照一個可變的串列
...     a[1] = a[1] + 1        # 改變共享的物件
...
>>> args = ['old-value', 99]
>>> func2(args)
>>> args
['new-value', 100]
```

4) By passing in a dictionary that gets mutated:

```
>>> def func3(args):
...     args['a'] = 'new-value'     # args is a mutable dictionary
...     args['b'] = args['b'] + 1   # change it in-place
...
>>> args = {'a': 'old-value', 'b': 99}
>>> func3(args)
>>> args
{'a': 'new-value', 'b': 100}
```

5) Or bundle up values in a class instance:

```
>>> class Namespace:
...     def __init__(self, /, **args):
...         for key, value in args.items():
...             setattr(self, key, value)
...
>>> def func4(args):
...     args.a = 'new-value'        # args is a mutable Namespace
...     args.b = args.b + 1         # change object in-place
...
>>> args = Namespace(a='old-value', b=99)
>>> func4(args)
>>> vars(args)
{'a': 'new-value', 'b': 100}
```

There's almost never a good reason to get this complicated.

Your best choice is to return a tuple containing the multiple results.

2.2.11 How do you make a higher order function in Python?

You have two choices: you can use nested scopes or you can use callable objects. For example, suppose you wanted to define `linear(a,b)` which returns a function `f(x)` that computes the value `a*x+b`. Using nested scopes:

```
def linear(a, b):
    def result(x):
        return a * x + b
    return result
```

或者使用可呼叫物件：

```
class linear:

    def __init__(self, a, b):
        self.a, self.b = a, b

    def __call__(self, x):
        return self.a * x + self.b
```

在這兩種情F下：

```
taxes = linear(0.3, 2)
```

gives a callable object where `taxes(10e6) == 0.3 * 10e6 + 2`.

The callable object approach has the disadvantage that it is a bit slower and results in slightly longer code. However, note that a collection of callables can share their signature via inheritance:

```
class exponential(linear):
    # __init__ inherited
    def __call__(self, x):
        return self.a * (x ** self.b)
```

物件可以封裝多個方法的狀態：

```
class counter:

    value = 0

    def set(self, x):
        self.value = x

    def up(self):
        self.value = self.value + 1

    def down(self):
        self.value = self.value - 1

count = counter()
inc, dec, reset = count.up, count.down, count.set
```

這🄵的 `inc()`、`dec()` 和 `reset()` 就像共享相同計數變數的函式一樣。

2.2.12 如何在 Python 中🄵🄵物件？

In general, try `copy.copy()` or `copy.deepcopy()` for the general case. Not all objects can be copied, but most can.

Some objects can be copied more easily. Dictionaries have a `copy()` method:

```
newdict = olddict.copy()
```

序列可以透過切片 (slicing) 🄵🄵：

```
new_l = l[:]
```

2.2.13 如何找到物件的方法或屬性？

For an instance `x` of a user-defined class, `dir(x)` returns an alphabetized list of the names containing the instance attributes and methods and attributes defined by its class.

2.2.14 我的程式碼如何發現物件的名稱？

Generally speaking, it can't, because objects don't really have names. Essentially, assignment always binds a name to a value; the same is true of `def` and `class` statements, but in that case the value is a callable. Consider the following code:

```
>>> class A:
...     pass
...
>>> B = A
>>> a = B()
```

(繼續下一頁)

(繼續上一頁)

```
>>> b = a
>>> print(b)
<__main__.A object at 0x16D07CC>
>>> print(a)
<__main__.A object at 0x16D07CC>
```

Arguably the class has a name: even though it is bound to two names and invoked through the name `B` the created instance is still reported as an instance of class `A`. However, it is impossible to say whether the instance's name is `a` or `b`, since both names are bound to the same value.

Generally speaking it should not be necessary for your code to "know the names" of particular values. Unless you are deliberately writing introspective programs, this is usually an indication that a change of approach might be beneficial.

在 comp.lang.python 中，Fredrik Lundh 曾針對這個問題給出了一個極好的比喻：

> 就像你在門廊上發現的那□□的名字一樣：□（物件）本身不能告訴你它的名字，它也不關心 - 所以找出它叫什□的唯一方法是詢問所有鄰居（命名空間）是否是他們的□（物件）...
>
> .... 如果你發現它有很多名字，或者根本□有名字，請不要感到驚訝！

2.2.15 逗號運算子的優先級是什□？

Comma is not an operator in Python. Consider this session:

```
>>> "a" in "b", "a"
(False, 'a')
```

Since the comma is not an operator, but a separator between expressions the above is evaluated as if you had entered:

```
("a" in "b"), "a"
```

而不是：

```
"a" in ("b", "a")
```

The same is true of the various assignment operators (=, += etc). They are not truly operators but syntactic delimiters in assignment statements.

2.2.16 是否有等效於 C 的"?:" 三元運算子？

有的，語法如下：

```
[on_true] if [expression] else [on_false]

x, y = 50, 25
small = x if x < y else y
```

Before this syntax was introduced in Python 2.5, a common idiom was to use logical operators:

```
[expression] and [on_true] or [on_false]
```

However, this idiom is unsafe, as it can give wrong results when *on_true* has a false boolean value. Therefore, it is always better to use the `... if ... else ...` form.

2.2.17 Is it possible to write obfuscated one-liners in Python?

Yes. Usually this is done by nesting `lambda` within `lambda`. See the following three examples, slightly adapted from Ulf Bartelt:

```
from functools import reduce

# Primes < 1000
print(list(filter(None,map(lambda y:y*reduce(lambda x,y:x*y!=0,
map(lambda x,y=y:y%x,range(2,int(pow(y,0.5)+1))),1),range(2,1000)))))

# First 10 Fibonacci numbers
print(list(map(lambda x,f=lambda x,f:(f(x-1,f)+f(x-2,f)) if x>1 else 1:
f(x,f), range(10))))

# Mandelbrot set
print((lambda Ru,Ro,Iu,Io,IM,Sx,Sy:reduce(lambda x,y:x+'\n'+y,map(lambda y,
Iu=Iu,Io=Io,Ru=Ru,Ro=Ro,Sy=Sy,L=lambda yc,Iu=Iu,Io=Io,Ru=Ru,Ro=Ro,i=IM,
Sx=Sx,Sy=Sy:reduce(lambda x,y:x+y,map(lambda x,xc=Ru,yc=yc,Ru=Ru,Ro=Ro,
i=i,Sx=Sx,F=lambda xc,yc,x,y,k,f=lambda xc,yc,x,y,k,f:(k<=0)or (x*x+y*y
>=4.0) or 1+f(xc,yc,x*x-y*y+xc,2.0*x*y+yc,k-1,f):f(xc,yc,x,y,k,f):chr(
64+F(Ru+x*(Ro-Ru)/Sx,yc,0,0,i)),range(Sx))):L(Iu+y*(Io-Iu)/Sy),range(Sy
))))(-2.1, 0.7, -1.2, 1.2, 30, 80, 24))
#    \___ ___/  \___ ___/  |   |   |__ lines on screen
#        V          V      |   |______ columns on screen
#        |          |      |__________ maximum of "iterations"
#        |          |_________________ range on y axis
#        |____________________________ range on x axis
```

孩子們，不要在家□嘗試這個！

2.2.18 函式參數串列中的斜□ (/) 是什□意思？

A slash in the argument list of a function denotes that the parameters prior to it are positional-only. Positional-only parameters are the ones without an externally usable name. Upon calling a function that accepts positional-only parameters, arguments are mapped to parameters based solely on their position. For example, `divmod()` is a function that accepts positional-only parameters. Its documentation looks like this:

```
>>> help(divmod)
Help on built-in function divmod in module builtins:

divmod(x, y, /)
    Return the tuple (x//y, x%y).  Invariant: div*y + mod == x.
```

參數串列最後的斜□表示兩個參數都是僅限位置參數。因此使用關鍵字引數呼叫 `divmod()` 會導致錯誤：

```
>>> divmod(x=3, y=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: divmod() takes no keyword arguments
```

2.3 數字和字串

2.3.1 如何指定十六進位和八進位整數？

To specify an octal digit, precede the octal value with a zero, and then a lower or uppercase ”o”. For example, to set the variable ”a” to the octal value ”10” (8 in decimal), type:

```
>>> a = 0o10
>>> a
8
```

Hexadecimal is just as easy. Simply precede the hexadecimal number with a zero, and then a lower or uppercase ”x”. Hexadecimal digits can be specified in lower or uppercase. For example, in the Python interpreter:

```
>>> a = 0xa5
>>> a
165
>>> b = 0XB2
>>> b
178
```

2.3.2 🄵什🄵 -22 // 10 回傳 -3？

It's primarily driven by the desire that `i % j` have the same sign as `j`. If you want that, and also want:

```
i == (i // j) * j + (i % j)
```

那🄵整數除法必須回傳向下取整的結果。C 還要求保留 該識🄵性，然後截斷 `i // j` 的編譯器需要使 `i % j` 具有與 `i` 相同的符號。

There are few real use cases for `i % j` when `j` is negative. When `j` is positive, there are many, and in virtually all of them it's more useful for `i % j` to be `>= 0`. If the clock says 10 now, what did it say 200 hours ago? `-190 % 12 == 2` is useful; `-190 % 12 == -10` is a bug waiting to bite.

2.3.3 How do I get int literal attribute instead of SyntaxError?

嘗試以正常方式查找 `int` 字面值屬性會給出一個 `SyntaxError`，因🄵句點被視🄵小數點：

```
>>> 1.__class__
  File "<stdin>", line 1
  1.__class__
   ^
SyntaxError: invalid decimal literal
```

解🄵方式是用空格或圓括號將字面值與句點分開。

```
>>> 1 .__class__
<class 'int'>
>>> (1).__class__
<class 'int'>
```

2.3.4 如何將字串轉🄵🄵數字？

For integers, use the built-in `int()` type constructor, e.g. `int('144') == 144`. Similarly, `float()` converts to a floating-point number, e.g. `float('144') == 144.0`.

By default, these interpret the number as decimal, so that `int('0144') == 144` holds true, and `int('0x144')` raises `ValueError`. `int(string, base)` takes the base to convert from as a second optional argument, so `int( '0x144', 16) == 324`. If the base is specified as 0, the number is interpreted using Python's rules: a leading '0o' indicates octal, and '0x' indicates a hex number.

Do not use the built-in function `eval()` if all you need is to convert strings to numbers. `eval()` will be significantly slower and it presents a security risk: someone could pass you a Python expression that might have unwanted side effects. For example, someone could pass `__import__('os').system("rm -rf $HOME")` which would erase your home directory.

`eval()` also has the effect of interpreting numbers as Python expressions, so that e.g. `eval('09')` gives a syntax error because Python does not allow leading '0' in a decimal number (except '0').

2.3.5 如何將數字轉🄵🄵字串？

To convert, e.g., the number `144` to the string `'144'`, use the built-in type constructor `str()`. If you want a hexadecimal or octal representation, use the built-in functions `hex()` or `oct()`. For fancy formatting, see the f-strings and formatstrings sections, e.g. `"{:04d}".format(144)` yields `'0144'` and `"{:.3f}".format(1.0/ 3.0)` yields `'0.333'`.

2.3.6 如何原地修改字串？

這□辦法做到，因□字串是不可變的。在大多數情□下，你應以要拿來組裝的各個部分建構出一個新字串。但是如果你需要一個能□原地修改 unicode 資料的物件，請嘗試使用 `io.StringIO` 物件或 `array` 模組：

```
>>> import io
>>> s = "Hello, world"
>>> sio = io.StringIO(s)
>>> sio.getvalue()
'Hello, world'
>>> sio.seek(7)
7
>>> sio.write("there!")
6
>>> sio.getvalue()
'Hello, there!'

>>> import array
>>> a = array.array('w', s)
>>> print(a)
array('w', 'Hello, world')
>>> a[0] = 'y'
>>> print(a)
array('w', 'yello, world')
>>> a.tounicode()
'yello, world'
```

2.3.7 如何使用字串呼叫函式/方法？

有各式各樣的技法。

- The best is to use a dictionary that maps strings to functions. The primary advantage of this technique is that the strings do not need to match the names of the functions. This is also the primary technique used to emulate a case construct:

```
def a():
    pass

def b():
    pass

dispatch = {'go': a, 'stop': b}  # Note lack of parens for funcs

dispatch[get_input()]()  # Note trailing parens to call function
```

- 使用□建函式 `getattr()`：

```
import foo
getattr(foo, 'bar')()
```

請注意 `getattr()` 適用於任何物件，包括類□、類□實例、模組等。

This is used in several places in the standard library, like this:

```
class Foo:
    def do_foo(self):
        ...

    def do_bar(self):
        ...

```

(繼續下一頁)

(繼續上一頁)

```
f = getattr(foo_instance, 'do_' + opname)
f()
```

- 使用 `locals()` 解析函式名稱：

```
def myFunc():
    print("hello")

fname = "myFunc"

f = locals()[fname]
f()
```

2.3.8 是否有與 Perl 的 chomp() 等效的方法，能用於從字串中🄵除後綴的🄵行符號？

You can use `S.rstrip("\r\n")` to remove all occurrences of any line terminator from the end of the string `S` without removing other trailing whitespace. If the string `S` represents more than one line, with several empty lines at the end, the line terminators for all the blank lines will be removed:

```
>>> lines = ("line 1 \r\n"
...          "\r\n"
...          "\r\n")
>>> lines.rstrip("\n\r")
'line 1 '
```

Since this is typically only desired when reading text one line at a time, using `S.rstrip()` this way works well.

2.3.9 是否有 scanf() 或 sscanf() 的等效方法？

Not as such.

For simple input parsing, the easiest approach is usually to split the line into whitespace-delimited words using the `split()` method of string objects and then convert decimal strings to numeric values using `int()` or `float()`. `split()` supports an optional "sep" parameter which is useful if the line uses something other than whitespace as a separator.

For more complicated input parsing, regular expressions are more powerful than C's `sscanf` and better suited for the task.

2.3.10 'UnicodeDecodeError' 或'UnicodeEncodeErro' 錯誤是什🄵意思？

請參🄵 unicode-howto。

2.3.11 Can I end a raw string with an odd number of backslashes?

A raw string ending with an odd number of backslashes will escape the string's quote:

```
>>> r'C:\this\will\not\work\'
  File "<stdin>", line 1
    r'C:\this\will\not\work\'
         ^
SyntaxError: unterminated string literal (detected at line 1)
```

There are several workarounds for this. One is to use regular strings and double the backslashes:

```
>>> 'C:\\this\\will\\work\\'
'C:\\this\\will\\work\\'
```

Another is to concatenate a regular string containing an escaped backslash to the raw string:

```
>>> r'C:\this\will\work' '\\'
'C:\\this\\will\\work\\'
```

It is also possible to use `os.path.join()` to append a backslash on Windows:

```
>>> os.path.join(r'C:\this\will\work', '')
'C:\\this\\will\\work\\'
```

Note that while a backslash will ”escape” a quote for the purposes of determining where the raw string ends, no escaping occurs when interpreting the value of the raw string. That is, the backslash remains present in the value of the raw string:

```
>>> r'backslash\'preserved'
"backslash\\'preserved"
```

Also see the specification in the language reference.

2.4 Performance

2.4.1 我的程式太慢了。我該如何加快速度？

That's a tough one, in general. First, here are a list of things to remember before diving further:

- Performance characteristics vary across Python implementations. This FAQ focuses on *CPython*.
- Behaviour can vary across operating systems, especially when talking about I/O or multi-threading.
- You should always find the hot spots in your program *before* attempting to optimize any code (see the `profile` module).
- Writing benchmark scripts will allow you to iterate quickly when searching for improvements (see the `timeit` module).
- It is highly recommended to have good code coverage (through unit testing or any other technique) before potentially introducing regressions hidden in sophisticated optimizations.

That being said, there are many tricks to speed up Python code. Here are some general principles which go a long way towards reaching acceptable performance levels:

- Making your algorithms faster (or changing to faster ones) can yield much larger benefits than trying to sprinkle micro-optimization tricks all over your code.
- Use the right data structures. Study documentation for the bltin-types and the `collections` module.
- When the standard library provides a primitive for doing something, it is likely (although not guaranteed) to be faster than any alternative you may come up with. This is doubly true for primitives written in C, such as builtins and some extension types. For example, be sure to use either the `list.sort()` built-in method or the related `sorted()` function to do sorting (and see the sortinghowto for examples of moderately advanced usage).
- Abstractions tend to create indirections and force the interpreter to work more. If the levels of indirection outweigh the amount of useful work done, your program will be slower. You should avoid excessive abstraction, especially under the form of tiny functions or methods (which are also often detrimental to readability).

If you have reached the limit of what pure Python can allow, there are tools to take you further away. For example, Cython can compile a slightly modified version of Python code into a C extension, and can be used on many different platforms. Cython can take advantage of compilation (and optional type annotations) to make your code significantly faster than when interpreted. If you are confident in your C programming skills, you can also write a C extension module yourself.

也參考

有個 wiki 頁面專門介紹效能改進小提示。

2.4.2 What is the most efficient way to concatenate many strings together?

`str` and `bytes` objects are immutable, therefore concatenating many strings together is inefficient as each concatenation creates a new object. In the general case, the total runtime cost is quadratic in the total string length.

To accumulate many `str` objects, the recommended idiom is to place them into a list and call `str.join()` at the end:

```
chunks = []
for s in my_strings:
    chunks.append(s)
result = ''.join(chunks)
```

(another reasonably efficient idiom is to use `io.StringIO`)

To accumulate many `bytes` objects, the recommended idiom is to extend a `bytearray` object using in-place concatenation (the += operator):

```
result = bytearray()
for b in my_bytes_objects:
    result += b
```

2.5 Sequences (Tuples/Lists)

2.5.1 How do I convert between tuples and lists?

型□建構函式 `tuple(seq)` 將任何序列（實際上是任何可□代物件）轉□□具有相同順序的相同項的元組。

For example, `tuple([1, 2, 3])` yields `(1, 2, 3)` and `tuple('abc')` yields `('a', 'b', 'c')`. If the argument is a tuple, it does not make a copy but returns the same object, so it is cheap to call `tuple()` when you aren't sure that an object is already a tuple.

The type constructor `list(seq)` converts any sequence or iterable into a list with the same items in the same order. For example, `list((1, 2, 3))` yields `[1, 2, 3]` and `list('abc')` yields `['a', 'b', 'c']`. If the argument is a list, it makes a copy just like `seq[:]` would.

2.5.2 What's a negative index?

Python sequences are indexed with positive numbers and negative numbers. For positive numbers 0 is the first index 1 is the second index and so forth. For negative indices -1 is the last index and -2 is the penultimate (next to last) index and so forth. Think of `seq[-n]` as the same as `seq[len(seq)-n]`.

Using negative indices can be very convenient. For example `S[:-1]` is all of the string except for its last character, which is useful for removing the trailing newline from a string.

2.5.3 How do I iterate over a sequence in reverse order?

Use the `reversed()` built-in function:

```
for x in reversed(sequence):
    ...  # do something with x ...
```

This won't touch your original sequence, but build a new copy with reversed order to iterate over.

2.5.4 How do you remove duplicates from a list?

See the Python Cookbook for a long discussion of many ways to do this:

> https://code.activestate.com/recipes/52560/

If you don't mind reordering the list, sort it and then scan from the end of the list, deleting duplicates as you go:

```
if mylist:
    mylist.sort()
    last = mylist[-1]
    for i in range(len(mylist)-2, -1, -1):
        if last == mylist[i]:
            del mylist[i]
        else:
            last = mylist[i]
```

If all elements of the list may be used as set keys (i.e. they are all *hashable*) this is often faster

```
mylist = list(set(mylist))
```

This converts the list into a set, thereby removing duplicates, and then back into a list.

2.5.5 How do you remove multiple items from a list

As with removing duplicates, explicitly iterating in reverse with a delete condition is one possibility. However, it is easier and faster to use slice replacement with an implicit or explicit forward iteration. Here are three variations.:

```
mylist[:] = filter(keep_function, mylist)
mylist[:] = (x for x in mylist if keep_condition)
mylist[:] = [x for x in mylist if keep_condition]
```

The list comprehension may be fastest.

2.5.6 How do you make an array in Python?

Use a list:

```
["this", 1, "is", "an", "array"]
```

Lists are equivalent to C or Pascal arrays in their time complexity; the primary difference is that a Python list can contain objects of many different types.

The `array` module also provides methods for creating arrays of fixed types with compact representations, but they are slower to index than lists. Also note that NumPy and other third party packages define array-like structures with various characteristics as well.

To get Lisp-style linked lists, you can emulate *cons cells* using tuples:

```
lisp_list = ("like",  ("this",  ("example", None) ) )
```

If mutability is desired, you could use lists instead of tuples. Here the analogue of a Lisp *car* is `lisp_list[0]` and the analogue of *cdr* is `lisp_list[1]`. Only do this if you're sure you really need to, because it's usually a lot slower than using Python lists.

2.5.7 How do I create a multidimensional list?

You probably tried to make a multidimensional array like this:

```
>>> A = [[None] * 2] * 3
```

如果你印出它，這看起來是正確的：

```
>>> A
[[None, None], [None, None], [None, None]]
```

But when you assign a value, it shows up in multiple places:

```
>>> A[0][0] = 5
>>> A
[[5, None], [5, None], [5, None]]
```

The reason is that replicating a list with `*` doesn't create copies, it only creates references to the existing objects. The `*3` creates a list containing 3 references to the same list of length two. Changes to one row will show in all rows, which is almost certainly not what you want.

The suggested approach is to create a list of the desired length first and then fill in each element with a newly created list:

```
A = [None] * 3
for i in range(3):
    A[i] = [None] * 2
```

This generates a list containing 3 different lists of length two. You can also use a list comprehension:

```
w, h = 2, 3
A = [[None] * w for i in range(h)]
```

Or, you can use an extension that provides a matrix datatype; NumPy is the best known.

2.5.8 How do I apply a method or function to a sequence of objects?

To call a method or function and accumulate the return values is a list, a *list comprehension* is an elegant solution:

```
result = [obj.method() for obj in mylist]

result = [function(obj) for obj in mylist]
```

To just run the method or function without saving the return values, a plain `for` loop will suffice:

```
for obj in mylist:
    obj.method()

for obj in mylist:
    function(obj)
```

2.5.9 Why does a_tuple[i] += ['item'] raise an exception when the addition works?

This is because of a combination of the fact that augmented assignment operators are *assignment* operators, and the difference between mutable and immutable objects in Python.

This discussion applies in general when augmented assignment operators are applied to elements of a tuple that point to mutable objects, but we'll use a `list` and `+=` as our exemplar.

If you wrote:

```
>>> a_tuple = (1, 2)
>>> a_tuple[0] += 1
Traceback (most recent call last):
   ...
TypeError: 'tuple' object does not support item assignment
```

The reason for the exception should be immediately clear: `1` is added to the object `a_tuple[0]` points to (`1`), producing the result object, `2`, but when we attempt to assign the result of the computation, `2`, to element `0` of the tuple, we get an error because we can't change what an element of a tuple points to.

Under the covers, what this augmented assignment statement is doing is approximately this:

```
>>> result = a_tuple[0] + 1
>>> a_tuple[0] = result
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

It is the assignment part of the operation that produces the error, since a tuple is immutable.

When you write something like:

```
>>> a_tuple = (['foo'], 'bar')
>>> a_tuple[0] += ['item']
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

The exception is a bit more surprising, and even more surprising is the fact that even though there was an error, the append worked:

```
>>> a_tuple[0]
['foo', 'item']
```

To see why this happens, you need to know that (a) if an object implements an `__iadd__()` magic method, it gets called when the += augmented assignment is executed, and its return value is what gets used in the assignment statement; and (b) for lists, `__iadd__()` is equivalent to calling `extend()` on the list and returning the list. That's why we say that for lists, += is a "shorthand" for `list.extend()`:

```
>>> a_list = []
>>> a_list += [1]
>>> a_list
[1]
```

這等價於：

```
>>> result = a_list.__iadd__([1])
>>> a_list = result
```

The object pointed to by a_list has been mutated, and the pointer to the mutated object is assigned back to `a_list`. The end result of the assignment is a no-op, since it is a pointer to the same object that `a_list` was previously pointing to, but the assignment still happens.

因此，在我們的元組範例中，發生的事情等同於：

```
>>> result = a_tuple[0].__iadd__(['item'])
>>> a_tuple[0] = result
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

The `__iadd__()` succeeds, and thus the list is extended, but even though `result` points to the same object that `a_tuple[0]` already points to, that final assignment still results in an error, because tuples are immutable.

2.5.10 我想做一個🄵雜的排序：你能用 Python 做一個 Schwartzian 變🄵嗎？

The technique, attributed to Randal Schwartz of the Perl community, sorts the elements of a list by a metric which maps each element to its "sort value". In Python, use the `key` argument for the `list.sort()` method:

```
Isorted = L[:]
Isorted.sort(key=lambda s: int(s[10:15]))
```

2.5.11 如何根據另一個串列中的值對一個串列進行排序？

將它們合□到一個元組□代器中，對結果的串列進行排序，然後挑選出你想要的元素。

```
>>> list1 = ["what", "I'm", "sorting", "by"]
>>> list2 = ["something", "else", "to", "sort"]
>>> pairs = zip(list1, list2)
>>> pairs = sorted(pairs)
>>> pairs
[("I'm", 'else'), ('by', 'sort'), ('sorting', 'to'), ('what', 'something')]
>>> result = [x[1] for x in pairs]
>>> result
['else', 'sort', 'to', 'something']
```

2.6 物件

2.6.1 什□是類□ (class)？

A class is the particular object type created by executing a class statement. Class objects are used as templates to create instance objects, which embody both the data (attributes) and code (methods) specific to a datatype.

A class can be based on one or more other classes, called its base class(es). It then inherits the attributes and methods of its base classes. This allows an object model to be successively refined by inheritance. You might have a generic `Mailbox` class that provides basic accessor methods for a mailbox, and subclasses such as `MboxMailbox`, `MaildirMailbox`, `OutlookMailbox` that handle various specific mailbox formats.

2.6.2 什□是方法 (method)？

A method is a function on some object `x` that you normally call as `x.name(arguments...)`. Methods are defined as functions inside the class definition:

```
class C:
    def meth(self, arg):
        return arg * 2 + self.attribute
```

2.6.3 什□是 self？

Self 只是方法第一個引數的約定名稱。對於所定義類□的某個實例 `x`，一個定義□ `meth(self, a, b, c)` 的方法應該以 `x.meth(a, b, c)` 形式來呼叫；被呼叫的方法會認□它是以 `meth(x, a, b, c)` 來呼叫的。

另請參□□何「*self*」在方法 *(method)* 定義和呼叫時一定要明確使用？。

2.6.4 How do I check if an object is an instance of a given class or of a subclass of it?

Use the built-in function `isinstance(obj, cls)`. You can check if an object is an instance of any of a number of classes by providing a tuple instead of a single class, e.g. `isinstance(obj, (class1, class2, ...))`, and can also check whether an object is one of Python's built-in types, e.g. `isinstance(obj, str)` or `isinstance(obj, (int, float, complex))`.

請注意，`isinstance()` 還會檢查來自抽象基底類□ *(abstract base class)* 的□擬繼承。因此對已□□類□的檢驗會回傳 `True`，即使□有直接或間接繼承自它。要測試「真正繼承」，請掃描該類□的*MRO*：

```
from collections.abc import Mapping

class P:
     pass

```

(繼續下一頁)

(繼續上一頁)

```
class C(P):
    pass

Mapping.register(P)
```

```
>>> c = C()
>>> isinstance(c, C)        # 直接
True
>>> isinstance(c, P)        # 間接
True
>>> isinstance(c, Mapping)  # 🄵擬
True

# 實際的繼承鏈結
>>> type(c).__mro__
(<class 'C'>, <class 'P'>, <class 'object'>)

# 「真正繼承」的檢驗
>>> Mapping in type(c).__mro__
False
```

Note that most programs do not use `isinstance()` on user-defined classes very often. If you are developing the classes yourself, a more proper object-oriented style is to define methods on the classes that encapsulate a particular behaviour, instead of checking the object's class and doing a different thing based on what class it is. For example, if you have a function that does something:

```
def search(obj):
    if isinstance(obj, Mailbox):
        ...  # 搜尋信箱的程式碼
    elif isinstance(obj, Document):
        ...  # 搜尋文件的程式碼
    elif ...
```

更好的方法是在所有類🄵上定義一個 `search()` 方法然後呼叫它：

```
class Mailbox:
    def search(self):
        ...  # 搜尋信箱的程式碼

class Document:
    def search(self):
        ...  # 搜尋文件的程式碼

obj.search()
```

2.6.5 What is delegation?

Delegation is an object oriented technique (also called a design pattern). Let's say you have an object `x` and want to change the behaviour of just one of its methods. You can create a new class that provides a new implementation of the method you're interested in changing and delegates all other methods to the corresponding method of `x`.

Python programmers can easily implement delegation. For example, the following class implements a class that behaves like a file but converts all written data to uppercase:

```
class UpperOut:

    def __init__(self, outfile):
        self._outfile = outfile

    def write(self, s):
```

(繼續下一頁)

(繼續上一頁)

```
        self._outfile.write(s.upper())

    def __getattr__(self, name):
        return getattr(self._outfile, name)
```

Here the `UpperOut` class redefines the `write()` method to convert the argument string to uppercase before calling the underlying `self._outfile.write()` method. All other methods are delegated to the underlying `self._outfile` object. The delegation is accomplished via the `__getattr__()` method; consult the language reference for more information about controlling attribute access.

Note that for more general cases delegation can get trickier. When attributes must be set as well as retrieved, the class must define a `__setattr__()` method too, and it must do so carefully. The basic implementation of `__setattr__()` is roughly equivalent to the following:

```
class X:
    ...
    def __setattr__(self, name, value):
        self.__dict__[name] = value
    ...
```

許多 `__setattr__()` 的實作會呼叫 `object.__setattr__()` 以設定 self 的屬性，而不會導致無限遞🄵。

```
class X:
    def __setattr__(self, name, value):
        # 自訂邏輯放在這🄵...
        object.__setattr__(self, name, value)
```

Alternatively, it is possible to set attributes by inserting entries into `self.__dict__` directly.

2.6.6 How do I call a method defined in a base class from a derived class that extends it?

使用🄵建的 `super()` 函式：

```
class Derived(Base):
    def meth(self):
        super().meth()  # 呼叫 Base.meth
```

In the example, `super()` will automatically determine the instance from which it was called (the `self` value), look up the *method resolution order* (MRO) with `type(self).__mro__`, and return the next in line after `Derived` in the MRO: `Base`.

2.6.7 How can I organize my code to make it easier to change the base class?

You could assign the base class to an alias and derive from the alias. Then all you have to change is the value assigned to the alias. Incidentally, this trick is also handy if you want to decide dynamically (e.g. depending on availability of resources) which base class to use. Example:

```
class Base:
    ...

BaseAlias = Base

class Derived(BaseAlias):
    ...
```

2.6.8 How do I create static class data and static class methods?

Both static data and static methods (in the sense of C++ or Java) are supported in Python.

For static data, simply define a class attribute. To assign a new value to the attribute, you have to explicitly use the class name in the assignment:

```
class C:
    count = 0   # C.__init__ 被呼叫的次數

    def __init__(self):
        C.count = C.count + 1

    def getcount(self):
        return C.count  # 或回傳 self.count
```

`c.count` also refers to `C.count` for any `c` such that `isinstance(c, C)` holds, unless overridden by `c` itself or by some class on the base-class search path from `c.__class__` back to `C`.

Caution: within a method of C, an assignment like `self.count = 42` creates a new and unrelated instance named ”count” in `self`’s own dict. Rebinding of a class-static data name must always specify the class whether inside a method or not:

```
C.count = 314
```

🄵態方法是可能的：

```
class C:
    @staticmethod
    def static(arg1, arg2, arg3):
        # 🄵有 'self' 參數!
        ...
```

However, a far more straightforward way to get the effect of a static method is via a simple module-level function:

```
def getcount():
    return C.count
```

If your code is structured so as to define one class (or tightly related class hierarchy) per module, this supplies the desired encapsulation.

2.6.9 How can I overload constructors (or methods) in Python?

這個答案實際上適用於所有方法，但這個問題通常會先出現在建構函式的情境中。

在 C++ 中你會寫成

```
class C {
    C() { cout << "No arguments\n"; }
    C(int i) { cout << "Argument is " << i << "\n"; }
}
```

在 Python 中，你必須編寫一個建構函式來捕獲所有使用預設引數的情🄵。例如：

```
class C:
    def __init__(self, i=None):
        if i is None:
            print("No arguments")
        else:
            print("Argument is", i)
```

這🄵不完全等價，但在實際情🄵中已🄵接近。

你也可以嘗試長度可變的引數串列，例如：

```
def __init__(self, *args):
    ...
```

相同的手段適用於所有方法的定義。

2.6.10 我嘗試使用 __spam，但收到有關 _SomeClassName__spam 的錯誤。

Variable names with double leading underscores are ”mangled” to provide a simple but effective way to define class private variables. Any identifier of the form `__spam` (at least two leading underscores, at most one trailing underscore) is textually replaced with `_classname__spam`, where `classname` is the current class name with any leading underscores stripped.

The identifier can be used unchanged within the class, but to access it outside the class, the mangled name must be used:

```
class A:
    def __one(self):
        return 1
    def two(self):
        return 2 * self.__one()

class B(A):
    def three(self):
        return 3 * self._A__one()

four = 4 * A()._A__one()
```

In particular, this does not guarantee privacy since an outside user can still deliberately access the private attribute; many Python programmers never bother to use private variable names at all.

也參考

The private name mangling specifications for details and special cases.

2.6.11 我的類F⃞定義了 __del__ 但是當我F⃞除物件時它F⃞有被呼叫。

這有幾個可能的原因。

The `del` statement does not necessarily call `__del__()` -- it simply decrements the object's reference count, and if this reaches zero `__del__()` is called.

If your data structures contain circular links (e.g. a tree where each child has a parent reference and each parent has a list of children) the reference counts will never go back to zero. Once in a while Python runs an algorithm to detect such cycles, but the garbage collector might run some time after the last reference to your data structure vanishes, so your `__del__()` method may be called at an inconvenient and random time. This is inconvenient if you're trying to reproduce a problem. Worse, the order in which object's `__del__()` methods are executed is arbitrary. You can run `gc.collect()` to force a collection, but there *are* pathological cases where objects will never be collected.

Despite the cycle collector, it's still a good idea to define an explicit `close()` method on objects to be called whenever you're done with them. The `close()` method can then remove attributes that refer to subobjects. Don't call `__del__()` directly -- `__del__()` should call `close()` and `close()` should make sure that it can be called more than once for the same object.

Another way to avoid cyclical references is to use the `weakref` module, which allows you to point to objects without incrementing their reference count. Tree data structures, for instance, should use weak references for their parent and sibling references (if they need them!).

最後，如果你的 `__del__()` 方法引發例外，則會將一條警告訊息印出到 `sys.stderr`。

2.6.12 How do I get a list of all instances of a given class?

Python does not keep track of all instances of a class (or of a built-in type). You can program the class's constructor to keep track of all instances by keeping a list of weak references to each instance.

2.6.13 🄵什🄵 id() 的結果看起來不唯一？

The `id()` builtin returns an integer that is guaranteed to be unique during the lifetime of the object. Since in CPython, this is the object's memory address, it happens frequently that after an object is deleted from memory, the next freshly created object is allocated at the same position in memory. This is illustrated by this example:

```
>>> id(1000)
13901272
>>> id(2000)
13901272
```

The two ids belong to different integer objects that are created before, and deleted immediately after execution of the `id()` call. To be sure that objects whose id you want to examine are still alive, create another reference to the object:

```
>>> a = 1000; b = 2000
>>> id(a)
13901272
>>> id(b)
13891296
```

2.6.14 我什🄵時候可以依靠 *is* 運算子進行識🄵性測試？

`is` 運算子測試物件識🄵性。測試 `a is b` 等同於 `id(a) == id(b)` 。

識🄵性測試最重要的屬性是物件始終與自身相同，`a is a` 總是回傳 `True` 。識🄵性測試通常比相等性測試更快。與相等性測試不同，識🄵性測試保證回傳布林值 `True` 或 `False` 。

然而，只有當物件識🄵性得到保證時，識🄵性測試才能代替相等性測試。一般來🄵，保證識🄵性的情🄵有以下三種：

1) Assignments create new names but do not change object identity. After the assignment `new = old`, it is guaranteed that `new is old`.

2) Putting an object in a container that stores object references does not change object identity. After the list assignment `s[0] = x`, it is guaranteed that `s[0] is x`.

3) If an object is a singleton, it means that only one instance of that object can exist. After the assignments `a = None` and `b = None`, it is guaranteed that `a is b` because `None` is a singleton.

在大多數其他情🄵下，識🄵性測試是不可取的，相等性測試是首選。特🄵是，識🄵性測試不應用於檢查常數，例如不能保證是單例的 `int` 和 `str`：

```
>>> a = 1000
>>> b = 500
>>> c = b + 500
>>> a is c
False

>>> a = 'Python'
>>> b = 'Py'
>>> c = b + 'thon'
>>> a is c
False
```

同樣地，可變容器的新實例永遠不會相同：

```
>>> a = []
>>> b = []
```

(繼續下一頁)

(繼續上一頁)

```
>>> a is b
False
```

在標准函式庫程式碼中，你將看到幾種正確使用識🄵性測試的常見模式：

1) 正如 **PEP 8** 所推薦的，識🄵性測試是檢查 `None` 的首選方法。這在程式碼中讀起來像簡單的英語，🄵避免與其他可能具有評估🄵 false 的布林值的物件混淆。

2) Detecting optional arguments can be tricky when `None` is a valid input value. In those situations, you can create a singleton sentinel object guaranteed to be distinct from other objects. For example, here is how to implement a method that behaves like `dict.pop()`:

```
_sentinel = object()

def pop(self, key, default=_sentinel):
    if key in self:
        value = self[key]
        del self[key]
        return value
    if default is _sentinel:
        raise KeyError(key)
    return default
```

3) 容器實作有時需要透過識🄵性測試來增🄵相等性測試。這可以防止程式碼被諸如 float('NaN') 之類的不等於自身的物件所混淆。

例如，以下是 `collections.abc.Sequence.__contains__()` 的實作：

```
def __contains__(self, value):
    for v in self:
        if v is value or v == value:
            return True
    return False
```

2.6.15 子類🄵如何控制不可變實例中存儲的資料？

When subclassing an immutable type, override the `__new__()` method instead of the `__init__()` method. The latter only runs *after* an instance is created, which is too late to alter data in an immutable instance.

所有這些不可變類🄵都具有與其父類🄵不同的簽名：

```
from datetime import date

class FirstOfMonthDate(date):
    "總是選擇每個月的第一天"
    def __new__(cls, year, month, day):
        return super().__new__(cls, year, month, 1)

class NamedInt(int):
    "允許一些數字的文字名稱"
    xlat = {'zero': 0, 'one': 1, 'ten': 10}
    def __new__(cls, value):
        value = cls.xlat.get(value, value)
        return super().__new__(cls, value)

class TitleStr(str):
    "將 str 轉🄵成適合作🄵 URL 路徑的名稱"
    def __new__(cls, s):
        s = s.lower().replace(' ', '-')
        s = ''.join([c for c in s if c.isalnum() or c == '-'])
        return super().__new__(cls, s)
```

這些類□可以像這樣使用：

```
>>> FirstOfMonthDate(2012, 2, 14)
FirstOfMonthDate(2012, 2, 1)
>>> NamedInt('ten')
10
>>> NamedInt(20)
20
>>> TitleStr('Blog: Why Python Rocks')
'blog-why-python-rocks'
```

2.6.16 如何快取方法呼叫？

The two principal tools for caching methods are `functools.cached_property()` and `functools.lru_cache()`. The former stores results at the instance level and the latter at the class level.

The *cached_property* approach only works with methods that do not take any arguments. It does not create a reference to the instance. The cached method result will be kept only as long as the instance is alive.

The advantage is that when an instance is no longer used, the cached method result will be released right away. The disadvantage is that if instances accumulate, so too will the accumulated method results. They can grow without bound.

lru_cache 方法適用於具有可雜□引數的方法。除非特□努力傳遞弱參照，否則它會建立對實例的參照。

The advantage of the least recently used algorithm is that the cache is bounded by the specified *maxsize*. The disadvantage is that instances are kept alive until they age out of the cache or until the cache is cleared.

這個例子展示了各種技術：

```
class Weather:
    "Lookup weather information on a government website"

    def __init__(self, station_id):
        self._station_id = station_id
        # The _station_id is private and immutable

    def current_temperature(self):
        "Latest hourly observation"
        # Do not cache this because old results
        # can be out of date.

    @cached_property
    def location(self):
        "Return the longitude/latitude coordinates of the station"
        # Result only depends on the station_id

    @lru_cache(maxsize=20)
    def historic_rainfall(self, date, units='mm'):
        "Rainfall on a given date"
        # Depends on the station_id, date, and units.
```

The above example assumes that the *station_id* never changes. If the relevant instance attributes are mutable, the *cached_property* approach can't be made to work because it cannot detect changes to the attributes.

To make the *lru_cache* approach work when the *station_id* is mutable, the class needs to define the `__eq__()` and `__hash__()` methods so that the cache can detect relevant attribute updates:

```
class Weather:
    "Example with a mutable station identifier"

    def __init__(self, station_id):
        self.station_id = station_id
```

(繼續下一頁)

(繼續上一頁)

```
    def change_station(self, station_id):
        self.station_id = station_id

    def __eq__(self, other):
        return self.station_id == other.station_id

    def __hash__(self):
        return hash(self.station_id)

    @lru_cache(maxsize=20)
    def historic_rainfall(self, date, units='cm'):
        'Rainfall on a given date'
        # Depends on the station_id, date, and units.
```

2.7 模組

2.7.1 如何建立.pyc 檔案？

When a module is imported for the first time (or when the source file has changed since the current compiled file was created) a .pyc file containing the compiled code should be created in a __pycache__ subdirectory of the directory containing the .py file. The .pyc file will have a filename that starts with the same name as the .py file, and ends with .pyc, with a middle component that depends on the particular python binary that created it. (See **PEP 3147** for details.)

One reason that a .pyc file may not be created is a permissions problem with the directory containing the source file, meaning that the __pycache__ subdirectory cannot be created. This can happen, for example, if you develop as one user but run as another, such as if you are testing with a web server.

Unless the PYTHONDONTWRITEBYTECODE environment variable is set, creation of a .pyc file is automatic if you're importing a module and Python has the ability (permissions, free space, etc...) to create a __pycache__ subdirectory and write the compiled module to that subdirectory.

Running Python on a top level script is not considered an import and no .pyc will be created. For example, if you have a top-level module foo.py that imports another module xyz.py, when you run foo (by typing python foo.py as a shell command), a .pyc will be created for xyz because xyz is imported, but no .pyc file will be created for foo since foo.py isn't being imported.

如果你需要□ foo 建立一個 .pyc 檔案——也就是□，要□一個未引入的模組建立一個 .pyc 檔案——你可以使用 py_compile 和 compileall 模組。

py_compile 模組允許手動編譯任何模組。其中一種方法是在該模組中以交互方式使用 compile() 函式：

```
>>> import py_compile
>>> py_compile.compile('foo.py')
```

這會將 .pyc 寫入與 foo.py 相同位置的 __pycache__ 子目□（或者你可以使用可選參數 cfile 覆蓋它）。

You can also automatically compile all files in a directory or directories using the compileall module. You can do it from the shell prompt by running compileall.py and providing the path of a directory containing Python files to compile:

```
python -m compileall .
```

2.7.2 如何找到當前模組名稱？

A module can find out its own module name by looking at the predefined global variable __name__. If this has the value '__main__', the program is running as a script. Many modules that are usually used by importing them also provide a command-line interface or a self-test, and only execute this code after checking __name__:

```
def main():
    print('Running test...')
    ...

if __name__ == '__main__':
    main()
```

2.7.3 要怎樣才能擁有相互引入的模組？

假設你有以下模組：

foo.py：

```
from bar import bar_var
foo_var = 1
```

bar.py：

```
from foo import foo_var
bar_var = 2
```

問題是直譯器將執行以下步驟：

- 主要引入 foo
- 建立了 foo 的空全域變數
- foo 被編譯🄵開始執行
- foo 引入 bar
- 建立了 bar 的空全域變數
- bar 已被編譯🄵開始執行
- bar 引入 foo（這是一個空操作，因🄵已經有一個名🄵 foo 的模組）
- 引入機制嘗試從 foo 全域變數中讀取 foo_var，以設定 bar.foo_var = foo.foo_var

最後一步失敗了，因🄵 Python 還🄵有完成對 foo 的直譯，而 foo 的全域符號字典仍然是空的。

The same thing happens when you use import foo, and then try to access foo.foo_var in global code.

此問題有（至少）三種可能的解🄵方法。

Guido van Rossum recommends avoiding all uses of from <module> import ..., and placing all code inside functions. Initializations of global variables and class variables should use constants or built-in functions only. This means everything from an imported module is referenced as <module>.<name>.

Jim Roskind 建議在每個模組中按以下順序執行各個步驟：

- exports (globals, functions, and classes that don't need imported base classes)
- import 陳述式
- 活躍程式碼（包括從引入值初始化的全域變數）。

Van Rossum 不太喜歡這種方法，因🄵引入出現在一個奇怪的地方，但它確實有效。

Matthias Urlichs 建議重組 (restructuring) 你的程式碼，以便打從一開始就不需要遞🄵引入。

這些方案🄵不衝突。

2.7.4 __import__('x.y.z') 回傳 <module 'x'>，那我怎🄵得到 z？

Consider using the convenience function import_module() from importlib instead:

```
z = importlib.import_module('x.y.z')
```

2.7.5 當我編輯需要引入的模組🄵重新引入它時，更動🄵有反應出來。🄵什🄵會這樣？

For reasons of efficiency as well as consistency, Python only reads the module file on the first time a module is imported. If it didn't, in a program consisting of many modules where each one imports the same basic module, the basic module would be parsed and re-parsed many times. To force re-reading of a changed module, do this:

```
import importlib
import modname
importlib.reload(modname)
```

警告：此技術🄵非 100% 萬無一失。尤其是包含像這樣陳述式的模組：

```
from modname import some_objects
```

will continue to work with the old version of the imported objects. If the module contains class definitions, existing class instances will *not* be updated to use the new class definition. This can result in the following paradoxical behaviour:

```
>>> import importlib
>>> import cls
>>> c = cls.C()                # 建立一個 C 的實例
>>> importlib.reload(cls)
<module 'cls' from 'cls.py'>
>>> isinstance(c, cls.C)       # isinstance 🄵 false？！ ？
False
```

如果印出類🄵物件的「識🄵性」，問題的本質就很清楚了：

```
>>> hex(id(c.__class__))
'0x7352a0'
>>> hex(id(cls.C))
'0x4198d0'
```

CHAPTER 3

設計和歷史常見問答集

3.1 🄵什🄵 Python 使用縮排將陳述式進行分組？

Guido van Rossum 相信使用縮排來分組超級優雅，🄵且對提高一般 Python 程式的清晰度有許多貢獻。許多人在學習一段時間之後就愛上了這個功能。

因🄵🄵有開始/結束括號，因此剖析器和人類讀者感知到的分組就不存在分歧。偶爾 C 語言的程式設計師會遇到這樣的程式碼片段：

```
if (x <= y)
        x++;
        y--;
z++;
```

如果條件🄵真，只有 x++ 陳述式會被執行，但縮排會讓很多人對他有不同的理解。即使是資深的 C 語言開發者有時也會盯著他許久，思考🄵何即便 x > y，但 y 還是🄵少了。

因🄵🄵有開頭與結尾的括號，Python 比起其他語言會更不容易遇到程式碼風格的衝突。在 C 語言中，有多種不同的方法來放置花括號。在習慣讀寫特定風格後，去讀（或是必須去寫）另一種風格會覺得不太舒服。

很多程式碼風格會把 begin/end 獨立放在一行。這會讓程式碼很長且浪費珍貴的螢幕空間，要概覽程式時也變得較🄵困難。理想上來🄵，一個函式應該要🄵一個螢幕（大概 20 至 30 行）。20 行的 Python 程式碼比起 20 行的 C 程式碼可以做更多事。雖然🄵有開頭與結尾的括號🄵非單一原因（🄵有變數宣告及高階的資料型🄵同樣有關），但縮排式的語法確實給了幫助。

3.2 🄵什🄵我會從簡單的數學運算得到奇怪的結果？

請見下一個問題。

3.3 🄵何浮點數運算如此不精確？

使用者時常對這樣的結果感到驚訝：

```
>>> 1.2 - 1.0
0.19999999999999996
```

然後認🄵這是 Python 的 bug，但這🄵不是。這跟 Python 幾乎🄵有關🄵，而是和底層如何處理浮點數有關🄵。

CPython 的 `float` 型🄵使用了 C 的 `double` 型🄵來儲存。一個 `float` 物件的值會以固定的精度（通常🄵 53 位元）存🄵二進制浮點數，Python 使用 C 來運算浮點數，而他的結果會依處理器中的硬體實作方式來🄵定。這表示就浮點數運算來🄵，Python 和 C、Java 等很多受歡迎的語言有一樣的行🄵。

很多數字可以簡單地寫成十進位表示，但🄵無法簡單地以二進制浮點數表示。比方🄵，在以下程式碼執行後：

```
>>> x = 1.2
```

`x` 🄵的值是一個（很接近）1.2 的估計值，但🄵非精確地等於 1.2。以一般的電腦來🄵，他實際儲存的值是：

```
1.0011001100110011001100110011001100110011001100110011 (binary)
```

而這個值正是：

```
1.1999999999999999555910790149937383830547332763671875 (decimal)
```

53 位元的精度讓 Python 可以有 15 至 16 小數位的准確度。

要更完全的解釋可以查🄵在 Python 教學的浮點運算一章。

3.4 🄵什🄵 Python 字串不可變動？

有許多優點。

其一是效能：知道字串不可變動後，我們就可以在創造他的時候就分配好空間，而後他的儲存空間需求就是固定不變的。這也是元組 (tuple) 和串列 (list) 相🄵的其中一個原因。

另一個優點是在 Python 中，字串和數字一樣「基本」。🄵有任何行🄵會把 8 這個數值改成其他數值；同理，在 Python 中也🄵有任何行🄵會修改字串「eight」。

3.5 🄵何「self」在方法 (method) 定義和呼叫時一定要明確使用？

此構想從 Modula-3 而來。因🄵許多原因，他可以🄵是非常實用。

第一，這樣可以更明顯表現出你在用方法 (method) 或是實例 (instance) 的屬性，而非一個區域變數。即使不知道類🄵 (class) 的定義，當看到 `self.x` 或 `self.meth()`，就會很清楚地知道是正在使用實例的變數或是方法。在 C++ 🄵，你可以藉由🄵有區域變數宣告來判斷這件事 —— 但在 Python 🄵🄵有區域變數宣告，所以你必須去看類🄵的定義來確定。有些 C++ 和 Java 的程式碼規格要求要在實例屬性的名稱加上前綴 `m_`，所以這種明確性在那些語言也是很好用的。

第二，當你想明確地使用或呼叫在某個類🄵🄵的方法的時候，你不需要特殊的語法。在 C++ 🄵，如果你想用一個在繼承類🄵時被覆寫的基底類🄵方法，必須要用 `::` 運算子 -- 但在 Python 🄵，你可以直接寫成 `baseclass.methodname(self, <argument list>)`。這在 `__init__()` 方法很好用，特🄵是在一個繼承的類🄵要擴充基底類🄵的方法而要呼叫他時。

最後，他解🄵了關於實例變數指派的語法問題：因🄵區域變數在 Python 是（定義🄵）在函式🄵被指派值的變數（且🄵有被明確宣告成全域），所以會需要一個方法來告訴直譯器這個指派運算是針對實例變數，而非針對區域變數，這在語法層面處理較好（🄵了效率）。C++ 用宣告解🄵了這件事，但 Python 🄵有，而🄵了這個原因而引入變數宣告機制又略嫌浪費。但使用明確的 `self.var` 就可以把這個問題圓滿解🄵。同理，在用實例變數的時候必須寫成 `self.var` 即代表對於在方法中不特定的名稱不需要去看實例的🄵容。🄵句話🄵，區域變數和實例變數存在於兩個不同的命名空間 (namespace)，而你需要告訴 Python 要使用哪一個。

3.6 🄵何我不能在運算式 (expression) 中使用指派運算？

從 Python 3.8 開始，你可以這🄵做了！

指派運算式使用海象運算子 := 來在運算式中指派變數值：

```
while chunk := fp.read(200):
   print(chunk)
```

更多資訊請見 **PEP 572**。

3.7 🄵何 Python 對於一些功能實作使用方法（像是 list.index()），另一些使用函式（像是 len(list)）？

如 Guido 所🄵：

> （一）對一些運算來🄵，前綴寫法看起來會比後綴寫法好 —— 前綴（和中綴！）運算在數學上有更久遠的傳統，這些符號在視覺上幫助數學家們更容易思考問題。想想把 x*(a+b) 這種式子展開成 x*a + x*b 的簡單，再比較一下古老的圈圈符號記法的笨拙就知道了。
>
> （二）當我看到一段程式碼寫著 len(x)，我知道他要找某個東西的長度。這告訴了我兩件事：結果是一個整數、參數是某種容器。相對地，當我看到 x.len()，我必須先知道 x 是某種容器，🄵實作了一個介面或是繼承了一個有標准 len() 的類🄵。遇到一個🄵有實作對映 (mapping) 的類🄵🄵有 get() 或 keys() 方法，或是不是檔案但🄵有 write() 方法時，我們偶爾會覺得困惑。
>
> —https://mail.python.org/pipermail/python-3000/2006-November/004643.html

3.8 🄵何 join() 是字串方法而非串列 (list) 或元組 (tuple) 方法？

自 Python 1.6 之後，字串變得很像其他標准的型🄵，也在此時，一些可以和字串模組的函式有相同功能的方法也被加入。大多數的新方法都被廣泛接受，但有一個方法似乎讓一些程式人員不舒服：

```
", ".join(['1', '2', '4', '8', '16'])
```

結果是：

```
"1, 2, 4, 8, 16"
```

通常有兩個反對這個用法的論點。

第一項這🄵🄵：「用字串文本 (string literal)（字串常數）看起來真的很醜」，也許真的如此，但字串文本就只是一個固定值。如果方法可以用在值🄵字串的變數上，那🄵道理字串文本不能被使用。

第二個反對意見通常是：「我是在叫一個序列把它的成員用一個字串常數連接起來」。但很遺憾地，你🄵不是在這樣做。因🄵某種原因，把 `split()` 當成字串方法比較簡單，因🄵這樣我們可以輕易地看到：

```
"1, 2, 4, 8, 16".split(", ")
```

這是在叫一個字串文本回傳由指定的分隔符號（或是預設🄵空白）分出的子字串的指令。

`join()` 是一個字串方法，因🄵在用他的時候，你是告訴分隔字串去走遍整個字串序列，🄵將自己插入到相鄰的兩項之間。這個方法的參數可以是任何符合序列規則的物件，包括自定義的新類🄵。在 bytes 和 bytearray 物件也有類似的方法可用。

3.9 例外處理有多快？

如果🄵有例外被🄵出，一個 `try/except` 區塊是非常有效率的。事實上，抓捕例外要付出昂貴的代價。在 Python 2.0 以前，這樣使用是相當常見的：

```
try:
    value = mydict[key]
except KeyError:
    mydict[key] = getvalue(key)
    value = mydict[key]
```

這只有在你預料這個字典大多數時候都有鍵的時候才合理。如果□非如此，你應該寫成：

```
if key in mydict:
    value = mydict[key]
else:
    value = mydict[key] = getvalue(key)
```

單就這個情□來□，你也可以用 `value = dict.setdefault(key, getvalue(key))`，不過只有在 `getvalue()` 代價不大的時候才能用，畢竟他每次都會被執行。

3.10 □什□ Python □□有 switch 或 case 陳述式？

In general, structured switch statements execute one block of code when an expression has a particular value or set of values. Since Python 3.10 one can easily match literal values, or constants within a namespace, with a `match ... case` statement. An older alternative is a sequence of `if... elif... elif... else`.

如果可能性很多，你可以用字典去對映要呼叫的函式。舉例來□：

```
if key in mydict:
    value = mydict[key]
else:
    value = mydict[key] = getvalue(key)
```

對於呼叫物件□的方法，你可以利用□建用來找尋特定方法的函式 `getattr()` 來做進一步的簡化：

```
class MyVisitor:
    def visit_a(self):
        ...

    def dispatch(self, value):
        method_name = 'visit_' + str(value)
        method = getattr(self, method_name)
        method()
```

我們建議在方法名稱加上前綴，以這個例子來□是像是 `visit_`。□有前綴的話，一旦收到從不信任來源的值，攻擊者便可以隨意呼叫在你的專案□的方法。

Imitating switch with fallthrough, as with C's switch-case-default, is possible, much harder, and less needed.

3.11 □何不能在直譯器上模擬執行緒，而要使用作業系統的特定實作方式？

答案一：很不幸地，直譯器對每個 Python 的堆□框 (stack frame) 會推至少一個 C 的堆□框。同時，擴充套件可以隨時呼叫 Python，因此完整的實作必須要支援 C 的執行緒。

答案二：幸運地，無堆□ (Stackless) Python 完全重新設計了直譯器□圈，□避免了 C 堆□。

3.12 □何 lambda 運算式不能包含陳述式？

Python 的 lambda 運算式不能包含陳述式是因□ Python 的語法框架無法處理包在運算式中的陳述式。然而，在 Python □這□不是一個嚴重的問題。不像在其他語言中有獨立功能的 lambda，Python 的 lambda 只是一個在你懶得定義函式時可用的一個簡寫表達法。

函式已經是 Python 🄵的一級物件 (first class objects)，而且可以在區域範圍🄵被宣告。因此唯一用 lambda 而非區域性的函式的優點就是你不需要多想一個函式名稱—但這樣就會是一個區域變數被指定成函式物件（和 lambda 運算式的結果同類）！

3.13 Python 可以被編譯成機器語言、C 語言或其他種語言嗎？

Cython 可以編譯一個調整過有選擇性🄵解的 Python 版本。Nuitka 是一個有🄵力編譯器，可以把 Python 編譯成 C++，他的目標是支援完整的 Python 語言。

3.14 Python 如何管理記憶體？

Python 記憶體管理的細節取🄵於實作。Python 的標准實作*CPython* 使用參照計次 (reference counting) 來偵測不再被存取的物件，🄵用另一個機制來收集參照循環 (reference cycle)、定期執行循環偵測演算法來找不再使用的循環🄵🄵除相關物件。gc 模組提供了可以執行垃圾收集、抓取除錯統計數據和調整收集器參數的函式。

然而，在其他實作（像是 Jython 或 PyPy）中，會使用像是成熟的垃圾收集器等不同機制。如果你的 Python 程式碼的表現取🄵於參照計次的實作，這個相🄵處會導致一些微小的移植問題。

在一些 Python 實作中，下面這段程式碼（在 CPython 可以正常運作）可能會把檔案描述子 (file descriptor) 用盡：

```
for file in very_long_list_of_files:
    f = open(file)
    c = f.read(1)
```

實際上，使用 CPython 的參照計次和解構方案 (destructor scheme)，每個對 f 的新指派都會關閉前面打開的檔案。然而用傳統的垃圾回收 (GC) 的話，這些檔案物件只會在不固定且有可能很長的時間後被收集（🄵關閉）。

如果你希望你的程式碼在任何 Python 實作版本中都可以運作，那你應該清楚地關閉檔案或是使用 with 陳述式，如此一來，不用管記憶體管理的方法，他也會正常運作：

```
for file in very_long_list_of_files:
    with open(file) as f:
        c = f.read(1)
```

3.15 🄵何 CPython 不使用更多傳統的垃圾回收機制？

第一，這🄵不是 C 的標准功能，因此他的可🄵性低。（對，我們知道 Boehm GC 函式庫。他有可相容於大多數平台的組合語言程式碼，但依然不是全部，而即便它大多數是通透的，也🄵不完全，要讓它跟 Python 相容還是需要做一些修補。）

傳統的垃圾收集 (GC) 在 Python 被嵌入其他應用程式時也成了一個問題。在獨立的 Python 程式🄵當然可以把標准的 malloc() 和 free() 🄵成 GC 函式庫提供的其他版本；但一個嵌著 Python 的應用程式可能想用自己的 malloc() 和 free() 替代品，而不是用 Python 的。以現在來🄵，CPython 和實作 malloc() 和 free() 的程式相處融洽。

3.16 當 CPython 結束時，🄵何所有的記憶體不會被釋放？

當離開 Python 時，從 Python 模組的全域命名空間來的物件🄵非總是會被釋放。在有循環引用的時候，這可能會發生。有些記憶體是被 C 函式庫取用的，他們不可能被釋放（例如：像是 Purify 之類的工具會抱怨）。然而，Python 在關閉的時候會積極清理記憶體🄵嘗試🄵除每個物件。

如果你想要🄵迫 Python 在釋放記憶體時🄵除特定的東西，你可以用 atexit 模組來執行會🄵制🄵除的函式。

3.17 🄵何要把元組 (tuple) 和串列 (list) 分成兩個資料型態？

串列和元組在很多方面相當相似，但通常用在完全不同的地方。元組可以想成 Pascal 的 `record` 或是 C 的 `struct`，是一小群相關聯但可能是不同型🄵的資料集合，以一組🄵單位進行操作。舉例來🄵，一個笛卡兒坐標系可以適當地表示成一個有二或三個值的元組。

另一方面，串列更像是其他語言的陣列 (array)。他可以有不固定個同類🄵物件，且🄵逐項操作。舉例來🄵，`os.listdir('.')` 回傳當下目🄵🄵的檔案，以包含字串的串列表示。如果你新增了幾個檔案到這個目🄵，一般來🄵操作結果的函式也會正常運作。

元組則是不可變的，代表一旦元組被建立，你就不能🄵改變🄵面的任何一個值。而串列可變，所以你可以改變🄵面的元素。只有不可變的元素可以成🄵字典的鍵，所以只能把元組當成鍵，而串列則不行。

3.18 串列 (list) 在 CPython 中是怎🄵實作的？

CPython 的串列 (list) 事實上是可變長度的陣列 (array)，而不是像 Lisp 語言的鏈接串列 (linked list)。實作上，他是一個連續的物件參照 (reference) 陣列，🄵把指向此陣列的指標 (pointer) 和陣列長度存在串列的標頭結構🄵。

因此，用索引來找串列特定項 `a[i]` 的代價和串列大小或是索引值無關。

當新物件被新增或插入時，陣列會被調整大小。🄵了改善多次加入物件的效率，我們有用一些巧妙的方法，當陣列必須變大時，會多收集一些額外的空間，接下來幾次新增時就不需要再調整大小了。

3.19 字典 (dictionaries) 在 CPython 中是怎🄵實作的？

CPython 的字典是用可調整大小的雜🄵表 (hash table) 實作的。比起 B 樹 (B-tree)，在搜尋（目前🄵止最常見的操作）方面有更好的表現，實作上也較🄵簡單。

字典利用🄵建 `hash()` 函式，對每個鍵做雜🄵計算。雜🄵結果依據鍵的值和個🄵執行緒 (processes) 的種子而有相當大的差距。例如，`'Python'` 的雜🄵是 `-539294296`，而只差一個字的 `'python'` 則是 `1142331976`。雜🄵結果接著被用來計算值在🄵部陣列儲存的位置。假設你存的鍵都有不同的雜🄵值，那字典只需要常數時間—用大 O 表示法 (Big-O notation) 就是 $O(1)$ —來找任意一個鍵。

3.20 🄵何字典的鍵一定是不可變的？

實作字典用的雜🄵表是根據鍵的值做計算從而找到鍵的。如果鍵可變的話，他的值就可以改變，則雜🄵的結果也會一起變動。但改變鍵的物件的人無從得知他被用來當成字典的鍵，所以無法修改字典的🄵容。然後，當你嘗試在字典中尋找這個物件時，因🄵雜🄵值不同的緣故，你找不到他。而如果你嘗試用舊的值去尋找，也一樣找不到，因🄵他的雜🄵結果和原先物件不同。

如果你想要用串列作🄵字典的索引，把他轉🄵成元組即可。`tuple(L)` 函式會建立一個和串列 `L` 一樣🄵容的元組。而元組是不可變的，所以可以用來當成字典的鍵。

也有人提出一些不能接受的方法：

- 用串列的記憶體位址（物件 id）來雜🄵。這不會成功，因🄵你如果用同樣的值建立一個新的串列，是找不到的。舉例來🄵：

  ```
  mydict = {[1, 2]: '12'}
  print(mydict[[1, 2]])
  ```

  這將會導致 `KeyError` 例外，因🄵 `[1, 2]` 的 id 在第一行和第二行是不同的。🄵句話🄵，字典的鍵應該要用 `==` 來做比較，而不是用 `is`。

- 🄵🄵一個串列作🄵鍵。這一樣不會成功，因🄵串列是可變的，他可以包含自己的參照，所以🄵🄵會形成一個無窮🄵圈。

- 允許串列作🄵鍵，但告訴使用者不要更動他。當你不小心忘記或是更動了這個串列，會🄵生一種難以追🄵的 bug。他同時也違背了一項字典的重要定則：在 `d.keys()` 的每個值都可以當成字典的鍵。

- 一旦串列被當成鍵，把他標記成只能讀取。問題是，這不只要避免最上層的物件改變值，就像用元組包含串列來做□鍵。把一個物件當成鍵，需要將從他開始可以接觸到的所有物件都標記成只能讀取—所以再一次，自己參照自己的物件會導致無窮□圈。

如果你需要的話，這□有個小技巧可以幫你，但請自己承擔風險：你可以把一個可變物件包裝進一個有 `__eq__()` 和 `__hash__()` 方法的類□實例。只要這種包裝物件還存在於字典（或其他類似結構）中，你就必須確定在字典（或其他用雜□□基底的結構）中他們的雜□值會保持□定。

```python
class ListWrapper:
    def __init__(self, the_list):
        self.the_list = the_list

    def __eq__(self, other):
        return self.the_list == other.the_list

    def __hash__(self):
        l = self.the_list
        result = 98767 - len(l)*555
        for i, el in enumerate(l):
            try:
                result = result + (hash(el) % 9999999) * 1001 + i
            except Exception:
                result = (result % 7777777) + i * 333
        return result
```

請注意，雜□的計算可能變得□雜，因□有串列成員不可雜□ (unhashable) 和算術溢位的可能性。

此外，不管物件是否在字典中，如果 `o1 == o2`（即 `o1.__eq__(o2) is True`），則 `hash(o1) == hash(o2)`（即 `o1.__hash__() == o2.__hash__()`），這個事實必須要成立。如果無法滿足這項限制，那字典和其他用雜□□基底的結構會出現不正常的行□。

至於 `ListWrapper`，只要這個包裝過的物件在字典中，□面的串列就不能改變以避免不正常的事情發生。除非你已經謹慎思考過你的需求和無法滿足條件的後果，不然請不要這□做。請自行注意。

3.21 □何 list.sort() 不是回傳排序過的串列？

在重視效能的情□下，把串列□□一份有些浪費。因此，`list.sort()` 直接在串列□做排序。□了提醒你這件事，他不會回傳排序過的串列。這樣一來，當你需要排序過和未排序過的串列時，你就不會被誤導而不小心覆蓋掉串列。

如果你想要他回傳新的串列，那可以改用□建的 `sorted()`。他會用提供的可□代物件 (iterable) 來排序建立新串列，□回傳之。例如，以下這個範例會□明如何有序地□代字典的鍵：

```python
for key in sorted(mydict):
    ...  # do whatever with mydict[key]...
```

3.22 如何在 Python 中指定和□制使用一個介面規范 (interface spec)？

像是 C++ 和 Java 等語言提供了模組的介面規範，他描述了該模組的方法和函式的原型。很多人認□這種在編譯時□制執行的介面規範在建構大型程式時十分有幫助。

Python 2.6 加入了 `abc` 模組，讓你可以定義抽象基底類□ (Abstract Base Class, ABC)。你可以使用 `isinstance()` 和 `issubclass()` 來確認一個實例或是類□是否實作了某個抽象基底類□。而 `collections.abc` 模組定義了一系列好用的抽象基底類□，像是 `Iterable`、`Container` 和 `MutableMapping`。

對 Python 來□，很多介面規範的優點可以用對元件適當的測試規則來達到。

一個針對模組的好測試套件提供了回歸測試 (regression testing)，□作□模組介面規範和一組範例。許多 Python 模組可以直接當成□本執行，□提供簡單的「自我測試」。即便模組使用了□雜的外部介面，他依

然可以用外部介面的簡單的「樁」(stub) 模擬來獨立測試。doctest 和 unittest 模組或第三方的測試框架可以用來建構詳盡徹底的測試套件來測試模組□的每一行程式碼。

就像介面規範一樣，一個適當的測試規則在建造大型又□雜的 Python 應用程式時可以幫上忙。事實上，他可能可以有更好的表現，因□介面規範無法測試程式的特定屬性。舉例來□，list.append() 方法應該要在某個□部的串列最後面加上新的元素，而介面規範□辦法測試你的 list.append() 是不是真的有正確的實作，但這在測試套件□是件很簡單的事。

撰寫測試套件相當有幫助，而你會像要把程式碼設計成好測試的樣子。測試驅動開發 (test-driven development) 是一個越來越受歡迎的設計方法，他要求先完成部分的測試套件，再去撰寫真的要用的程式碼。當然 Python 也允許你草率地不寫任何測試。

3.23 □何□有 goto 語法？

在 1970 年代，人們了解到□有限制的 goto 會導致混亂、難以理解和修改的「義大利□」程式碼 ("spaghetti" code)。在高階語言□，這也是不需要的，因□有方法可以做邏輯分支（以 Python 來□，用 if 陳述式和 or、and 及 if/else 運算式）和□圈（用 while 和 for 陳述式，可能會有 continue 和 break）。

我們也可以用例外來做「結構化的 goto」，這甚至可以跨函式呼叫。很多人覺得例外可以方便地模擬在 C、Fortran 和其他語言□各種合理使用的 go 和 goto。例如：

```
class label(Exception): pass  # declare a label

try:
    ...
    if condition: raise label()  # goto label
    ...
except label:  # where to goto
    pass
...
```

這依然不能讓你跳進□圈□，這通常被認□是對 goto 的濫用。請小心使用。

3.24 □何純字串 (r-string) 不能以反斜□結尾？

更精確地來□，他不能以奇數個反斜□結尾：尾端未配對的反斜□會使結尾的引號被轉義 (escapes)，變成一個未結束的字串。

設計出純字串是□了提供有自己反斜□轉義處理的處理器（主要是正規表示式）一個方便的輸入方式。這種處理器會把未配對的結尾反斜□當成錯誤，所以純字串不允許如此。相對地，他讓你用一個反斜□轉義引號。這些規則在他們預想的目的上正常地運作。

如果你嘗試建立 Windows 的路徑名稱，請注意 Windows 系統指令也接受一般斜□：

```
f = open("/mydir/file.txt")  # works fine!
```

如果你嘗試建立 DOS 指令的路徑名稱，試試看使用以下的範例：

```
dir = r"\this\is\my\dos\dir" "\\"
dir = r"\this\is\my\dos\dir\ "[:-1]
dir = "\\this\\is\\my\\dos\\dir\\"
```

3.25 □何 Python □有屬性賦值的 with 陳述式？

Python 的 with 陳述式包裝了一區塊程式的執行，在進入和離開該區塊時執行程式碼。一些語言會有像如下的結構：

```
with obj:
    a = 1               # equivalent to obj.a = 1
    total = total + 1   # obj.total = obj.total + 1
```

但在 Python，這種結構是模糊的。

在其他語言□，像是 Object Pascal、Delphi 和 C++，使用的是□態型□，所以我們可以清楚地知道是哪一個成員被指派值。這是□態型□的重點—在編譯的時候，編譯器永遠都知道每個變數的作用域 (scope)。

Python 使用的是動態型□。所以我們不可能提前知道在執行時哪個屬性會被使用到。成員屬性可能在執行時從物件中被新增或移除。這使得如果簡單來看的話，我們無法得知以下哪個屬性會被使用：區域的、全域的、或是成員屬性？

以下列不完整的程式碼□例：

```
def foo(a):
    with a:
        print(x)
```

這段程式碼假設 a 有一個叫做 x 的成員屬性。然後，Python □□□有任何□象告訴直譯器這件事。在假設「a」是一個整數的話，那會發生什□事？如果有一個全域變數稱□ x，那在這個 with 區塊會被使用嗎？如你所見，Python 動態的天性使得這種選擇更加困難。

然而 with 陳述式或類似的語言特性（□少程式碼量）的主要好處可以透過賦值來達成。相較於這樣寫：

```
function(args).mydict[index][index].a = 21
function(args).mydict[index][index].b = 42
function(args).mydict[index][index].c = 63
```

應該寫成這樣：

```
ref = function(args).mydict[index][index]
ref.a = 21
ref.b = 42
ref.c = 63
```

這也有提升執行速度的副作用，因□ Python 的名稱綁定解析會在執行的時候發生，而第二版只需要執行解析一次即可。

類似的提案包括引入語法以進一步□少程式碼量，例如使用「前導點 (leading dot)」，但這些提案已被舍□，以維持程式的明確性（參□ https://mail.python.org/pipermail/python-ideas/2016-May/040070.html）。

3.26 □何□生器 (generator) 不支援 with 陳述式？

出於技術原因，把□生器直接用作情境 (context) 管理器會無法正常運作。因□通常來□，□生器是被當成□代器 (iterator)，到最後完成時不需要被手動關閉。但如果你需要的話，你可以在 with 陳述式□用 contextlib.closing(generator) 來包裝他。

3.27 □何 if、while、def、class 陳述式□需要冒號？

需要冒號主要是□了增加可讀性（由 ABC 語言的實驗得知）。試想如下範例：

```
if a == b
    print(a)
```

以及：

```
if a == b:
    print(a)
```

注意第二個例子稍微易讀一些的原因。可以更進一步觀察，一個冒號是如何放在這個 FAQ 答案的例子□的，這是標准的英文用法。

另一個小原因是冒號會使編輯器更容易做語法突顯，他們只需要看冒號的位置就可以□定是否需要更多縮排，而不用做更多繁□精密的程式碼剖析。

3.28 □何 Python 允許在串列和元組末端加上逗號？

Python 允許你在串列、元組和字典的結尾加上逗號：

```
[1, 2, 3,]
('a', 'b', 'c',)
d = {
    "A": [1, 5],
    "B": [6, 7],  # last trailing comma is optional but good style
}
```

這有許多原因可被允許。

當你要把串列、元組或字典的值寫成多行時，這樣做會讓你新增元素時較□方便，因□你不需要在前一行加上逗號。這幾行的值也可以被重新排序，而不會導致語法錯誤。

不小心遺漏了逗號會導致難以發現的錯誤，例如：

```
x = [
  "fee",
  "fie"
  "foo",
  "fum"
]
```

這個串列看起來有四個元素，但他其實只有三個：「fee」、「fiefoo」、「fum」。永遠記得加上逗號以避免這種錯誤。

允許結尾逗號也讓生成的程式碼更容易□生。

CHAPTER 4

函式庫和擴充功能的常見問題

4.1 常見函式問題

4.1.1 我如何找到執行任務 X 的模組或應用程式？

Check the Library Reference to see if there's a relevant standard library module. (Eventually you'll learn what's in the standard library and will be able to skip this step.)

For third-party packages, search the Python Package Index or try Google or another web search engine. Searching for "Python" plus a keyword or two for your topic of interest will usually find something helpful.

4.1.2 哪🄵可以找到 math.py (socket.py, regex.py, 等...) 來源檔案？

If you can't find a source file for a module it may be a built-in or dynamically loaded module implemented in C, C++ or other compiled language. In this case you may not have the source file or it may be something like `mathmodule.c`, somewhere in a C source directory (not on the Python Path).

有（至少）三種 Python 模組：

1) 以 Python 編寫的模組 (.py)；

2) 用 C 編寫🄵動態載入的模組（.dll、.pyd、.so、.sl 等）；

3) 用 C 編寫🄵與直譯器鏈接的模組；要獲得這些 list，請輸入：

```
import sys
print(sys.builtin_module_names)
```

4.1.3 我如何使 Python script 執行在 Unix？

You need to do two things: the script file's mode must be executable and the first line must begin with `#!` followed by the path of the Python interpreter.

The first is done by executing `chmod +x scriptfile` or perhaps `chmod 755 scriptfile`.

The second can be done in a number of ways. The most straightforward way is to write

```
#!/usr/local/bin/python
```

as the very first line of your file, using the pathname for where the Python interpreter is installed on your platform.

If you would like the script to be independent of where the Python interpreter lives, you can use the **env** program. Almost all Unix variants support the following, assuming the Python interpreter is in a directory on the user's PATH:

```
#!/usr/bin/env python
```

Don't do this for CGI scripts. The PATH variable for CGI scripts is often very minimal, so you need to use the actual absolute pathname of the interpreter.

Occasionally, a user's environment is so full that the **/usr/bin/env** program fails; or there's no env program at all. In that case, you can try the following hack (due to Alex Rezinsky):

```
#! /bin/sh
""":"
exec python $0 ${1+"$@"}
"""
```

The minor disadvantage is that this defines the script's __doc__ string. However, you can fix that by adding

```
__doc__ = """...Whatever..."""
```

4.1.4 是否有適用於 Python 的 curses/termcap 套件？

For Unix variants: The standard Python source distribution comes with a curses module in the Modules subdirectory, though it's not compiled by default. (Note that this is not available in the Windows distribution -- there is no curses module for Windows.)

The curses module supports basic curses features as well as many additional functions from ncurses and SYSV curses such as colour, alternative character set support, pads, and mouse support. This means the module isn't compatible with operating systems that only have BSD curses, but there don't seem to be any currently maintained OSes that fall into this category.

4.1.5 Python 中是否有等同於 C 的 onexit() 的函式？

The atexit module provides a register function that is similar to C's onexit().

4.1.6 🄵什🄵我的信號處理程式不起作用？

The most common problem is that the signal handler is declared with the wrong argument list. It is called as

```
handler(signum, frame)
```

所以它應該用兩個參數聲明：

```
def handler(signum, frame):
    ...
```

4.2 常見課題

4.2.1 如何測試 Python 程式或元件？

Python comes with two testing frameworks. The doctest module finds examples in the docstrings for a module and runs them, comparing the output with the expected output given in the docstring.

The unittest module is a fancier testing framework modelled on Java and Smalltalk testing frameworks.

To make testing easier, you should use good modular design in your program. Your program should have almost all functionality encapsulated in either functions or class methods -- and this sometimes has the surprising and delightful effect of making the program run faster (because local variable accesses are faster than global accesses). Furthermore the program should avoid depending on mutating global variables, since this makes testing much more difficult to do.

The ”global main logic” of your program may be as simple as

```
if __name__ == "__main__":
    main_logic()
```

at the bottom of the main module of your program.

Once your program is organized as a tractable collection of function and class behaviours, you should write test functions that exercise the behaviours. A test suite that automates a sequence of tests can be associated with each module. This sounds like a lot of work, but since Python is so terse and flexible it's surprisingly easy. You can make coding much more pleasant and fun by writing your test functions in parallel with the ”production code”, since this makes it easy to find bugs and even design flaws earlier.

”Support modules” that are not intended to be the main module of a program may include a self-test of the module.

```
if __name__ == "__main__":
    self_test()
```

Even programs that interact with complex external interfaces may be tested when the external interfaces are unavailable by using ”fake” interfaces implemented in Python.

4.2.2 How do I create documentation from doc strings?

The pydoc module can create HTML from the doc strings in your Python source code. An alternative for creating API documentation purely from docstrings is epydoc. Sphinx can also include docstring content.

4.2.3 How do I get a single keypress at a time?

For Unix variants there are several solutions. It's straightforward to do this using curses, but curses is a fairly large module to learn.

4.3 執行緒

4.3.1 如何使用執行緒編寫程式？

Be sure to use the threading module and not the _thread module. The threading module builds convenient abstractions on top of the low-level primitives provided by the _thread module.

4.3.2 我的執行緒似乎都□有運行：□什□？

As soon as the main thread exits, all threads are killed. Your main thread is running too quickly, giving the threads no time to do any work.

A simple fix is to add a sleep to the end of the program that's long enough for all the threads to finish:

```
import threading, time

def thread_task(name, n):
    for i in range(n):
        print(name, i)

for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(i), i))
    T.start()

time.sleep(10)  # <---------------------------!
```

But now (on many platforms) the threads don't run in parallel, but appear to run sequentially, one at a time! The reason is that the OS thread scheduler doesn't start a new thread until the previous thread is blocked.

A simple fix is to add a tiny sleep to the start of the run function:

```
def thread_task(name, n):
    time.sleep(0.001)  # <--------------------!
    for i in range(n):
        print(name, i)

for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(i), i))
    T.start()

time.sleep(10)
```

Instead of trying to guess a good delay value for `time.sleep()`, it's better to use some kind of semaphore mechanism. One idea is to use the `queue` module to create a queue object, let each thread append a token to the queue when it finishes, and let the main thread read as many tokens from the queue as there are threads.

4.3.3 How do I parcel out work among a bunch of worker threads?

The easiest way is to use the `concurrent.futures` module, especially the `ThreadPoolExecutor` class.

Or, if you want fine control over the dispatching algorithm, you can write your own logic manually. Use the `queue` module to create a queue containing a list of jobs. The `Queue` class maintains a list of objects and has a `.put(obj)` method that adds items to the queue and a `.get()` method to return them. The class will take care of the locking necessary to ensure that each job is handed out exactly once.

Here's a trivial example:

```
import threading, queue, time

# The worker thread gets jobs off the queue.  When the queue is empty, it
# assumes there will be no more work and exits.
# (Realistically workers will run until terminated.)
def worker():
    print('Running worker')
    time.sleep(0.1)
    while True:
        try:
            arg = q.get(block=False)
        except queue.Empty:
            print('Worker', threading.current_thread(), end=' ')
            print('queue empty')
            break
        else:
            print('Worker', threading.current_thread(), end=' ')
            print('running with argument', arg)
            time.sleep(0.5)

# Create queue
q = queue.Queue()

# Start a pool of 5 workers
for i in range(5):
    t = threading.Thread(target=worker, name='worker %i' % (i+1))
    t.start()

# Begin adding work to the queue
for i in range(50):
    q.put(i)

# Give threads time to run
print('Main thread sleeping')
time.sleep(5)
```

When run, this will produce the following output:

```
Running worker
Running worker
Running worker
Running worker
Running worker
Main thread sleeping
Worker <Thread(worker 1, started 130283832797456)> running with argument 0
Worker <Thread(worker 2, started 130283824404752)> running with argument 1
Worker <Thread(worker 3, started 130283816012048)> running with argument 2
Worker <Thread(worker 4, started 130283807619344)> running with argument 3
Worker <Thread(worker 5, started 130283799226640)> running with argument 4
Worker <Thread(worker 1, started 130283832797456)> running with argument 5
...
```

Consult the module's documentation for more details; the `Queue` class provides a featureful interface.

4.3.4 What kinds of global value mutation are thread-safe?

A *global interpreter lock* (GIL) is used internally to ensure that only one thread runs in the Python VM at a time. In general, Python offers to switch among threads only between bytecode instructions; how frequently it switches can be set via `sys.setswitchinterval()`. Each bytecode instruction and therefore all the C implementation code reached from each instruction is therefore atomic from the point of view of a Python program.

In theory, this means an exact accounting requires an exact understanding of the PVM bytecode implementation. In practice, it means that operations on shared variables of built-in data types (ints, lists, dicts, etc) that "look atomic" really are.

For example, the following operations are all atomic (L, L1, L2 are lists, D, D1, D2 are dicts, x, y are objects, i, j are ints):

```
L.append(x)
L1.extend(L2)
x = L[i]
x = L.pop()
L1[i:j] = L2
L.sort()
x = y
x.field = y
D[x] = y
D1.update(D2)
D.keys()
```

這些不是：

```
i = i+1
L.append(L[-1])
L[i] = L[j]
D[x] = D[x] + 1
```

Operations that replace other objects may invoke those other objects' `__del__()` method when their reference count reaches zero, and that can affect things. This is especially true for the mass updates to dictionaries and lists. When in doubt, use a mutex!

4.3.5 不能擺F全局直譯器鎖嗎？

The *global interpreter lock* (GIL) is often seen as a hindrance to Python's deployment on high-end multiprocessor server machines, because a multi-threaded Python program effectively only uses one CPU, due to the insistence that (almost) all Python code can only run while the GIL is held.

With the approval of **PEP 703** work is now underway to remove the GIL from the CPython implementation of Python. Initially it will be implemented as an optional compiler flag when building the interpreter, and so separate builds will be available with and without the GIL. Long-term, the hope is to settle on a single build, once the performance implications of removing the GIL are fully understood. Python 3.13 is likely to be the first release containing this work, although it may not be completely functional in this release.

The current work to remove the GIL is based on a fork of Python 3.9 with the GIL removed by Sam Gross. Prior to that, in the days of Python 1.5, Greg Stein actually implemented a comprehensive patch set (the ”free threading” patches) that removed the GIL and replaced it with fine-grained locking. Adam Olsen did a similar experiment in his python-safethread project. Unfortunately, both of these earlier experiments exhibited a sharp drop in single-thread performance (at least 30% slower), due to the amount of fine-grained locking necessary to compensate for the removal of the GIL. The Python 3.9 fork is the first attempt at removing the GIL with an acceptable performance impact.

The presence of the GIL in current Python releases doesn’t mean that you can’t make good use of Python on multi-CPU machines! You just have to be creative with dividing the work up between multiple *processes* rather than multiple *threads*. The `ProcessPoolExecutor` class in the new `concurrent.futures` module provides an easy way of doing so; the `multiprocessing` module provides a lower-level API in case you want more control over dispatching of tasks.

Judicious use of C extensions will also help; if you use a C extension to perform a time-consuming task, the extension can release the GIL while the thread of execution is in the C code and allow other threads to get some work done. Some standard library modules such as `zlib` and `hashlib` already do this.

An alternative approach to reducing the impact of the GIL is to make the GIL a per-interpreter-state lock rather than truly global. This was first implemented in Python 3.12 and is available in the C API. A Python interface to it is expected in Python 3.13. The main limitation to it at the moment is likely to be 3rd party extension modules, since these must be written with multiple interpreters in mind in order to be usable, so many older extension modules will not be usable.

4.4 輸入與輸出

4.4.1 如何🄵除檔案？(以及其他檔案問題...)

Use `os.remove(filename)` or `os.unlink(filename)`; for documentation, see the `os` module. The two functions are identical; `unlink()` is simply the name of the Unix system call for this function.

To remove a directory, use `os.rmdir()`; use `os.mkdir()` to create one. `os.makedirs(path)` will create any intermediate directories in `path` that don’t exist. `os.removedirs(path)` will remove intermediate directories as long as they’re empty; if you want to delete an entire directory tree and its contents, use `shutil.rmtree()`.

要重新命名檔案，請使用 `os.rename(old_path, new_path)`。

To truncate a file, open it using `f = open(filename, "rb+")`, and use `f.truncate(offset)`; offset defaults to the current seek position. There’s also `os.ftruncate(fd, offset)` for files opened with `os.open()`, where *fd* is the file descriptor (a small integer).

The `shutil` module also contains a number of functions to work on files including `copyfile()`, `copytree()`, and `rmtree()`.

4.4.2 如何🄵🄵檔案？

The `shutil` module contains a `copyfile()` function. Note that on Windows NTFS volumes, it does not copy alternate data streams nor resource forks on macOS HFS+ volumes, though both are now rarely used. It also doesn’t copy file permissions and metadata, though using `shutil.copy2()` instead will preserve most (though not all) of it.

4.4.3 如何讀取（或寫入）二進位制資料？

To read or write complex binary data formats, it’s best to use the `struct` module. It allows you to take a string containing binary data (usually numbers) and convert it to Python objects; and vice versa.

For example, the following code reads two 2-byte integers and one 4-byte integer in big-endian format from a file:

```
import struct

with open(filename, "rb") as f:
    s = f.read(8)
    x, y, z = struct.unpack(">hhl", s)
```

The '>' in the format string forces big-endian data; the letter 'h' reads one "short integer" (2 bytes), and 'l' reads one "long integer" (4 bytes) from the string.

For data that is more regular (e.g. a homogeneous list of ints or floats), you can also use the `array` module.

> **備🄵**
>
> To read and write binary data, it is mandatory to open the file in binary mode (here, passing `"rb"` to `open()`). If you use `"r"` instead (the default), the file will be open in text mode and `f.read()` will return `str` objects rather than `bytes` objects.

4.4.4 I can't seem to use os.read() on a pipe created with os.popen(); why?

`os.read()` is a low-level function which takes a file descriptor, a small integer representing the opened file. `os.popen()` creates a high-level file object, the same type returned by the built-in `open()` function. Thus, to read *n* bytes from a pipe *p* created with `os.popen()`, you need to use `p.read(n)`.

4.4.5 如何存取序列 (RS232) 連接埠？

對於 Win32、OSX、Linux、BSD、Jython、IronPython：

pyserial

對於 Unix，請參🄵 Mitch Chapman 的 Usenet 貼文：

https://groups.google.com/groups?selm=34A04430.CF9@ohioee.com

4.4.6 🄵什🄵關閉 sys.stdout (stdin, stderr) 🄵🄵有真正關閉它？

Python 檔案物件是低階 C 檔案描述器的高階抽象層。

For most file objects you create in Python via the built-in `open()` function, `f.close()` marks the Python file object as being closed from Python's point of view, and also arranges to close the underlying C file descriptor. This also happens automatically in `f`'s destructor, when `f` becomes garbage.

But stdin, stdout and stderr are treated specially by Python, because of the special status also given to them by C. Running `sys.stdout.close()` marks the Python-level file object as being closed, but does *not* close the associated C file descriptor.

To close the underlying C file descriptor for one of these three, you should first be sure that's what you really want to do (e.g., you may confuse extension modules trying to do I/O). If it is, use `os.close()`:

```
os.close(stdin.fileno())
os.close(stdout.fileno())
os.close(stderr.fileno())
```

或者你可以分🄵使用數字常數 0、1 和 2。

4.5 網路 (Network)/網際網路 (Internet) 程式

4.5.1 Python 有哪些 WWW 工具？

See the chapters titled internet and netdata in the Library Reference Manual. Python has many modules that will help you build server-side and client-side web systems.

A summary of available frameworks is maintained by Paul Boddie at https://wiki.python.org/moin/WebProgramming.

4.5.2 我應該使用什🄵模組來輔助🄵生 HTML？

You can find a collection of useful links on the Web Programming wiki page.

4.5.3 如何從 Python 🄵本發送郵件？

使用標准函式庫模組 smtplib。

Here's a very simple interactive mail sender that uses it. This method will work on any host that supports an SMTP listener.

```
import sys, smtplib

fromaddr = input("From: ")
toaddrs  = input("To: ").split(',')
print("Enter message, end with ^D:")
msg = ''
while True:
    line = sys.stdin.readline()
    if not line:
        break
    msg += line

# The actual mail send
server = smtplib.SMTP('localhost')
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

A Unix-only alternative uses sendmail. The location of the sendmail program varies between systems; sometimes it is /usr/lib/sendmail, sometimes /usr/sbin/sendmail. The sendmail manual page will help you out. Here's some sample code:

```
import os

SENDMAIL = "/usr/sbin/sendmail"  # sendmail location
p = os.popen("%s -t -i" % SENDMAIL, "w")
p.write("To: receiver@example.com\n")
p.write("Subject: test\n")
p.write("\n")  # blank line separating headers from body
p.write("Some text\n")
p.write("some more text\n")
sts = p.close()
if sts != 0:
    print("Sendmail exit status", sts)
```

4.5.4 How do I avoid blocking in the connect() method of a socket?

select 模組通常用於幫助處理 socket 上的非同步 I/O。

To prevent the TCP connect from blocking, you can set the socket to non-blocking mode. Then when you do the connect(), you will either connect immediately (unlikely) or get an exception that contains the error number as .errno. errno.EINPROGRESS indicates that the connection is in progress, but hasn't finished yet. Different OSes will return different values, so you're going to have to check what's returned on your system.

You can use the connect_ex() method to avoid creating an exception. It will just return the errno value. To poll, you can call connect_ex() again later -- 0 or errno.EISCONN indicate that you're connected -- or you can pass this socket to select.select() to check if it's writable.

備□

asyncio 模組提供了一個通用的單執行緒□發非同步函式庫，可用於編寫非阻塞網路程式碼。第三方 Twisted 函式庫是一種流行且功能豐富的替代方案。

4.6 資料庫

4.6.1 Python 中是否有任何資料庫套件的介面？

有的。

Interfaces to disk-based hashes such as DBM and GDBM are also included with standard Python. There is also the sqlite3 module, which provides a lightweight disk-based relational database.

Support for most relational databases is available. See the DatabaseProgramming wiki page for details.

4.6.2 How do you implement persistent objects in Python?

The pickle library module solves this in a very general way (though you still can't store things like open files, sockets or windows), and the shelve library module uses pickle and (g)dbm to create persistent mappings containing arbitrary Python objects.

4.7 數學和數值

4.7.1 如何在 Python 中生成隨機數？

標准模組 random 實作了一個隨機數生成器。用法很簡單：

```
import random
random.random()
```

這將回傳 [0, 1) 範圍□的隨機浮點數。

該模組中還有許多其他專用生成器，例如：

- randrange(a, b) 會選擇 [a, b) 範圍□的一個整數。
- uniform(a, b) 會選擇 [a, b) 範圍□的浮點數。
- normalvariate(mean, sdev) 對常態（高斯）分□進行取樣 (sample)。

一些更高階的函式會直接對序列進行操作，例如：

- choice(S) 會從給定序列中選擇一個隨機元素。
- shuffle(L) 會原地 (in-place) 打亂 list，即隨機排列它。

還有一個 Random 類□，你可以將它實例化以建立多個獨立的隨機數生成器。

CHAPTER 5

擴充/嵌入常見問題集

5.1 我可以在 C 中建立自己的函式嗎？

是的，你可以在 C 中建立包含函式、變數、例外甚至新型🄵的🄵建模組，extending-index 文件中有相關🄵明。

大多數中級或進階 Python 書籍也會涵蓋這個主題。

5.2 我可以在 C++ 中建立自己的函式嗎？

Yes, using the C compatibility features found in C++. Place `extern "C" { ... }` around the Python include files and put `extern "C"` before each function that is going to be called by the Python interpreter. Global or static C++ objects with constructors are probably not a good idea.

5.3 寫 C 很難；還有其他選擇嗎？

要編寫你自己的 C 擴充有許多替代方法，取🄵於你要執行的具體操作🄵何。

Cython and its relative Pyrex are compilers that accept a slightly modified form of Python and generate the corresponding C code. Cython and Pyrex make it possible to write an extension without having to learn Python's C API.

If you need to interface to some C or C++ library for which no Python extension currently exists, you can try wrapping the library's data types and functions with a tool such as SWIG. SIP, CXX Boost, or Weave are also alternatives for wrapping C++ libraries.

5.4 如何從 C 執行任意 Python 陳述式？

The highest-level function to do this is `PyRun_SimpleString()` which takes a single string argument to be executed in the context of the module `__main__` and returns `0` for success and `-1` when an exception occurred (including `SyntaxError`). If you want more control, use `PyRun_String()`; see the source for `PyRun_SimpleString()` in `Python/pythonrun.c`.

5.5 How can I evaluate an arbitrary Python expression from C?

Call the function `PyRun_String()` from the previous question with the start symbol `Py_eval_input`; it parses an expression, evaluates it and returns its value.

5.6 如何從 Python 物件中提取 C 值？

That depends on the object's type. If it's a tuple, `PyTuple_Size()` returns its length and `PyTuple_GetItem()` returns the item at a specified index. Lists have similar functions, `PyList_Size()` and `PyList_GetItem()`.

For bytes, `PyBytes_Size()` returns its length and `PyBytes_AsStringAndSize()` provides a pointer to its value and its length. Note that Python bytes objects may contain null bytes so C's `strlen()` should not be used.

要測試物件的型F，首先確保它不是 `NULL`，然後再使用 `PyBytes_Check()`、`PyTuple_Check()`、`PyList_Check()` 等函式。

There is also a high-level API to Python objects which is provided by the so-called 'abstract' interface -- read `Include/abstract.h` for further details. It allows interfacing with any kind of Python sequence using calls like `PySequence_Length()`, `PySequence_GetItem()`, etc. as well as many other useful protocols such as numbers (`PyNumber_Index()` et al.) and mappings in the PyMapping APIs.

5.7 如何使用 Py_BuildValue() 建立任意長度的元組？

這無法做到。請改用 `PyTuple_Pack()`。

5.8 如何從 C 呼叫物件的方法？

The `PyObject_CallMethod()` function can be used to call an arbitrary method of an object. The parameters are the object, the name of the method to call, a format string like that used with `Py_BuildValue()`, and the argument values:

```
PyObject *
PyObject_CallMethod(PyObject *object, const char *method_name,
                    const char *arg_format, ...);
```

This works for any object that has methods -- whether built-in or user-defined. You are responsible for eventually `Py_DECREF()`'ing the return value.

例如，使用引數 10、0 呼叫檔案物件的"seek" 方法（假設檔案物件指標F"f"）：

```
res = PyObject_CallMethod(f, "seek", "(ii)", 10, 0);
if (res == NULL) {
        ... an exception occurred ...
}
else {
        Py_DECREF(res);
}
```

Note that since `PyObject_CallObject()` *always* wants a tuple for the argument list, to call a function without arguments, pass "()" for the format, and to call a function with one argument, surround the argument in parentheses, e.g. "(i)".

5.9 我如何捕捉 PyErr_Print() 的輸出（或任何印出到 stdout/stderr 的東西）？

In Python code, define an object that supports the `write()` method. Assign this object to `sys.stdout` and `sys.stderr`. Call print_error, or just allow the standard traceback mechanism to work. Then, the output will go wherever your `write()` method sends it.

最簡單的方法是使用 `io.StringIO` 類🄵：

```
>>> import io, sys
>>> sys.stdout = io.StringIO()
>>> print('foo')
>>> print('hello world!')
>>> sys.stderr.write(sys.stdout.getvalue())
foo
hello world!
```

A custom object to do the same would look like this:

```
>>> import io, sys
>>> class StdoutCatcher(io.TextIOBase):
...     def __init__(self):
...         self.data = []
...     def write(self, stuff):
...         self.data.append(stuff)
...
>>> import sys
>>> sys.stdout = StdoutCatcher()
>>> print('foo')
>>> print('hello world!')
>>> sys.stderr.write(''.join(sys.stdout.data))
foo
hello world!
```

5.10 如何從 C 存取用 Python 編寫的模組？

You can get a pointer to the module object as follows:

```
module = PyImport_ImportModule("<modulename>");
```

If the module hasn't been imported yet (i.e. it is not yet present in `sys.modules`), this initializes the module; otherwise it simply returns the value of `sys.modules["<modulename>"]`. Note that it doesn't enter the module into any namespace -- it only ensures it has been initialized and is stored in `sys.modules`.

You can then access the module's attributes (i.e. any name defined in the module) as follows:

```
attr = PyObject_GetAttrString(module, "<attrname>");
```

Calling `PyObject_SetAttrString()` to assign to variables in the module also works.

5.11 How do I interface to C++ objects from Python?

Depending on your requirements, there are many approaches. To do this manually, begin by reading the "Extending and Embedding" document. Realize that for the Python run-time system, there isn't a whole lot of difference between C and C++ -- so the strategy of building a new Python type around a C structure (pointer) type will also work for C++ objects.

對於 C++ 函式庫，請參🄵寫 *C* 很難；還有其他選擇嗎？。

5.12 我使用安裝檔案新增了一個模組，但 make 失敗了；🄵什🄵？

Setup must end in a newline, if there is no newline there, the build process fails. (Fixing this requires some ugly shell script hackery, and this bug is so minor that it doesn't seem worth the effort.)

5.13 如何🄵擴充套件除錯？

When using GDB with dynamically loaded extensions, you can't set a breakpoint in your extension until your extension is loaded.

In your `.gdbinit` file (or interactively), add the command:

```
br _PyImport_LoadDynamicModule
```

然後，當你運行 GDB 時：

```
$ gdb /local/bin/python
gdb) run myscript.py
gdb) continue # repeat until your extension is loaded
gdb) finish   # so that your extension is loaded
gdb) br myfunction.c:50
gdb) continue
```

5.14 我想在我的 Linux 系統上編譯一個 Python 模組，但是缺少一些檔案。🄵什🄵？

Most packaged versions of Python omit some files required for compiling Python extensions.

在 Red Hat 上，請安裝 python3-devel RPM 來取得必要的檔案。

對於 Debian，運行 `apt-get install python3-dev`。

5.15 如何從「無效輸入」區分出「不完整輸入」？

Sometimes you want to emulate the Python interactive interpreter's behavior, where it gives you a continuation prompt when the input is incomplete (e.g. you typed the start of an "if" statement or you didn't close your parentheses or triple string quotes), but it gives you a syntax error message immediately when the input is invalid.

在 Python 中，你可以使用 `codeop` 模組，它充分模擬了剖析器 (parser) 的行🄵。像是 IDLE 就有使用它。

The easiest way to do it in C is to call `PyRun_InteractiveLoop()` (perhaps in a separate thread) and let the Python interpreter handle the input for you. You can also set the `PyOS_ReadlineFunctionPointer()` to point at your custom input function. See `Modules/readline.c` and `Parser/myreadline.c` for more hints.

5.16 如何找到未定義的 g++ 符號 __builtin_new 或 __pure_virtual？

To dynamically load g++ extension modules, you must recompile Python, relink it using g++ (change LINKCC in the Python Modules Makefile), and link your extension module using g++ (e.g., `g++ -shared -o mymodule.so mymodule.o`).

5.17 Can I create an object class with some methods implemented in C and others in Python (e.g. through inheritance)?

是的，你可以繼承🄵建類🄵，例如 `int`、`list`、`dict` 等。

Boost Python 函式庫（BPL，https://www.boost.org/libs/python/doc/index.html）提供了一種從 C++ 執行此操作的方法（即你可以使用 BPL 來繼承用 C++ 編寫的擴充類🄵）。

CHAPTER 6

在 Windows 使用 Python 的常見問答集

6.1 如何在 Windows 作業系統🄵運行 Python 程式？

這個問題的答案可能有點🄵雜。如果你經常使用「命令提示字元」執行程式，那這對你來🄵不會是什🄵難事。如果不然，那就需要更仔細的🄵明了。

除非你使用某種整合開發環境，否則你最終將會在所謂的「命令提示字元視窗」中 打字輸入 Windows 命令。通常，你可以透過從搜尋欄中搜尋 cmd 來建立這樣的視窗。你應該能🄵認出何時已🄵動這樣的視窗，因🄵你將看到 Windows「命令提示字元」，它通常看起來像這樣：

```
C:\>
```

第一個字母可能不一樣，且後面也可能還有其他🄵容，因此你可能會很容易看到類似以下的文字：

```
D:\YourName\Projects\Python>
```

取🄵於你的電腦如何被設置，以及你最近對它所做的其他操作。一旦你🄵動了這樣一個視窗，你就即將可以運行 Python 程式了。

你需要了解，你的 Python 🄵本必須被另一個稱🄵 Python 直譯器的程序來處理。直譯器會讀取你的🄵本，將其編譯🄵位元組碼，然後執行該位元組碼以運行你的程式。那🄵，你要如何安排直譯器來處理你的 Python 呢？

首先，你需要確保你的命令視窗會將單字"py" 識🄵🄵🄵動直譯器的指令。如果你已經開🄵一個命令視窗，則你應該試試輸入命令 py 🄵按下 return 鍵：

```
C:\Users\YourName> py
```

然後，你應該看到類似下面的🄵容：

```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

你已經🄵動直譯器中的「互動模式」。這表示你能🄵以互動方式輸入 Python 陳述式或運算式，🄵在等待時執行或計算它們。這是 Python 最🄵大的功能之一。輸入你所選的幾個運算式🄵查看結果，可以檢驗此功能：

```
>>> print("Hello")
Hello
>>> "Hello" * 3
'HelloHelloHello'
```

許多人將互動模式作🄵方便但可高度程式化的計算機。如果你要結束互動式 Python 對話，請呼叫 exit() 函式或是按住 Ctrl 鍵再輸入 Z，然後按下"Enter" 鍵以返回 Windows 命令提示字元。

你可能還會發現你有一個開始功能表項目，像是：開始 ▸ 所有程式 ▸ *Python 3.x* ▸ *Python*（命令行），它會讓你在一個新視窗中看到 >>> 提示字元。如果是這樣，該視窗將在你呼叫 exit() 函式或輸入 Ctrl-Z 字元後消失；Windows 正在該視窗中運行單一個「python」命令，🄵在你終止直譯器時將其關閉。

現在我們知道 py 命令已被識🄵，而你可以將你的 Python 🄵本提供給它。你必須🄵 Python 🄵本給定🄵對路徑或相對路徑。假設你的 Python 🄵本位於桌面上，🄵被命名🄵 hello.py，且你的命令提示字元在你的家目🄵 (home directory) 中順利地被開🄵，那🄵你就會看到類似以下的🄵容：

```
C:\Users\YourName>
```

因此，現在你將透過鍵入 py 加上🄵本路徑，來使用 py 命令將你的🄵本提供給 Python：

```
C:\Users\YourName> py Desktop\hello.py
hello
```

6.2 如何使 Python 🄵本可以執行？

在 Windows 上，標准的 Python 安裝程式已將.py 副檔名與一種檔案類型 (Python.File) 進行關聯，🄵🄵該檔案類型提供一個開🄵命令來運行直譯器 (D:\Program Files\Python\python.exe "%1" %*)。這足以使🄵本能以類似'foo.py' 的形式從命令提示字元被執行。如果你希望能🄵簡單地輸入'foo' 來執行🄵本，而不用加上副檔名，則需要將.py 新增至 PATHEXT 環境變數中。

6.3 🄵什🄵 Python 有時需要這🄵長的時間才能開始？

通常 Python 在 Windows 上🄵動得非常快，但偶爾會有一些錯誤報告，🄵容是 Python 突然開始需要很長的時間才能🄵動。這種情形更令人費解，因🄵 Python 在其他 Windows 系統上可以正常工作，而那些系統似乎也有相同的配置。

這個問題可能是由發生此問題的電腦上的病毒檢查軟體配置錯誤所引起的。目前已知某些病毒掃描程式，在它們被配置🄵監視來自檔案系統的所有讀取時，會引入兩個數量級的🄵動負擔。請試著檢查你系統上的病毒掃描軟體配置，以確保它們的配置確實相同。當 McAfee 被配置🄵掃描所有檔案系統的讀取活動時，它是一個特定的違規者。

6.4 如何從 Python 🄵本🄵作可執行檔？

請參🄵*如何從 Python 🄵本建立獨立的二進位檔案？*該章節列出了用於🄵作可執行檔的工具清單。

6.5 *.pyd 檔是否與 DLL 相同？

是的，.pyd 檔類似於 dll，但也有一些區🄵。如果你有一個名🄵 foo.pyd 的 DLL，則它必須具有函式 PyInit_foo()。接著你可以將"import foo" 寫入 Python 🄵本，Python 將會搜尋 foo.pyd（以及 foo.py、foo.pyc），如果 Python 找到它，將會嘗試呼叫 PyInit_foo() 來將它初始化。你🄵不會將你的.exe 與 foo.lib 連結 (link)，因🄵這會導致 Windows 要求 DLL 的存在。

請注意，foo.pyd 的搜尋路徑是 PYTHONPATH，與 Windows 用於搜尋 foo.dll 的路徑不同。此外，foo.pyd 不需存在即可運行你的程式，然而如果你將程式連結了一個 dll，則該 dll 會是必要的。當然，如果你想要 import foo，foo.pyd 就是必要的。在 DLL 中，連結是以 __declspec(dllexport) 在原始碼中被宣告。在.pyd 中，連結是在一個可用函式的 list（串列）中被定義。

6.6 如何將 Python 嵌入 Windows 應用程式中？

在 Windows 應用程式中嵌入 Python 直譯器的過程可以總結如下：

1. **不要**直接將 Python 建置到你的.exe 檔中。在 Windows 上，Python 必須是一個 DLL 來處理模組的 import，而那些模組本身也是 DLL。（這是第一個未正式記載的關鍵事實。）請改🄵連結到 `pythonNN.dll`；它通常被安裝在 `C:\Windows\System` 中。*NN* 是 Python 版本，例如”33” 就是指 Python 3.3。

   你可以透過兩種不同的方式連結到 Python。載入時連結 (load-time linking) 表示要連結到 `pythonNN.lib`，而執行環境連結 (run-time linking) 表示要連結到 `pythonNN.dll`。（一般🄵解：`pythonNN.lib` 是 `pythonNN.dll` 相對應的所謂”import lib”。它只會🄵鏈接器定義符號。）

   執行環境連結大大簡化了連結選項；所有事情都會發生在執行環境。你的程式碼必須使用 Windows `LoadLibraryEx()` 常式 (routine) 來載入 `pythonNN.dll`。該程式碼也必須用 Windows `GetProcAddress()` 常式所取得的指標，來使用 `pythonNN.dll` 中的（即🄵 Python C API 的）存取常式和資料。對於任何呼叫 Python C API 常式的 C 程式碼，巨集可以讓使用這些指標的過程透明化。

2. 如果你使用 SWIG，則可輕松地建立一個 Python「擴充模組」，該模組將使應用程式的資料和 method（方法）可供 Python 使用。SWIG 會🄵你處理幾乎所有的繁瑣細節。結果就是，你會將 C 程式碼連結到你的.exe 檔🄵（！）你**不必**建立 DLL 檔，而這也簡化了連結。

3. SWIG 將建立一個 init 函式（一個 C 函式），其名稱取🄵於擴充模組的名稱。例如，如果模組的名稱是 leo，則該 init 函式會命名🄵 initleo()。如果你使用 SWIG shadow class（類🄵），則 init 函式會命名🄵 initleoc()。這會初始化被 shadow class 所用的大多數隱藏的 helper class。

   你可以將步驟 2 中的 C 程式碼連結到.exe 檔中的原因是，呼叫初始化函式就等效於 import 模組進 Python！（這是第二個未正式記載的關鍵事實。）

4. 簡而言之，你可以使用以下程式碼，以你的擴充模組初始化 Python 直譯器。

```
#include <Python.h>
...
Py_Initialize();  // Initialize Python.
initmyAppc();  // Initialize (import) the helper class.
PyRun_SimpleString("import myApp");  // Import the shadow class.
```

5. Python 的 C API 有兩個問題，如果你使用 MSVC（用於建置 pythonNN.dll 的編譯器）以外的編譯器，這些問題將會變得明顯。

   問題 1：使用 `FILE *` 引數的所謂「非常高階」的函式，在多編譯器 (multi-compiler) 的環境中會無法作用，因🄵每個編譯器對 `struct FILE` 的概念不同。從實作的觀點來看，這些都是非常 _ 低階 _ 的函式。

   問題 2：SWIG 在🄵 void 函式🄵生包裝函式 (wrapper) 時會🄵生以下程式碼：

```
Py_INCREF(Py_None);
_resultobj = Py_None;
return _resultobj;
```

   唉，Py_None 是一個巨集，它會延伸到一個參照，指向 pythonNN.dll 🄵部的一種稱🄵 _Py_NoneStruct 的🄵雜資料結構。同樣的，此程式碼在多編譯器環境中將會失效。請將此類程式碼替🄵🄵：

```
return Py_BuildValue("");
```

   有可能可以使用 SWIG 的 `%typemap` 命令以自動進行程式碼變更，雖然我未曾這樣正常運作過（我是一個完全的 SWIG 新手）。

6. 使用 Python shell 🄵本從你的 Windows 應用程式🄵部建造一個 Python 直譯器視窗不是一個好主意；該視窗將會獨立於你的應用程式視窗系統。與其如此，你（或 wxPythonWindow class）應該要建立一個「本機」直譯器視窗。將該視窗連接到 Python 直譯器是很容易的。你可以將 Python 的 i/o 重定向 (redirect) 到可支援讀取和寫入的任何物件，因此你只需要一個包含 read() 和 write() method 的 Python 物件（在你的擴充模組中被定義）就可以了。

6.7 如何防止編輯器在我的 Python 原始碼中插入 tab？

FAQ 不建議使用 tab，且 Python 風格指南 **PEP 8** 建議在分散式 Python 程式碼使用 4 個空格；這也是 Emacs 的 python 模式預設值。

在任何編輯器下，將 tab 和空格混合都是一個壞主意。MSVC 在這方面也是一樣，且可以輕松配置🄵使用空格：選擇工具 ‣ 選項 ‣ *Tabs*，然後對於「預設」檔案類型，將「Tab 大小」和「縮排大小」設定🄵 4，然後選擇「插入空格」單選鈕。

如果混合 tab 和空格造成前導空白字元出現問題，則 Python 會引發 `IndentationError` 或 `TabError`。你也可以運行 `tabnanny` 模組，在批次模式下檢查目🄵樹。

6.8 如何在不阻塞的情🄵下檢查 keypress？

使用 `msvcrt` 模組。這是一個標准的 Windows 專用擴充模組。它定義了一個函式 `kbhit()`，該函式會檢查是否出現鍵盤打擊 (keyboard hit)，以及函式 `getch()`，該函式會取得一個字元且不會將其印出。

6.9 如何解🄵遺漏 api-ms-win-crt-runtime-l1-1-0.dll 的錯誤？

使用 Windows 8.1 或更早版本時，若尚未安裝所有的更新，則可能會在 Python 3.5 以上的版本發生這種情🄵。首先要確保你的作業系統仍受支援🄵且是最新的，如果這無法解🄵問題，請造訪 Microsoft 支援頁面以取得關於手動安裝 C Runtime 更新的指南。

CHAPTER 7

圖形使用者介面常見問答集

7.1 圖形使用者介面 (GUI) 的常見問題

7.2 Python 有哪些 GUI 套件？

Python 的標准版本會包含一個 Tcl/Tk 小工具集 (widget set) 的物件導向介面，稱🄵 tkinter。這可能是最容易安裝（因🄵它已包含在 Python 的大多數二進制發行版本中）和使用的。有關 Tk 的詳細資訊（包含指向原始碼的指標），請參🄵 Tcl/Tk 首頁。Tcl/Tk 在 macOS、Windows 和 Unix 平台上是完全可🄵 (portable) 的。

根據你要使用的平台，還有其他幾種選擇。在 python wiki 上可以找到一份跨平台的以及各平台專屬的 GUI 框架清單。

7.3 Tkinter 的問答

7.3.1 如何凍結 Tkinter 應用程式？

凍結 (freeze) 是一個能建立獨立應用程式的工具。在凍結 Tkinter 應用程式時，該應用程式不是真正的獨立，因🄵該應用程式仍然需要 Tcl 和 Tk 函式庫。

將應用程式與 Tcl 和 Tk 函式庫一🄵發送是一種解🄵方法，🄵在執行環境 (run-time) 使用 `TCL_LIBRARY` 和 `TK_LIBRARY` 環境變數來指向該函式庫。

Various third-party freeze libraries such as py2exe and cx_Freeze have handling for Tkinter applications built-in.

7.3.2 是否可以在等待 I/O 時處理 Tk 事件？

在 Windows 以外的平台上是可以的，你甚至不需要執行緒！但是，你必須稍微調整你的 I/O 程式碼。Tk 具有等效於 Xt 的 `XtAddInput()` 的函式呼叫，它能讓你記🄵一個回呼 (callback) 函式，當 I/O 在一個檔案描述符 (file descriptor) 上可進行時，該函式將會從 Tk mainloop 被呼叫。請參🄵 tkinter-file-handlers。

7.3.3 我無法讓鍵🄵結 (key binding) 在 Tkinter 中作用：🄵什🄵？

一個經常聽到的抱怨是，🄵管事件處理程式 (event handler) 已經🄵結到帶有 `bind()` method 的事件，但在按下相應的鍵時，該事件也🄵有被處理。

最常見的原因是，🄵結到的小工具🄵🄵有「鍵盤焦點 (keyboard focus)」。請查看 Tk 🄵明文件中關於焦點命令的🄵述。通常，點擊一個小工具，會讓它得到鍵盤焦點（但不適用於標🄵；請參🄵 takefocus 選項）。

CHAPTER 8

「🄵什🄵 Python 被安裝在我的機器上 ?」常見問答集

8.1 什🄵是 Python ?

Python 是一種程式語言。它被使用於不同種類的應用程式中。因🄵 Python 屬於容易學習的語言，它在一些高中和大學課程中被用作介紹程式語言的工具；但它也被專業的軟體開發人員所使用，例如 Google、美國太空總署與盧卡斯電影公司。

若你想學習更多關於 Python 的知識，可以先從 Python 初學者指引開始🄵讀。

8.2 🄵什🄵 Python 被安裝在我的機器上?

若你發現曾安裝 Python 於系統中，但不記得何時安裝過，那有可能是透過以下幾種途徑安裝的。

- 也許其他使用此電腦的使用者想要學習撰寫程式🄵且安裝了 Python；你需要回想一下誰曾經使用此機器且可能進行安裝。
- 安裝於機器的第三方應用程式可能以 Python 語言撰寫🄵且安裝了 Python。這樣的應用程式🄵不少，從 GUI 程式到網路伺服器和管理者🄵本都有。
- 一些安裝 Windows 的機器也被安裝 Python。截至撰寫此文件的當下，我們得知 HP 與 Compaq 出廠的機器都預設安裝 Python。顯然的 HP 與 Compaq 部分的管理工具程式是透過 Python 語言所撰寫。
- 許多相容於 Unix 系統，例如 macOS 和一些 Linux 發行版本預設安裝 Python；它被包含在基礎安裝🄵。

8.3 我能🄵自行🄵除 Python 嗎 ?

需要依據 Python 的安裝方式🄵定。

若有人是有意地安裝 Python，你可自行移除移除它，這不會造成其他影響。Windows 作業系統中，請於控制台 (Control Panel) 中尋找新增/移除程式來解除安裝。

若 Python 是透過第三方應用程式安裝時，你也可自行移除，不過該應用程式將無法正常執行。你應該使用應用程式解除安裝功能而非直接🄵除 Python。

當作業系統預設安裝 Python，不建議移除它。對你而言某些工具程式是重要不可或缺的，若自行移除它，透過 Python 撰寫的工具程式將無法正常執行。重新安裝整個系統，才能再次解🄵這些問題。

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。
- 建常數 `Ellipsis`。

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。

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）。

在執行環境 (runtime)，區域變數的釋無法被存取，但全域變數、class 屬性和函式的解，會分被儲存在模組、class 和函式的 `__annotations__` 特殊屬性中。

請參*variable annotation*、*function annotation*、**PEP 484** 和 **PEP 526**，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。

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訪問各個項目時，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.

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)

```

（繼續下一頁）

(繼續上一頁)

```
@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。

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。

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）
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（例如標准輸入／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。建立檔案物件的標准方法是使用 `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，來編譯當前的模組。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（與那些每次調用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，無論是標准的或是第三方的，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**。

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 的標准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。

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塊和包的非常🄵大的方法（請記住 help(x)）。更多互動式模式相關資訊請見 tut-interac。

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）

An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict`, *file objects*, and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements *sequence* semantics.

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□。

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 代碼頁（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]                    # indexed access
1024
>>> sys.float_info.max_exp               # named field access
1024
>>> isinstance(sys.float_info, tuple)    # kind of tuple
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）

A *package* which serves only as a container for subpackages. Namespace packages may have no physical representation, and specifically are not like a *regular package* because they have no `__init__.py` file.

Namespace packages allow several individually installable packages to have a common parent package. Otherwise, it is recommended to use a *regular package*.

For more information, see **PEP 420** and 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□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□不會無端地□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□「最後的解□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□方案。

這個過程使得標准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□ **PEP 411** 了解更多細節。

provisional package（暫行套件）

請參□*provisional API*（暫行 API）。

Python 3000

Python 3.x 系列版本的□稱（很久以前創造的，當時第 3 版的發布是在□遠的未來。）也可以縮寫□「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□代物件的所有元素進行□圈。許多其他語言□□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):
```

(繼續下一頁)

(繼續上一頁)

```
...                 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)」🄵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🄵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。

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🄵。

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。

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。

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）。

text encoding（文字編碼）
Python 中的字串是一個 Unicode 碼點 (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。

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：

```python
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：

```python
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。

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🄵一種稱🄵 ABC 語言的後繼者。🄵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，🄵在那🄵發🄵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 🄵成立了 BeOpen PythonLabs 團隊。同年十月，PythonLabs 團隊轉移到 Digital Creations，後來成🄵 Zope Corporation。2001 年，Python 軟體基金會（PSF，見 https://www.python.org/psf/）成立，這是一個專🄵擁有 Python 相關的智慧財🄵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🄵 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🄵。

發🄵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🄵

(1) GPL 相容🄵不表示我們是在 GPL 下發🄵 Python。不像 GPL，所有的 Python 授權都可以讓你發🄵修改後的版本，但不一定要使你的變更成🄵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 © 2001-2024 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/）F，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 執行追蹤

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` 擴充時設定了 `--with-system-expat`，否則該擴充會用一個內含 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` 擴充程式時設定了 `--with-system-libffi`，否則該擴充會用一個包含 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，則該擴充會用一個包含 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.
```

APPENDIX D

版權宣告

Python 和這份🄵明文件的版權：

Copyright © 2001-2024 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com 保留一切權利。

Copyright © 1995-2000 Corporation for National Research Initiatives 保留一切權利。

Copyright © 1991-1995 Stichting Mathematisch Centrum 保留一切權利。

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

- ..., **69**
- >>>, **69**
- __future__, **75**
- __slots__, **82**
- 環境變數
  - PATH, 48
  - PYTHON_GIL, 76
  - PYTHONDONTWRITEBYTECODE, 34

A

- abstract base class（抽象基底類🄵）, **69**
- annotation（🄵釋）, **69**
- argument（引數）, **69**
  - 與 parameter（參數）的差🄵, 12
- asynchronous context manager（非同步情境管理器）, **70**
- asynchronous generator iterator（非同步🄵生器🄵代器）, **70**
- asynchronous generator（非同步🄵生器）, **70**
- asynchronous iterable（非同步可🄵代物件）, **70**
- asynchronous iterator（非同步🄵代器）, **70**
- attribute（屬性）, **70**
- awaitable（可等待物件）, **70**

B

- BDFL, **70**
- binary file（二進位檔案）, **70**
- borrowed reference（借用參照）, **71**
- bytecode（位元組碼）, **71**
- bytes-like object（類位元組串物件）, **71**

C

- callable（可呼叫物件）, **71**
- callback（回呼）, **71**
- C-contiguous（C 連續的）, 72
- class variable（類🄵變數）, **71**
- class（類🄵）, **71**
- closure variable（閉包變數）, **71**
- complex number（🄵數）, **72**
- context management protocol（情境管理協定）, **72**
- context manager（情境管理器）, **72**
- context variable（情境變數）, **72**
- context（情境）, **72**
- contiguous（連續的）, **72**
- coroutine function（協程函式）, **72**
- coroutine（協程）, **72**
- CPython, **72**
- current context, **72**

D

- decorator（裝飾器）, **72**
- descriptor（描述器）, **73**
- dictionary comprehension（字典綜合運算）, **73**
- dictionary view（字典檢視）, **73**
- dictionary（字典）, **73**
- docstring（🄵明字串）, **73**
- duck-typing（鴨子型🄵）, **73**

E

- EAFP, **73**
- expression（運算式）, **73**
- extension module（擴充模組）, **73**

F

- f-string（f 字串）, **73**
- file object（檔案物件）, **74**
- file-like object（類檔案物件）, **74**
- filesystem encoding and error handler（檔案系統編碼和錯誤處理函式）, **74**
- finder（尋檢器）, **74**
- floor division（向下取整除法）, **74**
- Fortran contiguous（Fortran 連續的）, 72
- free threading（自由執行緒）, **74**
- free variable（自由變數）, **74**
- function annotation（函式🄵釋）, **74**
- function（函式）, **74**

G

- garbage collection（垃圾回收）, **75**
- generator expression（🄵生器運算式）, **75**
- generator iterator（🄵生器🄵代器）, **75**
- generator（🄵生器）, **75**
- generic function（泛型函式）, **75**
- generic type（泛型型🄵）, **75**

GIL, 75
global interpreter lock（全域直譯器鎖）, 75

H

hash-based pyc（雜□架構的 pyc）, 76
hashable（可雜□的）, 76

I

IDLE, 76
immortal（不滅）, 76
immutable（不可變物件）, 76
import path（引入路徑）, 76
importer（引入器）, 76
importing（引入）, 76
interactive（互動的）, 76
interpreted（直譯的）, 76
interpreter shutdown（直譯器關閉）, 76
iterable（可□代物件）, 77
iterator（□代器）, 77

K

key function（鍵函式）, 77
keyword argument（關鍵字引數）, 77

L

lambda, 77
LBYL, 77
list comprehension（串列綜合運算）, 78
list（串列）, 77
loader（載入器）, 78
locale encoding（區域編碼）, 78

M

magic
    method（方法）, 78
magic method（魔術方法）, 78
mapping（對映）, 78
meta path finder（元路徑尋檢器）, 78
metaclass（元類□）, 78
method resolution order（方法解析順序）, 78
method（方法）, 78
    magic, 78
    special, 83
module spec（模組規格）, 79
module（模組）, 78
MRO, 79
mutable（可變物件）, 79

N

named tuple（附名元組）, 79
namespace package（命名空間套件）, 79
namespace（命名空間）, 79
nested scope（巢狀作用域）, 79
new-style class（新式類□）, 79

O

object（物件）, 79
optimized scope（最佳化作用域）, 80

P

package（套件）, 80
parameter（參數）, 80
    與 argument（引數）的差□, 12
PATH, 48
path based finder（基於路徑的尋檢器）, 80
path entry finder（路徑項目尋檢器）, 80
path entry hook（路徑項目□）, 80
path entry（路徑項目）, 80
path-like object（類路徑物件）, 81
PEP, 81
portion（部分）, 81
positional argument（位置引數）, 81
provisional API（暫行 API）, 81
provisional package（暫行套件）, 81
Python 3000, 81
Python Enhancement Proposals
    PEP 1, 81
    PEP 5, 5
    PEP 8, 8, 32, 64
    PEP 238, 74
    PEP 278, 84
    PEP 302, 78
    PEP 343, 72
    PEP 362, 70, 80
    PEP 373, 4
    PEP 387, 3
    PEP 411, 81
    PEP 420, 79, 81
    PEP 443, 75
    PEP 483, 75
    PEP 484, 69, 75, 84
    PEP 492, 70, 72
    PEP 498, 73
    PEP 519, 81
    PEP 525, 70
    PEP 526, 69, 84
    PEP 572, 39
    PEP 585, 75
    PEP 602, 4
    PEP 683, 76
    PEP 703, 51, 74, 76
    PEP 3116, 84
    PEP 3147, 34
    PEP 3155, 81
PYTHON_GIL, 76
PYTHONDONTWRITEBYTECODE, 34
Pythonic（Python 風格的）, 81

Q

qualified name（限定名稱）, 81

R

reference count（參照計數）, 82
regular package（正規套件）, 82
REPL, 82

S

sequence（序列），82
set comprehension（集合綜合運算），82
single dispatch（單一調度），82
slice（切片），82
soft deprecated（軟性🄵用），82
special
    method（方法），83
special method（特殊方法），83
statement（陳述式），83
static type checker（🄵態型🄵檢查器），83
strong reference（🄵參照），83

T

text encoding（文字編碼），83
text file（文字檔案），83
triple-quoted string（三引號🄵字串），83
type alias（型🄵🄵名），83
type hint（型🄵提示），84
type（型🄵），83

U

universal newlines（通用🄵行字元），84

V

variable annotation（變數🄵釋），84
virtual environment（🄵擬環境），84
virtual machine（🄵擬機器），84

Z

Zen of Python（Python 之🄵），84