

Python Frequently Asked Questions

發  3.11.13

Guido van Rossum and the Python development team

7 月 08, 2025

Python Software Foundation
Email: docs@python.org

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?	2
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?	3
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?	4
1.1.10	有⌈有 Python 專屬的新聞群組或郵件討論群?	4
1.1.11	如何取得 Python 的 beta 測試版本?	4
1.1.12	如何提交 Python 的錯誤報告和修補程式?	4
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⌈?	5
1.1.16	⌈什⌈要取名⌈ Python?	5
1.1.17	我需要喜歡「Monty Python 的飛行馬戲團」嗎?	5
1.2	在真實世界中的 Python	5
1.2.1	Python 的穩定性如何?	5
1.2.2	有多少人在使用 Python?	5
1.2.3	有⌈有任何重要的專案使用 Python 完成開發?	5
1.2.4	Python 未來預期會有哪些新的開發?	6
1.2.5	對 Python 提出不相容的變更建議是否適當?	6
1.2.6	Python 對於入門的程式設計師而言是否⌈好的語言?	6
2	程式開發常見問答集	9
2.1	常見問題	9
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?	9
2.1.2	有⌈有工具能⌈幫忙找 bug 或執行⌈態分析?	10
2.1.3	如何由 Python 腳本創建能獨立運行的二進制程序?	10
2.1.4	是否有 Python 編碼標準或風格指南?	10
2.2	語言核心內容	10
2.2.1	變量明明有值, 為什麼還會出現 UnboundLocalError?	10
2.2.2	Python 的區域變數和全域變數有什⌈規則?	12

2.2.3	为什么在循环中定义参数各异的 lambda 都返回相同的结果?	12
2.2.4	如何跨模块共享全局变量?	13
2.2.5	导入模块的“最佳实践”是什么?	13
2.2.6	为什么对象之间会共享默认值?	14
2.2.7	如何将可选参数或关键字参数从一个函数传递到另一个函数?	14
2.2.8	引數 (arguments) 和參數 (parameters) 有什麼區別?	15
2.2.9	什麼更改 list 'y' 也會更改 list 'x'?	15
2.2.10	如何编写带有输出参数的函数 (按照引用调用)?	16
2.2.11	如何在 Python 中创建高阶函数?	17
2.2.12	如何在 Python 中创建物件?	18
2.2.13	如何找到物件的方法或屬性?	18
2.2.14	我的程式碼如何發現物件的名稱?	18
2.2.15	逗号运算符的优先级是什么?	19
2.2.16	是否有等效於 C 的“?:”三元運算子?	19
2.2.17	是否可以用 Python 编写让人眼晕的单行程序?	19
2.2.18	函数形参列表中的斜杠 (/) 是什么意思?	20
2.3	數字和字串	20
2.3.1	如何指定十六進位和八進位整數?	20
2.3.2	什麼 -22 // 10 回傳 -3?	21
2.3.3	我如何获得 int 字面属性而不是 SyntaxError?	21
2.3.4	如何將字串轉成數字?	21
2.3.5	如何將數字轉成字串?	22
2.3.6	如何修改字符串?	22
2.3.7	如何使用字符串调用函数/方法?	22
2.3.8	是否有与 Perl 的 chomp() 等效的方法, 用于从字符串中删除尾随换行符?	23
2.3.9	是否有 scanf() 或 sscanf() 的等价函数?	23
2.3.10	'UnicodeDecodeError' 或 'UnicodeEncodeError' 錯誤是什麼意思?	24
2.3.11	我能以奇数个反斜杠来结束一个原始字符串吗?	24
2.4	性能	24
2.4.1	我的程序太慢了。该如何加快速度?	24
2.4.2	将多个字符串连接在一起的最有效方法是什么?	25
2.5	序列 (元组/列表)	25
2.5.1	如何在元组和列表之间进行转换?	25
2.5.2	什么是负数索引?	26
2.5.3	序列如何以逆序遍历?	26
2.5.4	如何从列表中删除重复项?	26
2.5.5	如何从列表中删除多个项?	26
2.5.6	如何在 Python 中创建数组?	27
2.5.7	如何创建多维列表?	27
2.5.8	我如何将一个方法或函数应用于由对象组成的序列?	28
2.5.9	为什么 a_tuple[i] += ['item'] 会引发异常?	28
2.5.10	我想做一个复杂的排序: 能用 Python 进行施瓦茨变换吗?	29
2.5.11	如何根据另一个列表的值对某列表进行排序?	29
2.6	物件	30
2.6.1	什麼是類 (class)?	30
2.6.2	什麼是方法 (method)?	30
2.6.3	什麼是 self?	30
2.6.4	如何检查对象是否为给定类或其子类的一个实例?	30
2.6.5	什麼是委托?	31
2.6.6	如何在扩展基类的派生类中调用基类中定义的方法?	32
2.6.7	如何让代码更容易对基类进行修改?	32
2.6.8	如何创建静态类数据和静态类方法?	32
2.6.9	在 Python 中如何重载构造函数 (或方法)?	33
2.6.10	在用 __spam 的时候得到一个类似 _SomeClassName__spam 的错误信息。	34

2.6.11	类定义了 <code>__del__</code> 方法，但是删除对象时没有调用它。	34
2.6.12	如何获取给定类的所有实例的列表？	34
2.6.13	为什么 <code>id()</code> 的结果看起来不是唯一的？	34
2.6.14	什么情况下可以依靠 <code>is</code> 运算符进行对象的身份相等性测试？	35
2.6.15	子类如何控制不可变实例中存储的资料？	36
2.6.16	我该如何缓存方法调用？	37
2.7	模組	38
2.7.1	如何创建 <code>.pyc</code> 文件？	38
2.7.2	如何找到当前模块名称？	38
2.7.3	如何让模块相互导入？	39
2.7.4	<code>__import__('x.y.z')</code> 回傳 <code><module 'x'></code> ，那我怎得到 <code>z</code> ？	40
2.7.5	对已导入的模块进行了编辑并重新导入，但变动没有得以体现。这是为什么？	40
3	設計和歷史常見問答集	41
3.1	什 Python 使用縮排將陳述式進行分組？	41
3.2	什我會從簡單的數學運算得到奇怪的結果？	42
3.3	何浮點數運算如此不精確？	42
3.4	什 Python 字串不可變動？	42
3.5	何「self」在方法 (method) 定義和呼叫時一定要明確使用？	43
3.6	何我不能在運算式 (expression) 中使用指派運算？	43
3.7	何 Python 對於一些功能實作使用方法 (像是 <code>list.index()</code>)，另一些使用函式 (像是 <code>len(list)</code>)？	43
3.8	何 <code>join()</code> 是字串方法而非串列 (list) 或元組 (tuple) 方法？	44
3.9	例外處理有多快？	44
3.10	什 Python 有 <code>switch</code> 或 <code>case</code> 陳述式？	45
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？	45
3.12	何 <code>lambda</code> 運算式不能包含陳述式？	45
3.13	Python 可以被編譯成機器語言、C 語言或其他種語言嗎？	46
3.14	Python 如何管理記憶體？	46
3.15	何 CPython 不使用更多傳統的垃圾回收機制？	46
3.16	當 CPython 結束時，何所有的記憶體不會被釋放？	47
3.17	何要把元組 (tuple) 和串列 (list) 分成兩個資料型態？	47
3.18	串列 (list) 在 CPython 中是怎實作的？	47
3.19	字典 (dictionaries) 在 CPython 中是怎實作的？	47
3.20	何字典的鍵一定是不可變的？	48
3.21	何 <code>list.sort()</code> 不是回傳排序過的串列？	49
3.22	如何在 Python 中指定和制使用一個介面規範 (interface spec)？	49
3.23	何有 <code>goto</code> 語法？	49
3.24	何純字串 (r-string) 不能以反斜結尾？	50
3.25	何 Python 有屬性賦值的 <code>with</code> 陳述式？	50
3.26	何生器 (generator) 不支援 <code>with</code> 陳述式？	51
3.27	何 <code>if</code> 、 <code>while</code> 、 <code>def</code> 、 <code>class</code> 陳述式需要冒號？	51
3.28	何 Python 允許在串列和元組末端加上逗號？	51
4	函式庫和擴充功能的常見問題	53
4.1	常見函式問題	53
4.1.1	如何找到可以用来做 X 任务的模块或应用？	53
4.1.2	哪可以找到 <code>math.py</code> (<code>socket.py</code> , <code>regex.py</code> , 等...) 來源檔案？	53
4.1.3	我如何使 Python script 執行在 Unix？	54
4.1.4	是否有適用於 Python 的 <code>curses/termcap</code> 套件？	54
4.1.5	Python 中存在类似 C 的 <code>onexit()</code> 函数的东西吗？	54
4.1.6	为什么我的信号处理函数不能工作？	55
4.2	常見課題	55
4.2.1	如何測試 Python 程式或元件？	55
4.2.2	怎样用 <code>docstring</code> 创建文档？	55

4.2.3	怎样一次只获取一个按键？	56
4.3	執行緒	56
4.3.1	如何使用執行緒編寫程式？	56
4.3.2	我的執行緒似乎都「有」運行：「什麼」？	56
4.3.3	如何将任务分配给多个工作线程？	57
4.3.4	怎样修改全局变量是线程安全的？	58
4.3.5	不能擺「全局直譯器鎖」嗎？	58
4.4	輸入與輸出	59
4.4.1	如何「除」檔案？（以及其他檔案問題...）	59
4.4.2	如何「開」檔案？	59
4.4.3	如何讀取（或寫入）二進位制資料？	59
4.4.4	似乎 <code>os.popen()</code> 创建的管道不能使用 <code>os.read()</code> ，这是为什么？	60
4.4.5	如何存取序列 (RS232) 連接埠？	60
4.4.6	为什么关闭 <code>sys.stdout</code> (<code>stdin</code> , <code>stderr</code>) 并不会真正关掉它？	60
4.5	網路 (Network)/網際網路 (Internet) 程式	60
4.5.1	Python 有哪些 WWW 工具？	60
4.5.2	如何模擬 CGI 表單送出 (submission) (METHOD=POST)？	61
4.5.3	我應該使用什麼模組來輔助「生」HTML？	61
4.5.4	如何從 Python 「本」發送郵件？	61
4.5.5	socket 的 <code>connect()</code> 方法怎样避免阻塞？	62
4.6	資料庫	62
4.6.1	Python 中有数据库包的接口吗？	62
4.6.2	在 Python 中如何实现持久化对象？	63
4.7	數學和數值	63
4.7.1	如何在 Python 中生成隨機數？	63
5	擴充/嵌入常見問題集	65
5.1	我可以在 C 中建立自己的函式嗎？	65
5.2	我可以在 C++ 中建立自己的函式嗎？	65
5.3	寫 C 很難；還有其他選擇嗎？	65
5.4	如何從 C 執行任意 Python 陳述式？	66
5.5	如何在 C 中对任意 Python 表达式求值？	66
5.6	如何從 Python 物件中提取 C 值？	66
5.7	如何使用 <code>Py_BuildValue()</code> 建立任意長度的元組？	66
5.8	如何從 C 呼叫物件的方法？	66
5.9	我如何捕捉 <code>PyErr_Print()</code> 的輸出（或任何印出到 <code>stdout/stderr</code> 的東西）？	67
5.10	如何從 C 存取用 Python 編寫的模組？	68
5.11	如何在 Python 中对接 C++ 对象？	68
5.12	我使用安裝檔案新增了一個模組，但 <code>make</code> 失敗了；「什麼」？	68
5.13	如何「擴充」套件除錯？	68
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。「什麼」？	69
5.15	如何從「無效輸入」區分出「不完整輸入」？	69
5.16	如何找到未定義的 g++ 符號 <code>__builtin_new</code> 或 <code>__pure_virtual</code> ？	69
5.17	能否创建一个对象类，其中部分方法在 C 中实现，而其他方法在 Python 中实现（例如通过继承）？	69
6	在 Windows 使用 Python 的常見問答集	71
6.1	如何在 Windows 作業系統「開」運行 Python 程式？	71
6.2	如何使 Python 「本」可以執行？	72
6.3	「什麼」Python 有時需要這「長」的時間才能開始？	72
6.4	如何從 Python 「本」作可執行檔？	73
6.5	*.pyd 檔是否與 DLL 相同？	73
6.6	如何將 Python 嵌入 Windows 應用程式中？	73
6.7	如何防止編輯器在我的 Python 原始碼中插入 tab？	74

6.8	如何在不阻塞的情況下檢查 keypress?	74
6.9	如何解決遺漏 api-ms-win-crt-runtime-l1-1-0.dll 的錯誤?	74
7	圖形使用者介面常見問答集	75
7.1	圖形使用者介面 (GUI) 的常見問題	75
7.2	Python 有哪些 GUI 套件?	75
7.3	Tkinter 的問答	75
7.3.1	如何凍結 Tkinter 應用程式?	75
7.3.2	是否可以在等待 I/O 時處理 Tk 事件?	76
7.3.3	我無法讓鍵結 (key binding) 在 Tkinter 中作用: 為什麼?	76
8	「為什麼 Python 被安裝在我的機器上？」常見問答集	77
8.1	什麼是 Python?	77
8.2	為什麼 Python 被安裝在我的機器上?	77
8.3	我能自行刪除 Python 嗎?	78
A	術語表	79
B	關於這些開明文件	95
B.1	Python 文件的貢獻者們	95
C	沿革與授權	97
C.1	軟體沿革	97
C.2	關於存取或以其他方式使用 Python 的合約條款	98
C.2.1	用於 PYTHON 3.11.13 的 PSF 授權合約	98
C.2.2	用於 PYTHON 2.0 的 BEOPEN.COM 授權合約	99
C.2.3	用於 PYTHON 1.6.1 的 CNRI 授權合約	100
C.2.4	用於 PYTHON 0.9.0 至 1.2 的 CWI 授權合約	101
C.2.5	用於 PYTHON 3.11.13 開明文件程式碼的 ZERO-CLAUSE BSD 授權	101
C.3	被收買軟體的授權與致謝	102
C.3.1	Mersenne Twister	102
C.3.2	Sockets	103
C.3.3	非同步 socket 服務	103
C.3.4	Cookie 管理	104
C.3.5	執行追尋	104
C.3.6	UUencode 與 UUdecode 函式	105
C.3.7	XML 遠端程序呼叫	105
C.3.8	test_epoll	106
C.3.9	Select kqueue	106
C.3.10	SipHash24	107
C.3.11	strtod 與 dtoa	107
C.3.12	OpenSSL	108
C.3.13	expat	111
C.3.14	libffi	111
C.3.15	zlib	112
C.3.16	cfuhash	113
C.3.17	libmpdec	113
C.3.18	W3C C14N 測試套件	114
C.3.19	audioop	115
C.3.20	asyncio	115
D	版權宣告	117
	索引	119

一般的 Python 常見問答集

1.1 一般資訊

1.1.1 什麼是 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、動態型別 (dynamic typing)、非常高階的動態資料型別，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 是可攜的 (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。

要尋找更多內容，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 什麼是 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 是免稅的。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](#)[的](#)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[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準函式庫，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 各版本會被編號為“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準備階段，會發布一系列開發版本，標示為 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。

句話說，所有標記為 2.0aN 的版本都在標記為 2.0bN 的版本之前，而 2.0bN 版本都在標記為 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，成為「a0」版，例如「2.4a0」。

請參閱 [Developer's Guide](#) 获取更多有關開發流程的信息，並參閱 [PEP 387](#) 了解更多有關 Python 的向下兼容策略的信息。另請參閱有關 `sys.version`、`sys.hexversion` 和 `sys.version_info` 的文檔。

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](https://www.python.org/downloads/) 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 格式的說明文件、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，請參閱 Python 開發人員指南中的“Getting Started”段落。

1.1.8 我要如何取得 Python 的說明文件？

Python 目前穩定版本的標準說明文件可在 <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。

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

1.1.9 我從來沒有寫過程式，有 Python 的教學？

有許多可用的教學和書籍。標準說明文件包括 `tutorial-index`。

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

1.1.10 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閱。

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，請參閱 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> 的問題追蹤系統。

關於如何開發 Python 的更多資訊，請參閱 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。請參閱 `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 專案的基礎建設遍佈世界各地，由 Python 基礎建設團隊管理。詳細資訊在此。

1.1.16 什麼要取名 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 認為他需要一個簡短、獨特且略帶神秘的名字，因此他決定將該語言稱作 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，並且會保證介面在一系列的錯誤修正發布版本中維持不變。

最新的穩定發布版本隨時都可以在 [Python 下載頁面上](#) 找到。Python 有兩個生產就緒 (production-ready) 的版本：2.x 和 3.x。推薦的版本是 3.x，此版本能被那些最廣泛使用的函式庫所支援。雖然 2.x 仍然被廣泛使用，但它已不再被維護。

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況。

comp.lang.python 新聞群組非常活躍，但並非所有 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，請參閱 <https://www.python.org/about/success>。藉由查詢過去的 Python 會議記錄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。內部使用 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準函式庫，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 有一個大型的標準函式庫，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準的四功能計算機和平衡檢驗程式。透過使用標準函式庫，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。使用標準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__',
 '__getattr__', '__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',
```

(繼續下一頁)

(繼續上一頁)

```
↪ 'reverse', 'sort']

>>> 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。

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。

以下介绍了一些 Python 的调试器, 用内置函数 `breakpoint()` 即可切入这些调试器中。

`pdb` 模块是一个简单但是够用的控制台模式 Python 调试器。它是标准 Python 库的一部分, 并且已收录于库参考手册。你也可以通过使用 `pdb` 代码作为样例来编写你自己的调试器。

作为标准 Python 发行版组成部分的 IDLE 交互式开发环境 (通常位于 `Tools/scripts/idle3`), 包括一个图形化的调试器。

PythonWin 是一种 Python IDE, 其中包含了一个基于 `pdb` 的 GUI 调试器。PythonWin 的调试器会为断点着色, 并提供了相当多的超酷特性, 例如调试非 PythonWin 程序等。PythonWin 是 `pywin32` 项目的组成部分, 也是 `ActivePython` 发行版的组成部分。

Eric 是一个基于 PyQt 和 Scintilla 编辑组件的 IDE。

`trepan3k` 是一个类似 `gdb` 的调试器。

Visual Studio Code 是包含了调试工具的 IDE, 并集成了版本控制软件。

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 有工具能幫忙找 bug 或執行態分析？

有的。

`Pylint` 和 `Pyflakes` 可执行基本检查来帮助你尽早捕捉漏洞。

静态类型检查器例如 `Mypy`, `Pyre` 和 `Pytype` 可以检查 Python 源代码中的类型提示。

2.1.3 如何由 Python 脚本创建能独立运行的二进制程序？

如果只是想要一个独立的程序，以便用户不必预先安装 Python 即可下载和运行它，则不需要将 Python 编译成 C 代码。有许多工具可以检测程序所需的模块，并将这些模块与 Python 二进制程序捆绑在一起生成单个可执行文件。

一种方案是使用 `freeze` 工具，它以 `Tools/freeze` 的形式包含在 Python 源代码树中。它可将 Python 字节码转换为 C 数组；你可以使用 C 编译器将你的所有模块嵌入到一个新程序中，再将其与标准 Python 模块进行链接。

它的工作原理是递归扫描源代码，获取两种格式的 `import` 语句，并在标准 Python 路径和源码目录（用于内置模块）检索这些模块。然后，把这些模块的 Python 字节码转换为 C 代码（可以利用 `marshal` 模块转换为代码对象的数组初始化器），并创建一个定制的配置文​​件，该文件仅包含程序实际用到的内置模块。然后，编译生成的 C 代码并将其与 Python 解释器的其余部分链接，形成一个自给自足的二进制文件，其功能与 Python 脚本代码完全相同。

下列包可以用于帮助创建控制台和 GUI 的可执行文件：

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

2.1.4 是否有 Python 编码标准或风格指南？

有的。标准库模块所要求的编码风格记录于 [PEP 8](#) 之中。

2.2 语言核心内容

2.2.1 变量明明有值，为什么还会出现 `UnboundLocalError`？

当在函数内部某处添加了一条赋值语句，因而导致之前正常工作的代码报出 `UnboundLocalError` 错误，这确实有点令人惊讶。

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

原因就是, 当对某作用域内的变量进行赋值时, 该变量将成为该作用域内的局部变量, 并覆盖外部作用域中的同名变量。由于 `foo` 的最后一条语句为 `x` 分配了一个新值, 编译器会将其识别为局部变量。因此, 前面的 `print(x)` 试图输出未初始化的局部变量, 就会引发错误。

在上面的示例中, 可以将外部作用域的变量声明为全局变量以便访问:

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

与类和实例变量貌似但不一样, 其实以上是在修改外部作用域的变量值, 为了提示这一点, 这里需要显式声明一下。

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

你可以使用 `nonlocal` 关键字在嵌套作用域中执行类似的操作:

```
>>> 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麼規則？

函数内部只作引用的 Python 变量隐式视为全局变量。如果在函数内部任何位置为变量赋值，则除非明确声明为全局变量，否则均将其视为局部变量。

起初尽管有点令人惊讶，不过考虑片刻即可释然。一方面，已分配的变量要求加上 `global` 可以防止意外的副作用发生。另一方面，如果所有全局引用都要加上 `global`，那处处都得用上 `global` 了。那么每次对内置函数或导入模块中的组件进行引用时，都得声明为全局变量。这种杂乱会破坏 `global` 声明用于警示副作用的有效性。

2.2.3 为什么在循环中定义的参数各异的 lambda 都返回相同的结果？

假设用 `for` 循环来定义几个取值各异的 `lambda`（即便是普通函数也一样）：

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

以上会得到一个包含 5 个 `lambda` 函数的列表，这些函数将计算 `x**2`。大家或许期望，调用这些函数会分别返回 0、1、4、9 和 16。然而，真的试过就会发现，他们都会返回 16：

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

这是因为 `x` 不是 `lambda` 函数的内部变量，而是定义于外部作用域中的，并且 `x` 是在调用 `lambda` 时访问的——而不是在定义时访问。循环结束时 `x` 的值是 4，所以此时所有的函数都将返回 `4**2`，即 16。通过改变 `x` 的值并查看 `lambda` 的结果变化，也可以验证这一点。

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

为了避免发生上述情况，需要将值保存在 `lambda` 局部变量，以使其不依赖于全局 `x` 的值：

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

以上 `n=x` 创建了一个新的 `lambda` 本地变量 `n`，并在定义 `lambda` 时计算其值，使其与循环当前时点的 `x` 值相同。这意味着 `n` 的值在第 1 个 `lambda` 中为 0，在第 2 个 `lambda` 中为 1，在第 3 个中为 2，依此类推。因此现在每个 `lambda` 都会返回正确结果：

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

请注意，上述表现并不是 `lambda` 所特有的，常规的函数也同样适用。

2.2.4 如何跨模块共享全局变量？

在单个程序中跨模块共享信息的规范方法是创建一个特殊模块（通常称为 `config` 或 `cfg`）。只需在应用程序的所有模块中导入该 `config` 模块；然后该模块就可当作全局名称使用了。因为每个模块只有一个实例，所以对该模块对象所做的任何更改将会在所有地方得以体现。例如：

`config.py`：

```
x = 0    # Default value of the 'x' configuration setting
```

`mod.py`：

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

`main.py`：

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

请注意，出于同样的原因，使用模块也是实现单例设计模式的基础。

2.2.5 导入模块的“最佳实践”是什么？

通常请勿使用 `from modulename import *`。因为这会扰乱 `importer` 的命名空间，且会造成未定义名称更难以被 `Linter` 检查出来。

请在代码文件的首部就导入模块。这样代码所需的模块就一目了然了，也不用考虑模块名是否在作用域内的问题。每行导入一个模块则增删起来会比较容易，每行导入多个模块则更节省屏幕空间。

按如下顺序导入模块就是一种好做法：

1. 标准库模块——例如：`sys`、`os`、`argparse`、`re` 等。
2. 第三方库模块（安装于 `Python site-packages` 目录中的内容）——例如：`dateutil`、`requests`、`PIL`、`Image` 等。
3. 本地开发的模块

为了避免循环导入引发的问题，有时需要将模块导入语句移入函数或类的内部。`Gordon McMillan` 的说法如下：

当两个模块都采用“`import <module>`”的导入形式时，循环导入是没有问题的。但如果第 2 个模块想从第 1 个模块中取出一个名称（“`from module import name`”）并且导入处于代码的最顶层，那导入就会失败。原因是第 1 个模块中的名称还不可用，这时第 1 个模块正忙于导入第 2 个模块呢。

如果只是在函数中用到第 2 个模块，那这时将导入语句移入该函数内部即可。当调用到导入语句时，第 1 个模块将已经完成初始化，第 2 个模块就可以进行导入了。

如果某些模块是平台相关的，可能还需要把导入语句移出最顶级代码。这种情况下，甚至有可能无法导入文件首部的所有模块。于是在对应的平台相关代码中导入正确的模块，就是一种不错的选择。

只有为了避免循环导入问题，或有必要减少模块初始化时间时，才把导入语句移入类似函数定义内部的局部作用域。如果根据程序的执行方式，许多导入操作不是必需的，那么这种技术尤其有用。如果模块仅在某个函数中用到，可能还要将导入操作移入该函数内部。请注意，因为模块有一次初始化过程，所以第一次加载模块的代价可能会比较高，但多次加载几乎没有什么花费，代价只是进行几次字典检索而已。即使模块名超出了作用域，模块在 `sys.modules` 中也是可用的。

2.2.6 为什么对象之间会共享默认值？

新手程序员常常中招这类 Bug。请看以下函数：

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

第一次调用此函数时，mydict 中只有一个数据项。第二次调用 mydict 则会包含两个数据项，因为 foo() 开始执行时，mydict 中已经带有一个数据项了。

大家往往希望，函数调用会为默认值创建新的对象。但事实并非如此。默认值只会在函数定义时创建一次。如果对象发生改变，就如上例中的字典那样，则后续调用该函数时将会引用这个改动的对象。

按照定义，不可变对象改动起来是安全的，诸如数字、字符串、元组和 None 之类。而可变对象的改动则可能引起困惑，例如字典、列表和类实例等。

因此，不把可变对象用作默认值是一种良好的编程做法。而应采用 None 作为默认值，然后在函数中检查参数是否为 None 并新建列表、字典或其他对象。例如，代码不应如下所示：

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

但是：

```
def foo(mydict=None):
    if mydict is None:
        mydict = {} # create a new dict for local namespace
```

参数默认值的特性有时会很有用处。如果有个函数的计算过程会比较耗时，有一种常见技巧是将每次函数调用的参数和结果缓存起来，并在同样的值被再次请求时返回缓存的值。这种技巧被称为“memoize”，实现代码可如下所示：

```
# 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
```

也可以不用参数默认值来实现，而是采用全局的字典变量；这取决于个人偏好。

2.2.7 如何将可选参数或关键字参数从一个函数传递到另一个函数？

请利用函数参数列表中的标识符 * 和 ** 归集实参；结果会是元组形式的位置实参和字典形式的关键字实参。然后就可利用 * 和 ** 在调用其他函数时传入这些实参：

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

2.2.8 引數 (arguments) 和參數 (parameters) 有什麼區別？

形參 是由出現在函數定義中的名稱來定義的，而參數 則是在調用函數時實際傳入的值。形參定義了一個函數能接受什麼參數種類。例如，對於以下函數定義：

```
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 為什麼更改 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]
```

你可能想知道為什麼將一個元素附加到 `y` 時也會改變 `x`。

產生這個結果的原因有兩個：

- 1) 變量只是指向對象的一個名稱。執行 `y = x` 並不會創建列表的副本——而只是創建了一個新變量 `y`，並指向 `x` 所指的同一對象。這就意味着只存在一個列表對象，`x` 和 `y` 都是對它的引用。
- 2) `list` 是 *mutable*，這意味著你可以變更它們的內容。

在調用 `append()` 之後，該可變對象的內容由 `[]` 變為 `[10]`。由於兩個變量引用了同一對象，因此用其中任意一個名稱所訪問到的都是修改後的值 `[10]`。

如果把賦給 `x` 的對象換成一個不可變對象：

```
>>> x = 5 # ints are immutable
>>> y = x
>>> x = x + 1 # 5 can't be mutated, we are creating a new object here
>>> x
6
>>> y
5
```

可見這時 `x` 和 `y` 就不再相等了。因為整數是 *immutable* 對象，在執行 `x = x + 1` 時，並不會修改整數對象 5，給它加上 1；而是創建了一個新的對象（整數對象 6）並將其賦給 `x`（也就是改變了 `x` 所指向的對象）。在賦值完成後，就有了兩個對象（整數對象 6 和 5）和分別指向他倆的兩個變量（`x` 現在指向 6 而 `y` 仍然指向 5）。

某些操作（例如 `y.append(10)` 和 `y.sort()`）是改變原對象，而看上去相似的另一些操作（例如 `y = y + [10]` 和 `sorted(y) <sorted>`）則是創建新對象。通常在 Python 中（以及在標準庫的所有代碼中）會改變原對象的方法將返回 `None`（）以幫助避免混淆這兩種不同類型的操作。因此如果你錯誤地使用了 `y.sort()` 並期望它將返回一個經過排序的 `y` 的副本，你得到的結果將會是 `None`，這將導致你的程序產生一個容易診斷的錯誤。

不过还存在一类操作，用不同的类型执行相同的操作有时会发生不同的行为：即增量赋值运算符。例如，`+=` 会修改列表，但不会修改元组或整数（`a_list += [1, 2, 3]` 与 `a_list.extend([1, 2, 3])` 同样都会改变 `a_list`，而 `some_tuple += (1, 2, 3)` 和 `some_int += 1` 则会创建新的对象）。

☞ 句話 ☞：

- 对于一个可变对象（`list`、`dict`、`set` 等等），可以利用某些特定的操作进行修改，所有引用它的变量都会反映出改动情况。
- 对于一个不可变对象（`str`、`int`、`tuple` 等），所有引用它的变量都会给出相同的值，但所有改变其值的操作都将返回一个新的对象。

如要知道两个变量是否指向同一个对象，可以利用 `is` 运算符或内置函数 `id()`。

2.2.10 如何编写带有输出参数的函数（按照引用调用）？

请记住，Python 中的实参是通过赋值传递的。由于赋值只是创建了对象的引用，所以调用方和被调用方的参数名都不存在别名，本质上也就不存在按引用调用的方式。通过以下几种方式，可以得到所需的效果。

1) 返回一个元组：

```
>>> def func1(a, b):
...     a = 'new-value'           # a and b are local names
...     b = b + 1                 # assigned to new objects
...     return a, b              # return new values
...
>>> x, y = 'old-value', 99
>>> func1(x, y)
('new-value', 100)
```

这差不多是最明晰的解决方案了。

2) 使用全局变量。这不是线程安全的方案，不推荐使用。

3) 传递一个可变（即可原地修改的）对象：

```
>>> def func2(a):
...     a[0] = 'new-value'       # 'a' references a mutable list
...     a[1] = a[1] + 1          # changes a shared object
...
>>> args = ['old-value', 99]
>>> func2(args)
>>> args
['new-value', 100]
```

4) 传入一个接收可变对象的字典：

```
>>> 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) 或者把值用类实例封装起来：


```
>>> 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}
```

没有什么理由要把问题搞得这么复杂。

最佳选择就是返回一个包含多个结果值的元组。

2.2.11 如何在 Python 中创建高阶函数？

有两种选择：嵌套作用域、可调用对象。假定需要定义 `linear(a,b)`，其返回结果是一个计算出 $a \cdot x + b$ 的函数 $f(x)$ 。采用嵌套作用域的方案如下：

```
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
```

采用这两种方案时：

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

都会得到一个可调用对象，可实现 `taxes(10e6) == 0.3 * 10e6 + 2`。

可调用对象的方案有个缺点，就是速度稍慢且生成的代码略长。不过值得注意的是，同一组可调用对象能够通过继承来共享签名（类声明）：

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

物件可以封装多个方法的状态：

```
class counter:
```

(繼續下一頁)

(繼續上一頁)

```

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？

一般情况下，用 `copy.copy()` 或 `copy.deepcopy()` 基本就可以了。并不是所有对象都支持复制，但多数是可以的。

某些对象可以用更简便的方法进行复制。比如字典对象就提供了 `copy()` 方法：

```
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？

对于一个用户定义类的实例 `x`，`dir(x)` 将返回一个按字母顺序排列的名称列表，其中包含实例属性及由类定义的方法和属性。

2.2.14 我的程式碼如何發現物件的名稱？

一般而言这是无法实现的，因为对象并不存在真正的名称。赋值本质上是把某个名称绑定到某个值上；`def` 和 `class` 语句同样如此，只是值换成了某个可调用对象。比如以下代码：

```

>>> class A:
...     pass
...
>>> B = A
>>> a = B()
>>> b = a
>>> print(b)
<__main__.A object at 0x16D07CC>
>>> print(a)
<__main__.A object at 0x16D07CC>

```

可以不太严谨地说上述类有一个名称：即使它绑定了两个名称并通过名称 `B` 唤起所创建的实例仍将被报告为类 `A` 的实例。但是，没有办法肯定地说实例的名称是 `a` 还是 `b`，因为这两个名称都被绑定到同一个值上了。

代码一般没有必要去“知晓”某个值的名称。通常这种需求预示着还是改变方案为好，除非真的是要编写内审程序。

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個的名字一樣：（物件）本身不能告訴你它的名字，它也不關心 - 所以找出它叫什麼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 逗号运算符的优先级是什么？

逗号不是 Python 的运算符。请看以下例子：

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

由于逗号不是运算符，而只是表达式之间的分隔符，因此上述代码就相当于：

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

而不是：

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

对于各种赋值运算符（=、+= 等）来说同样如此。他们并不是真正的运算符，而只是赋值语句中的语法分隔符。

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
```

在 Python 2.5 引入上述语法之前，通常的做法是使用逻辑运算符：

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

然而这种做法并不保险，因为当 `on_true` 为布尔值“假”时，结果将会出错。所以肯定还是采用 ... `if` ... `else` ... 形式为妙。

2.2.17 是否可以用 Python 编写让人眼晕的单行程序？

可以。这一般是通过在 `lambda` 中嵌套 `lambda` 来实现的。请参阅以下三个示例，它们是基于 Ulf Bartelt 的代码改写的：

```
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%x, range(2, int(pow(y, 0.5)+1))), 1), range(2, 1000)))))
```

(繼續下一頁)

(繼續上一頁)

```
# 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, 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 函数形参列表中的斜杠 (/) 是什么意思？

函数参数列表中的斜杠表示在它之前的形参都是仅限位置形参。仅限位置形参没有可供外部使用的名称。在调用接受仅限位置形参的函数时，参数将只根据其位置被映射到形参上。例如，`divmod()` 就是一个接受仅限位置形参的函数。它的文档说明是这样的：

```
>>> 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.
```

形参列表尾部的斜杠说明，两个形参都是仅限位置形参。因此，用关键字参数调用 `divmod()` 将会引发错误：

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

2.3 数字和字符串

2.3.1 如何指定十六进制和八进制整数？

要给出八进制数，需在八进制数值前面加上一个零和一个小写或大写字母“o”作为前缀。例如，要将变量“a”设为八进制的“10”（十进制的 8），写法如下：

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

十六进制数也很简单。只要在十六进制数前面加上一个零和一个小写或大写的字母“x”。十六进制数中的字母可以为大写或小写。比如在 Python 解释器中输入：

```
>>> a = 0xa5
>>> a
165
>>> b = 0xB2
>>> b
178
```

2.3.2 什 -22 // 10 回傳 -3 ?

这主要是为了让 $i \% j$ 的正负与 j 一致，如果期望如此，且期望如下等式成立：

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

那么整除就必须返回向下取整的结果。C 语言同样要求保持这种一致性，于是编译器在截断 $i // j$ 的结果时需要让 $i \% j$ 的正负与 i 一致。

对于 $i \% j$ 来说 j 为负值的应用场景实际上是非常少的。而 j 为正值的情况则非常多，并且实际上在所有情况下让 $i \% j$ 的结果为 ≥ 0 会更有用处。如果现在时间为 10 时，那么 200 小时前应是几时？ $-190 \% 12 == 2$ 是有用处的； $-190 \% 12 == -10$ 则是会导致意外的漏洞。

2.3.3 我如何获得 int 字面属性而不是 SyntaxError ?

尝试以正式方式查找一个 int 字面值属性会发生 SyntaxError 因为句点会被当作是小数点：

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

解决办法是用空格或括号将字词与句号分开。

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

2.3.4 如何將字串轉成數字 ?

对于整数，可使用内置的 `int()` 类型构造器，例如 `int('144') == 144`。类似地，可使用 `float()` 转换为浮点数，例如 `float('144') == 144.0`。

默认情况下，这些操作会将数字按十进制来解读，因此 `int('0144') == 144` 为真值，而 `int('0x144')` 会引发 `ValueError`。`int(string, base)` 接受第二个可选参数指定转换的基数，例如 `int('0x144', 16) == 324`。如果指定基数为 0，则按 Python 规则解读数字：前缀 `0o` 表示八进制，而 `0x` 表示十六进制。

如果只是想把字符串转为数字，请不要使用内置函数 `eval()`。`eval()` 的速度慢很多且存在安全风险：别人可能会传入带有不良副作用的 Python 表达式。比如可能会传入 `__import__('os').system("rm -rf $HOME")`，这会把 home 目录给删了。

`eval()` 还有把数字解析为 Python 表达式的后果，因此如 `eval('09')` 将会导致语法错误，因为 Python 不允许十进制数带有前导 0（0 除外）。

2.3.5 如何將數字轉成字串？

例如，要把数字 144 转换为字符串 '144'，可使用内置类型构造器 `str()`。如果你需要十六进制或八进制表示形式，可使用内置函数 `hex()` 或 `oct()`。更复杂的格式化方式，请参阅 `f-strings` 和 `formatstrings` 等章节，例如 `"{:04d}".format(144)` 将产生 '0144' 而 `"{: .3f}".format(1.0/3.0)` 将产生 '0.333'。

2.3.6 如何修改字符串？

无法修改，因为字符串是不可变对象。在大多数情况下，只要将各个部分组合起来构造出一个新字符串即可。如果需要一个能原地修改 Unicode 数据的对象，可以试试 `io.StringIO` 对象或 `array` 模块：

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

2.3.7 如何使用字符串调用函数/方法？

有多种技巧可供选择。

- 最好的做法是采用一个字典，将字符串映射为函数。其主要优势就是字符串不必与函数名一样。这也是用来模拟 case 结构的主要技巧：

```
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
```

- 利用内置函数 `getattr()`：

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

请注意 `getattr()` 可用于任何对象，包括类、类实例、模块等等。

标准库就多次使用了这个技巧，例如：

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

    def do_bar(self):
        ...

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

- 用 `locals()` 解析出函数名：

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

fname = "myFunc"

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

2.3.8 是否有与 Perl 的 `chomp()` 等效的方法，用于从字符串中删除尾随换行符？

可以使用 `S.rstrip("\r\n")` 从字符串 `S` 的末尾删除所有的换行符，而不删除其他尾随空格。如果字符串 `S` 表示多行，且末尾有几个空行，则将删除所有空行的换行符：

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

由于通常只在一次读取一行文本时才需要这样做，所以使用 `S.rstrip()` 这种方式工作得很好。

2.3.9 是否有 `scanf()` 或 `sscanf()` 的等价函数？

没有。

对于简单的输入解析，最容易的做法通常是用字符串对象的 `split()` 方法将一行内容按空白分隔符拆分为多个单词再用 `int()` 或 `float()` 将十进制数值字符串转换为数值。`split()` 支持可选的“sep”形参，适用于分隔符不是空白符的情况。

对于更复杂的输入解析，正则表达式相比 C 的 `sscanf` 更为强大也更为适合。

2.3.10 'UnicodeDecodeError' 或 'UnicodeEncodeError' 錯誤是什麼意思？

請參閱 [unicode-howto](#)。

2.3.11 我能以奇数个反斜杠来结束一个原始字符串吗？

以奇数个反斜杠结尾的原始字符串将会转义用于标记字符串的引号：

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

有几种绕过此问题的办法。其中之一是使用常规字符串以及双反斜杠：

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

另一种办法是将一个包含被转义反斜杠的常规字符串拼接到原始字符串上：

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

在 Windows 上还可以使用 `os.path.join()` 来添加反斜杠：

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

请注意虽然在确定原始字符串的结束位置时反斜杠会对引号进行“转义”，但在解析原始字符串的值时并不会发生转义。也就是说，反斜杠会被保留在原始字符串的值中：

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

另请参阅 [语言参考](#) 中的规范说明。

2.4 性能

2.4.1 我的程序太慢了。该如何加快速度？

总的来说，这是个棘手的问题。在进一步讨论之前，首先应该记住以下几件事：

- 不同的 Python 实现具有不同的性能特点。本 FAQ 着重解答的是 [CPython](#)。
- 不同操作系统可能会有不同表现，尤其是涉及 I/O 和多线程时。
- 在尝试优化代码之前，务必要先找出程序中的热点（请参阅 `profile` 模块）。
- 编写基准测试脚本，在寻求性能提升的过程中就能实现快速迭代（请参阅 `timeit` 模块）。
- 强烈建议首先要保证足够高的代码测试覆盖率（通过单元测试或其他技术），因为复杂的优化有可能会導致代码回退。

话虽如此，Python 代码的提速还是有很多技巧的。以下列出了一些普适性的原则，对于让性能达到可接受的水平会有很大帮助：

- 相较于试图对全部代码铺开做微观优化，优化算法（或换用更快的算法）可以产出更大的收益。
- 使用正确的数据结构。参考 `bltin-types` 和 `collections` 模块的文档。
- 如果标准库已为某些操作提供了基础函数，则可能（当然不能保证）比所有自编的函数都要快。对于用 C 语言编写的基础函数则更是如此，比如内置函数和一些扩展类型。例如，一定要用内置方法 `list.sort()` 或 `sorted()` 函数进行排序（某些高级用法的示例请参阅 `sortingshowto`）。
- 抽象往往会造成中间层，并会迫使解释器执行更多的操作。如果抽象出来的中间层级太多，工作量超过了要完成的有效任务，那么程序就会被拖慢。应该避免过度的抽象，而且往往也会对可读性产生不利影响，特别是当函数或方法比较小的时候。

如果你已经达到纯 Python 允许的限制，那么有一些工具可以让你走得更远。例如，`Cython` 可以将稍加修改的 Python 代码版本编译为 C 扩展，并能在许多不同的平台上使用。`Cython` 可以利用编译（和可选的类型标注）来让你的代码显著快于解释运行时的速度。如果你对自己的 C 编程技能有信心，还可以自行编写 C 扩展模块。

也参考：

有個 [wiki 頁面](#) 專門介紹效能改進小提示。

2.4.2 将多个字符串连接在一起的最有效方法是什么？

`str` 和 `bytes` 对象是不可变的，因此连接多个字符串的效率会很低，因为每次连接都会创建一个新的对象。一般情况下，总耗时与字符串总长是二次方的关系。

如果要连接多个 `str` 对象，通常推荐的方案是先全部放入列表，最后再调用 `str.join()`：

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

（还有一种合理高效的习惯做法，就是利用 `io.StringIO`）

如果要连接多个 `bytes` 对象，推荐做法是用 `bytearray` 对象的原地连接操作（`+=` 运算符）追加数据：

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

2.5 序列（元组/列表）

2.5.1 如何在元组和列表之间进行转换？

类型构造器 `tuple(seq)` 可将任意序列（实际上是任意可迭代对象）转换为数据项和顺序均不变的元组。

例如，`tuple([1, 2, 3])` 会生成 `(1, 2, 3)`，`tuple('abc')` 则会生成 `('a', 'b', 'c')`。如果参数就是元组，则不会创建副本而是返回同一对象，因此如果无法确定某个对象是否为元组时，直接调用 `tuple()` 也没什么代价。

类型构造器 `list(seq)` 可将任意序列或可迭代对象转换为数据项和顺序均不变的列表。例如，`list((1, 2, 3))` 会生成 `[1, 2, 3]` 而 `list('abc')` 则会生成 `['a', 'b', 'c']`。如果参数即为列表，则会像 `seq[:]` 那样创建一个副本。

2.5.2 什么是负数索引？

Python 序列的索引可以是正数或负数。索引为正数时，0 是第一个索引值，1 为第二个，依此类推。索引为负数时，-1 为倒数第一个索引值，-2 为倒数第二个，依此类推。可以认为 `seq[-n]` 就相当于 `seq[len(seq)-n]`。

使用负数序号有时会很方便。例如 `s[:-1]` 就是原字符串去掉最后一个字符，这可以用来移除某个字符串末尾的换行符。

2.5.3 序列如何以逆序遍历？

使用内置函数 `reversed()`：

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

原序列不会变化，而是构建一个逆序的新副本以供遍历。

2.5.4 如何从列表中删除重复项？

许多完成此操作的详细介绍，可参阅 Python Cookbook：

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

如果列表允许重新排序，不妨先对其排序，然后从列表末尾开始扫描，依次删除重复项：

```
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]
```

如果列表的所有元素都能用作集合的键（即都是 *hashable*），以下做法速度往往更快：

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

以上操作会将列表转换为集合，从而删除重复项，然后返回成列表。

2.5.5 如何从列表中删除多个项？

类似于删除重复项，一种做法是反向遍历并根据条件删除。不过更简单快速的做法就是切片替换操作，采用隐式或显式的正向迭代遍历。以下是三种变体写法：

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

列表推导式可能是最快的。

2.5.6 如何在 Python 中创建数组？

用列表：

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

列表在时间复杂度方面相当于 C 或 Pascal 的数组；主要区别在于，Python 列表可以包含多种不同类型的对象。

array 模块也提供了一些创建具有紧凑表示形式的固定类型数据的方法，但其索引速度要比列表慢。还可关注 NumPy 和其他一些第三方包也定义了一些各具特色的数组类结构体。

要获得 Lisp 风格的列表，可以使用元组来模拟 *cons* 单元：

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

如果需要可变特性，你可以用列表来代替元组。在这里模拟 Lisp *car* 的是 `lisp_list[0]` 而模拟 *cdr* 的是 `lisp_list[1]`。只有在你确定真有需要时才这样做，因为这通常会比使用 Python 列表要慢上许多。

2.5.7 如何创建多维列表？

多维数组或许会用以下方式建立：

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

打印出来貌似没错：

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

但如果给某一项赋值，结果会同时在多个位置体现出来：

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

原因在于用 `*` 对列表执行重复操作并不会创建副本，而只是创建现有对象的引用。`*3` 创建的是包含 3 个引用的列表，每个引用指向的是同一个长度为 2 的列表。1 处改动会体现在所有地方，这一定不是应有的方案。推荐做法是先创建一个所需长度的列表，然后将每个元素都填充为一个新建列表。

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

以上生成了一个包含 3 个列表的列表，每个子列表的长度为 2。也可以采用列表推导式：

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

或者，你也可以使用提供矩阵数据类型的扩展；其中最著名的是 NumPy。

2.5.8 我如何将一个方法或函数应用于由对象组成的序列？

要调用一个方法或函数并将返回值累积到一个列表中，*list comprehension* 是一种优雅的方案：

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

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

如果只需运行方法或函数而不保存返回值，那么一个简单的 `for` 循环就足够了：

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

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

2.5.9 为什么 `a_tuple[i] += ['item']` 会引发异常？

这是由两个因素共同导致的，一是增强赋值运算符属于赋值运算符，二是 Python 可变和不可变对象之间的差别。

只要元组的元素指向可变对象，这时对元素进行增强赋值，那么这里介绍的内容都是适用的。在此只以 `list` 和 `+=` 举例。

如果你写成这样：

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

触发异常的原因显而易见：1 会与指向 (1) 的对象 `a_tuple[0]` 相加，生成结果对象 2，但在试图将运算结果 2 赋值给元组的 0 号元素时就会报错，因为元组元素的指向无法更改。

其实在幕后，上述增强赋值语句的执行过程大致如下：

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

由于元组是不可变的，因此赋值这步会引发错误。

如果写成以下这样：

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

这时触发异常会令人略感惊讶，更让人吃惊的是虽有报错，但加法操作却生效了：

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

要明白为什么会这样，你需要知道 (a) 如果一个对象实现了 `__iadd__()` 魔术方法，那么它就会在执行 `+=` 增强赋值时被调用，并且其返回值将在赋值语句中被使用；(b) 对于列表而言，`__iadd__()` 等价于在列表上调用 `extend()` 并返回该列表。所以对于列表我们可以这样说，`+=` 就是 `list.extend()` 的“快捷方式”：

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

這等價於：

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

`a_list` 所引用的对象已被修改，而引用被修改对象的指针又重新被赋值给 `a_list`。赋值的最终结果没有变化，因为它是引用 `a_list` 之前所引用的同一对象的指针，但仍然发生了赋值操作。

因此，在此元组示例中，发生的事情等同于：

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

`__iadd__()` 执行成功，因此列表得到了扩充，但是即使 `result` 是指向 `a_tuple[0]` 所指向的同一个对象，最后的赋值仍然会导致错误，因为元组是不可变的。

2.5.10 我想做一个复杂的排序：能用 Python 进行施瓦茨变换吗？

归功于 Perl 社区的 Randal Schwartz，该技术根据度量值对列表进行排序，该度量值将每个元素映射为“顺序值”。在 Python 中，请利用 `list.sort()` 方法的 `key` 参数：

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

2.5.11 如何根据另一个列表的值对某列表进行排序？

将它们合并到元组的迭代器中，对结果列表进行排序，然后选择所需的元素。

```
>>> 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 什麼是類 (class) ?

类是通过执行 `class` 语句创建的某种对象的类型。创建实例对象时，用 `Class` 对象作为模板，实例对象既包含了数据（属性），又包含了数据类型特有的代码（方法）。

类可以基于一个或多个其他类（称之为基类）进行创建。基类的属性和方法都得以继承。这样对象模型就可以通过继承不断地进行细化。比如通用的 `Mailbox` 类提供了邮箱的基本访问方法，它的子类 `MboxMailbox`、`MaildirMailbox`、`OutlookMailbox` 则能够处理各种特定的邮箱格式。

2.6.2 什麼是方法 (method) ?

方法是属于对象的函数，对于对象 `x`，通常以 `x.name(arguments...)` 的形式调用。方法以函数的形式给出定义，位于类的定义内：

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

2.6.3 什么是 self ?

`Self` 只是方法的第一个参数的习惯性名称。假定某个类中有个方法定义为 `meth(self, a, b, c)`，则其实例 `x` 应以 `x.meth(a, b, c)` 的形式进行调用；而被调用的方法则应视其为做了 `meth(x, a, b, c)` 形式的调用。

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

2.6.4 如何检查对象是否为给定类或其子类的一个实例？

使用内置函数 `isinstance(obj, cls)`。你可以检测对象是否属于多个类中的某一个的实例，只要提供一个元组而非单个类即可，如 `isinstance(obj, (class1, class2, ...))`，还可以检测对象是否属于 Python 的某个内置类型，如 `isinstance(obj, str)` 或 `isinstance(obj, (int, float, complex))`。

请注意 `isinstance()` 还会检测派生自 *abstract base class* 的虚继承。因此对于已注册的类，即便没有直接或间接继承自抽象基类，对抽象基类的检测都将返回 `True`。要想检测“真正的继承”，请扫描类的 *MRO*：

```
from collections.abc import Mapping

class P:
    pass

class C(P):
    pass

Mapping.register(P)
```

```
>>> c = C()
>>> isinstance(c, C)      # direct
True
```

(繼續下一頁)

(繼續上一頁)

```

>>> isinstance(c, P)          # indirect
True
>>> isinstance(c, Mapping)    # virtual
True

# Actual inheritance chain
>>> type(c).__mro__
(<class 'C'>, <class 'P'>, <class 'object'>)

# Test for "true inheritance"
>>> Mapping in type(c).__mro__
False

```

请注意，大多数程序不会经常用 `isinstance()` 对用户自定义类进行检测。如果是自己开发的类，更合适的面向对象编程风格应该是在类中定义多种方法，以封装特定的行为，而不是检查对象属于什么类再据此干不同的事。假定有如下执行某些操作的函数：

```

def search(obj):
    if isinstance(obj, Mailbox):
        ... # code to search a mailbox
    elif isinstance(obj, Document):
        ... # code to search a document
    elif ...

```

更好的方法是在所有类上定义一个 `search()` 方法，然后调用它：

```

class Mailbox:
    def search(self):
        ... # code to search a mailbox

class Document:
    def search(self):
        ... # code to search a document

obj.search()

```

2.6.5 什么是委托？

委托是一种面向对象的技术（也称为设计模式）。假设对象 `x` 已经存在，现在想要改变其某个方法的行为。可以创建一个新类，其中提供了所需修改方法的新实现，而将所有其他方法都委托给 `x` 的对应方法。

Python 程序员可以轻松实现委托。比如以下实现了一个类似于文件的类，只是会把所有写入的数据转换为大写：

```

class UpperOut:

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

    def write(self, s):
        self._outfile.write(s.upper())

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

```

这里 UpperOut 类重新定义了 write() 方法，在调用下层的 self._outfile.write() 方法之前将参数字符串转换为大写形式。所有其他方法都被委托给下层的 self._outfile 对象。委托是通过 __getattr__() 方法完成的；请参阅 语言参考 了解有关控制属性访问的更多信息。

请注意在更一般的情况下委托可能会变得比较棘手。当属性即需要被设置又需要被提取时，类还必须定义 __setattr__() 方法，而这样做必须十分小心。__setattr__() 的基本实现大致如下所示：

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

大多数 __setattr__() 实现都必须修改 self.__dict__ 来为自身保存局部状态而不至于造成无限递归。

2.6.6 如何在扩展基类的派生类中调用基类中定义的方法？

使用内置的 super() 函数：

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

在下面的例子中，super() 将自动根据它的调用方(self 值)来确定实例对象，使用 type(self).__mro__ 查找 *method resolution order* (MRO)，并返回 MRO 中位于 Derived 之后的项：Base。

2.6.7 如何让代码更容易对基类进行修改？

可以为基类赋一个别名并基于该别名进行派生。这样只要修改赋给该别名的值即可。顺便提一下，如要动态地确定（例如根据可用的资源）该使用哪个基类，这个技巧也非常方便。例如：

```
class Base:
    ...

BaseAlias = Base

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

2.6.8 如何创建静态类数据和静态类方法？

Python 支持静态数据和静态方法（以 C++ 或 Java 的定义而言）。

静态数据只需定义一个类属性即可。若要为属性赋新值，则必须在赋值时显式使用类名：

```
class C:
    count = 0  # number of times C.__init__ called

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

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


对于所有符合 `isinstance(c, C)` 的 `c`, `c.count` 也同样指向 `C.count`, 除非被 `c` 自身或者被从 `c.__class__` 回溯到基类 `C` 的搜索路径上的某个类所覆盖。

注意: 在 `C` 的某个方法内部, 像 `self.count = 42` 这样的赋值将在 `self` 自身的字典中新建一个名为“count”的不相关实例。想要重新绑定类静态数据名称就必须总是指明类名, 无论是在方法内部还是外部:

```
C.count = 314
```

Python 支持静态方法:

```
class C:
    @staticmethod
    def static(arg1, arg2, arg3):
        # No 'self' parameter!
        ...
```

不过为了获得静态方法的效果, 还有一种做法直接得多, 也即使用模块级函数即可:

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

如果代码的结构化比较充分, 每个模块只定义了一个类 (或者多个类的层次关系密切相关), 那就具备了应有的封装。

2.6.9 在 Python 中如何重载构造函数 (或方法)?

这个答案实际上适用于所有方法, 但问题通常首先出现于构造函数的应用场景中。

在 C++ 中, 代码会如下所示:

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

在 Python 中, 只能编写一个构造函数, 并用默认参数捕获所有情况。例如:

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

这不完全等同, 但在实践中足够接近。

也可以试试采用变长参数列表, 例如:

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

上述做法同样适用于所有方法定义。

2.6.10 在用 `__spam` 的时候得到一个类似 `_SomeClassName__spam` 的错误信息。

以双下划线打头的变量名会被“破坏”，以便以一种简单高效的方式定义类私有变量。任何形式为 `__spam` 的标识符（至少前缀两个下划线，至多后缀一个下划线）文本均会被替换为 `_classname__spam`，其中 `classname` 为去除了全部前缀下划线的当前类名称。

这并不能保证私密性：外部用户仍然可以访问“`_classname__spam`”属性，私有变量值也在对象的 `__dict__` 中可见。许多 Python 程序员根本不操心要去使用私有变量名。

2.6.11 类定义了 `__del__` 方法，但是删除对象时没有调用它。

这有几个可能的原因。

`del` 语句不一定要调用 `__del__()` -- 它只是减少对象的引用计数，如果计数达到零才会调用 `__del__()`。

如果你的数据结构包含循环链接（如树每个子节点都带有父节点的引用，而每个父节点也带有子节点的列表），引用计数永远不会回零。尽管 Python 偶尔会用某种算法检测这种循环引用，但在数据结构的最后一条引用消失之后，垃圾收集器可能还要过段时间才会运行，因此 `__del__()` 方法可能会在不方便或随机的时刻被调用。这对于重现一个问题是非常不方便的。更糟糕的是，各个对象的 `__del__()` 方法是以随机顺序执行的。虽然你可以运行 `gc.collect()` 来强制执行垃圾回收操作，但仍会存在一些对象永远不会被回收的失控情况。

尽管有垃圾回收器，但当对象使用完毕时要在要调用的对象上定义显式的 `close()` 方法仍然是个好主意。`close()` 方法可以随后移除引用子对象的属性。请不要直接调用 `__del__()` -- `__del__()` 应当调用 `close()` 并且 `close()` 应当确保被可以同一对象多次调用。

另一种避免循环引用的做法是利用 `weakref` 模块，该模块允许指向对象但不增加其引用计数。例如，树状数据结构应该对父节点和同级节点使用弱引用（如果真要用的话！）

最后，如果你的 `__del__()` 方法引发了异常，会将警告消息打印到 `sys.stderr`。

2.6.12 如何获取给定类的所有实例的列表？

Python 不会记录类（或内置类型）的实例。可以在类的构造函数中编写代码，通过保留每个实例的弱引用列表来跟踪所有实例。

2.6.13 为什么 `id()` 的结果看起来不是唯一的？

`id()` 返回一个整数，该整数在对象的生命周期内保证是唯一的。因为在 CPython 中，这是对象的内存地址，所以经常发生在从内存中删除对象之后，下一个新创建的对象被分配在内存中的相同位置。这个例子说明了这一点：

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

这两个 `id` 属于不同的整数对象，之前先创建了对对象，执行 `id()` 调用后又立即被删除了。若要确保检测 `id` 时的对象仍处于活动状态，请再创建一个对该对象的引用：

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

2.6.14 什么情况下可以依靠 `is` 运算符进行对象的身份相等性测试？

`is` 运算符可用于测试对象的身份相等性。`a is b` 等价于 `id(a) == id(b)`。

身份相等性最重要的特性就是对象总是等同于自身，`a is a` 一定返回 `True`。身份相等性测试的速度通常比相等性测试要快。而且与相等性测试不一样，身份相等性测试会确保返回布尔值 `True` 或 `False`。

但是，身份相等性测试只能在对象身份确定的场景下才可替代相等性测试。一般来说，有以下 3 种情况对象身份是可以确定的：

- 1) 赋值操作创建了新的名称但没有改变对象身份。在赋值操作 `new = old` 之后，可以保证 `new is old`。
- 2) 将对象置入存放对象引用的容器，对象身份不会改变。在列表赋值操作 `s[0] = x` 之后，可以保证 `s[0] is x`。
- 3) 单例对象，也即该对象只能存在一个实例。在赋值操作 `a = None` 和 `b = None` 之后，可以保证 `a is b`，因为 `None` 是单例对象。

其他大多数情况下，都不建议使用身份相等性测试，而应采用相等性测试。尤其是不应将身份相等性测试用于检测常量值，例如 `int` 和 `str`，因为他们并不一定是单例对象：

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

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

同样地，可变容器的新实例，对象身份一定不同：

```
>>> a = []
>>> b = []
>>> a is b
False
```

在标准库代码中，给出了一些正确使用对象身份测试的常见模式：

- 1) 正如 **PEP 8** 所推荐的，对象身份测试是 `None` 值的推荐检测方式。这样的代码读起来就像自然的英文，并可以避免与其他可能为布尔值且计算结果为 `False` 的对象相混淆。
- 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) 编写容器的实现代码时, 有时需要用对象身份测试来加强相等性检测。这样代码就不会被 `float('NaN')` 这类与自身不相等的对象所干扰。

例如, 以下是 `collections.abc.Sequence.__contains__()` 的实现代码:

```
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？

当子类化一个不可变类型时, 请重写 `__new__()` 方法而不是 `__init__()` 方法。后者只在一个实例被创建之后运行, 这对于改变不可变实例中的数据来说太晚了。

所有这些不可变的类都有一个与它们的父类不同的签名:

```
from datetime import date

class FirstOfMonthDate(date):
    "Always choose the first day of the month"
    def __new__(cls, year, month, day):
        return super().__new__(cls, year, month, 1)

class NamedInt(int):
    "Allow text names for some numbers"
    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):
    "Convert str to name suitable for a URL path"
    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 我该如何缓存方法调用？

缓存方法的两个主要工具是 `functools.cached_property()` 和 `functools.lru_cache()`。前者在实例层级上存储结果而后者在类层级上存储结果。

`cached_property` 方式仅适用于不接受任何参数的方法。它不会创建对实例的引用。被缓存的方法结果将仅在实例的生存期内被保留。

其优点是，当一个实例不再被使用时，缓存的方法结果将被立即释放。缺点是，如果实例累积起来，累积的方法结果也会增加。它们可以无限制地增长。

`lru_cache` 方法适用于具有可雜引數的方法。除非特努力傳遞弱引用，否則它會建立對實例的引用。

最少近期使用算法的优点是缓存会受指定的 `maxsize` 限制。它的缺点是实例会保持存活，直到其达到生存期或者缓存被清空。

这个例子演示了几种不同的方式：

```
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.
```

上面的例子假定 `station_id` 从不改变。如果相关实例属性是可变对象，则 `cached_property` 方式就不再适用，因为它无法检测到属性的改变。

要让 `lru_cache` 方式在 `station_id` 可变时仍然适用，类需要定义 `__eq__()` 和 `__hash__()` 方法以便缓存能检测到相关属性的更新：

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

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

    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):
```

(繼續下一頁)

(繼續上一頁)

```

    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 如何创建.pyc 文件？

当首次导入模块时（或当前已编译文件创建之后源文件发生了改动），在 .py 文件所在目录的 `__pycache__` 子目录下会创建一个包含已编译代码的 .pyc 文件。该 .pyc 文件的名称开头部分将与 .py 文件名相同，并以 .pyc 为后缀，中间部分则依据创建它的 python 版本而各不相同。（详见 [PEP 3147](#)。）

.pyc 文件有可能会无法创建，原因之一是源码文件所在的目录存在权限问题，这样就无法创建 `__pycache__` 子目录。假如以某个用户开发程序而以另一用户运行程序，就有可能发生权限问题，测试 Web 服务器就属于这种情况。

除非设置了 `PYTHONDONTWRITEBYTECODE` 环境变量，否则导入模块并且 Python 能够创建 `__pycache__` 子目录并把已编译模块写入该子目录（权限、存储空间等等）时，.pyc 文件就将自动创建。

在最高层级运行的 Python 脚本不会被视为经过了导入操作，因此不会创建 .pyc 文件。假定有一个最高层级的模块文件 `foo.py`，它导入了另一个模块 `xyz.py`，当运行 `foo` 模块（通过输入 shell 命令 `python foo.py`），则会为 `xyz` 创建一个 .pyc，因为 `xyz` 是被导入的，但不会为 `foo` 创建 .pyc 文件，因为 `foo.py` 不是被导入的。

若要为 `foo` 创建 .pyc 文件——即为未做导入的模块创建 .pyc 文件——可以利用 `py_compile` 和 `compileall` 模块。

`py_compile` 模块能够手动编译任意模块。一种做法是交互式地使用该模块中的 `compile()` 函数：

```

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

```

这将会将 .pyc 文件写入与 `foo.py` 相同位置下的 `__pycache__` 子目录（或者你也可以通过可选参数 `cfile` 来重写该行为）。

还可以用 `compileall` 模块自动编译一个或多个目录下的所有文件。只要在命令行提示符中运行 `compileall.py` 并给出要编译的 Python 文件所在目录路径即可：

```
python -m compileall .
```

2.7.2 如何找到当前模块名称？

模块可以查看预定义的全局变量 `__name__` 获悉自己的名称。如其值为 `'__main__'`，程序将作为脚本运行。通常，许多通过导入使用的模块同时也提供命令行接口或自检代码，这些代码只在检测到处于 `__name__` 之后才会执行：

```

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

```

(繼續下一頁)

(繼續上一頁)

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

2.7.3 如何让模块相互导入？

假设有以下模块：

foo.py:

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

bar.py:

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

问题是解释器将执行以下步骤：

- 首先导入 foo
- 为 foo 创建空的全局变量
- 编译 foo 并开始执行
- foo 导入 bar
- 为 bar 创建空的全局变量
- bar 已被編譯開始執行
- bar 导入 foo (该步骤无操作，因为已经有一个名为 foo 的模块)。
- 导入机制尝试从 foo_var 全局变量读取 foo，用来设置 bar.foo_var = foo.foo_var

最后一步失败了，因为 Python 还没有完成对 foo 的解释，foo 的全局符号字典仍然是空的。

当你使用 `import foo`，然后尝试在全局代码中访问 `foo.foo_var` 时，会发生同样的事情。

此問題有（至少）三種可能的解方法。

Guido van Rossum 建议完全避免使用 `from <module> import ...`，并将所有代码放在函数中。全局变量和类变量的初始化只应使用常量或内置函数。这意味着导入模块中的所有内容都以 `<module>.<name>` 的形式引用。

Jim Roskind 建議在每個模組中按以下順序執行各個步驟：

- 导出（全局变量、函数和不需要导入基类的类）
- `import` 陳述式
- 本模块的功能代码（包括根据导入值进行初始化的全局变量）。

Van Rossum 不太喜欢这种方法，因为 `import` 出现在一个奇怪的地方，但它确实有效。

Matthias Urlichs 建议对代码进行重构，使得递归导入根本就没必要发生。

這些方案不相互排斥。

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

不妨考虑换用 `importlib` 中的函数 `import_module()`：

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

2.7.5 对已导入的模块进行了编辑并重新导入，但变动没有得以体现。这是为什么？

出于效率和一致性的原因，Python 仅在第一次导入模块时读取模块文件。否则，在一个多模块的程序中，每个模块都会导入相同的基础模块，那么基础模块将会被一而再、再而三地解析。如果要强行重新读取已更改的模块，请执行以下操作：

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

警告：这种技术并非万无一失。尤其是模块包含了以下语句时：

```
from modname import some_objects
```

仍将继续使用前一版的导入对象。如果模块包含了类的定义，并不会用新的类定义更新现有的类实例。这样可能会导致以下矛盾的行为：

```
>>> import importlib
>>> import cls
>>> c = cls.C()                                # Create an instance of C
>>> importlib.reload(cls)
<module 'cls' from 'cls.py'>
>>> isinstance(c, cls.C)                       # isinstance is false!?!
False
```

只要把类对象的 `id` 打印出来，问题的性质就会一目了然：

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

設計和歷史常見問答集

3.1 為什麼 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，而且對提高一般 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。

因為有開始/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++;
```

如果條件為真，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，思考為何即便 `x > y`，但 `y` 還是變少了。

因為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為困難。理想上來講，一個函式應該要佔一個螢幕（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並非單一原因（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 為什麼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 為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
```

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

CPython 的 float 型別使用了 C 的 double 型別來儲存。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，而它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說，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，但無法簡單地變成二進制表示。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。以一般的電腦來說，他實際儲存的值是：

```
1.0011001100110011001100110011001100110011001100110011001100110011 (binary)
```

而這個值正是：

```
1.1999999999999999555910790149937383830547332763671875 (decimal)
```

53 位元的精度讓 Python 可以有 15 至 16 小數位的準確度。

要更完全的解釋可以查在 Python 教學的浮點運算一章。

3.4 為什麼 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 中也沒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++ 中，你可以藉由有區域變數宣告來判斷這件事——但在 Python 中，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++ 中，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 中，你可以直接寫成 `baseclass.methodname(self, <argument list>)`。這在 `__init__()` 方法很好用，特別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，他解決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（效率）。C++ 用宣告解決了這件事，但 Python 有，而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決。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內容。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，你可以這麼做了！

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準 `len()` 的類。遇到一個有實作映射 (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 為何 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準的型別，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，那麼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，你並不是在這樣做。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)
```

單就這個情況來，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 為什麼 Python 沒有 switch 或 case 陳述式？

总的来说，结构化分支语句会在一个表达式具有特定值或值的集合时执行某个代码块。从 Python 3.10 开始可以简单地通过 `match ... case` 语句来匹配字面值，或特定命名空间中的常量。一种较旧的替代方案是通过一系列的 `if... elif... elif... else`。

如果可能性很多，你可以用字典去映射要呼叫的函式。舉例來說：

```
functions = {'a': function_1,
             'b': function_2,
             'c': self.method_1}

func = functions[value]
func()
```

對於呼叫物件的方法，你可以利用 `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，以這個例子來說是像是 `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。

模仿帶有穿透方式的分支，就像 C 的 `switch-case-default` 那樣是有可能的，但更為困難，也無甚必要。

3.11 為什麼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 為什麼 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 是一個有力量編譯器，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決於實作。Python 的標準實作 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 程式碼的表現取決於參照計次的實作，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 的標準功能，因此他的可移植性低。（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 程式當然可以把標準的 malloc() 和 free() 換成 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。以現在來說，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 的紀 (record) 或是 C 的結構 (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)。他可以有不同個同類物件，且逐項操作。舉例來，`os.listdir('.')` 回傳當下目錄的檔案，以包含字串的串列表示。如果你新增了幾個檔案到這個目錄，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。

Dictionaries work by computing a hash code for each key stored in the dictionary using the `hash()` built-in function. The hash code varies widely depending on the key and a per-process seed; for example, "Python" could hash to -539294296 while "python", a string that differs by a single bit, could hash to 1142331976. The hash code is then used to calculate a location in an internal array where the value will be stored. Assuming that you're storing keys that all have different hash values, this means that dictionaries take constant time -- $O(1)$, in Big-O notation -- to retrieve a key.

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，則雜表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內容。然後，當你嘗試在字典中尋找這個物件時，因雜表值不同的緣故，你找不到他。而如果你嘗試用舊的值去尋找，也一樣找不到，因他的雜表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 一樣內容的元組。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）來雜表。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（或其他用雜表基底的結構）中他們的雜表值會保持定。

```
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，雜表的計算可能變得雜，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，那字典和其他用雜表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 為何 `list.sort()` 不是回傳排序過的串列？

在重視效能的情況下，把串列複製一份有些浪費。因此，`list.sort()` 直接在串列上做排序。為了提醒你這件事，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，那可以改用建立的 `sorted()`。他會用提供的可迭代物件 (iterable) 來排序建立新串列，並回傳之。例如，以下這個範例會說明如何有序地迭代字典的鍵：

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

3.22 如何在 Python 中指定和限制使用一個介面規範 (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。很多人認為這種在編譯時限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 來說，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)，作為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」。即便模組使用了複雜的外部介面，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。

适当的測試規程能像完善的接口规范一样帮助在 Python 构建大型的复杂应用程序。事实上，它能做得更好因为接口规范无法测试程序的某些属性。例如，`list.append()` 方法被期望向某个内部列表的末尾添加新元素；接口规范无法测试你的 `list.append()` 实现是否真的能正确执行该操作，但在测试套件中检查该属性却是很容易的。

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 為何有 `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)。在高階語言中，這也是不需要的，因為有方法可以做邏輯分支（以 Python 來說，用 `if` 陳述式和 `or`、`and` 及 `if-else` 運算式）和迴圈（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
    ...
```

(繼續下一頁)

(繼續上一頁)

```
except label: # where to goto
    pass
...
```

這依然不能讓你跳進圈，這通常被認為是對 goto 的濫用。請小心使用。

3.24 何純字串 (r-string) 不能以反斜結尾？

更精確地來說，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。

設計出純字串是為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為例：

```
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 有任何對象告訴直譯器這件事。在假設「a」是一個整數的話，那會發生什麼事？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。

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？

需要冒號主要是為了增加可讀性（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 答案的例子中的，這是標準的英文用法。

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成。

函式庫和擴充功能的常見問題

4.1 常見函式問題

4.1.1 如何找到可以用来做 X 任务的模块或应用？

在标准库参考中查找是否有适合的标准库模块。（如果你已经了解标准库的内容，可以跳过这一步）

对于第三方软件包，请搜索 [Python Package Index](#) 或者是尝试 [Google](#) 或其他网络搜索引擎。搜索“Python”加上一两个你感兴趣的关键词通常就会找到一些有用的信息。

4.1.2 哪裏可以找到 `math.py` (`socket.py`, `regex.py`, 等...) 來源檔案？

如果找不到模块的源文件，可能它是一个内建的模块，或是使用 C，C++ 或其他编译型语言实现的动态加载模块。这种情况下可能是没有源码文件的，类似 `mathmodule.c` 这样的文件会存放在 C 代码目录中（但在 Python 目录中）。

有（至少）三種 Python 模組：

- 1) 以 Python 編寫的模組 (`.py`)；
- 2) 用 C 編寫並動態載入的模組 (`.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 ?

你需要做兩件事：文件必須是可執行的，並且第一行需要以 `#!` 開頭，後面跟上 Python 解釋器的路徑。

第一點可以用執行 `chmod +x scriptfile` 或是 `chmod 755 scriptfile` 做到。

第二點有很多種做法，最直接的方式是：

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

在文件第一行，使用你所在平台上的 Python 解釋器的路徑。

如果你希望腳本不依賴 Python 解釋器的具體路徑，你也可以使用 `env` 程序。假設你的 Python 解釋器所在目錄已經添加到了 `PATH` 環境變量中，几乎所有的类 Unix 系統都支持下面的寫法：

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

不要在 CGI 腳本中這樣做。CGI 腳本的 `PATH` 環境變量通常會非常精簡，所以你必须使用解釋器的完整絕對路徑。

有時候，用戶的環境變量如果太長，可能會導致 `/usr/bin/env` 執行失敗；又或者甚至根本就不存在 `env` 程序。在這種情況下，你可以嘗試使用下面的 hack 方法（來自 Alex Rezinsky）：

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

這樣做有一個小小的缺點，它會定義腳本的 `__doc__` 字符串。不過可以這樣修復：

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

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

對於類 Unix 系統：標準 Python 源碼發行版會在 `Modules` 子目錄中附帶 `curses` 模塊，但默認並不會編譯。（注意：在 Windows 平台下不可用——Windows 中沒有 `curses` 模塊。）

`curses` 模塊支持基本的 `curses` 特性，同時也支持 `ncurses` 和 `SYSV curses` 中的很多額外功能，比如顏色、不同的字符集支持、填充和鼠標支持。這意味著這個模塊不兼容只有 BSD `curses` 模塊的操作系统，但是目前仍在維護的系統應該都不會存在這種情況。

4.1.5 Python 中存在類似 C 的 `onexit()` 函數的東西嗎？

`atexit` 模塊提供了一個與 C 的 `onexit()` 類似的注冊函數。

4.1.6 为什么我的信号处理函数不能工作？

最常见的问题是信号处理函数没有正确定义参数列表。它会被这样调用：

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

因此它应当声明为带有两个形参：

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

4.2 常見課題

4.2.1 如何測試 Python 程式或元件？

Python 带有两个测试框架。doctest 模块从模块的 docstring 中寻找示例并执行，对比输出是否与 docstring 中给出的是否一致。

unittest 模块是一个模仿 Java 和 Smalltalk 测试框架的更棒的测试框架。

为了使测试更容易，你应该在程序中使用良好的模块化设计。程序中的绝大多数功能都应该用函数或类方法封装——有时这样做会有额外惊喜，程序会运行得更快（因为局部变量比全局变量访问要快）。除此之外，程序应该避免依赖可变的局部变量，这会使得测试困难许多。

程序的“全局主逻辑”应该尽量简单：

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

并放置在程序主模块的最后面。

一旦你的程序已经组织为一个函数和类行为的有完整集合，你就应该编写测试函数来检测这些行为。可以将自动执行一系列测试的测试集关联到每个模块。这听起来似乎需要大量的工作，但是由于 Python 是如此简洁灵活因此它会极其容易。你可以通过与“生产代码”同步编写测试函数使编程更为愉快和有趣，因为这将更容易并更早发现代码问题甚至设计缺陷。

程序主模块之外的其他“辅助模块”中可以增加自测试的入口。

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

通过使用 Python 实现的“假”接口，即使是需要与复杂的外部接口交互的程序也可以在外部接口不可用时进行测试。

4.2.2 怎样用 docstring 创建文档？

pydoc 模块可以用你的 Python 源代码中的文档字符串来创建 HTML。纯粹通过文档字符串来创建 API 文档的一种替代方案是 [epydoc](#)。Sphinx 也可以包括文档字符串的内容。

4.2.3 怎样一次只获取一个按键？

在类 Unix 系统中有多种方案。最直接的方法是使用 `curses`，但是 `curses` 模块太大了，难以学习。

4.3 執行緒

4.3.1 如何使用執行緒編寫程式？

一定要使用 `threading` 模块，不要使用 `_thread` 模块。`threading` 模块对 `_thread` 模块提供的底层线程原语做了更易用的抽象。

4.3.2 我的執行緒似乎都「有運行」：「什」？

一旦主线程退出，所有的子线程都会被杀掉。你的主线程运行得太快了，子线程还没来得及工作。

简单的解决方法是在程序中加一个时间足够长的 `sleep`，让子线程能够完成运行。

```
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) # <-----! 
```

但目前（在许多平台上）线程不是并行运行的，而是按顺序依次执行！原因是系统线程调度器在前一个线程阻塞之前不会启动新线程。

简单的解决方法是在运行函数的开始处加一个时间很短的 `sleep`。

```
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)
```

比起用 `time.sleep()` 猜一个合适的等待时间，使用信号量机制会更好些。有一个办法是使用 `queue` 模块创建一个 `queue` 对象，让每一个线程在运行结束时 `append` 一个令牌到 `queue` 对象中，主线程中从 `queue` 对象中读取与线程数量一致的令牌数量即可。

4.3.3 如何将任务分配给多个工作线程？

最简单的方式是使用 `concurrent.futures` 模块，特别是其中的 `ThreadPoolExecutor` 类。

或者，如果你想更好地控制分发算法，你也可以自己写逻辑实现。使用 `queue` 模块来创建任务列表队列。`Queue` 类维护了一个存有对象的列表，提供了 `.put(obj)` 方法添加元素，并且可以用 `.get()` 方法获取元素。这个类会使用必要的加锁操作，以此确保每个任务只会执行一次。

这是一个简单的例子：

```
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)
```

运行时会产生如下输出：

```
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
```

(繼續下一頁)

(繼續上一頁)

```
Worker <Thread(worker 5, started 130283799226640)> running with argument 4
Worker <Thread(worker 1, started 130283832797456)> running with argument 5
...
```

查看模块的文档以获取更多信息；Queue 类提供了多种接口。

4.3.4 怎样修改全局变量是线程安全的？

Python VM 内部会使用 *global interpreter lock* (GIL) 来确保同一时间只有一个线程运行。通常 Python 只会 在字节码指令之间切换线程；切换的频率可以通过设置 `sys.setswitchinterval()` 指定。从 Python 程序的角度来看，每一条字节码指令以及每一条指令对应的 C 代码实现都是原子的。

理论上说，具体的结果要看具体的 PVM 字节码实现对指令的解释。而实际上，对内建类型 (`int`, `list`, `dict` 等) 的共享变量的“类原子”操作都是原子的。

举例来说，下面的操作是原子的 (`L`、`L1`、`L2` 是列表，`D`、`D1`、`D2` 是字典，`x`、`y` 是对象，`i`、`j` 是 `int` 变量)：

```
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()
```

这些不是原子的：

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

替换其他对象的操作可能会在其他对象的引用计数变为零时唤起这些对象的 `__del__()` 方法，这可能会产生一些影响。对字典和列表进行大量更新尤其如此。如有疑问，请使用互斥锁！

4.3.5 不能擺 全局直譯器鎖嗎？

global interpreter lock (GIL) 通常被视为 Python 在高端多核服务器上开发时的阻力，因为（几乎）所有 Python 代码只有在获取到 GIL 时才能运行，所以多线程的 Python 程序只能有效地使用一个 CPU。

在 Python 1.5 时代，Greg Stein 开发了一个完整的补丁包（“free threadings”补丁），移除了 GIL，并用粒度更合适的锁来代替。Adam Olsen 最近也在他的 `python-safethread` 项目里做了类似的实验。不幸的是，由于为了移除 GIL 而使用了大量细粒度的锁，这两个实验在单线程测试中的性能都有明显的下降（至少慢 30%）。

但这并不意味着你不能在多核机器上很好地使用 Python！你只需将任务划分为多 * 进程 *，而不是多 * 线程 *。新的 `concurrent.futures` 模块中的 `ProcessPoolExecutor` 类提供了一个简单的方法；如果你想对任务分发做更多控制，可以使用 `multiprocessing` 模块提供的底层 API。

恰当地使用 C 拓展也很有用；使用 C 拓展处理耗时较久的任务时，拓展可以在线程执行 C 代码时释放 GIL，让其他线程执行。`zlib` 和 `hashlib` 等标准库模块已经这样做了。

也有建议说 GIL 应该是解释器状态锁，而不是完全的全局锁；解释器不应该共享对象。不幸的是，这也不可能发生。由于目前许多对象的实现都有全局的状态，因此这是一个艰巨的工作。举例来说，小整型数和短字符串会缓存起来，这些缓存将不得不移动到解释器状态中。其他对象类型都有自己的自由变量列表，这些自由变量列表也必须移动到解释器状态中。等等。

我甚至怀疑这些工作是否可能在有限的时间内完成，因为同样的问题在第三方拓展中也会存在。第三方拓展编写的速度可比你将它们转换为把全局状态存入解释器状态中的速度快得多。

最后，假设多个解释器不共享任何状态，那么这样做比每个进程一个解释器好在哪里呢？

4.4 輸入與輸出

4.4.1 如何删除档案？(以及其他档案问题...)

使用 `os.remove(filename)` 或 `os.unlink(filename)`。查看 `os` 模块以获取更多文档。这两个函数是一样的，`unlink()` 是这个函数在 Unix 系统调用中的名字。

如果要删除目录，应该使用 `os.rmdir()`；使用 `os.mkdir()` 创建目录。`os.makedirs(path)` 会创建 `path` 中任何不存在的目录。`os.removedirs(path)` 则会删除其中的目录，只要它们都是空的；如果你想删除整个目录以及其中的内容，可以使用 `shutil.rmtree()`。

要重新命名档案，请使用 `os.rename(old_path, new_path)`。

如果需要截断文件，使用 `f = open(filename, "rb+")` 打开文件，然后使用 `f.truncate(offset)`；`offset` 默认是当前的搜索位置。也可以对使用 `os.open()` 打开的文件使用 `os.ftruncate(fd, offset)`，其中 `fd` 是文件描述符（一个小的整型数）。

`shutil` 模块也包含了一些处理文件的函数，包括 `copyfile()`，`copytree()` 和 `rmtree()`。

4.4.2 如何复制档案？

`shutil` 模块包含一个 `copyfile()` 函数。注意，在 Windows NTFS 卷上，它不复制替代数据流，也不复制 macOS HFS+ 卷上的资源分叉，尽管这两者现在很少使用。它也不复制文件权限和元数据，尽管使用 `shutil.copy2()` 可以保留大部分（但不是全部）的内容。

4.4.3 如何读取（或写入）二进制制资料？

要读写复杂的二进制数据格式，最好使用 `struct` 模块。该模块可以读取包含二进制数据（通常是数字）的字符串并转换为 Python 对象，反之亦然。

举例来说，下面的代码会从文件中以大端序格式读取一个 2 字节的整型和一个 4 字节的整型：

```
import struct

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

格式字符串中的 ‘>’ 强制以大端序读取数据；字母 ‘h’ 从字符串中读取一个“短整型”（2 字节），字母 ‘I’ 读取一个“长整型”（4 字节）。

对于更常规的数据（例如整型或浮点类型的列表），你也可以使用 `array` 模块。

備註：要读写二进制数据的话，需要强制以二进制模式打开文件（这里为 `open()` 函数传入 `"rb"`）。如果（默认）传入 `"r"` 的话，文件会以文本模式打开，`f.read()` 会返回 `str` 对象，而不是 `bytes` 对象。

4.4.4 似乎 `os.popen()` 创建的管道不能使用 `os.read()`，这是为什么？

`os.read()` 是一个底层函数，它接收的是文件描述符——用小整型数表示的打开的文件。`os.popen()` 创建的是一个高级文件对象，和内建的 `open()` 方法返回的类型一样。因此，如果要从 `os.popen()` 创建的管道 `p` 中读取 `n` 个字节的话，你应该使用 `p.read(n)`。

4.4.5 如何存取序列 (RS232) 連接埠？

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

<https://pypi.org/project/pyserial/>

對於 Unix，請參閱 Mitch Chapman 的 Usenet 貼文：

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

4.4.6 为什么关闭 `sys.stdout` (`stdin`, `stderr`) 并不会真正关掉它？

Python 文件对象 是一个对底层 C 文件描述符的高层抽象。

对于在 Python 中通过内建的 `open()` 函数创建的多数文件对象来说，`f.close()` 从 Python 的角度将其标记为已关闭，并且会关闭底层的 C 文件描述符。在 `f` 被垃圾回收的时候，析构函数中也会自动处理。

但由于 `stdin`, `stdout` 和 `stderr` 在 C 中的特殊地位，在 Python 中也会对它们做特殊处理。运行 `sys.stdout.close()` 会将 Python 的文件对象标记为已关闭，但是 不会关闭与之关联的文件描述符。

要关闭这三者的 C 文件描述符的话，首先你应该确认确实需要关闭它（比如，这可能会影响到处理 I/O 的拓展）。如果确实需要这么做的话，使用 `os.close()`：

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

或者也可以使用常量 0, 1, 2 代替。

4.5 網路 (Network)/網際網路 (Internet) 程式

4.5.1 Python 有哪些 WWW 工具？

参阅代码库参考手册中 `internet` 和 `netdata` 这两章的内容。Python 有大量模块来帮助你构建服务端和客户端 web 系统。

Paul Boddie 维护了一份可用框架的概览，见 <https://wiki.python.org/moin/WebProgramming>。

Cameron Laird 在 https://web.archive.org/web/20210224183619/http://phaseit.net/claird/comp.lang.python/web_python 维护了一组有关 Python web 技术的实用网页。

4.5.2 如何模擬 CGI 表單送出 (submission) (METHOD=POST) ?

我需要通过 POST 表单获取网页，有什么代码能简单做到吗？

是的，這是一個 `urllib.request` 的簡單範例：

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

import urllib.request

# build the query string
qs = "First=Josephine&MI=Q&Last=Public"

# connect and send the server a path
req = urllib.request.urlopen('http://www.some-server.out-there'
                              '/cgi-bin/some-cgi-script', data=qs)

with req:
    msg, hdrs = req.read(), req.info()
```

注意，通常在百分号编码的 POST 操作中，查询字符串必须使用 `urllib.parse.urlencode()` 处理一下。举个例子，如果要发送 `name=Guy Steele, Jr.` 的话：

```
>>> import urllib.parse
>>> urllib.parse.urlencode({'name': 'Guy Steele, Jr.'})
'name=Guy+Steele%2C+Jr.'
```

也参考：

`urllib-howto` 有大量範例。

4.5.3 我應該使用什麼模組來輔助生成 HTML ?

你可以在 [Web 编程 wiki 页面](#) 找到许多有用的链接。

4.5.4 如何從 Python 本發送郵件 ?

使用標準函式庫模組 `smtplib`。

下面是一个很简单的交互式发送邮件的代码。这个方法适用于任何支持 SMTP 协议的主机。

```
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()
```

在 Unix 系统中还可以使用 `sendmail`。`sendmail` 程序的位置在不同系统中不一样，有时是在 `/usr/lib/sendmail`，有时是在 `/usr/sbin/sendmail`。`sendmail` 手册页面会对你有所帮助。以下是示例代码：

```
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)
```

4.5.5 socket 的 `connect()` 方法怎样避免阻塞？

通常会用 `select` 模块处理 `socket` 异步 I/O。

要防止 TCP 连接发生阻塞，你可以将 `socket` 设为非阻塞模式。这样当你执行 `connect()` 时，你将要么立即完成连接（不大可能）要么得到一个包含错误编号如 `.errno` 的异常。`errno.EINPROGRESS` 表示连接正在进行中，但尚未完成。不同的操作系统将返回不同的值，所以你需要检查一下你的系统会返回什么值。

你可以使用 `connect_ex()` 方法来避免创建异常。它将只返回 `errno` 值。要进行轮询，你可以稍后再次调用 `connect_ex()` -- 0 或 `errno.EISCONN` 表示已经连接 -- 或者你也可以将此 `socket` 传给 `select.select()` 来检查它是否可写。

備註： `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 中有数据库包的接口吗？

有的。

标准 Python 还包含了基于磁盘的哈希接口例如 `DBM` 和 `GDBM`。除此之外还有 `sqlite3` 模块，该模块提供了一个轻量级的基于磁盘的关系型数据库。

大多数关系型数据库都已经支持。查看 [数据库编程 wiki 页面](#) 获取更多信息。

4.6.2 在 Python 中如何实现持久化对象？

`pickle` 库模块以一种非常通用的方式解决了这个问题（虽然你依然不能用它保存打开的文件、套接字或窗口之类的东西），此外 `shelve` 库模块可使用 `pickle` 和 `(g)dbm` 来创建包含任意 Python 对象的持久化映射。

4.7 數學和數值

4.7.1 如何在 Python 中生成隨機數？

標準模組 `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）分佈進行樣本 (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。

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、例外甚至新型態的模組，`extending-index` 文件中有相關說明。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？

是的，可以使用 C++ 中兼容 C 的功能。在 Python include 文件周围放置 `extern "C" { ... }`，并在 Python 解释器调用的每个函数之前放置 `extern "C"`。具有构造函数的全局或静态 C++ 对象可能不是一个好主意。

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，取決於你要執行的具體操作何。

[Cython](#) 及其相关的 [Pyrex](#) 是接受略微修改过的 Python 形式并生成相应 C 代码的编译器。Cython 和 Pyrex 使得人们无需学习 Python 的 C API 即可编写扩展。

如果你需要针对某些当前不存在 Python 扩展的 C 或 C++ 库的接口，你可以尝试使用 [SWIG](#) 等工具来包装这些库的数据类型和函数。[SIP](#), [CXX Boost](#) 或者 [Weave](#) 也是用于包装 C++ 库的替代方案。

5.4 如何從 C 執行任意 Python 陳述式？

执行此操作的最高层级函数为 `PyRun_SimpleString()`，它接受单个字符串参数用于在模块 `__main__` 的上下文中执行并在成功时返回 0 而在发生异常 (包括 `SyntaxError`) 时返回 -1。如果你想要更多可控性，可以使用 `PyRun_String()`；请在 `Python/pythonrun.c` 中查看 `PyRun_SimpleString()` 的源码。

5.5 如何在 C 中对任意 Python 表达式求值？

可以调用前一问题中介绍的函数 `PyRun_String()` 并附带起始标记符 `Py_eval_input`；它会解析表达式，对其求值并返回结果值。

5.6 如何從 Python 物件中提取 C 值？

这取决于对象的类型。如果是元组，`PyTuple_Size()` 将返回其长度而 `PyTuple_GetItem()` 将返回指定索引号上的项。对于列表也有类似的函数 `PyList_Size()` 和 `PyList_GetItem()`。

对于字节串，`PyBytes_Size()` 将返回其长度而 `PyBytes_AsStringAndSize()` 将提供一个指向其值和长度的指针。请注意 Python 字节串对象可能包含空字节因此不应使用 C 的 `strlen()`。

要测试物件的型，首先确保它不是 `NULL`，然后再使用 `PyBytes_Check()`、`PyTuple_Check()`、`PyList_Check()` 等函式。

还有一个针对 Python 对象的高层级 API，通过所谓的‘抽象’接口提供——请参阅 `Include/abstract.h` 了解详情。它允许使用 `PySequence_Length()`、`PySequence_GetItem()` 这样的调用来与任意种类的 Python 序列进行对接，此外还可使用许多其他有用的协议例如数字 (`PyNumber_Index()` 等) 以及 `PyMapping` API 中的各种映射等等。

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？

可以使用 `PyObject_CallMethod()` 函数来调用某个对象的任意方法。形参为该对象、要调用的方法名、类似 `Py_BuildValue()` 所用的格式字符串以及要传给方法的参数值：

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

这适用于任何具有方法的对象——不论是内置方法还是用户自定义方法。你需要负责对返回值进行最终的 `Py_DECREF()` 处理。

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”）：

```

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

```

请注意由于 `PyObject_CallObject()` 总是接受一个元组作为参数列表，要调用不带参数的函数，则传入格式为 `()`，要调用只带一个参数的函数，则应将参数包含于圆括号中，例如 `(i)`。

5.9 我如何捕捉 `PyErr_Print()` 的输出（或任何印出到 `stdout/stderr` 的东西）？

在 Python 代码中，定义一个支持 `write()` 方法的对象。将此对象赋值给 `sys.stdout` 和 `sys.stderr`。调用 `print_error` 或者只是允许标准回溯机制生效。在此之后，输出将转往你的 `write()` 方法所指向的任何地方。

最简单的方法是使用 `io.StringIO` 类：

```

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

```

实现同样效果的自定义对象看起来是这样的：

```

>>> 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？

你可以通过如下方式获得一个指向模块对象的指针：

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

如果模块尚未被导入（即它还不存在于 `sys.modules` 中），这会初始化该模块；否则它只是简单地返回 `sys.modules["<modulename>"]` 的值。请注意它并不会将模块加入任何命名空间——它只是确保模块被初始化并存在于 `sys.modules` 中。

之后你就可以通过如下方式来访问模块的属性（即模块中定义的任何名称）：

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

调用 `PyObject_SetAttrString()` 为模块中的变量赋值也是可以的。

5.11 如何在 Python 中对接 C++ 对象？

根据你的需求，可以选择许多方式。手动的实现方式请查阅“扩展与嵌入”文档来入门。需要知道的是对于 Python 运行时系统来说，C 和 C++ 并没有什么太大的区别——因此围绕一个 C 结构（指针）类型构建新 Python 对象的策略同样适用于 C++ 对象。

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 ？

安装程序必须以换行符结束，如果没有换行符，则构建过程将失败。（修复这个需要一些丑陋的 shell 脚本编程，而且这个 bug 很小，看起来不值得花这么大力气。）

5.13 如何 擴充套件除錯？

将 GDB 与动态加载的扩展名一起使用时，在加载扩展名之前，不能在扩展名中设置断点。

在您的 `.gdbinit` 文件中（或交互式）添加命令：

```
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。 什麼？

大多数打包的 Python 版本不包含 `/usr/lib/python2.x/config/` 目录，该目录中包含编译 Python 扩展所需的各种文件。

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

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

5.15 如何從「無效輸入」區分出「不完整輸入」？

有时，希望模仿 Python 交互式解释器的行为，在输入不完整时（例如，您键入了“if”语句的开头，或者没有关闭括号或三个字符串引号），给出一个延续提示，但当输入无效时，立即给出一条语法错误消息。

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

在 C 中执行此操作的最简单方法是调用 `PyRun_InteractiveLoop()`（可能在单独的线程中）并让 Python 解释器为您处理输入。您还可以设置 `PyOS_ReadlineFunctionPointer()` 指向您的自定义输入函数。有关更多提示，请参阅 `Modules/readline.c` 和 `Parser/myreadline.c`。

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

要动态加载 g++ 扩展模块，必须重新编译 Python，要使用 g++ 重新链接（在 Python Modules Makefile 中更改 `LINKCC`），及链接扩展模块（例如：`g++ -shared -o mymodule.so mymodule.o`）。

5.17 能否创建一个对象类，其中部分方法在 C 中实现，而其他方法在 Python 中实现（例如通过继承）？

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）。

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複雜。如果你經常使用「命令提示字元」執行程式，那這對你來說不會是什麼難事。如果不然，那就需要更仔細的明白了。

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啟動這樣的視窗，因為你將看到 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內容，因此你可能會很容易看到類似以下的文字：

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

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

你需要了解，你的 Python 腳本必須被另一個稱作 Python 直譯器的程序來處理。直譯器會讀取你的腳本，將其編譯成位元組碼，然後執行該位元組碼以運行你的程式。那麼，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”識別為啟動直譯器的指令。如果你已經開了一個命令視窗，則你應該試試輸入命令 py 並按下 return 鍵：

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

然後，你應該看到類似下面的內容：

```
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.
>>>
```


你已經啟動直譯器中的「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（命令列），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錄（home directory）中順利地被開，那你就會看到類似以下的內容：

```
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，標準的 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。請試著檢查您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？

請參閱如何由 Python 脚本创建能独立运行的二进制程序？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，但也有一些區別。如果你有一個名 的 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，則可輕鬆地建立一個 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 initleo()。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，且可以輕鬆配置使用空格：選擇工具 ▸ 選項 ▸ *Tab*s，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。這是一個標準的 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。

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 的標準版本會包含一個 Tcl/Tk 小工具集 (widget set) 的物件導向介面，稱之為 `tkinter`。這可能是最容易安裝（因為它已包含在 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），請參閱 [Tcl/Tk 首頁](#)。Tcl/Tk 在 macOS、Windows 和 Unix 平台上是完全可移植 (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，因為該應用程式仍然需要 Tcl 和 Tk 函式庫。

一種解決方案是將应用程序与 Tcl 和 Tk 库同一起发布，并在运行时使用 `TCL_LIBRARY` 和 `TK_LIBRARY` 环境变量指向它们的位置。

要得到真正獨立的應用程式，必須將構成函式庫的 Tcl 腳本也整合到應用程式中。一個可支援該方法的工具是 SAM (stand-alone modules, 獨立模組)，它是 Tix 發行版的一部分 (<https://tix.sourceforge.net/>)。

請在 SAM 被使用的情況下建置 Tix，對 Python 的 `Modules/tkappinit.c` 中的 `Tclsam_init()` 等函式執行適當的呼叫，以與 `libtclsam` 和 `libtkjam` 連結（你可能也會 include Tix 函式庫）。

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，你甚至不需要使用线程！但您必须稍微调整一下你的 I/O 代码。Tk 有与 Xt 的 `XtAddInput()` 对应的调用，它允许你注册一个回调函数，当可以对一个文件描述符进行 I/O 操作时，该函数将从 Tk 的主循环中被调用。参见 `tkinter-file-handlers`。

7.3.3 我無法讓鍵結 (key binding) 在 Tkinter 中作用：什麼？

一个经常听到的抱怨是：已经通过 `bind()` 方法 绑定到事件的事件处理器在对应的键被按下时并没有被处理。

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）。

「[Python](#) 被安裝在我的機器上？」常見問答集

8.1 什麼是 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，它的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](#) 開始閱讀。

8.2 [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中。

8.3 我能自行解除 Python 嗎？

需要依據 Python 的安裝方式而定。

若有人是有意地安裝 Python，你可自行移除它，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解除 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，才能再次解除這些問題。

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）[\[F\]](#)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。
- [\[F\]](#)建常數 *Ellipsis*。

2to3

一個試著將 Python 2.x 程式碼轉[\[F\]](#)[\[F\]](#) Python 3.x 程式碼的工具，它是透過處理大部分的不相容性來達成此目的，而這些不相容性能[\[F\]](#)透過剖析原始碼和遍歷剖析樹而被檢測出來。

2to3 在標準函式庫中以 `lib2to3` 被使用；它提供了一個獨立的入口點，在 `Tools/scripts/2to3`。請參[\[F\]](#) [2to3-reference](#)。

abstract base class（抽象基底類[\[F\]](#)）

抽象基底類[\[F\]](#)（又稱[\[F\]](#) *ABC*）提供了一種定義介面的方法，作[\[F\]](#)[\[F\]](#)*duck-typing*（鴨子型[\[F\]](#)）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* [\[F\]](#)用[\[F\]](#)擬的 *subclass*（子類[\[F\]](#)），它們[\[F\]](#)不繼承自另一個 *class*（類[\[F\]](#)），但仍可被 `isinstance()` 及 `issubclass()` 辨識；請參[\[F\]](#) *abc* 模組的[\[F\]](#)明文件。Python 有許多[\[F\]](#)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（[\[F\]](#)釋）

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

在執行環境（*runtime*），區域變數的[\[F\]](#)釋無法被存取，但全域變數、*class* 屬性和函式的[\[F\]](#)解，會分[\[F\]](#)被儲存在模組、*class* 和函式的 `__annotations__` 特殊屬性中。

請參[\[F\]](#)[\[F\]](#)*variable annotation*、*function annotation*、**PEP 484** 和 **PEP 526**，這些章節皆有此功能的[\[F\]](#)明。關於[\[F\]](#)釋的最佳實踐方法也請參[\[F\]](#) [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別](#), 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 (非同步生成器)

一個會回傳 *asynchronous generator iterator* (非同步生成器代器) 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成器函式, 但在某些情境中, 也可能是表示非同步生成器代器 (*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。

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

asynchronous generator iterator (非同步生成器代器)

一個由 *asynchronous generator* (非同步生成器) 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成器的函式主體, 直到遇到下一個 `yield` 運算式。

每個 `yield` 會暫停處理程序, 記住位置執行狀態 (包括區域變數及擱置中的 `try` 陳述式)。當非同步生成器代器以另一個被 `__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 (二進制檔案)

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

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，借用引用是指一种对象引用，使用该对象的代码并不持有该引用。如果对象被销毁则它就会变成一个悬空指针。例如，垃圾回收器可以移除对象的最后一个 *strong reference* 来销毁它。

對 *borrowed reference* 呼叫 `Py_INCREF()` 以將它原地 (in-place) 轉為 *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毀。`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制資料的各種運算；這些運算包括壓縮、儲存至二進制檔案和透過 *socket* (插座) 發送。

有些運算需要二進制資料是可變的。明文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制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 直譯器中的內部表示法。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 (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* (虛擬機器) 上，該虛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 虛擬機器之間運作的，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。

complex number (複數)

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成一個實部和一個虛部之和。複數就是虛數單位（-1 的平方根）的實數倍，此單位通常在數學中被寫成 i ，在工程學中被寫成 j 。Python 建立了對複數的支援，它是用後者的記法來表示複數；虛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複數，請使用 `cmath` 模組。複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，那麼幾乎能確定你可以安全地忽略它們。

context manager (情境管理器)

在 `with` 语句中通过定义 `__enter__()` 和 `__exit__()` 方法来控制环境状态的对象。参见 [PEP 343](#)。

context variable (情境變數)

一個變數，其值可以根據上下文的情境而有所不同。這類似執行緒區域儲存區 (Thread-Local Storage)，在其中，一個變數在每個執行緒可能具有不同的值。然而，關於情境變數，在一個執行緒中可能會有多个情境，而情境變數的主要用途，是在行的非同步任務 (concurrent asynchronous task) 中，對於變數狀態的追蹤。請參閱 `contextvars`。

contiguous (連續的)

如果一個緩衝區是 *C-contiguous* 或是 *Fortran contiguous*，則它會確切地被視作是連續的。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並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。另請參閱 [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，它可能會包含 `await`、`async for` 和 `async with` 關鍵字。這些關鍵字由 [PEP 492](#) 引入。

CPython

Python 程式語言的標準實作 (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。

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換 (function transformation)。裝飾器的常見範例是 `classmethod()` 和 `staticmethod()`。

裝飾器語法只是語法糖。以下兩個函式定義在語義上是等效的：

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

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

Class 也存在相同的概念，但在那比較不常用。關於裝飾器的更多內容，請參閱函式定義和 class 定義的說明文件。

descriptor (描述器)

任何定义了 `__get__()`、`__set__()` 或 `__delete__()` 方法的对象。当一个类属性为描述器时，它的特殊绑定行为就会在属性查找时被触发。通常情况下，使用 `a.b` 来获取、设置或删除一个属性时会在 `a` 类的字典中查找名称为 `b` 的对象，但如果 `b` 是一个描述器，则会调用对应的描述器方法。理解描述器的概念是更深层次理解 Python 的关键，因为这是许多重要特性的基础，包括函数、方法、特征属性、类方法、静态方法以及对超类的引用等等。

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)

一個關聯數組，其中的任意鍵都映射到相應的值。鍵可以是任何具有 `__hash__()` 和 `__eq__()` 方法的對象。在 Perl 中稱為 `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，[它](#)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` 映射到值 `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)

作為類、函數或模組之內的第一個表达式出現的字符串字面值。它在代碼被執行時會被忽略，但會被編譯器識別並放入所在類、函數或模組的 `__doc__` 屬性中。由於它可用於代碼內省，因此是存放對象的文檔的規範位置。

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)

對外公開面向文件的 API (帶有 `read()` 或 `write()` 等方法) 以使用下層資源的對象。根據其創建方式的不同，文件對象可以處理對真實磁盤文件、其他類型的存儲或通信設備的訪問（例如標準輸入/輸出、內存緩衝區、套接字、管道等）。文件對象也被稱為文件型對象或流。

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

file-like object (類檔案物件)

[file object](#) (檔案物件) 的同義字。

filesystem encoding and error handler (檔案系統編碼和錯誤處理函式)

Python 所使用的一種編碼和錯誤處理函式，用來解碼來自作業系統的位元組，以及將 Unicode 編碼到

作業系統。

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

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

filesystem encoding and error handler (檔案系統編碼和錯誤處理函式) 會在 Python 啟動時由 `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)。

從 Python 3.3 開始，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 [PEP 302](#)、[PEP 420](#) 和 [PEP 451](#) 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` 的計算結果是 `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`，因為 `-2.75` 被向下無條件舍去。請參 [PEP 238](#)。

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準的語法或語義，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 (生成器)

一個會回傳 *generator iterator* (生成器代器) 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，能生成一系列的 `yield` 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成器函式，但在某些情境中，也可能是表示生成器代器。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 (生成器代器)

一個由 *generator* (生成器) 函式所建立的物件。

每個 `yield` 會暫停處理程序，記住位置執行狀態 (包括區域變數及擱置中的 `try` 陳述式)。當生成器代器回復時，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 (生成器運算式)

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成多個值：

```
>>> sum(i*i for i in range(10))           # sum of squares 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) 來決定。

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`。它被用於型提示和解釋。

詳情請參泛型名、[PEP 483](#)、[PEP 484](#)、[PEP 585](#) 和 `typing` 模組。

GIL

請參 *global interpreter lock* (全域直譯器鎖)。

global interpreter lock (全域直譯器鎖)

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

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

過去對於建立「無限制執行緒」直譯器 (以更高的精細度鎖定共享資料的直譯器) 的努力未成功，因在一般的單一處理器情況下，效能會有所損失。一般認為，若要克服這個效能問題，會使實作變得複雜許多，進而付出更高的維護成本。

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)

一个对象如果具有在其生命期内绝不改变的哈希值 (它需要有 `__hash__()` 方法)，并可以同其他对象进行比较 (它需要有 `__eq__()` 方法) 就被称为 可哈希对象。可哈希对象必须具有相同的哈希值比较结果才会相等。

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內部使用了雜值。

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為可雜的。它們在互相比較時都是不相等的（除非它們與自己比較），而它們的雜值則是衍生自它們的 `id()`。

IDLE

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

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。它們在需要固定雜值的地方，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。只要啟動 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)`）。

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，因為有位元組碼（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，因為它所依賴的資源可能不再有了（常見的例子是函式庫模組或是警告機制）。

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

iterable（可代物件）

一種能夠逐個返回其成員項的對象。可迭代對象的例子包括所有序列類型（如 list, str 和 tuple 等）以及某些非序列類型如 dict, 文件對象以及任何定義了 `__iter__()` 方法或實現了 *sequence* 語義的 `__getitem__()` 方法的自定義類的對象。

可迭代對象可被用於 for 循環以及許多其他需要一個序列的地方（`zip()`, `map()`, ...）。當一個可迭代對象作為參數被傳給內置函數 `iter()` 時，它會返回該對象的迭代器。這種迭代器適用於對值集合的一次性遍歷。在使用可迭代對象時，你通常不需要調用 `iter()` 或者自己處理迭代器對象。for 語句會自動為你處理那些操作，創建一個臨時的未命名變量用來在循環期間保存迭代器。參見 *iterator*, *sequence* 和 *generator*。

iterator（代器）

用來表示一連串數據流的對象。重複調用迭代器的 `__next__()` 方法（或將其傳給內置函數 `next()`）將逐個返回流中的項。當沒有數據可用時則將引發 `StopIteration` 異常。到這時迭代器對象中的數據項已耗盡，繼續調用其 `__next__()` 方法只會再次引發 `StopIteration`。迭代器必須具有 `__iter__()` 方法用來返回該迭代器對象自身，因此迭代器必定也是可迭代對象，可被用於其他可迭代對象適用的大部分場合。一個顯著的例外是那些會多次重複訪問迭代項的代碼。容器對象（例如 list）在你每次將其傳入 `iter()` 函數或是在 for 循環中使用時都會產生一個新的迭代器。如果在此

情況下你嘗試用迭代器則會返回在之前迭代過程中被耗盡的同一迭代器對象，使其看起來就像是一個空容器。

在 `typeiter` 文中可以找到更多資訊。

CPython 實作細節： CPython 沒有統一應用迭代器定義 `__iter__()` 的要求。

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()` 被用來產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 可以作爲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，請參閱如何排序。

keyword argument (關鍵字引數)

請參閱 [argument](#) (引數)。

lambda

由單一 *expression* (運算式) 所組成的一個匿名行內函式 (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，明確地測試先驗條件。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」之間引入了競態條件 (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)

一種 Python 內置 *sequence*。雖然名為列表，但它更類似於其他語言中的數組而非鏈表，因為訪問元素的時間複雜度為 $O(1)$ 。

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]` 會產生一個字串 `list`，其中包含 0 到 255 範圍內，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。它必須定義一個名爲 `load_module()` 的 `method` (方法)。載入器通常是被 *finder* (尋檢器) 回傳。更多細節請參閱 [PEP 302](#)，關於 *abstract base class* (抽象基底類)，請參閱 `importlib.abc.Loader`。

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" 作爲區域編碼。

`locale.getencoding()` 可被用來獲取語言區域編碼格式。

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。關於第 2.3 版至今，`Python` 直譯器所使用的演算法細節，請參 `Python 2.3 版方法解析順序`。

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。

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
```

有些具名元組是內置類型（比如上面的例子）。此外，具名元組還可通過常規類定義從 `tuple` 繼承並定義指定名稱的字段的方式來創建。這樣的類可以手工編號，或者也可以通過繼承 `typing.NamedTuple`，或者使用工廠函數 `collections.namedtuple()` 來創建。後一種方式還會添加一些手工編寫或內置的具名元組所沒有的額外方法。

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、全域的及建立的命名空間，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，這些函式分別是由 `random` 和 `itertools` 模組在實作。

namespace package（命名空間套件）

一個 [PEP 420 package](#)（套件），它只能作子套件（subpackage）的一個容器。命名空間套件可能沒有實體的表示法，而且具體來說它們不像是 [regular package](#)（正規套件），因為它們沒有 `__init__.py` 這個檔案。

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。舉例來說，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，在預設情況下，巢狀作用域僅適用於參照，而無法用於賦值。區域變數能在最內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）

對目前已被應用於所有類對象的類形式的舊稱謂。在較早的 Python 版本中，只有新式類能夠使用 Python 新增的更靈活者，如 `__slots__`、描述器、特征屬性、`__getattr__()`、類方法和靜態方法等。

object（物件）

具有狀態（屬性或值）及被定義的行為（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）。

package（套件）

一個 Python 的 [module](#)（模組），它可以包含子模組（submodule）或是遞歸的子套件（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），或在某些情況下指示多個引數。共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。

另請參閱術語表的 *argument* (引數) 條目、常見問題中的 *引數和參數之間的差別*、`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`，請參閱 `importlib.abc.PathEntryFinder`。

path entry hook (路徑項目)

一種可調用對象，它在知道如何查找特定 *path entry* 中的模块的情况下能够使用 `sys.path_hooks` 列表返回一个 *path entry finder*。

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為 `str` 或 `bytes` 檔案系統路徑；而 `os.fsdecode()` 及 `os.fsencode()` 則分別用於確保 `str` 及 `bytes` 的結果。由 **PEP 519** 引入。

PEP

Python Enhancement Proposal (Python 增進提案)。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反對意見。

請參閱 **PEP 1**。

portion (部分)

在單一目標中的一組檔案（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)

請參閱 *argument* (引數)。

provisional API (暫行 API)

暫行 API 是指，從標準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為有必要，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為「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。

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

provisional package (暫行套件)

請參閱 [provisional API](#) (暫行 API)。

Python 3000

Python 3.x 系列版本的暱稱 (很久以前創造的，當時第 3 版的發布是在遙遠的未來。) 也可以縮寫為「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，對一個可迭代物件的所有元素進行遍歷。許多其他語言也有這種類型的架構，所以不熟悉 Python 的人有時會使用一個數值計數器來代替：

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

相較之下，以下方法更簡潔、更具有 Python 風格：

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

qualified name (限定名稱)

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

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

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

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

reference count (參照計數)

對於一個物件的參照次數。當一個物件的參照計數下降到零時，它會被解除配置 (*deallocated*)。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` 檔案的目錄。

另請參閱 [namespace package](#) (命名空間套件)。

__slots__

在 `class` 內部的一個宣告，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況。

sequence (序列)

一種 *iterable*，它支持通过 `__getitem__()` 特殊方法来使用整数索引进行高效的元素访问，并定义了一个返回序列长度的 `__len__()` 方法。内置的序列类型有 `list`, `str`, `tuple` 和 `bytes` 等。请注意虽然 `dict` 也支持 `__getitem__()` 和 `__len__()`，但它被归类为映射而非序列，因为它使用任意 *immutable* 键而不是整数来查找元素。

`collections.abc.Sequence` 抽象基类定义了一个更丰富的接口，它在 `__getitem__()` 和 `__len__()` 之外，还添加了 `count()`, `index()`, `__contains__()` 和 `__reversed__()`。实现此扩展接口的类型可以使用 `register()` 来显式地注册。要获取有关通用序列方法的更多文档，请参阅通用序列操作。

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。

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 -- 静态类型检查器

读取 Python 代码并进行分析，以查找问题例如拼写错误的外部工具。另请参阅 *类型提示* 以及 `typing` 模块。

strong reference (參照)

在 Python 的 C API 中，强引用是指为持有引用的代码所拥有的对象的引用。在创建引用时可通过调用 `Py_INCREF()` 来获取强引用而在删除引用时可通过 `Py_DECREF()` 来释放它。

`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](#) (二進制檔案)，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別有用。

type (型)

一個 Python 物件的型定义了它是什麼類型的物件；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)` 來檢查。

type alias (型名)

一個型的同義詞，透過將型指定給一個識符 (identifier) 來建立。

型名對於簡化型提示 (*type hint*) 很有用。例如：

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

可以寫成這樣，更具有可讀性：

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

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

請參 [typing](#) 和 [PEP 484](#)，有此功能的描述。

type hint (型提示)

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

类型提示是可选的而不是 Python 的强制要求，但它们对静态类型检查器很有用处。它们还能协助 IDE 实现代码补全与重构。

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](#) (函式註釋)、[PEP 484](#) 和 [PEP 526](#)，皆有此功能的描述。關於註釋的最佳實踐方法，另請參 [annotations-howto](#)。

virtual environment (虛擬環境)

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 發行套件，而不會對同一個系統上運行的其他 Python 應用程式的行為產生干擾。

另請參 [venv](#)。

virtual machine (虛擬機器)

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

Zen of Python (Python 之禪)

Python 設計原則與哲學的列表，其內容有助於理解和使用此語言。此列表可以透過在互動式提示字元後輸入 `import this` 來找到它。

關於這些文檔文件

這些文檔文件是透過 [Sphinx](#)（一個專為 Python 文檔文件所撰寫的文件處理器）將使用 [reStructuredText](#) 撰寫的原始檔轉換而成。

如同 Python 自身，透過自願者的努力下輸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，包含相關資訊。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內容的作者；
- 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

許多人都曾為 Python 這門語言、Python 標準函式庫和 Python 文檔文件貢獻過。Python 所發出的原始碼中含有部份貢獻者的清單，請見 [Misc/ACKS](#)。

正因為 Python 社群的撰寫與貢獻才有這份這棒的文檔文件 -- 感謝所有貢獻過的人們！

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，目的是作一種稱作 ABC 語言的後繼者。儘管 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，在那發行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（現 Zope Corporation；見 <https://www.zope.org/>）。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 相關的智慧財產權而創立的非營利組織。Zope Corporation 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，參見 <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異。

發行版本	源自	年份	擁有者	GPL 相容性？
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	否
2.1	2.0+1.6.1	2001	PSF	否
2.0.1	2.0+1.6.1	2001	PSF	是
2.1.1	2.1+2.0.1	2001	PSF	是
2.1.2	2.1.1	2002	PSF	是
2.1.3	2.1.2	2002	PSF	是
2.2 以上	2.1.1	2001 至今	PSF	是

備註：GPL 相容並不表示我們是在 GPL 下發 Python。不像 GPL，所有的 Python 授權都可以讓您發修改後的版本，但不一定要使您的變更成開源。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。

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 *PSF* 授權合約。

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 授權合約以及 *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 用於 PYTHON 3.11.13 的 PSF 授權合約

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),
→ and
the Individual or Organization ("Licensee") accessing and otherwise using
→ Python
3.11.13 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby
grants Licensee a nonexclusive, royalty-free, world-wide license to
→ reproduce,
analyze, test, perform and/or display publicly, prepare derivative works,
distribute, and otherwise use Python 3.11.13 alone or in any derivative
version, provided, however, that PSF's License Agreement and PSF's notice
→ of
copyright, i.e., "Copyright © 2001–2023 Python Software Foundation; All
→ Rights
Reserved" are retained in Python 3.11.13 alone or in any derivative version
prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or
incorporates Python 3.11.13 or any part thereof, and wants to make the
derivative work available to others as provided herein, then Licensee
→ hereby
agrees to include in any such work a brief summary of the changes made to
→ Python
3.11.13.
4. PSF is making Python 3.11.13 available to Licensee on an "AS IS" basis.
PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF
EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION
→ OR
WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT
→ THE
USE OF PYTHON 3.11.13 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.11.13 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
 →MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.11.13, OR ANY
 →DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach
 →of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any
 →relationship of agency, partnership, or joint venture between PSF and Licensee. This
 →License Agreement does not grant permission to use PSF trademarks or trade name in
 →a trademark sense to endorse or promote products or services of Licensee, or
 →any third party.
8. By copying, installing or otherwise using Python 3.11.13, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.2 用於 PYTHON 2.0 的 BEOPEN.COM 授權合約

BEOPEN PYTHON 開源授權合約第 1 版

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

(繼續下一頁)

(繼續上一頁)

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 用於 PYTHON 1.6.1 的 CNRI 授權合約

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the internet using the following URL: <http://hdl.handle.net/1895.22/1013>."
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

(繼續下一頁)

(繼續上一頁)

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 用於 PYTHON 0.9.0 至 1.2 的 CWI 授權合約

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

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

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

C.2.5 用於 PYTHON 3.11.13 明文文件程式碼的 ZERO-CLAUSE BSD 授權

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
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 使用了 getaddrinfo() 和 getnameinfo() WIDE 项目的不同源文件中: <https://www.wide.ad.jp/>

Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

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

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

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

C.3.3 非同步 socket 服務

asynchat 和 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 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F

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,
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 模組包含以下聲明：

```
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 模块包含以下说明:

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

(繼續下一頁)

(繼續上一頁)

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

C.3.10 SipHash24

Python/pyhash.c 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換。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.
 */
```

(繼續下一頁)

(繼續上一頁)

```
*
*****/
```

C.3.12 OpenSSL

如果操作系统提供支持, 则 `hashlib`, `posix`, `ssl`, `crypt` 模块会使用 OpenSSL 库来提高性能。此外, Python 的 Windows 和 macOS 安装程序可能会包括 OpenSSL 库的副本, 所以我们在此包括一份 OpenSSL 许可证的副本。对于 OpenSSL 3.0 版及其后续衍生版本, 均使用 Apache 许可证 v2:

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

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

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

"Licensor" shall mean the copyright owner or entity authorized by 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.

(繼續下一頁)

(繼續上一頁)

"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.

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

(繼續下一頁)

(繼續上一頁)

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 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,

(繼續下一頁)

(繼續上一頁)

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

pyexpat 擴展是使用所包括的 **expat** 源副本來構建的，除非配置了 `--with-system-expat`:

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

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

C.3.14 libffi

除非在建置 `_ctypes` 擴充時設定 `--with-system-libffi`，否則該擴充會用一個含 **libffi** 原始碼的副本來建置：

Copyright (c) 1996-2008 Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the ``Software''), to deal in the Software without restriction, including

(繼續下一頁)

(繼續上一頁)

without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

C.3.15 zlib

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

Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it 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
jloup@gzip.org

Mark Adler
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.
```

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.

(繼續下一頁)

(繼續上一頁)

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

C.3.18 W3C C14N 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 3.11.13:

```
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 audioop

audioop 模块使用了 SoX 项目的 g771.c 文件中的基础代码。<https://sourceforge.net/projects/sox/files/sox/12.17.7/sox-12.17.7.tar.gz>

此源代码是 Sun Microsystems, Inc. 的产品并可供无限制地使用。用户可以拷贝或修改此源代码而无须付费。

SUN SOURCE CODE IS PROVIDED AS IS WITH NO WARRANTIES OF ANY KIND INCLUDING THE WARRANTIES OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, OR ARISING FROM A COURSE OF DEALING, USAGE OR TRADE PRACTICE.

提供的 Sun 源代码不附带技术支持并且 Sun Microsystems, Inc. 也没有义务协助其使用、排错、修改或增强。

SUN MICROSYSTEMS, INC. SHALL HAVE NO LIABILITY WITH RESPECT TO THE INFRINGEMENT OF COPYRIGHTS, TRADE SECRETS OR ANY PATENTS BY THIS SOFTWARE OR ANY PART THEREOF.

在任何情况下 Sun Microsystems, Inc. 均不对任何收入或利润损失或其他特殊的、间接的和后续的伤害负责, 即使 Sun 已被告知可能发生此类伤害。

Sun Microsystems, Inc. 2550 Garcia Avenue Mountain View, California 94043

C.3.20 asyncio

asyncio 模块的某些部分来自 uvloop 0.16, 它是基于 MIT 许可证发行的:

```
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.
```

版權宣告

Python 和這份圖明文件的版權：

Copyright © 2001-2023 Python Software Foundation 保留一切權利。

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](#)。

非依字母順序

..., 79
 2to3, 79
 >>>, 79
 __future__, 84
 __slots__, 91
 環境變數
 PATH, 54
 PYTHONDONTWRITEBYTECODE, 38

A

abstract base class (抽象基底類 \boxplus), 79
 annotation (註釋), 79
 argument (引數), 80
 與 parameter (參數) 的差 \boxplus , 14
 asynchronous context manager (非同步情境管理器), 80
 asynchronous generator iterator (非同步 \boxplus 生器 \boxplus 代器), 80
 asynchronous generator (非同步 \boxplus 生器), 80
 asynchronous iterable (非同步可 \boxplus 代物件), 80
 asynchronous iterator (非同步 \boxplus 代器), 80
 attribute (屬性), 80
 awaitable (可等待物件), 81

B

BDFL, 81
 binary file (二進制檔案), 81
 borrowed reference (借用參照), 81
 bytecode (位元組碼), 81
 bytes-like object (類位元組串物件), 81

C

callable (可呼叫物件), 81
 callback (回呼), 81
 C-contiguous (C 連續的), 82
 class variable (類 \boxplus 變數), 81
 class (類 \boxplus), 81

complex number (複數), 82
 context manager (情境管理器), 82
 context variable (情境變數), 82
 contiguous (連續的), 82
 coroutine function (協程函式), 82
 coroutine (協程), 82
 CPython, 82

D

decorator (裝飾器), 82
 descriptor (描述器), 82
 dictionary comprehension (字典綜合運算), 83
 dictionary view (字典檢視), 83
 dictionary (字典), 83
 docstring (註明字串), 83
 duck-typing (鴨子型 \boxplus), 83

E

EAFP, 83
 expression (運算式), 83
 extension module (擴充模組), 83

F

f-string (f 字串), 83
 file object (檔案物件), 83
 file-like object (類檔案物件), 83
 filesystem encoding and error handler (檔案系統編碼和錯誤處理函式), 83
 finder (尋檢器), 84
 floor division (向下取整除法), 84
 Fortran contiguous (Fortran 連續的), 82
 function annotation (函式 \boxplus 釋), 84
 function (函式), 84

G

garbage collection (垃圾回收), 84
 generator expression (生器 \boxplus 運算式), 85
 generator iterator (生器 \boxplus 代器), 85

generator (生成器), 84
 generic function (泛型函式), 85
 generic type (泛型型別), 85
 GIL, 85
 global interpreter lock (全域直譯器鎖), 85

H

hash-based pyc (雜項架構的 pyc), 85
 hashable (可雜項的), 85

I

IDLE, 86
 immutable (不可變物件), 86
 import path (引入路徑), 86
 importer (引入器), 86
 importing (引入), 86
 interactive (互動的), 86
 interpreted (直譯的), 86
 interpreter shutdown (直譯器關閉), 86
 iterable (可迭代物件), 86
 iterator (迭代器), 86

K

key function (鍵函式), 87
 keyword argument (關鍵字引數), 87

L

lambda, 87
 LBYL, 87
 list comprehension (串列綜合運算), 87
 list (串列), 87
 loader (載入器), 87
 locale encoding (區域編碼), 87

M

magic
 method (方法), 88
 magic method (魔術方法), 88
 mapping (對映), 88
 meta path finder (元路徑尋檢器), 88
 metaclass (元類), 88
 method resolution order (方法解析順序), 88
 method (方法), 88
 magic, 88
 special, 92
 module spec (模組規格), 88
 module (模組), 88
 MRO, 88
 mutable (可變物件), 88

N

named tuple (附名元組), 88
 namespace package (命名空間套件), 89

namespace (命名空間), 89
 nested scope (巢狀作用域), 89
 new-style class (新式類), 89

O

object (物件), 89

P

package (套件), 89
 parameter (參數), 89
 與 argument (引數) 的差別, 14
 PATH, 54
 path based finder (基於路徑的尋檢器), 90
 path entry finder (路徑項目尋檢器), 90
 path entry hook (路徑項目), 90
 path entry (路徑項目), 90
 path-like object (類路徑物件), 90
 PEP, 90
 portion (部分), 90
 positional argument (位置引數), 90
 provisional API (暫行 API), 90
 provisional package (暫行套件), 91
 Python 3000, 91
 Python Enhancement Proposals
 PEP 1, 90
 PEP 5, 6
 PEP 8, 10, 35, 74
 PEP 238, 84
 PEP 278, 93
 PEP 302, 84, 87
 PEP 343, 82
 PEP 362, 80, 90
 PEP 387, 3
 PEP 411, 91
 PEP 420, 84, 89, 90
 PEP 443, 85
 PEP 451, 84
 PEP 483, 85
 PEP 484, 79, 84, 85, 93, 94
 PEP 492, 8082
 PEP 498, 83
 PEP 519, 90
 PEP 525, 80
 PEP 526, 79, 94
 PEP 572, 43
 PEP 585, 85
 PEP 602, 5
 PEP 3116, 93
 PEP 3147, 38
 PEP 3155, 91
 PYTHONDONTWRITEBYTECODE, 38
 Pythonic (Python 風格的), 91

Q

qualified name (限定名稱), 91

R

reference count (參照計數), 91

regular package (正規套件), 91

S

sequence (序列), 92

set comprehension (集合綜合運算), 92

single dispatch (單一調度), 92

slice (切片), 92

special

 method (方法), 92

special method (特殊方法), 92

statement (陳述式), 92

static type checker -- 静态类型检查器, 92

strong reference (強參照), 92

T

text encoding (文字編碼), 92

text file (文字檔案), 92

triple-quoted string (三引號字串), 93

type alias (型名), 93

type hint (型提示), 93

type (型), 93

U

universal newlines (通用行字元), 93

V

variable annotation (變數釋), 93

virtual environment (擬環境), 94

virtual machine (擬機器), 94

Z

Zen of Python (Python 之), 94