Python Frequently Asked Questions

发行版本 *3.13.6*

Guido van Rossum and the Python development team

八月 08, 2025

Python Software Foundation
Email: docs@python.org

Contents

CHAPTER 1

Python 常见问题

1.1 一般信息

1.1.1 什么是 Python？

Python 是一种解释型、交互式、面向对象的编程语言。它包含了模块、异常、动态类型、高层级动态数据类型以及类等特性。在面向对象编程以外它还支持多种编程范式，例如过程式和函数式编程等。Python 结合了超强的功能和极清晰的语法。它带有许多系统调用和库以及多种窗口系统的接口，并且能用 C 或 C++ 来进行扩展。它还可用作需要可编程接口的应用程序的扩展语言。最后，Python 非常易于移植：它可以在包括 Linux 和 macOS 在内的许多 Unix 变种以及 Windows 上运行。

要了解更多详情，请先查看 tutorial-index。Python 新手指南 提供了学习 Python 的其他入门教程及资源的链接。

1.1.2 什么是 Python 软件基金会？

Python 软件基金会（Python Software Foundation，简称 PSF）是一个独立的非盈利组织，它拥有 Python 2.1 及以上各版本的版权。PSF 的使命是推进与 Python 编程语言相关的开源技术，并推广 Python 的使用。PSF 的主页是 https://www.python.org/psf/。

向 PSF 提供捐助在美国是免税的。如果你在使用 Python 并且感觉它对你很有帮助，可以通过 PSF 捐助页 进行捐助。

1.1.3 使用 Python 是否存在版权限制？

你可以任意使用源码，只要你保留版权信息并在你基于 Python 的产品文档中显示该版权信息。如果你遵守此版权规则，就可以将 Python 用于商业领域，以源码或二进制码的形式（不论是否经过修改）销售 Python 的副本，或是以某种形式包含了 Python 的产品。当然，我们仍然希望获知所有对 Python 的商业使用。

请参阅 许可页 以查看进一步的说明以及 PSF 许可的完整文本。

Python 的徽标是注册商标，在某些情况下需要获得允许方可使用。请参阅 商标使用政策 了解详情。

1.1.4 创造 Python 的最初理由是什么？

以下是有关最初缘起的一份 非常简短的摘要，由 Guido van Rossum 本人撰写:

> 我在 CWI 的 ABC 部门时在实现解释型语言方面积累了丰富经验，通过与这个部门成员的协同工作，我学到了大量有关语言设计的知识。这是许多 Python 特性的最初来源，包括使用缩进来组织语句以及包含非常高层级的数据结构（虽然在 Python 中具体的实现细节完全不同）。
>
> 我对 ABC 语言有过许多抱怨，但同时也很喜欢它的许多特性。没有可能通过扩展 ABC 语言（或它的实现）来弥补我的不满——实际上缺乏可扩展性就是它最大的问题之一。我也有一些使用 Modula-2+ 的经验，并曾与 Modula-3 的设计者进行交流，还阅读了 Modula-3 的报告。Modula-3 是 Python 中异常机制所用语法和语义，以及其他一些语言特性的最初来源。
>
> 我还曾在 CWI 的 Amoeba 分布式操作系统部门工作。当时我们需要有一种比编写 C 程序或 Bash 脚本更好的方式来进行系统管理，因为 Amoeba 有它自己的系统调用接口，并且无法方便地通过 Bash 来访问。我在 Amoeba 中处理错误的经验令我深刻地意识到异常处理在编程语言特性当中的重要地位。
>
> 我发现，某种具有 ABC 式的语法而又能访问 Amoeba 系统调用的脚本语言将可满足需求。我意识到编写一种 Amoeba 专属的语言是愚蠢的，所以我决定编写一种具有全面可扩展性的语言。
>
> 在 1989 年的圣诞假期中，我手头的时间非常充裕，因此我决定开始尝试一下。在接下来的一年里，虽然我仍然主要用我的业余时间来做这件事，但 Python 在 Amoeba 项目中的使用获得了很大的成功，来自同事的反馈让我得以增加了许多早期的改进。
>
> 到 1991 年 2 月，经过一年多的开发，我决定将其发布到 USENET。之后的事情就都可以在 `Misc/HISTORY` 文件里面看了。

1.1.5 Python 适合做什么？

Python 是一种高层级的多用途编程语言，可用于解决许多不同门类的问题。

本语言自带一个庞大标准库，所涵盖的编程领域包括字符串处理（正则表达式、Unicode、文件间的差异比较等），互联网协议（HTTP, FTP, SMTP, XML-RPC, POP, IMAP），软件工程（单元测试、日志记录、性能分析、Python 代码解析），以及操作系统接口（系统调用、文件系统、TCP/IP 套接字）。请查看 library-index 目录页以获取所有可用内容的概览。此外还有大量第三方扩展包可供使用。请访问 Python 软件包索引 来查找你感兴趣的软件包。

1.1.6 Python 版本的编号形式是怎样的？

Python 版本的编号形式为”A.B.C” 或”A.B”:

- *A* 是主版本号 -- 它仅会针对语言中非常重大的改变而递增。
- *B* 是次版本号 -- 它会针对不太重大的改变而递增。
- *C* 是微版本号 -- 它针对每次问题修正发布而递增。

并非所有发布版本都是问题修正版本。在新特征发布版本的开发过程中，会制作一系列的开发版本，它们以 alpha, beta 或 release candidate 来标示。其中 alpha 版本是早期发布版，它的接口尚未最终确定；在两个 alpha 发布版本间出现接口的改变并不意外。而 beta 版本更为稳定，它会保留现有的接口，但也可能增加新的模块，而 release candidate 版则会保持冻结状态，不做任何改变，除非有需要修复的严重问题。

Alpha, beta 和候选发布版带有额外的后缀:

- 带有某个小数字 *N* 的 alpha 版后缀是”aN”。
- 带有某个小数字 *N* 的 beta 版后缀是”bN”。
- 带有某个小数字 *N* 的候选发布版后缀是”rcN”。

换句话说，所有标记为 *2.0aN* 的版本都早于标记为 *2.0bN* 的版本，后者又都早于标记为 *2.0rcN* 的版本，而 后者又都早于标记为 2.0 的版本。

你还可能看到带有 “+” 后缀的版本号，例如 “2.2+”。这表示未发布版本，直接基于 CPython 开发代码仓库构建。在实际操作中，当一个小版本最终发布后，未发布版本号会递增到下一个小版本号，成为 “a0” 版本，例如 “2.4a0”。

请参阅 Developer's Guide 获取更多有关开发流程的信息，并参阅 **PEP 387** 了解更多有关 Python 的向下兼容策略的信息。另请参阅有关 `sys.version`, `sys.hexversion` 和 `sys.version_info` 的文档。

1.1.7 我应如何获取一份 Python 源代码的副本？

最新的 Python 发布版源代码总能从 python.org 获取，下载页链接为 https://www.python.org/downloads/。最新的开发版源代码可以在 https://github.com/python/cpython/ 获取。

发布版源代码是一个以 gzip 压缩的 tar 文件，其中包含完整的 C 源代码、Sphinx 格式的文档、Python 库模块、示例程序以及一些有用的自由分发软件。该源代码将可在大多数 UNIX 类平台上直接编译并运行。

请参阅 Python 开发者指南的初步上手部分 了解有关获取源代码并进行编译的更多信息。

1.1.8 我应如何获取 Python 的文档？

当前的 Python 稳定版本的标准文档可在 https://docs.python.org/3/ 查看。也可在 https://docs.python.org/3/download.html 获取 PDF、纯文本以及可下载的 HTML 版本。

文档以 reStructuredText 格式撰写并使用 Sphinx 文档工具 生成。文档的 reStructuredText 源文件是 Python 源代码发布版的一部分。

1.1.9 我之前从未接触过编程。哪里有 Python 的教程？

有许多可选择的教程和书籍。标准文档中也包含有 tutorial-index。

请参阅 新手指南 以获取针对 Python 编程初学者的信息，包括教程的清单。

1.1.10 是否有专门针对 Python 的新闻组或邮件列表？

有一个新闻组 *comp.lang.python* 和一个邮件列表 python-list。新闻组和邮件列表是彼此互通的——如果你可以阅读新闻就不必再订阅邮件列表。*comp.lang.python* 的流量很大，每天会收到数以百计的发帖，Usenet 使用者通常更擅长处理这样大的流量。

有关新软件发布和活动的公告可以在 comp.lang.python.announce 中找到，这是个严格管理的低流量列表，每天发帖五个左右。可在 python-announce 邮件列表 订阅。

有关其他邮件列表和新闻组的更多信息可以在 https://www.python.org/community/lists/ 找到。

1.1.11 我应如何获取 Python 的公开测试版本？

可以从 https://www.python.org/downloads/ 下载 alpha 和 beta 发布版。所有发布版都会在 comp.lang.python 和 comp.lang.python.announce 新闻组以及 Python 主页 https://www.python.org/ 上进行公告；并会推送到 RSS 新闻源。

你还可以通过 Git 访问 Python 的开发版。请参阅 Python 开发者指南 了解详情。

1.1.12 我应如何为 Python 提交错误报告和补丁？

要报告问题或提交补丁，请使用位于 https://github.com/python/cpython/issues 的问题追踪器。

有关 Python 开发流程的更多信息，请参阅 Python 开发者指南。

1.1.13 是否有任何公开发表的 Python 相关文章可以供我参考引用？

可能作为参考文献的最好方式还是引用你喜欢的 Python 相关书籍。

有关 Python 的 最早的文章 撰写于 1991 年因而现在已相当过时。

> Guido van Rossum 与 Jelke de Boer, ” 使用 Python 编程语言交互式地测试远程服务器”, CWI 季刊, 第 4 卷, 第 4 期 (1991 年 12 月), 阿姆斯特丹, 第 283--303 页。

1.1.14 是否有任何 Python 相关的书籍？

是的，相关的书籍很多，还有更多即将发行。请访问 python.org 的 wiki 页面 https://wiki.python.org/moin/PythonBooks 获取一份清单。

你也可以到各大在线书店搜索"Python" 并过滤掉对 Monty Python 的引用；或者也可以搜索"Python" 加"language"。

1.1.15 www.python.org 具体位于世界上的哪个地点？

Python 项目的基础设施分布于世界各地并由 Python 基础设施团队负责管理。相关细节请访问 这里。

1.1.16 为何命名为 Python？

在着手编写 Python 实现的时候，Guido van Rossum 同时还阅读了刚出版的 "Monty Python 的飞行马戏团" 剧本，这是一部自 1970 年代开始播出的 BBC 系列喜剧。Van Rossum 觉得他需要选择一个简短、独特而又略显神秘的名字，于是他决定将这个新语言命名为 Python。

1.1.17 我必须喜欢"Monty Python 的飞行马戏团" 吗？

不必，但这对学习会有帮助。:)

1.2 现实世界中的 Python

1.2.1 Python 有多稳定？

非常稳定。自 1991 年起大约每隔 6 至 18 个月就会推出新的稳定发布版，这种状态看来还会持续下去。从 3.9 版开始，Python 将会每隔 12 个月推出一个新增特征版本 (**PEP 602**)。

开发者也会推出较旧版本的问题修正发布版，因此现有发布版的稳定性还会逐步提升。问题修正发布版会以版本号第三部分的数字来标示（例如 3.5.3, 3.6.2），用于稳定性管理；只有对已知问题的修正会包含在问题修正发布版中，而同一系列的问题修正发布版中的接口将会始终保持一致。

最新的稳定发布版总是可以在 Python 下载页 中找到。Python 3.x 是推荐的版本并被大多数广泛使用的库所支持。Python 2.x **已不再维护**。

1.2.2 有多少人在使用 Python？

使用者应该数以百万计，但很难获得一个精确的数字。

Python 可以免费下载，因此并不存在销量数据，此外它也可以从许多不同网站获取，并且包含于许多 Linux 发行版之中，因此下载量统计同样无法完全说明问题。

comp.lang.python 新闻组非常活跃，但不是所有 Python 用户都会在新闻组发帖，许多人甚至不会阅读新闻组。

1.2.3 有哪些重要的项目是用 Python 开发的？

请访问 https://www.python.org/about/success 查看使用了 Python 的项目列表。阅览 历次 Python 会议 的日程纪要可以看到许多不同公司和组织所做的贡献。

高水准的 Python 项目包括 Mailman 邮件列表管理器 和 Zope 应用服务器。多个 Linux 发行版，其中最著名的是 Red Hat，都使用 Python 来编写其部分或全部的安装器和系统管理软件。在内部使用 Python 的公司包括了 Google, Yahoo 和 Lucasfilm 等等。

1.2.4 在未来可以期待 Python 将有什么新进展？

请访问 https://peps.python.org/ 查看 Python 增强提议（PEP）。PEP 是为 Python 加入某种新特性的提议进行描述的设计文档，其中会提供简明的技术规格说明与基本原理。可查找标题为"Python X.Y Release Schedule" 的 PEP，其中 X.Y 是某个尚未公开发布的版本。

新版本的开发会在 python-dev 邮件列表 中进行讨论。

1.2.5 提议对 Python 加入不兼容的更改是否合理？

通常来说是不合理的。世界上已存在的 Python 代码数以亿计，因此，任何对该语言的更改即便仅会使得现有程序中极少的一部分失效也是难以令人接受的。就算你可以提供一个转换程序，也仍然存在需要更新全部文档的问题；另外还有大量已出版的 Python 书籍，我们不希望让它们在一瞬间全部变成废纸。

如果必须更改某个特性，则应该提供渐进式的升级路径。**PEP 5** 描述了引入向后不兼容的更改所需遵循的流程，以尽可能减少对用户的干扰。

1.2.6 Python 是一种对编程初学者友好的语言吗？

是的。

从过程式、静态类型的编程语言例如 Pascal, C 或者 C++ 以及 Java 的某一子集开始引导学生入门仍然是常见的做法。但以 Python 作为第一种编程语言进行学习对学生可能更有利。Python 具有非常简单和一致的语法和庞大的标准库，而且最重要的是，在编程入门教学中使用 Python 可以让学生专注于更重要的编程技能，例如问题分解与数据类型设计。使用 Python，可以快速向学生介绍基本概念例如循环与过程等。他们甚至有可能在第一次课里就开始接触用户自定义对象。

对于之前从未接触过编程的学生来说，使用静态类型语言会感觉不够自然。这会给学生带来必须掌握的额外复杂性，并减慢教学的进度。学生需要尝试像计算机一样思考，分解问题，设计一致的接口并封装数据。虽然从长远来看，学习和使用一种静态类型语言是很重要的，但这并不是最适宜在学生的第一次编程课上就进行探讨的主题。

还有许多其他方面的特点使得 Python 成为很好的入门语言。像 Java 一样，Python 拥有一个庞大的标准库，因此可以在课程非常早期的阶段就给学生布置一些 *实用* 的编程项目。编程作业不必仅限于标准四则运算和账目检查程序。通过使用标准库，学生可以在学习编程基础知识的同时开发真正的应用，从而获得更大的满足感。使用标准库还能使学生了解代码重用的概念。而像 PyGame 这样的第三方模块同样有助于扩大学生的接触领域。

Python 的解释器使学生能够在编程时测试语言特性。他们可以在一个窗口中输入程序源代码的同时开启一个解释器运行窗口。如果他们不记得列表有哪些方法，他们这以这样做:

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

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

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

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

通过使用解释器，学生编写程序时参考文档总是能伴随在他们身边。

Python 还拥有一些很好的 IDE。IDLE 是一个以 Python 基于 Tkinter 编写的跨平台 Python IDE。Emacs 用户将高兴地了解到 Emacs 具有非常好的 Python 模式。所有这些编程环境都提供语法高亮、自动缩进以及在编写代码时使用交互式解释器等功能。请访问 Python wiki 查看 Python 编程环境的完整列表。

如果你想要讨论 Python 在教育中的使用，你可能会有兴趣加入 edu-sig 邮件列表。

CHAPTER 2

编程常见问题

2.1 一般问题

2.1.1 Python 有没有提供带有断点、单步调试等功能的源码级调试器？

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。

有许多商业 Python IDE 都包含了图形化调试器。包括：

- Wing IDE
- Komodo IDE
- PyCharm

2.1.2 是否有能帮助寻找漏洞或执行静态分析的工具？

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。

以下代码:

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

正常工作，但是以下代码

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

在 `UnboundLocalError` 中的结果:

```
>>> 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 的局部变量和全局变量有哪些规则？

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   # 'x' 配置设置的默认值
```

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。

如果只是在一个函数中用到第 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={}):  # 危险：所有调用共享对一个字典的引用
    ... 执行一些计算 ...
    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={}):
    ...
```

而应这么写：

```
def foo(mydict=None):
    if mydict is None:
        mydict = {}  # 为局部命名空间新建一个字典
```

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：

```
# 调用方只能提供两个形参并可选择以关键字形式传入 _cache
def expensive(arg1, arg2, *, _cache={}):
    if (arg1, arg2) in _cache:
        return _cache[(arg1, arg2)]

    # 计算结果值
    result = ... 高耗费的计算 ...
    _cache[(arg1, arg2)] = result           # 将结果保存在缓存中
    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 形参和实参之间有什么区别？

形参 是由出现在函数定义中的名称来定义的，而*参数* 则是在调用函数时实际传入的值。形参定义了一个函数能接受什么*参数种类*。例如，对于以下函数定义：

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

foo 、*bar* 和 *kwargs* 是 `func` 的形参。不过在调用 `func` 时，例如：

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

`42` 、`314` 和 `somevar` 则是实参。

2.2.9 为什么修改列表'y' 也会更改列表'x'？

如果代码编写如下：

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

或许大家很想知道，为什么在 y 中添加一个元素时，x 也会改变。

产生这种结果有两个因素：

1) 变量只是指向对象的一个名称。执行 `y = x` 并不会创建列表的副本——而只是创建了一个新变量 `y`，并指向 `x` 所指的同一对象。这就意味着只存在一个列表对象，`x` 和 `y` 都是对它的引用。

2) 列表属于*mutable* 对象，这意味着它的内容是可以修改的。

在调用 `append()` 之后，该可变对象的内容由 `[]` 变为 `[10]`。由于两个变量引用了同一对象，因此用其中任意一个名称所访问到的都是修改后的值 `[10]`。

如果把赋给 `x` 的对象换成一个不可变对象：

```
>>> x = 5  # 整数是不可变对象
>>> y = x
>>> x = x + 1  # 5 不能被修改，在此我们会新建一个对象
>>> x
6
>>> y
5
```

可见这时 `x` 和 `y` 就不再相等了。因为整数是*immutable* 对象，在执行 `x = x + 1` 时，并不会修改整数对象 `5`，给它加上 1；而是创建了一个新的对象（整数对象 `6` ）并将其赋给 `x`（也就是改变了 `x` 所指向的对

象)。在赋值完成后，就有了两个对象（整数对象 6 和 5 ）和分别指向他俩的两个变量（x 现在指向 6 而 y 仍然指向 5)。

某些操作 (例如 y.append(10) 和 y.sort()) 是改变原对象，而看上去相似的另一些操作 (例如 y = y + [10] 和 sorted(y) <sorted>) 则是创建新对象。通常在 Python 中 (以及在标准库的所有代码中) 会改变原对象的方法将返回 ``None`() 以帮助避免混淆这两种不同类型的操作。因此如果你错误地使用了 y.sort() 并期望它将返回一个经过排序的 y 的副本，你得到的结果将会是 None，这将导致你的程序产生一个容易诊断的错误。

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)。

换而言之：

- 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 和 b 是局部名称
...     b = b + 1              # 赋值为新的对象
...     return a, b            # 返回新的值
...
>>> 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' 引用了一个可变的列表
...     a[1] = a[1] + 1        # 修改一个共享对象
...
>>> 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 是一个可变的字典
...     args['b'] = args['b'] + 1   # 对其进行原地修改
...
>>> 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 是一个可变的 Namespace
...     args.b = args.b + 1         # 原地修改对象
...
>>> 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*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
```

或者可采用可调用对象：

```
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):
    # 继承了 __init__
    def __call__(self, x):
        return self.a * (x ** self.b)
```

对象可以为多个方法的运行状态进行封装：

```
class counter:

    value = 0

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

(续下页)

(接上页)

```
    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
```

以上 `inc()` 、`dec()` 和 `reset()` 的表现，就如同共享了同一计数变量一样。

2.2.12 如何复制 Python 对象？

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()
```

序列可以用切片操作进行复制：

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

2.2.13 如何找到对象的方法或属性？

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 如何用代码获取对象的名称？

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 在回答这样的问题时曾经给出过一个绝佳的类比：

> 这就像要知道家门口的那只猫的名字一样：猫（对象）自己不会说出它的名字，它根本就不在乎自己叫什么——所以唯一方法就是问一遍你所有的邻居（命名空间），这是不是他们家的猫（对象）……
>
> ……并且如果你发现它有很多名字或根本没有名字，那也不必惊讶！

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 是否提供等价于 C 语言"?:" 三目运算符的东西？

有的。语法如下：

```
[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

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

# 前 10 个斐波那契数字
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))))

# 曼德布罗集
print((lambda Ru,Ro,Iu,Io,IM,Sx,Sy:reduce(lambda x,y:x+'\n'+y,map(lambda y,
Iu=Iu,Io=Io,Ru=Ru,Ro=Ro,Sy=Sy,L=lambda yc,Iu=Iu,Io=Io,Ru=Ru,Ro=Ro,i=IM,
Sx=Sx,Sy=Sy:reduce(lambda x,y:x+y,map(lambda x,xc=Ru,yc=yc,Ru=Ru,Ro=Ro,
i=i,Sx=Sx,F=lambda xc,yc,x,y,k,f=lambda xc,yc,x,y,k,f:(k<=0)or (x*x+y*y
>=4.0) or 1+f(xc,yc,x*x-y*y+xc,2.0*x*y+yc,k-1,f):f(xc,yc,x,y,k,f):chr(
64+F(Ru+x*(Ro-Ru)/Sx,yc,0,0,i)),range(Sx))):L(Iu+y*(Io-Iu)/Sy),range(Sy
))))(-2.1, 0.7, -1.2, 1.2, 30, 80, 24))
#     \___ ___/  \___ ___/  |   |   |__ 屏幕上的行
#         V          V      |   |______ 屏幕上的列
#         |          |      |_________ “迭代”的最大次数
#         |          |________________ y 轴上的取值范围
#         |___________________________ x 轴上的取值范围
```

请不要在家里尝试，骚年！

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 如何给出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 为什么 -22 // 10 会返回 -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 如何将字符串转换为数字？

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 如何将数字转换为字符串？

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

(续下页)

(接上页)

```
>>> print(a)
array('w', '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}  # 注意函数名后不带圆括号

dispatch[get_input()]()  # 注意末尾要带圆括号以调用函数
```

- 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 Perl 的 `chomp()` 等价物用于从字符串中移除末尾换行符？

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() 的等价物？

没有。

对于简单的输入解析，最简单的方法通常是使用字符串对象的 `split()` 方法将行分割为空白符分隔的单词，然后使用 `int()` 或 `float()` 将十进制字符串转换为数字值。`split()` 支持可选的"sep" 形参，如果行中使用空白符以外的其他分隔符，可以使用该参数。

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 错误的含义是什么？

见 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导致代码回退。

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 sortinghowto ）。
- 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。

> **参见**
>
> 专门介绍 性能提示 的 wiki 页面。

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):
    ...  # 对 x 执行某些操作 ...
```

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* 是一种优雅的解决方案:

```
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]
```

这相当于：

```
>>> 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 对象

2.6.1 什么是类？

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 什么是方法？

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。

另请参阅*为什么必须在方法定义和调用中显式使用“self”？*。

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)        # 直接
True
>>> isinstance(c, P)        # 间接
True
>>> isinstance(c, Mapping)  # 虚拟
True

# 实际的继承链
>>> type(c).__mro__
(<class 'C'>, <class 'P'>, <class 'object'>)

# 测试“真正的继承”
>>> 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):
        ...  # 搜索邮箱的代码
    elif isinstance(obj, Document):
        ...  # 搜索文档的代码
    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):
        ...  # 搜索邮箱的代码

class Document:
    def search(self):
        ...  # 搜索文档的代码

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
    ...
```

许多 __setattr__() 实现都会调用 object.__setattr__() 在 self 上设置属性，而不会导致无限递归：

```
class X:
    def __setattr__(self, name, value):
        # 这里添加自定义的逻辑...
        object.__setattr__(self, name, value)
```

另外，也可以通过直接在 self.__dict__ 中插入条目来设置属性。

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()  # 调用 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
```

(续下页)

(接上页)

```
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   # C.__init__ 被调用的次数

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

    def getcount(self):
        return C.count  # 或返回 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):
        # 没有 'self' 形参！
        ...
```

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")
```

(续下页)

(接上页)

```
        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。

标识符可以在类的内部不加改变地使用，但要在类的外部访问它，就必须使用被混淆的名称：

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

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

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

需要特别指出，这并不能保证私密性因为外部用户仍然可以有意地访问私有属性；许多 Python 程序员根本就不屑于使用私有变量名。

参见

私有名称调整规范说明了解相关详情和特例。

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()` 应当确保被可以被同一对象多次调用。

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) 赋值操作将创建新的名称但不会改变对象标识号。在赋值操作 `new = old` 之后，可以保证 `new is old`。

2) 将对象放入存储对象引用的容器不会改变对象的标识号。在列表赋值操作 `s[0] = x` 之后，将可保证 `s[0] is x`。

3) 如果一个对象是单例，则意味着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** 所建议的，标识测试是检查 None 的推荐方式。这样的代码读起来就像直白的英语并可避免与具有结果为假的布尔值的对象相混淆。

2) 当 None 是一个有效的输入值时检查可选参数会有点麻烦。在这些情况下，你可以创建一个保证与其他对象不同的单例哨兵对象。例如，以下代码演示了如何实现一个行为与 dict.pop() 类似的方法：

```
_sentinel = object()

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

3) 容器的实现有时需要用标识测试来增强相等性测试。这样可以防止代码被 float('NaN') 这类不等于自身的对象所干扰。

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 一个子类如何控制哪些数据被存储在一个不可变的实例中？

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)
```

这些类可以这样使用:

```
>>> 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其内被保留。

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 方式适用于具有*hashable* 参数的方法。它会创建对实例的引用，除非特别设置了传入弱引用。

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"
        # 取决于 station_id、date 和 unit
```

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
```

(续下页)

(接上页)

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

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

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

    @lru_cache(maxsize=20)
    def historic_rainfall(self, date, units='cm'):
        'Rainfall on a given date'
        # 取决于 station_id、date 和 unit
```

2.7 模块

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...')
```

(续下页)

(接上页)

```
    ...

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
- 编译 bar 并开始执行
- 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。

这个问题有（至少）三种可能的解决方法。

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 建议每个模块都应遵循以下顺序：

- 导出（全局变量、函数和不需要导入基类的类）
- import 语句
- 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。

这些解决方案并不相互排斥。

2.7.4 __import__('x.y.z') 返回的是 <module 'x'>；该如何得到 z 呢？

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'
```

CHAPTER 3

设计和历史常见问题

3.1 为什么 Python 使用缩进来分组语句？

Guido van Rossum 认为使用缩进进行分组非常优雅，并且大大提高了普通 Python 程序的清晰度。大多数人在一段时间后就会喜欢上这个特性。

由于没有开始/结束括号，因此解析器感知的分组与人类读者之间不会存在分歧。偶尔 C 程序员会遇到像这样的代码片段:

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

如果条件为真，则只执行 `x++` 语句，但缩进会使你认为情况并非如此。即使是经验丰富的 C 程序员有时也会长久地盯着它发呆，不明白为什么在 `x > y` 时 `y` 也会减少。

因为没有开始/结束花括号，所以 Python 更不容易发生代码风格冲突。在 C 中有许多不同的放置花括号的方式。在习惯了阅读和编写某种特定风格的代码之后，当阅读（或被要求编写）另一种风格的代码时通常都会令人感觉有点不舒服）。

许多编码风格将开始/结束括号单独放在一行上。这使得程序相当长，浪费了宝贵的屏幕空间，使得更难以对程序进行全面的了解。理想情况下，函数应该适合一个屏幕（例如，20--30 行）。20 行 Python 可以完成比 20 行 C 更多的工作。这不仅仅是由于没有开始/结束括号 -- 无需声明以及高层级的数据类型也是其中的原因 -- 但基于缩进的语法肯定有帮助。

3.2 为什么简单的算术运算得到奇怪的结果？

请看下一个问题。

3.3 为什么浮点计算不准确？

用户经常对这样的结果感到惊讶:

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

并且认为这是 Python 中的一个 bug。其实不是这样。这与 Python 关系不大，而与底层平台如何处理浮点数字关系更大。

CPython 中的 `float` 类型使用 C 语言的 `double` 类型进行存储。`float` 对象的值是以固定的精度（通常为 53 位）存储的二进制浮点数，由于 Python 使用 C 操作，而后者依赖于处理器中的硬件实现来执行浮点运算。这意味着就浮点运算而言，Python 的行为类似于许多流行的语言，包括 C 和 Java。

许多可以轻松地用十进制表示的数字不能用二进制浮点表示。例如，在输入以下语句后:

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

为 `x` 存储的值是与十进制的值 `1.2` (非常接近) 的近似值，但不完全等于它。在典型的机器上，实际存储的值是:

```
1.0011001100110011001100110011001100110011001100110011 (二进制)
```

它对应于十进制数值:

```
1.1999999999999999555910790149937383830547332763671875 (十进制)
```

典型的 53 位精度为 Python 浮点数提供了 15-16 位小数的精度。

要获得更完整的解释，请参阅 Python 教程中的 浮点算术一章。

3.4 为什么 Python 字符串是不可变的？

有几个优点。

一个是性能：知道字符串是不可变的，意味着我们可以在创建时为它分配空间，并且存储需求是固定不变的。这也是元组和列表之间区别的原因之一。

另一个优点是，Python 中的字符串被视为与数字一样“基本”。任何动作都不会将值 8 更改为其他值，在 Python 中，任何动作都不会将字符串”8” 更改为其他值。

3.5 为什么必须在方法定义和调用中显式使用“self”？

这个想法借鉴了 Modula-3 语言。出于多种原因它被证明是非常有用的。

首先，更明显的显示出，使用的是方法或实例属性而不是局部变量。阅读 `self.x` 或 `self.meth()` 可以清楚地表明，即使您不知道类的定义，也会使用实例变量或方法。在 C++ 中，可以通过缺少局部变量声明来判断（假设全局变量很少见或容易识别）——但是在 Python 中没有局部变量声明，所以必须查找类定义才能确定。一些 C++ 和 Java 编码标准要求实例属性具有 `m_` 前缀，因此这种显式性在这些语言中仍然有用。

其次，这意味着当要显式引用或从特定类调用该方法时无须特殊语法。在 C++ 中，如果你想要使用在派生类中被重写的基类方法，你必须使用 `::` 运算符 -- 在 Python 中你可以写成 `baseclass.methodname(self, <argument list>)`。这特别适用于 `__init__()` 方法，并且也适用于派生类方法想要扩展同名的基类方式因而必须以某种方式调用基类方法的情况。

最后，它解决了变量赋值的语法问题：为了 Python 中的局部变量（根据定义!）在函数体中赋值的那些变量（并且没有明确声明为全局）赋值，就必须以某种方式告诉解释器一个赋值是为了分配一个实例变量而不是一个局部变量，它最好是通过语法实现的（出于效率原因）。C++ 通过声明来做到这一点，但是 Python 没有声明，仅仅为了这个目的而引入它们会很可惜。使用显式的 `self.var` 很好地解决了这个问题。类似地，对于使用实例变量，必须编写 `self.var` 意味着对方法内部的非限定名称的引用不必搜索实例的目录。换句话说，局部变量和实例变量存在于两个不同的命名空间中，您需要告诉 Python 使用哪个命名空间。

3.6 为什么不能在表达式中赋值？

自 Python 3.8 开始，你能做到的！

赋值表达式使用海象运算符 := 在表达式中为变量赋值:

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

请参阅 **PEP 572** 了解详情。

3.7 为什么 Python 对某些功能（例如 list.index()）使用方法来实现，而其他功能（例如 len(List)）使用函数实现？

正如 Guido 所说:

> (a) 对于某些操作，前缀表示法比后缀更容易阅读 -- 前缀（和中缀！）运算在数学中有着悠久的传统，就像在视觉上帮助数学家思考问题的记法。比较一下我们将 x*(a+b) 这样的公式改写为 x*a+x*b 的容易程度，以及使用原始 OO 符号做相同事情的笨拙程度。
>
> (b) 当读到写有 len(X) 的代码时，就知道它要求的是某件东西的长度。这告诉我们两件事：结果是一个整数，参数是某种容器。相反，当阅读 x.len() 时，必须已经知道 x 是某种实现接口的容器，或者是从具有标准 len() 的类继承的容器。当没有实现映射的类有 get() 或 key() 方法，或者不是文件的类有 write() 方法时，我们偶尔会感到困惑。
>
> —https://mail.python.org/pipermail/python-3000/2006-November/004643.html

3.8 为什么 join() 是一个字符串方法而不是列表或元组方法？

从 Python 1.6 开始，字符串变得更像其他标准类型，当添加方法时，这些方法提供的功能与始终使用 String 模块的函数时提供的功能相同。这些新方法中的大多数已被广泛接受，但似乎让一些程序员感到不舒服的一种方法是:

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

结果如下:

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

反对这种用法有两个常见的论点。

第一条是这样的：“使用字符串文本 (String Constant) 的方法看起来真的很难看”，答案是也许吧，但是字符串文本只是一个固定值。如果在绑定到字符串的名称上允许使用这些方法，则没有逻辑上的理由使其在文字上不可用。

第二个异议通常是这样的：“我实际上是在告诉序列使用字符串常量将其成员连接在一起”。遗憾的是并非如此。出于某种原因，把 `split()` 作为一个字符串方法似乎要容易得多，因为在这种情况下，很容易看到:

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

是对字符串文本的指令，用于返回由给定分隔符分隔的子字符串（或在默认情况下，返回任意空格）。

`join()` 是字符串方法，因为在使用该方法时，您告诉分隔符字符串去迭代一个字符串序列，并在相邻元素之间插入自身。此方法的参数可以是任何遵循序列规则的对象，包括您自己定义的任何新的类。对于字节和字节数组对象也有类似的方法。

3.9 异常有多快？

如果没有引发异常则 `try`/`except` 代码块是非常高效的。实际上捕获异常是很消耗性能的。在 2.0 之前的 Python 版本中通常使用这例程:

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

只有当你期望 dict 在任何时候都有 key 时，这才有意义。如果不是这样的话，你就是应该这样编码:

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

对于这种特定的情况，您还可以使用 `value = dict.setdefault(key, getvalue(key))`，但前提是调用 `getvalue()` 足够便宜，因为在所有情况下都会对其进行评估。

3.10 为什么 Python 中没有 switch 或 case 语句？

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`。

对于需要从大量可能性中进行选择的情况，可以创建一个字典，将 case 值映射到要调用的函数。例如:

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

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

对于对象调用方法，可以通过使用 `getattr()` 内置检索具有特定名称的方法来进一步简化:

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

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

建议对方法名使用前缀，例如本例中的 `visit_`。如果没有这样的前缀，如果值来自不受信任的源，攻击者将能够调用对象上的任何方法。

模仿带有穿透方式的分支，就像 C 的 switch-case-default 那样是有可能的，但更为困难，也无甚必要。

3.11 难道不能在解释器中模拟线程，而非得依赖特定于操作系统的线程实现吗？

答案 1: 不幸的是，解释器为每个 Python 堆栈帧推送至少一个 C 堆栈帧。此外，扩展可以随时回调 Python。因此，一个完整的线程实现需要对 C 的线程支持。

答案 2: 幸运的是，Stackless Python 有一个完全重新设计的解释器循环，可以避免 C 堆栈。

3.12 为什么 lambda 表达式不能包含语句？

Python 的 lambda 表达式不能包含语句，因为 Python 的语法框架不能处理嵌套在表达式内部的语句。然而，在 Python 中，这并不是一个严重的问题。与其他语言中添加功能的 lambda 形式不同，Python 的 lambda 只是一种速记符号，如果您懒得定义函数的话。

函数已经是 Python 中的第一等对象，并且可以在局部作用域中声明。因此使用 lambda 而非局部定义函数的唯一优点是你不需要为函数指定名称 -- 但那只是一个被赋值为函数对象（它的类型与 lambda 表达式所产生的对象完全相同）的局部变量！

3.13 可以将 Python 编译为机器代码，C 或其他语言吗？

Cython 会将带有可选标注的修改版 Python 编译为 C 扩展。Nuitka 是一个 Python 转 C++ 代码的新兴编译器，其目标是支持完整的 Python 语言。

3.14 Python 如何管理内存？

Python 内存管理的细节取决于实现。Python 的标准实现*CPython* 使用引用计数来检测不可访问的对象，并使用另一种机制来收集引用循环，定期执行循环检测算法来查找不可访问的循环并删除所涉及的对象。`gc` 模块提供了执行垃圾回收、获取调试统计信息和优化收集器参数的函数。

不过，其他实现 (如 Jython 或 PyPy)，可能会依赖不同的机制，如完全的垃圾回收器。如果你的 Python 代码依赖于引用计数实现的行为则这种差异可能会导致某些微妙的移植问题。

在一些 Python 实现中，以下代码（在 CPython 中工作的很好）可能会耗尽文件描述符:

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

实际上，使用 CPython 的引用计数或器方案，每次对 `f` 的新赋值都会关闭之前的文件。然而，对于传统的 GC，这些文件对象将只能以不同的并且可能很长的间隔被收集（和关闭）。

如果要编写可用于任何 python 实现的代码，则应显式关闭该文件或使用 `with` 语句；无论内存管理方案如何，这都有效:

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

3.15 为什么 CPython 不使用更传统的垃圾回收方案？

首先，这不是 C 标准特性，因此不能移植。(是的，我们知道 Boehm GC 库。它包含了大多数常见平台（但不是所有平台）的汇编代码，尽管它基本上是透明的，但也不是完全透明的; 要让 Python 使用它，需要使用补丁。)

当 Python 嵌入到其他应用程序中时传统的 GC 也会成为一个问题。在独立的 Python 中可以用 GC 库提供的版本来替换标准的 `malloc()` 和 `free()`，而嵌入 Python 的应用程序可能想要自行替代 `malloc()` 和 `free()`，并不想要 Python 的版本。现在，CPython 可以适用于任何正确实现了 `malloc()` 和 `free()` 的版本。

3.16 CPython 退出时为什么不释放所有内存？

当 Python 退出时，从全局命名空间或 Python 模块引用的对象并不总是被释放。如果存在循环引用，则可能发生这种情况 C 库分配的某些内存也是不可能释放的（例如像 Purify 这样的工具会抱怨这些内容）。但是，Python 在退出时清理内存并尝试销毁每个对象。

如果要强制 Python 在释放时删除某些内容，请使用 `atexit` 模块运行一个函数，强制删除这些内容。

3.17 为什么有单独的元组和列表数据类型？

列表和元组虽然在许多方式都很相似，但它们的使用方式有本质上的不同。元组可被当作是类似于 Pascal `records` 或 C `structs`；它们是由可能具有不同类型但可作为一个分组进行操作的相关数据组成的小多项集。例如，一个笛卡尔坐标可适当地用由两个或三个数字组成的元组来表示。

另一方面，列表更像其他语言中的数组。它们倾向于保存可变数量的全都具有相同类型并将被逐个操作的对象。例如，`os.listdir('.')` 返回一个代表当前目录中文件的字符串列表。当你向该目录添加一两个文件时在此输出上执行操作的函数通常不会中断。

元组是不可变的，这意味着一旦创建了元组，就不能用新值替换它的任何元素。列表是可变的，这意味着您始终可以更改列表的元素。只有不变元素可以用作字典的 key，因此只能将元组和非列表用作 key。

3.18 列表是如何在 CPython 中实现的？

CPython 的列表实际上是可变长度的数组，而不是 lisp 风格的链表。该实现使用对其他对象的引用的连续数组，并在列表头结构中保留指向该数组和数组长度的指针。

这使得索引列表 `a[i]` 的操作成本与列表的大小或索引的值无关。

当添加或插入项时，将调整引用数组的大小。并采用了一些巧妙的方法来提高重复添加项的性能; 当数组必须增长时，会分配一些额外的空间，以便在接下来的几次中不需要实际调整大小。

3.19 字典是如何在 CPython 中实现的？

CPython 的字典实现为可调整大小的哈希表。与 B-树相比，这在大多数情况下为查找（目前最常见的操作）提供了更好的性能，并且实现更简单。

字典的工作方式是使用 `hash()` 内置函数计算字典中存储的每个键的哈希码。哈希码会根据键和基于进程的种子值而大幅改变；例如，`'Python'` 的哈希码可能为 `-539294296` 而 `'python'` 这个只相差一丁点的字符串的哈希码却可能为 `1142331976`。随后哈希码将被用来计算在一个内部数组中相应值的存储位置。假设你存储的键都具有不同的哈希值，这意味着字典会耗费恒定的时间 -- 即大 O 表示法的 $O(1)$ -- 要检索一个键。

3.20 为什么字典 key 必须是不可变的？

字典的哈希表实现使用从键值计算的哈希值来查找键。如果键是可变对象，则其值可能会发生变化，因此其哈希值也会发生变化。但是，由于无论谁更改键对象都无法判断它是否被用作字典键值，因此无法在字典中修改条目。然后，当你尝试在字典中查找相同的对象时，将无法找到它，因为其哈希值不同。如果你尝试查找旧值，也不会找到它，因为在该哈希表中找到的对象的值会有所不同。

如果你想要一个用列表索引的字典，只需先将列表转换为元组；用函数 `tuple(L)` 创建一个元组，其条目与列表 `L` 相同。元组是不可变的，因此可以用作字典键。

已经提出的一些不可接受的解决方案：

- 哈希按其地址（对象 ID）列出。这不起作用，因为如果你构造一个具有相同值的新列表，它将无法找到；例如:

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

  会引发一个 `KeyError` 异常，因为第二行中使用的 `[1, 2]` 的 id 与第一行中的 id 不同。换句话说，应该使用 `==` 来比较字典键，而不是使用 `is` 。

- 使用列表作为键时进行复制。这没有用的，因为作为可变对象的列表可以包含对自身的引用，然后复制代码将进入无限循环。

- 允许列表作为键，但告诉用户不要修改它们。当你意外忘记或修改列表时，这将产生程序中的一类难以跟踪的错误。它还使一个重要的字典不变量无效：`d.keys()` 中的每个值都可用作字典的键。

- 将列表用作字典键后，应标记为其只读。问题是，它不仅仅是可以改变其值的顶级对象；你可以使用包含列表作为键的元组。将任何内容作为键关联到字典中都需要将从那里可到达的所有对象标记为只读——并且自引用对象可能会导致无限循环。

如果你有需要，以下技巧可以绕过这个问题，但使用它必须自担风险：你可以将一个可变结构体包装在一个同时具有 `__eq__()` 和 `__hash__()` 方法的类实例中。然后你必须确保存放在字典（或其他基于哈希值的结构体）中的所有此类包装器对象的哈希值在该字典（或其他结构体）中保持固定。

```
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
```

注意，哈希计算由于列表的某些成员可能不可用以及算术溢出的可能性而变得复杂。

此外，必须始终如此，如果 `o1 == o2`（即 `o1.__eq__(o2) is True`）则 `hash(o1) == hash(o2)`（即 `o1.__hash__() == o2.__hash__()`），无论对象是否在字典中。如果你不能满足这些限制，字典和其他基于 hash 的结构将会出错。

对于 `ListWrapper` 的情况，只要包装器位于字典中那么被包装的列表就不能更改以避免发生意外。除非你准备好认真考虑相关要求以及未能正确满足这些要求的后果否则请不要这样做。你已经收到警告了。

3.21 为什么 list.sort() 没有返回排序列表？

在性能很重要的情况下，仅仅为了排序而复制一份列表将是一种浪费。因此，`list.sort()` 对列表进行了适当的排序。为了提醒您这一事实，它不会返回已排序的列表。这样，当您需要排序的副本，但也需要保留未排序的版本时，就不会意外地覆盖列表。

如果要返回新列表，请使用内置 `sorted()` 函数。此函数从提供的可迭代列表中创建新列表，对其进行排序并返回。例如，下面是如何迭代遍历字典并按 keys 排序:

```
for key in sorted(mydict):
    ...  # 对 mydict[key] 做点什么
```

3.22 如何在 Python 中指定和实施接口规范？

由 C++ 和 Java 等语言提供的模块接口规范描述了模块的方法和函数的原型。许多人认为接口规范的编译时强制执行有助于构建大型程序。

Python 2.6 添加了一个 `abc` 模块，允许定义抽象基类 (ABCs)。然后可以使用 `isinstance()` 和 `issubclass()` 来检查实例或类是否实现了特定的 ABC。`collections.abc` 模块定义了一组有用的 ABCs 例如 `Iterable`，`Container`，和 `MutableMapping`

对于 Python，接口规范的许多好处可以通过组件的适当测试规程来获得。

一个好的模块测试套件既可以提供回归测试，也可以作为模块接口规范和一组示例。许多 Python 模块可以作为脚本运行，以提供简单的“自我测试”。即使是使用复杂外部接口的模块，也常常可以使用外部接口的简单“桩代码（stub）”模拟进行隔离测试。可以使用 `doctest` 和 `unittest` 模块或第三方测试框架来构造详尽的测试套件，以运行模块中的每一行代码。

适当的测试规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。

编写测试套件非常有用，并且你可能希望将你的代码设计为易于测试。一种日益流行的技术是面向测试的开发，它要求在编写任何实际代码之前首先编写测试套件的各个部分。当然 Python 也允许你采用更粗率的方式，不必编写任何测试用例。

3.23 为什么没有 goto？

在 1970 年代人们了解到不受限制的 goto 可能导致混乱得像“意大利面”那样难以理解和修改的代码。在高级语言中，它也是不必要的，只需能够执行分支（在 Python 中是使用 `if` 语句和 `or`, `and` 以及 `if`/`else` 表达式）和循环（使用 `while` 和 `for` 语句，还可能包含 `continue` 和 `break` 语句）就足够了。

人们还可以使用异常来提供甚至能跨函数调用的“结构化 goto”。许多人觉得异常可以方便地模拟 C, Fortran 和其他语言中所有合理使用的 `go` 或 `goto` 结构:

```
class label(Exception): pass  # 声明label

try:
    ...
    if condition: raise label()  # 跳转到 label
    ...
except label:  # 跳转到哪里
    pass
...
```

这并不允许跳到一个循环的中间，但这通常被视为是对 `goto` 的滥用。应当谨慎使用。

3.24 为什么原始字符串（r-strings）不能以反斜杠结尾？

更准确地说，它们不能以奇数个反斜杠结束：结尾处的不成对反斜杠会转义结束引号字符，留下未结束的字符串。

原始字符串的设计是为了方便想要执行自己的反斜杠转义处理的处理器 (主要是正则表达式引擎) 创建输入。此类处理器将不匹配的尾随反斜杠视为错误，因此原始字符串不允许这样做。反过来，允许通过使用引号字符转义反斜杠转义字符串。当 r-string 用于它们的预期目的时，这些规则工作的很好。

如果您正在尝试构建 Windows 路径名，请注意所有 Windows 系统调用都使用正斜杠:

```
f = open("/mydir/file.txt")  # 效果很好!
```

如果您正在尝试为 DOS 命令构建路径名，请尝试以下示例

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

3.25 为什么 Python 没有属性赋值的“with”语句？

Python 有一种 `with` 语句能将一个代码块的执行包装起来，在进入和退出该代码块时调用特定的代码。某些语言具有类似这样的结构:

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

在 Python 中，这样的结构是不明确的。

其他语言，如 ObjectPascal、Delphi 和 C++ 使用静态类型，因此可以毫不含糊地知道分配给什么成员。这是静态类型的要点 -- 编译器 总是在编译时知道每个变量的作用域。

Python 使用动态类型。事先不可能知道在运行时引用哪个属性。可以动态地在对象中添加或删除成员属性。这使得无法通过简单的阅读就知道引用的是什么属性：局部属性、全局属性还是成员属性？

例如，采用以下不完整的代码段:

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

该代码段假设 `a` 必须有一个名为 `x` 的成员属性。然而，Python 中没有什么能告诉解释器这一点。举例来说，如果 `a` 是一个整数那么会发生什么？如果有一个名为 `x` 的全局变量，它是否会在 `with` 代码块内被使用？如你所见，Python 的动态特性使得这样的选择更为困难。

然而，`with` 及类似语言特性的主要好处（减少代码量）在 Python 中可以通过赋值轻松地实现。而不是使用:

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

写成这样:

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

这也具有提高执行速度的附带效果，因为 Python 在运行时解析名称绑定，而第二个版本只需要执行一次解析。

引入可以进一步减小代码量的类似提议，例如使用 “前导点号”，出于明白胜于隐晦的理由而被拒绝了 (参见 https://mail.python.org/pipermail/python-ideas/2016-May/040070.html)。

3.26 生成器为什么不支持 with 语句？

由于技术原因，直接作为上下文管理器使用的生成器将无法正常工作。最常见的情况下，当一个生成器被用作迭代器运行到完成时，不需要手动关闭。如果需要，请在 `with` 语句中将它包装为 `contextlib.closing(generator)`。

3.27 为什么 if/while/def/class 语句需要冒号？

冒号主要用于增强可读性 (ABC 语言实验的结果之一)。考虑一下这个:

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

与

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

注意第二种方法稍微容易一些。请进一步注意，在这个 FAQ 解答的示例中，冒号是如何设置的；这是英语中的标准用法。

另一个次要原因是冒号使带有语法突出显示的编辑器更容易工作；他们可以寻找冒号来决定何时需要增加缩进，而不必对程序文本进行更精细的解析。

3.28 为什么 Python 在列表和元组的末尾允许使用逗号？

Python 允许您在列表，元组和字典的末尾添加一个尾随逗号:

```
[1, 2, 3,]
('a', 'b', 'c',)
d = {
    "A":[1, 5],
    "B":[6, 7], # 最后的逗号是可选的，但风格很好
}
```

有几个理由允许这样做。

如果列表，元组或字典的字面值分布在多行中，则更容易添加更多元素，因为不必记住在上一行中添加逗号。这些行也可以重新排序，而不会产生语法错误。

不小心省略逗号会导致难以诊断的错误。例如:

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

这个列表看起来有四个元素，但实际上包含三个: ”fee”, ”fiefoo” 和”fum” 。总是加上逗号可以避免这个错误的来源。

允许尾随逗号也可以使编程代码更容易生成。

CHAPTER 4

代码库和插件 FAQ

4.1 通用的代码库问题

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 math.py（socket.py，regex.py 等）的源文件在哪？

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（但不在 Python 目录中）。

Python 中（至少）有三类模块：

1) 使用 Python 编写的模块（.py）；
2) 使用 C 编写的动态加载模块（.dll，.pyd，.so，.sl 等）；
3) 使用 C 编写并链接到解释器的模块，要获取此列表，输入：

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

4.1.3 在 Unix 中怎样让 Python 脚本可执行？

你需要做两件事：文件必须是可执行的，并且第一行需要以 `#!` 开头，后面跟上 Python 解释器的路径。

第一点可以用执行 `chmod +x scriptfile` 或是 `chmod 755 scriptfile` 做到。

第二点有很多种做法，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 解释器的路径。

如果你希望脚本不依赖 Python 解释器的具体路径，你也可以使用 **env** 程序。假设你的 Python 解释器所在目录已经添加到了 `PATH` 环境变量中，几乎所有的类 Unix 系统都支持下面的写法：

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

不要在 CGI 脚本中这样做。CGI 脚本的 PATH 环境变量通常会非常精简，所以你必须使用解释器的完整绝对路径。

有时候，用户的环境变量如果太长，可能会导致 **/usr/bin/env** 执行失败；又或者甚至根本就不存在 env 程序。在这种情况下，你可以尝试使用下面的 hack 方法（来自 Alex Rezinsky）：

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

这样做有一个小小的缺点，它会定义脚本的 __doc__ 字符串。不过可以这样修复：

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

4.1.4 Python 中有 curses/termcap 包吗？

对于类 Unix 系统：标准 Python 源码发行版会在 Modules 子目录中附带 curses 模块，但默认并不会编译。（注意：在 Windows 平台下不可用——Windows 中没有 curses 模块。）

curses 模块支持基本的 curses 特性，同时也支持 ncurses 和 SYSV curses 中的很多额外功能，比如颜色、不同的字符集支持、填充和鼠标支持。这意味着这个模块不兼容只有 BSD curses 模块的操作系统，但是目前仍在维护的系统应该都不会存在这种情况。

4.1.5 Python 中存在类似 C 的 onexit() 函数的东西吗？

atexit 模块提供了一个与 C 的 onexit() 类似的注册函数。

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 通用任务

4.2.1 怎样测试 Python 程序或组件？

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 线程相关

4.3.1 程序中怎样使用线程？

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 我的线程都没有运行，为什么？

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))
```

(续下页)

(接上页)

```
    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` 类维护一个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

# 工作线程会将任务移出队列。 当队列为空时，
# 它将认为工作已完成并退出。
# （在真实场景下工作线程将持续运行直到被终结。）
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)

# 创建队列
q = queue.Queue()

# 启动包含 5 个工作线程的线程池
for i in range(5):
    t = threading.Thread(target=worker, name='worker %i' % (i+1))
    t.start()

# 开始向队列添加任务
for i in range(50):
    q.put(i)

# 为线程留出运行的时间
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
```

(续下页)

(接上页)

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

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 不能删除全局解释器锁吗？

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 **PEP 703** 通过后目前已着手从 Python 的 CPython 实现中移除 GIL。最初它将它将作为构建解释器时的可选编译器旗标来实现，因此将会存在有 GIL 和没有 GIL 的构建版本。从长远来看，目标是在移除 GIL 对性能的影响被完全了解之后确定唯一的构建版本。Python 3.13 大概是第一个包含此项工作的发布版，尽管在这个发布版中该功能可能尚不完整。

当前移除 GIL 的工作是基于 Sam Gross 的 移除了 GIL 的 Python 3.9 分叉。更早的时候，在 Python 1.5 时期，Greg Stein 实际上实现了一个完整的补丁集（“自由线程”补丁），移除了 GIL 并用更细粒度的锁来代替。Adam Olsen 也在他的 python-safethread 项目里做了类似的实验。不幸的是，由于为移除 GIL 而使用了大量细粒度的锁这两个早期实验在单线程中的性能都有明显的下降（至少慢 30%）。Python 3.9 分叉是在移除 GIL 的同时保持可接受的性能影响的首次尝试。

当前 Python 发布版存在 GIL 并不意味着你无法在多 CPU 机器上很好地使用 Python！你仅需发挥创造性将任务在多个 进程 而不是多个 *threads* 线程之间进行分配。新的 `concurrent.futures` 模块中的

`ProcessPoolExecutor` 类提供了完成此项工作的简单方式；如果你想要对任务分发有更强的控制那么 `multiprocessing` 模块提供了更低层级的 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。

减小 GIL 的影响的一种替代方式是让 GIL 成为每解释器状态锁而不是真正的全局状态锁。此特性 在 Python 3.12 中首次实现并在 C API 中可用。预期会在 Python 3.13 中提供它的 Python 接口。目前它的主要限制在于第 3 方扩展模块，因此这些模块的编写必须考虑到多解释器的情况才能够被使用，这样大量较旧的扩展模块将不再可用。

4.4 输入与输出

4.4.1 怎样删除文件？(以及其他文件相关的问题……)

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()`。

重命名文件可以使用 `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 怎样复制文件？

`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 怎样读取（或写入）二进制数据？

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(">hhl", 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‘l’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。

> **备注**
>
> 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 怎样访问（RS232）串口？

对于 Win32, OSX, Linux, BSD, Jython, IronPython:

pyserial

对于 Unix，查看 Mitch Chapman 发布的帖子：

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 网络 / Internet 编程

4.5.1 Python 中的 WWW 工具是什么？

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 。

4.5.2 生成 HTML 需要使用什么模块？

你可以在 Web 编程 wiki 页面 找到许多有用的链接。

4.5.3 怎样使用 Python 脚本发送邮件？

使用 `smtplib` 标准库模块。

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()
```

(续下页)

(接上页)

```
    if not line:
        break
    msg += line

# 实际发送的邮件
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 的位置
p = os.popen("%s -t -i" % SENDMAIL, "w")
p.write("To: receiver@example.com\n")
p.write("Subject: test\n")
p.write("\n")  # 分隔标头和消息体的空白行
p.write("Some text\n")
p.write("some more text\n")
sts = p.close()
if sts != 0:
    print("Sendmail exit status", sts)
```

4.5.4 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()` 来检查它只否可写。

> **备注**
>
> `asyncio` 模块提供了通用的单线程并发异步库，它可被用来编写非阻塞的网络代码。第三方的 Twisted 库是一个热门且功能丰富的替代选择。

4.6 数据库

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 数学和数字

4.7.1 Python 中怎样生成随机数？

random 标准库模块实现了随机数生成器，使用起来非常简单：

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

这将返回一个 [0, 1) 区间的随机浮点数。

该模块中还有许多其他的专门的生成器，例如：

- randrange(a, b) 返回 [a, b) 区间内的一个整型数。
- uniform(a, b) 在 [a, b) 区间选择一个浮点数。
- normalvariate(mean, sdev) 使用正态（高斯）分布采样。

还有一些高级函数直接对序列进行操作，例如：

- choice(S) 从给定的序列中随机选择一个元素。
- shuffle(L) 会对列表执行原地重排，即将其随机地打乱。

还有 Random 类，你可以将其实例化，用来创建多个独立的随机数生成器。

CHAPTER 5

扩展/嵌入常见问题

5.1 可以使用 C 语言创建自己的函数吗？

是的，您可以在 C 中创建包含函数、变量、异常甚至新类型的内置模块。在文档 extending-index 中有说明。

大多数中级或高级的 Python 书籍也涵盖这个主题。

5.2 可以使用 C++ 语言创建自己的函数吗？

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 C 很难写，有没有其他选择？

有多个替代选择可用来编写你自己的 C 扩展，具体取决于你想要做什么。推荐的第三方工具提供了更简单或更复杂的方式来为 Python 创建 C 和 C++ 扩展。

5.4 如何在 C 中执行任意 Python 语句？

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 如何从 Python 对象中提取 C 的值？

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()`。

要检测一个对象的类型，首先要确保它不为 `NULL`，然后使用 `PyBytes_Check()`, `PyTuple_Check()`, `PyList_Check()` 等等。

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() 创建任意长度的元组？

不可以。应该使用 `PyTuple_Pack()` 。

5.8 如何从 C 调用对象的方法？

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。

例如调用某个文件对象的”seek” 方法并传入参数 10, 0 (假定文件对象的指针为”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 如何捕获 PyErr_Print()（或打印到 stdout / stderr 的任何内容）的输出？

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。

做到这一点的最简单方式是使用 `io.StringIO` 类:

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

(续下页)

(接上页)

```
>>> 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 如何从 C 访问用 Python 编写的模块？

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++ 并不没有太大的区别——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。

有关 C ++ 库，请参阅*C 很难写，有没有其他选择？*

5.12 我使用 Setup 文件添加了一个模块，为什么 make 失败了？

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 如何调试扩展？

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
```

然后运行 GDB：

```
$ gdb /local/bin/python
gdb) run myscript.py
gdb) continue # 重复直到你的扩展被载入
gdb) finish   # 你的扩展已被载入
gdb) br myfunction.c:50
gdb) continue
```

5.14 我想在 Linux 系统上编译一个 Python 模块，但是缺少一些文件。为什么？

大多数打包的 Python 版本都会省略一些编译 Python 扩展所必需的文件。

对于 Red Hat，请安装 python3-devel RPM 来获取必需的文件。

对于 Debian，请运行 `apt-get install python3-dev`。

5.15 如何区分“输入不完整”和“输入无效”？

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，您可以使用 `codeop` 模块，该模块非常接近解析器的行为。例如，IDLE 就使用了这个。

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 如何找到未定义的 g++ 符号 __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，您可以继承内置类，例如 `int` ， `list` ， `dict` 等。

Boost Python Library (BPL, https://www.boost.org/libs/python/doc/index.html) 提供了一种从 C++ 执行此操作的方式（即你可以使用 BPL 来继承用 C++ 编写的扩展类）。

CHAPTER 6

Python 在 Windows 上的常见问题

6.1 我怎样在 Windows 下运行一个 Python 程序？

这不一定是一个简单的问题。如果你已经熟悉在 Windows 的命令行中运行程序的方法，一切都显而易见；不然的话，你也许需要额外获得些许指导。

除非你使用某种集成开发环境，否则你最终会在所谓的" 命令提示窗口" 中 输入 Windows 命令。通常情况下，你可以在搜索栏中搜索 cmd 来创建这样一个窗口。你应该能够发现你已经启动了这样一个窗口，因为你会看到一个 Windows " 命令提示符"，它通常看起来像这样。

```
C:\>
```

前面的字母可能会不同，而且后面有可能会有其他东西，所以你也许会看到类似这样的东西：

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

出现的内容具体取决与你的电脑如何设置和最近用它做的事。当你启动了这样一个窗口后，就可以开始运行 Python 程序了。

Python 脚本需要被另外一个叫做 Python 解释器的程序来处理。解释器读取脚本，把它编译成字节码，然后执行字节码来运行你的程序。所以怎样安排解释器来处理你的 Python 脚本呢?

首先，确保命令窗口能够将 "py" 识别为指令来开启解释器。如果你打开过一个命令窗口，尝试输入命令 py 然后按回车：

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

然后你应当看见类似类似这样的东西：

```
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.
>>>
```

解释器已经以 "交互模式" 打开。这意味着你可以交互输入 Python 语句或表达式，并在等待时执行或评估它们。这是 Python 最强大的功能之一。输入几个表达式并看看结果：

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

许多人把交互模式当作方便和高度可编程的计算器。想结束交互式 Python 会话时，调用 `exit()` 函数，或者按住 `Ctrl` 键时输入 `Z`，之后按 `Enter` 键返回 Windows 命令提示符。

你可能发现在开始菜单有这样一个条目 *开始 ▸ 所有程序 ▸ Python 3.x ▸ Python (命令行)*，运行它后会出现一个有着 `>>>` 提示的新窗口。在此之后，如果调用 `exit()` 函数或按 `Ctrl-Z` 组合键后窗口将会消失。Windows 会在这个窗口中运行一个“python”命令，并且在你终止解释器的时候关闭它。

现在我们知道 `py` 命令已经被识别，可以输入 Python 脚本了。你需要提供 Python 脚本的绝对路径或相对路径。假设 Python 脚本位于桌面上并命名为 `hello.py`，并且命令提示符在用户主目录打开，那么可以看到类似于这样的东西:

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

那么现在可以让 `py` 命令执行你的脚本，只需要输入 `py` 和脚本路径:

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

6.2 我怎么让 Python 脚本可执行？

在 Windows 上，标准 Python 安装程序已将.py 扩展名与文件类型 (Python.File) 相关联，并为该文件类型提供运行解释器的打开命令 (`D:\Program Files\Python\python.exe "%1" %*`)。这足以使脚本在命令提示符下作为“foo.py”命令被执行。如果希望通过简单地键入“foo”而无需输入文件扩展名来执行脚本，则需要将.py 添加到 PATHEXT 环境变量中。

6.3 为什么有时候 Python 程序会启动缓慢？

通常，Python 在 Windows 上启动得很快，但偶尔会有错误报告说 Python 突然需要很长时间才能启动。更令人费解的是，在其他配置相同的 Windows 系统上，Python 却可以工作得很好。

该问题可能是由于计算机上的杀毒软件配置错误造成的。当将病毒扫描配置为监视文件系统中所有读取行为时，一些杀毒扫描程序会导致两个数量级的启动开销。请检查你系统安装的杀毒扫描程序的配置，确保两台机它们是同样的配置。已知的，McAfee 杀毒软件在将它设置为扫描所有文件系统访问时，会产生这个问题。

6.4 我怎样使用 Python 脚本制作可执行文件？

请参阅*如何由 Python 脚本创建能独立运行的二进制程序？* 查看可用来生成可执行文件的工具清单。

6.5 `*.pyd` 文件和 DLL 文件相同吗？

是的，.pyd 文件也是 dll，但有一些差异。如果你有一个名为 `foo.pyd` 的 DLL，那么它必须有一个函数 `PyInit_foo()`。然后你可以编写 Python 代码“import foo”，Python 将搜索 foo.pyd（以及 foo.py、foo.pyc）。如果找到它，将尝试调用 `PyInit_foo()` 来初始化它。你不应将.exe 与 foo.lib 链接，因为这会导致 Windows 要求存在 DLL。

请注意，foo.pyd 的搜索路径是 PYTHONPATH，与 Windows 用于搜索 foo.dll 的路径不同。此外，foo.pyd 不需要存在来运行你的程序，而如果你将程序与 dll 链接，则需要 dll。当然，如果你想 `import foo`，则需要 foo.pyd。在 DLL 中，链接在源代码中用 `__declspec(dllexport)` 声明。在.pyd 中，链接在可用函数列表中定义。

6.6 我怎样将 Python 嵌入一个 Windows 程序？

在 Windows 应用程序中嵌入 Python 解释器可以总结如下:

1. 请 **不要**直接将 Python 编译到你的.exe 文件中。在 Windows 上，Python 必须是一个 DLL 以便处理导入本身就是 DLL 的模块。(这是首先要知道的未写入文档的关键事实。) 正确的做法，应该是链接到 `pythonNN.dll`；它通常安装在 `C:\Windows\System` 中。*NN* 是 Python 的版本号，例如数字"33"代表 Python 3.3。

   你可以通过两种不同的方式链接到 Python 。加载时链接意味着链接到 `pythonNN.lib` ，而运行时链接意味着链接 `pythonNN.dll` 。(一般说明：`python NN.lib` 是所谓的 "import lib"，对应于 `pythonNN.dll` 。它只定义了链接器的符号。)

   运行时链接极大地简化了链接选项，一切都在运行时发生。你的代码必须使用 Windows 的 `LoadLibraryEx()` 程序加载 `pythonNN.dll` 。代码还必须使用使用 Windows 的 `GetProcAddress()` 例程获得的指针访问 `pythonNN.dll` 中程序和数据 (即 Python 的 C API)。宏可以使这些指针对任何调用 Python C API 中的例程的 C 代码都是透明的。

2. 如果你是使用 SWIG，那么很容易创建一个将使得应用的数据和方法可供 Python 使用的" 扩展模块"。SWIG 将为你处理所有繁琐的细节。结果是让你链接 置入你的.exe 文件当中的 C 代码 (!) 你 **无需**创建一个 DLL 文件，而这也简化了链接过程。

3. SWIG 将创建一个 init 函数 (一个 C 函数)，其名称取决于扩展模块的名称。例如，如果模块的名称是 leo ，则 init 函数将被称为 initleo() 。如果您使用 SWIG 阴影类，则 init 函数将被称为 initleoc() 。这初始化了一个由阴影类使用的隐藏辅助类。

   你可以将步骤 2 中的 C 代码链接到.exe 文件的原因是调用初始化函数等同于将模块导入 Python ! (这是第二个关键的未记载事实。)

4. 简而言之，你可以用以下代码使用扩展模块初始化 Python 解释器。

```
#include <Python.h>
...
Py_Initialize();  // 初始化 Python。
initmyAppc();  // 初始化 (导入) 辅助类。
PyRun_SimpleString("import myApp");  // 导入影子类。
```

5. Python C API 存在两个问题，如果你使用除 MSVC 之外的编译器用于构建 python.dll ，这将会变得明显。

   问题 1: 接受 `FILE *` 参数的所谓的" 极高层级" 函数在多编译器环境中将不起作用，因为每个编译器中 `struct FILE` 的概念都会是不同的。从实现的角度看来这些都是极低层级的函数。

   问题 2：在为 void 函数生成包装器时，SWIG 会生成以下代码：

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

   Py_None 是一个宏，它扩展为对 pythonNN.dll 中名为 _Py_NoneStruct 的复杂数据结构的引用。同样，此代码将在多编译器环境中失败。将此类代码替换为：

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

   有可能使用 SWIG 的 `%typemap` 命令自动进行更改，但我无法使其工作 (我是一个完全的 SWIG 新手)。

6. 使用 Python shell 脚本从 Windows 应用程序内部建立 Python 解释器窗口并不是一个好主意；生成的窗口将独立于应用程序的窗口系统。相反，你 (或 wxPythonWindow 类) 应该创建一个 "本机" 解释器窗口。将该窗口连接到 Python 解释器很容易。你可以将 Python 的 i/o 重定向到支持读写的 _ 任意 _ 对象，因此你只需要一个包含 read() 和 write() 方法的 Python 对象 (在扩展模块中定义)。

6.7 如何让编辑器不要在我的 Python 源代码中插入 tab ？

本 FAQ 不建议使用制表符，Python 样式指南 **PEP 8** ，为发行的 Python 代码推荐 4 个空格；这也是 Emacs python-mode 默认值。

在任何编辑器下，混合制表符和空格都是一个坏主意。MSVC 在这方面没有什么不同，并且很容易配置为使用空格：点击 *Tools* ‣ *Options* ‣ *Tabs*，对于文件类型“Default”，设置“Tab size”和“Indent size”为 4，并选择“插入空格”单选按钮。

如果混合制表符和空格导致前导空格出现问题，Python 会引发 `IndentationError` 或 `TabError`。你还可以运行 `tabnanny` 模块以批处理模式检查目录树。

6.8 如何在不阻塞的情况下检查按键？

使用 `msvcrt` 模块。这是一个标准的 Windows 专属扩展模块。它定义了一个函数 `kbhit()` 用于检查是否有键盘中的某个键被按下，以及 `getch()` 用于获取一个字符而不将其回显。

6.9 我该如何解决缺失 api-ms-win-crt-runtime-l1-1-0.dll 错误？

这将在使用未安装全部更新的 Windows 8.1 或更旧的系统时发生于 Python 3.5 及之后的版本上。首先请确保你的操作系统受支持并且已经更新补丁，如果此问题仍未解决，请访问 Microsoft support page 获取有关手动安装 C 运行时更新补丁的指导。

CHAPTER 7

图形用户界面（GUI）常见问题

7.1 图形界面常见问题

7.2 Python 有哪些 GUI 工具包？

Python 的标准构建包括一个指向 Tcl/Tk 部件集的面向对象的接口，称为 tkinter 。这可能是最容易安装（因为它包含在大多数 Python 的 二进制发行版 中）和使用的。关于 Tk 的更多信息，包括指向源代码的信息，见 Tcl/Tk 主页 。Tcl/Tk 可以完全移植到 macOS 、Windows 和 Unix 平台。

存在多种选项，具体取决于你的目标平台。Python Wiki 上提供了一个 跨平台 和 平台专属 的 GUI 框架列表。

7.3 有关 Tkinter 的问题

7.3.1 我怎样“冻结” Tkinter 程序？

Freeze （意为“冻结”）是一个用来创建独立应用程序的工具。当“冻结” Tkinter 程序时，程序并不是真的能够独立运行，因为程序仍然需要 Tcl 和 Tk 库。

一种解决方案是将应用程序与 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。

各种第三方冻结库例如 py2exe 和 cx_Freeze 都能够处理 Tkinter 应用程序的内置对象。

7.3.2 在等待 I/O 操作时能够处理 Tk 事件吗？

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 在 Tkinter 中键绑定不工作：为什么？

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。

最常见的原因是，那个绑定的控件没有“键盘焦点”。请在 Tk 文档中查找 focus 指令。通常一个控件要获得“键盘焦点”，需要点击那个控件（而不是标签；请查看 takefocus 选项）。

CHAPTER 8

“为什么我的电脑上安装了 Python ?”

8.1 什么是 Python ?

Python 是一种程序语言，被许多应用程序使用。它不仅因易学而在许多高校用于编程入门，还被工作于 Google、NASA 和卢卡斯影业等公司的软件开发人员使用。

如果你想学习更多 Python，看看 Beginner's Guide to Python.

8.2 为什么我的电脑上安装了 Python ?

如果你不记得你曾主动安装过 Python，但它却出现在了你的电脑上，这里有一些可能的原因。

- 可能是这台电脑的其他用户因想学习编程而安装了它，你得琢磨一下谁用过这台电脑并安装了 Python。
- 电脑上安装的第三方应用程序可能由 Python 写成并附带了一份 Python。这样的应用程序有很多，例如 GUI 程序、网络服务器、管理脚本等。
- 一些 Windows 可能预装了 Python。在撰写本文时，我们了解到 Hewlett-Packard 和 Compaq 的计算机包含 Python。显然，HP/Compaq 的一些管理工具是用 Python 编写的。
- 许多与 Unix 兼容的操作系统，如 macOS 和一些 Linux 发行版，都默认安装了 Python；它包含在基本安装中。

8.3 我能删除 Python 吗 ?

这取决于所安装 Python 的来源

如果有人主动安装了 Python，你可以在不影响其它程序的情况下安全移除它。在 Windows 中，可使用“控制面板”中的“添加/删除程序”卸载。

如果 Python 来源于第三方应用程序，你也能删除它，但那些程序将不能正常工作。你应该使用那些应用程序的卸载器而不是直接删除 Python。

如果 Python 来自于你的操作系统，不推荐删除！如果删除了它，任何用 Python 写成的工具将无法工作，其中某些工具对于你来说可能十分重要。你甚至可能需要重装整个系统来修复因删除 Python 留下的烂摊子。

APPENDIX A

术语对照表

>>>

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

...

具有以下含义：

- *interactive* shell 中输入特殊代码时默认的 Python 提示符，特殊代码包括缩进的代码块，左右成对分隔符（圆括号、方括号、花括号或三重引号等）之内，或是在指定一个装饰器之后。
- `Ellipsis` 内置常量。

abstract base class -- 抽象基类

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

annotation -- 标注

关联到某个变量、类属性、函数形参或返回值的标签，被约定作为类型注解 来使用。

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

参见*variable annotation*, *function annotation*, **PEP 484** 和 **PEP 526**，对此功能均有介绍。另请参见 annotations-howto 了解使用标注的最佳实践。

argument -- 参数

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

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

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

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

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

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

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

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

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

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

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

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

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

每个 `yield` 会临时暂停处理过程，记住执行状态（包括局部变量和挂起的 try 语句）。当 异步生成器迭代器通过 `__anext__()` 所返回的另一个可等待对象有效地恢复时，它会从离开位置恢复处理过程。参见 **PEP 492** 和 **PEP 525**。

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

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

attribute -- 属性
关联到一个对象的值，通常使用点号表达式按名称来引用。举例来说，如果对象 *o* 具有属性 *a* 则可以用 *o.a* 来引用它。

如果对象允许，将未被定义为 identifiers 的非标识名称用作一个对象的属性也是可以的，例如使用 `setattr()`。这样的属性将无法使用点号表达式来访问，而是必须通过 `getattr()` 来获取。

awaitable -- 可等待对象
一个可在 `await` 表达式中使用的对象。可以是*coroutine* 或是具有 `__await__()` 方法的对象。参见 **PEP 492**。

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

binary file -- 二进制文件
file object 能够读写字节型对象。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。

另请参见*text file* 了解能够读写 `str` 对象的文件对象。

borrowed reference -- 借入引用
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。

推荐在*borrowed reference* 上调用 `Py_INCREF()` 以将其原地转换为*strong reference*，除非是当该对象无法在借入引用的最后一次使用之前被销毁。`Py_NewRef()` 函数可以被用来创建一个新的*strong reference*。

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

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

bytecode -- 字节码
Python 源代码会被编译为字节码，即 CPython 解释器中表示 Python 程序的内部代码。字节码还会缓存在 `.pyc` 文件中，这样第二次执行同一文件时速度更快（可以免去将源码重新编译为字节码）。这种” 中间语言” 运行在根据字节码执行相应机器码的*virtual machine* 之上。请注意不同 Python 虚拟机上的字节码不一定通用，也不一定能在不同 Python 版本上兼容。

字节码指令列表可以在 dis 模块的文档中查看。

callable -- 可调用对象
可调用对象就是可以执行调用运算的对象，并可能附带一组参数 (参见*argument*)，使用以下语法:

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

function，还可扩展到*method* 等，就属于可调用对象。实现了 `__call__()` 方法的类的实例也属于可调用对象。

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

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

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

closure variable -- 闭包变量
引用自*nested scope* 的*free variable*，它是在外层作用域中定义而不是在运行时自全局或内置命名空间中取得。可能使用 `nonlocal` 关键字显式地定义以允许写入访问，或者如果变量仅供读取则只需隐式地定义。

例如，在以下代码的 `inner` 函数中，`x` 和 `print` 均为自由变量，但只有 `x` 属于 闭包变量:

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

由于 `codeobject.co_freevars` 属性的存在（虽然名称如此，但其仅包括闭包变量名称而非列出所有被引用的自由变量），更一般化的术语*free variable* 有时在即便本意是专指闭包变量时也会被使用。

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

context -- 上下文
此术语根据其所在场合和使用方式的不同而具有不同的含义。一些常见的含义为:

- 通过 `with` 语句由一个*context manager* 所创建的临时环境或状态。
- 一组关联到特定 `contextvars.Context` 对象并通过 `ContextVar` 对象来访问的键值绑定。另请参见*context variable*。
- 一个 `contextvars.Context` 对象。另请参见*current context*。

上下文管理协议
`__enter__()` 和 `__exit__()` 方法将由 `with` 语句来调用。参见 **PEP 343**。

context manager -- 上下文管理器
一个实现了*context management protocol* 并负责控制某个 `with` 语句内的环境的对象。参见 **PEP 343**。

context variable -- 上下文变量
一个具体值取决于哪个上下文是*current context* 的变量。这些值是通过 `contextvars.ContextVar` 对象来访问的。上下文变量主要被用来隔离并发的异步任务之间的状态。

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

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

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

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

current context -- 当前上下文
在当前被 `ContextVar` 对象用来访问 (获取或设置) 上下文变量 的值的*context* (`contextvars.Context` 对象)。每个线程都具有它自己的当前上下文。用于执行异步任务 (参见 `asyncio`) 的框架会在每个任务开始或恢复执行时将任务关联到一个成为当前上下文的上下文。

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

装饰器语法只是一种语法糖，以下两个函数定义在语义上完全等价:

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

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

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

descriptor -- 描述器
任何定义了 `__get__()`, `__set__()` 或 `__delete__()` 方法的对象。当一个类属性为描述器时，它的特殊绑定行为就会在属性查找时被触发。通常情况下，使用 *a.b* 来获取、设置或删除一个属性时会在 *a* 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。

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

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

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

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

docstring -- 文档字符串
作为类、函数或模块之内的第一个表达式出现的字符串字面值。它在代码块被执行时将被忽略，但会被编译器识别并放入所在类、函数或模块的 `__doc__` 属性中。由于它可用于代码内省，因此是存放对象的文档的规范位置。

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

dunder
An informal short-hand for "double underscore", used when talking about a *special method*. For example, `__init__` is often pronounced "dunder init".

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

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

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

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

file object -- 文件对象
对外公开面向文件的 API（带有 `read()` 或 `write()` 等方法）以使用下层资源的对象。根据其创建方式的不同，文件对象可以处理对真实磁盘文件、其他类型的存储或通信设备的访问（例如标准输入/输出、内存缓冲区、套接字、管道等）。文件对象也被称为 文件型对象或 流。

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

file-like object -- 文件型对象
file object 的同义词。

filesystem encoding and error handler -- 文件系统编码格式与错误处理器
Python 用来从操作系统解码字节串和向操作系统编码 Unicode 的编码格式与错误处理器。

文件系统编码格式必须保证能成功解码长度在 128 以下的所有字节串。如果文件系统编码格式无法提供此保证，则 API 函数可能会引发 `UnicodeError`。

`sys.getfilesystemencoding()` 和 `sys.getfilesystemencodeerrors()` 函数可被用来获取文件系统编码格式与错误处理器。

filesystem encoding and error handler 是在 Python 启动时通过 `PyConfig_Read()` 函数来配置的：请参阅 `PyConfig` 的 `filesystem_encoding` 和 `filesystem_errors` 等成员。

另请参见*locale encoding*。

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

存在两种类型的查找器: 元路径查找器 配合 `sys.meta_path` 使用，以及路径条目查找器 配合 `sys.path_hooks` 使用。

请参阅 finders-and-loaders 和 `importlib` 以了解更多细节。

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

free threading -- 自由线程
一种线程模型，在同一个解释器内部的多个线程可以同时运行 Python 字节码。与此相对的是*global interpreter lock*，在同一时刻只允许一个线程执行 Python 字节码。参见 **PEP 703**。

free variable -- 自由变量
在正式场合下，如 语言执行模型所定义的，自由变量是指在某个命名空间中被使用的不属于该命名空间中的局部变量的任何变量。参见*closure variable* 中的样例。在实际应用中，由于 `codeobject.co_freevars` 属性被如此命名，该术语有时也被用作*closure variable* 的同义词。

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

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

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

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

函数标注语法的详解见 function 一节。

参见*variable annotation* 和 **PEP 484**，其中描述了此功能。另请参阅 annotations-howto 以了解使用标的最佳实践。

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

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

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

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

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

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

每个 `yield` 会临时暂停处理过程，记住执行状态（包括局部变量和挂起的 try 语句）。当 生成器迭代器恢复时，它会从离开位置继续执行（不同于每次被唤起时都从新开始的普通函数）。

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

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

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

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

generic type -- 泛型
可参数化的*type*；通常为 `list` 或 `dict` 这样的 容器类。用于类型提示 和注解。

更多细节参见 泛型别名类型、**PEP 483**、**PEP 484**、**PEP 585** 和 `typing` 模块。

GIL
参见*global interpreter lock*。

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

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

在 Python 3.13 中，GIL 可以使用 `--disable-gil` 构建配置选项来禁用。在使用此选项构建 Python 之后，代码必须附带 `-X gil=0` 或在设置 `PYTHON_GIL=0` 环境变量后运行。此特性将为多线程应用程序启用性能提升并让高效率地使用多核 CPU 更加容易。要了解详情，请参阅 **PEP 703**。

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

hashable -- 可哈希
一个对象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。

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

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

IDLE
Python 的集成开发与学习环境。idle 是 Python 标准发行版附带的基本编辑器和解释器环境。

immortal -- 永生对象
永生对象是在 **PEP 683** 中引入的 CPython 实现细节。

如果对象属于永生对象，则它的*reference count* 永远不会被修改，因而它在解释器运行期间永远不会被取消分配。例如，`True` 和 `None` 在 CPython 中都属于永生对象。

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

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

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

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

interactive -- 交互
Python 带有一个交互式解释器，这意味着你可以在解释器提示符后输入语句和表达式，立即执行并查看其结果。只需不带参数地启动 `python` 命令（也可以在你的计算机主菜单中选择相应菜单项）。在测试新想法或检验模块和包的时候这会非常方便（记住 `help(x)` 函数）。有关交互模式的详情，参见 tut-interac。

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

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

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

iterable -- 可迭代对象
一种能够逐个返回其成员项的对象。可迭代对象的例子包括所有序列类型（如 `list`, `str` 和 `tuple` 等）以及某些非序列类型如 `dict`, 文件对象 以及任何定义了 `__iter__()` 方法或实现了*sequence* 语义的 `__getitem__()` 方法的自定义类的对象。

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

iterator -- 迭代器
用来表示一连串数据流的对象。重复调用迭代器的 `__next__()` 方法 (或将其传给内置函数 `next()`) 将逐个返回流中的项。当没有数据可用时则将引发 `StopIteration` 异常。到这时迭代器对象中的数据项已耗尽，继续调用其 `__next__()` 方法只会再次引发 `StopIteration`。迭代器必须具有 `__iter__()` 方法用来返回该迭代器对象自身，因此迭代器必定也是可迭代对象，可被用于其他可迭代对象适用的大部分场合。一个显著的例外是那些会多次重复访问迭代项的代码。容器对象 (例如 `list`) 在你每次将其传入 `iter()` 函数或是在 `for` 循环中使用时都会产生一个新的迭代器。如果在此情况下你尝试用迭代器则会返回在之前迭代过程中被耗尽的同一迭代器对象，使其看起来就像是一个空容器。

更多信息可查看 typeiter。

CPython 没有强制推行迭代器定义 `__iter__()` 的要求。还要注意的是自由线程 CPython 并不保证迭代器操作的线程安全性。

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

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

有多种方式可以创建一个键函数。例如，`str.lower()` 方法可以用作忽略大小写排序的键函数。或者，键函数也可通过 `lambda` 表达式来创建例如 `lambda r: (r[0], r[2])`。此外，`operator.attrgetter()`, `operator.itemgetter()` 和 `operator.methodcaller()` 是键函数的三个构造器。请查看 排序指引来获取创建和使用键函数的示例。

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

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

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

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

lexical analyzer -- 词法分析器
分词器的正式名称；参见*token*。

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

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

loader -- 加载器
负责加载模块的对象。它必须定义 `exec_module()` 和 `create_module()` 方法以实现 `Loader` 接口。加载器通常由一个*finder* 返回。另请参阅：

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

locale encoding -- 语言区域编码格式
在 Unix 上，它是 LC_CTYPE 语言区域的编码格式。它可以通过 `locale.setlocale(locale.LC_CTYPE, new_locale)` 来设置。

在 Windows 上，它是 ANSI 代码页 (如: `"cp1252"`)。

在 Android 和 VxWorks 上，Python 使用 `"utf-8"` 作为语言区域编码格式。

`locale.getencoding()` 可被用来获取语言区域编码格式。

另请参阅*filesystem encoding and error handler*。

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

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

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

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

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

更多详情参见 metaclasses。

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

method resolution order -- 方法解析顺序
方法解析顺序就是在查找成员时搜索基类的顺序。请参阅 python_2.3_mro 了解自 2.3 发布版起 Python 解释器所使用算法的详情。

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

另见*package*。

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

另请参阅 module-specs。

MRO
参见*method resolution order*。

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

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

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

```
>>> sys.float_info[1]                   # 索引访问
1024
>>> sys.float_info.max_exp              # 命名字段访问
1024
>>> isinstance(sys.float_info, tuple)   # 属于元组
True
```

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

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

namespace package -- 命名空间包
一种仅被用作子包的容器的*package*。命名空间包可以没有实体表示物，具体而言就是不同于*regular package* 因为它们没有 `__init__.py` 文件。

命名空间包允许几个可单独安装的包具有共同的父包。在其他情况下，则推荐使用*regular package*。

要了解更多信息，请参阅 **PEP 420** 和 reference-namespace-package。

另可参见*module*。

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

new-style class -- 新式类
对目前已被应用于所有类对象的类形式的旧称谓。在较早的 Python 版本中，只有新式类能够使用 Python 新增的更灵活我，如 `__slots__`、描述器、特征属性、`__getattribute__()`、类方法和静态方法等。

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

optimized scope -- 已优化的作用域
当代码被编译时编译器已充分知晓目标局部变量名称的作用域，这允许对这些名称的读写进行优化。针对函数、生成器、协程、推导式和生成器表达式的局部命名空间都是这样的已优化作用域。注意：大部分解释器优化将应用于所有作用域，只有那些依赖于已知的局部和非局部变量名称集合的优化会仅限于已优化的作用域。

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

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

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

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

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

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

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

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

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

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

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

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

形参可以同时指定可选和必选参数，也可以为某些可选参数指定默认值。

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

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

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

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

path entry hook -- 路径入口钩子
一种可调用对象，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 -- 基于路径的查找器
默认的一种元路径查找器，可在一个*import path* 中查找模块。

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

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

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

参见 **PEP 1**。

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

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

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

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

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

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

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

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

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

而相应的更简洁更 Pythonic 的方法是这样的:

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

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

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

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

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

reference count -- 引用计数
指向某个对象的引用的数量。当一个对象的引用计数降为零时，它就会被释放。特殊的*immortal* 对象具有永远不会被修改的引用计数，因此这种对象永远不会被释放。引用计数对 Python 代码来说通常是不可见的，但它是*CPython* 实现的一个关键元素。程序员可以调用 `sys.getrefcount()` 函数来返回特定对象的引用计数。

In *CPython*, reference counts are not considered to be stable or well-defined values; the number of references to an object, and how that number is affected by Python code, may be different between versions.

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

另参见*namespace package*。

REPL
“读取-求值-打印循环” read-eval-print loop 的缩写，*interactive* 解释器 shell 的另一个名字。

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

sequence -- 序列
一种*iterable*，它支持通过 `__getitem__()` 特殊方法来使用整数索引进行高效的元素访问，并定义了一个返回序列长度的 `__len__()` 方法。内置序列类型有 `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，因为它使用任意的*hashable* 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 -- 集合推导式
处理一个可迭代对象中的所有或部分元素并返回结果集合的一种紧凑写法。`results = {c for c in 'abracadabra' if c not in 'abc'}` 将生成字符串集合 `{'r', 'd'}`。参见 comprehensions。

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

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

soft deprecated -- 软弃用
被软弃用的 API 不应在新编写的代码中使用，但在已有代码中使用仍是安全的。这样的 API 将保留在文档中并被测试，但不会再获得进一步的功能增强。

与普通的弃用不同，软弃用没有 API 移除计划也不会发出弃用警告。

参见 PEP 387: Soft Deprecation。

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

standard library -- 标准库
作为官方 Python 解释器软件包的组成部分一同发布的包, 模块 和扩展模块 集合。该集合成员的实际内容可能因系统平台、可用系统库或其他条件的不同而发生变化。相关文档可在 library-index 查看。

另请参阅 `sys.stdlib_module_names` 获取所有可用标准库模块名称的列表。

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

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。

stdlib -- 标准库
standard library 的缩写。

strong reference -- 强引用
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()` 函数可被用于创建一个对象的强引用。通常，必须在退出某个强引用的作用域时在该强引用上调用 `Py_DECREF()` 函数，以避免引用的泄漏。

另请参阅*borrowed reference*。

text encoding -- 文本编码格式
在 Python 中，一个字符串是一串 Unicode 代码点（范围为 `U+0000--U+10FFFF`）。为了存储或传输一个字符串，它需要被序列化为一串字节。

将一个字符串序列化为一个字节序列被称为“编码”，而从字节序列中重新创建字符串被称为“解码”。

有各种不同的文本序列化 编码器，它们被统称为” 文本编码格式”。

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

另请参看*binary file* 了解能够读写字节型对象 的文件对象。

形符
一个源代码小单元，由 词法分析器 (或称 分词器) 生成。名称、数字、字符串、运算符、换行符等均由词元来表示。

`tokenize` 模块对外暴露了 Python 的词法分析器。`token` 模块包含了有关各种词元类型的信息。

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

type -- 类型
Python 对象的类型决定它属于什么种类；每个对象都具有特定的类型。对象的类型可通过其 `__class__` 属性来访问或是用 `type(obj)` 来获取。

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

类型别名的作用是简化*类型注解*。例如:

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

可以这样提高可读性:

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

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

参见 `typing` 和 **PEP 484**，其中有对此功能的详细描述。

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

类型提示是可选的而不是 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。

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

参见 `typing` 和 **PEP 484**，其中有对此功能的详细描述。

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

variable annotation -- 变量标注
对变量或类属性的*annotation*。

在标注变量或类属性时，还可选择为其赋值:

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

变量标注通常被用作*类型提示*：例如以下变量预期接受 `int` 类型的值:

```
count: int = 0
```

变量标注语法的详细解释见 annassign 一节。

参见*function annotation*, **PEP 484** 和 **PEP 526**，其中描述了此功能。另请参阅 annotations-howto 以了解使用标注的最佳实践。

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

另参见 `venv`。

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

海象运算符
A light-hearted way to refer to the assignment expression operator `:=` because it looks a bit like a walrus if you turn your head.

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

APPENDIX B

关于本文档

Python 的文档是用 Sphinx 从 reStructuredText 源生成的，*Sphinx* 是一个专为处理 Python 文档而编写的文档生成器。

本文档及其工具链之开发，皆在于志愿者之努力，亦恰如 Python 本身。如果您想为此作出贡献，请阅读 reporting-bugs 了解如何参与。我们随时欢迎新的志愿者！

特别鸣谢：

- Fred L. Drake, Jr.，原始 Python 文档工具集之创造者，众多文档之作者；
- 用于创建 reStructuredText 和 Docutils 套件的 Docutils 项目；
- Fredrik Lundh 的 Alternative Python Reference 项目，为 Sphinx 提供许多好的点子。

B.1 Python 文档的贡献者

有很多对 Python 语言，Python 标准库和 Python 文档有贡献的人，随 Python 源代码分发的 Misc/ACKS 文件列出了部分贡献者。

有了 Python 社区的输入和贡献，Python 才有了如此出色的文档——谢谢你们！

APPENDIX C

历史和许可证

C.1 该软件的历史

Python 是在 1990 年代初由 Guido van Rossum 在荷兰的 Stichting Mathematisch Centrum (CWI，见 https://www.cwi.nl) 作为一门名为 ABC 的语言的后继者创造的。Guido 仍然是 Python 的主要作者，尽管它也包括了许多来自其他人的贡献。

在 1995 年，Guido 在弗吉尼亚州 Reston 市的 Corporation for National Research Initiatives (CNRI，见 https://www.cnri.reston.va.us) 继续他对 Python 的工作，他在那里发布了该软件的多个版本。

在 2000 年 5 月，Guido 和 Python 核心开发团队移至 BeOpen.com 组建了 BeOpen PythonLabs 团队。同年 10 月，PythonLabs 团队移至 Digital Creations，后改名为 Zope Corporation。在 2001 年，Python Software Foundation (PSF，见 https://www.python.org/psf/) 成立，这是一个专门为持有与 Python 相关的知识产权而创立的非营利组织。Zope Corporation 是 PSF 的一个赞助成员。

所有 Python 发布版都是开源的 (有关开源的定义见 https://opensource.org)。在历史上，大多数但并非全部 Python 发布版还是 GPL 兼容的；下表总结了各个版本的情况。

发布版本	源自	年份	所有者	GPL 兼容? (1)
0.9.0 至 1.2	n/a	1991-1995	CWI	是
1.3 至 1.5.2	1.2	1995-1999	CNRI	是
1.6	1.5.2	2000	CNRI	否
2.0	1.6	2000	BeOpen.com	否
1.6.1	1.6	2001	CNRI	是 (2)
2.1	2.0+1.6.1	2001	PSF	否
2.0.1	2.0+1.6.1	2001	PSF	是
2.1.1	2.1+2.0.1	2001	PSF	是
2.1.2	2.1.1	2002	PSF	是
2.1.3	2.1.2	2002	PSF	是
2.2 及更高	2.1.1	2001 至今	PSF	是

备注

(1) GPL 兼容并不意味着我们是基于 GPL 发布 Python。与 GPL 不同，所有 Python 许可证都允许你分发经修改的版本而无需开源你所做的修改。GPL 兼容的许可证使得 Python 可以与其他基于

GPL 发布的软件结合使用；其他许可证则不可以。

(2) 按照 Richard Stallman 的说法，1.6.1 不是 GPL 兼容的，因为它的许可证包含特定的法律条款。但是按照 CNRI 的说法，Stallman 的律师已告诉 CNRI 的律师 1.6.1 与 GPL " 并非不兼容"。

感谢众多在 Guido 指导下工作的外部志愿者，使得这些发布成为可能。

C.2 获取或以其他方式使用 Python 的条款和条件

Python 软件和文档的使用许可均为 Python Software Foundation License Version 2。

从 Python 3.8.6 开始，文档中的示例、操作指导和其他代码均采用 PSF License Version 2 和*Zero-Clause BSD license* 双重使用许可。

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

C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise using this
   software ("Python") in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001-2024 Python Software Foundation; All Rights
   Reserved" are retained in Python alone or in any derivative version
   prepared by Licensee.

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

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

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

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

7. Nothing in this License Agreement shall be deemed to create any relationship
   of agency, partnership, or joint venture between PSF and Licensee.  This License
   Agreement does not grant permission to use PSF trademarks or trade name in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.

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

C.2.2 用于 PYTHON 2.0 的 BEOPEN.COM 许可协议

BEOPEN PYTHON 开源许可协议第 1 版

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

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

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

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

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

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

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

C.2.3 用于 PYTHON 1.6.1 的 CNRI 许可协议

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

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

(续下页)

(接上页)

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

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

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

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

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

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

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

C.2.4 用于 PYTHON 0.9.0 至 1.2 的 CWI 许可协议

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

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

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

(续下页)

(接上页)

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

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

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

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

C.3 收录软件的许可与鸣谢

本节是 Python 发行版中收录的第三方软件的许可和致谢清单，该清单是不完整且不断增长的。

C.3.1 Mersenne Twister

作为 `random` 模块下层的 `_random` C 扩展包括基于从 http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html 下载的代码。以下是原始代码的完整注释:

```
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
```

(续下页)

(接上页)

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


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

C.3.2 套接字

`socket` 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 异步套接字服务

`test.support.asynchat` 和 `test.support.asyncore` 模块包含以下说明。:

```
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
```

(续下页)

(接上页)

```
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.4 Cookie 管理

`http.cookies` 模块包含以下声明:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

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

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

C.3.5 执行追踪

`trace` 模块包含以下声明:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
```

(续下页)

(接上页)

```
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 UUencode 与 UUdecode 函数

uu 编解码器包含以下声明:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 XML 远程过程调用

xmlrpc.client 模块包含以下声明:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

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

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
```

(续下页)

(接上页)

```
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` 模块关于 kqueue 的接口包含以下声明:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

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

C.3.10 SipHash24

Python/pyhash.c 文件包含 Marek Majkowski' 对 Dan Bernstein 的 SipHash24 算法的实现。它包含以下声明:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

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

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

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod 和 dtoa

Python/dtoa.c 文件提供了 C 函数 dtoa 和 strtod，用于 C 双精度数值和字符串之间的转换，它派生自由 David M. Gay 编写的同名文件。目前该文件可在 https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c 访问。在 2009 年 3 月 16 日检索到的原始文件包含以下版权和许可声明:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.12 OpenSSL

如果操作系统有支持则 hashlib, posix 和 ssl 会使用 OpenSSL 库来提升性能。此外，Python 的 Windows 和 macOS 安装程序可能会包括 OpenSSL 库的副本，所以我们也在此包括一份 OpenSSL 许可证的副本。对于 OpenSSL 3.0 发布版，及其后续衍生版本，均使用 Apache License v2:

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

(续下页)

(接上页)

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

1. Definitions.

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

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

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

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

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

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

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

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

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

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

(续下页)

(接上页)

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

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

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

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

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

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

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

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

(续下页)

(接上页)

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

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

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

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

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

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

END OF TERMS AND CONDITIONS
```

C.3.13 expat

`pyexpat` 扩展是使用所包括的 expat 源副本来构建的，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
```

(续下页)

(接上页)

```
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

作为 `ctypes` 模块下层的 `_ctypes` C 扩展是使用包括了 libffi 源的副本构建的，除非构建时配置了 `--with-system-libffi`:

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

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

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

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

C.3.15 zlib

如果系统上找到的 zlib 版本太旧而无法用于构建，则使用包含 zlib 源代码的拷贝来构建 `zlib` 扩展:

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

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

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
```

(续下页)

(接上页)

```
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

tracemalloc 使用的哈希表的实现基于 cfuhash 项目:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

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

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

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

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

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

C.3.17 libmpdec

作为 decimal 模块下层的 _decimal C 扩展是使用包括了 libmpdec 库的副本构建的，除非构建时配置了 --with-system-libmpdec:

```
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:
```

(续下页)

(接上页)

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

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

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

C.3.18 W3C C14N 测试套件

`test` 包中的 C14N 2.0 测试集 (`Lib/test/xmltestdata/c14n-20/`) 提取自 W3C 网站 https://www.w3.org/TR/xml-c14n2-testcases/ 并根据 3 条款版 BSD 许可证发行:

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

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

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work without
  specific prior written permission.

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

C.3.19 mimalloc

MIT 许可证:

```
Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

Permission is hereby granted, free of charge, to any person obtaining a copy
```

(续下页)

(接上页)

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

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

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

C.3.20 asyncio

`asyncio` 模块的某些部分来自 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.
```

C.3.21 Global Unbounded Sequences (GUS)

文件 `Python/qsbr.c` 改编自 subr_smr.c 中 FreeBSD 的"Global Unbounded Sequences" 安全内存回收方案。该文件是基于 2 条款 BSD 许可证分发的:

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice unmodified, this list of conditions, and the following
   disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR
```

(续下页)

(接上页)

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

APPENDIX D

版权所有

Python 与这份文档：

有关完整的许可证和许可信息，请参见历史和许可证。

索引

非字母

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z