
Python Frequently Asked Questions

发布 3.7.8rc1

Guido van Rossum
and the Python development team

六月 23, 2020

Python Software Foundation
Email: docs@python.org

Contents

1 Python 常见问题	1
2 编程常见问题	7
3 设计和历史常见问题	33
4 代码库和插件 FAQ	45
5 扩展/嵌入常见问题	55
6 Python 在 Windows 上的常见问题	63
7 图形用户界面 (GUI) 常见问题	67
8 “为什么我的电脑上安装了 Python ?”	71
A 术语对照表	73
B 文档说明	85
C 历史和许可证	87
D Copyright	103
索引	105

1.1 一般信息

1.1.1 什么是 Python ?

Python 是一种解释性、交互式、面向对象的编程语言。它包含了模块、异常、动态类型、非常高层级的动态数据类型以及类的概念。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各种窗口系统的接口，并且可以用 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 还是可移植的：它可以在许多 Unix 变种、Mac 以及 Windows 2000 以上的操作系统中运行。

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 [PSF 许可页](#) 以查看进一步的说明以及许可的完整文本内容的链接。

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。

该语言附带一个庞大的标准库，涵盖了字符串处理（正则表达式，Unicode，比较文件间的差异等），因特网协议（HTTP，FTP，SMTP，XML-RPC，POP，IMAP，CGI 编程等），软件工程（单元测试，日志记录，性能分析，Python 代码解析等），以及操作系统接口（系统调用，文件系统，TCP/IP 套接字等）。请查看 [library-index](#) 的目录以了解所有可用的内容。此外还可以获取到各种各样的第三方扩展。请访问 [Python 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 版本的编号形式是 A.B.C 或 A.B。A 称为大版本号——它仅在对语言特性进行非常重大改变时才会递增。B 称为小版本号，它会在语言特性发生较小改变时递增。C 称为微版本号——它会在每次发布问题修正时递增。请参阅 [PEP 6](#) 了解有关问题修正发布版的详情。

发布版本并非都是问题修正版本。在新的主要发布版本开发过程中，还会发布一系列的开发版，它们以 alpha (a), beta (b) 或 release candidate (rc) 来标示。其中 alpha 版是早期发布的测试版，它的接口并未最终确定；在两个 alpha 发布版本间出现接口的改变并不意外。而 beta 版更为稳定，它会保留现有的接口，但也可能增加新的模块，release candidate 版则会保持冻结状态不会再进行改变，除非有重大问题需要修正。

以上 alpha, beta 和 release candidate 版本会附加一个后缀。用于 alpha 版本的后缀是带有一个小数字 N 的“aN”，beta 版本的后缀是带有一个小数字 N 的“bN”，release candidate 版本的后缀是带有一个小数字 N 的“cN”。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 2.0cN 的版本，而这些版本全都早于 2.0。

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”。

另请参阅 `sys.version`, `sys.hexversion` 以及 `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](https://www.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，每天会收到五个左右的发帖。可以在 [Python 公告邮件列表](#) 页面进行订阅。

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 `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？

要报告错误或提交补丁，请使用安装于 <https://bugs.python.org/> 上的 Roundup。

你必须拥有一个 Roundup 账号才能报告错误；这样我们就可以在有后续问题时与你联系。这也使得 Roundup 能在我们处理所报告的错误时向你发送更新消息。如果你之前使用过 SourceForge 向 Python 报告错误，你可以通过 Roundup 的 [密码重置操作](#) 来获取你的 Roundup 密码。

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。

第一篇有关 Python 的文章写于 1991 年，现在其内容已经相当过时了。

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](https://wiki.python.org/moin/PythonBooks) 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 项目的基础架构分布于世界各地并由 Python 基础架构团队负责管理。详情请访问 [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 到 18 个月就会推出新的稳定发布版，这种状态看来还将持续下去。目前主要发布版本的间隔通常为 18 个月左右。

开发者也会推出旧版本的”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），用于稳定性的管理；只有对已知问题的修正会包含在问题修正发布版中，同一系列的问题修正发布版中的接口确定将会始终保持一致。

最新的稳定版本总是可以在 [Python 下载页面](#) 上找到。有两个生产环境可用的 Python 版本：2.x 和 3.x。推荐的版本是 3.x，大多数广泛使用的库都支持它。虽然 2.x 仍然被广泛使用，但它在 2020 年 1 月 1 日之后将不再维护。

1.2.2 有多少人在使用 Python ?

使用者的数量肯定非常庞大，不过想要进行精确统计则是相当困难的。

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，其中最著名的有 Red Hat 均已使用 Python 来编写部分或全部的安装程序和系统管理软件。在内部使用 Python 的大公司包括了 Google, Yahoo 以及 Lucasfilm 等。

1.2.4 在未来可以期待 Python 将有什么新进展 ?

请访问 <https://www.python.org/dev/peps/> 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。可以查找标题为“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 ?

有的。

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，他们可以这样做：

```
>>> L = []
>>> dir(L)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattr__', '__getitem__', '__gt__', '__hash__', '__iadd__',
 '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
 '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',
 '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
 'reverse', 'sort']
>>> [d for d in dir(L) if '__' not in d]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
 ↪ '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 还拥有很好的 IDE。IDLE 是一个跨平台的 Python IDE，它基于 Tkinter 库，使用 Python 语言编写。PythonWin 是一个 Windows 专属的 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，你可能会感兴趣加入 [edu-sig](#) 邮件列表。

2.1 一般问题

2.1.1 Python 有没有提供断点与单步调试等功能的，源码层次的调试器？

有的。

以下介绍了一些 Python 的调试器，内置函数 `breakpoint()` 允许你使用其中的任何一种。

`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 发行版附带组件的 IDLE 交互式环境（通常位于 `Tools/scripts/idle`）中包含一个图形化的调试器。

PythonWin 是一个包含有基于 `pdb` 的 GUI 调试器的 Python IDE。Pythonwin 调试器会为断点加上颜色，并具有许多很棒的特性，例如也可以非 Pythonwin 程序。Pythonwin 是 [Python for Windows Extensions](#) 项目的一部分，也是 ActivePython 发行版的一部分（参见 <https://www.activestate.com/activepython>）。

[Boa Constructor](#) 是一个使用 `wxWidgets` 的 IDE 和 GUI 构建器。它提供可视化框架创建和操作，对象检查器，源对象浏览器上的许多视图，继承层次结构，doc 字符串生成的 html 文档，高级调试器，集成帮助和 Zope 支持。

[Eric](#) 是一个基于 PyQt 和 Scintilla 编辑组件构建的 IDE。

`Pydb` 是标准 Python 调试器 `pdb` 的一个版本，经过修改后可与 DDD（数据显示调试器）一起使用，DDD 是一种流行的图形化调试器前端。`Pydb` 可以在 <http://bashdb.sourceforge.net/pydb/> 找到，DDD 可以在 <https://www.gnu.org/software/ddd> 找到。

有许多商业 Python IDE 包括图形调试器。他们包括：

- Wing IDE (<https://wingware.com/>)
- Komodo IDE (<https://komodoide.com/>)
- PyCharm (<https://www.jetbrains.com/pycharm/>)

2.1.2 有没有工具来帮助找寻漏洞或进行静态分析？

有的。

PyChecker 是一个寻找 Python 代码漏洞以及对代码复杂性和风格给出警告的工具。你可以从这里获得 PyChecker: <http://pychecker.sourceforge.net/>。

Pylint 是另一个检查模块是否满足编码标准的工具，也可以编写插件来添加自定义功能。除了 PyChecker 执行的错误检查之外，Pylint 还提供了一些额外的功能，例如检查行长度，变量名称是否根据您的编码标准格式良好，声明的接口是否完全实现等等。<https://docs.pylint.org/> 提供了 Pylint 功能的完整列表。

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 我如何能够通过一个 Python 脚本创建一个独立运行的二进制文件？

如果你想要的只是一个独立的程序，用户可以下载和运行而不必先安装 Python 发行版，你就不需要将 Python 编译成 C 代码。有许多工具可以确定程序所需的模块集，并将这些模块与 Python 二进制文件绑定在一起以生成单个可执行文件。

一种是使用冻结工具，它包含在 Python 源代码树 `Tools/freeze` 中。它将 Python 字节码转换为 C 数组；一个 C 编译器，你可以将所有模块嵌入到一个新程序中，然后将其与标准 Python 模块链接。

它的工作原理是递归扫描源代码以获取 `import` 语句（两种形式），并在标准 Python 路径和源目录（用于内置模块）中查找模块。然后，它将由 Python 编写的模块的字节码转换为 C 代码（可以使用编组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的，该文件仅包含程序中实际使用的内置模块。然后，它编译生成的 C 代码并将其与 Python 解释器的其余部分链接，以形成一个独立的二进制文件，其行为与你的脚本完全相同。

显然，freeze 需要一个 C 编译器。有几个其他实用工具不需要。一个是 Thomas Heller 的 `py2exe`（仅限 Windows）

<http://www.py2exe.org/>

另一个工具是 Anthony Tuininga 的 `cx_Freeze`。

2.1.4 是否有 Python 程序规范代码标准或风格指南？

有的。请参阅标准库模块所要求的代码风格描述文档 [PEP 8](#)。

2.2 核心语言

2.2.1 当变量有值时，为什么会出现 `UnboundLocalError`？

通过在函数体中的某处添加赋值语句，导致以前正常工作的代码被修改而得到 `UnboundLocalError` 会令人感到意外。

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
```

会得到一个 `UnboundLocalError` :

```
>>> foo()
Traceback (most recent call last):
...
UnboundLocalError: local variable 'x' referenced before assignment
```

这是因为当你对作用域中的变量进行赋值时，该变量将成为该作用域的局部变量，并在外部作用域中隐藏任何类似命名的变量。由于 `foo` 中的最后一个语句为 `x` 分配了一个新值，编译器会将其识别为局部变量。因此，当先前的 `print(x)` 尝试打印未初始化的局部变量时会导致错误。

在上面的示例中，你可以通过将其声明为全局来访问外部作用域变量：

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

这个显式声明是必需的，以便提醒你（与类和实例变量的表面类似情况不同），你实际上是在外部作用域中修改变量的值

```
>>> 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 中的局部变量和全局变量有哪些规则？

在 Python 中，仅在函数内引用的变量是隐式全局变量。如果在函数体内的任何位置为变量赋值，则除非明确声明为全局，否则将其视为局部值。

虽然起初有点令人惊讶，但片刻考虑就可以解释。一方面，要求 `global` 表示已分配的变量可以防止意外的副作用。另一方面，如果所有全局引用都需要 `global`，那么你一直都在使用 `global`。你必须将对内置函数或导入模块的组件的每个引用声明为全局。这种杂乱会破坏 `global` 声明用于识别副作用的有用性。

2.2.3 为什么在具有不同值的循环中定义的 lambdas 都返回相同的结果？

假设你使用 for 循环来定义几个不同的 lambda（甚至是普通函数），例如：

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

这给你一个包含 5 个 lambdas 的列表，它们计算 x^2 。你可能会期望，当它们被调用时，它们将分别返回 0、1、4、9 和 16。但是，当你真正尝试时，你会看到它们都返回 16。：

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

发生这种情况是因为 x 不是 lambdas 的内部变量，而是在外部作用域中定义，并且在调用 lambda 时访问它 - 而不是在定义它时。在循环结束时， x 的值是 4，所以所有的函数现在返回 4^2 ，即 16。你还可以通过更改 x 的值来验证这一点，并查看 lambdas 的结果如何变化：

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

为了避免这种情况，你需要将值保存在 lambdas 的局部变量中，这样它们就不依赖于全局“ x ”的值

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

这里， $n=x$ 在 lambda 本地创建一个新的变量 n ，并在定义 lambda 时计算，使它具有与 x 在循环中该点相同的值。这意味着 n 的值在第一个 lambda 中为 0，在第二个 lambda 中为 1，在第三个中为 2，依此类推。因此每个 lambda 现在将返回正确的结果：

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

请注意，这种行为并不是 lambda 所特有的，但也适用于常规函数。

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配置模块；然后该模块可用作全局名称。因为每个模块只有一个实例，所以对模块对象所做的任何更改都会在任何地方反映出来。例如：

config.py:

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

mod.py:

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

main.py:

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


请注意，出于同样的原因，使用模块也是实现 Singleton 设计模式的基础。

2.2.5 导入模块的“最佳实践”是什么？

通常，不要使用 `from module_name import *`。这样做会使导入器的命名空间变得混乱，并且使得连接器更难以检测未定义的名称。

在文件的顶部导入模块。这样做可以清楚地了解代码所需的其他模块，并避免了模块名称是否在范围内的问题。每行导入一个模块可以轻松添加和删除导入的模块，但每行导入多个模块会占用更少的屏幕空间。

如果按以下顺序导入模块，这是一种很好的做法：

1. 标准库模块 -- 例如： `sys, os, getopt, re`
2. 第三方库模块（安装在 Python 的 `site-packages` 目录中的任何内容） -- 例如 `mx.DateTime, ZODB, PIL.Image` 等
3. 本地开发的模块

有时需要将模块导入语句移动到函数或类里面，以避免循环导入问题。Gordon McMillan 说：

当两个模块都使用“`import <module>`”的导入形式时，循环导入就可以了。但是当第 2 个模块想从第 1 个模块中获取一个名称（“`from module import name`”）并且导入位于顶层时，就会出错。这是因为第 1 个模块中的名称还不可用，因为第 1 个模块正在忙着导入第 2 个模块。

在这种情况下，如果第二个模块仅用于一个函数，则可以轻松地将模块导入语句移动到该函数中。调用导入时，第一个模块将完成初始化，第二个模块可以进行导入。

如果某些模块是特定于平台的，则可能还需要将模块导入语句移出顶级代码。在这种情况下，甚至可能无法导入文件顶部的所有模块。在这种情况下，在相应的特定于平台的代码中导入正确的模块是一个很好的选择。

只有当需要解决诸如避免循环导入或试图减少模块初始化时间的问题时，才可以将导入移动到本地范围，例如在函数定义中。如果根据程序的执行方式，许多导入是不必要的，这种技术尤其有用。如果仅在某个函数中使用模块，您还可能希望将导入移到该函数中。请注意，第一次加载模块可能会因为模块的一次初始化而代价高昂，但多次加载模块实际上是免费的，只需进行几次字典查找。即使模块名称超出了作用域，模块也可能在 `sys.modules` 中可用。

2.2.6 为什么对象之间会共享默认值？

这种类型的缺陷通常会惹恼新手程序员。考虑这个函数

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

第一次调用此函数时，`mydict` 包含一项。第二次，`mydict` 包含两项，因为当 `foo()` 开始执行时，`mydict` 中已经有一项了。

函数调用经常被期望为默认值创建新的对象。但实际情况并非如此。默认值会在函数定义时一次性地创建。如果对象发生改变，就如本示例中的字典那样，则对函数的后续调用将会引用这个被改变的对象。

按照定义，不可变对象例如数字、字符串、元组和 `None` 因为不可变所以是安全的。对可变对象例如字典、列表和类实例的改变则可能造成迷惑。

由于这一特性，在编程中应遵循的一项好习惯是不使用可变对象作为默认值。而应使用 `None` 作为默认值和函数中的值，检查值为 `None` 的形参并创建相应的列表、字典或其他可变对象。例如，不要这样写：

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

而要这样写:

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

这一特性有时会很有用处。当你有一个需要进行大量耗时计算的函数时，一个常见技巧是将每次调用函数的参数和结果值缓存起来，并在同样的值被再次请求时返回缓存的值。这称为“记忆”，具体实现方式可以是这样的:

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

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

你也可以使用包含一个字典的全局变量而不使用参数默认值；这完全取决于个人偏好。

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？

使用函数参数列表中的 * 和 ** 说明符收集参数; 这会将位置参数作为元组，将关键字参数作为字典。然后，您可以使用 * 和 ** 调用另一个函数时传递这些参数:

```
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 是指出现在函数定义中的名称，而实参 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何种类型的实参。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。但是，在调用 `func` 时，例如:

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

实际的值 42, 314 和 `somevar` 则是实参。

2.2.9 为什么更改列表'y' 也会更改列表'x'？

如果你编写的代码就像下面一样:

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


你可能想知道为什么追加一个元素也改变了 `x`。

产生这种结果有两个因素：

- 1) 变量只是指向具体对象的名称。执行 `y = x` 并不会为列表创建一个副本——它只是创建了一个新变量 `y` 指向 `x` 所指向的同一对象。这意味着只存在一个对象（列表），`x` 和 `y` 都是对它的引用。
- 2) 列表属于 *mutable* 对象，这意味着你可以改变它的内容。

在调用 `append()` 之后，这个可变对象的内容由 `[]` 变为 `[10]`。由于两个变量都指向同一对象，因此使用任何一个名称所访问到的都是修改后的值 `[10]`。

如果我们改为将不可变对象赋值给 `x`：

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

我们可以看到在此情况下 `x` 和 `y` 就不再相等了。这是因为整数是 *immutable* 对象，当我们执行 `x = x + 1` 时我们并不是改变了 5 这个对象的值；而是创建了一个新的对象（整数 6）并将其赋值给 `x`（也就是改变了 `x` 所指向的对象）。在赋值之后我们就有了两个对象（整数 6 和 5）以及分别指向它们的两个变量（`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，而看上去相似的另一一些操作（例如 `y = y + [10]` 和 `sorted(y)`）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。

但是，还存在一类操作，不同的类型执行相同的操作会有不同的行为：那就是增强赋值运算符。例如，`+=` 会原地改变列表，但不会改变元组或整数（`a_list += [1, 2, 3]` 与 `a_list.extend([1, 2, 3])` 一样都会改变 `a_list`，而 `some_tuple += (1, 2, 3)` 和 `some_int += 1` 则会创建新的对象）。

换言之：

- 如果我们有一个可变对象（`list`, `dict`, `set` 等等），我们可以使用某些特定的操作来改变它，所有指向它的变量都会显示它的改变。
- 如果我们有一个不可变对象（`str`, `int`, `tuple` 等等），所有指向它的变量都将显示相同样的值，但凡会改变这个值的操作将总是返回一个新对象。

如果你想知道两个变量是否指向相同的对象，你可以使用 `is` 运算符，或内置函数 `id()`。

2.2.10 如何编写带输出参数的函数（通过引用调用）？

请记住在 Python 中参数是通过赋值来传递的。由于赋值只是创建了对对象的引用，因此在调用者和被调用者的参数名称之间没有别名，所以本身是没有按引用调用的。你可以通过多种方式实现所需的效果。

- 1) 通过返回一个结果元组：

```
def func2(a, b):
    a = 'new-value'          # a and b are local names
    b = b + 1                 # assigned to new objects
    return a, b              # return new values

x, y = 'old-value', 99
x, y = func2(x, y)
print(x, y)                  # output: new-value 100
```

这几乎总是最清晰明了的解决方案。

- 2) 通过使用全局变量。这种方式不是线程安全的，而且也不受推荐。
- 3) 通过传递一个可变(即可原地修改的)对象:

```
def func1(a):
    a[0] = 'new-value'      # 'a' references a mutable list
    a[1] = a[1] + 1         # changes a shared object

args = ['old-value', 99]
func1(args)
print(args[0], args[1])    # output: new-value 100
```

- 4) 通过传递一个会被改变的字典:

```
def func3(args):
    args['a'] = 'new-value'  # args is a mutable dictionary
    args['b'] = args['b'] + 1 # change it in-place

args = {'a': 'old-value', 'b': 99}
func3(args)
print(args['a'], args['b'])
```

- 5) 或者在一个类实例中捆绑值:

```
class callByRef:
    def __init__(self, **args):
        for (key, value) in args.items():
            setattr(self, key, value)

def func4(args):
    args.a = 'new-value'      # args is a mutable callByRef
    args.b = args.b + 1       # change object in-place

args = callByRef(a='old-value', b=99)
func4(args)
print(args.a, args.b)
```

几乎没有任何适当理由将问题如此复杂化。

你的最佳选择是返回一个包含多个结果的元组。

2.2.11 如何在 Python 中创建高阶函数？

你有两种选择：使用嵌套作用域，或者使用可调用对象。例如，假设你想要定义 `linear(a,b)` 使其返回一个函数 `f(x)` 来设计 `a*x+b` 的值。可以使用以下嵌套作用域：

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

或使用一个可调用对象：

```
class linear:
    def __init__(self, a, b):
        self.a, self.b = a, b

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

在两种情况下，：

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

都会给出一个可调用对象，使得 `taxes(10e6) == 0.3 * 10e6 + 2`。

可调用对象方式的缺点是速度略慢且生成的代码略长。但是，请注意一组可调用对象能够通过继承来共享签名：

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

对象可以封装多个方法的状态：

```
class counter:

    value = 0

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

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

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

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

这里 `inc()`、`dec()` 和 `reset()` 将表现为共享同一计数变量的多个函数。

2.2.12 如何在 Python 中复制对象？

一般来说，通常情况下请尝试 `copy.copy()` 或 `copy.deepcopy()`。不是所有对象都可以复制，但多数都是可以的。

某些对象可以方便地复制。例如字典具有 `copy()` 方法：

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

序列可以通过切片来复制：

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

2.2.13 如何找到对象的方法或属性？

对于一个用户自定义类的实例 `x`，`dir(x)` 将返回一个按字母顺序排序的包含实例属性和方法及其类所定义的属性名称的列表。

2.2.14 我的代码如何才能发现对象的名称？

通常来说是做不到的，因为对象并不真正具有名称。在本质上，赋值总是会将一个名称绑定到某个值；`def` 和 `class` 语句也是如此，但在这种情况下该值是一个可调用对象。考虑以下代码：

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

不严谨地讲，该类有一个名称：虽然它是绑定了两个名称并通过名称 `B` 发起调用，所创建的实例仍然被视为类 `A` 的一个实例。但是实例的名称则无法确定地说是 `a` 或是 `b`，因为有两个名称被绑定到了同一个值。

一般来说你的代码应该没有必要“知道”特定值的名称。除非你是在编写特殊的内省程序，出现这样的问题通常表明如果改变方式可能会更有利。

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：

跟你找出在你家门廊见到的某只猫的名字所用的办法一样：猫（对象）自己无法告诉你它的名字，它根本就不在乎——所以找出它叫什么名字的唯一办法是问你的所有邻居（命名空间）那是不是他们的猫（对象）……

……并且如果你发现它有很多名字或根本没有名字也不必觉得惊讶！

2.2.15 逗号运算符的优先级是什么？

逗号在 Python 中不是运算符。考虑这个例子：

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

由于逗号不是运算符而是表达式之间的分隔符，以上代码的含义就相当于：

```
("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。它们并不是真正的运算符而是赋值语句中的语法分隔符。

2.2.16 是否有与 C 的“?:” 三目运算符等价的东西？

有的。相应语法如下：

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

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

在 Python 2.5 引入此语法之前，常见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` 具有布尔假值时将会给出错误的结果。所以，使用 `... if ... else ...` 形式总是会更好。

2.2.17 是否可以用 Python 编写混淆的单行程序？

可以。通常是在 `lambda` 中嵌套 `lambda` 来实现的。请参阅以下三个来自 Ulf Bartelt 的示例代码：

```
from functools import reduce

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

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

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

请不要在家里尝试，骚年！

2.2.18 函数参数列表中的斜杠 (/) 是什么意思？

函数参数列表中的斜杠表示在它之前的形参是仅限位置形参。仅限位置形参没有外部可用的名称。在调用接受仅限位置形参的函数时，参数只会基于它们的位置被映射到形参。例如，`pow()` 是一个接受仅限位置形参的函数。它的文档是这样的：

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

pow(x, y, z=None, /)
    Equivalent to x**y (with two arguments) or x**y % z (with three arguments)

    Some types, such as ints, are able to use a more efficient algorithm when
    invoked using the three argument form.
```

在形参列表末尾的斜杠意味着所有三个形参都是仅限位置形参。因此，附带关键字参数调用 `pow()` 将会导致报错：

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

请注意目前此特性只记录于文档而不是有效的 Python 语法，不过已经有 [PEP 570](#) 提出了 Python 中使用仅位置形参的语法。

2.3 数字和字符串

2.3.1 如何指定十六进制和八进制整数？

要指定一个八进制数码，则在八进制值之前加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），就输入：

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

十六进制数也同样简单。只要在十六进制数之前加一个零和一个小写或大写字母“x”。十六进制数码中的字母可以为大写或小写。例如在 Python 解释器中输入：

```
>>> a = 0xa5
>>> a
165
>>> b = 0xb2
>>> b
178
```

2.3.2 为什么 `-22 // 10` 返回 `-3`？

这主要是为了让 `i % j` 的正负与 `j` 一致，如果你想要这样的结果，并且又想要：

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

那么整除就必须向下取整。C 同样要求保持一致，并且编译器在截短 `i // j` 的结果值时需要使 `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` 是有用的；`-190 % 12 == -10` 则是会导致意外的漏洞。

2.3.3 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` 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。

如果你只是想将字符串转为数字，请不要使用内置函数 `eval()`。`eval()` 的速度会慢很多并且有安全风险：别人可能会传入具有你不想要的附带效果的 Python 表达式。例如，别人可以传入 `__import__('os').system("rm -rf $HOME")` 这将删除你的家目录。

`eval()` 还具有将数字解读为 Python 表达式的效果，这样 `eval('09')` 将会导致语法错误，因为 Python 不允许十进制数的首位是“0”（“0”除外）。

2.3.4 如何将数字转换为字符串？

例如要将数字 144 转换为字符串'144'，可使用内置类型构造器 `str()`。如果想要表示为十六进制或八进制数，可使用内置函数 `hex()` 或 `oct()`。想要更好地格式化，请参阅 `f-strings` 和 `formatstrings` 等小节，例如 `"{:04d}".format(144)` 生成 '0144' 而 `"{:0.3f}".format(1.0/3.0)` 生成 '0.333'。

2.3.5 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，你应该使用你想要的各种部分来构造一个新字符串。但是，如果你想要一个可以原地修改 `Unicode` 数据的对象，可尝试使用 `io.StringIO` 对象或 `array` 模块：

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

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

2.3.6 如何使用字符串调用函数/方法？

有多种技巧可供选择。

- 最好的做法是使用一个将字符串映射到函数的字典。这一技巧的主要优势在于字符串不必与函数名称一致。这也是用于模拟其他语言中 `case` 结构的主要技巧：

```
def a():
    pass

def b():
    pass

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

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

- 使用内置函数 `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。

在标准库中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()
```

- 使用 `locals()` 或 `eval()` 来解析出函数名:

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

fname = "myFunc"

f = locals()[fname]
f()

f = eval(fname)
f()
```

注意: 使用 `eval()` 速度慢而且危险。如果你不能绝对掌控字符串的内容, 别人将能传入可被解析为任意函数直接执行的字符串。

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

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

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

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

2.3.8 是否有 `scanf()` 或 `sscanf()` 的对应物?

没有这样的对应物。

对于简单的输入解析, 最方便的做法通常是使用字符串对象的 `split()` 方法将一行内容拆解为以空格分隔的单词, 然后使用 `int()` 或 `float()` 将表示十进制数的字符串转换为数值。`split()` 支持可选的“sep”形参, 适用于内容行使用空格符以外的分隔符的情况。

对于更复杂的输入解析, 正则表达式会比 C 的 `sscanf()` 更强大, 也更适合此类任务。

2.3.9 'UnicodeDecodeError' 或 'UnicodeEncodeError' 错误是什么意思？

见 [unicode-howto](#)

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。首先，下面列出了深入了解前需要记住的事情：

- 不同的 Python 实现具有不同的性能特点。本 FAQ 着重解答的是 *CPython*。
- 行为可能因操作系统而异，尤其是在谈论 I/O 或多线程时。
- 在尝试优化任何代码前，应始终找到程序中的热点（请参阅 `profile` 模块）。
- 编写基准脚本将允许您在搜索改进时快速迭代（请参阅 `timeit` 模块）。
- 强烈建议在可能引入隐藏在复杂优化中的回归之前，要有良好的代码覆盖率（通过单元测试或任何其他技术）。

话虽如此，加速 Python 代码有很多技巧。以下是一些可以达到可接受的性能水平的一般原则：

- 使您的算法更快（或更改为更快的算法）可以产生比尝试在代码中使用微优化技巧更大的好处。
- 使用正确的数据结构。参考文档 `builtin-types` 和 `collections` 模块。
- 当标准库提供用于执行某些操作的原语时，可能（尽管不能保证）比您可能提出的任何替代方案更快。对于用 C 编写的原语，例如内置函数和一些扩展类型，这是真的。例如，请确保使用 `list.sort()` 内置方法或相关的 `sorted()` 函数进行排序（有关适度高级用法的示例，请参阅 `sortinghowto`）。
- 抽象倾向于创造间接性并迫使翻译更多地工作。如果间接级别超过完成的有用工作量，则程序将变慢。你应该避免过度抽象，特别是在微小的功能或方法的形式下（这通常也会对可读性产生不利影响）。

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* 可以将稍微修改的 Python 代码版本编译为 C 扩展，并且可以在许多不同的平台上使用。*Cython* 可以利用编译（和可选的类型注释）来使代码明显快于解释运行时的速度。如果您对 C 编程技能有信心，也可以自己编写 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，因此将多个字符串连接在一起效率很低，因为每个连接都会创建一个新对象。在一般情况下，总运行时间是总字符串长度的二次方。

要连接多个 `str` 对象，通常推荐的用法是将它们放入一个列表中并在结尾处调用 `str.join()`：

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

（另一个合理有效的惯用方法是 `io.StringIO`）

要连接多个 `str` 对象，建议使用本地连接（`+=` 运算符）扩展 `bytearray` 对象：

```
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）转换为具有相同排列顺序的相同条目的元组。

例如, `tuple([1, 2, 3])` 产生 `(1, 2, 3)` 而 `tuple('abc')` 产生 `('a', 'b', 'c')`。如果参数为一个元组，它不会创建副本而是返回同一对象，因此如果你不确定某个对象是否为元组时也可简单地调用 `tuple()`。

类型构造器 `list(seq)` 可将任意序列或可迭代对象转换为具有相同排列顺序的相同条目的列表。例如, `list((1, 2, 3))` 产生 `[1, 2, 3]` 而 `list('abc')` 产生 `['a', 'b', 'c']`。如果参数为一个列表，它会像 `seq[:]` 那样创建一个副本。

2.5.2 什么是负数序号？

Python 序列使用正数或负数作为序号或称索引号。对于正数序号，第一个序号为 0 而 1 为第二个序号，依此类推。对于负数序号，倒数第一个序号为 -1 而倒数第二个序号为 -2，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 如何以相反的顺序迭代序列？

使用 `reversed()` 内置函数，这是 Python 2.4 中的新功能：

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

这不会修改您的原始序列，而是构建一个反向顺序的新副本以进行迭代。

在 Python 2.3 里，您可以使用扩展切片语法：

```
for x in sequence[::-1]:
    ... # do something with x ...
```

2.5.4 如何从列表中删除重复项？

有关执行此操作的许多方法的详细讨论，请参阅 Python Cookbook：

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

如果您不介意重新排序列表，请对其进行排序，然后从列表末尾进行扫描，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]:
```

(下页继续)

(续上页)

```

del mylist[i]
else:
    last = mylist[i]

```

如果列表的所有元素都可以用作设置键（即：它们都是`hashable`），这通常会更快：

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

这会将列表转换为集合，从而删除重复项，然后返回到列表中。

2.5.5 如何在 Python 中创建数组？

使用列表：

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

列表在时间复杂度方面相当于 C 或 Pascal 数组；主要区别在于，python 列表可以包含许多不同类型的对象。

`array` 模块还提供了创建具有紧凑表示的固定类型的数组的方法，但它的索引速度比列表慢。还要注意，数字扩展和其他扩展还定义了具有各种特性的类似数组的结构。

要获取 Lisp 样式的列表，可以使用元组模拟 `cons` 单元：

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

如果需要可变性，可以使用列表而不是元组。这里模拟 lisp car 的是 `lisp_list[0]`，模拟 cdr 的是 `lisp_list[1]`。只有在你确定真的需要的时候才这样做，因为它通常比使用 Python 列表慢得多。

2.5.6 如何创建多维列表？

你可能试图制作一个像这样的多维数组：

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

如果你打印它，看起来是正确的：

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

但是，当你给某一项赋值时，会同时在多个位置显示变化：

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

其中的原因在于使用 `*` 对列表执行重复操作并不是创建副本，它只是创建现有对象的引用。`*3` 创建了对长度为二的同一列表的 3 个引用。对某一行的改变会作用于所有行，通常这一定不是你所希望的。

建议的做法是先创建一个所需长度的列表，然后其中的元素再以一个新创建的列表来填充：

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

这样就生成了一个包含 3 个长度为二的不同列表的列表。你也可以使用列表推导式：

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

或者你还可以使用提供矩阵类型的扩展包；其中最著名的是 [NumPy](#)。

2.5.7 如何将方法应用于一系列对象？

可以使用列表推导式：

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

2.5.8 为什么 `a_tuple[i] += 'item'` 会在执行加法时引发异常？

这是由两个事实共同导致的结果，一是增强赋值运算符属于赋值运算符，二是在 Python 中存在可变和不可变两种不同的对象。

此处的讨论在任何对元组中指向可变对象的元素使用增强赋值运算符的情况都是普遍成立的，但在此我们只以 list 和 += 来举例。

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
```

发生异常的原因是显而易见的：1 会与对象 `a_tuple[0]` 相加，而该对象为 (1)，得到结果对象 2，但我们试图将运算结果 2 赋值给元组的 0 号元素时就将报错，因为我们不能改变元组的元素所指向的对象。

在表层之处，以上增强赋值语句所做的大致是这样：

```
>>> 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，因此操作的赋值部分会引发错误。

当你这样写的时候：

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

发生异常会令人略感吃惊，还有一个更为令人吃惊的事实：虽然有报错，但是添加操作却生效了：

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

要明白为何会这样，你需要知道 (a) 如果一个对象实现了 `__iadd__` 魔术方法，它会在执行 += 增强赋值时被调用，并且其返回值将用于该赋值语句；(b) 对于列表来说，`__iadd__` 等价于在列表上调用 `extend` 并返回该列表。因此对于列表我们可以说 += 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，在我们的元组示例中，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__` 成功执行，因此列表得到了扩充，但是虽然 `result` 指向了 `a_tuple[0]` 已经指向的同一对象，最后的赋值仍然导致了报错，因为元组是不可变的。

2.5.9 我想做一个复杂的排序：你能用 Python 做一个 Schwartzian 变换吗？

该技术归功于 Perl 社区的 Randal Schwartz，它通过将每个元素映射到其“排序值（sort value）”的度量对列表中的元素进行排序。在 Python 中，使用 `list.sort()` 方法的 `key` 参数：

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

2.5.10 如何按其他列表中的值对一个列表进行排序？

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']
```

最后一步的替代方案是：

```
>>> result = []
>>> for p in pairs: result.append(p[1])
```

如果你觉得这个更容易读懂，那么你可能更喜欢使用这个而不是前面的列表推导。然而，对于长列表来说，它的速度几乎是原来的两倍。为什么？首先，`append()` 操作必须重新分配内存，虽然它使用了一些技巧来避免每次都这样做，但它仍然偶尔需要这样做，而且代价相当高。第二，表达式“`result.append`”需要额外的属性查找。第三，必须执行所有这些函数调用会降低速度。

2.6 对象

2.6.1 什么是类？

类是通过执行类语句创建的特定对象类型。类对象被当作模板来创建实例对象，实例对象包含了特定于数据类型的数据（属性）和代码（方法）。

类可以基于一个或多个的其他类，称之为基类（ES），它继承基类的属性和方法，这样就可以通过继承来连续地细化对象模型。例如：您可能有一个 Mailbox 类提供邮箱的基本访问方法，它的子类 MboxMailbox, MaildirMailbox, OutlookMailbox 用于处理各种特定邮箱格式。

2.6.2 什么是方法？

方法实际上就是类定义中的函数。对于某个对象 `x` 上的方法，通常称为 `x.name(arguments...)`。

```
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常规名称。例如：对于某个类的某个实例 `x`，其方法 `meth(self, a, b, c)` 实际上应该被称为 `x.meth(a, b, c)`；对于被调用的方法会被称为 `meth(x, a, b, c)`。

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？

可使用内置函数 `isinstance(obj, cls)`。你可以提供一个元组而不是单个类来检查某个对象是否为任意多个类当中某一个类的实例，例如 `isinstance(obj, (class1, class2, ...))`，也可以检查某个对象是否为 Python 内置类型当中某一个类型的对象，例如 `isinstance(obj, str)` 或 `isinstance(obj, (int, float, complex))`。

请注意大多数程序不会经常对用户自定义类使用 `isinstance()`。如果是你自己开发的类，更正确的面向对象风格是在类中定义方法来封装特定的行为，而不是检查对象的类并根据它属于什么类来做不同的事。例如，如果你有一个执行某些操作的函数：

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

更好的方法是在所有类上定义一个 `search()` 方法，然后调用它：

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

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

obj.search()
```

2.6.5 什么是委托？

委托是一种面向对象的技巧（也称为设计模式）。假设您有一个对象 `x` 并且想要改变其中一个方法的行为。您可以创建一个新类，它提供您感兴趣的方法的新实现，并将所有其他方法委托给 `x` 的相应方法。

Python 程序员可以轻松实现委托。例如，以下类实现了一个类，该类的行为类似于文件，但将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)
```

在这里 `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__` 方法来完成的；请参阅 [语言参考](#) 了解有关控制属性访问的更多信息。

请注意对于更一般的情况来说，委托可能包含更多细节问题。当某些属性既需要读取又需要设置时，类还必须定义 `__setattr__()` 方法，并且这样做必须小心谨慎。`__setattr__()` 的基本实现大致相当于以下代码：

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

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

2.6.6 如何从覆盖基类的派生类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(Derived, self).meth()
```

对于 Python 3.0 之前的版本，您可能正在使用经典类：对于诸如 `class Derived(Base): ...` 之类的类定义，可以将 `Base`（或 `Base` 中的一个的基类）中定义的方法 `meth()` 调用为 `Base.meth(self, arguments...)`。这里，`Base.meth` 是一个未绑定的方法，因此您需要提供 `self` 参数。

2.6.7 如何组织代码以便更改基类？

可以为基类定义别名，在类定义之前为其分配实际基类，并在整个类中使用别名。然后更改分配给别名的值，就能实现上述要求。顺便提一下，如果你想动态决定（例如，取决于资源的可用性）要使用哪个基类，这个技巧也很方便。例如：

```
BaseAlias = <real base class>

class Derived(BaseAlias):
    def meth(self):
```

(下页继续)

(续上页)

```
BaseAlias.meth(self)
...
```

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（在 C++ 或 Java 的意义上）。

对于静态数据，只需定义一个类属性。要为属性分配新值，就必须在赋值中显式使用类名：

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

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

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

对于任意 `c` 来说只要 `isinstance(c, C)` 为真，则 `c.count` 同样也指向 `C.count`，除非被 `c` 自身，或者从 `c.__class__` 回到 `C` 的基类搜索路径上的某个类所重载。

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
```

静态方法是可行的：

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

然而，获得静态方法效果的更直接的方法是通过一个简单的模块级函数：

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

如果您的代码是结构化的，以便为每个模块定义一个类（或紧密相关的类层次结构），那么这就提供了所需的封装。

2.6.9 如何在 Python 中重载构造函数（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在构造函数的上下文中。

在 C++ 中，你会这样写

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

在 Python 中，您必须编写一个构造函数，使用默认参数捕获所有情况。例如：


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

这不完全等同，但在实践中足够接近。

你也可以尝试一个可变长度的参数列表，例如：

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

相同的方法适用于所有方法定义。

2.6.10 我尝试使用 `__spam`，但是得到一个关于 `_SomeClassName__spam` 的错误信息。

以双下划线打头的变量会被“更名”以提供一种定义类私有变量的简单而有效的方式。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）文本会被替换为 `_classname__spam`，其中 `classname` 为去除了全部前缀下划线的当前类名称。

这并不能保证私密性：外部用户仍然可以访问“`_classname__spam`”属性，私有变量值也在对象的 `__dict__` 中可见。许多 Python 程序员从来都不使用这种私有变量名称。

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` 语句不一定调用 `__del__()` ——它只是减少对象的引用计数，如果（引用计数）达到零，才会调用 `__del__()`。

如果数据结构包含循环链接（例如，每个子级都有一个父级引用，每个父级都有一个子级列表的树），则引用计数将永远不会返回零。尽管 Python 偶尔会运行一个算法来检测这样的循环，但在数据结构的引用计数清零后，垃圾收集器可能需要一段时间来运行，因此 `__del__()` 方法可能会在不方便和随机的时间被调用。这对于重现一个问题，是非常不方便的。更糟糕的是，对象 `__del__()` 的方法执行顺序是任意的。虽然可以运行 `gc.collect()` 来强制回收，但在一些病态的情况下，对象永远不会被回收。

尽管有循环收集器，但在对象上定义一个显式的 `close()` 方法以便在用完之后调用它仍然是一个好主意。这样 `close()` 方法可以随即删除引用子对象的属性。不要直接调用 `__del__()` ——应该由 `__del__()` 调用 `close()`，并且 `close()` 能确保可以被同一对象多次地调用。

另一种避免循环引用的方法是使用 `weakref` 模块，该模块允许您指向对象而不增加其引用计数。例如，树状数据结构应该对其父级和同级引用使用弱引用（如果需要的话！）

最后，如果 `__del__()` 方法引发异常，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 不跟踪类（或内置类型）的所有实例。您可以对类的构造函数进行编程，以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` 属于之前创建的不同整数对象，并在执行 `id()` 调用后立即删除。要确保要检查其 `id` 的对象仍处于活动状态，请创建对该对象的另一个引用：

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

2.7 模块

2.7.1 如何创建 `.pyc` 文件？

当一个模块首次被导入时（或自当前已编译文件创建后源文件被修改时），将会在对应该 `.py` 文件所在目录的 `__pycache__` 子目录下创建一个包含已编译代码的 `.pyc` 文件。该 `.pyc` 文件的文件名的开头部分将与对应 `.py` 文件名相同，并以 `.pyc` 为后缀，中间部分则是基于创建它的特定 Python 二进制代码版本。（详情参见 [PEP 3147](#)。）

无法创建 `.pyc` 文件的可能原因是包含源文件的目录存在权限问题，这意味着 `__pycache__` 子目录无法被创建。举例来说，如果你以某一用户来开发程序但以另一用户身份来运行程序时就可能发生问题，测试 Web 服务器就属于这种情况。

除非设置了 `PYTHONDONTWRITEBYTECODE` 环境变量，否则当你导入模块并且 Python 具有创建 `__pycache__` 子目录并将已编译模块写入该子目录的能力（权限、存储空间等等）时就会自动创建 `.pyc` 文件。

在最高层级运行的 Python 脚本不被视为导入，因此不会创建 `.pyc` 文件。例如，如果你有一个最高层级模块文件 `foo.py`，它又导入了另一个模块 `xyz.py`，当你运行 `foo` 模块（通过输入终端命令 `python foo.py`），则将为 `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。

如果你需要为 `foo` 创建 `.pyc` 文件——即为不是被导入的模块创建 `.pyc` 文件——你可以使用 `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` 来重载该行为）。

你还可以使用 `compileall` 模块自动编译一个目录或多个目录下的所有文件。具体做法可以在命令行提示符中运行 `compileall.py` 并提供包含要编译 Python 文件的目录路径：

```
python -m compileall .
```

2.7.2 如何找到当前模块名称？

模块可以通过查看预定义的全局变量 `__name__` 找到自己的模块名称。如果它的值为 `'__main__'`，程序将作为脚本运行。通常，通过导入使用的许多模块也提供命令行界面或自检，并且只在检查 `__name__` 之后，才执行之后的代码：

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

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

2.7.3 如何让模块相互导入？

假设您有以下模块：

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
- 创建用于 foo 的空全局变量
- foo 被编译并开始执行
- foo 导入 bar
- 创建了用于 bar 的空全局变量
- bar 被编译并开始执行
- bar 导入 foo（这是一个空操作（no-op），因为已经有一个名为 foo 的模块）
- bar.foo_var = foo.foo_var

最后一步失败了，因为 Python 还没有解释 foo，而 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能使用常量或内置函数。这意味着导入模块中的所有内容都被引用为 `<module>.<name>`。

Jim Roskind 建议在每个模块中按以下顺序执行步骤：

- 导出（全局变量，函数和不需要导入基类的类）
- 导入声明
- 活动代码（包括从导入值初始化的全局变量）。

van Rossum 不喜欢这种方法，因为导入出现在一个陌生的地方，但这种方法确实有效。

Matthias Urlichs 建议重构代码，以便首先不需要递归导入。

这些解决方案并不相互排斥。

2.7.4 `__import__('x.y.z')` 返回 `<module 'x'>`; 如何获取 `z`?

考虑使用 `importlib` 中的函数 `import_module()` :

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

2.7.5 当我编辑了导入过的模块并重新导入它时，这些变化没有显示出来。为什么会这样？

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。如果不这么做，在一个由许多模块组成的程序中，每个模块都会导入相同的基本模块，那么基本模块将被解析和重新解析多次。要强制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：这种技术不是 100% 万无一失。特别是包含如下语句的模块

```
from modname import some_objects
```

将继续使用旧版本的导入对象。如果模块包含类定义，则不会更新现有的类实例以使用新的类定义。这可能导致以下矛盾行为：

```
>>> 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
```

如果打印出类对象的“标识”，问题的本质就会明确：

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

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学会并喜欢上这个功能。

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 程序员有时会长时间盯着它，想知道为什么即使 `x > y`，`y` 也在减少。

因为没有开始/结束括号，所以 Python 不太容易发生编码式冲突。在 C 中，括号可以放到许多不同的位置。如果您习惯于阅读和编写使用一种风格的代码，那么在阅读（或被要求编写）另一种风格时，您至少会感到有些不安。

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。这不仅仅是由于缺少开始/结束括号 -- 缺少声明和高级数据类型也是其中的原因 -- 但缩进基于语法肯定有帮助。

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:

```
1.001100110011001100110011001100110011001100110011001100110011 (binary)
```

它对应于十进制数值:

```
1.1999999999999999555910790149937383830547332763671875 (decimal)
```

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，这意味着如果要显式引用或从特定类调用该方法，不需要特殊语法。在 C++ 中，如果你想使用在派生类中重写基类中的方法，你必须使用 :: 运算符 -- 在 Python 中你可以编写 baseclass.methodname(self, <argument list>)。这对于 __init__() 方法非常有用，特别是在派生类方法想要扩展同名的基类方法，而必须以某种方式调用基类方法时。

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？

许多习惯于 C 或 Perl 的人抱怨，他们想要使用 C 的这个特性：

```
while (line = readline(f)) {
    // do something with line
}
```

但在 Python 中被强制写成这样：

```
while True:
    line = f.readline()
    if not line:
        break
    ... # do something with line
```

不允许在 Python 表达式中赋值的原因是这些其他语言中常见的、很难发现的错误，是由这个结构引起的：

```
if (x = 0) {
    // error handling
}
else {
    // code that only works for nonzero x
}
```

错误是一个简单的错字：`x = 0`，将 0 赋给变量 `x`，而比较 `x == 0` 肯定是可以预期的。

已经有许多替代方案提案。大多数是为了少打一些字的黑客方案，但使用任意或隐含的语法或关键词，并不符合语言变更提案的简单标准：它应该直观地向尚未被介绍到这一概念的人类读者提供正确的含义。

一个有趣的现象是，大多数有经验的 Python 程序员都认识到 `while True` 的习惯用法，也不太在意是否能在表达式构造中赋值；只有新人表达了强烈的愿望希望将其添加到语言中。

有一种替代的拼写方式看起来很有吸引力，但通常不如“`while True`”解决方案可靠：

```
line = f.readline()
while line:
    ... # do something with line...
    line = f.readline()
```

问题在于，如果你改变主意（例如你想把它改成 `sys.stdin.readline()`），如何知道下一行。你必须记住改变程序中的两个地方 -- 第二次出现隐藏在循环的底部。

最好的方法是使用迭代器，这样能通过 `for` 语句来循环遍历对象。例如 *file objects* 支持迭代器协议，因此可以简单地写成：

```
for line in f:
    ... # do something with line...
```


3.7 为什么 Python 对某些功能（例如 `list.index()`）使用方法来实现在，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` 块的效率极高。实际上捕获异常是昂贵的。在 2.0 之前的 Python 版本中，通常使用这个习惯用法：

```
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？

你可以通过一系列 `if... elif... elif... else` 轻松完成这项工作。对于 `switch` 语句语法已经有了一些建议，但尚未就是否以及如何进行范围测试达成共识。有关完整的详细信息和当前状态，请参阅 [PEP 275](#)。

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：

```

def function_1(...):
    ...

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：

```

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。

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 表单不同，Python 的 lambdas 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 中的第一类对象，可以在本地范围内声明。因此，使用 lambda 而不是本地定义的函数的唯一优点是你不需要为函数创建一个名称 -- 这只是一个分配了函数对象 (与 lambda 表达式生成的对象类型完全相同) 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 将带有可选注释的 Python 修改版本编译到 C 扩展中。Nuitka 是一个将 Python 编译成 C++ 代码的新兴编译器，旨在支持完整的 Python 语言。要编译成 Java，可以考虑 VOC。

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。

但是，其他实现 (如 Jython 或 PyPy)，) 可以依赖不同的机制，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一些微妙的移植问题。

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 的引用计数和析构函数方案，每个新赋值的 *f* 都会关闭前一个文件。然而，对于传统的 GC，这些文件对象只能以不同的时间间隔 (可能很长的时间间隔) 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 也成为一个问题。在独立的 Python 中，可以用 GC 库提供的版本替换标准的 malloc() 和 free()，嵌入 Python 的应用程序可能希望用它自己替代 malloc() 和 free()，而可能不需要 Python 的。现在，CPython 可以正确地实现 malloc() 和 free()。

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？

虽然列表和元组在许多方面是相似的，但它们的使用方式通常是完全不同的。可以认为元组类似于 Pascal 记录或 C 结构；它们是相关数据的小集合，可以是不同类型的数据，可以作为一个组进行操作。例如，笛卡尔坐标适当地表示为两个或三个数字的元组。

另一方面，列表更像其他语言中的数组。它们倾向于持有不同数量的对象，所有对象都具有相同的类型，并且逐个操作。例如，`os.listdir('.')` 返回表示当前目录中的文件的字符串列表。如果向目录中添加了一两个文件，对此输出进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 `hash` 代码。`hash` 代码根据键和每个进程的种子而变化很大；例如，“Python”的 `hash` 值为 -539294296，而“python”（一个按位不同的字符串）的 `hash` 值为 1142331976。然后，`hash` 代码用于计算内部数组中将存储该值的位置。假设您存储的键都具有不同的 `hash` 值，这意味着字典需要恒定的时间 -- $O(1)$ ，用 Big-O 表示法 -- 来检索一个键。

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。

如果需要，可以使用以下方法来解决这个问题，但使用它需要你自担风险：你可以将一个可变结构包装在一个类实例中，该实例同时具有 `__eq__()` 和 `__hash__()` 方法。然后，你必须确保驻留在字典（或其他基于 `hash` 的结构）中的所有此类包装器对象的哈希值在对象位于字典（或其他结构）中时保持固定：

```
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`，只要包装器对象在字典中，包装列表就不能更改以避免异常。除非你准备好认真考虑需求以及不正确地满足这些需求的后果，否则不要这样做。请留意。

3.21 为什么 `list.sort()` 没有返回列表？

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):
    ... # do whatever with mydict[key]...
```

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，通过对组件进行适当的测试规程，可以获得接口规范的许多好处。还有一个工具 `PyChecker`，可用于查找由于子类化引起的问题。

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。

适当的测试规程可以帮助在 Python 中构建大型的、复杂的应用程序以及接口规范。事实上，它可能会更好，因为接口规范不能测试程序的某些属性。例如，`append()` 方法将向一些内部列表的末尾添加新元素；接口规范不能测试您的 `append()` 实现是否能够正确执行此操作，但是在测试套件中检查这个属性是很简单的。

编写测试套件非常有用，您可能希望设计代码时着眼于使其易于测试。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 允许您草率行事，根本不编写测试用例。

3.23 为什么没有 goto？

可以使用异常捕获来提供“goto 结构”，甚至可以跨函数调用工作的。许多人认为异常捕获可以方便地模拟 C，Fortran 和其他语言的“go”或“goto”结构的所有合理用法。例如：

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

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

但是不允许你跳到循环的中间，这通常被认为是滥用 goto。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") # works fine!
```

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 有一个‘with’语句，它封装了块的执行，在块的入口和出口调用代码。有些语言的结构是这样的：

```
with obj:
    a = 1                # equivalent 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 中并没有告诉解释器这一点。假设“a”是整数，会发生什么？如果有一个名为“x”的全局变量，它是否会在 with 块中使用？如您所见，Python 的动态特性使得这样的选择更加困难。

然而，Python 可以通过赋值轻松实现“with”和类似语言特性（减少代码量）的主要好处。代替：

```
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副作用，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。

3.26 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。

3.27 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], # last trailing comma is optional but good style
}
```

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。

4.1 通用的代码库问题

4.1.1 如何找到可以用来做 XXX 的模块或应用？

在代码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](#) 或是 [Google](#) 等其他搜索引擎。用“Python”加上一两个你需要的关键字通常会找到有用的东西。

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 `SVSV 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。

对于 Windows 平台：使用 `consolelib` 模块。

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()` 函数类似的注册函数。

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)
```

因此函数应该定义两个参数：

```
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。

一旦你的程序已经用函数和类完善地组织起来，你就应该编写测试函数来测试其行为。可以使用自动执行一系列测试函数的测试集与每个模块进行关联。听起来似乎需要大量的工作，但是因为 Python 非常简洁和灵活，所以实际上会相当简单。在编写“生产代码”的同时别忘了也要编写测试函数，你会发现编程会变得更愉快、更有趣，因为这样会使得发现 bug 和设计缺陷更加容易。

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 Python 源码中的 docstring 创建 HTML 文件。也可以使用 `epydoc` 来只通过 docstring 创建 API 文档。`Sphinx` 也可以引入 docstring 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。

Aahz 的非常实用的 `threading` 教程中有一些幻灯片；可以参阅 <http://www.pythoncraft.com/OSCON2001/>。

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))
    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？

最简单的方法是使用新的 `concurrent.futures` 模块，尤其是其中的 `ThreadPoolExecutor` 类。

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

这是一个简单的例子：

```
import threading, queue, time

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

```

(下页继续)

(续上页)

```

        break
    else:
        print('Worker', threading.currentThread(), end=' ')
        print('running with argument', arg)
        time.sleep(0.5)

# Create queue
q = queue.Queue()

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

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

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

```

运行时会产生如下输出：

```

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

```

(下页继续)

(续上页)

```
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
```

覆盖其他对象的操作会在其他对象的引用计数变成 0 时触发其 `__del__()` 方法，这可能会产生一些影响。对字典和列表进行大量操作时尤其如此。如果有疑问的话，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 Python 1.5 时代，Greg Stein 开发了一个完整的补丁包（“free threadings”补丁），移除了 GIL，并用粒度更合适的锁来代替。Adam Olsen 最近也在他的 [python-safethread](#) 项目里做了类似的实验。不幸的是，由于为了移除 GIL 而使用了大量细粒度的锁，这两个实验在单线程测试中的性能都有明显的下降（至少慢 30%）。

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

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

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

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

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

4.4 输入输出

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 模块有一个 `copyfile()` 函数。注意在 MacOS 9 中不会复制 resource fork 和 Finder info。

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。

注解：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, POSIX (Linux, BSD 等), Jython:

<http://pyserial.sourceforge.net>

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>。

Cameron Laird 维护了一份关于 Python web 技术的实用网页的集合，见 http://phaseit.net/claird/comp.lang.python/web_python。

4.5.2 怎样模拟发送 CGI 表单 (METHOD=POST)？

我需要通过 POST 表单获取网页，有什么代码能简单做到吗？

是的，这里有一个使用 `urllib.request` 的简单例子：

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

import urllib.request

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

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

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

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

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

参见：

查看 `urllib-howto` 获取更多示例。

4.5.3 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。

4.5.4 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()
    if not line:
        break
    msg += line

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

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

```
import os

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

4.5.5 socket 的 connect() 方法怎样避免阻塞？

通常会用 `select` 模块处理 socket 异步 I/O。

要避免 TCP 连接阻塞，你可以设置将 socket 设置为非阻塞模式。此时当调用 `connect()` 时，要么连接会立刻建立好（几乎不可能），要么会收到一个包含了错误码 `.error` 的异常。`errno.EINPROGRESS` 表示连接正在进行，但还没有完成。不同的系统会返回不同的值，所以你需要确认你使用的系统会返回什么。

你可以使用 `connect_ex()` 方法来避免产生异常。这个方法只会返回错误码。如果需要轮询的话，你可以再次调用 `connect_ex()` —— 0 或 `errno.EISCONN` 表示连接已建立，或者你也可以用 `select` 检查这个 socket 是否可写。

注解： `asyncore` 模块提供了编写非阻塞网络代码框架性的方法。第三方的 `Twisted` 库也很常用且功能强大。

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()
```

这个函数会返回 [0, 1) 之间的随机浮点数。

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)` 返回 [a, b) 区间之间的浮点数。
- `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)` 对列表进行原地重排，也就是说随机打乱。

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。

扩展/嵌入常见问题

5.1 可以使用 C 语言中创建自己的函数吗？

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++ 语言中创建自己的函数吗？

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？

编写自己的 C 扩展有很多选择，具体取决于您要做的事情。

[Cython](#) 及其相关的 [Pyrex](#) 是接受稍微修改过的 Python 形式并生成相应 C 代码的编译器。[Cython](#) 和 [Pyrex](#) 可以编写扩展而无需学习 Python 的 C API。

如果需要连接到某些当前不存在 Python 扩展的 C 或 C++ 库，可以尝试使用 [SWIG](#) 等工具包装库的数据类型和函数。[SIP](#)，[CXX Boost](#)，或 [Weave](#) 也是包装 C++ 库的替代方案。

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()` 可返回其长度而 `PyTuple_GetItem()` 可返回指定序号上的项。对于列表也有类似的函数 `PyList_Size()` 和 `PyList_GetItem()`。

对于字节串，`PyBytes_Size()` 可返回其长度而 `PyBytes_AsStringAndSize()` 提供一个指向其值和长度的指针。请注意 Python 字节串可能为空，因此 C 的 `strlen()` 不应被使用。

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!')
>>> 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 # repeat until your extension is loaded
gdb) finish   # so that your extension is loaded
gdb) br myfunction.c:50
gdb) continue
```

5.14 我想在 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 版本不包含 /usr/lib/python2.x/config/ 目录，该目录中包含编译 Python 扩展所需的各种文件。

对于 Red Hat，安装 python-devel RPM 以获取必要的文件。

对于 Debian，运行 `apt-get install python-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`。

但是，有时必须在与其它应用程序相同的线程中运行嵌入式 Python 解释器，并且不能允许 `PyRun_InteractiveLoop()` 在等待用户输入时停止。那么另一个解决方案是调用

PyParser_ParseString() 并测试 e.error 等于 E_EOF，如果等于，就意味着输入不完整。这是一个示例代码片段，未经测试，灵感来自 Alex Farber 的代码：

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include <node.h>
#include <errcode.h>
#include <grammar.h>
#include <parsetok.h>
#include <compile.h>

int testcomplete(char *code)
/* code should end in \n */
/* return -1 for error, 0 for incomplete, 1 for complete */
{
    node *n;
    PyErrDetail e;

    n = PyParser_ParseString(code, &_PyParser_Grammar,
                             Py_file_input, &e);

    if (n == NULL) {
        if (e.error == E_EOF)
            return 0;
        return -1;
    }

    PyNode_Free(n);
    return 1;
}
```

另一个解决方案是尝试使用 Py_CompileString() 编译接收到的字符串。如果编译时没有出现错误，请尝试通过调用 PyEval_EvalCode() 来执行返回的代码对象。否则，请将输入保存到以后。如果编译失败，找出是错误还是只需要更多的输入-从异常元组中提取消息字符串，并将其与字符串“分析时意外的 EOF”进行比较。下面是使用 GNUreadline 库的完整示例（您可能希望在调用 readline() 时忽略 SIGINT）：

```
#include <stdio.h>
#include <readline.h>

#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include <object.h>
#include <compile.h>
#include <eval.h>

int main (int argc, char* argv[])
{
    int i, j, done = 0; /* lengths of line, code */
    char ps1[] = ">>> ";
    char ps2[] = "... ";
    char *prompt = ps1;
    char *msg, *line, *code = NULL;
    PyObject *src, *glb, *loc;
    PyObject *exc, *val, *trb, *obj, *dum;

    Py_Initialize ();
    loc = PyDict_New ();
    glb = PyDict_New ();
    PyDict_SetItemString (glb, "__builtins__", PyEval_GetBuiltins ());

    while (!done)
    {
```

(下页继续)

(续上页)

```

line = readline (prompt);

if (NULL == line)                                /* Ctrl-D pressed */
{
    done = 1;
}
else
{
    i = strlen (line);

    if (i > 0)
        add_history (line);                      /* save non-empty lines */

    if (NULL == code)                             /* nothing in code yet */
        j = 0;
    else
        j = strlen (code);

    code = realloc (code, i + j + 2);
    if (NULL == code)                             /* out of memory */
        exit (1);

    if (0 == j)                                   /* code was empty, so */
        code[0] = '\0';                          /* keep strncat happy */

    strncat (code, line, i);                      /* append line to code */
    code[i + j] = '\n';                          /* append '\n' to code */
    code[i + j + 1] = '\0';

    src = Py_CompileString (code, "<stdin>", Py_single_input);

    if (NULL != src)                             /* compiled just fine - */
    {
        if (ps1 == prompt ||                     /* ">>> " or */
            '\n' == code[i + j - 1])             /* "... " and double '\n' */
        {                                         /* so execute it */
            dum = PyEval_EvalCode (src, glb, loc);
            Py_XDECREF (dum);
            Py_XDECREF (src);
            free (code);
            code = NULL;
            if (PyErr_Occurred ())
                PyErr_Print ();
            prompt = ps1;
        }
        /* syntax error or E_EOF? */
    else if (PyErr_ExceptionMatches (PyExc_SyntaxError))
    {
        PyErr_Fetch (&exc, &val, &trb);        /* clears exception! */

        if (PyArg_ParseTuple (val, "sO", &msg, &obj) &&
            !strcmp (msg, "unexpected EOF while parsing")) /* E_EOF */
        {
            Py_XDECREF (exc);
            Py_XDECREF (val);
            Py_XDECREF (trb);
            prompt = ps2;
        }
        /* some other syntax error */
    else
    {
        PyErr_Restore (exc, val, trb);
    }
}

```

(下页继续)

(续上页)

```

        PyErr_Print ();
        free (code);
        code = NULL;
        prompt = ps1;
    }
}
else /* some non-syntax error */
{
    PyErr_Print ();
    free (code);
    code = NULL;
    prompt = ps1;
}

free (line);
}
}

Py_XDECREF (glb);
Py_XDECREF (loc);
Py_Finalize();
exit(0);
}

```

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 库（BPL, <http://www.boost.org/libs/python/doc/index.html>）提供了一种从 C++ 执行此操作的方法（即，您可以使用 BPL 继承自 C++ 编写的扩展类）。

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。

除非你在使用某种集成开发环境，否则你将会在被称为“DOS 窗口”或“命令提示符窗口”的地方输入 Windows 命令。通常你可以在搜索栏搜索 cmd 来创建这种窗口。你应该能够识别你何时打开了这样的窗口，因为你将看到一个 Windows “命令提示符”，通常看起来是这样：

```
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于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一个这样的选项 开始 ▸ 所有程序 ▸ Python 3.x ▸ Python (command line)，运行它后会出现一个有着 `>>>` 提示的想不出来口。如果这样的话，在调用 `exit()` 函数或按 `Ctrl-Z` 组合键后窗口会消失。Windows 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 [cx_Freeze](#) 了解 `distutils` 扩展，它允许你从 Python 代码创建控制台和 GUI 可执行文件。[py2exe](#)，是构建基于 Python 2.x 的可执行文件的最流行扩展，它还不支持 Python 3，但这个版本正在开发。

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 _ 直接在你的.exe 文件中内置 Python。在 Windows 上，Python 必须是一个 DLL，这样才可以处理导入的本身就是 DLL 的模块。（这是第一个未记录的关键事实。）相反，链接到 pythonNN.dll；它通常安装在 C:\Windows\System 中。NN 是 Python 版本，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。

Borland 提示：首先使用 Coff2Omf.exe 将 pythonNN.lib 转换为 OMF 格式。

2. 如果你使用 SWIG，很容易创建一个 Python “扩展模块”，它将使应用程序的数据和方法可供 Python 使用。SWIG 将为你处理所有蹩脚的细节。结果是你将链接到.exe 文件中的 C 代码(!) 你不必创建 DLL 文件，这也简化了链接。
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(); // Initialize Python.
initmyAppc(); // Initialize (import) the helper class.
PyRun_SimpleString("import myApp"); // Import the shadow class.
```

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：采用 FILE* 参数的所谓“极高级”函数在多编译器环境中不起作用，因为每个编译器的 FILE 结构体概念都不同。从实现的角度来看，这些是非常 _ 低 _ 级的功能。

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。是标准的 Windows 特定扩展模块。它定义了一个函数 `kbhit()` 来检查是否存在键盘命中，而 `getch()` 来获取一个字符而不回显它。

图形用户界面（GUI）常见问题

7.1 图形界面常见问题

7.2 Python 是否有平台无关的图形界面工具包？

针对不同的（操作系统或）平台，有多种工具包可供选择。虽然有些工具包还没有移植到 Python 3 上，但至少目前 *Tkinter* 以及 *Qt* 是兼容 Python 3 的。

7.2.1 Tkinter

Python 的标准编译包含了 *tkinter*。这是一个面向对象的接口，指向 Tcl/Tk 微件包。该接口大概是最容易安装（因为该接口包含在 Python 的大部分二进制发行版中）和使用的工具包。要了解 Tk 的详情，比如源代码等，可访问 [Tcl/Tk 项目主页](#)。Tcl/Tk 可完整移植至 Mac OS X，Windows 和 Unix 操作系统上。

7.2.2 wxWidgets

wxWidgets (<https://www.wxwidgets.org>) 是一个自由、可移植的 GUI 图形用户界面类库，使用 C++ 编写。它可以在多个操作系统平台上提供原生自然的界面观感。包括 Windows、Mac OS X、GTK 和 X11 平台在内，都是 *wxWidgets* 当前稳定支持的平台。在语言绑定适配方面，*wxWidgets* 类库可用于较多语言，包括 Python，Perl，Ruby 等。

wxPython 是 *wxWidgets* 的 Python 适配。虽然该绑定在更新进度上经常会稍稍落后于 *wxWidgets*，但它利用纯 Python 扩展，提供了许多其他语言绑定没有实现的特性。*wxPython* 有一个活跃的用户和开发者社区。

wxWidgets 和 *wxPython* 都是自由开源库。宽松的许可证允许人们在商业软件、自由软件和共享软件中使用它们。

7.2.3 Qt

Qt 工具包 (可使用 [PyQt](#) 或 [PySide](#)) 及 KDE ([PyKDE4](#)) 有多个绑定适配可供选择。PyQt 当前相较 PySide 更成熟, 但如果你想编写专有软件, 就必须要从 [Riverbank Computing](#) 购买 PyQt 许可证。PySide 则可以自由使用于各类软件。

Qt 4.5 以上版本使用 LGPL 进行许可; 此外, 商业许可证可从 [Qt 公司](#) 那里获得。

7.2.4 Gtk+

针对 Python 的 [GObject](#) 内省绑定 可以用于编写 GTK+ 3 应用。另请参阅 [Python GTK+ 3 教程](#)。

更早的、针对 [Gtk+ 2](#) 工具包的 [PyGtk](#) 绑定, 是由 James Henstridge 实现的。具体请参考 <http://www.pygtk.org>。

7.2.5 Kivy

[Kivy](#) 是一种跨平台图形用户界面库, 同时支持桌面操作系统 (Windows, macOS 和 Linux) 以及移动设备 (Android, iOS)。该库使用 Python 和 Cython 编写, 可以使用一系列窗口后端。

Kivy 是自由的开源软件, 使用 MIT 许可证分发。

7.2.6 FLTK

the [FLTK toolkit](#) 的 Python 绑定是简单却功能强大且成熟的跨平台窗口系统, 可以在 the [PyFLTK project](#) 里获得相关信息。

7.2.7 OpenGL

对于 OpenGL 绑定, 请参阅 [PyOpenGL](#)。

7.3 有哪些 Python 的 GUI 工具是某个平台专用的 ?

通过安装 [PyObjc Objective-C bridge](#), Python 程序可以使用 Mac OS X 的 Cocoa 库。

Mark Hammond 的 [Pythonwin](#) 包括一个微软基础类 (MFC) 的接口和一个绝大多数由使用 MFC 类的 Python 写成的 Python 编程环境。

7.4 有关 Tkinter 的问题

7.4.1 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。

一种解决方法是将程序与 Tcl 和 Tk 库一同发布, 并且在运行时使用环境变量 `TCL_LIBRARY` 和 `TK_LIBRARY` 指向他们的位置。

为了获得真正能独立运行的应用程序, 来自库里的 Tcl 脚本也需要被整合进应用程序。一个做这种事情的工具叫 SAM (stand-alone modules, 独立模块), 它是 Tix distribution (<http://tix.sourceforge.net/>) 的一部分。

在启用 SAM 时编译 Tix, 在 Python 文件 `Modules/tkappinit.c` 中执行对 `Tclsam_init()` 等的适当调用, 并且将程序与 `libtclsam` 和 `libtkjam` 相链接 (可能也要包括 Tix 的库)。

7.4.2 在等待 I/O 操作时能够处理 Tk 事件吗？

在 Windows 以外的其他平台上可以，你甚至不需要使用线程！但是你必须稍微修改一下你的 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，当一个文件描述符可以进行 I/O 操作的时候，Tk 主循环将会调用这个回调函数。参见 `tkinter-file-handlers`。

7.4.3 在 Tkinter 中键绑定不工作：为什么？

经常听到的抱怨是：已经通过 `bind()` 方法绑定了事件的处理程序，但是，当按下相关的按键后，这个处理程序却没有执行。

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）。

“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 Unix 兼容的操作系统，例如 Mac OS X 和一些 Linux 发行版，默认安装了 Python；它被包含在基本安装套件中。

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。

术语对照表

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

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

2to3 一个将 Python 2.x 代码转换为 Python 3.x 代码的工具，能够处理大部分通过解析源码并遍历解析树可检测到的不兼容问题。

2to3 包含在标准库中，模块名为 `lib2to3`；并提供一个独立入口点 `Tools/scripts/2to3`。参见 `2to3-reference`。

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

annotation -- 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 *type hint* 来使用。

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

binary file -- 二进制文件 *file object* 能够读写字节类对象。二进制文件的例子包括以二进制模式（`'rb'`，`'wb'` 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。

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

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

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

字节码指令列表可以在 `dis` 模块的文档中查看。

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

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

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

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

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

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

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

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

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

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

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

装饰器语法只是一种语法糖，以下两个函数定义在语义上完全等价：

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

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

同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` 的类字典中查找名称为 `b` 的对象，但如果 `b` 是一个描述器，则会调用对应的描述器方法。理解描述器的概念是更深层次理解 Python 的关键，因为这是许多重要特性的基础，包括函数、方法、属性、类方法、静态方法以及对超类的引用等等。

有关描述符的方法的详情可参看 [descriptors](#)。

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

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

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

duck-typing -- 鸭子类型 指一种编程风格，它并不依靠查找对象类型来确定其是否具有正确的接口，而是直接调用或使用其方法或属性（“看起来像鸭子，叫起来也像鸭子，那么肯定就是鸭子。”）由于强调接口而非特定类型，设计良好的代码可通过允许多态替代来提升灵活性。鸭子类型避免使用

`type()` 或 `isinstance()` 检测。(但要注意鸭子类型可以使用[抽象基类](#)作为补充。)而往往会采用 `hasattr()` 检测或是[EAFP](#) 编程。

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

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

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

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

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

实际上共有三类别的文件对象：原始[二进制文件](#)，缓冲[二进制文件](#)以及[文本文件](#)。它们的接口定义均在 `io` 模块中。创建文件对象的规范方式是使用 `open()` 函数。

file-like object -- 文件类对象 [file object](#) 的同义词。

finder -- 查找器 一种会尝试查找被导入模块的[loader](#)的对象。

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

更多详情可参见 [PEP 302](#), [PEP 420](#) 和 [PEP 451](#)。

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

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

function annotation -- 函数标注 即针对函数形参或返回值的[annotation](#)。

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

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

函数标注语法的详解见 [function](#) 一节。

请参看[variable annotation](#) 和 [PEP 484](#) 对此功能的描述。

__future__ 一种伪模块，可被程序员用来启用与当前解释器不兼容的新语言特性。

通过导入 `__future__` 模块并对其中的变量求值，你可以查看新特性何时首次加入语言以及何时成为默认：

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

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

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

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

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

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

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

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

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

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

GIL 参见 *global interpreter lock*。

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

更多信息可查看 [typeiter](#)。

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

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

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

keyword argument -- 关键字参数 参见 [argument](#)。

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

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

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

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

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

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

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

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

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

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

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

更多详情参见 `metaclasses`。

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

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

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

另见 *package*。

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

MRO 参见 *method resolution order*。

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

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

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

```
>>> sys.float_info[1]                # indexed access
1024
>>> sys.float_info.max_exp           # named field access
1024
>>> isinstance(sys.float_info, tuple) # kind of tuple
True
```

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

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

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

另可参见 *module*。

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

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

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

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

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

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

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

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

- *positional-only*：仅限位置，指定一个只能按位置传入的参数。Python 中没有定义仅限位置形参的语法。但是一些内置函数有仅限位置形参（比如 `abs()`）。
- *keyword-only*：仅限关键字，指定一个只能通过关键字传入的参数。仅限关键字形参可通过在函数定义的形参列表中包含单个可变位置形参或者在多个可变位置形参之前放一个 `*` 来定义，例如下面的 *kw_only1* 和 *kw_only2*：

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

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

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

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

形参可以同时指定可选和必选参数，也可以为某些可选参数指定默认值。

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

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

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

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

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

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

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

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

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

参见 [PEP 1](#)。

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

positional argument -- 位置参数 参见 [argument](#)。

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

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

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

provisional package -- 暂定包 参见 [provisional API](#)。

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

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

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

而相应的更简洁更 Pythonic 的方法是这样的：

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

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

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

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

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

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

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

另参见 *namespace package*。

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

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

`collections.abc.Sequence` 抽象基类定义了一个更丰富的接口，它超越了 `__getitem__()` 和 `__len__()`，添加了 `count()`、`index()`、`__contains__()` 和 `__reversed__()`。可以使用 `register()` 显式注册实现此扩展接口的类型。

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

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

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

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

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

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

另请参看 *binary file* 了解能够读写字节类对象的文件对象。

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

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

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

类型别名的作用是简化类型提示。例如：

```
from typing import List, Tuple

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：

```
from typing import List, Tuple

Color = Tuple[int, int, int]

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


参见 `typing` 和 [PEP 484](#)，其中有对此功能的详细描述。

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

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

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

参见 `typing` 和 [PEP 484](#)，其中有对此功能的详细描述。

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

variable annotation -- 变量标注 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。

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

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

另参见 `venv`。

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

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

文档说明

这些文档生成自 [reStructuredText](#) 原文档，由 [Sphinx](#)（一个专门为 Python 文档写的文档生成器）创建。

本文档和它所用工具链的开发完全是由志愿者完成的，这和 Python 本身一样。如果您想参与进来，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., 创造了用于早期 Python 文档的工具链，以及撰写了非常多的文档；
- [Docutils](#) 软件包 项目，创建了 [reStructuredText](#) 文本格式和 [Docutils](#) 软件套件；
- Fredrik Lundh, Sphinx 从他的 [Alternative Python Reference](#) 项目中获得了很多好的想法。

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 源代码发布的 [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！

历史和许可证

C.1 该软件的历史

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

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

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

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

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

注解：GPL 兼容并不意味着 Python 在 GPL 下发布。与 GPL 不同，所有 Python 许可证都允许您分发修改后的版本，而无需开源所做的更改。GPL 兼容的许可证使得 Python 可以与其它在 GPL 下发布的软件结合使用；但其它的许可证则不行。

感谢众多在 Guido 指导下工作的外部志愿者, 使得这些发布成为可能。

C.2 获取或以其他方式使用 Python 的条款和条件

C.2.1 用于 PYTHON 3.7.8rc1 的 PSF 许可协议

1. This LICENSE AGREEMENT is between the Python Software Foundation
→ ("PSF"), and
the Individual or Organization ("Licensee") accessing and otherwise
→ using Python
3.7.8rc1 software in source or binary form and its associated
→ documentation.
2. Subject to the terms and conditions of this License Agreement, PSF
→ hereby
grants Licensee a nonexclusive, royalty-free, world-wide license to
→ reproduce,
analyze, test, perform and/or display publicly, prepare derivative
→ works,
distribute, and otherwise use Python 3.7.8rc1 alone or in any derivative
version, provided, however, that PSF's License Agreement and PSF's
→ notice of
copyright, i.e., "Copyright © 2001-2020 Python Software Foundation; All
→ Rights
Reserved" are retained in Python 3.7.8rc1 alone or in any derivative
→ version
prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or
incorporates Python 3.7.8rc1 or any part thereof, and wants to make the
derivative work available to others as provided herein, then Licensee
→ hereby
agrees to include in any such work a brief summary of the changes made
→ to Python
3.7.8rc1.
4. PSF is making Python 3.7.8rc1 available to Licensee on an "AS IS" basis.
PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY
→ OF
EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY
→ REPRESENTATION OR
WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR
→ THAT THE
USE OF PYTHON 3.7.8rc1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.7.
→ 8rc1
FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A
→ RESULT OF
MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.7.8rc1, OR ANY
→ DERIVATIVE
THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material
→ breach of
its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 3.7.8rc1, 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 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 ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3 被收录软件的许可证与鸣谢

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

C.3.1 Mersenne Twister

`_random` 模块包含基于 <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html> 下载的代码。以下是原始代码的完整注释（声明）：

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 – 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

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

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

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

3. The names of its contributors may not be used to endorse or promote
   products derived from this software without specific prior written
   permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
```

(下页继续)

(续上页)

```
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

Any feedback is very welcome.

<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>

email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)

C.3.2 套接字

socket 模块使用 `getaddrinfo()` 和 `getnameinfo()` 函数, 这些函数源代码在 WIDE 项目 (<http://www.wide.ad.jp/>) 的单独源文件中。

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

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

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

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

C.3.3 异步套接字服务

asynchat and asyncore 模块包含以下声明:

```
Copyright 1996 by Sam Rushing
```

```
All Rights Reserved
```

```
Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
```

(下页继续)

(续上页)

```
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.
```

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

C.3.4 Cookie 管理

http.cookies 模块包含以下声明:

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

```
All Rights Reserved
```

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

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

C.3.5 执行追踪

trace 模块包含以下声明:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err... reserved and offered to the public under the terms of the
Python 2.2 license.
```

```
Author: Zooko O'Whielacronx
```

```
http://zooko.com/
```

```
mailto:zooko@zooko.com
```

```
Copyright 2000, Mojam Media, Inc., all rights reserved.
```

```
Author: Skip Montanaro
```

```
Copyright 1999, Bioreason, Inc., all rights reserved.
```

```
Author: Andrew Dalke
```

```
Copyright 1995-1997, Automatrix, Inc., all rights reserved.
```

(下页继续)

(续上页)

Author: Skip Montanaro

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

Permission to use, copy, modify, and distribute this Python software and its associated documentation for any purpose without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of neither Automatrix, Bioreason or Mojam Media be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

C.3.6 UUencode 与 UUdecode 函数

uu 模块包含以下声明:

Copyright 1994 by Lance Ellinghouse

Cathedral City, California Republic, United States of America.

All Rights Reserved

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

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

Modified by Jack Jansen, CWI, July 1995:

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

C.3.7 XML 远程过程调用

xmlrpc.client 模块包含以下声明:

The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB

Copyright (c) 1999-2002 by Fredrik Lundh

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

Permission to use, copy, modify, and distribute this software and its associated documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in

(下页继续)

(续上页)

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

C.3.8 test_epoll

test_epoll 模块包含以下声明:

Copyright (c) 2001-2006 Twisted Matrix Laboratories.

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

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

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

C.3.9 Select kqueue

select 模块关于 kqueue 的接口包含以下声明:

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

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

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

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND

(下页继续)

(续上页)

```
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 and dtoa

Python/dtoa.c 文件提供了 C 语言的 dtoa 和 strtod 函数, 用于将 C 语言的双精度型和字符串进行转换, 该文件由 David M. Gay 的同名文件派生而来, 当前可从 <http://www.netlib.org/fp/> 下载。2009 年 3 月 16 日检索到的原始文件包含以下版权和许可声明:

```
/* *****
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
```

(下页继续)

(续上页)

```
* WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
* REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
* OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
*
*****/
```

C.3.12 OpenSSL

如果操作系统可用, 则 `hashlib`, `posix`, `ssl`, `crypt` 模块使用 **OpenSSL** 库来提高性能。此外, 适用于 Python 的 Windows 和 Mac OS X 安装程序可能包括 **OpenSSL** 库的拷贝, 所以在此处也列出了 **OpenSSL** 许可证的拷贝:

```
LICENSE ISSUES
=====
```

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of the OpenSSL License and the original SSLeay license apply to the toolkit. See below for the actual license texts. Actually both licenses are BSD-style Open Source licenses. In case of any license issues related to OpenSSL please contact openssl-core@openssl.org.

```
OpenSSL License
-----
```

```
/* =====
 * Copyright (c) 1998-2008 The OpenSSL Project.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 *
 * 3. All advertising materials mentioning features or use of this
 *    software must display the following acknowledgment:
 *    "This product includes software developed by the OpenSSL Project
 *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
 *
 * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
 *    endorse or promote products derived from this software without
 *    prior written permission. For written permission, please contact
 *    openssl-core@openssl.org.
 *
 * 5. Products derived from this software may not be called "OpenSSL"
 *    nor may "OpenSSL" appear in their names without prior written
 *    permission of the OpenSSL Project.
 *
 * 6. Redistributions of any form whatsoever must retain the following
 *    acknowledgment:
 *    "This product includes software developed by the OpenSSL Project
 *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
 *
 * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
```

(下页继续)

(续上页)

```

* EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
* PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
* ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
* NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
* STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
* OF THE POSSIBILITY OF SUCH DAMAGE.
* =====
*
* This product includes cryptographic software written by Eric Young
* (eay@cryptsoft.com).  This product includes software written by Tim
* Hudson (tjh@cryptsoft.com).
*
*/

```

Original SSLeay License

```

/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
* All rights reserved.
*
* This package is an SSL implementation written
* by Eric Young (eay@cryptsoft.com).
* The implementation was written so as to conform with Netscapes SSL.
*
* This library is free for commercial and non-commercial use as long as
* the following conditions are aheared to.  The following conditions
* apply to all code found in this distribution, be it the RC4, RSA,
* lhash, DES, etc., code; not just the SSL code.  The SSL documentation
* included with this distribution is covered by the same copyright terms
* except that the holder is Tim Hudson (tjh@cryptsoft.com).
*
* Copyright remains Eric Young's, and as such any Copyright notices in
* the code are not to be removed.
* If this package is used in a product, Eric Young should be given attribution
* as the author of the parts of the library used.
* This can be in the form of a textual message at program startup or
* in documentation (online or textual) provided with the package.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
* 1. Redistributions of source code must retain the copyright
* notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
* notice, this list of conditions and the following disclaimer in the
* documentation and/or other materials provided with the distribution.
* 3. All advertising materials mentioning features or use of this software
* must display the following acknowledgement:
* "This product includes cryptographic software written by
* Eric Young (eay@cryptsoft.com)"
* The word 'cryptographic' can be left out if the rouines from the library
* being used are not cryptographic related :-).
* 4. If you include any Windows specific code (or a derivative thereof) from
* the apps directory (application code) you must include an acknowledgement:
* "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
*

```

(下页继续)

(续上页)

```
* THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
*
* The licence and distribution terms for any publically available version or
* derivative of this code cannot be changed. i.e. this code cannot simply be
* copied and put under another distribution licence
* [including the GNU Public Licence.]
*/
```

C.3.13 expat

除非使用 `--with-system-expat` 配置了构建, 否则 `pyexpat` 扩展都是用包含 `expat` 源的拷贝构建的:

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

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

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

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

C.3.14 libffi

除非使用 `--with-system-libffi` 配置了构建, 否则 `_ctypes` 扩展都是包含 `libffi` 源的拷贝构建的:

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

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

(下页继续)

(续上页)

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

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

C.3.15 zlib

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

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

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

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

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly
jloup@gzip.org

Mark Adler
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

(下页继续)

(续上页)

copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

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

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

C.3.17 libmpdec

除非使用 `--with-system-libmpdec` 配置了构建, 否则 `_decimal` 模块都是用包含 `libmpdec` 库的拷贝构建的。

Copyright (c) 2008–2016 Stefan Krah. All rights reserved.

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

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

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.

APPENDIX D

Copyright

Python 与这份文档：

Copyright © 2001-2020 Python Software Foundation。保留所有权利。

版权所有 © 2000 BeOpen.com。保留所有权利。

版权所有 © 1995-2000 Corporation for National Research Initiatives。保留所有权利。

版权所有 © 1991-1995 Stichting Mathematisch Centrum。保留所有权利。

有关完整的许可证和许可信息，参见[历史和许可证](#)。

非字母

..., 73
 2to3, 73
 >>>, 73
 __future__, 76
 __slots__, 82
 环境变量
 PATH, 46
 PYTHONDONTWRITEBYTECODE, 30
 TCL_LIBRARY, 68
 TK_LIBRARY, 68

A

abstract base class -- 抽象基类, 73
 annotation -- 标注, 73
 argument
 difference from parameter, 12
 argument -- 参数, 73
 asynchronous context manager -- 异步上下文管理器, 74
 asynchronous generator -- 异步生成器, 74
 asynchronous generator iterator -- 异步生成器迭代器, 74
 asynchronous iterable -- 异步可迭代对象, 74
 asynchronous iterator -- 异步迭代器, 74
 attribute -- 属性, 74
 awaitable -- 可等待对象, 74

B

BDFL, 74
 binary file -- 二进制文件, 74
 bytecode -- 字节码, 74
 bytes-like object -- 字节类对象, 74

C

C-contiguous, 75
 class -- 类, 74
 class variable -- 类变量, 74
 coercion -- 强制类型转换, 74
 complex number -- 复数, 75
 context manager -- 上下文管理器, 75
 context variable -- 上下文变量, 75

contiguous -- 连续, 75
 coroutine -- 协程, 75
 coroutine function -- 协程函数, 75
 CPython, 75

D

decorator -- 装饰器, 75
 descriptor -- 描述器, 75
 dictionary -- 字典, 75
 dictionary view -- 字典视图, 75
 docstring -- 文档字符串, 75
 duck-typing -- 鸭子类型, 75

E

EAFP, 76
 expression -- 表达式, 76
 extension module -- 扩展模块, 76

F

f-string -- f-字符串, 76
 file object -- 文件对象, 76
 file-like object -- 文件类对象, 76
 finder -- 查找器, 76
 floor division -- 向下取整除法, 76
 Fortran contiguous, 75
 function -- 函数, 76
 function annotation -- 函数标注, 76

G

garbage collection -- 垃圾回收, 76
 generator, 76
 generator -- 生成器, 76
 generator expression, 77
 generator expression -- 生成器表达式, 77
 generator iterator -- 生成器迭代器, 77
 generic function -- 泛型函数, 77
 GIL, 77
 global interpreter lock -- 全局解释器锁, 77

H

hash-based pyc -- 基于哈希的 pyc, 77
 hashable -- 可哈希, 77

I

IDLE, [77](#)
immutable -- 不可变, [77](#)
import path -- 导入路径, [77](#)
importer -- 导入器, [77](#)
importing -- 导入, [77](#)
interactive -- 交互, [77](#)
interpreted -- 解释型, [78](#)
interpreter shutdown -- 解释器关闭, [78](#)
iterable -- 可迭代对象, [78](#)
iterator -- 迭代器, [78](#)

K

key function -- 键函数, [78](#)
keyword argument -- 关键字参数, [78](#)

L

lambda, [78](#)
LBYL, [78](#)
list -- 列表, [78](#)
list comprehension -- 列表推导式, [78](#)
loader -- 加载器, [79](#)

M

magic
 method, [79](#)
magic method -- 魔术方法, [79](#)
mapping -- 映射, [79](#)
meta path finder -- 元路径查找器, [79](#)
metaclass -- 元类, [79](#)
method
 magic, [79](#)
 special, [82](#)
method resolution order -- 方法解析顺序, [79](#)
method 方法, [79](#)
module spec -- 模块规格, [79](#)
module 模块, [79](#)
MRO, [79](#)
mutable -- 可变, [79](#)

N

named tuple -- 具名元组, [79](#)
namespace -- 命名空间, [79](#)
namespace package -- 命名空间包, [79](#)
nested scope -- 嵌套作用域, [80](#)
new-style class -- 新式类, [80](#)

O

object -- 对象, [80](#)

P

package -- 包, [80](#)
parameter
 difference from argument, [12](#)
parameter -- 形参, [80](#)
PATH, [46](#)

path based finder -- 基于路径的查找器, [80](#)
path entry -- 路径入口, [80](#)
path entry finder -- 路径入口查找器, [80](#)
path entry hook -- 路径入口钩子, [80](#)
path-like object -- 路径类对象, [80](#)
PEP, [81](#)
portion -- 部分, [81](#)
positional argument -- 位置参数, [81](#)
provisional API -- 暂定 API, [81](#)
provisional package -- 暂定包, [81](#)
Python 3000, [81](#)
Python 提高建议

 PEP 1, [81](#)
 PEP 5, [5](#)
 PEP 6, [2](#)
 PEP 8, [8, 66](#)
 PEP 238, [76](#)
 PEP 275, [37](#)
 PEP 278, [83](#)
 PEP 302, [76, 79](#)
 PEP 343, [75](#)
 PEP 362, [74, 80](#)
 PEP 411, [81](#)
 PEP 420, [76, 79, 81](#)
 PEP 443, [77](#)
 PEP 451, [76](#)
 PEP 484, [73, 76, 83](#)
 PEP 492, [74, 75](#)
 PEP 498, [76](#)
 PEP 519, [80](#)
 PEP 525, [74](#)
 PEP 526, [73, 83](#)
 PEP 570, [17](#)
 PEP 3116, [83](#)
 PEP 3147, [30](#)
 PEP 3155, [81](#)

PYTHONDONTWRITEBYTECODE, [30](#)
Pythonic, [81](#)

Q

qualified name -- 限定名称, [81](#)

R

reference count -- 引用计数, [82](#)
regular package -- 常规包, [82](#)

S

sequence -- 序列, [82](#)
single dispatch -- 单分派, [82](#)
slice -- 切片, [82](#)
special
 method, [82](#)
special method -- 特殊方法, [82](#)
statement -- 语句, [82](#)

T

TCL_LIBRARY, [68](#)
text encoding -- 文本编码, [82](#)

text file -- 文本文件, [82](#)
TK_LIBRARY, [68](#)
triple-quoted string -- 三引号字符串, [82](#)
type -- 类型, [82](#)
type alias -- 类型别名, [82](#)
type hint -- 类型提示, [83](#)

U

universal newlines -- 通用换行, [83](#)

V

variable annotation -- 变量标注, [83](#)
virtual environment -- 虚拟环境, [83](#)
virtual machine -- 虚拟机, [83](#)

Z

Zen of Python -- Python 之禅, [83](#)