



EFFECTIVE
E 系列丛书

PEARSON

Google高级软件工程师Brett Slatkin融合自己多年Python开发实战经验，深入探讨编写高质量Python代码的技巧、禁忌和最佳实践

涵盖Python 3.x和Python 2.x主要应用领域，汇聚59条优秀实践原则、开发技巧和便捷方案，包含大量实用范例代码

Effective Python

59 Specific Ways to Write Better Python

Effective Python

编写高质量Python代码的
59个有效方法

[美] 布雷特·斯拉特金 (Brett Slatkin) 著
爱飞翔 译



机械工业出版社
China Machine Press

Effective Python
59 Specific Ways to Write Better Python

Effective Python

编写高质量Python代码的
59个有效方法

[美] 布雷特·斯拉特金 (Brett Slatkin) 著
爱飞翔 译



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

Effective Python: 编写高质量 Python 代码的 59 个有效方法 / (美) 斯拉特金 (Slatkin, B.) 著; 爱飞翔译. —北京: 机械工业出版社, 2016.1

(Effective 系列丛书)

书名原文: Effective Python: 59 Specific Ways to Write Better Python

ISBN 978-7-111-52355-0

I. E… II. ①斯… ②爱… III. 软件工具 – 程序设计 IV. TP311.56

中国版本图书馆 CIP 数据核字 (2015) 第 305873 号

本书版权登记号: 图字: 01-2015-2812

Authorized translation from the English language edition, entitled *Effective Python: 59 Specific Ways to Write Better Python*, 9780134034287 by Slatkin, published by Pearson Education, Inc., Copyright © 2015.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Chinese simplified language edition published by Pearson Education Asia Ltd., and China Machine Press Copyright © 2016.

本书中文简体字版由 Pearson Education (培生教育出版集团) 授权机械工业出版社在中华人民共和国境内 (不包括中国台湾地区和中国香港、澳门特别行政区) 独家出版发行。未经出版者书面许可, 不得以任何方式抄袭、复制或节录本书中的任何部分。

本书封底贴有 Pearson Education (培生教育出版集团) 激光防伪标签, 无标签者不得销售。

Effective Python

编写高质量 Python 代码的 59 个有效方法

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 关 敏

责任校对: 董纪丽

印 刷: 三河市宏图印务有限公司

版 次: 2016 年 1 月第 1 版第 1 次印刷

开 本: 186mm×240mm 1/16

印 张: 14

书 号: ISBN 978-7-111-52355-0

定 价: 59.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有 • 侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

Praise 本书赞誉

“Slatkin 所写的这本书，其每个条目（item）都是一项独立的教程，并包含它自己的源代码。这种编排方式使我们可以随意跳读：大家可以按照学习的需要来浏览这些条目。本书涉及的话题十分广泛，作者针对这些话题，给出了相当精练而又符合主流观点的建议，我把这本书推荐给中级 Python 程序员。”

——Brandon Rhodes，Dropbox 的软件工程师、2016 ~ 2017 年 PyCon 会议的主席

“我用 Python 写了很多年程序，自认为对 python 已经了解得很透彻了。但在看过这本讲解诀窍和技巧的好书之后我才发现，其实我还能把 Python 代码写得更高效（例如，使用内置的数据结构）、更易读（例如，设定只允许通过关键字形式来指定的参数），以令其更加符合 Python 风格（例如，用 zip 函数来同时迭代两个列表）。”

——Pamela Fox，Khan 学院的教师

“当初我刚从 Java 转向 Python 时，要是能先看到这本书的话，那就能节省好几个月的时间。这本书使我意识到：以前反复编写的那些代码，都不是很符合 Python 的编程风格。这本书包含了 Python 语言的绝大部分必备知识，使我们无需通过数月乃至数年的艰难探索，即可逐个了解它们。本书的内容非常丰富，从 PEP8 的重要性和 Python 语言的主要编程习惯开始，然后谈到如何设计函数、方法和类，如何高效地使用标准库，以及如何设计高质量的 API，最后，又讲了测试及性能问题。新手和老手都可以通过这本优秀教程来领略 Python 编程的真谛。”

——Mike Bayer，SQLAlchemy 的创立者

“这本书会清楚地告诉你如何改善 Python 代码的风格及函数的质量，它会令你的 Python 技能更上一层楼。”

——Leah Culver, Dropbox 的开发者代言人 (developer advocate)

“这是一本极好的书，对其他编程语言较有经验的开发者，可以通过本书迅速学习 Python，并了解更符合 Python 风格的基础语言结构。本书内容清晰、简明，而且易于理解，只需阅读某个条目或某一章，即可单独研究某个话题。书中讲解了大量纯 Python 的语言结构，使读者不会把它们与 Python 生态圈中的其他复杂事物相混淆。经验更多的开发者可以通过书中提供的一些深度范例来了解自己尚未遇到的语言特性，以及原来不常使用的语言功能。作者肯定是一位非常熟悉 Python 的人，他用自己丰富的经验来给读者指出各种经常出现的 bug 以及经常出错的写法。另外，本书也恰当地说明了 Python 2.X 与 Python 3.X 之间的微妙区别，大家在各种版本的 Python 之间迁移时，可以把本书用作参考资料。”

——Katherine Scott, Tempo Automation 的软件主管

“这是一本对初级开发者和熟练开发者都适用的好书。代码范例及其讲解都写得非常细致、非常简洁、非常透彻。”

——C. Titus Brown, 加州大学戴维斯分校副教授

“这本参考书非常有用，它提供了很多高级的 Python 用法，并讲解了如何构建更清晰、更易维护的软件。把书中的建议付诸实践，就可以令自己的 Python 技能得到提升。”

——Wes McKinney, pandas 程序库的创立者《Python for Data Analysis》[⊖]

的作者、Cloudera 的软件工程师

[⊖] 中文版为《利用 Python 进行数据分析》，已由机械工业出版社引进出版。——编辑注

The Translator's Words 译者序

自 Scott Meyers 撰写的《Effective C++》问世以来，出现了很多以 Effective 命名的技术书籍。Effective 一词，并不单单局限于执行速度层面的高效率，同时有着令代码易于阅读、易于测试且易于维护等意思，此外，它还蕴涵着易于扩展、易于修改和易于多人协作等更为高阶的理念。

因此，从上述宏观层面来看，Effective 式的心得手册，无论是对初学者还是熟练者，都有较大意义。本书自然也不例外。对于初学者来说，书中展示了该语言的大体轮廓，使我们能够知道 Python 的强项和弱项。在知道了这些特性之后，开发者就可以结合自己的兴趣与需求，有选择、有顺序地学习。

而对于熟练者来说，则可以把书中的心得与自己的经验相比对，看看自己还有哪些区域尚未深入研究，同时思考一下书中的方案与自己常用的方案各有什么优点与缺点。

从本书各条技巧的具体编排方式来看，本书既可以像字典那样查阅，也可以像普通图书那样通读。很多条目都是用渐进的方式来编写的。作者不会在一开始就给出最佳方案，而是会先从简单的写法入手，逐步发现其缺点并加以完善，最后总结出一套便于使用且易于扩充的解决办法。

这样的演进方式既适用于本书所列的各个场景，也适用于日常的编程工作。如果能通过这些具体的条目来培养一套分析并解决问题的思路，那就可以更加深刻地体会 Python 语言的设计哲学及实践艺术。很多 Python 开发者都崇尚 Pythonic 编程方式，这种 Pythonic 方式不仅应该体现在代码风格和项目规范之中，而且更应该体现在思维模式和架构设计层面，这一点，我想应该是 Effective 系列的书籍值得反复品味的缘由吧。

虽说这 59 条技巧并不能涵盖所有的 Python 领域，但在经常接触的那几个主要领域中，它们却是相当有代表性和启发性的。我们可以把这些技巧以自己的方式实现出

来，并封装成模块及软件包，以便在后续的工作中使用。而对于本书没有专门涉及的领域，如游戏开发、图形绘制、网络通信等，大家不妨也沿用 Effective 书系的一贯做法，把自己的经验总结成条目，进而以博客或开源项目的形式互相交流。这可以说是对 Effective 理念的一种延伸和发展。

由于许多 Python 开发者都同时具备 C++ 及 Java 等其他语言的开发背景，所以 Python 中的很多概念都有好几种不同的称呼方式，而这些术语的中文翻译，自然也就呈现出了一词多译的现象。本书将尽量采用较为折中的办法来处理这些问题。

本书的翻译过程中，得到了机械工业出版社华章公司诸位编辑和工作人员的帮助，在此深表谢意。

由于译者水平有限，不足与疏漏之处，请大家发邮件至 eastarstormlee@gmail.com，或访问 github.com/jeffreybaoshenlee/zh-translation-errata-effective-python/issues 留言，给我以批评和指教。该网页还有《中英文词汇对照表》，以供参考。

Preface 前 言

Python 编程语言很强大、很有魅力，但同时也很独特，所以掌握起来比较困难。许多程序员从他们所熟悉的语言转入 Python 之后，没能把思路打开，以致写出的代码无法完全发挥出 Python 的特性，而另外一些程序员则相反，他们滥用 Python 的特性，导致程序可能在将来出现严重问题。

本书会深入讲解如何以符合 Python 风格的（*Pythonic*）方式来编写程序，这种方式就是运用 Python 语言的最佳方式。笔者假定你对这门语言已经有了初步了解。编程新手可以通过本书学到各种 Python 功能的最佳用法，而编程老手则能够学会如何自信地运用一种功能强大的新工具。

笔者的目标是令大家学会用 Python 来开发优秀的软件。

本书涵盖的内容

本书每一章都包含许多互相关联的条目，大家可以按照自己的需要，随意阅读这些条目。每个条目都包含简明而具体的教程，告诉你应该如何更高效地编写 Python 程序。笔者在每个条目里面都给出了建议，告诉大家应该怎样做、应该避免哪些用法，以及如何在各种做法之间求得平衡，并解释了笔者所选的做法好在哪里。

本书中的各项条目，适用于 Python 3 和 Python 2（请参阅本书第 1 条）。对于 Jython、IronPython 或 PyPy 等其他运行时环境，大部分条目应该同样适用。

第 1 章：用 *Pythonic* 方式来思考

Python 开发者用 *Pythonic* 这个形容词来描述具有特定风格的代码。这种风格是大家在使用 Python 语言进行编程并相互协作的过程中逐渐形成的习惯。本章讲解如何以

该风格来完成常见的 Python 编程工作。

第 2 章：函数

Python 中的函数具备多种特性，这可以简化编程工作。Python 函数的某些性质与其他编程语言中的函数相似，但也有些性质是 Python 独有的。本章介绍如何用函数来表达意图、提升可复用程度，并减少 bug。

第 3 章：类与继承

Python 是面向对象的语言。用 Python 编程时，通常需要编写新类，并定义这些类应该如何通过其接口及继承体系与外界相交互。本章讲解如何使用类和继承来表达对象所应具备的行为。

第 4 章：元类及属性

元类（metaclass）及动态属性（dynamic attribute）都是很强大的 Python 特性，然而它们也可能导致极其古怪、极其突然的行为。本章讲解这些机制的常见用法，以确保读者写出来的代码符合最小惊讶原则（rule of least surprise）。

第 5 章：并发及并行

用 Python 很容易就能写出并发程序，这种程序可以在同一时间做许多件不同的事情。我们也可以通过系统调用、子进程（subprocess）及 C 语言扩展来实现并行处理。本章讲解如何在不同情况下充分利用这些 Python 特性。

第 6 章：内置模块

Python 预装了许多写程序时会用到的重要模块。这些标准软件包与通常意义上的 Python 语言联系得非常紧密，我们可以将其当成语言规范的一部分。本章将会讲解基本的内置模块。

第 7 章：协作开发

如果许多人要开发同一个 Python 程序，那就得仔细商量代码的写法了。即便你是一个人开发，也需要理解其他人所写的模块。本章讲解多人协作开发 Python 程序时所

用的标准工具及最佳做法。

第 8 章：部署

Python 提供了一些工具，使我们可以把软件部署到不同的环境中。它也提供了一些模块，令开发者可以把程序编写得更加健壮。本章讲解如何使用 Python 调试、优化并测试程序，以提升其质量与性能。

本书使用的约定

本书在 Python 代码风格指南（Python style guide）的基础上做了一些修改，使范例代码便于印刷，也便于凸显其中的重要内容。一行代码比较长时，会以 ➞ 字符来表示折行。代码中的某些部分，与当前要讲的问题联系不大，笔者会将这部分代码略去，并在注释中以省略号来表示（# ...）。为了缩减范例代码的篇幅，笔者也把内嵌的文档删去了。读者在开发自己的项目时不应该这么做，而是应该遵循 Python 风格指南（参见本书第 2 条），并为源代码撰写开发文档（参见本书第 49 条）。

书中大部分代码，运行之后都会产生输出（output）。笔者所谓的输出，意思是说：在互动式解释器（interactive interpreter）中运行这些 Python 程序时，控制台或终端机里面会打印出一些信息。这些打印出来的信息，以等宽字体印刷，它们上方的那一行会标有 >>> 符号（这个 >>> 符号是 Python 解释器的提示符）。笔者使用这个符号是想告诉大家：把 >>> 上方的那些范例代码输入 Python shell 之后，会产生与 >>> 下方文字相符的输出信息。

除此之外，还有一些上方虽无 >>> 符号，但却以等宽字体印刷的代码段。这些内容用来表示产生于 Python 解释器之外的输出信息。它们的上方通常都会有 \$ 字符，这表示笔者是在 Bash 之类的命令行 shell 里面先运行了程序，然后才产生这些输出的。

获取源代码及勘误表

大家可以抛开本书的讲解部分，把某些范例作为完整的程序运行一遍，这样是很有好处的。你可以用这些代码做实验，以了解整个程序的运行原理。全部源码都可以从本书网站 (<http://www.effectivepython.com/>) 下载^①。书中的错误也会张贴到该网站^②。

① 范例代码的网址是：<https://github.com/bslatkin/effectivepython>。——译者注

② 这里针对的是英文原书，勘误表的网址是：<https://github.com/bslatkin/effectivepython/issues>。——译者注

致 谢 *Acknowledgements*

在生活中，有很多人给了我指导、支持及鼓励，没有他们，本书就不会面世。

感谢《Effective Software Development》系列的顾问 Scott Meyers。笔者 15 岁那年初次阅读了 Scott 所写的《Effective C++》，当时我就迷上了这门语言。我后来的教育经历，以及在 Google 的第一份工作，无疑都得益于 Scott 的那本书。这次有机会写作本书，本人深感荣幸。

感谢核心技术评审者 Brett Cannon、Tavis Rudd 和 Mike Taylor，他们为本书提供了深刻而透彻的反馈意见。感谢 Leah Culver 和 Adrian Holovaty，他们两位认为写作这样一本书很有意义。感谢友人 Michael Levine、Marzia Niccolai、Ade Oshineye 和 Katrina Sostek，他们耐心阅读了本书的初稿。也感谢 Google 诸位同事审读本书。若没有以上诸君的帮助，本书读起来可能就会比较费解。

感谢制作本书的每一位工作人员。感谢编辑 Trina MacDonald 启动本书制作流程，并提供大力支持。感谢诸位团队成员帮助制作本书，他们是：策划编辑 Tom Cirtin 和 Chris Zahn、助理编辑 Olivia Basegio、营销经理 Stephane Nakib、文字编辑 Stephanie Geels，以及生产编辑 Julie Nahil。

感谢与我共事的诸位优秀 Python 程序员：Anthony Baxter、Brett Cannon、Wesley Chun、Jeremy Hylton、Alex Martelli、Neal Norwitz、Guido van Rossum、Andy Smith、Greg Stein 和 Ka-Ping Yee。很高兴你们能督促并指引我学习 Python。Python 开发社团构建得非常优秀，成为一名 Python 开发者，令我感到特别荣幸。

感谢诸位同事这些年来对我的关照。感谢 Kevin Gibbs 帮助我应对风险。感谢 Ken Ashcraft、Ryan Barrett 和 Jon McAlister 教会我如何工作。感谢 Brad Fitzpatrick 帮助我提升工作能力。感谢 Paul McDonald 陪我一起创建我们的搞怪项目。感谢 Jeremy Ginsberg

和 Jack Hebert 令其成为现实。

感谢激发我编程兴趣的诸位老师：Ben Chelf、Vince Hugo、Russ Lewin、Jon Stemmle、Derek Thomson 和 Daniel Wang。正因为有了你们的指引，我才会努力磨练编程技术，进而使自己有能力去教导他人。

感谢母亲使我找到了人生的目标并鼓励我做程序员。感谢兄弟、祖父母、众亲戚以及儿时的玩伴，从小你们就是我的榜样，也使我找到了成长的快乐。

最后要感谢我的妻子 Colleen，感谢她的关爱和支持，感谢她带来的欢笑。

目 录 *Contents*

本书赞誉
译者序
前言
致谢

第1章 用 Pythonic 方式来思考	1
第1条：确认自己所用的 Python 版本	1
第2条：遵循 PEP 8 风格指南	3
第3条：了解 bytes、str 与 unicode 的区别	5
第4条：用辅助函数来取代复杂的表达式	8
第5条：了解切割序列的办法	10
第6条：在单次切片操作内，不要同时指定 start、end 和 stride	13
第7条：用列表推导来取代 map 和 filter	15
第8条：不要使用含有两个以上表达式的列表推导	16
第9条：用生成器表达式来改写数据量较大的列表推导	18
第10条：尽量用 enumerate 取代 range	20
第11条：用 zip 函数同时遍历两个迭代器	21
第12条：不要在 for 和 while 循环后面写 else 块	23
第13条：合理利用 try/except/else/finally 结构中的每个代码块	25
第2章 函数	28
第14条：尽量用异常来表示特殊情况，而不要返回 None	28

第 15 条：了解如何在闭包里使用外围作用域中的变量	30
第 16 条：考虑用生成器来改写直接返回列表的函数	35
第 17 条：在参数上面迭代时，要多加小心	37
第 18 条：用数量可变的位置参数减少视觉杂讯	41
第 19 条：用关键字参数来表达可选的行为	43
第 20 条：用 None 和文档字符串来描述具有动态默认值的参数	46
第 21 条：用只能以关键字形式指定的参数来确保代码明晰	49
第 3 章 类与继承	53
第 22 条：尽量用辅助类来维护程序的状态，而不要用字典和元组	53
第 23 条：简单的接口应该接受函数，而不是类的实例	58
第 24 条：以 @classmethod 形式的多态去通用地构建对象	62
第 25 条：用 super 初始化父类	67
第 26 条：只在使用 Mix-in 组件制作工具类时进行多重继承	71
第 27 条：多用 public 属性，少用 private 属性	75
第 28 条：继承 collections.abc 以实现自定义的容器类型	79
第 4 章 元类及属性	84
第 29 条：用纯属性取代 get 和 set 方法	84
第 30 条：考虑用 @property 来代替属性重构	88
第 31 条：用描述符来改写需要复用的 @property 方法	92
第 32 条：用 __getattr__、__getattribute__ 和 __setattr__ 实现按需生成的属性	97
第 33 条：用元类来验证子类	102
第 34 条：用元类来注册子类	104
第 35 条：用元类来注解类的属性	108
第 5 章 并发及并行	112
第 36 条：用 subprocess 模块来管理子进程	113
第 37 条：可以用线程来执行阻塞式 I/O，但不要用它做平行计算	117
第 38 条：在线程中使用 Lock 来防止数据竞争	121

第 39 条：用 Queue 来协调各线程之间的工作	124
第 40 条：考虑用协程来并发地运行多个函数	131
第 41 条：考虑用 concurrent.futures 来实现真正的平行计算	141
第 6 章 内置模块	145
第 42 条：用 functools.wraps 定义函数修饰器	145
第 43 条：考虑以 contextlib 和 with 语句来改写可复用的 try/finally 代码	148
第 44 条：用 copyreg 实现可靠的 pickle 操作	151
第 45 条：应该用 datetime 模块来处理本地时间，而不是用 time 模块	157
第 46 条：使用内置算法与数据结构	161
第 47 条：在重视精确度的场合，应该使用 decimal	166
第 48 条：学会安装由 Python 开发者社区所构建的模块	168
第 7 章 协作开发	170
第 49 条：为每个函数、类和模块编写文档字符串	170
第 50 条：用包来安排模块，并提供稳固的 API	174
第 51 条：为自编的模块定义根异常，以便将调用者与 API 相隔离	179
第 52 条：用适当的方式打破循环依赖关系	182
第 53 条：用虚拟环境隔离项目，并重建其依赖关系	187
第 8 章 部署	193
第 54 条：考虑用模块级别的代码来配置不同的部署环境	193
第 55 条：通过 repr 字符串来输出调试信息	195
第 56 条：用 unittest 来测试全部代码	198
第 57 条：考虑用 pdb 实现交互调试	201
第 58 条：先分析性能，然后再优化	203
第 59 条：用 tracemalloc 来掌握内存的使用及泄漏情况	208

用 Pythonic 方式来思考

一门语言的编程习惯是由用户来确立的。这些年来，Python 开发者用 Pythonic 这个形容词来描述那种符合特定风格的代码。这种 Pythonic 风格，既不是非常严密的规范，也不是由编译器强加给开发者的规则，而是大家在使用 Python 语言协同工作的过程中逐渐形成的习惯。Python 开发者不喜欢复杂的事物，他们崇尚直观、简洁而又易读的代码（请在 Python 解释器中输入 `import this`）。

对 C++ 或 Java 等其他语言比较熟悉的人，可能还在按自己喜欢的风格来使用 Python；而刚刚接触 Python 的程序员，则需要逐渐熟悉许多可以用 Python 代码来表达的概念。但无论哪一种开发者，都必须知道如何以最佳方式完成常见的 Python 编程工作，这种最佳方式，就是 Pythonic 方式。该方式将会影响你所写的每个程序。

第 1 条：确认自己所用的 Python 版本

本书绝大部分范例代码都遵循 Python 3.4（发布于 2014 年 3 月 17 日）的语法。某些范例还会同时给出 Python 2.7（发布于 2010 年 7 月 3 日）版本的代码，以强调两者的区别。笔者给出的建议适用于 CPython、Jython、IronPython 及 PyPy 等流行的 Python 运行时环境。

很多电脑都预装了多个版本的标准 CPython 运行时环境[⊖]。然而，在命令行中输入

[⊖] 在不引起混淆的情况下，可以把 CPython 理解成默认的 Python 解释器。下同。——译者注

默认的 python 命令之后，究竟会执行哪个版本则无法肯定。python 通常是 python 2.7 的别名，但也有可能是 python 2.6 或 python 2.5 等旧版本的别名。请用 --version 标志来运行 python 命令，以了解所使用的具体 Python 版本。

```
$ python --version
Python 2.7.8
```

通常可以用 python3 命令来运行 Python 3。

```
$ python3 --version
Python 3.4.2
```

运行程序的时候，也可以在内置的 sys 模块里查询相关的值，以确定当前使用的 Python 版本。

```
import sys
print(sys.version_info)
print(sys.version)

>>>
sys.version_info(major=3, minor=4, micro=2,
➥releaselevel='final', serial=0)
3.4.2 (default, Oct 19 2014, 17:52:17)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.51)]
```

Python 2 和 Python 3 都处在 Python 社区的积极维护之中。但是 Python 2 的功能开发已经冻结，只会进行 bug 修复、安全增强以及移植等工作，以便使开发者能顺利从 Python 2 迁移到 Python 3。2to3 与 six[⊖]等工具可以帮助大家把代码轻松地适配到 Python 3 及其后续版本上面。

Python 3 经常会添加新功能并提供改进，而这些功能与改进不会出现在 Python 2 中。笔者写作本书时，大部分 Python 开源代码库都已经兼容 Python 3 了，所以强烈建议大家使用 Python 3 来开发自己的下一个 Python 项目。

要点

- 有两个版本的 Python 处于活跃状态，它们是：Python 2 与 Python 3。
- 有很多种流行的 Python 运行时环境，例如，CPython、Jython、IronPython 以及 PyPy 等。
- 在操作系统的命令行中运行 Python 时，请确保该 Python 的版本与你想使用的 Python 版本相符。

[⊖] 该工具的网址是：<https://pythonhosted.org/six/>。——译者注

- 由于 Python 社区把开发重点放在 Python 3 上，所以在开发后续项目时，应该优先考虑采用 Python 3。

第2条：遵循 PEP 8 风格指南

《Python Enhancement Proposal #8》（8号 Python 增强提案）又叫 PEP 8，它是针对 Python 代码格式而编订的风格指南。尽管可以在保证语法正确的前提下随意编写 Python 代码，但是，采用一致的风格来书写可以令代码更加易懂、更加易读。采用和其他 Python 程序员相同的风格来写代码，也可以使项目更利于多人协作。即便代码只会由你自己阅读，遵循这套风格也依然可以令后续的修改变得容易一些。

PEP 8 列出了许多细节，以描述如何撰写清晰的 Python 代码。它会随着 Python 语言持续更新。大家应该把整份指南都读一遍 (<http://www.python.org/dev/peps/pep-0008>)。下面列出几条绝对应该遵守的规则。

空白：Python 中的空白（whitespace）会影响代码的含义。Python 程序员使用空白的时候尤其在意，因为它们还会影响代码的清晰程度。

- 使用 space（空格）来表示缩进，而不要用 tab（制表符）。
- 和语法相关的每一层缩进都用 4 个空格来表示。
- 每行的字符数不应超过 79。
- 对于占据多行的长表达式来说，除了首行之外的其余各行都应该在通常的缩进级别之上再加 4 个空格。
- 文件中的函数与类之间应该用两个空行隔开。
- 在同一个类中，各方法之间应该用一个空行隔开。
- 在使用下标来获取列表元素、调用函数或给关键字参数赋值的时候，不要在两旁添加空格。
- 为变量赋值的时候，赋值符号的左侧和右侧应该各自写上一个空格，而且只写一个就好。

命名：PEP 8 提倡采用不同的命名风格来编写 Python 代码中的各个部分，以便在阅读代码时可以根据这些名称看出它们在 Python 语言中的角色。

- 函数、变量及属性应该用小写字母来拼写，各单词之间以下划线相连，例如，`lowercase_underscore`。

- 受保护的实例属性，应该以单个下划线开头，例如，`_leading_underscore`。
- 私有的实例属性，应该以两个下划线开头，例如，`__double_leading_underscore`。
- 类与异常，应该以每个单词首字母均大写的形式来命名，例如，`CapitalizedWord`。
- 模块级别的常量，应该全部采用大写字母来拼写，各单词之间以下划线相连，例如，`ALL_CAPS`。

- 类中的实例方法（instance method），应该把首个参数命名为 `self`，以表示该对象自身。
- 类方法（class method）的首个参数，应该命名为 `cls`，以表示该类自身。

表达式和语句：《The Zen of Python》（Python 之禅）中说：“每件事都应该有直白的做法，而且最好只有一种。”PEP 8 在制定表达式和语句的风格时，就试着体现了这种思想。

- 采用内联形式的否定词，而不要把否定词放在整个表达式的前面，例如，应该写 `if a is not b` 而不是 `if not a is b`。
- 不要通过检测长度的办法（如 `if len(somelist) == 0`）来判断 `somelist` 是否为 `[]` 或 `"` 等空值，而是应该采用 `if not somelist` 这种写法来判断，它会假定：空值将自动评估为 `False`。
- 检测 `somelist` 是否为 `[1]` 或 `'hi'` 等非空值时，也应如此，`if somelist` 语句默认会把非空的值判断为 `True`。
- 不要编写单行的 `if` 语句、`for` 循环、`while` 循环及 `except` 复合语句，而是应该把这些语句分成多行来书写，以示清晰。
- `import` 语句应该总是放在文件开头。
- 引入模块的时候，总是应该使用绝对名称，而不应该根据当前模块的路径来使用相对名称。例如，引入 `bar` 包中的 `foo` 模块时，应该完整地写出 `from bar import foo`，而不应该简写为 `import foo`。
- 如果一定要以相对名称来编写 `import` 语句，那就采用明确的写法：`from.import foo`。
- 文件中的那些 `import` 语句应该按顺序划分成三个部分，分别表示标准库模块、第三方模块以及自用模块。在每一部分之中，各 `import` 语句应该按模块的字母顺序来排列。



注意 Pylint (<http://www.pylint.org/>) 是一款流行的 Python 源码静态分析工具。它可以自动检查受测代码是否符合 PEP 8 风格指南，而且还能找出 Python 程序里的多种常见错误。

要点

- 当编写 Python 代码时，总是应该遵循 PEP 8 风格指南。
- 与广大 Python 开发者采用同一套代码风格，可以使项目更利于多人协作。
- 采用一致的风格来编写代码，可以令后续的修改工作变得更为容易。

第3条：了解 bytes、str 与 unicode 的区别

Python 3 有两种表示字符序列的类型：bytes 和 str。前者的实例包含原始的 8 位值[⊖]；后者的实例包含 Unicode 字符。

Python 2 也有两种表示字符序列的类型，分别叫做 str 和 unicode。与 Python 3 不同的是，str 的实例包含原始的 8 位值；而 unicode 的实例，则包含 Unicode 字符。

把 Unicode 字符表示为二进制数据（也就是原始 8 位值）有许多种办法。最常见的编码方式就是 *UTF-8*。但是大家要记住，Python 3 的 str 实例和 Python 2 的 unicode 实例都没有和特定的二进制编码形式相关联。要想把 Unicode 字符转换成二进制数据，就必须使用 encode 方法。要想把二进制数据转换成 Unicode 字符，则必须使用 decode 方法。

编写 Python 程序的时候，一定要把编码和解码操作放在界面最外围来做。程序的核心部分应该使用 Unicode 字符类型（也就是 Python 3 中的 str、Python 2 中的 unicode），而且不要对字符编码做任何假设。这种办法既可以令程序接受多种类型的文本编码（如 *Latin-1*、*Shift JIS* 和 *Big5*），又可以保证输出的文本信息只采用一种编码形式（最好是 UTF-8）。

由于字符类型有别，所以 Python 代码中经常会出现两种常见的使用情境：

- 开发者需要原始 8 位值，这些 8 位值表示以 UTF-8 格式（或其他编码形式）来编码的字符。
- 开发者需要操作没有特定编码形式的 Unicode 字符。

所以，我们需要编写两个辅助（helper）函数，以便在这两种情况之间转换，使得转换后的输入数据能够符合开发者的预期。

在 Python 3 中，我们需要编写接受 str 或 bytes，并总是返回 str 的方法：

[⊖] 就是原始的字节，由于每个字节有 8 个二进制位，所以是原始的 8 位值。也叫做原生 8 位值、纯 8 位值。——译者注

```
def to_str(bytes_or_str):
    if isinstance(bytes_or_str, bytes):
        value = bytes_or_str.decode('utf-8')
    else:
        value = bytes_or_str
    return value # Instance of str
```

另外，还需要编写接受 str 或 bytes，并总是返回 bytes 的方法：

```
def to_bytes(bytes_or_str):
    if isinstance(bytes_or_str, str):
        value = bytes_or_str.encode('utf-8')
    else:
        value = bytes_or_str
    return value # Instance of bytes
```

在 Python 2 中，需要编写接受 str 或 unicode，并总是返回 unicode 的方法：

```
# Python 2
def to_unicode(unicode_or_str):
    if isinstance(unicode_or_str, str):
        value = unicode_or_str.decode('utf-8')
    else:
        value = unicode_or_str
    return value # Instance of unicode
```

另外，还需要编写接受 str 或 unicode，并总是返回 str 的方法：

```
# Python 2
def to_str(unicode_or_str):
    if isinstance(unicode_or_str, unicode):
        value = unicode_or_str.encode('utf-8')
    else:
        value = unicode_or_str
    return value # Instance of str
```

在 Python 中使用原始 8 位值与 Unicode 字符时，有两个问题要注意。

第一个问题可能会出现在 Python 2 里面。如果 str 只包含 7 位 ASCII 字符，那么 unicode 和 str 实例似乎就成了同一种类型。

- 可以用 + 操作符把这种 str 与 unicode 连接起来。
- 可以用等价与不等价操作符，在这种 str 实例与 unicode 实例之间进行比较。
- 在格式字符串中，可以用 '%s' 等形式来代表 unicode 实例。

这些行为意味着，在只处理 7 位 ASCII 的情境下，如果某函数接受 str，那么可以给它传入 unicode；如果某函数接受 unicode，那么也可以给它传入 str。而在 Python 3 中，bytes 与 str 实例则绝对不会等价，即使是空字符串也不行。所以，在传入字符序列的时候必须留意其类型。

第二个问题可能会出现在 Python 3 里面。如果通过内置的 open 函数获取了文件句柄^①，那么请注意，该句柄默认会采用 UTF-8 编码格式来操作文件。而在 Python 2 中，文件操作的默认编码格式则是二进制形式。这可能会导致程序出现奇怪的错误，对习惯了 Python 2 的程序员来说更是如此。

例如，现在要向文件中随机写入一些二进制数据。下面这种用法在 Python 2 中可以正常运作，但在 Python 3 中不行。

```
with open('/tmp/random.bin', 'w') as f:  
    f.write(os.urandom(10))  
  
>>>  
TypeError: must be str, not bytes
```

发生上述异常的原因在于，Python 3 给 open 函数添加了名为 encoding 的新参数，而这个新参数的默认值却是 'utf-8'。这样在文件句柄上进行 read 和 write 操作时，系统就要求开发者必须传入包含 Unicode 字符的 str 实例，而不接受包含二进制数据的 bytes 实例。

为了解决这个问题，我们必须用二进制写入模式 ('wb') 来开启待操作的文件，而不能像原来那样，采用字符写入模式 ('w')。按照下面这种方式来使用 open 函数，即可同时适配 Python 2 与 Python 3：

```
with open('/tmp/random.bin', 'wb') as f:  
    f.write(os.urandom(10))
```

从文件中读取数据的时候也有这种问题。解决办法与写入时相似：用 'rb' 模式（也就是二进制模式）打开文件，而不要使用 'r' 模式。

要点

- 在 Python 3 中，bytes 是一种包含 8 位值的序列，str 是一种包含 Unicode 字符的序列。开发者不能以 > 或 + 等操作符来混同操作 bytes 和 str 实例。
- 在 Python 2 中，str 是一种包含 8 位值的序列，unicode 是一种包含 Unicode 字符的序列。如果 str 只含有 7 位 ASCII 字符，那么可以通过相关的操作符来同时使用 str 与 unicode。
- 在对输入的数据进行操作之前，使用辅助函数来保证字符序列的类型与开发者的期望相符（有的时候，开发者想操作以 UTF-8 格式来编码的 8 位值，有的时候，则想操作 Unicode 字符）。

① 文件句柄（file handle）其实是一种标识符或指针，也可以理解为文件描述符，用来指代开发者将要操作的文件。本书会酌情按照习惯称呼，将 handle 一词译为句柄、操作标识或描述符。——译者注

- 从文件中读取二进制数据，或向其中写入二进制数据时，总应该以 'rb' 或 'wb' 等二进制模式来开启文件。

第 4 条：用辅助函数来取代复杂的表达式

Python 的语法非常精练，很容易就能用一行表达式来实现许多逻辑。例如，要从 URL 中解码查询字符串。在下例所举的查询字符串中，每个参数都可以表示一个整数值：

```
from urllib.parse import parse_qs
my_values = parse_qs('red=5&blue=0&green=',
                      keep_blank_values=True)
print(repr(my_values))

>>>
{'red': ['5'], 'green': [''], 'blue': ['0']}
```

查询字符串中的某些参数可能有多个值，某些参数可能只有一个值，某些参数可能是空白（blank）值，还有些参数则没有出现在查询字符串之中。用 get 方法在 my_values 字典中查询不同的参数时，就有可能获得不同的返回值。

```
print('Red:      ', my_values.get('red'))
print('Green:    ', my_values.get('green'))
print('Opacity: ', my_values.get('opacity'))

>>>
Red:      ['5']
Green:    []
Opacity: None
```

如果待查询的参数没有出现在字符串中，或当该参数的值为空白时能够返回默认值 0，那就更好了。这个逻辑看上去似乎并不值得用完整的 if 语句或辅助函数来实现，于是，你可能会考虑用 Boolean 表达式。

由于 Python 的语法非常精练，所以我们很容易就想到了这种做法。空字符串、空列表及零值，都会评估为 False。因此，在下面这个例子中，如果 or 操作符左侧的子表达式估值为 False，那么整个表达式的值就将是 or 操作符右侧那个子表达式的值。

```
# For query string 'red=5&blue=0&green='
red = my_values.get('red', [''])[0] or 0
green = my_values.get('green', [''])[0] or 0
opacity = my_values.get('opacity', [''])[0] or 0
print('Red:    %r' % red)
print('Green:  %r' % green)
print('Opacity: %r' % opacity)
```

```
>>>
Red:      '5'
Green:    0
Opacity: 0
```

`red`那一行代码是正确的，因为 `my_values` 字典里确实有 '`red`' 这个键。该键所对应的值是个列表，列表中只有一个元素，也就是字符串 '`5`'。这个字符串会自动估值为 `True`，所以，`or` 表达式第一部分的值就会赋给 `red`。

`green`那一行代码也是正确的，因为 `get` 方法从 `my_values` 字典中获取的值是个列表，该列表只有一个元素，这个元素是个空字符串。由于空字符串会自动估值为 `False`，所以整个 `or` 表达式的值就成了 `0`。

`opacity`那一行代码也没有错。`my_values` 字典里面没有名为 '`opacity`' 的键，而当字典中没有待查询的键时，`get` 方法会返回第二个参数的值，所以，在本例中，`get` 方法就会返回仅包含一个元素的列表，那个元素是个空字符串。当字典里没有待查询的 '`opacity`' 键时，这行代码的执行效果与 `green` 那行代码相同。

这样的长表达式虽然语法正确，但却很难阅读，而且有时也未必完全符合要求。由于我们想在数学表达式中使用这些参数值，所以还要确保每个参数的值都是整数。为了实现这一点，我们需要把每个长表达式都包裹在内置的 `int` 函数中，以便把字符串解析为整数。

```
red = int(my_values.get('red', ['']))[0] or 0
```

这种写法读起来很困难，而且看上去很乱。这样的代码不容易理解，初次拿到这种代码的人，可能先要花些功夫把表达式拆解开，然后才能看明白它的作用。即使想把代码写得简省一些，也没有必要将全部内容都挤在一行里面。

Python 2.5 添加了 `if/else` 条件表达式（又称三元操作符），使我们可以把上述逻辑写得清晰一些，同时还能保持代码简洁。

```
red = my_values.get('red', [''])
red = int(red[0]) if red[0] else 0
```

这种写法比原来好了一些。对于不太复杂的情况来说，`if/else` 条件表达式可以令代码变得清晰。但对于上面这个例子来说，它的清晰程度还是比不上跨多行的完整 `if/else` 语句。如果把上述逻辑全都改成下面这种形式，那我们能就感觉到：刚才那种紧缩的写法其实挺复杂的。

```
green = my_values.get('green', [''])
if green[0]:
```

```

green = int(green[0])
else:
    green = 0

```

现在应该把它总结成辅助函数了，如果需要频繁使用这种逻辑，那就更应该这样做。

```

def get_first_int(values, key, default=0):
    found = values.get(key, [''])
    if found[0]:
        found = int(found[0])
    else:
        found = default
    return found

```

调用这个辅助函数时所使用的代码，要比使用 or 操作符的长表达式版本，以及使用 if/else 表达式的两行版本更加清晰。

```
green = get_first_int(my_values, 'green')
```

表达式如果变得比较复杂，那就应该考虑将其拆解成小块，并把这些逻辑移入辅助函数中。这会令代码更加易读，它比原来那种密集的写法更好。编写 Python 程序时，不要一味追求过于紧凑的写法，那样会写出非常复杂的表达式。

要点

- 开发者很容易过度运用 Python 的语法特性，从而写出那种特别复杂并且难以理解的单行表达式。
- 请把复杂的表达式移入辅助函数之中，如果要反复使用相同的逻辑，那就更应该这么做。
- 使用 if/else 表达式，要比用 or 或 and 这样的 Boolean 操作符写成的表达式更加清晰。

第 5 条：了解切割序列的办法

Python 提供了一种把序列切成小块的写法。这种切片（slice）[⊖]操作，使得开发者能够轻易地访问由序列中的某些元素所构成的子集。最简单的用法，就是对内置的 list、str 和 bytes 进行切割。切割操作还可以延伸到实现了 __getitem__ 和 __setitem__ 这两个特殊方法的 Python 类上（参见本书第 28 条）。

[⊖] 在不引起混淆时，本书也将其称为切割，下同。——译者注

切割操作的基本写法是 `somelist[start:end]`，其中 `start`（起始索引）所指的元素涵盖在切割后的范围内，而 `end`（结束索引）所指的元素则不包括在切割结果之中。

```
a = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
print('First four:', a[:4])
print('Last four:', a[-4:])
print('Middle two:', a[3:-3])

>>>
First four: ['a', 'b', 'c', 'd']
Last four: ['e', 'f', 'g', 'h']
Middle two: ['d', 'e']
```

如果从列表开头获取切片，那就不要在 `start` 那里写上 0，而是应该把它留空，这样代码看起来会清爽一些。

```
assert a[:5] == a[0:5]
```

如果切片一直要取到列表末尾，那就应该把 `end` 留空，因为即便写了，也是多余。

```
assert a[5:] == a[5:len(a)]
```

在指定切片起止索引时，若要从列表尾部向前算，则可使用负值来表示相关偏移量。如果采用下面这些写法来切割列表，那么即便是刚刚接触代码的人也能立刻明白程序的意图。由于这些写法都不会令人惊讶，所以笔者推荐大家在代码中放心地使用。

```
a[:]      # ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
a[:5]     # ['a', 'b', 'c', 'd', 'e']
a[:-1]    # ['a', 'b', 'c', 'd', 'e', 'f', 'g']
a[4:]     #                   ['e', 'f', 'g', 'h']
a[-3:]   #                   ['f', 'g', 'h']
a[2:5]    #           ['c', 'd', 'e']
a[2:-1]  #           ['c', 'd', 'e', 'f', 'g']
a[-3:-1] #           ['f', 'g']
```

切割列表时，即便 `start` 或 `end` 索引越界也不会出问题。利用这一特性，我们可以限定输入序列的最大长度[⊖]。

```
first_twenty_items = a[:20]
last_twenty_items = a[-20:]
```

反之，访问列表中的单个元素时，下标不能越界，否则会导致异常。

```
a[20]

>>>
IndexError: list index out of range
```

[⊖] 也就是只取该序列开头的 `n` 个值或末尾的 `n` 个值。——译者注



注意 请注意，如果使用负变量作为 start 索引来切割列表，那么在极个别情况下，可能会导致奇怪的结果。例如，`somelist[-n:]` 这个表达式，在 `n` 大于 1 时可以正常运作[⊖]，如当 `n` 为 3 时，`somelist[-3:]` 的结果是正常的。然而，当 `n` 为 0 时，表达式 `somelist[-0:]` 则成了原列表的一份拷贝。

对原列表进行切割之后，会产生另外一份全新的列表。系统依然维护着指向原列表中各个对象的引用。在切割后得到的新列表上进行修改，不会影响原列表。

```
b = a[4:]
print('Before: ', b)
b[1] = 99
print('After: ', b)
print('No change:', a)

>>>
Before: ['e', 'f', 'g', 'h']
After: ['e', 99, 'g', 'h']
No change: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

在赋值时对左侧列表使用切割操作，会把该列表中处在指定范围内的对象替换为新值。与元组 (tuple) 的赋值（如 `a, b = c[:2]`）不同，此切片的长度无需新值的个数相等。位于切片范围之前及之后的那些值都保留不变。列表会根据新值的个数相应地扩张或收缩。

```
print('Before ', a)
a[2:7] = [99, 22, 14]
print('After ', a)

>>>
Before ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
After ['a', 'b', 99, 22, 14, 'h']
```

如果对赋值操作右侧的列表使用切片，而把切片的起止索引都留空，那就会产生一份原列表的拷贝。

```
b = a[:]
assert b == a and b is not a
```

如果对赋值操作左侧的列表使用切片，而又没有指定起止索引，那么系统会把右侧的新值复制一份，并用这份拷贝来替换左侧列表的全部内容，而不会重新分配新的列表。

```
b = a
print('Before', a)
a[:] = [101, 102, 103]
```

[⊖] 当 `n` 等于 1 时，也可以正常运作。——译者注

```

assert a is b          # Still the same list object
print('After ', a)    # Now has different contents

>>>
Before ['a', 'b', 99, 22, 14, 'h']
After [101, 102, 103]

```

要点

- 不要写多余的代码：当 start 索引为 0，或 end 索引为序列长度时，应该将其省略。
- 切片操作不会计较 start 与 end 索引是否越界，这使得我们很容易就能从序列的前端或后端开始，对其进行范围固定的切片操作（如 a[:20] 或 a[-20:]）。
- 对 list 赋值的时候，如果使用切片操作，就会把原列表中处在相关范围内的值替换成新值，即便它们的长度不同也依然可以替换。

第 6 条：在单次切片操作内，不要同时指定 start、end 和 stride

除了基本的切片操作（参见本书第 5 条）之外，Python 还提供了 somelist[start:end:stride] 形式的写法，以实现步进式切割，也就是从每 n 个元素里面取 1 个出来。例如，可以指定步进值（stride），把列表中位于偶数索引处和奇数索引处的元素分成两组：

```

a = ['red', 'orange', 'yellow', 'green', 'blue', 'purple']
odds = a[::2]
evens = a[1::2]
print(odds)
print(evens)

>>>
['red', 'yellow', 'blue']
['orange', 'green', 'purple']

```

问题在于，采用 stride 方式进行切片时，经常会出现不符合预期的结果。例如，Python 中有一种常见的技巧，能够把以字节形式存储的字符串反转过来，这个技巧就是采用 -1 做步进值。

```

x = b'mongoose'
y = x[::-1]
print(y)

>>>
b'esoognom'

```

这种技巧对字节串和 ASCII 字符有用，但是对已经编码成 UTF-8 字节串的 Unicode

字符来说，则无法奏效。

```
w = '謝謝'
x = w.encode('utf-8')
y = x[::-1]
z = y.decode('utf-8')

>>>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x9d in
➥position 0: invalid start byte
```

除了 -1 之外，其他的负步进值有没有意义呢？请看下面的例子。

```
a = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
a[::-2] # ['a', 'c', 'e', 'g']
a[:::-2] # ['h', 'f', 'd', 'b']
```

上例中，::2 表示从头部开始，每两个元素选取一个。::−2 则表示从尾部开始，向前选取，每两个元素里选一个。

2::2 是什么意思？−2::−2、−2:2:−2 和 2:2:−2 又是什么意思？请看下面的例子。

```
a[2::2] # ['c', 'e', 'g']
a[-2::-2] # ['g', 'e', 'c', 'a']
a[-2:2:-2] # ['g', 'e']
a[2:2:-2] # []
```

通过上面几个例子可以看出：切割列表时，如果指定了 stride，那么代码可能会变得相当费解。在一对中括号里写上 3 个数字显得太过拥挤，从而导致代码难以阅读。这种写法使得 start 和 end 索引的含义变得模糊，当 stride 为负值时，尤其如此。

为了解决这种问题，我们不应该把 stride 与 start 和 end 写在一起。如果非要用 stride，那就尽量采用正值，同时省略 start 和 end 索引。如果一定要配合 start 或 end 索引来使用 stride，那么请考虑先做步进式切片，把切割结果赋给某个变量，然后在那个变量上面做第二次切割。

```
b = a[::-2] # ['a', 'c', 'e', 'g']
c = b[1:-1] # ['c', 'e']
```

上面这种先做步进切割，再做范围切割的办法[⊖]，会多产生一份原数据的浅拷贝。执行第一次切割操作时，应该尽量缩减切割后的列表尺寸。如果你所开发的程序对执行时间或内存用量的要求非常严格，以致不能采用两阶段切割法，那就请考虑 Python 内置的 `itertools` 模块。该模块中有个 `islide` 方法，这个方法不允许为 start、end 或 stride 指定负值（参见本书第 46 条）。

[⊖] 也可以先做范围切割，再做步进切割。例如，`b = a[2:6]; c = b[::-2]`。——译者注

要点

- 既有 start 和 end，又有 stride 的切割操作，可能会令人费解。
- 尽量使用 stride 为正数，且不带 start 或 end 索引的切割操作。尽量避免用负数做 stride。
- 在同一个切片操作内，不要同时使用 start、end 和 stride。如果确实需要执行这种操作，那就考虑将其拆解为两条赋值语句，其中一条做范围切割，另一条做步进切割，或考虑使用内置 `itertools` 模块中的 `islice`。

第 7 条：用列表推导来取代 map 和 filter

Python 提供了一种精练的写法，可以根据一份列表来制作另外一份。这种表达式称为 *list comprehension*（列表推导）。例如，要用列表中每个元素的平方值构建另一份列表。如果采用列表推导来实现，那就同时指定制作新列表时所要迭代的输入序列，以及计算新列表中每个元素的值时所用的表达式。

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squares = [x**2 for x in a]
print(squares)

>>>
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

除非是调用只有一个参数的函数，否则，对于简单的情况来说，列表推导要比内置的 `map` 函数更清晰。如果使用 `map`，那就要创建 `lambda` 函数，以便计算新列表中各个元素的值，这会使代码看起来有些乱。

```
squares = map(lambda x: x ** 2, a)
```

列表推导则不像 `map` 那么复杂，它可以直接过滤原列表中的元素，使得生成的新列表不会包含对应的计算结果。例如，在计算平方值时，我们只想计算那些可以为 2 所整除的数。如果采用列表推导来做，那么只需在循环后面添加条件表达式即可：

```
even_squares = [x**2 for x in a if x % 2 == 0]
print(even_squares)

>>>
[4, 16, 36, 64, 100]
```

把内置的 `filter` 函数与 `map` 结合起来，也能达成同样的效果，但是代码会写得非常难懂。

```
alt = map(lambda x: x**2, filter(lambda x: x % 2 == 0, a))
assert even_squares == list(alt)
```

字典（dict）与集（set），也有和列表类似的推导机制。编写算法时，可以通过这些推导机制来创建衍生的数据结构。

```
chile_ranks = {'ghost': 1, 'habanero': 2, 'cayenne': 3}
rank_dict = {rank: name for name, rank in chile_ranks.items()}
chile_len_set = {len(name) for name in rank_dict.values()}
print(rank_dict)
print(chile_len_set)

>>>
{1: 'ghost', 2: 'habanero', 3: 'cayenne'}
{8, 5, 7}
```

要点

- 列表推导要比内置的 map 和 filter 函数清晰，因为它无需额外编写 lambda 表达式。
- 列表推导可以跳过输入列表中的某些元素，如果改用 map 来做，那就必须辅以 filter 方能实现。
- 字典与集也支持推导表达式。

第 8 条：不要使用含有两个以上表达式的列表推导

除了基本的用法（参见本书第 7 条）之外，列表推导也支持多重循环。例如，要把矩阵（也就是包含列表的列表，即二维列表）简化成一维列表，使原来的每个单元格都成为新列表中的普通元素。这个功能采用包含两个 for 表达式的列表推导即可实现，这些 for 表达式会按照从左至右的顺序来评估。

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flat = [x for row in matrix for x in row]
print(flat)

>>>
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

上面这个例子简单易懂，这就是多重循环的合理用法。还有一种包含多重循环的合理用法，那就是根据输入列表来创建有两层深度的新列表。例如，我们要对二维矩阵中的每个单元格取平方，然后用这些平方值构建新的矩阵。由于要多使用一对中括号，所

以实现该功能的代码会比上例稍微复杂一点，但是依然不难理解。

```
squared = [[x**2 for x in row] for row in matrix]
print(squared)

>>>
[[1, 4, 9], [16, 25, 36], [49, 64, 81]]
```

如果表达式里还有一层循环，那么列表推导就会变得很长，这时必须把它分成多行来写，才能看得清楚一些。

```
my_lists = [
    [[1, 2, 3], [4, 5, 6]],
    # ...
]
flat = [x for sublist1 in my_lists
        for sublist2 in sublist1
        for x in sublist2]
```

可以看出，此时的列表推导并没有比普通的写法更加简洁。于是，笔者改用普通的循环语句来实现相同的效果。由于循环语句带有适当的缩进，所以看上去要比列表推导更清晰。

```
flat = []
for sublist1 in my_lists:
    for sublist2 in sublist1:
        flat.extend(sublist2)
```

列表推导也支持多个 if 条件。处在同一循环级别中的多项条件，彼此之间默认形成 and 表达式。例如，要从数字列表中选出大于 4 的偶数，那么下面这两种列表推导方式是等效的。

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
b = [x for x in a if x > 4 if x % 2 == 0]
c = [x for x in a if x > 4 and x % 2 == 0]
```

每一级循环的 for 表达式后面都可以指定条件。例如，要从原矩阵中把那些本身能为 3 所整除，且其所在行的各元素之和又大于等于 10 的单元格挑出来。我们只需编写很简短的代码，就可用列表推导来实现此功能，但是，这样的代码非常难懂。

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
filtered = [[x for x in row if x % 3 == 0]
            for row in matrix if sum(row) >= 10]
print(filtered)

>>>
[[6], [9]]
```

尽管这个例子稍微有点复杂，但在实际编程中，确实会出现这种看上去似乎适合用列表推导来实现的情况。笔者强烈建议大家尽量不要编写这种包含复杂式子的列表推导。这样会使其他人很难理解这段代码。这么写虽然能省下几行空间，但却会给稍后阅读代码的人带来很大障碍。

在列表推导中，最好不要使用两个以上的表达式。可以使用两个条件、两个循环或一个条件搭配一个循环。如果要写的代码比这还复杂，那就应该使用普通的 if 和 for 语句，并编写辅助函数（参见本书第 16 条）。

要点

- 列表推导支持多级循环，每一级循环也支持多项条件。
- 超过两个表达式的列表推导是很难理解的，应该尽量避免。

第 9 条：用生成器表达式来改写数据量较大的列表推导

列表推导（参见本书第 7 条）的缺点是：在推导过程中，对于输入序列中的每个值来说，可能都要创建仅含一项元素的全新列表。当输入的数据比较少时，不会出问题，但如果输入的数据非常多，那么可能会消耗大量内存，并导致程序崩溃。

例如，要读取一份文件并返回每行的字符数。若采用列表推导来做，则需把文件每一行的长度都保存在内存中。如果这个文件特别大，或是通过无休止的 network socket（网络套接字）来读取，那么这种列表推导就会出问题。下面的这段列表推导代码，只适合处理少量的输入值。

```
value = [len(x) for x in open('/tmp/my_file.txt')]
print(value)

>>>
[100, 57, 15, 1, 12, 75, 5, 86, 89, 11]
```

为了解决此问题，Python 提供了生成器表达式（*generator expression*），它是对列表推导和生成器的一种泛化（generalization）。生成器表达式在运行的时候，并不会把整个输出序列都呈现出来，而是会估值为迭代器（iterator），这个迭代器每次可以根据生成器表达式产生一项数据。

把实现列表推导所用的那种写法放在一对圆括号中，就构成了生成器表达式。下面给出的生成器表达式与刚才的代码等效。二者的区别在于，对生成器表达式求值的时

候，它会立刻返回一个迭代器，而不会深入处理文件中的内容。

```
it = (len(x) for x in open('/tmp/my_file.txt'))
print(it)

>>>
<generator object <genexpr> at 0x101b81480>
```

以刚才返回的那个迭代器为参数，逐次调用内置的 next 函数，即可使其按照生成器表达式来输出下一个值。可以根据自己的需要，多次命令迭代器根据生成器表达式来生成新值，而不用担心内存用量激增。

```
print(next(it))
print(next(it))

>>>
100
57
```

使用生成器表达式还有个好处，就是可以互相组合。下面这行代码会把刚才那个生成器表达式所返回的迭代器用作另外一个生成器表达式的输入值。

```
roots = ((x, x**0.5) for x in it)
```

外围的迭代器每次前进时，都会推动内部那个迭代器，这就产生了连锁效应，使得执行循环、评估条件表达式、对接输入和输出等逻辑都组合在了一起。

```
print(next(roots))

>>>
(15, 3.872983346207417)
```

上面这种连锁生成器表达式，可以迅速在 Python 中执行。如果要把多种手法组合起来，以操作大批量的输入数据，那最好是用生成器表达式来实现。只是要注意：由生成器表达式所返回的那个迭代器是有状态的，用过一轮之后，就不要反复使用了（参见本书第 17 条）。

要点

- 当输入的数据量较大时，列表推导可能会因为占用太多内存而出问题。
- 由生成器表达式所返回的迭代器，可以逐次产生输出值，从而避免了内存用量问题。
- 把某个生成器表达式所返回的迭代器，放在另一个生成器表达式的 for 子表达式中，即可将二者组合起来。
- 串在一起的生成器表达式执行速度很快。

第 10 条：尽量用 enumerate 取代 range

在一系列整数上面迭代时，内置的 range 函数很有用。

```
random_bits = 0
for i in range(64):
    if randint(0, 1):
        random_bits |= 1 << i
```

对于字符串列表这样的序列式数据结构，可以直接在上面迭代。

```
flavor_list = ['vanilla', 'chocolate', 'pecan', 'strawberry']
for flavor in flavor_list:
    print('%s is delicious' % flavor)
```

当迭代列表的时候，通常还想知道当前元素在列表中的索引。例如，要按照喜好程度打印出自己爱吃的冰淇淋口味。一种办法是用 range 来做。

```
for i in range(len(flavor_list)):
    flavor = flavor_list[i]
    print('%d: %s' % (i + 1, flavor))
```

与单纯迭代 flavor_list 或是单纯使用 range 的代码相比，上面这段代码有些生硬。我们必须获取列表长度，并且通过下标来访问数组^①。这种代码不便于理解。

Python 提供了内置的 enumerate 函数，以解决此问题。enumerate 可以把各种迭代器^②包装为生成器，以便稍后产生输出值。生成器每次产生一对输出值，其中，前者表示循环下标，后者表示从迭代器中获取到的下一个序列元素。这样写出来的代码会非常简洁。

```
for i, flavor in enumerate(flavor_list):
    print('%d: %s' % (i + 1, flavor))
>>>
1: vanilla
2: chocolate
3: pecan
4: strawberry
```

还可以直接指定 enumerate 函数开始计数时所用的值（本例从 1 开始计数），这样能把代码写得更短。

```
for i, flavor in enumerate(flavor_list, 1):
    print('%d: %s' % (i, flavor))
```

^① 这里的数组是一种泛称，凡是可以用递增的非负整数做下标来访问其元素的那些数据结构都不妨视为数组，并不一定专指狭义的 array。下同。——译者注

^② 也包括各种序列以及各种支持迭代的对象，比如，本例中的 flavor_list。——译者注

要点

- enumerate 函数提供了一种精简的写法，可以在遍历迭代器时获知每个元素的索引。
- 尽量用 enumerate 来改写那种将 range 与下标访问相结合的序列遍历代码。
- 可以给 enumerate 提供第二个参数，以指定开始计数时所用的值（默认为 0）。

第 11 条：用 zip 函数同时遍历两个迭代器

在编写 Python 代码时，我们通常要面对很多列表，而这些列表里的对象，可能也是相互关联的。通过列表推导，很容易就能根据某个表达式从源列表推算出一份派生类表（参见本书第 7 条）。

```
names = ['Cecilia', 'Lise', 'Marie']
letters = [len(n) for n in names]
```

对于本例中的派生列表和源列表来说，相同索引处的两个元素之间有着关联。如果想平行地迭代这两份列表，那么可以根据 names 源列表的长度来执行循环。

```
longest_name = None
max_letters = 0

for i in range(len(names)):
    count = letters[i]
    if count > max_letters:
        longest_name = names[i]
        max_letters = count

print(longest_name)
>>>
Cecilia
```

上面这段代码的问题在于，整个循环语句看上去很乱。用下标来访问 names 和 letters 会使代码不易阅读。用循环下标 *i* 来访问数组的写法一共出现了两次。改用 enumerate 来做（参见本书第 10 条）可以稍稍缓解这个问题，但仍然不够理想。

```
for i, name in enumerate(names):
    count = letters[i]
    if count > max_letters:
        longest_name = name
        max_letters = count
```

使用 Python 内置的 zip 函数，能够令上述代码变得更为简洁。在 Python 3 中的 zip 函数，可以把两个或两个以上的迭代器封装为生成器，以便稍后求值。这种 zip 生成

器，会从每个迭代器中获取该迭代器的下一个值，然后把这些值汇聚成元组（tuple）。与通过下标来访问多份列表的那种写法相比，这种用 zip 写出来的代码更加明晰。

```
for name, count in zip(names, letters):
    if count > max_letters:
        longest_name = name
        max_letters = count
```

内置的 zip 函数有两个问题。

第一个问题是，Python 2 中的 zip 并不是生成器，而是会把开发者所提供的那些迭代器，都平行地遍历一次，在此过程中，它都会把那些迭代器所产生的值汇聚成元组，并把那些元组所构成的列表完整地返回给调用者。这可能会占用大量内存并导致程序崩溃。如果要在 Python 2 里用 zip 来遍历数据量非常大的迭代器，那么应该使用 itertools 内置模块中的 izip 函数（参见本书第 46 条）。

第二个问题是，如果输入的迭代器长度不同，那么 zip 会表现出奇怪的行为。例如，我们又给 names 里添加了一个名字，但却忘了把这个名字的字母数量更新到 letters 之中。现在，如果用 zip 同时遍历这两份列表，那就会产生意外的结果。

```
names.append('Rosalind')
for name, count in zip(names, letters):
    print(name)

>>>
Cecilia
Lise
Marie
```

新元素 'Rosalind' 并没有出现在遍历结果中。这正是 zip 的运作方式。受封装的那些迭代器中，只要有一个耗尽，zip 就不再产生元组了。如果待遍历的迭代器长度都相同，那么这种运作方式不会出问题，由列表推导所推算出的派生列表一般都和源列表等长。如果待遍历的迭代器长度不同，那么 zip 会提前终止，这将会导致意外的结果。若不能确定 zip 所封装的列表是否等长，则可考虑改用 itertools 内置模块中的 zip_longest 函数（此函数在 Python 2 里叫做 izip_longest）。

要点

- 内置的 zip 函数可以平行地遍历多个迭代器。
- Python 3 中的 zip 相当于生成器，会在遍历过程中逐次产生元组，而 Python 2 中的 zip 则是直接把这些元组完全生成好，并一次性地返回整份列表。

- 如果提供的迭代器长度不等，那么 zip 就会自动提前终止。
- itertools 内置模块中的 zip_longest 函数可以平行地遍历多个迭代器，而不用在乎它们的长度是否相等（参见本书第 46 条）。

第 12 条：不要在 for 和 while 循环后面写 else 块

Python 提供了一种很多编程语言都不支持的功能，那就是可以在循环内部的语句块后面直接编写 else 块。

```
for i in range(3):
    print('Loop %d' % i)
else:
    print('Else block!')

>>>
Loop 0
Loop 1
Loop 2
Else block!
```

奇怪的是，这种 else 块会在整个循环执行完之后立刻运行。既然如此，那它为什么叫做 else 呢？为什么不叫 and？在 if/else 语句中，else 的意思是：如果不执行前面那个 if 块，那就执行 else 块。在 try/except 语句中，except 的定义也类似：如果前面那个 try 块没有成功执行，那就执行 except 块。

同理，try/except/else 也是如此（参见本书第 13 条），该结构的 else 的含义是：如果前面的 try 块没有失败，那就执行 else 块。try/finally 同样非常直观，这里的 finally 的意思是：执行过前面的 try 块之后，总是执行 finally 块。

明白了 else、except 和 finally 的含义之后，刚接触 Python 的程序员可能会把 for/else 结构中的 else 块理解为：如果循环没有正常执行完，那就执行 else 块。实际上刚好相反——在循环里用 break 语句提前跳出，会导致程序不执行 else 块。

```
for i in range(3):
    print('Loop %d' % i)
    if i == 1:
        break
else:
    print('Else block!')

>>>
Loop 0
Loop 1
```

还有一个奇怪的地方：如果 for 循环要遍历的序列是空的，那么就会立刻执行 else 块。

```
for x in []:
    print('Never runs')
else:
    print('For Else block!')

>>>
For Else block!
```

初始循环条件为 false 的 while 循环，如果后面跟着 else 块，那它也会立刻执行。

```
while False:
    print('Never runs')
else:
    print('While Else block!')

>>>
While Else block!
```

知道了循环后面的 else 块所表现出的行为之后，我们会发现：在搜索某个事物的时候，这种写法是有意义的。例如，要判断两个数是否互质（也就是判断两者除了 1 之外，是否没有其他的公约数），可以把有可能成为公约数的每个值都遍历一轮，逐个判断两数是否能以该值为公约数。尝试完每一种可能的值之后，循环就结束了。如果两个数确实互质，那么在执行循环的过程中，程序就不会因 break 语句而跳出，于是，执行完循环后，程序会紧接着执行 else 块。

```
a = 4
b = 9
for i in range(2, min(a, b) + 1):
    print('Testing', i)
    if a % i == 0 and b % i == 0:
        print('Not coprime')
        break
else:
    print('Coprime')

>>>
Testing 2
Testing 3
Testing 4
Coprime
```

实际上，我们不会这样写代码，而是会用辅助函数来完成计算。这样的辅助函数，有两种常见的写法。

第一种写法是，只要发现受测参数符合自己想要搜寻的条件，就尽早返回。如果整

个循环都完整地执行了一遍，那就说明受测参数不符合条件，于是返回默认值。

```
def coprime(a, b):
    for i in range(2, min(a, b) + 1):
        if a % i == 0 and b % i == 0:
            return False
    return True
```

第二种写法是，用变量来记录受测参数是否符合自己想要搜寻的条件。一旦符合，就用 `break` 跳出循环。

```
def coprime2(a, b):
    is_coprime = True
    for i in range(2, min(a, b) + 1):
        if a % i == 0 and b % i == 0:
            is_coprime = False
            break
    return is_coprime
```

对于不熟悉 `for/else` 的人来说，这两种写法都要比早前那种写法清晰很多。`for/else` 结构中的 `else` 块虽然也能够实现相应功能，但是将来回顾这段程序的时候，却会令阅读代码的人（包括你自己）感到相当费解。像循环这种简单的语言结构，在 Python 程序中应该写得非常直白才对。所以，我们完全不应该在循环后面使用 `else` 块。

要点

- Python 有种特殊语法，可在 `for` 及 `while` 循环的内部语句块之后紧跟一个 `else` 块。
- 只有当整个循环主体都没遇到 `break` 语句时，循环后面的 `else` 块才会执行。
- 不要在循环后面使用 `else` 块，因为这种写法既不直观，又容易引人误解。

第 13 条：合理利用 `try/except/else/finally` 结构中的每个代码块

Python 程序的异常处理可能要考虑到四种不同的时机。这些时机可以用 `try`、`except`、`else` 和 `finally` 块来表述。复合语句中的每个块都有特定的用途，它们可以构成很多种有用的组合方式（参见本书第 51 条）。

1. `finally` 块

如果既要将异常向上传播，又要在异常发生时执行清理工作，那就可以使用 `try/`

finally 结构。这种结构有一项常见的用途，就是确保程序能够可靠地关闭文件句柄（还有另外一种写法，参见本书第 43 条）。

```
handle = open('/tmp/random_data.txt') # May raise IOError
try:
    data = handle.read() # May raise UnicodeDecodeError
finally:
    handle.close() # Always runs after try:
```

在上面这段代码中，read 方法所抛出的异常会向上传播给调用方，而 finally 块中的 handle.close 方法则一定能够执行。open 方法必须放在 try 块外面，因为如果打开文件时发生异常（例如，由于找不到该文件而抛出 IOError），那么程序应该跳过 finally 块。

2. else 块

try/except/else 结构可以清晰地描述出哪些异常会由自己的代码来处理、哪些异常会传播到上一级。如果 try 块没有发生异常，那么就执行 else 块。有了这种 else 块，我们可以尽量缩减 try 块内的代码量，使其更加易读。例如，要从字符串中加载 JSON 字典数据，然后返回字典里某个键所对应的值。

```
def load_json_key(data, key):
    try:
        result_dict = json.loads(data) # May raise ValueError
    except ValueError as e:
        raise KeyError from e
    else:
        return result_dict[key] # May raise KeyError
```

如果数据不是有效的 JSON 格式，那么用 json.loads 解码时，会产生 ValueError。这个异常会由 except 块来捕获并处理。如果能够解码，那么 else 块里的查找语句就会执行，它会根据键来查出相关的值。查询时若有异常，则该异常会向上传播，因为查询语句并不在刚才那个 try 块的范围内。这种 else 子句，会把 try/except 后面的内容和 except 块本身区分开，使异常的传播行为变得更加清晰。

3. 混合使用

如果要在复合语句中把上面几种机制都用到，那就编写完整的 try/except/else/finally 结构。例如，要从文件中读取某项事务的描述信息，处理该事务，然后就地更新该文件。为了实现此功能，我们可以用 try 块来读取文件并处理其内容，用 except 块来应对 try 块中可能发生的相关异常，用 else 块实时地更新文件并把更新中可能出现的异常回报给上级代码，然后用 finally 块来清理文件句柄。

```

UNDEFINED = object()

def divide_json(path):
    handle = open(path, 'r+')      # May raise IOError
    try:
        data = handle.read()      # May raise UnicodeDecodeError
        op = json.loads(data)     # May raise ValueError
        value = (
            op['numerator'] /
            op['denominator'])   # May raise ZeroDivisionError
    except ZeroDivisionError as e:
        return UNDEFINED
    else:
        op['result'] = value
        result = json.dumps(op)
        handle.seek(0)
        handle.write(result)     # May raise IOError
        return value
    finally:
        handle.close()          # Always runs

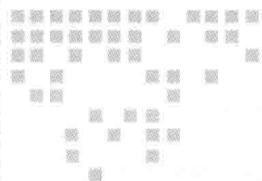
```

这种写法很有用，因为这四块代码互相配合得非常到位。例如，即使 else 块在写入 result 数据时发生异常，finally 块中关闭文件句柄的那行代码，也依然能执行。

要点

- 无论 try 块是否发生异常，都可利用 try/finally 复合语句中的 finally 块来执行清理工作。
- else 块可以用来缩减 try 块中的代码量，并把没有发生异常时所要执行的语句与 try/except 代码块隔开。
- 顺利运行 try 块后，若想使某些操作能在 finally 块的清理代码之前执行，则可将这些操作写到 else 块中[⊖]。

[⊖] 这种写法必须要有 except 块。——译者注



函 数

Python 程序员首先接触的代码组织工具，就是函数（*function*）。与其他编程语言类似，Python 的函数也可以把一大段程序分成几个小部分，使每个小部分都简单一些。这样做可以令代码更加易读，也更便于使用。函数还为复用和重构提供了契机。

Python 中的函数有很多性质，能够简化程序员的编程工作。某些性质与其他编程语言类似，另外一些则是 Python 独有的。这些性质能够彰显函数的功能、减少杂乱的语句并阐明调用者的意图，也可以有力地防止程序里出现难于查找的 bug。

第 14 条：尽量用异常来表示特殊情况，而不要返回 None

编写工具函数（utility function）时，Python 程序员喜欢给 `None` 这个返回值赋予特殊意义。这么做有时是合理的。例如，要编写辅助函数，计算两数相除的商。在除数为 0 的情况下，计算结果是没有明确含义的（undefined，未定义的），所以似乎应该返回 `None`。

```
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return None
```

此函数的调用者，可以对这种特殊的返回值做相应的解读。

```

result = divide(x, y)
if result is None:
    print('Invalid inputs')

```

分子若是 0，会怎么样呢？在那种情况下，如果分母非零，那么计算结果就是 0。当在 if 等条件语句中拿这个计算结果做判断时，会出现问题。我们可能不会专门去判断函数的返回值是否为 None，而是会假定：只要返回了与 False 等效的运算结果，就说明函数出错了（类似的用法，请参见本书第 4 条）。

```

x, y = 0, 5
result = divide(x, y)
if not result:
    print('Invalid inputs') # This is wrong!

```

如果 None 这个返回值，对函数有特殊意义，那么在编写 Python 代码来调用该函数时，就很容易犯上面这种错误。由此可见，令函数返回 None，可能会使调用它的人写出错误的代码。有两种办法可以减少这种错误。

第一种办法，是把返回值拆成两部分，并放到二元组（two-tuple）里面。二元组的首个元素，表示操作是否成功，接下来的那个元素，才是真正的运算结果。

```

def divide(a, b):
    try:
        return True, a / b
    except ZeroDivisionError:
        return False, None

```

调用该函数的人需要解析这个元组。这就迫使他们必须根据元组中表示运算状态的那个元素来做判断，而不能像从前那样，直接根据相除的结果做判断。

```

success, result = divide(x, y)
if not success:
    print('Invalid inputs')

```

问题在于，调用者可以通过以下划线为名称的变量，轻易跳过元组的第一部分（Python 程序员习惯用这种写法来表示用不到的变量）。这样写出来的代码，看上去似乎没错，但实际上，却和直接返回 None 的那种情况有着相同的错误。

```

_, result = divide(x, y)
if not result:
    print('Invalid inputs')

```

第二种办法更好一些，那就是根本不返回 None，而是把异常抛给上一级，使得调用者必须应对它。本例中，把 ZeroDivisionError 转化成 ValueError，用以表示调用者所给的输入值是无效的：

```
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError as e:
        raise ValueError('Invalid inputs') from e
```

现在，调用者就需要处理因输入值无效而引发的异常了（这种抛出异常的行为，应该写入开发文档，参见本书第 49 条）。调用者无需用条件语句来判断函数的返回值，因为如果函数没有抛出异常，返回值自然就是正确的。这样写出来的异常处理代码，也比较清晰。

```
x, y = 5, 2
try:
    result = divide(x, y)
except ValueError:
    print('Invalid inputs')
else:
    print('Result is %.1f' % result)

>>>
Result is 2.5
```

要点

- 用 None 这个返回值来表示特殊意义的函数，很容易使调用者犯错，因为 None 和 0 及空字符串之类的值，在条件表达式里都会评估为 False。
- 函数在遇到特殊情况时，应该抛出异常，而不要返回 None。调用者看到该函数的文档中所描述的异常之后，应该就会编写相应的代码来处理它们了。

第 15 条：了解如何在闭包里使用外围作用域中的变量

假如有一份列表，其中的元素都是数字，现在要对其排序，但排序时，要把出现在某个群组内的数字，放在群组外的那些数字之前。这种用法在绘制用户界面时候可能会遇到，我们可以用这个办法把重要的消息或意外的事件优先显示在其他内容前面。

实现该功能的一种常见做法，是在调用列表的 sort 方法时，把辅助函数传给 key 参数。这个辅助函数的返回值，将会用来确定列表中各元素的顺序。辅助函数可以判断受测元素是否处在重要群组中，并据此返回相应的排序关键字（sort key）。

```
def sort_priority(values, group):
    def helper(x):
```

```

if x in group:
    return (0, x)
return (1, x)
values.sort(key=helper)

```

这个函数能够应对比较简单的输入值。

```

numbers = [8, 3, 1, 2, 5, 4, 7, 6]
group = {2, 3, 5, 7}
sort_priority(numbers, group)
print(numbers)

>>>
[2, 3, 5, 7, 1, 4, 6, 8]

```

这个函数之所以能够正常运作，是基于下列三个原因：

- Python 支持闭包 (*closure*)：闭包是一种定义在某个作用域中的函数，这种函数引用了那个作用域里面的变量。`helper` 函数之所以能够访问 `sort_priority` 的 `group` 参数，原因就在于它是闭包。
- Python 的函数是一级对象 (*first-class object*)，也就是说，我们可以直接引用函数、把函数赋给变量、把函数当成参数传给其他函数，并通过表达式及 if 语句对其进行比较和判断，等等。于是，我们可以把 `helper` 这个闭包函数，传给 `sort` 方法的 `key` 参数。
- Python 使用特殊的规则来比较两个元组[⊕]。它首先比较各元组中下标为 0 的对应元素，如果相等，再比较下标为 1 的对应元素，如果还是相等，那就继续比较下标为 2 的对应元素，依次类推。

这个 `sort_priority` 函数如果能够改进一下，就更好了，它应该返回一个值，用来表示用户界面里是否出现了优先级较高的元件，使得该函数的调用者，可以根据这个返回值做出相应的处理。添加这样的功能，看似非常简单。既然该函数里的闭包函数，能够判断受测数字是否处在群组内，那么不妨在发现优先级较高的元件时，从闭包函数中翻转某个标志变量，然后令 `sort_priority` 函数把经过闭包修改的那个标志变量，返回给调用者。

我们先试试下面这种简单的写法：

```

def sort_priority2(numbers, group):
    found = False
    def helper(x):
        if x in group:
            found = True # Seems simple

```

[⊕] 这种规则也适用于两个列表之间的比较。——译者注

```

        return (0, x)
    return (1, x)
numbers.sort(key=helper)
return found

```

用刚才那些输入数据，来运行这个函数：

```

found = sort_priority2(numbers, group)
print('Found:', found)
print(numbers)

>>>
Found: False
[2, 3, 5, 7, 1, 4, 6, 8]

```

排序结果是对的，但是 `found` 值不对。`numbers` 里面的某些数字确实包含在 `group` 中，可是函数却返回了 `False`。这是为什么呢？

在表达式中引用变量时，Python 解释器将按如下顺序遍历各作用域，以解析该引用：

- 1) 当前函数的作用域。
- 2) 任何外围作用域（例如，包含当前函数的其他函数）。
- 3) 包含当前代码的那个模块的作用域（也叫全局作用域，*global scope*）。
- 4) 内置作用域（也就是包含 `len` 及 `str` 等函数的那个作用域）。

如果上面这些地方都没有定义过名称相符的变量，那就抛出 `NameError` 异常。

给变量赋值时，规则有所不同。如果当前作用域内已经定义了这个变量，那么该变量就会具备新值。若是当前作用域内没有这个变量，Python 则会把这次赋值视为对该变量的定义。而新定义的这个变量，其作用域就是包含赋值操作的这个函数。

上面所说的这种赋值行为，可以解释 `sort_priority2` 函数的返回值错误的原因。将 `found` 变量赋值为 `True`，是在 `helper` 闭包里进行的。于是，闭包中的这次赋值操作，就相当于在 `helper` 内定义了名为 `found` 的新变量，而不是给 `sort_priority2` 中的那个 `found` 赋值。

```

def sort_priority2(numbers, group):
    found = False          # Scope: 'sort_priority2'
    def helper(x):
        if x in group:
            found = True   # Scope: 'helper' -- Bad!
            return (0, x)
        return (1, x)
    numbers.sort(key=helper)
    return found

```

这种问题有时称为作用域 bug[⊖] (*scoping bug*)，它可能会使 Python 新手感到困惑。

[⊖] 也叫范围 bug。——译者注

其实，Python 语言是故意要这么设计的。这样做可以防止函数中的局部变量污染函数外面的那个模块。假如不这么做，那么函数里的每个赋值操作，都会影响外围模块的全局作用域。那样不仅显得混乱，而且由于全局变量还会与其他代码产生交互作用，所以可能引发难以探查的 bug。

1. 获取闭包内的数据

Python 3 中有一种特殊的写法，能够获取闭包内的数据。我们可以用 nonlocal 语句来表明这样的意图，也就是：给相关变量赋值的时候，应该在上层作用域中查找该变量。nonlocal 的唯一限制在于，它不能延伸到模块级别，这是为了防止它污染全局作用域。

下面用 nonlocal 来实现这个函数：

```
def sort_priority3(numbers, group):
    found = False
    def helper(x):
        nonlocal found
        if x in group:
            found = True
            return (0, x)
        return (1, x)
    numbers.sort(key=helper)
    return found
```

nonlocal 语句清楚地表明：如果在闭包内给该变量赋值，那么修改的其实是闭包外那个作用域中的变量。这与 global 语句互为补充，global 用来表示对该变量的赋值操作，将会直接修改模块作用域里的那个变量。

然而，nonlocal 也会像全局变量那样，遭到滥用，所以，建议大家只在极其简单的函数里使用这种机制。nonlocal 的副作用很难追踪，尤其是在比较长的函数中，修饰某变量的 nonlocal 语句可能和修改该变量的赋值操作离得比较远，从而导致代码更加难以理解。

如果使用 nonlocal 的那些代码，已经写得越来越复杂，那就应该将相关状态封装成辅助类（helper class）。下面定义的这个类，与 nonlocal 所达成的功能相同。它虽然有点长，但是理解起来相当容易（其中有个名叫 `_call_` 的特殊方法，详情参见本书第 23 条）。

```
class Sorter(object):
    def __init__(self, group):
        self.group = group
        self.found = False
```

```

def __call__(self, x):
    if x in self.group:
        self.found = True
        return (0, x)
    return (1, x)

sorter = Sorter(group)
numbers.sort(key=sorter)
assert sorter.found is True

```

2. Python 2 中的值

不幸的是，Python 2 不支持 `nonlocal` 关键字。为了实现类似的功能，我们需要利用 Python 的作用域规则来解决。这个做法虽然不太优雅，但已经成了一种 Python 编程习惯。

```

# Python 2
def sort_priority(numbers, group):
    found = [False]
    def helper(x):
        if x in group:
            found[0] = True
            return (0, x)
        return (1, x)
    numbers.sort(key=helper)
    return found[0]

```

运行上面这段代码时，Python 要解析 `found` 变量的当前值，于是，它会按照刚才所讲的变量搜寻规则，在上级作用域中查找这个变量。上级作用域中的 `found` 变量是个列表，由于列表本身是可供修改的（`mutable`，可变的），所以获取到这个 `found` 列表后，我们就可以在闭包里面通过 `found[0] = True` 语句，来修改 `found` 的状态。这就是该技巧的原理。

上级作用域中的变量是字典（`dictionary`）、集（`set`）或某个类的实例时，这个技巧也同样适用。

要点

- 对于定义在某作用域内的闭包来说，它可以引用这些作用域中的变量。
- 使用默认方式对闭包内的变量赋值，不会影响外围作用域中的同名变量。
- 在 Python 3 中，程序可以在闭包内用 `nonlocal` 语句来修饰某个名称，使该闭包能够修改外围作用域中的同名变量。

- 在 Python 2 中，程序可以使用可变值（例如，包含单个元素的列表）来实现与 nonlocal 语句相仿的机制。
- 除了那种比较简单的函数，尽量不要用 nonlocal 语句。

第 16 条：考虑用生成器来改写直接返回列表的函数

如果函数要产生一系列结果，那么最简单的做法就是把这些结果都放在一份列表里，并将其返回给调用者。例如，我们要查出字符串中每个词的首字母，在整个字符串里的位置。下面这段代码，用 append 方法将这些词的首字母索引添加到 result 列表中，并在函数结束时将其返回给调用者。

```
def index_words(text):
    result = []
    if text:
        result.append(0)
    for index, letter in enumerate(text):
        if letter == ' ':
            result.append(index + 1)
    return result
```

输入一些范例值，以验证该函数能够正常运作：

```
address = 'Four score and seven years ago...'
result = index_words(address)
print(result[:3])
>>>
[0, 5, 11]
```

index_words 函数有两个问题。

第一个问题是，这段代码写得有点拥挤。每次找到新的结果，都要调用 append 方法。但我们真正应该强调的，并不是对 result.append 方法的调用，而是该方法给列表中添加的那个值，也就是 index + 1。另外，函数首尾还各有一行代码用来创建及返回 result 列表。于是，在函数主体部分的约 130 个字符（不计空白字符）里，重要的大概只有 75 个。

这个函数改用生成器（*generator*）来写会更好。生成器是使用 yield 表达式的函数。调用生成器函数时，它并不会真的运行，而是会返回迭代器。每次在这个迭代器上面调用内置的 next 函数时，迭代器会把生成器推进到下一个 yield 表达式那里。生成器传给 yield 的每一个值，都会由迭代器返回给调用者。

下面的这个生成器函数，会产生和刚才那个函数相同的效果。

```
def index_words_iter(text):
    if text:
        yield 0
    for index, letter in enumerate(text):
        if letter == ' ':
            yield index + 1
```

这个函数不需要包含与 `result` 列表相交互的那些代码，因而看起来比刚才那种写法清晰许多。原来那个 `result` 列表中的元素，现在都分别传给 `yield` 表达式了。调用该生成器后所返回的迭代器，可以传给内置的 `list` 函数，以将其转换为列表（相关的原理可参见本书第 9 条）。

```
result = list(index_words_iter(address))
```

`index_words` 的第二个问题是，它在返回前，要先把所有结果都放在列表里面。如果输入量非常大，那么程序就有可能耗尽内存并崩溃。相反，用生成器改写后的版本，则可以应对任意长度的输入数据。

下面定义的这个生成器，会从文件里面依次读入各行内容，然后逐个处理每行中的单词，并产生相应结果。该函数执行时所耗的内存，由单行输入值的最大字符数来界定。

```
def index_file(handle):
    offset = 0
    for line in handle:
        if line:
            yield offset
        for letter in line:
            offset += 1
            if letter == ' ':
                yield offset
```

运行这个生成器函数，也能产生和原来相同的效果。

```
with open('/tmp/address.txt', 'r') as f:
    it = index_file(f)
    results = islice(it, 0, 3)
    print(list(results))

>>>
[0, 5, 11]
```

定义这种生成器函数的时候，唯一需要留意的就是：函数返回的那个迭代器，是有状态的，调用者不应该反复使用它（参见本书第 17 条）。

要点

- 使用生成器比把收集到的结果放入列表里返回给调用者更加清晰。
- 由生成器函数所返回的那个迭代器，可以把生成器函数体中，传给 `yield` 表达式的那些值，逐次产生出来。
- 无论输入量有多大，生成器都能产生一系列输出，因为这些输入量和输出量，都不会影响它在执行时所耗的内存。

第 17 条：在参数上面迭代时，要多加小心

如果函数接受的参数是个对象列表，那么很有可能要在这个列表上面多次迭代。例如，要分析来美国 Texas 旅游的人数。假设数据集是由每个城市的游客数量构成的（单位是每年百万人）。现在要统计来每个城市旅游的人数，占总游客数的百分比。

为此，需要编写标准化函数（normalization function）[⊖]。它会把所有的输入值加总，以求出每年的游客总数。然后，用每个城市的游客数除以总数，以求出该城市所占的比例。

```
def normalize(numbers):
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result
```

把各城市的游客数量放在一份列表里，传给该函数，可以得到正确结果。

```
visits = [15, 35, 80]
percentages = normalize(visits)
print(percentages)

>>>
[11.538461538461538, 26.923076923076923, 61.53846153846154]
```

为了扩大函数的应用范围，现在把 Texas 每个城市的游客数放在一份文件里面，然后从该文件中读取数据。由于这套流程还能够分析全世界的游客数量，所以笔者定义了生成器函数来实现此功能，以便稍后把该函数重用到更为庞大的数据集上面（参见本书第 16 条）。

[⊖] 也称正规化函数或归一函数。——译者注

```
def read_visits(data_path):
    with open(data_path) as f:
        for line in f:
            yield int(line)
```

奇怪的是，以生成器所返回的那个迭代器为参数，来调用 normalize，却没有产生任何结果。

```
it = read_visits('/tmp/my_numbers.txt')
percentages = normalize(it)
print(percentages)

>>>
[]
```

出现这种情况的原因在于，迭代器只能产生一轮结果。在抛出过 StopIteration 异常的迭代器或生成器上面继续迭代第二轮，是不会有结果的。

```
it = read_visits('/tmp/my_numbers.txt')
print(list(it))
print(list(it)) # Already exhausted

>>>
[15, 35, 80]
[]
```

通过上面这段代码，我们还可以看出一个奇怪的地方，那就是：在已经用完的[⊖]迭代器上面继续迭代时，居然不会报错。for 循环、list 构造器以及 Python 标准库里的其他许多函数，都认为在正常的操作过程中完全有可能出现 StopIteration 异常，这些函数没办法区分这个迭代器是本来就没有输出值，还是本来有输出值，但现在已经用完了。

为解决此问题，我们可以明确地使用该迭代器制作一份列表，将它的全部内容都遍历一次，并复制到这份列表里，然后，就可以在复制出来的数据列表上面多次迭代了。下面这个函数的功能，与刚才的 normalize 函数相同，只是它会把包含输入数据的那个迭代器，小心地复制一份：

```
def normalize_copy(numbers):
    numbers = list(numbers) # Copy the iterator
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result
```

这次再把调用生成器所返回的迭代器传给 normalize_copy，就能产生正确结果了：

[⊖] exhausted，也称为已耗尽的。——译者注

```

it = read_visits('/tmp/my_numbers.txt')
percentages = normalize_copy(it)
print(percentages)

>>>
[11.538461538461538, 26.923076923076923, 61.53846153846154]

```

这种写法的问题在于，待复制的那个迭代器，可能含有大量输入数据，从而导致程序在复制迭代器的时候耗尽内存并崩溃。一种解决办法是通过参数来接受另外一个函数，那个函数每次调用后，都能返回新的迭代器。

```

def normalize_func(get_iter):
    total = sum(get_iter())
    result = []
    for value in get_iter():
        percent = 100 * value / total
        result.append(percent)
    return result

```

使用 `normalize_func` 函数的时候，可以传入 `lambda` 表达式，该表达式会调用生成器，以便每次都能产生新的迭代器。

```
percentages = normalize_func(lambda: read_visits(path))
```

这种办法虽然没错，但是像上面这样传递 `lambda` 函数，毕竟显得生硬。还有个更好的办法，也能达成同样的效果，那就是新编一种实现迭代器协议 (*iterator protocol*) 的容器类。

Python 在 `for` 循环及相关表达式中遍历某种容器的内容时，就要依靠这个迭代器协议。在执行类似 `for x in foo` 这样的语句时，Python 实际上会调用 `iter(foo)`。内置的 `iter` 函数又会调用 `foo.__iter__` 这个特殊方法。该方法必须返回迭代器对象，而那个迭代器本身，则实现了名为 `_next_` 的特殊方法。此后，`for` 循环会在迭代器对象上面反复调用内置的 `next` 函数，直至其耗尽并产生 `StopIteration` 异常。

这听起来比较复杂，但实际上，只需要令自己的类把 `__iter__` 方法实现为生成器，就能满足上述要求。下面定义一个可以迭代的容器类，用来从文件中读取游客数据：

```

class ReadVisits(object):
    def __init__(self, data_path):
        self.data_path = data_path

    def __iter__(self):
        with open(self.data_path) as f:
            for line in f:
                yield int(line)

```

这种新型容器，可以直接传给原来那个 `normalize` 函数，无需再做修改，即可正常运行。

```
visits = ReadVisits(path)
percentages = normalize(visits)
print(percentages)

>>>
[11.538461538461538, 26.923076923076923, 61.53846153846154]
```

`normalize` 函数中的 `sum` 方法会调用 `ReadVisits.__iter__`，从而得到新的迭代器对象，而调整数值所用的那个 `for` 循环，也会调用 `__iter__`，从而得到另外一个新的迭代器对象，由于这两个迭代器会各自前进并走完一整轮，所以它们都可以看到全部的输入数据。这种方式的唯一缺点在于，需要多次读取输入数据。

明白了 `ReadVisits` 这种容器的工作原理之后，我们可以修改原来编写的 `normalize` 函数，以确保调用者传进来的参数，并不是迭代器对象本身。迭代器协议有这样的约定：如果把迭代器对象传给内置的 `iter` 函数，那么此函数会把该迭代器返回，反之，如果传给 `iter` 函数的是个容器类型的对象，那么 `iter` 函数则每次都会返回新的迭代器对象[⊖]。于是，我们可以根据 `iter` 函数的这种行为来判断输入值是不是迭代器对象本身，如果是，就抛出 `TypeError` 错误。

```
def normalize_defensive(numbers):
    if iter(numbers) is iter(numbers): # An iterator -- bad!
        raise TypeError('Must supply a container')
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result
```

如果我们不愿意像原来的 `normalize_copy` 那样，把迭代器中的输入数据完整复制一份，但却想多次迭代这些数据，那么上面这种写法就比较理想。这个函数能够处理 `list` 和 `ReadVisits` 这样的输入参数，因为它们都是容器。凡是遵从迭代器协议的容器类型，都与这个函数相兼容。

```
visits = [15, 35, 80]
normalize_defensive(visits) # No error
visits = ReadVisits(path)
normalize_defensive(visits) # No error
```

⊖ 详情可参阅 Python 开发文档：docs.python.org/3/library/stdtypes.html#typeiter。——译者注

如果输入的参数是迭代器而不是容器，那么此函数就会抛出异常。

```
it = iter(visits)
normalize_defensive(it)

>>>
TypeError: Must supply a container
```

要点

- 函数在输入的参数上面多次迭代时要当心：如果参数是迭代器，那么可能会导致奇怪的行为并错失某些值。
- Python 的迭代器协议，描述了容器和迭代器应该如何与 `iter` 和 `next` 内置函数、`for` 循环及相关表达式相互配合。
- 把 `__iter__` 方法实现为生成器，即可定义自己的容器类型。
- 想判断某个值是迭代器还是容器，可以拿该值为参数，两次调用 `iter` 函数，若结果相同，则是迭代器，调用内置的 `next` 函数，即可令该迭代器前进一步。

第 18 条：用数量可变的位置参数减少视觉杂讯

令函数接受可选的位置参数（由于这种参数习惯上写为 `*args`，所以又称为 *star args*，星号参数），能够使代码更加清晰，并能减少视觉杂讯（*visual noise*）[⊖]。

例如，要定义 `log` 函数，以便把某些调试信息打印出来。假如该函数的参数个数固定不变，那它就必须接受一段信息及一份含有待打印值的列表。

```
def log(message, values):
    if not values:
        print(message)
    else:
        values_str = ', '.join(str(x) for x in values)
        print('%s: %s' % (message, values_str))

log('My numbers are', [1, 2])
log('Hi there', [])

>>>
My numbers are: 1, 2
Hi there
```

[⊖] 这是一种比喻，意思是使代码看起来不要太过杂乱，以强调其中的重要内容。——译者注

即便没有值要打印，只想打印一条消息，调用者也必须像上面那样，手工传入一份空列表。这种写法既麻烦，又显得杂乱。最好是能令调用者把第二个参数完全省略掉。若想在 Python 中实现此功能，可以把最后那个位置参数前面加个 *，于是，对于现在的 log 函数来说，只有第一个参数 message 是调用者必须要指定的，该参数后面，可以跟随任意数量的位置参数。函数体不需要修改，只需修改调用该函数的代码。

```
def log(message, *values): # The only difference
    if not values:
        print(message)
    else:
        values_str = ', '.join(str(x) for x in values)
        print('%s: %s' % (message, values_str))

log('My numbers are', 1, 2)
log('Hi there') # Much better

>>>
My numbers are: 1, 2
Hi there
```

如果要把已有的列表，传给像 log 这样带有变长参数的函数，那么调用的时候，可以给列表前面加上 * 操作符。这样 Python 就会把这个列表里的元素视为位置参数。

```
favorites = [7, 33, 99]
log('Favorite colors', *favorites)

>>>
Favorite colors: 7, 33, 99
```

接受数量可变的位置参数，会带来两个问题。

第一个问题是，变长参数在传给函数时，总是要先转化成元组（tuple）。这就意味着，如果用带有 * 操作符的生成器为参数，来调用这种函数，那么 Python 就必须先把该生成器完整地迭代一轮，并把生成器所生成的每一个值，都放入元组之中。这可能会消耗大量内存，并导致程序崩溃。

```
def my_generator():
    for i in range(10):
        yield i

def my_func(*args):
    print(args)

it = my_generator()
my_func(*it)

>>>
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

只有当我们能够确定输入的参数个数比较少时，才应该令函数接受 *arg 式的变长参数。在需要把很多字面量或变量名称一起传给某个函数的场合，使用这种变长参数，是较为理想的。该参数主要是为了简化程序员的编程工作，并使得代码更加易读。

使用 *arg 参数的第二个问题是，如果以后要给函数添加新的位置参数，那就必须修改原来调用该函数的那些旧代码。若是只给参数列表前方添加新的位置参数，而不更新现有的调用代码，则会产生难以调试的错误。

```
def log(sequence, message, *values):
    if not values:
        print('%s: %s' % (sequence, message))
    else:
        values_str = ', '.join(str(x) for x in values)
        print('%s: %s: %s' % (sequence, message, values_str))
log(1, 'Favorites', 7, 33)      # New usage is OK
log('Favorite numbers', 7, 33) # Old usage breaks

>>>
1: Favorites: 7, 33
Favorite numbers: 7: 33
```

问题在于：上面的第二条 log 语句是以前写好的，当时的 log 函数还没有 sequence 参数，现在多了这个参数，使得 7 从 values 值的一部分，变成了 message 参数的值。这种 bug 很难追踪，因为现在这段代码仍然可以运行，而且不抛出异常。为了彻底避免此类情况，我们应该使用只能以关键字形式指定的参数（keyword-only argument），来扩展这种接受 *args 的函数（参见本书第 21 条）。

要点

- 在 def 语句中使用 *args，即可令函数接受数量可变的位置参数。
- 调用函数时，可以采用 * 操作符，把序列中的元素当成位置参数，传给该函数。
- 对生成器使用 * 操作符，可能导致程序耗尽内存并崩溃。
- 在已经接受 *args 参数的函数上面继续添加位置参数，可能会产生难以排查的 bug。

第 19 条：用关键字参数来表达可选的行为

与其他编程语言一样，调用 Python 函数时，可以按位置传递参数。

```
def remainder(number, divisor):
    return number % divisor

assert remainder(20, 7) == 6
```

Python 函数中的所有位置参数，都可以按关键字传递。采用关键字形式来指定参数值时，我们会在表示函数调用操作的那一对圆括号内，以赋值的格式，把参数名称和参数值分别放在等号左右两侧。关键字参数的顺序不限，只要把函数所要求的全部位置参数都指定好即可。还可以混合使用关键字参数和位置参数来调用函数。下面这些调用，都是等效的：

```
remainder(20, 7)
remainder(20, divisor=7)
remainder(number=20, divisor=7)
remainder(divisor=7, number=20)
```

位置参数必须出现在关键字参数之前。

```
>>> remainder(number=20, 7)
SyntaxError: non-keyword arg after keyword arg
```

每个参数只能指定一次。

```
>>> remainder(20, number=7)
TypeError: remainder() got multiple values for argument
  'number'
```

灵活使用关键字参数，能带来三个重要的好处。

首先，以关键字参数来调用函数，能使读到这行代码的人更容易理解其含义。如果读到了 `remainder(20, 7)` 这样的调用代码，那么必须查看方法的实现代码，才能够明白这两个参数里面，究竟哪一个是被除数，哪一个是除数。若是改用关键字的形式来调用，则立刻就能根据 `number=20` 和 `divisor=7` 等写法来获知每个参数的含义。

关键字参数的第二个好处是，它可以在函数定义中提供默认值。在大部分情况下，函数调用者只需使用这些默认值就够了，若要开启某些附加功能，则可以指定相应的关键字参数。这样做可以消除重复代码，并使代码变得整洁。

例如，要计算液体流入容器的速率。如果容器比较大，那么可以根据两个时间点上的重量差及时间差来判断流率。

```
def flow_rate(weight_diff, time_diff):
    return weight_diff / time_diff
```

```

weight_diff = 0.5
time_diff = 3
flow = flow_rate(weight_diff, time_diff)
print('%.3f kg per second' % flow)

>>>
0.167 kg per second

```

通常情况下，求出每秒钟流过的千克数就可以了。然而某些时候，可能想根据传感器上一次的读数，在更大的时间跨度上面估算流率，如以小时或天来估算。只需给函数添加一个参数，用来表示两种时间段的比例因子[⊖]，即可提供这种行为。

```

def flow_rate(weight_diff, time_diff, period):
    return (weight_diff / time_diff) * period

```

这样写的缺点是，每次调用函数时，都要指定 period 参数，即便我们想计算最常见的每秒流率，也依然要把 1 传给 period 参数。

```
flow_per_second = flow_rate(weight_diff, time_diff, 1)
```

为了使函数调用语句能写得简单一些，我们可以给 period 参数定义默认值。

```

def flow_rate(weight_diff, time_diff, period=1):
    return (weight_diff / time_diff) * period

```

现在的 period 参数，就成了可选参数。

```

flow_per_second = flow_rate(weight_diff, time_diff)
flow_per_hour = flow_rate(weight_diff, time_diff, period=3600)

```

这种办法适用于比较简单的默认值。如果默认值比较复杂，这样写就不太好了，那种情况可以参考本书第 20 条。

使用关键字参数的第三个好处，是可以提供一种扩充函数参数的有效方式，使得扩充之后的函数依然能与原有的那些调用代码相兼容。我们不需要迁移大量代码，即可给函数添加新的功能，这减少了引入 bug 的概率。

例如，要扩充上述的 flow_rate 函数，使它能够根据千克之外的其他重量单位来计算流率。为此，我们添加一个可选的参数，用以表示千克与那种重量单位之间的换算关系。

```

def flow_rate(weight_diff, time_diff,
              period=1, units_per_kg=1):
    return ((weight_diff * units_per_kg) / time_diff) * period

```

units_per_kg 参数的默认值是 1，也就是说，如果该参数取默认值，那么 flow_rate

[⊖] scaling factor，也叫换算系数或换算因数。——译者注

函数仍然会以千克为重量单位来进行计算。这可以保证原来编写的那些函数调用代码，其行为都保持不变。而现在调用 `flow_rate` 的人，则可以通过这个新的关键字参数来使用该函数的新功能。

```
pounds_per_hour = flow_rate(weight_diff, time_diff,
                             period=3600, units_per_kg=2.2)
```

这种写法只有一个缺陷，那就是像 `period` 和 `units_per_kg` 这种可选的关键字参数，仍然可以通过位置参数的形式来指定。

```
pounds_per_hour = flow_rate(weight_diff, time_diff, 3600, 2.2)
```

以位置参数的形式来指定可选参数，是容易令人困惑的，因为 3600 和 2.2 这样的值，其含义并不清晰。最好的办法，是一直以关键字的形式来指定这些参数，而决不采用位置参数来指定它们。



提示 对于接受 `*args` 的函数（参见本书第 18 条），如果要在扩充其参数的时候，与已有的那些函数调用代码保持兼容，那么就应该把新参数定义为可选的关键字参数。但是还有一种更好的办法，就是采用只能通过关键字形式来指定的参数（参见本书第 21 条）。

要点

- 函数参数可以按位置或关键字来指定。
- 只使用位置参数来调用函数，可能会导致这些参数值的含义不够明确，而关键字参数则能够阐明每个参数的意图。
- 给函数添加新的行为时，可以使用带默认值的关键字参数，以便与原有的函数调用代码保持兼容。
- 可选的关键字参数，总是应该以关键字形式来指定，而不应该以位置参数的形式来指定。

第 20 条：用 `None` 和文档字符串来描述具有动态默认值的参数

有时我们想采用一种非静态的类型，来做关键字参数的默认值。例如，在打印日志

消息的时候，要把相关事件的记录时间也标注在这条消息中。默认情况下，消息里面所包含的时间，应该是调用 log 函数那一刻的时间。如果我们以为参数的默认值会在每次执行函数时得到评估，那可能就会写出下面这种代码。

```
def log(message, when=datetime.now()):
    print('%s: %s' % (when, message))

log('Hi there!')
sleep(0.1)
log('Hi again!')

>>>
2014-11-15 21:10:10.371432: Hi there!
2014-11-15 21:10:10.371432: Hi again!
```

两条消息的时间戳 (timestamp) 是一样的，这是因为 datetime.now 只执行了一次，也就是它只在函数定义的时候执行了一次。参数的默认值，会在每个模块加载进来的时候求出，而很多模块都是在程序启动时加载的。包含这段代码的模块一旦加载进来，参数的默认值就固定不变了，程序不会再次执行 datetime.now。

在 Python 中若想正确实现动态默认值，习惯上是把默认值设为 None，并在文档字符串 (docstring) 里面把 None 所对应的实际行为描述出来 (参见本书第 49 条)。编写函数代码时，如果发现该参数的值是 None，那就将其设为实际的默认值。

```
def log(message, when=None):
    """Log a message with a timestamp.

Args:
    message: Message to print.
    when: datetime of when the message occurred.
          Defaults to the present time.
"""
    when = datetime.now() if when is None else when
    print('%s: %s' % (when, message))
```

现在，两条消息的时间戳就不同了。

```
log('Hi there!')
sleep(0.1)
log('Hi again!')

>>>
2014-11-15 21:10:10.472303: Hi there!
2014-11-15 21:10:10.573395: Hi again!
```

如果参数的实际默认值是可变类型 (mutable)，那就一定要记得用 None 作为形式上的默认值。例如，从编码为 JSON 格式的数据中载入某个值。若解码数据时失败，则

默认返回空的字典。我们可能会采用下面这种办法来实现此功能：

```
def decode(data, default={}):
    try:
        return json.loads(data)
    except ValueError:
        return default
```

这种写法的错误和刚才的 `datetime.now` 类似。由于 `default` 参数的默认值只会在模块加载时评估一次，所以凡是以默认形式来调用 `decode` 函数的代码，都将共享同一份字典。这会引发非常奇怪的行为。

```
foo = decode('bad data')
foo['stuff'] = 5
bar = decode('also bad')
bar['meep'] = 1
print('Foo:', foo)
print('Bar:', bar)

>>>
Foo: {'stuff': 5, 'meep': 1}
Bar: {'stuff': 5, 'meep': 1}
```

我们本以为 `foo` 和 `bar` 会表示两份不同的字典，每个字典里都有一对键和值，但实际上，修改了其中一个之后，另外一个似乎也会受到影响。这种错误的根本原因是：`foo` 和 `bar` 其实都等同于写在 `default` 参数默认值中的那个字典，它们都表示的是同一个字典对象。

```
assert foo is bar
```

解决办法，是把关键字参数的默认值设为 `None`，并在函数的文档字符串中描述它的实际行为。

```
def decode(data, default=None):
    """Load JSON data from a string.

Args:
    data: JSON data to decode.
    default: Value to return if decoding fails.
            Defaults to an empty dictionary.
    """
    if default is None:
        default = {}
    try:
        return json.loads(data)
    except ValueError:
        return default
```

现在，再来运行和刚才相同的测试代码，就能产生符合预期的结果了。

```
foo = decode('bad data')
foo['stuff'] = 5
bar = decode('also bad')
bar['meep'] = 1
print('Foo:', foo)
print('Bar:', bar)

>>>
Foo: {'stuff': 5}
Bar: {'meep': 1}
```

要点

- 参数的默认值，只会在程序加载模块并读到本函数的定义时评估一次。对于 {} 或 [] 等动态的值，这可能会导致奇怪的行为。
- 对于以动态值作为实际默认值的关键字参数来说，应该把形式上的默认值写为 None，并在函数的文档字符串里面描述该默认值所对应的实际行为。

第 21 条：用只能以关键字形式指定的参数来确保代码明晰

按关键字传递参数，是 Python 函数的一项强大特性（参见本书第 19 条）。由于关键字参数很灵活，所以在编写代码时，可以把函数的用法表达得更加明确。

例如，要计算两数相除的结果，同时要对计算时的特殊情况进行小心的处理。有时我们想忽略 ZeroDivisionError 异常并返回无穷。有时又想忽略 OverflowError 异常并返回 0。

```
def safe_division(number, divisor, ignore_overflow,
                  ignore_zero_division):
    try:
        return number / divisor
    except OverflowError:
        if ignore_overflow:
            return 0
        else:
            raise
    except ZeroDivisionError:
        if ignore_zero_division:
            return float('inf')
        else:
            raise
```

这个函数用起来很直观。下面这种调用方式，可以忽略除法过程中的 float 溢出，并返回 0。

```
result = safe_division(1.0, 10**500, True, False)
print(result)

>>>
0.0
```

下面这种调用方式，可以忽略拿 0 做除数的错误，并返回无穷。

```
result = safe_division(1, 0, False, True)
print(result)

>>>
inf
```

该函数用了两个 Boolean 参数，来分别决定是否应该跳过除法计算过程中的异常，而问题就在于，调用者写代码的时候，很可能分不清这两个参数，从而导致难以排查的 bug。提升代码可读性的一种办法，是采用关键字参数。在默认情况下，该函数会非常小心地进行计算，并且总是会把计算过程中发生的异常重新抛出。

```
def safe_division_b(number, divisor,
                     ignore_overflow=False,
                     ignore_zero_division=False):
    # ...
```

现在，调用者可以根据自己的具体需要，用关键字参数来覆盖 Boolean 标志的默认值，以便跳过相关的错误。

```
safe_division_b(1, 10**500, ignore_overflow=True)
safe_division_b(1, 0, ignore_zero_division=True)
```

上面这种写法还是有缺陷。由于这些关键字参数都是可选的，所以没办法确保函数的调用者一定会使用关键字来明确指定这些参数的值。即便使用新定义的 `safe_division_b` 函数，也依然可以像原来那样，以位置参数的形式调用它。

```
safe_division_b(1, 10**500, True, False)
```

对于这种复杂的函数来说，最好是能够保证调用者必须以清晰的调用代码，来阐明调用该函数的意图。在 Python 3 中，可以定义一种只能以关键字形式来指定的参数，从而确保调用该函数的代码读起来会比较明确。这些参数必须以关键字的形式提供，而不能按位置提供。

下面定义的这个 `safe_division_c` 函数，带有两个只能以关键字形式来指定的参数。参数列表里的 * 号，标志着位置参数就此终结，之后的那些参数，都只能以关键字形式

来指定。

```
def safe_division_c(number, divisor, *,
                     ignore_overflow=False,
                     ignore_zero_division=False):
    # ...
```

现在，我们就不能用位置参数的形式来指定关键字参数了。

```
safe_division_c(1, 10**500, True, False)
>>>
TypeError: safe_division_c() takes 2 positional arguments but
→4 were given
```

关键字参数依然可以用关键字的形式来指定，如果不指定，也依然会采用默认值。

```
safe_division_c(1, 0, ignore_zero_division=True) # OK

try:
    safe_division_c(1, 0)
except ZeroDivisionError:
    pass # Expected
```

在 Python 2 中实现只能以关键字来指定的参数

不幸的是，与 Python 3 不同，Python 2 并没有明确的语法来定义这种只能以关键字形式指定的参数。不过，我们可以在参数列表中使用 `**` 操作符，并且令函数在遇到无效的调用时抛出 `TypeErrors`，这样就可以实现与 Python 3 相同的功能了。`**` 操作符与 `*` 操作符类似（参见本书第 18 条），但区别在于，它不是用来接受数量可变的位置参数，而是用来接受任意数量的关键字参数。即便某些关键字参数没有定义在函数中，它也依然能够接受。

```
# Python 2
def print_args(*args, **kwargs):
    print 'Positional:', args
    print 'Keyword: ', kwargs

print_args(1, 2, foo='bar', stuff='meep')
>>>
Positional: (1, 2)
Keyword: {'foo': 'bar', 'stuff': 'meep'}
```

为了使 Python 2 版本的 `safe_division` 函数具备只能以关键字形式指定的参数，我们可以先令该函数接受 `**kwargs` 参数，然后，用 `pop` 方法把期望的关键字参数从 `kwargs` 字典里取走，如果字典的键里面没有那个关键字，那么 `pop` 方法的第二个参数就会成为

默认值。最后，为了防止调用者提供无效的参数值，我们需要确认 `kwargs` 字典里面已经没有关键字参数了。

```
# Python 2
def safe_division_d(number, divisor, **kwargs):
    ignore_overflow = kwargs.pop('ignore_overflow', False)
    ignore_zero_div = kwargs.pop('ignore_zero_division', False)
    if kwargs:
        raise TypeError('Unexpected **kwargs: %r' % kwargs)
    # ...
```

现在，既可以用不带关键字参数的方式来调用 `safe_division_d` 函数，也可以用有效的关键字参数来调用它。

```
safe_division_d(1, 10)
safe_division_d(1, 0, ignore_zero_division=True)
safe_division_d(1, 10**500, ignore_overflow=True)
```

与 Python 3 版本的函数一样，我们也不能以位置参数的形式来指定关键字参数的值。

```
safe_division_d(1, 0, False, True)

>>>
TypeError: safe_division_d() takes 2 positional arguments but 4
were given
```

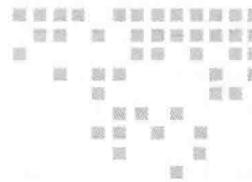
此外，调用者还不能传入不符合预期的关键字参数。

```
safe_division_d(0, 0, unexpected=True)

>>>
TypeError: Unexpected **kwargs: {'unexpected': True}
```

要点

- 关键字参数能够使函数调用的意图更加明确。
- 对于各参数之间很容易混淆的函数，可以声明只能以关键字形式指定的参数，以确保调用者必须通过关键字来指定它们。对于接受多个 Boolean 标志的函数，更应该这样做。
- 在编写函数时，Python 3 有明确的语法来定义这种只能以关键字形式指定的参数。
- Python 2 的函数可以接受 `**kwargs` 参数，并手工抛出 `TypeError` 异常，以便模拟只能以关键字形式来指定的参数。



第3章

Chapter 3

类与继承

作为一门面向对象的编程语言，Python 提供了继承、多态、封装等各种面向对象的特性。用 Python 编程时，我们经常需要编写新的类，并且需要规定这些新类的使用者应该如何通过接口与继承体系同该类相交互。

Python 的类和继承使得我们很容易在程序中表达出对象所应具备的行为，也使得我们能够随时改进程序并扩充其功能，以便灵活地应对不断变化的需求。善用类和继承，就可以写出易于维护的代码。

第 22 条：尽量用辅助类来维护程序的状态，而不要用字典和元组

Python 内置的字典类型可以很好地保存某个对象在其生命周期里的动态内部状态。所谓动态 (*dynamic*)，是指这些待保存的信息，其标识符无法提前获知。例如，要把许多学生的成绩记录下来，但这些学生的名字，我们事先并不知道。于是，可以定义一个类，把学生名字全部保存到字典里面，这样就不用把每个学生都表示成对象了，也无需在每个对象中预设一个存放其名字的属性。

```
class SimpleGradebook(object):
    def __init__(self):
        self._grades = {}

    def add_student(self, name):
```

```

        self._grades[name] = []

def report_grade(self, name, score):
    self._grades[name].append(score)
def average_grade(self, name):
    grades = self._grades[name]
    return sum(grades) / len(grades)

```

这个类用起来非常简单。

```

book = SimpleGradebook()
book.add_student('Isaac Newton')
book.report_grade('Isaac Newton', 90)
# ...
print(book.average_grade('Isaac Newton'))

>>>
90.0

```

由于字典用起来很方便，所以有可能因为功能过分膨胀而导致代码出问题。例如，要扩充 SimpleGradebook 类，使它能够按照科目来保存成绩，而不是像原来那样，把所有科目的成绩都保存到一起。要实现这个功能，可以修改 _grades 字典的结构，把每个学生的名字与另外一份字典关联起来，使得学生的名字成为 _grades 字典中每个条目的键，使得另外的那份字典成为该键所对应的值。然后，在另外那份字典中，把每个科目当作键，把该科目下的各项成绩当作值，建立映射关系。

```

class BySubjectGradebook(object):
    def __init__(self):
        self._grades = {}
    def add_student(self, name):
        self._grades[name] = {}

```

上面这部分内容改起来很简单，但是 report_grade 和 average_grade 方法就比较复杂了，因为它们需要处理嵌套了两层的字典。改动后的代码虽然比原来多，但毕竟还是可以维护的。

```

def report_grade(self, name, subject, grade):
    by_subject = self._grades[name]
    grade_list = by_subject.setdefault(subject, [])
    grade_list.append(grade)

def average_grade(self, name):
    by_subject = self._grades[name]
    total, count = 0, 0
    for grades in by_subject.values():
        total += sum(grades)
        count += len(grades)
    return total / count

```

这个类的用法仍然比较简单。

```
book = BySubjectGradebook()
book.add_student('Albert Einstein')
book.report_grade('Albert Einstein', 'Math', 75)
book.report_grade('Albert Einstein', 'Math', 65)
book.report_grade('Albert Einstein', 'Gym', 90)
book.report_grade('Albert Einstein', 'Gym', 95)
```

现在假设需求又变了。除了要记录每次考试的成绩，还需记录此成绩占该科目总成绩的权重，例如，期中考试和期末考试所占的分量，要比随堂考试大。实现该功能的方式之一，是修改内部的字典。原来我们是把科目当作键，把该科目各次考试的分数当作值，而现在，则改用一系列元组作为值，每个元组都具备(score, weight)(分数, 权重)的形式。

```
class WeightedGradebook(object):
    # ...
    def report_grade(self, name, subject, score, weight):
        by_subject = self._grades[name]
        grade_list = by_subject.setdefault(subject, [])
        grade_list.append((score, weight))
```

report_grade方法修改起来还不太难，只需要把放入grade_list中的元素从普通的成绩改为元组即可。然而average_grade方法就比较难懂了，因为它要在大循环里面嵌一层小循环。

```
def average_grade(self, name):
    by_subject = self._grades[name]
    score_sum, score_count = 0, 0
    for subject, scores in by_subject.items():
        subject_avg, total_weight = 0, 0
        for score, weight in scores:
            # ...
    return score_sum / score_count
```

这个类用起来也比刚才麻烦。我们很难从调用代码中看出这些充当位置参数的数字究竟是何含义。

```
book.report_grade('Albert Einstein', 'Math', 80, 0.10)
```

如果代码已经变得如此复杂，那么我们就该从字典和元组迁移到类体系了。

起初，我们并不知道后来需要实现带权重的分数统计，所以根据当时的复杂度来看，没有必要编写辅助类。我们很容易就能用Python内置的字典与元组类型构建出分层的数据结构，从而保存程序的内部状态。但是，当嵌套多于一层的时候，就应该避免

这种做法了（例如，不要使用包含字典的字典）。这种多层嵌套的代码，其他程序员很难看懂，而且自己维护起来也很麻烦。

用来保存程序状态的数据结构一旦变得过于复杂，就应该将其拆解为类，以便提供更为明确的接口，并更好地封装数据。这样做也能够在接口与具体实现之间创建抽象层。

把嵌套结构重构为类

我们可以从依赖关系树的最底层开始重构，也就是从每次考试的成绩开始做起。这么简单的信息，似乎没有必要专门写一个类，由于分数和权重都不再变化，所以用元组也许就足够了。于是，我们用 (score, weight) 元组来记录某科目的历次考试成绩：

```
grades = []
grades.append((95, 0.45))
# ...
total = sum(score * weight for score, weight in grades)
total_weight = sum(weight for _, weight in grades)
average_grade = total / total_weight
```

问题在于，普通的元组只是按位置来排布其中的各项数值。如果我们要给每次考试的成绩上面附加一些信息，比如，把老师的一些评语也记录上去，那就需要重新修改原来使用二元组的那些代码，因为现在每个元组里面有三项，而不是两项。下面这段代码用 _ 符号来表示每个元组的第三项，并将其跳过（Python 程序习惯用下划线表示无用的变量）：

```
grades = []
grades.append((95, 0.45, 'Great job'))
# ...
total = sum(score * weight for score, weight, _ in grades)
total_weight = sum(weight for _, weight, _ in grades)
average_grade = total / total_weight
```

与字典嵌套层级逐渐变深一样，元组长度逐步扩张，也意味着代码渐趋复杂。元组里的元素一旦超过两项，就应该考虑用其他办法来实现了。

`collections` 模块中的 `namedtuple`（具名元组）类型非常适合实现这种需求。它很容易就能定义出精简而又不可变的数据类。

```
import collections
Grade = collections.namedtuple('Grade', ('score', 'weight'))
```

构建这些具名元组时，既可以按位置指定其中各项，也可以采用关键字来指定。这些字段都可以通过属性名称访问。由于元组的属性都带有名称，所以当需求发生变化，

以致要给简单的数据容器添加新的行为时，很容易就能从 namedtuple 迁移到自己定义的类。

namedtuple 的局限

尽管 namedtuple 在很多场合都非常有用，但大家一定要明白，有些时候使用 namedtuple 反而不好。

namedtuple 类无法指定各参数的默认值。对于可选属性比较多的数据来说，namedtuple 用起来很不方便。如果这些数据并不是一系列简单的属性，那还是定义自己的类比较好。

namedtuple 实例的各项属性，依然可以通过下标及迭代来访问。这可能导致其他人以不符合设计者意图的方式使用这些元组，从而使以后很难把它迁移为真正的类，对于那种公布给外界使用的 API 来说，更要注意这个问题。如果没办法完全控制 namedtuple 实例的用法，那么最好是定义自己的类。

接下来编写表示科目的类，该类包含一系列考试成绩。

```
class Subject(object):
    def __init__(self):
        self._grades = []

    def report_grade(self, score, weight):
        self._grades.append(Grade(score, weight))

    def average_grade(self):
        total, total_weight = 0, 0
        for grade in self._grades:
            total += grade.score * grade.weight
            total_weight += grade.weight
        return total / total_weight
```

然后，可以编写表示学生的类，该类包含此学生正在学习的各项课程。

```
class Student(object):
    def __init__(self):
        self._subjects = {}

    def subject(self, name):
        if name not in self._subjects:
            self._subjects[name] = Subject()
        return self._subjects[name]
```

```
def average_grade(self):
    total, count = 0, 0
    for subject in self._subjects.values():
        total += subject.average_grade()
        count += 1
    return total / count
```

最后，编写包含所有学生考试成绩的容器类，该容器以学生的名字为键，并且可以动态地添加学生。

```
class Gradebook(object):
    def __init__(self):
        self._students = {}

    def student(self, name):
        if name not in self._students:
            self._students[name] = Student()
        return self._students[name]
```

这些类的代码量，基本上是刚才那种实现方式的两倍。但这种程序理解起来要比原来容易许多，而且使用这些类的范例代码写起来也比原来更清晰、更易扩展。

```
book = Gradebook()
albert = book.student('Albert Einstein')
math = albert.subject('Math')
math.report_grade(80, 0.10)
# ...
print(albert.average_grade())

>>>
81.5
```

如果有必要，还可以编写向后兼容的方法（backwards-compatible method），以便把使用旧式 API 的代码迁移到新的对象体系之中。

要点

- 不要使用包含其他字典的字典，也不要使用过长的元组。
- 如果容器中包含简单而又不可变的数据，那么可以先使用 namedtuple 来表示，待稍后有需要时，再修改为完整的类。
- 保存内部状态的字典如果变得比较复杂，那就应该把这些代码拆解为多个辅助类。

第 23 条：简单的接口应该接受函数，而不是类的实例

Python 有许多内置的 API，都允许调用者传入函数，以定制其行为。API 在执行的

时候，会通过这些挂钩（hook）函数，回调函数内的代码。例如，list 类型的 sort 方法接受可选的 key 参数，用以指定每个索引位置上的值之间应该如何排序。下面这段代码，用 lambda 表达式充当 key 挂钩，以便根据每个名字的长度来排序：

```
names = ['Socrates', 'Archimedes', 'Plato', 'Aristotle']
names.sort(key=lambda x: len(x))
print(names)

>>>
['Plato', 'Socrates', 'Aristotle', 'Archimedes']
```

其他编程语言可能会用抽象类来定义挂钩。然而在 Python 中，很多挂钩只是无状态的函数，这些函数有明确的参数及返回值。用函数做挂钩是比较合适的，因为它们很容易就能描述出这个挂钩的功能，而且比定义一个类要简单。Python 中的函数之所以能充当挂钩，原因就在于，它是一级（first-class）对象，也就是说，函数与方法可以像语言中的其他值那样传递和引用。

例如，要定制 defaultdict 类（参见本书第 46 条）的行为。这种数据结构允许使用者提供一个函数，以后在查询本字典时，如果里面没有待查的键，那就用这个函数为该键创建新值。当字典中没有待查询的键时，此函数必须返回那个键所应具备的默认值。下面定义的这个挂钩函数会在字典里找不到待查询的键时打印一条信息，并返回 0，以作为该键所对应的值。

```
def log_missing():
    print('Key added')
    return 0
```

一开始，我们在字典里放入一系列键值对，并给出某些值的增量，然后，就可以两次触发 log_missing 函数，并在控制台里打印两次消息（一次是在查询 'red' 键的时候，还有一次是在查询 'orange' 键的时候）。

```
current = {'green': 12, 'blue': 3}
increments = [
    ('red', 5),
    ('blue', 17),
    ('orange', 9),
]
result = defaultdict(log_missing, current)
print('Before:', dict(result))
for key, amount in increments:
    result[key] += amount
print('After: ', dict(result))

>>>
```

```
Before: {'green': 12, 'blue': 3}
Key added
Key added
After: {'orange': 9, 'green': 12, 'blue': 20, 'red': 5}
```

提供 `log_missing` 这样的函数，可以使 API 更易构建，也更易测试，因为它能够把附带的效果与确定的行为分隔开。例如，现在要给 `defaultdict` 传入一个产生默认值的挂钩，并令其统计出该字典一共遇到了多少个缺失的键。一种实现方式是使用带状态的闭包（参见本书第 15 条）。下面定义的这个辅助函数就使用这种闭包作为产生默认值的挂钩函数：

```
def increment_with_report(current, increments):
    added_count = 0

    def missing():
        nonlocal added_count # Stateful closure
        added_count += 1
        return 0

    result = defaultdict(missing, current)
    for key, amount in increments:
        result[key] += amount

    return result, added_count
```

尽管 `defaultdict` 并不知道 `missing` 挂钩函数里保存了状态，但是运行上面这个函数之后，依然可以产生预期的结果，也就是说，`count` 的值会是 2。这就是令接口接受简单函数的又一个好处。把状态隐藏到闭包里面，稍后我们就可以很方便地为闭包函数添加新的功能。

```
result, count = increment_with_report(current, increments)
assert count == 2
```

把带状态的闭包函数用作挂钩有一个缺点，就是读起来要比无状态的函数难懂一些。还有个办法也能实现上述功能，那就是定义一个小型的类，把需要追踪的状态封装起来。

```
class CountMissing(object):
    def __init__(self):
        self.added = 0

    def missing(self):
        self.added += 1
        return 0
```

在其他编程语言中，为了适配 CountMissing 的接口，我们可能要修改 defaultdict 类，而 Python 的函数则是一级函数，所以可以直接在 CountMissing 实例上面引用 CountMissing.missing 方法，并将其传给 defaultdict，以便用它作为默认值挂钩。令方法满足函数接口，是相当容易的。

```
counter = CountMissing()
result = defaultdict(counter.missing, current) # Method ref

for key, amount in increments:
    result[key] += amount
assert counter.added == 2
```

使用上述辅助类来改写带状态的闭包，确实要比 increment_with_report 函数清晰。但是，如果单看这个类，我们依然不太容易理解 CountMissing 的意图。CountMissing 对象由谁来构建？missing 方法由谁来调用？该类以后是否需要添加新的公共方法？这些问题，都必须等看过了 defaultdict 的用法之后，才能明白。

为了厘清这些问题，我们可以在 Python 代码中定义名为 `_call_` 的特殊方法。该方法使相关对象能够像函数那样得到调用。此外，如果把这样的实例传给内置的 `callable` 函数，那么 `callable` 函数会返回 `True`。

```
class BetterCountMissing(object):
    def __init__(self):
        self.added = 0

    def __call__(self):
        self.added += 1
        return 0

counter = BetterCountMissing()
counter()
assert callable(counter)
```

下面这段代码，把 BetterCountMissing 实例用作 defaultdict 的默认值挂钩，以便记录该字典在查询过程中一共添加了多少个原来所没有的键：

```
counter = BetterCountMissing()
result = defaultdict(counter, current) # Relies on __call__
for key, amount in increments:
    result[key] += amount
assert counter.added == 2
```

上面这段代码要比 CountMissing.missing 清晰许多。`__call__` 方法表明：BetterCountMissing 类的实例也会像函数那样，在合适的时候充当某个 API 的参数（例如，充

当 API 的挂钩）。于是，刚读到这段代码的人就可以从这个方法开始，来理解 BetterCountMissing 类的主要功能。`__call__` 方法强烈地暗示了该类的用途，它告诉我们，这个类的功能就相当于一个带有状态的闭包。

像上面这样修改之后，`defaultdict` 仍然无需关注 `__call__` 方法得到调用时的效果，而是只要求使用者传入一个用来生成默认值的挂钩函数即可。在 Python 程序中，我们可以通过各种手段来满足这种接受简单函数的接口，以实现自己想要的功能。

要点

- 对于连接各种 Python 组件的简单接口来说，通常应该给其直接传入函数，而不是先定义某个类，然后再传入该类的实例。
- Python 中的函数和方法都可以像一级类那样引用，因此，它们与其他类型的对象一样，也能够放在表达式里面。
- 通过名为 `__call__` 的特殊方法，可以使类的实例能够像普通的 Python 函数那样得到调用。
- 如果要用函数来保存状态，那就应该定义新的类，并令其实现 `__call__` 方法，而不要定义带状态的闭包（参见本书第 15 条）。

第 24 条：以 `@classmethod` 形式的多态去通用地构建对象

在 Python 中，不仅对象支持多态，类也支持多态。那么，类的多态是什么意思？它又有什么样的好处？

多态，使得继承体系中的多个类都能以各自所独有的方式来实现某个方法。这些类，都满足相同的接口或继承自相同的抽象类，但却有着各自不同的功能（关于多态的范例，请参见本书第 28 条）。

例如，为了实现一套 MapReduce[⊖] 流程，我们需要定义公共基类来表示输入的数据。下面这段代码就定义了这样的基类，它的 `read` 方法必须由子类来实现：

```
class InputData(object):
    def read(self):
        raise NotImplementedError
```

[⊖] 中文称为“映射—化简”或“映射—推导”。——译者注

现在编写 InputData 类的具体子类，以便从磁盘文件里读取数据：

```
class PathInputData(InputData):
    def __init__(self, path):
        super().__init__()
        self.path = path

    def read(self):
        return open(self.path).read()
```

我们可能需要很多像 PathInputData 这样的类来充当 InputData 的子类，每个子类都需要实现标准接口中的 read 方法，并以字节的形式返回待处理的数据。其他的 InputData 子类可能会通过网络读取并解压缩数据。

此外，我们还需要为 MapReduce 工作线程定义一套类似的抽象接口，以便用标准的方式来处理输入的数据。

```
class Worker(object):
    def __init__(self, input_data):
        self.input_data = input_data
        self.result = None

    def map(self):
        raise NotImplementedError

    def reduce(self, other):
        raise NotImplementedError
```

下面定义具体的 Worker 子类，以实现我们想要的 MapReduce 功能。本例所实现的功能，是一个简单的换行符计数器：

```
class LineCountWorker(Worker):
    def map(self):
        data = self.input_data.read()
        self.result = data.count('\n')

    def reduce(self, other):
        self.result += other.result
```

刚才这套 MapReduce 实现方式，看上去很好，但接下来却会遇到一个大问题，那就是如何把这些组件拼接起来。上面写的那些类，都具备合理的接口与适当的抽象，但我们必须把对象构建出来才能体现出那些类的意义。现在，由谁来负责构建对象并协调 MapReduce 流程呢？

最简单的办法是手工构建相关对象，并通过某些辅助函数将这些对象联系起来。下面这段代码可以列出某个目录的内容，并为该目录下的每个文件创建一个 PathInputData 实例：

```
def generate_inputs(data_dir):
    for name in os.listdir(data_dir):
        yield PathInputData(os.path.join(data_dir, name))
```

然后，用 generate_inputs 方法所返回的 InputData 实例来创建 LineCountWorker 实例。

```
def create_workers(input_list):
    workers = []
    for input_data in input_list:
        workers.append(LineCountWorker(input_data))
    return workers
```

现在执行这些 Worker 实例，以便将 MapReduce 流程中的 map 步骤派发到多个线程之中（参见本书第 37 条）。接下来，反复调用 reduce 方法，将 map 步骤的结果合并成一个最终值。

```
def execute(workers):
    threads = [Thread(target=w.map) for w in workers]
    for thread in threads: thread.start()
    for thread in threads: thread.join()

    first, rest = workers[0], workers[1:]
    for worker in rest:
        first.reduce(worker)
    return first.result
```

最后，把上面这些代码片段都拼装到函数里面，以便执行 MapReduce 流程的每个步骤。

```
def mapreduce(data_dir):
    inputs = generate_inputs(data_dir)
    workers = create_workers(inputs)
    return execute(workers)
```

用一系列输入文件来测试 mapreduce 函数，可以得到正常的结果。

```
from tempfile import TemporaryDirectory

def write_test_files(tmpdir):
    # ...

with TemporaryDirectory() as tmpdir:
    write_test_files(tmpdir)
    result = mapreduce(tmpdir)

print('There are', result, 'lines')
>>>
There are 4360 lines
```

但是，这种写法有个大问题，那就是 MapReduce 函数不够通用。如果要编写其他的 InputData 或 Worker 子类，那就得重写 generate_inputs、create_workers 和 mapreduce

函数，以便与之匹配。

若要解决这个问题，就需要以一种通用的方式来构建对象。在其他编程语言中，可以通过构造器多态来解决，也就是令每个 `InputData` 子类都提供特殊的构造器，使得协调 `MapReduce` 流程的那个辅助方法可以用它来通用地构造 `InputData` 对象。但是，Python 只允许名为 `_init_` 的构造器方法，所以我们不能要求每个 `InputData` 子类都提供兼容的构造器。

解决这个问题的最佳方案，是使用 `@classmethod` 形式的多态。这种多态形式，其实与 `InputData.read` 那样的实例方法多态非常相似，只不过它针对的是整个类，而不是从该类构建出来的对象。

现在我们用这套思路来实现与 `MapReduce` 流程有关的类。首先，修改 `InputData` 类，为它添加通用的 `generate_inputs` 类方法，该方法会根据通用的接口来创建新的 `InputData` 实例。

```
class GenericInputData(object):
    def read(self):
        raise NotImplementedError

    @classmethod
    def generate_inputs(cls, config):
        raise NotImplementedError
```

新添加的 `generate_inputs` 方法，接受一份含有配置参数的字典，而具体的 `GenericInputData` 子类则可以解读这些参数。下面这段代码通过 `config` 字典来查询输入文件所在的目录：

```
class PathInputData(GenericInputData):
    # ...
    def read(self):
        return open(self.path).read()

    @classmethod
    def generate_inputs(cls, config):
        data_dir = config['data_dir']
        for name in os.listdir(data_dir):
            yield cls(os.path.join(data_dir, name))
```

接下来，按照类似方式实现 `GenericWorker` 类的 `create_workers` 辅助方法。为了生成必要的输入数据，调用者必须把 `GenericInputData` 的子类传给该方法的 `input_class` 参数。该方法用 `cls()` 形式的通用构造器，来构造具体的 `GenericWorker` 子类实例。

```

class GenericWorker(object):
    # ...
    def map(self):
        raise NotImplementedError

    def reduce(self, other):
        raise NotImplementedError

    @classmethod
    def create_workers(cls, input_class, config):
        workers = []
        for input_data in input_class.generate_inputs(config):
            workers.append(cls(input_data))
        return workers

```

上面这段代码所要展示的重点，就是 `input_class.generate_inputs`，它是个类级别的多态方法。此外，我们还看到：`create_workers` 方法用另外一种方式构造了 `GenericWorker` 对象，它是通过 `cls` 形式来构造的，而不是像以前那样，直接使用 `__init__` 方法。

至于具体的 `GenericWorker` 子类，则只需修改它所继承的父类即可：

```

class LineCountWorker(GenericWorker):
    # ...

```

最后，重写 `mapreduce` 函数，令其变得完全通用。

```

def mapreduce(worker_class, input_class, config):
    workers = worker_class.create_workers(input_class, config)
    return execute(workers)

```

在测试文件上面执行修改后的 `MapReduce` 程序，所得结果与原来那套实现方式是相同的。区别在于，现在的 `mapreduce` 函数需要更多的参数，以便用更加通用的方式来操作相关对象。

```

with TemporaryDirectory() as tmpdir:
    write_test_files(tmpdir)
    config = {'data_dir': tmpdir}
    result = mapreduce(LineCountWorker, PathInputData, config)

```

我们可以继续编写其他的 `GenericInputData` 和 `GenericWorker` 子类，而且不用再修改刚才写好的那些拼接代码了。

要点

- 在 Python 程序中，每个类只能有一个构造器，也就是 `__init__` 方法。
- 通过 `@classmethod` 机制，可以用一种与构造器相仿的方式来构造类的对象。
- 通过类方法多态机制，我们能够以更加通用的方式来构建并拼接具体的子类。

第 25 条：用 super 初始化父类

初始化父类的传统方式，是在子类里用子类实例直接调用父类的 `__init__` 方法。

```
class MyBaseClass(object):
    def __init__(self, value):
        self.value = value

class MyChildClass(MyBaseClass):
    def __init__(self):
        MyBaseClass.__init__(self, 5)
```

这种办法对于简单的继承体系是可行的，但是在许多情况下会出问题。

如果子类受到了多重继承的影响（通常应该避免这种做法，请参见本书第 26 条），那么直接调用超类的 `__init__` 方法，可能会产生无法预知的行为。

在子类里调用 `__init__` 的问题之一，是它的调用顺序并不固定。例如，下面定义两个超类，它们都操作名为 `value` 的实例字段：

```
class TimesTwo(object):
    def __init__(self):
        self.value *= 2

class PlusFive(object):
    def __init__(self):
        self.value += 5
```

下面这个类用其中一种顺序来定义它所继承的各个超类。

```
class OneWay(MyBaseClass, TimesTwo, PlusFive):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        TimesTwo.__init__(self)
        PlusFive.__init__(self)
```

构建该类实例之后，我们发现，它所产生的结果与继承时的超类顺序相符。

```
foo = OneWay(5)
print('First ordering is (5 * 2) + 5 =', foo.value)

>>>
First ordering is (5 * 2) + 5 = 15
```

下面这个类，用另外一种顺序来定义它所继承的各个超类：

```
class AnotherWay(MyBaseClass, PlusFive, TimesTwo):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        TimesTwo.__init__(self)
        PlusFive.__init__(self)
```

但是，上面这段代码并没有修改超类构造器的调用顺序，它还是和以前一样，先调用 TimesTwo.`__init__`，然后才调用 PlusFive.`__init__`，这就导致该类所产生的结果与其超类的定义顺序不相符。

```
bar = AnotherWay(5)
print('Second ordering still is', bar.value)

>>>
Second ordering still is 15
```

还有一个问题发生在钻石形继承之中。如果子类继承自两个单独的超类，而那两个超类又继承自同一个公共基类，那么就构成了钻石形继承体系^①。这种继承会使钻石顶部的那个公共基类多次执行其`__init__`方法，从而产生意想不到的行为。例如，下面定义的这两个子类，都继承自 MyBaseClass。

```
class TimesFive(MyBaseClass):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        self.value *= 5

class PlusTwo(MyBaseClass):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        self.value += 2
```

然后定义一个子类，同时继承上面这两个类，这样 MyBaseClass 就成了钻石顶部的那个公共基类。

```
class ThisWay(TimesFive, PlusTwo):
    def __init__(self, value):
        TimesFive.__init__(self, value)
        PlusTwo.__init__(self, value)

foo = ThisWay(5)
print('Should be (5 * 5) + 2 = 27 but is', foo.value)

>>>
Should be (5 * 5) + 2 = 27 but is 7
```

我们可能认为输出的结果会是 27，因为 $(5*5) + 2 = 27$ ，但实际上却是 7，因为在调用第二个超类的构造器，也就是 PlusTwo.`__init__` 的时候，它会再度调用 MyBaseClass.`__init__`，从而导致 `self.value` 重新变成 5。

Python 2.2 增加了内置的 `super` 函数，并且定义了方法解析顺序^② (method resolution

^① 这种继承体系，很像竖立的菱形 (□)，也叫做菱形继承。——译者注。

^② 这种解析算法也称为 C3 linearization，其技术细节请参考 en.wikipedia.org/wiki/C3_linearization 和 www.python.org/download/releases/2.3/mro/。——译者注

order, MRO), 以解决这一问题。MRO 以标准的流程来安排超类之间的初始化顺序(例如, 深度优先、从左至右), 它也保证钻石顶部那个公共基类的 `__init__` 方法只会运行一次。

下面重新创建钻石形的继承体系, 但是这一次, 我们用 `super` 初始化超类(范例代码以 Python 2 的风格来使用 `super`):

```
# Python 2
class TimesFiveCorrect(MyBaseClass):
    def __init__(self, value):
        super(TimesFiveCorrect, self).__init__(value)
        self.value *= 5

class PlusTwoCorrect(MyBaseClass):
    def __init__(self, value):
        super(PlusTwoCorrect, self).__init__(value)
        self.value += 2
```

现在, 对于处在钻石顶部的那个 `MyBaseClass` 类来说, 它的 `__init__` 方法只会运行一次。而其他超类的初始化顺序, 则与这些超类在 `class` 语句中出现的顺序相同。

```
# Python 2
class GoodWay(TimesFiveCorrect, PlusTwoCorrect):
    def __init__(self, value):
        super(GoodWay, self).__init__(value)

foo = GoodWay(5)
print 'Should be 5 * (5 + 2) = 35 and is', foo.value
>>>
Should be 5 * (5 + 2) = 35 and is 35
```

看到上面的运行结果之后, 我们可能觉得程序的计算顺序和自己所想的刚好相反。应该先运行 `TimesFiveCorrect.__init__`, 然后运行 `PlusTwoCorrect.__init__`, 并得出 $(5*5) + 2 = 27$ 才对啊。但实际上却不是这样的。程序的运行顺序会与 `GoodWay` 类的 MRO 保持一致, 这个 MRO 顺序可以通过名为 `mro` 的类方法来查询。

```
from pprint import pprint
pprint(GoodWay.mro())
>>>
[<class '__main__.GoodWay'>,
 <class '__main__.TimesFiveCorrect'>,
 <class '__main__.PlusTwoCorrect'>,
 <class '__main__.MyBaseClass'>,
 <class 'object'>]
```

调用 `GoodWay(5)` 的时候, 它会调用 `TimesFiveCorrect.__init__`, 而 `TimesFive-`

Correct.`__init__` 又会调用 PlusTwoCorrect.`__init__`，PlusTwoCorrect.`__init__` 会调用 MyBaseClass.`__init__`。到达了钻石体系的顶部之后，所有的初始化方法会按照与刚才那些`__init__`相反的顺序来运作。于是，MyBaseClass.`__init__` 会先把 value 设为 5，然后 PlusTwoCorrect.`__init__` 会为它加 2，使 value 变成 7，最后，TimesFiveCorrect.`__init__` 会将 value 乘以 5，使其变为 35。

内置的 super 函数确实可以正常运作，但在 Python 2 中有两个问题值得注意：

- super 语句写起来有点麻烦。我们必须指定当前所在的类和 self 对象，而且还要指定相关的方法名称（通常是`__init__`）以及那个方法的参数。对于 Python 编程新手来说，这种构造方式有些费解。
- 调用 super 时，必须写出当前类的名称。由于我们以后很可能会修改类体系，所以类的名称也可能会变化，那时，必须修改每一条 super 调用语句才行。

Python 3 则没有这些问题，因为它提供了一种不带参数的 super 调用方式，该方式的效果与用`__class__` 和 self 来调用 super 相同。Python 3 总是可以通过 super 写出清晰、精练而又准确的代码。

```
class Explicit(MyBaseClass):
    def __init__(self, value):
        super().__init__(value * 2)

class Implicit(MyBaseClass):
    def __init__(self, value):
        super().__init__(value * 2)

assert Explicit(10).value == Implicit(10).value
```

由于 Python 3 程序可以在方法中通过`__class__` 变量准确地引用当前类，所以上面这种写法能够正常运作，而 Python 2 则没有定义`__class__`，故而不能采用这种写法。你可能想试着用`self.__class__` 做参数来调用 super，但实际上这么做不行，因为 Python 2 是用特殊方式来实现 super 的^①。

要点

- Python 采用标准的方法解析顺序来解决超类初始化次序及钻石继承问题。

^① 详情可参阅：[stack overflow.com/questions/18208683/when-calling-super-in-a-derived-class-can-i-pass-in-self-class](http://stackoverflow.com/questions/18208683/when-calling-super-in-a-derived-class-can-i-pass-in-self-class)。——译者注

- 总是应该使用内置的 super 函数来初始化父类。

第 26 条：只在使用 Mix-in 组件制作工具类时进行多重继承

Python 是面向对象的编程语言，它提供了一些内置的编程机制，使得开发者可以适当地实现多重继承（参见本书第 25 条）。但是，我们仍然应该尽量避开多重继承。

若一定要利用多重继承所带来的便利及封装性，那就考虑编写 *mix-in* 类[⊖]。*mix-in* 是一种小型的类，它只定义了其他类可能需要提供的一套附加方法，而不定义自己的实例属性，此外，它也不要求使用者调用自己的 `_init_` 构造器。

由于 Python 程序可以方便地查看各类对象的当前状态，所以编写 *mix-in* 比较容易。我们可以在 *mix-in* 里面通过动态检测机制先编写一套通用的功能代码，稍后再将其应用到其他很多类上面。分层地组合 *mix-in* 类可以减少重复代码并提升复用度。

例如，要把内存中的 Python 对象转换为字典形式，以便将其序列化（serialization），那我们就不妨把这个功能写成通用的代码，以供其他类使用。

下列代码定义了实现该功能所用的 *mix-in* 类，并在其中添加了一个新的 public 方法，使其他类可以通过继承这个 *mix-in* 类来具备此功能：

```
class ToDictMixin(object):
    def to_dict(self):
        return self._traverse_dict(self.__dict__)

    def _traverse_dict(self, instance_dict):
        output = {}
        for key, value in instance_dict.items():
            output[key] = self._traverse(key, value)
        return output

    def _traverse(self, key, value):
        if isinstance(value, ToDictMixin):
            return value.to_dict()
        elif isinstance(value, dict):
            return self._traverse_dict(value)
        else:
            return value
```

[⊖] 中文称为混合类、混搭类或混成类。本书中的 *mix-in* 有两种意思，既可以指实现单个功能的小组件本身，也可以指继承了这些小组件的那个类。为了避免混淆，译文酌情使用 *mix-in* 组件来称呼前者。——译者注

```

    elif isinstance(value, list):
        return [self._traverse(key, i) for i in value]
    elif hasattr(value, '__dict__'):
        return self._traverse_dict(value.__dict__)
    else:
        return value

```

下面定义的这个类演示了如何用 mix-in 把二叉树表示为字典：

```

class BinaryTree(ToDictMixin):
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

```

现在，我们可以把一大批互相关联的 Python 对象都轻松地转换成字典。

```

tree = BinaryTree(10,
                  left=BinaryTree(7, right=BinaryTree(9)),
                  right=BinaryTree(13, left=BinaryTree(11)))
print(tree.to_dict())

>>>
{'left': {'left': None,
          'right': {'left': None, 'right': None, 'value': 9},
          'value': 7},
 'right': {'left': {'left': None, 'right': None, 'value': 11},
            'right': None,
            'value': 13},
 'value': 10}

```

mix-in 的最大优势在于，使用者可以随时安插这些通用的功能，并且能在必要的时候覆写它们。例如，下面定义的这个 `BinaryTree` 子类，会持有指向父节点的引用。假如采用默认的 `ToDictMixin.to_dict` 来处理它，那么程序就会因为循环引用而陷入死循环。

```

class BinaryTreeWithParent(BinaryTree):
    def __init__(self, value, left=None,
                 right=None, parent=None):
        super().__init__(value, left=left, right=right)
        self.parent = parent

```

解决办法是在 `BinaryTreeWithParent` 类里面覆写 `ToDictMixin._traverse` 方法，令该方法只处理与序列化有关的值，从而使 mix-in 的实现代码不会陷入死循环。下面覆写的这个 `_traverse` 方法，不再遍历父节点，而是只把父节点所对应的数值插入最终生成的字典里面。

```
def _traverse(self, key, value):
```

```

if (isinstance(value, BinaryTreeNode) and
    key == 'parent'):
    return value.value # Prevent cycles
else:
    return super().__traverse(key, value)

```

现在调用 `BinaryTreeNode.to_dict` 是不会有问题的，因为程序已经不再追踪导致循环引用的那个 `parent` 属性了。

```

root = BinaryTreeNode(10)
root.left = BinaryTreeNode(7, parent=root)
root.left.right = BinaryTreeNode(9, parent=root.left)
print(root.to_dict())

>>>
{'left': {'left': None,
          'parent': 10,
          'right': {'left': None,
                     'parent': 7,
                     'right': None,
                     'value': 9},
          'value': 7},
 'parent': None,
 'right': None,
 'value': 10}

```

定义了 `BinaryTreeNode._traverse` 方法之后，如果其他类的某个属性也是 `BinaryTreeNode` 类型，那么 `ToDictMixin` 会自动地处理好这些属性。

```

class NamedSubTree(ToDictMixin):
    def __init__(self, name, tree_with_parent):
        self.name = name
        self.tree_with_parent = tree_with_parent

my_tree = NamedSubTree('foobar', root.left.right)
print(my_tree.to_dict()) # No infinite loop

>>>
{'name': 'foobar',
 'tree_with_parent': {'left': None,
                      'parent': 7,
                      'right': None,
                      'value': 9}}

```

多个 mix-in 之间也可以相互组合。例如，可以编写这样一个 mix-in，它能够为任意类提供通用的 JSON 序列化功能。我们可以假定：继承了 mix-in 的那个类，会提供名为 `to_dict` 的方法（此方法有可能是那个类通过多重继承 `ToDictMixin` 而具备的，也有可能不是）。

```

class JsonMixin(object):
    @classmethod
    def from_json(cls, data):
        kwargs = json.loads(data)
        return cls(**kwargs)

    def to_json(self):
        return json.dumps(self.to_dict())

```

请注意，JsonMixin 类既定义了实例方法，又定义了类方法。这两种行为都可以通过 mix-in 来提供。在本例中，凡是想继承 JsonMixin 的类，只需要符合两个条件即可，第一，包含名为 to_dict 的方法；第二，__init__ 方法接受关键字参数（参见本书第 19 条）。

有了这样的 mix-in 之后，我们只需编写极少量的例行代码，就可以通过继承体系，轻松地创建出相关的工具类，以便实现序列化数据以及从 JSON 中读取数据的功能。例如，我们用下面这个继承了 mix-in 组件的数据类来表示数据中心的拓扑结构：

```

class DatacenterRack(ToDictMixin, JsonMixin):
    def __init__(self, switch=None, machines=None):
        self.switch = Switch(**switch)
        self.machines = [
            Machine(**kwargs) for kwargs in machines]

class Switch(ToDictMixin, JsonMixin):
    # ...

class Machine(ToDictMixin, JsonMixin):
    # ...

```

对这样的类进行序列化，以及从 JSON 中加载它，都是比较简单的。下面的这段代码，会重复执行序列化及反序列化操作，以验证这两个功能有没有正确地实现出来。

```

serialized = """{
    "switch": {"ports": 5, "speed": 1e9},
    "machines": [
        {"cores": 8, "ram": 32e9, "disk": 5e12},
        {"cores": 4, "ram": 16e9, "disk": 1e12},
        {"cores": 2, "ram": 4e9, "disk": 500e9}
    ]
}"""

deserialized = DatacenterRack.from_json(serialized)
roundtrip = deserialized.to_json()
assert json.loads(serialized) == json.loads(roundtrip)

```

使用这种 mix-in 的时候，既可以像本例这样，直接继承多个 mix-in 组件，也可以

先令继承体系中的其他类继承相关的 mix-in 组件，然后再令本类继承那些类，以达到同样的效果。

要点

- 能用 mix-in 组件实现的效果，就不要用多重继承来做。
- 将各功能实现为可插拔的 mix-in 组件，然后令相关的类继承自己需要的那些组件，即可定制该类实例所应具备的行为。
- 把简单的行为封装到 mix-in 组件里，然后就可以用多个 mix-in 组合出复杂的行为了。

第 27 条：多用 public 属性，少用 private 属性

对 Python 类来说，其属性的可见度只有两种，也就是 *public*（公开的、公共的）和 *private*（私密的、私有的）。

```
class MyObject(object):
    def __init__(self):
        self.public_field = 5
        self.__private_field = 10

    def get_private_field(self):
        return self.__private_field
```

任何人都可以在对象上通过 dot 操作符（即 . 操作符，点操作符）来访问 public 属性。

```
foo = MyObject()
assert foo.public_field == 5
```

以两个下划线开头的属性，是 *private* 字段[⊖]。本类的方法可以直接访问它们。

```
assert foo.get_private_field() == 10
```

在类的外面直接访问 *private* 字段会引发异常。

```
foo.__private_field
>>>
AttributeError: 'MyObject' object has no attribute
'__private_field'
```

由于类级别的方法仍然声明在本类的 class 代码块之内，所以，这些方法也是能够

[⊖] 严格来说，*private* 属性是开头至少有两个下划线，且结尾至多有一个下划线的属性。——译者注

访问 private 属性的。

```
class MyOtherObject(object):
    def __init__(self):
        self.__private_field = 71

    @classmethod
    def get_private_field_of_instance(cls, instance):
        return instance.__private_field

bar = MyOtherObject()
assert MyOtherObject.get_private_field_of_instance(bar) == 71
```

正如大家所预料的那样，子类无法访问父类的 private 字段。

```
class MyParentObject(object):
    def __init__(self):
        self.__private_field = 71

class MyChildObject(MyParentObject):
    def get_private_field(self):
        return self.__private_field

baz = MyChildObject()
baz.get_private_field()

>>>
AttributeError: 'MyChildObject' object has no attribute
➥ '_MyChildObject__private_field'
```

Python 会对私有属性的名称做一些简单的变换，以保证 private 字段的私密性。

当编译器看到 MyChildObject.get_private_field 方法要访问私有属性时，它会先把 __private_field 变换为 _MyChildObject__private_field，然后再进行访问。本例中，__private_field 字段只在 MyParentObject.__init__ 里面做了定义，因此，这个私有属性的真实名称，实际上是 _MyParentObject__private_field。子类之所以无法访问父类的私有属性，只不过是因为变换后的属性名与待访问的属性名不相符而已。

了解这套机制之后，我们就可以从任意类中访问相关类的私有属性了。无论是从该类的子类访问，还是从外部访问，我们都不受制于 private 属性的访问权限。

```
assert baz._MyParentObject__private_field == 71
```

查询该对象的属性字典，我们就能看到，私有属性实际上是按变换后的名称来保存的。

```
print(baz.__dict__)

>>>
{'_MyParentObject__private_field': 71}
```

Python 为什么不从语法上严格保证 private 字段的私密性呢？用最简单的话来说，就是：We are all consenting adults here（我们都是成年人）。这句广为流传的格言，表达了很多 Python 程序员的观点，大家都认为开放要比封闭好。

另外一个原因在于：Python 语言本身就已经提供了一些属性挂钩（参见本书第 32 条），使得开发者能够按照自己的需要来操作对象内部的数据。既然如此，那为什么还要阻止访问 private 属性呢？

为了尽量减少无意间访问内部属性所带来的意外，Python 程序员会遵照《Python 风格指南》（参见本书第 2 条）的建议，用一种习惯性的命名方式来表示这种字段。也就是说：以单个下划线开头的字段，应该视为 *protected*（受保护的）字段，本类之外的那些代码在使用这种字段的时候要多加小心。

虽说有了这种约定，但仍然有很多 Python 编程新手会使用私有字段表示那种不应该由子类或外部来访问的 API。

```
class MyClass(object):
    def __init__(self, value):
        self.__value = value

    def get_value(self):
        return str(self.__value)

foo = MyClass(5)
assert foo.get_value() == '5'
```

这种写法不好。因为包括你自己在内的许多开发者，以后可能都需要从这个类中继承子类，并在子类里面添加新的行为，或是改进现有方法中效率不高的那些部分（例如，上面那个类的 `MyClass.get_value` 方法总是返回字符串，这可能就需要改进）。假如超类使用了 `private` 属性，那么子类在覆写或扩展的时候，就会遇到麻烦和错误。继承该类的那些子类，在万不得已的时候，仍然要去访问 `private` 字段。

```
class MyIntegerSubclass(MyClass):
    def get_value(self):
        return int(self._MyClass__value)

foo = MyIntegerSubclass(5)
assert foo.get_value() == 5
```

可是，一旦子类后面的那套继承体系发生变化，这些对 `private` 字段的引用代码就会失效，从而导致子类出现错误。就本例来说，`MyIntegerSubclass` 类的直接超类 `MyClass` 本来是直接继承自 `object` 的，但是现在，我们改令其继承自名为 `MyBaseClass` 的类：

```

class MyBaseClass(object):
    def __init__(self, value):
        self._value = value
    # ...

class MyClass(MyBaseClass):
    # ...

class MyIntegerSubclass(MyClass):
    def get_value(self):
        return int(self._MyClass__value)

```

原来的 `_value` 属性是在 `MyIntegerSubclass` 的直接超类 `MyClass` 里面赋值的，但是现在，我们把它上移，放在 `MyClass` 的超类 `MyBaseClass` 里面来赋值。于是，`MyIntegerSubclass` 类里面指向私有变量 `self._MyClass__value` 的引用就失效了。

```

foo = MyIntegerSubclass(5)
foo.get_value()

>>>
AttributeError: 'MyIntegerSubclass' object has no attribute
➥'_MyClass__value'

```

一般来说，恰当的做法应该是：宁可叫子类更多地去访问超类的 `protected` 属性，也别把这些属性设成 `private`。我们应该在文档中说明每个 `protected` 字段的含义，解释哪些字段是可供子类使用的内部 API、哪些字段是完全不应触碰的数据。这种建议信息，不仅可以给其他程序员看，而且也能在将来扩展代码的时候提醒自己，应该如何保证数据安全。

```

class MyClass(object):
    def __init__(self, value):
        # This stores the user-supplied value for the object.
        # It should be coercible to a string. Once assigned for
        # the object it should be treated as immutable.
        self._value = value

```

只有一种情况是可以合理使用 `private` 属性的，那就是用它来避免子类的属性名与超类相冲突。如果子类无意中定义了与超类同名的属性，那么程序就可能出问题。

```

class ApiClass(object):
    def __init__(self):
        self._value = 5

    def get(self):
        return self._value

class Child(ApiClass):
    def __init__(self):

```

```

super().__init__()
self._value = 'hello' # Conflicts

a = Child()
print(a.get(), 'and', a._value, 'should be different')

>>>
hello and hello should be different

```

当超类是公共 API 的一部分时，可能就需要考虑上面这个问题了。由于子类不在开发者的控制范围之内，所以我们不能通过重构来解决这种冲突。如果属性名称是个很常用的词语（如本例中的 value），那么更容易引发重名现象。为了降低风险，我们可以在超类中使用 `private` 属性，以确保子类的属性名不会与之重复。

```

class ApiClass(object):
    def __init__(self):
        self._value = 5

    def get(self):
        return self._value

class Child(ApiClass):
    def __init__(self):
        super().__init__()
        self._value = 'hello' # OK!

a = Child()
print(a.get(), 'and', a._value, 'are different')

>>>
5 and hello are different

```

要点

- Python 编译器无法严格保证 `private` 字段的私密性。
- 不要盲目地将属性设为 `private`，而是应该从一开始就做好规划，并允许子类更多地访问超类的内部 API。
- 应该多用 `protected` 属性，并在文档中把这些字段的合理用法告诉子类的开发者，而不要试图用 `private` 属性来限制子类访问这些字段。
- 只有当子类不受自己控制时，才可以考虑用 `private` 属性来避免名称冲突。

第 28 条：继承 `collections.abc` 以实现自定义的容器类型

大部分的 Python 编程工作，其实都是在定义类。类可以包含数据，并且能够描述

出这些数据对象之间的交互方式。Python 中的每一个类，从某种程度上来说都是容器，它们都封装了属性与功能。Python 也直接提供了一些管理数据所用的内置容器类型，例如，list（列表）、tuple（元组）、set（集）、dictionary（字典）等。

如果要设计用法比较简单的序列，那我们自然就会想到直接继承 Python 内置的 list 类型。例如，要创建一种自定义的列表类型，并提供统计各元素出现频率的方法。

```
class FrequencyList(list):
    def __init__(self, members):
        super().__init__(members)

    def frequency(self):
        counts = {}
        for item in self:
            counts.setdefault(item, 0)
            counts[item] += 1
        return counts
```

上面这个 FrequencyList 类继承了 list，并获得了由 list 所提供的全部标准功能，使得所有 Python 程序员都可以用他们所熟悉的写法来使用这个类。此外，我们还根据自己的需求，在子类里添加了其他的方法，以定制其行为。

```
foo = FrequencyList(['a', 'b', 'a', 'c', 'b', 'a', 'd'])
print('Length is', len(foo))
foo.pop()
print('After pop:', repr(foo))
print('Frequency:', foo.frequency())

>>>
Length is 7
After pop: ['a', 'b', 'a', 'c', 'b', 'a']
Frequency: {'a': 3, 'c': 1, 'b': 2}
```

现在，假设要编写这么一种对象：它本身虽然不属于 list 子类，但是用起来却和 list 一样，也可以通过下标访问其中的元素。例如，我们要令下面这个表示二叉树节点的类，也能够像 list 或 tuple 等序列那样来访问。

```
class BinaryNode(object):
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right
```

上面这个类，如何才能够表现得和序列类型一样呢？我们可以通过特殊方法完成此功能。Python 会用一些名称比较特殊的实例方法，来实现与容器有关的行为。用下标访问序列中的元素时：

```
bar = [1, 2, 3]
bar[0]
```

Python 会把访问代码转译为：

```
bar.__getitem__(0)
```

于是，我们提供自己定制的 `__getitem__` 方法，令 `BinaryNode` 类可以表现得和序列一样。下面这个方法按深度优先的次序来访问二叉树中的对象：

```
class IndexableNode(BinaryNode):
    def _search(self, count, index):
        # ...
        # Returns (found, count)

    def __getitem__(self, index):
        found, _ = self._search(0, index)
        if not found:
            raise IndexError('Index out of range')
        return found.value
```

构建二叉树的代码，依然与平常一样。

```
tree = IndexableNode(
    10,
    left=IndexableNode(
        5,
        left=IndexableNode(2),
        right=IndexableNode(
            6, right=IndexableNode(7))),
    right=IndexableNode(
        15, left=IndexableNode(11)))
```

但是访问它的时候，除了可以像普通的二叉树那样进行遍历之外，还可以使用与 `list` 相同的写法来访问树中的元素。

```
print('LRR =', tree.left.right.right.value)
print('Index 0 =', tree[0])
print('Index 1 =', tree[1])
print('11 in the tree?', 11 in tree)
print('17 in the tree?', 17 in tree)
print('Tree is', list(tree))

>>>
LRR = 7
Index 0 = 2
Index 1 = 5
11 in the tree? True
17 in the tree? False
Tree is [2, 5, 6, 7, 10, 11, 15]
```

然而只实现 `__getitem__` 方法是不够的，它并不能使该类型支持我们想要的每一种序列操作。

```
len(tree)

>>>
TypeError: object of type 'IndexableNode' has no len()
```

想要使内置的 `len` 函数正常运作，就必须在自己定制的序列类型中实现另外一个名叫 `__len__` 的特殊方法。

```
class SequenceNode(IndexableNode):
    def __len__(self):
        _, count = self._search(0, None)
        return count

tree = SequenceNode(
    # ...
)

print('Tree has %d nodes' % len(tree))

>>>
Tree has 7 nodes
```

实现了 `__len__` 方法之后，这个类的功能依然不完整。其他 Python 程序员还希望这个序列能够像 `list` 或 `tuple` 那样，提供 `count` 和 `index` 方法。这样看来，定义自己的容器类型，似乎要比想象中困难得多。

为了在编写 Python 程序时避免这些麻烦，我们可以使用内置的 `collections.abc` 模块。该模块定义了一系列抽象基类，它们提供了每一种容器类型所应具备的常用方法。从这样的基类中继承了子类之后，如果忘记实现某个方法，那么 `collections.abc` 模块就会指出这个错误。

```
from collections.abc import Sequence

class BadType(Sequence):
    pass

foo = BadType()

>>>
TypeError: Can't instantiate abstract class BadType with
►abstract methods __getitem__, __len__
```

如果子类已经实现了抽象基类所要求的每个方法，那么基类就会自动提供剩下的那些方法。例如，刚才的 `SequenceNode` 类就满足这一点，于是，它会自动具备 `index` 和

count 等方法。

```
class BetterNode(SequenceNode, Sequence):
    pass

tree = BetterNode(
    # ...
)

print('Index of 7 is', tree.index(7))
print('Count of 10 is', tree.count(10))

>>>
Index of 7 is 3
Count of 10 is 1
```

对于 Set 和 MutableMapping 等更为复杂的容器类型来说，若不继承抽象基类，则必须实现非常多的特殊方法，才能令自己所定制的子类符合 Python 编程习惯。在这种情况下，继承抽象基类所带来的好处会更加明显。

要点

- 如果要定制的子类比较简单，那就可以直接从 Python 的容器类型（如 list 或 dict）中继承。
- 想正确实现自定义的容器类型，可能需要编写大量的特殊方法。
- 编写自制的容器类型时，可以从 collections.abc 模块的抽象基类中继承，那些基类能够确保我们的子类具备适当的接口及行为。

元类及属性

谈论 Python 语言的特性时，经常会提到元类，但是很少有人理解它的实际用途。*metaclass*（元类）这个词，只是模糊地描述了一种高于类，而又超乎类的概念[⊖]。简单来说，就是我们可以把 Python 的 class 语句转译为元类，并令其在每次定义具体的类时，都提供独特的行为。

Python 还有另外一个奇妙的特性，那就是可以动态地定义对属性的访问操作。把这种动态属性机制与 Python 的面向对象机制相结合，就可以非常顺利地将简单的类逐渐变换为复杂的类。

但是，这些强大的功能也有弊端。动态属性可能会覆盖对象的某些行为，从而产生令人意外的副作用。元类也可能会创建出极其古怪的程序，使得刚接触 Python 的程序员无所适从。使用这些特性时，要遵循最小惊讶原则（*rule of least surprise*，最少意外原则），也就是说，这些机制只适合用来实现那些广为人知的 Python 编程范式。

第 29 条：用纯属性取代 get 和 set 方法

从其他语言转入 Python 的开发者，可能会在类中明确地实现 getter（获取器）和 setter（设置器）方法。

[⊖] 也可以理解为：能够定制其他类的类。又称为描述类。——译者注

```
class OldResistor(object):
    def __init__(self, ohms):
        self._ohms = ohms

    def get_ohms(self):
        return self._ohms

    def set_ohms(self, ohms):
        self._ohms = ohms
```

这种 setter 和 getter 用起来虽然简单，但却不像 Python 的编程风格。

```
r0 = OldResistor(50e3)
print('Before: %5r' % r0.get_ohms())
r0.set_ohms(10e3)
print('After:  %5r' % r0.get_ohms())

>>>
Before: 50000.0
After:  10000.0
```

对于就地自增的代码来说，这种用法尤其显得麻烦。

```
r0.set_ohms(r0.get_ohms() + 5e3)
```

setter 和 getter 等工具方法，确实有助于定义类的接口，它也使得开发者能够更加方便地封装功能、验证用法并限定取值范围。在设计自己的类时，应该认真考虑这些机制，因为自己的类以后可能会逐渐进化，而这些机制可以确保进化过程不会影响原有的调用代码。

但是，对于 Python 语言来说，基本上不需要手工实现 setter 或 getter 方法，而是应该先从简单的 public 属性开始写起。

```
class Resistor(object):
    def __init__(self, ohms):
        self.ohms = ohms
        self.voltage = 0
        self.current = 0

r1 = Resistor(50e3)
r1.ohms = 10e3
```

改用简单的属性来实现 Resistor 类之后，原地自增操作就变得清晰而自然了。

```
r1.ohms += 5e3
```

以后如果想在设置属性的时候实现特殊行为，那么可以改用 @property 修饰器和 setter 方法来做。下面这个子类继承自 Resistor，它在给 voltage(电压) 属性赋值的时候，

还会同时修改 current (电流) 属性。请注意：setter 和 getter 方法的名称必须与相关属性相符，方能使这套机制正常运作。

```
class VoltageResistance(Resistor):
    def __init__(self, ohms):
        super().__init__(ohms)
        self._voltage = 0

    @property
    def voltage(self):
        return self._voltage

    @voltage.setter
    def voltage(self, voltage):
        self._voltage = voltage
        self.current = self._voltage / self.ohms
```

现在，设置 voltage 属性时，将会执行名为 voltage 的 setter 方法，该方法会更新本对象的 current 属性，令其与电压和电阻相匹配。

```
r2 = VoltageResistance(1e3)
print('Before: %5r amps' % r2.current)
r2.voltage = 10
print('After: %5r amps' % r2.current)

>>>
Before:    0 amps
After:   0.01 amps
```

为属性指定 setter 方法时，也可以在方法里面做类型验证及数值验证。下面定义的这个类，可以保证传入的电阻值总是大于 0 欧姆：

```
class BoundedResistance(Resistor):
    def __init__(self, ohms):
        super().__init__(ohms)

    @property
    def ohms(self):
        return self._ohms

    @ohms.setter
    def ohms(self, ohms):
        if ohms <= 0:
            raise ValueError('%f ohms must be > 0' % ohms)
        self._ohms = ohms
```

如果传入无效的属性值，程序就会抛出异常。

```
r3 = BoundedResistance(1e3)
r3.ohms = 0
```

```
>>>
ValueError: 0.000000 ohms must be > 0
```

给构造器传入无效数值，同样会引发异常。

```
BoundedResistance(-5)
```

```
>>>
```

```
ValueError: -5.000000 ohms must be > 0
```

之所以会抛出异常，是因为 `BoundedResistance.__init__` 会调用 `Resistor.__init__`，而 `Resistor.__init__` 又会执行 `self.ohms = -5`。这条赋值语句使得 `BoundedResistance` 中的 `@ohms.setter` 得到执行，于是，在对象构造完毕之前，程序会先运行 `setter` 里面的验证代码。

我们甚至可以用 `@property` 来防止父类的属性遭到修改。

```
class FixedResistance(Resistor):
    # ...
    @property
    def ohms(self):
        return self._ohms

    @ohms.setter
    def ohms(self, ohms):
        if hasattr(self, '_ohms'):
            raise AttributeError("Can't set attribute")
        self._ohms = ohms
```

构建好对象之后，如果试图修改 `ohms` 属性，那就会引发异常。

```
r4 = FixedResistance(1e3)
r4.ohms = 2e3
```

```
>>>
AttributeError: Can't set attribute
```

`@property` 的最大缺点在于：和属性相关的方法，只能在子类里面共享，而与之无关的其他类，则无法复用同一份实现代码。不过，Python 也提供了描述符机制（参见本书第 31 条），开发者可以通过它来复用与属性有关的逻辑，此外，描述符还有其他一些用法。

最后，要注意用 `@property` 方法来实现 `setter` 和 `getter` 时，不要把程序的行为写得太过奇怪。例如，我们不应该在某属性的 `getter` 方法里面修改其他属性的值。

```
class MysteriousResistor(Resistor):
    @property
    def ohms(self):
        self.voltage = self._ohms * self.current
        return self._ohms
    # ...
```

上面这种写法，会导致非常古怪的行为。

```
r7 = MysteriousResistor(10)
r7.current = 0.01
print('Before: %5r' % r7.voltage)
r7.ohms
print('After: %5r' % r7.voltage)

>>>
Before:    0
After:    0.1
```

最恰当的做法是：只在 `@property.setter` 里面修改相关的对象状态，而且要防止该对象产生调用者所不希望有的副作用。例如，不应该动态地引入模块、不应该执行缓慢的辅助函数，也不应该执行开销比较大的数据库查询操作等。我们所编写的类，要迅速为用户返回简单的属性，而这种属性，也应该和其他 Python 对象一样，非常易于使用。至于那些比较复杂或速度比较慢的操作，还是应该放在普通的方法里面。

要点

- 编写新类时，应该用简单的 `public` 属性来定义其接口，而不要手工实现 `set` 和 `get` 方法。
- 如果访问对象的某个属性时，需要表现出特殊的行为，那就用 `@property` 来定义这种行为。
- `@property` 方法应该遵循最小惊讶原则，而不应产生奇怪的副作用。
- `@property` 方法需要执行得迅速一些，缓慢或复杂的工作，应该放在普通的方法里面。

第 30 条：考虑用 `@property` 来代替属性重构

Python 内置的 `@property` 修饰器，使开发者可以把类设计得较为灵巧，从而令调用者能够轻松地访问该类的实例属性（参见本书第 29 条）。此外，`@property` 还有一种高级的用法，就是可以把简单的数值属性迁移为实时计算（on-the-fly calculation，按需计算、动态计算）的属性，这种用法也是比较常见的。采用 `@property` 来迁移属性时，我们只需要给本类添加新的功能，原有的那些调用代码都不需要修改，因此，它是一种非常有效的编程手法。而且在持续完善接口的过程中，它也是一种重要的缓冲方案。

例如，要用纯 Python 对象实现带有配额的漏桶^①。下面这段代码，把当前剩余的配额以及重置配额的周期，放在了 Bucket 类里面：

```
class Bucket(object):
    def __init__(self, period):
        self.period_delta = timedelta(seconds=period)
        self.reset_time = datetime.now()
        self.quota = 0

    def __repr__(self):
        return 'Bucket(quota=%d)' % self.quota
```

漏桶算法若要正常运作，就必须保证：无论向桶中加多少水，都必须在进入下一个周期时将其清空^②。

```
def fill(bucket, amount):
    now = datetime.now()
    if now - bucket.reset_time > bucket.period_delta:
        bucket.quota = 0
        bucket.reset_time = now
    bucket.quota += amount
```

每次在执行消耗配额的操作之前，都必须先确认桶里有足够的配额可供使用。

```
def deduct(bucket, amount):
    now = datetime.now()
    if now - bucket.reset_time > bucket.period_delta:
        return False
    if bucket.quota - amount < 0:
        return False
    bucket.quota -= amount
    return True
```

使用这个类的对象之前，要先往桶里添水。

```
bucket = Bucket(60)
fill(bucket, 100)
print(bucket)

>>>
Bucket(quota=100)
```

然后，消耗自己所需的配额。

```
if deduct(bucket, 99):
    print('Had 99 quota')
else:
```

① 漏桶算法是一种具备传输、调度和统计等用途的算法。它把容器比作底部有漏洞的桶（leaky bucket），而把配额（quota）比作桶底漏出的水。——译者注
 ② 如果加水之前发现桶里原有的水已经过期，那就必须先把水全部倒掉，然后再加。——译者注

```

    print('Not enough for 99 quota')
print(bucket)

```

```

>>>
Had 99 quota
Bucket(quota=1)

```

这样继续消耗下去，最终会导致待消耗的配额比剩余的配额还多。到了那时，Bucket 对象就会阻止这一操作，而漏桶中剩余的配额则将保持不变。

```

if deduct(bucket, 3):
    print('Had 3 quota')
else:
    print('Not enough for 3 quota')
print(bucket)

>>>
Not enough for 3 quota
Bucket(quota=1)

```

上面这种实现方式的缺点是：以后无法得知漏桶的初始配额。配额会在每个周期内持续流失，如果降到 0，那么 deduct 就总是会返回 False。此时，依赖 deduct 的那些操作，就会受到阻塞，但是，我们却无法判断出：这究竟是由于 Bucket 里面所剩的配额不足，还是由于 Bucket 刚开始的时候根本没有配额。

为了解决这一问题，我们在类中使用 max_quota 来记录本周期的初始配额，并且用 quota_consumed 来记录本周期内所消耗的配额。

```

class Bucket(object):
    def __init__(self, period):
        self.period_delta = timedelta(seconds=period)
        self.reset_time = datetime.now()
        self.max_quota = 0
        self.quota_consumed = 0

    def __repr__(self):
        return ('Bucket(max_quota=%d, quota_consumed=%d)' %
               (self.max_quota, self.quota_consumed))

```

我们可以根据这两个新属性，在 @property 方法里面实时地算出当前所剩的配额。

```

@property
def quota(self):
    return self.max_quota - self.quota_consumed

```

在设置 quota 属性的时候，setter 方法应该采取一些措施，以保证 quota 能够与该类接口中的 fill 和 deduct 相匹配。

```

@quota.setter
def quota(self, amount):
    delta = self.max_quota - amount
    if amount == 0:
        # Quota being reset for a new period
        self.quota_consumed = 0
        self.max_quota = 0
    elif delta < 0:
        # Quota being filled for the new period
        assert self.quota_consumed == 0
        self.max_quota = amount
    else:
        # Quota being consumed during the period
        assert self.max_quota >= self.quota_consumed
        self.quota_consumed += delta

```

运行测试代码之后，可以产生与刚才那种实现方案相同的结果。

```

bucket = Bucket(60)
print('Initial', bucket)
fill(bucket, 100)
print('Filled', bucket)

if deduct(bucket, 99):
    print('Had 99 quota')
else:
    print('Not enough for 99 quota')

print('Now', bucket)

if deduct(bucket, 3):
    print('Had 3 quota')
else:
    print('Not enough for 3 quota')

print('Still', bucket)

>>>
Initial Bucket(max_quota=0, quota_consumed=0)
Filled Bucket(max_quota=100, quota_consumed=0)
Had 99 quota
Now Bucket(max_quota=100, quota_consumed=99)
Not enough for 3 quota
Still Bucket(max_quota=100, quota_consumed=99)

```

上面这种写法，最好的地方就在于：从前使用 Bucket.quota 的那些旧代码，既不需要做出修改，也不需要担心现在的 Bucket 类是如何实现的。而将来要使用 Bucket 的那些新代码，则可以直接访问 max_quota 和 quota_consumed，以执行正确的操作。

笔者特别喜欢使用 @property，因为它可以帮助开发者逐渐完善数据模型。看了上面这个 Bucket 范例之后，你可能在想：为什么一开始不把 fill 和 deduct 直接设计成 Bucket 类的实例方法，而要把它们放在 Bucket 外面呢？这样想确实有道理（参见本书第 22 条），但在实际工作中，我们经常接触到的对象，恰恰就是这种接口设计得比较糟糕，或仅仅能够充当数据容器的对象。若是代码越写越多、功能越来越膨胀，或是参与项目的人都不考虑长远的维护事宜，那就更容易出现这种局面。

在处理实际工作中的代码时，@property 固然是一项非常有效的工具，但是也不能滥用。如果你发现自己正在不停地编写各种 @property 方法，那恐怕就意味着当前这个类的代码写得确实很差。此时，应该彻底重构该类，而不应该继续修补这套糟糕的设计。

要点

- @property 可以为现有的实例属性添加新的功能。
- 可以用 @property 来逐步完善数据模型。
- 如果 @property 用得太过频繁，那就应该考虑彻底重构该类并修改相关的调用代码。

第 31 条：用描述符来改写需要复用的 @property 方法

Python 内置的 @property 修饰器（参见本书第 29 条和第 30 条），有个明显的缺点，就是不便于复用。受它修饰的这些方法，无法为同一个类中的其他属性所复用，而且，与之无关的类，也无法复用这些方法。

例如，要编写一个类，来验证学生的家庭作业成绩都处在 0 ~ 100。

```
class Homework(object):
    def __init__(self):
        self._grade = 0

    @property
    def grade(self):
        return self._grade

    @grade.setter
    def grade(self, value):
        if not (0 <= value <= 100):
            raise ValueError('Grade must be between 0 and 100')
        self._grade = value
```

由于有了 @property，所以上面这个类用起来非常简单。

```
galileo = Homework()
galileo.grade = 95
```

现在，假设要把这套验证逻辑放在考试成绩上面，而考试成绩又是由多个科目的小成绩组成的，每一科都要单独计分。

```
class Exam(object):
    def __init__(self):
        self._writing_grade = 0
        self._math_grade = 0

    @staticmethod
    def _check_grade(value):
        if not (0 <= value <= 100):
            raise ValueError('Grade must be between 0 and 100')
```

Exam 类的代码写起来非常枯燥，因为每添加一项科目，就要重复编写一次 @property 方法，而且还要把相关的验证逻辑也重做一遍。

```
@property
def writing_grade(self):
    return self._writing_grade

@writing_grade.setter
def writing_grade(self, value):
    self._check_grade(value)
    self._writing_grade = value

@property
def math_grade(self):
    return self._math_grade

@math_grade.setter
def math_grade(self, value):
    self._check_grade(value)
    self._math_grade = value
```

此外，这种写法也不够通用。如果要把这套百分制的验证逻辑放在家庭作业和考试之外的场合，那就需要反复编写例行的 @property 代码和 _check_grade 方法。

还有一种方式能够更好地实现上述功能，那就是采用 Python 的描述符 (*descriptor*) 来做。Python 会对访问操作进行一定的转译，而这种转译方式，则是由描述符协议来确定的。描述符类可以提供 __get__ 和 __set__ 方法，使得开发者无需再编写例行代码，即可复用分数验证功能。由于描述符能够把同一套逻辑运用在类中的不同属性上面，所以从这个角度来看，描述符也要比 mix-in 好一些（参见本书第 26 条）。

下面定义了名为 Exam 的新类，该类将几个 Grade 实例用作自己的类属性。Grade 类实现了描述符协议。在解释 Grade 类的工作原理之前，大家首先要明白的是：当程序访问到 Exam 实例的描述符属性时，Python 会对这种访问操作进行转译。

```
class Grade(object):
    def __get__(*args, **kwargs):
        # ...

    def __set__(*args, **kwargs):
        # ...

class Exam(object):
    # Class attributes
    math_grade = Grade()
    writing_grade = Grade()
    science_grade = Grade()
```

为属性赋值时：

```
exam = Exam()
exam.writing_grade = 40
```

Python 会将代码转译为：

```
Exam.__dict__['writing_grade'].__set__(exam, 40)
```

而获取属性时：

```
print(exam.writing_grade)
```

Python 也会将其转译为：

```
print(Exam.__dict__['writing_grade'].__get__(exam, Exam))
```

之所以会有这样的转译，关键就在于 object 类的 `__getattribute__` 方法（参见本书第 32 条）。简单来说，如果 Exam 实例没有名为 writing_grade 的属性，那么 Python 就会转向 Exam 类，并在该类中查找同名的类属性。这个类属性，如果是实现了 `__get__` 和 `__set__` 方法的对象，那么 Python 就认为此对象遵从描述符协议。

明白了这种转译方式之后，我们可以先按照下面这种写法，试着把 Homework 类里面的 @property 分数验证逻辑，改用 Grade 描述符来实现。

```
class Grade(object):
    def __init__(self):
        self._value = 0

    def __get__(self, instance, instance_type):
        return self._value
```

```
def __set__(self, instance, value):
    if not (0 <= value <= 100):
        raise ValueError('Grade must be between 0 and 100')
    self._value = value
```

不幸的是，上面这种实现方式是错误的，它会导致不符合预期的行为。在同一个 Exam 实例上面多次操作其属性时，尚且看不出错误。

```
first_exam = Exam()
first_exam.writing_grade = 82
first_exam.science_grade = 99
print('Writing', first_exam.writing_grade)
print('Science', first_exam.science_grade)

>>>
Writing 82
Science 99
```

但是，如果在多个 Exam 实例上面分别操作某一属性，那就会导致错误的结果。

```
second_exam = Exam()
second_exam.writing_grade = 75
print('Second', second_exam.writing_grade, 'is right')
print('First ', first_exam.writing_grade, 'is wrong')

>>>
Second 75 is right
First  75 is wrong
```

产生这种问题的原因是：对于 writing_grade 这个类属性来说，所有的 Exam 实例都要共享同一份 Grade 实例。而表示该属性的那个 Grade 实例，只会在程序的生命期中构建一次，也就是说：当程序定义 Exam 类的时候，它会把 Grade 实例构建好，以后创建 Exam 实例时，就不再构建 Grade 了。

为了解决此问题，我们需要把每个 Exam 实例所对应的值记录到 Grade 中。下面这段代码，用字典来保存每个实例的状态。

```
class Grade(object):
    def __init__(self):
        self._values = {}

    def __get__(self, instance, instance_type):
        if instance is None: return self
        return self._values.get(instance, 0)

    def __set__(self, instance, value):
        if not (0 <= value <= 100):
            raise ValueError('Grade must be between 0 and 100')
        self._values[instance] = value
```

上面这种实现方式很简单，而且能够正确运作，但它仍然有个问题，那就是会泄漏内存。在程序的生命期内，对于传给 `_set_` 方法的每个 `Exam` 实例来说，`_values` 字典都会保存指向该实例的一份引用。这就导致该实例的引用计数无法降为 0，从而使垃圾收集器无法将其回收。

使用 Python 内置的 `weakref` 模块，即可解决此问题。该模块提供了名为 `WeakKeyDictionary` 的特殊字典，它可以取代 `_values` 原来所用的普通字典。`WeakKeyDictionary` 的特殊之处在于：如果运行期系统发现这种字典所持有的引用，是整个程序里面指向 `Exam` 实例的最后一份引用[⊖]，那么，系统就会自动将该实例从字典的键中移除。Python 会做好相关的维护工作，以保证当程序不再使用任何 `Exam` 实例时，`_value` 字典会是空的。

```
class Grade(object):
    def __init__(self):
        self._values = WeakKeyDictionary()
    # ...
```

改用 `WeakKeyDictionary` 来实现 `Grade` 描述符，即可令程序的行为符合我们的需求。

```
class Exam(object):
    math_grade = Grade()
    writing_grade = Grade()
    science_grade = Grade()

first_exam = Exam()
first_exam.writing_grade = 82
second_exam = Exam()
second_exam.writing_grade = 75
print('First ', first_exam.writing_grade, 'is right')
print('Second', second_exam.writing_grade, 'is right')

>>>
First 82 is right
Second 75 is right
```

要点

- 如果想复用 `@property` 方法及其验证机制，那么可以自己定义描述符类。
- `WeakKeyDictionary` 可以保证描述符类不会泄漏内存。
- 通过描述符协议来实现属性的获取和设置操作时，不要纠结于 `__getattribute__` 的方法具体运作细节。

[⊖] 也可以理解为：如果运行期系统发现整个程序里面已经没有指向 `Exam` 实例的强引用了。——译者注

第32条：用`__getattr__`、`__getattribute__`和`__setattr__`实现按需生成的属性

Python语言提供了一些挂钩，使得开发者很容易就能编写出通用的代码，以便将多个系统黏合起来。例如，我们要把数据库的行（row）表示为Python对象。由于数据库有自己的一套结构（schema）[⊖]，所以在操作与行相对应的对象时，我们必须知道这个数据库的结构。然而，把Python对象与数据库相连接的这些代码，却不需要知道行的结构，所以，这部分代码应该写得通用一些。

那么，如何实现这种通用的代码呢？普通的实例属性、@property方法和描述符，都不能完成此功能，因为它们都必须预先定义好，而像这样的动态行为，则可以通过Python的`__getattr__`特殊方法来做。如果某个类定义了`__getattr__`，同时系统在该类对象的实例字典中又找不到待查询的属性，那么，系统就会调用这个方法。

```
class LazyDB(object):
    def __init__(self):
        self.exists = 5

    def __getattr__(self, name):
        value = 'Value for %s' % name
        setattr(self, name, value)
        return value
```

下面，访问`data`对象所缺失的`foo`属性。这会导致Python调用刚才定义的`__getattr__`方法，从而修改实例的`__dict__`字典：

```
data = LazyDB()
print('Before:', data.__dict__)
print('foo: ', data.foo)
print('After: ', data.__dict__)

>>>
Before: {'exists': 5}
foo: Value for foo
After: {'exists': 5, 'foo': 'Value for foo'}
```

然后，给`LazyDB`添加记录功能，把程序对`__getattr__`的调用行为记录下来。请注意，为了避免无限递归，我们需要在`LoggingLazyDB`子类里面通过`super().__getattr__()`来获取真正的属性值。

```
class LoggingLazyDB(LazyDB):
    def __getattr__(self, name):
```

[⊖] 也称架构、模式、纲要、概要、大纲。——译者注

```

print('Called __getattr__(%s)' % name)
return super().__getattr__(name)

data = LoggingLazyDB()
print('exists:', data.exists)
print('foo:    ', data.foo)
print('foo:    ', data.foo)

>>>
exists: 5
Called __getattr__(foo)
foo:    Value for foo
foo:    Value for foo

```

由于 `exists` 属性本身就在实例字典里面，所以访问它的时候，绝不会触发 `__getattr__`。而 `foo` 属性刚开始并不在实例字典中，所以初次访问的时候会触发 `__getattr__`。由于 `__getattr__` 又会调用 `setattr` 方法，并把 `foo` 放在实例字典中，所以第二次访问 `foo` 的时候，就不会再触发 `__getattr__` 了。

这种行为非常适合实现无结构数据（schemaless data，无模式数据）的按需访问[⊖]。初次执行 `__getattr__` 的时候进行一些操作，把相关的属性加载进来，以后再访问该属性时，只需从现有的结果之中获取即可。

现在假设我们还要在数据库系统中实现事务（transaction，交易）处理。用户下次访问某属性时，我们要知道数据库中对应的行是否依然有效，以及相关事务是否依然处于开启状态。这样的需求，无法通过 `__getattr__` 挂钩可靠地实现出来，因为 Python 系统会直接从实例字典的现存属性中迅速查出该属性，并返回给调用者。

为了实现此功能，我们可以使用 Python 中的另外一个挂钩，也就是 `__getattribute__`。程序每次访问对象的属性时，Python 系统都会调用这个特殊方法，即使属性字典里面已经有了该属性，也依然会触发 `__getattribute__` 方法。这样就可以在程序每次访问属性时，检查全局事务状态。下面定义的这个 `ValidatingDB` 类，会在 `__getattribute__` 方法里面记录每次调用的时间。

```

class ValidatingDB(object):
    def __init__(self):
        self.exists = 5

    def __getattribute__(self, name):
        print('Called __getattribute__(%s)' % name)
        try:

```

[⊖] 也称惰性访问。本书将酌情使用“按需”一词来翻译 lazy 和 lazily。——译者注

```

        return super().__getattribute__(name)
    except AttributeError:
        value = 'Value for %s' % name
        setattr(self, name, value)
        return value

data = ValidatingDB()
print('exists:', data.exists)
print('foo: ', data.foo)
print('foo: ', data.foo)

>>>
Called __getattribute__(exists)
exists: 5
Called __getattribute__(foo)
foo: Value for foo
Called __getattribute__(foo)
foo: Value for foo

```

按照 Python 处理缺失属性的标准流程，如果程序动态地访问了一个不应该有的属性，那么可以在 `__getattr__` 和 `__getattribute__` 里面抛出 `AttributeError` 异常。

```

class MissingPropertyDB(object):
    def __getattr__(self, name):
        if name == 'bad_name':
            raise AttributeError('%s is missing' % name)
        # ...

data = MissingPropertyDB()
data.bad_name

>>>
AttributeError: bad_name is missing

```

实现通用的功能时，我们经常会在 Python 代码里使用内置的 `hasattr` 函数来判断对象是否已经拥有了相关的属性，并用内置的 `getattr` 函数来获取属性值。这些函数会先在实例字典中搜索待查询的属性，然后再调用 `__getattr__`。

```

data = LoggingLazyDB()
print('Before: ', data.__dict__)
print('foo exists: ', hasattr(data, 'foo'))
print('After: ', data.__dict__)
print('foo exists: ', hasattr(data, 'foo'))

>>>
Before: {'exists': 5}
Called __getattr__(foo)
foo exists: True
After: {'exists': 5, 'foo': 'Value for foo'}
foo exists: True

```

在上例中，`__getattr__` 方法只调用了一次。反之，如果本类实现的是 `__getattribute__` 方法，那么每次在对象上面调用 `hasattr` 或 `getattr` 函数时，此方法都会执行。

```
data = ValidatingDB()
print('foo exists: ', hasattr(data, 'foo'))
print('foo exists: ', hasattr(data, 'foo'))

>>>
Called __getattribute__(foo)
foo exists: True
Called __getattribute__(foo)
foo exists: True
```

现在，假设当程序把值赋给 Python 对象之后，我们要以惰性的方式将其推回数据库。此功能可以用 Python 所提供的 `__setattr__` 挂钩来实现，它与前面所讲的那两个挂钩类似，可以拦截对属性的赋值操作。但是与 `__getattr__` 和 `__getattribute__` 不同的地方在于，我们不需要分成两个方法来处理。只要对实例的属性赋值，无论是直接赋值，还是通过内置的 `setattr` 函数赋值，都会触发 `__setattr__` 方法。

```
class SavingDB(object):
    def __setattr__(self, name, value):
        # Save some data to the DB log
        #
        super().__setattr__(name, value)
```

下面定义的这个 `LoggingSavingDB` 类，是 `SavingDB` 的子类，每次对它的属性赋值时，都会触发 `__setattr__` 方法。

```
class LoggingSavingDB(SavingDB):
    def __setattr__(self, name, value):
        print('Called __setattr__(%s, %r)' % (name, value))
        super().__setattr__(name, value)

data = LoggingSavingDB()
print('Before: ', data.__dict__)
data.foo = 5
print('After: ', data.__dict__)
data.foo = 7
print('Finally:', data.__dict__)

>>>
Before: {}
Called __setattr__(foo, 5)
After: {'foo': 5}
Called __setattr__(foo, 7)
Finally: {'foo': 7}
```

使用 `__getattribute__` 和 `__setattr__` 挂钩方法时要注意：每次访问对象属性时，它

们都会触发，而这可能并不是你想要的效果。例如，我们想在查询对象的属性时，从对象内部的一份字典里面，搜寻与待查属性相关联的属性值。

```
class BrokenDictionaryDB(object):
    def __init__(self, data):
        self._data = data

    def __getattribute__(self, name):
        print('Called __getattribute__(%s)' % name)
        return self._data[name]
```

上面这段代码，会在 `__getattribute__` 方法里面访问 `self._data`。试着运行一下，你就会发现：这段代码将导致 Python 程序反复递归，从而令其突破最大的栈深度并崩溃。

```
data = BrokenDictionaryDB({'foo': 3})
data.foo

>>>
Called __getattribute__(foo)
Called __getattribute__(_data)
Called __getattribute__(_data)

...
Traceback ...
RuntimeError: maximum recursion depth exceeded
```

问题在于，`__getattribute__` 会访问 `self._data`，而这就意味着需要再次调用 `__getattribute__`，然后它又会继续访问 `self._data`，并无限循环。解决办法是采用 `super().__getattribute__` 方法，从实例的属性字典里面直接获取 `_data` 属性值，以避免无限递归。

```
class DictionaryDB(object):
    def __init__(self, data):
        self._data = data

    def __getattribute__(self, name):
        data_dict = super().__getattribute__('_data')
        return data_dict[name]
```

与之类似，如果要在 `__setattr__` 方法中修改对象的属性，那么也需要通过 `super().__setattr__` 来完成。

要点

- 通过 `__getattr__` 和 `__setattr__`，我们可以用惰性的方式来加载并保存对象的属性。
- 要理解 `__getattr__` 与 `__getattribute__` 的区别：前者只会在待访问的属性缺失时触发，而后者则会在每次访问属性时触发。

- 如果要在 `__getattribute__` 和 `__setattr__` 方法中访问实例属性，那么应该直接通过 `super()`（也就是 `object` 类的同名方法）来做，以避免无限递归。

第 33 条：用元类来验证子类

元类最简单的一种用途，就是验证某个类定义得是否正确。构建复杂的类体系时，我们可能需要确保类的风格协调一致、确保某些方法得到了覆写，或是确保类属性之间具备某些严格的关系。元类提供了一种可靠的验证方式，每当开发者定义新的类时，它都会运行验证代码，以确保这个新类符合预定的规范。

开发者一般会把验证代码放在本类的 `__init__` 方法里面运行，这是由于程序在构建该类的对象时（参见本书第 28 条），会调用本类型的 `__init__` 方法。但如果改用元类来进行验证，我们还可以把验证时机再往前推一些，以便尽早发现错误。

在讲解如何用元类来验证子类之前，首先要明白如何为一般的对象定义元类。定义元类的时候，要从 `type` 中继承，而对于使用该元类的其他类来说，Python 默认会把那些类的 `class` 语句体中所含的相关内容，发送给元类的 `__new__` 方法。于是，我们就可以在系统构建出那种类型之前，先修改那个类的信息：

```
class Meta(type):
    def __new__(meta, name, bases, class_dict):
        print((meta, name, bases, class_dict))
        return type.__new__(meta, name, bases, class_dict)

class MyClass(object, metaclass=Meta):
    stuff = 123

    def foo(self):
        pass
```

元类可以获知那个类的名称、其所继承的父类，以及定义在 `class` 语句体中的全部类属性：

```
>>>
(<class '__main__.Meta'>,
 'MyClass',
 (<class 'object'>,),
 {'__module__': '__main__',
  '__qualname__': 'MyClass',
  'foo': <function MyClass.foo at 0x102c7dd08>,
  'stuff': 123})
```

Python 2 的写法稍有不同，它是通过名为 `__metaclass__` 的类属性来指定元类的。而 `Meta.__new__` 接口则一致。

```
# Python 2
class Meta(type):
    def __new__(meta, name, bases, class_dict):
        # ...
        pass

class MyClassInPython2(object):
    __metaclass__ = Meta
    # ...
```

为了在定义某个类的时候，确保该类的所有参数都有效，我们可以把相关的验证逻辑添加到 `Meta.__new__` 方法中。例如，要用类来表示任意多边形。为此，我们可以定义一种特殊的验证类，使得多边形体系中的基类^①，把这个验证类当成自己的元类。要注意的是，元类中所编写的验证逻辑，针对的是该基类的子类^②，而不是基类本身。

```
class ValidatePolygon(type):
    def __new__(meta, name, bases, class_dict):
        # Don't validate the abstract Polygon class
        if bases != (object,):
            if class_dict['sides'] < 3:
                raise ValueError('Polygons need 3+ sides')
        return type.__new__(meta, name, bases, class_dict)

class Polygon(object, metaclass=ValidatePolygon):
    sides = None # Specified by subclasses

    @classmethod
    def interior_angles(cls):
        return (cls.sides - 2) * 180

class Triangle(Polygon):
    sides = 3
```

假如我们尝试定义一种边数少于 3 的多边形子类，那么 `class` 语句体刚一结束，元类中的验证代码立刻就会拒绝这个 `class`。也就是说，如果开发者定义这样一种子类，那么程序根本就无法运行。

```
print('Before class')
class Line(Polygon):
    print('Before sides')
    sides = 1
    print('After sides')
```

^① 也就是范例代码中的 `Polygon`。——译者注

^② 也就是范例代码中的 `Triangle`（三角形）。——译者注

```

print('After class')

>>>
Before class
Before sides
After sides
Traceback ...
ValueError: Polygons need 3+ sides

```

要点

- 通过元类，我们可以在生成子类对象之前，先验证子类的定义是否合乎规范。
- Python 2 和 Python 3 指定元类的语法略有不同。
- Python 系统把子类的整个 class 语句体处理完毕之后，就会调用其元类的 `__new__` 方法。

第 34 条：用元类来注册子类

元类还有一个用途，就是在程序中自动注册类型。对于需要反向查找 (reverse lookup，简称反查) 的场合，这种注册操作是很有用的，它使我们可以在简单的标识符与对应的类之间，建立映射关系。

例如，我们想按照自己的实现方式，将 Python 对象表示为 JSON 格式的序列化数据，那么，就需要用一种手段，把指定的对象转换成 JSON 字符串。下面这段代码，定义了一个通用的基类，它可以记录程序调用本类构造器时所用的参数，并将其转换为 JSON 字典：

```

class Serializable(object):
    def __init__(self, *args):
        self.args = args

    def serialize(self):
        return json.dumps({'args': self.args})

```

有了这个类，就可以把一些简单且不可变的数据结构，轻松地转换成字符串了。例如，下面这个 Point2D 类，很容易就能转为字符串。

```

class Point2D(Serializable):
    def __init__(self, x, y):
        super().__init__(x, y)
        self.x = x
        self.y = y

```

```

def __repr__(self):
    return 'Point2D(%d, %d)' % (self.x, self.y)

point = Point2D(5, 3)
print('Object:    ', point)
print('Serialized:', point.serialize())

>>>
Object:    Point2D(5, 3)
Serialized: {"args": [5, 3]}

```

现在，我们需要对这种 JSON 字符串执行反序列化（deserialize）操作，并构建出该字符串所表示的 Point2D 对象。下面定义的这个 Deserializable 类，继承自 Serializable，它可以把 Serializable 所产生的 JSON 字符串还原为 Python 对象：

```

class Deserializable(Serializable):
    @classmethod
    def deserialize(cls, json_data):
        params = json.loads(json_data)
        return cls(*params['args'])

```

有了 Deserializable，我们就可以用一种通用的方式，对简单且不可变的对象执行序列化和反序列化操作。

```

class BetterPoint2D(Deserializable):
    # ...

point = BetterPoint2D(5, 3)
print('Before:    ', point)
data = point.serialize()
print('Serialized:', data)
after = BetterPoint2D.deserialize(data)
print('After:     ', after)

>>>
Before:    BetterPoint2D(5, 3)
Serialized: {"args": [5, 3]}
After:     BetterPoint2D(5, 3)

```

这种实现方案的缺点是，我们必须提前知道序列化的数据是什么类型（例如，是 Point2D 或 BetterPoint2D 等），然后才能对其做反序列化操作。理想的方案应该是：有很多类都可以把本类对象转换为 JSON 格式的序列化字符串，但是只需要一个公用的反序列化函数，就可以将任意的 JSON 字符串还原成相应的 Python 对象。

为此，我们可以把序列化对象的类名写到 JSON 数据里面。

```

class BetterSerializable(object):
    def __init__(self, *args):
        self.args = args

```

```

def serialize(self):
    return json.dumps({
        'class': self.__class__.__name__,
        'args': self.args,
    })
def __repr__(self):
    # ...

```

然后，把类名与该类对象构造器之间的映射关系，维护到一份字典里面。这样凡是经由 `register_class` 注册的类，就都可以拿通用的 `deserialize` 函数做反序列化操作。

```

registry = {}

def register_class(target_class):
    registry[target_class.__name__] = target_class

def deserialize(data):
    params = json.loads(data)
    name = params['class']
    target_class = registry[name]
    return target_class(*params['args'])

```

为了确保 `deserialize` 函数正常运作，我们必须用 `register_class` 把将来可能要执行反序列化操作的那些类，都注册一遍。

```

class EvenBetterPoint2D(BetterSerializable):
    def __init__(self, x, y):
        super().__init__(x, y)
        self.x = x
        self.y = y

register_class(EvenBetterPoint2D)

```

接下来，就可以对任意的 JSON 字符串执行反序列化操作了，执行操作时，我们不需要知道该字符串表示的是哪种类型的数据。

```

point = EvenBetterPoint2D(5, 3)
print('Before: ', point)
data = point.serialize()
print('Serialized:', data)
after = deserialize(data)
print('After: ', after)

>>>
Before:  EvenBetterPoint2D(5, 3)
Serialized: {"class": "EvenBetterPoint2D", "args": [5, 3]}
After:  EvenBetterPoint2D(5, 3)

```

这种方案也有缺点，那就是开发者可能会忘记调用 `register_class` 函数。

```

class Point3D(BetterSerializable):
    def __init__(self, x, y, z):
        super().__init__(x, y, z)
        self.x = x
        self.y = y
        self.z = z

# Forgot to call register_class! Whoops!

```

如果要对某份数据执行反序列化操作，而开发者又没有提前把该数据所在的类注册好，那么程序就会在运行的时候崩溃。

```

point = Point3D(5, 9, -4)
data = point.serialize()
deserialize(data)

>>>
KeyError: 'Point3D'

```

如果写完 class 语句体之后，忘记调用 register_class，那么即使从 BetterSerializable 中继承了子类，也依然无法利用 deserialize 函数对其执行反序列化操作。所以，这种写法很容易出错，而且对于编程新手尤其危险。在 Python 3 中使用类修饰器（*class decorator*）时，也会出现同样的问题。

我们应该想个办法，保证开发者在继承 BetterSerializable 的时候，程序会自动调用 register_class 函数，并将新的子类注册好。这个功能可以通过元类来实现。定义完子类的 class 语句体之后，元类可以拦截这个新的子类（参见本书第 33 条）。于是，我们就能够在子类的 class 语句体得到处理之后，立刻注册这一新的类型。

```

class Meta(type):
    def __new__(meta, name, bases, class_dict):
        cls = type.__new__(meta, name, bases, class_dict)
        register_class(cls)
        return cls

class RegisteredSerializable(BetterSerializable,
                           metaclass=Meta):
    pass

```

现在，定义完 RegisteredSerializable 的子类之后，开发者可以确信：该类肯定已经通过 register_class 函数注册好了，于是 deserialize 函数也就可以正常运作了。

```

class Vector3D(RegisteredSerializable):
    def __init__(self, x, y, z):
        super().__init__(x, y, z)
        self.x, self.y, self.z = x, y, z

```

```
v3 = Vector3D(10, -7, 3)
print('Before:    ', v3)
data = v3.serialize()
print('Serialized:', data)
print('After:     ', deserialize(data))

>>>
Before:    Vector3D(10, -7, 3)
Serialized: {"class": "Vector3D", "args": [10, -7, 3]}
After:     Vector3D(10, -7, 3)
```

只要类的继承体系正确无误，我们就可以用元类来实现类的注册，以确保每一个子类都不会遗漏。通过刚才的范例可以看出：这种方案，适用于序列化和反序列化操作。此外，它还适用于数据库的对象关系映射（object-relationship mapping，ORM）、插件系统和系统挂钩。

要点

- 在构建模块化的 Python 程序时，类的注册是一种很有用的模式。
- 开发者每次从基类中继承子类时，基类的元类都可以自动运行注册代码。
- 通过元类来实现类的注册，可以确保所有子类都不会遗漏，从而避免后续的错误。

第 35 条：用元类来注解类的属性

元类还有一个更有用处的功能，那就是可以在某个类刚定义好但是尚未使用的时候，提前修改或注解[⊖]该类的属性。这种写法通常会与描述符（descriptor）搭配起来（参见本书第 31 条），令这些属性可以更加详细地了解自己在外围类中的使用方式。

例如，要定义新的类，用来表示客户数据库里的某一行。同时，我们还希望在该类的相关属性与数据库表的每一列之间，建立对应关系。于是，用下面这个描述符类，把属性与列名联系起来。

```
class Field(object):
    def __init__(self, name):
        self.name = name
        self.internal_name = '_' + self.name
```

[⊖] annotate，也可以理解为诠释、解释。注解某个类的属性，就相当于在该属性上面附加一些信息，以阐明其意图。具体到本例来说，就是给 Customer 类中那些 Field 类型的属性都加上 name 和 internal_name 信息，使这些属性的意图更为明确。——译者注

```

def __get__(self, instance, instance_type):
    if instance is None: return self
    return getattr(instance, self.internal_name, '')

def __set__(self, instance, value):
    setattr(instance, self.internal_name, value)

```

由于列的名称已经保存到了 Field 描述符中，所以我们可以内置的 setattr 和 getattr 函数，把每个实例的所有状态都作为 protected 字段，存放在该实例的字典里面。在本书前面的例子中[⊖]，为了避免内存泄漏，我们曾经用 weakref 字典来构建描述符，而刚才的那段代码，目前看来，似乎要比 weakref 方案便捷得多。

接下来定义表示数据行的 Customer 类，定义该类的时候，我们要为每个类属性指定对应的列名。

```

class Customer(object):
    # Class attributes
    first_name = Field('first_name')
    last_name = Field('last_name')
    prefix = Field('prefix')
    suffix = Field('suffix')

```

Customer 类用起来比较简单。通过下面这段演示代码可以看出，Field 描述符能够按照预期，修改 __dict__ 实例字典：

```

foo = Customer()
print('Before:', repr(foo.first_name), foo.__dict__)
foo.first_name = 'Euclid'
print('After: ', repr(foo.first_name), foo.__dict__)

>>>
Before: ''
After: 'Euclid' {'_first_name': 'Euclid'}

```

问题在于，上面这种写法显得有些重复。在 Customer 类的 class 语句体中，我们既然要将构建好的 Field 对象赋给 Customer.first_name，那为什么还要把这个字段名（本例中是 'first_name'）再传给 Field 的构造器呢？

之所以还要把字段名传给 Field 构造器，是因为定义 Customer 类的时候，Python 会以从右向左的顺序解读赋值语句，这与从左至右的阅读顺序恰好相反。首先，Python 会以 Field('first_name') 的形式来调用 Field 构造器，然后，它把调用构造器所得的返回值，赋给 Customer.field_name。从这个顺序来看，Field 对象没有办法提前知道自己会赋给

[⊖] 参见本书第 31 条。——译者注

Customer 类里的哪一个属性。

为了消除这种重复代码，我们现在用元类来改写它。使用元类，就相当于直接在 class 语句上面放置挂钩，只要 class 语句体处理完毕，这个挂钩就会立刻触发。于是，我们可以借助元类，为 Field 描述符自动设置其 Field.name 和 Field.internal_name，而不用再像刚才那样，把列的名称手工传给 Field 构造器。

```
class Meta(type):
    def __new__(meta, name, bases, class_dict):
        for key, value in class_dict.items():
            if isinstance(value, Field):
                value.name = key
                value.internal_name = '_' + key
        cls = type.__new__(meta, name, bases, class_dict)
        return cls
```

下面定义一个基类，该基类使用刚才定义好的 Meta 作为其元类。凡是代表数据库里面某一行的类，都应该从这个基类中继承，以确保它们能够利用元类所提供的功能：

```
class DatabaseRow(object, metaclass=Meta):
    pass
```

采用元类来实现这套方案时，Field 描述符类基本上是无需修改的。唯一要调整的地方就在于：现在不再需要再给构造器传入参数了，因为刚才编写的 Meta.__new__ 方法会自动把相关的属性设置好。

```
class Field(object):
    def __init__(self):
        # These will be assigned by the metaclass.
        self.name = None
        self.internal_name = None
    # ...
```

有了元类、新的 DatabaseRow 基类以及新的 Field 描述符之后，我们在为数据行定义 DatabaseRow 子类时，就不用再像原来那样，编写重复的代码了。

```
class BetterCustomer(DatabaseRow):
    first_name = Field()
    last_name = Field()
    prefix = Field()
    suffix = Field()
```

新的 BetterCustomer 类的行为与旧的 Customer 类相同：

```
foo = BetterCustomer()
print('Before:', repr(foo.first_name), foo.__dict__)
foo.first_name = 'Euler'
print('After: ', repr(foo.first_name), foo.__dict__)
```

```
>>>  
Before: '' {}  
After: 'Euler' {'_first_name': 'Euler'}
```

要点

- 借助元类，我们可以在某个类完全定义好之前，率先修改该类的属性。
- 描述符与元类能够有效地组合起来，以便对某种行为做出修饰，或在程序运行时探查相关信息[⊖]。
- 如果把元类与描述符相结合，那就可以在不使用 weakref 模块的前提下避免内存泄漏。

[⊖] 原文为 runtime introspection，字面意思是运行时自省。——译者注



Chapter 3

第 5 章

并发及并行

并发 (concurrency) 的意思是说，计算机似乎 (*seemingly*) 是在同一时间做着很多不同的事。例如，某台电脑如果只有一个 CPU 核心，那么操作系统就会在各程序之间迅速切换，使其都有机会运行在这一个处理器上面。这种交错执行程序的方式，造成了一种假象，使我们以为这些程序可以同时运行。

并行 (parallelism) 的意思则是说，计算机确实 (*actually*) 是在同一时间做着很多不同的事[⊖]。具备多个 CPU 核心的计算机，能够同时执行多个程序。各程序中的指令，都分别运行在每个 CPU 内核上面，于是，这些程序就能够在同一时刻向前推进。

在同一个程序内部，并发是一种工具，它使程序员可以更加方便地解决特定类型的问题。在并发程序中，不同的执行路径都能够以某种方式向前推进，而这种方式，使人感觉那些路径可以在同一时间独立地运行。

并行与并发的关键区别，就在于能不能提速 (*speedup*)。某程序若是并行程序，其中有两条不同的执行路径都在平行地向前推进，则总任务的执行时间会减半，执行速度会变为普通程序的两倍。反之，假如该程序是并发程序，那么它即使可以用看似平行的方式分别执行多条路径，也依然不会使总任务的执行速度得到提升。

用 Python 语言编写并发程序，是比较容易的。通过系统调用、子进程和 C 语言扩展等机制，也可以用 Python 平行地处理一些事务。但是，要想使并发式的 Python 代码

⊖ 在不引发歧义的前提下，本书将酌情采用并行或平行来翻译这一概念。——译者注

以真正平行的方式来运行，却相当困难。所以，我们一定要明白：如何才能在这些有着微妙差别的境地中，最为恰当地利用 Python 所提供的特性。

第 36 条：用 subprocess 模块来管理子进程

Python 提供了一些非常健壮的程序库，用来运行并管理子进程，这使得 Python 语言能够很好地将命令行实用程序（command-line utility）等工具黏合起来。现有的 shell 脚本一般都会越写越复杂，在这种情况下，为了使程序代码更易读懂且更易维护，我们很自然地就会考虑用 Python 改写。

由 Python 所启动的多个子进程，是可以平行运作的，这使得我们能够在 Python 程序里充分利用电脑中的全部 CPU 核心，从而尽量提升程序的处理能力（throughput，吞吐量）。虽然 Python 解释器本身可能会受限于 CPU（参见本书第 37 条），但是开发者依然可以用 Python 顺畅地驱动并协调那些耗费 CPU 的工作任务。

在多年的发展过程中，Python 演化出了许多种运行子进程的方式，其中包括 `open`、`open2` 和 `os.exec*` 等。然而，对于当今的 Python 来说，最好用且最简单的子进程管理模块，应该是内置的 `subprocess` 模块。

用 `subprocess` 模块运行子进程，是比较简单的。下面这段代码，用 `Popen` 构造器来启动进程。然后用 `communicate` 方法读取子进程的输出信息，并等待其终止。

```
proc = subprocess.Popen(
    ['echo', 'Hello from the child!'],
    stdout=subprocess.PIPE)
out, err = proc.communicate()
print(out.decode('utf-8'))

>>>
Hello from the child!
```

子进程将会独立于父进程而运行，这里的父进程，指的是 Python 解释器。在下面这个范例程序中，可以一边定期查询子进程的状态，一边处理其他事务。

```
proc = subprocess.Popen(['sleep', '0.3'])
while proc.poll() is None:
    print('Working...')
    # Some time-consuming work here
    # ...

print('Exit status', proc.poll())

>>>
```

```
Working...
Working...
Exit status 0
```

把子进程从父进程中剥离（decouple，解耦），意味着父进程可以随意运行很多条平行的子进程。为了实现这一点，我们可以先把所有的子进程都启动起来。

```
def run_sleep(period):
    proc = subprocess.Popen(['sleep', str(period)])
    return proc

start = time()
procs = []
for _ in range(10):
    proc = run_sleep(0.1)
    procs.append(proc)
```

然后，通过 `communicate` 方法，等待这些子进程完成其 I/O 工作并终结。

```
for proc in procs:
    proc.communicate()
end = time()
print('Finished in %.3f seconds' % (end - start))

>>>
Finished in 0.117 seconds
```

 提示 假如这些子进程逐个运行，而不是平行运行，那么总的延迟时间就会达到 1 秒钟，而不会像本例这样，只用了 0.1 秒左右就运行完毕。

开发者也可以从 Python 程序向子进程输送数据，然后获取子进程的输出信息。这使得我们可以利用其他程序来平行地执行任务。例如，要用命令行式的 `openssl` 工具加密一些数据。下面这段代码，能够以相关的命令行参数及 I/O 管道，轻松地创建出完成此功能所需的子进程。

```
def run_openssl(data):
    env = os.environ.copy()
    env['password'] = b'\xe2\x40\x0d\x35\x11'
    proc = subprocess.Popen(
        ['openssl', 'enc', '-des3', '-pass', 'env:password'],
        env=env,
        stdin=subprocess.PIPE,
        stdout=subprocess.PIPE)
    proc.stdin.write(data)
    proc.stdin.flush() # Ensure the child gets input
    return proc
```

然后，把一些随机生成的字节数据，传给加密函数。请注意，在实际工作中，传入的应该是用户输入信息、文件句柄、网络套接字等内容：

```
procs = []
for _ in range(3):
    data = os.urandom(10)
    proc = run_openssl(data)
    procs.append(proc)
```

接下来，这些子进程就可以平行地运作并处理它们的输入信息了。此时，主程序可以等待这些子进程运行完毕，然后获取它们最终的输出结果：

```
for proc in procs:
    out, err = proc.communicate()
    print(out[-10:])

>>>
b'o4,G\x91\x95\xfe\xaa\xb7'
b'\x0b\x01\\xb1\xb7\xfb\xb2C\xe1b'
b'ds\xc5\xf4;j\x1f\xd0c-'
```

此外，我们还可以像 UNIX 管道那样，用平行的子进程来搭建平行的链条，所谓搭建链条（chain），就是把第一个子进程的输出，与第二个子进程的输入联系起来，并以此方式继续拼接下去。下面这个函数，可以启动一个子进程，而该进程会用命令行式的 md5 工具来处理输入流中的数据：

```
def run_md5(input_stdin):
    proc = subprocess.Popen(
        ['md5'],
        stdin=input_stdin,
        stdout=subprocess.PIPE)
    return proc
```

 提示 由于 Python 内置的 hashlib 模块本身就提供了 md5 函数，所以在实际工作中，未必要像本例这样，专门运行子进程。笔者此处之所以这样写，只是为了演示如何把某个子进程的输出信息，当成另一个子进程的输入信息。

现在，启动一套 openssl 进程，以便加密某些数据，同时启动另一套 md5 进程，以便根据加密后的输出内容来计算其哈希码（hash，杂凑码）。

```
input_procs = []
hash_procs = []
for _ in range(3):
```

```

data = os.urandom(10)
proc = run_openssl(data)
input_procs.append(proc)
hash_proc = run_md5(proc.stdout)
hash_procs.append(hash_proc)

```

启动起来之后，相关的子进程之间就会自动进行 I/O 处理。主程序只需等待这些子进程执行完毕，并打印最终的输出内容即可。

```

for proc in input_procs:
    proc.communicate()
for proc in hash_procs:
    out, err = proc.communicate()
    print(out.strip())

>>>
b'7a1822875dcf9650a5a71e5e41e77bf3'
b'd41d8cd98f00b204e9800998ecf8427e'
b'1720f581cfdc448b6273048d42621100'

```

如果你担心子进程一直不终止，或担心它的输出管道及输出管道由于某些原因发生了阻塞，那么可以给 `communicate` 方法传入 `timeout` 参数。该子进程若在指定时间段内没有给出响应，`communicate` 方法则会抛出异常，我们可以在处理异常的时候，终止出现意外的子进程。

```

proc = run_sleep(10)
try:
    proc.communicate(timeout=0.1)
except subprocess.TimeoutExpired:
    proc.terminate()
    proc.wait()

print('Exit status', proc.poll())

>>>
Exit status -15

```

不幸的是，`timeout` 参数仅在 Python 3.3 及后续版本中有效。对于早前的 Python 版本来说，我们需要使用内置的 `select` 模块来处理 `proc.stdin`、`proc.stdout` 和 `proc.stderr`，以确保 I/O 操作的超时机制能够生效。

要点

- 可以用 `subprocess` 模块运行子进程，并管理其输入流与输出流。
- Python 解释器能够平行地运行多条子进程，这使得开发者可以充分利用 CPU 的处理能力。

- 可以给 `communicate` 方法传入 `timeout` 参数，以避免子进程死锁或失去响应（*hanging*，挂起）。

第37条：可以用线程来执行阻塞式I/O，但不要用它做平行计算

标准的 Python 实现叫做 CPython。CPython 分两步来运行 Python 程序。首先，把文本形式的源代码解析并编译成字节码。然后，用一种基于栈的解释器来运行这份字节码。执行 Python 程序时，字节码解释器必须保持协调一致的状态。Python 采用 GIL (*global interpreter lock*, 全局解释器锁) 机制来确保这种协调性[⊖]。

GIL 实际上就是一把互斥锁 (mutual-exclusion lock, 又称为 mutex, 互斥体)，用以防止 CPython 受到占先式多线程切换 (preemptive multithreading) 操作的干扰。所谓占先式多线程切换，是指某个线程可以通过打断另外一个线程的方式，来获取程序控制权。假如这种干扰操作的执行时机不恰当，那就会破坏解释器的状态。而有了 GIL 之后，这些干扰操作就不会发生了，GIL 可保证每条字节码指令均能够正确地与 CPython 实现及其 C 语言扩展模块协同运作。

GIL 有一种非常显著的负面影响。用 C++ 或 Java 等语言写程序时，可以同时执行多条线程，以充分利用计算机所配备的多个 CPU 核心。Python 程序尽管也支持多线程，但由于受到 GIL 保护，所以同一时刻，只有一条线程可以向前执行。这就意味着，如果我们想利用多线程做平行计算 (parallel computation)，并希望借此为 Python 程序提速，那么结果会非常令人失望。

例如，要用 Python 执行一项计算量很大的任务。为了模拟此任务，笔者编写了一种非常原始的因数分解算法。

```
def factorize(number):
    for i in range(1, number + 1):
        if number % i == 0:
            yield i
```

如果逐个地分解许多数字，就会耗费比较长的时间。

```
numbers = [2139079, 1214759, 1516637, 1852285]
start = time()
```

[⊖] 此语境下的协调性 (coherence)，也称为一致性、连贯性。所谓确保协调性，大致意思是说：保证多个线程或进程不会同时修改同一份数据，并且要保证它们在同一时刻所看到的数据值，是相同的。——译者注

```

for number in numbers:
    list(factorize(number))
end = time()
print('Took %.3f seconds' % (end - start))

>>>
Took 1.040 seconds

```

假如使用其他语言编写程序，那我们就可以采用多线程来进行计算，因为那样做能够利用计算机所配备的全部 CPU 核心，但是对 Python 来说，却未必如此。我们不妨先试试看。下面定义的这个 Python 线程，可以执行与刚才那段范例代码相同的运算：

```

from threading import Thread

class FactorizeThread(Thread):
    def __init__(self, number):
        super().__init__()
        self.number = number

    def run(self):
        self.factors = list(factorize(self.number))

```

然后，为了实现平行计算，我们为 numbers 列表中的每个数字，都启动一条线程。

```

start = time()
threads = []
for number in numbers:
    thread = FactorizeThread(number)
    thread.start()
    threads.append(thread)

```

最后，等待全部线程执行完毕。

```

for thread in threads:
    thread.join()
end = time()
print('Took %.3f seconds' % (end - start))

>>>
Took 1.061 seconds

```

令人惊讶的是，这样做所耗费的时间，竟然比逐个执行 factorize 所耗的还要长。由于每个数字都有专门的线程负责分解，所以假如改用其他编程语言来实现，那么扣除创建线程和协调线程所需的开销之后，程序的执行速度在理论上应该接近原来的 4 倍。笔者运行范例代码所用的计算机，拥有两个 CPU 核心，所以程序执行速度应该变为原来的 2 倍。我们本来打算利用多个 CPU 核心来提升程序的速度，但却没有料到多线程的 Python 程序执行得比单线程还要慢。这样的结果说明，标准 CPython 解释器中的多线程

程序受到了 GIL 的影响。

通过其他一些方式，我们确实可以令 CPython 解释器利用 CPU 的多个内核，但是那些方式所使用的并不是标准的 Thread 类（参见本书第 41 条），而且还需要开发者编写较多的代码。明白了这些限制之后，你可能会问：那既然如此，Python 为什么还要支持多线程呢？下面有两个很好的理由。

首先，多线程使得程序看上去好像能够在同一时间做许多事情。如果要自己实现这种效果，并手工管理任务之间的切换，那就显得比较困难（参见本书第 40 条）。而借助多线程，则能够令 Python 程序自动以一种看似平行的方式，来执行这些函数。之所以能如此，是因为 CPython 在执行 Python 线程的时候，可以保证一定程度的公平。不过，由于受到 GIL 限制，所以同一时刻实际上只能有一个线程得到执行。

Python 支持多线程的第二条理由，是处理阻塞式的 I/O 操作，Python 在执行某些系统调用时，会触发此类操作。执行系统调用，是指 Python 程序请求计算机的操作系统与外界环境相交互，以满足程序的需求。读写文件、在网络间通信，以及与显示器等设备相交互等，都属于阻塞式的 I/O 操作。为了响应这种阻塞式的请求，操作系统必须花一些时间，而开发者可以借助线程，把 Python 程序与这些耗时的 I/O 操作隔离开。

例如，我们要通过串行端口（serial port，简称串口）发送信号，以便远程控制一架直升飞机。笔者采用一个速度较慢的系统调用（也就是 select）来模拟这项活动。该函数请求操作系统阻塞 0.1 秒，然后把控制权还给程序，这种效果与通过同步串口来发送信号是类似的。

```
import select

def slow_systemcall():
    select.select([], [], [], 0.1)
```

如果逐个执行上面这个系统调用，那么程序所耗的总时间，就会随着调用的次数而增加。

```
start = time()
for _ in range(5):
    slow_systemcall()
end = time()
print('Took %.3f seconds' % (end - start))

>>>
Took 0.503 seconds
```

上面这种写法的问题在于：主程序在运行 slow_systemcall 函数的时候，不能继续

向下执行，程序的主线程会卡在 `select` 系统调用那里。这种现象在实际的编程工作中是非常可怕的。因为发送信号的同时，程序必须算出直升飞机接下来要移动到的地点，否则飞机可能就会撞毁。如果要同时执行阻塞式 I/O 操作与计算操作，那就应该考虑把系统调用放到其他线程里面。

下面这段代码，把多个 `slow_systemcall` 调用分别放到多条线程中执行，这样写，使得程序既能够与多个串口通信（或是通过多个串口来控制许多架直升飞机），又能够同时在主线程里执行所需的计算。

```
start = time()
threads = []
for _ in range(5):
    thread = Thread(target=slow_systemcall)
    thread.start()
    threads.append(thread)
```

线程启动好之后，我们先算出直升机接下来要移动到的地点，然后等待执行系统调用的线程都运行完毕。

```
def compute_helicopter_location(index):
    # ...

for i in range(5):
    compute_helicopter_location(i)
for thread in threads:
    thread.join()
end = time()
print('Took %.3f seconds' % (end - start))

>>>
Took 0.102 seconds
```

与早前那种逐个执行系统调用的方案相比，这种平行方案的执行速度，接近于原来的 5 倍。这说明，尽管受制于 GIL，但是用多个 Python 线程来执行系统调用的时候，这些系统调用可以平行地执行。GIL 虽然使得 Python 代码无法并行，但它对系统调用却没有任何负面影响。由于 Python 线程在执行系统调用的时候会释放 GIL，并且一直要等到执行完毕才会重新获取它，所以 GIL 是不会影响系统调用的。

除了线程，还有其他一些方式，也能处理阻塞式的 I/O 操作，例如，内置的 `asyncio` 模块等。虽然那些方式都有着非常显著的优点，但它们要求开发者必须花些功夫，将代码重构为另外一种执行模型（参见本书第 40 条）。如果既不想大幅度地修改程序，又要平行地执行多个阻塞式 I/O 操作，那么使用多线程来实现，会比较简单一些。

要点

- 因为受到全局解释器锁（GIL）的限制，所以多条 Python 线程不能在多个 CPU 核心上面平行地执行字节码。
- 尽管受制于 GIL，但是 Python 的多线程功能依然很有用，它可以轻松地模拟出同一时刻执行多项任务的效果。
- 通过 Python 线程，我们可以平行地执行多个系统调用，这使得程序能够在执行阻塞式 I/O 操作的同时，执行一些运算操作。

第 38 条：在线程中使用 Lock 来防止数据竞争

明白了全局解释器锁（GIL，参见本书第 37 条）机制之后，许多 Python 编程新手可能会认为：自己在编写 Python 代码时，也不需要再使用互斥锁（也称为 mutex，互斥体）了。他们觉得：既然 GIL 使得 Python 线程无法平行地运行在多个 CPU 核心上面，那么它必然也会对程序中的数据结构起到锁定作用，不是吗？用列表或字典这样的数据类型做一些测试之后，我们甚至会认为上面这种说法很有道理。

但是请注意，真相并非如此。实际上，GIL 并不会保护开发者自己所编写的代码。同一时刻固然只能有一个 Python 线程得以运行，但是，当这个线程正在操作某个数据结构时，其他线程可能会打断它，也就是说，Python 解释器在执行两个连续的字节码指令时，其他线程可能会在中途突然插进来。如果开发者尝试从多个线程中同时访问某个对象，那么上述情形就会引发危险的结果。这种中断现象随时都可能发生，一旦发生，就会破坏程序的状态，从而使相关的数据结构无法保持其一致性^②。

例如，我们要编写一个程序，平行地统计许多事物。现在假设该程序要从一整套传感器网络中对光照级别进行采样，那么采集到的样本总数，就会随着程序的运行不断增多，于是，新建名为 Counter 的类，专门用来表示样本数量。

```
class Counter(object):
    def __init__(self):
        self.count = 0

    def increment(self, offset):
        self.count += offset
```

^② 原文的说法是：令数据结构的不变条件（invariant）遭到违背。——译者注

在查询传感器读数的过程中，会发生阻塞式 I/O 操作，所以，我们要给每个传感器分配它自己的工作线程（worker thread）。每采集到一次读数，工作线程就会给 Counter 对象的 value 值加 1，然后继续采集，直至完成全部的采样操作。

```
def worker(sensor_index, how_many, counter):
    for _ in range(how_many):
        # Read from the sensor
        # ...
        counter.increment(1)
```

下面定义的这个 run_threads 函数，会为每个传感器启动一条工作线程，然后等待它们完成各自的采样工作：

```
def run_threads(func, how_many, counter):
    threads = []
    for i in range(5):
        args = (i, how_many, counter)
        thread = Thread(target=func, args=args)
        threads.append(thread)
        thread.start()
    for thread in threads:
        thread.join()
```

然后，平行地执行这 5 条线程。我们觉得：这个程序的结果，应该是非常明确的。

```
how_many = 10**5
counter = Counter()
run_threads(worker, how_many, counter)
print('Counter should be %d, found %d' %
      (5 * how_many, counter.count))

>>>
Counter should be 500000, found 278328
```

但是，看到输出信息之后，我们却发现，它与正确的结果相差很远。这么简单的程序，怎么会出这么大的错呢？由于 Python 解释器在同一时刻只能运行一个线程，所以这种错误就更令人费解了。

为了保证所有的线程都能够公平地执行，Python 解释器会给每个线程分配大致相等的处理器时间。而为了达成这样的分配策略，Python 系统可能当某个线程正在执行的时候，将其暂停（suspend），然后使另外一个线程继续往下执行。问题就在于，开发者无法准确地获知 Python 系统会在何时暂停这些线程。有一些操作，看上去好像是原子操作（atomic operation）[⊖]，但 Python 系统依然有可能在线程执行到一半的时候将其暂停。

[⊖] 是指应该作为一个整体来执行的操作。比如，本例中的自增赋值操作（self.count += offset）。——译者注

于是，就发生了上面那种情况。

Counter 对象的 increment 方法看上去很简单。

```
counter.count += offset
```

但是，在对象的属性上面使用 `+=` 操作符，实际上会令 Python 于幕后执行三项独立的操作。上面那条语句，可以拆分成下面这三条语句：

```
value = getattr(counter, 'count')
result = value + offset
setattr(counter, 'count', result)
```

为了实现自增，Python 线程必须依次执行上述三个操作，而在任意两个操作之间，都有可能发生线程切换。这种交错执行的方式，可能会令线程把旧的 `value` 设置给 Counter，从而使程序的运行结果出现问题。我们用 A 和 B 这两个线程，来演示这种情况：

```
# Running in Thread A
value_a = getattr(counter, 'count')
# Context switch to Thread B
value_b = getattr(counter, 'count')
result_b = value_b + 1
setattr(counter, 'count', result_b)
# Context switch back to Thread A
result_a = value_a + 1
setattr(counter, 'count', result_a)
```

在上例中，线程 A 执行到一半的时候，线程 B 插了进来，等线程 B 执行完整个递增操作之后，线程 A 又继续执行，于是，线程 A 就把线程 B 刚才对计数器所做的递增效果，完全抹去了[⊖]。传感器采样程序所统计到的样本总数之所以会出错，正是这个原因。

为了防止诸如此类的数据竞争（data race，数据争用）行为，Python 在内置的 `threading` 模块里提供了一套健壮的工具，使得开发者可以保护自己的数据结构不受破坏。其中，最简单、最有用的工具，就是 `Lock` 类，该类相当于互斥锁（也叫做互斥体）。

我们可以用互斥锁来保护 Counter 对象，使得多个线程同时访问 `value` 值的时候，不会将该值破坏。同一时刻，只有一个线程能够获得这把锁。下面这段范例代码，用 `with` 语句来获取并释放互斥锁，这样写，能够使阅读代码的人更容易看出：线程在拥有互斥锁时，执行的究竟是哪一部分代码（详情参见本书第 43 条）。

[⊖] 原书把这种现象叫做线程 A 踩踏（stomp）线程 B。——译者注

```
class LockingCounter(object):
    def __init__(self):
        self.lock = Lock()
        self.count = 0

    def increment(self, offset):
        with self.lock:
            self.count += offset
```

接下来，还是和往常一样，启动工作线程，只不过这次改用 LockingCounter 来做计数器。

```
counter = LockingCounter()
run_threads(worker, how_many, counter)
print('Counter should be %d, found %d' %
      (5 * how_many, counter.count))

>>>
Counter should be 500000, found 500000
```

这样的运行结果，才是我们想要的答案。由此可见，Lock 对象解决了数据竞争问题。

要点

- Python 确实有全局解释器锁，但是在编写自己的程序时，依然要设法防止多个线程争用同一份数据。
- 如果在不加锁的前提下，允许多条线程修改同一个对象，那么程序的数据结构可能会遭到破坏。
- 在 Python 内置的 `threading` 模块中，有个名叫 `Lock` 的类，它用标准的方式实现了互斥锁。

第 39 条：用 Queue 来协调各线程之间的工作

如果 Python 程序同时要执行许多事务，那么开发者经常需要协调这些事务。而在各种协调方式中，较为高效的一种，则是采用函数管线[⊖]。

管线的工作原理，与制造业中的组装生产线（assembly line）相似。管线分为许多首尾相连的阶段（phase，环节），每个阶段都由一种具体的函数来负责。程序总是把待处理的新部件添加到管线的开端。每一种函数都可以在它所负责的那个阶段内，并发地

[⊖] pipeline。为了和 pipe（管道）一词相区隔，译文酌情采用管线来对译 pipeline。——译者注

处理位于该阶段的部件。等负责本阶段的那个函数，把某个部件处理好之后，该部件就会传送到管线中的下一个阶段，以此类推，直到全部阶段都经历一遍。涉及阻塞式 I/O 操作或子进程的工作任务，尤其适合用此办法处理，因为这样的任务，很容易分配到多个 Python 线程或进程之中（参见本书第 37 条）。

例如，要构建一个照片处理系统，该系统从数码相机里面持续获取照片、调整其尺寸，并将其添加到网络相册中。这样的程序，可以采用三阶段的管线来做。第一阶段获取新图片。第二阶段把下载好的图片传给缩放函数。第三阶段把缩放后的图片交给上传（upload，上载）函数。

假设我们已经用 Python 代码，把负责这三个阶段的 download、resize 和 upload 函数都写好了。那么，如何将其拼接为一条可以并发处理照片的管线呢？

首先要做的，是设计一种任务传递方式，以便在管线的不同阶段之间传递工作任务。这种方式，可以用线程安全的生产者—消费者队列[⊖]来建模（线程安全的重要性请参阅本书第 38 条，deque 类的用法请参阅本书第 46 条）。

```
class MyQueue(object):
    def __init__(self):
        self.items = deque()
        self.lock = Lock()
```

数码相机在程序中扮演生产者的角色，它会把新的图片添加到 items 列表的末端，这个 items 列表，用来存放待处理的条目。

```
def put(self, item):
    with self.lock:
        self.items.append(item)
```

图片处理管线的第一阶段，在程序中扮演消费者的角色，它会从待处理的条目清单顶部移除图片。

```
def get(self):
    with self.lock:
        return self.items.popleft()
```

我们用 Python 线程来表示管线的各个阶段，这种 Worker 线程，会从 MyQueue 这样的队列中取出待处理的任务，并针对该任务运行相关函数，然后把运行结果放到另一个 MyQueue 队列里。此外，Worker 线程还会记录查询新任务的次数，以及处理完的任务数量。

[⊖] producer-consumer queue，也可简称为生产—消费队列。——译者注

```
class Worker(Thread):
    def __init__(self, func, in_queue, out_queue):
        super().__init__()
        self.func = func
        self.in_queue = in_queue
        self.out_queue = out_queue
        self.polled_count = 0
        self.work_done = 0
```

对于 Worker 线程来说，最棘手的部分，就是如何应对输入队列为空的情况。如果上一个阶段没有及时地把相关任务处理完，那就会引发此问题。在下面的范例代码中，我们通过捕获 IndexError 异常来处理这种状况。你可以将其想象为生产线上的某个环节发生了阻滞。

```
def run(self):
    while True:
        self.polled_count += 1
        try:
            item = self.in_queue.get()
        except IndexError:
            sleep(0.01) # No work to do
        else:
            result = self.func(item)
            self.out_queue.put(result)
            self.work_done += 1
```

现在，创建相关的队列，然后根据队列与工作线程之间的对应关系，把整条管线的三个阶段拼接好。

```
download_queue = MyQueue()
resize_queue = MyQueue()
upload_queue = MyQueue()
done_queue = MyQueue()
threads = [
    Worker(download, download_queue, resize_queue),
    Worker(resize, resize_queue, upload_queue),
    Worker(upload, upload_queue, done_queue),
]
```

启动这些线程，并将大量任务添加到管线的第一个阶段。在范例代码中，笔者用简单的 object 对象，来模拟 download 函数所需下载的真实数据：

```
for thread in threads:
    thread.start()
for _ in range(1000):
    download_queue.put(object())
```

最后，等待管线将所有条目都处理完毕。完全处理好的任务，会出现在 done_

queue 队列里面。

```
while len(done_queue.items) < 1000:  
    # Do something useful while waiting  
    # ...
```

这个范例程序可以正常运行，但是线程在查询其输入队列并获取新的任务时，可能会产生一种副作用，这是值得我们注意的。`run` 方法中有一段微妙的代码，用来捕获 `IndexError` 异常的，而通过下面的输出信息，可以得知：这段代码运行了很多次。

```
processed = len(done_queue.items)  
polled = sum(t.polled_count for t in threads)  
print('Processed', processed, 'items after polling',  
      polled, 'times')  
  
>>>  
Processed 1000 items after polling 3030 times
```

在管线中，每个阶段的工作函数，其执行速度可能会有所差别，这就使得前一阶段可能会拖慢后一阶段的进度，从而令整条管线迟滞。后一个阶段会在其循环语句中，反复查询输入队列，以求获取新的任务，而前一个阶段又迟迟不能把任务交过来，于是就令后一个阶段陷入了饥饿（starve）。这样做的结果是：工作线程会白白地浪费 CPU 时间，去执行一些没有用的操作，也就是说，它们会持续地抛出并捕获 `IndexError` 异常。

上面那种实现方式有很多缺陷，刚才说的那个问题只是其中的一小部分而已。除此之外，还有三个较大的问题，也应该设法避免。首先，为了判断所有的任务是否都彻底处理完毕，我们必须再编写一个循环，持续判断 `done_queue` 队列中的任务数量。其次，`Worker` 线程的 `run` 方法，会一直执行其循环。即便到了应该退出的时候，我们也没有办法通知 `Worker` 线程停止这一循环。

第三个问题更严重：如果管线的某个阶段发生迟滞，那么随时都可能导致程序崩溃。若第一阶段的处理速度很快，而第二阶段的处理速度较慢，则连接这两个阶段的那个队列的容量就会不断增大。第二阶段始终没有办法跟上第一阶段的节奏。这种现象持续一段时间之后，程序就会因为收到大量的输入数据而耗尽内存，进而崩溃。

这些问题并不能证明管线是一种糟糕的设计方式，它们只是在提醒大家：想要自己打造一种良好的生产者—消费者队列，是非常困难的。

用 `Queue` 类来弥补自编队列的缺陷

内置的 `queue` 模块中，有个名叫 `Queue` 的类，该类能够彻底解决上面提出的那些问题。

Queue 类使得工作线程无需再频繁地查询输入队列的状态，因为它的 get 方法会持续阻塞，直到有新的数据加入。例如，我们启动一条线程，并令该线程等待 Queue 队列中的输入数据：

```
from queue import Queue
queue = Queue()

def consumer():
    print('Consumer waiting')
    queue.get()           # Runs after put() below
    print('Consumer done')

thread = Thread(target=consumer)
thread.start()
```

线程虽然已经启动了，但它却并不会立刻就执行完毕，而是会卡在 queue.get() 那里，我们必须调用 Queue 实例的 put 方法，给队列中放入一项任务，方能使 queue.get() 方法得以返回。

```
print('Producer putting')
queue.put(object())          # Runs before get() above
thread.join()
print('Producer done')

>>>
Consumer waiting
Producer putting
Consumer done
Producer done
```

为了解决管线的迟滞问题，我们用 Queue 类来限定队列中待处理的最大任务数量，使得相邻的两个阶段，可以通过该队列平滑地衔接起来。构造 Queue 时，可以指定缓冲区的容量，如果队列已满，那么后续的 put 方法就会阻塞。例如，定义一条线程，令该线程先等待片刻，然后再去消费 queue 队列中的任务：

```
queue = Queue(1)           # Buffer size of 1

def consumer():
    time.sleep(0.1)         # Wait
    queue.get()              # Runs second
    print('Consumer got 1')
    queue.get()              # Runs fourth
    print('Consumer got 2')

thread = Thread(target=consumer)
thread.start()
```

之所以要令消费线程等待片刻，是想给生产线程留出一定的时间，使其可以在 consumer() 方法调用 get 之前，率先通过 put 方法，把两个对象放到队列里面。然而，我们刚才在构建 Queue 的时候，把缓冲区的大小设成了 1，这就意味着，生产线程在放入第一个对象之后，会卡在第二个 put 方法那里，它必须等待消费线程通过 get 方法将第一个对象消费掉，然后才能放入第二个对象。

```
queue.put(object())           # Runs first
print('Producer put 1')
queue.put(object())           # Runs third
print('Producer put 2')
thread.join()
print('Producer done')

>>>
Producer put 1
Consumer got 1
Producer put 2
Consumer got 2
Producer done
```

我们还可以通过 Queue 类的 task_done 方法来追踪工作进度。有了这个方法，我们就不用再像原来那样，在管线末端的 done_queue 处进行轮询，而是可以直接判断：管线中的某个阶段，是否已将输入队列中的任务，全都处理完毕。

```
in_queue = Queue()

def consumer():
    print('Consumer waiting')
    work = in_queue.get()      # Done second
    print('Consumer working')
    # Doing work
    # ...
    print('Consumer done')
    in_queue.task_done()       # Done third

Thread(target=consumer).start()
```

现在，生产者线程的代码，既不需要在消费者线程上面调用 join 方法，也不需要轮询消费者线程。生产者只需在 Queue 实例上面调用 join，并等待 in_queue 结束即可。即便调用 in_queue.join() 时队列为空，join 也不会立刻返回，必须等消费者线程为队列中的每个条目都调用 task_done() 之后，生产者线程才可以从 join 处继续向下执行。

```
in_queue.put(object())           # Done first
print('Producer waiting')
in_queue.join()                 # Done fourth
```

```

print('Producer done')

>>>
Consumer waiting
Producer waiting
Consumer working
Consumer done
Producer done

```

我们把这些行为都封装到 Queue 的子类里面，并且令工作线程可以通过这个 ClosableQueue 类，判断出自己何时应该停止处理。这个子类定义了 close 方法，此方法会给队列中添加一个特殊的对象，用以表明该对象之后再也没有其他任务需要处理了：

```

class ClosableQueue(Queue):
    SENTINEL = object()

    def close(self):
        self.put(self.SENTINEL)

def __iter__(self):
    while True:
        item = self.get()
        try:
            if item is self.SENTINEL:
                return # Cause the thread to exit
            yield item
        finally:
            self.task_done()

```

现在，根据 ClosableQueue 类的行为，来重新定义工作线程。这一次，只要 for 循环耗尽，线程就会退出。

```

class StoppableWorker(Thread):
    def __init__(self, func, in_queue, out_queue):
        # ...

    def run(self):
        for item in self.in_queue:
            result = self.func(item)
            self.out_queue.put(result)

```

接下来，用新的工作线程类，来重新创建线程列表。

```

download_queue = ClosableQueue()
# ...
threads = [

```

```

    StoppableWorker(download, download_queue, resize_queue),
    # ...
]

```

然后，还是像从前那样，运行工作线程。把所有待处理的任务，都添加到管线第一阶段的输入队列之后，则给该队列发出终止信号。

```

for thread in threads:
    thread.start()
for _ in range(1000):
    download_queue.put(object())
download_queue.close()

```

最后，我们针对管线中相邻两个阶段连接处的那些队列，分别调用 `join` 方法。也就是说，只要当前阶段处理完毕，我们就给下一个阶段的输入队列里面放入终止信号。等到这些队列全部完工之后，所有的产品都会输出到 `done_queue` 之中。

```

download_queue.join()
resize_queue.close()
resize_queue.join()
upload_queue.close()
upload_queue.join()
print(done_queue.qsize(), 'items finished')

>>>
1000 items finished

```

要点

- 管线是一种优秀的任务处理方式，它可以把处理流程划分为若干阶段，并使用多条 Python 线程来同时执行这些任务。
- 构建并发式的管线时，要注意许多问题，其中包括：如何防止某个阶段陷入持续等待的状态之中、如何停止工作线程，以及如何防止内存膨胀等。
- `Queue` 类所提供的机制，可以彻底解决上述问题，它具备阻塞式的队列操作、能够指定缓冲区尺寸，而且还支持 `join` 方法，这使得开发者可以构建出健壮的管线。

第 40 条：考虑用协程来并发地运行多个函数

Python 程序员可以用线程来运行多个函数，使这些函数看上去好像是在同一时间得到执行的（参见本书第 37 条）。然而，线程有三个显著的缺点：

- 为了确保数据安全，我们必须使用特殊的工具来协调这些线程（参见本书第 38 条和第 39 条）。这使得多线程的代码，要比单线程的过程式代码更加难懂。这种复杂的多线程代码，会逐渐令程序变得难于扩展和维护。
- 线程需要占用大量内存，每个正在执行的线程，大约占据 8MB 内存。如果只开十几个线程，多数计算机还是可以承受的。但是，如果要在程序中运行成千上万个函数，并且想用线程来模拟出同时运行的效果，那就会出现问题。在这些函数中，有的函数与用户发送给服务器的请求相对应，有的函数与屏幕上面的像素相对应，还有的函数与仿真程序中的粒子相对应。如果每调用一次函数，就要开一个线程，那么计算机显然无法承受。
- 线程启动时的开销比较大。如果程序不停地依靠创建新线程来同时执行多个函数，并等待这些线程结束，那么使用线程所引发的开销，就会拖慢整个程序的速度。

Python 的协程 (*coroutine*) 可以避免上述问题，它使得 Python 程序看上去好像是在同时运行多个函数。协程的实现方式，实际上是对生成器（参见本书第 16 条）的一种扩展。启动生成器协程所需的开销，与调用函数的开销相仿。处于活跃状态的协程，在其耗尽之前，只会占用不到 1KB 的内存。

协程的工作原理是这样的：每当生成器函数执行到 `yield` 表达式的时候，消耗生成器的那段代码，就通过 `send` 方法给生成器回传一个值。而生成器在收到了经由 `send` 函数所传进来的这个值之后，会将其视为 `yield` 表达式的执行结果[⊖]。

```
def my_coroutine():
    while True:
        received = yield
        print('Received:', received)

it = my_coroutine()
next(it)           # Prime the coroutine
it.send('First')
it.send('Second')

>>>
Received: First
Received: Second
```

在生成器上面调用 `send` 方法之前，我们要先调用一次 `next` 函数，以便将生成器推

[⊖] 评估完当前这个 `yield` 表达式之后，生成器还会继续推进到下一个 `yield` 表达式那里，并把那个 `yield` 关键字右侧的内容，当成 `send` 方法的返回值，返回给外界。——译者注

进到第一条 yield 表达式那里。此后，我们可以把 yield 操作与 send 操作结合起来，令生成器能够根据外界所输入的数据，用一套标准的流程来产生对应的输出值。

例如，我们要编写一个生成器协程，并给它依次发送许多数值，而该协程每收到一个数值，就会给出当前所统计到的最小值。在下面这段范例代码中，第一条 yield 语句中的 yield 关键字，后面没有跟随其他内容，这条语句的意思是，把外界传进来的首个值，当成目前的最小值。此后，生成器会屡次执行 while 循环中的那条 yield 语句，以便将当前统计到的最小值告诉外界，同时等候外界传入下一个待考察的值。

```
def minimize():
    current = yield
    while True:
        value = yield current
        current = min(value, current)
```

外界的代码在消耗该生成器时，可以每次将其推进一步，而生成器在收到外界发过来的值之后，就会给出当前所统计到的最小值。

```
it = minimize()
next(it)          # Prime the generator
print(it.send(10))
print(it.send(4))
print(it.send(22))
print(it.send(-1))

>>>
10
4
4
-1
```

生成器函数似乎会一直运行下去，每次在它上面调用 send 之后，都会产生新的值。与线程类似，协程也是独立的函数，它可以消耗由外部环境所传进来的输入数据，并产生相应的输出结果。但与线程不同的是，协程会在生成器函数中的每个 yield 表达式那里暂停，等到外界再次调用 send 方法之后，它才会继续执行到下一个 yield 表达式。这就是协程的奇妙之处。

这种奇妙的机制，使得消耗生成器的那段代码，可以在每执行完协程中的一条 yield 表达式之后，就进行相应的处理。例如，那段代码可以用生成器所产生的输出值，来调用其他函数，并更新程序的数据结构。更为重要的是，它可以通过这个输出值，来推进其他的生成器函数，使得那些生成器函数也执行到它们各自的下一条 yield 表达式处。接连推进多个独立的生成器，即可模拟出 Python 线程的并发行为，令程序看上去

好像是在同时运行多个函数。

1. 生命游戏

现在用一个例子，来演示协程的协同运作效果。我们用协程实现康威（John Horton Conway）的生命游戏（The Game of Life）。游戏规则很简单。在任意尺寸的二维网格中，每个细胞^①都处在生存（alive）或空白（empty）^②状态。

```
ALIVE = '*'
EMPTY = '-'
```

时钟每走一步，生命游戏就前进一步。向前推进时，我们要点算每个细胞周边的那八个单元格，看看该细胞附近有多少个存活的细胞。本细胞需要根据相邻细胞的存活量，来判断自己在下一轮是继续存活、死亡，还是再生（regenerate）。下面从左至右列出五张 5×5 的生命游戏网格，它们演示了这些细胞在历经四个世代（generation）的变化之后，所呈现的情况。笔者稍后会解释具体的规则。

0	1	2	3	4
-----	-----	-----	-----	-----
-*---	--*--	--***-	--*--	-----
--**-	--***	-*---	-*---	--*--
--*-	--***	--***	--*--	-----
-----	-----	-----	-----	-----

我们用生成器协程来建模，把每个细胞都表示为一个协程，并令这些协程步调一致地向前推进。

为了实现这套模型，我们首先要定义一种方式，来获取相邻细胞的生存状态。笔者用名为 `count_neighbors` 的协程制作该功能，这个协程会产生 `Query` 对象，而这个 `Query` 类，则是笔者自己定义的。该类的作用，是向生成器协程提供一种手段，使得协程能够借此向外围环境查询相关的信息。

```
Query = namedtuple('Query', ('y', 'x'))
```

下面这个协程，会针对本细胞的每一个相邻细胞，来产生与之对应的 `Query` 对象。每个 `yield` 表达式的结果，要么是 `ALIVE`，要么是 `EMPTY`。这就是协程与消费代码之间的接口契约（interface contract）。其后，`count_neighbors` 生成器会根据相邻细胞的状态，来返回本细胞周边的存活细胞个数。

① cell，此处也可以理解为单元格。——译者注

② 也称为死亡（dead）状态。——译者注

```

def count_neighbors(y, x):
    n_ = yield Query(y + 1, x + 0) # North
    ne = yield Query(y + 1, x + 1) # Northeast
    # Define e_, se, s_, sw, w_, nw ...
    # ...
    neighbor_states = [n_, ne, e_, se, s_, sw, w_, nw]
    count = 0
    for state in neighbor_states:
        if state == ALIVE:
            count += 1
    return count

```

接下来，我们用虚构的数据测试这个 `count_neighbors` 协程。下面这段代码，会针对本细胞的每个相邻细胞，向生成器索要一个 `Query` 对象，并根据该对象，给出那个相邻细胞的存活状态。然后，通过 `send` 方法把状态发给协程，使 `count_neighbors` 协程可以收到上一个 `Query` 对象所对应的状态。最后，由于协程中的 `return` 语句会把生成器耗竭，所以程序届时将抛出 `StopIteration` 异常，而我们可以在处理该异常的过程中，得知本细胞周边的存活细胞总量。

```

it = count_neighbors(10, 5)
q1 = next(it)                      # Get the first query
print('First yield: ', q1)
q2 = it.send(ALIVE)                 # Send q1 state, get q2
print('Second yield:', q2)
q3 = it.send(ALIVE)                 # Send q2 state, get q3
# ...
try:
    count = it.send(EMPTY)          # Send q8 state, retrieve count
except StopIteration as e:
    print('Count: ', e.value)      # Value from return statement

>>>
First yield: Query(y=11, x=5)
Second yield: Query(y=11, x=6)
...
Count: 2

```

`count_neighbors` 协程把相邻的存活细胞数量统计出来之后，我们必须根据这个数量来更新本细胞的状态，于是，就需要用一种方式来表示状态的迁移。为此，笔者又定义了另一个名叫 `step_cell` 的协程。这个生成器会产生 `Transition` 对象，用以表示本细胞的状态迁移。这个 `Transition` 类，与 `Query` 一样，也是笔者自己定义的。

```
Transition = namedtuple('Transition', ('y', 'x', 'state'))
```

`step_cell` 协程会通过参数来接收当前细胞的网格坐标。它会针对此坐标产生 `Query`

对象，以查询本细胞的初始状态。接下来，它运行 `count_neighbors` 协程，以检视本细胞周边的其他细胞。此后，它运行 `game_logic` 函数，以判断本细胞在下一轮应该处于何种状态。最后，它生成 `Transition` 对象，把本细胞在下一轮所应有的状态，告诉外部代码。

```
def game_logic(state, neighbors):
    # ...

def step_cell(y, x):
    state = yield Query(y, x)
    neighbors = yield from count_neighbors(y, x)
    next_state = game_logic(state, neighbors)
    yield Transition(y, x, next_state)
```

请注意，`step_cell` 协程用 `yield from` 表达式来调用 `count_neighbors`。在 Python 程序中，这种表达式可以把生成器协程组合起来，使开发者能够更加方便地复用小段的功能代码，并通过简单的协程来构建复杂的协程。`count_neighbors` 协程耗竭之后，其最终的返回值（也就是 `return` 语句的返回值）会作为 `yield from` 表达式的结果，传给 `step_cell`。

现在，我们终于可以来定义游戏的逻辑函数了，康威生命游戏的规则很简单，只有下面三条^②。

```
def game_logic(state, neighbors):
    if state == ALIVE:
        if neighbors < 2:
            return EMPTY      # Die: Too few
        elif neighbors > 3:
            return EMPTY      # Die: Too many
        else:
            if neighbors == 3:
                return ALIVE     # Regenerate
    return state
```

我们现在用虚拟的数据来测试 `step_cell` 协程。

```
it = step_cell(10, 5)
q0 = next(it)          # Initial location query
print('Me: ', q0)
q1 = it.send(ALIVE)    # Send my status, get neighbor query
print('Q1: ', q1)
# ...
t1 = it.send(EMPTY)    # Send for q8, get game decision
print('Outcome: ', t1)
```

^② 用口语表述就是：①若本细胞存活，且周围的存活者不足两个，则本细胞下一轮死亡；②若本细胞存活，且周围的存活者多于 3 个，则本细胞下一轮死亡；③若本细胞死亡，且周围的存活者恰有 3 个，则本细胞下一轮再生。——译者注

```
>>>
Me:      Query(y=10, x=5)
Q1:      Query(y=11, x=5)
...
Outcome: Transition(y=10, x=5, state=' ')

```

生命游戏的目标，是要同时在网格中的每个细胞上面，运行刚才编写的那套游戏逻辑。为此，我们把 `step_cell` 协程组合到新的 `simulate` 协程之中。新的协程，会多次通过 `yield from` 表达式，来推进网格中的每一个细胞。把每个坐标点中的细胞都处理完之后，`simulate` 协程会产生 `TICK` 对象，用以表示当前这代的细胞已经全部迁移完毕。

```
TICK = object()

def simulate(height, width):
    while True:
        for y in range(height):
            for x in range(width):
                yield from step_cell(y, x)
        yield TICK
```

`simulate` 的好处在于，它和外界环境完全脱离了关联。我们目前还没有定义如何用 Python 对象来表示网格，也没有定义外部代码应该如何处理 `Query`、`Transition`、`TICK` 值并设置游戏的初始状态。尽管如此，游戏的逻辑依然是清晰的。每个细胞都可以通过运行 `step_cell` 来迁移到下一个状态。待所有细胞都迁移好之后，游戏的时钟就会向前走一步。只要 `simulate` 协程还在推进，这个过程就会一直持续下去。

协程的优势正在于此。它令开发者可以把精力放在当前所要完成的逻辑上面。协程会把程序对环境所下的指令，与发令时所用的实现代码相互解耦。这使得程序好像能够平行地运行多个协程，也使得开发者能够在不修改协程的前提下，逐渐改进发布指令时所用的实现代码。

现在，我们要在真实环境中运行 `simulate`。为此，我们需要把网格中每个细胞的状态表示出来。下面定义的这个 `Grid` 类，代表整张网格：

```
class Grid(object):
    def __init__(self, height, width):
        self.height = height
        self.width = width
        self.rows = []
        for _ in range(self.height):
            self.rows.append([EMPTY] * self.width)

    def __str__(self):
        # ...
```

在查询或设置该网格中的细胞时，调用者可以把任意值当成坐标。如果传入的坐标值越界，那就自动折回，这使得网格看上去好像是一种无限循环的空间。

```
def query(self, y, x):
    return self.rows[y % self.height][x % self.width]

def assign(self, y, x, state):
    self.rows[y % self.height][x % self.width] = state
```

最后，定义下面这个函数，它可以对 simulate 及其内部的协程所产生的各种值进行解释。该函数会把协程所产生的指令，转化为与外部环境相关的交互操作。这个函数会把网格内的所有细胞都向前推进一步，待各细胞的状态迁移操作完成之后，这些细胞就构成了一张新的网格，而 live_a_generation 函数会把这张新网格返回给调用者。

```
def live_a_generation(grid, sim):
    progeny = Grid(grid.height, grid.width)
    item = next(sim)
    while item is not TICK:
        if isinstance(item, Query):
            state = grid.query(item.y, item.x)
            item = sim.send(state)
        else: # Must be a Transition
            progeny.assign(item.y, item.x, item.state)
            item = next(sim)
    return progeny
```

为了验证这个函数的效果，我们需要创建网格并设置其初始状态。下面这段代码，会制作一种名叫 glider（滑翔机）的经典形状。

```
grid = Grid(5, 9)
grid.assign(0, 3, ALIVE)
# ...
print(grid)

>>>
---*-----
----*----
--***---
-----
```

现在，我们可以逐代推进这张网格，每推进一次，它就迁移到下一代。刚才绘制的那个滑翔机形状，会逐渐朝网格的右下方移动，而这种移动效果，正是由 game_logic 函数里那些简单的规则所确立的。

```
class ColumnPrinter(object):
    # ...
```

```

columns = ColumnPrinter()
sim = simulate(grid.height, grid.width)
for i in range(5):
    columns.append(str(grid))
    grid = live_a_generation(grid, sim)

print(columns)

>>>
      0      1      2      3      4
---*---- | ----- | ----- | ----- | -----
---*---- | --*-*---- | ---*---- | ---*---- | ---*----
---***--- | ---**--- | --*-*---- | ---**--- | ---*----
----- | ---*---- | ---**--- | ---**--- | ---***---
----- | ----- | ----- | ----- | -----

```

上面这套实现方式，其最大的优势在于：开发者能够在不修改 game_logic 函数的前提下，更新该函数外围的那些代码。我们可以在现有的 Query、Transition 和 TICK 机制的基础之上修改相关的规则，或施加影响范围更为广泛的效果。上面这套范例代码，演示了如何用协程来分离程序中的各个关注点，而关注点的分离（the separation of concerns），正是一条重要的设计原则。

2. Python 2 的协程

Python 3 的协程写起来非常优雅，这是因为它提供了方便开发者编写代码的语法糖^①，而 Python 2 则缺乏这样的机制。下面我们来讲解协程在 Python 2 中的两项限制。

首先，Python 2 没有 yield from 表达式。这就意味着，如果要在 Python 2 程序里面把两个生成器协程组合起来，那就需要在委派给另一个协程的地方^②，多写一层循环。

```

# Python 2
def delegated():
    yield 1
    yield 2

def composed():
    yield 'A'
    for value in delegated(): # yield from in Python 3
        yield value
    yield 'B'

print list(composed())

```

① syntactical sugar，也称为糖衣语法，是一种不影响程序功能，但可以提升代码可读性的便捷语法。——译者注

② delegation point，委派点，该地点相当于 Python 3 的 yield from 表达式所在的地点。——译者注

>>>

['A', 1, 2, 'B']

第二个限制是：Python 2 的生成器不支持 return 语句[⊖]。为了通过 try/except/finally 代码块正确实现出与 Python 3 相同的行为，我们需要定义自己的异常类型，并在需要返回某个值的时候，抛出该异常。

```
# Python 2
class MyReturn(Exception):
    def __init__(self, value):
        self.value = value

def delegated():
    yield 1
    raise MyReturn(2) # return 2 in Python 3
    yield 'Not reached'

def composed():
    try:
        for value in delegated():
            yield value
    except MyReturn as e:
        output = e.value
    yield output * 4

print list(composed())
>>>
[1, 8]
```

要点

- 协程提供了一种有效的方式，令程序看上去好像能够同时运行大量函数。
- 对于生成器内的 yield 表达式来说，外部代码通过 send 方法传给生成器的那个值，就是该表达式所要具备的值。
- 协程是一种强大的工具，它可以把程序的核心逻辑，与程序同外部环境交互时所用的代码相分离。
- Python 2 不支持 yield from 表达式，也不支持从生成器内通过 return 语句向外界返回某个值。

[⊖] 作者是想强调，Python 2 不支持在生成器中编写带返回值的 return 语句。该问题的详细讨论，可参阅：<https://github.com/bslatkin/effectivepython/issues/3>。——译者注

第 41 条：考虑用 concurrent.futures 来实现真正的平行计算

编写 Python 程序时，我们可能会遭遇性能问题。即使优化了代码（参见本书第 58 条），程序也依然有可能运行得很慢，从而无法满足我们对执行速度的要求。目前的计算机，其 CPU 核心数越来越多，于是，我们可以考虑通过平行计算（parallelism）来提升性能。能不能把代码的总计算量分配到多个独立的任务之中，并在多个 CPU 核心上面同时运行这些任务呢？

很遗憾，Python 的全局解释器锁（GIL）使得我们没有办法用线程实现真正的平行计算（参见本书第 37 条），因此，上面那个想法行不通。另一种常见的建议，是用 C 语言把程序中对性能要求较高的那部分代码，改写为扩展模块。由于 C 语言更贴近硬件，所以运行得比 Python 快，一旦运行速度达到要求，我们自然就不用再考虑平行计算了。C 语言扩展也可以启动并平行地运行多条原生线程（native thread），从而充分利用 CPU 的多个内核。Python 中的 C 语言扩展 API，有完备的文档可供查阅，这使得它成为解决性能问题的一个好办法。

但是，用 C 语言重写代码，是有很大代价的。短小而易读的 Python 代码，会变成冗长而费解的 C 代码。在进行这样的移植时，必须进行大量的测试，以确保移植之后的 C 程序，在功能上与原来的 Python 程序等效，而且还要确保移植过程中没有引入 bug。有的时候，这些努力是值得的。例如，Python 开发社区中的各种 C 语言扩展模块，就构成了一套庞大的生态系统，这些模块，能够提升文本解析、图像合成和矩阵运算等操作的执行速度。此外，还有如 Cython (<http://cython.org/>) 和 Numba (<http://numba.pydata.org/>) 等开源工具，可以帮助开发者把 Python 代码更加顺畅地迁移到 C 语言。

然而问题在于：只把程序中的一小部分迁移到 C，通常 是不够的。一般来说，Python 程序之所以执行得比较慢，并不是某个主要因素单独造成的，而是多个因素联合导致的。所以，要想充分利用 C 语言的硬件和线程优势，就必须把程序中的大量代码移植到 C，而这样做，又大幅增加了测试量和风险。于是，我们应该思考一下：有没有一种更好的方式，只需使用较少的 Python 代码，即可有效提升执行速度，并迅速解决复杂的计算问题。

我们可以试着通过内置的 concurrent.futures 模块，来利用另外一个名叫 multiprocessing 的内置模块，从而实现这种需求。该做法会以子进程的形式，平行地运行多个解释器，从而令 Python 程序能够利用多核心 CPU 来提升执行速度。由于子进程

与主解释器相分离，所以它们的全局解释器锁也是互相独立的。每个子进程都可以完整地利用一个 CPU 内核，而且这些子进程，都与主进程之间有着联系，通过这条联系渠道，子进程可以接收主进程发过来的指令，并把计算结果返回给主进程。

例如，我们现在要编写一个运算量很大的 Python 程序，并且要在该程序中充分利用 CPU 的多个内核。笔者采用查找两数最大公约数的算法，来演示这种编程方式。在实际工作中，这样的程序可能要执行运算量更为庞大的算法，例如，它可能要通过纳维 - 斯托克斯方程（Navier-Stokes equation）来模拟流体的运动。

```
def gcd(pair):
    a, b = pair
    low = min(a, b)
    for i in range(low, 0, -1):
        if a % i == 0 and b % i == 0:
            return i
```

由于我们没有做平行计算，所以程序会依次用 gcd 函数来求各组数字的最大公约数，这将导致程序的运行时间随着数据量的增多而变长。

```
numbers = [(1963309, 2265973), (2030677, 3814172),
           (1551645, 2229620), (2039045, 2020802)]
start = time()
results = list(map(gcd, numbers))
end = time()
print('Took %.3f seconds' % (end - start))

>>>
Took 1.170 seconds
```

用多条 Python 线程来改善上述程序，是没有效果的，因为全局解释器锁（GIL）使得 Python 无法在多个 CPU 核心上面平行地运行这些线程。下面这个程序，借助 concurrent.futures 模块来执行与刚才相同的运算，它使用 ThreadPoolExecutor 类及两个工作线程来实现（max_workers 表示工作线程的数量，此参数应该与 CPU 的核心数同）：

```
start = time()
pool = ThreadPoolExecutor(max_workers=2)
results = list(pool.map(gcd, numbers))
end = time()
print('Took %.3f seconds' % (end - start))

>>>
Took 1.199 seconds
```

线程启动的时候，是有一定开销的，与线程池进行通信，也会有开销，所以上面这个程序运行得比单线程版本还要慢。

然而神奇的是：我们只需改动一行代码，就可以提升整个程序的速度。只要把 ThreadPoolExecutor 换成 concurrent.futures 模块里的 ProcessPoolExecutor，程序的速度就上去了。

```
start = time()
pool = ProcessPoolExecutor(max_workers=2) # The one change
results = list(pool.map(gcd, numbers))
end = time()
print('Took %.3f seconds' % (end - start))

>>>
Took 0.663 seconds
```

笔者在自己的双核电脑上运行这段程序，发现它果然比前两个版本快很多。这是什么原因呢？这是因为 ProcessPoolExecutor 类会利用由 multiprocessing 模块所提供的底层机制，来逐步完成下列操作：

- 1) 把 numbers 列表中的每一项输入数据都传给 map。
- 2) 用 pickle 模块（参见本书第 44 条）对数据进行序列化，将其变成二进制形式。
- 3) 通过本地套接字（local socket），将序列化之后的数据从主解释器所在的进程，发送到子解释器所在的进程。
- 4) 接下来，在子进程中，用 pickle 对二进制数据进行反序列化操作，将其还原为 Python 对象。
- 5) 引入包含 gcd 函数的那个 Python 模块。
- 6) 各条子进程平行地针对各自的输入数据，来运行 gcd 函数。
- 7) 对运行结果进行序列化操作，将其转变为字节。
- 8) 将这些字节通过 socket 复制到主进程之中。
- 9) 主进程对这些字节执行反序列化操作，将其还原为 Python 对象。
- 10) 最后，把每条子进程所求出的计算结果合并到一份列表之中，并返回给调用者。

从编程者的角度看，上面这些步骤，似乎是比较简单的，但实际上，为了实现平行计算，multiprocessing 模块和 ProcessPoolExecutor 类在幕后做了大量的工作。如果改用其他编程语言来写，那么开发者只需用一把同步锁或一项原子操作，就可以把线程之间的通信过程协调好，而在 Python 语言中，我们却必须使用开销较高的 multiprocessing 模块。multiprocessing 的开销之所以比较大，原因就在于：主进程和子进程之间，必须进行序列化和反序列化操作，而程序中的大量开销，正是由这些操作所引发的。

对于某些较为孤立，且数据利用率较高的任务来说，这套方案非常合适。所谓孤立，是指待运行的函数不需要与程序中的其他部分共享状态。所谓利用率高，是指只需要在主进程与子进程之间传递一小部分数据，就能完成大量的运算。本例中的最大公约数算法，满足这两个条件，其他一些类似的数学算法，也可以通过这套方案实现平行计算。

如果待执行的运算不符合上述特征，那么 multiprocessing 所产生的开销，可能使我们无法通过平行化（parallelization，并行化）来提升程序速度。在那种情况下，可以求助 multiprocessing 所提供的一些高级机制，如共享内存（shared memory）、跨进程锁定（cross-process lock）、队列（queue）和代理（proxy）等。不过，那些特性用起来非常复杂。想通过那些工具令多条 Python 线程共享同一个进程的内存空间，本身已经相当困难，若还想经由 socket 把它们套用到其他进程，则会使代码变得更加难懂。

笔者建议大家多使用简单的 concurrent.futures 模块，并且尽量避开 multiprocessing 里的那些复杂特性。对于较为孤立且数据利用率较高的函数来说，我们一开始可以试着用多线程的 ThreadPoolExecutor 类来运行。稍后，可以将其迁移到 ProcessPoolExecutor 类，看看能不能提升程序的执行速度。如果试遍了各种方案，还是无法达到理想的执行速度，那我们再考虑直接使用 multiprocessing 模块中的那些复杂特性。

要点

- ❑ 把引发 CPU 性能瓶颈的那部分代码，用 C 语言扩展模块来改写，即可在尽量发挥 Python 特性的前提下，有效提升程序的执行速度。但是，这样做的工作量比较大，而且可能会引入 bug。
- ❑ multiprocessing 模块提供了一些强大的工具。对于某些类型的任务来说，开发者只需编写少量代码，即可实现平行计算。
- ❑ 若想利用强大的 multiprocessing 模块，最恰当的做法，就是通过内置的 concurrent.futures 模块及其 ProcessPoolExecutor 类来使用它。
- ❑ multiprocessing 模块所提供的那些高级功能，都特别复杂，所以开发者尽量不要直接使用它们。

内置模块

Python 采用 batteries included 思路[⊖]来设计标准库。其他许多编程语言，只自带了少量的公用程序包，如果开发者要实现某些重要功能，那就得寻觅第三方的程序包。Python 社区中确实也包含了很多那样的模块，但是 Python 语言与其他语言不同，它的许多重要模块，都是在安装时默认包含进来的，开发者只需使用这些模块，就能实现许多常见的功能。

限于篇幅，本书不可能把庞大的标准模块全都讲一遍。但这其中有一些内置的软件包，与 Python 的习惯用法结合得非常紧密，这些软件包，实际上已经成了语言规范的一部分。对于程序中那些比较微妙且容易出错的部分来说，这些关键的内置模块显得非常重要。

第 42 条：用 `functools.wraps` 定义函数修饰器

Python 用特殊的语法来表示修饰器 (*decorator*)，这些修饰器可以用来修饰函数。对于受到封装的原函数来说，修饰器能够在那个函数执行之前以及执行完毕之后，分别运行一些附加代码。这使得开发者可以在修饰器里面访问并修改原函数的参数及返回值，以实现约束语义 (enforce semantics)、调试程序、注册函数等目标。

[⊖] 这种思路可以理解为：在标准库中提供许多实用的模块。——译者注

例如，我们要打印某个函数在受到调用时所接收的参数以及该函数的返回值。对于包含一系列函数调用的递归函数来说，这样的调试功能尤其有用。下面就来定义这种修饰器：

```
def trace(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        print('%s(%r, %r) -> %r' %
              (func.__name__, args, kwargs, result))
        return result
    return wrapper
```

可以用 @ 符号把刚才那个修饰器套用到某个函数上面。

```
@trace
def fibonacci(n):
    """Return the n-th Fibonacci number"""
    if n in (0, 1):
        return n
    return (fibonacci(n - 2) + fibonacci(n - 1))
```

使用 @ 符号来修饰函数，其效果就等于先以该函数为参数，调用修饰器，然后把修饰器所返回的结果，赋给同一个作用域中与原函数同名的那个变量。

```
fibonacci = trace(fibonacci)
```

如果调用这个修饰之后的 fibonacci 函数，那么它就会在执行原函数之前及之后，分别运行 wrapper 中的附加代码，使开发者能够看到原 fibonacci 函数在递归栈的每一个层级上所具备的参数和返回值。

```
>>> fibonacci(3)
>>>
fibonacci((1,), {}) -> 1
fibonacci((0,), {}) -> 0
fibonacci((1,), {}) -> 1
fibonacci((2,), {}) -> 1
fibonacci((3,), {}) -> 2
```

上面这个修饰器虽然可以正常运作，但却会产生一种我们不希望看到的副作用。也就是说，修饰器所返回的那个值，其名称会和原来的函数不同，它现在不叫 fibonacci 了。

```
print(fibonacci)
>>>
<function trace.<locals>.wrapper at 0x107f7ed08>
```

这种效果的产生原因比较微妙。trace 函数所返回的值，是它内部定义的那个

wrapper。而我们又使用 trace 来修饰原有的 fibonacci 函数，于是，Python 就会把修饰器内部的那个 wrapper 函数，赋给当前模块中与原函数同名的 fibonacci 变量。对于调试器（参见本书第 57 条）和对象序列化器（参见本书第 44 条）等需要使用内省（introspection）机制的那些工具来说，这样的行为会干扰它们的正常运作。

例如，修饰后的 fibonacci 函数，会令内置的 help 函数失效。

```
help(fibonacci)
>>>
Help on function wrapper in module __main__:

wrapper(*args, **kwargs)
```

这个问题，可以用内置的 functools 模块中名为 wraps 的辅助函数来解决。wraps 本身也是修饰器，它可以帮助开发者编写其他修饰器。将 wraps 修饰器运用到 wrapper 函数之后，它就会把与内部函数相关的重要元数据全部复制到外围函数。

```
def trace(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # ...
    return wrapper

@trace
def fibonacci(n):
    # ...
```

现在，即使函数经过修饰，运行 help 也依然能打印出合理的结果。

```
help(fibonacci)
>>>
Help on function fibonacci in module __main__:

fibonacci(n)
    Return the n-th Fibonacci number
```

刚才我们看到：如果编写修饰器的时候，没有用 wraps 做相应的处理，那就会令 help 函数失效。除了 help 函数，修饰器还会导致其他一些难于排查的问题。为了维护函数的接口，修饰之后的函数，必须保留原函数的某些标准 Python 属性，例如，`_name_` 和 `_module_`。因此，我们需要用 wraps 来确保修饰后的函数具备正确的行为。

要点

- Python 为修饰器提供了专门的语法，它使得程序在运行的时候，能够用一个函数来修改另一个函数。

- 对于调试器这种依靠内省机制的工具，直接编写修饰器会引发奇怪的行为。
- 内置的 `functools` 模块提供了名为 `wraps` 的修饰器，开发者在定义自己的修饰器时，应该用 `wraps` 对其做一些处理，以避免一些问题。

第 43 条：考虑以 `contextlib` 和 `with` 语句来改写可复用的 `try/finally` 代码

有些代码，需要运行在特殊的情境之下，开发者可以用 Python 语言的 `with` 语句来表达这些代码的运行时机。例如，如果把互斥锁（参见本书第 38 条）放在 `with` 语句之中，那就表示：只有当程序持有该锁的时候，`with` 语句块里的那些代码，才会得到运行。

```
lock = Lock()
with lock:
    print('Lock is held')
```

由于 `Lock` 类对 `with` 语句提供了适当的支持，所以上面那种写法，可以达到与 `try/finally` 结构相仿的效果。

```
lock.acquire()
try:
    print('Lock is held')
finally:
    lock.release()
```

在上面两种写法中，使用 `with` 语句的那个版本更好一些，因为它免去了编写 `try/finally` 结构所需的重复代码。开发者可以用内置的 `contextlib` 模块来处理自己所编写的对象和函数，使它们能够支持 `with` 语句。该模块提供了名为 `contextmanager` 的修饰器。一个简单的函数，只需经过 `contextmanager` 修饰，即可用在 `with` 语句之中。这样做，要比标准的写法更加便捷。如果按标准方式来做，那就要定义新的类，并提供名为 `_enter_` 和 `_exit_` 的特殊方法。

例如，当程序运行到某一部分时，我们希望针对这部分代码，打印出更为详细的调试信息。下面定义的这个函数，可以打印两种严重程度（severity level）不同的信息：

```
def my_function():
    logging.debug('Some debug data')
    logging.error('Error log here')
    logging.debug('More debug data')
```

如果程序的默认信息级别是 `WARNING`（警告），那么运行该函数时，就只会打印

出 ERROR (错误) 级别的消息[⊖]。

```
my_function()
>>>
Error log here
```

我们可以定义一种情境管理器[⊖], 来临时提升该函数的信息级别 (log level, 日志级别)。下面这个辅助函数, 会在运行 with 块内的代码之前, 临时提升信息级别, 待 with 块执行完毕, 再恢复原有级别。

```
@contextmanager
def debug_logging(level):
    logger = logging.getLogger()
    old_level = logger.getEffectiveLevel()
    logger.setLevel(level)
    try:
        yield
    finally:
        logger.setLevel(old_level)
```

yield 表达式所在的地方, 就是 with 块中的语句所要展开执行的地方。with 块所抛出任何异常, 都会由 yield 表达式重新抛出, 这使得开发者可以在辅助函数里面捕获它(具体用法请参见本书第 40 条)。

现在重新运行这个 my_function 函数, 但是这一次, 我们把它放在 debug_logging 情境之下。大家可以看到: with 块中的那个 my_function 函数, 会把所有 DEBUG (调试) 级别的信息打印出来, 而 with 块外的那个 my_function 函数, 则不会打印 DEBUG 级别的信息。

```
with debug_logging(logging.DEBUG):
    print('Inside:')
    my_function()
print('After:')
my_function()

>>>
Inside:
Some debug data
Error log here
More debug data
After:
Error log here
```

[⊖] logging 模块只会打印出严重程度大于或等于自身级别的消息, 例如, 当自身级别为 DEBUG 时, 它可以打印 DEBUG、WARNING 和 ERROR 级别的消息, 而当自身级别为 WARNING 时, 则只会打印 WARNING 和 ERROR 级别的消息。——译者注

[⊖] context manager, 也称上下文管理器、环境管理器。——译者注

使用带有目标的 with 语句

传给 with 语句的那个情境管理器，本身也可以返回一个对象。而开发者可以通过 with 复合语句中的 as 关键字，来指定一个局部变量，Python 会把那个对象，赋给这个局部变量。这使得 with 块中的代码，可以直接与外部情境相交互。

例如，我们要向文件中写入数据，并且要确保该文件总是能正确地关闭。这个功能，可以通过 with 语句实现。把 open 传给 with 语句，并通过 as 关键字来指定一个目标变量，用以接收 open 所返回的文件句柄，等到 with 语句块退出时，该句柄会自动关闭。

```
with open('/tmp/my_output.txt', 'w') as handle:
    handle.write('This is some data!')
```

与每次手工开启并关闭文件句柄的写法相比，上面这个写法更好一些。它使得开发者能够确信：只要程序离开 with 语句块，文件就一定会关闭。此外，它也促使开发者在打开文件句柄之后，尽量少写一些代码，这通常是一种良好的编程习惯。

我们只需在情境管理器里，通过 yield 语句返回一个值，即可令自己的函数把该值提供给由 as 关键字所指定的目标变量。例如，下面定义的这个情境管理器，能够获取 Logger 实例、设置其级别，并通过 yield 语句将其提供给由 as 关键字所指定的目标：

```
@contextmanager
def log_level(level, name):
    logger = logging.getLogger(name)
    old_level = logger.getEffectiveLevel()
    logger.setLevel(level)
    try:
        yield logger
    finally:
        logger.setLevel(old_level)
```

由于 with 语句块可以把严重级别调低，所以在 as 目标变量上面调用 debug 等方法时，可以打印出 DEBUG 级别的调试信息。与之相反，若直接在 logging 模块上面调用 debug，则不会打印出任何 DEBUG 级别的消息，因为 Python 自带的那个 logger，默认会处在 WARNING 级别。

```
with log_level(logging.DEBUG, 'my-log') as logger:
    logger.debug('This is my message!')
    logging.debug('This will not print')

>>>
This is my message!
```

退出 with 语句块之后，我们在名为 'my-log' 的 Logger 上面调用 debug 方法，是打印不出消息的，因为该 Logger 的严重级别，已经恢复到默认的 WARNING 了。但是，

由于 ERROR 级别高于 WARNING 级别，所在这个 Logger 上面调用 error 方法，仍然能够打印出消息。

```
logger = logging.getLogger('my-log')
logger.debug('Debug will not print')
logger.error('Error will print')

>>>
Error will print
```

要点

- 可以用 with 语句来改写 try/finally 块中的逻辑，以便提升复用程度，并使代码更加整洁。
- 内置的 contextlib 模块提供了名叫 contextmanager 的修饰器，开发者只需用它来修饰自己的函数，即可令该函数支持 with 语句。
- 情境管理器可以通过 yield 语句向 with 语句返回一个值，此值会赋给由 as 关键字所指定的变量。该机制阐明了这个特殊情境的编写动机，并令 with 块中的语句能够直接访问这个目标变量。

第 44 条：用 copyreg 实现可靠的 pickle 操作

内置的 pickle 模块能够将 Python 对象序列化为字节流，也能把这些字节反序列化为 Python 对象。经过 pickle 处理的字节流，不应该在未受信任的程序之间传播。pickle 的设计目标是提供一种二进制渠道，使开发者能够在自己所控制的各程序之间传递 Python 对象。



由 pickle 模块所产生的序列化数据，采用的是一种不安全的格式。这种序列化数据，实际上就是一个程序，它描述了如何来构建原始的 Python 对象。这意味着：经过 pickle 处理之后的数据，如果混入了恶意信息，那么 Python 程序在对其进行反序列化时，这些恶意信息可能对程序造成损害。

与 pickle 相反，由 json 模块所产生的数据，采用的是一种安全的格式。序列化之后的 JSON 数据，只是包含简单的描述信息，这些信息用来描述由对象所构成的体系。对 JSON 数据执行反序列化操作，并不会给 Python 程序带来任何额外的风险。彼此不信任的人或程序之间，如果要进行通信，那就应该使用 JSON 这样的格式。

例如，我们要用 Python 对象表示玩家的游戏进度。下面这个 GameState 类，包含了玩家当前的级别，以及剩余的生命数。

```
class GameState(object):
    def __init__(self):
        self.level = 0
        self.lives = 4
```

程序会在游戏过程中修改 GameState 对象。

```
state = GameState()
state.level += 1 # Player beat a level
state.lives -= 1 # Player had to try again
```

玩家退出游戏时，程序可以把游戏状态保存到文件里，以便稍后恢复。使用 pickle 模块来实现这个功能，是非常简单的。下面这段代码，会把 GameState 对象直接写到一份文件里：

```
state_path = '/tmp/game_state.bin'
with open(state_path, 'wb') as f:
    pickle.dump(state, f)
```

以后，可以用 load 函数来加载这个文件，并把 GameState 对象还原回来。还原好的 GameState 对象，与没有经过序列化操作的普通对象一样，看不出太大区别。

```
with open(state_path, 'rb') as f:
    state_after = pickle.load(f)
print(state_after.__dict__)

>>>
{'lives': 3, 'level': 1}
```

在游戏功能逐渐扩展的过程中，上面那种写法会暴露出一些问题。例如，为了鼓励玩家追求高分，我们想给游戏添加计分功能。于是，给 GameState 类添加 points 字段，以表示玩家的分数。

```
class GameState(object):
    def __init__(self):
        # ...
        self.points = 0
```

针对新版的 GameState 类来使用 pickle 模块，其效果与早前相同。下面这段代码，先用 dumps 函数将对象序列化为字符串，然后又用 loads 方法将字符串还原成对象，这样做，与通过文件来保存并还原对象，是相似的：

```
state = GameState()
serialized = pickle.dumps(state)
```

```
state_after = pickle.loads(serialized)
print(state_after.__dict__)
```

```
>>>
{'lives': 4, 'level': 0, 'points': 0}
```

但是，如果有一份存档，是用旧版的 GameState 格式保存的，而现在玩家又要用这份存档来继续游戏，那会出现什么情况呢？下面这段范例代码，根据新版的 GameState 定义，来对旧版的游戏存档进行 unpickle 操作：

```
with open(state_path, 'rb') as f:
    state_after = pickle.load(f)
print(state_after.__dict__)
```

```
>>>
{'lives': 3, 'level': 1}
```

还原出来的对象，竟然没有 points 属性！由 pickle.load 所返回的对象，是个新版的 GameState 实例，可是这个新版的实例，怎么会没有 points 属性呢？这非常令人困惑^①。

```
assert isinstance(state_after, GameState)
```

这种行为，是 pickle 模块的工作机制所表现出的副作用。pickle 模块的主要意图，是帮助开发者轻松地在对象上面执行序列化操作。如果对 pickle 模块的运用超越了这个范围，那么该模块的功能就出现奇怪的问题。

要想解决这些问题，也非常简单，只需借助内置的 copyreg 模块即可。开发者可以用 copyreg 模块注册一些函数，Python 对象的序列化，将由这些函数来负责。这使得我们可以控制 pickle 操作的行为，令其变得更加可靠。

1. 为缺失的属性提供默认值

在最简单的情况下，我们可以用带默认值的构造器（参见本书第 19 条），来确保 GameState 对象在经过 unpickle 操作之后，总是能够具备所有的属性。下面重新来定义 GameState 类的构造器：

```
class GameState(object):
    def __init__(self, level=0, lives=4, points=0):
        self.level = level
        self.lives = lives
        self.points = points
```

为了使用这个构造器进行 pickle 操作，笔者定义了下面这个辅助函数，它接受

^① pickle.load 函数在还原 GameState 对象时，并不会调用 GameState 类的构造器，所以导致还原后的对象没有 points 字段。——译者注

GameState 对象，并将其转换为一个包含参数的元组，以便提供给 copyreg 模块。返回的这个元组，含有 unpickle 操作所使用的函数，以及要传给那个 unpickle 函数的参数[⊖]。

```
def pickle_game_state(game_state):
    kwargs = game_state.__dict__
    return unpickle_game_state, (kwargs,)
```

现在，定义 unpickle_game_state 这个辅助函数。该函数接受由 pickle_game_state 所传来的序列化数据及参数，并返回响应的 GameState 对象。这其实就是对构造器做了小小的封装。

```
def unpickle_game_state(kwargs):
    return GameState(**kwargs)
```

下面通过内置的 copyreg 模块来注册 pickle_game_state 函数。

```
copyreg.pickle(GameState, pickle_game_state)
```

序列化与反序列化操作，都可以像原来那样，照常进行。

```
state = GameState()
state.points += 1000
serialized = pickle.dumps(state)
state_after = pickle.loads(serialized)
print(state_after.__dict__)

>>>
{'lives': 4, 'level': 0, 'points': 1000}
```

注册好 pickle_game_state 函数之后，我们可以修改 GameState 的定义，给玩家一定数量的魔法卷轴。这次修改与早前给 GameState 类添加 points 字段时所做的修改是类似的。

```
class GameState(object):
    def __init__(self, level=0, lives=4, points=0, magic=5):
        # ...
```

但是这一次，对旧版的 GameState 对象进行反序列化操作，就可以得到正确的游戏数据了，而不会再像原来那样，丢失某些属性。由于 unpickle_game_state 会直接调用 GameState 构造器，所以反序列化之后的 GameState 对象，其属性是完备的。构造器的关键字参数，都带有默认值，如果某个参数缺失，那么对应的属性就会自动具备相应的默认值。于是，旧版的游戏存档在经过反序列化操作之后，就会拥有新的 magic 字段，

[⊖] 该元组各元素的详细含义，请参阅：https://docs.python.org/3.5/library/pickle.html#object.__reduce__。——译者注

而该字段的值，正是构造器的关键字参数所提供的默认值。

```
state_after = pickle.loads(serialized)
print(state_after.__dict__)

>>>
{'level': 0, 'points': 1000, 'magic': 5, 'lives': 4}
```

2. 用版本号来管理类

有的时候，我们要从现有的 Python 类中移除某些字段，而这种操作，会导致新类无法与旧类相兼容。刚才那种通过带有默认值的参数来进行反序列化的方案，无法应对这种情况。

例如，我们现在认为，游戏不应该限制玩家的生命数量，于是，我们就想把生命数量这一概念，从游戏中拿掉。下面定义的这个 GameState 构造器，不再包含 lives 字段：

```
class GameState(object):
    def __init__(self, level=0, points=0, magic=5):
        # ...
```

修改了构造器之后，程序就无法对旧版的游戏数据进行反序列化操作了。因为旧版游戏数据中的所有字段，都会通过 unpickle_game_state 函数，传给 GameState 构造器，即使某个字段已经从新的 GameState 类里移除，它也依然要传入 GameState 构造器。

```
pickle.loads(serialized)

>>>
TypeError: __init__() got an unexpected keyword argument
  'lives'
```

解决办法是：修改我们向 copyreg 模块注册的那个 pickle_game_state 函数，在该函数里添加一个表示版本号的参数。在对新版的 GameState 对象进行 pickle 的时候，pickle_game_state 函数会在序列化后的新版数据里面，添加值为 2 的 version 参数。

```
def pickle_game_state(game_state):
    kwargs = game_state.__dict__
    kwargs['version'] = 2
    return unpickle_game_state, (kwargs,)
```

而旧版的游戏数据里面，并没有这个 version 参数，所以，在把参数传给 GameState 构造器的时候，我们可以根据数据中是否包含 version 参数，来做相应的处理。

```
def unpickle_game_state(kwargs):
    version = kwargs.pop('version', 1)
    if version == 1:
        kwargs.pop('lives')
    return GameState(**kwargs)
```

现在，我们就可以照常对旧版的游戏存档进行反序列化操作了。

```
copyreg.pickle(GameState, pickle_game_state)
state_after = pickle.loads(serialized)
print(state_after.__dict__)

>>>
{'magic': 5, 'level': 0, 'points': 1000}
```

以后如果还要在这个类上面继续做修改，那我们依然可以用这套办法来管理不同的版本。把旧版的类适配到新版的类时，需要编写一些代码，而这些代码，都可以放到 unpickle_game_state 函数里面。

3. 固定的引入路径

使用 pickle 模块时，还会出现一个问题，那就是：当类的名称改变之后，原有的数据无法正常执行反序列化操作。在程序的生命期内，我们通常会重构自己的代码，修改某些类的名称，并把它们移动到其他模块。在做这种重构时，必须多加小心，否则会令程序无法正常使用 pickle 模块。

下面把 GameState 类的名称改为 BetterGameState，并把原来的类从程序中彻底删掉：

```
class BetterGameState(object):
    def __init__(self, level=0, points=0, magic=5):
        # ...
```

如果对旧版的 GameState 对象进行反序列化操作，那么程序就会出错，因为它找不到这个类。

```
pickle.loads(serialized)
>>>
AttributeError: Can't get attribute 'GameState' on <module
'__main__' from 'my_code.py'>
```

发生这个异常的原因在于：序列化之后的数据，把该对象所属类的引入路径，直接写在了里面。

```
print(serialized[:25])
>>>
b'\x80\x03c__main__\nGameState\nq\x00'
```

这个问题也可以通过 copyreg 模块来解决。我们可以给函数指定一个固定的标识符，令它采用这个标识符来对数据进行 unpickle 操作。这使得我们在反序列化的时候，能够把原来的数据迁移到名称不同的其他类上面。我们可以利用这种间接的机制，来处理类名变更问题。



```
copyreg.pickle(BetterGameState, pickle_game_state)
```

使用了 copyreg 之后，我们可以看到：嵌入序列化数据之中的引入路径，不再指向 BetterGameState 这个类名了，而是会指向 unpickle_game_state 函数。

```
state = BetterGameState()
serialized = pickle.dumps(state)
print(serialized[:35])

>>>
b'\x80\x03c__main__\nunpickle_game_state\nq\x00}'
```

唯一要注意的地方是，不能修改 unpickle_game_state 函数所在的模块路径。程序把这个函数写进了序列化之后的数据里面，所以，将来执行反序列化操作的时候，程序也必须能找到这个函数才行。

要点

- 内置的 pickle 模块，只适合用来在彼此信任的程序之间，对相关对象执行序列化和反序列化操作。
- 如果用法比较复杂，那么 pickle 模块的功能也许就会出问题。
- 我们可以把内置的 copyreg 模块同 pickle 结合起来使用，以便为旧数据添加缺失的属性值、进行类的版本管理，并给序列化之后的数据提供固定的引入路径。

第 45 条：应该用 datetime 模块来处理本地时间，而不是用 time 模块

协调世界时（Coordinated Universal Time, UTC）是一种标准的时间表述方式，它与时区无关。有些计算机，用某一时刻与 UNIX 时间原点[⊖]之间相差的秒数，来表示那个时刻所对应的时间，对于这些计算机来说，UTC 是一种非常好的计时方式。但是对于普通人来说，使用 UTC 来描述时间，却不太合适，因为我们通常都是根据当前所在的地点来描述时间的。我们会说“正午”（noon）或“早晨 8 点”（8am），而不会说“离 UTC 时间 15 点还差 7 个小时”（UTC 15:00 minus 7 hours）。如果我们要在程序里面处理时间，那么可能需要寻找一种方式，以便在 UTC 与当地时间之间进行转换，并以用户容易理解的说法，将其描述出来。

[⊖] UNIX epoch，可以理解为 UTC 时间 1970 年 1 月 1 日 0 时 0 分 0 秒。——译者注

Python 提供了两种时间转换方式。旧的方式，是使用内置的 time 模块，这是一种极易出错的方式。而新的方式，则是采用内置的 datetime 模块。该模块的效果非常好，它得益于 Python 开发者社区所构建的 pytz 软件包。

为了详细了解 datetime 模块的优点和 time 模块的缺点，我们必须熟悉这两个模块的用法。

1. time 模块

在内置的 time 模块中，有个名叫 localtime 的函数，它可以把 UNIX 时间戳（UNIX timestamp，也就是某个 UTC 时刻距离 UNIX 计时原点的秒数）转换为与宿主计算机的时区相符的当地时间（笔者所用电脑的时区是太平洋夏令时，Pacific Daylight Time，PDT）。

```
from time import localtime, strftime
now = 1407694710
local_tuple = localtime(now)
time_format = '%Y-%m-%d %H:%M:%S'
time_str = strftime(time_format, local_tuple)
print(time_str)

>>>
2014-08-10 11:18:30
```

程序通常还需要做反向处理，也就是说，要把用户输入的本地时间，转换为 UTC 时间。我们可以用 strftime 函数来解析包含时间信息的字符串，然后调用 mktime 函数，将本地时间转换为 UNIX 时间戳。

```
from time import mktime, strftime

time_tuple = strftime(time_str, time_format)
utc_now = mktime(time_tuple)
print(utc_now)

>>>
1407694710.0
```

如何把某个时区的当地时间转换为另一个时区的当地时间呢？例如，坐飞机从旧金山到达纽约之后，我想知道现在是旧金山的几点钟。

想通过直接操作 time、localtime 和 strftime 函数的返回值来进行时区转换，不是一个好办法。由于时区会随着当地法规而变化，所以手工管理起来太过复杂，在处理全球各个城市的航班起降问题时，显得尤其困难。

许多操作系统都提供了时区配置文件，如果时区信息发生变化，它们就会自动更新。我们可以在 Python 程序中借助 time 模块来使用这些时区信息。例如，下面这段代

码会以太平洋夏令时 (PDT) 为标准，把航班从旧金山的起飞时间解析出来^②。

```
parse_format = '%Y-%m-%d %H:%M:%S %Z'
depart_sfo = '2014-05-01 15:45:16 PDT'
time_tuple = strptime(depart_sfo, parse_format)
time_str = strftime(time_format, time_tuple)
print(time_str)

>>>
2014-05-01 15:45:16
```

我们看到，strptime 函数可以正确解析 PDT 时间，那么，它是不是也能解析电脑所支持的其他时区呢？实际上是不行的。我们用纽约所在的美国东部夏令时（Eastern Daylight Time，EDT）来实验一下，就会发现，strptime 函数抛出了异常。

```
arrival_nyc = '2014-05-01 23:33:24 EDT'
time_tuple = strptime(arrival_nyc, time_format)

>>>
ValueError: unconverted data remains: EDT
```

之所以会出现这个问题，是因为 time 模块需要依赖操作系统而运作。该模块的实际行为，取决于底层的 C 函数如何与宿主操作系统相交互。这种工作方式，使得 time 模块的功能不够稳定。它无法协调地处理各种本地时区，因此，我们应该尽量少用这个模块。如果一定要使用 time 模块，那就只应该用它在 UTC 与宿主计算机的当地时区之间进行转换。对于其他类型的转换来说，还是使用 datetime 模块比较好。

2. datetime 模块

在内置的 datetime 模块中，有个名叫 datetime 的类，它也能像刚才所讲的 time 模块那样，用来在 Python 程序中描述时间。与 time 模块类似，datetime 可以把 UTC 格式的当前时间，转换为本地时间。

下面这段代码，会把 UTC 格式的当前时间，转换为计算机所用的本地时间（笔者的计算机，采用太平洋夏令时）：

```
from datetime import datetime, timezone

now = datetime(2014, 8, 10, 18, 18, 30)
now_utc = now.replace(tzinfo=timezone.utc)
now_local = now_utc.astimezone()
print(now_local)
```

^② 为了模拟该效果，读者可能需要把代码第 2 行的 PDT 改成当前操作系统所用的时区，如 CST。——译者注

```
>>>
2014-08-10 11:18:30-07:00
```

datetime 模块还可以把本地时间轻松地转换成 UTC 格式的 UNIX 时间戳。

```
time_str = '2014-08-10 11:18:30'
now = datetime.strptime(time_str, time_format)
time_tuple = now.timetuple()
utc_now = mktime(time_tuple)
print(utc_now)
```

```
>>>
1407694710.0
```

与 time 模块不同的是，datetime 模块提供了一套机制，能够把某一种当地时间可靠地转换为另外一种当地时间。然而，在默认情况下，我们只能通过 datetime 中的 tzinfo 类及相关方法，来使用这套时区操作机制，因为它并没有提供 UTC 之外的时区定义。

所幸 Python 开发者社区提供了 pytz 模块，填补了这一空缺。该模块可以从 Python Package Index 下载 (<https://pypi.python.org/pypi/pytz/>)[⊖]。pytz 模块带有完整的数据库，其中包含了开发者可能会用到的每一种时区定义信息。

为了有效地使用 pytz 模块，我们总是应该先把当地时间转换为 UTC，然后针对 UTC 值进行 datetime 操作（例如，执行与时区偏移有关的操作），最后再把 UTC 转回当地时间。

例如，我们可以用下面这段代码，把航班到达纽约的时间，转换为 UTC 格式的 datetime 对象。某些函数调用语句，看上去似乎显得多余，但实际上，为了正确使用 pytz 模块，我们必须编写这些语句。

```
arrival_nyc = '2014-05-01 23:33:24'
nyc_dt_naive = datetime.strptime(arrival_nyc, time_format)
eastern = pytz.timezone('US/Eastern')
nyc_dt = eastern.localize(nyc_dt_naive)
utc_dt = pytz.utc.normalize(nyc_dt.astimezone(pytz.utc))
print(utc_dt)

>>>
2014-05-02 03:33:24+00:00
```

得到 UTC 格式的 datetime 之后，再把它转换成旧金山当地时间。

```
pacific = pytz.timezone('US/Pacific')
sf_dt = pacific.normalize(utc_dt.astimezone(pacific))
print(sf_dt)

>>>
2014-05-01 20:33:24-07:00
```

[⊖] 安装方式参见本书第 48 条。——译者注

我们还可以把这个时间，轻松地转换成尼泊尔（Nepal）当地之间。

```
nepal = pytz.timezone('Asia/Katmandu')
nepal_dt = nepal.normalize(utc_dt.astimezone(nepal))
print(nepal_dt)

>>>
2014-05-02 09:18:24+05:45
```

无论宿主计算机运行何种操作系统，我们都可以通过 `datetime` 模块和 `pytz` 模块，在各种环境下协调一致地完成时区转换操作。

要点

- 不要用 `time` 模块在不同时区之间进行转换。
- 如果要在不同时区之间，可靠地执行转换操作，那就应该把内置的 `datetime` 模块与开发者社区提供的 `pytz` 模块搭配起来使用。
- 开发者总是应该先把时间表示成 UTC 格式，然后对其执行各种转换操作，最后再把它转回本地时间。

第 46 条：使用内置算法与数据结构

如果 Python 程序要处理的数量比较可观，那么代码的执行速度会受到复杂算法的拖累。然而这并不能证明 Python 是一门执行速度很低的语言（参见本书第 41 条），因为这种情况很可能是算法和数据结构选择不佳而导致的。

幸运的是 Python 的标准程序库里面，内置了各种算法与数据结构，以供开发者使用。这些常见的算法与数据结构，不仅执行速度比较快，而且还可以简化编程工作。其中某些实用工具，是很难由开发者自己正确实现出来的。所以，我们应该直接使用这些 Python 自带的功能，而不要重新去实现它们，以节省时间和精力。

1. 双向队列

`collections` 模块中的 `deque` 类，是一种双向队列（double-ended queue，双端队列）。从该队列的头部或尾部插入或移除一个元素，只需消耗常数级别的时间[⊖]。这一特性，使得它非常适合用来表示先进先出（first-in-first-out，FIFO）的队列。

[⊖] 意思是说，算法所耗费的时间在某个固定范围内，不会因数据的大小及位置等因素而变化。这一概念，与算法复杂度中的 $O(1)$ 相对应。——译者注

```

fifo = deque()
fifo.append(1)      # Producer
x = fifo.popleft() # Consumer

```

内置的 list 类型，也可以像队列那样，按照一定的顺序来存放元素。从 list 尾部插入或移除元素，也仅仅需要常数级别的时间。但是，从 list 头部插入或移除元素，却会耗费线性级别^①的时间，这与 deque 的常数级时间相比，要慢得多。

2. 有序字典

标准的字典是无序的。也就是说，在拥有相同键值对的两个 dict 上面迭代，可能会出现不同的迭代顺序。标准的字典之所以会出现这种奇怪的现象，是由其快速哈希表（fast hash table）的实现方式而导致的。

```

a = {}
a['foo'] = 1
a['bar'] = 2

# Randomly populate 'b' to cause hash conflicts
while True:
    z = randint(99, 1013)
    b = {}
    for i in range(z):
        b[i] = i
    b['foo'] = 1
    b['bar'] = 2
    for i in range(z):
        del b[i]
    if str(b) != str(a):
        break

print(a)
print(b)
print('Equal?', a == b)

>>>
{'foo': 1, 'bar': 2}
{'bar': 2, 'foo': 1}
Equal? True

```

collections 模块中的 OrderedDict 类，是一种特殊的字典，它能够按照键的插入顺序，来保留键值对在字典中的次序。在 OrderedDict 上面根据键来迭代，其行为是确定的。这种确定的行为，可以极大地简化测试与调试工作。

^① 意思是说，算法所耗费的时间与数据的大小成正比。这一概念，与算法复杂度中的 $O(n)$ 相对应。
——译者注

```

a = OrderedDict()
a['foo'] = 1
a['bar'] = 2
b = OrderedDict()
b['foo'] = 'red'
b['bar'] = 'blue'

for value1, value2 in zip(a.values(), b.values()):
    print(value1, value2)

>>>
1 red
2 blue

```

3. 带有默认值的字典

字典可用来保存并记录一些统计数据。但是，由于字典里面未必有我们要查询的那个键，所以在用字典保存计数器的时候，就必须要用稍微麻烦一些的方式，才能够实现这种简单的功能。

```

stats = {}
key = 'my_counter'
if key not in stats:
    stats[key] = 0
stats[key] += 1

```

我们可以用 `collections` 模块中的 `defaultdict` 类来简化上述代码。如果字典里面没有待访问的键，那么它就会把某个默认值与这个键自动关联起来。于是，我们只需提供返回默认值的函数即可，字典会用该函数为每一个默认的键指定默认值。在本例中，我们使用内置的 `int` 函数来创建字典，这使得该字典能够以 0 为默认值（参见本书第 23 条）。现在，计数器的实现代码就变得非常简单了。

```

stats = defaultdict(int)
stats['my_counter'] += 1

```

4. 堆队列（优先级队列）

堆（heap）是一种数据结构，很适合用来实现优先级队列。`heapq` 模块提供了 `heappush`、`heappop` 和 `nsmallest` 等一些函数，能够在标准的 `list` 类型之中创建堆结构。

各种优先级的元素，都可以按任意顺序插入堆中。

```

a = []
heappush(a, 5)
heappush(a, 3)
heappush(a, 7)
heappush(a, 4)

```

这些元素总是会按照优先级从高到低的顺序，从堆中弹出（数值较小的元素，优先级较高）。

```
print(heapq.heappop(a), heapq.heappop(a), heapq.heappop(a), heapq.heappop(a))
>>>
3 4 5 7
```

用 heapq 把这样的 list 制作好之后，我们可以在其他场合使用它。只要访问堆中下标为 0 的那个元素，就总是能够查出最小值。

```
a = []
heappush(a, 5)
heappush(a, 3)
heappush(a, 7)
heappush(a, 4)
assert a[0] == nsmallest(1, a)[0] == 3
```

在这种 list 上面调用 sort 方法之后，该 list 依然能够保持堆的结构[⊖]。

```
print('Before:', a)
a.sort()
print('After: ', a)

>>>
Before: [3, 4, 7, 5]
After: [3, 4, 5, 7]
```

这些 heapq 操作所耗费的时间，与列表长度的对数成正比。如果在普通的 Python 列表上面执行相关操作，那么将会耗费线性级别的时间。

5. 二分查找

在 list 上面使用 index 方法来搜索某个元素，所耗的时间会与列表的长度呈线性比例。

```
x = list(range(10**6))
i = x.index(991234)
```

bisect 模块中的 bisect_left 等函数，提供了高效的二分折半搜索算法，能够在一系列排好顺序的元素之中搜寻某个值。由 bisect_left 函数所返回的索引，表示待搜寻的值在序列中的插入点[⊖]。

```
i = bisect_left(x, 991234)
```

二分搜索算法的复杂度，是对数级别的。这就意味着，用 bisect 来搜索包含一百万

[⊖] 堆中每个父节点的值，总是小于或等于其子节点的值。下标为 k 的父节点，其左右两个子节点的下标分别是 $2k+1$ 与 $2k+2$ 。——译者注

[⊖] 将该值插在此处，能够使序列依然保持有序。——译者注

个元素的列表，与用 `index` 来搜索包含 14 个元素的列表，所耗的时间差不多。由此可见，这种对数级别的算法，要比线性级别的算法快很多。

6. 与迭代器有关的工具

内置的 `itertools` 模块中，包含大量的函数，可以用来组合并操控迭代器（相关的知识，请参见本书第 16 条和第 17 条。虽然这些工具未必都能够直接在 Python 2 中使用，但是模块的文档里面提供了一些简单的教程[⊖]，开发者可以根据这些教程，在 Python 2 中轻松地构建与之相仿的功能。请在交互式的 Python 界面中输入 `help(itertools)`，以查看详细的信息。

`itertools` 函数分为三大类：

□ 能够把迭代器连接起来的函数：

- `chain`: 将多个迭代器按顺序连成一个迭代器。
- `cycle`: 无限地重复某个迭代器中的各个元素。
- `tee`: 把一个迭代器拆分成多个平行的迭代器。
- `zip_longest`: 与内置的 `zip` 函数相似，但是它可以应对长度不同的迭代器。

□ 能够从迭代器中过滤元素的函数：

- `islice`: 在不进行复制的前提下，根据索引值来切割迭代器。
- `takewhile`: 在判定函数（predicate function，谓词函数）为 `True` 的时候，从迭代器中逐个返回元素。
- `dropwhile`: 从判定函数初次为 `False` 的地方开始，逐个返回迭代器中的元素。
- `filterfalse`: 从迭代器中逐个返回能令判定函数为 `False` 的所有元素。其效果与内置的 `filter` 函数相反。

□ 能够把迭代器中的元素组合起来的函数：

- `product`: 根据迭代器中的元素计算笛卡儿积 (Cartesian product)，并将其返回。可以用 `product` 来改写深度嵌套的列表推导操作。
- `permutations`: 用迭代器中的元素构建长度为 N 的各种有序排列，并将所有排列形式返回给调用者。
- `combination`: 用迭代器中的元素构建长度为 N 的各种无序组合，并将所有组合形式返回给调用者。

⊖ 这里所说的教程 (recipe)，是一些演示代码或解说性质的等价代码，开发者可以据此编写功能相仿的替代函数。——译者注

除了笔者上面提到的这些，`itertools` 模块里面还有其他一些函数及教程。如果你发现自己要编写一段非常麻烦的迭代程序，那就应该先花些时间来阅读 `itertools` 的文档，看看里面有没有现成的工具可供使用。

要点

- 我们应该用 Python 内置的模块来描述各种算法和数据结构。
- 开发者不应该自己去重新实现那些功能，因为我们很难把它写好。

第 47 条：在重视精确度の場合，应该使用 `decimal`

Python 语言很适合用来编写与数值型数据打交道的代码。Python 的整数类型，可以表达任意长度的值，其双精度浮点数类型，也遵循 IEEE 754 标准。此外，Python 还提供了标准的复数类型，用来表示虚数值。然而这些数值类型，并不能覆盖每一种情况。

例如，要根据通话时长和费率，来计算用户拨打国际长途电话所应支付的费用。假如用户打了 3 分 42 秒，从美国打往南极洲的电话，每分钟 1.45 美元，那么，这次通话的费用是多少呢？

我们可能认为，只要使用浮点数，就能算出合理的结果。

```
rate = 1.45
seconds = 3*60 + 42
cost = rate * seconds / 60
print(cost)

>>>
5.364999999999999
```

但是，把计算结果向分位取整^①之后，却发现，`round` 函数把分位右侧的那些数字完全舍去了。实际上，不足 1 分钱的部分，是应该按 1 分钱收取的，我们希望 `round` 函数把该值上调为 5.37，而不是下调为 5.36。

```
print(round(cost, 2))

>>>
5.36
```

假设我们还要对那种通话时长很短，而且费率很低的电话呼叫进行计费。下面这段代码，按照每分钟 0.05 美元的费率，来计算长度为 5 秒的通话费用：

^① 也就是精确到小数点之后第 2 位。——译者注

```
rate = 0.05
seconds = 5
cost = rate * seconds / 60
print(cost)

>>>
0.004166666666666667
```

由于计算出来的数值很小，所以 round 函数会把它下调为 0，而这当然不是我们想要的结果。

```
print(round(cost, 2))  
>>>  
0.0
```

内置的 decimal 模块中，有个 Decimal 类，可以解决上面那些问题。该类默认提供 28 个小数位，以进行定点（fixed point）数学运算。如果有需要，还可以把精确度调得更高一些。Decimal 类解决了 IEEE 754 浮点数所产生的精度问题，而且开发者还可以更加精准地控制该类的舍入行为。

例如，我们可以把刚才计算美国与南极洲长途电话费的那个程序，用 Decimal 类改写。改写之后，程序会算出精确的结果，而不会像原来那样，只求出近似值。

```
rate = Decimal('1.45')
seconds = Decimal('222') # 3*60 + 42
cost = rate * seconds / Decimal('60')
print(cost)

>>>
5.365
```

Decimal 类提供了一个内置的函数，它可以按照开发者所要求的精度及舍入方式，来准确地调整数值。

```
rounded = cost.quantize(Decimal('0.01'), rounding=ROUND_UP)
print(rounded)

>>>
5.37
```

这个 quantize 方法，也能对那种时长很短、费用很低的电话，正确地进行计费。我们用 Decimal 类来改写之前的那段代码。改写之后，计算出来的电话费用，还是不足 1 分钱：

但是，我们可以在调用 `quantize` 方法时，指定合理的舍入方式，从而确保该方法能够把不足 1 分钱的部分，上调为 1 分钱。

```
rounded = cost.quantize(Decimal('0.01'), rounding=ROUND_UP)
print(rounded)

>>>
0.01
```

虽然 `Decimal` 类很适合执行定点数的运算，但它在精确度方面仍有局限，例如， $1/3$ 这个数，就只能用近似值来表示。如果要用精度不受限制的方式来表达有理数，那么可以考虑使用 `Fraction` 类，该类包含在内置的 `fractions` 模块里。

要点

- 对于编程中可能用到的每一种数值，我们都可以拿对应的 Python 内置类型，或内置模块中的类表示。
- `Decimal` 类非常适合用在那种对精度要求很高，且对舍入行为要求很严的场合，例如，涉及货币计算的场合。

第 48 条：学会安装由 Python 开发者社区所构建的模块

Python 有个中央仓库 (<https://pypi.python.org>)，里面存放着各种模块，以供程序开发者安装并使用。这些模块都是由 Python 社区构建并维护的，模块作者与大家一样，都是 Python 程序员。如果你碰到了一个自己不太熟悉的编程难题，那就应该先去 Python Package Index (简称 PyPI) 看看。PyPI 里面的代码，很有可能会帮你尽快找到答案。

为了安装由 Package Index 所提供的模块，我们需要使用名为 pip 的命令行工具。在 Python 3.4 及后续版本中，pip 是默认安装好的，开发者也可以通过 `python -m pip` 命令来使用该工具。对于较早的 Python 版本来说，我们可以在 Python Packaging 的网站 (<https://packaging.python.org>) 上面找到 pip 工具的安装方式[⊖]。

有了 pip 工具，我们就可以非常方便地安装新模块了。例如，本章前面曾经用到了 `pytz` 模块（参见本书第 45 条），现在来安装这个模块：

[⊖] 也可以参考 <https://pip.pypa.io/en/latest/installing.html>。——译者注

```
$ pip3 install pytz
Downloading/unpacking pytz
  Downloading pytz-2014.4.tar.bz2 (159kB): 159kB downloaded
    Running setup.py (...) egg_info for package pytz

Installing collected packages: pytz
  Running setup.py install for pytz

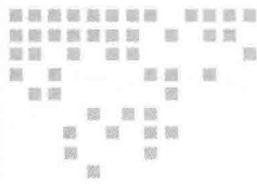
Successfully installed pytz
Cleaning up...
```

在上面这个范例中，笔者用 pip3 命令安装了 Python 3 版本的 pytz 软件包。如果使用不带 3 的 pip 命令，那就会安装 Python 2 版本的 pytz 软件包。大部分流行的软件包，都同时提供了 Python 2 版本和 Python 3 版本（参见本书第 1 条）。pip 工具也可以同 pyvenv 工具结合起来，以确定用户在使用你的项目时，必须安装哪些软件包（参见本书第 53 条）。

PyPI 中的每个模块，都有软件许可协议。大部分软件包，尤其是流行的软件包，都采用自由的或开源的协议（详情参见 <http://opensource.org>）。大多数情况下，这些协议都允许开发者在自己的程序中，包含本模块的一份拷贝（如有疑问，请咨询法律人士）。

要点

- Python Package Index (PyPI) 包含了许多常用的软件包，它们都是由 Python 开发者社区来构建并维护的。
- pip 是个命令行工具，可以从 PyPI 中安装软件包。
- Python 3.4 及后续版本，默认装有 pip，使用早前 Python 版本的开发者，必须自行安装 pip。
- 大部分 PyPI 模块，都是自由软件或开源软件。



协作开发

Python 语言的某些特性，能够帮助开发者构建接口清晰、边界明确的优秀 API。Python 开发者之间也形成了一套固定的做法，能够在程序的演化过程中尽量保持代码的可维护性。此外，Python 还自带了一些标准的工具，使得分布在不同环境之中的大型开发团队，可以借助这些工具来进行协作。

如果要和别人一起研发 Python 程序，那就必须仔细考虑代码的写法。即使程序只由你一个人来开发，你也依然会用到某些标准程序库或开源软件包，而那些模块中的代码，可能是由其他人编写的。所以，我们必须了解这套多人协作机制，以便更好地与其他 Python 程序员合作。

第 49 条：为每个函数、类和模块编写文档字符串

由于 Python 是一门动态语言，所以文档显得极其重要。Python 对文档提供了内置的支持，使得开发者可以把文档与代码块关联起来。与其他许多编程语言不同，Python 程序在运行的时候，能够直接访问源代码中的文档信息。

例如，在为函数编写了 `def` 语句之后，我们可以紧接着提供 `docstring`，以便将一段开发文档与该函数关联起来：

```
def palindrome(word):
    """Return True if the given word is a palindrome."""
```

```
return word == word[::-1]
```

在 Python 程序中，我们可以通过名为 `__doc__` 的特殊属性，来访问该函数的文档。

```
print(repr(palindrome.__doc__))
>>>
'Return True if the given word is a palindrome.'
```

函数、类和模块，都可以与文档字符串相关联。系统会在编译和运行 Python 程序的过程中，维护这种联系。因为 Python 代码支持文档字符串和 `__doc__` 属性，所以产生了下面三个好处：

- 由于能够访问代码中的文档，所以交互式开发变得更加方便了。我们可以用内置的 `help` 函数来查看函数、类和模块的文档，也可以更加方便地采用互动式 Python 解释器（也就是 Python shell、Python “壳”）及 IPython Notebook (<http://ipython.org>) 之类的工具来开发算法、测试 API 并编写代码片段。
- 这种标准的文档定义方式，使得开发者很容易就能构建出一些工具，把纯文本转换成 HTML 等更为友好的格式。例如，Python 开发者社区就构建了 Sphinx (<http://sphinx-doc.org/>) 等优秀的文档生成工具。此外，还有像 Read the Docs (<https://readthedocs.org/>) 这样，由 Python 开发者社区协力搭建的网站，能够为开源的 Python 项目提供美观的文档及免费的存放空间。
- 由于 Python 将文档视为第一等级的 (first-class) 对象，而且可以令开发者在程序中访问格式良好的文档信息，所以我们很乐意编写更多的文档。Python 社区的开发者都坚信：文档是非常重要的。代码必须要有完备的文档，才能称得上是好代码。由于有了这一信念，所以大部分开源的 Python 库，都应该会带有优雅的文档。

为了融入这种良好的文档撰写氛围，我们在编制文档字符串时，应该遵循一些规范。对细节问题的详细讨论，可以参阅 PEP 257 (<http://www.python.org/dev/peps/pep-0257/>)。下面这几条规范，是大家都应该遵守的。

1. 为模块编写文档

每个模块都应该有顶级的 `docstring`。这个字符串字面量，应该作为源文件的第一条语句。开发者应该在字符串两端，使用三重双引号 ("""") 把它括起来。之所以要编写 `docstring`，是为了介绍当前这个模块，以及模块之中的内容。

`docstring` 的头一行文字，应该是一句话，用以描述本模块的用途。它下面那段话，应该包含一些细节信息，把与本模块的操作有关的内容，告诉模块的使用者。我们还可

以在模块的 docstring 中，强调本模块里面比较重要的类和函数，使得开发者能够据此来了解该模块的用法。

下面列举一个模块级别的 docstring：

```
# words.py
#!/usr/bin/env python3
"""Library for testing words for various linguistic patterns.

Testing how words relate to each other can be tricky sometimes!
This module provides easy ways to determine when words you've
found have special properties.

Available functions:
- palindrome: Determine if a word is a palindrome.
- check_anagram: Determine if two words are anagrams.
...
"""

# ...
```

如果该模块是个命令行工具，那我们可以考虑把工具的用法，写在本模块的 docstring 里面，以便告诉用户应该如何从命令行里运行这个工具。

2. 为类编写文档

每个类都应该有类级别的 docstring，它的写法与模块级的 docstring 大致相同。头一行，也是用一句话来阐明本类的用途。接下来，用一段话详述该类的操作方式。

类中比较重要的 public 属性及方法，也应该在这个 docstring 里面加以强调。此外，还应该告诉子类的实现者，如何才能正确地与 protected 属性（参见本书第 27 条）及超类方法相交互。

下面示范一个类级别的 docstring：

```
class Player(object):
    """Represents a player of the game.

    Subclasses may override the 'tick' method to provide
    custom animations for the player's movement depending
    on their power level, etc.

    Public attributes:
    - power: Unused power-ups (float between 0 and 1).
    - coins: Coins found during the level (integer).
    ...
"""

# ...
```

3. 为函数编写文档

每个 public 函数及方法，都应该有 docstring，其写法，与模块和类级别的 docstring 相似。第一行，还是用一句话来描述本函数的功能。接下来，用一段话描述具体的行为和函数的参数。若函数有返回值，则应该在 docstring 中写明。如果函数可能抛出某些调用者必须处理的异常，而这些异常又是函数接口的一部分，那么 docstring 应该对其进行解释。

下面演示函数级别的 docstring：

```
def find_anagrams(word, dictionary):
    """Find all anagrams for a word.

    This function only runs as fast as the test for
    membership in the 'dictionary' container. It will
    be slow if the dictionary is a list and fast if
    it's a set.

    Args:
        word: String of the target word.
        dictionary: Container with all strings that
                    are known to be actual words.

    Returns:
        List of anagrams that were found. Empty if
        none were found.
    """
    # ...
    
```

为函数撰写 docstring 时，还有一些特例，大家应该了解：

- 如果函数没有参数，且仅有一个简单的返回值，那么，只需用一句话来描述该函数，应该就够了。
- 如果函数根本就没有返回值，那么最好别在 docstring 里面提到它，也就是说，不要在 docstring 里面出现“returns None”这样的说法。
- 如果函数在正常的使用过程中不会抛出异常，那就不要在 docstring 里面提到异常。
- 如果函数接受数量可变的位置参数（参见本书第 18 条）或数量可变的关键字参数（参见本书第 19 条），那就应该在文档的参数列表中，使用 *args 和 **kwargs 来描述它们的用途。
- 如果函数的参数有默认值，那么应该指出这些默认值（参见本书第 20 条）。
- 如果函数是个生成器（参见本书第 16 条），那么应该描述该生成器在迭代时所产生的内容。

- 如果函数是个协程（参见本书第 40 条），那么应该描述协程所产生的值，以及这个协程希望通过 `yield` 表达式来接纳的值，同时还要说明该协程会于何时停止迭代。

 提示 为模块编写 `docstring` 之后，一定要保证文档能够及时更新。通过内置的 `doctest` 模块，我们很容易就能运行 `docstring` 中的范例代码，以确保源代码和文档不会在开发过程中产生偏差。

要点

- 我们应该通过 `docstring`，为每个模块、类和函数编写文档。在修改代码的时候，应该更新这些文档。
- 为模块撰写文档时，应该介绍本模块的内容，并且要把用户应该了解的重要类及重要函数列出来。
- 为类撰写文档时，应该在 `class` 语句下面的 `docstring` 中，介绍本类的行为、重要属性，以及本类的子类应该实现的行为。
- 为函数及方法撰写文档时，应该在 `def` 语句下面的 `docstring` 中，介绍函数的每个参数，函数的返回值，函数在执行过程中可能抛出的异常，以及其他行为。

第 50 条：用包来安排模块，并提供稳固的 API

程序的代码量变大之后，我们自然就需要重新调整其结构。我们会把大函数分割成小函数，会把某些数据结构重构为辅助类（参见本书第 22 条），也会把功能分散到多个相互依赖的模块之中。

到了一定的阶段，我们就会发现，模块的数量实在太多了，于是，我们需要在程序之中引进一种抽象层，使得代码更加便于理解。Python 的包（*package*）就可以充当这样的抽象层。包，是一种含有其他模块的模块。

在大多数情况下，我们会给目录中放入名为 `__init__.py` 的空文件，并以此来定义包。只要目录里有 `__init__.py`，我们就可以采用相对于该目录的路径，来引入目录中的其他 Python 文件。例如，某个程序的目录结构如下。

```
main.py
mypackage/__init__.py
mypackage/models.py
mypackage/utils.py
```

为了以相对的方式引入 utils 模块，我们需要把上级模块的绝对名称，写在引入语句的 from 部分之中，也就是说，我们要在 from 关键字右侧，写出与 mypackage 包相对应的目录名。

```
# main.py
from mypackage import utils
```

如果某个包目录中还嵌套有其他的包（如 mypackage.foo.bar），那么也需要采用上述形式来编写 import 语句。

 提示 Python 3.4 提供了名称空间包（namespace package）这一机制，使我们能够以更加灵活的方式来定义包。名称空间包中的模块，可以来自完全不同的目录、zip 压缩文档，甚至远端系统。PEP 420 (<http://www.python.org/dev/peps/pep-0420/>) 详细介绍了开发者应该如何使用名称空间包的高级特性。

对于 Python 程序来说，包所提供的能力，主要有两大用途。

1. 名称空间

包的第一种用途，是把模块划分到不同的名称空间之中。这使得开发者可以编写多个文件名相同的模块，并把它们放在不同的绝对路径之下。例如，下面这个程序，能够从两个同名的 utils.py 模块中，分别引入属性[⊖]。由于这两个模块可以通过绝对路径来区分，所以这种引入方式是可行的。

```
# main.py
from analysis.utils import log_base2_bucket
from frontend.utils import stringify

bucket = stringify(log_base2_bucket(33))
```

但是，如果这些包里面定义的函数、类或子模块相互重名，那么上面的做法就失效了。例如，我们要从 analysis.utils 和 frontend.utils 包中，分别引入名为 inspect 的函数。此时不能像上面那样，直接引入这两个包中的属性，因为第二条 import 语句，会把当

[⊖] 本节中的属性（attribute）一词，多是泛称，用以指代本模块及其子模块中的变量、函数和类。——译者注

前作用域中的 `inspect` 值覆盖掉。

```
# main2.py
from analysis.utils import inspect
from frontend.utils import inspect # Overwrites!
```

解决办法是：在 `import` 语句中，通过 `as` 子句，给引入当前作用域中的属性重新起一个名字。

```
# main3.py
from analysis.utils import inspect as analysis_inspect
from frontend.utils import inspect as frontend_inspect

value = 33
if analysis_inspect(value) == frontend_inspect(value):
    print('Inspection equal!')
```

凡是通过 `import` 语句引入的内容，都可以用 `as` 子句来改名，即使引入整个模块，我们也依然能用 `as` 为其改名。这使得开发者能够轻松地访问位于不同名称空间之中的代码，并在引入该段代码的时候，阐明其身份。



还有一种办法，也可以避免引入互相冲突的名称，那就是：每次使用模块时，都从最高级的路径开始，完整地写出各模块的名称。

对于上例来说，我们可以先编写 `import analysis.utils` 和 `import frontend.utils` 语句，然后，分别通过 `analysis.utils.inspect` 和 `frontend.utils.inspect` 这样的完整路径，来访问那两个模块里的 `inspect` 函数。

这种办法完全不需要使用 `as` 子句，而且对于头一次接触本代码的读者而言，它可以非常清晰地指出每个 `inspect` 函数究竟定义在哪个模块里面。

2. 稳固的 API

Python 包的第二种用途，是为外部使用者提供严谨而稳固的 API。

如果要编写使用范围较广的 API，如编写开源包（参见本书第 48 条），那么，就需要提供一些稳固的功能，并保证它们不会因为版本的变动而受到影响。为此，我们必须把代码的内部结构对外隐藏起来，以便在不影响现有用户的前提下，通过重构来改善包内的模块。

在 Python 程序中，我们可以为包或模块编写名为 `_all_` 的特殊属性，以减少其暴露给外围 API 使用者的信息量。`_all_` 属性的值，是一份列表，其中的每个名称，都将作为本模块的一条公共 API，导出给外部代码。如果外部用户以 `from foo import *` 的

形式来使用 foo 模块，那么只有 foo.`__all__` 中列出的那些属性，才会从 foo 中引入。若是 foo 模块没有提供 `__all__`，则只会引入 public 属性，也就是说，只会引入不以下划线开头的那些属性（参见本书第 27 条）。

例如，要提供一个包，以计算移动的抛射体（projectile）之间的碰撞。下面定义的这个 models 模块，位于 mypackage 包中，该模块用来表示抛射体：

```
# models.py
__all__ = ['Projectile']

class Projectile(object):
    def __init__(self, mass, velocity):
        self.mass = mass
        self.velocity = velocity
```

然后，我们在这个 mypackage 包中，再定义一个 utils 模块，用来对 Projectile 实例执行某些操作，例如，模拟 Projectile 之间的碰撞。

```
# utils.py
from . models import Projectile

__all__ = ['simulate_collision']

def _dot_product(a, b):
    # ...

def simulate_collision(a, b):
    # ...
```

现在，我们想把这套 API 中所有的 public 部分，都作为 mypackage 模块的属性，提供给外界用户，使他们总是能够直接引入 mypackage，而不用再执行 mypackage.models 和 mypackage.utils 等形式的引入操作。这样一来，即使 mypackage 包的内部结构发生变化（例如，删掉了 models.py 模块），使用这些 API 的外部代码，也依然能照常运行。

为了在 Python 包中实现此功能，我们需要修改 mypackage 目录下的 `__init__.py` 文件。当外界代码引入 mypackage 模块的时候，该文件实际上也会成为 mypackage 的一部分内容。因此，我们可以限定 `__init__.py` 文件所引入的名称，以此来指明 mypackage 模块所要暴露给外界用户的 API。由于 mypackage 内部的那些模块，都已经提供了 `__all__` 属性，所以，我们只需把内部模块中的所有内容都引入进来，并据此更新 `__init__.py` 的 `__all__` 属性，即可把 mypackage 的 public 接口适当地公布给外界。

```
# __init__.py
__all__ = []
from . models import *
```

```
__all__ += models.__all__
from .utils import *
__all__ += utils.__all__
```

经过上述处理之后，API 的使用者就可以直接引入 mypackage，而不用再访问具体的内部模块了：

```
# api_consumer.py
from mypackage import *

a = Projectile(1.5, 3)
b = Projectile(4, 1.7)
after_a, after_b = simulate_collision(a, b)
```

请注意，像 mypackage.utils._dot_product 这样，专门在内部使用的函数，是不会作为 mypackage 包的 API 提供给使用者的，因为这些函数，没有出现在 __all__ 列表之中。在 __all__ 里省略某个名称，就意味着外部代码不会通过 from mypackage import * 语句导入该名称，这实际上相当于把内部专用的名称给隐藏起来了。

在必须给外界提供严谨而稳固的 API 时，上面这套方案相当合宜。但是，如果我们只想在自己所拥有的各模块之间构建内部的 API，那恐怕就用不到 __all__ 所提供的功能了，此时应该避免使用它。对于在大型代码库上相互协作的编程团队来说，包本身所提供的命名空间机制，通常来说，就已经足够用了，它可以令开发者在控制这份代码库的同时，保持合理的接口边界。

谨慎使用 import * 形式的引入语句

像 from x import y 这样的引入语句，是比较清晰的，因为我们可以明确看出 y 来源于 x 包或 x 模块。包含通配符的 from foo import * 等语句，也是很有用的，它特别适合在进行交互式 Python 编程的时候使用。但是，通配符会令代码变得有些难懂。

- 对于刚读到这份代码的人来说，他们无法通过 from foo import * 语句，指出某个名称的来源。如果模块里面有多条 import * 语句，那我们必须检查这些语句所引用的全部模块，才能确定某个名称到底定义在哪个模块之中。
- import * 语句会把外围模块中的同名内容覆盖掉。我们在代码中使用别的模块时，可能无意间通过多条 import * 语句，引入了一些重复的名称，而这种现象，会引发奇怪的 bug。

最安全的做法是：尽量不要在代码中使用 import * 语句，而是应该以 from x import y 形式的语句，明确指出自己想要引入的名称。

要点

- Python 包是一种含有其他模块的模块。我们可以用包把代码划分成各自独立且互不冲突的名称空间，使得每块代码都能具备独有的绝对模块名称。
- 只要把 `__init__.py` 文件放入含有其他源文件的目录里，就可以将该目录定义为包。目录中的文件，都将成为包的子模块。该包的目录下面，也可以含有其他包。
- 把外界可见的名称，列在名为 `__all__` 的特殊属性里，即可为包提供一套明确的 API。
- 如果想隐藏某个包的内部实现，那么我们可以在包的 `__init__.py` 文件中，只把外界可见的那些名称引入进来，或是给仅限内部使用的那些名称添加下划线前缀。
- 如果软件包只在开发团队或代码库内部使用，那可能没有必要通过 `__all__` 来明确地导出 API。

第 51 条：为自编的模块定义根异常，以便将调用者与 API 相隔离

在为模块定义其 API 时，该模块所抛出的异常，与模块里定义的函数和类一样，都是接口的一部分（参见本书第 14 条）。

Python 内置了一套异常体系，以供语言本身及标准库使用。于是，我们也总想使用这些内置的异常类型来报告错误，而不想自己去定义新的类型。例如，当外界给函数传入了无效的参数时，我们可能想抛出 `ValueError` 异常，以指出这一错误。

```
def determine_weight(volume, density):
    if density <= 0:
        raise ValueError('Density must be positive')
    # ...
```

在某些情况下，使用 `ValueError` 也许是比较合适的，但是在设计 API 时，还是应该自己来定义一套新的异常体系，这样会令 API 更加强大。我们可以在模块里面提供一种根异常（root Exception），然后，令该模块所抛出的其他异常，都继承自这个根异常。

```
# my_module.py
class Error(Exception):
    """Base-class for all exceptions raised by this module."""

class InvalidDensityError(Error):
    """There was a problem with a provided density value."""
```

模块里有了这种根异常之后，API 的使用者就可以轻松地捕获该模块所抛出的各种异常。例如，API 的使用者在调用模块中的某个函数时，可能就会通过 try/except 语句来捕获这个根异常：

```
try:
    weight = my_module.determine_weight(1, -1)
except my_module.Error as e:
    logging.error('Unexpected error: %s', e)
```

API 所抛出的异常，如果向上传播得太远，就会令程序崩溃，而使用 try/except 语句，则可以防止这种情况，因为它会把调用代码与 API 隔开。这种隔离，有三个好处。

首先，通过捕获根异常，调用者可以得知他们在使用你的 API 时，所编写的调用代码是否正确。如果调用者以合理的方式来使用这套 API，那么他们应该会捕获该模块所抛出的各种异常。若是某种异常没有得到处理，那么该异常就会传播到 try/except 语句中负责处理模块根异常的那个 except 块里面，而那个 except 块，则会把该异常告知 API 的使用者，提醒他们应该为这种类型的异常添加适当的处理逻辑。

```
try:
    weight = my_module.determine_weight(1, -1)
except my_module.InvalidDensityError:
    weight = 0
except my_module.Error as e:
    logging.error('Bug in the calling code: %s', e)
```

使用根异常的第二个好处，是可以帮助模块的开发者找寻 API 里的 bug。如果在编写模块代码时，只抛出本模块的异常体系中定义过的那些异常，那么，其他类型的异常，就不应该由这个模块抛出。若发现本模块抛出了其他类型的异常，则说明 API 的代码里有 bug。

不过，上面那种 try/except 语句，并不能把 API 使用者与 API 模块代码中的 bug 相隔离。如果要隔离，那么调用者需要再添加一个 except 块，以捕获 Python 的 Exception 基类。这样他们就能够查出：API 模块的实现代码里面是不是留有尚待修复的 bug。

```
try:
    weight = my_module.determine_weight(1, -1)
except my_module.InvalidDensityError:
    weight = 0
except my_module.Error as e:
    logging.error('Bug in the calling code: %s', e)
except Exception as e:
    logging.error('Bug in the API code: %s', e)
    raise
```

使用根异常的第三个好处，是便于 API 的后续演化。将来我们可能会在 API 里面提供更为具体的异常，以便在特定的情况下抛出。例如，可以添加下面这个 Exception 子类，当调用者提供了负的密度值时，就抛出该异常：

```
# my_module.py
class NegativeDensityError(InvalidDensityError):
    """A provided density value was negative."""

def determine_weight(volume, density):
    if density < 0:
        raise NegativeDensityError
```

添加了这个新的 NegativeDensityError 异常之后，原有的调用代码仍然能够继续运作，因为它所捕获的 InvalidDensityError 异常，正是这个新异常的父类。调用者以后可以针对这种新的异常类型，做出特殊的处理，并据此修改程序的行为。

```
try:
    weight = my_module.determine_weight(1, -1)
except my_module.NegativeDensityError as e:
    raise ValueError('Must supply non-negative density') from e
except my_module.InvalidDensityError:
    weight = 0
except my_module.Error as e:
    logging.error('Bug in the calling code: %s', e)
except Exception as e:
    logging.error('Bug in the API code: %s', e)
    raise
```

我们还可以在模块的根异常之下，直接定义一套更为广泛的异常，以便为 API 的后续演化做出更加充分的准备。例如，我们可以把计算重量时发生的错误归到某一类异常里面，把计算体积时发生的错误归到另一类异常里面，再把计算密度时发生的错误归到第三类异常里面。

```
# my_module.py
class WeightError(Error):
    """Base-class for weight calculation errors."""

class VolumeError(Error):
    """Base-class for volume calculation errors."""

class DensityError(Error):
    """Base-class for density calculation errors.”””
```

我们可以从上面这三个通用的异常类之中，继承更为具体的异常子类。上面这三个异常，介于模块的根异常和具体的子异常之间，所以它们本身也可以看作各自门类的根

异常。模块的使用者，可以根据这三个大的门类，把调用代码与 API 代码轻松地隔离开，而不用再像原来那样，编写冗长的 catch 块，把每一种具体的 Exception 子类都捕获一遍。

要点

- 为模块定义根异常，可以把 API 的调用者与模块的 API 相隔离。
- 调用者在使用 API 时，可以通过捕获根异常，来发现调用代码中隐藏的 bug。
- 调用者可以通过捕获 Python 的 Exception 基类，来帮助模块的研发者找寻 API 实现代码中的 bug。
- 可以从模块的根异常里面，继承一些中间异常，并令 API 的调用者捕获这些中间异常。这样模块开发者将来就能在不破坏原有调用代码的前提下，为这些中间异常分别编写具体的异常子类。

第 52 条：用适当的方式打破循环依赖关系

在和他人协作时，我们难免会写出相互依赖的模块。而有的时候，即使自己一个人开发程序，也仍然会写出相互依赖的代码。

例如，GUI（图形用户界面）程序要显示一个对话框，请用户来选择文档的保存地点。程序可以在事件处理器（event handler）里，把需要显示在对话框中的数据，通过参数传递过去。而对话框那边，也需要读取一些全局状态，例如，它要根据用户的配置信息（user preferences，用户偏好）来决定如何把自身正确地渲染出来。

下面定义的这个 dialog 模块，会从 app 模块的全局配置信息中，获取默认的文档保存地点：

```
# dialog.py
import app

class Dialog(object):
    def __init__(self, save_dir):
        self.save_dir = save_dir
    # ...

save_dialog = Dialog(appprefs.get('save_dir'))

def show():
    # ...
```

问题在于：包含 `prefs` 对象的那个 `app` 模块，又需要引入 `dialog` 模块，这样才能在程序启动的时候，把对话框显示出来。

```
# app.py
import dialog

class Prefs(object):
    # ...
    def get(self, name):
        # ...

prefs = Prefs()
dialog.show()
```

这就形成了循环依赖关系。如果想从主程序里面使用 `app` 模块，那么引入该模块时，就会出现异常。

```
Traceback (most recent call last):
  File "main.py", line 4, in <module>
    import app
  File "app.py", line 4, in <module>
    import dialog
  File "dialog.py", line 16, in <module>
    save_dialog = Dialog(app.prefs.get('save_dir'))
AttributeError: 'module' object has no attribute 'prefs'
```

为了理解上面这个错误的产生原因，我们必须要知道 Python 系统在执行 `import` 语句时的详细机制。引入模块的时候，Python 会按照深度优先的顺序执行下列操作：

- 1) 在由 `sys.path` 所指定的路径中，搜寻待引入的模块。
- 2) 从模块中加载代码，并保证这段代码能够正确编译。
- 3) 创建与该模块相对应的空对象。
- 4) 把这个空的模块对象，添加到 `sys.modules` 里面。
- 5) 运行模块对象中的代码，以定义其内容。

循环依赖所产生的问题是：某些属性必须要等 Python 系统把对应的代码执行完毕之后（也就执行完第 5 步之后），才可以具备完整的定义。但是，包含该属性的模块，却只需要等 Python 系统执行完第 4 步，就可以用 `import` 语句引入并添加到 `sys.modules` 里面了。

在上例中，`app` 模块在尚未定义任何内容之前，就先引入了 `dialog` 模块，然后，`dialog` 模块又引入了 `app` 模块。而这个时候，`app` 模块尚未完成整个引入过程，它还处于正在引入 `dialog` 的状态之中，按照上面第 4 步的规则，我们可以说，此时的 `app` 模

块，只是个空壳而已。由于 app 模块的第 5 步还没有完成，所以在 app 模块中，定义 prefs 对象所用的那段代码，就尚未得到运行，而 dialog 模块的第 5 步，却需要用到这个 prefs，于是，就抛出了 `AttributeError` 异常。

解决该问题的最佳方案，就是重构代码，把 `prefs` 数据结构放在依赖树的最底层。然后，令 `app` 与 `dialog` 模块都引入那个包含 `prefs` 的工具模块，以避免出现循环依赖关系。但是，我们未必总能够实现这种清晰的划分方式，而且有的时候，这样重构所耗费的精力实在太大了。

下面还有三种办法，也可以避免循环依赖关系。

1. 调整引入顺序

要介绍的第一个办法，是调整引入顺序。例如，我们可以把引入 `dialog` 模块所用的那条 `import` 语句，移动到 `app` 模块底部，也就是说，先等 `app` 模块的主要内容运行完毕，然后再引入 `dialog` 模块，这样 `AttributeError` 错误就会消失。

```
# app.py
class Prefs(object):
    # ...

prefs = Prefs()

import dialog # Moved
dialog.show()
```

这种办法是可行的。由于 `app` 模块在相当晚的时候才引入 `dialog` 模块，所以当 `dialog` 模块反向引用 `app` 时，`app` 的第 5 步几乎已经执行完了，因此，`dialog` 模块能够找到 `app.prefs` 的定义。

这个办法虽然能避开 `AttributeError` 错误，但是却与 PEP 8 风格指南（参见本书第 2 条）不符。PEP 8 推荐开发者总是在 Python 文件顶部引入其他模块。这样可以令第一次阅读该代码的人，知道本模块会依赖其他哪些模块，而且也能够保证引入的模块都位于本模块的范围内，从而使本模块内的所有代码都能使用那些模块。

这种稍后引入模块的做法，容易使代码出现问题，而且会导致代码的顺序发生少许变动，而这种变动，可能会令整个模块都无法运作。因此，我们不应该通过调整引入顺序来解决循环依赖问题。

2. 先引入、再配置、最后运行

解决循环引入的第二种办法，是尽量缩减模块在引入时所产生的副作用。也就是

说，只在模块中给出函数、类和常量的定义，而不要在引入的时候真正去运行那些函数。每个模块都将提供 `configure` 函数，等其他模块都引入完毕之后，我们要在该模块上面调用一次 `configure`，而这个 `configure` 函数，则会访问其他模块的属性，以便将本模块的状态准备好。等到所有模块都引入完毕（也就是第 5 步执行完），那些模块中的属性肯定已经定义好了，于是我们就可以放心地执行 `configure` 了。

下面重新定义 `dialog` 模块，令该模块只会在外界调用 `configure` 函数的时候，才去访问 `prefs` 对象：

```
# dialog.py
import app

class Dialog(object):
    # ...

save_dialog = Dialog()

def show():
    # ...

def configure():
    save_dialog.save_dir = app.prefs.get('save_dir')
```

同时也重新定义 `app` 模块，令它不要在引入的时候执行任何动作。

```
# app.py
import dialog

class Prefs(object):
    # ...

prefs = Prefs()

def configure():
    # ...
```

现在，我们在 `main` 模块中，分三个阶段来执行代码：首先引入所有模块，然后配置它们，最后执行程序中的第一个动作。

```
# main.py
import app
import dialog

app.configure()
dialog.configure()

dialog.show()
```

这种方案在很多情况下都非常合适，而且方便开发者实现依赖注入 (*dependency injection*) 等模式。但是，有时我们很难从代码中清晰地提取出 `configure` 步骤。另外，在模块内部划分不同的阶段，也会令代码变得不易理解，因为这样做，会把对象的定义与对象的配置分开。

3. 动态引入

解决循环引入的第三种办法，是在函数或方法内部使用 `import` 语句，这种办法是最为简单的。程序会等到真正需要运行相关的代码时，才去触发模块的引入操作，而不会在刚开始启动并初始化其他模块时，就去引入那个模块，所以，这种方案又称为动态引入 (*dynamic import*)。

下面，我们采用动态引入方案，来重新定义 `dialog` 模块。这一次，`dialog.show` 函数要等到运行的时候，才会引入 `app` 模块，而不是像原来那样，在初始化的时候就引入 `app` 模块。

```
# dialog.py
class Dialog(object):
    # ...

    save_dialog = Dialog()

def show():
    import app  # Dynamic import
    save_dialog.save_dir = app.prefs.get('save_dir')
    # ...

app
```

`app` 模块的写法，与最早那份范例代码一样，也是在开头就引入 `dialog`，并在结尾调用 `dialog.show`。

```
# app.py
import dialog

class Prefs(object):
    # ...

    prefs = Prefs()
    dialog.show()
```

该方案的实际效果，与刚才提到的先引入、再配置、最后运行的那套方案，是相似的。区别在于，本方案不需要从结构上面修改模块的定义方式和引入方式。我们只是把循环引入推迟到了程序真正需要访问其他模块的那一刻。而在那个时间点上，我们则可以确信，其他模块都已经彻底初始化好了（因为每个模块的第 5 步都已经完成了）。

一般来说，我们还是尽量不要使用这种动态引入方案。因为 import 语句的执行开销，还没有小到可以忽略不计的地步，而且在循环中反复引入模块，更是一种不好的编程方式。此外，这种旨在推迟代码执行时机的动态引入方案，还可能会在程序运行的时候，导致非常奇怪的错误，例如，程序会在运行了很久之后，突然抛出 SyntaxError 异常（参见本书第 56 条会给出避免此类问题的办法）。

要点

- 如果两个模块必须相互调用对方，才能完成引入操作，那就会出现循环依赖现象，这可能导致程序在启动的时候崩溃。
- 打破循环依赖关系的最佳方案，是把导致两个模块互相依赖的那部分代码，重构为单独的模块，并把它放在依赖树的底部。
- 打破循环依赖关系的最简方案，是执行动态的模块引入操作，这样既可以缩减重构所花的精力，也可以尽量降低代码的复杂度。

第 53 条：用虚拟环境隔离项目，并重建其依赖关系

如果程序构建得比较庞大、比较复杂，那么它通常会依赖 Python 社区中的许多软件包（参见本书第 48 条）。我们可能要在开发过程中，通过 pip 命令，安装 pytz、numpy 及其他一些软件包。

问题在于，通过 pip 命令安装的新软件包，是全局性的，也就是说，这些安装好的模块，可能会影响系统里的所有 Python 程序。从理论上讲，好像不应该出现这种问题。如果安装了某个软件包，但却从来不引入它，那么该软件包怎么会影响自己的程序呢？

然而，真正麻烦的地方却在于依赖性的传递（transitive dependency），也就是说，我们所安装的包，可能还要依赖其他一些包。例如，在安装完 Sphinx 包之后，可以通过 pip 命令来看看：该软件包还依赖其他哪些软件包。

```
$ pip3 show Sphinx
---
Name: Sphinx
Version: 1.2.2
Location: /usr/local/lib/python3.4/site-packages
Requires: docutils, Jinja2, Pygments
```

如果还安装了 flask 等包，那么也可以用 pip 命令查询它的依赖关系。我们会看到，

它与 Sphinx 一样，都依赖 Jinja2 包。

```
$ pip3 show flask
---
Name: Flask
Version: 0.10.1
Location: /usr/local/lib/python3.4/site-packages
Requires: Werkzeug, Jinja2, itsdangerous
```

Sphinx 包与 flask 包以后可能会各自演化，而这也许就会导致冲突。这两个包，目前或许都在使用相同版本的 Jinja2，所以暂时相安无事。但半年或一年之后，Jinja2 可能会发布新的版本，而那个新版本，可能包含一些重大的变化，从而对使用 Jinja2 库的其他模块造成影响。如果我们通过 pip install --upgrade 命令来更新整个系统的 Jinja2 包，那么可能就会出现 Sphinx 包无法使用，而 flask 包却可以照常使用的奇怪现象。

这个问题的根源是：在同一时刻，Python 只能把模块的某一个版本，安装为整个系统的全局版本。如果某个已经安装好的软件包，必须使用新版模块，而另外一个已经安装好的软件包，又必须使用旧版模块，那么系统就没办法正常运作了。

即使软件包的维护者尽力保持新旧版本之间的 API 兼容性（参见本书第 50 条），这种问题也依然有可能发生。因为新版的程序库在行为上面可能发生了比较微妙的变化，而使用这套 API 的原有代码，又依赖于这些行为。用户可能更新了系统中的某一个软件包，但却没有更新其他软件包，从而导致依赖关系遭到破坏。所以说，软件包之间的这种递进式依赖关系，总是有风险的。

当我们与其他开发者相互协作，而那些开发者又分别在各自的电脑上面编程时，这个问题就更加严重了。有理由相信：他们所安装的 Python 版本及全局软件包的版本，可能与你所安装的版本略有区别。这就会产生一种尴尬的局面，也就是说：同一份代码，在某位程序员的电脑上可以很好地运行，而在另一位程序员的电脑上，却完全无法运作。

这些问题都可以通过名为 pyvenv 的工具来解决，此工具提供了一套虚拟环境 (*virtual environment*)。从 Python 3.4 开始，这个工具会随着 Python，默认安装到电脑中，开发者可以在命令行界面里，通过 pyvenv 来调用它，也可以通过 python -m venv 命令来访问它。对于早前的 Python 版本，我们必须用 pip install virtualenv 命令单独安装这个工具包，并在命令行里通过 virtualenv 来使用它。

pyvenv 使得我们可以创建版本互不相同的 Python 环境。通过 pyvenv，我们可以在同一个系统上面，同时安装某软件包的多个版本，并且使这些版本之间不发生冲突。这



样就能在同一台电脑上面，用多种不同的工具来开发多个不同的项目。

借助 pyenv，我们可以把不同版本的软件包以及其依赖关系，分别安装在彼此隔绝的目录结构之中，这使得我们可以重现一套特定的 Python 开发环境，以确保某个项目的代码肯定能够在这套环境下面正常运作。于是，就可以有效地避免因软件包的依赖关系而导致的各种奇怪问题。

1.pyenv 命令

下面我们扼要地讲解 pyenv 命令的使用方式。在使用该工具之前，首先要确定命令行中 python3 命令，在系统里的含义。在笔者的电脑上，python3 位于 /usr/local/bin 目录下，并且会指向 3.4.2 版本（参见本书第 1 条）。

```
$ which python3
/usr/local/bin/python3
$ python3 --version
Python 3.4.2
```

我们可以运行一条引入 pytz 模块的 Python 命令，看它会不会出错，以此来检验这套开发环境配置得是否正确。由于笔者已经把 pytz 软件包安装为全局模块，所以这条 Python 命令能够正确地执行。

```
$ python3 -c 'import pytz'
$
```

现在，用 pyenv 命令来新建名为 myproject 的虚拟环境。每一套虚拟环境，都必须位于各自独立的目录之中。这条命令执行完毕后，该目录下面会产生相应的目录树与文件。

```
$ pyenv /tmp/myproject
$ cd /tmp/myproject
$ ls
bin      include      lib      pyvenv.cfg
```

为了启用这套虚拟环境，我们在命令行界面中，使用 source 命令来运行 bin/activate 脚本。activate 脚本会修改所有的环境变量，使之与虚拟环境相匹配。它还会更新命令提示符，把虚拟环境的名称（本例中，是 'myproject'）包含进来，使开发者可以明确地知道自己当前所处的工作环境。

```
$ source bin/activate
(myproject)$
```

激活这套虚拟环境之后，可以看到，命令行中的 python3，已经不再直接指向整个系统中的 Python 命令了，而是会指向虚拟环境目录中的那个 Python 命令。

```
(myproject)$ which python3
/tmp/myproject/bin/python3
(myproject)$ ls -l /tmp/myproject/bin/python3
... -> /tmp/myproject/bin/python3.4
(myproject)$ ls -l /tmp/myproject/bin/python3.4
... -> /usr/local/bin/python3.4
```

这就能够确保外围系统不会影响这套虚拟环境。即便外围系统中的 `python3` 命令已经更新到 3.5 版，虚拟环境中的 `python3` 命令依然会指向 3.4 版。

用 `pyvenv` 所创建的这套虚拟环境，除了 `pip` 与 `setuptools`，是没有安装任何软件包的。外围系统虽然已经把 `pytz` 包安装为全局模块，但是虚拟环境却并不知道有这个包，所以，如果我们在虚拟环境里使用它，就会报错。

```
(myproject)$ python3 -c 'import pytz'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named 'pytz'
```

我们可以用 `pip` 命令把 `pytz` 模块安装到虚拟环境里。

```
(myproject)$ pip3 install pytz
```

安装好之后，可以用刚才那条命令验证。

```
(myproject)$ python3 -c 'import pytz'
(myproject)$
```

使用完虚拟环境之后，可以通过 `deactivate` 命令返回默认的系统。这将把开发环境恢复到系统的默认值，其中，`python3` 命令行工具，也将指向原来的位置。

```
(myproject)$ deactivate
$ which python3
/usr/local/bin/python3
```

如果想重新回到 `myproject` 环境，那就和原来一样，用 `source bin/activate` 命令运行 `myproject` 目录中的 `activate` 脚本。

2. 重建项目的依赖关系

有了虚拟环境之后，我们就可以用 `pip` 命令来安装自己所需的软件包了。以后，我们可能想把自己这套环境复制到其他地方。例如，要把这套开发环境部署到产品服务器中，或要把别人的开发环境克隆到自己的电脑上面，以运行其中的代码。

`pyvenv` 可以轻松地解决上述需求。我们用 `pip freeze` 命令，把开发环境对软件包的依赖关系，明确地保存到文件之中。按惯例，这个文件应该叫做 `requirements.txt`。

```
(myproject)$ pip3 freeze > requirements.txt
(myproject)$ cat requirements.txt
numpy==1.8.2
pytz==2014.4
requests==2.3.0
```

现在，假设想再构建一套与 myproject 相符的虚拟环境，那么，还是可以像原来那样，用 pyvenv 命令来新建环境目录，并且用 activate 脚本来激活新环境。

```
$ pyvenv /tmp/otherproject
$ cd /tmp/otherproject
$ source bin/activate
(otherproject)$
```

这套新的开发环境，目前并没有安装其他软件包。

```
(otherproject)$ pip3 list
pip (1.5.6)
setuptools (2.1)
```

刚才我们在第一套虚拟环境里，通过 pip freeze 命令生成了 requirements.txt 文件，而现在，则可以根据该文件，用 pip install 命令把所有相关的软件包都安装到第二套虚拟环境之中。

```
(otherproject)$ pip3 install -r /tmp/myproject/requirements.txt
```

上面这条命令，必须花费一定的时间才能运行完毕，因为它要把所需的软件包下载并安装到目前的环境中，使其与第一套虚拟环境相匹配。执行完这条命令后，我们列出第二套环境里面所安装的软件包，大家可以看到，这份依赖关系列表，与第一套环境中的列表是相同的。

```
(otherproject)$ pip list
numpy (1.8.2)
pip (1.5.6)
pytz (2014.4)
requests (2.3.0)
setuptools (2.1)
```

如果我们正在通过修订控制系统 (revision control system)^②与他人协作，那么使用 requirements.txt 文件来描述依赖关系，是一种相当好的办法。在提交代码的时候，我们可以同时更新这份描述软件包依赖关系的列表，使两者之间保持同步。

使用虚拟环境时，有个地方要注意，就是别去移动环境目录，因为所有的路径（包括 python3 命令所指向的路径），都以硬编码的形式写在了安装目录之中，如果移动了，

^② 也称版本控制系统。——译者注

那么环境就会失效。然而这并不是个大问题。因为虚拟环境的用途，就在于使开发者能够方便地重建一套与原环境相似的配置。所以，我们一般都不会移动虚拟环境所在的目录，而是会用 `pip freeze` 命令把旧环境的依赖关系导出，然后创建新的环境，并根据 `requirements.txt` 文件，把旧环境中的软件包重新安装到新环境之中。

要点

- 借助虚拟环境，我们可以在同一台电脑上面同时安装某软件包的多个版本，而且能保证它们不会冲突。
- `pyvenv` 命令可以创建虚拟环境，`source bin/activate` 命令可以激活虚拟环境，`deactivate` 命令可以停用虚拟环境。
- `pip freeze` 命令可以把某套环境所依赖的软件包，汇总到一份文件里面。我们把这个 `requirements.txt` 文件提供给 `pip install -r` 命令，即可重建一套与原环境相仿的新环境。
- 如果使用 Python 3.4 之前的版本做开发，那么必须单独下载并安装类似的 `pyvenv` 工具。那个命令行工具不叫 `pyvenv`，而是叫做 `virtualenv`。

部署

若想将 Python 程序付诸应用，就必须把它从开发环境部署到生产环境之中。令程序支持好几套不同的配置方案颇有些难度。我们要设法保证程序在各种情境下都能可靠地运作，这与实现正确的功能，是一样重要的。

所以，我们要将 Python 程序产品化 (*productionize*)，令其能够稳定地运行在实际环境中。开发者可以通过某些内置的 Python 模块，把程序打造得更加健壮。这些模块所提供的调试、优化和测试功能，可以尽力提升程序的品质及运行时的效率。

第 54 条：考虑用模块级别的代码来配置不同的部署环境

所谓部署环境 (*deployment environment*) 就是程序在运行的时候所用的一套配置。每个程序至少都会有一种部署环境，这就是生产环境 (*production environment*, 产品环境)。我们制作程序的首要目标，就是为了能在生产环境中运行它，并产生某种成效。

但是，在编写和修改程序代码的过程中，我们却必须要在开发程序所用的那台电脑上面运行它。而这套开发环境的配置方式，可能与生产环境有很大区别。例如，我们可能会在 Linux 工作站上面，为超级计算机编写程序。

`pyvenv` 等工具（参见本书第 53 条），使得开发者能够保证所有的环境都装有同一套 Python 软件包。但问题在于，生产环境通常还会依赖很多外部的先决条件（*external*

assumption)，而那些先决条件，很难在开发环境里重现。

例如，我们要在 Web 服务器容器中运行某个程序，并通过该程序访问数据库，那么，每次修改完程序的代码，我们都要把服务器容器运行起来，把数据库设置好，并输入访问数据库所需的密码。我们可能只是修改了程序中的一行代码而已，但要验证修改后的程序是否正确，却要花费这么大的精力。

解决此类问题的最佳方案，是在程序启动的时候，覆写其中的某些部分，以便根据部署环境，来提供不同的功能。例如，我们可能会编写两份不同的 `__main__` 文件，一份用于生产环境，另一份用于开发环境。

```
# dev_main.py
TESTING = True
import db_connection
db = db_connection.Database()

# prod_main.py
TESTING = False
import db_connection
db = db_connection.Database()
```

这两份文件唯一的区别，就在于 `TESTING` 常量的取值。于是，程序中的其他模块就可以引入 `__main__` 模块，并通过 `TESTING` 的值来决定如何定义自身的属性。

```
# db_connection.py
import __main__

class TestingDatabase(object):
    # ...

class RealDatabase(object):
    # ...

if __main__.TESTING:
    Database = TestingDatabase
else:
    Database = RealDatabase
```

这个范例的关键点在于：出现在模块范围之内，但又不包含在函数或方法之中的那些代码，实际上就是普通的 Python 代码。于是，我们可以在这种模块级的代码中，用 `if` 语句来决定本模块应该如何定义相关的变量。这使得开发者可以根据各种不同的部署环境来定制这些模块。在不需要配置数据库的时候，我们就可以不配置数据库，从而避免高昂的开销，此时，我们可以用一些虚构的或伪造的数据来实现相关的逻辑，以简化互动式开发及测试的过程（参见本书第 56 条）。



如果部署环境变得过于复杂，那我们就应该考虑把 TESTING 这样的 Python 常量，从代码中移走，并把它们放到专门的配置文件里面。开发者可以通过 configparser 等内置模块，把生产环境中所需的配置信息与产品代码相分离，尤其是在与运维团队相互协作的时候，更应注意这一点。

这套方案不仅可以应对某些外部的先决条件，而且还有着其他的用途。例如，有的程序必须根据宿主操作系统来决定其运作方式，此时，我们就可以在定义顶级的结构之前，先通过 sys 模块来判断当前的操作系统。

```
# db_connection.py
import sys

class Win32Database(object):
    # ...

class PosixDatabase(object):
    # ...

if sys.platform.startswith('win32'):
    Database = Win32Database
else:
    Database = PosixDatabase
```

同理，我们也可以通过 os.environ 来查询环境变量，并据此来定义模块的内容。

要点

- 程序通常需要运行在各种不同的部署环境之中，而这些环境所需的先决条件及配置信息，也都互不相同。
- 我们可以在模块范围内，编写普通的 Python 语句，以便根据不同的部署环境，来定制本模块的内容。
- 我们可以根据外部条件来决定模块的内容，例如，通过 sys 和 os 模块来查询宿主操作系统的特性，并以此来定义本模块中的相关结构。

第 55 条：通过 repr 字符串来输出调试信息

调试 Python 程序时，print 函数（以及内置的 logging 模块中的那些输出函数）可以给我们极大的帮助。由于 Python 的内部信息，一般都可以通过普通的属性来获取（参见

本书第 27 条)，所以我们只需在程序运行的时候，用 print 打印出程序状态，并根据状态的变化来寻找相关的错误即可。

print 函数会根据开发者传给它的参数，打印出一条便于认读的字符串。例如，如果把某个简单的字符串传给 print，那它就会打印出不带外围引号的字符串内容。

```
print('foo bar')
>>>
foo bar
```

这样做的效果，就相当于用 '%s' 做格式化字符串，并用 % 操作符来打印 'foo bar'。

```
print('%s' % 'foo bar')
>>>
foo bar
```

然而问题在于，这种便于阅读的字符串，并不能清晰地展示该值的类型。例如，在默认情况下，我们无法根据 print 函数所打印的内容，来区分数值型的 5 与字符串类型的 '5'。

```
print(5)
print('5')

>>>
5
5
```

在通过 print 函数来调试程序时，这种类型之间的差别是相当重要的。所以，我们在调试某个对象时，应该打印 repr 版本的字符串。内置的 repr 函数，会根据某个对象，返回可供打印的表示形式 (*printable representation*)，而这种形式，应该算是最为清晰且又易于理解的一种字符串表示形式。对于内置的类型来说，由 repr 函数所返回的字符串，是一条有效的 Python 表达式。

```
a = '\x07'
print(repr(a))

>>>
'\x07'
```

把 repr 函数所返回的值，传给内置的 eval 函数，应该就会得到与原来的那个 Python 对象了（下面的这种写法，纯粹是为了演示。在实际编程中，我们当然会非常谨慎地使用 eval 函数）。

```
b = eval(repr(a))
assert a == b
```

调试程序的时候，我们应该先把待调试的值传给 repr 函数，然后再用 print 将其打

印出来，这样可以明确体现出类型之间的差别。

```
print(repr(5))
print(repr('5'))

>>>
5
'5'
```

这样做的效果，与用 '%r' 做格式化字符串，并用 % 操作符来打印该值是相同的。

```
print('%r' % 5)
print('%r' % '5')

>>>
5
'5'
```

对于动态的 Python 对象来说，默认的易读字符串，与 repr 函数所返回的字符串是相同的。这就是说，我们只需把动态对象直接传给 print 函数，即可打印出 repr 字符串的内容，而不需要再于打印前先调用 repr 函数。可是，object 实例默认给出的那个 repr 值，对调试来说，并不是特别有用。下面定义一个简单的类，并打印出该类对象的值：

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

obj = OpaqueClass(1, 2)
print(obj)

>>>
<__main__.OpaqueClass object at 0x107880ba8>
```

print 所输出的内容，并不能传给 eval 函数，而且从中也看不出该对象各实例字段的取值。

解决此问题有两种办法。如果我们可以控制该类的源代码，那么就定义名为 __repr__ 的特殊方法，并令该方法返回一个包含 Python 表达式的字符串，那条 Python 表达式，可以用来重建该对象。下面就来给刚才那个类定义 __repr__ 方法：

```
class BetterClass(object):
    def __init__(self, x, y):
        # ...

    def __repr__(self):
        return 'BetterClass(%d, %d)' % (self.x, self.y)
```

现在，repr 字符串的值，就显得更有意义了。

```
obj = BetterClass(1, 2)
print(obj)
```

```
>>>
BetterClass(1, 2)
```

若是无法修改该类的定义，那我们可以通过对象的 `__dict__` 属性来查询它的实例字典。下面这段代码，可以打印出 `OpaqueClass` 实例的内容：

```
obj = OpaqueClass(4, 5)
print(obj.__dict__)
```

```
>>>
{'y': 5, 'x': 4}
```

要点

- 针对内置的 Python 类型来调用 `print` 函数，会根据该值打印出一条易于阅读的字符串，这个字符串隐藏了类型信息。
- 针对内置的 Python 类型来调用 `repr` 函数，会根据该值返回一条可供打印的字符串。把这个 `repr` 字符串传给内置的 `eval` 函数，就可以将其还原为初始的那个值。
- 在格式化字符串里使用 `%s`，能够产生与 `str` 函数的返回值相仿的易读字符串，而在格式化字符串里使用 `%r`，则能够产生与 `repr` 函数的返回值相仿的可打印字符串。
- 可以在类中编写 `__repr__` 方法，用自定义的方式来提供一种可供打印的字符串，并在其中给出更为详细的调试信息。
- 可以在任意对象上面查询 `__dict__` 属性，以观察其内部信息。

第 56 条：用 `unittest` 来测试全部代码

Python 没有静态类型检查机制。编译器不能保证程序一定会在运行的时候正确地执行。Python 并不清楚程序里面调用的那些函数，在运行的时候是否会具备正确的定义，即便源代码中已经明确给出了这些函数的定义，Python 也依然不清楚这一点。所以，这种动态的行为，是既有利也有弊的。

许多 Python 程序员都认为，这样做是值得的，因为它可以令代码更加短小、更加简洁，从而提高编程效率。但是，我们都会或多或少地听人说起：Python 程序有可能在运行的时候突然出现荒唐的错误。

在笔者所知的事例中，比较糟糕的一种情况是：某个已投入运营的 Python 程序，因为使用了动态引入（参见本书第 52 条），而导致 SyntaxError 异常。这个突然出现的错误，把那位程序员吓坏了，他从此再也不用 Python 编程了。

不过，笔者感到奇怪的是，那个程序的代码，为什么还没有经过测试，就直接部署到生产环境中了呢？就算有类型安全机制，我们也不能掉以轻心。无论用哪种语言编程，我们都应该测试自己所写的代码。Python 语言与其他编程语言相比，确实有个显著的区别，那就是：只有通过编写测试，我们才能够确信程序在运行的时候不会出问题。我们不能通过静态类型检查来获得安全感。

Python 语言的动态特性，一方面阻碍了静态类型检查，另一方面却也使得开发者能够非常容易地为代码编写测试。我们利用 Python 的动态特性来覆写相关的行为，以实现测试，并确保程序能按预期的方式运作。

我们可以把编写测试看成给程序代码购买保险。良好的测试使我们能够确信：代码可以在运行的时候表现出正确的行为。重构代码或扩充代码之后，可以通过测试非常方便地判断出代码的行为有没有发生改变。良好的测试，实际上会使开发者在修改 Python 程序时感到更加方便，而不是更加困难，这话初听起来似乎有点不可思议。

要编写测试，最简单的办法，就是使用内置的 unittest 模块。例如，在 utils.py 文件中定义了下面这个工具函数：

```
# utils.py
def to_str(data):
    if isinstance(data, str):
        return data
    elif isinstance(data, bytes):
        return data.decode('utf-8')
    else:
        raise TypeError('Must supply str or bytes, '
                        'found: %r' % data)
```

然后，创建一份 test_utils.py 或 utils_test.py 文件，并在该文件中测试自己所期望的行为。

```
# utils_test.py
from unittest import TestCase, main
from utils import to_str

class UtilsTestCase(TestCase):
    def test_to_str_bytes(self):
        self.assertEqual('hello', to_str(b'hello'))
```

```

def test_to_str_str(self):
    self.assertEqual('hello', to_str('hello'))

def test_to_str_bad(self):
    self.assertRaises(TypeError, to_str, object())

if __name__ == '__main__':
    main()

```

测试是以 TestCase 类的形式来组织的。每个以 test 开头的方法，都是一项测试。如果测试方法在运行过程中，既没有抛出任何类型的 Exception，也没有因 assert 语句而导致 AssertionError，那么该测试就算顺利通过。

TestCase 类提供了一些辅助方法，以供开发者在编写测试的时候做出各种断言 (assertion)，例如，`assertEqual` 可以判断两值是否相等，`assertTrue` 可以验证 Boolean 表达式是否为真，`assertRaises` 可以验证程序能否在适当的时机抛出相关异常（详细用法可参阅 `help(TestCase)`）。在 TestCase 子类中，我们也可以自己定义一些辅助方法，令测试代码更加便于阅读，只是要注意，这些辅助方法不能以 test 开头。



编写测试时，还有一种常用的做法，就是通过 `mock` 函数或 `mock` 类^① 来替换受测程序中的某些行为。Python 3 内置的 `unittest.mock` 模块可以实现此功能，Python 2 中也有开源软件包可供使用。

有的时候，我们在运行测试方法之前，需要先在 TestCase 类里面把测试环境配置好。于是，我们就覆写 `setUp` 和 `tearDown` 方法。系统在执行每个测试之前，都会调用一次 `setUp` 方法，在执行完每个测试之后，也都会调用一次 `tearDown` 方法，这就使得各项测试之间，可以彼此独立地运行（在测试程序时，一定要确保各项测试之间不能相互干扰）。例如，下面定义的这个 TestCase 子类，会在执行每项测试前，先创建临时目录，并在执行完每项测试后，删除目录内容。

```

class MyTest(TestCase):
    def setUp(self):
        self.test_dir = TemporaryDirectory()
    def tearDown(self):
        self.test_dir.cleanup()
    # Test methods follow
    # ...

```

^① mock 一词，中文称作模拟、模仿、仿制或仿件。——译者注

笔者经常会把一组相关的测试，放入一个 `TestCase` 子类中。有的时候，如果某函数有很多种边界状况（edge case）[⊖]需要测试，那就针对这个函数专门编写一个 `TestCase` 子类。其他情况下，笔者可能会把某个模块内的所有函数，全部放在同一个 `TestCase` 里面测试。另外，笔者也会针对每个类来编写 `TestCase`，以便测试该类及类中的所有方法。

当程序变得复杂之后，我们可能还要编写另外一些测试。那种测试，并不是孤立地检验各个模块，而是要验证模块之间能否正确地互动。这就是单元测试（unit test）与集成测试（integration test）的区别。对于 Python 程序来说，这两种测试都很重要，我们必须通过集成测试来证明各模块之间确实能够协调地运作，否则，程序的正确性就得不到保证。



提 示 有些项目可能还需要定义数据驱动测试（data-driven test），或是需要把测试项目按照功能划分到不同的套件（suite）之中。关于这些用法，以及测试覆盖度报告（code coverage report）和其他一些高级用法，请参见 nose (<http://nose.readthedocs.org/>) 或 pytest (<http://pytest.org/>) 等开源软件包。

要点

- 要想确信 Python 程序能够正常运行，唯一的办法就是编写测试。
- 内置的 `unittest` 模块提供了测试者所需的很多功能，我们可以借助这些机制写出良好的测试。
- 我们可以在 `TestCase` 子类中，为每一个需要测试的行为，定义对应的测试方法。`TestCase` 子类里的测试方法，其名称必须以 `test` 开头。
- 我们必须同时编写单元测试和集成测试，前者用来独立检验程序中的每个功能，而后者则用来检验模块之间的交互行为。

第 57 条：考虑用 `pdb` 实现交互调试

编写程序的时候，我们总会遇到代码中的 bug。`print` 函数可以帮我们追查到很多问题的来源（参见本书第 55 条）。针对具体用例编写测试，也是一种隔离可疑代码并寻找

[⊖] 是指理论上完全合理，但实际上比较罕见的状况。——译者注

错误根源的好办法。

但是这些手段未必总是能查到问题的根源。如果要使用更为强大的调试工具，那就请试试 Python 内置的交互调试器（*interactive debugger*）。这种调试器能够检视程序状态、打印局部变量，并能够以步进的方式来执行程序中的每一条语句。

在其他大部分编程语言中，我们先必须告诉调试器应该在源代码的哪一行停下来^②，然后再调试程序。但 Python 则不是这样，最简单的调试手法，就是修改程序，并在可疑的代码上方直接启动调试器，以探查代码中的问题。用调试器来运行 Python 程序，与正常运行 Python 程序，是没有区别的。

我们只需引入内置的 `pdb` 模块，并运行其 `set_trace` 函数，即可触发调试器。这两个操作，通常会写在同一行之中，这使得开发者在不需要调试的时候，能够用一个井号，把整行代码注释掉。

```
def complex_func(a, b, c):
    # ...
    import pdb; pdb.set_trace()
```

只要运行到那行语句，程序就会暂停。执行该程序所用的那个终端机，会转入交互式的 Python 提示符界面。

```
-> import pdb; pdb.set_trace()
(Pdb)
```

在 (Pdb) 提示符界面中，我们可以输入局部变量的名称，以打印它们的值^③，也可以调用内置的 `locals` 函数，来列出所有的局部变量，还可以引入模块、检视全局状态、构建新对象、运行内置的 `help` 函数^④，甚至修改程序中的某个部分。凡是有助于调试的操作，都可以在调试器界面中执行。此外，调试器还提供了三个命令，可以令我们更加方便地查看正在调试的程序。

- `bt`：针对当前执行点的调用栈，打印其回溯（`traceback`）信息。可以据此判断出程序当前执行到了哪个位置，也可以看出程序是如何从最开头运行到触发 `pdb.set_trace` 函数的这一点的。
- `up`：把调试范围沿着函数调用栈上移一层，回到当前函数的调用者那里。该命令使得我们可以检视调用栈上层的局部变量。
- `down`：把调试范围沿着函数调用栈下移一层。

^② 这种操作，称为设置断点。——译者注

^③ 在变量名称前面加英文半角惊叹号，即可打印该变量。如 `!a`。——译者注

^④ 也需要在前面加 `!`，否则会打印 `pdb` 本身的命令帮助信息。——译者注

检视过当前的状态之后，我们可以用下面几个调试器命令来精确地控制程序，使其继续往下执行。

- ❑ `step`：执行当前这行代码，并把程序继续运行到下一条可执行的语句开头，然后把控制权交还给调试器。如果当前这行代码中带有函数调用操作，那么调试器会进入受调用的那个函数，并停留在那个函数开头。
- ❑ `next`：执行当前这行代码，并把程序继续运行到下一条可执行的语句开头，然后把控制权交还给调试器。如果当前这行代码中带有函数调用操作，那么调试器不会停留在函数里面，而是会调用那个函数，并等待其返回。
- ❑ `return`：继续运行程序，直至到达当前函数的 `return` 语句开头，然后把控制权交还给调试器。
- ❑ `continue`：继续运行程序，直至到达下一个断点或下一个 `set_trace` 调用点。

要点

- ❑ 我们可以修改 Python 程序，在想要调试的代码上方直接加入 `import pdb; pdb.set_trace()` 语句，以触发互动调试器。
- ❑ Python 调试器也是一个完整的 Python 提示符界面，我们可以检视并修改受测程序的状态。
- ❑ 我们可以在 `pdb` 提示符中输入命令，以便精确地控制程序的执行流程，这些命令使得我们能够交替地查看程序状态并继续向下运行程序。

第 58 条：先分析性能，然后再优化

由于 Python 是一门动态语言，所以 Python 程序的运行效率可能与我们预想的结果有很大差距。有一些操作，我们认为应该执行得比较慢，但实际上却很快，例如，对字符串的各种操作，以及生成器等；而另外一些语言特性，我们认为应该执行得比较快，但实际上却很慢，例如，属性访问及函数调用等操作。所以，导致 Python 程序效率低下的真正原因，可能是很难看出来的。

应对性能问题的最佳方式，是在优化程序之前先分析其性能，而不是靠直觉去判断。Python 提供了内置的性能分析工具 (*profiler*)[⊖]，它可以计算出程序中某个部分

[⊖] 也称性能分析器、效能测评器，本书将酌情保留该词的英文写法。——译者注

的执行时间，在总体执行时间中所占的比率。通过这些数据，可以找到最为显著的性能瓶颈，并把注意力放在优化这部分代码上面，而不要在不影响速度的那些地方浪费精力。

例如，我们想查明程序中的某个算法为什么运行得比较慢。下面定义的这个函数，采用插入排序法（insertion sort）来排列一组数据：

```
def insertion_sort(data):
    result = []
    for value in data:
        insert_value(result, value)
    return result
```

插入排序法的核心机制，就是 `insert_value` 函数，该函数用来查找每项数据的插入点。下面定义的这个 `insert_value` 函数，要对外界输入的 `array` 序列进行线性扫描，以确定插入点，因此，它的效率是非常低的：

```
def insert_value(array, value):
    for i, existing in enumerate(array):
        if existing > value:
            array.insert(i, value)
            return
    array.append(value)
```

为了分析 `insertion_sort` 和 `insert_value` 的效率[⊖]，笔者创建了一组随机数字，并定义了 `test` 函数，以便将该函数传给 `profiler`。

```
from random import randint

max_size = 10**4
data = [randint(0, max_size) for _ in range(max_size)]
test = lambda: insertion_sort(data)
```

Python 提供了两种内置的 `profiler`，一种是纯 Python 的 `profiler`（名字叫做 `profile`），另外一种是 C 语言扩展模块（名字叫做 `cProfile`）。在这两者中，内置的 `cProfile` 模块更好，因为它在做性能分析时，对受测程序的效率只会产生很小的影响，而纯 Python 版的 `profiler`，则会产生较大的开销，从而使测试结果变得不够准确。



提示 分析 Python 程序的性能时，我们要分析的是程序代码本身的性能，而不是外部系统的性能。因此，对于访问网络信息或磁盘资源的那些函数来说，这一点要

[⊖] 分析某函数的效率或性能，也称为对该函数做 `profile`。——译者注

多加注意。由于底层网络系统或磁盘系统的速度比较慢，所以从分析结果上看，我们也许会误以为那些函数占据了大量的执行时间。如果程序采用缓存来降低此类资源的延迟时间，那么在开始分析性能之前，一定要先把缓存配置好。

下面实例化 cProfile 模块中的 Profile 对象，并通过 runcall 方法来运行刚才定义的 test 函数：

```
profiler = Profile()
profiler.runcall(test)
```

test 函数运行完毕之后，我们采用内置的 pstats 模块和模块中的 Stats 类，来剖析由 Profile 对象所收集到的性能统计数据。Stats 对象提供了各种方法，我们可以用这些方法对性能分析数据进行遴选及排序，以便把自己所关注的那部分信息单独列出来。

```
stats = Stats(profiler)
stats.strip_dirs()
stats.sort_stats('cumulative')
stats.print_stats()
```

上面那段代码会输出一张表格，其中的信息是按照函数来分组的。表格中的数据，是在 profiler 处于激活状态的时候统计出来的，也就是说，这些时间数据，都是在执行刚才那个 runcall 方法的过程中统计出来的。

```
>>>
20003 function calls in 1.812 seconds

Ordered by: cumulative time

      ncalls  tottime  percall  cumtime  percall filename:lineno(function)
            1    0.000    0.000    1.812    1.812 main.py:34(<lambda>)
            1    0.003    0.003    1.812    1.812 main.py:10(insertion_sort)
        10000    1.797    0.000    1.810    0.000 main.py:20(insert_value)
       9992    0.013    0.000    0.013    0.000 {method 'insert' of 'list' objects}
         8    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
            1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

下面简述性能统计表中每一列的含义：

- ncalls：该函数在性能分析期间的调用次数。
- tottime：执行该函数所花的总秒数。本函数因调用其他函数所耗费的时间，不记入在内。
- tottime percall：每次调用该函数所花的平均秒数。本函数因调用其他函数所耗费的时间，不记入在内。此值等于 tottime 与 ncalls 相除的商。
- cumtime：执行该函数及其中的全部函数调用操作，所花的总秒数。

□ cumtime percall：每次执行该函数及其中的全部函数调用操作，所花的平均秒数。

此值等于 cumtime 与 ncalls 相除的商。

通过 profiler 给出的统计表可以看到，在 cumulative time（累积时间）一栏中，insert_value 函数所占用的 CPU 份额是最大的。于是，我们改用内置的 bisect 模块（参见本书第 46 条）来重新定义此函数：

```
from bisect import bisect_left

def insert_value(array, value):
    i = bisect_left(array, value)
    array.insert(i, value)
```

再次运行 profiler。根据新生成的这张性能统计表可以看出，新函数要比原来快得多，insert_value 函数所耗费的累积时间，几乎是原来的百分之一。

```
>>> 30003 function calls in 0.028 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1    0.000    0.000    0.028    0.028 main.py:34(<lambda>)
      1    0.002    0.002    0.028    0.028 main.py:10(insertion_sort)
  10000    0.005    0.000    0.026    0.000 main.py:112(insert_value)
  10000    0.014    0.000    0.014    0.000 {method 'insert' of 'list' objects}
  10000    0.007    0.000    0.007    0.000 {built-in method bisect_left}
      1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

有时，在分析整个程序的性能时，可能会发现，某个常用的工具函数，占据了大部分执行时间。而从 profiler 所给出的默认统计数据里，我们却无法清晰地分辨出：程序中的不同部分，究竟是如何调用这个工具函数的。

例如，程序中有两个不同的函数，都会频繁调用下面这个名为 my_utility 的工具函数：

```
def my_utility(a, b):
    # ...

def first_func():
    for _ in range(1000):
        my_utility(4, 5)

def second_func():
    for _ in range(10):
        my_utility(1, 3)

def my_program():
    for _ in range(20):
        first_func()
        second_func()
```

在分析完范例代码的性能，并用 print_stats 打出默认的结果之后，我们就会发现：输出的统计结果是令人困惑的。

```
>>>
20242 function calls in 0.208 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    1    0.000    0.000    0.208    0.208 main.py:176(my_program)
    20    0.005    0.000    0.206    0.010 main.py:168(first_func)
20200    0.203    0.000    0.203    0.000 main.py:161(my_utility)
    20    0.000    0.000    0.002    0.000 main.py:172(second_func)
    1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

毫无疑问，my_utility 函数是占据执行时间最多的函数，但是，我们并不清楚程序究竟为什么要把该函数调用那么多次。在翻阅程序代码的过程中，可以看到很多个调用 my_utility 的地方，所以，这种格式的性能统计结果，是不够清晰的。

Python 的 profiler 提供了一种方式，可以在性能分析数据中列出每个函数的调用者，使我们可以据此看出该函数所耗费的执行时间，究竟是由哪些调用者所分别引发的。

```
stats.print_callers()
```

在这次打印出来的性能统计数据中，左边列出的是受测函数，右边列出的是该函数的调用者。据此可以看出，调用 my_utility 最为频繁的是 first_func 函数，my_utility 的大部分执行时间都是由 first_func 所引发的：

```
>>>
Ordered by: cumulative time

Function                                was called by...
                                         ncalls  tottime  cumtime
main.py:176(my_program)                  <-
main.py:168(first_func)                  <-      20    0.005    0.206  main.py:176(my_program)
main.py:161(my_utility)                  <- 20000    0.202    0.202  main.py:168(first_func)
                                         <-      200    0.002    0.002  main.py:172(second_func)
main.py:172(second_func)                  <-      20    0.000    0.002  main.py:176(my_program)
```

要点

- 优化 Python 程序之前，一定要先分析其性能，因为 Python 程序的性能瓶颈通常很难直接观察出来。
- 做性能分析时，应该使用 cProfile 模块，而不要使用 profile 模块，因为前者能够给出更为精确的性能分析数据。

- 我们可以通过 Profile 对象的 runcall 方法来分析程序的性能，该方法能够提供性能分析所需的全部信息，它会按照树状的函数调用关系，来单独地统计每个函数的性能。
- 我们可以用 Stats 对象来筛选性能分析数据，并打印出我们所需的那一部分，以便据此了解程序的性能。

第 59 条：用 tracemalloc 来掌握内存的使用及泄漏情况

在 Python 的默认实现方式，也就是 CPython 中，内存管理是通过引用计数来处理的。这样做可以保证：当指向某个对象的全部引用都过期的时候，受引用的这个对象也能够同时得到清理。另外，CPython 还内置了循环检测器（cycle detector），使得垃圾回收机制能够把那些自我引用的（self-referencing）对象清理掉。

从理论上说，这意味着大部分 Python 程序员都不用担心程序的内存分配和内存释放问题，因为 Python 语言及 CPython 运行时系统会自动把这些问题处理好。但实际上，程序还是会因为保留了过多的引用而导致内存耗尽。要想在 Python 程序中找出消耗内存或泄漏内存的地方，其实是比较困难的。

调试内存使用状况的第一种办法，是向内置的 gc 模块进行查询，请它列出垃圾收集器当前所知的每个对象。尽管这是个相当笨拙的工具，但这种做法确实能够使我们迅速得知程序的内存使用状况。

下面先运行一个程序，该程序会保留许多个指向相关对象的引用，进而浪费一些内存。然后，打印出程序执行期间所创建的对象数量，并在程序所分配的这些对象之中，选取一个较小的样本，将其展示出来。

```
# using_gc.py
import gc
found_objects = gc.get_objects()
print('%d objects before' % len(found_objects))

import waste_memory
x = waste_memory.run()
found_objects = gc.get_objects()
print('%d objects after' % len(found_objects))
for obj in found_objects[:3]:
    print(repr(obj)[:100])

>>>
4756 objects before
```

```
14873 objects after
<waste_memory.MyObject object at 0x1063f6940>
<waste_memory.MyObject object at 0x1063f6978>
<waste_memory.MyObject object at 0x1063f69b0>
```

`gc.get_objects` 函数的缺点是，它不能告诉我们这些对象到底是如何分配出来的。在较为复杂的程序中，代码会以多种不同的方式，来分配某个类的对象，所以，我们不仅要知道对象的总数量，而且更为重要的是，要知道这些对象究竟是由哪一部分代码分配出来的，了解到这一信息，才可以更好地判断内存泄漏的原因。

Python 3.4 推出了一个新的内置模块，名叫 `tracemalloc`，它可以解决这个问题。`tracemalloc` 可以把某个对象与该对象的内存分配地点联系起来。下面用 `tracemalloc` 打印出导致内存用量增大的前三个对象：

```
# top_n.py
import tracemalloc
tracemalloc.start(10) # Save up to 10 stack frames

time1 = tracemalloc.take_snapshot()
import waste_memory
x = waste_memory.run()
time2 = tracemalloc.take_snapshot()

stats = time2.compare_to(time1, 'lineno')
for stat in stats[:3]:
    print(stat)

>>>
waste_memory.py:6: size=2235 KiB (+2235 KiB), count=29981 (+29981), average=76 B
waste_memory.py:7: size=869 KiB (+869 KiB), count=10000 (+10000), average=89 B
waste_memory.py:12: size=547 KiB (+547 KiB), count=10000 (+10000), average=56 B
```

通过上述信息，我们立刻就能看出导致内存用量变大的主要因素，以及分配那些对象的语句在源代码中的位置。

`tracemalloc` 模块也可以打印出 Python 系统在执行每一个分配内存操作时，所具备的完整堆栈信息（full stack trace），打印的最大栈帧数量，由传给 `start` 函数的参数来决定。下面找到程序中最消耗内存的那个内存分配操作，并将该操作的堆栈信息打印出来。

```
# with_trace.py
# ...
stats = time2.compare_to(time1, 'traceback')
top = stats[0]
print('\n'.join(top.traceback.format()))

>>>
File "waste_memory.py", line 6
```

```

self.x = os.urandom(100)
File "waste_memory.py", line 12
    obj = MyObject()
File "waste_memory.py", line 19
    deep_values.append(get_data())
File "with_trace.py", line 10
    x = waste_memory.run()

```

如果程序中有多个地点都调用了同一个函数，那么通过上述方式就可以更好地看出，究竟是哪一行调用代码导致内存占用量变大的。

Python 2 虽然没有内置这个 `tracemalloc` 模块，但是有许多开源软件包（如 `heapy`[⊖] 等）也可以追踪内存用量，然而它们在功能上面，与 `tracemalloc` 并不是完全相同的。

要点

- Python 程序的内存使用情况和内存泄漏情况是很难判断的。
- 我们虽然可以通过 `gc` 模块来了解程序中的对象，但是并不能由此看出这些对象究竟是如何分配出来的。
- 内置的 `tracemalloc` 模块提供了许多强大的工具，使得我们可以找出导致内存使用量增大的根源。
- 只有 Python 3.4 及后续版本，才支持 `tracemalloc` 模块。

[⊖] 该项目的主页是 pypi.python.org/pypi/guppy/。——译者注

推荐阅读



Python入门经典：以解决计算问题为导向的Python编程实践

作者：(美) William F. Punch 等 ISBN: 978-7-111-39413-6 定价: 79.00元



编写高质量代码：改善Python程序的91个建议

作者：张颖等 ISBN: 978-7-111-46704-5 定价: 59.00元



Python编程实战：运用设计模式、并发和程序库创建高质量程序

作者：(美) Mark Summerfield ISBN: 978-7-111-47394-7 定价: 69.00元



树莓派Python编程指南

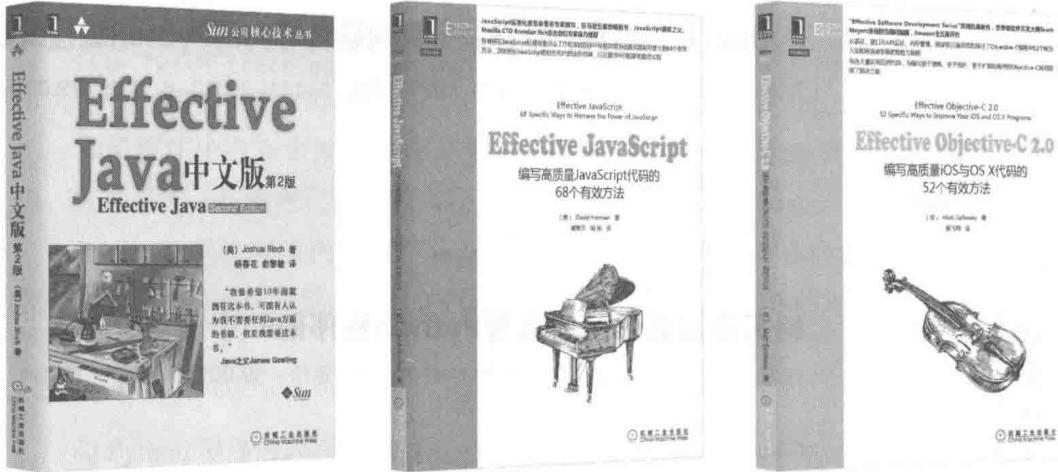
作者：(美) Alex Bradbury 等 ISBN: 978-7-111-48986-3 定价: 59.00元



Python学习手册 (原书第4版)

作者：(美) Mark Lutz ISBN: 978-7-111-32653-3 定价: 119.00元

推荐阅读



Effective Java中文版 第2版

作者：Joshua Bloch ISBN：978-7-111-25583-3 定价：52.00元

本书介绍了在Java编程中78条极具实用价值的经验规则，这些经验规则涵盖了大多数开发人员每天所面临的问题的解决方案。通过对Java平台设计专家所使用的技术的全面描述，揭示了应该做什么，不应该做什么才能产生清晰、健壮和高效的代码。

本书中的每条规则都以简短、独立的小文章形式出现，并通过例子代码加以进一步说明。本书内容全面，结构清晰，讲解详细。可作为技术人员的参考用书。

Effective JavaScript：编写高质量JavaScript代码的68个有效方法

作者：David Herman ISBN：978-7-111-44623-1 定价：49.00元

作者将在JavaScript标准化委员会工作和实践的多年经验浓缩为极具实践指导意义的68个有效方法，深刻辨析JavaScript的特性和内部运作机制，以及编码中的陷阱和最佳实践。

Effective Objective-C 2.0：编写高质量iOS与OS X代码的52个有效方法

作者：Matt Galloway ISBN：978-7-111-45129-7 定价：69.00元

本书是世界级C++开发大师Scott Meyers亲自担当顾问编辑的“Effective Software Development Series”系列丛书中的新作，Amazon全五星评价。从语法、接口与API设计、内存管理、框架等7大方面总结和探讨了Objective-C编程中52个鲜为人知和容易被忽视的特性与陷阱。书中包含大量实用范例代码，为编写易于理解、便于维护、易于扩展和高效的Objective-C应用提供了解决方案。



“Slatkin所写的这本书，其每个条目（item）都是一项独立的教程，并包含它自己的源代码。这种编排方式，使我们可以随意跳读：大家可以按照学习的需要来浏览这些item。本书涉及的话题十分广泛，作者针对这些话题，给出了相当精练而又符合主流观点的建议，我把这本书推荐给中级Python程序员。”

—— Brandon Rhodes，Dropbox的软件工程师、2016至2017年PyCon会议主席

用Python编写程序，是相当容易的，所以这门语言非常流行。但若想掌握Python所特有的优势、魅力和表达能力，则相当困难，而且语言中还有很多隐藏的陷阱，容易令开发者犯错。

本书可以帮你掌握真正的Pythonic编程方式，令你能够完全发挥出Python语言的强大功能，并写出健壮而高效的代码。Scott Meyers在畅销书《Effective C++》中开创了一种以使用场景为主导的精练教学方式，本书作者Brett Slatkin就以这种方式汇聚了59条优秀的实践原则、开发技巧和便捷方案，并以实用的代码范例来解释它们。

Slatkin根据自己在Google公司多年开发Python基础架构所积累的经验，揭示Python语言中一些鲜为人知的微妙特性，并给出了能够改善代码功能及运行效率的习惯用法。通过本书，你能够了解到解决关键编程任务所用的最佳方式，并学会编写易于理解、便于维护且利于改进的代码。

本书核心内容包括：

- 涵盖Python 3.x及Python 2.x主要应用领域的实用指南，以及与之配套的详细解释及代码范例。
- 与函数相关的编程建议，这些建议有助于我们写出意图清晰、便于复用且缺陷较少的函数。
- 如何准确地表达类与对象的行为。
- 在使用元类和动态属性时，如何避免错误的用法。
- 更为高效的并发及并行方式。
- 与Python内置模块相关的编程技巧和习惯用法。
- 多人协作时所用的开发工具和最佳实践方式。
- 旨在改善代码质量和程序性能的调试、测试与优化方案。

