

Effective 系列丛书

# Effective Python: 编写高质量 Python 代码的 59 个有效方法

Effective Python: 59 Specific Ways to Write Better Python

[ 美 ] 布雷特 · 斯拉特金 (Brett Slatkin) 著

爱飞翔译



## 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:

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》<sup>⊖</sup>  
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](mailto:eastarstormlee@gmail.com)，或访问 [github.com/jeffreybaoshenlee/zh-translation-errata-effective-python/issues](https://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<sup>⊖</sup>。书中的错误也会张贴到该网站<sup>⊖</sup>。

---

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

⊖ 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

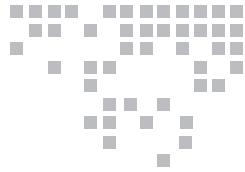
致 谢



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
<b>第 3 章 类与继承 .....</b>	<b>53</b>
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
<b>第 4 章 元类及属性 .....</b>	<b>84</b>
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
<b>第 5 章 并发及并行 .....</b>	<b>112</b>
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
<b>第 6 章 内置模块 .....</b>	<b>145</b>
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
<b>第 7 章 协作开发 .....</b>	<b>170</b>
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
<b>第 8 章 部署 .....</b>	<b>193</b>
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<sup>⊖</sup>。然而，在命令行中输入

<sup>⊖</sup> 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`<sup>②</sup> 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。

---

② 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。



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<sup>⊖</sup>；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：

---

<sup>⊖</sup> 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<sup>⊖</sup>，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）。

---

<sup>⊖</sup> 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）<sup>◎</sup>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）。

---

<sup>◎</sup> 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<sup>⊖</sup>。

```
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
```

---

<sup>⊖</sup> 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<sup>⊖</sup>，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]
```

---

<sup>⊖</sup> 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<sup>⊖</sup>，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）。

---

<sup>⊖</sup> 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<sup>⊖</sup>。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<sup>⊖</sup>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))
```

---

<sup>⊖</sup> 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

<sup>⊖</sup> 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

```

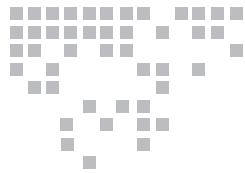
这种写法很有用，因为这四块代码互相配合得非常到位。例如，即使 else 块在写入 result 数据时发生异常，finally 块中关闭文件句柄的那行代码，也依然能执行。

## 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<sup>⊖</sup>。

---

<sup>⊖</sup> 这种写法必须要有 except 块。——译者注



Chapter 2

## 第 2 章

# 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。二元组的第一个元素，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<sup>⊖</sup>。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

```

---

<sup>⊖</sup> 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<sup>⊖</sup> (*scoping bug*)，它可能会使 Python 新手感到困惑。

---

<sup>⊖</sup> 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）<sup>⊖</sup>。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）。

---

<sup>⊖</sup> 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<sup>①</sup>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：

---

<sup>①</sup> 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())    # New iterator
    result = []
    for value in get_iter():   # New iterator
        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<sup>⊖</sup>。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
```

---

<sup>⊖</sup> 详情可参阅 Python 开发文档：[docs.python.org/3/library/stdtypes.html#typeiter](https://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*）<sup>⊖</sup>。

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
```

---

<sup>⊖</sup> 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<sup>⊖</sup>，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，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`

---

<sup>⊖</sup> 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 `TypeError`s，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。