

Python 社区集体智慧的结晶



# Python Cookbook™

(第2版) 中文版

Alex Martelli

[美] Anna Martelli Ravenscroft 编

David Ascher

高铁军 译

O'REILLY®

人民邮电出版社  
POSTS & TELECOM PRESS

O'REILLY®

# Python Cookbook™

## (第2版) 中文版

[美] Alex Martelli Anna Martelli Ravenscroft David Ascher 编  
高铁军 译

人民邮电出版社

北京



## 图书在版编目 (C I P) 数据

Python Cookbook (第2版) 中文版 / (美) 马特利  
(Martelli, A.) , (美) 马特利 (Martelli, A.) , (美)  
阿舍尔 (Ascher, D.) 编 ; 高铁军译. -- 北京 : 人民邮  
电出版社, 2010.5  
ISBN 978-7-115-22266-4

I. ①P… II. ①马… ②阿… ③高… III. ①软件工  
具—程序设计 IV. ①TP311.56

中国版本图书馆CIP数据核字(2010)第022417号

## 版权声明

Copyright©2005 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom  
Press, 2010. Authorized translation of the English edition, 2005 O'Reilly Media, Inc., the owner  
of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

本书中文简体版由 O'Reilly Media, Inc. 授权人民邮电出版社出版。未经出版者书面许可，  
对本书的任何部分不得以任何方式复制或抄袭。

版权所有，侵权必究。

## Python Cookbook (第2版) 中文版

◆ 编 [美] Alex Martelli Anna Martelli Ravenscroft  
David Ascher

译 高铁军

责任编辑 刘映欣

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号

邮编 100061 电子函件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

北京鑫正大印刷有限公司印刷

◆ 开本：787×1000 1/16

印张：49.5

字数：1031 千字

2010 年 5 月第 1 版

印数：1~4 000 册

2010 年 5 月北京第 1 次印刷

著作权合同登记号 图字：01-2009-1822 号

ISBN 978-7-115-22266-4

定价：99.00 元

读者服务热线：(010) 67132705 印装质量热线：(010) 67129223

反盗版热线：(010) 67171154

## 内 容 提 要

本书介绍了 Python 应用在各个领域中的一些使用技巧和方法，从最基本的字符、文件序列、字典和排序，到进阶的面向对象编程、数据库和数据持久化、XML 处理和 Web 编程，再到比较高级和抽象的描述符、装饰器、元类、迭代器和生成器，均有涉及。书中还介绍了一些第三方包和库的使用，包括 Twisted、GIL、PyWin32 等。本书覆盖了 Python 应用中的很多常见问题，并提出了通用的解决方案。书中的代码和方法具有很强的实用性，可以方便地应用到实际的项目中，并产生立竿见影的效果。尤为难得的是，本书的各位作者都具有丰富的业界实践经验，因此，本书不仅给出了对各种问题的解决方案，同时还体现了很多专家的思维方式和良好的编程习惯，与具体的细节性知识相比，这部分内容无疑是本书的精华。

本书适合具有一定 Python 基础的读者阅读参考。



## O'Reilly Media, Inc.介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构 O'Reilly Media, Inc. 授权人民邮电出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在 UNIX、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的 *The Whole Internet User's Guide & Catalog*（被纽约公共图书馆评为 20 世纪最重要的 50 本书之一）到 GNN（最早的 Internet 门户和商业网站），再到 WebSite（第一个桌面 PC 的 Web 服务器软件），O'Reilly Media, Inc. 一直处于 Internet 发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc. 是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得 O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc. 还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc. 依靠他们及时地推出图书。因为 O'Reilly Media, Inc. 紧密地与计算机业界联系着，所以 O'Reilly Media, Inc. 知道市场上真正需要什么图书。



---

# 译者序

我认为 Python Cookbook 是一本很独特的书。很多 Python 的书籍，其内容都具有一定的相似性和重叠，比如 Python 的学习书籍和教材，我们会想起 Core Python Programming、Python Programming 等，说到 Python 的系统管理，会想起 Python for UNIX and Linux System Administration、Pro Python System Administration 等书，说起 Python 的 XML 处理，会想起 XML Processing with Python 和 Python & XML，而像 Python Cookbook 这样的书，则是独一无二的，我想不出另外一本和它相似的书。它的独特，体现在几个方面。首先，它不是由一个或几个人写的，而是由几百人贡献的心得和技巧综合而成，所以，它包含了各种思考的方式和看待问题的角度。这些作者中，既有普通的爱好者，也有一些名气极大的技术“牛人”们（如 Guido van Rossum、Tim Peters、Raymond Hettinger、Mark Hammond 等）。他们都有着极其丰富的工程实践经验，因此，他们给出的技巧心得都非常实用，具有极强的可操作性，读者可以轻易地修改书中的示例并应用于自己的项目。另一方面，这些作者对 Python 的理解非常深入，他们不仅仅提出了对问题的解决之道，同时也在字里行间注入了 Python 的思维方式，即解决问题只是一方面，更重要的是怎么用更 Pythonic 的方式解决问题，怎么赋予解决过程更多的美感。比起具体的各种问题的解决之法，我认为这才是本书最精华的部分。最后，此书各个部分并没有严密的逻辑联系，所以，看书的方式也可以很随意，读者既可以从头到尾循序渐进地逐章逐节阅读，也可以跳跃式阅读，甚至可以随机抽取章节阅读，一切悉听尊便，只要适合个人习惯即可。

这本书主要是针对 Python 2.4，同时也考虑到了对 Python 2.3 的支持。不过自从此书出版之后，Python 的发展也似乎得到了加速，2006 年 8 月，Python 2.5 发布，2008 年 10 月，Python 2.6 发布，仅仅几个月之后，Python 3.0 横空出世，现在最新的版本已经是 3.1 了。有人可能会担心，现在这本书的内容会不会太陈旧了。我认为其实无须担心，请参考 <http://www.python.org> 列出的各个版本的 What's New。从 Python 2.4 到 Python 2.5，一些新的模块被加入了，比如 `xml.etree`、`sqlite` 以及 `ctypes`，还包括了一些新的特性，比如条件表达式的新写法、`with` 语句、统一的 `try/except/finally` 等，此外还有几百个补丁以及 bug 的修正。至于从 Python 2.5 到 Python 2.6 的变化，根据 A.M. Kuchling 的说法，2.6 主要是为 Python 3.0 做铺垫的，Python 2.6 在保持对以前代码的兼容性的同时还包含了 3.0 的新特性和新语法。所以，至少 99% 的 Python 2.4 的代码在 Python 2.6 中仍可以通行无阻。虽然 Python 3.0 和 Python 3.1 不再考虑对 Python 2.x 代码的向后兼容性了，但绝大多数的 Python 2.x 的代码都可以略作修改继续运行。在 3.x 中，一些语法或

者内建函数和对象的行为略有调整，但核心的机制和特性并未改变，所以此书中的大多数技巧仍然有效。关于 Python 2.5、2.6 以及 3.0 和 3.1 的 What's New，请参看 <http://docs.python.org/dev/whatsnew/2.5.html>、<http://docs.python.org/dev/whatsnew/2.6.html>、<http://docs.python.org/dev/py3k/whatsnew/3.0.html> 和 <http://docs.python.org/dev/py3k/whatsnew/3.1.html>。

本书显然不是为程序的初学者准备的，甚至不是为有其他编程语言经验的初次接触 Python 的读者准备的。当然，阅读本书并不要求读者首先成为 Python 语言的专家，只要有 Python 编程语言的基础就可以了，如果读者已经拥有了其他语言的编程经验，或者有很多 Python 编程的经验，那本书就更适用了。

最后，由于本人才疏学浅，虽然竭力想做到译文准确流畅（优美就更不奢望了），但终究力所不逮，不妥和疏漏肯定是难以避免的。因此，我非常欢迎读者的批评指正和宝贵意见。

# 前言

这本书不是一本典型的 O'Reilly 风格的书，而是一本集合了多个作者的手稿的作品。实际上，这也是一种将开源开发的方式应用到书籍出版业的尝试。Python 社区有超过 300 个成员在本书中贡献了他们的心得和资料。在这里，我们作为编辑，想给你——本书的读者，介绍一些重要的背景资料，这些背景资料是关于此书是如何编著出来，以及这个过程和涉及的人，并提出一些关于这种崭新的风格的思考。

## 本书的构思

在 2000 年初，O'Reilly 的主编 Frank Willison 联系到我（David Ascher），问我是否想写一本书。Frank 曾是 *Learning Python* 一书的编辑，那本书是由我和 Mark Lutz 合著而成。由于当时我正在致力于 *Perl shop* (ActiveState) 一书的编写，实在没有精力再同时开始另一本书的工作。但我和 Frank 总是定期地通过 E-mail 或者聊天工具对该书的一些主题进行探讨。Frank 有个主意，他受到 Tom Chriistiansen 和 Nathan Torkington 编写的 *Perl Cookbook* 一书的启发，认为出一本类似的 *Python Cookbook* 将会很有趣。Frank 想重复 *Perl Cookbook* 一书的成功，但他也认为应该让更多的人参与到此书的编写中。他的想法是，作为一本真正的 cookbook，更多人的参与将会提供更多的思路、角度和品味。至于书籍的质量，他认为可以通过技术编辑的把关和 O'Reilly 严格的编辑流程来保障。

Frank 和 ActiveState 的 CEO Dick Hardt 意识到 Frank 的目标和 ActiveState 的目标其实是殊途同归的。ActiveState 的想法是为开源程序员创造一个网络社区，叫做 ActiveState Programmer's Network (ASPN)。ActiveState 有一个人气很高的网站，提供了很丰富和多样化的内容。ActiveState 充分认识到开源社区具有很强大的力量，它能够提供最新和最准确的内容，无论多么生僻的方面均有涉及。

O'Reilly 和 ActiveState 很快认识到他们的目标上的相似性，并且意识到双方合作是最好的达成目标的方式。他们想要实现的包括：

- 创建一个由 Python 程序员提供素材的在线 Python Cookbook，为 Python 程序员服务；
- 出版一本包括了最好素材的书，该书由 Python 社区中的一些关键人物提供的材料和想法综合而成；
- 双方通过完成这本书亦可学习到一种不同的著书模式。

与此同时，有两件事情也需提及。首先，ActiveState 的一些人，包括 Paul Prescod，开始积极地寻找一些星级人物来加入 ActiveState 的开发小组中。其中一位候选者是著名的 Alex Martelli（但那时我们还没听说过他）。Alex 出名的原因是他在 Python 邮件列表中发表过大量透彻全面的文章，同时他也具有极好的耐心来解释 Python 的各种细微精巧之处，而且他非常高兴通过他自身的努力为 Python 带来了更多的新用户。我们不知道他的原因是，他住在意大利，而且对于 Python 社区来说他还算是一个新用户，Python 老手们也从来没见过他。但其实在 20 世纪 80 年代的时候，Alex 居住在美国，为 IBM 研发中心工作，并且很狂热地使用和推介其他的一些高阶语言（那时，大多是 IBM 的 Rexx）。

ActiveState 热切地期望 Alex 的到来，并试图劝说他搬到温哥华。我们已经是如此接近成功，可惜他的雇主给他带上了个金手铐，而且温哥华的天气明显无法与意大利的宜人气候竞争。所以，最后 Alex 仍然留在意大利，这让我很失望。即便如此，Alex 当时仍然在与 O'Reilly 接洽和商议写一本书。Alex 想写一本 cookbook 类型的书，但 *Python Cookbook* 当时已经签约。所以来 Alex 和 O'Reilly 签了另外一本书 *Python in Nutshell* 的合同。

第二件同时进行的事就是创建 Python 软件基金会。由于各种各样的原因，这件事被当做最好的部分，在某个会议的结束酒会上被提出来，Python 社区的成员想要创立一个非盈利性的组织，作为各种各样 Python 作品的知识产权的拥有者，这样可以确保 Python 及其衍生的作品有更加坚实的法律支撑。当然，这样的一个组织需要经济上的支持和 Python 社区的支持才能成功。

在综合考虑当时的各种情况之后，各方达成了以下协议：

- ActiveState 将会创建一个在线 cookbook，提供一种机制让任何人可以提交技巧（比如，针对某个特定问题的一段 Python 代码，以及相关的一些探讨，用以阐明为何以及在什么条件下使用这个窍门）。为了激发作者的积极性和读者的互动性，这个网站还允许读者提出修改建议和问题。
- 作为我的 ActiveState 工作的一部分，我将编辑和确保这些材料的质量。Alex Martelli 也加入了这个项目，作为合作编辑，为那些将要出版的材料把关，另外还有 Anna Martelli Ravenscroft，我们一起作为此书第 2 版的主要编辑。
- O'Reilly 将那些最好的技巧和心得集中起来出版，形成本书。
- 作为本书作者版税，此书售卖的部分收入将捐赠给 Python 软件基金会。

## 本书的实现

在线 cookbook（在 <http://aspn.activestate.com/ASPN/Cookbook/Python/>）是所有的技巧和心得材料的一个入口。读者可以通过填写表格和一些相关信息来注册免费的账户。那

些材料现在已经变成了本书的一部分。成千上万的用户阅读过那些材料，并且给出了相应的评论，因此，在本书的出版发行过程中，相关的材料已经变得越来越成熟和丰富。同时，作者的名字被印在书中也会吸引更多的人去访问在线 cookbook。随着每月新的技巧和心得的持续增加，以及邮件列表中越来越多的人频繁地引用在线 cookbook，可以期望这个在线 cookbook 也会取得极大的成功，这将再次证明作者的工作带给读者的巨大价值。

既然已经拥有了网站中的大量素材，那么实现这本书的工作就变成了挑选、合并、排序以及一些编辑工作了。这部分工作的一些细节将会在前言的“组织方式”部分介绍。

## 使用本书中的代码

本书的目的是为了帮助你更好地完成工作。一般而言，可以随意在自己的程序或者文档中使用书中代码。不需要联系我们来获得许可，除非使用了相当大一部分代码作为产品的一部分。比如，使用书中的代码写几个程序，无须获得许可。但是贩卖含有本书的代码的光盘则需要获得许可。回答问题时引用本书中的代码和文字不需要获得许可。而从本书中择取一定数量的代码用于你自己产品的文档则需要获得许可。我们感激但不强求使用者给出出处。一个完整的出处通常包括了标题、作者、出版商以及 ISBN。举个例子：“*Python Cookbook, 2<sup>nd</sup> ed.*, by Alex Martelli, Anna Martelli Ravenscroft, and David Ascher (O'Reilly Media, 2005) 0-596-00797-3”。如果你觉得使用代码的情况可能符合需要获取许可的条件，请通过 [permissions@oreilly.com](mailto:permissions@oreilly.com) 与我们联系。

## 本书的读者

我们希望你至少能够懂一点 Python。本书并不是一本循序渐进教授 Python 语言的书。实际上，它针对某些特定的任务和问题，提出一些特别的技术和概念（有时是花招）来应对。如果想找一本完整的介绍 Python 语言的书，可以考虑前言中“参考资料”部分里面提到的书。不过也不需要完全了解 Python 就能从本书获益。一些章节展示了完成一些基本和通用任务的最佳技术，而另一些章节则展示了更加复杂和特殊的技术。我们在书中加了很多边栏来提示一些书中提到的概念，以便将它们和你听到的一些概念区分开来，但你也许仍会感到混淆。所以，这绝对不是一本给初学者看的书。本书针对的最大的读者群应该是 Python 社区的人，他们中有很多很棒的程序员，他们既不是刚上手的新手，也不是 Python 用得出神入化的高手。如果你已经对 Python 有了很多了解，本书仍可能给你一个惊喜，因为我们也把一些最新的和不为人所知的领域的高级技巧收进了本书。你会学到一些新的东西，就像我们一样。无论你在 Python 技术方面处于哪个级别，我们都相信你能从本书获得一些有价值的东西。

如果你已经有了本书的第 1 版，可能会问，我还需要购买第 2 版吗？我们认为，答案是“是的”。第 1 版有 245 节；我们只保留了其中的 146 个（而且绝大多数都重新编辑过），并增加了 192 个新的，这样第 2 版中总共有 338 节。所以，本书中有超过半数内

容是全新的，而且所有的例子都被更新和修改过，以便适用于 Python 2.3 和 2.4。这也是本书比第 1 版多了约 100 节却仍能够保持页数大致相同的主要原因。第 1 版覆盖的 Python 版本包括了 1.5.2（相当的老的版本）到 2.2；本书则集中于 2.3 和 2.4。这应当归功于如今 Python 的强大表现力，而且我们也避免和删除了一些为了兼容 5 年甚至更久之前的 Python 的“历史包袱”般的内容，因此我们能够用大致相同的篇幅，提供更多有价值和更实用的内容。

## 组织方式

本书有 20 章。每章都集中提供某一特定类型的技巧和方法，比如有关算法的、文本处理的以及数据库操作的等。第 1 版有 17 章。不过 Python 也在不断进步，无论是语言本身还是相关的库都在不断变化，另外 Python 社区发表在 *cookbook* 在线的文章也在不停地更新和增加，所以我们加入了全新的 3 个章节：在 Python 2.3 中引入的迭代器 (iterator) 和生成器 (generator)；关于 Python 对时间和财务的操作，新旧并呈；Python 2.2 中导入了新的工具和类型（自定义描述符 (custom descriptor)，装饰器 (decorator)，元类 (metaclass)）。每章包括一个总体介绍，由这方面的一个专家执笔，然后是从在线 *cookbook* 中撷取的技巧和方法（其中有大约 5% 的内容是专为本书撰写的），并被编辑和修改，以符合全书的整体格式与风格。Alex（在 Anna 的帮助下）主要负责对第 1 版的内容进行筛选，确定哪些需要保留，哪些需要更新，并从网站的近 1000 个材料中挑选新的内容，再进行合并或修改（所以，如果你发现在线 *cookbook* 提交的内容和印刷出来的版本不一样，找 Alex 去吧，一定是这家伙干的）。他也需要确定涵盖哪些主题，但有时他可能难以找到符合主题的好材料，所以某些内容并没有被选入，或者最后选用了一些不是从在线 *cookbook* 中找来的材料，这也不完全是他的错。

当材料的选择工作完成之后，就进入了编辑及合并的工作，有时还需要把在线 *cookbook* 上的一些重要的相关评论融合到内容中。这的确是个挑战，就像此书的第 1 版一样，甚至更具挑战性。那些技巧和方法涵盖范围极广，复杂度、完整度和组织方式也相应变化。此书涉及了超过了 300 名作者，他们每个人的声音和风格都包含在书中。我们必须设法让本书能够容纳多种风格，以反映出本书最本质的特点——由整个 Python 社区集体完成。当然，我们也必须小心翼翼地进行编辑工作，以确保内容尽可能地易于使用和查阅，同时还要维持着全书统一的结构和呈现方式。绝大多数材料，包括本书第 1 版和在线 *cookbook* 中的材料，都必须先根据当前情况更新，有的甚至还要重写。不过，通过借助新工具和更好的方法，这个工作完成得很顺利。然后，我们要仔细考虑各章节的排列顺序、每个章节中的布局以及相关技巧和方法的排列顺序。我们做这种排序的工作主要是为了让本书更加易于使用，无论是对刚上手的新人还是 Python 老手都是如此，同时，对于各种不同的阅读习惯，无论是按照逐字逐页的顺序阅读本书，还是为了重点加强某个领域的知识而跳跃性地阅读，甚至是漫无目的地随机阅读，我们都希望我们的排序和组织方式能够对读者有所帮助。

虽然本书很适于“跳跃性”阅读，我们仍然相信，花一些时间循序渐进地一览全书，会为你节约更多的时间。在这次初览中，可以跳过那些你觉得难的，或者不感兴趣的章节。虽然你跳过了一些内容，但是仍可以粗粗地领略到本书的构建方式以及各个主题是如何覆盖和解释的，这将为你以后更深层次的精读打好基础，如果愿意，随机的跳跃性的阅读当然也没问题。为了帮助你更好地了解本书的内容，我们给出了各个章节的主要内容介绍，以及为各个章节写下引言的 Python 专家的介绍。

第 1 章，引言由 Fred L. Drake, Jr 撰写。

本章包含了操纵文本的一系列方法和窍门，包括了合并、过滤、格式化字符串以及在整个文本中对部分字符串的替换，还有 Unicode 的处理。

Fred Drake 是 PythonLabs group 的成员，一直致力于 Python 的开发。他是 3 个孩子的父亲，Fred 在 Python 社区很有名气，因为他一人独立完成了整个官方文档的维护。Fred 还是 *Python & XML* 一书的合著者。

第 2 章，引言由 Mark Lutz 撰写。

本章呈现了操作文件中的数据以及操纵文件系统下的文件和目录的一些技术，还包括了处理特定的文件格式以及存档格式的内容，比如 tar 和 zip 格式。

Mark Lutz 以著作多而闻名，他的书包括了 *Programming Python*、*Python Pocket Reference* 以及 *Learning Python*（都由 O'Reilly 出版），最后一本由他和 David Ascher 合作完成。Mark 同时还是一流的 Python 培训教师，为 Python 在全世界的传播和流行尽心尽力。

第 3 章，引言由 Gustavo Niemeyer 和 Facundo Batista 撰写。

本章（第 2 版中的新章节）介绍了处理日期、时间、十进制数以及一些和财务相关的问题的工具和技术。

Gustavo Niemeyer 是第三方模块 dateutil 的作者，他参与了很多 Python 的扩展和项目的开发。Gustavo 住在巴西。Facundo Batista 是 Decimal PEP 327 的作者，同时也是标准库模块 decimal 的作者，由于这个模块，Python 2.4 才支持浮点十进制计算。他住在阿根廷。作为本书的编辑，我们非常高兴能够请到他们俩为本章撰写引言。

第 4 章，引言由 David Ascher 撰写。

本章包括了一些很通用的，可以应用到各处的技术，这些内容难于归结到某个类别。

David Ascher 是本书的编辑之一。David 的背景颇为斑驳，他做过物理研究、视觉研究、科学可视化、计算机图形学，还摆弄过一堆编程语言，他还是 *Learning Python* 一书的合著者，也做过 Python 的教学工作，最近，他又接手了一些非技术性的任务，比如管理 ActiveState 的团队。David 还定期地组织 Python 会议。

第 5 章，引言由 Tim Peters 撰写。

本章覆盖了 Python 中的搜索和排序技术。很多例子展示了将稳定快速的 `list.sort` 和 `decorate-sort-undecorate` (DSU) (在 Python 2.4 中新导入的能力) 结合在一起的创造性应用，其余例子还展示了 `heapq`、`bisect` 的威力，并介绍了 Python 中其他的搜索和排序工具。

Tim Peters 以 tim-bot 的诨号为人所知，他是 Python 界中的一个传奇性的人物。他犹如一个导师或先知，在 Guido van Rossum 抽不开身的时候，他能够化身为另一个 Guido van Rossum 来指引方向；当有人略微提及有关 IEEE 标准的问题时，他又能够化身为 IEEE-754 浮点标准委员会，引经据典，滔滔不绝；当有人试图鼓吹对 Python 进行某些激进的改革时，他还能够适时地表现出中流砥柱的一面，稳健而坚定。Tim 也是 PythonLabs 团队中的一员。

第 6 章，引言由 Alex Martelli 撰写。

本章的内容展示了在 Python 中应用面向对象编程模式的威力，包括了一些很重要的技术，比如通过一些特殊方法来托管和控制属性，用一些中级的技术来实现不同的设计模式，还有一些对于高级概念的简单应用，比如自定义元类，关于元类的有关内容在第 20 章会有更深入的探讨。

Alex Martelli，外号 martelli-bot，是本书的编辑之一。他在 IBM 研发中心工作过约十年，然后又在 think3, inc. 工作了更久的时间。Alex 现在是自由职业的顾问，最近在为 AB Strakt 工作，那是个以 Python 为业务中心的瑞典公司。他有时也编辑 Python 文章或者著书，比如 *Python in a Nutshell*，偶尔，他还会研究一下合约桥牌。

第 7 章，引言由 Aaron Watters 撰写。

本章着重介绍了 Python 的持久化技术，包括序列化的方法以及和不同的数据库交互的方式。

Aaron Watters 是最早的 Python 倡导者之一，也是数据库专家。他作为第一本 Python 书 (*Internet Programming with Python*, M&T Books, 现在已经绝版) 的主要作者而闻名遐迩，他也是很多广为流传的 Python 扩展的作者，比如 `kjBuckets` 和 `kwParsing`。Aaron 目前也是自由职业的顾问。

第 8 章，引言由 Mark Hammond 撰写。

本章介绍了一系列有关调试和测试的方法和窍门，包括了可定制错误日志和回溯信息，以及使用可定制模块 (`unittest` 和 `doctest`) 进行单元测试。

为了使 Windows 平台支持 Python，Mark Hammond 做了很多工作并因此而出名。他和 Greg Stein 创建了一个包含很多模块的强大的库，这个库将 Python 完全地映射到了 Windows 的一系列 API、库甚至组件模型，如 COM。他是个专家级的设计者，还是很

多开发工具的作者，其中最为人所知的工具就是 Pythonwin 和 Komodo。最后，Mark 还是调试高手，在 Komodo 开发中，常常面对各种棘手的状况，他有时也被邀请去调试涉及三种语言（Python、C++和 Javascript）的、多线程、多进程的系统。Mark 和 Andy Robinson 合著了 *Python Programming on Win32* 一书。

第 9 章，引言由 Greg Wilson 撰写。

本章涵盖了一系列同步编程技术，包括了线程、队列以及多进程。

Greg Wilson 写过儿童书籍，也写过并行编程和数据处理的书。当他不写书的时候，他是 Doctor Dobb's Journal 的得力编辑，多伦多大学计算科学系的助理教授，同时还是个自由职业的软件开发者。Greg 是 Software Carpentry 项目的最初发起者和推动者，最近他又得到 Python 软件基金会的认可，为需要进行科学计算的科学家和工程师开发一系列具有 Python 特色的课程材料。

第 10 章，引言由 Donn Cave 撰写。

本章介绍了利用 Python 进行一些通用的系统管理任务的内容，包括生成密码和操作 Windows 的注册表，以及处理信箱和 Web 服务器的问题。

Donn Cave 是华盛顿大学计算中心的软件工程师。多年以来，Donn 已经在 comp.lang.python 充分证明了他在有关系统调用、UNIX、系统管理、文件、信号等方面丰富的知识和技能。

第 11 章，引言由 Fredrik Lundh 撰写。

本章包括了一些通用的界面任务，主要使用 Tkinter，但也涉及了一些 wxPython、Qt、图像处理，以及和 Jython（用于 JVMJava 虚拟机）相关的一些特定的界面处理，还有 Mac OS X 和 IronPython（用于 dotNet）。

Fredrik Lundh 也以 eff-bot 之名著称，他是 Secret Labs AB 的 CTO，该公司主要提供一些基于 Python 的产品和技术。Fredrik 是 Tkinter（Python 最流行的界面工具包）领域的一流专家，也是 Python 图形库（Python Image Library, PIL）的主要作者。他还是 *Python Standard Library* 一书的作者，该书对 Python 的标准库进行了全面的介绍，可以说是本书非常有益的补充。他在 comp.lang.python 发文极多，极大地帮助了广大的 Python 新手，甚至很多老手和专家也从他的文章中获益良多。

第 12 章，引言由 Paul Prescod 撰写。

本章介绍了如何使用一系列 Python 工具来分析、处理以及生成 XML。

Paul Rescod 在三种技术方面完全可以称得上是专家：Python，在这个方面他已经无须向世人来证明什么了；XML，他在 XML 的应用上的功力也毋庸置疑（Paul 和 Charles Goldfarb 合作，完成了 *XML handbook* 一书）；Unicode，不像前面两种技术，Unicode 总是让广大程序员痛苦而迷惘。Paul 目前是 Blast Radius 的产品经理。

第 13 章，引言由 Guido van Rossum 撰写。

本章包括了各种网络编程技术，从基本的 TCP 客户端和服务器，一直到操纵 MIME 消息的技术。

Guido 创造了 Python，并在 Python 的整个幼儿期精心照顾它，现在 Python 的发展方向还完全由他控制着。对于这位 Python 之父，我们实在没有什么可说的了。

第 14 章，引言由 Andy McKay 撰写。

本章主要介绍了 Web 相关的技术，包括 CGI 脚本、在 Jython 中运行 Java servlet 以及访问 web 页面的内容。

Andy McKay 是 Enfold Systems 的副总裁和创始人之一。过去几年来，Andy 从快乐的 Perl 用户转变成了狂热的 Python、Zope 以及 Plone 专家。他写了 *Definitive Guide to Plone* 一书，并且还运营着一个热门的 Zope 讨论网站，<http://www.zopezen.org>。

第 15 章，引言由 Jeremy Hylton 撰写。

本章介绍了将 Python 用于简单的分布式系统的一些技术，包括了 XML-RPC、CORBA 以及 Twisted 的 Perspective Broker。

Jeremy Hylton 在 Google 工作。除了他年幼的双胞胎孩子，他还喜欢计算机编程理论、解析器等。作为他为 CNRI 工作的一部分，Jeremy 在许多分布式系统上做过工作。

第 16 章，引言由 Paul F. Dubois 撰写。

本章涉及了程序内省、currying 机制、动态导入以及程序发布、词法分析和解析。

Paul Dubois 在 Lawrence Livermore 国家实验室工作过多年，从天气模型到原子模拟器，他为那里的科学家开发过各种各样的软件系统。他在有关科学计算方面有着无与伦比的经验，同时还精于程序设计和高级的面向对象编程技术。

第 17 章，引言由 David Beazley 撰写。

本章提供了一些技巧和方法，可以帮助读者在 Python 扩展的开发中事半功倍。

David Beazley 最主要的名声来自于 SWIG，这是一个精巧强大的工具，可以快速地将 C 程序和相关的库打包，再由 Python、Tcl、Perl 等其他语言来调用。这个看似语言无关的中立的工具实际上是以支持 Python 为第一要务的，关于此点，在他的著作 *Python Essential Reference* 中可见端倪。David Beazley 是相当变态（这里没有贬义）的家伙，他总是能让我们相信他的脑子里随时会产生出更多惊人而强大的工具。他现在正在芝加哥大学向计算机科学系的学生灌输他的幽默感。

第 18 章，引言由 Tim Peters 撰写。

本章提供了一些用 Python 实现的、优美而实用的算法和数据结构。

Tim Peters 的相关信息见前面第 5 章内容介绍部分。

第 19 章，引言由 Raymond Hettinger 撰写。

本章（第 2 版中新加人的章节）介绍和展示了迭代器和生成器的强大威力，它们让你的循环结构变得更简单、更快速，同时也更具有复用性。

Raymond Hettinger 是 `itertools` 包的开发者，也是生成器表达式方案的提出者。他已经成为了 Python 发展中重要的贡献者。如果你不知道是谁构思和实现了 Python 2.3 和 2.4 中的一些新颖而重要的优化，我们建议你打赌 Raymond 就是那个家伙。

第 20 章，引言由 Raymond Hettinger 撰写。

本章（第 2 版中新加人的章节）从深层次审视了 Python 语言的一些基础构架和元素，正是这些部分使得 Python 的面向对象编程技术更加强大而平滑。无论是为了找乐子还是应用于实际，你都可以自行发掘和定制这些元素。从创建属性的惯用法到重命名和缓存属性，到装饰器（decorator），再到一个自定义元类工厂（a factory of custom metaclasses），这些方法允许你操纵字节码从而优化程序功能，还允许你避免元类型冲突。本章表明，即使这些部分看起来很复杂、很吓人，只要你静下心来认真研究，你的 Python 应用能力会获得极大的提高。

Raymond Hettinger 的信息见前面第 19 章内容介绍部分。

## 参考资料

从一般的介绍性文章到非常正式的语言描述文档，有很多文本内容可以帮助你学习 Python 或者帮助你加深对某些技术的理解。

对于一般学习而言，我们推荐下列书目（所有的这些书都至少涵盖了 Python 2.2，除非特别注明）：

- *Python Programming for the Absolute Beginner*, Michael Dawson 著 (Thomson Course Technology)，对于从未编写过程序的人而言，这是一本非常实用的、简洁易读的 Python 介绍性读物。
- *Learning Python*, Mark Lutz, David Ascher 著 (O'Reilly)，这是一本非常全面的关于 Python 基础知识的介绍性读物。
- *Practical Python*, Magnus Lie Hetland 著 (APress)，此书介绍了 Python，同时也在书中开发了很多实用而重要的程序，细节描述多，完成度也高。
- *Dive into Python*, Mark Pilgrim 著 (APress)，这是一本针对有经验的程序员的快速入门读物，也可以免费在线阅读或者下载 (<http://diveintopython.org>)。
- *Python Standard Library*, Fredrik Lundh 著 (O'Reilly)，此书为 Python 发行版的庞大的标准库中的每个模块提供了一个有用实例(目前此书的第 1 版只支持到 Python 2.0)。

- *Programming Python*, Mark Lutz 著 (O'Reilly), 这是一本有关 Python 编程技术的全面的纲要性读物 (目前此书第 2 版只支持到 Python 2.0)。
- *Python Essential Reference*, David Beazley 著 (New Riders), 这是一本针对 Python 语言本身和核心的 Python 库的快速查阅参考书 (目前此书第 2 版只支持到 Python 2.1)。
- *Python in a Nutshell*, Alex Martelli 著 (O'Reilly), 这是一本广为使用的、可以快速查阅 Python 语言和一些关键库的参考书。

另外，其他一些具有特殊用途的书也可以帮助你了解 Python 编程在特定领域的应用。至于是否喜欢那些书，则要看你感兴趣的领域是什么。根据个人经验，编辑推荐下列书目：

- *Python and XML*, Christopher A. Jones, Fred L. Drake, Jr. 著 (O'Reilly), 介绍了关于 Python 读取、处理和转换 XML 的各种技术。
- *Jython Essential*, Samuele Pedroni, Noel Rappin 著 (O'Reilly), 这是一本关于 Jython 的权威书籍，也是 Python 到 JVM 的一个桥梁。如果读者已经懂一些 Java，此书的用处就更大了。
- *Game Programming with Python*, Sean Riley 著 (Charles River Media), 此书涵盖了用 Python 进行游戏编程的方方面面，包括从高级图形操作到不错的人工智能。
- *Python Web Programming*, Steve Holden 著 (New Riders), 涵盖了用 Python 构建网络系统的各种技术，并介绍了其他一些相关内容 (数据库、HTTP、HTML 等)。对于在这些领域没有经验或者经验很少的读者来说，这是一本很实用的书。

如果你在阅读本书时遇到问题，除了上面这些书，还有一些重要的资源可以帮助你找到答案。我们将在每个小节的“更多资料”的条目下给出一些信息和指引。通常，我们可以参考和查阅标准的 Python 文档：最常用的是“Library Reference”，有时查看“Reference Manual”，偶尔也会看看“Tutorial”。这些文档都可以以多种形式免费获得。

- python.org 网站 (<http://www.python.org/doc/>)，总是提供最新的关于 Python 的文档。
- pydoc.org 网站 (<http://pydoc.org/>)，提供标准库中每个模块的文档，这些文档由 pydoc 这个强大工具自动生成。
- Python 本身。最近的 Python 版本提供了一个漂亮的在线帮助系统，如果没用过可以去试试看。只要在交互的 Python 解释器提示符下键入 `help()`，就可以启动帮助系统。
- 你的 Python 安装版本中所含的部分在线文档。ActivePython 的安装程序，包括了一个可以搜索的 Windows 帮助文件。虽然当前标准的 Python 发布包只是提供

HTML 页面，但是已经有计划要在将来的版本中提供一个类似的 Windows 帮助文件。

在引用 Python 的标准文档时，我们没有加入章节数字来进行标记，因为这些文档总是随着版本的变化而变化的。可以通过查看内容目录和索引来找到相关材料。对于“Library Reference”而言，模块索引（所有标准库模块的字母表顺序列表，每个模块名都是一个可点击的超链接，指向了“Library Reference”文档中对该模块的介绍）非常有用。同样地，我们在引用 *Python in a Nutshell* 的时候，也没有给出确切的位置：在本书编写过程中，那本书还只是第 1 版（仅仅覆盖到 Python 2.2），当你开始阅读本书的时候，第 2 版的 *Python in a Nutshell*（覆盖 Python 2.3 和 2.4）如果还没有出版，也应该近在咫尺了。

## 阅读须知

**宣告** 书中的第一人称单数，是指该材料或者章节引言的执笔人（当一段材料的致谢针对多个人的时候，作者被放在首位）。由于有很多评论的内容也被融入文中，因此原作的某些意思可能难以充分地表达出来（作为编辑，我们尽量避免这种状况的发生，但我们知道一定会有疏漏的地方，因为评论的内容非常多，而原作的意思有时也不是很清晰）。而第二人称则是代表你，本书的读者。第一人称的复数也是指你，本书的读者，但还包括了本篇材料的作者和合作者、本书的编辑还有我的朋友 Joe (Hi Joe!)，总而言之，那个词的包容性非常强。

**代码** 每段代码都代表着一段完整的脚本或者一个模块（通常，Python 的源码文件可以被当作脚本或者作为模块来复用），或者从某个假设的模块或脚本中抽取的片段，或者只是从 Python 的交互解释器中提取的部分（通常会带有一个>>>提示符）。

## 如何与我们联系

我们已经尽力验证和检查全书，但你仍有可能发现某些语言特征已经发生变化了，或者你可能会发现我们忽略的一些错误。如果是这样，请与我们联系，告诉我们你发现的错误，当然我们也欢迎好的建议，来信请寄：

**美国：**

O'Reilly Media, Inc.  
1005 Gravenstein Highway North  
Sebastopol, CA 95472

**中国：**

100080 北京市西城区西直门成铭大厦 C 座 807 室  
奥莱利技术咨询（北京）有限公司

我们为本书建立了一个网站，列出了相关示例、勘误表以及本书下一版的计划。可以通过下面的网址访问：

<http://www.oreilly.com/catalog/pythoncook2>

关于此书的任何问题或者评论，请发信到：

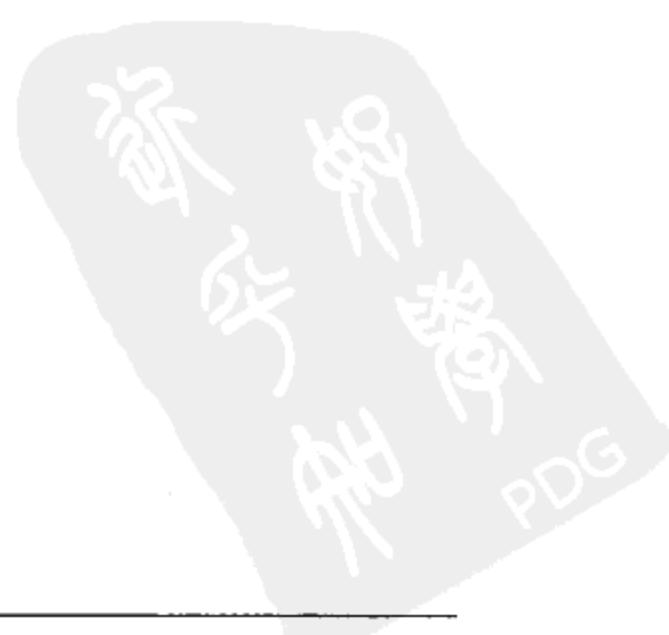
[bookquestions@oreilly.com](mailto:bookquestions@oreilly.com)

关于我们的出版物的更多信息、参考、资源中心以及 O'Reilly Network，请访问：

<http://www.oreilly.com/>

提供本书绝大部分素材的在线 cookbook 可通过下面网址访问：

<http://aspn.activestate.com/ASP/N/Cookbook/Python>



---

# 致谢

绝大多数出版物，从玄幻文学到科学论文，再到计算机书籍，都会宣称，若非由于某些人的合作和工作，该书没有可能顺利出版，这些人往往是同僚或者孩子。本书亦是如此。书中绝大多数的文字、代码和点子都是由一些名字并未出现在封面上的人提供的。那些技巧和方法的原作者，从在线 cookbook 提交了各种有用的和独特的见解的读者，以及撰写各章引言的专家，他们才是本书的真正作者，他们当然是最值得感谢的人。

## David Ascher

Andy McKay 辛勤和杰出的工作保证了在线 cookbook 正常运行。从开始的数据搜集阶段，Andy 就是 ActiveState 的主力 Zope 开发人员，也是 ASPN (<http://aspn.activestate.com>) 的关键开发人员，该网站为各种开源程序语言（比如 Python、Perl、PHP、Tcl 以及 XSLT 等）的程序员提供了大量的信息。顺便提一句，Andy McKay 过去是个 Perl 程序员。那时，我才刚到 ActiveState，公司决定使用 Zope 来构建 ASPN。随后的几年，Andy 变成了 Zope 高手和狂热的 Python 用户（那可不是我鼓动的），接着他又为 Zope 和 Plone 写书，然后成为了顾问并开办了自己的公司。根据我和同样来自于 ActiveState 的 Diane Mueller 的原始设计，Andy 一人独力实现了整个 ASPN，然后他又根据环境和情况的变化不停地修正和加强系统，很多改进都是我们当初的设计没有覆盖和考虑到的地方，而且在整个开发过程中，他始终保持着专注，并表现出卓越的专业能力。我很高兴能够请他为本书的关于 Web 的章节撰写引言。自从 Andy 离去之后，Jams McGill 接手了他的工作，维护着在线 cookbook 并保证它全天 24 小时的正常工作，为全世界的 Python 迷们服务。

Paul Prescod，同样来自 ActiveState，他的影响在整个项目中无处不在，他帮助处理在线编辑流程，提出各种建议，鼓励 comp.lang.python 的用户去访问在线 cookbook 并提交内容和给出建议。当我们试图从 Zope 中抽取数据用于出版流程时，Paul 又凭借他在 XML 方面的丰富知识，帮了我们一个大忙。

最后一个我要感谢的是 Dick Hardt，ActiveState 的 CEO 和创始者，主要有两个原因。首先，Dick 同意让我加入 cookbook 的工作中，并作为我的工作职责的一部分。如果不

是这样，我根本没机会参与此事。其次，Dick 建议将本书的部分版税捐赠给 Python 软件基金会。这个决定不仅鼓励了更多的 Python 用户成为投稿人，同时也为这个组织带来了一项长期的收益，这个组织需要支持，也值得支持，因为所有的 Python 用户都将从中受益。

重写一个软件系统是危险的：在工程上，“第 2 版系统”综合症意味着，为了重建一个“好”的系统，常常要经受冗长的周期和过度的工程设计的煎熬。我很高兴本书的第 2 版是个例外，主要有两个原因。第一，此书的覆盖范围缩小并集中到更新的 Python 版本上，这使得本书的内容更加具有可控性，而且读者也更加喜欢。第二，所有人都知道我不可能有时间完成第 2 版的所有编辑工作。所以我很高兴我能作为成员之一参与到本书的编辑工作中，我非常欣赏由其他人完成的部分。当像 Alex 和 Anna 这样的人表示愿意参与到编辑工作中时，我真的感到很走运很放心。

最后我还要感谢 O'Reilly 的编辑，他们为本书的最后定型出力甚多。Laura Lewin 是本书第 1 版的编辑，为了保证项目的顺利进行，她核对和调整作者的稿子。然后 Paula Ferguson 接过接力棒，他提供了大量的珍贵的反馈，编辑最终的手稿，并尽可能地让手稿能够保持原貌，以便让读者能够体会到文中的各种不同风格。Jonathon Gennick 也是本书第 2 版的编辑，他在 Alex 和 Anna 的带领下，完成了大量的琐碎工作。还有一个我要提及的是 Tim O'Reilly，他对此书非常投入，在此书的初期阶段，大量的手工输入工作由他完成。

每当我读到致谢这一节，我都忍不住会想起 O'Reilly 的主编 Frank Willison。Frank 在 2001 年 7 月 30 日突然逝世。他是最想看到本书出版的人，他认为 Python 社区应该有这样一本好书。Frank 总是提出新点子，对批评和错误也非常虚心大度。像这种有超过 100 个作者的书会吓倒绝大多数编辑，可是 Frank 却把它看成是一个挑战和经历。我一直很怀念他。

### Alex Martelli

我和 Python 的初次接触也许应该归功于我那个有点倔的前同事 Alessandro Bottoni。他很有礼貌地一次又一次地建议我去试试 Python，虽然我已经告诉他无数次，我懂的编程语言已经够多了，可是我还是不知道用这些语言来做什么。如果我当初没有相信他的技术眼光和他的热忱，我今天可能不会有有机会编写任何一本 Python 书籍。谢谢你恰到好处的固执，Alessandro！

一旦我开始尝试 Python，我就无可救药地痴迷于这种综合了各种语言优点的高级语言。在这里，它拥有 Rexx 的简易的句法、Tcl 的简洁的语义、Scheme（还有其他一些 Lisp 的变种）的智能化以及 Perl 的强大能力。我怎么能抵御这种诱惑？首先，我得感谢 Mike Cowlishaw（Rexx 的发明者），他是我在 IBM 研发中心时的同事，他首先让我对脚本语言产生了兴趣。我还得感谢 John Ousterhout 和 Larry Wall，Tcl 和 Perl 的发明者，他们两人发明的语言让我对脚本的瘾越来越大。

Greg Wilson 首先把我介绍给 O'Reilly，我当然也得感谢他。我也很高兴把他拉来为此书撰写引言。我很感谢 David Ascher 和 O'Reilly 的几个人，他们和我签约并让我成为了本书第 1 版的编辑之一，而且他们非常支持我出第 2 版的想法（现在回想起来，我怀疑我根本忘记了当初在做第 1 版的时候工作是多么辛苦，而且我还没有意识到，ActiveState 网站上的新内容已经堆积如山，经历 3 年的发展 Python 也已经日新月异，第 2 版的工作量肯定远超第 1 版……）。

如果没有一系列强大的技术工具在后面支撑，我是无法完成这个工作的。我不知道很多我应该感谢的人的名字，我想我应该感谢 Internet、ADSL 和 Google 搜索引擎，它们使得我可以轻易地获取各种资料，在各种软件硬件的协同下，极大地放大了我的创造力。我可以很肯定，如果没有 Theo de Raadt 的 OpenBSD 操作系统，Steve Jobs 的 Mac OS X 系统和 iBook G4（我在那上面完成了大部分工作），Bram Moolenaar 的 VIM 编辑器，当然，还有 Guido van Rossum 的 Python 语言，我肯定无法完成这个任务。所以我要单独对 Theo、Steve、Bram 以及 Guido 提出特别感谢。

我不希望像很多书的作者一样，宣称书的完成要感谢朋友和家人的耐心支持，比如我的孩子 Lucio 和 Flavia、我的姐姐 Elisabetta、我的父亲 Lanfranco，等等。但其实只有一个人是真正不可或缺的，那就是我的妻子 Anna，她也是本书的编辑之一。当初，由于 Python 我和她才重新联系起来（我们曾经分开多年），后来又在一次开源软件会议上度过我们的蜜月，我们还联合表演了一次闪电秀，谈论我们的 Python 风格的婚礼。我曾设想，如果能够和她一起为这个复杂庞大的项目日日夜夜并肩工作，该是多么棒的一件事。实际也确实如此，本书的形成，伴随了无数次白热化的辩论，涉及了各种技术点或组织点的选择，甚至遣词造句。这整个过程中，我们经受住了压力，也付出了努力，而她的技能、她的爱、她的喜悦，每次都能照亮我、支持我，并不断地给我提供前进的动力。谢谢你，Anna！

#### **Anna Martelli Ravenscroft**

我大约两年前才发现 Python，然后我就爱上了 Python 这门语言，还同时爱上了有名的 martelli-bot。Python 是一种贴心的语言，因为它非常易于上手和使用。你不需要为了使用这门语言先去隐居修行 4 年。感谢 Guido，也感谢 Python 社区营造的一种对新用户非常友善的气氛。

我在这本书上的所做的工作，实际上对我而言也是个学习的过程。除了所有这些 Python 代码，我还学习了 XML 和 VI，同时也重新认识了 Subversion。感谢 Holger Krekel 和 codespeak，他们在我们旅行的时候为我们架设了 subversion 服务器。这也为我们带来了一群我们需要感谢的人：我们的半成品的读者。Holger Krekel 接着又再一次帮助了我们，确保了我们的文档对 Unicode 的支持。Raymond Hettinger 给了我们很多有价值的意见，主要是关于迭代器和生成器的内容。Raymond 和 Holger 总是能够对当前的“解决方案”提出另一种可参考的思路。Valentino Volonghi 在读完初稿之后又给出了很多

关于程序风格和排版方面的建议。Ryan Alexander，作为一个有着 Java 背景的 Python 新人，给出了很多关于对章节和材料排序的建议，并指出某些部分的解释不够清楚或者缺失。他的建议让本书变得对 Python 新手更加友好和易用。还有很多人对特定的章节和材料给出了反馈，但人数太多，此处难以一一列出。我们非常感谢你们对于我们工作的支持。

当然，我还要感谢我的丈夫。我对于 Alex 的耐心非常地吃惊（我总是提出大量的问题）。他精益求精的精神使他成为了一个不可多得的编辑。每当我们遇到了反馈，他总是会迅速地做出反应，向反馈者道谢，并且考虑如何应用这些建议以使得此书更加完美。他是我见到的最无私的人之一。

我还要感谢 Dan，在他的鼓励下我才开始使用 Linux，他还教了我一些有关的 Linux 术语，并让我在 Internet 上流连忘返。最后，我得特别感谢我的孩子 Inanna 和 Graeme。在此书的最后完成阶段，当我整天处于一种 geek 模式的时候，他们给了我拥抱、理解和支持。你们是一个母亲希望拥有的最好的孩子。

# 计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

**Java** 一览无余: **Java** 视频教程 | **Java SE** | **Java EE**

**.Net** 技术精品资料下载汇总: **ASP.NET** 篇

**.Net** 技术精品资料下载汇总: **C#语言** 篇

**.Net** 技术精品资料下载汇总: **VB.NET** 篇

撼世出击: **C/C++** 编程语言学习资料尽收眼底 电子书+视频教程

**Visual C++(VC/MFC)** 学习电子书及开发工具下载

**Perl/CGI** 脚本语言编程学习资源下载地址大全

**Python** 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总: **MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

**UML** 学习电子资下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网: 精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

**Linux** 系统管理员必备参考资料下载汇总

**Linux shell**、内核及系统编程精品资料下载汇总

**UNIX** 操作系统精品学习资料<电子书+视频>分类总汇

**FreeBSD/OpenBSD/NetBSD** 精品学习资源索引 含书籍+视频

**Solaris/OpenSolaris** 电子书、视频等精华资料下载索引

# 目录

<b>第 1 章 文本 .....</b>	<b>1</b>
引言 .....	1
1.1 每次处理一个字符 .....	6
1.2 字符和字符值之间的转换 .....	7
1.3 测试一个对象是否是类字符串 .....	8
1.4 字符串对齐 .....	10
1.5 去除字符串两端的空格 .....	11
1.6 合并字符串 .....	11
1.7 将字符串逐字符或逐词反转 .....	14
1.8 检查字符串中是否包含某字符集合中的字符 .....	15
1.9 简化字符串的 translate 方法的使用 .....	18
1.10 过滤字符串中不属于指定集合的字符 .....	20
1.11 检查一个字符串是文本还是二进制 .....	23
1.12 控制大小写 .....	25
1.13 访问子字符串 .....	26
1.14 改变多行文本字符串的缩进 .....	29
1.15 扩展和压缩制表符 .....	31
1.16 替换字符串中的子串 .....	33
1.17 替换字符串中的子串——Python 2.4 .....	34
1.18 一次完成多个替换 .....	36
1.19 检查字符串中的结束标记 .....	39
1.20 使用 Unicode 来处理国际化文本 .....	40
1.21 在 Unicode 和普通字符串之间转换 .....	43
1.22 在标准输出中打印 Unicode 字符 .....	45
1.23 对 Unicode 数据编码并用于 XML 和 HTML .....	46
1.24 让某些字符串大小写不敏感 .....	49
1.25 将 HTML 文档转化为文本显示到 UNIX 终端上 .....	52
<b>第 2 章 文件 .....</b>	<b>55</b>
引言 .....	55
2.1 读取文件 .....	59

2.2	写入文件	62
2.3	搜索和替换文件中的文本	64
2.4	从文件中读取指定的行	65
2.5	计算文件的行数	66
2.6	处理文件中的每个词	68
2.7	随机输入/输出	70
2.8	更新随机存取文件	71
2.9	从 zip 文件中读取数据	73
2.10	处理字符串中的 zip 文件	74
2.11	将文件树归档到一个压缩的 tar 文件	76
2.12	将二进制数据发送到 Windows 的标准输出	77
2.13	使用 C++ 的类 <code>iostream</code> 语法	78
2.14	回退输入文件到起点	80
2.15	用类文件对象适配真实文件对象	83
2.16	遍历目录树	84
2.17	在目录树中改变文件扩展名	85
2.18	从指定的搜索路径寻找文件	86
2.19	根据指定的搜索路径和模式寻找文件	87
2.20	在 Python 的搜索路径中寻找文件	88
2.21	动态地改变 Python 搜索路径	89
2.22	计算目录间的相对路径	91
2.23	跨平台地读取无缓存的字符	93
2.24	在 Mac OS X 平台上统计 PDF 文档的页数	94
2.25	在 Windows 平台上修改文件属性	95
2.26	从 OpenOffice.org 文档中提取文本	96
2.27	从微软 Word 文档中抽取文本	97
2.28	使用跨平台的文件锁	98
2.29	带版本号的文件名	100
2.30	计算 CRC-64 循环冗余码校验	102
<b>第 3 章 时间和财务计算</b>		105
	引言	105
3.1	计算昨天和明天的日期	111
3.2	寻找上一个星期五	112
3.3	计算日期之间的时段	114
3.4	计算歌曲的总播放时间	115
3.5	计算日期之间的工作日	116

3.6	自动查询节日	118
3.7	日期的模糊查询	121
3.8	检查夏令时是否正在实行	123
3.9	时区转换	124
3.10	反复执行某个命令	125
3.11	定时执行命令	127
3.12	十进制数学计算	129
3.13	将十进制数用于货币处理	130
3.14	用 Python 实现的简单加法器	133
3.15	检查信用卡校验和	136
3.16	查看汇率	137
	<b>第 4 章 Python 技巧</b>	<b>139</b>
	引言	139
4.1	对象拷贝	140
4.2	通过列表推导构建列表	144
4.3	若列表中某元素存在则返回之	146
4.4	循环访问序列中的元素和索引	147
4.5	在无须共享引用的条件下创建列表的列表	148
4.6	展开一个嵌套的序列	149
4.7	在行列表中完成对列的删除和排序	152
4.8	二维阵列变换	154
4.9	从字典中取值	155
4.10	给字典增加一个条目	157
4.11	在无须过多援引的情况下创建字典	158
4.12	将列表元素交替地作为键和值来创建字典	159
4.13	获取字典的一个子集	161
4.14	反转字典	163
4.15	字典的一键多值	164
4.16	用字典分派方法和函数	166
4.17	字典的并集与交集	167
4.18	搜集命名的子项	169
4.19	用一条语句完成赋值和测试	171
4.20	在 Python 中使用 printf	174
4.21	以指定的概率获取元素	174
4.22	在表达式中处理异常	176
4.23	确保名字已经在给定模块中被定义	178

<b>第 5 章 搜索和排序</b>	180
引言	180
5.1 对字典排序	185
5.2 不区分大小写对字符串列表排序	185
5.3 根据对象的属性将对象列表排序	187
5.4 根据对应值将键或索引排序	189
5.5 根据内嵌的数字将字符串排序	192
5.6 以随机顺序处理列表的元素	193
5.7 在增加元素时保持序列的顺序	195
5.8 获取序列中最小的几个元素	197
5.9 在排序完毕的序列中寻找元素	199
5.10 选取序列中最小的第 n 个元素	200
5.11 三行代码的快速排序	203
5.12 检查序列的成员	206
5.13 寻找子序列	208
5.14 给字典类型增加排名功能	210
5.15 根据姓的首字母将人名排序和分组	214
<b>第 6 章 面向对象编程</b>	217
引言	217
6.1 温标的转换	223
6.2 定义常量	225
6.3 限制属性的设置	227
6.4 链式字典查询	229
6.5 继承的替代方案——自动托管	231
6.6 在代理中托管特殊方法	234
6.7 有命名子项的元组	237
6.8 避免属性读写的冗余代码	239
6.9 快速复制对象	240
6.10 保留对被绑定方法的引用且支持垃圾回收	243
6.11 缓存环的实现	245
6.12 检查一个实例的状态变化	249
6.13 检查一个对象是否包含某种必要的属性	252
6.14 实现状态设计模式	255
6.15 实现单例模式	257
6.16 用 Borg 惯用法来避免“单例”模式	259

6.17	Null 对象设计模式的实现 .....	263
6.18	用 <code>__init__</code> 参数自动初始化实例变量 .....	266
6.19	调用超类的 <code>__init__</code> 方法 .....	267
6.20	精确和安全地使用协作的超类调用 .....	270
<b>第 7 章 持久化和数据库 .....</b>		<b>273</b>
	引言 .....	273
7.1	使用 <code>marshal</code> 模块序列化数据 .....	275
7.2	使用 <code>pickle</code> 和 <code>cPickle</code> 模块序列化数据 .....	277
7.3	在 Pickling 的时候压缩 .....	280
7.4	对类和实例使用 <code>cPickle</code> 模块 .....	281
7.5	Pickling 被绑定方法 .....	284
7.6	Pickling 代码对象 .....	286
7.7	通过 <code>shelve</code> 修改对象 .....	288
7.8	使用 Berkeley DB 数据库 .....	291
7.9	访问 MySQL 数据库 .....	294
7.10	在 MySQL 数据库中储存 BLOB .....	295
7.11	在 PostgreSQL 中储存 BLOB .....	296
7.12	在 SQLite 中储存 BLOB .....	298
7.13	生成一个字典将字段名映射为列号 .....	300
7.14	利用 <code>dtuple</code> 实现对查询结果的灵活访问 .....	302
7.15	打印数据库游标的内容 .....	304
7.16	适用于各种 DB API 模块的单参数传递风格 .....	306
7.17	通过 ADO 使用 Microsoft Jet .....	308
7.18	从 Jython Servlet 访问 JDBC 数据库 .....	310
7.19	通过 Jython 和 ODBC 获得 Excel 数据 .....	313
<b>第 8 章 调试和测试 .....</b>		<b>315</b>
	引言 .....	315
8.1	阻止某些条件和循环的执行 .....	316
8.2	在 Linux 上测量内存使用 .....	317
8.3	调试垃圾回收进程 .....	318
8.4	捕获和记录异常 .....	320
8.5	在调试模式中跟踪表达式和注释 .....	322
8.6	从 <code>traceback</code> 中获得更多信息 .....	324
8.7	当未捕获异常发生时自动启用调试器 .....	327
8.8	简单的使用单元测试 .....	328

8.9	自动运行单元测试.....	330
8.10	在 Python 2.4 中使用 doctest 和 unittest.....	331
8.11	在单元测试中检查区间.....	334
<b>第 9 章 进程、线程和同步.....</b>		<b>336</b>
	引言.....	336
9.1	同步对象中的所有方法.....	339
9.2	终止线程.....	342
9.3	将 Queue.Queue 用作优先级队列.....	344
9.4	使用线程池.....	346
9.5	以多组参数并行执行函数.....	349
9.6	用简单的消息传递协调线程.....	351
9.7	储存线程信息.....	353
9.8	无线程的多任务协作.....	357
9.9	在 Windows 中探测另一个脚本实例的运行.....	359
9.10	使用 MsgWaitForMultipleObjects 处理 Windows 消息.....	360
9.11	用 popen 驱动外部进程.....	363
9.12	获取 UNIX Shell 命令的输出流和错误流.....	364
9.13	在 UNIX 中 fork 一个守护进程.....	367
<b>第 10 章 系统管理.....</b>		<b>370</b>
	引言.....	370
10.1	生成随机密码.....	371
10.2	生成易记的伪随机密码.....	372
10.3	以 POP 服务器的方式验证用户.....	375
10.4	统计 Apache 中每个 IP 的点击率.....	376
10.5	统计 Apache 的客户缓存的命中率.....	378
10.6	在脚本中调用编辑器.....	379
10.7	备份文件.....	381
10.8	选择性地复制邮箱文件.....	383
10.9	通过邮箱创建一个邮件地址的白名单.....	384
10.10	阻塞重复邮件.....	386
10.11	检查你的 Windows 声音系统.....	388
10.12	在 Windows 中注册和反注册 DLL.....	388
10.13	检查并修改 Windows 自动运行任务.....	390
10.14	在 Windows 中创建共享.....	391
10.15	连接一个正在运行的 Internet Explorer 实例.....	392

10.16	读取 Microsoft Outlook Contacts .....	393
10.17	在 Mac OS X 中收集详细的系统信息 .....	396

## 第 11 章 用户界面 ..... 400

引言.....	400	
11.1	在文本控制台中显示进度条 .....	402
11.2	避免在编写回调函数时使用 lambda.....	404
11.3	在 tkSimpleDialog 函数中使用默认值和区间 .....	405
11.4	给 Tkinter 列表框增加拖曳排序能力.....	406
11.5	在 Tkinter 部件中输入一个重音字符.....	408
11.6	在 Tkinter 中嵌入内联的 GIF .....	410
11.7	转换图片格式.....	412
11.8	在 Tkinter 中实现一个秒表.....	415
11.9	用线程实现 GUI 和异步 I/O 的结合.....	417
11.10	在 Tkinter 中使用 IDLE 的 Tree 部件 .....	421
11.11	在 Tkinter Listbox 中支持单行多值 .....	423
11.12	在 Tkinter 部件之间复制 Geometry 方法和选项 .....	427
11.13	在 Tkinter 中实现一个带标签的记事本 .....	429
11.14	使用 wxPython 实现带面板的记事本 .....	431
11.15	在 Jython 中实现一个 ImageJ 插件 .....	433
11.16	用 Swing 和 Jython 来通过 URL 查看图片 .....	434
11.17	在 Mac OS 中获得用户输入 .....	434
11.18	程序化地创建 Python Cocoa GUI.....	437
11.19	用 IronPython 实现淡入窗口 .....	439

## 第 12 章 XML 处理 ..... 441

引言.....	441	
12.1	检查 XML 的格式完好性 .....	443
12.2	计算文档中标签的个数 .....	444
12.3	获得 XML 文档中的文本 .....	445
12.4	自动探测 XML 的编码 .....	447
12.5	将一个 XML 文档转化成 Python 对象树.....	449
12.6	从 XML DOM 节点的子树中删除仅有空白符的文本节点 .....	451
12.7	解析 Microsoft Excel 的 XML .....	452
12.8	验证 XML 文档.....	454
12.9	过滤属于指定命名空间的元素和属性 .....	455
12.10	用 SAX 合并连续的文本事件 .....	458

12.11 使用 MSHTML 来解析 XML 或 HTML .....	461
<b>第 13 章 网络编程 .....</b>	<b>462</b>
引言 .....	462
13.1 通过 Socket 数据报传输消息 .....	464
13.2 从 Web 抓取文档 .....	466
13.3 过滤 FTP 站点列表 .....	467
13.4 通过 SNTP 协议从服务器获取时间 .....	468
13.5 发送 HTML 邮件 .....	469
13.6 在 MIME 消息中绑入文件 .....	471
13.7 拆解一个分段 MIME 消息 .....	474
13.8 删 除邮件消息中的附件 .....	475
13.9 修复 Python 2.4 的 email.FeedParser 解析的消息 .....	477
13.10 交互式地检查 POP3 邮箱 .....	479
13.11 探测不活动的计算机 .....	482
13.12 用 HTTP 监视网络 .....	487
13.13 网络端口的转发和重定向 .....	489
13.14 通过代理建立 SSL 隧道 .....	492
13.15 实现动态 IP 协议 .....	495
13.16 登录到 IRC 并将消息记录到磁盘 .....	498
13.17 访问 LDAP 服务 .....	500
<b>第 14 章 Web 编程 .....</b>	<b>502</b>
引言 .....	502
14.1 测试 CGI 是否在工作 .....	503
14.2 用 CGI 脚本处理 URL .....	506
14.3 用 CGI 上传文件 .....	507
14.4 检查 web 页面的存在 .....	509
14.5 通过 HTTP 检查内容类型 .....	510
14.6 续传 HTTP 下载文件 .....	512
14.7 抓取 Web 页面时处理 Cookie .....	513
14.8 通过带身份验证的代理进行 HTTPS 导航 .....	516
14.9 用 Jython 实现 Servlet .....	517
14.10 寻找 Internet Explorer 的 cookie .....	519
14.11 生成 OPML 文件 .....	521
14.12 聚合 RSS Feed .....	524
14.13 通过模板将数据放入 Web 页面 .....	527

14.14 在 Nevow 中呈现任意对象	530
<b>第 15 章 分布式编程</b>	<b>534</b>
引言	534
15.1 实现一个 XML-RPC 方法调用	536
15.2 服务 XML-RPC 请求	537
15.3 在 Medusa 中使用 XML-RPC	539
15.4 允许 XML-RPC 服务被远程终止	541
15.5 SimpleXMLRPCServer 的一些细节	542
15.6 给一个 XML-RPC 服务提供一个 wxPython GUI	544
15.7 使用 Twisted 的 Perspective Broker	546
15.8 实现一个 CORBA 服务和客户	549
15.9 使用 telnetlib 执行远程登录	551
15.10 使用 SSH 执行远程登录	554
15.11 通过 HTTPS 验证一个 SSL 客户端	557
<b>第 16 章 关于程序的程序</b>	<b>559</b>
引言	559
16.1 验证字符串是否代表着一个合法的数字	564
16.2 导入一个动态生成的模块	565
16.3 导入一个名字在运行时被确定的模块	567
16.4 将参数和函数联系起来	568
16.5 组合函数	571
16.6 使用内建的 Tokenizer 给 Python 源码上色	572
16.7 合并和拆解 Token	575
16.8 检查字符串是否有平衡的圆括号	577
16.9 在 Python 中模拟枚举	580
16.10 在创建列表推导时引用它自身	583
16.11 自动化 py2exe 将脚本编译成 Windows 可执行文件的过程	585
16.12 在 UNIX 中将主脚本和模块绑成一个可执行文件	587
<b>第 17 章 扩展和嵌入</b>	<b>590</b>
引言	590
17.1 实现一个简单的扩展类型	592
17.2 用 Pyrex 实现一个简单的扩展类型	597
17.3 在 Python 中使用 C++ 库	598
17.4 调用 Windows DLL 的函数	601

17.5 在多线程环境中使用 SWIG 生成的模块 .....	603
17.6 用 PySequence_Fast 将 Python 序列转为 C 数组 .....	604
17.7 用迭代器逐个访问 Python 序列的元素 .....	608
17.8 从 Python 可调用的 C 函数中返回 None .....	611
17.9 用 gdb 调试动态载入的 C 扩展 .....	613
17.10 调试内存问题 .....	614
<b>第 18 章 算法 .....</b>	<b>616</b>
引言 .....	616
18.1 消除序列中的重复 .....	619
18.2 在保留序列顺序的前提下消除其中的重复 .....	621
18.3 生成回置采样 .....	625
18.4 生成无回置的抽样 .....	626
18.5 缓存函数的返回值 .....	627
18.6 实现一个 FIFO 容器 .....	629
18.7 使用 FIFO 策略来缓存对象 .....	631
18.8 实现一个 Bag (Multiset) 收集类型 .....	634
18.9 在 Python 模拟三元操作符 .....	637
18.10 计算素数 .....	640
18.11 将整数格式化为二进制字符串 .....	642
18.12 以任意数为基将整数格式化为字符串 .....	644
18.13 通过法雷分数将数字转成有理数 .....	646
18.14 带误差传递的数学计算 .....	648
18.15 以最大精度求和 .....	651
18.16 模拟浮点数 .....	653
18.17 计算二维点集的凸包和直径 .....	656
<b>第 19 章 迭代器和生成器 .....</b>	<b>660</b>
引言 .....	660
19.1 编写一个类似 range 的浮点数递增的函数 .....	663
19.2 从任意可迭代对象创建列表 .....	665
19.3 生成 Fibonacci 序列 .....	667
19.4 在多重赋值中拆解部分项 .....	669
19.5 自动拆解出需要的数目的项 .....	670
19.6 以步长 n 将一个可迭代对象切成 n 片 .....	672
19.7 通过重叠窗口循环序列 .....	674
19.8 并行地循环多个可迭代对象 .....	678

19.9	循环多个可迭代对象的矢量积 .....	680
19.10	逐段读取文本文件 .....	683
19.11	读取带有延续符的行 .....	685
19.12	将一个数据块流处理成行流 .....	687
19.13	用生成器从数据库中抓取大记录集 .....	688
19.14	合并有序序列 .....	690
19.15	生成排列、组合以及选择 .....	694
19.16	生成整数的划分 .....	696
19.17	复制迭代器 .....	697
19.18	迭代器的前瞻 .....	701
19.19	简化队列消费者线程 .....	703
19.20	在另一个线程中运行迭代器 .....	705
19.21	用 <code>itertools.groupby</code> 来计算汇总报告 .....	706
<b>第 20 章 描述符、装饰器和元类 .....</b>		<b>710</b>
	引言 .....	710
20.1	在函数调用中获得常新的默认值 .....	712
20.2	用嵌套函数来编写 <code>property</code> 属性 .....	715
20.3	给属性值起别名 .....	717
20.4	缓存属性值 .....	719
20.5	用同一个方法访问多个属性 .....	722
20.6	封装一个方法来给类增加功能 .....	723
20.7	增强所有方法来给类增加功能 .....	726
20.8	在运行时给一个类实例添加方法 .....	728
20.9	检查接口的实现 .....	730
20.10	在自定义元类中正确地使用 <code>__new__</code> 和 <code>__init__</code> .....	732
20.11	允许对 <code>List</code> 的可变方法的链式调用 .....	734
20.12	通过更紧凑的语法使用协作的超类调用 .....	736
20.13	不使用 <code>__init__</code> 来初始化实例属性 .....	738
20.14	实例属性的自动初始化 .....	740
20.15	重新加载时自动更新类实例 .....	743
20.16	在编译时绑定常量 .....	747
20.17	解决元类冲突 .....	752

# 计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

**Java** 一览无余: **Java** 视频教程 | **Java SE** | **Java EE**

**.Net** 技术精品资料下载汇总: **ASP.NET** 篇

**.Net** 技术精品资料下载汇总: **C#语言** 篇

**.Net** 技术精品资料下载汇总: **VB.NET** 篇

撼世出击: **C/C++** 编程语言学习资料尽收眼底 电子书+视频教程

**Visual C++(VC/MFC)** 学习电子书及开发工具下载

**Perl/CGI** 脚本语言编程学习资源下载地址大全

**Python** 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总: **MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

**UML** 学习电子资下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网: 精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

**Linux** 系统管理员必备参考资料下载汇总

**Linux shell**、内核及系统编程精品资料下载汇总

**UNIX** 操作系统精品学习资料<电子书+视频>分类总汇

**FreeBSD/OpenBSD/NetBSD** 精品学习资源索引 含书籍+视频

**Solaris/OpenSolaris** 电子书、视频等精华资料下载索引

# 文本

## 引言

感谢：Fred L. Drake, Jr., PythonLabs

对于脚本语言来说，文本处理任务构成了一个重要的组成部分，每个人都会同意文本处理非常有用。每个人都会有一些文本需要重新格式化或者转化为另一种形式。问题是，每个程序都与另一个程序有点不同，无论它们是多么相似，想提取出一些可复用的代码片段并用它来处理不同的文件格式仍然是非常困难的。

## 什么是文本

看起来问题有点简单得过分了，事实上，我们看到了文本，就知道了什么是文本，文本是一串字符，这正是它与二进制数据之间的不同。二进制数据是一串字节。

不幸的是，所有的数据进入程序中都只是一串字节。没有什么库函数能够帮助我们确定某一个特定的字节串是否代表文本，我们只能自己创造一些试探的方法来判断那些数据是否能够用文本的方式来处理（不一定正确）。第 1.11 节就展示了一种试探的方法。

Python 的字符串是一串不可改变的字节或者字符。绝大多数我们创造的方法以及处理字符串的方式都是把它们当做一串字符来处理的，但是一些字节串也是可以处理的。Unicode 字符串是一串不可改变的 Unicode 字符：由 Unicode 字符串转到普通字符串或者由普通字符串转化为 Unicode 字符串的过程是通过 `codecs`（编码器-解码器）对象来完成的，在这些对象涉及的很多标准转化方法中，字符串可以被表示为一串字节（也被称为编码和字符集）。但是需要注意的是，Unicode 字符串不像字节串那样可以身兼两职。第 1.20 节、第 1.21 节和第 1.22 节展示了 Python 中 Unicode 的一些基本方法。

现在假设我们的程序通过上下文环境确认了它要处理的是文本。程序一般会接收外部输入，这通常是最好的方法。我们能够识别该文件的原因是，它有个已知的名字和已经定义好了的格式（在 UNIX 世界这很普遍），或者它有个著名的文件扩展名来指示出

它内容的格式（在 Windows 世界这很普遍）。现在有个问题：我们必须使用“格式”这种东西，要不然上面这段话毫无意义。人们不是认为文本很简单吗？

让我们面对这个问题：其实并没有所谓的“纯”文本这样的东西，即使有，我们也可能不会关心（也有例外，计算机语言学家在某些情况下可能会研究纯粹的文本）。我们的程序想处理的东西是包含在文本中的信息。我们关心的文本可能包含了配置命令、控制命令、流程定义命令、供人类阅读的文档、甚至制表信息。包含配置数据和一系列命令的文本通常可以通过严格的语法检查来验证，然后才能决定是否可以依赖其中的信息。向用户提示输入文本中的错误还远远不够，这也不是我们要讨论的主题。

供人类阅读的文档似乎比较简单，但仍然有很多细节需要注意。由于它们通常被写为自然语言的形式，它们的语法和句法很难检查。不同的文本可能会使用不同的字符集和编码，如果事先不知道相关的编码信息，想检查出一段文本使用了何种字符集和编码是极其困难的，甚至是不可能的。然而，对于自然语言文档的正确呈现，这却是非常必要的。自然语言文本也有结构，但是这种结构并不明显而且还需要你至少懂一点该种自然语言。字符组成了单词，单词再形成了句子，然后句子构成了段落，但仍然有更大的结构。单独的一个段落很难定位，除非你知道文档的排版约定：究竟是每行构成一个段，还是多行组成一个段？如果是后者，我们怎么判断哪些行构成了一段？段之间可能会被空白行、缩进或者其他特殊符号隔开。第 19.10 节就给出了一个读取文本文件中由空白行隔开段落的例子。

制表信息就像自然语言文本一样也有很多值得讨论的地方，它实际上给输入的格式增加了第二个维度：文本不再是线性的，也不再是一串字符，而是一个字符的矩阵，每一个单独的文本块可以被缩进和组织起来。

## 基本的文本操作

就像其他的数据格式，我们也需要用一些不同的方式在不同的场合下处理文本。总地来说，有三种基本的操作：

- 解析数据并将数据放入程序内部的结构中；
- 将数据以某种方式转化为另一种相似的形式，数据本身发生了改变；
- 生成全新的数据。

有很多方式可以完成解析，一些特别的解析器可以有效处理有诸多限制的数据格式，并可以解析大量其他的格式。比如 RFC 2822 型的邮件头格式（参看 Python 标准库的 rfc822 模块）和由 ConfigParser 模块处理的配置文件格式。netrc 模块则提供了一种解析应用特有格式的解析器例子，这个模块基于 shlex 模块。shlex 提供一个典型的分词器（tokenizer），可应用于一些基本的语言，特别适合用来创建可读的配置文件以及让用户

在一个具有交互和提示能力的环境下输入命令。Python 标准库中有很多类似的特别解析器，关于它们的使用示例和方法请参看第 2 章和第 13 章。Python 中还有一些正式的解析工具，它们依赖于一些庞大的附加包，可以阅读第 16 章的引言部分以获得一个大致的了解。

当我们提到文本的时候，我们的第一个念头往往是，所谓文本处理，有时候就是文本格式转换。在本章中，我们将会看到一些可以应用于各种场合的转换方法。有时我们要处理的文本是存储于外部文件中的，有时我们则直接处理内存中的字符串。

通过 Python 的 `print` 语句，文件对象或者类文件对象的 `write` 方法，我们可以轻易地产生基于程序特定结构的文本数据。对于那些将输出文件作为一个传入参数的应用，我们通常是用该程序的一个方法或者函数来完成此功能。举个例子，函数可以像下面这样使用 `print` 语句：

```
print >> thefile, sometext  
thefile.write(sometext)
```

这就将产生的输出写入到了文件。不过，这并不是通常我们所认为的文本处理，因为在这个过程中并没有文本输入。本书有很多使用 `print` 和 `write` 的例子，在进一步的阅读中你会看到更多的示例。

## 文本的来源

当文本处于内存的时候，如果数据量不是很大，处理起来相对比较容易。对文本的搜索可以轻易和快速地跨越多行，而且不用担心搜索会超出缓存的边界。只要我们能设法让文本处于内存，并以一种普通字符串的形式呈现，就可以充分利用 `string` 对象的各种内建的操作来简单而快速地处理文本。

基于文件的转化则需要一些特别的对待，因为需要考虑 I/O 性能和可观的开销，以及实际上需要放入内存的数据量。当处理位于磁盘上的数据时，由于数据的尺寸，我们常常避免将整个文件载入内存：把一个 80MB 的文件全部放入内存并不是一件随随便便的事情！当我们的程序只需要处理一部分数据时，尽量让它只处理较小的数据段往往能够得到可观的性能提升，因为这样我们可以让程序拥有更多的资源来运行。相对于涉及大量磁盘读写的大块数据处理，使用谨慎的缓存管理通常能够取得更好的效果。与文件相关的内容请参看第 2 章。

另一个有趣的文本来源是网络。通过 `socket` 可以从网络中取回文本。我们可以把 `socket` 看成是一个文件（使用 `socket` 对象的 `makefile` 方法），从 `socket` 中取回的数据可能是完整的一块，也可能是不完整的数据块，这时我们可以继续等待直到更多的数据到达。由于在最后的数据块到达之前文本数据可能是不完整的，所以用 `makefile` 创建的文件对象可能不太适合被传递给文本处理代码。在处理来自网络连接的文本时，在进一步的处理之前，我们通常需要完全读取连接中的所有数据。如果数据很庞大，我们可以

将其存入一个文件，每当有新数据部分到达，就在文件末尾不断添加，直到最后接收完所有数据，然后再将这个文件用于文本处理。如果要求文本处理这一步要在接收完所有数据之前启动，则需要在具体的处理上采用更加精巧的方法。这方面的解析器例子可以参考标准库中的 `htmlllib` 和 `HTMLParser` 模块。

## 字符串基础

Python 提供的用于文本处理的最主要的工具就是字符串——不可改变的字符序列。实际上存在两种字符串：普通字符串，包含了 8 位（ASCII）字符；Unicode 字符串，包含了 Unicode 字符。我们这里不对 Unicode 字符串做太多讨论：它们的处理方法和普通字符串很类似，只不过它们每个字符占用 2（或者 4）个字节，所以它们拥有成千上万（甚至上亿）个不同的字符，而普通字符串却仅有 256 个不同字符。当需要处理一些有不同字母表的文本时，尤其是亚洲的象形文字，Unicode 字符串的价值就体现出来了。而普通字符串对于英文以及一些非亚洲的精简语言已经够用了。比如，所有的欧洲字母表都可以用普通字符串表示，我们采取的典型的方法是使用国际标准编码 ISO-8859-1（或者 ISO-8859-15，如果需要欧洲的货币符号）。

在 Python 中，可以用下列方式表现一个文本字符串：

```
'this is a literal string'  
"this is another string"
```

字符串的值被单引号或者双引号圈起。这两种表示法在程序中完全一样，都允许你在字符串内部把另一种表示法的引用符号包括进来，而无须用反斜线符号进行转义：

```
'isn\'t that grand'  
"isn't that grand"
```

为了让一个文本字符串扩展到多行，可以在一行的末尾使用反斜线符号，那意味着下面的一行仍是上面字符串的延续：

```
big = "This is a long string\  
that spans two lines."
```

如果想让字符串的输出分为两行，可以在字符串中嵌入换行符：

```
big = "This is a long string\n\  
that prints on two lines."
```

还有一种方式是用一对连续的三引用符将字符串圈起：

```
bigger = """  
This is an even  
bigger string that  
spans three lines.  
"""
```

使用这种三引用符，无须在文本中加入续行符和换行符，因为文本将按照原貌被储存

在 Python 的字符串对象中。也可以在字符串前面加一个 r 或者 R，表示该字符串是一个真正的“原”字符串，需要它的原貌：

```
big = r"This is a long string\
with a backslash and a newline in it"
```

使用“原”字符串，反斜线转义被完全忽略了。还可以在字符串前面加一个 u 或者 U 来使之成为一个 Unicode 字符串：

```
hello = u'Hello\u0020World'
```

字符串是无法改变的，那意味着无论你对它进行什么操作，你总是创建了一个新的字符串对象，而不是改变了原有的字符串。字符串是字符的序列，所以也可以通过索引的方法访问单个字符：

```
mystr = "my string"
mystr[0]      # 'm'
mystr[-2]     # 'n'
```

也可以用切片的方法访问字符串的一个部分：

```
mystr[1:4]    # 'y s'
mystr[3:]     # 'string'
mystr[-3:]    # 'ing'
```

切片的方法还可以扩展，增加第三个参数，作为切片的步长：

```
mystr[:3:-1]  # 'gnirt'
mystr[1::2]    # 'ysrn'
```

可以通过循环遍历整个字符串：

```
for c in mystr:
```

上述方法将 c 依次绑定到了 mystr 中的每一个字符。还可以构建另一个序列：

```
list(mystr)    # 返回 ['m', 'y', ' ', 's', 't', 'r', 'i', 'n', 'g']
```

通过简单的加法，还可以实现字符串的拼接：

```
mystr+'oid'   # 'my stringoid'
```

乘法则完成了对字符串多次重复：

```
'xo'*3       # 'xoxoxo'
```

总之，可以对字符串做任何你能够对其他序列所做的操作，前提是不能试图改变字符序列，因为字符串是不能改变的。

字符串对象有很多有用的方法。比如，可以用 s.isdigit() 来测试字符串的内容，如果 s 不是空的而且所有的字符都是数字，该方法将会返回 true (否则返回 false)。还可以用 s.upper() 来创建一个修改过的字符串，该字符串很像原字符串 s，不过其中的每一个字符都是原对应字符的大写形式。可以用 haystack.count('needle') 在一个字符串中搜索另一

个字符串，该方法返回了子串‘needle’在字符串 haystack 中出现的次数。如果有一个庞大的包含多行文本的字符串，可以用 `splitlines` 来将其分隔为多个单行字符串并置入一个列表中：

```
list_of_lines = one_large_string.splitlines()
```

然后还可以用 `join` 来重新生成一个庞大的单个字符串：

```
one_large_string = '\n'.join(list_of_lines)
```

本章展示了字符串对象的许多方法。可以在 Python 的 *Library Reference* 和 *Python in a Nutshell* 中读到更多的相关内容。

Python 中的字符串还可以通过 `re` 模块用正则表达式来操作。正则表达式是一种强大无比（但也很复杂）的工具，你可能已经通过其他语言（比如 Perl）、使用 vi 编辑器、或者使用命令行方式的工具（比如 grep）对它有所了解了。在本章的下半章中你会看到一些使用正则表达式的例子。更多相关文档，请参考 *Library Reference* 和 *Python in a Nutshell*。如果想掌握这方面的知识，J.E.F. Friedl 的 *Mastering Regular Expressions* (O'Reilly) 也是值得推荐的读本。Python 的正则表达式和 Perl 的正则表达式基本相同，Friedl 的书对这方面的内容介绍得非常全面。

Python 的标准模块 `string` 提供的很多功能和字符串方法提供的功能基本相同，后者被打包成一系列函数，而非方法。`string` 模块还提供一些附加的功能，比如 `string.maketrans` 函数，本章中有几节展示了其用法；还有一些有用的字符串常量（比如，`string.digits`，值为‘0123456789’），再比如 Python 2.4 中引入的新类 `Template`，提供了一种简单而灵活的方式来格式化带有内嵌变量的字符串，你将在本章中看到其应用。字符串格式化操作符，%，提供了一种简洁的方法将字符串拼接并根据某些对象（比如浮点数）来精确格式化字符串。在本章中你同样会看到大量相关例子并学会如何根据自己的目的来使用%。Python 还有一些标准模块或者扩展模块，可处理一些特定种类的字符串。本章并未覆盖这样的特殊领域，不过本书第 12 章将全面而深入地讨论 XML 处理方面的内容。

## 1.1 每次处理一个字符

感谢：Luther Blissett

### 任务

用每次处理一个字符的方式处理字符串。

### 解决方案

可以创建一个列表，列表的子项是字符串的字符（意思是每个子项是一个字符串，长度为一。Python 实际上并没有一个特别的类型来对应“字符”并以此和字符串区分开

来)。我们可以调用内建的 list，用字符串作为参数，如下：

```
thelist = list(thestring)
```

也可以不创建一个列表，而直接用 for 语句完成对该字符串的循环遍历：

```
for c in thestring:  
    do_something_with(c)
```

或者用列表推导中的 for 来遍历：

```
results = [do_something_with(c) for c in thestring]
```

再或者，和列表推导效果完全一样，可以用内建的 map 函数，每次取得一个字符就调用一次处理函数：

```
results = map(do_something, thestring)
```

## 讨论

在 Python 中，字符就是长度为 1 的字符串。可以循环遍历一个字符串，依次访问它的每个字符。也可以用 map 来实现差不多的功能，只要你愿意每取到一个字符就调用一次处理函数。还可以用内建的 list 类型来获得该字符串的所有长度为 1 的子串列表（该字符串的字符）。如果想获得的是该字符串的所有字符的集合，还可以调用 sets.Set，并将该字符串作为参数（在 Python 2.4 中，可以用同样的方式直接调用内建的 set）：

```
import sets  
magic_chars = sets.Set('abracadabra')  
poppins_chars = sets.Set('supercalifragilisticexpialidocious')  
print ''.join(magic_chars & poppins_chars) # 集合的交集  
acrd
```

## 更多资料

*Library Reference* 中关于序列的章节；*Perl Cookbook*, 1.5 节。

# 1.2 字符和字符值之间的转换

感谢：Luther Blissett

## 任务

将一个字符转化为相应的 ASCII (ISO) 或者 Unicode 码，或者反其道而行之。

## 解决方案

这正是内建的函数 ord 和 chr 擅长的任务：

```
>>> print ord('a')  
97  
>>> print chr(97)  
a
```

内建函数 `ord` 同样也接收长度为 1 的 Unicode 字符串作为参数，此时它返回一个 Unicode 的码值，最大到 65535。如果想把一个数字的 Unicode 码值转化为一个长度为 1 的 Unicode 字符串，可以用内建函数 `unichr`：

```
>>> print ord(u'\u2020')
8224
>>> print repr(unichr(8224))
u'\u2020'
```

## 讨论

这是个很普通的任务，有时我们需要将字符（在 Python 中是长度为 1 的字符串）转换为 ASCII 或者 Unicode 码，有时则反其道而行之，很常见也很有用。内建的 `ord`、`chr` 和 `unichr` 函数完全满足了相关的需求。但请注意，新手们常常混淆 `chr(n)` 和 `str(n)` 之间的区别：

```
>>> print repr(chr(97))
'a'
>>> print repr(str(97))
'97'
```

`chr` 将一个小整数作为参数并返回对应于 ASCII 单字符的字符串，而 `str`，能够以任何整数为参数，返回一个该整数的文本形式的字符串。

如果想把一个字符串转化为一个包含各个字符的值的列表，可以像下面这样同时使用内建的 `map` 和 `ord` 函数：

```
>>> print map(ord, 'ciao')
[99, 105, 97, 111]
```

若想通过一个包含了字符值的列表创建字符串，可以使用 `'join`、`map` 和 `chr`；比如：

```
>>> print ''.join(map(chr, range(97, 100)))
abc
```

## 更多资料

参见 *Library Reference* 中的内建函数 `chr`、`ord` 和 `unichr` 有关章节以及 *Python in a Nutshell*。

## 1.3 测试一个对象是否是类字符串

感谢：Luther Blissett

### 任务

有时候需要测试一个对象，尤其是当你在写一个函数或者方法的时候，经常需要测试传入的参数是否是一个字符串（或者更准确地说，这个对象是否具有类似于字符串的行为模式）。

## 解决方案

下面给出一个利用内建的 `isinstance` 和 `basestring` 来简单快速地检查某个对象是否是字符串或者 `Unicode` 对象的方法，如下：

```
def isAString(anobj):
    return isinstance(anobj, basestring)
```

## 讨论

很多遇到这个问题的程序员第一反应是进行类型测试：

```
def isExactlyAString(anobj):
    return type(anobj) is type('')
```

然而，这种方法非常糟糕，因为它破坏了 Python 强大力量的源泉——平滑的、基于签名的多态机制。很明显 `Unicode` 对象无法通过这个测试，用户自己编写的 `str` 的子类也不行，甚至任何一种行为表现类似于字符串的用户自定义类型的实例都无法通过测试。

本节推荐的内建函数 `isinstance` 则要好很多。内建类型 `basestring` 的存在使得这个方法成为可能。`basestring` 是 `str` 和 `unicode` 类型的共同基类，任何类字符串的用户自定义类型都应该从基类 `basestring` 派生，这样能保证 `isinstance` 的测试按照预期工作。本质上 `basestring` 是一个“空”的类型，就像 `object`，所以从它派生子类并没有什么开销。

不幸的是，这个似乎完美的 `isinstance` 检查方案，对于 Python 标准库中的 `UserString` 模块提供的 `UserString` 类的实例，完全无能为力。而 `UserString` 对象是非常明显的类字符串对象，只不过它不是从 `basestring` 派生的。如果想支持这种类型，可以直接检查一个对象的行为是否真的像字符串一样，比如：

```
def isStringLike(anobj):
    try: anobj + ''
    except: return False
    else: return True
```

这个 `isStringLike` 函数比方案中给出的 `isAString` 函数慢且复杂得多，但它的确适用于 `UserString`（以及其他类字符串的类型）的实例，也适用于 `str` 和 `unicode`。

Python 中通常的类型检查方法是所谓的鸭子判断法：如果它走路像鸭子，叫声也像鸭子，那么对于我们的应用而言，就可以认为它是鸭子了。`IsStringLike` 函数只不过检查了叫声部分，那其实还不够。如果需要检查 `anobj` 对象的更多的类字符串特征，可以改造 `try` 子句，让它检查更多细节，比如：

```
try: anobj.lower() + anobj + ''
```

根据我的经验，`isStringLike` 函数的测试通常就已经满足需要了。

进行类型验证（或者任何验证任务）的最具 Python 特色的方法是根据自己的预期去执行任务，在此过程中检测并处理由于不匹配产生的所有错误和异常。这是一个著名的处理方式，叫做“获得事后原谅总是比事先得到许可要容易得多（It's easier to ask forgiveness than permission）”，或简称 EAFP。`try/except` 是保证 EAFP 处理风格的关键工具。有时，像本节中的例子一样，可以选择一个简单的判断方法，比如拼接一个空字符串，作为对一系列属性的集合（字符串对象提供的各种操作和方法）的一个替代性判断。

## 更多资料

参见 *Library Reference* 中内建的 `isinstance` 和 `basestring` 的文档以及 *Python in a Nutshell*。

## 1.4 字符串对齐

感谢：Luther Blissett

### 任务

实现字符串对齐：左对齐，居中对齐，或者右对齐。

### 解决方案

这正是 `string` 对象的 `ljust`、`rjust` 和 `center` 方法要解决的问题。每个方法都需要一个参数，指出生成的字符串的宽度，之后返回一个在左端、右端、或者两端都添加了空格的字符串拷贝：

```
>>> print '|', 'hej'.ljust(20), '|', 'hej'.rjust(20), '|', 'hej'.center(20), '|'  
| hej | hej | hej |
```

### 讨论

我们常常能够碰到居中、左对齐或右对齐的文本——比如，你可能会打印一个简单的报告，并以 `monospace` 字体居中显示页码。正因为这种需求很常见，Python 的 `string` 对象提供了三个简单好用的方法。在 Python 2.3 中，填充字符只能是空格。在 Python 2.4 中，默认情况下仍然使用空格，但是可以给这三种方法第二个参数，指定一个填充字符：

```
>>> print 'hej'.center(20, '+')  
++++++hej++++++
```

### 更多资料

*Library Reference* 有关 `string` 的方法；*Java Cookbook*，第 3.5 节。

## 1.5 去除字符串两端的空格

感谢: Luther Blissett

### 任务

获得一个开头和末尾都没有多余空格的字符串。

### 解决方案

字符串对象的 `lstrip`、`rstrip` 和 `strip` 方法正是为这种任务而设计的。这几个方法都不需要参数，它们会直接返回一个删除了开头、末尾或者两端的空格的原字符串的拷贝：

```
>>> x = '    hej    '
>>> print '|', x.lstrip(), '|', x.rstrip(), '|', x.strip(), '|'
| hej | hej | hej |
```

### 讨论

有时候需要给字符串添加一些空格，让其符合预先规定的固定宽度，以完成左右对齐或居中对齐（如前面 1.4 节所介绍的），但有时也需要从两端移除所有的空格（空白、制表符、换行符等）。因为这种需求是如此常见，Python 的字符串对象给出了 3 个方法来提供这种功能。也可以选择去除其他字符，只需提供一个字符串作为这 3 种方法的参数即可：

```
>>> x = 'xyxxxx hejyx yyx'
>>> print '|' + x.strip('xy') + '|'
| hejyx |
```

注意，上面例子中最后获得的字符串的开头和结尾的空格都被保留下来，因为“yx”后面接着的是一些空格：只有开头和结尾的“x”和“y”被真正移除了。

### 更多资料

*Library Reference* 中关于字符串的方法；第 1.4 节；*Java Cookbook* 3.12。

## 1.6 合并字符串

感谢: Luther Blissett

### 任务

有一些小的字符串，想把这些字符串合并成一个大字符串。

## 解决方案

要把一系列小字符串连接成一个大字符串，可以使用字符串操作符 `join`。假如 `pieces` 是一个字符串列表，想把列表中所有的字符串按顺序拼接成一个大字符串，可以这么做：

```
largeString = ''.join(pieces)
```

如果想把存储在一些变量中的字符串片段拼接起来，那么使用字符串格式化操作符%会更好一些：

```
largeString = '%s%s something %s yet more' % (small1, small2, small3)
```

## 讨论

Python 中，+操作符也能够将字符串拼接起来，从而实现类似的功能。假如有一些保存在变量中的字符串片段，使用下面这种代码似乎是一种很自然的方式：

```
largeString = small1 + small2 + ' something ' + small3 + ' yet more'
```

类似地，如果有一个小字符串序列，假设叫做 `pieces`，那么很自然地，可以像这样编写代码：

```
largeString = ''  
for piece in pieces:  
    largeString += piece
```

或者，用完全等同但却更加漂亮和紧凑的方式：

```
import operator  
largeString = reduce(operator.add, pieces, '')
```

不过，不要认为上述例子中给出的方法已经足够好了，上面给出的方法都有许多值得推敲的地方。

Python 中的字符串对象是无法改变的。任何对字符串的操作，包括字符串拼接，都将产生一个新的字符串对象，而不是修改原有的对象。因此拼接 N 个字符串将涉及创建并丢弃 N-1 个中间结果。当然，不创建中间结果的操作会有更佳的性能，但往往不能一步到位地取得最终结果。

如果有少量字符串（尤其是那些绑定到变量上的）需要拼接，甚至有时还需要添加一些额外的信息，Python 的字符串格式化操作符%通常是更好的选择。性能对这种操作完全不是一个问题。和使用多个+操作符相比，%操作符还有一些其他的潜在优点。一旦习惯了它，%也会让你的代码的可读性更好。也无须再对所有的非字符串（如数字）部分调用 `str`，因为格式指定符%s已经暗中做完了这些工作。另一个优点是，还可以使用除%s之外的其他格式指定符，这样可以实现更多的格式要求，比如将浮点数转化为字符串的表示时，可以控制它的有效位数。

## 什么是“序列”？

Python 并没有一个特别的类型叫做 sequence，但序列是 Python 中一个非常常用的术语。序列，严格地讲，是一个可以迭代的容器，可以从中取出一定数目的子项，也可以一次取出一个，而且它还支持索引、切片，还可以传递给内建函数 len（返回容器中子项的数目）。Python 的 list 就是“序列”，你已经见过多次了，但还有很多其他的“序列”（string、unicode 对象、tuple、array.array 等）。

通常，一个对象即使不支持索引、切片和 len。只要具有一次获得一项的迭代能力，对应用而言就已经够用了。这叫做可迭代对象（或者，如果我们把范围限定在拥有有限子项的情况下，那就叫做有边界可迭代对象）。可迭代对象不是序列，而是如字典（迭代操作将以任意顺序每次取得一个 key）、文件对象（迭代操作将给出文本文件的行数）等，还有其他一些，包括迭代器和生成器等。任何可迭代对象都能用在 for 循环语句以及一些等价的环境中（Python 2.4 的生成器表达式、列表推导中的 for 子句、以及很多内建的方法，比如 min、max、zip、sum、str.join 等）。

在 <http://www.python.org/moin/PythonGlossary>，可以发现一个词汇表，Python Glossary，它能够帮助你了解很多术语。虽然本书的编辑尽量严格按照那个词汇表的描述来使用术语，仍可能发现本书在很多地方提到了序列、可迭代对象、甚至列表，实际上，严格地讲，我们都应该指明为有边界的可迭代对象。比如，在本节的开头，我们说“一个小字符串的序列”，实际上那是一个有边界的字符串的可迭代对象。在全书使用“有边界的可迭代对象”这样的术语给人的感觉像是在读一本数学书，而不是一本实践性很强的编程书！所以我们决定采用略微偏离严格术语系统的词，这样有助于获得更好的可读性，同时也能够更好地体现本书的多元化。最后结果还不错，根据上下文环境，那些不怎么严密的术语的真实含义仍然能够被清晰地表达出来。

当一个序列中包含了很多的小字符串的时候，性能就变成了一个很现实的问题。在内部使用了+或者+=（和内建函数 reduce 作用相同，但是更漂亮）的循环所需要的时间跟需要累加的字符数的平方成正比，因为分配空间并填充一个大字符串所需要的时间大致正比于该字符串的长度。幸好 Python 提供了另一个更好的选择。对于字符串对象 s 的 join 方法，我们可以传入一个字符串序列作为其参数，它将返回一个由字符串序列中所有子项字符串拼接而成的大字符串，而且这个过程中只使用了一个 s 的拷贝用于串接所有的子项。举个例子，".join(pieces) 把 pieces 中所有的子项一口吞下，而无须产生子项之间的中间结果，再比如，','.join(pieces) 拼接了所有的子项字符串，并在邻接的两项之间插入了一个逗号和空格。这是一种快速、整洁、优雅且兼具良好可读性的合并大字符串的方法。

但有时并不是所有的数据在一开始就已经就位，比如数据可能来自于输入或计算，这

时可以使用一个 `list` 作为中间数据结构来容纳它们（可以使用 `list` 的 `append` 或 `extend` 方法在末尾添加新的数据）。在取得了所有的数据之后，再调用`".join(thelist)`就可以得到合并之后的大字符串。在我能教给你的 Python 的字符串处理的各种技巧和方法中，这是最重要的一条：很多 Python 程序效能低下的原因是由于它们使用了`+`和`+=`来创建大字符串。因此，一定要提醒自己永远不要使用那种做法，而应该使用本节推荐的`".join`方法。

Python 2.4 在这个问题的改善上做了很多工作，在 Python 2.4 中使用`+=`，性能损失要比以前的版本小一些。但`".join` 仍然要快许多，而且在各方面都更具优势，至少对于新人和粗心的开发人员来讲，它消耗的时钟周期也更少。类似地，psyco（一种特制的 just-in-time[JIT] Python 编译器，请查看 <http://psyco.sourceforge.net/>）能更大幅度地减少`+=`带来的性能损失。不过，我还是要强调，“`.join` 依然是最值得选择的方式。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中关于字符串的方法、字符串格式化操作以及 `operator` 模块的章节。

## 1.7 将字符串逐字符或逐词反转

感谢：Alex Martelli

### 任务

把字符串逐字符或逐词反转过来。

### 解决方案

字符串无法改变，所以，反转一个字符串需要创建一个拷贝。最简单的方法是使用一种“步长”为`-1`的特别的切片方法，这样可立即产生一个完全反转的效果：

```
revchars = astring[::-1]
```

如果要按照单词来反转字符串，我们需要先创建一个单词的列表，将这个列表反转，最后再用 `join` 方法将其合并，并在相邻两词之间都插入一个空格：

```
revwords = astring.split( )          # 字符串->单词列表
revwords.reverse( )                  # 反转列表
revwords = ' '.join(revwords)        # 单词列表->字符串
```

或者，如果喜欢简练而紧凑的“一行解决”的代码：

```
revwords = ' '.join(astring.split( )[::-1])
```

如果想逐词反转但又同时不改变原先的空格，可以用正则表达式来分隔原字符串：

```
import re
revwords = re.split(r'(\s+)', astring)          # 切割字符串为单词列表
revwords.reverse()                               # 反转列表
revwords = ''.join(revwords)                     # 单词列表 -> 字符串
```

注意，最后的 `join` 操作要使用空字符串，因为空格分隔符已经被保存在 `revwords` 列表中了（通过 `re.split`，使用了一个带括弧的组的正则表达式）。当然，也可以写成“一行解决”的形式，只要你乐意：

```
revwords = ''.join(re.split(r'(\s+)', astring)[::-1])
```

不过这样显得过于紧凑，也失去了可读性，不是好的 Python 代码。

## 讨论

在 Python 2.4 中，可以改写那个“一行解决”的逐词反转的代码，使用新的内建函数 `reversed` 来替代原先的可读性略差的切片指示符`[::-1]`：

```
revwords = ' '.join(reversed(astring.split()))
revwords = ''.join(reversed(re.split(r'(\s+)', astring)))
```

至于逐字符反转，`astring[::-1]`仍然是最好的方式，即使在 Python 2.4 中，因为如果要使用 `reversed`，还得调用`"join"`：

```
revchars = ''.join(reversed(astring))
```

新的内建函数 `reversed` 返回一个迭代器（iterator），该对象可以被用于循环或者传递给其他的“累加器”，比如`"join"`，但它并不是一个已经完成的字符串。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中关于序列的切片和类型，以及内建的 `reversed`（Python 2.4）的内容；*Perl Cookbook 1.6*。

## 1.8 检查字符串中是否包含某字符集合中的字符

感谢：Jürgen Hermann、Horst Hansen

### 任务

检查字符串中是否出现了某字符集合中的字符。

### 解决方案

最简单的方法如下，兼具清晰、快速、通用（适用于任何序列，不仅仅是字符串，也适用于任何容器，不仅仅是集合）：

```
def containsAny(seq, asset):
    """ 检查序列 seq 是否含有 asset 中的项 """
    for c in seq:
        if c in asset: return True
    return False
```

也可以使用更高级和更复杂的基于标准库 `itertools` 模块的方法来提高一点性能，不过它们本质上其实是同一种方法：

```
import itertools
def containsAny(seq, asset):
    for item in itertools.ifilter(asset.__contains__, seq):
        return True
    return False
```

## 讨论

对于大多数涉及集合的问题，我们最好使用 Python 2.4 中引入的内建类型 `set`（如果还在使用 Python 2.3，可以使用 Python 标准库中的等价的 `sets.Set` 类型）。然而，总是有例外的情况。比如，一个纯粹的基于集合的方法应该是像这样的：

```
def containsAny(seq, asset):
    return bool(asset & set(seq))
```

不过这种方法就意味着 `seq` 中的每个成员都不可避免地要被检查。而本节方法中给出的函数，从某种角度讲，可以被叫做“短路法”：它一知道答案就迅速返回。如果答案是 `False`，它必须检查 `seq` 中的每个子项——因为除非检查所有的子项，否则我们无法确保 `seq` 中不含有 `asset` 中的元素。不过如果答案是 `True`，我们通常可以很快知道，因为只要找到一个子项是 `asset` 的成员就可以了。当然这通常是依赖于数据的。如果 `seq` 很短或者答案是 `False`，实际上并没有多大区别，但如果 `seq` 很长，用哪种方式来检查就变得极其关键了（尤其是答案是 `True` 而且又可以很快探知结果的情况）。

本节给出的 `containsAny` 的第一个版本有着简洁和清晰的优点：它直观地表达了它背后蕴藏的思想。而第二个版本则可能显得有点“聪明”，这个词在 Python 的世界中可不是一个正面的表达赞美的形容词，因为简洁和清晰才是这个世界的核心价值。不过，第二个版本也有值得思考的地方，因为它展示了一种高级的、基于标准库的 `itertools` 模块的方法。大多数情况下高级方法总是比低级方法更好（当然在本节的这个特别的例子中，情况正好相反）。`itertools.ifilter` 要求传入一个断定（译者注：`Predicate`，原意为断言、谓词或述词）和一个可迭代对象，然后筛选出可迭代对象中的满足该“断定”的描述的所有子项。这里，所谓的“断定”，我们用的是 `anyset.__contains__`，当我们编写 `in anyset` 这样的代码来做检查的时候，其内部调用的就是 `anyset.__contains__`。所以，如果 `ifilter` 找到了什么东西，比如它找出一个 `seq` 的子项，同时也正是 `anyset` 的成员，函数就会立刻返回 `True`。如果程序运行到了 `for` 语句之后，那肯定表示 `return`

`True` 根本没有被执行，因为 `seq` 中的任何子项都不是 `anyset` 的成员，此时只能返回 `False`。

### 什么是“断定”？

在一些编程的讨论中你常常可以看到这个词(`predicate`)：意为一个返回 `True` 或 `False` 的函数(或者其他可调用对象)。如果它返回 `True`，我们就称这个断定成立。

如果你的程序需要用到像 `containsAny` 这样的函数来检查一个字符串(或其他序列)是否包含了某个集合的成员，也可能会写出这样的变种：

```
def containsOnly(seq, aset):
    """ 检查序列 seq 是否含有 aset 中的项 """
    for c in seq:
        if c not in aset: return False
    return True
```

`containsOnly` 和 `containsAny` 完全一样，只不过正好逻辑颠倒了一下。下面还有一个类似的例子，完全没办法短路(必须检查所有的子项)，我们最好使用于内建的 `set` 类型(Python 2.4；或 Python 2.3 中的 `sets.Set`，使用方法一样)：

```
def containsAll(seq, aset):
    """ 检查序列 seq 是否含有 aset 的所有的项 """
    return not set(aset).difference(seq)
```

如果不习惯用 `set`(或 `sets.Set`)的方法 `difference`，可以记住它的语义：任何一个 `set` 对象 `a`, `a.difference(b)`(就像 `a-set(b)`) 返回 `a` 中所有不属于 `b` 的元素。比如：

```
>>> L1 = [1, 2, 3, 3]
>>> L2 = [1, 2, 3, 4]
>>> set(L1).difference(L2)
set([ ])
>>> set(L2).difference(L1)
set([4])
```

希望这样的结果可以解释得更清楚：

```
>>> containsAll(L1, L2)
False
>>> containsAll(L2, L1)
True
```

另一方面，不要把 `difference` 和 `set` 的其他方法搞混了，比如 `symmetric_difference`，它返回的集合包含了所有属于其中一个集合且不属于另一个集合的元素。

如果需要处理 `seq` 和 `aset` 中的字符串(非 Unicode)，可能不需要本节中这些通用的函数，而可以尝试更加特殊的方式，如第 1.10 节中的方法，基于字符串的方法 `translate` 和 Python 标准库中的 `string.maketrans` 函数：

```
import string
notrans = string.maketrans('', '') # identity "translation"
def containsAny(astr, strset):
    return len(strset) != len(strset.translate(notrans, astr))
def containsAll(astr, strset):
    return not strset.translate(notrans, astr)
```

这看起来有点诡异的方法主要依赖于这个事实：`strset.translate(notrans, astr)`是`strset`的子序列，而且是由所有不属于`astr`的字符组成的。如果这个子序列和`strset`有同样的长度，说明`strset.translate`没有删除任何字符，因此`strset`中任何一个字符都不属于`astr`。相反，如果子序列是空，说明所有的字符都被移除了，所以所有属于`strset`的字符也属于`astr`。当程序员们把字符串当做字符集合的时候，很自然地就会想到使用`translate`办法，因为这个方法速度不错，而且灵活易用，更多细节参看第 1.10 节。

不过本节中的两种方式的通用性完全不同。早先提出的方式通用性非常好：并不局限于字符串处理，它对你要处理的对象类型的要求也更少。而基于`translate`方法的方式则相反，它要求`astr`和`strset`都是普通字符串，或者在行为和功能上要与普通字符串非常相似。甚至连 Unicode 字符串都不行，因为 Unicode 字符串的`translate`方法的签名不同于普通字符串对应的`translate`版本——Unicode 版本只需要一个参数（该参数是一个把码值映射到 Unicode 字符串或者`None`的`dict`对象），普通字符串版本则需要两个（必须都是普通字符串）。

## 更多资料

1.10 节；*Library Reference* 和 *Python in a Nutshell* 中关于普通字符串和 Unicode 字符串的`translate`方法的文档，`string` 模块的`maketrans` 函数的内容；同上阅读材料中的内建`set` 对象，`sets` 和 `itertools` 模块，以及特殊的`__contains__` 方法相关内容。

## 1.9 简化字符串的 `translate` 方法的使用

感谢：Chris Perkins、Raymond Hettinger

### 任务

用字符串的`translate`方法来进行快速编码，但却发现很难记住这个方法和`string.maketrans`函数的应用细节，所以需要对它们做个简单的封装，以简化其使用流程。

### 解决方案

字符串的`translate`方法非常强大而灵活，具体细节可参考第 1.10 节。正因为它的威力和灵活性，将它“包装”起来以简化应用就成了个好主意。一个返回闭包的工厂函数可以很好地完成这种任务：

```
import string
def translator(frm='', to='', delete='', keep=None):
    if len(to) == 1:
        to = to * len(frm)
    trans = string.maketrans(frm, to)
    if keep is not None:
        allchars = string.maketrans('', '')
        delete = allchars.translate(allchars, keep.translate(allchars,
delete))
    def translate(s):
        return s.translate(trans, delete)
    return translate
```

## 讨论

我经常发现我有使用字符串的 `translate` 方法的需求，但每次我都得停下来回想它的用法细节（见第 1.10 节提供的更多细节信息）。所以，我干脆给自己写了个类（后来改写成了本节中展示的工厂闭包的形式），把各种可能性封闭在一个简单易用的接口后面。现在，如果我需要一个函数来选出属于指定集合的字符，我就可以简单地创建并使用它：

```
>>> digits_only = translator(keep=string.digits)
>>> digits_only('Chris Perkins : 224-7992')
'2247992'
```

移除属于某字符集合的元素也同样简单：

```
>>> no_digits = translator(delete=string.digits)
>>> no_digits('Chris Perkins : 224-7992')
'Chris Perkins : -'
```

甚至，我可以用某个字符替换属于某指定集合的字符：

```
>>> digits_to_hash = translator(from=string.digits, to='#')
>>> digits_to_hash('Chris Perkins : 224-7992')
'Chris Perkins : #####-####'
```

虽然后面那个应用显得有点特殊，但我仍然不时地碰到有这种需求的任务。

当然，我的设计有点武断，当 `delete` 参数和 `keep` 参数有重叠部分的时候，我让 `delete` 参数优先：

```
>>> trans = translator(delete='abcd', keep='cdef')
>>> trans('abcdefg')
'ef'
```

对于你的程序，如果 `keep` 被指定了，可能忽略掉 `delete` 会更好一些，再或者，如果两者都被指定了，抛出个异常也不错，因为在一个对 `translator` 的调用中同时指定两者可能没什么意义。另外，和第 1.8 节和第 1.10 节相似，本节代码只适用于普通字符串，对 Unicode 字符串并不适用。参看第 1.10 节，可以了解到怎样为 Unicode 字符串

编写类似功能的代码，并可看到 Unicode 的 `translate` 方法与普通（单字节）字符串的 `translate` 的区别。

### 闭包

闭包（closure）不是什么复杂得不得了的东西：它只不过是个“内层”的函数，由一个名字（变量）来指代，而这个名字（变量）对于“外层”包含它的函数而言，是本地变量。我们用一个教科书般的例子来说明：

```
def make_adder(addend):
    def adder(augend): return augend+addend
    return adder
```

执行 `p = make_addr(23)` 将产生内层函数 `adder` 的一个闭包，这个闭包在内部引用了名字 `addend`，而 `addend` 又绑定到数值 23。`q = make_adder(42)` 又产生另一个闭包，这次名字 `addend` 则绑定到了值 42。`q` 和 `p` 相互之间并无关联，因此它们可以相互独立地和谐共存。现在我们就可以执行它们了，比如，`print p(100), q(100)` 将打印出 123 142。

实际上，我们一般认为 `make_adder` 指向一个闭包，而不是说什么迂腐拗口的“一个返回闭包的函数”——幸运的是，根据上下文环境，通常这样也不至于造成误解。称 `make_adder` 为一个工厂（或者工厂函数）也是简洁明确的；还可以称它为一个闭包工厂来强调它创建并返回闭包，而不是返回类或者类的实例。

## 更多资料

参看第 1.10 节中关于本节 `translator(keep=...)` 的一个等价实现，以及该节对 `translate` 方法的更多描述，还有 Unicode 字符串的对应方案；*Library Reference* 和 *Python in a Nutshell* 中的字符串的 `translate` 方法的文档，`string` 模块的 `maketrans` 函数的相关内容。

## 1.10 过滤字符串中不属于指定集合的字符

感谢：Jürgen Hermann、Nick Perkins、Peter Cogolo

### 任务

给定一个需要保留的字符的集合，构建一个过滤函数，并可将其应用于任何字符串 `s`，函数返回一个 `s` 的拷贝，该拷贝只包含指定字符集合中的元素。

### 解决方案

对于此类问题，`string` 对象的 `translate` 方法是又快又好用的工具。不过，为了有效地使用 `translate` 来解决问题，事先我们必须做一些准备工作。传递给 `translate` 的第一个参数是一个翻译表：在本节中，我们其实不需要什么翻译，所以我们必须准备一个特制的

参数来指明“无须翻译”。第二个参数指出了我们需要删除的字符：这个任务要求我们保留（正好反过来，不是删除）属于某字符集合的字符，所以我们必须为该字符集合准备一个补集，作为第二个参数——这样就可以删除所有我们不想保留的字符。闭包是一次性完成所有准备工作的最好方法，它能够返回一个满足需求的快速过滤函数：

```
import string
# 生成所有字符的可复用的字符串，它还可以作为
# 一个翻译表，指明“无须翻译”
allchars = string.maketrans('', '')
def makefilter(keep):
    """ 返回一个函数，此返回函数接受一个字符串为参数
        并返回字符串的一个部分拷贝，此拷贝只包含在
        keep 中的字符，注意 keep 必须是一个普通字符串
    """
    # 生成一个由所有不在 keep 中的字符组成的字符串：keep 的
    # 补集，即所有我们需要删除的字符
    delchars = allchars.translate(allchars, keep)
    # 生成并返回需要的过滤函数（作为闭包）
    def thefilter(s):
        return s.translate(allchars, delchars)
    return thefilter
if __name__ == '__main__':
    just_vowels = makefilter('aeiouy')
    print just_vowels('four score and seven years ago')
# 输出: ouceaaeyeaaoo
    print just_vowels('tiger, tiger burning bright')
# 输出: ieieuuii
```

## 讨论

理解本节技巧的关键在于对 Python 标准库的 `string` 模块的 `maketrans` 函数以及字符串对象的 `translate` 方法的理解。`translate` 应用于一个字符串并返回该字符串的一个拷贝，这个拷贝中的所有字符都将按照传入的第一个参数（翻译表）指定的替换方式来替换，而且，第二个参数指定的所有字符都将被删除。`maketrans` 是创建翻译表的一个工具函数。（翻译表是一个正好有 256 个字符的字符串 `t`：当你把 `t` 作为第一个参数传递给 `translate` 方法时，原字符串中的每一个字符 `c`，在处理之后都被翻译成了字符 `t[ord(c)]`。）

在本节的技巧中，我们将整个过滤任务分解为准备阶段和执行阶段，使效能达到了最大化。由于包含所有字符的字符串需要重复使用，我们只创建它一次，并在模块导入后将它设置为全局变量。采用这种方式的原因是我们确定每个过滤函数都使用同样的翻译表，因此它非常节省内存。而我们要传递给 `translate` 的第二个参数——需要删除的字符，则依赖于需要保留的字符集合，因为它完全是后者的“补集”：我们需要通知 `translate` 删除我们不想保留的字符。所以，我们用 `makefilter` 工厂函数来创建需要删除

的字符集合（字符串），即通过使用 `translate` 方法来删除“需要保留的字符”，这一步很快得以完成。和本节的其他函数一样，无论是创建还是执行，`translate` 的速度都非常快。程序中的执行部分给出的测试代码，则展示了如何通过调用 `makefilter` 构建一个过滤函数，并给这个过滤函数绑定一个名字（只需简单地给 `makefilter` 的返回结果指定个名字即可），然后对一些测试字符串调用该函数并打印出结果。

顺带一提，用 `allchars` 作为参数调用过滤函数会把所有需要保留的字符处理成一种非常规整的字符串——严格按照字母表排序而且没有重复的字符。可以根据这种思路编写一个很简单的函数，将以字符串形式给出的字符集合处理成规整的形式：

```
def canonicform(s):
    """ 给定字符串 s，将 s 的字符以一种规整的字符串形式返回：
        按照字母顺序排列且没有重复 """
    return makefilter(s)(allchars)
```

在“解决方案”小节中给出的代码，使用了 `def` 语句来建立一个嵌套的函数（闭包），这是因为 `def` 是最常见、最通用，也是最清晰的创建函数的语句。不过如果你乐意，也可以用 `lambda` 来代替原来的语句，只需修改 `makefilter` 函数中的 `def` 和 `return` 语句，然后写成只需一行的 `return lambda` 语句：

```
return lambda s: s.translate(allchars, delchars)
```

大多数 Python 玩家认为，相对于 `lambda`，`def` 更清晰且更具可读性。

既然本节处理的一些字符串可以被看作字符集合，也可以用 `sets.Set` 类型（或 Python 2.4 中的内建 `set` 类型）来完成相同的任务。但得益于 `translate` 方法的威力和速度，在类似的这种直接处理字符串的任务中，使用 `translate` 总是比通过 `set` 来实现要快一些。不过，正如第 1.8 节提到的，本节中给出的技巧只适用于普通字符串，对 Unicode 字符串则不适用。

为了能够解决 Unicode 字符串的问题，我们需要做完全不同的准备工作。Unicode 字符串的 `translate` 方法只需要一个参数：一个序列或者映射，并且根据字符串中的每个字符的码值进行索引。码值不是映射的键（或者序列的索引值）的字符会被直接复制，不做改变。与每个字符码对应的值必须是一个 Unicode 字符串（该字符的替换物）或者 `None`（这意味着该字符需要被删除）。这种用法看上去既优雅又强大，可惜对于普通字符串却不适用，所以我们还得重写代码。

通常，我们使用 `dict` 或 `list` 作为 Unicode 字符串的 `translate` 方法的参数，来翻译或者删除某些字符。但由于本节任务有些特殊（保留一些字符，删掉所有其余字符），我们可能会需要一个非常庞大的 `dict` 或 `string`——但仅仅是把所有的其他字符映射到 `None`。更好的办法是，编写一个简单的大致实现了 `__getitem__`（进行索引操作时会调用的特殊方法）方法的类。一旦我们花点功夫完成了这个小类，我们可以让这个类的实例可被调用，还可以直接给这个类起个 `makefilter` 的别名：

```

import sets
class Keeper(object):
    def __init__(self, keep):
        self.keep = sets.Set(map(ord, keep))
    def __getitem__(self, n):
        if n not in self.keep:
            return None
        return unichr(n)
    def __call__(self, s):
        return unicode(s).translate(self)
makefilter = Keeper
if __name__ == '__main__':
    just_vowels = makefilter('aeiouy')
    print just_vowels(u'four score and seven years ago')
# 输出: ouoeaeeeyeao
    print just_vowels(u'tiger, tiger burning bright')
# 输出: ieieuui

```

我们也可以直接把这个类命名为 `makefilter`, 但是, 基于传统, 一个类的名字通常应该首字母大写; 遵循传统一般来说没有什么害处, 所以, 代码就成了这个样子。

## 更多资料

第 1.8 节, *Library Reference* 和 *Python in a Nutshell* 中普通字符串和 Unicode 字符串的 `translate` 方法的有关内容, 以及 `string` 模块的 `maketrans` 函数的介绍。

## 1.11 检查一个字符串是文本还是二进制

感谢: Andrew Dalke

### 任务

在 Python 中, 普通字符串既可以容纳文本, 也可以容纳任意的字节, 现在需要探知(当然, 完全是启发式的试探: 对于这个问题并没有什么精准的算法)一个字符串中的数据究竟是文本还是二进制。

### 解决方案

我们采取 Perl 的判定方法, 如果字符串中包含了空值或者其中有超过 30% 的字符的高位被置 1 (意味着该字符的码值大于 126) 或是奇怪的控制码, 我们就认为这段数据是二进制数据。我们得自己编写代码, 其优点是对于特殊的程序需求, 我们随时可以调整这种启发式的探知方式:

```

from __future__ import division          # 确保/不会截断
import string
text_characters = """join(map(chr, range(32, 127))) + "\n\r\t\b"

```

```
_null_trans = string.maketrans("", "")  
def istext(s, text_characters=text_characters, threshold=0.30):  
    # 若 s 包含了空值, 它不是文本  
    if "\0" in s:  
        return False  
    # 一个“空”字符串是“文本”(这是一个主观但又很合理的选择)  
    if not s:  
        return True  
    # 获得 s 的由非文本字符构成的子串  
    t = s.translate(_null_trans, text_characters)  
    # 如果不超过 30% 的字符是非文本字符, s 是字符串  
    return len(t)/len(s) <= threshold
```

## 讨论

可以轻易地修改函数 `istext` 的启发式探知部分, 只需传递一个指定的阀值作为判断某字符串所含数据是“文本”(即正常的 ASCII 字符加上 4 个“正常”的控制码, 在文本中这几个控制码都是有意义的) 的基准, 默认的阀值是 0.30 (30%)。举个例子, 如果期望它是采用了 iso-8859-1 的意大利文本, 可以给 `text_characters` 参数添加意大利语中的一些重音字母, “à è é ì ò ù”。

很多时候, 需要检查的对象是文件, 而不是字符串, 也就是说要判断文件中的内容是文本还是二进制数据。同样地, 我们仍可采用 Perl 的启发式方法, 用前面提供的 `istext` 函数来检查文件的第一个数据块:

```
def istextfile(filename, blocksize=512, **kwds):  
    return istext(open(filename).read(blocksize), **kwds)
```

注意, 默认情况下, `istext` 函数中的 `len(t)/len(s)` 将被截断成 0, 因为这是一个整数之间的除法结果。以后的版本(估计是 Python 3.0, 几年后发布), Python 中的/操作符的意义会被改变, 这样我们在做除法运算的时候就不会发生截断——如果你确实需要截断, 可以用截断除法操作符//。

不过, 现在 Python 还没有改变除法的语义, 这是为了保证一定的向后兼容性。为了让成千上万行的现有的 Python 程序和模块平滑地工作于所有的 Python 2.x 版本, 这非常重要。不过, 对于语言版本的主版本号的改变, Python 允许进行不考虑向后兼容性的改变。

因此, 对于本节的解决方案中的模块, 按照未来版本中计划的行为模式来改变除法的行为是非常方便的, 我们用这种方式来引入模块:

```
from __future__ import division
```

这条语句并不影响程序的其余部分, 只影响紧随此声明的模块; 通过这个模块, /表现得像“真实的除法”(没有截断)。对于 Python 2.3 和 2.4, division 可能是唯一需要从 `__future__` 导入的模块。其他的一些未来版本中计划的特性, `nested_scope` 和生成

器，现在已经是语言的一部分了，因而无法被关闭——当然明确导入它们没有什么坏处，但只有在你的程序需要能够运行在老版本的 Python 环境下时，这种做法才有意义。

## 更多资料

1.10 节中关于 `maketrans` 函数以及字符串方法 `translate` 的诸多细节；*Language Reference* 中关于真实除法和截断除法的内容。

## 1.12 控制大小写

感谢：Luther Blissett

### 任务

将一个字符串由大写转成小写，或者反其道而行之。

### 解决方案

这正是字符串对象提供 `upper` 和 `lower` 方法的原因。每个方法都不需要参数，直接返回一个字符串的拷贝，其中的每个字母都被改变成大写形式——或小写形式：

```
big = little.upper()
little = big.lower()
```

非字母的字符按照原样被复制。

`s.capitalize` 和 `s[:1].upper() + s[1:].lower()` 相似：第一个字符被改成大写，其余字符被转成小写。`s.title` 也很相似，不过它将每个单词的第一个字母大写（这里的单词可以是字母的序列），其余部分则转成小写：

```
>>> print 'one tWo thrEe'.capitalize()
One two three
>>> print 'one tWo thrEe'.title()
One Two Three
```

### 讨论

操作字符串大小写是很常见的需求，有很多方法可以让你创建需要的字符串。另外，还可以检查一个字符串是否已经是满足需求的形式，比如 `isupper`、`islower` 和 `istitle` 方法，如果给定的字符串不是空的，至少含有一个字母，而且分别满足全部大写、全部小写、每个单词开头大写的条件，这三种方法都会返回一个 `True`，但是却没有类似的 `iscapitalized` 方法。不过如果我们需要一个行为方式类似于“`is...`”的方法，自己编写代码也很简单。如果给定的字符串是空的，那些方法都会返回 `False`。如果给定的字符串非空，但是却不包含任何字母字符，也将全部返回 `False`。

最清楚简单的 `iscapitalized`，仅需简洁的一行：

```
def iscapitalized(s):
    return s == s.capitalize()
```

不过，这偏离了“is...”方法们的行为模式，对于空字符串和不含字母的字符串，它也返回 True。我们再给出一个严格点的版本：

```
import string
notrans = string.maketrans('', '') #identity translation
def containsAny(str, strset):
    return len(strset) != len(strset.translate(notrans, str))
def iscapitalized(s):
    return s == s.capitalize() and containsAny(s, string.letters)
```

这里，我们用了第 1.8 节中的函数来确保，当遇到了空字符串或不含字母的字符串，返回值是 False。不过也正如第 1.8 节的提示一样，那意味着这个特别的 iscapitalized 只适用于普通字符串，对 Unicode 字符串不适用。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中关于字符串方法的介绍；*Perl Cookbook* 1.9，1.8 节。

## 1.13 访问子字符串

感谢：Alex Martelli

### 任务

获取字符串的某个部分。比如，你读取了一条定长的记录，但只想获取这条记录中的某些字段的数据。

### 解决方案

切片是个好方法，但是它一次只能取得一个字段：

```
afield = theline[3:8]
```

如果还需考虑字段的长度，`struct.unpack` 可能更适合。比如：

```
import struct
# 得到一个 5 字节的字符串，跳过 3 字节，得到两个 8 字节字符串，以及其余部分：
baseformat = "5s 3x 8s 8s"
# theline 超出的长度也由这个 base-format 确定
# (在本例中是 24 字节，但 struct.calcsize 是很通用的)
numremain = len(theline) - struct.calcsize(baseformat)
# 用合适的 s 或 x 字段完成格式，然后 unpack
format = "%s %ds" % (baseformat, numremain)
l, s1, s2, t = struct.unpack(format, theline)
```

如果想跳过“其余部分”，只需要给出正确的长度，拆解出 `theline` 的开头部分的数据即可：

```
l, s1, s2 = struct.unpack(baseformat, theline[:struct.calcsize(baseformat)])
```

如果需要获取 5 字节一组的数据，可以利用带列表推导（LC）的切片方法，代码很简单：

```
fivers = [theline[k:k+5] for k in xrange(0, len(theline), 5)]
```

将字符切成一个个单独的字符更加容易：

```
chars = list(theline)
```

如果想把数据切成指定长度的列，用带 LC 的切片方法通常是最简单的：

```
cuts = [8, 14, 20, 26, 30]
pieces = [theline[i:j] for i, j in zip([0]+cuts, cuts+[None])]
```

在 LC 中调用 `zip`，返回的是一个列表，其中每项都是形如(`cuts[k], cuts[k+1]`)这样的数对，除了第一项和最后一项，这两项分别是(`0, cuts[0]`)和(`cuts[len(cuts)-1], None`)。换句话说，每一个数对都给出了用于切割的正确的(`i, j`)，仅有第一项和最后一项例外，前者给出的是切割之前的切片方式，后者给出的是切割完成之后到字符串末尾的剩余部分。LC 利用这些数对就可以正确地将 `theline` 切分开来。

## 讨论

本节受到了 *Perl Cookbook* 1.1 的启发。Python 的切片方法，取代了 Perl 的 `substr`。Perl 的内建的 `unpack` 和 Python 的 `struct.unpack` 也非常相似。不过 Perl 的手段更丰富一点，它可以用`*`来指定最后一个字段长度，并指代剩余部分。在 Python 中，无论是为了获取或者跳过某些数据，我们都得计算和插入正确的长度。不过这不是什么大问题，因为此类抽取字段数据的任务往往可以被封装成小函数。如果该函数需要反复被调用的话，`memoizing`，通常也被称为自动缓存机制，能够极大地提高性能，因为它避免了为 `struct.unpack` 反复做一些格式准备工作。参见第 18.5 节中关于 `memoizing` 的更多细节。

在纯 Python 的环境中，`struct.unpack` 作为字符串切片的一种替代方案，非常好用（当然不能和 Perl 的 `substr` 比，虽然它不接受用`*`指定的区域长度，但仍是值得推荐的好东西）。

这些代码片段，最好被封装成函数。封装的一个优点是，我们不需要每次使用时都计算最后一个区域的长度。下面的函数基本上等价于“解决方案”小节给出的直接使用 `struct.unpack` 的代码片段：

```
def fields(baseformat, theline, lastfield=False):
    # theline 超出的长度也由这个 base-format 确定
    # (通过 struct.calcsize 计算确切的长度)
    numremain = len(theline)-struct.calcsize(baseformat)
```

```
# 用合适的 s 或 x 字段完成格式，然后 unpack
format = "%s %d%s" % (baseformat, numremain, lastfield and "s" or "x")
return struct.unpack(format, theline)
```

一个值得注意（或者说值得批评）的设计是该函数提供了 `lastfield=False` 这样一个可选参数。这基于一个经验，虽然我们常常需要跳过最后的长度未知的部分，有时候我们还是需要获取那段数据。采用 `lastfield and s or x`（等同于 C 语言中的三元运算符，`lastfield?"s":"c"`）这样的表达式，我们省去了一个 `if/else`，不过是否需要为这点紧凑牺牲可读性还有值得商榷之处。参看第 18.9 节中有关在 Python 中模拟三元运算符的内容。

若 `fields` 函数在一个循环内部被调用，使用元组(`baseformat, len(theline), lastfield`)作为 `key` 来充分利用 `memoizing` 机制将极大地提高性能。这里给出一个使用 `memoizing` 机制的 `fields` 版本：

```
def fields(baseformat, theline, lastfield=False, _cache={}):
    # 生成键并尝试获得缓存的格式字符串
    key = baseformat, len(theline), lastfield
    format = _cache.get(key)
    if format is None:
        # 没有缓存的格式字符串，创建并缓存之
        numremain = len(theline)-struct.calcsize(baseformat)
        _cache[key] = format = "%s %d%s" % (
            baseformat, numremain, lastfield and "s" or "x")
    return struct.unpack(format, theline)
```

这种利用缓存的方法，目的是将比较耗时的格式准备工作一次完成，并存储在 `_cache` 字典中。当然，正像所有的优化措施一样，这种采用了缓存机制的优化也需要通过测试来确定究竟能在多大程度上提高性能。对这个例子，我的测试结果是，通过缓存优化的版本要比优化之前快约 30% 到 40%，换句话说，如果这个函数不是你的程序的性能瓶颈部分，其实没有什么必要多此一举。

“解决方案中”给出的另一个关于 LC 的代码片段，也可以封装成函数：

```
def split_by(theline, n, lastfield=False):
    # 切割所有需要的片段
    pieces = [theline[k:k+n] for k in xrange(0, len(theline), n)]
    # 若最后一段太短或不需要，丢弃之
    if not lastfield and len(pieces[-1]) < n:
        pieces.pop()
    return pieces
```

对最后一个代码片段的封装：

```
def split_at(theline, cuts, lastfield=False):
    # 切割所有需要的片段
    pieces = [theline[i:j] for i j in zip([0]+cuts, cuts+[None])]
```

```
# 若不需要最后一段，丢弃之
if not lastfield:
    pieces.pop()
return pieces
```

在上面这些例子中，利用列表推导来切片要比用 struct.unpack 略好一些。

用生成器可以实现一个完全不同的方式，像这样：

```
def split_at(the_line, cuts, lastfield=False):
    last = 0
    for cut in cuts:
        yield the_line[last:cut]
        last = cut
    if lastfield:
        yield the_line[last:]
def split_by(the_line, n, lastfield=False):
    return split_at(the_line, xrange(n, len(the_line), n), lastfield)
```

当需要循环遍历获取的结果序列时，无论是显式调用，还是借助一些可调用的“累加器”，比如”join 来进行隐式调用，基于生成器的方式都会更加合适。如果需要的是各字段数据的列表，你手上得到的结果却是一个生成器，可以调用内建的 list 来完成转化，像这样：

```
list_of_fields = list(split_by(the_line, 5))
```

## 更多资料

第 18.9 节和第 18.5 节；*Perl Cookbook* 1.1。

## 1.14 改变多行文本字符串的缩进

感谢：Tom Good

### 任务

有个包含多行文本的字符串，需要创建该字符串的一个拷贝，并在每行行首添加或者删除一些空格，以保证每行的缩进都是指定数目的空格数。

### 解决方案

字符串对象已经提供了趁手的工具，我们只需写个简单的函数即可满足需求：

```
def reindent(s, numSpaces):
    leading_space = numSpaces * ' '
    lines = [ leading_space + line.strip()
              for line in s.splitlines( ) ]
    return '\n'.join(lines)
```

## 讨论

处理文本的时候，我们常常需要改变一块文本的缩进。“解决方案”给出的代码，在多行文本的每行行首增减了空格，这样每行开头都有相同的空格数。比如：

```
>>> x = """ line one
...     line two
... and line three
...
>>> print x
line one
line two
and line three
>>> print reindent(x, 4)
line one
line two
and line three
```

即使每行的缩进都截然不同，该函数仍能够使它们的缩进变得完全一致，这有时正是我们所需要的，但有时却不是。一个常见的需求是，调整每行行首的空格数，并确保整块文本的行之间的相对缩进不发生变化。无论是正向还是反向调整，这都不是难事。不过，反向调整需要检查一下每行行首的空格，以确保不会把非空格字符截去。因此，我们需要将这个任务分解，用两个函数来完成转化，再加上一个计算每行行首空格并返回一个列表的函数：

```
def addSpaces(s, numAdd):
    white = " "*numAdd
    return white + white.join(s.splitlines(True))
def numSpaces(s):
    return [len(line)-len(line.lstrip()) for line in s.splitlines()]
def delSpaces(s, numDel):
    if numDel > min(numSpaces(s)):
        raise ValueError, "removing more spaces than there are!"
    return '\n'.join([line[numDel:] for line in s.splitlines()])
```

所有这些函数都依赖字符串的方法 `splitlines`，它和根据'\n'来切分的 `split` 很相似。不过 `splitlines` 还有额外的好处，它保留了每行末尾的换行符（当你传入的参数是 `True` 的时候）。有时这非常方便：如果 `splitlines` 这个字符串方法没有提供这个能力，`addSpaces` 不可能这么短小精悍。

然后，我们用这些函数组合成另一个函数来删除行首空格。该函数可以在保持各行之间的相对缩进不变的情况下，只删除它能够删除的空格，让缩进最小的行与左端边界平齐。

```
def unIndentBlock(s):
    return delSpaces(s, min(numSpaces(s)))
```

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中关于序列类型的部分。

## 1.15 扩展和压缩制表符

感谢: Alex Martelli、David Ascher

### 任务

将字符串中的制表符转化成一定数目的空格，或者反其道而行之。

### 解决方案

将制表符转换为一定数目的空格是一种很常见的需求，用 Python 的字符串提供的 `expandtabs` 方法可以轻松解决问题。由于字符串不能被改变，这个方法返回的是一个新的字符串对象，是原字符串的一个修改过的拷贝。不过，仍可以将修改过的拷贝绑定到原字符串的名字：

```
mystring = mystring.expandtabs( )
```

这样并不会改变 `mystring` 原先指向的字符串对象，只不过将名字 `mystring` 绑定到了一个新创建的修改过的字符串拷贝上了，该字符串拷贝中的制表符也已经被扩展为一些空格了。对 `expandtabs` 来说，默认情况下，制表符的宽度为 8；可以给 `expandtabs` 传递一个整数参数来指定新的制表符宽度。

将空格转成制表符则比较少见和怪异。如果真的想要压缩制表符，最好还是用别的办法来解决，因为 Python 没有提供一个内建的方法来“反扩展”空格，将其转化为制表符。当然，我们可以自己写个函数来完成任务。字符串的切分、处理以及重新拼接，往往比对整个字符串反复转换要快得多：

```
def unexpand(astring, tablen=8):
    import re
    # 切分成空格和非空格的序列
    pieces = re.split(r'( +)', astring.expandtabs(tablen))
    # 记录目前的字符串总长度
    lensofar = 0
    for i, piece in enumerate(pieces):
        thislen = len(piece)
        lensofar += thislen
        if piece.isspace():
            # 将各个空格序列改成 tabs+spaces
            numblanks = lensofar % tablen
            numtabs = (thislen-numblanks+tablen-1)/tablen
            pieces[i] = '\t'*numtabs + ' '*numblanks
    return ''.join(pieces)
```

例子中的 `unexpand` 函数，只适用于单行字符串；要处理多行字符串，用`".Join([unexpand(s) for s in astring.splitlines(True)])`即可。

## 讨论

虽然在 Python 的字符串操作中，正则表达式从来不是必不可少的部分，但有时它真的很方便。正如代码所示，`unexpand` 函数利用了 `re.split` 相对于字符串的 `split` 额外提供的特性：当正则表达式包含了一个括弧组时，`re.split` 返回了一个 `list` 列表，列表中的每两个相邻的分隔片段之间都被插入了一个用于分隔的“分隔器”。这样，我们就得到了一个 `pieces` 列表，所有的连续空白字符串和非空白字符串都成为了它的子项；接着我们在 `for` 循环中持续跟踪已处理字符串的长度，并将所有的空白字符片段尽可能地转换成相应的制表符，最后加上必要的剩余空格以保持总体的长度。

对于很多编程任务来说，扩展制表符并不是简简单单地调用 `expandtabs` 方法。比如，需要整理一些 Python 源代码，这些代码写得不是很规范，不只用空格来控制缩进（只用空格是最好的方式），而是采取了制表符和空格的混合方式（这是非常糟糕的做法）。这实际上加剧了复杂性，比如，需要猜测制表符的宽度（采用标准的 4 个空格的缩进方式是值得强烈推荐的）。另外，你可能还需要保留一些在字符串内部的并非用于控制缩进的制表符（有人可能错误地使用了实际的制表符，而不是“\t”，来指代字符串中间的制表字符），甚至你还可能需要对具有不同意义的文本做不同的处理。在某些情况下，问题还不算棘手，比如，假设只需要处理每行文本行首的空白符，而无须理会其他的制表符。一个像下面这样的运用正则表达式的小函数就足够了：

```
def expand_at_linenstart(P, tablen=8):
    import re
    def exp(mo):
        return mo.group( ).expand(tablen)
    return ''.join([re.sub(r'^\s+', exp, s) for s in P.splitlines(True)])
```

`expand_at_linenstart` 函数充分利用了 `re.sub` 函数，`re.sub` 在一个字符串中搜寻符合其正则表达式描述的片段，每当它找到一个匹配，便将匹配的字符串作为一个参数传递给一个函数，并调用该函数返回一个替换匹配字符串的字符串。为了方便，`expand_at_linenstart` 被设计为可以处理多行文本字符串 `P`，并对调用 `splitlines` 的结果使用了列表推导处理，最后`\n.join` 将完成后的各行字符串拼接起来。当然，这样的设计完全也可用于单行文本字符串。

实际的制表符扩展的任务可能会很特殊，比如需要考虑制表符是在一个字符串的中间还是外部，是处于哪种类型的文本中（比如，源代码中的注释文本和代码文本），但不管怎么样，至少都需要做一个断词的工作。另外，也可能需要对待处理的源代码进行一个完全的解析，而不是简单地依赖一些字符串和正则表达式操作。如果你想完成的任务具有这种要求，其工作量会相当可观。可以仔细阅读第 16 章，那一章的内容对初学者有很大帮助。

当你汗流浃背地最终完成了转化任务，一定能深深体会到，不管是写代码还是编辑代码，一定要遵循那种常用的、被推荐的 Python 编码风格：只使用空格、4 个空格缩进一级、不用制表符、在字符串中间的制表符应该用 “\t”，而不是实际的制表符。你喜爱的 Python 编辑器也应该被强化支持这些使用习惯，这样在保存 Python 源代码文件时你能得到一个遵守约定的格式；比如，IDLE（Python 所带的免费的集成开发环境）附带的编辑器就完全支持这些约定。最好能够在问题出现之前就配置好你的编辑器，而不是在问题出现之后采取弥补措施。

## 更多文档

*Library Reference* 的“序列类型”一节下的字符串 `expandtabs` 方法；*Perl Cookbook 1.7*，*Library Reference* 和 *Python in a Nutshell* 文档中的 `re` 模块。

## 1.16 替换字符串中的子串

感谢：Scott David Daniels

### 任务

需要一个简单的方法来完成这样一个任务：给定一个字符串，通过查询一个替换字典，将字符串中被标记的子字符串替换掉。

### 解决方案

下面给出的解决办法既适用于 Python 2.3，也适用于 2.4：

```
def expand(format, d, marker=''', safe=False):
    if safe:
        def lookup(w): return d.get(w, w.join(marker*2))
    else:
        def lookup(w): return d[w]
    parts = format.split(marker)
    parts[1::2] = map(lookup, parts[1::2])
    return ''.join(parts)
if __name__ == '__main__':
    print expand('just "a" test', {'a': 'one'})
# 输出: just one test
```

如果参数 `safe` 是 `False`，则默认条件下，字符串中所有被标记的子字符串必须能够在字典 `d` 中找到，否则，`expand` 会抛出一个 `KeyError` 异常并终止执行。当参数 `safe` 被明确指定为 `True` 时，如果被标记的子字符串在字典中找不到，则被标记的部分也不会被改变。

### 讨论

`expand` 函数代码的主体部分有个很有趣的地方：根据操作是否被要求为安全，它使用

两个不同的嵌套函数（两者有着同样的名字 `lookup`）中的一个。安全意味着被标记的子字符串应该能够在字典查到，如果查不到，不抛出 `KeyError` 异常。如果这个函数并不是必须安全的（默认情况下不安全），`lookup` 根据索引访问字典 `d`，并在该索引（子字符串）不存在的时候抛出个错误。但如果 `lookup` 被要求为安全的，它将使用 `d` 的方法 `get`，`get` 返回根据索引能够查到的值，若找不到就返回在两边加上了标记的被查询的子字符串。给 `safe` 传入 `True`，表明你宁可看到输出中有标记符也不愿看到异常信息。`marker+w+marker` 是可以替换 `w.join(marker*2)` 的另一种方式，但我采用后者的原因是，它展示了一种不太简明却很有意思的构造带引号字符串的方法。

不管用哪个版本的 `lookup`，`expand` 都会执行切分、修改、拼接——这些在 Python 的字符串处理中最重要的操作。在 `expand` 中进行修改的部分，使用了指定了步长的列表切片方法。确切地说，`expand` 访问并重新绑定了 `parts` 的奇数索引的项，因为这些项正好是原字符串中位于两个标记符之间的部分。因此，它们就是被标记的子字符串，也就是需要在字典中查找的字符串。

本节解决方案给出的 `expand` 函数接受非常灵活的字符串语法形式，比基于\$的 `string.Template` 更灵活。你如果想让输出字符串包括双引号，也可以指定其他的标记符。当然，函数也没有限制被标记的子串不能是标识符，可以轻松地插入 Python 表达式（`d` 的 `__getitem__` 方法会执行 `eval` 操作）或者任意其他占位符。而且，还可以轻易地搞出点有些不同的更有趣的效果，比如：

```
print expand('just "a" ""little"" test', {'a' : 'one', '' : ''})
```

输出结果是 `just one "little" test`。高级用户可以定制 Python 2.4 的 `string.Template` 类，通过继承来实现上述的所有功能，甚至更多其他高级功能。但本节解决方案中的小巧的 `expand` 函数却更加简洁易用。

## 更多文档

*Library Reference* 关于 `string.Template` (Python 2.4)、序列类型 (关于字符串的 `split`、`join` 方法以及切片操作) 以及字典 (关于索引和 `get` 方法) 的文档资料。更多的关于 Python 2.4 中的 `string.Template` 类的信息，参看第 1.17 节。

## 1.17 替换字符串中的子串——Python 2.4

感谢：John Nielsen、Lawrence Oluyede、Nick Coghlan

### 任务

在 Python 2.4 的环境下，你想完成这样的任务：给定一个字符串，通过查询一个字符串替换字典，将字符串中被标记的子字符串替换掉。

## 解决方案

Python 2.4 提供了一个新的 `string.Template` 类，可以应用于这个任务。下面给出一段代码以展示怎样使用这个类：

```
import string
# 从字符串生成模板，其中标识符被$标记
new_style = string.Template('this is $thing')
# 给模板的 substitute 方法传入一个字典参数并调用之
print new_style.substitute({'thing':5})      # 输出: this is 5
print new_style.substitute({'thing':'test'}) # 输出: this is test
# 另外，也可以给 substitute 方法传递关键字参数
print new_style.substitute(thing=5)          # 输出: this is 5
print new_style.substitute(thing='test')       # 输出: this is test
```

## 讨论

Python 2.3 中，用于标记——替换的字符串格式被写为更加繁琐的形式：

```
old_style = 'this is %(thing)s'
```

标识符被放在一对括弧中，括弧前面一个%，后面一个 s。然后，还需要使用%操作符，使用的格式是将需要处理的字符串放在%操作符左边并在右边放上字典：

```
print old_style % {'thing':5}      # emits: this is 5
print old_style % {'thing':'test'} # emits: this is test
```

当然，这样的代码在 Python 2.4 中也可以正常工作。不过，新的 `string.Template` 提供了一个更简单的替代方法。

当你创建 `string.Template` 实例时，在字符串格式中，可以用两个美元符 (\$) 来代表\$，还可以让那些需要被替换的标识后面直接跟上用于替换的文本或者数字，并用一对花括号 ({ }) 将它们括起来。下面是一个例子：

```
form_letter = '''Dear $customer,
I hope you are having a great time.
If you do not find Room $room to your satisfaction,
let us know. Please accept this $$5 coupon.

Sincerely,
$manager
${name}Inn'''
letter_template = string.Template(form_letter)
print letter_template.substitute({'name':'Sleepy', 'customer':'Fred Smith',
                                  'manager':'Barney Mills', 'room':307,
                                  })
```

上面的代码片段给出下列输出：

```
Dear Fred Smith,
I hope you are having a great time.
If you do not find Room 307 to your satisfaction,
```

```
let us know. Please accept this $5 coupon.
```

Sincerely,  
Barney Mills  
SleepyInn

有时，为了给 `substitute` 准备一个字典做参数，最简单的方法是设定一些本地变量，然后将所有这些变量交给 `locals()`（此函数将创建一个字典，字典的 `key` 就是本地变量，本地变量的值可通过 `key` 来访问）：

```
msg = string.Template('the square of $number is $square')
for number in range(10):
    square = number * number
    print msg.substitute(locals( ))
```

另一个简单的办法是使用关键字参数语法而非字典，直接将值传递给 `substitute`：

```
msg = string.Template('the square of $number is $square')
for i in range(10):
    print msg.substitute(number=i, square=i*i)
```

甚至可以同时传递字典和关键字参数：

```
msg = string.Template('the square of $number is $square')
for number in range(10):
    print msg.substitute(locals( ), square=number*number)
```

为了防止字典的条目和关键字参数显式传递的值发生冲突，关键字参数优先。比如：

```
msg = string.Template('an $adj $msg')
adj = 'interesting'
print msg.substitute(locals( ), msg='message')
# emits an interesting message
```

## 更多资料

*Library Reference* 文档中关于 `string.Template` (2.4) 部分，以及内建的 `locals` 函数的相关信息。

## 1.18 一次完成多个替换

感谢：Xavier Defranc、Alex Martelli

### 任务

你想对字符串的某些子串进行替换。

### 解决方案

正则表达式虽然不易读懂，但有时它的确是最快的方法。`re` 对象（标准库中的 `re` 模

块) 提供的强大 `sub` 方法, 非常利于进行高效的正则表达式匹配替换。下面给出一个函数, 该函数返回一个输入字符串的拷贝, 该拷贝中的所有能够在指定字典中找到的子串都被替换为字典中的对应值:

```
import re
def multiple_replace(text, adict):
    rx = re.compile('|'.join(map(re.escape, adict)))
    def one_xlat(match):
        return adict[match.group(0)]
    return rx.sub(one_xlat, text)
```

## 讨论

本节展示了怎样使用 Python 的标准模块 `re` 来一次完成多个子串的替换。假设你有个基于字典的字符串的映射关系。字典的 `key` 就是你想要替换的子串, 而字典中 `key` 的对应值则正是被用来做替代物的字符串。也可以针对字典的键值对应关系, 调用字符串方法 `replace` 来完成替换, 它将多次处理和创建原文本的复制, 但逻辑却很清晰, 速度也不错。不过 `re.sub` 的回调函数机制可以让处理方式变得更加简单。

首先, 我们根据想要匹配的 `key` 创建一个正则表达式。这个正则表达式形式为 `a1|a2|...|aN`, 由 `N` 个需要被替换的字符串组成, 并被竖线隔开, 创建的方法也很简单, 如代码所示, 一行代码完成。然后, 我们不直接给 `re.sub` 传递用于替换的字符串, 而是传入一个回调函数参数。这样, 每当遇到一次匹配, `re.sub` 就会调用该回调函数, 并将 `re.MatchObject` 的实例作为唯一参数传递给该回调函数, 并期望着该回调函数返回作为替换物的字符串。在本例中, 回调函数在字典中查找匹配的文本, 并返回了对应值。

本节展示的函数 `multiple_replace`, 每次被调用时都会重新计算正则表达式并重新定义 `one_xlat` 辅助函数。但你经常只需要使用同一个固定不变的翻译表来完成很多文本的替换, 这种情况下也许会希望只做一次准备工作。出于这种需求, 也许会使用下面的基于闭包的方式:

```
import re
def make_xlat(*args, **kwds):
    adict = dict(*args, **kwds)
    rx = re.compile('|'.join(map(re.escape, adict)))
    def one_xlat(match):
        return adict[match.group(0)]
    def xlat(text):
        return rx.sub(one_xlat, text)
    return xlat
```

可以给 `make_xlat` 函数传递一个字典参数, 或者其他的可以传递给内建的 `dict` 用于创建一个字典的参数组合; `make_xlat` 返回一个 `xlat` 闭包, 它只需要一个字符串参数 `text`, 并返回 `text` 的一个拷贝, 该拷贝是根据字典给出的翻译表完成了替换之后的结果。

下面给出应用此函数的例子。通常我们可以把这个片段中的示例代码作为本节给出的代码源文件的一部分，这段代码受到前面的 Python 语句的保护不会被执行，除非这个模块被作为主脚本被执行：

```
if __name__ == "__main__":
    text = "Larry Wall is the creator of Perl"
    adict = {
        "Larry Wall" : "Guido van Rossum",
        "creator" : "Benevolent Dictator for Life",
        "Perl" : "Python",
    }
    print multiple_replace(text, adict)
    translate = make_xlat(adict)
    print translate(text)
```

本节中的替换任务常常是基于单词的替换任务，而不是基于任意一个子字符串。通过特殊的 `r'\b'` 序列，正则表达式可以很好地找出单词的开始和结束位置。我们可以修改 `multiple_replace` 和 `make_xlat` 中创建和分配正则表达式 `rx` 的部分，从而完成一些自定义任务：

```
rx = re.compile(r'\b%s\b' % r'\b|\b'.join(map(re.escape, adict)))
```

其余的代码和本节前面给出的一样。但是，这种代码相似性可不是好事：那意味着我们需要很多相似的版本，每个创建正则表达式的部分都有点不同，我们可能会需要做大量的复制粘贴工作，这是代码复用中最糟糕的情况，另外在未来的维护上也增加了麻烦。

编写好代码的一个关键规则是：“一次，只做一次！”当我们注意到代码重复的时候，应该能够很快嗅到“不妙”的气味，并对原来的代码进行重构以提高复用性。因此，为了便于定制，我们更需要的是一个类，而不是函数或者闭包。下面给出个例子，我们实现了一个类，功能近似于 `make_xlat`，但却能够通过子类化和重载进行定制：

```
class make_xlat:
    def __init__(self, *args, **kwds):
        self.adict = dict(*args, **kwds)
        self.rx = self.make_rx()
    def make_rx(self):
        return re.compile('|'.join(map(re.escape, self.adict)))
    def one_xlat(self, match):
        return self.adict[match.group(0)]
    def __call__(self, text):
        return self.rx.sub(self.one_xlat, text)
```

这是对 `make_xlt` 函数的一个完全的替代：另一方面，我们在这之前展示的代码，由于有 `if __name__ == '__main__'` 来保护，即使 `make_xlat` 由以前的函数变成了类，也不会有什么问题。函数更加简单快速，但是类的优势是可以通过面向对象的方法——子类化或

重载某些函数，轻易地实现重新定制。为了对单词进行翻译替换，代码可以这样写：

```
class make_xlat_by_whole_words(make_xlat):
    def make_rx(self):
        return re.compile(r'\b%s\b' % r'\b|\b'.join(map(re.escape,
self.adict)))
```

通过简单的子类化和重载来实现定制化，我们避免了对代码的大量的复制和粘贴，这也是有时我们宁可舍弃更加简单的函数或者闭包不用，而使用面向对象结构的原因。仅仅把相关功能打包成一个类并不能自动实现需要的定制。要实现高度的可定制性，在把功能划分成独立的类方法时必须具有一定的前瞻性。幸好，你不用逼自己第一次就把代码写得很完美；如果代码中没有能够符合任务需求的内部结构时（在这个例子中，我们通过子类化和选择性重载来复用代码），可以而且也应该对代码进行重构，构建出符合需求的结构。当然需要进行一些合适的测试来确保没有把原有的逻辑破坏掉，与此同时，你也完成了对自己的思想内容的重构。访问 <http://www.refactoring.com> 可以获得更多关于重构的艺术和实践的信息。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中关于 `re` 模块的文档；Refactoring 主页 (<http://www.refactoring.com>)。

## 1.19 检查字符串中的结束标记

感谢：Michele Simionato

### 任务

给定一个字符串 `s`，你想检查 `s` 中是否含有多个结束标记中的一个。需要一种快捷、优雅的方式，来替换掉 `s.endswith(end1)`、`s.endswith(end2)` 或 `s.endswith(end3)` 之类的笨重用法。

### 解决方案

对于类似于本节的问题，`itertools imap` 给出了一种快速方便的解决办法：

```
import itertools
def anyTrue(predicate, sequence):
    return True in itertools.imap(predicate, sequence)
def endsWith(s, *endings):
    return anyTrue(s.endswith, endings)
```

### 讨论

一个典型的 `endsWith` 应用是打印出当前目录中所有的图片文件：

```
import os
for filename in os.listdir('.'):
    if endsWith(filename, '.jpg', '.jpeg', '.gif'):
        print filename
```

本节解决方案中给出的思想可以很容易地应用到其他类似的检查任务中去。辅助函数 `anyTrue` 是一个通用而快速的函数，可以给它传入其他的被绑定方法(bound method)作为第一个参数，比如 `s.startswith` 或 `s.__contains__`。事实上，不使用辅助函数而直接编码也许更好：

```
if anyTrue(filename.endswith, ('.jpg', '.gif', '.png')):
```

我认为它的可读性也没什么问题。

### 被绑定方法 (Bound Method)

如果一个 Python 对象提供一个方法，可以直接获得一个已经绑定到该对象的方法，从而直接使用此方法。（比如，可以将其赋值给别的对象、将它作为一个参数传递、或者在一个函数中直接返回它，等等。）举个例子：

```
L = ['fee', 'fie', 'foo']
x = L.append
```

现在 `x` 指向了列表对象 `L` 的一个被绑定方法。调用 `x`，比如 `x('fum')`，和调用 `L.append('fum')` 是完全等价的：结果都是对象 `L` 变成了 `['fee', 'fie', 'foo', 'fum']`。

如果访问的是一个类型或者一个类的方法，而不是一个类型或者类的实例的方法，你得到的是一个非绑定方法，该方法并未“依附”于此类型或者类的任何一个实例：当调用它时，需要提供该类型或类的一个实例作为第一个参数。比如，如果设定 `y = list.append`，你不能直接调用 `y('I')`，因为 Python 猜不出你想给哪个列表对象添加一个 `I`。可以调用 `y(L, 'I')`，这和调用 `L.append('I')` 效果完全一样（只要 `isinstance(L, list)` 成立）。

本节的解决方案和想法来源于 news:comp.lang.python 的一个讨论，并综合和概括了很多人的观点，包括了 Raymond Hettinger、Chris Perkins、Bengt Richter 等。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中关于 `itertools` 和字符串方法的内容。

## 1.20 使用 Unicode 来处理国际化文本

感谢：Holger Krekel

### 任务

需要处理包含了非 ASCII 字符的文本字符串。

## 解决方案

可以在一些使用普通的字节串 str 类型的场合，使用 Python 提供的内置的 unicode 类型。用法很简单，只要接受了在字节串和 unicode 字符串之间的显式转换的方式：

```
>>> german_ae = unicode('\xc3\xa4', 'utf8')
```

这里 german\_ae 是一个 unicode 字符串，代表了小写的德语元音变音（umlaut，或其他分音符）字符“æ”。根据指定的 UTF-8 编码方式，通过解析单字节字符串'\xc3\xa4'，这段代码创建了一个 unicode 字符串。还有很多其他的编码方式，不过 UTF-8 最常用，因为它是最通用的（UTF-8 可以编码任何 unicode 字符串），而且也和 7 位的 ASCII 字符集兼容（任何 ASCII 单字节字符串，也是正确的 UTF-8 编码字符串）。

一旦跨过这一屏障，生活就变得更美好了！可以像处理普通的 str 字符串那样操纵 unicode 字符串：

```
>>> sentence = "This is a " + german_ae
>>> sentence2 = "Easy!"
>>> para = ". ".join([sentence, sentence2])
```

注意，para 是一个 unicode 字符串，这是因为一个 unicode 字符串和一个字节串之间的操作总会产生一个 unicode 字符串——除非这个操作发生错误并抛出异常：

```
>>> bytestring = '\xc3\x4'      #某个非 ASCII 字节串
>>> german_ae += bytestring
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in
position 0: ordinal not in range(128)
```

字符'0xc3'不是 7 位 ASCII 编码中的有效字符，Python 也拒绝猜测其编码。所以，在 Python 中使用 unicode 的关键点是，你要随时明确编码是什么。

## 讨论

如果你遵守一些规范，并且学会处理一些常见的问题，则 Python 中的 unicode 处理是非常简单的事情。这不是说完成一个高效的 Unicode 实现是个简单的任务。不过，正如其他的一些难题一样，无须担心太多：只管使用 Python 的高效的 Unicode 实现就行了。最重要的一点是，首先要完全接受字节串和 unicode 字符串的差异。正如解决方案小节所示，你经常需要通过一个字节串和一个编码方式显式地创建一个 unicode 字符串。不指定编码方式，字节串基本没有什么意义，除非你很有运气而且碰巧那个字节串是 ASCII 文本。

在 Python 中使用 unicode 字符串的最常见的问题是，你正在处理的文本一部分是 unicode 对象，另一部分则是字节串。Python 会简单地尝试把你的字节串隐式地转换成 unicode。它通常假设那些是 ASCII 编码，如果其中碰巧含有非 ASCII 字符，它会给你一个 UnicodeDecodeError 的异常。UnicodeDecodeError 异常通知你，你把 Unicode 和字节串

混在了一起，而且 Python 无法（它根本也不会去尝试）猜测你的字节串代表何种文本。各个 Python 大项目的开发人员们总结出了一些简单的规则，来避免这种运行时的 `UnicodeDecodeError` 异常，该规则可以被总结为一句话：总是在 IO 动作的关口做转换。下面更深入地解释一下。

- 无论何时，当你的程序接收到了来自“外部”的文本数据（来自网络、文件、或者用户输入等）时，应当立刻创建一个 `unicode` 对象，找出最适合的编码，如查看 HTTP 头，或者寻找一个合适的转化方法来确定所用的编码方式。
- 无论何时，当你的程序需要向“外部”发送文本数据（发到网络、写入文件、或者输出给用户等）时，应当探察正确的编码，并用那种编码将你的文本转化成字节串。（否则，Python 会尝试把 `Unicode` 转成 ASCII 字节串，这很有可能发生 `UnicodeEncodeError` 异常，正好是前面例子中给出 `UnicodeDecodeError` 的相反情况）。

遵循这两个规则，可以解决绝大多数的 `Unicode` 问题。如果你仍然遇到了那两种 `UnicodeError` 之一，应当赶快检查是否忘记了在什么地方创建一个 `unicode` 对象，或者忘记了把它转化为编码过的字节串，再或者使用了完全不正确的编码方式。（编码错误也有可能来自于用户，或者其他与你的程序进行交互的程序，因为它们没有遵循编码规则或惯例。）

为了将一个 `Unicode` 字符串转回到编码过的字节串，你通常可以这么做：

```
>>> bytestring = german_ae.decode('latin1')
>>> bytestring
'\xe4'
```

现在，`bytestring` 是德语中的用'latin1'进行编码的 `æ` 字符。注意，'`\xe4`'（Latin1）以及前面展示的'`\xc3\xa4`'（UTF-8）代表了同样的德语字符，但使用了不同的编码。

至此为止，应该能够了解为什么 Python 拒绝在几百种可能的编码中进行猜测了吧。这是一种很重要的设计选择，基于了 *Zen of Python* 原则中的一条：“在模糊含混面前拒绝猜测。”在任何一个 Python 的交互式 shell 提示符下，输入 `import this` 语句，你就可以阅读 *Zen of Python* 中的重要原则。

## 更多资料

`Unicode` 是一个很宽泛的主题，值得推荐的书有：`Unicode: A Primer`，Tony Graham 著 (Hungry Minds, Inc.)，更多细节见 <http://www.menteith.com/unicode/primer/>；以及 Joel Spolsky 写的一篇短小但透彻的文章，“`The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses)!`”具体可见 <http://www.joelonsoftware.com/articles/Unicode.html>。另外参阅 `Library Reference` 和 `Python in a Nutshell` 中关于内建 `str` 和 `unicode` 类型、`unidata` 模块和 `codecs` 模块。

## 1.21 在 Unicode 和普通字符串之间转换

感谢：David Ascher、Paul Prescod

### 任务

需要处理一些可能不符合 ASCII 字符集的文本数据。

### 解决方案

普通字符串可以用多种方式编码成 Unicode 字符串，具体要看你究竟选择了哪种编码：

```
unicodestring = u"Hello world"
# 将 Unicode 转化为普通 Python 字符串: "encode"
utf8string = unicodestring.encode("utf-8")
asciistring = unicodestring.encode("ascii")
isostring = unicodestring.encode("ISO-8859-1")
utf16string = unicodestring.encode("utf-16")
# 将普通 Python 字符串转化为 Unicode: "decode"
plainstring1 = unicode(utf8string, "utf-8")
plainstring2 = unicode(asciistring, "ascii")
plainstring3 = unicode(isostring, "ISO-8859-1")
plainstring4 = unicode(utf16string, "utf-16")
assert plainstring1 == plainstring2 == plainstring3 == plainstring4
```

### 讨论

如果想处理含有非 ASCII 字符的文本数据，首先要懂一些 Unicode——什么是 `unicode`、`unicode` 怎么工作、Python 如何处理 `unicode`。前一节提供了少量却很重要的指导，本节将在这个话题下继续深入讨论。

不用在完全了解 Unicode 的一切之后，才去处理现实世界中的有关 `unicode` 的问题，但是一些基本知识却是不可或缺的。首先，需要理解字节和字符之间的区别。在过去的以 ASCII 字符为主体的语言以及环境中，字节和字符被认为是同一种东西。一个字节可以有 256 个不同的值，因此环境被限制为只能处理不超过 256 个不同的字符。而另一方面，Unicode 则支持成千上万的字符，那也意味着每个 `unicode` 字符占用超过 1 个字节的宽度。因此，首先需要搞清楚字符和字节之间的区别。

标准的 Python 字符串实际上是字节串，这种字符串中的每个字符，长度为 1，实际上就是一个字节。我们还可以用 Python 的标准字符串类型的其他一些术语来称其为 8 位字符串或者普通字符串。在本节中，我们称这种类型的字符串为字节串，以此来提醒你它们的单字节的特点。

Python 的 Unicode 字符是一个抽象的对象，它足够大，能够容纳任何字符，可以同 Python

的长整数进行类比。完全无须担心它的内部表示；只有当你试图将它们传递给一些基于字节处理的函数时——比如文件的 `write` 方法和网络 socket 的 `send` 方法，`unicode` 字符的表示才会成为一个问题。基于这个原因，必须选择以何种方式将这些字符表示成字节。将 `unicode` 字符串转化成字节串被称为对该字符串编码。同样的，如果从一个文件、socket 或者其他基于字节的对象中载入一个 `unicode` 字符串，必须对其解码，将其从字节转成字符。

从 `unicode` 对象转化成字节串有很多方法，这些方法都被称为某种编码。由于一系列历史的、政治的、以及技术上的原因，没有哪种编码是“正确”的。每种编码都有个大小写不敏感的名字，这个名字可以被传递给 `encode` 和 `decode` 函数作为参数。下面给出一些应该知道的编码。

- UTF-8 编码可以应用于任何 Unicode 字符。它也向后兼容 ASCII，所以一个纯粹的 ASCII 文件也可以被认为是一个 UTF-8 文件，而只使用 ASCII 字符的 UTF-8 文件，也完全等同于使用这些字符的 ASCII 文件。这个属性使得 UTF-8 具有极好的向后兼容能力，特别是对一些老的 UNIX 工具来说。UTF-8 是迄今为止 UNIX 上最具主导性的编码，同时也是 XML 文档的默认编码。UTF-8 的主要弱点是，对于一些东方的语言文本，它的效率比较低。
- UTF-16 是微软操作系统和 Java 环境喜爱的编码。它对于西方语言效率略低，但对于东方语言却更有效率。UTF-16 有一个变种，被称为 UCS-2。
- ISO-8859 系列的编码是 ASCII 的超集，每种编码都能处理 256 个不同的字符。这些编码不能支持所有的 Unicode 字符；它们只支持一些特定的语系或语言。ISO-8859-1，也以“Latin-1”的名字为人所知，覆盖了大多西欧和非洲的语言，但不包括阿拉伯语。ISO-8859-2，也被称为“Latin-2”，覆盖了很多东欧的语言，比如匈牙利和波兰。ISO-8859-15，现在在欧洲非常流行，基本上它和 ISO-8859-1 一样，但增加了对欧洲货币符号的支持。

如果想对所有的 Unicode 字符编码，你可能需要用 UTF-8。当需要处理别的程序或者输入设备用其他编码创建的数据时，你才可能需要用到其他编码，或者相反，在需要输出数据给你的下游程序或者输出设备，而它们采用的是另一种特定的编码时。第 1.22 节展示了一个例子，这个例子中，可以看到怎样通过程序的标准输出来驱动其他下游程序或设备。

## 更多资料

Unicode 是一个宽泛的主题，值得推荐的书有：Unicode: A Primer, Tony Graham 著 (Hungry Minds, Inc.)，更多细节见 <http://www.menteith.com/unicode/primer/>；以及 Joel Spolsky 写的一篇短小但透彻的文章，“The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses)！”具体可

见 <http://www.joelonsoftware.com/articles/Unicode.html>。另外参阅 *Library Reference* 和 *Python in a Nutshell* 中关于内建 str 和 unicode 类型、unidata 模块和 codecs 模块，以及第 1.20 节和第 1.22 节。

## 1.22 在标准输出中打印 Unicode 字符

感谢：David Ascher

### 任务

你想将 Unicode 字符串打印到标准输出中（比如为了调试），但是这些字符串并不符合默认的编码。

### 解决方案

通过 Python 标准库中的 codecs 模块，将 sys.stdout 流用转换器包装起来。比如，如果你知道输出会被打印到一个终端，而且该终端以 ISO-8859-1 的编码显式字符，可以这样编写代码：

```
import codecs, sys
sys.stdout = codecs.lookup('iso8859-1')[-1](sys.stdout)
```

### 讨论

Unicode 涵盖极广，全世界的语言字符都在 Unicode 的表示范围之内，另外，Unicode 字符串的内部表示也与 Unicode 使用者没有关系。一个用于处理字节的文件流，比如 sys.stdout，都有自己的编码。可以通过修改 site 模块改变其默认的编码，该文件流将对新文件使用新编码。不过，这样也需要完全改变你的 Python 安装，而且其他一些程序则可能会被搞乱，它们依然会按照你原先的编码设置工作（一般是典型的 Python 标准编码，ASCII）。因此，这种修改并不值得推荐。

本节的方法则用了一个技巧：将 sys.stdout 绑定到一个使用 Unicode 输入和 ISO-8859-1（也就是 Latin-1）输出的流。这种方法并不改变之前 sys.stdout 上的任何编码，如下面代码所示。首先，我们用一个变量指向原来的基于 ASCII 的 sys.stdout：

```
>>> old = sys.stdout
```

然后，我们可以创建一个 Unicode 字符串，这个字符串通常情况下是不能通过 sys.stdout 输出的：

```
>>> char = u"\N{LATIN SMALL LETTER A WITH DIAERESIS}"
>>> print char
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeError: ASCII encoding error: ordinal not in range(128)
```

如果这个操作没有出现错误，那是因为 Python 认为它知道你的“终端”用了什么编码（特别是，如果你的“终端”是 IDLE——Python 所附的免费的开发环境，Python 极有可能能够确认正确的编码）。如果出现了错误，或者没有提示错误，但是输出的字符却不是你期望的，那是因为你的“终端”使用了 UTF-8 编码，而 Python 却不知道。如果属于后者的情况，可以用 `codecs` 流对 `sys.stdout` 进行包装以解决 UTF-8 编码问题，将 `sys.stdout` 绑定到被封装过的流，然后重新试一次：

```
>>> sys.stdout = codecs.lookup('utf-8')[-1](sys.stdout)
>>> print char
ä
```

这个方法只在你的“终端”、终端模拟器或者其他类型的交互式 Python 解释窗口支持 UTF-8 编码时才有效，而且具有极强的字符表现力，能够显示出任何需要的字符。如果没有这样的程序或设备，可以在因特网上找一个适用于你的平台的免费的程序。

Python 会尝试确认你的“终端”的编码，并把编码的名字存在 `sys.stdout.encoding` 中作为一个属性。有时（但不是总是），它能够判断出正确的编码。IDLE 已经对 `sys.stdout` 进行了包装，正如本节解决方案的方法一样，所以，在 Python 的交互式环境之下，可以直接打印出 Unicode 字符串。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中关于 `codecs` 和 `site` 模块、以及 `sys` 模块中的 `setdefaultencoding`；第 1.20 节和第 1.21 节。

## 1.23 对 Unicode 数据编码并用于 XML 和 HTML

感谢：David Goodger、Peter Cogolo

### 任务

你想对 Unicode 文本进行编码，使用一种有限制，但很流行的编码，如 ASCII 或 Latin-1，并将处理后的结果用于 HTML 输出或者某些 XML 应用。

### 解决方案

Python 提供了一种编码错误处理工具，叫做 `xmlcharrefreplace`，它会将所有不属于所选编码的字符用 XML 的数字字符引用来替代：

```
def encode_for_xml(unicode_data, encoding='ascii'):
    return unicode_data.encode(encoding, 'xmlcharrefreplace')
```

也可以将此法用于 HTML 输出，不过你可能会更喜欢 HTML 的符号实体引用。出于这个目的，需要定义并注册一个自定义的编码错误处理函数。要实现这样一个处理函

数非常简单，因为 Python 标准库已经提供了一个叫做 `htmlentitydefs` 的模块，包含了所有的 HTML 实体定义：

```
import codecs
from htmlentitydefs import codepoint2name
def html_replace(exc):
    if isinstance(exc, (UnicodeEncodeError, UnicodeTranslateError)):
        s = [u'&%s;' % codepoint2name[ord(c)]
              for c in exc.object[exc.start:exc.end] ]
        return ''.join(s), exc.end
    else:
        raise TypeError("can't handle %s" % exc.__name__)
codecs.register_error('html_replace', html_replace)
```

注册完错误处理函数之后，可以再写个包装函数，以简化使用：

```
def encode_for_html(unicode_data, encoding='ascii'):
    return unicode_data.encode(encoding, 'html_replace')
```

## 讨论

如同其他的一些 Python 模块一样，这个模块也将提供一个测试的示例；由 `if __name__ == '__main__'` 这一行语句进行保护：

```
if __name__ == '__main__':
    # demo
    data = u'''\
<html>
<head>
<title>Encoding Test</title>
</head>
<body>
<p>accented characters:
<ul>
<li>\xe0 (a + grave)
<li>\xe7 (c + cedilla)
<li>\xe9 (e + acute)
</ul>
<p>symbols:
<ul>
<li>\xa3 (British pound)
<li>\u20ac (Euro)
<li>\u221e (infinity)
</ul>
</body></html>
'''

    print encode_for_xml(data)
    print encode_for_html(data)
```

如果将此模块作为主脚本来运行，你会看到如下的输出（来自于 `encode_for_xml`）：

```
<li>&#224; (a + grave)
<li>&#231; (c + cedilla)
<li>&#233; (e + acute)

...
<li>&#163; (British pound)
<li>&#8364; (Euro)
<li>&#8734; (infinity)
```

还有这些（来自于 `encode_for_html`）：

```
<li>&grave; (a + grave)
<li>&ccedil; (c + cedilla)
<li>&acute; (e + acute)

...
<li>&pound; (British pound)
<li>&euro; (Euro)
<li>&infin; (infinity)
```

这两段输出都很清晰，不过 `encode_for_xml` 更加具有通用性（它可以用于任何 XML 应用，不仅仅是 HTML），但 `encode_for_html` 却能够生成更易读的结果——如果希望直接读取或者编辑那些结果。如果给浏览器提供这两种形式的数据，能够看到渲染输出实际上是一样的。为了看到这两种方式在浏览器中的展示，可以将上述代码作为主脚本运行，将输出导向到一个磁盘文件，并用文本编辑器将这两种输出分隔开来，然后用浏览器分别查看。（再或者，运行脚本两次，一次将调用 `encode_for_xml` 的输出注释掉，一次将 `encode_for_html` 的输出注释掉）。

请记住，Unicode 数据在被打印或者写到文件之前一定要先编码。由于 UTF-8 能够处理任何 Unicode 字符，所以它是理想的编码。但对很多用户和应用而言，ASCII 或 Latin-1 比 UTF-8 更受欢迎。当 Unicode 数据包含了指定编码之外的字符时（比如，一些重音字符和很多符号大多不在 ASCII 或 Latin-1 之中，比如，Latin-1 无法表现“无穷”符号），单靠这些编码本身，根本无法处理这些数据。Python 提供一个内建的编码错误处理函数，叫做 `xmlcharrefreplace`，将所有的不能被编码的字符替换为 XML 的数字字符引用，比如“`&#8734;`”，表示“无穷”符号。本节还展示了怎样编写和注册另一个类似的错误处理函数 `html_replace`，针对 HTML 输出进行处理。`html_replace` 将不能编码的字符替换为更具可读性的 HTML 符号实体引用，比如用“`&infin;`”来表示“无穷”符号。`html_replace` 相比于 `xmlcharrefreplace`，它的通用性没那么好，因为它并不支持所有的 Unicode 字符，同时也不能用于非 HTML 的应用；但如果你希望 HTML 输出的源码文件能够具有更好的可读性，`html_replace` 是非常有用的。

如果输出的不是 HTML 或者任何形式的 XML，这些错误处理函数就没什么意义了。比如，TeX 和其他的标记语言并不认识 XML 的数字字符引用，但如果你知道怎样创建一个该标记语言中的字符引用，可以修改本节示例中的错误处理函数 `html_replace`，

并注册一个新的符合需求的处理函数。

当需要将 Unicode 数据写入文件时，可以使用 Python 标准库的 `codecs` 模块提供的另一个（非常高效）可以指定编码和错误处理函数的方法：

```
outfile = codecs.open('out.html', mode='w', encoding='ascii',
                      errors='html_replace')
```

现在，可以将 `outfile.write(unicode_data)` 应用于任何 Unicode 字符串 `unicode_data`，所有的编码和错误处理都会透明地自动进行。当然，在输出完成之后，还得调用 `outfile.close()`。

## 更多文档

*Library Reference* 和 *Python in a Nutshell* 中关于 `codecs` 模块和 `htmlentitydefs`。

## 1.24 让某些字符串大小写不敏感

感谢：Dale Strickland-Clark、Peter Cogolo、Mark McMahon

### 任务

你想让某些字符串在比较和查询的时候是大小写不敏感的，但在其他操作中却保持原状。

### 解决方案

最好的解决方式是，将这种字符串封装在 `str` 的一个合适的子类中：

```
class iStr(str):
    """
        大小写不敏感的字符串类
        行为方式类似于 str，只是所有的比较和查询
        都是大小写不敏感的
    """
    def __init__(self, *args):
        self._lowered = str.lower(self)
    def __repr__(self):
        return '%s(%s)' % (type(self).__name__, str.__repr__(self))
    def __hash__(self):
        return hash(self._lowered)
    def lower(self):
        return self._lowered
    def _make_case_insensitive(name):
        ''' 将 str 的方法封装成 iStr 的方法，大小写不敏感 '''
        str_meth = getattr(str, name)
        def x(self, other, *args):
            if isinstance(other, iStr):
                other = other._lowered
            return str_meth(self, other, *args)
        return x
```

```

    ... 先尝试将 other 小写化，通常这应该是一个字符串，
        但必须要做好准备应对这个过程中出现的错误，
        因为字符串是可以和非字符串正确地比较的
    ...

    try: other = other.lower( )
    except (TypeError, AttributeError, ValueError): pass
    return str_meth(self._lowered, other, *args)
# 仅 Python 2.4，增加一条语句: x.func_name = name
setattr(iStr, name, x)
# 将 _make_case_insensitive 函数应用于指定的方法
for name in 'eq lt le gt gt ne cmp contains'.split( ):
    _make_case_insensitive('__%s__' % name)
for name in 'count endswith find index rfind rindex startswith'.split( ):
    _make_case_insensitive(name)
# 注意，我们并不修改 replace、split、strip 等方法
# 当然，如果有需要，也可以对它们进行修改
del _make_case_insensitive # 删除帮助函数，已经不再需要了

```

## 讨论

iStr 类的一些实现上的选择很值得讨论。首先，我们在`__init__`中一次性生成了小写版本，这是因为我们认识到在 iStr 的典型应用中，这个小写版本将会被反复地使用。我们在一个私有的变量中保存这个小写版本，将其作为一个属性，当然，也别保护得太过分了（它以一个下划线开头，而不是两个下划线），因为如果从 iStr 再派生子类（比如，进一步对其扩展，支持大小写不敏感的切分和替换等，正如“解决方案”注释中所说的），iStr 的子类很有可能会需要访问其父类 iStr 的一些关键的“实现细节”。

这里我们没有提供其他一些方法的大小写不敏感的版本，如 `replace`，因为这个例子已经清晰地展示了一种通用的建立输入和输出之间联系的方式。根据应用进行特别定制的子类将提供最能够满足需求的功能。比如，`replace` 方法并没有被封装，则我们对一个 iStr 的实例调用 `replace`，返回的是 `str` 的实例，而不是 `iStr`。如果这会给你的应用带来问题，可以将所有的返回字符串的 iStr 方法封装起来，这就可以确保所有返回的结果是 `iStr` 的实例。基于这个目的，需要另一个单独的助手函数，相似但不完全等同于解决方案中给出的`_make_case_insensitive`：

```

def _make_return_iStr(name):
    str_meth = getattr(str, name)
    def x(*args):
        return iStr(str_meth(*args))
    setattr(iStr, name, x)

```

需要对所有返回字符串的方法的名字应用这个助手函数，`_make_return_iStr`：

```

for name in 'center ljust rjust strip lstrip rstrip'.split( ):
    _make_return_iStr(name)

```

字符串有约 20 种方法（包括一些特殊方法，比如`__add__`和`__mul__`），需要考虑哪

些方法应该被封装起来。也可以把一些额外的方法，比如 `split` 和 `join`（它们可能需要一些特别的处理）封装起来，或者其他的方法，如 `encode` 和 `decode`，对于此类方法，除非定义了一个大小写不敏感的 `unicode` 子类型，否则无法处理它们。而实际上，针对一个特定的应用，可能不是所有的未封装的方法都会引起问题。正如你所见的那样，由于 Python 字符串的方法和功能很丰富，要想用一种通用的不依赖于特定应用的方式，完全彻底地定制出一个子类型，还是要花点功夫的。

`iStr` 的实现很谨慎，主要是为了避免一些重复性的例行公事般的代码（通常是冗长且容易滋生 `bug` 的代码），如果我们用普通的方式重载 `str` 每一个需要的方法，在类的实现中写上一堆 `def` 语句，很有可能就会陷入这种尴尬的境地。使用可自定义的元类或者其他高级技术对这个例子而言也不会有什么特别的优势，但使用一个辅助函数来生成和安装封装层闭包，就可以轻易地避开问题。然后我们在两个循环中使用该辅助函数，一个循环处理常用的方法，另一个则处理特殊的方法。这两个循环都必须被放置在 `class` 语句之后，正如我们在解决方案中给出的代码所示，这是因为这两个循环需要修改 `iStr` 类对象，但除非用 `class` 语句完成对 `iStr` 类的声明，否则那个类对象根本就不存在（因此当然也无法修改）。

在 Python 2.4 中，可以重新指定函数对象的 `func_name` 属性，在本例中，当对 `iStr` 实例应用内省机制时，可以用这种方法让代码变得更加清晰和易读。但在 Python 2.3 中，函数对象的 `func_name` 属性是只读的。因此，在本节的讨论中，我们仅仅是指出了一种可能性，我们不想因为这个小问题失去对 Python 2.3 的兼容性。

大小写不敏感（但仍保留了大小写信息）的字符串有很多用途，包括提高对用户输入进行解析的宽松度，在文件系统（比如 Windows 和 Macintosh 的文件系统）中查找名字包含指定字符的文件，等等。你可能会发现，有很多地方需要“大小写不敏感”的容器类型，比如字典、列表、集合等——它们都需要在某些场合，忽略掉 `key` 或者子项的大小写的信息。很明显，一个好的方法是一次性构建出“大小写不敏感”的比较和查询功能；现在你的工具箱中已经增加了本节提供的解决方案，可以对字符串进行任何需要的封装和定制，你甚至还能定制其他一些你希望具备“大小写不敏感”能力的容器类型。

比如，一个所有子项都是字符串的列表，你希望能够进行一些大小写无关的处理（如用 `count` 和 `index` 进行排序），完全可以基于 `iStr` 的实现，轻易地构建一个 `iList`：

```
class iList(list):
    def __init__(self, *args):
        list.__init__(self, *args)
        # 依赖__setitem__将各项封装为iStr
        self[:] = self
        wrap_each_item = iStr
    def __setitem__(self, i, v):
        if isinstance(i, slice): v = map(self.wrap_each_item, v)
```

```

        else: v = self.wrap_each_item(v)
        list.__setitem__(self, i, v)
    def append(self, item):
        list.append(self, self.wrap_each_item(item))
    def extend(self, seq):
        list.extend(self, map(self.wrap_each_item, seq))

```

本质上，我们做的事情是把 `iList` 实例中每个子项都通过调用 `iStr` 来封装，其余部分则保持原状。

另外提一句，`iList` 的实现方式使得可以根据应用提供特定的 `iStr`，轻易地完成对子类的定制：只需在 `iList` 的子类中重载成员变量 `wrap_each_item` 即可。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中关于 `str` 的章节，主要是字符串方法，以及一些特殊的用于比较和哈希的方法。

## 1.25 将HTML文档转化为文本显示到UNIX终端上

感谢：Brent Burley、Mark Moraes

### 任务

需要将 HTML 文档中的文本展示在 UNIX 终端上，同时还要支持粗体和下划线的显示。

### 解决方案

最简单的方法是写一个过滤的脚本，从标准输入接收 HTML，将输出文本和终端控制序列打印到标准的输出上。由于本节的问题只针对 UNIX，我们可以借助 Python 标准库的 `os` 模块提供的 `popen` 函数，通过 UNIX 的命令 `tput` 获取所需的终端控制序列：

```

#!/usr/bin/env python
import sys, os, htmllib, formatter
# 使用UNIX的tput来获得粗体、下划线和重设的转义序列
set_bold = os.popen('tput bold').read()
set_underline = os.popen('tput smul').read()
perform_reset = os.popen('tput sgr0').read()
class TtyFormatter(formatter.AbstractFormatter):
    '''一个保留粗体和斜体状态的格式化对象，并输出
    相应的终端控制序列
    '''
    def __init__(self, writer):
        # 首先，像往常一样，初始化超类
        formatter.AbstractFormatter.__init__(self, writer)

```

```
# 一开始既没有粗体也没有斜体状态，未保存任何信息
self.fontState = False, False
self.fontStack = [ ]
def push_font(self, font):
    # font 元组有 4 项，我们只看与粗体和斜体的状态
    # 有关的两个标志
    size, is_italic, is_bold, is_tt = font
    self.fontStack.append((is_italic, is_bold))
    self._updateFontState( )
def pop_font(self, *args):
    # 回到前一个 font 状态
    try:
        self.fontStack.pop( )
    except IndexError:
        pass
    self._updateFontState( )
def updateFontState(self):
    # 输出正确的终端控制序列，如果粗体和/或斜体（==underline）
    # 的状态被刚刚改变的话
    try:
        newState = self.fontStack[-1]
    except IndexError:
        newState = False, False
    if self.fontState != newState:
        # 相关的状态改变：重置终端
        print perform_reset,
        # 如果需要的话，设置下划线与/或粗体状态
        if newState[0]:
            print set_underline,
        if newState[1]:
            print set_bold,
        # 记住当前的两个状态
        self.fontState = newState
# 生成写入、格式化、解析对象，根据需要将它们连接起来
myWriter = formatter.DumbWriter( )
if sys.stdout.isatty( ):
    myFormatter = TtyFormatter(myWriter)
else:
    myFormatter = formatter.AbstractFormatter(myWriter)
myParser = htmllib.HTMLParser(myFormatter)
# 将标准输入和终端操作提供给解析器
myParser.feed(sys.stdin.read( ))
myParser.close( )
```

## 讨论

Python 标准库提供的 `formatter.AbstractFormatter` 类，可以在任何场合工作。另一方面，它的子类 `TtyFormatter` 提供的一些改良，则主要是为了操纵和使用类 UNIX(UNIX-like)

终端，具体地说，也就是通过 UNIX 命令 `tput` 获取控制粗体和下划线的转义序列，并将终端重置为基本状态。

很多系统并没有通过 UNIX 认证，比如 Linux 和 Mac OS X，但它们也提供了一个可用的 `tput` 命令，因此本节的 `TtyFormatter` 子类在这样的系统中仍然可以正常工作。或者说，可以用更宽泛的眼光来看待本节提及的“UNIX”，就好像我们在一些其他讨论中所用的“UNIX”概念：如果你愿意，可以认为它指的是“\*ix”。

如果你的“终端”模拟器支持其他的一些控制输出表现的转义序列，也可以根据情况修改 `TtyFormatter` 类。比如，据说在 Windows 中，`cmd.exe` 命令能够支持所有标准的 ANSI 转义序列，所以如果你只想在 Windows 上运行你的脚本，可以用硬编码的方式在类中写入那些序列。

很多时候，你可能会更喜欢用 UNIX 已经提供的命令，比如 `lynx -dump -`，相比于本节方案中提供的方法，这些命令能够提供更丰富更具表现力的输出。但有时你会发现 Python 安装在一个不提供这种有用的命令（如 `lynx`）的系统上，那么本节给出的方法就显得很方便和简洁了。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 文档中关于 `formatter` 和 `htmllib` 模块的内容；在 UNIX 或者类 UNIX 系统中用 `man` 命令查看 `tput` 命令的有关信息。

# 文件

## 引言

感谢：Mark Lutz，《Programming Python》和《Python Quick Reference》的作者，*Learning Python* 的合著者之一

任何一个有经验的程序员接触一门新语言时，都会首先在该语言的工具箱中寻找文件的相关工具。因为处理外部文件是一种非常实用和常见的任务，该语言的文件处理接口设计的好坏，在很大程度上决定着这门语言工具的实用性。

正如本章各节将要介绍那样，Python 在这方面非常优秀。Python 中对文件的支持，体现在很多层次上：从内建的 `open` 函数（标准文件对象类型的一个同义词），到标准库模块提供的一些特定的工具，比如 `os`，再到遍布网络的各种第三方提供的实用工具。一言以蔽之，Python 强大的文件工具库提供了各种方法来访问文件。

## 文件基础

在 Python 中，文件对象是内建类型 `file` 的实例。内建函数 `open` 会创建并返回一个文件对象。第一个参数是一个字符串，指定了文件路径（文件名之前有一个可选的目录路径）。`open` 的第二个参数也是个字符串，指定了打开文件的模式。比如：

```
input = open('data', 'r')
output = open('/tmp/spam', 'w')
```

`open` 接受由斜线字符 (/) 分隔开的目录和文件名构成的文件路径，而完全不管操作系统本身的倾向。在不使用斜线的系统中，可以使用反斜线字符 (\)，不过如果没有什么好理由的话最好不要这么做。反斜线在字符串中的文本表示不太美观，你得用两个反斜线来表示单个斜线字符或者用“原”字符串 (raw string)。如果文件路径参数中没有包括文件的路径，则该文件被认为处于当前工作路径中（这个路径可能游离于 Python 模块的搜索路径之外）。

至于模式参数，使用'r'意味着以文本模式读取文件；这是默认的值而且常常被忽略，所以 `open` 也可以只使用第一个参数。其他常见的模式是'rb'，表示以二进制模式读取文件，'w'表示创建文件并以文本模式写文件，'wb'表示创建并以二进制模式写文件。'r'的一种变体是'rU'，意味着将以支持“通用换行符”（universal newline）的文本模式读取文件：'rU'模式可以用一种独立于该文件所用的断行约定的方式来读取文件，可以是 UNIX 方式，也可以是 Windows 方式，甚至是（老的）Mac 方式。（今天的 Mac OS X 从各方面来讲都可以算是 UNIX，但前几年的 Mac OS 9 还完全不一样）

对于那些非类 UNIX 平台（non-UNIX-like platforms），文本模式和二进制模式之间的区别非常重要，这和那些系统所用的行结束标记有关。当你以二进制模式打开一个文件，Python 知道它无须关心行结束标记；它只是在文件和内存中的字符串之间移动字节，不需要任何翻译转化。但当你在一个非类 UNIX 系统中以文本模式打开一个文件时，Python 知道它必须在字符串所用的“\n”换行符和当前系统所用的行结束标记之间做翻译和转化工作。只要你能正确地指定你打开的文件的模式，你的所有的 Python 代码总可以依赖“\n”作为行结束标记。

一旦有了一个文件对象，你就可以执行该对象的各种 I/O 操作，我们接下来会介绍那些操作。当你完成了处理，应该对该对象调用 `close` 方法来完成收尾工作，关闭所有和这个文件对象的联系：

```
input.close()
```

在一些短的脚本中，用户常常忽略这个步骤，因为当一个文件对象在垃圾回收阶段被收回的时候（在主流的 Python 中这意味着该文件会被立即关闭，但其他的一些重要的 Python 实现，比如 Jython 和 IronPython，则有其他更宽松的垃圾回收策略），Python 会自动关闭该文件对象。不管怎样，用完文件对象之后尽可能马上关闭文件是一个好的编程习惯，尤其是在写一些大型程序的时候，不然你总会让一些无用的打开的文件对象占着内存资源。注意，`Try/finally` 在确保文件被关闭时非常好用，即使函数抛出了一个无法处理的异常，并因此而终止。

在写入文件的时候，使用 `write` 方法：

```
output.write(s)
```

假如 `s` 是一个字符串，当 `output` 是用文本模式打开的时候，可以将 `s` 看成是字符串，当 `output` 以二进制模式打开的时候，则可以将 `s` 看做是字节串。文件还有其他的一些与写入有关的方法，比如 `flush`，可以发送出缓存中所有的数据，还有 `writelines`，一次调用可以将整个字符串序列写入。不过，`write` 仍然是最常用的方法。

从文件中读取数据比把数据写入文件更常见，也涉及更多的内容，因此文件对象的有关读取的方法要比写入的方法多。`readline` 方法可从一个文本文件中读取并返回一行文本数据。考虑这样一个循环：

```
while True:  
    line = input.readline()  
    if not line: break  
    process(line)
```

在 Python 中，这曾经是从一个文件读取并处理每行数据的最惯用的方式，但现在已经不是了。另一个过时的方式是使用 `readlines` 方法，一次读完整个文件，并返回一个各行数据的列表：

```
for line in input.readlines():  
    process(line)
```

`readlines` 方法只有在物理内存足够用的情况下才会很有用。如果文件非常庞大，`readlines` 可能会失败，或者性能急剧降低（虚拟内存不足，操作系统会将物理内存中的数据复制到磁盘上）。在现在的 Python 中，只需对这个文件对象执行一个循环，每次取得一行并处理，这样可获得更好的性能和效率：

```
for line in input:  
    process(line)
```

当然，也不总是逐行的读取一个文件。可以读取文件中的一些或者所有的字节，特别是在用二进制模式读取的情况下。对于二进制数据，行是一个没有意义的概念。根据这种情况，可以使用 `read` 方法。当不指定参数时，`read` 会读取并返回文件中所有剩余的字节。当 `read` 被传入一个整数参数 `N` 并调用时，它读取并返回下 `N` 个字节（或者所有剩余的字节，如果剩余的字节数少于 `N`）。其他的值得一提的是 `seek` 和 `tell` 方法，支持对文件的随机访问。在处理包含很多固定长度记录的二进制文件时，这些方法很常用。

## 可移植性和灵活性

从表面看，Python 对文件的支持非常直接。然而，在你仔细阅读本章的代码之前，我想指出关于 Python 文件支持的两个重要方面：代码的可移植性和接口的灵活性。

请记住，Python 中的大多数文件接口都是完全跨平台的。这种设计的重要性怎么赞美也不为过。比如，在一个在目录树中搜索所有文件中的文本的脚本，可以一点代码不改地运行在各个平台上：只需要将脚本文件复制到目标机器上。我一直都是这么做的——完全不用关心操作系统的差异性。由于 Python 的强大的可移植性，运行平台几乎成为一个无足轻重的问题。

还有，Python 的文件处理接口往往并不局限在只用于真实的物理文件，这也让我很吃惊。实际上，大多数文件工具也可以用于那些暴露的接口与文件对象类似的任何类型的对象。因此，文件读取者只需要关心读取方法，文件写入者则只需关心写入方法。只要目标对象实现了期望的协议，一切工作都可以平滑自然地进行。

比如，假设你要写一个通用的文件处理函数，像下面这样，传入一个函数来处理输入

文件中的每行数据：

```
def scanner(fileobject, linehandler):
    for line in fileobject:
        linehandler(line)
```

如果将这个函数写入一个模块文件，并将该文件放入包括在 Python 搜索路径（`sys.path`）中的一个“目录”里，就可以在任何时候调用该函数来逐行扫描文本文件，无论何时。为了展示这一点，下面给出一个客户脚本，只是简单地打印出每行的第一个单词：

```
from myutils import scanner
def firstword(line):
    print line.split()[0]
file = open('data')
scanner(file, firstword)
```

至此为止，一切都很好；我们完成了一个小的可复用的软件组件。但请注意，在 `scanner` 函数中并没有对类型有任何假设，只要是能被逐行迭代的满足接口要求的对象就可以。比如，假如你以后想测试一些来自于字符串对象的输入，而不是真实的物理文件。标准的 `StringIO` 模块，以及等价但更快速的 `cStringIO`，提供了一种适用的封装和类似的接口：

```
from cStringIO import StringIO
from myutils import scanner
def firstword(line): print line.split()[0]
string = StringIO('one\nntwo xxx\nnthree\nn')
scanner(string, firstword)
```

`StringIO` 对象完全兼容文件对象，即插即用，所以，`scanner` 从内存中的一个字符串对象中读取三行文本，而不是从一个真正的外部文件中读取。完全不用更改原先的 `scanner` 来完成任务，只需传递一个正确的对象。为了更具通用性，还可以实现一个类并提供 `scanner` 期望的接口：

```
class MyStream(object):
    def __iter__(self):
        # 获取并返回字符串
        return iter(['a\n', 'b c d\n'])
from myutils import scanner
def firstword(line):
    print line.split()[0]
object = MyStream()
scanner(object, firstword)
```

这次，当 `scanner` 尝试读取文件时，它实际调用的是在类中实现的 `__iter__` 方法。在实践中，这个方法可以借助 Python 标准工具从各种源抓取文本：一个交互的用户，一个弹出式的图形界面输入框，一个 `shelve` 对象，一个 SQL 数据库，一页 XML 或者 HTML，一个网络 `socket` 等。关键点在于，`scanner` 根本就不知也不关心究竟是什么类型的对象实现了它期望的接口，以及这些接口究竟在背地里干什么。

面向对象语言的程序员都知道一个重要的概念，多态。被处理对象的类型决定了究竟

进行什么样的操作——比如 scanner 中的循环迭代。而在 Python 中，对象的接口，而非类型，才是真正的连接器。这种机制造成的效果是一个函数的应用面往往要比你设想的广得多。特别是如果你有一些静态类型语言如 C 或 C++ 的背景，对此会有更深的体会。就好像我们在 Python 中突然获得了免费的 C++ 模板。Python 的强大的动态类型带来的一个副产品是，代码具有了与生俱来的灵活性。

当然，代码的可移植性和灵活性在 Python 的开发中无处不在，不仅仅局限于文件接口。这两者都是语言本身的特性，这段文件处理脚本只是简单地继承了这些特性。其他的 Python 的优点，还包括代码容易编写和易读，当需要修改你的文件处理程序的时候，你就能体会到这些好处了。Python 除了这些值得赞美的优点之外，在本章和本书中还有很多精彩的内容和细节值得你去发掘和探索。阅读愉快！

## 2.1 读取文件

感谢：Luther Blissett

### 任务

你想从文件中读取文本或数据。

### 解决方案

最方便的方法是一次性读取文件中的所有内容并放置到一个大字符串中：

```
all_the_text = open('thefile.txt').read()      # 文本文件中的所有文本  
all_the_data = open('abinfile','rb').read()    # 二进制文件中的所有数据
```

为了安全起见，最好还是给打开的文件对象指定一个名字，这样在完成操作之后可以迅速关闭文件，防止一些无用的文件对象占用内存。举个例子，对文本文件读取：

```
file_object = open('thefile.txt')  
try:  
    all_the_text = file_object.read()  
finally:  
    file_object.close()
```

不一定要在这里用 Try/finally 语句，但是用了效果更好，因为它可以保证文件对象被关闭，即使在读取中发生了严重错误。

最简单、最快，也最具 Python 风格的方法是逐行读取文本文件内容，并将读取的数据放置到一个字符串列表中：

```
list_of_all_the_lines = file_object.readlines()
```

这样读出的每行文本末尾都带有 “\n” 符号；如果你不想这样，还有另一个替代的办法，比如：

```
list_of_all_the_lines = file_object.read().splitlines()
list_of_all_the_lines = file_object.read().split('\n')
list_of_all_the_lines = [L.rstrip('\n') for L in file_object]
```

最简单最快的逐行处理文本文件的方法是，用一个简单的 for 循环语句：

```
for line in file_object:
    process line
```

这种方法同样会在每行末尾留下 “\n” 符号，可以在 for 循环的主体部分加一句：

```
line = line.rstrip('\n')
```

或者，你想去除每行的末尾的空白符（不只是’\n’），常见的办法是：

```
line = line.rstrip()
```

## 讨论

除非要读取的文件非常巨大，不然一次性读出所有内容放进内存并进一步处理是最快和最方便的办法。内建函数 `open` 创建了一个 Python 的文件对象（另外，也可以通过调用内建类型 `file` 创建文件对象）。你对该对象调用 `read` 方法将读出所有内容（无论是文本还是二进制数据），并放入一个大字符串中。如果内容是文本，可以选择用 `split` 方法或者更专用的 `splitlines` 将其切分成一个行列表。由于切分字符串到单行是很常见的需求，还可以直接对文件对象调用 `readlines`，进行更方便更快速的处理。

可以直接对文件对象应用循环语句，或者将它传递给一个需要可迭代对象的处理器，比如 `list` 或者 `max`。当它被当做可迭代对象处理时，一个被打开并被读取的文件对象中的每一个文本行都变成了迭代子项（因此，这也只适用于文本文件）。这种逐行迭代的处理方式很节省内存资源，速度也不错。

在 UNIX 或者类 UNIX 系统中，比如 Linux, Mac OS X, 或者其他 BSD 变种，文本文档和二进制文件其实并没有什么区别。在 Windows 和老的 Macintosh 系统中，换行符不是标准的 ‘\n’，而是分别是 ‘\r\n’ 和 ‘\r’。Python 会帮助你把这些换行符转化成 ‘\n’。这意味着当你打开二进制文件时，需要明确告诉 Python，这样它就不会做任何转化。为了达到这个目的，必须传递 ‘rb’ 给 `open` 的第二个参数。在类 UNLX 平台上，这么做也不会有什么坏处，而且总是区分文本文档和二进制文件是一个好习惯，当然在那些平台上这并不是强制性的要求。不过这些好习惯会让你的程序具有更好的可读性，也更易于理解，同时还能具有更好的平台兼容性。

如果不确定某文本文档会用什么样的换行符，可以将 `open` 的第二个参数设定为 ‘rU’，指定通用换行符转化。这让你可以自由地在 Windows、UNIX（包括 Mac OS X），以及其他的老 Macintosh 平台上交换文件，完全不用担心任何问题：无论你的代码在什么平台上运行，各种换行符都被映射成 ‘\n’。

可以对 `open` 函数产生的文件对象直接调用 `read` 方法，如解决方案中给出的第一个代码片段所示。当你这么做的时候，你在完成读取的同时，也失去了对那个文件对象的引用。在实践中，Python 注意到了这种当场即时失去引用的情况，它会迅速关闭该文件。然而，更好的办法仍然是给 `open` 产生的结果指定一个名字，这样当你完成了处理，可以显式地自行关闭该文件。这能够确保该文件处于被打开状态的时间尽量的短，即使是在 Jython，IronPython 或其他变种 Python 平台上（这些平台的高级垃圾回收机制可能会推迟自动回收，不像现在的基于 C 的 Python 平台，CPython 会立刻执行回收）。为了确保文件对象即使在处理过程发生错误的情况下仍能够正确关闭，应该使用 `try/finally` 语句，这是一种稳健而严谨的处理方式。

```
file_object = open('thefile.txt')
try:
    for line in file_object:
        process line
finally:
    file_object.close()
```

注意，不要把对 `open` 的调用放入到 `try/finally` 语句的 `try` 子句中（这是初学者很常见的错误）。如果在打开文件的时候就发生了错误，那就没有什么东西需要关闭，而且，也没有什么实质性的东西绑定到了 `file_object` 这个名字上，当然也就不应该调用 `file_object.close()`。

如果选择一次读取文件的一小部分，而不是全部，方式就有点不同了。下面给出一个例子，一次读取一个二进制文件的 100 个字节，一直读到文件末尾：

```
file_object = open('abinfile', 'rb')
try:
    while True:
        chunk = file_object.read(100)
        if not chunk:
            break
        do_something_with(chunk)
finally:
    file_object.close()
```

给 `read` 方法传入一个参数 `N`，确保了 `read` 方法只读取下 `N` 个字节（或更少，如果读取位置已经很接近文件末尾的话）。当抵达文件末尾时，`read` 返回空字符串。复杂的循环最好被封装成可复用的生成器（generator）。对于这个例子，我们只能将其逻辑的一部分进行封装，这是因为生成器（generator）的 `yield` 关键字不被允许出现在 `try/finally` 语句的 `try` 子句中。如果要抛弃 `try/finally` 语句对文件关闭的保护，我们可以这么做：

```
def read_file_by_chunks(filename, chunksize=100):
    file_object = open(filename, 'rb')
    while True:
```

```
chunk = file_object.read(chunksize)
if not chunk:
    break
yield chunk
file_object.close()
```

一旦 `read_file_by_chunks` 生成器完成，以固定长度读取和处理二进制文件的代码就可以写得极其简单：

```
for chunk in read_file_by_chunks('abinfile'):
    do_something_with(chunk)
```

逐行读取文本文件的任务更为常见。只需对文件对象应用循环语句，如下：

```
for line in open('thefile.txt', 'rU'):
    do_something_with(line)
```

为了 100% 确保完成操作之后没有无用的已打开的文件对象存在，可以将上述代码修改得更加严密稳固：

```
file_object = open('thefile.txt', 'rU')
try:
    for line in file_object:
        do_something_with(line)
finally:
    file_object.close()
```

## 更多资料

第 2.2 节；*Library Reference* 和 *Python in a Nutshell* 中关于内建 `open` 函数和文件对象的内容。

## 2.2 写入文件

感谢：Luther Blissett

### 任务

你想写入文本或者二进制数据到文件中。

### 解决方案

下面是最方便的将一个长字符串写入文件的办法：

```
open('thefile.txt', 'w').write(all_the_text) # 写入文本到文本文件
open('abinfile', 'wb').write(all_the_data) # 写入数据到二进制文件
```

不过，最好还是给文件对象指定个名字，这样你就可以在完成操作之后调用 `close` 关闭

文件对象。比如，对一个文本文件：

```
file_object = open('thefile.txt', 'w')
file_object.write(all_the_text)
file_object.close()
```

可是，很多时候想写入的数据不是在一个大字符串中，而是在一个字符串列表（或其他序列）中。为此，应该使用 `writelines` 方法（不要望文生义，这个方法并不局限于行写入，而且二进制文件和文本文件都适用）：

```
file_object.writelines(list_of_text_strings)
open('abinfile', 'wb').writelines(list_of_data_strings)
```

当然也可以先把子串拼接成大字符串（比如用`"join"`）再调用 `write` 写入，或者在循环中写入，但直接调用 `writelines` 要比上面两种方式快得多。

## 讨论

要创建一个用于写入的文件，必须将 `open`（或文件对象）的第二个参数指定为“`w`”以允许写入文本数据，或者“`wb`”以允许写入二进制数据。当你试图写入而不是读取文件的时候，你更应该重视 2.1 节提到的关闭文件的建议。只有在文件被正确关闭之后，你才能确信数据被写入了磁盘，而不是暂存于内存中的临时缓存中。

分批次将数据写入文件的应用，甚至比分批次从文件中读取数据的应用更为常见。只需准备妥当字符串或者字符串序列，然后反复地调用 `write` 或 `writelines` 即可。每个 `write` 操作都会在文件末尾增添新数据，紧随已经写入的旧数据。当你完成了写入工作，可调用 `close` 方法来关闭文件对象。如果你能够一次准备好所有需要写入的数据，可以调用 `writelines` 来一次性完成写入任务，这样更快也更简单。但如果你每次只能准备好一部分数据，那么最好分批次写入。比起另一个方法，即，先在内存中建立一个大的临时数据序列用于存储所有的数据，然后用 `writelines` 一次性将所有数据写入文件，分批次写入明显要更好一些。读取和写入这两种操作，在一次性大数据处理和分批次小数据处理这两个方面，性能和方便程度都表现出很大的区别。

当用“`w`”（或“`wb`”）选项打开一个文件准备写入数据的时候，文件中原有的数据都将被清除；即使在打开之后迅速关闭，得到的仍然是一个空文件。如果你想把新数据添加在原有的数据之后，应该用“`a`”（或“`ab`”）选项来打开文件。还有一些更高级的选项可允许你对同一个文件同时进行读取和写入操作，见第 2.8 节中提到的“`r+b`”选项，那实际上也是高级选项中最常用的一个。

## 更多资料

第 2.1 节；第 2.8 节；*Library Reference* 和 *Python in a Nutshell* 中关于内建 `open` 函数和文件对象的内容。

## 2.3 搜索和替换文件中的文本

感谢：Jeff Bauer, Adam Krieg

### 任务

需要将文件中的某个字符串改变成另一个。

### 解决方案

字符串对象的 `replace` 方法提供了字符串替换的最简单的办法。下面的代码支持从一个特定的文件（或标准输入）读取数据，然后写入一个指定的文件（或标准输出）：

```
#!/usr/bin/env python
import os, sys
nargs = len(sys.argv)
if not 3 <= nargs <= 5:
    print "usage: %s search_text replace_text [infile [outfile]]" % \
        os.path.basename(sys.argv[0])
else:
    stext = sys.argv[1]
    rtext = sys.argv[2]
    input_file = sys.stdin
    output_file = sys.stdout
    if nargs > 3:
        input_file = open(sys.argv[3])
    if nargs > 4:
        output_file = open(sys.argv[4], 'w')
    for s in input_file:
        output_file.write(s.replace(stext, rtext))
    output.close()
    input.close()
```

### 讨论

本节给出的解决方案非常简单，但那也正是精彩的地方——如果简单的东西已经够用了，为什么要用复杂的东西？正如开始的“shebang”（shell 头描述）行所示，这个脚本是一个简单的主脚本，而不是那些被用来导入的模块，它直接运行在一个 shell 命令行提示中。脚本检查传递给它的参数以确定要搜索的文本，用于替代的文本，输入文件（默认是标准输入），输出文件（默认是标准输出）。然后循环遍历输入文件中的每一行，完成了对每行文本的字符串替换之后，再写入到输出文件。这就结束了。准确点说，最后还关闭了所有的文件。

如果内存充裕到能够轻松放入两份输入文件（一份是原版的，另一份是完成了替代之后的，因为字符串是不能被改变的，所以必须有两个拷贝），我们就可以进一步地提高速度。一次性完成对所有内容的处理，而不用循环。今天的低端计算机至少都有 256MB

以上的内存，处理一个 100MB 左右的文件并不是什么问题，而且处理超过 100MB 的文件的情况也很少。所以，我们也可以用一行语句替换掉那个循环：

```
output_file.write(input_file.read().replace(stext, rtext))
```

正如你所看到的，简单得不得了。

## 更多资料

参看 *Library Reference* 和 *Python in a Nutshell* 中关于内建 `open` 函数，文件对象，字符串的 `replace` 方法。

## 2.4 从文件中读取指定的行

感谢：Luther Blissett

### 任务

你想根据给出的行号，从文本文件中读取一行数据。

### 解决方案

Python 标准库 `linecache` 模块非常适合这个任务：

```
import linecache  
theline = linecache.getline(thefilepath, desired_line_number)
```

### 讨论

对这个任务而言，标准的 `linecache` 模块是 Python 能够提供的最佳解决工具。当你想要对文件中的某些行进行多次读取时，`linecache` 特别有用，因为 `linecache` 会缓存一些信息以避免重复一些工作。当你不需要从缓存中获得行数据时，可以调用模块的 `clearcache` 函数来释放被用作缓存的内存。当磁盘上的文件发生了变化时，还可以调用 `checkcache`，以确保缓存中存储的是最新的信息。

`linecache` 读取并缓存你指定名字的文件中的所有文本，所以，如果文件非常大，而你只需要其中一行，为此使用 `linecache` 则显得不是那么必要。如果这部分可能是你的程序的瓶颈，可以使用显式的循环，并将其封装在一个函数中，这样可以获得速度上的一些提升，像这样：

```
def getline(thefilepath, desired_line_number):  
    if desired_line_number < 1: return ''  
    for current_line_number, line in enumerate(open(thefilepath, 'rU')):  
        if current_line_number == desired_line_number-1: return line  
    return ''
```

唯一需要注意的细节是 `enumerate` 从 0 开始计数，因此，既然我们假设 `desired_line_number` 参数从 1 开始计算，需要在用 == 比较的时候减去 1。

## 更多资料

参见 *Library Reference* 和 *Python in a Nutshell* 中的 `linecache` 模块；*Perl Cookbook 8.8*。

## 2.5 计算文件的行数

感谢：Luther Blissett

### 任务

需要计算一个文件中有多少行。

### 解决方案

对于尺寸不大的文件，最简单的方式是将文件读取放入一个行列表中，然后计算列表的长度即可。假设文件路径由变量 `filepath` 指定，那么以这种方式实现的代码如下：

```
count = len(open(filepath, 'rU').readlines())
```

对于非常大的文件，这种简单的处理方式极有可能会很慢，甚至会失败。如果你确实担心大文件的问题，用循环来计数是一个可行的办法：

```
count = -1
for count, line in enumerate(open(filepath, 'rU')):
    pass
count += 1
```

如果行结束标记是 “\n”（或者含有 “\n”，就像 Windows 平台），我们还有一个更巧妙的，对于大文件也更快的方式：

```
count = 0
thefile = open(filepath, 'rb')
while True:
    buffer = thefile.read(8192*1024)
    if not buffer:
        break
    count += buffer.count('\n')
thefile.close()
```

给 `open` 的 ‘rb’ 参数是必要的，如果你追求速度，那么没有那个参数，这段代码在 Windows 上的运行可能会比较慢。

### 讨论

如果有外部程序提供文件行统计的功能，比如类 UNIX 平台中的 `wc -l`，你当然也可以

选择使用它们（比如，通过 `os.popen`）。但是，如果能够实现自己的行计算程序，代码通常会更简单、更快，也更具移植性。对于那些大小比较适合的文件，一次全部读取到内存中再处理是最简单的方式。对于这种文件，用 `len` 对 `readlines` 返回的结果计算长度即可获取行数。

如果文件大到超过了可用的内存（比如，几百兆字节，对于今天的典型的个人计算机来说），这种最简单的方式将慢得根本无法接受。操作系统费尽九牛二虎之力，试图把文件内容放入虚拟内存，而且这个过程还可能失败，如果交换区空间耗尽，虚拟内存也无以为继。假设在一个典型的个人计算机上，装有 256MB 内存和无限制的虚拟磁盘，你尝试一次性读取超过 1GB 或 2GB 的文件，需要当心这个过程中可能发生的错误，不过这跟你使用的操作系统还有一些关系。（一些操作系统在极端的高负载压力下处理虚拟内存会比其他系统脆弱得多）在这个场合中，用循环来处理文件对象，如本节解决方案所示，会更好一些。内建的 `enumerate` 函数会自行计算行数，无须你用代码明确指定。

一次读取适量的字节，并计算其中的换行符，这是本节第三个处理方式的思路。这可能不是那么直观，而且也不能很完美地跨平台，但它可能是最快的办法（可以将它和 *Perl Cookbook 8.2* 节比较一下）。

然而，大多数时候，性能不是那么重要。如果性能的确值得考虑，你的第一感直觉也往往不能告诉你程序中真正耗时的代码段是哪个部分，事实上，你绝不应该相信直觉——而应该进行基准测试。比如，有个典型的中等大小的 UNIX `syslog` 文件，18MB 略多一点，230 000 行文本：

```
[situ@tioni nuc]$ wc nuc  
231581 2312730 18508908 nuc
```

考虑下面的基准测试框架脚本，`bench.py`：

```
import time  
  
def timeo(fun, n=10):  
    start = time.clock()  
    for i in xrange(n): fun()  
    stend = time.clock()  
    thetime = stend-start  
    return fun.__name__, thetime  
  
import os  
  
def linecount_w():  
    return int(os.popen('wc -l nuc').read().split()[0])  
def linecount_1():  
    return len(open('nuc').readlines())  
def linecount_2():  
    count = -1  
    for count, line in enumerate(open('nuc')): pass  
    return count+1  
def linecount_3():
```

```
count = 0
thefile = open('nuc', 'rb')
while True:
    buffer = thefile.read(65536)
    if not buffer: break
    count += buffer.count('\n')
return count
for f in linecount_w, linecount_1, linecount_2, linecount_3:
    print f.__name__, f()
for f in linecount_1, linecount_2, linecount_3:
    print "%s: %.2f" % timeo(f)
```

首先，我将各种方法统计行数的结果打印出来，以确保没有什么错误或反常发生（众所周知，行统计任务会因为一点小错而失败）。然后，通过控制和计时函数 `timeo`，我再将各个任务都运行 10 次，并观察结果。在一台可靠的老机器上，我的程序得出如下结果：

```
[situ@tioni nuc] $ python -O bench.py
linecount_w 231581
linecount_1 231581
linecount_2 231581
linecount_3 231581
linecount_1:4.84
linecount_2:4.54
linecount_3:5.02
```

正如你所见的，性能差异几乎可以忽略：用户对这类辅助性任务从来都感觉不出 10% 的性能差异。然而，最快的方式却是简单朴实地循环遍历每一行（我的测试环境是一台老旧但可靠的个人计算机，运行着一个流行的 Linux 发行版本），最慢的竟然是更具技巧性的逐次读取数据并计算换行符的方式。在实践中，除非需要处理非常大的文件，我一般总会选择最简单的方式（本节提供的第一个方法）。

准确地衡量代码的性能（要比盲目使用一些复杂的方式并寄希望能提高性能好的多）非常重要——重要到 Python 标准库要专门提供一个模块，`timeit`，用来测量程序的速度。我建议你用 `timeit`，而不是用自己创造的一些测量方法，就像我在这里所做的。但是这个测试方法我多年前就在使用了，甚至比 `timeit` 模块出现在 Python 标准库的时间还早，所以，在这个例子中我没有用 `timeit` 也算情有可原吧。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中关于文件对象，内建 `enumerate` 函数，`os.popen` 以及 `time` 和 `timeit` 模块；*Perl Cookbook* 8.2。

## 2.6 处理文件中的每个词

感谢：Luther Blissett

## 任务

你想对一个文件中的每个词做一些处理。

## 解决方案

完成这个任务的最好的办法是使用两重循环，一个用于处理行，另一个则处理每一行中的每个词：

```
for line in open(thefilepath):
    for word in line.split():
        dosomethingwith(word)
```

for语句假定了词是一串非空的字符，并由空白字符隔开（和UNIX程序wc一样）。如果词的定义有变化，还可以使用正则表达式。比如：

```
import re
re_word = re.compile(r"[\w'-]+")
for line in open(thefilepath):
    for word in re_word.finditer(line):
        dosomethingwith(word.group(0))
```

在此例中，词被定义为数字字母，连字符或单引号构成的序列。

## 讨论

如果还需要其他的对词的定义，当然也需要使用不同的正则表达式。外层关于文件行的循环，则不用改变。

通常把迭代封装成迭代器对象是个好主意，这种封装也很常见和易于使用，如下：

```
def words_of_file(thefilepath, line_to_words=str.split):
    the_file = open(thefilepath):
        for line in the_file:
            for word in line_to_words(line):
                yield word
    the_file.close()
for word in words_of_file(thefilepath):
    dosomethingwith(word)
```

这种方式可以清晰有效地将两部分内容分开：一个是怎么迭代所有的元素（本例中，指的是文件中的词），另一个是要对每个元素做什么处理。一旦你将迭代操作的部分封装在一个迭代器对象中（常常表现为一个生成器(generator）），你就可以只使用一个for语句来完成迭代操作了。可以在你的程序各处重复地使用这个迭代器，而且如果需要维护代码的话，你只需要修改一处——即迭代器的定义和实现部分，而不用到处寻找需要修改的部分。这种好处，类似于任何其他语言中正确地定义和使用了函数，就不必到处复制粘贴代码段。通过Python的迭代器，可充分复用循环控制结构。

通过重构将循环放入一个生成器，我们还获得其他两个小小的增强——文件被显式地确保关闭了，行文本被划分成单词的方式也更通用了（默认使用字符串对象的 `split` 方法，但是却给了指定其他方式的可能性）。比如，如果我们需要用正则表达式来取词，可以对 `words_of_file` 再做一层包装：

```
import re
def words_by_re(thefilepath, repattern=r"\w'-'+"):
    wre = re.compile(repatter)
    def line_to_words(line):
        for mo in wre.finditer(line):
            return mo.group(0)
    return words_of_file(thefilepath, line_to_words)
```

这里，我们也给出了一种默认词定义的正则表达式定义，当然如果有必要使用其他不同的词定义，也可以传入不同的表达式。过度追求通用化是一种有害的诱惑，但是基于经验所做的一些适度的通用化总是能够事半功倍。采用一个函数，接受可选的参数，并在参数为默认值时提供一些合适的值，是一种快速而方便的实现通用化的方法。

## 更多资料

第 19 章关于迭代器和生成器的更多内容；*Library Reference* 和 *Python in a Nutshell* 中关于文件对象和 `re` 模块；*Perl Cookbook* 8.3。

## 2.7 随机输入/输出

感谢：Luther Blissett

### 任务

给定一个包含很多固定长度记录的大二进制文件，你想读取其中某一条记录，而且还不需要逐条读取记录。

### 解决方案

一条记录相对于文件头部的偏移字节，就是这条记录的长度再乘以记录的条数（正整数，从 0 开始计数）。因此，可以直接将读取位置设置在正确的点上，然后读取数据。比如，如果每条记录长度是 48 字节长，则从二进制文件中读取第 7 条记录的方法如下：

```
thefile = open('somebinfile', 'rb')
record_size = 48
record_number = 6
thefile.seek(record_size * record_number)
buffer = thefile.read(record_size)
```

注意，第 7 条，也就是 `record_number` 是 6：记录条数从 0 开始统计。

## 讨论

本节的方法适用于包含相同长度的记录的文件（通常是二进制的），而不适用于普通的文本文件。为了表明这一点，本节给出的代码在调用 `seek` 之前，给 `open` 函数传递了一个“rb”参数以指明读取二进制文件。只要被读取的文件是作为二进制文件被打开的，在最终关闭文件之前，就可以根据需要随意地使用 `seek` 和 `read` 方法——不一定要正好在执行 `seek` 之前打开文件。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中关于文件对象的章节；*Perl Cookbook* 8.12。

## 2.8 更新随机存取文件

感谢：Luther Blissett

### 任务

给定一个包含很多固定长度记录的大二进制文件，你想读取其中某一条记录，并且修改该条记录的某些字段的值，然后写回到文件中。

### 解决方案

读取记录，解包，执行任何需要的数据更新，然后将所有字段重新组合成记录，接着找到正确的位置，最后再写入。见如下代码：

```
import struct
format_string = '8l'                                # 或者说，一条记录是 8 个 4 字节整数
thefile = open('somebinfile', 'r+b')
record_size = struct.calcsize(format_string)
record_numberb
thefile.seek(record_size * record_number)
buffer = thefile.read(record_size)
fields = list(struct.unpack(format_string, buffer))
# 进行计算，并修改相关的字段，然后：
buffer = struct.pack(format_string, *fields)
thefile.seek(record_size * record_number)
thefile.write(buffer)
thefile.close()
```

## 讨论

本节的方法适用于包含相同长度的记录的文件（通常是二进制的），而不适用于普通的文本文件。而且，每条记录的长度由一个结构化的格式化字符串来定义，如代码所示。一个典型的格式化字符串，比如，“8l”，指明每条记录是由 8 个 4 字节整数构成，每个

整数都有个被指定的值而且可以被解包到一个 Python int 类型的对象中。在这个例子中，`fields` 变量被绑定到了一个 8 个整数的列表上。注意，`struct.unpack` 返回的是一个元组。由于元组是不可改变的，通过计算完成数据更新之后只能重新绑定 `fields` 变量。而列表则是可改变的，每个字段可以根据需要重新绑定。因此，为了方便，绑定 `fields` 的时候我们显式地要求一个列表，同时确保不改变这个列表的长度。对这个例子而言，列表中需要恰好含有 8 个整数，否则当我们利用值为 “8I” 的 `format_string` 来打包时，`struct.pack` 就会抛出异常。如果记录的长度不一，本节的方法也不适用。

为了迅速定位记录的起始地址，可以选择使用相对位置寻址方式，而不是用计算出的 `record_size*record_number` 偏移字节：

```
thefile.seek(-record_size, 1)
```

传递给 `seek` 方法的第二个参数值（1），告诉文件对象定位到相对当前位置的某处（此处，需要回退一些字节，因为我们给了第一个参数一个负值）。`seek` 的默认寻址方式是在文件中用绝对偏移量来定位（从文件开头开始计算）。也可以给 `seek` 的第二个参数传个 0 值，显式地要求使用默认的行为方式。

不用正好在第一次使用 `seek` 之前打开文件，也不用调用 `write` 之后就马上关闭文件。一旦正确地打开了文件（作为需要更新的二进制文件，不是文本文件），在关闭文件之前，可以根据需要对此文件进行任意次更新。在此展示这些调用是为了强调打开文件做随机存取更新的关键所在，同时也是为了再次提醒读者，完成所有操作之后迅速关闭文件的重要性。

这文件需要更新（同时允许读和写）。这正是传递给 `open` 的 “r+b” 这个参数的意思：打开文件用于读写，但并不隐式地对文件内容做任何转化，因为这是一个二进制文件。（对于 UNIX 和类 UNIX 系统，“b” 部分并不是必需的，但是为了清晰，却是值得推荐的。而且，对某些平台来说，指定这个部分是非常关键的，比如 Windows）如果你准备从头开始创建这个文件，而且还要反复定位，读取并更新一些记录，同时还用关闭此文件并重新打开，可以将 `open` 的第二个参数指定为 “w+b”。然而，这样的奇怪要求我还从来没见过。通常二进制文件是在第一次创建之后（通过 “wb” 打开，写入数据，然后关闭文件），再用 “r+b” 重新打开并进行更新的。

虽然本节方法只适用于记录长度一致的文件，不过，适应更高级需求的可能性也存在：一个单独的“索引文件”，提供了另一个“数据文件”中的每条记录的长度和偏移量的信息。这种连续的不定长记录的方式已经不太流行了，但是在过去却是很重要的方式。如今，我们遇到的也就是文本文件（各种类型，不过 XML 越来越多），数据库，偶尔还会碰到记录长度固定的二进制文件。如果你确实需要处理这种带索引的连续的二进制文件，代码也不会改动太多，只需要从索引文件中读取记录长度 `record_size` 和偏移量，并把两者作为参数传递给 `thefile.seek`，不需要如本节解决方案所做的那样，自己计算偏移。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 关于文件对象和 `struct` 模块; *Perl Cookbook* 8.13。

## 2.9 从 zip 文件中读取数据

感谢: Paul Prescod、Alex Martelli

### 任务

你想直接检查一个 zip 格式的归档文件中部分或者所有的文件，同时还要避免将这些文件展开到磁盘上。

### 解决方案

zip 文件是一种流行的跨平台的归档文件。Python 标准库提供了 `zipfile` 模块来简便地访问这种文件：

```
import zipfile
z = zipfile.ZipFile("zipfile.zip", "r")
for filename in z.namelist():
    print 'File:', filename,
    bytes = z.read(filename)
    print 'has', len(bytes), 'bytes'
```

### 讨论

Python 能直接处理 zip 文件中的数据。可以直接看到归档目录中的所有子项，并且直接处理这些“数据文件”。解决方案中给出的代码片段能够列出归档文件 `Zipfile.zip` 中所有文件的名字和长度。

`zipfile` 模块现在还不能处理分卷 zip 文件和带有注释的 zip 文件。注意，要使用 `r` 作为标志参数，而不是 `rb`，虽然 `rb` 看起来挺自然的（特别是在 Windows 下）。对于 `zipfile`，它的标志与 `open` 所用的打开一个文件的标志不太一样，它根本不认识 `rb`。`r` 标志可以应付所有平台上的各种 zip 文件。如果 zip 文件中包含一些 Python 模块（也即`.py` 或者`.pyc` 文件），也许还有一些其他的（数据）文件，可以把这个文件的路径加入到 Python 的 `sys.path` 中，并用 `import` 语句来导入处于这个 zip 文件中的模块。下面给出一个玩具示例，这只是一个自包含的、纯粹的展示型的例子，它会凭空创建一个 zip 文件，并从中导入一个模块，最后再删除该文件——仅仅是为了向你展示如何做到这一切的：

```
import zipfile, tempfile, os, sys
handle, filename = tempfile.mkstemp('.zip')
os.close(handle)
z = zipfile.ZipFile(filename, 'w')
```

```
z.writestr('hello.py', 'def f(): return "hello world from "+__file__\n')
z.close()
sys.path.insert(0, filename)
import hello
print hello.f()
os.unlink(filename)
```

运行这个脚本会产生一些类似这样的输出：

```
hello world from /tmp/tmpESVzeY.zip/hello.py
```

除了展示 Python 从 zip 文件中导入模块的能力，这段代码还显示了怎样制造出一个临时文件，以及怎样使用 `writestr` 方法来向 zip 文件中添加一个成员，你甚至不用事先在磁盘上创建这个成员。

注意，从 zip 文件导入模块所用的路径会被看做是个目录。（在这个特定的例子里，路径是`/tmp/tmpESVzeY.zip`，不过，由于我们处理的是一个临时文件，因此每次运行脚本这个值都会不同，具体取决于你用的系统）具体地说，在模块 `hello` 这个被导入的模块中，全局变量 `__file__` 的值是`/tmp/tmpESVzeY.zip/hello.py` 这样一种伪路径，它由被看做“目录”的 zip 文件路径加上位于 zip 文件中的 `hello.py` 的相对路径组合而成。如果从 zip 文件中导入模块，处于 zip 文件中的模块的数据文件将通过相对路径获取，需要适应从 zip 文件导入模块的这种特性，因为你无法用 `open` 函数来打开一个“伪路径”以获取文件对象：要想读取或者写入 zip 文件中的文件，必须使用 Python 标准库的 `zipfile` 模块，如同解决方案给出的代码那样。

更多的关于从 zip 文件中导入模块的资料，请参看第 16.12 节。虽然本节示例主要针对 UNIX，但本节讨论中关于从 zip 文件导入模块的信息和内容在 Windows 下同样有效。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中 `zipfile`, `tempfile`, `os`, `sys` 模块的资料；关于归档文件树的内容，参见第 2.11 节；更多有关从 zip 文件导入模块的信息，参见第 16.12 节。

## 2.10 处理字符串中的 zip 文件

感谢：Indyana Jones

### 任务

你的程序接收到了一个字符串，其内容是一个 zip 文件，需要读取这个 zip 文件中的信息。

### 解决方案

应对这种问题，正是 Python 标准库的 `cStringIO` 模块的拿手好戏：

```
import cStringIO, zipfile
class ZipString(ZipFile):
    def __init__(self, datastring):
        ZipFile.__init__(self, cStringIO.StringIO(datastring))
```

## 讨论

我总是遇到这类任务——比如 zip 文件可能来自于数据库的 BLOB 字段，或者来自于网络连接。我过去把这些二进制数据先存成临时文件，然后用标准库模块 `zipfile` 打开此文件。当然，我会确保完成任务之后删除临时文件。一天，我想到了使用标准库模块 `cStringI`，之后我就再也没用过老办法了。

`cStringIO` 模块可以将一串字节封装起来，让你像访问文件对象一样访问其中的数据。另一方面，还可以用把 `cStringIO.StringIO` 的实例当做一个文件对象，向其中写入数据，最后得到的是一串内存中的字节。很多处理文件的 Python 模块其实根本不检查你传递给它们的是不是一个真正的文件——任何像文件一样的对象它们都接受。它们只是在需要的时候调用文件的方法，只要给这些对象提供了相关的方法，并且做出了正确的反应，一切都会正常工作。这展示了基于签名的多态机制的强大力量，同时也解释了为什么最好不要在你的代码中进行类型检查（比如可怕的 `if type(x) is y`，以及稍微好点的 `if isinstance(x, y)`）。一些低级的模块，比如 `marshal`，很不幸地会执着要求“真实”的文件，但 `zipfile` 很通融，本节的例子也说明了，有了它生活多么美好。

如果用的 Python 版本和主流的基于 C 的 Python 版本（也称为 CPython）不同，可能在标准库中找不到 `cStringIO` 模块。模块名最前面的 `c`，表示它是个基于 C 的模块，为速度做过优化，不保证能够在其他兼容的 Python 实现体的标准库中找到对应的版本。这些兼容的 Python 实现体包括了产品级品质的（比如 Jython，用 Java 实现并运行在 JVM 上）以及实验性的（比如 pypy，用 Python 代码产生机器码结果，以及 IronPython，基于 C# 并运行在微软的.NET CLR）。别担心，Python 标准库总会包含 `StringIO` 模块，它由纯 Python 代码实现（因此也能适用于任何兼容的 Python 实现体），并实现了和 `cStringIO` 一样的功能（不过没有那么快，至少在主流的 CPython 上是这样）。只需略微修改一下 `import` 语句，以确保如果有 `cStringIO`，就导入 `cStringIO`，如果没有，则导入 `StringIO` 作为代替。比如，代码可能会变成这样：

```
import zipfile
try:
    from cStringIO import StringIO
except ImportError:
    from StringIO import StringIO
class ZipString(ZipFile):
    def __init__(self, datastring):
        ZipFile.__init__(self, StringIO(datastring))
```

经过这次修改，这段代码就可以在 Jython 或其他的实现体中工作了。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中的 zipfile 和 cStringIO 模块；Jython 的信息参见 <http://www.jython.org/>；pypy 的信息参见 <http://codespeak.net/pypy/>；IronPython 的信息参见 <http://ironpython.com/>。

## 2.11 将文件树归档到一个压缩的 tar 文件

感谢：Ed Gordon、Ravi Teja Bhupatiraju

### 任务

需要将一个文件树中的所有文件和子目录归档到一个 tar 归档文件，然后用流行的 gzip 方式或者更高压缩率的 bzip2 方式来压缩。

### 解决方案

Python 标准库的 tarfile 模块直接提供了这两种压缩方式，你只需在调用 tarfile.TarFile.open 创建归档文件时，传入一个选项字符串以指定需要的压缩方式即可。比如：

```
import tarfile, os
def make_tar(folder_to_backup, dest_folder, compression='bz2'):
    if compression:
        dest_ext = '.' + compression
    else:
        dest_ext = ''
    arcname = os.path.basename(folder_to_backup)
    dest_name = '%s.tar%s' % (arcname, dest_ext)
    dest_path = os.path.join(dest_folder, dest_name)
    if compression:
        dest_cmp = ':' + compression
    else:
        dest_cmp = ''
    out = tarfile.TarFile.open(dest_path, 'w'+dest_cmp)
    out.add(folder_to_backup, arcname)
    out.close()
    return dest_path
```

### 讨论

可以给函数 make\_tar 传递一个指定压缩方式的参数，字符串 “gz” 则表明使用 gzip 压缩，默认的则是 “bz2”，指定 bzip2 压缩。也可以传递一个空字符串”，表明你压根就不需要压缩。当你选择不压缩、用 gzip 压缩或者 bzip2 压缩时，除了会影响到文件扩展名成为.tar、.tar.gz 或.tar.bz2 之外，你的选择还决定了 “w”，“w:gz” 和 “w:bz2” 中

哪个字符串会被作为第二个参数传递给 `tarfile.TarFile.open`。

除了 `open` 之外，类 `tarfile.TarFile` 提供了几种其他的类方法（`classmethods`），可以用这些方法生成一个合适的实例。我发现 `open` 最方便灵活，因为它可以从模式字符串参数中获取指定压缩方式的信息。当然，如果确定无条件地使用 `bzip2` 压缩方式，也可以用类方法 `bz2open` 来替代 `open`。

一旦我们拥有了 `tarfile.TarFile` 的一个实例，并且也设置好了我们需要的压缩方式，这个实例的 `add` 方法将完成所有剩下的工作。特别是，当字符串 `folder_to_backup` 是“目录”而不是普通文件的名字时，`add` 会递归地把该目录中所有的子树添加进来。在另一些场合中，我们可能会希望改变默认的行为，并精确地控制需要被归档的文件和目录，我们可以给 `add` 传递一个额外的参数 `recursive=False` 来关闭默认的递归添加功能。在调用 `add` 之后，留给 `make_tar` 函数做的事情就是关闭 `TarFile` 实例并返回所写入的 tar 文件路径，这是因为有时调用者会需要使用这个信息。

## 更多资料

*Library Reference* 中关于 `tarfile` 模块的文档。

## 2.12 将二进制数据发送到 Windows 的标准输出

感谢：Hamish Lawson

### 任务

在 Windows 平台上，你想把二进制数据（比如一张图片）发送到 `stdout` 中。

### 解决方案

Python 标准库中，依赖特定平台（Windows）的模块 `msvcrt` 提供了 `setmode` 函数，可用来完成这个任务：

```
import sys
if sys.platform == "win32":
    import os, msvcrt
    msvcrt.setmode(sys.stdout.fileno(), os.O_BINARY)
```

现在可以给 `sys.stdout.write` 任何字节或者字符串参数，这些字节和字符串会被不加修改地传递到标准输出中。

### 讨论

由于 UNIX 并不（或不需要）区分文本和二进制模式，如果打算在 Windows 中读取或者写入二进制数据，比如图片，则必须以二进制模式打开文件。这对于向标准输出（比

如，CGI 脚本就可能会这么干）写入二进制数据的程序而言是个问题，因为 Python 通常以文本模式打开 `sys.stdout` 文件对象。

可以指定命令行选项`-u` 以二进制模式打开 `stdout`。比如，如果你知道你的 CGI 脚本运行在 Apache web 服务器中，可以这样写你的脚本的第一行：

```
#! c:/python23/python.exe -u
```

这假设了用的是 Python 2.3 的标准安装。不过，并不能控制你的脚本在什么样的命令行中运行。解决方案中给出的是一个可行的选择。`setmode` 函数提供了 Windows 专用的 `msvcrt` 模块，以方便用户修改 `stdout` 固有的文件描述符。通过这个函数，可以在程序内部确认 `sys.stdout` 被设置为二进制模式。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中 `msvcrt` 模块的文档。

## 2.13 使用 C++的类 `iostream` 语法

感谢：Erik Max Francis

### 任务

你喜爱 C++ 的基于 `ostream` 和操纵符（插入了这种特定的对象后，它会在 `stream` 中产生特定的效果）的 I/O 方式，并想将此形式用在自己的 Python 程序中。

### 解决方案

Python 允许使用对特殊方法（即名字前后带有连续两个下划线的方法）进行了重定义的类来重载原有的操作符。为了将`<<`用于输出，如同在 C++ 中所做的一样，需要编写一个输出流类，并定义特殊方法`__lshift__`：

```
class IOManipulator(object):
    def __init__(self, function=None):
        self.function = function
    def do(self, output):
        self.function(output)
def do_endl(stream):
    stream.output.write('\n')
    stream.output.flush()
endl = IOManipulator(do_endl)
class OStream(object):
    def __init__(self, output=None):
        if output is None:
            import sys
            output = sys.stdout
```

```

        self.output = output
        self.format = '%s'
    def __lshift__(self, thing):
        ''' 当你使用<<操纵符并且左边操作对象是OStream时,
            Python会调用这个特殊方法 '''
        if isinstance(thing, Iomanipulator):
            thing.do(self)
        else:
            self.output.write(self.format % thing)
            self.format = '%s'
        return self
    def example_main( ):
        cout = OStream( )
        cout<< "The average of " << 1 << " and " << 3 << " is " << (1+3)/2 << endl
    # 输出: The average of 1 and 3 is 2
    if __name__ == '__main__':
        example_main( )

```

## 讨论

在 Python 中包装一个像文件一样的对象，模拟 C++ 的 `ostream` 的语法，还算比较容易。本节展示了怎样编写代码实现插入操作符`<<`的效果。解决方案中的代码实现了一个 `Iomanipulator` 类（像 C++ 中一样）来调用插入到流中的任意函数，还实现了预定义的操纵符 `endl`（猜猜它得名何处）来写入新行和刷新流。

`Ostream` 类的实例有一个叫做 `format` 的属性，在每次调用 `self.output.write` 之后，这个属性都会被设置为默认值“`%s`”，这样的好处是，每次创建一个操纵符之后我们可在其中临时保存流对象的格式化状态，比如：

```

def do_hex(stream):
    stream.format = '%x'
hex = IManipulator(do_hex)
cout << 23 << ' in hex is ' << hex << 23 << ', and in decimal ' << 23 << endl
# 输出: 23 in hex is 17, and in decimal 23

```

一些人很讨厌 C++ 的 `cout<<something` 的语法，另一些人却很喜欢。在本节的例子中，所用的语法至少在可读性和简洁方面都胜过下面的语法：

```
print>>somewhere, "The average of %d and %d is %f\n" % (1, 3, (1+3)/2)
```

这种方式是 Python “原生”的方式（看上去很像 C 的风格）。这要看你更习惯 C++ 还是 C，至少本节给了你另一个选择。即使最终没有使用本节提供的方法，了解 Python 中简单的操作符重载还是蛮有趣的。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中关于文件对象以及特殊方法 `__lshift__` 的文档，第 4.20 节中关于 C 函数 `printf` 的 Python 实现的信息。

## 2.14 回退输入文件到起点

感谢: Andrew Dalke

### 任务

需要创建一个输入文件对象 (数据可能来自于网络 socket 或者其他输入文件句柄), 此文件对象允许回退到起点, 这样就可以完全读取其中所有数据。

### 解决方案

将文件对象封装到一个合适的类中:

```
from cStringIO import StringIO
class RewindableFile(object):
    """ 封装一个文件句柄以便重定位到开始位置 """
    def __init__(self, input_file):
        """ 将 input_file 封装到一个支持回退的类文件对象中 """
        self.file = input_file
        self.buffer_file = StringIO( )
        self.at_start = True
        try:
            self.start = input_file.tell( )
        except (IOError, AttributeError):
            self.start = 0
        self._use_buffer = True
    def seek(self, offset, whence=0):
        """ 根据给定的字节定位.
        必须: whence == 0 and offset == self.start
        """
        if whence != 0:
            raise ValueError("whence=%r; expecting 0" % (whence,))
        if offset != self.start:
            raise ValueError("offset=%r; expecting %s" % (offset,
                self.start))
        self.rewind( )
    def rewind(self):
        """ 回到起始位置 """
        self.buffer_file.seek(0)
        self.at_start = True
    def tell(self):
        """ 返回文件的当前位置 (必须在开始处) """
        if not self.at_start:
            raise TypeError("RewindableFile can't tell except at start
                of file")
        return self.start
    def _read(self, size):
```

```

if size < 0:          # 一直读到文件末尾
    y = self.file.read( )
    if self._use_buffer:
        self.buffer_file.write(y)
    return self.buffer_file.read( ) + y
elif size == 0:        # 不必读空字符串
    return ""
x = self.buffer_file.read(size)
if len(x) < size:
    y = self.file.read(size - len(x))
    if self._use_buffer:
        self.buffer_file.write(y)
    return x + y
return x
def read(self, size=-1):
    """ 根据 size 指定的大小读取数据
    默认为-1，意味着一直读到文件结束
    """
    x = self._read(size)
    if self.at_start and x:
        self.at_start = False
    self._check_no_buffer( )
    return x
def readline(self):
    """ 从文件中读取一行"""
    # buffer_file 中有吗？
    s = self.buffer_file.readline( )
    if s[-1:] == "\n":
        return s
    # 没有，从输入文件中读取一行
    t = self.file.readline( )
    if self._use_buffer:
        self.buffer_file.write(t)
    self._check_no_buffer( )
    return s + t
def readlines(self):
    """读取文件中所有剩余的行"""
    return self.read( ).splitlines(True)
def _check_no_buffer(self):
    # 如果'nobuffer'被调用，而且我们也完成了对缓存文件的处理
    # 那就删掉缓存，把所有的東西都重定向到原来的输入文件
    if not self._use_buffer and \
        self.buffer_file.tell() == len(self.buffer_file.
            getvalue()):
        # 为了获得尽可能高的性能，我们重新绑定了 self 中的所有相关方法
        for n in 'seek tell read readline readlines'.split( ):
            setattr(self, n, getattr(self.file, n, None))
    del self.buffer_file
def nobuffer(self):
    """通知 RewindableFile，一旦缓存耗尽就停止继续使用缓存"""
    self._use_buffer = False

```

## 讨论

有时，从 `socket` 或其他输入文件句柄中得来的数据并不是我们想要的。比如，假设从一个有问题的服务器读取数据，此服务器应该返回 XML 流，但它有时却给你未格式化的错误信息。（这种情况时常发生，因为很多服务器并不能正确处理错误输入。）

本节的 `RewindableFile` 类能够帮助你解决此类问题。`r = RewindableFile(f)` 将原来的输入流 `f` 封装进了一个“可回退的文件”的实例 `r`，`r` 模仿 `f` 的行为，但同时提供了缓存。对 `r` 的读取请求被转移到了 `f`，读取的数据则添加进了缓存，然后返回给调用者。缓存中保存着到目前为止读取的所有数据。

`r` 可以回退，也即定位到开始位置。下一个请求读取的内容可能会来自于缓存，直到缓存被全部读取，此时它会从输入流中获取数据。新读取的数据也被添加到缓存中。

如果不再需要缓存了，可直接调用 `r` 的 `nobuffer` 方法。这相当于告诉 `r`，一旦它读完了缓存中的当前内容，它就可以丢弃缓存。调用 `nobuffer` 之后，`seek` 的行为就处于未定义状态了。

举个例子，假设有個服务器，它可能会给你错误信息，其形式为“ERROR: cannot do that”，或者给你一个 XML 数据流，其内容以“<?xml...”开始：

```
import RewindableFile
infile = urllib2.urlopen("http://somewhere/")
infile = RewindableFile.RewindableFile(infile)
s = infile.readline()
if s.startswith("ERROR:"):
    raise Exception(s[:-1])
infile.seek(0)
infile.nobuffer()      # 不再缓存数据
...process the XML from infile...
```

一些在某些场合下很有用的 Python 惯用方式在此类中不被支持：无法可靠地将 `RewindableFile` 实例的被绑定方法（bound method）隐藏起来（如果不知道什么叫做被绑定方法，没关系，反正在这里肯定无法把它们藏在任何地方）。出现这种限制的原因是，当缓存空了，`RewindableFile` 代码会给输入文件的 `read`、`readlines` 等方法重新赋值，作为 `self` 实例的变量。这样会有略好的性能，代价是不再支持常用的那种保存被绑定方法的惯用方式。6.11 节中会给出另一个相似的技术，不过其中的实例不可逆地改变了它自己的方法。

获取当前位置的 `tell` 方法，只能正好在完成了封装之后，在读取任何数据之前，由 `RewindableFile` 实例调用，以获取起始字节位置。`RewindableFile` 实现了 `tell` 方法，是为了得到被封装的文件对象的真实位置，并以此为起始位置。如果被封装的文件不支持 `tell`，那么 `RewindableFile` 实现的 `tell` 会返回 0。

## 更多资料

在 <http://www.dalkescientific.com/Python/> 有本节代码的最新版本；*Library Reference* 和 *Python in a Nutshell* 中关于文件对象和 cStringIO 模块的内容；参考第 6.11 节中另一个通过重绑定不可逆地改变自身行为的例子。

## 2.15 用类文件对象适配真实文件对象

感谢：Michael Kent

### 任务

需要传递一个类似文件的对象（比如，调用 `urllib.urlopen` 返回的结果）给一个函数或者方法，但这个函数或方法要求只接受真实的文件对象（比如，像 `marshal.load` 这样的函数）。

### 解决方案

为了过类型检查这一关，我们需要将类文件对象中的所有数据写入到磁盘中的一个临时文件。然后使用临时文件的（真实）文件对象。下面给出一个实现这个想法的函数：

```
import types, tempfile
CHUNK_SIZE = 16 * 1024
def adapt_file(fileObj):
    if isinstance(fileObj, file): return fileObj
    tmpFileObj = tempfile.TemporaryFile
    while True:
        data = fileObj.read(CHUNK_SIZE)
        if not data: break
        tmpFileObj.write(data)
    fileObj.close( )
    tmpFileObj.seek(0)
    return tmpFileObj
```

### 讨论

本节展示的其实是设计模式中的适配器（即 Adapter，比如你想要 X，我却给你 Y 以替换 X）的 Python 风格的实现。虽然设计模式通常被认为是一种面向对象的设计方式，因此一般需要用类来实现，但具体实现并没有什么限制。比如此例中，我们根本不需要引入任何新类，因为 `adapt_file` 函数已经足够了。这里我们遵守奥卡姆剃刀原理（译者注：Occam's Razor，奥卡姆是 14 世纪的一个逻辑学家和天主教修道士，奥卡姆剃刀原理即“如无必要，勿增实体”），不在没有必要的情况下引入任何实体。

当需要依赖一些底层的、要求精确类型的工具时，应该首先考虑适配，而不是类型检查。当获得一个适合的可以绕过类型检查的对象时，应该考虑将其配接成需要的对象。用这种方式，你的代码会更加灵活，也更具复用性。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中内建文件对象、`tempfile` 和 `marshal` 模块的内容。

# 2.16 遍历目录树

感谢：Robin Parmar、Alex Martelli

## 任务

需要检查一个“目录”，或者某个包含子目录的目录树，并根据某种模式迭代所有的文件（也可能包含子目录）。

## 解决方案

Python 标准库模块 `os` 中的生成器（generator）`os.walk` 对于这个任务来说完全够用了，不过我们可以给它打扮打扮，将其封装为一个我们自己的函数：

```
import os, fnmatch
def all_files(root, patterns='*', single_level=False, yield_folders=False):
    # 将模式从字符串中取出放入列表中
    patterns = patterns.split(';')
    for path, subdirs, files in os.walk(root):
        if yield_folders:
            files.extend(subdirs)
        files.sort()
        for name in files:
            for pattern in patterns:
                if fnmatch.fnmatch(name, pattern):
                    yield os.path.join(path, name)
                    break
        if single_level:
            break
```

## 讨论

标准文件树遍历生成器 `os.walk` 既强大又简单灵活。不过，`os.walk` 还缺乏应用程序需要的一些细节上的处理能力，比如根据某种模式选择文件，扁平（线性）地以排序后的顺序循环所有文件（也可能包括子目录），检查一个单一目录（不进入其子目录）。本节代码则展示了，通过将 `os.walk` 封装到另一个简单的生成器中，并使用标准库模块

`fnmatch` 来检查文件名匹配模式，是多么的简单方便。

文件名匹配模式可能是大小写无关的（这依赖于平台），也可能是相关的，比如 UNIX 风格，不过这些能力都是标准 `fnmatch` 模块能够提供的。为了指定多个模式，可用分号将它们连接起来。注意，这意味着那些分号本身不是模式的一部分。

举个例子，可以很容易地从/tmp 目录及其子目录中获得一个包括所有 Python 和 HTML 文件的列表：

```
thefiles = list(all_files('/tmp', '*.py;*.htm;*.html'))
```

如果想一次处理一个文件的路径（比如，逐行打印它们），不需要先建立一个列表：可以直接对 `all_files` 调用的结果进行循环操作：

```
for path in all_files('/tmp', '*.py;*.htm;*.html'):
    print path
```

如果你的平台是大小写敏感的，而且你也希望严格匹配大小写，那么需要在指定模式的时候稍微辛苦点，比如，用 “`*.[Hh][Tt][Mm][Ll]`” 来替代原来的 “`*.html`”。

## 更多资料

参看 *Library Reference* 和 *Python in a Nutshell* 中的 `os.path`, `fnmatch` 模块，以及 `os.walk` 生成器。

## 2.17 在目录树中改变文件扩展名

感谢：Julius Welby

### 任务

需要在一个目录的子树中重命名一系列文件，具体地说，你想将某一指定类型的文件的扩展名改成另一种扩展名。

### 解决方案

用 Python 标准库提供的 `os.walk` 函数来处理子目录中的所有文件，任务变得非常容易：

```
import os
def swapextensions(dir, before, after):
    if before[:1] != '.':
        before = '.'+before
    thelen = -len(before)
    if after[:1] != '.':
        after = '.'+after
    for path, subdirs, files in os.walk(dir):
        for oldfile in files:
```

```
if oldfile[thelen:] == before:  
    oldfile = os.path.join(path, oldfile)  
    newfile = oldfile[:thelen] + after  
    os.rename(oldfile, newfile)  
  
if __name__=='__main__':  
    import sys  
    if len(sys.argv) != 4:  
        print "Usage: swapext rootdir before after"  
        sys.exit(100)  
    swapextensions(sys.argv[1], sys.argv[2], sys.argv[3])
```

## 讨论

本节展示了怎样改变一个指定目录中所有文件的扩展名，涉及范围包括了所有的子目录，以及更下级子目录，以此类推。这种技术很适合在一个文件夹结构中批量修改文件的扩展名，比如针对一个 web 站点的目录树进行修改。可以用这个脚本纠正用程序批量生成文件时所犯的错误。

本节给的代码既可以被用作一个可以随时导入的模块，也可以作为一个脚本并运行在命令行中，而且代码设计得很谨慎，完全是平台无关的。可以传入带点(.) 的扩展名，也可以传入不带点的，程序在必要时会自行插入点。(作为这种方便性的一个直接后果是，此程序不能处理没有扩展名的文件，也不能直接处理点，在 UNIX 系统中这种限制有时很让人恼火。)

实现本节解决方案所用的技术，一些完美主义者会认为过于底层——直接用操作字符串的方式来修改处理文件名和扩展名，而不是用 os.path 提供的函数。不过这没什么大不了的：用 os.path 很好，但是用 Python 的强大的字符串工具也很好。

## 更多资料

参看作者的主页 <http://www.outwardlynormal.com/python/swapextensions.htm>。

## 2.18 从指定的搜索路径寻找文件

感谢：Chui Tey

### 任务

给定一个搜索路径（一个描述目录信息的字符串），需要根据这个路径和请求的文件名找到第一个符合要求的文件。

### 解决方案

基本上，需要循环指定的搜索路径中的目录：

```
import os
def search_file(filename, search_path, pathsep=os.pathsep):
    """ 给定一个搜索路径，根据请求的名字找到文件 """
    for path in search_path.split(pathsep):
        candidate = os.path.join(path, filename)
        if os.path.isfile(candidate):
            return os.path.abspath(candidate)
    return None
if __name__ == '__main__':
    search_path = '/bin' + os.pathsep + '/usr/bin' # ; on Windows, : on UNIX
    find_file = search_file('ls', search_path)
    if find_file:
        print "File 'ls' found at %s" % find_file
    else:
        print "File 'ls' not found"
```

## 讨论

本节的任务是个很常见的需求，Python 对这个需求的解决办法也极其简单。本章其他一些节也会处理相似或相关的一些任务：见第 2.20 节，在 Python 自身的搜索路径中找文件，以及第 2.19 节，在指定的搜索路径中根据匹配模式寻找文件。

进行搜索的循环可以被写成很多形式，但一旦找到就立刻返回路径（这里用绝对路径，主要基于统一性和方便性的考虑）是最简单的，而且速度很快。在循环完成之后显式地 `return None` 并不是必须的，因为在 Python 中一个函数执行完毕后会自行返回 `None`。在这里画蛇添足的加一句 `return` 语句，仅仅是为了让人能够一目了然地看清 `search_file` 的所做的事情。

## 更多资料

2.20, 2.10; *Library Reference* 和 *Python in a Nutshell* 中 os 模块的内容。

# 2.19 根据指定的搜索路径和模式寻找文件

感谢： Bill McNeill、 Andrew Kirkpatrick

## 任务

给定一个搜索路径（一个描述目录信息的字符串），需要在此目录中找出所有符合匹配模式的文件。

## 解决方案

基本上，需要循环路径中的所有目录。这个循环最好被封装成一个生成器：

```
import glob, os
def all_files(pattern, search_path, pathsep=os.pathsep):
    """ 给定搜索路径，找出所有满足匹配条件的文件 """
    for path in search_path.split(pathsep):
        for match in glob.glob(os.path.join(path, pattern)):
            yield match
```

## 讨论

生成器的好处是，可以很容易地获取第一个子项，或者所有子项，再或者其中任意一个子项。比如，打印出你的环境变量 PATH 中第一个符合 “\*.pye” 模式的文件：

```
print all_files('*.pye', os.environ['PATH']).next()
```

打印所有这种文件，一行一个：

```
for match in all_files('*.pye', os.environ['PATH']):
    print match
```

以列表形式一次全部打印出来：

```
print list(all_files('*.pye', os.environ['PATH']))
```

我也给 all\_files 函数提供了一个主脚本，以方便打印出我的 PATH 中所有符合匹配模式的文件。因此，不仅能够看到根据指定名字将被执行的那个文件（第一个），还能看到被第一个文件“屏蔽”掉的其他同名文件：

```
if __name__ == '__main__':
    import sys
    if len(sys.argv) != 2 or sys.argv[1].startswith('-'):
        print 'Use: %s <pattern>' % sys.argv[0]
        sys.exit(1)
    matches = list(all_files(sys.argv[1], os.environ['PATH']))
    print '%d match:' % len(matches)
    for match in matches:
        print match
```

## 更多资料

见第 2.18 节中的那个更简单的根据搜索路径寻找文件的例子； *Library Reference* 和 *Python in a Nutshell* 中的 os 和 glob 的相关文档。

## 2.20 在 Python 的搜索路径中寻找文件

感谢： Mitch Chapman

### 任务

一个大的 Python 应用程序包括了资源文件（比如 Glade 项目文件、SQL 模板和图片）

以及 Python 包（Python package）。你想把所有这些相关文件和用到它们的 Python 包储存起来。

## 解决方案

可以在 Python 的 `sys.path` 中寻找文件或目录：

```
import sys, os
class Error(Exception): pass
def _find(pathname, matchFunc=os.path.isfile):
    for dirname in sys.path:
        candidate = os.path.join(dirname, pathname)
        if matchFunc(candidate):
            return candidate
    raise Error("Can't find file %s" % pathname)
def findFile(pathname):
    return _find(pathname)
def findDir(path):
    return _find(path, matchFunc=os.path.isdir)
```

## 讨论

比较大的 Python 应用程序由一系列 Python 包和相关的资源文件组成。将这些相关文件和用到它们的 Python 包一起储存起来是很方便的，可以很容易地对 2.18 提供的代码略加修改，使之能根据 Python 搜索路径的相对路径来寻找文件和目录。

## 更多资料

2.18 节；*Library Reference* 和 *Python in a Nutshell* 中的 `os` 模块相关内容。

## 2.21 动态地改变 Python 搜索路径

感谢：Robin Parmar

### 任务

模块必须处于 Python 搜索路径中才能被导入，但你不想设置个永久性的大路径，因为那样可能会影响性能，所以，你希望能够动态地改变这个路径。

## 解决方案

只需简单地在 Python 的 `sys.path` 中加入一个“目录”，不过要小心重复的情况：

```
def AddSysPath(new_path):
    """ AddSysPath(new_path): 给 Python 的 sys.path 增加一个“目录”
        如果此目录不存在或者已经在 sys.path 中了，则不操作
```

```
返回 1 表示成功, -1 表示 new_path 不存在, 0 表示已经在 sys.path 中了
already on sys.path.

"""

import sys, os
# 避免加入一个不存在的目录
if not os.path.exists(new_path): return -1
# 将路径标准化。Windows 是大小写不敏感的, 所以若确定在
# Windows 下, 将其转成小写
new_path = os.path.abspath(new_path)
if sys.platform == 'win32':
    new_path = new_path.lower( )
# 检查当前所有的路径
for x in sys.path:
    x = os.path.abspath(x)
    if sys.platform == 'win32':
        x = x.lower( )
    if new_path in (x, x + os.sep):
        return 0
sys.path.append(new_path)
# 如果想让 new_path 在 sys.path 处于最前
# 使用: sys.path.insert(0, new_path)
return 1

if __name__ == '__main__':
    # 测试, 显示用法
    import sys
    print 'Before:'
    for x in sys.path: print x
    if sys.platform == 'win32':
        print AddSysPath('c:\\Temp')
        print AddSysPath('c:\\temp')
    else:
        print AddSysPath('/usr/lib/my_modules')
    print 'After:'
    for x in sys.path: print x
```

## 讨论

模块要处于 Python 搜索路径中的目录里才能被导入, 但我们不喜欢维护一个永久性的大目录, 因为其他所有的 Python 脚本和应用程序导入模块的时候性能都会被拖累。本节代码动态地在该路径中添加了一个“目录”, 当然前提是此目录存在而且此前不在 sys.path 中。

sys.path 是个列表, 所以在末尾添加目录是很容易的, 用 sys.path.append 就行了。当这个 append 执行完之后, 新目录即时起效, 以后的每次 import 操作都可能会检查这个目录。如同解决方案所示, 可以选择用 sys.path.insert(0,...), 这样新添加的目录会优先于其他目录被 import 检查。

即使 `sys.path` 中存在重复，或者一个不存在的目录被不小心添加进来，也没什么大不了，Python 的 `import` 语句非常聪明，它会自己应付这类问题。但是，如果每次 `import` 时都发生这种错误（比如，重复的不成功搜索，操作系统提示的需要进一步处理的错误），我们会被迫付出一点小小的性能代价。为了避免这种无谓的开销，本节代码在向 `sys.path` 添加内容时非常谨慎，绝不加入不存在的目录或者重复的目录。程序向 `sys.path` 添加的目录只会在此程序的生命周期之内有效，其他所有的对 `sys.path` 的动态操作也是如此。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中 `sys` 和 `os.path` 模块的内容。

## 2.22 计算目录间的相对路径

感谢：Cimarron Taylor、Alan Ezust

### 任务

需要知道一个目录对另一个目录的相对路径是什么——比如，有时需要创建一个符号链接或者一个相对的 URL 引用。

### 解决方案

最简单的方法是把目录拆分到一个目录的列表中，然后对列表进行处理。我们需要用到一些辅助函数和助手函数，代码如下：

```
import os, itertools
def all_equal(elements):
    ''' 若所有元素都相等，则返回 True，否则返回 False'''
    first_element = elements[0]
    for other_element in elements[1:]:
        if other_element != first_element: return False
    return True
def common_prefix(*sequences):
    ''' 返回所有序列开头部分共同元素的列表
        紧接一个各序列的不同尾部的列表'''
    # 如果没有 sequence，完成
    if not sequences: return [ ], [ ]
    # 并行地循环序列
    common = [ ]
    for elements in itertools.izip(*sequences):
        # 若所有元素相等，跳出循环
        if not all_equal(elements): break
        # 得到一个共同的元素，添加到末尾并继续
        common.append(elements[0])
```

```

# 返回相同的头部和各自不同的尾部
    return common, [sequence[len(common):] for sequence in sequences]
def relpath(p1, p2, sep=os.path.sep, pardir=os.path.pardir):
    ''' 返回 p1 对 p2 的相对路径
        特殊情况: 空串, if p1 == p2;
                    p2, 如果 p2 和 p1 完全没有相同的元素
    '''
    common, (u1, u2) = common_prefix(p1.split(sep), p2.split(sep))
    if not common:
        return p2      # 如果完全没有共同元素, 则路径是绝对路径
    return sep.join([pardir]*len(u1) + u2)
def test(p1, p2, sep=os.path.sep):
    ''' 调用 relpath 函数, 打印调用参数和结果 '''
    print "from", p1, "to", p2, " -> ", relpath(p1, p2, sep)
if __name__ == '__main__':
    test('/a/b/c/d', '/a/b/c1/d1', '/')
    test('/a/b/c/d', '/a/b/c/d', '/')
    test('c:/x/y/z', 'd:/x/y/z', '/')

```

## 讨论

本节解决方案给出的代码中, 简单而通用的 `common_prefix` 是关键部分, 给它任意 N 个序列, 它能返回 N 个序列共同的头部, 和一个各不相同的尾部列表。为了计算两个目录之间的相对路径, 可以忽略掉它们的共同头部。我们只需要一定数目的“向上一级”标记 (通常用 `os.path.pardir`, 比如类 UNIX 系统中的`..`; 需要和尾部长度相同数目的这种符号), 然后再添上目标目录的尾部。`relpath` 函数将一个完整路径拆成一个目录的列表, 然后调用 `common_prefix`, 接着执行我们刚才描述过的操作。

`common_prefix` 的核心部分在那个循环, `for elements in itertools.izip(*sequences)`, 它依赖这个事实: 当最短的序列循环到头时, `izip` 也结束了。循环的主体部分必须在遇到一个不全相等的元组 (根据 `izip` 的说明, 每个元素都来自各序列) 时立刻结束, 同时还要在这个过程中把全等的元素放进 `common` 列表中保存起来。一旦循环结束, 剩下要做的事情就是根据 `common` 列表, 把各个序列的相同头部全部切掉。

`all_equal` 函数还能用另一种方式实现, 没那么简洁明快, 但是也很有趣:

```
def all_equal(elements):
    return len(dict.fromkeys(elements)) == 1
```

或者, 等价但更简洁一点, 适用于 Python 2.4 以上:

```
def all_equal(elements):
    return len(set(elements)) == 1
```

所有元素相等, 等价于包含且只包含这些元素的集合的势 (`cardinality`) 为 1。在使用 `dict.fromkeys` 的变体中, 用 `dict` 来代替 `set`, 所以那个例子可同时适用于 Python 2.3 和 2.4。用 `set` 的那个例子更清晰, 但只能在 Python 2.4 以上的版本中使用 (可以用 Python

标准库中的 `sets` 模块来改写，让它也同时适用于 Python 2.3)。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中的 `os` 和 `itertools` 模块。

## 2.23 跨平台地读取无缓存的字符

感谢： Danny Yoo

### 任务

你的程序需要从标准输出中，读取无缓存的单个的字符，而且它还必须能够同时在 Windows 和类 UNIX 系统中工作。

### 解决方案

当我们手上的工具是平台依赖性的，而需求却是平台无关的时候，可以试试将这些差异封装起来：

```
try:
    from msvcrt import getch
except ImportError:
    ''' 我们不在 windows 中，所以可以尝试类 UNIX 的方式 '''
    def getch( ):
        import sys, tty, termios
        fd = sys.stdin.fileno( )
        old_settings = termios.tcgetattr(fd)
        try:
            tty.setraw(fd)
            ch = sys.stdin.read(1)
        finally:
            termios.tcsetattr(fd, termios.TCSADRAIN, old_settings)
        return ch
```

### 讨论

在 Windows 中，Python 标准库模块 `msvcrt` 提供了方便的 `getch` 函数来读取无缓存的单个字符，直接从键盘读取，且不会在屏幕上回显。但是这个模块并不是 UNIX 和类 UNIX 平台中 Python 标准库的一部分，比如 Linux 和 Mac OS X 的 Python 标准库中就不提供这个模块。在这些平台中，只能利用 Python 标准库的 `tty` 和 `termios` 模块（同样的，这些模块在 Windows 平台中也未提供对应功能）来实现类似的功能。

在应用级的程序代码中，我们几乎从来不应该考虑这种问题；我们更倾向于将程序写得更加平台无关，并依赖库函数来实现在不同系统间的跨越。Python 标准库对于大多

数任务的跨平台能力都提供了极其优秀的支持，但本节任务中的需求，却正好是 Python 标准库没有提供相应的跨平台解决方案的一个例子。

当我们不能在标准库中找到可用的跨平台包或工具时，应该自己设法打包，并作为自己的附加自定义库的一部分。本节的解决方案，不仅解决了一个特别的任务，同时还展示了一种好的通用封装方式。（也可以选择测试 `sys.platform`，但我更喜欢本节给出的方法。）

你自己的库模块在特定系统下导入标准库模块时，应该尝试用 Try 语句，包括其对应的 `except ImportError` 语句，当运行的系统不符合要求时 `except ImportError` 会被激发。在 `except` 语句中，你的库模块可以选择任何其他可以在当前系统工作的方法。在一些比较少见的情况下，你可能会需要两种以上的平台相关的方法，但绝大多数情况下，你都只需准备一个在 Windows 下工作的方法和另一个适用于所有其他平台的方法。这是因为当今的绝大多数非 Windows 平台基本上都是 UNIX 或者类 UNIX。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中 `msvcrt`, `tty` 和 `termios` 的内容。

## 2.24 在 Mac OS X 平台上统计 PDF 文档的页数

感谢：Dinu Gherman、Dan Wolfe

### 任务

你的计算机运行着比较新的 Mac OS X 系统（10.3 的“Panther”或更新的版本），现在需要知道一个 PDF 文档的页数。

### 解决方案

PDF 格式和 Python 都已经集成到了 Mac OS X 系统中（10.3 或更高版本），因而这个问题解决起来也相对比较容易：

```
#!/usr/bin/python
import CoreGraphics
def pageCount(pdfPath):
    "返回指定路径的 PDF 文档的页数"
    pdf = CoreGraphics.CGPDFDocumentCreateWithProvider(
        CoreGraphics.CGDataProviderCreateWithFilename(pdfPath)
    )
    return pdf.getNumberOfPages()
if __name__ == '__main__':
    import sys
    for path in sys.argv[1:]:
        print pageCount(path)
```

## 讨论

另一个完成任务的方法是使用 Python 扩展, PyObjC, 它使得 Python 代码可以利用 Mac OS X 所带的 Foundation 和 AppKit 框架的能力。该方案也可以让你的代码运行在较老版本的 Mac OS X 中, 比如 10.2 Jaguar。不过依赖并使用 Mac OS X 10.3 或更高版本提供的集成 Python 的环境和 CoreGraphics 扩展 (也是 Mac OS X “Panther”的一部分), 可以使我们的代码可直接利用 Apple 强大的 Quartz 图形引擎。

## 更多资料

关于 PyObjC, 参看 <http://pyobjc.sourceforge.net/>; 更多有关 CoreGraphics 模块的资料则请访问 [http://www.macdevcenter.com/pub/a/mac/2004/03/19/core\\_graphics.html](http://www.macdevcenter.com/pub/a/mac/2004/03/19/core_graphics.html)。

## 2.25 在 Windows 平台上修改文件属性

感谢: John Nielsen

### 任务

需要修改 Windows 上一系列文件的属性, 比如将某些文件设置为只读、归档等。

### 解决方案

PyWin32 的 win32api 模块提供了一个 SetFileAttributes 函数, 正好可以用来完成这种任务:

```
import win32con, win32api, os
# 创建一个文件, 并展示如何操纵它
thefile = 'test'
f = open('test', 'w')
f.close()
# 设置成隐藏文件...
win32api.SetFileAttributes(thefile, win32con.FILE_ATTRIBUTE_HIDDEN)
# 设置成只读文件
win32api.SetFileAttributes(thefile, win32con.FILE_ATTRIBUTE_READONLY)
# 为了删除它先把它设成普通文件
win32api.SetFileAttributes(thefile, win32con.FILE_ATTRIBUTE_NORMAL)
# 最后删掉该文件
os.remove(thefile)
```

### 讨论

win32api.SetFileAttributes 的一个有趣的用法是用来删除文件。用 os.remove 在 Windows 中删除非普通文件会遭遇失败。为了删除文件, 必须先通过 Win32 调用 SetFileAttributes

把该文件的属性设置为普通，如同本节代码最后一段所展示的那样。当然，这么做可能会遇到警告，因为一个文件没有被设置为普通文件通常是有很好的理由的。只有当你很明确并很有把握的时候，才应该真正的删掉一个文件。

## 更多资料

查看位于 <http://ASPN.ActiveState.com/ASPN/Python/Reference/Products/ActivePython/PythonWin32Extensions/win32file.html> 的 win32file 模块文档。

## 2.26 从 OpenOffice.org 文档中提取文本

感谢：Dirk Holtwick

### 任务

需要从 OpenOffice.org 文档的文本内容（无论有无 XML 标记）中抽取数据。

### 解决方案

OpenOffice.org 文档其实就是一个聚合了 XML 文件的 zip 文件，遵循一种良好的文档规范。如果只是为了访问其中的数据，我们甚至不用安装 OpenOffice.org：

```
import zipfile, re
rx_stripxml = re.compile("<[^>]*?>", re.DOTALL|re.MULTILINE)
def convert_OO(filename, want_text=True):
    """ 将一个OpenOffice.org文件转换成XML或文本 """
    zf = zipfile.ZipFile(filename, "r")
    data = zf.read("content.xml")
    zf.close()
    if want_text:
        data = " ".join(rx_stripxml.sub(" ", data).split())
    return data
if __name__=="__main__":
    import sys
    if len(sys.argv)>1:
        for docname in sys.argv[1:]:
            print 'Text of', docname, ':'
            print convert_OO(docname)
            print 'XML of', docname, ':'
            print convert_OO(docname, want_text=False)
    else:
        print 'Call with paths to OO.o doc files to see Text and XML forms.'
```

### 讨论

OpenOffice.org 文档就是 zip 文件，并包含了一些其他内容，其中一般都会有 content.xml

文件。所以本节的任务其实是从 zip 文件中抽取数据。本节的方法完全抛开了 XML 标记，只是用了一个简单的正则表达式，用空白符将内容切开，然后在片段间插入一个空格并最后合并起来，以节省空间。当然，我们也可以利用 XML 解析器，以更结构化的手段来获取信息，但是这里我们需要的只是一些文本内容，所以这种快速粗放的方法已经满足需要了。

特别指出一点，正则表达式 rx\_stripxml 匹配的是 XML 标签（开始和结束）的起始符< 和结束符>。在函数 convert\_OO 中，在 if want\_text 语句之后，我们用这个正则表达式将所有的 XML 标签替换为空格，然后根据空白符进行切分（调用字符串方法 split，能够切割任何空白符序列），之后再合并（使用 " ".join，用一个空格符作连接串）。这种切割再合并的方法，本质上是把任何空白符序列都变成一个空格符。更多的有关从 XML 文档中提取文本的内容可参看第 12.3 节。

## 更多资料

*Library Reference* 文档中的 zipfile 和 re 模块；OpenOffice.org 的主页 <http://www.openoffice.org/>，12.3。

## 2.27 从微软 Word 文档中抽取文本

感谢：Simon Brunning、Pavel Kosina

### 任务

你想从 Windows 平台下某个目录树中的各个微软 Word 文件中抽取文本，并保存为对应的文本文件。

### 解决方案

借助 PyWin32 扩展，通过 COM 机制，可以利用 Word 来完成转换：

```
import fnmatch, os, sys, win32com.client
wordapp = win32com.client.gencache.EnsureDispatch("Word.Application")
try:
    for path, dirs, files in os.walk(sys.argv[1]):
        for filename in files:
            if not fnmatch.fnmatch(filename, '*.doc'): continue
            doc = os.path.abspath(os.path.join(path, filename))
            print "processing %s" % doc
            wordapp.Documents.Open(doc)
            docastxt = doc[:-3] + 'txt'
            wordapp.ActiveDocument.SaveAs(docastxt,
                FileFormat=win32com.client.constants.wdFormatText)
            wordapp.ActiveDocument.Close()
```

```
finally:  
    # 确保即使有异常发生 Word 仍能被正常关闭  
    wordapp.Quit( )
```

## 讨论

关于 Windows 应用程序的一个有趣的地方是，可以通过 COM 以及 Python 提供的 PyWin32 扩展，编写一些简单的脚本对这些应用程序进行控制。这个扩展允许你用 Python 脚本来完成各种 Windows 下的任务。本节的脚本，从目录树下的所有的 Word 文档（即.doc 文件）中抽取文本，并存为对应的.txt 文本文件。通过使用 os.walk 函数，并利用 for 循环语句，我们无须递归即可遍历树中的所有子目录。通过 fnmatch.fnmatch 函数，可以检查文件名以确认它是否符合我们给出的通配符，这里的通配符是 “.doc”。一旦我们确认了这是一个 Word 文档，我们就用此文件名和 os.path 来得到一个绝对路径，再用 Word 打开它，存为文本文件，然后关闭。

如果没有安装 Word，可能需要完全不同的方法来达成目标。一种可能是使用 OpenOffice.org，它也可以载入 Word 文档。另一种可能是使用可以读取 Word 文档的程序，比如 Antiword，其网址是 <http://www.winfield.demon.nl/>。但这里不准备探讨这两种方式。

## 更多资料

Mark Hammond、Andy Robinson 所著的 *Python Programming on Win32* (O'Reilly) 一书中关于 PyWin32 的介绍；<http://msdn.microsoft.com> 介绍了微软 Word 的对象模型；*Library Reference* 和 *Python in a Nutshell* 中关于 fnmatch 和 os.path 以及 os.walk 的相关章节。

## 2.28 使用跨平台的文件锁

感谢： Jonathan Feinberg、 John Nielsen

### 任务

希望某个能同时运行在 Windows 和类 UNIX 平台的程序具有锁住文件的能力，但 Python 标准库提供的锁定文件的方法却是平台相关的。

### 解决方案

如果 Python 标准库没有提供合适的跨平台解决方案，我们可以自己实现一个：

```
import os  
# 需要 win32all 来工作在 Windows 下 (NT、2K、XP、不包括 9x)  
if os.name == 'nt':  
    import win32con, win32file, pywintypes
```

```
LOCK_EX = win32con.LOCKFILE_EXCLUSIVE_LOCK
LOCK_SH = 0 # 默认
LOCK_NB = win32con.LOCKFILE_FAIL_IMMEDIATELY
__overlapped = pywintypes.OVERLAPPED( )
def lock(file, flags):
    hfile = win32file._get_osfhandle(file.fileno( ))
    win32file.LockFileEx(hfile, flags, 0, 0xfffff0000, __overlapped)
def unlock(file):
    hfile = win32file._get_osfhandle(file.fileno( ))
    win32file.UnlockFileEx(hfile, 0, 0xfffff0000, __overlapped)
elif os.name == 'posix':
    from fcntl import LOCK_EX, LOCK_SH, LOCK_NB
    def lock(file, flags):
        fcntl.flock(file.fileno( ), flags)
    def unlock(file):
        fcntl.flock(file.fileno( ), fcntl.LOCK_UN)
else:
    raise RuntimeError("PortaLocker only defined for nt and posix platforms")
```

## 讨论

当很多程序或线程需要访问一个共享的文件时，应该确保所有的访问是同步的，这样就不会出现两个进程或线程同时修改文件内容的情况。失败的同步访问在某些情况下会完全破坏掉整个文件。

本节代码给出了两个函数，lock 和 unlock，分别用于请求和释放一个文件的锁。对 portalocker.py 模块的使用其实就是简单地调用 lock 函数，传递一个文件给它，再用一个参数来指定需要的锁的类型：

Shared lock（默认）

这种锁会拒绝所有进程的写入请求，包括最初设定锁的进程。但所有的进程都可以读取被锁定的文件。

Exclusive lock

拒绝其他所有进程的读取和写入的请求。

Nonblocking lock

当这个值被指定时，如果函数不能获取指定的锁会立刻返回。否则，函数会处于等待状态。使用 Python 的位操作符，或操作 (|)，可以将 LOCK\_NB 和 LOCK\_SH 或 LOCK\_EX 进行或操作。

举个例子：

```
import portalocker
afile = open("somefile", "r+")
portalocker.lock(afile, portalocker.LOCK_EX)
```

在不同系统中 lock 和 unlock 的实现完全不同。类 UNIX 系统(包括 Linux 和 Mac OS X)中，本节代码的功能依赖于标准的 fcntl 模块。在 Windows 系统中 (NT、2000、XP、Win95 和 Win98 不适用，因为它们并没有提供相应的底层支持)，则使用 win32file 模块，该模块是为 Windows 量身定做的流行的 PyWin32 扩展的一个组成部分，PyWin32 的作者是 Mark Hammond。重点是不管内部实现如何不同，这两个函数（包括需要传递给 lock 函数的标志）在不同的平台下却表现得完全一致。这种基于不同包的实现但却表现出功能一致性的方法，有助于写出跨平台的应用程序，这也正是 Python 的力量所在。

当写跨平台的程序时，最好是将各种功能以平台无关的方式封装起来。比如本节的文件锁部分，对于 Perl 用户是很有帮助的，他们很习惯直接使用系统调用 lock。更普遍的是，虽然 if os.name==不属于应用层面的代码，但是这类平台测试代码却总是应该被安插到标准库或者与应用无关的模块中。

## 更多资料

*Library Reference* 中 fcntl 模块的文档；<http://ASPN.ActiveState.com/ASPN/Python/Reference/Products/ActivePython/PythonWin32Extensions/win32file.html> 中有关 win32file 模块的介绍；Jonathan Feinberg 的主页 (<http://MrFeinberg.com>)。

## 2.29 带版本号的文件名

感谢：Robin Parmar、Martin Miller

### 任务

如果你想在改写某文件之前对其做个备份，可以在老文件的名字后面根据惯例加上三个数字的版本号。

### 解决方案

我们需要编写一个函数来完成备份工作：

```
def VersionFile(file_spec, vtype='copy'):
    import os, shutil
    if os.path.isfile(file_spec):
        # 检查'vtype'参数
        if vtype not in ('copy', 'rename'):
            raise ValueError, 'Unknown vtype %r' % (vtype,)
        # 确定根文件名，所以扩展名不会太长
        n, e = os.path.splitext(file_spec)
        # 是不是一个以点为前导的三个数字？
        if len(e) == 4 and e[1:].isdigit():
            pass
        else:
            v = 1
            while True:
                v += 1
                vstr = '%03d' % v
                newname = n + '.' + vstr + e
                if not os.path.exists(newname):
                    break
    else:
        v = 1
        while True:
            v += 1
            vstr = '%03d' % v
            newname = n + '.' + vstr + e
            if not os.path.exists(newname):
                break
    if vtype == 'copy':
        shutil.copy(file_spec, newname)
    elif vtype == 'rename':
        os.rename(file_spec, newname)
```

```

        num = 1 + int(e[1:])
        root = n
    else:
        num = 0
        root = file_spec
    # 寻找下一个可用的文件版本
    for i in xrange(num, 1000):
        new_file = '%s.%03d' % (root, i)
        if not os.path.exists(new_file):
            if vtype == 'copy':
                shutil.copy(file_spec, new_file)
            else:
                os.rename(file_spec, new_file)
            return True
    raise RuntimeError,"Can't%s%r, all names taken"%(vtype,file_spec)
return False
if __name__ == '__main__':
    import os
    # 创建一个test.txt 文件
    tfn = 'test.txt'
    open(tfn, 'w').close()
    # 对它取 3 次版本
    print VersionFile(tfn)
    # 输出: True
    print VersionFile(tfn)
    # 输出: True
    print VersionFile(tfn)
    # 输出: True
    # 删除我们刚刚生成的 test.txt*文件
    for x in ('', '.000', '.001', '.002'):
        os.unlink(tfn + x)
    # 展示当文件不存在时取版本操作的结果
    print VersionFile(tfn)
    # 输出: False
    print VersionFile(tfn)
    # 输出: False

```

## 讨论

`VersionFile` 函数是为了确保在打开文件进行写入或更新等修改前, 对已存在的目标文件完成了备份 (或者重命名, 由可选的第二个参数来决定)。在处理文件之前进行备份是很明智的举措 (这也是一些人仍然怀念旧的 VMS 操作系统的原因, 这种备份是自动进行的)。实际的复制和重命名是分别由 `shutil.copy` 和 `os.rename` 完成的, 所以唯一的问题是, 怎么确定文件的名字。

一个流行的决定备份的名字的方法是使之版本化 (比如, 给文件名增加一个逐渐增大

的数字)。本节确定文件名的方法是，首先从文件名中分解出名字根(因为有可能这已经是一个版本化的文件名了)，然后在这个名字根之后添加进一步的扩展名，比如.000, .001, 等等，直到以此种命名方式确定的文件名也无法对应任何一个存在的文件。注意，`VersionFile` 被限制为只能有 1 000 个版本，所以需要有个归档备份的计划。在进行版本化之前，首先要确保该文件存在——不能对不存在的东西进行备份。如果文件不存在，`VersionFile` 函数只是简单地返回 `False` (如果文件存在而且函数执行无误则返回 `True`)，所以在调用之前也无须检查文件是否存在。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中关于 `os` 和 `shutil` 模块的文档。

## 2.30 计算 CRC-64 循环冗余码校验

感谢：Gian Paolo Ciceri

### 任务

需要对某些数据进行循环冗余码校验 (CRC) 以确定数据的完整无误，而且必须遵循 ISO-3309 关于 CRC-64 校验的规定。

### 解决方案

Python 标准库并没有提供 CRC-64 的任何实现(但提供了 CRC-32 函数，即 `zlib.crc32`)，所以我们需要自己来提供实现。幸而 Python 能够处理位操作(与、或、非、异或、移位等)，就像 C 那样(实际上，它们的语法几乎完全相同)，所以很容易根据 CRC-64 的参考实现写出如下的 Python 函数：

```
# 使用两个辅助表(为了速度我们用了一个函数),
# 之后删除该函数, 因为我们再也用不着它了:
CRCTableh = [0] * 256
CRCTablel = [0] * 256
def _inittables(CRCTableh, CRCTablel, POLY64REVh, BIT_TOGGLE):
    for i in xrange(256):
        partl = i
        parth = 0L
        for j in xrange(8):
            rflag = partl & 1L
            partl >>= 1L
            if parth & 1L:
                partl ^= BIT_TOGGLE
            parth >>= 1L
            if rflag:
                parth ^= POLY64REVh
```

```

        CRCTableh[i] = parth
        CRCTablel[i] = partl
    # CRC64 的高 32 位的生成多项式 (低 32 位被假设为 0)
    # 以及_inittables 所用的 bit-toggle 掩码
    POLY64REVh = 0xd8000000L
    BIT_TOGGLE = 1L << 31L
    # 运行函数来准备表
    _inittables(CRCTableh, CRCTablel, POLY64REVh, BIT_TOGGLE)
    # 删除我们不需要的名字, 包括生成表的函数
    del _inittables, POLY64REVh, BIT_TOGGLE
    # 此模块公开了这两个函数: crc64 和 crc64digest
    def crc64(bytes, (crch, crcl)=(0,0)):
        for byte in bytes:
            shr = (crch & 0xFF) << 24
            templh = crch >> 8L
            templl = (crcl >> 8L) | shr
            tableindex = (crcl ^ ord(byte)) & 0xFF
            crch = templh ^ CRCTableh[tableindex]
            crcl = templl ^ CRCTablel[tableindex]
        return crch, crcl
    def crc64digest(aString):
        return "%08X%08X" % (crc64(bytes))
if __name__ == '__main__':
    # 当此模块作为主脚本运行时, 一个小测试/展示
    assert crc64("IHATEMATH") == (3822890454, 2600578513)
    assert crc64digest("IHATEMATH") == "E3DCADD69B01ADD1"
    print 'crc64: dumb test successful'

```

## 讨论

循环冗余码校验 (CRC) 是一种流行的确保数据 (例如, 文件) 未被损坏的方法。CRC 可以稳定地检查出一些随机偶发的损坏, 但是对于恶意的针对性攻击并不像其他一些加密的校验和方法那么强劲。CRC 的计算比其他校验和方式都要快, 因此在那些只需要检测偶发和随机损坏, 而不用担心别人故意伪造数据进行欺骗的情况下, 它比其他校验方式用得更多。

从数学的角度讲, CRC 是把需要校验的数据的位当做多项式进行计算的。实际上, 正如本节代码所示, 经过正确的索引处理, 计算可以一次完成并将结果存储在表中, 数据中的每一个字节都对最终的结果产生影响。这样, 在初始化之后 (我们用一个辅助函数来初始化, 这是因为在 Python 中使用局部变量进行计算要比使用全局变量计算快得多), CRC 计算的速度非常快。表的计算和相关的处理用到了很多位操作, 不过, 幸运的是, Python 对这类操作的处理效果和其他语言一样, 比如 C, 速度非常快。(实际上 Python 关于位操作的语法和 C 完全一样。)

这个标准的 CRC-64 校验和的算法在 ISO-3309 标准中有详细说明，本节的实现方法完全地遵照了该标准的描述。生成器多项式是  $x^{64} + x^4 + x^3 + x + 1$ 。（在“更多资料”小节中会提供更多的关于其计算方法的信息。）

用一对 Python 的 int 类型来存储 64 位的结果，分别代表其高 32 位和低 32 位。为了能够递进地计算 CRC——有时数据可能是逐步到达的，给 `crc64` 调用函数提供了一个可选的“初始值”，即(`crch`, `crcl`)构成的数对，这样可以在前一步计算结果的基础上继续计算。如果要一次计算出全部数据的 CRC，只需要提供完整的数据（字节的序列），如本节中一样，同时数对会被默认地设置为(0, 0)。

## 更多资料

W.H. Press, S.A. Teukolsky, W.T. Vetterling, 以及 B.P. Flannery 所著的 Numerical Recipes in C, 2d ed. (Cambridge University Press), pp. 896ff。

# 时间和财务计算

### 引言

感谢: Gustavo Niemeyer、Facundo Batista

今天、上周、明年。这些词听起来是如此的普通。你可能会想知道，我们的生活与时间观念的关联究竟有多深。关于时间的概念无处不在，当然，它们也在大多数的软件中有所体现。即使是一些非常简单的程序都可能与时间有关，比如时间戳、延迟、超时、速度测量、日历等。为了满足通用程序的这类需求，Python 标准库提供了坚实的基础支持，而更多的其他支持则来自于第三方模块和包。

涉及到财务的计算则是另一个引人注意的有趣问题，因为它与我们的日常生活联系非常紧密。Python 2.4 引入了对十进制数字的支持（当然也可以在 Python 2.3 中引入，参看 [http://www.taniquetil.com.ar/facundo/bdvfiles/get\\_decimal.html](http://www.taniquetil.com.ar/facundo/bdvfiles/get_decimal.html)），这使得用户可以避免使用二进制浮点数，也使得 Python 成为了执行这类计算的一个很好的选择。

本章将覆盖着两个主题，财务和时间。我们也许应该说这一章其实只有一个主题，毕竟大家都知道一句老话，时间就是金钱。

### 时间模块

Python 标准库的时间模块使得 Python 应用程序可以利用和借助其运行平台提供的各种与时间相关的功能。因此，在你的平台中，提供等价功能的 C 库文档可能会很有用，而且，在一些比较奇特的平台上，Python 也可能会受到平台自身的一些影响。

时间模块中最常用的一个函数就是获取当前时间的函数 `time.time`。在未初始化的情况下其返回值看起来实在是不够直观：一个浮点数，代表了从某个特定时间点——也被称为纪元（epoch）——开始所经历的秒数，这个时间点根据不同的平台也可能会有些不同，但通常都是 1970 年 1 月 1 日午夜。

要检查平台所使用的纪元，可以在 Python 交互式解释器的提示符下输入下面语句：

```
>>> import time  
>>> print time.asctime(time.gmtime(0))
```

注意，我们给 `time.gmtime` 函数传递了参数 0（表示从纪元之后 0 秒开始）。`time.gmtime` 将任何时间戳（从纪元开始所经历的秒数）转化为一个元组，该元组代表了人类容易理解的一种时间格式，而无须进行任何时区转化（GMT 代表了“格林威治标准时间”，也就是大家通常所知的 UTC，“世界标准时间”的另一种说法）。也可以传递一个时间戳（从纪元开始所经历的秒数）给 `time.localtime`，它会根据当前时区进行时间转化。

理解这个区别很重要，如果获得一个已经根据当地时间进行调整了的时间戳，将其传递给 `time.localtime` 函数，将不会获得一个预期的结果——除非非常走运，你的当前时区恰好就是 UTC 时区。

这里给出一个从返回的元组中获取当前本地时间的方法：

```
year, month, mday, hour, minute, second, wday, yday = time.localtime()
```

虽然代码能运行，但是却不是很优雅，最好不要经常这么用。也可以完全避免用这种方式获取时间，因为 `time` 函数返回的元组提供了有意义的属性名，更加易于使用。比如，获取当前月份，就可以写成简洁而优雅的一行：

```
time.localtime().tm_mon
```

注意，我们忽略了传递给 `localtime` 的参数。当调用 `localtime`、`gmtime` 或者 `asctime` 而不提供参数时，默认使用当前时间。

时间模块中两个非常有用的函数是 `strftime`——它可以根据返回的时间元组构建一个字符串，以及 `strptime`——与前者完全相反，它将解析给定的字符串并产生一个时间元组。这两个函数都接受一个格式化字符串，可以指明真正感兴趣的部分（或者你希望从字符串中解析出的部分）。关于传递给这两个函数的格式字符串的更多信息和细节请参看 <http://docs.python.org/lib/module-time.html>。

时间模块中最后一个重要的函数是 `time.sleep`，它使得可以在 Python 程序中实现延时。虽然这个函数的 POSIX 对应版本只接受一个整数参数，Python 的版本则支持一个浮点数，并允许非整数秒的延时。比如：

```
for i in range(10):  
    time.sleep(0.5)  
    print "Tick!"
```

这段代码大概耗时 5s，并不断打印出“Tick！”，大约每秒两次。

## 时间和日期对象

除了非常有用的时间模块，Python 标准库也同时引入了 `datetime` 模块，该模块提供了

能更好地对应于抽象的日期和时间的各种类型，比如 time、date 和 datetime 类型。构造这些类型实例的代码也非常优雅和简单：

```
today = datetime.date.today()
birthday = datetime.date(1977, 5, 4)      #5月4日
currenttime = datetime.datetime.now().time()
lunchtime = datetime.time(12, 00)
now = datetime.datetime.now()
epoch = datetime.datetime(1970, 1, 1)
meeting = datetime.datetime(2005, 8, 3, 15, 30)
```

更进一步，正如我们所料，通过属性和方法，这些类型提供了很方便的获取和操作信息的方法。下面的代码创建了一个 date 类型，代表当前日期，然后获得下一年的同样的日期，最后将结果打印出来：

```
today = datetime.date.today()
next_year = today.replace(year=today.year+1).strftime("%Y.%m.%d")
print next_year
```

注意年是怎样递增的，以及 replace 方法的使用。直接给 date 和 time 实例的属性赋值这一想法颇具诱惑力，但实际上这些实例是无法改写的（这其实是好事，意味着我们可以将这些实例用作集合的成员，或者字典的键），所以，新实例只能通过创建获得，而无法通过修改一个旧的实例来达成。

datetime 模块通过 timedelta 类型为时间差（即两个时间实例的差值；可以把它想象为一段时间）提供了一些基本支持。这个类型允许你使用给定的时间片，在一个指定的日期上增减时间，同时这个类型也可以作为 time 和 date 实例的差值计算结果。

```
>>> import datetime
>>> NewYearsDay = datetime.date(2005, 01, 01)
>>> NewYearsEve = datetime.date(2004, 12, 31)
>>> oneday = NewYearsDay - NewYearsEve
>>> print oneday
1 day, 0:00:00
>>>
```

timedelta 实例在内部被表示为天数、秒数和微秒数，但也可以通过直接提供这些参数或者其他参数，比如分钟数、小时数和周数来构建一个 timedelta 实例。其他类型的差值，比如月的，年的，则故意没有提供，因为它们的含义和操作结果等，还存在一些争议。（但第三方 dateutil 包则提供了这些特性，见 <https://moin.conectiva.com.br/DateUtil>）。

datetime 的设计很有远见，同时也很谨慎。其策略是，不实现难以预测的任务以及那些在不同的系统中需要用到不同实现的任务，不强求做到无所不能。当前的实现，已经提供了良好的接口，并适用于大多数情况，更重要的是，还提供了一个坚实的基础以便第三方模块可以在其基础上继续发展。

另一个反映 datetime 设计谨慎的地方是该模块对时区的支持。虽然 datetime 提供了很好

的查询和设置时区信息的方法，但如果没有一个外部的来源来提供 `tzinfo` 类型的非抽象化的子类，那些方法不会有什么用。至少有两个第三方包为 `datetime` 提供了时区支持：前面已经提到过的 `dateutil` 以及 `pyTZ`，见 <http://sourceforge.net/projects/pytz/>。

## 十进制

`decimal` 是 Python 标准库提供的模块，也是 Python 2.4 新引入的内容，并最终给 Python 带来了十进制数学计算。感谢 `decimal`，我们现在终于拥有了十进制数数据类型，具有有界精度和浮点。让我们来仔细看看这三个方面：

### 十进制数字数据类型

数不是被储存为二进制，而是存为一个十进制数字的序列。

### 有界精度

用于存储数的数字的数目是固定的。（对于每个十进制数对象而言，那是一个固定的参数，但是不同的十进制数对象也可以被设置为使用不同数目的数字。）

### 浮点

十进制小数点并没有一个固定的位置。（或者这样讲：所有数字的总数是固定的，小数点后面的数字并没有一个固定的数目。如果数目是固定的，那应该叫做固点数——而非浮点数——数据类型。）

这样的数据类型有很多用途（最大用途就是用于财务计算），特别是，`decimal.Decimal` 比标准的二进制 `float` 提供了更多的高级功能。最大的优点是，所有用户输入的数（也就是所有的具有有限位数字的数），都能够被精确地表示出来（作为反例的是，用二进制浮点存储用户输入的数据，常常不能够严格按照原样表示出来）：

```
>>> import decimal
>>> 1.1
1.100000000000001
>>> 2.3
2.299999999999998
>>> decimal.Decimal("1.1")
Decimal("1.1")
>>> decimal.Decimal("2.3")
Decimal("2.3")
```

这种完全一致的表示也可用于计算。而关于二进制浮点，举个例子：

```
>>> 0.1 + 0.1 + 0.1 - 0.3
5.5511151231257827e-17
```

虽然其差异极小，接近于 0，但是这种微小差异却不利于我们进行可靠的相等比较运算；另外，这些微小的差异值还会不断累积。基于这个原因，对于那些涉及账目计算，对相等比较计算要求很严格的程序，`decimal` 比二进制浮点更加适用：

```
>>> d1 = decimal.Decimal("0.1")
>>> d3 = decimal.Decimal("0.3")
>>> d1 + d1 + d1 - d3
Decimal("0.0")
```

我们可通过整数、字符串或者元组构建 `decimal.Decimal`。要从一个浮点数创建一个 `decimal.Decimal`，首先需要将浮点转化为字符串。这是必要的一步，我们在这里可以显式地指明转化的细节，甚至包括了对错误的表示。十进制数字包括了一些特殊的值，比如 `NaN`（代表了“非数字”）、正负无穷大，还有`-0`。一旦创建成功，`decimal.Decimal` 对象也是无法修改的，就像 Python 中的其他数字一样。

`decimal` 模块本质上只是实现了我们在学校学到的一些数学运算法则。对于指定的精度要求，它总是尽可能给出没有截断的结果：

```
>>> 0.9 / 10
0.08999999999999997
>>> decimal.Decimal("0.9") / decimal.Decimal(10)
Decimal("0.09")
```

当结果中的数字的数目超过了精度要求，则根据当前的舍入方式对结果进行舍入处理。有几种舍入方式；但默认的是 `round-half-even` 方式（译者注：四舍六入，如果是 5，则舍入到最接近的偶数。举个例子：2.45 只保留一位小数，结果是 2.4）。

`decimal` 模块引入了有效位的概念，例如 `1.30+1.20` 结果是 `2.50`。末尾的 0 是为了指出有效位。对于涉及到财务计算的应用程序，这是经常采用的表示方法。对于乘法，“教科书式的”方法会使用乘数中的所有数字：

```
>>> decimal.Decimal("1.3") * decimal.Decimal("1.2")
Decimal("1.56")
>>> decimal.Decimal("1.30") * decimal.Decimal("1.20")
Decimal("1.5600")
```

和其他内建数字类型一样，如 `float` 和 `int`，`decimal` 对象拥有数的一些标准属性，除此之外它还有一些特殊的方法。具体信息请参看文档中对其方法的介绍以及相关的示例。

`decimal` 数据类型通常都工作在一个具体的环境中，也就是说一些配置信息已经被预先设定了。每个线程都有它自己的当前环境（拥有独立的上下文环境意味着每个线程都可以做自己的改变和设定而不互相影响），通过 `decimal` 模块提供的 `getcontext` 和 `setcontext` 函数，我们可以访问和修改当前线程的环境。

不像基于硬件的二进制浮点数，`decimal` 模块的精度可以由用户设置（默认是 28 位）。对于任意一个给定的问题，它都可以被设置得足够大：

```
>>> decimal.getcontext().prec = 6                      # 精度被设为 6...
>>> decimal.Decimal(1) / decimal.Decimal(7)
```

```
Decimal("0.142857")
>>> decimal.getcontext().prec = 60 # ...被设为 60 个数字
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal("0.142857142857142857142857142857142857142857142857142857142857142857")
```

但 `decimal` 中还有一些不是那么简单和基本的东西。本质上，`decimal` 实现了标准的数学计算，可以在 <http://www2.hursley.ibm.com/decimal/> 查到更多细节。这也意味着 `decimal` 支持信号的概念。信号表示在计算中出现的一些不正常的情况（比如， $1/0$ ， $0/0$ ，无穷大/无穷大）。根据各个应用程序的不同需求，有的信号被忽略掉了，有的被用来提供额外信息，有的则被作为异常。对于每个信号，都有一个标志和一个陷阱控制器。当一个信号发生了，它的标志由 0 开始递增，然后如果陷阱控制器被设置为 1，它会抛出一个异常。这给了程序员极大的权力和自由度来配置 `decimal`，以满足他们特定的需求。

`decimal` 的优点如此的多，为什么还有人坚持要用 `float`? 为什么 Python（像很多其他流传广泛的语言一样，但 Cobol 和 Rexx 也是很容易想到的两个例外）会一开始就采用浮点二进制数作为它默认的（也是唯一的）非整数数据类型？当然，理由可以给出一大堆，但最终都归结到一点上，速度！看看下面的实验：

```
$ python -mtimeit -s'from decimal import Decimal as D' 'D("1.2")+D("3.4")'
10000 loops, best of 3: 191 usec per loop
$ python -mtimeit -s'from decimal import Decimal as D' '1.2+3.4'
1000000 loops, best of 3: 0.339 usec per loop
```

对上面的输出做个简单的翻译：在这台机器上（一台老的 1.2 GHz Athlon PC，运行 Linux），Python 每秒可以执行 300 万次 `float` 加法（使用个人计算机提供的支持数学计算的硬件），但只能执行 5000 次 `Decimal` 的加法（完全由软件完成）。

基本上，如果需要对非整数做上百万次的加法，应该坚持用 `float`。在过去，一台普通计算机的运算速度上千倍地落后于现在的计算机的速度（其实那也不是太久之前），因此，当程序运行在廉价的机器上时，即使只是做少量的计算也会有诸多限制。而 Rexx 和 Cobol 诞生于大型机系统，这些系统还没有今天最廉价的计算机运算速度快，价格却是这些计算机的成千上万倍。这类大型机系统的买主也许可以负担得起这种方便而易用的十进制数学计算，但绝大多数其他语言，通常诞生于更加廉价的性能有限的机器之上，却无法负担这种开销。

幸运的是，需要对非整数进行大量数学计算的应用程序的数量相对还比较少，在当今的普通计算机上，适量的十进制数学计算也不会引起一些可见的性能问题。因此，绝大多数应用程序都可以利用 `decimal` 在各方面的优势，包括那些需要继续运行在 Python 2.3 上的程序。从版本 2.4 之后，`decimal` 已经成为了 Python 标准库的一部分。要了解更多在 Python 2.3 中集成 `decimal` 的细节，参看 [http://www.taniquetil.com.ar/facundo/bdvfiles/get\\_decimal.html](http://www.taniquetil.com.ar/facundo/bdvfiles/get_decimal.html)。

# 3.1 计算昨天和明天的日期

感谢: Andrea Cavalcanti

## 任务

你想获得今天的日期，并以此计算昨天和明天的日期。

## 解决方案

无论何时你遇到有关“时间变化”或者“时间差”的问题，先考虑 timedelta：

```
import datetime
today = datetime.date.today()
yesterday = today - datetime.timedelta(days=1)
tomorrow = today + datetime.timedelta(days=1)
print yesterday, today, tomorrow
#输出: 2004-11-17 2004-11-18 2004-11-19
```

## 讨论

自从 datetime 模块出现以来，这个问题在 Python 邮件列表中频频露面。当首次碰到这个问题时，人们第一个想法是写这样的代码：yesterday = today - 1，但其结果是一个 TypeError: unsupported operand type(s) for -: ‘datetime.date’ and ‘int’。

一些人认为这是个 bug，并暗示 Python 应该能够猜测出他们的意图。然而，Python 的一个指导原则是：“在模糊含混面前拒绝猜测”，这也是 Python 简洁和强大的原因。猜测意味着需要用启发的方式将 datetime 割裂开来，需要猜测你想减去的究竟是 1 天还是 1 秒，再或者干脆是 1 年？

Python 如同它一贯的方式，并不尝试猜测你的意图，而是期待你明确指定你自己的意图。如果想减去长度为 1 天的一个时间差，应当明确地编写相关代码。再或者，想加上长度为 1 秒的时间差值，可以使用 timedelta 配合 datetime.datetime 对象，这样可以用同样的语法编写相关操作。也许对每个任务都想使用这种方法，因为这种方法给了你足够的自由度，同时还保持着简洁直观。考虑下面的片段：

```
>>> anniversary = today + datetime.timedelta(days=365) # 增加 1 年
>>> print anniversary
2005-11-18
>>> t = datetime.datetime.today() # 获得现在的时间
>>> t
datetime.datetime(2004, 11, 19, 10, 12, 43, 801000)
>>> t2 = t + datetime.timedelta(seconds=1) # 增加 1 秒
>>> t2
datetime.datetime(2004, 11, 19, 10, 12, 44, 801000)
```

```
>>> t3 = t + datetime.timedelta(seconds=3600)          # 增加 1 小时  
>>> t3  
datetime.datetime(2004, 11, 19, 11, 12, 43, 801000)
```

如果你想在日期和时间的计算上有点新花样，可以使用第三方包，如 `dateutil`（可以和内建的 `datetime` 协同工作）和经典的 `mx.DateTime`。举个例子：

```
from dateutil import relativedelta  
nextweek = today + relativedelta.relativedelta(weeks=1)  
print nextweek  
# 输出: 2004-11-25
```

然而，“总是应该让它尽可能简单地工作”。为了保持简单，本节解决方案中使用了 `datetime.timedelta`。

## 更多资料

见 <https://moin.conectiva.com.br/DateUtil?action=highlight&value= DateUtil> 的 `dateutil` 文档，*Library Reference* 中关于 `datetime` 的文档。`mx.DateTime` 的资料可在 <http://www.egenix.com/files/python/mxDateTime.html> 找到。

## 3.2 寻找上一个星期五

感谢：Kent Johnson、Danny Yoo、Jonathan Gennick、Michael Wener

### 任务

你想知道上一个星期五的日期（包括今天，如果今天是星期五）并以特定格式将其打印出来。

### 解决方案

通过 Python 标准库的 `datetime` 模块，此任务可轻松完成：

```
import datetime, calendar  
lastFriday = datetime.date.today()  
oneday = datetime.timedelta(days=1)  
while lastFriday.weekday() != calendar.FRIDAY:  
    lastFriday -= oneday  
print lastFriday.strftime('%A, %d-%b-%Y')  
# 输出: Friday, 10-Dec-2004
```

### 讨论

上面的代码片段可帮助我们找到上一个星期五的日期，并以正确的格式打印，无论上个星期五是否在同一个月，甚至不在同一年也没有关系。在这个例子中，我们试图寻

找星期五（包括今天，如果今天是星期五）。星期五的整数表示是 4，但我们应该避免直接依赖和使用这个数字，我们导入 Python 标准库的 `calendar` 模块，并利用它的 `calendar.FRIDAY` 属性（其实这个属性值就是 4）。首先我们创建一个叫做 `lastFriday` 的变量，并将它设定为今天的日期，然后不断地向前对比检查，直到找到某个日期，它的 `weekday` 的值是 4。

一旦找到了需要的日期，使用 `datetime.date` 类的 `strftime` 方法，可以很容易地将其转化为我们需要的格式。

还有另一个方法，看上去也更简洁一点，即利用内建的 `datetime.date.resolution`，而不是显式地创建一个 `datetime.timedelta` 实例来表示一天的时间长度：

```
import datetime, calendar
lastFriday = datetime.date.today()
while lastFriday.weekday() != calendar.FRIDAY:
    lastFriday -= datetime.date.resolution
print lastFriday.strftime('%d-%b-%Y')
```

`datetime.date.resolution` 这个类属性的值和第一段代码中的 `oneday` 变量的值完全一样。不过，`resolution` 可能会让你犯错。`datetime` 模块的不同类的 `resolution` 属性的值也不同——对于 `date` 类，这个值是 `timedelta(days=1)`，但对于 `time` 和 `datetime` 类，其值为 `timedelta(microseconds=1)`。可以将它们混合搭配（比如，给 `datetime.datetime` 实例加上一个 `datetime.date.resolution`），但有时却很容易混淆并误用。本节解决方案中用了一个命名的 `onday` 变量，更加通用也更加明确，没有任何可能引起误解的地方。因此，这种方式也更加具有 Python 风格（这也是为什么它被当做“正式”的解决方案的原因）。

还有一点可以改进，连循环都可以省略，所以也不用在循环中每次减去一天并做比较：借助模运算，可以一次计算出需要减去的天数：

```
import datetime, calendar
today = datetime.date.today()
targetDay = calendar.FRIDAY
thisDay = today.weekday()
deltaToTarget = (thisDay - targetDay) % 7
lastFriday = today - datetime.timedelta(days=deltaToTarget)
print lastFriday.strftime('%d-%b-%Y')
```

如果对这段代码如何工作还有疑问，也许需要复习一下模运算的知识，参看：  
<http://www.cut-the-knot.org/blue/Modulo.shtml>。

但应该使用你认为最清楚明白的方法，而不用担心性能问题。还记得 Hoare 的名言吗（总是被错误的归为 Knuth 的言论，但实际上他只是引用 Hoare）：“过早优化是万恶之源。”让我们看看为什么在这里的优化还太早。

减去公共的计算部分（计算当前日期，格式化并打印），在一台 4 岁高龄的运行着 Linux 和 Python 2.4 的老人计算机上，最慢的方法（也就是被用作“解决方案”的方法，因

为它足够清晰直观) 耗时  $18.4\mu\text{s}$ ; 最快的方法(避免循环, 并使用各种技巧使之提速)耗时  $10.1\mu\text{s}$ 。

真的需要很频繁地运行这段代码吗? 以至于那  $8\mu\text{s}$  的差异都变得很重要(如果用当前较新的硬件平台, 这个差异值还会变得更小)? 如果要考虑计算今天的日期和格式化结果的开销, 还需要再加上  $37\mu\text{s}$ , 包括了 print 语句的 I/O 开销; 这样, 最慢的也是最清晰的方法将耗时  $55\mu\text{s}$ , 而最快的, 也是最精炼的方式, 耗时  $47\mu\text{s}$ , 这点差异实在是不值得担心。

## 更多资料

*Library Reference* 中 datetime 模块和 strftime 的文档(见 <http://www.python.org/doc/lib/module-datetime.html> 和 <http://www.python.org/doc/current/lib/node208.html>)。

## 3.3 计算日期之间的时段

感谢: Andrea Cavalcanti

### 任务

给定两个日期, 需要计算这两个日期之间隔了几周。

### 解决方案

标准的 datetime 和第三方的 dateutil 模块(准确地说是 dateutil 的 rrule.count 方法)很容易使用。在导入了正确的模块之后, 任务变得非常简单:

```
from dateutil import rrule
import datetime
def weeks_between(start_date, end_date):
    weeks = rrule.rrule(rrule.WEEKLY, dtstart=start_date, until=end_date)
    return weeks.count()
```

### 讨论

函数 weeks\_between 用一个开始日期和一个结束日期作为参数, 并实例化一个规则用于计算两者之间的周数, 最后返回此规则的 count 方法的结果——写代码比描述这个过程简单多了。这个方法只返回一个整数(它不可能是所谓的“半”周)。举个例子, 8 天被认为是 2 周。下面做个测试:

```
if __name__=='__main__':
    starts = [datetime.date(2005, 01, 04), datetime.date(2005, 01, 03)]
    end = datetime.date(2005, 01, 10)
    for s in starts:
        days = rrule.rrule(rrule.DAILY, dtstart=s, until=end).count()
        print "%d days shows as %d weeks "% (days, weeks_between(s, end))
```

测试结果是：

```
7 days shows as 1 weeks
8 days shows as 2 weeks
```

如果不喜欢，就没有必要给递归规则指定名字，而是修改函数体，比如下面这样，用一条语句完成所有操作：

```
return rrule.rrule(rrule.WEEKLY, dtstart=start_date, until=end_date).count()
```

这样也工作得很好。但坦率地说，我还是倾向于给递归规则指定一个名字，不然我会觉得不习惯。当然代码已经很好用了，这也许只是我的个人偏好。

## 更多资料

参考 dateutil 模块的文档，见 <https://moin.conectiva.com.br/DateUtil?action=highlight&value=DateUtil>，以及 *Library Reference* 中 datetime 的文档。

## 3.4 计算歌曲的总播放时间

感谢：Anna Martelli Ravenscroft

### 任务

你想获取一个列表中的所有歌曲的播放时间之和。

### 解决方案

我们使用 datetime 标准模块和内建的 sum 函数来完成这个任务：

```
import datetime
def totaltimer(times):
    td = datetime.timedelta(0)          # 将总和初始化（必须是 timedelta）
    duration = sum([
        datetime.timedelta(minutes=m, seconds=s) for m, s in times],
        td)
    return duration
if __name__ == '__main__':
    times1 = [(2, 36),                  # 测试模块是否作为主脚本运行
              (3, 35),                  # 列出包含的元组（分，秒）
              (3, 45),]
    times2 = [(3, 0),
              (5, 13),
              (4, 12),
              (1, 10),]
    assert totaltimer(times1) == datetime.timedelta(0, 596)
    assert totaltimer(times2) == datetime.timedelta(0, 815)
```

```
print ("Tests passed.\n"
       "First test total: %s\n"
       "Second test total: %s" % (
           totaltimer(times1), totaltimer(times2)))
```

## 讨论

在工作之余，我喜欢唱歌，我有很长的一个歌曲列表。我希望能够从中挑选一些歌曲并获得它们的总播放时间，而无须事先建立一个新的播放列表。我写了一个脚本来完成这个任务。

当计算两个 `datetime` 对象之间的差异时，一般返回一个 `datetime.timedelta` 对象，可以创建你自己的 `timedelta` 实例来代表任何给定的时长（`datetime` 模块中其他的类，比如 `datetime` 类，只是代表一个时间点）。这里，我们需要计算的是总时长，所以很明显，我们需要使用 `timedelta`。

`datetime.timedelta` 可以有很多不同的可选参数：天数、秒数、微秒数、毫秒数、分钟数、小时数和周数。因此，要创建一个实例，需要明确地传递一个参数以避免混淆。如果简单地调用 `datetime.timedelta(m, n)`，而不指明参数，这个类会通过位置来确定参数，把 `m` 和 `n` 当做天数和秒数，从而产生奇怪的结果。（我为这个错误曾经郁闷过一段时间，所以一定要做好测试工作。）

要对一个对象列表，比如 `timedelta` 列表，使用内建的 `sum` 函数，必须给 `sum` 传入第二个参数作为初始化的值——否则，默认的初始值是 0，整数 0。当你试图给它加上一个 `timedelta` 对象时，你会得到一个错误。另外，传入的第一个参数是一个可迭代体，其中的所有对象都应该支持数字加法。（特别指出，字符串不支持数字加法，我在这里强烈建议：不要使用 `sum` 把很多的列表连接起来）在 Python 2.4 中，我们可以将方括号[] 替换成圆括号()，从而使用生成器表达式而不是列表推导来作为 `sum` 的第一个参数，当需要处理几千首歌的时候，这样做会更方便。

至于测试用例，我手工创建了一个元组的列表，每个元素记录了歌曲的时长，具体的数据是分钟数和秒数。脚本还可以再进一步改进加强，比如增强对不同格式的辨识能力（如类似 mm:ss 的格式），甚至增加直接从文件中读取信息或者直接访问音乐库的能力。

## 更多资料

*Library Reference* 中 `sum` 和 `datetime` 部分。

## 3.5 计算日期之间的工作日

感谢：Anna Martelli Ravenscroft

## 任务

你想计算两个日期之间的个工作日，而非天数。

## 解决方案

由于工作日和节日在不同的国家，不同地区，甚至在同一家公司不同部门之间，都会有所不同，所以并不存在一个内建的简单办法来解决这个问题。不过，利用 `dateutil`，配合 `datetime` 对象，完成这个任务也不是很难：

```
from dateutil import rrule
import datetime
def workdays(start, end, holidays=0, days_off=None):
    if days_off is None:
        days_off = 5, 6                      # 默认: 周六和周日
    workdays = [x for x in range(7) if x not in days_off]
    days = rrule.rrule(rrule.DAILY, dtstart=start, until=end,
                       byweekday=workdays)
    return days.count() - holidays
if __name__ == '__main__':
    # 主脚本运行时的测试代码
    testdates=[(datetime.date(2004, 9, 1), datetime.date(2004, 11, 14), 2),
               (datetime.date(2003, 2, 28), datetime.date(2003, 3, 3), 1),]
    def test(testdates, days_off=None):
        for s, e, h in testdates:
            print 'total workdays from %s to %s is %s with %s holidays'%(s, e, workdays(s, e, h, days_off), h)
    test(testdates)
    test(testdates, days_off=[6])
```

## 讨论

这也是我的第一个 Python 项目：给定一个开始日期和一个结束日期（含），我需要为培训者们确切地计算出培训天数。在 Python 2.2 中这个问题可能会有点麻烦；而现在，在 `datetime` 模块和第三方包的 `dateutil` 的帮助下，这个问题简单多了。

函数 `workdays` 给变量 `day_off` 赋了一个合适的默认值（除非明确地传入了一个参数作为 `dayoff` 的值），这个值其实是一个非工作日的序列。在我的公司里，不同的人有不同的非工作日，但非工作日数量上通常都是少于工作日的，所以记录和修改非工作日总的来说要更容易一些。我把非工作日当做一个参数，是为了在遇到不同需求时可以给 `days_off` 传递不同的值。接着，这个函数使用列表推导来创建一个实际的工作日列表，也就是那些不在非工作日列表 `days_off` 中的日子。之后，函数就可以进行实际计算了。

本节例子中的主要变量叫做 `days`，它是 `dateutil` 的 `rrule`（递归规则）类的一个实例。我们可以给 `rrule` 类传入不同的参数以创建不同的规则对象。在本例中，我传递了一个常

用的(`rrule.DAILY`)的规则，指定了起始日期和结束日期——它们必须都是`datetime.date`对象，同时还指定了需要统计在内的工作日(`weekdays`)。最后，根据这个规则，我只需要调用`days.count`方法来计算符合条件的情况发生了多少次即可。(见第3.3节中`rrule`的`count`方法的其他应用。)

可以很容易地定义自己的周末：给`days_off`传递任何需要的值。在本节中，其默认值是标准的美国周末时间，周六和周日。然而，如果你的公司只需要工作4天，比如，周二到周五，可以传递参数使得`days_off=(5, 6, 0)`。即使如第二个测试所示，容器中实际上只有一天，也要确保传递给`days_off`的值是可迭代对象，如列表或者元组。

还可以做一点简单但有用的加强，比如检查你的开始日期和结束日期是否是周末，并用一个`if/else`语句来处理周末轮班，同时适当地修改`days_off`。当然还有更多可以加强的地方，比如加入病休处理，进行自动的节日查询处理，因而无须直接传入节日的数目(本节的做法)。基于这个目的，第3.6节实现了一个节日列表。

## 更多资料

参考`dateutil`文档，见<https://moin.conectiva.com.br/DateUtil?action=highlight&value=DateUtil>，*Library Reference*中的`datetime`部分；第3.3节中`rrule.count`的其他用法；第3.5节中的自动节日查询。

## 3.6 自动查询节日

感谢：Anna Martelli Ravenscroft、Alex Martelli

### 任务

不同的国家，不同的地区，甚至同一个公司不同的工会，节日都可能有所不同。需要找到一个办法，能够在给定起始日期和结束日期的条件下，自动获取总共的节假日天数。

### 解决方案

给定两个日期，它们之间的节日的日期有可能是不固定的，比如复活节和劳工节(美国)，基于复活节的节日，比如节礼日；以及固定日期的节日，比如圣诞节；或者你们公司的节日(比如CEO的生日)。不过你都可以利用`datetime`和第三方模块`dateutil`处理所有这些节日。

一个灵活的结构，应该能够提取出诸多的可能性，将其分别包装成函数，并在适当的时候调用那些函数：

```
import datetime
from dateutil import rrule, easter
try: set
```

```

except NameError: from sets import Set as set
def all_easter(start, end):
    # 返回在开始和结束日期之间的复活节列表
    easters = [easter.easter(y)
               for y in xrange(start.year, end.year+1)]
    return [d for d in easters if start<=d<=end]
def all_boxing(start, end):
    # 返回在开始和结束日期之间的节礼日列表
    one_day = datetime.timedelta(days=1)
    boxings = [easter.easter(y)+one_day
               for y in xrange(start.year, end.year+1)]
    return [d for d in boxings if start<=d<=end]
def all_christmas(start, end):
    # 返回在开始和结束日期之间的圣诞节列表
    christmases = [datetime.date(y, 12, 25)
                   for y in xrange(start.year, end.year+1)]
    return [d for d in christmases if start<=d<=end]
def all_labor(start, end):
    # 返回在开始和结束日期之间的劳动节列表
    labors = rrule.rrule(rrule.YEARLY, bymonth=9, byweekday=rrule.MO(1),
                         dtstart=start, until=end)
    return [d.date() for d in labors] # 无须测试是否在两个日期之间
def read_holidays(start, end, holidays_file='holidays.txt'):
    # 返回在开始和结束日期之间的假期列表
    try:
        holidays_file = open(holidays_file)
    except IOError, err:
        print 'cannot read holidays (%r):' % (holidays_file,), err
        return []
    holidays = []
    for line in holidays_file:
        # 跳过空行和注释
        if line.isspace() or line.startswith('#'):
            continue
        # 试图解析格式: YYYY, M, D
        try:
            y, m, d = [int(x.strip()) for x in line.split(',')]
            date = datetime.date(y, m, d)
        except ValueError:
            # 检测无效行并继续
            print "Invalid line %r in holidays file %r" % (
                line, holidays_file)
            continue
        if start<=date<=end:
            holidays.append(date)
    holidays_file.close()
    return holidays
holidays_by_country = {
    # 将各个国家代码映射到一系列函数
}

```

```

'US': (all_easter, all_christmas, all_labor),
'IT': (all_easter, all_boxing, all_christmas),
}

def holidays(cc, start, end, holidays_file='holidays.txt'):
    # 从文件中读取可用的假期
    all_holidays = read_holidays(start, end, holidays_file)
    # 加入用函数计算出的假期
    functions = holidays_by_country.get(cc, ())
    for function in functions:
        all_holidays += function(start, end)
    # 消除重复
    all_holidays = list(set(all_holidays))
    # 使用下面两行可以返回一个排序后的列表
    # all_holidays.sort()
    # return all_holidays
    return len(all_holidays)    # 如果想返回列表注释此行

if __name__ == '__main__':
    test_file = open('test_holidays.txt', 'w')
    test_file.write('2004, 9, 6\n')
    test_file.close()
    testdates = [ (datetime.date(2004, 8, 1), datetime.date(2004, 11, 14)),
                  (datetime.date(2003, 2, 28), datetime.date(2003, 5, 30)),
                  (datetime.date(2004, 2, 28), datetime.date(2004, 5, 30)),
                ]
    def test(cc, testdates, expected):
        for (s, e), expect in zip(testdates, expected):
            print 'total holidays in %s from %s to %s is %d(exp %d)' % (
                cc, s, e, holidays(cc, s, e, test_file.name), expect)
            print
    test('US', testdates, (1, 1, 1))
    test('IT', testdates, (1, 2, 2))
    import os
    os.remove(test_file.name)

```

## 讨论

我曾经工作的一家公司里有 3 个不同的工会，根据契约，这 3 个工会也有不同的节假日。同时，我们还得考虑诸如下雪天（snow days）或其他类型的休假，基本上它们也可以等同于节日。为了处理所有这些可能发生变化的节假日，我们最好将标准的节日计算抽取出来，封装成单个的函数，比如我们完成了 all\_easter, all\_labor, 等等。不同类型的计算可以在需要的时候调用。

虽然半开区间（左边界是包括的，但右边界却并不包括）在 Python 中是允许的（而且，在计算上也更具灵活性，出问题的可能也更小），本节仅仅处理闭区间（左右边界都包括）。而那也正是日期区间的概念所定义的，dateutil 也正是基于这个概念工作，所以，这是一个很显然的选择。

每个函数都负责确保返回的结果满足我们的需要：返回在两个日期（含）之间的 `datetime.date` 实例的列表。举个例子，比如 `all_labor`，我们用 `date` 方法，强制将 `dateutil` 的 `rrule` 返回的 `datetime.datetime` 结果转化为 `datetime.date` 实例。

公司也可能会将某天设定为一个节假日（比如下雪天），“只此一次”，并可能会用一个文本文件来记录这些特殊的日子。在本例中，`read_holidays` 函数执行读取文本文件的任务，每行读取一个日期，格式为年、月、日。也可以选择将这个函数重构为更“模糊”的日期解析器，如同第 3.7 节中展示的那样。

如果每次运行程序都需要查询节日很多次，可能想做一些优化工作，让它只读取和解析一次，然后每次在需要的时候调用这些解析结果。不过，“过早优化是万恶之源，” Knuth 引用 Hoare 的话这么说过：通过避免这种“显然的”优化，我们获得了清晰和灵活。假设这些函数是在交互式环境中调用的，而日期文件有可能在两次读取之间被编辑和修改：如果我们每次都不做任何假设地读取整个文件，则完全不需要检查自从上次读取之后文件有无做过任何修改。

由于不同的国家庆祝不同的节日，本节例子中提供了一个很基本的 `holidays_by_country` 字典。可以在因特网上做很多调查并根据需要不断地充实这个字典。很重要的一点是，这个字典允许调用不同的节日函数，根据传递给 `holidays` 函数的不同的国家编码来确定。如果你的公司有很多工会，也可以很容易地创建一个基于工会的字典，传递工会编码而非国家编码给 `holidays`。`holidays` 函数再去调用合适的函数（包括了无条件调用的 `read_holidays` 函数），连接所有的结果，清除重复内容，然后返回列表的长度。如果你乐意，也可以直接返回列表，只需简单地让代码中注释掉的两行开始工作即可。

## 更多资料

3.7 节中的模糊解析；`dateutil` 的文档，见 <https://moin.conectiva.com.br/DateUtil?action=highlight&value=DateUtil>，*Library Reference* 中的 `datetime` 文档。

## 3.7 日期的模糊查询

感谢：Andrea Cavalcanti

### 任务

你的程序需要读取并接受一些并不符合标准的“`yyyy, mm, dd`” `datetime` 格式。

### 解决方案

第三方 `dateutil.parser` 模块给出了一个简单的解答：

```
import datetime
import dateutil.parser
```

```

def tryparse(date):
    # dateutil.parser 需要一个字符串参数：根据一些惯例，我们
    # 可以从 4 种 “date” 参数创建一个
    kwargs = {}                                     # 假设没有命名参数
    if isinstance(date, (tuple, list)):
        date = ''.join([str(x) for x in date]) # 拼接序列
    elif isinstance(date, int):
        date = str(date)                         # 将整数变为字符串
    elif isinstance(date, dict):
        kwargs = date                           # 接受命名参数字典
        date = kwargs.pop('date')                # 带有一个'date'字符串
    try:
        try:
            parsedate = dateutil.parser.parse(date, **kwargs)
            print 'Sharp %r -> %s' % (date, parsedate)
        except ValueError:
            parsedate=dateutil.parser.parse(date,fuzzy=True,**kwargs)
            print 'Fuzzy %r -> %s' % (date, parsedate)
        except Exception, err:
            print 'Try as I may, I cannot parse %r (%s)' % (date, err)
    if __name__ == "__main__":
        tests = (
            "January 3, 2003",                      # 字符串
            (5, "Oct", 55),                         # 元组
            "Thursday, November 18",                 # 没有年的长字符串
            "7/24/04",                             # 带斜线的字符串
            "24-7-2004",                            # 欧式的字符串格式
            {'date':"5-10-1955","dayfirst":True},   # 包括了 keyword 的字典
            "5-10-1955",                            # 日在前，无 keyword
            19950317,                               # 非字符串
            "11AM on the 11th day of 11th month, in the year of our Lord 1945",
            )
        for test in tests:                       # 测试日期格式
            tryparse(test)                      # 尝试解析

```

## 讨论

`dateutil.parser` 的 `parse` 函数可以用于很多数据格式。本节代码中展示了其中的一部分。这个解析器可以处理英语的月名以及两位或者四位的年（带有一些限制）。如果不带名字参数调用 `parse`，它首先尝试用如下的顺序来解析字符串：mm-dd-yy。如果解析的结果不合逻辑，正如例子中的那样，它尝试解析“27-7-2004”这样的字符串，无果。最后它会尝试“yy-mm-dd”。如果传入了 `dayfirst` 或 `yearfirst` 这样的“关键字”（我们在测试中正是这样做的），`parse` 会试图根据关键字进行解析。

本节的测试代码定义了解析器可能会碰到的一些边界测试用例，比如通过一个元组，一个整数（ISO 格式，无空格），甚至一个短语来传入日期。为了测试关键字参数，本

节的 `tryparse` 函数也接受一个字典作为参数，该函数找到“date”键对应的值作为解析的对象，并将其余部分作为关键字参数传递给 `dateutil` 的解析器。

`dateutil` 的解析器能够提供一定程度的“模糊”解析，只要你能够给它一点提示以便确定各部分的处理方式，比如小时（测试中所用的短语包含了 AM）。在正式的编写代码工作中，应该避免依赖模糊解析，或者做一些预处理工作，或者至少提供某种机制来检查需要解析的日期的准确性。

## 更多资料

更多的有关日期解析算法的资料，可参看 `dateutil` 的文档 <https://moin.conectiva.com.br/DateUtil?action=highlight&value=DateUtil>；而关于日期处理部分，可参看 *Library Reference* 中 `datetime` 的文档。

## 3.8 检查夏令时是否正在实行

感谢： Doug Fort

### 任务

你想了解当前时区的夏令时是否正在执行。

### 解决方案

你第一个想法可能是去检查 `time.daylight`，但很遗憾，那不可行。应该这样做：

```
import time
def is_dst():
    return bool(time.localtime().tm_isdst)
```

### 讨论

在我的时区（和大多数其他时区一样），`time.daylight` 永远是 1，因为 `time.daylight` 的含义是，该时区是否在一年中的某些时候会执行夏令时制（Daylight Saving Time, DST），这和当前是否正在实行夏令时制无关。

只有当 DST 正在实行的时候，调用 `time.localtime` 获得元组的最后一项的值才是 1，否则会是 0——这正好就是我们需要检查的。本节将这个检查操作封装为一个函数，调用了内建的 `bool` 类型来确保返回结果是一个优雅的 `true` 或者 `false`，而不是一个粗糙的 1 或者 0——这部分改良不是必须的，但我认为效果还不错。也可以直接访问相关的项，比如 `time.localtime()[-1]`，但是通过 `tm_isdst` 属性来获取信息显然可读性会更好。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中关于 time 模块的内容。

## 3.9 时区转换

感谢: Gustavo Niemeyer

### 任务

假设你现在身处西班牙，你想将发生在中国的某个事件的时间转换为西班牙时间。

### 解决方案

第三方包 dateutil 中对 datetime 提供了时区支持。下面给出一个设置本地时区的方法，并最后打印出当前时间来检查结果的正确性：

```
from dateutil import tz
import datetime
posixstr = "CET-1CEST-2,M3.5.0/02:00,M10.5.0/03:00"
spaintz = tz.tzstr(posixstr)
print datetime.datetime.now(spaintz).ctime()
```

不同时区之间的转换不仅是可能的，而且对于我们来说也是必要的。举个例子，让我们看看如果根据西班牙时间，下一届奥运会什么时候开始：

```
chinatz = tz.tzoffset("China", 60*60*8)
olympicgames = datetime.datetime(2008, 8, 8, 20, 0, tzinfo=chinatz)
print olympicgames.astimezone(spaintz)
```

### 讨论

名为 posixstr 的字符串是西班牙时区的 POSIX 风格的表示法。这个字符串提供了标准时和夏令时的名字 (CST 和 CEST)，它们的偏移量 (UTC+1 和 UTC+2)，以及 DST 开始和结束的确切时间（三月最后一个星期天的凌晨 2 点，以及十月最后一个星期天的凌晨 3 点）。我们还可以检查一下 DST 时区的边界以确保其正确性：

```
assert spaintz.tzname(datetime.datetime(2004, 03, 28, 1, 59)) == "CET"
assert spaintz.tzname(datetime.datetime(2004, 03, 28, 2, 00)) == "CEST"
assert spaintz.tzname(datetime.datetime(2004, 10, 31, 1, 59)) == "CEST"
assert spaintz.tzname(datetime.datetime(2004, 10, 31, 2, 00)) == "CET"
```

所有的这些 asserts 都应该能顺利地通过测试，从而确保时区名在这些不同时间之间的切换是成立的。

注意返回到标准时的时间是凌晨 3 点，而实际的切换时间点却记为凌晨 2 点。这中间有一个小时的差异，凌晨 2 点和凌晨 3 点，未能保持一致。这种情况发生了两次：

一次是在 CEST 时区，还有一次是 CET 时区。目前，通过标准的 Python 日期和时间支持来无歧义地表示这个时刻还不可能。这就是为什么我会推荐你存储无歧义的 UTC 的 `datetime` 实例，并为了显示的目的而进行时区转换。

为了将中国时区转换成西班牙时区，我们使用了 `tzoffset` 来记录这个事实，即中国比 UTC 时间提前了 8 个小时（`tzoffset` 总是用来和 UTC 比较，而不是和某个特定的时区比较）。注意我们是如何根据时区信息创建出 `datetime` 实例的。对于两个不同时区之间的转换，即使给出的时间是基于本地时区的，这一步也总是很必要的。如果你不根据时区信息创建实例，你会得到一个 `ValueError: astimezone( ) cannot be applied to a naive datetime`。在没有时区信息的情况下，创建的 `datetime` 实例会被简化处理——完全忽略掉时区信息。基于这个目的，`dateutil` 提供了 `tzlocal` 类型，创建基于当前平台的本地时区的实例。

除了前面提到的类型，`dateutil` 还提供了 `tzutc`，创建基于 UTC 的实例；`tzfile`，可以使用标准的二进制时区文件；`tzical`，创建基于 iCalendar 时区的实例；当然还有更多其他类型，这里就不详细介绍了。

## 更多资料

`dateutil` 模块的文档，见 <https://moin.conectiva.com.br/DateUtil?action=highlight&value=DateUtil>，以及 *Library Reference* 中 `datetime` 的文档。

## 3.10 反复执行某个命令

感谢： Philip Nunez

### 任务

需要反复地以需要的时间间隔执行某个命令。

### 解决方案

`time.sleep` 函数提供了一个简单的解决办法：

```
import time, os, sys
def main(cmd, inc=60):
    while True:
        os.system(cmd)
        time.sleep(inc)
if __name__ == '__main__':
    numargs = len(sys.argv) - 1
    if numargs < 1 or numargs > 2:
        print "usage: " + sys.argv[0] + " command [seconds_delay]"
        sys.exit(1)
```

```
cmd = sys.argv[1]
if numargs < 3:
    main(cmd)
else:
    inc = int(sys.argv[2])
    main(cmd, inc)
```

## 讨论

可以用本节中的方法来周期性地运行某命令，以完成某种检查（比如轮询），或者执行不断重复的操作，比如让浏览器加载某个内容不断发生变化的 URL，这样可以确保目前的浏览结果是较新的。本节代码创建了一个叫做 `main` 的函数，还有一个由 `if __name__ == '__main__'` 限定的主体部分，这部分只有在脚本作为主脚本运行时才会被执行。主体部分检查命令行的参数，并调用 `main` 函数（或者输出使用方法的信息，如果给的参数数目不对的话）。这是很好的脚本编写结构，同时也使得它的功能可以被其他脚本通过模块导入的方式调用。

`main` 函数接受一个叫做 `cmd` 的字符串参数，那是你想传递给操作系统 shell 用于执行的命令，还有一个可选的时间周期，默认是 60 秒（1 分钟）。`main` 函数主体处于永久的循环之中，它或者用 `os.system` 来执行命令，或者通过 `time.sleep` 来等待（无须消耗资源）。

脚本的主体会检查你传递给脚本的命令行参数，它通过访问 `sys.argv` 来获取那些参数。第一个参数，`sys.argv[0]`，是脚本的名字，当脚本需要确定自己的身份并打印信息时这是很有用的参考。脚本的主体检查一到两个参数。第一个（强制性的）是需要运行的命令。（你可能需要用引号将命令括起来，这是为了防止 shell 解析命令时，将运行命令的参数和脚本的参数混淆起来：最重要的是，加了引号之后，无论里面的内容是什么样的，它都是一个单一的参数。）第二个（可选的）参数是两次运行之间的间隔秒数。如果忽略第二个参数，主体部分会用默认的间隔时间（60s）为周期来调用传入的命令。

注意，如果有第二个参数，主体部分会将它由字符串（`sys.argv` 中所有的元素都是字符串）转化为整数，可以通过调用内建的 `int` 类型完成这一转换：

```
inc = int(sys.argv[2])
```

如果第二个参数是一个无法转换成整数的字符串（比如，它是一个不含数字的字符序列），用上面的方式调用 `int` 会产生一个异常，然后脚本的执行会结束并打印出一些错误信息。正如 Python 的设计哲学那样，“除非明确地指定，错误不应该悄无声息地被略过。”所以，如果允许脚本接受任何字符串作为第二个参数，并在发生转换错误时采取默认的行动，那其实不是一个很好的设计。

如果不愿使用循环和 `time.sleep`，本章还有一些涉及 Python 标准库模块 `sched` 的内容，见第 3.11 节。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中关于标准库模块 os、time 和 sys 的文档；3.11 节。

## 3.11 定时执行命令

感谢：Peter Cogolo

### 任务

需要在某个确定的时刻执行某个命令。

### 解决方案

这正是标准库的 `sched` 模块所针对的任务：

```
import time, os, sys, sched
schedule = sched.scheduler(time.time, time.sleep)
def perform_command(cmd, inc):
    schedule.enter(inc, 0, perform_command, (cmd, inc)) # re-scheduler
    os.system(cmd)
def main(cmd, inc=60):
    schedule.enter(0, 0, perform_command, (cmd, inc)) # 0==right now
    schedule.run()
if __name__ == '__main__':
    numargs = len(sys.argv) - 1
    if numargs < 1 or numargs > 2:
        print "usage: " + sys.argv[0] + " command [seconds_delay]"
        sys.exit(1)
    cmd = sys.argv[1]
    if numargs < 3:
        main(cmd)
    else:
        inc = int(sys.argv[2])
        main(cmd, inc)
```

### 讨论

上述代码提供了和第 3.10 节中代码类似的功能，只不过它没有使用一个简单的轮询，而是使用了标准库的 `sched` 模块。

`sched` 是一个简单灵活却又非常强大的模块，可被用来在未来某个时刻执行某些特定的任务。要使用 `sched`，首先需要创建一个 `scheduler` 实例对象，即代码中的 `schedule`，创建时要带有两个参数。第一个参数是一个被用来调用的函数，此函数确定任务的时间——通常是 `time.time`，该函数返回从某个特定的时间点到现在所经历的秒数。第二

个参数也是一个供调用的函数，用来等待一段时间——通常是 `time.sleep`。也可以传递其他函数，该函数以某种人为的方式衡量时间。举个例子，可以将 `sched` 用于一些模拟程序。不过，以人为的方式干预时间的衡量属于 `sched` 的较高级的应用，本节并未覆盖这方面内容。

一旦有了一个 `sched.scheduler` 实例 `s`，可以通过调用 `s.enter` 来安排某事件的发生时间，即从现在起的第 `n` 秒开始启动（也可以将 `n` 设为 0，这样事件会立即发生），或者调用 `s.enterabs`，以某个给定的绝对时间作为事件启动时间。对于这两种方式，都需要传递时间（相对时间或者绝对时间），优先级（如果有多个事件被计划在同一时间执行，它们必须按照优先级顺序执行，值最小的会最先执行），一个供调用的函数，一个容纳前面函数所需的参数的元组。这两个调用方式都会返回一个事件标识，可以将这个标识存在什么地方，如果你改变了主意，还可以轻易地取消这个计划中的事件——只需将事件标识作为参数，调用 `s.cancel`。不过这又是一个较高级的用法，本节并未涵盖。

安排了一些事件之后，可以调用 `s.run`，它会持续运行，直到计划事件队列变成空的为止。在本节中，我们展示了怎样计划一个周期性反复执行的事件：函数 `perform_command` 每次被执行，首先都会安排在 `inc` 秒之后再次运行它自己，然后才运行指定的系统命令。以这种方式安排任务，计划事件队列永远不会变空，函数 `perform_command` 不断地被周期性调用。这种自我计划安排是一个很重要的概念，不仅仅用于和 `sched` 相关的应用，任何时候，如果你只有一次机会来计划一个事件，而没有周期性的机会来反复做此事，你都可以考虑这种自我计划安排方式。（举个例子，Tkinter 的 `after` 方法也正是以此种方式工作的，可以作为这种自我计划安排的一个应用典型）。

即使对于本节所示这么简单的例子，相比于简单轮询，`sched` 的优势也极其明显。在第 3.10 节中，指定的延迟时间是指从本次 `cmd` 的执行完成之后，到下一次 `cmd` 执行之前。如果 `cmd` 的执行需要较多的时间（这很有可能，比如，某些命令可能需要等待网络返回数据，或者由于某些繁忙的服务器，而不得不等待），那么这些命令其实并不是真正的“周期性的”执行。而本节的时间延迟是指从本次开始运行 `cmd`，到下一次执行 `cmd` 之前，因此周期性是被严格保障的。如果某个特定命令所用的时间超过了 `inc` 秒，那么 `schedule` 会暂时落后于进度，但最终它会慢慢赶上来，只要 `cmd` 的平均执行时间不超过 `inc` 秒：`sched` 永远不会“略过”事件。（如果你确实需要略过某事件，假设这个事件对你而言不再重要了，必须保存原先获得的事件标识，并调用 `cancel` 方法来取消此事件。）

关于本节的结构和主体的一些细节的解释，请参考第 3.10 节。

## 更多资料

第 3.10 节；*Library Reference* 和 *Python in a Nutshell* 中的标准库模块 `os`、`time`、`sys` 和 `sched` 的文档。

## 3.12 十进制数学计算

感谢: Anna Martelli Ravenscroft

### 任务

需要在 Python 2.4 中进行一些简单的数学计算, 但需要的是十进制的结果, 而不是 Python 默认的 float 类型。

### 解决方案

为了从这些简单的计算中获得预期的结果, 必须导入 decimal 模块:

```
>>> import decimal
>>> d1 = decimal.Decimal('0.3')      # 指定一个十进制对象
>>> d1/3                            # 试试除法
Decimal("0.1")
>>> (d1/3)*3                      # 看看能否复原?
Decimal("0.3")
```

### 讨论

Python 的新手 (尤其是那些没有使用其他语言进行二进制浮点计算经验的) 常常会惊异于一些简单计算的结果。举个例子:

```
>>> f1 = .3                         # 赋值为一个浮点数
>>> f1/3                            # 试试除法
0.09999999999999992
>>> (f1/3)*3                      # 看看能否复原?
0.2999999999999999
```

Python 采用二进制浮点数学计算作为默认的计算方式是有很好的理由的。可以读一读 Python FAQ (Frequently Asked Questions) 文档, 见 <http://www.python.org/doc/faq/general.html#why-are-floating-point-calculations-so-inaccurate>, 或者 *Python Tutorial* 的附录部分, 见 <http://docs.python.org/tut/node15.html>。

很多人对于唯一的选择——二进制浮点, 并不满意, 他们希望能够指定精度, 能够使用十进制数学计算, 以便用于财务计算并生成可预期的结果。很多人需要的仅仅是可预测性。(一个真正的数值分析专家, 当然会认为二进制浮点的计算其实是完全可以预测的。如果你们三人的某人正在读本章节, 请跳过此节继续下一节, 谢谢。)(译者注: 我猜作者这句话是对她的三个专家朋友说的, 其他读者略过就好。)

新的 decimal 类型可以通过环境上下文设置进行深入的控制, 这使得可以简单地设置精度以及对结果的截取方法。不过, 如果你所需要的只是让那些简单的数学计算返回可预测的结果, 那么 decimal 默认的环境已经工作得很好了。

有几点请记住：可以传递字符串、整数、元组或者其他 decimal 对象来创建一个新的 decimal 对象，但如果你有一个浮点数 n，你想通过它来创建一个 decimal 对象，请确保传递的是 str(n)，而不是 n。decimal 对象可以和其他整数、长整数、decimal 对象交互（比如，混用于数学运算），但 float 类型不行。这些限制是强制性的。十进制数被引入到 Python 中，是因为它能提供 float 无法提供的精确度和可预测性：如果允许直接从 float 对象构建十进制数，或者允许二者混用于数学计算，就完全违背了设计十进制数的初衷。另一方面，正如你所预料的那样，decimal 对象也可以被强制转化为 float, long 以及 int 类型。

请记住，decimal 仍然是浮点，而非固点。如果需要的是固点，请参考 Tim Peters 的 FixedPoint，见 <http://fixedpoint.sourceforge.net/>。另外，Python 中也没有专门的 money 数据类型，不过可以参考 3.13 节，了解如何基于 decimal 建立你自己的财务数据格式。最后一点，也许不是那么明显（至少对我而言），一个中间计算结果产生了比输入更多的数字，可以保留那些多余的数字，并用于进一步的计算，到最后完成了所有计算（准备显示或者存储结果时）时，再对结果进行截取，或者干脆每一步都可以完成截取。对于这一点，不同的书有不同的建议。不过我倾向于前者，因为它至少更加方便。

如果你还在坚持 Python 2.3，可以通过下载和安装第三方扩展来利用 decimal 模块，见 [http://www.taniquetil.com.ar/facundo/bdvfiles/get\\_decimal.html](http://www.taniquetil.com.ar/facundo/bdvfiles/get_decimal.html)。

## 更多资料

Python Tutorial 的附录 B 中对于浮点数学运算的解释，见 <http://docs.python.org/tut/node15.html>；Python FAQ，<http://www.python.org/doc/faq/general.html#why-are-floating-point-calculations-so-inaccurate>；Tim Peter 的 FixedPoint，见 <http://fixedpoint.sourceforge.net/>；将 decimal 用于货币，参考第 3.13 节；decimal 已经被记录在 Python 2.4 的 *Library Reference* 文档中了，2.3 的使用者也可通过下载安装使用，见 <http://cvs.sourceforge.net/viewcvs.py/python/python/dist/src/Lib/decimal.py>；decimal PEP (Python Enhancement Proposal)，PEP 327，见 <http://www.python.org/peps/pep-0327.html>。

## 3.13 将十进制数用于货币处理

感谢：Anna Martelli Ravenscroft,、Alex Martelli、Raymond Hettinger

### 解决方案

我们可以使用新的 decimal 模块，配以一个修改过的 moneyfmt 函数（原始版本由 Raymond Hettinger 编写，那是 Python 标准库中 decimal 的文档的组成部分）：

```
import decimal
""" 计算意大利支票税 """
```

```

def italformat(value, places=2, curr='EUR', sep='.', dp=',', pos='', neg='-', overall=10):
    """ 将十进制 value 转化为财务格式的字符串
    places: 十进制小数点后面的数字的位数
    curr: 可选的货币符号 (可能为空)
    sep: 可选的分组 (三个一组) 分隔符 (逗号、句号或空白)
    dp: 小数点指示符 (逗号或句号); 当 places 是 0 时
        小数点被指定为空白
    pos: 正数的可选的符号: "+", 空格或空白
    neg: 正数的可选的符号: "-","(", 空格或空白
    overall: 最终结果的可选的总长度, 若长度不够, 左边货币符号
        和数字之间会被填充符占据
    """
    q = decimal.Decimal((0, (1,), -places))           # 2 places --> '0.01'
    sign, digits, exp = value.quantize(q).as_tuple()
    result = []
    digits = map(str, digits)
    append, next = result.append, digits.pop
    for i in range(places):
        if digits:
            append(next())
        else:
            append('0')
    append(dp)
    i = 0
    while digits:
        append(next())
        i += 1
        if i == 3 and digits:
            i = 0
            append(sep)
    while len(result) < overall:
        append(' ')
    append(curr)
    if sign: append(neg)
    else: append(pos)
    result.reverse()
    return ''.join(result)

# 获得计算用的小计
def getsubtotal(subtin=None):
    if subtin == None:
        subtin = input("Enter the subtotal: ")
    subtotal = decimal.Decimal(str(subtin))
    print "\n subtotal:      ", italformat(subtotal)
    return subtotal

# 指定意大利税法函数
def cnpcalc(subtotal):
    contrib = subtotal * decimal.Decimal('.02')
    print "+ contributo integrativo 2%:    ", italformat(contrib, curr='')
    return contrib

```

```

def vatcalc(subtotal, cnp):
    vat = (subtotal+cnp) * decimal.Decimal('.20')
    print "+ IVA 20%:", italformat(vat, curr='')
    return vat
def ritacalc(subtotal):
    rit = subtotal * decimal.Decimal('.20')
    print "-Ritenuta d'acconto 20%:", italformat(rit, curr='')
    return rit
def dototal(subtotal, cnp, iva=0, rit=0):
    totl = (subtotal+cnp+iva)-rit
    print "TOTAL: ", italformat(totl)
    return totl
# 最终计算报告
def invoicer(subtotal=None, context=None):
    if context is None:
        decimal.getcontext().rounding="ROUND_HALF_UP" # 欧洲截断规则
    else:
        decimal.setcontext(context) # 设置上下文环境
    subtot = getsubtotal(subtotal)
    contrib = cnpcalc(subtot)
    dototal(subtot, contrib, vatcalc(subtot, contrib), ritacalc(subtot))
if __name__=='__main__':
    print "Welcome to the invoice calculator"
    tests = [100, 1000.00, "10000", 555.55]
    print "Euro context"
    for test in tests:
        invoicer(test)
    print "default context"
    for test in tests:
        invoicer(test, context=decimal.DefaultContext)

```

## 讨论

意大利的税务计算颇有点繁琐，比本节展示的情况要复杂多了。本节提供的代码仅仅适用于意大利使用发票的用户。我早就厌倦手工处理发票了，所以我写了个简单的脚本来完成计算。最终，在重构之后代码变成了现在这样，它使用了新的 `decimal` 模块，并且遵循一个财务计算的原则，那就是永远不使用二进制浮点。

怎么才能尽量地利用 `decimal` 模块为财务计算服务呢？十进制的数学运算非常直接简单，虽然显示结果时所用的选项没有那么清楚明白。本节的 `italformat` 函数基于 Raymond Hettinger 的 `moneyfmt` 函数，你可在 Python 2.4 的 *Library Reference* 的 `decimal` 模块章节找到。为了更好地生成财报，代码有一些小小的修改。最主要的变化是 `overall` 参数。这个参数使得函数创建一个由 `overall` 指定数字位数的 `decimal`，并在货币符号（如果有的话）和数字之间填充空白符。这将有助于生成标准的预期长度的结果。

注意，我把 `subtotal = decimal.Decimal(str(subtin))` 中的 `subtin` 强制转化为一个字符串。这将允许 `getsubtotal` 函数接受浮点数（当然也包括整数和字符串）为参数——而无须担

心浮点数可能会引发异常。如果你的程序还可能处理元组，也可以重构代码以适应那种需求。对我来说，浮点是 `getsubtotal` 最可能遇到的参数类型，我无须担心元组。

当然，如果想显示美元货币符号，并使用不同的截断规则，也可以很容易地修改代码以符合你的需求。举个例子，为了显示美元符号，只需将 `curr`, `sep` 和 `dp` 的默认值修改为：

```
def USformat(value, places=2, curr='$', sep=',', dp='.', pos='', neg='-', overall=10):
    ...
    ...
```

如果经常需要处理很多不同的货币，也可以重构此函数，使得它可以在字典中寻找一些合适的参数，或者能够用其他方法给它传递正确的参数。在理论上，使用 Python 标准库中的 `locale` 模块应当是最好的用于确定使用者喜好以及关于财务信息的方法，但实际上我觉得使用 `locale` 并不能给我帮什么忙，不过在这里，我很希望你能够独立实现这个额外的任务，作为检验学习成果的一个练习。

不同的国家通常有不同的截断规则：`decimal` 使用 `ROUND_HALF_EVEN` 作为默认的截断方式。然而，欧洲的规矩应当是 `ROUND_HALF_UP`。为了使用不同的截断规则，如代码所示，只需修改上下文环境即可。最终结果可能不会有太大的变化，但是必须要清楚，修改截断规则（可能影响很小，但是不能忽略）的确会引起差异。

还可以更深入地修改上下文环境，可以创建并设置自己的 `context` 类实例。无论你以哪种方式影响环境，通过简单的 `getcontext` 修改属性，或者通过 `setcontext(mycontext)` 传入一个定制的 `context` 类实例，这些修改对于活动的线程都会即时生效，直到你再次修改上下文环境。如果你考虑要在正式的工作代码（或者为了你自己的家务管理目的）中使用 `decimal`，请确保根据你的国家的账务统计惯例使用正确的上下文环境（即，正确的截断规则）。

## 更多资料

参考 Python 2.4 的 *Library Reference* 关于 `decimal` 的文档，尤其是 `decimal.context` 部分。

## 3.14 用 Python 实现的简单加法器

感谢：Brett Cannon

### 任务

你想用 Python 制作一个简单的加法器，使用精确的十进制数（而不是二进制浮点数），并以整洁的纵列显示计算结果。

### 解决方案

为了执行计算，必须使用 `decimal` 模块。我们的程序接受输入行，每行由一个数字紧跟

一个数学运算符组成，空行表示收到了计算当前结果的请求，输入 q 则表示结束当前程序：

```
import decimal, re, operator
parse_input = re.compile(r'''(?x)
    (\d+\.\?\d*)
    \s*
    ([+-/*])
    \$''') # 允许 RE 中的注释和空白符
# 带有可选的小数部分的数
# 可选的空白符
# 运算符
# 字符串结束

oper = { '+': operator.add, '-': operator.sub,
          '*': operator.mul, '/': operator.truediv,
        }
total = decimal.Decimal('0')
def print_total():
    print '== == =\n', total
print """Welcome to Adding Machine:
Enter a number and operator,
an empty line to see the current subtotal,
or q to quit: """
while True:
    try:
        tape_line = raw_input().strip()
    except EOFError:
        tape_line = 'q'
    if not tape_line:
        print_total()
        continue
    elif tape_line == 'q':
        print_total()
        break
    try:
        num_text, op = parse_input.match(tape_line).groups()
    except AttributeError:
        print 'Invalid entry: %r' % tape_line
        print 'Enter number and operator, empty line for total, q to quit'
        continue
    total = oper[op](total, decimal.Decimal(num_text))
```

## 讨论

Python 的交互式解释器是很有用的计算器，但一个简单的“加法器”也还是有用的。举个例子，像 2345634+2894756-2345823 这样的表达式很不易读，所以检查用于计算的输入数字并不像想象的那么简单。而一个简单的加法器可以用一种很整洁的、纵列的方式显示数字，这样可以很容易地多次检查你的输入。另外，decimal 模块使用一种基于十进制的计算方式，那也正是我们需要的，我们很多时候并不需要科学家、工程师或计算机所偏爱的浮点数学计算。

当在命令行 shell 下运行此脚本时（此脚本并不是为在 Python 交互式解释器中运行而设

计的），它会给出使用提示，然后一直等待输入。可以敲入一个数字（一位或者多位数字，加上一个可选的小数点，然后是可选的小数位数字），紧跟一个运算符（/、\*、- 和 +，这 4 个字符可以在键盘的数字区找到），然后回车。脚本会根据运算符将输入的数计入到总数中。当你输入空行时，程序将打印出当前的计算结果。当输入 q 并回车时，程序会退出。这种简单的输入输出的界面符合一个典型的简单加法器的需求。

decimal 包是 Python 2.4 标准库的一个组成部分。如果你使用 Python 2.3，请访问 [http://www.taniquetil.com.ar/facundo/bdvfiles/get\\_decimal.html](http://www.taniquetil.com.ar/facundo/bdvfiles/get_decimal.html)，下载并安装这个包。decimal 可支持高精度的十进制数学运算，相比于二进制浮点运算，它的应用面更加广泛（比如涉及财务的计算），当然浮点运算是计算机上速度最快的运算，也是 Python 默认的运算方式。你再也无须花费时间和精力去学习难于理解的二进制浮点运算。如 3.13 节所示，可以将默认的截断规则 ROUND\_HALF\_EVEN 按需求修改成其他规则。

本节的代码非常简单，也有很多可以提升性能的地方。一个有用的加强方法是，将外界输入记录到硬盘以备查用。可以简单地做到这一点——只需要在循环之前加一条语句，打开一个文本文件以供添加信息：

```
tapefile = open('tapefile.txt', 'a')
```

然后，在获得 tape\_line 的值的 try/except 语句之后，加一条语句以便将值写入文件之中：

```
tapefile.write(tape_line+'\n')
```

如果做了上面这些操作，也许你还想继续加强 print\_total 函数，使得它既能够输出信息到命令行窗口，同时还能将信息写入到文件中。因此，该函数可以被修改为：

```
def print_total():
    print '==== =\n', total
    tapefile.write('==== =\n' + str(total) + '\n')
```

file 对象的 write 方法接受一个字符串作为参数，但是并不默认地在字符串末尾加上换行，这和 print 语句的行为方式有些不同，所以我们要明确地调用内建的 str 函数，明确地在末尾增加了一个'\n'。最后，此版本的 print\_total 函数的第二条语句还可以被改写为：

```
print >>tapefile, '==== =\n', total
```

有些人非常讨厌这种 print >>> somefile 的语法，但有时这种写法真的很方便，上面的应用就是一个例子。

还有一些可能的改进，比如现在每次输入完运算符之后都需要按回车键，进一步的改进可以不用按回车就直接进行处理（不过这意味着我们需要处理无缓冲的输入，并且每次要单个处理一个字符，而不能使用方便的基于行的内建函数 raw\_input，可以参考 2.23 节提供的一种跨平台的处理无缓冲输入的方法），再比如增加一个 clear 函数（或者增加提示功能，如果用户输入 0\*会把结果清空成 0），甚至给这个加法器增加一个 GUI 外壳。不过，我准备把这些部分留给读者作为练习。

本节实现中非常重要的一点是 `oper` 字典，它用运算符（/、\*、-和+）作为键，用内建的 `operator` 模块中的数学运算函数作为键的对应值。这个功能也可以用更冗长一点的方法实现，比如，一长串 `if/elif`，如下：

```
if op == '+':
    total = total + decimal.Decimal(num_text)
elif op == '-':
    total = total - decimal.Decimal(num_text)
elif op == '*':
    <line_annotation>... and so on ...</line_annotation>
```

不过，Python 的字典用法明显更理想也更方便，而且不会产生重复性的代码，也更容易维护。

## 更多资料

`decimal` 在 Python 2.4 的 *Library Reference* 中有详细介绍，Python 2.3 的用户也可以通过下载来使用，见 [http://www.taniquetil.com.ar/facundo/bdvfiles/get\\_decimal.html](http://www.taniquetil.com.ar/facundo/bdvfiles/get_decimal.html)；也可以参考十进制 PEP 327，见 <http://www.python.org/peps/pep-0327.html>。

## 3.15 检查信用卡校验和

感谢：David Shaw、Miika Keskinen

### 任务

需要检查一个信用卡号是否遵循工业标准 Luhn 校验和算法。

### 解决方案

Luhn mod 10 是信用卡业检验和的标准。它不是 Python 内建的算法，不过我们可以很容易地实现这个算法：

```
def cardLuhnChecksumIsValid(card_number):
    """ 通过 luhn mod-10 校验和算法检查信用卡号 """
    sum = 0
    num_digits = len(card_number)
    oddeven = num_digits & 1
    for count in range(num_digits):
        digit = int(card_number[count])
        if not ((count & 1) ^ oddeven):
            digit = digit * 2
        if digit > 9:
            digit = digit - 9
        sum = sum + digit
    return (sum % 10) == 0
```

## 讨论

本节代码的原型来自于 Zope 中一个已经不再使用的电子商务程序。

这个程序完成的简单验证工作，可以让你避免提交一个错误的卡号给信用卡提供商，从而节省时间和金钱，因为没人愿意验证一个错误的信用卡号。本节代码的应用面很广，因为很多政府的身份认证号码也使用了 Luhn (modulus 10) 算法。

一个完整的信用卡验证的套件，见 <http://david.theresistance.net/files/creditValidation.py>。

如果喜欢一行代码解决问题，而不是简洁清晰的编码风格，那么我认为，(a)你可能选错了书 (*Perl Cookbook* 是那种会让你满意的类型)，(b)同时，在想换本书之前，瞧瞧下面这种写法能否让你高兴：

```
checksum = lambda a: (10 - sum([int(y)*[7,3,1][x%3] for x, y in enumerate(str(a)[::-1])]))%10)%10
```

## 更多资料

如果你确实喜欢这个单行解决问题的校验和版本，需要找个心理医生。

## 3.16 查看汇率

感谢：Victor Yongwei Yang

### 任务

你想周期性地（用 crontab 或者 Windows 计划任务来运行某 Python 脚本）从 Web 获取数据，监视某两种货币之间的兑换比例，并在两者之间的汇率达到某个阈值时发送提醒邮件。

### 解决方案

这个任务和一系列的从 Web 获取数据的监控任务很类似，它们包括了监视汇率变化、监视股票行情、天气变化等。下面让我们看看，根据加拿大银行的网站提供的报告（一个简单的 CSV（逗号隔开数值），易于解析），怎样实现对美元和加元之间的汇率变化的监视：

```
import httplib
import smtplib
# 在这里配置脚本的参数
thresholdRate = 1.30
smtpServer = 'smtp.freebie.com'
fromaddr = 'foo@bar.com'
toaddrs = 'your@corp.com'
```

```

# 配置结束
url = '/en/financial_markets/csv/exchange_eng.csv'
conn = httplib.HTTPConnection('www.bankofcanada.ca')
conn.request('GET', url)
response = conn.getresponse()
data = response.read()
start = data.index('United States Dollar')
line = data[start:data.index('\n', start)]      # 获得相关行
rate = line.split(',')[-1]                      # 行的最后字段
if float(rate) < thresholdRate:
    # 发送 email
    msg = 'Subject: Bank of Canada exchange rate alert %s' % rate
    server = smtplib.SMTP(smtpServer)
    server.sendmail(fromaddr, toaddrs, msg)
    server.quit()
conn.close()

```

## 讨论

在处理外国货币时，自动完成转换的能力是非常有用的。本节用一种简单而直接的方式实现了这个功能。当 cron 运行这个脚本时，脚本先访问网站，获得 CSV，该文件提供了包括今天的最近 7 天的最新汇率：

```

Date (m/d/year),11/12/2004,11/15/2004, ...,11/19/2004,11/22/2004
$Can/US closing rate,1.1927,1.2005,1.1956,1.1934,1.2058,1.1930,
United States Dollar,1.1925,1.2031,1.1934,1.1924,1.2074,1.1916,1.1844
...

```

然后脚本继续找特定的货币（“United States Dollar”）并读取最后一列的数据，以获得今天的汇率。如果你觉得理解起来还有困难，可以把它一步一步拆解开来，这样可能会更清楚：

```

US = data.find('United States Dollar')          # 寻找货币的索引
endofUSline = data.index('\n', US)                # 找到行末的索引
USline = data[US:endofUSline]                     # 切片获取一个字符串
rate = USline.split(',')[-1]                      # 根据逗号切割并返回最后的字段

```

本节代码还提供了一个重要功能，当汇率触及阈值时，它会自动发送邮件提醒用户。当然这个阈值也可以随意设置（比如可以设置当汇率跳出了预先设置的阈值范围时发送邮件提醒）。

## 更多资料

见 *Library Reference* 和 *Python in a Nutshell* 中的 `http://`、`smtplib` 和字符串函数的相关文档。

# Python 技巧

### 引言

感谢：David Ascher, ActiveState, *Learning Python* 合著者之一

编程语言就像自然语言一样。对于通晓多种语言的人来说，每种语言都有自己独特的一些属性和特征。俄语和法语以奔放著称，而英语则以精确性和活力著称：不像学院派的法语，英语总是为了满足使用上的需求而不停地增加新词，比如“carjacking”、“earwitness”、“snailmail”、“email”、“googlewhacking” 和 “blogging”。在计算机语言的世界中，Perl 以它随心所欲的自由性而闻名：TMTOWTDI(There's More Than One Way To Do It) 是 Perl 程序员对它的最好的赞美。在 Perl 语言和 APL 社区中，简明扼要被认为是一个非常重要的优点。正如你在本书的各个章节的中读到的那样，作为对比，Python 程序员总是不断表达他们对清晰和优雅的推崇。一个非常有名的 Perl 高手曾经跟我说过，Python 确实漂亮得多，但是 Perl 更好玩。我不得不同意他，Python 对美学的追求无处不在（通过良好的设计和定义），而 Perl 则表达了不可救药的幽默感。

我之所以要在引言开头提到这些看上去与主题毫无关联的语言特性，是因为本章的内容将会涉及 Python 的设计美学和语言的活力。如果本书是关于 Perl 的，那么类似于本章这样关于语言窍门的内容，多半会让读者挠头、深思，然后发出一个“啊哈”的赞叹，说不定还会哈哈大笑，因为读者会慢慢领会到各个小把戏后面的天才思想。作为对比，在本章的大部分内容中，作者都在尝试展示语言的优雅特性，因为他认为这样美好的东西还没有引起足够的注意和赞美。就像我一样，我是温哥华的一个骄傲的居民，我会自告奋勇地带着旅游者，向他们展示这个美丽城市的所有的美好的地方，从公园到海滩，再到山峦，而一个 Python 用户则可能会找到他的朋友和同事，兴冲冲地说，“你一定得瞧瞧这个！”对于我和一些我了解的程序员来说，在 Python 中写程序有时候是一种喜悦的分享，而不是一种迫于竞争的追求。学习它的新特性，欣赏它的设计，它的优雅，它的洗练，完全是一种乐趣，但同时教授它的各种细节，讲解它的

微妙用法，其实也是一种极大的快乐。

还得提一提本章的历史：最初在为第 1 版决定各章内容时，我们认为应该有一系列的章节，每章内容都基于不同的目的——比如，蛋奶酥、果馅饼、炖小牛肘。那些章节都自然地分为各个不同的类别，比如甜食、开胃小食、肉食，或者没那么美好的，让人没食欲的文件，算法等。所以，我们选了一系列可能的分类，并在 Zope 的网站上征集内容，之后我完全地打开了思潮的闸门。

结果，很快我们就发现有一些内容无法归入任何预先设置的分类。这是可以解释的，拿烹饪来说吧。本章的内容，可以用制作掺油面糊来做比喻（如果你没有法式烹饪的背景，我要略作解释，这是一种面粉和油脂的混合物，一般用于制作酱汁），捏面团、加面、加入蛋清、在平底锅上翻面、烹煮，这些常见的技巧和知识，任何一个合格的厨师都知道，但是任何一本菜谱上都不会有。这些知识和技巧常常被用来准备菜肴，但却难于归档到任何一种菜肴的制作方法。如果你是一个新手厨师，想尝试某个很棒的菜谱，往往会铩羽而归，那是因为菜谱的作者已经假设你具备那些必要的知识了。作者们只会在类似 *Cooking for Divorced Middle-Aged Man* 的书中解释那些必要的知识（而且通常还配上插图）。但我们并不想把这些宝贵的内容从书中剔除，所以，新的一章就这么诞生了（抱歉，没有配上精美的插图）。

在本书第 1 版的本章的引言中，我是这么说的：

我相信本章中的内容是对时间最敏感的部分。这是因为这些技巧和窍门看上去和最近的语言特性有较高的关联度。

我为我的预言感到骄傲，我的说法完全正确。新版本的内容，完全关注当前的语言特点，最初的一些早期内容已经不再适用了。在本书第 1 版发行之后，Python 已经发布了两个重要的版本，Python 2.3 和 Python 2.4，这门语言的发展也使得初期的一些内容需要适应新特性和新的库函数，语言本身变得更简洁、紧凑和强大，同时有无数新出现的有趣的细节值得我们去发掘。

总之，本章约有一半内容（和本书的比例差不多）是全新的，另外一半则对第一版的内容进行了改编（大多是简化）。由于这些简化工作，以及把本书的重点放到仅有的两个语言版本（2.3 和 2.4）之上，而不再考虑和覆盖第 1 版涉及的很多旧的语言版本，本章比第 1 版的内容多出了超过 1/3，和本书的总的内容增幅差不多。

值得注意的是，很多在本章中经过改编的内容都涉及一些最基本的，没有改变的语言特性：赋值的语义、绑定、复制、引用；序列；字典。所有这些都是进行 Python 编程的最关键的要素，不过我也有点疑惑，不知道 Python 在以后几年的发展中会不会考虑改动这些关键特性。

## 4.1 对象拷贝

感谢：Anna Martelli Ravenscroft、Peter Cogolo

## 任务

你想拷贝某对象。不过，当你对一个对象赋值，将其作为参数传递，或者作为结果返回时，Python 通常会使用指向原对象的引用，并不是真正的拷贝。

## 解决方案

Python 标准库的 `copy` 模块提供了两个函数来创建拷贝。第一个常用的函数叫做 `copy`，它会返回一个具有同样的内容和属性的对象：

```
import copy
new_list = copy.copy(existing_list)
```

某些特殊的时候，你可能会需要对象中的属性和内容被分别地和递归地拷贝，可以使用 `deepcopy`：

```
import copy
new_list_of_dicts = copy.deepcopy(existing_list_of_dicts)
```

## 讨论

当给一个对象赋值（或者将其作为参数传递，或者作为结果返回时）时，Python（像 Java 一样）使用了一个指向原对象的引用，并不是真正的拷贝。其他一些语言则在每次赋值时都进行拷贝操作。Python 从来不为赋值操作进行“隐式”的拷贝：要得到一个拷贝，必须明确地要求，需要的是拷贝。

Python 的行为模式既简单又快速，而且很一致。如果需要拷贝但却不明确地要求，很有可能会遇到麻烦。举个例子：

```
>>> a = [1, 2, 3]
>>> b = a
>>> b.append(5)
>>> print a, b
[1, 2, 3, 5] [1, 2, 3, 5]
```

这里，名字 `a` 和 `b` 都引用到同样的对象（列表 `a`），所以，无论我们通过哪个名字修改了对象的内容，之后，无论通过哪个名字来查看对象，修改结果都是一样的。这个过程中，并没有一个原始的，未被修改的拷贝。

### 警告



要想成为一个好的 Python 程序员，必须了解修改对象和将对象赋值给变量的区别，赋值使用的是引用。这两种操作相互之间并没有什么关联。类似于 `a=[ ]` 这样的语句，是对名字 `a` 做了重新绑定，但却不会影响原先绑定到 `a` 的对象。因此，这里完全没有引用和拷贝的问题：只有当需要修改某些对象的时候，引用和拷贝才有可能成为问题。

如果想修改一个对象，但又需要不改动原对象，必须做一个拷贝。如同前面提到的，Python 标准库的 `copy` 模块提供了两个函数来制作拷贝。一般情况下，可以使用 `copy.copy`，它完成的是对一个对象的浅拷贝——虽然生成了一个新对象，但是对象内部的属性和内容仍然引用原对象，这样的操作速度很快而且节省内存。

如果想从原对象真正地“复制”一个全新的对象，或者想修改的是对象内部的属性和内容，而不是对象本身，那么浅拷贝不能满足你的需求：

```
>>> list_of_lists = [ ['a'], [1, 2], ['z', 23] ]
>>> copy_lol = copy.copy(lists_of_lists)
>>> copy_lol[1].append('boo')
>>> print list_of_lists, copy_lol
[['a'], [1, 2], 'boo'], ['z', 23]] [[['a'], [1, 2], 'boo'], ['z', 23]]
```

这里，名字 `list_of_lists` 和 `copy_lol` 指向了两个不同的对象（两个列表），所以我们可以分别修改它们而不互相影响。然而，`list_of_lists` 中的元素同样也是 `copy_lol` 的对应元素，所以无论通过哪个名字，一旦我们通过索引修改了元素，以后无论通过哪个名字访问其内容，我们会看到修改已经对两者同时生效了。

所以，如果需要拷贝一些容器对象，还必须递归地拷贝其内部引用的对象（包括所有的元素、属性、元素的元素、元素的属性等），使用 `copy.deepcopy` 这种深拷贝操作，会消耗相当的时间和内存，但如果深拷贝确实是需要的效果，你别无选择。而要实现深拷贝，`copy.deepcopy` 是唯一可用的方法。

对于普通的浅拷贝，你可能会需要另外一些方法实现同样的功能，当然，假设你知道想拷贝的对象的类型。对于列表 `L`，调用 `list(L)`；对于字典 `d`，调用 `dict(d)`；为了拷贝集合 `s`（Python 2.4 中已经引入了内建的类型 `set`），调用 `set(s)`。（由于 `list`、`dict` 和 2.4 中的 `set` 已经是内建的名字了，你无须做任何“准备工作”就可以直接使用。）你现在应该能够领会到这种通用的方法了：为了拷贝一个可被拷贝（copyable）的对象 `o`，该对象属于内建的 Python 类型 `t`，可以简单地调用 `t(o)` 来创建拷贝。字典还提供了另一个浅拷贝方法：`d.copy()`，它做的事情和 `dict(d)` 完全一样。不过在这两者之中，我建议你选择 `dict(d)`：这种方式可以和其他类型的拷贝方式统一起来，而且也短一些，少一个字符。

对于任意类型或者类的实例，无论是自己编写的类或者从库中引入的类，一般用 `copy.copy` 就行了。如果是自己编写的类，通常也不值得专门定义一个 `copy` 或者 `clone` 方法，如果想定制自己的类的浅拷贝方式，可以提供一个特殊的 `__copy__` 方法（关于实现这个方法的细节请参看 6.9 节），或者特殊的 `__getstate__` 和 `__setstate__` 方法（参考 7.4 节中关于这些特殊方法的细节，这些方法也有助于深拷贝和序列化——比如对实例进行 `pickling` 操作——的实现）。如果想实现自己的类的独有的深拷贝方式，需要提供一个特殊的 `__deepcopy__` 方法（参考 6.9 节）。

注意，没有必要拷贝那些不可改变的对象（字符串、数字、元组等），因为完全不必担心会不经意改动它们。如果尝试进行拷贝操作，仍然会得到原对象，当然这也不会有太大害处，只不过浪费了一些时间和代码。举个例子：

```
>>> s = 'cat'  
>>> t = copy.copy(s)  
>>> s is t  
True
```

`is` 操作符检查两个对象是否相同，而不是相等（`is` 检查对象是否相同；`==` 操作符则检查两个对象是否相等）。对于不可改变的对象来说，检查是否相同几乎没有用处（这里我们只是为了展示对不可改变对象调用 `copy.copy` 是没意义的，但也是无害的）。对于可改变的对象，检查相同性有时却是至关重要的。举个例子，假设你不确定两个名字 `a` 和 `b` 是分别指向不同的对象还是引用同一个对象，可以用 `a is b` 这样一条简单快速的检查语句来找到答案。当需要确保原对象不被改变时，就可以考虑对原对象进行拷贝操作了。

### 警告

可以使用其他的方法来创建拷贝。给定一个列表 `L`，使用“整个对象的切片”`L[:]`以及列表推导`[x for x in L]`都可以完成对 `L` 的浅拷贝，而使用添加空列表，`L+[]`，以及用 `L` 乘以 1，`L*1`，诸如此类的方式都只是无谓的浪费时间和内存，直接调用 `list(L)` 速度更快也更清晰。不过，由于某些历史原因以及应用的广泛性，你也应该熟悉 `L[:]` 这种用法。所以，即使我不建议你在自己的代码中这么写，但你也可能在别人的代码中看到这种用法。



类似的，给定一个字典 `d`，也可以通过循环来创建一个浅拷贝 `d1`：

```
>>> d1 = {}  
>>> for somekey in d:  
...     d1[somekey] = d[somekey]
```

或者更简明的方式：`d1 = {};` `d1.update(d)`。不过，这仍然是一种时间和代码上的浪费，除了增加一些迷惑性和降低速度，没有更多的东西。用 `d1=dict(d)` 这样一句话就可以了，又快又轻便。

### 更多资料

*Library Reference* 和 *Python in a Nutshell* 中关于 `copy` 模块的内容。

## 4.2 通过列表推导构建列表

感谢：Luther Blisett

### 任务

你想通过操作和处理一个序列（或其他的可迭代对象）中的元素来创建一个新的列表。

### 解决方案

假设你想通过给某个列表中的每个元素都加上 23 来构建一个新列表。用列表推导可以很直接地实现这个想法：

```
thenewlist = [x + 23 for x in theoldlist]
```

同样，假设需要用某列表中的所有大于 5 的元素来构成一个新列表。用列表推导代码可以写成这样：

```
thenewlist = [x for x in theoldlist if x > 5]
```

当你试图将这两种想法合二为一的时候，可以增加一个 if 子句，并对选定的项使用某些表达式，比如加上 23，这些都可以用一行概括成：

```
thenewlist = [x + 23 for x in theoldlist if x > 5]
```

### 讨论

优雅、清晰和务实，都是 Python 的核心价值观，列表推导说明了这三点是怎样和谐地统一起来的。事实上，当你直觉地考虑“改变某列表”而不是新建某列表时，列表推导常常是最好的办法。比如，假如需要将某列表 L 中的所有大于 100 的元素设置成 100，最好的方法是：

```
L[:] = [min(x, 100) for x in L]
```

上面代码在给一个“整个列表的切片”赋值的同时，修改了该列表对位的数据，而不是试图对名字 L 重新绑定，比如写成 L = ...。

当你只是需要执行一个循环的时候，不应该使用列表推导。如果需要循环，那就写相应的循环代码。关于循环列表的例子，参看 4.4 节。第 19 章还会有更多的有关 Python 的迭代的内容。

另外，如果 Python 有内建的操作或者类型能够以更直接的方式实现你的需求，你也不要使用列表推导。比如，为了复制一个列表，用 L1 = list(L)就好，不要这么用：

```
L1 = [x for x in L]
```

类似地，如果想对每个元素都调用一个函数，并使用函数的返回结果，应该用 `L1=map(f, L)`，而不是 `L1 = [f(x) for x in L]`。不过大多数情况下，这种列表推导用法也没问题。

在 Python 2.4 中，当序列过长，而你每次只需要获取一个元素的时候，应当考虑使用生成器表达式，而不是列表推导。生成器表达式的语法和列表推导一样，只不过生成器表达式是被()圆括号括起的，而不是方括号[]。比如，如果我们需要计算本节解决方案里的列表的某些元素之和，在 Python 2.3 中可以这么做：

```
total = sum([x + 23 for x in theoldlist if x > 5])
```

在 Python 2.4 中，代码可以写得更自然一点，方括号可以省略掉（也不用增加多余的圆括号——有调用内建的 `sum` 函数的圆括号就足够了）：

```
total = sum(x + 23 for x in theoldlist if x > 5)
```

除了更加简洁，这个方法还可以避免一次性将整个列表载入内存，从而在列表特别长的时候，在一定程度上提高处理速度。

## 更多资料

参考 *Reference Manual* 中 `list display`（列表推导的另一个名字）和 Python 2.4 中生成器表达式的内容；第 19 章；*Library Reference* 和 *Python in a Nutshell* 中 `itertools` 模块和内建的 `map`、`filter` 和 `sum` 函数；以及 Haskell，见 <http://www.haskell.org>。

### 警告

Python 从函数式语言 Haskell (<http://www.haskell.org>) 借来了列表推导，当然在语法和关键字上有一些差异，而不是完全一样。如果你懂 Haskell，那么小心点，因为 Haskell 的列表推导正如这门语言的其余部分一样，使用惰性求值 (lazy evaluation) 的方式（也被称为正则序 (normal order) 或按需调用 (call by need))。每一项都只在真正需要的时候才计算。Python，像很多语言一样，使用（包括列表推导和其他部分）主动求值 (eager evaluation) 方式（也被称为应用序 (application order)，按值调用 (call by value)，或者严格求值 (strict evaluation)）。也就是说，整个列表在列表推导执行的时候就完成了计算，而且结果被尽量长久地保存在内存中以供以后使用。如果你正在将一个 Haskell 程序改写成 Python 程序，而且这个 Haskell 程序用列表推导来表示无限序列或任何一个很长的序列。那么 Python 的列表推导可能不是很合适。不过 Python 2.4 中出现了生成器表达式，它的语义更加接近 Haskell 的惰性求值方式，同样是按需调用，所以它应该是更好的选择。



## 4.3 若列表中某元素存在则返回之

感谢：Nestor Nissen、A. Bass

### 任务

你有一个列表 L，还有一个索引号 i，你希望当 i 是 L 的有效索引时获取 L[i]，若不是有效索引，则返回一个默认值 v。如果 L 是字典，可以使用 L.get(i, v) 来满足需求，可是列表并没有 get 这个方法。

### 解决方案

很明显，我们得自己写个函数，在这里，最简单直接的方法就是最好的方法：

```
def list_get(L, i, v=None):
    if -len(L) <= i < len(L): return L[i]
    else: return v
```

### 讨论

解决方案中的函数根据 Python 的索引规则来检查 i 的有效性：有效索引只能在大于等于 -len(L) 和小于 len(L) 这个区间中。但如果所有传递给 list\_get 函数的参数 i 都是有效的索引，你可能会喜欢另外一种方式：

```
def list_get_egfp(L, i, v=None):
    try: return L[i]
    except IndexError: return v
```

但是，除非传递给此函数的索引绝大多数都是有效索引，否则这个函数（通过某些测试工具测量）将会比解决方案中的 list\_get 函数慢 4 倍。因此，这个“获得原谅总是比获得许可容易（easier to get forgiveness than permission, EGFP）”的函数，虽然更具有 Python 的精神和风格，但在这种特殊的情况下，并不值得推荐。

我还试过几个看上去更漂亮、更复杂和更迷惑人的方法，不过，除了更加难于解释和理解之外，它们无一例外地比那个朴实无华的 list\_get 函数慢。这里给出一个通用的准则：当你写 Python 程序时，应当倾向于清晰和可读性，而不是紧凑和精炼——选择简单，而不是精巧。只要你坚持这么做，你常常会发现你的代码跑得更快，而且也更强健，更易于维护。在真实世界中，对于 99.9% 的应用而言，遵循这个原则要比获得一点速度提升重要的多。

### 更多资料

*Language Reference* 和 *Python in a Nutshell* 中关于列表索引的文档。

## 4.4 循环访问序列中的元素和索引

感谢: Alex Martelli、Sami Hangaslammi

### 任务

需要循环访问一个序列，并且每一步都需要知道已经访问到的索引（因为需要重新绑定序列的入口），但 Python 提供的首选的循环方式完全不用依赖索引。

### 解决方案

内建函数 `enumerate` 正是为此而生。看例子：

```
for index, item in enumerate(sequence):
    if item > 23:
        sequence[index] = transform(item)
```

它看上去很干净易读，而且比那种通过索引访问元素的方式快：

```
for index in range(len(sequence)):
    if sequence[index] > 23:
        sequence[index] = transform(sequence[index])
```

### 讨论

循环遍历一个序列是很常见的需求，Python 强烈建议你用一种最直接的方式。事实上这也是最具有 Python 风格的访问序列中每个元素的方式：

```
for item in sequence:
    process(item).
```

而其他一些典型的比较底层的语言，不是用这种直接的循环方式，而是通过序列的索引，根据索引找到每一个对应的子项：

```
for index in range(len(sequence)):
    process(sequence[index])
```

直接的循环方式更加干净、更易读、更快，而且也更通用（因为根据定义，此法可以应用于任何可迭代对象，而根据索引访问的方式则只适用于序列，如列表）。

但是，有时候在循环中，你的确需要同时获得索引和索引对应的子项。一个常见的理由是，你想重新绑定列表的新入口，必须将 `thelist[index]` 赋值为一个新的子项。为了支持这种需求，Python 提供了内建函数 `enumerate`，它接受任何可迭代的参数，并返回一个迭代器，迭代器产生的是一个（两个子项的元组）形如(`index, item`)的结果，一次一项。因此你的 `for` 子句的头部可以写成：

```
for index, item in enumerate(sequence):
```

这样，在 for 的主体中，索引和子项都是可以访问的。

为了帮助你记忆 enumerate 产生的结果，考虑惯用法 `d=dict(enumerate(L))`。实际上从某种意义上讲，用此法获得的字典 `d` 是等价于列表 `L` 的，因为对于任意一个有效的非负索引 `i`, `d[i] is L[i]` 都成立。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中 `enumerate` 的相关内容；第 19 章。

## 4.5 在无须共享引用的条件下创建列表的列表

感谢：David Ascher

### 任务

你想创建一个多维度的列表，且同时避免隐式的引用共享。

### 解决方案

为了创建列表且避免隐式地共享引用，我们可使用列表推导。举个例子，创建一个  $5 \times 10$  的全为 0 的阵列：

```
multilist = [[0 for col in range(5)] for row in range(10)]
```

### 讨论

当 Python 新手首次发现列表乘以一个整数得到的列表是原列表的多次重复时，他会感到非常兴奋，因为看上去这种处理极其优雅。举个例子：

```
>>> alist = [0] * 5
```

很明显，这是很棒的一种获得  $1 \times 5$  维度的全为 0 的数组的方法。

问题是，一维的任务常常会根据需要扩展成二维的，所以，一个自然的想法是进行如下的处理：

```
>>> multi = [[0] * 5] * 3
>>> print multi
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

看上去它工作完全正常，但是新手们常常会发现自己被 bug 困扰。举个简单的例子，下面的代码片段很有说服力：

```
>>> multi[0][0] = 'oops!'
>>> print multi
[['oops!', 0, 0, 0, 0], ['oops!', 0, 0, 0, 0], ['oops!', 0, 0, 0, 0]]
```

绝大多数程序员都至少在这个问题上栽过一次跟头（见位于 <http://www.python.org/doc/FAQ.html#4.50> 的 FAQ 条目）。为了理解这个问题，可以将多维列表的创建分解成两个步骤：

```
>>> row = [0] * 5          # row 列表中的 5 个子项都引用 0  
>>> multi = [row] * 3      #multi 列表中的 3 个子项都引用 row
```

分解过后的代码片段生成的 multi 列表完全等价于前面的更简洁的  $[[0]*5]*3$  代码片段产生的结果，而且问题也完全一样：如果你给  $multi[0][0]$  指定一个新值，会改变  $multi[1][0]$  的值， $multi[2][0]$  的值…，甚至改变了  $row[0]$  的值！

代码中的注释是理解这个混淆的关键之处。将序列和数字相乘产生的新序列，将含有多个引用原始内容的子项，其数量等于乘数。在  $row$  的创建中，有无引用被复制完全不重要，因为被引用的是数字，而数字是不可改变的。换句话说，如果对象是不可改变的，则对象和对对象的引用实际上没什么区别。第二行，我们创建了一个新的列表，其中包含了 3 个对  $[row]$  的内容的引用，而其内容则是对一个列表的引用。因此， $multi$  包含了 3 个对列表的引用。这样，当  $multi$  的第一个子项的第一个子项被修改的时候，你会发现共享的列表的第一个子项被修改了。奇迹就是这么发生的。

而解决方案中所示的列表推导则避免了问题。使用列表推导，没有任何引用共享的问题——你事实上完成了一个真正的嵌套计算。如果确实领会了上面的讨论，你可能会意识到可以不用内层的列表推导，只需要外层的。换句话说，代码这样改一下是不是也可以？

```
multilist = [[0]*5 for row in range(10)]
```

答案是，可以，事实上在最内层使用列表乘法，而在其他层次使用列表推导，速度会快不少——要比示例快两倍。那么我为什么不推荐最后讨论这种方法呢？答案是：对于这个例子，在 Python 2.3 中，速度的提升是从  $57\mu\text{s}$  降低到了  $24\mu\text{s}$ ，在 Python 2.4 中，从  $49\mu\text{s}$  降低到了  $21\mu\text{s}$ ，运行平台是一台又老又破的个人计算机（AMD Athlon 1.2 GHz CPU，运行 Linux）。优化一个列表的创建操作，从而节省几十  $\mu\text{s}$  并不会对你的应用程序的性能产生什么大的影响：如果你真的需要优化，那就应该去找真正重要的地方，真正对应用程序的整体性能产生巨大影响的地方。所以，我更倾向于解决方案中给出的代码，它更加简单，因为它从内层到外层的列表创建都使用了相同的结构，看上去也更具有对称性，当然也更易读。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中的内建函数 `range` 的文档。

## 4.6 展开一个嵌套的序列

感谢：Luther Blissett、Holger Krekel、Hemanth Sethuram、ParzAspen Aspen

## 任务

序列中的子项可能是序列，子序列的子项仍可能是序列，以此类推，则序列嵌套可以达到任意的深度。需要循环遍历一个序列，将其中所有的子序列展开成一个单一的、只具有基本子项的序列。（一个基本子项或者原子，可以是任何非序列的对象——或者说叶子，假如你认为嵌套序列是一棵树。）

## 解决方案

我们需要能够判断哪些我们正在处理的子项是需要被展开的，哪些是原子。为了获得通用性，我们使用了一个断定来作为参数，由它来判断子项是否是可以展开的。（断定 [predicate] 是一个函数，每当我们处理一个元素时就将其应用于该元素并返回一个布尔值：在这里，如果元素是一个需要展开的子序列就返回 True，否则返回 False。）我们假定每一个列表或者元组都是需要被展开的，而其他类型则不是。那么最简单的解决方法就是提供一个递归的生成器：

```
def list_or_tuple(x):
    return isinstance(x, (list, tuple))
def flatten(sequence, to_expand=list_or_tuple):
    for item in sequence:
        if to_expand(item):
            for subitem in flatten(item, to_expand):
                yield subitem
        else:
            yield item
```

## 讨论

展开一个嵌套的序列，或者等价地，按照顺序“遍历”一棵树的所有叶子，是在各种应用中很常见的任务。如果有一个嵌套的结构，元素都被组织成序列或者子序列，而且，基于某些理由，你并不关心结构本身，需要的只是一个接一个地处理所有的元素。举个例子：

```
for x in flatten([1, 2, [3, [ ], 4, [5, 6], 7, [8, ], ], 9]):
    print x,
输出 1 2 3 4 5 6 7 8 9.
```

这个任务唯一的问题是，怎样在尽量通用的尺度下，判断什么是需要展开的，什么是需要被当做原子的，这其实没有看上去那么容易。所以，我绕开直接的判断，把这个工作交给了一个可调用的断定参数，调用者可以将其传递给 flatten，如果调用者满足于 flatten 简单的默认行为方式，即只展开元组和列表。

在 flatten 所在的模块中，我们还需要提供另一个调用者可能需要用到的断定——它将

展开任何非字符串（无论是普通字符串还是 Unicode）的可迭代对象。字符串是可迭代的，但是绝大多数应用程序还是想把它们当成原子，而不是子序列。

至于判断对象是否可迭代，我们只需要对该对象调用内建的 `iter` 函数：若该对象是不可迭代的，此函数将抛出 `TypeError` 异常。为了判断对象是否是类字符串的，我们则简单地检查它是否是 `basestring` 的实例，当 `obj` 是 `basestring` 的任何子类的实例时，`isinstance(obj, basestring)` 的返回值将是 `True`——这意味着任何类字符串类型。因此，这样的一个断定并不难写：

```
def nonstring_iterable(obj):
    try: iter(obj)
    except TypeError: return False
    else: return not isinstance(obj, basestring)
```

当具体的需求是展开任何可迭代的非字符串对象时，调用者可以选择调用 `flatten(seq, nonstring_iterable)`。无疑，不把 `nonstring_iterable` 断定作为 `flatten` 的默认选项是一个更好的选择：在简单的需求中，如我们前面展示的示例代码片段，使用 `nonstring_iterable` 会比使用 `list_or_tuple` 慢 3 倍以上。

我们也可以写一个非递归版本的 `flatten`。这种写法可以超越 Python 的递归层次的极限，一般不超过几千层。实现无递归遍历的要点是，采用一个明确的后进先出（LIFO）栈。在这个例子中，我们可以用迭代器的列表实现：

```
def flatten(sequence, to_expand=list_or_tuple):
    iterators = [ iter(sequence) ]
    while iterators:
        # 循环当前的最深嵌套（最后）的迭代器
        for item in iterators[-1]:
            if to_expand(item):
                # 找到了子序列，循环子序列的迭代器
                iterators.append(iter(item))
                break
            else:
                yield item
        else:
            # 最深嵌套的迭代器耗尽，回过头来循环它的父迭代器
            iterators.pop()
```

其中 `if` 语句块的 `if` 子句会展开每一个我们需要展开的元素——即我们需要循环遍历的子序列；所以在该子句中，我们将那个子序列的迭代器压入栈的末尾，再通过 `break` 打断 `for` 的执行，回到外层的 `while`，外层 `while` 会针对我们刚刚压入的新的迭代器执行一个新的 `for` 语句。`else` 子句则用于处理那些不需要展开的元素，它直接产生元素本身。

如果 for 循环未被打断，for 语句块所属的 else 子句将得以执行——换句话说，当 for 循环完全执行完毕，说明它已经遍历完当前的最新的迭代器。所以，在 else 子句中，我们移除了已经耗尽的嵌套最深（最近）的迭代器，之后外层的 while 循环继续执行，如果栈已经空了，则中止循环，如果栈中还有迭代器，则执行一个新的 for 循环来处理之——正好是上次执行中断的地方，本质上，迭代器的任务就是记忆迭代的状态。

flatten 的非递归实现产生的结果和前面的简单一些的递归版本的结果完全一致。如果你认为非递归实现会比递归方式快，那么你可能会失望：我采用了一系列的测试用例进行观察测量，发现非递归版本要比递归版本慢约 10%。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中关于序列类型和内建的 `iter`、`isinstance` 以及 `basestring` 的文档。

## 4.7 在行列表中完成对列的删除和排序

感谢： Jason Whitlark

### 任务

你有一个包含列表（行）的列表，现在你想获得另一个列表，该列表包含了同样的行，但是其中一些列被删除或者重新排序了。

### 解决方案

列表推导很适合这个任务。假设你有：

```
listOfRows = [ [1,2,3,4], [5,6,7,8], [9,10,11,12] ]
```

需要另一个有同样行的列表，但是其中第二列被删除了，第三和第四列则互换了位置。我们用一个简单的列表推导来完成任务：

```
newList = [ [row[0], row[3], row[2]] for row in listOfRows ]
```

还有一个方法同样具有可操作性，也许还更优雅一些，即采用一个辅助的序列（列表或元组），此序列的列索引位置正好是需要的顺序。这样，在外层对 `listOfRows` 进行循环操作的列表推导的内部，又加入了一个嵌套的对辅助序列进行循环操作的内层列表推导。

```
newList = [ [row[ci] for ci in (0, 3, 2)] for row in listOfRows ]
```

## 讨论

我一般用列表的列表来代表二维的阵列。我把这些列表看做以二维阵列的“行”为元素的列表。我常常需要处理这种二维阵列的列，一般是重新排序，有时还会忽略列中的部分内容。尽管列表推导在别处大显神通，但粗略一看，它在这里似乎用处不大（至少我的第一感觉是这样）。

列表推导会产生一个新的列表，而不是修改现有的列表。当需要修改一个现有的列表时，最好的办法是将现有列表的内容赋值为一个列表推导。举个例子，假设要修改 `listOfRows`，对于本节的任务，可以写成：

```
listOfRows[:] = [row[0], row[3], row[2]] for row in listOfRows ]
```

正如本节解决方案的第二个例子所示，还可以考虑使用一个辅助的序列，其中包含的列索引按需要的顺序排列，这比第一个例子用硬编码指定顺序要好。你可能对嵌套的列表推导感到不放心，但事实上它比你想象的更简单安全。如果采用解决方案中的第二种方式，会同时获得更多的通用性，因为可以给辅助序列一个名字，并使用这个名字来对一些行列表进行重新排序，或者将其作为参数传递给一个函数，等等：

```
def pick_and_reorder_columns(listofRows, column_indexes):
    return [row[ci] for ci in column_indexes] for row in listofRows ]
columns = 0, 3, 2
newListOfPandas = pick_and_reorder_columns(oldListOfPandas, columns)
newListOfCats = pick_and_reorder_columns(oldListOfCats, columns)
```

上例所做的事情和本节前面的代码片段做的事情完全一样，不过它操作两个独立的“旧的”列表，并分别获得两个对应的“新的”列表。追求最大程度的通用性是一种难以抵御的诱惑，但在这里，这个 `pick_and_reorder_columns` 所表现出的通用性似乎恰到好处。

最后一条，在前面用到的一些函数中，一些人喜欢用看上去更漂亮的方式来表示“内层”列表推导，而不采用简单直接的方式，比如：

```
[row[ci] for ci in column_indexes]
```

他们喜欢用内置的 `map` 函数以及 `row` 的特殊的 `__getitem__` 方法（常被用作被绑定方法）来完成索引子任务，因此他们这样编写代码：

```
map(row.__getitem__, column_indexes)
```

具体到某些不同的 Python 的版本，这种看上去漂亮但又有点让人困扰的方法可能速度会快一些。但我仍然认为体现简洁性的列表推导是最佳方式。

## 更多资料

*Language Reference* 和 *Python in a Nutshell* 中的列表推导的文档。

## 4.8 二维阵列变换

感谢: Steve Holden、Raymond Hettinger、Attila Vásárhelyi、Chris Perkins

### 任务

需要变换一个列表的列表，将行换成列，列换成行。

### 解决方案

需要一个列表，其中的每一项都是同样长度的列表，像这样：

```
arr = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
```

列表推导提供了简单方便的方法以完成二维阵列的转换：

```
print [[r[col] for r in arr] for col in range(len(arr[0]))]  
[[1, 4, 7, 10], [2, 5, 8, 11], [3, 6, 9, 12]]
```

另一个更快也更让人困惑的方法（输出是一样的）是利用内建函数 `zip` 实现的：

```
print map(list, zip(*arr))
```

### 讨论

本节展示了一种简洁而清晰的转换方式，还有一个更快速的备选方案。在需要简洁和清晰并存的时候，列表推导通常是很好的选择，而备选方案利用内建函数 `zip` 以另外一种方式达到目的，显得很晦涩难懂。

有时，你获得的数据的顺序是不正确的。举个例子，如果使用微软的 ActiveX Data Objects (ADO) 数据库接口，由于 Python 和微软的首选实现语言 (Visual Basic) 在对数组元素排序上的差异，`Getrows` 方法返回的实际上是 Python 中的列。本节针对这种常见需求提出的两种解决方案，让你有机会在清晰和速度之间进行选择。

在列表推导的解决方案中，内层推导改变的是（从行中）选出的元素，外层推导则影响选择子 (selector，即列)。由此实现转换。

而基于 `zip` 的解决方案，我们使用了`*a` 语法将 `arr` 中的每个元素（行），根据顺序，作为分隔开的参数传递给 `zip`。`zip` 返回的是元组的列表，其实已经完成了转换。通过 `map` 调用，我们可以对每个元组调用 `list`，以获得一个列表的列表。既然我们不能将 `zip` 的结果直接当做列表使用，我们可以通过使用 `itertools.izip` 来得到一点改进（因为 `izip` 并不会将结果当做列表载入内存，而是每次生成一个子项）：

```
import itertools  
print map(list, itertools.izip(*arr))
```

不过，对这个例子而言，这一点速度提升也许并不能抵消它所带来的复杂性。

### \*args 和 \*\*kwds 语法

`*args` (\*通常紧跟一个标识符，你会看到 `a` 或者 `args` 都是标识符) 是 Python 用于接收或者传递任意基于位置的参数的语法。当你接收到一个用这种语法描述的参数时（比如你在函数的 `def` 语句中对函数签名使用了星号语法），Python 会将此标识符绑定到一个元组，该元组包含了所有基于位置的隐式地接收到的参数。当你用这种语法传递参数时，标识符可以被绑定到任何可迭代对象（事实上，它也可以是任何表达式，并不必须是一个标识符，只要这个表达式的结果是一个可迭代对象即可）。

`**kwds` (标识符可以是任意的，通常用 `k` 或者 `kwds` 表示) 是 Python 用于接收或者传递任意命名的参数的语法。（Python 有时候会将命名参数称为关键字参数，它们其实并不是关键字——只是用它们来给关键字命名，比如 `pass`、`for` 或 `yield`，还有很多。不幸的是，这种让人疑惑的术语目前仍是这门语言及其文化的根深蒂固的一个组成部分。）当你接收到用这种语法描述的一个参数时（比如你在函数的 `def` 语句中对函数签名使用了双星号语法），Python 会将标识符绑定到一个字典，该字典包含了所有接收到的隐式的命名参数。当你用这种语法传递参数时，标识符只能被绑定到字典（其实它也可以是表达式，不一定是一个标识符，只要这个表达式的结果是一个字典即可）。

当你在定义或调用一个函数的时候，必须确保 `*a` 和 `**k` 在其他所有参数之后。如果这两者同时出现，要将 `**k` 放在 `*a` 之后。

如果要转换非常巨大的数字阵列，可以考虑 Numeric Python 和其他的第三方包。Numeric Python 支持一系列变换以及轴旋转，这些数学转换能把大多数人绕晕。

## 更多资料

Reference Manual 和 *Python in a Nutshell* 中 `list displays` (列表推导的另一种叫法) 以及对于基于位置的参数 `*a` 和命名参数 `**k` 的传递；内建函数 `zip` 和 `map`；Numeric Python (<http://www.pfdubois.com/numpy/>)。

## 4.9 从字典中取值

感谢： Andy McKay

### 任务

你想从字典中取值，但是又不想由于你搜寻的键不存在而处理异常。

## 解决方案

字典的 `get` 方法正是为取值而准备的。假设你有一个字典 `d = {'key': 'value', }`。为了得到 `key` 在 `d` 中对应的值，且不希望担心异常的问题，可以这样编写代码：

```
print d.get('key', 'not found')
```

如果想在取值之后将该条目删去，用 `d.pop`（执行 `get` 和 `remove` 操作）替换 `d.get`（只读取 `d`，从不修改 `d` 的值）即可。

## 讨论

为了在键不存在的时候取值且并不引发异常，用字典的简单的 `get` 方法即可。

如果试图通过索引的方式取值，比如 `d[x]`，而且 `x` 并不是字典 `d` 的键，你的举动会引发 `KeyError` 异常。这通常也没什么问题。如果期望获取字典中 `x` 对应的值，异常是最好的提醒你所犯的错误的方式（比如，可能需要调试你的程序）。

然而，有时候只是想尝试一下，因为你已经知道，`x` 可能不是 `d` 的键。这种情况下，不用引入 `in` 测试，如下：

```
if 'key' in d:  
    print d['key']  
else:  
    *args 和 **kwds 语法 print 'not found'
```

或者使用 `try/except` 语句，如下：

```
try:  
    print d['key']  
except KeyError:  
    print 'not found'
```

而应该使用 `get` 方法，就像“解决方案”所示的那样。如果调用 `d.get(x)`，不会有任何异常抛出：如果 `x` 是字典 `d` 中的键，你会得到 `d[x]`，如果不是，你只能得到 `None`（可以检查或者继续传递）。当 `x` 不是 `d` 的键的时候，如果 `None` 不是你期望的值，还可以调用 `d.get(x, somethingelse)`。这样，如果 `x` 不是 `d` 的键，得到的值是 `somethingelse`。

`get` 是一种简单而有用的机制，Python 的文档对此有很好的解释，奇怪的是有相当多的人并不清楚这一点。另一个类似的方法是 `pop`，与 `get` 很类似，只不过当键在字典中时，`pop` 会同时删除该条目。还有一条附加说明：`get` 和 `pop` 并不完全对应。如果 `x` 不是 `d` 的键，`d.pop(x)` 会抛出 `KeyError` 异常；如果要想获得和 `d.get(x)` 同样的效果，同时还具有删除条目的能力，调用 `d.pop(x, None)` 即可。

## 更多资料

4.10 节，*Library Reference* 和 *Python in a Nutshell* 中关于映射类型的文档。

## 4.10 给字典增加一个条目

感谢：Alex Martelli、Martin Miller、Matthew Shomphue

### 任务

给定一个字典 `d`, 当 `k` 是字典的键时, 你想直接使用 `d[k]`, 若 `k` 不是 `d` 的键, 则这个操作会给字典增加一个新条目 `d[k]`。

### 解决方案

字典的 `setdefault` 方法正是为此而设计的。假设我们正在创建一个由单词到页数的映射, 字典将把每个单词映射到这个词出现过的页的页码构成的列表。这个应用中关键的代码段可能是这样的:

```
def addword(theIndex, word,pagenumber):
    theIndex.setdefault(word, [ ]).append(pagenumber)
```

这段代码等价于下面的更加详尽的版本:

```
def addword(theIndex, word,pagenumber):
    if word in theIndex:
        theIndex[word].append(pagenumber)
    else:
        theIndex[word] = [pagenumber]
```

以及:

```
def addword(theIndex, word,pagenumber):
    try:
        theIndex[word].append(pagenumber)
    except KeyError:
        theIndex[word] = [pagenumber]
```

使用 `setdefault` 方法能在相当大的程度上简化实现。

### 讨论

对于一个字典 `d`, `d.setdefault(k, v)` 非常接近于 `d.get(k, v)`, 后者在前面的 4.9 节曾介绍过。最本质的区别是, 如果 `k` 不是字典的键, `setdefault` 方法会将 `d[k]` 赋值为 `v`, 即 `d[k]=v` (`get` 则仅仅返回 `v`, 对 `d` 不会有任何影响)。因此, 当需要类似于 `get` 的功能, 但又需要同时提供这种特殊的效果时, 请考虑 `setdefault` 方法。

如果字典中的值是列表, `setdefault` 方法尤其有用, 4.15 节会提供更多的细节。最经典的 `setdefault` 应用大概会是这样:

```
somedict.setdefault(somekey, [ ]).append(somevalue)
```

对于不可改变的值，`setdefault` 方法就没那么有用了。如果想对单词计数，最好的方法是使用 `get`，而不是 `setdefault`：

```
theIndex[word] = theIndex.get(word, 0) + 1
```

这是因为反正你也必须要将字典的 `theIndex[word]` 的条目重新绑定（因为数字是不可改变的）。不过，针对我们这个单词到页码的例子，你一定不想由于下面这样的写法而造成性能上的损失：

```
def addword(theIndex, word, pagenumber):
    theIndex[word] = theIndex.get(word, []) + [pagenumber]
```

让我们看看最后这个版本的 `addword`，每次调用它的时候都将创建 3 个新的列表：一个被作为第二个参数传递给 `theIndex.get` 的空列表，一个只含有一个元素 `pagenumber` 的列表，以及将前面两者拼接而得到的有  $N+1$  个元素的列表 ( $N$  是 `word` 之前出现的次数)。创建这么多列表很显然会降低整体性能。举个例子，在我的计算机上，我对出现在 1 000 页中的 4 个单词所进行的索引操作计时。用 `addword` 的第一个版本来作为参考，第二个版本（使用 `try/except`）快了约 10%，第三个版本（使用 `setdefault`）则慢了约 20%——这种性能差异基本上无关紧要。但第四个版本（使用 `get`）慢了 4 倍，这样的性能损失恐怕就不能视若无睹了。

## 更多资料

4.9 节；4.15 节；*Library Reference* 和 *Python in a Nutshell* 中字典的相关文档。

## 4.11 在无须过多援引的情况下创建字典

感谢：Brent Burley、Peter Cogolo

### 任务

当你逐渐习惯了 Python，会发现自己已经创建了大量的字典。当键是标识符时，可以用 `dict` 加上命名的参数来避免援引它们：

```
data = dict(red=1, green=2, blue=3)
```

这看上去比直接用字典形式的语法要整洁一些：

```
data = {'red': 1, 'green': 2, 'blue': 3}
```

### 讨论

一种创建字典的好方法是调用内建的 `dict` 类型，但有时用带有花括号和冒号的字典形式也不错。本节的代码说明，如果键都是文字的字符串并且在语法上对用户而言也是有效且适合作为标识符的，则通过调用 `dict`，无须援引字典的键。不能对像 “12ba” 和

“for”一样的字符串应用此法，因为“12ba”以数字开头，而 for 正好是 Python 关键字，不能作为标识符。

字典形式的语法是 Python 中唯一需要用到花括号的地方：如果不喜欢单引号，或者正好所用的键盘不太容易打出花括号（所有的意大利布局的键盘都这样），那么可以用 dict() 而不是 {} 来创建一个空字典。

调用 dict 还能给你带来一些其他额外的好处。dict(d) 返回一个完全独立于原字典 d 的拷贝，就像 d.copy() 一样。但当 d 不是一个字典，而是一个数对(key, value)的序列时，dict(d) 仍然有效（如果 key 在序列中出现多次，那么只有最后一次出现的 key 会被计入）。一般创建字典的操作大概是这样：

```
d = dict(zip(the_keys, the_values))
```

the\_key 是键的序列，the\_values 则是键的对应值的序列。内建的 zip 函数创建并返回一个数对(key, value)构成的列表，内建类型 dict 接受这个列表作为参数并创建相应的字典。如果序列非常长，那么用 Python 标准库中的 itertools 模块能有效地提高速度：

```
import itertools
d = dict(itertools.izip(the_keys, the_values))
```

内建函数 zip 会在内存中创建出包含所有数对的列表，而 itertools.izip 一次只生成一对数据。在我的计算机上，对于长度为 10 000 的序列，后面这种方式可以快两倍左右——Python 2.3 中是 18ms 对 45ms，Python 2.4 中则是 17ms 对 32ms。

还可以在 dict 调用中使用基于位置的参数和命名参数（如果命名的参数正好和基于位置的参数冲突，则前者生效）。举个例子，下面是一个字典的创建，其中用到了前面提到的 Python 关键字和另一个不宜做命名参数的键名：

```
d = dict({'12ba':49, 'for': 23}, rof=41, fro=97, orf=42)
```

如果想创建一个字典，其中每个键都对应相同的值，只需调用 dict.fromkeys(key\_sequence, value)（如果你忽略了 value，它默认使用 None）即可。下面给个例子，用很清爽的方式初始化一个字典，并用它来统计不同的小写 ASCII 字母的出现次数：

```
import string
count_by_letter = dict.fromkeys(string.ascii_lowercase, 0)
```

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中内建的 dict 和 zip，以及模块 itertools 和 string 的文档。

## 4.12 将列表元素交替地作为键和值来创建字典

感谢：Richard Philips、Raymond Hettinger

## 任务

给定一个列表，需要交替地使用列表中的元素作为键和对应值来创建一个字典。

## 解决方案

内建的 `dict` 提供了很多创建字典的方法，但是并没有提供这种方式，所以我们得自己写一个函数来达到目的。一个方法是，对扩展的列表切片调用内建的 `zip` 函数：

```
def dictFromList(keysAndValues):
    return dict(zip(keysAndValues[::2], keysAndValues[1::2]))
```

一个更通用的适合任何序列或者可迭代参数的方式是，把从给定序列中获取多个数对 (pair) 的过程独立出来，变成一个单独的生成器。这方法不如 `dictFromList` 简洁，但是速度却更快，通用性也更好：

```
def pairwise(iterable):
    itnext = iter(iterable).next
    while True:
        yield itnext(), itnext()
def dictFromSequence(seq):
    return dict(pairwise(seq))
```

定义 `pairwise` 函数也使得我们可以用任意序列来更新任意一个已存在的字典，比如，`mydict.update(pairwise(seq))`。

## 讨论

本节介绍的两种“工厂函数”的实现方式本质上都是用同样的方法创建字典：都生成了一个(`key, value`)值对的序列，并将其作为参数传递给 `dict`。区别是它们具体怎么生成这个值对的序列。

`dictFromList` 使用内建函数 `zip`，并用 `keysAndValue` 列表的两个切片作为参数——两个切片分别根据奇偶索引搜集元素（一个的索引是 0、2、4…另一个则是 1、3、5…）。这个方法不错，但只在 `keysAndValues` 是支持扩展切片的类型或者类的实例时才有效，比如 `list`、`tuple` 或者 `str`。另外，它还会在内存中创建一些临时列表，如果 `keysAndValues` 是个很长的序列，这些列表的创建过程会降低一些性能。

而 `dictFromSequence` 则把创建值对序列的任务委托给了一个叫做 `pairwise` 的生成器。`pairwise` 的实现方式使得它能够使用任何可迭代对象——不仅仅是列表（或者元组、字符串等），甚至包括其他生成器的结果、文件、字典等。而且 `pairwise` 一次只生成一对，它从来不在内存中创建大列表，因此即使给定的序列很长，它的性能也不会有什么降低。

`pairwise` 的实现也很有趣。`pairwise` 的第一行语句将内建函数 `iter` 应用于传入的 `iterable`

参数，获得一个迭代器，然后再将一个本地变量 `itnext` 绑定到这个迭代器的 `next` 方法。这看上去有点奇怪，但这是 Python 中一种很好的通用的技巧：如果你有一个对象，你想对这个对象所做的事是在循环中不断调用它的一个方法，可以给它的这个被绑定的方法赋予一个本地的名字，然后就可以直接调用这个本地名字了，就像调用一个函数一样。对某些习惯其他语言的用户来说，像下面这样调用 `next` 方法可能会更舒适一些：

```
def pairwise_slow(iterable):
    it = iter(iterable)
    while True:
        yield it.next(), it.next()
```

不过，这个 `pairwise_slow` 变体并不比解决方案中的 `pairwise` 简单（“对不懂 Python 的人来说显得更熟悉”并不表示“更简单”），而且它慢了约 60%。重视简洁和清晰的确很重要，而且也是 Python 的核心价值观。完全不考虑任何性能问题也是一种主张，但在实践中这种观点不可能在任何语言中得到推荐。所以，既然我们都知道编写正确、清晰、简单代码的重要性，那么为什么不学习和遵循那些更加符合我们需要的 Python 用法呢？

## 更多资料

见 19.7 节中的关于用滑动窗口来循环可迭代对象的通用方法。参考 Python Reference Manual 中更多关于扩展切片的资料。

## 4.13 获取字典的一个子集

感谢：David Benjamin

### 任务

你有一个巨大的字典，字典中的一些键属于一个特定的集合，而你想创建一个包含这个键集合及其对应值的新字典。

### 解决方案

如果你不想改动原字典：

```
def sub_dict(somedict, somekeys, default=None):
    return dict([(k, somedict.get(k, default)) for k in somekeys])
```

如果你从原字典中删除那些符合条件的条目：

```
def sub_dict_remove(somedict, somekeys, default=None):
    return dict([(k, somedict.pop(k, default)) for k in somekeys])
```

下面是两个函数的使用和效果：

```
>>> d = {'a': 5, 'b': 6, 'c': 7}
>>> print sub_dict(d, 'ab'), d
{'a': 5, 'b': 6} {'a': 5, 'b': 6, 'c': 7}
>>> print sub_dict_remove(d, 'ab'), d
{'a': 5, 'b': 6} {'c': 7}
```

## 讨论

在 Python 中，我在很多地方都用到了字典——数据库的行、主键和复合键，用于模板解析的变量名字空间等。我常常需要基于另外一个已有的大字典创建一个新字典，此字典的键是大字典的键的一个子集。在大多数情况下，原字典应该保持不变；但有时，我也需要在完成了抽取之后删除在原字典中的子集。本节的解决方案对两种可能性都给出了答案。区别仅仅在于，如果需要原字典保持原样不变，使用 `get` 方法，如果需要删除子集，则使用 `pop` 方法。

如果 `somekeys` 中的某元素 `k` 并不是 `somedict` 的键，解决方案提供的函数会将 `k` 作为结果的键，并对应一个默认值（可以作为一个可选的参数传递给这两个函数，默认情况下是 `None`）。所以，最终结果也不一定是 `somedict` 的子集。不过我却发现这种行为方式对我的应用非常有帮助。

当你认为 `somekeys` 中的所有的元素都应当是 `somedict` 的键时，也许会希望在键“缺失”的时候获得一个异常，它可以提示和警告你程序中的 bug。记住，Tim Peters 在 *The Zen of Python* 中说过“错误不应该被静静地略过，除非有意为之”（在 Python 的交互式解释器的提示符下敲入 `import this` 并回车，你将看到精炼的 Python 设计原则）。所以，如果从你的应用的角度看，键不匹配是一个错误，那么会希望马上得到一个异常来提醒你错误的发生。如果这的确是你所希望的，可以对解决方案中的函数略作修改：

```
def sub_dict_strict(somedict, somekeys):
    return dict([(k, somedict[k]) for k in somekeys])
def sub_dict_remove_strict(somedict, somekeys):
    return dict([(k, somedict.pop(k)) for k in somekeys])
```

这些更加严格的变体版本甚至比原版本更简单——这充分说明了 Python 本来就喜欢在意外发生时抛出异常。

或者，你希望在键不匹配时直接将其忽略。这也只需要一点点修改：

```
def sub_dict_select(somedict, somekeys):
    return dict([(k, somedict[k]) for k in somekeys if k in somedict])
def sub_dict_remove_select(somedict, somekeys):
    return dict([(k, somedict.pop(k)) for k in somekeys if k in somedict])
```

列表推导中的 `if` 子句做完了我们期望的事，即在应用 `k` 之前先做鉴别工作。

在 Python 2.4 中可以用生成器表达式来替代列表推导，用它作为本节中的函数的参数。

我们只需略微修改 dict 的调用，将 `dict([ ...])` 改成 `dict( ... )`（移除临近圆括号的方括号），就能享受进一步的简化和速度的提升。不过这些修改不适用 Python 2.3，因为它只支持列表推导而不支持生成器表达式。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 文档中关于 dict 的部分。

## 4.14 反转字典

感谢：Joel Lawhead、Ian Bollinger、Raymond Hettinger

### 任务

给定一个字典，此字典将不同的键映射到不同的值。而你想创建一个反转的字典，将各个值映射到键。

### 解决方案

可以创建一个函数，此函数传递一个列表推导作为 dict 的参数以创建需要的字典。

```
def invert_dict(d):
    return dict([(v, k) for k, v in d.iteritems()])
```

对于比较大的字典，用 Python 标准库 `itertools` 模块提供的 `izip` 会更快一些：

```
from itertools import izip
def invert_dict_fast(d):
    return dict(izip(d.itervalues(), d.iterkeys()))
```

### 讨论

如果字典 `d` 中的值不是独一无二的，那么 `d` 无法被真正地反转，也就是不存在这样的字典，对于任意给定的键 `k`，满足 `id[d[k]]==k`。不过，本节展示的函数在这种情况下仍然能够创建一个“伪反转”字典 `pd`，对于任何属于字典 `d` 地值 `v`，`d[pd[v]]==v`。如果你原始的字典 `d`，以及用本节函数获得的字典 `x`，可以很容易地检查 `x` 是 `d` 的反转字典还是伪反转字典：当且仅当 `len(x)==len(d)` 时，`x` 才是 `d` 的真正的反转字典。这是因为，如果两个不同的键对应相同的值，对于解决方案给出的两个函数来说，两个键中的一个一定会消失，因而生成的伪反转字典的长度也会比原字典的长度短。在任何情况下，只有当 `d` 中的值是可哈希（`hashable`，意味着可以用它们做字典的键）的，前面展示的函数才能正常工作，否则，函数会抛出一个 `TypeError` 异常。

当我们编写 Python 程序时，我们通常会“无视小的优化”，正如 Donald Knuth 在 30 年前所说的“比起速度，我们更珍视清晰和正确性。”不过，了解更多让程序变快的知识

也没有害处：当我们为了简单和清晰而采用某种方法编写程序时，我们最好深入地考虑一下我们的决定，不要懵懵懂懂。

在这里，解决方案中的 `invert_dict` 函数可能会被认为更清晰，因为它清楚地表达了它在做的事。该函数取得了由 `iteritems` 方法生成的成对的键及其对应值 `k` 和 `v`，将它们包裹成`(value, key)`的顺序，并把最后生成的序列作为参数赋给 `dict`，这样 `dict` 就构建出了一个值成为键，而原先的键变成了对应值的新字典——正是我们需要的反转字典。

而解决方案中 `invert_dict_fast` 函数其实也没有那么复杂，它的操作更加抽象，它首先将所有的键和值分别转为两个独立的迭代器，再通过调用 Python 标准库 `itertools` 模块提供的 `izip` 将两个迭代器转化为一个迭代器，其中每个元素都是像`(value, key)`一样的一对值。如果你能够习惯于这种抽象层次，你将体会到更高层次的简洁和清晰。

由于这种高度的抽象性，以及不具化（materialize）整个列表（而是通过生成器和迭代器一次生成一项）的特性，`invert_dict_fast` 能够比 `invert_dict` 快很多。比如，在我的计算机上，反转 10 000 个条目的字典，`invert_dict` 耗时 63ms，而 `invert_dict_fast` 则仅用时 20ms。速度提升了 3 倍，颇为可观。当你处理大规模数据时，由于代码的高度抽象性而带来的性能提升将会变得更加明显。特别是当你使用 `itertools` 来替换循环和列表推导时，执行速度同样也能获得极大提升，因为你无须在内存中具化一些超大的列表。当你习惯了更高的抽象层次，性能的提升只是一个额外收益，除此之外，你在观念和创造性上也会有所进步。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中的映射类型和 `itertools` 的文档；第 19 章。

## 4.15 字典的一键多值

感谢：Credit: Michael Chermside

### 任务

需要一个字典，能够将每个键映射到多个值上。

### 解决方案

正常情况下，字典是一对一映射的，但要实现一对多映射也不难，换句话说，即一个键对应多个值。你有两个可选方案，但具体要看你怎么看待键的多个对应值的重复。下面这种方法，使用 `list` 作为 `dict` 的值，允许重复：

```
d1 = {}  
d1.setdefault(key, []).append(value)
```

另一种方案，使用子字典作为 dict 的值，自然而然地消灭了值重复的可能：

```
d2 = {}  
d2.setdefault(key, {})[value] = 1
```

在 Python 2.4 中，这种无重复值的方法可等价地被修改为：

```
d3 = {}  
d3.setdefault(key, set()).add(value)
```

## 讨论

正常的字典简单地将一个键映射到一个值上。本节则展示了三个简单有效的方法来实现一个键对应多个值的功能，即将字典的值设为列表或字典，若在 Python 2.4 中，还有可能是集合。基于列表的方法的语义和其他两者差别不大，最重大的差别是它们对待值重复的态度。每种方式都依赖字典的 `setdefault` 方法（4.10 节有相关内容）来初始化字典的一个键所对应的条目，并在需要的时候返回上述条目。

除了给键增加对应值之外，还要做更多的事情。对于使用列表并允许重复的第一个方式，下面代码可取得键对应的值列表：

```
list_of_values = d1[key]
```

如果不介意当键的所有值都被移除后，仍留下一个空列表作为 `d1` 的值，可以用下面方法删除键的对应值：

```
d1[key].remove(value)
```

虽然有空列表，但要想检查一个键是否至少有一个值还是很容易的，使用一个总是返回列表（也可能是空列表）的函数就行了：

```
def get_values_if_any(d, key):  
    return d.get(key, [])
```

比如，为了检查 “freep” 是否是字典 `d1` 的键 “somekey” 的对应值之一，可以这样编写代码：`if 'freep' in get_values_if_any(d1, 'somekey')`。

使用子字典且没有值重复的第二种方式的用法非常类似。为了获得键的对应值列表，具体做法是：

```
list_of_values = list(d2[key])
```

为了移除某键的一个值，可以像下面这样做，当然，当键的值都被删除之后，字典 `d2` 中仍会留下一个空字典：

```
del d2[key][value]
```

第三种方式，只适用于 Python 2.4 以上并使用了集合的方式，它的移除键值的操作如下：

```
d3[key].remove(value)
```

第二和第三种方式（无重复）的 `get_value_if_any` 函数：

```
def get_values_if_any(d, key):
    return list(d.get(key, ()))
```

本节讨论了如何实现一个很基本的功能，但并没有提到如何以一种系统化的方式来应用它，你也许会考虑将这些代码封装成一个类。要达到这个目的，必须得通盘考虑你的设计。你能否接受某个值和某个键的对应关系出现多次？（用数学的语言可表述为，对于每个键而言，条目究竟是包还是集合？）如果是的话，则 `remove` 方法究竟是将总的对应次数减一，还是完全地删掉那些对应关系？这只是你面临各种各样决定的一个开始，不过，要想做出正确选择，必须基于应用的实际需求来考虑。

## 更多资料

4.10 节，*Library Reference* 和 *Python in a Nutshell* 关于映射类型的章节；18.8 节中对于 `bag` 类型的实现。

## 4.16 用字典分派方法和函数

感谢：Dick Wall

### 任务

需要根据某个控制变量的值执行不同的代码片段——在其他的语言中你可能会使用 `case` 语句。

### 解决方案

归功于面向对象编程的优雅的分派概念，`case` 语句的使用大多（但不是所有）都可以被替换成其他分派形式。在 Python 中，字典及函数是一等（first-class）对象这个事实（比如函数可以作为字典中的值被存储），使得“`case` 语句”的问题更容易被解决。比如，考虑下面的代码片段：

```
animals = []
number_of_felines = 0
def deal_with_a_cat():
    global number_of_felines
    print "meow"
    animals.append('feline')
    number_of_felines += 1
def deal_with_a_dog():
    print "bark"
    animals.append('canine')
def deal_with_a_bear():
    print "watch out for the *HUG*!"
```

```
    animals.append('ursine')
tokenDict = {
    "cat": deal_with_a_cat,
    "dog": deal_with_a_dog,
    "bear": deal_with_a_bear,
}
# 模拟, 比如, 从文件中读取的一些单词
words = ["cat", "bear", "cat", "dog"]
for word in words:
    # 查找每个单词对应的函数调用并调用之
    return tokenDict[word]()
nf = number_of_felines
print 'we met %d feline%s' % (nf, 's'[nf==1:])
print 'the animals we met were:', ''.join(animals)
```

## 讨论

本节的要点是, 构建一个字典, 以字符串 (或其他对象) 为键, 以被绑定的方法、函数或其他的可调用体作为值。在每一步的执行过程中, 我们都先用字符串键来选择需要执行的可调用体。这个方法可以被当做一个通用的 `case` 语句用于各处。

这确实非常简单, 我也经常使用这种技术。还可以使用被绑定的方法或者其他可调用体来替换本节示例中查找的函数。但当你使用未绑定的方法时, 需要传递一个正确的对象作为第一个参数来调用它们。还可以将可调用体和它所需要的参数放在一个元组中, 然后把元组当做字典的值存储起来, 这样具有更强的通用性。

在别的语言中, 我可能需要 `case`、`switch` 或者 `select` 语句, 但在 Python 中, 所有类似的地方我都用这个技术来实现同样的功能。

## 更多资料

*Library Reference* 中关于映射类型的章节; *Reference Manual* 中关于绑定和非绑定方法的介绍; *Python in a Nutshell* 中关于字典和可调用体的介绍。

## 4.17 字典的并集与交集

感谢: Tom Good、Andy McKay、Sami Hangaslammi、Robin Siebler

### 任务

给定两个字典, 需要找到两个字典都包含的键 (交集), 或者同时属于两个字典的键 (并集)。

### 解决方案

有时, 尤其是在 Python 2.3 中, 你会发现对字典的使用完全是对集合的一种具体化的

体现。在这个要求中，只需要考虑键，不用考虑键的对应值，一般可以通过调用 `dict.fromkeys` 来创建字典，像这样：

```
a = dict.fromkeys(xrange(1000))
b = dict.fromkeys(xrange(500, 1500))
```

最快计算出并集的方法是：

```
union = dict(a, **b)
```

而最快且最简洁地获得交集的方法是：

```
inter = dict.fromkeys([x for x in a if x in b])
```

如果字典 `a` 和 `b` 的条目的数目差异很大，那么在 `for` 子句中用较短的那个字典，在 `if` 子句中用较长的字典会有利于提升运算速度。在这种考虑之下，牺牲简洁性以获取性能似乎是值得的，交集计算可以被改为：

```
if len(a) < len(b):
    inter = dict.fromkeys([x for x in a if x not in b])
else:
    inter = dict.fromkeys([x for x in b if x not in a])
```

Python 也提供了直接代表集合的类型（标准库中的 `sets` 模块，在 Python 2.4 中已经成为了内建的部分）。可以把下面的代码片段用在模块的开头，这个代码片段确保了名字 `set` 被绑定到了适合的类型，这样在整个模块中，无论你用 Python 2.3 还是 2.4，都可以使用同样的代码：

```
try:
    set
except NameError:
    from sets import Set as set
```

这样做好处是，可以到处使用 `set` 类型，同时还获得了清晰和简洁，以及速度的提升（在 Python 2.4 中）：

```
a = set(xrange(1000))
b = set(xrange(500, 1500))
union = a | b
inter = a & b
```

## 讨论

虽然 Python 2.3 的标准库模块 `sets` 已经提供了一个优雅的数据类型 `set` 来代表集合（带有可哈希（`hashable`）的元素），但由于历史原因，使用 `dict` 来代表集合仍然是很普遍的。基于这个目的，本节展示了如何用最快的方法来计算这种集合的交集和并集。本节的代码在我的计算机上，并集计算耗时  $260\mu\text{s}$ ，交集计算则耗时  $690\mu\text{s}$ （Python 2.3；在 Python 2.4 中，这两个数字分别是  $260\mu\text{s}$  和  $600\mu\text{s}$ ），而其他的基于循环或者生成器表达式的方法会更慢。

不过，最好还是用 set 类型而不是字典来代表集合。如同本节所示，使用 set 能让代码更加直接和易读。如果你不喜欢或操作符 ( $|$ ) 和与操作符 ( $\&$ )，可以使用等价的 `a.union(b)` 和 `a.intersection(b)`。这样操作除了清晰，速度也有提升，特别是在 Python 2.4 中，计算并集需要  $260\mu\text{s}$ ，但计算交集只需要  $210\mu\text{s}$ 。即使是在 Python 2.3，其速度也是可以接受的：并集计算耗时  $270\mu\text{s}$ ，交集计算耗时  $650\mu\text{s}$ ，没有在 Python 2.4 快，但如果你仍然用字典来代表集合的话，速度其实是相当的。最后一点，一旦你引入 set 类型（无论是 Python 2.4 内建的，还是通过 Python 标准库 `sets` 模块引入的，接口是一样的），你将获得丰富的集合操作。举个例子，属于 `a` 或者 `b` 但却不属于 `a` 和 `b` 的交集的集合是 `a ^ b`，可以等价地被表示为 `a.symmetric_difference(b)`。

即使由于某些原因使用了 `dict`，也应当尽可能地用 `set` 来完成集合操作。举个例子，假设你有个字典 `phones`，将人名映射到电话号码，还有个字典 `addresses`，将人名映射到地址。最清楚简单地打印所有同时知道地址和电话号码的人名及其相关数据的方法是：

```
for name in set(phones) & set(addresses):
    print name, phones[name], addresses[name]
```

跟下面的方法比，这非常简洁，虽然清晰度可能还有争议：

```
for name in phones:
    if name in addresses:
        print name, phones[name], addresses[name]
```

另一个很好的可选方法是：

```
for name in set(phones).intersection(addresses):
    print name, phones[name], addresses[name]
```

如果使用 `intersection` 方法，而不是 $\&$ 交集操作，就不需要将两个字典都转化成 `set`，只需要其中一个。然后再对转化后的 `set` 调用 `intersection`，并传入另一个 `dict` 作为 `intersection` 方法的参数。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 的映射类型、`sets` 模块及 Python 2.4 中的内建 `set` 类型。

## 4.18 搜集命名的子项

感谢：Alex Martelli、Doug Hudgeon

### 任务

你想搜集一系列的子项，并命名这些子项，而且你认为用字典来实现有点不便。

## 解决方案

任意一个类的实例都继承了一个被封装到内部的字典，它用这个字典来记录自己的状态。我们可以很容易地利用这个被封装的字典达到目的，只需要写一个内容几乎为空的类：

```
class Bunch(object):
    def __init__(self, **kwds):
        self.__dict__.update(kwds)
```

现在，为了将变量组织起来，创建一个 Bunch 实例：

```
point = Bunch(datum=y, squared=y*y, coord=x)
```

现在就可以访问并重新绑定那些刚被创建的命名属性了，也可以进行添加、移除某些属性之类操作。比如：

```
if point.squared > threshold:
    point.isok = True
```

## 讨论

我们常常需要搜集一些元素，然后给它们命名。这个需求用字典来实现完全没有问题，但是利用一个几乎什么都不做的小类明显更加方便美观。

如同解决方案的代码所示，创建一个小小的类，提供参数访问的语法，我们几乎不用写什么东西。字典也适合用来搜集一些子项，每个子项都有自己的名字（根据环境，字典中的子项的键可以被认为是该子项的名字），但如果所有的名字都是标识符，而且被当做变量使用，字典并不是最好的方案。在类 Bunch 的 `__init__` 方法中，通过 `**kwds` 语法，可以接受任意的命名参数，并且用 `kwds` 参数来更新实例的空字典，这样，每个命名的参数都成为实例的一个属性。

与访问属性的语法相比，字典索引语法不是那么简洁和易读。比如，如果 `point` 是个字典，解决方案中的最后的那个代码片段就应该是这样：

```
if point['squared'] > threshold:
    point['isok'] = True
```

此外，还有另一个备选的实现方案，看上去也很吸引人：

```
class EvenSimplerBunch(object):
    def __init__(self, **kwds):
        self.__dict__ = kwds
```

将实例的字典重新绑定可能会让人感到不安，但比起调用字典的 `update` 方法，它不会造成什么不好的后果。所以，你可能会喜欢这个备选的 Bunch 实现在速度上的优势。不过，我从来没有在任何 Python 文档中看到对下面用法的担保：

```
d = {'foo': 'bar'}
x = EvenSimplerBunch(**d)
```

最好使 `x._dict_` 成为字典 `d` 的一个独立的拷贝，而不是共享一个引用。现在这个方法的确有效，在各个版本中都能工作，但除非语法文档规定了其语义，否则我们不能确信这种做法会永远有效。所以，如果选择了 `EvenSimplerBunch` 的实现，你可能会选择赋值一个拷贝 (`dict(kwds)` 或者 `kwds.copy()`)，而不是 `kwds` 本身。而且，如果这样做，那一点速度优势也就消失了。总之，最好还是将原先的 `Bunch` 的实现方法作为首选。

另一个富诱惑的做法是直接让 `Bunch` 类继承 `dict`，并将属性访问的特殊方法设为该子项本身的属性方法，像下面这样：

```
class DictBunch(dict):
    __getattr__ = dict.__getitem__
    __setattr__ = dict.__setitem__
    __delattr__ = dict.__delitem__
```

这个方法的一个问题是，根据定义，`DictBunch` 的一个实例 `x` 会拥有很多它实际上没有的属性，因为它获得了 `dict` 所有的属性（实际上是方法，但在这个环境中其实没什么区别）。所以，你通过 `hasattr(x, someattr)` 来检查属性没有意义，但可以对先前实现的 `Bunch` 和 `EvenSimplerBunch` 这么做，而且，你还得事先排除 `someattr` 的值是一些通用语如“`keys`”、“`pop`”和“`get`”等的可能性。

Python 的关于属性和子项的区别是这门语言清晰和简洁的源泉。不幸的是，很多 Python 新手错误地以为将属性和子项混为一谈并没有什么问题，这大概是因为先前的 JavaScript 和其他语言的经验，在 JavaScript 中，属性和条目通常是可以混为一谈的。不过新手应该把概念厘清，不要继续稀里糊涂。

## 更多资料

Python Tutorial 关于类的章节，*Language Reference* 和 *Python in a Nutshell* 中对类的介绍；第 6 章关于 Python 面向对象编程的介绍；4.18 节对于 `**kwds` 语法的介绍。

## 4.19 用一条语句完成赋值和测试

感谢：Alex Martelli、Martin Miller

### 任务

你正在将 C 或者 Perl 代码转换成 Python 代码，并试图尽量保留原有的结构，你现在需要一种表达方式，能够同时完成赋值和测试（如同其他语言中的 `if((x=foo( ))` 或 `while((x=foo( ))`）。

## 解决方案

在 Python 中，不能这么写代码：`if x=foo():...` 赋值是一个语句，不是一个表达式，而你只能在 `if` 和 `while` 中使用表达式作为条件。不过问题不大，只需要将代码修改得更 Python 化一点。举个例子，要对一个文件对象 `f` 逐行处理，C 风格的写法（在 Python 中这样的句法是错误的）应该是这样：

```
while (line=f.readline( )) != '':
    process(line)
```

而 Python 风格的写法（更易读、清爽及快速）：

```
for line in f:
    process(line)
```

有时，需要将 C、Perl 或其他语言编写的程序转换成 Python 代码，而且希望尽可能保留原有的结构。写一个简单的工具类会起到很大的作用：

```
class DataHolder(object):
    def __init__(self, value=None):
        self.value = value
    def set(self, value):
        self.value = value
        return value
    def get(self):
        return self.value
# 可选的，强烈不建议使用，但有时确实很方便：
import __builtin__
__builtin__.DataHolder = DataHolder
__builtin__.data = data = DataHolder()
```

在 `DataHolder` 类和它的实例 `data` 的帮助下，原有的 C 风格结构得以保留：

```
while data.set(file.readline( )) != '':
    process(data.get( ))
```

## 讨论

在 Python 中赋值是语句，不是表达式。因此，在 `if`、`elif` 或 `while` 语句中，你无法将正在测试的东西赋值给其他名字或变量。这也没什么，调整你的程序结构，避免在测试的时候赋值即可（这样代码会显得更清晰）。具体地说，在任何时候如果你感到需要在一个 `while` 循环中同时完成赋值和测试，都应该认识到，你的循环结构可能需要被重构成一个生成器（或者其他迭代器）。一旦以这种思路进行重构，你的循环就变成了简单而直接的 `for` 语句。解决方案给的例子，循环读取文本文件中的行，正是通过 Python 本身完成了重构，因为 `file` 对象就是一个迭代器，其中的元素是文件的行。

不过，有时必须转换由 C、Perl 或其他语言编写的程序到 Python 代码，这些语言本身是支持赋值作为表达式的。当实现某个已经提供了参考实现的算法或者书上的算法等，并用 Python 编写其最初版本的时候，经常会遇到这种用赋值语句作表达式的情况。在这个条件下，让最初的 Python 实现代码贴近原有的结构是明智的。可以以后再对你的代码重构，使之更像 Python——清晰、快速等。不过首先，需要尽可能快地完成一个能工作的版本，而且需要你的代码接近原型以便进行错误和兼容性的检查。幸运的是，Python 具有足够的能力来满足你的需求。

Python 不让我们重新定义赋值的含义，不过我们可以写一个方法（或函数），把参数存在“某处”，同时也能返回那个参数用于测试。在这里，说到“某处”，我们会很自然地想到用一个对象的属性来代表这个“某处”，因此对象的方法是比函数更自然的选择。当然也可以直接去访问属性（这样，`get` 方法就变得多余了），不过，我感觉提供 `data.set` 和 `data.get` 会更匀称整齐一点。

`data.set(whatever)` 比起 `data.value=whatever` 多了一点句法上的好处，即，这个加入的值可以是一个表达式。因此，它是一个很棒的方法，能够帮助我们完成忠实于原型的代码翻译。这样的 Python 代码和原型代码如 C 或 Perl 代码的唯一区别仅仅是句法上的微小差异——但总体结构一致，这是我们最关心的议题。

引入`__builtin__` 并给它的属性赋值是一个小花招，其本质上是在运行时定义了一个新的内建对象。可以在你的应用程序开头玩这个花招，但马上所有的其他模块都自动地具有了访问这个新内建对象的能力，而且还无须引入模块。这不是好的 Python 解决问题的方式，相反，这是在挑战 Python 的良好品味的尺度，因为其他模块完全不应该为你的应用程序所造成的副作用负责。不过，既然本节的目的是提供一个快速且肮脏的花招来解决代码的初次翻译问题，之后将要进行的重构会改良整个代码，那么在这样特殊情况下，只要我们的花招不被用于正常的产品代码，这种手段还可以让人忍受。

另一方面，还有一个花招你是绝对不应该用的，即利用列表推导的一个漏洞：

```
while [line for line in [f.readline() if line!='']:  
    process(line)
```

这个花招目前还能正常工作，这是因为 Python 2.3 和 2.4 都将列表推导的控制变量（这里是 `line`）“泄露”到周边的空间中。这是一个很容易混淆和不易读的手段，而且它也被废弃了，列表推导控制变量的泄露问题将在未来的 Python 版本中被修正，那时这个招数就彻底失灵了。

## 更多资料

Tutorial 的关于类的文档；*Library Reference* 和 *Python in a Nutshell* 中关于`__builtin__` 模块的文档；*Language Reference* 和 *Python in a Nutshell* 中列表推导的文档。

## 4.20 在 Python 中使用 printf

感谢: Tobias Klausmann、Andrea Cavalcanti

### 任务

你喜欢用 C 提供的 printf 函数将某些东西打印到程序的标准输出，但 Python 并不提供这样的函数。

### 解决方案

在 Python 中实现 printf 是很简单的事：

```
import sys
def printf(format, *args):
    sys.stdout.write(format % args)
```

### 讨论

Python 分开了输出（print 语句）和格式化（% 操作符），如果你希望将这两个融合在一起，如同解决方案所示，它的实现是很简单的。无须担心空格和新行的自动插入，而只需要担心格式和参数的正确匹配问题。

举个例子，Python 的普遍做法是：

```
print 'Result tuple is: %r' % (result_tuple,),
```

它考虑得非常周到，但这两个逗号的意义不是很明确（一个在 result\_tuple 之后，用于创建单元素元组，另一个则是避免了 print 插入默认的换行符），不过既然有了解决方案中提供的 printf 函数，就可以这样写：

```
printf('Result tuple is: %r', result_tuple)
```

### 更多资料

*Library Reference* 和 *Python in a Nutshell* 中关于 sys 模块以及字符串格式化操作符%的文档，2.13 节提出的在 Python 中实现 C++ 风格的<<的方法。

## 4.21 以指定的概率获取元素

感谢: Kevin Parks、Peter Cogolo

### 任务

你想从一个列表中随机获取元素，就像 random.choice 所做的一样，但同时必须根据另

一个列表指定的各个不同元素的概率来获取元素，而不是用等同的概率撷取元素。

## 解决方案

Python 标准库中的 random 模块提供了生成和使用伪随机数的能力，但是它并没有提供这样特殊的功能，所以，我们必须得自己写一个函数：

```
import random
def random_pick(some_list, probabilities):
    x = random.uniform(0, 1)
    cumulative_probability = 0.0
    for item, item_probability in zip(some_list, probabilities):
        cumulative_probability += item_probability
        if x < cumulative_probability: break
    return item
```

## 讨论

Python 标准库中的 random 模块并没有提供根据权重做出选择的功能，这种功能在游戏、模拟和随机测试中是很常见的需求，所以，本节的目标是提供此功能的实现。解决方案使用了 random 模块的 uniform 函数获得了一个在 0.0 和 1.0 之间分布的伪随机数，之后同时循环元素及其概率，计算不断增加的累积概率，直到这个概率值大于伪随机数。

本节假设（但并未检查）概率序列 probabilities 具有和 some\_list 一样的长度，其所有元素都在 0.0 和 1.0 之间，且相加之和为 1.0；如果违反了这个假设，仍能进行一些随机的撷取，但不能完全地遵循（不连贯）函数的参数所规定的行为。可能想在函数开头加上一些 assert 语句以确保参数的有效性：

```
assert len(some_list) == len(probabilities)
assert 0 <= min(probabilities) and max(probabilities) <= 1
assert abs(sum(probabilities)-1.0) < 1.0e-5
```

不过，这些检查会消耗一些时间，所以我通常都不这么做，在正式的解决方案中我也没有把它们纳入。

正如我前面提到的，这个任务要求每一项都有一个应对的概率，这些概率分布在 0 和 1 之间，且总和相加为 1。另一个有点类似的任务是根据一个非负整数的序列所定义的权重进行随机撷取——基于机会，而不是概率。对于这个问题，最好的解决方案是使用生成器，其内部结构和解决方案中的 random\_pick 函数差异很大：

```
import random
def random_picks(sequence, relative_odds):
    table = [z for x, y in zip(sequence, relative_odds) for z in [x]*y]
    while True:
        yield random.choice(table)
```

生成器首先准备一个 table，它的元素的数目是 sum(relative\_odds)个，sequence 中的每个元素都可以在 table 中出现多次，出现的次数等于它在 relative\_odds 序列中所对应的非负整数。一旦 table 被制作完毕，生成器的主体就可以变得又小又快，因为它只需要将随机撷取的工作委托给 random.choice。举个例子，关于这个 random\_picks 的典型应用：

```
>>> x = random_picks('ciao', [1, 1, 3, 2])
>>> for two_chars in zip('boo', x): print ''.join(two_chars),
bc oa oa
>>> import itertools
>>> print ''.join(itertools.islice(x, 8))
icacaoco
```

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中的 random 模块。

## 4.22 在表达式中处理异常

感谢：Chris Perkins、Gregor Rayman、Scott David Daniels

### 任务

你想写一个表达式，所以你无法直接用 try/except 语句，但你仍需要处理表达式可能抛出的异常。

### 解决方案

为了抓住异常，try/except 是必不可少的，但 try/except 是一条语句，在表达式内部使用它的唯一方法是借助一个辅助函数：

```
def throws(t, f, *a, **k):
    ''' 如果 f(*a, **k) 抛出一个异常且其类型是 t 的话则返回 True
    (或者，如果 t 是一个元组的话，类型是 t 中的某项) '''
    try:
        f(*a, **k)
    except t:
        return True
    else:
        return False
```

举个例子，假设你有一个文本文件，每行有一个数字，但文件也可能有多余的内容如空格行及注释行等。可以生成一个包含文件中的所有数字的列表，只需略去那些不是数字的行即可：

```
data = [float(line) for line in open(some_file)
        if not throws(ValueError, float, line)]
```

## 讨论

你可能会喜欢将函数命名为 `raises`, 但我个人更喜欢 `throws`, 可能是出于对 C++ 的感情。不过不管什么名字, 这个辅助函数都接受一个异常类型 `t` 作为第一个参数, 接着是一个可调用体 `f`, 然后是任意的基于位置的参数 `a` 和命名参数 `k`, 两者都将被传递给 `f`。像 `if not throws(ValueError, float(line))` 这样的写法是不行的。当你调用函数时, Python 在将控制权交给函数之前会对参数求值, 如果参数的求值引发了异常, 函数永远不会得到机会执行。这种情况, 在很多人刚开始用 Python 标准库的 `unittest.TestCase` 类的 `assertRaises` 方法时屡有发生, 我见过不止一次。

当 `throws` 函数执行时, 它在 `try/except` 语句的 `try` 子句中调用 `f`, 将那个任意的基于位置的参数和命名参数传递给 `f`。如果在 `try` 子句中对 `f` 的调用引发了异常, 且异常的类型是 `t` (或者是列出的异常类型中的一种, 如果 `t` 是一个异常类型的元组的话), 则控制权交给了对应的 `except` 子句, 在此例中, 返回 `true` 作为 `throws` 的结果。如果 `try` 子句中没有异常发生, 控制权会交给对应的 `else` 子句 (如果有), 它将返回 `false` 作为 `throws` 的结果。

注意, 如果有什么非预期的异常 (类型不在 `t` 的指定范围内), `throws` 函数并不会尝试截获异常, 因而 `throws` 就将被终止, 异常则交给它的调用者。这是一个有意为之的设计。在 `except` 子句中用太大的网去捕获异常并不是什么好主意, 那常常意味着查错查得头昏脑胀。如果调用者真的想要 `throws` 截获所有的异常, 它可以这样调用: `throws(Exception, ...)` 然后, 就等着头疼吧。

`throws` 函数的问题是, 实际上做了两次关键操作: 一次是看它有没有抛出异常, 先把结果抛诸脑后, 另一次是获得结果。所以最好的结局是同时获得结果和被截获异常的提示。我刚开始是这么做的:

```
def throws(t, f, *a, **k):
    """ 如果 f(*a, **k) 抛出异常且异常类型为 t, 返回 (True, None)
    或者 (False, x), 其中 x 是 f(*a, **k) 的结果 """
    try:
        return False, f(*a, **k)
    except t:
        return True, None
```

不幸的是, 这个版本不符合列表推导的要求, 没有什么优雅的办法能够同时得到标志和结果。因此, 我选择了一个不同的方法: 一个在任何情况下都返回 `list` 的函数——如果有异常被捕获就返回空列表, 否则就返回仅包含结果的列表。这个方法工作得很好, 但是为了清晰起见, 最好把函数名改一改:

```
def returns(t, f, *a, **k):
    """ 正常情况下返回 [f(*a, **k)], 若有异常返回 [ ] """
    try:
```

```
        return [ f(*a, **k) ]
    except t:
        return [ ]
```

最后生成的列表推导变得更加优雅，比解决方案中的版本好多了，至少我这么认为。

```
data = [ x for line in open(some_file)
         for x in returns(ValueError, float, line) ]
```

## 更多资料

*Python in a Nutshell* 中关于截获和处理异常的文档; 4.8 节对 \*args 和 \*\*kwds 语法的介绍。

## 4.23 确保名字已经在给定模块中被定义

感谢: Steven Cummings

### 任务

你想确保某个名字已经在一个给定的模块中定义过了（比如，你想确认已经存在一个内建的名字 `set` 了），如果该名字未被定义，你想执行一些代码来完成定义。

### 解决方案

这个任务的解决办法是我见到过的 `exec` 语句最好的用武之地。`exec` 使得我们可以执行一个字符串中的任意 Python 代码，这让我们可以写一个很简单的函数来达到目的：

```
import __builtin__
def ensureDefined(name, defining_code, target=__builtin__):
    if not hasattr(target, name):
        d = {}
        exec defining_code in d
        assert name in d, 'Code %r did not set name %r' % (
            defining_code, name)
        setattr(target, name, d[name])
```

### 讨论

如果你的代码要支持很多版本的 Python（或者支持第三方的包），那么你的很多模块可能得以这样的代码片段作为开头（这样可以确保 `set` 在 Python 2.4 或者 2.3 中都被正确地设置了，当然在 Python 2.4 中 `set` 其实是内建类型，而在 Python 2.3 中我们需要从标准库中导入 `set`）：

```
try:
    set
except NameError:
    from sets import Set as set
```

本节解决方案将逻辑直接封装起来，默认情况下即可工作于 `__builtin__` 模块，这是因为当你在老的 Python 版本中处理名字问题时，它是典型的需要用到的模块。用本节的解决方案，只需在程序初始化的时候运行以下代码一次，就可以确保名字 `set` 被正确地定义了：

```
ensureDefined('set', 'from sets import Set as set')
```

这个方法的最大优势是，你只需在初始化的时候，在程序中的某处调用 `ensureDefined` 即可，而无须在各个模块的开头写一堆 `try/except` 语句。另外，`ensureDefined` 使得代码可读性更好，因为它只做一件事，因此调用它的目的也一目了然，而 `try/except` 语句却应用面很广，需要花些时间才能看清楚和理解它们。最后一点，`try/except` 在类似于 `pychecker` 的检查工具中可能引发警告，而本节的做法却可以绕过这个问题（如果你没有用过 `pychecker` 或者其他类似的工具，应该赶紧去试试。<http://pychecker.sourceforge.net/>）

我们使用了一个辅助性的字典 `d` 作为 `exec` 语句的目标，而且也只转换被要求的名字，我们竭力避免对 `target` 造成的一些预期外的副作用。也可以使用一些非模块的对象（类，甚至类的实例）来作为 `target`，这样就无须给 `target` 加上一个叫做 `__builtins__` 的属性来引用 Python 内建类型的字典。如果想用得更随意一些，`if` 语句的主体部分还可以变成：

```
exec defining_code in vars(target)
```

你将不可避免地造成一些副影响，见 <http://www.python.org/doc/current/ref/exec.html> 的介绍。

很重要的一点是，一定要清楚 `exec` 能够执行给它的任何包含 Python 代码的有效字符串。因此，必须确保在你调用函数 `ensureDefined` 时，传递给参数 `defining_code` 的值不是来自于一个不可信的来源，比如一个被恶意修改过的文本文件。

## 更多资料

Python Language Reference Manual 中关于 `exec` 的在线文档：<http://www.python.org/doc/current/ref/exec.html>

## 第 5 章

# 搜索和排序

### 引言

感谢：Tim Peters，PythonLabs

在 1960 年代，计算机制造商们曾经估计，如果将所有的用户计入，他们制造的计算机有 25% 的运行时间被用于排序。实际上，有很多计算机花了超过一半的计算时间在排序上。通过这样的评估结果，我们可以得出结论，可能 (i) 确实有很多非常重要的和排序相关的应用，或者 (ii) 很多人在进行一些不必要的排序计算，再或者 (iii) 低效的排序算法被广泛应用造成了计算时间的浪费。

——Donald Knuth

The Art of Computer Programming, vol.3, Sorting and Searching, 第 3 页

在 Knuth 教授的巨著中，有关搜索和排序主题的部分是长达近 800 页的复杂的技术文献。在 Python 实践中，我们把它归纳为最重要的两条（我们已经读过了 Knuth 的书，所以不用去读了）：

- 当需要排序的时候，尽量设法使用内建 Python 列表的 sort 方法；
- 当需要搜索的时候，尽量设法使用内建的字典。

本章的很多内容将会展示这两条原则。最常见的主题是使用 `decorate-sort-undecorate` (DSU) 模式，这是一种通用的方法，通过创建一个辅助的列表，我们可以将问题转化为列表的排序，从而可以利用默认的快速的 sort 方法。这个技术是在本章中最有用的部分。事实上，DSU 是如此常用，以至于 Python 2.4 导入了新的特性来使之更易于使用。因此很多节的解决方案在 Python 2.4 中变得简单多了，但本章也讨论了老版本中已经更新过的解决方案。

DSU 依赖 Python 的内建比较 (built-in comparison) 的一个不常见的特性：序列是按照条目的顺序 (lexicographically) 进行比较的。条目顺序 (lexicographical order) 是对列

表和元组的字符串比较（即字母顺序）规则的归纳。假设 s1 和 s2 是序列，内建函数 cmp(s1, s2) 等价于下面的 Python 代码：

```
def lexicmp(s1, s2):
    # 找到最靠左的不相等的一对
    i = 0
    while i < len(s1) and i < len(s2):
        outcome = cmp(s1[i], s2[i])
        if outcome:
            return outcome
        i += 1
    # 全部相等，其中一个序列已经消耗完所有元素
    return cmp(len(s1), len(s2))
```

这段代码试图找到第一个不相等的对应元素。如果不相等的一对元素被找到，则通过这对元素计算其结果。或者，如果一个序列正好是另一个序列的前半截，那么较短的那个序列被认为是较小的序列。最终，如果上述情况都没有发生，则这两个序列完全一样，被认为相等。下面是一些例子：

```
>>> cmp((1, 2, 3), (1, 2, 3))      # 相等
0
>>> cmp((1, 2, 3), (1, 2))        # 第一个大，因为第二个是第一个的前一部分
1
>>> cmp((1, 100), (2, 1))        # 第一个小，因为 1<2
-1
>>> cmp((1, 2), (1, 3))          # 第一个小，因为 1==1，然后 2<3
-1
```

如果想根据主键及二级主键对一个对象列表进行排序，可以简单地创建一个元组的列表，其中每个元组都遵循相同的顺序来存储主键、二级主键以及对象本身，这样我们就可以基于条目顺序进行比较。由于元组是按照顺序比较的，所以比较操作可以顺利成章地得到正确的结果。在比较元组时，主键被首先比较，当且仅当主键相等时，二级主键才会继续进行比较。

本章的 DSU 模式的很多例子展示了这种思想在各种需求中的应用。DSU 技术可以用于任意数目的主键。只要愿意，可以给元组增加足够多的键，当然它们的顺序得按照你希望进行的比较顺序进行排列。在 Python 2.4 中，用 sort 的新的 key= 可选参数也可以得到同样效果，有几节内容将会展示这一用法。相比于手工构建一个辅助的元组列表，使用 sort 方法的 key= 参数更加容易、更节省内存，而且速度也更快。

Python 2.4 还为排序提供了其他改进，包括一个方便的快捷方法：内建函数 sorted 可以将任何可迭代对象排序，并且不改动原对象，而是首先将其复制到一个新的列表。Python 2.3（没有那个新的可选的关键字参数，该参数既可用于内建的 sorted 函数也可用于 list.sort），可以编写如下代码来实现相同的功能：

```
def sorted_2_3(iterable):
    alist = list(iterable)
    alist.sort( )
    return alist
```

由于列表复制和列表排序都不是很轻松的操作，而内建的 `sorted` 需要执行这些操作，所以使用内建的 `sorted` 函数不会获得速度上的优势，它的优势就在于方便。有个预先实现的函数在手边，总比每次都要写四行程序来完成同样功能让人心情愉快一些，这是个实用与否的问题。另一方面，一些小函数用得非常广泛和频繁，对原有的内建对象和函数进行扩展变得非常必要。Python 2.4 增加的 `sorted` 和 `reversed` 函数在之前已经被要求了很多年。

自从本书第一版出版之后，Python 的排序的最大的变化是 Python 2.3 采用了新的排序实现。由此产生的明显区别是，很多常用操作的速度变快了，而且新的排序是稳定排序（即如果原列表中的两个被比较的元素相等，那么在完成排序之后它们的相对顺序不变）。这个新的实现是如此成功，以至于进一步提高的空间似乎都不大了，Guido 也被说服并宣布 Python 的 `list.sort` 方法将永远是稳定排序。这个始于 Python 2.4 的保证其实已经在 Python 2.3 中实现了。当然，排序的发展历史不断地提醒我们，更好的方法也许还没被发现。因此，我们也会概要介绍一下 Python 的排序的发展历史。

## Python 排序的简短历史

在早期的 Python 发行版中，`list.sort` 使用平台 C 库提供的 `qsort` 例程。这个排序方法最终被替换的原因有好几个，但主要是由于 `qsort` 的质量在不同的计算机上差异很大。在对一个带有很多相等的值或者完全反序的列表的排序中，一些版本慢得让人无法容忍。一些版本甚至会引发 Core Dump，因为它们是不可重入的。用户定义的 `__cmp__` 函数也可能会调用 `list.sort`，所以一个 `list.sort` 会调用其他的 `list.sort`，这是比较操作的副作用。一些平台的 `qsort` 例程无法处理这种状况。一个用户定义的 `__cmp__` 函数也可能（假如它是恶意的或者疯了）在正在排序的时候修改列表，很多平台的 `qsort` 例程在这种情况下都会引发 Core Dump。

Python 因而发展了自己的快速排序算法。每个发行版都会重写此算法，因为总是能在实际应用中发现慢得无法接受的情况。快速排序真的是一种脆弱而精巧的算法。

在 Python 1.5.2 中，快速排序算法被替换成了抽样排序和折半插入排序的混合体，这个算法超过四年保持不变，直到 Python 2.3 问世。抽样排序可以被看做是快速排序的变种，它使用很大的样本空间来挑选划分元素（partitioning element），也被称为轴心（它对元素的一个大的随机子集递归地抽样排序并挑选它们的中值）。这个变种使得二次方时间复杂度的行为变得几乎不可能，同时也让平均的比较次数非常接近理论上的最小值。

不过，抽样排序是一个复杂的算法，对于小列表而言，它有比较大的管理上的开销。因此，小列表（以及抽样排序划分出来的小片结果）由独立的折半插入排序算法处

理——其实就是一种普通的插入排序，只不过它用二分搜索来决定新元素的归属。很多关于排序的文章说这不值得烦恼，因为他们假设比较两个元素的开销要比在内存中交换元素的开销小，但对于 Python 的排序来说，这个假设不成立。移动一个对象非常容易，因为复制的只不过是一个对象的引用。比较两个对象的开销则比较昂贵，这是因为要通过面向对象机制寻找合适的代码来比较两个对象，同时这部分代码每次都被强制调用。正因为这一点，二分搜索在 Python 的排序中是很成功的应用。

基于这个混合方法，一些特殊情况被特别照顾以利于提速。首先，已经排序或者反序的列表会被检查出来，并以线性的时间复杂度来处理。对于一些应用而言，这种类型的列表很常见。其次，如果一个序列已经大部分完成排序，只有少数位于末尾的元素仍然处于乱序状态，排序工作将由折半插入排序算法接手。这比完全交由抽样排序算法处理要快一些，尤其是一些特殊的应用需要不断地将列表排序，加入一个新元素，然后再排序。然后，抽样排序算法中的一些特别的代码会检查相等且相邻一段元素，并将这些片段标记为已完成部分。

最后，所有这些努力产生了一个极其优异的排序算法，无论是在实际的使用中体现出来的高效，还是针对一些常见的特殊例子所展现的梦幻般的速度，都充分证明了此算法的成功。这个算法包括了大约 500 行非常复杂的 C 代码，和 5.11 节展示的例子可以说有天壤之别。

抽样排序算法已经行之有年，我也曾说过要请能够写出更快的 Python 排序的人吃饭。不过迄今为止，我还是只能一个人去吃饭。我也仍然在继续关注一些文献资料，因为混合抽样排序算法的一些问题仍然让我耿耿于怀。

- 虽然还没有在实际中发现二次方时间复杂度行为的例子，但我知道这样的例子一定是可以设计出来的，因为要设计一个比平均速度慢两到三倍的用例是非常容易的。
- 针对某些极端偏序（partial order）的特例的优化在实际应用中非常有用，但真实数据中经常出现其他类型的偏序，这些类型的偏序也应当被特别处理。事实上，我已经开始相信真正的随机输入顺序在现实生活中根本不可能存在（当然要把用于测试排序算法的时间复杂度的例子排除掉）。
- 如果不增加内存的使用量，现在还没有什么可行的方法来使抽样排序成为稳定排序。
- 为了对一些特例进行优化而使代码变得极其复杂、晦涩及丑陋。

## 当前的排序

很明显归并排序有几个优点，它的最坏结果可保证为  $n \log n$  时间复杂度，而且很容易实现稳定排序。问题是在 Python 中的多次对归并算法的实现产生的只是更慢的结果（比

起抽样排序，归并排序过多地移动了数据）和更高的内存消耗。

很多文献资料——而且是越来越多的，开始关注适应性排序算法（adaptive sorting algorithm），这种算法试图探测不同的输入中的元素顺序。我写了很多这种算法的实现，但它们都比 Python 的抽样排序慢得多，除非例子是被专门设计的。这些算法的理论基础比较复杂，因此难于产生有效的可行算法。后来我读到了一篇文章，该文指出列表的合并自然地揭示了很多类型的偏序，只需简单地注意一下每个输入列表连续“贏”的频率。这个信息很简单但也很具有普遍性。当我意识到可以将它应用在一个自然的归并排序中，而且这种方法能够很好地应对我所知道和关心的各种特例时，我就痴迷地投入到了提高随机数据处理的速度以及减轻内存负担的工作中。

最后，Python 2.3 中的“适应性强的、自然的、稳定的”归并排序成为了一个很大的成功，但同时也是一个工程上的硬骨头——魔鬼藏在细节之中。该算法的实现包括了大约 1 200 行 C 程序，但不像混合抽样排序的代码，这些代码没有一行是为特例准备的，而且大概有一半是在实现一个技术上的技巧，以使得最坏情况下的内存负担可以减轻一半。我对这个算法感到骄傲，但是引言部分已经没有太多篇幅可供我解释细节了。如果你很好奇，我写了一个很长的技术描述文档，可以在 Python 的源码发行包中找到：主目录中——也就是你解压 Python 源码发行包的地方——（比如，Python-2.3.5 或者 Python-2.4）的 Objects/listsort.txt。在下面的列表中，我提供了 Python 2.3 的归并排序可以利用的偏序的例子，“排序完毕”意味着正向或者反向序。

- 输入序列已经是排序完毕状态。
- 输入序列接近于排序完毕状态，但是有一些随机元素处于尾部或中间，或者两处都有。
- 输入序列是两个或多个排序完毕列表的拼接。事实上，在 Python 中最快的归并多个排序完毕的列表的方法就是先将它们连接起来，然后执行 list.sort 即可。
- 输入序列有多个键对应相同的值。比如，对股票交易所的数据库中的美国公司排序，这些公司大多和 NYSE 或者 NASDAQ 联系起来。算法能够利用此特例的原因是：根据对“稳定”的定义，拥有相同的键的记录已经是排序完毕状态！算法能够自然地探测到这一点，无须特意寻找相等键的代码。
- 输入序列是排序完毕状态，但是不小心掉在地板上摔成很多块了；每一块都在随机的位置，而且其中的某些块内部也已经重新洗牌了。虽然这看上去是挺傻的例子，但它仍然能够被算法利用并提升处理性能，由此可见这种方法的通用性。

长话短说，Python 2.3 的提姆排序（timsort，嗯，它毕竟得有个短点的名字吧）是稳定的、健壮的，而且在实际应用中也快得像飞一样，尽可能的选择用它吧！

## 5.1 对字典排序

感谢: Alex Martelli

### 任务

你想对字典排序。这可能意味着需要先根据字典的键排序，然后再让对应值也处于同样的顺序。

### 解决方案

最简单的方法可以通过这样的描述来概括：先将键排序，然后由此选出对应值：

```
def sortedDictValues(adict):
    keys = adict.keys( )
    keys.sort( )
    return [adict[key] for key in keys]
```

### 讨论

排序的概念仅仅适用于那些有顺序的集——换句话说，一个序列。而一个映射，比如字典，是没有顺序的，因此它无法被排序。然而，“我怎么才能将一个字典排序”是 Python 邮件列表中一个很常见的问题，理论上这个问题没有什么意义。在绝大多数情况下，其实际目的是将字典中的键构成的序列排序。

至于实现部分，一些人总是考虑更复杂的方式，但其实解决方案中给出的最简单的方法也是最快的方法（对 Python 来说，这样的情况并不少见）。在 Python 2.3 中，在函数的最后的 `return` 语句中，将列表推导转换成对 `map` 的调用还可以获得一些速度的提升，大约 20%。比如：

```
return map(adict.get, keys)
```

解决方案中的代码在 Python 2.4 下已经比 Python 2.3 要快了，按照上面的方式进行改写也不会获得很大的速度提升。而使用其他方法，比如用 `adict.__getitem__` 来代替 `adict.get`，并不会提供任何性能的提升，反而会引起性能的些微下降，无论是在 Python 2.3 还是 2.4 中。

### 更多资料

5.4 节的方案，根据字典的对应值而不是键进行排序。

## 5.2 不区分大小写对字符串列表排序

感谢: Kevin Altis、Robin Thomas、Guido van Rossum、Martin V. Lewis、Dave Cross

## 任务

你想对一个字符串列表排序，并忽略掉大小写信息。举个例子，你想要小写的 a 排在大写的 B 前面。默认的情况下，字符串比较是大小写敏感的（比如所有的大写字符排在小写字符之前）。

## 解决方案

采用 `decorate-sort-undecorate` (DSU) 用法既快又简单：

```
def case_insensitive_sort(string_list):
    auxiliary_list = [(x.lower( ), x) for x in string_list]      # decorate
    auxiliary_list.sort( )                                         # sort
    return [x[1] for x in auxiliary_list]                         # undecorate
```

Python 2.4 已经提供对 DSU 的原生支持了，因此（假设 `string_list` 的元素都是真正的普通字符串，而不是 Unicode 对象之类），可以用更简短更快的方式：

```
def case_insensitive_sort(string_list):
    return sorted(string_list, key=str.lower)
```

## 讨论

一个很明显的可选方案是编写一个比较函数，并将其传递给 `sort` 方法：

```
def case_insensitive_sort_1(string_list):
    def compare(a, b): return cmp(a.lower( ), b.lower( ))
    string_list.sort(compare)
```

不过，在每次比较中，`lower` 方法都会被调用两次，而对于长度为  $n$  的列表来说，比较的次数与  $n \log(n)$  成正比。

DSU 方法则创建了一个辅助的列表，每个元素都是元组，元组的元素则来自原列表并被当做“键”处理。这个排序是基于键的，因为 Python 的元组的比较是根据条目顺序进行的（比如，它会首先比较元组的第一个元素）。要将一个长度为  $n$  的字符串列表排序，配合 DSU 的使用，`lower` 方法只需要被调用  $n$  次，因而在第一步，`decorate` 阶段，以及最后一步，`undecorate` 阶段节省了很多时间。

DSU 有时也被称为——但这种叫法不是很准确——Schwartzian 变换，这是对 Perl 的一个著名应用的一个不太准确的类比。（如果说相似，DSU 更接近于 Guttman-Rosler 变换，见 [http://www.sysarch.com/perl/sort\\_paper.html](http://www.sysarch.com/perl/sort_paper.html)。）

DSU 是如此重要，因此 Python 2.4 提供了对它的直接支持。可以给列表的 `sort` 方法传递一个可选的命名参数 `key`，而且它可以被调用，作用于列表中的每个元素并获得用于排序的键。如果传递这样的一个参数，排序会在内部使用 DSU。因此，在 Python 2.4 中，`string_list.sort(key = str.lower)` 实际上等价于 `case_insensitive_sort` 函数，只不过 `sort` 方法

会直接作用于原列表（且返回 `None`），而不是返回一个排序完毕的拷贝且不对原列表做任何修改。如果你希望 `case_insensitive_sort` 函数也能够直接作用于原列表，只需要将 `return` 语句修改为对列表本体的赋值：

```
string_list[:] = [x[1] for x in auxiliary_list]
```

反过来，在 Python 2.4 中，如果你希望获得一个排序完毕的拷贝，且让原列表保持不变，可以使用新的内建的 `sorted` 函数。比如，在 Python 2.4 中：

```
for s in sorted(string_list, key=str.lower): print s
```

上述代码打印列表中的每一个字符串，这些字符串根据大小写无关的规则进行排序，而且不会影响到 `string_list` 本身。

在 Python 2.4 的解决方案中，将 `str.lower` 作为 `key` 参数限制了你以特定的方式将字符串排序（不包括 Unicode 对象）。如果你知道你正在排序的是 Unicode 对象列表，可以使用 `key = unicode.lower`。如果你希望函数能够同时适用于普通字符串和 Unicode 对象，可以 `import string` 并使用 `key = string.lower`；另外，也可以使用 `key = lambda s: s.lower()`。如果需要对字符串列表进行大小写无关的排序，可能也需要用大小写无关的字符串作为键的字典和集合，需要列表对 `index` 和 `count` 方法表现出与大小写无关的行为方式，需要在各种搜索匹配的任务中忽略掉大小写，等等。如果这是你的需求，那么真正需要的是 `str` 的一个子类型，从而在比较和哈希的时候忽略大小写——相比于实现各种容器和函数来满足这些需求，这是解决这类问题的最好的方法。参考 1.24 节内容，可以看到如何实现这样一种子类型。

## 更多资料

Python Frequently Asked Questions, <http://www.python.org/cgi-bin/faqw.py?req=show&file=faq04.051.htm>; 5.3 节; Python 2.4 的 *Library Reference* 中关于 `sorted` 内建函数, `sort` 和 `sorted` 的 `key` 参数; 1.24 节。

## 5.3 根据对象的属性将对象列表排序

感谢: Yakov Markovitch、Nick Perkins

### 任务

需要根据各个对象的某个属性来完成对整个对象列表的排序。

### 解决方案

DSU 方法仍然一如既往地有效：

```
def sort_by_attr(seq, attr):
    intermed = [ (getattr(x, attr), i, x) for i, x in enumerate(seq) ]
```

```
intermed.sort( )
    return [ x[-1] for x in intermed ]
def sort_by_attr_inplace(lst, attr):
    lst[:] = sort_by_attr(lst, attr)
```

由于 Python 2.4 的对 DSU 的原生支持，代码可以写得更短、跑得更快：

```
import operator
def sort_by_attr(seq, attr):
    return sorted(seq, key=operator.attrgetter(attr))
def sort_by_attr_inplace(lst, attr):
    lst.sort(key=operator.attrgetter(attr))
```

## 讨论

根据对象属性将对象排序的最佳方法仍然是 DSU，如同前面 5.2 节所介绍的。在 Python 2.3 和 2.4 中，DSU 不再像过去那样，是用来确保排序的稳定性的方法了（因为从 Python 2.3 开始，排序将一直保持稳定），但 DSU 的速度优势仍然如故。

排序，针对最普遍的用例，采用最好的算法，其时间复杂度是  $O(n \log n)$ （如同常见的数学公式一样，这里  $n$  和  $\log n$  之间是相乘的关系）。通过使用 Python 原生的比较操作（也是最快的），DSU 的速度大多来自于对  $O(n \log n)$  部分的加速， $O(n \log n)$  决定着长度为  $n$  的序列的排序时间。在预备的 decoration 阶段，即准备辅助的元组列表阶段，以及成功后的 undecoration 阶段，即在完成排序后的中间结果的列表的元组中取出需要的元素的阶段，时间复杂度仅仅是  $O(n)$ 。因此，如果  $n$  非常大，这两个阶段的一些低效的操作并不会造成很大的影响，在实际应用中，它们也确实影响甚微。

## O()记法

当我们需要思考性能问题时，采用的最有效的方法就是众所周知的大 O 分析法以及记法（O 表示的是“order”）。可以在 [http://en.wikipedia.org/wiki/Big\\_O\\_notation](http://en.wikipedia.org/wiki/Big_O_notation) 看到非常详细的解释，但这里我们只是给出了一个概要。

如果我们考虑对尺度为  $N$  的输入数据采用某个算法，则运行时间是可以描述的，例如针对足够大的  $N$ （具有很大输入数据的应用通常会最关心性能问题），时间和  $N$  的函数成正比。对于这种表述，我们记录为相应的符号  $O(N)$ （运行时间正比于  $N$ ：处理 2 倍的数据需要 2 倍的时间，10 倍的数据则需要 10 倍的时间，以此类推；也被称为线性时间复杂度）， $O(N \text{ squared})$ （运行时间正比于  $N$  的平方：处理 2 倍的数据，需要花费 4 倍的时间，10 倍的数据，则需要 100 倍的时间；这也被称为二次方时间复杂度），等等。另一个常见的情况是  $O(N \log N)$ ，它比  $O(N \text{ squared})$  快但比  $O(N)$  慢。

常数比率通常是被忽略的（至少在理论分析中是这样），因为它常常依赖于一些非算法的因素，如计算机的时钟频率，而不是算法本身。比如你买了一台计算机，它比你的老计算机快两倍，所有的事情处理起来都只需要一半时间，但这并不会改变不同算法之间的差异。

本节的方案是，给 `intermed` 列表的每一个元素所在的元组中加入了一个索引 `i`，位于对应的 `x` 之前（`x` 是 `seq` 的第 `i` 个元素）。这个举措保证了 `seq` 中任意两个子项都不会被直接用于比较，即使对同一个属性名 `attr` 它们都具有相同的值。在这种情况下，它们的索引仍然会保持不同，因此基于 Python 的根据条目顺序比较（lexicographical comparison）的规则，元组的最后一个元素（即 `seq` 的元素）无须被用于比较。避免对象的比较将极大地提高性能。举个例子，我们可以根据 `real` 属性对一个复数列表进行排序。如果直接比较两个复数，我们会引发一个异常，因为复数之间并没有定义顺序。但是正如我们前面提到的，这样的情况永远不会发生，因此排序将会正确地进行下去。

5.2 节曾经提到过，Python 2.4 直接支持 DSU。可以传递一个可选的关键字参数 `key` 给 `sort`，这样每个元素都可以用它来获取排序的键。标准库模块 `operator` 有两个新函数，`attrgetter` 和 `itemgetter`，它们被用来返回适用的可调用体。在 Python 2.4 中，针对这个问题的解决方案就变成了：

```
import operator
seq.sort(key=operator.attrgetter(attr))
```

这个片段执行的排序是直接应用于原列表的，因此速度快得惊人——在我的计算机上，比解决方案给出的第一个 Python 2.3 的函数快 3 倍。如果需要的是一个排序后的拷贝，而不想影响 `seq`，可以使用 Python 2.4 新的内建的 `sorted` 函数：

```
sorted_copy = sorted(seq, key=operator.attrgetter(attr))
```

不过它没有直接应用于原列表的排序快，这个代码片段比解决方案的第一个函数快 2.5 倍。另外，Python 2.4 保证了，如果传入了可选的 `key` 命名参数，列表的元素永远不会被直接比较，因此无须其他的安全保障。而且，排序的稳定性也是有保证的。

## 更多资料

5.2 节；Python 2.4 的 *Library Reference* 文档中有关 `sorted` 内建函数，`operator` 模块的 `attrgetter` 和 `itemgetter` 函数，以及 `sort` 和 `sorted` 的 `key` 参数。

## 5.4 根据对应值将键或索引排序

感谢：John Jensen、Fred Bremmer、Nick Coghlan

### 任务

需要统计不同元素出现的次数，并且根据它们的出现次数安排它们的顺序——比如，你想制作一个柱状图。

### 解决方案

柱状图，如果不考虑它在图形图像上的含义，实际上是基于各种不同元素（用 Python

的列表或字典很容易处理) 出现的次数, 根据对应值将键或索引排序。下面是 `dict` 的一个子类, 它为了这种应用加入了两个方法:

```
class hist(dict):
    def add(self, item, increment=1):
        ''' 为 item 的条目增加计数 '''
        self[item] = increment + self.get(item, 0)
    def counts(self, reverse=False):
        ''' 返回根据对应值排序的键的列表 '''
        aux = [ (self[k], k) for k in self ]
        aux.sort()
        if reverse: aux.reverse()
        return [k for v, k in aux]
```

如果想将元素的统计结果放到一个列表中, 做法也非常类似:

```
class hist1(list):
    def __init__(self, n):
        ''' 初始化列表, 统计 n 个不同项的出现 '''
        list.__init__(self, n*[0])
    def add(self, item, increment=1):
        ''' 为 item 的条目增加计数 '''
        self[item] += increment
    def counts(self, reverse=False):
        ''' 返回根据对应值排序的索引的列表 '''
        aux = [ (v, k) for k, v in enumerate(self) ]
        aux.sort()
        if reverse: aux.reverse()
        return [k for v, k in aux]
```

## 讨论

`hist` 的 `add` 方法展示了 Python 用于统计任意 (可哈希的) 元素的常用方法, 并使用 `dict` 来记录次数。在类 `hist1` 中, 在一个普通的列表的基础上, 我们采用了不同的方法, 并在 `__init__` 中将所有的次数都设置成 0, 因而 `add` 方法就变得更简单了。

`counts` 方法生成了一个键或者索引的列表, 并且根据对应值进行了排序。这两个类针对的问题很类似, 因此解决方式也几乎完全一样, 都使用了前面 5.2 节和 5.3 节展示过的 DSU。如果我们想要在自己的程序中使用这两个类, 由于它们的相似性, 我们应该进行代码重构, 从中间分离出共性并置入一个单独的辅助函数 `_sorted_keys`:

```
def _sorted_keys(container, keys, reverse):
    ''' 返回 keys 的列表, 根据 container 中的对应值排序 '''
    aux = [ (container[k], k) for k in keys ]
    aux.sort()
    if reverse: aux.reverse()
    return [k for v, k in aux]
```

然后实现各个类的 `counts` 方法, 其实就是对 `_sorted_keys` 函数进行一层很薄的封装:

```

class hist(dict):
    ...
    def counts(self, reverse=False):
        return _sorted_keys(self, self, reverse)
class hist1(list):
    ...
    def counts(self, reverse=False):
        return _sorted_keys(self, xrange(len(self)), reverse)

```

DSU 在 Python 2.4 中非常重要，前面 5.2 节和 5.3 节已经介绍过了，列表的 sort 方法和新的内建的 sorted 函数提供了一个快速的、原生的 DSU 实现。因此，在 Python 2.4 中，\_sorted\_keys 还可以变得更简单快速：

```

def _sorted_keys(container, keys, reverse):
    return sorted(keys, key=container.__getitem__, reverse=reverse)

```

被绑定的 container.\_\_getitem\_\_ 方法和 Python 2.3 中实现的获取索引的操作 container[k] 所做的事情完全一样，但是对于我们正在排序的序列而言，它是一个可调用体，可以应用于序列中的每个元素，即命名的键，因此我们可以将它传递给内建 sorted 函数的作为 key 关键字参数的值。Python 2.4 还提供了一个简单直接的方法来获取字典元素根据值排序后的列表：

```

from operator import itemgetter
def dict_items_sorted_by_value(d, reverse=False):
    return sorted(d.iteritems(), key=itemgetter(1), reverse=reverse)

```

如果想排序一个元素为子容器的容器，Python 2.4 新出现的高级函数 operator.itemgetter 是一个很方便的提供 key 参数的方法，它可以针对每个子容器的特定元素建立键。这正是我们想要的，因为字典的条目实际上就是一个键和值构成的对（两元素的元组）的序列，所谓根据对应值排序，就是根据每个元组的第二个元素进行排序。

回到本节的主题，下面是本节解决方案中的 hist 类的一个使用示例：

```

sentence = """ Hello there this is a test. Hello there this was a test,
              but now it is not. """
words = sentence.split()
c = hist()
for word in words: c.add(word)
print "Ascending count:"
print c.counts()
print "Descending count:"
print c.counts(reverse=True)

```

上述代码片段产生了如下的输出：

```

Ascending count:
[(1, 'but'), (1, 'it'), (1, 'not.'), (1, 'now'), (1, 'test,'), (1, 'test.'), (1, 'was'), (2, 'Hello'), (2, 'a'), (2, 'is'), (2, 'there'), (2, 'this')]
Descending count:

```

```
[(2, 'this'), (2, 'there'), (2, 'is'), (2, 'a'), (2, 'Hello'), (1, 'was'),  
(1, 'test.'), (1, 'test,'), (1, 'now'), (1, 'not.'), (1, 'it'), (1, 'but')]
```

## 更多资料

*Language Reference* 的“特殊方法名字”一节以及 *Python in a Nutshell* 中 OOP 章节，特殊的 `__getitem__` 方法；Library Reference 中关于 Python 2.4 的内建 `sorted` 函数以及 `sort` 和 `sorted` 的 `key=` 参数。

## 5.5 根据内嵌的数字将字符串排序

感谢： Sébastien Keim、Chui Tey、Alex Martelli

### 任务

你想将一个字符串列表进行排序，这些字符串都含有数字的子串（比如一系列邮寄地址）。举个例子，“foo2.txt”应该出现在“foo10.txt”之前。然而，Python 默认的字符串比较是基于字母顺序的，所以默认情况下，“foo10.txt”会在“foo2.txt”之前。

### 解决方案

需要先将每个字符串切割开，形成数字和非数字的序列，然后将将每个序列中的数字转化成一个数。这会产生一个数的列表，可以用来做排序时比较的键，可以对这个排序应用 DSU——写两个函数即可，做起来很快捷：

```
import re  
re_digits = re.compile(r'(\d+)')  
def embedded_numbers(s):  
    pieces = re_digits.split(s)                      # 切成数字与非数字  
    pieces[1::2] = map(int, pieces[1::2])           # 将数字部分转成整数  
    return pieces  
def sort_strings_with_embedded_numbers(alist):  
    aux = [ (embedded_numbers(s), s) for s in alist ]  
    aux.sort()  
    return [ s for _, s in aux ]                     # 惯例：__意味着“忽略”
```

在 Python 2.4 中，用相同的 `embedded_number` 函数，加上 DSU 的原生支持，代码变成：

```
def sort_strings_with_embedded_numbers(alist):  
    return sorted(alist, key=embedded_numbers)
```

### 讨论

假设有一个未排序的文件名的列表，比如：

```
files = 'file3.txt file11.txt file7.txt file4.txt file15.txt'.split()
```

如果只是排序并打印列表，比如在 Python 2.4 中用 `print ''.join(sorted(files))` 这样的代码，你的输出会是这样：`file11.txt file15.txt file3.txt file4.txt file7.txt`，因为默认情况下，字符串是根据字母顺序排序的（或者换句话说，排序顺序是由条目顺序指定的）。Python 猜不到你的真实意图其实是希望让它以另外的方式处理那些含有数字的子串，所以必须准确地告诉 Python 你想要什么，解决方案中的代码主要所做的工作其实就是这么一件事。

基于解决方案的代码，也能获得一个更好看的结果：

```
print ' '.join(sort_strings_with_embedded_numbers(files))
```

现在输出变成了 `file3.txt file4.txt file7.txt file11.txt file15.txt`，这应该正好就是需要的顺序。

这个实现基于 DSU。如果想在 Python 2.3 中达到同样目的，需要手工制作 DSU，但如果您的代码只需要在 Python 2.4 中运行，直接使用原生内建的 DSU 即可。我们传递了一个叫做 `key` 的参数（一个函数，该函数对每个元素都会调用一次以获取正确的比较键来用于排序）给新的内建函数 `sorted`。

本节解决方案中的 `embedded_numbers` 函数正是用来为每个元素获取正确的比较键的方法：一个非数字子串交替出现的列表，`int` 获取了每个数字子串。`re_digits.split(s)` 给了我们一个交替出现的数字子串和非数字子串的列表（数字子串拥有偶数索引），然后我们使用了内建的 `map` 和 `int`（采用了扩展切片的方式获得并设置了偶数索引号的元素）来将数字序列转化成整数。现在，对这个混合类型的列表进行的条目顺序比较就可以产生正确的结果了。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 文档中关于扩展切片以及 `re` 模块部分；Python 2.4 *Library Reference* 中的内建函数 `sorted` 以及 `sort` 和 `sorted` 的 `key` 参数；5.3 节和 5.2 节。

## 5.6 以随机顺序处理列表的元素

感谢：Iuri Wickert、Duncan Grisby、T. Warner、Steve Holden、Alex Martelli

### 任务

你想以随机的顺序处理一个很长的列表。

### 解决方案

一如既往的，在 Python 中最简单的方法常常是最好的。如果我们允许修改输入列表中

的元素的顺序，那么下面的函数就是最简单和最快的：

```
def process_all_in_random_order(data, process):
    # 首先，将整个列表置于随机顺序
    random.shuffle(data)
    # 然后，根据正常顺序访问
    for elem in data: process(elem)
```

如果我们需要保证输入列表不变，或者输入列表可能是其他可迭代对象而不是列表，可以在函数主体开头加上一条赋值语句 `data = list(data)`。

## 讨论

虽然过度关心速度常常是个错误，但我们也不能忽略不同算法的性能。假设我们必须以随机顺序处理一个不重复的长列表的元素。第一个想法可能会是这样：我们可以反复地、随机地挑出元素（通过 `random.choice` 函数），并将原列表中被挑选的元素删除，以避免重复挑选：

```
import random
def process_random_removing(data, process):
    while data:
        elem = random.choice(data)
        data.remove(elem)
        process(elem)
```

然而，这个函数慢得可怕，即使输入列表只有几百个元素。每个 `data.remove` 调用都会线性地搜索整个列表以获取要删除的元素。由于第  $n$  步的时间消耗是  $O(n)$ ，因此整个处理过程的消耗时间是  $O(n^2)$ ，正比于列表长度的平方（而且要乘上一个很大的常数）。

对第一个想法的一点提高是将注意力集中在获取随机索引上，并使用列表的 `pop` 方法来同时获取和删除元素，这种更底层的方式避免了较大的消耗，尤其是在某些情况下，比如要被挑选的元素位于列表的最后，或者使用的压根不是列表，而是字典或集合。若我们面对的是字典或集合，一条思路是寄希望于使用 `dict` 的 `popitem` 方法（或者 `sets.Set` 或 Python 2.4 内建类型 `set` 的等价的 `pop` 方法），看上去这个函数好像被设计为随机选择一个元素并删除之，但是，小心上当。

`dict.popitem` 的文档指出，它返回并删除字典中的任意一个元素，但这和真正的随机元素还差得很远。看看这个：

```
>>> d=dict(enumerate('ciao'))
>>> while d: print d.popitem()
```

你可能会很吃惊，在大多数的 Python 实现中，这个代码片段都将以看上去不太随机的方式打印 `d` 的元素，通常是`(0, 'c')`，然后`(1, 'i')`，等等。一句话，如果需要 Python 中的伪随机行为，需要的是标准库的 `random.popitem` 模块。

如果你考虑使用字典而不是列表，那么你肯定在“Python 式思维”的路上又前进了一步，虽然字典并不会针对这个特定问题提供什么性能优势。但相比于选择正确的数据结构，更具有 Python 风格的方式是：总是利用标准库。Python 标准库是个庞大、丰富的库，塞满了各种有用的、强健的、快速的函数和类，可满足各种应用的需求。在这个前提下，最关键的一点是要意识到，想要以随机的顺序访问序列，最简单的方法是首先将序列转化成随机的顺序（也被称为对序列洗牌，是对扑克洗牌的类比），然后再线性的访问洗完牌的序列即可。`random.shuffle` 函数就可以执行洗牌操作，本节解决方案正是利用了这个函数。

实际性能总是需要测试，而不是猜测出来的，那也正是标准库模块 `timeit` 存在的原因。使用一个空的 `process` 函数和一个长度为 1 000 的列表作为 `data`, `process_all_in_random_order` 能比 `process_random_removing` 快大约 10 倍；对于长度为 2 000 的列表，这个比例变成了 20。如果提升仅仅是 25%，或者是一个常数因子 2，那么通常这个性能差异是可以忽略的，因为这不会对你的整体应用产生什么性能影响，但如果算法慢了 10 或 20 倍，情况就不同了。这种可怕的低效会成为整个程序的瓶颈。当我们谈到  $O(n^2)$  和  $O(n)$  的行为对比时，问题的严重性根本无法忽视：对于这两种大  $O$  的行为，随着输入数据的增长，它们消耗时间的差异可以无限地递增下去。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中的 random 和 timeit 模块。

## 5.7 在增加元素时保持序列的顺序

感谢：John Nielsen

### 任务

你需要维护一个序列，这个序列不断地有新元素加入，但始终处于排序完毕的状态，这样你可以在任何需要的时候检查或者删除当前序列中最小的元素。

### 解决方案

假设有一个未排序的列表，比如：

```
the_list = [903, 10, 35, 69, 933, 485, 519, 379, 102, 402, 883, 1]
```

可以调用 `the_list.sort()` 将列表排序，然后用 `result = the_list.pop(0)` 来获得和删除最小的元素。但是，每当加入一个元素（比如 `the_list.append(0)`），都需要再次调用 `the_list.sort` 来排序。

可以使用 Python 标准库的 `heapq` 模块：

```
import heapq
heapq.heapify(the_list)
```

现在列表并不一定完成了排序，但是它却满足堆的特性（若所有涉及的索引都是有效的，则 `the_list[i] <= the_list[2*i + 1]` 且 `the_list[i] <= the_list[2*i+2]`），所以，`the_list[0]`就是最小的元素。为了保持堆特性的有效性，我们使用 `result = heapq.heappop(the_list)`来获取并删除最小的元素，用 `heapq.heappush(the_list, newitem)`来加入新的元素。如果需要同时做这两件事：加入一个新元素并删除之前的最小的元素，可以用 `result=heapq.heapreplace(the_list, newitem)`。

## 讨论

当需要以一种有序的方式获取数据时（每次都选择你手中现有的最小元素），可以选择在获取数据时付出运行时代价，或者在加入数据时付出代价。一种方式是将数据放入列表并对列表排序。这样，可以很容易地让你的数据按照顺序从小到大排列。然而，你不得不每次在加入新数据时调用 `sort`，以确保每次在增加新元素之后仍能够获取最小的元素。Python 列表的 `sort` 实现采用了一种不太有名的自然的归并排序，它的排序开销已经被尽力地压缩了，但仍然难以让人接受：每次添加（和排序）及每次获取（以及删除，通过 `pop`）的时间，与当前列表中元素的数目成正比 ( $O(N)$ )，准确地说)。

另一种方案是使用一种叫做堆的组织数据的结构，这是一种简洁的二叉树，它能确保父节点总是比子节点小。在 Python 中维护一个堆的最好方式是使用列表，并用库模块 `heapq` 来管理此列表，如同本节解决方案所示的那样。这个列表无须完成排序，但你却能够确保每次你调用 `heappop` 从列表中获取元素时，总是得到当前最小的元素，然后所有节点会被调整，以确保堆特性仍然有效。每次通过 `heappush` 添加元素，或者通过 `heappop` 删除元素，它们所花费的时间都正比于当前列表长度的对数 ( $O(\log N)$ )，准确地说)。只需要付出一点点代价（从总体来说，代价也非常小）。

举例来说，很适合使用堆方式的场合是这样的：假设有一个很长的队列，并且周期性地有新数据到达，你总是希望能够从队列中获取最重要的元素，而无须不断地重新排序或者在整个队列中搜索。这个概念叫做优先级队列，而堆正是最适合实现它的数据结构。注意，本质上，`heapq` 模块在每次调用 `heappop` 时向你提供最小的元素，因此需要安排你的元素的优先级值，以反映出元素的这个特点。举个例子，假设你每次收到数据都付出一个价钱，而任何时候最重要的元素都是队列中价钱最高的那个；另外，对于价钱相同的元素，先到达的要重要一些。下面是一个创建“优先级队列”的类，我们遵循上面提到的要求并使用了 `heapq` 模块的函数：

```
class prioq(object):
    def __init__(self):
        self.q = []
        self.i = 0
    def push(self, item, cost):
```

```
    heapq.heappush(self.q, (-cost, self.i, item))
    self.i += 1
def pop(self):
    return heapq.heappop(self.q)[-1]
```

这段代码的意图是将价钱设置为负，作为元组的第一个元素，并将整个元组压入堆中，这样更高的出价会产生更小的元组（基于 Python 的自然比较方式）；在价钱之后，我们放置了一个递增的索引，这样，当元素拥有相同的价钱时，先到达的元素将会处于更小的元组中。

在 Python 2.4 中，`heapq` 模块又被重新实现和进一步优化了，见 5.8 节中更多有关 `heapq` 的信息。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中 `heapq` 模块的文档；Python 源码的 `heapq.py` 中包含了有关堆的一些非常有趣的讨论；5.8 节中更多的关于 `heapq` 的信息；19.14 节中使用 `heapq` 对完成排序的多个序列进行合并。

## 5.8 获取序列中最小的几个元素

感谢： Matteo Dell'Amico、 Raymond Hettinger、 George Yoshida、 Daniel Harding

### 任务

你需要从一个给定的序列中获取一些最小的元素。可以将序列排序，然后使用 `seq[:n]`，但还有没有更好的办法呢？

### 解决方案

如果需要的元素数目 `n` 远小于序列的长度，也许还能做得更好。`sort` 是很快的，但它的时间复杂度仍然是  $O(n \log n)$ ，但如果 `n` 很小，我们获取前 `n` 个最小元素的时间是  $O(n)$ 。下面给出一个简单可行的生成器，在 Python 2.3 和 2.4 中都同样有效：

```
import heapq
def isorted(data):
    data = list(data)
    heapq.heapify(data)
    while data:
        yield heapq.heappop(data)
```

在 Python 2.4 中，如果事先知道 `n`，还有更简单和更快的方法从 `data` 中获取前 `n` 个最小的元素：

```
import heapq
def smallest(n, data):
    return heapq.nsmallest(n, data)
```

## 讨论

`data` 可能是任何有边界的可迭代对象，解决方案中的 `isorted` 函数通过调用 `list` 来确保它是序列。也可以删掉 `data = list(data)` 这一行，假如下列条件满足的话：你知道 `data` 是一个序列，你并不关心生成器是否重新排列了 `data` 的元素，而且需要在获取的同时从 `data` 中删除元素。

- 如同 5.7 节所示，Python 标准库提供了 `heapq` 模块，它支持人们熟知的数据结构——堆。解决方案中的 `isorted` 先创建了一个堆（通过 `heap.heapify`），然后在每一步获取元素的时候（通过 `heap.heappop`），生成并删除堆中最小的元素。

在 Python 2.4 中，`heapq` 模块引入了两个新函数。`heapq.nlargest(n, data)` 返回的是一个长度为 `n` 的 `data` 中最大的元素的列表，`heapq.nsmallest(n, data)` 则返回一个包含前 `n` 个最小元素的列表。这些函数并不要求 `data` 满足堆的条件，它们甚至不要求 `data` 是一个列表——任何有边界的、元素是可以比较的可迭代对象都适用。解决方案中的函数 `smallest` 除了调用 `heapq.smallest`，其实什么也没干。

关于速度，我们总是应该进行实际测量，猜测不同代码片段的相对运行速度是毫无意义的行为。因此，当我们只是循环获取前几个（最小）元素的时候，`isorted` 的性能和 Python 2.4 内建的 `sorted` 函数的性能比起来究竟怎么样呢？为了进行测试，我写了一个 `top10` 函数，它可以调用这两种方法，然后我也为 Python 2.3 实现了一个 `sorted` 函数，因为在 Python 2.3 中此函数并未得到原生的支持：

```
try:
    sorted
except:
    def sorted(data):
        data = list(data)
        data.sort()
        return data
import itertools
def top10(data, howtosort):
    return list(itertools.islice(howtosort(data), 10))
```

在我的计算机上，在 Python 2.4 中处理一个洗过牌的 1 000 个整数的列表，`top10` 调用 `isorted` 耗时 260μs，但采用内建的 `sorted` 则耗时 850μs。而 Python 2.3 甚至还要慢得多：`isorted` 耗时 12ms，`sorted` 耗时 2.7ms。换句话说，Python 2.3 的 `sorted` 比 Python 2.4 的 `sorted` 要慢 3 倍，但在 `isorted` 上则要慢 50 倍。需要记住这个重要的经验：当需要进行优化时，首先进行测量。不应该仅仅根据基本原理选择优化的方式，因为真实的性能数据变化多端，即使在两个兼容的发行版本之间也会有很大的差异。第二个经验是：如果真的很关心性能，那赶紧转移到 Python 2.4 吧。和 Python 2.3 相比，Python 2.4 经过了极大的优化和加速，特别是在搜索和排序的方面。

如果确定你的代码只需要支持 Python 2.4，那么，如同本节解决方案所建议的那样，应

当使用 `heapq` 的新函数 `nsmallest`。它速度快、使用简便，远胜于自己编写代码。举个例子，实现 Python 2.4 中的 `top10`，你只需要：

```
import heapq
def top10(data):
    return heapq.nsmallest(10, data)
```

比起前面展示的那个基于 `isorted` 的 `top10`，对同样的洗过牌的 1 000 个整数的列表进行处理，消耗的时间还可以削减一半。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中 `list` 类型的 `sort` 方法，以及 `heapq` 和 `timeit` 模块；第 19 章中关于 Python 的迭代的内容；*Python in a Nutshell* 中有关优化的章节；Python 源码中的 `heap.py` 所包含的关于堆的有趣的讨论；5.7 节关于 `heapq` 的介绍。

## 5.9 在排序完毕的序列中寻找元素

感谢：Noah Spurrier

### 任务

你需要寻找序列中的一系列元素。

### 解决方案

如果列表 `L` 已经是排序完毕的状态，则 Python 标准库提供的 `bisect` 模块可以很容易地检查出元素 `x` 是否在 `L` 中：

```
import bisect
x_insert_point = bisect.bisect_right(L, x)
x_is_present = L[x_insert_point-1:x_insert_point] == [x]
```

### 讨论

对 Python 来说，在列表 `L` 中寻找一个元素 `x` 是很简单的任务：要检查元素是否存在，`if x in L`；要知道 `x` 的确切位置，`L.index(x)`。然而，`L` 的长度为 `n`，这些操作所花费的时间与 `n` 成正比，因为它们只是循环地检查每一个元素，看看是否与 `x` 相等。其实，如果 `L` 是排过序的，我们还可以做得更好。

在完成了排序的序列中寻找元素的最经典算法就是二分搜索，因为每一步它都将查找的范围减半——一般情况下它只需要  $\log_2 n$  步即可完成搜索。但当需要多次查找某些元素时，这个问题就很值得仔细探讨了，通过使用一些方法，可以为多次搜索尽量少付出一些开销。在调用 `L.sort()` 之后，一旦你决定用二分搜索在 `L` 中查找 `x`，应该马上想到 Python 标准库的 `bisect` 模块。

具体地说，我们需要 `bisect.bisect_right` 函数来保持原列表的排序状态，它将返回一个索引，指示出我们要插入的元素的位置，而且它也不会修改列表；如果列表中已经存在相等的元素了，`bisect_right` 将返回拥有相同值的元素右边邻接的索引。因此，在调用 `bisect.bisect_right(L, x)` 获得了“插入点”之后，只需立刻检查插入点之前的位置，看看是否已经有一个等于 `x` 的元素存在了。

解决方案中计算 `x_is_present` 的方式可能不是那么直观。如果知道 `L` 不是空列表，我们还能写得更简单、更直观：

```
x_is_present = L[x_insert_point-1] == x
```

不过，这个更简单的方式在对空列表进行索引操作的时候会引发异常。当切片的边界无效时，切片操作比索引操作更加“不严谨”，因为它只是生成了一个空的切片，没有引发任何异常。一般来说，当 `i` 是 `somelist` 的有效索引时，`somelist[i:i+1]` 是和 `[somelist[i]]` 一样的单元素列表，但当索引操作引发 `IndexError` 异常时，它却是一个空的列表 `[]`。对 `x_is_present` 的计算充分利用了这种重要的特性，避免了处理异常，同时也能以统一的方式对 `L` 处理空和非空的情况。另一个可选的方式是：

```
x_is_present = L and L[x_insert_point-1] == x
```

这种方式利用了 `and` 的短路的行为模式来保护索引操作，避免了使用切片操作。

如果元素是可哈希的（意味着元素可以被用作 `dict` 的键），如 5.12 节的做法一样，使用一个辅助的 `dict` 也是一个可行的方法。不过，若元素是可比较的（`comparable`，若不可比较，排序是不可能实行的）且不可哈希的（因此字典无法将它们当键用），本节的这个基于已经排序的列表的方法就可能是唯一可行的方案了。

如果列表已经排序完毕，而且需要查找的元素数目不是非常多，那么大多数情况下，用 `bisect` 要比构建一个辅助字典快，因为在构建字典上的时间投资无法被大量查找分摊。尤其是在 Python 2.4 中，`bisect` 已经被高度优化，比在 Python 2.3 中的对应版本要快得多。比如，在我的计算机上，用 `bisect.bisect_right` 在含有 10 000 个元素的列表中查找一个元素，Python 2.4 要比 Python 2.3 快约 4 倍。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中的 `bisect` 模块；5.12 节。

## 5.10 选取序列中最小的第 $n$ 个元素

感谢：Raymond Hettinger、David Eppstein、Shane Holloway、Chris Perkins

### 任务

需要根据排名顺序从序列中获得第  $n$  个元素（比如，中间的元素，也被称为中值）。如

果序列是已经排序的状态，应该使用 `seq[n]`，但如果序列还未被排序，那么除了先对整个序列进行排序之外，还有没有更好的方法？

## 解决方案

如果序列很长，洗牌洗得很充分，而且元素之间的比较开销也大，那么也许还能找到更好的方式。排序的确很快，但不管怎样，它（一个长度为  $n$  的充分洗牌的序列）的时间复杂度仍然是  $O(n \log n)$ ，而时间复杂度为  $O(n)$  的取得第  $n$  个最小元素的算法也的确是存在的。下面我们给出一个函数来实现此算法：

```
import random
def select(data, n):
    """ 寻找第 n 个元素 (最小的元素是第 0 个). """
    # 创建一个新列表，处理小于 0 的索引，检查索引的有效性
    data = list(data)
    if n < 0:
        n += len(data)
    if not 0 <= n < len(data):
        raise ValueError, "can't get rank %d out of %d" % (n, len(data))
    # 主循环，看上去类似于快速排序但不需要递归
    while True:
        pivot = random.choice(data)
        pcount = 0
        under, over = [ ], [ ]
        uappend, oappend = under.append, over.append
        for elem in data:
            if elem < pivot:
                uappend(elem)
            elif elem > pivot:
                oappend(elem)
            else:
                pcount += 1
        numunder = len(under)
        if n < numunder:
            data = under
        elif n < numunder + pcount:
            return pivot
        else:
            data = over
            n -= numunder + pcount
```

## 讨论

本节解决方案也可用于重复的元素。举个例子，列表[1, 1, 1, 2, 3]的中值是 1，因为它是将这 5 个元素按顺序排列之后的第 3 个。如果由于某些特别的原因，你不想考虑重复，而需要缩减这个列表，使得所有元素都是唯一的（比如，通过 18.1 节提供的方法），可以完成这一步骤之后再回到本节的问题。

输入参数 `data` 可以是任意有边界的可迭代对象。首先我们对它调用 `list` 以确保得到可迭代的对象，然后进入持续循环过程。在循环的每一步中，首先随机选出一个轴心元素；以轴心元素为基准，将列表切片成两个部分，一个部分“高于”轴心，一个部分“低于”轴心；然后继续在下一轮循环中对列表的这两个部分中的一个进行深入处理，因为我们可以判断第 `n` 个元素处于哪一个部分，所以另外一个部分就可以丢弃了。这个算法的思想很接近经典的快速排序算法（只不过快速排序无法丢弃任何部分，它必须用递归的方法，或者用一个栈来替换递归，以确保对每个部分都进行了处理）。

随机选择轴心使得这个算法对于任意顺序的数据都适用（但不同于原生的快速排序，某些顺序的数据将极大地影响它的速度）；这个实现花费大约  $\log_2 N$  时间用于调用 `random.choice`。另一个值得注意的是算法统计了选出轴心元素的次数，这是为了在一些特殊情况下仍能够保证较好的性能，比如 `data` 中可能含有大量的重复数据。

将局部变量列表 `under` 和 `over` 的被绑定方法 `append` 抽取出来，看起来没什么意义，而且还增加了一点小小的复杂性，但实际上，这是 Python 中一个非常重要的优化技术。为了保持编译器的简单、直接、可预期性以及健壮性，Python 不会将一些恒定的计算移出循环，它也不会“缓存”对方法的查询结果。如果你在内层循环调用 `under.append` 和 `over.append`，每一轮都会付出开销和代价。如果想把某些事情一次性做好，那么需要自己动手完成。当你考虑优化问题时，你总是应该对比优化前和优化后的效率，以确保优化真正起到了作用。根据我的测试，对于获取 `range(10 000)` 的第 5 000 个元素这样的任务，去掉优化部分之后，性能下降了 50%。虽然增加一点小小的复杂性，但仍然是值得的，毕竟那是 50% 的性能差异。

关于优化的一个自然的想法是，在循环中调用 `cmp(elem, pivot)`，而不是用一些独立的 `elem < pivot` 或 `elem > pivot` 来测试。不幸的是，`cmp` 不会提高速度；事实上它还有可能会降速，至少当 `data` 的元素是一些基本类型比如数字和字符串的时候，的确是这样。

那么，`select` 的性能和下面这个简单方法的性能相比如何呢？

```
def selsor(data, n):
    data = list(data)
    data.sort()
    return data[n]
```

在我的计算机上，获取一个 3 001 个整数的充分洗牌的列表的中值，本节解决方案的代码耗时 16ms，而 `selsor` 耗时 13ms，再考虑到 `sort` 在序列部分有序的情况下速度会更快，元素是基本类型且比较操作执行得很快，而且列表长度也不大，所以使用 `select` 并没有什么优势。将长度增加到 30 001，这两个方法的性能变得非常接近，都是约 170ms。但当我将列表长度修改成 300 001，`select` 终于表现出了优势，它用了 2.2s 获得了中值，而 `selsor` 需要 2.5s。

但如果序列中元素的比较操作非常耗时，那么这两个方式刚刚表现出的大致平衡就被

彻底打破了，因为这两个方式的最关键的差异就是比较操作执行的次数——`select` 执行  $O(n)$  次，而 `selsor` 执行  $O(n \log n)$  次。举个例子，假如我们需要比较的是某个类的实例，其比较操作的开销相当大（模拟某些四维的坐标点，其前三维坐标通常总是相同的）：

```
class X(object):
    def __init__(self):
        self.a = self.b = self.c = 23.51
        self.d = random.random()
    def __datas(self):
        return self.a, self.b, self.c, self.d
    def __cmp__(self, oth):
        return cmp(self.__datas(), oth.__datas())
```

现在，即使只对 201 个实例求中值，`select` 就已经表现得比 `selsor` 快了。

换句话说，基于列表的 `sort` 方法的实现的确要简洁得多，实现 `select` 也确实需要多付出一点力气，但如果  $n$  足够大而且比较操作的开销也无法忽略的话，`select` 就体现出它的价值了。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 文档中关于 `list` 类型的 `sort` 方法，以及 `random` 模块。

## 5.11 三行代码的快速排序

感谢：Nathaniel Gray、Raymond Hettinger、Christophe Delord、Jeremy Zucker

### 任务

你想证明，Python 对函数式编程范式的支持比第一眼看上去的印象强多了。

### 解决方案

函数式编程语言非常漂亮，比如 Haskell 就是个例子，但 Python 的表现也不遑多让：

```
def qsort(L):
    if len(L) <= 1: return L
    return qsort([lt for lt in L[1:] if lt < L[0]]) + L[0:1] + \
           qsort([ge for ge in L[1:] if ge >= L[0]])
```

我个人认为，上述代码和 Haskell 的版本 (<http://www.haskell.org>) 比起来一点也不逊色：

```
qsort [] = []
qsort (x:xs) = qsort elts_lt_x ++ [x] ++ qsort elts_greq_x
               where
```

```
elts_lt_x = [y | y <- xs, y < x]
elts_greq_x = [y | y <- xs, y >= x]
```

下面给出一个 Python 版本的测试函数：

```
def qs_test(length):
    import random
    joe = range(length)
    random.shuffle(joe)
    qsJoe = qsort(joe)
    for i in range(len(qsJoe)):
        assert qsJoe[i] == i, 'qsort is broken at %d!' % i
```

## 讨论

相比于快速排序的原生实现，我们给出的实现能展现列表推导的强大表现力。但千万不要在真实的代码中使用这种方法。Python 列表有一个 `sort` 方法，速度要比我们的实现快得多，无疑应该是首选的方案：在 Python 2.4 中，新的内建函数 `sorted` 接受任意的有限序列并返回一个新的完成了排序的序列。这段代码的唯一用处是向朋友们炫耀，尤其是那些函数式编程的狂热分子，比如 Haskell 语言的爱好者。

我在 <http://www.haskell.org/aboutHaskell.html> 看到了 Haskell 的漂亮的快速排序之后，产生了写一个 Python 的对应版本的想法。在看到 Haskell 的函数优雅的表现能力之后，我意识到借助 Python 中的列表推导也能以同样方式实现。除了使用从 Haskell 偷学来的列表推导，我们也设法加入了很多 Python 风格的东西。

这两个实现都是以列表的第一个元素为轴心，因而对于一个普通的，已经完成排序的列表它们会有最差的性能表现  $O(n)$ 。绝不应该在工作代码中使用它，但既然本节的目标是显摆，那也无所谓了。

还可以写一个没有那么紧凑，但具有同样结构的版本，并使用命名的局部变量和函数来增加清晰度和可读性：

```
def qsort(L):
    if not L: return L
    pivot = L[0]
    def lt(x): return x<pivot
    def ge(x): return x>=pivot
    return qsort(filter(lt, L[1:]))+[pivot]+qsort(filter(ge, L[1:]))
```

一旦沿着这条思路走下去，就可以很容易地对原来的做法继续改进，比如使用一个随机的轴心元素来尽可能地避免最糟糕的情况，并对选用的轴心计数，来应对序列中有太多相等地元素的情况：

```
import random
def qsort(L):
    if not L: return L
    pivot = random.choice(L)
```

```
def lt(x): return x<pivot
def gt(x): return x>pivot
return qsort(filter(lt, L))+[pivot]*L.count(pivot)+qsort(filter(gt,
L))
```

虽然代码得到了增强，但是看上去没那么有趣了，而且不利于炫耀。而真正的工作级别的排序代码是另一码事：不管我们多么喜欢这些看上去可爱的代码，它们在性能和稳固度方面永远也无法和 Python 内建的排序相提并论。

除了提高清晰度和健壮性，我们还可以把力气用在相反的方向上，即利用 Python 的 `lambda` 写出更紧凑和晦涩的东西：

```
q=lambda x:(lambda o=lambda s:[i for i in x if cmp(i,x[0])==s]:
            len(x)>1 and q(o(-1))+o(0)+q(o(1)) or x)()
```

至少，它还足够漂亮（一行代码，当然由于长度的问题只好被截为两行），但你应该明白，真实的应用绝对不应该是这样的。我们用可读性较好的 `def` 语句来替换掉晦涩难懂的 `lambda`，得到的等价代码仍然不是很好读：

```
def q(x):
    def o(s): return [i for i in x if cmp(i,x[0])==s]
    return len(x)>1 and q(o(-1))+o(0)+q(o(1)) or x
```

但如果我们将那个过于精炼的 `len(x)>1 and ... or x` 拆开，并用 `if/else` 语句代替，同时引入一些局部名字，清晰度似乎提高了不少：

```
def q(x):
    if len(x)>1:
        lt = [i for i in x if cmp(i,x[0]) == -1 ]
        eq = [i for i in x if cmp(i,x[0]) == 0 ]
        gt = [i for i in x if cmp(i,x[0]) == 1 ]
        return q(lt) + eq + q(gt)
    else:
        return x
```

幸亏在真实世界中，Python 用户并不喜欢写出那种盘绕的、层层叠叠的充满 `lambda` 的代码。事实上，很多（但也得承认不是所有）人都对 `lambda` 感到厌烦（可能是由于看到过多的对 `lambda` 的滥用），因此都尽量地使用可读性较好的 `def` 语句。因此，Python 并不要求用户具有能够从乱麻一般的代码中解码的能力，但也许别的语言是需要这种能力的。语言的任何一个特性都可能被程序员滥用，以显得更“聪明”。这样导致的结果是，一些 Python 用户（虽然是少数）甚至对列表推导（列表推导可以被揉进一堆事物，和简单的 `for` 循环相比，的确看上去没那么清晰）和 `and/or` 操作符的短路行为模式（因为利用它们可以写出非常晦涩精炼的代码，完全无法和 `if` 语句的清晰易读相比）也感到恐惧。

## 更多资料

Haskell 的主页，<http://www.haskell.org>。

## 5.12 检查序列的成员

感谢: Alex Martelli

### 任务

你需要对一个列表执行很频繁的成员资格检查。而 `in` 操作符的  $O(n)$  时间复杂度对性能的影响很大，你也不能将序列转化为一个字典或者集合，因为你还需要保留原序列的元素顺序。

### 解决方案

假设需要给列表添加一个在该列表中不存在的元素。一个可行的方法是写这样一个函数：

```
def addUnique(baseList, otherList):
    auxDict = dict.fromkeys(baseList)
    for item in otherList:
        if item not in auxDict:
            baseList.append(item)
            auxDict[item] = None
```

如果你的代码只是在 Python 2.4 下运行，那么将辅助字典换成辅助集合效果是完全一样的。

### 讨论

下面给出一个简单（天真？）的方式，看上去相当不错：

```
def addUnique_simple(baseList, otherList):
    for item in otherList:
        if item not in baseList:
            baseList.append(item)
```

如果列表很短的话，这个方法倒也没问题。

但是，如果列表不是很短，这个简单的方法会非常慢。当你用 `if item not in baseList` 这样的代码进行检查时，Python 只会用一种方式执行 `in` 操作：对列表 `baseList` 的元素进行内部的循环遍历，如果找到一个元素等于 `item` 则返回 `True`，如果直到循环结束也没有发现相等的元素则返回 `False`。`in` 操作的平均执行时间是正比于 `len(baseList)` 的。`addUnique_simple` 执行了 `len(otherList)` 次 `in` 操作，因此它消耗的时间正比于这两个列表长度的乘积。

而解决方案给出的 `addUnique` 函数，首先创建了一个辅助的字典 `auxDict`，这一步的时间正比于 `len(baseList)`。然后在循环中检查 `dict` 的成员——这是造成巨大差异的一步，

因为检查一个元素是否处于一个 dict 中的时间大致是一个常数，而与 dict 中元素的数目没有关系。因此，那个 for 循环消耗的时间正比于 len(otherList)，这样，整个函数所需要的时间就正比于这两个列表的长度之和。

对于运行时间的分析还可以挖得更深一点，因为在 addUnique\_simple 中 baseList 的长度并不是一个常量；每当找到一个不属于 baseList 的元素，baseList 的长度就会增加。但这样的分析结果不会与前面的简化版的结果有太大出入。我们可以准备一些用例进行测试。当每个列表中有 10 个整数且有 50% 的重叠时，简化版比解决方案给出的函数慢 30%，这样的性能下降还可以忽略。若每个列表都有 100 个整数，而且仍然有 50% 的重叠部分，简化版比解决方案的函数慢 12 倍——这种级别的减速效果就无法忽略了，而且当列表变得更长的时候，情况也变得更糟。

有时，将一个辅助的 dict 和序列一起使用并封装成一个对象能提高你的应用程序的性能。但在这个例子中，必须在序列被修改时不断地维护 dict，以保证它总是和序列当前所拥有的元素保持同步。这个维护任务并不是很简单，我们有很多方法来实现同步。下面给出一种“即时”的同步方式，当需要检查某元素，或者字典的内容可能已经无法和列表内容保持同步时，我们就重新构建一个辅助 dict。由于开销很小，下面的类优化了 index 方法和成员检查部分的代码：

```
class list_with_aux_dict(list):
    def __init__(self, iterable=()):
        list.__init__(self, iterable)
        self._dict_ok = False
    def _rebuild_dict(self):
        self._dict = {}
        for i, item in enumerate(self):
            if item not in self._dict:
                self._dict[item] = i
        self._dict_ok = True
    def __contains__(self, item):
        if not self._dict_ok:
            self._rebuild_dict()
        return item in self._dict
    def index(self, item):
        if not self._dict_ok:
            self._rebuild_dict()
        try: return self._dict[item]
        except KeyError: raise ValueError
    def _wrapMutatorMethod(methname):
        _method = getattr(list, methname)
        def wrapper(self, *args):
            # 重置字典的OK标志，然后委托给真正的方法
            self._dict_ok = False
            return _method(self, *args)
    # 只适用于 Python 2.4: wrapper.__name__ = _method.__name__
```

```
    setattr(list_with_aux_dict, methname, wrapper)
for meth in 'setitem delitem setslice delslice iadd'.split( ):
    _wrapMutatorMethod('__%s__' % meth)
for meth in 'append insert pop remove extend'.split( ):
    _wrapMutatorMethod(meth)
del _wrapMethod                      # 删除辅助函数, 已经不再需要它了
```

`list_with_aux_dict` 扩展了 `list`, 并将原 `list` 的所有方法仍然委托给它, 除了 `__contains__` 和 `index`。所有能够修改 `list` 的方法都被封装进了一个闭包, 该闭包负责重置一个标志, 以确保辅助字典的有效性。Python 的 `in` 操作符调用 `__contains__` 方法。除非标志被设置, 否则 `list_with_aux_dict` 的 `__contains__` 方法会重建辅助字典 (标志被设置时, 重建没有必要), 而 `index` 方法仍然像原先一样工作。

上述 `list_with_aux_dict` 类并没有用帮助函数为列表的所有属性方法绑定和安装一个闭包, 而是只取所需, 我们也可以在 `list_with_aux_dict` 的主体中写出所有的 `def` 语句来替代 `wrapper` 方法。但是上述代码有个重要的优点是消除了冗余和重复 (重复和啰嗦的代码让人生厌, 而且容易滋生 bug)。Python 在自省和动态改变方面的能力给你提供了一个选择: 可以创建一个 `wrapper` 方法, 用一种聪明而简练的方式; 或者, 如果你想避免使用被人称为黑魔法的类对象的自省和动态改变, 也可以写一堆重复啰嗦的代码。

`list_with_aux_dict` 的结构很适合通常的使用模式, 即对序列的修改操作一般总是集中出现, 然后接着又会有一段时间序列无须被修改, 但需要检查元素的成员资格。如果参数 `baseList` 不是一个普通的列表, 而是 `list_with_aux_dict` 的一个实例, 早先展示的 `addUnique_simple` 函数也不会因此得到任何性能上的提升, 因为这个函数会交替地进行成员资格检查和序列修改。因此, 类 `list_with_aux_dict` 中过多的辅助字典的重建影响了函数的性能。(除非是针对某个特例, 比如 `otherList` 中绝大多数元素都已经在 `baseList` 中出现过了, 因此对序列的修改相比于对元素的检查, 发生的次数要少得多。)

对这些成员资格的检查所做的优化有个重要的前提, 即序列中的值必须是可哈希的 (不然的话, 它们不能被用来做字典的键或者集合的元素)。举个例子, 元组的列表仍适用于本节的解决方案, 但对于列表的列表, 我们恐怕得另外想办法了。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中关于序列类型和映射类型的章节。

## 5.13 寻找子序列

感谢: David Eppstein、Alexander Semenov

### 任务

你需要在某大序列中查找子序列。

## 解决方案

如果序列是字符串（普通的或者 Unicode），Python 的字符串的 `find` 方法以及标准库的 `re` 模块是最好的工具。否则，应该使用 Knuth-Morris-Pratt 算法（KMP）：

```
def KnuthMorrisPratt(text, pattern):
    ''' 在序列 text 中找到 pattern 的子序列的起始位置
        每个参数都可以是任何可迭代对象
        在每次产生一个结果时，对 text 的读取正好到达（包括）
        对 pattern 的一个匹配 '''
    # 确保能对 pattern 进行索引操作，同时制作 pattern 的
    # 一个拷贝，以防在生成结果时意外地修改 pattern
    pattern = list(pattern)
    length = len(pattern)
    # 创建 KMP “偏移量表” 并命名为 shifts
    shifts = [1] * (length + 1)
    shift = 1
    for pos, pat in enumerate(pattern):
        while shift <= pos and pat != pattern[pos-shift]:
            shift += shifts[pos-shift]
        shifts[pos+1] = shift
    # 执行真正的搜索
    startPos = 0
    matchLen = 0
    for c in text:
        while matchLen == length or matchLen >= 0 and pattern[matchLen] != c:
            startPos += shifts[matchLen]
            matchLen -= shifts[matchLen]
        matchLen += 1
        if matchLen == length: yield startPos
```

## 讨论

本节实现的 Knuth-Morris-Pratt 算法可被用于在一个大文本的连续序列中查找某种指定的模式。由于 KMP 是以顺序的方式访问文本，所以很自然地，我们也可以将其应用于包括文本在内的任意可迭代对象。在处理阶段，算法会创建一个关于偏移量的表，它消耗的时间正比于模式的长度，而每个文本标志则以恒定的时间处理。在所有关于文本处理的基础算法书中都可以看到 KMP 算法的解释和示例。（更多资料一栏中也提供了推荐读物。）

如果 `text` 和 `pattern` 都是 Python 字符串，可以通过使用 Python 的内建搜索方法得到一个更快的方案：

```
def finditer(text, pattern):
    pos = -1
    while True:
        pos = text.find(pattern, pos+1)
```

```
if pos < 0: break
yield pos
```

比如，使用一个长度为 4 的字母表 (“ACGU”), 在长度为 100000 的文本中查找长度为 8 的一个模式，在我的计算机上，借助 `finditer` 函数耗时为 4.3ms，但使用 `KnuthMorrisPratt` 函数执行同样任务则需要 540ms（在 Python 2.3 中；而在 Python 2.4 会快一些，约 480ms，但仍然比 `finditer` 慢了超过 100 倍）。所以请记住：本节的算法适用于在通用的序列中进行搜索，包括那些数据量大到无法放入内存的情况，如果只需要对字符串搜索，Python 内建的搜索方法具有完全压倒性的优势。

## 更多资料

关于基础算法的优秀书籍有很多；其中一本备受推崇的书是 Thomas H. Cormen、Charles E. Leiserson、Ronald L. Rivest、Clifford Stein，合著的 *Introduction to Algorithms* (MIT Press)，第二版。

## 5.14 给字典类型增加排名功能

感谢：Dmitry Vasiliev、Alex Martelli

### 任务

你需要用字典存储一些键和“分数”的映射关系。你经常需要以自然顺序（即以分数的升序）访问键和分数值，并能够根据那个顺序检查一个键的排名。对这个问题，用 `dict` 似乎不太合适。

### 解决方案

我们可以使用 `dict` 的子类，根据需要增加或者重写一些方法。在我们使用多继承、将 `UserDict.DictMixin` 放置在基类 `dict`、并仔细安排各种方法的委托或重写之前，我们可以设法获得一种美妙的平衡，既拥有极好的性能又避免了编写一些冗余代码。

我们可以在文档字符串中加入很多示例，还可以用标准库的 `doctest` 模块来提供单元测试的功能，这也能够确保我们在文档字符串中编写的例子的准确性：

```
#!/usr/bin/env python
''' 一个反映键到分数的映射的字典 '''
from bisect import bisect_left, insort_left
import UserDict
class Ratings(UserDict.DictMixin, dict):
    """ Ratings 类很像一个字典，但有一些额外特性：每个键
        的对应值都是该键的“分数”，所有键都根据它们的
        分数排名。对应值必须是可以比较的，同样，键则必须
        是可哈希的（即可以“绑”在分数上）
```

所有关于映射的行为都如同预期一样，比如：

```
>>> r = Ratings({ "bob": 30, "john": 30})
>>> len(r)
2
>>> r.has_key("paul"), "paul" in r
(False, False)
>>> r["john"] = 20
>>> r.update({ "paul": 20, "tom": 10 })
>>> len(r)
4
>>> r.has_key("paul"), "paul" in r
(True, True)
>>> [r[key] for key in ["bob", "paul", "john", "tom"]]
[30, 20, 20, 10]
>>> r.get("nobody"), r.get("nobody", 0)
(None, 0)
```

除了映射的接口，我们还提供了和排名相关的方法。

r.rating(key) 返回了某个键的排名，其中排名为 0 的是最低的分数（如果两个键的分数相同，则直接比较它们两者，“打破僵局”，较小的键排名更低）：

```
>>> [r.rating(key) for key in ["bob", "paul", "john", "tom"]]
[3, 2, 1, 0]
```

getValueByRating(ranking) 和 getKeyByRating(ranking) 对于给定的排名索引，分别返回分数和键：

```
>>> [r.getValueByRating(rating) for rating in range(4)]
[10, 20, 20, 30]
>>> [r.getKeyByRating(rating) for rating in range(4)]
['tom', 'john', 'paul', 'bob']
```

一个重要的特性是 keys() 返回的键是以排名的升序排列的，而其他所有返回的相关的列表或迭代器都遵循这个顺序：

```
>>> r.keys()
['tom', 'john', 'paul', 'bob']
>>> [key for key in r]
['tom', 'john', 'paul', 'bob']
>>> [key for key in r.iterkeys()]
['tom', 'john', 'paul', 'bob']
>>> r.values()
[10, 20, 20, 30]
>>> [value for value in r.itervalues()]
[10, 20, 20, 30]
>>> r.items()
[('tom', 10), ('john', 20), ('paul', 20), ('bob', 30)]
>>> [item for item in r.iteritems()]
[('tom', 10), ('john', 20), ('paul', 20), ('bob', 30)]
```

实例可以被修改（添加、改变和删除键-分数对应关系）

而且实例的每个方法都反映了实例的当前状态：

```
>>> r["tom"] = 100
>>> r.items()
[('john', 20), ('paul', 20), ('bob', 30), ('tom', 100)]
```

```

>>> del r["paul"]
>>> r.items( )
[('john', 20), ('bob', 30), ('tom', 100)]
>>> r["paul"] = 25
>>> r.items( )
[('john', 20), ('paul', 25), ('bob', 30), ('tom', 100)]
>>> r.clear( )
>>> r.items( )
[ ]
"""

''' 这个实现小心翼翼地混合了继承和托管，因此在尽量减少
冗余代码的前提下获得了不错的性能，当然，同时也保
证了语义的正确性。所有未被实现的映射方法都通过
继承来获得，大多来自 DictMixin，但关键的 __getitem__
来自 dict. '''
def __init__(self, *args, **kwds):
    ''' 这个类就像 dict 一样被实例化 '''
    dict.__init__(self, *args, **kwds)
    # self._rating 是关键的辅助数据结构：一个所有（值，键）
    # 的列表，并保有一种“自然的”排序状态
    self._rating = [ (v, k) for k, v in dict.iteritems(self) ]
    self._rating.sort()
def copy(self):
    ''' 提供一个完全相同但独立的拷贝 '''
    return Ratings(self)
def __setitem__(self, k, v):
    ''' 除了把主要任务委托给 dict，我们还维护 self._rating '''
    if k in self:
        del self._rating[self.rating(k)]
    dict.__setitem__(self, k, v)
    insort_left(self._rating, (v, k))
def __delitem__(self, k):
    ''' 除了把主要任务委托给 dict，我们还维护 self._rating '''
    del self._rating[self.rating(k)]
    dict.__delitem__(self, k)
''' 显式地将某些方法委托给 dict 的对应方法，以免继承了
DictMixin 的较慢的（虽然功能正确）实现 '''
__len__ = dict.__len__
__contains__ = dict.__contains__
has_key = __contains__
''' 在 self._rating 和 self.keys() 之间的关键的语义联系——DictMixin
“免费”给了我们所有其他方法，虽然我们直接实现它们能够
获得稍好一点的性能. '''
def __iter__(self):
    for v, k in self._rating:
        yield k
iterkeys = __iter__
def keys(self):
    return list(self)

```

```

''' 三个和排名相关的方法 '''
def rating(self, key):
    item = self[key], key
    i = bisect_left(self._rating, item)
    if item == self._rating[i]:
        return i
    raise LookupError, "item not found in rating"
def getValueByRating(self, rating):
    return self._rating[rating][0]
def getKeyByRating(self, rating):
    return self._rating[rating][1]
def _test( ):
    ''' 我们使用 doctest 来测试这个模块，模块名必须为
        rating.py，这样 docstring 中的示例才会有效 '''
    import doctest, rating
    doctest.testmod(rating)
if __name__ == "__main__":
    _test()

```

## 讨论

在很多方面，字典都是很自然地被应用于存储键（比如，竞赛中参与者的姓名）和“分数”（比如参与者获得的分数，或者参与者在拍卖中的出价）的对应关系的数据结构。如果我们希望在这些应用中使用字典，我们可能会希望以自然的顺序访问——即键对应的“分数”的升序——我们也希望能够迅速获得基于当前分数的排名（比如，参与者现在排在第三位，排在第二位的参与者的分数，等等）。

为了达到这个目的，本节给 `dict` 的子类增加了一些它本身完全不具备的功能（`rating` 方法、`getValueByRating`、`getKeyByRating`），同时，最关键和巧妙的地方是，我们修改了 `keys` 方法和其他相关的方法，这样它们就能返回按照指定顺序排列的列表或者可迭代对象（比如按照分数的升序排列；对于两个有同样分数的键，我们继续比较键本身）。大多数的文档都放在类的文档字符串中——保留文档和示例是很重要的，可以用 Python 标准库的 `doctest` 模块来提供单元测试的功能，以确保给出的例子是正确的。

关于这个实现的有趣之处是，它很关心消除冗余（即那些重复和令人厌烦的代码，很可能滋生 bug），但同时没有损害性能。`Ratings` 类同时从 `dict` 和 `DictMixin` 继承，并把后者排在基类列表的第一位，因此，除非明确地覆盖了基类的方法，`Ratings` 的方法基本来自于 `DictMixin`，如果它提供了的话。

Raymond Hettinger 的 `DictMixin` 类最初是发布在 *Python Cookbook* 在线版本中的一个例子，后来被吸收到了 Python 2.3 的标准库中。`DictMixin` 提供了各种映射的方法，除了 `__init__`、`copy`、以及四个基本方法：`__getitem__`、`__setitem__`、`__delitem__` 和 `keys`。如果需要的是一个映射类并且想要支持完整映射所具有的各种方法，可以从 `DictMixin` 派生子类，并且提供那些基本的方法（具体依赖于你的类的语义——比如，如果你的

类有不可修改的实例，你无须提供属性设置方法`__setitem__`和`__delitem__`。还可以添加一些可选的方法以提升性能，覆盖 DictMixin 所提供的原有方法。整个 DictMixin 的架构可以被看做是一个经典的模板方法设计模式（Template Method Design Pattern），它用一种混合的变体提供了广泛的适用性。

在本节的类中，从基类继承了`__getitem__`（准确地说，是从内建的`dict`类型继承），出于性能上的考虑，我们把能委托的都委托给了`dict`。我们必须自己实现基本的属性设置方法（`__setitem__`和`__delitem__`），因为除了委托给基类的方法，还需要维护一个数据结构`self._rating`——这是一个列表，包含了许多`(score, key)`值对，此列表在标准库模块`bisect`的帮助下完成了排序。我们也重新实现了`keys`（在这个步骤中，还重新实现了`__iter__`，即`iterkeys`，很明显，借助`__iter__`可以更容易地实现`keys`）来利用`self._rating`并按照我们需要的顺序返回键。最后，除了上面三个和排名有关的方法，我们又为`__init__`和`copy`添加了实现。

这个结果是一个很有趣的例子，它取得了简洁和清晰的平衡，并最大化地重用了 Python 标准库的众多功能。如果你在应用程序中使用这个模块，测试结果可能会显示，本节的类从 DictMixin 继承来的方法的性能不是太让人满意，毕竟 DictMixin 的实现是基于必要的通用性的考虑。如果它的性能不能满足你的要求，可以自己提供一个实现来获取最高性能。假设有个 Ratings 类的实例`r`，你的应用程序需要对`r.iteritems()`的结果进行大量的循环处理，可以给类的主体部分增加这个方法的实现以获得更好的性能：

```
def iteritems(self):
    for v, k in self._rating:
        yield k, v
```

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中 UserDict 模块下的 DictMixin 类，以及 bisect 模块。

## 5.15 根据姓的首字母将人名排序和分组

感谢：Brett Cannon、Amos Newcombe

### 任务

你想将一组人名写入一个地址簿，同时还希望地址簿能够根据姓的首字母进行分组，且按照字母顺序表排序。

### 解决方案

Python 2.4 的新`itertools.groupby`函数使得这个任务很简单：

```

import itertools
def groupnames(name_iterable):
    sorted_names = sorted(name_iterable, key=_sortkeyfunc)
    name_dict = { }
    for key, group in itertools.groupby(sorted_names, _groupkeyfunc):
        name_dict[key] = tuple(group)
    return name_dict
pieces_order = { 2: (-1, 0), 3: (-1, 0, 1) }
def _sortkeyfunc(name):
    ''' name 是带有名和姓以及可选的中名或首字母的字符串,
       这些部分之间用空格隔开; 返回的字符串的顺序是
       姓-名-中名, 以满足排序的需要 '''
    name_parts = name.split( )
    return ' '.join([name_parts[n] for n in
pieces_order[len(name_parts)]]])
def _groupkeyfunc(name):
    ''' 返回的键 (即姓的首字母) 被用于分组 '''
    return name.split( )[-1][0]

```

## 讨论

本节解决方案中的 `name_iterable` 必须是一个可迭代对象，它的元素是遵循名-中名-姓格式的人名字字符串，其中中名是可选的且各部分以空格隔开。对这个可迭代对象调用 `groupnames` 得到的结果是一个字典，它的键是姓的首字母，而对应的值则是完整的名、中名和姓的构成的元组。

不管是“名 姓”还是“名 中名 姓”的格式，辅助的 `_sortkeyfunc` 函数都能将人名字字符串切割开，并将各部分记录到一个列表中，其顺序是先姓后名，如果有中名，还要加上中名或首字母，最后将这个列表拼接成一个字符串并返回。根据任务的描述，这个字符串是用来排序的关键。Python 2.4 的内建函数 `sorted` 用这个函数（它将被应用到每个元素上用于获取排序的键）作为可选的名为 `key` 的参数。

辅助函数 `_groupkeyfunc` 也接受同样格式的人名，并返回姓的首字母——根据问题的描述，这是我们用来将人名分组的关键。

方案中的主函数 `groupnames` 使用了两个辅助函数和 Python 2.4 的 `sorted` 和 `itertools.groupby` 来解决问题，创建并返回了我们要求的字典。

如果想在 Python 2.3 中完成这个任务，仍然可以使用这两个支持函数并重新编写 `groupnames`。由于 Python 2.3 的标准库中并没有提供 `groupby` 函数，先分组再分别对各个组排序会更方便一些：

```

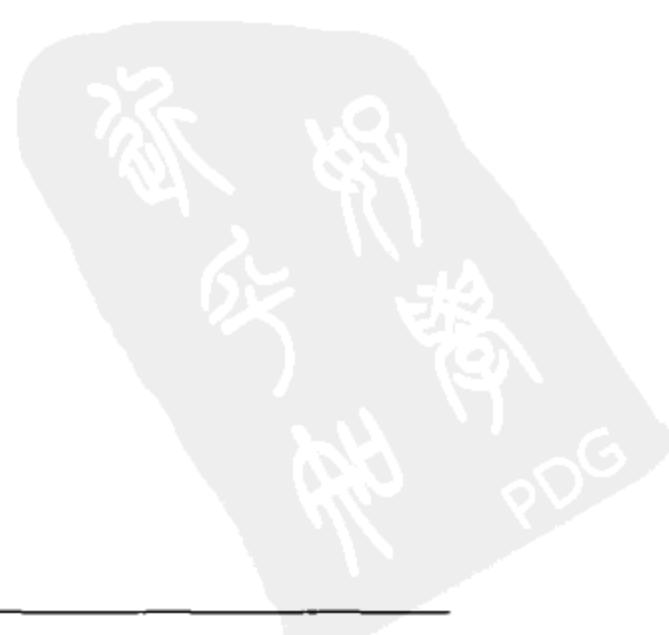
def groupnames(name_iterable):
    name_dict = { }
    for name in name_iterable:
        key = _groupkeyfunc(name)

```

```
name_dict.setdefault(key, [ ]).append(name)
for k, v in name_dict.iteritems( ):
    aux = [(_sortkeyfunc(name), name) for name in v]
    aux.sort( )
    name_dict[k] = tuple([ n for _, n in aux ])
return name_dict
```

## 更多资料

19.21 节, *Library Reference* (Python 2.4) 关于 `itertools` 的文档。



# 面向对象编程

### 引言

感谢：Alex Martelli，*Python in a Nutshell* (O'Reilly) 的作者

面向对象编程 (OOP) 是 Python 强大力量的源泉之一。Python 的 OOP 特性也随着 Python 本身的发展，稳定而持续地进步。不像别的程序（不包括 Lisp 及其变种：只要你能够忍受无穷无尽的括号语法，我怀疑没有什么事情是 Lisp 不能干的），在 Python 1.5.2（那是一个古老的、长期稳定的版本，我第一次用 Python 工作时用的就是它）中你就已经可以写出很好的面向对象的程序了。最近几年来，自从 Python 2.2 发布之后，Python OOP 比起 1.5.2 时又进步了很多。Python 在没有牺牲可靠性、稳定性和向后兼容性的前提下取得如此大的进步，确实让人感到不可思议。

为了尽可能地利用 Python 的 OOP 特性，应该尽量用 Python 的方式，而不是模仿 C++、Java、SmallTalk，或其他你熟悉的语言。由于 Python 的强大，也可以用它做很多模仿的事情。但如果花更多的时间和精力在学习和理解 Python 特有的方式上，你会获得更好的成果。而且这些时间和精力也有助于加深对 OOP 本身的理解：什么是 OOP，为什么要用它，你的面向对象程序能够利用它的什么机制？当然，也需要用一些时间和精力去理解 Python 提供的一些特别的机制。

不过我还是要事先提个醒。对于这种高层语言，Python 在幕后使用的 OOP 机制是非常明确的：对于你的探索和修改行为，这些机制是充分暴露的。进行一些探索并尝试理解它的实质是好事，但是一定要抵御修改的诱惑。换句话说，不要因为具有使用黑魔法的能力就滥用这些没有必要的特殊手段。尤其要指出的是，不要在生产级别的代码中使用黑魔法。如果用简洁的方式就能达成目标（在 Python 中，大多数时候都可以简洁地实现目标），那你就应该保持代码的简单。简洁换来了可读性、可维护性，甚至还常常带来更好的性能。在 Python 的文化中，用“聪明”描述某事物并不是什么赞美之辞。

那么 OOP 是什么呢？首先，它能够将状态（数据）和一些行为（代码）包裹在一起，置入一个趁手的包中。这里，“趁手的包”就是关键。每个程序都有状态和行为——编程范式的差别在于怎么查看、组织和包装它们。如果包装意味着封装了状态和行为的对象，说明使用的就是 OOP。一些面向对象的语言强迫你对任何事物使用 OOP，因而会得到一大堆没有状态或者没有行为的对象。而 Python 却支持多种范式。虽然 Python 中的所有事物都是对象，但可以在需要的时候才将事物用 OOP 的方式打包。其他语言会试图强迫你用一种预先定义的模式和风格来编程，但 Python 则在背后提供支持并允许你充分表达自己的设计选择。

借助 OOP 的设计和支持，一旦指定了对象如何构成，就可以根据需要实例化任意多的这种类型的对象。当不需要多个对象时，还可以考虑使用 Python 的其他结构，比如模块。在本章中，你将看到单例模式——一种面向对象的设计模式，它能够消除对象的多重性，还有 Borg 模式，它使得多个实例可以共享状态。但如果需要的仅仅是一个实例，在 Python 中最好的方式是使用模块，而不是一个 OOP 对象。

要描述对象是如何构建的，可以使用 `class` 声明：

```
class SomeName(object):
    """ 通常在这里定义数据以及编写代码（在类的主体中） """
```

`SomeName` 是一个类对象。它是第一等级的对象，就像 Python 的其他对象一样，这意味着可以把它放到列表中或者字典中，将它作为参数传递给函数等。你不一定要在 `class` 的头部子句中引入 `(object)` 部分，`class SomeName` 本身就是有效的 Python 语法，但通常应该包括那个部分，我们在后面会继续讨论。

当需要一个对象的新实例时，像函数一样调用类对象即可。下面代码中的每次调用都将返回一个新的实例对象：

```
anInstance = SomeName()
another = SomeName()
```

`anInstance` 和 `another` 是 `SomeName` 类的两个独立的实例对象。（参看 4.18 节中那个简单的类，没有比上面给出示例类多做什么事情，但却非常有用。）可以随意地绑定（比如，赋值或者设置）并访问（比如 `get`）一个实例对象的属性：

```
anInstance.someNumber = 23 * 45
print anInstance.someNumber          # 输出: 1035
```

一个像 `SomeName` 一样的“空”类的实例没有什么行为，但它们却可能有状态。而很多时候，需要实例有自己的行为。可以在类的主体部分定义方法（通过 `def` 声明，就像定义其他的函数一样）来指定其行为：

```
class Behave(object):
    def __init__(self, name):
        self.name = name
    def once(self):
```

```
    print "Hello,", self.name
def rename(self, newName)
    self.name = newName
def repeat(self, N):
    for i in range(N): self.once()
```

在 Python 中定义类的方法和函数都使用 `def` 语句，这是因为在本质上方法其实就是函数。然而，方法同时还是一个类对象的属性，它的第一个正式的参数（根据通用约定）名为 `self`，此参数总是引用被调用方法的对象自身。

拥有特殊名字的 `__init__` 方法也被称为构造函数（或者更确切地称作初始化函数），用于创建类实例。Python 调用这个特殊的方法，并使用传入的参数（除了 `self`。你无须显式地传递这个参数，因为 Python 会自动提供）来初始化一个新建的实例。`__init__` 的主体部分则将新创建的实例的属性正确绑定，并初始化其状态。

类实例的行为则由其他方法实现。一般而言，这些方法会访问实例的属性。它们也可能将实例的属性重新绑定，并调用其他的方法。在类的内部定义中，这些行为可以经由 `self.something` 语法完成。一旦实例化了一个类，你是对类实例调用方法，访问的是类实例的属性，重新绑定的也是类实例的属性，需要使用 `theobject.something` 语法：

```
beehive = Behave("Queen Bee")
beehive.repeat(3)
beehive.rename("Stinger")
beehive.once()
print beehive.name
beehive.name = 'See, you can rebind it "from the outside" too, if you want'
beehive.repeat(2)
```

### self

我提到过的 `self.something` 语法和 `theobject.something` 语法并没有实质性的区别：前者是后者的一个特例，只是 `theobject` 的名字正好是 `self`。

如果你是 Python OOP 的新手，应该试试 Python 的交互式环境，并在这个环境中测试已经展示过的以及将要展示的例子。作为 Python 的一个重要组成部分的免费的 IDLE 开发环境对于这类测试和探索任务是一个很好的工具。

除了构造函数 (`__init__`)，你的类可能还有其他特殊的方法，即，前后都带有两个下划线的方法。当类实例被用于各种操作和内建函数时，Python 会自动调用那些特殊的方法。举个例子，`len(x)` 返回的是 `x.__len__()`；`a + b` 通常则返回 `a.__add__(b)`；`a[b]` 返回 `a.__getitem__(b)`。因此，在类中定义了特殊方法后，你的类实例就可以像其他内建类型一样交互，如数字、列表、字典等。



每种操作以及内建函数都能以某种特定顺序执行一些特殊的方法。比如，`a+b` 首先尝试 `a.__add__(b)`，如果不奏效，Python 会给对象 `b` 一个机会，继续尝试 `b.__add__(a)`。特殊方法的这种内在的，针对各种操作和内建函数的自我组织和调整的能力，是对函数和方法的纯 OO 记法（如 `someobject.somemethod(arguments)`）的重要补充和增强。

能够以相似的方式处理各种不同对象的能力，也被称为多态，这是 OOP 的一个重要优点。归功于多态，你能够对不用的对象调用相同的方法，每个对象都在特定的层次上实现了该方法。比如，还可以在另一个类中实现 `repeat` 方法，并具有不同的行为：

```
class Repeater(object):
    def repeat(self, N): print N***-
```

如果愿意，可以混合使用 `Behave` 和 `Repeater` 的实例，只要调用的方法是每种实例都具有的 `repeat`：

```
aMix = beehive, Behave('John'), Repeater(), Behave('world')
for whatever in aMix: whatever.repeat(3)
```

其他的语言需要继承，或者正式定义并实现一个接口，以实现类似的多态性。在 Python 中，你所做的仅仅是让这些方法具有相同的签名（相同方法的名字，相同的参数）。这种基于签名的多态机制使得一种风格非常接近泛型编程的编程方式成为可能（比如，C++ 支持的模板类和模板函数），而且无须引入冗余的语法，也不用在概念上增加任何复杂度。

继承是一种方便、优雅、结构化的重用代码的方式，而 Python 也同样支持继承。可以定义一个类，使其从其他的类派生（其他类的子类），并且增加或者重定义（也称为重写）一些方法：

```
class Subclass(Behave):
    def once(self): print '(%s)' % self.name
subInstance = Subclass("Queen Bee")
subInstance.repeat(3)
```

`Subclass` 子类重写了 `once` 方法，但仍能够调用 `subInstance` 的 `repeat` 方法，因为 `Subclass` 从 `Behave` 超类继承了这个方法。`repeat` 方法的主体部分对指定的实例调用了 `n` 次 `once` 方法，它并不关心这个实例拥有的是什么版本的 `once` 方法。因此，在 `Subclass` 中，对这个方法的每一次调用都将打印出被括号括住的名字，而这并不是 `Behave` 类的原版方法的行为，原版方法将打印出问候语和名字。同一个实例中，用一个方法调用其他方法并获得正确的被覆盖版本，这是面向对象语言中一种非常重要的思想，这里提到的面向对象语言当然也包括了 Python。这种方式被称为模板方法设计模式。

子类的方法常常会覆盖从超类获得的方法，但有时它也需要在自己的操作中调用超类的方法。在 Python 中，只需将这个方法作为类的属性来获取，并传入实例作为它的第一个参数即可：

```
class OneMore(Behave):
    def repeat(self, N): Behave.repeat(self, N+1)
zealant = OneMore("Worker Bee")
zealant.repeat(3)
```

OneMore 类实现了它自己的 repeat 方法，方法名和超类 Behave 的方法完全一样，但行为略有不同。这种方式，也叫托管，在编程中是很常见的。托管，意味着让原有的代码片段完成大部分工作，从而实现某些功能，但常常也带有一些细微的修改。对于重写的方法来说，最好将其中一部分工作委托给与超类同名的方法。在 Python 中，语法 Classname.method (self, ...) 将工作委托给了 Classname 版本的 method。但执行超类托管的首选方式还是使用 Python 内建的 super：

```
class OneMore(Behave):
    def repeat(self, N): super(OneMore, self).repeat(N+1)
```

在本例中，super 结构等价于使用 Behave.repeat，但它也使得 OneMore 类可以被平滑地用于多继承。即使一开始对多继承不感兴趣，也应该养成习惯用 super，而不是显式地通过名字委托给基类，而且 super 也没有什么额外的开销，你会在以后发现它的更多好处。

Python 对多继承提供全面的支持：一个类可以从多个类继承。反映到代码中，这个特性能够让你使用混合类的方式，通过一系列的类提供各种功能（见 6.20 节和 6.12 节的几个不常见但强大的混合用法例子）。除此之外，对于面向对象的分析——即如何看待和概念化你的问题和任务，多继承有着更为重要的作用。单继承把你的问题变成了生物分类（即类别是互斥的）。而真实世界并不这么简单。它更类似于 Jorge Luis Borges 在 *The Analytical Language of John Wilkins* 中的解释，他在其中描述了一个传说中的中国百科全书，*The Celestial Emporium of Benevolent Knowledge*。此书认为所有的动物可以分为：

- 属于皇帝的动物；
- 制成标本的动物；
- 受过训练的动物；
- 乳猪；
- 美人鱼；
- 神奇的动物；
- 走失的狗；
- 在当前分类中的动物；
- 像疯了一样发抖的动物；

- 数不清的动物；
- 用驼毛笔画成的动物；
- 其他的动物；
- 刚打破花瓶的动物；
- 看上去不像苍蝇的动物。

你会看到：生物分类方式强迫你进行分类，将所有的事物都归入某个类中，而分类却未必是真正的排他的。这种生物分类方式，限制了你的程序对真实世界的塑造。而多继承则将你从这种限制中解放出来。

关于那个 (`object`) ——我前面答应过会继续讨论它。现在你已经看到了对于继承的 Python 的记法，可以看出来 `class X(object)` 这种写法意味着类 `X` 从类 `object` 派生而来。如果写 `class Y:`，则相当于告诉 Python，`Y` 并没有继承任何类，这种写法非常清楚明白，“望文生义”即可。出于向后兼容性的考虑，Python 允许创建一个无根的类，而且如果这么做，Python 会让类 `Y` 成为一个“旧风格”的类，也被称为经典类，其工作方式像 Python 老版本中的所有的类的工作方式一样。Python 非常重视向后兼容性。

对于很多基本的应用，你无法觉察出经典类和新风格类的区别，当然对于新的 Python 代码，我还是推荐使用后者。必须要强调，经典类是一个历史遗留的特性，在新代码中并不推荐使用。即使把范围仅仅限于前面我们讨论到的一些基本的 OOP 特性，你也能感觉到经典类的局限性：比如，你无法在经典类中使用 `super`，而且要避免过多地使用多继承。另外，很多新的 Python OOP 中的重要特征，比如内建的特性（`property`），经典类也完全不支持。

在实际工作中，假设需要维护一大段陈旧的 Python 遗留代码，如果想完成一些实质性的维护工作，应该花一点精力来确保所有的类都是新风格的，工作量不大，但这种做法能够在很大程度上减少以后的维护工作。可以明确地让你的类继承于 `object`，另一个等价的方式是，在每个模块中类定义开始的位置，加入下面的赋值声明：

```
__metaclass__ = type
```

内建的 `type` 是 `object` 和其他的新风格类甚至内建的 `type` 的元类。这可以解释为何从 `object` 或者内建的 `type` 继承就能够创建出新风格的类：你编写的类实际上从基类继承了同样的元类。一个没有基类的类也能够从模块的全局 `__metaclass__` 变量获得它的元类，这就是为什么我说即使不明确指定基类，也能确保创建出的类是新风格的。即使你从来不会显式地用到元类（这是一个比较高级的主题，本章将有几节涉及对它的讨论），这种简单的用法也会给你带来好处。

## 什么是元类？

元类并不是什么“阴森的黑魔法”。当执行任何类声明时，Python 会进行下列步骤：将类名作为一个字符串记住，设为 n，并将这个类的基类部分存为一个元组，设为 b。对类主体的执行，会将主体中所有被绑定的名字作为键放入到一个新的字典 d 中，每一个键都有它的对应值（比如，对于 def f(self)这样的声明，d['f']对应到用 def 声明创建出的函数对象）。

而获取元类这一步，假设元类是 M，则可以通过在 d 和 globals 中查找名字 \_\_metaclass\_\_ 来达成：

```
if '__metaclass__' in d: M = d['__metaclass__']
elif b: M = type(b[0])
elif '__metaclass__' in globals(): M = globals()['__metaclass__']
else: M = types.ClassType
```

types.ClassType 是旧风格类的元类，所以上述代码意味着，如果在当前模块的全局变量中和类的主体中名字 \_\_metaclass\_\_ 未被设置，创建出来的是一个旧风格的类。

class 声明会在执行的作用域中调用 M(n, b, d) 并将结果记录为一个变量，名字为 n。

因此，元类 M 会参与到类声明的执行中。对于新风格类而言，这个元类通常是 type，而旧风格类的元类则是 types.ClassType。可以设置并使用自己定制过的元类（通常 is type 的子类），当然也可能会认为这方面的内容有点过于超前。但是，可以尝试从另一个角度理解 class 声明，如下面这个例子：

```
class Someclass(Somebase):
    __metaclass__ = type
    x = 23
```

它其实等价于一个赋值声明：

```
Someclass = type('Someclass', (Somebase,), {'x': 23})
```

这样是不是能够让你更好地理解 class 声明的语义？

## 6.1 温标的转换

感谢：Artur de Sousa Rocha、Adde Nilsson

### 任务

你想在开氏温度 (Kelvin)、摄氏度 (Celsius)、华氏温度 (Fahrenheit) 以及兰金 (Rankine) 温度之间做转换。

## 解决方案

与其写一堆的函数来完成各种转换，不如将这些功能优雅地封装到一个类中：

```
class Temperature(object):
    coefficients = {'c': (1.0, 0.0, -273.15), 'f': (1.8, -273.15, 32.0),
                     'r': (1.8, 0.0, 0.0)}
    def __init__(self, **kwargs):
        # 默认是绝对(开氏)温度0, 但可接受一个命名的参数
        # 名字可以是k、c、f或r, 分别对应不同的温标
        try:
            name, value = kwargs.popitem()
        except KeyError:
            # 无参数 默认k=0
            name, value = 'k', 0
        # 若参数过多或参数不能识别, 报错
        if kwargs or name not in 'kcfr':
            kwargs[name] = value                      # 将其置回, 做诊断用
            raise TypeError, 'invalid arguments %r' % kwargs
        setattr(self, name, float(value))
    def __getattr__(self, name):
        # 将c、f、r的获取映射到k的计算
        try:
            eq = self.coefficients[name]
        except KeyError:
            # 未知名字, 提示错误
            raise AttributeError, name
        return (self.k + eq[1]) * eq[0] + eq[2]
    def __setattr__(self, name, value):
        # 将对k, c, f, r的设置映射到对k的设置; 并禁止其他的选项
        if name in self.coefficients:
            # 名字是c、f或r——计算并设置k
            eq = self.coefficients[name]
            self.k = (value - eq[2]) / eq[0] - eq[1]
        elif name == 'k':
            # 名字是k, 设置之
            object.__setattr__(self, name, value)
        else:
            # 未知名, 给出错误信息
            raise AttributeError, name
    def __str__(self):
        # 以易读和简洁的格式打印
        return "%s K" % self.k
    def __repr__(self):
        # 以详细和准确的格式打印
        return "Temperature(k=%r)" % self.k
```

## 讨论

在不同的度量衡和单位之间进行转换是一种所谓的“组合爆炸”(combinatorial explosion)

的任务：如果我们用看上去最直观的方式为每一种转换都提供一个函数，那么为了处理 n 个不同的单位，我们需要写  $n*(n-1)$  个函数。

Python 的类能够解读对属性的设置和读取，并进行即时解算。这种能力促成了一种更方便和优雅的结构，如解决方案所示。

在类的实现中，我们始终以某个单位或度量衡为标尺，在这个例子中我们选择的是开氏（绝对）温度。我们允许设置的值可以是四种属性名（'k', 'r', 'c', 'f', 即四种温标的简写），然后在内部计算并设置为正确的开氏温度值。同样，我们也允许以四种温标中的任意一种来获取温度值，即时解算出结果。（假设你将解决方案中的代码存为 te.py，并置入你的 Python 的 sys.path，就可以将它作为模块引入）比如：

```
>>> from te import Temperature
>>> t = Temperature(f=70)      # 70 F 是……
>>> print t.c                  #……比 21 C 高点
21.111111111
>>> t.c = 23                   # 23 C 是……
>>> print t.f                  # ……比 73 F 高点
73.4
```

在本例中使用 `__getattr__` 和 `__setattr__` 比使用命名的属性（named properties）更好，因为针对每种属性的计算模式都差不多（除了作为参考的 'k'），我们只需要使用在类中指定的字典 `self.coefficients`，并采用不同的折算系数即可。很重要的一点是，每当我们对一个属性进行设置时，`__setattr__` 就会被调用，因此，对于那些需要被记录在实例（本例中的 `__setattr__` 的实现将属性 k 的工作委托出去了）中的属性，我们必须把 `__setattr__` 委托给 `object` 来完成属性设置，而且对无法设置的属性也必须抛出 `AttributeError` 异常。另一方面，只有在获取某个不能以其他“普通”方式（比如在此例的类中，当访问属性 `k` 时，`__getattr__` 并未被调用。属性 `k` 被记录在实例中，因此可以以普通的方式获取）获取的属性时，`__getattr__` 才会被调用。当遇到无法访问的属性时，`__getattr__` 也必须抛出 `AttributeError` 异常。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中关于属性以及特殊方法 `__getattr__` 和 `__setattr__` 的文档。

## 6.2 定义常量

感谢：Alex Martelli

### 任务

你需要定义一些模块级别的变量（比如命名的常量），而且客户代码无法将其重新绑定。

## 解决方案

你可以把任何对象当做模块一样安装。将下列代码存为一个模块 `const.py`，并放入你的 Python 的 `sys.path` 指定的目录中：

```
class _const(object):
    class ConstError(TypeError): pass
    def __setattr__(self, name, value):
        if name in self.__dict__:
            raise self.ConstError, "Can't rebind const(%s)" % name
        self.__dict__[name] = value
    def __delattr__(self, name):
        if name in self.__dict__:
            raise self.ConstError, "Can't unbind const(%s)" % name
        raise NameError, name
import sys
sys.modules['__name__'] = _const()
```

现在，任何客户代码都可以导入 `const`，并将 `const` 模块的一个属性绑定一次，但仅能绑定一次，如下：

```
const.magic = 23
```

一旦某属性已经被绑定，程序无法将其重新绑定或者解除绑定：

```
const.magic = 88      # 抛出 const.ConstError
del const.magic      # 抛出 const.ConstError
```

## 讨论

在 Python 中，变量可以被随意绑定，而模块不同于类，你无法定义一个特殊的像 `__setattr__` 一样的方法来阻止对它的重新绑定。一个简单的解决办法是像安装模块一样安装对象。

Python 并没有对 `sys.modules` 的条目做类型检查以确保它们是模块对象。因此，可以在那里安装一个对象，并利用其特殊的属性存取的方法（比如在 `__getattr__` 等方法中阻止对属性的处理或者重新绑定），而且客户代码仍然可以通过 `import somename` 来访问这个对象。还可以在更具有 Python 风格的单例模式用法（见 6.16 节）中看到这种技巧。

本节解决方案可以确保：当一个模块级别的名字被首次绑定到了一个对象，以后它会始终恒定不变地绑定到同一个对象。本节并没有处理某个对象的不可变性，那是另一个不同的主题。修改一个对象和重新绑定一个名字是完全不同的概念，这些概念在 4.1 节中有详细的解释。数字、字符串和元组都是不可改变的：如果你将 `const` 中的一个名字绑定到这样的一个对象，那么不仅该名字会始终绑定到此对象，而且对象的内容也会永远保持不变，因为这个对象具有不可变性。但其他对象比如列表和字典则是可变的：如果绑定 `const` 中的一个名字到一个列表对象，虽然这个名字会始终保持绑定到该

列表，但列表的内容却可能发生变化（比如其中的元素可能被重新绑定或者解除绑定，我们通过 `append` 方法也可以给它增加更多的元素，等等）。

关于怎样给可改变的对象制作一个“只读”的封装层的内容，请参考 6.5 节。可以选择用类 `_const` 的 `__setattr__` 方法来完成这个封装。假设将 6.5 节中的代码存为 `ro.py` 模块，并放入你的 Python 的 `sys.path` 目录。那么，需要在 `const.py` 模块的开头增加一句：

```
import ro
```

并修改类 `_const` 的 `__setattr__` 方法中的赋值语句 `self.__dict__[name] = value`，将其改为：

```
self.__dict__[name] = ro.Readonly(value)
```

现在，当你将 `const` 中的属性设置为某值时，实际上只是把只读的封装绑定到了那个值。我们通过这个值（对象）的其他的引用，调用相关的修改函数或方法仍然能够修改它，当然通过“伪模块”`const` 的属性无法实现修改。如果你想避免这种“意外地通过其他的引用修改”的情况，需要使用一个拷贝，如同 4.1 节所解释的那样，这样才能保证没有任何其他引用指向那个值。为了达到这个目的，必须在 `const.py` 模块开始的位置编写这样的代码：

```
import ro, copy  
并且将类 _const 中的 __setattr__ 方法改成:  
self.__dict__[name] = ro.Readonly(copy.deepcopy(value))
```

如果你够偏执，你可能还会想用 `copy.deepcopy` 替代最后一段代码中的 `copy.copy`。但是，过度谨慎会消耗更多的内存，并损失一些性能。必须根据自己特定的应用需求来仔细衡量这样的举措是否真的有必要。无论你最终的决定是什么，Python 已经向我们证明了，它提供了足够的工具和机制来实现我们所需要的不可变性。

本节展示的 `_const` 类，从某种意义来说，它可以被看做是 6.3 节中展示的 `NoNewAttrs` 类的“补充”。`_const` 类确保已经被绑定的属性无法被重新绑定，但允许你随意绑定新属性；而 `NoNewAttrs` 则相反，可以随意地重新绑定那些已经被绑定的属性，但是却禁止绑定任何新的属性。

## 更多资料

6.5 节，6.13 节，4.1 节，*Library Reference* 和 *Python in a Nutshell* 中关于模块对象、`import` 声明，以及内建模块 `sys` 的属性的文档。

## 6.3 限制属性的设置

感谢： Michele Simionato

## 任务

通常情况下，Python 允许你随意给类和类实例增加属性。但对于某些特定的类，你却希望这种自由受到限制。

## 解决方案

特殊方法`__setattr__`会解读对属性的设置操作，它让你有机会限制新属性的添加。一种优雅的实现方法是写一个类和一个简单的自定义元类，再加上一个封装函数，像下面这样：

```
def no_new_attributes(wrapped_setattr):
    """ 尝试添加新属性时，报错
        但允许已经存在的属性被随意设置
    """
    def __setattr__(self, name, value):
        if hasattr(self, name):      # 非新属性，允许
            wrapped_setattr(self, name, value)
        else:                      # 新属性，禁止
            raise AttributeError("can't add attribute %r to %s" % (name,
self))
        return __setattr__
    class NoNewAttrs(object):
        """ NoNewAttrs 的子类会拒绝新属性的添加
            但允许已存在的属性被赋予新值
        """
        # 向此类的实例添加新属性的操作被屏蔽
        __setattr__ = no_new_attributes(object.__setattr__)
    class __metaclass__(type):
        """ 一个简单的自定义元类，禁止向类添加新属性 """
        __setattr__ = no_new_attributes(type.__setattr__)
```

## 讨论

由于某些原因，有时你会想要限制 Python 的动态能力。比如，对于某个特定的类或其实例，有时你可能会意外地给它设置了新的属性，而你不希望这种情况发生。本节的解决方案介绍了怎样实现这样的限制。其要点是，不要为这个目的使用`__slots__`: `__slots__`应该被用于其他的任务（比如，你的类有一些固定的属性并有数量很多的实例，利用`__slots__`可以避免每个实例都拥有一个字典，从而节省很多内存）。`__slots__`应该被用来做它适合做的事，如果你试图用它来完成本节的任务，会遇到很多限制（见 6.7 节中一个使用`__slots__`来节省内存的例子）。

需要注意的是，本节解决方案给出的方法阻止了在运行时添加属性，而且不仅对类实例有效，对类本身也适用，这应该归功于那个简单的自定义元类。当你想要限制一些意外添加属性的行为时，你通常会希望类和它的每个实例都受到限制。另一方面，类

和类实例已经存在的属性则可以被随意地设置新值。

下面给出一个使用这个类的例子：

```
class Person(NoNewAttrs):
    firstname = ''
    lastname = ''
    def __init__(self, firstname, lastname):
        self.firstname = firstname
        self.lastname = lastname
    def __repr__(self):
        return 'Person(%r, %r)' % (self.firstname, self.lastname)
me = Person("Michere", "Simionato")
print me
# 输出: Person('Michere', 'Simionato')
# firstname 的值是错的, 能修正吗? 当然没问题!
me.firstname = "Michele"
print me
# 输出: Person('Michele', 'Simionato')
```

从 NoNewAttrs 继承将迫使你只能在类的内部“声明”那些允许被设置的属性。如果你想在进一步使用中设置一个“未声明”的属性，只会得到一个 `AttributeError`：

```
try: Person.address = ''
except AttributeError, err: print 'raised %r as expected' % err
try: me.address = ''
except AttributeError, err: print 'raised %r as expected' % err
```

从某种意义上讲，`NoNewAttr` 的子类以及它们的实例其实很像 Java 或 C++ 的类与实例，而不像普通的 Python 类和实例。因此，当你在用 Python 构建某种原型而且最后会用其他动态性逊于 Python 的语言来重新编写代码的时候，本节提供的方法会很适合你的需求。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中关于特殊方法 `__setattr__` 和自定义元类的文档；6.18 节中使用 `__slots__` 来节省内存的例子；6.2 节提供的类可作为本节的类的有益补充。

## 6.4 链式字典查询

感谢：Raymond Hettinger

### 任务

你有一些映射（通常是 `dict`），而且你想在这些映射中以一种链式的方式进行查询（首先在第一个映射中查找，如果键不在其中，尝试第二个，以此类推）。准确地说，你需

要的是一个单一的映射，这个映射跨越了许多“虚拟融合”的映射对象，而且可以在其中以指定的优先级顺序进行查询，这样的结构可以在使用上带来很多方便。

## 解决方案

映射是字典的概括和抽象的说法，一个映射提供的接口非常类似于字典，但其实现却可能完全不同。所有的字典都是映射，但反过来不成立。在这个需求中，需要实现的是一个映射，在内部这个映射可以将任务按顺序委托给其他映射。用一个类来封装这个功能是很好的选择：

```
class Chainmap(object):
    def __init__(self, *mappings):
        # 记录映射的序列
        self._mappings = mappings
    def __getitem__(self, key):
        # 在序列的字典中查询
        for mapping in self._mappings:
            try:
                return mapping[key]
            except KeyError:
                pass
        # 没有在任何字典中找到 key, 所以抛出 KeyError 异常
        raise KeyError, key
    def get(self, key, default=None):
        # 若 self[key] 存在则返回之, 否则返回 default
        try:
            return self[key]
        except KeyError:
            return default
    def __contains__(self, key):
        # 若 key 在 self 中返回 True, 否则返回 False
        try:
            self[key]
            return True
        except KeyError:
            return False
```

举个例子，你现在就可以实现 Python 用来查询名字的机制：先在局部变量中查找，然后（如果没有找到）再到全局变量中查找，最后（如果还没有找到）到内置量中查找：

```
import __builtin__
pylookup = Chainmap(locals(), globals(), vars(__builtin__))
```

## 讨论

Chainmap 依赖于它封装的映射的很少一部分功能：内部的所有映射必须允许索引操作（比如提供了特殊方法 `__getitem__`），而且当索引操作遇到了不在任何一个内部映射中

的键时，必须抛出标准的 `KeyError` 异常。`Chainmap` 的实例提供同样的行为方式，但增加了一个方便的 `get` 方法——在 4.9 节有介绍，以及特殊方法 `__contains__`（它可以让你方便地检查某个键 `k` 是否在 `Chainmap` 的实例 `c` 中，`if k in c`）。

除了很明显的只读限制，`Chainmap` 类还有其他的限制，从本质上说，即使只读是我们设计上的选择，它也不是一个“完全的映射”。你可以通过继承 `DictMixin` 类（在标准库的 `UserDict` 模块中）并提供少量的关键方法（`DictMixin` 实现了其他的方法）将一个部分映射改变成“完全映射”。下面是基于 `Chainmap` 和 `UserDict.DictMixin` 的一个完全（只读）映射：

```
import UserDict
from sets import Set
class FullChainmap(Chainmap, UserDict.DictMixin):
    def copy(self):
        return self.__class__(self._mappings)
    def __iter__(self):
        seen = Set()
        for mapping in self._mappings:
            for key in mapping:
                if key not in seen:
                    yield key
                    seen.add(key)
    iterkeys = __iter__
    def keys(self):
        return list(self)
```

除了 `Chainmap` 的对映射的要求，`FullChainmap` 类又对封装的映射增加了一项要求：映射必须是可迭代的。同时我们也注意到，如果我们从 `DictMixin` 继承，`Chainmap` 中的对 `get` 和 `__contains__` 的实现就变得多余了（虽然是无害的），因为 `DictMixin` 已经实现了这两个比较底层的方法（当然还有其他方法）。参看 5.14 节中更多的关于 `DictMixin` 的细节。

## 更多资料

4.9 节；5.14 节；*Library Reference* 和 *Python in a Nutshell* 中的映射类型。

## 6.5 继承的替代方案——自动托管

感谢：Alex Martelli、Raymond Hettinger

### 任务

你需要从某个类或者类型继承，但是需要对继承做一些调整。比如，需要选择性地隐藏某些基类的方法，而继承并不能做到这一点。

## 解决方案

继承是很方便的，但它并不是万用良药。比如，它无法让你隐藏基类的方法或者属性。而自动托管技术则提供了一种很好的选择。假设需要把一些对象封起来变成只读对象，从而避免意外修改的情况。那么，除了禁止属性设置的功能，还需要隐藏修改属性的方法。下面我们给出一个办法：

```
# 同时支持 2.3 和 2.4
try: set
except NameError: from sets import Set as set
class ROError(AttributeError): pass
class Readonly: # 这里并没有用继承，我们会在后面讨论其原因
    mutators = {
        list: set(''__delitem__ __delslice__ __iadd__ __imul__'
                  '__setitem__ __setslice__ append extend insert
                  pop remove sort''.split()),
        dict: set(''__delitem__ __setitem__ clear pop popitem
                  setdefault update''.split(),
                  )
    }
    def __init__(self, o):
        object.__setattr__(self, '_o', o)
        object.__setattr__(self, '_no', self.mutators.get(type(o), ()))
    def __setattr__(self, n, v):
        raise ROError, "Can't set attr %r on RO object" % n
    def __delattr__(self, n):
        raise ROError, "Can't del attr %r from RO object" % n
    def __getattr__(self, n):
        if n in self._no:
            raise ROError, "Can't get attr %r from RO object" % n
        return getattr(self._o, n)
```

通过修改 mutators，即 Readonly.mutators[sometype] = the\_mutators，还可以轻松地增加其他需要处理的类型。

## 讨论

自动托管是一种强大而通用的技术。在本节的例子中，通过使用这个技术我们能得到和类继承几乎完全一样的效果，同时还能隐藏一些名字。我们在任务中使用这个模拟的子类将一些对象封装起来，使之变成只读对象。它的性能也许不如真正的继承，但另一方面，作为补偿，我们获得了更好的灵活度和更精细的粒度控制。

基本的想法是，我们的类的每个实例都含有我们想要封装的类型的实例。每当客户代码试图从我们的类的实例中获取属性时，除非该属性已经在类中被定义了（比如定义在 Readonly 类的 mutators 字典中），否则 \_\_getattr\_\_ 在完成检查之后，会透明地将这个请求转交给被封装的实例。在 Python 中，方法同样也是属性，访问的方式也一样，所以无论是访问方法还是属性，代码无须改变。用来访问属性的 \_\_getattr\_\_ 方法同时也可用于访问方法。

解决方案的注释没有解释不使用继承的原因，这里我们会给出一点解释。这种基于`__getattr__`的方式也可用于特殊方法，但仅对旧风格类的实例有效。在新的对象模型中，Python 操作直接通过类的特殊方法来进行，而不是实例的。关于这个问题的更多内容可以在 6.6 节和 20.8 节中看到。本节采用的方案——让  `Readonly` 类成为旧风格类，从而避开这个问题，并把相关内容留到其他章节——在真实的生产代码中是不值得推荐的。我在这里用仅仅是为了控制篇幅，同时避免重复其他章节的内容。

`__setattr__`的角色类似于`__getattr__`，当客户代码设置实例的属性时，它就会被调用；这个任务要求某些属性为只读，我们只需简单地禁止属性访问操作即可。记住，要在方法的代码编写中避免激发对`__setattr__`的调用，在有`__setattr__`的类的方法中你不应该使用`self.n = v`这样的语句。最简单的是直接把设置操作委托给类 `object`，如同类  `Readonly` 在它的`__init__`方法中所做的那样。方法`__delattr__`完成了最后拼图，它会处理那些试图从实例中删除属性的操作。

以自动托管方式完成的封装并不适用于采用了类型检查的客户代码或者框架代码。在那种情况下，客户代码或框架代码完全破坏了多态性，代码本身应该是被重写的。记住不要在你自己的代码中使用类型检查，因为你可能根本无须那么做。见 6.13 节提供的更好的选择。

在 Python 的老版本中，自动托管的流行程度甚至比现在还高，那是因为当时 Python 不支持从内建的类型继承。而对于现在的 Python，从内建类型继承是允许的，因此自动托管就用得不那么频繁了。不过，自动托管仍然具有它的地位——它只是稍微远离了聚光灯一点点。托管比继承更加灵活，而有时这种灵活是无价的。除了选择性地托管（从而高效地实现了某些属性的“隐藏”），一个对象还可以在不同的时间托管给不同的子对象，或者一次托管给多个子对象，继承无法提供任何能够与之相比的特性。

下面给出托管给多个特定子对象的例子。假设你有个类，提供各种“转发方法”，比如：

```
class Pricing(object):
    def __init__(self, location, event):
        self.location = location
        self.event = event
    def setlocation(self, location):
        self.location = location
    def getprice(self):
        return self.location.getprice()
    def getquantity(self):
        return self.location.getquantity()
    def getdiscount(self):
        return self.event.getdiscount()
    and many more such methods
```

继承很明显不适用，因为 `Pricing` 的实例需要托管给特定的 `location` 和 `event` 实例，这些实例在初始化阶段传入而且可能会被修改。自动托管的补救方法是：

```

class AutoDelegator(object):
    delegates = ( )
    do_not_delegate = ( )
    def __getattr__(self, key):
        if key not in self.do_not_delegate:
            for d in self.delegates:
                try:
                    return getattr(d, key)
                except AttributeError:
                    pass
        raise AttributeError, key
class Pricing(AutoDelegator):
    def __init__(self, location, event):
        self.delegates = [location, event]
    def setlocation(self, location):
        self.delegates[0] = location

```

在此例中，我们没有托管属性的删除和设置，而只是托管了属性的获取（还有一些非特殊方法）。当然，这种方式只有在我们想要托管的各个对象的方法（以及其他属性）不会互相干扰的情况下才会充分有效；比如，`location`最好不要有个`getdiscount`方法，否则它会抢先进行方法的托管，而此方法原本应该是由`event`对象来执行的。

如果一个需要大量托管的类涉及这种问题，它可以简单地定义一些对应的方法，这是因为只有在用别的方式无法找到属性和方法时，`__getattr__`才会介入。而通过`do_not_delegate`属性还可以隐藏托管对象的一些属性和方法，而且它也可以被子类改写。举个例子，如果类`Pricing`想要隐藏一个叫做`setdiscount`的方法，此方法由`event`提供，做一点点修改就可以了：

```

class Pricing(AutoDelegator):
    do_not_delegate = ('set_discount',)

```

其余部分则与前面代码片段相同。

## 更多资料

6.13 节；6.6 节；20.8 节；*Python in a Nutshell* 的关于 OOP 的章节；PEP 253 (<http://www.python.org/peps/pep-0253.html>) 中关于 Python 当前（新风格）对象模型的细节。

## 6.6 在代理中托管特殊方法

感谢：Gonçalo Rodrigues

### 任务

在新风格对象模型中，Python 操作其实是在类中查找特殊方法的（而不是在实例中，那是经典对象模型的处理方式）。现在，需要将一些新风格的实例包装到代理类中，此

代理可以选择将一些特殊方法委托给内部的被包装对象。

## 解决方案

你需要即时地生成各个代理类。如下：

```
class Proxy(object):
    """ 所有代理的基类 """
    def __init__(self, obj):
        super(Proxy, self).__init__(obj)
        self._obj = obj
    def __getattr__(self, attrib):
        return getattr(self._obj, attrib)
def make_binder(unbound_method):
    def f(self, *a, **k): return unbound_method(self._obj, *a, **k)
    # 仅 2.4: f.__name__ = unbound_method.__name__
    return f
known_proxy_classes = {}
def proxy(obj, *specials):
    '''能够委托特殊方法的代理的工厂函数'''
    # 我们手上有合适的自定义的类吗?
    obj_cls = obj.__class__
    key = obj_cls, specials
    cls = known_proxy_classes.get(key)
    if cls is None:
        # 我们手上没有合适的类, 那就现做一个
        cls = type("%sProxy" % obj_cls.__name__, (Proxy,), {})
        for name in specials:
            name = '__%s__' % name
            unbound_method = getattr(obj_cls, name)
            setattr(cls, name, make_binder(unbound_method))
        # 缓存之以供进一步使用
        known_proxy_classes[key] = cls
    # 实例化并返回需要的代理
    return cls(obj)
```

## 讨论

代理和自动托管都是 Python 中的玩具，这得归功于`__getattr__`的机制。在查询任何属性时（包括方法，Python 并不区分两者），Python 都会自动调用`__getattr__`。

在旧风格（经典）对象模型中，`__getattr__`同样适用于特殊方法，这些方法常常被认为是 Python 操作的一部分。在使用中也需要当心错误地提供一个我们不想提供的特殊方法。而现在，在新代码中使用新风格的对象模型成为推荐方式：速度更快，更符合规定，特性也更丰富。当你从`object`或任何内建类型派生子类时就得到了新风格的类。也许几年后的某一天，Python 3.0 将彻底地剔除经典对象模型以及其他一些仅仅是为了保证向后兼容的特性。（参看 <http://www.python.org/peps/pep-3000.html> 提供的关于

Python 3.0 的计划细节，几乎都是以简化语言为主，而不是新增功能。)

在新风格对象模型中，Python 操作并不会在运行时查找特殊方法：它们依赖于类对象的“槽”(slots)。这些槽会在类对象被创建或者修改时更新。因此，对于一个代理对象，如果它要把特殊方法托管给被封装的对象，它本身必须属于某个量身定做的类。幸好，如同解决方案的代码所示，在 Python 中，凭空创造并实例化类是一件很简单的事情。

在本节，我们不使用任何高级的 Python 概念，比如自定义元类和描述符。事实上，每个代理都是由一个工厂函数 proxy 创建的，该函数接受一个封装的对象以及要托管的特殊方法的名字作为参数（除去前面和后面的两个下划线符号）。如果你把解决方案中的代码存为一个文件 proxy.py 并放入你的 Python 的 sys.path 中，就可以用下面的方法在 Python 解释器中使用它：

```
>>> import proxy
>>> a = proxy.proxy([ ], 'len', 'iter')    # 只托管 len 和 iter
>>> a                                         # __repr__ 未被托管
<proxy.listProxy object at 0x0113C370>
>>> a.__class__
<class 'proxy.listProxy'>
>>> a._obj
[ ]
>>> a.append                                # 所有的非特殊方法都被托管了
<built-in method append of list object at 0x010F1A10>
```

由于 \_\_len\_\_ 被托管了，于是 len(a) 像预期那样工作：

```
>>> len(a)
0
>>> a.append(23)
>>> len(a)
1
```

由于 \_\_iter\_\_ 被托管了，for 循环也如同预期那样工作，和通过内建的 list、sum、max 等操作执行的循环一样：

```
>>> for x in a: print x
...
23
>>> list(a)
[23]
>>> sum(a)
23
>>> max(a)
23
```

不过，由于 \_\_getitem\_\_ 没有被托管，a 无法进行索引或切片操作：

```
>>> a.__getitem__
<method-wrapper object at 0x010F1AF0>
>>> a[1]
```

```
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
    TypeError: unindexable object
```

函数 proxy 使用的是以前创建的类的“缓存”，即全局字典 known\_proxy\_classes，它以被封装的对象的类和被托管的特殊方法的名字为键。如果要生成一个新类，proxy 就调用内建的 type，将新类的名字作为一个参数传入（在被包装的对象的类名后加了个“Proxy”），类 Proxy 作为唯一的基类，是一个“空”的类字典（它还没有加入任何属性）。基类 Proxy 进行初始化处理和对普通属性的查询的托管。然后，工厂函数 proxy 循环处理被托管的特殊方法名：对每一个名字，它都从被封装对象的类获得未绑定方法，并将其作为一个属性赋给闭包 make\_binder 中的新类。当遇到对这些未绑定方法的调用时，make\_binder 会提供一个适合的参数（比如被封装对象本身，self.\_obj）。

一旦完成了新类的准备，proxy 将其存入 known\_proxy\_classes，并以适当的键标记之。最后，无论类是被创建或者从 known\_proxy\_classes 中获取的，proxy 都会使用被封装对象作为唯一参数，将其实例化，然后返回最后的代理实例。

## 更多资料

6.5 节中关于自动托管的细节；6.9 节提供的另一个即时生成类的例子（使用的是 class 声明而不是调用 type）。

## 6.7 有命名子项的元组

感谢：Gonçalo Rodrigues、Raymond Hettinger

### 任务

Python 元组可以很方便地被用来将信息分组，但是访问每个子项都需要使用数字索引，所以这种用法有点不便。你希望能够创建一种可以通过名字属性访问的元组。

### 解决方案

工厂函数是生成符合要求的元组的子类的最简单方法：

```
# 若在 2.4 中可使用 operator.itemgetter，若在 2.3 中则需要实现 itemgetter
try:
    from operator import itemgetter
except ImportError:
    def itemgetter(i):
        def getter(self): return self[i]
        return getter
def superTuple(typename, *attribute_names):
    """创建并返回拥有名字属性的元组子类"""
    # 给子类适合的__new__和__repr__特殊方法
```

```

nargs = len(attribute_names)
class supertup(tuple):
    __slots__ = ()          # 我们不需要每个实例提供一个字典，节省内存
    def __new__(cls, *args):
        if len(args) != nargs:
            raise TypeError, '%s takes exactly %d arguments (%d given)' % (typename, nargs, len(args))
        return tuple.__new__(cls, args)
    def __repr__(self):
        return '%s(%s)' % (typename, ', '.join(map(repr, self)))
    # 给我们的元组子类添加一些键
    for index, attr_name in enumerate(attribute_names):
        setattr(supertup, attr_name, property(itemgetter(index)))
    supertup.__name__ = typename
    return supertup

```

## 讨论

你常常需要用元组传递数据，就像 C 语言中的结构，或者其他语言中的简单记录一样。但你总是需要记住每个数字索引对应到哪个域并需要用数字索引来访问数据，这种用法令人心烦。一些 Python 标准库模块，比如 time 和 os，在旧版 Python 中总是返回元组，让这种烦恼更是雪上加霜。事实上，如果有一个类似于元组的类型，可以允许你通过名字访问值，就像通过属性名访问一样，同时也提供了数字索引的访问方式，那一定会是个很受欢迎的方案。本节展示了如何编程来实现这样的效果，本质上我们只需要自动地创建一个元组的定制子类。

要创建一个新的定制类型有很多方法；定制元类对于此类任务通常是最好的方式。但在本节中，一个简单的工厂函数足矣，而且你也应该避免使用超出需求的技术。下面展示怎样在代码中使用工厂函数 superTuple，此处假设你已经将解决方案中的代码存为一个名为 supertuple.py 的模块，并将它放入了你的 Python 的 sys.path 中：

```

>>> import supertuple
>>> Point = supertuple.superTuple('Point', 'x', 'y')
>>> Point
<class 'supertuple.Point'>
>>> p = Point(1, 2, 3)                      # 故意给错误的数字
Traceback (most recent call last):
  File "", line 1, in ?
  File "C:\Python24\Lib\site-packages\superTuple.py", line 16, in __new__
    raise TypeError, '%s takes exactly %d arguments (%d given)' %
TypeError: Point takes exactly 2 arguments (3 given)
>>> p = Point(1, 2)                          # 这次给正确的数字
>>> p
Point(1, 2)
>>> print p.x, p.y
1 2

```

函数 `superTuple` 的实现相当直接易懂。为了创建新的子类，`superTuple` 使用了一个 `class` 声明，在声明的主体部分，它定义了三个特殊方法：一个“空”的`__slots__`（为了节省内存，我们的 `supertuple` 类并不需要为每个实例提供一个字典）；一个`__new__`方法，这是为了在托管给 `tuple.__new__` 之前先检查参数的个数；以及一个`__repr__`方法。当一个新的类对象被创建之后，我们为每个我们需要的命名属性设置了一个 `property`。每个这样的 `property` 都只是一个“getter”，这是因为我们的 `supertuples` 就像元组一样，是不可改变的——没有对值域进行设置的方法。最后，我们设置新类的名字并返回类对象。

通过简单地调用标准库模块 `operator` 的内建的 `itemgetter` 就可以创建这些 `getter` 方法。不过由于 `operator.itemgetter` 在 Python 2.4 中才被引入，在模块的开头我们需要确定用什么方式才能获得可用的 `itemgetter`，即使是在 Python 2.3 中，我们也可以提供自己的实现。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 文档关于 `property`、`__slots__`、`tuple`，以及特殊方法`__new__`和`__repr__`；(Python 2.4) 模块 `operator` 的 `itemgetter` 函数。

## 6.8 避免属性读写的冗余代码

感谢：Yakov Markovitch

### 任务

你的类会用到某些 `property` 实例，而 `getter` 或者 `setter` 都是一些千篇一律的获取或者设置实例属性的代码。你希望只用指定属性名，而不用写那些非常相似的代码。

### 解决方案

需要一个工厂函数，用它来处理那些 `getter` 或 `setter` 的参数是字符串的情况，并将正确的参数封装到一个函数中，然后将其余的工作委托给 Python 内建的 `property`：

```
def xproperty(fget, fset, fdel=None, doc=None):
    if isinstance(fget, str):
        attr_name = fget
        def fget(obj): return getattr(obj, attr_name)
    elif isinstance(fset, str):
        attr_name = fset
        def fset(obj, val): setattr(obj, attr_name, val)
    else:
        raise TypeError, 'either fget or fset must be a str'
    return property(fget, fset, fdel, doc)
```

## 讨论

Python 内建的 `property` 非常有用，但它也有一点小小的烦人之处（有 Delphi 经验的开发者能比较容易地看出这一点）。尤其是当你同时需要一个 `setter` 和一个 `getter` 时，其中的一个需要执行一些额外的代码；而另一个则只需简单地读取或者写入实例的属性。此时，`property` 需要两个函数作为它的参数。其中的一个函数就是所谓的“样板代码”（即重复的长篇大论的冗余代码，既无趣也容易滋生 bug）。

举个例子：

```
class Lower(object):
    def __init__(self, s=''):
        self.s = s
    def _getS(self):
        return self._s
    def _setS(self, s):
        self._s = s.lower()
    s = property(_getS, _setS)
```

方法 `_getS` 就是样板代码，但你仍需要编写这些代码，因为你要将它传递给 `property`。使用本节的方案，可以让你的代码变得更简洁，同时丝毫不改变原意：

```
class Lower(object):
    def __init__(self, s=''):
        self.s = s
    def _setS(self, s):
        self._s = s.lower()
    s = xproperty('_s', _setS)
```

在这个小例子中，这点简化看起来没省掉多少代码，但是如果在一个大项目的所有代码中应用这种简化，省去的冗余代码将极为可观。

本节解决方案中的工厂函数 `xproperty` 的实现对参数有比较严格的要求：它需要你同时传入 `fget` 和 `fset`，而且其中之一必须是一个字符串。要求两者都是字符串没有什么用处；如果两者都不是字符串，或者你只需要其中的一个，可以（且应当）直接使用内建的 `property`。但用 `xproperty` 预先检查会更好，它既不会带来什么大的性能损失也不会造成任何功能上的损失。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 文档中关于内建 `property` 的资料。

## 6.9 快速复制对象

感谢：Alex Martelli

## 任务

为了使用 `copy.copy`, 需要实现特殊方法`__copy__`。而且你的类的`__init__`比较耗时, 所以你希望能够绕过它并获得一个“空的”未初始化的类实例。

## 解决方案

下面的解决方案可同时适用于新风格和经典类:

```
def empty_copy(obj):
    class Empty(obj.__class__):
        def __init__(self): pass
    newcopy = Empty()
    newcopy.__class__ = obj.__class__
    return newcopy
```

你的类可以使用这个函数来实现`__copy__`:

```
class YourClass(object):
    def __init__(self):
        assume there's a lot of work here
    def __copy__(self):
        newcopy = empty_copy(self)
        copy some relevant subset of self's attributes to newcopy
        return newcopy
```

下面是用法示例:

```
if __name__ == '__main__':
    import copy
    y = YourClass()      # 很显然, __init__会被调用
    print y
    z = copy.copy(y)     # ...不调用__init__
    print z
```

## 讨论

如同 4.1 节的讨论, 当你进行赋值操作时, Python 并不会暗中对对象进行复制操作, 这是一个很好的处理, 因为这样的机制更快、更灵活, 而且也具有语义的一致性。当需要复制时, 要明确地提出请求, 通常使用 `copy.copy` 函数, 它知道怎样复制内建的类型, 对于自定义的对象也具有默认的行为方式, 如果你想定制这个复制过程, 可以定义类的`__copy__`特殊方法。如果你的类实例是不允许复制的, 可以定义一个`__copy__`并抛出一个 `TypeError` 异常。大多数情况下, 你只需要让 `copy.copy` 按照默认的机制工作, 就可以很轻松地获得克隆能力。这确实是很棒的设计, 其他很多语言会强迫你实现一个特殊的 `clone` 方法来完成实例克隆。

如果`__init__`是一个耗时的操作, `__copy__`常常需要以一个“空”实例开始, 从而绕

开`__init__`。最简单和通用的方法是使用 Python 提供的直接修改实例的类的能力：在本地的空类中创建一个新对象，然后设置此对象的`__class__`属性，如代码所示。对旧风格（经典）类而言，从`obj.__class__`继承`Empty`类显得比较多余（但也没什么害处），但继承机制使得本节的方案对于各种对象包括经典的和新风格的类（包括内建的和扩展类型），都具有很好的兼容性。一旦你选择从`obj`的类继承，必须重写`Empty`类中的`__init__`，否则就无法实现本节方案的初衷。重写，意味着`obj`类的`__init__`不会被执行，这是因为在 Python 中，父类的初始化方法并不会自动执行。

一旦得到了一个符合要求的类的“空”对象，就需要复制`self`属性的一个子集。当需要所有属性时，最好就不要自定义`__copy__`，因为`copy.copy`的默认行为就是复制实例的所有属性。除非你除了复制所有属性之外还想做更多的工作，在本例中，下面两种复制所有属性的方式都是可行的：

```
newcopy.__dict__.update(self.__dict__)
newcopy.__dict__ = dict(self.__dict__)
```

新风格类的实例并不一定会在`__dict__`中保存它的所有状态，所以你可能需要复制一些与类相关特定的状态。

基于标准模块`new`的方式对于经典类和新风格类无法做到完全透明，而且`__new__`静态方法也不能生成一个空的实例——后者只在新风格类中存在。不过，本节的方案能够解决这些问题。

实现`__copy__`的一种好的替代方式是实现`__getstate__`和`__setstate__`方法：这些特殊方法以明确的和原生的方式定义了你的对象的状态，绕过了`__init__`。另外，它们也支持类实例的序列化：见 7.4 节中关于这些方法的更多信息。

到此为止我们讨论的是浅拷贝，这也是你大多数时候需要的复制方式。使用浅拷贝时，虽然你的对象被复制了，但是它指向的很多对象（内部的属性或者子项）却并未被复制，所以新复制的对象和原对象指向相同的属性和子项——这是一种轻量级的快速操作。而深拷贝则是一种重量级的深度操作，它根据对象的指向图谱逐一地完成对所有对象的拷贝。可以调用`copy.deepcopy`完成对一个对象的深拷贝。如果你想定制你的类实例的深拷贝方式，需要定义一个特殊方法`__deepcopy__`：

```
class YourClass(object):
    ...
    def __deepcopy__(self, memo):
        newcopy = empty_copy(self)
        # 使用 copy.deepcopy(self.x, memo) 来获得 self 的相关属性
        # 元素的子集的深拷贝，并放入 newcopy 中。
        return newcopy
```

如果决定要实现`__deepcopy__`，记住要遵守 Python 标准库模块`copy`的文档定义的 memoization 协议——应该复制有第二个参数的`copy.deepcopy`要求的全部属性和子项。

同一个 memo 字典，还会被传递给`__deepcopy__`方法。另外，实现`__getstate__`和`__setstate__`也是很好的选择，因为这些方法同样支持深拷贝：Python 会处理并复制那些由`__getstate__`返回的“状态”对象，然后传递给一个新的，空实例的`__setstate__`方法。参看 7.4 节关于这些特殊方法的更多内容。

## 更多资料

4.1 节中关于深拷贝和浅拷贝的内容；7.4 节中关于`__getstate__`和`__setstate__`的内容；*Library Reference* 和 *Python in a Nutshell* 中关于 copy 模块的章节。

# 6.10 保留对被绑定方法的引用且支持垃圾回收

感谢：Joseph A. Knapka、Frédéric Jolliton、Nicodemus

## 任务

你想保存一些指向被绑定方法的引用，同时还需要让关联的对象可以被垃圾收集机制处理。

## 解决方案

弱引用（weak reference）（弱引用在一个对象处于生存期时指向该对象，但如果没有任何其他的引用指向该对象时，这个对象不会被保留）在一些高级编程方法中是一个重要的工具。Python 标准库的 `weakref` 模块允许我们使用弱引用。

然而，`weakref` 的功能无法被直接应用于被绑定方法，除非你采取了一些特殊的预防措施。为了让一个对象在有引用指向其被绑定方法的时候能被垃圾回收机制处理，需要做一些封装工作。将下列代码存为一个名为 `weakmethod.py` 的文件，并放入你的 Python 的 `sys.path` 目录中：

```
import weakref, new
class ref(object):
    """能够封装任何可调用体，特别是被绑定方法，而且被绑定方法仍然能被回收处理。与此同时，提供一个普通的弱引用的接口"""
    def __init__(self, fn):
        try:
            # 尝试获得对象、函数和类
            o, f, c = fn.im_self, fn.im_func, fn.im_class
        except AttributeError:
            # 非被绑定方法
            self._obj = None
            self._func = fn
            self._clas = None
        else:
            # 绑定方法
            self._obj = o
            self._func = f
            self._clas = c
```

```

        if o is None: self._obj = None           # …实际上没绑定
        else: self._obj = weakref.ref(o)        # …确实绑定了
        self._func = f
        self._clas = c
    def __call__(self):
        if self._obj is None: return self._func
        elif self._obj() is None: return None
        return new.instancemethod(self._func, self._obj(), self._clas)

```

## 讨论

一个正常的被绑定方法拥有一个指向此方法所属对象的强引用。这意味着除非这个被绑定方法被消灭掉，否则该对象不能被当做垃圾重新回收。

```

>>> class C(object):
...     def f(self):
...         print "Hello"
...     def __del__(self):
...         print "C dying"
...
>>> c = C()
>>> cf = c.f
>>> del c      # c 眨巴眨巴眼睛活得好好的...
>>> del cf     # …直到我们干掉了被绑定的方法，它才终于安心地去了
C dying

```

很多时候这种行为方式很方便，但有时它却达不到你想要的效果。比如，你想实现一个事件分发的系统，你可能不希望由于一个事件处理器（比如一个被绑定函数）的存在，整个关联对象都无法被回收。因而一个很自然的想法是使用弱引用。不过，一个普通的指向被绑定方法的 `weakref.ref` 并不会按照我们的期望工作，那是因为被绑定方法是一级对象（first-class objects）。指向被绑定方法的弱引用的保质期总是很短的，在解除引用时它们总是返回 `None`，除非还有另外一个强引用指向了这个被绑定方法。

举个例子，下面的代码基于 Python 标准库 `weakref` 模块，并不会打印 “hello”，却会抛出一个异常：

```

>>> import weakref
>>> c = C()
>>> cf = weakref.ref(c.f)
>>> cf      # 先瞧瞧这是什么东西...
<weakref at 80ce394; dead>
>>> cf()
Traceback (most recent call last):
File "", line 1, in ?
TypeError: object of type 'None' is not callable

```

另一方面，下面展示的在 `weakmethod` 模块中的 `ref` 类则允许你使用指向被绑定方法的弱引用：

```
>>> import weakmethod
>>> cf = weakmethod.ref(c.f)
>>> cf( )()      # 哇哦！它是活的！
Hello
>>> del c      # …然后就死掉了
C dying
>>> print cf( )
None
```

调用 `weakmethod.ref` 实例，即指向一个被绑定方法的引用，与调用 `weakref.ref` 指向一个函数对象的实例具有相同的语义：如果引用无效，它返回 `None`；否则就返回引用。事实上，此例中它返回了一个新建的 `new.instancemethod`（它持有一个指向对象的强引用，因此，应当确保不持有这个引用，除非你想让那个对象多存活一段时间）。

注意，本节的解决方案代码的设计很讲究，可以很容易地把任何你想封装的可调用体放入一个 `ref` 实例中，无论是方法（绑定或未绑定的）、函数或其他任何东西。但只有当你试图封装一个被绑定的方法时，它才会有弱引用的语义；对其他的情况，`ref` 就像一个普通（强）引用，一直指向处于生存期的可调用体。这种方式让可以用 `ref` 封装任意的可调用体，而无须为任何特殊情况做特殊处理。

如果需要的语义接近于 `weakref.proxy` 的语义，那就更容易实现了，例如从本节的 `ref` 类派生一个子类。当你调用 `proxy` 时，`proxy` 会用相同的参数调用引用。如果被引用的对象已经消亡，会生成 `weakref.ReferenceError` 异常。下面是 `proxy` 类的一个实现：

```
class proxy(ref):
    def __call__(self, *args, **kwargs):
        func = ref.__call__(self)
        if func is None:
            raise weakref.ReferenceError('referent object is dead')
        else:
            return func(*args, **kwargs)
    def __eq__(self, other):
        if type(other) != type(self):
            return False
        return ref.__call__(self) == ref.__call__(other)
```

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中关于 `weakref` 和 `new` 模块，以及被绑定方法对象的章节。

## 6.11 缓存环的实现

感谢：Sébastien Keim、Paul Moore、Steve Alexander、Raymond Hettinger

## 任务

你想定义一个固定尺寸的缓存，当它被填满时，新加入的元素会覆盖第一个（最老的）元素。这种数据结构在存储日志和历史信息时非常有用。

## 解决方案

当缓存填满时，本节解决方案及时地修改了缓存对象，使其从未填满的缓存类变成了填满的缓存类：

```
class RingBuffer(object):
    """ 这是一个未填满的缓存类 """
    def __init__(self, size_max):
        self.max = size_max
        self.data = []
    class __Full(object):
        """ 这是一个填满了的缓存类 """
        def append(self, x):
            """ 加入新的元素覆盖最旧的元素 """
            self.data[self.cur] = x
            self.cur = (self.cur+1) % self.max
        def tolist(self):
            """ 以正确的顺序返回元素列表 """
            return self.data[self.cur:] + self.data[:self.cur]
    def append(self, x):
        """ 在缓存末尾增加一个元素 """
        self.data.append(x)
        if len(self.data) == self.max:
            self.cur = 0
            # 永久性地将 self 的类从非满改成满
            self.__class__ = self.__Full
    def tolist(self):
        """ 返回一个从最旧的到最新的元素的列表 """
        return self.data
# 用法示例
if __name__ == '__main__':
    x = RingBuffer(5)
    x.append(1); x.append(2); x.append(3); x.append(4)
    print x.__class__, x.tolist()
    x.append(5)
    print x.__class__, x.tolist()
    x.append(6)
    print x.data, x.tolist()
    x.append(7); x.append(8); x.append(9); x.append(10)
    print x.data, x.tolist()
```

## 讨论

缓存环是有固定大小的缓存。当它被填满时，加入新元素会覆盖掉它持有的最旧的元

素。在存储日志和历史信息时缓存环是非常有用数据结构。Python 并没有为这种数据结构提供直接支持，但用它构建一个这种结构却轻而易举。本节解决方案专门为元素插入进行了优化。

一个值得注意的设计要点是，这些对象在它们的生命周期中会经历某种不可逆转的状态转变——从未填满的缓存变成填满的缓存（从此时开始它的行为方式也发生了变化），我通过修改 `self.__class__` 来完成这个转变。这种方式对经典类和新风格类都同样有效，只要这些旧的或者新的类的对象都拥有相同的槽（但对两个没有槽的新风格类也有效，比如本节的 `RingBuffer` 和 `_Full` 类）。注意，和其他语言不同，虽然 `_Full` 类的实现位于 `RingBuffer` 类的内部，但两者并没有什么特别的联系，这样其实很好，因为我们完全不需要这种联系。

修改类实例在很多语言中会显得很古怪，但在 Python 中，相比于其他一些随意的、无法逆转的、零散的修改方式，它是一种很好的选择。而且，这样的修改对于所有的类都是可行的。

缓存环（或者叫做有界的队列）是一个很棒的点子，但是总测试缓存环有没有被填满是很低效的操作，我们也可以找到别的办法，但是测试本身就是一件很讨厌的事情。这种讨厌的事情在 Python 世界中是不受欢迎的，虽然用 Python 来做这些事并没有什么难度，而且测试还涉及了更多的内存使用。关键就在于当环被填满时，给 `__class__` 赋值以改变其行为方式，这也是它效率出众的原因：这种类转换是一次性的操作，所以它不会带来任何性能上的开销。

另一种选择是，我们可以切换实例的两个方法而非整个类，来使它变成填满的状态：

```
class RingBuffer(object):
    def __init__(self, size_max):
        self.max = size_max
        self.data = []
    def _full_append(self, x):
        self.data[self.cur] = x
        self.cur = (self.cur+1) % self.max
    def _full_get(self):
        return self.data[self.cur:]+self.data[:self.cur]
    def append(self, x):
        self.data.append(x)
        if len(self.data) == self.max:
            self.cur = 0
            # Permanently change self's methods from non-full to full
            self.append = self._full_append
            self.tolist = self._full_get
    def tolist(self):
        return self.data
```

这种切换的方式本质上等价于解决方案给出的类切换方式，尽管具体机制不同。当需要成组地切换所有的方法时，类切换的方式可能是最佳的，而方法切换则在需要更细的行为粒度控制的时候更加合适。当需要在新风格类中切换某些特殊方法时，类切换是唯一的办法，这是因为它固有的特殊方法查询是针对类进行的，而不是针对实例的（经典类和新风格类在这个方面截然不同）。

还可以使用其他很多方法来实现缓存环。在 Python 2.4 中，可以考虑从新类型 `collections.deque` 派生一个子类，这个类型提供了“双头队列”，因而从任意一头添加或者删除数据的效率都是相当的：

```
from collections import deque
class RingBuffer(deque):
    def __init__(self, size_max):
        deque.__init__(self)
        self.size_max = size_max
    def append(self, datum):
        deque.append(self, datum)
        if len(self) > self.size_max:
            self.popleft()
    def tolist(self):
        return list(self)
```

或者，当环处于稳定状态时，为了避免 `if` 语句，还可以混用方法切换：

```
from collections import deque
class RingBuffer(deque):
    def __init__(self, size_max):
        deque.__init__(self)
        self.size_max = size_max
    def _full_append(self, datum):
        deque.append(self, datum)
        self.popleft()
    def append(self, datum):
        deque.append(self, datum)
        if len(self) == self.size_max:
            self.append = self._full_append
    def tolist(self):
        return list(self)
```

在最后的这个实现中，我们只需要切换 `append` 方法（而 `tolist` 方法则保持原状），在这里方法切换显得比类切换更方便。

## 更多资料

`Reference Manual` 和 *Python in a Nutshell* 中关于标准类型层次体系以及经典和新风格对象模型的内容；Python 2.4 的 *Library Reference* 中关于 `collections` 模块的内容。

## 6.12 检查一个实例的状态变化

感谢: David Hughes

### 任务

一个实例在上次“保存”操作之后又被修改了，需要检查它的状态变化以便有选择地保存此实例。

### 解决方案

一个有效的方案是 mixin 类，这个类可以从多个类继承并能对一个实例的状态进行快照操作，这样就可以用此实例的当前状态和上次的快照做比较，来判断它是否被修改过了：

```
import copy
class ChangeCheckerMixin(object):
    containerItems = {dict: dict.iteritems, list: enumerate}
    immutable = False
    def snapshot(self):
        ''' 创建 self 状态的“快照”——就像浅拷贝，但
            只对容器的类型递归（而不是对整个实例：
            在需要时实例会自行记录自己的状态变化）'''
        if self.immutable:
            return
        self._snapshot = self._copy_container(self.__dict__)
    def makeImmutable(self):
        ''' 实例状态无法被修改 设置immutable '''
        self.immutable = True
        try:
            del self._snapshot
        except AttributeError:
            pass
    def _copy_container(self, container):
        ''' 半浅拷贝，只对容器类型递归 '''
        new_container = copy.copy(container)
        for k, v in self.containerItems[type(new_container)](new_container):
            if type(v) in self.containerItems:
                new_container[k] = self._copy_container(v)
            elif hasattr(v, 'snapshot'):
                v.snapshot()
        return new_container
    def isChanged(self):
        ''' 从上次快照之后如果有变化返回 True '''
        if self.immutable:
            return False
```

```

# 从 self.__dict__ 中删除快照，并置之于末尾
snap = self.__dict__.pop('_snapshot', None)
if snap is None:
    return True
try:
    return self._checkContainer(self.__dict__, snap)
finally:
    self._snapshot = snap
def _checkContainer(self, container, snapshot):
    ''' 如果容器和快照不同，返回 True'''
    if len(container) != len(snapshot):
        return True
    for k, v in self.containerItems[type(container)](container):
        try:
            ov = snapshot[k]
        except LookupError:
            return True
        if self._checkItem(v, ov):
            return True
    return False
def _checkItem(self, newitem, olditem):
    ''' 比较新旧元素。如果它们是容器类型，递归调用
        self._checkContainer recursively. 如果它们是带有 "isChanged"
        方法的实例，则委托给该方法。如果两者不相同
        则返回 True'''
    if type(newitem) != type(olditem):
        return True
    if type(newitem) in self.containerItems:
        return self._checkContainer(newitem, olditem)
    if newitem is olditem:
        method_isChanged = getattr(newitem, 'isChanged', None)
        if method_isChanged is None:
            return False
        return method_isChanged()
    return newitem != olditem

```

## 讨论

我常常需要在应用中能够检查状态变化的功能。比如，当一个用户关闭了某个文档的界面窗口，我需要检查这个文档是否在上次“保存”之后又被修改过；如果是，我需要弹出一个小窗口向用户提供几个选择，即保存文档、忽略修改或者取消关闭的操作。

解决方案中展示的 `ChangeCheckerMixin` 类，完全满足需要。主要思路是，从 `ChangeCheckerMixin` 派生全部的数据类，也就是持有可能被用户视图修改的数据的类，当数据刚从磁盘中载入或刚存入磁盘时，调用 `snapshot` 方法对文档数据实例进行快照操作。这个调用记录了当前状态，基本上是对此对象的基于容器递归的一个浅拷贝，并且还调用了所有被包含对象的 `snapshot` 方法，如果说有的话。在这之后的任何时候，你都可

以对任何一个数据类实例调用 `isChanged` 方法来检查实例状态在上次快照之后是否发生了变化。

至于容器类型，`ChangeCheckerMixin`，如同展示的那样，它只考虑 `list` 和 `dict`。如果你还需要用到其他类型作为容器，你只需把它们正确地添加到 `containerItems` 字典中。那个字典将每个容器类型映射到了一个可用于该类型实例的可调用的函数或方法，并获得一个关于索引或值的迭代器（用于索引该容器）。容器类型实例必须能够由 Python 标准库函数 `copy.copy` 拷贝，即它们必须支持浅拷贝。比如，为了将 Python 2.4 的 `collections.deque` 作为 `ChangeCheckerMixin` 的子类的容器，可以这样编写代码：

```
import collections
class CCM_with_deque(ChangeCheckerMixin):
    containerItems = dict(ChangeCheckerMixin.containerItems)
    containerItems[collections.deque] = enumerate
```

这是因为 `collections.deque` 可以通过 `enumerate` “遍历”，就像 `list` 一样。

下面是一个使用 `ChangeCheckerMixin` 的示例：

```
if __name__ == '__main__':
    class eg(ChangeCheckerMixin):
        def __init__(self, *a, **k):
            self.L = list(*a, **k)
        def __str__(self):
            return 'eg(%s)' % str(self.L)
        def __getattr__(self, a):
            return getattr(self.L, a)
    x = eg('ciao')
    print 'x =', x, 'is changed =', x.isChanged()
    # 输出: x = eg(['c', 'i', 'a', 'o']) is changed = True
    # 现在, 假设 x 已经被保存了, 然后:
    x.snapshot()
    print 'x =', x, 'is changed =', x.isChanged()
    # 输出: x = eg(['c', 'i', 'a', 'o']) is changed = False
    # 现在我们修改 x:
    x.append('x')
    print 'x =', x, 'is changed =', x.isChanged()
    # 输出: x = eg(['c', 'i', 'a', 'o', 'x']) is changed = True
```

在 `eg` 类中我们只是从 `ChangeCheckerMixin` 派生，因为我们不需要其他基类。具体地说，即使我们从 `list` 派生也没什么用处，因为状态检查功能只能被用于那些保存在实例的字典里的状态。所以，我们必须把一个 `list` 对象放在实例的字典中，并根据情况将某些任务委托给它（在上述示例中，我们通过 `__getattr__` 委托了所有的非特殊方法）。这样，我们就可以看到 `isChanged` 方法能够敏锐地反应出任何细微的改变——从上次调用 `snapshot` 之后实例的状态是否发生了变化。

本节方案的一个潜在的假设是，你的应用中的数据类都是以层级式的方式组织起来的。

一个很陈旧的（但仍然有效）的例子是发票，它含有数据头和详细的数据行。每个详细数据类的实例都可能包含其他实例，如产品细节，产品细节在本次操作中也许是不能被修改的，但它们在别处却可能是可修改的。这就是我们提供 `immutable` 属性和 `makeImmutable` 方法原因：当调用此方法设置了此属性之后，任何对实例的快照都会被忽略以节省内存，这样，进一步对 `snapshot` 或者 `isChanged` 的调用都能够更快地返回。

如果你的数据无法满足这样一种层级结构，你可能必须进行深度拷贝，或者对文档实例进行一个深度的“快照”，然后通过比较上次快照和当前状态来判断是否发生了改变。在很快的机器上这不是什么问题，在需要处理的数据量比较少的时候也没什么大影响。但是在我的测试中，在普通的计算机上，对较大数据量的处理速度完全无法让人接受。而如果你的数据组织方式比较适合这种应用，那么本节的解决方案能够提供更好的性能。如果你的一些数据类包含了那些自动计算出来的数据，并且由于某些原因这些数据也无须保存，可以把这种数据存到附属类的实例中（这些类也不用从 `Change CheckerMixin` 派生），而不应该将这些数据作为容器属性或者存在容器中，如列表和字典这样的容器。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 文档中关于多继承的内容，字典的 `iteritems` 方法，以及内建的 `enumerate`、`isinstance` 和 `hasattr` 函数。

## 6.13 检查一个对象是否包含某种必要的属性

感谢：Alex Martelli

### 任务

你想在进行状态修改操作之前检查一个对象是否有某种必要的属性，但你想避免使用类型检查的方式，因为那样会打破多态机制的灵活性。

### 解决方案

在 Python 中，通常你想做什么操作可以直接进行。下面是一个简单的例子，在没有检查的情况下对 `list` 参数做了一系列的操作：

```
def munge1(alist):
    alist.append(23)
    alist.extend(range(5))
    alist.append(42)
    alist[4] = alist[3]
    alist.extend(range(2))
```

如果 `alist` 缺少你调用的方法（明面上的，如 `append` 和 `extend`；暗地里的，如 `__getitem__`

和 `__setitem__`, 这是赋值声明 `alist[4] = alist[3]` 要求的), 企图调用不存在的方法会引发一个异常。函数 `munge1` 并未试图去截获异常, 所以 `munge1` 的异常会终结其运行, 并且向上扩散, 传递给调用 `munge1` 的调用者。调用者可以选择捕获异常并处理之, 或者不做处理任由异常继续向上层调用链扩散。

这种方法通常都工作得很好, 但有时也会遇到问题。假设 `alist` 对象有 `append` 方法但却没有 `extend` 方法, 在这个特例中, `munge1` 函数会在异常触发之前, 部分修改 `alist`。这种部分修改通常都无法被干净地恢复原样; 基于你的应用情况, 有时它们会造成很大的麻烦。

为了预防这种“部分修改”的问题, 第一个想法是检查 `alist` 的类型。这种天真的“先看后过”(Look Before You Leap, LBYL) 的方法看上去比不预先检查要安全一些, 但 LBYL 有个致命的缺陷: 它以伤害多态性为代价, 最糟糕的方法就是检查类型:

```
def munge2(alist):
    if type(alist) is list:      # 这是很糟糕的做法
        munge1(alist)
    else: raise TypeError, "expected list, got %s" % type(alist)
```

这个方法很容易失败, 比如 `alist` 可能是 `list` 的子类。至少还可以使用 `isinstance` 来做点改良:

```
def munge3(alist):
    if isinstance(alist, list):
        munge1(alist)
    else: raise TypeError, "expected list, got %s" % type(alist)
```

不过, 当 `alist` 是某种模拟 `list` 的类或类型的实例时, `munge3` 仍然会失败。换句话说, 这种类型检查牺牲了 Python 最强大的力量: 基于签名的多态机制。比如, 你无法给 `munge3` 传递一个 Python 2.4 中的 `collections.deque` 的实例, 这实在是很遗憾的事情, 因为这个 `deque` 能够提供所有需要的功能, 而且它也可以被传递给 `munge1`, 工作得很好。有无数的类似 `deque` 的序列类型, 可以很好地工作于 `munge1` 却不能被 `munge3` 接受。所以, 即使有 `isinstance` 来帮助, 类型检查仍然代价高昂。

一个更好的方式是使用准确的 LBYL, 这样更加安全而且全面支持多态:

```
def munge4(alist):
    # 首先得到所有需要的被绑定方法 (迅速获取异常, 而且
    # 不会造成部分修改现象, 如果需要的方法不存在的话):
    append = alist.append
    extend = alist.extend
    # 检查索引之类的操作以便尽快地获得异常
    # 如果不满足签名兼容性的话:
    try: alist[0] = alist[0]
    except IndexError: pass      # 空的 alist 也没问题
    # 开始操作: 这之后不会再有异常发生了
    append(23)
```

```
extend(range(5))
append(42)
alist[4] = alist[3]
extend(range(2))
```

## 讨论

Python 函数对于传给它们的参数天然地支持多态性，因为本质上它们只依赖参数本身的方法和行为方式，而不依赖于其类型。如果你检查参数的类型，你就失去了宝贵的多态性，所以，千万别这么做。不过，你也许想做一些早期检查以获得更好的安全性（尤其是避免部分修改现象），而且还不想付出太多的代价。

### 什么是多态

多态（polymorphism，其希腊词根意味着“很多形状”）是代码的一种能够以适当的方式处理各种不同类型对象的能力。不过，这个有用的概念已经在某种程度上被曲解了，很多人常常把多态和重载（即在调用时根据签名确定使用不同的函数）或者子类型（子类）联系起来，但其实这个概念远不止于此。

子类化是一种有用的实现技术，但对多态而言并不是必要条件。重载则明确多了：Python 不能让多个有相同名字的对象在同一时间在同一作用域中存在，所以在同一个作用域中，你无法同时有几个同名的函数或者方法，而只能用签名的方式将它们区分开来——有点烦人，在最坏的情况下，仅仅修改它们的名字就足以区分它们了。

Python 的函数是多态的（除非你用一些特别的方式来抑制这种特性），因为它们只关心对参数调用方法（显式地调用或隐式地调用，如算数运算和索引操作）：只要参数提供了需要的方法，调用体具有需要的签名，而且这些调用能够提供适当的行为，一切就能正常工作。

一般的 Python 风格的工作方式可以被描述为“获得事后原谅总是比事先得到许可要容易得多”（EAFP）：总是先试图执行需要的操作，如果过程中出现了异常，再处理或者忽略异常。这种方法通常能够很好地工作。偶尔碰到的最难解决的问题是“部分修改”：当需要对一个对象进行多个操作时，以某种自然的顺序执行这些操作有时会在异常触发之前部分地修改该对象。

比如，考虑解决方案中最先提出的 `munge1`，假设我们给它一个有 `append` 方法但没有 `extend` 方法的参数 `alist`。在这种情况下，`alist` 会被第一个 `append` 调用修改；之后，获取并调用 `extend` 的企图触发了异常，这使得 `alist` 处于一种被部分修改的状态，这种状态也很难修复。有时，一系列的操作在理想状态下应该是原子化的：或者所有的修改都成功进行，工作正常，或者没有任何一个操作可以执行，并抛出异常。

可以用一种精确的、谨慎的方式使用 LBYL 方法，最大程度地接近那种理想化的原子

操作。首先获取所有需要的被绑定方法，以一种无副作用的方式测试必要的操作（比如用赋值的方法来检查索引操作）。只有当这些测试都成功之后我们才能进行真正的操作，修改对象的状态。完成测试之后，任何测试过的操作几乎不可能（不是完全没可能）触发异常并引起部分修改现象。另外，即使进行严格的类型检查你也无法 100% 保证安全性，举个例子，你可能在操作到一半的时候用光了内存。因此，无论有没有类型检查，你都不能保证真正的原子化操作，而你只能尽量地接近那种理想化的处理方式。

如果我们想尽力避免部分修改现象，则准确的 LBYL 方法提供了比 EAFP 法更好的平衡和折中。略微提升的复杂度也可以接受，而预先检查所引起的一些性能损失，也可以从其他方面得到的补偿，即，通过局部名字获取和使用被绑定方法，要比直接访问属性的方式快（特别是在涉及循环的情况下，这也是很常见的应用场景）。很重要的一点是不要过度检查，而 `assert` 声明能起到很大作用。比如，可以给 `munge4` 添加如 `assert callable (append)` 这样的检查。编译器会在你指定了优化开关（比如给 `python` 命令加上`-O` 或`-OO` 标志）之后彻底地移除 `assert`，但在测试和调试运行（即没有指定优化标志）中这些检查会被执行。

## 更多资料

*Language Reference* 和 *Python in a Nutshell* 关于 `assert` 以及`-O` 和`-OO` 命令行参数的内容；*Library Reference* 和 *Python in a Nutshell* 中关于序列类型的内容，尤其是 `list`。

## 6.14 实现状态设计模式

感谢：Elmar Bschorer

### 任务

你希望你程序中的某个对象能在不同的“状态”之间切换，而且该对象的行为方式也能随着状态的变化而变化。

### 解决方案

状态设计模式的关键思路是将“状态”（带有它自身的行为方式）对象化，使其成为一个类实例（带有一些方法）。在 Python 中，你不用创建一个抽象类来表现这些不同状态共同的接口：只需为这些“状态”本身编写不同的类即可。比如：

```
class TraceNormal(object):
    '正常的状态'
    def startMessage(self):
        self.nstr = self.characters = 0
    def emitString(self, s):
```

```

        self.nstr += 1
        self.characters += len(s)
    def endMessage(self):
        print '%d characters in %d strings' % (self.characters, self.nstr)
class TraceChatty(object):
    '详细的状态'
    def startMessage(self):
        self.msg = []
    def emitString(self, s):
        self.msg.append(repr(s))
    def endMessage(self):
        print 'Message: ', ', '.join(self.msg)
class TraceQuiet(object):
    '无输出的状态'
    def startMessage(self): pass
    def emitString(self, s): pass
    def endMessage(self): pass
class Tracer(object):
    def __init__(self, state): self.state = state
    def setState(self, state): self.state = state
    def emitStrings(self, strings):
        self.state.startMessage()
        for s in strings: self.state.emitString(s)
        self.state.endMessage()
if __name__ == '__main__':
    t = Tracer(TraceNormal())
    t.emitStrings('some example strings here'.split())
# 输出: 21 characters in 4 strings
    t.setState(TraceQuiet())
    t.emitStrings('some example strings here'.split())
# 无输出
    t.setState(TraceChatty())
    t.emitStrings('some example strings here'.split())
# 输出: Message: 'some', 'example', 'strings', 'here'

```

## 讨论

通过状态设计模式，我们能够“发掘出”一个对象的一些相关的行为（还可能包括一些与这些行为相关的数据），并将其置入一个辅助的状态对象，然后主对象可以在需要的时候通过调用“状态”对象的方法委托给这些行为。在 Python 的术语中，这种设计模式和重新绑定整个对象的`__class__`惯用法有一些联系，详情请参看 6.11 节，和重新绑定某些方法的方式也有些关联（如 2.14 节所示）。从某种意义上说，状态设计模式是处于前两种概念之间的：你将一些关联的行为组织起来，而不是通过更改对象的`__class__`在这些行为之间切换，也不是修改每一个方法。如果用经典设计模式的术语来描述，本节提供的模式处于经典状态设计模式和经典策略设计模式之间。

与一些相关的 Python 概念相比，状态设计模式看上去更有魅力，因为适当数量的数据

可以和你要委托的行为共生，而且数量正合适，正好可用于支持每种特别的行为。在本节解决方案的例子中，不同的状态对象需要截然不同的数据类型和不同数量的数据：对于类 TraceQuiet 而言，根本就无须任何数据，而 TraceNormal 只需要几个数字，TraceChatty 则需要所有的字符串列表。这些责任通常都被主对象委托给了各个“状态对象”。

不过有时会有一些超出我们的示例的情况，状态对象可能会需要更紧密地同主对象合作，在某些环境下甚至还需要调用主对象的方法，访问主对象的属性。为了实现这个目的，主对象可以传递一个参数给“状态”对象，可以是 self 或者某个 self 的被绑定方法。比如，假设本节解决方案中的示例需要扩展功能，主对象必须记录它送出的消息已经输出了多少行。Tracer.\_\_init\_\_ 必须给每个实例增加一个新的初始化项 self.lines = 0，而“状态”对象的 endMessage 方法的签名也需要被扩展为 def endMessage(self, tracer):。不过在类 TraceQuiet 的 endMessage 中会忽略掉这个 tracer 参数，因为它从来不用输出任何行；其他两个类的实现则需要增加一行声明 tracer.lines += 1，这是因为它们每条消息只输出一行。

如你所见，这种额外新增的功能要求更紧密的联络，但并不是什么难对付的问题。具体地说，经典状态设计模式的关键特性是，状态切换由状态对象掌握（而在策略设计模式中，切换则来自外部），不过这并不是区分两种设计模式的关键。

## 更多资料

见 <http://exciton.cs.rice.edu/JavaResources/DesignPatterns/> 提供的关于经典设计模式的内容，虽然是基于 Java 的。

## 6.15 实现单例模式

感谢：Jürgen Hermann

### 任务

你想保证某个类从始至终最多只能有一个实例。

### 解决方案

\_\_new\_\_ 静态方法使得这个任务极其简单：

```
class Singleton(object):
    """ 一个 Python 风格的单例模式 """
    def __new__(cls, *args, **kwargs):
        if '_inst' not in vars(cls):
            cls._inst = super(Singleton, cls).__new__(cls, *args,
**kwargs)
        return cls._inst
```

然后只需从 Singleton 派生子类即可，而且不要重载 `__new__`。然后，所有对此类的调用（通常是创建新实例）都将返回同一个实例（实例只会被创建一次。当你的程序第一次调用 Singleton 的子类时，唯一的实例就会被创建出来）。

## 讨论

本节方案展示了一个很浅显的在 Python 中实现“单例”设计模式的方法（见 E.Gamma 与其他著者合作的 *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley 出版）。所谓单例，是指一个类的实例从始至终只能被创建一次。具体地说，这种类常常被用来管理一些资源，这些资源根据其本身属性也只需要退出一次。见 6.16 节提供的一些关于 Python 中的单例模式的思考和设计选择。

我们可以加上常见的自我测试部分来完成整个模块，并展示其行为方式：

```
if __name__ == '__main__':
    class SingleSpam(Singleton):
        def __init__(self, s): self.s = s
        def __str__(self): return self.s
    s1 = SingleSpam('spam')
    print id(s1), s1.spam()
    s2 = SingleSpam('eggs')
    print id(s2), s2.spam()
```

当我们把这个模块作为一个脚本运行时，我们会得到下面这样的输出（`id` 的确切值显然是不断变化的）：

```
8172684 spam
8172684 spam
```

当我们试图实例化 `s2` 并传递 “egg” 参数时，“egg” 被忽略了——这就是单例模式的代价。

关于单例的一个常见议题是子类化。Singleton 类的编写方式意味着它的每个子类，包括直接子类或者间接子类，都能够分别获得一个独立的实例。从字面意义上讲，似乎违背了单一实例的原则，不过这依赖于“单一”的具体含义：

```
class Foo(Singleton): pass
class Bar(Foo): pass
f = Foo(); b = Bar()
print f is b, isinstance(f, Foo), isinstance(b, Foo)
# 输出: False True True
```

`f` 和 `b` 是单独的实例，但根据内建函数 `isinstance` 的结果，它们都是 `Foo` 的实例，因为 `isinstance` 检查时遵循的是 OOP 中的 “IS-A” 规则：子类的一个实例同时也是基类的一个实例。另一方面，`b` 是通过调用 `Bar` 来实例化的，如果我们试图干扰并返回一个 `f`，就违反了一个通常的假定，即调用类 `Bar` 返回的应该是类 `Bar` 的实例，而不是 `Bar` 的某个更早被实例化的父类实例。

在实践中，“单例”的子类化是一个让人头痛的问题，并没有什么明确的解决方案。如果这个问题对你很重要，那么 6.16 节提供并解释了另一个选择——Borg 惯用法，也许是一个更好的解决方式。

## 更多资料

6.16 节；E. Gamma、R. Helm、R. Johnson、J. Vlissides 的 *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley)。

# 6.16 用 Borg 惯用法来避免“单例”模式

感谢：Alex Martelli、Alex A. Naanou

## 任务

你想保证某个类从始至终只创建了一个实例：你并不关心生成的实例的 id，只关心其状态和行为方式，而且你还想确保它具有子类化能力。

## 解决方案

和“单例”模式相关的应用程序可以用另一种方式来实现，即允许多个实例被创建，但所有的实例都共享状态和行为方式。比起在实例的创建中做手脚，这种方式要灵活得多。可以从下面的 Borg 类派生子类：

```
class Borg(object):
    _shared_state = { }
    def __new__(cls, *a, **k):
        obj = object.__new__(cls, *a, **k)
        obj.__dict__ = cls._shared_state
        return obj
```

如果你重载了子类的 `__new__`（只有极少数类才需要这么做），要记住使用 `Borg.__new__`，而不是 `object.__new__`。如果需要你的类实例能够相互共享状态，但却不和 Borg 的其他子类实例共享，需要在类作用域中加入这样的声明：

```
_shared_state = { }
```

通过这种“数据重载”，你的类不会从 Borg 继承 `_shared_state` 属性，而是定义自己的数据。为了允许这种“数据重载”，Borg 的 `__new__` 应当使用 `cls._shared_state`，而不是 `Borg._shared_state`。

## 讨论

Borg

下面是一个典型的 Borg 用法：

```

if __name__ == '__main__':
    class Example(Borg):
        name = None
        def __init__(self, name=None):
            if name is not None: self.name = name
        def __str__(self): return 'name->%s' % self.name
    a = Example('Lara')
    b = Example()                      # 实例化 b, 和 a 共享 self.name
    print a, b
    c = Example('John Malkovich')      # 让 c 用改过的 self.name, 同时也改了 a&b
    print a, b, c
    b.name = 'Seven'                  # 设置 b.name, 同时也改了 a&c 的 name
    print a, b, c

```

如果把这个模块当作脚本运行，输出是：

```

name->Lara name->Lara
name->John Malkovich name->John Malkovich name->John Malkovich
name->Seven name->Seven name->Seven

```

`Example` 的所有实例都共享状态，所以对任何实例的名字属性的设置，无论是在`__init__`中修改还是直接修改，所有的实例都会受到影响。然而，请注意实例的 `id` 是不同的，既然我们没有定义特殊方法`__eq__`和`__hash__`，则每个实例都可以作为字典的独立的键。因此，如果我们继续我们的示例代码：

```

adict = {}
j = 0
for i in a, b, c:
    adict[i] = j
    j = j + 1
for i in a, b, c:
    print i, adict[i]

```

其输出是：

```

name->Seven 0
name->Seven 1
name->Seven 2

```

如果这种行为方式不是你想要的，可以给 `Example` 类或者 `Borg` 超类增加`__eq__`和`__hash__`方法。增加了这些方法之后，我们的示例就能够更好地模拟单例模式了，不过这取决于应用的实际需求。比如，下面给出一个增加了这些特殊方法的 `Borg` 版本：

```

class Borg(object):
    _shared_state = {}
    def __new__(cls, *a, **k):
        obj = super(Borg, cls).__new__(cls, *a, **k)
        obj.__dict__ = cls._shared_state
        return obj
    def __hash__(self): return 9      # 任意常数

```

```
def __eq__(self, other):
    try: return self.__dict__ is other.__dict__
    except AttributeError: return False
```

使用这个进一步修改后的 Borg 版本，例子的输出变成了：

```
name->Seven 2
name->Seven 2
name->Seven 2
```

选 Borg，还是单例模式，还是都不用？

单例模式是一个很容易记住的名字，但不幸的是，大多数的用途的关注点和它的关注点都不太契合：它关注对象的身份，而不是对象的状态和行为。Borg 准模式（Borg design nonpattern）则使得所有的实例共享状态，Python 的能力也使得我们可以轻易地实现这样的模式。

在很多需要考虑使用单例还是 Borg 的时候，你其实可能并不需要用它们中的任何一种。可以直接写一个 Python 模块，带有函数和模块作用域中的全局变量，而不需要定义一个带有方法以及属性的类。只有当需要从类派生或者利用类的特性定义特殊方法时，才应当使用类（见 6.2 节展示的一种综合类和模块的优点的方法）。即使你确实需要类，通常也没必要在类中加入一些代码来迫使其不支持多个实例。其实，更简单的用法通常也更好用。举个例子：

```
class froober(object):
    def __init__(self):
        etc, etc
froober = froober()
```

很自然地，现在 `froober` 就是它自己类的唯一实例，这是因为名字“`froober`”已经被重新绑定到了实例，而不是类。当然，别人也可以调用 `froober.__class__()`，但在阻止别人故意滥用你的设计上付出太多的努力并不值得。无论你在任何可能的滥用上花多少精力来防范，总会有人找到办法绕过去。采取措施阻止一些意外和偶然的误用就已经足够了。如果最后展示的简单代码片段能够满足你的需要，就用它好了，不管是单例还是 Borg。要记住：在能正确工作的前提下做最简单的事。在少数情况下，这种简单的用法也可能无法工作，那么你就需要考虑得多一些了。

单例模式（在前面 6.15 节介绍过）仅仅是为了保证某个类最多只能有一个实例。根据我的经验，单例模式通常并不是它试图解决的问题的最佳方案，而且它还带来了和不同的对象模式有关的各种问题。一种典型的做法是我们可以允许多个实例的创建，但这些实例共享状态。我们需要关心身份吗？我们只关心状态（和行为）。这种可选的模式基于状态共享，正是为了解决单例模式试图解决的问题，这种方式也被称为 Monostate。顺便说一句，我喜欢称单例模式为“Highlander”，因为实例是唯一的。

在 Python 中，可以用很多方法实现 Monostate，但 Borg 准模式常常是最好的。简洁是

Borg 的最大的优点。由于任何实例的 `__dict__` 都可以被重新绑定，Borg 在它的 `__new__` 中将它的每个实例的 `__dict__` 重新绑定到一个类属性字典。现在，对一个实例属性的引用或者绑定都将立刻影响到所有的实例。感谢 David Ascher 为这个模式所建议的名字 Borg。Borg 也许只能说是准模式，因为在它第一次公开的时候，没人听说过关于它的实际使用的案例（当然现在有一些案例了）：两个或者更多的应用案例被认为是成为一种设计模式的前提条件之一。更多的细节讨论请参看 <http://www.aleax.it/5ep.html>。

Robert Martin 关于单例和 Monostate 有一篇很好的文章，请参看 <http://www.objectmentor.com/resources/articles/SingletonAndMonostate.pdf>。请注意，绝大多数 Martin 提到的关于 Monostate 的劣势都可以归于他考虑过的语言的限制，如 C++ 和 Java，但这些劣势在 Python 的 Borg 中都消失了。比如，Martin 指出，Monostate 的第一个主要的缺点是“一个非 Monostate 的类无法通过派生转化为 Monostate 类”，但对于 Borg，通过多继承完成转化的难度可以说是微不足道。

## Borg 的一些零碎

在 Borg 的操作中，`__getattr__` 和 `__setattr__` 特殊方法并未被涉及。为此，可以在你的子类中独立地定义它们，以满足你的各种需求，或者也可以放任其不管。这两种方式都不是问题，因为在重新绑定实例的 `__dict__` 属性时，Python 并不会调用 `__setattr__`。

Borg 对于那种把所有或者部分状态保存在 `__dict__` 之外的类并不太适用。因此，在 Borg 的子类中，避免定义 `__slots__` 以优化内存占用并没有什么意义，虽然它是为那些有很多实例的类设计的，但 Borg 的子类会只拥有一个实例。而且，不要从内建类型如 `list` 或 `dict` 派生，你的 Borg 子类应该进行一些封装工作并自动委托，如前面 6.5 节所示（我把后者称为“DeleBorg”，读者可以在 <http://www.aleax.it/5ep.html> 读到我的文章。）

Borg 就是单例这种说法是很荒谬的，就好比说门廊是雨伞一样。从设计模式的角度看，它们有着相似的目的（让可以雨中漫步又不至于淋湿），解决相似的问题，但是它们本质上是用不同的方式实现的，它们不是同一种模式的实例。如果说有什么相似之处，我们前面已经提到，Borg 和作为单例的一种替换方案的 Monostate 有一些相似。不过，Monostate 是一种模式，而 Borg 还不完全是；另外，不用成为 Borg，Python 的 Monostate 也可以独立存在。我们可以说 Borg 是一种惯用法，可以比较容易和高效地在 Python 中实现 Monostate。

由于某些对我来说完全无法理解的原因，人们总是把 Borg 和单例的问题与其他一些独立的问题——如访问控制，尤其是多线程的访问——混在一起。如果需要控制对一个对象的访问，无论这个对象的类有一个实例还是二十个实例，也无论这些实例在共享或不共享状态，你要做的事情都是一样的。有一种能够高效解决问题的方法，被称为分而治之（divide and conquer），能够将问题分解成不同方面的子问题，从而极大地简化了求解难度。而这种把各方面的问题聚在一起从而成功地增加了问题的难度的方法

也许可以被称为聚而苦之 (unite and suffer)。

## 更多资料

6.5 节, 6.15 节; Alex Martelli 的 Five Easy Pieces: Simple Python Non-Patterns 一文, 参看 <http://www.aleax.it/5ep.html>

## 6.17 Null 对象设计模式的实现

感谢: Dinu C. Gherman、Holger Krekel

### 任务

你想减少代码中的条件声明, 尤其是针对特殊情况的检查。

### 解决方案

一种常见的代表“这里什么也没有”的占位符是 `None`, 但我们还可以定义一个类, 其行为方式和这种占位符相似, 而且效果更好:

```
class Null(object):
    """ Null 对象总是很可靠地“什么也不做” """
    # 可选的优化: 确保每个子类只有一个实例
    # (完全是为了节省内存, 功能上没有任何差异)
    def __new__(cls, *args, **kwargs):
        if '_inst' not in vars(cls):
            cls._inst = type.__new__(cls, *args, **kwargs)
        return cls._inst
    def __init__(self, *args, **kwargs): pass
    def __call__(self, *args, **kwargs): return self
    def __repr__(self): return "Null()"
    def __nonzero__(self): return False
    def __getattr__(self, name): return self
    def __setattr__(self, name, value): return self
    def __delattr__(self, name): return self
```

### 讨论

可以使用 `Null` 类的一个实例而不是原生的 `None`。使用这种实例作为占位符, 而不是 `None`, 就可以在你的代码中避免很多条件声明, 而且能够用极少的特殊值检查来实现算法。本节解决方案是 `Null` 对象设计模式的一个实现。(参考 B. Woolf, *Pattern Languages of Programming* 中的 “The Null Object Pattern” [PLoP 96, 1996 年 9 月]。)

本节中的 `Null` 类忽略了所有在构建时和调用实例时传入的参数, 也忽略了所有的设置和删除属性的操作。任何调用或者访问属性 (或者方法, 因为 Python 并不区分两者,

一概调用`__getattr__`）的操作都返回同一个 Null 实例（即 self——没有什么必要创建一个新实例）。比如，假如你有这样一个计算：

```
def compute(x, y):
    try:
        lots of computation here to return some appropriate object
    except SomeError:
        return None
```

而你这样用它：

```
for x in xs:
    for y in ys:
        obj = compute(x, y)
        if obj is not None:
            obj.somemethod(y, x)
```

可以将计算修改为：

```
def compute(x, y):
    try:
        lots of computation here to return some appropriate object
    except SomeError:
        return Null()
```

对它的用法可以简化为：

```
for x in xs:
    for y in ys:
        compute(x, y).somemethod(y, x)
```

其要点是你无须检查 `compute` 究竟是返回了一个真实的结果还是 Null 的一个实例：即使是后者，也可以安全无害地调用它的任何方法。下面是另一个更具体的使用例子：

```
log = err = Null()
if verbose:
    log = open('/tmp/log', 'w')
    err = open('/tmp/err', 'w')
log.write('blabla')
err.write('blabla error')
```

很明显可以避免某种代码“污染”，如 `if verbose:` 这样的防护性代码，总是散布于各处。可以直接调用 `log.write('bla')`，无须用以往的表达方式，如 `if log is not None: log.write('bla')`。

在新的对象模型中，对于执行某些操作所需要的特殊方法，Python 并不会对实例调用 `__getattr__` 方法（而是检查该实例的类的槽（slot））。需要谨慎地定制 Null 类来满足你的应用对空对象操作的需求，因此需要仔细设计空对象类的特殊方法，不管它们是直接嵌入基类的代码中的，还是以子类化的方式扩展的。举个例子，对于本节解决方案中的 Null，你无法索引 Null 实例，也不能取得它的长度，还无法对它迭代。如果这对

于你的应用而言是个问题，可以根据需要增加一些特殊方法（在 Null 中直接增加或者从 Null 派生并扩展）并提供适当的实现，比如：

```
class SeqNull(Null):
    def __len__(self): return 0
    def __iter__(self): return iter(())
    def __getitem__(self, i): return self
    def __delitem__(self, i): return self
    def __setitem__(self, i, v): return self
```

利用此法也可增加其他操作。

Null 对象的关键设计目标是为我们在 Python 中常用的原始的 None 提供一种智能的替代物（其他语言中用 null 或者 null 指针来代表无值或空值）。这种“这里什么也没有”的标记或占位符可以用于多种情况，比如其中一种情况是，一组元素除了其中一个比较特殊其他都是类似的。这种用法经常导致到处都是条件声明，而条件声明的目的常常只是为了将普通的元素和原始的 null 值（比如 None）区分开来，Null 对象则能够帮助你避免过多的条件声明。

使用 Null 对象的优势如下：

- 通过提供一个第一等的对象作为原始的 None 值的代替，过多的条件声明得以避免，同时还提升了代码的可读性；
- Null 对象可以作为那些行为还没有完全实现的对象的占位符；
- Null 对象能够以一种多态的方式被任何其他类的实例使用（可能需要为某些特殊方法进行子类化扩展，正如前面提到过的那样）；
- Null 对象具有很好的可预测性。

Null 的一个很大的缺点是它可能会隐藏一些错误。如果一个函数返回 None，而调用者并不期望这样的返回值，调用者极有可能会接着对 None 调用一个它并不支持的方法或者操作，从而引发一个提示异常并回溯。如果返回值是调用者并不期望获得的 Null，则问题可能会被掩盖很长一段时间，当最终异常和回溯发生时，重新定位到代码最初的纰漏会更加困难。这个问题严重到影响 Null 的实用性了吗？答案是，这仍然是个人选择。如果你的代码在开发中总会有一些适当的单元测试，那么这个问题可能不会发生；而如果你根本就没有单元测试，使用 Null 带来的问题通常也只会是最小的问题。但正如我说的，这是个人的取舍。我喜欢到处用 Null，我非常满意它给我带来的生产率的提升。

本节展示的 Null 类使用了“单例”模式（见 6.15 节）的一个简单的变体，仅仅只是为了性能优化——说白了，就是为了避免创建很多什么也不做的对象，空耗内存。这个“单例”的实现确保了 Null 的每个子类在实例化的时候都只能有一个实例，这是很关键的。很显然，子类的数量不可能多到吃掉大量的内存，而各个子类之间的区分在语义

上也非常重要。

## 更多资料

B. Woolf, *Pattern Languages of Programming* 中的 “The Null Object Pattern” (PLoP 96, 1996 年 9 月), <http://www.cs.wustl.edu/~schmidt/PLoP-96/woolf1.ps.gz>; 6.15 节。

## 6.18 用 `__init__` 参数自动初始化实例变量

感谢: Peter Otten、Gary Robinson、Henry Crutcher、Paul Moore、Peter Schwalm、Holger Krekel

### 任务

你想避免编写和维护一种烦人的几乎什么也不做的`__init__`方法, 这种方法中含有一大堆形如 `self.something = something` 的赋值语句。

### 解决方案

可以把那些属性赋值任务抽取出来置入一个辅助函数中:

```
def attributesFromDict(d):
    self = d.pop('self')
    for n, v in d.iteritems( ):
        setattr(self, n, v)
```

而`__init__`方法里的那种千篇一律的赋值语句大概是这个样子的:

```
def __init__(self, foo, bar, baz, boom=1, bang=2):
    self.foo = foo
    self.bar = bar
    self.baz = baz
    self.boom = boom
    self.bang = bang
```

现在可以被缩减为清晰的一行:

```
def __init__(self, foo, bar, baz, boom=1, bang=2):
    attributesFromDict(locals( ))
```

### 讨论

如果`__init__`的主体中没有其他的逻辑, 调用内建函数 `locals` 返回的 `dict` 只包含了被传递给`__init__`的参数 (包括那些并未被传递但却有默认值的参数)。首先函数 `attributesFromDict` 获取对象, 假定这个对象是一个名字为 “`self`” 的参数, 然后将其他的所有元素作为它的属性名进行设置。一个相似但更简单的技术是, 不使用辅助函数, 而是像下面这样:

```
def __init__(self, foo, bar, baz, boom=1, bang=2):
    self.__dict__.update(locals())
    del self.self
```

不过，后来给出的这种技术同解决方案给出的方法相比，有一个重大的缺陷：它直接设置 `self.__dict__` 中的属性（通过 `update` 方法），对于一些特性（`property`）和高级描述符（`descriptor`），它无法正常地工作。而解决方案给出的方法使用了内建的 `setattr`，在这方面表现得很完美。

`attributesFromDict` 并不适用于使用了更多代码尤其是使用了本地变量的 `__init__` 方法，因为对于传递给它的唯一的字典参数，`attributesFromDict` 无法区分子典里的传递给 `__init__` 的参数以及 `__init__` 内部的局部变量。如果你在辅助函数中加入一点内省的能力，这个限制就可以被打破：

```
def attributesFromArguments(d):
    self = d.pop('self')
    codeObject = self.__init__.im_func.func_code
    argumentNames = codeObject.co_varnames[1:codeObject.co_argcount]
    for n in argumentNames:
        setattr(self, n, d[n])
```

通过获取 `__init__` 方法的代码对象，`attributesFromArguments` 函数能够只处理传递给 `__init__` 的参数名。因此你的 `__init__` 方法应当调用 `attributesFromArguments(locals())`，而不是 `attributesFromDict(locals())`，在这个调用之后如果还有需要，可以继续加入更多代码并定义其他局部变量。

`attributesFromArguments` 最关键的限制是它不支持 `__init__` 的某种特殊参数，即 `**kw`。我们可以通过引入更多的内省来获得处理 `**kw` 的能力，但那就要求使用更多的黑魔法和引入更多的复杂性，比起获得的这点功能似乎有些不值得。如果你确实想在这方面做些探索，为了实现内省，应当使用标准库的 `inspect` 模块，而不是在 `attributesFromArguments` 函数中自己实现这个功能。通过使用 `inspect.getargspec(self.__init__)`，你能够同时知道参数名以及 `self.__init__` 是否接受 `**kw` 形式的参数。关于 `inspect.getargspec` 的更多信息请参看 6.19 节。最后，请记住 Python 编程中的一个至理名言：“尽量用标准库搞定一切！”

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 文档中关于内建函数 `locals`，类型 `dict` 的方法，特殊方法 `__init__` 以及内省技术（包括 `inspect` 模块）的内容。

## 6.19 调用超类的 `__init__` 方法

感谢：Alex Martelli

## 任务

你想确保所有的超类的`__init__`方法被自动调用（如果该超类定义过`__init__`方法），但 Python 并不会自动调用此方法。

## 解决方案

如果你的类是新风格类，内建的 `super` 会使这个任务变得极其简单（如果所有的超类的`__init__`方法也用相似的方式使用 `super`）：

```
class NewStyleOnly(A, B, C):
    def __init__(self):
        super(NewStyleOnly, self).__init__()
        initialization specific to subclass NewStyleOnly
```

## 讨论

对新的开发任务，经典类不太值得推荐：它们的存在只是为了保证对于老版本 Python 的向后兼容性。应当尽量使用新风格类（直接或间接从 `object` 派生）。在新风格类中唯一你不能做的事情是将它的实例当做异常对象抛出，异常对象必须是旧风格的，不过，对这些类你并不需要使用本节提供的功能。因此，下面将要进行的讨论会比较深入同时适用面也更窄，你如果不感兴趣的话可以直接略过。

你可能需要在经典类中使用这个功能，或者，你也想在新风格类中使用这个功能，但是这个类的某些超类并不满足前面提到的条件，即以正确的方式调用内建函数 `super`。针对这些情况，首先你要做的是前期准备工作——将所有的类转变为新风格类并且使它们正确地调用 `super`。如果你确实无法完成准备工作，那么你能做的就是循环检查基类——对每个基类，都看它是否拥有`__init__`，如果有的话，调用该方法：

```
class LookBeforeYouLeap(X, Y, Z):
    def __init__(self):
        for base in self.__class__.__bases__:
            if hasattr(base, '__init__'):
                base.__init__(self)
        initialization specific to subclass LookBeforeYouLeap
```

更一般的，并不仅限于`__init__`方法，我们常常需要调用一个实例或者类的其他方法，如果这个方法存在的话；如果实例或类的此方法不存在，我们就什么也不做，或者只做默认的动作。解决方案中展示的基于内建的 `super` 技术并不具有通用性：它只适用于当前对象的超类，要求那些超类以正确的方式调用 `super`，还要求被调用的方法在那些超类中存在。不过，所有的新风格类都拥有`__init__`方法：它们都是 `object` 的子类，而 `object` 定义了`__init__`（默认什么也不做，接受并忽略任何参数）。因此，所有的新风格类都拥有`__init__`方法，无论是继承来的还是重载过的。

在 `LookBeforeYouLeap` 类中展示的 LBYL 技术具有更宽广的应用面，它可被用于包括`__`

`init_` 在内的各种方法。LBYL 还可以和 `super` 合用，如下面这个例子所示：

```
class Basel(object):
    def met(self):
        print 'met in Basel'
class Der1(Basel):
    def met(self):
        s = super(Der1, self)
        if hasattr(s, 'met'):
            s.met()
        print 'met in Der1'
class Base2(object):
    pass
class Der2(Base2):
    def met(self):
        s = super(Der2, self)
        if hasattr(s, 'met'):
            s.met()
        print 'met in Der2'
Der1().met()
Der2().met()
```

这个片段的输出是：

```
met in Basel
met in Der1
met in Der2
```

`met` 的实现对于所有的派生类，包括 `Der1`（超类是 `Basel`，超类有 `met` 方法）和 `Der2`（超类是 `Base2`，超类没有这个方法），都具有相同的结构。通过将局部名字 `s` 绑定到 `super` 的结果，并在调用方法之前先用 `hasattr` 检查超类是否拥有这个方法，这个 LBYL 结构使得可以用相似的代码处理两种情况。当然，在编写子类的时候，你通常就已经知道了超类有哪些方法以及是否需要和怎样调用它们。有时，当需要子类略微偏离它的超类的时候，这种技术能够提供一些额外的灵活度。

不过 LBYL 技术还远远谈不上完美：超类可能定义一个属性名为 `met`，这个 `met` 不能被调用也不需要参数。如果你确实非常需要灵活性，必须排除这种情况，可以获取超类的方法对象（如果有的话），然后用标准库模块 `inspect` 提供的 `getargspec` 函数来检查。

但是当试图将这个概念推及到一般的情况下，复杂性就增加了。下面是一个例子，这个类的方法会试图不用参数调用超类中同名的方法（如果在超类中该属性存在而且可调用）：

```
import inspect
class Der(A, B, C, D):
    def met(self):
        s = super(Der, self)
        # 获取超类的被绑定对象，如果没有的话返回 None
```

```
m = getattr(s, 'met', None)
try:
    args, varargs, varkw, defaults = inspect.getargspec(m)
except TypeError:
    # m 不是方法，忽略
    pass
else:
    # m 是方法，它的所有参数都有默认值吗？
    if len(defaults) == len(args):
        # 是的，调用之：
        m()
print 'met in Der'
```

如果传递给 `inspect.getargspec` 的参数不是方法或函数，它会抛出一个 `TypeError` 的异常，这样我们就可以通过 `try/except` 声明发现这种情况，如果确实有异常发生，我们只需在 `except` 子句中放入一个什么也不做的 `pass` 声明将其忽略即可。为了简化代码，我们并没有首先用 `hasattr` 进行检查工作，而是用带有第三个参数的 `getattr` 来获取“`met`”属性。因此，如果超类并没有任何一个叫做“`met`”的属性，`m` 会被设置成 `None`，如果我们用 `None` 作为 `inspect.getargspec` 的参数，同样会引发 `TypeError` 异常——这样起到了一石二鸟的作用。如果在 `try` 子句中对 `inspect.getargspec` 的调用没有引发 `TypeError` 异常，程序会继续执行 `else` 子句。

如果 `inspect.getargspec` 没有抛出 `TypeError`，它会返回一个含有四个子项的元组，我们可以将各项绑定到一个本地名。在这里，我们关心的是 `args`，它是 `m` 的参数名列表，以及 `defaults`，一个 `m` 为它的参数提供的默认值元组。很显然，当且仅当 `m` 为它的参数提供了默认值时，我们才可以不使用参数调用 `m`。因此，通过比较 `args` 列表和 `defaults` 元组的长度，我们可以知道默认值的数目是否和参数的数目一致，只要两者一致我们就可以调用 `m`。

很显然，在大多数的情况下没必要使用这种高级的内省技术并进行细致地检查，但如果你确实需要这种能力，Python 提供了所有需要的技术。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中关于内建函数 `super`、`getattr` 和 `inspect` 模块的文档。

## 6.20 精确和安全地使用协作的超类调用

感谢：Paul McNett、Alex Martelli

### 任务

你很欣赏通过内建的 `super` 来支持的多继承的代码的协作方式，但同时你也希望能够

更加简洁和精确地使用这种方式。

## 解决方案

一个好的方案是使用支持多继承的 mixin 类，此类使用了内省机制，且更加简洁：

```
import inspect
class SuperMixin(object):
    def super(cls, *args, **kwargs):
        frame = inspect.currentframe(1)
        self = frame.f_locals['self']
        methodName = frame.f_code.co_name
        method = getattr(super(cls, self), methodName, None)
        if inspect.ismethod(method):
            return method(*args, **kwargs)
        super = classmethod(super)
```

任何从 SuperMixin 派生的类都将获得一个名为 super 的魔术般的方法：在 cls 类的一个叫做 somename 的方法中调用 cls.super(args) 等于是在调用 super(cls, self).somename(args)。而且，即使在方法解析顺序（Method Resolution Order, MRO）中，在 cls 类之后没有任何类定义过一个叫做 somename 的方法，这个调用也是安全的。

## 讨论

下面是一个用法示例：

```
if __name__ == '__main__':
    class TestBase(list, SuperMixin):
        # 注意: myMethod 并未在此定义
        pass
    class MyTest1(TestBase):
        def myMethod(self):
            print "in MyTest1"
            MyTest1.super( )
    class MyTest2(TestBase):
        def myMethod(self):
            print "in MyTest2"
            MyTest2.super( )
    class MyTest(MyTest1, MyTest2):
        def myMethod(self):
            print "in MyTest"
            MyTest.super( )
            MyTest( ).myMethod( )
# 输出:
# in MyTest
# in MyTest1
# in MyTest2
```

Python 已经支持“新风格”类好些年了，作为比经典类更好的选择，新风格类也是默

认的选项。经典类的存在仅仅是为了满足一些老版本 Python 向后兼容的需要，对于新代码是不推荐使用的。使用新风格类的一个优点就是，我们可以很轻松地以一种“协作”的方式调用超类的实现，而且还完全支持多继承，这应当归功于内建的 `super`。

假设你的新风格类 `cls` 中有一个方法，这个方法需要先完成自己的任务然后将余下的工作委托给超类的同名方法。代码一般为：

```
def somename(self, *args):
    ...some preliminary task...
    return super(cls, self).somename(*args)
```

这个做法有两个小问题：比较啰嗦，而且它依赖于超类提供了方法 `somename`。如果你想把 `cls` 和其他类隔离得更开一些，从而获得更强的健壮性，可以除掉其依赖性，但代码就变得更啰嗦了：

```
def somename(self, *args):
    ...some preliminary task...
    try:
        super_method = super(cls, self).somename
    except AttributeError:
        return None
    else:
        return super_method(*args)
```

而解决方案中的 `mixin` 类 `SuperMixin` 则完美解决了这两方面的问题。只要 `cls` 直接或间接地从 `SuperMixin` 派生（也可以同时从其他需要的类派生），你就能够以一种精确和健壮的方式完成任务：

```
def somename(self, *args):
    ...some preliminary task...
    return cls.super(*args)
```

类方法 `SuperMixin.super` 依赖于一个简单的内省来获取 `self` 对象以及方法的名字，然后在内部使用内建的 `super` 和 `getattr` 来获取超类方法，如果此方法存在的话，再安全地调用之。内省是通过 Python 标准库的方便的 `inspect` 模块实现的，这使得整个任务变得更加简单。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 文档中关于 `super`，新对象模型以及 MRO，内建的 `getattr`，标准库模块 `inspect` 的内容；20.12 节中给出的另外一种不同的简化内建的 `super` 的使用的方法。

# 持久化和数据库

### 引言

感谢：Aaron Watters，软件顾问

世界上有三种人：会数数的和不会数数的。

但只有两种计算机程序：玩具程序和那种需要和持久化数据库交互的程序。也就是说，大多数真实应用中的程序需要获取存储的信息并记录信息以供进一步使用。这个描述也适用于几乎所有的计算机游戏，这些游戏都需要能够在任何时候记录并存储游戏状态。当我提到玩具程序时，我是指那些出于练习目的，或者纯粹写来玩的程序。几乎所有的真实应用中的程序（比如那些受雇佣的程序员编写的程序）都有一些持久数据库的存储和获取的组件。

20 世纪 80 年代我还是一名 Fortran 程序员，我注意到虽然几乎所有的程序都需要获取和存储信息，但它们几乎都是用一种手工的自制方式来实现这部分功能的。而且，从程序员角度看，程序的存储和获取信息的组件是让人最不感兴趣的，这部分代码常常写得极乱，而且隐藏着大量难以跟踪和解决的问题。这样的观察让我得出一个结论，数据库系统的研究和实现是软件实用化的核心，但根据我观察到的这方面的进展和状态来看，实在是还差得远。

后来，在研究所我很欣喜地发现了一个复杂的，但令人印象深刻的数据库实现。这个所谓的数据库覆盖了并行性、容错性、分布性、查询优化、数据库设计以及事务语义等。它有一种典型的学院派的风格，很多概念都建立在一些很荒谬的出发点上（比如有条件的多值依赖性的概念），但很多关于它的工作却可以直接用于实现可靠高效的信息存储和获取的实用系统。这些工作在很大程度上是来源于一篇论文，即 E.F. Codd 的 “*A Relational Model of Data for Large Shared Data Banks*”。(E.F. Codd, "*A Relational Model of Data for Large Shared Data Banks*," *Communications of the ACM*, 13, no. 6 (1970),

pp. 377-87, <http://www.acm.org/classics/nov95/toc.html>)

但我的其他研究生同学，甚至大多数的教员，都对之不屑一顾。每个人都知道这种传统的关系型技术会很有赚头，但他们一般认为这种技术和怎样写（或者更重要的维护）COBOL 程序的技术是一个等级的。即使数据库接口标准 SQL（现在已经极其成熟了），看上去很像 COBOL 的扩展也没能改变其处境，当然，SQL 其实和任何现代编程语言都没有太大的关联。

十几年后，对大多数基于数据的应用程序，基于 SQL 的关系型技术已经统治了整个领域，短期内不会有任何技术能对其形成挑战。事实上，关系型数据库技术几乎无所不在。最大的软件提供商——IBM、Microsoft 以及 Oracle，提供了各种关系型数据库的实现，并将这些实现作为它们的核心产品。其他的软件公司，比如 SAP 和 PeopleSoft，本质上只是提供构建在关系型数据库核心之上的软件层。

一般而言，关系型数据库只会不断地被修补，而不是被替代。企业软件工程的信条是三层系统，最低的一层就是设计良好的关系型数据库，中间层将数据库的视图定义为事务对象，而最高层则包括了应用和会话，最高层在实际上操纵了业务对象，后者将各种操作迅速地翻译成底层的关系表的变化。

微软的 Open Database Connectivity (ODBC) 标准为基于 SQL 的关系型数据库提供了一种通用的编程 API，这种接口可允许程序在不用改动或者做很少改动的情况下和不同的数据库引擎打交道。举个例子，一个 Python 程序可以在首次实现时用 Microsoft Jet (Microsoft Jet 的名气很响，但很多人错误地把它当做了“Microsoft Access database”。Access 是 Microsoft 销售的一种产品，主要是为了设计和实现数据库的前端；而 Jet 则是后端，可以免费地从 Microsoft 的网站下载。) 作为后端数据库，作为测试和调试之用。一旦程序稳定之后，只需（最多）改动一行，它就可以轻易地接入远程数据库，例如位于另一个大洲的运行在 IBM 大型机上的 DB2 数据库后端。

关系型数据库并不适用于所有的应用。尤其是那些总需要保存和读取会话的计算机游戏或者工程设计工具，它们更适合使用直接储存程序逻辑对象的方式，而不是关系型数据库推崇的扁平的表格式记录。不过，即使在工程和科学领域，部分采用关系型记录的混合方式仍是很好的选择。比如，我就见过一个用于储存基因序列信息的复杂的关系型数据库模式，序列是以二进制大对象 (BLOB) 存入的，但大量的辅助信息却不太容易放入关系表中。说到这里，我还真有点担心，我看上去就像一个彻头彻尾的关系型数据库的狂热分子。

在 Python 世界中，有很多方法可以提供持久化和数据库功能。我个人的最爱是 Gadfly，<http://gadfly.sourceforge.net/>，一个极其小巧的 SQL 实现，它主要是作为内存数据库来工作的。我喜欢它的最大原因是它是我的作品，它最大的优点是，如果它对你而言不敷使用了，可以轻松地切换到其他的工业级强度的 SQL 引擎。很多 Gadfly 用户都是从 Gadfly 开始开发应用（因为它很容易使用），然后再切换到其他数据库（因为他们需要

更多的功能)。

但很多人也许会更喜欢用其他 SQL 实现, 比如 MySQL、Microsoft Access、Oracle、Sybase、Microsoft SQL Server、SQLite, 或其他的提供了 ODBC 接口的实现 (Gadfly 并未提供此接口)。

Python 提供了一个标准接口用于访问关系型数据库: Python DB Application Programming Interface (Py-DBAPI), 这个接口最初是由 Greg Stein 设计的。每种数据库 API 都需要有一个 Py-DBAPI 封装的实现, 而几乎所有的数据库接口都有对应的实现, 如 Oracle 和 ODBC。

如果关系型数据库显得过于沉重, Python 还提供了内建的用于储存和获取数据的工具。从最基本的说起, 程序员们至少可以直接操纵文件, 这是第 2 章覆盖的内容。在操纵文件的基础上更进一步, marshal 模块允许程序对简单的 Python 类型 (不包括类和类实例) 构建成的数据结构进行序列化操作。marshal 能够以惊人的速度获取大量的数据结构。而 pickle 和 cPickle 模块则允许存储一般性对象, 包括类、类实例和环形结构 (circular structures)。cPickle 是用 C 语言实现的并因此而得名, 所以它的速度很快, 但仍然慢于 marshal。为了以某种人类能够读取的格式存储和获取结构化的数据, 使用 XML 格式来存取数据也是个不错的选择 (而且还可以利用 Python 中的几个 XML 解析和生成模块), 第 12 章将介绍 XML——但它更适合那些只用写入一次, 而需要读取很多次的应用。序列化数据以及 XML 可以被存入 SQL 数据库, 作为一种混合存储方法的一部分。

虽然 marshal 和 pickle 提供了基本的关于结构的序列化和反序列化操作, 程序员们仍然希望能够获得更多的功能, 比如事务的支持, 再比如并发控制。当关系型模型不能够满足应用的需要时, 一种直接的对象数据库实现可能会更加适用, 比如 Z-Object Database (ZODB), 参看 <http://zope.org/Products/ZODB3.2>。

对于那些轻视关系型数据库技术的人, 我希望他们能够改变一下观念。要知道, 关系型数据库如此成功不是没有原因的, 这是很值得我们深思的。下面转述丘吉尔的名言:

```
text = """ Indeed, it has been said that democracy is the worst form of
government, except for all those others that have been tried
from time to time. """
import string
for a, b in [("democracy", "SQL"), ("government", "database")]:
    text = string.replace(text, a, b)
print text
```

## 7.1 使用 marshal 模块序列化数据

感谢: Luther Blissett

## 任务

你想以尽量快的速度来序列化并重新构造那些由基本 Python 对象（比如列表、元组、数字和字符串，但不包括类、类实例等）构成的数据结构。

## 解决方案

如果你知道你的数据是由基本的 Python 对象构成的（而且你只想支持一种 Python 版本，当然仍需要此应用能够跨越几个不同的平台），那么，最底层的和最快速的序列化你的数据（即将其转化为字节串，以后可以通过此串重新构建数据）的方法是使用 `marshal` 模块。假设 `data` 是一个只含有 Python 基本数据类型的结构，如下：

```
data = {12:'twelve', 'feep':list('ciao'), 1.23:4+5j, (1,2,3):u'wer'}
```

可以很快的速度将 `data` 转化为字节串：

```
import marshal  
bytes = marshal.dumps(data)
```

现在可以将 `bytes` 放到任何地方（比如通过网络传递，或者放入数据库存为 BLOB，等等），当然你不应该修改它的任何一个字节。可以在任何时候通过此字符串重新构建出数据结构：

```
redata = marshal.loads(bytes)
```

如果你想将此数据写入磁盘文件（注意是以二进制方式打开的，而不是默认的文本输入输出模式），可以使用 `marshal` 模块的 `dump` 函数，它可以帮助你以一定的顺序将多个数据结构写入到同一个文件中：

```
ouf = open('datafile.dat', 'wb')  
marshal.dump(data, ouf)  
marshal.dump('some string', ouf)  
marshal.dump(range(19), ouf)  
ouf.close()
```

以后还可以以固定的顺序从 `datafile.dat` 文件中恢复出所有的数据结构：

```
inf = open('datafile.dat', 'rb')  
a = marshal.load(inf)  
b = marshal.load(inf)  
c = marshal.load(inf)  
inf.close()
```

## 讨论

Python 提供了几种方法来序列化数据（即将数据转化为字节串，然后可以将此串存在硬盘上，放到数据库中，或者通过网络传递，等等）以及从序列化形态重新构建出原数据。这种底层方法是通过 `marshal` 模块实现的，Python 常常需要将它产生的字节码存入文件。`marshal` 仅支持基本数据类型（比如字典、列表、元组、数字和字符串）以及

它们的组合。但 `marshal` 并不保证不同的 Python 发行版本之间的兼容性，因此通过 `marshal` 序列化的数据，在你升级 Python 之后，可能就不能被读取了。不过，`marshal` 保证了对于特定计算机体系的独立性，所以当你在不同的计算机之间传递数据时，只要这些计算机运行着相同版本的 Python，序列化都能正常工作，就好像在分布式环境下共享编译过的 Python 字节码。

`marshal` 的 `dumps` 函数接受任何适合的 Python 数据结构并返回一个字节串。可以将字节串传递给 `loads` 函数，它返回一个 Python 数据结构，此结构完全等同于（`==`）原数据结构。但特别的是，无论是在原数据结构还是新构建的数据结构中，字典的键总是以一种随机的顺序排列，不过这个顺序本身是有意义的，因此会被保存下来。在 `dump` 和 `load` 调用之间，可以做各种事情，通过网络传递字节码，将其存入数据库并再度读取，或者加密并解密。只要字符串本身的二进制结构没有被修改过，`load` 就能够很好地完成它的任务（如前面所述，必须用相同的 Python 发行版本才能确保它正常工作）。

当需要将数据存入文件时，可能会需要用到 `marshal` 的 `dump` 函数，它接受两个参数：你希望处理的数据结构以及一个打开的文件对象。注意，这个文件必须以二进制输入输出方式打开（而不是默认的文本模式），而且也不能是一个类文件对象，因为 `marshal` 对于它是不是真实的文件对象非常挑剔。`dump` 的优势是可以对多个不同的数据结构调用 `dump`，并写入同一个文件对象：这是因为当每个数据结构被序列化时，其他信息，如序列化后的字节码长度等也会被一并写入。因此，当以后你打开此文件读取二进制数据时，可以将文件作为参数传入 `marshal.load` 并调用之，可以一个一个按照顺序将原先的数据结构恢复，每次调用 `load` 恢复一个数据结构。`load` 的返回值是一个全新的数据结构，但仍然等同于原先的数据。（再次提醒，`dump` 和 `load` 只能在同一个 Python 发行版本下正常工作。）

对于那些习惯于其他语言或库的“序列化”工具的用户可能会问，`marshal` 是否对想要序列化或反序列化的对象的大小有限制。答案是：没有。你的计算机的内存可能是唯一的限制，如果你的计算机的内存非常大，`marshal` 在实践中几乎没有限制的。

## 更多资料

7.2 节中的 cPickle——`marshal` 的大哥；*Library Reference* 和 *Python in a Nutshell* 中的标准库模块 `marshal` 的文档。

## 7.2 使用 pickle 和 cPickle 模块序列化数据

感谢：Luther Blissett

### 任务

你想以某种可以接受的速度序列化和重建 Python 数据结构，这些数据既包括基本

Python 对象也包括类和实例。

## 解决方案

如果你不想假设你的数据完全由基本 Python 对象组成，或者需要在不同的 Python 版本之间移植，再或者需要将序列化后的形态作为文本传递，那么最好的序列化数据的方法是 cPickle 模块（pickle 模块是完全用 Python 实现的，而且完全可以替代 cPickle，但问题是它比较慢，除非你没有 cPickle 用，不然它并不是最好的选择）。举个例子，假设你有：

```
data = {12:'twelve', 'feep':list('ciao'), 1.23:4+5j, (1,2,3):u'wer'}
```

可以将 data 序列化为文本字符串：

```
import cPickle  
text = cPickle.dumps(data)
```

或者转化为二进制串，这种选择通常更快而且更节省空间：

```
bytes = cPickle.dumps(data, 2)
```

现在，可以将 text 和 bytes 按照意愿进行各种处理（比如，通过网络传递，作为 BLOB 放入数据库中，见 7.10 节、7.11 节和 7.12 节），只要你不修改 text 和 bytes 本身。对于 bytes 而言，那意味着应该保证其二进制字节序不被改变，对于 text 而言，应该保证它的文本结构不被改变，包括换行符。之后，无论在什么计算机体系结构之下，无论用什么版本的 Python，你都可以重新构建出那些数据：

```
redatal = cPickle.loads(text)  
redat2 = cPickle.loads(bytes)
```

每个调用创建出来的数据结构都等于原数据。比较特别的是，字典的键在原数据和新建数据中的顺序都是任意的，但这个顺序本身是有意义的，因此也会被保存。无须告诉 cPickle.loads 原先的 dumps 是否使用了文本模式（这是默认的模式，可以由一些很老的 Python 版本读取）或二进制模式（快速且紧凑），loads 可以通过检查参数的内容自行判断。

当你希望将数据写入文件时，可以直接使用 cPickle 的 dump 函数，它允许你将多个数据结构一个接一个地写入到同一个文件：

```
ouf = open('datafile.txt', 'w')  
cPickle.dump(data, ouf)  
cPickle.dump('some string', ouf)  
cPickle.dump(range(19), ouf)  
ouf.close()
```

一旦你做完这些事，就可以从 datafile.txt 中以相同的顺序一个接一个地恢复原来的数据结构：

```
inf = open('datafile.txt')
```

```
a = cPickle.load(inf)
b = cPickle.load(inf)
c = cPickle.load(inf)
inf.close()
```

还可以给 `cPickle.dump` 传递值为 2 的第三个参数，这相当于告诉 `cPickle.dump` 以二进制的形式（快速且紧凑）序列化数据，但同时数据文件也必须以二进制模式打开，而不能是默认的文本模式，无论你是想写入或是取出数据。

## 讨论

Python 提供了几种方法来序列化数据（比如，将数据转化为字节串，存入磁盘或数据库，或者通过网络传递）以及从序列化形式重新构建数据。最好的方式是使用 `cPickle` 模块。它有一个纯 Python 的对应物，叫做 `pickle` (`cPickle` 模块是用 C 编写的 Python 扩展)，但速度就慢得多了，只有在没有 `cPickle` 的情况下才应该考虑使用它（比如，将 Python 移植到容量很小的手机中，必须节省每一个字节，所以只能安装那些完全不可或缺的 Python 标准库的子集）。然而，在任何能够使用 `pickle` 的地方，都可以将其替换为 `cPickle`：可以用其中的一个将数据序列化，用另一个恢复并重新构建数据，而不会有任何问题。

`cPickle` 支持绝大多数的基本数据类型（如字典、列表、元组、数字、字符串）以及它们的各种组合形式，同时也支持类和实例。而对类和实例所做的处理仅仅涉及到数据，与代码无关（`cPickle` 对象并不知道怎样序列化代码对象，这可能是因为在不同的 Python 发行版之间的可移植性完全无法保证，因此序列化代码对象意义不大。如果你不需要考虑跨版本问题的话，可参看 7.6 节介绍的对代码对象的序列化方法）。7.4 节还有更多的用 `cPickle` 处理类和实例的细节。

`cPickle` 保证了在不同 Python 发行版之间的兼容性，也保证了对于特定计算机体系结构的独立性。即使你升级了你的 Python 发行版，通过 `cPickle` 处理的序列化数据仍然可以被读取，而且即使在不同的计算机上，这种序列化和反序列化操作也是保证可用的。

`cPickle` 的 `dumps` 函数接受任何 Python 数据结构作为参数，并返回一个字符串。如果用 2 作为 `dumps` 的第二个参数，`dumps` 将返回一个字节串：操作速度会更快，而且产生的串长度会更短。可以给 `loads` 函数传递字符串或者字节串，它都将返回一个 Python 数据结构，此数据等同于（`==`）原来的数据。在 `dumps` 和 `loads` 调用之间，可以以任意方式处置那个字符串或字节串，比如通过网络传输，存入数据库并读取，或者加密之后再解密。只要这个串本身的结构没有被改变，`loads` 都能够正确地读取（即使是在不同的平台上，或者用更新的发行的版本）。如果需要用老版本（早于 2.3）的 Python 处理数据，可考虑用 1 作为第二个参数：其操作速度会比较慢，产生的结果串也不如用 2 作参数产生的串紧凑短小，但这个串可以被老版本和最新的 Python，甚至以后的 Python 版本读取。

当需要将数据存入文件时，可以使用 cPickle 的 dump 函数，它接受两个参数：需要处理的数据结构和一个打开的文件对象，或者一个类文件对象。如果文件是以二进制输入输出方式打开的，而不是默认方式（文本模式），可以把 2 作为它的第三个参数，明确要求使用二进制格式，这样速度更快也更紧凑（或用 1 作为第三个参数来产生结果串，这样速度不快，要求的空间也更多，但是它能够被很老的，甚至早于 2.3 的 Python 版本读取）。dump 优于 dumps 的地方在于，通过 dump，可以进行几个调用，一个接一个地将不同的数据结构存入到同一个打开的文件。每个数据结构在存入时还会带有附加的信息，如这个串的长度。因此，当你以后打开此文件读取数据（二进制读取，如果你当初要求以二进制格式存入的话）并反复调用 cPickle.load 时，可将文件作为其参数，根据顺序一个接一个地构建出原来的数据结构。而 load 的返回值，就像 loads 的返回值一样，是一个完全等同于原数据的新的数据结构。

那些习惯于其他语言和库提供的“序列化”工具的用户可能会问，对于想要序列化和反序列化的对象的大小，pickle 会不会有什么限制。答案是：没有。你的计算机的内存可能是唯一的限制，如果你的计算机的内存非常大，pickle 在实践中几乎没有限制的。

## 更多资料

7.2 节和 7.4 节；*Library Reference* 和 *Python in a Nutshell* 中的标准库模块 cPickle 的文档。

## 7.3 在 Pickling 的时候压缩

感谢：Bill McNeill、Andrew Dalke

### 任务

你想以一种压缩的方式来 pickle 一般的 Python 对象。

### 解决方案

标准库模块 cPickle 和 gzip 提供了所需的功能；你只需以适当的方式将它们粘合起来即可：

```
import cPickle, gzip
def save(filename, *objects):
    ''' 将对象存为压缩过的磁盘文件 '''
    fil = gzip.open(filename, 'wb')
    for obj in objects: cPickle.dump(obj, fil, proto=2)
    fil.close()
def load(filename):
    ''' 从压缩的磁盘文件中载入对象 '''
    fil = gzip.open(filename, 'rb')
    while True:
```

```
try: yield cPickle.load(fil)
except EOFError: break
fil.close()
```

## 讨论

一般来说，持久化和压缩能够很好地协作。cPickle 使用协议 2 将 Python 对象存为一种紧凑的格式，但其生成的文件仍可以继续压缩。比如，在我的 Linux box 中，`open('/usr/share/dict/words').readlines()` 产生了超过 45 000 个字符串的列表。用协议 0 处理此列表产生的磁盘文件是 972KB，而用协议 2 则只需 716KB。然而，如果同时使用 gzip 和协议 2，如解决方案所示，产生的文件只有 268KB，这样能节省很多磁盘空间。实际情况是，协议 0 产生的结果压缩效果最好，同时使用 gzip 和协议 0 能够节省更多的磁盘空间，产生的文件只有 252KB。不过，268 和 252 之间的差异没有太大的意义，协议 2 有其他的优势，特别是在用于新风格类的实例时，所以我推荐解决方案中的函数混用方式。

不管你采用什么协议来储存数据，都不需要在重新载入数据的时候担心任何问题。协议本身是和数据一起存入文件的，所以 `cPickle.load` 能够检测出它需要采用的协议。只要给它传递一个文件实例或者一个带有 `read` 方法的伪文件对象，`cPickle.load` 会一个接一个地依照顺序返回每个对象，并且在完成整个文件时抛出 `EOFError` 异常。在本节方案中，我们用 `cPickle.load` 封装了一个生成器，这样你就可以用一个 `for` 声明简单地循环所有的对象，或者，根据你的需要，也可以使用类似于 `list(load('somefile.gz'))` 这样的调用，返回一个包含所有被恢复的对象的列表。

## 更多资料

*Library Reference* 中的 gzip 和 cPickle 模块。

## 7.4 对类和实例使用 cPickle 模块

感谢：Luther Blissett

### 任务

你想通过 cPickle 模块来存取类和实例对象。

### 解决方案

在用 cPickle 处理你的类和实例时，有很多需要注意的地方。比如，下面这段代码看上去似乎工作得很好：

```
import cPickle
class ForExample(object):
```

```
def __init__(self, *stuff):
    self.stuff = stuff
anInstance = ForExample('one', 2, 3)
saved = cPickle.dumps(anInstance)
reloaded = cPickle.loads(saved)
assert anInstance.stuff == reloaded.stuff
```

然而，有时仍会出一些问题：

```
anotherInstance = ForExample(1, 2, open('three', 'w'))
wontWork = cPickle.dumps(anotherInstance)
```

这个片段子句引发了一个 `TypeError: "can't pickle file objects"` 异常，这是因为 `anotherInstance` 包含了一个文件对象，而文件对象是无法用 `pickle` 处理的。如果你试图 `pickle` 另一个含有文件对象的容器，你会得到相同的异常。

然而，在某些情况下，可以为此做点什么：

```
class PrettyClever(object):
    def __init__(self, *stuff):
        self.stuff = stuff
    def __getstate__(self):
        def normalize(x):
            if isinstance(x, file):
                return 1, (x.name, x.mode, x.tell())
            return 0, x
        return [normalize(x) for x in self.stuff]
    def __setstate__(self, stuff):
        def reconstruct(x):
            if x[0] == 0:
                return x[1]
            name, mode, offs = x[1]
            openfile = open(name, mode)
            openfile.seek(offs)
            return openfile
        self.stuff = tuple([reconstruct(x) for x in stuff])
```

通过定义你的类的 `__getstate__` 和 `__setstate__` 特殊方法，你能够以某种更好的方式控制你的类实例，并影响到它把什么作为自己的状态。只要你能够用一种可被 `pickle` 处理的方式定义状态，并以某种对你的应用而言已经够用的方式从处理过的状态重新构建实例，你就能够为你的实例实现 `pickle` 操作及反操作。

## 讨论

`cPickle` 通过名字处理类和函数对象（比如通过它们的模块名和它们在模块内部的名字）。因此，你只能处理模块级别（不在其他类或函数中）的类。只有各个模块是能够被导入的，我们才能重新载入这些类对象。而且只有在实例属于这些类的前提下，它们才能够被保存和重新载入。除此之外，这些实例的状态也必须是可被 `pickle` 处理的。

默认情况下，实例的状态就是实例的`__dict__`的内容，再加上一些继承来的内建类型，如果有继承的话。比如，一个从`list`派生来的新风格类的实例包含列表的子项，这些子项也是它的状态的一个部分。`cPickle`同样能够处理定义或继承了名为`__slots__`（因此在那些预定义的槽中会含有实例的状态，而不是在`__dict__`中）的类属性的新风格类的实例。总的来说，对我们的应用而言，`cPickle`的默认方式通常就可以满足需要了。

然而有时候，你的实例会含有一些不能被`pickle`处理的属性或子项，并将这些属性或子项作为自身的状态（前一段提到过，`cPickle`默认会将这些属性或子项作为状态）。在本节中，作为示例，我展示了一个类的实例，它含有一个可以是任何类型的`stuff`，`stuff`有可能是打开的文件对象。为了处理这个问题，你的类可以定义特殊方法`__getstate__`。如果你的对象的类定义或继承了此方法，`cPickle`会对你的对象调用它，而不是直接到对象的`__dict__`中刨根问底（或者`__slots__`中，或内建类型的基类中）。

正常情况下，如果你定义了`__getstate__`方法，你也会定义`__setstate__`方法，就像解决方案中的做法。`__getstate__`可返回所有能被`pickle`处理的对象，以及被`pickle`处理过的对象，当以后进行反`pickle`操作的时候，那些对象还会被当做参数传递给`__setstate__`。在本节的解决方案中，`__getstate__`返回了一个列表，很像实例的默认状态（属性`self.stuff`），只是每个子项都被转变为一个含有两个子项的元组。如果元组子项的第一项被设置为0，则表示要逐字地接纳第二个子项，如果为1，则表示第二个子项会被用来构建一个打开的文件（当然，重新构建可能会失败或者无效。并没有一种通用的方法能够记录一个打开的文件的状态，这就是为什么`cPickle`连试都不想试一下。但在我们的应用环境下，我们可以假设给出的方法能够工作）。当我们进行反`pickle`操作并重新载入实例时，`cPickle`会对列表中的子项调用`__setstate__`，`__setstate__`能够用嵌套的`reconstruct`函数正确地处理每一对子项，并构建出`self.stuff`。对于那些在正常情况下无法用`pickle`处理的对象的状态，这种模式具有一定通用性，只是你要记得用不同的数字来标示你想支持的各种不同的“无法逐字接纳”的类型。

在某种特殊情况下，可以只定义`__getstate__`，而不定义`__setstate__`：`__getstate__`必须返回一个字典，而当我们进行反`pickle`操作并恢复实例时，就可以像使用实例的`__dict__`一样直接使用这个字典。在重新载入的时候，不能运行你自己的代码可能是一个不便之处，但可以使用`__getstate__`，它非常方便，用它的目的并不是为了保存状态，只是为了优化。一种可以用它优化的典型情况是，假设你的实例缓存了一些结果，当使用缓存时，如果访问到不存在的数据会引发重新计算，而你认为最好不要将这些缓存结果作为状态保存。在这种情况下，应当定义`__getstate__`来返回一个字典，作为实例的`__dict__`的不可或缺的一个子集（见4.13节，用一种简单而方便的方法“获取字典的子集”）。

除了`pickling`支持，定义`__getstate__`（以及通常也会定义的`__setstate__`）也会带给你更多的其他好处：如果一个类提供了这些方法而没有提供特殊方法`__copy__`或`__`

`deepcopy__`, 则这些方法不仅可以用于序列化也可以用于拷贝, 包括浅拷贝和深拷贝。如果`__getstate__`返回的状态数据是被深度复制的, 那么整个对象就被深度复制了, 不过, 除了这点区别, 通过`__getstate__`来实现的浅拷贝和深拷贝非常相似。见 4.1 节中更多的关于类如何控制它的实例的复制方式的内容。

无论是通过默认的 `pickling` 和 `unpickling` 方式, 还是使用自身的`__getstate__`和`__setstate__`, 当实例从被 `pickle` 过的状态恢复时, 它的特殊方法`__init__`都不会被调用。如果你来说最方便的方法是通过调用带有参数的`__init__`方法来重建实例, 你可能会需要定义一个特殊方法`__getinitargs__`, 而不是`__getstate__`。在这种情况下, `cPickle` 将不用参数调用此方法: 在重新载入的过程中, 这个方法必须返回一个可以被 `pickle` 处理的元组, `cPickle` 用此元组的子项作为`__init__`的参数并调用之。`__getinitargs__`, 就像`__getstate__`和`__setstate__`, 也可用于复制。

*Library Reference* 中关于 `pickle` 和 `copy_reg` 模块的内容涉及了几乎所有关于 `pickling` 和 `unpickling` 的细节, 甚至还包括了棘手的安全性问题, 特别是当你试图从一个不可信任的源 `unpickle` 数据的时候。(注意: 别这么做——如果你坚持这样做, Python 无法保护你) 不过, 我在这里谈到的一些技术对于绝大多数实践工作已经够用了, 只要别涉及太多的安全性问题(如果确实有安全性问题, 那么最具操作性的建议是: 不要用 `pickle`)。

## 更多资料

7.2 节; *Library Reference* 和 *Python in a Nutshell* 中关于标准库模块 `cPickle` 的文档。

# 7.5 Pickling 被绑定方法

感谢: Peter Cogolo

## 任务

需要 `pickle` 一个对象, 但是该对象持有(作为属性或子项)另一个对象的被绑定方法, 而被绑定方法是无法被 `pickle` 处理的。

## 解决方案

假设你有如下的对象:

```
import cPickle
class Greeter(object):
    def __init__(self, name):
        self.name = name
    def greet(self):
        print 'hello', self.name
```

```
class Repeater(object):
    def __init__(self, greeter):
        self.greeter = greeter
    def greet(self):
        self.greeter()
        self.greeter()
r = Repeater(Greeter('world')).greet
```

如果不是因为 `r` 持有了一个被绑定方法作为它的 `greeter` 属性，你本来可以很简单地实现对 `r` 的 `pickle` 处理：

```
s = cPickle.dumps(r)
```

而且，在遇到被绑定方法时，调用 `cPickle.dumps` 会抛出 `TypeError` 异常。一个简单的解决办法是让 `Repeater` 类的每个实例都不直接持有被绑定方法，而是持有一个可以被 `pickle` 处理的对该方法的封装。比如：

```
class picklable_boundmethod(object):
    def __init__(self, mt):
        self.mt = mt
    def __getstate__(self):
        return self.mt.im_self, self.mt.im_func.__name__
    def __setstate__(self, (s,fn)):
        self.mt = getattr(s, fn)
    def __call__(self, *a, **kw):
        return self.mt(*a, **kw)
```

现在，将 `Repeater.__init__` 的主体改成 `self.greeter = picklable_boundmethod(greeter)`，就可以保证以前的代码片段正常工作。

## 讨论

Python 标准库 `pickle` 模块（和比它更快的等价的表兄 `cPickle` 一样）是通过名字处理函数和类的，这意味着，只有在模块顶层定义的函数才能被 `pickle` 处理（即通过模块名和函数名 `pickle` 该函数）。

如果你有一系列的对象互相持有，不是直接持有，而是持有其他对象的一个被绑定方法（这在 Python 中常常是很好的想法），这种限制会让这一系列的对象都不能被 `pickle`。一个解决办法是教会 `pickle` 怎么序列化被绑定对象，就像 7.6 节那样。另一个可能的解决办法是定义适当的 `__getstate__` 和 `__setstate__` 方法，将被绑定方法转变成某种可以在 `dump` 的时候被 `pickle` 以及能够在 `load` 的时候被重建的东西，如 7.4 节那样。不过，如果你有较多的类实例互相持有被绑定方法，第二种办法并不是一个好的解决之道。

本节试图找到一个更简洁的思路，仍然是基于持有被绑定方法这个事实，但不再直接持有，而是通过 `picklable_boundmethod` 封装类持有。`picklable_boundmethod` 的编写有个前提，即假设唯一一件你要对被绑定方法做的事情是调用它，因此它仅仅把 `__call__` 功能给委托了（除此之外，你还应该使用 `__getattr__`，这样可以委托对其他属性的访问）。

你持有 `picklable_boundmethod` 的实例而非直接持有一个被绑定方法对象，这对普通操作来说是完全透明的。当 `pickle` 处理开始的时候，`picklable_boundmethod` 的特殊方法 `__getstate__` 会参与进来，这部分我们在前面的 7.4 节中介绍过。而 `picklable_boundmethod` 的 `__getstate__` 会返回被绑定方法所属的对象以及被绑定方法的函数名。之后，在 `unpickle` 操作的时候，`__setstate__` 会在重建对象的同时，对该名字使用内建的 `getattr` 来恢复一个等价的被绑定方法。这个方法不是百分之百精确完美的，因为一个对象可能会以假设的名字（与方法的真实函数名不同）持有它。不过，如果你不故意做某些怪异的事情来为难 `picklable_boundmethod`，应该不会遇到什么难以应付的麻烦。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 文档中关于 `pickle` 和 `cPickle`，被绑定方法对象，以及内建的 `getattr` 的内容。

# 7.6 Pickling 代码对象

感谢：Andres Tremols、Peter Cogolo

## 任务

你想对代码对象进行 `pickle` 处理，但标准库的 `pickle` 模块并不支持这个功能。

## 解决方案

可以通过使用 `copy_reg` 模块来扩展 `pickle` 模块的能力。下面的程序完成的工作是对代码对象进行 `pickle` 操作，但必须确保下列模块已经导入，而在 `unpickle` 的时候，则要求这些模块已经导入或者能够在需要的时候导入：

```
import new, types, copy_reg
def code_ctor(*args):
    # 委托给 new.code，构造一个新的代码对象
    return new.code(*args)
def reduce_code(co):
    # 还原函数必须返回一个带两个子项的元组：第一个，是
    # 在反序列化初期被用来构建参数对象的构造函数，第二个
    # 则是参数的元组，将被传递给构造函数作为参数
    if co.co_freevars or co.co_cellvars:
        raise ValueError, "Sorry, cannot pickle code objects from closures"
    return code_ctor, (co.co_argcount, co.co_nlocals, co.co_stacksize,
                      co.co_flags, co.co_code, co.co_consts, co.co_names,
                      co.co_varnames, co.co_filename, co.co_name, co.co_firstlineno,
                      co.co_lnotab)
# 注册 reductor 用于 pickle “CodeType” 类型的对象
copy_reg.pickle(types.CodeType, reduce_code)
```

```
if __name__ == '__main__':
    # pickle 代码对象用法示例
    import cPickle
    # afunction (which, inside, has a code object, of course)
    def f(x): print 'Hello,', x
    # 将此函数的代码对象序列化为字节串
    pickled_code = cPickle.dumps(f.func_code)
    # 从字节串恢复一个相等的代码对象
    recovered_code = cPickle.loads(pickled_code)
    # 用重建的代码对象创建一个新函数
    g = new.function(recovered_code, globals())
    # 看看调用新函数的效果
    g('world')
```

## 讨论

Python 标准库的 pickle 模块（和它的更快的等价的表兄 cPickle 一样）通过名字来处理函数和类。不过，对代码对象进行 pickle 处理之后所获得的字节码，并不能完全地确定函数（以及方法）的各个方面。在某些情况下，你可能更希望通过值把所有一切都用 pickle 处理，这样所有相关的东西都可以从字节码中恢复。有时可以用 marshal 而不是 pickle 来解决问题，不过 marshal 不允许你序列化代码对象，而且 marshal 在很多方面都有限制。比如，你不能用 marshal 处理你编写的类的实例（一旦你完成了对代码对象的序列化，那它就跟某个具体版本的 Python 绑定了，这样 pickle 也就拥有了 marshal 的最大的限制：不能保证在不同的 Python 版本之间的数据保存和数据重新载入的能力）。

一个可选的方案是继续发掘潜力，Python 标准库允许扩展 pickle 的可用的类型集。基本上，可以“教会”pickle 怎样保存和重新载入代码对象；反过来，它也可以让你根据值，而不是“名字”来 pickle 函数和类对象。（解决方案中的在 if \_\_name\_\_ == '\_\_main\_\_' 之下的代码展示了怎样通过扩展 pickle 来处理函数。）

要教会 pickle 一些新类型，我们得使用 copy\_reg，这也是 Python 标准库的一部分。通过函数 copy\_reg.pickle，可以为指定的类型注册一个还原函数。还原函数接受一个需要被 pickle 处理的实例作为参数，并返回一个含有两个子项的元组：一个构造函数，在重建实例的时候会被调用，以及一个参数元组，它会被传递给构造函数（还原函数也可以返回其他不同的结果，但基于本节的任务，两个子项的元组已经够用了）。

本节的模块定义了函数 reduce\_code，然后将该函数注册为类型 type.CodeType 的还原函数——type.CodeType 就是代码对象。当 reduce\_code 被调用时，它首先会检查它的代码对象是否来自于一个闭包（嵌套到其他函数），这是因为它不能处理这种情况，我也找不到办法来解决这个问题，所以如果碰到这种状况，reduce\_code 只是抛出个异常通知我们它碰到麻烦了。

一般情况下，`reduce_code` 返回 `code_ctor` 来作为构造函数，并返回一个由 `co` 的所有属性构成的元组，这个元组作为参数元组会被传递给构造函数。当代码对象重新载入时，`code_ctor` 会被调用并使用那些参数，它只是简单地把任务委托给 `new.code`，这才是真正的构造函数。不幸的是，`reduce_code` 不能返回 `new.code` 作为结果元组的第一个子项，这是因为 `new.code` 是内建的（用 C 编写的可调用体），通过内建名无法访问它。因此，大体上，`code_ctor` 主要的任务只是为真正干活的 `new.code` 提供一个名字。

解决方案中的 `if __name__ == '__main__'` 以下的部分提供了一个典型的玩具般的例子——它将一个代码对象序列化成了串，又从这个结果串中恢复出了一个拷贝，然后用这个拷贝创建函数并调用之。一个更典型的应用场合是，在一个脚本中用 `pickle` 完成序列化，然后在另一个脚本中完成反序列化。假设本节的模块被存为文件 `reco.py` 并被放入 Python 的 `sys.path` 目录，它就可以被其他 Python 脚本和模块导入了。这样你就可以在一个导入了 `reco` 的脚本中对代码对象进行序列化操作，像这样：

```
import reco, pickle
def f(x):
    print 'Hello,', x
pickle.dump(f.func_code, open('saved.pickle', 'wb'))
```

而 `unpickle` 反操作以及使用重新构建的代码对象的示例代码大概会是这样的：

```
import new, cPickle
c = cPickle.load(open('saved.pickle', 'rb'))
g = new.function(c, globals())
g('world')
```

注意，第二个脚本不需要导入 `reco`——这是因为导入会在需要时自动发生（`pickle` 存在 `saved.pickle` 中部分信息是为了重建对象而用的，它需要调用 `reco.code_ctor`；所以，它也知道首先要导入 `reco`）。我曾说过，`pickle` 和 `cPickle` 是可以互换的。`cPickle` 快一些，除此之外没有任何区别，如果你愿意的话，可以用两者之一对对象进行 `pickle` 处理，再用另一种完成重新载入。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中的 `pickle` 和 `cPickle`，以及 `copy_reg` 模块。

## 7.7 通过 `shelve` 修改对象

感谢：Luther Blissett

### 任务

你正在使用标准库模块 `shelve`。你用 `shelve` 处理过的一些值是易变的对象（mutable objects），而且你需要修改这些对象。

## 解决方案

shelve 模块提供了一种持久的字典——在强大的关系型数据库和简洁的 marshal、pickle、dbm 以及类似的文件格式之间，它有着重要的地位。然而，在使用 shelve 的时候有一些很典型的陷阱需要注意。先看看下面的交互式 Python 会话：

```
>>> import shelve
>>> # 创建一个简单的例子 shelf
>>> she = shelve.open('try.she', 'c')
>>> for c in 'spam': she[c] = {c:23}
...
>>> for c in she.keys(): print c, she[c]
...
p {'p': 23}
s {'s': 23}
a {'a': 23}
m {'m': 23}
>>> she.close()
```

这样我们就创建了 shelve 文件，向其中添加了数据，之后又关闭此文件，到现在为止一切正常。现在我们需要再次打开该文件并处理其中的数据：

```
>>> she=shelve.open('try.she', 'c')
>>> she['p']
{'p': 23}
>>> she['p']['p'] = 42
>>> she['p']
{'p': 23}
```

这是怎么回事？我们明明把这个值设为 42，怎么 shelve 好像根本不理会我们的设定？问题的关键在于我们处理的其实只是 shelve 给我们的一个临时对象而已，而不是“真家伙”。当我们以默认选项打开 shelve 时，就像上面做法一样，它并不会跟踪对临时对象的修改。一个可行的方案是我们给临时对象绑定一个名字，然后完成我们的修改，之后再把改过的对象赋值给 shelve 的子项：

```
>>> a = she['p']
>>> a['p'] = 42
>>> she['p'] = a
>>> she['p']
{'p': 42}
>>> she.close()
```

我们可以验证这个修改是否持久：

```
>>> she=shelve.open('try.she', 'c')
>>> for c in she.keys(): print c,she[c]
...
p {'p': 42}
s {'s': 23}
```

```
a {'a': 23}  
m {'m': 23}
```

一个更简单的方法是将 `writeback` 选项设置为 `True`, 然后打开 `shelve` 对象:

```
>>> she = shelve.open('try.she', 'c', writeback=True)
```

打开 `writeback` 选项后, `shelve` 会跟踪所有从文件生成的对象, 并在关闭之前将所有项回写到文件, 这是因为在这个过程中它们可能被修改过。虽然代码简化了不少, 但代价是高昂的, 尤其是内存的消耗增加了很多。当我们需要从一个用 `writeback=True` 选项打开的 `shelve` 对象中读取很多对象时, 即使我们只修改了这些对象中的少数几个, `shelve` 仍然会把所有对象放入内存, 这是因为它事先也不知道哪个对象会被修改。而前一个方法, 我们主动担起责任来通知 `shelve` 发生的变化 (通过将修改过的对象赋值回去), 需要多花工夫, 但是收获也不小, 因为它的效率很高。

## 讨论

Python 标准库模块 `shelve` 在很多应用场合下都是很方便的工具, 但它却隐藏了一个令人生厌的陷阱, 虽然在 Python 的在线文档中对此有很详细的说明, 但仍然很容易被忽略。假设你正在用 `shelve` 处理一些易变的对象, 如字典或列表。很自然的, 你可能会希望更改其中的一些对象, 比如调用一些修改的方法 (列表的 `append`、字典的 `update` 等) 或给对象的子项或属性赋予一个新值。不过, 当你在做这些事的时候, 变化并不会发生在 `shelve` 对象中。这是因为我们更改的不过是 `shelve` 对象通过 `__getitem__` 方法提供的一个临时对象而已, 而且 `shelve` 对象在默认情况下并不会跟踪临时对象的变化, 事实上, 在把临时对象返回给我们之后, 它就对临时对象完全不闻不问, 漠不关心了。

如前面代码所示, 一个方法是给临时对象绑定一个名字, 通过这个名字完成对临时对象的所有操作, 然后将更改过的新对象赋值给 `shelve` 对象的子项。当你给 `shelve` 对象的子项赋值时, `shelve` 对象的 `__setitem__` 方法会被调用, 它会以适当的方式更新 `shelve` 对象, 这样变化就被保存下来了。

另一个可选的方法是, 可以在打开 `shelve` 对象的时候增加 `writeback = True` 标志, 之后 `shelve` 会跟踪每一个它给你的对象, 并最终将它们存回磁盘。这个方法会节省一些琐碎的代码, 更加省心, 但是你要当心: 如果你从 `shelve` 对象读取很多子项, 但却只修改其中的少数几项, `writeback` 方式的开销会很大, 尤其是内存消耗。当你用 `writeback=True` 选项打开 `shelve` 时, 它会把所有它给你的子项放入内存, 并在最后保存, 这是因为它没有什么可靠的办法来判断你要改哪些项, 而且也不知道当你关闭 `shelve` 对象时究竟哪些项是被改过的。除非你要修改每一个读取的项 (或者 `shelve` 对象的大小和你的计算机内存总数比起来微不足道, 可以忽略), 否则我推荐前一种方法: 给你从 `shelve` 对象获得的需要修改的子项绑定一个名字, 改完之后将每个子项赋值回 `shelve` 对象。

## 更多资料

7.1 节和 7.2 节中的两种序列化方法；*Library Reference* 和 *Python in a Nutshell* 中关于标准库模块 `shelve` 的文档。

## 7.8 使用 Berkeley DB 数据库

感谢：Farhad Fouladi

### 任务

你想将一些数据做持久化处理，而且也想体验一下 Berkeley DB 数据库的简洁和高效。

### 解决方案

如果以前在你的计算机中安装过 Berkeley DB，Python 标准库附带的 `bsddb` 包（以及可选的 `bsddb3`，用于访问 Berkeley DB release 3.2 数据库）可以被用来作为 Berkeley DB 接口。为了得到 `bsddb` 或者 `bsddb3`，如果没有 `bsddb` 的话，应当在 `import` 声明的时候使用 `try/exception`：

```
try:  
    from bsddb import db          # 先试试 release 4  
except ImportError:  
    from bsddb3 import db         # 没有，再试试 release 3  
print db.DB_VERSION_STRING  
# 输出，示例: Sleepycat Software: Berkeley DB 4.1.25: (December 19, 2002)
```

为了创建一个数据库，我们要初始化一个 `db.DB` 对象，然后向它的 `open` 方法传入适当的参数并调用之，如下：

```
adb = db.DB()  
adb.open('db_filename', dbtype=db.DB_HASH, flags=db.DB_CREATE)
```

当你想创建一个数据库时，`db.DB_HASH` 是几种可以选择的访问方法中的一种，一个很常见的选择是 `db.DB_BTREE`，使用 B+树访问（如果你想以排序的方式获取记录，这很方便）。也可以选择构建一个内存数据库，不使用任何持久化文件，只要将 `None` 作为文件名和第一个参数传递给 `open` 方法即可。

一旦有了一个打开的 `db.DB` 实例，你就可以添加记录了。每条记录由两个字符串构成，键和数据：

```
for i, w in enumerate('some words for example'.split()):  
    adb.put(w, str(i))
```

可以通过数据库的游标访问记录：

```
def irecords(curs):
    record = curs.first( )
    while record:
        yield record
        record = curs.next( )
for key, data in irecords(adb.cursor( )):
    print 'key=%r, data=%r' % (key, data)
# 输出 (the order may vary):
# key='some', data='0'
# key='example', data='3'
# key='words', data='1'
# key='for', data='2'
```

做完之后，需要关闭数据库：

```
adb.close( )
```

以后，在同一个或其他的 Python 程序中，可以给新创建的 db.DB 实例的 open 方法传递同样的文件名，再次打开该数据库：

```
the_same_db = db.DB( )
the_same_db.open('db_filename')
```

然后用同样的方式继续工作：

```
the_same_db.put('skidoo', '23')          # 添加一条记录
the_same_db.put('words', 'sweet')         # 替换一条记录
for key, data in irecords(the_same_db.cursor( )):
    print 'key=%r, data=%r' % (key, data)
# 输出 (the order may vary):
# key='some', data='0'
# key='example', data='3'
# key='words', data='sweet'
# key='for', data='2'
# key='skidoo', data='23'
```

再次提醒，在所有操作结束之后不要忘记关闭数据库：

```
the_same_db.close( )
```

## 讨论

Berkeley DB 是一个流行的开源数据库。它不支持 SQL，但却很容易使用，并提供了优异的性能，如果你希望掌控全局的话，它给了你充分的自由来控制所发生的一切，可以通过大量的选项、标志以及方法来完成控制。除了 Python，我们也可以通过其他语言来访问 Berkeley DB：比如，可以用 Python 程序完成一些修改和查询，然后再使用一个独立的 C 程序，使用相同的基本开源库（可以从 Sleepycat 下载），对同样的数据库做同样的事情。

Python 标准库模块 shelve 可以使用 Berkeley DB 作为它的数据库引擎，就像它使用

cPickle 进行序列化操作一样。然而，如果记录都是由 pickle.dumps 产生的串，而这些串对于除 Python 之外的任何语言来说都是很难处理的数据，那你将无法用其他语言访问 Berkeley DB 数据库文件。通过 bsddb 直接访问 Berkeley DB，数据库引擎能够提供很多高级的功能，而这些都是 shelve 无法提供的。

### 数据库还是 pickle，或者两者都用？

pickle 和 marshal，以及数据库系统如 Berkeley DB 或关系型数据库，它们的应用场合有很大不同，但是仍然有一些重叠的地方。

pickle（以及 marshal）本质上是序列化处理：将 Python 对象转变成可以传输或者储存的 BLOB，之后可以由另一方接收或者恢复。序列化的数据是被用来重建 Python 对象的，基本上也只能被 Python 应用程序访问。而针对对象或对象的一部分的搜索和选择操作，pickle 不能提供任何支持。

数据库（Berkeley DB，关系型数据库，以及其他类型）本质上是以数据为核心的：可以保存或者获取成组的基本类型的数据（大多是字符串和数字），还能够得到很多有关选取和搜索（对于关系型数据，这种支持可以说是海量的）以及跨语言的支持。数据库对于将 Python 对象序列化成数据或者从数据中反序列化出 Python 对象一无所知。

数据库和序列化这两种方式也可以被混用。可以用 pickle 将 Python 对象序列化成字节码，然后用数据库储存，或者反过来。但 Python 标准库 shelve 模块只是在一个很底层的层面工作，比如，它可以借助 pickle 来完成序列化和反序列化，或使用 bsddb 来作为底层的数据库引擎。因此，不要认为两种方式是一种“竞争的”关系——而应该认为两者是一种互相补充的关系。

举个例子，如解决方案中的代码所示，创建一个带有 db.DB\_HASH 访问方式的数据库，我们可以获得最大的效率，不过，你可能也注意到了，用生成器 irecords 列出所有的记录时，哈希算法使得记录以一种随机的和无法预测的顺序排列。如果你想以一种排序的方式访问记录，应该使用 db.DB\_BTREE 方法。Berkeley DB 支持很多高级功能，如事务，可以直接访问数据库来获得这种功能，而不是试图通过 anydbm 或者 shelve 来访问。更多的关于 Python 标准库 bsddb 包的详细文档，请参看 <http://pybsddb.sourceforge.net/bsddb3.html>。而关于 Berkeley DB 本身的文档、下载以及其他资料，请访问 <http://www.sleepycat.com>。

### 更多资料

*Library Reference* 和 *Python in a Nutshell* 中关于模块 anydbm、shelve 以及 bsddb 的内容；位于 <http://pybsddb.sourceforge.net/bsddb3.html> 的关于 bsddb 和 bsddb3 的更多细节；关

于 Berkeley DB 本身的文档，下载等资料请参看 <http://www.sleepycat.com>。

## 7.9 访问 MySQL 数据库

感谢：Mark Nenadov

### 任务

你想访问一个 MySQL 数据库。

### 解决方案

MySQLdb 模块正是为这种任务而设计的：

```
import MySQLdb
# 创建一个连接对象，再用它创建游标
con = MySQLdb.connect(host="127.0.0.1", port=3306,
                      user="joe", passwd="egf42", db="tst")
cursor = con.cursor()
# 执行一个 SQL 字符串
sql = "SELECT * FROM Users"
cursor.execute(sql)
# 从游标中取出所有记录放入一个序列并关闭连接
results = cursor.fetchall()
con.close()
```

### 讨论

MySQLdb 的主页是 <http://sourceforge.net/projects/mysql-python>。它是用 Python DB API 2.0 完成的一个简单的实现，Python DB API 2.0 适用于 Python 2.3（以及更老的版本），以及从 3.22 到 4.0 的 MySQL，在本文编写过程中，它还没有正式地提供对 Python 2.4 的支持。不过，如果你安装了正确的 C 编译器来编译 Python 扩展（对于所有的 Linux、Mac OS X、以及其他 UNIX 用户，包括很多 Windows 开发者，这都是应当做的），用当前的源码编译的 MySQLdb 在 Python 2.4 中工作得很好。MySQLdb 的新版本也正在开发中，它将正式支持高于 2.3 的 Python 版本以及高于 4.0 的 MySQL。

如同其他的 Python DB API 实现（假设你已经下载并安装了需要的 Python 扩展，该数据库引擎也已经运行起来），可以导入模块并用适当的参数调用 `connect` 函数。调用 `connect` 时的关键参数依赖于涉及到的数据库：主机（默认是本地计算机）、用户名和密码以及数据库名，这些都是典型的参数。在本节代码中，我显式地传递了默认的本机 IP 地址以及默认的 MySQL 端口（3306），只是为了展示之用，其实只需要传递默认值即可（这样也可以使得你的代码更加清晰，更具可读性，也更好维护）。

`connect` 函数返回了一个连接对象，可以对这个对象继续调用各种方法；当你完成所有

操作之后，应当对它调用 `close` 方法来关闭连接。你常常需要对连接对象调用的方法是 `cursor`，它可以返回一个游标对象，你通过这个对象向数据库传递 SQL 命令并获取命令的结果。MySQL 数据库引擎事实上是不支持 SQL 游标的，不过这也不算什么问题——MySQLdb 模块能够完全透明地为你模拟游标，这是因为 Python DB API 2.0 对游标的需求不大，不过这并不意味着可以在一个完全不支持游标的数据库中使用如 `WHERE CURRENT OF CURSOR` 这样的 SQL 语句。一旦你得到了一个 `cursor` 对象，你就可以对它调用各种方法。本节使用了 `execute` 方法来执行一个 SQL 声明，然后使用 `fetchall` 方法来获得所有的结果，其结果是一个元组的序列——每一行的结果用一个元组表示。还可以使用更多的优化和改良措施，但这些基本的 Python DB API 功能已经能够满足很多任务的需求了。

## 更多资料

Python-MySQL 接口模块 (<http://sourceforge.net/projects/mysql-python>)；Python DB API (<http://www.python.org/topics/database/DatabaseAPI-2.0.html>)；*Python in a Nutshell* 中的 DB API 文档。

## 7.10 在 MySQL 数据库中储存 BLOB

感谢：Luther Blissett

### 任务

你想把一个二进制的大对象（BLOB）存入 MySQL 数据库。

### 解决方案

MySQLdb 模块并不支持完整的占位符，不过可以使用模块的 `escape_string` 函数来解决：

```
import MySQLdb, cPickle
# 连接到数据库，用你的本机来测试数据库，并获得游标
connection = MySQLdb.connect(db="test")
cursor = connection.cursor()
# 创建一个新表以用于试验
cursor.execute("CREATE TABLE justatest (name TEXT, ablob BLOB)")
try:
    # 准备一些 BLOB 用于测试
    names = 'aramis', 'athos', 'porthos'
    data = {}
    for name in names:
        datum = list(name)
        datum.sort()
        data[name] = cPickle.dumps(datum, 2)
    # 执行插入
    cursor.executemany("INSERT INTO justatest VALUES (%s, %s)", data.items())
    connection.commit()
finally:
    cursor.close()
    connection.close()
```

```
sql = "INSERT INTO justatest VALUES(%s, %s)"
for name in names:
    cursor.execute(sql, (name, MySQLdb.escape_string(data[name])) )
# 恢复数据以进行检查
sql = "SELECT name, ablob FROM justatest ORDER BY name"
cursor.execute(sql)
for name, blob in cursor.fetchall():
    print name, cPickle.loads(blob), cPickle.loads(data[name])
finally:
    # 完成。删除表并关闭连接
    cursor.execute("DROP TABLE justatest")
    connection.close()
```

## 讨论

MySQL 支持二进制数据 (BLOG 和它的其他变种)，但当你通过 SQL 操纵此类数据时，必须得小心。尤其是，当你使用正常的 INSERT SQL 语句并想把二进制串放入你想插入的记录中时，应当根据 MySQL 的规则对二进制串中的一些字符进行转义操作。幸运的是，你完全不用自己去搞明白它的规则：MySQL 提供了一个函数用于处理转义，MySQLdb 也开放了这个接口给你的 Python 程序，那就是 `escape_string` 函数。

本节展示了一个典型的应用：你要插入的 BLOB 来自 `cPickle.dumps`，所以它们可能代表着任何 Python 对象（当然，在这个例子中，我们只是使用了一些字符的列表）。由于代码的目的是纯粹的展示，所以创建的表在最后会被删除（使用了 `try/finally` 声明，即使程序由于某些未处理的异常而必须终止时，我们也能确保收尾工作正常完成）。如果使用较新的 MySQL 和 MySQLdb，你甚至都不需要调用 `escape_string` 函数，因此还可以把相关的语句改得更加简洁：

```
cursor.execute(sql, (name, data[name]))
```

## 更多资料

见 7.11 节和 7.12 节的分别针对 PostgreSQL 和 SQLite 的同一个问题的解决方案；MySQL 的主页 (<http://www.mysql.org>)，Python/MySQL 接口模块 (<http://sourceforge.net/projects/mysql-python>)。

## 7.11 在 PostgreSQL 中储存 BLOB

感谢：Luther Blissett

### 任务

需要将 BLOB 存入一个 PostgreSQL 数据库。

## 解决方案

PostgreSQL 7.2 以及更新的版本支持大对象，而 psycopg 模块提供了二进制转义函数：

```
import psycopg, cPickle
# 连接到数据库，用你的本机来测试数据库，并获得游标
connection = psycopg.connect("dbname=test")
cursor = connection.cursor()
# 创建一个新表用于试验
cursor.execute("CREATE TABLE justatest (name TEXT, ablob BYTEA)")
try:
    # 准备一些 BLOB 用于测试
    names = 'aramis', 'athos', 'porthos'
    data = { }
    for name in names:
        datum = list(name)
        datum.sort( )
        data[name] = cPickle.dumps(datum, 2)
    # 执行插入
    sql = "INSERT INTO justatest VALUES(%s, %s)"
    for name in names:
        cursor.execute(sql, (name, psycopg.Binary(data[name])))
    # 恢复数据以进行检查
    sql = "SELECT name, ablob FROM justatest ORDER BY name"
    cursor.execute(sql)
    for name, blob in cursor.fetchall():
        print name, cPickle.loads(blob), cPickle.loads(data[name])
finally:
    # 完成。删除表并关闭连接
    cursor.execute("DROP TABLE justatest")
    connection.close()
```

## 讨论

PostgreSQL 支持二进制数据（BYTEA 及其变种），但当你通过 SQL 操纵此类数据时，必须得小心。尤其是，当你使用正常的 INSERT SQL 语句并想把二进制串放入你想插入的记录中时，应当根据 PostgreSQL 的规则对二进制串中的一些字符进行转义操作。幸运的是，你完全不用自己去搞明白它的规则：PostgreSQL 提供了一些函数用于处理转义，而且 psycopg 开放了这个接口给你的 Python 程序，那就是 Binary 函数。本节展示了一个典型的应用：你要插入的 BLOB 来自 cPickle.dumps，所以它们可能代表着任何 Python 对象（当然，在这个例子中，我们只是使用了一些字符的列表）。由于代码的目的是纯粹的展示，所以创建的表在最后会被删除（使用了 try/finally 声明，以确保即使程序由于某些未处理的异常必须终止时，收尾工作仍能完成）。

早期的 PostgreSQL 发行版本限制你在数据库的一个普通字段里最多只能存几 KB 的数

据。要想储存真正的大数据对象，必须得使用一些迂回的方法才能将数据存入数据库（比如 PostgreSQL 提供的非标准的 SQL 函数 LO\_IMPORT，它可以将位于运行数据库的计算机中的数据文件作为一个对象载入到数据库中，而且函数的使用还要求超级用户的权限），而且只能存在 OID 类型的字段中，以后还要以一种非直接的方式恢复对象。幸运的是，这些方法现在已经不再需要了：从 7.1 发行版开始（在本文编写的时候最新版本是 8.0），PostgreSQL 内置了 TOAST 项目的成果，避免了对字段的值的大小限制，也避免了那种迂回的非直接方式。psycopg 模块提供了方便的 Binary 函数来将任意二进制字节串转化为可以被 SQL 的 INSERT 和 UPDATE 接受的形式。

## 更多资料

见 7.10 节和 7.12 节的分别针对 MySQL 和 SQLite 的同一问题的解决方案；PostgreSQL 的主页 (<http://www.postgresql.org>)；Python/PostgreSQL 模块 (<http://initd.org/software psycopg>)。

## 7.12 在 SQLite 中储存 BLOB

感谢：John Barham

### 任务

你想将 BLOB 存入一个 SQLite 数据库。

### 解决方案

Python 的 PySQLite 扩展提供了 `sqlite.encode` 函数，它可帮助你在 SQLite 数据库中插入二进制串。可以基于这个函数编写一个小巧的适配器类：

```
import sqlite, cPickle
class Blob(object):
    ''' 自动转换二进制串 '''
    def __init__(self, s): self.s = s
    def __quote__(self): return "'%s'" % sqlite.encode(self.s)
# 创建一个测试的内存数据库，获得游标，创建一个表
connection = sqlite.connect(':memory:')
cursor = connection.cursor()
cursor.execute("CREATE TABLE justatest (name TEXT, ablob BLOB)")
# 准备一些 BLOB 用于测试
names = 'aramis', 'athos', 'porthos'
data = {}
for name in names:
    datum = list(name)
    datum.sort()
    data[name] = cPickle.dumps(datum, 2)
```

```

# 执行插入
sql = 'INSERT INTO justatest VALUES(%s, %s)'
for name in names:
    cursor.execute(sql, (name, Blob(data[name])))
# 恢复数据以进行检查
sql = 'SELECT name, ablob FROM justatest ORDER BY name'
cursor.execute(sql)
for name, blob in cursor.fetchall():
    print name, cPickle.loads(blob), cPickle.loads(data[name])
# 完成。删除表并关闭连接
connection.close()

```

## 讨论

SQLite 并不直接支持二进制数据，但它仍然允许你在一个 CREATE TABLE 的 DDL 语句中声明字段为二进制类型。当你对 SQLite 数据库执行 SELECT 语句并取得了数据时，Python 的 PySQLite 扩展能够使用这些被声明的类型，并将字段数据转化 Python 值。尽管如此，使用 SQL 操纵此类数据时我们仍需小心。

具体地说，当使用正常的 INSERT 或 UPDATE 的 SQL 语句并想把二进制串放入你想插入的记录时，应当根据 SQLite 的规则将二进制串中的一些字符进行转义。幸运的是，你完全不用自己去搞明白它的规则：SQLite 提供了函数用于转义处理，而且 PySQLite 扩展开放了这个接口给你的 Python 程序，那就是 `sqlite.encode` 函数。本节展示了一个典型的应用：你要插入的 BLOB 来自 `cPickle.dumps`，所以它们可能代表着任何 Python 对象（当然，在这个例子中，我们只是使用了一些字符的列表）。由于代码的目的是纯粹的展示，创建的也是一个内存数据库，所以在脚本运行之后这个数据库会被丢弃。

当你将二进制串作为参数传递给游标的 `execute` 方法时，也可以直接调用 `sqlite.encode` 来处理数据，但本节使用的方法却有点不同，我们定义了一个 `Blob` 类来封装二进制串，并把 `Blob` 的实例传递给 `execute`。PySQLite 收到的任何作为参数的类实例，其所属的类都必须定义一个叫做 `_quote` 的方法，PySQLite 会对每个实例调用此方法，并期待此方法返回一个已经准备妥当的可用于 SQL 的 INSERT 语句串。当你对自定义的复杂类应用此法时，可能需要给 `connect` 方法传递一个 `decoders` 关键字参数，以便将正确的解码函数和特定的 SQL 类型联系起来。在默认情况下，BLOB 类型是和解码函数 `sqlite.decode` 相联系的，它正好是 `sqlite.encode` 的反向函数；对于本节的简单的 `Blob` 类，由于默认的解码函数已经能够完美工作，我们也没必要指定其他的解码函数。

## 更多资料

见 7.10 节和 7.11 节的分别针对 MySQL 和 PostgreSQL 的同一问题的解决方案；SQLite 的主页 (<http://www.sqlite.org>)；PySQLite 手册 (<http://pysqlite.sourceforge.net/manual.html>)；SQLite 的 FAQ (“ Does SQLite support a BLOB type?”)，见 <http://www.hwaci.com/sw/sqlite/faq.html#q12>。

## 7.13 生成一个字典将字段名映射为列号

感谢: Thomas T. Jenkins

### 任务

你想访问一个从 DB API 游标对象获得的数据，但你希望能够根据字段名访问列，而不是根据列号。

### 解决方案

通过列号访问从数据库获得的记录的列是可行的，但可读性不太好，而且健壮性也差，这是因为当数据库的模式发生变化时，列有可能会被重新排序（很少见，但是确实偶有发生）。本节将利用 Python DB API 的游标对象的 `description` 属性，创建一个字典来将列名映射为列号，可以通过 `cursor_row[field_dict[fieldname]]` 来获得命名的列的值：

```
def fields(cursor):
    """ 假设 DB API 2.0 游标对象已经被执行并返回了
    a dictionary that maps each field name to a column index, 0 and up. """
    results = { }
    for column, desc in enumerate(cursor.description):
        results[desc[0]] = column
    return results
```

### 讨论

当你使用 `cursor` 的各种 `fetch` 方法 (`fetchone`、`fetchmany`、`fetchall`) 获得了一个记录行的集合时，你肯定希望能够通过列名而不是列号，来访问某行的特定的列的值。本节展示的函数接受一个 DB API 2.0 游标对象为参数，并返回一个以列名为键，列号为值的字典。

下面是用法示例（假设你已经将解决方案的代码存为 `dbutils.py`，并放入了 Python 的 `sys.path` 目录）。`conn` 是兼容于任何 DB API 2 的 Python 模块的连接对象，我们的代码从它开始。

```
>>> c = conn.cursor( )
>>> c.execute('''select * from country_region_goal
...             where crg_region_code is null''')
>>> import pprint
>>> pp = pprint.pprint
>>> pp(c.description)
([('CRG_ID', 4, None, None, 10, 0, 0),
 ('CRG_PROGRAM_ID', 4, None, None, 10, 0, 1),
 ('CRG_FISCAL_YEAR', 12, None, None, 4, 0, 1),
 ('CRG_REGION_CODE', 12, None, None, 3, 0, 1),
```

```

('CRG_COUNTRY_CODE', 12, None, None, 2, 0, 1),
('CRG_GOAL_CODE', 12, None, None, 2, 0, 1),
('CRG_FUNDING_AMOUNT', 8, None, None, 15, 0, 1))
>>> import dbutils
>>> field_dict = dbutils.fields(c)
>>> pp(field_dict)
{'CRG_COUNTRY_CODE': 4,
'CRG_FISCAL_YEAR': 2,
'CRG_FUNDING_AMOUNT': 6,
'CRG_GOAL_CODE': 5,
'CRG_ID': 0,
'CRG_PROGRAM_ID': 1,
'CRG_REGION_CODE': 3}
>>> row = c.fetchone()
>>> pp(row)
(45, 3, '2000', None, 'HR', '26', 48509.0)
>>> ctry_code = row[field_dict['CRG_COUNTRY_CODE']]
>>> print ctry_code
HR
>>> fund = row[field_dict['CRG_FUNDING_AMOUNT']]
>>> print fund
48509.0

```

如果你认为通过 `row[field_dict['CRG_COUNTRY_CODE']]` 的访问不够优雅，可以将行和字段字典封装到一个对象中，从而允许更优雅的访问方式。下面给出一个简单的例子：

```

class neater(object):
    def __init__(self, row, field_dict):
        self.r = row
        self.d = field_dict
    def __getattr__(self, name):
        try:
            return self.r[self.d[name]]
        except LookupError:
            raise AttributeError

```

如果 `neater` 类也在你的 `dbutils` 模块中，可以继续进行那个交互式过程，像这样：

```

>>> row = dbutils.neater(row, field_dict)
>>> print row.CRG_FUNDING_AMOUNT
48509.0

```

不过，如果对这种漂亮的方式很有兴趣，我建议你不要自己去实现，应当先看看 7.14 节 `dbtuple` 模块的示例。比起浪费时间实现自己的各种构架，重复使用好的、坚固的且经受过考验的代码明显是更聪明的选择。

## 更多资料

见 7.14 节给出的利用第三方的 `dbtuple` 模块完成类似任务的更精致的方式。

## 7.14 利用 dtuple 实现对查询结果的灵活访问

感谢: Steve Holden、Hamish Lawson、Kevin Jacobs

### 任务

你想通过列名或列号，以一种灵活的方式来访问序列，比如访问数据库查询获得的行序列。

### 解决方案

使用已有的好代码总是比自己动手去实现要聪明。对于本节的任务，Greg Stein 的 dtuple 模块提供了一个很好的解决方法：

```
import dtuple
import mx.ODBC.Windows as odbc
flist = ["Name", "Num", "LinkText"]
descr = dtuple.TupleDescriptor([[n] for n in flist])
conn = odbc.connect("HoldenWebSQL")      # 连接数据库
curs = conn.cursor()                      # 创建游标
sql = """SELECT %s FROM StdPage
          WHERE PageSet='Std' AND Num<25
          ORDER BY PageSet, Num""", ".join(flist)
print sql
curs.execute(sql)
rows = curs.fetchall()
for row in rows:
    row = dtuple.DatabaseTuple(descr, row)
    print "Attribute: Name: %s Number: %d" % (row.Name, row.Num or 0)
    print "Subscript: Name: %s Number: %d" % (row[0], row[1] or 0)
    print "Mapping:   Name: %s Number: %d" % (row["Name"], row["Num"] or 0)
conn.close()
```

### 讨论

有时，Python 编程新手在使用数据库时会感到烦恼，因为 DB API 兼容模块展现的查询结果是一个元组的列表。而元组只能用数字标注，所以使用这些查询结果的代码可能变得晦涩难懂，难于维护。Greg Stein 的 dtuple 模块，可以从 <http://www.lyra.org/greg/python/dtuple.py> 下载，它定义了两个有用的类从而解决了问题：TupleDescriptor 和 DatabaseTuple。为了能够访问任意的 SQL 数据库，本节方案借助 mxODBC 模块使用了 ODBC 协议，见 <http://www.egenix.com/files/python/mxODBC.html>，但换用任何其他的标准 DB API 兼容模块也不会影响本节的任务。

TupleDescriptor 类通过一个序列的列表（每个子序列的第一个元素都是列名）创建了一

个描述的元组。用这种序列描述数据总是很方便的。举个例子，对于一个交互式的应用程序，每个列名后面都可能跟着验证信息，如数据类型和允许的长度。使用 `TupleDescriptor` 的目的是为了便于创建 `DatabaseTuple` 对象。对于这个特定的应用，除了名字之外，列的其他信息都不是必需的，因此，要求的序列列表其实是根据字段名列表通过列表推导（list comprehension）创建的一个单列表（只含有一个元素的列表）的列表。

`DatabaseTuple` 是通过 `TupleDescriptor` 和一个元组（如数据库的行）创建的对象，它的元素既可以通过数字标注（就像元组），也可以通过列名标注（就像字典）。如果列名是合法的 Python 名，也可以将列作为你的 `DatabaseTuple` 的属性来访问。有人可能会反对这种子项和属性的交叉混用，但对于这个例子是个很实用的选择。如果 Python 不是一门实用的语言，那它就什么也不是了，所以我并不认为这种方便有什么错。

为了展示 `DatabaseTuple` 的效用，本节的简单程序创建了 `TupleDescriptor`，接着用它把 SQL 查询获得的每一个记录行转换为 `DatabaseTuple`。另外，代码使用了同样的字段名列表来创建 `TupleDescriptor` 和 SQL 的 SELECT 语句，这说明将数据库代码参数化是一件相对比较容易的事情。

另外，如果你希望获得所有的字段（一个 SQL 的 SELECT \* 查询），并从游标中动态地获得字段名，如前面 7.13 节那样，可以这么做，首先删掉变量 `flist`，你再也用不着它了，然后，将变量 `descr` 的创建移到对游标的 `execute` 方法的调用之后：

```
curs.execute(sql)
descr = dtuple.TupleDescriptor(curs.description)
```

代码的其他部分完全不用改变。

OPAL Group 免费提供的 Python Database Row Module（也称为 `db_row`），提供了一个更复杂的方式，其功能和 `dtuple` 很相似但性能更好。关于它的下载地址以及其他信息，详见：<http://opensource.theopalgroup.com>。

模块 `pysqlite` 能够操作内存关系型数据库和由 SQLite 库封装的文件，但对于如 `fetchall` 之类的调用并不返回真正的元组：它返回的是一个对元组进行了简便包装的类的实例，而且它还允许通过属性访问字段的语法，和本节提供的方法很相似。

## 更多资料

见 7.13 节提供的更简单，功能更弱的转化字段名到列号的方法；`dtuple` 模块，见 <http://www.lyra.org/greg/python/dtuple.py>；OPAL 的 `db_row`，见 <http://opensource.theopalgroup.com>；SQLite，一种快速、轻量级、嵌入式的关系新数据库（<http://www.sqlite.org>），以及 Python DB API 接口模块 `pysqlite`（<http://pysqlite.sourceforge.net>）。

## 7.15 打印数据库游标的内容

感谢: Steve Holden、Farhad Fouladi、Rosendo Martinez、David Berry、Kevin Ryan

### 任务

你想使用合适的列头（以及可选的宽度）来展现一次查询的结果，但你不希望在程序中采取硬编码的方式。事实上，在你编写代码的时候，你甚至对列头和宽度都一无所知。

### 解决方案

动态地探测列头和宽度是一种非常灵活的方法，对于此类展现数据的任务，这种方法具有较高的代码复用价值：

```
def pp(cursor, data=None, check_row_lengths=False):
    if not data:
        data = cursor.fetchall()
    names = []
    lengths = []
    rules = []
    for col, field_description in enumerate(cursor.description):
        field_name = field_description[0]
        names.append(field_name)
        field_length = field_description[2] or 12
        field_length = max(field_length, len(field_name))
        if check_row_lengths:
            # 如果不可靠 复查字段长度
            data_length = max([len(str(row[col])) for row in data])
            field_length = max(field_length, data_length)
        lengths.append(field_length)
        rules.append('-' * field_length)
    format = " ".join(["%%-%ss" % l for l in lengths])
    result = [format % tuple(names), format % tuple(rules)]
    for row in data:
        result.append(format % tuple(row))
    return "\n".join(result)
```

### 讨论

关系型数据库常被认为难于使用，而 Python DB API 大大简化了使用难度。但如果您的程序需要和几种不同的数据库引擎打交道，你不得不做一些工作来协调不同模块在实现上的差异，甚至还要协调连接的数据库引擎的差异，这部分工作有时候也挺乏味的。关于数据库的一个典型问题是，当你对库中的数据不了解时，怎么实现对查询结果的展示。本节方案利用了游标的 `description` 属性来获得正确的列头，另外还提供了可选

的宽度检查，即检查输出的每一行以确保列宽度是足够的。

在某些情况下，游标能够生成它返回的数据的可靠的描述，但不是所有的数据库模块都那么友善，提供那么好用的游标。解决方案展示的打印函数 `pp` 以一个游标为它的第一个参数，当然你首先必须对这个游标进行一次数据获取操作（通常是执行了 SQL 的 `SELECT` 语句）。对于返回的数据它也接受可选的参数，还可以先把数据提取出来用于其他目的，比如使用 `fetchall` 方法从游标中获得数据，然后再将数据传递给 `pp` 的 `data` 参数。第二个可选的参数被用来通知打印函数必须检查数据的列长度，而不是无条件地信任游标的描述；检查数据的列长度比较消耗时间，但对于某些 RDBMS 引擎和 DB API 模块的组合，数据检查是很有必要的，因为由它们的游标的 `description` 属性给出的宽度值非常不准确。

我们给出一个简单的测试程序来展示第二个可选参数的价值，这个例子使用了 `mxODBC` 模块和 Microsoft Jet 数据库：

```
import mx.ODBC.Windows as odbc
import dbcp # 包含 pp 函数
conn = odbc.connect("MyDSN")
curs = conn.cursor()
curs.execute("""SELECT Name, LinkText, Pageset FROM StdPage
                ORDER BY PageSet, Name""")
rows = curs.fetchall()
print "\n\nWithout rowlens:"
print dbcp.pp(curs, rows)
print "\n\nWith rowlens:"
print dbcp.pp(curs, rows, rowlens=1)
conn.close()
```

在这种情况下，游标的 `description` 并不包括列长度。第一个输出说明了默认长度为 12 的列长度不够用。而第二个输出则通过检查数据长度修正了这个问题：

```
Without rowlens:
Name      LinkText      Pageset
-----  -----
ERROR      ERROR: Cannot Locate Page None
home      Home None
consult    Consulting Activity Std
ffx       FactFaxer     Std
hardware  Hardware Platforms Std
python    Python        Std
rates     Rates         Std
technol   Technologies Std
wcb      WebCallback  Std

With rowlens:
Name      LinkText          Pageset
-----  -----
ERROR      ERROR: Cannot Locate Page None
```

home	Home	None
consult	Consulting Activity	Std
ffx	FactFaxer	Std
hardware	Hardware Platforms	Std
python	Python	Std
rates	Rates	Std
technol	Technologies	Std
wcb	WebCallback	Std

模块 `pysqlite` 能够操作内存关系型数据库和由 SQLite 库封装的文件，但它也无法为字段长度提供可靠的游标描述。而且，对于如 `fetchall` 之类的调用它并不返回真正的元组：它返回的是一个对元组进行了简便包装的类的实例，而且它还允许通过属性访问字段的语法和 7.14 节展示的方法很相似。为了处理这种略微偏离 DB API 规格的小变种，本节方案很谨慎地使用了 `tuple(row)`，而不仅仅是 `row`，来作为 `result.append(format % tuple(row))` 语句中操作符%的右操作元。Python 的语义规定了如果右操作元不是一个元组，那么左操作元（格式化字符串）只能包含一个格式指定符。我们使用元组来作为右操作元的原因是，创建并使用带有多个格式指定符的格式化字符串是本节的要点之一。

本节中的函数在测试时尤其有用，因为可以很容易地验证你是否从数据库返回了需要的数据。对于某些查询，函数的输出结果对用户而言已经足够漂亮了。目前这个函数还不能表现 null 值，只能表示 DB API 返回的 `None`，不过对它做一些修改以便支持 null 字符串或者其他有意义的值也是很容易的。

## 更多资料

`mxODBC` 包，一个针对 ODBC 的 DB API 兼容接口 (<http://www.egenix.com/files/python/mxODBC.html>)；`SQLite`，一种快速、轻量级、嵌入式的关系新数据库 (<http://www.sqlite.org>)，以及它的 Python DB API 接口模块 `pysqlite` (<http://pysqlite.sourceforge.net>)。

## 7.16 适用于各种 DB API 模块的单参数传递风格

感谢：Denis S. Otkidach

### 任务

你想编写可以运行在任何 DB API 模块下的 Python 程序，但是这些模块却使用不同的方式传递参数。

### 解决方案

我们需要一些支持函数将 SQL 的查询和参数转化为 5 种可能的参数传递方式：

```
class Param(object):
    ''' 封装单参数的类 '''
    def __init__(self, value):
        self.value = value
    def __repr__(self):
        return 'Param(%r)' % (self.value,)
def to_qmark(chunks):
    ''' 用"?"风格准备 SQL 查询 '''
    query_parts = [ ]
    params = [ ]
    for chunk in chunks:
        if isinstance(chunk, Param):
            params.append(chunk.value)
            query_parts.append('?')
        else:
            query_parts.append(chunk)
    return ''.join(query_parts), params
def to_numeric(chunks):
    ''' 用":1"风格准备 SQL 查询 '''
    query_parts = [ ]
    params = [ ]
    for chunk in chunks:
        if isinstance(chunk, Param):
            params.append(chunk.value)
            query_parts.append(':%d' % len(params))
        else:
            query_parts.append(chunk)
    # DCOracle2 needs, specifically, a _tuple_ of parameters:
    return ''.join(query_parts), tuple(params)
def to_named(chunks):
    ''' 用":name"风格准备 SQL 查询 '''
    query_parts = [ ]
    params = { }
    for chunk in chunks:
        if isinstance(chunk, Param):
            name = 'p%d' % len(params)
            params[name] = chunk.value
            query_parts.append(':%s' % name)
        else:
            query_parts.append(chunk)
    return ''.join(query_parts), params
def to_format(chunks):
    ''' 用"%s"风格准备 SQL 查询 '''
    query_parts = [ ]
    params = [ ]
    for chunk in chunks:
        if isinstance(chunk, Param):
            params.append(chunk.value)
            query_parts.append('%s')
        else:
```

```

        query_parts.append(chunk.replace('%', '%%'))
    return ''.join(query_parts), params
def to_pyformat(chunks):
    '''用"%s"风格准备SQL查询'''
    query_parts = []
    params = {}
    for chunk in chunks:
        if isinstance(chunk, Param):
            name = 'p%d' % len(params)
            params[name] = chunk.value
            query_parts.append('%%(%s)s' % name)
        else:
            query_parts.append(chunk.replace('%', '%%'))
    return ''.join(query_parts), params
converter = {}
for paramstyle in ('qmark', 'numeric', 'named', 'format', 'pyformat'):
    converter[paramstyle] = globals['to_' + param_style]
def execute(cursor, converter, chunked_query):
    query, params = converter(chunked_query)
    return cursor.execute(query, params)
if __name__=='__main__':
    query = ('SELECT * FROM test WHERE field1>', Param(10),
             ' AND field2 LIKE ', Param('%value%'))
    print 'Query:', query
    for paramstyle in ('qmark', 'numeric', 'named', 'format', 'pyformat'):
        print '%s: %r' % (paramstyle, converter[param_style](query))

```

## 讨论

DB API 规格说明有个很烦人的问题：它允许符合规格的模块使用 5 种参数风格中的任意一种。所以，你无法简单地通过更改数据库模块切换到其他数据库：如果这两个模块的参数传递风格不一致，需要重写所有使用了参数替换的 SQL 查询。通过使用本节的方案，你也许可以稍微改善一下这种窘境。可以从 converter 字典挑一个合适的转换函数（用你当前的 DB API 模块的 paramstyle 属性作为索引），用 SQL 字符串和方案中给出的 Param 类的实例来编写查询语句（如代码中的 if \_\_name\_\_=='\_\_main\_\_' 部分），然后用 execute 函数执行查询。无论如何，这不是一个完美的方案，但比没有强多了！

## 更多资料

DB API 文档，见 <http://www.python.org/peps/pep-0249.html>；符合 DB API 的模块列表，见 <http://www.python.org/topics/database/modules.html>。

## 7.17 通过 ADO 使用 Microsoft Jet

感谢： Souman Deb

## 任务

你想通过 Microsoft 的 ADO 访问 Microsoft Jet 数据库，例如，工作在 Apache Web 服务器上的用 Python 编写的 CGI 脚本很可能有这种需求。

## 解决方案

CGI 脚本一定是存在于 Apache 的 cgi-bin 目录中的，我们可以使用 PyWin32 扩展，通过 COM 连接到 ADO 以及 Microsoft Jet。举个例子：

```
#!C:\Python23\python
print "Content-type:text/html\n\n"
import win32com
db='C:\\Program Files\\Microsoft
    Office\\Office\\Samples\\Northwind.mdb'
MAX_ROWS=2155
def connect(query):
    con = win32com.client.Dispatch('ADODB.Connection')
    con.Open("Provider=Microsoft.Jet.OLEDB.4.0; Data Source="+db)
    result_set = con.Execute(query + ';')
    con.Close( )
    return result_set
def display(columns, MAX_ROWS):
    print "<table border=1>"
    print "<th>Order ID</th>"
    print "<th>Product</th>"
    print "<th>Unit Price</th>"
    print "<th>Quantity</th>"
    print "<th>Discount</th>"
    for k in range(MAX_ROWS):
        print "<tr>"
        for field in columns:
            print "<td>", field[k], "</td>"
        print "</tr>"
    print "</table>"
result_set = connect("select * from [Order details]")
columns = result_set[0].GetRows(MAX_ROWS)
display(columns, MAX_ROWS)
result_set[0].Close
```

## 讨论

本例使用了 Microsoft 发布的几个产品中的“Northwind Database”示例，Microsoft Access 也带有这个例子。为了运行此程序，需要一台运行着 Microsoft Windows 的计算机，还得确保安装了其他的 Microsoft 附件，如 OLEDB、ADO 以及 Jet 数据库驱动，Jet 数据库驱动常常被人认为是“Access 数据库”（其实是不正确的）。（Microsoft Access 是用于创建数据库前端应用程序的产品，它可以和其他数据库驱动一起工作，如 Microsoft SQL

Server，而不是免费分发的可以下载的 Microsoft Jet 数据库引擎。) 而且，你还需要安装 Mark Hammond 的 PyWin32 包 (曾用过 win32all 这个名字)；而 ActiveState 的 Python 发行版，ActivePython，已经集成了 PyWin32 (当然不仅仅是 PyWin32)。

如果你想将本节代码作为一个 Apache CGI 脚本来运行，那么，你首先应该先安装 Apache，并把此脚本放入 cgi-bin 目录 (Apache 会在此目录寻找 CGI 脚本，而 cgi-bin 的目录位置取决于你是如何安装 Apache 的)。

然后，你要确保脚本中的路径都是正确的，这取决于 Python.exe 和 Northwind.mdb 在你计算机上的安装位置。代码中的路径是默认安装 Python 2.3 和“Northwind”示例数据库后的路径。如果脚本不能正常工作，可以检查 Apache 的 error.log 文件，其中的错误信息能够帮助你迅速定位问题。

假设你把脚本存为了 cgi-bin/adoexample.py，而且你的 Apache 服务器工作得很正常，那么可以用任意浏览器打开 URL：<http://localhost/cgi-bin/adoexample.py>。Python 通过 ADO 访问 Jet 数据库的一个已知的限制是类型为 currency 的字段：返回的这个字段的数据是某种奇怪的元组，而不是简单的数字。本节代码并没有对此限制作任何处理。

## 更多资料

关于 PyWin32 的 Win32 API 文档 (<http://starship.python.net/crew/mhammond/win32/Downloads.html>) 和 ActivePython (<http://www.activestate.com/ActivePython>)；Microsoft 的 Windows API 文档 (<http://msdn.microsoft.com>)；Mark Hammond 和 Andy Robinson 合著的 *Python Programming on Win32* (O'Reilly)。

## 7.18 从 Jython Servlet 访问 JDBC 数据库

感谢：Brian Zhou

### 任务

你正在编写 Jython 的一个 servlet，而且需要通过 JDBC 连接到一个数据库服务器 (如 Oracle、Sybase、Microsoft SQL Server 和 MySQL)。

### 解决方案

基本上对于各种数据库方法都是大同小异的，区别只是语句多几行或少几行而已。下面是使用 Oracle 数据库的代码：

```
import java, javax
class emp(javax.servlet.http.HttpServlet):
    def doGet(self, request, response):
        ''' 遇到获取查询时，Servlet 一般是在回应输出流中写入
```

结果。在这个例子中我们忽略了请求，但在正常情况下，那正是我们获得表单输入的地方

```
'''  
# 我们回以文本，所以必须相应地设置 content type  
response.setContentType("text/plain")  
# 获得输出流，用它完成查询，再关闭  
out = response.getOutputStream()  
self.dbQuery(out)  
out.close()  
def dbQuery(self, out):  
    # 连接到 Oracle 驱动，创建它的一个实例  
    driver = "oracle.jdbc.driver.OracleDriver"  
    java.lang.Class.forName(driver).newInstance()  
    # 指定用户名和密码获得一个连接  
    server, db = "server", "ORCL"  
    url = "jdbc:oracle:thin:@{} + server + ":" + db  
    usr, passwd = "scott", "tiger"  
    conn = java.sql.DriverManager.getConnection(url, usr, passwd)  
    # 向连接对象发送一个 SQL 查询  
    query = "SELECT EMPNO, ENAME, JOB FROM EMP"  
    stmt = conn.createStatement()  
    if stmt.execute(query):  
        # 获得查询结果并打印到输出流  
        rs = stmt.getResultSet()  
        while rs and rs.next():  
            out.println(rs.getString("EMPNO"))  
            out.println(rs.getString("ENAME"))  
            out.println(rs.getString("JOB"))  
            out.println()  
        stmt.close()  
        conn.close()
```

如果你的数据库是 Sybase 或者 Microsoft SQL Server，使用下列代码（上例中的注释部分此处不再显示，完全通用）：

```
import java, javax  
class titles(javax.servlet.http.HttpServlet):  
    def doGet(self, request, response):  
        response.setContentType("text/plain")  
        out = response.getOutputStream()  
        self.dbQuery(out)  
        out.close()  
    def dbQuery(self, out):  
        driver = "sun.jdbc.odbc.JdbcOdbcDriver"  
        java.lang.Class.forName(driver).newInstance()  
        # Use "pubs" DB for mssql and "pubs2" for Sybase  
        url = "jdbc:odbc:myDataSource"  
        usr, passwd = "sa", "password"  
        conn = java.sql.DriverManager.getConnection(url, usr, passwd)
```

```

query = "select title, price, ytd_sales, pubdate from titles"
stmt = conn.createStatement( )
if stmt.execute(query):
    rs = stmt.getResultSet( )
    while rs and rs.next( ):
        out.println(rs.getString("title"))
        if rs.getObject("price"):
            out.println("%2.2f" % rs.getFloat("price"))
        else:
            out.println("null")
        if rs.getObject("ytd_sales"):
            out.println(rs.getInt("ytd_sales"))
        else:
            out.println("null")
        out.println(rs.getTimestamp("pubdate").toString( ))
        out.println( )
stmt.close( )
conn.close( )

```

下面代码用于 MySQL：

```

import java, javax
class goosebumps(javax.servlet.http.HttpServlet):
    def doGet(self, request, response):
        response.setContentType("text/plain")
        out = response.getOutputStream( )
        self.dbQuery(out)
        out.close( )
    def dbQuery(self, out):
        driver = "org.gjt.mm.mysql.Driver"
        java.lang.Class.forName(driver).newInstance( )
        server, db = "server", "test"
        usr, passwd = "root", "password"
        url = "jdbc:mysql://%s/%s?user=%s&password=%s" % (
            server, db, usr, passwd)
        conn = java.sql.DriverManager.getConnection(url)
        query = "select country, monster from goosebumps"
        stmt = conn.createStatement( )
        if stmt.execute(query):
            rs = stmt.getResultSet( )
            while rs and rs.next( ):
                out.println(rs.getString("country"))
                out.println(rs.getString("monster"))
                out.println( )
            stmt.close( )

```

## 讨论

你可能使用不同的 JDBC 驱动和 URL，但你会发现基本技术既简单又相似。本节的代码使用了 text/plain 的内容类型，这是因为本文的主旨是访问数据库，不是格式化数据。

很显然，根据你的应用的需要，可以将内容类型改成其他类型。

在每个示例中，第一步都是通过 Java 动态载入工具实例化我们需要的驱动（它的包名，作为字符串，存在了变量 `driver` 中）。`java.lang.Class` 类的 `forName` 方法载入并提供相关的 Java 类，这个类的 `newInstance` 方法确保了我们需要的驱动被实例化。然后，我们提供正确的 URL（或者用户名和密码）调用 `java.sql.DriverManager` 的 `getConnection` 方法，得到一个连接对象，并把它放入变量 `conn` 中。通过这个连接对象，我们可以用 `createStatement` 创建语句对象，并用它的 `execute` 方法执行保存在变量 `query` 中的查询语句。如果查询成功，我们就用 `getResultSet` 方法获得结果。最后，Oracle 和 MySQL 还允许对结果进行简单的顺序导航（sequential navigation），这样我们就可以轻松展示所有结果，不过对于 Sybase 和 Microsoft SQL Server 则需要多花点功夫。总的来说，流程都是相似的。

## 更多资料

Jython 主页 (<http://www.jython.org>)，JDBC 的主页 (<http://java.sun.com/products/jdbc>)。

## 7.19 通过 Jython 和 ODBC 获得 Excel 数据

感谢：Zabil CM

### 任务

你的 Jython 脚本需要从 Microsoft Excel 文件中获取数据。

### 解决方案

Jython，就像 Java 一样，可以通过 JDBC-ODBC 桥访问 ODBC，而 Microsoft Excel 也能够通过 ODBC 被查询：

```
from java import lang, sql
lang.Class.forName('sun.jdbc.odbc.JdbcOdbcDriver')
excel_file = 'values.xls'
connection = sql.DriverManager.getConnection(
    'jdbc:odbc:Driver={Microsoft Excel Driver (*.xls)};DBQ=%s;READONLY=true' %
    excelfile, '', '')
# Sheet1 是我们需要的 Excel 工作簿的名字。查询的字段名
# 是由第一行的每列的值设定的
record_set = connection.createStatement().executeQuery(
    'SELECT * FROM [Sheet1$]')
# print the first-column field of every record (==row)
# 打印每个记录的第一列字段
while record_set.next():
```

```
    print record_set.getString(1)
# 完成后, 关闭连接和记录集
record_set.close()
connection.close()
```

## 讨论

本节的方案最适于运行在 Microsoft Window 平台, Windows 平台一般都安装并配置了 ODBC, 而且 Microsoft Excel ODBC 驱动也能得到最佳的支持。不过, 如果有合适的商用产品, 也可以在 Apple Macintosh 或者任何其他 UNIX 版本上使用本节代码。

比起其他的选择, 使用 ODBC 来访问 Microsoft Excel 有一个很大的优点, 我们前面并未提及: 使用 ODBC, 你就可以使用一个宽广的 SQL 子集。比如, 通过一个 WHERE 子句, 你能够很容易地获得工作簿的行的子集, 如下:

```
SELECT * FROM [Sheet1$] WHERE DEPARTMENT=9
```

由于所有的选择逻辑都可以轻松地被表示为你传递给 executeSQL 方法的 SQL 字符串, 这就使得此方案可以被封装为一个简单的, 可复用的函数。

如果你使用的是传统的 Python (CPython) 而不是 Jython, 你无法使用 JDBC, 但可以直接使用 ODBC (比如遵从 DB API 规范的 mxODBC, <http://www.egenix.com/files/python/mxODBC.html>) 以相似的方式完成本节的任务。

## 更多资料

Jython 主页 (<http://www.jython.org>) ; JDBC 的主页 (<http://java.sun.com/products/jdbc>)。12.7 节提供的另一种访问 Excel 数据的方法 (通过解析 XML 文件可以要求 Excel 输出)。

# 调试和测试

### 引言

感谢：Mark Hammond, *Python Programming on Win32* (O'Reilly) 合著者

我在家里拥有的第一台计算机是一台 64KB 内存的 Z80 CP/M 机器。在家中拥有计算机意味着我有很多时间来深度发掘这个有趣的玩具。Turbo Pascal 当时刚刚发布，它看上去比我曾经用过的 BASIC 语言和汇编语言都要先进。甚至在那个时候，我就喜欢为自己的程序开发可重用的库，随着我的技能和经验的提升，我仍然执着于创建工具并协助开发者们工作，热情丝毫不逊于开发面向终端用户的应用程序。

为开发者们创建工具意味着必须非常重视调试和测试。虽然你可能会将这些问题理解为交互式的调试器，但调试和测试的概念比你这个想法要宽泛得多。调试和测试有时候是完全不可分割的。测试总是导致新的 bug 被发现。你会一直不断地调试，直到你相信自己已经明白了引发错误的原因并做了必要的修改。反复漂洗总是必要的。

在调试和测试中遇到的问题常常都很隐秘。我很喜欢 Python 的 `assert` 声明，每次我用它的时候，我都在调试和测试自己的程序。大型项目通常会在程序中构建自己的调试和测试的能力，比如中央日志以及错误处理。对于大型项目，究竟是这种方式好呢，还是我前面提到的那种事后检查的方式更好，这个问题还存在争论。

Python 提供了很多技术来帮助开发者们。Python 的内省和动态特质（这是 Guido 的 we-are-all-consenting-adults 编程哲学的产物）意味着调试和测试的方法的唯一限制可能只是你的想象力。可以在运行时替换掉函数，给类增加方法，获得程序的所有你需要知道的信息。一切都发生在运行时，一切都极其简单和 Python 化。

在本章中将会遇到一个新的主题，单元测试，对于今天的编程方式而言，它和传统的测试大不相同，后者只是发掘一个已经编码完毕的系统中的 bug。而现在越来越多的程序员开始接受单元测试并在开发的初期实施这种测试方法，以便在第一时间阻止 bug 的

产生，并且它对于重构、优化以及移植也很有意义。Python 的标准库提供了两个模块专用于单元测试，`unittest` 和 `doctest`，而且，Python 2.4 还提供了一个沟通两者的桥，你会在本章中的一节看到其详细内容。如果你还不了解单元测试的概念，想获得更多信息和指导，本章将正合你胃口。而且，你还会在本章中发现不少关于这个主题的近期的书的引用和链接。

除了测试，在本章中你还能找到不少很棒的窍门，即使是最严苛的评论家也会喜欢这部分内容。无论你需要的是错误日志定制，Python 回溯信息的深入诊断，还是垃圾处理的优化，你都来对了地方。所以，请系上餐巾，菜来了！

## 8.1 阻止某些条件和循环的执行

感谢：Chris McDonough、Srinivas B、Dinu Gherman

### 任务

在开发或者调试的时候，有时候你会希望某些条件或循环部分的代码可以被暂时忽略掉。

### 解决方案

最简单的方法是编辑你的代码，在 `if` 或 `while` 关键字之后插入 `0: #`。由于 0 被估值为 `False`，那段代码就不会被执行了。比如：

```
if i < 1:  
    doSomething()  
while j < k:  
    j = fleep(j, k)
```

改成：

```
if 0: # i < 1:  
    doSomething()  
while 0: # j < k:  
    j = fleep(j, k)
```

在开发和调试过程中，如果有很多这样的代码片需要同时地打开或关闭，你还有一个选择，即，定义一个布尔变量（一般称为标志），假设 `doit = False`，代码如下：

```
if doit and i < 1:  
    doSomething()  
while doit and j < k:  
    j = fleep(j, k)
```

通过这种方式，可以临时地允许所有的代码片段执行，只要将标志重新设置一下：`doit = True`，当然反过来也很容易。还可以定义很多这种标志。但是，请记住当你完成开发和调试工作后，应当移除那些 `doit and` 部分，因为那时这些部分唯一的作用就是降低运行速度。

## 讨论

当然，你还有别的选择。很多好的编辑器提供了很好的命令来插入或移除注释标记，比如随 Python 发布的 IDLE IDE (Integrated Development Environment) 提供的 Alt-3 和 Alt-4；在这种编辑器中一种常见的惯例是用两个注释标记，##，来标记那些临时注释行，并以此区别于那些“正常的”注释。

还可以使用 Python 中的一种特有的技术，即，只读的全局变量 `__debug__`。如果 Python 运行时没有-O (optimize) 命令行参数选项，`__debug__` 的值是 `True`，如果带有该选项，`__debug__` 的值是 `False`。而且，Python 编译器知道 `__debug__` 的存在，当 Python 运行时带有优化选项，编译器会移除所有被 `if __debug__` 保护的代码块，这样可以节省更多内存，并提升执行速度。

## 更多资料

*Language Reference* 和 *Python in a Nutshell* 中关于 `__debug__` 和 `assert` 声明的内容。

## 8.2 在 Linux 上测量内存使用

感谢：Jean Brouwers

### 任务

你很想监控运行在 Linux 上的程序究竟占用了多少内存，然而标准库模块 `resource` 并不能在 Linux 中正确地工作。

### 解决方案

我们可以基于 Linux 的`/proc`伪文件系统编写我们的测量程序：

```
import os
_proc_status = '/proc/%d/status' % os.getpid()
_scale = {'kB': 1024.0, 'mB': 1024.0*1024.0,
          'KB': 1024.0, 'MB': 1024.0*1024.0}
def _VmB(VmKey):
    ''' 给定 VmKey 字符串，返回字节数 '''
    # 获得伪文件/proc/<pid>/status
    try:
        t = open(_proc_status)
        v = t.read()
        t.close()
    except IOError:
        return 0.0 # non-Linux?
    # 获得 VmKey 行，如'VmRSS: 9999 kB\n ...'
    i = v.index(VmKey)
```

```

v = v[i:].split(None, 3) # 依照空白符切割
if len(v) < 3:
    return 0.0 # 无效格式?
# 将 Vm 值转成字节
return float(v[1]) * _scale[v[2]]
def memory(since=0.0):
    ''' 返回虚拟内存使用的字节数 '''
    return _VmB('VmSize:') - since
def resident(since=0.0):
    ''' 返回常驻内存使用的字节数 '''
    return _VmB('VmRSS:') - since
def stacksize(since=0.0):
    ''' 返回栈使用的字节数 '''
    return _VmB('VmStk:') - since

```

## 讨论

本节的每个函数都接受一个可选的参数 `since`，因为这些函数的典型用法是测量由于某个代码片段而多占用了的内存（虚拟、常驻或者栈）。用 `since` 作为可选参数使得它们的典型用法变得更加简洁和优雅：

```

m0 = memory()
section of code you're monitoring
m1 = memory(m0)
print 'The monitored section consumed', m1, 'bytes of virtual memory'.

```

获取并解析`/proc/pid/status`的伪文件内容可能不是最有效的获得内存使用情况的方法，而且也无法移植到非 Linux 系统。不过它很简单而且代码容易编写，而且，在非 Linux 系统上，也可以直接调用 Python 标准库的 `resource` 模块。

事实上，也可以在 Linux 上用 `resource`，只不过内存消耗的各个字段，如 `ru_maxrss`，都是一个恒定的 0 值，就像 Linux 的 shell 命令 `time` 输出的各个内存消耗字段一样。出现这种现象的根本原因是 Linux 的系统调用 `getrusage` 在实现上的限制，可以用 `man getrusage` 查看相关文档。

## 更多资料

*Library Reference* 中的标准库模块 `resource` 的文档。

## 8.3 调试垃圾回收进程

感谢：Dirk Holtwick

### 任务

你已经知道你的程序中有内存泄漏现象，但你并不清楚究竟是什么东西造成的。因此

需要获得更多的信息来帮助你定位泄漏发生的地点，这样就可以修复问题并更好地减轻定期执行垃圾回收工作的标准模块 gc 的负担。

## 解决方案

可以借助 gc 模块来探究垃圾收集的奥秘：

```
import gc
def dump_garbage( ):
    """ 展示垃圾都是什么东西"""
    # 强制收集
    print "\nGARBAGE:"
    gc.collect()
    print "\nGARBAGE OBJECTS:"
    for x in gc.garbage:
        s = str(x)
        if len(s) > 80: s = s[:77] + '...'
        print type(x),"\n  ", s
    if __name__=="__main__":
        gc.enable()
        gc.set_debug(gc.DEBUG_LEAK)
        # 模拟一个泄漏（一个引用自身的列表）并展示
        l = []
        l.append(l)
        del l
        dump_garbage()
    # 输出:
    # GARBAGE:
    # gc: collectable <list 0x38c6e8>
    # GARBAGE OBJECTS:
    # <type 'list'>
    #   [[...]]
```

## 讨论

除了 gc 正常的调试输出，本节还展示了一些垃圾对象，这是为了帮助你了解泄漏。应当避免能够造成循环垃圾回收的情况。大多数时候，这种现象的原因都是由于某些对象引用了自身，或者形成了类似的一个引用循环（也被称为引用环）。

Python 提供了各种工具来清除引用循环，尤其是弱引用（标准库模块 `weakref`），一旦你发现了引用循环，你就可以轻松完成清除工作。但在大程序里，你首先得知道在什么地方找到泄漏，然后你才能修补漏洞并提高程序性能。基于这个目的，如果我们能够知道泄漏的对象的类型，必将对泄漏的定位有很大好处，而本节的 `dump_garbage` 函数正好能够满足这种需求。

解决方案的第一步是调用 `gc.set_debug`，通知 gc 模块将泄漏的对象放入 `gc.garbage` 列表中，而不是将它们回收利用。然后，`dump_garbage` 函数调用 `gc.collect` 来强制垃圾

回收进程的运行，即使当前内存还很充裕，这样它就可以检查 `gc.garbage` 中的每一项并打印出类型和内容（将每个垃圾对象的输出限制为 80 个字符，以避免大块信息充斥整个屏幕）。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中的 `gc` 和 `weakref` 模块的文档。

## 8.4 捕获和记录异常

感谢：Mike Foord

### 任务

你需要捕获异常并记录它们的回溯和错误信息，然后再继续执行程序。

### 解决方案

一种典型的情况是程序需要一个接一个地处理许多独立的文件。其中一些的格式可能是有问题的，因而会引发异常。而你需要捕获这种异常，并记录错误信息，然后继续你的文件处理流程。比如：

```
import cStringIO, traceback
def process_all_files(all_filenames,
                      fatal_exceptions=(KeyboardInterrupt, MemoryError)):
    bad_filenames = { }
    for one_filename in all_filenames:
        try:
            process_one_file(one_filename)
        except fatal_exceptions:
            raise
        except Exception:
            f = cStringIO.StringIO( )
            traceback.print_exc(file=f)
            bad_filenames[one_filename] = f.getvalue()
    return bad_filenames
```

### 讨论

Python 的异常是一种极其强有力的工具，但你需要一个清晰而简洁的思路来应用这种工具。本节的内容可能不会正好符合你的需求，但从应用策略设计的角度来说，它是一个良好的开始。

本节的方法来自于我的一个应用程序，我编写该程序的目的为了解析一些被认为是某

种固定格式的文本。但是坏格式的文件非常容易引起异常，我必须获得那些错误回溯信息，以便以后将程序编写得更加具有容错性或者即时地修复坏格式文件。不过与此同时，我还得让我的程序继续它的文件处理流程。

一个要点是，不是所有的异常都应该被截获、记录，而且还允许程序继续运行。一个 `KeyboardInterrupt` 异常意味着用户按下了 `Ctrl-C`（或者 `Ctrl-Break`，或其他的键组合），这是一个很明确的终止程序的要求；我们当然应该尊重这种请求，而不是无礼地忽略之。一个 `MemoryError` 异常意味着内存不足——除非你缓存了前期的一些重要结果，不然应该迅速删除缓存以提供更多的内存，一般来说，对于这种情况，你能做的事情很少。具体依赖于你的应用程序以及运行环境，一些其他的错误也可能被认为是致命的。因此，`process_all_files` 接受一个 `fatal_exceptions` 参数，这是一个异常类的元组，指明了它不应该截获异常（而是应该任由这些异常继续扩散），默认的异常就是我们刚刚提到的那两种。一个设计良好的 `try/except` 声明的结构可以捕获且继续传递那些它不应该处理的异常，并利用通用的 `except Exception` 处理子句来截获其他所有的异常。

在通用的处理子句中，我们以最简单的方式获得了所有的错误和回溯信息：通过 `traceback.print_exc` 将消息和回溯写入到“文件”，这个所谓的“文件”事实上是 `cStringIO.StringIO` 的一个实例，一种为了简化内存型信息的获取而设计的类文件对象，一般来说它的内容仍然需要写入到真实的文件中。`StringIO` 实例的 `getvalue` 方法将消息和回溯作为字符串返回，我们再把这个字符串存入字典 `bad_filenames` 中，用引发问题的文件的名字作为对应键。`process_all_files` 的 `for` 循环会继续移到下一个它需要处理的文件。

当 `process_all_files` 完成了处理，它会返回字典 `bad_filenames`，如果在处理过程中没有发生问题，这个字典应该是空的。一些顶层的调用 `process_all_files` 的代码会以适当的方式使用此字典，展示或者存储它持有的错误信息。

在技术上，抛出非内建 `Exception` 的子类的异常也是可能的（虽然已经废弃了），你甚至可以抛出字符串。如果你想截获这种不同常规的异常（出于向后兼容的目的，这种可能性肯定会长达很多年），你需要在 `try/except` 声明之后再添加一个无条件的 `except` 子句：

```
except fatal_exceptions:  
    raise  
except Exception:  
    ...  
except:  
    ...
```

当然，如果你想截获所有的正常（非致命）异常，以及那种非常规的情况，你完全不需要 `except Exception` 子句——使用无条件的 `except` 即可。不过一般来说，你可能会希望能够用一些更好的方法记录非常规异常，因为现在（已经是 21 世纪了）那种异常在任何环境下都绝对都不是我们所期望的。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中的 traceback 和 cStringIO 模块的内容；*Language Reference* 和 *Python in a Nutshell* 中关于 try/except 以及 exception 类的内容。

## 8.5 在调试模式中跟踪表达式和注释

感谢：Olivier Dagenais

### 任务

你正在编写的程序无法使用可交互的、单步执行的调试器。因此，你需要详细记录状态和控制分支的信息来进行高效的调试。

### 解决方案

traceback 模块的 extract\_stack 函数是解决问题的关键，此函数使得你的调试代码能够很容易地执行运行时内省（runtime introspection），从而找出调用它的代码：

```
import sys, traceback
traceOutput = sys.stdout
watchOutput = sys.stdout
rawOutput = sys.stdout
# 调用 watch(secretOfUniverse) 打印出如下的信息：
# File "trace.py", line 57, in __testTrace
#   secretOfUniverse <int> = 42
watch_format = ('File "%(fileName)s", line %(lineNumber)d, in'
                '%(methodName)s\n  %(varName)s <%(varType)s>\n'
                '  = %(value)s\n\n')
def watch(variableName):
    if __debug__:
        stack = traceback.extract_stack()[-2:][0]
        actualCall = stack[3]
        if actualCall is None:
            actualCall = "watch([unknown])"
        left = actualCall.find('(')
        right = actualCall.rfind(')')
        paramDict = dict(varName=actualCall[left+1:right]).strip(),
                    varType=str(type(variableName))[7:-2],
                    value=repr(variableName),
                    methodName=stack[2],
                    lineNumber=stack[1],
                    fileName=stack[0])
        watchOutput.write(watch_format % paramDict)
# 调用 trace("this line was executed") 打印出如下信息：
# File "trace.py", line 64, in ?
```

```

#     this line was executed
trace_format = ('File "%(fileName)s", line %(lineNumber)d, in'
                ' %(methodName)s\n    %(text)s\n\n')
def trace(text):
    if __debug__:
        stack = traceback.extract_stack()[-2:][0]
        paramDict = dict(text=text,
                          methodName=stack[2],
                          lineNumber=stack[1],
                          fileName=stack[0])
        watchOutput.write(trace_format % paramDict)
# 调用 raw("some raw text") 打印出如下信息:
# some raw text
def raw(text):
    if __debug__:
        rawOutput.write(text)

```

## 讨论

对于很多各种类型的程序来说，要想使用传统的、交互式的单步调试器是很不容易的。比如 CGI（公共网关接口）程序；对服务器的访问一般是通过 Web 进行的，或者是通过 CORBA、XML-RPC 或 SOAP 协议；再比如 Windows 服务和 UNIX 守护进程。

可以在程序中加上一堆 print 语句来应对这种无法交互调试的困境，但这种方式很不系统，而且在问题修复之后还要做清理工作。本节展示的是一种架构良好的可行方式，即提供一些函数使得可以打印出表达式、变量或函数调用的值，同时还带有范围信息、跟踪信息以及通用的注释。

traceback 模块的关键就是 extract\_stack 函数。traceback.extract\_stack 返回的是 4 个子项的元组的列表，4 个子项分别是文件名、行号、函数名以及调用声明的代码，每个元组都代表着栈中的一次调用记录。列表的 Item[-2]（倒数第二个子项）是直接调用者的元组，它也正是我们将要输出到类文件对象（绑定到了 traceOutput 和 watchOutput 变量）的信息。

如果你将 traceOutput、watchOutput 或 rawOutput 变量绑定到适合的类文件对象，各种输出都会被定向到你希望的地方去。当 \_\_debug\_\_ 是 False 的时候（当你以 -O 或 -OO 开关运行 Python 解释器的时候），所有与调试相关的代码都会被自动清除。这种方式不会使得你的执行码变大，因为编译器知道 \_\_debug\_\_ 变量的存在，当打开了优化选项，它会清除被 if \_\_debug\_\_ 保护的代码。

下面是一个使用示例，它把所有的信息打印到了标准输出设备，采用的测试方式也是通常我们用的自测的方式，即将测试用例直接放到模块的末尾：

```

def __testTrace():
    secretOfUniverse = 42
    watch(secretOfUniverse)
if __name__ == "__main__":

```

```
a = "something else"
watch(a)
__testTrace( )
trace("This line was executed!")
raw("Just some raw text...")
```

当仅运行 Python 命令时（没有-O 开关），代码输出：

```
File "trace.py", line 61, in ?
    a <str> = 'something else'
File "trace.py", line 57, in __testTrace
    secretOfUniverse <int> = 42
File "trace.py", line 64, in ?
    This line was executed!
Just some raw text...
```

本节代码的输出被故意设计得很像旧的 Python 1.5.2 打印的回溯信息，这也是出于兼容的目的。不过可以根据自己的喜好随意修改格式。

## 更多资料

8.6 节，*Library Reference* 和 *Python in a Nutshell* 中的标准库模块 traceback 的文档；*Language Reference* 和 *Python in a Nutshell* 中关于 \_\_debug\_\_ 标志和 assert 声明的章节。

## 8.6 从 traceback 中获得更多信息

感谢：Bryn Keller

### 任务

当一个未被捕获的异常发生时，你希望能够打印出所有变量的信息。

### 解决方案

一个 traceback 对象，基本上是一个互相关联的节点的列表，每个节点都指向一个帧对象。而帧对象则反过来根据 traceback 的关联节点列表以相反的顺序建立它们自己的链表，所以，我们可以根据需要向前或向后移动。本节方案利用了这个结构和帧对象持有的丰富信息，具体地说，我们利用了帧的对应函数的局部变量的字典：

```
import sys, traceback
def print_exc_plus( ):
    """ 打印通常的回溯信息，且附有每帧中的局部变量的列表 """
    tb = sys.exc_info()[2]
    while tb.tb_next:
        tb = tb.tb_next
    stack = [ ]
    f = tb.tb_frame
    while f:
```

```

        stack.append(f)
        f = f.f_back
    stack.reverse( )
    traceback.print_exc( )
    print "Locals by frame, innermost last"
    for frame in stack:
        print
        print "Frame %s in %s at line %s" % (frame.f_code.co_name,
                                              frame.f_code.co_filename,
                                              frame.f_lineno)
    for key, value in frame.f_locals.items( ):
        print "\t%20s = " % key,
        # 我们必须_绝对_避免异常的扩散，而 str(value)
        # _能够_引发任何异常，所以我们_必须_捕获所有异常
        try:
            print value
        except:
            print "<ERROR WHILE PRINTING VALUE>"

```

## 讨论

标准的 Python 模块 `traceback` 提供了关于错误发生的地点和原因的有用信息。但 `traceback` 对象包含了太多的信息（非直接的、由帧对象引用的信息），比它显示的信息还多。这些额外的信息能够有效地协助我们查明错误的原因。本节提供的是一个扩展的回溯信息的打印函数，可以使用它来获得所有的细节。

下面是一个简化的展示，用来展示此方法能够应对的问题类型。基本上，我们有了一个简单的函数来操纵列表中的字符串。该函数并未执行任何检查，所以，如果我们传递一个含有非字符串子项的列表，就会出错。我们的新的 `print_exc_plus` 函数能够很容易地找出引发错误的数据：

```

data = ["1", "2", 3, "4"]      # 手误：我们忘记了给 data[2] 打上引号
def pad4(seq):
    """

```

给每个字符串填充 0 直到总长度为 4。注意，完全没必要写这样一个函数；Python 已经提供了工具，而且做得更好。这仅仅是一个例子！

```

"""
return_value = []
for thing in seq:
    return_value.append("0" * (4 - len(thing)) + thing)
return return_value

```

下面是我们从 `traceback.print_exc` 获得的信息（很有限）：

```

>>> try:
...     pad4(data)
... except:

```

```
...     traceback.print_exc( )

...
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
  File "<stdin>", line 9, in pad4
TypeError: len() of unsized object
```

下面是用本节提供的函数打印出来的信息：

```
>>> try:
...     pad4(data)
... except:
...     print_exc_plus( )

...
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
  File "<stdin>", line 9, in pad4
TypeError: len() of unsized object
Locals by frame, innermost last
Frame ? in <stdin> at line 4
    sys = <module 'sys' (built-in)>
    pad4 = <function pad4 at 0x007C6210>
    __builtins__ = <module '__builtin__' (built-in)>
    __name__ = __main__
    data = ['1', '2', 3, '4']
    __doc__ = None
    print_exc_plus = <function print_exc_plus at 0x00802038>
Frame pad4 in <stdin> at line 9
    thing = 3
    return_value = ['0001', '0002']
    seq = ['1', '2', 3, '4']
```

应该能注意到，现在找出引发问题的坏数据是多么容易的一件事。`thing` 变量的值是 3，所以我们就能了解为什么会得到 `TypeError`。对 `data` 的值快速一扫，我们马上就能意识到忘记了给那个元素加上引号。现在，我们可以选择修改数据，或者提升 `pad4` 函数的容错性（比如，将循环改成 `for thing in map(str, seq)`）。这些设计的选择很重要，但本节的要旨是帮助你发掘和理解发生的一切，因此可以根据所有信息自行决定你的设计。

本节的方案依赖于这样一个事实，即每个 `traceback` 对象都通过 `tb_next` 字段引用栈中的下一个 `traceback` 对象，从而建立了一个链表。每个 `traceback` 对象同时也通过 `tb_frame` 字段指向一个对应的帧，每帧又通过 `f_back` 字段指向前一帧（与 `traceback` 对象的链表方向相反）。

为了简化，本节方案首先在一个叫做 `stack` 的本地列表中存放指向帧对象的所有引用，然后循环此列表，输出每一帧的信息。对于每一帧，它首先输出基本信息（比如函数名、文件名、行号等），然后转向 `f_locals` 字段指向的该帧的局部变量的字典。就像通过内建的 `locals` 和 `global` 函数创建并返回的字典一样，该字典的每一个键都是一个变量的名字，而对应值则是变量的值。注意，虽然打印名字是安全的（因为它只是字符串），

打印对应值则可能会失败，因为它可能会调用用户自定义对象的一个随意的、有问题的`__str__`方法。因此，在打印值时我们用`try/except`声明加以保护，以避免当我们在处理一个异常时另一个未捕获的异常又开始扩散的情况出现。如果一个`except`子句并未列出感兴趣的异常，那它就干脆捕获所有的异常，通常情况下这么用会是个错误，但本节却展示了它的一个适用场景。

我在我开发的一个应用程序中使用了类似的技术，将所有详细的信息写入到了一个日志文件，以备日后分析。在交互式状态下抛出这么多的信息则可能会让人无从下手。向可怜的用户抛出这么多调试信息，即使只是缩减版的回溯信息，也可以算是用户接口设计上的失误。而悄悄地把这些信息写入日志文件，对于开发者和用户都是可以接受的。日志文件中的信息就像是一片宝石矿：宝石就在里面，你只要有时间去筛查，就能找到宝石。

## 更多资料

8.5 节，*Library Reference* 和 *Python in a Nutshell* 中的 `traceback` 模块，以及 `sys` 模块中的 `exc_info` 函数的文档。

# 8.7 当未捕获异常发生时自动启用调试器

感谢：Thomas Heller、Christopher Prinos、Syver Enstad、Adam Hupp

## 任务

当脚本激发了异常时，Python 在通常情况下会打印回溯信息以及终止脚本运行，但你却希望当这种情况发生时，Python 能够自动启动一个交互式的调试器，如果条件允许的话。

## 解决方案

通过设置 `sys.excepthook`，可以控制当未捕获异常（uncaught exception）发生时 Python 的行为方式：

```
# 此代码片段需要被置入你的 sitecustomize.py
import sys
def info(type, value, tb):
    if hasattr(sys, 'ps1') or not (
        sys.stderr.isatty() and sys.stdin.isatty()
        ) or issubclass(type, SyntaxError):
        # 交互模式，没有类似 TTY 的设备或语法错误：什么也不做
        # 只能调用默认的 hook
        sys.__excepthook__(type, value, tb)
    else:
        import traceback, pdb
```

```
# 非交互模式；因此打印异常
traceback.print_exception(type, value, tb)
print
# 在事后分析阶段开启调试器
pdb.pm()
sys.excepthook = info
```

## 讨论

当一个 Python 脚本在运行期间发生了未捕获异常，而且异常一直向上传播，则回溯信息会被输出到标准错误输出，脚本的执行也会被终止。不过，Python 公开了 `sys.excepthook`，因此可以利用它来重定义对未捕获异常的应对方式。这样，当 Python 运行在一个非交互的模式下且类 TTY 设备可用的时候，你就可以在未捕获异常发生时自动启用调试器。语法错误没有什么需要调试的，所以本节代码对这类异常只是使用默认的处理机制。

本节的代码需要被放入到 `sitecustomize.py` 中，Python 在启动时会自动导入这个文件。当且仅当 Python 运行在非交互模式以及交互式调试器需要的类 TTY 设备可用时，函数 `info` 才会启动调试器。因此，对于 CGI 脚本、守护进程等，调试器不会被启动；为了解决这类问题，请参看 8.5 节。如果你没有 `sitecustomize.py` 文件，可以在你的 Python 库目录的 `site-packages` 子目录中创建一个。

本节的方案还可以继续扩展，检测 GUI IDE 是否正在使用中，如果用户使用了 IDE，程序还可以激活适用于此 IDE 的调试环境，而不是使用默认的 Python 的 `pdb`，后者只适用于文本交互模式。不过，这种检测和激活完全依赖于特定的 IDE。举个例子，为了启动 Windows 中 PythonWin IDE 的调试器，而不是默认地导入 `pdb` 并调用 `pdb.pm`，可以导入 `pywin` 并调用 `pywin.debugger.pm`——但我不知道怎样以一种通用和安全的方式完成这种检测和激活。

## 更多资料

8.5 节，*Library Reference* 和 *Python in a Nutshell* 中的 `sys` 模块的 `__excepthook__` 函数、`tracebook`、`sitecustomize` 以及 `pdb` 模块的文档。

## 8.8 简单的使用单元测试

感谢：Justin Shaw

### 任务

你发现对标准库模块 `unittest` 中的测试器的使用还未简化到极致，你想改进对单元测试的使用以确保那是一件非常简单清楚的事，从而让自己养成勤于测试的习惯。

## 解决方案

将下列代码存为模块 microtest.py，放入 Python 的 sys.path 中：

```
import types, sys, traceback
class TestException(Exception): pass
def test(modulename, verbose=None, log=sys.stdout):
    """
    执行命名模块中的所有名字中含__test__的,
    不接受参数的函数
    modulename: 需要被测试的模块的名字
    verbose: 如果为真, 执行时打印测试名
    成功时返回 None, 否则抛出异常
    """
    module = __import__(modulename)
    total_tested = 0
    total_failed = 0
    for name in dir(module):
        if '__test__' in name:
            obj = getattr(module, name)
            if (isinstance(obj, types.FunctionType) and
                not obj.func_code.co_argcount):
                if verbose:
                    print>>log, 'Testing %s' % name
                try:
                    total_tested += 1
                    obj()
                except Exception, e:
                    total_failed += 1
                    print>>sys.stderr, '%s.%s FAILED' % (modulename,
                                                        name)
                    traceback.print_exc()
    message = 'Module %s failed %s out of %s unittests.' % (
        modulename, total_failed, total_tested)
    if total_failed:
        raise TestException(message)
    if verbose:
        print>>log, message
def __test__():
    print 'in __test__'
import pretest
pretest.pretest('microtest', verbose=True)
```

## 讨论

Python 标准库中的 unittest 模块比起这个简单的 microtest 模块要复杂多了，我诚恳地建议你仔细研究 unittest 模块。不过，如果你需要一个简单的单元测试接口，microtest 可能会比较合你胃口。

关于 unittest 的一个特殊的地方是可以站在作者的肩膀上看待整个模块，可以通读 Kent Beck 的大作 *Test Driven Development By Example* (Addison-Wesley)：此书有一整章的内容介绍通过显示早期开发过程来实现由测试驱动的开发，这一章使用 Python 语言作为示例，而这部分工作后来就变成了 unittest。Beck 的书值得强烈推荐，而且我认为此书会点燃你对测试驱动的开发以及更一般性的单元测试的热情。

Beck 的开发哲学中有一条被称为极限编程：“在能正确工作的前提下做最简单的事。”对于我自身的需要，使用本节展示的 microtest 模块和 8.9 节展示的 pretest 模块混用，就可以算是“最简单的事”了——而且它也确实工作得很好，对于我的日常开发工作来说，完全是量身定做。

从某种角度说，本节也揭示了另一个重要的方面：Python 的内省能力是如此的简单易用，所以，根据自己的实际需要打造自己的单元测试框架，是一种可行的、合理的方式。无论你使用简单的 microtest，还是标准库的复杂的 unittest，或者任何其他你自己发明的框架，当需要编写并运行大量的单元测试时，它们都能提供实质性的帮助。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中关于标准库模块 unittest 的文档；Kent Beck 的 *Test Driven Development By Example* (Addison-Wesley)。

## 8.9 自动运行单元测试

感谢：Justin Shaw

### 任务

你想确保在每次编译模块时模块的单元测试会自动运行。

### 解决方案

执行测试的工作最好留给测试运行函数来做，如同 8.8 节介绍的 microtest.test。为了让其自动化，可以将下列代码存为模块文件 pretest.py，并放入你的 Python 的 sys.path 中（如果你使用的测试运行函数不是 microtest.test，应当修改模块导入声明以及默认的 runner=microtest.test）：

```
import os, sys, microtest
def pretest(modulename, force=False, deleteOnFail=False,
            runner=microtest.test, verbose=False, log=sys.stdout):
    module = __import__(modulename)
    # 仅测试未编译的模块，除非强制
    if force or module.__file__.endswith('.py'):
        if runner(modulename, verbose, log):
            pass                                # 通过所有测试
```

```
elif deleteOnFail:  
    # 移除 pyc 文件以便下次运行测试套件  
    filename = module.__file__  
    if filename.endswith('.py'):  
        filename = filename + 'c'  
    try: os.remove(filename)  
    except OSError: pass
```

现在，可以将下列代码放入你的各个模块中：

```
import pretest  
if __name__ != '__main__':    # 当模块导入时，不作为主脚本运行  
    pretest.pretest(__name__)
```

## 讨论

当你需要重复地修改一些模块时，你会觉得，如果那些代码能够在每次改动发生时“自我测试”（意味着它会自动运行单元测试）该是多么美好的事（每次模块发生改变和每次模块被重新编译是同一个概念。Python 会照料一切，它会在导入模块时自动编译之。在上次导入模块之后，只要模块被修改了，自动编译就会发生）。通过让单元测试成为一种自动化的过程，你就可以从反复运行单元测试的重复性劳动中解放出来。注意，解决方案中的代码会在模块被导入的时候运行测试，而不是在它作为主脚本的时候。虽然 `if __name__ != '__main__'` 和平时所用的版本只有丝毫差异，但是意思却完全相反。

注意，不要把你的模块源代码（除非同时放入一个更新的编译过的字节码文件）放入一个通常情况下你没有权限写入的目录。在任何情况下这都是糟糕的做法，因为 Python 无法保存编译过的.pyc 文件，因此每次导入模块时都被迫重新编译，这拖慢了导入模块的应用程序的速度。除了这个完全可以避免的反复编译的问题之外，你还得忍受每次导入模块时自动发生的单元测试，它也会拖慢整体性能。类似的例子还有，不要将你的模块源码放入一个 zip 文件并让 Python 直接从 zip 文件中导入模块，如果你不能同时提供一个更新的编译过的字节码文件，就不要这么做；否则，你会苦于无止境的反复编译（而且，如果采用了本节的测试方案，还需要同时忍受反复测试）。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中标准库模块 `unittest` 的文档。

## 8.10 在 Python 2.4 中使用 doctest 和 unittest

感谢：John Nielsen

### 任务

你想通过 `doctest` 的简单而直观的方法来编写一些单元测试。然而，你不想让那些

单元测试的“用例”把你代码中的 docstrings 给搞乱，同时还要保留 unittest 强大的能力。

## 解决方案

假设你有个典型的 doctest 的应用，比如像下面这个玩具般的例子，toy.py：

```
def add(a, b):
    """ 将任意两个对象相加并返回和
    >>> add(1, 2)
    3
    >>> add([1], [2])
    [1, 2]
    >>> add([1], 2)
    Traceback (most recent call last):
    TypeError: can only concatenate list (not "int") to list
    """
    return a + b
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

在你的函数的 docstrings 中有一些示例用法，通过 doctest 来测试它们的准确性是个不错的主意。不过，你不希望那些容易编写的单元测试用例弄乱了你的 docstrings。在 Python 2.4 中，可以将那些 doctest 的单元测试放入一个单独的文件，并为它们创建一个“测试套件”，然后用 unittest 运行它们。比如，在 test\_toy.txt 中放入下列行（不需要引号）：

```
>>> import toy
>>> toy.add('a', 'b')
'ab'
>>> toy.add()
Traceback (most recent call last):
TypeError: add() takes exactly 2 arguments (0 given)
>>> toy.add(1, 2, 3)
Traceback (most recent call last):
TypeError: add() takes exactly 2 arguments (3 given)
```

然后在 toy.py 的末尾加入几行：

```
import unittest
suite = doctest.DocFileSuite('test_toy.txt')
unittest.TextTestRunner().run(suite)
```

现在，在命令提示符下运行 python toy.py，会得到如下输出：

```
.
-----
Ran 1 test in 0.003s
OK
```

## 讨论

Python 标准库模块 `doctest` 为你的代码提供了一种简单高效的方式来创建大量的单元测试。你需要做的只是在 Python 交互式会话中导入并使用你的模块。然后将会话的输出拷贝并粘贴到 `docstring`, 或许需要少许编辑（比如，将每个异常回溯信息的除第一行之外的内容删除，只留下第一行，始于“`traceback`”，以及最后一行，始于“`TypeError:`”等类型的异常名）。

### Docstrings

文档字符串（`docstrings`）是 Python 提供的一个帮助程序员实现代码文档化的重要特性。任何模块、类、函数或方法都可以拥有一个描述性的字符串作为它的“声明”。Python 会将那个字符串作为这个模块、类、函数或方法的 `docstring` 并存入该对象的 `__doc__` 属性中。没有 `docstrings` 的模块、类、函数或方法的 `__doc__` 属性的值是 `None`。

在 Python 的交互式解释器中，可以通过命令 `help(theobject)` 检视一个对象的 `docstring` 和其他的关于此对象的有用的信息。Python 标准库中的模块 `pydoc`，会利用 `docstring` 和内省机制来生成关于模块、类、函数或方法的 web 页面，还可以使用此模块来提供 web 信息服务。（见 <http://pydoc.org>，它包含了由 `pydoc` 生成的 Python 标准库文档和标准 Python 在线文档。）

Python 标准库的 `unittest` 模块非常强大，所以可以创建更多的高级单元测试并使用更复杂的方式来运行它们。但比起 `doctest`，用它编写单元测试并没有那么简单和快速。

归功于 `doctest` 的简洁，很多 Python 程序员在各处使用这个模块，但是，除了没有 `unittest` 的结构化的运行单元测试的能力，它还把大量的“用例”和 `docstrings` 混在一起，而且这些“用例”还可能会干扰读者们的眼睛。这些“用例”存在的目的，是为了对各种边界条件、极限条件、困难条件提供深入的单元测试。

换个角度说：`doctest` 是个强大的工具，它能保证你放入 `docstrings` 的例子是正确有效的，这会鼓励你一开始就在 `docstrings` 中放入例子——这是好事。但是 `doctest` 的目标是快速地制造各种简单的单元测试——不过这些单元测试不应该被放入到 `docstrings` 中，因为如果将测试放入 `docstrings`，它们就会弄乱文档，对于读者来说，其结果不是加强反而是削弱。

Python 2.4 版本的 `doctest` 允许你做“做不到的事”，而且提供了 `doctest` 的简洁高效和 `unittest` 的强大（不再会弄乱 `docstrings`）。具体地说，那个“做不到的事”现在可以通过新函数 `doctest.DocFileSuite` 来做到了。传递给这个函数的参数是一个文本文件的路径，此文件中包含的是类似于 `doctest` 格式的一串文本行（如使用`>>>`提示符的 Python 声明，以及紧随每个声明后的预期的结果或错误信息）。这个函数返回一个“测试套件”对象，

此对象兼容于 unittest 能够创建或接受的套件对象。比如，如解决方案中所示，可以给一个 TextTestRunner 实例的 run 方法传递一个套件对象来作为参数。注意，你传递给 doctest.DocFileSuite 的文本文件，其中的提示、声明以及结果并没有像 docstring 那样用三引号括住。本质上，那个文本文件可以直接从 Python 交互式解释器会话中复制内容（需要一点编辑，比如对异常回溯的修改，前面已经提到过了）。

## 更多资料

*Language Reference* 和 *Python in a Nutshell* 中的标准库模块 unittest 和 doctest 的文档。

## 8.11 在单元测试中检查区间

感谢：Javier Burroni

### 任务

你发现你的单元测试总需要检查结果值，不是为了验证它等于或不等于某个指定的值，而是为了确定它在或不在某个特定的区间。你很希望能够用 unittest 像检查值相等或不相等那样来检查值是否在区间中。

### 解决方案

最好的方法是从 unittest.TestCase 派生子类并加入一些额外的检查方法：

```
import unittest
class IntervalTestCase(unittest.TestCase):
    def failUnlessInside(self, first, second, error, msg=None):
        """ 如果 first 不在区间中，失败
            区间由 second+error 构成
        """
        if not (second-error) < first < (second+error):
            raise self.failureException, (
                msg or '%r != %r (+-%r)' % (first, second, error))
    def failIfInside(self, first, second, error, msg=None):
        """ 如果 first 在区间中，失败
            区间由 second+error 构成
        """
        if (second-error) < first < (second+error):
            raise self.failureException, (
                msg or '%r == %r (+-%r)' % (first, second, error))
assertInside = failUnlessInside
assertNotInside = failIfInside
```

### 讨论

下面是 IntervalTestCase 类的一个用例，if \_\_name\_\_ == '\_\_main\_\_' 的保护使得我们可

以将它和类定义放入到同一个模块中，当模块作为主脚本运行时，这段代码会被执行：

```
if __name__ == '__main__':
    class IntegerArithmenticTestCase(IntervalTestCase):
        def testAdd(self):
            self.assertInside((1 + 2), 3.3, 0.5)
            self.assertInside(0 + 1, 1.1, 0.01)
        def testMultiply(self):
            self.assertNotInside((0 * 10), .1, .05)
            self.assertNotInside((5 * 8), 40.1, .2)
    unittest.main()
```

当你正在开发的组件涉及到大量的浮点计算时，你不太可能去测试结果是否精确地等于某个参考值。你一般会围绕参考值指定一个正确范围，提供一定程度的容错性。因此，`unittest.TestCase.assertEquals` 和它的类似方法就不再适用了，结果你只好使用一般性的方法，如 `unittest.TestCase.failUnless` 和类似的方法来完成检查，而且这个过程还会产生大量的像 `x-toler < result < x+toler` 一样的比较表达式。

本节的 `IntervalTestCase` 类提供的方法，如 `assertInside`，能够帮助你以优雅的方式进行一种“约等于”检查，就像 `unittest` 本身提供的精确值检查方式那样。如果你正在实现这种约等于函数，或者像我一样正在研究数值分析，你会发现本节提供的内容非常有用。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中的标准库模块 `unittest` 的文档。

## 第9章

# 进程、线程和同步

### 引言

感谢：Greg Wilson、Third Bit

30 年前，Fred Brooks 在他的经典著作 *The Mythical Man-Month: Essays on Software Engineering* (Addison-Wesley) 中对偶发复杂度和原生复杂度进行了区分。语言如英语和 C++ 之类，都有它们自身的不一致的规则、例外以及特例，而这正好可以作为偶发复杂度的例证：它们使得沟通和编程变得比实际需要的还难。而另一方面，并发性则可以作为原生复杂度的一个主要的例子。大多数人都必须经过努力才能在脑中保持对事件链条的跟踪；追踪两个、三个或十几个事件，还要加上它们之间可能的交互，实在是痛苦极了。

计算机科学家们在上世纪 60 年代中期开始研究让多个进程在单个物理地址空间中安全和有效率地运行的方法。从那时起，关于进程交互的行为理论开始发展，这方面的论文专著也不断涌现，专用于并发以及并行编程的语言也被创建出来。*Foundations of Multithreaded, Parallel, and Distributed Programming* (Addison-Wesley)，Gregory R. Andrews 著，不仅是对这种理论的杰出介绍，也包含了理论主要思想发展轨迹的历史信息。

在过去 20 年中，由于好的时机和并发本身的必要性，并发已经成为了程序员日常生活的一部分。所谓好的时机是指不断发展的多处理器计算机带来的更快的速度。在 1980 年代早期，它们还是让人啧啧称奇的新玩意儿；现在，很多程序员已经拥有了双处理器桌面工作站，在里屋还有四路或八路的服务器。如果计算能够被分解为独立（或近乎独立）的片段，则这些计算机可能比单处理器计算机快 2 倍、4 倍或 8 倍。不过这种方式的潜在增益受到了一些限制，它只适用于图像处理、应答 HTTP 请求以及并行编译等。

并发编程的必要性则来自于 GUI 和网络程序。图形界面一般总是显得能同时做很多事情。

情，比如在显示图片的时候还可以在屏幕底部滚动广告。虽然用手工的方式进行分工也是可能的，但是最简单的方式还是编程完成自身的操作，让底层的操作系统来安排操作的顺序。类似的，网络应用程序也总是需要同时监听多个 socket，或者在某个频道正在接收数据的时候，另一个频道还要发送数据。

一般来说，操作系统给了程序员两种并发支持。进程运行在分离的被保护的逻辑地址空间中。出于提高性能的目的，并发进程，尤其是在多处理器计算机中，看上去要比线程有吸引力，线程只是在同一个程序中、同一个地址空间中并行地执行。由于没有相互之间的保护并且共享地址空间，线程的开销更小，线程间的互相通信也更简单和快速。因此，为提高性能而采用的并发方式大多是给程序增加多线程支持。

不过，在 Python 程序中通过增加线程来提速往往不是一个好的策略。原因就在于全局解释器锁（Global Interpreter Lock，GIL），它被用来保护 Python 的内部数据结构。在一个线程能够安全地访问 Python 对象之前，它必须持有这个锁。如果没有这个锁，即使是最简单的操作（比如整数加法）都会失败。因此，只有带有 GIL 的线程能够操纵 Python 对象或者调用 Python/C API 函数。

为了便于程序员使用，解释器会每 100（这个值可以通过 `sys.setcheckinterval` 修改）个字节指令（bytecode instruction）释放并获取锁一次。在进行 I/O 操作时，如读写文件，锁也会被释放和获取，这样，需要进行 I/O 操作的线程在等待 I/O 操作结束时，其他线程也会获得机会运行。不过，对于多处理器计算机，在同一个进程中多个纯 Python 线程的程序无法利用多处理器来提升性能。如果你把你的多线程程序放到多处理器计算机上，你不会看到任何实质性的性能提升，除非你的 Python 程序的 CPU 性能瓶颈不在别处，而是在释放了 GIL 的用 C 编写的扩展中。

不过，线程对于多处理器计算机并不仅仅意味着性能。GUI 不知道用户会按下一个键还是移动鼠标，HTTP 服务器也不知道下一个到来的数据报是什么。在吞吐量不是我们主要关心的问题的情况下，用单独的控制线程来处理事件流通常是应对这种不可预测性的最简单的方法，即使是在单处理器计算机上。在这种类型的应用程序中，程序员常常会应用事件驱动编程的方式，而一些 Python 框架，如 `asyncore` 和 `Twisted`，也证明了这种方式一般总是能够提供优良的性能。当然，还有随之而来的复杂度，它的复杂度不同于多线程本身固有的复杂度，但不一定更难对付。

Python 标准库允许程序员在两个不同的层面使用多线程技术。核心模块 `thread`，是对基本原语的一个薄薄的封装，而这些基本的原语，是任何线程库都必须提供的东西。这些原语中有三个被用于创建、识别和结束线程；其他的则被用于创建、测试、获取以及释放简单的互斥锁（或二进制信号量）。开发者们应当避免直接使用这些原语，而使用包含在高阶模块 `threading` 中的各种工具，这些工具拥有更友好的界面，而且在性能上也并不逊色。

无论你使用 `thread` 或 `threading`，关于 Python 线程模型的一些基本的东西不会变化。尤

其是 GIL，它的工作方式和原理始终如一。GIL 的最大的优势是它使得用 C 编写 Python 扩展成为一件容易的事情：可以很确定，在你的 C 代码回调 Python 代码之前，线程切换都不会发生，除非你的 C 扩展明确地释放了 GIL。这个优点极其重要，特别对一些底层的 C 库，它们其实不是线程安全的，而你却可以放心地用它们编写扩展。再次提醒一下，在你调用任何 Python C API 入口点之前，一定要先获得 GIL。

任何时候当你的代码要访问被线程共享的数据结构时，你都很关心这个操作是否是原子操作（即操作过程中不会发生线程切换）。一般来说，和多个字节码相关的一定不是原子的，因为在当前字节码和下一个（可以使用标准库函数 `dis.dis` 来将 Python 代码分解成字节码）之间，线程切换是可能的。甚至一个单独的字节码也不是原子的，如果它还能回调任何其他 Python 代码的话（比如，字节码可能以执行某个 Python 编写的方法为结束）。当你有所怀疑时，明智的做法是，假设任何引起你怀疑的东西都不是原子操作，应当尽量把被线程共享的数据结构缩减到最小（除了 `Queue.Queue` 的实例，这个类被设计为线程安全的），并确保你用锁保护了任何对数据结构的访问。

使用锁的形式几乎是不变的，大概是这样：

```
somelock.acquire( )
try:
    # 需要锁的操作（尽量简短）
finally:
    somelock.release()
```

`try/finally` 结构确保了即使在 `try` 字句中发生某些异常，锁也一定会被释放。如果不使用 `try/finally` 结构，由于某些未预见到的异常，锁有时可能无法释放，那种情况可以很快地造成你的程序骤然挂起。另外，你还应当注意获取锁的顺序；如果你确实需要获取多个锁，一定要确保在代码的各处以相同的顺序获取锁。否则，迟早你会碰到一种灾难性的后果，即，两个线程都尝试获取对方持有的锁——这也被称为死锁，在这种情况下你唯一能做的就是给你的程序收尸了。

`threading` 模块中最重要的部分是代表着线程和各种高阶同步结构的类。而 `thread` 类代表一个单独的控制线程，可以传递一个可调用的对象给它的构造函数或者重写它的 `run` 方法，从而定制它的行为。一个线程可以调用另一个线程的 `start` 方法，从而启动该线程，还可以调用 `join` 来等待它的结束。Python 也支持守护线程，当程序中所有的非守护线程退出并自动关闭自身后，它可以进行一些后台处理。

`threading` 模块中的同步结构包括了锁、可重入锁（`reentrant lock`，一个单独线程可以安全地锁定很多次而不会造成死锁）、计数信号量、条件以及事件。线程用事件来作为信号，通知其他线程某件有趣的事情发生了（比如队列中添加了新元素，或者某个共享数据结构可以被修改了）。Python 所带的文档，也就是 *Library Reference* 手册，对这些类进行了详尽的描述。

本章和书中的其他章相比，篇幅较短，这充分说明 Python 很关注程序员的生产率（而

不是绝对性能)，同时各种包（如 `httplib` 和 `wxPython`）也非常好地掩盖了那些令人不快的并发编程的细节。本章相对较短的篇幅还反映了另一个事实，即 Python 程序员总是试图用最简单的方法解决特定问题，而复杂的多线程编程没有多少机会成为最简单的方法。

不过，本章的简短篇幅可能也说明了 Python 社区对于这个简单的 `threading` 还不太欣赏，虽然正确地使用此模块能够大大简化工作量。`Queue` 模块提供了一种赏心悦目的同步和协作结构（而且还能扩展和定制），它能够提供所有需要的线程间的管理和控制。考虑一个典型的应用，一个从 GUI（或网络）接收请求的程序。作为请求的“结果”，这个程序发现自己总是必须执行大块的工作。而大块的工作需要的时间太多，除非预先采取了一些措施，不然程序就像对 GUI（或网络）失去了响应一样。

在一个纯粹的事件驱动的架构中，为了避免这种不响应状态，开发者可以尝试将大块的工作切分成细碎的能够片刻完成的工作，但这种方式往往吃力不讨好。对于这种情况，使用一点多线程技术会有极大的好处。主线程将描述任务的工作请求放入一个专用的 `Queue` 实例中，然后回到它本来的任务——维持程序界面的可操作性。

而在 `Queue` 的另外一端，一群守护工作线程等待机会从 `Queue` 中得到工作请求，如果获得了请求就直接处理之。这种总体架构综合了事件驱动和多线程的方式，体现了一种简洁的思想以及 Python 的独有风格。如果工作线程的处理结果还需要呈现给主线程，那你还得多做一点工作（当然，通过另一个 `Queue` 来实现），这部分工作和 GUI 相关性较高。如果你希望省点事，可以让事件驱动的主线程以轮询的方式去访问守护工作线程返回的结果的 `Queue`。参见 11.9 节，你会发现那是多么容易的一件事。

## 9.1 同步对象中的所有方法

感谢：André Bjärb、Alex Martelli、Radovan Chytracek

### 任务

你希望在多个线程中共享一个对象，但为了避免冲突，需要确保任何时间只有一个线程在对象中——可能还要除去一些你想手工调整锁定行为的方法。

### 解决方案

Java 提供了这种同步作为内建特性，但在 Python 中必须通过编程将对象和它的方法封装起来。这个封装非常有用，而且适用面也广，所以有必要提取出来做成通用工具：

```
def wrap_callable(any_callable, before, after):
    '''用 before/after 调用将任何可调用体封装起来'''
    def _wrapped(*a, **kw):
        before()
        try:
```

```

        return any_callable(*a, **kw)
    finally:
        after( )
# 只针对 Python 2.4: _wrapped.__name__ = any_callable.__name__
return _wrapped
import inspect
class GenericWrapper(object):
    ''' 将对象的所有方法用 before/after 调用封装起来 '''
    def __init__(self, obj, before, after, ignore=()):
        # 我们必须直接设置__dict__来绕过__setattr__; 因此,
        # 我们需要重写带下划线的名字
        # we need to reproduce the name-mangling for double-underscores
        classname = 'GenericWrapper'
        self.__dict__['__%s__methods' % classname] = {}
        self.__dict__['__%s__obj' % classname] = obj
        for name, method in inspect.getmembers(obj, inspect.ismethod):
            if name not in ignore and method not in ignore:
                self.__methods[name] = wrap_callable(method, before,
                                                       after)

    def __getattr__(self, name):
        try:
            return self.__methods[name]
        except KeyError:
            return getattr(self.__obj, name)
    def __setattr__(self, name, value):
        setattr(self.__obj, name, value)

```

通过使用这些简单而通用的工具，同步变得容易了：

```

class SynchronizedObject(GenericWrapper):
    ''' 封装一个对象及其所有方法，支持同步 '''
    def __init__(self, obj, ignore=(), lock=None):
        if lock is None:
            import threading
            lock = threading.RLock()
        GenericWrapper.__init__(self, obj, lock.acquire, lock.release,
                               ignore)

```

## 讨论

对于每个 Python 实践，我们都可以增加一个自我测试部分来完成整个模块，测试只有在模块被当做主脚本执行时才会运行。下面的代码片段用来展示怎样使用此模块的功能：

```

if __name__ == '__main__':
    import threading
    import time
    class Dummy(object):
        def foo(self):

```

```

        print 'hello from foo'
        time.sleep(1)
    def bar(self):
        print 'hello from bar'
    def baaz(self):
        print 'hello from baaz'
tw = SynchronizedObject(Dummy( ), ignore=['baaz'])
threading.Thread(target=tw.foo).start( )
time.sleep(0.1)
threading.Thread(target=tw.bar).start( )
time.sleep(0.1)
threading.Thread(target=tw.baaz).start( )

```

归功于同步机制，对 `bar` 的调用只有在对 `foo` 的调用完成之后才会发生。不过，由于 `ignore = keyword` 参数的指定，对 `baaz` 的调用绕过了同步并因此提前完成了。下面是输出：

```

hello from foo
hello from baaz
hello from bar

```

如果你发现需要对一个对象的几乎所有方法使用同样的单锁（single-lock）锁定代码，通过本节的方法，你就可以将加锁操作从对象的特定逻辑中隔离开来。使用本节方法的效果和用下列代码替换对象的每个方法获得效果一样：

```

self.lock.acquire( )
try:
    # 该方法的“真实”代码
finally:
    self.lock.release( )

```

这个代码的形态完全是加锁的步骤：`try/finally` 声明确保无论发生什么锁都能够被释放，无论应用代码正确地结束还是有异常发生。还可以注意到工厂 `wrap_callable` 返回的是一个闭包，其代码形态也是如此。

从某种角度讲，当你想要推迟考虑类的加锁时，本节的方案会非常方便有用。然而，如果你想将这些代码用于产品开发，必须全盘理解它。尤其是，本节的方案并没有封装对对象属性的直接访问（获得和设置）。如果你希望能够封装这种访问，需要在封装层的 `__getattr__` 和 `__setattr__` 特殊方法中使用 `try/finally` 加锁形式，将这些方法中对内建函数的 `getattr` 和 `setattr` 的调用分别包裹起来。不过对于我的应用来说，我不认为封装到这种程度是必要的（我的方式——只封装方法，已经被证明够用了）。

如果你很喜欢用自定义元类，你可能会奇怪为什么我没有提供一个元类来实现同步呢。其实，封装是一种更具动态性、更灵活的方式，比如，一个对象可以以两种状态存在，封装过的（同步的）和未封装的（原始对象），可以根据你的需要使用合适的一个。虽然你为这种封装灵活度付出了一点代价——每次调用方法，都会有一点小小的运行时

开销。但是比起获取和释放锁的巨大开销，这点开销完全可以忽略不计。本节展示的封装-闭包的工厂以及封装类都具有很好的可重用性，这充分显示了 Python 的能力，它轻松地实现了一个面向方向编程爱好者们喜爱的设计模式，为对象的各个方法的调用插入了一个“before-and-after”调用。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中的标准库模块 `threading` 以及 `inspect` 的文档。

## 9.2 终止线程

感谢： Doug Fort

### 任务

需要从外部结束一个线程，可是 Python 不允许一个线程强行杀死另一个线程，所以你得采用一种适当的受控终止的方式。

### 解决方案

一个常常被问到的问题是：我怎么杀死一个线程？答案是：你不能。事实上，你只能请求它结束。每个线程必须周期性地检查是否被要求结束，然后服从命令（一般还要完成一些清理收尾工作）。下面是个例子：

```
import threading
class TestThread(threading.Thread):
    def __init__(self, name='TestThread'):
        """ 构造函数，设置初始值 """
        self._stopevent = threading.Event()
        self._sleepperiod = 1.0
        threading.Thread.__init__(self, name=name)
    def run(self):
        """ 主控循环 """
        print "%s starts" % (self.getName(),)
        count = 0
        while not self._stopevent.isSet():
            count += 1
            print "loop %d" % (count,)
            self._stopevent.wait(self._sleepperiod)
        print "%s ends" % (self.getName(),)
    def join(self, timeout=None):
        """ 停止线程并等待其结束 """
        self._stopevent.set()
        threading.Thread.join(self, timeout)
if __name__ == "__main__":
    testthread = TestThread()
```

```
testthread.start()
import time
time.sleep(5.0)
testthread.join()
```

## 讨论

你常常想在外部对线程施加一些控制，但杀死线程的能力却有点过头了。Python 没有给你那个权力，它以此来迫使你更谨慎地设计你的线程系统。本节的基本想法是在线程的主函数中使用一个循环。这个循环周期性地检查 `threading.Event` 对象的状态。如果该对象被设置，线程就会结束；否则，它会一直等待。

本节的 `TestThread` 类还重写了 `threading.Thread` 的 `join` 方法。通常，`join` 只是用来等待某个线程的结束（或者等待一段指定的时间，如果设置了等待时间的话），而与具体的终止线程的动作无关。在本节中，`join` 被重写并被用来设置结束事件对象，之后才托管给正常的（基类的）`join` 方法。因此，本节的 `join` 能很快地终止目标线程。

还可以在其他情况下应用本节的核心思想（周期性检查 `threading.Event` 对象来决定是否要终止）。`Event` 的 `wait` 方法能够暂停目标线程。可以暴露 `Event`，让控制代码设置它，然后继续执行任务，从而不用去 `join` 那个线程，因为那个线程会在短时间内意识到自己应该要退出了。一旦事件被公开，你可能希望同一个事件能够控制多个线程的终止——比如，如果某个线程池中的所有线程共享的事件对象被设置了，则所有线程终止。这个简单的方法已经提供了我需要的控制力，也没什么大毛病，所以我也就不再去探寻更复杂的方式了。

Python 还允许你以另一种方式终止线程：在线程中抛出一个异常。这种“粗鲁”的方式有一些局限性：它不能中断对操作系统的一个阻塞式调用，而且，如果你想终止的线程中有一个 `try` 子句，且 `except` 子句的捕获范围很广泛，它也可能失效。虽然有这些限制，这个方法在某些时候仍然很有用，特别是在你写调试器的时候。当你不能指望目标线程中的执行代码没有问题时候，你仍可以希望它的代码没有差到离谱的地步。一个常见的方式是生成一个独立的线程来监视一切，然后在主线程中使用那些可能有问题的代码的功能。当监视线程认为应该终止主线程中运行的代码时，它可以调用 `thread.interrupt_main`，并将需要的异常类作为参数传递给该方法。

在少数情况下，你编写的调试器无法在进程的主线程中运行那些可能有问题的代码，因为该线程被你依赖的某些框架用作其他用途，比如你的 GUI 代码。为了支持这种远程的偶发性的不测事件，Python 解释器有一个函数能够在任何线程中抛出异常，并给出目标线程的 ID。不过，这种特殊功能仅仅是为极少数的 Python 应用程序设计的，如调试器。为了避免诱惑其他 Python 程序员（超过 99.99%）误用这种终止线程的方式，这个函数在 Python 代码中是不可以直接调用的，而只在 Python 的 C API 中公开。这个函数的名字是 `PyThreadState_SetAsyncExc`，函数的两个参数分别是线程 ID 和期望的异

常类。如果你正在编写 Python 调试器而且你也需要这种功能，那么你很有可能还有一个向高阶 Python 代码提供各种特殊底层功能的用 C 编写的 Python 扩展模块，此扩展模块也是你的代码的一部分。你只需要在 C 代码中加入一个可调用 Python 的函数，让它调用 `PyThreadState_SetAsyncExc`，你的调试器就能够获得这种特殊的功能。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中的标准库模块 `threading` 的文档。

## 9.3 将 Queue.Queue 用作优先级队列

感谢：Simo Salminen、Lee Harr、Mark Moraes、Chris Perkins、Greg Klanderman

### 任务

你很想使用 `Queue.Queue` 实例，因为它是最好的线程间通信的方式。不过，需要一点附加的功能，即给队列中的每一个子项都指定一个优先值。这样，拥有更低优先值（更急迫）的子项就会先于更高的优先值的子项被获取。

### 解决方案

`Queue.Queue` 有很多优点，它提供的一种优雅的架构使得对它的队列行为的定制变得极其容易。`Queue.Queue` 特意暴露了一些方法以便用户在子类中重写，而且在获得了定制的队列行为之后我们仍然不需要担心同步问题。

我们利用这种良好的架构以及 Python 标准库的 `heapq` 模块就可以创建出满足需要的优先队列的功能。但我们还需要接管和封装 `Queue.Queue` 的 `put` 和 `get` 方法，在用 `put` 获得子项时给子项加上优先值和获取时间，在用 `get` 剥离子项时则需要删除子项的所有附加信息：

```
import Queue, heapq, time
class PriorityQueue(Queue.Queue):
    # 初始化队列
    def __init__(self, maxsize):
        self.maxsize = maxsize
        self.queue = []
    # 返回队列子项的数目
    def __qsize__(self):
        return len(self.queue)
    # 检查队列是否为空
    def __empty__(self):
        return not self.queue
    # 检查队列是否已满
    def __full__(self):
        return self.maxsize > 0 and len(self.queue) >= self.maxsize
    # 将子项放入队列
    def put(self, item, priority, timestamp):
        if self.__full__():
            raise Queue.Full
        entry = (priority, timestamp, item)
        heapq.heappush(self.queue, entry)
    # 从队列中取出子项
    def get(self):
        if self.__empty__():
            raise Queue.Empty
        entry = heapq.heappop(self.queue)
        item = entry[2]
        del entry
        return item
```

```

# 给队列加入一个新项
def _put(self, item):
    heapq.heappush(self.queue, item)
# 从队列中获取一项
def _get(self):
    return heapq.heappop(self.queue)
# shadow and wrap Queue.Queue's own 'put' to allow a 'priority' argument
# 屏蔽并封装 Queue.Queue 的 put，使之允许 priority 参数
def put(self, item, priority=0, block=True, timeout=None):
    decorated_item = priority, time.time(), item
    Queue.Queue.put(self, decorated_item, block, timeout)
#屏蔽并封装 Queue.Queue 的 get，以去除修饰
def get(self, block=True, timeout=None):
    priority, time_posted, item = Queue.Queue.get(self, block, timeout)
    return item

```

## 讨论

使用本节提供的 PriorityQueue 类的实例 q，可以调用 q.put(anitem)方法，在队列中加入带有“普通”优先值（这里定义为 0）的新项，或者调用 q.put(anitem, prio)以指定的优先值 prio 来加入新项。当 q.get()被调用时（可能在另一个线程中），优先值最小的项会被首先返回，高优先值的项会被继续保留。负优先值比“正常”值小，因此适合表现“紧急”的项；正优先值比“正常”值大，表明这些项可能需要等很久，因为连“普通”优先级的项都会先于它们被处理。当然，如果你对这种优先级概念感到不习惯的话，你完全可以根据自己的意愿来更改代码：比如，在 put 开始创建 decorated\_item 的时候，使用优先值的负值。如果你这么做，正优先值的项就变成了紧急项，而负优先值的项则需要等待很长时间了。

Queue.Queue 的架构值得学习、尊敬和效仿。不仅仅是因为 Queue.Queue 是最好的线程间通信的方式，而且它的设计也使得它很容易被子类化，我们可以很容易地用特殊的队列行为来替代它默认的 FIFO（先进先出）行为，如本节介绍的基于优先级的队列。具体地说，Queue.Queue 使用了漂亮的 Template Method Design Pattern（模板方法设计模式，[http://www.aleax.it/Python/os03\\_template\\_dp.pdf](http://www.aleax.it/Python/os03_template_dp.pdf)）。这种模式使得 Queue.Queue 把注意力集中到和同步加锁相关的问题上，但却把队列行为方式交给了方法\_put、\_get 等，它们可以被子类重写；这些钩子方法会在不用考虑同步的环境中被调用。

我们还需要重写 Queue.Queue 的 put 和 get 方法，因为我们需要给 put 的签名增加一个可选的 priority 参数，我们把新项放入队列之前先将它修饰一番（heapq 模块的本身的机制就能创造出我们想要的顺序——低优先值先被处理，而对于优先值相等的项，采用 FIFO 方式），然后从队列中取出项时先去除修饰，之后再返回裸项。所有这些附加的东西都是局部变量，因此它们和同步问题毫不相干。每个线程都有自己的栈；因此，只使用局部变量的代码（因此它们也无法修改能被其他线程读取的任何状态，或者读

取可被其他线程修改的状态)一定是线程安全的。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中的 Python 标准库模块 Queue 和 heapq；模板方法设计模式的展示，见 [http://www.strakt.com/docs/os03\\_template\\_dp.pdf](http://www.strakt.com/docs/os03_template_dp.pdf)；19.14 节和 5.7 节，展示了其他有关编写和使用优先级队列的例子。

## 9.4 使用线程池

感谢：John Nielsen、Justin A

### 任务

需要你的主线程把处理任务托付给一个工作线程池。

### 解决方案

`Queue.Queue` 是用来协调工作线程池的最简单和最有效率的方法。我们可以把所有需要的数据结构和函数放入一个类中，但具体采用什么方式并不是强制的。所以，你看，在下面代码中它们全部是全局的：

```
import threading, Queue, time, sys
# 全局变量(大写字母开头)
Qin = Queue.Queue()
Qout = Queue.Queue()
Qerr = Queue.Queue()
Pool = []
def report_error():
    '''通过将错误信息放入Qerr来报告错误'''
    Qerr.put(sys.exc_info()[:2])
def get_all_from_queue(Q):
    '''
        可以获取队列Q中所有项，无须等待
    '''
    try:
        while True:
            yield Q.get_nowait()
    except Queue.Empty:
        raise StopIteration
def do_work_from_queue():
    '''工作线程的“获得一点工作”，“做一点工作的主循环”'''
    while True:
        command, item = Qin.get()          # 这里可能会停止并等待
        if command == 'stop':
            break
        try:
```

```

# 模拟工作线程的工作
if command == 'process':
    result = 'new' + item
else:
    raise ValueError, 'Unknown command %r' % command
except:
    # 无条件 except 是对的，因为我们要报告所有错误
    report_error( )
else:
    Qout.put(result)
def make_and_start_thread_pool(number_of_threads_in_pool=5, daemons=True):
    ''' 创建一个N线程的池子，使所有线程成为守护线程，启动所有线程 '''
    for i in range(number_of_threads_in_pool):
        new_thread = threading.Thread(target=do_work_from_queue)
        new_thread.setDaemon(daemons)
        Pool.append(new_thread)
        new_thread.start( )
def request_work(data, command='process'):
    ''' 工作请求在Qin中是形如(command, data)的数对 '''
    Qin.put((command, data))
def get_result( ):
    return Qout.get( )      # 这里可能会停止并等待
def show_all_results( ):
    for result in get_all_from_queue(Qout):
        print 'Result:', result
def show_all_errors( ):
    for etyp, err in get_all_from_queue(Qerr):
        print 'Error:', etyp, err
def stop_and_free_thread_pool( ):
    # 顺序是很重要的，首先要求所有线程停止
    for i in range(len(Pool)):
        request_work(None, 'stop')
    # 然后，等待每个线程的终止
    for existing_thread in Pool:
        existing_thread.join( )
    # 清除线程池
    del Pool[:]

```

## 讨论

在多线程程序的构建中有一个常见的错误，即，根据需要生成任意数目的线程。事实上，大多数时候，程序将任务委托给固定的、数目相对较少的一群线程是最好的架构——这也被称为线程池。本节展示了线程池的一个很简单的应用例子，主要关注点是对于 Queue.Queue 的使用，它是线程间通信和同步的最有用和最简单的方式。

在本节中，工作线程运行了 do\_work\_from\_queue 函数。对于工作线程的典型用法，此函数的结构是没有任何问题的，它确实只进行了最少的“处理”。在这个例子中，工作线程通过在每个到来的项前面加上一个“new”字符串来计算出“结果”（注意，这里

其实假设了到来的项是字符串)。在你的应用程序中,你可能会有等同于 `do_work_from_queue` 函数的处理函数,它会做更实质性的处理,还会根据不同的 `command` 参数值进行不同类型的处理。

除了工作线程池,一个多线程程序通常还有用于其他目的的专用线程;比如程序的一些外部实体的接口(GUI、数据库等,它们的库是不保证线程安全的)。本节的方案并未展示这种专用线程。不过,本节方案至少还包含了一个“主线程”,它会开始和结束线程池,确定委托的工作,收集最终的结果和可能需要报告的错误。

在你的应用中,有时需要多次反复开始和结束线程池,有时却不一定。最典型的情况是,你可能在程序的初始化阶段开启线程池,然后让它始终保持运行,最后在程序最终退出前完成收尾工作,结束此线程池。如果你的工作线程是“守护线程”,就像本节方案中的函数 `make_and_start_thread_pool` 对线程所做的设置那样,那意味着如果你的程序中只剩下了工作线程,程序也就不会继续运行了。只要你的主线程一结束,整个程序就结束了。不过这只是一种建议使用的典型框架。至少,方案还提供了一个 `stop_and_free_thread_pool` 函数,这样你就可以在某些时间点上结束并清理你的线程池(并可能用 `make_and_start_thread_pool` 函数再次创建和开始另一个新的线程池)。

使用本节提供的功能的一个例子:

```
for i in ('_ba', '_be', '_bo'): request_work(i)
make_and_start_thread_pool()
stop_and_free_thread_pool()
show_all_results()
show_all_errors()

输出片段通常应该是这样:
Result: new_ba
Result: new_be
Result: new_bo
```

结果的顺序是可能变化的(可能性不是很大)。(如果结果的顺序对你很重要,可以在主线程的工作请求中加入一个数字,并将最后的结果或错误报告给主线程。)

下面是一个发生了错误的例子:

```
for i in ('_ba', 7, '_bo'): request_work(i)
make_and_start_thread_pool()
stop_and_free_thread_pool()
show_all_results()
show_all_errors()
```

这个代码片段的输出通常应该是这样(那两行“结果”不太可能交换顺序):

```
Result: new_ba
Result: new_bo
Error: exceptions.TypeError cannot concatenate 'str' and 'int' objects
```

得到子项 7 的工作线程报告了 `TypeError`,因为它试图将子项和字符串“new”拼接起

来，对于 int 而言这是非法的。不过不用担心，在函数 `do_work_from_queue` 中我们有 `try/except` 声明来捕获任何错误，队列 `Qerr` 和函数 `report_error` 以及 `show_all_errors` 确保不会放过任何错误，除非你明确指明对某些异常开绿灯。在 Python 的通用编程方法中这是一个重要的原则：错误不应该被静静地略过，除非有意为之。

## 更多资料

*Library Reference* 中关于 `threading` 和 `Queue` 模块的文档；*Python in a Nutshell* 中的关于线程的章节。

## 9.5 以多组参数并行执行函数

感谢：Guy Argo

### 任务

需要用不同的多组参数来同时执行一个函数（这个函数可能是“I/O 绑定的”，即它需要花费相当多的时间来完成输入输出操作；若非如此，同时执行没有多大用处）。

### 解决方案

我们对于每组参数使用一个线程。但为了获得好的性能，最好将线程的使用限定于一个有限的线程池：

```
import threading, time, Queue
class MultiThread(object):
    def __init__(self, function, argsVector, maxThreads=5, queue_results=False):
        self._function = function
        self._lock = threading.Lock()
        self._nextArgs = iter(argsVector).next
        self._threadPool = [threading.Thread(target=self._doSome)
                            for i in range(maxThreads)]
        if queue_results:
            self._queue = Queue.Queue()
        else:
            self._queue = None
    def _doSome(self):
        while True:
            self._lock.acquire()
            try:
                try:
                    args = self._nextArgs()
                except StopIteration:
                    break
            finally:
```

```

        self._lock.release( )
    result = self._function(args)
    if self._queue is not None:
        self._queue.put((args, result))
def get(self, *a, **kw):
    if self._queue is not None:
        return self._queue.get(*a, **kw)
    else:
        raise ValueError, 'Not queueing results'
def start(self):
    for thread in self._threadPool:
        time.sleep(0)      # 有必要给其他线程一个执行的机会
        thread.start( )
def join(self, timeout=None):
    for thread in self._threadPool:
        thread.join(timeout)
if __name__=="__main__":
    import random
    def recite_n_times_table(n):
        for i in range(2, 11):
            print "%d * %d = %d" % (n, i, n * i)
            time.sleep(0.3 + 0.3*random.random( ))
    mt = MultiThread(recite_n_times_table, range(2, 11))
    mt.start( )
    mt.join( )
    print "Well done kids!"

```

## 讨论

本节的 `MultiThread` 类提供了一个简单的方法，以多组参数和一个有限的线程池来并行地执行函数。另外，作为一个选项，还可以要求把函数调用的结果放入队列，以方便获取，默认情况下结果是被丢弃的。

`MultiThread` 接受的参数包括：一个函数，一个为该函数准备的参数元组序列，一个可选的线程数目固定且有限的线程池，还有一个指定结果的处理方式的标志。除了构造函数，它还公开了三个方法：`start`，用来启动池中的所有线程，并以给定的参数并行地执行函数；`join`，对池中的所有线程执行 `join` 操作（意味着必须等待池中的所有线程终止）；以及 `get`，获得队列中的结果（当然首先必须在实例化时将可选参数 `queue_results` 设定为 `True`，明确要求把结果放入队列）。在内部，`MultiThread` 类使用私有方法 `doSome` 作为池中的线程的目标调用体。每个线程使用下一个可用的参数元组（由可迭代对象的 `next` 方法提供，并且对 `next` 的调用也被加锁保护），直到所有的工作完成。

像 Python 的通常做法一样，这个模块也可以被作为独立的主脚本运行，这时它只是对自身功能的一个简单展示和自我测试。在这个例子中，它模拟一个班级的孩子们用最快的方式背诵乘法表。

而这个类在真实的应用中则可能涉及与 I/O 紧密相关的函数，那意味着函数会花相当多的时间执行 I/O 操作。如果一个函数是与 CPU 活动紧密关联的，那意味着该函数大部分时间是运行在 CPU 计算上，你简单地一个接一个地完成计算任务，反而可能比并行方式的总体性能要高。对于 Python 而言，即使把你的程序放在多处理器计算机上运行，其表现仍然会符合上面提到的观察结果，这是因为 Python 使用了 GIL（全局解释器锁），纯 Python 代码不能在同一时刻运行在不同的 CPU 上。

输入输出操作释放了 GIL，任何基于 C 的执行大量计算而不用回调 Python 代码的扩展也会（应该）释放 GIL。所以，并行执行是有可能提速的，但仅限于有不少 I/O 操作或者涉及一个合适的基于 C 的扩展的程序，对于执行密集计算的 Python 代码则并不适用。（基于 Python 的不同虚拟机的实现，如 Jython，运行在 JVM[Java 虚拟机]之上，或者 IronPython，运行在 Microsoft 的.NET 平台上，它们理所当然地不受限制：这些观察仅仅适用于广泛流传的“经典 Python”，即 CPython 实现。）

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中的 `threading` 和 `Queue` 模块。

## 9.6 用简单的消息传递协调线程

感谢：Michael Hobbs

### 任务

你想编写一个多线程程序，在其中使用一种简单而强大的消息传递模式，作为同步和通信的基础。

### 解决方案

`candygram` 模块允许你使用在语义上等同于 Erlang 的并行编程的 Python 对应物。为了使用 `candygram`，你先要定义一个适合的类来规范你的线程功能，比如下面这个：

```
import candygram as cg
class ExampleThread(object):
    """只有一个 counter 值和结束标志的线程类"""
    def __init__(self):
        """ 将 counter 设置为 0，运行标志设置为 True """
        self.val = 0
        self.running = True
    def increment(self):
        """ counter 递增 1 """
        self.val += 1
    def sendVal(self, msg):
        """ 将当前 counter 值发给询问的线程 """
```

```

        req = msg[0]
        req.send((cg.self( ), self.val))
    def setStop(self):
        """ 将运行标志设成 False """
        self.running = False
    def run(self):
        """ 线程的入口点 """
        # Register the handler functions for various messages:
        r = cg.Receiver( )
        r.addHandler('increment', self.increment)
        r.addHandler((cg.Process, 'value'), self.sendVal, cg.Message)
        r.addHandler('stop', self.setStop)
        # 继续处理新消息，直到被要求结束
        while self.running:
            r.receive( )

```

为了启动一个线程，要这样做：

```
counter = cg.spawn(ExampleThread( ).run)
```

为了处理 counter 线程的回应，需要一个 Receiver 对象，并且正确注册：

```
response = cg.Receiver( )
response.addHandler((counter, int), lambda msg: msg[1], cg.Message)
```

下面是一些例子指导你使用 counter 和 response 对象：

```

# 告诉线程递增两次
counter.send('increment')
counter.send('increment')
# 请求线程的当前值，并打印线程的回应
counter.send((cg.self( ), 'value'))
print response.receive( )
# 告诉线程再递增一次
counter.send('increment')
# 再次请求线程的当前值，并打印线程的回应
counter.send((cg.self( ), 'value'))
print response.receive( )
# 告诉线程停止运行
counter.send('stop')

```

## 讨论

有了 candygram 模块 (<http://candygram.sourceforge.net>)，Python 开发者们就能够像在 Erlang 语言中一样，采用几乎相同的方式在线程间发送和接受消息。Erlang 因为它优雅的并行编程的内建机制而享有盛名。

Erlang 的方式简单而强大。为了联络另一个线程，我们简单地发个消息给它就行了。为了在并行任务中共享信息，你完全不用考虑什么锁、信号量、互斥量或其他原语。多任务软件的开发者们常常使用消息传递技术来实现一个生产者和消费者模型。当你

将消息传递和一个灵活的 `Receiver` 对象结合起来时，它就变得更加强大了。比如，使用超时和消息模式，线程可以轻松地将消息作为状态机或优先级队列处理。

如果你想更多地了解 Erlang，<http://www.erlang.org/download/erlang-book-part1.pdf> (*Concurrent Programming in Erlang*) 提供了一个完整的介绍。特别要指出的是，`candygram` 模块实现了此书第 5 章和 7.2 节、7.3 节以及 7.5 节描述的所有函数。

本节介绍了通过 `candygram` 在线程间传递消息的基本方法。如果你把本节的代码作为脚本运行，`print` 声明会打印出值 2 和 3。

了解 `candygram.Receiver` 类如何工作是非常重要的。`addHandler` 方法要求至少两个参数：第一个是消息模式，第二个是处理函数。如果发现收到的消息符合消息模式，`Receiver.receive` 方法会调用被注册的函数，并返回函数的结果。当 `Receiver` 调用 `addHandler` 方法时，如果除了提供前两个参数，我们还给出了可选的第三个参数，则第三个参数会被传递给处理函数。如果一个参数是 `candygram.Message` 常量，则 `receive` 在调用处理函数时，将用匹配的消息将此参数替换掉。

本节的代码有四种不同的消息模式：“increment”，(`cg.Process, 'value'`)，“stop”以及(`counter, int`)。“increment”和“stop”模式是很简单的模式，分别匹配任何仅仅含有字符串“increment”和“stop”的字符串。而(`cg.Process, 'value'`)模式匹配一个两元素的元组，其中第一个元素是 `cg.Process` 的实例而第二个是字符串 `value`。最后，(`counter, int`) 模式也匹配一个两元素的元组，其中第一个元素是 `counter` 对象而第二个元素是一个整数。

可以在 <http://candygram.sourceforge.net> 找到更多的有关 `candygram` 包的信息。在这个 web 地址，你还能找到如何指定消息模式、如何设置 `Receiver.receive` 方法的超时以及如何监控衍生的线程的状态的各种细节。

## 更多资料

*Concurrent Programming in Erlang*，见 <http://www.erlang.org/download/erlang-book-part1.pdf>，`candygram` 的主页 <http://candygram.sourceforge.net>。

## 9.7 储存线程信息

感谢：John E. Barham、Sami Hangaslammi、Anthony Baxter

### 任务

你想给每个线程分配一些空间来存储只有它自己能访问的信息。

### 解决方案

线程持有特定的存储是一种有用的设计模式，Python 2.3 还不能直接支持这种方式。不

过，即使是在 Python 2.3 中，我们也可以利用被锁保护的字典来实现类似的功能。另外要提及的是，比起 Python 提供的广泛使用的 `threading` 模块，利用 `thread` 模块支持编写的更底层代码的通用性略好一点，而且也不是很难。

```
_tss = {}

try:
    import thread
except ImportError:
    # 我们运行在一个单线程平台上（至少，Python 解释器
    # 并未被编译成支持多线程），所以对每个调用我们都返回
    # 同样的字典——毕竟只有一个线程
    def get_thread_storage():
        return _tss

else:
    # 我们有线程；所以，开始干活：
    _tss_lock = thread.allocate_lock()
    def get_thread_storage():
        """ 返回一个线程特有的存储字典 """
        thread_id = thread.get_ident()
        _tss_lock.acquire()
        try:
            return _tss.set_default(thread_id, {})
        finally:
            _tss_lock.release()
```

Python 2.4 提供了更简单更快的实现，这要归功于新的 `threading.local` 函数：

```
try:
    import threading
except ImportError:
    import dummy_threading as threading
_tss = threading.local()
def get_thread_storage():
    return _tss.__dict__
```

## 讨论

多线程的一大好处是所有的线程都可以根据需要共享全局对象。不过，常常每个线程也需要为自己准备一点存储空间，比如，储存一个独有的网络或数据库连接。确实，每个面向外部的对象一般来说最好由一个单独的线程来控制，以避免增加出现异常行为的可能性，比如竞争条件等。本节的 `get_thread_storage` 函数通过实现“线程特定存储”的设计模式和返回线程的特定的存储字典解决了这个问题。调用的线程可以使用返回的字典来储存线程私有的数据。本节的解决方案，从某种意义上说，可以算是对 Zope 的面向对象数据库 ZODB 的 `get_transaction` 函数的一个概括。

一个可能的扩展是增加 `delete_thread_storage` 函数。这样的函数会很有用，尤其是如果能找到办法在线程结束时自动调用此函数的话。Python 的线程架构使得这样的任务非

常不易。可以生成一个观察线程，对调用的线程进行 `join`，最后完成删除工作，但此法显得过于笨重。而本节展示的方法无须删除，尤其适合那种备受推荐的通用架构，即可以拥有一个工作线程（典型情况下都是守护线程）池（也可能其中一些是普通的工作线程，而另一些则专注于处理特定的外部源的接口工作），线程在程序开始的时候生成并一直存在。

在进行多线程编程时，我们必须非常小心地探测并阻止竞争条件、死锁等冲突。在本节的代码中，我决定不假设字典的 `set_default` 方法是原子的（原子操作意味着在 `set_default` 工作时不会有线程切换），毕竟增加一个键可能会改变字典的整个结构。如果我做这种假设，我可以去掉加锁的动作，性能将获得极大的提高，但我怀疑这样的假设会使代码更加脆弱并依赖于某个特定版本的 Python（测试结果指出似乎我的假设在 Python 2.3 中是可行的，但那还不够，我希望即使 Python 以后又进行了某些内部的精细调整，我的程序仍能够工作正常）。另一个风险是，如果一个线程终止而另一个新线程又开始了，那么新线程的 ID 可能会和那个刚终止的线程的 ID 一样，因此，新线程可能会错误地使用刚刚结束的线程留下来的“线程特定存储”字典。如果使用前面提到的 `delete_thread_storage` 函数，这个风险会被极大地减轻（但也没有完全消灭）。不过，基于我在应用程序中使用的多线程架构，这个特殊的问题并未给我造成麻烦。如果你使用的架构有些不同，你可能需要对本节的方案进行相应的修改。

对于你的应用需求，如果由于过多地获取和释放锁，本节方案提供的性能不能满足需要，你也不应该直接去掉锁的保护并使自己的代码变得更脆弱，而是应该考虑一个替代的方法：

```
_creating_threads = True
_tss_lock = thread.allocate_lock()
_tss = {}

class TssSequencingError(RuntimeError): pass
def done_creating_threads():
    """ 从线程创建切换到无线程创建过程"""
    global _creating_threads
    if not _creating_threads:
        raise TssSequencingError('done_creating_threads called twice')
    _creating_threads = False
def get_thread_storage():
    """ 返回一个线程特有存储字典 """
    thread_id = thread.get_ident()
    # 如果线程创建过程已经完成
    if not _creating_threads: return _tss[thread_id]
    # 如果仍在创建线程
    try:
        _tss_lock.acquire()
        return _tss.setdefault(thread_id, {})
    finally:
        _tss_lock.release()
```

这个变体增加了一个布尔值开关 `_creating_threads`, 初始化为 `True`。只要这个开关是 `True`, 变体就会使用一种基于锁的方式, 类似于解决方案中的方式。在某个时间点, 当所有应该存在的 (或所有需要访问 `get_thread_storage` 的) 线程都被启动了, 每个线程都获得了它自己的局部存储字典, 你的程序就可以调用 `done_creating_threads`。这个调用将 `_creating_threads` 设成了 `False`, 以后, 任何对 `get_thread_storage` 的调用都将采用一种快捷的方式, 即直接访问全局字典 `_tssno`, 而不再获取锁和释放锁, 也不再创建一个线程的局部存储字典。

只要你的程序能够确定调用 `done_creating_threads` 函数的时间点, 这一小片代码就能够提供更快的执行速度。你极有可能会使用这种变体形式, 如果你的应用程序使用前面提到的那种流行且备受推荐的架构的话: 一个数量有限的生命期较长的守护线程集合, 所有线程在程序运行的早期被创建。如果你的应用程序非常关注性能, 而且你担心锁的开销, 那么毫无疑问应该尝试将它重构成这种架构。那种有很多短命线程的架构通常会严重地影响整体性能。

如果你的程序只需要在 Python 2.4 中运行, 可以使用新的 `threading.local` 函数来提供一个更简单、更快速、更坚固的实现。`threading.local` 返回一个新对象, 任意线程都可以通过此对象获取或者设置任意属性, 与其他线程对同一对象进行的获取和设置是完全独立无关的。在本节解决方案的 Python 2.4 的代码中, 返回的是对象的一个 `__dict__`, 这是为了和 2.3 的代码统一起来。采用这种方式, 你的程序就可以同时运行在 2.3 和 2.4 下, 并可以对两种情况使用不同的最适用的版本:

```
import sys
if sys.version >= '2.4':
    # 插入 2.4 的对 get_local_storage 的定义
else:
    # 插入 2.3 的对 get_local_storage 的定义
```

方案中针对 2.4 的代码还展示了对模块 `dummy_threading` 的使用, 这个模块就像它的兄弟 `dummy_thread` 一样, 在 Python 2.3 中也是可用的。这两个模块在任何平台上都是可用的, 无论 Python 是否被编译为支持多线程。通过使用这些 `dummy` 模块, 可以写出能够运行在任何平台上的应用程序, 如果平台支持多线程的话, 你的程序能够利用其多线程能力, 如果平台不支持多线程, 你的程序一样可以正常工作。在针对 2.3 的代码中, 我们并没有使用基于 `dummy_thread` 的类似方法, 这是因为对于非多线程的平台, 它的开销太高了; 而针对 2.4 的代码, 它的开销则小得多, 所以我们为了获得更好的简洁性而使用了 `dummy_threading` 模块。

## 更多资料

关于设计模式对于线程局部存储的详尽描述 (尽管是针对 C++程序员的), 参考 Douglas Schmidt、Timothy Harrisson、Nat Pryce 的 Thread-Specific Storage: An Object Behavioral Pattern for Efficiently Accessing per-Thread State (<http://www.cs.wustl.edu/~schmidt/>)

PDF/TSS-pattern.pdf); *Library Reference* 中关于 dummy\_thread、dummy\_threading 的文档, 以及 Python 2.4 的 threading.local; 关于 ZODB 的资料见 <http://zope.org/Wikis/ZODB/FrontPage>。

## 9.8 无线程的多任务协作

感谢: Brian Bush、Troy Melhase、David Beach、Martin Miller

### 任务

你有个任务看上去很适合采用多线程技术, 但是你不想忍受由于线程切换而造成的开销。

### 解决方案

生成器被设计来简化迭代, 但它却很适合被当做多任务协作的基础, 这种方式也称为微线程 (microthreading) :

```
import signal
# 致谢: 原思路来自于 David Mertz 的文章
# http://gnosis.cx/publish/programming/charming_python_b7.txt
# 一些“微线程”生成器示例
def empty(name):
    """ 这是一个出于展示目的的空任务 """
    while True:
        print "<empty process>", name
        yield None
def terminating(name, maxn):
    """ 这是一个出于展示目的的计数任务 """
    for i in xrange(maxn):
        print "Here %s, %s out of %s" % (name, i, maxn)
        yield None
    print "Done with %s, bailing out after %s times" % (name, maxn)
def delay(duration=0.8):
    """ 在'duration'秒时间内什么也不做 """
    import time
    while True:
        print "<sleep %d>" % duration
        time.sleep(duration)
        yield None
class GenericScheduler(object):
    def __init__(self, threads, stop_asap=False):
        signal.signal(signal.SIGINT, self.shutdownHandler)
        self.shutdownRequest = False
        self.threads = threads
        self.stop_asap = stop_asap
    def shutdownHandler(self, n, frame):
```

```

    """ 初始化一个关闭的请求 SIGINT."""
    print "Request to shut down."
    self.shutdownRequest = True
def schedule(self):
    def noop( ):
        while True: yield None
    n = len(self.threads)
    while True:
        for i, thread in enumerate(self.threads):
            try: thread.next( )
            except StopIteration:
                if self.stop_asap: return
                n -= 1
                if n==0: return
                self.threads[i] = noop( )
        if self.shutdownRequest:
            return
    if __name__=="__main__":
        s = GenericScheduler([empty('boo'), delay( ), empty('foo'),
                             terminating('fie', 5), delay(0.5),
                             ], stop_asap=True)
        s.schedule( )
        s = GenericScheduler([empty('boo'), delay( ), empty('foo'),
                             terminating('fie', 5), delay(0.5),
                             ], stop_asap=False)
        s.schedule( )

```

## 讨论

微线程（或多任务协作）是一种重要的技术。如果你想在某些复杂应用中使用这种技术，你绝对应该先去看看 Christian Tismer 的 *stackless*，这是一本基于 Python 的关于微线程的书，见 <http://www.stackless.com>。不过通过创造性地使用生成器，也可以尝试协同多任务，而不用涉及 Python 的核心，如同本节的做法一样。

不过本节给出的这种简单的协同多任务方式，并不适合于那种需要长时间运行的任务，尤其是涉及到可能会阻塞系统调用的 I/O 任务。对于这些应用程序，我们最好使用真正的线程，或者使用更强大的工具，如 Python 标准库 `asyncore` 模块提供的事件驱动方式，再比如 Twisted 包，见 <http://twistedmatrix.com/products/twisted>（这可以算是重型武器）。如果你的应用程序只有少量的 I/O 需求，而且可以把你的任务的任何计算切分成小块，每一块都可使用 `yield`，那么本节的方案可能正是你想要的东西。

## 更多资料

David Mertz 的主页，<http://gnosis.cx>，充满了各种奇思妙想；Christian Tismer 的 *stackless python*，Python 中最好的实现协同多任务的方法，见 <http://www.stackless.com>；Twisted Matrix，最好的事件驱动（异步）编程方式，见 <http://twistedmatrix.com>。

## 9.9 在 Windows 中探测另一个脚本实例的运行

感谢: Bill Bell

### 任务

你想在 Windows 环境中确保你的脚本在任何时候都只有一个运行的实例。

### 解决方案

有很多方法可以做到只能启动应用程序的一个实例，但它们都很脆弱——除了那些基于互斥内核对象的方法，比如下面这个。Mark Hammond 的 PyWin32 包提供了所有我们需要的可用来操纵互斥量的 Windows API 的封装接口：

```
from win32event import CreateMutex
from win32api import GetLastError
from winerror import ERROR_ALREADY_EXISTS
from sys import exit
handle = CreateMutex(None, 1, 'A unique mutex name')
if GetLastError() == ERROR_ALREADY_EXISTS:
    # 由于这是脚本的第二个实例，所以要采取
    # 适当的行动，比如：
    print 'Oh! dear, I exist already.'
    exit(1)
else:
    # 这是脚本的第一个实例，让它做它应该
    # 做的工作即可。比如：
    from time import sleep
    for i in range(10):
        print "I'm running", i
        sleep(1)
```

### 讨论

字符串“`A unique mutex name`”被用来作为脚本唯一性的标示，而且它绝对不能是动态生成的，因为这个字符串的值对于所有可能的同时运行的实例都是一样的。手工生成一个新的全局唯一的 ID，并在脚本验证期间插入，也是个不错的选择。根据 Windows 的文档，这个字符串可以包含任何字符，除了反斜线 (\)。在实现了终端服务的 Windows 平台上，可以给字符串增加一个可选的前缀，如 `Global\` 或 `Local\`，但这些前缀可能会使得此字符串对于大多数 Windows 版本不可用，包括 Windows NT、95、98 以及 Me。

Win32 API 调用 `CreateMutex` 创建了一个互斥类型的 Windows 内核对象并返回了它的一个句柄。注意，我们不能关闭这个句柄，因为它必须在我们的程序运行期间始终存在。另外一件重要的事是当我们的进程结束时，我们应当让 Windows 内核来处理有关句柄

清除的工作（以及句柄所指示的对象，如果这个被移除的句柄是指向这个内核对象的唯一句柄的话）。

我们真正关心的事情是 API 调用的返回码，即我们调用 GetLastError API 所获得的返回值。当且仅当我们想要创建的互斥对象已经存在时，返回码才会是 ERROR\_ALREADY\_EXISTS（比如，另一个脚本的实例正在运行中）。

这个方法绝对安全，而且不会受到竞争条件等问题的困扰，即使两个脚本实例试图同时启动（这是可能且合理的，比如，用户可能在活动桌面上双击某图标启动程序，但实际上单击就可以启动程序了）。Windows 的规格说明也保证了只有一个实例能够创建互斥量，另一个会被通知互斥量已经存在。由此可见，互斥是在操作系统内核级别上得到保证的，本节的方法自然也如操作系统一般坚固。

## 更多资料

PyWin32 的 Win32 API 文档 (<http://starship.python.net/crew/mhammond/win32/Downloads.html>) 或 ActivePython (<http://www.activestate.com/ActivePython/>)；微软的 Windows API 文档 (<http://msdn.microsoft.com>)；Mark Hammond 和 Andy Robinson 所著的 *Python Programming on Win32* (O'Reilly)

## 9.10 使用 MsgWaitForMultipleObjects 处理 Windows 消息

感谢：Michael Robin

### 任务

在 Win32 程序中，需要处理消息，但你也同时需要内核级别的可等待对象 (waitable object)，并用它来协调一些活动。

### 解决方案

Windows 应用程序的消息循环，也被称为消息泵，是 Windows 的心脏。花点功夫和精力来确保心脏跳动的正确性和规律性显然是值得的：

```
import win32event
import pythoncom
TIMEOUT = 200 # ms
StopEvent = win32event.CreateEvent(None, 0, 0, None)
OtherEvent = win32event.CreateEvent(None, 0, 0, None)
class myCoolApp(object):
    def OnQuit(self):
        # 假设 areYouSure 是一个全局函数，通过消息对话框
```

```

# 或其他漂亮的窗口来进行最后的检查
if areYouSure( ):
    win32event.SetEvent(StopEvent) #退出消息泵

def _MessagePump( ):
    waitables = StopEvent, OtherEvent
    while True:
        rc = win32event.MsgWaitForMultipleObjects(
            waitables,
            , # Wait for all = false, 所以现在它等待任意一个
            TIMEOUT, # (或 win32event.INFINITE)
            win32event.QS_ALLEVENTS) # 接受各种事件
        # 可以在这里调用函数, 如果它花的时间不长的话。至少
        # 每 TIMEOUT 毫秒它会被执行一次——还可能更频繁,
        # 具体取决于收到的 Windows 消息的数目
        if rc == win32event.WAIT_OBJECT_0:
            # 我们的第一个列表中的事件, StopEvent 被激发
            # 所以我们必须退出, 终止消息泵
            break
        elif rc == win32event.WAIT_OBJECT_0+1:
            # 我们的第二个列表中的事件, OtherEvent 被设置。
            # 做任何需要做的事情即可——可以根据需要
            # 等待任意多的内核对象 (事件、锁、进程、线程、通知, 等等)
            pass
        elif rc == win32event.WAIT_OBJECT_0+len(waitables):
            # 一个 Windows 消息在等待——处理之。(别问
            # 我为什么 WAIT_OBJECT_MSG 没被定义 <
            # WAIT_OBJECT_0...!)。
            # 这种消息服务是 COM、DDE, 以及其他
            # Windows 组件正常工作的重要保障
            if pythoncom.PumpWaitingMessages( ):
                break # 收到了一个wm_quit 消息
        elif rc == win32event.WAIT_TIMEOUT:
            # 超时了
            # 我们在这里做点事 (比如轮询某事)
            pass
        else:
            raise RuntimeError("unexpected win32wait return value")

```

## 讨论

绝大多数 Win32 应用程序需要处理消息, 但你通常也需要等待内核的可等待对象并同时协调很多事情。对于这类应用, 一个良好的消息泵结构是关键, 而本节方案则展示了一个简单而有效的消息泵。

使用本节的消息泵, 消息和其他事件在被寄送后得以很快的分发, 并且超时事件也允许你周期性地轮询其他组件。如果某些正确的调用或事件对象没有暴露在你的 Win32 事件循环中, 你可能需要轮询, 因为很多组件只能运行在应用程序的主线程中, 而不

能运行在生成的（次等的）线程中。

关键是对 Windows API `MsgWaitForMultipleObjects` 的调用，它接受好几个参数。第一个是你想等待的内核对象的元组。第二个参数是一个标志，通常为 0，如果值为 1 则表示必须等到第一个参数指定的所有内核对象变为有信号状态，因此它的值通常总是 0。第三个参数是用来指定能够打断等待的 Windows 消息类型；我们在这里给它传递的值是 `win32event.QS_ALLEVENTS`，确保任何 Windows 消息都能够打断等待状态。第四个参数是超时周期 (ms)，或 `win32event.INFINITE`，如果你确定不需要任何周期性轮询的话。

这个函数是一个轮询的循环（带有一个 `while True`，那意味着只有内部的 `break` 能终止循环）。在循环的每一轮，它调用 API 来等待多个对象。当 API 停止等待时，它返回一个解释停止原因的码值。它的值在 `win32event.WAIT_OBJECT_0` 和 `win32event.WAIT_OBJECT_0+N-1` 之间（N 是作为第一个参数的元组的可等待内核对象数目），包括两端，这意味着如果等待结束了，一定有某个对象成为了有信号状态（对于不同类型的可等待内核对象，有信号的具体含义差别很大）。返回码的值减去 `win32event.WAIT_OBJECT_0` 的差值就是引起等待结束的对象在元组中的索引号。

返回 `win32event.WAIT_OBJECT_0+N` 则表明等待结束的原因是某消息处于等待未决 (pending) 的状态，在这种情况下，本节的代码通过调用 `pythoncom.PumpWaitingMessages` 来处理所有的等待未决的消息（此函数如果收到了一个 `WM_QUIT` 消息会返回 `True`，这样我们就结束了 `while` 循环）。而 `win32event.WAIT_TIMEOUT` 则表明由于超时等待自动结束，我们可以在这里进行轮询。在此例中，没有消息在等待，也没有我们感兴趣的内核对象变成了有信号状态。

基本上，对于本节方案的调整优化应该集中在选择正确的内核对象作为可等待对象（并对各个对象作出适当的反应），以及在周期性轮询的时候做什么事。不过这要求你对 Win32 的细节有一定的认识，无论如何，利用本节的方案肯定要比从头设计一个专用的消息循环函数简单多了。

我怀疑一些讲究血统纯正的极端分子会用一些办法把消息循环封装到一个整洁的模块中，然后传递内核对象的列表让各个程序自己定制对模块的使用，或者使用字典将各种不同的可等待的内核对象映射到不同的执行代码，等等。如果是这样，那不如更进一步，如果愿意还可以使用自定义的元类。不过，我仍然认为正确地使用上面代码的方式是将它复制到你的程序代码的目录中，然后使用你喜爱的文本编辑器来修改这个消息泵，以符合你的程序的需要。

## 更多资料

PyWin32 的 Win32 API 文档 (<http://starship.python.net/crew/mhammond/win32/Downloads.html>) 或 ActivePython (<http://www.activestate.com/ActivePython/>)；微软的 Windows API

文档 (<http://msdn.microsoft.com>)，Mark Hammond 和 Andy Robinson 所著的 *Python Programming on Win32* (O'Reilly)。

## 9.11 用 `popen` 驱动外部进程

感谢: Sébastien Keim、Tino Lange、Noah Spurrier

### 任务

你想驱动一个从标准输入接受命令的外部进程，但你并不关心这个外部进程在标准输出上给出的回应（如果有的话）。

### 解决方案

如果需要操纵其他进程的输入且不关心它的输出，简单的 `os.popen` 函数就够用了。比如，下面是一个通过 `os.popen` 驱动自由软件 gnuplot 完成一些动画的例子：

```
import os
f = os.popen('gnuplot', 'w')
print >>f, "set yrang[-300:+300]"
for n in range(300):
    print >>f, "plot %i*cos(x)+%i*log(x+10)" % (n, 150-n)
    f.flush()
f.close()
```

### 讨论

当你将 Python 作为胶水语言时，有时（尤其是一些比较简单的情况）简洁的 `popen`（来自于标准库模块 `os`）函数就能完全符合你的所有需要。具体地说，当需要驱动一个从标准输入接受命令的外部程序时，只要可以忽略这个程序在标准输出中给出的任何可能的反馈（包括程序可能输出到标准错误输出的错误信息），`os.popen` 就够我们使用了。下面是驱使自由软件 gunplot 的一个例子（当需要获得一个程序的输出而不用考虑它的标准输入时，`os.popen` 同样非常好用）。

语句 `f = os.popen('gnuplot', 'w')` 创建了一个类文件对象，此对象连接到名为“gnuplot”的程序启动的标准输出（为了运行本节的代码，必须在你的 PATH 中安装 gnuplot，由于 gnuplot 是自由软件，并被广泛地移植到了各个平台，因此那应该不是什么问题）。无论你对 `f` 写入了什么东西，外部进程都能在它的标准输入中收到这些内容，就好像我们正在交互地使用这个程序一样。关于 gnuplot 的更多信息，参见 <http://sourceforge.net/projects/gnuplot-py/>，这是一个有趣的对 gnuplot 进行了封装的 Python 接口，其设计实现完全基于本节提出的思路。

当需要的功能比 `os.popen` 能提供的功能更复杂的时候，你可能需要看看 `os` 模块中

`os.popen2` 及其他拥有更大数字的对应函数，或者，在 Python2.4 中，可以看看新的标准模块 `subprocess`。不过，在很多情况下，你都可能会感到失望：一旦超出了基本的部分，驱动（从你自己的程序中）其他的交互式外部程序就变得比较困难了。幸好，我们手上还有个方案：`pexpect`，这是一个第三方的 Python 模块，可以在 <http://pexpect.sourceforge.net/> 找到。`pexpect` 是专为驱动其他程序而设计的，它允许你在检查其他程序反馈的同时发送命令到这些程序的标准输入中。虽然 `pexpect` 是一个可以满足你任何需求的强大模块，但在很多情况下，你根本不需要那些花哨和强大的能力，`os.popen` 就足够用了。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中的 os 模块（特别是 `os.popen`）的文档；`gnuplot`，见 <http://www.gnuplot.info>；`gnuplot.py`，见 <http://sourceforge.net/projects/gnuplot-py/>；`pexpect`，见 <http://pexpect.sourceforge.net/>。

## 9.12 获取 UNIX Shell 命令的输出流和错误流

感谢：Brent Burley、Bradey Honsinger、Tobias Polzin、Jonathan Cano、Padraig Brady

### 任务

需要在一个类 UNIX 的环境中运行一个外部进程并从这个外部进程中获取输出流以及错误流。

### 解决方案

`popen2` 模块能够帮助你获取这两种流，但同时你还需要 `fcntl` 模块的帮助，使得这些流变成非阻塞状态，从而避免死锁，还有一个要用到的模块是 `select`，具体代码如下：

```
import os, popen2, fcntl, select
def makeNonBlocking(fd):
    fl = fcntl.fcntl(fd, fcntl.F_GETFL)
    try:
        fcntl.fcntl(fd, fcntl.F_SETFL, fl | os.O_NDELAY)
    except AttributeError:
        fcntl.fcntl(fd, fcntl.F_SETFL, fl | os.FNDELAY)
def getCommandOutput(command):
    child = popen2.Popen3(command, 1) # 获得命令的 stdout 和 stderr
    child.tochild.close()           # 不需要写入 child 的 stdin
    outfile = child.fromchild
    outfd = outfile.fileno()
    errfile = child.childerr
    errfd = errfile.fileno()
    makeNonBlocking(outfd)         # 不能死锁！使 fd 非阻塞
```

```

makeNonBlocking(errfd)
outdata, errdata = [ ], [ ]
outeof = erreof = False
while True:
    to_check = [outfd]*(not outeof) + [errfd]*(not erreof)
    ready = select.select(to_check, [ ], [ ]) # 等待输入
    if outfd in ready[0]:
        outchunk = outfile.read( )
        if outchunk == '':
            outeof = True
        else:
            outdata.append(outchunk)
    if errfd in ready[0]:
        errchunk = errfile.read( )
        if errchunk == '':
            erreof = True
        else:
            errdata.append(errchunk)
    if outeof and erreof:
        break
    select.select([ ], [ ], [ ], .1) # 给一点时间填充缓冲区
err = child.wait( )
if err != 0:
    raise RuntimeError, '%r failed with exit code %d\n%s' % (
        command, err, ''.join(errdata))
return ''.join(outdata)

def getCommandOutput2(command):
    child = os.popen(command)
    data = child.read( )
    err = child.close( )
    if err:
        raise RuntimeError, '%r failed with exit code %d' % (command, err)

```

## 讨论

本节展示了如何执行一个 UNIX shell 命令并在 Python 中获取此命令的输出流与错误流。与之形成对比的是 `os.system`，它会将两种流直接发送到终端。函数 `getCommandOutput` 执行一个命令，并返回命令的输出。如果命令失败，`getCommandOutput` 会抛出异常，并使用命令启动的 `stderr` 中的文本作为异常的参数的一部分。

任务要求从一个运行的子进程中同时且独立地获取输出和错误流，这个要求引起了代码的绝大部分的复杂性。如果子进程尝试向流中写入而父进程正在等待另一个流中的数据，普通（阻塞）读取调用就可能造成死锁，因此，必须将流设成非阻塞状态，而且必须用 `select` 来等待两个流中的数据。

注意，第二个 `select` 调用仅仅是为了在每次读取之后增加 0.1s 的休眠时间。与我们的直

觉相反，这使得代码运行得更快，因为它给了子进程时间将更多数据放入缓冲区。如果没有这个步骤，父进程可能每次只能读取几个字节，代价过于高昂。调用 `time.sleep(0.1)` 也是可以的，但是既然我已经在别处调用过 `select.select` 了，我认为没有必要再导入 `time` 模块了。

如果你只想获取输出流，并不关心输出到终端的错误流，可以使用更简单的 `getCommandOutput2`。为了把错误流压缩掉，很简单——只要给命令增加一个 `2>/dev/null` 即可，比如：

```
listing = getCommandOutput2('ls -l 2>/dev/null')
```

还有一种可能是使用 `os.popen4` 函数，它将子进程的输出与错误流结合在了一起。不过，对于这个例子，根据输出流和错误流在子线程中缓存的方式，它们结合在一起的输出可能会很混乱。

在 Python2.4 中，可以使用新的标准模块 `subprocess` 中的 `Popen` 类来替代 `popen2.Popen3`。不过，本节展示的主要问题（即使用 `fcntl` 模块和 `select` 使文件成为非阻塞状态并协助父进程与子进程的交互）并不会因为你选择 `popen2` 或 `subprocess` 而有所不同。

本节方案适用于一个类 UNIX 的平台。Cygwin，它能在 Windows 平台中模拟很多 UNIX 下的任务，但它仍可能不敷使用；比如，它没有提供方法来设置文件的非阻塞模式，也没有对一般文件进行 `select` 的能力（在 Windows 中，你只能对 `socket` 进行 `select` 操作，而不能操作文件）。如果必须使用这种有问题的非 UNIX 平台，你可能需要用另一种方式，即使用临时文件：

```
import os, tempfile
def getCommandOutput(command):
    outfile = tempfile.mktemp()
    errfile = tempfile.mktemp()
    cmd = "(%s) > %s 2> %s" % (command, outfile, errfile)
    err = os.system(cmd) >> 8
    try:
        if err != 0:
            raise RuntimeError, '%r failed with exit code %d\n%s' % (
                command, err, file(errfile).read())
        return file(outfile).read()
    finally:
        os.remove(outfile)
        os.remove(errfile)
```

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中标准库模块 `os`、`popen2`、`fcntl`、`select` 以及 `tempfile` 的文档；*Library Reference* 中的 `subprocess` 模块（仅 Python 2.4）。

## 9.13 在 UNIX 中 fork 一个守护进程

感谢: Jürgen Hermann、Andy Gimblett、Josh Hoyt、Noah Spurrier、Jonathan Bartlett、Greg Stein

### 任务

需要在 UNIX 或类 UNIX 系统中 fork 出一个守护进程，这个过程要求一系列精确的系统调用。

### 解决方案

UNIX 守护进程必须同它们的控制终端和进程组分离。要做到这一点并不困难，但我们仍需要小心从事，因此很有必要写一个 `daemonize.py` 模块来应对这种常见的需求：

```
import sys, os
''' 将当前进程 fork 为一个守护进程

注意：如果你的守护进程是由 inetd 启动的，不要这样做！inetd
完成了所有需要做的事，包括重定向标准文件描述符，需要
做的事情也许只有 chdir() 和 umask() 了。
'''

def daemonize(stdin='/dev/null', stdout='/dev/null', stderr='/dev/null'):
    ''' Fork 当前进程为守护进程，重定向标准文件描述符
    (默认情况下会定向到 /dev/null)
    '''

    # Perform first fork.
    try:
        pid = os.fork()
        if pid > 0:
            sys.exit(0) # 第一个父进程退出。
    except OSError, e:
        sys.stderr.write("fork #1 failed: (%d) %s\n" % (e.errno, e.strerror))
        sys.exit(1)

    # 从母体环境分离
    os.chdir("/")
    os.umask(0)
    os.setsid()

    # 执行第二次 fork.
    try:
        pid = os.fork()
        if pid > 0:
            sys.exit(0) # 第二个父进程退出
    except OSError, e:
        sys.stderr.write("fork #2 failed: (%d) %s\n" % (e.errno,
                                                          e.strerror))
        sys.exit(1)
```

```
# 线程已经是守护进程了，重定向标准文件描述符
for f in sys.stdout, sys.stderr: f.flush()
si = file(stdin, 'r')
so = file(stdout, 'a+')
se = file(stderr, 'a+', 0)
os.dup2(si.fileno(), sys.stdin.fileno())
os.dup2(so.fileno(), sys.stdout.fileno())
os.dup2(se.fileno(), sys.stderr.fileno())

def _example_main():
    '''示例函数：每秒打印一个数字和时间戳'''
    import time
    sys.stdout.write('Daemon started with pid %d\n' % os.getpid())
    sys.stdout.write('Daemon stdout output\n')
    sys.stderr.write('Daemon stderr output\n')
    c = 0
    while True:
        sys.stdout.write('%d: %s\n' % (c, time.ctime()))
        sys.stdout.flush()
        c = c + 1
        time.sleep(1)
if __name__ == '__main__':
    daemonize('/dev/null','/tmp/daemon.log','/tmp/daemon.log')
    _example_main()
```

## 讨论

要在 UNIX 平台中 fork 一个守护进程，我们必须执行一系列特定的系统调用，在 W. Richard Stevens 的巨著 *Advanced Programming in the UNIX Environment* (Addison-Wesley) 中对此有详细的介绍。我们需要 fork 两次，结束每个父进程并让原始进程的孙进程运行守护进程的代码。这种做法能够将守护进程和调用终端分离开来，之后，守护进程就可以始终不受干扰地运行（尤其是作为没有用户交互过程的服务进程，如 web 服务等），即使调用终端被关闭了。当你的脚本运行本模块的 `daemonize` 函数时，你唯一能注意到的视觉上的变化是，shell 提示符很快就回来了。

至于如何以及为何这种方式能够工作于 UNIX 和类 UNIX 系统，请阅读 Stevens 的大作。另一个关于“守护进程的 fork”的实践和理论的重要信息源是 UNIX Programming FAQ，见 [http://www.erlenstar.demon.co.uk/UNIX/faq\\_2.html#SEC16](http://www.erlenstar.demon.co.uk/UNIX/faq_2.html#SEC16)。

总结一下：第一个 fork 是为了让 shell 返回，同时让你完成 `setsid`（从你的控制终端移除，这样就不会意外地收到信号）。`setsid` 使得这个进程成为了“会话领导（session leader）”，即如果这个进程打开任何终端，该终端就会成为此进程的控制终端。我们并不需要一个守护进程有任何控制终端，所以我们又 fork 一次。在第二次 fork 之后，此进程不再是一个“会话领导”，这样它就能打开任何文件（包括终端）且不会意外地再次获得一个控制终端。

Stenvens 的书以及 UNIX Programming FAQ 都提供了 C 编程语言的例子，但既然 Python 标准库暴露了所有的 POSIX 接口，你完全可以用 Python 实现一切。用于守护进程的标准 C 代码完全可以被翻译成 Python，唯一需要注意的区别是 Python 的 os.fork 并不返回 -1 来提示错误，而是抛出一个 OSError 异常。因此，我们不像在 C 代码中那样测试 fork 的返回码是否小于 0，而是在一个 try/except 语句的 try 子句中执行 fork，这样我们就可以在异常发生时捕获错误，并将适当的诊断信息打印到标准错误输出。

## 更多资料

*Library Reference* 和 *Python in a Nutshell* 中关于标准库模块 os 的文档；UNIX manpage 中的 fork、umask 和 setsid 系统调用；W.Richard Stevens 的 Advanced Programming in the UNIX Environment (Addison-Wesley)；以及关于守护进程的 UNIX Programming FAQ，见 [http://www.erlenstar.demon.co.uk/UNIX/faq\\_2.html#SEC16](http://www.erlenstar.demon.co.uk/UNIX/faq_2.html#SEC16)。