Learning Python

Python 语言入门

O'REILLY®

中国电力出版社

Mark Lutz & David Ascher 著

陈革 冯大辉 译

Python 语言入门

Mark Lutz & David Ascher 著

陈革 冯大辉 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

中国电力出版社

图书在版编目（CIP）数据

Python 语言入门 /（美）鲁兹（Lutz, M.），（美）阿舍尔（Ascher, D.）编著；陈革，冯大辉译．- 北京：中国电力出版社，2001

（开源软件丛书）

书名原文：Learning Python

ISBN 7-5083-0580-9

Ⅰ.P... Ⅱ.①鲁... ②阿... ③陈... ④冯... Ⅲ.PYTHON 语言－程序设计、Ⅳ.TP312

中国版本图书馆 CIP 数据核字（2001）第 16635 号

北京市版权局著作权合同登记

图字：01-2001-0880 号

书　　名 / Python 语言入门

书　　号 / ISBN 7-5083-0580-9

责任编辑 / 刘江

封面设计 / Edie Freedman，张健

出版发行 / 中国电力出版社

地　　址 / 北京三里河路 6 号（邮政编码 100044）

经　　销 / 全国新华书店

印　　刷 / 北京市地矿印刷厂

开　　本 / 787 毫米 × 1092 毫米　16 开本　25.75 印张　400 千字

版　　次 / 2001 年 4 月第一版　2001 年 4 月第一次印刷

印　　数 / 0001-5000 册

定　　价 / 55.00 元（册）

译者序

能为读者介绍Python和本书我们感到十分荣幸。Python是一种自由的开放源码的解释性语言，简单易学且不失严谨，所以得到初学者和编程专家的共同赞誉。它像BASIC一样的交互式特点使得它非常容易接近和了解，很适合于进行快速原型开发，也适合于教学和程序设计的初学者；它严谨的模块和对象体系使得它同样适用于大型软件开发。

Python的设计者Guido一开始就很注重它的易学性和可读性，他还有一个伟大的梦想，就是像普及读写能力一样普及编程能力，你也许某一天可以很方便地给自己的手机或微波炉写一段程序。

本书是关于Python的入门和提高的书籍，原作者富有经验，书中有大量例子、练习及答案，很适合于自学。

本书由陈革和冯大辉先生合作译出，第一至第五章和附录由冯大辉翻译，前言和第六至第十章由陈革翻译。全书的统稿由陈革负责。感谢中国电力出版社的刘江对本书全稿做了仔细的审校。译校过程中对原书的一些错误已经做了更正。

我们已竭尽全力保证翻译的质量，但时间仓促，错误难免。对于书中的不足之处敬请读者赐教，请发邮件到chenge99@263.net。关于本书的更多信息请注意http://python123.yeah.net。

现在就踏上Python的探索学习之旅吧！

译者

2001年3月

作者简介

Mark Lutz 是一位软件设计师和 Python 专家。他还是 O'Reilly 公司出版的另两本 Python 书、《Programming Python》和《Python Pocket Reference》的作者。自 1992 年以来，Mark 为各种系统编写了大量的 Python 程序，并有丰富的 Python 教学经验。他是 Python 社区的积极分子。

David Ascher 多才多艺，是科学家/软件工程师/培训专家的三位一体。白天，他研究视觉。夜晚，他花大量的时间钻研计算机科学。和 Mark 一样，他也是受欢迎的 Python 老师。

封面介绍

本书封面上的动物是林鼠（wood rat，鼠科林鼠属）。林鼠可以在中北美各种环境中生存（大多在多岩石、灌木和沙漠地区），通常远离人类，虽然它们偶尔也会损害一些庄稼。它们善于攀援，在 6 米高的树上筑巢而居。也有一些种类的林鼠居住在地下或岩石中的洞穴里。

这些灰褐色、中等个头的啮齿类动物又称 pack rat（收集鼠），它们爱把各种各样的东西运回家里，无论是否有用，尤其喜欢闪闪发亮的东西，如易拉罐、玻璃和金属器皿。

目录

第二部分 外围层

第三部分 附录

前言

关于本书

这本书将对Python编程语言做全面而精炼的介绍。Python是一种流行的面向对象语言，既可用于独立的程序，也可用于脚本程序，适用于各种领域。它是自由的、可移植的、强大的，而且非常易于使用。无论你是新手还是职业开发者，本书都将快速地把Python语言的核心介绍给你。在我们进入细节前，先在前言里谈谈本书的整体思路。

本书的范围

本书涵盖了Python语言的精华，但我们的介绍出于速度和篇幅的考虑，将限制在一个较小的范围，主要集中于核心概念，有时谨慎地做了一些简化。所以本书是一本入门教程，同时也是更高级、更复杂课程的铺路石。

例如，我们将不涉及Python/C的集成——这是一个大而复杂的课题，有许多大而复杂的例子，而且对很多基于Python的系统是很重要的。我们也不会谈论Python社区、历史以及开发Python的哲学等等。而对流行的Python应用如GUI（图形用户界面）、系统工具、网络脚本、数值计算等，我们只是在最后作了一个简短的介绍（如果提到了的话），当然，这将错过一些重要的东西。

总体来说，Python的宗旨是想提高脚本语言的水平。它的一些概念需要更多的背景知识，所以我们建议你在学完本书后要进行更深入的学习。我们希望本书的读者最终将获得更深入和更完整的理解，比如通过学习O'Reilly公司出版的《Programming Python》。再接下去就得研究现实中的例子了（注1）。

尽管范围有限（也许正因如此），你将发现本书非常适合作为学习Python的第一本书。你将学到用于编写独立程序或脚本程序需要的一切。学完本书，你将不仅学到语言本身，而且知道如何应用在日常的工作中。碰上更高级的课题你也能应付。

本书的风格

本书大部分内容是以一个三天的Python实践训练课程的材料为基础的。你将在每一章后面看到练习，附录三里有每个练习的答案。练习是为了让你能够立即开始写程序，而这是本书的一个亮点。我们强烈推荐你彻底地研究这些练习，以获得Python编程经验，练习中还有一些内容是正文里没有讲到的。如果你在哪儿卡住了，书后的答案应该是有帮助的。自然，你需要安装Python来运行这些练习，我们很快会讲到如何安装。

本书旨在快速地介绍语言基础，因此我们按语言的主要特征，而不是按例子来组织材料。我们将自底向上介绍：从内置对象类型到语句，再到程序模块单元等等。每一章都是基本独立的，但后面的章节将用到前面介绍的概念（例如我们介绍类时假设你知道如何写函数），所以按顺序阅读可能最容易理解。总体来看，本书分成三个部分：

第一部分：语言核心

书的这一部分自底向上地介绍了Python语言，每个主要的部件构成了一章（类型、函数等等），大多数例子较小，相对独立（揭示了我们想说明的要点）。这部分占了本书的大部分，也是本书的重点。

注1：请访问http://www.ora.com和http://www.python.org以获得更多的细节。《Programming Python》是由本书的作者之一写的。正如它的标题所示，它探讨了实践中的编程细节。

第一章“开始”。我们以对Python的快速介绍开始，然后是如何运行Python程序，以便你可以立刻开始运行例子和练习。

第二章“类型与操作符”。接下来，我们探索Python的主要内置对象类型：数字、列表、字典等等。单凭这些工具你就可以用Python做很多事了。

第三章“基本语句”。这一章介绍Python的语句——Python里用来创建和处理对象的代码。

第四章“函数”。本章开始介绍Python的高层结构工具，函数是一种组织可重用代码的简单方式。

第五章“模块”。Python的模块使你可以把语句和函数组织成大的部件，本章将介绍如何创建、使用和重新装入模块。

第六章“类”。我们将探索Python的面向对象编程（OOP）工具：类。你将会看到，Python的面向对象编程主要是在链接的对象里查找名字。

第七章“异常”。这一章介绍了Python的异常处理模型和语句。这安排在本部分最后一章，因为如果你愿意的话，异常可以用类来表示。

第二部分：外围层

在这一部分，我们展示了Python的内置工具，并用在一些小的例子程序里。

第八章“内置工具”。本章介绍了一组模块和函数，它们包含在缺省的Python安装里。当然，你可以把它们当做任何Python用户都可以访问的最小模块组。了解这些标准工具很可能节省你几周的工作时间。

第九章“用Python完成常见的任务”。本章介绍了一些较重要的程序。我们介绍了一些小但有用的程序，以展示如何把前面介绍的语言核心和内置工具组合起来。我们介绍了大多数Python用户都感兴趣的三个领域：基本任务、文本处理和系统接口。

第十章“框架和应用”。最后一章介绍了如何用标准库和第三方工具做真正的应用，这一章的程序是最复杂的。我们最后以对JPython的简短介绍和一个来自实践的例子结束全书，JPython是Python在Java上的移植。

第三部分：附录

本书最后是三个附录，列出了网上的Python资源（附录一），给出了特定平台Unix，Windows和Macintosh相关的提示（附录二），并给出了每章后练习的答案（附录三）。O'Reilly（http://www.ora.com）的《Python Pocket Reference》和可自由获取的参考手册（译注1）可作为细节的补充材料。

对读者的要求

实际上没有太多要求。这是一本入门书，它也许不太适合从未接触过电脑的人（我们不会花时间解释计算机是什么），但我们对你的编程背景没有做太多的假设。另一方面，我们也没有假设读者是“傻瓜”。用Python可以很容易地做有意义的事，我们希望能向你展示如何做。偶尔我们会把Python与其他语言如C、C++、Pascal等作比较，但如果你以前没有用过这些语言，你完全可以把它们忽略掉。

我们应该先声明的是：Python这个名字是它的创造者Guido van Rossum根据英国广播公司的喜剧系列“*Monty Python*的飞行马戏团”取出来的。由于这个原因，本书的许多例子与这个喜剧有关。例如，传统的名字“foo”和“bar”变成了“spam”和“eggs”。你用不着熟悉这些喜剧也可以理解书中的例子（符号只是符号而已）。

本书更新

本书的更新、补充和校正都在Web上提供，网址是下列之一：

- http://www.oreilly.com（O'Reilly的网站）

译注1：指Guido编写的《Library Reference》（下称库参考）和《Language Reference》（下称语言参考）。

- http://rmi.net/~lutz（Mark的网站）
- http://starship.skyport.net/~da（David的网站）
- http://www.python.org（Python的主站）

排版约定

本书中使用如下排版字体约定：

斜体（*Italic*）
　用于文件名和命令名称。

固定宽度（Constant Width）
　用于代码实例和代码元素名。

等宽粗体（**constant width bold**）
　用于在一些代码例子中作强调作用或用户输入。

等宽斜体（Courier Italic）
　用于强调代码段中的那些由工具自动生成的代码。

关于本书的程序

本书及书中的程序基于Python1.5版，但由于我们主要是讲语言核心，我们所讲的大部分在后续版本中不会有太多改变（注2）。本书的大部分也适用于以前的版本。根据经验，最新的Python也是最好的。因为本书主要讲语言核心，大部分也适用于JPython，JPython是基于Java的Python。

注2：很可能。从《Programming Python》这几年的流行来看，这门语言本身改变很少，而当它改变时，也通常是向后兼容的（Guido增加了一些东西，但很少改变已经有的东西）。外围的工具如Python/C API和Tkinter GUI接口似乎更容易改变，但我们在这里将忽略它们。当然你还是应该检查新版本的发行声明看有什么新东西。

书中的例子、联系和答案的代码可以从O'Reilly的网站获得：http://www.oreilly.com/catalog/lpython/。

你该怎样运行例子呢？我们将介绍启动Python的细节，第一步当然是安装Python。你总可以在Python的主站点www.python.org获得最新和最好的Python版本。在那里，你将找到Python的可执行版本（解压缩并运行），也会找到源代码版本（需要自己编译）。在一些Linux光盘或较厚的Python书附带的光盘上也有Python。两种版本的安装都有文档，我们就不多说了，请看第一章的简略介绍（详细的安装细节参见《Programming Python》一书）。

建议与评论

本书的内容都经过测试，尽管我们做了最大的努力，但错误和疏忽仍然是在所难免的。如果你发现有什么错误，或者是对将来的版本有什么建议，请通过下面的地址告诉我们：

美国：

O'Reilly & Associates,Inc.
101 Morris Street
Sebastopol, CA 95472

中国：

100080北京市海淀区知春路49号希格玛公寓B座809室
奥莱理软件（北京）有限公司

询问技术问题或对本书的评论，请发电子邮件到：

info@mail.oreilly.com.cn

最后，你可以在WWW上找到我们：

http://www.oreilly.com
http://www.oreilly.com.cn

感谢

我们要感谢所有参与完成本书的人。特别要感谢我们的编辑 Frank Willison 和 O'Reilly。也要感谢早期参与评审本书的每个人：Eric Raymond, Guido van Rossum, Just van Rossum, Andrew Kuchling, Dennis Allison, Greg Ward 和 Jennifer Tanksley（译注 2）。我们要特别感谢 Guido van Rossum 和 Python 社区创建了如此有趣和有用的语言。与大多数自由软件一样，Python 是许多英雄努力的成果。

Mark 的话：

因为写作《Programming Python》，使得我有机会周游全国教授 Python 的初学者。除了累积起来的飞行距离外，这些课程帮助我提炼核心的语言材料，构成了本书的第一部分。我想感谢参与我早期课程的学生，你们的反馈对我形成本书的构想十分重要。我还要特别感谢 Softronex，使我这个夏季有机会在波多黎各教授 Python。

最后是一些个人的感谢。感谢合作者 David Ascher，感谢他的辛勤工作和耐心。感谢我写本书期间在洛克希德·马丁公司的同事。感谢 Carl Sagan 的灵感，感谢老子的深奥思想。尤其得感谢我的妻子 Lisa 和我的孩子 Michael、Samantha 和 Roxanne，感谢她们忍耐又一个写作计划。我还欠孩子们一次旅行呢。

1998 年 11 月

科罗拉多州，Longmont 市

David 的话：

除了感谢前面列出的人外，我还想特别感谢几个人。

首先感谢 Mark Lutz 邀请我与他合作此书，并大力支持我的工作。感谢在早期鼓

译注 2：从这个评审名单中你可以发现许多开源软件和 Python 社区的名人。如此强大的技术评审阵容正是 O'Reilly 图书成为名牌的原因之一。

励我的人们，特别是Guido、Tim Peters、Don Beaudry和Andrew Mullhaupt，这是一个迟到的感谢。

与Mark一样，我也开设了教授Python和JPython的课程。这些课程的学生帮助我发现Python中最难学习的部分（幸运的是并不多），也提醒我这门语言的令人愉快的一面。我感谢他们的反馈意见，我也要感谢那些给予我机会发展这些课程的人们：Jim Anderson（布朗大学），Cliff Dutton（当时他在Distributed Data Systems公司），Geoffrey Philbrick（Hibbitt, Karlsson & Sorensen, Inc.），Paul Dubois（Lawrence Livermore国家实验室）和Ken Swisz（KLA-Tencor）。

感谢我的学科指导教授Jim Anderson、Leslie Welch和Norberto Grzywacz，他们支持我在Python上的努力，尤其是这本书，虽然他们不一定理解我为什么在写这本书，但仍然信任我。

听我的Python福音的第一批牺牲者忍受了我早期的热情（也许有人说是狂热），他们应被授予金奖，他们是：Thanassi Protopapas、Gary Strangman和Steven Finney。Thanassi还为本书的样稿提出了独特而有用的反馈意见。

最后感谢我的家人：我的父母总是鼓励我做想做的事，我的兄弟Ivan让我想起了最早接触编程的日子，我的妻子Emily支持并深信我可以写一本书。谢谢我的儿子Hugo至少是有时候允许我使用键盘。当我收到Mark关于本书的电子邮件时，他刚三天大。现在他已经18个月大了。真是美妙的一年半。

对本书的读者，我希望你们能喜欢这本书，而且通过它喜欢Python语言。要知道，通过学习Python，我自己在计算方面比预期的收获大得多。我协助写本书的目标也是想给读者带来同样的经验。如果你学Python的目标是解决一个具体的问题，我希望Python变得不可见的透明，让你把精力集中在你的问题上。然而，我猜测至少有一些读者会有我发现Python时同样的反应，发现Python是一个值得进一步学习的世界。如果你就是这样，要注意探索Python不一定是一个短时间的过程。经过了无数小时，我仍然在四处搜索，兴趣盎然。

1998年11月

加州，旧金山

第一部分

语言核心

在第一部分，我们学习Python语言本身。我们称这部分为“语言核心”。因为我们要把精力集中于Python编程基础：它的内置类型、语句以及程序模块化的工具。读完这一部分并做完练习，你就可以写自己的脚本了。

我们在标题中用“核心”一词的目的，也因为这一部分并非这门语言的全部细节内容。当我们在学习这部分的过程中，也许需要解决一两个费解的问题，但更重要的是，在这里学到的基础知识能帮助你理解可能出现的例外情况。

本章内容：

- 为什么要选择Python?
- 如何运行Python程序?
- 初览模块文件
- Python设置细节
- 总结
- 练习

第一章

开始

这一章以一个非技术性的介绍开始，然后快速地讲一下运行Python程序的方法。主要的目的是使你能在自己的机器上运行Python代码。这样在以后的章节中，你就可以随例子和练习一起学习了。用这种方式，我们还将研究Python设置的基础——使你开始工作就可以了，你不必非要在自己的机器上照着书这样做；但如果可能的话，我们强烈鼓励你这样做。即使不能的话，当你在自己机器上开始编码时这一章仍是很有用的。

我们还将在这里快速浏览一下模块文件，在本书中你先看到的大部分例子都是在Python的交互解释器命令行下输入的。一旦你退出Python，这种方式下输入的代码就会消失。如果想在文件中保存你的代码，你需要知道一点模块的知识，所以在这里也介绍了模块基础，在本书的后面章节我们有关于模块的详细内容。

为什么要选择 Python?

如果你已经买了这本书的话，你很可能已经知道Python是什么，以及为什么它是一门值得学习的工具。如果没有的话，你可能要读完本书后才会喜欢Python。但在转入细节前，我们想用一些篇幅介绍一下在Python流行背后的主要原因。（即使你不注意这些技术性细节，你的经理可能会的呦！）

一个高层的总结

把Python描述成一种面向对象的脚本语言可能是最合适的：它的设计混合了传统语言的软件工程的特点和脚本语言的易用性，Python的最棒的那些特性可以告诉我们全部。

它是面向对象的

Python是一种面向对象的语言，它的类模式支持了诸如多态、操作符重载、多重继承等高级概念。而且Python内容的动态类型、面向对象编程（OOP）都十分易于应用。事实上，即使你不懂这些术语，你仍将发现学习Python比起学习其他面向对象语言容易得多。

除了作为一种强大的代码结构和可重用手段外，Python的面向对象编程的特性使它成为面向对象的系统语言如C++和Java的理想的脚本工具。例如用适当的粘合（glue）代码，Python程序可以对C++或Java的类实现子类化定制。当然了，面向对象编程只是Python中的一个选择，不强迫自己成为面向对象的高手同样可以继续进行学习。

它是自由的

Python是自由（free）软件，即近一段时间所说的开源软件。像Tcl和Perl一样，你可以从Internet上免费得到它的整个系统，拷贝它或在你的系统中嵌入它都没有限制，随你的产品分发也无所谓，实际上只要你愿意的话，你甚至可以出售它。

但别理解错了，“free”并不代表“无支持”，相反，Python在线社区对用户需求的响应和商用软件一样快。而且，Python是完全开放源码的，它加强了开发者的能力并产生了一个大的专家团体，虽然研究或改变一个程序语言的实现不是对每一个人来说都那么有趣，但知道有源码可作为最终的帮助和文档资源是令人欣慰的。

它是可移植的

Python是用可移植的ANSI C写成的，实际上可以在今天所用的各主要平台上编

译、运行。例如，它运行在UNIX系统、Linux、MS-DOS、MS-Windows（95，98，NT）、Macintosh、Amiga、BeOS、OS/2、VMS、QNX等等。而且，Python程序可以自动编译成可移植的字节码（*bytecode*），可在安装了兼容的Python版本的平台上一样运行（更多的内容请看“它易于使用”一节）。

这意味着使用Python语言核心的程序可以运行在Unix、MS-Windows和其他有Python解释器的任何平台上。大多数的Python移植也包括了特定平台上的扩展程序（如MS-Windows上的COM支持），但是核心的Python语言和库在各处都一样工作。

Python还包含一个叫Tkinter的与Tk GUI系统的标准接口。可移植到X Window系统、MS-Windows和苹果系统上，在每一个平台上都有一个友好的界面。通过使用Python Tkinter的API接口，Python程序无需改变，就可以在各主要GUI系统上实现图形用户接口的全部特性。

它是功能强大的

从它的某种特点上来看，Python是一个杂合体。它丰富的工具集使它位于传统脚本语言（如Tcl、Scheme和Perl）和系统语言（如C、C++和Java）之间。Python提供了所有脚本语言的简单、易用性，同时具有在程序开发语言中可以找到的更高级的典型编程工具。而与有些脚本语言不同，Python这些特点使它在实际开发项目的时候十分有用。下面是我们在Python高级工具箱中可以找到的东西：

动态类型

在代码中，Python跟踪程序运行时使用的各种对象，它们不需要复杂的类型和大小的声明。

内置的对象类型

Python提供常用的数据结构如列表（list）、字典（dictionary）、字符串（string）作为语言的基本部分。我们将看到，它们既有弹性又易于使用。

内置工具

为了处理这些对象类型，Python进行功能强大的、标准的操作，包括合并（concatenation）、片段（slice）、排序（sort）、映射（map）等等。

库软件

为了处理更特定的任务，Python也收集了很多预编码的库工具，从正则表达式到网络，以及对象的持续性都可以支持。

第三方软件

因为Python是自由软件，它鼓励开发者提供Python内置工具之外的预编码工具，你可以找到对COM、图像、CORBA ORB、XML等等的免费支持。

自动内存管理

Python自动分配、回收（“垃圾收集”）不再使用的对象，并尽可能地根据需要增加或缩小使用内存。是Python，而不是你，跟踪底层内存的细节。

大型程序的支持

最后，为了创建大型程序，Python也包括了模块、类、扩展程序这样的工具；它们允许你把系统组成组件，执行面向对象编程操作，完美地处理事件。

尽管Python有不少工具，但它保持了十分简单的语法和设计。我们将会看到，其结果是产生了强有力的编程工具，提供了脚本语言的易用性。

它是可混合的

Python程序可十分轻易地与其他语言写成的组件“粘合”在一起。严格地说，通过使用Python/C集成API，Python程序可以通过C或C++写成的组件进行扩展，也可以嵌入到C/C++程序中。这意味着你可以根据需要向Python程序添加功能，或在其他环境系统中使用Python。

虽然我们不会对Python/C的集成谈论太多，但这是这门语言的一个特点，也是通常Python被称为脚本语言的一个原因。通过Python和可编译语言（C/C++）的混合，它成为易用的前端语言和定制工具。这也使Python在快速原型方面很优秀：Python首先可以使系统实现快速开发，以后可移向C，根据性能的需要逐步实现。

说到“粘合”，Python在MS-Windows平台上的Pythonwin移植API也允许Python程序同其他用COM写成的组件通信，允许与Visual Basic进行强有力的交互。一个新的Python交互的实现是JPython，用于Python程序同Java程序的通信，使Python成为基于Java Web应用的一个理想的工具。

它易于使用

Python 兼顾了快速开发周期和语言简单性，这使编程更加有趣。运行 Python 程序，你只要简单地键入并运行就可以了，没有中间的编译和连接步骤（如同C/C++那样）。同其他解释性的语言类似，程序修改后 Python 立刻执行，这带给我们交互的编程经验和快速开发。严格地说，Python 程序被编译成称为字节码（bytecode）的中间介质的形式，然后由解释器运行。但编译这一步自动且对程序员来说是隐藏的。Python达到了解释器的开发速度，而又避免了作为纯解释语言性能上的损失。

当然了，开发周期短只是Python易用性的一个方面。它还有简单、缜密的语法和功能强大的高级语言内置工具。Python 曾被称为“可执行的伪代码”，因为它减少了在其他工具中常见的复杂性。你会发现Python程序的大小常常只有用某些语言（如 C/C++ 和 Java）写的同样程序的几分之一。

它简单易学

这把我们带回这本书的主题：与其他编程语言对比，核心的Python语言惊人地简单易学。事实上你可以在几天内编写出较大的Python代码（可能是几小时，如果你是有经验的程序员的话）。这对那些想学习语言以在工作中应用的专业人员来说是一个好消息。同样，对那些使用Python接口定制或控制系统的最终用户也是一个好消息（注 1）。

实践中的 Python

Python 不只是一个设计得很优秀的程序语言，能实现现实中的各种任务——开发者日复一日所做的事情，而且是应用十分广泛的。它通常用于各种领域，作为编制一些组件的工具，实现独立程序等等。Python的一些主要的作用有助于定义它到底是什么。

注 1：你或许会问，究竟Python培训者能有什么生意呢？首先，学习它的核心语言很有挑战性，在最开始的一段时间会使你很忙。正像我们看到的那样，Python的库集合，也就是它的外围工具（如：Tkinter GUI 以及 Python/C 的集成 API）是 Python 实际编程的一大部分。

系统软件

Python的对操作系统的内置接口，使它成为书写可移植程序、可维护的系统管理工具（也叫Shell脚本）的理想工具。Python上绑定了POSIX，并支持通常的操作系统工具：环境变量、文件、套接字、管道、进程、线程、正则表达式等等。

图形用户接口

Python的简单性和快速开发周期对开发GUI程序也十分适合。在前面提到，它有一个叫Tkinter的与TK API的标准面向对象接口，可以允许Python生成可移植的有友好界面的GUI程序。如果不优先考虑移植的话，你可以在Windows的Pythonwin移植上用MFC类建立GUI，使用Unix上的X系统接口，Macintosh上绑定的工具箱，Linux上的KDE和GNOME接口等。如果要在Web上应用的话，JPython是另一个GUI选择。

组件集成

Python可被C/C++系统扩展和嵌入其他系统的能力，使它可作为一个“粘合”语言描述其他系统和组件的行为。例如，把一个C库集成到Python中，Python可以通过在一个产品中嵌入Python测试、启动它的组件。你用不着重新编译整个产品（或传送给你的客户源代码）就能定制它。Python的COM支持MS-Windows，JPython系统提供了另外一个脚本应用的方法。

快速原型（Rapid prototyping）

对Python程序来说，用C或者用Python写的组件看起来都是一样的。正因如此，我们可以先在Python中初始化原型系统，再移植到可编译语言（如C或C++）中去。与有些原型工具不同，一旦原型固定后Python无需完全重写。系统中不需要C++语言的效率的部分在Python中仍保持不变，这样可以简化维护和使用。

Internet 脚本

Python提供标准的Internet软件模块并允许Python程序通过套接字通信，从发送给服务器方的CGI脚本中提取信息，解析HTML，通过FTP传输文件，处理XML

文件等等。在Python中有不少的"外围"工具处理Internet程序。例如，HTMLGen和Pythondoc系统从Python的基于类的描述中生成HTML文件。上面提到的JPython提供了Python/Java的无缝集成（注2）。

数字编程

NumPy数字扩展程序模块包括诸多高级工具，如数组对象、标准数学库接口等等。因为速度的关系，把Python集成的数字例程是用已编译语言写的。NumPy把Python变成了精密的、但易于使用的数字编程工具。

数据库编程

Python的标准模块提供了一个简单的对象持久系统，允许程序可简单地保存整个Python对象到文件中。应传统数据库要求，Python提供了与Sybase、Oracle、Informix、ODBC等数据库的接口。Python上甚至还有一个可移植的SQL数据库API，可在不同数据库下运行。一个叫*gadfly*的系统实现了针对Python程序的SQL数据库。

其他的：图像处理、人工智能、分布式对象等等

Python应用于很多领域，远比这里提到的更多。一般地说，在实际中很多应用从本质上说是Python的组件集成。通过向Python库中添加编译语言（如C）写的组件，在不同的领域Python都会成为有使用价值的脚本语言。

例如，图像处理就是在一系列由C等编译语言写的组件上实现的。通过Python前端层设置并启动编译的组件，易于使用的Python层与编译语言组件的效率相互补充。由于这样的编程主要是用Python完成的，大多数用户从不需要处理优化组件的复杂性。

注2：我们将在第十章"框架与应用"中详细讨论JPython与其他系统。JPython能把Python程序编译成Java虚拟机代码（这样它们就可以作为客户端脚本运行在任何识别Java的浏览器上），并允许Python程序与Java库通信（例如，在客户端生成AWT GUI）。

Python 的商业产品

更具体地说，Python 已被许多公司应用到真正产生效益的产品上了。这是当前 Python 用户的部分清单：

- Red Hat 公司在它的 Linux 安装工具中使用 Python。
- 微软曾发布过部分用 Python 写的程序。
- Infoseek 使用 Python 作为 Web 搜索产品中的实现与最终用户语言。
- Yahoo! 在大量 Internet 服务中使用 Python。
- NASA（美国国家宇航局）把 Python 用于任务控制系统的实现。
- Lawrence Livermore 国家实验室把 Python 用在大量的数字编程任务中。
- Industrial Light and Magic 公司（译注 1）以及其他公司用 Python 生产标准的商业动画。

有更令人激动的应用我们在这里没提到。哈哈，有的公司使用 Python 却不想让人知道，主要是因为它有竞争优势。访问 Python 站点（http://www.python.org）可以知道更多内容，可以知道当前共有多少公司在使用 Python。

Python 和类似工具的比较

最后，用你已知道的术语说，人们有时拿 Python 和 Perl、Tcl、Java 这样的语言相比较。这些都是已知的有用的工具。我们认为 Python：

- 比 Tcl 强大，可用于大型系统的开发。
- 比 Perl 有更清晰的语法、更简单的设计，这使它更可读、更容易理解。
- 不要与 Java 正面比较，Python 只是一个脚本语言，Java 是一个像 C++ 一样的系统语言。

尤其对不只是用于浏览文本文件的程序而言，因为它可能在将来要被别人读到

译注 1：这可是大名鼎鼎的电脑特技公司。知道它的代表作是什么吗？《侏罗纪公园》。

(也可能是你自己!),我们认为Python比其他脚本语言用起来更划算。本书的两个作者都是正儿八经的Python传教士,我们要说,尽你的所能学习Python吧!

上面是这本书的宣传部分。判断一本书优劣的最好方法是用实际行动。现在我们开始对这门语言进行严格的介绍。在本章的剩余部分,我们要研究运行Python的方法。看一看有用的设置与安装的细节,并向你介绍有关代码持久性的模块文件的概念。再说一遍,我们在这里的目标是让你可以运行本书其他部分的练习和代码。直到第二章"类型和操作符"我们才开始真正的编程工作。在进一步学习之前,要确保你已掌握了启动的细节。

如何运行 Python 程序?

到现在为止,我们所说的Python是指一种编程语言。不过它也是一个称为解释器的软件包。解释器是一种可以执行其他程序的程序。当你写Python程序时,Python解释器读取你的程序,执行它所包含的指令(注3)。在这一节,我们将研究Python解释器运行的方法。

当Python软件安装在你的机器上时,会产生许多组件。按你使用它们的方式,Python解释器可能采取可执行程序的方式,或作为一系列库连接到其他程序上。一般地说,通过Python解释器至少有5种方法可运行程序:

- 交互式
- 作为Python模块文件
- 作为Unix文件型脚本
- 在其他系统中嵌入
- 特定平台的启动方法

让我们依次看一下每种方法。

注3: 严格地说,Python程序首先被编译(如:翻译)成一种中间件的形式(字节码),可以被Python解释器扫描。这个字节码编译步骤是隐藏的、自动的,而且Python比纯的解释器更快。

其他启动 Python 程序的方法

警告：为了使事情更简单，在本章中对使用解释器的描述都很基本，并且强调最低条件的运行程序的方法（例如：命令行，无论Python在哪个平台都是一样的）。在特定平台上运行Python程序的有关内容请看附录二“特定平台的有关信息”。例如：对于MS-Windows的Python移植和苹果机上的用于编辑运行代码的图像接口而言，更多的可能要靠你自己尝试了。

根据你的平台和技术背景，你可能会对集成开发环境感兴趣——一个用于编辑运行、调试的图形界面，可以运行在安装了Tk支持的任何平台上（IDLE是一个使用Tkinter扩展的Python解释器，在第二部分我们会碰到）。你可以在附录一“Python资源”中找到这个描述。Emacs使用者可以在Python Web站点发现在Emacs环境启动Python代码的支持；再说一遍，可以在附录一中找到有关细节。

交互命令行

运行Python程序最简单的方法可能就是在交互式命令行下输入代码了。假设解释器是作为你系统中的可执行程序安装的。在你的操作系统命令行下输入python，不用任何变量，即可启动交互式解释器。例如：

```
% python
>>> print 'Hello world!'
Hello world!
>>> lumberjack = "okay"
>>>                          # 按Ctrl-D退出（在某些平台上是Ctrl-Z）
```

在这里Python是在Unix（或MS-DOS）提示符下开始交互对话的，当Python等待你输入新的语句时，提示符是>>>。交互工作时，语句结果在>>>行后显示。在大多数Unix机器上，两键组合Ctrl-D（按下Ctrl键后不松手再按下D键）将退出交互命令行，返到你的操作系统命令行下。在MS-DOS和Windows中，你可能需要按Ctrl-Z退出。

上面的例子中，我们并没有做太多事：只键入了Python的print语句和一个赋值语句。我们会在以后研究赋值语句的细节。但要注意：我们输入的代码将立即

被解释器执行。例如：在>>>提示符后输入一个print语句，输出内容（一个Python串）就会立刻显示出来。无须先通过编译、连接再运行代码，而你运行C/C++代码时就不得不那样做。

因为代码立刻被执行，交互模式提示符不成为试验这门语言的最方便的地方。我们在这本书中会经常用它来演示小的例子。事实上，这是第一准则：如果你想知道某小段Python代码是如何工作的，在交互命令行下输入并测试它们的输出。就是这样简单。

交互模式通常也用于测试大型系统组件。我们将看到，交互式命令行允许我们交互式导入组件，并迅速测试它们的接口。部分因为这种交互特性，Python支持试验探索性的程序类型，在开始工作时，你会发现这很有用。

注意： 关于提示符：我们直到第三章"基本语句"才会遇到复合多行语句。但作为预览，你应该知道在交互式命令行下输入超出两行的语句时，提示符将变为...而不是>>>。在...提示符下，一个空行（敲回车键）告诉Python你已经输完语句，这与文件中的复合语句不同，在文件中的空白行会被简单地忽略。在第三章你会知道原因。这两个提示符也是可以改变的（在第二部分，我们将看到它们都是在内置sys模块中的属性）。但现在假定在我们的例子中还没有这些东西。

运行模块文件

虽然交互模式对试验和测试来说很有用，但它有一个最大的缺点：Python解释器一执行，你输入的代码就消失了。交互式输入的代码不会保存到一个文件中，不从头输入代码的话你就无法再次运行它。剪切、粘贴或用历史命令可能有些帮助，但作用不大，尤其是在你开始写大型程序时。

为了持久地保持程序代码，你就需要Python模块文件了。模块文件是包含Python语句的简单的文本文件，你可以让Python解释器通过在python命令中列出文件名，来执行这样的文件。作为一个例子，我们启动自己最爱用的文本编辑器（在Unix或Linux上是vi或Emacs，在Windows上可能是记事本，Word等等），并在一个叫*spam.py*的文本文件中输入如下Python语句：

```
import sys
```

```
print sys.argv    # 更多的在后面
```

而且，我们忽略了文件中语句的语法，不去追究细节。我们要注意的一点是，我们是在一个文件中输入代码，而不是在交互提示符下，一旦我们保存了这个文本文件，我们就可以在操作系统shell下，用文件名作为python命令的一个参数运行该文件。

```
% python spam.py -i eggs -o bacon
['spam.py', '-i', 'eggs', '-o', 'bacon']
```

请注意：我们用*spam.py*调用这个模块，也可以简单地以*spam*名调用它。不过因为某些以后我们才会解释的原因，我们要导入给用户的代码文件必须以*.py*结尾。我们也列出了四个被Python程序所用的参数（在python spam.py后面的部分），它们被传给Python程序，可以通过名字sys.argv取得，就像C的参数数组。顺便说一下，如果你在Windows或DOS平台上工作的话，这个例子是一样的，但系统提示符不同：

```
C:\book\tests> python spam.py -i eggs -o bacon
['spam.py','-i','eggs', '-o', 'bacon']
```

运行Unix类型的脚本

现在我们已经看了如何在交互提示符下输入代码，以及如何运行用文本编辑器生成的代码（模块）。如果你打算在Unix、Linux或类Unix系统上使用Python的话，你也可以把Python代码转变为可执行程序，就像你在shell语言如*csh*、*ksh*中的编码一样。这些文件一般称为脚本，用简单的术语说，Unix类型的脚本只不过是一些包含Python语句的文本文件，但它有两个属性：

第一行是特殊的

脚本文件一般第一行以#!字符开头，随后是解释器在机器上的路径。

它们通常有可执行权限

脚本语言通常被标识为可执行的，以告诉操作系统它们可以作为顶层程序运行，在Unix系统上，用chmod +x file.py这样的命令就可以做到这一点。

让我们看个例子，假定在此用自己喜爱的文本编辑器生成一段叫*brian*的Python代码：

```
#! usr local bin Python
print 'The Bright Side of Life...'     # 这是另一个注释
```

我们在文件顶部放入特殊的行来告诉Python解释器的位置。严格地说，第一行是一个Python注释。在Python中的所有的注释都以#开头一直到行尾，这个地方可以为代码的读者插入额外的信息，但这个文件中在第一行的注释是特殊的，系统用它来发现运行程序代码的解释器。

我们可以简单地以*brian*文件名调用这个文件，这里无需用我们前面使用的模块文件的*.py*后缀。添加一个*.py*后缀不会有什么妨碍（可能会帮助我们记起这是一个Python程序文件）；不过既然我们不打算让其他的模块导入这个文件中的代码，这个文件名就是不合适的。如果我们用shell命令`chmod -x brain`给我们的文件以可执行权，我们就可以在操作系统shell下像二进制程序一样运行它：

```
% brian
The Bright Side of Life...
```

Windows和MS-DOS用户请注意：在这里描述的方法是一种Unix技巧，可能不会在你的平台上正常运行，不用担心，只需使用上一节提到的模块文件技巧，在Python命令行下列出文件名即可：

```
C:\book\tests> python brian
The Bright Side of Life...
```

在这个例子中，你不需要在第一行的的特殊注释（虽然即使存在的话Python也会忽略它），文件也无需给以可执行权限。事实上，你如果想在Unix、MS-Windows之间运行可移植的文件的话，用模块文件启动程序而不是Unix类型的脚本，这会变得更简单。

注意： 在某些系统上，你可以通过写`#!/usr/bin/env python`这样特殊的一行注释，避免硬性指定Python解释器的路径。用这种方式编码，Python env程序将按系统查找路径的设定定位解释器（例如：在大多数的Unix shell中，通过查找在path环境变量中列出的所有目录）。既然你不常在你的脚本的第一行硬性指定一个Python安装路径，这种基于env的设计可以更加有可移植性。假设对系统中的任何地方都有访问权，当Python在你的系统中时，你运行脚本就应该没什么问题。

嵌入的代码和对象

我们已经讲述了如何交互运行代码，如何启动模块文件和Unix类型的脚本文件。这几乎包括了我们将在这本书中看到的全部内容。但在某些特殊领域Python代码也可能在一个系统内运行。在这种情况下，我们说Python程序是嵌入的（例如：被另一个程序运行），Python代码自己可被存入一个文本文件，存入数据库中，从HTML页面中取回等等。但是从操作性的角度看，是另一个系统而不是你，让Python运行生成的代码。例如，在C程序中通过调用Python的运行时API（Python在你的系统上编译后生成的库所输出的一组服务），生成并运行Python的字符串是可能的：

```
#include <Python.h>
. . .
Py_Initialize();
PyRun_SimpleString("x= brave + sir + robin");
```

在这一小段代码中，一段C语言程序（somefile.c）通过链接库而嵌入了Python解释器，并传递了一个Python赋值语句用以执行。C程序也可以得到对Python对象的访问，用其他的Python API工具处理或执行它。

这本书不是讨论Python/C集成的，我们不会在这里讨论真会发生的细节内容（注4）。你应该知道，你可不可以真正启动你的Python程序，取决于你如何组织Python。不管怎样，你可以使用这里描述的交互的基于文件的启动技巧，去测试那些偶尔包含在其他程序中的代码。

特定平台启动方法

最后，根据你所使用的计算机类型，可能有比我们上面列出的更多的特定方法启动Python程序。你可以从类Unix的命令行界面运行代码，或在Python程序上双击图标运行它，在苹果机的移植版本上，你可以将程序图标拖动到移植程序的图标上，以执行你的程序。在这本书的附录中我们将讨论特定平台有关的细节问题。

注4：　参考《Programming Python》一书可获得在C/C++中嵌入Python更多的细节。

输入什么，在哪里输入

有这么多的选项和命令，初学者很容易疑惑，究竟哪个命令在哪个提示符下输入。下面是一个快速总结：

启动交互式 Python

Python 解释器经常从系统命令行启动：

```
% python
```

交互式输入代码

程序可以在 Python 交互解释器命令行下输入：

```
>>> print X
```

把代码输入到文件以备以后使用

程序可以输入到文本文件中，用你最喜爱的文本编辑器：

```
print X
```

启动脚本文件

Unix 类型的脚本文件从系统 shell 下启动：

```
% brian
```

启动程序（模块）文件模块文件

从系统 shell 下运行：

```
% python spam.py
```

运行嵌入式代码

当 Python 程序是嵌入的时候，Python 代码可以用任意方式输入。

注意：输入 Python 程序时（交互式模式或在文本文件中），要保证所有的第一行是以非嵌套语句开始的，否则的话，Python 会显示“语法错（SyntaxError）”信息。直到第三章的中间部分，所有的语句都是非嵌套的，所以现在这已经够用了。我们会在以后解释原因（不得不面对的 Python 的缩进原则）。在介绍 Python 的类时这似乎是一种冲突。

初览模块文件

在本章的前面的部分，我们看到了在操作系统命令行下如何运行模块文件（如：

包含Python语句的文本文件）。其结果是我们可以通过导入或重载，也可以从Python解释命令行下运行模块文件，像我们从其他系统组件正常做的那样。这个处理细节将在第五章“模块”中讲述。不过由于它对保存、运行例子很方便，我们将对这个过程做个快速介绍。

导入模块的背后的基本思想是，导入者可以访问在模块顶层赋值的名字。模块输出的服务通常赋值给这些名字。举个例子，假如我们用最爱用的文本编辑器生成一个只有一行的Python模块文件*myfile.py*，如下面的小段代码所示，这可能是世界上最短的Python代码之一，但它足以描述模块的基本用法：

```
title = "The Meaning of Life"
```

请注意该文件有一个*.py*后缀。当从其他组件中导入模块时就需要这个命名约定了。现在我们用两种方法从其他组件中访问这个模块的变量title，用一个import语句整体的导入模块，通过我们要访问的变量名限定模块：

```
% python                          # 启动Python
>>> import myfile                 # 运行文件，作为整体载入模块
>>> print myfile.title            # 用它的名字：已限定
The Meaning of Life
```

或用from语句从一个模块中取到名字（实际上是拷贝）：

```
% python                          # 启动Python
>>> from myfile import title      # 运行文件，载入它的名字
>>> print title                   # 直接使用名字：无需限定
The Meaning of Life
```

我们将看到，from和import非常相似，在导入组件过程中，跟随了一个对名字的额外赋值。注意到两个语句中列出的模块都是简单的*myfile*，没有*.py*后缀。当Python寻找真正的文件时，它知道应该包括后缀。

无论我们用import还是from，模块*myfile.py*中的语句都会被执行。导入组件（在这里是交互提示符）获得了在文件顶层定义的名字——title变量，赋值了一个字符串。但当我们开始定义对象如函数和类时这些概念就会特别有用了。这样的对象成为可以被一个或多个客户模块用名字访问的服务。

在一次会话中导入模块文件时，Python在内部从头到尾执行所有代码。正因为如此，交互式导入模块是立刻执行代码的另一种方法（而不是从一个系统shell下以python myfile.py的形式运行）。但这个过程应注意到这一点：Python只在它导入的时候执行一次代码，如果你在交互对话中再导入它，Python不会重新执行文件的代码。即使你已经在编辑器中改变了它的代码。要真正地重新运行一个文件的代码而不用停止、重新启动交互式解释器，你可以使用Python的reload函数，如下：

```
% python                          # 启动Python
>>> import myfile                 # 运行/载入模块
>>> print myfile.title            # 限定取得名字
The Meaning of Life

在你的编辑器中改动myfile.py

>>> import myfile                 # 不会返回文件代码
>>> reload(myfile)                # 会返回文件（当前的）代码
```

用这种方案工作时，reload稍有些复杂。我们现在建议你不要这样做（文件改变时先退出解释器，再重新进入解释器）。另一方面，这是一种测试Python类时很流行的技巧，你自己决定好了。

初览名称空间检测

另一个流行的技巧是用内置函数跟踪程序交互运行时的名字，在后面我们将更多的说说这一点。但在你做某些练习前，我们将作一个简短的介绍，如果你不用任何参数调用dir函数，可以返回一个Python列表（第二章中的内容），该列表包含在交互环境的名字空间内定义的所有名字：

```
>>> x = 1
>>> y = "hrubbery"
>>> dir()
['__builtins__', '__doc__', '__name__', 'x', 'y']
```

在这里，表达式dir()是一个函数调用：它请求Python运行叫dir的函数。我们会在第四章“函数”中遇到。但现在，记住你需要在函数名后添加圆括号来调用它（无论它是否有参数）。

当dir函数被调用时，返回的有些名字是你可以“自由”得到的：它们都是内置名字，总是被Python预定义好的。例如，__name__为模块的文件名，__builtins__是一个包含了Python中所有内置名字的模块（包括dir）。其他的名字是你定义的变量（在这里是x和y）。如果你调用dir并用一个模块作为参数，可以得到在那个模块中定义的名字（注5）：

```
% cat threenames.py
a = 'dead'
b = 'parrot'
c = 'sketch'
% python
>>> import threenames
>>> dir(threenames)
['__builtins__', '__doc__', '__file__', '__name__', 'a', 'b', 'c']
>>> dir(__builtins__)
Python将为你预定义所有的名字
```

在以后，我们将看到有些对象有其他方式，可以告诉客户它们所提供的名字（例如__methods__和__members__这样的特殊属性）。但现在，dir函数让你可以完成你可能关心的事情。

Python设置细节

到现在，我们已经知道如何使Python程序执行我们输入的程序。但除了解释器，Python也安装了一系列的程序软件，存储在Python的源库中。而且，Python解释器可识别系统shell变量中的设定。这可以让我们定制解释器的行为（例如：在哪里可以找到源代码文件）。这一节讨论Python程序员通常使用的环境设定。看一下安装细节，并实现一个脚本，来描述你需要知道的大多数步骤。如果你已经可以运行准备就绪的Python的话，可以跳过这一节的大多数内容，或以后再看也不迟。

注5：严格地说，是在模块的名字空间内——一个我们很快将频繁用到的术语，你很快就会听烦的。既然我们一直保持用技术术语，交互命令行实际上也是一个模块，叫做__main__；你输入的代码就仿佛是在一个模块文件中那样工作，不同的是表达式结果被打印给你。注意dir调用的结果是一个列表，它可以被一个Python程序处理。不管怎样，要这样想：名字空间也可以通过其他方式被取得。

环境变量

Python的解释器识别一些环境变量的设定，但只有少数几个经常使用的、重要的需要解释。表 1-1 总结了主要的 Python 变量。

表 1-1　重要的环境变量

作用	变量
系统 shell 查找路径（用于发现“python”）	PATH（或path）
Python 模块查找路径（用于导入）	PYTHONPATH
Python 交互式启动文件路径	PYTHONSTARTUP
GUI 扩展变量 (Tkinter)	TCL_LIBRARY, TK_LIBRARY

这些变量可以直接使用，但有几点要注意：

- PATH列出了一系列操作系统查找可执行程序的路径。正常情况下应包括你的Python解释器所在的路径(Unix上的python程序或Windows上的*Python.exe*程序)。在 Macintosh 上你不需要设定（安装过程处理了路径的细节）。
- PYTHONPATH设定的功能类似于 PATH：当你在一个程序中导入一个模块时，解释器由 PYTHONPATH变量定位模块文件。这个变量的设定用于查找在运行时导入的模块。你会经常想让它包含你自己的源代码目录和Python源库目录(除非已在你安装时实现了)。
- 如果 PYTHONSTARTUP设定了一个 Python 代码的路径名，无论你何时启动交互式解释器，Python都会自动执行这个文件，就像你已在交互命令行下输入的一样，这是一个便利的方法，交互式工作时总能载入程序。
- 倘若你想用 Tkinter GUI 扩展程序的话（参见第八章“框架和应用”），在表 1-1 中的两个GUI变量应设定为Tcl、Tk 系统的源库路径名（与 PYTHONPATH 特别像）。

不幸的是，设定这些变量的方式和将它们设定成什么，还依赖与你的系统设置。例如，在Unix 系统上，设定变量的方法取决于 shell 的设置；在 *csh* shell 下，你可能要这样设定 Python 模块查找路径：

```
setenv PYTHONPATH .:/usr/local/lib/Python:/usr/local/lib/Python/tkinter
```

它告诉Python到三个目录中查找导入模块：当前目录（.），机器安装Python源库的目录（在这里为*/usr/local/lib/python*），和tkinter源库子目录即Python扩展程序提供源码的地方。但如果你使用ksh shell，设定可能会这样：

```
export PYTHONPATH=".:/usr/local/lib/python:/usr/local/lib/python/tkinter"
```

如果你用MS-DOS，这个环境变量会大不相同：

```
set PYTHONPATH=.;c:\python\lib;c:\python\lib\tkinter
```

既然这不是一本关于操作系统shell的书，我们宁愿少讨论一些关于源文件的细节，从你的系统shell手册中或其他文档可以找到有关详细内容。若你对必须设定什么拿不准的话，向你的系统管理员（或身边的高手）寻求帮助。

一个启动脚本例子

下面这段叫*runpy*的代码，把这些细节都收集到一个简单的Python启动脚本中。它将某些重要的环境变量设置为合理的值（至少在作者的机器上是这样），并启动了Python的交互解释器。在你的系统shell提示符下输入runpy就可以运行它：

```
#!/bin/csh
#给这个文件可执行权（chmod +x runpy）
#把这个放到你的.cshrc文件中

# 1）添加路径到命令行解释器
set path = (/usr/local/bin $path)

# 2）设定Python库查找路径（除非预定义了）
# 添加你的模块文件路径到列表中
setenv PYTHONPATH \
    .:/usr/local/lib/Python:/usr/local/lib/Python/tkinter

# 3）设定tk库查找路径用于GUIs（除非已经预定义了）
setenv TCL_LIBRARY /usr/local/lib/tcl8.0
setenv TK_LIBRARY  /usr/local/lib/tk8.0

# 4）启动交互式命令行
Python
```

runpy 描述了一个典型的 Python 设置，但它有一些不足：

- 它只是为 *csh* shell 写的，一个在 Unix 和 Linux 平台上命令行上的普通的处理程序，如果你不是 *csh* 用户的话，你将需要额外的字段。
- 它描述的设置通常在你的 shell 启动文件中运行一次（*csh* 用户的 *~/.cshrc*），而不是你每次运行 Python 时都需要运行它。
- 与 Python 的建立方式有关，既然标准的源库路径可能被硬编码到你的安装中，你不需要列出标准的源库设置路径

MS-DOS 用户应注意的一点：可以用 MS-DOS 的批处理文件生成一个相似的设置。取决于你安了哪一个 Windows 的移植，这个文件看起来类似这样：

```
PATH c: python;%PATH%
set PYTHONPATH=.;c:\python\lib;c:\python\lib tkinter
set TCL_LIBRARY=c:\Program Files\Tcl\lib\tcl8.0
set TK_LIBRARY=c:\Program Files\Tcl\lib\tk8.0
Python
```

一个 GUI 对话测试

如果你或你的管理员已安装了 Tkinter GUI 扩展程序的话，下面的对话展示了一种测试 Python/GUI 设置的方法。（若你不想使用 Tkinter 的话可以跳过这一节。）

```
% runpy
版本/版权信息...
>>> from Tkinter import *
>>> w = Button(text="Hello", command='exit')
>>> w.pack()
>>> w.mainloop()
```

在系统 shell 不输入 runpy，随后在 Python 的 >>> 提示符不显示所有代码。现在忽略实例代码中的细节，我们将在第十章学习 Tkinter，如果所有都设置正确的话，你应该在你的屏幕上看到一个类似图 1-1 的窗口（在 MS-Windows 机器上显示的与在 X-Windows、Mac 上的看起来略微有点不同）。

图 1-1 Tkinter 安装测试屏幕图

如果这个测试没有工作的话，请开始检查一下你的环境变量路径的设定，以及你的 Python 安装。Tkinter 是一个可选的扩展程序，并且必须显性的激活，请确定它在你的 Python 版本中，还要保证你有对 Tcl/Tk 源库的访问权，当前版本中的 Python Tkinter 的实现需要它们。查看 Python 发布文档中的 *README* 文件和 Python 的 Web 站点可以知道更多细节内容。

安装过程纵览

出于对某些组件的兴趣，这一节提供了关于 Python 安装过程的几点内容。当你开始学习 Python 时，正常情况下你不需要关心 Python 安装过程。要是另外有人 —— 可能是你的系统管理员已在你使用的平台上安装了 Python，你可以跳过这里的大多数内容。

但情况不总是这样的，并且即使 Python 已经在你的机器上安装了，随着你的 Python 水平的不断提高，安装过程的有关细节就会变得十分重要了。有的场合，知道如何从源代码建立起 Python 是十分重要的，这样就可以静态的绑定你的扩展程序。但是再说一遍，这不是一本有关 Python/C 集成的书，所以如果 Python 已为你安装了，你可以把这一章放到将来再读。

随二进制或 C 源代码形式提供的 Python

你可以以二进制可执行形式或以源代码形式（在运行前必须先在你的机器上编译）得到 Python。两种形式都可以在各种媒体中找到 —— Python 的 Web/FTP 站点（看附录一），随某些 Python 书籍附带的或独立的光盘，Linux 发行版附带等等。一般情况下，如果你得到的是二进制形式，则它必须与你的机器相兼容；若你用的是源代码的发布，则需要在你的系统上有 C 语言的编译器。这两种形式通常都以压缩文档形式发布的，这意味着你通常要用 `gzip` 或 `tar` 这样的软件来解开机器上的文件。

C 代码的设置/建立是自动的

虽然得到二进制形式的 Python 意味着你无须自己编译，这也意味着你对扩展程序的激活只有很少的控制权；有的扩展程序可能是创建人凑巧认为它重要就加进去的，而你得到的就是这个。除了 Python 的二进制本身，你需要得到并安装 Python 源库，而这些在 Python 的二进制中有没有却是没准的事情。想得到更多的控制，要取得整个 Python 源代码发行版，并在你的机器上编译它。我们不想在这里列出编译命令，但源代码的建立过程是高度自动化的。Python 根据你的平台设置它的 `makefiles`。Python 在你提到的任何平台上编译都没什么问题。

除非你曾用过 C 语言，否则不要从源代码建立 Python

已经说了，即使 C 自动编译一个像 Python 这样大的系统也不是很轻松的事情。如在以前没用过 C 编译器的话，我们建议在从你的机器上用源代码建立你的 Python 之前，首先获得你的二进制包。通常情况下，在安装或建立时你可以去问你身边的 C 语言高手。

现在预编译的 Python 二进制形式在大多数平台上都有，包括 MS-Windows、Macintosh 和大多数的 Unix；参见 Python 的 Web 站点上的链接。我们还应注意到完整的 C 源代码发行版包括了整个的 Python 系统，并且它是真正的自由软件，可以在你的产品中使用它而没有版权的限制。虽然不是每个人都喜欢钻研一个解释器的代码，但在 Python 系统中如果可以控制整个源代码是一件很棒的事情。

想得到更多的安装和编译的细节，可以看随源代码发布的 README 文件或 Python 站点或其他 Python 书籍，如《Programming Python》。大量的 Python 发布的注意事项，可以看附录一所列的网址。

总结

在本章，我们探讨了启动 Python 程序的方法，模块文件的基础知识和名字空间检查，以及 Python 设置和安装细节。希望你已经知道了有关交互启动 Python 解释器的足够知识。在学习表达式和大型程序组件前，我们将在第二章探讨 Python 的基本对象类型。

练习

好了：现在是你自己开始编写一些小段代码的时候了。这一任务十分简单。但有些问题的提示在以后的章节中。记住，查看一下附录三“练习解答”可以找到答案；它们有时包括了一些在本章中没有讨论的补充信息。换句话说，即使你自己可以设法得出答案，你也应该看一下这个附录。

1. **交互。**启动Python命令行，输入表达式"Hello world!"（包括引号），字符串会回显出来。这个练习的目的是获得Python运行的环境变量设置。你可能要在PATH环境变量中添加Python可执行文件路径。在*.cshrc*或*.kshrc*中设定以确保在Unix系统中总能被执行，在Windows上，用一个*setup.bat*或*autoexec.bat*文件。

2. **编程。**用你喜欢的文本编辑器，写一个简单的模块文件——包含一个简单语句：`print 'Hello module world!'`的文件，把这个语句存入文件*module1.py*中，在系统shell命令行下传递给Python解释器执行。

3. **模块。**接下来，启动Python命令行并导入上面练习中你写的模块，你的PYTHONPATH设定包括存储文件的的目录了吗？试着移动文件到一个不同的目录中并再次导入它，发生了什么？（提示：有一个*module1.pyc*文件在原来的目录中吗？）

4. **脚本。**如果平台支持的话，添加一个#!行到*module1.py*的顶部，给这个文件可执行权，直接作为可执行程序运行它，第一行应包括什么？

5. **错误。**在Python命令行试验输入数学表达式和赋值。首先输入表达式`1 / 0`，发生了什么？接下来，输入一个还没定义的变量名，这次发生了什么？你可能还不知道，不过你正做着“异常”的过程，我们在第七章将讨论该主题。在第八章“内置工具”中我们还将看到Python源代码除错工具，pdb，如果你不能等那么长时间，跳到那一章，或是其他的Python文档资源。Python的默认出错信息可能和你开始学习Python的时候需要的错误处理一样多。

6. **中断。**在Python命令行下，输入：

```
L = [1, 2]
L.append(L)
L
```

发生了什么？如果你正用的版本比 1.5.1 版新，Ctrl-C 组合键可能在大多数平台上都有用，你认为这个发生的原因是什么？**警告**：如果你用的版本比 1.5.1还旧的话，在运行这个测试时，请保证你的机器可以用中断键组合停止一个程序，否则你可能要等很长时间了。

第二章

类型与操作符

本章内容：

- Python 程序结构
- 为什么要使用内置类型？
- 数字
- 字符串
- 列表
- 字典
- 元组
- 文件
- 共有的对象属性
- 内置类型的常见问题
- 总结
- 练习

这一章将开始我们的 Python 语言之旅。从抽象的角度看，我们在 Python 里写的程序都是通过一些“原料”来行使功能的（注 1）。程序采用我们在后面会遇到的语句（statement）的形式。这里，我们对这些程序处理的“原料”感兴趣。在 Python 中，它们总是以对象（object）的形式出现，可能是 Python 提供给我们的各种内置对象，也可能是我们用 Python 或 C 语言的工具生成的对象，两种方式都允许我们对 Python 对象行使功能。

当然，Python 的开发远远不止于处理“原料”。但既然 Python 程序的主体是 Python 编程中最基本的概念，我们还是从对 Python 内置对象的简介开始吧。

Python 程序结构

沿着我们介绍的路线，首先让我们对所学习的这一章有一个清晰、总体的概念。从更具体的角度看，Python 可以分成模块（module）、语句和对象。关系如下：

注 1：　原谅我们的拘泥：我们是计算机科学家。

1. 程序是由模块组成的。
2. 模块中包含语句。
3. 语句生成并处理对象。

为什么要使用内置类型？

如果用过C或C++这样的低级语言，你就会知道大多数的工作都集中在对象的实现上（也就是有些人所说的数据结构）以表示自己应用领域内的组件。你需要设计内存结构，管理内存分配，实现查找和访问例程等等。这些杂事听起来就很烦人（而且还容易出错），并经常会令你偏离真正的编程目标。

在典型的Python程序中，大多数这样令人厌烦的事都没有了。因为Python提供了功能强大的对象类型，作为语言固有的一部分，在你开始解决问题之前不需要为对象的实现书写代码。事实上，除非你需要内置类型不提供的特殊处理，否则，内置对象总比你实现的要好用。这里有几点原因：

内置对象使简单程序写起来更容易

对于简单的任务，要表示问题的结构，只要用内置类型就行了。因为我们可以免费的获得集合（列表）和查找表（字典）这样的东西，并且可以立刻使用它们。仅用内置对象类型就能做很多工作。

Python提供对象并支持扩展程序

在某些方面，Python不但借鉴了某些依赖内置工具的语言（如LISP）的长处，也吸收了那些依赖程序员自己提供工具实现或框架的语言（如：C++）中的东西。尽管你可以在Python中实现唯一的对象类型，但在开始时你无需这样做。而且，因为Python的内置类型都是标准的，它们都是一样的，框架则因环境不同而趋于不同。

内置对象是扩展程序的组件

为了复杂的任务，你可能仍需使用Python语句和C语言的接口，来提供自己的对象。但在后面的章节我们看到，手工实现的对象通常建立在列表和字典这样的内置类型之上。例如，一个栈数据结构可以用一个管理内置列表的类来实现。

内置对象总比定制的数据结构类型更加有效

Python的内置类型实现了已在C中实现的优化数据结构算法。尽管为了追求速度你可以自己写相似的对象结构，但想要达到内置对象提供的那么好的性能，通常是很难的。

换句话说，内置类型不仅使程序更简单，而且也比从头创建的类型更加强大、有效。不管是否实现新的对象类型，每个 Python 程序的核心都有内置对象。

表 2-1 列出了这一章用到的内置对象类型。如果你已用过其他语言，有的对象你可能已很熟悉了（如数字、字符串和文件），但其他的更通用、更强大。例如，你会发现列表和字典可以使你不用把大部分精力放在低级语言中必须支持的集合和查找上。

表 2-1 内置对象预览

对象类型	常量示例 / 用法
Number（数字）	`3.1415, 1234, 999L, 3+4j`
String（字符串）	`'spam', "guido's"`
List（列表）	`[1, [2, 'three'], 4]`
Dictionary（字典）	`{'food':'spam', 'taste':'yum'}`
Tuple（元组）	`(1,'spam', 4, 'U')`
File（文件）	`text = open('eggs','r').read()`

数字

为了直入正题，我们的 Python 之旅要学习的第一个对象类型是数字（number）。如果你用过其他语言的话，你会发现，Python的数字类型和其他语言中的很类似。Python 支持常见的数字类型（整数、浮点数），常量和表达式。另外 Python 提供更高级的数字类型支持，包括复数类型，无限精确整数，大量的数字工具库。不面几节将为你提供 Python 中对数字支持的概貌。

标准的数字类型

Python支持常见的数字类型：不但有整数和浮点数，也包括所有相关联的语法和操作。像C语言一样，Python允许你用十六进制和八进制书写整数。与C不同的是，Python还有一个复数类型（Python 1.4所引入的）。无限精确长整数可扩展到你的内存所允许的任意位。表2-2显示了数字在Python程序里的写法（以常量为例）。

表2-2 数字常量

常量	解释
1234, -24, 0	正常整数（C的长整型）
99999999999L	长整数（无限大小）
1.23, 3.14e-10, 4E210, 4.0e+210	浮点数（C的双整型）
0177, 0x9ff	八进制和十六进制常量
3+4j, 3.0+4.0j, 3J	复数常量

一般来说，Python的数字类型较为简单，但有几个要点值得注意：

整数和浮点常量

整数以10进制字符串写出，浮点数带一小数点，和/或一个可选的幂标志：e或E。若你写了一个带10进位的小数点或幂的数，在一个表达式中使用时，Python将其作为浮点对象计算（不是整数），浮点数的规则和C中是一样的。

数字的精确度

简单的Python整数（表中第一行）在内部都是作为C长整数实现的（例如，至少32位）；Python的浮点数是作为C语言中的双整数型实现的；Python的数字型与C在编译Python解释器的时候长整型和双整型的精确度是一样的。另一方面，若一个整数常量以l或L结尾，它就成为Python的长整型数，并根据需要而增长（不要与C的长整型弄混了）。

十六进制和八进制常量

书写十六进制和八进制整数的规则与C语言是一样的：八进制常量以一个0开头，十六进制则以0x或0X开头。注意，这意味着你不能写一个以0开头的正常的十进制数（如：01）；Python将其解释为八进制，这通常不会像你期待的那样正常工作。

复数

Python的复数常量由“实部+虚部”组成，以一个J或j结束。从本质上讲是用一对浮点数实现的，但对于复数，所有的数字操作都施行复数运算。

内置工具和扩展程序

除了表2-2列出的内置数字类型外，Python还提供了一系列用于处理数字和对象的工具：

表达式操作符

+，*，>>，**等等。

内置的数学函数

pow，abs等等。

工具模块

rand，math等等。

在我们的学习过程中会遇到所有这些内容。最后，如果你需要做重要的数字分析的话，Python有一个叫*Numeric Python*的可选扩展程序，提供了高级的数字编程工具，例如矩阵数据类型的精密运算库。正因为它属于高级主题，我们不会在这一章对它谈论太多，请参考书后面的例子和附录一“Python资源”。还要注意，如这里所说的，Numeric Python是一个可选的扩展程序，它不是随Python一起安装的，必须单独安装。

Python表达式操作

处理数字的最基本工具可能就是表达式了，它将数字（或其他对象）和操作符组合起来，当Python执行时它算出一个结果值。在Python中表达式通常由数学记号和操作符写成。例如，X与Y相加，我们可以写成X+Y，告诉Python在值X和Y间应用+操作符，结果是X和Y的和——另一个数字对象。

表2-3列出了Python中允许的操作符表达式，很多都是无需解释的。例如，通常支持的数学操作：+，-，*，/等等。如果你用过其他语言的话，有一些你可能会很熟悉：%计算余数，<<执行左移位，&进行按位与运算等等。其他的都是Python

特有的，有些不是真正的数字操作，is操作符测试对象身份（identity，即地址）是否相同，lambda生成匿名函数等等。

表2-3 Python的表达式操作和预处理

操作	描述
x or y	逻辑或（只有x为假，y才被计算）
lambda args: expression	匿名函数
x and y	逻辑与（只有x为真的时候，y才被计算）
not x	逻辑反
<, <=, >, >=, ==, <>, !=,	比较操作
is, is not,	身份测试
in, not in	序列成员关系测试
x | y	位或
x ^ y	位异或
x & y	位与
x << y, x >> y	把x向左或右移动y位
x + y, x - y	相加/合并，相减
x * y, x / y, x % y	乘/重复，除，余数/格式
-x, +x, ~x	一元取反，一致，位取补
x[i], x[i:j], x.y, x(...)	索引，分片，限定，函数调用
(...), [...], {...}, `...`	元组，列表，字典，转化为字符串

表2-3包括了大多数可供参考的东西。既然以后我们会在学习中看到这些操作符，在此就不把每一个都加以描述了。但有下面几点应注意。

混合操作符，表中越往下的结合性更强

像大多数语言一样，更加复杂的表达式都是由表中操作符表达式串在一起组成的，例如，两个乘积的和可以写作A*B+C*D。可是Python如何知道先用哪一个操作符呢？当你用一个以上的操作符写一个表达式的时候，Python根据优先级规则将它划分为几个部分，并且这个划分决定了计算的先后顺序。例如，对于X+Y*Z，Python先算乘法（Y*Z），随后加X，因为*比+有更高的优先级（在表中的位置越低优先级越高）。

用括号将表达式分组

如果上面这些听起来有些令人迷惑，放松一下：如果使用括号将表达式分组的话，你就可以完全忘掉优先级。当把表达式括起来的时候，你可以不用管Python优先级规则；Python总是先计算括号里表达式。例如：你可以不写成 X + Y * Z，而写成(X + Y) * Z或是X + (Y * Z)，以强制Python按你想要的顺序来计算表达式。对于前一个表达式，首先在X和Y上进行+运算，而对于后一个表达式，*先执行（跟没有括号时一样）。一般说来，在一个表达式中添加括号总是个不错的主意；不仅可以强制按你想的顺序计算，还增加了可读性。

混合类型：像C一样转化

除了在表达式中混合操作符之外，你也可以混合数字类型，例如你可以将一个整数与一个浮点数相加。但这将导致另一个困境：结果是什么类型呢——整数还是浮点数?答案是简单的，尤其是如果你以前用过其他语言的话：在混合类型的表达式上，Python先将操作对象转化为最复杂的操作类型，然后再运行同种类型的数学运算。Python的数字类型复杂程度如下:整数比长整数简单，长整数比浮点简单，浮点数比复数简单。所以当一个整数与一个浮点数混合计算时，首先是整数转化为浮点数，在浮点数与浮点数间计算结果。类似的，任何含有复数的混合表达式都要把其他的类型转为复数，再进行复数运算。

预览：操作符重载

虽然我们现在把注意力集中在内置数字上，但要记住，所有的Python操作符都可以被Python的类或C扩展类型进行重载，并在你实现的对象上工作。例如在后面你将看到用类编码的对象可以添加上+表达式，用[i]表达式索引等等。而且，有的操作符已被Python自身重载了，它们依据所处理的内置对象类型的不同而执行不同的指令。例如，+操作符应用在数字时是加号，而（一会儿我们将看到）应用在顺序对象如字符串和列表时则是合并（*concatention*）操作.(注2)。

注2：这通常称之为多态——指一种依赖于被操作对象类型的操作。不过我们对于像这样的面向对象的概念还没有足够的认识准备，所以现在只要这样认为就可以了。

数字在实际应用中

理解数字对象和表达式最好的方法是看实际的例子。让我们启动交互式命令行，并输入些基本的、但有代表性的操作符。

基本操作

首先练习基本的计算：加和除。在下面的交互中，首先把两个变量（a 和 b）赋值为整数。在Python中，变量是在赋值的同时生成的，不用在使用前预先声明变量名。换句话说，赋值会使变量自动存在：

```
% python
>>> a = 3                # 名字生成
>>> b = 4
```

在这里，我们还用了一个注释。这已经在第一章“开始”中介绍了，但作为Python编程新手，你应该知道，以#标记开始直到行尾的文本都被认为是一个注释，会被Python忽略。（这是你在代码中书写可读性文档的地方，既然在交互提示符下你输入的文档都是临时性的，正常情况下你不用在此书写注释。我们只是为了帮助理解代码，在我们的例子中添加了些注释）。现在在表达式中使用整数对象，变量被它们的值替代，表达式结果在交互提示符下回显给我们：

```
>>> b / 2 + a        # 等同于(1 + 2) - 3
5
>>> b / (2.0 + a)    # 等同于(1 / (2.0 + a))
0.8
```

第一个表达式中没有括号，所以Python根据它的优先级规则自动进行分组计算。既然在表2-3中/比+号位置低，它的结合性更强，所以首先运算。结果同我们右边注释中括号显示的表达式一样。还要注意到第一个表达式中所有数都是整数，正因如此，Python进行整数除和加运算。

第二个表达式中，在加号的部分我们添加了括号，强制Python先运行它（先于/运算）。我们也有一个浮点数的操作——通过添加一个小数点。因为是混合类型，在运行+之前，Python把整数转化为浮点数(3.0)，也把b转化为浮点的数值

(4.0)，运行浮点除法(4.0/5.0)得到一个浮点数结果0.8。相反，如果这是一个整数除法的话，结果将是一个被截断的整数。

位操作

除了正常的数字运算（加、减等等），Python支持C语言中所允许的大多数数字表达式。例如，下面是位移运算和布尔运算：

```
>>> x = 1          # 0001
>>> x << 2         # 左移两位：0100
4
>>> x | 2          # 位或：0011
3
>>> x & 1          # 位与：0001
1
```

在第一个表达式中，一个二进制数1（二进位为0001），左移2位生成二进制数4。最后两个运算执行了二进制的或（or）运算（0001|0010=0011），和二进制的与(and）运算（0001&0001=0001）。我们不在这里讨论关于位运算的太多细节，但是如果你需要的话Python是支持的。你还应该知道在高级语言中它通常不那么重要，不像低级语言中（如C）那样。如果你想了解Python中的运算，你应该好好回忆一下曾用过的语言，一般的说，在Python中总有比位串运算更好的编码方式(注3)。

长整数

现在说些更少见的东西：在实践中看一下长整型数。当一个整数常量以一个L（或小写的l）结尾时，Python生成一个长整数，该数可以任意大：

```
>>> 99999999999999999999999999999 + 1
OverflowError: integer literal too large
>>> 99999999999999999999999999999L + 1
100000000000000000000000000000L
```

注3：一般情况是这样。每一种规则总有例外。例如，如果你与C库接口希望位字符串可以被传进去，我们所讲的就不适用了。

在这里第一个表达式失败并出错，因为正常的整数不能容下这么大的数。相反，第二个工作正常，因为我们让 Python 生成的是一个长整数。

注意： 长整数是个方便的工具。事实上，如果需要的话，你可以以分为单位计算国债。但为了支持附加的精度，Python必须做额外工作，因此长整数计算比正常整数要慢得多。这总是没错的，天下没有免费的午餐。

复数

复数是最近加到Python中的，如果你知道什么是复数，你就知道为什么说它们是有用的。如果不是这样的话，可以认为这一节是选读内容（注4）。复数是作为两个浮点数实现的——实部和虚部，或虚部添加一个j或者J后缀。我们可以通过一个+号书写非0的实数部分。例如，复数实部2，虚部-3，可写作2+-3j。一些复数运算的例子：

```
>>> 1j * 1J
(-1+0j)
>>> 2 + 1j * 3
(2+3j)
>>> (2+1j)*3
(6+3j)
```

复数也允许我们提取它们的实部或虚部作为属性，但既然复数运算是一个高级工具，查看网上可获取的Python语言参考手册可得到更多细节。

其他的数字工具

除了上面提到的，Python也提供了内置函数和内置模块用于数字处理。下面有些内置的数学模块和几个内置的数学函数。我们将在第八章“内置工具”中遇到更多的内置函数与模块。

```
>>> import math
```

注4： 作者之一立刻指出在他将近15年的开发工作中从未需要使用到复数，另一个作者就不那么幸运了

```
>>> math.pi
3.14159265359
>>>
>>> abs(-42), 2**4, pow(2, 4)
(42, 16, 16)
```

请注意使用math这样的内置模块时，必须先导入（import）并加以限定，但abs这样的内置函数却无需导入即可使用。实际上，模块是外部组件，但内置函数却包含在名字空间中。在名字空间中，Python会查寻并找到你在程序中使用的名字。这个名字空间与__builtin__模块相对应。在第四章“函数”中我们将讨论名字解析，现在，只要我们说到“模块”，就要想到“导入”。

字符串

下一个主要的内置类型是Python的字符串（*string*）——一个有序的字符集合，用于存储、表示基于文本的信息。从功能的角度上看，字符串可以被用于实现任何可以作为文本编码的东西：符号和单词（如你的名字），载入内存的文本文件的内容等等。

你可能已在其他语言中用过字符串了；Python的字符串同C语言中的字符数组的作用是一样的，但Python的字符串是一个高级的工具。与C语言不同，Python中没有char类型，只有单字符字符串。而且严格的说，Python字符串是不可变序列（*immutable sequence*）——这个长词的意思是，只有普通的顺序操作，而不能在原位（in-place）改变。事实上，字符串表示的是被称为序列的对象的较大的类，一会儿我们将讨论序列的含义。但请注意这里介绍的操作，因为在我们以后看到的类型上这些操作也一样工作。

表2-4介绍了普通的字符串常量和操作。字符串支持*concatenation*（合并）这样的表达式操作，以及*slice*（分片），*index*（索引）等等。Python还提供了一套工具模块用于处理导入的字符串。例如string模块可导出大多数标准C库的字符串处理工具，regex和re模块为字符串添加了正则表达式匹配（所有这些将在第八章中讨论）。

表 2-4 普通字符串常量和操作

操作	解释
s1 = ''	空字符串
s2 = "spam's"	双引号
block = """.."""	三引号块
s1 + s2,	合并
s2 * 3	重复
s2[i],	索引
s2[i:j],	分片
len(s2)	长度
"a %s parrot" % 'dead'	字符串格式
for x in s2,	迭代
'm' in s2	成员关系

空字符串可以写成中间为空的两个引号。注意字符串常量可以用单引号或双引号包围；两种方式都同样工作，但它们都允许一个引号字符出现在字符串内部而无需用反斜线（许多关于反斜线的内容在后面讲述）转义。表中第三行也提到了一种三引号的形式；当字符串被三个引号围起时，它们可以跨过多行。Python将所有三引号文本收集到一个多行字符串中，该字符串带有嵌入的换行符。

实际操作中的字符串

如果愿意立刻得到更多的细节，我们可以再一次与Python解释器对话，并描述一下表 2-4 中的操作。

基本操作

字符串可以用 + 操作符加以合并，用 * 操作符加以重复。正式的说法是，两个字符串对象相加生成一个新的字符串对象，该新字符串包含操作符连接后的内容。重复这一操作特别像字符串自我添加很多相同的项。两个例子中，Python允许你

生成任意大小的字符串。无需在Python中预声明什么，如数据结构的大小（注5）。Python提供了一个len内置函数用以返回字符串（和其他有长度的对象）的长度：

```
% python
>>> len('abc')          # 长度：数字项
3
>>> 'abc'+ 'def'        # 合并：一个新的字符串
'abcdef'
>>> 'Ni!'* 4            # 同"Ni!"+ "Ni!"+ ...
'Ni!Ni!Ni!Ni!'
```

注意在这里操作符重载已经发挥作用了：我们在查看数字的时候，使用了称之为加和乘的同样的操作符。Python非常智能化，足以纠正操作，因为它知道加和乘的对象的类型，但要小心，Python不允许你在+和*表达式中混合数字和字符串，'abc'+ 9会出错，而不是自动将9转化为字符串。如表2-4中最后一行所示，你可以在循环中用for语句迭代字符串，并用in操作符测试成员关系：

```
>>> myjob = "hacker"
>>> for c in myjob: print c,          # 遍历项
...
h a c k e r
>>> "k" in myjob                      # 1代表真
1
```

但是要真正理解for和in，你还要了解Python中语句和“真”所代表的含义，还是在以后再谈论这个例子的细节吧！

索引和分片

因为字符串的定义是有序的字符集合，你可以通过位置访问它们的内容。在Python中，字符串中的字符是通过索引取得的——在字符串后面的方括号中提

注5：与C语言中的字符数组不同，使用Python字符串的时候，你不需要分配或管理存储数组。简单地生成需要的字符串对象，让Python在底层内存空间管理就可以了。从本质上说，Python自动回收未使用的对象内存空间，使用一个引用计数进行“垃圾收集”的策略。每一个对象都会跟踪引用它的名字、数据结构的数目；当引用计数为零的时候，Python释放对象的空间。这个方案意味着Python无需停下来，扫描整个内存，以发现未使用的要释放空间；这也意味着引用自身的对象不会被自动收集。

供所需字符的位移值。如同在C语言中一样，Python的偏移从0开始，以比字符串长度小1的值结尾。与C语言不同的是，Python还允许你用负偏移从序列中取得内容。技术上将负偏移与字符串长度相加以生成正偏移。当然你也可以把负偏移当成从串尾向前数。

```
>>> S = 'spam'
>>> S[0], S[-2]              # 从头或尾部索引
('s','a')
>>> S[1:3], S[1:], S[:-1]    # 分片：提取片段
('pa', 'pam', 'spa')
```

第一行，我们定义了一个四字符的字符串并赋予它一个名字S。然后用两种方法索引：S[0]从左取得偏移0上的内容（单字符字符串'S'）。S[-2]从结尾取回偏移2上的内容（等效于从头数的偏移4+-2）。参见图2-1。

上面的例子中最后一行是我们第一次看到分片(slice)。当我们用一对偏移索引一个顺序对象（如一个字符串）的时候，Python返回一个新对象，它含有这对偏移所标识的相邻片段。左偏移是下边界，右偏移是上边界。Python取得从下边界开始直到上边界但是不包括上边界的所有内容，并返回一个包含取回内容的新对象。

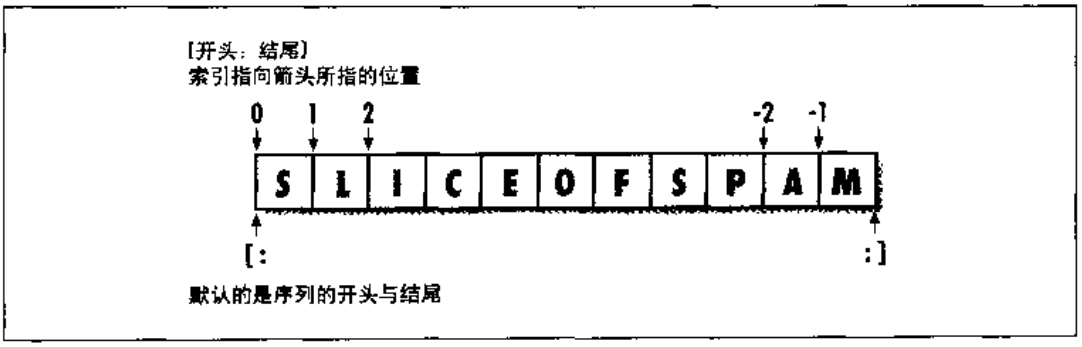


图2-1 使用偏移和分片

例如，S[1:3]提取偏移1和2处的内容，S[1:]得到第一位外的所有内容（上边界默认为字符串长度）。S[:-1]得到除最后一个字符之外的所有内容（下边界默认为0）。刚接触这些概念你可能会有些疑惑，但一旦你掌握了窍门，索引和分片使用起来非常简单而且功能强大。这里有一个关于细节的总结以供参考；记住，如果你确定不了一个分片的含义，在交互模式下试一下：

索引（s[i]）：

- 取偏移 i 处的内容（第一项在偏移量 0 处）
- 负索引意味着从末尾开始数（相当于加上长度）
- S[0]取回第一项
- S[-2]取回倒数第二个内容 （等同于 S[len(S) -2]）

分片（S[i:j]）：

- 提取序列中的邻接段
- 若忽略的话，分片的边界默认为 0 和序列的长度
- S[1:3]从偏移 1 向上取到第三项，但不包括第三项
- S[1:]从位移量 1 处一直取到结尾
- S[:-1]从位移量 0 处直至取到最后，但不包括最后的一项

在本章的后面，我们会看到通过偏移进行索引的语法（方括号），也用于通过键（*key*）来索引字典；操作看起来是一样的，但解释不同。

修改和格式化

还记得那个长词 —— 不可变序列吗？不可变的意思是不能在原处改变一个字符串（例如通过赋值给一个索引）。那么我们怎么在 Python 中改变文本信息呢？要改变一个字符串，我们只需要用合并、分片这样的工具建立并赋值一个新的即可：

```
>>> S = 'spam'
>>> S[0] = "x"
Raises an error!
>>> S = S + 'spam!'      # 要改变一个字符串，应生成一个新的
>>> S
'spamSpam!'
>>> S = S[:4] + 'Burger'+ S[-1]
>>> S
'spamBurger!'
>>> 'That is %d %s bird!'% (1, 'dead ')     # 类似 C 的 sprintf
That is 1 dead bird!
```

请留意：分片

这种提示框将贯穿全书始终，我们在此加入了选读内容，以向你介绍这门语言的某些特性是如何在真正的程序中应用的。不过，在你学会大部分Python的知识前，我们不能展示给你更多真正的应用。这些提示框内容包含很多还未介绍的主题的必要参考知识。你可以把它们当成一种预览，或许你会发现这些抽象的语言概念，在日常的编程任务中是十分有用的。

例如，你将在下面的命令行中看到参数，它们用来启动一个Python程序，这个程序应用了内置模块sys的argv属性：

```
% cat echo.py
import sys
print sys.argv

% Python echo.py -a -b -c
['echo.py', '-a', '-b', '-c']
```

一般我们只对检查传递程序名字的参数感兴趣。这把我们带向一个标准的分片应用：一个分片表达式可以除去列表中的第一项之外的所有项。在这里，sys.argv[1:]返回一个你想要的列表`['-a', '-b', 'c']`。你可以无需在前面加上程序名进行处理。

分片也经常用于清除从输入文件中读入的行。如果你知道在一行结尾会有一个行尾符（一个'\n'换行符），你可以用一个表达式如line[:-1]除去它。这个表达式抽取了除了最后字符外的所有内容（下界默认为0）。这两个例子中，分片做了逻辑上的工作，而这在低级语言中必须是显性的。

Python在对字符串操作的时候也重载%操作符（代表除法余数）。它应用在字符串上的时候，和C语言的sprintf函数作用同样：它提供了简单的方法来格式化字符串。具体的说，就是简单的在左侧放一个格式化字符串（带有嵌入的转化目标，如%c），在右侧放一个（或多个）对象，这些对象是你想在左侧插入的转换目标。例如，上面的最后一行，整数1插入字符串中%d出现的地方。在字符串中'dead'在%s处插入。字符串格式化很重要，我们多举几个例子看一看：

```
>>> exclamation = "Ni"
```

```
>>> "The knights who say %s!" % exclamation
'The knights who say Ni!'
>>> "%d %s %d you" % (1,  'spam', 4)
'1spam 4 you'
>>> "%s -- %s -- %s" % (42, 3.14159, [1, 2, 3])
'42 -- 3.14159 -- [1, 2, 3] '
```

在第一个例子中，在目标左侧插入字符串"Ni"代替%s标志。第二个例子在目标字符串中插入三个值。当不止一个值待插入的时候，你应该在右侧用括号把它们括起来（这意味着它们放到了元组中，我们一会儿将遇到）。

Python的字符串%操作符经常返回一个新的字符串作为它的结果。你可以打印也可以不打印。它也支持所有通常的C语言的printf格式代码。表2-5列出了更常见的字符串格式目标代码。一个特别值得注意的是，%s把任何对象转换为字符串，所以它通常是你要记住的唯一的转化工具。例如，上面的例子中最后一行用%s把整数、浮点数、表对象转换到字符串。格式化操作也允许我们在右侧使用字典值。但既然我们还没告诉你什么是字典（dictionary），只在这里提一下。

表2-5 字符串格式代码

%s	字符串（或任何对象的打印格式）	%X	十六进制整数（大写）
%c	字符	%e	浮点数格式1[注]
%d	十进制（整数）	%E	浮点数格式2
%i	整数	%f	浮点数格式3
%u	无符号（整数）	%g	浮点数格式4
%o	八进制整数	%G	浮点数格式5
%x	十六进制整数	%%	文字%

注：浮点代码对浮点数有不同的表示形式。参见printf文档以获知详情。更好的方法是在Python交互式解释器中试验这些格式，看看浮点格式是什么样的（如"%e %f %g" % (1.1,2.2,3.3)）。

普通的字符串工具

如前面提到的，Python提供了处理字符串的工具模块。这个string模块可能是最普通、最有用的。它包括转换大小写、查找字符串的子串，将字符串转化为数字形式等等。（Python库参考手册中有详尽的字符串工具清单。）

```
>>> import string                # 标准的模块
>>> S = "spammify"
>>> string.upper(S)              # 转化为大写
'SPAMMIFY'
>>> string.find(S, "mm")         # 返回索引的子串
3
>>> string.atoi("42"), `42`      # 转化字符串
(42, '42')
>>> string.join(string.split(S, "mm", "XX")
'spaXXify'
```

最后一个例子复杂一些，在本书的后面我们才会给出详尽的细节。简单地说，就是split把一个字符串截成一列被定界符和空格符围绕的子字符串。join把它们放在一起，两个中间有一个传递性的分界符或空格。以"XX"代替"mm"像一个迂回方法，但是这是一个任意替代全局子字符串的方法。我们只会研究这些，更高级的文本处理工具在本书后面再介绍。

顺便说一下，注意上面例子中倒数第二行atoi函数是如何把字符串转化为一个数字，并把反引号所引的任何对象都转化为字符串形式的（这里，`42`转换一个数为字符串）。记住，你不能用操作符（如+）混合操作字符串和数字类型，但是如果操作的话可以进行手工转换：

```
>>> "spam"+ 42
Raises an error
>>> "spam"+ `42`
'spam42'
>>> string.atoi("42") + 1
43
```

在后面我们会遇到一个叫eval的内置函数，可以把字符串转换为任意种类的对象。string.atoi和它相关的函数只能转换为数字，但这种限制通常意味着更快的速度。

字符串常量变量

最后，我们向你展示各种写字符串常量的方法。所有这些方法都产生同一种对象（字符串），所以这里的特殊语法只是为了我们方便。前面，我们提到字符串可以用单引号也可以用双引号括起来，这使我们可以嵌入相反类型的引号。下面是一个例子：

```
>>> mixed = "Guido's"      # 单引号在双引号中
>>> mixed
"Guido's"
>>> mixed = 'Guido"s'      # 双引号在单引号中
>>> mixed
"Guido's"
>>> mixed = 'Guido\'s'     # 反斜线转义符
>>> mixed
"Guido's"
```

注意最后两行，你可以通过一个反斜线使引号转义（告诉Python它不是真的字符串结尾）。事实上，你能在字符串中转义任何特殊字符，如表2-6中所列；Python用自己实现的特殊字符代替转义码字符。通常Python字符串中的转义码和C中的类似（注6）。同样与C类似的是Python可以为我们合并邻近的字符串常量。

```
>>> split = "This" "is" "concatenated"
>>> split
'Thisisconcatenated'
```

而且，还有一种实际中使用的Python三引号字符串常量的形式：Python收集三对引号之间括起来的块中的所有行，并将它们合并为一个单一的多行字符串，在每两行之间放一个行结束符。在这里行结束符以"\012"形式打印（记住，这是一个八进制的整数）。你可以像在C中那样叫它"\n"。例如，一行带有内嵌tab字符且在结尾处有一个换行符的文本可以在程序中写作python\tstuff\n（见表2-6）。

```
>>> big = """This is
... a multi-line block
... of text; Python puts
... an end-of-line marker
... after each line."""
>>>
>>> big
```

注6：不过要注意，正常情况下你不需要像在C语言中那样以一个\0空字符终止一个Python字符串。既然Python会在内部对字符串的长度跟踪，通常不需要在你的程序中管理终止符。事实上，Python字符串可以包含空字符\0，与C语言中的一般用法不同。例如，一会儿我们会看到在Python程序中文件数据作为字符串实现；从文件中读取或是向文件中写入二进制数据能包含空字符，这是因为字符串也可以这样。

```
'This is\012a multi-line block\012of text; Python pits\012an end-of-line
marker\012after each line.'
```

Python还有一种特殊的字符串常量形式，称为原始字符串（*raw string*）。它不把反斜线当作潜在的转义符（见表2-6）。例如串r'a\b\c'和R"a\b\c"仍保持它们的反斜线。因为原始字符串通常用于书写正则表达式，到第八章我们讨论正则表达式之后再谈论这些细节。

表2-6 字符串的反斜线字符

\newline	忽略（继续）	\n	新行（换行符）
\\	反斜线（保持一个\）	\v	垂直制表符
\'	单引号（保持一个'）	\t	水平制表符
\"	双引号（保持一个"）	\r	回车
\a	响铃	\f	进纸符
\b	退格	\0XX	八进制值XX
\e	退出（通常情况下）	\xXX	十六进制值XX
\000	空	\other	任何其他的字符（剩下的）

普通类型的概念

现在你已经看到了我们的第一个复合数据类型。让我们暂停一会儿以便定义一些通用的类型概念，这可以应用在我们以后的大多数类型上。Python很棒的一点就是一些通用的概念可以在很多场合应用。就内置类型而言，同属一个类别的所有类型的操作都是相同的，所以我们只需将大多数的概念定义一次即可。到现在为止，我们只见过数字和字符串。但它们只是Python中三个主要类型中的两个。所以你所知道的有关类型的内容要比你想像的多得多。

同种的类型共享操作

在我们介绍字符串的时候，我们提到过它们都是不可变的序列：它们不能在原位被改变，并且是有序的集合，可根据偏移访问。现在，在这一章中你看到的序列都可进行同样的序列操作（合并、索引、迭代等等）。事实上，Python中有三种操作类别：

- 数字，支持加和乘等操作。
- 序列，支持索引、分片、合并等操作。
- 映射，支持键索引等操作。

我们还没有见过映射(再过几页就要讨论字典了)，但是其他的类型大部分都是一样的。例如，对于任意的序列对象X和Y：

- X + Y生成一个新的对象，该对象包含两个操作数中的内容。
- X * Y生成一个新的对象，该对象包含X的N次拷贝。

换句话说，这些操作在各种序列下都同样工作。唯一的差别是，所得的新结果的类型，和X、Y的类型一样（如果你合并一个字符串，你将返回一个新的字符串，而不是一个列表）。索引、分片以及其他的序列操作在所有的序列上也同样工作。被处理的对象类型会告诉Python该如何运行。

可变类型可在原位改变

按可变不可变来分类可能有点抽象，但是对新手而言，跨越这一门槛是很重要的。如果说一个对象类型是不可变的，不制作一个拷贝我们就无法改变它。如果你直接改变它的话，Python会报错。一般来说，不可变的类型给了我们某种程度的完整性。保证了一个对象不会被程序的另一部分改变。在本章稍后，当我们学习共享对象引用时，我们就会明白其中原因。

列表

我们Python之旅的下一站是列表（*list*），列表是Python中最有弹性的有序集合对象类型。与字符串不同，列表可以包含任何种类的对象：数字、字符串甚至其他列表。列表可以做集合数据结构的大多数工作，而这些在C这样的低级语言中你可能不得不手工实现。Python中列表的主要属性有：

任意对象的有序集合

从功能上看，列表就是收集其他对象的地方。你可以把它们看作一个组。列表还定义了其中各项从左到右的位置顺序。

通过偏移存取

同字符串一样，通过对象的偏移索引列表，你可以取得对象的某一部分内容。既然列表是有序的，你也可以执行诸如分片、合并之类的任务。

可变长度、异构、任意嵌套

与字符串不同，列表可以增长、缩短（它们的长度是可变的）；并可以包含各种对象，而不只是单字符串（它们是异构的）。因为列表包含其他复杂对象，列表也支持任意嵌套；你能生成列表中的列表等等。

属于序列可变的类别

按我们对类型的分类，列表可以在原位上被改变（它们是可变的），也可以进行上一节我们看到的针对字符串的所有操作。事实上，序列操作在列表中工作情况相同，所以我们没有太多的可说。另一方面，因为列表是可变的。它们也支持字符串不支持的其他操作，例如，删除、索引赋值和方法等等。

对象引用的数组

从技术上讲，Python列表包含了对其他对象的0个或多个引用。如果你曾经用过一门语言，如C语言，列表可能使你想起指针的数组。从Python的列表中，取一个项如同在C的数组中引用一个值差不多一样快；事实上，在Python解释器内部，列表就是C的数组。而且，引用有些像C语言中的指针（地址），只不过你从来不用处理一个引用自身。当使用引用的时候，Python总是用引用指向的对象，这样你的程序只用处理对象操作即可。无论何时你把对象放到一个数据结构或变量名中，Python存储的总是一个对象的引用，而不是对象的一份拷贝（除非你明确指示要一份拷贝）。

表2-7 常用列表常量和操作

操作	解释
`L1 = []`	一个空的列表
`L2 = [0, 1, 2, 3]`	四项：索引为0到3
`L3 = ['abc', ['def', 'ghi']]`	嵌套的子列表
`L2[i], L3[i][j]`	索引
`L2[i:j],`	分片
`len(L2)`	求长度

表 2-7 常用列表常量和操作（续）

操作	解释
L1 + L2,	合并
L2 * 3	重复
for x in L2,	迭代
3 in L2	成员关系
L2.append(4),	方法：增长，
L2.sort(),	排序，
L2.index(1),	查找，
L2.reverse()	反转，等等。
del L2[k],	缩小
L2[i:j] = []	
L2[i] = 1,	索引赋值
L2[i:j] = [4,5,6]	分片赋值
range(4),xrange(0, 4)	生成整数的列表 / 元组

列表可以写成一串在方括号中的对象（实际上是生成对象的表达式），被逗号分开的嵌套列表可写成一串嵌套的方括号。空列表则就是一个内空的方括号（注 7）。

表 2-7 中的大多数操作看上去应该是熟悉的，因为它们都与我们先前在字符串上使用的序列操作——索引、合并、迭代等相同。表格中的最后几项是新增的。列表还可以进行方法调用（提供如排序、反转操作以及在结尾添加项等等），以及原位改变操作（删除项、赋值给索引和分片，等等）。

实际中的列表

理解列表最好的方法可能是在实践中看它们如何运用，让我们再用一些简单的解释器交互来描述表 2-7 中的操作。

注 7：不过你应该注意，实践中在列表处理程序里你不会看到像这样写出的许多列表。更常见的那些处理列表的代码都是动态构建的（在运行时）。事实上，虽然常量语法很重要值得掌握，但是大多数的 Python 数据结构都是通过在运行时运行程序代码建立的。

基本操作

对列表进行“+”和“*”操作与字符串相同，在这里它们的意思也是合并和重复，但结果是一个新列表，而不是一个字符串。就像阿甘所说的：“那就是我们所有要说的话。”将类型分成几类可以节省我们的脑力（可以使我们这样的作者更轻松）。

```
% python
>>> len([1, 2, 3])                    # 长度
3
>>> [1, 2, 3] + [4, 5, 6]             # 合并
[1, 2, 3, 4, 5, 6]
>>> ['Ni!'] * 4                       # 重复
['Ni!', 'Ni!', 'Ni!', 'Ni!']
>>> for x in [1, 2, 3]: print x,      # 迭代
...
1 2 3
```

我们将在第三章“基本语句”中谈到迭代（以及内置range函数）。值得提及的一点是：“+”需要两边序列的类型相同，否则运行时会出错。例如，你不能合并一个列表和一个字符串，除非你首先用反引号或%格式（我们在最后一节会遇到）把列表转化为字符串。你还可以将字符串转换为列表，列表的内置函数可以实现这个转换：

```
>>> `[1, 2]` + "34"          # 相当于"[1, 2]" + "34"
'[1, 2]'+34'
>>> [1, 2] + list("34")      # 相当于[1, 2] + ["3", "4"]
[1, 2, '3', '4']
```

索引和分片

因为列表是序列，索引和分片在这里同样工作。但是索引的结果是你指定的偏移处的对象类型，分片操作总是返回一个新的列表：

```
>>> L = ['spam', 'spam', 'SPAM!']
>>> L[2]                              # 从0开始的偏移
'SPAM!'
>>> L[-2]                             # 负偏移：从右数起
'Spam'
>>> L[1:]                             # 分片取得片段
['Spam', 'SPAM!']
```

原位改变列表

最后有点新内容：因为列表都是可变的，它们支持原位改变列表对象的操作；也就是说，本节中的操作都是对列表对象的直接修改，而无需像在字符串中那样必须做一份拷贝。不过Python只针对对象引用，在原位改变和生成新的对象是应该区分的。如果你原位改变对象，可能立刻影响指向它的一个以上的引用。更多内容在这一章的后半部分介绍。

在用列表的时候，你可以通过赋值一个特殊项（偏移）或是整个片段（分片）来改变它的内容：

```
>>> L = ['spam', 'Spam', 'SPAM!']
>>> L[1] = 'eggs'                    # 索引赋值
>>> L
['spam', 'eggs', 'SPAM!']
>>> L[0:2] = ['eat', 'more']         # 分片赋值：删除+插入
>>> L                                # 代替项0,1
['eat', 'more', 'SPAM!']
```

索引的赋值和C中较为相似：Python用一个新值取代已定位的对象引用。分片的赋值最好认为可分为两步：Python先删除等号左边指定的分片，然后在旧分片删除的位置插入一个新的。事实上，插入的项的数与删除的项数并不匹配。例如，给定一个列表L、值为[1, 2, 3]，赋值操作L[1:2] = [4, 5]将把L设定为[1, 4, 5, 3]。Python首先删除2（仅有一项的分片），再在2被删除的位置插入4和5。Python列表对象也支持方法调用：

```
>>> L.append('please')               # append方法调用
>>> L
['cat', 'more', 'SPAM!' 'please']
>>> L.sort()                         # 排序列表项 ('S'< 'e')
>>> L
['SPAM!', 'eat', 'more', 'please']
```

方法类似于函数，但它与特定对象相关联。调用方法的语法也是相似的（后带括号里的参数），但你要用列表对象限制方法名。限制可以写成你想要的方法名后面加一个句点。它告诉Python在对象所限制的名字空间集中去查找。严格地讲，像append和sort这样的名字可称之为属性（与对象相关的名字）。在本书后面，我们将看到导出属性的大量对象。

列表的append方法就是在列表尾部填上一项（对象引用）。与合并操作不同，append要我们传递一个单独对象，而不是一个列表。L.append(X)与L+[X]效果是类似的，但前者在原位改变，后者生成一个新的列表（注8）。sort方法在原位排序一个列表；默认情况下，它使用Python标准的比较测试（在这里，是字符串比较，你也可以使用自己的比较函数，但我们此处将忽略这种情况）。

注意：阻碍新手前进的另一个门槛是：append和sort在原位改变了关联的列表对象，但是不返回列表（严格的说，它们都返回一个叫None的值，一会儿我们会遇到），如果写成L = L.append(X)的话，你不会得到L调整后的值（事实上，你同时失去对列表的引用）。当使用append、sort这样的属性时，对象本身会同时被改变，所以无需重新赋值。

最后，因为列表是可变的，你可以用del语句删除一项或者一段。既然分片的赋值是删除操作加上插入操作，你也可以空列表赋值给一个分片（L[i:j] = []）来删除表中的片断；Python删除左边指定的分片，插入空值。而将一个索引赋值为一个空表，相当于在特定位存储了一个空表的引用：L[0] = []设定L的一项为对象[]，而不是删除它（L变为[[],...]）：

```
>>> L
['SPAM!', 'eat', 'more', 'please']
>>> del L[0]                          # 删除一项
>>> L
['eat', 'more', 'please']
>>> del L[1:]                         # 删除整个分片
>>> L                                 # 同L[1:] = []一样
['eat']
```

在继续进行之前还有几点忠告。上面提到的操作都是典型的，还有其他一些我们

注8：也与“+”合并不同，append无需生成新的对象，所以它通常更快。在另一方面，你可以用一种更好的分片语句模拟append：L[len(L):]=[X]和L.append(X)相仿，L[:0]=[X]类似于在一个列表的头部追加。两种形式都删除了一个空的片并插入X，像append一样在适当的位置迅速改变L。Python列表是作为单一的堆块实现的（而不是一个连接的列表），C程序员可能会对这一点感兴趣，并且append实际上是一个对幕后的realloc函数的调用。由于堆栈管理器不用拷贝和重分配，append可以很快。另一方面，合并一定总是生成新的列表对象并在两个操作中复制项。

没有描述的列表方法和操作（包括反转和查找）。你应该经常查阅《Python Pocket Reference》以获得全面和更新的类型工具清单。即使本书内容完备，但是新的工具随时会增加，所以本书不可能总是最新的。我还想再次提醒你，上面的原位改变只对可变的对象适用：它们不能在字符串（或者元组）上工作，不管你如何费力去试。

字典

除了列表之外，字典（*dictionary*）可能是Python中最有弹性的内置数据结构类型。如果你把列表认为是有序的对象集合，那么字典就是无序的集合；它们主要的区分就是字典中的项是通过键来存、取的，而不是通过偏移。我们将看到，内置字典可以代替你在低级语言中不得不手工实现的查找算法和数据结构，字典有时也执行其他语言中的记录、符号表的功能。字典的主要属性如下：

通过键而不是偏移量来存取

字典有时候又称为关联数组（*associative array*）或者是散列（*hash*），通过键将一系列值联系起来。这样你可以使用键从一个字典中取出一项。你可以同样使用索引操作，在字典中获取内容。但是索引采取键的形式，而不是相对偏移。

无序的任意对象集合

与列表不同，保存在字典中的项并没有特殊的顺序；事实上，Python随机排序各项，以产生快速的查用。键提供了字典中项的象征性的（而非物理性的）位置。

可变长度、异构、任意嵌套

与列表相似，字典可以在原位增长或是缩短（无需生成一份拷贝）。它们可以包含任何类型的对象，并支持任意深度的嵌套（可以包含其他的字典等等）。

属于可变映射类型

字典可以通过给索引赋值改变，但不支持我们在字符串和列表中看到的序列操作。事实上，这是因为字典是无序集合，而根据固定顺序进行操作，根本行不通。相反，字典是唯一内置的映射类型——对象将键映射到值。

对象引用表（散列表）

如果说列表是对象引用的数组，字典就是无序的对象引用表。本质上说字典是作为散列表（支持快速搜索的表）实现的，开始的时候尺寸很小，并根据要求增长。而且，Python使用了优化的散列算法来发现键，所以搜索是很快的。但是在底层，字典像列表一样存储对象引用（而不是拷贝）。

表2-8总结了一些最普通的字典操作（查看《Python库手册》可以得到完整的清单）。字典以一系列key:value（键：值）对形式写出的，用逗号分开，用花括号括起来（注9）。空的字典就是一个空的括号对，字典可以作为一个值或者是列表（元组）中的一项写在另一个字典中实现嵌套。

表2-8 常见的字典对象常量和操作

操作	解释
`d1 = {}`	空的字典
`d2 = {'spam': 2, 'eggs': 3}`	两项的字典
`d3 = {'food': {'ham': 1,'egg': 2}}`	嵌套
`d2['eggs'], d3['food']['ham']`	通过键索引

`d2.has_key('eggs'),`
`d2.keys(),`
`d2.values()`
	方法：成员关系测试，
	键的列表，
	值的列表，等等

`len(d1)`	长度（所存储的项的数目）
`d2[key] = new,` `del d2[key]`	添加/改变， 删除

正如表2-8中描述的那样，字典用键来索引；在这个例子中，键是一个字符串对象（'eggs'），嵌套字典项被一系列索引（方括号中的键）所引用。当Python生成一个字典的时候，它用自己选择的任意顺序存储项，为了取回一个值，要提供存储它的键。

注9：同样也要注意这里相对少见的常量应用：我们常常通过在运行时对新的键赋值建立字典。不过参见下一节关于修改字典的内容，列表和字典增长的方式不同。对新的键赋值对字典来说起作用，对列表则不起作用（列表通过append语句增长）。

实际应用中的字典

让我们返回到解释器中熟悉一下表 2-8 中所列的字典的一些操作。

基本操作

通常情况下，你创建字典并根据键访问其中的项。内置函数len在这里同样工作；它返回字典中存储的项的数目，或等效的键列表的长度。键列表就是字典的keys方法返回的字典中的所有键，集中为一个列表。这在顺序处理字典的时候是非常有用的。但你不应该依靠键表的顺序（记住，字典顺序是随机的）。

```
% python
>>> d2 = {'spam': 2, 'ham': 1, 'eggs': 3}
>>> d2['spam']                  # 用键取值
2
>>> len(d2)                     # 字典中的项数
3
>>> d2.has_key('ham')           # 键成员关系测试（1 代表真）
1
>>> d2.keys()                   # 键的列表
['eggs', 'spam', 'ham']
```

修改字典

字典是可变的，所以你可以改变、扩展、缩小它们而无需生成一个新的字典，就像在列表中那样。简单的给一个键赋值就可以改变或者生成项。del 语句在这里也可以工作，它删除作为索引的键相关联的项。注意这个例子里我们在一个字典中嵌套了一个列表（键值为 "ham"）：

```
>>> d2['ham'] = ['grill', 'bake', 'fry']       # 改变项
>>> d2
{'eggs': 3, 'spam': 2, 'ham': ['grill', 'bake', 'fry']}

>>> del d2['eggs']                             # 删除项
>>> d2
{'spam': 2, 'ham': ['grill', 'bake', 'fry']}

>>> d2['brunch'] = 'Bacon'                     # 添加新的项
>>> d2
{'brunch': 'Bacon', 'spam': 2, 'ham': ['grill', 'bake', 'fry']}
```

和列表一样，向字典中已经存在的索引进行赋值会改变与它相关联的值。与列表不同的是，一旦你给一个新的字典键（例如：一个以前你未定义的）赋值，你就在字典中生成了一个新的项，就如前面对 'brunch' 做的那样。在列表中情况不同，因为Python认为超出了列表末尾的偏移就是越界的。要扩充一个列表，你要用 append 方法或者分片赋值。

一个更实际的例子

这里有一个更实际的字典的例子。下面的例子生成一个列表，该列表将程序语言的名字（键）映射到它们的创造者（值）上。通过语言名字索引可以获得语言创造者的名字（译注 1）：

```
>>> table = {'Python':  'Guido van Rossum',
...          'Perl':    'Larry Wall',
...          'Tcl':     'John Ousterhout'}
...
>>> language = 'Python'
>>> creator  = table[language]
>>> creator
'Guido van Rossum'
>>> for lang in table.keys(): print lang, '\t', table[lang]
...
Tcl     John Ousterhout
Python  Guido van Rossum
Perl    Larry Wall
```

注意最后的命令。因为字典不是序列的，你不能像在字符串和列表中那样直接通过一个 for 语句迭代它们。但是如果你需要遍历各项是很容易的：调用字典的 keys 方法，返回一个你不能通过 for 迭代的所有存储键的列表，如果需要的话，你可以像上面所做的那样在 for 循环中从键到值进行索引。我们将在第三章讨论 print 和 for 语句的更多细节。

译注 1：看懂这个例子需要一点历史知识。作为脚本语言程序员，你当然应该记住几个名字：Python 语言的创造者 Guido van Rossum，Perl 语言之父，来自 O'Reilly 公司的 Larry Wall，还有 Tcl（以及 Tk）的创造者 John Ousterhout 博士。

请留意：字典接口

除了作为一种在你的程序中通过键存储信息的简便方法之外，有些Python扩展程序还提供了表面与实际工作都和字典一样的接口。例如，Python的*dbm*按键存取文件接口看上去特别像一个已打开的字典；可用键索引来存储、取得字符串：

```
import anydbm
file = anydbm.open("filename")      # 连接到扩展文件
file['key'] = 'data'                # 通过键存储数据
data = file['key']                  # 通过键取得数据
```

以后，我们会看到，在上面若我们用“shelve”代替“anydbm”，可以存储整个Python的对象（shelve是通过键访问的持久的Python数据库）。在Internet工作中，Python的CGI脚本也支持实现了一个类似字典的接口。一个对*cgi.FieldStorage*范围的调用产生了一个类似字典的对象。在客户端Web页上每一个输入字段有一项：

```
import cgi
form = cgi.FieldStorage()           # 解析格式数据 (stdin, environ)
if form.has_key('name'):
    showReply('Hello, ' + form['name'].value)
```

所有这些（和字典）都是映射的例子，更多的CGI脚本在第九章“Python常见任务”中介绍。

使用字典的注意事项

在我们学习更多的类型之前，这里有些额外的细节在你使用字典时应该知道：

序列操作不能工作

这里我们可能有些多余了，但这是新手的一个常见的问题。字典是映射，不是序列。所以在它的项中没有“有序”这个概念。像合并（有序连接）、分片（提取相邻片段）这样简单的操作都无法运用。事实上，当你尝试这样做的话，Python运行你的代码时会出错。

赋值给新的索引将添加项

键可以在写一个字典常量的时候生成（此时它们可在常量本身嵌入），或对一个已经存在字典对象中的一个新键赋值。最后的结果是相同的。

键没有必要总是字符串

我们在这里曾经用字符串作为键。但其他不可变对象（非列表）也可以照样工作。你可以用整数作键，这样可以生成一个看上去特别像列表的字典（尽管无序）。元组（下面会遇到）也是可以用作字典的键，允许复合键，甚至类实例对象（将在第六章“类”中讨论）也可以被用作键，只要它们有合适的协议方法就行；它们需要告诉Python它们的值不能改变，或者它们很少作为固定的键使用。

元组

我们总结的最后一个集合类型是元组（*tuple*）。元组由简单的对象组构成。除了不能原位改变（它们是不可变的），并且通常写成一串圆括号（而不是方括号）中的项之外，元组与列表类似。元组有列表的大多数属性，它们是：

任意对象的有序集合

与字符串一样，元组是一个有序的对象的集合；与表类似，可以嵌入到任何类别的对象中。

通过偏移存取

同字符串、列表一样，在元组中的项通过偏移（而不是键）访问。它支持我们所见过的所有基于偏移的操作，例如索引和分片。

属于不可变序列类型

类似于字符串，元组是不可变的，它们不支持我们应用在列表中的任何原处改变操作。元组是序列，它支持许多同样的操作。

固定长度、异常、任意嵌套

因为元组是不可变的，不能在不生成一个新的元组的情况下，增长或缩短。另一方面，元组可以包含其他的复合对象（列表、字典、其他元组等等），因此支持嵌套。

对象引用的数组

与列表相似，元组最好被认为是对象引用的数组；元组存储指向其他对象的存取点（引用），并且检索一个元组相对较快。

表2-9中列出了常见的元组操作。元组的书写形式是一串对象（事实上，是表达式），用逗号分开，并且用圆括号括起来。一个空的元组就是一个内空的括号对。

表2-9 常见的元组常量和操作

操作	解释
`()`	一个空的元组
`t1 = (0,)`	只有一项的元组（不是一个表达式）
`t2 = (0, 1, 2, 3)`	一个有四项的元组
`t2 = 0, 1, 2, 3`	另一个有四项的元组（同上一行一样）
`t3 = ('abc', ('def', 'ghi'))`	嵌套的元组
`t1[i], t3[i][j]`	索引
`t1[i:j],`	分片
`len(t1)`	求长度
`t1 + t2`	合并
`t2 * 3`	重复
`for x in t2,`	迭代
`3 in t2`	成员关系

表2-9中的第二个和第四项还需要一些解释。因为括号可以括起表达式（参见前面“数字”一节），你要做些特别的事情告诉Python，一个括号中对象是一个元组对象而不是一个简单的表达式。如果你真的想得到一个单项元组，只需在单项的后面和括号结尾访问添加一个尾部逗号即可。

作为特殊的例子，在不是特别容易引起语法冲突的情况不，Python也允许我们忽略元组的括号。例如，表的第四行，我们简单的列出四项，被逗号分开；在一个赋值语句内容中，即使我们没加上括号，Python可以识别出它是一个元组。作为初学者，计算的时候加上括号可能更简单。

为了与常量语法相区分，元组操作（表中最后的三行）与字符串和表是相一致的，这样我们不在这里举例子了。唯一值得注意的不同是，+、* 和分片操作将返回新的元组，并且，元组不提供我们在列表和字典中的方法。一般地说，在Python中，只有可变对象可以输出可调用方法。

有了列表，为何还要用元组？

在教初学者元组的时候，这似乎总会是第一个问题：既然已经有列表了，为什么还要使用元组？这可能有历史原因，但最好的答案应该是元组的不可变性可以提供某些整体性；你能肯定的是在一个程序中一个元组不会被另一个引用改变。而列表就没有这样的保证了。一会儿我们就会发现。有些内置操作也需要元组，而不是列表。例如当用内置函数 apply（当然我们还没有遇到）动态调用函数的时候，参数表由元组构成。有一个来自实践的原则：列表可以用于想变动的有序集合的工具；元组则处理其他事情。

文件

想必大多数读者都熟悉文件的概念 —— 计算机中由操作系统管理的取了名字的存储区段。我们最后要讲的这个内置对象类型提供了在 Python 程序内部访问文件的方法。内置函数 open 创建了一个 Python 文件对象，可作为计算机中一个文件的链接。在调用 open 后，通过调用文件对象的方法，你可以读写相关的外部文件。

比较我们现在见到的类型，文件对象多少有点不寻常。它不是数字、序列，也不是映像；相反，它只是普通文件处理任务输出模块。严格地说，文件就是一个预建立（prebuild）的C扩展类型，提供了低层C stdio文件系统之上的一个包裹层。事实上，文件对象方法和C标准库中的函数是一一对应的。

表2-10总结了常见的文件操作。为打开一个文件，程序调用 open 函数。首先是外部名，跟着是处理模式（'r' 代表为输入打开文件，用以输入；'w' 代表为输出生成和打开文件；'a' 代表为在文件尾部追加内容而打开文件，，其他的我们在这里先忽略了），两个参数必须都是 Python 的字符串。

表2-10 常见的文件操作

操作	解释
output = open(' tmp/spam', 'w')	生成输出文件（'w'代表写）
input = open('data', 'r')	生成输入文件（'r'代表读）
S = input.read()	把整个文件读到一个字符串中
S = input.read(N)	读N个字节（1或多个）
S = input.readline()	读下一行（越过行结束标志）
L = input.readlines()	读取整个文件到一个行字符串的列表中
output.write(S)	把字符串S写入文件
output.writelines(L)	将列表L中所有的行字符串写到文件中
output.close()	手工关闭（或者在垃圾收集时进行）

一旦你有一个文件对象，调用它的方法对外部文件进行读、写操作。任何情况下，Python程序中的文本采取字符串的形式。读取一个文件将返回字符串形式的文本，并且文本作为字符串传递给write方法。读和写都有多种形式，表2-10给出了最常见的一些。

调用文件close的方法将终止对外部文件的连接。我们在前面的脚注中曾谈到"垃圾收集（*garbage collection*）"，在Python中，一旦你对对象不再引用，则这个对象的内存空间自动被收回。当文件对象被收回的时候，如果需要的话，Python自动的关闭文件。正因为如此，你不必总要手工关闭你的文件，尤其不运行长时间的简单脚本；另一方面，手工关闭调用没有任何坏处，并且在大型程序中这通常是一个很不错的主意。

实际操作中的文件

这里有一个简单的例子可展示文件原理。我们先打开一个文件来输出一个字符串（以一个行终止符结束，'\n'）并关闭文件，接下来我们在输入模式不再打开文件，读取该行。注意，第二个readline调用返回一个空的字符串；这是Python的文件方法在告诉我们，已经到达文件的底部（空行是一个行终止的字符串，不是空的字符串）。

```
>>> myfile = open('myfile', 'w')              # 打开文件用以输出（生成）
>>> myfile.write('hello text file\n')         # 写文本的一行
>>> myfile.close()

>>> myfile = open('myfile', 'r')              # 打开用以输入
>>> myfile.readline()                         # 读回该行
'hello text file\012'
>>> myfile.readline()                         # 空字符串：文件结尾
''
```

另外，更高级的文件方法在表2-10中没有列出；例如，在一个文件中seek函数复位当前位置。flush强制刷新缓存终止输出便于写等等。查阅《Python库手册》或者其他的Python书籍可得到完整的文件方法清单。既然我们在第五章准备看一些文件的例子，在这里就不多说了。

相关的Python工具

open函数返回的文件对象可处理繁琐的文件接口，在第八章中，你会看到相关的有用的但更高级的Python工具，下面是对所有文件类的可用工具的一个快速预览：

基于描述文件的文件

os模块提供了使用低级基于描述文件的接口。

DBM键（keyed）文件

anydbm提供了一个通过键访问文件的接口。

持久文件

shelve和pickle模块对保存整个对象提供支持（不只是简单的字符串）。

管道

os模块也提供了POSIX接口以便于处理管道。

其他

还有些对数据库系统的可选接口，基于B树的文件，等等。

共有的对象属性

现在我们已经看到所有的Python内置类型，让我们看一下一些共有的属性吧！本节中的一些内容是对我们已学内容的复习。

类型类别复习

表2-11根据我们在前面介绍的类型类别对所有见过的类型加以分类。我们已经知道，对象根据它所在的类别共享操作。例如，字符串、表、元组（注10）都共享序列操作，我们已经看到，只有可变对象可在该位置被改变。你可以在原位置改变列表和字典，但是不可以改变数字、字符串或者元组。文件只可以输出方法，所以可变的对象无法真正的应用（在写的时候可能被改变，但是这与Python的类强制性不同）。

表2-11 对象分类

对象类型	种类	可变？
数字	数字的	不
列表	序列的	可
字符串	序列的	不
字典	映射的	可
元组	序列的	不
文件	扩展程序	–

共性

我们已经看到不少的复杂类型（组件的集合），一般来说：

- 列表、字典和元组可以包括任何种类的对象。

注10： 你可能认为不用说，数字也是不可变的，但不是在每一种编程语言中都是这样的。例如，有一些FORTRON语言的早期版本允许用户通过对一个数字常量赋值改变它的值。在Python中这可行不通，因为数字是不可变的类型；你可以确认为2永远是2。

请留意：操作符重载

在后面，我们会看到，我们自己实现的带类的对象可以从这些类别中任意的选取、选择。例如，如果你想提供一个新的特殊的序列对象，它由内置序列组成，写一个类可以重载索引、分片、合并等操作：

```
class MySequence:
    def __getitem__(self, index):
        # [index], for x in self, x in self
    def __getslice__(self, low, high):
        # called on self[low:high]
    def __add__(self, other):
        # called on self + other
```

你还可以生成新的可变对象或者不生成。通过for这个选择性的实现方法改变了操作（例如 __setitem__ 调用自身[index]=赋值的值）。虽然本书不是关于C语言集成的，也可以在C中实现新的对象作为C的扩展程序类型。因为这些，用函数指针位填上以在数字序列、映射操作设置中选择。Python的内置类型是真正的预编码C扩展程序类型。和作者一样，在你写自己的代码的时候，你需要知道类型的类别。

- 列表、字典和元组可以任意嵌套。
- 列表、字典可以动态扩大和缩小。

因为它们支持任意结构。Python的复合对象类型在程序中实现复杂信息的时候是很棒的。例如，下面的交互性操作定义了一个嵌套了复合序列对象的树。要访问它的内容，我们串起需要的许多索引操作。Python从左到右计算索引。在每一步取回一个对更深嵌套的引用（这可能是一个不合理的数据结构，但它描述了一般情况下访问嵌套对象的方法）。

```
>>> L = ['abc', [(1, 2), ([3], 4)], 5]
>>> L[1]
[(1, 2), ([3], 4)]
>>> L[1][1]
([3], 4)
>>> L[1][1][0]
[3]
```

```
>>> L[1][1][0][0]
3
```

共享的引用

我们先前提到赋值总是存储对一个对象的引用而不是拷贝。实际情况下，这通常是我们想要的。不过因为赋值可以对同一对象生成多引用。有的时候你应该知道在原处改变一个可变的对象会影响程序中其他的对同一个对象的引用。例如，在下面，我们生成一个赋值为X的表，另一个为L，对表L有一个嵌入的引用。我们也生成一个字典D包含对X的另一个引用：

```
>>> X = [1, 2, 3]
>>> L = ['a', X, 'b']
>>> D = {'x':x, 'y':2}
```

在这一点上，对我们先前生成的表有三个引用：从名字X，从表L，从表D。关系如图 2-2 所示。

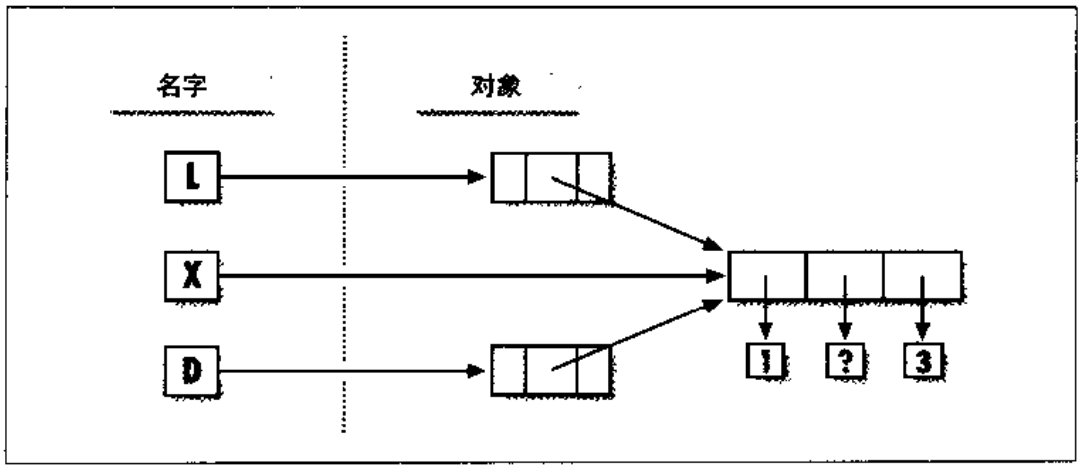


图 2-2 共享对象引用

既然列表是可变的，从三个引用中的任意一个改变共享列表对象也改变另外两个：

```
>>> X[1] = 'surprise'          #改变了所有的3个引用!
>>> L
['a', [1, 'surprise', 3], 'b']
>>> D
{'x': [1, 'surprise', 3], 'y':2}
```

理解它的另一种方法就是要知道引用是高级语言（如C）中指针的模拟。虽然你不能“抓住”引用本身，但是在不只一个地方存储同样的索引是可能的。

比较、相等性和真值

所有的Python对象也可进行比较操作：测试相等性、关系等值等等。与C这样的语言不同，Python的比较总是检查复合对象的所有部分直到可以得出一个结果。事实上，当嵌套的对象实现的时候，Python自动遍历数据结构并应用递归比较。例如，列表对象的比较操作自动的比较它们的所有内容：

```
>>> L1 = [1, ('a', 3)]           # 同样的值，唯一的对象
>>> L2 = [1, ('a', 3)]
>>> L1 == L2, L1 is L2           # 相等？同样的对象？
(1, 0)
```

在这里，L1和L2被赋值为列表，并且是相等的，但是却为不同的对象。因为Python的引用的特性，有两种方法可以测试相等性：

== 操作符测试值是否相等

Python运行了一个相等测试，递归比较所有的内嵌对象。

is 操作符测试对象的一致性

Python测试它们是否真的是同一个对象（例如：在同一个地址中）。

在我们的例子中，L1和L2通过了==测试（它们的值相等，因为它们的所有内容都是相等的）。但是is测试失败（它们是不同的对象）。有一条实践原则，==操作符可用于几乎所有的相等性检查上。但我们将在这本书的后面看到两个操作符都用的例子。等值比较可用于递归嵌套的数据结构上：

```
>>> L1 = [1, ('a', 3)]
>>> L2 = [1, ('a', 2)]
>>> L1 < L2, L1 == L2, L1 > L2         # 小于，等于，大于：一个结果的元组
(0, 0, 1)
```

在这里，L1大于L2，因为嵌套的3比2大。注意到上面最后一行的结果确实是一个有三个对象的元组——我们输入的三个表达式的结果（这是一个不带括号的元组实例）。三个值代表着真值和假值。在Python中和C一样，整数0代表假，整

数1代表真值。与C不同的是Python还识别任意空的数据结构为假，任何非空的数据结构为真，表2-12给出了Python中的对象的真值与假值。

表2-12 示例对象真值

对象	值
"spam"	真
""	假
[]	假
{}	假
1	真
0.0	假
None	假

Python还有个特殊的对象：None（表2-12的最后一项），总被认为是假的。None是Python中唯一的特殊数据结构。它通常起一个空的占位作用，特别类似于C语言中的Null指针。一般来说，我们在这一章见到的类型，Python的比较方法如下：

- 数字通过关系值比较。
- 字符串按字典顺序，字符到字符的比较（“abc” < “ac”）。
- 列表和元组通过对每一个内容作比较，从左到右。
- 字典通过比较排序后的（键、值）列表进行比较。

在以后的章节中，我们将看到其他的对象类型可以改变比较方法。例如：除非有特别的比较方法，类实例通过默认的值比较。

Python的类型层次

最后，图2-3总结了Python允许的所有内置对象类型和它们的关系。在这一章中，我们已经看到它们中的大多数比较重要的。表2-3中的其他种类的对象不是相应的程序单位（如函数、模块），就是输出的内部解释器内容（例如，栈帧和编译码）。

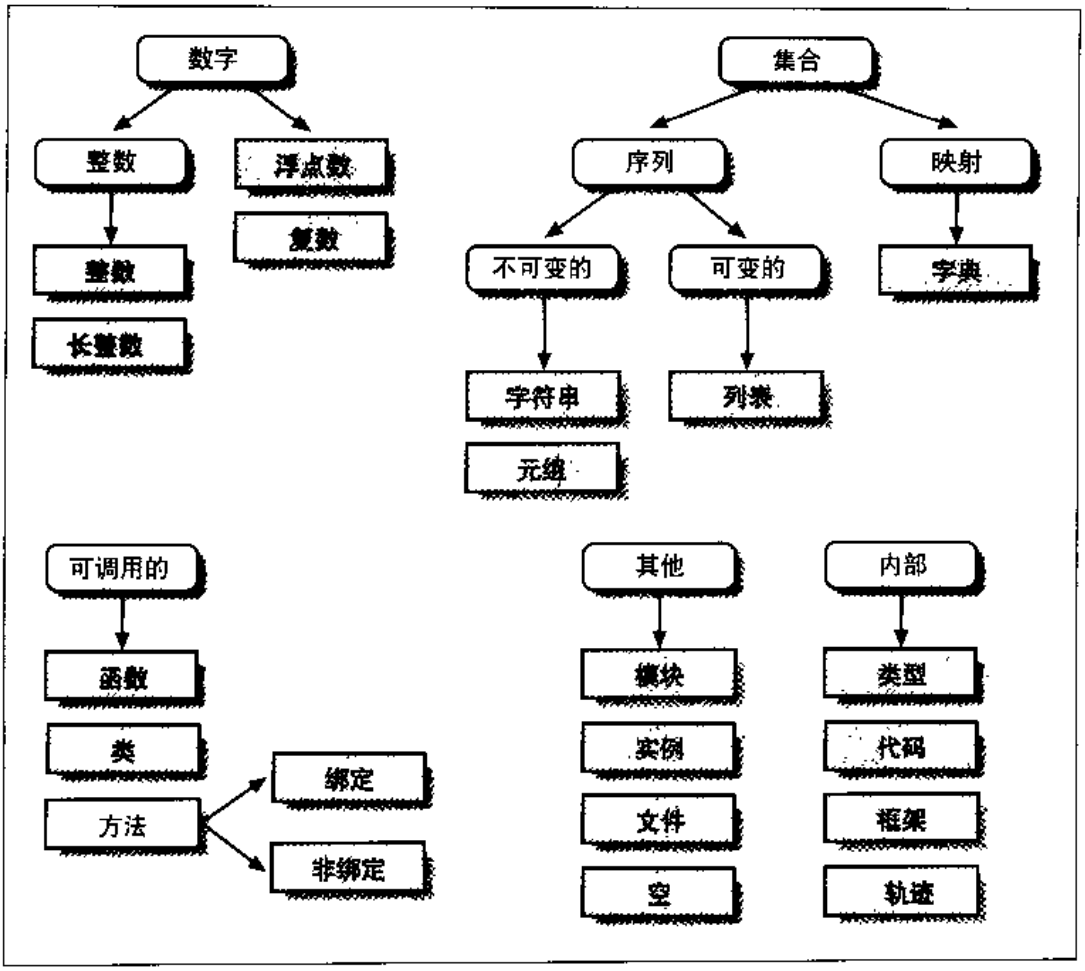


图 2-3 内置类型

我们想让你注意的最重要的一点就是，在Python中任何东西都是对象类型，可以由Python程序处理。例如，你可以传递一个栈帧给一个函数，把它赋值改变量放入列表或字典中等等。即使类型是Python中的对象类型：对内置函数 `type(X)` 的调用返回对象 X 的类型对象。除了生成惊人的绕口令之外，对象类型还可以用于Python中的手工类型的比较。

内置类型的常见问题

在这一章和后面的大多数章节中，我们将包括一个对常见问题的讨论和它们的解决办法。这些问题经常困扰新手(有时也让专家头疼)。我们称之为误区(*gotcha*)，

因为有的问题会让你困惑，尤其在你刚开始学习Python的时候。其他的问题展示了Python的奇异行为，通常在现实的编程中很少出现，但是却吸引了Internet上的编程语言爱好者（比如我们）的注意力（注11）。不管怎样，这一切都能教给我们关于Python的知识。如果你理解了这些内容，剩下的就变得简单了。

赋值生成引用，而不是拷贝

在前面我们已说过了，但是我们想在这里再说一次。如果你不理解什么是程序中的共享引用的话，下划线就可能令你迷惑不解。例如，在下面赋值为L的列表对象不但被L引用，还可被赋值为M的内部列表引用，引用改变了L也改变的M引用：

```
>>> L = [1, 2, 3]
>>> M = ['X', L, 'Y']          # 对 L的一个嵌入的引用
>>> M
['X', [1, 2, 3], 'Y']

>>> L[1] = 0                   # 也改变了M
>>> M
['X', [1, 0, 3], 'Y']
```

解决办法

这种影响通常只在大型程序中会很重要，而有的时候共享引用绝不是你真正想要的。如果不是这样，你可以通过拷贝它们以避免共享对象。在列表中，你总能生成使用一个空的有限分片生成的高级拷贝。

```
>>> L = [1, 2, 3]
>>> M = ['X', L[:], 'Y']          #  一份嵌入的L的拷贝
>>> L[1] = 0                      # 只改变L，不改变M
>>> L
[1, 0, 3]
>>> M
['X', [1, 2, 3], 'Y']
```

注11： 我们想指出的是Guido在将来的Python的版本中会去掉我们在这里描述的一些问题，不过大多数语言的基本属性不可能改变（但是不要到处引用我们的话）。

记住，分片的限定默认为0和被分开的片的长度。如果两个都被分开的话，分片在序列中展开每一项。这样就生成一个顶部拷贝（一个新的、不共享的对象）（注12）。

同层深度的重复加

当我们介绍序列重复的时候，我们说它有些像在序列后加上自身的某些倍数。这是正确的，但是当可变的对象嵌套的时候，效果可能不总像我们想的那样。例如在后面，X被赋值重复四次。然而Y赋值给一个列表，列表包含4倍的L：

```
>>> L = [4, 5, 6]
>>> X = L * 4                  # 类似[4, 5, 6] + [4, 5, 6] + ...
>>> Y = [L] * 4                # [L] + [L] + ... = [L, L,...]

>>> X
[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
>>> Y
[[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]
```

因为L在第二个重复中是嵌套的，Y结束嵌套引用，返回到开始赋值为L的列表。并且打开我们在上一节中提到的相同类型：

```
>>> L[1] = 0                   # 影响Y但不是X
>>> X
[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
>>> Y
[[4, 0, 6], [4, 0, 6], [4, 0, 6], [4, 0, 6]]
```

解决办法

这的确是另一个触发共享可被对象引用的方法，这样，上面同样的方法可以应用在左边。并且如果你记得重复、合并和分片复制只是在操作对象的顶层的话，这一类的例子更加有用。

注12：空的有限分片仍只生成一个顶部拷贝；如果你需要生成一个深层嵌套的数据结构的完整拷贝，你也可以用标准的copy模块循环引用对象。查看库手册可以得到更多的细节内容。

循环数据结构不能打印

我们实际上是在原来的练习中遇到这个问题的：如果一个复合对象包含一个指向自身的引用，就称这之为循环对象。在Python1.5.1版本以前，打印这样的对象会失败。因为Python打印机还没有智能化到可以注意到循环这个地步（你会看到同样的文本不断被打印，直到你中断执行）。这样的事现在已被注意到了，但是它值得了解一下；如果不小心，循环结构可能导致你的代码陷入预想之外的循环中。看第一章中的解决办法可以得到更多的细节！

```
>>> L = ['hi.']; L.append(L)    # 追加引用到同一个对象中
>>> L                           # 在 1.5.1 之前：这是一个循环！（ctl-C 中断）
```

解决办法

别那样做。虽然有生成循环的很好的理由，但是除非你知道代码如何处理自己，你可能不想在实际中让你的对象频繁的引用自身(除了作为一种无意义的技巧)。

不可变类型在原位不可以被改变

最后，到现在我们已经提到很多次了，你不能在原位上改变不可变类型：

```
T = (1, 2, 3)
T[2] = 4                # 错!
T = T[:2] + (4,)        # okay: (1, 2, 4)
```

解决办法

用分片、合并等等操作生成一个新的对象。但是如果需要的话向后赋值给原引用。这看上去像额外的编码工作，但是这样做的优点就是，当使用元组和字符串这样的不可变对象的时候，不会发生先前的问题。因为它们在原位置可以被改变，不会产生类似列表中的副作用。

总结

在本章中，我们学习了Python的内置对象类型——数字、字符串、字典、元组

和文件，以及Python提供的处理它们的相关操作。我们还注意到了Python的对象不包含的主题；特别是操作符重载和对象类型类别有助于简化Python中的类型。最后，我们看了一些关于内置类型的一些常见问题。

在这一章中几乎所有的例子都是经过深思熟虑而拟定的，目的是讲清楚原理。在下一章中，我们将开始语句的学习，生成处理对象，生成程序以处理更多的实际工作。

Python中的其他类型

除了在这一章我们学到的Python核心对象之外，一个典型的Python安装有数以十计的其他的对象类型，允许作为C的扩展程序或Python的类。我们在稍后会介绍一些例子：正则表达式对象，DBM文件，GUI组件等等。这些附带工具和我们所见到的数据类型的主要区别在于，内置类型有针对对象的特殊语言生成语法（例如：4用于整数，[1,2]是一个列表，open函数用于文件等等）。你在内置模块中输入的其他工具必须先导入才可以使用。参阅《Python库手册》，可以得到Python程序中允许的工具的全部指南。

练习

任务：要求你理解对象的基础知识。同前面一样，在这个过程中，一些新的概念会随时出现，当你做完的时候要浏览一下附录三。（没做完的时候也可以看！）

1. **基础。**用本章表格中的基本类型操作进行交互式试验。开始，启动Python解释器。输入下面的表达式并试着解释每列中都会发生什么：

```
2 ** 16
2 / 5, 2 / 5.0

"spam" + "eggs"
S = "Ham"
"eggs" + S
S * 5
S[:0]
"green %s and %s" % ("eggs", S)
```

```
('x',)[0]
('x', 'y')[1]

L = [1,2,3] + [4,5,6]
L, L[:], L[:0], L[-2], L[-2:]
([1,2,3] + [4,5,6])[2:4]
[L[2], L[3]]
L.reverse(); L
L.sort(); L
L.index(4)

{'a':1, 'b':2}['b']
D = {'x':1, 'y':2, 'z':3}
D['w'] = 0
D['x'] + D['w']
D[(1,2,3)] = 4
D.keys(), D.values(), D.has_key((1,2,3))

[[]], ["",[],(),{},None]
```

2. **索引和分片。**在交互式提示符下定义一个叫L的列表，包含四个字符串或者数字（如L=[1,]），现在，用一些边界的例子来试验。

   a. 试图在边界上索引会发生什么（例如：L[4]）?

   b. 界外分片会发生什么（例如：L[-1000:100]）?

   c. 最后，当你试图反向提取序列——低边界大于高边界（例如：L[3:1]）时会发生什么？提示：试着给片赋值（L[3:1] = ['?']），看一下值在哪里输出。你认为这会和超界分片出现同一现象吗？

3. **索引、分片和删除。**再定义一个列表，包括四项。并且，对一个偏移量赋值为一个空的列表（如L[2] = []），发生了什么？再试着给一个分片赋值为一个空的列表，现在发生了什么？重新调用分片赋值删除该分片在该处插入一个新的值。del语句删除偏移、键、属性和命名：试图用它在你的列表中删除一项（如del L[0]）。如果删除整个的分片会发生什么？当你给一个分片赋值一个非序列的时候会发生什么？

4. **元组赋值。**当你输入这个序列的时候你认为会在X和Y上发生什么？我们将在第三章研究这一个结构，这里只是提一下。

```
>>> X = 'spam'
>>> Y = 'eggs'
>>> X, Y = Y, X
```

5. **字典的键。**注意下面的代码片段：

```
>>> D = {}
>>> D[1] = 'a'
>>> D[2] = 'b'
```

我们知道字典不通过偏移访问；这里会发生什么？这与主题有关联吗？（提示：串、整数、元组共享哪些类型、类别？）

```
>>> D[1,2,3] = 'c'
>>> D
{1:'a', 2:'b', (1,2,3):'c'}
```

6. **字典索引。**生成一个叫Y的有三个入口的字典，键分别为a、b、c。如果你试图索引一个不存在的键d会有什么发生（如D['d'] = 'spam'）？这与列表的界外赋值索引比较怎样？这有些像变量命名规则吗？

7. **一般操作。**运行交互式测试，回答以下问题：

   a. 当你试图对不同的混合的类型使用“+”操作符时会发生什么（如：字符串 + 列表，列表 + 元组）？

   b. 当其中一个操作对象是字典的时候，“+”工作吗？

   c. append方法对列表和字符串都适用吗？在列表中用键的方法会怎样？（提示：什么是关于主题对象的append？）

   d. 最后，当你对两个列表或者两个字符串分片或者合并的时候，你可以取回什么类型的对象？

8. **字符串索引。**定义一个含有四个字符的字符串S = "spam"。输入不列表达式：S[0][0][0][0][0]，会发生什么？（提示，重调用一个字符串是一个字符的集合，但是Python的字符串都是单字符的字符串。）如果你把这应用于一个列表如['s', 'p', 'a', 'm']，这个索引表达式工作吗？为什么？

9. **不可改变性。**在此定义一个4字符的字符串：S = "spam"。写一个赋值改变字符串"spam"为，只用分片和合并。你能用检索和合并完成同样的操作索引赋值吗？索引赋值如何？

10. **嵌套。**写一个数据结构，实现你的个人信息：名字、姓、名、年龄、工作地址、Email、电话号码，你可以用你喜欢的任意内置对象类型完成这个数据

结构：表、元组、字典、字符串、数字都可以。通过检索访问你的数据结构中单独的元件，在这个对象中的结构比其他的更棒吗？

11. **文件**。写一个脚本，可以生成一个叫作*myfile.txt*的输出文件并在其中写下字符串 "Hello file world!"。再写另一个脚本打开 *myfile.txt*，读取打印其内容。从你的系统命令行下运行脚本，在你运行脚本的时候，新文件在目录中显示吗？如果你给 open 函数传去一个文件名添加一个不同的目录路径会怎样？

12. **dir 函数复习**。在交互提示符下输入如下的表达式。从 1.5 版开始，我们在第一章中遇到的 dir 函数可以这样概括：列出任何你感兴趣的对象的所有属性。如果你正在用着一个比 1.5 版本旧的版本，使用 __methods__ 可以达到同样的效果：

```
[].__methods__              # 1.4 版或者 1.5 版
dir([])                     # 1.5 或者以后
{}.__methods__
   dir({})
```

本章内容：

- 赋值
- 表达式
- print
- if条件测试
- while循环
- for循环
- 代码编写的常见问题
- 总结
- 练习

第三章

基本语句

我们已学习了Python中的基本内置对象类型，在这一章中将继续探讨它的基本语句类型。如果程序需要一些“原料”才能行使一些功能的话，语句就是你用来指定程序该做哪件事情的东西。可以说，Python是一个面向过程、基于语句的语言。通过组合这些语句，你可以指定一个过程，让Python实现令人满意的目标。

另一种理解语句的方法是回头再看一下我们在第二章“类型和操作符”中介绍的概念层次。在该章中我们讨论了内置对象。现在我们在这个层次上升到另一个水平上：

1. 程序由模块组成。
2. 模块包含语句。
3. 语句生成并处理对象。

语句处理我们已经学过的对象。而且，语句也是生成对象的地方（如在赋值语句表达式中），有的语句生成完整的对象类型（函数、类等等）。尽管我们在第五章“模块”中才会讨论这个细节，不过语句总是存在于模块中，而模块自己又是由语句所管理的。

表3-1总结了Python的语句集。我们已经介绍了其中的一部分，例如在第二章我们看到了del语句删除数据结构组件，赋值语句生成对对象的引用等等。在这一

章里，我们讲解前面忽略的细节内容和剩下的Python基本过程语句。当语句不得不处理大的程序单位：函数、类、模块、异常的时候，我们就不能再用短小的程序了。既然这些语句会引出更加复杂的编程概念，每一个概念我们将用一章来阐述。更多的独特语句像exec（作为字符串执行我们生成的代码）和assert都在本书的后面介绍。

表3-1 Python语句

语句	作用	例子
赋值	生成引用	`curly, moe, larry = 'good', 'bad', 'ugly'`
函数调用	运行函数	`stdout.write("spam, ham, toast\n")`
print	打印对象	`print 'The Killer', joke`
if/elif/else	选择事务	`if "python" in text: print text`
for/else	序列迭代	`for x in mylist: print x`
while/else	普通循环	`while 1: print 'hello'`
pass	空的占位操作	`while 1: pass`
break, continue	循环跳转	`while 1:` `  if not line: break`
try/except/ finally	缓存异常	`try: action()` `except: print 'action error'`
raise	激发异常	`raise endSearch, location`
import, from	模块访问	`import sys; from sys import stdin`
def, return	返回建立的函数	`def f(a, b, c=1, *d): return a+b+c+d[0]`
class	建立对象	`class subclass: staticData = []`
global	名字空间	`def function(): global x, y; x = 'new'`
del	删除	`del data[k]; del data[i:j]; del obj.attr`
exec	运行代码字符串	`exec "import " + "modName in gdict, ldict`
assert	除错检查	`assert X > Y`

赋值

为了给对象赋值名字，我们已用过赋值语句了。它的基本形式是在等号左边写下要赋值的目标，右边写上被赋值的对象。左边的目标可以是一个名字或对象组件。右边的对象可以是一个对象的任意计算表达式。大多数场合中，赋指可以直接使用，但这里有几个属性你应该牢牢记住：

赋值生成对象索引

正如我们已经看到的，Python 赋值在名字或数据结构中存储对象的引用（*reference*）。它总是生成对象的引用，而不是对象的拷贝。正因为如此，与数据存储结构相比，Python 变量更像 C 语言中的指针。

名字第一次赋值即已生成

正如我们也已经看到的，Python 在第一次给变量赋值（一个对象引用）时就生成变量名。无需事先预声明。有些（但不是全部）数据结构当赋值的时候也会生成（例如：词典项、一些对象属性）。一旦赋值，一个名字出现在一个表达式中时，就被它引用的值所代替。

名字在引用前必须赋值

相反，用一个你还没赋值的名字是一个错误。如果尝试这样，Python 出现一个异常，而不是返回一些意义模糊和很难注意到的默认值。

隐式赋值：import，from，def，class，for，函数参数等等

在这一节中，我们关心的是 = 语句，不过赋值在 Python 很多场合都有。例如，稍候我们将看到模块导入，函数和类定义，for 循环变量和函数参数都是隐式的赋值。既然赋值在它出现的任何场合都同样工作，所有这些场合在运行时简单把对象与名字结合起来。

表 3-2 描述了 Python 中赋值语句的各种用法。

表 3-2　赋值语句形式

操作	解释
`spam = 'Spam'`	基本形式
`spam, ham = 'yum', 'YUM'`	元组赋值（位置上的）

表3-2 赋值语句形式（续）

操作	解释
[spam, ham] = ['yum', 'YUM']	列表赋值（位置上的）
spam = ham = 'lunch'	多目标

第一行可能是最常见的：将一个对象与一个名字（或一个数据结构）结合起来。表中其他项表示的是特殊的形式：

元组与列表赋值

第二行、第三行是相关的，当你在“=”左边使用元组或列表的时候，Python将左方的目标与右边的对象匹配，并且从左到右赋值。例如，在表中第二行，名字spam赋值为字符串'yum'，名字ham定义为'YUM'。在内部，Python先在右侧生成元组的项，所以这通常叫做元组（和列表）的析取（unpacking）赋值。

多目标赋值

最后一行显示了赋值的多目标形式。在这种形式中，Python将把同一个对象的引用（对象在最右边）赋给左边所有的目标。在表中，名字spam和ham都将被赋给一个指向字符串'lunch'的引用，这样就共享同一个对象。效果与你先写ham='lunch'，随后写spam=ham等同，反正ham与源字符串对象相等。

这里有一个析取赋值的简单实例。我们在第二章练习4的解答中介绍了其中最后的一行的效果；既然Python会生成一个临时的元组，保存右边的项，析取赋值也是一种交换两个变量值的方式，而且无需生成临时元组。

```
>>> nudge = 1
>>> wink  = 2
>>> A, B = nudge, wink                  # 元组
>>> A, B
(1, 2)
>>> [C, D] = [nudge, wink]              # 列表
>>> C, D
(1, 2)
>>> nudge, wink = wink, nudge           # 元组：交换值
>>> nudge, wink                         # 等同于T=nudge; nudge=wink; wink=T
(2, 1)
```

变量名字规则

现在我们已讲完了赋值语句的全部内容，在使用变量名的时候我们应该更正规些。在Python中，当你对变量赋值时，名字就生成了。但是在程序中选取名字的时候，有一些规则要遵守。Python的变量名规则有些类似于C语言：

语法：（下划线或字母）+（任意数量的字母、数字或下划线）

变量名必须以下划线或字母开头，随之可以是任意位的字母、数字或者下划线。_spam、spam和spam_1都是合法的名字，但1_Spam 、spam$和@#!不合法。

大小写敏感：SPAM与spam不同

Python程序中对大小写是敏感的，无论是生成的名字，还是保留字都是如此。例如：名字X和x指的是不同的变量。

保留字的限制

我们定义的名字不可以与Python中有特殊意义的词相同。例如，如果我们试图用一个class这样的变量名，Python会报错，但是Klass和Class工作良好。下面的列表显示了Python中的保留字（所以限制我们使用）。

and	assert	break	class	continue
def	del	elif	else	except
exec	finally	for	from	global
if	import	in	is	lambda
not	or	pass	print	raise
return	try	while		

继续学习前，我们想提醒你的是，将名字和对象明确区分开是十分重要的。正如我们在第二章中看到的，对象有一个类型（如整数、列表），可能是可变的或不可变的。名字则相反，只是对对象的引用，它们没有可变的概念，与它们所绑定的对象类型也没有相关的类型信息。事实上，在不同的时间，赋值同样的名字为不同的种类的对象上是完全可以的：

```
>>> x = 0                    # x绑定为一个整数对象
>>> x = "Hello"              # 现在是一个字符串
>>> x = [1, 2, 3]            # 现在是一个列表
```

在后面的章节中，我们将看到名字的这一本质，在Python编程中是一个有决定性意义的优点（注1）。

表达式

在Python中，你也可以使用表达式作为语句。不过，既然表达式结果不会被保留，只有当表达式起辅助作用时，才有必要这样做。表达式通常在两种情形下作为语句使用：

为了函数和方法的调用

有的函数和方法有很多功能而无需返回值，既然你对保留返回值不感兴趣，可以用一个表达式语句调用这样的函数。在某些语言中，这样的函数有时被称为过程（*procedure*）。在Python中，它们采用函数的形式，没有返回值。

在交互提示符下打印值

类似我们已经见到的，Python回显在交互命令行下输入的表达式的结果。严格地说，这些也是表达式语句。它是print语句的速写形式。

表3-3列出了一些Python中常用的表达式语句；大多数我们以前见过。对函数和方法的调用在编写时，应把对象的列表（事实上，是计算对象的）写在函数或方法后的括号中。

表3-3 普通Python表达式语句

操作	解释
`spam(eggs, ham)`	函数调用
`spam.ham(eggs)`	方法调用
`spam`	交互打印
`spam < ham and ham != eggs`	复合表达式
`spam < ham < eggs`	范围测试

注1：如果你过去曾用过C++，你可能会对在Python中没有const声明的概念感兴趣；当然某些对象可以是不可变的。不过名字总是（或是通常）能被赋值。在以后的章节中我们会看到，Python也有在类和模块中隐藏名字的方法，但这与C++的声明不同。

表中最后一行是一个特殊的形式：Python 允许我们将大小比较条件测试串起来，以用于范围测试这样的场合。例如，表达式（A<B<C）测试 B 是否介于 A 和 C 之间。它可与布尔测试等价（A<B and B<C），但它看起来更加简单（输入也方便）。复合语句，正常情况下，不写成语句，但它的句法是合法的。在你不能确定一个表达式结果的时候，在交互模式下它非常有用。

注意： 虽然 Python 表达式可以出现在语句中，语句却不能作为表达式使用。例如，与 C 语言不同，Python 不允许我们在其他表达式中嵌入赋值语句（=）。基本理由是避免常见的编码错误。当你真正想使用“==”测试相等性的时候，你不会因为错误的输入了“=”而意外的改变一个变量。

print

print 语句的作用就是打印对象。严格的说，它把对象的文本表示写到标准的输出流中。这个标准输出流恰好与 C 的 stdout 流相一致。当你启动 Python 程序时，通常是映射到你的窗口（除非你在 shell 中把它重定向给了一个文件）。

在第二章，我们看到了写文本的文件方法。print 语句与它类似，但是更为集中：Python 将对象写到标准输出流（用默认的格式），而文件 write 方法是将字符串写到文件中。既然 Python 中有标准输出流，同在内置模块 sys 中的 stdout 一样，就有可能用文件的 write 方法模拟 print（参见下面），但是 print 易于使用。

表 3-4 列出了 print 语句的格式。

表 3-4　print 语句格式

操作	解释
print spam, ham	打印对象到 sys.stdout，中间加空格
print spam, ham,	同上，但是在尾部不添加新行

默认情况下，print 在被逗号分开的项中间添加了一个空格，在当前输出行的尾部添加了一个换行符。为了禁止换行符（这样你可以在同一行后添加更多的文

本），用逗号结束你的print语句，像表中的第二行那样。要在项中间加入空格，你可以用在第二章中提到的字符串合并和格式化工具建一个输出字符串：

```
>>> print "a", "b"
a b
>>> print "a" + "b"
ab
>>> print "%s...%s" % ("a", "b")
a...b
```

Python 的 "Hello World" 程序

现在，别再耽搁了，这是一个你已经等待很久的脚本程序了 —— Python的*Hello World*程序。真有点虎头蛇尾的感觉，为了在Python中打印一条*Hello World*信息，你只需这样就行：

```
>>> print 'hello world'                    # 打印一个字符串对象
hello world

>>> 'hello world'                          # 交互打印
'hello world'

>>> import sys                             # "硬"打印
>>> sys.stdout.write('hello world\n')
hello world
```

打印在Python中就像应该的那样简单。虽然你可以通过调用sys.stdout文件对象的write方法达到同样的效果，但print语句在处理简单的打印工作时是一件简单的工具。当然，既然表达式结果在交互命令行下回显，你通常不需要在这里使用print语句。简单的输入你想要打印的表达式就可以了。

if 条件测试

Python的if语句对执行的动作进行选择。这是Python中最重要的选择工具，实现了Python程序处理的大部分逻辑。它也是我们的第一个复合语句；像所有的Python复合语句一样，if可以包含其他语句，包括其他的if语句。事实上，Python不但允许你在程序中顺序性的组合语句（这样它们就会一个接一个执行），还允许你任意嵌套（这样它们就会在特定条件下执行）。

请留意：print和标准输出（stdout）

在print语句和向sys.stdout写内容之间的等价性是十分值得注意的。有可能将sys.stdout重赋值给一个用户定义的对象，提供和文件一样的方法（例如：write）。既然print语句只是发送文本给sys.stdout.write方法，通过将sys.stdout赋值给一个具有write的方法的对象就可以保存文本，你可以在程序中捕获打印文本。例如，你可以通过定义一个由write方法的对象来发送打印文本给一个GUI窗口。在这本书的后面我们将看到关于该技巧的一个例子。抽象一点，看起来会像这样：

```
class FileFaker:
    def write(self, string):
      # do something with the string

import sys
sys.stdout = FileFaker()
print someObjects                # 发送到写类的方法
```

Python的内置函数raw_input()从文件sys.stdin中读取，这样你可以用类似的方法拦截读取请求（用类实现类似文件的读方法）。注意，既然print文本要送到stdout流，这也是在CGI脚本中打印HTML的方法（见第九章"Python中的常用任务"）。这意味着你可以在操作系统命令行下重定向Python脚本的输入输出，通常是这样的：

```
Python script.py < inputfile > outputfile
Python script.py | filter
```

一般格式

Python的if语句与大多数过程语言很类似。它采取一种if条件测试的形式，随之以一个或多个可选的elif条件测试（即else if，意思是："否则如果"），并且以一个可选的else块结束。每一个测试和else都有一个与嵌套语句相关联的块，并且按首行缩进的格式编码。当语句运行时，如果第一个测试条件为真的话，Python执行与其相关联的代码；否则的话执行else块的代码，一个if的一般形式是这样的：

```
if <条件1>:              # if条件测试
    <语句1>              # 相关联的块
```

```
elif <条件2>:        # elif选项
    <语句2>
else:                # else选项
    <语句3>
```

例子

这有两个if语句的简单例子，除了初始化if条件测试和它的关联语句之外，所有的部分都是可选项。这是第一个：

```
>>> if 1:
...     print 'true'
...
true
>>> if not 1:
...     print 'true'
... else:
...     print 'false'
...
false
```

现在这里有一个关于if语句的最复杂的例子——包括所有可选项。语句从if行开始，然后是else块。Python执行嵌套在第一个条件下为真的语句，或者是else的部分。实际上，不仅else和elif的部分可以忽略，在每一节中还可以嵌套不止一句的语句：

```
>>> x = 'killer rabbit'
>>> if x == 'oger':
...     print "how's jessica?"
... elif x == 'bugs':
...     print "what's up doc?"
... else:
...     print 'Run away! Run away!'
...
Run away! Run away!
```

如果你已经用过C或者Pascal这样的语言，你可能对在Python中有没有switch(或case)语句感兴趣。相反，多路分支可以被一系列if/elif，或者是通过检索字典、查找列表所代替。既然词典和列表在运行时可以生成，它们有时比逻辑上的硬性编码更有弹性：

```
>>> choice = 'ham'
>>> print {'spam':  1.25,              #  一个基于字典的'switch'
...        'ham':   1.99,              # 默认用has_key()条件
...        'eggs':  0.99,
...        'bacon': 1.10}[choice]
1.99
```

一个几乎等价的if语句类似于这样：

```
>>> if choice == 'spam':
...     print 1.25
... elif choice == 'ham':
...     print 1.99
... elif choice == 'eggs':
...     print 0.99
... elif choice == 'bacon':
...     print 1.10
... else:
...     print 'Bad choice'
...
1.99
```

字典很善于用键来关联值，但用if语句编写更复杂的动作会怎样呢？我们还不能得到更多细节。不过在第四章“函数”中，我们可以看到字典还可以包含函数以表示更复杂的动作。

Python的语法规则

既然if语句是我们的第一个复合语句，现在我们需要讲一下Python的语法规则。一般的说，Python的语法很简单，以语句为基础。但这里有一些属性你需要知道：

语句：一条接一条执行，一直到你说不为止

正常情况下Python在文件中或一个嵌套块中从头到尾运行语句。但像if这样的语句（和一会儿我们将见到的语句loop）会导致解释器在你的代码中跳转。因为Python在程序中的路径称之为控制流（*control flow*），影响它的语句如if等称之为控制流语句。

块和语句边界自动探测

在代码块周围没有括号或开始/结束界定符号。相反，Python在嵌套块中用

首行缩进形式将语句分组。类似的，Python的语句也不像C中的那样以一个分号正常结束。而且，结尾的行标志所有语句的结束。

复合语句 = 首行，":"，缩进语句

Python中的所有复合语句都有同样的模式：首行以一个分号结束，接下来是一个或多个首行缩进的嵌套语句，这些缩进的语句称之为块（*block*，或有时称为suite，套）。在if语句中，elif和else从句都是if的一部分。但是首行是独立的。

空格和注释通常都被忽略掉

在语句和表达式中的空格几乎总被忽略掉（除了在字符串常量和缩进中）；至于注释：它们以#符号开头（不在字符串常量中）一直到当前行的结尾。Python也支持与对象相关联的文档字符串。但我们将忽略这些。

我们已经看到，在Python中没有可变类型的声明；这与我们用过的语言的语法有些相似。但对大多数新用户来说，没有括号和分号来标识块和语句好像是它最独特的语法特点。所以让我们进一步在更多的细节中探讨它的具体意义吧（注2）！

块界定符

已提到过了，块界定符是通过行缩进来探测的：在同一段代码中，所有的语句都向右缩进同样的距离，一直到块被一行无缩进所结束。缩进可以包含任何空格和制表符的组合。制表符可以很方便地将当前栏号移到8的倍数个空格外（但是通常混合制表符和空格不是一个好主意）。代码块可以通过比包含它们的块更深一层缩进来嵌套，图3-1描述了这个例子的块结构。

```
x = 1
if x:
    y = 2
    if y:
        print 'block2'
```

注2：如果你是一个C或Pascal程序员的话这可能十分新奇，Python的缩进语法实际上是建立在对非程序员的易用性的研究结果上的。Python的语法通常被称为语言的"所见即所得"，它可以保持一致的代码外观，有助于可读性，并可避免普通的C/C++错误。

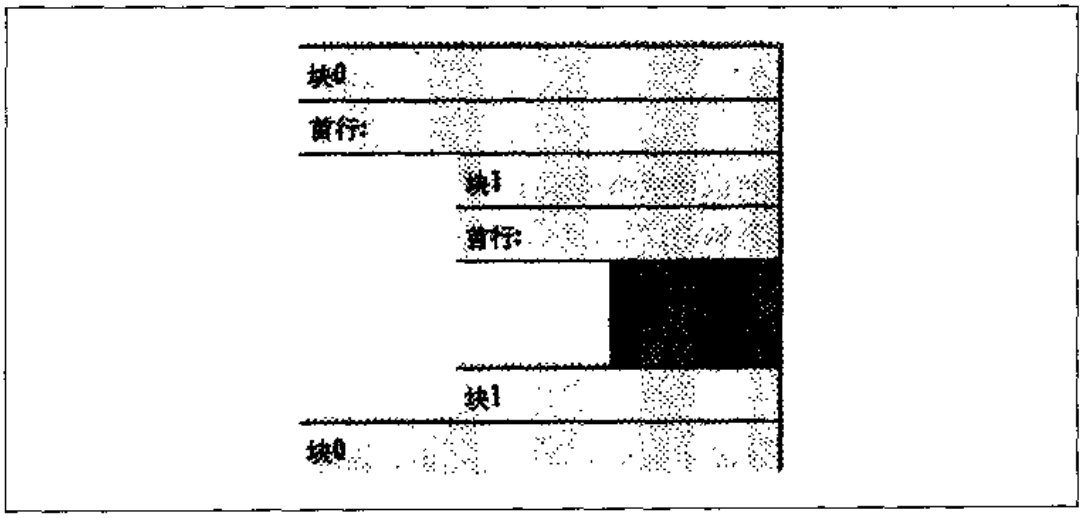


图3-1 嵌套代码块

```
        print 'block1'
    print 'block0'
```

注意，最外面的块在第一栏开头必须是非嵌套的，嵌套的块可以在任意一列的开始，但是4的倍数是一个通用的缩进风格。如果这看起来有些复杂，就像你在C或Pascal中那样书写代码吧，忽略界定符，一致的缩进代码总是符合Python规则的。

语句界定符

我们也提到过语句正常在它所在的行末尾结束，但是当语句太长了以至于一行放不下时，一些特定的规则就用得上了：

如果你正在写完一个开放的句法对，语句可以跨越多行

语句太长，一行放不下了，Python允许你继续在下一行输入语句，如果你正在写一些用()或者[]括起的成分。例如，括号括起的表达式和字典以及列表常量可以跨越任意行。后续的行可以在任意嵌套层开头。

语句如果以反斜线开头的话可以跨越多行

这是一个多少有些过时的特点，不过如果一个语句可以跨越多行，你可以在初始行加上一反斜线(\)以指示要在下一行继续(特别类似于C的#define

宏)。不过既然你还可通过在长结构上添加括号继续编码，反斜线几乎从不使用。

其他规则

很长的字符串常量可以任意跨越多行。事实上，第二章的三个括号的字符串就是被设计用来这样做的。你还可以用分号终止语句(这在一行中有不止一条语句时更有用，待会儿我们会看到)。最后，注释可以在任何地方出现。

一些特殊的例子

这里有几个例子，使用开放取对原则；我们可以跨越任意数量的行来使用界定结构：

```
L = ["Good",
     "Bad",
     "Ugly"]                          # 开放对可以跨越多行
```

在括号中放任何东西都可以：表达式，函数参数，函数头（见第四章）等等。如果你想用反斜线也可以，但这要增加额外的工作，并不要求：

```
if a == b and c == d and   \
   d == e and f == g:
   print 'olde'                       # 反斜线允许继续

if (a == b and c == d and
    d == e and e == f):
    print 'new'                       # 不过括号通常也可以
```

作为一个特例，Python允许你在同一行下写不止一个的简单语句(语句中没有嵌套语句)，用分号隔开，有的时候编码者用这个形式省地儿：

```
x = 1; y = 2; print x                 # 不只一个简单语句
```

最后，Python允许你把一个复合的语句体移到首行上去。该语句体只是一个简单的语句。你在实际中会经常看到这通常用于简单 if 语句上的条件测试：

```
if 1: print 'hello'                   # 首行的简单语句
```

你可以组合这些特殊的例子用于书写代码，但很不易读，我们不推荐这样做。实

用的原则是，尽量让每一语句都单独执行，几个月下来，你会发现这样做会使你很高兴。

复习真值测试

我们在第二章中介绍了比较相等性、真值的概念。既然if语句是用户可以测试结果的第一个语句，我们要在这里扩展一下这些概念。实际中Python的布尔操作与C这样的语言稍有不同。在Python中：

- 真代表着任意非空的数或非空的对象。
- 假代表着不真：数0、空的对象或者None。
- 比较或判断相等返回1或者0（真或假）。
- and和or布尔操作返回一个真或假的操作数对象。

最后一项是新的内容；简单地说，布尔操作用于组合其他测试的值。这儿有Python中的三个布尔表达式操作：

`X and Y`
　　如果X和Y都为真的时候值为真

`X or Y`
　　X和Y有一个为真则值为真

`not X`
　　如果X为假则值为真（表达式返回1或者0）

在这里，X和Y可以是任意真实的数值或一个返回真实数值的表达式（如一个相等性测试，范围比较等）。与C不同的是，布尔操作在Python中以词的形式输出（而不是C语言中的||和!），同样与C不同的是，and或or布尔操作在Python中返回一个真或者假的对象，而不是整数1或0，让我们看看它们是如何工作的：

```
>>> 2 < 3, 3 < 2          # 比较大小，返回1或者0
(1, 0)
```

这样的大小比较返回一个整数1或者0，作为它们真值结果。但是and和or操作总是返回一个对象。对于or测试，Python从左到右检测操作对象，返回的一个值为真的对象。而且，Python通常在发现第一个值为真的时候就停止检测。既然发现一个真值结果就终止表达式其余部分的检测，这通常称之为短路检测。让我们看看它们是如何工作的：

```
>>> 2 or 3, 3 or 2        # 如果真则返回左边的操作数
(2, 3)                    # 否则返回右边的操作数（无论真或假）
>>> [] or 3
3
>>> [] or {}
{}
```

在上边的的一行，两个操作数都为真（2，3），这样Python总是停止并返回左边的一个，在第二个测试中，左边的操作数为假（[]），这样Python停止并返回它作为测试结果。在最后的测试中，左边的为真（3），这样Python检测并返回在右边的对象（恰好为假）。最后结果同C一样（真或假），但它是基于对象的，不是整数标志（注3）。

while循环

Python的while循环语句是它最常用的递归结构。简单地说，它重复执行一段递归结构，只要顶部的计算值为真就继续。当条件测试变为假时，则继续while内后面所有语句的内容。while内的结构体从不在测试条件下为假的时候运行。

while语句是两个循环语句中的一个（另一个是for，下面会遇到）。我们称它为循环，因为控制流总是循环返回语句的开头，直到条件为假为止。最后的效果就是当顶部的条件为真时，循环体重复执行。Python也提供了一些便利的工具处理

注3：一个通用的使用Python布尔操作符的方法，是用一个or语句从一个或多个对象中选择一个或是多个对象：像X=A or B or C这样的语句把X设为A，B，C中第一个非空的对象。短路计算对于加强理解十分重要，因为在一个布尔运算符右边的表达式可能会调用函数做许多的工作或者产生其他效果，如果短路规则生效的话这些就不会发生。

复杂循环（迭代），如map、reduce、filter函数和in成员测试函数。在这本书后面我们将讨论其中的一部分。

一般格式

while语句的最复杂形式包含一个带有条件表达式的首行，一个或多个缩进的语句和一个可选的else部分，该部分在控制流退出循环而没有再运行一个break语句的时候执行。Python不断在顶部检测条件并执行在while部分嵌套的语句，直到测试返回一个假值：

```
while <条件>:        # 循环测试
    <语句1>          # 循环体
else:                # 可选的else
    <语句2>          # 如果不是因为break而退出循环则运行
```

例子

为了便于描述，这里有几个while循环的简单的例子。第一个只是永远打印信息，通过在一个while循环中嵌套一个print语句。请记住一个整数1意味着真。既然条件总为真，Python将不断执行循环体直到你终止它的执行。这一类行为通常称为无限循环（当你不想这样的时候它很烦人）：

```
>>> while 1:
...     print 'Type Ctrl-C to stop me!'
```

第一个例子每次取到一个字符串的第一个字符，一直到字符串为空。在这一章后面，我们将看到遍历字符串的其他更便利的方法。

```
>>> x = 'spam'
>>> while x:
...     print x,
...     x = x[1:]   # 从x中取出第一个字符
...
spam pam am m
```

最后，下面的代码从a的值递增，直到b但是不包括b，它有些类似C语言的for循环。一会儿我们将看到使用for和range语句的更简单的方法。

```
>>> a=0; b=10
>>> while a < b:    # 编码计数循环的一种方法
...     print a,
...     a = a+1
...
0 1 2 3 4 5 6 7 8 9
```

break、continue、pass、else循环

现在已看到了我们用Python写的第一个循环，我们应向你介绍一下两个语句——break和continue。仅在循环内部嵌套的时候，这两个简单的语句才有用。若你曾用过C语言，你可以忽略这一节的大多数内容。因为它们在Python中是一样。既然else和break关系密切，我们在这里将对else多叙述一些。我们还要看一下Python的空语句pass，它有些类似于C语言中的空语句（只一个分号）。在Python中：

`break`

跳出最近的包围它的循环（通过整个循环语句）。

`continue`

跳至最近的包围它的循环顶部（到循环的首行）。

`pass`

什么都不做：只是一个占位的空语句。

`else`循环块

运行并且只有在循环正常退出的情况下运行。如：没有碰到break语句。

一般的循环格式

当包含break和continue语句的时候，while循环的一般格式像这样：

```
while <条件测试>:
    <语句>
    if <条件测试>: break          # 现在跳出循环，忽略else
    if <条件测试>: continue       # 现在转到循环顶部
else:
    <语句>                        # 如果我们没遇上break
```

break和continue语句可以在while循环体的任何位置出现，但正如我们展示的那样，它们通常被写在更深一层的if嵌套中，对某些条件进行响应。

例子

让我们转向一些简单的例子，来看一看这些语句在实际中是怎样组织的。pass语句通常被用于为一个复合语句编写一个空的语句。例如，如果你想写一个无限循环，每次循环什么都不做，就要用pass：

```
while 1: pass    # 输入Ctrl-C停止!
```

既然语句体只是一个空的语句，Python将陷入这个循环，消耗CPU的时间(注4)。pass和语句的关系恰如None与对象的关系——一个明确的空。注意到while循环体与首行在同一行。如同在if中一样，这只在语句体不是一个复合语句时才会正常工作。

continue语句有的时候让你可以避免语句嵌套；下面的例子可以跳过奇数打印所有的小于10大于或等于0的偶数。记住，0代表假，%代表取余数。所以这个循环一直计算到0，跳过那些不是2的倍数的数（它打印8、6、4、2、0）。

```
x = 10
while x:
    x = x-1
    if x % 2 != 0: continue    # 奇数? --打印
    print x,
```

因为continue跳至循环的顶部，你无需在一个if测试中嵌套print语句：只有在continue不运行时才可以达到print语句。这看上去有些像其他语言中的goto语句。Python中没有goto，但因为continue也允许你在程序中跳转，所有你听到过的关于goto可读性的警告都是适用的。尤其是当你刚开始学习Python的时候使用continue应该保守一些。

注4：这可能是我们写过的Python程序中最没有用的，不过坦率的说，我们想不出一个更好的pass例子。在本书后面我们会看到它发挥作用的场合（例如，定义一个空的类）。

break语句常可以消除在其他语言中使用的查找状态标志。例如，下面这段代码通过查找比1大的因子来判断y是否为一个素数：

```
x = y / 2
while x > 1:
    if y % x == 0:                    # 余数
        print y, 'has factor', x
        break                         # 跳过else
    x = x-1
else:                                 # 正常退出
    print y, 'is prime'
```

当循环退出时没有设定待检测的标志，而是在发现因数后插入一个break。在这种方式下，else循环可以认为只在没有发现因数时，它才会被执行，如果没有碰上break,该数就是素数。注意，else循环在循环体不执行时也会被运行。在while循环中，当一开始首行为假时就会执行循环体。在上面的例子中，如果x初始化小于或等于1，你也会得到一个“is prime”（是素数）的信息（例如：如果y等于2）。

for循环

for循环是Python中一个常见的顺序迭代操作：它可以步进遍历任意对象，只要对象中的项可进行序列索引操作。for可以工作在字符串、列表、元组和以后我们用类生成的新对象上。实际上我们已经看到for的应用了，记得吗？我们在第二章谈到序列的迭代操作时曾经提到过。在这里，我们将补上先前忽略的细节。

一般格式

Python的for循环以一个首行开始，该行指定一个（或多个）赋值目标，紧接着你想步进遍历的一个对象。首行后跟随一个你要重复执行的缩进语句块：

```
for <目标> in <对象>:      # 该目标赋以对象的项
    <语句>                 # 重复循环体：使用目标
else:
    <语句>                 # 如果我们没有遇上break
```

当Python运行for循环时，它将序列对象中的项一个接着一个赋值给目标，并且为每一个执行一次循环体（注5）。循环体通常使用赋值目标指向序列中当前的项，如同一个光标在遍历序列一样。严格的说，for通过用连续的更高级的索引（从0开始）重复检索序列对象，直至检索出界，出现异常。因为for循环在幕后自动管理序列检索，它们代替了可能在C语言中用到的大多数计数类型循环。

for也支持一个可选的else块，和在while循环中的完全一样。如果循环没有遇到break语句而退出，它就会执行（也就是说序列中所有的项都被访问了）。上面介绍的break和continue语句在for语句中一样可用，在这儿我们就不做重复介绍了。不过for循环的完整格式可以用这种方式描述：

```
for <目标> in <对象>:              # 对象的项赋值给目标
    <语句>
    if <条件测试>: break           # 现在跳出循环，忽略else
    if <条件测试>: continue        # 现在到循环顶部
else:
    <语句>                         # 如果我们没有遇上'break'
```

例子

让我们交互输入几个for循环。在下面的第一个例子中，名字x被依次赋值为列表中的每一项，从左到右，print语句为每一项打印。在print语句内（循环体内），x指的是列表中的当前项：

```
>>> for x in ["spam", "eggs", "ham"]:
...     print x,
...
spam eggs ham
```

接下来的两个例子计算列表中的所有项的结果。在第八章“内置工具”中，我们可以看到对列表中的项运用+和*等操作的内置工具，但用for通常更简单：

注5：作为赋值目标在一个for首行使用的名字，只是一个（可能是新的）在for语句的名字空间（作用域）内的变量。关于这一点没有什么特别的；它甚至可以在一个for的循环体内被改变，不过当控制权再一次到达循环体的顶部的时候，它自动地设置为序列的下一项。

```
>>> sum = 0
>>> for x in [1, 2, 3, 4]:
...     sum = sum + x
...
>>> sum
10
>>> prod = 1
>>> for item in [1, 2, 3, 4]: prod = prod * item
...
>>> prod
24
```

我们已经提到for循环也适用于字符串和元组。我们没有提及的一件事是：如果你通过元组中的序列迭代的话，循环目标可以是一个元组的目标。这也是元组在工作中析取赋值的另一个例子。for将序列中的项赋值给目标，赋值同样可在任意位置工作：

```
>>> S, T = "lumberjack", ("and", "I'm", "okay")

>>> for x in S: print x,
...
l u m b e r j a c k

>>> for x in T: print x,
...
and I'm okay

>>> T = [(1, 2), (3, 4), (5, 6)]
>>> for (a, b) in T:                    # 工作中的元组赋值
...     print a, b
...
1 2
3 4
5 6
```

现在，让我们看些更复杂的事情。下面的例子描述了在for语句中的else循环和语句的嵌套。给定一个对象的列表（项）和一个键（条件）的列表，这段代码在对象列表中查找每一个键，查找成功就报告：

```
>>> items = ["aaa", 111, (4, 5), 2.01]      # 一个对象集合
>>> tests = [(4, 5), 3.14]                  # 用于查找的键
>>>
```

```
>>> for key in tests:                          # 所有的键
...     for item in items:                      # 所有的项
...         if item == key:                     # 检查是否匹配
...             print key, "was found"
...             break
...     else:
...         print key, "not found!"
...
(4, 5) was found
3.14 not found!
```

既然嵌套的if在匹配时才运行break，else循环可以认为查找已经失败。注意这里的嵌套：代码运行时同一时间有两个循环，外部循环扫描键列表，内部循环扫描每一个键的列表项。嵌套的else循环是重要的；它与内部for循环首行的缩进处于同一列，所以它是与内部循环关联的（而不是if或外部的for）。顺便说一下，如果你为了测试成员关系使用了第二章中的in操作的话，这个例子很好写。既然in隐含扫描一个列表以查找匹配，它可以代替内部循环：

```
>>> for key in tests:                          # 用于所有的键
...     if key in items:                        # 让Python检查匹配
...         print key, "was found"
...     else:
...         print key, "not found!"
...
(4, 5) was found
3.14 not found!
```

一般的说，让Python作类似这样的工作是一个好主意。下一个例子用for执行一个典型的数据结构任务——把普通的项收集到两个序列中（字符串）。这是一个简单的求交集的例程。循环运行后，指向列表的res返回所有在seq1和seq2中公共的项（注6）：

```
>>> seq1 = "spam"
```

注6：这与人们通常所说的集合的交集不同（如果一项在seq1中出现多次的话，在结果中也可以出现多次）。为了避免重复，可以在循环的内部用if x in seq2 and x not in res。顺便说一下，对列表如何动态建立（通过程序代码），而不是作为常量被写出而言，这是一个很棒的例子。就像我们前面提到的，大多数的数据结构是被建立的，而不是被写出来的。

```
>>> seq2 = "scam"
>>>
>>> res = []                          # 开始为空
>>> for x in seq1:                    # 扫描第一个序列
...     if x in seq2:                 # 公共项?
...         res.append(x)             # 向结尾添加结果
...
>>> res
['s', 'a', 'm']
```

不幸的是，这个代码只对两个特殊变量适用：seq1和seq2。如果这个循环可以变成一个工具可以让我们多次使用它就太棒了。我们将看到，这个简单的想法把我们带到“函数”——下一章的主题。

range和计数循环

for循环包括了大多数计数类型的循环。当你想遍历一个序列的时候，它应该是你想到的第一个工具。但有些情形需要用一些更复杂的方法迭代，你可以总用while循环编写一个独一无二的迭代，但Python还提供了一个用for语句的特定索引方式。内置的range函数返回一个连续的更大整数的列表，它可以在for中作为索引使用（注7）。

范例

一些例子可以使这这些内容更具体些。range函数真正独立于for循环，虽然它大多数用在一个for语句中生成的检索，当你需要一个整数列表时，你可以在任何地方使用它：

```
>>> range(5), range(2, 5), range(0, 10, 2)
([0, 1, 2, 3, 4], [2, 3, 4], [0, 2, 4, 6, 8])
```

带一个参数时，range从0生成一个带整数的列表，直到（但不包括）参数的值。如果你传递两个参数，第一个用于下边界。一个可选的第三个参数可以给出一个

注7：Python也有一个内置的xrange函数，一次生成一个索引而不是立即在一个列表中全部存储它们。xrange没有速度上的优点，不过如果你不得不生成一个大量的值的话，它很有用。

请留意：文件扫描循环

一般说来，当你要不止一次地重复或处理某件事的时候，使用循环是很方便的。既然文件包含多个字符、多行，它们就可以是for循环的一个典型应用。例如，用while和break，而不是在顶部使用文件结束条件，来扫描文件是很常见的：

```
file = open("name", "r")
while 1:
    line = file.readline()       # 取得下一行
    if not line: break           # 在文件结束时跳出循环（空字符串）
    在这里处理行
```

for循环扫描文件也很方便。readlines文件方法可以给你一个用来遍历的行列表，在第二章中已经介绍了：

```
file = open("name", "r")
for line in file.readlines():   # 读入一个行列表
  在这里处理行
```

在其他的情况下，你可以按字节扫描（用while和file.read(1)），或者是一次载入文件的全部（如for char in file.read()）。在本书的后面，我们将学到更多的关于文件处理的内容。

步长（*step*），如果用它的话，Python给连续的每一个节点加上这个步长（默认的步长为1）。现在，最简单的遍历一个序列的方法是用一个简单的for。Python为你处理了大多数细节：

```
>>> X = 'spam'
>>> for item in X: print item,          # 简单的迭代
...
s p a m
```

在内部，for初始化一个索引，检测序列是否结束，索引序列以取得当前的项。在每一个迭代上增加索引。如果你真想知道索引的逻辑复杂性，你可以用一个while循环做一下：这个形式和C语言中的for循环相似，你可以在Python中实现：

```
>>> i = 0
```

```
>>> while i < len(X):                    # while迭代
...     print X[i],; i = i+1
...
s p a m
```

最后，你仍然可以用 for 进行手工索引，用 range 生成一个检索列表用来迭代：

```
>>> for i in range(len(X)): print X[i],   # 手工检索
...
s p a m
```

不过除非有一个特别的检索需求，你最好还是用 Python 中简单的 for 形式。当在事务中需要对一个数重复很多次的时候，这个时候用 range 是很方便的。例如，打印三行内容，用 range 生成整数：

```
>>> for i in range(3): print i, 'Pythons'
...
0 Pythons
1 Pythons
2 Pythons
```

代码编写的常见问题

在你开始练习前，我们想指出新手在编写 Python 语句和程序的时候似乎总会犯的错误。一旦你有了一些 Python 编程经验，你就会学会如何避免它们（事实上，Mark —— 本书作者之一 —— 通常遇到的麻烦都是因为在 C++ 代码中使用了 Python 语法）。不过简单的几句话可以帮助你避免陷入这些初始时的陷阱。

不要忘了冒号

不要忘了在一个复合语句头后输入一个冒号（if、while、for 的第一行）。开始的时候你可能有些不习惯，实际上这样做会使你很方便。

第一列开始

在第一章“开始”中我们提到过，这里再提个醒儿：一定要从第一列开始写顶层（无嵌套的层）。包括输入到模块文件的无嵌套代码，以及交互模式下输入的无嵌套代码。

交互提示符下的空白行

模块文件中，复合语句的空白行总是被忽略掉。但是，在交互模式下输入代码的时候空白行代表结束语句。如果你想继续的话，在...提示符下别敲回车，直到你真正的结束为止。

缩进的一致性

避免在缩进的时候混合使用制表符和空格符，除非你确信你的编辑器可以处理制表符。否则，你在编辑器中看到的可能不像你在Python中看到的那样，认为制表符代表好几个空白符。

在Python中不要写C的代码

C/C++程序员应注意：你没必要在if的条件和while的头加上括号（例如，`if (X==1): print X`），不过如果你喜欢的话这样也可以。任何表达式都可以被括号括起。也请记住，你不能在块周围用{}；应该用缩进的嵌套代码块。

不要总想得到结果

另外要提醒你的是：在第二章中提到的在原位修改操作（如`list.append()`和`list.sort()`）不返回值（事实上，返回的是空值None）；无需赋值结果即可调用它们。新手通常想通过`list=list.append(X)`试图得到追加后的结果；相反，把空值给了列表，而不是调整后的列表（实际上，你随列表一起丢掉了一个引用）。

适当的使用call和import

最后要提醒你的一点是：你必须在函数名后添加括号才可以调用它。无论它是否用了参数（如：`function()`，而不是`function`），在import中你不应包括这样的文件后缀（如：`import mod`，而不是`import mod.py`）。在第四章，我们将看到函数有特殊操作的简单对象——以括号启动的调用。在第五章，我们将看到模块除了*.py*还有其他的后缀（如*.pyc*）；硬性编码一个特定的后缀不仅是不合语法的，而且毫无意义。

总结

在这一章，我们探讨了Python的基本过程语句：

- 赋值存储对对象的引用。
- 表达式调用函数和方法。
- print向标准输出流发送文本。
- if/elif/else在一个或多个条件中选择。
- while/else循环重复一个动作直到条件为假。
- for/else循环步进遍历一个序列对象中的各项。
- break和continue在循环中跳转。
- pass语句是一个空的占位符。

我们也学习了Python的语法规则，以及布尔操作和真值测试。我们还谈到了Python中的基本编程概念。

通过组合基本语句，我们能编程处理基本的逻辑需求。在第四章，我们可以看到一些其他用于写函数的语句，可以析取数据以供重复使用。以后的章节，我们将看到处理大型程序单元的更多的语句，以及异常。表3-5总结了在以后各章节中我们会遇到的一系列语句：

表3-5 预览

单位	作用
函数	过程单元
模块	代码/数据包
类	新的对象
异常	错误和特别的事例

练习

现在你知道如何编写基本的程序。这部分的目的是让你用语句实现一些简单的任务。大多数工作都在练习4中，它可让你探索一下交互式编码。有很多方式组织语句，学习Python的部分目的就是学习哪一种组织方式比其他的更好。

1. **编写基本循环。**

   a. 写一个 for 循环，打印一个名字为 S 的字符串中的每一个字符的 ASCII 码，用内置函数 ord(character)将每个字符转换为 ASCII 整数（交互式测试它，看它如何工作）。

   b. 接下来，改变你的循环，在一个字符串中计算所有字符串的 ASCII 码的和。

   c. 最后，再一次调整你的代码，返回一个包含该字符串中的每一个字符的 ASCII 码的新的列表。表达式 map(ord, S)能达到类似的效果吗？（提示：参考第四章“函数”）

2. **反斜线字符。**在你的机器交互地输入下列代码，会发生什么？

```
for i in range(50):
    print 'hello %d\n\a' % i
```

   警告：这个例子会出现响铃，你可能不想在一间人很多的实验室运行它。（除非你想得到大家的注意）。提示：参考表 2-6 中的反斜线转义字符。

3. **字典排序。**在第二章中，我们得知字典是一个无序集合。写一个 for 循环以升序打印字典中的项。提示：用字典的 keys 和列表的 sort 方法。

4. **程序逻辑的选择。**分析下面的代码，使用了一个 while 循环和一个 found 标志。在一个 2 的幂的列表中查找值，直到找到 2 的 5 次方（32）。它保存在一个叫 *power.py* 的模块文件中。

```
L = [1, 2, 4, 8, 16, 32, 64]
X = 5

found = i = 0
while not found and i < len(L):
    if 2 ** X == L[i]:
        found = 1
    else:
        i = i+1

if found:
    print 'at index', i
else:
    print X, 'not found'
```

```
C:\book\tests> Python power.py
```

```
at index 5
```

该例子并没有因循正常的编码技巧。随着下面的步骤改进它：对于所有的改变你都可以交互的输入你的代码，或存储在一个脚本文件中。（虽然用一个文件练习起来更简单。）

a. 首先用一个 while/else 循环重写代码。删除 found 和末端的 if 语句。

b. 接下来，用一个 else 的 for 循环重写这个例子，除去复杂的逻辑上的列表检索。提示：要得到检索中的一项，用列表的检索方法（L.index(X) 可返回 L 中的第一个的偏移量）。

c. 现在，用一个简单的 in 操作成员表达式重写该例子，完全去掉循环（看第二章，可得到更多的细节，或这样输入：2 in [1,2,3]）。

d. 最后，用一个 for 循环和列表的 append 方法来生成一个 2 的幂的列表 L，代替一个硬性编码列表常量。

e. 进一步思考：（1）你认为把 2**x 表达式移到循环外部会提高程序性能吗？（2）我们在练习 1 中看到，Python 也包括了 map(function, list) 工具，也可以生成 2 的幂的列表，像下面这样：map(lambda x: 2**x, range(7))。试着交互输入这些代码；在第四章我们会正式遇到 lambda 语句。

本章内容：

- 为什么要使用函数？
- 函数基础
- 函数中的作用域规则
- 参数传递
- 其他内容
- 函数的常见问题
- 总结
- 练习

第四章

函数

在上一章中，我们学习了Python中的基本语句。在这里，我们将继续探讨其他语句。这些语句可生成我们自己的函数。用简单的术语来讲，函数就是组合一组语句的一种工具，这样我们可以在程序中不止一次的运行它。函数也允许我们指定参数，这样每次函数代码在运行时都会有不同。表4-1总结了在本章我们要学到的函数语句。

表4-1　函数相关语句

语句	例子
调用函数	`myfunc ("spam, ham, toast\n")`
`def, return`	`def adder (a, b, c=1, *d): return a+b+c+d[0]`
`global`	`def function (): global x, y; x = 'new'`

为什么要使用函数？

谈论细节之前，让我们先对什么是函数有一个清晰的概念。函数是一种通用的程序结构化工具。你可能在其他语言中已见过它了。不过简短地来说，函数有两个主要作用：

代码重用

像在大多数语言中一样，要想在不同地方，不同时间不止一次的使用代码，函数是最简单的逻辑打包方式。直到现在我们写的所有代码都是立刻运行的；函数允许我们组合并参数化大量的代码以便以后重复地使用。

过程的分解

函数也是把系统分割成许多片段的一种工具，每个函数都有一个定义良好的作用。例如，要从头开始做比萨饼，你要从和面开始，添加作料、烘焙等等一系列工作。如果你为一个做饼的机器人编程，函数会把你“做饼”的任务分成许多单一的小任务。单独的实现一个较小的任务要比一次实现整个处理过程容易。一般的说，函数是关于过程的 —— 如何做，而不是要做什么。在第六章“类”中我们将明白这两者的区别。

在这里，我们将讨论函数基础，作用域规则和参数传递，以及一些相关概念。你将看到，函数没有应用太多的新语法，但是它带给我们一些书写大型程序的概念。

函数基础

虽然我们还没正式接触函数，但在先前的章节中我们已使用过了。例如，要生成一个文件对象，我们调用内置函数 open。类似地，我们用内置函数 len 来得到一个集合对象的项数。

在这一章中，我们将学习如何在 Python 中写新的函数，我们自己写的函数与已看到的内置对象有一样的行为方式 —— 在表达式中被调用，被传递值，返回值。但书写函数需要一些新的概念；这有一个对主要概念的介绍：

def 生成一个函数对象并赋给它一个名字

Python 用一个新语句 def 写函数。与 C 这样的编译语言不同，在运行时，def 是一个可执行语句。它生成一个函数对象并给它取一个函数名。同所有的赋值一样，函数名字成为对函数对象的一个引用。

return 给调用者返回一个结果对象

一个函数被调用后，要等到函数完成了工作并将控制权返回给调用者，调用才会结束。函数用一个 return 语句计算一个值，并返回给调用者。

global 声明模块级被赋值的变量

默认情况下，所有对函数命名的赋值都是局部的，只在函数运行时存在。为了在一个包含它的模块中赋值一个名字，函数需要在一个 global 语句中列出它。

参数通过赋值传递（对象引用）

在Python中，函数通过赋值传递参数（也就是通过对象引用）。我们将看到，它与C的传递规则或是C++的参数引用不尽相同 —— 调用者和函数通过引用共享对象，但没有别名（改变一个参数名并不改变调用者中的名字）。

参数、返回类型和变量不用声明

像在Python中的所有东西一样，在函数上没有类型的约束。事实上，不需要事先对函数声明什么，我们在参数中可以传递任何类型，返回任何种类的对象，等等。因此，一个单一的函数通常能应用在大量的对象类型上。

下面让我们展开这些概念，看几个例子。

一般的形式

def 语句生成一个函数对象并赋值一个函数名。同所有复合的 Python 语句一样，它包含一个首行，跟着一个缩进的语句块。缩进的语句就是函数体 —— 每次函数被调用时所执行的 Python 代码。首行指定了一个函数名（赋值为函数对象），接着是一个参数（*argument*，有时也写作 *parameter*）列表，其赋值为调用时在括号中传递的对象：

```
def <名字>(arg1, arg2,... argN):
    <语句>
return <值>
```

Python 的 return 语句能在函数体内显示，它结束函数调用并传给调用者一个返回值。它包含了一个可以给出函数结果的对象表达式。return 是可选的；如果它不存在，控制流到函数体尾部时函数退出。严格地说，没有 return 语句的函数自动地返回一个 None 对象（更多的内容在本章的后面）。

定义和调用

让我们看一个例子，这实际上就是函数的两个方面:定义（def生成函数）和调用（一个表达式，告诉Python运行函数）。定义遵从上面的一般格式。此处定义了一个叫times的函数，它返回它的两个参数的积:

```
>>> def times (x, y):     # 生成并赋值函数
...     return x * y       # 在调用的时候函数体执行
...
```

当Python运行这个def时，它生成一个新的函数对象，打包函数的代码，并赋之以times的名字。在def已运行后，程序通过在函数名后添加括号运行（调用）该函数。函数括号内可以包含一个或多个对象参数选项，传递（赋值）给函数首部的名字：

```
>>> times (2, 4)          # 在括号内的参数
8
>>> times ('Ni', 4)       # 函数是“无类型的”
'NiNiNiNi'
```

在第一行，我们传递给times两个变量:函数首部的名字x被赋值为2，y赋值为1，函数体运行。在该例中，函数体是一个return语句，向调用表达式发送返回值8。

在第二个调用中，我们分别传递给x和y一个字符串和一个整数。还记得吗，*在数字和序列上都可以工作，因为在函数中不需要类型声明:你可以用times实现多个数字或重复序列。Python为人熟知的一点是就是动态类型。事实上，不同的时间，一个指定的名字可以赋值为不同类型对象，而不是由程序本身声明类型（注1）。

注1： 如果你曾用过如C/C++这样的编译语言，你可能会发现Python的动态类型使它成为一个难以置信的具有很大弹性的编程语言。这也意味着某些编译错误一直到程序运行的时候才会被Python捕捉到（例如，给一个整数加上一个字符串）。幸运的是，在Python中，错误很容易发现、修复。

范例:序列的交集

这里有一个描述函数基础的更实际的例子。在第三章“基本语句”的结尾处，我们看到一个 for 循环，用于收集在两个字符串中都含有的项。我们在那里提到，该段代码实际意义不大，因为只适用于特殊的变量，并且以后不会重新运行。当然了，你可以在需要它的地方剪切并粘贴代码，但这不是一个通用的解决办法：你不得不编辑每一份拷贝以支持不同的序列名，并且改变算法时需要修改多份拷贝。

定义

现在，你可能已想到，解决这个困境的办法就是在一个函数中打包循环。通过把代码放到函数中，它就变成了一个工具，喜欢的话你可以运行它任意次。并且通过允许调用者传递任意要处理的参数，使它充分通用，可以在任意两个你想求交集的序列上使用。从作用上讲，在函数中封装这些代码，生成了一个通用的求交集的程序：

```
def intersect (seq1, seq2):
    res = []                    # 开始为空
    for x in seq1:              # 扫描 seq1
        if x in seq2:           # 公共项?
            res.append (x)      # 添加到尾部
return res
```

把简单代码转换到这个函数非常容易，在 def 首部下嵌入原始逻辑并生成一个操作参数的对象。这个函数算出了一个结果，可以添加一个 return 语句发送返回值给调用者。

调用

```
>>> s1 = "SPAM"
>>> s2 = "SCAM"

>>> intersect (s1, s2)                  # 字符串
['S', 'A', 'M']

>>> intersect ([1, 2, 3], (1, 4))       # 混合类型
[1]
```

再说一次，我们传递不同类型的对象给函数——先是两个字符串，后是一个列表和一个元组（混合类型）。你无需事先指定参数类型，intersect函数很乐意地遍历你发送的任何种类序列对象（注2）。

函数中的作用域规则

既然已经开始写自己的函数，我们需要对Python中的名字含义有更正式的了解。当你在Python程序中使用一个名字时，Python在一个所谓的名字空间——名字所在的地方生成、改变名字。我们已知道Python中的名字在它们被赋值时就开始生成。因为无须事先声明，Python把名字的赋值与一个特定的名字空间相关联。除了打包代码，函数给你的程序添加了一个额外的名字空间层——默认时，函数内部赋值的名字与函数的名字空间关联。

具体工作方式如下。在你开始写函数之前，所有的代码都写在一个模块的顶层中，所以名字要么在模块本身中，要么是在Python预定义的内置名字空间中（如：open）（注3）。函数提供了一个嵌套的名字空间（有时称为作用域）。这样函数内部的名字不会与外部的（在模块或其他函数中的）冲突，我们通常说函数定义了一个局部作用域，模块定义了一个全局作用域。这两个作用域关系如下：

模块是一个全局作用域

每一个模块是一个全局作用域——一个名字空间，变量名在一个模块文件顶层生成（赋值）。

对函数的每个调用是一个新的局部作用域

每次你调用一个函数，生成一个新的局部作用域——是在函数的内部生成的名字所在的一个名字空间。

注2：严格的说，任何对象都对索引响应。for循环与in测试通过重复索引一个对象工作。当我们在第六章学习类的时候，你会看到如何实现用户定制的索引和迭代以及成员关系。

注3：记住，交互式命令行下输入的代码实际上是输入到一个叫__main__的内置模块中，所以交互生成的名字也是在一个模块中。在第五章“模块”中有关于模块的更多内容。

赋值的名字是局部的，除非声明是全局

默认情况下，在函数中赋值的名字都放到局部作用域中（函数调用相关的名字空间）。如果你需要赋值一个名字，而该名字又在包括该函数的模块顶层的话，你可以通过在一个函数中用一个global语句声明是全局的。

所有其他的名字都是全局的或内置的

函数定义中没赋值的名字均被认为是全局的（在包含它的模块的名字空间中）或内置的（在Python提供的预定义名字空间中）。

名字解析：LGB规则

上一节的内容学起来可能有些困惑，实际上可归纳为三个简单的规则：

- 大多数名字引用在三个作用域中查找：先局部，次之全局，再次之内置。
- 名字赋值默认情况下生成、改变局部的名字。
- “global”声明把赋值的名字映射到一个包含它的模块的作用域中。

换句话说，函数中的def语句里的所有赋值名字默认情况下都是局部的；函数可以使用全局作用域的名字，但必须声明为全局才可以改变它。Python的名字解析有时被称为*LGB*规则，*LGB*分别是作用域的名字：

- 当在函数中使用一个无限制型的名字时，Python查找三个作用域，局部的（Local，L），次之全局（Global，G），再次之内置的（Built-in，B）——在名字被发现的第一个位置处停止。
- 当你在一个函数中赋值一个名字（而不是只在一个表达式中引用它）时，Python总是在局部作用域中生成或改变它，除非在该函数已对它进行了全局声明。
- 当在函数外部（例如，在一个模块顶层或交互提示符下）时，局部作用域与全局作用域一样——都是一个模块的名字空间。

图4-1描述了Python的三个作用域。作为一个预览，我们还想告诉你，这个规则只适用于简单的名字（如spam）。在下两章中，我们会看到用在特定对象中的限制型名字的规则（如object.spam，称之为属性）。

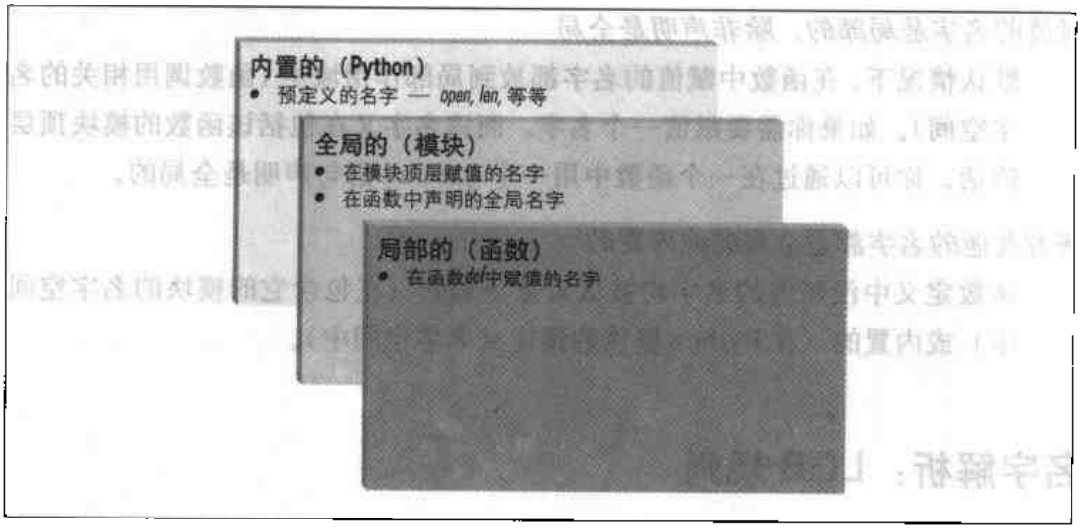


图 4-1 LGB 作用域查找规则

范例

让我们看一个展示作用域概念的例子，假定我们在一个模块文件中写下如下代码：

```
# 全局作用域
X = 99                  # X和func在模块中被赋值：全局的

def func(Y):            # Y和Z在函数中被赋值：局部的
    # Local scope
    Z = X + Y           # X没被赋值，所以它是全局的
    return Z

func(1)                 # func在模块中：结果=100
```

这个模块以及它包含的函数，用了一些名字来实现其功能。使用Python的作用域规则，我们可以对名字分类如下：

全局名字：X，func

X是全局的，因为它在模块文件顶层赋值；它可以在函数内部被引用，无须事先声明全局性。同样原因，func也是全局的，def语句在模块的顶层给func赋值了一个函数对象。

局部名字：X，Z

X和Z对函数来说是局部的（只在函数运行时才存在），因为它们都在函数定义时被赋给一个值；Z是因为=语句，X则是因为参数总是被赋值传递。（稍后会进行更详细的介绍。）

这种名字分类模式的要点就是，局部变量作为临时名字只在函数运行时你才需要。例如，参数X和结果Z只在函数内部存在，它们并不妨碍包含它的模块的名字空间（或者其他函数）。局部/全局的区别也使函数更易于理解；大多数名字只在函数本身内部出现，而不是在模块中的任意位置（注4）。

global语句

global语句就像Python中的一个声明，它告诉Python，一个函数计划改变全局名字——包围它的模块的作用域（名字空间）内的名字。我们以前已经谈过global了，现在总结一下：

- global意味着“一个在模块文件顶层的名字”。
- 全局名字只在函数中要被赋值时才必须被声明。
- 全局名字可以在函数中无须声明即被引用。

global语句由关键词global和一个由逗号分开的名字组成，所有的列表名字在函数体内部被赋值或索引时会被映射到包含它们的模块作用域上。例如：

```
y, z = 1, 2                    # 模块中的全局变量

def all_global():
    global x                   # 声明全局的赋值
    x = y + z                  # 无需声明y, z：3作用域规则
```

在这里，x，y，z都是函数all_global中的全局名字。y和z是全局的，因为它们在函数中没被赋值，x是全局的，因为我们说过，我们在global语句中列出它

注4：细心的读者会发现，正是因为LGB规则，本地作用域内的名字会忽略全局与内置作用域内相同名字的变量，全局的会忽略内置的。例如，一个函数能生成一个叫open的本地变量，但它会隐藏存在于内置（外部）作用域内的叫做open的内置函数。

可以显式的映射到模块的作用域中。若在这里不用global,x会被认为是局部的。注意，y和z没有被声明为全局的；Python使用LGB查找规则在模块中自动的发现它们。还要注意在函数运行前，x也许在包含它的模块中不存在，如果不存在，函数中的赋值将在模块中生成x。

参数传递

让我们详细叙述一下Python中参数传递的概念。早些时候，我们提到过参数是通过赋值传递的，这有一些歧义，对初学者来说不是很明白。

参数通过局部名字传递

函数参数现在已经是我们熟悉的领域了:它们只是工作中Python赋值的又一个例子而已。函数参数（可能）是对共享对象的引用，被调用者所引用。

在一个函数中对参数名赋值不影响调用者

函数首部中的参数名当函数运行时在作用域中成为新的局部名字。函数的参数名字不是调用者使用的名字的别名。

在一个函数中改变一个可变的对象参数可能影响调用者

在另一方面，既然参数只是简单的赋值给对象，函数能改变被传递的可变对象，结果可能影响调用者。

这里有一个例子，描述了这些属性:

```
>>> def changer (x, y):
...     x = 2               # 只改变局部的名字值
...     y[0] = 'spam'       # 在该位置改变共享的对象
...
>>> X = 1
>>> L = [1, 2]
>>> changer (X, L)          # 传递可变的和不可变的
>>> X, L                    # X没改变，L改变了
(1, ['spam', 2])
```

在这段代码中，changer函数对参数名x和被参数y引用的对象中的组件赋值。既然x是个函数作用域中的局部名字，第一个赋值对调用者不影响，并且没有在

调用中改变X的绑定。参数y也是个局部名字，但它传递了一个可变对象（在调用中的叫L的列表）；对在函数中对y[0]赋值的结果影响了函数返回后L的值，图4-2描绘了名字对象绑定在函数调用后的情况。

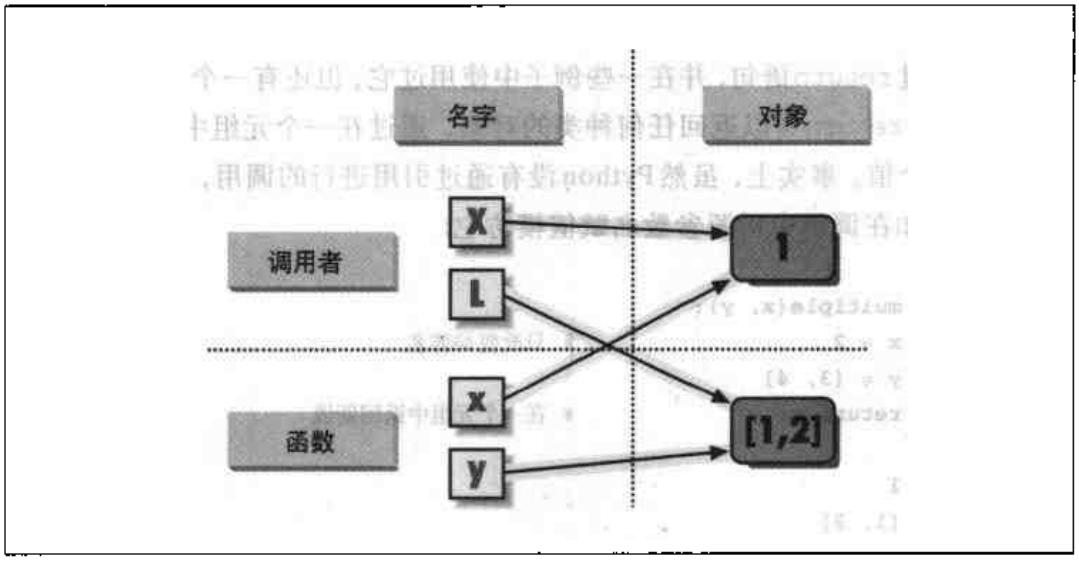


图4-2 引用：参数和调用者共享对象

如果你回忆一下在第二章“类型和操作符”中关于共享的可变对象的讨论，你会发现这恰好是在工作中同样的现象:在原位改变一个可变的对象能影响其他相关的引用。在这儿，它的影响是使一个参数成为函数的输出。(为了避免这一点，输入y=y[:]，生成一份拷贝。)

Python通过赋值传递参数的方式，与C++中的参数引用不同，但实际中它与C的较为类似:

不可变的参数作用类似于C中“传值”模式

类似整数和字符串这样的对象，通过对象引用（赋值）传递。不过既然你无法改变不可变对象，该作用有些像一个拷贝。

可变参数作用类似于C的“传指针”模式

像列表和字典这样的对象也通过对象引用传递，这与C中的把数组作为指针传递很像 —— 可变对象在函数中可以改变，特别像C语言中的数组。

当然了，若你从未用过C，Python的参数传递模式很简单:它只是一个把对象赋值给一个名字，无论对象可变与否它都一样工作。

关于return的更多内容

我们已讨论过return语句，并在一些例子中使用过它。但还有一个我们还没提及的技巧：因为return可以返回任何种类的对象，通过在一个元组中打包它们，它可以返回多个值。事实上，虽然Python没有通过引用进行的调用，但我们可以通过返回元组和在调用中对源参数名赋值模仿它:

```
>>> def multiple(x, y):
...     x = 2                       # 只改变局部名
...     y = [3, 4]
...     return x, y                 # 在一个元组中返回新值
...
>>> X = 1
>>> L = [1, 2]
>>> X, L = multiple(X, L)# 对调用者的名字赋值结果
>>> X, L
(2, [3, 4])
```

看上去，我们在这里有两个返回值，但只有一个——一个两项的元组带着一个被忽略的括号，如果你忘了原因，回到第二章的元组部分看看吧。

特殊参数匹配模式

虽然参数总是被赋值传递，Python提供了额外的工具改变在调用中的参数对象和首部的参数名配对的方法。默认情况下，它们通过位置匹配，从左到右，并且在函数首部的参数你必须都传递。不过你也可以通过名字、默认值和额外参数的集合来指定一个匹配。

这一节中的有些内容较复杂，在我们学习更多的语法细节之前，我们想强调这些特殊模式都是可选的，只在不得不处理对象名字匹配时使用。在匹配发生后，传递的机制还是赋值。但作为一个介绍，下面是允许的匹配模式的一个摘要。

位置：匹配是从左到右的

我们通常遇到的正常情况都是通过位置匹配参数。

关键字：通过参数名匹配

通过在调用中使用参数的名字，调用者能指定函数中由哪个参数接收值。

可变参数（varargs）：捕捉未匹配位置或关键字参数

函数能使用特殊的参数来收集任意多的额外参数（特别像C中的varargs，它支持可变长参数列表）。

默认情况：参数的指定值不被传递

如果调用者传递了太少的值的话，函数也可作为参数指定默认值。

表4-2总结了指定特殊匹配模式的语法。

表4-2　函数参数匹配形式

语法	位置	解释
func(value)	调用者	正常参数：通过位置匹配
func (name=value)	调用者	关键字参数：通过名字匹配
def func (name)	函数	正常参数：匹配任何位置和名字
def func (name=value)	函数	默认的参数值，如果没有在调用中传递的话
def func (*name)	函数	匹配剩下的位置变量（在一个元组中）
def func (**name)	函数	匹配剩下的关键字变量（在一个字典中）

在调用者（表中前两列）中，简单名字通过位置匹配。但使用“名字=值(name=value)”的形式告诉Python通过名字匹配；这些称为关键字参数。

在函数首部，简单名字通过位置或名字（取决于使用者如何传递它）匹配。但“name=value”的形式指定了一个默认值。“*name”在一个元组中收集了任何其他位置上的参数，“**name”的形式在字典中收集了其他的关键字参数。

因此，特殊匹配模式允许你真正的自上的决定在函数中必须被传递的参数的个数。如果一个函数指定默认值，只有传递的参数不够时它们才会被使用。如果一个函数使用可变参数形式，你可以传递很多的参数，可变参数名字把额外的参数收集到一个数据结构中。

第一个例子

让我们看一个例子，在实际中示范一下关键字和默认情况。其中调用者必须总是传递至少两个参数（以匹配spam和eggs），下过其他的两个都是可选的；如果忽略的话，Python赋给toast和ham在首部指定的默认值：

```
def func(spam, eggs, toast=0, ham=0):        # 头两个是必需的
    print (spam, eggs, toast, ham)
func(1, 2)                                   # 输出：(1, 2, 0, 0)
func(1, ham=1, eggs=0)                       # 输出：(1, 0, 0, 1)
func(spam=1, eggs=0)                         # 输出：(1, 0, 0, 0)
tunc(toast=1, eggs=2, spam=3)                # 输出：(3, 2, 1, 0)
func(1, 2, 3, 4)                             # 输出：(1, 2, 3, 4)
```

注意，当关键字参数在调用中使用时，参数列出的顺序无关紧要；Python通过名字而不是位置匹配，调用者必须为spam和eggs提供值，但它们可以通过位置或名字匹配。也要注意“name=value”的形式在调用和def中有不同的含义：一个是调用中的关键字，而另一个是在首部的默认值。

第二个例子：任意参数的集合函数

这里有一个更有用的特殊参数匹配模式的例子。在这一章前面，我们写过一个函数可以返回两个序列的交集（选出二者共有的项）。这里有一个版本，可以求任意多个序列的交集（1或多个），通过使用可变参数匹配形式*args收集所有的参数。因为参数是作为一个元组，我们可以用一个简单的for来处理它。只是为了有趣，我们还写了一个任意数参数的union函数，求并集：

```
def intersect(*args):
    res = []
    for x in args[0]:                    # 扫描第一个序列
        for other in args[1:]:           # 所有其他的序列
            if x not in other: break     # 项存在每一个序列中？
        else:                            # 否：跳出循环
            res.append(x)                # 是：添加项到尾部
    return res

def union(*args):
    res = []
    for seq in args:                     # 所有的参数
        for x in seq:                    # 所有的节点
```

```
            if not x in res:
                res.append(x)          # 添加新的项到结果中
    return res
```

因为这些工具值得再次使用(并且太大了不宜交互地重新输入),我们在一个模块中存储我们的函数：*inter2.Py*（关于模块的更多细节在第五章“模块”中)。在两个函数中，在调用中传递的参数的形式是args元组。同最初的intersect一样，它们都可以在任何种类的序列上工作。在这里，它们处理了字符串、混合类型，以及多于两个的序列。

```
% python
>>> from inter2 import intersect, union
>>> s1, s2, s3 = "SPAM", "SCAM", "SLAM"
>>> intersect(s1, s2), union(s1, s2)              # 2个操作数
(['S', 'A', 'M'], ['S', 'P', 'A', 'M', 'C'])

>>> intersect([1,2,3], (1,4))                     # 混合类型
[1]

>>> intersect(s1, s2, s3)                         # 3个操作数
['S', 'A', 'M']

>>> union(s1, s2, s3)
['S', 'P', 'A', 'M', 'C','L']
```

进一步的细节

若你要使用特定的匹配模式的组合，Python 有两个排序规则:

- 在调用中，关键字参数必须出现在所有非关键字参数后。
- 在一个函数首部中，*name必须在普通参数和默认值后，**name必须在最后。

而且，Python 内部在赋值前采取了下列步骤匹配参数:

1. 通过位置赋值非关键字参数
2. 通过匹配名字赋值关键字参数
3. 赋值额外的非关键字参数给 *name 元组
4. 赋值额外的关键字参数给 **name 字典

5. 在首部给无赋值的参数赋以默认值

这看上去十分复杂。不过描述一下匹配算法可能会使我们加深理解，尤其是混合模式。在做本章结尾的练习之前，我们不再介绍关于这些特殊匹配模式的其他的例子。

你会知道，高级参数匹配模式可能是复杂的。它们完全都是可选的，你可以只用简单的位置匹配，刚开始时这可能是一个好主意。不管怎样，一些Python工具用到了它们，它们的重要性需要知道。

请留意:关键字参数

关键字参数在Tkinter中扮演了一个重要的角色。Tkinter是Python中事实标准的GUI API。在第十章“框架与应用”我们会遇到Tkinter，但作为一个预览，关键字参数在GUI组件组建时设定设置选项。例如，一个调用形式:

```
from Tkinter import *
widget = button (text="Press me", command=someFuntion)
```

用text和command关键词生成一个新的按钮，并指定它的文本和回调函数，因为组件的设置选项数可以很大，关键字参数允许你挑取、选择。没有它们，你可能不得不通过位置列出所有的可能选项或希望有一个明确的位置参数默认协议可以处理每一个可能的选项参数。

其他内容

到现在我们已看到在Python中如何写函数，在这一节中我们想介绍一些其他的概念。

- lambda生成匿名函数。
- apply用参数元组调用函数。
- map对一个序列运行一个函数，并收集结果。
- 如果不用一个return的话，函数返回一个None值。

- 函数表示设计的选择。
- 函数是对象，就像数字和字符串一样。

lambda 表达式

除了def语句，Python还提供了一个可以生成函数对象的表达式。因为它与LISP语言中的一个工具类似，所以叫lambda，一般的形式是关键字lambda，后面跟有一个或多个参数，在一个括号后跟随一个表达式：

```
lambda 参数1，参数2，...参数N：使用参数的表达式
```

被lambda返回的对象同def赋值生成的对象是一样的，但lambda略有些不同，在特定场合中它十分有用：

lambda是一个表达式，不是一个语句

正因如此，lambda可以出现在一个def语句所不能出现的地方——如在一个列表常量内部。作为一个表达式，lambda返回一个值（一个新的函数），它可以被赋给一个名字。def语句总是对首部的名字赋一个新的函数，而不是作为结果返回它。

lambda的结构体是一个单一表达式，不是一个语句块

lambda的结构体与放入def结构体中的return语句很相像。只返回一个表达式的结果，而不是表达式本身。正因它受表达式的限制，lambda不如def这样具有一般性，不能使用像if这样的语句，你只能在一个lambda的结构体中压入逻辑。

抛开这些区分不谈，def和lambda做的是类似的工作。例如，我们已看到如过你使用def生成语句：

```
>>> def func (x, y, z): return x + y + z
...
>>> func (2, 3, 4)
9
```

你也可以用一个lambda表达式达到同样效果，通过对它的名字明确的赋值：

```
>>> f = lambda x, y, z: x + y + z
>>> f(2, 3, 4)
9
```

在这里，f被赋值为lambda表达式生成的函数对象（这是def的工作方式，但赋值是自动的）。默认情况下在lambda参数下也工作，就像def一样：

```
>>> apply(func, (2, 3, 4))
9
>>> apply(f, (2, 3, 4))
9
```

lambda对函数速记很方便，如，在后面我们将见到回调处理，它通常被编码为lambda表达式。直接在注册调用中嵌入，而不是在文件的其他地方定义并通过名字引用。

请留意:lambda

当你要在某处填入一小段可执行代码（在该处语句在语法上是非法的）时，lambda表达式作为def语句的一个简写形式是很方便的。例如，你可以通过在一个列表常量中嵌入lambda表达式，建立一个函数的列表：

```
L = [lambda x: x**2, lambda x: x**3, lambda x: x**4]

for f in L:
    print f(2)      # 打印4, 8, 16

print L[0](3)       # 打印9
```

不使用lambda，你就不得不在列表外部使用三个def语句，在函数中这些定义将被使用。lambda处理函数参数列表也很方便;它的一个很常见的应用是为Tkinter API定义内嵌回调函数（更多的关于Tkinter的内容在第十章）。下面的代码处理后在控制台上生成一个按钮，被按下时打印信息：

```
import sys
widget = Button(text    ="Press me"
                command = lambda: sys.stdout.write("Hello world\n"))
```

内置的 apply 函数

有的程序需要调用任意的函数，而无须事先知道它们的名字或参数。随着我们的介绍，我们将看到一些很有用的例子。内置函数 apply 可以做到这一点。例如，在运行上一节中的代码之后，你可以通过把生成的函数作为apply的参数来调用，用一个参数元组：

```
>>> apply(func, (2, 3, 4))
9
>>> apply(f, (2, 3, 4))
9
```

apply 函数简单的调用传递的参数，用函数所需的参数匹配传递的参数。既然参数列表作为一个元组（一个数据结构）传递，它可以在运行时被计算。apply 的一个真正优点是，它无须知道在一个函数中有多个参数被调用，例如，你可以用 if 从一系列函数和参数列表中选择，并使用 apply 调用任何一个：

```
if <test>:
    action, args = func1, (1,)
else:
    action, args = func2, (1, 2, 3)
. . .
apply(action, args)
```

内置函数 map

程序通常可以做的一件事是：对列表中的每一个成员进行一个操作并收集结果。例如，更新一个列表中的所有数据，可以用一个 for 循环轻松实现：

```
>>> counters = [1, 2, 3, 4]
>>>
>>> updated = []
>>> for x in counters:
...     updated.append(x + 10)                # 给每一项加上 10
...
>>> updated
[11, 12, 13, 14]
```

正因为这是一个如此常见的操作，Python 提供的内置工具为你做了大部分工作：

map函数对在一个常列对象中的每一项都调用了一个传递的函数，并返回一个包含所有调用结果的列表，例如:

```
>>> def inc(x): return x + 10                    # 要运行的函数
...
>>> map(inc, counters)                           # 收集结果
[11, 12, 13, 14]
```

既然map处理的是函数，有时lambda也会在其中出现:

```
>>> map((lambda x: x + 3), counters)             # 函数表达式
[4, 5, 6, 7]
```

map是Python用于函数化编程的一类内置工具的最简单的代表。它的相关工具有：基于一个测试条件的过滤（filter），对成对的项进行的操作（reduce）（对序列使用函数的工具）。在第八章“内置工具”中我们会详细的讨论这些内置工具。

Python的过程

在Python函数中，return语句是可选的。当一个函数无法准确的返回值时，函数就跳至尾部并退出。严格地说，所有的函数都返回一个值，若你不能提供一个返回值，你的函数自动返回None对象:

```
>>> def proc(x):
...     print x          # 没用return语句，返回值就是None
...
>>> x = proc('testing 123...')
testing 123...
>>> print x
None
```

这种不带return的函数与其他语言（如Pascal）中所说的过程是等效的。既然它们可执行自己的功能而无需产生有用的结果，所以通常作为一个语句被调用（None结果被忽略了）。了解这一点是很值得的，因为在你试图使用一个无返回值函数的结果时，Python不会告诉你。例如，给一个列表append方法赋值不会出错，但你将得到一个None，而不是被修改过的列表:

```
>>> list = [1, 2, 3]
>>> list = list.append(4)        # append是一个"过程"
>>> print list                   # append在原位修改列表
None
```

函数设计概念

当你开始使用函数时，你就需要选择如何把它们组织在一起——例如，如果你把一个任务分解成函数，函数如何通信等等。这些中的一部分属于结构分析和设计的内容。相对这本书来说，这个主题太宽泛了。但仍有一些对Python初学者通用的提示：

输入用参数，输出用return语句

一般的说，你应该尽力使函数独立于外部的事物，参数和return语句通常是使其具有独立性的最好方法。

只在绝对必要时才用全局变量

全局变量通常是对函数通信的一种很不好的工具，它们会产生依赖性，使程序不易修改。

不要改变参数除非调用者期望这样做

函数也可以改变传送的部分可变对象，但作为全局变量，它包含了许多调用者和被调用者之间的联合，这容易生成十分特殊、粗糙的函数。

表4-3总结了函数与外部对话的方式。左边栏中的项用于输入结果，结果可以是右边的任何形式。正确的函数设计者通常只把参数用于输入，return语句用以输出，但这有很多例外，包括Python的面向对象编程支持——我们在第六章"类"中会见到，Python的类依赖于改变一个传入的可变对象。类函数设置一个自动传递的self对象的属性，以改变该对象的状态信息（如self.name ='bob'）；若想这样做的话，副作用就并不危险。

表4-3 常见的函数输入与输出

函数输入	函数输出
参数	返回语句
全局（模块）变量	可变的参数
文件，流	全局的（模块）变量

函数是对象：非直接调用

因为函数在运行时是对象，你一般可以写程序处理它们。函数对象可以被赋值，传递给其他函数，在数据结构中存储等等，就像它们是简单的数字或字符串一样。函数对象有时输出一个特定的操作，它们可以通过一个函数表达式括号中的列表参数来调用，不过函数和其他对象都属同一个通用类别。

例如，正如我们已经看到的，关于我们在 def 中使用的语句真的没什么特别的：它只是一个在当前作用域赋值的变量。就像它已在等号左边出现一样，在 def 运行后，函数名字就是一个对象的引用，你可以重赋值该对象到其他名字，并通过任意引用调用它 —— 不只是初始名字。

```
>>> def echo(message):              # 回显对一个对象的赋值
...     print message
...
>>> x = echo                        # 现在 x 也引用它
>>> x('Hello world!')               # 通过添加()调用对象
Hello world!
```

既然参数通过赋值传递，传递函数参数同传其他参数一样简单。被调用者可以在括号中添加参数来调用传递的函数：

```
>>> def indirect(func, arg):
...     func(arg)                        # 通过添加()来调用对象
...
>>> indirect(echo, 'Hello jello!')       # 传递函数到一个函数
Hello jello!
```

你甚至可以把函数对象放到数据结构中，就像它们是整数或字符串一样。既然，Python 复合类型能包含任何种类的对象，也没什么例外的：

```
>>> schedule = [ (echo, 'Spam!'), (echo, 'Ham!')]
>>> for (func, arg) in schedule:
...     apply(func, (arg,))
...
Spam!
Ham!
```

这段代码简单的遍历了 schedule 表，每次用一个参数调用 echo 函数。我们希望

你已从现在开始注意，Python缺少类型声明这一特点，使它成为一个十分奇妙的有惊人的弹性的语言。请注意apply调用函数的一般用法，apply函数中的第二个单项元组参数，以及在for循环的元组解析赋值（所有概念在前面已介绍了）。

函数的常见问题

这里有一些你不希望见到的关于函数的问题，它们都较难于理解，尤其对新手而言。

局部名字静态检测

我们已看到，默认情况下，Python把函数中赋值的名字归类为局部的。它们在函数的作用域中，并只在函数运行时才存在。我们没告诉你的是Python如何静态地探测局部作用域：是在Python编译代码时检测，而不是通过它们在运行时的赋值。通常情况，我们不注意这些，不过这容易把我们导向一个常见的困惑，新手总在Python新闻组上提出这些问题。

正常情况下，没在函数中赋值的名字将在包含它的模块中查找：

```
>>> X = 99
>>> def selector():        # X被使用了但是还没有赋值
...     print X            # X在一个全局作用域被发现
...
>>> selector()
99
```

在这里，函数中的X解析为外部模块中的X。不过，如果在引用后对X添加一个赋值，看看会发生什么？

```
>>> def selector():
...     print X            # 还没有存在!
...     X = 88             # X被归类为局部名字（任何地方）
...                        # "import X", "def X", ...同样可以发生
>>> selector()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
```

```
  File '<stdin>", line 2, in selector
NameError: X
```

你会得到一个未定义的名字错误，不过原因是微妙的。当交互式输入或从一个模块中导入时，Python读取并编译这段代码。当编译时Python查看X的赋值，并决定在函数中任何地方X都将是个局部名字。到后来函数真的运行，print执行时，赋值还没发生，Python会说你正使用一个未定义的名字。根据它的名字规则，应该是这样，局部的X在赋值前被使用了（注5）。

解决办法

这个问题的发生是因为，赋值名字在函数中的任何地方都被当作局部的，而不是在名字赋值后的语句后面。事实上，上面的代码是十分歧义的，你是想打印全局的X再生成一个局部的X，或这的确是一个编码错误？既然Python在任何地方都把X当作一个局部赋值，这就是一个错误；但如果你真想打印全局X，你应该在一个global语句中声明它：

```
>>> def selector():
...     global X              # 强迫X成为全局的（任何地方）
...     print X
...     X = 88
...
>>> selector()
99
```

记住，这意味着该赋值也改变全局赋值X，而不是局部的X。在一个函数中，你不能对同一个简单的名字同时使用局部和全局的形式。如果你真想打印出全局赋值，再设定一个局部的，导入包含它的模块并用限定得到这个全局的版本：

```
>>> X = 99
>>> def selector():
...     import __main__          # 导入包含它的模块
...     print __main__.X         # 限定型的得到全局版本的名字
...     X = 88                   # 非限定型的X被归类为局部的
...     print X                  # 打印局部版本的名字
```

注5：实际上，在一个函数体中的任何赋值会生成一个本地的名字：import，=，嵌套的def，嵌套的类，等等。

```
...
>>> selector()
99
88
```

限定（.X部分）从一个名字空间对象中得到一个值。交互环境的名字空间是一个叫__main__的模块，因此__main__.X得到X的全局形式，若不清楚可以参见第五章。

嵌套函数不是嵌套的作用域

正如我们已看到的，Python的def是一个可执行语句：当它执行时它把一个新的函数对象赋给一个名字。因为它是一个语句，它可以出现在一个语句的任意位置——甚至在其他语句中嵌套。例如，在一个if语句中，嵌套一个def函数是完全合法的，在可选的定义间选择：

```
if test:
    def func():          # 用这种方式定义 func
        ...
else:
    def func():          # 或者用这种方式代替
        ...
...
func()
```

理解这段代码的一个方式是要知道def与一个“=”语句特别像：它在运行时赋值了一个名字。与C不同，Python函数在运行前无需全部定义。既然def是一个可执行语句，它也可以在其他def中嵌套。不过与Pascal这样的语言不同，嵌套的def不包含Python中嵌套的作用域。例如，这个例子定义的一个函数（outer），它定义并调用另一个函数（inter），递归调用自身（注6）：

注6：这里的“递归”是指函数在一个优先调用生成之前、一再地被调用。在这个例子中，函数调用自身，不过也可以调用另外调用它的函数等等。在这里递归可以被一个简单的while或是for循环代替（我们做的是一直向下数到0），不过我们试图阐明关于自递归函数名字与嵌套。在你写一个程序的时候，当一个数据结构的情况无法预测时使用递归会很有用。

```
>>> def outer(x):
...     def inner(i):              # 在outer的局部赋值
...         print i,               # i在inner的局部
...         if i: inner(i-1)       # 不在局部或者全局!
...     inner(x)
...
>>> outer(3)
3
Traceback (innermost last):
  File '<stdin>', line 1, in ?
  File '<stdin>', line 5, in outer
  File '<stdin>', line 4, in inner
NameError: inner
```

这不会工作的。一个嵌套的def实际上只包含在它的函数作用域名字空间中，对一个名字赋一个新的函数对象。在嵌套函数内部，LGB作用域规则仍对所有的名字适用。嵌套的函数对自己的局部作用域，包含自己的模块的全局作用域以及内置的名字作用域有访问权，它对包含自己的函数的作用域无访问权。无论多深的函数嵌套，每个只能看见三个作用域。

例如，在上面的例子中，嵌套的def在outer函数的局部作用域生成名字inter（像在outer中的任何其他赋值）。但在inter函数内部名字inter是不可见的，它不在inter的局部作用域中，不在包含它的模块的作用域中，当然也不在内置的作用域中。因为inter没有对outer作用域的访问权，从inter对inter的调用失败，并出现一个异常。

解决办法

不要指望作用域在Python中嵌套，这的确很难理解。def语句只是一个对象构造者，而不是作用域嵌套者。不管怎样，如果你真需要从嵌套函数inter中对嵌套的函数名字访问的话，简单的在outer中用一个全局声明，强迫嵌套的函数名字跳出包含它的模块的作用域，这可依据LGB规则：

```
>>> def outer(x):
...     global inner
...     def inner(i):              # 在包含它的作用域中赋值
...         print i,
...         if i: inner(i-1)       # 现在在全局作用域中可以发现
```

```
...     inner(x)
...
>>> outer(3)
3 2 1 0
```

使用默认值保存引用

事实上嵌套的函数对包含它的函数中的任何名字都没有访问权，这样通常会比上面的例子产生更多的问题。要得到对嵌套函数的def前赋值的名字的访问权，你还可以把它作为嵌套函数的默认参数。因为默认参数在def运行时保存它们的值，而不是当函数被实际调用时，它们能从包含它的函数中去掉对象：

```
>>> def outer(x, y):
...     def inner(a=x, b=y):    # 保存outer的x,y的绑定/对象
...         return a**b         # 在这里不能直接使用x、y
...     return inner
...
>>> x = outer(2, 4)
>>> x()
16
```

在这里，一个对outer函数的调用返回被嵌套的def生成的新的函数。当嵌套的def语句运行时，inter的参数a和b从outer函数的局部作用域赋值为x和y。在后面inter的函数体中a和b被使用时，在outer运行时它们仍指向x和y（即使outer已返回它的调用者）（注7）。这对lambda也适用，因为lambda只是def的一种简化形式：

```
>>> def outer(x, y):
...     return lambda a=x, b=y: a**b
...
>>> y = outer(2, 5)
>>> y()
32
```

注7：用计算机的行话来说，这类行为通常被称为*closure*（闭包）——一个对象记得包含它的作用域的值，即使这些作用域已经不再包含它了。在Python中，你需要确切的列出哪个值应该被记住，使用默认的参数（或者是类对象属性，在第六章我们会见到）。

注意默认赋值在上节的例子中不适用这样的技巧，因为名字inter直到内部的def完成时才被赋值。当嵌套函数调用自身时，全局声明可能是最好的工作方式：

```
>>> def outer(x):
...     def inner(i, self=inner):      # 名字还没定义
...         print i,
...         if i: self(i-1)
...     inner(x)
...
>>> outer(3)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in outer
NameError: inner
```

不过如果你正对钻研Python晦涩的部分有兴趣，你也可以默认保存一个可变的对象，并在包含它的函数体中插入一个对 inner 的引用：

```
>>> def outer(x):
...     fillin = [None]
...     def inner(i, self=fillin):    # 保存可变的
...         print i,
...         if i: self[0](i-1)        # 假定它是集合
...     fillin[0] = inner             # 现在插入值
...     inner(x)
...
>>> outer(3)
3 2 1 0
```

虽然这段代码描述了Python的属性，但我们不推荐这样做。在这个例子中费了很大的劲儿避免函数在一起嵌套：

```
>>> def inner(i):                    # 定义模块层次的名字
...     print i,
...     if i: inner(i-1)              # 不用担心：它是全局的
...
>>> def outer(x):
...     inner(x)
...
>>> outer(3)
3 2 1 0
```

这是一条实践原则：最简单的方式总是最正确的方式。

默认的可变对象

默认参数值是在def语句运行时计算保存的，而不是在函数被调用时。正如我们刚看到的那样，既然允许你在包含它的作用域中保存值，这就是你想要的。不过既然默认对象在调用中仍是一个对象，在改变不可变的默认对象的时候，你不得不小心。例如，下面的函数用一个空列表作为默认值，并在每次函数被调用时都在原位改变它：

```
>>> def saver(x=[]):                # 保存起一个列表对象
...     x.append(1)                 # 每次改变同样的对象
...     print x
...
>>> saver([2])                      # 默认的没有使用
[2, 1]
>>> saver()                         # 默认的使用
[1]
>>> saver()                         # 每次调用后生长
[1, 1]
>>> saver()
[1, 1, 1]
```

问题是，这里只有一个列表对象——def执行时生成的一个。在每一次函数被调用时，你不会得到新的列表对象，每个append后这个列表都要增长。

解决办法

如果那不是你想要的行为，简单的移动默认值到函数体中。只要代码里的值在每一次函数运行时都执行，你每次将得到一个新的对象：

```
>>> def saver(x=None):
...     if x is None:               # 没有参数传递?
...         x = []                  # 运行代码生成一个新的列表
...     x.append(1)                 # 改变新的列表对象
...     print x
...
>>> saver([2])
[2, 1]
```

```
>>> saver()                    # 在这里不增长
[1]
>>> saver()
[1]
```

顺便说一下，上面的if语句几乎可以被赋值x = x or []代替，因为Python的or将返回它的操作对象中的一个:如果没有参数被传递，x默认为None，所以or在右侧返回一个生成的空列表。但这不完全一样：当传递的是空列表时，函数将扩展并返回一个新生成的列表，而不是像前面的版本那样扩展并返回一个被传递的列表。(表达式变成[] or {}，这将计算出新的列表:若你真的没想起为什么，看一下第三章中关于真值测试的部分。)既然实际的编程需要可能调用任何一种行为，我们不在这里讲孰优孰劣。

总结

在这一章中，我们已学习了如何书写并调用你自己的函数，我们也讨论了作用域、名字空间问题，谈到了参数传递，看了些像lambda和map这样的函数工具，并学习了新的与函数相关的语句def, return, global。我们也简略说了一些如何连接函数的问题，看到了些可能迷惑新手的常见的函数问题。在第五章，我们将学习模块如何把你的函数集合成相关的工具包。

练习

在这一练习中，我们将开始编写些复杂的程序。如你被卡住了就看一下附录三。同时要确保在模块和文件中写你的代码，你应该不想因一点点错误就从头做起吧？！

1. **基础**。在Python交互提示符下写一个函数，在屏幕上打印它的单一参数，并交互式调用它。传递多种对象类型:字符串、整数、列表、字典。再试着不用任何参数调用它，看看发生了什么?当你传递两个参数时发生了什么?
2. **参数**。在Python模块文件中写一个叫adder的函数：adder应该可以接受两个参数并返回两个参数的和（或合并值）。再在文件底部添加代码，用多种

对象类型调用它（两个字符串、两个列表、两个浮点数）。从系统命令行以脚本方式运行该文件，要在屏幕上查看结果，你非得打印调用语句的结果吗？

3. **可变参数。**扩展你在上一个练习中写的 adder 函数：可以计算任意个数的参数，改变调用以传递多于或少于两个参数。返回和的值是什么类型？（提示：一个像 S[:0]这样的分片返回与 S 同类型的一个空的序列，内置 type 函数可以测试类型。）如果你传递不同类型参数会发生什么？传递字典会怎样？

4. **关键字。**改变练习 2 的函数 adder，可以接受并增加三个参数：def adder(good, bad, ugly)。现在对每个参数提供默认值，并在交互模式下实验调用该函数，试着传递一个、二个、三个、四个参数，接着试着传递关键字参数。adder(ugly=1, good=2)。这个调用工作吗？为什么？最后，写一个新的 adder，可以接受并对任意数目的关键字参数求和，很像练习 3，不过你需要通过一个字典迭代，而不是一个元组。（提示：dictionary.keys()方法返回一个列表，你可以用一个 for 或者 while 循环遍历它。）

5. 写一个 copyDict(dict)的函数，拷贝它的字典参数。它应该返回一个包含所有项的新字典。用字典的 keys 方法遍历。拷贝序列是简单的（X[:]可以进行一个顶层拷贝），这对字典也适用吗？

6. 写一个函数 addDict(dict1, dict2)计算两个字典的集合，它返回一个新字典，包含所有的参数中的项（假设都是字典）。如果同样的键出现在两个参数中，从任何一个中随便地选一个值。把函数写在一个文件中，作为脚本运行，测试它。如果你传递列表而不是字典会发生什么？如何实现函数以处理这种情况？（提示：看一下早先使用内置函数 type。）参数传递顺序重要吗？

7. **更多参数匹配的例子。**首先定义下面六个函数（交互式或在一个可导入的文件模块中都行）：

```
def f1 (a, b): print a, b              # 正常的参数
def f2 (a, *b): print a, b             # 位置上的可变参数
def f3 (a, **b): print a, b            # 关键字可变参数
def f4 (a, *b, **c): print a, b, c     # 混合模式
def f5 (a, b=2, c=3): print a, b, c    # 默认的
def f6 (a, b=2, *c): print a, b, c     # 默认情况 - 位置上的可变参数
```

现在，交互的测试下列调用并试着解释每个结果；在一些情况下你可能要复习一下本章提到的匹配算法，你认为一般情况下，混合的匹配模式是一个好主意吗?你认为在什么情况下它才会很有用?

```
>>> f1(1, 2)
>>> f1(b=2, a=1)

>>> f2(1, 2, 3)
>>> f3(1, x=2, y=3)
>>> f4(1, 2, 3, x=2, y=3)

>>> f5(1)
>>> f5(1, 4)

>>> f6(1)
>>> f6(1, 3, 4)
```

本章内容：

- 为什么要使用模块？
- 模块基础
- 模块文件是名字空间
- 导入模式
- 重载模块
- 其他内容
- 模块的常见问题
- 总结
- 练习

第五章

模块

这一章介绍Python的模块——最高级的程序组织单位。它可以打包程序代码和数据以备重用。具体的说：模块采用Python程序的文件形式（和C扩展程序的形式），客户导入模块并使用它们所定义的名字。模块可以被如下两种新语句和一种重要的内置函数所处理：

`import`

允许客户按整体取得一个模块

`from`

允许客户从一个模块中取得某些特定的名字

`reload`

提供了一种无需停止Python就可以重载模块代码的方法

我们在第一章“开始”介绍了模块的基础知识，你可能在练习中已经用过模块文件了，这一章中的有些内容可能是复习。但我们也将丰富一下至今为止忽略掉的细节：重载、模块编译语义，等等。因为模块和类实际上是修饰过的名字空间（*namespace*），我们要在此探讨一下名字空间的基础。所以在你学习后面的章节前，要确保已读过本章的大部分内容。

为什么要使用模块？

让我们从最显而易见的第一个问题开始：我们为什么要使用模块呢？最简短的答案是模块提供了一种简单的方法，可以把组件组织成一个系统。但从抽象的角度看，模块至少有三个作用：

代码重用

正如我们在第一章看到的，模块可以让我们在文件中长久地保存代码（注1）。与在交互提示符下键入的代码不同（当你退出Python的时候，代码就会消失），在模块中的代码是持久的 —— 如果你需要的话可以重载、运行多次。不仅如此，模块还是一个可以定义名字（也称属性）的地方，可以被外部客户所引用。

系统名字空间的划分

模块是Python最高级的程序组织单位，我们将看到任何Python中的东西都是“活”在模块里的，你执行的代码、生成的对象总是隐含地包含在一个模块中。正因为如此，模块是天然的组织系统组件的工具。

实现服务或数据的共享

从功能的角度上看，模块也用于在整个系统里实现组件共享，这只需要一个拷贝。例如，如果你要提供一个可以在多个函数中使用的全局数据结构，你可以将其写在一个模块文件中，再被许多客户导入。

模块基础

Python模块很容易生成：它们只是些包含Python代码的文件，可以用你最爱用的文本编辑器生成它。你无需写特别的语法告诉Python你已经编写了一个模块；几乎所有的文本文件都行。模块文件还易于使用，因为Python可以处理寻找、载入模块的大部分细节；客户简单的导入模块或模块中定义的特定名字，并使用它们引用的对象，下面是对模块基础的一个概括：

生成的模块形式：Python文件、C扩展程序

模块通常被编写成Python文件或者C语言的扩展程序的形式。在本书中我

注1： 至少是在你删除这个模块文件之后。

们不学习C扩展程序，但是我们将用到一些。很多Python的内置工具事实上都是导入的C扩展模块；对客户来说，它们看上去与Python文件模块没什么两样。

使用模块：import、from、reload()

我们将看到，用户可以用import或者from语句载入模块。通过调用内置函数reload，可以不用停止Python就能重载一个模块代码。模块文件还可以在系统提示符下作为顶层程序运行，像我们在第一章中看到的那样。

模块查找路径：PYTHONPATH

我们已在第一章中看到，Python通过检查所有在PYTHONPATH环境变量中列出的目录，来查找可以导入的模块。只要将你所有的源程序目录添加到该变量中，就可以在任何地方存储模块。

定义

让我们在实际中看一个关于模块基础知识的简单例子。为了定义一个模块，用文本编辑器在一个文本文件中输入Python代码，模块顶层赋值的名字成为它的属性（与模块对象有关联的名字），然后导出给客户使用。例如，如果我们在一个*module1.py*的文件中输入def语句，我们就生成了有一个属性的模块——名字printer，它恰好是对一个函数对象的引用：

```
def printer(x):                # 模块属性
    print x
```

关于文件名：你可以按自己喜欢的方式命名你的模块文件，不过如果你打算导入它们的话，模块文件应该以一个*.py*后缀结尾。既然它们的名字去掉*.py*后缀就成为了Python程序中的变量，它们也应该遵从第三章“基本语句”中的命名规则。例如，一个叫*if.py*的模块是不会工作的，因为if是一个保留字（你会得到一个语法错）。当模块导入的时候，Python把内部的模块名字映射为外部的文件名，规则是在名字头部添加在PYTHONPATH变量内的目录路径名，在尾部加上*.py*：一个叫M的模块就变成了外部文件*<directory-path>/M.py*（注2）。

注2：如果已经存在一个该模块的已编译形式，它也可以映射到*〈directory-path〉/M.pyc*；更多的内容在后面讲述。动态的载入C的扩展模块也可以在PYTHONPATH中发现，不过这超出了本书的范围。

用法

用户可以通过 import 或 from 语句运行我们刚写过的代码，二者都会载入模块文件代码。主要的区别是 import 作为一个整体取得代码（故你必须加以限制才可以取出它的名字），而 from 可以从模块中取得指定的名字，这里有三个工作中的模块客户：

```
% python
>>> import module1                      # 得到模块
>>> module1.printer('Hello world!')     # 限定性取得名字 （模块．名字）
Hello world!

>>> from module1 import printer         # 得到一个输出
>>> printer('Hello world!')             # 不需要限定名
Hello world!

>>> from module1 import *               # 得到所有输出
>>> printer('Hello world!')
Hello world!
```

最后的例子用了 from 的一个特殊形式：使用 * 可以得到在被引用模块顶层赋值的所有名字的拷贝。在每一个例子中，我们都进行了对 printer 函数的调用，而 printer 函数在外部模块文件中定义。确实如此，模块真的很容易使用。但是你自己定义、使用模块的时候会发生什么呢？为了让你更好的理解这一点，让我们更详细的看一下它的一些属性。

模块文件是名字空间

模块最好理解为一个地方，在这里可以定义系统其余部分都能看见的名字。用 Python 的术语来说，模块就是名字空间 —— 生成名字的地方。存在模块中的名字称为它的属性（*attribute*）。严格地说，模块对应于文件，Python 生成一个模块对象包含所有文件中定义的名字；但用简洁的术语来讲，模块就是名字空间。

那么，文件是如何成为名字空间的？在一个模块文件顶层（即不是在一个函数体中），赋值的每一个名字都成为模块的一个属性。例如，在模块文件 *M.py* 的顶层给出一个像 *X=1* 这样的赋值语句，名字 *X* 就称为 *M* 的属性，可以从外部以 *M.X*

引用它。名字*X*对*M.py*中其他代码来说就成为了一个全局变量。不过我们需要更正式一点地解释一下模块载入和作用域的概念，以更好的理解这是为什么：

模块语句在第一次导入的时候执行

在系统任何地方第一次导入一个模块，Python生成一个空的模块对象，并从头到尾依次执行模块文件中的语句。

顶层赋值生成模块的属性

在导入的过程中，在文件顶层赋值名字的语句（如：=，def）生成模块对象的属性，赋值的名字在模块的名字空间里存储。

模块的名字空间：attribute_ _dict_ _，或dir( )

通过导入而生成的模块空间是字典（*dictionary*）；它们可以通过与模块对象关联的内置__dict__属性所访问，并可用我们在第一章遇到的dir函数来检察。

模块文件是单一作用域的（局部的就是全局的）

正如我们在第四章“函数”中看到的，一个模块顶层的名字与函数中的名字一样，遵从同样的引用/赋值规则。不过局部的和全局的是一样的（或者你愿意的话，LGB规则中去掉G）。但是在模块中，当模块已被载入后，局部作用域成为一个模块对象的属性字典，这与函数不同（函数的局部名字空间只在函数运行时才存在）。一个模块的作用域是一个模块对象的属性名字空间，在导入之后就可以使用。

让我们看一下有关这些概念的一个例子。假设我们用自己喜欢的编辑器生成一个叫*moldule2.py*的模块文件：

```
rodule2.py :
print 'starting to load...'

import sys
name = 42

def func(): pass

class klass: pass

print 'done loading.'
```

首先模块被导入（或者作为程序运行），Python从头到尾执行它的语句。有的语句的作用只是在模块名字空间中生成名字，不过其他的语句在import运行的时候可能做一些实际的工作。例如，这个文件中的两个print语句在导入的时候会运行：

```
>>> import module2
starting to load...
done loading.
```

不过一旦模块载入后，它的作用域就成为我们从import中得到的模块对象的一个属性名字空间。用包含它的模块名加以限定，我们可以访问名字空间里的属性：

```
>>> module2.sys
<module 'sys'>
>>> module2.name
42
>>> module2.func, module2.klass
(<function func at 765f20>, <class klass at 76df60>)
```

在这里，sys，name、func和klass都在模块语句运行时被赋值。这样在导入后它们就是属性。我们将在第六章“类”中讨论类，但注意sys属性：import语句真正的给名字赋值了模块对象（更多的内容在后面讲述）。在内部，模块名字空间作为字典对象存储，实际上，我们可以通过模块的__dict__属性访问名字空间字典；它只是普通的字典对象，有通常的方法：

```
>>> module2.__dict__.keys()
['__file__', 'name', '__name__', 'sys', '__doc__', '__builtins__', 'klass',
'func']
```

我们在模块文件中赋值的名字在内部成为字典的键（key），你可以看到在模块名字空间里的一些名字都是Python为我们加的；例如__file__给出载入的模块文件名，__name__把知道的名字给导入者（不带有.py扩展名和目录路径）。

名字限定

现在你开始熟悉模块了，我们要解释一下名字限定（qualification）的概念。在Python中，你可以访问任何有属性的对象的属性，使用这样的限定语法：对象.属性（object.attriute）。限定事实上是一个表达式，返回与一个对象关联的一个

属性名字的值。例如，在上面倒数第二个例子中的module2.sys表达式取得在module2中赋值给sys的值。类似的，如果我们有一个内置列表对象L，L.append返回与列表关联的方法。

这对我们在第四章看到的作用域规则有什么影响呢？没有，实际上，它是一个独立的概念，当你用限定去访问名字时，你让Python去获取一个明确的对象。LGB规则只对无限定性的名字适用，规则如下：

简单变量

"X"代表着从当前作用域中查找名字X（LGB规则）

限定

"X.Y"代表在对象中查找Y属性（而不是作用域中）

限定路径

"X.Y.Z"代表在对象X中查找名字Y，再在对象X.Y中查找Z

一般性

限定可以对所有拥有属性的对象工作：模块、类、C类型等等

在第六章，我们将看到限定与类（也通常是继承出现的地方）的关系更密切。一般情况下，这里的规则适用于Python中的所有名字。

导入模式

正如我们看到的那样，限定只有在你用import作为整体去导入一个模块时才需要。当你用from语句的时候，你从模块中拷贝名字给导入者，所以使用导入的名字不用限定。这里有一些更详尽的关于导入过程的细节。

导入只发生一次

在使用模块时，新手最常问的问题就是：为什么我的导入不继续工作了？第一次导入工作良好，但在一个交互的任务的过程中，以后几次导入似乎没有任何作用。不如设想的那样，原因在这里：

- 模块在第一次 import 或 from 的时候载入并运行。
- 运行一个模块的代码生成它的顶层名字。
- 以后的 import 或 from 操作取一个已经载入的模块。

Python 有目的地在第一次导入时载入、编译运行一个模块文件中的代码；既然这是一个代价很大的操作，默认情况下，每个过程 Python 只处理一次。而且，既然模块中的代码通常只执行一次，你可以用它来初始化变量。例如：

```
% cat simple.py
print 'hello'
spam = 1                    # 初始化变量

% python
>>> import simple           # 第一次导入：载入并运行代码
hello
>>> simple.spam             # 赋值生成一个属性
1
>>> simple.spam = 2         # 在模块中改变属性
>>>
>>> import simple           # 只是取到已经载入的模块
>>> simple.spam             # 代码不重新运行：属性没有改变
2
```

在这个例子中，print 和 = 语句在模块导入的第一次运行，第二个导入不重新运行模块的代码，而是只取得 Python 内部模块表中已经生成的模块。当然了，有时候你也需要重新运行一个模块的代码；一会儿我们将看到如何用 reload 来做。

import 或 from 都是赋值

类似于 def，import 和 from 都是可执行的语句，不是编译时的声明。它们可以在 if 条件中嵌套，可以出现在 def 函数中等等。导入的模块和名字直到导入语句运行后才可用，而且，import 和 from 也是隐含的赋值，像 def 一样：

- import 把整个模块对象赋值给一个名字。
- from 把一个或多个名字赋值给另一个模块中的同名字对象。

我们所说的所有关于赋值的内容也适用于模块访问。例如，用 from 拷贝的名字就是一个对可以被共享的对象的引用；与函数参数一样，重新赋值一个已经取得的名字对一个用from拷贝的模块没有影响，但改变一个已经取得的可变对象可能会改变用 from 导入的模块中的对象（注 3）：

```
% cat small.py
x = 1
y = [1, 2]

% python
>>> from small import x, y      # 把两个名字拷贝出来
>>> x = 42                      # 只改变局部的 x
>>> y[0] = 42                   # 在原处改变可变的共享对象
>>>
>>> import small                # 得到模块名字（用的不是 from 语句）
>>> small.x                     # small 的 x 不是局部的 x
1
>>> small.y                     # 不过我们共享一个改变了的可变对象
[42, 2]
```

在这里，我们改变了一个用 from 赋值得到的共享的可变对象：导入者和被导入者中的名字 y 引用同样的列表对象。这样从一个地方改变它，在另一个地方也会改变。顺便说一下，为了获得对限定它的模块名字的访问权，我们不得不在 from 后边执行了一个 import 操作；from 只在模块中拷贝名字，并不赋值模块名字本身。至少从象征意义上讲，from 与下面的序列等价：

```
import module                   # 取得模块对象
name1 = module.name1            # 通过赋值拷出名字
name2 = module.name2
...
del module                      # 去掉模块名字
```

重载模块

在上一节的开头，我们提到过默认情况下，在每一过程中一个模块的代码只能运

注 3：事实上，要从图上看一下 from 都做了些什么，返回到图 4-2（函数参数传递）。用“被导入者”与“导入者”代替“调用者”与“函数”，看一下 from 赋值对引用都做了些什么；效果是完全一样的，不同点就是在模块里而不是在函数中处理名字。

行一次。为了强迫一个模块代码重新载入、重运行，你需要通过调用reload内置函数明确地告诉Python这样做。在这一节中，我们将探讨如何使用reload使你的系统更加动态化，简单的描述一下：

- import只在第一次载入并运行模块中的代码。
- 以后的import使用已经载入的模块对象而无需重新运行代码。
- reload函数强迫一个已经载入的模块代码重新载入，重运行。

为什么所有的麻烦都是关于重载模块的？reload函数允许部分改变程序而无需停止整个程序。用reload，组件的改变可以立刻观察到。重载不是在任何情况下都有用，但是用它的时候，可以使你的开发周期大大缩短。例如，既然程序的改变可以在重载后立刻测试到，想像一个数据库程序启动时必须连接到一个服务器上，除错的时候你只需连接一次就可以了（注4）。

一般形式

与import和from不同：

- reload是Python中的一个内置函数，不是一个语句。
- reload的参数是现有的模块对象，不只是一个名字。

因为reload的参数是一个对象，一个模块在你能重载之前必须已经被成功的导入了。（事实上，如果导入不成功的话将导致一个语法错或其他错误，在你能重载之前，你可能要重复一次导入。）重载过程是这样的：

注4：我们也应该指出，因为Python是解释性的（或多或少），它已经去掉了编译、连接步骤，而这些步骤对于一个C程序而言是必须经过的；当一个程序运行的时候，模块动态载入。通过允许你可以无须停止就改变部分运行程序重载加入。我们还应该指出reload当前只对用Python写的模块起作用；C的扩展模块也可以在运行的时候被动态载入，但它们不能被重载；最后我们要指出既然这本书不是关于C模块的，我们不会在这方面叙述太多。

```
import module                          # 初始化导入
使用module.attributes（模块．属性）
...                                    # 现在，去改变模块文件
...
reload(module)                         # 得到更新后的输出
使用module.attributes（模块．属性）
```

通常你导入一个模块，在文本编辑器中改变它的源代码重载它。当你调用reload的时候，Python重新读取模块文件的源代码并运行它的顶层语句。不过最重要的事情可能是，reload在原位改变一个模块对象；正因为如此，reload后一个模块对象的每一个引用都自动受影响。细节如下：

reload在模块的当前名字空间运行一个模块文件的新代码

重新运行一个模块文件的代码重写已经存在的名字空间，而不是对它的删除与重新生成。

文件顶层赋值的名字用新值代替

例如，重运行一个def语句代替模块名字空间的先前的版本。

reload影响所有用import取得模块的客户

因为客户用import限定取得属性，在reload后，我们在模块中会发现新值。

reload只影响将来的from客户

先前用from取得属性的客户不受reload影响，它们仍拥有reload前对旧对象的引用（后面我们将详加讨论）。

例子

这里有一个reload的实例。在下面的交互中，我们改变并重载一个模块文件而无需停止交互式Python的对话，重载也用在许多其他场合（见下面文本框中的选读内容），为了便于描述，我们将保持简单性。首先，用我们喜看的编辑器写一个模块文件：

```
% cat changer.py
message = "First version"

def printer():
    print message
```

这个模块生成并输出两个名字——其中一个绑定了一个字符串，另一个绑定了一个函数。现在，启动Python解释器，导入模块，调用它输出的函数；现在你该知道，该函数打印全局变量message：

```
% python
>>> import changer
>>> changer.printer()
First version
>>>
```

接下来，保持解释器的活动状态并在另一窗口编辑模块文件；在这里，我们改变全局message变量，在printer函数体中也进行修改：

```
不终止Python，修改changer.py
% vi changer.py
% cat changer.py
message = "After editing"

def printer():
    print 'reloaded:', message
```

最后，我们返回Python窗口，重载模块，取得我们刚修改的代码。注意，又一次导入模块没什么作用；即使文件已经修改了，我们仍得到原来的信息。要得到新的版本我们不得不使用reload：

```
返回Python解释器/程序

>>> import changer
>>> changer.printer()      # 没有作用：用已经载入的模块
First version

>>> reload(changer)        # 强制新的代码载入/运行
<module 'changer'>
>>> changer.printer()      # 现在运行新的代码
reloaded: After editing
```

注意reload真的为我们返回模块对象：它的结果通常被忽略，不过既然表达式结果在交互提示符下打印，Python给我们显示了一个默认的<module name>表示法。

请留意：模块重载

除了允许你在交互模式下（也就可以重新运行）模块，模块重载在大型程序中也是很有用的，尤其是需要避免整个应用的重启动时。例如，系统必须通过网络连接到服务器上，首先考虑的就应该是动态重载。

在GUI中（在GUI仍保持活动时，一个组件回调动作仍可被改变），和Python作为一种嵌入语言应用在C或C++程序中（包含它的程序能要求它运行的代码重载，而无需停止）时，这也是十分有用的。参考《Programming Python》一书可以得到更多的关于重载GUI回调和嵌入的Python代码的知识。

其他内容

在这一节中，我们介绍一些与模块相关的概念。这些概念本身就很重要（或者说非常抽象，足以挑战我们的组织技能）。

模块编译模式

Python系统通常被称作一个解释器。不过它实际上介于传统解释器和编译器二者之间。与Java相同，Python程序被编译为一种称之为字节码的中间形式。它在一个叫虚拟机的东西上执行。既然Python虚拟机解释字节码，我们可以说Python是解释性的，不过首先它仍将经过一个编译阶段。

幸运的是，编译步骤在Python中是自动的和隐藏的。Python程序简单的导入模块文件并使用它们定义的名字；Python在模块第一次导入的时候自动把它们编译成字节码。而且，Python试图保存字节码到一个文件中，这样将来如果源文件没有改变的话可以避免重新编译。从效果上说，Python用一个*make*系统自动管理重编译（注5）。

注5：对那些从来没有用过C或者C++的读者来说，*make*系统就是自动编译连接程序的一种方式。当一个文件必须要重新编译的时候，*make*系统通过文件调整数据的值，（与Python一样。）

它的工作原理是这样的。运行程序后，你可能已经注意到*.pyc*文件保存在你的目录中。这些是Python生成的用以保存生成的模块字节码的（假设你有对源代码目录的访问权）。

当一个模块*M*导入时，Python载入一个*M.pyc*的字节码文件以代替相应的*M.py*源文件，只要*M.pyc*保存后*M.py*没改变即可。如果你改变源代码文件（或删除*.pyc*文件），Python足够的智能化，会在导入时重编译模块；如果不是这样，存好的字节码文件通过避免运行时的重编译可以更快地启动文件。

请留意：发送选择

顺便说一下，编译过的*.pyc*字节码文件恰好是一种可以不用源代码发放系统的方法。在Python的模块查找路径中若没有发现*.py*源文件，就会载入一个*.pyc*文件，所以你要发送给用户的就是*.pyc*文件。而且，既然Python的字节码是可移植的，你通常可以在多平台上执行*.pyc*文件。为了强制预编译的文件成为*.pyc*文件，简单的导入模块即可（另见：compileall工具模块）。

把Python的程序变为C语言的可执行形式也是可能的；标准的*freeze*工具打包了你的程序编译的字节码，它生成一个C程序，你可以用一个生成的*makefile*文件产生一个独立的可执行程序。这个可执行程序，可以像你的Python程序中的文件一样工作。转化的可执行程序不需要在目标机器上安装Python解释器，并可以更快速启动；另一方面，因为大量的解释器代码也被包含进去了，程序不再短小。一个类似的工具，squeeze，在一个Python程序中打包Python字节码，查找Python的Web站点可以得到细节内容。

数据隐藏是一种约定

正如我们看到的，Python模块导出所有在文件顶层赋值的名字。没有什么概念声明哪个名字在模块外可见哪个不可见。事实上，没有什么方法可以阻止一个用户改变一个模块内部的名字，如果想做就可以。

在Python中，数据隐藏是一个约定，而不是一种语法限制。如果你想通过废掉一个模块的名字来中断你的模块的话，你可以这样（虽然我们还没遇到一个程序员

想这样做)。有些纯粹主义者,因为反对这种数据隐藏的自由态度,就断言Python不能实现封装(encapsulation)。我们并不同意这种说法(但吃不准我们是否能说服他们)。因为严格的说,Python中的封装不只是打包(package)(注6)。

作为一个特例,带下划线前缀的名字(比如_X),当一个用户用from*语句导入时可以不让自己被拷贝出去。这实际上只是想最小化名字空间污染(*namespace pollution*);由于from*可以拷贝出所有的名字,你得到的可能比你想的要多(包括在导入者中被重写的名字)。不过下划线不是"私有的"声明:你仍可以用其他的导入形式查找并改变它。

混合模式:__name__和__main__

这里有一个特别的与模块相关的技巧,既可以允许你导入一个模块,又可作为一个独立程序运行它。每个模块都有一个内置属性叫__name__,Python中设定如下:

- 如果文件作为程序运行,在启动的时候,__name__设定为字符串__main__
- 如果文件是被导入的,__name__被设定为用户已知的模块名

结果,模块可以检测自己的__name__,决定是运行还是被导入。例如,假设我们生成下面的模块文件,导出一个叫tester的单一函数:

```
def tester():
    print "It's Christmas in Heaven..."
if __name__ == '__main__':          # 只在独立运行时才这样
    tester()                        # 而不是被导入时
```

这个模块为用户定义了一个函数用以进行通常的导入与使用:

```
% python
>>> import runme
>>> runme.tester()
It's Christmas in Heaven...
```

注6: 纯粹主义者可能会被那些"流氓"C++程序员吓坏:他们用#public private public来打破C++隐藏机制。对你来说可能也会有同样的感觉。

但是模块也包含了在底部的代码，该代码被用于当文件作为程序运行的时候，调用函数：

```
% python
It's Christmas in Heaven...
```

可能最常见的__main__测试都应用在自测试代码上：你可以通过在模块底部包含一个__main__测试，将测试模块自身的输出打包在模块中。用这种方式，你可以在客户程序里使用文件，并通过在系统 shell 下运行以测试它的逻辑性。

改变模块查找路径

我们已提到过模块查找路径就是环境变量 PYTHONPATH 中的一个目录清单。我们还没有告诉你一个 Python 程序实际上可以改变查找路径，通过对一个内置列表 sys.path 赋值即可（path 属性在内置的 sys 模块中）。sys.path 在启动时从 PYTHONPATH 初始化（加上默认编译值），不过喜欢的话你可以删除、追加、重设置 PYTHONPATH 的内容：

```
>>> import sys
>>> sys.path
['.', 'c:\\Python\\lib', 'c:\\Python\\lib\\tkinter']

>>> sys.path = ['.']                                    # 改变模块查找路径
>>> sys.path.append('c:\\book\\examples')               # 转义字符 "\\"
>>> sys.path
['.', 'c:\\book\\examples']

>>> import string
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ImportError: No module named string
```

你可以用这种方法在一个Python程序中动态设置查找路径。不过仍要小心，如果你从路径中删除了一个重要目录的话，你可能会丢失对某些程序的访问权。例如，上面的最后一条命令，我们从路径中删除了库目录后，我们就不再拥有对模块 string 的访问权。

模块包（1.5版本新内容）

包（package）是一种高级的工具，我们两个人在是否在本书中讲述它还发生了争执。不过，因为你可能在别人的代码遇见它们，还是快速地看一下它的机制吧！

简单的说，Python程序包允许你用目录路径导入模块；import语句中的限定名反映了你机器上的目录结构。例如，如果模块C存在于目录中，B是目录A中的子目录。你可以用 `import A.B.C` 载入模块。只要目录A可以在PYTHONPATH变量中的目录列表中发现即可，因为从A到C的路径通过限定已经给出了。

当需要集成多个独立的开发者开发的系统的时候，包就显得很有用了。通过在每一个自己的子目录中存储每个系统的模块设定，我们可以减少名字冲突的风险。例如，每一个开发者都写了一个叫做*spam.py*的模块。如果没使用包限定路径的话，就不能知道那一个会先从PYTHONPATH中发现。若另一系统的目录首先出现在PYTHONPATH中，子系统可能会看到错误的一个。

再说一遍，如果你是Python新手的话，在学习包之前应确信你已掌握了简单模块的知识。包远比我们这里描述的更加复杂；例如，每个作为包使用的目录必须包括一个 `__init__.py` 以标识自己。查阅Python参考手册可以得到全部内容。

请留意：模块包

现在包已是Python标准的一部分了，你可能已经开始看到了一些以包目录形式分发的第三方扩展程序，而不只是一组模块。Python针对MS-Windows的Pythonwin扩展程序是第一个使用包的，它的很多模块在包中，你可用限定路径导入；例如，载入客户端COM工具，我们这样写：

```
from win32com.client import constants, Dispatch
```

它从Pythonwin的 `win32com` 包（一个安装目录）中的 `client` 模块取得名字，在第十章“框架与应用”我们会看到更多关于COM的内容。

模块设计的概念

同函数一样，模块表示了设计上的一些权衡：决定哪个函数在哪个模块中，模块

的通信机制等等。这是一个大的主题，超出了本书范围。所以我们只接触一些一般概念，这样在你开始写一个大型程序的时候思路会更加清晰：

在Python中，代码总在一个模块里

书写不存在于模块中的代码是不可能的，事实上，交互提示符下输入的代码实际上也是在一个内置的模块__main__中。

最小化模块连接（coupling）：全局变量

像函数一样，如果模块作为封闭盒子工作最好。实践中，应尽可能的让它们作为独立的全局名字存在于其他的模块中。

最大化模块内聚（cohesion）：统一目标

你可以通过最大化模块的内聚来最小化它的连接。如果所有的模块组件都目标一致的话，你依赖外部名字的可能性就更少。

模块应尽量少改变其他模块的变量

使用另一个模块中的全局名字很好（毕竟，那是用户导入服务的方式）。不过在其他模块中改变全局名字通常代表设计有问题。当然也有例外，不过你应该尽量通过像函数返回值这样的方式传送结果，而不是直接改变模块。

模块是对象：元程序

最后，因为模块把很多有意义的量作为自己的内置属性，书写管理其他程序的程序是简单的。我们通常称这样的程序为元程序（*metaprogram*）。因为它在其他系统的顶层工作。有时也叫“内省（introspection）”，因为程序可以查看，处理对象内部。

例如，为得到模块M中叫name的属性，我们可以用限定做到这一点；或者，索引在内置属性__dict__中的模块属性字典。而且，Python以sys.modules字典的形式导出所有载入模块的列表（也就是sys模块的模块属性），并提供了一个内置工具getattr，可以让我们从它们的名字字符串取得属性。正因如此，下面的所有表达式均可以得到同样的属性与对象：

```
M.name                          # 限定对象
M.__dict__['name']              # 手工检索名字空间字典
```

```
sys.modules['M'].name          # 手工检索载入的模块表
getattr(M, 'name')             # 调用内置的函数
```

通过与此类似的显示模块内部的方式，Python帮助你创建关于程序的程序(注7)。例如，有这样一个模块包含了这些思想，它可以实现内置函数dir的定制版本。它定义并输出了一个叫listing的函数，listing把模块对象当作一个参数，并打印格式化的模块的名字空间列表：

```
# 一个模块，可以列出其他模块的名字空间

verbose = 1

def listing(module):
    if verbose:
        print "-"*30
        print "name:", module.__name__, "file:", module.__file__
        print "-"*30

    count = 0
    for attr in module.__dict__.keys():    # 扫描名字空间
        print "%02d) %s" % (count, attr),
        if attr[0:2] == "__":
            print "<built-in name>"          # 跳过__file__，等等
        else:
            print getattr(module, attr)      # 如同.__dict__[attr]
        count = count+1

    if verbose:
        print "-"*30
        print module.__name__, "has %d names" % count
        print "-"*30

if __name__ == "__main__":
    import mydir
listing(mydir)             # 自测试代码：列出自己
```

注7：注意因为一个函数可以通过检查这样的sys.modules表访问包含它的模块，它也可能模拟我们在第四章中遇到的global语句的效果。例如，global X; X=0的作用可以在一个函数中这样说：import sys; glob=sys.modules[__name__]; glob.X=0(尽管需要更多的键盘输入)，记住，每一个模块得到一个__name__属性用来释放内存空间。在一个模块的函数中它可作为全局名字访问。这个技巧提供了一种在方式一个函数中改变同一个名字的全局和本地的变量。

我们还在模块底部提供了逻辑自测试，它自己导入并列出了自己。生成的输出如下：

```
C:\python> python mydir.py
-------------------------------------
name: mydir file: mydir.py
-------------------------------------
00) __file__ <built-in name>
01) __name__ <built-in name>
02) listing <function listing at 885450>
03) __doc__ <built-in name>
04) __builtins__ <built-in name>
05) verbose 1
-------------------------------------
mydir has 6 names
-------------------------------------
```

我们会再一次遇到getattr和它相关的函数。值得注意的一点是，mydir是一个允许你浏览其他程序的程序。因为Python可以显示其内部情况，这使得你可以按总称（generically）处理对象（注8）。

模块的常见问题

最后，这里收集了大量的例子，它们可能使新手的编程更有趣。也有些十分抽象不易理解。不过大多数都描述了Python的重要内容。

通过名字字符串导入模块

我们已经看到，在import或from语句中的模块名是一个硬性编码的变量名；你不能用这些语句直接导入以Python的字符串形式给定名字的模块。例如：

```
>>> import "string"
```

注8：顺便说一下，像mydir.listing这样的工具能被预载入交互式的名字空间内，通过在被PYTHONSTARTUP环境变量引用的文件中导入它们。既然在开始文件中的代码在交互式名字空间里运行（模块__main__）。开始文件中的普通工具的导入可以节省输入。参阅第一章可以得到更多内容。

```
File "<stdin>", line 1
    import "string"
                  ^
SyntaxError: invalid syntax
```

解决办法

你需要用特殊工具动态的从一个在运行时生成的字符串载入模块。最通用的方法是构建一个 import 语句作为一段 Python 的字符串，并把它传给 exec 语句执行：

```
>>> modname = "string"
>>> exec "import "+ modname          # 运行一个代码字符串
>>> string                           # 在这个名字空间中导入
<module 'string'>
```

exec 语句（和它的同类函数 eval）编译一个代码的字符串，并传递给 Python 解释器来执行。在 Python 中，字节码编译器可在运行时得到。所以你可以像这样写出程序，构建运行其他的程序。默认情况下，在当前作用域运行代码，不过你可以通过传递名字空间的字典选项得到更特定的，在本书的后面将加以讨论。

exec 唯一真正的缺点是，每次运行的时候都必须重新运行编译 import；如果它运行许多次，你最好用内置的__import__函数从一个名字字符串载入它，作用相似，但是__import__返回模块对象，所以我们在这里赋值给它一个名字：

```
>>> modname = "string"
>>> string = __import__(modname)
>>> string
<module 'string'>
```

from 拷贝名字但并不连接

先前，我们提到 from 语句实际上是导入者作用域中的对名字的一个赋值——是一个名字拷贝操作，而不是给一个名字起别名。它的含义与所有 Python 中的赋值都相同，不过有些微妙之处，尤其在指定了不同文件中都共享的对象代码时。例如，假定我们定义了一个如下的 nested1 模块：

```
X = 99
def printer(): print X
```

现在，如果我们在另一个模块中用from导入这两个名字。我们得到两个名字的拷贝，而不是对它们的连接。在导入者中改变名字只改变了局部的这个版本，而不是在nested1中的名字：

```
from nested1 import X, printer          # 拷出名字
X = 88                                  # 只改变局部的"X"!
printer()                               # 嵌套的X仍旧是99

% python nested2.py
99
```

解决办法

相反，若你用import得到整个模块并赋予它一个限定的名字，你就改变了nested1的名字。限定把Python指向模块对象中的一个名字，而不是导入者中的名字：

```
import nested1              # 作为整体得到一个模块
nested1.X = 88              # OK；修改了嵌套的X
nested1.printer()

% python nested3.py
88
```

顶层代码语句的顺序问题

先前我们看到，在一个代码第一次导入（或重载）的时候，Python执行它的代码，从文件的头部到尾部，这里有些微妙的细节值得强调：

- 在一个导入过程中，模块文件顶层代码（而非嵌套在函数中的）只要Python解释到它的位置就会运行；正因为如此，它不能引用在下部赋值的名字。
- 在函数体中的代码一直到函数调用后才会运行；因为函数中的名字要一直到函数真正运行后才会解析。在文件中它们通常可在任意位置引用名字。

一般地说，向前引用只与模块代码顶层立即执行代码有关；函数可以任意引用名字。这是一个例子，描述向前引用的含义：

```
func2()                    # 错："func1"还没赋值

def func1():
    print func2()          # OK："func2"在后面查找

func1()                    # 错："func2"还没赋值

def func2():
    return "Hello"

func1()                    # OK："func1"、"func2"赋值了
```

当这个文件导入时（或者作为单独的程序运行时），Python从头到尾执行它的语句，第一个对func1的调用失败是因为func1的def还没执行，只要当func1被调用时，程序已经运行了func2的def语句，对func1中的func2调用就会工作（第二个func1调用的时候还没有运行func2的def）。最后对func1的调用在文件底部，工作正常，因为func1与func2都已经被赋值。

解决办法

别那样做，在顶层代码中混杂def不仅仅使程序不易读，还要依赖于语句顺序。来自实践的原则是，若你需要立刻用def混合代码，把你的det放在文件前面，顶层代码放在文件底部，用这种方式，你的函数在运行前已经被定义和赋值。

递归的from导入可能不工作

因为导入从头到尾地执行文件的语句，我们在用模块互相导入（有时叫递归导入）时需要加以小心，既然在导入其他模块时模块中的有些语句还没运行，它的一些名字可能还不存在。若你用import作为整体取得一个模块，可能无关紧要；直到以后使用限定方法取得它们的值时，模块的名字才能被访问。不过如果你用from来取得特定的名字，你只对已赋值的名字有访问权。

例如，有下面两个模块：recur1和recur2。recur1赋值了名字X，接着在赋值名字Y前，导入recur2。这时，recur2可以用一个import语句作为整体取得recur1（它已经在Python内部模块表中存在了）。下过，如果使用from语句的话只能看到名字X；在recur1的import时名字Y还不存在，这样会得到一个错误：

module recur1.py

```
X = 1
import recur2                    # 如果还不存在的话运行 recur2
Y = 2
```

module recur2.py

```
from recur1 import X             # 好的："X" 已经赋值
from recur1 import Y             # 错："Y" 还没赋值
```

```
>>> import recur1
Traceback (innermost last):
  File "<stdin>"> line 1, in ?
  File "recur1.py", line 2, in ?
    import recur2
  File "recur2.py", line 2, in ?
    from recur1 import Y         # 错："Y" 还没赋值
ImportError: cannot import name Y
```

当从 recur2 中递归导入 recur1 时，Python 的智能化很强，能知道避免重运行 recur1 中的语句（否则，导入会使脚本进入无限循环中），不过被 recur2 导入时，recur1 的名字空间是不完整的。

解决办法

别那样做，真的！Python不会卡在一个循环中，但你的程序会不止一次的依赖于模块中的语句顺序。摆脱这个问题有两种方法：

- 你通常可以通过细心设计去除导入循环，最小化模块间的连接、最大化内聚是好的开端。
- 如你不能完全的去掉循环，通过使用 import 和限定方法（代替 from）延迟对模块名字的访问。或是从函数内部（而不是在函数顶部）运行你的 from。

reload 可能不影响 from 导入

from语句是Python中的问题之源。这是另一种情况：因为from在运行时拷贝（赋值）名字，没有与名字来源模块的连接。用from导入的名字只是简单地成为对对象的引用，在from运行该对象时碰巧已经被导入者中的同一个名字引用。因为这种行为，重载被导入者对使用 from的客户没有影响；客户的名字仍旧引用 from 取得的对象，即便源模块中的名字已经被重设定过了：

```
from module import X       # X可能不影响任何模块重载!
. . .
reload(module)             # 改变模块，不是我的名字
X                          # 仍然引用旧的对象
```

解决办法

别那样做，为了使重载更加有效，用import和名字限定取代from。因为限定总返回到模块中，调用reload后，它们会发现模块名字的新的绑定：

```
import module              # 得到模块，而不是名字
. . .
reload(module)             # 在原位置改变模块
module.X                   # 得到当前的X：反映了模块的重载
```

重载无传递性

当你重载一个模块的时候，Python只重载特定模块的文件，它并不自动载入该模块也载入的文件。例如，如果我们重载一个模块A，A导入B和C，重载只适用于A，对B和C无效。A中的导入和C的语句在重载过程中重新运行。但它们将只取得已经载入的B和C模块对象（假定它们以前已经被导入）：

```
% cat A.py
import B                   # 当A导入的时候不导入
import C                   # 只是对已经导入的模块的一个导入

% python
>>> . . .
>>> reload(A)
```

解决办法

不要依靠于此，用多个reload调用来单独更新子组件。需要的话，你可以设计你的系统，通过在像A这样的父模块中添加reload调用来实现子组件的自动重载（注9）。

注9：你还能写一个通用的工具，通过扫描模块__dict__s自动地执行传递重载（参见“模块是对象：元程序”一节。）检查每一项的type()发现嵌套的模块以递归地重载入。这是一个高级的练习。

总结

在这一章中我们学习了模块——如何写，如何用。接着我们探讨了名字和限定方法；如何重载模块以改变运行的程序，并看了看模块设计问题。还学习了一些表5-1中列出的模块相关的语句和函数。在下一章，我们将学习Python的类，我们将看到类和模块很相似：它们也定义名字空间，不过添加了多重拷贝，用继承定制等等支持。

表5-1 模块操作

操作	解释
import 模块名	取得整个模块
from 模块名 import name	从一个模块中取得一个特殊的名字
from 模块名 import*	取得所有的顶层名字
reload(模块名)	强迫重载一个已经载入的模块的代码

练习

1. **基础，导入。**用你最爱用的编辑器，写一个叫*mymod.py*的Python模块，输出三个顶层名字：

   — 一个countLines(name)函数读入一个文件并计算它的行数。(提示：file.readlines()可以为你做大多数工作。)

   — 一个countChars(name)函数读入一个文件并计算其中的字符数。(提示：file.read()返回一个单一字符串。)

   — 一个test(name)函数用一个给定的输入文件名调用上述两个函数。

   一个文件字符串应传递给这三个mymod函数。现在，交互测试你的模块，用import和名字限定得到你的输出。你的PYTHONPATH包括生成的*mymod.py*文件所在的目录吗？试着让你的模块自测试，例如：test("mymod.py")。注意test打开文件两次，若你感到有信心，可以通过传递一个打开的文件对象给这两个count函数，以改进它们。

2. **from/from**。交互测试练习1中你编写的mymod模块，通过使用from直接导入输出内容，首先通过名字，再使用from*变量取得所有的东西。

3. **__main__**。现在，在你的mymod模块中添加一行，只在模块作为脚本运行时才自动调用test函数，试着从系统命令行下运行你的模块；接着交互导入模块并测试它的函数，两种方式都工作吗？

4. **嵌套的导入**。最后，写第二个模块*myclient.py*，导入mymod并测试它的函数；从系统命令行运行*myclient*。若*myclient*用from从mymod中获取对象，从*myclient*的顶层可以访问mymod的函数吗？若用import代替会怎样？试图用两种方式编码，通过导入*myclient*和检测它的__dict__，并进行交互式测试。

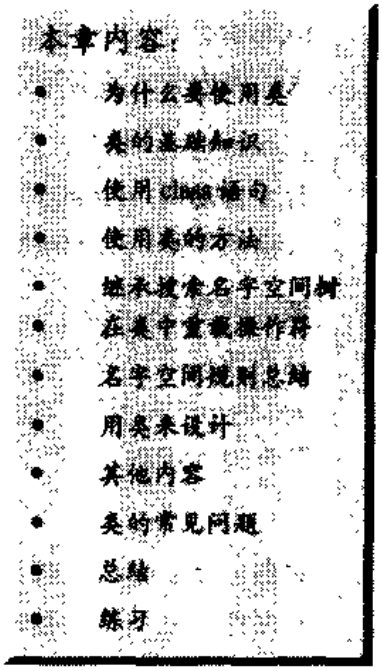
5. **重载**。试验模块重载：在*change.py*中进行测试，无需停止Python解释器，重复的改变调用函数的信息或行为。取决于你的系统，你或许可以在另一个窗口编辑changer，或者是挂起Python解释器，在同一个窗口编辑（在Unix中，Ctrl-z组合键可挂起当前进程，fg命令可以恢复它）。

6. **循环导入**（注10）。在递归导入产生的问题一节中，导入recur1出错，不过若我们重启动Python并交互的导入recur2，错误并没发生，自己测试看一下。为什么导入recur2正常工作，而recur1不行？（提示：在运行代码前，Python在内置模块sys.modules表（一个字典）中存储新的模块，以后导入的时候首先从这个表中取得模块，无论这个模块是否是“完整”的。）现在作为脚本运行recur1：% Python recur1.py。当你交互式导入recur1还会发生同样的错误吗？为什么？（提示：当模块作为程序运行时，它们还没被导入，所以这个例子与交互式导入recur2有同样的影响；recur2是第一个被导入的模块。）当你作为脚本运行recur2的时候会发生什么？

注10：我们应该指出，在实际中循环导入是绝对少见的。事实上，在我们六年的Python编码经验中，我们没有写过或遇到过一个循环导入——除非是在Internet上（在那里这样的事情受到不同寻常的高度关注），以及在写这样的一本书的时候。在另一方面，如果你理解这是一个潜在问题的原因，关于Python的导入语义你就知道很多了。

第六章

类



本章将探讨Python的类（class）——我们用它来实现新类型的实例对象。类是Python面向对象编程（OOP）的主要工具，所以本章里我们将研讨面向对象编程的基本知识。Python的类是用一个新的语句class来创建的。我们将发现，用类定义的对象与我们在本书前面见到的内置类型十分相似。

首先，在起步时你不一定要用类，Python的面向对象部分是可选的。事实上，你用函数这样的简单结构就可以干很多事情。但是类是Python提供的最有用的工具之一，我们希望本章能告诉你为什么。在流行的Python工具中，比如Tkinter图形界面应用编程接口（GUI API）中就用到了类，所以大多数的Python程序员通常会发现，关于类的基本知识是很有用的。

为什么要使用类

还记得我们曾告诉你程序是用来处理一些东西的吗？简单地说，类就是定义一些新东西的方式，它把现实中的对象转换到你的程序中。例如，假设我们决定实现

一个做比萨饼的机器人，这是我们在第四章“函数”里用过的例子。如果我们用类来实现它，我们可以把它在现实世界中的结构和关系描绘得更逼真。

继承（inheritance）

做比萨饼的机器人是机器人的一种，所以有普通的机器人的特性。用面向对象编程的行话来说，它们继承了一般机器人的特性。这些公有的特性只需要实现一次，而在未来建造的各种机器人中都可以重复使用。

合成（composition）

做比萨饼的机器人实际上是一组协调工作的部件。例如，它也许需要能转动的手臂，行走的马达，等等。用面向对象编程的说法，这里的机器人是合成的一个例子，它包含了听它指挥的对象。每一个部件也许都用类编写而成，这些类定义了自身的行为和关系。

当然，我们大多数人并不会花钱做一个比萨饼的机器人。对于任何可以分解为一组对象的应用来说，继承和合成这些面向对象编程概念都是适用的。例如，在典型的图形界面系统里，界面是用一组组件（widget）来描述的（按钮、标签等），当它们的容器出现时它们都会显示出来（合成）。而且，我们也能写我们自己定制的组件，它们可以用一般的界面定制更特别的版本（继承）。

从更具体的编程观点来看，类是Python程序的一个组成部分，就像函数和模块一样。它们是另一种打包逻辑和数据的方式。实际上，类也像模块一样定义了新的名字空间。但与其他我们见过的程序组成单位相比，类有三个关键特征，这使得它在创建新对象时更有用。

多个实例

类是一个产生一个或多个实例的模板。每次我们调用一个类，就产生了一个具有不同名字空间的对象。我们将看到，每个由类产生的对象可以访问类的属性，但有各自不同的存放数据的名字空间。

通过继承来定制

类也支持面向对象编程的继承概念，可以通过覆盖（override）它们的属性来扩展。通过继承可以定义名字空间层次。

操作符重载

通过提供特殊的协议方法，对内置类型适用的操作符也适用于用类定义的对象。例如可以对用户自定义的对象使用分片、合并、索引等操作。

类的基础知识

如果你从来没有接触过面向对象编程，一开始可能会觉得类的概念有点复杂。为了让类的概念容易理解，我们在这里快速地扫描一下实际的类，以说明刚才描述的三个特征。我们以后将逐渐展开细节，而类的基本形式应该是容易理解的。

类可产生多个实例对象

正如我们在第五章“模块”中提到的，类很像模块，差不多就是一个名字空间。但与模块不同的是，类支持多重拷贝、名字空间继承和操作符重载。让我们先看看第一点。

要理解多重拷贝的概念，你必须先理解在Python的面向对象编程模型中有两种对象 —— 类对象和实例对象。类对象定义了缺省的行为，并负责产生实例对象。实例对象是你的程序真正处理的对象，每一个实例都有它自己的名字空间，但都是从创建它的类那里继承的。类对象由语句产生，而实例对象由一个对类的调用产生，每当你调用一个类，你就得到一个新的实例。请注意，我们下面将总结Python面向对象编程的要点。

类对象定义缺省的行为

class 语句创建了一个类对象并赋给它一个名字

像 `def` 一样，Python的 `class` 语句是可执行的，当运行时，它产生一个新的类对象，并赋予它 `class` 首部的类名字。

class 语句内部的赋值生成类的属性

像模块一样，class语句内部的赋值产生了类对象的属性，类的属性通过名字限定来访问（`object.namme`）。

类的属性输出对象的状态和行为

一个类对象的属性记录了状态和行为信息，由该类的所有实例共享，类内部的函数语句def生成方法，它们负责处理实例。

实例对象是由类产生的

像调用一个函数一样调用类就生成了一个新的实例对象

每当调用一个类时，它就产生并返回一个新的实例。

每个实例对象都继承了类的属性，并有自己的名字空间

由类生成的实例对象是新的名字空间，一开始只包含了从产生它的类继承来的类属性。

方法中对self的赋值就生成了实例的属性

在类的方法中，第一个参数（按惯例称为self）引用的是要处理的实例对象，对self的属性的赋值，创建或改变的是实例中的数据，而不是类的数据。

一个例子

除了还有一些细节外，这些就是Python的面向对象编程的一切。让我们看一个实际的例子吧。首先我们用class语句定义一个FirstClass类。

```
>>> class FirstClass:                    # 定义一个类对象
...     def setdata(self, value):        # 定义类方法
...         self.data = value            # self就是实例
...     def display(self):
...         print self.data              # self.data: 每个实例的数据
```

与所有的复合语句一样，类有一个首行，列出了类的名字，接下来的语句体里是一条或多条嵌套的缩进语句。这里嵌套的语句是def，它定义了将要输出的类的行为。def是一个赋值，这里它是赋值给class语句作用域内的名字，所以就生成了类的属性。类内部的函数通常称为方法函数，它们只是普通的函数，但当调用时，第一个参数自动接收一个隐含的实例对象。我们看两个例子：

```
>>> x = FirstClass()                     # 生成两个实例
>>> y = FirstClass()                     # 每一个是新的名字空间
```

通过调用类，我们生成了两个实例对象。更准确地说，此时我们有三个对象——两个实例和一个类，实际上，我们有三个链接起来的名字空间（见示意图6-1）。用面向对象编程的行话说，x是一个FirstClass，y也是。实例一开始是空的，但有指向类的链接，如果我们对实例引用类属性的名字，Python将从类中取数据（除非实例中有同样的名字）。

```
>>> x.setdata("King Arthur")    # 调用方法：self是x或y
>>> y.setdata(3.14159)          # 运行：FirstClass.setdata(y, 3.14159)
```

x和y都没有自己的setdata，实际上，如果在实例中一个属性不存在，Python将沿着链接到类里寻找。这就是Python的继承的运行机制，它发生在引用属性的时候，从链接的对象里查找名字（图6-1中沿“是一个”链接）。

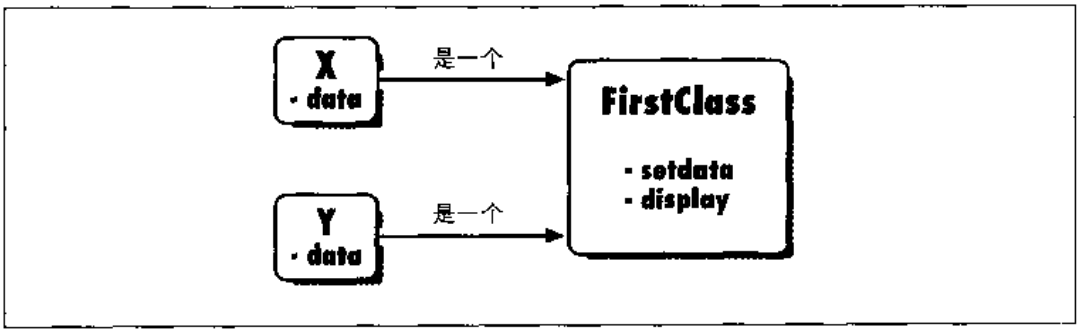


图6-1 类和实例是链接的名字空间对象

在FirstClass的setdata方法里，传来的参数值赋给了self.data。在一个方法内，self自动地引用要处理的实例（x或y），所以赋值将把值存在实例的名字空间里，而不是类的名字空间（这就是图6-1中data名字的来历）。由于类可产生多个实例，方法必须通过self参数取得要处理的对象。当我们调用类的display方法打印self.data时，我们看见每个实例是不同的，而在x和y里display是一样的，因为它来自（继承的）类：

```
>>> x.display()      # 每个实例的self.data不同
King Arthur
>>> y.display()
3.14159
```

通过继承来定制类

与模块不同的是，类可以产生新的部件（子类，subclass）而不必影响已存在的部分。我们已经看到一个类产生的实例对象继承了类的属性。Python也允许从其他类继承新类，这使得我们可以创建一个类层次，较低的类通过覆盖属性而获得特定的行为，这通常被称为框架（*framework*）。这个机制背后的思想是：

超类列在类首部的括号里

为了从别的类继承，只要在class语句的首部把类名字列在括号里。继承的类的称为子类，被继承的类称为超类（superclass）。

类从超类继承属性

就像实例一样，一个类自动地得到所有超类中定义的名字。如果名字在子类中找不到，Python会自动到超类中去找。

实例从所有相关类中继承

实例从产生它的类中继承，也从超类中继承。当查找一个名字时，Python先在实例中找，然后是它的类，然后是超类。

通过子类改变逻辑，而不是直接改变超类

通过在子类里覆盖超类里的名字，子类就覆盖了继承的行为。

一个例子

我们的这个例子是基于前一例的。我们先定义一个新类SecondClass，它继承了FirstClass的所有名字并提供了一个自己的：

```
>>> class SecondClass(FirstClass):          # 继承了setdata
...     def display(self):                  # 改变了display
...         print 'Current value = "%s"' % self.data
```

SecondClass覆盖了display方法，用不同的格式显示。但因为SecondClass定义了相同的属性名，它取代了FirstClass的display属性。继承的工作机制是搜索，先是搜索实例，然后是类、超类，一直到找到这个属性名。由于先在SecondClass里找到了名字display，我们就说SecondClass覆盖了FirstClass的display。换句话说，SecondClass通过改变display方法而成为了FirstClass

的特例。同时，SecondClass（和它创建的实例）仍然继承了FirstClass的setdata方法。图6-2说明了相关的名字空间，让我们举例说明：

```
>>> z = SecondClass()
>>> z.setdata(42)              # 在FirstClass里找到了setdata
>>> z.display()                # 在SecondClass里找到了重定义的方法
Current value = "42"
```

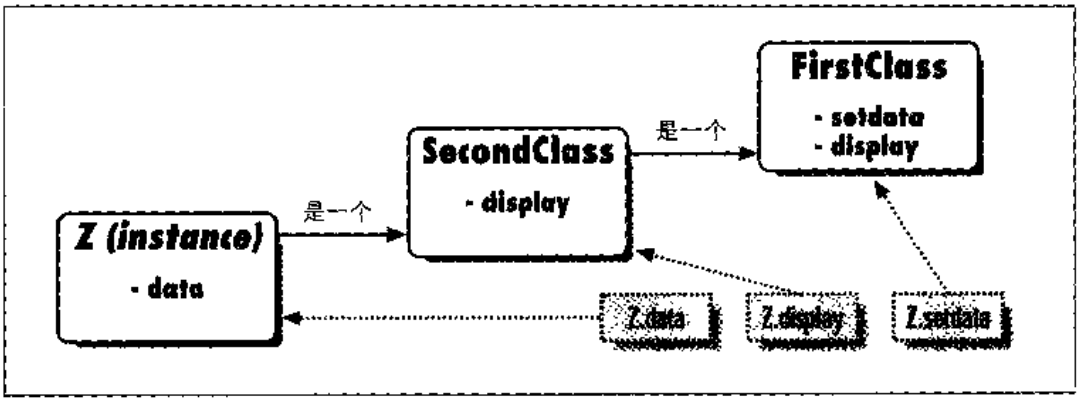


图6-2 通过覆盖继承的名字来定制

同前面一样，我们调用了SecondClass就产生了一个它的实例。setdata仍然是用FirstClass中定义的，但这次display来自SecondClass，它打印了不同的信息。这里有一个关于面向对象编程的要点：在SecondClass中重定义的方法是在FirstClass之外的，它不影响FirstClass对象，比如前面的x：

```
>>> x.display()                # x仍然是FirstClass实例（旧的信息）
New value
```

当然这只是一个人为简化设计的例子，但一般而言，由于可以在外部的部件（子类）里做改变，类对于扩展和重用的支持要胜过模块和函数。

类可以重载Python的操作符

最后让我们快速的扫描一下类的第三个特征：操作符重载。简单地说，操作符重载使我们可以用类实现的对象进行与内置类型一样的操作：+、[:]等等。尽管我们可以用方法来实现行为，但操作符重载让我们实现的对象与Python的对象模

型结合得更紧密。而且，因为操作符重载使我们的对象与内置的一样，这使得对象的接口更一致和易于学习。要点如下：

__X__的名字的方法是特殊的挂钩（hook）

Python通过特殊命名的方法拦截操作符，以实现操作符重载。

Python在计算操作符时会自动调用这样的方法

例如，如果对象继承了__add__方法，当它出现在+表达式中时会调用这个方法。

类可以重定义大多数内置的操作符

有几十个特殊的操作符方法用于捕捉几乎每一个内置的操作。

操作符重载使得类与Python的对象模型紧密集成

通过重载，用户定义的对象就像内置的一样。

一个例子

这一次，我们定义了一个SecondClass的子类，它有三个特殊的方法：__init__是在创建新的实例时调用的（self就是新的ThirdClass对象），而__add__和__mul__是当实例出现在+和*表达式时分别调用的：

```
>>> class ThirdClass(SecondClass):                # 是一个SecondClass
...     def __init__(self, value):                # 当调用"ThirdClass(value)"时
...         self.data = value
...     def __add__(self, other):                  # 当用"self + other"时
...         return ThirdClass(self.data + other)
...     def __mul__(self, other):
...         self.data = self.data * other          # 当用"self * other"时

>>> a = ThirdClass("abc")          # 调用新的__init__
>>> a.display()                    # 继承的方法
Current value = "abc"

>>> b = a + 'xyz'                  # 调用新的__add__：产生一个新的实例
>>> b.display()
Current value = "abcxyz"

>>> a * 3                          # 调用新的__mul__：改变实例本身
>>> a.display()
Current value = "abcabcabc"
```

ThirdClass是一个SecondClass，所以它的实例从SecondClass继承了display。但ThirdClass的调用中有一个参数"abc"，它作为__init__构造函数的参数，并赋值给self.data。而且，ThirdClass对象可以出现在+和*表达式中，Python把操作符左边的对象传给self参数，而把右边的值传给other参数，见图6-3。

像__init__和__add__这样的特殊方法同样可以被子类和对象继承。注意，这里的__add__方法生成了一个新对象（用结果值调用ThirdClass），而__mud__改变了当前的对象（重新给self赋值）。对于内置对象（如数的列表）来说，*操作符是生成新的对象，但在类中你可以用你喜欢的任何方式来解释（注1）。

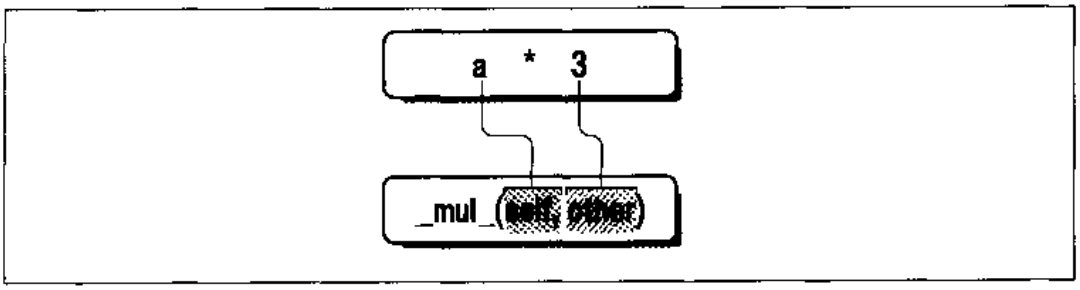


图6-3 操作符与特殊的方法

使用class语句

前面的内容理解了吗？如果没有，也别着急。既然我们已经做了快速旅行，我们将更深入一些，了解更多的细节。我们在第一个例子里介绍了class语句，现在我们更正式地介绍它。像C++一样，class语句是Python的主要面向对象编程工具，与C++不同的是，Python的class不仅是一个说明，像def一样，class是一个对象建造者，是一个隐含的赋值（运行时它产生了一个类对象，并用列在首部的类名字来引用这个类对象）。

注1：但你很可能不应该（一个评论者称这个例子为"邪恶！"）。常见的做法表明重载的操作符应该与内置的操作符工作方式相同。在我们这个例子中，__mul__方法应该返回一个新的结果对象，而不是直接修改实例（self），一个mul方法也许比一个*重载更好（如a.mul(3)而不是a*3）。另一方面，一个人的常见做法却可能是另一个人的任意限制。

一般形式

我们在快速的旅行中看到，class是一个复合语句，语句体是一组缩进的语句。在首部，超类（列在括号里）紧跟在类名后面，用逗号分开。列表中如有多个超类就表示多重继承（本章后面会说明）：

```
class <name>(superclass,...):          # 赋值给name
    data = value                       # 共享的类数据
    def method(self,...):              # 方法
        self.member = value            # 实例的数据
```

在class语句内，用特殊命名的方法重载操作符。例如，如果定义了__init__方法的话，将在创建实例对象时调用它。

例子

在本章的开头，我们曾提到类差不多就是名字空间——一个为客户输出数据和方法定义名字（称为属性）的工具。那么，怎样从一个语句得到名字空间呢？解释如下：

就像模块一样，嵌套在class语句体中的语句创建了它的属性。当Python执行一个class语句时（而下是调用一个类），它将从上到下执行语句体里的语句。在这个过程中发生的赋值就在类的局部作用域里创建名字，这些名字就成了该类对象的属性。所以类既像模块和又像函数：

- 与函数一样，class语句是一个局部作用域，包含了嵌套的赋值语句创建的名字。
- 与模块一样，在class语句中赋值的名字成为了类对象的属性。

Python类的主要特征是它们的名字空间也是继承的基础。如果在一个类或实例对象中没有找到属性，将继续在超类里找寻。由于class是一个复合语句，任何语句都可以嵌套在它的语句体内——例如，print、if、=和def等。嵌套的def定义了类的方法，而其他赋值语句还定义了属性。例如，假设我们运行下面的类：

```
class Subclass(aSuperclass):                    # 定义子类
```

```
    data = 'spam'                          # 给类属性赋值
    def __init__(self, value):             # 给类属性赋值
        self.data = value                  # 给实例属性赋值
    def display(self):
        print self.data, Subclass.data     # 实例，类
```

这个类有两个def语句，说明了两个方法。它还有一个赋值语句，由于data是在class中赋值，它位于类的局部空间而成为类对象的一个属性。像所有类属性一样，data由所有的该类的实例继承和共享（注2）：

```
>>> x = Subclass(1)              # 生成两个实例对象
>>> y = Subclass(2)              # 每一个都有自己的"data"属性
>>> x.display(); y.display()     # "self.data"是不同的，"Subclass.data"是相同的
1 spam
2 spam
```

当我们运行这段代码时，在两个地方都有名字data —— 在实例和类中。类的display方法打印出两个版本，首先是引用self实例，然后是类本身。由于类是有属性的对象，我们可以直接引用它们的属性名字，即使不涉及实例。

使用类的方法

知道了函数也就知道了类的方法。方法就是嵌套在class语句体里的def语句创建的函数对象。抽象地看，方法描述了实例的行为。从程序的角度看，方法与普通函数的工作机制是一样的，只是有一点重要区别：方法的第一个参数总是（隐含地）一个实例对象，也就是调用方法的主体。换句话说，Python自动地把对实例的方法调用转换为对类的方法调用：

```
instance.method(args...)   => 变成了 => class.method(instance, args...)
```

这里的class是由Python的继承搜索过程决定的。方法中的特殊的第一个参数按惯例称为self，它很像C++中的this指针，但Python的方法必须显式地用self来获取或者改变要处理的实例的属性。

注2：如果你用过C++，你也许会发现这与C++的类的静态数据相似 —— 存储在类里而独立于实例的数据。在Python里它并不特殊：所有的类属性只是在class语句里赋值的名字，无论它们是函数（C++的方法）或是别的东西（C++的数据成员）。

例子

让我们看一个例子，假设我们定义了如下的类：

```
class NextClass:                        # 定义类
    def printer(self, text):            # 定义方法
        print text
```

printer引用了一个函数对象，由于它的赋值是在class语句内，所以它是一个类属性而且可以由该类的所有实例继承。printer可以通过两种方式调用——通过实例或通过类：

```
>>> x = NextClass()                     # 生成实例
>>> x.printer('Hello world!')           # 调用它的方法
Hello world!
```

当像这样通过限定实例来调用时，printer的self参数自动地被赋予实例对象(x)，而字符串"Hello world!"传给text参数。在printer内self可以访问和设置每个实例的数据，因为它引用的是正在处理的实例。我们也可以通过类来调用printer，我们可以显式地把实例传给self参数：

```
>>> NextClass.printer(x, 'Hello world!')      # class method
Hello world!
```

如果在类调用方式里传的是同一个实例的话，以上两种方式效果是完全一样的。很快我们将看到通过类的调用是扩展（而不是代替）继承的行为的基础。

继承搜索名字空间树

class语句这样的名字空间工具完全是为了支持名字的继承。Python里继承发生在引用一个对象的时候，而且将搜索一个属性定义的树（一个或多个名字空间）。每次你使用object.attr这样一个表达式时，这里的object是一个实例或类对象，Python将从object往上搜索，直到找到第一个attr。因为在树中较低层的定义重定义了高层的属性，所以继承是定制（specialization）的基础。

建立属性树

图 6-4 说明了名字空间树的建立方式，一般地：

- 实例的属性是由方法里对 self 属性赋值生成的。
- 类的属性是由 class 语句内的赋值语句创建的。
- 到超类的链接是由 class 语句首部的括号里的类列表产生的。

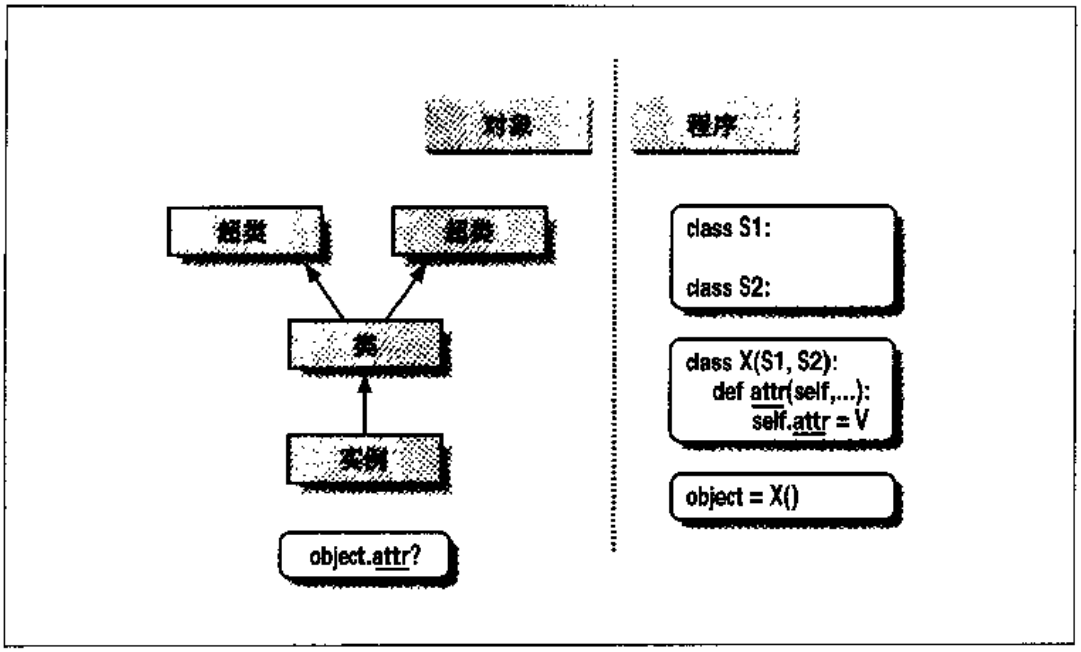


图 6-4 名字空间树与继承

结果就形成了一个名字空间树，由一个实例开始，到产生它的类，再到所有的超类。每当你引用一个实例的属性时，Python从实例到超类向上搜索这个树（注3）。

定制继承的方法

我们刚才描述的继承搜索模式是一种很好的定制系统的方法，因为继承时先在子

注 3： 这个描述并不完全对，因为实例和类属性也可以通过class语句外的赋值来创建。但那并不常见而且有时更容易出错。在Python里所有的属性缺省地都可以被访问，本章后面我们会谈到私有属性。

类中查找名字，所以子类可以覆盖超类的名字来代替缺省的行为。事实上你可以把整个系统建成一个类层次，通过增加新的子类来扩展而不是改变已存在的逻辑。

由覆盖继承来的名字的概念可以引申出很多定制的技术，例如子类可以完全代替超类的名字，或者通过回调超类来扩展。我们已见过替代的例子，这里是一个展示扩展的例子：

```
>>> class Super:
...     def method(self):
...         print 'in Super.method'
...
>>> class Sub(Super):
...     def method(self):                    # 覆盖方法
...         print 'starting Sub.method'     # 这里是新增的操作
...         Super.method(self)               # 调用缺省的操作
...         print 'ending Sub.method
...
```

直接对超类的方法进行调用是这里的关键。Sub 类用自己的定制版本取代了 Super 类的 method 函数，但 Sub 类回调了 Super 类输出的缺省版本，换句话说，Sub 类的方法扩展了 Super 类方法的行为，而不是完全代替：

```
>>> x = Super()              # 生成 Super 的实例
>>> x.method()               # 调用 Super.method
in Super.method

>>> x = Sub()                # 生成 Sub 的实例
>>> x.method()               # 调用 Sub.method，间接地调用了 Super.method
starting Sub.method
in Super.method
ending Sub.method
```

通常这种扩展用在构造函数中，由于特殊的 __init__ 方法是继承的，当实例创建时只能找到并运行一个，为了运行超类的构造函数，应通过子类的 __init__ 方法间接地调用超类的 __init__（通过类引用的方式，如 Class.__init__(self, ...)）。

扩展只是同超类接口的一种方式，下面展示了一些常见的模式：

- Super定义了一个method函数和一个delegate方法，并期待由子类实现一个action方法。
- Inheritor不提供新的名字，所以与Super类是一样的。
- Replacer 用自己的版本取代了Super类的方法。
- Extender通过重定义和回调来定制Super类的方法。
- Provider实现了Super类的delegate方法所期待的action方法。

```
class Super:
    def method(self):
        print 'in Super.method'        # 缺省
    def delegate(self):
        self.action()                  # 期待的

class Inheritor(Super):
    pass

class Replacer(Super):
    def method(self):
        print 'in Replacer.method'

class Extender(Super):
    def method(self):
        print 'starting Extender.method'
        Super.method(self)
        print 'ending Extender.method'

class Provider(Super):
    def action(self):
        print 'in Provider.action'

if __name__ == '__main__':
    for klass in (Inheritor, Replacer, Extender):
        print '\n'+ klass.__name__ + '...'
        klass().method()
print'\nProvider...'
Provider().delegate()
```

一些值得指出的要点是：上面例子中结尾处的自测试代码创建了三个类的实例，因为类是对象，所以你可以把它们放在元组里并创建实例(后面详述)。类与模块一样有特殊的__name__属性，它设置为类的名字。当你通过一个Provider实例调用delegate方法时，Python通过搜索树而找到Provider的action方法，在Super类的delegate方法中，self引用的是Provider的一个实例。

```
% python specialize.py

Inheritor...
in Super.method

Replacer...
in Replacer.method

Extender...
starting Extender.method
in Super.method
ending Extender.method

Provider...
in Provider.action
```

在类中重载操作符

在本章开始我们介绍了操作符重载，我们来看一下常用的重载方法。这里是关于重载的要点回顾：

- 操作符重载使得类能拦截标准的 Python 操作。
- 类可以重载所有的 Python 的表达式操作符。
- 类可以重载对象操作：print、函数调用、限定（qualification）等。
- 重载使得类的实例看起来更像内置的。
- 重载是通过特殊命名的类方法来实现的。

这里是一个重载的简单例子。如果我们在一个类里提供特殊命名的方法，当该类的实例出现在相关的操作时Python会自动地调用对应的特殊方法。例如，Number类提供了一个方法拦截构造函数（__init__），以及一个减法方法（__sub__）。这些特殊的方法是你拦截内置操作的挂钩：

```
class Number:
    def __init__(self, start):                  # 在Number(start)时调用构造函数
        self.data = start
    def __sub__(self, other):                   # 在表达式 instance - other
        return Number(self.data - other)        # 结果是一个新的实例
```

```
>>> from number import Number          # 从模块里取类
>>> X = Number(5)                      # 调用 Number.__init__(X, 5)
>>> Y = X - 2                          # 调用Number.__sub__(X, 2)
>>> Y.data
3
```

常见的重载方法

内置对象如整数和列表的所有操作都有对应的特殊命名的方法用于重载。表 6-1 列出了常用的，更多的本书就没有时间涉及了。请参见其他Python书籍或《Python 库参考》中的完整的特殊方法列表。所有的重载方法首尾都有两个下划线，以便区别于你在类中定义的别的名字。

表 6-1 常用的操作符重载方法

方法名	重载的操作说明	调用表达式
__init__	构造函数	创建对象：Class()
__del__	析构函数	释放对象的时候
__add__	'+'	X + Y
__or__	'|'（按位或）	X | Y
__repr__	打印，转换	print X, `X`
__call__	函数调用	X()
__getattr__	属性引用（限定）	X.undefined
__getitem__	索引	X[key]，for循环，in测试
__setitem__	按索引赋值	X[key] = value
__getslice__	分片	X[low:high]
__len__	长度	len(X)
__cmp__	比较	X == Y, X < Y
__radd__	右边的操作符 '+'	Noninstance (非实例) + X

例子

让我们举几个例子来说明表 6-1 的一些方法。

__getitem__拦截了所有的索引操作

__getitem__方法拦截了实例的索引操作：当一个实例X出现在X[I]这样的索引表达式里时，Python调用实例继承的__getitem__方法（如果有定义），把X作为第一个参数而把方括号里的索引作为第二个参数。例如，下面的类返回索引值的平方：

```
>>> class indexer:
...     def __getitem__(self, index):
...         return index ** 2
...
>>> X = indexer()
>>> for i in range(5):
...     print X[i],              # X[i]将调用__getitem__(X, i)
...
0 1 4 9 16
```

这里有一个对初学者并不显而易见但却是非常有用的特殊技巧，当我们在第二章“基本语句”里介绍for语句时，我们曾提到它是从索引0开始索引一个序列，直到检测到出界异常。所以，__getitem__也是用来重载遍历和成员测试操作的方法，一举两得。对任何内置的或用户定义的对象来说，如果它响应索引操作，也就自动地响应遍历和成员测试操作：

```
>>> class stepper:
...     def __getitem__(self, i):
...         return self.data[i]
...
>>> X = stepper()               # X是一个stepper对象
>>> X.data = "Spam"
>>>
>>> for item in X:              # for循环调用__getitem__
...     print item,
...
S p a m
>>>
>>> 'p'in X                     # 'in'操作符也调用__getitem__
1
```

__getattr__捕捉未定义的属性引用

__getattr__方法拦截属性引用。具体地说，每当你试图引用一个实例的未定义

（不存在）属性名时，就会以属性名为参数调用它。如果Python在继承树搜索中找到了属性就不会调用它。所以，__getattr__作为通用的形式响应对属性的访问，例如：

```
>>> class empty:
...     def __getattr__(self, attrname):
...         if attrname == "age":
...             return 36
...         else:
...             raise AttributeError, attrname
...
>>> X = empty()
>>> X.age
36
>>> X.name
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 6, in __getattr__
AttributeError: name
```

这里empty类和它的实例X没有自己的属性，所以访问X.age会调用__getattr__方法，self将赋值为实例X，attrname赋值为未定义的属性名串"age"，我们的类对X.age引用将返回一个值，使得age看起来就像一个属性。

这个类对别的属性不知道如何处理，它将引发内置的AttributeError异常，表示这是个未定义名字，X.name就引发了异常。当我们展示授权（delegation）时还会看到__getattr__，在第七章“异常”里会更多地介绍异常。

__repr__返回一个说明字符串

这里是一个例子，它演示了构造函数（__init__）和我们已见过的+方法（__add__），而且也定义了一个__repr__方法，它返回实例的说明字符串。反引号(``)把self.data转换为字符串。如果定义了__repr__的话，__repr__当打印或把对象转换为字符串时，会自动地调用__repr__方法：

```
>>> class adder:
...     def __init__(self, value=0):
...         self.data = value                    # 初始化data
...     def __add__(self, other):
```

```
...         self.data = self.data + other
...     def __repr__(self):
...         return `self.data`                  # 转换为字符串
...
>>> X = adder(1)          # __init__
>>> X + 2; X + 2          # __add__
>>> X                     # __repr__
5
```

还有许许多多的例子，大多数与我们见过的相似，都是拦截学过的内置操作的挂钩，不过一些重载方法有特殊的参数列表和返回值。后面还会看到一些例子，更完整的说明必须参考其他文档。

名字空间规则总结

在我们学习了类和实例对象后，关于Python的名字空间的内容就完整了。我们对名字解析的规则做一个快速的总结。你需要记住的第一件事是限定型名字（object.X）和非限定型名字（X）是不同的：

- 非限定型名字（X）与作用域有关。
- 限定型名字（object.X）使用对象object的名字空间。
- （在模块和类的）作用域内初始化对象的名字空间。

非限定型名字

非限定型名字遵循我们在第四章描述的关于函数的LGB规则。

赋值：X=value

X是局部的：除非用global声明，将在局部作用域里创建和改变X。

访问：X

在当前的局部作用域里找寻X，然后依次是全局作用域和内置作用域。

限定型名字：对象的名字空间

限定型名字访问特定对象的属性，并遵循我们讨论过的与模块一样的规则。对实例和类对象来说，访问规则还包括继承搜索过程：

赋值：object.X=value

在被引用的对象 object 的名字空间内创建和改变属性名 X。

访问：object.X

在对象 object 里搜寻属性名 X，然后是搜寻所有的超类（但不包括模块）。

名字空间字典

在第五章中我们看到，模块的名空间实际上是用字典实现的，并用内置的 __dict__ 属性输出。对类和实例来说同样如此：引用属性时实际上在内部是对字典的索引，而属性继承只是搜索链接的字典。

下面的例子跟踪了在操作类的过程中名字空间字典增长的情况，要点是：每当在这两个类中对self的属性赋值时，就在实例的名字空间字典里创建（或改变）一个对应的项，注意不是类的名字空间。不同实例的名字空间内的数据是不一样的，它们也都有指向它们的类的名字空间的链接。例如，X.hello最后是在 super 类的名字空间字典里找到的：

```
>>> class super:
...     def hello(self):
...         self.data1 = "spam"
...
>>> class sub(super):
...     def howdy(self):
...         self.data2 = "eggs"
...
>>> X = sub()           # 生成一个新的名字空间（字典）
>>> X.__dict__
{}
>>> X.hello()           # 改变实例的名字空间
>>> X.__dict__
{'data1': 'spam'}
```

```
>>> X.howdy()            # 改变实例的名字空间
>>> X.__dict__
{'data2': 'eggs', 'data1': 'spam'}

>>> super.__dict__
{'hello': <function hello at 88d9b0>, '__doc__': None}

>>> sub.__dict__
{'__doc__': None, 'howdy': <function howdy at 88ea20>}

>>> X.data3 = "toast"
>>> X.__dict__
{'data3': 'toast', 'data2': 'eggs', 'data1': 'spam'}
```

注意我们在第一、二章碰到的dir函数对类和实例也适用。事实上，它适用于任何有属性的对象。dir(object)等同于object.__dict__.keys()，返回一个列表。

用类来设计

到目前为止，我们都在专注于Python的面向对象编程工具——类。但面向对象编程也是与设计相关的——如何用类来设计有用的对象。在这一节里，我们将涉及到面向对象编程的核心概念，还将看到一些比前面更实际的例子。我们在这里提到的大多数设计术语需要更多的解释，然而在这里难以详述，如果这一节激起了你的好奇，我们建议你参考关于面向对象编程设计或设计模式的书籍。

Python和面向对象编程

Python的面向对象编程实现可以总结为三个概念：

继承（inheritance）

在Python里是基于属性查找。

多态（polymorphism）

对于X.method来说，method方法的含义依赖于X的类型。

封装（encapsulation）

方法和操作符实现了行为，而数据隐藏是一种约定。

现在，你应该已经很好地理解了Python的继承究竟是什么。Python式的多态来自于它的无说明类型。因为属性总是在运行时决定的，具有同样接口的对象可以互相交换，客户不必知道是什么类型的对象实现了被调用的方法（注4）。Python的封装是为了打包，不是隐藏，隐藏是可选的，本章后面会再讨论。

面向对象编程和继承："is-a（是一个）"

我们已经深入地讨论了继承的机制，但我们还想通过一个例子来展示如何用它来描述现实世界中的关系。从一个程序员的观点看，继承是属性引用和对名字的搜索，从实例到类再到超类。从设计者的观点看，继承是一种指定集合成员身份的方式。一个类就定义了一个属性集合，并可以被更特定的集合继承（子类）。

为了说明，我们就实现本章开头谈到的做比萨饼的机器人的例子。假设我们已决定了开一个比萨饼店，需要做的第一件事就是雇佣员工为顾客服务，做比萨饼等等。我们决定给我们的机器人付工资。

我们的比萨饼店团队可由文件*employee.py*里的这些类来定义。它定义了四个类和一些测试代码。最一般的类Employee（雇员）提供了常见的行为如涨工资（giveRaise）和打印（__repr__）。有两种雇员，所以有两个Employee的子类——Chef（主厨）和Server（服务员）。都覆盖了继承来的work方法以打印出更特定的信息。最后，我们的比萨饼机器人用更特殊的类来描述：PizzaRobot，它是Chef，也是Employee。用面向对象编程的行话来说，我们称这些关系是"is-a（是一个）"链接：一个比萨饼机器人是一个主厨，也是一个雇员。

```
class Employee:
    def __init__(self, name, salary=0):
        self.name   = name
```

注4：有些OOP语言也把多态性定义为基于参数类型的函数重载。由于Python里没有类型声明，这种概念就不适用，但基于类型的选择也可以用if和type(x)函数来表达（例如，if type(X) is type(0): doIntegerCase()）。

```
        self.salary = salary
    def giveRaise(self, percent):
        self.salary = self.salary + (self.salary * percent)
    def work(self):
        print self.name, "does stuff"    def __repr__(self):
        return "<Employee: name=%s, salary=%s>"% (self.name, self.salary)

class Chef(Employee):
    def __init__(self, name):
        Employee.__init__(self, name, 50000)
    def work(self):
        print self.name, "makes food"

class Server(Employee):
    def __init__(self, name):
        Employee.__init__(self, name, 40000)
    def work(self):
        print self.name, "interfaces with customer"

class PizzaRobot(Chef):
    def __init__(self, name):
        Chef.__init__(self, name)
    def work(self):
        print self.name, "makes pizza"

if __name__ == "__main__":
    bob = PizzaRobot('bob')          # 生成一个名为bob的机器人
    print bob                        # 运行继承的__repr__
    bob.giveRaise(0.20)              # 给bob长20%工资
    print bob; print

    for klass in Employee, Chef, Server, PizzaRobot:
        obj = klass(klass.__name__)
        obj.work()
```

当我们运行这个模块的自测试代码时，我们创建了一个比萨饼制作机器人bob，它从三个类继承了名字：PizzaRobot、Chef和Employee。例如，打印bob就会运行Employee.__repr__方法，而给bob涨工资就会访问Employee.giveRaise方法，因为继承在那里找到了它。

```
C:\python\examples> python employees.py
<Employee: name=bob, salary=50000>
<Employee: name=bob, salary=60000.0>
```

```
Employee does stuff
Chef makes food
Server interfaces with customer
PizzaRobot makes pizza
```

在这样的类层次里，你可以生成任何类的实例，而不只是在底层的类。例如，在上面的for循环中就产生了四个类的实例：每一个在被要求工作时的响应是不同的，因为各自的work方法是不同的。实际上这些类就是模仿了现实世界的对象，work暂时是打印一条信息，但以后可以扩展为真正的工作。

面向对象编程与合成（composition）："has-a（有一个）"

本章开头介绍了合成的概念。从程序员的观点看，合成是指在一个容器对象里嵌入其他对象，并激活它们来实现容器的方法。对设计者来说，合成是另一种表达问题的关系的方式。但合成不再针对集合成员，它与部件——整体的部分有关。合成也反映了两个部分间的关系。当面向对象编程人员谈起它时，通常称它为"has-a（有一个）"关系。

既然我们已经实现了雇员类，让我们把它们送到比萨饼店，忙碌起来。我们的比萨饼店是一个合成的对象，它有一个烤炉，以及服务员和主厨这两种雇员。当一个顾客进来买比萨饼时，店里的各个部件就运行起来——服务员接订单，主厨开始做饼，等等。下面的例子模仿了这个场景的所有对象和关系：

```
from employees import PizzaRobot, Server

class Customer:
    def __init__(self, name):
        self.name = name
    def order(self, server):
        print self.name, "orders from", server
    def pay(self, server):
        print self.name, "pays for item to", server

class Oven:
    def bake(self):
        print "oven bakes"

class PizzaShop:
    def __init__(self):
```

```
        self.server = Server('Pat')          # 嵌入其他对象
        self.chef   = PizzaRobot('Bob')      # 一个名为bob的机器人
        self.oven   = Oven()

    def order(self, name):
        customer = Customer(name)            # 激活别的对象
        customer.order(self.server)          # 顾客向服务员购买
        self.chef.work()
        self.oven.bake()
        customer.pay(self.server)

if __name__ == "__main__":
    scene = PizzaShop()                      # 生成这个合成对象
    scene.order('Homer')                     # 模拟Homer的订单
    print '...'
scene.order('Shaggy')                        # 模拟Shaggy的订单
```

PizzaShop类是一个容器，也是一个控制器，它的构造方法生成了上节定义的雇员类Employee的实例，还定义了一个Oven（烤炉）类。当运行模块的自测代码时调用了PizzaShop的order方法，内部的对象就依次行动。注意每个订单对我们而言都是一个新的Customer（顾客），并把嵌入的Server对象传给Customer的order方法，顾客来来往往，而服务员是比萨饼店的一部分。注意雇员仍然是一个继承关系，合成和继承是互补的工具。

```
C:\python\examples> python pizzashop.py
Homer orders from <Employee: name=Pat, salary=40000>
Bob makes pizza
oven bakes
Homer pays for item to <Employee: name=Pat, salary=40000>
...
Shaggy orders from <Employee: name=Pat, salary=40000>
Bob makes pizza
oven bakes
Shaggy pays for item to <Employee: name=Pat, salary=40000>
```

当我们运行这个模块时，我们的比萨饼店处理了两个订单——先是Homer，然后是Shaggy。这只是一个简单的模拟，一个真正的比萨饼店有更多的部件，而这里没有真正的比萨饼。最重要的是，类几乎可以表示任何句子里表达的对象和关系，用类来表示名词，方法表示动词。

请留意：类和持久性存储

除了允许我们模拟现实中的互动关系以外，比萨饼店类也可以作为一个持久存储的餐厅数据库的基础。正如我们将在第十章“框架和应用”中看到的，用Python的pickle或shelve模块，类的实例可以一步就存到磁盘上。对象的pickle接口非常易于使用：

```
import pickle
object = someClass()
file   = open(filename, 'W')          # 创建外部文件
pickle.dump(object, file)             # 把对象存在文件里

file   = open(filename, 'r')
object = pickle.load(file)            # 取回对象
```

shelve与此类似，但它自动地把对象存放到按键访问的数据库里：

```
import shelve
object = someClass()
dbase  = shelve.open('filename')
dbase['key'] = object                 # 存在key下
object = dbase['key']                 # 取回
```

(pickle把对象转换为字节流序列，可以存到文件里，在网络上传送等。)在我们的例子中，用类模仿雇员，我们就免费地得到了一个简单的雇员和比萨饼店的数据库：pickle使得在Python程序执行后这些对象可永久保存。第十章有更多pickle的细节。

面向对象编程和授权

面向对象程序员也会谈到授权(delegation)，指的是内嵌有其他对象的控制对象，它们向内嵌对象传递响应操作请求。控制对象负责管理跟踪访问这样的事务。Python里授权经常是用__getattr__方法来实现的，因为它拦截了对不存在的属性的访问，一个包裹类可以用__getattr__把请求转向被包裹的类，例如：

```
class wrapper:
    def __init__(self, object):
        self.wrapped = object                    # 保存对象
    def __getattr__(self, attrname):
        print 'Trace:', attrname                 # 跟踪访问
```

```
        return getattr(self.wrapped, attrname)    # 授权访问
```

你可以用这个wrapper类控制任何有属性的对象——列表、字典，甚至是类和实例。这里只是简单地打印每个访问的跟踪信息：

```
>>> from trace import wrapper
>>> x = wrapper([1,2,3])                  # 封装一个列表
>>> x.append(4)                           # 授权给列表的方法
Trace: append
>>> x.wrapped                             # 打印成员
[1, 2, 3, 4]

>>> x = wrapper({"a": 1, "b": 2})         # 封装一个字典
>>> x.keys()                              # 授权给字典的方法
Trace: keys
['a', 'b']
```

扩展内置对象类型

类也常用来扩展Python内置类型的功能，以支持更特殊的数据结构。例如，为了给列表增加队列插入和删除方法，你可以在类里包装一个列表对象，然后输出处理列表的插入和删除方法。

还记得我们在第四章写的集合函数吗？这里我们用Python的类来重写。下面的类实现了一个新的集合类型，我们把前面写的函数改为方法，并增加了一些基本的操作符重载。这个类主要是封装了一个带有集合操作的列表，但由于它是一个类，它也支持多个实例和用继承定制。

```
class Set:
    def __init__(self, value = []):    # 构造函数
        self.data = []                  # 管理一个列表
        self.concat(value)

    def intersect(self, other):         # other是任何序列
        res = []
        for x in self.data:
            if x in other:              # 取公共的成员
                res.append(x)
        return Set(res)                 # 返回一个新的集合

    def union(self, other):             # other是任何序列
```

```
        res = self.data[:]                  # 拷贝列表
        for x in other:                     # 增加other里的成员
            if not x in res:
                res.append(x)
        return Set(res)

    def concat(self, value):                # value: 列表, 集合...
        for x in value:                     # 删除重复的
            if not x in self.data:
                self.data.append(x)

    def __len__(self):          return len(self.data)          # len(self)
    def __getitem__(self, key): return self.data[key]          # self[i]
    def __and__(self, other):   return self.intersect(other)  # self & other
    def __or__(self, other):    return self.union(other)       # self | other
    def __repr__(self):         return 'Set:' + `self.data`    # print
```

通过重载索引操作，我们的集合类可以冒充一个列表。由于本章的练习要求你扩展这个类，更多的解释就留到附录三的练习答案里。

多重继承

当我们讨论class语句的细节时，我们曾提到在首行的括号里可列出多个超类。如果你这样写，你就是在使用多重继承（multiple inheritance），类和它的实例从所有超类中继承名字。当搜索一个属性时，Python在类的首部从左至右的搜索超类直到找到为止。从技术上讲，搜索是从左至右深度优先的，因为任何一个超类也许还有它自己的超类。

理论上讲，多重继承适合于描述属于多个集合的对象。例如，一个人也许是一个工程师，一个作家，一个音乐家等等，从所有的集合继承属性。但实际上多重继承是一个高级工具，如果用得太多会变得复杂，而如果使用得当它是有用的工具。

多重继承的一种常见的方式是从超类中“混入（mix in）”通用方法。这样的超类称为混入类（*mixin* class），它们通过继承把方法提供给子类。例如，Python缺省的打印类实例的方法不是十分有用：

```
>>> class Spam:
...     def __init__(self):                # 没有__repr__
...         self.data1 = "food"
```

```
...
>>> X = Spam()
>>> print X                                  # 缺省格式：类，地址
<Spam instance at 87f1b0>
```

正如前面的操作符重载，你可提供一个__repr__方法实现自己的打印。但与其在每一个想打印的类里写一个__repr__，为什么不写一个通用的工具类，然后让其他类继承呢？

这就是混入的目的。下面的例子定义了一个名为Lister的混入类，它为所有继承它的类重载了__repr__方法，它只是扫描实例的属性字典（__dict__）并生成一个显示所有实例属性的名字和值的字符串。Lister的格式化逻辑可用于任何子类的实例，它是一个通用工具。

Lister用了两个特别的技巧来提取实例的类名和地址。实例有一个内置的__class__属性的值的是对应的类名，而类有一个__name__属性的内容是类的首行里的名字，所以self.__class__.__name__就取到了实例的所属类的名字。内置的id函数取得了实例的内存地址：

```
# Lister 可以混入任何类，并通过对__repr__的继承而提供实例的格式化打印
# self 是最低层的实例

class Lister:
    def __repr__(self):
        return ("<Instance of %s, address %s:\n%s>" %
                            (self.__class__.__name__,     # 类名
                             id(self),                    # 地址
                             self.attrnames()) )          # “属性名=值”列表
    def attrnames(self):
        result = ''
        for attr in self.__dict__.keys():        # 扫描实例的名字空间字典
            if attr[:2] == '__':
                result = result + "\tname %s=<built-in>\n" % attr
            else:
                result = result + "\tname %s=%s\n" % (attr, self.__dict__[attr])
        return result
```

现在，Lister对你写的任何类都是有用的——甚至对已有一个超类的类。这就是多重继承派用场的地方：把Lister放在类首部的超类列表中，你就免费的得到了__repr__，原有的超类仍然有效：

```
from mytools import Lister                  # 获得工具类

class Super:
    def __init__(self):                     # 超类的__init__
        self.data1 = "spam"

class Sub(Super, Lister):                   # 混入__repr__
    def __init__(self):                     # Lister可访问self
        Super.__init__(self)
        self.data2 = "eggs"                 # 别的实例属性
        self.data3 = 42

if __name__ == "__main__"
    X = Sub()
    print X                                 # 混入的repr
```

这里，Sub从Super和Lister继承名字，当你生成并打印一个Sub实例时，你将得到定制的由Lister混入的打印：

```
C:\python\examples> python testmixin.py
<Instance of Sub, address 7833392:
        name data3=42
        name data2=eggs
        name data1=spam
>
```

Lister对任何继承它的类都适用，无论是什么，因为self引用的是拉入Lister的子类的实例。如果你决定扩展Lister的__repr__的功能，使它可打印出实例所继承的类的属性，直接修改就行了。因为它是一个被继承的方法，改变Lister的__repr__将更新每个混入的下类（注5）。某种意义上讲，混入类是等效于模块的。这里是例子，Lister工作于单一继承模式下，针对不同类的实例。正如我们所说的，面向对象编程的目的就是代码重用：

```
>>> from mytools import Lister
>>> class x(Lister):
...     pass
```

注5：对好奇的读者来说，类也有一个内置属性__bases__，它是类的超类的元组。一个通用的类层次浏览器可以通过一个实例的__class__找到它的类，然后由类的__bases__递归地找到所有的超类。我们将在练习里复习这个概念，但对特殊的对象属性的细节请参看其他书或Python的手册。

```
...
>>> t = x()
>>> t.a = 1; t.b = 2; t.c = 3
>>> t
<Instance of x, address 7797696:
        name b=2
        name a=1
        name c=3
>
```

类是对象：通用的对象工厂

因为类是对象，很容易实现在程序中传递、在数据结构里存放等操作。你也可以把类传给函数，生成任意种类的对象，在面向对象编程设计界里这样的函数称为工厂（*factory*）。在C++这样的强类型语言里，设计这种函数是主要的工作，但对Python来说却是微不足道的事：在第四章里见到的apply函数可以用任何参数调用任何类，只需要一步就可产生任何类的实例（注6）：

```
def factory(aClass, *args):                          # 变参数元组
    return apply(aClass, args)                       # 调用aClass

class Spam:
    def doit(self, message):
        print message

class Person:
    def __init__(self, name, job):
        self.name = name
        self.job  = job

object1 = factory(Spam)                              # 生成一个Spam
object2 = factory(Person, "Guido", "guru")           # 生成一个Person
```

在这段代码里，我们定义了一个名为factory的对象生成函数。它希望把一个类的实例（任何类），以及一个或多个给类的构造函数的参数传给它。它用apply调用类并返回一个实例。代码的其余部分就是定义两个类，并调用factory函数来生成两个实例。而这是在Python里需要写的唯一的factory函数，它对任何类和

注6：事实上，apply可以调用任何可调用对象、类和方法。这个factory函数可以运行任何可调用对象，而不仅仅是类。

任何构造方法都适用。唯一值得注意的改进是：在对构造方法的调用中支持关键字参数，factory可用**args收集关键字参数，并作为apply的第三个参数：

```
def factory(aClass, *args, **kwargs):        # 增加了kwargs字典
    return apply(aClass, args, kwargs)       # 调用aClass
```

现在，你应该知道Python里的一切都是“对象”，甚至类也是，而在C++里类只是编译的输入。然而，Python里你不能从内置的列表和数字这样的对象继承，除非你把它们包在类里。

方法是对象：绑定或未绑定

像函数一样，原来方法也是一种对象。由于既可以通过实例也可以通过类来访问方法，所以在Python里有两种风格：

未绑定的类方法：没有self

通过类来引用方法返回一个未绑定的方法对象。要调用它，你必须显式地提供一个实例作为第一参数。

绑定的实例方法：有self

通过实例访问方法返回一个绑定的方法对象。Python自动地（隐含地）给方法绑定一个实例，所以我们调用它时不用再传一个实例参数。

两种方法都是对象，它们可以被传递、存入列表等等。两者运行时都需要一个实例作第一参数（即一个self值），但当通过一个实例调用一个绑定方法时Python自动会提供一个。例如我们定义如下的类：

```
class Spam:
    def doit(self, message):
        print message
```

现在，我们可生成一个实例，取一个绑定的实例方法但不调用它。object.name是一个对象表达式，这里它返回一个绑定的方法，把实例（object）和方法（spam.doit）绑定，我们可以把它赋给另一个名字，然后像调普通函数一样调用它：

```
object1 = Spam()
x = object1.doit          # 绑定方法对象
x('hello world')          # 实例是隐含的
```

另一方面，如果我们用类去引用doit方法，我们就得到一个未绑定的方法对象。要调用它就得传一个实例参数：

```
t = Spam.doit             # 未绑定的方法对象
t(object1, 'howdy')       # 传递一个实例
```

大多数时候，你都直接调用方法（如：self.attr(args)），所以一般不会注意到方法对象。但如果你开始写通用的调用对象的代码时，你需要特别仔细地注意未绑定方法，它们需要显式地传一个实例参数。

其他内容

私有属性（1.5版新增）

在上一章里，我们提到在一个文件里最上层赋值的名字都由模块输出，这对类也适用，数据隐藏只是一种约定，而客户可以取值或改变任何类和实例的属性。事实上用C++的概念来讲，所有属性都是公共的（public）和虚拟的（virtual），它们在任何地方都可以访问，并在运行时动态地查找。

直到1.5版，Guido才引入了名字压缩（*name mangle*）的概念，这使得类的一些属性得以局部化。私有属性是一种高级的可选特征，对于写大的类层次体系会很有用。

在Python1.5里，class语句里以两个下划线开始的名字（但要求结尾不是两个下划线）自动地变换为类名加该名字。例如，一个类Class中的属性__X自动变换为_Class__X。因为修改后的名字包含了类的名字，它就不会与同一个类层次体系里的其他类的相同属性名冲突。

Python对出现在类里的这样的名字都作变换，例如，名为self.__X的实例属性就变换为self._Class__X，也就是说对实例的属性名也会做名字压缩变换。由于也许有多于一个的类会给实例增加属性，名字压缩有助于自动地避免冲突。

名字变换只发生在class语句里，而且只对你写的有两个前导下划线的名字作变换，这使得代码有点不好读。这也与C++的私有定义不完全一样（如果你知道类名，你仍然可以取到属性值!），但在一个类层次体系里，这可以避免偶然的名字冲突。

文档字符串

到目前为止，我们一直在用#开头的注释来描绘代码。注释有助于别人阅读程序，但当程序运行时却无法看到。Python允许我们把一个文档字符串和一个对象相关联，并提供了一个特别的句法。如果def语句或class语句的第一个句子是一个字符串而不是一个语句，Python就把这个字符串放到__doc__属性中。例如，下面的代码为多个对象定义了字符串：

```
"I am: docstr.__doc__"

class spam:
    "I am: spam.__doc__ or docstr.spam.__doc__"

    def method(self, arg):
        "I am: spam.method.__doc__ or self.method.__doc__"
        pass

def func(args):
    "I am: docstr.func.__doc__"    pass
```

文档字符串的优势在于运行时可以看到。

```
>>> import docstr
>>> docstr.__doc__
'I am: docstr.__doc__'
>>> docstr.spam.__doc__
'I am: spam.__doc__ or docstr.span.__doc__'
>>> docstr.spam.method.__doc__
'I am: spam.method.__doc__ or self.method.__doc__'
>>> docstr.func.__doc__
'I am: docstr.func.__doc__'
```

这在开发期间特别有用。例如，你可以像上面那样在交互式命令行上查看组件的文档，而不需要看源程序里的注释。同样地，Python的对象浏览器可以利用文档字符串来显示对象描述。

然而，文档串并没有被Python程序员广泛使用。文档串在运行时显示，但它不如注释灵活（注释可以出现在程序的任何地方）。两种形式都是有用的，只要文档是准确的，有文档总是好事。

类与模块

最后，我们比较一下这两章的主题——模块和类。由于它们都与名字空间有关，有时容易混淆：

模块

- 将数据和逻辑打包
- 通过写Python文件或C扩展来创建
- 通过导入来使用

类

- 实现新的对象
- 用class语句创建
- 通过类似函数调用的方式使用
- 总是在一个模块里

类还支持模块没有的特征，如操作符重载、多个实例以及继承。尽管都与名字空间有关，但我们希望你现在就能分辨它们是非常不同的。

类的常见问题

大多数类的问题可以归结为名字空间的问题。

改变类属性会有副作用

理论上讲，类（和实例）都是可变对象。就像列表和字典一样可以改变。同样地，改变一个类或实例对象也会影响其他对它的引用。

这也通常是我们期望的，但对改变类的属性要特别注意，因为一个类产生的实例共享一个类的名字空间，类一级的任何变化会影响所有实例。

由于类、模块和实例都是有属性名字空间的对象，你可以在运行时用赋值改变它们的属性。看下面的类，在class内对a的赋值产生了属性X.a，它属于类对象并由所有X的实例继承：

```
>>> class X:
...     a = 1          # 类属性
...
>>> I = X()
>>> I.a               # 被实例继承
1
>>> X.a
1
```

如果我们动态的改变类属性，这会影响到继承这个类的所有对象。而且，新创建的实例得到动态设置的值，而不管类的源代码是怎样写的：

```
>>> X.a = 2           # X变了
>>> I.a               # I也变了
2
>>> J = X()           # J继承了X的运行时的值
>>> J.a               # (但向J.a赋值改变了J中的a，而不是X或I)
2
```

解决办法

这是一个有用的特征还是一个危险的陷阱呢？这取决于你，你甚至没生成一个实例，也可以改变类属性。事实上这个技术可以模拟其他语言中的“记录”或“结构”，例如：

```
class X: pass                          # 生成一些属性名字空间
class Y: pass

X.a = 1                                # 把类属性当作变量
X.b = 2                                # 没有任何实例
X.c = 3
Y.a = X.a + X.b + X.c
```

```
for X.i in range(Y.a): print X.i      # 打印0..5
```

此处类X和Y好像是无文件的模块——用于存储不希望发生名字冲突的变量的名字空间。这是完全合法的Python编程技巧，但应用于别人写的类时就不太合适了。你无法确定你所改变的值对于类的内部行为是否非常关键。如果要模拟一个C的结构（struct），你应该改变实例而不是类，因为这样只影响一个对象：

```
>>> class Record: pass
...
>>> X = Record()
>>> X.name = 'bob'
>>> X.job  = 'Pizza maker'
```

多重继承：顺序问题

这似乎是显然的，但值得强调：如果你使用多重继承，class语句首部里列出的超类次序是重要的。例如在下面的例子里，假设Super也有一个__repr__方法，那我们是要哪一个呢？我们将得到在Sub首部先列出的一个，因为搜索从左至右。但现在假设Super和Lister都有自己的other方法，如果我们想引用Lister的other方法，那么我们必须手工地用Lister.other：

```
class Lister:
    def __repr__(self): ...
    def other(self): ...

class Super:
    def __repr__(self): ...
    def other(self): ...

class Sub(Super, Lister):          # 取Super的__repr__，因为列在前面
    other = Lister.other           # 显式地取Lister的other
    def __init__(self):
        ...
```

解决办法

多重继承是一个高级工具，即使你理解了上面的部分，你最好还是节制并仔细地使用它。否则，一个名字的意义也许依赖于在遥远的子类里的超类的顺序。

类的函数属性是特别的

如果你理解Python的对象模型的话，这一点是很简单的，但它会对有其他面向对象编程语言（特别是Smalltalk）背景的新手造成麻烦。在Python里，类的方法必须带有实例参数。在本章前面我们谈到了未绑定方法：当我们通过类来访问方法时，我们就得到一个未绑定方法。虽然它们是用def定义的，但未绑定方法不是普通的函数，没有实例就无法调用它们。

例如，假设我们想用类属性计算一个类产生了几个实例。记住，类属性由所有该类的实例共享，所以我们可把计数器放在类里面：

```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1
    def printNumInstances():
        print "Number of instances created: ", Spam.numInstances
```

但这无法工作，printNumInstances方法仍然要一个实例参数，因为这是一个与类相关的函数（即使在def首部里没有参数）：

```
>>> from spam import *
>>> a = Spam()
>>> b = Spam()
>>> c = Spam()
>>> Spam.printNumInstances()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: unbound method must be called with class instance 1st argument
```

解决办法

未绑定方法并不是普通函数。如果你想要不用实例就可调用而且能访问类成员的函数，只需把它定义成普通函数，而不是类方法：

```
def printNumInstances():
    print "Number of instances created: ", Spam.numInstances

class Spam:
    numInstances = 0
```

```
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1

>>> import spam
>>> a = spam.Spam()
>>> b = spam.Spam()
>>> c = spam.Spam()
>>> spam.printNumInstances()
Number of instances created:  3
```

当然我们也可以通过实例来调用：

```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1
    def printNumInstances(self):
        print "Number of instances created: ", Spam.numInstances

>>> from spam import Spam
>>> a, b, c = Spam(), Spam(), Spam()
>>> a.printNumInstances()
Number of instances created:  3
>>> b.printNumInstances()
Number of instances created:  3
>>> Spam().printNumInstances()
Number of instances created:  4
```

一些语言理论家说Python没有类方法，只有实例方法。我们觉得他们实际上是指Python类的工作方式与别的语言不同。Python仔细地定义了绑定和未绑定方法的语义，通过类引用你会得到未绑定方法，它是一种特殊的函数。Python确实有类属性，但类的函数需要一个实例参数。

而且，既然Python提供了模块作为名字空间分割工具，通常就没有必要把函数放到类里面，除非它们实现了对象的行为。用普通函数就可以了。例如在本节的例子里，printNumInstances已经与类相关了，因为它们在同一个模块里。

方法、类和嵌套作用域

类与函数一样引入了一个局部作用域，所以同样的作用域问题也发生在类语句体。

而且，当类是嵌套的时候，这是很容易混淆的。比如，下面的例子中，假设generate函数返回一个嵌套的Spam类的实例。在这个代码里，类名Spam是在generate函数的局部作用域里赋值的。但在类的方法里，类名Spam是不可见的，方法只能访问它自己的局部作用域，包含generate的模块，以及内置的名字：

```
def generate():
    class Spam:
        count = 1
        def method(self):          # Spam 不可见
            print Spam.count       # 不是局部的、模块的和内置的
    return Spam()

generate().method()
```

```
C:\python\examples> python nester.py
Traceback (innermost last):
  File "nester.py", line 8, in ?
    generate().method()
  File "nester.py", line 5, in method
    print Spam.count               # 不是局部的、模块的和内置的
NameError: Spam
```

解决办法

最一般的建议是要记住LGB规则，它适用于函数、类和方法。比如，在一个方法内，代码只能访问局部的名字（方法内），全局的名字（模块内），以及内置的。为了访问到类属性，方法需要用self。为了从一个方法内调用另一个方法，也需要用self（self.method()）。

有各种修改上面的代码的方式，最简单的是用global声明Spam，使它进入模块的作用域。由于根据LGB规则，方法可看见模块里的名字，所以是可行的：

```
def generate():
    global Spam                    # Spam进入模块作用域
    class Spam:
        count = 1
        def method(self):
            print Spam.count       # 可行：在全局作用域（模块）
    return Spam()

generate().method()                # 打印1
```

我们可以把Spam定义到模块的最上层，这也许比global声明更好。嵌套的的函数和最上层的generate函数都可以访问到全局作用域的Spam：

```
def generate():
    return Spam()

class Spam:                          # 定义在模块的顶层
    count = 1
    def method(self):
        print Spam.count             # 可行：在全局作用域（模块）

generate().method()
```

我们也可以在方法里去掉对Spam的引用，用特别的__class__属性，它返回实例的类对象：

```
def generate():
    class Spam:
        count = 1
        def method(self):
            print self.__class__.count        # 可行的
    return Spam()

generate().method()
```

最后，我们也可用第四章的可变的缺省参数技巧，但这有些太复杂，前面的方案更好些：

```
def generate():
    class Spam:
        count = 1
        fillin = [None]
        def method(self, klass=fillin):
            print klass[0].count
    Spam.fillin[0] = Spam
    return Spam()

generate().method()
```

注意，我们不能在方法的首部中用klass=Spam，因为在Spam体内，名字Spam同样是不可见的，它不是局部的，不是全局的，也不是内置的。Spam只存在于generate函数的局部作用域，嵌套的类和方法都看不见。

总结

本章讲了Python的两个特别的对象——类和实例，以及创建和处理它们的语言工具。类对象是用class语句创建的，提供缺省的行为，并作为多个实例的生成器。类和实例一起全面地支持了面向对象开发和代码重用。简单地说，类使得我们能实现新的实例对象，并输出数据和行为。

类的主要特征是支持多个实例拷贝，通过继承进行定制，以及操作符重载，而我们在本章探讨了这些特性。由于类主要是与名字空间有关，我们也学习了类对模块和函数名字空间概念的扩展。最后通过一些例子，我们讨论了一些面向对象的设计思想，如合成、授权等。

下一章将结束我们的语言核心之旅，将快速的扫描一下异常处理——一个处理事件的工具。作为对本章的总结，下面是我们谈到的关于类的概念的一览表：

类

一个定义了可继承属性的对象

实例

由类创建的对象，它继承了类的属性，并有自己的名字空间

方法

类对象的函数属性

self

按惯例，这个名字在方法里指实例对象

继承

当一个实例或类访问（通过'.'限定访问）一个类属性时，就要用到继承搜索

超类

一个由其他类继承的类

子类

一个从其他类继承属性的类

练习

这个实验环节要求你写一些类，并用一些已有的代码做试验。为了在练习5中练习集合类，你要么从网上下载源码（见前言），要么手工键入（它很小）。这些程序逐渐变得复杂，所以请参考书后答案中的提示。如果你感到沉闷，最后一个处理合成的练习可能是最有趣的。

1. **基础**。写一个名为Adder的类，输出一个方法add(self, x, y)，它打印一个“未实现”信息。然后定义Adder的两个子类实现add方法：

   — ListAdder，它的add方法返回两个列表参数的连接

   — DictAdder，用add方法返回一个新的字典，包含了两个字典参数的所有成员

   交互式的生成三个类的实例并调用它们的add方法。最后，给你的类加一个构造方法，并在方法里保存一个对象（列表或字典），然后用重载的+操作符取代add方法。最好是加在哪个类里呢？你的类实例可以与什么对象相加呢？

2. **操作符重载**。写一个MyList类包装一个Python列表：要求重载大多数列表操作符——+、[]、迭代、分片等，以及append和sort这样的方法。参见《Python库参考》里的可重载方法列表。也可以为你的类提供一个构造方法，把一个存在的列表（或一个MyList实例）拷贝到实例的成员。交互式的实验你的类，要点是：

   — 为什么这里的对初始值的拷贝很重要？

   — 如果参数是一个MyList实例，你可以用空分片（start[:]）拷贝初始值吗？

   — 有通用的把对列表的访问转到被封装的列表上的办法吗？

   — 你能用MyList加一个普通列表吗？

3. **子类**。现在定义一个练习2的MyList的子类MyListSub。它扩展了MyList，在每个重载操作调用时计算调用的次数，并打印出来。MyListSub应继承MyList的基本行为，例如MyListSub增加一个序列时，应当把+操作的次

数加1，再调用超类的方法，同时要增加一个新方法显示操作计数。你的计数器是属于类还是实例呢？你要如何实现这两种情形呢？（提示：这取决于计数器是赋给哪一个对象的，类的成员由所有实例共享，self的成员是属于实例的。）

4. **元类方法。**写一个Meta类，它有一个方法拦截每一个属性访问（取值和赋值），并打印一条信息。创建一个Meta实例，并作试验。当你在表达式里使用它时会发生什么？试一试+、索引和分片操作。

5. **集合对象。**测试本章的集合类（“扩展内置对象类型”一节）。做如下操作：

   a. 创建两个整数集合，用&和|操作符计算它们的交集合并集。

   b. 创建一个字符串集合，试验集合的索引，它会调用哪个方法呢？

   c. 用一个for循环遍历字符串集合里的成员，这次会调用哪个方法呢？

   d. 试一试计算你的字符串集合与普通字符串的交集和并集，这可行吗？

   e. 用子类扩展你的集合，用一个*args参数使它可以处理任意多的操作数（参见第四章这些算法的函数版本）。用新的子类计算多个参数的交集和并集。&和|都只有两个边，你如何表示三个集合的交集呢？

   f. 如何能在集合类里模拟别的列表操作呢？（提示：__add__可拦截连接操作，而__getattr__可把大多数列表操作传给封装的列表。）

6. **树的链接。**在关于多重继承一节的脚注里，我们提到类有一个__base__属性，它返回一个超类的元组（列在类首部的括号里），用__base__扩充Lister类，让它也打印实例所属类的直接超类。第一行应显示如下：

```
<Instance of Sub(Super, Lister), address 7841200:.
```

   你又应怎样列出类的属性呢？

7. **合成。**定义4个类模拟一个购买快餐的场景：

   — Lunch：一个容器类，也是控制类

   — Customer（顾客）：买食品的演员

   — Employee（雇员）：为顾客服务的演员

   — Food（食物）：顾客买的东西

下面是你需要定义的类和方法：

```
class Lunch:
    def __init__(self)          # 生成嵌入的 Customer 和 Employee
    def order(self, foodName)   # 开始模拟 Customer 的购买
    def result(self)            # 问 Customer 它有什么食品

class Customer:
    def __init__(self)                                  # 把我的食品初始化为 None
    def placeOrder(self, foodName, employee)            # 向一个 Employee 购买
    def printFood(self)                                 # 打印食品的名字

class Employee:
    def takeOrder(self, foodName)     # 返回请求的食品

class Food:
    def __init__(self, name)          # 保存食品的名字
```

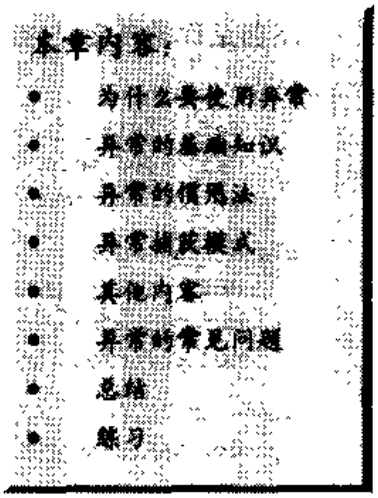
对购买的模拟运行如下：

- Lunch的构造函数应生成并嵌入一个Customer和Employee的实例，并输出一个order方法。当调用时，order方法应通过调用placeOrder方法要求Customer购买。Customer的placeOrder方法应向Employee对象要求一个新的食品对象，这是通过调用takeOrder方法实现的。
- Food对象应保存一种食品的名字字符串（如："burritors"），由Lunch.order到Customer.placeOrder，再到Employee.takeOrder，最后传给Food的构造函数。最上层的Lunch类应输出一个名为result的方法，它要求顾客打印它收到的食品名字。
- 注意，Lunch要么把Employee传给Customer，要么把自己传给Customer，以便让Customer能调用Employee的方法。

8. 交互式的导入并试验练习7中的类，导入Lunch类，调用它的order方法启动一次交互，然后调用它的result方法验证Customer是否得到了它订购的食品。在这个模拟中，Customer是主动的一方，如果由Employee启动交互对话，你的类要怎样修改呢？

第七章

异常



本书第一部分的最后一章将讨论异常(exception) —— 可以改变程序控制流的事件。在 Python 里，异常可以由程序拦截和引发。我们将学习两条新的语句：

`try`

捕获由 Python 或程序本身引发的异常

`raise`

手工地引发一个异常

我们将发现 Python 的异常处理很简单，因为它是语言本身的一部分。

为什么要使用异常

异常使我们能跳过任何大段大段的程序。还记得上一章讲的做比萨饼的机器人吗？假设我们真的做了这样一个机器人。为了做比萨饼，我们的烹饪机器人需要执行一个计划，我们把它写成了一个 Python 程序。它将接受订单，准备生面团，加一些料，烘烤，等等。

假设在烘烤阶段出了严重的问题，也许烤炉破了，也许我们的机器人计算错误，不小心着火了。显然我们希望能快速地跳到处理这种情况的代码（尤其在我们的

机器人快要被烤化的时候!)。既然我们在这种情况没有完成做饼任务的希望，我们也许应该放弃整个计划。

那恰好是异常所做的事情，你可以一步跳到异常处理，跨过所有挂起的函数调用。它是一种“超级 goto”（注 1）。一个异常处理（try 语句）留下了一个记号，并执行了一些代码。在程序往下执行时，引发的异常就会立刻跳到做记号的地方，不管有多少函数正在调用。异常处理的代码可以很恰当地对异常作出反应（呼叫救火部门）。而且，因为 Python 直接跳到处理部分，通常就没有必要检查被调用的可能出错的函数的状态码。

在典型的 Python 程序里，异常可用来做各种事情：

错误处理

当Python检查到程序运行时的错误就引发异常，你可以在程序里捕捉和处理这些错误，或者忽略它们。如果你忽略了，就由 Python 的缺省异常处理负责，它将结束程序并打印出错信息，显示错误在哪里发生。

事件通知

异常也可以作为某个条件的信号，而不需要在程序里传送结果标志或显式地测试它们。例如，一个搜索程序在成功时也许引发一个异常，而不是返回整数 1。

特殊情形处理

有时有些情况是很少发生的，把相应的处理代码改为异常处理更好一些。

奇特的控制流

最后，因为异常是一种高层次的 goto，你可以把它作为实现奇特的控制流的基础。例如，尽管反向跟踪（backtracking）不是 Python 语言的一部分，但可以用异常和一些支持逻辑实现解除赋值（unwind assingment）（注 2）。

注 1：实际上，如果你用过C，你也许有兴趣知道Python的异常与C的一对 setjmp/longjmp 标准函数大致相当。try 语句对应于 setjmp，而 raise 语句对应于 longjmp。但在 Python 里，异常是基于对象的，而且是执行模型的标准部分。

注 2：反向跟踪不是Python语言的一部分，所以我们不会谈太多。如果感兴趣的话，参看关于人工智能或 Prolog 语言、icon 编程语言的书。

我们在本章将看到这些典型用法。首先我们来更仔细地观察Python的异常处理工具。

异常的基础知识

Python的异常是一种高层次的控制流转移机制。它们可以由Python或我们的程序引发，在这两种情况下，都可以被try语句捕获。Python的try语句有两种风格——一种是处理异常（try/except/else），一种是无论是否发生异常都将执行最后的代码（try/finally）。

try/except/else

try是一条复合语句，下面是它的完整形式。它以一个try开始，紧接着是一组缩进语句，然后是一个或多个可选的except子句，它们指明了要捕获的异常的名字，最后是一个可选的else子句：

```
try:
    <语句>          # 运行别的代码
except <名字>:
    <语句>          # 如果在try部分引发了'name'异常
except <名字>, <数据>:
   <语句>           # 如果引发了'name'异常，获得附加的数据
else:
   <语句>           # 如果没有异常发生
```

我们讲讲try语句的工作原理。当开始一个try语句后，Python在当前程序的上下文中作标记，这样当异常出现时就可以回到这里。try子句（try后面的语句块）先执行，接下来会发生什么依赖于执行时是否出现异常：

- 如果当try后的语句执行时发生异常，Python就跳回到try并执行第一个匹配该异常的except子句。异常处理完毕，控制流就通过整个try语句（除非在异常处理时又引发新的异常）。
- 如果在try后的语句里发生了异常，却没有匹配的except子句，异常将被递交到上层的try，或者到程序的最上层（这将结束程序，并打印缺省的出错信息）。

- 如果在try子句执行时没有异常发生，Python将执行else后的语句（如果有else的话），然后控制流通过整个try语句。

换句话说，except子句捕获try子句执行时发生的异常，而else子句只有在执行try子句时没有异常才执行。except是很专注的，它们只捕获相关的try子句出现的异常。

然而，由于try子句可以调用别的函数，所以异常源也许在try语句以外。

try/finally

另一种try语句较为特殊，它必须进行结束操作。如果在一个try里用了finally子句，Python总会执行该子句，无论try子句执行时是否发生了异常：

- 如果没有异常发生，Python运行try子句，然后是finally子句，然后继续。
- 如果在try子句发生了异常，Python就会回来执行finally子句，然后把异常递交给上一层try，控制流不会通过整个try语句。

当你想无论是否发生异常都确保执行某些代码时，try/finally是有用的。finally子句不能与except和else一起用，所以最好把这两种try语句看作不同的语句：

```
try:
    <语句>
finally:
    <语句>          # 退出try时总会执行
raise
```

raise

要引发异常，你需要写raise语句。它的形式很简单：raise后面跟着要引发的异常。你也可以伴随该异常传一个附加的数据（一个对象），只需把它列在异常名的后面。如果传递了附加的数据，可以被try得到，需要写一个接受它的赋值目标：except name,data。

```
raise <name>              # 手工地引发异常
raise <name>, <data>      # 传递一个附加的数据
```

那么什么是异常名呢？它也许是内置作用域内的内置异常（如IndexError），或者是你程序里的任意字符串对象，它也可以是一个类或类实例，我们将推迟到本章后面去讨论这种一般意义的raise语句。异常是用对象来标志的，而任何时候至多有一个是活跃的，一旦被except捕获到，该异常就死掉了（不会再传到别的异常处理），除非用raise重新引发异常。

第一个例子

异常比看起来的样子更简单，我们看一些说明异常基础的简单例子。

缺省的行为：显示错误信息

前面提到，没有被捕获的异常将传到Python的最上层，并运行Python的缺省处理逻辑，Python将终止程序，并打印描述该异常的错误信息，显示异常发生时程序在哪里。例如，运行下面的模块产生了被零除异常，由于程序忽略了它，Python就结束程序并打印信息：

```
% cat bad.py
def gobad(x, y):
    return x / y

def gosouth(x):
    print gobad(x, 0)

gosouth(1)

% python bad.py
Traceback (innermost last):
  File "bad.py", line 7, in ?
    gosouth(1)
  File "bad.py", line 5, in gosouth
    print gobad(x, 0)
  File "bad.py", line 2, in gobad
    return x / y
ZeroDivisionError: integer division or modulo
```

当一个未捕获的异常发生时，Python将结束程序并打印一个堆栈跟踪信息，以及异常名和附加的数据。堆栈跟踪信息显示了异常发生时每个正在调用的函数的信息，如所在文件名、行号和源码，按调用的先后次序。例如，你可以看到被零除异常发生在最后——*bad.py*的第2行，一个return语句。

因为Python的几乎所有的运行错误都是用异常报告的，异常与错误处理的概念是紧密相关的。如果你试验过一些例子，几乎肯定会遇见一两个异常。一般你会得到一个有用的错误提示，帮助你解决问题（注3）。

捕获内置异常

如果你不想在Python的异常发生时结束你的程序，只需要在try里捕获它。例如，下面的代码捕获了当列表索引出界时Python引发的IndexError(索引从0开始)：

```
def kaboom(list, n):
    print list[n]              # 引发 IndexError

try:
    kaboom([0, 1, 2], 3)
except IndexError:             # 这里捕获异常
    print 'Hello world!'
```

当异常在函数kaboom发生时，控制就跳到try的except子句，打印一条信息。由于异常被捕获后就死掉了，程序就通过整个try语句，而不是被终止。实际上，你捕获并忽略了这个错误。

引发并捕获用户定义的异常

Python的程序也可用raise语句引发自定义的异常。较为简单的形式是字符串对象，就像下面的MyError：

```
MyError = "my error"
```

注3：你也可以用Python的标准调试器pdb来隔离错误。与C的调试器dbx和gdb一样，pdb允许你逐行地运行程序，检查变量值，设置断点等等。pdb是Python的标准模块，并且是用Python写的。关于pdb的使用信息请参看Python的库手册或其他书籍。

```
def stuff(file):
    raise MyError

file = open('data', 'r')          # 打开存在的文件
try:
    stuff(file)                   # 引发异常
finally:
file.close()                      # 总是关闭文件
```

用户定义的异常与内置的异常一样是用try捕获的。这里，我们在finally子句里调用了一个关闭文件的函数，以确保无论是否有异常，文件总是关闭的。这个函数并不是太有用（它只是引发异常！），但try/finally是一种很好的形式，它确保结束的动作总是被执行。

异常的惯用法

我们已经学习了异常背后的机制，现在我们来看一些异常的典型用法。

异常并不总是坏事情

Python在出错时引发异常，但不是所有的异常都是错误。例如，我们在第二章“类型和操作符”里见过，文件对象的read方法在文件尾时返回一个空串。Python也提供了一个内置函数raw_input，它从标准输入流读入。与read方法不同，当遇到文件末尾时，raw_input引发内置的EOFError。所以，raw_input常常是这样用的，请看下面的代码：

```
while 1:
    try:
        line = raw_input()        # 从stdin读入行
    except EOFError:
        break                     # 在文件末尾退出循环
    else:
(这里处理下一行)
```

有时用异常传递成功的信号

用户定义的异常也可以标志非错误条件。例如，一个搜索程序在匹配成功时可引

发异常，而不是产生一个必须由调用者解释的状态码。下面的try/except/else代替了对返回值的if/else测试：

```
Found = "Item found"

def searcher():
    引发Found或者返回

try:
    searcher()
except Found:                    # 如果找到了就引发异常
    成功
else:                            # 未找到
失败
```

外部的try语句可以调试代码

你也可以用自己的异常处理替换Python缺省的异常处理。把整个程序封装在一个外部的try里面，你可以捕获运行时的任何异常，也就阻止了程序的结束。下面的代码中，空的except子句捕获了任何未被捕获的异常。为了获得已发生的实际的异常，从sys模块里取exc_type和exc_value属性，它们的值是当前的异常名和数据（注4）：

```
try:
    运行程序
except:                    # 所有未被捕获的异常
    import sys
    print 'uncaught!', sys.exc_type, sys.exc_value
```

异常捕获模式

我们来看一下Python的异常模式的细节。

注4：内置的traceback模块用一种通用的形式处理当前异常，Python 1.5.1中，有一个新的函数sys.exc_info()用于返回一个元组，它包含了当前异常的类型、数据和反向跟踪。sys.exc_type和sys.exc_value仍然有效，但是管理的是一个单个的全局异常，exc_info()跟踪每个线程的异常信息，所以是线程相关的。只有当在程序里使用多线程时这个区别才有意义（这个话题超出了本书的范围）。更多细节见Python库参考。

try 语句的子句

当你写try语句时，try子句后可跟各种子句，表7-1总结了所有可能的形式，我们在前面的例子中基本都见过了——空的except子句将捕获任何异常，finally在结束时执行，等等。可以有任意多的except子句，但finally只能单独出现(没有except或else)，而在一个try里只有一个else子句。

表7-1 try语句子句形式表

子句形式	解释
except:	捕获所有异常
except name:	只捕获特定的异常
except name, value:	捕获异常和它的附加数据
except (name1, name2):	捕获任何列出的异常
else:	如果没有异常
finally:	总是执行

捕获多个异常中的一个

表7-1的第4项是新的，except子句可在括号里列出一组要捕获的异常。当其中任何一个异常发生时，都将被捕获。由于Python是从上到下地查看except子句，括号里列出多个异常与列出单独的异常是一样的，只是更简洁一些。

这里是多个except子句的例子。当调用action函数发生异常时，Python返回到try并搜寻能匹配异常的except，它从上到下从左至右的搜寻，并执行第一个匹配的except子句后的语句。如果没有匹配的，异常将向上递交，else只有当没有异常时才执行。如果你想捕获所有的异常，写一个空的except就可以了：

```
try:
    action()
except NameError:
    ...
except IndexError
    ...
except KeyError:
    ...
```

```
except (AttributeError, TypeError, SyntaxError):
    ...
else:
...
```

运行时嵌套的异常

目前为止，我们的例子只用了单个的try，但当一个try嵌套在另一个try里面时会怎样呢？如果一个try调用一个运行另一个try的函数会怎样呢？如果你知道运行时Python把try语句放在堆栈里，这两种情况都可以理解。当引发异常时，Python回到最近的有except匹配的try语句。由于每个try语句都有记号，Python可以通过检查堆栈里的记号返回到前面的try。

用一个例子可看得更清楚。下面的模块定义了两个函数，action2触发了一个异常（数不能加序列），而action1函数用来捕捉异常。最后的模块一级的代码也有一个try语句。当action2引发TypeError异常时，将有两个活跃的try语句——一个在action1，一个在模块级。Python找最近的能匹配except的一个，在这里是action1内部的try。

```
def action2():
    print 1 + []            # 产生TypeError

def action1():
    try:
        action2()
    except TypeError:       # 最近的匹配的try
        print 'inner try'

try:
    action1()
except TypeError:           # 只有当action1重新引发时
    print 'outer try'

% python nestexc.py
inner try
```

结束时执行finally子句

我们已经讨论过finally子句，但这里是一个更复杂的例子。正如所见到的，

finally子句实际上不捕获特定的异常，当退出一个try语句时，无论是正常退出还是因为异常，都会执行finally子句。所以，finally子句是做清除动作（如关闭文件一类）的好地方。

下面的代码展示了有异常和没有异常的finally。它定义了两个函数，divide和tester：

```
def divide(x, y):
    return x / y                          # 被零除错误?

def tester(y):
    try:
        print divide(8, y)
    finally:
        print 'on the way out...'

print '\nTest 1:'; tester(2)
print '\nTest 2:'; tester(0)               # 引发错误
```

```
% python finally.py

Test 1:
4
on the way out...
Test 2:
on the way out...
Traceback (innermost last):
  File "Finally.py", line 11, in ?
    print 'Test 2:'; tester(0)
  File "Finally.py", line 6, in tester
    print divide(8, y)
  File "Finally.py", line 2, in divide
    return x / y                          # 被零除错误?
ZeroDivisionError: integer division or modulo
```

模块级的代码两次调用tester：

- 第一次调用没有异常，结果打印出来。finally子句执行了。
- 第二次调用产生了异常，控制立即跳到finally，打印信息。Python继续向上传递异常，最后由顶层的缺省异常处理。

其他内容

传递可选的数据

正如所见到的，raise语句可以随异常一起传递一个附加的数据。一般来讲，你可用附加的数据传递上下文信息。事实上，每个异常都有附加数据，很像函数结果，如果没有显式地传送，它就是一个特殊的None对象。看下面的代码：

```
myException = 'Error'                    # 字符串对象

def raiser1():
    raise myException, "hello"           # raise传递附加数据

def raiser2():
    raise myException                    # raise,隐含为None

def tryer(func):
    try:
        func()
    except myException, extraInfo:       # run func, catch exception + data
        print 'got this:', extraInfo

% python
>>> from helloexc import *
>>> tryer(raiser1)                       # 取得附加的数据
got this: hello
>>> tryer(raiser2)                       # 取得了None
got this: None
```

assert语句

Python1.5增加了一个assert语句，它似乎是raise语句的缩写，格式如下：

```
assert <test>, <data>                    #   <data>部分是可选的
```

等同于下面的代码：

```
if __debug__:
    if not <test>:
        raise AssertionError, <data>
```

请留意：偷懒的程序

一个评价异常是否有用的方式，是比较Python和无异常语言的编码方式。例如，如果你想用C语言写机器人程序，你不得不在每个可能出错的操作后检查返回值或状态码：

```
doStuff()
{                                          // C程序：
     if (doFirstThing() == ERROR)          // 必须到处检查错误
          return ERROR;                    // 即使不在这里处理
     if (doNextThing() == ERROR)
          return ERROR;
     ...
     return doLastThing();
}

main()
{
     if (doStuff() == ERROR)
          badEnding();
     else
          goodEnding();
}
```

事实上，实际的C程序中检查错误的代码与做实际工作的代码一样多。但在Python里，你不必这样小心，你可以把异常处理部分独立出来，而在写作实际工作的代码时假设一切正常：

```
def doStuff():
     doFirstThing()       # we don't care about exceptions here
     doNextThing()        # so we don't need to detect them here
     ...
     doLastThing()

if __name__ == '__main__':
     try:
          doStuff()       # this is where we care about the result
     except:              # so it's the only place we need to check
          badEnding()
     else:
          goodEnding()
```

因为异常发生时，控制立即自动地跳到异常处理，不需要让你的所有代码都提防错误。结果是异常使你忽略掉不常见的情况，并避免了很多错误检测代码。

但如果用了-O命令行标志，assert将会从编译的字节码中移去，这样就优化了程序。AssertionError是一个内置的异常，而__debug__标志是内置的名字，如果不用-O则它的值为1。assert常用于开发时验证程序的条件，当显示时，它会显示源码行的信息。

类异常

最近，Python扩展了异常的概念，它也可以是一个类或类实例。与模块包和私有属性一样，类异常是一个高级课题，你可以选择用还是不用。如果你刚起步，你可以把这一节看作选读内容。

目前我们是用字符串来标志我们的异常，当异常引发时，Python对异常和except子句的匹配是以对象的身份（identity）比较（如用第11章的is测试）为基础的。但当引发一个异常类时，只要except子句中是该类或它的超类就算匹配上了。结果是类异常支持建立异常层次体系：命名一个通用的异常的超类，一条except子句可以捕获整类的异常。

一般来说，用户定义的异常可以是字符串或类对象。从Python1.5开始，所有的内置异常都定义为类，而不是字符串。你通常不必在意，除非你以为内置的异常是字符串而想不转换就做连接操作。

raise的一般形式

增加了基于类的异常后，raise语句可以有下列5种形式：前两个是字符串异常，后两个是类异常，最后一个是Python1.5新增的，它重新引发当前的异常（当你需要向上传递已捕获的异常时要用到它）。引发一个实例实际上就是引发一个实例的类，实例是伴随类的附加数据。

```
raise string                # 匹配的except有同样的字符串
raise string, data          # 可选的附加数据data(缺省=None)

raise class, instance       # 匹配的except是这个类或它的超类
raise instance              # 等同于: raise instance.__class__, instance

raise                       # 重新引发当前的异常(1.5新增)
```

为了与用字符串异常的老版本兼容，你也可以用下面的形式：

```
raise class                    # 等同于：raise class()
raise class, arg               # 实际上都是：raise instance
raise class, (arg, arg,...)
```

这些形式的意思都是raise class(arg…)，也就是说与上面的raise instance是一样的（Python调用class类创建一个实例，并引发它）。例如你可以用raise KeyError来引发一个内置的KeyError的实例，KeyError是一个类。

如果这听起来有些混淆，只要记住异常可以是字符串、类或类实例，你可以与异常一起传递一个附加数据。如果你传递的附加数据不是一个实例，Python将为你生成一个实例。

例子

我们通过一个例子来了解类异常的原理，定义一个超类General和一个子类Specific。我们试图说明异常分类的概念，能捕获General也就能捕获子类Specific：

```
class General:          pass
class Specific(General): pass
def raiser1():
    X = General()       # 引发列出的类实例
    raise X

def raiser2():
    X = Specific()      # 引发子类的实例
    raise X

for func in (raiser1, raiser2):
    try:
        func()
    except General:     # 匹配General 或任何它的子类
        import sys
        print 'caught:', sys.exc_type
```

```
% python classexc.py
caught: <class General at 881ee0>
caught: <class Specific at 881100>
```

由于这里只有两种异常，所以还不足以显示类异常的好处。我们同样可以在except子句里列出一组字符串异常（如except(a,b,c)），但对于大的异常体系来说，捕获类异常要比列出所有该分类的成员要简单些。而且，可以通过增加子类来扩展异常体系，而不影响存在的代码。

例如，内置异常ArithmeticError是超类，OverflowError和ZeroDivisionError是子类，而在try里捕获ArithmeticError时也可以捕获它的子类。

除了支持异常体系外，类异常也为附加的状态信息提供了存储（作为实例的属性），但这不比用字符串异常更方便。在Python里是否使用面向对象编程可以由你自己选择。

异常的常见问题

这里是关于异常使用的一般性提示。

捕获太多

因为Python让你选择捕获哪一个异常，有时要注意不要包括太多。例如，一个空的except子句就捕获了所有的异常。有时这是期望的，但你也可能拦截了上一级的异常。下面的代码捕获了所有的异常，而不管是否有别的异常处理在等它：

```
try:
    ...
except:
    ...                              # 所有的异常
```

这里的问题是，你也许不能预料到一个操作中发生的所有异常：

```
try:
    x = mydictionary[spam]           # 拼写错误：dict误写为ditc
except:
x = None                             # 假设是KeyError或IndexError
```

解决办法

这种情况下，你假设当索引字典时发生的唯一错误是索引错。但因为你的名字拼写错了，Python 引发了 NameError，它将被捕获，但被忽略掉了。你应该写：except(KeyError,IndexError)，这样你的意图更清楚。

捕获太少

反过来，你有时不要排除太多。在 try 里列出了特定的异常后，就只能捕获列出的异常了。这没有什么错，但当你的系统将来扩展后有了新的异常，你就要回头来修改你的代码。例如，下面的代码中，你必须更新你的异常列表，否则将会漏掉新的异常：

```
try:
  ...
except (myerror1, myerror2):    # 如果要增加myerror3怎么办?
    ...                         # 非错误
else:
...
```

解决办法

仔细地用类异常可以完全消除这个常见问题。你可以捕获一个一般的超类，然后在将来可以增加子类，而不必修改 except 子句。

无论你是否用类，有一些设计总是好的。你必须注意异常处理不要太一般或太特殊。特别是在大的系统里，异常处理应该是整体设计的一部分。

总结

在本章里我们学习了异常——怎样用 try 来捕获它，以及怎样用 raise 来引发它。异常由字符串或类来标识。在 Python1.5 中，内置的异常预定义为类，但用户定义的异常可以是字符串或类。异常使得我们可以在程序里任意跳转，提供了一种一致的方式处理错误或别的事件。我们还学了常见的异常惯用法，错误处理，各种捕获和匹配异常的方式。

这一章结束了我们的Python语言核心之旅。如果你已到达这里，你可以把自己看作一个正式的Python程序员了。在这一部分里，我们学了内置类型、语句和异常，以及写大程序的工具——函数、模块和类。一般来说，Python提供了一个工具体系：

内置的

内置类型如字符串、列表和字典，使我们很容易快速地写出简单的程序。

Python扩展

对要求更高的任务，我们通过写函数、模块和类来扩展Python。

C扩展

尽管本书没有讲到，但可以用C或C++写的模块来扩展Python。

因为Python把它的工具集分了层，我们可以根据任务的情况选择工具的复杂程度。我们在本书介绍了前两类，这已足以用Python来编写真正的程序了。

本书的下一部分带你进入Python的标准模块和常见任务。表7-2总结了Python程序员可用的工具，以及我们在后半部分将讨论的课题。到目前为止，我们的大多数例子都很小。我们是有意这样写的，目的是帮助你掌握基础知识。但既然你已经学习了语言的核心，现在是开始学习如何使用它们做实际工作的时候了。我们将发现，用Python这样简单的语言，常见的任务将比你所想像的要简单得多。

表7-2 Python的内置工具箱

类别	例子
对象类型	列表、字典、文件、字符串
函数	`range, apply, open`
模块	`string, os, Tkinter, pickle`
异常	`IndexError, KeyError`
属性	`__dict__, __name__`
外围的工具	`NumPy, SWIG, JPython, PythonWin`

练习

我们将做几个简单的异常练习。异常实际上是一个简单的工具，如果你完成了这些练习，你也就掌握了异常。

1. **try/except**。写一个名为oops的函数，它引发一个IndexError。然后写一个函数调用oops，并在一个try/except语句里使用以捕获这个错误。如果你把oops改为引发KeyError将会发生什么呢？IndexError和KeyError来自哪里呢？（提示：LGB规则。）

2. **异常列表**。修改oops函数，让它引发你自定义的异常MyError，并传递一个附加数据。然后扩展try语句，增加捕获这个异常和它的附加数据，并打印附加数据。

3. **错误处理**。写一个名为safe(func, *args)的函数，它用apply运行任何函数，捕获任何异常，并用sys模块的exc_type和exc_value属性打印异常。然后用safe函数运行你写的oops函数。把safe放在*tools.py*文件里，并交互式地把oops函数传给它。你得到什么样的错误信息？最后扩展safe，通过调用traceback模块里的函数print_exc()，打印发生异常时的堆栈跟踪信息。（参见《Python库参考》或其他Python书籍以获得更多信息。）

第二部分

外围层

第一部分中，我们讲述了Python语言的核心。有了这些基础知识，你应该可以阅读大多数Python代码了，不会有语言方面的问题。但是，任何看过计算机程序的人都知道，仅仅理解语言的语法，并不能保证能清楚而简单地理解程序，即使程序写得很好。实际上，了解所用的工具，如简单函数、复杂的框架，是从理论到实践的重要一步。

怎样才能完成这种转换呢？仅仅阅读木工杂志，是无法使一个门外汉变成一名木工大匠的。天赋当然是不可缺少的，但是要想成为大师，必须花多年时间观察、拆装、制作各种家具，从自己的错误和别人的成功中学习。编程也是一样。教科书的作用是对各种问题和解决方案进行概括性的阐述，给出一些基本技巧，并通过展示别人的优秀作品最终激励初学者努力学习。本部分的每一章都将讲述Python某一方面的卓越性能，并提供大量的信息资源。

本章内容：

- 内置函数
- 库模块
- 练习

第八章

内置工具

本章将介绍Python标准库的一部分重要工具——内置函数、库模块以及它们中最有用的函数和类。即使你很可能在任何一个程序里都不会用到所有这一切，但我们见过的有用程序里没有能少了它们的。正因为在程序中常出现对序列的操作，所以Python提供了列表这个数据类型，而库里也提供了一组常能派用场的模块。在开始设计和写任何有用的代码前，先看看是否有类似的模块存在。如果是Python的标准库的一部分，你就可以确信它已经过大量测试；更妙的是，会有其他人会尽力解决其中存在的任何问题——这都是免费的。

注意本章只是对标准库的最好部分作了一个简短的扫描。在写本书时，《Python库参考》已经超过200页。更多的参考细节可看附录一“Python资源”，你应该知道它是本书的理想伴侣，它提供了这里无法容纳的完整性，而且可以在网上得到最新的标准Python工具集的描述。也可参考O'Reilly出版的《Python Pocket Reference》，作者是本书作者之一Mark，该书包括了标准库的重要模块以及语法和内置函数。

本章包括对两种工具的描述——内置函数和标准模块。但在我们深入讨论以前将简短地谈论内置对象。例如在介绍列表时，我们介绍了列表最重要的方法(append, insert等)，我们没有介绍所有的一切，以便我们能集中精力于它最重要的方面。如果你对我们忽略的部分好奇，可以查阅库参考，或者玩一玩交互式

的解释器。从1.5版开始，dir这个内置函数返回对象重要属性的列表，它同type函数一起为了解要操作的对象提供了极佳的方式。例如：

```
>>> dir([])                                    # 列表的属性是什么?
['append', 'count', 'index', 'insert', 'remove', 'reverse', 'sort']
>>> dir(())                                    # 元组的属性是什么?
[]                                             # 元组没有属性!
>>> dir(sys.stdin)                             # 文件的属性是什么?
['close', 'closed', 'fileno', 'flush', 'isatty', 'mode', 'name', 'read',
'readinto', 'readline', 'readlines', 'seek', 'softspace', 'tell', 'truncate';
'write', 'writelines']
>>> dir(sys)                                   # 模块也是对象
['__doc__', '__name__', 'argv', 'builtin_module_names', 'copyright',
'dllhandle' 'exc_info', 'exc_type', 'exec_prefix', 'executable', 'exit',
'getrefcount','maxint', 'modules', 'path', 'platform', 'prefix', 'ps1',
'ps2','setcheckinterval', 'setprofile', 'settrace', 'stderr', 'stdin',
'stdout','version', 'winver']
>>> type(sys.version)                          # 'version'是什么类型?
<type 'string'>
>>> print sys.version                          # 字符串的值是什么?
1.5 (#0, Dec 30 1997, 23:24:20) [MSC 32 bit (Intel)]
```

sys模块

sys中有几个Python的内部函数和属性，sys在这里的意思是Python系统，而不是指操作系统。一些有用的属性：

sys.path

导入模块时，Python要查找的目录路径的列表。

sys.modules

包含当前已装入模块的字典。

sys.platform

当前平台的名字字符串。可能的值包括'win32'、'mac'、'osf1'、'linux-i386'、'sunos4'等。这在做与平台相关的事情时会有用（比如启动一个窗口管理器）。

sys.ps1和sys.ps2

两个可打印对象，分别是Python的交互式解释器的主、次提示符。它们的缺

省值是>>>和...。你可以把它们设为字符串，或者是定义了__repr__方法的类实例。

sys模块的最常用属性是sys.argv，它是命令行输入的单词列表。如果输入包含Python的话，sys.argv中不包含Python本身的引用。换句话说：如果你在操作系统的shell上键入：

```
csh> python run.py a x=3 foo
```

那么当run.py启动时，sys.argv的值是['run.py', 'a', 'x=3', 'foo']。sys.argv是可修改的（毕竟它只是一个列表）。常见的用法是遍历这些参数，写作sys.argv[1:]，即从1到结尾的所有参数，但不包括存放在sys.argv[0]的程序（模块）名。

最后，sys模块有三个文件属性：sys.stdin、sys.stdout和sys.stderr。它们分别是标准输入流、标准输出流和标准错误流。标准输入流一般与键盘相关联，标准输出和标准错误通常与屏幕相连，这由操作系统控制。Python的print语句输出到标准输出，而错误信息如异常就输出到标准错误流。我们可以看到它们都是可以修改的：你可以把一个Python程序的输出重定向到一个文件，这只需要一个赋值：

```
sys.stdout = open('log.out', 'W')
```

内置函数

dir函数是一个内置函数：它位于内置名字空间（namespace）里。应用LGB规则，也就意味着函数总是可以使用的，而且不需要import语句（注1）。你已经

注1：这也意味着，如果你定义了一个局部的或模块范围内全局的同名引用，则随后的dir将变成你的新变量而不是内置函数。这也是一些微妙的错误产生的原因。我们最近写了一个程序，程序用了一个变量o以及一个这种变量的列表，名为os（o的复数形式）。奇怪得很，原来好好的代码由于用了os.system而开始抱怨AttributeErrors！另一个常见的同类错误是type = type(myObject)，这只在第一次有效。第二次Python就会引用这个局部变量type，肯定会失败。

见过许多内置函数了，比如len、open、type、list、map等。你可以在__builtins__名字空间里找到它们和标准异常。

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'EOFError',
'Ellipsis', 'Exception', 'FloatingPointError', 'IOError', 'ImportError',
'IndexError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError',
'NameError', 'None', 'OverflowError', 'RuntimeError', 'StandardError',
'SyntaxError', 'SystemError', 'SystemExit', 'TypeError', 'ValueError',
'ZeroDivisionError', '__debug__', '__doc__', '__import__', '__name__', 'abs',
'apply', 'callable', 'chr', 'cmp', 'coerce', 'compile', 'complex', 'delattr',
'dir', 'divmod', 'eval', 'execfile', 'filter', 'float', 'getattr', 'globals',
'hasattr', 'hash', 'hex', 'id', 'input', 'int', 'intern', 'isinstance',
'issubclass', 'len', 'list', 'locals', 'long', 'map', 'max', 'min', 'oct',
'open', 'ord', 'pow', 'range', 'raw_input', 'reduce', 'reload', 'repr',
'round', 'setattr', 'slice', 'str', 'tuple', 'type' 'vars', 'xrange']
```

转换、数字和比较

一些函数用于转换对象类型。我们已见过str,它任何时候都返回一个字符串，而list和tuple函数靶序列分别转换为列表和元组。int、complex、float和long则把数字分别转换为相应类型。hex和oct把整数转换为16进制和8进制。

int、long和float有一些容易混淆的特征。首先，如果操作时需要的话，int和long会切掉数字参数的一部分，所以会丢失信息，也许不是你所想要的转换。其次，如果参数字符串是合法数字的话，int、long和float将把字符串转换为相应的类型（注2）：

```
>>> int(1.0), int(1.4), int(1.9), round (1.9), int (round (1.9))
(1, 1, 1, 2.0, 2)
>>> int("1")
1
>>> int("1.2")                                  # 这行不通
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ValueError: invalid literal for int(): 1.2
```

注2：数字串在编译时转化为对应的数值。所以“1234”是一个有效的整数串，而“def foo():”却不是。

```
>>> int("1.0")                                    # 这也不行
Traceback (innermost last):                       # 因为1.0也不是合法的整数
  File "<stdin>", line 1, in ?
ValueError: invalid literal for int(): 1.0
>>> hex(1000), oct(1000), complex(1000), long(1000)
('0x3e8', '01750', (1000+0j), 1000L)
```

以int为例，用一个定制的转换函数也许是有意义的，只作转换而拒绝切断原数字：

```
>>> def safeint(candidate):
...     import math
...     truncated = math.floor(float(candidate))
...     rounded = round(float(candidate))
...     if truncated == rounded:
...             return int(truncated)
...     else:
...             raise ValueError, "argument would lose precision when cast to
integer"
...
>>> safeint(3.0)
3
>>> safeint("3.0")
3
>>> safeint(3.1)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 6, in safeint
ValueError: argument would lose precision when cast to integer
```

abs函数返回数字（整数、长整数、浮点数）的绝对值，或者返回复数的模（复数的实部和虚部平方和的平方根）：

```
>>> abs(-1), abs(-1.2), abs(-3+4j)
(1, 1.2, 5.0)                                     # 5是(3*3 + 4*4)的平方根
```

ord和chr函数在ASCII值和字符间相互转换：

```
>>> map(ord, "test")                 # 字符串是字符的序列，所以可以用map
[116, 101, 115, 116]
>>> chr(64)
'@'
>>> ord('@')
```

```
64
# map返回单个字符的列表，所以需要连接成一个字符串
>>> map(chr, (83, 112, 97, 109, 33))
['S', 'p', 'a', 'm', '!']
>>> import string
>>> string.join(map(chr, (83, 112, 97, 109, 33)), '')
'Spam!'
```

cmp函数用于比较运算，当第一参数大于、等于或小于第二参数时，分别返回正整数、零或负整数。值得补充的是，cmp不仅可比较数字，而且可比较字符的ASCII值，以及序列的成员值。比较可能引发异常，所以cmp不保证对所有对象都适用，但所有合理的比较都是可行的。cmp与列表的sort排序方法用的是同一个比较过程。min和max也是一样的，分别返回对象序列中最小和最大的成员：

```
>>> min("pif", "paf", "pof")          # 当用多个参数调用时
'paf' arguments                       # 返回其中合适的一个
>>> min("ZELDA!"), max("ZELDA!")      # 当用一个序列调用时
'!' 'Z'                               # 返回最小/最大的成员
```

表8-1总结了进行转换的内置函数。

表8-1 内置转换函数

函数名	行为

str(*string*)	返回任何对象的字符串表示：
	`>>> str(dir())`
	`"['__builtins__', '__doc__', '__name__']"`

list(*seq*)	返回一个序列的对应列表：
	`>>> list("tomato")`
	`['t', 'o', 'm', 'a', 't', 'o']`

tuple(*seq*)	返回一个序列对应的元组：
	`>>> tuple("tomato")`
	`('t', 'o', 'm', 'a', 't', 'o')`
	`>>> tuple([0])`
	`(0,)`

int(*x*)	把一个字符串或数字转换为整数，去掉浮点部分：
	`>>> int("3")`
	`3`

表8-1 内置转换函数（续）

函数名	行为
long(*x*)	把一个字符串或数字转换为长整数，去掉浮点部分： `>>> long("3")` 3L
float(*x*)	把一个字符串或数字转换为浮点数： `>>> float("3")` 3.0
Complex(*real*,*imag*)	创建一个值为 real + imag*j 的复数： `>>> complex(3,5)` (3+5j)
hex(*i*)	把一个字符串或数字转换为16进制数： `>>> hex(10000)` 'ox2710'
oct(*i*)	把一个字符串或数字转换为8进制数： `>>> oct(10000)` '023420'
ord(*c*)	返回一个单字符的ASCII值： `>>> ord('A')` 65
chr(*i*)	返回ASCII值为 *i* 的单个字符： `>>> chr(65)` 'A'

min(*i* [, *i*]*)	返回非空序列的最小成员：
	`>>> min([5,1,2,3,4])`
	1
	`>>> min(5,1,2,3,4)`
	1

max(*i* [, *i*]*)	返回非空序列的最大成员：
	`>>> max([5,1,2,3,4])`
	5
	`>>> max(5,1,2,3,4)`
	5

属性操作函数

四个内置的函数hasattr、getattr、setattr和delattr分别对一个名字空间的属性作如下操作：是否存在、取值、设置值和删除。它们在操作之前没有名字的对象和属性很有用，可用于模块、类和实例，表8-2对此做了总结。

表8-2 操作对象属性的内置函数

函数名	行为
hasattr(*object*, *attributename*)	如果 *object* 有属性 *attributename*，则返回1，否则返回0
getattr(*object*, *attributename* [, *default*])	返回 *object* 的 *attributename* 属性；如果属性不存在，若指定了缺省值 *default* 就返回 *default*，否则引发 AttributeError 异常
setattr(*object*, *attributename*, *value*)	把 *object* 的属性 *attibutename* 赋值为 *value*，如果不支持创建和修改属性就引发异常 *TypeError*
delattr(objeot, attributename)	删除 *object* 的 *attributename* 属性，如果不存在就引发异常 AttributeError

我们在第六章“类”里曾看到它们的用法，现在看一个简单的例子，在一个名字空间里（这里是一个类对象）创建一个属性，如果已存在就加1：

```
>>> def increment_attribute(object, attrname):
...     if not hasattr(object, attrname):
...         setattr(object, attrname, 1)
...     else:
...         setattr(object, attrname, getattr(object, attrname) + 1)
...
>>> class Test: pass
...
>>> aname = 'foo'
>>> increment_attribute(Test, aname)        # 创建Test.foo并设为1
>>> increment_attribute(Test, aname)        # Test.foo加1
>>> Test.foo
2
```

在Python 1.5.2中，为getattr增加了一个可选的第三参数，用于在对象没有指定的情况下指定使用值。这样，上面的代码可以简化成：

```
def increment_attribute(object, attrname):
    setattr(object, attrrname, getattr(object, attrname, 0) + 1)
```

执行程序

本节里最后一组内置函数是关于创建、操作和调用Python代码的。请参见表8-3的总结。

表8-3 执行Python代码的内置函数

函数名	行为
import	执行模块里的代码作为导入模块操作的一部分，并返回模块对象
exec *code* [in *globaldict* [, *localdict*]]	在可选的全局和局部名字空间里执行指定的代码
compile(*string*, *filename*, *kind*)	把*string*编译成为一个代码对象
execfile(*filename* [, *globaldict* [, *localdict*]])	在可选的全局和局部名字空间里执行指定文件里的代码
eval(*code*[, *globaldict* [, *localdict*]])	在可选的全局和局部名字空间里计算指定的表达式（字符串或编译好的代码对象）

写一个运行其他程序的程序是简单的。我们将讨论在Python程序里调用任何程序的方式。我们已见过import语句，它可以执行在Python的路径里存在的文件里的代码。另外还有几种执行任意的Python代码的机制。第一种是用exec，它是一个语句而不是函数，它的语法是：

```
exec code [ in globaldict [, localdict]]
```

exec可由1到3个参数。第一个参数必须是Python的代码——一个字符串，或者在一个打开的文件里，或者是一个编译过的代码对象（后面详述）。例如：

```
>>> code = "x = 'Something'"
>>> x = "Nothing"                    # 给x赋值
>>> exec code                        # 修改x的值!
>>> print x
'Something'
```

exec的参数是可选的。如果是一个参数，它就用作执行代码的全局和局部名字空间。如果是两个字典参数，它们就分别用作全局和局部名字空间。如果像上例一样省略掉，就使用当前的全局和局部名字空间。

注意：当执行exec时，Python需要解析要执行的代码。这可能会消耗很大，特别是要执行上千次的大段代码。如果是这种情况，就值得先一次性的编译好，再重复地执行它。compile函数就是把一个串变成一个编译好的代码对象，然后可用exec语句高效地执行。

compile有三个参数。第一个是代码字符串。第二个是对应的Python源程序的文件名（如果不是从一个文件读入的就是'*<string>*'），它用于在执行这个代码并产生异常时的堆栈跟踪。第三个参数是'single'，'exec'，'eval'三者之一，这取决于代码是一条立即打印结果的语句，还是一组语句，还是一个表达式（为eval函数编译）。

一个与exec语句相关的函数是execfile，它与exec相似，但它的第一个参数必须是一个Python脚本的文件名，而不是一个文件对象（文件对象是由open函数返回的）。那么，如果想要你的Python脚本把它的参数当脚本执行，可以这样写：

```
import sys
for argument in sys.argv[1:]:        # 我们将跳过我们自己，否则会循环!
    execfile(argument)               # 做任何事
```

还有另两个函数可运行Python代码。第一个是eval函数，它以一个代码串或者是一个编译过的代码对象为参数，并执行这个表达式，例如：

```
>>> z = eval("'xo'*10")
>>> print z
'xoxoxoxoxoxoxoxoxoxo'
```

eval函数不能用于执行语句，因为语句和表达式的句法是不同的，例如：

```
>>> z = eval("x = 3")
Traceback (innermost last):
  File "<stdin>", line 1, in ?
```

```
  File "<string>", line 1
    x = 3
      ^
SyntaxError: invalid syntax
```

最后一个执行代码的函数是apply。它以一个可调用对象，以及一个可选的元组位置参数，一个可选的关键字字典作为参数。可调用对象可以是任何函数（普通函数、方法等），或者任何定义了__call__方法的类或实例。如果你拿不准，可以用callable函数来测试（注3）：

```
>>> callable(sys.exit), type(sys.exit)
(1, <type 'builtin_function_or_method'>)
>>> callable(sys.version), type(sys.version)
(0, <type 'string'>)
```

还有一些别的函数没有谈到，如果你感到好奇的话，可查阅库参考（2.3节）。

库模块

现在在Python标准版本里有超过200个模块，包括文本字符串处理、网络和Web工具、数据库接口、对象串行化、算法和数据结构、用户界面、数字计算等等。我们在这里只介绍最广泛使用的，在第九章和第十章里还将介绍一些更强大和更特殊的模块。

基本字符串操作：string模块

string模块的存在有一些历史原因。如果是现在来设计Python的话，string模块里的许多函数很可能会设计为string对象的方法（注4）。string模块的作用是操作字符串。表8-4列出了最有用的函数及其简短描述。这个表是不完整的，完

注3：查阅Python参考，你会发现很多关于可调用对象的信息，比如它们需要多少个参数，参数名以及参数的缺省值，特别是3.2节。

注4：关于这个问题的细节以及其他许多的常见问题，请参考FAQ列表http://www.python.org/doc/FAQ.html。关于字符串方法的问题参看该文档中的6.4节。

整的描述参见库参考或《Python Pocket Reference》。除了特别注明以外，每个函数都返回一个字符串。

表 8-4 string 模块的函数

函数名	行为
atof(*string*)	把一个字符串转换为浮点数： `>>> string.atof("1.4")` 1.4
atoi(*string* [, *base*])	以指定的基（缺省为 10）把字符串转换为整数： `>>> string.atoi("365")` 365
atol(*string* [, *base*])	与 atoi 类似，只是转换为长整数： `>>> string.atol("987654321")` 987654321L
capitalize(*word*)	把单词的首字母转换为大写： `>>> string.capitalize("tomato")` 'Tomato'
capwords(*string*)	把 string 里的每个单词首字母都转换为大写： `>>> string.capwords("now is the time")` 'Now Is The Time'
expandtabs(*string, tabsize*)	用指定的 tab 字符的大小扩展 *string* 里的 tab 字符
find(*s, sub* [, *start* [, *end*]])	返回子串 *sub* 在字符串 *s* 里首次出现的位置索引，如果没找到就返回 -1： `>>> string.find("now is the time", 'is')` 4
rfind(*s, sub* [, *start* [, *end*]])	与 find 相似，只是返回的是 *s* 中子串 *sub* 的最后一个出现的索引
index(*s, sub* [, *start* [, *end*]])	与 find 类似，只是当没找到时引发异常 ValueError
rindex(*s, sub* [, *start* [, *end*]])	与 rfind 类似，只是当没找到时引发异常 ValueError

表 8-4 string 模块的函数（续）

函数名	行为

count(*s*, *sub* [, *start* [, *end*]])	返回 *sub* 在 *s* 里出现的次数：
	`>>> string.count("now is the time", 'i')`
	`2`

replace(*str*, *old*, *new*[, *maxsplit*])	在 str 里把 old 替换为 new：
	`>>> string.replace("now is the time", '', '_')`
	`'now_is_the_time'`

lower(*string*), upper(*string*)	分别返回 *string* 的小写和大写

split(s [, *sep* [, *maxsplit*]])	在指定的分隔符位置把string分开（缺省分隔符为空格），并返回分开的子串的列表：
	`>>> string.split("now is the time")`
	`['now', 'is', 'the', 'time']`

join(*wordlist* [, *sep* [, *maxsplit*]])	连接一个字符串的序列，中间插入分隔符（缺省是一个空格）：
	`>>> string.join(["now", "is", "the", "time", '*')`
	`'now*is*the*time'`
	`>>> string.join("now is the time", '*')`
	`'n*o*w* *i*s* *t*h*e* *t*i*m*e'`
	记住字符串本身是一个单字符的序列

lstrip(*s*), rstrip(*s*), strip(*s*)	分别去掉 *s* 左边、右边和两边的空白符：
	`>>> string.strip(" before and after ")`
	`'before and after'`

swapcase(*s*)	把 *s* 里的字符大小写对换
ljust(*s*, *width*), rjust(*s*, *width*), center(*s*, *width*)	分别对 *s* 作左对齐、右对齐和居中，返回的字符串宽度为 *width*，不足的部分填上空格

string 模块里也定义了一些有用的常量，见表 8-5。

表 8-5 string 模块里的常量

常量名	值
digits	'0123456789'
octdigits	'01234567'
hexdigits	'0123456789abcdefABCDEF'
lowercase	'abcdefghijklmnopqrstuvwxyz'
uppercase	'ABCDEFGHIJKLMNOPQRSTUVWXYZ'[注]
letters	小写加上大写字母
whitespace	' \t\n\r\v'（所有的空白字符）

注：大多数系统上，string.lowercase, string.uppercase和string.letters有上述值。如果使用locale模块指定不同的文化背景，它们将被更新。例如，执行locale.setlocale (locale.LC_ALL, 'fr')后，string.letters属性也包含带重音符的字母和其他合法法语字母。

表 8-5 中的常量用于测试字符 —— 例如，x in string.whitespace 只有当 x 是一个空白符时返回真。

string模块的典型用法是简化用户输入。下面的例子是去掉所有的“多余的”空格符，用一个单空白符代替，并删除首尾空白。

```
thestring = string.strip(string.join(string.split(thestring)))
```

高级的字符串操作：re 模块

string模块定义了基本的字符串操作。它在几乎所有与文件或用户交互的程序里都会出现。因为Python的字符串里可包含null字符（0），所以也可以处理二进制数据 —— 讨论 struct 模块时再详述。

而且，Python还提供了一个特别的使用正则表达式的字符串处理工具。很久以来，Python的正则表达式（regex和regsub模块）对一些任务是够用的，但与一些竞争性语言如Perl相比还有差距。到了Python1.5时，提供了一个全新的正则表达式模块 re，这明显地提高了 Python 的字符串处理能力。

正则表达式

正则表达式是一种字符串，用于定义复杂的字符串匹配模式和替换规则，它由一些紧凑的助记符号组成。例如字符“.”的意思是“匹配任何单个字符”。“+”的意思是“一个或多个在我之前的字符”。表8-6列出了常用的正则表达式符号及意义。

表8-6 正则表达式常用符号的含义

特殊字符	意义
.	匹配除换行符外的任何字符
^	匹配字符串的开始
$	匹配字符串的结尾
*	在我前面的任意个字符
+	在我前面的一个或多个字符
|	左右两边的字符任选其一
\w	匹配任何字母
\d	匹配任何十进制数字
tomato	匹配字符串 tomato

一个实际的正则表达式问题

假设你需要写一个程序，它的作用是用“bell pepper”代替字符串“green pepper”和“red pepper”，条件是它们在一个段落里同时出现并且不在单词“salad”前，并且后面不能紧跟“corn”。这一类的要求在计算中是常见的。假设你需要处理的文件名叫*pepper.txt*。这里是一个文件的例子：

```
This is a paragraph that mentions bell peppers multiple times. For
one, here is a red pepper and dried tomato salad recipe. I don't like
to use green peppers in my salads as much because they have a harsher
flavor.

This second paragraph mentions red peppers and green peppers but not
the "s" word (s-a-l-a-d), so no bells should show up.

This third paragraph mentions red peppercorns and green peppercorns,
which aren't vegetables but spices (by the way, bell peppers really
```

```
aren t peppers, they're chilies, but would you rather have a good cook
or a good botanist prepare your salad?).
```

第一个任务是打开文件并读入文本：

```
file = open('pepper.txt')
text = file.read()
```

我们一次读入整个文件而不是分割成行，因为我们假设段落是由两个连续的换行符定义的。这可用 string 模块的 split 函数轻易地实现：

```
import string
paragraphs = string.split(text, '\n\n')
```

现在我们把文本分割成段落的列表，接下来要做的就是替换操作。是使用正则表达式的时候了：

```
import re
matchstr = re.compile(
    r"""\b(red|green)        # 'red' 或 'green' 开始的新单词
        (\s+                 # 跟随的空白符
         pepper              # 单词 'pepper'
         (?!corn)            # 如果后面不紧跟 'corn'
         (?=.*salad))""",    # 并且后面某个地方有 'salad',
      re.IGNORECASE |        # 允许 pepper, Pepper, PEPPER 等
      re.DOTALL |            # 允许 '.' 匹配换行符
      re.VERBOSE)            # 允许上面的注释和换行
for paragraph in paragraphs:
    fixed_paragraph = matchstr.sub(r'bell\2',paragraph)
    print fixed_paragraph+'\n'
```

粗体的行是最难的部分，它创建了一个编译好的表达式模式，就像一个程序一样。这个模式指定了两件事：我们对字符串的哪一个部分感兴趣，以及如何分组。

定义我们感兴趣的部分也就是指定一个字符串匹配模式。这是用一连串小的模式实现的，每一个小模式都指定了一个简单的匹配标准（例如“匹配字符串 'pepper'”，“匹配一个或多个空白符”，“不要匹配 'corn'”等等）。我们逐行来看 re.compile(...)表达式。

第一个要注意的是，re.compile()里的字符串是“原始”字符串（引号前有一个

r)。字符串的这个r关闭了对字符串中反斜杠字符（'\'）的解释（注5）。我们也可用普通的字符串，用\\b代替\b，\\s代替\s，这没有什么区别。对复杂的正则表达式来说，原始字符串看起来语法更清晰一些。

模式的第一行是\b(red|green)。\b代表“在单词头和尾部的空字符串”，用在这里是防止匹配red或green是一个单词的一部分的情况（如“tired pepper”）。(red|green)表是一种选择：'red'或'green'。先暂时忽略掉左括号。\s是一个特殊符号，意思是“任何空白符”，而+的意思是“我前面字符的一个或多个重复”，所以加在一起\s意思是“一个或多个空白符”。接下来的pepper就是指字符串"pepper"。(?!corn)防止匹配紧跟有'corn'的情况，也就是避免了匹配'peppercorn'。最后，(?!=.*salad)说的是在后面有任意的字符（.*的意思），后面跟随一个salad。现在我们已经定义了匹配的模式。

现在注意有两组括号我们还没有解释——一个在\s前，一个在最后。括号是用来定义组的，我们将在后面的替换操作里用到组。我们先看看三个用逻辑或连接起来的标志。第一个re.IGNORECASE意思是在比较时忽略掉大小写。第二个re.DOTALL意思是“.”字符可匹配任何字符，包括换行符（缺省不包括）。最后一个，re.VERBOSE使我们可以在正则表达式里插入注释和换行，这样更易于阅读和理解。我们也可以写得更紧凑一些：

```
matchstr = re.compile(r"\b(red|green)(\s+pepper(?!corn)(?=.*salad))", re.I| re.S)
```

实际的替换操作如下：

```
fixed_paragraph = matchstr.sub(r'bell\2', paragraph)
```

首先很清楚的是，我们在调用matchstr对象的sub方法。该对象是一个编译好的正则表达式对象，意思是表达式的一些工作已经完成了（这里是在循环外），所以可加速整个程序的执行。我们又使用了原始字符串。\2是对正则表达式的第二组括号的引用。那么，它能工作吗？pepper.txt文件有三个段落：第一段是两次满足匹配，第二段不满足是因为它没有单词'salad'，而第三段不满足是因为

注5：原字符串不能用奇数个反斜杠字符结尾。用于正则表达式时这不是问题，因为正则表达式不可能用反斜杠字符结尾。

'red'和'green'是在peppercorn前而不是pepper前。正如我们所设想的，我们的程序（保存为文件*pepper.py*）只修改了第一段：

```
/home/David/book$ python pepper.py
This is a paragraph that mentions bell peppers multiple times. For
one, here is a bell pepper and dried tomato salad recipe. I don't like
to use bell peppers in my salads as much because they have a harsher
fiavor.

This second paragraph mentions red peppers and green peppers but not
the "s" word (s-a-l-a-d), so no bells should show up.

This third paragraph mentions red peppercorns and green peppercorns,
which aren't vegetables but spices (by the way, bell peppers really
aren't peppers, they're chilies, but would you rather have a good cook
or a good botanist prepare your salad?).
```

这个例子显示了正则表达式可以紧凑的表达复杂的匹配规则。如果这类的问题经常出现在你的工作中，那么掌握正则表达式是很值得的。

对正则表达式的完整讲述已经超出了本书的范围。Jeffrey Friedl的《Mastering Regular Expression》(O'Reilly & Associates出版)是一本优秀的书。他对Python正则表达式的描述（至少是第一版里）使用了老式的语法，这是需要忽略掉的。Python现在使用的正则表达式很像Perl语言。但他的书对任何想认真作文本处理的人仍然是不可缺少的。对于偶尔的用户(如那些作者)，库参考中的描述大多数时候就够用了。使用re模块，不要使用regexp、regex、regsub这些过时的模块。

通用的操作系统接口：os模块

操作系统接口定义了程序要处理的文件、进程、用户以及线程这些东西。

os和os.path模块

os模块提供了操作系统最基本的工具集的接口。一些特定的接口与你使用的平台有关。(例如与权限相关的调用只对支持它们的平台适用，如Unix和Windows。)尽管这样，还是推荐你使用os模块，而不是平台相关的模块版本（如posix、nt

和mac)。表8-7列出了os模块中最常用的函数。当在os模块的上下文中使用文件时，我们用的是文件名而不是文件对象。

表8-7 os模块最常用的函数

函数名	行为

getcwd()	返回当前目录的路径字符串：
	`>>> print os.getcwd()`
	`h:\David\book`

listdir(*path*)	返回指定目录的所有文件名的列表：
	`>>> os.listdir(os.getcwd())`
	`['preface.doc', 'part1.doc', 'part2.doc']`

chown(*path*, *uid*, *gid*)	改变指定文件的拥有者ID（owner ID）和组ID（group ID）
chmod(*path*, *mode*)	用数字*mode*改变制定文件的许可权限（如0644意思是拥有者有读/写权限，其他人有读权限）
rename(*src*, *dest*)	把名为*src*的文件改名为*dest*
remove(*path*)或unlink(*path*)	删除指定的文件（参见rmdir用于删除目录）
mkdir(*path* [, *mode*])	创建名为path,权限模式为mode的目录(参见os.chmod)： `>>> os.mkdir('newdir')`
rmdir(*path*)	删除名为*path*的目录
system(*command*)	在一个子shell（subshell）里执行一个shell命令，并把该命令返回的数字作为返回值
symlink(*src*, *dest*)	创建从*src*到*dest*的软链接文件
link(*src*, *dest*)	创建从*src*到*dest*的硬链接

在os模块里还有很多其他函数，事实上，只要是POSIX的标准函数以及在大多数Unix平台广泛使用的函数在Python的Unix版本里都能支持。这些函数的接口都遵循POSIX的惯例。你可以取得和设置UID、GID以及进程组，还可创建管道、操作文件描述符、操作进程等等。

os 模块里也定义了一些不是函数的重要属性:

- os.name属性定义了当前使用的操作系统的版本。已注册的名字有'posix'、'nt'、'dos'、'mac'等。它与我们前面讨论的 sys.platform是不同的。
- os.error定义了os模块内引发异常时使用的类。当这个异常引发时，该异常带有两个信息。第一个是对应的错误号(称为errno)，第二个是解释它的字符串信息(称为strerror):

```
>>> os.rmdir('nonexistent_directory')    # 它通常的显示方式
Traceback (innermost last):
  File "<stdin>", line 1, in ?
os.error: (2, 'No such file or directory')
>>> try:                                 # 我们可以捕捉这个错误
...     os.rmdir('nonexistent directory')
... except os.error, value:
...     print value[0], value[1]
...
2 No such file or directory
```

- os.environ字典里包含了启动Python的shell的环境变量。因为用os.system调用的命令会继承这个环境，修改它也就修改了环境:

```
>>> print os.environ['SHELL']
/bin/sh
>>> os.environ['STARTDIR'] = 'MyStartDir'
>>> os.system('echo $STARTDIR')          # 'echo %STARTDIR%' (DOS/Win上)
MyStartDir                               # shell打印变量值
0                                        # echo的返回值
```

os模块也定义了一组与目录操作相关的字符串，而且适用于不同平台，见表8-8。

表8-8 os模块的字符串属性

属性名	意义和值
curdir	表示当前目录的字符串: Unix和Windows上是'.'; Mac上是':'
pardir	表示当前目录的字符串: Unix和Windows上是'..'; Mac上是'::'
sep	路径名的分隔符: Unix是'/'; DOS, Windows是'\'; Mac上是':'

表8-8 os模块的字符串属性（续）

属性名	意义和值
altsep	另一个分隔符，DOS，Windows上是'/'，其他系统上为None
pathsep	用来分隔路径的分隔符： Unix上是':'，DOS，Windows上是';'

当与os.path模块里的函数结合时这些字符串特别有用，在os.path模块里有很多操作文件路径的函数（参见表8-9）。注意os.path模块是os模块的一个属性，所以你不必显式地导入它，当装入os模块时就自动地装入它。表8-9的例子中的输出是在Windows或DOS上运行的。在其他平台分隔符会不同。

表8-9 os.path模块的最常用函数

函数名	行为
split(*path*) 等价于元组： (dirname(*path*), basename(*path*))	把一个路径分解为头尾两个部分，头是到该目录的路径，尾是文件名： **>>> os.path.split("h:/David/book/part2.doc"** ('h:/David/book', 'part2.doc')
join(*path*, ...)	智能化地把各个部分串为一个路径： **>>> print os.path.join(os.getcwd(),** **... os.pardir, 'backup', 'part2.doc')** h:\David\book\..\backup\part2.doc
exists(*path*)	如果*path*路径存在返回真
expanduser(*path*)	用可选的用户名扩展变量： **>>> print os.path.expanduser('~/mydir')** h:\David\mydir
expandvars(*path*)	返回对应于*path*的环境变量的值： **>>> print os.path.expandvars('$TMP')** C:\TEMP
isfile(*path*), isdir(*path*), islink(*path*), ismount(*path*)	如果*path*分别是文件、目录、链接或安装点（mount point）时返回真

表 8-9 os.path 模块的最常用函数（续）

函数名	行为

normpath(*path*)	对路径作规范化，去掉多余的分隔符：
	`>>> print os.path.normpath("/foo/bar\\../tmp")`
	`\foo\tmp`

samefile(*p*, *q*)	如果 p、q 引用的是同一个文件返回真

walk(*p*, *visit*, *arg*)	对以 p 开始的目录树（包括 p 在内）里的每个目录调用函数 visit，参数为 (arg, dirname, names)。其中参数 dirname 是被访问的目录，参数 names 是目录里的文件列表：
	`>>> def test_walk(arg, dirname, names):`
	`... print arg, dirname, names`
	`...`
	`>>> os.path.walk('..', test_walk, 'show')`
	`show ..\logs ['errors.log', 'access.log']`
	`show ..\cgi-bin ['test.cgi']`
	`...`

拷贝文件和目录：shutil 模块

目光锐利的读者也许已注意到 os 模块不包括 copy 函数，尽管它提供了许多文件相关的函数。在 DOS 上拷贝一个文件也就是用二进制模式打开文件，读入所有数据，再用二进制写模式打开另一个文件，把数据全写到第二个文件中。在 Unix 和 Windows 上这样的拷贝无法拷贝文件相关的 stat 位（权限、修改时间等）。换句话说拷贝不是那么简单。尽管如此，通常你可用一个简单的函数 copyfile，它属于 shutil 模块，这对 Unix、Windows、DOS 都适用。在 shutil 模块里还包括一些有用的函数，见表 8-10。

表 8-10 shutil 模块的函数

函数名	行为
copyfile(*src*, *dest*)	拷贝 *src* 文件到 *dest* 文件（直接的二进制拷贝）
copymode(*src*, *dest*)	把 *src* 的模式信息（权限许可）拷贝到 *dest* 文件

表8-10 shutil模块的函数（续）

函数名	行为
copystat(*src*, *dest*)	把*src*的所有状态信息（mode,utime）拷贝到*dest*文件
copy(*src*, *dest*)	把*src*的数据和状态信息拷贝到*dest*文件（不包括Mac机上的资源派生）
copy2(*src*, *dest*)	把*src*的数据和状态（stat）信息拷贝到*dest*文件（不包括Mac机上的资源派生）
copytree(src, dest, symlinks=0)	递归地用copy2函数拷贝一个目录。symlinks标志指示是拷贝链接还是被链接的文件。目标目录必须是不存在的
rmtree(*path*, *ignore_errors=0*, *onerror=None*)	递归地删除*path*目录。如果*ignore_errors*为0（缺省）就忽略错误，否则若设置了*onerror*就用它来处理错误，若未设置就引发异常

Internet相关的模块

公共网关接口：cgi模块

Python程序常要处理来自Web页的表格。为了简化这个工作，Python标准版本里有一个cgi模块。第十章有一个使用CGI的Python的例子。

操作URL：urllib和urlparse模块

统一资源定位符（URL）是http://www.python.org这样无处不在的字符串（注6）。urllib和urlparse这两个模块提供了处理URL的工具。

urllib定义了一些用于写Web上的活跃分子（机器人、代理等）的函数。见表8-11。

注6：URL的语法是在Web的早期设计的，当时希望用户很少看见它，而只是点击超链接。如果预见到将用于广告的话，很可能就选用更容易发音的语法了。

表 8-11 urllib 模块的函数

函数名	行为

urlopen (*url* [, *data*])	打开一个由URL表示的网络对象，也可以打开本地文件：
	`>>> page = urlopen('http://www.python.org')`
	`>>> page.readline()`
	`'<HTML>\012'`
	`>>> page.readline()`
	`'<!-- THIS PAGE IS AUTOMATICALLY GENERATED.`
	`DO NOT EDIT. -->\012'`

urlretrieve (*url* [, *filename*] [, *hook*])	把一个URL表示的网络对象拷贝到一个本地文件：
	`>>> urllib.urlretrieve('http://www.python.org/',`
	`'wwwpython.html')`

urlcleanup()	清除urlretrive使用的缓冲区

quote(*string* [, *safe*])	用%xx替换string里的特殊字符：
	`>>> quote('this & that @ home')`
	`'this%20%26%20that%20%40%20home'`

quote_plus (*string* [, *safe*])	与quote()一样，但也用+符号替换空格

unquote (*string*)	把%xx替换为对应的单个字符：
	`>>> unquote('this%20%26%20that%20%40%20home')`
	`'this & that @ home'`

urlencode (*dict*)	把一个字典转换为URL编码的字符串，可作为传给urlopen()的可选参数data：
	`>>> locals()`
	`{'urllib': <module 'urllib'>, '__doc__': None, 'X':`
	`3, '__name__': '__main__', '__builtins__': <module`
	`'__builtin__'>}`
	`>>> urllib.urlencode(locals())`
	`'urllib=%3cmodule+%27urllib%27%3e&__doc__=None&x=3&`
	`__name__=__main__&__builtins__=%3cmodule+%27`
	`__builtin__%27%3e'`

urlparse定义了一些用于简化URL分解与合成的函数。见表8-12。

表8-12 urlparse模块的函数

函数名	行为
urlparse(*urlstring* [, *default_scheme* [,*allow fragments*]])	把URL解析为6个部分，形成一个元组（协议，网络位置，路径，参数，查询，段ID）： >>> urlparse('http://www.python.org/FAQ.html') ('http', 'www.python.org', '/FAQ.html', '', '', '')
urlunparse(*tuple*)	把urlparse返回的元组合成为一个URL字符串
urljoin(*base*, *url* [,*allow fragments*])	把一个基本的URL（*base*）和一个相对的URL（*url*）组合为一个完整的（绝对）URL： >>> **urljoin('http://www.python.org', 'doc/lib'** 'http://www.python.org/doc/lib'

特殊的Internet协议

大多数常用的基于TCP/IP的协议都有相关名字的支持模块，有httplib（处理Web页），ftplib（支持文件传输），gopherlib（支持浏览gopher服务），poplib和imaplib分别支持POP3和IMAP服务器。nntplib支持从NNTP服务器上读Usenet新闻，smtplib支持与标准邮件服务器通信。我们在第九章会用到一部分。也有用于建造Internet服务器的模块，比如一个通用的基于socket的服务器（SocketServer），一个简单的Web服务器（SimpleHTTPServer）和一个兼容CGI的HTTP服务器（CGIHTTPServer）。

处理Internet数据

在你用一个Internet协议从网上取得文件后（或在你把文件放到网上前），你必须处理这些文件。它们的格式各不相同。表8-13列出了处理各种特定格式的模块（也有处理声音和图像格式的模块，参见库参考）。

表 8-13 处理 Internet 文件的模块

模块名	文件格式
sgmllib	一个简单的 SGML 解析器
htmllib	HTML 文档的解析器
xmllib	XML 文档的解析器
formatter	通用的输出格式化模块和设备接口
rfc822	解析 RFC-822 邮件首部（如："主题：你好啊！"）
mimetools	解析 MIME 格式的信息体（即文件附件）的工具
binhex	用 binhex4 格式编码和解码文件
uu	用 uuencode 格式编码和解码文件
binascii	在二进制和各种 ASCII 编码之间做转换
xdrllib	编码和解码 XDR 数据
mimetypes	把文件名后缀映射为 MIME 类型
base64	编码和解码 MIME base64 数据
quopri	编码和解码 MIME quoted-printable 数据
mailbox	读各种邮件箱格式
mimify	在邮件信息和 MIME 格式之间做转换

处理二进制数据：struct 模块

一个关于文件操作的常见问题是："Python 是如何处理二进制文件的？"答案是 struct 模块，它输出了三个函数 pack、unpack 和 calcsize，用法很简单。

让我们从一个二进制文件的解码任务开始。想像一个有特定数据格式的二进制文件 *binary.dat*：先是一个版本号浮点数，接下来是表示数据长度的长整数，然后是无符号数据字节。使用 struct 模块的关键是定义一个"格式"字符串，它对应于你想读的数据格式，例如：

```
import struct

data = open('bindat.dat').read()
start, stop = 0, struct.calcsize('fl')
version_number, num_bytes = struct.unpack('fl', data[start:stop])
```

```
start, stop = stop, start + struct.calcsize('B'*num_bytes)
bytes = struct.unpack('B'*num_bytes, data[start:stop])
```

'f'的意思是一个浮点数，'l'的意思是长整数，而'B'的意思是无符号字符。可用的unpack格式见表8-14，使用细节参见库参考。

表8-14 struct模块使用的格式码

格式	C语言类型	Python类型
x	pad byte	无值
c	char	长度为1的字符串
b	signed char	Integer
B	unsigned char	Integer
h	short	Integer
H	unsigned short	Integer
i	int	Integer
I	unsigned int	Integer
l	long	Integer
L	unsigned long	Integer
f	float	Float
d	double	Float
s	char[]	String
p	char[]	String
P	void *	Integer

bytes是一个整数的元组。如果我们知道data里保存的是字符，我们也可用chars=map(chr,bytes)。为了更有效，我们也可以在后一个unpack中用'c'代替'B'，它将返回一个单字符的元组。更有效的办法是指定一个定长的字符串，如下：

```
chars = struct.unpack(str(num_bytes)+'s', data[start:stop])
```

压缩操作恰好相反，它有两个参数，一个格式串和一个变长参数(pack(fmt, v1, v2, ...))，它的工作是把v1,v2...这些参数用格式串fmt压缩为一个“压缩串”。

注意 struct 模块可处理用任何字节序编码的数据（注 7），所以你能写平台独立的二进制文件处理程序。对大文件来说，可考虑使用 array 模块。

调试、时间、优化

最后的几个模块是关于调试、时间和优化的。

首先是调试。Python 有一个标准的调试器模块 pdb。使用 pdb 是很直观的，先导入 pdb 模块，并调用它的 run 函数执行你想调试的代码。例如，你正调试第六章的 *spam.py*，可操作如下：

```
>>> import spam                          # 导入想调试的模块
>>> import pdb                           # 导入 pdb 模块
>>> pdb.run('instance = spam.Spam()')    # 用一个执行语句启动 pdb
> <string>(0)?()
(Pdb) break spam.Spam.__init__                          # 设置断点
(Pdb) next
> <string>(1)?()
(Pdb) n                                                 # 'n' 是 'next' 的缩写
spam.py(3)__init__()
-> def __init__(self):
(Pdb) n
> spam.py(4)__init__()
-> Spam.numInstances = Spam.numInstances + 1
(Pdb) list                                              # 显示源代码
  1     class Spam:
  2         numInstances = 0
  3 B       def __init__(self):                       # 注意 B 是断点
  4  ->         Spam.numInstances = Spam.numInstances + 1  # 执行点
  5         def printNumInstances(self):
  6             print "Number of instances created: ", Spam.numInstances
  7
[EOF]
(Pdb) where                                             # 显示调用堆栈
  <string>(1)?()
```

注 7：计算机列出多字节字的顺序依赖于使用的芯片。Intel 和 DEC 的系统使用低位在前顺序，而 Motorola 和 Sun 的系统使用高位在前顺序。网络传递也是用高位在前顺序，所以 struct 模块在 PC 上做网络 I/O 时就很有用了。

```
> spam.py(4)__init__()
-> Spam.numInstances = Spam.numInstances + 1
(Pdb) Spam.numInstances = 10          # 注意调试时我们可修改变量
(Pdb) print Spam.numInstances
10
(Pdb) continue                        # 执行到下一个断点
--Return--                            # 这里没有断电，所以就结束了
> <string>(1)?()->None
(Pdb) c                               # 退出pdb
<spam.Spam instance at 80ee60>        # 返回的实例
>>> instance.numInstances
11
```

如上所示，用pdb你可列出执行代码（箭头指出将执行的下一行），检查修改变量，设置断点。库参考中有更多细节。

程序有时可能太慢，如果你知道瓶颈并且知道其他算法的话，你也许可以测试不同方法的时间看谁更快。标准的库模块time提供了很多时间操作函数。我们使用其中的一个，它用机器的最高精度返回自上一个起始点的时间。由于我们使用相对时间的比较算法，精度就不太重要。这里是两种创建10000个零的列表的方式：

```
def lots_of_appends():
  zeros = []
  for i in range(10000):
    zeros.append(0)

def one_multiply():
  zeros = [0] * 10000
```

如何计时呢？下面是一个简单办法：

```
import time, makezeros

def do_timing(num_times, *funcs):
    totals = {}
    for func in funcs: totals[func] = 0.0
    for x in range(num_times):
        for func in funcs:
            starttime = time.time()          # 记录起始时间
            apply(func)
            stoptime = time.time()           # 记录结束时间
            elapsed = stoptime-starttime     # 差值就是消耗的时间
```

```
            totals[func] = totals[func] + elapsed
    for func in funcs:
        print "Running %s %d times took %.3f seconds" % (func.__name__,
                                                        num_times
                                                        totals[func])
do_timing(100, (makezeros.lots_of_appends, makezeros.one_multiply))
```

运行这个程序的输出是：

```
csh> python timings.py
Running lots_of_appends 100 times took 7.891 seconds
Running one_multiply 100 times took 0.120 seconds
```

你也许会怀疑，一个单列表的乘法会比许多的append快得多。注意在计时的时候最好是大量的函数调用，否则计时可能被一些无关的部分影响，例如网络堵塞和GUI事件。

如果你写了一个复杂的程序，而它比你预想的慢，但你不能断定问题在哪里，怎么办呢？这时候你需要剖析你的程序，看是哪一部分消耗时间，是否能优化，或者修改程序结构以摆脱瓶颈。Python有一个标准的工具*profile*模块，参见库参考。假设你想剖视一个在当前名字空间的函数，操作如下：

```
>>> from timings import *
>>> from makezeros import *
>>> profile.run('do_timing(100, (lots_of_appends, one_multiply))')
Running lots_of_appends 100 times took 8.773 seconds
Running one_multiply 100 times took 0.090 seconds
         203 function calls in 8.823 CPU seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      100    8.574    0.086    8.574    0.086 makezeros.py:1(lots_of_appends)
      100    0.101    0.001    0.101    0.001 makezeros.py:6(one_multiply)
        1    0.001    0.001    8.823    8.823 profile:0(do_timing(100,
                                                (lots_of_appends, one_multiply)))
        0    0.000             0.000          profile:0(profiler)
        1    0.000    0.000    8.821    8.821 python:0(194.C.2)
        1    0.147    0.147    8.821    8.821 timings.py:2(do_timing)
```

你可以看到这给出了一个较复杂的列表，包括每次调用的时间和调用次数。在复杂程序里profile可帮助找到问题。优化Python程序已经超出了本书的范围，不

过如果你感兴趣，可访问Python的新闻组：定期地会有用户要求帮助优化一个程序，而一个自发的竞争就开始了，并且有一些来自专家的建议。

练习

1. **列出目录的内容。**写一个以目录名为参数的函数，递归地列出目录的内容，打印每个文件的名字和大小，一直到最终的目录。

2. **修改提示符。**修改你的解释器的提示符，要求显示当前目录和已经键入的行数，以取代 >>> 提示符。

3. **避开正则表达式。**写一个程序做与*pepper.py*同样的事情，但不使用正则表达式。这有点困难，但它是一个对程序逻辑的有益的练习。

4. **用一个类打包一个文本文件。**写一个类，以一个文件名为参数，读入文件里的文本。这个类有三个方法：`paragraph`，`line`，`word`，每个方法都有一个整数参数，假设`mywrapper`是这个类的实例，打印`mywrapper.paragraph[0]`就打印出文件的第一段，`mywrapper.line[-2]`就打印出文件的倒数第二行，而`mywrapper.word[3]`就打印出文件的第四个单词。

5. **常见任务。**这些练习在附录三中没有答案，而是取自第九章的例子。如果你喜欢挑战的话现在就试一试！

   — 如何拷贝一个列表对象？字典呢？

   — 如何对一个列表排序？如何让列表的成员的次序随机化？

   — 如果你已经知道堆栈这个数据结构，如何实现它？

   — 写一个程序计算文件的行数。

   — 写一个程序打印一个文件的所有以 # 开头的行。

   — 写一个程序打印一个文件的每一行的第四个单词。

   — 写一个程序计算在一个文件里一个单词出现了多少次。

   — 写一个程序在一个目录里的所有文件里找出一个字符串的所有出现。

第九章

用Python完成常见的任务

本章内容：
- 数据结构操作
- 文件操作
- 操作程序
- 与Internet相关的任务
- 较大的例子
- 练习

现在，我们已学习了Python的语法，它的基本的数据类型，和很多我们喜欢的Python的库函数。本章假设你至少理解了这门语言的所有基本成分，并且除了Python的优雅和“酷”的方面外，也了解了它实用的方面。我们将介绍Python程序员要面对的常见的任务。这些任务分为——数据结构操作，文件操作等等。

数据结构操作

Python的最大的特点之一是它把列表、元组和字典作为内置类型。它们非常灵活和容易使用，一旦你开始使用它们，你将发现你会不由自主地想到它们。

内嵌（inline）拷贝

由于Python引用的管理模式，语句a = b并没有对b引用的对象作拷贝，而只是对那个对象产生了新的引用。有时需要一个对象的新的拷贝，而不只是共享一个引用。怎样做到这一点依赖于对象的类型。拷贝列表和元组的最简单的方式有点奇怪。如果myList是一个列表，那么要对它做拷贝，你可以用：

```
newList = myList[:]
```

你可以理解为“从开始到结尾的分片”，因为我们在第二章“类型和操作符”里学

到，一个分片开始的缺省索引是序列的开始（0），而缺省的结尾是序列的结尾。由于元组支持同样的分片操作，这个技术也适用于拷贝元组。而字典却不支持分片操作。为了拷贝字典 myDict，你可以用：

```
newDict={}
for key in myDict.keys():
    newDict[key] = myDict[key]
```

这个操作很常见，所以在Python 1.5里为字典对象增加了一个新方法来完成这个任务，就是 copy()方法。所以前面的代码可以替换为一句话：

```
newDict = myDict.copy()
```

另一个常见的字典操作现在也是标准的字典特性了。如果你有一个字典 oneDict，而想用另一个不同的字典 otherDict 的内容替换它，只需要用：

```
oneDict.update(otherDict)
```

这与下面的代码相同：

```
for key in otherDict.keys():
    oneDict[key] = otherDict[key]
```

如果在 update()操作前 oneDict 与otherDict 共享一些键时，在 oneDict 中的键关联的旧值将被删除掉。这也许是你所想要的（通常是这样，这也是为什么选择这个操作并称之为update()的原因）。如果这不是你期望的，那么要做的也许是抱怨（引发异常），如下：

```
def mergeWithoutOverlap(oneDict, otherDict):
    newDict = oneDict.copy()
    for key in otherDict.keys():
        if key in oneDict.keys():
            raise ValueError, "the two dictionaries are sharing keys!"
        newDict[key] = otherDict[key]
    return newDict
```

或者把二者的值结合为一个元组，例如：

```
def mergeWithOverlap(oneDict, otherDict):
    newDict = oneDict.copy()
```

```
    for key in otherDict.keys():
        if key in oneDict.keys():
            newDict[key] = oneDict[key], otherDict[key]
        else:
            newDict[key] = otherDict[key]
    return newDict
```

为了说明前面三个算法的不同，考虑下面两个字典：

```
phoneBook1 = {'michael': '555-1212', 'mark': '554-1121', 'emily': '556-0091'}
phoneBook2 = {'latoya': '555-1255', 'emily': '667-1234}
```

如果 phoneBook1 可能是过时的，而 phoneBook2 更新一些但不够完整，那么正确的用法可能是 phoneBooke1.update(phoneBook2)。如果认为两个电话本不应该有重复的键时，使用 newBook = mergeWithoutOverlap(phoneBook1, phoneBook2) 可以让你知道假设是否有错。最后一种，如果一个是家里的电话本而另一个是办公室的电话本，那么只要是后续的引用 newBook['emily'] 的代码能够处理 newBook['emily'] 是元组 ('556-0091','667-1234') 这一事实，就可以用：

```
newBook = mergeWithoutOverlap(phoneBook1, phoneBook2)
```

拷贝：copy 模块

回到拷贝主题上来：[:] 和 .copy() 技巧适用于90%的情况。如果你正按照Python的精神，编写可以处理任何参数类型的函数，有时需要拷贝 X 而不管 X 是什么。这时就需要 copy 模块。它提供了两个函数，copy 和 deepcopy。第一个就像序列的分片操作 [:] 或是字典的 copy 方法。第二个函数更微妙并且与深度嵌套结构有关（这正是 *deepcopy* 的意思）。例如用分片操作 [:] 完整地拷贝 listOne。这个技术产生了新的列表，如果原来的列表中的内容是不变的对象，如数字或字符串，这个拷贝就是“真正的”拷贝。然而假设 listOne 的第一项是一个字典（或任何其他容易变化的对象），那么 listOne 的拷贝的第一项只是对同一个字典的新的引用。所以如果你修改了那个字典，显然 listOne 和它的拷贝都修改了。用一个例子可以看得更清楚些：

```
>>> import copy
>>> listOne = [{"name": "Willie", "city": "Providence, RI"}, 1, "tomato", 3.0]
```

```
>>> listTwo = listOne[:]   # or listTwo=copy.copy(listOne)
>>> listThree = copy.deepcopy(listOne)
>>> listOne.append("kid")
>>> listOne[0]["city"] = "San Francisco, CA"
>>> print listOne, listTwo, listThree
[{'name': 'Willie', 'city': 'San Francisco, CA'}, 1, 'tomato', 3.0, 'kid']
[{'name': 'Willie', 'city':'San Francisco, CA'}, 1, 'tomato', 3.0]
[{'name': 'Willie', 'city': 'Providence, RI'}, 1, 'tomato', 3.0]
```

正如你所见，直接修改 listOne 仅仅修改了 listOne。对 listOne 的第一项的修改影响到 listTwo，但没有影响 listThree。这就是浅度拷贝（[:]）和深度拷贝的区别。copy 模块的函数知道如何拷贝可以拷贝的内置函数（注 1），包括类和实例。

排序

在第二章你知道列表有一个排序方法，有时你想要遍历整个排好序的列表而不想影响列表的内容。或者你也许想列出一个排好序的元组，而元组是不可变的，不允许有排序的方法。唯一的解决办法是拷贝一个列表，然后派序列表。例如：

```
listCopy = list(myTuple)
listCopy.sort()
for item in listCopy:
    print item                                  # 或者做别的任何事情
```

这也是处理那些没有内在顺序的数据结构的办法，例如字典。字典非常快的一个原因是实现时保留了改变键的顺序的权利。这其实不是一个问题，因为你可以拷贝字典的键然后遍历它：

```
keys = myDict.keys()                            # 返回字典的未排序的键

keys.sort()
for key in keys:                                # 答应以键为序的健值对
    print key, myDict[key]
```

列表的 sort 方法使用的是 Python 的标准比较方案。但有时你需要别的方案。例

注 1. 有些对象是不可拷贝的，如模块、文件对象和套接字。记住，文件对象与磁盘上的文件是不同的。

如当你对一个单词列表排序时，大小写也许是没有意义的。而对字符串的标准比较中，所有大写字母都在小写之前，所以'Baby'小于'apple'而'baby'大于'apple'。为了进行大小写无关排序，你需要定义一个有两个参数的函数，并且根据第一个参数是小于，等于或大于第二个参数，分别返回-1，0，1。所以你可以这样写：

```
>>> def caseIndependentSort(something, other):
...     something, other  = string.lower(something), string.lower(other)
...     return cmp(something, other)
...
>>> testList = ['this', 'is', 'A', 'sorted', 'List']
>>> testList.sort()
>>> print testList
['A', 'List', 'is', 'sorted', 'this']
>>> testList.sort(caseIndependentSort)
>>> print testList
['A', 'is', 'List', 'sorted', 'this']
```

我们使用内置的函数cmp来完成比较工作。我们的排序函数只是把两项变成小写字母然后排序。也请注意小写转换是在比较函数局部范围内的，所以列表中的元素并没有因排序而修改。

随机：random模块

怎样随机排列一个序列呢？比如一个文本行的列表。最简单的办法是使用random模块里的choice函数，它随机地返回序列的元素作为它的参数（注2）。为了避免重复地得到同样的行，记住删除已经选择了的项。当操作一个列表对象时，使用remove方法：

```
while myList:                          # 当myList空时停止循环
    element = random.choice(myList)
    myList.remove(element)
    print element,
```

如果你需要随机处理一个非列表对象，通常最简单的办法是把它转换为一个列表，

注2：random模块提供了很多有用的函数，例如random函数，它返回一个介于0和1之间的随机浮点数。

然后对这个列表作随机处理，而不是对每种数据类型都采用新的办法。这似乎是一个浪费的办法，也许要产生一个很巨大的列表。然而一般来说，对你似乎很大的数据，对于计算机来说可能不那么大，感谢Python的引用系统。而且不用对每种数据类型采用不同的方法，所节约的时间是很多的。Python的设计初衷就是要节约时间；如果那意味着运行一个稍慢一点或者大一点的程序，那就让它这样吧。如果你正在处理大量的数据，也许值得优化。但只有当确实需要优化时才去优化，否则将是浪费时间。

定义新的数据结构

对于数据结构来说，不要重复发明轮子这一原则尤其重要。例如，Python的列表和字典也许不是你习惯于使用的，但如果这些数据结构可以满足要求，你应当避免设计自己的数据结构，它们使用的算法已经在各种情形下测试过，并且快而稳定。但有时这些算法的接口对某个特别的任务不方便。

例如，计算机科学的教科书中经常用其他数据结构术语如队列、堆栈来描述算法。为了使用这些算法，定义与这些数据结构有同样方法的数据结构也许是有意义的。(比如堆栈的pop和push，或者队列的enqueue和dequeue)。而且，重用内置的列表类型来实现堆栈也是有意义的。换句话说，你需要行为像堆栈但却是基于列表的结构。最简单的办法是用一个类来包裹一个列表。为了最低限度地实现一个类，你可以这样写：

```
class Stack:
    def __init__(self, data):
        self._data = list(data)
    def push(self, item):
        self._data.append(item)
    def pop(self):
        item = self._data[-1]
        del self._data[-1]
        return item
```

下面的语句不仅易写，易懂，而且易读，易用。

```
>>> thingsToDo = Stack(['write to mom', 'invite friend over', 'wash the
kid'])
```

```
>>> thingsToDo.push('do the dishes')
>>> print thingsToDo.pop()
do the dishes
>>> print thingsToDo.pop()
wash the kid
```

在上面的堆栈类中用了两个标准的Python命名习惯，第一个是类名由大写字母开始，以便与函数名区别开。另一个是_data属性以一个下划线开始，这介于公共属性（不以下划线开始）和私有属性之间（以两个下划线开始，参见第六章“类”）。而Python的保留字在开始和结尾都有两个下划线。这里的意思是：_data是一个属性，用户不应该直接访问，类的设计者希望这个“伪私有”属性只被类和子类的方法使用。

定义新的列表和字典：UserList 和 UserDict 模块

前面展示的堆栈类作了恰当的工作。它采取了关于堆栈的最小定义，只支持两个操作：push 和pop。然而，很快你就发现列表的特性确实好，比如可以用for...in...的方式访问所有的成员。这可以通过重用已有的代码来实现。在这里你应当应用UserList模块里定义的类UserList作为基类，堆栈由此派生而来。库里也包括 UserDict 模块，它是一个封装字典的类。一般来说，它们是用于特别子类的基类。

```
# 从UserList 模块中导入 UserList 类
from UserList import UserList

# 继承 UserList 类
class Stack(UserList):
    push = UserList.append
    def pop(self):
        item = self[-1]                    # 使用__getitem__
        del self[-1]
        return item
```

这个堆栈是 UserList 的一个子类。UserList 类通过定义特别的__getitem__和__delitem__方法实现了方括号的运算，所以前面代码里的pop能工作。你不必定义你自己的__init__方法，因为UserList已经定义了一个相当不错的。最后只是说明 push 方法等于 UserList 的append方法。现在我们可以用列表和堆栈两种方式来操作了。

```
>>> thingsToDo = Stack(['write to mom', 'invite friend over', 'wash the
kid'])
>>> print thingsToDo                    # 从UserList继承
['write to mom', 'invite friend over', 'wash the kid']
>>> thingsToDo.pop()
'wash the kid'
>>> thingsToDo.push('change the oil')
>>> for chore in thingsToDo:            # 我们也可以用"for ..in .."遍历内容
...     print chore                     # 因为有__getitem__
...
write to mom
invite friend over
change the oil
```

注意：当我们写这本书时，Guido van Rossom宣布在Python 1.5.2(以及后续版本里)，列表对象将增加一个pop方法，它也有一个可选参数来指定pop的索引(缺省是列表最后的一个成员)。

文件操作

脚本语言的设计目标之一是帮助人们快速而简单地做重复工作。Web管理员，系统管理员和程序员的经常需要做的一件事是：从一个文件集合中选出一个子集，对这个子集做某种操作，并把结果写到一个或一组输出文件中(例如，在某个目录里的每个文件里，隔行查找以非#字符开头的行的最后一个词，并把它与文件名一起打印出来)。人们为这类任务已经设计了特定的工具，例如*sed*和*awk*。我们发现Python能很简单地完成这个工作。

操作一个文本文件里的每一行

当解析一个包含文本的输入文件时，sys模块是非常有用的。在它的属性中有三个文件对象，分别称为sys.stdin、sys.stdout和sys.stderr。名字来源于三个流的概念：分别为标准输入、标准输出和标准错误。它们与命令行工具有关，print语句使用标准输出。它是一个文件对象，具有以写模式打开的文件对象的所有输出方法，如write和writelines。另一个常用的流是标准输入(stdin)，它也是一个文件对象，不过它拥有的是输入方法，例如read、readline和readlines。下面的脚本会算出通过“管道”输入的文件行数：

```
import sys
data = sys.stdin.readlines()
print "Counted", len(data), "lines."
```

在 Unix 上你可以做如下的测试：

```
% cat countlines.py | python countlines.py
Counted 3 lines.
```

在 Windows 或 DOS 上，你可以：

```
C:\> type countlines.py | python countlines.py
Counted 3 lines.
```

当实现简单的过滤操作时，readlines 函数是有用的。这里有一些过滤操作的例子：

寻找所有以 # 开始的行

```
import sys
for line in sys.stdin.readlines():
    if line[0] == '#':
        print line,
```

注意 print 语句后的逗号是需要的，因为 line 字符串里已经有一个换行符。

取出一个文件的第四列（这里列是由空白符定义的）

```
import sys, string
for line in sys.stdin.readlines():
    words = string.split(line)
    if len(words) >= 4:
        print words[3]
```

我们通过 words 列表的长度判断是否确实至少有四个列，最后两行可以用 try/except 惯用法代替，这在 Python 里是常见的：

```
    try:
        print words[3]
    except IndexError:                    # 没有足够的列
        pass
```

取出文件的第四列，列由冒号分开，并用小写输出

```
import sys, string
for line in sys.stdin.readlines():
    words = string.split(line, ':')
```

```
    if len(words) >= 4:
        print string.lower(words[3])
```

打印头 10 行，最后 10 行，并隔行打印输出

```
import sys, string
lines = sys.stdin.readlines()
sys.stdout.writelines(lines[:10])          # 头10行
sys.stdout.writelines(lines[-10:])         # 最后10行
for lineIndex in range(0, len(lines), 2):  # 0, 2, 4, ......
    sys.stdout.write(lines[lineIndex])     # 0, 2, 4, ......行
```

计算单词“Python”在一个文件里出现的次数

```
import string
text = open(fname).read()
print string.count(text, 'Python')
```

把一个文件的列变换为一个列表的行

在这个比较复杂的例子里，任务是“转置”一个文件，设想你有这样一个文件：

```
Name:   Willie   Mark   Guido   Mary  Rachel   Ahmed
Level:     5       4       3       1     6        4
Tag#:    1234    4451    5515    5124   1881    5132
```

而你希望它变成这样：

```
Name:  Level:  Tag#:
Willie 5       1234
Mark   4       4451
...
```

你可以用下面的代码：

```
import sys, string
lines = sys.stdin.readlines()
wordlists = []
for line in lines:
    words = string.split(line)
    wordlists.append(words)
for row in range(len(wordlists[0])):
    for col in range(len(wordlists)):
        print wordlists[col][row] + '\t',
    print
```

当然你应当用更多的防卫性编程技巧来处理一些可能的情况，比如也许不是所有的行都有相同的单词数，也许丢失了数据等等。这些就作为练习留给读者。

选择数据块的大小

前面的所有例子都假设你能一次读入整个文件。然而有时候这是不可能的，比如在内存较小的计算机上处理大文件，或者处理不断地增加的文件（如日志文件）。对这种情况你可以用一个while/readline组合，一次读入文件的一小部分直到读完。对于不是基于行的文本文件，你必须一次读入一个字符：

```
# 逐字地读入
while 1:
    next = sys.stdin.read(1)          # 读入一个单字符串
    if not next:                      # 或者读到 EOF 时是空串
        break
    处理字符串 'next'
```

注意文件对象的read()方法在文件结尾时返回一个空串，并由此而跳出循环。然而更常见的是你将处理基于行的文本文件，并且一次处理一行：

```
# 逐行地读入
while 1:
    next = sys.stdin.readline()       # 读入一个单行字符串
    if not next:                      # 或者读到 EOF 时是空串
        break
    处理行 'next'
```

处理命令行上指定的一组文件

能够读stdin是一个重要的特征，它是Unix工具集的基础。可是一个输入并不总是足够的：很多任务需要操作一组文件。这通常是由Python解释器解析作为命令选项传给脚本的参数列表。例如，你键入：

```
% python myScript.py input1.txt input2.txt input3.txt output.txt
```

你也许打算让 *myScript.py* 处理前三个文件，并写一个名为 *output.txt* 的新文件。让我们看这样一个程序的开头是怎样的：

```
import sys
inputfilenames, outputfilename = sys.argv[1:-1], sys.argv[-1]
for inputfilename in inputfilenames:
    inputfile = open(inputfilename, "r")
    do_something_with_input(inputfile)
```

```
outputfile = open(outputfilename, "w")
write_results(outputfile)
```

第二行提取了sys模块的argv属性的部分。回忆一下那是命令行上的单词列表。列表以这个脚本的名字开始，所以在上面的例子中sys.argv的值是：

```
['myScript.py', 'input1.txt', 'input2.txt', 'input3.txt', 'output.txt']
```

脚本假设命令行有一个或多个输入文件以及一个输出文件组成，所以输入文件名的分片起始于1（跳过脚本名本身，大多数情况下它都不是脚本的输入文件），而结束于最后一个单词前，最后一个单词是输出文件。脚本的其余部分应当是相当容易理解的。

注意前面的脚本实际上没有从文件里读数据，而是把文件对象传递给做实际工作的函数，该函数常常使用文件对象的readlines()方法，返回文件的行的列表。do_something_with_input()函数的通用版本如下：

```
def do_something_with_input(inputfile):
    for line in inputfile.readlines()
        process(line)
```

处理一个或多个文件的每一行：fileinput模块

上一个例子是用sys.argv[1:]这个惯用组合来打开文件，这种情况十分常见，所以Python 1.5提供了一个新的模块来做这个工作。它的名字是fileinput，它是这样工作的：

```
import fileinput
for line in fileinput.input():
    process(line)
```

fileinput.input()调用解析命令行参数，如果脚本没有参数就以stdin代替。它也提供了一组有用的函数帮助你了解正在处理的文件名和行号：

```
import fileinput, sys, string
# 从sys.argv里取第一个参数并赋值给searchterm
searchterm, sys.argv[1:] = sys.argv[1], sys.argv[2:]
for line in fileinput.input():
    num_matches = string.count(line, searchterm)
```

```
    if num_matches:                 # 大于零表示有匹配
        print "found '%s' %d times in %s on line %d."% (searchterm, num_matches,
            fileinput.filename(), fileinput.filelineno())
```

如果这个脚本称为*mygrep.py*，它可以这样用：

```
% python mygrep.py in *.py
found 'in' 2 times in countlines.py on line 2.
found 'in' 2 times in countlines.py on line 3.
found 'in' 2 times in mygrep.py on line 1.
found 'in' 4 times in mygrep.py on line 4.
found 'in' 2 times in mygrep.py on line 5.
found 'in' 2 times in mygrep.py on line 7.
found 'in' 3 times in mygrep.py on line 8.
found 'in' 3 times in mygrep.py on line 12.
```

文件名和目录

我们已经讨论了如何读存在的文件，如果你还记得在第二章讨论过的open函数的话，你一定知道如何创建新的文件。然而有许多任务需要不同的文件操作，例如目录管理以及删除文件。你处理这些事务的最好帮手是os和os.path模块，我们在第八章“内置工具”里有介绍。

让我们来看一个典型的例子：你有很多文件，它们的名字都有一个空格，而你想用下划线代替空格。你所需要知道的一切就是os.curdir属性（当前目录），os.listdir函数（返回指定目录的文件名列表），以及os.rename函数：

```
import os, string
if len(sys.argv) == 1:                          # 如果没有指定目录
    filenames = os.listdir(os.curdir)           # 就用当前目录
else:                                           # 否则用命令行指定的目录
    filenames = sys.argv[1:]
for filename in filenames:
    if ' ' in filename:
        newfilename = string.replace(filename, ' ', '_')
        print "Renaming", filename, "to", newfilename, "..."
        os.rename(filename, newfilename)
```

这个程序工作得很好，但它显示出一些以Unix为中心的倾向。那就是当你用通配符调用它时，例如：

```
python despacify.py *.txt
```

你会发现在 Unix 系统上它对所有以 *.txt* 结尾的文件都作了替换处理，而在 DOS 下它无法工作，因为 DOS 和 Windows 的 shell（命令解释器）并不把 **.txt* 转换为一个文件名列表，而是希望由应用程序来完成转换，* 的意思是匹配任意的字符。

匹配一组文件：glob 模块

glob 模块只输出一个函数，名字也叫 glob，它以文件名模式为参数并返回匹配这个模式的所有文件名（在当前工作目录）：

```
import sys, glob, operator
print sys.argv[1:]
sys.argv = reduce(operator.add, map(glob.glob, sys.argv))
print sys.argv[1:]
```

在 Unix 上测试表明 glob 函数没有做什么，因为 Unix 的 shell 已经完成了 glob 的工作，而在 DOS 上 Python 的 glob 函数得到了同样的结果：

```
/usr/python/book$ python showglob.py *.py
['countlines.py', 'mygrep.py', 'retest.py', 'showglob.py', 'testglob.py']
['countlines.py', 'mygrep.py', 'retest.py', 'showglob.py', 'testglob.py']

C:\python\book> python showglob.py *.py
['*.py']
['countlines.py', 'mygrep.py', 'retest.py', 'showglob.py', 'testglob.py']
```

这个脚本绝非无足轻重，因为它用到了两个概念上较难理解的函数：一个 map，接着是一个 reduce。map 在第四章“函数”里有介绍，而 reduce 现在对你是全新的（除非你有 LISP 类语言的背景）。map 以一个可调用对象（通常是一个函数）和一个序列为参数，依次地用序列中每一个成员为参数调用这个可调用对象，并返回由该对象（即函数）返回的值组成的列表。图 9-1 有一个 map 的示意图（注 3）。

注 3：map 还有更多的功能。例如，如果 None 是第一个参数，map 就把作为第二个参数的序列转换为一个列表。它一次可以操作多于一个序列。

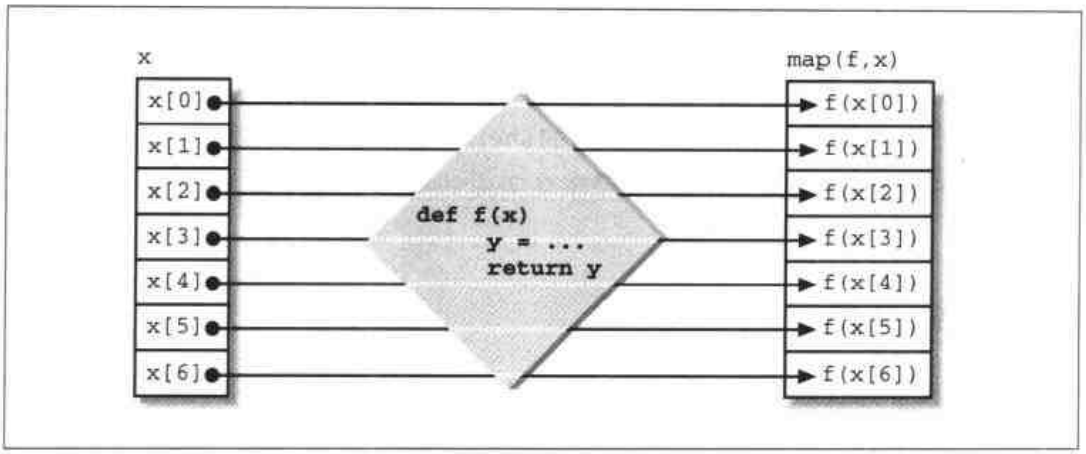


图 9-1 内置 map 函数行为的图形表示

map在这里是需要的，因为你不知道命令行上输入了多少个参数（也许是*.py，*.txt，*.doc）。所以，依次地用每个参数调用glob.glob函数，而每次调用glob就返回匹配文件名模式的文件名列表，map操作就返回一个列表的列表，你需要把它转换为一个单个的列表 —— 把其中的所有列表组合在一起，也就是list1 + list2 + … + lintN。这恰好是reduce函数派上用场的地方。

与map一样，reduce的第一个参数是函数，并用它的第二个参数（必须是序列）的前两个成员作为参数调用该函数，然后再用返回的结果和序列的下一个成员作为参数再调用该函数（直到结束，请看图9-2 reduce的示意图）。但且慢：你需要对一组成员应用“+”，而“+”不像一个函数，所以你需要一个与“+”一样的工作函数，这里有一个：

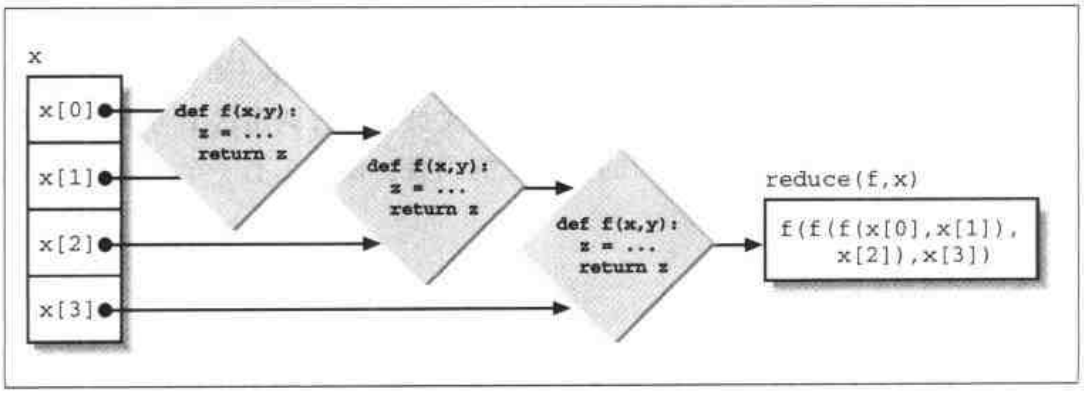


图 9-2 内置 reduce 函数行为的图形表示

```
define myAdd(something, other):
    return something + other
```

于是你可以用reduce(myAdd, map(…))。这是可行的，但你也可以用operator模块定义的add函数，这也许更好些。operator模块为每个Python里的操作都定义了函数（包括取属性和分片操作）。你应当用它而不是你自制的函数，这有两个原因。第一，它们已经由Guido编码、调试和测试过，他在写无故障代码方面有良好的记录；其次，它们实际上是C函数，而对reduce（或map，filter）使用C函数要比用Python函数快得多。当你一次只处理几百个文件时显然是没多大关系，然而如果是数千个文件，速度就很重要了，而现在你知道怎么做更快。

内置的filter函数也像map和reduce一样，以一个函数和序列作为参数，它返回该序列的子集，其中子集的成员能满足函数的条件。下面的例子是找出一个集合里的偶数：

```
>>> numbers = range(30)
>>> def even(x):
...     return x % 2 == 0
...
>>> print numbers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29]
>>> print filter(even, numbers)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

或者如果你想找出一个文件中所有至少10个字符长的单词，可以这样：

```
import string
words = string.split(open('myfile.txt').read())     # 所有的单词

def at_least_ten(word):
    return len(word) >= 10

longwords = filter(at_least_ten, words)
```

图9-3展示了filter的工作。filter有一个特别不错的特点，如果你以None为第一个参数，它将过滤掉序列中所有假值。所以为了找出一个文件中的所有非空行，可这样写：

```
lines = open('myfile.txt').readlines()
lines = filter(None, lines)          # 记住，空串是假
```

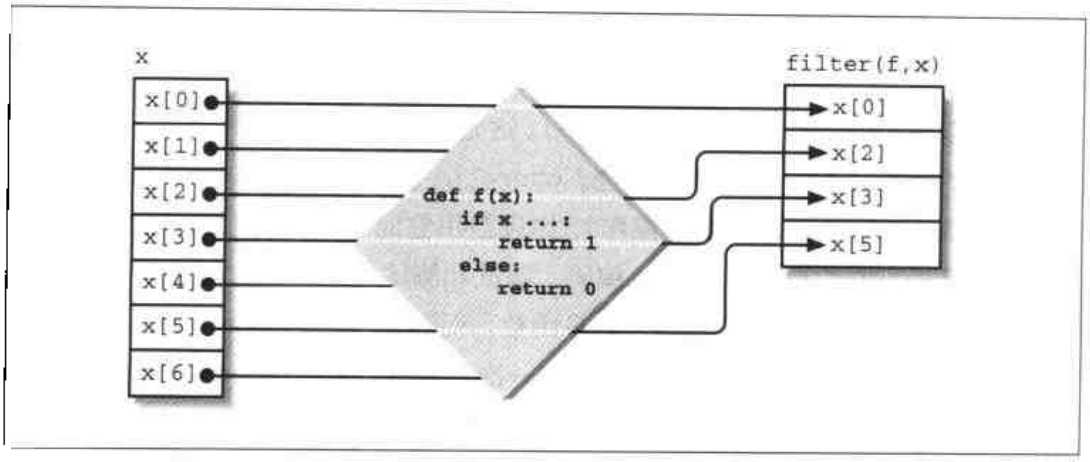


图 9-3 内置 filter 函数行为的图形表示

map、filter和reduce是三个功能很强的函数，它们是值得了解的。然而它们不是必需的。写一个同样功能的Python函数是很简单的，只是内置的版本更快，特别是当操作的是C语言写的函数时，比如operator模块里的函数。

使用临时文件

如果你曾经写过shell脚本，并且用中间文件来存放处理的中间结果，你很可能被目录里的垃圾困扰。你以20个*log_001.txt*，*log_002.txt*等开始，而你想要的只是名为*log_sum.txt*的最终文件，而且你机器里塞满了一系列的*log_001.tmp*、*log_001.tmp2*等临时文件，至少我们的生活是这样的。为了保持你的目录的秩序，用完临时文件就要立即清除掉。

为了帮助解决临时文件的管理问题，Python提供了一个不错的小模块tempfile，它发布了两个函数：mktemp()和TemporaryFile()。前者返回你机器的临时文件目录（如Unix里的*/tmp*和Windows的*c:\tmp*）中未使用的文件名，后者直接返回一个文件对象。例如：

```
# 读输入文件
inputFile = open('input.txt', 'r')

import tempfile
# 创建临时文件
tempFile = tempfile.TemporaryFile()                    # 我们甚至不需要知道文件名
first_process(input = inputFile, output = tempFile)
```

```
# 创建输出文件
outputFile = open('output.txt', 'w')
second_process(input = tempFile, output = outputFile)
```

当需要立即操作文件对象时使用tempfile.TemporaryFile()很合适。它的一个好处是当删除它时它会自动删除对应的文件，保持磁盘的干净。临时文件的一个重要用法是用于os.system调用，而这需要文件名而不是文件对象。让我们看一个例子程序，它创建信件并邮寄给一组邮件地址（只用于Unix）：

```
formletter = """Dear %s,\nI'm writing to you to suggest that ..."""   # 等等
myDatabase = [('Bill Clinton', 'bill@whitehouse.gov.us'),
              ('Bill Gates', 'bill@microsoft.com'),
              ('Bob', 'bob@subgenius.org')]
for name, email in myDatabase:
    specificLetter = formletter % name
    tempfilename = tempfile.mktemp()
    tempfile = open(tempfilename, 'w'
    tempfile.write(specificLetter)
    tempfile.close()
    os.system('/usr/bin/mail %(email)s -s "Urgent!" < %(tempfilename)s' % vars())
    os.remove(tempfilename)
```

for循环的第一行返回一个基于给定名字定制的信件，然后把它写入一个临时文件，并准备用os.system调用把文件邮寄给合适的邮件地址，最后清除掉临时文件。如果你忘记了%的用法，请回到第二章去复习它，它是值得学习的。vars()函数是一个内置函数，它返回对应于局部名字空间的变量的字典，字典的关键字是变量名，字典的值是变量的值。vars()非常适合于探索名字空间，也可以用一个对象为参数调用它（如一个模块，一个类或者一个实例），而它将返回该对象的名字空间。另外两个内置函数locals()和globals()函数分别返回局部和全局名字空间。在这三种情况下，修改返回的字典不保证对相应的名字空间有效，所以最好把它们看作只读的。例子中vars()函数创建了一个字典，并用于修改字符串，在字符串中的%(…)s部分的名字必须与程序里的变量名匹配。

扫描文件技术的更多细节

假设你运行了一个程序，把它的输出存在一个文本文件里，你需要载入它。程序创建一个文件，由一系列行组成，每一行包含由空白符分隔的一个值和一个键：

```
value key
value key
value key
等等...
```

在文件里一个键可在不同行出现，你也许想把一个键的所有值收集在一起，下面是一种可行的办法：

```
#!/usr/bin/env python
import sys, string
entries = {}
for line in open(sys.argv[1], 'r').readlines():
    left, right = string.split(line)
    try:
        entries[right].append(left)        # 扩展列表
    except KeyError:
        entries[right] = [left]            # 第一次遇上

for (right, lefts) in entries.items():
  print "%04d '%s'\titems => %s" % (len(lefts), right, lefts)
```

这个脚本利用了readlines方法逐行扫描文本文件，并调用内置的string.split函数把每一行切割成一个字符串表 —— 表中是值和键。为了保存一个键的所有出现，脚本里用了一个entries字典，循环中的try语句试图把一个键的新值增加到一个存在的成员中，如果没有对应键的成员就创建一个，注意这里的try可以用一个if代替：

```
if entries.has_key(right):             # 在字典里吗?
    entries[right].append(left)        # 把键的当前值加入列表中
else:
    entries[right] = [left]            # 初始化键的列表
```

测试一个字典是否有一个键，有时比用try捕捉异常要快，这依赖于符合测试的次数。这里是运行这个脚本的实际例子，文件名由命令行参数传递(sys.argv[1])：

```
% cat data.txt
1       one
2       one
3       two
7       three
```

```
8       two
10      one
14      three
19      three
20      three
30      three

% python collector1.py data.txt
0003 'one'      items => ['1', '2', '10']
0005 'three'    items => ['7', '14', '19', '20', '30']
0002 'two'      items => ['3', '8']
```

你可以把扫描逻辑打包到一个函数里，函数返回entries字典，并且把循环打印逻辑用一个if测试打包在脚本底部，这回更有用：

```
#!/usr/bin/env python
import sys, string
def collect(file):
    entries = {}
    for line in file.readlines():
        left, right = string.split(line)
        try:
            entries[right].append(left)          # 扩展列表
        except KeyError:
            entries[right] = [left]              # 第一次遇上
    return entries

if __name__ == "__main__":                       # 当作为脚本运行时
    if len(sys.argv) == 1:
        result = collect(sys.stdin)              # 从stdin读如
    else:
        result = collect(open(sys.argv[1], 'r')) # 从文件里读如
    for (right, lefts) in result.items():
        print "%04d '%s'\titems => %s" % (len(lefts), right, lefts)
```

这样程序更灵活一些。通过这个if __name__=="__main__"技巧，你仍然可以把它作为独立脚本来用，或者导入它定义的函数，并处理该函数返回的结果：

```
# 作为独立脚本运行
% collector2.py < data.txt
这里显示结果...

# 用在别的程序中（或交互式的使用）
from collector2 import collect
```

```
result = collect(open("spam.txt", "r"))
这里处理结果...
```

由于collect函数的参数是一个打开的文件，它也可以操作任何提供文件方法的对象。如果你只想扫描一个字符串，可以用一个实现了所需接口的类封装该字符串，并把该类的一个实例传给collect函数：

```
>>> from collector2 import collect
>>> from StringIO import StringIO
>>>
>>> str = StringIO("1 one\n2 one\n3 two")
>>> result = collect(str)                    # 扫描封装的字符串
>>> print result                             # {'one':['1','2'],'two':['3']}
```

这个代码用了标准Python库里的StringIO类，它封装了字符串并提供了所有的文件对象的方法，更多的细节请参考库参考。如果你想修改它的特征，你可以写一个不同的类或从StringIO继承一个子类。无论如何，collect函数愉快地从串str读入，而str是一个内存里的对象，不是一个磁盘文件。

这里行得通的主要原因是，collect函数避免了对它的file参数作任何类型的假设，只要对象输出了一个返回串表的readlines方法，collect不关心它处理的对象的类型，这种运行期绑定（注4）是Python对象系统的重要特征，这使得你可容易地写出与其他不同部件通信的程序。例如，考虑一个用标准文件接口读写卫星遥测数据的程序，你可以把它的输入输出流重定向到socket、图形界面窗口、Web或者数据库，而不需要改变程序，甚至都不需要重新编译。

操作程序

调用别的程序的程序

Python可以像一个shell语言一样用，Python程序可以用运行时确定的参数调用别的工具。所以，如果你必须运行一个特定的程序（名为analyzeData），它运行

注4：运行时绑定的意思是，Python在程序运行时才知道对象的接口。这是因为Python没有类型的声明，这导致了多态性概念的产生。在Python里一个对象操作的意义依赖于被操作的对象。

时需要不同的数据文件和参数，你可以用os.system调用，它的参数是一个命令字符串，如下：

```
for datafname in ['data.001', 'data.002', 'data.003']:
  for parameter1 in range(1, 10):
  os.system("analyzeData -in %(datafname)s -param1 %(paramter1)d" % vars())
```

如果analyzeData是一个Python程序的话，你最好不要激活一个子shell，只需要用import语句导入，并在循环中调用一个函数就可以了。然而，不是每一个有用的程序都是Python程序。

在前面的例子中，analyzeData的输出很可能是一个文件或标准输出。如果是标准输出的话，能够捕捉到它的输出是很不错的。popen()函数就是做这件事的标准方法，我们将用一个实际的例子来展示它的用法。

当我们写这本书时，要求我们避免在源码列表中使用制表符（tab），而代之以空格字符。制表符可能受排版的影响，而Python的缩进格式是重要的，不正确的排版有可能破坏例子的使用。但由于旧有的习惯难以消除（至少我们中的一个喜欢用制表符），所以在交付排版前，我们需要一个工具找出任何可能进入我们代码的制表符。下面的脚本*findtabs.py*就是做这件事的：

```
#!/usr bin/env python
# find files, search for tabs

import string, os
cmd = 'find . -name "*.py" -print'          # find是一个标准Unix工具

for file in os.popen(cmd).readlines():     # 运行find命令
    num  = 1
    name = file[:-1]                       # 去掉'\n'
    for line in open(name).readlines():    # 扫描文件
        pos = string.find(line, "\t")
        if pos >= 0:
            print name, num, pos           # 报告找到的tab
            print '....', line[:-1]        # [:-1]去掉最后的\n
            print '....', ' '*pos + '*', '\n'
        num = num+1
```

这个脚本用了两个嵌套的for循环，外面的循环用os.poepn运行一个find命令，返回在当前目录和子目录里所有的Python程序名。内部的循环逐行阅读当前文

件，用string.find函数寻找制表符。这个脚本的真正魔术就是它调用的内置工具：os.popen：

os.popen

以一个shell命令字符串作为参数，并返回一个链接到标准输入或标准输出的文件对象，如果你没有给出一个"r"或"w"参数的话，缺省是标准输出。通过读入这个文件对象，你可以像我们一样截取命令的输出——find的结果。在标准库里有一个名为find.py的模块提供了一个类似的函数，作为练习，你可以用它来重写*findtabs.py*。

string.find

在一个字符串里从左到右地找一个子串，并返回的一个位置的索引，我们用它来找制表符，'\t'（转义的字符）。

当找到一个制表符时，脚本打印匹配的行，以及一个指示制表符位置的指针。注意字符串重复的用法：表达式' '*pos把打印光标恰好移动到一个制表符的位置。在cmd这个单引号引用的串里使用了双引号，而不是反斜杠转义符号。下面是脚本的工作结果，捕获了非法的制表符：

```
C:\python\book-examples> python findtabs.py
./happyfingers.py 2 0
....    for i in range(10):
.... *

./happyfingers.py 3 0
....            print "oops..."
.... *

./happyfingers.py 5 5
.... print       "bad style"
....       *
```

关于移植性的说明：脚本里用的find命令是一个Unix命令，在其他平台里也许没有。os.popen在Windows版本的win32扩展里对应的是win32pipe.popen(注5)。如果你希望捕获shell命令输出的代码可移植，可使用下面的代码：

注5：两个关于兼容性的重要注释：win32pipe模块也有一个popen2调用，与Unix的popen2调用相似，但它返回读写的管道的次序不同（参阅文档，可获得posix模块中poprn2的更多信息）。Mac机上没有popen的等价物，因为不存在管道。

```
import sys
if sys.platform == "win32":                    # Windows
    try:
        import win32pipe
        popen = win32pipe.popen
    except ImportError:
        raise ImportError, "win32pipe模块找不到"
else:                                          # POSIX
    import os
    popen = os.popen
...随后可使用popen
```

sys.platform属性总是预设为一个标识所在平台的字符串。尽管Python语言是不依赖于平台的，但某些库也许是依赖的，检查sys.platform是一种标准的处理方式。注意这里嵌套的import语句，我们可看到，import是一个可执行语句，它给一个变量名赋值。

与 Internet 相关的任务

Internet是一个信息的宝库，但它的指数级的增长使它很难管理。而且大多数冲浪的工具都是不可编程的。用 Python 标准版的一些工具可以很简单地使很多 Web 相关的任务自动化。

自动下载一个 Web 页

如果你对找出一个地方数月来的天气情况感兴趣，你可以做一个程序自动地去收集信息。

这里是一个程序，它从 weather.com 网站寻找几个州和城市的天气：

```
import urllib, urlparse, string, time

def get_temperature(country, state, city):
    url = urlparse.urljoin('http://www.weather.com/weather/cities/',
                           string.lower(country)+'_' + \
                           string.lower(state) + '_' + \
                           string.replace(string.lower(city), '',
                                          '_') + '.html')
```

```
    data = urllib.urlopen(url).read()
    start = string.index(data, 'current temp: ') + len('current temp: ')
    stop = string.index(data, '&deg;F', start-1)
    temp = int(data[start:stop])
    localtime = time.asctime(time.localtime(time.time()))
    print ("On %(localtime)s, the temperature in %(city)s, " +\
           "%(state)s %(country)s is %(temp)s F.") % vars()

get_temperature('FR', '', 'Paris')
get_temperature('US', 'RI', 'Providence')
get_temperature('US', 'CA', 'San Francisco')
```

它运行时的输出是：

```
~/book:> python get_temperature.py
On Wed Nov 25 16:22:25 1998, the temperature in Paris,  FR is 39 F.
On Wed Nov 25 16:22:30 1998, the temperature in Providence, RI US is 39 F.
On Wed Nov 25 16:22:35 1998, the temperature in San Francisco, CA US is 58 F.
```

这段代码有一个缺点，创建URL和提取温度的逻辑依赖于Web站点产生的HTML文件。如果某天该站点的图形设计员决定“current temp:”应该是大写，这个脚本就失效了。这只有等Web也采用更结构化的格式（如XML）时，用程序解析Web页的问题才能得以解决（注6）。

检查超链接的正确性和做Web镜像：Webchecker.py

维护一个Web站点的大麻烦之一是：随着站点里的超链接增加，一些链接失败的机会也增加了。好的Web站点维护会定期检查链接情况，Python标准版里有这样一个工具名为*Webchecker.py*，位于*tools/Webchecker*目录。

在同一个目录里还有一个*Websucker.py*，它能为远程的Web站点做本地拷贝。做的时候要小心，因为如果你不小心也许会把整个Web都下载到你的机器！在同一个目录里还有*wsgui.py*和*Webgui.py*这两个程序，它们分别是前面两个程序的基于Tkinter的前端程序。我们建议你看一看这些程序的源代码，学习如何用Python的标准工具建造复杂的Web管理系统。

注6：XML是一种标记结构化文本的语言，它强调了文档的结构，而不是图形特征。XML处理是Python文本处理的全新领域，正在开发中。参见附录一“Python资源”。

在 *tools/scripts* 目录里，你将发现很多其他也许是有趣的中小规模的脚本，例如与 *Websucker.py* 相似的 ftp 版本 *ftpmirror.py*。

检查邮件

在当今的 Internet 上最重要的媒介很可能是电子邮件，它肯定是个人之间传递大多数信息的协议。Python 有几个处理邮件的库。你需要用哪一个依赖于你使用的邮件服务器类型。其中包括 poplib 模块（POP3 服务器）和 imaplib 模块（IMAP 服务器）。如果你需要与微软的 Exchange 服务器对话，你将需要一些 win32 版的工具（参见附录二中 win32 扩展的 Web 地址）。

这里有一个 poplib 的简单测试，它与运行 POP 协议的服务器对话：

```
>>> from poplib import *
>>> server = POP3('mailserver.spam.org')
>>> print server.getwelcome()
+OK QUALCOMM Pop server derived from UCB (version 2.1.4-R3) at spam starting.
>>> server.user('da')
'+OK Password required for da.'
>>> server.pass_('youllneverguess')
'+OK da has 153 message(s) (458167 octets).'
>>> header, msg, octets = server.retr(152)  # 取最新的信息
>>> import string
>>> print string.join(msg[:3], '\n')   # 看前三行
Return-Path: <jim@bigbad.com>
Received: from gator.bigbad.com by mailserver.spam.org (4.1/SMI 4.1)
        id AA29605; Wed, 25 Nov 98 15:59:24 PST
```

在实际的应用里你需要使用特殊的模块，比如用 rfc822 来解析邮件首部，也许要用 mimetools 和 mimify 模块从邮件信息体里取数据（比如处理附件）。

较大的例子

计算你的复利

有时我们希望把一些钱存在银行帐号里，银行也很欢迎并愿意付利息。一般你的银行是根据你的存款数付给你利息，而且他们会把增加的利息加到你的总数里，每年你的存款会增加一些。这里是一个简单计算每年增长的 Python 程序：

```
trace = 1  # 打印每一年吗?

def calc(principal, interest, years):
    for y in range(years):
        principal = principal * (1.00 + (interest / 100.0))
        if trace: print y+1, '=> %.2f' % principal
    return principal
```

这个函数逐年地累积你的存款数(你的初始存款加上利息),它假设你不会中途取走钱。现在假设我们有¥65000,利息率为5.5%,而我们想知道十年后会增加多少。我们导入并调用我们的利息计算函数,参数是初始存款、利息率和计划存入的年数:

```
% python
>>> from interest import calc
>>> calc(65000, 5.5, 10)
1 => 68575.00
2 => 72346.63
3 => 76325.69
4 => 80523.60
5 => 84952.40
6 => 89624.78
7 => 94554.15
8 => 99754.62
9 => 105241.13
10 => 111029.39
111029.389793
```

而我们最后获得¥111029。如果我们只想知道最后的结果,我们可以在调用前把trace变量设为0:

```
>>> import interest
>>> interest.trace = 0
>>> calc(65000, 5.5, 10)
111029.389793
```

自然地有很多方式计算复合利息。例如不面是另一个例子,它逐年地打印出赚取的利息和存款数:

```
def calc(principal, interest, years):
    interest = interest / 100.0
    for y in range(years):
```

```
        earnings  = principal * interest
        principal = principal + earnings
        if trace: print y+1, '(+%d)' % earnings, '=> %.2f' % principal
    return principal
```

我们得到同样的结果，但信息更多：

```
>>> interest.trace = 1
>>> calc(65000, 5.5, 10)
1 (+3575) => 68575.00
2 (+3771) => 72346.63
3 (+3979) => 76325.69
4 (+4197) => 80523.60
5 (+4428) => 84952.40
6 (+4672) => 89624.78
7 (+4929) => 94554.15
8 (+5200) => 99754.62
9 (+5486) => 105241.13
10 (+5788) => 111029.39
111029.389793
```

对这个脚本的最后注释是，它也许与银行计算的不完全一样。银行的程序是精确到分，我们的程序在打印结果时也精确到分（`%.2f`的意思请参见第二章），但保留了计算机提供的计算结果的完整精度（见最后一行）。

自动拨号脚本

曾经有一个朋友在一个没有 Internet 连接的公司工作。但系统支持部门安装了一个拨号 modem，所以任何有 Internet 帐号并知道一点 Unix 的人就可以上网了。拨号时使用 Kermit 文件传送工具。

使用 modem 的一个缺点是，想拨出去的人不得不不断地尝试 10 个可用的 modem，直到找到一个空闲的。由于在 Unix 上可用文件名模式 */dev/modem** 访问 modem，用 */var/spool/locks/LCK*modem** 锁住 modem，所以只需要一个简单的 Python 脚本就可以自动的检查空闲的 modem。下面的程序 *dokermit.py* 使用了一个整数列表来跟踪锁住的 modem，`glob.glob`用来做文件名扩展。当找到空闲的 modem 后用 `os.system`运行一个 Kermit 命令：

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

```
# find a free modem to dial out on

import glob, os, string
LOCKS = "/var/spool/locks/"

locked = [0] * 10
for lockname in glob.glob(LOCKS + "LCK*modem*"): # 找锁住的modem
    print "Found lock:", lockname
    locked[string.atoi(lockname[-1])] = 1        # 0..9在名字末尾

print 'free: ',
for i in range(10):                              # 拨号报告
    if not locked[i]: print i,
print

for i in range(10):
    if not locked[i]:
        if raw_input("Try %d? " % i) == 'y':
            os.system("kermit -m hayes -l /dev/modem%d -b 19200 -S" % i)
            if raw_input("More? ") != 'y': break
```

按惯例，modem锁的文件名末尾是modem的号，我们就用这个信息来构造Kermit命令里的modem设备名。注意脚本里用了10个整数的列表来标识空闲的modem（1表示锁住）。这个程序只能用于10个modem以内的情况，如果有更多的modem，你需要用更大的列表和循环，并解析文件名，而不只是看最后一个字符。

一个交互式的朋友名单

尽管大多数前面的例子都用列表作主要的数据结构，但字典在很多方面更强大和更有趣。正是它的存在使得Python的层次较高，也就是说“容易用来对付复杂的问题”。对这个丰富的内置数据类型的一个重要的补充是一个扩展的标准库，一个强大的模块 cmd —— 它提供了一个 Cmd 类，你可以继承它来生成一个简单的命令行解释器。下面的例子有点大，但并不大复杂，它很好地说明了字典的威力和对标准模块的重用。

我们的任务是记录名字和电话号码，并为使用者提供交互式操作界面，提供错误检查和友好的在线帮助。下面演示了交互的过程：

```
% python rolo.py
Monty's Friends: help
```

```
Documented commands (type help <topic>):
========================================
EOF             add             find            list            load
save

Undocumented commands:
======================
help
```

我们可获得特定命令的帮助：

```
Monty's Friends: help find              # 与help_find()方法比较
Find an entry (specify a name)
```

我们可以简单的操作记录本里的记录：

```
Monty's Friends: add larry                  # 我们可以增加
Enter Phone Number for larry: 555-1216
Monty's Friends: add                        # 如果没有给出名字
Enter Name: tom                             # 程序会问
Enter Phone Number for tom: 555-1000
Monty's Friends: list
==========...=====================
               larry : 555-1216
                 tom : 555-1000
=  ...=============...=============
Monty's Friends: find larry
The number for larry is 555-1216.
Monty's Friends: save myNames               # 保存我们的工作
Monty's Friends: ^D                         # 退出程序(Windows上是^Z)
```

而最妙的是当我们重新启动程序时，我们可列出保存的数据：

```
% python rolo.py                        # 重启动
Monty's Friends: list                   # 缺省地，没有任何数据
Monty's Friends: load myNames           # 重新装入数据
Monty's Friends: list
==========...=====...=====...======
               larry : 555-1216
                 tom : 555-1000
=========...=====...=====...======
```

大多数的交互式解释器功能是由cmd模块的Cmd类提供的，我们只需要作一些定制，我们需要设置prompt属性并增加一些以do_和help_开头的方法。do_方法

必须有一个参数，而do_后面的部分就是命令的名字。一旦你调用了cmdloop()方法，一切都由Cmd类来完成了。读下面的程序*rolo.py*，一次读一个方法并与前面的输出作比较：

```
#!/usr/bin/env python
# 一个交互式的电话本

import string, sys, pickle, cmd

class Rolodex(cmd.Cmd):

    def __init__(self):
        cmd.Cmd.__init__(self)                  # 初始化基类
        self.prompt = "Monty's Friends: "       # 定制提示符prompt
        self.people = {}                        # 一开始我们谁都不认识

    def help_add(self):
        print "Adds an entry (specify a name)"
    def do_add(self, name):
        if name == "": name = raw_input("Enter Name: ")
        phone = raw_input("Enter Phone Number for "+ name+": ")
        self.people[name] = phone               # 增加对应于姓名的电话号码

    def help_find(self):
        print "Find an entry (specify a name)"
    def do_find(self, name):
        if name == "": name = raw_input("Enter Name: ")
        if self.people.has_key(name):
            print "The number for %s is %s." % (name, self.people[name])
        else:
            print "We have no record for %s." % (name,)

    def help_list(self):
        print "Prints the contents of the directory"
    def do_list(self, line):
        names = self.people.keys()              # 键是姓名
        if names == []: return                  # 如果没有姓名就退出
        names.sort()                            # 我们希望是字母排序
        print '='*41
        for name in names:
           print string.rjust(name, 20), ":"; string.ljust(self.people[name],
20)
        print '='*41
```

```
    def help_EOF(self):
        print "Quits the program"
    def do_EOF(self, line):
        sys.exit()
    def help_save(self):
        print "save the current state of affairs"
    def do_save(self, filename):
        if filename == "": filename = raw_input("Enter filename: ")
        saveFile = open(filename, 'w')
        pickle.dump(self.people, saveFile)

    def help_load(self):
        print "load a directory"
    def do_load(self, filename):
        if filename == "": filename = raw_input("Enter filename: ")
        saveFile = open(filename, 'r')
        self.people = pickle.load(saveFile)      # 注意这将覆盖
                                                 # 任何存在的人的记录

if __name__ == '__main__':                       # 这样模块也可以被别的程序导入
    rolo = Rolodex()
    rolo.cmdloop()
```

people变量只是姓名和电话号码之间的的映射，add和find方法使用这个映射。命令是以do_开头的方法，而它们的帮助是对应的help_方法。最后，load和save用了pickle模块，我们将在第十章作更详细的解释。

这个例子演示了扩展Python已有模块的威力。cmd模块负责处理提示符、帮助功能和解析输入。pickle模块做了所有保存和装入的工作。我们需要写的就是与我们的任务相关的部分。通用的交互式解释器部分是免费的。

练习

本章里充满了我们建议你尝试的例子，但如果你真的想练习，这里有一些更具挑战性的习题：

1. **重定向标准输出stdout。**修改脚本*mygrep.py*的输出到命令行上指定的最后一个文件，而不是屏幕。

请留意Cmd类是怎样工作的？

为了理解Cmd类是怎样工作的，请阅读已安装的Python标准库里的cmd模块。

Cmd解释器的onecmd()方法完成了我们感兴趣的工作的大部分，每次用户输入一行后就会调用它。这个方法从输入行中识别出第一个单词，然后它会查找Cmd的子类的实例中是否有对应的属性（如果命令是"find tom"，它会查找do_find属性）。如果它找到了对应的属性，就会用命令行的参数（这里是"tom"）调用对应方法并返回结果。onecmd()方法曾经是这样的（你的版本也许有所增加）：

```
# Cmd类的onecmd 方法，见 Lib/cmd.py
def onecmd(self, line):                 # 输入行例子 "find tom"
    line =   string.strip(line)         # 去掉多余的空白符
    if not line:                        # 如果没有输入
        line = self.lastcmd             # 重复上一次的命令
    else:
        self.lastcmd = line             # 为下 次保存命令
    i, n = 0, len(line)
                                        # 下一行作用是找出第 个单词尾
    while i < n and line[i] in self.identchars: i = i+1
                                        # 把输入分解为command + arguments
    cmd, arg = line[:i], string.strip(line[i:])
    if cmd == '':                       # 如果输入不是以A-z开始
        return self.default(line)
    else:                               # cmd是'find', line是'tom'
        try:
            func = getattr(self, 'do_' + cmd)  # 寻找方法
        except AttributeError:
            return self.default(line)
        return func(arg)                # 用输入行的剩余部分调用方法
```

2. **写一个简单的shell。**用cmd模块的Cmd类和第八章中操作文件和目录的函数写一个shell，它能解释标准Unix命令（或者DOS命令）：ls（dir）列出当前目录，cd改变当前目录，mv（ren）移动/改名一个文件，cp（copy）拷贝一个文件。

3. **理解map、reduce和filter。**如果你是第一次遇到这类函数，它们有点难以理解，部分原因是它们的参数有函数，而且这样一个小名字却干了很多工

作。一个确保你理解它们工作原理的好方法是重写它们，在这个练习里，写三个函数（map2、reduce2、filter2），分别做与map、reduce、filter同样的事情：

- map2有两个参数。第一个参数应当是一个函数，或者是None。第二个参数应该是一个序列。如果第一参数是一个函数，那么将用序列里的成员作为参数调用它，并把结果值以一个列表返回。如果第一参数是None，序列就被转换为一个列表并返回。
- reduce2有两个参数。第一个必须是一个有两个参数的函数，第二个必须是一个序列。序列里的前两个成员作为调用该函数的参数，返回的结果又作为新一次调用的第一参数，序列的第三个成员作为第二参数再次调用函数，依此类推，直到遍历了整个序列。最后一次调用的返回值就作为reduce2的返回值。
- filter2有两个参数。第一个可以是None或者是一个有两个参数的函数。第二个必须是一个序列。如果第一个是一个None，filter2就返回序列中为真值的成员。如果第一个是函数，filter2就依次用序列里的成员调用该函数，而最后将返回那些在调用中结果为真值的成员。

第十章

框架和应用

本章内容：

- 自动化客户支持系统
- 与COM的接口：廉价的公共关系
- 一个基于Tkinter的管理表格数据的编辑器
- 设计上的考虑
- JPython：Python与Java的结合
- 其他的框架和应用
- 练习

目前为止这本书里的所有例子都很小，与实际的应用相比它们似乎像玩具。本章展示了一些框架，它们可供那些想在特定的领域创建真正应用程序的Python程序员使用。一个框架可以看作是某个特定领域的一组类与期望的类之间的交互模式。我们将介绍三个：COM框架，也就是与微软的公共对象模型（Common Object Model，COM）的接口，Tkinter图形用户界面，以及Java的Swing图形界面工具库。我们也将涉及到标准库里与Web相关的模块。

假设有一个小公司的Web站点，它需要收集、维护和响应客户通过表格对产品的反馈，我们将以此来说明框架的威力。我们将涉及三个程序，第一个是基于Web的数据录入表格，要求用户从浏览器输入信息，然后把信息存到磁盘里。第二个程序是使用这些数据让Microsoft Word软件打印出定制的基于那些数据的信件。最后是一个浏览已存数据的浏览器，使用的是Tkinter模块。Tkinter是以一个强大的可移植的可以管理窗口、按钮、菜单等的图形界面工具集Tk为基础的。希望这些例子能让你认识到，当这些工具与Python的快速开发能力结合时，创建实际应用程序是多么的快。每一个程序都建筑在前一个的基础上，所以我们强烈建议你按顺序阅读每一个程序，即使你不能（或不想）让它们运行起来。

本章的最后一节谈到了JPython，它是Python在Java上的移植。本章以一个中等

程度的JPython程序结束，借助于Swing工具库，用户可以图形化地操作数学函数。

自动化客户支持系统

我们这个例子的主角是一个新创公司——乔氏牙膏公司，它销售最新型的牙膏。由于只有自己一个工作人员，乔无法应付太多的工作，所以创建了一个Web站点www.toftoot.com做推广和支持。网站上有各种漂亮的图片，也为顾客提供了一个输入抱怨或建议的地方。网页如图10-1所示。

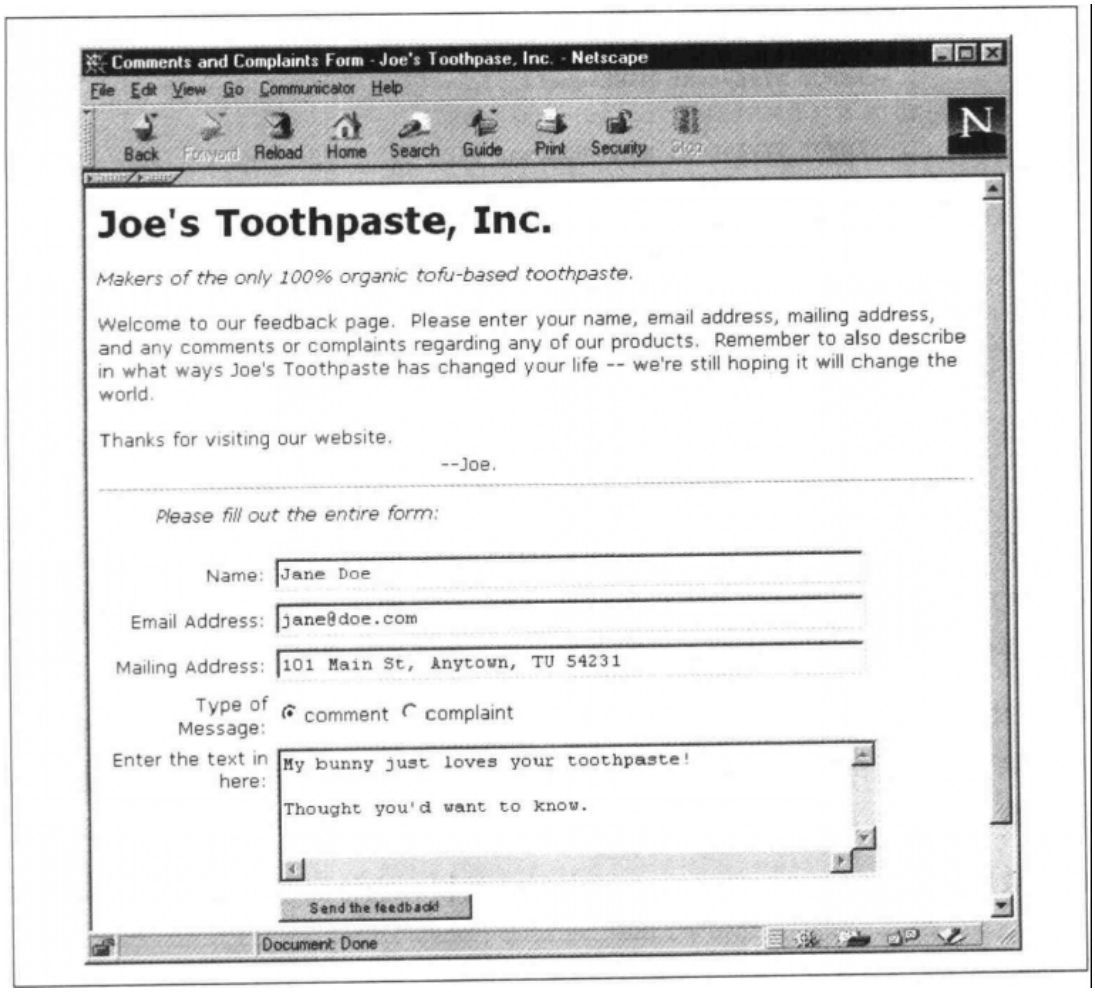


图10-1 顾客在http://www.toftoot.com/comment.html所看到的页

这一页关键的HTML代码显示在后面的选读部分“摘自HTML文件”中。由于本书不是讲CGI、HTML或其他技术的（注1），我们假设你已经知道这些技术。HTML代码里的重要部分用黑体字标出，这里是一个简短的描述：

摘自HTML文件

这是图10-1中网页的关键代码：

```
<FORM METHOD=POST ACTION="http://toftoot.com/cgi-bin/feedback.py">
  <UL><I>Please fill out the entire form:</I></UL>
  <CENTER><TABLE WIDTH="100%" >
  <TR><TD ALIGN=RIGHT WIDTH="20%">Name:</TD>
      <TD><INPUT TYPE=text NAME=name SIZE=50 VALUE=""></TD></TR>
  <TR><TD ALIGN=RIGHT>Email Address:</TD>
      <TD><INPUT TYPE=text NAME=email SIZE=50 VALUE=""></TD></TR>
  <TR><TD ALIGN=RIGHT>Mailing Address:</TD>
      <TD><INPUT TYPE=text NAME=address SIZE=50 VALUE=""></TD></TR>
  <TR><TD ALIGN=RIGHT>Type of Message:</TD>
  <TD><INPUT TYPE=radio NAME=type CHECKED VALUE=comment>comment 
      <INPUT TYPE=radio NAME=type VALUE=complaint>complaint</TD></TR>
  <TR><TD ALIGN=RIGHT VALIGN=TOP>Enter the text in here:</TD>
      <TD><TEXTAREA NAME=text ROWS=5, COLS=50  VALUE="">
          </TEXTAREA></TD></TR>
  <TR><TD></TD>
  <TD><INPUT type=submit name=send value="Send the feedback!"></TD></TR>
  </TABLE></CENTER>
  </FORM>
```

- FORM行指定了当链接到表格时将激活哪一个CGI程序，URL指向了我们将详细介绍的名为*feedback.py*的脚本。
- INPUT标签标出了表格字段的名字（姓名、地址、电子邮件、文本和类型）。除了类型是由用户选择的“建议”或“抱怨”以外，这些字段的值都是用户输入的。
- 最后，INPUT TYPE=SUBMIT标签是发送按钮，它将真正调用CGI脚本。

注1：如果你从来没有听说过这些缩写词，我可以解释一下：CGI表示公用网关接口，它是一个由Web浏览器调用服务器上程序的协议。HTML表示超文本标记语言，它是Web页的编码格式。

我们现在进入与Python相关的有趣部分：对请求的处理。这里是整个*feedback.py*程序：

```
import cgi, os, sys, string

def gush(data):
    print "Content-type: text/html\n"
    print "<h3>Thanks, %(name)s!</h3>" % vars(data)
    print "Our customer's comments are always appreciated."
    print "They drive our business directions, as well as"
    print "help us with our karma."
    print "<p>Thanks again for the feedback!<p>"
    print "And feel free to enter more comments if you wish."
    print "<p>"+10*" "+"--Joe."

def whimper(data):
    print "Content-type: text/html\n"
    print "<h3>Sorry, %(name)s!</h3>" % vars(data)
    print "We're very sorry to read that you had a complaint"
    print "regarding our product_ _We'll read your comments"
    print "carefully and will be in touch with you."
    print "<p>Nevertheless, thanks for the feedback.<p>"
    print "<p>"+10*" "+"--Joe."

def bail():
    print "<H3>Error filling out form</H3>"
    print "Please fill in all the fields in the form.<P>"
    print '<a href="http://localhost/comment.html">'
    print   o back to the form</a>'
    sys.exit(1)

class FormData:
    """A repository for information gleaned from a CGI form """
    def __init__(self, form):
        for fieldname in self.fieldnames:
            if not form.has_key(fieldname) or form[fieldname].value == "":
                bail()
            else:
                setattr(self, fieldname, form[fieldname].value)

class FeedbackData(FormData):
    """A FormData generated by the comment.html form. """
    fieldnames = ('name', 'address', 'email', 'type', 'text')
    def __repr__(self):
        return "%(type)s from %(name)s on %(time)s" % vars(self)
```

```
DIRECTORY = r'C:\complaintdir'

if __name__ == '__main__':
    sys.stderr = sys.stdout
    form = cgi.FieldStorage()
    data = FeedbackData(form)
    if data.type == 'comment':
        gush(data)
    else:
        whimper(data)

    # save the data to file
    import tempfile, pickle, time
    tempfile.tempdir = DIRECTORY
    data.time = time.asctime(time.localtime(time.time()))
    pickle.dump(data, open(tempfile.mktemp(), 'w'))
```

这个脚本的输出显然取决于输入，图10-1中的输入所产生的输出如图10-2所示。

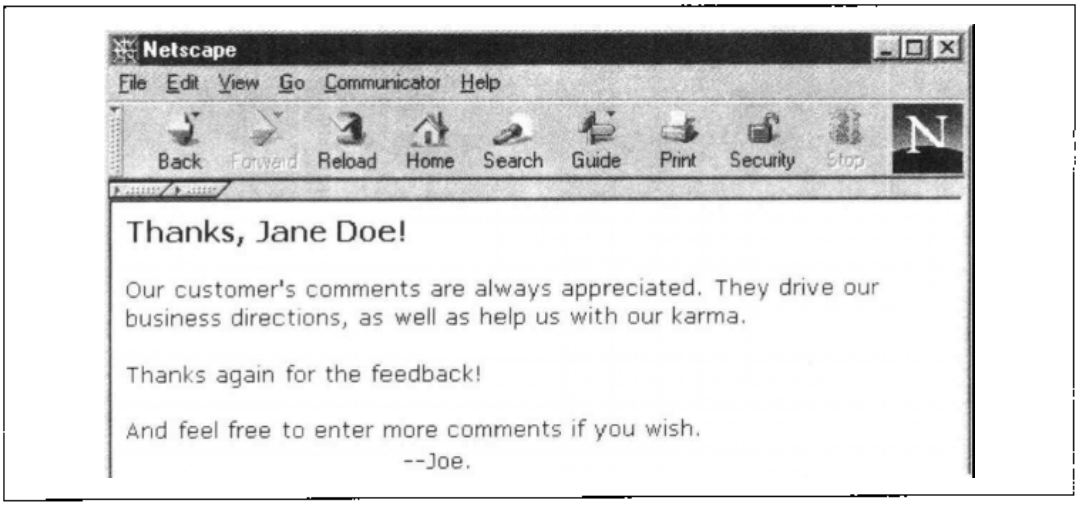


图10-2 用户按了“send the feedback（发送反馈）”按钮后所看到的页面

*feedback.py*是怎样工作的呢？这个脚本具有所有CGI程序常见的某些特点，都用黑体字标出。为了能启动，程序的第一行需要指向Python解释器。这是我们所用的Web服务器的要求，但也许你的环境要求不一样；即使是一样的，你的Python解释器的位置很可能不同。第二行导入包括cgi在内的模块，它负责处理CGI较难的部分，如解析环境变量和处理换义字符。（如果你从来没有手工的做这些事情，你是幸运的。）cgi模块的文档描述了它的直接而简单的使用方式。然而对于这个例子，脚本显得有点复杂。

让我们逐句地看 if __name__ == '__main__' 语句块的代码（注 2）。第一句把 sys.stderr 重定向到标准输出流。这是为了方便调试，因为CGI程序的标准输出流是 Web 浏览器，而 stderr 标准错误流是写到服务器的错误日志文件，阅读可能比看浏览器更容易一些。我们可以在浏览器上看到发生的运行错误，而不是去猜测发生了什么。第二行是重要的，它做了所有复杂的CGI的工作：它返回一个字典一样的对象（名为FieldStorge对象），键是表格里填写的变量名，可以通过它的 value 属性取到对应的值。听起来有些复杂，表格对象有键 'name'，'type'，'email'，'address' 和 'text'，通过 form['name'].value 可以找出用户在 Web 表格里填写的是什么。

第三行创建了一个用户定义的类FeedbackData的实例，把 form 对象作为参数传给它。如果你看 FeedbackData 的定义，你将发现它是另一个用户定义的 FormData 类的很简单的子类。我们在 FeedbackData 里定义了一个类属性 fieldnames，以及一个用于print语句的 __repr__ 函数。显然FormData类的 __init__ 方法一定会用到 FieldStorge 参数。实际上，它查看了实例里的 fieldnames类属性的每一个字段名字，对应于每个字段名字，检查在FieldStorge 对象里的对应非空键。如果有对应键，就在实例里设置同样名字的属性，把用户输入的文字作为属性的值；如果没有，就调用 bail 函数。

我们将很快就知道bail的作用，但首先让我们来看通常的情况：用户认真地输入了要求的所有数据。这时，FieldStorge 拥有所有 FeedbackData 所需的键（'name','type' 等等）。FormData类的__init__方法也就在实例里设置属性。所以当 data = FeedbackData(form) 调用返回时，将保证 data 是一个 FeedbackData的实例，它具有 name，type，email 等属性，以及有对应的用户输入的值。下面的代码也许有同样的效果：

```
form = cgi.FieldStorage()
form_ok = 1
if not form.has_key("name") or form["name"].value == "":
    form_ok = 0
else:
```

注 2：你要记住只有当这个程序作为脚本运行，而不是被导入时这个 if 语句才为真。CGI程序是一个脚本，所以当服务器调用它时将执行 if 语句块。我们在后面将导入它，所以要注意。

```
    data_name = form["name"].value
if not form.has_key("email") or form["email"].value == "":
    form_ok = 0
else:
    data_email = form["email"].value
...
```

但这种写法是冗长乏味而容易出错的。我们的计划是，当乔氏公司改变网页的域名时，我们需要改变的就是FeedbackData类的fieldnaes属性。我们也可以在别的CGI脚本里重用同样的FormData类。

如果用户没有输入所有要求的字段会怎样呢？要么是FieldStorge字典里少一个键，要么是对应的值是空字符串。FormData.__init__方法就会调用bail函数，它将礼貌地显示错误信息并退出脚本。控制永远不会返回主程序，所以没有必要测试data变量的有效性。

我们可检查data实例来查看用户反馈的是否是我们要感谢的建议。如果反馈的类型是抱怨，我们将毫不吝惜地道歉，并承诺与他们联系。

我们现在有了一个基本的CGI结构。把数据保存到文件非常简单。首先我们在if测试的外面定义DIRECTORY变量，因为我们将在另一个脚本里使用它，所以我们希望这个脚本不是单独运行时也定义这个变量。

浏览*feedback.py*的最后几行：

- 导入tempfile，pickle和time模块。tempfile模块如前一章所讲提供了未使用的文件名，我们不需要担心任何的名字冲突。pickle模块用来保存任何的Python对象。time模块告诉我们当前的时间，乔用它来判断反馈的时间。
- 下一行设置tempfile模块的tempdir属性为DIRECTORY变量的值，也就是我们希望保存数据的地方。这是一个通过直接修改名字空间来定制模块的例子，就像前面修改sys模块的stderr属性一样。
- 下一行用了几个time模块的函数，提供了当前日期和时间的字符串（如'Sat Jul 04 18:09:00 1998'，这对乔已经足够精确了），给data实例创建了一个新的属性time，它也随data一起保存。

- 最后一行作了实际的保存，它以写模式打开一个由tempfile模块产生文件名的文件，把实例数据写到文件里。工作完成了！现在指定的文件里保存了实例。

与COM的接口：廉价的公共关系

我们用保存的数据做两件事。我们将写一个周期性运行的程序（比如每晚2点）（注3）扫描保存的数据，找出哪些是客户的抱怨，并打印一份相应的信件。听起来很复杂，但你会惊讶于用恰当的工具来做是如此的简单。乔氏公司的网站运行在Windows上，所以我们假设这个程序也是在Windows上，但其他平台与此相似。

在我们讨论如何写这个程序前，先谈谈它使用的技术即微软的COM（Common Object Model）。COM是两个程序间交互的标准（用术语说是对象请求代理服务），使得COM兼容的程序可以互相通信，交换数据，在另一个COM兼容程序上执行命令。调用程序称为COM客户，而执行命令的一方称为COM服务器。所有的微软产品都是能运行COM的，几乎都能作为COM服务器。我们这里用的微软Word 8就是其中之一。事实上Word很适合于写信。幸运的是，至少在Windows 95、98和Windows NT上Python也能感知COM。Mark Hammond和Greg Stein为Python的Windows版做了扩展，名为*win32com*，这使得你的Python程序可以做COM相关的事情。你可以用Python写COM客户、服务器、ActiveX脚本等等。我们只需要写COM客户就行了，这也是最简单的。我们的格式信件程序需要做以下几件事：

1. 在适当的目录里打开文件并提取数据。
2. 对每个实例文件测试，反馈是否是抱怨。如果是就找出填表人的名字和地址，进入第3步；否则跳过第3步。
3. 打开一个我们想发出的Word信件文档模板，用相应的信息填写。

注3：设置这种自动定时程序在许多平台上都很简单，例如在Unix上用cron，在Windows NT上用AT。

4. 打印信件并关闭文档。

这非常简单。这里是称为*formletter.py*的小程序：

```
from win32com.client import constants, Dispatch
WORD = 'Word.Application.8'
False, True = 0, -1
import string

class Word:
    def __init__(self):
        self.app = Dispatch(WORD)
    def open(self, doc):
        self.app.Documents.Open(FileName=doc)
    def replace(self, source, target):
        self.app.Selection.HomeKey(Unit=constants.wdLine)
        find = self.app.Selection.Find
        find.Text = "%"+source+"%"
        self.app.Selection.Find.Execute()
        self.app.Selection.TypeText(Text=target)
    def printdoc(self):
        self.app.Application.PrintOut()
    def close(self):
        self.app.ActiveDocument.Close(SaveChanges=False)

def print_formletter(data):
    word.open(r"h:\David\Book\tofutemplate.doc")
    word.replace("name", data.name)
    word.replace("address", data.address)
    word.replace("firstname", string.split(data.name)[0])
    word.printdoc()
    word.close()

if __name__ == '__main__':
    import os, pickle
    from feedback import DIRECTORY, FormData, FeedbackData
    word = Word()
    for filename in os.listdir(DIRECTORY):
        data = pickle.load(open(os.path.join(DIRECTORY, filename)))
        if data.type == 'complaint':
            print "Printing letter for %(name)s." % vars(data)
            print_formletter(data)
        else:
            print "Got comment from %(name)s, skipping printing." % vars(data)
```

主程序的前几行显示了一个设计良好的框架的威力。第一行是一个标准的import语句，*win32com*是一个包而不是一个模块。它实际上是一组子包、模块和函数。我们需要其中的两个东西：client模块里的Dispatch函数，它使得我们可以把函数分发到别的对象（这里是指COM服务器），另一个是同一个模块里的constants子模块，它定义了我们需要的常量。

第二行定义我们感兴趣的COM服务器的名称变量。名字是Word.Application.8，你可以通过使用COM浏览器或读Word的API（应用程序接口）来找出该名字（参见后面的选读部分“了解COM接口”）。

现在我们集中看if __name__ == '__main__'这一块，也就是类和函数定义的下一行。

第一个任务是读数据。显然地我们需要os和pickle模块，以及我们刚写的feedback模块中的三项：DIRECTORY是存贮数据的地方（如果在*feedback.py*我们改变了它，这个模块下次运行时也会相应改变），以及FormData和FeedbackData类。下一行创建了Word类的实例，它打开了与Word这个COM服务器的连接，如果需要的话就启动Word。

for循环简单的遍历目录里保存的所有文件。这个目录里只保存反馈的数据，我们就不作任何错误检查。通常我们应当让代码更健壮，但为了简单我们忽略了。

for循环的第一行是提取数据。它用了pickle模块里的load函数，该函数的唯一参数是被提取的文件。它将返回文件里的所有数据——我们这里只有一个。现在，data里存的就是FeedbackData实例。在文件里并没有保存类的定义，而只是保存了实例的值和对类的引用（注4）。

循环里的if语句是一目了然的。剩下要解释的就是打印信件的函数，以及Word类。print_formletter函数用提取的数据调用了word实例的不同方法。

注4：这可以减少存储的对象的大小，而更重要的是它允许你解开以前版本的类，并自动地升级到新的类定义。

注意： 在解开时间的时候，解开的实例会自动地把对应类所定义的模块导入。为什么我们需要导入它呢？第五章“模块”里我们说过当前运行的模块名是__main__。换句话说，包含定义的类的模块名是__main__（即使文件名是*feedback.py*），而且当我们解开实例时导入__main__也就导入了当前运行模块（*formletter.py*），它没有包含解开的实例的类定义。这就是为什么我们需要显式地从feedback模块里导入类定义。如果在调用pickle.load时没有这些定义，解开实例会失败。或者我们可以把类定义放在一个文件里，并在任何实例前导入它，或者甚至更简单，把类定义放在由*feedback.py*导入的单独模块里，在*formletter.py*解开实例的过程中隐含地导入。后一种是通常的情况，你不必显式地导入类的定义，解开实例时就完成了，很神吧（注5）。

注意我们用string.split函数提取了客户的名，这是为了信件更友好，但如果是非标准的姓名就会有出错的危险。

在Word类里，__init__方法看起来简单却隐藏了许多工作。它创建了与COM服务器的连接，并存放在一个实例变量app里。现在，有两种使用服务器的方式：动态分发和非动态分发。如果是动态分发，Python在运行这个程序时不知道COM服务器的接口。不过没关系，因为COM允许Python询问服务器并确定协议，例如函数需要的参数个数和类型，然而这种方式可能会慢。一个加速的办法是运行*makepy.py*程序，它对指定的COM服务器做一次询问操作，并把信息存放到磁盘上。当一个是用该服务器的程序运行时，分发就用预先算好的信息而不是动态分发。我们这个程序用两种方法都可以，如果曾对Word运行过*makepy.py*，就会用快速地分发，否则就用动态分发。关于这个问题的更多信息，参见*win32*扩展，网址在：http://www.python.org/windows/win32all/。

为了解释Word类的方法，我们先打开一个文档模板，看看我们需要做什么。如图10-3所示。

你可以看出这是一个很平常的文档，除了一些文字在%符号之间以外。我们用这个标记告诉程序哪一部分需要定制，但也可用别的方法。要使用这个模板，我们需要打开它，定制，打印，最后关闭它。Word类的open方法用来打开它。打印

注5： 存储顶级类是很微妙的，这超出了本书的范围。我们提到它只是因为我们需要解释我们用的代码。


JOE'S TOOTHPASTE, INC

February 18, 1999

%name%
%address%

Dear %firstname%,

Thank you for filling out the feedback form at our website (www toftoot com) We're sorry to hear of your complaint, and we'll do our best to remedy it in the future. As a token of our deepest apologies, please accept a coupon for a free tube of our newest toothpaste, the "Firm Tofu Plus Toothpaste", enriched with Vitamin A, B and Z12.

Sincerely,

Joe Smith

FREE COUPON

Coupon for 1 (one) free tube of Firm Tofu Plus Toothpaste (5oz)

Available at all WackyStores™

Expires: 12/99

SUITE 101, 22 MAIN ST • TOFUTOWN, CA • 04122
PHONE 415-555-1212 • FAX. 415-555-1211

图 10-3 乔氏公司给抱怨者的信件模板

与关闭也类似。定制时我们用相应的字符串替代 %name%，%firstname% 以及 %address%，这是由 replace 方法完成的。

让这一切运转起来，产生的输出文本如下：

```
C:\Programs> python formletter.py
Printing letter for John Doe.
Got comment from Your Mom, skipping printing.
Printing letter for Susan B. Anthony.
```

打印了两份定制的准备邮寄的信件。注意 Word 程序在桌面上并不显示，缺省情况下 COM 服务器是不可见的，所以 Word 只是在后台行动。如果此时 Word 运行在桌面上，每一步用户都可以看到。

请留意：了解COM接口

你怎样找出COM对象的各种方法和属性呢？一般来说，COM对象与别的程序一样应该有文档。然而很可能我们有软件但没有文档。如果你想了解COM接口，有三种可能的方法：

- 寻找并购买文档，一些COM程序在Web上有文档，或者是印刷好的。
- 使用COM浏览器。Pythonwin（见附录二）就带有一个COM浏览器可用于探索COM对象的复杂等级体系。它也就是列出可用的对象与函数，有时这也就是你需要的。微软Visual Studio这样的开发工具也带有COM浏览器。
- 用另一个工具。在上面的例子中，我们用Word的“宏记录器”功能产生一个VBA脚本，它可以相当直接地翻译成Python脚本。宏可以找出文件菜单里的选择打印项的等效命令是：`ActiveDocument.PrintOut()`。

一个基于Tkinter的管理表格数据的编辑器

我们回顾一下：我们写了一个CGI程序（*feedback.py*），它从Web上获得输入并把信息保存在我们服务器的磁盘上。我们又写了一个程序（*formletter.py*），用这些信息产生了道歉的信件。下一个任务就是建造一个由人来查看建议和抱怨的程序，使用Tkinter工具库来建造一个反馈信息的浏览器。

Tkinter工具库是一个Tk图形库的Python接口。Tk事实上是大多数Python程序员的选择，因为它提供了职业的图形界面并且相当容易使用。它产生的界面与Windows、Mac或任何Unix都不完全一样，但看起来很接近，而且同样的Python程序在所有平台都可以工作，这对那些特定平台的工具来说是不可能的。另一个值得考虑的可移植工具包是wxPython。它位于：http://www.alldunn.com/wxPython。

我们将在这个例子中用Tk。它是由John Ousterhout开发的另一个脚本语言Tcl的工具包。Tk已经被许多别的脚本语言采用，包括Python和Perl。更多的关于Perl和Tk的信息请参见O'Reilly出版的《Learning Perl/Tk》。

这个程序的目标是简单的：在一个窗口中显示列出所有的反馈数据项，允许用户选择并查看细节。而且乔希望能删除处理过的项。实际运行的该程序的屏幕图如图 10-4 所示。

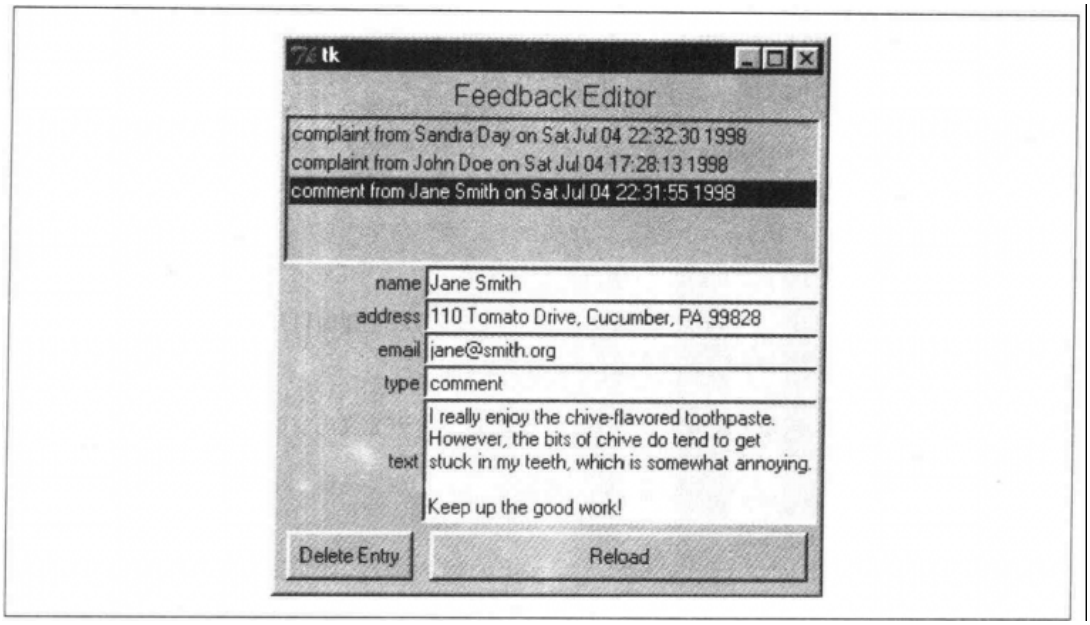


图 10-4　feedbackeditor.py 程序的屏幕图

我们将彻底地考察一种编程方式。我们的程序称为*feedbackeditor.py*，如下：

```
from FormEditor import FormEditor
from feedback import FeedbackData, FormData
from Tkinter import mainloop
FormEditor("Feedback Editor", FeedbackData, r"c:\Complaintdir")
mainloop()
```

更多的细节都隐藏在*FormEditor*里。把这几行单独出来的意义在于，我们把与表格有关的部分独立出来，*FormEditor*程序完全独立于特定的表格。另外，这个小程序展示了如何与Tkinter接口，你创建组件和窗口，然后调用*mainloop*函数启动GUI运行。接下来发生的就是GUI动作的结果。与`formletter`一样，它从`feedback`模块导入了类，道理是一样的(解开实例)。然后创建了一个`FormEditor`的实例，传递给初始化函数的参数是编辑器的名字，要解开的实例的类，以及要解开的实例的位置。

FormEditor的代码只有一个类的定义，我们将一次说明一个方法。首先是import语句和初始化方法：

```
from Tkinter import *
import string, os, pickle

class FormEditor:
    def __init__(self, name, dataclass, storagedir):
        self.storagedir = storagedir        # 保存一些引用
        self.dataclass = dataclass
        self.row = 0
        self.current = None

        self.root = root = Tk()             # 创建窗口并设置大小
        root.minsize(300,200)

        root.rowconfigure(0, weight=1)      # 定义当窗口调整尺寸时列和行的尺度
        root.columnconfigure(0, weight=1)
        root.columnconfigure(1, weight=2)
        # 创建标题标签
        Label(root, text=name, font='bold').grid(columnspan=2)
        self.row = self.row + 1
        # 创建主列表框并设置它
        self.listbox = Listbox(root, selectmode=SINGLE)
        self.listbox.grid(columnspan=2,sticky=E+W+N+S)
        self.listbox.bind('<ButtonRelease-1>', self.select)
        self.row = self.row + 1

    # 每个类的fieldnames变量里的变量调用一次self.add_variable
        for fieldname in dataclass.fieldnames:
            setattr(self, fieldname, self.add_variable(root, fieldname))

    # 创建几个按钮
        self.add_button(self.root, self.row, 0, 'Delete Entry', self.delentry)
        self.add_button(self.root, self.row, 1, 'Reload', self.load_data)

        self.load_data()
```

我们用了from ... import *的写法，我们曾警告过这有时是危险的。在Tkinter程序里，这通常是很安全的，因为Tkinter只输出显然与GUI相关的变量（标签、按钮等），而且都是以大写字母开头。

要理解__init__方法最好是把代码结构与屏幕结构对照。它们几乎是完全一致的。

前几行只是保存实例变量里的东西，并为一些变量设置缺省值。接下来的几行访问一个Toplevel组件（一个窗口，Tk()调用返回当前定义的顶层组件），设置它的最小值和一些别的属性。行和列的设置使得窗口里的组件随着窗口的放大而变化，并决定内部两列组件的相对宽度。

下一个调用创建了一个在Tkinter模块里定义的标签组件，你可以从屏幕看到它也就是一个文本标签。它横跨两列，也就是说它从窗口的最左边伸展到最右边。GUI调用的主要职责就是安排图形元素的位置。

接下来创建的是列表框组件。它是一个文本行的列表，用户可以用方向键和鼠标做选择。这个列表框只允许一次选择一行（selectmode=SINGLE）并填充所有的空间（sticky选项）。

for循环是这个方法里最有趣的部分，通过遍历dataclass变量（也就是FeedbackData）的fieldnames属性，它找出哪一个变量在实例的数据里，并依次调用FormEditor类的add_variable方法，把返回值放到一个对应的实例变量里。这也等同于：

```
    ...
self.name    self.add_variable(root, 'name')
self.email = self.add_variable(root, 'email')
self.address = self.add_variable(root, 'address')
self.type = self.add_variable(root, 'type')
self.text = self.add_variable(root, 'text')
```

不过代码例子里的版本更好些，因为字段名列表对程序是已知的，重复键入常常是不良设计的标志。而且对于我们的表格来说，FormData没有什么是特殊的。它可以用来浏览任何可变化的字段名的类。像这样的通用程序可以更好地重用在别的任务中。

在这个方法的结束部分，我们看到两个按钮，每一个都有一个被单击时要执行的命令。一个是删除当前项的delentry方法，另一个用于重读字典里数据的reloading函数。

最后用load_data方法来装入数据。在对设置组件的add_varible和add_button方法的介绍结束后，我们再来介绍它。

add_varible创建了一个标签组件，显示字段的名字，同一行里还有一个标签对应于列表框里选项的值：

```
def add_variable(self, root, varname):
    Label(root, text=varname).grid(row=self.row, column=0, sticky=E)
    value = Label(root, text='', background='gray90',
                  relief=SUNKEN, anchor=W, justify=LEFT)
    value.grid(row=self.row, column=1, sticky=E+W)
    self.row = self.row + 1
    return value
```

add_button更简单些，它只需要创建一个组件：

```
def add_button(self, root, row, column, text, command):
    button = Button(root, text=text, command=command)
    button.grid(row=row, column=column, sticky=E+W, padx=5, pady=5)
```

load_data函数清除列表框的任何内容并重新设置item属性。循环与*printcomplaints.py*里用的很像，说明如下：

- 保存实例的文件名，被作为实例的一个属性（我们将很快知道为什么）
- 实例被加入item属性
- 项对应的字符串（注意反引号`的用法）被加入列表框
- 最后，选择列表框的第一项：

```
def load_data(self):
    self.listbox.delete(0,END)
    self.items = []
    for filename in os.listdir(self.storagedir):
        item = pickle.load(open(os.path.join(self.storagedir, filename)))
        item._filename = filename
        self.items.append(item)
        self.listbox.insert('end', `item`)
    self.listbox.select_set(0)
    self.select(None)
```

我们现在介绍前面提到的select方法。有两种情况会调用它，第一种就像上面的方法发生在数据装入完毕，第二种就是__init__方法里的捆绑操作，我们重新写在下面：

```
self.listbox.bind('<ButtonRelease-1>', self.select)
```

这个调用把一个特定组件的特定事件（'<ButtonRelease-1>'）与一个称为self.select捆绑起来。换句话说，每当你的鼠标落在列表框的一项上时，就会调用编辑器的select方法。调用时的参数是事件类型，使我们知道是哪一个按钮按下了，但由于我们只需要知道事件发生了，所以我们将忽略这个参数。选择时应发生什么？必须识别出对应于被选择项的实例，然后对应的显示字段必须更新。这是通过遍历每个字段名（参考fieldname变量），找出选择的实例的对应值，并设置对应的标签（注6）：

```
def select(self, event):
    selection = self.listbox.curselection()
    self.selection = self.items[int(selection[0])]
    for fieldname in self.dataclass.fieldnames:
        label = getattr(self, fieldname)                    # GUI字段
        labelstr = getattr(self.selection, fieldname)       # 实例的属性
        labelstr = string.replace(labelstr,'\r', '')
        label.config(text=labelstr)
```

load_data方法也就是我们需要的重新装入数据的功能，所以把它与reload（重新装入）按钮绑定。然而，删除一项是很简单的。我们曾提到，当我们从磁盘装入实例时做的第一件事就是记录对应的文件名，这里我们就用这个信息来删除文件，然后重新装入数据，列表框就自动地更新了：

```
def delentry(self):
    os.remove(os.path.join(self.storagedir,self.selection._filename))
    self.load_data()
```

这个程序很可能是本书里最难理解的一个，因为它大量地使用了复杂而强大的Tkinter库。Tk和Tkinter都有文档。

设计上的考虑

可以把CGI脚本*feedback.py*和GUI程序FormEditor看成操作同一个数据集（存

注6：　需要replace操作是因为Tk把Windows的\r\n序列看作两个回车。

Tkinter 文档

Tkinter 的文档正变得越来越好。Tkinter 原来是 Steen Lumholdt 写的，当时他在用Python时觉得需要一个图形界面。不过他没有写太多的文档。Tkinter 已经更新了多次，主要都是 Guido van Rossum 做的。Tkinter 的文档还是不够完整。不过，还是有一些文档可用，到你读到这里时，也许会有更多的文档。

- 最完整的文档是 Fredrik Lundh 的，可在下面的 Web 地址找到：http://www.pythonware.com/library/tkinter/introduction/index.htm
- 一个较早的但仍然有用的称为“Matt Conway 的救命稻草”的文档的网址是：http://www.python.org/doc/life-preserver/index.html
- 《Programming Python》也有关于 Tkinter 的文档，特别是第于一、第十二和第十六章。
- 可能有更多的：参见 python.org 网站的 Tkinter 部分：http://www.python.org/topics/tkinter/。

在磁盘上的实例）的不同方式。什么时候应该用基于Web的接口？什么时候应使用一个普通的GUI接口？选择应基于几个因素：

- 用一个框架实现需要的功能是否容易？
- 你可以要求用户安装什么软件来访问或修改数据？

Web 界面很适合于这样的情况，数据操作的复杂性较低，并且方便用户比功能齐全更重要。另一方面，用一个图形界面库建造的实际程序有最大的灵活性，代价是必须教用户安装使用专门的程序。Python在有经验的程序员中成功的原因之一，就是允许他们选择编程框架，而不是强迫他们使用某种语言设计者头脑里的框架。使用浏览器做界面开发一个全功能的应用也是可能的。Zope就是这样一个框架，可以免费从Digital Creations得到（遵循一种开源许可证）。如果你对开发基于 Web 的成熟程序感兴趣，试一试 Zope（参见附录一“Python 资源”）。

JPython：Python 和 Java 的结合

JPython是最近由Jim Hugunin用Java写的Python。这对于Python社团和Java社团来说都是令人激动的进展。Python的用户很高兴他们的Python知识可以转移到基于Java的开发环境，Java程序员很高兴能够用Python脚本语言作为控制Java系统的一种方式，通过一个解释环境来测试和学习Java库。

JPython可以从http://www.jpython.org/得到，与CPython（为区分JPython而使用对Python本身的称呼）的许可条款相似。

JPython安装包括几个部分：

- JPython：等价于我们书中用的Python。
- JPythonc：把JPython程序编译成Java类文件。结果类文件可用在任何Java类文件可用的地方，例如作为applet，Servlet或bean。
- 标准库里面的大多数模块。
- 一些演示JPython编程的程序。

使用JPython与使用Python十分相似：

```
~/book> jpython
JPython 1.0.3 on Java1.2beta4
Copyright 1997 1998 Corporation for National Research Initiatives
>>> 2 + 3
5
```

实际上，JPython与CPython看起来几乎完全一样。两者区别的最新列表可以访问http://www.jpython.org/docs/differences.html，最重要的区别是：

- JPython比CPython慢。慢多少依赖于测试的代码以及使用的Java虚拟机。但是，JPython的作者承诺会进行优化，将来的版本会与Python一样快。
- JPython里缺少一些内置成分和库模块。例如os.system()调用就没有实现，原因是在Java与操作系统的交互下这样做很困难。一些大的扩展模块如Tkinter图形界面框架没有实现，因为基础工具（如Tk/Tcl）在Java里没有。

JPython 使 Python 程序员可以访问 Java 库

JPython 与 CPython 最重要的区别是，JPython 使 Python 程序员可以无缝地访问 Java 库。考虑下面的程序 *jpythondemo.py*，输出显示在图 10-5。

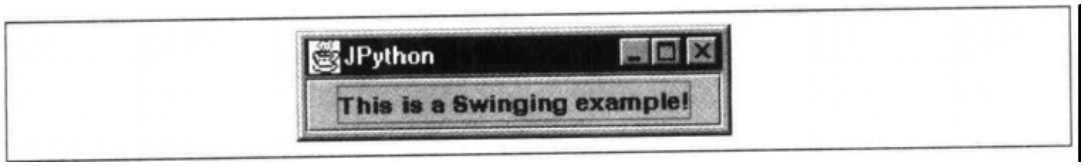


图 10-5 jpythondemo.py 的输出

```
from pawt import swing
import java

def exit(e): java.lang.System.exit(0)

frame = swing.JFrame('Swing Example', visible=1)
button = swing.JButton('This is a Swinging button!', actionPerformed=exit)
frame.contentPane.add(button)
frame.pack()
```

这个程序演示了一个使用 Java Swing GUI（注 7）框架的 Python 程序，它非常简单。第一行导入了 Java 的 swing 包（pawt 模块计算 swing 包的准确位置，可能是 java.awt.swing，或 com.sun.java.swing，或 javax.swing）。第二行导入另一个 Java 包，我们需要它的 java.lang.System.exit() 调用。第四行创建了一个 JFrame，把它的 visible 属性设为真。第五行创建了一个 JButton，指定了相应的标签和命令。最后两行把 JButton 放到 JFrame 里，并把它们都设为可见的。

有经验的 Java 程序员也许对 *jpythondemo.py* 里的部分代码感到意外，因为它与等价的 Java 程序有些不同。为了使 Python 程序员尽可能地容易使用 Java 库，JPython 在幕后做了许多工作。例如，当 JPython 导入一个 Java 包时，它积极地跟踪相应的包，然后使用 Java 的反射（reflection）API 找到包的内容，以及相应的类和方法。JPython 也会在 Python 和 Java 的数据类型之间作转换。例如在 jpythondemo.py 里，按钮的文字（'This is a Swinging example!'）是一个 Python 字符串。

注 7：Swing 和 Java 基础类的文档可参考 http://java.sun.com/products/jfc/index.html。或者参考 O'Reilly 出版的《Java Swing》。

在调用JButton的构造函数前，JPython查看可以用哪一个构造函数（比如不会使用第一个参数是Icon的），并自动地把Python字符串转换为Java字符串。更复杂的机制使得我们可在JButton的构造函数里方便地用actionPerformed=exit这个关键字参数。这个惯用法在Java里是不可能的，因为Java不能把函数（或方法）当作对象来操作。JPython里就不必创建只有一个方法actionPerformed的ActionListener类，当然如果你喜欢烦琐的用法，这也是可以的。

JPython: Java 的脚本语言

JPython正变得流行起来，因为它可以帮助程序员探索大量正在出现的Java库，而且是在交互式的、快捷的环境里。而且实践证明把Python作为脚本语言嵌入到Java框架里也是有用的，可以让最终用户（不是开发人员）做定制、测试以及别的编程任务。在JPython的目录*demo/embed*里有一个把Python解释器嵌入Java程序的例子。

一个真正的 JPython/Swing 应用：grapher.py

*grapher.py*程序（输出显示在图10-6中）允许用户图形化地探索数学函数的行为。它也是基于Swing GUI工具库的。有两个输入Python代码的文本输入框。第一个是在画出函数前引用的任意的Python程序，通过它导入需要的模块，定义也许会用到的函数。第二个输入框（名为Expression:）应该是一个Python的表达式（比如sin(x)），而不是一个语句。依次用每个数据点调用它（变量x的值设为水平坐标）。

用户可以选择是画一个线图还是填充图，以及画的点数、颜色。最后，用户可以把设置保存到磁盘上，以便以后再装入使用（用pickle模块）。下面就是*grapher.py*程序：

```
from pawt import swing, awt, colors, GridBag
RIGHT = swing.JLabel.RIGHT
APPROVE_OPTION = swing.JFileChooser.APPROVE_OPTION
import java.io
import pickle, os

default_setup = """from math import *
```

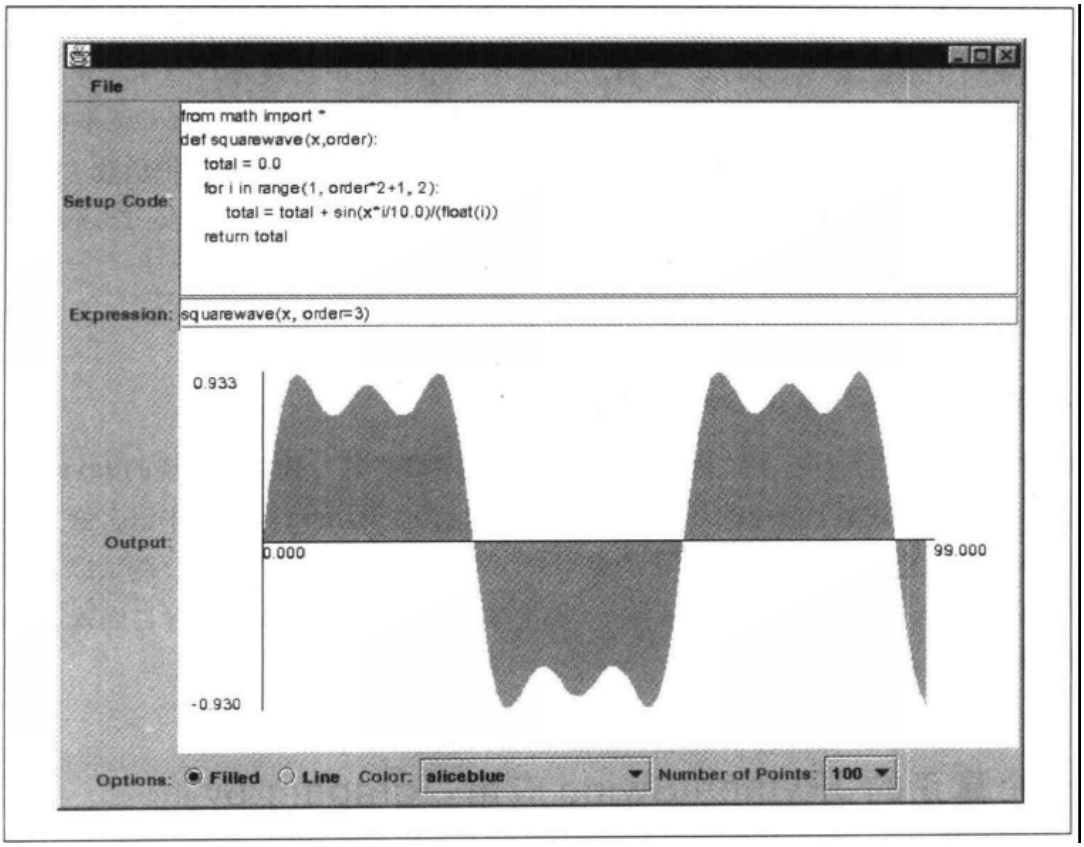


图 10-6 grapher.py 的输出

```
def squarewave(x,order):
    total = 0.0
    for i in range(1, order*2+1, 2):
        total = total + sin(x*i/10.0)/(float(i))
    return total
"""
default_expression = "squarewave(x, order=3)"

class Chart(awt.Canvas):
    color = colors.darkturquoise
    style = 'Filled'

    def getPreferredSize(self):
        return awt.Dimension(600,300)

    def paint(self, graphics):
        clip = self.bounds
        graphics.color = colors.white
```

```
        graphics.fillRect(0, 0, clip.width, clip.height)

        width = int(clip.width * .8)
        height = int(clip.height * .8)
        x_offset = int(clip.width * .1)
        y_offset = clip.height - int(clip.height * .1)

        N = len(self.data); xs = [0]*N; ys = [0]*N

        xmin, xmax = 0, N-1
        ymax = max(self.data)
        ymin = min(self.data)

        zero_y = y_offset - int(-ymin/(ymax-ymin)*height)
        zero_x = x_offset + int(-xmin/(xmax-xmin)*width)

        for i in range(N):
            xs[i] = int(float(i)*width/N) + x_offset
            ys[i] = y_offset - int((self.data[i]-ymin)/(ymax-ymin)*height)
        graphics.color = self.color
        if self.style == "Line":
            graphics.drawPolyline(xs, ys, len(xs))
        else:
            xs.insert(0, xs[0]); ys.insert(0, zero_y)
            xs.append(xs[-1]); ys.append(zero_y)
            graphics.fillPolygon(xs, ys, len(xs))

        # draw axes
        graphics.color = colors.black
        graphics.drawLine(x_offset,zero_y, x_offset+width, zero_y)
        graphics.drawLine(zero_x, y_offset, zero_x, y_offset-height);

        # draw labels
        leading = graphics.font.size
        graphics.drawString("%.3f" % xmin, x_offset, zero_y+leading)
        graphics.drawString("%.3f" % xmax, x_offset+width, zero_y+leading)
        graphics.drawString("%.3f" % ymin, zero_x-50, y_offset)
        graphics.drawString("%.3f" % ymax, zero_x-50, y_offset-height+leading)

class GUI:
    def __init__(self):
        self.numelements = 100
        self.frame = swing.JFrame(windowClosing=self.do_quit)

        # build menu bar
        menubar = swing.JMenuBar()
```

```
file = swing.JMenu("File")
file.add(swing.JMenuItem("Load", actionPerformed = self.do_load))
file.add(swing.JMenuItem("Save", actionPerformed = self.do_save))
file.add(swing.JMenuItem("Quit", actionPerformed = self.do_quit))
menubar.add(file)
self.frame.JMenuBar = menubar

# create widgets
self.chart = Chart(visible=1)
self.execentry = swing.JTextArea(default_setup, 8, 60)
self.evalentry = swing.JTextField(default_expression,
                                  actionPerformed = self.update)

# create options panel
optionsPanel = swing.JPanel(awt.FlowLayout(
    alignment=awt.FlowLayout.LEFT))

# whether the plot is a line graph or a filled graph
self.filled = swing.JRadioButton("Filled",
                                 actionPerformed=self.set_filled)
optionsPanel.add(self.filled)
self.line = swing.JRadioButton("Line",
                               actionPerformed=self.set_line)
optionsPanel.add(self.line)
styleGroup = swing.ButtonGroup()
styleGroup.add(self.filled)
styleGroup.add(self.line)

# color selection
optionsPanel.add(swing.JLabel("Color:", RIGHT))
colorlist = filter(lambda x: x[0] != '_', dir(colors))
self.colorname = swing.JComboBox(colorlist)
self.colorname.itemStateChanged = self.set_color
optionsPanel.add(self.colorname)

# number of points
optionsPanel.add(swing.JLabel("Number of Points:", RIGHT))
self.sizes = [50, 100, 200, 500]
self.numpoints = swing.JComboBox(self.sizes)
self.numpoints.selectedIndex = self.sizes.index(self.numelements)
self.numpoints.itemStateChanged = self.set_numpoints
optionsPanel.add(self.numpoints)

# do the rest of the layout in a GridBag
self.do_layout(optionsPanel)
```

```
def do_layout(self, optionsPanel):
    bag = GridBag(self.frame.contentPane, fill='BOTH',
                  weightx=1.0, weighty=1.0)
    bag.add(swing.JLabel("Setup Code: ", RIGHT))
    bag.addRow(swing.JScrollPane(self.execentry), weighty=10.0)
    bag.add(swing.JLabel("Expression: ", RIGHT))
    bag.addRow(self.evalentry, weighty=2.0)
    bag.add(swing.JLabel("Output: ", RIGHT))
    bag.addRow(self.chart, weighty=20.0)
    bag.add(swing.JLabel("Options: ", RIGHT))
    bag.addRow(optionsPanel, weighty=2.0)
    self.update(None)
    self.frame.visible = 1
    self.frame.size = self.frame.getPreferredSize()

    self.chooser = swing.JFileChooser()
    self.chooser.currentDirectory = java.io.File(os.getcwd())

def do_save(self, event=None):
    self.chooser.rescanCurrentDirectory()
    returnVal = self.chooser.showSaveDialog(self.frame)
    if returnVal == APPROVE_OPTION:
        object = (self.execentry.text,  self.evalentry.text,
                  self.chart.style,
                  self.chart.color.RGB,
                  self.colorname.selectedIndex,
                  self.numelements)
        file = open(os.path.join(self.chooser.currentDirectory.path,
                    self.chooser.selectedFile.name), 'w')
        pickle.dump(object, file)
        file.close()

def do_load(self, event=None):
    self.chooser.rescanCurrentDirectory()
    returnVal = self.chooser.showOpenDialog(self.frame)
    if returnVal == APPROVE_OPTION:
        file = open(os.path.join(self.chooser.currentDirectory.path,
                self.chooser.selectedFile.name))
        (setup, each, style, color,
         colorname, self.numelements) = pickle.load(file)
        file.close()
        self.chart.color = java.awt.Color(color)
        self.colorname.selectedIndex = colorname
        self.chart.style = style
        self.execentry.text = setup
```

```
            self.numpoints.selectedIndex = self.sizes.index(self.numelements)
            self.evalentry.text = each
            self.update(None)

    def do_quit(self, event=None):
        import sys
        sys.exit(0)

    def set_color(self, event):
        self.chart.color = getattr(colors, event.item)
        self.chart.repaint()

    def set_numpoints(self, event):
        self.numelements = event.item
        self.update(None)

    def set_filled(self, event):
        self.chart.style = 'Filled'
        self.chart.repaint()

    def set_line(self, event):
        self.chart.style = 'Line'
        self.chart.repaint()

    def update(self, event):
        context = {}
        exec self.execentry.text in context
        each = compile(self.evalentry.text, '<input>', 'eval')
        numbers = [0]*self.numelements
        for x in xrange(self.numelements):
            context['X'] = float(x)
            numbers[x] = eval(each, context)
        self.chart.data = numbers
        if self.chart.style == 'Line':
            self.line.setSelected(1)
        else:
            self.filled.setSelected(1)
        self.chart.repaint()
GUI()
```

这个程序的逻辑是相当简单直接的，类和方法的名字使我们很容易跟随控制流。这个程序的大部分可以用类似的Java代码来写（但要大得多）。粗体的部分显示了使用Python的威力：在模块的最前面定义了Setup和Expression这两个文本框的缺省值。前者从math模块里导入了函数，并定义了一个squarewave函数。

后者指定了一个对该函数的调用，有一个order参数（随着该参数的增加，画出的图越来越像一个方波，所以函数取名为squarewave，即方波）。如果你安装了Java、Swing和JPython，不妨用其他的Setup和Expression来玩一玩。

这个例子里用Python取代Java的关键是在update方法里：它简单地调用标准的Python的exec语句，以Setup里的代码作为参数，然后用编译好的Expression代码，循环地对每个坐标调用eval函数。用户可以在这个文本框里自由地使用其他的Python函数或表达式！

JPython仍然是一项正在进行的工作：Jim Hugunin不断地改进和优化Python和Java之间的接口。作为Python的另一个实现，JPython也影响着Guido van Rossum对Python语言核心的抉择。

其他的框架和应用

由于篇幅有限，我们只能介绍与Python相关的几个最流行的框架。另外还有几个值得介绍的（也许会很有用）框架，我们简短地介绍如下：

Python图像库（PIL）

Python图像库是一个Fredrik Lundh写的大型框架，用于创建、操作、转换和保存各种格式的位图（如GIF、JPEG、PNG）。它有Tk和Pythonwin的接口，使得我们可以用Tk或Pythonwin代码显示PIL产生的图像。PIL的网址在：http://www.pythonware.com。

数字Python（NumPy）

数字Python是Python的一组扩展，可快速而漂亮地操作较大的数字数组，它是由Jim Hugunin（JPython的作者）写的，得到了Matrix-SIG（SIG参见附录一）的订阅者的支持。由于Jim开始了JPython的工作，数字Python已经由Lawrence Livermore国家实验室的人接管了。NumPy是科学家和工程师喜欢的一个非常强大的工具。更多的信息可以在Python主站点找到（http://www.python.org/topics/scicomp/）。

下面是一个典型的NumPy的例子 *numpytest.py*，以及它产生的数据：

```
from Numeric import *
coords = arange(-6, 6, .02)                    # 创建一组坐标
xs = sin(coords)                               # 取得x坐标的正弦值
ys = cos(coords)*exp(-coords*coords/18.0)
zx = xs * ys[:,NewAxis]                        # x行乘以y列
```

如果你还记得你学过的数学，你也许算出xs是一个-6到6之间的数的正弦值数组，ys是同一组数的余弦值（乘以一个指数函数）数组。zs是这两个数组的外积。如果你想知道结果会是什么样，你可以把数组转换为一个图像（比如用上述的PIL），图像可参考图10-7。

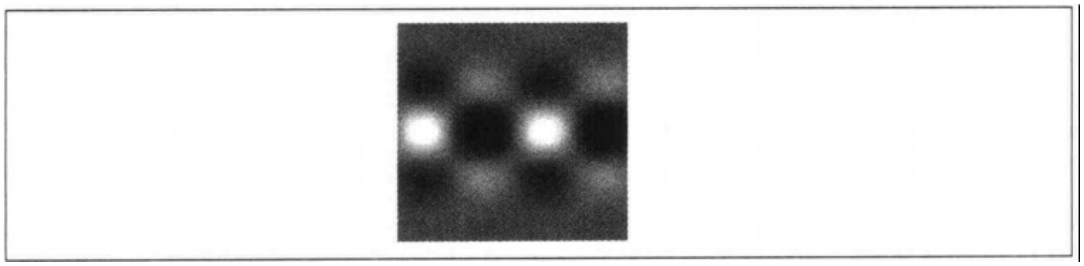


图10-7 numpytest.py的数组zs的图形表示

NumPy使得你可以高效地操作大型的数组。代码运行时比用大的列表要快得多。许多的Python用户永远不会碰上这样的问题，但很多科学家和工程师每天都用得着。

SWIG

Python的扩展模块可以用C或C++来写，可以很容易地扩展新的函数和类型。写这种扩展模块的指南可以在库参考里找到，在《Programming Python》一书里也有介绍。扩展模块的一个常见的用法是写一个已存在的库的接口，库里面也许有成百上千的函数。在这种情况不，自动化工具就可以救命了。David Beazley写的SWIG（简单封装接口产生器，the Simple Wrapper Interface Generator）是同类工具中最流行的。在http://www.swig.org可找到它并且文档齐全。用SWIG你可以写简单的接口定义，然后由SWIG按照规则产生C程序。SWIG有一个不错的特点，当用它来封装C++类库时，它自动地产生Python的所谓的影子类，让

用户像操纵Python的类一样操纵C++类。SWIG也可以为其他语言(包括Perl,Tcl)创建扩展。

Python复合组件（Pmw）

任何真打算用Tkinter做GUI工作的人应该看看Pmw，它是一个100%的基于Tkinter的框架，用于创建复合组件。它是由Greg McFarlane写的，学会它肯定会有回报。它的主页在http://www.dscpl.com.au/pmw/。

ILU和Fnorb

如果你对程序间的对话感兴趣，而且希望一个不需要COM支持的方案，那么有很多其他的类似功能的包。两个最常用的是ILU和Fnorb。

ILU意思是Xerox PARC的Inter Language Unification project（语言间归一工程）。它是自由、支持良好、稳定而高效率的，并且除了Python外，还支持C, C++, Java, Common Lisp, Modula-3以及Perl 5。网址在ftp://ftp.parc.xerox.com/pub/ilu/ilu.html。

与ILU不同，Fnorb是用Python写的，并且只支持Python。它对学习了解CORBA系统特别有帮助，一旦你学会了Python，它是很容易安装使用的。网址在http://www.dstc.edu.au/Fnorb/。

练习

1. **欺骗Web。**你也许没有一个Web服务器可用来使用*formletter.py*和*FormEditor.py*，因为它们要用到CGI脚本产生的数据。作为练习，写一个程序模拟产生与GCI脚本类似的文件。
2. **清除。***formletter.py*程序有一个严重的问题：也就是当在晚上运行时，任何抱怨都会导致打印一封信。每个晚上都会发生，因为没有机制能标识已经处理了一封信，而不需要再处理它了。纠正这个错误。

3. **给 grapher.py 增加绘图参数。**修改 *grapher.py*，让用户可指定返回 x 和 y 的表达式，而不只是 y。例如，用户可以写：`sin(x/3.1),cos(x/6.15)`，(注意逗号：这是一个元组！)，得到的输出类似于图 10-8 所示。

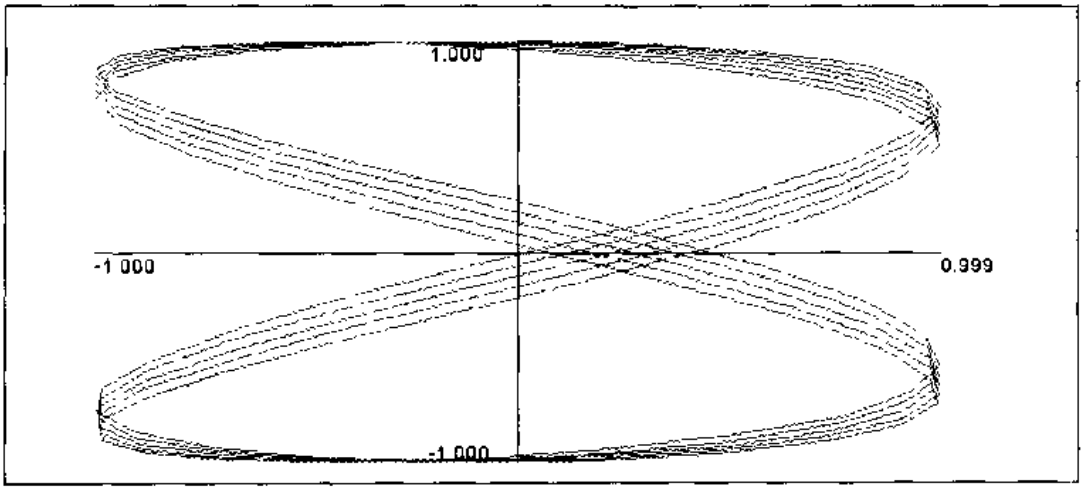


图 10-8 练习 3 的输出

第三部分

附录

本书的最后一个部分由三个附录组成，包含了许多其他资源的信息。

- 附录一“Python 资源”。提供了通用的Python资源，包括文档、建议和软件。
- 附录二“特定平台问题”。讲述了各种操作系统特定的资源。
- 附录三“练习解答”。给出了前两个部分每章后面练习的解答。

附录一

Python 资源

本附录为程序员精选了一些最有用的资源，包括了非标准发行版的部分，囊括了软件（Python 模块和扩展模块）、文档、讨论组、商业资源的支持。

Python 语言的 Web 站点

最重要的信息资源是http://www.python.org。这个站点是Python社区的中心。所有的Python软件、文档和其他可以得到的信息都在站点上直接，或者从站点上列出的位置上得到。我们鼓励你花时间探索一下它，它十分的大而且内容详尽。图 A-1 是该 Web 站点的首页。

Python 软件

标准的 Python 发行版

最基本的Python软件就是Python的解释器自身。在不同的平台上有多种形式。我们将在附录二讨论特殊平台上的有关问题。一般的说，得到当前发行版（distribution）最可靠的方法是从Python主要Web站点（http://www.python.org）上下载。Python 的 Web 站点由 Python 软件协会组织的志愿所管理。

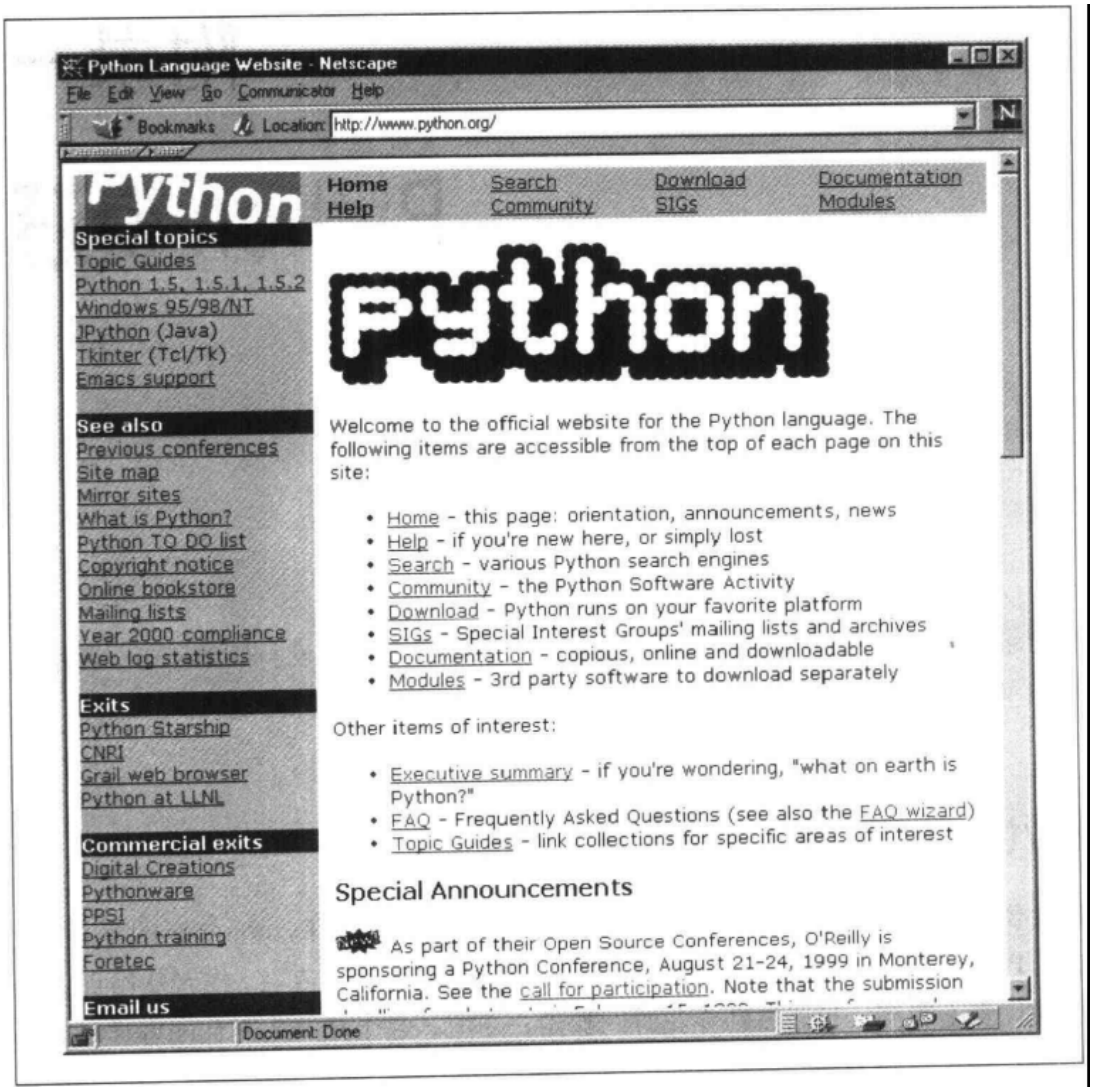


图 A-1 Python.org 站点首页

如果你想得到CD上的二进制的形式，Walnut Creek可以提供一个Python的CD-ROM，包含了所有常见平台（Windows，Macs，Unix的许多版本，BeOS和VMS）的二进制形式。在写这本书的时候，关于当前发行版的URL是http://www.cdrom.com/titles/prog/python.htm，检查目录可获得更新一些的版本。如在附录二中描述的那样，大多数Linux的发行版都包括Python。《Programming Python》和《Internet Programming with Python》都包括Python发行版的CD。

标准的发行版带有几百个模块，有C和Python两种形式。这些模块都被Guido及

其合作伙伴所支持（除非特别注明，当被更新工具代替时，老版本的模块保留一段时间以让用户升级它的软件，同时逐渐取消对它们的支持）。解释器、标准库和标准文档组成了 Python 用户可得到的最小程序集合。

在标准发行版之外，在Web上有数以百计的可用的包和模块，大多数都是自由的。我们将提到其中一些较特殊的，并指出在哪里可以发现它们。

Python 软件协会和 Python 联盟

Guido van Rossum 是 Python 的主要作者，近年来他已经得到了一些帮助，尤其是在 Python 语言进入公众视野以后。Python 软件协会（The Python Software Association，PSA）是一个由公司和个人组成的组织。这些公司和个人希望帮助维护一个自由的、有发展前景和良好支持的语言。PSA会员帮助维护 python.org 站点的正常运行，组织 Python 的会议，收集PSA 会员的会费以承担Web站点、会议和其他相关事情上的花费。如果你或你的公司对 PSA 感兴趣，可访问它的 Web 站点：http://www.python.org/psa 。

成为PSA成员的一个好处就是，可获得一个“Starship Python”的免费帐号。Starship Python是由很了不起的一个Python 用户 Christian Tismer运行的一个站点，为 Python 社区进行公共服务。该站点的URL 是：http://starship.python.net。

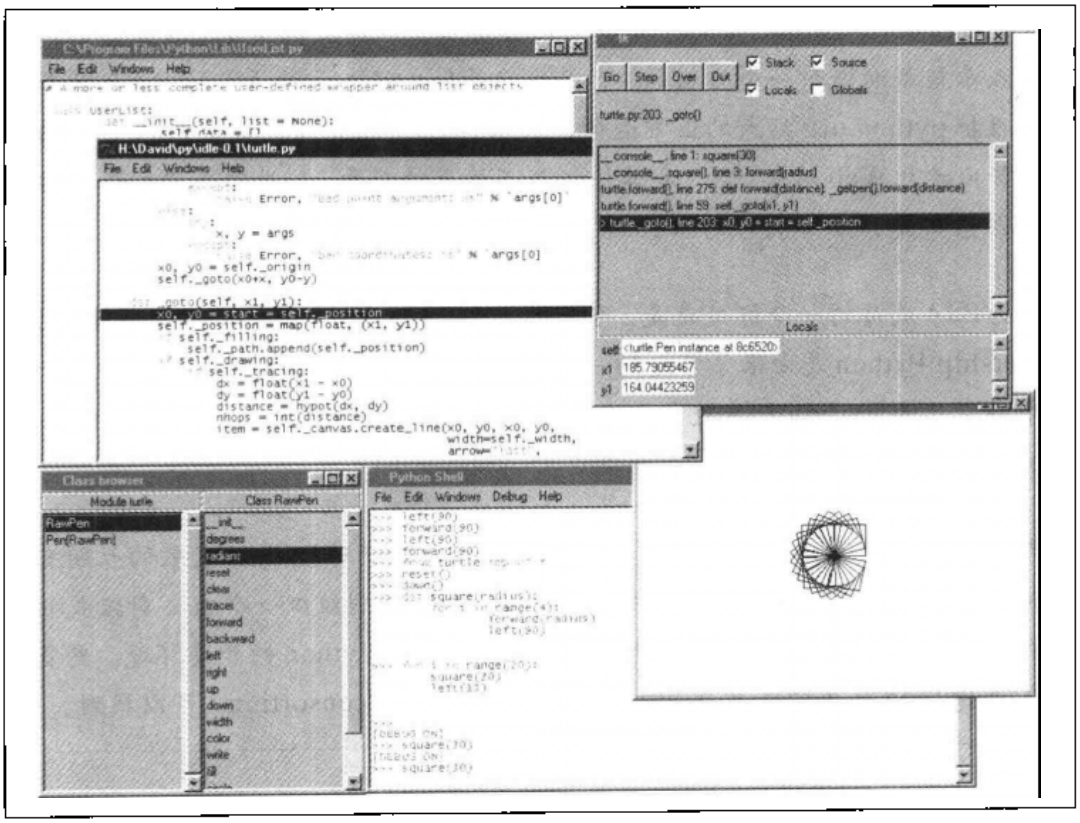
Python 联盟是最近发展起来的，它的目的就是提供对 Python 发展的长期支持。CNRI（Guido van Rossum 当前的雇主）正计划组织一个由公司组成的联盟，通过收取会员费用，支持、发展Python和JPython 的开发环境。更多关于 Python 联盟的信息在 http://www.python.org/consortium/ 可以找到。

Guido 的礼物

好像Python和它的标准库还不够，Guido发布了一些其他的程序作为标准发行版的一部分。它们都在 python 源目录树的 Tools 目录中（或者在 Windows 和 Mac 的安装路径中）。

在1.5.2这个系列中首先有一个Python的集成开发环境，叫做*idle*。就像图A-2显示的那样，它是基于Tkiner的GUI，所以它需要你安装Tk/Tcl。*idle*还不成熟，但它提供了一些很好的特性，使它成为理想的开发Python的环境：

- 一个Python的shell，比我们一直用的要更加智能化。
- 一个Python编辑器，可以自动的为Python的代码上色：语句用一种颜色，注释用另一种颜色，等等。在Emacs也是这样，易学好用。
- 一个类浏览器，允许你探测模块中的类并直接跳到源代码中的方法定义。
- 一个交互式的调试程序。



图A-2 idle IDE

商业投资者的贡献

注意： 在这一节中我们将提到一些商业软件公司。虽然这对于一个自由语言有些奇怪，但是Python社区基于Python的项目并不与“商业模式”相抵触。Python的许可证很特别，软件商可以不加限制的使用它。大多数Python用户都乐于听到公司成功的由Python构建的事情，有许多与Python有关的成功的故事，都是经理们不经意透露出来的（注1）。我们只提及那些为Python的程序员贡献代码的那些公司，而不包括那些在产品中包含Python的公司。

Scriptics 公司的 Tcl/Tk 工具包

本书中，我们所说的Tkinter GUI框架是建立在Tk GUI工具包之上的，该工具包用的是Tcl语言。二进制或者源代码的形式可以从Scriptics公司的Web站点上得到：http://www.scriptcs.com。要得到Tkinter相关的资源信息，看一下Tkinter的主题指导：http://www.python.org/topics/tkinter/。

Digital Creation 的自由版本

Digital Creation是一家软件公司，最近已经转向经营出售基于Python的软件包，以及在开源许可证下发行它们。他们已经为Python社区做出了卓著的贡献，通过发布标准发行版（例如，他们对`cPickle`和`cStringIO`模块的支持）以及许多方式帮助PSA成长。他们还贡献了两种自由的功能强大的工具，它们是：

ExtensionClass

一个C语言的扩展类型，可作为Python的类生成。而且，ExtensionClass允许你调整这些新类型工作的方式，包括对采集（Acquisition）的支持。采集是这样一种机制，通过它对象可以从部分对象那里得到属性，特别像例程从类中得到属性或是类从基类中得到属性。

Zope

一个用于在Web上发布Python对象层次的框架，用Zope能很容易地架起一

注1： Guido van Rossum 有的时候听到有些公司使用 Python。不过他们不想让公众知道，因为他们把自己使用 Python 的决定看成战略上的一种优势。

个对Python数据库对象的接口。通过提供模板支持、数据库引擎等等的接口，目前有不少针对Web的应用的Zope扩展程序。如果你正考虑开发复杂的Web应用（而不是第十章中我们展示的简单的处理形式），你应该仔细地研究一下这些工具。

Digital Creations公司的Web站点在http://www.digicool.com；它们的免费工具在http://www.digicool.com/site/Free/可找到，而Zope可在http://www.zope.org找到。

Pythonware

Pythonware是一家瑞典的Python软件供应商，有好几个项目正在开发过程中，包括一个Python的集成开发环境（IDE）、Windows平台上的对Tk的一个替代品以及一个图像处理框架。Pythonware已经以自由形式发布了一些其他的工具，它们十分有用，说明Pythonware的这些产品值得等待。这些自由产品中最重要的是第十章我们提到的Python的图像处理库（PIL）和最全面的Tkinter文档，要得到PIL和其他的Pythonware的工具，请访问Web站点http://www.pythonware.com。

其他的模块和程序包

在Web上还有很多其他的模块和程序包，在许多地方可以找得到：

- 在Python Web的主站点上（http://www.python.org/download/Contributed.html）列出了数以百计的模块，一系列的主题，包括网络工具，图像数据库，系统接口等等。
- PyModules FAQ是一个总在发展的模块列表，也是以分类的形式组织的。在http://starship.skyport.net/crew/aaron_watters/faqwiz/contrib.cgi。
- Startship的工作人员提供了许多工具。Startship项目是一个Web站点，任何PSA的成员都可以得到一个免费的帐号，包括Web页面，看一下前面的选读部分可以知道详细内容。

一些工具可以在以下三个值得特别注意的目录中找到，因为它们非常有用。它们是：

Gadfly

Aaron Watters用Python写的SQL数据库引擎，它的速度比不上那些高性能的商业软件上的数据库引擎，它的速度与微软的Access相当，Gadfly在http://www.chordate.com/gadfly.html。

Medusa

一个高性能的Internet服务器框架，也是完全用Python写成。作者通过运用多道I/O单处理服务，为HTTP、FTP和其他的IP服务提供了极高的性能；但是它只对非商业使用是免费的，商业使用证很贵的。Medusa的网站在http://www.nightmare.com/medusa/。

若你想寻找几个用于Python教学的工具，这里有几个：

turtle.py

Guido van Rossum写的Python模块，是Python 1.5.2的标准库的一部分。该模块在Tk窗口产生一个简单的“海龟图像”，海龟图像曾被广泛用来教孩子们学习Logo语言编程。

Alice

该程序设计的目的是允许非专业人员进行交互性的3D图像的研究。它的发展起源于弗吉尼亚大学的一个小组，但是现在它是由卡耐基-梅隆大学计算机科学系资助的。参见http://alice.cs.cmu.edu/。

Emacs支持

当Python还不是一个真正的软件的时候，它就对在Emacs中编辑Python代码有很好的支持了（在所有的Emacs允许的平台上）。在Emacs中你可以编辑彩色的代码，在缓冲中查看函数、类、方法，运行Python的解释器并且可以运行一个Pdb的调试器，所有这些在这个最流行最强大的编辑器中都是允许的。Emacs中对Python的支持的相关信息可以在这里找得到：http://www.python.org/emacs/。

Python 的文档和书籍

这里有三种关于 Python 的资料来源：标准的 Python 文档系列，出版的书籍，在线资料。

标准的 Python 文档

标准的 Python 文档包括 5 个单独的文档。它们有不同的格式（HTML，PDF，PostScript 和其他形式的），可在 http://www.python.org/doc/ 找到。它们是：

教程（The Tutorial）

对该语言的快速介绍，现在被大多数程序员用来学习 Python。它假定读者有些编程的知识，新手会发现它有些不太好懂，并且该文档没有给出面向对象的特性。

库参考（The Library Reference）

最重要的 Python 文档。它列出了所有的内置函数以及内置类型的方法和语法，并且描述了组成标准发行版的所有模块。有必要将它保存在你的硬盘上，当你对某个特定的函数接口或者语法拿不准的时候或者没记住某个特定的对象方法名字的时候，可以查看它。

语言参考（The Language Reference）

语言本身最正式的叙述，给出了语法操作、过程规则等等的精确定义。大多数的用户会忽略它，但是它给出了这个语言的复杂细节的准确描述。

扩展与嵌入（Extending and Embedding）

准确描述 Python 和 C 扩展程序之间交互性规则的文档（当 Python 被已经存在的 C 或者 C++ 程序调用的时候，与嵌入的情况类似）。若你想为 Python 写一个扩展模块，这本书讲述了做法。在跟踪模块中的错误的时候，保持跟踪引用一节尤为重要。

Python/C API

描述 Python 内部例程的文档，你也可以用这些例程从 C/C++ 程序中操作 Python 对象，通常在扩展模块中进行。

FAQ

同其他Internet上有趣的主题一样，Python也发展了一个常见问题（FAQ）列表。在http://www.python.org/doc/FAQ.html可以找到。它覆盖了Python编译、安装、编程的几乎所有信息（关于名字、起源、设计选择等等）。Python的FAQ列表由社区管理。任何PSA的成员都可以登录到一个Web驱动程序（一个CGI程序，类似与我们在第十章中见到的）上，更新已经存在的内容和添加新的内容。这样，FAQ内容既庞大又及时。

其他已经出版的书籍

除了你手中的这本书之外，书店中有三本Python的书。它们是：

- 《Programming Python》作者：Mark Lutz；出版社：O'Reilly & Associates。学过本书之后，你应该学习一下这本书（860页）。与本书相比，它更有深度，覆盖了Python的所有方面，并且带有些更复杂的例子。该书还讨论了Python/C的集成，以及像Tkinter GUI和代码持久性这样的高级应用。
- 《Internet Programming with Python》作者：Aaron Watters，Guido van Rossum，James Ahlstrom；出版社：M&T Books。这是一本477页的书，有对Python的全面的介绍，并特别强调了如何书写Web发布页面。
- 《Python Pocket Refercence》作者：Mark Lutz，出版社：O'Reilly & Associates。这是一本简洁的手册（75页），列出了语法的核心内容，以及最常用的模块和它们的函数标准，与Python1.5.1兼容。

其他的文档资源

Python的模块描述、howto文档、新手指南、普通任务等等这些Web页面十分重要，应该在这里列出它们的全部。不过相反，我们鼓励从Python的站点开始浏览，PSA的志愿者们（Ken Manheimer，Andrew Kuchling，Barry Warsaw，Guido van Rossum，等等）做了极大的努力使该站点内容全面而且有良好组织性。这样在你想查找东西的时候不会出现问题。大多数重要的程序和模块都有相关联的页面和文档。

新闻组，讨论组和 Email 帮助

Python 的发展很快，世界范围内的 Python 社区的用户要通过 Internet 交换关于 Python 的信息。每天的 Python 的信息交换大多以各种电子论坛的形式进行，每一个都有特定的目标和范围。

comp.lang.python/python-list

关于 Python 讨论的主要场所，是 comp.lang.python 用户新闻组。它的另一个途径是订阅邮件列表 python-list@cwi.nl（虽然正准备移到 python-list@python.org）。若你还无法访问 comp.lang.python 新闻组上的新闻资料，可以用 Dejanews（www.dejanews.com）服务读取站点上的内容。或者用 eGroups（http://www.egroups.com/list/python-list/）服务读取同样的 Python 列表。这个邮件列表 / 新闻组被用来提出问题，讨论特殊的 Python 问题，张贴工作广告等。

comp.lang.python.announce/python-list-announce

最近，产生了一个新的新闻组，作为一个低流量的邮件列表，用于 Python 相关新闻的重要通告（可以通过 python-list-announce@cwi.nl 得到）。comp.lang.python.announce 新闻组是一个中等大小的论坛，所以只有那些合适的帖子才可以张贴。

python-help@python.org

这个新闻组 / 邮件列表的一个主要特征就是自动向世界上成千上万个用户发送信息。但是这要求人们总要有快速的响应时间（无论在什么时间、什么人、什么地点读这个 Python 的新闻组），可能会吓住某些用户，尤其是新手。一个更个性化的地方是在 *Python-help* 地址提问，它作为一个在线的帮助进行服务。向 python-help@python.org 发送 Email 就是向众多的的志愿者发送广播，他们会尽最大的努力回答问题，然后发送到 *Python-help* 上。向列表写文章，写明你的应用软件的配置（Python 的版本，操作系统等等），并且要准确描述你的问题会很有帮助。这样做有助于志愿者理解你的问题，并有希望快速解决它。

特殊兴趣小组

在这里还应该提到更多的邮件列表。主要的Python新闻组因为它的普遍性值得一提。但有的时候，用户群因为一个特定的目标而为了一个特定的项目一起工作，如：为一个工具或这一系列工具开发重要的扩展程序或者格式化标准的接口。这些志愿者的团体称之为特殊兴趣小组（*special interest groups* 或 SIG）。这些志愿者的团体有自己的邮件列表，如果你觉得现在对他们有兴趣的话，你可以加入到他们当中。成功的 SIG包括 Matrix-SIG，帮助Jim Hugunin 开发 Numeric Python 扩展程序；String-SIG，开发正则表达式；以及 XML-SIG，开发用以解析处理 XML（可扩展标记语言）的工具。当前的 SIG（它们随任务完成而取消，随新的需要而产生）可以在http://www.python.org/sigs/找到。每一个SIG都有自己的邮件列表，自己的文档页以及描述。

JPython-interest

这是一个讨论 JPython 特定问题的邮件列表。如果你对其感兴趣的话，它值得一读。这是 Jim Hugunin 和 Barry Warsaw 用来发布关与 JPython 有关的信息以及反馈请求的地方。关于这个列表的信息在 http://www.python.org/mailman/listinfo/jpython-interes。

会议

虽然大多数的 Python 信息的交流都是以电子形式发生的，PSA 也组织定期的会议。这个会议是人们可以见到自己同行的唯一的场合（注 2）。这也是一个十分重要的讨论会：用来实现新工作，通过参加指导和讨论学习Python的方方面面。同样这也是一个讨论 Python 将来发展方向的地方。关于会议的信息会同 Python 的站点上显示的一样，正式张贴在新闻组上。以前会议的地点包括华盛顿特区、圣何塞和加州的 Livermore 以及德州的休斯顿。

注 2： 到目前为止，本书的两位作者也只在 Python 会议上见过面！

支持的公司，咨询和培训

Python 用户的资源还包括提供技术支持的工司、咨询以及培训。

Python 专业服务公司是一家专门提供不同类型 Python 技术支持和相关模块的公司，网址在 http://www.pythonpros.com。

目前没有关于Python咨询的在线列表，几个有经验的用户许诺可以提供短期或长期的咨询工作。查看http://www.python.org可以得到相关信息，也可以在Python的新闻组上张贴一个帮助请求。若你想得到更个性化的帮助，向 python-help@python.org 发邮件。

最后，有几个教学程序，是为那些想在线培训员工的公司准备的。现在没有全部的列表，但在你读这本书的时候或许就有了。再说一次，查看 http://www.python.org，新闻组或 Python-help@Python.org。

Tim Peters

Tim Peters 总在无偿劳动，不过，这个名字因为回答的问题比任何人都多而广为人知。他的注解十分有用、十分高明，而且往往都十分风趣。Peters 神龙见首不见尾，我们从没见过他，不过他很有可能就是一个编得十分巧妙的 Python 程序。设想一下，我们谈到过的人中只有一个人说曾见过 Tim 本人，这个人就是 Guido，想像一下吧……

附录二

特定平台问题

本附录的内容是特定平台问题——在哪里可以得到每个特定平台（即硬件和操作系统的结合）的Python发行版，与兼容有关的重要注解，或是特定平台上可以使用的工具。

Unix

一直以来，虽然Windows用户数量正稳定增加，Python的最大用户群却可能是Unix用户。Unix有好几种发行版，标准的获取方法是下载源代码文件（http://www.Python.org/download/down_source.html），并自己设置、编译、安装Python。过去，在Python的Web站点上曾为大多数主要Pytbon平台上保存一系列Python预编译的二进制代码，不过这些努力的结果大部分是徒劳的，因为没办法进行管理；Unix的版本太多而且每一种都有许多设置Python的方法。

我们在这本书中并没谈到Python的设置，那是因为我们用的是Python发行版最标准的部分。但是，如果你下载的是源代码文件，可以在*modules*目录中发现一个叫*setup*的参考文件。这个文件告诉你如何设置、编译每个文件，静态或者动态连接。可选模块可随各种发行形式改变，也可下载第三方扩展程序来进行参数化。

对Linux来说，有一个例外：没有二进制发布形式。大多数版本的Linux安装时已包括了Python，有的已用于设置管理系统。在你的Linux上的Python可能不是最新版本。Oliver Andrich管理了一系列的RPM（Red Hat Linux的标准格式文件包）的文件，也包括最流行的扩展程序，可在http://www.python.org/download/download_linux.html找到。

Unix上特定的扩展程序

在大多数（不是全部）Unix上有几个扩展程序，包括：

标准发行版

有Unix上大多数服务的接口，这些服务已得到很好实现。例如，有一个pwd模块，用于与口令文件进行交互性操作，一个grp模块用于管理Unix的数据库，还有与crypt库函数的接口（crypt模块），*dbm/ndbm/gdbm*数据库（dbm和gdbm），tty I/O控制调用（termios）和文件描述I/O接口（fcntl），衡量测试系统资源的模块（resources），一个系统登录工具接口模块（syslog），一个popen调用的包裹模块可以使与系统shell调用的接口更简单（commands），最后一个模块给出系统调用访问权，以发现文件调整时间或类似事件（stat）。

标准库也包括运行在某些特定Unix平台上的模块，如在SGI和SunOS/Solaris上的以下模块。

SGI特殊扩展程序

在SGI系统上，标准分布包括AL图像库接口模块（al和AL），CD库（cd），Mark Overmas所写的FORMS库（fl，flp和FL），字体管理器（fm），老式IRIX GL库（gl，GL和DEVICE）（注1），还有imglib图像库格式模块（imgfile）。

注1：OpenGL接口被一系列模块跨平台支持，这些模块当前由本书的作者之一（David Ascher）管理，在http://starship.python.net/~da/PyOpenGL/可以找到。当前它在SGI系统上同在UNIX平台和Windows平台上一样良好，可以被OpenGL或者是兼容的Mesa工具包连接。

SUN OS 特殊扩展程序

在SunOS/Solaris上，标准发行版包括一个模块，sunaudiodev，提供对视频设备的接口。

其他 Unix 资源

已发布了许多模块，用以支持不同的Unix工具或已在Unix上测试过，包括对视频系统的接口，扫描仪、摄像机的接口，对X-Windows系统的接口，和它的层次工具包以及许多其他的东西。若你认为某个特定扩展程序已经有接口了，可以查找Python的Web站点。

Windows 平台特定信息

Windows平台（Windows95、98和NT）是Python成长最快的领域之一，从用户数量、扩展程序上看都是这样。www.python.org上的标准发行版可以在Windows上工作得很好，Mark Hammond写的*win32all*包可作为Windows特定扩展程序，可从http://www.python.org/windows/win32all/获得，并包括几个强大的程序的扩展模块。

最值得一看的是*Pythonwin*程序，这是一个Python的集成开发环境，提供了一个交互解释器接口（带快捷键、字体颜色等），一个编辑器和一个对象浏览器（见图B-1）。*Pythonwin*事实上使用了几个大的程序包，允许Python做为Windows的一部分来驱动大量的库，如MFC，ODBC数据库接口、NT特殊服务如日志、性能监视器、内存映像文件、管道、时钟以及最重要的，COM的所有库，微软的通用组件模型。这意味着在第十章“框架与应用”中提到的，大多数现在的为Windows而写的软件可以从Python脚本运行，只要它支持脚本就行。一般地说，你能用VB这样的语言做出的任何东西，你都可以用支持COM的Python做出来。Python也可以作为一个ActiveX脚本宿主使用在IE浏览器这样的程序中。

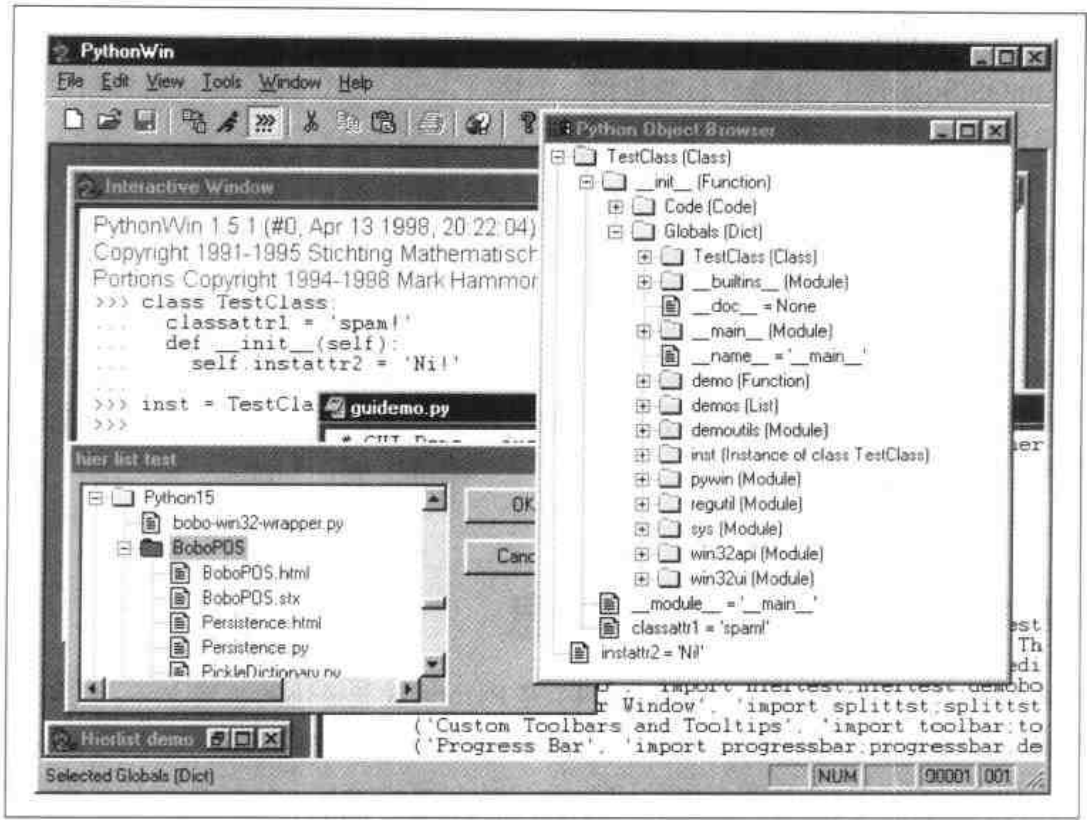


图 B-1　正在使用的 Pythonwin 程序

Macintosh 平台上的特定信息

Macintosh 平台也完全支持 Python，感谢 Janson 所作的努力。有一些 Mac 平台特性值得了解，首先你可在脚本外生成 applet，这样在脚本中删除一个文件同在 *sys.argv* 中去掉一个文件名是一样。而且，Just van Rossum （Guido 的兄弟）写了一个可以工作在 Mac 上的集成开发环境，包含在发行版中。最新版本可在 Web 地址：http://www.Python.org/download/download_mac.html 上找到。图 B-2 显示了一幅屏幕图。

而且，做为 MacPython 发行版的一部分，有几个模块还提供了对 Mac 特定服务的接口。接口包括苹果事件、组件、控制、对话、事件、字体、表单和菜单管理器、QuickDraw、QuickTime、Resource、Scrap 和 Sound 管理器、TextEdit、窗口管理器。还包括了 os 和 os.path 模块的接口实现，对通信工具箱、域名解析器、

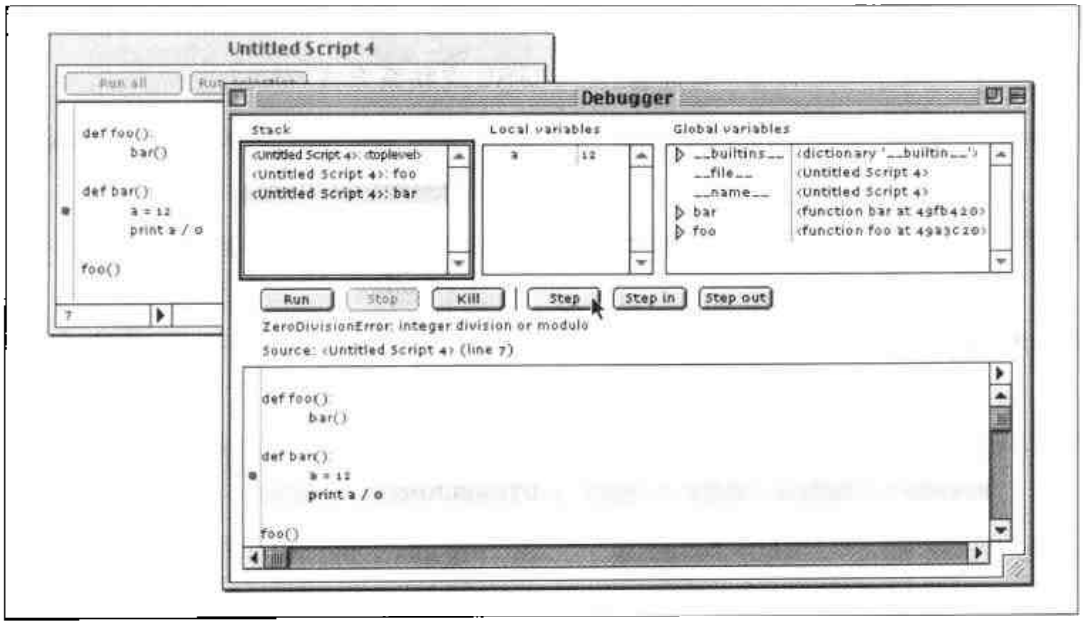


图 B-2 Macintosh IDE 调试器的屏幕图

FSSpec、别名管理器、finder 别名、标准文件包 Internet 设置、Speech 管理器等等其他接口（库手册中有一个补充文档，可在 http://www.python.org/doc/mac 上找到）。

Java

如你们在第十章见到的，JPython 是 Jim Hugunin 所写的 Java 在 Python 上的完整实现。下面是本附录中提到的最与众不同的一点：它不共享任何基于C的代码（但是大多数 Python 代码是基于 C 的）。JPython 的主页在 http://www.Python.org/JPython。JPython 扩展程序库与 Java 库是一样的，换句话说这是个巨大的列表。查找 Java 信息的一个好地方是 Sun 的 Web 站点 http://java.sun.com。

其他平台

最后要说的是，很多杰出的人已或多或少的把 Python 移植到其他的平台上了。表 B-1 列出我们所知的每个移植的平台、作者或管理者、可以得到信息的 URL 或联系方式。

表 B-1 信息来源

平台	作者 / 管理者	URL 或联系方式
Amiga	Irmen De Jong	http://www.geocities.com/ResearchTriangle/Lab/3172/python.html
BeOS	Chris Herborth	http://www.qnx.com/~chrish/Be/software/#programming
Windows CE	Brian Lloyd	http://www.digicool.com/~brian/PythonCE/index.htm
DOS/Windows 3.1	Hans Novak	http://www.cuci.nl/~hnowak/python/python.htm
QNX	Chris Herborth	ftp://ftp.qnx.com/usr/free/qnx4/os/language/python-1.5.tgz
Psion Series 5	Duncan Booth	http://dales.rmplc.co.uk/Duncan/PyPsion.htm
OpenVMS	Uwe Zessin	http://decus.decus.de/~zessin/
VxWorks	Jeff Stearns	电子邮件：jeffstearns@home.com

附录三

练习解答

第一章 开始

1. **交互**。假定你的Python设置正确，你应该这样的交互操作：

```
% python
copyright information lines...
>>> "Hello World!"
'Hello World!'
>>>                          # <Ctrl-D 或 Ctrl-Z 退出>
```

2. **编程**。这里是你的代码（也就是模块）文件和shell的交互：

```
% cat module1.py
print 'Hello module world!'

% python module1.py
Hello module world!
```

3. **模块**。下面的交互操作描述了通过导入一个模块文件来运行它。记住要想不停止并重新启动解释器，你需要重载来再次运行。把文件移动到一个不同的目录中并且再次的导入它是一个技巧性问题：如果Python在原来的目录生成一个*module1.pyc*文件，当你导入的时候就会使用它，即使源代码文件（*.py*）已经移到Python查找路径不包括的目录中了。如果Python有权访问源文件的目录并且包含了一个模块编译后的字节码形式，*.pyc*文件会被自动的写入。我们在第五章“模块”中讲述了更多的细节问题。

```
% python
>>> import module1
Hello module world!
>>>
```

4. **脚本**。假定你的平台支持#!技巧，你的解决方法看起来应该像这样（尽管你的#!行可能需要在机器上列出其他路径）：

```
% cat module1.py
#!/usr/local/bin/python          (或者是#!/usr/bin/env python)
print 'Hello module world!'

% chmod +x module1.py

% module1.py
Hello module world!
```

5. **错误**。下面的交互示范了完成这个练习可能得到的错误信息的种类。真的，你正在触发Python的异常；默认的异常处理行为终止运行Python程序并在屏幕上打印一个错误信息和栈轨迹（stack trace）。栈轨迹显示了异常发生的时候你在程序中所处的位置（在这里它不是很有趣，既然异常产生在交互提示符的顶部，在进行中没有函数调用）。在第七章“异常”你将看到可以用try语句捕捉异常并任意地处理它们；你还会看到Python为了特定的错误声明需要，还包含了一个成熟的源代码调试器。现在，注意在程序错误产生的时候Python给出了有用的信息（而不是静静地宕机）：

```
% python
>>> 1 / 0
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo
>>>
>>> x
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: x
```

6. **中断**。当你输入这些代码时：

```
L = [1, 2]
L.append(L)
```

会生成了一个Python中的循环数据结构。在1.5.1版以前，Python的打印还没有智能化到可以探测对象中的循环，并且会打印一个无终止的流[1, 2,

[1, 2, [1, 2, [1, 2, 等等，直到你敲击机器上的中断组合键（严格的说，如果你中断了一个程序的话，除非你的解释器在一个程序中，否则的话会出现一个键盘中断异常在顶部打印默认的信息）。从 Python 1.5.1 版本开始，打印足够智能化了，可以探测循环并用[[...]]代替打印。

循环的原因是微妙的，并且需要你具备第二章中的知识。不过简而言之，Python中的语句总生成对对象的引用(你可以想像成隐含的跟着一个指针)。当你运行上面的第一个赋值，名字L成为对一个含有两项的列表对象名字的引用。现在，Python列表实际是对象引用的数组，用一个 append方法通过包含另一个对象引用在原处改变数组。在这里，append调用从头到尾对L添加了一个引用，图 C-1 中描述了循环。信不信由你，循环数据结构有的时候是有用的。(但是被打印的时候除外！)

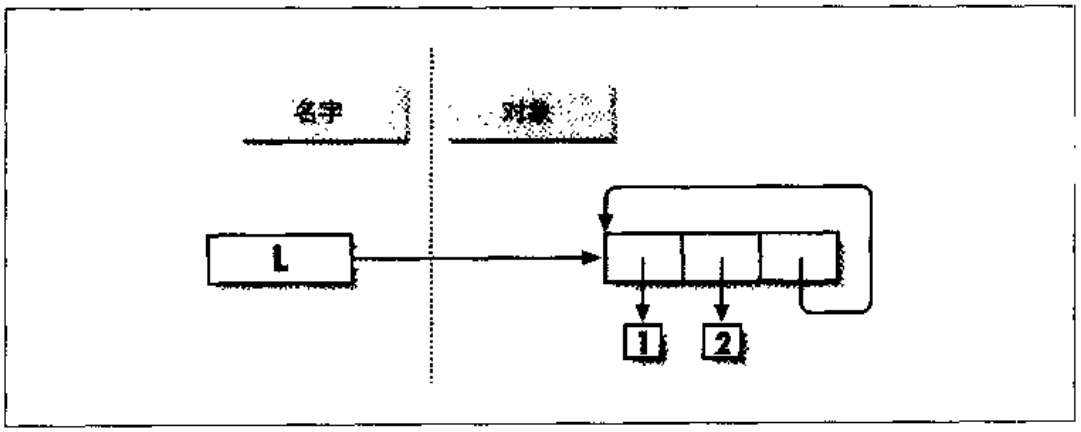


图 C-1 一个循环的列表

第二章 类型和操作符

1. **基础。**这些是你应该得到的结果，关于它们的含义有一些注释：

```
>>> 2 ** 16                          # 2的16次幂
65536
>>> 2 / 5, 2 / 5.0                   # 整数/截取，浮点数/不截取
(0, 0.4)
```

字符串

```
>>> "spam" + "eggs"                  # 合并
```

```
'spameggs'
>>> S = "ham"
>>> "eggs " + S
'eggs ham'
>>> S * 5                                  # 重复
'hamhamhamhamham'
>>> S[:0]                                  # 在前头的空分片--[0:0]
''
>>> "green %s and %s" % ("eggs", S)        # 格式
'green eggs and ham'
```

元组

```
>>> ('X',)[0]                              # 索引一个单项的元组
'X'
>>> ('X', 'y')[1]                          # 索引一个两项的元组
'y'
```

列表

```
>>> L = [1,2,3] + [4,5,6]                  # 列表操作
>>> L, L[:], L[:0], L[-2], L[-2:]
([1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6], [], 5, [5, 6])
>>> ([1,2,3]+[4,5,6])[2:4]
[3, 4]
>>> [L[2], L[3]]                           # 从偏移中取值，在一个列表中存储
[3, 4]
>>> L.reverse(); L                         # 方法：在原处反转列表
[6, 5, 4, 3, 2, 1]
>>> L.sort(); L                            # 方法：在原处排序列表
[1, 2, 3, 4, 5, 6]
>>> L.index(4)                             # 方法：前4项的位移（查找）
3
```

字典

```
>>> {'a':1, 'b':2}['b']                    # 通过键索引字典
2
>>> D = {'x':1, 'y':2, 'z':3}
>>> D['w'] = 0                             # 生成一个新的入口
>>> D['x'] + D['w']
1
>>> D[(1,2,3)] = 4                         # 作为一个键使用的元组（不可变的）
>>> D
{'w': 0, 'z': 3, 'y': 2, (1, 2, 3): 4, 'x': 1}
>>> D.keys(), D.values(), D.has_key((1,2,3))          # 方法
(['w', 'z', 'y', (1, 2, 3), 'x'], [0, 3, 2, 4, 1], 1)
```

空

```
>>> [[]], ["",[],(),{},None]
([[]], ['', [], (), {}, None])
```

2. **索引和分片。**

   a. 越界索引（如，L[4]）出错；Python 总要检查以确信所有的偏移都是在一个序列的界内（这一点与C不同， C语言的越界索引有时会宕掉你的系统）。

   b. 相反，界外的分片操作（如，L[-1000:100]）工作，因为 Python 依比例决定出界分片，这样它们总是适合的（如果需要的话，它们被设定为 0 和序列的长度）。

   c. 反向析取一个序列 —— 低边界 >高边界（如，L[3:1]） —— 会真正的工作的。我们得到一个空的分片（[]），因为Python衡量分片的界限来确保低边界总要小于或者等于高边界（如，L[3:1]被按比例设为L[3:3]，在偏移量 3 之后空的插入点）。Python 的分片操作总是从左到右析取，即使你用负索引也一样（它们首先通过添加长度转化为正索引）。

```
>>> L = [1, 2, 3, 4]
>>> L[4]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>> L[-1000:100]
[1, 2, 3, 4]
>>> L[3:1]
[]
>>> L
[1, 2, 3, 4]
>>> L[3:1] = ['?']
>>> L
[1, 2, 3, '?', 4]
```

3. **索引、分片和删除。**你与解释器的交互看起来应该像下面这样。注意对一个偏移赋值一个空的列表存储了一个空的列表对象。不过在这里对一个分片的赋值删除了分片。分片操作针对另外一个序列。否则你会得到一个类型错。

```
>>> L = [1,2,3,4]
>>> L[2] = []
>>> L
```

```
[1, 2, [], 4]
>>> L[2:3] = []
>>> L
[1, 2, 4]
>>> del L[0]
>>> L
[2, 4]
>>> del L[1:]
>>> L
[2]
>>> L[1:2] = 1
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation
```

4. **元组赋值。**X 和 Y 的值被交换。当元组出现在一个赋值操作符（=）的左右的时候，Python根据位置来为左侧的目标赋值为右侧的对象。最简单的理解方式是，放在等号左边的目标不是一个实际的元组，即使它们看上去像是一个；它们是一系列简单的独立的赋值对象。等号右边的是元组，在赋值的时候得到解析（元组为达到交换作用提供了临时的赋值）。

```
>>> X = 'spam'
>>> Y = 'eggs'
>>> X, Y = Y, X
>>> X
'eggs'
>>> Y
'spam'
```

5. **字典的键。**任何不可变的对象都可以作为一个字典的键整数使用：元组，字符串等等。这真的是一个字典，即使有的键看起来像整数偏移。混合的键类型也照常工作。

```
>>> D = {}
>>> D[1] = 'a'
>>> D[2] = 'b'
>>> D[(1, 2, 3)] = 'c'
>>> D
{1: 'a', 2: 'b', (1, 2, 3): 'c'}
```

6. **字典索引。**索引一个不存在的键（D['d']）出错；赋值一个不存在的键（D['d'] = 'spam'）生成一个新的字典入口。另一方面，列表的出界索引也会出错，出界赋值也一样。变量名工作类似于字典键，在它们被引用的时

候必须已经被赋值了，不过要在第一次赋值的时候生成。事实上，如果你需要的话，变量名可以作为字典的键被处理（在模块的名字空间里或是在栈帧字典中它们是不可见的）。

```
>>> D = {'a':1, 'b':2, 'c':3}
>>> D['a']
1
>>> D['d']
Traceback (innermost last):
  File "<stdin>", line 1, in ?
KeyError: d
>>> D['d'] = 4
>>> D
{'b': 2, 'd': 4, 'a': 1, 'c': 3}
>>>
>>> L = [0,1]
>>> L[2]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>> L[2] = 3
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list assignment index out of range
```

7. **一般的操作。**

   a. +操作符不能用在不同的／混合的类型上（如，字符串+列表，列表+元组）。

   b. +不能用在字典上，因为它们不是序列。

   c. append方法只能用在列表而不是字符串上，并且键只用在字典上。append认为它的目标是可变的，既然这是一个在原位的扩展；字符串是不可变的。

   d. 分片与合并总是返回与被处理的对象同类型的一个新的对象。

```
>>> "X" + 1
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation
>>>
>>> {} + {}
```

```
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: bad operand type(s) for +
>>>
>>> [].append(9)
>>> "".append('s')
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: attribute-less object
>>>
>>> {}.keys()
[]
>>> [].keys()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: keys
>>>
>>> [][:]
[]
>>> ""[:]
''
```

8. **字符串索引**。既然字符串都是单字符字符串的集合，每次索引一个字符串，你都得到一个字符串，这个字符串可以被再次索引。S[0][0][0][0][0]保持一次又一次的索引第一个字符 。除非列表包含字符串，否则它一般不可以为列表工作（列表可以包含任意的对象）。

```
>>> S = "spam"
>>> S[0][0][0][0][0]
's'
>>> L = ['s', 'p']
>>> L[0][0][0]
's'
```

9. **不可变性**。下面任何一个方法都可以。索引并不赋值，因为字符串是不可变的。

```
>>> S = "spam"
>>> S = S[0] + '1' + S[2:]
>>> S
's1am'
>>> S = S[0] + '1' + S[2] + S[3]
>>> S
'slam'
```

10. **嵌套。**你的内容可能不同：

```
>>> me = {'name':('mark', 'e', 'lutz'), 'age':'?', 'job':'engineer'}
>>> me['job']
'engineer'
>>> me['name'][2]
'lutz'
```

11. **文件。**

```
% cat maker.py
file = open('myfile.txt', 'w')
file.write('Hello file world!\n')
file.close()                          # close不总是需要的

% cat reader.py
file = open('myfile.txt', 'r')
print file.read()

% python maker.py
% python reader.py
Hello file world!

% ls -l myfile.txt
-rwxrwxrwa   1 0        0              19 Apr 13 16:33 myfile.txt
```

12. **dir函数复习。**只是你用列表得到的；字典可以作同样的工作（但是用不同的方法名字）。

```
>>> [].__methods__
['append', 'count', 'index', 'insert', 'remove', 'reverse', 'sort']
>>> dir([])
['append', 'count', 'index', 'insert', 'remove', 'reverse', 'sort']
```

第三章 基本语句

1. **编写基本循环。**如果做完了这个练习，你的代码应该看起来这样：

```
>>> S = 'spam'
>>> for c in S:
...     print ord(c)
...
115
112
97
109
```

```
>>> x = 0
>>> for c in S: x = x + ord(c)
...
>>> x
433

>>> x = []
>>> for c in S: x.append(ord(c))
...
>>> x
[115, 112, 97, 109]

>>> map(ord, S)
[115, 112, 97, 109]
```

2. **反斜线字符**。假定你的机器可以处理的话，这个例子打印响铃字符（\a）50次。你应该得到一系列哔哔声（或者是一长声，如果你的机器足够快的话）。反正我们已经警告过你了。

3. **字典排序**。这是应该练习的一个解决方法；有什么问题的话请看第二章。

```
>>> D = {'a':1, 'b':2, 'c':3, 'd':4, 'e':5, 'f':6, 'g':7}
>>> D
{'f': 6, 'c': 3, 'a': 1, 'g': 7, 'e': 5, 'd': 4, 'b': 2}
>>>
>>> keys = D.keys()
>>> keys.sort()
>>> for key in keys:
...     print key, '=>', D[key]
...
a => 1
b => 2
c => 3
d => 4
e => 5
f => 6
g => 7
```

4. **程序逻辑的选择**。这是我们的解决方案；你的可能有些不同。

a.

```
L = [1, 2, 4, 8, 16, 32, 64]
X = 5

i = 0
while i < len(L):
```

```
        if 2 ** X == L[i]:
            print 'at index', i
            break
            i = i+1
        else:
            print X, 'not found'
```

b.

```
L = [1, 2, 4, 8, 16, 32, 64]
X = 5

for p in L:
    if (2 ** X) == p:
        print (2 ** X), 'was found at', L.index(p)
        break
else:
    print X, 'not found'
```

c.

```
L = [1, 2, 4, 8, 16, 32, 64]
X = 5

if (2 ** X) in L:
    print (2 ** X), 'was found at', L.index(2 ** X)
else:
    print X, 'not found'
```

d.

```
X = 5
L = []
for i in range(7): L.append(2 ** i)
print L

if (2 ** X) in L:
    print (2 ** X), 'was found at', L.index(2 ** X)
else:
    print X, 'not found'
```

e.

```
X = 5
L = map(lambda x: 2**x, range(7))
print L

if (2 ** X) in L:
    print (2 ** X), 'was found at', L.index(2 ** X)
else:
    print X, 'not found'
```

第四章 函数

1. **基础。**

```
% python
>>> def func(x): print x
...
>>> func("spam")
spam
>>> func(42)
42
>>> func([1, 2, 3])
[1, 2, 3]
>>> func({'food': 'spam'})
{'food': 'spam'}
```

2. **参数。**这是一个解决办法。在这个测试中你不得不使用print查看输出代码，因为一个文件与交互式输入的代码有所不同；Python不回显表达式语句的结果。

```
% cat mod.py
def adder(x, y):
    return x + y

print adder(2, 3)
print adder('spam', 'eggs')
print adder(['a', 'b'], ['c', 'd'])
```

```
% python mod.py
5
spameggs
['a', 'b', 'c', 'd']
```

3. **可变参数。**两个交互式adder函数代码显示如下。最难的部分是解决如何初始化一个可以传递任何类型的空值的存储器。在第一个解决方案中，我们使用手工的类型测试去查找一个整数和第一个参数的空的分片(假设是一个序列)。在第二个方案中，我们只用第一个参数初始化，并扫描第二项与后面的项。第二个解决方案更好(说实话，在Python课程中的学生，不能理解第一个方案)。这两种解决方案都假定所有的参数都是同样的类型，并且都不能用字典上；就像我们在第二章中看到的那样，+不能用在混合的类型或是字典上。我们可以添加一个类型测试并指定一个特别的代码来添加字典，不过那都是额外的事情。

```
% cat adders.py

def adder1(*args):
    print 'adder1',
    if type(args[0]) == type(0):            # 整数?
        sum = 0                             # 初始化为0
    else:                                   # 其他序列:
        sum = args[0][:0]                   # 用arg1的空分片
    for arg in args:
        sum = sum + arg
    return sum

def adder2(*args):
    print 'adder2',
    sum = args[0]                           # 初始化为 arg1
    for next in args[1:]:
        sum = sum + next                    # 添加项 2..N
    return sum

for func in (adder1, adder2):
    print func(2, 3, 4)
    print func('spam', 'eggs', 'toast')
    print func(['a', 'b'], ['c', 'd'], ['e', 'f'])
```

```
% python adders.py
adder1 9
adder1 spameggstoast
adder1 ['a', 'b', 'c', 'd', 'e', 'f']
adder2 9
adder2 spameggstoast
adder2 ['a', 'b', 'c', 'd', 'e', 'f']
```

4. **关键字。**这是我们对第一部分的解决方法。要通过关键字参数迭代，在一个函数首部使用**args for并使用一个这样的循环：for x in args.keys(): use args[x]

```
% cat mod.py
def adder(good=1, bad=2, ugly=3):
    return good + bad + ugly

print adder()
print adder(5)
print adder(5, 6)
print adder(5, 6, 7)
print adder(ugly=7, good=6, bad=5)
```

```
% python mod.py
```

```
6
10
14
18
18
```

5/6. 这是我们对练习 5 和练习 6 的解决方法，不过 Guido 已经使它们多余了；Python 1.5 包括可信的字典方法，可以作类似拷贝与添加字典之类的工作。参考 Python 库手册或者是《Python Pocket Reference》可以得到更多的细节。既然不是字典，X[:]并不为字典工作（见第二章）。注意如果不拷贝而是赋值（e = d），我们可以生成一个对共享的字典对象的引用，改变 d 也改变了 e。

```
% cat dict.py

def copyDict(old):
    new = {}
    for key in old.keys():
        new[key] = old[key]
    return new

def addDict(d1, d2):
    new = {}
    for key in d1.keys():
        new[key] = d1[key]
    for key in d2.keys():
        new[key] = d2[key]
    return new

% python
>>> from dict import *
>>> d = {1:1, 2:2}
>>> e = copyDict(d)
>>> d[2] = '?'
>>> d
{1: 1, 2: '?'}
>>> e
{1: 1, 2: 2}

>>> x = {1:1}
>>> y = {2:2}
>>> z = addDict(x, y)
>>> z
{1: 1, 2: 2}
```

7. **更多参数匹配的例子。**这里是你应该得到的交互类型，注释可以对匹配加以解释：

```
def f1(a, b): print a, b                    # 正常的参数
def f2(a, *b): print a, b                   # 位置上的可变参数varargs
def f3(a, **b): print a, b                  # 关键字可变参数
def f4(a, *b, **c): print a, b, c           # 混合模式
def f5(a, b=2, c=3): print a, b, c          # 默认情况
def f6(a, b=2, *c): print a, b, c           # 默认情况+位置上的可变参数

% python
>>> f1(1, 2)                                # 通过位置匹配（顺序问题）
1 2
>>> f1(b=2, a=1)                            # 通过名字匹配（顺序无关）
1 2

>>> f2(1, 2, 3)                             #  一个元组中的额外集合
1 (2, 3)

>>> f3(1, x=2, y=3)                         #  一个字典中的额外的关键字集合
1 {'x': 2, 'y': 3}

>>> f4(1, 2, 3, x=2, y=3)                   # 另外的两种方式
1 (2, 3) {'x': 2, 'y': 3}

>>> f5(1)                                   # 扔掉了两个参数
1 2 3
>>> f5(1, 4)                                # 只有一个默认值使用了
1 4 3

>>> f6(1)                                   #  一个参数：匹配"a"
1 2 ()
>>> f6(1, 3, 4)                             # 额外的位置上的集合
1 3 (4,)
```

第五章 模块

1. **基础，导入。**这比你想像的要简单。当你做完的时候，文件和交互与下面的代码类似；记住Python可以把整个文件读入一个字符串或行列表中，内置函数len返回字符串和列表的长度：

```
% cat mymod.py
```

```
def countLines(name):
    file = open(name, 'r')
    return len(file.readlines())

def countChars(name):
    return len(open(name, 'r').read())

def test(name):                                    # 或传递文件对象
    return countLines(name), countChars(name)      # 或是返回字典
```

```
% python
>>> import mymod
>>> mymod.test('mymod.py')
(10, 291)
```

在 Unix 上，你可以用一个 wc 调整你的输出。顺便说一下，要完成比较难的部分（在一个文件对象中传递，这样你只用打开文件一次），你可能需要使用内置文件对象的 seek 方法。在这本书中不包括这个内容，不过它很像 C 语言的 fseek 调用（并且是在幕后调用）；seek 在文件中把当前的位置重设定为一个传递的偏移。要返到文件的开头而不用关闭并重新打开，调用 *file.seek (0)*；文件的读方法可以获得文件中当前的位置，所以你需要返回重新读取。如下：

```
% cat mymod2.py
def countLines(file):
    file.seek(0)                                   # 返到文件开头
    return len(file.readlines())

def countChars(file):
    file.seek(0)                                   # 同上(如果需要的话就返到开头)
    return len(file.read())

def test(name):
    file = open(name, 'r')                         # 传递文件对象
    return countLines(file), countChars(file) # 只打开文件一次
```

```
>>> import mymod2
>>> mymod2.test("mymod2.py")
(11, 392)
```

2. from/from*。这是 from*；用 countChars 代替 * 做剩下的工作：

```
% python
>>> from mymod import *
>>> countChars("mymod.py")
291
```

3. __main__。如果你编码正确的话，两种方式都会工作(程序运行或是模块导入)：

```
% cat mymod.py
def countLines(name):
    file = open(name, 'r')
    return len(file.readlines())

def countChars(name):
    return len(open(name, 'r').read())

def test(name):                                    # 或者是传递文件对象
    return countLines(name), countChars(name)      # 或者是返回一个字典

if __name__ == '__main__':
    print test('mymod.py')

% python mymod.py
(13, 346)
```

4. 嵌套的导入。我们的解决方案如下：

```
% cat myclient.py
from mymod import countLines
from mymod import countChars
print countLines('mymod.py'), countChars('mymod.py')

% python myclient.py
13 346
```

作为这个例子的剩余部分：*mymod* 的函数从myclient的顶部是可以访问的（也就是说，可导入的），既然from只是在导入者中对名字赋值（就好像mymod的def语句出现在myclient中一样）。如果 *myclient* 使用导入，你就不得不在mymod中使用一个路径从 *myclient* 得到该函数（例如：myclient.mymod.countLines）。实际上，你可以定义集合模块从其他的模块中导入所有的名字，这样它们在一个单一的有用的模块中是可用的。用下面的代码，用三个不同的名字somename拷贝来结束：mod1.somename，collector.somename和__main__.somename；这三个在初始化的时候共享同样的整数：

```
% cat mod1.py
somename = 42

% cat collector.py
from mod1 import *                    # 在这里集合了许多的名字
from mod2 import *                    # from对许多的名字赋值
```

```
from mod3 import *

>>> from collector import somename
```

5. **重载。**这个练习中要求你通过改变书中的*changer.py*例子来进行练习，所以在这里我们就不向你展示什么了。如果你还有什么有趣的想法，自己另外加以练习吧。

6. **循环导入。**简单的说，导入的 recur2 先工作，因为递归得到的导入是在recur1中的import语句而不是在recur2的from语句发生的。复杂一点说，类似于这样：导入 recur2 先工作，因为从 recur1 到 recur2 的递归是作为整体取得的 recur2，而不是得到特定的名字。当从 recur1 导入的时候 recur2 是不完整的，不过正因为是使用 import 而不是 from，所以你是安全的：Python 发现并返回已经生成的 recur2 模块对象，并且顺利继续运行剩余的 recur1。当 recur2 重新导入的时候，第二个 from 在 recur1 中发现名字 Y（它已经完全运行了），所以不报错。作为一个脚本运行文件，与作为模块导入不太一样；这种情况与在脚本的交互过程中先运行 import 还是 from 是个道理。例如，既然 recur2 是 recur1 中导入的第一个模块，作为一个脚本运行 recur1 与交互导入 recur2 是一样的。

第六章 类

1. **基础。**下面是我们的代码，以及一些交互式的测试。__add__ 方法在超类里只能出现一次。注意当在表达式里实例出现在 + 的右边时，将会出错；请用 __radd__ 方法。（这是我们忽略了的一个高级话题，请参见其他 Python 书籍和 Python 库参考。）

```
% cat adder.py

class Adder:
    def add(self, x, y):
        print 'not implemented!'    def __init__(self, start=[]):
        self.data = start
    def __add__(self, other):
        return self.add(self.data, other)

class ListAdder(Adder):
    def add(self, x, y):
        return x + y
```

```
class DictAdder(Adder):
    def add(self, x, y):
        new = {}
        for k in x.keys(): new[k] = x[k]
        for k in y.keys(): new[k] = y[k]
        return new
```

```
% python
>>> from adder import *
>>> x = Adder()
>>> x.add(1, 2)
not implemented!
>>> x = ListAdder()
>>> x.add([1], [2])
[1, 2]
>>> x = DictAdder()
>>> x.add({1:1}, {2:2})
{1: 1, 2: 2}

>>> x = Adder([1])
>>> x + [2]
not implemented!
>>>
>>> x = ListAdder([1])
>>> x + [2]
[1, 2]
>>> [2] + x
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: __add__ nor __radd__ defined for these operands
```

2. **操作符重载。**这里我们用了一些没有详细讲述的重载方法，但它们应该是好理解的。在构造函数里拷贝初始值是重要的，因为它是可变的，而你不想改变一个共享的对象。对封装的列表的访问是通过__getattr__方法实现的：

```
% cat mylist.py

class MyList:
    def __init__(self, start):
        #self.wrapped = start[:]                    # 拷贝start：没有副作用
        self.wrapped = []                           # 明确它是一个列表
        for x in start: self.wrapped.append(x)
    def __add__(self, other):
        return MyList(self.wrapped + other)
    def __mul__(self, time):
        return MyList(self.wrapped * time)
```

```
    def __getitem__(self, offset):
        return self.wrapped[offset]
    def __len__(self):
        return len(self.wrapped)
    def __getslice__(self, low, high):
        return MyList(self.wrapped[low:high])
    def append(self, node):
        self.wrapped.append(node)
    def __getattr__(self, name):          # 其他成员--sort/reverse，等等
        return getattr(self.wrapped, name)
    def __repr__(self):
        return `self.wrapped`

if __name__ == '__main__':
    x = MyList('spam')
    print x
    print x[2]
    print x[1:]
    print x + ['eggs']
    print x * 3
    x.append('a')
    x.sort()
    for c in x: print c,
```

```
% python mylist.py
['s', 'p', 'a', 'm']
a
['p', 'a', 'm']
['s', 'p', 'a', 'm', 'eggs']
['s', 'p', 'a', 'm', 's', 'p', 'a', 'm', 's', 'p', 'a', 'm']
a a m p s
```

3. **子类。**下面是我们的答案，你的解答应该是相似的。

```
% cat mysub.py

from mylist import MyList

class MyListSub(MyList):
    calls = 0                                       # 由实例共享

    def __init__(self, start):
        self.adds = 0                               # 每个实例都不同
        MyList.__init__(self, start)

    def __add__(self, other):
        MyListSub.calls = MyListSub.calls + 1       # 类 -级的计数器
        self.adds = self.adds + 1                   # 每个实例的计数器
```

```
        return MyList.__add__(self, other)

    def stats(self):
        return self.calls, self.adds                    # 所有的计数，自身计数

if __name__ == '__main__':
    x = MyListSub('spam')
    y = MyListSub('foo')
    print x[2]
    print x[1:]
    print x + ['eggs']
    print x + ['toast']
    print y + ['bar']
    print x.stats()
```

```
% python mysub.py
a
['p', 'a', 'm']
['s', 'p', 'a', 'm', 'eggs']
['s', 'p', 'a', 'm', 'toast']
['f', 'o', 'o',  'bar']
(3, 2)
```

4. **元类方法。**注意操作符也是通过__getattr__取属性值，为了让它们能工作，你需要返回一个值。

```
>>> class Meta:
...     def __getattr__(self, name):        print 'get', name
...     def __setattr__(self, name, value): print 'set', name, value
...
>>> x = Meta()
>>> x.append
get append
>>> x.spam = "pork"
set spam pork
>>>
>>> x + 2
get __coerce__
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: call of non-function
>>>
>>> x[1]
get __getitem__
Traceback (innermost last):
  File "<stdin>", line 1, in ?
```

```
TypeError: call of non-function

>>> x[1:5]
get __len__
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: call of non-function
```

5. **集合对象。**这里是一种交互，注释解释了调用哪一个方法。

```
% python
>>> from set import Set
>>> x = Set([1,2,3,4])          # 运行__init__
>>> y = Set([3,4,5])

>>> x & y                       # __and__，交集，然后__repr__
Set:[3, 4]
>>> x | y                       # __or__，并集，然后__repr__
Set:[1, 2, 3, 4, 5]

>>> z = Set("hello")            # __init__去掉重复
>>> z[0], z[-1]                 # __getitem__
('h', 'o')

>>> for c in z: print c,        # __getitem__
...
h e l o
>>> len(z), z                   # __len__, __repr__
(4, Set:['h', 'e', 'l' 'o'])

>>> z & "mello", z | "mello"
(Set:['e', 'l', 'o'], Set:['h', 'e', 'l', 'o', 'm'])
```

下面的类是对多个操作数扩展的答案。只需要代替两个方法。类的文档字符串解释了它的原理：

```
from set import Set

class MultiSet(Set):
    """
    扩展了交集和并集操作以支持多操作数（存储在*args参数里）；也要注意继承的&和
    |操作符用两个参数调用新的方法，但当需要处理多于两个操作数时，需要用函数调用方
    式而不是表达式：
    """

    def intersect(self, *others):
        res = []
        for x in self:                      # 扫描第一个序列
            for other in others:            # 所有别的参数
```

```
                if x not in other: break    # 成员在每一个序列里吗?
            else:                          # 否：跳出循环
                res.append(x)              # 是：添加一项
        return Set(res)

    def union(*args):                      # self是args[0]
        res = []
        for seq in args:                   # 所有参数
            for x in seq:                  # 所有成员
                if not x in res:
                    res.append(x)          # 把新成员加到结果中
        return Set(res)
```

假设新的集合存贮在一个模块 *multiset.py* 里：

```
>>> from multiset import *
>>> x = MultiSet([1,2,3,4])
>>> y = MultiSet([3,4,5])
>>> z = MultiSet([0,1,2])

>>> x & y, x | y                                  # 2个操作数
(Set:[3, 4], Set:[1, 2, 3, 4, 5])

>>> x.intersect(y, z)                             # 3个操作数
Set:[]
>>> x.union(y, z)
Set:[1, 2, 3, 4, 5, 0]

>>> x.intersect([1,2,3], [2,3,4], [1,2,3])        # 4 个操作数
Set:[2, 3]
>>> x.union(range(10))                            # 单个个集合也可以
Set:[1, 2, 3, 4, 0, 5, 6, 7, 8, 9]
```

6. **树的链接。**我们扩展了Lister类。

```
class Lister:
    def __repr__(self):
        return ("<Instance of %s(%s), address %s:\n%s>" %
                            (self.__class__.__name__,  # 我的类名
                             self.supers(),            # 我的超类
                             id(self),                 # 我的地址
                             self.attrnames()) )       # “属性名=值”列表
    def attrnames(self):
        未改变的...
    def supers(self):
        result = ""
        first = 1
        for super in self.__class__.__bases__:         # 类的上一级
```

```
            if not first:
                result = result + ", "          first = 0
            result = result + super.__name__
        return result
```

7. **合成。**

```
class Lunch:
    def __init__(self):
        # 生成嵌入的Customer和Employee
        self.cust = Customer()
        self.empl = Employee()
    def order(self, foodName):
        # 开始模拟Customer的购买
        self.cust.placeOrder(foodName, self.empl)
    def result(self):
        # 问Customer 有什么食品
        self.cust.printFood()

class Customer:
    def __init__(self):
        # 把我的食品初始化为None
        self.food = None
    def placeOrder(self, foodName, employee):
        # 向Employee买食品
        self.food = employee.takeOrder(foodName)
    def printFood(self):
        # 打印我的食品名
        print self.food.name

class Employee:
    def takeOrder(self, foodName):
        # 返回要求的食品
        return Food(foodName)

class Food:
    def __init__(self, name):
        # 保存食品的名字
        self.name = name

if __name__ == '__main__':
    x = Lunch()
    x.order('burritos')
    x.result()
    x.order('pizza')
    x.result()
```

```
% python lunch.py
burritos
pizza
```

第七章 异常

1. **try/except。**下面是我们的oops函数。把oops改为引发KeyError后，我们将不能捕获到这个异常（它将被传给Python的缺省异常处理）。KeyError和IndexError来自最外层的内置名的范围。如果你不相信我们，导入__builtin__，并用dir函数来显示它。

```
% cat oops.py

def oops():
    raise IndexError

def doomed():
    try:
        oops()
    except IndexError:
        print 'caught an index error!'    else:
        print 'no error caught...'

if __name__ == '__main__': doomed()

% python oops.py
caught an index error!
```

2. **异常列表。**这里是我们对这个模块的扩展，以支持新的异常。

```
% cat oops.py
MyError = 'hello'

def oops():
    raise MyError, 'world'

def doomed():
    try:
        oops()
    except IndexError:
        print 'caught an index error!'    except MyError, data:
        print 'caught error:', MyError, data
    else:
        print 'no error caught...'

if __name__ == '__main__':
```

```
    doomed()
```

```
% python oops.py
caught error: hello world
```

3. **错误处理。**最后，我们决定用一个文件来测试，而不是交互式的，但结果几乎是一样的。

```
% cat safe2.py
import sys, traceback
def safe(entry, *args):
    try:
        apply(entry, args)
    except:
        traceback.print_exc()
        print 'Got', sys.exc_type, sys.exc_value

import oops
safe(oops.oops)
```

```
% python safe2.py
Traceback (innermost last):
  File "safe2.py", line 5, in safe
    apply(entry, args)
  File "oops.py", line 4, in oops
    raise MyError, 'world'
hello: world
Got hello world
```

第八章 内置工具

1. **列出目录的内容。**这个练习有几种答案，一种较简单的答案是：

```
import os, sys, stat

def describedir(start):
    def describedir_helper(arg, dirname, files):
        """列出目录的辅助函数"""
        print "Directory %s has files:" % dirname
        for file in files:
            # 找出文件的完整路径(directory + filename)
            fullname = os.path.join(dirname, file)
            if os.path.isdir(fullname):
                # 如果是目录就不需要找出长度
                print ' ' + file + ' (subdir)'
```

```
            else:
                # 计算长度，并打印出来
                size = os.stat(fullname)[stat.ST_SIZE]
                print '   '+file+' size='   + `size`

    # 开始'walk'
    os.path.walk(start, describedir_helper, None)
```

使用os.path模块的walk函数，效果很好：

```
>>> import describedir
>>> describedir.describedir2('testdir')
Directory testdir has files:
  describedir.py size=939
  subdir1 (subdir)
  subdir2 (subdir)
Directory testdir\subdir1 has files:
  makezeros.py size=125
  subdir3 (subdir)
Directory testdir\subdir1\subdir3 has files:
Directory testdir\subdir2 has files:
```

你也许觉得可以用len(open(fullname, 'rb').read())来得出文件的长度，但这需要你对所有的文件有读权限，而这个条件是苛刻的。os模块的stat调用以一个元组的形式给出了各种有用的信息，而stat模块里定义的一些名字使你不需要记住元组里成员的次序。细节请参见库参考。

2. **修改提示符。**这个练习的关键是要记住，sys模块里的ps1和ps2属性可以是任何东西，甚至可以是一个有__repr__或__str__方法的类实例。例如：

```
import sys, os
class MyPrompt:
    def __init__(self, subprompt='>>> '):
        self.lineno = 0
        self.subprompt = subprompt
    def __repr__(self):
        self.lineno = self.lineno + 1
        return os.getcwd()+'|%d'%(self.lineno)+self.subprompt

sys.ps1 = MyPrompt()
sys.ps2 = MyPrompt('... ')
```

代码运行如下（使用解释器的-i选项以保证首先启动你的程序）：

```
h:\David\book> python -i modifyprompt.py
h:\David\book|1>>> x = 3
h:\David\book|2>>> y = 3
h:\David\book|3>>> def foo():
h:\David\book 3...    x = 3                    # 第二提示符是支持的
h:\David\book|3...
h:\David\book|4>>> import os
h:\David\book|5>>> os.chdir('..')
h:\David|6>>>                                  # 注意提示符改变了!
```

3. **避开正则表达式。**这个程序长而乏味，但并不特别复杂。看看你能否理解它。对你来说这是否比正则表达式简单依赖于很多因素，比如你对正则表达式以及 string 模块的熟悉程度。

```
import string
file = open('pepper.txt')
text = file.read()
paragraphs = string.split(text, '\n\n')

def find_indices_for(big, small):
    indices = []
    cum = 0
    while 1:
        index = string.find(big, small)
        if index == -1:
            return indices
        indices.append(index+cum)
        big = big[index+len(small):]
        cum = cum + index + len(small)

def fix_paragraphs_with_word(paragraphs, word):
    lenword = len(word)
    for par_no in range(len(paragraphs)):
        p = paragraphs[par_no]
        wordpositions = find_indices_for(p, word)
        if wordpositions == []: return
        for start in wordpositions:
            # 先寻找'pepper'
            indexpepper = string.find(p, 'pepper')
            if indexpepper == -1: return -1
            if string.strip(p[start:indexpepper]) != '':
                continue
            where = indexpepper+len('pepper')
            if p[where:where+len('corn')] == 'corn':
                # 后跟'corn'!
```

```
                continue
            if string.find(p, 'salad') < where:
                # 后面没有'salad'
                continue
            p = p[:start] + 'bell' + p[start+lenword:]
            paragraphs[par_no] = p

fix_paragraphs_with_word(paragraphs, 'red')
fix_paragraphs_with_word(paragraphs, 'green')

for paragraph in paragraphs:
    print paragraph+'\n'
```

我们这里就不重复它的输出了，输出与正则表达式的方法一样。

4. **用一个类打包一个文本文件。**如果你理解了类和string模块里的split函数，这个练习意外的简单。

```
import string

class FileStrings:
    def __init__(self, filename=None, data=None):
        if data == None:
            self.data = open(filename).read()
        else:
            self.data = data
        self.paragraphs = string.split(self.data, '\n\n')
        self.lines = string.split(self.data, '\n')
        self.words = string.split(self.data)
    def __repr__(self):
        return self.data
    def paragraph(self, index):
        return FileStrings(data=self.paragraphs[index])
    def line(self, index):
        return FileStrings(data=self.lines[index])
    def word(self, index):
        return self.words[index]
```

用它来处理文件*pepper.txt*，如下：

```
>>> from FileStrings import FileStrings
>>> bigtext = FileStrings('pepper.txt')
>>> print bigtext.paragraph(0)
This is a paragraph that mentions bell peppers multiple times.  For
one, here is a red Pepper and dried tomato salad recipe.  I don't like
to use green peppers in my salads as much because they have a harsher
flavor.
```

```
>>> print bigtext.line(0)
This is a paragraph that mentions bell peppers multiple times.  For
>>> print bigtext.line(-4)
aren't peppers, they're chilies, but would you rather have a good cook
>>> print bigtext.word(-4)
botanist
```

它是怎么工作的呢？构造函数简单的将所有文件都读入一个大的字符串（实例属性数据），并按不同类别把它分解，将分解结果存在实例属性（字符串列表）中。当从存取方法中返回时，数据本身被打包进一个FileStrings对象。这不是赋值所必需的，但这样你可以串联操作，以找到第三段第三行的最后一个词。这样写就行：

```
>>> print bigtext.paragraph(2).line(2).word(-1)
'cook'
```

第九章 常见任务

1. **重定向标准输出 stdout。**这很简单：

```
import fileinput, sys, string
sys.stdout = open(sys.argv[-1], 'w')          # 打开输出文件
del sys.argv[-1]                              # 我们已经处理过这个参数了
...
```

2. **写一个简单的shell。**下面的实现了部分Unix命令的脚本很可能是不用解释的。注意我们只给 ls 命令加了帮助，而别的命令也应该有帮助：

```
import cmd, os, string, sys, shutil

class UnixShell(cmd.Cmd):
    def do_EOF(self, line):
        """ The do_EOF command is called when the user presses Ctrl-D (unix)
            or Ctrl-Z (PC). """
        sys.exit()

    def help_ls(self):
        print "ls <directory>: list the contents of the specified directory"
        print "                (current directory used by default)"

    def do_ls(self, line):
        # 'ls' by itself means 'list current directory'
        if line == '': dirs = [os.curdir]
        else: dirs = string.split(line)
```

```
            for dirname in dirs:
                print 'Listing of %s:' % dirname
                print string.join(os.listdir(dirname), '\n')

        def do_cd(self, dirname):
            # 'cd' 意为 'go home'
            if dirname == '': dirname = os.environ['HOME']
            os.chdir(dirname)

        def do_mkdir(self, dirname):
            os.mkdir(dirname)

        def do_cp(self, line):
            words = string.split(line)
            sourcefiles,target = words[:-1], words[-1] # target 应该是一个目录
            for sourcefile in sourcefiles:
                shutil.copy(sourcefile, target)

        def do_mv(self, line):
            source, target = string.split(line)
            os.rename(source, target)

        def do_rm(self, line):
            map(os.remove, string.split(line))

    class DirectoryPrompt:
        def __repr__(self):
            return os.getcwd()+'> '

    cmd.PROMPT = DirectoryPrompt()
    shell = UnixShell()
    shell.cmdloop()
```

注意我们用了第八章练习2一样的技巧：通过修改cmd模块的prompt属性，让提示符显示当前目录。

```
h:\David\book> python -i shell.py
h:\David\book> cd ../tmp
h:\David\tmp> ls
Listing of .:
api
ERREUR.DOC
ext
giant_-1.jpg
icons
index.html
lib
pythlp.hhc
```

```
pythlp.hhk
ref
tut
h:\David\tmp> cd ..
h:\David> cd tmp
h:\David\tmp> cp index.html backup.html
h:\David\tmp> rm backup.html
h:\David\tmp> ^Z
```

当然，为了得到真正有用的脚本，需要增加一些错误检查和别的特征。

3. **理解map、reduce和filter。**下面的函数与我们介绍过的*map*、*reduce*和*filter*差不多，如果你想知道区别，请查阅参考手册。

```
def map2(function, sequence):
    if function is None: return list(sequence)
    retvals = []
    for element in sequence:
        retvals.append(function(element))
    return retvals

def reduce2(function, sequence):
    arg1 = function(sequence[0])
    for arg2 in sequence[1:]:
        arg1 = function(arg1, arg2)
    return arg1

def filter2(function, sequence):
    retvals = []
    for element in sequence:
        if (function is None and element) or function(element):
            retvals.append(element)
    return retvals
```

第十章 框架和应用

1. **欺骗Web。**你所要做的就是创建有相应字段名属性和实例变量的类的实例。下面是参考答案：

```
class FormData:
    def __init__(self, dict):
        for k, v in dict.items():
            setattr(self, k, v)
class FeedbackData(FormData):
```

```
    """ A FormData generated by the comment.html form. """
    fieldnames = ('name', 'address', 'email', 'type', 'text')
    def __repr__(self):
        return "%(type)s from %(name)s on %(time)s" % vars(self)

fake_entries = [
     {'name': "John Doe",
      'address': '500 Main St., SF CA 94133',
      'email': 'john@sf.org',
      'type': 'comment',
      'text': 'Great toothpaste!'},
     {'name': "Suzy Doe",
      'address': '500 Main St., SF CA 94133',
      'email': 'suzy@sf.org',
      'type': 'complaint',
      'text': "It doesn't taste good when I kiss John!"},
     ]

DIRECTORY = r'C:\complaintdir' if __name__ == '__main__':
    import tempfile, pickle, time
    tempfile.tempdir = DIRECTORY
    for fake_entry in fake_entries:
        data = FeedbackData(fake_entry)
        filename = tempfile.mktemp()
        data.time = time.asctime(time.localtime(time.time()))
        pickle.dump(data, open(filename, 'w'))
```

正如你所见，你需要做的唯一的事情就是改变FormData的构造函数的工作方式，因为它必须从一个字典设置属性，而不是FieldStorage对象。

2. **清除。**有很多解决这个问题的方式。一个较简单的办法是修改*formletter.py*程序，保存已处理的文件名的列表（用pickle!）。修改如下（粗体是新增加的行）：

```
if __name__ =  '__main__':
    import os, pickle
    CACHEFILE = 'C:\cache.pik'
    from feedback import DIRECTORY#, FormData, FeedbackData
    if os.path.exists(CACHEFILE):
        processed_files = pickle.load(open(CACHEFILE))
    else:
        processed_files = []
    for filename in os.listdir(DIRECTORY):
        if filename in processed_files: continue # skip this filename
        processed_files.append(filename)
```

```
        data = pickle.load(open(os.path.join(DIRECTORY, filename)))
        if data.type == 'complaint':
            print "Printing letter for %(name)s." % vars(data)
            print_formletter(data)
        else:
            print "Got comment from %(name)s, skipping printing." %\
                  vars(data)

pickle.dump(processed_file, open(CACHEFILE, 'w')
```

如你所见，你只是把已存在的列表装入（否则就用空列表），然后比较并决定是否跳过。如果你不跳过，就需要在列表里增加一项。最后在程序退出时，保存新的列表。

3. **给grapher.py增加绘图参数。**这个练习很简单，需要做的就是改变Chart类的绘画代码。特别是xmin, xmax=0, N-1和graphics.fillPolygon(…)之间的代码应该放置在一个if测试里，新代码如下：

```
if not hasattr(self.data[0], '__len__'): # 它很可能是一个数值 (1D)
    xmin, xmax = 0, N-1
                                         # 来自现有程序的代码，直到
                                         graphics.fillPolygon(xs, ys, len(xs))
elif len(self.data[0]) == 2:             # 我们只处理 2-D
    xmin = reduce(min, map(lambda d: d[0], self.data))
    xmax = reduce(max, map(lambda d: d[0], self.data))

    ymin = reduce(min, map(lambda d: d[1], self.data))
    ymax = reduce(max, map(lambda d: d[1], self.data))

    zero_y = y_offset - int(-ymin/(ymax-ymin)*height)
    zero_x = x_offset + int(-xmin/(xmax-xmin)*width)

    for i in range(N):
        xs[i] = x_offset + int((self.data[i][0]-xmin)/(xmax-xmin)*width)
        ys[i] = y_orfset - int((self.data[i][1]-ymin)/(ymax-ymin)*height)
    graphics.color = self.color
    if self.style == "Line":
       graphics.drawPolyline(xs, ys, len(xs))
    else:
       xs.append(xs[0]); ys.append(ys[0])
       graphics.fillPolygon(xs, ys, len(xs))
```

词汇表

API (Application Programming Interface)
　　应用程序编程接口

argument
　　参数

assignment
　　赋值

backtracking
　　反向跟踪

bind
　　绑定

built-in tool
　　内置工具

built-in type
　　内置类型

bytecode
　　字节码

CGI(Common Gateway Interface)
　　公共网关接口

class
　　类

cohesion
　　（模块）内聚

COM(Component Object Model)
　　组件对象模型

composition
　　合成

concatenation
　　合并

constructor
　　构造函数

couple
　　（模块）连接

delegation
　　授权

dictionary
　　字典

encapsulation
　　封装

exception
　　异常

factory
　　（类、对象的）工厂

file
　　文件

function
函数

global
全局的

GUI (Graphical User Interface)
图形用户界面

hash
散列

hierarchy
层次

hook
挂钩

identity
身份

import
导入

index
索引

inheritance
继承

instance
实例

introspection
内省

iterate
迭代

key
键

library
库

list
列表

local
局部的

map
映射

match
匹配

matching mode
匹配模式

metaclass
元类

metaprogram
元程序

method
方法

mix in
混入

module
模块

multiple inheritance
多重继承

namespace
名字空间

number
数字

object
对象

OOP (Object-Oriented Programming)
面向对象编程

operator
操作符

override
覆盖

package
包

polymorphism
多态

PSA (Python Software Association)
　　Python 软件协会
qualification
　　限定
raw string
　　原始字符串
reference
　　引用
regular expression
　　正则表达式
reload
　　重载
scope
　　作用域
sequence
　　序列
shell
　　命令解释程序
slice
　　分片
sort
　　排序
statement
　　语句
string
　　字符串
subclass
　　子类
superclass
　　超类
tuple
　　元组
widget
　　（窗口小）组件
wrapper
　　打包程序

PSA (Python Software Association)
　　Python 软件协会
qualification
　　限定
raw string
　　原始字符串
reference
　　引用
regular expression
　　正则表达式
reload
　　重载
scope
　　作用域
sequence
　　序列
shell
　　命令解释程序
slice
　　分片
sort
　　排序
statement
　　语句
string
　　字符串
subclass
　　子类
superclass
　　超类
tuple
　　元组
widget
　　（窗口小）组件
wrapper
　　打包程序

PSA (Python Software Association)
　　Python 软件协会
qualification
　　限定
raw string
　　原始字符串
reference
　　引用
regular expression
　　正则表达式
reload
　　重载
scope
　　作用域
sequence
　　序列
shell
　　命令解释程序
slice
　　分片
sort
　　排序
statement
　　语句
string
　　字符串
subclass
　　子类
superclass
　　超类
tuple
　　元组
widget
　　（窗口小）组件
wrapper
　　打包程序