



学习在线

视频资料下载
电子图书交流

www.eimhe.com



Core Python
Programming



软件开发技术丛书

Python

核心编程

(美) Wesley J. Chun (陈仲才) 著

杨涛 王建桥 杨晓云 高文雅 等译



机械工业出版社
China Machine Press

Prentice Hall
PTR



Core Python Programming

Python 核心编程

- Python核心概念
- Web编程、GUI开发的方法
- Python高级编程技巧
- 配套光盘包括书中实例代码



《Windows 核心编程》

ISBN 7-111-07942-0 TP·1427
定价：98.00元



《Java 编程思想》

ISBN 7-111-07864-1 TP·1391
定价：98.00元

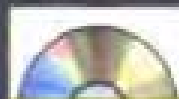


《C++ 编程思想》

ISBN 7-111-07116-7 TP·1928
定价：142.00元

- 《Windows 2000 编程技术内幕》
- 《Windows WDM 设备驱动程序开发指南》
- 《Visual C++ 6.0 编程实例与技巧》
- 《Visual C++ MFC 编程实例》
- 《Visual C++ MFC 扩展编程实例》
- 《MFC 开发人员指南》
- 《MFC Visual C++ 6 编程技术内幕》
- 《OpenGL 参考手册》(第3版)
- 《Delphi 5 编程实例与技巧》
- 《Delphi 5 企业级解决方案及应用剖析》
- 《Perl 5 编程详解》
- 《Java 2 核心技术 卷I: 基础知识》
- 《Java 2 核心技术 卷II: 高级特性》
- 《Java 2 图形设计 卷I: AWT》
- 《Java 2 图形设计 卷II: SWING》
- 《Java 2 类库》(增补版)
- 《Jini 核心技术》
- 《JSP 高级编程》
- 《CORBA 企业解决方案》
- 《C++Builder 5 编程实例与技巧》
- 《SQL Server 7 编程技术内幕》
- 《SQL-3 参考大全》
- 《JavaScript 技术大全》
- 《Python 核心编程》

适用水平：中、高级



附赠
CD-ROM

光盘中间附
赠简体中文版
Adobe Acrobat
Reader 4.0

ISBN 7-111-08983-9



学 苑 出 版 社

www.china-pub.com

北京市西城区百万庄南街1号 100037
购书热线：(010)68995265、8006100260 (北京地区)

ISBN 7-111-08983-9/TP·1939
定价：75.00元 (附光盘)

软件开发技术丛书

Python核心编程

(美) Wesley J. Chun (陈仲才) 著

杨 涛 王建桥 杨晓云 高文雅 等译



机械工业出版社
China Machine Press

Python是一种不复杂但很健全的编程语言。它不光具备传统编译型程序设计语言强大的功能和复杂性，还在某种程度上具备比较简单的脚本和解释型程序设计语言的易用性。该书向读者介绍了这种语言的核心内容，并展示了Python语言可以完成哪些任务。其主要内容包括：语法和编程风格、Python语言的对象、Web程序设计、执行环境等。该书条理清晰、通俗易懂，是学习Python语言的最好教材及参考手册。

所附光盘包括Python语言最新的三个版本及书中示例代码。

Wesley J.Chun: Core Python Programming.

Copyright © 2000 by Prentice Hall PTR.

All rights reserved.

Chinese simplified language edition published by China Machine Press.

Copyright © 2001 by China Machine Press.

本书中文简体字版由美国Prentice Hall公司授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

本书版权登记号：图字：01-2001-0493

图书在版编目(CIP)数据

Python核心编程/(美)魏斯理(Wesley, J. C.)著;杨涛等译. -北京:机械工业出版社, 2001.8

(软件开发技术丛书)

书名原文: Core Python Programming

ISBN 7-111-08983-9

I. P… II. ①魏… ②杨… III. Python语言-程序设计 IV. TP312

中国版本图书馆CIP数据核字(2001)第038081号

机械工业出版社(北京市西城区百万庄大街22号 邮政编码 100037)

责任编辑: 宋 宏 张鸿斌

北京昌平第二印刷厂印刷·新华书店北京发行所发行

2001年8月第1版第1次印刷

787mm×1092mm 1/16·33.25印张

印数: 0 001-5 000册

定价: 75.00元(附光盘)

凡购本书, 如有倒页、脱页、缺页, 由本社发行部调换

前 言

欢迎进入奇妙的Python世界！如果你是一名具备某种高级编程语言使用经验的职业程序员或者学生，本书将帮助你以最少的代价掌握Python语言。本书的目的是以轻松的交谈式风格和大量示例引导你逐步掌握Python语言的编程方法。

在本书出版的时候，Python 2.0刚刚发布，所以你拥有的极可能是最新和最好的东西。书后所附的光盘里有Python语言最新的三个版本，它们是：1.5.2版本、1.6版本和2.0版本，还包括有最新发布的Python语言解释器的Java版本——JPython。

本书的风格：技术化、但同时也很容易阅读理解

我的教学经验表明：把这本书写成一本严格意义上的“初学者”书籍，或者是一本硬邦邦的计算机科学方面的参考书并不是最好的办法，而编写一本容易阅读和理解，同时又面向技术的书是我们的目的，这能够帮助读者尽快掌握Python语言，并立刻把它运用到自己的实际工作中去。在学习过程中，我们将逐步介绍有关的概念，并且会给出一些适当的例子。在每一章的结尾部分都为读者准备了大量的练习，强化该部分内容中介绍过的概念和思路。

在对Python语言稍做介绍，并开始学习这个编程语言之前，我们将通过第2章“快速入门”使读者对Python语言先有一个基本的认识。这一章是为那些等不及按部就班的阅读学习，想立刻找找Python语言感觉的人们准备的。如果你不想走这条路，可以按照正常的方法从介绍Python语言中的对象的后续章节开始学习。随后三章介绍Python语言的基本数据类型：数值、字符串、列表、表列和字典等方面的内容。

Python的错误处理能力对程序员和用户都非常有用，而我们也将专门用一章的篇幅来讨论这个问题。最后，我们涉及到的Python语言的“核心”内容主要集中在函数（function）、模块（module）和类（class）方面，分别有相应的章节进行论述。本书的最后一章介绍了Python语言的扩展办法。

作者的Python语言经验

我是几年前在一个名为Four11的公司工作时开始接触Python语言的。当时该公司的一个主要产品是Four11.com政府机关目录服务。Python最初是用来设计名为Rocketmail的因特网电子邮件服务的，它最终演变成为今日的Yahoo!Mail。

除了C++之外，许多控制性的软件和网络上的前端软件都完全是用Python语言编写的。我参加了Yahoo!Mail的地址簿和拼写检查器的开发工作。从那时开始，Python语言的身影逐渐出现在Yahoo!的其他站点上，包括“People Search”（网上寻友）、“Yellow Pages”（电话黄页）、“Maps and Driving Directions”（地图和交通路线）等。

虽然Python语言对当时的我来说是一个新东西，但上手相当容易，和我以前学习过的其他编

程语言相比它也简单得多。因为当时非常缺少关于Python语言的教科书，我不得不使用其“Library Reference”（函数库大全）和“Quick Reference Guide”（快速参考指南）作为我的学习工具，而正是这一点成为我编写这本书的动机。

本书的内容

本书主要分为三个部分。第一部分占用了大约三分之二的篇幅，主要向读者介绍这种编程语言的核心内容；第二部分涉及到一系列高级论题，向读者展示了Python语言可以完成哪些工作；第三部分是附录。

Python可以说是无处不在，有时候你都想象不到有哪些人在使用它，而他们又都用它来干什么。虽然我很想再用一些篇幅来介绍Python在数据库（RDBMS关系数据库管理系统、SQL类数据库语言等）、应用CGI处理进程、XML、数学/科学计算、视频和图形图像、Zope等方面应用的内容，但实在是没有足够的时间对这些论题分别进行详细的讨论了。但是，令人欣慰的是我们至少能够向读者提供一些关于Python语言的许多关键性的发展方向的充分介绍。

下面是对本书各章内容的简单介绍。

第一部分：程序设计语言Python

第1章——欢迎使用Python 我们从向读者介绍Python语言入手，给出了它的历史、功能特色等，以及如何获得和安装Python语言的方法。

第2章——快速入门 如果你是一名有经验的程序员，并且想体验一下Python语言，就可以从本章入手。我们在此介绍了Python语言的基本概念和语句。因为其中大部分东西对你来说都应该是比较熟悉的，所以你不必花费太多的阅读时间，只需简单地学习一些Python语言的语法知识就可以立刻着手于自己的项目工作了。

第3章——语法和程序设计风格 这个部分向读者详细介绍了Python语言的语法和编程风格方面的注意事项，还介绍了Python语言的关键字和它的内存管理能力。在这一章的结尾部分给出了一个Python语言的应用程序，你可以通过它了解真实Python代码的样子。

第4章——对象 本章介绍Python语言的对象。除了基本对象属性之外，我们还在此介绍了Python语言所有的数据类型和操作符，并且向读者展示了标准数据类型的各种组织方法。同时在本章还涉及到适用于大多数Python对象的内建函数。

第5章——数字 Python语言有四种数值类型，它们是：正常或者说“普通”的整数、长整数、浮点实数和复数。本章介绍了这四种数值类的数据类型、操作符以及适用于数值类型数据的内建函数。

第6章——序列：字符串、列表和表列 这一章的内容相当丰富，它向读者揭示了Python语言强大的序列类数据类型：字符串、列表和表列。我们将向读者介绍适用于各数据类型和全体操作符的内建函数、使用方法以及特性。

第7章——字典数据类型 字典是Python语言中对映射或者哈希表数据类型的称呼。与其他的数据类型一样，字典也有它自己的操作符和适用的内建函数与使用方法。

第8章——条件语句和循环语句 类似于其他的高级编程语言，Python也支持诸如for和while

之类的循环语句以及if语句（及其相关语句）。Python语言中还有一个名为range()的内建函数，它使Python语言中的for循环语句更像是一个传统的计数类型的循环而不是它本身意义上的那种遍历类型的循环。

第9章——文件和输入/输出操作 本章介绍了标准的文件对象和输入输出操作，还介绍了文件系统访问、文件执行以及数据的固定存储等方面的内容。

第10章——错误和例外处理 Python语言中最为强大的结构之一就是它的例外处理能力。读者可以在这一章找到对此问题完整的解决办法、处理或者忽略例外情况的指导意见；更重要的是如何才能建立自己的例外处理手段。

第11章——函数 建立和调用函数相对来说是比较容易的，但Python语言还另外提供了许多有用的特性，比如默认参数、命名或者关键字参数、可变长参数以及函数性编程结构等。我们还将深入讨论变量的作用范围，并简单涉及一些递归调用方面的内容。

第12章——模块 Python语言的特色之一就是它的可扩展性。这个特色允许“即编即用”方式的程序设计，鼓励代码的重复使用。只用一条代码语句就可以把那些编写为模块的应用程序插入到其他Python模块中去使用。而多个模块软件构成的集合可以简单地通过软件包的方式加以组织。

第13章——类和OOP Python语言是一个完全面向对象的程序设计语言，并且是在它产生之初就这样设计的。因此，Python语言并不需要程序员特意编写面向对象的程序——你可以继续按照自己的想法编写结构化/过程化的代码，然后在必要的时候把它转化为“OO”（面向对象的）程序。另外，这一章的内容也对高级论题中的某些概念进行了介绍，比如操作符过载（operator overloading）、定制、程序优化等。

第14章——执行环境 “执行”这个术语可以有好几个意思，既可以表示可调用和可运行的对象，也可以表示运行其他的程序（Python程序或者其他语言编写的程序）。我们将在这一章里对这些方面进行讨论，另外，还将有限度地讨论受限制的执行情况以及终止程序执行的各种方法。

第二部分：高级论题

第15章——规则表达式 规则表达式是一个功能强大的工具，主要用途是模式匹配、内容摘取以及搜索与替换等功能。本章介绍的是这一方面的内容。

第16章——网络程序设计 如今的应用程序大部分都要面向网络，而读者也不能例外。在这一章里，你将学习如何通过TCP/IP和UDP/IP协议建立客户和服务端。

第17章——多线程程序设计 多线程程序设计是许多类型的应用程序改进其执行性能的有力武器。我们在这一章里将解释有关的概念，示范如何正确地建立一个Python多线程应用程序，以完整的书面文档形式向读者介绍如何在Python语言中进行多线程设计。

第18章——使用Tkinter进行GUI程序设计 Tkinter是以Tk图形工具库为基础的，它是Python语言缺省的GUI开发模块。在介绍Tkinter的时候我们将向你展示如何建立简单的GUI应用程序样本（比如说10次，确实够快！）。

第19章——Web程序设计 使用Python语言进行的Web程序设计主要有三种形式：Web客户、Web服务器和帮助Web服务器传输动态生成的Web主页的通用网关接口（Common Gateway

Interface) 应用程序。我们将在这一章里全面介绍这些内容: 简单和高级的Web客户和CGI应用程序, 以及如何建立自己的Web服务器。

第20章——扩展Python语言 我们在前面已经提到过代码复用和编程语言的可扩展能力会有多么大的作用。在纯粹的Python语言里, 这些扩展都是模块, 但你也可以通过C、C++或者Java语言开发出底层的代码, 再把这些代码接口到Python语言中去。

选读章节

本书的某些标题上标有一个星号(*), 它表示该部分是高级论题或者可以选读的内容, 读者可以跳过它们。它们的内容通常都是自成体系, 可以在今后有时间的时候再研究。

如果读者已经具备了足够的编程经验或者已经设置好了自己的Python开发环境, 可以略过最开始的第1章, 直接跳到第2章“快速入门”。

体例

Python解释器有用C语言和Java语言分别编写的两种。为了区别这两种解释器, 我们把用C语言编写的原始版本称为“CPython”, 把用Java语言编写的版本称为“JPython”。同时, 我们把“Python”定义为此编程语言的原始定义, 用CPython和JPython表示此语言两种具体的解释器实现。我们用“python”表示CPython的可执行文件名, 用“jpython”表示JPython的可执行文件名。

本书中所有的程序输出和源代码都以英文Courier字体出现。Python语言的保留字是加黑的英文Courier字体。以三个大于号(>>>)开始的输出行表示Python解释器的提示符。

在“编程提示”内容的旁边我们加上如右所示的一个标记。[CN]

在“编程风格”提示内容的旁边我们加上如右所示的一个标记。[CS]

在“模块”内容的旁边我们加上如右所示的一个标记。[CM]

Python语言中的新增功能旁边会出现几个标记[1.5.2]、[1.6]、[2.0], 标记中的数字表示的是该功能第一次出现在Python语言中的哪个版本。

联系信息

欢迎对本书提出任何反馈意见。如果读者有意见、建议、诀窍、投诉、程序缺陷、疑问……无论什么, 都可以通过cyberweb_consulting@yahoo.com与我联系。

在这本书的Web站点“Python Starship”上可以找到勘误表和其他资料, 该站点的地址是:
<http://starship.python.net/crew/wesc/cpp/>。

本书英文原书书名: Core Python Programming.

英文原书书号: ISBN 0-13-026036-3

英文原书出版社网址: www.phptr.com

参加本书翻译工作的人员除封面署名外还有: 陈文军、李龙浩、张向荣、肖志勇、段晓明、张玉亭、韩兰、李京山、林红、李崧、张亦涵、张玉乔、郭达生、李利、吴明、司晓东。

谨以此书献给我亲爱的外祖母，黄门周氏月桂。在工作和生活中，她的爱与关怀，是我努力的动力和源泉。在此书的中文版出版之即，我那已远在天国的外祖母却无缘赏阅。我希望用我们的努力表达对她的爱。

目 录

前言

第一部分 程序设计语言Python

第1章 欢迎使用Python	1
1.1 什么是Python语言	1
1.2 Python语言的历史	1
1.3 Python语言的特色	1
1.4 获得Python语言	5
1.5 安装Python语言	6
1.6 运行Python语言	7
1.6.1 命令行上的交互式解释器	7
1.6.2 命令行上的脚本程序	9
1.6.3 集成开发环境	10
1.7 Python语言的文档	13
1.8 Python与其他语言的比较	13
1.9 JPython的特色	14
1.10 练习	15
第2章 快速入门	16
2.1 程序输出、print语句和“Hello World!”	16
2.2 程序输入和raw_input()内建函数	17
2.3 程序注释	17
2.4 操作符	17
2.5 变量和赋值	19
2.6 数字	19
2.7 字符串	20
2.8 列表和表列	20
2.9 字典	21
2.10 代码段使用缩进	22
2.11 if语句	22
2.12 while循环	23
2.13 for循环和range()内建函数	23
2.14 文件和open()内建函数	25
2.15 错误和例外	26
2.16 函数	26
2.17 类	27
2.18 模块	29
2.19 练习	31
第3章 语法和程序设计风格	34
3.1 语句和语法	34
3.2 变量分配	36
3.3 标识符	38
3.4 程序设计风格准则	39
3.4.1 模块的结构和布局	40
3.4.2 在主体部分里加上测试方面的内容	42
3.5 内存管理	43
3.5.1 变量定义	43
3.5.2 动态确定变量的类型	43
3.5.3 内存分配	43
3.5.4 废弃物回收	44
3.5.5 引用计数	44
3.5.6 del语句	44
3.5.7 减少引用计数	45
3.6 第一个Python应用程序	45
3.7 练习	49
第4章 对象	50
4.1 Python语言中的对象	50
4.2 标准数据类型	51
4.3 其他内建的数据类型	51
4.3.1 type类型和type()内建函数	52
4.3.2 None空类型	52
4.4 内部数据类型	52
4.4.1 代码对象	53
4.4.2 框架对象	53
4.4.3 跟踪记录对象	53

4.4.4 序列切片对象	53	6.1.1 操作符	88
4.4.5 Ellipsis对象	54	6.1.2 内建函数	92
4.4.6 Xrange对象	54	6.2 字符串	93
4.5 与数据类型有关的标准操作符	55	6.3 字符串和操作符	94
4.5.1 值的比较	55	6.3.1 标准类型操作符	94
4.5.2 对象实体的比较	56	6.3.2 序列操作符	95
4.5.3 布尔表达式	58	6.4 只作用于字符串的操作符	99
4.6 与数据类型有关的标准内建函数	59	6.4.1 格式操作符 (%)	99
4.6.1 cmp()	59	6.4.2 生字符串操作符 (r/R)	102
4.6.2 str()和repr()	59	6.4.3 Unicode字符串操作符 (u/U)	103
4.6.3 深入type()	60	6.5 内建函数	103
4.7 标准数据类型的分类	63	6.5.1 标准类型函数	103
4.7.1 存储模型	64	6.5.2 序列类型函数	104
4.7.2 修改模型	64	6.5.3 字符串类型函数	104
4.7.3 访问模型	66	6.6 字符串的内建方法	104
4.8 Python语言不支持的数据类型	67	6.7 字符串的特性	107
4.9 练习	68	6.7.1 特殊或控制字符	107
第5章 数字	69	6.7.2 三引号	108
5.1 数字简介	69	6.7.3 字符串的不可变性	109
5.2 整数	69	6.7.4 Unicode支持	111
5.2.1 (普通)整数	70	6.7.5 Python语言没有字符或数组	112
5.2.2 长整数	70	6.8 相关模块	113
5.3 浮点实数	70	6.9 字符串总结	114
5.4 复数	71	6.10 列表	115
5.5 操作符	72	6.11 操作符	117
5.5.1 混状态操作符	72	6.11.1 标准类型操作符	117
5.5.2 标准类型的操作符	74	6.11.2 序列类型操作符	117
5.5.3 数值类型操作符	74	6.11.3 列表类型操作符	120
5.5.4 *位操作符	76	6.12 内建函数	120
5.6 内建函数	77	6.12.1 标准类型函数	120
5.6.1 标准类型函数	77	6.12.2 序列类型函数	121
5.6.2 数值类型函数	78	6.12.3 列表类型内建函数	123
5.6.3 只适用于整数的函数	82	6.13 列表类型的内建方法	123
5.7 相关模块	83	6.14 列表的特性	125
5.8 练习	85	6.14.1 利用列表创建其他数据结构	125
第6章 序列: 字符串、列表和表列	88	6.14.2 列表的子类	130
6.1 序列	88	6.15 表列	131

6.16 表列的操作符和内建函数	132	8.5 for语句	163
6.16.1 标准和序列操作符与内建函数	132	8.5.1 一般语法	163
6.16.2 表列类型操作符和内建函数及方法	133	8.5.2 与序列类型一起使用	163
6.17 表列的特性	133	8.5.3 switch/case语句的代理	164
6.17.1 不可变性对表列有何影响	133	8.5.4 range()内建函数	165
6.17.2 表列也不是绝对“不可变的”	133	8.6 break语句	167
6.17.3 括号的作用	134	8.7 continue语句	167
6.17.4 单元素表列	135	8.8 pass语句	168
6.18 相关模块	136	8.9 else语句之二	169
6.19 *浅拷贝与深拷贝	137	8.10 练习	170
6.20 练习	141	第9章 文件和输入/输出操作	173
第7章 字典数据类型	144	9.1 文件对象	173
7.1 字典简介	144	9.2 文件的内建函数	173
7.2 操作符	147	9.3 文件的内建方法	175
7.2.1 标准类型操作符	147	9.3.1 输入	175
7.2.2 字典的键字检索操作符[]	147	9.3.2 输出	175
7.3 内建函数	148	9.3.3 文件内移动	175
7.3.1 标准类型函数type()、str() 和cmp()	148	9.3.4 其他	176
7.3.2 映射类型函数len()	150	9.3.5 其他各种文件方法	176
7.4 内建方法	150	9.4 文件的内建属性	179
7.5 字典键字	152	9.5 标准文件	179
7.5.1 不允许一个键字对应一个以上的 数据项	153	9.6 命令行参数	180
7.5.2 键字必须是不可变的	153	9.7 文件系统	181
7.6 练习	156	9.8 文件的执行	186
第8章 条件语句和循环语句	159	9.9 永久性存储模块	186
8.1 if语句	159	9.9.1 pickle和marshal模块	187
8.1.1 多重条件表达式	159	9.9.2 DBM风格的模块	187
8.1.2 单语句子句	159	9.9.3 shelve模块	187
8.2 else语句	159	9.10 相关模块	189
8.3 elif语句	161	9.11 练习	190
8.4 while语句	161	第10章 错误和例外处理	193
8.4.1 一般语法	161	10.1 什么是例外	193
8.4.2 计数循环	162	10.1.1 错误	193
8.4.3 无限循环	162	10.1.2 例外	194
8.4.4 单语句子句	163	10.2 Python语言中的例外	194
		10.3 检测和处理例外	196
		10.3.1 try-except语句	196

10.3.2 打包一个内建函数	197	11.5.1 位置参数	232
10.3.3 带多个except的try语句	199	11.5.2 缺省参数	232
10.3.4 处理多个例外的except语句	200	11.6 可变长参数	235
10.3.5 不带例外名参数的try-except语句	201	11.6.1 非关键字可变长参数	235
10.3.6 例外参数	202	11.6.2 关键字可变长参数	236
10.3.7 把打过包的函数用在一个应用程 序里	204	11.6.3 调用带有可变长参数对象的函数	238
10.3.8 else从句	206	11.7 函数化的程序设计	239
10.3.9 try-except语句用法总结	206	11.7.1 匿名函数和lambda	239
10.3.10 try-finally语句	207	11.7.2 内建函数: apply()、filter()、map()、 reduce()	241
10.4 *例外的字符串形式	209	11.8 变量的作用范围	253
10.5 *例外的类形式	210	11.8.1 全局变量和局部变量的比较	254
10.5.1 通过对象的标识符进行挑选	211	11.8.2 global语句	255
10.5.2 例外之间的关系	211	11.8.3 作用范围到底有几个	255
10.6 引发例外	212	11.8.4 作用范围的其他特性	256
10.7 确认	214	11.9 *递归	257
10.8 标准例外	215	11.10 练习	258
10.9 *创建例外	216	第12章 模块	261
10.10 为什么会发生例外	222	12.1 什么是模块	261
10.11 为什么要有例外	222	12.2 模块和文件	261
10.12 例外和sys模块	223	12.2.1 名字空间基本概念	261
10.13 相关模块	224	12.2.2 搜索路径和路径搜索	262
10.14 练习	224	12.3 名字空间	263
第11章 函数	226	12.3.1 名字空间与变量作用范围的比较	264
11.1 什么是函数	226	12.3.2 名字的查找、确定作用范围和覆盖	264
11.1.1 函数与过程的比较	226	12.4 导入模块	265
11.1.2 返回值和函数类型	226	12.4.1 模块加载时的执行情况	265
11.2 函数的调用	228	12.4.2 导入与加载的比较	266
11.2.1 函数操作符	228	12.5 导入模块属性	266
11.2.2 关键字参数	228	12.5.1 把名字导入当前名字空间	266
11.2.3 缺省参数	228	12.5.2 被导入到导入者作用范围的名字	266
11.3 函数的创建	229	12.6 模块的内建函数	267
11.3.1 def语句	229	12.6.1 __import__()	267
11.3.2 函数声明与函数定义的比较	229	12.6.2 globals()和locals()	268
11.3.3 向前引用	229	12.6.3 reload()	268
11.4 函数可以用做其他函数的参数	230	12.7 软件包	269
11.5 正式参数	231	12.7.1 目录结构	269

12.7.2 软件包的from-import语句操作	270	13.10.3 对标准类型进行推导	303
12.8 模块的其他特性	270	13.10.4 多重继承	304
12.8.1 自动加载模块	270	13.11 类、实例和其他对象的内建函数	304
12.8.2 阻止某个属性的导入	271	13.11.1 issubclass()	304
12.9 练习	271	13.11.2 isinstance()	305
第13章 类和OOP	272	13.11.3 hasattr()、getattr()、setattr()、delattr()	307
13.1 简介	272	13.11.4 dir()	308
13.2 面向对象的程序设计	277	13.11.5 vars()	308
13.2.1 OOD和OOP之间的联系	278	13.12 类型和类/实例的比较	309
13.2.2 现实世界中的问题	278	13.13 用特殊方法对类进行定制	310
13.2.3 抽象世界里的模型	279	13.13.1 对类进行简单定制的例子	312
13.3 类	281	13.13.2 *对类进行较复杂定制的例子	314
13.3.1 类的创建	282	13.14 私密性	319
13.3.2 声明和定义的比较	282	13.15 对类型进行打包	319
13.4 类的属性	282	13.15.1 打包	319
13.4.1 *类的数据属性	283	13.15.2 实现对类型进行的打包	320
13.4.2 方法	283	13.16 相关模块和文档	326
13.4.3 确定类的属性	284	13.17 练习	328
13.4.4 类的特殊属性	285	第14章 执行环境	333
13.5 实例	286	14.1 可调用对象	333
13.5.1 实例化: 调用类对象创建实例	287	14.1.1 函数	334
13.5.2 __init__() 构造器方法	287	14.1.2 方法	335
13.5.3 __del__() 拆除器方法	288	14.1.3 类	337
13.6 实例的属性	289	14.1.4 类的实例	338
13.6.1 “实例化”实例的属性	289	14.2 代码对象	338
13.6.2 确定实例的属性	291	14.3 可执行对象语句和内建函数	339
13.6.3 实例的特殊属性	292	14.3.1 callable()	339
13.6.4 内建类型的属性	293	14.3.2 compile()	340
13.6.5 实例属性和类属性的比较	293	14.3.3 eval()	341
13.7 绑定和方法的调用	295	14.3.4 exec	341
13.7.1 调用绑定方法	296	14.3.5 input()	346
13.7.2 调用未绑定方法	297	14.3.6 内置字符串和intern()	347
13.8 构造	298	14.4 执行其他Python程序	348
13.9 子类的分离和推导	299	14.4.1 导入	348
13.10 继承性	300	14.4.2 execfile()	349
13.10.1 类属性__bases__	301	14.5 执行其他非Python程序	349
13.10.2 通过继承覆盖掉方法	302		

14.5.1 os.system()	351	16.3.5 执行TCP客户-服务器应用程序	397
14.5.2 os.popen()只适用于UNIX和 Windows	351	16.3.6 创建一个UDP服务器	398
14.5.3 os.fork()、os.exec*()、os.wait*() 只适用于UNIX	352	16.3.7 创建一个UDP客户	399
14.5.4 os.spawn*()只适用于Windows	354	16.3.8 执行UDP客户-服务器应用程序	400
14.5.5 其他函数	354	16.3.9 其他socket模块函数	401
14.6 受限执行环境	354	16.4 相关模块	401
14.7 中断程序的执行	357	16.5 练习	402
14.7.1 sys.exit()和SystemExit	357	第17章 多线程程序设计	404
14.7.2 sys.exitfunc()	358	17.1 介绍	404
14.7.3 os._exit()函数	359	17.2 线程和进程	405
14.8 相关模块	359	17.2.1 什么是进程	405
14.9 练习	360	17.2.2 什么是线程	405
第二部分 高级论题		17.3 线程和Python	406
第15章 规则表达式	361	17.3.1 全局性解释器锁	406
15.1 介绍与动机	361	17.3.2 退出线程	407
15.2 规则表达式使用的特殊符号和字符	363	17.3.3 从Python访问线程	407
15.3 规则表达式和Python语言	368	17.3.4 不使用线程时的程序设计情况	407
15.3.1 re模块的核心函数和方法	368	17.3.5 Python语言中的线程化模块	408
15.3.2 re模块的其他函数和方法	375	17.4 thread模块	409
15.4 规则表达式的使用示例	379	17.5 threading模块	412
15.5 练习	385	17.5.1 Thread类	413
第16章 网络程序设计	388	17.5.2 菲波那契数列、阶乘、连加和	418
16.1 介绍	388	17.5.3 制造商-消费者问题和Queue模块	419
16.1.1 什么是客户-服务器体系结构	388	17.6 练习	422
16.1.2 客户-服务器网络程序设计	390	第18章 使用Tkinter进行GUI程序设计	423
16.2 套接字: 通信端点	391	18.1 介绍	423
16.2.1 什么是套接字	391	18.1.1 什么是Tel、Tk和Tkinter	423
16.2.2 套接字地址: 主机加端口	391	18.1.2 安装Tkinter并使它工作	423
16.2.3 面向连接方式和无连接方式	392	18.1.3 再论客户-服务器体系结构	424
16.3 使用Python语言进行网络程序设计	393	18.2 Tkinter和Python程序设计	424
16.3.1 socket()模块函数	393	18.2.1 Tkinter模块: 把Tk添加到应用 程序中去	424
16.3.2 套接字对象的内建方法	393	18.2.2 GUI程序设计简介	425
16.3.3 创建一个TCP服务器	394	18.2.3 顶层窗口: Tkinter.Tk()	426
16.3.4 创建一个TCP客户	396	18.2.4 Tk素材	426
		18.3 Tkinter程序示例	427
		18.3.1 Label素材	427

18.3.2 Button素材	428	操作和文件的上传	464
18.3.3 Label和Button素材	429	19.6.2 多取值输入域	465
18.3.4 Label、Button和Scale素材	430	19.6.3 cookie	465
18.3.5 中规模Tkinter程序示例	431	19.6.4 高级CGI实战	466
18.4 相关模块和其他GUI	436	19.7 Web服务器	474
18.5 练习	436	19.8 相关模块	477
第19章 Web程序设计	438	19.9 练习	478
19.1 介绍	438	第20章 扩展Python语言	483
19.1.1 网上冲浪: 客户-服务器计算	438	20.1 介绍	483
19.1.2 因特网	439	20.1.1 什么是扩展	483
19.2 用Python网上冲浪: 编写简单的Web 客户	441	20.1.2 为什么要扩展Python语言	483
19.2.1 统一资源定位器	441	20.2 用编写扩展的办法扩展Python语言	484
19.2.2 urlparse模块	442	20.2.1 编写应用程序代码	484
19.2.3 urllib模块	443	20.2.2 给代码加上程序接口	486
19.3 高级Web客户	446	20.2.3 编译	490
19.4 CGI: 帮助Web服务器处理客户数据	451	20.2.4 引用的计数	494
19.4.1 CGI简介	451	20.2.5 线程化和GIL方面的考虑	495
19.4.2 CGI应用程序	452	20.3 相关论题	495
19.4.3 cgi模块	452	20.4 练习	496
19.5 建立CGI应用程序	453		
19.5.1 制作结果网页	453		
19.5.2 制作表单和结果主页	456		
19.5.3 完全以交互方式运行的Web站点	459		
19.6 高级CGI	464		
19.6.1 包含多个组成部分的表单的提交			

第三部分 附 录

附录A 部分练习答案	497
附录B 参考信息	503
附录C Python操作符汇总	511
附录D Python版本2.0的新增功能	513

第一部分 程序设计语言Python

第1章 欢迎使用Python

我们将在这一章里介绍一些有关Python的背景知识，它是从何产生的，它的优势在哪里。在激起读者的兴趣和热情之后，我们将向大家介绍如何获取Python语言以及如何把它安装在自己的系统上。最后，本章末尾处的练习将使读者进一步掌握Python语言在交互式解释器和创建运行脚本程序两方面的用法。

1.1 什么是Python语言

Python是一种不复杂但是很健壮编程语言，它既具备传统编译型程序设计语言强大的功能和复杂性，又在某种程度上具备比较简单的脚本和解释型程序设计语言的易用性。熟悉和掌握Python语言的速度之快以及使用这种语言能够完成的事情之多都足以令读者感到惊讶，更不用说那些已经有现成解决方案的事情了。可以说，人有多大胆，Python语言就有多高产。

1.2 Python语言的历史

Python语言的开发工作由Guido van Rossum开始于1989年的下半年，接下来转移到荷兰的CWI公司，并最终于1991年初公开发表。这一切到底是如何开始的呢？一种程序设计语言的发明通常会归结到两个动机之一：一是有一个资金充裕的大型研发项目；二是因为缺乏某些软件工具而造成的困境，人们需要有新的工具来完成当时那些枯燥或者耗时的工作，而这些工作大部分又都是能够自动化完成的。

van Rossum是一名研究人员，对解释性程序设计语言ABC有丰富的实际经验，这种语言也是由CWI开发的。van Rossum不满足于ABC在软件开发能力方面的局限性，因为他准备开发的工具里有一些是为了完成系统管理方面某些一般性的任务的，因此他希望能够获取Amoeba机操作系统所提供的系统调用的能力。虽然开发一种Amoeba专用的语言也算是一个思路，但一种通用性的程序设计语言无疑是更明智的，到了1989年末，Python语言的种子发芽了。

1.3 Python语言的特色

虽然Python语言已经出现了10年多，但相对于软件开发工业来说仍然是比较新的。我们在使用“相对”这个词的时候必须注意这样一个事实：在“因特网时代”进行开发时，几年往往就像几十年一样。

当人们问到“什么是Python语言？”时，用一种东西来说明它是很困难的。人们更倾向于——

口气说出自己对Python语言的全部体会。Python语言是（请填空）。这个空格里可以有哪些答案呢？答案有许多。

1. 它是一种高级程序设计语言

程序设计语言每次更新换代都使我们进入更高一级。汇编语言是那些用机器码打天下的人们的随身利器；随后出现了FORTRAN、C和Pascal等语言，它们都把计算任务带到一个更高的水平，并且开创出软件开发业来。这些语言又演化为如今的编译性系统设计语言C++和Java。再向上就是Tcl、Perl和Python等功能强大、能够进行系统调用的解释性脚本程序设计语言了。这些语言中的每一种都具有更高级的数据结构，大大减少了曾经是不可或缺的“程序框架”（framework）的开发时间。Python语言还建立了更有效的数据类型，比如列表（list，即可变数组）和字典（hash table，即哈希表）等。提供了这些关键性的建筑材料就等于是鼓励人们使用它们，同时还能够减少软件开发时间和代码长度，最终使程序更容易阅读和理解。如果要用C语言来实现它们，就会弄得很复杂，大量的结构和指针会把人搞得头晕眼花，更不用说每一个大型项目上都要对某些基本相同的数据结构进行大量重复性的实现工作了。这个问题在C++和使用程序模板（template）之后有所缓解，但是仍然有许多与应用程序本身没有直接联系的工作需要开发完成。

2. 它是一种面向对象的程序设计语言

面向对象的程序设计（object-oriented programming, OOP）为结构化和过程化程序设计语言增添了新的活力，这些语言中的数据和逻辑关系都是程序设计中不可分割的元素。OOP允许把特定的行为、特性和/或功能与将要处理的数据或者它们所代表的数据关联在一起。Python语言面向对象的特性是与生俱来的。其他面向对象的脚本语言还包括SmallTalk、施乐公司早期的PARC语言的后代以及网景公司的JavaScript等。

3. 它是一种适应性很强的程序设计语言

人们通常会把Python语言与批处理或者Unix操作系统的shell脚本语言相提并论。简单的shell脚本程序处理的是简单的任务。它们在长度上可以（无限制地）增长，但是在功能方面总是有一个极限。shell脚本程序的代码很少能够重复使用，因此也就把你限制在小型项目上。而事实上，即使是一个小型的项目也有可能导致尺寸庞大而又不知所云的脚本程序。Python语言就不会出现这样的状况，你可以根据项目的不同来增减自己的代码，添加新的或者现有的Python元素，按照自己的想法重复使用编写好的代码。Python语言鼓励简洁的代码设计风格、高水平的结构设计、把多个组件“打包”到一起等等，这些特点能够有效地保证软件开发项目在广度和范围方面延伸时所要求的灵活性、一致性和更快的开发时间。

“适应性”这个术语往往用来衡量硬件的处理能力，它指的是系统添加了新的硬件设备后所增加的性能。在此需要把这个说法与我们的意思区分一下，我们是说Python语言有这样的一个特性：它向用户提供用来创建一个应用程序的建筑材料，在需要扩展和增加时，Python的即插即用及模块化体系结构能够适应用户项目的扩展，并保持项目的可管理性。

4. 它是一种可扩展的程序设计语言

一个项目的Python代码会不断增长，但是因为其双重的结构化和面向对象的编程环境，我们仍然能够对它进行有效的组织和管理。更好的办法是可以把代码划分为多个文件，或者说多个

“模块”，并且能够在一个模块里访问另外一个模块的代码和属性。Python语言中用来访问模块的语法对所有的模块都是相同的，不管是访问标准Python库还是访问用户一分钟之前刚建立的模块都没有差别。这个特性给人的感觉就好像是用户可以根据自己的需要对这种语言进行扩展一样，而事实也正是如此。

代码中最关键的部分——比如说资料处理过程中经常出现的热点或者那些必然会执行到的代码部分——最适合作为语言扩展的候选。而通过Python接口对底层代码进行打包则更利于用户建立“已编译”模块，而这些打包操作将要用到的接口与纯Python模块所使用的完全一样。代码和对象的访问方法是一模一样的，根本不需要对代码进行任何修改。从代码方面来说，需要注意的唯一区别就是代码执行性能上的改善。很自然，这将取决于用户的应用程序以及它对资源的要求情况。把应用程序中的瓶颈转换为编译后的代码肯定是有好处的，因为它会决定性地改善应用程序的整体性能。

程序设计语言中这类可扩展性使工程师在添加或者定制自己的工具时有很大的灵活性，这可以使那些工具更有效率，开发周期也会大大缩短。虽然主流的第三代程序设计语言（third-generation languages, 3GL）如C、C++、甚至是Java中都已经具备了这个特性，但是在脚本程序设计语言中还是不多见的。除了Python以外，目前脚本程序设计语言中真正具备可扩展性的只有“Tool Command Language”（工具命令语言，TCL）。Python语言的扩展在CPython方面可以用C或者C++语言来编写；在JPython方面可以用Java来编写。

5. 它是一种可移植的程序设计语言

能够运行Python语言的计算机平台范围相当广泛（请参考1.4节内容），而这导致了它在如今的计算领域里快速的发展和壮大。事实上，因为Python是用C语言编写的，再因为C语言的可移植性，所以Python能够工作在具有C语言编译器和通用操作系统接口的任何类型的系统上。

虽然会有一些针对某种系统特别开发的模块，但在一种系统上开发的通用Python应用程序只需很少或者不需要修改就能够运行在另外一种系统上。它的可移植性可以跨越多种体系结构和多种操作系统。

6. 它是一种易于学习的程序设计语言

相对来说，Python语言中的保留字是比较少的，程序结构比较简单，语法定义得也比较明确清晰。这就使学生们能够在一个相对较短的时间里掌握这门语言。因为没有完全陌生的概念或者不熟悉的保留字和语法，也就不会把功夫额外耗费在学习这些东西上面。对初学者来说，唯一的新东西可能就是Python语言的面向对象的特性了。那些还没有完全掌握面向对象的程序设计（object-oriented programming, OOP）方法的人们可能对使用Python语言有些犹豫，但OOP既不是必须的，也不是强制性的。入门很容易，你可以在做好准备之后再学习和使用OOP。

7. 它是一种易于阅读理解的程序设计语言

Python语言的语法与其他语言有一个非常明显的差异，那就是没有其他语言中通常用来完成存取变量、定义代码段、查找匹配模式等操作所使用的符号。这些符号常见的有：美元符号（\$）、分号（;）、波浪号（~）等。没有了这些符号，Python代码的定义更加整齐，也更适合于阅读。此外，与其他语言相比，让许多程序员沮丧（或者欣慰）的是：不太容易用Python语言写出重叠嵌套的复杂语句，这使其他人能够很容易读懂你写的程序，你也很容易读懂其他人写的程序。

如果一种语言比较易于阅读，通常也意味着它比较易于学习，我们刚才已经提到了这一点。我们甚至可以大胆地声明：即使有人一行Python代码也没有读过，也很容易看懂Python程序。看看下一章“快速入门”中的示例，告诉我们你的感受。

8. 它是一种易于维护的程序设计语言

维护源代码也是软件开发周期的一个组成部分。在被替换或者过时之前，你的软件不可能老是改来改去的；而我几乎可以肯定地说你的代码会比你在目前职位上待的时间要长。Python的成功很大一部分原因是它的源代码很容易维护，当然也要视长度和复杂性而言。但得出这个结论并不难，因为Python既易于学习也易于阅读理解。Python语言另外一个好处是：阅读6个月前写的某个脚本程序的时候，不太容易把自己搞糊涂，一般也不需要依靠参考书才能读懂自己写的软件。

9. 它是一种健壮的程序设计语言

没有什么能够比允许程序员识别出某些错误条件并且在这些个错误发生时提供一个软件句柄（software handler）更有效果的了。Python在错误发生时提供了“安全且理智”的退出机制，使程序员能够控制住局面。如果是因为致命错误而退出时，Python会通过一个完整的堆栈跟踪记录指明什么地方出现了错误，错误又是怎样发生的。Python错误会产生“例外”（exception），堆栈跟踪记录会指出例外出现的名称和类型。Python还向程序员提供了识别例外并采取相应的必要措施的功能。这些“例外句柄”（exception handler）可以用来编写例外发生时采取的措施，在体面地结束应用程序之前可以采取忽略该错误、重定向程序流、执行清理操作以及其他补救措施。无论哪一种情况，开发周期中的调试纠错工作都会大大减轻，因为Python具备帮助程序员迅速查找问题发生的位置，而不仅仅依靠程序员个人的捉虫能力。Python的健壮性对软件的设计人员和用户都有好处。如果某些特定的错误发生后处理不当，它还具备一定的统计定位功能。作为某个错误的结果而生成的堆栈跟踪记录功能不仅能够揭示该错误的类型和位置，还能够指出错误代码处于哪个模块中。

10. 它是一种非常有效的快速建模工具

我们已经在前面提到过Python语言非常容易学习和阅读理解。但读者会说BASIC语言也是这样的呀，Python语言又多些什么？Python语言与那些自我包容和较少灵活性的编程语言不一样，它有许多联系其他系统的不同接口，在功能上足够强大，也足够健壮，因此即使单独使用Python语言也完全可以建立起一个系统的整个模型。当然，用传统的编译型语言也能够实现同样的系统模型，但Python语言在工程方面的简单性使我们在完成同样的工作时游刃有余。此外，人们已经为Python语言开发出许许多多外挂的开发库，因此无论你的应用程序准备干什么用，都可能已经有人沿着那条路走过了。你只需要发挥一下拿来主义，来个即插即用就可以了（当然要自行调配一番）。这些外挂的开发库涉及的范围包括：网络、因特网Web/CGI、图形和图形化用户接口（graphical user interface, GUI）的开发（Tkinter）、图象处理（PIL）、数学计算和分析（NumPy）、数据库处理、超文本（HTML、XML、SGML等）、操作系统扩展、音频/视频、程序开发工具等等。

11. 它是一个内存管理器

使用C或C++语言编写程序一个最大的弊病是内存管理是由软件开发人员负责的。即使是一

个很少涉及内存访问、内存修改和内存管理的应用程序，程序员也必须在基本任务的基础上尽到为之管理内存使用情况的义务。这就给开发人员增添了不必要的负担和责任，也往往会因此而影响质量。

但在Python中就不同了，因为内存管理是由Python语言的解释器来完成的，所以应用程序的开发人员可以撇开内存方面的问题，专心按事先计划编写应用程序。而这将导致更少的程序错误、更健壮的应用程序、更短的整体开发时间。

12. 它既可以解释运行也可以（字节）编译运行

Python语言被归类为解释型语言，也就是说在开发阶段已经没有编译时间这个因素了。因为传统纯解释型语言的程序在执行时不使用计算机系统专用的二进制语言，所以它们几乎都要比编译型语言慢。但Python与Java很相似，它实际上是字节编译的，其结果是生成一种近似于机器代码的中间形式。这改善了Python的性能，而同时又使它能够保持解释型语言的各种优点。

编程提示：文件扩展名

[CN]

使用Python编写的源代码文件一般都以扩展名.py结尾。源代码会在被解释器加载到内存中去的时候进行字节编译，也可以单独进行字节编译。根据解释器被调用的方式，它可能会给经过字节编译的文件加上一个.pyc或.pyo扩展名。在第12章“模块”里有更多关于文件扩展名的内容。

1.4 获得Python语言

正如我们前面提到过的，Python可以运行的计算机平台相当广泛：

- Unix (Solaris、Linux、FreeBSD、AIX、HP/UX、SunOS、IRIX等)
- Win 9x/NT/2000 (32位的Windows系统)
- Macintosh (PPC、68K)
- OS/2
- DOS (多种版本)
- Windows 3.x
- PalmOS
- Windows CE
- Acorn/RISC OS
- BeOS
- Amiga
- VMS/OpenVMS
- QNX
- VxWorks
- Psion

Python语言目前共有三种最新的版本。版本1.5.2是最稳定的版本，至少发布于一年半以前。

Python 1.6公开发布于2000年的9月初，它以版本1.5系列为基础增加和改进了一些功能。但版本1.6被普遍认为是Python语言最新的版本2.0之前的一个过渡，它发表于2000年10月的中旬。那么到底用哪个版本好呢？答案取决于你的需要和期望。

如果读者不需要那些花哨的新功能，只希望有最好的稳定性、代码能够向下兼容旧版本（并且与现在安装的Python兼容）、能够运行的计算机平台种类最多，那版本1.5.2无疑是最好的选择。

对所有的新项目来说，如果不需要向下兼容其旧版本或早期Python语言，希望或者需要使用诸如Unicode支持等关键性新功能，更不用说那些希望享受Python语言最新最前沿的技术了，这样的情况应该从版本2.0起步。

版本1.6是那些从版本1.5.2向版本2.0过渡的人们的第二选择，但是没有什么推荐意义，因为它只是早一个月推出的最新版Python语言罢了。

获得Python语言的方法

要想获取最新的源代码、二进制代码、文档、新闻等，请访问Python语言的主站点或PythonLabs公司的Web站点，如下所示：

<http://www.python.org>（社团主页）

<http://www.pythonlabs.com>（商业主页）

如果读者目前还不能连接上因特网，可以在书后所附的CD-ROM光盘里找到所有这三个版本（源代码和二进制代码）。这张CD-ROM光盘上还有完整的已经设置为能够脱机浏览在线文档或能够安装到硬盘上去的档案文件。本书所有的代码示例和在线资源附录（都是些Python热点）部分的内容也都在光盘上。

1.5 安装Python语言

如果有对应于某种计算机平台的二进制安装代码，只要把文件下载到机器里再开始安装操作过程就可以了。如果没有与读者计算机平台相对应的二进制版本的话，就需要你自行获取和编译其源代码了。编译源代码并不像它一开始看起来时那样糟，手工自行建立二进制代码可以提供最大的灵活性。

你可以对准备加到解释器中的各种功能进行取舍。你的可执行文件越小，它加载和执行的速率就越快。比如说，如果使用的是UNIX系统，就可能需要安装“GNU readline”模块，这个模块允许你利用Emacs或者Vi的编辑键查看以前的Python命令，允许查找、访问和修改以前输入过的命令。其他比较流行的取舍选择还包括加上Tkinter以建立GUI应用程序、加上线程开发库以建立多线程应用程序等。我们在这里介绍的各种取舍选择都可以通过修改源代码发行版本中的Modules/Setup文件来实现。

建立自己的Python解释器一般要经过以下几个步骤：

- 下载和释放文件，对制作文件进行定制（如果需要的话）。
- 运行./configure脚本程序。
- make。
- make install。

Python通常会被安装到一个标准位置，以便于用户找到它。在UNIX机器上，它的可执行文件通常被安装在/usr/local/bin子目录里；库文件通常被安装在/usr/local/python1.x子目录里，其中的“1.x”是你正在使用的Python的版本。

在DOS和Windows机器上，Python通常会被安装到C:\Python或者C:\Program Files \Python子目录中去。因为DOS不支持像“Program Files”这样的长文件名，所以它会用别名“Progra~1”代替之；在Windows系统上的DOS窗口里必须使用这个短名字才能进入Python语言所在的子目录。标准的库文件通常被安装到C:\Program Files\Python\Lib子目录中去。

1.6 运行Python语言

启动Python有三种办法。最简单的办法是交互式地启动解释器进行操作，每次输入一行Python语句去执行。另外一个办法是运行一个用Python语言编写的脚本程序，它会调用解释器去执行那个脚本应用程序。最后，我们可以从一个集成开发环境（integrated development environment, IDE）里通过一个图形化用户接口（GUI）运行之。IDE通常都额外带有调试器（debugger）和文本编辑器等工具。

1.6.1 命令行上的交互式解释器

从命令行启动交互式解释器进入Python后马上就能开始编写代码。在UNIX、DOS或者其他任何提供有一个命令行解释器或shell窗口的系统中都能这么做。学习Python最好的办法之一就是交互式地运行其解释器。如果读者想试验一下Python语言中的某些功能，交互模式也很有用。

1. UNIX

如果没有把Python所在的子目录加到搜索路径中去，那就必须输入它的完整路径名才能启动Python。Python一般安装在/usr/bin或者/usr/local/bin子目录中。

我们建议读者把Python（即可执行文件python，或者对应于Java版解释器的jpython）加到自己的搜索路径中去，这样就不必在每次交互运行它的时候输入一大串路径名了。这个操作完成后，只需输入解释器的名字就可以在任意位置启动它了。

把Python添加到搜索路径中去的具体办法是：检查系统开机时执行的启动脚本程序，找到以“set path”或者“path=”开始、后面跟着一串子目录名的那一行；然后把Python所在子目录的完整路径加上，再刷新一下shell的路径变量就行了。现在，在UNIX的提示符（根据shell的不同可能是“%”或“\$”）处敲入名字“python”（或者“Jpython”）再回车就可以启动解释器了，如下所示：

```
% python
```

Python启动时先显示解释器的启动信息，给出它的版本和运行平台名称，然后显示一个解释器提示符“>>>”等待输入Python命令。图1-1是在一个UNIX环境中启动Python时的屏幕画面。

2. DOS

要想在DOS中把Python添加到自己的搜索路径中去，需要编辑C:\autoexec.bat文件，把安装有解释器的子目录名称添加上去。它一般会是C:\Python或者C:\Program Files \Python（或者它在DOS中的短名字C:\Progra~1\Python）。在一个DOS窗口里（它可以是纯DOS环境或者Windows中

启动的一个DOS窗口)启动Python的命令与UNIX操作系统是一样的,都是“python”;它们唯一的区别是提示符有所不同,DOS中的是C:\>。如下所示:

1.1 > python



图1-1 在一个UNIX (Solaris) 窗口里启动Python

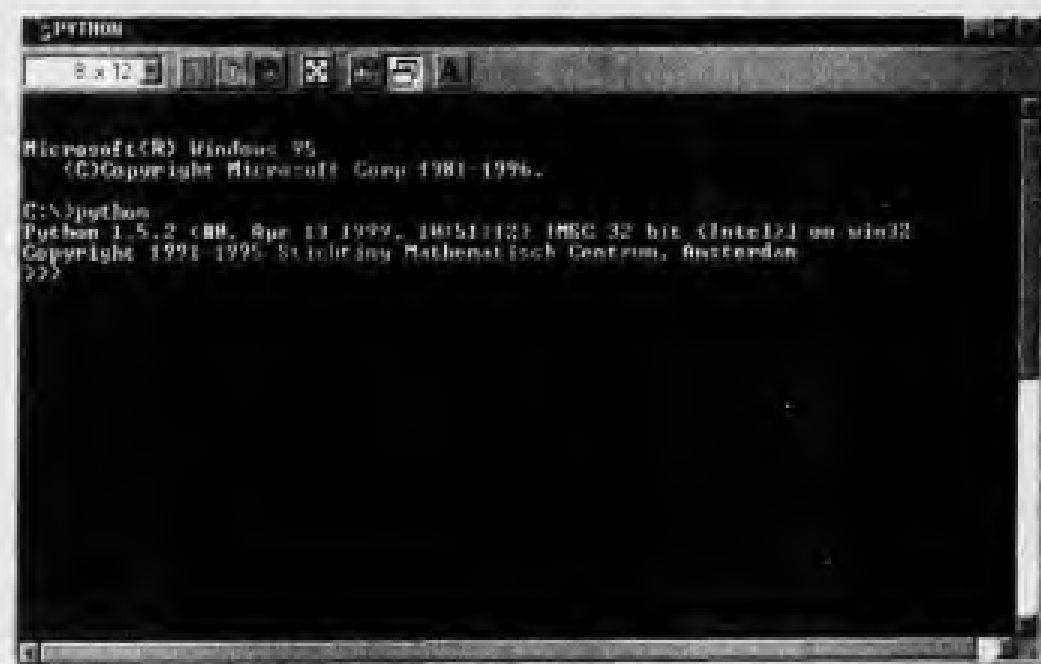


图1-2 在一个DOS窗口里启动Python

3. 命令行参数选项

在命令行上启动Python的时候可以给解释器加上几个参数选项。下面列出一些可供选用的参数选项:

- d 提供程序调试输出。

- O 生成优化的字节代码（会生成一个.pyo文件）。
- S 不在启动时用“run site”命令查找Python的路径。
- v 详细执行情况输出（对import语句的详细跟踪输出）。
- X 禁用基于类（class）的内建例外检查（只使用字符串）；从版本1.6起已不再必要。
- c cmd 执行cmd字符串给出的Python脚本程序。
- file 执行file给出的文件中的Python脚本程序（请参考下一小节）。

1.6.2 命令行上的脚本程序

在UNIX、DOS或者其他具有命令行操作界面的版本中，可以通过解释器直接执行一个Python脚本程序，如下所示：

```
C:\> python script.py
unix% python script.py
```

大多数Python脚本都是以文件扩展名.py结尾的，如上所示。

如果是在UNIX操作系统中，还有一个办法可以不必明确地在命令行上给出Python解释器就自动调用它。不管读者使用的是哪一种UNIX家族中的系统，都可以在自己程序的第一行使用一条shell引导语句，如下所示：

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

跟在“#!”后面的文件路径必须是Python解释器的完整路径。我们已经在前面讲过，它通常会被安装在/usr/local/bin或者/usr/bin子目录中。如果不是这样，一定要给出正确的路径名才能执行Python脚本程序。不正确的路径名会让你看到熟悉的“Command not found”（命令未找到）错误信息。

此外，许多UNIX系统还有一个名为env的命令，一般安装在/bin或者/usr/bin子目录里，它会在用户路径中查找Python解释器。如果你有env命令，可以把脚本程序的头一行修改为类似于下面这样的语句：

```
# ! //usr/bin/env python
```

如果读者不知道Python的可执行文件被保存在什么位置，或者它的保存位置总在变化——但还没有跑出你的子目录路径，就非常适合使用env命令。当把适当的启动引导语句加到脚本程序的第一行之后，它就真的成为可直接执行的文件了。这类可执行文件在执行时会先加载Python解释器，再运行脚本程序。正如我们在前面已经提到的，不必在命令行上明确地给出Python，只需像下面这样输入脚本程序的名字就可以了：

```
unix% script.py
```

注意，必须先把文件的权限设置为可执行。在文件长列表里对应于用户本人的权限必须是“rwx”标志。如果在查找Python的安装位置、设置文件权限或者使用chmod命令（用来改变文件权限的命令）时需要帮助，可以找你的系统管理员。

DOS不支持自动启动机制；但在Windows中可以使用它的“文件类型”的接口。这个文件类型的接口允许Windows通过文件的扩展名识别出一个文件的类型，再调用预先定义与之关联的

程序对它进行处理。举例来说,如果你通过PythonWin安装好Python(请参考下一小节),在一个带.py扩展名的Python脚本程序上双击鼠标就会启动Python或者PythonWin IDE(如果你安装了的话)来运行你的脚本程序。

1.6.3 集成开发环境

Python也可以从一个图形化用户接口(GUI)环境中运行,其前提只需要系统中有一个支持Python的GUI应用软件。如果读者找到一个这样的应用软件,那它极有可能同时也是一个IDE(integrated development environment,集成开发环境)。IDE可不仅仅是图形化的操作界面,它们通常都还带有源代码编辑器和跟踪调试工具。

1. UNIX

[1.5.2]

IDLE是UNIX中最早出现的一个Python语言的集成开发环境。它也是由Guido开发的,最早出现在Python版本1.5.2中。IDLE代表“Integrated DeveLopment Environment”,意思和IDE一样,只是多了个字母“L”。但Monty Python公司里恰好有位老哥的名字也叫IDLE,嘿嘿...(idle在英文里有懒惰的意思)IDLE是基于Tkinter的,因此要想使用它系统中必须安装有Tcl/Tk。如今的Python版本里都带有一个Tcl/Tk库的最小发行子集,这样就不需要一个完整安装了。

你可以在源代码发行版本里的Tools子目录里找到idle可执行文件。Windows上也有相应的Tk工具箱,所以Windows平台和Macintosh平台上也有与之对应的IDLE。UNIX操作系统中的IDLE屏幕画面如图1-3所示。

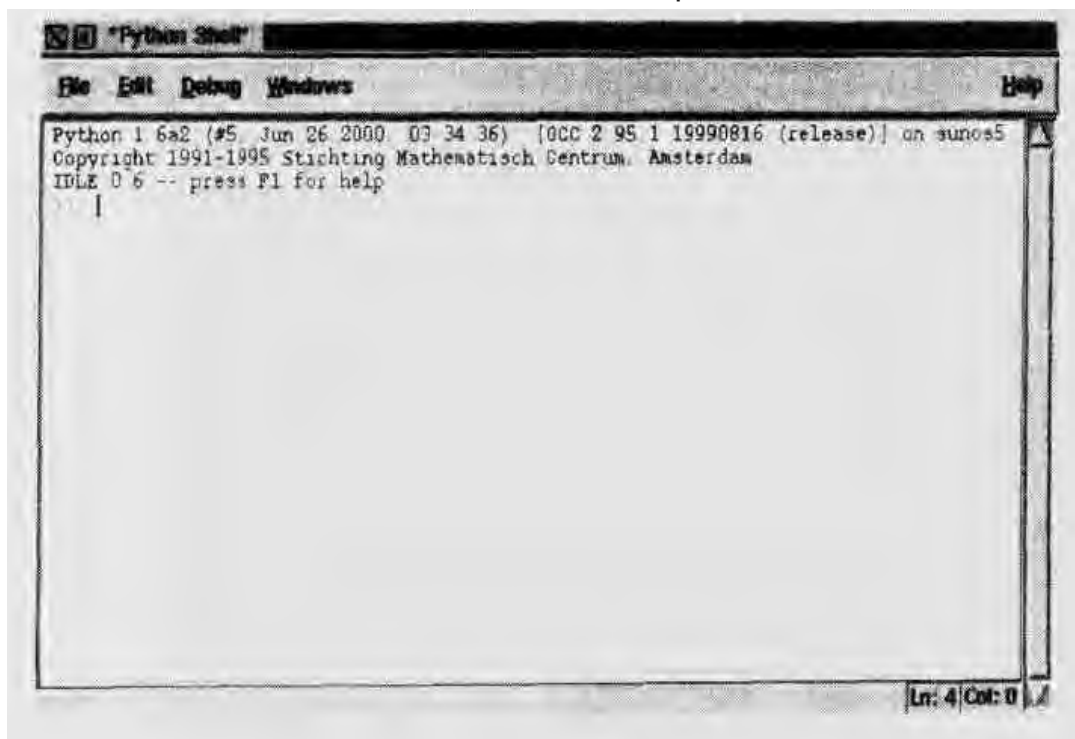


图1-3 在UNIX操作系统中启动IDLE

2. Windows

PythonWin是最早出现的一个Python语言的Windows接口,它是一个GUI操作界面的集成开

发环境。PythonWin的发行版本中还包括一个Windows应用程序接口（API）和COM（Common Object Model，通用对象模型，包括OLE [Object Linking and Embedding，对象链接和嵌入]和ActiveX）扩展。PythonWin本身是用MFC（Microsoft Foundation Class，微软基础类）库编写的，读者可以把它做为一个建立自己的Windows应用程序的开发环境。

PythonWin通常和Python安装在同一个子目录里，它自己的下级子目录是C:\Program Files\Python\PythonWin，可执行文件是pythonwin.exe。PythonWin有一个彩色的编辑器，一个经过改进的新调试器，一个交互式的shell窗口以及COM扩展等。运行在一台Windows机器上的PythonWin集成开发环境的屏幕画面如图1-4所示。



图1-4 Windows中的PythonWin环境

这个软件安装好以后，启动网络浏览器指向下面的地址（或者PythonWin的安装位置）可以查看到更多有关的文档：

```
file: //C:/Program Files/Python/PythonWin/readme.html
```

我们已经在前面提到过，因为Tcl/Tk和Python/Tkinter的可移植性，IDLE也有对应于Windows平台的版本。它与UNIX操作系统中的同类看起来很相似（如图1-5所示）。

Windows中的IDLE可以在Python解释器所在子目录的Tools\idle下级子目录找到，它一般会是C:\Program Files\Python\Tools\idle。在DOS窗口里启动IDLE的办法是调用脚本程序idle.py。idle.py也可以在Windows环境中调用，但这样会打开一个并不必要的DOS窗口，因此最好是双击idle.pyw文件。

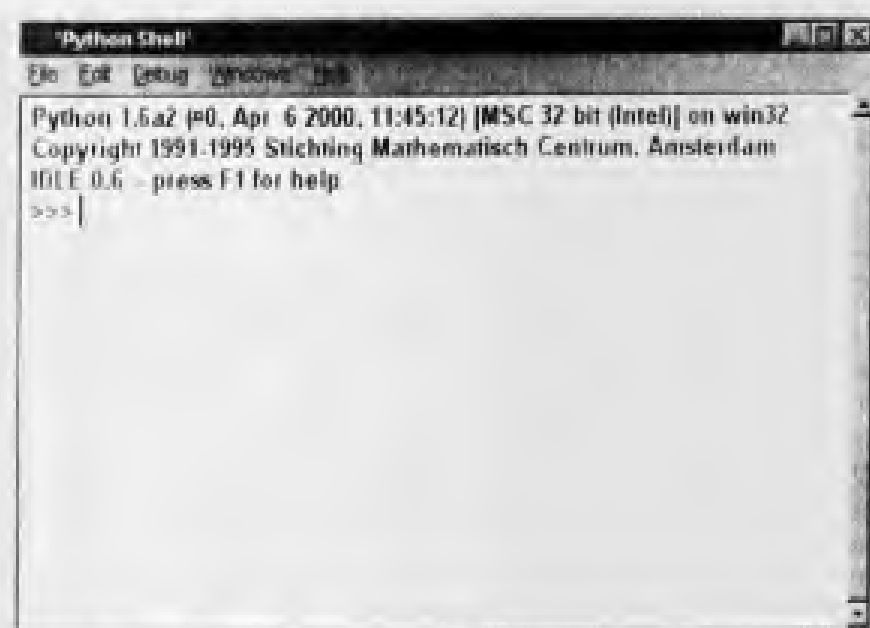


图1-5 在Windows中启动IDLE

3. Macintosh

Macintosh机器上实现的Python语言环境叫做MacPython，比较主要的网站上都有，下载文



图1-6 在MacPython中运行IDE

件是MacBinary或BinHex'd格式的。Python的源代码是一个Stuff-It档案文件。这个发行版本中带有在PowerPC和Motorola 68K系列机器上运行Python所需要的全部软件。MacPython包括一个集成开发环境、Python的数学计算模块(NumPy)、各种图形处理模块以及随软件包而来的Tk窗口应用工具箱,因此IDLE也可以工作在Mac机器上。图1-6就是MacPython环境的屏幕画面,画面中打开着一个编辑Python脚本的文本窗口和一个正运行着Python解释器的“shell”窗口。

1.7 Python语言的文档

读者需要用到的大多数Python文档都可以在书后的CD-ROM光盘或者其主站点上找到。可供下载的文档分别有可打印格式、超文本HTML文件以及在线(或脱机)浏览格式等。

如果下载的是Python语言的Windows版本,其发行版本中就有有一个HTML文档的安装选项了。如果你想把HTML文件安装到自己的Python子目录,一定要选中“Help Files”(帮助文件)单选框。在安装过程完成之后,就可以用网络浏览器来查阅那些Python文档了,具体办法是把链接指向如下所示的地址或该解释器被安装到的其他位置:

```
file: //C: /Program Files/Python/Doc/index.html
```

本书的附录里有一个Python语言可打印文档和在线文档的详细清单。

1.8 Python与其他语言的比较

Python语言能够与许多语言相提并论。原因之一是它具备其他语言中的许多功能。另外一个原因是Python语言本身就是从其他语言中提炼出来的,这些语言包括有ABC、Modula-3、C、C++、Algol-68、SmallTalk、UNIX操作系统的shell以及其他脚本类程序设计语言等等。van Rossum把他学过的其他程序设计语言中他最佩服的功能特色都集中到一起并体现在Python语言里,可以说是集程序设计之大成。

但是,因为Python终究只是一种解释性的程序设计语言,所以读者会发现它与其他语言的比较集中在Perl、Java、Tcl、和JavaScript等上面。Perl也是一种脚本程序设计语言,就它本身来说已经超越了普通shell脚本程序的范畴。类似于Python, Perl也向我们大家提供了一个完备的程序设计语言的功能和系统调用访问手段。

Perl语言最大的长处是其字符串模式的匹配能力,而这又来源于它有一个功能异常强大的规则表达式匹配引擎。这使Perl语言俨然成为对字符串文本流进行过滤、识别和抽取等操作的标准语言,而它又是开发经Web服务器通用网关接口(Common Gateway Interface, CGI)实现的因特网应用软件的最流行的程序设计语言。但Perl语言模糊且过于符号化的语法太晦涩难懂了,学习曲线陡得让初学者望而生畏,难住不少的人。再加上Perl语言提供了多种方法来完成同一项工作,使开发人员不易保持连贯性和统一性。最后,即使是阅读一个几分钟之前编写的Perl脚本程序,也往往不得不求助于参考书。

最经常用来与Python进行比较的是Java,因为二者都具备面向对象的特性,语法结构也差不多。Java的语法虽然比C++的要简单,但仍相当复杂,在只想完成一个很小的任务时更是如此。Python语言的简单性为我们提供了一个比只使用纯Java快得多的开发环境。Python语言与Java语言的关系最终演化为JPython的出现,它是完全用Java编写的Python语言的解释器。这使我们能

够在只有Java虚拟机 (virtual machine, VM) 的情况下运行Python程序。我们将在后续章节里简要介绍更多JPython的优点, 但我们现在可以告诉大家的是: 在JPython脚本环境中, 你可以操纵Java对象, Java又可以与Python对象交互作用, 而且你还能够正常访问各种Java语言的类 (class) 库——就好像Java就是Python语言的一个组成部分那样。

Tcl是另外一个与Python语言有几分相似的脚本程序设计语言。Tcl语言具备程序员扩展能力, 并且能够访问系统调用, 这使它成为第一个真正易于使用的脚本程序设计语言。Tcl到现在仍十分流行, 与Python相比可能更局促一些 (因为它在数据类型方面的局限性), 但它与Python一样具备能够超越其原始设计水平的扩展能力。更重要的是, Tcl通常是与其图形化工具包Tk一起被用来开发具有图形化用户操作界面 (GUI) 的应用软件的。因为它被广泛接受, 所以Tk也被引入到Perl语言 (Perl/Tk) 和Python语言 (Tkinter) 中来了。

Python稍微具备一些函数化程序设计 (functional programming, FP) 的结构, 这使它又与Lisp和Scheme等语言有几分相似。但Python并没有被认为是一种FP语言, 因此除了这几分相似外它没有多提供什么东西。

在与Python进行比较的语言当中, JavaScript与它最接近。它们在语法方面最为相似, 并且都是面向对象的。任何一个合格的JavaScript程序员都会发现学习Python根本不费什么力气。Python能够在Web浏览器环境以外执行, 也具备与系统调用交互作用和执行通常由shell脚本程序完成的一般系统级任务的能力。

在下面的地址可以找到许多对Python与其他语言加以比较的资料:

<http://www.python.org/doc/Comparisons.html>

1.9 JPython的特色

正如我们已经在前面提到过的, 如今已经有了一个完全用Java编写的Python语言的解释器, 它就是JPython。虽然两种解释器之间有很细微的差异, 但它们非常相似, 并且启动环境也没有什么差别。

JPython到底有什么优点呢? JPython...

- (几乎) 能够在任何能够找到Java虚拟机的地方运行。
- 能够访问Java软件包及其类 (class) 库。
- 为Java开发提供了一个脚本环境。
- 能够对Java语言的类库方便地进行测试。
- 满足面向对象的程序设计环境。
- 继承并发扬了JavaBeans的特点和思路。
- 鼓励用Python对Java进行开发 (或者反过来)。
- 使GUI开发人员能够访问Java语言的AWT/Swing库。
- 使用了Java语言的本地废弃物收集器 (native garbage collector), 而在CPython上没有此项功能。

对JPython的详细论述超出了本书的讨论范围, 但网上有大量与此方面内容有关的资料。JPython目前仍是一个还在开发中的项目, 请大家拭目以待它的新功能吧。

1.10 练习

1-1 **安装Python**。下载Python软件或者从CD-ROM光盘上加载它，再把它安装到自己的系统中去。

1-2 **执行Python**。有多少种办法可以运行Python？

1-3 **Python的标准库**。

a) 找出Python的可执行文件和标准库模块都安装在自己系统的什么位置。

b) 查看几个标准的库文件，比如string.py。这会帮助你熟悉用Python语言编写的脚本程序。

1-4 **交互式执行**。启动Python语言的交互式解释器，具体做法是输入它完整的路径名；如果你已经把它存放地址添加到搜索路径中，也可以只输入它的名字（python或者python.exe）。（如果愿意，也可以使用由Java语言编译的Python解释器[jpython或者jpython.exe]。）启动画面看上去应该和这一章里介绍过的一样。如果看到“>>>”提示符，就表示解释器已经准备好接受你的Python命令了。

试着输入print “Hello World!”，这个命令就是著名的“Hello World!”（世界你好）程序，然后退出解释器。在UNIX系统上，Ctrl-D组合键将送出EOF信号终止Python解释器的运行；在DOS系统上，这个组合键是Ctrl-Z。如果想退出图形化用户环境中的窗口——如Macintosh、Windows上的PythonWin或IDLE、或者UNIX操作系统上的IDLE，简单地关闭与之对应的窗口就可以了。

1-5 **编写脚本程序**。做为练习1到4的补充，用一个Python脚本程序来完成与上面交互式执行练习中显示“Hello World!”一模一样的事情。如果你使用的是UNIX系统，最好加上自动引导语句，这样不调用Python解释器也可以运行这个程序。

1-6 **编写脚本程序**。用print语句编写一个脚本程序来显示你的姓名、年龄、喜欢的颜色以及简单的自我介绍（背景、兴趣、爱好等）。

第2章 快速入门

本章既然取名为快速入门，目的就是要把Python的主要特色介绍给读者，使其能够根据以往的编程经验认出熟悉的语法结构，把它们迅速应用到自己的工作中。具体细节将在随后的各个章节里逐一介绍，但本章内容将是读者快速熟悉Python语言以及了解它所提供的功能的一个快速而又简单的办法。最好的办法是运行Python解释器，试试本章给出的一些示例，同时也可以编几个小程序练练手。

我们在第一章以及最后的练习里（练习1-4）介绍了如何启动Python语言的解释器。在所有交互使用的示例中，读者都可以发现Python的主提示符（>>>）和辅提示符（...）。解释器通过其主提示符告诉你它正准备接收下一个Python语句，而辅提示符表示解释器正等待输入更多的数据使当前语句成为一个完整的语句。

2.1 程序输出、print语句和“Hello World!”

软件开发老手无疑已经做好准备来看看著名的“Hello World!”程序，程序员在接触一种新语言时通常会把这个程序做为第一个试验品。本章也不例外。

```
>>> print 'Hello World!'
Hello World!
```

print语句用来在屏幕上显示输出结果。熟悉C语言的读者应该知道C语言的printf()语句也是用来产生屏幕输出的。其他一些shell脚本语言使用echo命令显示程序输出。

编程提示：在交互式解释器中查看变量的内容

[CN]

一般情况下，如果想查看对某个变量的赋值，我们会在代码中使用print语句。但在交互式的解释器里，我们可以用print语句查看赋值给某个变量的字符串内容，或者只是查看变量本身的值——这只需给出变量的名字即可。

在下面的例子里，我们先对一个字符串变量进行赋值，再用print语句显示出它的内容；然后单独输入变量名看看会出现什么情况。

```
>>> myString = 'Hello World!'
>>> print myString
Hello World!
>>> myString
'Hello World!'
```

请注意，单独使用变量名将在字符串的首尾加上一对单引号。其原因是为了使非字符串对象也能够以与该字符串同样的显示方式显示在屏幕上——即能够显示代表任何对象的可打印字符，而不仅仅是字符串。此处的单引号表示我们刚才显示在屏幕上的对象的值是

一个字符串。

我们要告诉大家的是：Python语言中的print语句及字符串格式操作符(%)与C语言中printf()语句的功能非常相似。如下所示：

```
>>> print " %s is number %d ! " % ( ' Python ', 1 )
Python is number 1!
```

有关字符串格式及其他操作符的情况请阅读6.4节。

2.2 程序输入和raw_input() 内建函数

从命令行获取用户输入最简单的办法是使用raw_input() 内建函数。它从标准输入读取数据，再把读到的字符串值赋值给指定的变量。读者可以用int()内建函数（版本1.5之前的Python使用string.atoi()函数）把输入的数值从字符串格式转换为整数形式。

```
>>> user = raw_input('Enter login name: ')
Enter login name: root
>>> print 'Your login is:', user
Your login is: root
```

上面的例子只限于文本形式的输入。下面是一个数值字符串的输入（并转换为一个真正的整数）示例：

```
>>> num = raw_input('Now enter a number: ')
Now enter a number: 1024
>>> print 'Doubling your number: %d' % (int(num) * 2)
Doubling your number: 2048
```

int()函数把字符串num转换为一个整数，这也是我们为什么要在字符串格式操作符里加上%d（表示对整数进行操作）操作符的原因。

关于raw_input() 内建函数的详细介绍请阅读6.5.3节。

2.3 程序注释

在大多数脚本程序设计语言和UNIX的shell语言里，井字符(#)表示从这个符号出现开始一直到该语句行结尾的文字将是一个程序注释。

```
>>> # one comment
... print 'Hello World!' # another comment
Hello World!
```

2.4 操作符

人们熟悉的数学运算符在Python语言中与它们在其他语言中的作用是完全一样的。这些数学运算符包括：

+ (加) - (减) * (乘) / (除) % (取除法余数) ** (乘方)

加、减、乘、除、取除法余数都是标准的数值运算符。Python语言额外准备了一个用两个星号(**)代表的乘方运算符。虽然我们强调的是这些运算符的数学特性，但是一定要注意这些运算符中有些还可以用在其他数据类型上，比如字符串和列表等。

```
>>> print -2*4+3**2
```

```
1
```

我们可以从上面的例子看出，各种运算符的优先级正是我们所预期的：加减法的优先级最低，然后是乘除和取除法余数，再往后是正负单元操作符，最后是顶部的乘方运算（在上面的例子里，3**2最先被计算，然后是-2*4，两项结果再相加）。

编程风格：使用括号

[CS]

根据Python语言对数学运算符运算顺序的优先级定义，上面print语句中的例子完全是一个合法的数学表达式，但好的程序设计风格要求我们适当地加上一些括号把想要进行的计算明确地一组一组地表示出来（请参考本章最后的练习）。维护你程序代码的那些人将会为此感谢你，你也会感谢自己的。

Python语言还提供了标准的比较运算符，如下所示：

< (小于) <= (小于等于) > (大于) >= (大于等于) == (等于) != (不等于) <> (不等于)

下面是一些比较运算的例子：

```
>>> 2 < 4
```

```
1
```

```
>>> 2 == 4
```

```
0
```

```
>>> 2 > 4
```

```
0
```

```
>>> 6.2 <= 6
```

```
0
```

```
>>> 6.2 <= 6.2
```

```
1
```

```
>>> 6.2 <= 6.20001
```

```
1
```

Python语言目前支持两个“不等于”操作符，一个是“!=”，另一个是“<>”。两种符号分别对应于C语言风格和ABC/Pascal语言风格，但使用后者的人在慢慢减少，因此我们建议尽量不要使用它。

Python语言还提供了表达式连接操作符，如下所示：

and (与) or (或) not (非)

我们可以通过这些操作符和括号把多个比较运算“链”在一起，如下所示：

```
>>> (2 < 4) and (2 == 4)
```

```
0
```

```
>>> (2 > 4) or (2 < 4)
```

```
1
```

```
>>> not (6.2 <= 6)
```

```
1
```

```
>>> 3 < 4 < 5
```

```
.
```

最后一个例子中的表达式在其他语言中可能是非法的，但在Python语言里它只是下面意思的简短形式：

```
>>> (3 < 4) and (4 < 5)
```

关于Python语言中的操作符的详细内容请阅读4.5节。

2.5 变量和赋值

Python语言中与变量有关的规则与其他高级语言一样：它们只是以一个字母打头的标识符名字而已；打头字母可以大写，也可以小写，并包括下划线（“_”）；随后的字符可以是字母、数字、或者下划线。Python语言是大小写敏感的，也就是说标识符“cAsE”和“CaSe”是不同的。

在Python语言中，变量类型是动态定义的，也就是说对变量及其类型的预定义是不必要的。变量的类型（及其赋值）是在赋值操作时确定的。赋值操作用等号来实现。如下所示：

```
>>> counter = 0
>>> miles = 1000.0
>>> name = 'Bob'
>>> counter = counter + 1
>>> kilometers = 1.609 * miles
>>> print '%f miles is the same as %f km' % (miles, kilometers)
1000.000000 miles is the same as 1609.000000 km
```

我们在上面给出了五个赋值语句示例。第一个是一个整数赋值语句，接下来依次是一个浮点数、一个字符串、一个整数递增语句，最后是一个浮点运算和赋值语句。

读者将在第3章里看到，等号（=）可以说是Python语言中唯一的赋值操作符。但从版本2.0开始，等号可以和一个数学运算符在一起合用，表示对变量进行计算后还赋给该变量。这种赋值语句被称为增量赋值语句，下面这个语句：

```
n = n*10
```

现在可以被写为：

```
n *= 10
```

Python语言不支持n++或者--n这样的操作符。

示例最后的那条print语句中又出现了字符串格式操作符（%）。每个“%x”代码分别对应于将被打印的参数类型。我们已经在本章的前面看到过%s（对应于字符串）和%d（对应于整数）。现在又向大家介绍了%f（对应于浮点数值）。关于字符串格式操作符的详细介绍请阅读6.4节。

2.6 数字

Python语言支持四种数值类型，如下所示：

- int（带符号整数）
- long（长整数[也可以以八进制和十六进制表示]）
- float（浮点实数）
- complex（复数）

下面是一些例子：


```

int      0101      84    -237    0x80 017    -680  -0X92
long     29979062458L  -84140l  0xDECADEDEADBEEFBADFEEDDEAL
float    3.14159      4.2E-10      -90.    6.022e23      -1.609E-19
complex  6.23+1.5j    -1.23-875j    0+1j    9.80665-8.31441j -0.224+0j

```

Python语言中最令人感兴趣的数值类型是长整数和复数类型。请不要把Python语言中的长整数类型与C语言中的长整数类型混为一谈。Python语言中的长整数其长度远远超过任何C语言中的长整数；就其数值范围来说，对Python长整数的唯一限制就是你的（虚拟）内存的容量。如果读者熟悉Java，就会知道Python语言中的长整数类似于它的BigInteger类的类型。

许多语言都不支持复数（带-1的平方根的数字即虚数），除了Python，它在其他语言中可能都是以类（class）的面目出现的。

数值类型将在第5章中讨论。

2.7 字符串

Python语言中的字符串被定义为引号之间连续的字符集合，单引号或双引号都可以用在Python中。字符串子集可以用分离操作符（“[]”或者“[:]”）选取，字符串中的第一个字符的索引下标是0，依次递增，索引值-1指向最后一个字符。加号（+）是字符串合并操作符，星号（*）是重复操作符。下面是一些字符串及其使用方法的例子：

```

>>> pystr = 'Python'
>>> iscool = 'is cool!'
>>> pystr[0]
'p'
>>> pystr[2:5]
'tho'
>>> iscool[:2]
'is'
>>> iscool[3:]
'cool!'
>>> iscool[-1]
'!'
>>> pystr + iscool
'Pythonis cool!'
>>> pystr + ' ' + iscool
'Python is cool!'
>>> pystr * 2
'PythonPython'
>>> '-' * 20
'-----'

```

在第6章可以学到更多关于字符串方面的知识。

2.8 列表和表列

列表（list）和表列（tuple）可以看做是一个个的“桶”，里面盛着任意个数的Python对象。这些对象都是排好序的，可以通过各自的索引下标进行存取；与数组的情况差不多，只是列表和表列可以容纳不同类型的对象罢了。

列表和表列的主要区别是：列表用方括号（“[]”）定义，其中元素及其长度可以变化；表列用圆括号（“（）”）定义，定义后不能再有变化。我们可以把表列看做是“只读”的列表。列表的子集可以用分离操作符（“[]”或者“[:]”）选取，具体做法类似于对字符串的操作，如下所示：

```
>>> aList = [1, 2, 3, 4]
>>> aList
[1, 2, 3, 4]
>>> aList[0]
1
>>> aList[2:]
[3, 4]
>>> aList[:3]
[1, 2, 3]
>>> aList[1] = 5
>>> aList
[1, 5, 3, 4]
```

对表列的分离操作也大同小异，只是不能（像上面aList[1]=5那样）对其元素进行赋值。如下所示：

```
>>> aTuple = ('robots', 77, 93, 'try')
>>> aTuple
('robots', 77, 93, 'try')
>>> aTuple[0]
'robots'
>>> aTuple[2:]
(93, 'try')
>>> aTuple[:3]
('robots', 77, 93)
>>> aTuple[1] = 5
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

刚才例子中的最后一条语句出现了错误，因为我们试图对一个表列元素进行赋值，而这是不允许的。第6章里有对列表、表列以及字符串的详细论述。

2.9 字典

字典就是Python语言里的哈希表（hash table）数据类型。它们的作用类似于Perl语言中的关联数组（associative array）或哈希表，其内容是成对出现的关键字和值。关键字几乎可以是任何一种Python数据类型，但通常是数字或字符串；值则可以是任意的Python基本数据类型。字典用花括号（“{}”）定义，如下所示：

```
>>> aDict = {}
>>> aDict['host'] = 'earth'
>>> aDict['port'] = 80
>>> aDict
{'host': 'earth', 'port': 80}
```

```
>>> aDict.keys()
['host', 'port']
>>> aDict['host']
'earth'
```

我们将在第7章详细讨论字典。

2.10 代码段使用缩进

Python语言中的代码段是用缩进而不是用花括号之类的符号来标识的。没有了额外的符号，程序阅读起来就更容易了。此外，缩进能够明显地表示出一条语句是属于哪个代码段的。当然，代码段也可以只由一条语句构成。

初次接触Python的时候，人们往往会惊讶于代码段使用的缩进。人们的思维是有一定惯性的，在习惯用括号做代码段分隔符这么多年后，对只使用缩进办法来标识代码段的第一印象可能不会太好。但不要忘记Python有两个特性是简单易学和便于阅读理解。在Python里缩进第一行语句的三百六十五天之后，回头看看，问问自己是否还坚持当年的想法。我想，你肯定会发现没有括号的日子并不象原来想象的那么糟糕。

2.11 if语句

标准的if条件语句遵循如下所示的语法：

```
if expression :
    if_suite
```

如果表达式expression是非零值或真，执行其子句；否则执行紧随其后的第一条语句。“子句”（suite）是Python语言的一个概念，指的是由单个或多个语句组成的子代码段。

```
>>> if counter > 5:
...     print 'stopping after 5 iterations'
...     break
```

Python只允许在if语句里使用一个else语句，具体语法如下所示：

```
if expression:
    if_suite
else:
    else_suite
```

Python中还有一个名为elif的“else-if”语句，它的语法如下所示：

```
if expression1:
    if_suite
elif expression2:
    elif_suite
else:
    else_suite
```

另一个惊讶是：Python语言里没有switch或case语句。这一点在一上来也让人摸不着头脑并感到不方便，但Python语言简洁的语法并不会使一连串的if-elif-else语句看起来不舒服。如果你

确实想避免大段的if-elif-else语句，另一个选择是使用for循环（马上就讲到）来遍历各种可能会出现的情况。

if、elif、和else语句将在第8章的有关章节里详细讨论。

2.12 while循环

标准的while条件循环语句和if很相似。再次提醒大家：在每个代码子段里，都要注意用缩进（和伸出）来分隔各段代码以及确定语句所属的代码段：

```
while expression :
    while_suite
```

这个语句的子句会不停地循环执行，直到表达式变为零值或假；然后回到紧随其后的第一条语句上继续执行。

```
>>> counter = 0
>>> while counter < 5:
...     print 'loop #%d' % (counter)
...     counter = counter + 1

loop #0
loop #1
loop #2
loop #3
loop #4
```

while和for循环（下面将会看到）将在第8章中介绍程序循环的有关小节里讨论。

2.13 for循环和range()内建函数

Python中的for循环与其说像是一个传统的计数器形式的for条件循环，不如说更像是shell脚本程序设计语言中的foreach遍历循环。Python语言中的for循环要用到一个我们称之为序列(sequence)的数据类型（列表、表列或字符串等），并遍历序列中的每个元素。

```
>>> print 'I like to use the Internet for:'
I like to use the Internet for:
>>> for item in ['e-mail', 'net-surfing', 'homework', 'chat']:
...     print item
...
e-mail
net-surfing
homework
chat
```

如果我们让刚才例子中的输出都显示到一行而不是分开的行上可能会更好看些。在缺省的情况下，print语句会自动在每行的末尾加上一个换行符。在print语句最后加上一个逗号(,)就能取消这种情况，如下所示：

```
print 'I like to use the Internet for:'
for item in ['e-mail', 'net-surfing', 'homework', 'chat']:
    print item,
print
```

在上面的代码中我们还加上了一个不带参数的print语句，它的作用是在输出行的最末尾加上一个换行符；否则提示符将出现在最后一个输出数据所显示的那一行上。下面是最终代码的结果输出：

```
I like to use Internet the for :
e-mail net-surfing homework chat
```

print语句中用逗号分隔的元素输出时会自动在彼此之间加上一个空格做分隔符。对程序员来说，控制输出格式最好的办法是使用字符串格式，因为它给出的就是输出时的准确排列，不必再考虑因逗号而产生的空格。它还可以把全部数据都集中到一个地方去——即格式操作符右侧的表列或者字典里。如下所示：

```
>>> who = 'knights'
>>> what = 'Ni!'
>>> print 'We are the', who, 'who say', what, what, what, what
We are the knights who say Ni! Ni! Ni! Ni!
>>> print 'We are the %s who say %s' % \
...      (who, ((what + ' ') * 4))
We are the knights who say Ni! Ni! Ni! Ni!
```

使用字符串格式操作符还可以在实际输出之前完成一些快速字符串处理工作，如我们在刚才的例子中看到的。

作为循环介绍部分的结尾，我们来看看怎样才能让Python语言的for语句像一个传统的数值计数循环那样工作。因为我们不能改变for语句的行为（遍历一个序列），所以要对序列进行处理，让它是一个数字列表。这样，即使我们仍然是在遍历一个序列，最终出现的还是我们预期的递增计数的效果，如下所示：

```
>>> for eachNum in [0, 1, 2, 3, 4, 5]:
...     print eachNum
...
0
1
2
3
4
5
```

在上面的循环里，eachNum中保存着将要显示或者用来进行数值计算的数字。因为我们选取的数字范围会发生变化，所以Python语言为我们大家准备了一个用来生成这样一个数字列表的range()内建函数。它会根据我们的希望按一个数字范围生成与之对应的数字列表，如下所示：

```
>>> for eachNum in range(6):
...     print eachNum
...
0
1
2
3
4
5
```

2.14 文件和open()内建函数

对一门计算机语言来说，到语法学得差不多的时候，剩下来最重要的一个内容就是文件访问这方面了；如果程序和数据没法存起来，什么实际工作也没有完成的可能。

如何打开一个文件

```
handle = open( file_name, access_mode = 'r' )
```

`file_name`变量保存着要打开的文件的文件名字符串，对应于读操作的`access_mode`是“r”，写操作是“w”，添加操作是“a”。还可以用在`access_mode`字符串里的标志有对应于读写操作的“+”和二进制存取操作的“b”。如果没有具体给出访问方式，就用缺省的只读操作方式（“r”）打开文件。

编程提示：什么是属性？

[CN]

属性是与一个数据相关联的几个数据项。属性可以简单到只是一个值，也可以是函数和方法（请参考2.17节中的内容）这样的可执行对象。那么，什么样的对象会有属性呢？很多类、模块、文件、复数等等都有属性。而这也只是一部分带属性的Python对象而已。

怎样才能访问对象的属性呢？使用属性的点记号，即把对象及其属性名放在一起，中间用一个句点隔开，比如：`object.attribute`

如果`open()`函数操作成功，就会返回一个文件对象做为句柄（即变量`handle`）。这个文件的所有后续操作都必须通过它的文件句柄来实现。再返回文件对象之后，我们才能用`readlines()`和`close()`等方法执行其他的功能。方法是对象的属性，必须通过属性的点记号来使用（请参考上面的编程提示）。

下面这几行代码将提示用户输入一个文本文件的名称，然后打开该文件并把其内容显示在屏幕上：

```
filename = raw_input('Enter file name: ')
file = open(filename, 'r')
allLines = file.readlines()
file.close()
for eachLine in allLines:
    print eachLine,
```

上面的代码没有采用读一行显示一行、依次循环到文件末尾的办法。我们一次性读入整个文件的全部文本行，关闭文件，再遍历显示文件的内容。这样编写程序的好处是可以更加迅速地完成任务。显示输出和文件访问两项操作不必交替进行，读入一行文本和在屏幕上显示一行文本的操作是分两个步骤完成的。这样做更干净利索，同时把两个没有什么关系的工作区分开来。这里唯一需要注意的是文件的长度。上面的代码适用于一般长度的文件。太大的文件会占用过多的内存，如果真是这样，最好还是回到一次读入一行文本的办法上去。

另外一个需要注意的语句是我们在`print`语句的结尾处加上了逗号以取消换行符的显示。为什么呢？因为文件中的每一行文本已经在其结尾处包含有换行符了。如果我们不取消由`print`语句添加的换行符，显示在屏幕上的文件内容就会是隔两行显示一行的样子。

我们将在第9章里详细讨论文件对象、它们的内建方法属性、以及访问你本地文件系统的具体办法。请到那里做进一步的学习。

2.15 错误和例外

语法错误是在编译的时候被检测出来的，但Python也允许在程序执行时对错误进行检测。如果真的检测到一个错误，Python解释器就会引发（也叫做抛出、生成、触发）一个例外。正是因为Python语言可以在程序执行时对例外做出报告，程序员才能够迅速找出他们程序中的错误，并且在出现预料外错误的情况下调整软件执行某些个特定的动作。

如果想在自己的代码里加上错误检测或例外处理功能，只需用try-except语句把有关语句部分“打包”即可。跟在try语句后面的部分是准备进行错误检测的代码，跟在except语句后面的是预期的例外情况出现后将要执行的代码。

```
try:
    try_running_this_suite
except someError:
    suite_if_someError_occurs
```

程序员可以用raise命令明确地引发一个例外情况。读者可以在第10章学习到更多关于例外的知识，那里还有一个Python语言例外情况的汇总表。

2.16 函数

Python语言中的函数所遵从的规则和语法与其他大多数语言很相似。函数的调用要使用函数操作符（`()`），函数在可调用之前必须先经过定义，函数的类型就是它返回的值的类型。

函数的所有参数都是以变量引用的方式调用的，也就是说，参数在函数中的任何变化都会影响到函数外部与之对应的原始对象。

1. 如何定义函数

```
def function_name([arguments]):
    "optional documentation string"
    function_suite
```

对函数进行定义的语法以关键字def开始，后面跟着函数名和该函数可能具有的任何参数。函数的参数（比如上面的定义语句中的arguments）是可选的，这也是为什么把它放在方括号里的原因。（在你的实际代码里千万不要加上方括号！）该语句以一个冒号结束（这与if或while语句的结束方式是一样的），后面跟着函数主体部分的代码。下面是一个简单的例子：

```
def addMe2Me(x):
    'apply + operation to argument'
    return (x + x)
```

这个函数的名字的意思是“自己加上自己”，它以一个对象为参数，把该对象的当前值和自己本身相加，返回得到的和。对于数值性的参数其结果是显而易见的，但我们要指出的是加号对几乎所有的类型都是有效的。换句话说，大多数标准的类型都支持“+”操作符，不管它是数值加法还是序列合并。

2. 如何调用函数

```
>>> addMe2Me(4.25)
8.5
>>>
>>> addMe2Me(10)
20
>>>
>>> addMe2Me('Python')
'PythonPython'
>>>
>>> addMe2Me([-1, 'abc'])
[-1, 'abc', -1, 'abc']
```

Python语言中的函数调用与其他高级语言中的情况是相似的，都是先给出函数的名字，再跟上一个函数操作符——一对括号。任何可选的参数都放在括号中间。请注意“+”操作符是如何对非数值类型起作用的。

3. 缺省参数

函数的参数可以有缺省值。如果需要用到缺省参数，在函数的定义部分将采用赋值语句的形式；但在实际情况中，这只是缺省参数的语法形式而已，它表明如果没有给这个参数准备一个值，它就将把赋值语句中定义的值做为参数的缺省值，如下所示：

```
>>> def foo(debug=1):
...     'determine if in debug mode with default argument'
...     if debug:
...         print 'in debug mode'
...     print 'done'
...
>>> foo()
in debug mode
done
>>> foo(0)
done
```

在上面的例子里，debug参数有一个缺省值1。如果我们在函数foo()中没有给出一个参数，debug就自动取为表示“真”值的1。在第二次调用foo()的时候，我们明确地给出了一个参数0，因此缺省参数就不起作用了。

这样一个简介性的章节是没有办法描述函数的各种功能的。详细内容请阅读第11章。

2.17 类

一个类（class）就是一个静态数据成员或者函数声明（它们也叫做类的属性）的包容器。我们通过类提供的东西来创建“真实”的对象，即类的实例（class instance）。包含在类中的函数叫做方法（method）。类是一个面向对象的结构，在现阶段的Python语言学习中还用不到，我们之所以在这里提到它，是为那些具备一些对象方法学背景并想了解类在Python语言中如何实现的人们准备的。

1. 如何定义一个类

```
class class_name[(base_classes_if_any)]:
```



```

"optional documentation string"
static_member_declarations
method_declarations

```

类是使用关键字class定义的。如果定义的是一个子类，就要在括号里给出它是从哪个母类或者基类中被推导出来的。类定义语句的标题行就这么点东西，它后面接下去是一个可选的类文档字符串、静态成员的定义以及各种方法的定义。

```

class FooClass:
    'my very first class: FooClass'
    version = 0.1          # class (data) attribute

    def __init__(self, nm='John Doe'):
        'constructor'
        self.name = nm    # class instance (data) attribute
        print 'Created a class instance for', nm

    def showname(self):
        'display instance attribute and class name'
        print 'Your name is', self.name
        print 'My name is', self.__class__    # full class name

    def showver(self):
        'display class (static) attribute'
        print self.version    # references FooClass.version

    def addMe2Me(self, x):    # does not use 'self'
        'apply + operation to argument'
        return (x + x)

```

在上面的类定义中，我们定义了一个静态数据类型的变量version，它由这个类的所有实例和四个方法所共享，这四个方法是：__init__()、showname()、showver()和我们见过的addMe2Me()。方法showname()和showver()没有太多的工作，只是把与自身创建有关的数据输出出来而已。方法__init__()的名字比较特殊，前后都使用双下划线字符(____)的名字都有特殊的意义。

方法__init__()是一个缺省提供的函数，它将在创建该类的一个实例时被调用执行；与构造器很相似，并将在对象生成之后立刻被调用。它的目的是让实例完成一些必要的“启动操作”，为具体的程序运行做好准备工作。我们创建的__init__()方法将取代Python语言为我们准备的缺省方法（缺省方法什么事情也不做），这样我们就可以在自己的实例被创建时实现一些定制或者“额外”的处理。在上面的例子里，我们初始化了一个名为name的实例属性。这个变量只与类的实例相关联，实际并不是类本身的组成部分。__init__()还有一个缺省参数（我们在前一小节刚介绍过缺省参数）。另外读者肯定也注意到了出现在每一个方法中的参数self。

self是什么？简单说来self是一个实例引用自身所用的句柄（在C++或Java等其他面向对象的语言中，与self对应的是this）。调用一个方法的时候，self指的就是发出这个调用的那个实例。如果没有实例，类的任何方法都无法被调用，而这也正是必须有句柄self的原因之一。那些属于某个实例的类方法（class method）又被称为绑定方法（bound method）。那些不属于某个实例的类方法（class method）被称为非绑定方法（unbound method），它们是无法被调用执行的（除非明确地传递了一个实例做为其第一个参数）。

2. 如何创建类的实例

```
>>> foo1 = FooClass()
Created a class instance for John Doe
```

所显示的字符串是调用一个__init__()方法后的执行结果，我们并不需要明确地做出__init__()调用。不管是由我们自行提供还是使用解释器缺省的__init__()方法，只要是创建了一个实例，__init__()就会被自动执行。

创建一个实例看上去就象是调用了函数，二者的语法也是完全一样的。创建类的实例明显地使用与调用函数或者方法相同的函数操作符。但是千万不要把这二者弄混了。使用相同的符号并不意味着操作是相同的。函数的调用和类实例的创建是完全不同的两件事情。“+”操作符也有同样的情况：如果给出的是两个整数，它完成的就是整数相加运算；如果给出的是两个浮点数，它完成的就是实数相加运算；而如果给出的是两个字符串，其结果将是字符串的合并结果。这三种操作彼此是有区别的。

既然我们已经成功地创建了我们的第一个类实例，现在来看几个方法的调用情况：

```
>>> foo1.showname()
Your name is John Doe
My name is __main__.FooClass
>>>
>>> foo1.showver()
0.1
>>> print foo1.addMe2Me(5)
10
>>> print foo1.addMe2Me('xyz')
xyzxyz
```

各函数的调用执行结果和我们的预期是一致的。请注意类名(class name)这个数据。在showname()方法里，我们将输出显示self.__class__变量，而它代表的应该是创建它的那个类的名字。在上面的例子里，我们并没有在创建实例时传递来一个名字，因此这里使用的就是缺省参数“John Doe”。在下面的例子里，我们不使用它了，如下所示：

```
>>> foo2 = FooClass('Jane Smith')
Created a class instance for Jane Smith
>>> foo2.showname()
Your name is Jane Smith
__main__.FooClass
```

第13章里有大量关于Python语言中的类和实例的内容。

2.18 模块

模块是这样一种逻辑方法：它把彼此相关的Python代码物理地组织和划分到一个一个的文件中去。模块可以包含可执行代码、函数、类或者这些东西的各种组合。

如果你创建了一个Python源代码文件，与之对应的模块的名字与该文件的名字是一样的，只是少了后缀的扩展名“.py”。模块建立之后，就可以用import语句把该模块导入到另外一个模块中去。

1. 如何导入一个模块

```
import module_name
```

2. 如何调用一个模块函数或者访问一个模块变量

在导入了一个模块之后，就可以用大家熟悉的点属性记号访问它的属性（函数和变量），如下所示：

```
module.function()
module.variable
```

我们现在重新给出一个“Hello World！”示例，但这次我们使用的是sys模块中的输出函数，如下所示：

```
>>> import sys
>>> sys.stdout.write('Hello World!\n')
Hello World!
```

这段代码与我们最初使用了print语句的“Hello World！”程序其行为是完全一样的。唯一的区别是这里调用了write()标准输出方法，它与print语句的不同之处在于需要明确地给出换行符（“\n”），这是因为write()不会自动加上这个符号。

现在，我们来看看sys模块中的其他属性和string模块中的其他函数：

```
>>> import sys
>>> import string
>>> sys.platform
'win32'
>>> sys.version
'1.5.2 (#0, Apr 13 1999, 10:51:12) [MSC 32 bit (Intel)]'
>>>
>>> up2space = string.find(sys.version, ' ')
>>> ver = sys.version[:up2space]
>>> ver
1.5.2
>>>
>>> print 'I am running Python %s on %s' % (ver, sys.platform)
I am running Python 1.5.2 on win32
```

读者可能已经自己总结出来了：sys.platform和sys.version变量所包含的信息是读者正在使用的计算机平台的名称和正在使用的Python语言的版本。

string.find()函数在字符串中查找子字符串。在上面的例子里，我们只想获取版本号，它的位置在字符串开始处第一个空格字符的右边。我们用find()找到该空格的位置，然后在字符串中截取下该空格前面的全部字符。

获取版本号的另一个办法是把整个字符串分解成一个个的单词（彼此用空格隔开）。版本号将是第一个单词，它也就是我们想要的东西。string.split()函数返回的是一个字符串中所有“单词”组成的一个列表，如下所示：

```
>>> verchunks = string.split(sys.version)
>>> verchunks
['1.5.2', '(#0,', 'Apr', '13', '1999,',
'10:51:12)', '[MSC', '32', 'bit', '(Intel)]']
>>> print 'I am running Python %s on %s' % \
...      (verchunks[0], sys.platform)
I am running Python 1.5.2 on win32
```

我们的输出和前面的例子是完全一样的。这个例子中，同样的工作明显有不只一个完成办法。在Python语言里这样的情况是不多的，但两个例子毕竟也可以让读者挑选一番。

你可以在第12章找到更多关于模块以及导入操作的内容。

我们将在随后的各个章节更加详细地讲述前面涉及的每一个论题，但如果你的初衷是不求甚解地尽快开始使用Python语言进行工作，希望这部分内容能够满足你的愿望。

2.19 练习

2-1 变量、print语句、和字符串格式操作符。启动交互式解释器，对一些变量（字符串、数值等等）进行赋值，然后在解释器中敲入变量名把它们显示出来。再使用print语句做同样的事情。单独使用变量名与使用print语句给出的结果有何不同？熟悉字符串格式操作符（%）的使用方法。

2-2 程序输出。请看如下所示的Python脚本：

```
# ! /usr/bin/env python
1 + 2 * 4
```

- 你认为这个脚本程序是干什么用的？
- 你认为这个脚本程序的输出是什么？
- 把代码输入一个脚本程序，然后执行它。它的动作与你的预期一致吗？为什么或者为什么不一？
- 如果你是在交互式解释器中执行上述代码，执行情况会有什么差别？实际执行之，并记下其执行结果。
- 应该如何改进脚本方式的输出情况才能让它到达我们预期的效果？

2-3 数字和操作符。进入解释器。使用Python语言对两个数字（各种类型）进行加、减、乘、除运算。再用取除法余数操作符得到两个数字相除的余数。最后，使用乘方操作符用两个数字求幂。

2-4 使用raw_input()函数获取用户输入。

- 编写一个使用raw_input()内建函数的小脚本程序，用它从用户那里接收一个字符串，向用户显示他或她刚才敲入的内容。
- 再添上一小段代码，但这次要输入的是一个数值。把输入数据转换为一个数字（使用int()或者其他数值转换函数），然后向用户显示这个数字（请注意：如果你使用的Python语言是1.5以前的版本，可能要使用string.atof()函数来实现数值转换）。

2-5 循环和数字。使用while和for语句编写几个循环。

- 编写一个从0到10计数的while循环（一定要注意你的解决方案确实是从0计数到10，而不是从0到9或从1到10）。
- 用for循环和range()内建函数做与(a)相同的工作。

2-6 条件。检查一个数字是否是正数、负数或零。先对固定数进行比较，然后修改程序使之能够接受来自用户的数值输入。

2-7 循环和字符串。由用户输入一个字符串，再以每次一个字符的方式把它显示出来。先用while循环完成这一工作，再用for循环完成之。

2-8 循环和操作符。建立一个有五个成员的固定列表或表列，输出它们的和。然后修改程序使这些数字都由用户来输入。要求用两种办法解决上面的问题，一次用while语句一次用for语句。

2-9 循环和操作符。建立一个有五个成员的固定列表或表列，输出它们的平均值。这个练习最困难的地方是用除法得出平均值的操作。你会发现整数除法的得数被截取为整数，因此你必须用浮点数除法才能获得精确的结果，float()内建函数会帮你这个忙。

2-10 在循环和条件语句中使用用户输入。用raw_input()内建函数提示用户输入一个1到100之间的数字。如果输入的数字符合条件，在屏幕上显示该情况后退出程序。否则，显示一条出错信息，重新提示用户输入直到接收到正确的输入数字为止。

2-11 菜单驱动的文本应用程序。从前面的5个问题的程序中任意选出一个，然后修改程序使它成为菜单驱动的应用程序，向用户提供一组选择，比如：(1) 5个数字的和，(2) 5个数字的平均值，...，(x) 退出。然后根据用户做出的选择来执行。如果用户选择了“退出”项，就退出程序。类似于这样的程序的最大的好处是用户不必反复运行你的这个程序就能执行任意次数的计算（这对做为应用程序第一用户的程序测试工程师们也大有好处）。

2-12 dir()内建函数。

a) 启动Python解释器。在提示符处敲入“dir()”来执行dir()内建函数。你看到些什么？请把你看到的列表中的每一个元素显示在屏幕上。写下每一个实际输出和你预期的输出。

b) 你可能会问dir()到底是干什么用的？我们已经看到在“dir”后面加上一对括号将运行这个函数。现在试试只在提示符处输入“dir”这几个字符。解释器给你什么结果？你认为它是什么意思？

c) type()内建函数以任意一个Python对象为参数，返回的是它的类型。输入“type(dir)”，让解释器对dir执行这个函数。你看到些什么？

d) 做为这个练习的最后一个部分，我们来看看Python的文档字符串。如果想访问dir()函数的文档，我们可以在它的名字后面加上“__doc__”。也就是说，在解释器的提示符处输入“print dir.__doc__”语句就可以看到dir()的文档字符串。许多内建函数、方法、模块和模块属性都有一个与之关联的文档字符串。我们建议你在编写代码的时候也加上自己的文档字符串，这对其他用户有帮助。

2-13 用dir()函数找出更多的sys模块。

a) 启动Python解释器。像刚才练习中那样执行dir()命令。现在，在提示符处输入“import sys”语句导入sys模块。再次执行dir()命令以确定出现有sys模块。现在输入“dir(sys)”在sys模块上执行dir()命令，你将看到sys模块的全部属性。

b) 显示sys模块的版本变量和计算机平台变量。记得要在变量名的前面加上“sys”字样以表明它们是sys的属性。版本变量包含的信息是你正在使用的Python解释器的版本；而计算机平台属性包含的是Python认为你正在使用的计算机系统的名字。

c) 最后，调用`sys.exit()`函数。如果在前面问题1中因为键盘输入有误而不能退出Python，这可以做为另外一种退出Python解释器的办法。

2-14 操作符优先级和括号的使用。重新编写2.4节里`print`语句中的算术表达式，但要用括号把操作数正确地分好组。

2-15 基本的排序。

a) 由用户输入三个都是数值的值，并把它们保存到三个不同的变量中去。不使用列表或排序算法，把这三个数字按由小到大的顺序人工排好序。

b) 如何修改a)中的程序才能使这些数字按由大到小的顺序排好序？

2-16 文件。输入并运行2.14节中的文件内容显示程序。确保它能在你的系统上顺利运行，再多试几个其他的文件。

第3章 语法和程序设计风格

我们的下一步目标是学习Python语言的语法，给出一些基本风格的准则，然后对标识符、变量和关键字等做一个简单的介绍。我们还将讨论需要为变量分配多少内存空间以及如何释放它们。最后，我们将面对一个比较大的Python程序示例——不要犹豫，冲吧。不必担心，保驾护航的人多着呐。

3.1 语句和语法

下面是一些与Python语言中的语句有关的规则和惯用符号：

- 井字号（#）后面是Python程序中的注释。
- 换行符（\n）是标准的文本行分隔符（每条语句占一行）。
- 反斜线字符（\）表示续行。
- 分号（;）用来隔开同一行上的两条语句。
- 冒号（:）用来分隔标题行及其后续子句。
- 语句（代码段）按子句的方式进行组织。
- 子句是通过语句的缩进来区分彼此的。
- Python文件被组织为“模块”。

1. 注释（#）

首先要说明的是：虽然Python是一个很容易阅读理解的语言，但这并不意味着程序员能够在自己的代码中适当和有效地使用和布置好注释与说明。Python程序中的注释语句是以井字号（#）打头的，这与其他UNIX操作系统中的脚本语言一样。注释可以从文本行的任意位置开始，解释器将忽略从井字号开始到文本行结束这一部分的所有字符。给程序添加注释要灵活而审慎。

2. 续行（\）

Python语句通常都是用换行符分隔的，也就是说每行只能有一条语句。但我们可以通过反斜线字符（\）把单个语句分成多行。放在换行符前面的反斜线字符（\）表示当前语句将延续到下一行去，如下所示：

```
# check conditions
if (weather_is_hot == 1) and \
    (shark_warnings == 0) :

    .send_goto_beach_mesg_to_pager()
```

不使用反斜线字符就能够实现语句续行的情况有两种：一是包容器对象中的元素在多行之间被断开；二是换行符包含在被扩在三个单引号中间的字符串之中。如下所示：

```
# display a string with triple quotes
print '''hi there, this is a long message for you
that goes over multiple lines... you will find
```

```
out soon that triple quotes in Python allows
this kind of fun! it is like a day on the beach!'''
```

```
# set some variables
go_surf, get_a_tan_while, boat_size, toll_money = ( 1,
    'windsurfing', 40.0, -2.00 )
```

3. 把多个语句组织为一个子句 (:))

在Python语言中，几个独立的语句组成的代码段叫做“子句”（suite，我们在第2章里做过简要的介绍）。if、while、def和class等复合语句都要有一个标题行和一个子句。整个语句（带有关键字）以标题行开始，以冒号(:)结束，后面跟着组成该子句的一行或者多行语句。我们将把标题行和子句部分合称为“从句”。

4. 用行缩进区分子句

我们已经在2.10节中介绍过，Python语言使用行缩进的办法来分隔代码段。内层的代码用空格或者TAB制表符来缩进。这里所说的缩进是严格精确意义上的缩进，换句话说，一个字句中的全体语句行都必须具备同样的缩进效果（比如同等数目的空格符）。同一个子句中的语句不允许出现缩进的起始位置不同的情况；否则，它们就会被看做是另外一个子句的组成部分，很可能引起语法错误。

缩进量的增加表示遇到了一个新的代码段，而它的结束信号是“扩出”，或者说是缩进量减少到上一个语句层水平。没有被缩进的代码——比如代码的最外层将被认为是脚本程序的主程序部分。

选择使用缩进方式来编写Python程序的原因是：人们认为这样编写出来的代码更加精致，更容易阅读。它还能帮助避免出现“悬垂的else”这类的问题，帮助避免出现某单个语句不知该归给哪一个从句的情况（比如说，在C语言中，有的if语句根本没有使用括号，可它后面却跟着两个缩进的语句。这样，第2条语句不理睬条件判断的结果，总是会执行，让程序员忙得晕头转向）。

最后，使用缩进之后将不再有“括号之战”。在C（也包括C++和Java）语言中，起始括号（左括号）可以放在标题行的同一行上，也可以放在下一行上，还可以缩进在下一行上。有的程序员喜欢这样放，有的程序员喜欢那样放，其结果可想而知。

使用缩进还能起到一定的性能改善作用，因为少用一个括号就意味着在执行时少占用一个字节。虽然括号本身毫不起眼，但在一天24小时、一周7天、一年365天不停运转着的因特网这样的全球性网络环境中会积少成多达到成千上万个字节，我相信读者能够体会到这一点。与缩进有关的诀窍和程序设计风格问题请阅读后面的3.4节。

5. 在一个文本行上编写多个语句 (;)

分号(;)允许在一个文本行上输入多个语句，但前提是这些语句都不会开始一个新的代码段。下面是一个使用分号的简单示例：

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

但我们要告诫读者不要过多地使用在一个文本行上输入多个语句的做法，因为这样会降低代码的可阅读性。你完全可以采用下面的方式：


```
import sys
x = 'foo'
sys.stdout.write(x + '\n')
```

在我们的例子里，把代码分几行输入能够显著地提高可阅读性。

6. 模块

每个Python脚本都被看做是一个模块。模块的物理存储方式就是磁盘上的文件。当一个模块大到一定程度或者功能多到一定程度时，把其中的部分代码移到另一个模块中去是很明智的做法。模块中保存的代码可能属于某个应用程序（比如一个可直接执行的脚本程序），也可能是一个从其他模块导入的程序库类型的模块中的可执行代码。我们在上一章里已经提到过，模块可以包含可执行代码段、类定义、函数定义或者这些东西的任意组合。

3.2 变量分配

本小节集中讨论变量分配方面的问题。我们将在3.3节里讨论什么样的标识符才是合法的变量名。

1. 等号(=)是赋值操作符

等号(=)是Python语言中的主赋值操作符。如下所示：

```
anInt = -12
aString = 'cart'
aFloat = -3.1415 * (5.0 ** 2)
anotherString = 'shop' + 'ping'
aList = [ 3.14e10, '2nd elmt of a list', 8.82-4.371j ]
```

请注意：赋值操作并不会明确地把一个值分配给变量，尽管根据其他程序设计语言的经验来看应该如此。在Python语言里，对象是通过引用方式来使用的，赋值操作也不例外；一个对象，不管它是新创建的还是已经存在的，赋值操作实际分配的是该对象的引用线索（reference）而不是一个值。如果现在还不能百分之百地理解这句话的含义，没关系，本章后面的内容还会专门讲述这个问题，现在只要知道有这回事就可以了。

此外，如果你熟悉C语言，就应该知道它是把赋值操作视为表达式的。但Python语言不采用这样的做法，赋值操作没有继承值。下面这些语句在Python语言里是不合法的：

```
>>> x = 1
>>> y = (x = x + 1) # assignments not expressions!
File "<stdin>", line 1
    y = (x = x + 1)
        ^
[2.0]
```

从Python 2.0开始，等号可以和一个算术运算符一起使用，计算结果还赋值给原来的变量。这叫做“增量赋值”，下面这条语句：

```
x = x + 1
```

可以写为：

```
x += 1
```

Python语言不支持象x++或--x这样的递增/递减操作符。

2. 把一个对象赋值给多个变量

```
>>> x = y = z = 1
>>> x
1
>>> y
1
>>> z
1
```

在上面的例子里，先创建了一个整数对象（它的值是1），再把该对象的引用线索分别都赋值给x、y和z。这是把一个对象分配给多个变量的做法。在Python语言里，我们还可以把多个对象赋值给多个变量。

3. 把多个对象赋值给多个变量

对多个变量进行赋值时可以使用我们称之为“表列方式”的赋值语句。这并不是一个Python语言中的正式叫法，但我们在这里使用“表列方式”这个词的原因是：用这种办法对变量进行赋值操作时，等号两边的对象都是表列（表列是我们在2.8节介绍过的一种标准的Python数据类型），如下所示：

```
>>> x, y, z = 1, 2, 'a string'
>>> x
1
>>> y
2
>>> z
'a string'
```

再上面的例子里，两个整数对象（它们的值是1和2）和一个字符串对象被分别分配给x、y和z。在用到表列的时候，我们通常都会用括号把它们凸显出来，虽然括号是可选性质的，但我们建议一定要使用它们以增加代码的可阅读性。

```
>>> (x, y, z) = (1, 2, 'a string')
```

如果读者在使用其他程序设计语言（比如C）的时候曾经遇到过需要互换变量值的情况，就肯定记得在交换一个值的同时必须要用一个临时变量（比如tmp）保存另一个值，如下所示：

```
/* swapping variables in C */
tmp = x;
x = y;
y = tmp;
```

上面的C语言代码段用来交换变量x和y的值。我们必须先把一个变量的值保存在tmp变量里，再把另一个变量的值赋值到第一个变量，最后再把临时变量中的值赋值给第二个变量。

Python语言中“表列方式”的赋值语句有一个有趣的副作用：就是在互换两个变量的值时不再需要一个临时变量了，如下所示：

```
# swapping variables in Python
>>> (x, y) = (1, 2)
>>> x
1
>>> y
2
```

```
>>> (x, y) = (y, x)
>>> x
2
>>> y
1
```

很明显，Python在实际完成赋值操作之前先进行了取值操作。

3.3 标识符

标识符就是一种计算机语言中允许做为名字的合法字符串。在标识符里，还有一些是关键字 (keyword)，即那些构成该语言结构的名字。做为关键字的标识符都是不能再用做其他用途的保留字，否则就会出现语法错误 (SyntaxError例外)。

Python语言还有一些用做“内建字”的标识符，虽然它们不是保留字，但我们不推荐在程序中使用这些特殊的名字。

1. 合法的Python标识符

Python标识符字符串的规则与其他大多数高级程序设计语言没什么大的区别：

- 第一个字符必须是字母或者下划线字符 (_)。
- 后续字符可以是字母、数字或下划线字符。
- 大小写敏感。

标识符不能以数字打头，除了下划线字符 (_)，其他的符号都不允许使用。因此，我们可以简单地把下划线字符也当做是一个字母字符。大小写敏感的意思是标识符foo不同于Foo，而这两者又都不同于FOO。

2. 关键字

Python语言目前有二十八个关键字，它们列在表3-1里。

一般说来，一种语言中的关键字必须相对保持稳定，但考虑到未来的发展变化（因为Python是一个正在成长和进化中的语言），在它的keyword模块里除有一个关键字清单外还准备了一个iskeyword()函数。

表3-1 Python语言中的关键字

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

出于兼容方面的原因，`assert`关键字是从Python 1.5版本开始出现的，而`access`关键字从1.4版本开始就不再使用了。 [1.5]

3. 内建字

除关键字以外，Python语言中还有一整套“内建字”，这些内建字是由解释器设置或者使用的，可以被解释器用在各级代码中。虽然不是关键字，但内建字也应被看做是“为系统保留的”，不要用于其他用途。但在某些情况下，这些内建字可能会被赋予（或者说重新定义为、替换为）其他含义。Python语言不支持标识符的重载，因此在任一给定时刻都只有一个名字“绑定”存在着。

4. 特殊的下划线标识符

在Python语言中还有一些更特殊的变量，其变量名里带有下划线前缀或下划线后缀。我们稍后就会知道其中有一些对程序设计人员非常有用，而剩下的作用不明或没什么用处。下面是对Python语言中下划线特殊用法的一个总结：

`__xxx` 不使用“`from module import *`”导入。

`__xxx` 系统定义的名字。

`__xxx` 类(class)中的局部变量名。

编程风格：不用下划线做标识符的开始字符

[CS]

因为Python系统/它的解释器/以及它的内建标识符都使用了下划线，我们建议程序设计人员最好不要用下划线做为标识符的起始字符。

3.4 程序设计风格准则

1. 注释

我们不必反复提醒注释对你本人和你的后继者是多么的有用。对一些很长时间（从软件开发的角度看几个月就足够长了）没有人碰过的代码来说，注释就更重要了。程序里不能没有注释，但也不能把注释写成小说。尽可能让注释短小精悍，并使它们出现在最需要的地方。最后，它将节省每一个人的时间和精力。

2. 文档

Python语言为了提供这样一种机制：文档字符串可以通过特殊变量`__doc__`来动态检索。模块、类定义或函数定义中的第一个未赋值字符串可以通过`obj.__doc__`来访问，其中的`obj`是模块、类或函数的名字。

3. 缩进

因为缩进是Python语言的一个重要角色，所以读者必须挑出一种最有利于阅读理解和最不容易引起误解的空格缩进方式。到底每次缩进几个空格或列呢，常识告诉我们：

1到2个空格可能缩进得不够；难于区分语句都属于哪些个代码段

8到10个空格可能缩进得太多；层次较多的代码会卷到下一行，使源代码阅读困难

四个空格挺好，这也是Python语言创始人最喜欢的选择。五个或六个空格也不坏，但文本编辑器一般不用这样的设置，所以它们也不很常用。三个或七个空格的情况说不上好坏。

如果你打算使用TAB制表符实现缩进，那就要注意这个问题：在不同的文本编辑器中TAB的概念是不尽相同的。如果你的代码会在不同的系统上存放和运行，或者会被不同的文本编辑器读写，我们建议你最好不要使用TAB制表符来缩进。

4. 选择标识符的名字

选择标识符名字同样需要有良好的判断力。给变量挑选标识符时尽量让它又短又有意义。虽然在如今的程序设计语言里变量名的长度不再是一个问题，但把它限制在一个合理的长度内还是应该的。给模块（即Python语言中的文件）起名字也要注意这些方面。

3.4.1 模块的结构和布局

模块简单说来就是把Python代码按一定条理组织好以后的物理保存方式。每个文件都应该有一个统一完整和便于阅读的结构。下面就是一个这样的布局形式：

#(1) 启动语句行（UNIX操作系统用）

#(2) 模块文档

#(3) 模块导入

#(4) 变量定义

#(5) 类定义

#(6) 函数定义

#(7) 程序“主体”，主程序

图3-1给出了一个典型化模块的内部结构：

(1) 启动语句行

一般只用在UNIX环境中，这个启动语句行允许只使用脚本程序的文件名就能够执行之（不要求调用解释器）。

(2) 模块文档

简要说明模块的功能和主要的全局性变量，可以从外部通过`module.__doc__`访问。

(3) 模块导入

导入当前模块中的代码所要求的全部模块；模块只（在该模块被加载时）导入一次；如果导入操作发生在函数内部，则导入发生在该函数被调用的时候。

(4) 变量定义

在这里定义的（全局性）变量是本模块中多个函数将要用到的（如果不是这样，请把它们定义为局部变量以改进内存的使用情况和执行性能）。

(5) 类定义

此处对用到的各种类进行定义，包括全体静态成员和方法属性；类是在这个模块被导入并且这个类的语句被执行时定义的。其文档变量是`calss.__doc__`。

(6) 函数定义

在此处定义的函数可以从函数外部通过`module.function()`的方式来访问；函数是在这个模块被导入并且`def`语句被执行时定义的。其文档变量是`function.__doc__`。

(7) “主”程序

不管这个模块是被导入的还是做为脚本程序来运行，这部分的代码都会执行；这部分所包含的功能性代码一般不是很多，更常见的是根据执行模式决定执行路线。

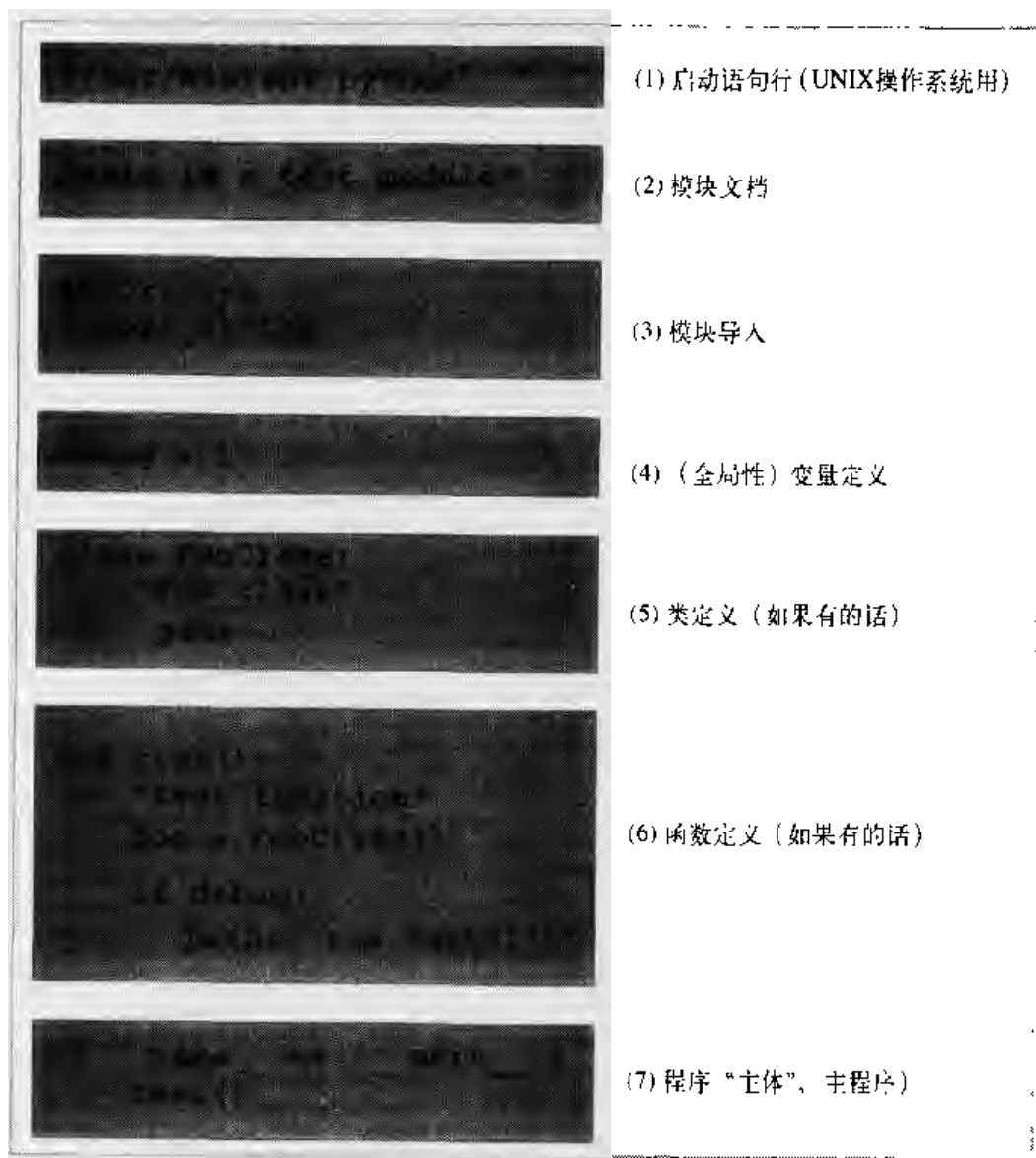


图3-1 典型模块的内部结构

编程风格：“主”程序调用main()

[CS]

代码的主体部分所包括的语句行一般都是读者在前面例子里看到的那样：先检查变量 `__name__`，再采取相应的动作（请参考下面的编程提示）。主体部分中的代码通常是完成类、函数和变量的定义，然后检查 `__name__` 变量看看是否需要调用完成该模块的其他函数（这个函数通常被称为 `main()`）。主体部分干的事情一般也就是这么多。（我们在刚才给出的例子里用的是 `test()` 而不是 `main()`，这是为了避免你在读到此编程风格之前被弄糊涂。）

撇开名字不说，我们想强调的是：代码中的这个位置是安排一个测试子局的绝佳场所。正像我们已经在3.4.2节里讲述过的，大多数Python模块都只是为了导入目的而创建的，直接调用一个这样的模块将引起对此模块中的可执行代码进行测试。

大多数项目都是由一个主应用程序和一些必要的导入模块组成的。因此我们一定要牢记：大多数模块是为了被导入而不是做为直接执行的脚本程序而创建的。我们创建的模块通常是一些Python程序库类型的模块，它们的唯一用途就是被导入到其他模块中去。总之，只有一个模块将被执行，就是那个容纳着主应用程序的模块，其执行方式既可以是用户从命令行上来执行之，也可以是批处理或UNIX操作系统中cron作业那样的定时机制；可以经由Web服务器调用，也可以从一个GUI操作界面中调用。

了解了这些事实之后，我们还必须记住所有的模块都有执行代码的能力。不管是否故意设置，代码最高层部分中的Python语句（即那些没有缩进的语句行）都将在所属模块被导入到其他模块时被执行。因为有一个“功能”，所以我们在编程时就应该采取一个“更安全”的办法，即把那些想在模块被导入时执行的代码以外一切东西都编写到一个函数中去。同时，通常只有主应用程序模块才有大段的可执行代码做为程序本身的最外层；其他被导入的所有模块外露的东西很少，具体内容都在函数或类里面。

编程提示：用__name__指示模块是如何加载的

[CN]

因为不管一个模块是被导入还是直接运行，“主”代码部分都会执行，所以我们通常要知道这个模块到底是怎样加载的，根据具体情况来选择执行路线。一个应用程序需要导入另外一个应用程序的模块，其目的可能是想要访问有用的代码，如果不采用导入的办法，就只能采取复制的手段了（不是面向对象方面的东西）。但在这种情况下，你想做的只是访问其他应用程序的代码而已，并不是要运行它。所以问题就来了：“Python能否在运行时检测出一个模块是被导入的还是直接执行的？”答案就是...，对了！系统变量__name__就是解决问题的关键。

- 如果是导入的，__name__变量里包含的就是该模块的名字
- 如果是直接执行的，__name__变量里包含的就是 '__main__'

3.4.2 在主体部分里加上测试方面的内容

对好的程序员和工程师来说，为整个应用程序提供一个测试手段或者保险措施是义不容辞的。Python语言很好地简化了这个问题，因为模块主要是为导入而创建的。对这些模块来说，你已经肯定地知道它们不会直接执行。那么，当它们被调用时对模块进行一番测试不是很好吗？这样设置起来难吗？一点儿也不。

一般情况下，测试软件只有在文件被直接执行时才运行，不会在从另外一个模块里被导入时运行。我们在上面的编程提示里介绍了如何判定一个模块是被导入还是被直接执行。我们可以通过__name__变量利用这种特性。如果这个模块是做为一个脚本程序被调用的，就可以在那里插入测试代码，也许是做为main()或test()函数（或者你给自己的“二级”代码所取的随便什么名字）的一部分，这些函数只有在模块直接执行时才会被调用。

代码的“测试器”部分需要随时与新的测试条件和测试结果保持一致，并且只要代码进行了修改升级，就必须进行测试。这些步骤对增强代码的健壮性很有帮助，更不用说验证和校验

新的功能或升级了。

3.5 内存管理

到目前为止我们已经看过很多Python代码示例了，可能大家已经注意到变量和内存管理方面一些有意思的细节。下面是一些比较明显的特点：

- 变量不必提前定义。
- 变量类型不必定义。
- 程序员不必关心内存管理。
- 变量名可以“回收使用”。
- del语句允许明示性地收回使用中的变量。

3.5.1 变量定义

在大多数编译型语言里，变量在使用之前必须先经过定义。事实上，C语言在这一点上更为挑剔：变量必须在代码的最开始还没有任何语句的时候进行定义。其他语言，比如C++和Java，允许随时进行变量定义——也就是说，在代码中间进行定义，但还是要先在变量使用前对它们的类型和名字进行定义。可Python语言就没有明确的变量定义。变量是在第一次赋值操作时被“定义”的。但变量在被（创建和）赋值之前是无法使用的，这一点倒是和其他语言一样，如下所示：

```
>>> a
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: a
```

变量在赋值之后，既可以用它的名字访问它了，如下所示：

```
>>> x = 4
>>> y = 'this is a string'
>>> x
4
>>> y
'this is a string'
```

3.5.2 动态确定变量的类型

除了缺少变量定义之外，我们的另外一个发现就是还缺少对变量类型的定义。在Python语言里，一个对象的类型和它所使用的内存空间是在运行时确定和分配的。虽然代码是字节编译的，Python仍算得上是一种解释型语言。对象在创建时（也就是在赋值时）先由解释器创建一个对象，它的类型是根据赋值语句右边的操作数所使用的语法来确定的。对象创建好以后，它的一个引用线索被赋值给赋值语句左边的变量。

3.5.3 内存分配

做为一个有责任心的程序员，我们知道在为变量分配内存空间的时候实际是借用了系统的

资源，并且我们最终还是要借用的东西还给系统。让人高兴的是，我们不仅不必明确地进行内存的分配工作，还不必负责回收它。真是内存管理简单化了。因此，我们可以得出这样一个结论：Python语言可以被形象地说成是一种应用程序编写人员的工具，使用它就不必再考虑底层的、面向操作系统或者面向机器的工作。

3.5.4 废弃物回收

不再使用的内存将通过一种被称为“废弃物回收”的机制收归公有。Python的废弃物回收器会在一个数据对象不再被需要时自动回收原来分配给它的内存空间，这一切都不需要程序员插手。那么，Python又是如何判定一个对象“不再被需要”了呢？答案是通过记录对象的引用线索个数。这叫做“引用线索计数”。

3.5.5 引用计数

为了跟踪记录已经分配的内存，Python的做法类似于扑克牌游戏中的记牌手法。一个对象在创建时就会被加上一个引用。一个内部的引用记录变量将跟踪记录下每个对象身上有多少个引用。当一个对象被创建和（它的引用）被赋值时，它的初始引用记数是1。

对象新的引用又叫做别名（alias），它一般发生在以下几种情况下：其他变量也被赋值到同一个对象；对象做为调用其他代码部分（比如函数、方法或者类的实例等）的参数被传递时；对象做为一个序列或映射图的成员被赋值时。如下所示：

```
# initialize string object, set reference count to 1
foo1 = 'foobar'

# increment reference count by assigning another variable
foo2 = foo1 # create an alias

# increment ref count again temporarily by calling function
check_val(foo1)
```

在上面的函数调用中，引用计数在创建对象时被设置为1，新创建一个别名后增加一个数，用做函数调用参数时又增加一个数。函数调用结束时引用会减少一个数，如果把foo2从变量名空间里删除又会减少一个数。如果再离开foo1的作用范围，这个对象的引用就会减少为0，它也就会被系统回收去了。（关于变量的作用范围请阅读11.8节。）

3.5.6 del语句

del语句会把一个对象的引用减少一个数，它的语法是：

```
del obj1 [, obj2 [, . . . objN . . .]]
```

举例来说，在上面的例子里执行“del foo2”将会有两个结果：

- 1) 从变量名空间里删除名字foo2。
- 2) 减少对象‘foobar’的引用（减去1）。

再进一步，执行“del foo1”将清除对象‘foobar’的最后一个引用，把它的引用计数减为0，

这将使该对象成为“不可访问的”和“不可到达的”。也就是在此时，该变量成为一个废弃物回收的候选。需要注意的是跟踪或调试程序可能会给一个对象额外增加一些个引用，这样就会延迟回收该对象的时间。

3.5.7 减少引用计数

读者可能已经注意到每当del语句执行的时候，并不是真的删除一个对象，它只是减少该对象的引用计数。因此，就有可能出现这样的情况：如果把一个对象的引用重新赋值为另外一个对象，就可能“丢失”原来的对象。如下所示：

```
foo1 = 'foobar' # create original string
foo1 = 'a new string' # 'foobar' 'lost' and reclaimed
```

这个例子告诉我们一个对象的全体引用在对原变量重新赋值时会发生怎样的情况。最常见的情况并不是重新赋值或者调用del语句，而是退出对象的当前作用范围。

退出当前“作用范围”（scope）的意思是当一个函数或方法结束执行的时候，在其作用范围内的所有对象都将被清除（除了那些做为返回值传递回来的对象），比如刚才那个以foo1做为check_val()函数的参数的这种情况。foo1的引用计数在这个函数被调用时增加一个数，而当函数执行完成后减少一个数。

下面是一个会引起引用计数减少的操作情况汇总。当一个与每个对象对应的变量发生以下情况时，该对象的引用计数会减少一个数：

- 明确地出现在del语句里。
- 被赋值（重新赋值）为另外一个对象。
- 超出作用范围。

3.6 第一个Python应用程序

到目前为止，我们已经对Python语言的语法、程序设计风格、变量赋值和内存分配等方面做了介绍，现在来看一个比较复杂的Python程序示例。这个程序里还有许多我们到目前为止没有介绍过的结构，但我们相信：因为Python语言很简单，很容易阅读理解，所以读者在读代码的时候是能够做出正确判断的。

我们将要阅读的源代码文件是fgrepwc.py，这个文件名由两个UNIX工具程序的名字（fgrep和wc）组成。fgrep是一个简单的字符串搜索命令，它会在一个文本文件里逐行检索，并输出包含有被检索字符串的任何文本行。要注意的是一个字符串可能会不止一次地出现在同一文本行上。wc是另一个UNIX命令；它对做为输入的文件中的字符数、单词数和行数进行统计。

我们的程序把这两项工作结合起来。它要用到一个搜索字符串和一个文件名、输出所有出现了匹配的文本行，并且给出匹配文本行的总数。因为一个字符串在一个文本行上可能会出现不止一次，所以我们要求必须让匹配文本行总数正确反映出匹配文本行的个数，不能是字符串在文本文件中出现过的次数。（本章结尾处的一道练习题是要求读者把这个程序升级为统计输出字符串总的匹配次数。）

还要注意的一点是：这本书里面的源代码一般不带注释，带注释的版本都放在书后所附的

CD-ROM光盘上了。但在这个例子里我们将给出注释，帮助读者探索包含前面各章内容的第一个比较长的Python程序。

下面就是fgrepwc.py程序的源代码——程序列表3-1，随后是有关的分析。

程序列表3-1 文件查找(fgrepwc.py)

这个应用程序在一个文件里查找一个搜索字符串，把每个匹配文本行和找到的匹配文本行总数显示出来。

```

1  #!/usr/bin/env python
2
3  "fgrepwc.py -- searches for string in text file"
4
5  import sys
6  import string
7
8  # print usage and exit
9  def usage():
10     print "usage: fgrepwc [ -i ] string file"
11     sys.exit(1)
12
13 # does all the work
14 def filefind(word, filename):
15
16     # reset word count
17     count = 0
18
19     # can we open file? if so, return file handle
20     try:
21         fh = open(filename, 'r')
22
23     # if not, exit
24     except:
25         print filename, ":", sys.exc_info()[1]
26         usage()
27
28     # read all file lines into list and close
29     allLines = fh.readlines()
30     fh.close()
31
32     # iterate over all lines of file
33     for eachLine in allLines:
34
35         # search each line for the word
36         if string.find(eachLine, word) > -1:
37             count = count + 1
38             print eachLine,
39
40     # when complete, display line count
41     print count
42
43 # validates arguments and calls filefind()
44 def checkargs():
45
46     # check args; 'argv' comes from 'sys' module
47     argc = len(sys.argv)
48     if argc != 3:
49         usage()
50
51     # call fgrepwc.filefind() with args
52     filefind(sys.argv[1], sys.argv[2])
53
54 # execute as application
55 if __name__ == '__main__':
56     checkargs()

```

1~3行

UNIX操作系统的启动语句行和模块的文档字符串。如果读者从另外一个模块导入fgrepwc模块,就可以通过fgrepwc.__doc__来访问这个字符串。这是一个很关键的功能,因为它能够在—个动态的执行环境中提供原来是静态的文本信息。我们还要指出的是这里讲述的通常也就是文档字符串的唯一用途了。它不会用于其他目的,但能够方便地在一个文件的开始给出该程序的某些注释。(我们建议读者去看看标准函数库中cgi模块文件开始部分的文档字符串,这是很重要的模块文档例子。)

5~6行

我们已经见过sys和string模块了。sys模块包含着Python解释器和操作系统之间交互作用的大部分变量和函数。读者可以在这里找到命令行参数、exit()函数、Python路径环境变量PYTHONPATH的内容、标准文件以及与错误有关的信息等。

string模块包含的是对字符串进行处理时所必须的各种函数,比如字符串-整数转换函数atoi() (及其相关函数)、各种字符串变量和其他对字符串进行处理的函数。

为程序准备导入模块的主要动机是让程序更小巧,更快和更高效,程序员可以专心于完成具体工作的软件的编写,对那些需要用到的模块可以采取即导入即用的办法。在Perl和Java语言里也有与此相似的设置,导入模块、软件包等特色;C和C++也通过头文件在一定程度上体现了这一思想。

8~11行

我们在这里定义了一个名为usage()的函数,它没有参数。这个函数的作用很简单,向用户显示一条信息,给出调用这个脚本程序的正确的命令行语法,然后用sys模块中的exit()函数退出。还要说明的是在Python的变量名空间里,从导入模块里调用一个函数需要使用一个“完全授权”名字。一切被导入的变量和函数都必须使用module.variable或者module.function()这样的格式。因此这里使用的就是sys.exit()。

如果想从一个模块里导入特定的函数或者变量,还可以使用from-import语句,把需要的东西导入到当前的变量名空间里来。如果使用了这种导入办法,调用函数时就可以只使用它的属性名了。

举例来说,如果我们只想从sys模块里导入exit()函数,其他东西都不要,就可以使用下面的语句替换程序中的语句:

```
from sys import exit
```

然后在usage()函数中,调用exit()函数时就不必再加上“sys.”了。关于exit()函数还有一点需要说明:sys.exit()函数的参数和C语言中exit()函数的情况是一样的,它是将返回到调用函数去的一个值,而调用函数通常就是一个shell中的命令行程序。根据这一点,我们必须指出的是:只有对命令行驱动的应用程序才能将我们例子里的“协议”用来显示和退出。

如果是在一个基于Web的应用程序里,这就不是一个退出运行中的程序的好办法了,因为发出调用指令的Web浏览器预期接收的是一个合法的HTML响应。对Web应用程序来说,返回一个HTML格式的错误信息更适当一些,这样终端用户才能改正他们的输入。因此,Web应用程序基

本上是不能用一个错误信息来终止运行的。退出一个程序会向用户发送一个系统或浏览器错误消息，这是不正确的行为，网站应用程序的开发人员要特别注意这一点。

基于GUI的应用程序也适用这一理论，这样就不会让它们的执行窗口“崩溃”掉。在这些应用程序里处理错误的正确办法是用一个错误消息对话框通知用户发生了什么事情，也许还可以修改参数使情况得到改善。

13~41行

我们的这个Python程序的核心部分是filefind()函数。filefind()函数需要两个执行参数：一个是用户准备搜索的字符串，另一个是准备对它进行搜索的文件的名字。

我们用一个计数器来记录成功匹配的总数（即包含搜索字符串的文本行个数）。下一个步骤是打开文件。try-except结构用来“捕捉”文件打开操作中可能出现的错误。Python语言的强项之一就是它能够让程序员对错误进行处理并执行相应的动作，而不是简单地退出程序运行。这就导致了更健壮的应用程序和更容易为人们接受的程序设计方法。关于错误和例外方面的内容请阅读第10章。

排除意外错误情况后，函数这一部分的目的是打开一个文件、把文本行读入到一个缓冲区里等待处理，再关闭文件。我们在前面已经介绍过：open()内建函数将返回一个文件对象（或者说一个文件句柄），其他任何后续操作，比如readlines()和close()，都必须通过这个句柄来完成。

函数的最后部分是逐个遍历每一个文本行，查找其中是否有搜索字符串。具体的搜索工作是用string模块中的find()函数完成的。如果出现匹配，find()函数将返回搜索字符串在文本行的起始位置；如果该文本行没有搜索字符串，就返回-1。成功匹配累加起来，并向用户显示匹配的文本行。

filefind()函数在结束时会显示已经找到的匹配文本行的总数。

sys.exc_info()是返回一个包含异常类、异常对象和反向跟踪对象的列表。它与类似于如下所示的操作没有区别。

```
>>> aList = [123, 'xyz', 45.67]
>>> print aList [1]
>>> xyz
```

43~52行

我们程序中的最后一个函数是checkargs()，它要做两件事情：检查命令行参数是否正确和调用filefind()函数完成具体的工作。命令行参数保存在sys.argv列表里。第一个参数是程序的名字，第二个是我们将要查找的字符串，最后一个参数是准备对它进行查找的文件的名字。

54~56行

这是我们前面介绍过的特殊代码：即用来确定（根据__name__变量）这个脚本程序被导入或直接执行两种情况下不同的执行路线。此处使用if语句以保证本模块被导入时checkargs()函数将不会被执行，当然我们也不想让它执行；如果是被导入，它会因为命令行参数检查的失败而退出执行。如果这段代码里没有if语句，主程序部分只有一个调用checkargs()函数的语句行，那么不管本模块是被导入还是直接运行，checkargs()函数都将会执行。

关于fgrepwc.py最后一点要说明的是：这个脚本程序是为命令行运行方式编写的。如果想让它从GUI或者Web环境中执行，就需要对它进行一番修改，主要是执行界面方面的修改。

这个例子是比较复杂的，但借助于本小节的说明和你以往的程序设计经验，读懂它应该没什么问题。我们将在下一章对Python语言中的对象、标准的数据类型及其分类方法等方面进行认真的讨论。

3.7 练习

3-1 标识符。为什么在Python语言里没有变量类型定义？

3-2 标识符。为什么在Python语言里没有变量名定义？

3-3 标识符。为什么要避免使用下划线字符做为变量名的开头？

3-4 语句。能否在一个文本行上写下多个Python语句？

3-5 语句。一条Python语句能否写在多个文本行上？

3-6 变量赋值。

a) 已知赋值语句“x, y, z = 1, 2, 3”，问x、y、z的值是多少？

b) 执行“x, y, z = y, z, x”之后，x、y、z的值是多少？

3-7 标识符。下面哪些是Python语言中的合法标识符？如果不是，又为什么？在那些不合法的标识符里，又有哪些是Python语言的关键字？

int32	40XL	char \$aving\$	printf	print
_print	a do	this self	__name__	0x40L
boolean	python	big-daddy	2hot2touch	type
thisIsn'tAVar	thisIsAVar	R_U_Ready		yes
if	no	counter-1	access	-

下面的问题与fgrepwc.py有关。

3-8 在刚才的fgrepwc.py程序里，读者可能已经注意到string.find()模块了。这个函数是干什么用的？执行成功或失败的返回值是什么？

3-9 我们从__name__变量入手对模块名进行了简要的讨论。如果直接执行fgrepwc.py脚本程序，那么这个变量的内容是什么？如果把fgrepwc用做一个导入模块，这个变量的内容又是什么？

3-10 fgrepwc模块里的usage()函数中有一个“-i”选项，但在整个程序里并没有涉及到它。这个选项的作用是以大小写敏感的方式进行搜索。请在fgrepwc.py里实现这一功能。你可以使用getopt()模块。

3-11 fgrepwc.py目前输出的是包含有搜索字符串的文本行总数。请修改这个脚本程序使其输出改为字符串在文本文件中出现的次数。换句话说，如果一个文本行里出现了多个搜索字符串的匹配，把这些情况都加在一起。

第4章 对 象

我们现在开始Python语言的核心之旅。我们先介绍Python语言中的对象是什么，然后对常用的内建数据类型进行讨论。接下来是对标准数据类型操作符和内建函数的介绍，然后对那些标准数据类型进行分类，这使我们能够更好地理解它们的工作情况。最后，我们向大家介绍一些Python语言中没有的数据类型（这主要是为那些具备其他高级语言使用经验的读者准备的）。

4.1 Python语言中的对象

Python语言在数据存储方面使用的是对象模型抽象。包含任意数据类型的任何构造都是对象。虽然Python被认为是一种“面向对象的程序设计语言”，但编写工作良好的Python应用程序并不一定要使用OOP技术。读者完全可以在不使用类及其实例的情况下用Python语言写出一个有用的脚本程序。但Python语言的对象语法和体系结构确实鼓励使用这类的行为。现在让我们来看看Python语言中的对象到底是什么。

Python语言中的一切对象都有下面三种特性：一个实体（identity）、一个类型（type）和一个值（value）。

实体（identity） 用来区分对象的独一无二的标识符。一个对象的标识符可以用内建函数id()获取。这个值可以说是该对象在Python语言中的“内存地址”（部分读者更容易理解这种说法）。你很少会用到这个值，因此不必太关心它到底是什么。

类型（type） 一个对象的类型指明该对象可以容纳什么样的值、对该对象可以进行什么样的操作以及该对象需要遵守什么样的行为准则。用内建函数type()可以查出某个Python对象的类型。因为类型也是Python语言中的一种对象（我们不是说过Python语言是面向对象的吗？），所以type()向你返回的实际也是一个对象而不是一个简单的类型名字。

值（value） 一个对象所代表的数值项。

创建对象时会同时对这三项内容进行赋值，除了对象的值，其他都是只读的。如果一个对象允许被修改，那只能改它的值；否则连值也是只读的。“一个对象的值能否被修改”叫做这个对象的可变性，我们将在4.7节讨论它。只要这个对象还存在，就少不了这些特性；如果一个对象被回收了，它们也会被收回。

Python语言支持一系列基本的（内建）数据类型，还有一些辅助性的数据类型只有在应用程序需要它们的时候才发挥作用。大多数应用程序只使用标准的数据类型，并且会为特殊的数据存储形式创建类及其实例。

对象的属性

有的Python对象会关联有属性、数据值或者（类中的）方法这样的可执行代码。属性是用属

性的点记号方式访问的，其中带有被关联对象的名字，我们在2.14节的编程提示里做过介绍。对象最常见的属性是函数和方法，但有些Python类型还关联有数据属性。带数据属性的对象包括（但不局限于）：类、类的实例、模块、复数和文件等。

4.2 标准数据类型

- 数字（又分为四种子类型）
 - 规则或“普通”整数
 - 长整数
 - 浮点实数
 - 复数
- 字符串（string）
- 列表（list）
- 表列（tuple）
- 字典（dictionary）

在本书里，我们把标准数据类型也称为“基本数据类型”，因为这些类型代表的是Python语言所提供的基本数据类型。我们将在第5、6、7章逐一介绍每一个类型。

编程提示：标准类型（type）不是类（class）

[CN]

在Java语言中，虽然基本数据类型是被支持的，但在需要一种数据类型时，它们常常会出现在一个将要创建其实例的类“打包器”里。在Python语言里，标准类型不是类，因此创建整数和字符串不会创建类的实例。同时这也意味着：尽管围绕一个Python类型打包一个类，再根据自己的需要修改那个类没有什么不对，你还是不能对一个标准类型进行划分子类的操作。Python还提供了一些仿真类型的类，可以对它们进行划分子类的操作。请参考13.15和13.16节。

4.3 其他内建的数据类型

- 类型（type）
- 空类型（none）
- 文件（file）
- 函数（function）
- 模块（module）
- 类（class）
- 类实例（class instance）
- 方法（method）

这些是你做为Python程序员工作时会遇到一些其他类型。我们将在第9、11、12和13章讨论它们，这里要讨论的是type和none类型。

4.3.1 type类型和type()内建函数

我们打算在这一章里向读者介绍Python语言中的所有类型。把类型本身也看做是对象多少有些不同寻常，但如果你还记得一个对象所继承的一系列行为和特性（比如它所支持的操作符和内建方法）都需要在什么地方定义，就会明白一个对象的类型是保存这些信息的最佳场所。描述一个类型所必须的信息是无法安排在一个字符串里的，因此类型不可能是一个简单的字符串；这些信息也不能与数据保存在一起；所以我们只能回到把类型看做是对象的路上来。

我们现在正式介绍type()内建函数。下面是它的语法：

```
type(object)
```

type()内建函数以对象为操作数，返回的是它的类型。返回对象是一个type对象。

```
>>> type(4)                #int type
<type 'int'>
>>>
>>> type('Hello World!') #string type
<type 'string'>
>>>
>>> type(type(4))          #type type
<type 'type'>
```

在上面的例子里，我们对一个整数和一个字符串进行了操作，并且用type()内建函数获取到它们的类型；为了验证类型本身也是类型，我们在type()内建函数里使用一个type()调用做为操作数。

请注意type()函数有趣的输出。它看上去不像是一个Python语言中典型的数据类型如一个数字或一个字符串，而是括在一对尖括号中的什么东西。这个语法通常表示你看到的东西是一个对象。对象可以带有一个可打印字符串；但并非总是如此。如果没有简单的办法可以“显示”某个对象，Python就会“近似”地显示一个代表该对象的字符串。一般采用如下所示的格式：<object_something_or_another>。以这种方式显示的对象通常还会给出它的类型、一个对象ID或地址或者是其他有关的信息。

4.3.2 None空类型

Python语言里有一个特殊的类型叫做空对象（Null object）。它只有一个值，None。None的类型还是None。它没有任何操作符或内建函数。如果读者熟悉C语言的话，与None类型最接近的C语言值就是NULL了（其他相似的对象和值还包括Perl语言中的undef和Java语言中的Void类型和null值）。None没有任何属性，对它的取值操作得到的永远是一个布尔假值（false）。

4.4 内部数据类型

- code（代码）
- frame（框架）
- traceback（跟踪记录）

- slice (序列切片)
- Ellipsis
- Xrange

我们将简单介绍一下这些Python语言的内部数据类型。一般的应用程序开发人员通常是不会直接和这些对象打交道的,但我们为了使内容完整起见把它们列在这里。详细资料请参考源代码或者Python的随机和在线文档。

为了让用户弄清例外的概念,它们现在是做为类而不是类型而实现的。在Python语言的旧版本里,例外是做为字符串而实现的。

4.4.1 代码对象

代码 (code) 对象是Python源代码经字节编译后得到的可执行片段,它们通常是compile()内建函数调用的返回值。这些对象适合通过exec或eval()内建函数执行。在第14章有对它们更进一步的讨论。

代码对象本身不包含任何与它们的运行环境有关的信息,但它们却位于每个用户定义函数的核心,而用户定义函数确实是包含有一些执行环境的上下文(做为代码对象的真正的字节编译代码是函数的一个属性)。一个函数的属性除了代码对象之外,还包含着一些必要的管理性支持,包括它的名字、文档字符串、缺省的参数以及全局性的变量名空间等。

4.4.2 框架对象

这些是Python语言中运行堆栈的对象。框架 (frame) 对象包含着Python解释器在一个实时运行环境过程中需要知道的全部信息。它的属性包括一个指向前一个堆栈框架的链接、正在执行的代码对象(见上文)、局部和全局变量名空间用的字典,以及当前指令。每个函数调用都会产生一些新的框架对象,而每个框架对象都会创建一个相应的C语言堆栈框架。访问框架对象的一个场所是跟踪记录 (traceback) 对象(请参考下面的内容)。

4.4.3 跟踪记录对象

如果在Python中出了错,就会引起一个例外 (exception)。如果例外没有被捕获或“处理”,解释器就会退出,同时给出一些像下面这样的诊断信息:

```
Traceback (innermost last):  
  File "<stdin>", line N?, in ???  
ErrorName:  error reason
```

跟踪记录traceback对象只是一个保存着例外的堆栈跟踪记录信息的数据项,它是在例外发生时创建的。如果为例外准备了一个处理器,那么这个处理器就能够访问该traceback对象。

4.4.4 序列切片对象

序列切片 (slice) 对象是在使用Python扩展的分离操作语法时创建的。这个扩展了的语法允许使用多种索引。这些索引方法包括步进索引法、多元索引法、以及使用Ellipsis类型的索引等。

多元索引的语法是: `sequence[start1:end1, start2:end2]`, 也可以使用枚举: `sequence[... , start1:end1]`。序列切片对象也可以用`slice()`内建函数生成。扩展的分离操作语法目前只有NumPy模块和JPython等第三方提供的外部模块支持使用。

序列类型的步进索引允许使用一个第三个分离操作元素进行步进方式的访问, 它的语法是: `sequence[starting_index:ending_index:stride]`。我们在这里使用JPython给出一个使用步进索引的示例, 如下所示:

```
% jpython
JPython 1.1 on java1.1.8 (JIT: sunwjit)
Copyright (C) 1997-1999 Corporation for National Research
Initiatives
>>> foostr = 'abcde'
>>> foostr[::-1]
'edcba'
>>> foostr[::-2]
'eca'
>>> foolist = [123, 'xba', 342.23, 'abc']
>>> foolist[::-1]
['abc', 342.23, 'xba', 123]
```

4.4.5 Ellipsis对象

Ellipsis对象用在扩展的分离记号中, 比如刚才给出的例子。这些对象在分离操作语法 (...) 里代表真正的枚举。类似于Null空对象, ellipsis对象也有一个独立的名字Ellipsis, 并且永远都有一个布尔真值 (true)。

4.4.6 Xrange对象

Xrange对象是由内建函数`xrange()`创建的, 它的功能与`range()`内建函数一样, 但用在内存不足或`range()`生成了一个非常之大的数据集的时候。对`range()`和`xrange()`的详细讨论请见第8章。

为了帮助读者更好地理解Python语言中的类型, 我们建议你去看看标准Python库中的`type`模块。

编程提示: 布尔值

[CN]

各种标准类型的对象都可以进行真假值检查, 也可以与类型相同的其他对象进行比较。对象有继承的真 (true) 或假 (false) 值。对象取假值的情况包括: 它们是空的; 0的各种数字表示法; Null空对象None。

下面这些情况在Python中被定义为有一个假值:

- None
 - 任何形式的0值:
 - 0 ([普通]整数)
 - 0.0 (浮点数)
 - 0L (长整数)
-

(续)

- 0.0+0.0j (复数)
- "" (空字符串)
- [] (空列表)
- () (空表列)
- {} (空字典)

一个对象的值如果不是上面列出的这些,就被认为有一个真(true)值,比如非空、非零等情况。对一个用户创建的类的实例来说,如果它的非零方法(`__nonzero__()`)或长度方法(`__len__()`) (如果定义了的话)返回了一个零值,就认为它有一个假(false)值。

4.5 与数据类型有关的标准操作符

4.5.1 值的比较

比较操作符用来判断同类型成员之间的数据值是否相等。各种内建类型都支持这些比较操作符。根据比较表达式是否成立,比较操作将产生真(true)或假(false)值。Python语言把真假值分别解释为普通整数0(假)和1(真),任何比较操作只能得出这两个值中的一个。Python语言的数值比较操作符列在表4-1中。

表4-1 标准类型的数值比较操作符

操作符	功 能
<code>expr1 < expr2</code>	表达式expr1小于表达式expr2
<code>expr1 > expr2</code>	表达式expr1大于表达式expr2
<code>expr1 <= expr2</code>	表达式expr1小于或等于表达式expr2
<code>expr1 >= expr2</code>	表达式expr1大于或等于表达式expr2
<code>expr1 == expr2</code>	表达式expr1等于表达式expr2
<code>expr1 != expr2</code>	表达式expr1不等于表达式expr2
<code>expr1 <> expr2</code>	表达式expr1不等于表达式expr2 (ABC/Pascal语言风格) ^①

① “<>”形式的不等于符号已经逐渐退出使用了,请使用“!=”。

请注意,比较操作的具体进行方式是由各个数据类型来决定的。换句话说,数值类型的对象将根据数字值的正负号和大小进行比较;字符串之间则依次比较处于同一位置上的字符,等等,如下所示:

```
>>> 2 == 2
1
>>> 2.46 <= 8.33
1
>>> 5+4j >= 2-3j
1
>>> 'abc' == 'xyz'
0
>>> 'abc' > 'xyz'
0
>>> 'abc' < 'xyz'
```

```

1
>>> [3, 'abc'] == ['abc', 3]
0
>>> [3, 'abc'] == [3, 'abc']
1

```

此外，与许多其他语言不同的是，Python允许在一个语句里使用多个比较，取值时按由左到右的顺序进行，如下所示：

```

>>> 3 < 4 < 7      # same as ( 3 < 4 ) and ( 4 < 7 )
1
>>> 4 > 3 == 3      与      ( 4 > 3 )  和  ( 3 == 3 ) 相同
1
>>> 4 < 3 < 5 != 2 < 7
0

```

我们要在这里指出的是比较操作严格限于对象的值与值之间，即比较操作针对的是数据的值而不是数据对象本身。数据对象本身的比较要用到下面将要介绍的对象实体比较操作符。

4.5.2 对象实体的比较

Python语言中除了值的比较之外，还提供了直接比较两个对象的记号方法，我们称之为“对象比较”。对象可以（通过引用线索）被赋值给另一个变量。因为多个变量指向的都是同一个（共享）的数据对象，所以通过一个变量进行的改动将影响到对象，再通过这个对象的所有引用线索反映出来。

为了更好地理解这一点，请读者暂时不要去考虑变量值本身；此时最好是把变量看做是到对象的链接。我们来看看下面的三个例子。

例1: foo1和foo2指向同一个对象

```
foo1 = foo2 = 4.0
```

从变量值的角度看上面这条语句时，你不过是执行了一次多重赋值操作，并且把数值4分别赋值给变量foo1和foo2。这是对的，但并不完全。在表面现象的背后，你会发现实际情况是创建了一个内容或值为4的数字对象；然后把这个对象的引用线索分别赋值给foo1和foo2，使foo1和foo2同时成为同一个对象的别名。图4-1给出了一个有两个引用线索的对象的示意图。



图4-1 foo1和foo2指向同一个对象

例2: foo1和foo2指向同一个对象

```
foo1 = 4.0
foo2 = foo1
```

这个例子和第一个很相似：创建一个值为4的数字对象，把它赋值给一个变量。当遇到语句“foo2 = foo1”时，变量foo2将直接指向对应于变量foo1的同一个对象，这是因为Python在处理对象时采用的是传递引用线索的办法。foo2就成为该对象的一个新的、额外的引用线索，foo1和foo2现在指向的是同一个对象。示意图也和上面的完全一样。

例3: foo1和foo2指向不同的对象

```
foo1 = 4.0
```

```
foo2 = 1.0 + 3.0
```

这个例子就不同了。首先，创建了一个数字对象，并且赋值给变量foo1。然后创建了第二个数字对象，这次把它赋值给变量foo2。虽然两个对象保存的数值是相同的，但系统内部实际上有两个彼此毫无关系的对象，变量foo1指向其中的一个，变量foo2指向另外一个。图4-2是虽然有同样的值，但实际是两个不同对象的示意图。



图4-2 foo1和foo2指向不同的对象

在上面的示意图里我们为什么要选择使用盒子呢？因为把对象想象为一个（里面装着东西的）盒子可以更形象地说明这个概念。在把一个对象赋值给一个变量的时候，就等于是盒子上贴了一个“标签”，表示已经给它加上了一个引用线索。以后，每给同一个对象增加一个引用线索的时候，就在盒子上多贴一个标签；当减少一个引用线索的时候，就撕下去一个标签。当全部标签都从盒子上被撕下去之后，那个盒子就可以被“回收”了。那么，系统是如何记录盒子上到底有多少个标签的呢？

每个对象都有一个关联的计数器，它跟踪记录着该对象上当前一共存在多少个引用线索。这个数字简单地指出有多少个变量指向那个特定的对象。这就是我们在3.5.5节到3.5.7节里所介绍过的“引用线索计数”。Python语言中的is和is not操作符用来测试两个变量是否指向同一个对象。具体的测试语句如下所示：

```
a is b
```

它相当于下面的表达式：

```
id(a) == id(b)
```

对象实体比较操作符的优先级都是一样的，请参考表4-2。

表4-2 标准类型对象实体的比较操作符

操作符	功 能
<code>obj1 is obj2</code>	obj1和obj2是同一个对象
<code>obj1 is not obj2</code>	obj1和obj2不是同一个对象

在下面的例子里，我们先创建一个变量，然后再创建另外一个，让它们两个都指向同一个对象。如下所示：

```
>>> a = [ 5, 'hat', -9.3]
>>> b = a
>>> a is b
1
```

```

>>> a is not b
0
>>>
>>> b = 2.5e-5
>>> b
2.5e-005
>>> a
[5, 'hat', -9.3]
>>> a is b
0
>>> a is not b
1

```

is和not这两个标识符都是Python语言中的关键字。

4.5.3 布尔表达式

表达式可以通过布尔逻辑操作符and、or和not连接在一起或者是取反，这几个都是Python语言中的关键字。我们把这些布尔操作的优先级按照从高到低的顺序排列在表4-3中。not操作符的优先级最高，只比各种比较操作符的优先级低一级。and和or操作符的优先级依次降低。

表4-3 标准类型的布尔操作符

操作符	功 能
not expr	表达式expr的逻辑非 NOT（否操作）
expr1 and expr2	表达式expr1和expr2的逻辑与 AND（交操作）
expr1 or expr2	表达式expr1和expr2的逻辑或 OR（并操作）

```

>>> x, y = 3.1415926536, -1024
>>> x < 5.0
1
>>> not (x < 5.0)
0
>>> (x < 5.0) or (y > 2.718281828)
1
>>> (x < 5.0) and (y > 2.718281828)
0
>>> not (x is y)
1

```

我们在前面已经介绍过：Python语言支持一个表达式中可以有多个比较。这些表达式是用隐含着的and连接在一起的。如下所示：

```

>>> 3 < 4 < 7      # same as "( 3 < 4 ) and ( 4 < 7 )"
1

```

4.6 与数据类型有关的标准内建函数

除我们刚才见到的各种操作符以外，Python语言还提供了一些能够作用于所有基本对象类型的内建函数，其中包括：`cmp()`、`repr()`、`str()`、`type()`和反单引号（```），反单引号和`repr()`的作用是完全一样的，见表4-4。

表4-4 标准类型的内建函数

函数	操 作
<code>com(obj1, obj2)</code>	对obj1和obj2进行比较，它的返回值是一个整数i，并且： $i < 0$ 如果 $obj1 < obj2$ $i > 0$ 如果 $obj1 > obj2$ $i == 0$ 如果 $obj1 == obj2$
<code>reper(obj) / 'obj'</code>	返回代表obj的取值字符串
<code>str(obj)</code>	返回代表obj的可打印字符串
<code>type(obj)</code>	确定obj的类型，返回一个type（类型）对象

4.6.1 `cmp()`

`cmp()`内建函数对两个对象（比如说obj1和obj2）进行比较。如果obj1比obj2小，就返回一个负值；如果obj1比obj2大，就返回一个正值；如果obj1等于obj2，就返回零值。它和C语言中`strcmp()`函数的返回值是相似的。不管进行比较的两个对象是一个标准的类型还是一个用户定义的类，在具体比较过程中使用的都是适用于该对象类型的操作；如果是后一种情况，`cmp()`会调用该用户定义类的特殊方法`__cmp__`。在讨论Python类的第13章里有对这些个特殊方法的详细论述。下面是一些使用`cmp()`内建函数对数字和字符串进行比较的例子：

```
>>> a, b = -4, 12
>>> cmp(a,b)
-1
>>> cmp(b,a)
1
>>> b = -4
>>> cmp(a,b)
0
>>>
>>> a, b = 'abc', 'xyz'
>>> cmp(a,b)
-23
>>> cmp(b,a)
23
>>> b = 'abc'
>>> cmp(a,b)
0
```

我们稍后再介绍几个对其他对象使用`cmp()`的例子。

4.6.2 `str()`和`repr()`

如果需要通过取值操作创建新的对象或者需要获取对象、数据值、对象类型等的可人工阅

读的内容,使用`str()`(取字符串)和`repr()`(取代表值)内建函数以及单反引号操作符(```)就非常方便。使用这些操作符时必须提供一个Python对象做为其执行参数,返回值将是一些代表该对象的类型或者字符串。

在下面的几个例子里,我们随机使用了一些Python类型,将其转换为它们的字符串表式形式:

```
>>> str(4.53-2j)
'(4.53-2j)'
>>>
>>> str(1)
'1'
>>>
>>> str(2e10)
'20000000000.0'
>>>
>>> str([0, 5, 9, 9])
'[0, 5, 9, 9]'
>>>
>>> repr([0, 5, 9, 9])
'[0, 5, 9, 9]'
>>>
>>> `[0, 5, 9, 9]`
'[0, 5, 9, 9]'
```

虽然这三个操作符在特性和功能方面很相似,但只有`repr()`和```完成的工作是一样的,使用它们可以获得一个对象“正式”的字符串表示形式,并且它能做为一个合法的Python表达式被取值(通过`eval()`内建函数)。与之对应的是:`str()`的作用是获取一个对象的可打印字符串表式形式,其结果可能不能用在`eval()`中,但适合用在`print`语句里。

区别在于`repr()`更适合于Python语言本身,而`str()`更适合于人们的阅读习惯。但因为这两种类型的字符串表式形式在很多情况下是一致的,所以它们三个返回的也经常是同样的字符串。

编程提示: 为什么有`repr()`和```两种操作符?

[CN]

在Python语言中,你有时会发现一个操作符和一个函数所做的事情是完全一样的。存在一个操作符和一个函数的原因之一是有时候一个函数比一个操作符更有用。举例来说,处理函数这样的可执行对象或者需要根据数据项的不同调用不同的函数时,函数就比操作符更方便。另外一个例子是双星号(`**`)和`pow()`内建函数,求 x 的 y 次方时可使用`x**y`或`pow(x, y)`两种方法。

4.6.3 深入`type()`

Python语言不支持方法或函数的复用,因此在调用函数时必须解决因为对象同名而产生的“冲突”(请参考Python语言的常见问题答疑FAQ 4.75)。幸运的是,有`type()`内建函数来帮助我

们应付这样的情况，这一点我们已经在4.3.1节里介绍过了。

名字里面有什么？如果它是一个类型的名字，就会有很多东西。根据对象的类型选择对它进行的计算处理通常是有利并且必要的。幸运的是，Python语言专门为此提供了一个内建函数。type()对任何Python对象都返回其类型，并且不仅仅局限于标准的类型。我们可以使用交互式解释器看看在下面的例子里，当我们给type()以不同的对象时，它都返回些什么内容。如下所示：

```
>>> type('')
<type 'string'>
>>>
>>> s = 'xyz'
>>>
>>> type(s)
<type 'string'>
>>>
>>> type(100)
<type 'int'>
>>>
>>> type(-2)
<type 'int'>
>>>
>>> type(0)
<type 'int'>
>>>
>>> type(0+0j)
<type 'complex'>
>>>
>>> type(0L)
<type 'long int'>
>>>
>>> type(0.0)
<type 'float'>
>>>
>>> type([])
<type 'list'>
>>>
>>> type(())
<type 'tuple'>
>>>
>>> type({})
<type 'dictionary'>
>>>
>>> type(type)
<type 'builtin_function_or_method'>
>>>
```

你会发现许多类型都是很面熟的，因为我们已经在本章的开始介绍过它们，不管怎么说，你现在应该见过Python是如何通过type()内建函数来识别各种类型的了。因为我们通常无法从交互式解释器的外部通过一个type（类型）对象看出它的值，所以type类型最好的用法就是把它与其他type对象进行比较，从而确定这一点。如下所示：


```
...
1.69 is a float type
```

表4-5是对能够作用于全体Python基本类型的操作符和内建函数的一个总结。它们的优先级按照从高到低的顺序排列。

表4-5 标准的类型操作符和内建函数

操作符/函数	说 明	结果 ^①
字符串表示形式		
''	字符串表示形式	字符串
内建函数		
cmp(obj1, obj2)	比较两个对象	整数
repr(obj)	字符串表示形式	字符串
str(obj)	字符串表示形式	字符串
type(obj)	确定对象的类型	type类型
数值比较		
<	小于	布尔值
>	大于	布尔值
<=	小于或等于	布尔值
>=	大于或等于	布尔值
=	等于	布尔值
!=	不等于	布尔值
<>	不等于	布尔值
对象比较		
is	两个对象是同一个	布尔值
is not	两个对象不是同一个	布尔值
布尔操作符		
not	逻辑非	布尔值
and	逻辑与	布尔值
or	逻辑或	布尔值

① 结果栏里注明为“布尔值”的表示是一个布尔比较操作；因为Python语言里没有布尔类型，因此其返回值是一个普通整数。0表示布尔假值（false），1表示布尔真值（true）。

4.7 标准数据类型的分类

在提到Python语言中的标准类型的时候，我们往往会把它们称为Python语言的“基本内建数据对象的初级类型”（basic built-in data object primitive types），其中：

- 基本（basic）表示它们是Python语言提供的标准或核心类型。
- 内建（built-in）指出这些类型都是Python语言缺省提供的这一事实。（不必太抠这个术语的字眼，因为我们不想把它和Python语言中的内建变量和内建函数混为一谈。）
- 数据（data），因为指出它们是与数据存储有关的事情。
- 对象（object），因为对象是数据和功能的缺省的抽象体。
- 初级（primitive），因为这些类型提供了数据存储的底层机制。

- 类型 (type)，因为它们就是数据的类型!

上面这些论述没有涉及到各类型的工作情况和对它们作用的功能情况。事实上，它们中有几个共享着某些特定的特性，比如它们功能的实现方法；其余的则在访问其数据值等方面有一定的共性。我们感兴趣的还有这些类型中保存的数据能否被修改以及它们提供了什么样的存储方式。

有三种不同的模型可以帮助我们对这些标准的类型进行分类，每种模型都向我们展示了类型之间的某些内在的联系。这些模型能够帮助我们更好地理解类型之间的相互关系以及它们的工作原理。

4.7.1 存储模型

对类型进行分类的第一个依据是该类型能够容纳多少个对象。Python语言中的类型（包括大多数其他语言中的类型）可以容纳一个或者多个值。我们把只能容纳一个对象的类型称为“文字”或“离散”存储，把能够容纳多个对象的类型称为“容器”存储（有关文档在提到容器类型的对象时往往会称之为“复合”对象；但复合对象有时指的不是类型而是其他对象，比如类的实例等。），容器类型又引出另外一个问题，就是能否存储不同类型的对象。Python语言中的容器类型能够保存不同类型的对象。表4-6根据存储模型对Python语言中的类型进行了分类。

表4-6 根据存储模型得到的类型分类

存储模型分类	属于这一分类的Python类型
文字/离散	数字（全部数值类型）、字符串
容器	列表、表列、字典

虽然字符串因为“包容”着字符（并且往往还不止一个字符）而看起来象是一个容器类型，但我们并不这样划分是因为Python语言中并没有一个字符类型（请参考4.8节的内容）。字符串是一个自我包容的文字类型。

4.7.2 修改模型

标准类型的另外一种分类方法是回答这样一个问题：“对象在创建之后还能再修改或者它们的值还能改变吗？”前面介绍Python类型时我们曾经指出：有的类型允许它们的值被改变，而有的就不行。我们把值可以改变的对象称为可变（mutable）对象；把值不能被改变的对象称为不可变（immutable）对象。表4-7给出了哪些类型允许修改，哪些不允许修改。

表4-7 根据修改模型得到的类型分类

修改模型分类	属于这一分类的Python类型
可变	列表、字典
不可变	数字、字符串、表列

看到这个表以后，你可能立刻会有这样的想法：“为什么说数字和字符串是不可变的？看看

下面这几个语句:

```
x = 'Python numbers and strings'
x = 'are immutable?!? What gives?'
i = 0
i = i + 1
```

“这看上去可不是不可变的!”这话没错,但并不完全,事情并不像看起来那么简单。在上面这几个例子里,表面现象背后实际发生了一些事情,最初的对象已经被替换了。确实如此,好好读读这一段。

原来的变量名获得了一个新创建的带有新值的新对象,不再指向原来的对象,那个旧对象被回收了。在赋值语句的前后用`id()`内建函数比较一下新旧两个对象实体就可以更清楚地确认这一点。

如果我们在上面的例子里加上`id()`调用,就可以看出对象在赋值语句前后发生了变化,如下所示:

```
x = 'Python numbers and strings'
print id(x)
x = 'are immutable?!? What gives?'
print id(x)
i = 0
print id(i)
i = i + 1
print id(i)
```

执行这小段代码之后,我们会得到下面的输出。根据读者使用的计算机系统的不同,这里的数字可能也不相同,它们和内存分配情况是密切相关的:

```
16191392
16191232
7749552
7749600
```

但在另一方面,列表在修改时是不会替换原来的对象的,这可以从下面的代码中看出来:

```
>>> aList = [ 'ammonia', 83, 85, 'lady' ]
>>> aList
['ammonia', 83, 85, 'lady']
>>>
>>> aList[2]
85
>>>
>>> id(aList)
135443480
>>>
>>> aList[2] = aList[2] + 1
>>> aList[3] = 'stereo'
>>> aList
['ammonia', 83, 86, 'stereo']
>>>
>>> id(aList)
135443480
>>>
```

```
>>> aList.append('gaudy')
>>> aList.append(aList[2] + 1)
>>> aList
['ammonia', 83, 86, 'stereo', 'gaudy', 87]
>>>
>>> id(aList)
135443480
```

请注意每次操作后，列表的ID还是保持着原来的样子。

4.7.3 访问模型

虽然前面介绍的两种类型分类模型对Python语言很有用，但它们并非区别类型的基本模型。因此我们再介绍一个访问模型。我们的意思是怎样才能访问存储数据的值呢？访问模型下有三种分类，它们是：直接（direct）访问、序列（sequence）方式访问和映射图方式访问（mapping）。这几种访问模型和归属于各分类的类型见表4-8。

表4-8 根据访问模型得到的类型分类

访问模型分类	属于这一分类的Python类型
直接访问	数字
序列方式访问	字符串、列表、表列
映射图方式访问	字典

直接访问类型指的是单个的元素，即不属于容器的类型。各种数值类型都归入这一分类中。

序列方式访问类型其元素需要通过下标值从0开始按顺序访问。被访问的数据项可以是单个的元素，也可以是一组元素——我们称之为切片（slice）。被归入这一分类的类型包括字符串、列表和表列。正像我们前面提到过的，Python语言不支持字符类型，因此，虽然字符串是文字，但因为需要按照一定顺序访问字符子串，所以它也属于序列方式访问类型。

映射图方式访问类型具有序列方式访问类型同样的下标特性，其区别在于它不使用一个顺序的数字偏移值对数据进行索引，它的元素（或值）是已经排好序的，访问时需要使用一个键字（key），这使映射图方式访问类型成为一个由键字-值对组成的哈希表形式。

我们在以后的章节里将主要使用这种模型，详细讲述各种访问模型的类型以及一个分类中的所有类型都有那些共同之处（比如操作符和内建函数等），然后讨论归入各个分类的每一种Python类型。某种类型特有的各种操作符、内建函数和方法将在相应的章节做详细说明。

那么，为什么要费力气对这些数据类型进行分类呢？首先，为什么要分类？因为Python语言提供的高级数据结构使我们需要把“初级”类型和那些功能更强大的扩展类型区别开。另外一个原因是要弄清楚某个类型应该具备哪些预期的行为。举例来说，当我们不再问自己“列表和表列之间有什么区别？”或者“哪些类型是不可变的，哪些是可变的？”等问题的时候，就说明我们已经达到目的了。最后，某些分类有一些适用于该分类中的所有类型的特性。一个高手应该知道自己的工具箱里都有些什么东西。

我们的第二个问题是为什么有这些不同的分类或不同的方面。那么多的数据类型看起来是很难分类的。它们之间有错综复杂的关系，因此最好把这些类型共有的各种关系——揭示出来。

我们还想让大家看看每一种类型都有什么独特之处。每一种类型都只能归入一个分类里。最后，我们相信搞清楚所有这些关系会在开发工作中起重要作用。对每种类型知道得越多，就越有可能在自己的应用程序里把最正确的类型用在最合适即最能提升性能的地方。

下面的汇总表（表4-9）里给出了所有的标准类型、我们对类型进行分类用的三种不同的模型和每个类型归入的分类。

表4-9 标准类型的分类

数据类型	存储模型	修改模型	访问模型
数字	文字/离散	不可变	直接
字符串	文字/离散	不可变	序列
列表	包容器	可变	序列
表列	包容器	不可变	序列
字典	包容器	可变	映射图

4.8 Python语言不支持的数据类型

在深入标准类型之前，我们先来看一下Python语言所不支持的类型的清单，并以此做为本章的结束内容。

1. Boolean（布尔）类型

Python语言没有Boolean类型，是用整数来代替的，这是它与Pascal和Java语言不一样的地方。

2. char（字符）或byte（字节）类型

Python语言没有用来保存单个字符的字符类型或者8比特整数的字节类型。单个字符使用的是长度为1的字符串类型；8比特数字归入整数类型。

3. pointer（指针）类型

因为内存管理是由Python语言负责的，所以不再需要访问指针地址。在Python语言里，与地址关系最近的操作是对一个对象的实体使用id()内建函数。但因为无法影响这个值，所以也没什么可说的。

4. int、short和long

Python语言的普通整数是全能的“标准”整数，有了它，就不再需要像C语言那样使用int、short和long三种不同的整数类型了。如果刨根问底的话，可以说Python语言中的整数相当于C语言中的long长整数。比普通整数（它通常就是计算机的体系结构尺寸，比如32位）更大的数值要使用Python语言的long类型整数。

5. float与double

C语言有一个单精度的float类型和一个双精度的double类型。Python语言的float类型实际上相当于C语言的double类型。Python语言没有单精度浮点类型，因为有这种类型的好处抵不上支持两种浮点类型所必须的开销。

4.9 练习

4-1 Python对象。与全部Python对象都有关联的三个值是什么？

4-2 类型。哪些Python类型是不可变的？

4-3 类型。哪些Python类型是序列？

4-4 type()内建函数。type()内建函数是做什么用的？type()返回的是哪一种对象——是一个整数还是一个对象？

4-5 str()和repr()内建函数。str()和repr()内建函数及反引号(``)操作符之间有什么区别？

4-6 对象比较。你认为表达式“type(a) = type(b)”和“type(a) is type(b)”之间有什么区别？

4-7 dir()内建函数。在练习2-12和2-13中，我们接触到一个名为dir()的内建函数，它以对象为操作数，给出的是该对象的属性。对types模块进行同样的操作。把你熟悉的类型写在一个清单里，包括你对其中每一种的全部了解；再把不熟悉的写在另外一个清单里。随着对Python语言更深入的学习，逐步把“不熟悉”清单上的东西划掉，写到“熟悉”清单里去。

第5章 数 字

我们将在本章介绍Python语言的数值类型。我们将详细讨论每一种类型，然后讲解可以用于数字的各种操作符和内建函数。最后以介绍对数字进行处理的几个标准库结束本章内容。

5.1 数字简介

数字具有文字和离散性质的存储和直接方式的访问。数字还是一种不可变类型，也就是说，对它的值进行修改将导致创建一个新的对象。这个活动对程序员和用户来说当然是透明的，因此不会对应用程序的开发产生什么影响。

Python语言有四种数字类型，它们是：“普通”整数、长整数、浮点实数和复数。

1. 如何创建并赋值数字（数字对象）

创建数字很简单，把一个数值赋值给一个变量就可以了，如下所示：

```
anInt = 1
aLong = -9999999999999999L
aFloat = 3.1415926535897932384626433832795
aComplex = 1.23 + 4.56j
```

2. 如何修改数字

给一个变量重新赋一个值就可以修改一个现有数字了。新值可以与原来的值有关系，也可以是一个全新的不同的值。

```
anInt = anInt + 1
aFloat = 2.718281828
```

3. 如何删除数字

正常情况下是用不着“删除”一个数字的；只要不再使用它就行了！如果你确实想删除某个数字对象的引用，用del语句（在3.5.6节介绍的）就行。变量名删除之后就不能再用了，除非又给它重新赋值一个对象；没有重新赋值就使用原来的变量名会引起一个NameError例外。

```
del anInt
del aLong, aFloat, aComplex
```

好了，知道怎样创建和修改数字后，我们来看看Python语言的四种数值类型。

5.2 整数

Python语言中有两种类型的整数，它们是“普通”整数和长整数。现阶段的大多数计算机系统都把普通整数识别为32位整数。Python语言中的长整数远远超过C语言中的long整数的表示范围。我们先来看看Python语言中的这两种整数类型，再对Python整数类型独有的操作符和内建函数做一番说明。

5.2.1 (普通) 整数

Python语言中的“普通”整数可以说是全能型的数值类型。在运行有Python的大多数机器(32位)上,它的取值范围是 -2^{31} 到 $2^{31}-1$,也就是从-2 147 483 648到2 147 483 647。下面是一些Python整数的例子:

```
0101      84      -237      0x80      017      -680      -0x92
```

Python整数在C语言中是以(带符号)long整数实现的。整数通常是以十进制数字表示的,但它们也能够用八进制或十六进制表示。八进制数字有一个“0”前缀,而十六进制数字有一个“0x”或者“0X”前缀。

5.2.2 长整数

关于Python语言的长整数首先要明确的是:不要把它和C语言或其他编译型语言中的长整数弄混了——这些编译型语言中长整数的长度一般被限制在32位或者64位,Python语言中长整数的长度只受你计算机(虚拟)内存容量的限制。换句话说,它们可以非常非常的长。

长整数是整数的一个超集,适用于需要在应用程序中使用超出普通整数表示范围的整数的情况,也就是要用到比 -2^{31} 小或比 $2^{31}-1$ 大的整数的情况。使用长整数时需要在该整数数字的尾部加上一个大写的“L”或小写的“l”字母,数值可以用十进制、八进制或十六进制表示。下面是一些长整数的例子:

```
16384L      -0x4E8L 017L  -2147483648L 052144364L
2997924581  0xDECADEDEADBEFFBADFEEDDEAL-5432101234L
```

编程风格:长整数要加上大写字母“L”

[CS]

虽然Python语言中的长整数可以用大写的“L”或小写的“l”字母标记,但我们建议你只使用大写的“L”字母以避免与数字一(1)弄混了。Python语言在显示长整数的时候是使用大写的“L”字母的。如下所示:

```
>>> aLong = 9999999991
>>> aLong
999999999L
```

5.3 浮点实数

Python语言中的浮点数对应于C语言中的double类型双精度浮点实数,其数值可以用十进制表示,也可以用科学计数法表示。这些八字节(64位)数值遵守IEEE协会第754文件中的规定(52M/11E/1S),即用52位表示底、11位表示幂、剩下1位表示正负号(其数值范围约为 $\pm 10^{308.25}$)。情况基本上就是如此;但你能够使用的实际精度(以及表示范围和越界处理)却完全取决于你计算机的体系结构和生成你Python语言解释器的编译器的具体情况。

浮点数值中会出现小数点;如果用科学计数法表示,还会有一个字母“e”,大小写均可。字母“e”后面的正(+)负(-)号是幂的符号,如果没有,就表示幂是正数。下面是一些浮点

数值的例子:

```
0.0      -777.      1.6      -5.555567119 96e3 * 1.0
4.3e25   9.384e-23 -2.172818 float(12)   1.0000000001
3.1416   4.2E-10   -90.      6.022e23   -1.609E-19
```

5.4 复数

在很久以前, 数学界被下面的等式难住了:

$$x^2 = -1$$

这是因为任何实数(不论正负)自乘的时候都会得到一个正数。什么数自乘后会得到一个复数呢? 这样的实数是不存在的。到了十八世纪, 数学家发明了叫做“虚数”的 i (或 j ——不同的教科书可能会使用不同的记号), 并且规定:

$$j = \sqrt{-1}$$

围绕着这个特殊的数字(或者叫概念)创建了一门新的数学分支, 现在虚数已经能够用在许多数值或科学计算应用程序中了。把一个实数和一个虚数写在一起就构成了一个“复数”。一个复数是一对浮点实数(x, y)以“ $x + yj$ ”的形式表示的, 其中的 x 是这个复数的实数部分, y 是这个复数的虚数部分。

下面是Python语言与复数有关的几个概念:

- Python语言是不支持虚数本身的。
- 复数由实数部分和虚数部分组成。
- 一个复数的语法是: `real + imagj`。
- 实数部分和虚数部分都是浮点值。
- 虚数部分必须加上后缀, 它可以是大写的“J”和小写的“j”字母。

下面是一些复数的例子:

```
64.375+1j   4.23-8.5j   0.23-8.55j   1.23e-045+6.7e+089j
6.23+1.5j   -1.23-875J  0+1j   9.80665-8.31441J   -.0224+0j
```

复数的内建属性

复数是一个带数据属性的对象的例子(请参考4.1.1节), 其数据属性就是该复数的实数部分和虚数部分。复数还有一个方法属性, 调用它的时候会返回该对象的复数记号形式, 如下所示:

```
>>> aComplex = -8.333-1.47j
>>> aComplex
(-8.333-1.47j)
>>> aComplex.real
-8.333
>>> aComplex.imag
-1.47
>>> aComplex.conjugate()
(-8.333+1.47j)
```

表5-1给出了复数所拥有的属性。

表5-1 复数的属性

属 性	说 明
<code>num.real</code>	复数num的实数部分
<code>num.imag</code>	复数num的虚数部分
<code>num.conjugate()</code>	返回num的复数记号形式

5.5 操作符

数值类型支持的操作符有很多，包括标准类型的操作符、为数字特意创建的操作符、甚至一些只能对整数进行操作的操作符。

5.5.1 混状态操作符

很难说那是什么时候了，但曾经有一个时期，当你把两个数字相加的时候，最重要的事情是要把两个“正确”的数字相加。加法一直使用着加号 (+)，但在程序设计语言里，事情就复杂化了，因为光是数字的类型就有许多种。

在把两个整数相加的时候，“+”代表的是整数加法；而在把两个浮点数相加的时候，“+”代表的是双精度浮点加法，以此类推。“+”的使用甚至可以扩展到Python语言的非数值类型中去。举例来说，字符串的“+”操作符表示对字符串进行合并操作，而不是加法操作，但使用的还是“+”操作符！问题的关键是：每一种支持“+”操作符的数据类型都已经预先规定好“+”操作符要完成什么样的功能，这也体现了“重载”的概念。

现阶段，我们还不能把数字和字符串相加，但Python语言在数值类型之间（也只有在数值类型之间）支持使用混状态操作。在把一个整数和一个浮点数相加的时候，必须选择是使用整数加法还是使用浮点加法，模棱两可是不行的。Python语言使用“数值协调”办法来解决这个问题，它的意思是在操作执行之前把一个操作数转换为与另一个操作数相同的类型。Python按照下面的规则进行“数值协调”：

如果两个数字的类型一开始就相同，则不必进行任何转换。如果两个数字的类型是不同的，就查看能否把一个数字转换为另一个数字的类型。如果能转换，开始操作，结果返回两个数字，其中的一个是经过类型转换得到的。转换必须遵守一定的规则，因为某些转换是不可能的，比如你不能把一个浮点数转换为一个整数，也不能把一个复数转换为任何非复数的数字类型。

能够进行的协调转换包括把一个整数转换为一个浮点数（在它后面加上“.0”）和把任一非复数类型转换为一个复数（加上一个零虚数部分，即“0j”）。协调转换的原则是：整数向浮点数转换，一切向复数转换。《Python Language Reference Guide》（Python语言参考指南）中对`coerce()`操作的描述是这样的：

- 如果两个数字中有一个是复数，把另外一个转换为复数。
- 否则，如果两个数字中有一个是浮点数，把另外一个转换为浮点数。
- 否则，如果两个数字中有一个是长整数，把另外一个转换为长整数。

- 否则，这两个数字必然都是普通整数，不需要进行转换操作（在下面给出的示意图里表示为最右边的箭头）。

下面就是这些协调转换规则的流程图：

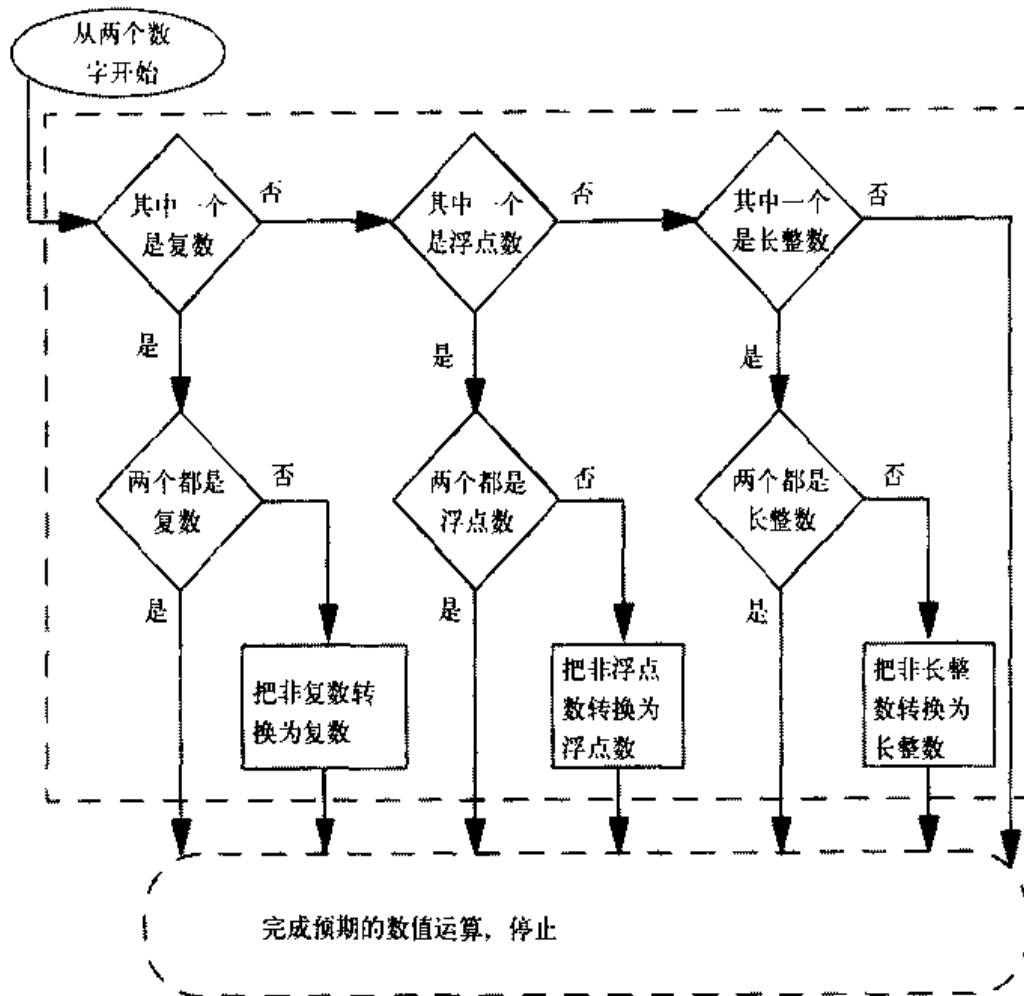


图5-1 数字协调转换

自动完成的数字协调解放了程序员，因为他们不再需要自己的应用程序里添加与数字协调有关的代码了。如果确实需要明确地执行协调操作，Python也准备好了一个`coerce()`内建函数（在5.6.2节里讲解）。

如果说数字协调和混状态操作有什么不足的话，那就是在一次操作的过程中是不进行协调的。举例来说，如果你把两个整数相乘，得到的结果超出了整数的表示范围，这个时候是不会进行数字协调的，你的操作将以失败告终。如下所示：

```
>>> 999999 * 999999
Traceback (innermost last):
  File "<stdin>", line 1, in ?
OverflowError: integer multiplication
```

这种情况的一个解决办法是事先检查一下是否会出现这样的情况，如果有可能出现这样的情况，就要在操作前利用`long()`内建函数人工把这两个整数转换为长整数。

下面是一个使用了Python语言的自动数字协调转换功能的例子。在操作执行之前，2被转换为一个长整数。

```
>>> 9999999L ** 2
9999980000001L
```

5.5.2 标准类型的操作符

在前面各章里介绍过的标准类型操作符都可以用在数值类型上。刚才介绍的混状态操作符是那些对两个不同类型的数字进行操作的操作符，在具体操作执行之前，有关数字将内部转换为同一种类型。

下面是一些能够用在数字上的标准类型操作符：

```
>>> 5.2 == 5.2
1
>>> -719 >= 833
0
>>> 5+4e >= 2-3e
1
>>> 2 < 5 < 9          # same as ( 2 < 5 ) and ( 5 < 9 )
1
>>> 77 > 66 == 66       # same as ( 77 > 66 ) and ( 66 == 66 )
1
>>> 0. < -90.4 < 55.3e2 != 3 < 181
0
>>> (-1 < 1) or (1 < -1)
1
```

5.5.3 数值类型操作符

Python语言支持单元操作符“+”（不改变数字）和“-”（把数字变为负数）；二元操作符“+”（加）、“-”（减）、“*”（乘）、“/”（除）、“%”（取除法余数）和“**”（乘方）。

规则和例外：如果除法和取除法余数操作的除数为零，将引起一个ZeroDivisionError例外。整数除法余数就是整数除法完成后剩余的整数；浮点除法余数是用被除数减去“浮点除法得数的下限整数与除数的积”（译者注：此处的引号是为了让读者能够看得更清楚）得到的，即“ $x - (\text{math.floor}(x/y) * y)$ ”或：

复数除法余数只取除法结果的实数部分，即“ $x - (\text{math.floor}((x/y).\text{real}) * y)$ ”。

乘方操作符与单元操作符之间的优先级关系是：乘方操作符的优先级高于出现在它左边的

$$x - \left\lfloor \frac{x}{y} \right\rfloor \times y$$

单元操作符，但低于紧挨着它出现在右边的单元操作符。因为这个原因，在下面的数值操作符汇总表里“**”操作符出现了两次。下面是几个例子：

```
>>> 3 ** 2
9
>>> -3 ** 2      # ** binds together than - to its left
-9
>>> (-3) ** 2    # group to cause - to bind first
```

```
9
>>> 4.0 ** -1.0 # ** binds looser than - to its right
0.25
```

在上面的第二个例子里，先求出3的2次方（即3的平方），然后再进行单元的求负操作。如果不想得到这样的结果，就要用括号把“-3”括起来。在最后一个例子里，最先执行的是单元求负操作，然后是1除以4的1次方，最后得数是四分之一（0.25）。需要注意的是，整数除法“1/4”的结果会是一个整数0，因此不能单独对整数求负数次幂（必须用浮点除法才能得到有用的结果），如下所示：

```
>>> 4 ** -1
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ValueError: integer to the negative power
```

表5-2是全体算术操作符的一个汇总，操作符按优先级从大到小的顺序依次排列。这个汇总表里所有操作符的优先级都比5.5.4节将要介绍的“整数逐位操作符”要高。

表5-2 数值类型算术操作符

算术操作符	功 能
<code>expr1 ** expr2</code>	<code>expr1</code> 的 <code>expr2</code> 次幂 ^①
<code>+expr</code>	（单元操作）不改变 <code>expr</code> 的符号
<code>-expr</code>	（单元操作）对 <code>expr</code> 求负
<code>expr1 ** expr2</code>	<code>expr1</code> 的 <code>expr2</code> 次幂 ^①
<code>expr1 * expr2</code>	<code>expr1</code> 乘以 <code>expr2</code>
<code>expr1 / expr2</code>	<code>expr1</code> 除以 <code>expr2</code>
<code>expr1 % expr2</code>	求 <code>expr1</code> 除以 <code>expr2</code> 的余数
<code>expr1 + expr2</code>	<code>expr1</code> 加上 <code>expr2</code>
<code>expr1 - expr2</code>	<code>expr1</code> 减去 <code>expr2</code>

① “**”操作符的优先级高于出现在它左边的单元操作符，但低于紧挨着它出现在右边操作符。

一定要注意整数除法的特性。要想得到正确的小数结果，必须用浮点数来做除法运算，如下所示：

```
>>> 3 / 4
0
>>> 3.0 / 4.0
0.75
```

下面再多给出一些Python语言数值操作符的使用示例：

```
>>> -442 - 77
-519
>>>
>>> 4 ** 3
64
>>>
>>> 4.2 ** 3.2
98.7183139527
>>> 8 / 3
```



```

2
>>> 8.0 / 3.0
2.666666666667
>>> 8 % 3
2
>>> (60. - 32.) * ( 5. / 9. )
15.55555555556
>>> 14 * 0x04
56
>>> 0170 / 4
30
>>> 0x80 + 0777
639
>>> 45L * 22L
990L
>>> 16399L + 0xA94E8L
709879L
>>> -2147483648L - 52147483648L
-54294967296L
>>> 64.375+1j + 4.23-8.5j
(68.605-7.5j)
>>> 0+1j ** 2      # same as 0+(1j**2)
(-1+0j)
>>> 1+1j ** 2      # same as 1+(1j**2)
0j
>>> (1+1j) ** 2
2j

```

请注意：乘方操作符的优先级要高于一个复数中连接实数部分和虚数部分的加法操作符（即复数中的“+”号）。试过两次之后，我们最后用括号把复数的两个部分括在一起得到了想要的结果。

5.5.4 *位操作符

在Python语言中，可以对整数逐位处理，它支持下面这些标准的位操作：求反；逐位的与（AND）、或（OR）和异或（XOR）操作；左右移位操作。下面是一些与位操作符有关的概念：

- 负数被当做它们的2的补集对待。
- 左右移N个位相当于用(2**N)做乘法或除法，但是不进行溢出检查。
- 对长整数操作时，位操作符采用一种“改进”的2的补集格式，就好像符号位向左无限扩展。

逐位求反操作符（“~”）与单元算术操作符的优先级一样，是所有位操作符中最高的；移位操作符（“<<”和“>>”）的优先级次之，比标准的加减操作符低一级；最后是逐位的AND（与）、XOR（异或）和OR（或）操作符（“&”、“^”、“|”）。表5-3按优先级从大到小的顺序列出了所有的位操作符。

表5-3 整数类型的位操作符

位操作符	功 能
<code>~num</code>	(单元操作) 对整数num逐位求反, 结果是 $-(num + 1)$
<code>num1 << num2</code>	expr1向左移expr2位
<code>num1 >> num2</code>	expr1向右移expr2位
<code>num1 & num2</code>	expr1和expr2做逐位的AND (与) 运算
<code>num1 ^ num2</code>	expr1和expr2做逐位的XOR (异或) 运算
<code>num1 num2</code>	expr1和expr2做逐位的OR (或) 运算

我们以整数30 (011110)、45 (101101) 和60 (111100) 为例给出几个位操作符的使用示例:

```
>>> 30 & 45
12
>>> 30 | 45
63
>>> 45 & 60
44
>>> 45 | 60
61
>>> ~30
-31
>>> ~45
-46
>>> 45 << 1
90
>>> 60 >> 2
15
>>> 30 ^ 45
51
```

5.6 内建函数

5.6.1 标准类型函数

在上一章里, 我们介绍了能够作用于全部标准类型的`cmp()`、`str()`和`type()`内建函数。对数字来说, 这些函数的功能分别是: 比较两个数字的大小、把数字转换为字符串、查看一个数字的类型。下面是使用这些函数的几个例子:

```
>>> cmp(-6, 2)
-1
>>> cmp(-4.333333, -2.718281828)
-1
>>> cmp(0xFF, 255)
0
>>> str(0xFF)
'255'
>>> str(55.3e2)
'5530.0'
```

```
>>> type(0xFF)
<type 'int'>
>>> type(98765432109876543210L)
<type 'long int'>
>>> type(2-1j)
<type 'complex'>
```

5.6.2 数值类型函数

Python语言目前支持多种数值类型的内建函数。有的函数用来把一种数值类型转换为另外一种，我们称之为转换函数；其他函数会对它们的数值参数进行一些计算，我们称之为可操作函数。

1. 转换函数

[1.5]

`int()`、`long()`、`float()`和`complex()`内建函数用来把其他数值类型转换为规定的类型。从Python 1.5开始这些函数也可以使用字符串作为参数，返回的是那些字符串代表的数字值。

下面是一些使用了数值类型转换函数的例子：

```
>>> int(4.25555)
4
>>> long(42)
42L
>>> float(4)
4.0
>>> complex(4)
(4+0j)
>>>
>>> complex(2.4, -8)
(2.4-8j)
>>>
>>> complex(2.3e-10, 45.3e4)
(2.3e-10+453000j)
```

表5-4列出了这些实现数值类型转换的内建函数。

表5-4 实现数值类型转换的内建函数

[1.6]

函数	操 作
<code>int(obj, base=10)</code>	把字符串或数字obj转换为（普通）整数；与 <code>string.atoi()</code> 的功能一样；可选的base参数（设定以什么进制进行转换）是从1.6版本开始出现的
<code>long(obj, base=10)</code>	把字符串或数字obj转换为长整数；与 <code>string.atol()</code> 的功能一样；可选的base参数（设定以什么进制进行转换）是从1.6版本开始出现的
<code>float(obj)</code>	把字符串或数字obj转换为浮点数；与 <code>string.atof()</code> 的功能一样
<code>complex(str)</code> 或 <code>complex(real, imag=0.0)</code>	把字符串str转换为复数；或者把real作为实数部分（也许还有作为虚数部分的imag）返回一个与之对应的复数

2. 操作型函数

Python语言里有五个数值类型的可操作内建函数，它们是：`abs()`、`coerce()`、`divmod()`、

`pow()`和`round()`。我们依次向大家介绍这些函数，并且给出一些使用示例。

`abs()`函数返回的是给定参数的绝对值。如果参数是一个复数，该函数返回的将是`math.sqrt(num.real2 + num.imag2)`。下面是一些使用了`abs()`函数的例子。

```
>>> abs(-1)
1
>>> abs(10.)
10.0
>>> abs(1.2-2.1j)
2.41867732449
>>> abs(0.23 - 0.78j)
0.55
```

`coerce()`函数从技术的角度看是一个数值类型的转换函数，但它完成的转换更像是一个操作符的行为，因此我们也把它归入到操作型内建函数里去。在5.5.1节里，我们介绍了协调转换操作以及如何在Python语言里实现它。

`coerce()`函数使程序员能够明确地协调两个数字的类型，而不依赖于解释器完成协调工作。在为新创建的数值类（numeric class）类型定义有关操作的时候，这个函数非常有用。`coerce()`函数返回一个表列，表列中包含的是转换后的一对数字。下面是几个例子：

```
>>> coerce(1, 2)
(1, 2)
>>>
>>> coerce(1.3, 134L)
(1.3, 134.0)
>>>
>>> coerce(1, 134L)
(1L, 134L)
>>>
>>> coerce(1j, 134L)
(1j, (134+0j))
>>>
>>> coerce(1.23-41j, 134L)
((1.23-41j), (134+0j))
```

`divmod()`内建函数把除法和取除法余数操作结合到一个函数里来，返回的是一个表列形式的(商, 余数)对。这两个返回值与整数类型除法和取除法余数操作符的执行结果是一样的。如果它的参数是浮点数，那么返回的商是`math.floor(num1/num2)`；如果它的参数是复数，那么返回的商是`math.floor((num1/num2).real)`。请看下面的例子：

```
>>> divmod(10,3)
(3, 1)
>>> divmod(3,10)
(0, 3)
>>> divmod(10,2.5)
(4.0, 0.0)
>>> divmod(2.5,10)
(0.0, 2.5)
>>> divmod(2+1j, 0.5-1j)
(0j, (2+1j))
```

`pow()`函数和“`***`”操作符执行的都是乘方运算，但它们之间的区别并不仅仅是一个是操作

符, 另一个是内建函数。

[1.5]

在Python 1.5之前是没有“**”操作符的, 而pow()内建函数还有一个可选的第三个参数, 即取除法余数参数。如果带有这个参数, pow()将首先执行乘方操作, 最后的返回结果是对第二个参数取除法余数的结果。这个形式主要用在图形或加密应用程序中, 它比“pow()%z”的性能要好, 因为后者是用Python语言而不是C语言中的pow(x, y, z)实现的。请看下面的例子: [1.5.2]

```
>>> pow(2,5)
32
>>>
>>> pow(5,2)
25
>>> pow(3.141592,2)
9.86960029446
>>>
>>> pow(1+1j, 3)
(-2+2j)
```

round()内建函数的语法是: round(flt, ndig=0)。它对浮点数进行操作, 其结果是与之最接近的整数, 但返回值仍然以浮点数的形式表示。可选的第二个参数告诉round()函数把结果精确到小数点后面的指定位数。请看下面的例子:

```
>>> round(3)
3.0
>>> round(3.45)
3.0
>>> round(3.4999999)
3.0
>>> round(3.4999999, 1)
3.5
>>> import math
>>> for eachNum in range(10):
...     print round(math.pi, eachNum)
...
3.0
3.1
3.14
3.142
3.1416
3.14159
3.141593
3.1415927
3.14159265
3.141592654
3.1415926536
>>> round(-3.5)
-4.0
>>> round(-3.4)
-3.0
>>> round(-3.49)
-3.0
>>> round(-3.49, 1)
-3.5
```

请注意, `round()`完成的求整操作是按四舍五入的规则执行的, 即`round(.5)`得数为1; 而`round(-.5)`得数为-1。这样看起来, `int()`、`round()`和`math.floor()`这几个函数干得好像都是同一件事情, 很容易把它们搞混。下面是它们之间的区别之处:

- `int()`直接去掉数字的小数点及小数部分 (截断)。
- `floor()`函数的结果是最接近原参数但又小于原参数的整数, 即沿数轴负方向前进遇到的第一个整数。
- `round()`函数的结果是与原参数最接近的整数 (求整)。

下面的例子用四个正数和四个负数做参数, 分别求出这八个数用上面三个函数调用后的结果 (为了更清晰地比较这三个函数的执行结果, 我们把`int()`函数的结果也转换为浮点数。)。如下所示:

```
>>> import math
>>> for eachNum in (.2, .7, 1.2, 1.7, -.2, -.7, -1.2, -1.7):
...     print "int(%.1f)\t%.1f" % (eachNum, float(int(eachNum)))
...     print "floor(%.1f)\t%.1f" % (eachNum,
...     math.floor(eachNum))
...     print "round(%.1f)\t%.1f" % (eachNum, round(eachNum))
...     print '-' * 20
...
int(0.2)          +0.0
floor(0.2)         +0.0
round(0.2)         +0.0
-----
int(0.7)          +0.0
floor(0.7)         +0.0
round(0.7)         +1.0
-----
int(1.2)          +1.0
floor(1.2)         +1.0
round(1.2)         +1.0
-----
int(1.7)          +1.0
floor(1.7)         +1.0
round(1.7)         +2.0
-----
int(-0.2)          +0.0
floor(-0.2)        -1.0
round(-0.2)        +0.0
-----
int(-0.7)          +0.0
floor(-0.7)        -1.0
round(-0.7)        -1.0
-----
int(-1.2)         -1.0
floor(-1.2)        -2.0
round(-1.2)        -1.0
-----
int(-1.7)         -1.0
floor(-1.7)        -2.0
round(-1.7)        -2.0
```

表5-5是对数值类型的操作型函数进行的汇总。

表5-5 数值类型的操作型内建函数^①

函 数	操 作
<code>abs(num)</code>	返回num的绝对值
<code>ccerce(num1, num2)</code>	把num1和num2转换为同一数值类型, 返回一个表列形式的转换结果
<code>divmod(num1, num2)</code>	除法-取余数操作的结合, 返回一个(num1/num2, num1%num2)形式的表列 对浮点数和复数的商进行“下舍入”复数只取实数部分的商
<code>pow(num1, num2, mod=1)</code>	求num1的num2次幂; 如果还有mod参数, 再对它求除法余数
<code>round(flt, ndig=0)</code>	(只对浮点数操作) 以浮点数flt为参数, 把它求整到小数点后面第ndig位。 如果ndig缺省, 就等于是求最接近整数操作

① 其中只有round()是只对浮点数起作用的。

5.6.3 只适用于整数的函数

除适用于各种数值类型的内建函数外, Python语言中还支持一些只用于整数(普通整数或长整数)的函数。这些函数又分为两大类, 一类是与进制有关的hex()和oct(), 另一类是与ASCII转换有关的chr()和ord()函数。

1. 与进制有关的函数

我们已经在前面看到过, Python整数除十进制外还自动支持八进制和十六进制的表示方式。此外, Python语言里还有两个内建函数专门用来返回一个整数的八进制和十六进制字符串表示方式。这两个函数是hex()和oct()内建函数。它们都以一个整数(可以是任意表示方式)对象为参数, 返回一个对应值的字符串。下面是几个使用这两个函数的例子;

```
>>> hex(255)
'0xff'
>>> hex(230948231)
'0x1606627L'
>>> hex(65535*2)
'0x1ffffe'
>>>
>>> oct(255)
'0377'
>>> oct(230948231)
'0130063047L'
>>> oct(65535*2)
'0377776'
```

2. 与ASCII转换有关的函数

Python语言里还提供了这样的函数, 它们可以在ASCII(American Standard Code for Information Interchange, 美国信息交换标准代码)字符及其对应的整数值之间来回转换。每个字符都被映射到一个从0到255的表格中的独一无二的数字上去。在使用ASCII表的任何计算机上, 这个数字都是一致的, 这就可以在不同系统之间保持数据的一致性和程序行为的统一性。chr()

函数以一个单字节整数为参数，返回一个单字符字符串，其内容是与之对应的ASCII字符。ord()的作用正好相反，它的参数是一个以长度为一的字符串表示的ASCII字符，返回的是与该ASCII字符对应的一个整数值。如下所示：

```
>>> ord('a')
97
>>> ord('A')
65
>>> ord('0')
48

>>> chr(97)
'a'
>>> chr(65)
'A'
>>> chr(48)
'0'
```

表5-6里列出了所有适用于整数类型的内建函数。

表5-6 适用于整数类型的内建函数

函 数	操 作
<code>hex(num)</code>	把num转换为十六进制，并以字符串形式返回
<code>oct(num)</code>	把num转换为八进制，并以字符串形式返回
<code>chr(num)</code>	把ASCII值num转换为对应的ASCII字符，并以字符串形式返回；其中的num必须满足 $0 \leq \text{num} \leq 255$
<code>ord(chr)</code>	把ASCII字符chr转换为对应的ASCII数值；chr必须是一个长度为1的字符串

5.7 相关模块

Python语言的标准库里有許多模块提供了对数值类型进行多种操作的操作符和内建函数功能。表5-7列出一些适用于数值类型的关键性模块。有关这些模块的详细资料请参考它们的材料或在线文档。

表5-7 与数值类型有关的模块

模 块	内 容
<code>array</code>	实现数组类型，这是一个有一定操作限制的序列类型
<code>math/cmath</code>	提供标准的C语言数学库函数；math库中的大多数函数都是在cmath模块里对复数实现的
<code>operator</code>	包含着以函数形式实现的数值操作符，比如说： <code>operator.sub(m, n)</code> 就相当于求数字m和n的差(m-n)
<code>random</code>	Python语言中缺省的RNG模块，不再使用rand和whrandom模块

对那些高级的数字和科学计算类应用程序来说，还有一个名为NumPy的扩展模块，读者肯定会对它感兴趣。

模块: random

[CM]

如果你需要用到随机数字,一般可以求助于random模块。它的随机数发生器(random number generator,简称RNG)基于Wichmann-Hill算法,以当前的系统时间值做为种子,只要加载了这个模块,就可以随时提供随机数字。下面是random模块中最常用的四个函数:

randint()以两个整数为参数,返回一个这两个整数之间的随机整数(包括做为参数的那两个整数)。

uniform()与randint()的功能差不多,但它返回的是一个浮点数;返回值可以取那个比较小的参数的数值(但不能是那个比较大的参数数值)。

random()与uniform()的功能差不多,但那个比较小的参数其数值固定为0.0,那个比较大的参数的数值固定为1.0。

choice()给定一个序列(请参考第6章内容),随机选择并返回该序列中的某个数据项。

讲到这儿,该是总结我们的Python数值类型之旅的时候了。对适用于数值类型的操作符和内建函数的汇总请见表5-8。

表5-8 数值类型的操作符和内建函数

操作符/内建函数	说明	int	long	float	complex	结果①
abs()	绝对值	•	•	•	•	数字①
chr()	字符	•	•			字符串
coerce()	数值协调	•	•	•	•	表列
complex()	复数转换	•	•	•	•	复数
divmod()	除法/取除法余数	•	•	•	•	表列
float()	浮点转换	•	•	•	•	浮点数
hex()	十六进制操作符	•	•			字符串
int()	整数转换	•	•	•	•	整数
long()	长整数转换	•	•	•	•	长整数
oct()	八进制操作符	•	•			字符串
ord()	ASCII顺序值			(string)		整数
pow()	乘方	•	•	•	•	数字
round()	浮点四舍五入			•		浮点数
**②	乘方	•	•	•	•	数字
+③	无变化	•	•	•	•	数字
-①	求负	•	•	•	•	数字
~②	按位求反	•	•			整数/长整数
**②	乘方	•	•	•	•	数字
*	乘法	•	•	•	•	数字
/	除法	•	•	•	•	数字
%	取除法余数	•	•	•	•	数字
+	加法	•	•	•	•	数字
-	减法	•	•	•	•	数字
<<	左移位	•	•			整数/长整数
>>	右移位	•	•			整数/长整数

(续)

操作符/内建函数	说明	int	long	float	complex	结果 ^①
&	按位与操作AND	•	•			整数/长整数
^	按位异或操作XOR	•	•			整数/长整数
	按位或操作OR	•	•			整数/长整数

① 结果栏中的“数字”表示可以是四种数值类型中的任何一种。

② “*”操作符与单元操作符的关系是独有的。请参考5.5.3节和表5-2。

③ 单元操作符。

5.8 练习

本章中的练习可以先以应用程序的形式实现。在确保功能及程序正确性的前提下，我们建议读者把自己的代码转换为函数的形式，这样就可以把它们用在今后的练习中。在此需要在编程风格方面提醒大家注意的是：不要在被调用函数中使用print语句，相反，应该让函数返回一个适当的结果，由调用函数完成预定的输出操作。这样可以使代码具备可移植性和可重复使用性。

5-1 整数。请说出Python语言中的普通整数和长整数之间的区别。

5-2 操作符。a) 编写一个函数把（任何类型的）两个参数相加并输出它们的和。

b) 再编写一个函数输出两个给定数字的积。

5-3 标准类型操作符。由用户输入考试分数，然后根据下面的规律输出与之对应的成绩水平：

A: 90 - 100

B: 80 - 89

C: 70 - 79

D: 60 - 69

F: < 60

5-4 模块。使用下面的公式确定某个年份是否是闰年：闰年的年份数字可以被4整除，但不能被100整除，除非它能被400整除。举例来说，1992年、1996年和2000年是闰年。但1967年和1900年就不是闰年。下一个能被400整除的闰年是2400年。

5-5 除法余数。随意取一个小于一美元的钱数，然后计算需要由多少枚硬币才能得出相应的钱数。美元硬币有四种，分别是：一美分、五美分、一角（10分）、二角五分（25分）。100美分等于一美元。对给定的一个小于一美元的钱数（如果使用浮点数，先把它转换为整数），请计算出需要多少枚各种面值的硬币才能凑够该钱数，但要尽量使用面值最大的硬币。举例来说，如果给定的钱数是\$0.76，正确的输出应该是“三枚二角五分硬币和一枚一美分硬币”。输出“76枚一美分硬币”或“二个二角五分硬币、二个一角硬币、一个五美分硬币和一个一分硬币”这样的答案是不正确的。

5-6 算术。编写一个计算器程序。在这段程序代码中，输入两个数字和一个操作符时请采用输入格式：“N1 OP N2”；其中的N1和N2是浮点数或整数，操作符可以是“+”、“-”、“*”、“/”、

“%”、“*”中的一个，分别代表加法、减法、乘法、除法、取除法余数和乘方，最后要把输入参数的计算结果显示出来。

5-7 税收计算。随意设定一个分期付款总额（可以是浮点美元数额，也可以是任何币种），然后根据你居住地的税率计算出需要为此支付的销售税总数。

5-8 几何。计算下面各项的面积和体积：

a) 矩形和立方体

b) 圆形和球体

5-9 整数的进制与编程风格。请回答下面关于数字格式方面的问题：

a) 在下面的例子里，为什么17+32等于49，但017+32等于47，而017+032等于41？

```
>>> 17 + 32
49
>>> 017+ 32
47
>>> 017 + 032
41
```

b) 在下面的例子里，为什么561+781等于134L而不是1342？

```
>>> 561 + 781
134L
```

5-10 转换。请编写一个华氏和摄氏温度值相互转换的程序。公式 $C = (F - 32) * (5 / 9)$ 可以帮助你开始程序设计。

5-11 余数。a) 使用循环和数值操作符求出0到20之间的全部偶数。

b) 同上，但是要求输出全部的奇数。

c) 根据a)和b)，区别偶数和奇数的最简单的方法是什么？

d) 从c)出发，编写一段代码检查一个数是否能够被另一个数整除。在你的代码里，先要求用户输入两个数字，再根据两个数是否有整除关系的yes或no回答让函数分别返回1或0。

5-12 数字极限。请确定你的计算机系统能够处理的整数、长整数、浮点数和复数的最大值与最小值。

5-13 转换。编写一个函数把由小时和分钟表示的时间转换为只以分钟表示的时间。

5-14 银行帐户利息。编写一个函数以某种银行帐户（比如定期存款帐户）的利息率为参数，计算在符合帐户余额最小要求的前提下的年本息合计与年回报率。

5-15 最大公约数和最小公积数。请确定一对整数的最大公约数和最小公积数。

5-16 家庭财务。给定一个初始余额和月支出数。用一个循环确定随后每个月的余额，包括最后一笔支出。这个“Payment()”函数要用到初始余额和月支出两个输入参数，输出应该是类似于下表的格式（下面例子中的数字只供演示用）：

```
Enter opening balance:100.00
Enter monthly payment: 16.13
```

Pymt #	Amount Paid	Remaining Balance
-----	-----	-----
0	\$ 0.00	\$100.00
1	\$16.13	\$ 83.87
2	\$16.13	\$ 67.74
3	\$16.13	\$ 51.61
4	\$16.13	\$ 35.48
5	\$16.13	\$ 19.35
6	\$16.13	\$ 3.22
7	\$ 3.22	\$ 0.00

5-17 *随机数。阅读random模块并解决下面的问题：生成一个随机个数 ($1 < N \leq 100$) 的随机数 ($0 \leq n \leq 2^{31}-1$) 清单。然后随机选取一些随机数 ($1 < N \leq 100$)，对它们进行排序，再把这个随机数子集显示出来。

第6章 序列：字符串、列表和表列

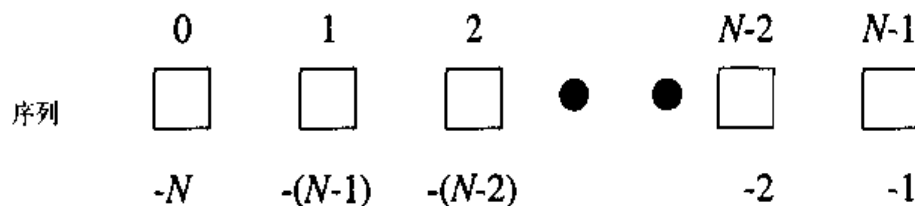
我们将要探索的下一个Python类型的家族是那些数据项按一定顺序排列，需要通过下标偏移量存取其数据项的数据组。这个类型统称为序列（sequences），其中包括以下类型：字符串、列表和表列。我们先介绍其常见的共同特性，然后再详细讨论每一种类型。我们将从介绍适用于序列类型的各种操作符和内建函数入手，再依次讨论每一种序列类型。对每一种序列类型，我们将提供下面的资料：

- 介绍
- 操作符
- 内建函数
- 内建方法（如果有的话）
- 特殊功用（如果有的话）
- 相关模块（如果有的话）

我们在本章的结尾为大家准备了一个适用于全部序列类型的操作符和内建函数的汇总表。我们先从大方面开始，对能够作用于全体序列类型的操作符和内建函数做个介绍。

6.1 序列

各种序列类型的访问模型是一样的：在次序固定的数据集合里通过顺序增加的下标到达各个元素。要想一次存取多个元素，就必须使用将在本章介绍的分离操作符。下标的标注方法是从零（0）开始，以序列长度减一结束，这是因为我们是从0开始计算下标的。图6-1给出了序列中数据项的存储示意图。



$N = \text{序列长度} = \text{len}(\text{序列})$

图6-1 序列中的数据项是如何存储和访问的

6.1.1 操作符

能够对全体序列类型进行操作的全部操作符列在表6-1中。表中的操作符是以从高到低的继承性顺序排列的，不同级别用不同的阴影效果加以区别。

表6-1 序列类型的操作符

序列操作符	功 能
<code>seq[ind]</code>	序列seq中下标为ind的元素
<code>seq[ind1: ind2]</code>	序列seq中下标从ind1到ind2之间的元素
<code>seq * expr</code>	序列seq重复expr次
<code>seq1 + seq2</code>	合并序列seq1和seq2
<code>obj in seq</code>	测试obj是否是序列seq中的一个元素
<code>obj not in seq</code>	测试obj是否不是序列seq中的一个元素

1. 是序列成员吗 (in, not in) ?

序列成员测试操作符用来确定一个元素是否是某个序列的成员。对字符串来说，这个测试的作用是确定它是否出现在该字符串中；对列表和表列来说，就是确定一个对象是否是该序列的一个元素。“in”和“not in”操作符的结果是布尔值，如果确定是一个序列的成员，它返回整数值1；否则返回整数值0。

成员操作符的使用语法如下所示：

```
obj [not] in sequence
```

2. 合并操作 (+)

这个允许我们把一个序列和另外一个同类型的序列合并在一起。合并操作符的使用语法如下所示：

```
sequence1 + sequence2
```

其结果表达式是是一个新的序列，其中包含着序列sequence1和sequence2合并后的内容。

3. 重复操作 (*)

如果某些个元素需要在序列中反复出现，重复操作符就非常有用。重复操作符的使用语法如下所示：

```
sequence * copies_int
```

表示复制次数的copies_int必须是一个普通整数，不允许是一个长整数或其他类型的数值。重复操作符返回的对象是一个新分配的、其中包含着多次重复的对象内容。

从Python 1.6版本开始copies_int可以是一个长整数了。 [1.6]

4. 分离操作 ([], [:])

序列是结构化的数据类型，它的元素是按照一定顺序排列好了的。这种格式就允许我们通过下标或者下标范围访问其中的元素，从序列中顺序“抽取”一段元素。这种访问类型叫做“分离”（有的书里称为“切片”），我们可以通过分离操作符实现这种类型的数据访问。

访问序列中单个元素的语法如下所示：

```
sequence[index]
```

其中：sequence是序列的名字，而index是准备访问的元素在序列中的下标。下标值可以从0开始到序列长度减1，即“ $0 \leq \text{index} \leq \text{len}(\text{sequence})$ ”范围内的正数；也可以是从-1开始到序列长度负数值-len(sequence)，即“ $-\text{len}(\text{sequence}) \leq \text{index} \leq -1$ ”范围内的负数。正负下标

值的区别在于：正数下标从序列的头部开始，而负数下标值是从序列的尾部开始。

访问一组元素的情况也与此类似。此时需要给出起始下标和结束下标两个值，中间用冒号(:) 隔开。访问一组元素的语法如下所示：

```
sequence[starting_index : ending_index] !
```

通过这个语法，我们就可以分离出序列中起始下标*starting_index*到结束下标*ending_index*（但不包括结束下标*ending_index*处的元素）之间的元素“切片”来。*starting_index*和*ending_index*都是可选的，如果没有给出，就表示从序列头开始和到序列尾结束。

在下面的图6-2到图6-6里，我们以一个长度为5的完整序列（足球运动员们）为例说明如何进行各种分离切片操作。



图6-2 完整的序列: `sequence`或`sequence[:]`



图6-3 序列切片: `sequence[0:3]`或`sequence[:3]`



图6-4 序列切片: `sequence[2:5]`或`sequence[2:]`



图6-5 序列切片: `sequence[1:3]`

我们将在介绍每一种序列类型的时候对分离切片操作做进一步的讲解。

图6-6 序列切片: `sequence[3]`

6.1.2 内建函数

1. 序列转换函数

`list()`、`str()`和`tuple()`内建函数用来把其他类型的序列转换为规定的类型。表6-2给出了这几个序列类型转换函数的说明。

表6-2 序列类型转换内建函数

函 数	操 作
<code>list(seq)</code>	把序列seq转换为列表
<code>str(obj)</code>	把序列seq转换为字符串
<code>tuple(seq)</code>	把序列seq转换为表列

我们在使用“转换”这个词的时候并不是很严谨。它实际上并没有把作为其参数的对象转换为别的类型；Python对象在创建之后，其实体（identity）或类型是不能被改变的。因此，这些函数实际上是按要求的类型另外创建了一个新的序列，然后把新序列做为返回值传递回来。这与6.6.1节中介绍的合并和重复操作是同一种思路。

`str()`函数最适合把一个对象转换为可打印的东西，它也能对其他类型的对象进行操作，并不局限于序列。`list()`和`tuple()`函数很适合彼此转换（把列表转换为表列或反过来）。但是，虽然`list()`和`tuple()`函数能够对字符串进行操作，并且操作符也是一种序列，可用它们把字符串转换为表列或是列表的情况是不常见的。

2. 操作性函数

Python语言为序列类型准备了下面这些操作性的内建函数（请参考表6-3）。

现在，我们已经做好依次学习每一种序列类型的准备工作了，让我们的旅程从Python语言中的字符串开始吧。

表6-3 序列类型的操作性内建函数

函 数	操 作
<code>len(seq)</code>	返回序列seq的长度（数据项个数）
<code>max(seq)</code>	返回序列seq中“最大的”元素
<code>min(seq)</code>	返回序列seq中“最小的”元素

6.2 字符串

字符串是Python语言中最常用的数据类型之一。只要用引号把字符括起来就可以创建一个字符串。在Python语言中单引号和双引号的作用是一样的。这与其他脚本语言不太一样，在其他脚本语言中，使用单引号的字符串就是一般的文字，而使用双引号的字符串具备字符转义功能。Python语言把引号解释为“纯字符串”操作符，创建的操作符都是文字性的，因此两种引号没有什么区别。其他语言如C等对单个字符要使用单引号，对字符串要使用双引号。在Python语言里不存在字符类型，这可能也是单引号和双引号没有区别的原因之一。

几乎每一个Python应用程序都会这样那样地用到字符串。字符串是一个文字或离散性质的数据类型，也就是说，解释器会把它们解释为一个独立的值，而不是容纳着其他Python对象的容器。字符串是不可变的，即对一个字符串中元素的修改会导致创建一个新的字符串。字符串是由一个一个的字符组成的，因此字符串中的每一个字符元素可以通过分离切片操作依次存取。

1. 如何创建并赋值一个字符串

字符串的创建工作很简单，就是给变量赋一个值而已，如下所示：

```
>>> aString = 'Hello World!'
>>> anotherString = "Python is cool!"
>>> print aString
Hello World!
>>> print anotherString
Python is cool!
>>> aBlankString = ''
>>> print aBlankString
''
```

2. 如何访问操作符中的值（字符或子字符串）

Python语言不支持字符类型；它们被看做是长度为1的字符串，因此也可以说是一个子字符串。访问一个子字符串要使用分离切片操作的方括号和分离该子字符串所必须的下标值或下标范围，如下所示：

```
>>> aString = 'Hello World!'
>>> aString[0]
'H'
>>> aString[1:5]
'ello'
>>> aString[6:]
'World!'
```

3. 如何对字符串内容进行修改

你可以通过给变量另外赋值一个字符串的办法“修改”一个现有的字符串。新字符串可以与旧字符串有一定联系，也可以是全新的另一个字符串：

```
>>> aString = aString[:6] + 'Python!'
>>> aString
'Hello Python!'
>>> aString = 'different string altogether'
>>> aString
'different string altogether'
```

和数字一样,字符串也是不可变的,因此不可能在没有创建一个新字符串的情况下对一个现有的字符串进行修改。这就意味着你不可能对一个字符串中的某个字符或者某个子字符串进行修改。但是,从刚才的例子里可以看出,用旧字符串加加减减地拼凑出一个新字符串也没有什么不妥。

4. 如何删除字符和字符串

我们再重复一遍,字符串是不可变的,因此不能从一个现有的字符串里单独删除字符。你只能把一个字符串重新赋值为一个空白字符串,或者在拼凑新字符串的时候丢弃不感兴趣的部分。

举例来说,假设你想在字符串'Hello World!'中删除一个字符——比如说一个小写的字母“l”吧,具体的操作情况如下所示:

```
>>> aString = 'Hello World!'
>>> aString = aString[:3] + aString[4:]
>>> aString
'Helo World!'
```

如果想清除或者删除一个字符串,可以给它赋值一个空白的字符串或者使用del语句:

```
>>> aString = ''
>>> aString
''
>>> del aString
```

在大多数应用程序中,通常并不需要明确地删除一个字符串。更常见的情况是定义该字符串的代码运行结束了,那个字符串也就被自动地回收了。

6.3 字符串和操作符

6.3.1 标准类型操作符

我们在第4章里已经介绍过一些能够对包括标准类型在内的大多数对象进行操作的操作符。我们现在来看看其中的一些是如何对字符串进行操作的。下面是几个使用字符串的简单示例:

```
>>> str1 = 'abc'
>>> str2 = 'lmn'
>>> str3 = 'xyz'
>>> str1 < str2
1
```

```
>>> str2 != str3
1
>>> (str1 < str3) and (str2 == 'xyz')
0
```

如果使用的是数值比较操作符，字符串之间比较的是它们的字符所对应的ASCII值。

6.3.2 序列操作符

1. 分离切片操作 ([] 和 [:])

我们在前面6.1.1节里已经介绍过怎样才能对一个序列中单个或成组的元素进行存取。在这一小节里，我们把那些知识运用到字符串上。准确地说，我们将讨论下面几个问题：

- 正向计数
- 反向计数
- 缺省下标

在下面的例子里，我们使用的字符串是'abcd'。右图中给出的是在一个字符串里各字符位置的正下标和负下标的标记方法。

0	1	2	3
a	b	c	d
-4	-3	-2	-1

我们可以用长度操作符确认该字符串的长度是4：

```
>>> string = 'abcd'
>>> len(string)
4
```

正向计数的时候，下标是从最左端的0开始的，字符串结尾处的下标是字符串长度值减1（因为是从0开始计数的）。在我们的例子里，字符串最后一个字符的下标值是：

```
最后下标值 = len(string) - 1
              = 4 - 1
              = 3
```

我们可以对这个下标范围内的任意子字符串进行访问。只带一个参数值的分离切片操作符取出的是单个的字符；而带下标范围参数（即使用了冒号(:)）的分离切片操作符取出的是连续多个字符。再讲一遍，下标范围[*start*:*end*]取出来的是从下标偏移量*start*开始到下标偏移量*end*（但不包括下标*end*处的那个字符）之间的全部字符。换句话说，对下标在[*start*:*end*]范围内的所有字符*x*来说，*start* ≤ *x* < *end*。

```
>>> string[0]
'a'
>>> string[1:3]
'bc'
>>> string[2:4]
'cd'
>>> string[4]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

任何超出有效下标范围（我们这个例子里是0到3）的下标都会导致一个错误。在上面的例子里，对string[2:4]的访问是合法的，因为它返回的是下标为2和3的字符，即‘c’和‘d’；但直接访问下标值4位置上的字符是非法的。

反向计数的时候，下标是从最右端的-1开始向字符串的头部递减的，最后一个下标是字符串长度的负值。在我们的例子里，字符串下标反向计数的最后一个值是：

```
最后下标值 = -len(string)
            = -4
```

```
>>> string[-1]
'd'
>>> string[-3:-1]
'bc'
>>> string[-4]
'a'
```

如果没有给出起始下标值或结束下标值，就分别默认为是字符串的开头或结尾，如下所示：

```
>>> string[2:]
'cd'
>>> string[1:]
'bcd'
>>> string[:-1]
'abc'
>>> string[:]
'abcd'
```

请注意，同时缺少起始和结束两个下标得到的是整个字符串的拷贝。

2. 是字符串的成员吗 (in、not in) ?

字符串的成员问题问的是一个字符（长度为1的字符串）是否出现在某个字符串中。如果该字符出现在字符串中，返回一个整数值1；否则返回整数值0。需要注意的是成员操作不用来确认某个子字符串是否是一个字符串的一部分，此功能是通过string方法，也就是string模块的函数find()和index()（以及它们的同类rfind()和rindex()）来实现的。

下面是一些使用字符串和成员操作符的例子：

```
>>> 'c' in 'abcd'
1
>>> 'n' in 'abcd'
0
>>> 'n' not in 'abcd'
1
```

在程序示例6-1中，我们将会用到string模块中的预定义字符串，如下所示：

```
>>> import string
>>> string.uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.lowercase
'abcdefghijklmnopqrstuvwxyz'
>>> string.letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.digits
'0123456789'
```

程序示例6-1是一个名为idcheck.py的小脚本程序，它的功能是检查一个字符串是否是一个合法的Python标识符。我们已经知道，Python标识符必须以一个字母字符打头，随后的字符可以是字母或数字。这个程序示例里还使用了将在本小节后面讲到的字符串合并操作符(+)。

这个脚本程序几次运行的输出结果如下所示：

```
% python idcheck.py
Welcome to the Identifier Checker v1.0
Testees must be at least 2 chars long.
Identifier to test? counter
okay as an identifier
%
% python idcheck.py
Welcome to the Identifier Checker v1.0
Testees must be at least 2 chars long.
Identifier to test? 3d_effects
invalid: first symbol must be alphabetic
```

让我们对这个应用程序做一个逐行的分析。

程序示例6-1 标识符合法性检查 (idcheck.py)

测试标识符的合法性。第一个字符必须是字母，后续字符必须是字母或数字。

这个测试程序只对那些长度最少为两个字符的标识符进行检查。

```
1  #!/usr/bin/env python
2
3  import string
4
5  alphas = string.letters + '_'
6  nums = string.digits
7
8  print 'Welcome to the Identifier Checker v1.0'
9  print 'Testees must be at least 2 chars long.'
10 inp = raw_input('Identifier to test? ')
11
12 if len(inp) > 1:
13
14     if inp[0] not in alphas:
15         print '''invalid: first symbol must be
16             alphabetic'''
17     else:
18         for otherChar in inp[1:]:
19
20             if otherChar not in alphas + nums:
21                 print '''invalid: remaining
22                     symbols must be alphanumeric'''
23                 break
24     else:
25         print "okay as an identifier"
```

3~6行

导入string模块，用预定义字符串生成我们将用做标识符合法性测试标准的字母和数字字符串。

8~12行

显示欢迎信息并提示用户输入。第12行的if语句过滤掉长度少于两个字符的字符串。

14~16行

检查第一个字符是否是一个字母。如果不是，显示有关信息，不再继续处理。

17~18行

如果第一个字符是一个字母，循环检查其余的字符。从第2个字符一直检查到字符串的结束。

20~23行

检查其余字符是否是字母或数字。请注意我们是如何使用合并操作符（马上就要讲到）来创建合法标识符的字符集的。只要发现有非法字符，就显示有关信息并用break退出循环，不再继续处理。

编程：程序执行性能**[CT]**

从程序的执行性能方面来看，在循环语句里像下面这样使用重复性操作或者用函数做为参数是不可取的。

```
while i < len(string):
    print 'character %d is: ', string[i]
```

上面的循环浪费宝贵的时间去重复计算字符串string的长度，因为这个函数是在每一次循环时都要执行的。我们可以先把这个值简单地保存起来，再像下面这样重写这个循环语句，结果就会有效率得多了：

```
length = len(string)
while i < length:
    print 'character %d is: ', string[i]
```

对程序示例6-1中的循环来说也是同样的道理。

```
for otherChar in input[1:] :
    if otherChar not in alphas + nums :
        :
```

从第19行开始的for循环包含着一个用来合并两个字符串的if语句。这两个字符串在整个应用程序里是不会发生变化的，但这个合并操作却要在每次循环时执行一次。如果我们先把这个新字符串保存起来，就可以引用这个字符串而不是反复执行同样的字符串合并操作了，如下所示：

```
alphanums = alphas + nums
for otherChar in input[1:] :
    if otherChar not in alphanums :
        :
```

24~25行

向大家解释for-else循环语句好像没有什么必要，但我们还是啰嗦一下吧（详细介绍请参考第8章内容）。for循环有一个可选的else语句，如果加上它，就会在循环部分正常结束之后（即没有用break跳出循环）执行它。在我们的应用程序里，如果对后续字符的检查全部通过，我们就会得到一个合法的标识符名字。其结果是显示一条有关信息，完成整个执行过程。

这个程序并不是没有缺陷的。其中的一个问题是检查标识符的长度必须大于1，也就是说，我们的应用程序好像并没有反映出Python语言对标识符的真正定义，因为它的长度也可以是一个字符。我们应用程序的另外一个问题是它并没有考虑到Python语言中的关键字，那都是一些不能被用做标识符的保留名字。我们把这些留给读者做为练习（请参考练习6-2）。

3. 合并操作 (+)

我们可以使用合并操作符用现有的字符串创建新的字符串。我们已经在程序示例6-1里看到合并操作符起到了怎样的作用。下面是另外一些例子：

```
>>> 'Spanish' + 'Inquisition'
'SpanishInquisition'
>>>
>>> 'Spanish' + ' ' + 'Inquisition'
'Spanish Inquisition'
>>>
>>> s = 'Spanish' + ' ' + 'Inquisition' + ' Made Easy'
>>> s
'Spanish Inquisition Made Easy'
>>>
>>> import string
>>> string.upper(s[:3] + s[20])
'SPAM'
```

最后一个例子演示了使用合并操作符把来自两个字符串的切片（‘Spanish’中的‘Spa’和‘Made’中的‘M’）合并在一起的情况。从两个字符串中取出来的两个切片合并在一起之后再被送到string.upper()函数去转换为一个全部是大写字符的新字符串。

4. 重复操作 (*)

重复操作符用来创建新的字符串，它的功能是把同一个字符串拷贝多次之后合并为一个新字符串，如下所示：

```
>>> 'Ni!' * 3
'Ni!Ni!Ni!'
>>>
>>> '*' * 40
'*****'
>>>
>>> print '-' * 20, 'Hello World!', '-' * 20
----- Hello World! -----
>>> who = 'knights'
>>> who * 2
'knightsknights'
>>> who
'knights'
```

与其他标准操作符一样，它不改变原始变量的内容，这一点可以从上面的最后一个例子里看出来。

6.4 只作用于字符串的操作符

6.4.1 格式操作符 (%)

Python语言最酷的一个特色是它的字符串格式操作符。这个操作符只能对字符串进行操作，与C语言中printf()语句中的格式字符有异曲同工之处。事实上，连它们使用的字符都是一样的，都是百分号(%)，并且它还支持所有的printf()格式代码。

格式操作符的使用语法如下所示:

```
format_string % (arguments_to_convert)
```

左边的format_string是你通常会在printf()语句里找到的第一个参数,也就是嵌入有%字符的格式字符串。在表6-4里列出了可以合法使用的转换代码。arguments_to_convert参数对应于你会在printf()语句里送出的其余参数,也就是准备转换并显示的变量们。

表6-4 格式操作符转换符号

格式符号	转 换
%c	字符
%s	在格式转换之前先用str()函数进行字符串转换
%i	带符号的十进制整数
%d	带符号的十进制整数
%u	不带符号的十进制整数
%o	八进制整数
%x	十六进制整数(小写字母)
%X	十六进制整数(大写字母)
%e	乘方记号(带小写的“e”字母)
%E	乘方记号(带大写的“E”字母)
%f	浮点实数
%g	%f和%e中比较短的格式
%G	%f和%E中比较短的格式

Python语言支持两种格式的输入参数。第一种是一个表列(简介见2.8节,正式讨论见6.15节),它基本上就是准备进行转换的参数集合,就像C语言中的printf()语句那样。Python语言支持的第二种格式是一个字典(请参考第7章内容)。一个字典基本上可以说是一个哈希化的“关键字-对应值”对的集合,其关键字出现在format_string部分中,而与之对应的值则用在对字符串进行格式转换的时候。

经过转换的字符串既可以用在print语句里组成输出显示给用户,也可以保存到一个新的字符串中供下一步处理或显示到一个图形化用户操作界面去。

Python语言支持的其他符号及其功能列在表6-5里。

表6-5 格式操作符的辅助命令

符 号	功 能
*	定义宽度或数值精度的参数
-	左对齐
+	显示数字的正负符号
<sp>	在正数前面留一个空格的位置
#	在数字最前面加上一个代表八进制数的零符号(“0”)或代表十六进制数的“0x”或“0X”,具体要看使用的是“x”还是“X”
0	在数字前加上一些对齐用的零(代替空格)
%	“%%”表示实际输出一个百分号(%)
(var)	映射变量(字典参数)
m.n.	m是总宽度的最小值,n是小数点后面的数字个数(如果有的话)

与C语言中printf ()语句中一样,星号(*)再加上参数表列中的一个值就能动态地定义输出数据的宽度和精度。在开始举例之前给大家提个醒:长整数可能会因为太大而无法转换为标准的整数,因此我们建议使用科学计数法来表示它们。

下面是一些使用了格式操作符的例子。

1. 十六进制输出

```
>>> "%x" % 108
'6c'
>>>
>>> "%X" % 108
'6C'
>>>
>>> "%#X" % 108
'0X6C'
>>>
>>> "%#x" % 108
'0x6c'
```

2. 浮点和乘方记号的输出

```
>>>
>>> '%f' % 1234.567890
'1234.567890'
>>>
>>> '%.2f' % 1234.567890
'1234.57'
>>>
>>> '%E' % 1234.567890
'1.234568E+03'
>>>
>>> '%e' % 1234.567890
'1.234568e+03'
>>>
>>> '%g' % 1234.567890
'1234.57'
>>>
>>> '%G' % 1234.567890
'1234.57'
>>>
>>> "%e" % (111111111111111111111111111111L)
'1.111111e+21'
```

3. 整数和字符串输出

```
>>> "%d" % 4
'+4'
>>>
>>> "%d" % -4
' -4'
>>>
>>> "we are at %d%%" % 100
'we are at 100%'
>>>
>>> 'Your host is: %s' % 'earth'
'Your host is: earth'
>>>
```

```
>>> 'Host: %s\tPort: %d' % ('mars', 80)
'Host: marsPort: 80'
>>>
>>> num = 123
>>> 'dec: %d/oct: %#o/hex: %#X' % (num, num, num)
'dec: 123/oct: 0173/hex: 0X7B'
>>>
>>> "MM/DD/YY = %02d/%02d/%d" % (2, 15, 67)
'MM/DD/YY = 02/15/67'
>>>
>>> w, p = 'web', 'page'
>>> 'http://xxx.yyy.zzz/%s/%s.html' % (w, p)
'http://xxx.yyy.zzz/web/page.html'
```

刚才的例子在转换时使用的都是表列类型的参数。下面的例子在格式操作符上使用了一个字典类型的参数：

```
>>> 'There are %(howmany)d %(lang)s Quotation Symbols' % \
...     {'lang': 'Python', 'howmany': 3}
'There are 3 Python Quotation Symbols'
```

4. 了不起的调试纠错工具

格式操作符不仅是一种很酷的既好用又被人们所熟悉的功能特色，还是一种了不起的调试纠错工具。可以说，任何Python对象都有一个字符串表示形式（可以通过repr()函数或‘’操作符取值，或者通过str()函数打印）。printf()语句遇到一个对象时会自动调用str()函数。这很有用。你在定义自己的对象时，可以给它创建一个字符串表示形式，这样就可以让repr()和str()函数（以及‘’操作符和printf()语句）返回一个适当的值做为输出。如果事情越弄越糟，repr()和str()函数都无法显示一个对象，Python语言至少还留了一手，它会在缺省的情况下以下面的格式多少显示一些东西：

```
< . . . something that is useful . . . >
```

6.4.2 生字符串操作符 (r/R)

[1.5]

从Python 1.5版本开始引入的生字符串（raw string）概念是相对于字符串中特殊的转义字符而言的（请参考下一节关于部分这类字符的论述）。生字符串里的全体字符都是做为最原始的字符来对待的，不会转换为特殊的或非打印字符。

这一特性使生字符串在需要用到它们的时候非常方便，比如说在组成规则表达式的时候（请参考re模块的文档）。规则表达式（regular expression，简称RE）是一些用来定义高级的字符串检索匹配模板的字符串，通常会包含一些特殊符号来指定字符、字符组合与匹配信息、变量名以及字符类别等。RE的语法中已经有足够多的符号了，但如果需要按照特殊字符所代表的符号本身含义进行操作时，你就不得不再插入一些额外的符号才能得到目的，这时候“字符数字符号”可就混成一锅粥了！生字符串可以在这种情况下帮到你，它在组成RE模板时可以完全不使用正常符号。

除了在定义字符串的引号前要加上一个生字符串操作符字母“r”以外，生字符串的语法和正常字符串的是完全一样的。我们既可以使用小写字母（r），也可以使用大写字母（R），并且它必须被紧贴在第一个引号的前面。如下所示：

```

>>> print r'\n'
\n
>>>
>>> print '\n'
\n
>>> print r'werbac'
werbac
>>>
>>> print r'webbac\n'
webbac\n
>>>
>>> print r'fglkjfg\[123=091'
fglkjfg\[123=091
>>>
>>> import re
>>> aFloatRE = re.compile(R'([+-]?\d+(\.\d*)?([eE][+-]
j?\d+)?)')
>>> match = aFloatRE.search('abcde')
>>> print "our RE matched:", match.group(1)
''
>>> match = aFloatRE.search('-1.23e+45')
>>> print 'our RE matched:', match.group(1)
'-1.23e+45'

```

6.4.3 Unicode字符串操作符 (u/U)

[1.6]

Unicode（统一字符集）字符串操作符大写字母（U）和小写字母（u）是从Python 1.6版本开始随Unicode支持引入的概念，它以标准字符串或者带Unicode字符的字符串为操作数，把它们转换为一个完全的Unicode字符串对象。关于Unicode更多的细节请参考6.7.4节。此外，新的字符串方法（请参考6.6节）和新的规则表达式引擎都具备Unicode支持。下面是一些例子：

```

u'abc'          U+0061 U+0062 U+0063
u'\u1234'       U+1234
u'abc\u1234\n'  U+0061 U+0062 U+0063 U+1234 U+0012

```

Unicode操作符也接受“生Unicode字符串”（raw Unicode string），但前提是必须与前一节中介绍的生字符串操作符联合使用。Unicode操作符必须放在生字符串操作符的前面。

```
ur'Hello\n World !
```

6.5 内建函数

6.5.1 标准类型函数

cmp()

类似于数值比较操作符，cmp()内建函数对字符串进行的也是字符的ASCII值比较操作，举例如下：

```

>>> str1 = 'abc'
>>> str2 = 'lmn'
>>> str3 = 'xyz'

```

```
>>> cmp(str1, str2)
-11
>>> cmp(str3, str1)
23
>>> cmp(str2, 'lmn')
0
```

6.5.2 序列类型函数

1. len()

```
>>> str1 = 'abc'
>>> len(str1)
3
>>> len('Hello World!')
12
```

len()内建函数返回的是字符串中的字符个数。

2. max()和min()

```
>>> str2 = 'lmn'
>>> str3 = 'xyz'
>>> max(str2)
'n'
>>> min(str3)
'x'
```

虽然更适合用于其他的序列类型，但max()和min()内建函数确实也能返回字符串中ASCII值最大或最小的字符。

6.5.3 字符串类型函数

内建的raw_input()函数向用户显示一个给定的提示性字符串，再接受并返回一个用户输入的字符串。下面是使用raw_input()内建函数的一个例子：

```
>>> user_input = raw_input("Enter your name: ")
Enter your name: John Doe
>>>
>>> user_input
'John Doe'
>>>
>>> len(user_input)
8
```

我们在前面曾经指出Python语言中的字符串在其结尾不像C语言中的字符串那样有一个终结的NUL字符。我们在上面的例子里加上了一个额外的len()调用让读者明白确实是所见即所得。

6.6 字符串的内建方法

[1.6/2.0]

字符串方法是最近才加到Python语言中的，它从Python 1.6（以及JPython 1.1）版本开始引入，并在2.0版本里进一步细化。引入这些方法的目的是为了代替string模块中的功能函数，同时也为字符串添加更多的功能。表6-6列出了目前适用于字符串的全部方法。全体字符串方法都完

全支持Unicode字符串，其中有些还只适用于Unicode字符串。

表6-6 字符串类型的内建方法

方 法	说 明
<code>string.capitalize()</code>	把字符串string的第一个字母变成大写
<code>string.center(width)</code>	返回一个用空格填充的字符串，原来的string字符串居中，两头用空格填充至width宽度
<code>string.count(str, beg=0, end=len(string))</code>	统计子字符串str在字符串string里出现了多少次；如果给出了起始下标beg和结束下标end，就统计该子字符串里str出现过的次数
<code>string.encode(encoding='UTF8', errors='strict')</code>	返回一个对原字符串编码后得到的字符串。如果出现错误并且事先没有指定'ignore'或'replace'，那么缺省的处理是引发一个ValueError
<code>string.endswith(str, beg=0, end=len(string))</code>	检查字符串string或string的一个子字符串（如果给出了起始下标beg和结束下标end的话）是否以str结尾。“是”返回1；“否”返回0
<code>string.expandtabs(tabsize=8)</code>	把字符串string中的制表符扩展为多个空格。如果没有指定tabsize参数，缺省的情况是一个制表符等于8个空格
<code>string.find(str, beg=0, end=len(string))</code>	检查字符串string中是否有str，如果给出了起始下标beg和结束下标end的话，就检查该子字符串。如果找到返回1；没有找到返回-1
<code>string.index(str, beg=0, end=len(string))</code>	同find()，但在没有找到的情况下会返回一个例外
<code>string.isalnum()</code>	如果字符串string中至少有一个字符并且全体字符都是字母和数字就返回1；否则返回0
<code>string.isalpha()</code>	如果字符串string中至少有一个字符并且全体字符都是字母就返回1；否则返回0
<code>string.isdecimal()</code>	如果字符串string中都是十进制数字就返回1；否则返回0
<code>string.isdigit()</code>	如果字符串string中都是数字就返回1；否则返回0
<code>string.islower()</code>	如果字符串string中至少有一个分大小写的字符并且所有分大小写的字符都是小写就返回1；否则返回0
<code>string.isnumeric()</code>	如果字符串string中都是数值字符就返回1；否则返回0
<code>string.isspace()</code>	如果字符串中都是空格就返回1；否则返回0
<code>string.istitle()</code>	如果字符串中都是“句首单词”形式（请参考title()方法）就返回1；否则返回0
<code>string.isupper()</code>	如果字符串string中至少有一个分大小写的字符并且所有分大小写的字符都是大写就返回1；否则返回0
<code>string.join(seq)</code>	把序列seq中元素的字符串表示形式合并为一个字符串，彼此之间用string隔开
<code>string.ljust(width)</code>	返回一个用空格填充的字符串，原来的string字符串居左，右边用空格填充至width宽度
<code>string.lower()</code>	把字符串string中的大写字符都改为小写
<code>string.lstrip()</code>	清除字符串string中所有的前导空格
<code>string.replace(str1, str2, num=string.count(str1))</code>	把字符串string中所有出现str1的地方替换为str2。如果给出了num值，就最多替换num次
<code>string.rfind(str, beg=0, end=len(string))</code>	同find()，但在字符串string中反方向查找
<code>string.rindex(str, beg=0, end=len(string))</code>	同index()，但在字符串string中反方向查找
<code>string.rjust(width)</code>	返回一个用空格填充的字符串，原来的string字符串居右，左边用空格填充至width宽度

(续)

方 法	说 明
<code>string.rstrip()</code>	清除字符串string中所有的尾续空格
<code>string.split(str="")</code> <code>num=string.count(str)</code>	以str为分隔符(如果没有指定str就使用空格做为分隔字符)拆分字符串string, 返回的是子字符串列表。如果给出了num值, 就最多拆分为num个子字符串
<code>string.splitlines()</code> <code>num=string.count('\n')</code>	按换行符拆分字符串string, 返回的是去掉换行符的文本行列表
<code>string.startswith(str, beg=0, end=len(string))</code>	检查字符串string或string的一个子字符串(如果给出了起始下标beg和结束下标end的话)是否以str开始。“是”返回1; “否”返回0
<code>string.strip([obj])</code>	对字符串string同时完成lstrip()和rstrip()操作
<code>string.swapcase()</code>	把字符串string中的大写字符换为对应的小写字符, 小写字符换为大写字符
<code>string.title()</code>	对字符串string进行“句首单词”处理, 也就是说, 单词的第一个字符是大写字母, 其余的都是小写字母(请参考istitle()方法)
<code>string.translate(str, del="")</code>	根据转换表str(256个字符)对字符串string进行转换, 清除那些在del字符串中的字符
<code>string.upper()</code>	把字符串string中的小写字符都改为大写
<code>string.zfill(width)</code>	在原字符串string的左边加零(0)直到宽度达到width个字符。如果是对数字进行操作的话, zfill()会保留正负符号(少一个“0”字符)

① 在1.6版本里只能处理Unicode字符串, 但在2.0版本里能处理所有字符串。

② 1.5.2版本里的string模块没有此功能的函数。

③ JPython 1.1版本里还不是一个字符串方法。

④ 只能处理Unicode字符串。

下面我们用JPython给大家几个使用字符串方法的例子:

```
>>> quest = 'what is your favorite color?'
>>> quest.capitalize()
'What is your favorite color?'
>>>
>>> quest.center(40)
'      what is your favorite color?'
>>>
>>> quest.count('or')
2
>>>
>>> quest.endswith('blue')
0
>>>
>>> quest.endswith('color?')
1
>>>
>>> quest.find('or', 30)
-1
>>>
>>> quest.find('or', 22)
25
>>
>>> quest.index('or', 10)
16
>>>
```

```
>>> ':'.join(quest.split())
'what:is:your:favorite:color?'
>>> quest.replace('favorite color', 'quest')
>>>
'what is your quest?'
>>>
>>> quest.upper()
'WHAT IS YOUR FAVORITE COLOR?'
```

上面的例子中最复杂的是那个带split()和join()两个函数的语句。我们先调用split()处理我们的字符串，因为没带参数，所以它以空格字符为分隔符号对字符串进行了分断。接着，分断得到的单词表被送到join()重新拼合，但这次使用了一个新的分隔符冒号。请注意，split()方法是对字符串进行操作的，对于单字符字符串‘:’使用的是join()方法。

6.7 字符串的特性

6.7.1 特殊或控制字符

Python语言与大多数其他的高级程序设计语言或脚本程序设计语言一样，都用一个反斜杠字符和另外一个字符组成的字符来表示一个特殊的字符，它通常是一个非打印字符。在程序或者命令执行时，这个字符对会被替代为那个特殊字符。我们在前面曾经介绍过，如果在包含这些特殊字符的字符串前面加上一个生字符操作符，它们就不会被解释为特殊字符了。

除那些常见的特殊字符如换行符(\n)和(水平)制表符TAB(\t)以外，还可以通过其ASCII值的\OOO和\xXX形式使用特定的字符，其中的OOO和XX是与之对应的八进制或十六进制ASCII值。下面是数字0、65、255的十、八、十六进制表示方法：

	ASCII	ASCII	ASCII
十进制	0	65	255
八进制	\000	\101	\177
十六进制	\x00	\x41	\xFF

所有这些特殊的字符，包括那些用反斜杠字符转义的，都可以像正常的字符那样被保存在Python字符串里。

Python语言中的字符串和C语言中的字符串还有一个不一样的地方是：Python字符串的尾部不是以NUL(\000)字符（ASCII值为0的字符）结束的。NUL字符和其他需要用反斜杠字符转义的特殊字符没什么区别。事实上，NUL字符不仅可以出现在Python字符串里，还可以在一个字符串里有任意多个；也就是说，它可以出现在字符串中的任何位置上。它与其他控制字符相比并不特殊。表6-7列出的是Python语言大多数都支持的转义字符。

表6-7 字符串文字反斜杠转义字符

/X	八进制	十进制	十六进制	字符	说明
\0	000	0	0x00	NUL	空字符Null
\a	007	7	0x07	BEL	铃音
\b	010	8	0x08	BS	退格

(续)

/X	八进制	十进制	十六进制	字符	说明
\t	011	9	0x09	HT	水平制表
\n	012	10	0x0A	LF	换行
\v	013	11	0x0B	VT	垂直制表
\f	014	12	0x0C	FF	进纸
\r	015	13	0x0D	CR	回车
\e	033	27	0x1B	ESC	转义
\"	042	34	0x22	"	双引号
\'	047	39	0x27	'	单引号
\\	134	92	0x5C	\	反斜杠

正像刚才提到的, ASCII值可以是八进制或十六进制的; 在一行的结尾处对换行字符进行转义就可以把语句续到下一行。合法的ASCII字符值都在0到255 (八进制的0177, 十六进制的0XFF) 之间。

\OOO 八进制值OOO (范围是0000到0177)
 \xXX 'x' 加上十六进制值XX (范围是0X00到0XFF)
 \ 转义换行符, 让语句延续到下一行

控制字符在字符串中的作用之一是作为分隔符号。数据库或因特网/Web的处理过程会把大多数可打印字符做为数据项来对待, 这就意味着用它们做分隔符是不好的。

区分某个字符是一个分隔符还是一个数据项是困难的。同时, 使用一个像冒号(:) 这样的可打印字符做分隔符又会对允许用在数据中的字符个数产生影响, 而这可不是我们想要的结果。

最常见的解决办法是把一个很少使用的非打印ASCII值用做分隔符。这样解决最好不过, 冒号或其他可打印字符可以用在更需要的地方。

6.7.2 三引号

字符串可以被放在单引号或者双引号中间, 但这样还是存在不足: 对包含有特殊或非打印字符, 特别是换行符的字符串进行处理通常是比较困难的。Python语言中的三引号可以解决这类的问题, 它允许字符串延伸到多个文本行, 字符串里可以包含换行符、制表符、或者其他特殊字符。

三引号的语法由三个连续的单引号或双引号构成 (当然要配对使用), 如下所示:

```
>>> para_str = """this is a long string that is made up of
... several lines and non-printable characters such as
... TAB ( \t ) and they will show up that way when displayed.
... NEWLINES within the string, whether explicitly given like
... this within the brackets [ \n ], or just a NEWLINE within
... the variable assignment will also show up.
... """
```

三引号使程序员摆脱了引号和转义字符的纠缠, 同时也使程序中的小段文字最大可能地接近WYSIWYG (what you see is what you get, 所见即所得) 格式。

下面的一个例子向大家演示了使用print语句打印这个字符串内容时会发生什么样的情况。请注意每一个特殊字符都是如何被转换为它的打印形式的，特别是字符串尾部“up.”和三括号结束之间的最后一个换行符。还要注意的是换行动作既可以明确地发生在每一行结尾（即我们按下了回车键），也可以发生在出现其转义代码（\n）的地方。如下所示：

```
>>> print para_str
this is a long string that is made up of
several lines and non-printable characters such as
TAB (    ) and they will show up that way when displayed.
NEWLINES within the string, whether explicitly given like
this within the brackets [
    ], or just a NEWLINE within
the variable assignment will also show up.
```

我们早些时候介绍过序列类型的len()内建函数，如果把它用在字符串上，会给出一个字符串里的字符个数。

```
>>> len(para_str)
307
```

在对我们的字符串使用这个函数的时候，得到的结果是307，它包括换行符和水平制表符。在交互式解释器里查看字符串的另外一个办法是直接准备查看的对象的名字提供给解释器。这样，我们就将看到该字符串的“内部”表示形式。如果你还没有弄清楚我们刚才提到的最后一个换行符（在最后一个单词“up.”的后面，三括号结束的前面）的概念，请看看下面给出的字符串内部表示形式，你会发现字符串的最后一个字符就是我们刚才提到的换行符。

```
>>> para_str
'this is a long string that is made up of\\012several lines
and non-printable characters such as\\012TAB ( \\011 ) and
they will show up that way when displayed.\\012NEWLINES
within the string, whether explicitly given like\\012this
within the brackets [ \\012 ], or just a NEWLINE
within\\012the variable assignment will also show up.\\012'
```

6.7.3 字符串的不可变性

我们在4.7.2节里介绍过，字符串是不可变的数据类型，也就是说，它们的值是不能被改动的。这就意味着如果你想对一个字符串的内容进行修改，可采取的办法只能是取出它的一个子字符串、把另一个字符串合并到该字符串尾部或者把该字符串合并到另一个字符串尾部，等等；总之，必须为它创建一个新的字符串。

这个问题说起来好像很复杂，但实际上却很简单。因为内存是由Python来管理的，所以你根本不会注意到这些情况的发生。无论何时，只要改动了字符串或者执行了与不可变性对立的操作，Python会自动分配一个新的字符串。在下面的例子里，Python为字符串'abc'和'def'分配了空间。但当执行合并操作产生字符串'abcdef'时，Python会为新字符串分配一个新的空间。

```
>>> abc' + def'
'abcdef'
```

对变量进行赋值操作也不例外：

```
>>> string = 'abc'
>>> string = string + 'def'
>>> string
'abcdef'
```

上面的例子可以看做是先把字符串'abc'赋值给string变量，再把字符串'def'添加到变量string的末尾。表面上看，字符串好像是可变的；但你看不到的实际情况是：当操作“s + 'def'”被执行的时候，一个新的字符串被创建出来了；接着，这个新对象被重新赋值给s。分配给字符串'abc'的空间被收回。

用id()内建函数可以帮助我们了解实际发生了什么事情。id()函数的功能是返回一个对象的“标识”。这个值是在Python语言里所能获得的最接近于“内存地址”的东西了。

```
>> string = 'abc'
>>>
>>> id(string)
135060856
>>>
>>> string = string + 'def'
>>> id(string)
135057968
```

请注意，字符串的标识在操作执行的前后是不同的。另外一个测试不可变性的实验是看看能不能对字符串中的单个字符或者它的子字符串进行修改。我们现在让大家看看对单个字符或字符串切片的修改是不允许的。如下所示：

```
>>> string
'abcdef'
>>>
>>> string[2] = 'C'
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: __setitem__
>>>
>>> string[3:6] = 'DEF'
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: __setslice__
```

两个操作都会导致出现错误。如果真的想完成这些操作，就必须通过现有字符串的子字符串来创建新的字符串，然后再把新字符串赋值给string变量。

```
>>> string
'abcdef'
>>>
>>> string = string[0:2] + 'C' + string[3:]
>>> string
'abCdef'
>>>
>>> string[0:3] + 'DEF'
'abCDEF'
>>>
>>> string = string[0:3] + 'DEF'
>>> string
'abCDEF'
```

我们发现：对字符串这样的不可变对象来说，能够合法地出现在赋值语句左边的表达式只能是代表整个对象的（比如一整个字符串）变量形式，不能是单个的字符或子字符串；对右边的表达式就没有这样的限制。

6.7.4 Unicode支持

[1.6]

对Unicode字符串的支持是从Python 1.6版本开始引入的，它用来在多种双字节字符编码格式之间进行转换，也包括对这些字符串进行处理的许多功能。有了这些额外提供的字符串方法（请参考6.6节），Python字符串的功能就完整了，能够对大量带有Unicode字符串存储、访问和操作的程序进行处理。在本书编写的时候，具体的Python语言方案还没有最终确定下来。我们在此将尽最大努力把Python语言对Unicode 3.0的支持比较全面地向大家做一个介绍。

1. unicode()内建函数

unicode()内建函数的操作方式类似于Unicode字符串操作符（u/U）。它以一个字符串为参数，返回的是一个Unicode字符串。

2. encode()内建方法

[2.0]

encode()内建方法以一个字符串为参数，返回的是一个经过编码的字符串。在Python 2.0版本中，encode()是一个能够对普通字符串和Unicode字符串同时进行操作的方法；1.6版本中，就只能对Unicode字符串进行操作。

3. Unicode类型

如果把一个Unicode字符串做为type()函数的一个参数，比如type(u'')，那么返回的将是一个名为unicode的新Unicode类型。

4. Unicode顺序

标准的ord()内建函数还按原来的方式执行。它最近经过改进，能够支持Unicode对象了。新的unichr()内建函数会返回字符（如果它是一个32位的值）的Unicode对象；否则将引发一个ValueError例外。

5. 类型协调

混模式字符串操作符要求把标准的字符串转换为Unicode对象。

6. 例外

UnicodeError在例外模块里被定义为ValueError的子类。全体与Unicode编码/解码（见表6-8）有关的例外都是UnicodeError的子类。请参考字符串的encode()方法。

表6-8 Unicode编码方案

编码方案	说 明
utf-8	8位可变长编码（默认的编码方案）
utf-16	16位可变长编码（小/大集合）
utf-16-le	utf-16编码方案，但明确地指定使用小集合
utf-16-be	utf-16编码方案，但明确地指定使用大集合
ascii	7位ASCII编码表
iso-8859-1	ISO 8859-1（拉丁1）编码表

(续)

编码方案	说 明
unicode-escape	(请参考Python语言中Unicode构造器中的定义)
raw-unicode-escape	(请参考Python语言中Unicode构造器中的定义)
native	Python语言内部使用的字符串格式

7. 可处理Unicode的规则表达式引擎

新的规则表达式引擎必须是能够处理Unicode字符串的。请参考下一节(6.8节)中的re代码模块注释部分。

8. 字符串格式操作符

至于Python语言中的格式字符串, '%s' 对Python字符串中嵌入的Unicode对象执行的是str(u)操作, 因此其输出将是u.encode(<default encoding>)。如果格式字符串本身就是一个Unicode对象, 那么全体参数都将先被协调转换为Unicode, 再组合起来按照格式字符串进行格式编排。数字先被转换为字符串, 再被转换为Unicode。Python字符串将使用<default encoding> (默认的编码方案) 解释为Unicode字符串。Unicode对象就不用再做什么转换了。其他字符串格式操作也都依此办理。下面是一个例子:

```
u'%s %s' % (u'abc', 'abc') ⇒ u'abc abc'
```

与Python语言对Unicode字符串的支持等方面有关的资料可以在软件发行版本中的Misc/unicode.txt中找到。这份文档的最新版本可以随时在下面这个网址查到:

<http://starship.python.net/~lennburg/unicode-proposal.txt>

要想获得关于Python语言中Unicode字符串更多的帮助和资料, 请在下面的网址查阅Python Unicode教程:

http://www.reportlab.com/118n/python_unicode_tutorial.html

6.7.5 Python语言没有字符或数组

我们在前面的章节里已经提到过, Python语言是不支持字符类型的。我们还可以说C语言也没有明确地支持字符串类型; 相反, C语言中的字符串只不过是单个字符组成的数组罢了。第三个事实是Python语言里没有把“数组”做为一个基本类型(如果非用不可, 倒是有一个array模块)。因为字符串本身就体现着顺序访问的能力, 所以把字符串实现为字符数组并没有多大的必要。

在单个字符和字符串之间, Python聪明地选择了字符串做为一个数据类型。把较大的一个东西整个做为一个“单位”来处理要简单得多, 因为大多数应用程序都是把字符串做为一个整体而不是单个的字符来操作处理的。应用程序可以把字符串转换为整数、让用户输入字符串、用规则表达式检索匹配子字符串、在文件里查找指定的字符串、甚至是对某些字符串(比如名字等)进行排序。对单个字符的处理和检索操作(比如搜索和替换、搜索分隔符等)又有多少呢? 对大多数应用程序而言, 应该说确实用得很少。

但Python操作员也不能因此而失去这些功能操作。搜索与替换可以用规则表达式和re模块来完成; 对字符串基于分隔符的搜索和断字操作可以用split()函数来完成; 对子字符串的搜索可以

通过find()和rfind()函数来实现；只有旧概念下的普通字符的成员性还需要用in和not in序列操作符来验证。

我们再来看看chr()和ord()内建函数，它们用来在ASCII整数及其对应的字符之间进行转换。在Python语言里，字符不像是在C语言里那样是一个整数类型，所以C语言有个“特色功能”在Python语言里消失了。

这个所谓“消失”了的C语言特色功能是：Python语言里不再允许直接对字符进行数值运算，比如'A'+3等等。在C语言里这是允许的，因为做为char类型的'A'和做为int类型的3都是整数（分别是一个1字节整数和2或4字节整数）；在Python语言里，因为'A'是一个字符串，而3是一个普通整数，数值类型和字符串类型之间又没有C语言那样的加法(+)操作关系，所以这会是一个类型不匹配情况。如下所示：

```
>>> 'B'
'B'
>>> 'B' + 1
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation
>>>
>>> chr('B')
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation
>>>
>>> ord('B')
66
>>> ord('B') + 1
67
>>> chr(67)
'C'
>>> chr(ord('B') + 1)
'C'
```

当我们试图把'B'的ASCII值加上1以通过加法得到'C'时，操作失败了。Python语言不使用这种一字节整数的算术，它的解决方案要用到chr()和ord()内建函数。

6.8 相关模块

表6-9列出字符串主要的相关模块，它们是Python标准库的组成部分。

表6-9 字符串类型的相关模块 (u: 只用于Unix)

模 块	内 容
string	字符串处理和实用性函数
re	规则表达式：强大的字符串模式匹配功能
struct	在字符串和二进制数据格式之间来回转换
c/StringIO	字符串缓冲区对象，其行为类似于一个文件
crypt	实现单向加密编码功能u
rotor	提供多平台的编码/解码服务

模块: string

[CM][1.6]

这里有许多与字符串打交道的实用性和操作性函数。如果读者曾经使用过其他高级程序设计语言如C、C++或Java,就会在这里遇到许多熟面孔。Python语言已经尽了最大的努力把最常用的功能集成为它的操作符和内建函数。但把一切都集成到一个语言里是不可能的。这就是为什么有string模块的原因。string模块提供了一整套常数和模块函数,对字符串提供了附加的支持。

string模块里的一些关键性函数包括: `atoi*()`——共有三个函数,把字符串转换为三种不同的数值类型; `split()`——把一个字符串分断为几个字符串; `join()`——作用正好与`split()`相反:把几个字符串合并为一个字符串; `find()`——搜索子字符串。

详细资料和字符串模块属性的使用方法请参考string模块的文格。从Python的1.6版本开始,许多string模块中的函数已经被实现为字符串方法,字符串的这个新特色看来会逐步淘汰掉这个模块。我们在6.6节向大家介绍过这些方法。

模块: re

[CM][1.5/1.6]

规则表达式(regular expression,简称RE)为字符串提供了先进的模板匹配方案。定义这些匹配模板用的是另外一种语法,当搜索操作从头到尾穿过文本的时候,读者可以有效地把它们用做“过滤器”。这些过滤器可以取出匹配的文本模板,完成搜索与替换操作,或者根据给出的模式对字符串进行分断。

从Python 1.5版本开始引入的re模块淘汰了以前版本中的regex和regsub模块。它在Python语言对规则表达式的支持方面有一个重大的升级,就是完全采纳了Perl语言的规则表达式语法。为了改善性能和增加对Unicode字符串的支持,Python 1.6中的规则表达式引擎被重新编写了。

re模块里的一些关键性函数包括: `compile()`——把一个规则表达式编译为一个可以反复使用的RE对象; `match()`——从一个字符串的开始查找某个匹配模板; `search()`——在字符串里查找有无匹配模板内容出现; `sub()`——完成的是查找与替换操作。有的函数返回的是匹配对象(match object),然后再通过这个对象去访问成组保存着的匹配(如果找到了的话)情况。第15章将全部用来讨论规则表达式。

6.9 字符串总结

1. 由括在引号内的字符组成

做为Python语言中的一种数据类型,字符串可以被想象为两个Python引用符号即引号之间的一个字符数组或一个连续的字符集合。Python语言中最常用的两种引用符号是单引号(')和双引号(")。实际的字符串由两个引号之间但不包括那两个引号在内的全部字符构成。

有两种引号可供选用很有好处:我们可以在把一种引号用做“字符串括号”,把另外一种引号用做字符串里的字符,这种做法不必使用特殊的转义字符。单引号之间的字符串里可以有有用做字符的双引号;反过来也行。如下所示:

```
>>> quote1 = 'George said, "Good day Madam. How are we today?"'
>>> print quote1
George said, "Good day Madam. How are we today?"
>>> quote2 = "Martha replies, 'We are fine, thank you.'"
>>> print quote2
Martha replies, 'We are fine, thank you.'
```

2. Python语言不另外支持字符类型

字符串是Python语言中唯一的文字序列类型，或者说是一个字符序列。但字符本身并不是一个类型，因此字符串就是对字符进行存储和处理的最底层的基本数据类型了。大多数应用程序倾向于把字符串看做是一个整体或一个单独的实体。因此，Python语言以操作符、内建函数以及字符串模块内容等形式准备了大量的字符串处理工具。但Python语言是灵活的，在需要的情况下，也允许访问单个或成组的字符。请参考6.7.1节。字符就是长度为1的简单字符串。

3. 字符串格式操作符(%)提供了类似于printf()语句的功能

在6.4.1节里，我们介绍了printf()风格的字符串格式操作符，它们用熟悉的方式对将要输出的数据的格式进行编排，既可以是对屏幕输出，也可以对其他设备输出。

4. 三引号

在6.7.2节里，我们介绍了三引号记号，括在它们中间的字符串里可以包含有换行符和水平制表符等特殊字符。三引号字符串可以成对使用连续的三个单引号('')或者双引号("")。

5. 生字符串允许使用特殊字符本身

在6.4.2节里，我们介绍了生字符串，它们不把反斜杠符号后面的字符解释为转义的特殊字符。在必须把字符串中的每个字符都看作是有效数据的情况下，比如在讨论规则表达式的时候，使用生字符串是非常理想的。

6. Python字符串不以NUL或“\0”结尾，这与C字符串不同

在C语言里有时会发生这样的问题，就是超出字符串的边界闯入本不属于你的内存空间中去。这种情况发生在C语言中的字符串没有正确地以NUL或“\0”字符（它的ASCII值是0）结尾的时候。Python语言除了为你管理内存之外，还消除了这个负担或不快。Python语言中的字符串不需要以NUL结尾，不用再担心有没有加上它们。字符串由定义给它的所有字符构成，也只有那些字符来构成。

6.10 列表

和字符串一样，列表也通过一个下标偏移值提供顺序的存储，通过切片对单个或连续的元素进行存取。但两者的相似之处一般也就是这么多了。字符串中只有字符，并且是不可变的（不能修改其中的单个元素）；列表却是能够容纳任意个Python对象的灵活的容器（container）对象。创建列表很简单，向列表添加东西也容易，从下面的例子里就可以看出来。

能放在列表里的对象可以是标准的类型和对象，也可以是用户定义的类型和对象。列表可以包容不同类型的对象，比C语言中的结构数组或Python语言中的数组（通过外部的array模块实现）更灵活，因为数组只能包含单一类型的对象。列表可以复制、清空、排序和反顺序排列其元素。列表可以增长或者缩短。它们可以切割后与其他列表重新组合。可以随意插入、修改或

删除单个或者多个数据项。

表列与列表有许多共同的特点，我们有一个专门讲解表列的小节，但列表部分的许多例子和函数对表列也是适用的。它们两者之间最关键的区别是表列是不可变的，即只读的；因此任何允许对列表进行修改的操作符或函数（比如出现在赋值语句左边的分离切片操作符）对表列来说都是非法的。

1. 如何创建和赋值列表

创建列表就像给变量附一个值那样简单，安排好一个列表（空的或有元素的）再执行赋值操作就可以了。列表是用方括号（[]）括起来的。如下所示：

```
>>> aList = [123, 'abc', 4.56, ['inner', 'list'], 7-9j]
>>> anotherList = [None, 'something to see here']
>>> print aList
[123, 'abc', 4.56, ['inner', 'list'], (7-9j)]
>>> print anotherList
[None, 'something to see here']
>>> aListThatStartedEmpty = []
>>> print aListThatStartedEmpty
[]
```

2. 如何访问列表中的值

分离切片操作与字符串的情况类似：把方括号分离切片操作符（[]）和下标或者下标范围一起使用即可。如下所示：

```
>>> aList[0]
123
>>> aList[1:4]
['abc', 4.56, ['inner', 'list']]
>>> aList[:3]
[123, 'abc', 4.56]
>>> aList[3][1]
'list'
```

3. 如何修改列表

对列表中的单个或多个元素进行修改是允许的，只要把该切片放在赋值操作符的左边就可以做到这一点；另外，还可以通过append()方法在列表里添加元素。如下所示：

```
>>> aList
[123, 'abc', 4.56, ['inner', 'list'], (7-9j)]
>>> aList[2]
4.56
>>> aList[2] = 'float replacer'
>>> aList
[123, 'abc', 'float replacer', ['inner', 'list'], (7-9j)]
>>>
>>> anotherList.append("hi, i'm new here")
>>> print anotherList
[None, 'something to see here', "hi, i'm new here"]
>>> aListThatStartedEmpty.append('not empty anymore')
>>> print aListThatStartedEmpty
['not empty anymore']
```

4. 如何删除列表中的元素和列表本身

删除列表中的某个元素有两个办法：如果准确地知道打算删除的元素到底是哪一个，可以

使用del语句；如果不太清楚，可以使用remove()方法。如下所示：

```
>>> aList
[123, 'abc', 'float replacer', ['inner', 'list'], (7-9j)]
>>> del aList[1]
>>> aList
[123, 'float replacer', ['inner', 'list'], (7-9j)]
>>> aList.remove(123)
>>> aList
['float replacer', ['inner', 'list'], (7-9j)]
```

我们还可以用pop()方法从一个列表里删除并返回一个特定的对象。

一般说来，完整删除某个列表并不是应用程序设计人员干的活。更常用的办法是离开列表的作用范围（比如程序终止、函数调用结束等），让它由系统回收。但如果你确实想明确地删除一整个列表，可以使用del语句，如下所示：

```
del aList
```

6.11 操作符

6.11.1 标准类型操作符

在第4章里，我们介绍了一些能够对大多数对象进行操作的操作符。我们来看看其中能够对列表进行操作的部分。

```
>>> list1 = [ 'abc', 123 ]
>>> list2 = [ 'xyz', 789 ]
>>> list3 = [ 'abc', 123 ]
>>> list1 < list2
1
>>> list2 < list3
0
>>> (list2 > list3) and (list1 == list3)
1
```

使用数值比较操作符对数字和字符串进行比较是容易理解的，但对列表来说就不那么好懂了。列表的比较操作有点古怪，但也很合理。比较操作符使用的是与cmp()内建函数同样的算法。该算法的基本工作流程是：比较两个列表中彼此对应的元素，直到确定出一个赢家为止。举例来说，在我们上面的例子里，'abc'和'xyz'之间进行比较的结果立刻就可以确定下来是'abc'<'xyz'，于是得到的结果为list1 < list2和list2 >= list3。表列比较操作在执行时和列表的情况是一样的。

6.11.2 序列类型操作符

1. 分离切片操作（[]和[:]）

对列表进行分离切片与对字符串的操作很相似，但涉及到的可不是单个的字符或子字符串，列表的切片取出的是一个或者一组对象，这些个对象是被操作列表中的元素。为了便于论述，我们先定义几个列表如下：

```
>>> num_list = [43, -1.23, -2, 6.19e5]
>>> str_list = ['jack', 'jumped', 'over', 'candlestick']
```

```
>>> mixup_list = [4.0, [1, 'x'], 'beef', -1.9+6j]
```

切片操作在正负下标、起始和结束下标以及缺省下标（即默认为序列的开始或结束下标）等方面遵守我们前面介绍过的原则。

```
>>> num_list[1]
-1.23
>>>
>>> num_list[1:]
[-1.23, -2, 619000.0]
>>>
>>> num_list[2:-1]
[-2]
>>>
>>> str_list[2]
'over'
>>> str_list[:2]
['jack', 'jumped']
>>>
>>> mixup_list
[4.0, [1, 'x'], 'beef', (-1.9+6j)]
>>> mixup_list[1]
[1, 'x']
```

与字符串不同的是，列表的某个元素也可以是一个序列，这就意味着对那个元素也可以使用各种序列操作或者执行各种序列内建函数。在下面的例子里，我们不仅可以从一个切片中分离出切片，还可以修改它，甚至修改为不同类型的另一个对象。请注意这与多维数组的相似性。

```
>>> mixup_list[1][1]
'x'
>>> mixup_list[1][1] = -64.875
>>> mixup_list
[4.0, [1, -64.875], 'beef', (-1.9+6j)]
```

下面是使用num_list的另外一个例子：

```
>>> num_list
[43, -1.23, -2, 6.19e5]
>>>
>>> num_list[2:4] = [16.0, -49]
>>>
>>> num_list
[43, -1.23, 16.0, -49]
>>>
>>> num_list[0] = [65535L, 2e30, 76.45-1.3j]
>>>
>>> num_list
[[65535L, 2e+30, (76.45-1.3j)], -1.23, 16.0, -49]
```

请注意：在最后一个例子里，我们只修改了列表中的一个元素，但我们是把它修改为一个列表的。所以我们说，在列表里删除、添加和替换东西是相当自由随意的。但要记住，如果想把一个列表分离切片为另一个列表，一定要保证赋值操作符（即等号=）的左边必须是一个列表，不能是单个的元素。

2. 成员操作 (in, not in)

成员操作符对字符串的操作是确定某个字符是否是一个字符串的成员。对列表（也包括表

列)而言, 我们可以检查某个对象是否是一个列表(或者表列)的成员, 如下所示:

```
>>> mixup_list
[4.0, [1, 'x'], 'beef', (-1.9+6j)]
>>>
>>> 'beef' in mixup_list
1
>>>
>>> 'x' in mixup_list
0
>>> num_list
[[65535L, 2e+030, (76.45-1.3j)], -1.23, 16.0, -49]
>>>
>>> -49 in num_list
1
>>>
>>> 34 in num_list
0
>>>
>>> [65535L, 2e+030, (76.45-1.3j)] in num_list
1
```

注意, 'x'并不是mix_list的一个成员, 这是因为'x'本身实际并不是mix_list的一个成员; 它是mixup_list[1]的一个成员, 而它本身又是一个列表。成员操作符对表列的执行情况也与此相同。

3. 合并操作 (+)

我们可以用合并操作符把几个列表组合在一起。请注意: 在下面的例子里, 对象的合并操作也是有限制的, 换句话说, 只能合并同类型的对象。即使都是序列, 不同类型之间也不能合并。

```
>>> num_list = [43, -1.23, -2, 6.19e5]
>>> str_list = ['jack', 'jumped', 'over', 'candlestick']
>>> mixup_list = [4.0, [1, 'x'], 'beef', -1.9+6j]
>>>
>>> num_list + mixup_list
[43, -1.23, -2, 619000.0, 4.0, [1, 'x'], 'beef', (-1.9+6j)]
>>>
>>> str_list + num_list
['jack', 'jumped', 'over', 'candlestick', 'park', 43, -1.23, -2, 619000.0]
```

我们将在6.13节发现: 从Python 1.5.2版本开始, 你就可以用extend()方法代替合并操作符把某个列表的内容追加到另一个的后面。使用extend()方法要比使用合并操作符更好, 因为它确实把新列表中的元素追加到原列表的尾部, 不像加号(+)那样需要创建一个新的列表。在Python 2.0版本里, extend()方法还可以用在新的增量赋值语句中, 即可以替换合并操作符(+=)。

需要指出的是, 合并操作符并不能够用于向一个列表里添加单个的元素。下面的例子就是试图向列表添加一个新数据项时导致了出错。

```
>>> num_list + 'new item'
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation
```

这个例子出错的原因是合并操作符左右两边出现了不同的类型。(列表+字符串)方式的合并操作是非法的。我们的目的是把'new item'字符串添加到列表中去,只是采取的办法不正确罢了。好在我们还有别的招数:

使用append()列表内建方法(我们将在6.13节正式介绍append()和其他的内建方法):

```
>>> num_list.append('new item')
```

4. 重复操作 (*)

对字符串使用重复操作符的优点就不必多说了,但同属序列类型的列表和表列也可以从这个操作符处受益,如下所示:

```
>>> num_list * 2
[43, -1.23, -2, 619000.0, 43, -1.23, -2, 619000.0]
>>>
>>> num_list * 3
[43, -1.23, -2, 619000.0, 43, -1.23, -2, 619000.0, 43, -1.23, -2, 619000.0]
```

Python 2.0版本里增加了一个可替代重复操作符的增量赋值语句语法。

6.11.3 列表类型操作符

[2.0]

目前,Python语言里还没有列表专用的操作符。大多数对象和序列操作符都可以对列表使用。除此之外,列表对象还有一些它们自己的方法。

6.12 内建函数

6.12.1 标准类型函数

cmp()

在4.6.1节里,我们以数字和字符串为例向大家介绍了cmp()内建函数。那么,cmp()对列表和表列这样的对象又有什么作用呢?要知道,它们不仅能够包含数字和字符串,还能够包含其他对象如列表、表列、字典,甚至用户创建的对象等。

```
>>> list1, list2 = [123, 'xyz'], [456, 'abc']
>>> cmp(list1, list2)
-1
>>>
>>> cmp(list2, list1)
1
>>> list3 = list2 + [789]
>>> list3
[456, 'abc', 789]
>>>
>>> cmp(list2, list3)
-1
```

对两个同类型的对象进行比较操作很直观。对数字和字符串来说,就是比较它们的值的大小。对序列类型来说,比较操作就有些复杂,但方式还是相似的。在没有明显的大小先后顺序

的情况下——比如对象之间没有什么关联，或者类型根本就没有比较函数等——Python会尽量做一个公平的比较，然后尽量得出一个“符合逻辑”的结果。

在到达这种“最后一博”的境地之前，Python还是会尽量选择一条更安全和更合理的不等性判断之路。这个算法到底是怎样的呢？我们在前面已经简要地提到过，列表中的元素将被遍历。如果这些元素是同一个类型，那么就使用该类型的标准比较操作；并且只要在某个元素比较时确定了一个不等性，该结果就成为这次列表比较操作的结果。再说一次，这些元素比较是对同类型元素进行的。我们刚才也解释过，如果对象的类型不同，进行准确或真实的比较是比较冒险的。

对list1和list2进行比较的时候，两个列表都将被遍历。两个列表各自的第一个元素进行真正的比较，即123比456；因为 $123 < 456$ ，所以list1被认为“比较小”。

如果两个值一样，就会继续遍历序列，直到找出一个不等式关系或者到达比较短的序列的尾部。如果出现后一种情况，那个元素比较多的序列将被认为是“比较大”的。这就是为什么我们在上面得出 $list2 < list3$ 结论的原因。表列之间的比较操作也使用同样的算法。下面是一些对比较操作所做的总结：

- 比较两个列表中的元素。
- 如果元素是同一种类型，执行比较操作并返回结果。
- 如果元素不是同一种类型，检查它们是否都是数字。
 - 如果都是数字，进行必要的类型协调后再进行比较。
 - 如果只有一个是数字，就判定另一个元素“比较大”（数字是“最小的”）。
 - 否则，根据类型名字的字母顺序排定大小。
- 如果到达一个列表的尾部，那么比较长的列表“比较大”。
- 如果遍历两个列表时数据都一样，结果是平局；返回一个零值（0）。

6.12.2 序列类型函数

1. len()

对字符串而言，len()给出的是字符串的长度，也就是其中字符的个数。对列表（或者表列）而言，如果len()返回的是列表（或者表列）中元素的个数，想必你也不会感到意外。包容器对象算做一个数据。我们下面的例子使用了前面小节里定义的几个列表。

```
>>> len(num_list)
4
>>>
>>> len(num_list*2)
8
>>>
>>> len(str_list[:4])
4
>>>
>>> len(str_list[:-1])
4
>>>
>>> len(mixup_list+num_list)
8
```

2. max()和min()

max()和min()对字符串用得不多，因为它们的作用只是找出字符串中“最大”和“最小”的字符而已（按ASCII表顺序）。对列表（和表列）而言，它们的功能就多一些了。给定一个同类对象的列表，比如全部都是数字或者字符串的列表，max()和min()可以方便地找出其中的最大值和最小值；如果遇到的是混合对象，它们返回值的质量就要打折扣。无论如何，你总会需要使用这两个函数求得结果。我们在下面给出几个使用前面定义的列表进行操作的例子。

```
>>> max(str_list)
'park'
>>>
>>> max(num_list)
[65535L, 2e+30, (76.45-1.3j)]
>>> max(mixup_list)
'beef'
>>> min(mixup_list)
(-1.9+6j)
>>>
>>> min(str_list)
'candlestick'
>>>
>>> min(num_list)
-49
```

3. list()和tuple()

list()和tuple()方法把序列类型分别转换为列表和表列。虽然字符串也算得上是序列类型，但它们很少和list()和tuple()方法一起使用。这些内建函数更经常的是用来把一种类型转换为另一种类型，比如把一个表列转换为列表（这样就可以修改其元素了）或者把列表转换为表列等。

```
>>> aList = [ 'tao', 93, 99, 'time' ]
>>> aTuple = tuple(aList)
>>> print aList
['tao', 93, 99, 'time']
>>>
>>> print aTuple
('tao', 93, 99, 'time')
>>>
>>> back2aList = list(aTuple)
>>> back2aList
['tao', 93, 99, 'time']
>>> back2aList == aList
1
>>> back2aList is aList
0
```

list()和tuple()都不真正执行具体的转换操作（请参考6.1.2节内容）。换句话说，你传递给tuple()的列表并不会变成一个表列，而传递给list()的表列也不会真的变成一个列表。这些内建函数会创建一个指定类型的新对象，把原来序列中的元素赋值到新对象里去。在上面最后的两个例子里，虽然两个列表中的数据是一样的（满足==关系），但两个变量指向的可不是同一个列表（不满足is关系）。

6.12.3 列表类型内建函数

目前，Python语言里还没有列表专用的内建函数。列表可以和大多数对象和序列内建函数一起使用。除此之外，列表对象还有一些它们自己的方法。

6.13 列表类型的内建方法

Python语言中的列表有自己的“方法”。我们将在第13章里向大家正式介绍面向对象的程序设计中的方法，但现在，把方法想象为只对特定对象起作用的内建函数或者过程就行了。因此，本节介绍的方法其行为就像是内建函数，只不过它们都只对列表起作用。因为这些函数都涉及到列表的可变性（或者叫刷新），所以都不适用于表列。

前面的论述中介绍过：访问对象的属性要使用点属性记号`object.attribute`。列表的方法也不例外，需要使用`list.method([arguments])`来访问。我们用点记号来访问属性（这里就是某个函数），再在一个函数性记号里使用函数操作符（即括号`()`）来调用该方法。

带方法的对象一般都有一个名为`object.__methods__`的属性，它会给出该类型所支持的全部方法。在我们的列表情况中，`list.__methods__`就是为这个目的预备好了的： [1.5.2]

```
>>> [].__methods__
['append', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```

表6-10列出了现时期适用于列表的所有方法。这些方法中的`extend()`和`pop()`是从Python 1.5.2版本开始有的。再后面是一些使用各种列表方法的例子。

表6-10 列表类型内建方法

列表方法	操 作
<code>list.append(obj)</code>	把对象obj追加在列表list的尾部
<code>list.count(obj)</code>	返回对象obj在列表list中出现的次数
<code>list.extend(seq)*</code>	把序列seq追加在列表list的尾部
<code>list.index(obj)</code>	返回对象obj在列表list中出现时的最小下标
<code>list.insert(index, obj)</code>	把对象obj插入列表list中下标为index的位置上
<code>list.pop(obj=list[-1])*</code>	删除并返回列表list中最顶端的对象或obj对象
<code>list.remove(obj)</code>	从列表list里删除对象obj
<code>list.reverse()</code>	反方向排列列表list中的对象
<code>list.sort([func])</code>	对列表中的对象进行排序；如果给出了func，就按它指定的比较操作进行

① 从Python 1.5.2版本开始具备的。

```
>>> music_media = [45]
>>> music_media
[45]
>>>
>>> music_media.insert(0, 'compact disc')
>>> music_media
['compact disc', 45]
>>>
>>> music_media.append('long playing record')
>>> music_media
```



```
['compact disc', 45, 'long playing record']
>>>
>>> music_media.insert(2, '8-track tape')
>>> music_media
['compact disc', 45, '8-track tape', 'long playing record']
```

在刚才的例子中，我们创建了一个只有一个元素的列表，然后在其中插入一个元素和在它的尾部追加一个元素后又查看了这个列表。我们现在来看看这样才能确定元素是否属于列表以及如何找出元素在列表里的位置。我们要使用`in`操作符和`index()`方法，如下所示：

```
>>> 'cassette' in music_media
0
>>> 'compact disc' in music_media
1
>>> music_media.index(45)
1
>>> music_media.index('8-track tape')
2
>>> music_media.index('cassette')
Traceback (innermost last):
  File "<interactive input>", line 0, in ?
ValueError: list.index(x): x not in list
```

哎呀！最后一个例子好像有点不对劲？看起来用`index()`检查数据项是否属于某个列表不是一个好办法，因为我们看到了一个错误信息。比较稳妥的办法是：先用成员操作符`in`（或者`not in`）进行检查，再用`index()`确定元素所在的位置。我们可以把最后的几个`index()`调用放在一个循环里，如下所示：

```
for eachMediaType in (45, '8-track tape', 'cassette'):
    if eachMediaType in music_media:
        print music_media.index(eachMediaType)
```

这个解决方案可以避免出现上面遇到的问题，因为我们是在确定了对象确实是在列表中以后才调用`index()`方法的。我们以后会介绍如何在遇到出错的情况下接手其处理工作，这样就不会像刚才那样被轰出来了。

我们再来看看`sort()`和`reverse()`方法，它们一个用来对列表中的元素进行排序；一个用来反向排列列表中的元素。如下所示：

```
>>> music_media
['compact disc', 45, '8-track tape', 'long playing record']
>>> music_media.sort()
>>> music_media
[45, '8-track tape', 'compact disc', 'long playing record']
>>> music_media.reverse()
>>> music_media
['long playing record', 'compact disc', '8-track tape', 45]
```

关于`sort()`和`reverse()`方法需要特别注意的一点是：它们都是直接在列表上实现它们的操作的，也就是说，列表原来的内容将被改变。这两种方法都没有返回值。

如果你对算法很有心，那我不妨在这里告诉你，`sort()`方法缺省使用的排序算法是随机版本的QuickSort算法。我们把这个排序算法的详细资料安排在其源程序代码里的注释中，你可以到

那里看看（文件Objects/listobject.c）。

新的extend()方法可以以一个列表为执行参数，把它的元素追加到另外一个列表的末尾，如下所示：

```
>>> new_media = ['24/96 digital audio disc', 'DVD Audio
disc', 'Super Audio CD']
>>> music_media.extend(new_media)
>>> music_media
['long playing record', 'compact disc', '8-track tape',
45, '24/96 digital audio disc', 'DVD Audio disc', 'Super
Audio CD']
```

从Python 1.6版本开始，extend()方法的执行参数可以是任何序列对象——这个序列对象会先被list()之类的方法转换为一个列表，然后再把它的内容追加到最初的列表后面去。在1.5.2版本里，该参数必须是一个列表。

pop()方法可以从列表里取得最顶部的或者用参数指定的那个数据项，把它返回给调用者。我们将在6.14.1节和本章练习里看到新的pop()方法。

6.14 列表的特性

6.14.1 利用列表创建其他数据结构

列表具有容器和可被修改的特性，也就是说它们具有很好的灵活性，所以通过列表建立其他种类的数据结构并不困难。我们下面就来介绍两种，它们是堆栈和队列。 [1.5.2]

在本节的两个例子里都使用了pop()方法，它是从Python 1.5.2版本开始引入的。如果读者使用的是一个比较早期的系统，在Python语言里也能很容易地把这个函数移植过去（请参考练习6-17）。

1. 堆栈

堆栈是一个后进先出（last-in-first-out, LIFO）的数据结构形式。我们可以把对象想象成盘子，那么堆栈就很像食堂里的碗架：从碗架上拿下来的第一个盘子应该是最后一个被放上去的；每一个新对象会被“堆放”到最新的对象的上面。往堆栈里添加数据项的术语叫做往堆栈“压入”一个数据；而从中移出数据项就叫做从堆栈“弹出”一个数据。程序示例6-2是一个菜单驱动的程序，它实现了一个用来保存字符串的简单堆栈。

程序示例6-2 使用列表的堆栈（stack.py）

下面这个简单的脚本程序用列表构造了一个堆栈。这个用菜单驱动的字符串处理程序实现字符串的保存和检索操作时只使用了append()和pop()两种列表方法。

```
1  #!/usr/bin/env python
2
3  stack = []
4
5  def pushit():
6      stack.append(raw_input('Enter new string: '))
7
8  def popit():
```

```

9     if len(stack) == 0:
10         print 'Cannot pop from an empty stack!'
11     else:
12         print 'Removed [' , stack.pop(), ']'
13
14 def viewstack():
15     print str(stack)
16
17 def showmenu():
18     prompt = ""
19     p(U)sh
20     p(O)p
21     (V)iew
22     (Q)uit
23
24 Enter choice: ""
25
26     done = 0
27     while not done:
28
29         chosen = 0
30         while not chosen:
31             try:
32                 choice = raw_input(prompt)[0]
33             except (EOFError, KeyboardInterrupt):
34                 choice = 'q'
35             print '\nYou picked: [%s]' % choice
36             if choice not in 'uovq':
37                 print 'invalid option, try again'
38             else:
39                 chosen = 1
40
41         if choice == 'q': done = 1
42         if choice == 'u': pushit()
43         if choice == 'o': popit()
44         if choice == 'v': viewstack()
45
46 if __name__ == '__main__':
47     showmenu()

```

1~3行

UNIX操作系统下的启动行，堆栈（即一个列表）清零。

5~6行

pushit()函数把一个元素（提示用户输入的一个字符串）压入堆栈。

8~12行

popit()函数从堆栈里移出一个元素（最新被压入的那个）。如果堆栈已经是空的，再试图从中移出一个元素时会引发一个错误；此时向用户送回一个警告信息。

14~15行

viewstack()函数显示列表中某个字符串的可打印表示形式。

17~44行

整个菜单驱动是由showmenu()函数控制的。它向用户显示一个菜单，用户通过菜单选择准备执行的操作；如果用户的选择有效，就调用相应的函数。我们还没有详细讨论过例外处理和try-except语句，但这段代码的基本作用是允许用户输入^D（EOF文件尾标志，它产生一个EOFError错误）和^C（中止执行并退出，它产生一个KeyboardInterrupt错误）符号，我们的脚本程序会把这两种情况都当做是用户输入“q”选项退出程序的情况来对待。从这里也可以看出

Python语言中的例外处理功能是非常方便的。

46 ~ 47行

如果程序是被直接调用的，这段程序就会被执行。如果这个脚本程序是做为一个模块被导入的，就只会对函数和变量进行定义，菜单是不会显示的。关于第46行和__name__变量的进一步说明请参考3.4.1节内容。

下面是我们脚本程序的一次执行情况：

```
% stack.py
p(U)sh
p(O)p
(V)iew
(Q)uit
Enter choice: u
You picked: [u]
Enter new string: Python

p(U)sh
p(O)p
(V)iew
(Q)uit

Enter choice: u
You picked: [u]
Enter new string: is

p(U)sh
p(O)p
(V)iew
(Q)uit

Enter choice: u
You picked: [u]
Enter new string: cool!

p(U)sh
p(O)p
(V)iew
(Q)uit

Enter choice: v
You picked: [v]
['Python', 'is', 'cool!']

p(U)sh
p(O)p
(V)iew
(Q)uit

Enter choice: o
You picked: [o]
Removed [ cool! ]

p(U)sh
p(O)p
(V)iew
(Q)uit
Enter choice: o
```

```
You picked: [o]
Removed [ is ]

p(U)sh
p(O)p
(V)iew
(Q)uit

Enter choice: o

You picked: [o]
Removed [ Python ]

p(U)sh
p(O)p
(V)iew
(Q)uit

Enter choice: o

You picked: [o]
Cannot pop from an empty stack!

p(U)sh
p(O)p
(V)iew
(Q)uit

Enter choice: ^D
You picked: [q]
```

2. 队列

队列是一个先进先出（first-in- first-out, FIFO）的数据结构形式。我们可以把它想象成一个只有一条通道的超市或者银行柜台：排在第一的顾客是第一个得到服务的（因此也应该是第一个离开的）；新元素加入队列时要排到最后一个位置，从队列前端移出一个元素就叫做“离开队列”。对刚才的堆栈脚本程序做了很少的修改就得到了程序示例6-3的代码，它用列表实现了一个简单的队列。

程序示例6-3 使用列表的队列（queue.py）

下面这个简单的脚本程序用列表构造了一个队列，实现了字符串的保存和检索操作。这个用菜单驱动的字符串处理程序只使用了 `append()` 和 `pop()` 两种列表方法。

```
1  #!/usr/bin/env python
2
3  queue = []
4
5  def enQ():
6      queue.append(raw_input('Enter new string: '))
7
8  def deQ():
9      if len(queue) == 0:
10         print 'Cannot dequeue from empty queue!'
11     else:
12         print 'Removed [' , queue.pop(0) , ']'
13
14 def viewQ():
```

```

15     print str(queue)
16
17 def showmenu():
18     prompt = ""
19     (E)nqueue
20     (D)equeue
21     (V)iew
22     (Q)uit
23
24 Enter choice: ""
25
26     done = 0
27     while not done:
28
29         chosen = 0
30         while not chosen:
31             try:
32                 choice = raw_input(prompt)[0]
33             except (EOFError, KeyboardInterrupt):
34                 choice = 'q'
35             print '\nYou picked: [%s]' % choice
36             if choice not in 'devq':
37                 print 'invalid option, try again'
38             else:
39                 chosen = 1
40
41         if choice == 'q': done = 1
42         if choice == 'e': enQ()
43         if choice == 'd': deQ()
44         if choice == 'v': viewQ()
45
46 if __name__ == '__main__':
47     showmenu()

```

因为这个脚本程序和stack.py脚本程序很相似，所以我们只对有明显改动的语句行加以详细的说明。

5~6行

enQ()函数的工作原理和pushit()完全一样，只是改了一个名字。

8~12行

这里是两个脚本程序的主要区别之处。deQ()函数与popit()的区别是：它不是对列表中最新的数据项进行操作，而是对列表中最早的数据项，即列表中的第一个元素进行操作。

下面给出一些程序执行时的输出：

```

% queue.py
(E)nqueue
(D)equeue
(V)iew
(Q)uit

Enter choice: e

You picked: [e]
Enter new queue element: Bring out

(E)nqueue
(D)equeue
(V)iew
(Q)uit

```

```
Enter choice: e

You picked: [e]
Enter new queue element: your dead!

(E)nqueue
(D)equeue
(V)iew
(Q)uit

Enter choice: v

You picked: [v]
['Bring out', 'your dead!']

(E)nqueue
(D)equeue
(V)iew
(Q)uit

Enter choice: d

You picked: [d]
Removed [ Bring out ]

(E)nqueue
(D)equeue
(V)iew
(Q)uit

Enter choice: d

You picked: [d]
Removed [ your dead! ]

(E)nqueue
(D)equeue
(V)iew
(Q)uit

Enter choice: d

You picked: [d]
Cannot dequeue from empty queue!

(E)nqueue
(D)equeue
(V)iew
(Q)uit

Enter choice: ^D
You picked: [q]
```

6.14.2 列表的子类

我们在本书前面曾经介绍过，Python语言中的类型不是类（class），因此我们不能从中引出

子类（请参考4.2节中的编程提示）。但在Python语言的标准函数库里有两个模块，它们分别包含了由列表和字典两个类型打包而来的类（class）做为其代理，我们可以对它们进行引出子类。这两个模块是UserList和UserDict。熟悉类以后，你就可以通过这些已经实现好的类从列表和字典类型开始创建自己的子类，还可以加上自己想要的函数。这些模块都是Python标准库的组成部分。详细情况请参考13.16节。

6.15 表列

表列是另外一种包容器类型，它和列表非常相似。表列和列表之间可观察到的区别只是：表列使用圆括号，列表使用方括号。但在功能上，它们有一个更明显的区别，那就是表列是不可变的。

我们在介绍某种对象的时候一般会先给出与之对应的操作符和内建函数，再给出该类对象（此处就是表列）专用的序列操作符和内建函数；但因为表列和列表之间有如此多的相似之处，所以有很多内容会与前面那一小节是重复的。因此我们尽量避免大量的重复内容，把注意力集中在表列独有的操作符和功能方面，然后向大家介绍表列独有的不可变性和其他特色功能。

1. 如何创建和赋值表列

表列的创建和赋值过程从操作过程来看与对列表的情况是一样的，但空表列是一个例外。空表列必须有一个用表列定义符号圆括号（()）括起来的尾缀逗号（,），如下所示：

```
>>> aTuple = (123, 'abc', 4.56, ['inner', 'tuple'], 7-9j)
>>> anotherTuple = (None, 'something to see here')
>>> print aTuple
(123, 'abc', 4.56, ['inner', 'tuple'], (7-9j))
>>> print anotherTuple
(None, 'something to see here')
>>> emptiestPossibleTuple = (None,)
>>> print emptiestPossibleTuple
(None,)
```

2. 如何访问表列中的值

分离切片操作与列表的情况类似：把方括号分离切片操作符（[]）和下标或者下标范围一起使用即可，如下所示：

```
>>> aTuple[1:4]
('abc', 4.56, ['inner', 'tuple'])
>>> aTuple[:3]
(123, 'abc', 4.56)
>>> aTuple[3][1]
'tuple'
```

3. 如何修改表列

类似于数字和字符串，表列是不可变的，也就是说，不能对表列进行修改或者改变表列元素的值。但我们在6.2和6.3.2节里演示过如何从一个现有字符串里取出一个部分并用它来构成一个新的字符串。这也同样适用于表列，如下所示：


```
>>> aTuple = aTuple[0], aTuple[1], aTuple[-1]
>>> aTuple
(123, 'abc', (7-9j))
>>> tup1 = (12, 34.56)
>>> tup2 = ('abc', 'xyz')
>>> tup3 = tup1 + tup2
>>> tup3
(12, 34.56, 'abc', 'xyz')
```

4. 如何删除表列中的元素和表列本身

删除表列中的某个元素是不可能的。但如果丢弃某些个不想要元素，再把剩余的组合为另一个表列可没有什么不对。如果确实想明确地删除一个表列，可以使用del语句，如下所示：

```
del aTuple
```

6.16 表列的操作符和内建函数

6.16.1 标准和序列操作符与内建函数

对象和序列的操作符与内建函数对表列的操作过程和它们对列表的操作过程是完全一样的。我们仍然能够对表列进行分离切片操作、合并与重复复制操作、验证其成员性，并且能够对表列进行比较，如下所示：

1. 创建、重复、合并操作

```
>>> t = (['xyz', 123], 23, -103.4)
>>> t
(['xyz', 123], 23, -103.4)
>>> t * 2
(['xyz', 123], 23, -103.4, ['xyz', 123], 23, -103.4)
>>> t = t + ('free', 'easy')
>>> t
(['xyz', 123], 23, -103.4, 'free', 'easy')
```

2. 成员、分离切片操作

```
>>> 23 in t
1
>>> 123 in t
0
>>> t[0][1]
123
>>> t[1:]
(23, -103.4, 'free', 'easy')
```

3. 内建函数

```
>>> str(t)
(['xyz', 123], 23, -103.4, 'free', 'easy')
>>> len(t)
5
>>> max(t)
'free'
```

```
>>> min(t)
-103.4
>>> cmp(t, (['xyz', 123], 23, -103.4, 'free', 'easy'))
0
>>> list(t)
[['xyz', 123], 23, -103.4, 'free', 'easy']
```

4. 操作符

```
>>> (4, 2) < (3, 5)
0
>>> (2, 4) < (3, -1)
1
>>> (2, 4) == (3, -1)
0
>>> (2, 4) == (2, 4)
1
```

6.16.2 表列类型操作符和内建函数及方法

类似于列表，表列也没有专用的操作符和内建函数。前面小节介绍的对列表进行处理的各种方法都是与列表对象的可变性有关的，比如排序、替换、元素追加等操作。因为表列是不可变的，超出了那些方法的能力范围，因此是不可用的。

6.17 表列的特性

6.17.1 不可变性对表列有何影响

我们在这本书里的许多地方都会遇到“不可变的”这个词。撇开它在计算机科学里的定义和意义，从应用程序的角度看，它的底线是什么？对一个不可变的数据类型来说，我们又可以对它进行什么操作呢？

在三个不可变的标准类型（数字、字符串、表列）里，表列所受的影响是最大的。一个不可变的数据类型说的就是只要一个对象被定义好了，它的值就不能再发生变化；除非另外创建一个全新的对象。因为数字和字符串都属于离散类型，所以这对它们的影响还不是很大；即使它们代表的值整个被改变的时候，其变化也在我们的意料之中，对它们的访问还可以照常进行。但对表列来说，事情就不一样了。

因为表列是一种包容器类型，所以该包容器中的单个或多个元素是极有可能会发生变化的。但不幸的是，这是不允许的。分离切片操作符不能出现在赋值语句的左边；这和字符串的情况是完全一样的，切片操作只能用于读操作访问目的。

不可变性并不一定是件坏事。亮点之一就是：如果把数据传递到一个我们不熟悉的API中，可以肯定的是数据不会被我们调用的函数所改变。而如果某个函数的返回参数是一个表列，而我们又想对它进行修改等操作，可以使用list()内建函数把它转换为一个可变的列表。

6.17.2 表列也不是绝对“不可变的”

虽然表列被定义为不可变的，可这并不会影响到它们的灵活性。表列并不像我们定义的那

样完全不可改变。哎呀，这是什么意思？表列的某些行为确实使它们看起来不像我们定义的那样完全的不可改变。

举例来说，我们可以把字符串合并在一起构成一个更大的字符串。类似地，把表列合并在一起构成一个更大的表列也没有什么不对，也就是说，合并操作是能够执行的。这一过程并不会单独改变那些比较小的表列。我们所做的只是把它们的元素结合在一起而已。下面是几个这样做的例子：

```
>>> s = 'first'
>>> s = s + ' second'
>>> s
'first second'
>>>
>>> t = ('third', 'fourth')
>>> t
('third', 'fourth')
>>>
>>> t = t + ('fifth', 'sixth')
>>> t
('third', 'fourth', 'fifth', 'sixth')
```

重复操作也具有同样的概念。重复操作只不过是把同样的元素多次重复之后再合并到一起罢了。此外，我们在前面也提到过可以通过一个简单的函数调用把一个表列转换为一个可变的列表。现在告诉你一个最让你吃惊的特色：你可以“修改”表列的某个特定的元素。哇，什么意思？

虽然表列本身是不可变的，但这个事实并不会妨碍表列里包含有可以被修改的可变对象。如下所示：

```
>>> t = (['xyz', 123], 23, -103.4)
>>> t
(['xyz', 123], 23, -103.4)
>>> t[0][1]
123
>>> t[0][1] = ['abc', 'def']
>>> t
(['xyz', ['abc', 'def']], 23, -103.4)
```

在上面的例子里，虽然t是一个表列，但我们完全可以修改第一个表列元素（那是一个列表）。我们把原来是一个整数的t[0][1]修改为列表['abc', 'def']。虽然我们只是修改了一个可变的对象，但从某种意义上来说，我们也就是修改了这个表列。

6.17.3 括号的作用

一个包含多个对象的集合，如果它是用逗号分隔的，并且没有标识性的定义符号（即定义列表用的方括号，或者定义表列用的圆括号），就会被认为是一个表列，如下面几个简单的例子所示：

```
>>> 'abc', -4.24e93, 18+6.6j, 'xyz'
('abc', -4.24e+093, (18+6.6j), 'xyz')
>>>
```

```
>>> x, y = 1, 2
>>> x, y
(1, 2)
```

从函数调用返回的任何多个对象（如果没有括号括住它们）也被认为是一个表列。请注意，括号符号会把一个包含多个对象的集合定义为一个单个的容器对象，如下所示：

```
def foo1():
    :
    return obj1, obj2, obj3

def foo2():
    :
    return [obj1, obj2, obj3]

def foo3():
    :
    return (obj1, obj2, obj3)
```

在上面的例子里，foo1()调用返回了三个对象，它们将被看做是一个由三个对象构成的表列；foo2()调用返回的是一个对象，即一个包含三个对象的列表；而foo3()与foo1()返回的东西是一样的，唯一的区别是明确地使用圆括号定义了表列。

在创建表达式或表列时明确地使用括号进行分组是非常好的做法，这样可以避免一些不可预料的副作用，如下所示：

```
>>> 4, 2 < 3, 5      # int, comparison, int
(4, 1, 5)
>>> (4, 2) < (3, 5) # tuple comparison
0
```

在第一个例子里，小于(<)操作符的优先级高于它两侧的表列元素分隔符逗号，因此“2<3”比较操作的结果成为表列的第二个元素。而在第二个例子里，给表列加上适当的括号之后，我们就得到预期的结果了。

6.17.4 单元素表列

有没有试过创建一个只有一个元素的表列？创建一个只有一个元素的列表——没问题；但在表列上试了又试，好象永远也没办法成功似的。你看到的是不是下面这样的结果？

```
>>> ['abc']
['abc']
>>> type(['abc'])      # a list
<type 'list'>
>>>
>>> [123]
[123]
>>> type([123])        # also a list
<type 'list'>
>>>
>>> ('xyz')
'xyz'
>>> type(('xyz'))      # a string, not a tuple
<type 'string'>
>>>
```

```
>>> (456)
456
>>> type((456))      # an int, not a tuple
<type 'int'>
```

因为圆括号同时也被用做是表达式的分组操作符，所以好象没有什么能帮助你创建一个元素的表列来。括住单个元素的括号被认为是一个表达式分组操作符而不是表列的定义符号。解决办法是在第一个元素的后面再加上一个尾缀的逗号(,)，用它来表示这是一个表列而不是一个表达式分组，如下所示：

```
>>> ('xyz',)
('xyz',)
>>> (456,)
(456,)
```

编程提示：列表和表列的对比

[CN]

Python常见问题答疑FAQ (6.15) 中有一个问题问到：“为什么会有列表和表列这两种数据类型？”这个问题也可以被表达为：“我们真的需要两种序列类型吗？”同时拥有列表和表列的好处之一是：在某些情况下，这一种会比那一种更适合我们的需要。

比较适合使用不可变数据类型的情况是：如果你正在处理很关键的数据，并且需要把一个可变的对象传递到一个不太熟悉的函数去（它可能是一个不是你自已编写的API！）。做为开发这一部分软件的工程师，如果了解到准备调用的函数不会改变自己的数据，肯定会觉得更加安全一些。

把可变数据类型做为一个参数的情况是你需要对数据集合进行动态管理的场合。你需要能够随时随地地创建它们、逐渐或任意地进行添加它们，或者时不时地删除个别的元素。这时候需要的数据类型肯定是要可变的。还有一个好消息，那就是你可以通过list()和tuple()内建转换函数轻松地把一种类型转换为另外一种类型。

list()和tuple()函数允许你把表列转换为列表，或者把列表转换为表列。如果需要对一个表列里的对象进行修改，就需要把它转换为一个列表，而这时list()函数就会是你最好的伙伴。如果需要把一个列表传递到一个函数（也许是某个API）中去，又不想这些数据在函数中被其他人修改，tuple()函数就正好管用。

6.18 相关模块

表6-11列出了几个主要的与序列类型有关的模块。这个清单里包括我们在前面曾经简单介绍过的array数组模块；数组与列表很相似，但其全体元素被限制为只能是同一种数据类型。copy拷贝模块（请参考6.19节的内容）用来实现对象的浅拷贝和深拷贝。operator操作符模块里除数值操作符的对等功能函数外，还包含着同样的四种序列类型。types类型模块是代表Python语言所支持的全体数据类型的各种类型对象的一个引用线索库，包括Python的各种序列类型。最后一个UserList模块是列表对象的一个功能完整的类（class）实现方式；因为Python语言中的类型不能进一步分出子类，所以它准备了这样一个模块，给用户一个具有列表特性的类，用户可以

再由此推导出新的类和功能函数。如果读者还不熟悉方向对象的程序设计，我们推荐你学习第13章。

表6-11 序列类型的相关模块

模 块	内 容
array	可变序列数据类型array数组的各种功能，它要求其元素都是同一种类型的
copy	提供了实现对象浅拷贝和深拷贝的函数功能（详细资料请参考6.19节内容）
operator	包含有以函数调用方式实现的序列操作符，比如说，函数operator.concat(m, n)就相当于序列m和n的合并操作(m+n)
types	包含着Python语言支持的各种类型的类型对象
UserList	把一个列表对象（包括操作符和方法）打包在一个类中，这个类可以用来构造子类（请参考13.16节）

6.19 *浅拷贝与深拷贝

早在3.5节我们就已经说明了对象的赋值只不过是对象的引用。这句话的意思是说当你创建了一个对象，然后再把它赋值给另外一个变量的时候，Python并不会复制这个对象；它复制的只是那个对象的引用。如下所示：

```
>>> aList = [[78, 'pyramid'], [84, 'vulture'], [81, 'eye']]
>>> anotherList = aList
>>> aList
[[78, 'pyramid'], [84, 'vulture'], [81, 'eye']]
>>>
>>> anotherList
[[78, 'pyramid'], [84, 'vulture'], [81, 'eye']]
```

在上面，我们创建了一个包含三个元素的列表，它的引用被赋值给aList。当把aList赋值给anotherList的时候，新创建的anotherList并没有复制以aList做为引用的列表的内容。anotherList“复制”的是aList的引用，不是具体的数据。通过查看两个引用指向的对象标识可以确认这一点，如下所示：

```
>>> id(aList)
1191872
>>> id(anotherList)
1191872
```

一个对象的“浅拷贝”被定义为一个新创建的对象，它的类型与原对象一样，而其内容却是指向原对象元素的引用。换句话说，被拷贝的对象本身是新的，但其内容却不然。序列对象的浅拷贝可能会有以下两种形式：（1）用分离切片操作符取出一个完整的切片；（2）使用copy拷贝模块中的copy()函数，如下面的例子所示：

```
>>> thirdList = aList[:]
>>> thirdList
[[78, 'pyramid'], [84, 'vulture'], [81, 'eye']]
>>> id(thirdList)
1192232
>>>
>>> import copy
```

```
>>> fourthList = copy.copy(aList)
>>> fourthList
[[78, 'pyramid'], [84, 'vulture'], [81, 'eye']]
>>> id(fourthList)
1192304
```

thirdList列表是通过切片操作符创建的，它取出的是aList列表的完整切片（同时缺少起始下标和结束下标）。我们还给出了新对象的标识以确认它确实与原来的对象没有关联。fourthList列表的创建也是如此，我们这次使用了copy.copy()函数来实现同样的目的。但是，这些列表指向的还是原来的对象元素，如下所示：

```
>>> id(aList[0]), id(aList[1]), id(aList[2])
(1064072, 1191920, 1191896)
>>> id(thirdList[0]), id(thirdList[1]), id(thirdList[2])
(1064072, 1191920, 1191896)
>>> id(fourthList[0]), id(fourthList[1]), id(fourthList[2])
(1064072, 1191920, 1191896)
```

我们输出了这些对象的标识以确认我们的看法。为了获得对象的完全性拷贝或者叫做“深拷贝”（即创建一个新的包容器，它包容的引用指向的是原对象中的元素的全新拷贝（引用））我们需要使用copy.deepcopy()函数。如下所示：

```
>>> lastList = copy.deepcopy(aList)
>>> lastList
[[78, 'pyramid'], [84, 'vulture'], [81, 'eye']]
>>> id(lastList)
1193248
>>> id(lastList[0]), id(lastList[1]), id(lastList[2])
(1192280, 1193128, 1193104)
```

关于拷贝有几个需要牢牢记住的注意事项。首先，非包容器类型（比如数字、字符串以及其他最基本的对象如code（代码）、type（类型）和xrange对象等）是不能被拷贝的。序列类型的浅拷贝可以通过全切片获得。将在第8章介绍的映射类型需要使用字典拷贝方法来进行复制。最后，如果一个表列中的所有元素都是最基本类型的对象，对它们的深拷贝也是不能成功的。如果我们把上面例子里那个大列表中的小列表都换成表列，那么即使我们执行的是深拷贝操作，其结果也将是一个浅拷贝。

模块: copy

[CM]

我们刚才介绍的浅拷贝和深拷贝操作都可以在copy模块中找到。而该模块实际上也只有两个函数：copy()——创建一个浅拷贝；deepcopy()——创建一个深拷贝。

序列类型为数据提供了多种有序的存储形式。字符串是存储数据的一个常用形式，它可以显示给用户、被保存在磁盘上、通过网络传输，还可以是多个信息源的单个包容器。列表和表列提供的是包容器存储方式，它们允许对多个对象进行简单的处理和访问，不管那些对象是Python的数据类型还是用户定义的对象。单个对象或者成组的对象可以通过由顺序编号的下标偏移量取出的切片来访问。这些数据类型的结合在一起就提供了Python开发环境中灵活而又易于使用的存储工具。我们把这一章的内容总结为一个序列类型的操作符、内建函数和方法的表格，

即表6-12。

表6-12 序列类型的操作符、内建函数和方法

操作符、内建函数和方法	字符串	列表	表列
1) (列表的创建)			
()		•	•
append()		•	
capitalize()	•		
center()	•		
chr()	•		
cmp()	•	•	•
count()	•	•	
encode()	•		
endswith()	•		
expandtabs()	•		
extend()		•	
find()	•		
hex()	•		
index()	•	•	
insert()		•	
isdecimal()	•		
isdigit()	•		
islower()	•		
isnumeric()	•		
isspace()	•		
istitle()	•		
isupper()	•		
join()	•		
len()	•	•	•
list()	•	•	•
ljust()	•		
lower()	•		

(续)

操作符、内建函数和方法	字符串	列表	表列
lstrip()	•		
max()	•	•	•
min()	•	•	•
oct()	•		
ord()	•		
pop()		•	
raw_input()	•		
remove()		•	
replace()	•		
repr()	•	•	•
reverse()		•	
rfind()	•		
rindex()	•		
rjust()	•		
rstrip()	•		
sort()		•	
split()	•		
splitlines()	•		
startswith()	•		
str()	•	•	•
strip()	•		
swapcase()	•		
split()	•		
title()	•		
tuple()	•	•	•
type()	•	•	•
upper()	•		
zfill()	•		
.(attributes)	•	•	
[] (slice)	•	•	•

(续)

操作符、内建函数和方法	字符串	列表	表列
[:]	•	•	•
•	•	•	•
*	•	•	•
+	•	•	•
in	•	•	•
not in	•	•	•

6.20 练习

6-1 字符串。string字符串模块里是否有什么字符串方法或函数可以帮助我确定某个字符串是否是更大的一个字符串的组成部分？

6-2 字符串标识符。请修改程序示例6-1中的idcheck.py脚本程序，使它可以判定长度为1的标识符是否合法，以及判定一个标识符是否是一个关键字。这个练习的后半部分可以利用keyword模块（特别是其中的keyword.kwlist列表）来帮助解决。

6-3 排序。

a) 输入一个列表的数字，然后按照从大到小的顺序进行排序。

b) 输入一个列表的字符串，然后按照逆字母表顺序(字母的从最大到最小的顺序)进行排序。

6-4 算术计算。对前一章里的考试分数练习进行修改，使考试分数被输入到一个列表里。你的程序还应该计算出考试成绩的平均分。请参考练习2-9和5-3。

6-5 字符串。

a) 按照从头到尾的顺序依次显示一个字符串中的每一个字符，再按照从尾到头的顺序显示每一个字符。

b) 用扫描每个字符串的办法检查两个字符串是否一致（不使用比较操作符和cmp()内建函数）。附加题：在你的解决方案里加上字母的大小写检查。

c) 检查一个字符串是否是一个回文（即正着念和倒着念都是一样的）。附加题：增加代码，除了严格检查回文文字外，忽略其他所有的符号和空格。

d) 把一个字符串前后颠倒之后附加在原来的字符串后面，构成一个回文。

6-6 字符串。编写一段与string.strip()功能相当的代码，即把一个字符串的前导和尾缀空格都删除掉。（如果使用string.*strip()就失去这个练习的意义了。）

6-7 调试纠错。检查下面程序示例6-4（buggy.py）中的代码。

a) 阅读这段代码，说出这个程序是干什么用的。在每一个出现井字符（#）的地方加上适当的注释。运行这个程序。

b) 这个程序中有一个大缺陷，它无法处理输入为6、12、20、30的情况，更不要说一般的

偶数了。这个程序有什么问题？

c) 改正b) 中的缺陷。

程序示例6-4 有缺陷的程序 (buggy.py)

这是练习6-7用的程序清单。需要你来确定这个程序的功能、在“#”符号处添加注释、找出它的问题所在并且解决找出的问题

```

1  #!/usr/bin/env python
2
3  #
4  import string
5
6  #
7  num_str = raw_input('Enter a number: ')
8
9  #
10 num_num = string.atoi(num_str)
11
12 #
13 fac_list = range(1, num_num+1)
14 print "BEFORE:", `fac_list`
15
16 #
17 i = 0
18
19 #
20 while i < len(fac_list):
21
22     #
23     if num_num % fac_list[i] == 0:
24         del fac_list[i]
25
26     #
27     i = i + 1
28
29 #
30 print "AFTER:", `fac_list`

```

6-8 列表。给定一个整数值，返回这个数值中每个数字的英文写法。比如说，如果输入的是89，返回的结果应该是“eight nine”。附加题：返回这个数值的正确的英文写法，比如“eighty-nine”。这个练习使用的数值其范围定为0到1000之间。

6-9 转换。编写一个功能类似于练习6-8的解决方案的函数，其输入是一个总的分钟数，请把该数字转换为以小时和分钟表示的时间形式，要使小时数最大。

6-10 字符串。编写一个函数，它把一个字符串中的字母大小写反转后做为其返回值。比如说，如果输入的是“Mr. Ed”，返回的字符串就应该是“mR. eD”。

6-11 转换。

a) 编写一个程序，它把一个整数转换为一个用四组八进制数表示的因特网协议（Internet Protocol，简称IP）地址 WWW.XXX.YYY.ZZZ。

b) 修改你的程序，使它可以把IP地址转换为整数。

6-12 字符串。

a) 编写一个名为findchr()的函数，它的定义如下：

```
def findchr(string, char)
```

findchr()将在字符串string里查找字符char，返回的是字符串中第一次出现char字符的位置的下标；如果char没有出现在字符串string里，就返回-1。你可以使用string.find()或者string.index()函数或方法。

b) 再编写一个名为rfindchr()的函数，它返回的是字符串中最后一次出现char字符的位置的下标。这个函数与findchr()函数很相似，只不过它是从字符串的尾部开始进行查找的。

c) 编写第三个名为subchr()的函数，它的定义如下：

```
def subchr(string, origchar, newchar)
```

subchr()类似于findchr()，只是它会在找到origchar的时候用newchar字符替换掉它。它的返回值是修改后的字符串。

6-13 字符串。string模块包含有三个函数，它们是atoi()、atol()和atof()，分别用来把字符串转换为整数、长整数和浮点数。Python 1.5版本中的内建函数int()、long()和float()也能够完成同样的任务，该版本中还有一个complex()内建函数用来把字符串转换为复数。（在1.5以前的版本里，这些内建函数只能在数值类型之间相互转换。） [1.5]

string模块里从来没有实现过atoc()功能，而这就是你在这个练习中的任务。atoc()以一个字符串做为输入，它必须是一个复数的字符串表示形式，比如'-1.23e+4-5.67j'，返回的是与之对应的复数数值对象。你不能使用eval()函数，但允许使用complex()内建函数，使用complex()时必须严格遵守下面的语法：complex(real, imag)——其中的real和imag都是浮点数。complex()的使用方法请参考表6-4。

6-14 随机数。设计一个“包袱、剪子、锤”的猜拳游戏，这是一个儿童游戏。它的规则是：你和你的同伴同时选择出以下三种手势中的一种：包袱、剪子、和锤。赢家是按下面的规则确定的，三种手势环环相扣：a) 包袱包起锤子；b) 锤砸掉剪子；c) 剪子剪掉包袱。在你编写的这个计算机游戏中，用户输入他或她选择的手势，计算机随机地选择一种手势，你的程序判定输赢或者平局。请注意：要用最少的if语句完成这些算术上的判断。

6-15 转换。

a) 以某种标准的格式给出两个日期，比如MM/DD/YY或者DD/MM/YY，请计算出两个日期中间间隔的天数。

b) 给定某人的生日，请计算出这个人已经活过的天数，包括所有的闰年天数。

c) 根据练习b)中同样的信息，计算出距这个人的下一个生日还有多少天。

6-16 矩阵。实现两个M乘N矩阵的加法和乘法操作。

6-17 方法。编写一个名为my pop()的函数，让它具备类似于列表的pop()函数的功能。即以—一个列表为输入，从列表中删除最后一个对象后返回它。

第7章 字典数据类型

我们将在本章里讨论Python语言中唯一的映射类型——字典。我们将介绍可以用在字典上的各种操作符和内建函数。最后再介绍一些对字典进行处理的标准库模块。

7.1 字典简介

我们将要介绍的最后一个标准类型是字典，它也是Python语言中唯一的映射类型。字典是不可变的，它是另外一种容器类型，能够存储任意数目的Python对象，包括其他容器类型。字典与列表和表列等序列类型容器的不同之处在于数据存储和访问的方式。

序列类型只使用数值键字（从序列头部开始键字依次被编号为下标偏移量）。映射类型可以使用其他对象类型做为键字，其中字符串是最常用的。映射键字与序列键字不一样，它们通常（如果不是直接地）与被保存的数据相关联着。但是，因为对映射类型不再使用“顺序编号”的键字，我们看到的是一个没有顺序可言的数据集合。这并不会影响到我们的使用，因为映射类型不需要用一个数字值对容器中的数据进行索引就能获得想要的项。通过键字，你被直接“映射”到键值上去，这也就是“映射类型”的来历。把键字映射到相关联的键值去的最常见数据结构是“哈希表”。

编程提示：什么是哈希表？它们与字典有什么关系？

[CN]

序列类型以顺序编号的数值键字做为下标偏移量把数据以一个数组的形式保存起来。下标数字与被保存的数据值通常没有什么必然的联系。因此，也需要有一种方法能够以另外一个相互关联的值（比如字符串）为依据来保存数据。在日常生活里我们经常是这样做的。你把别人的电话号码按照姓名记在自己的地址簿里，你把事情根据日期和时间记在日历或预约簿里，等等。在这些例子里，每一个与数据相互关联的值就是你的键字。

哈希表就是符合我们刚才描述的那样一种数据结构。它们保存的每一条数据叫做一个键值，而做为查找依据的那个关联数据叫做一个键字。它们组合在一起就叫做“键字-键值对”。哈希表算法以键值为依据，对它进行一定的操作（它被称为一个哈希函数）后，依据计算的结果在数据结构中选定保存键值的位置。一个数据具体的保存位置取决于它的键字是什么。因为这种操作的随机性，哈希表中的键值是没有顺序可言的。你得到的是一个未经排序的数据集合。

你能获得的唯一一种顺序是通过键字实现的。你能够取出一个字典的键字来，它们会做为一个列表返回给你，你再调用列表的`sort()`方法对该数据集合进行排序。而这只是能够对键字进行的排序操作中的一种而已。无论如何，只要你觉得键字集合已经按照自己的意愿“排好了序”，就可以从字典里检索出与它们相关联的键值来。哈希表的执行性能通常都很好，因为只要你有了一个键字，查找操作通常都完成的相当迅速。对一个顺序访问的

数据结构来说，你必须匹配上正确的下标地址后才能存取键值。一般来说，性能要取决于所选用的哈希函数类型。

Python中的字典被实现为长度可变的哈希表。如果读者熟悉Perl语言，我们可以说字典与Perl语言中的关联数组或哈希表很相似。

我们现在开始认真研究一下Python的字典。字典数据项的语法是：key:value。此外，字典的数据项是括在花括号（{}）中间的。

1. 如何创建和赋值字典

创建字典的操作很简单，把一个字典赋值给一个变量即可，不必理会字典中是否有元素。如下所示：

```
>>> dict1 = {}
>>> dict2 = {'name': 'earth', 'port': 80}
>>> dict1, dict2
({}, {'port': 80, 'name': 'earth'})
```

2. 如何访问字典中的值

要想访问字典中的元素，需要使用我们已经很熟悉的方括号和键字来获取它的键值：

```
>>> dict2['name']
'earth'
>>>
>>> print 'host %s is running on port %d' % \
... (dict2['name'], dict2['port'])
host earth is running on port 80
```

字典dict1是空的，dict2中有两个数据项。dict2中的键字是'name'和'port'，它们的关联键值分别是'earth'和80。访问键值要通过键字来实现，正如刚才明确地使用'name'键字进行访问那样。

如果访问某个数据项时给出的键字在字典里不存在，就会看到一条出错信息：

```
>>> dict2['server']
Traceback (innermost last):
  File "<stdin>", line 1, in ?
KeyError: server
```

在这个例子里，我们想通过键字'server'来访问一个键值，而这个键字我们知道在上面的代码里是不存在的。检查一个字典里是否有某个特定键字的最佳办法是使用字典的has_key()方法。我们将在后面的内容里介绍与字典有关的所有方法。布尔类型的has_key()方法会在一个字典有那个键字的时候返回1，否则返回0。如下所示：

```
>>> dict2.has_key('server')
0
>>> dict2.has_key('name')
1
>>> dict2['name']
'earth'
```

如果has_key()方法返回的是1，就说明键字是存在的，再使用它的时候就不必担心会像前面那样收到KeyError错误了。我们来看看另外一个字典例子，它使用的键字不全是字符串：

```
>>> dict3 = {}
```

```
>>> dict3[1] = 'abc'
>>> dict3['1'] = 3.14159
>>> dict3[3.2] = 'xyz'
>>> dict3
{3.2: 'xyz', 1: 'abc', '1': 3.14159}
```

除了逐个添加每一个键字-键值对以外，我们还可以把dict3中的全部数据一次性输入进去，如下所示：

```
dict3 = {3.2: 'xyz', 1: 'abc', '1': 3.14159}
```

如果事先知道数据项都有哪些当然好，可以一次性把全部键字-键值对都输入进去。上面dict3例子的目的是向大家演示一下键字可以有多种类型和形式。如果我们遇到这样一个问题：某特定键值的键字能否被修改，我想你的回答很可能是“不”，对不对？

在执行期间不允许修改键字是非常合理的，你可以这样想：假设你用一个键字和键值对创建了一个字典。但在你程序的执行过程中，键字被修改了，可能换成另一个变量了。那么，当你再想用原来的键字检索数据值的时候，就会收到一个KeyError错误（因为键字改变了），你也因为不知道键字改变了而不知所措。因此，键字必须是不可变的，所以数字和字符串都不错，但列表和其他字典就不行了（7.5.2节论述了为什么键字必须是不可变的）。

3. 如何对字典进行修改

对字典的修改有以下几种情况：添加一个新数据项或新元素（即一对新的键字和键值）、修改一个现有的数据项、删除一个现有的数据项（下一节里有关于数据项删除操作的详细讲解）等，如下所示：

```
>>> dict2['name'] = 'venus'           # update existing entry
>>> dict2['port'] = 6969              # update existing entry
>>> dict2['arch'] = 'sunos5'          # add new entry
>>>
>>> print 'host %(name)s is running on port %(port)d' % dict2
host venus is running on port 6969
```

如果某个键字已经存在，那它的旧键值将被新键值覆盖掉。上面的print语句又给出了字符串格式操作符(%)的一种新用法，就是使用字典。用字典做print语句的参数可以缩短整个语句的长度，因为字典定义一次就行了；而用一个表列做参数却必须为它的每个元素进行多次定义。

用update()内建方法可以把一个字典完整的内容添加到另一个字典里面去。我们将在本章的7.4节里介绍这个方法。

4. 如何删除字典中的元素和字典本身

对整个字典进行的删除操作并不常见。比较常见的是删除字典中的某个元素或者是清空字典的所有内容。但如果你真的想“删除”整个字典，可以使用del语句（参见3.5.6节里的介绍）。下面是一些删除字典和字典元素的例子：

```
del dict1['name']           # remove entry with key 'name'
dict1.clear()              # remove all entries in dict1
del dict1                   # delete entire dictionary
```

编程提示：为什么不像有list.remove()那样有一个dict.remove()方法？

[CN]

你可能还记得要想从列表中删除一个数据项可以采用两种办法：使用del语句或使用list.remove()方法。这就引出一个问题：为什么列表有一个用来删除数据项的方法而字典就没有？回答很简单：从列表里删除一个元素需要两个步骤，先要找到数据项保存的下标（也可以说是键字），然后再执行del语句。remove()方法被编写为能够完成这两个步骤，让程序员只要用一个调用就可以完成操作。而对字典来说，你已经有键字了，不再需要查询步骤，只要调用一次del语句就能够达到目的。因此，编写一个删除数据项的字典方法只是提供了一个函数接口而已。

7.2 操作符

字典不支持合并和重复等序列操作，但有一个update()内建方法来把一个字典的内容添加到另一个字典里去。字典也没有“成员”操作符，但has_key()内建方法基本上起到了同样的作用。

7.2.1 标准类型操作符

对字典可以使用所有的标准类型操作符。这些操作符在第4章里介绍过，我们这里只给出一些对字典使用它们的例子：

```
>>> dict4 = { 'abc': 123 }
>>> dict5 = { 'abc': 456 }
>>> dict6 = { 'abc': 123, 98.6: 37 }
>>> dict7 = { 'xyz': 123 }
>>> dict4 < dict5
1
>>> (dict4 < dict6) and (dict4 < dict7)
1
>>> (dict5 < dict6) and (dict5 < dict7)
1
>>> dict6 < dict7
0
```

这些比较操作都是如何实现的呢？它们与列表和表列上的操作很相似，比数值和字符串上的操作略微复杂一些。对算法的详细说明请参考7.3.1节。

7.2.2 字典的键字检索操作符[]

字典唯一的一个专用操作符是键字检索操作符，它与序列类型的单元素切片操作符非常相似。

对序列类型来说，下标偏移量是访问序列中某个特定元素唯一要用到的参数。对字典来说，检索是通过键字来实现的，因此它与一个下标的作用是差不多的。键字检索操作符既可以用在字典的赋值操作里，也可以用在字典的键值检索操作里。如下所示：

```
dict[k] = v    # set value 'v' in dictionary with key 'k'
dict[k]        # lookup value in dictionary with key 'k'
```


7.3 内建函数

7.3.1 标准类型函数type()、str()和cmp()

type()内建函数用在字典上的时候，它给出的是与字典类型有关的一个对象。str()内建函数能够产生一个字典的可打印字符串表示形式。这些都是比较明显的。

在前面的三个章节里，我们依次介绍了cmp()内建函数对数值、字符串、列表和表列的执行情况。那么对字典又是怎样的情形呢？字典之间的比较基于这样一个算法：先比较两者的长度，然后是键字、最后是键值。在我们下面的例子里，我们创建了两个字典，然后把它们进行比较；再一点一点地修改这两个字典，演示这些修改对比较操作将产生怎样的影响。如下所示：

```
>>> dict1 = {}
>>> dict2 = { 'host': 'earth', 'port': 80 }
>>> cmp(dict1, dict2)
-1
>>> dict1['host'] = 'earth'
>>> cmp(dict1, dict2)
-1
```

在第一个比较操作中，因为dict2比dict1多了两个元素（2个数据项对0个数据项），所以dict1被判定为比较小。给dict1添加了一个元素之后，虽然添加的元素也是字典dict2中的一个，可比较操作的结果仍然判定它比较小（2对1）。

```
>>> dict1['port'] = 8080
>>> cmp(dict1, dict2)
1
>>> dict1['port'] = 80
>>> cmp(dict1, dict2)
0
```

我们在dict1里又添加了一个元素，现在两个字典的长度是一样的了，因此要比较它们的键字。在这个例子里，正好两组键字也都是一样的，所以要继续比较它们的键值。'host'键字的键值是一样的；在'port'键字处，因为dict1的键值比dict2的'port'键字的键值要大（8080对80），所以dict1被判定为比较大（译者注：这一段原文把dict1和dict2弄拧了）。当我们把dict1的'port'键字的键值设为与dict2的'port'键字的键值一样时，这两个字典就完全相等了：它们的长度一样，它们的键字一样，它们的键值也一样。因此，cmp()内建函数返回了一个0值。再看下面的例子：

```
>>> dict1['prot'] = 'tcp'
>>> cmp(dict1, dict2)
1
>>> dict2['prot'] = 'udp'
>>> cmp(dict1, dict2)
-1
```

再往任何一个字典里添加一个元素，它就立刻成为那个“比较大的”，这就是上面dict1的情况。往dict2里也添加一个键字-键值对，两者又需要重新进行比较：因为两个字典的长度一样，所以比较操作将检查它们的键字和键值。

```
>>> cdict = { 'fruits':1 }
```

```
>>> ddict = { 'fruits':1 }
>>> cmp(cdict, ddict)
0
>>> cdict['oranges'] = 0
>>> ddict['apples'] = 0
>>> cmp(cdict, ddict)
14
```

刚才这个最后的例子提醒我们cmp()也可以返回不是-1、0、1的值。算法是按照下面的顺序进行比较的:

1. 比较字典的长度

如果两个字典的长度不一样,那么对cmp(dict1, dict2)来说:如果dict1比较长,cmp()将返回一个正数;如果dict2比较长,cmp()将返回一个负数。换句话说,键字比较多的字典将被判定为是比較大的,即

```
len(dict1) > len(dict2) ⇒ dict1 > dict2
```

2. 比较字典的键字

如果两个字典的长度是一样的,就继续比较它们的键字。键字将按照key()方法返回的顺序进行检查和比较。(这里需要重点提醒大家的是:相同的键字将映射到哈希表中同一个位置去。)只要遇到两个键字不一样的地方,就将直接对它们进行比较:如果dict1中的第一个不同的键字比dict2中的第一个不同的键字大,cmp()就会返回一个正数。

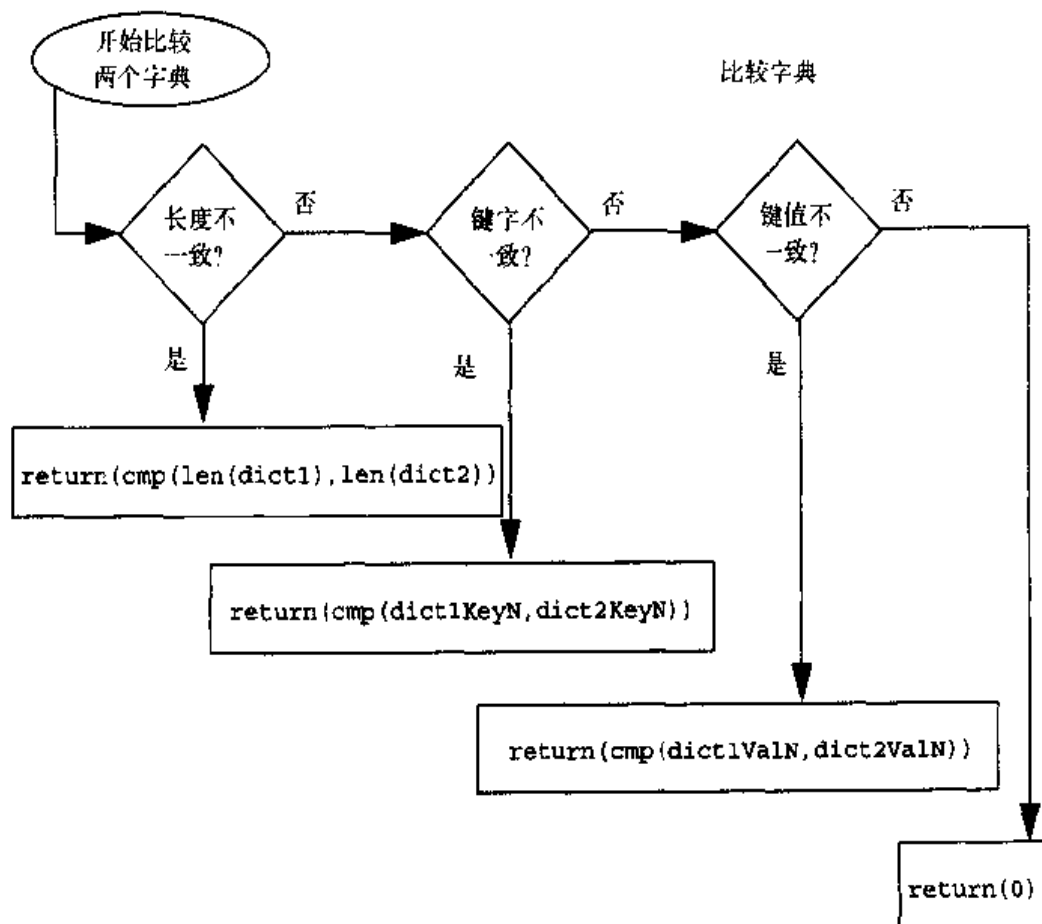


图7-1 字典是如何比较的

3. 比较字典的键值

如果两个字典的长度一样，它们的键字又完全相同，就继续比较两个字典中每一个键字对应的键值。只要遇到键值不匹配的键值，就直接比较这两个键值。在键字相同的情况下，如果dict1中的键值比dict2中的键值大，cmp()就会返回一个正数。

4. 精确匹配

如果到了这一步，两个字典已经是长度相同、键字相同，并且每一个键字对应的键值也相同了，而这样我们就可以说这两个字典达到了精确匹配的程度，返回值就将是一个0。

图7-1给出了我们刚才描述的字典比较算法的流程图。

7.3.2 映射类型函数len()

类似于序列类型的内建函数，映射类型的len()内建函数返回的是该类型对象中数据项的总个数，对字典来说就是键字-键值对的个数，如下所示：

```
>>> dict2 = { 'name': 'earth', 'port': 80 }
>>> dict2
{'port': 80, 'name': 'earth'}
>>> len(dict2)
2
```

我们前面曾经提到过，字典中的数据项是没有顺序可言的。我们可以从上面看出这一点：当查看dict2中的内容时，它的数据项是按照与它们被输入字典时相反的顺序列出来的。

7.4 内建方法

表7-1列出了字典对象的方法。clear()、copy()、get()和update()方法都是从Python 1.5版本开始新增加的。setdefault()是从Python 2.0版本开始出现的。

表7-1 字典类型的方法

字典方法	操 作
dict.clear ^① ()	删除字典dict中的全部元素
dict.copy ^② ()	返回字典dict的一个(浅 ^③)拷贝
dict.get(key, default = None) ^③	对键字key来说，如果key出现在字典里，返回它对应的键值；如果key没有出现在字典里，返回default的定义值（请注意：default的默认值是None）
dict.has_key(key)	如果key出现在字典dict里，返回1；否则返回0
dict.items()	返回字典dict的(key, value)（键字，键值）表列对的一个列表
dict.keys()	返回字典dict的键字列表
dict.setdefault(key, default = None) ^③	类似于get()，但在键字key没有出现在字典dict的情况下，它将设置 dict[key] = default
dict.update(dict2) ^③	把字典dict2的键字-键值对添加到字典dict中去
dict.values()	返回字典dict的键字列表

① 新出现于Python 1.5版本。

② 关于浅拷贝和深拷贝的详细资料请参考6.19节。

③ 新出现于Python 2.0版本。

我们在下面给出几个常用字典方法的使用示例:

```
>>> dict2 = { 'name': 'earth', 'port': 80 }
>>> dict2.has_key('name')
1
>>>
>>> dict2['name']
'earth'
>>>
>>> dict2.has_key('number')
0
```

`has_key()`方法是一个布尔函数, 它指出一个给定的键字对该方法正在操作的字典来说是否是合法的。试图访问一个不存在的键字将引发一个例外 (`KeyError`), 就象我们在本章开始部分的7.1节里看到的那样。映射类型不象序列类型那样支持`in`和`not in`操作符, 所以`has_key()`方法就是我们最好的选择了。

其他有用的字典方法完全集中在它们的键字和键值方面。其中包括返回字典的键字列表的`key()`、返回字典的键值列表的`value()`和返回字典的(键字, 键值)表列对以及列表的`item()`等。当你打算遍历一个字典键字或者键值(当然不要求什么特定的顺序)时, 这些方法就非常有用。如下所示:

```
>>> dict2.keys()
['port', 'name']
>>>
>>> dict2.values()
[80, 'earth']
>>>
>>> dict2.items()
[('port', 80), ('name', 'earth')]
>>>
>>> for eachKey in dict2.keys():
...     print 'dict2 key', eachKey, 'has value',
dict2[eachKey]
...
dict2 key port has value 80
dict2 key name has value earth
```

`keys()`方法返回的是一个字典的键字列表, 它与`for`循环联用时非常适合用来检索一个字典的键值。但因为它的数据项是没有经过排序的, 因此通常需要对它进行某种方式的排序操作。我们在下面还使用刚才的循环, 但在检索操作之前先(用列表的`sort()`方法)对键字进行排序。

```
>>> dict2Keys = dict2.keys()
>>> dict2Keys.sort()
>>> for eachKey in dict2Keys:
...     print 'dict2 key', eachKey, 'has value',
dict2[eachKey]
...
dict2 key name has value earth
dict2 key port has value 80
```

`update()`方法用来把一个字典的内容添加到另外一个字典中去。任何与新添加的数据重复的

数据项都将被覆盖掉；非现有的将被添加进来。一个字典中的全体数据项可以用`clear()`方法一次性全部删除。如下所示：

```
>>> dict2= { 'host':'earth', 'port':80 }
>>> dict3= { 'host':'venus', 'server':'http' }
>>> dict2.update(dict3)
>>> dict2
{'server': 'http', 'port': 80, 'host': 'venus'}
>>> dict3.clear()
>>> dict3
{}
```

`copy()`方法简单地返回一个原字典的复制品。需要注意的是这只是一个浅拷贝。关于浅拷贝和深拷贝的内容请参考6.19节。最后，`get()`方法类似于使用键字检索操作符(`[]`)的情况，但在一个键字不存在的情况下允许由你提供一个默认的返回值。如果键字不存在，并且又没有给出一个默认的返回值，就返回`None`。这比只使用键字检索操作符要灵活得多了，因为你不必再担心会在键字不存在的时候会引发一个例外。如下所示：

```
>>> dict4 = dict2.copy()
>>> dict4
{'server': 'http', 'port': 80, 'host': 'venus'}
>>> dict4.get('host')
'venus'
>>> dict4.get('xxx')
>>> type(dict4.get('xxx'))
<type 'None'>
>>> dict4.get('xxx', 'no such key')
'no such key'
```

Python 2.0引入了一个新的字典内建方法`setdefault()`，它的目的是缩短检查一个字典里是否有某个特定的键字的程序代码的长度。在许多情况下，如果一个键字存在，你就想要返回它所对应的键值；如果该字典里没有你正在查找的键字，你会希望能够设置一个默认值并返回那个默认值。这就是`setdefault()`方法所做的事情：

```
>>> myDict = { 'host': 'earth', 'port': 80 }
>>> myDict.keys()
['host', 'port']
>>> myDict.items()
[('host', 'earth'), ('port', 80)]
>>> myDict.setdefault('port', 8080)
80
>>> myDict.setdefault('prot', 'tcp')
'tcp'
>>> myDict.items()
[('prot', 'tcp'), ('host', 'earth'), ('port', 80)]
```

详细资料请参考“[What's New in 2.0](#)”（2.0版本的新功能）在线文档。它的URL地址可以在附录部分的在线资源小节里查到，本书所附的CD-ROM光盘上也有。

7.5 字典键字

对字典的键值没有什么限制，它们可以是任意的Python对象，既可以是标准对象，也可以是

用户定义的对象。但对键字来说就不同了。

7.5.1 不允许一个键字对应一个以上的数据项

规则之一是限制每个键字只能对应一个数据项。换句话说，同一个键字是不允许有多个键值的。（但键值可以是包容器对象，比如列表、表列和其他的字典等。）如果检测到键字“冲突”（意思是对同一个键字重复进行了多次赋值），留下来的是最后那个赋值。如下所示：

```
>>> dict1 = {'foo':789, 'foo': 'xyz'}
>>> dict1
{'foo': 'xyz'}
>>>
>>> dict1['foo'] = 123
>>> dict1
{'foo': 123}
```

Python根本就不对键字冲突进行检查，因为那样需要为保存每个键字-键值对耗费内存，因此也就不会产生一个错误了。在上面的例子里，键字'foo'在同一行里被赋值了两次，而Python对键字-键值对的处理是从左向右进行的。因此，数值789先被设置为键字'foo'的键值，但马上就被替换为字符串'xyz'。如果是给一个不存在的键字进行赋值，就会在字典里创建那个键字，键值也被添加到其中；而如果键字已经存在（即出现了冲突），就把它替换为当前的键值。在上面的例子里，键字'foo'的键值被替换了两次；到最后一个赋值语句时，字符串'xyz'又被替换为数值123。

7.5.2 键字必须是不可变的

我们已经在7.1节里讲过，大多数Python对象都可以被用做键字——只有可变类型的列表和字典是不允许的。换句话说，如果一个类型是通过它的值而不是通过它的标识进行比较的话，就不能被用做字典的键字。如果把一个可变类型用做键字，就会引发一个TypeError错误。如下所示：

```
>>> dict[[3]] = 14
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: unhashable type
```

键字为什么必须是不可变的呢？解释器用来计算数据的存放位置的哈希函数是以键字的值为依据的。如果键字是一个可变的对象，它的值就有可能发生变化；而如果键字变化了，哈希函数就会映射到另外一个位置去保存数据。如果真的出现这样的情况，哈希函数就不能可靠地保存和检索与之关联的键值了。之所以选择不可变对象做为键字，就是因为它们的值是不能被改变的。（请参考Python常见问题答疑FAQ中的问题6.18。）

我们知道数字和字符串可以被用做键字，那表列又怎么样呢？我们知道它们是不可变的，但在6.17.2节里，我们也提到过它们也并不是完全不可变的。最明显的例子就是我们可以修改做为表列元素的一个列表对象。因此，如果想表列做为合法的键字，就必须加上一条限制：表列只有在只包含数字和字符串等不可变参数时才能被用做合法的键字。

我们把这一章中关于字典的内容总结为一个程序（即程序示例7-1中的userpw.py），这个程

序在一个模拟的登录资料数据库系统里对用户名和口令字进行管理。这个脚本程序会在新用户给出了登录名和口令字的前提下接受他们。设立了一个“帐户”以后，老用户就可以在给出登录名和正确的口令字后返回程序。新用户不能用现有的登录名建立登录数据项。

程序示例7-1 字典示例 (userpw.py)

这个应用程序对通过一个登录名和一个口令字进入系统的一组用户进行管理。

设置好以后，现有用户可以在还记得住自己的登录名和口令字的情况下返回程序。新用户不能使用其他人的登录名建立新的登录数据项。

```

1  #!/usr/bin/env python
2
3  db = {}
4
5  def newuser():
6      prompt = 'login desired: '
7      while 1:
8          name = raw_input(prompt)
9          if db.has_key(name):
10             prompt = 'name taken, try another: '
11             continue
12         else:
13             break
14     pwd = raw_input('passwd: ')
15     db[name] = pwd
16
17 def olduser():
18     name = raw_input('login: ')
19     pwd = raw_input('passwd: ')
20     passwd = db.get(name)
21     if passwd == pwd:
22         pass
23     else:
24         print 'login incorrect'
25         return
26
27     print 'welcome back', name
28
29 def showmenu():
30     prompt = ""
31     (N)ew User Login
32     (E)xisting User Login
33     (Q)uit
34
35 Enter choice: ""
36
37     done = 0
38     while not done:
39
40         chosen = 0
41         while not chosen:
42             try:
43                 choice = raw_input(prompt)[0]
44             except (EOFError, KeyboardInterrupt):
45                 choice = 'q'
46             print '\nYou picked: [%s]' % choice
47             if choice not in 'neq':
48                 print 'invalid option, try again'
49             else:
50                 chosen = 1
51
52         if choice == 'q': done = 1

```

```

53         if choice == 'n': newuser()
54         if choice == 'e': olduser()
55
56 if __name__ == '__main__':
57     showmenu()

```

1~3行

在UNIX操作系统的启动行之后，我们用一个空白的用户数据库初始化这个程序。因为我们并不打算把数据保存起来，所以每次执行这个程序都会建立一个新的用户数据库。

5~15行

newuser()函数是向新用户提供服务的代码。它检查用户新输入的名字是否已经有人用过了，一旦验证了新名字的合法性，就提示用户输入自己的口令字（我们这个简单的程序没有使用加密编码），用户的口令字将被保存到字典里，他们的用户名就是键字。

17~27行

olduser()函数负责处理老用户。如果用户返回到这个程序时给出了正确的用户名和口令字，就会看到一条欢迎信息。否则就会通知用户他使用了非法的登录名并回到菜单去。我们并不想在这里用一个无限循环提示用户输入正确的口令字，因为用户可能会不小心输入了错误的菜单项。

29~54行

真正控制这个脚本程序的代码部分是这里的showmenu()函数。用户会看到一个友好的菜单。提示信息是用三引号给出的，因为整个提示信息要占用多个文本行，三引号的多行管理要比在一行里嵌入多个'\n'符号要容易得多。显示菜单之后，程序等待来自用户的合法输入并根据选中的菜单项进入相应的操作状态。我们在这里用到的try-except语句和上一章里的stack.py和queue.py程序示例中的作用是一样的（请参考6.14.1节）。

56~57行

这部分代码应该是我们比较熟悉的，它是在这个脚本程序被直接执行（而不是被导入）时只调用showmenu()启动这个应用程序用的。

下面是我们的脚本程序一次执行的情况：

```

% userpw.py

(N)ew User Login
(E)xisting User Login
(Q)uit

Enter choice: n

You picked: [n]
login desired: king arthur
passwd: grail

(N)ew User Login
(E)xisting User Login
(Q)uit

Enter choice: e

```



```

You picked: [e]
login: sir knight
passwd: flesh wound
login incorrect

(N)ew User Login
(E)xisting User Login
(Q)uit

Enter choice: e

You picked: [e]
login: king arthur
passwd: grail
welcome back king arthur

(N)ew User Login
(E)xisting User Login
(Q)uit

Enter choice: ^D
You picked: [q]

```

7.6 练习

7-1 字典方法。哪个字典方法可以用来把两个字典合并在一起？

7-2 字典的键字。我们知道字典的键值可以是任意的Python对象，但键字又怎么样呢？请试着把数字和字符串类型以外的其他对象用做键字。哪些能行，哪些不行？对那些不能用做键字的对象类型，你认为是什么原因使它们不能成功？

7-3 字典和列表方法。

- a) 创建一个字典，并且把它的键字按字母表顺序显示出来。
- b) 现在把该字典的键字和键值以键字的字母表顺序排好序后显示出来。
- c) 同练习b)，但这里是按键值的字母表顺序排好序。（请注意：对字典或哈希表来说，这样做一般没什么实际意义，因为大多数的访问操作和排序操作（如果有的话）都以键字为依据。这里只把它做为一个练习。）

7-4 创建字典。给定两个长度相同的列表，比如说是列表[1, 2, 3, ...]和列表['abc', 'def', 'ghi', ...]，用这两个列表中的所有数据构成一个字典，使它看起来是下面这样的形式：{1: 'abc', 2: 'def', 3: 'ghi', ...}。

7-5 userpw2.py。下面的问题都与程序示例7-1中管理登录名-口令字的键字-键值对的程序有关。

- a) 修改那个脚本程序，使它能够把记录上次登录时的日期与时间的时间标签（timestamp）和口令字一起保存起来。程序界面还应该像以前那样给出登录名和口令字输入提示并判定登录成功与否；如果登录成功，就修改与之对应的上次登录时的时间标签。如果本次登录与上次登录在时间上距离不超过四个小时，就通知用户“You already logged in at: <last_login_

timestamp>.”(你已经在<last_login_timestamp>登录过。)

b) 添加一个“主管”菜单,在其中要有以下两个选项:(1)删除一个用户;和(2)列出系统中所有用户及他们的口令字清单。

c) 口令字目前是没有加密的。请添加一段对口令字进行加密的代码(加密方法随意。请参考crypt、rotor或其他密码方面的模块)。

d) 给这个应用程序增加一种GUI操作界面,比如Tkinter。

7-6 列表和字典。建立一个粗略的股票资产数据库系统。至少要有四组数据,即股票代码、所持股数、购买价格和当前价格(你可以随意添加更多组数据)。用户一次要输入一支股票的四个数据,这四个数据构成一个列表。还要用一个总的列表把所有股票都包括进去。数据输入完毕后,提示用户选择一个数据项进行排序,把该数据项抽出来做为字典里的键字,其键值就是包含该键字的四个一组的列表。提醒大家:被选出来排序用的数据项必须是不会重复的键字,否则就会丢失股票数据;这是因为字典不允许一个键字有多个键值。你还可以随意进行各种计算并提供输出结果,比如盈亏率、当前帐面资产价值等。

7-7 颠倒字典中的键字和键值。用一个字典做输入,返回一个字典做输出。但前者的键值是后者的键字,前者的键字是后者的键值。

7-8 人力资源。用字典编写一个简单的对雇员姓名和雇员编号进行管理的应用程序。让用户输入一组姓名和雇员编号。你的程序应该提供(按姓名)排好序的输出,先显示雇员姓名,然后是雇员编号。附加题:增加一个功能,把输出按雇员编号排好序。

7-9 翻译。

a) 编写一个字符翻译程序(功能类似于UNIX中的tr命令)。我们将调用名为tr()的函数。它使用三个字符串做参数:源字符串、目的字符串、被处理字符串;语法定义如下:

```
def tr(srcstr, dststr, string)
```

srcstr包含的是你打算“翻译”的字符集合;dststr包含的是翻译后将要得到的字符集合;而string就是准备对之进行翻译的字符串了。举例来说:如果srcstr = 'abc', dststr = 'mno', string = 'abcdef',那么tr()的输出将是'mnodef'。注意这里的len(srcstr) == len(dststr)。在这个练习里,你可以使用chr()和ord()内建函数,但它们并不是解决这个问题所必不可少的。

b) 在这个函数里增加一个标志参数来处理大小写字母的翻译。

c) 改进你的解决方案,使它能够进行字符删除处理。srcstr字符串中不能够映射到dststr字符串中的字符都将被过滤掉。换句话说,这些字符没有映射到dststr字符串中的任何字符,因此就从函数返回的字符串里被过滤掉了。举例来说:如果srcstr = 'abcdef', dststr = 'mno', string = 'abcdefghi',那么tr()的输出将是'mnoghi'。注意这里的len(srcstr) >= len(dststr)。

7-10 加密。

a) 用一个练习中的思路编写一个“rot13”翻译器。“rot13”是一个古老而又简单的加密方法,它把字母表中的每个字母用其后第13个字母来代替。字母表中前半部分字母将被映射到后半部分,而后半部分字母将被映射到前半部分,大小写保持不变。举例来说:'a'将被替换为'n','X'将被替换为'K';数字和符号不进行翻译。

b) 在你的解决方案顶部再加上一个应用程序，让它提示用户输入准备加密的字符串（这个算法同时也可以对加密后的字符串进行解密），如下所示：

```
% rot13.py
Enter string to rot13: This is a short sentence.
Your string to en/decrypt was: [This is a short
sentence.].
The rot13 string is: [Guvf vf n fubeg fragrapr.].
%
% rot13.py
Enter string to rot13: Guvf vf n fubeg fragrapr.
Your string to en/decrypt was: [Guvf vf n fubeg
fragrapr.].
The rot13 string is: [This is a short sentence.].
```

第8章 条件语句和循环语句

本章的主要焦点是Python语言中的条件语句、循环语句，以及与它们有关的组件。我们将认真研究if、while、for和与它们配合使用的else、elif、break、continue、pass等语句。

8.1 if语句

Python语言中的if语句看起来很熟悉；它由三个主要部分组成：关键字本身、一个进行真值测试的表达式、表达式测试结果为非零或真时将要执行的代码子句。if语句的语法如下所示：

```
if expression :  
    expr_true_suite
```

if语句的子句expr_true_suite只有在前面的条件表达式结果为布尔真值（true）时才执行；否则，程序将跳到紧接在该子句后面的那条语句继续执行。

8.1.1 多重条件表达式

可以用布尔操作符and、or和not构成多重条件表达式或者对if语句中的条件表达式进行取反操作，如下所示：

```
if not warn and (system_load >= 10):  
    print "WARNING: losing resources"  
    warn = warn + 1
```

8.1.2 单语句子句

如果一个if语句的子句只由一条语句构成，可以把它和表达式部分写在同一行上，如下所示：

```
if (make_hard_copy == 1) : send_data_to_printer()
```

像上面这样的单行if语句从语法角度来看是合法的；可虽然这样做比较方便，但它可能会使你的代码比较难于阅读，因此我建议你把这单个的子句缩进在下一行上。另外一个理由是：如果需要在今后给这个子句添加新的程序行，那还是要把这一行移到下一行上去。

8.2 else语句

Python语言和其他程序设计语言一样准备了一个能够与if语句一起配对使用的else语句。当if语句中的条件表达式的结果是一个布尔假值的时候，就执行跟在它后面的代码段。它的语法如下所示：

```
if expression:  
    expr_true_suite  
else:  
    expr_false_suite
```

下面是一个用法示例:

```
if passwd == user.passwd:
    ret_str = "password accepted"
    id = user.id
    valid = 1
else:
    ret_str = "invalid password entered... try again!"
    valid = 0
```

避免悬垂的else

Python语言使用缩进而不是花括号来区分代码段, 这样不仅可以增加代码的正确性, 还可以帮助避免出现看起来语法正确、可实际上有潜在问题的情况, 其中就包括“著名”的“悬垂的else”问题, 这是一个程序设计中可能经常出现的问题。

我们在这里给出一段C语言代码做为示例 (在K&R和其他程序设计教材里也用它做例子):

```
/* dangling-else in C */
if (balance > 0.00)
    if ((balance - amt) > min_bal) && (atm_cashout() == 1)
        printf("Here's your cash; please take all bills.\n");
else
    printf("Your balance is zero or negative.\n");
```

我们的问题是: 这个else到底是属于哪个if的? C语言规则规定else属于与它最接近的那个if。因此, 我们例子中的else语句虽然和外层的if语句对齐, 可实际上属于那个内层的if语句, 因为C编译器并不受语句行前导空格的影响。这样做的结果就是即使你在自己的帐户上还有钱, 可那条烦人的 (也是错误的) 消息还是会告诉你说帐户余额已经是零或负数。

因为这个例子比较简单, 所以解决这个问题也并不太难。可类似的程序框架结构中如果有很多节代码, 就会导致很不容易清除的隐患。Python语言采用的缩进办法不能够完全阻止你把车开下悬崖, 但它可以使你远离危险。同样的例子, 用Python语言来编写会得到两种选择 (注意: 只有其中一种是正确的):

```
if (balance > 0.00):
    if ((balance - amt) > min_bal) and (atm_cashout() == 1):
        print "here's your cash; please take all bills."
else:
    print "your balance is zero or negative"
```

或者

```
if (balance > 0.00):
    if ((balance - amt) > min_bal) and (atm_cashout() == 1):
        print "here's your cash; please take all bills."
    else:
        print "your balance is zero or negative"
```

Python语言采用的缩进办法强制代码实现正确的对齐, 让程序设计人员能够对一个else应该属于哪个if做出明确的选择。这种在选择方面的限制减少了二义性, 因此Python语言能够让你从一开始就编写出正确的代码来。在Python语言里很少会出现悬垂的else问题。

8.3 elif语句

elif是Python语言中的else-if语句。它允许人们对一系列条件表达式进行检查，并在某个表达式为真的情况下执行相应的代码段。类似于else，elif也是可选的。但一个if语句只允许跟有一个else子句，而一个if语句可以有任意个数的elif语句。如下所示：

```
if expression1:
    expr1_true_suite
elif expression2:
    expr2_true_suite
    :
elif expressionN:
    exprN_true_suite
else:
    none_of_the_above_suite
```

目前，Python语言还不像其他语言那样支持switch或case语句。程序中使用大量的if-elif语句并不影响Python的易阅读性。

```
if (user.cmd == 'create'):
    action = "create item"
    valid = 1

elif (user.cmd == 'delete'):
    action = 'delete item'
    valid = 1

elif (user.cmd == 'quit'):
    action = 'quit item'
    valid = 1

else:
    action = "invalid choice... try again!"
    valid = 0
```

我们可以利用Python语言中的for语句很精巧地实现其他语言中switch或case语句的效果。人们可以用for来循环处理每一个可能的“情况”并采取相应的措施，从而“模仿”程序开关的作用。（请参考8.5.3节内容。）

8.4 while语句

Python语言中的while是我们将在本章详细讨论的第一个循环语句。事实上，它是一个条件循环语句。在if语句里，表达式为真将导致执行一次if语句的子句；而while语句的子句将连续不断地在一个循环里反复执行直到条件不再满足为止。

8.4.1 一般语法

下面是while循环的一般语法：

```
while expression :
    suite_to_repeat
```

`while`循环的`suite_to_repeat`子句将连续不断地在一个循环里反复执行，直到条件表达式`expression`的结果变为布尔假值为止。这种形式的循环机制通常被用在计数循环的情况中，比如下一节的示例中。

8.4.2 计数循环

```
count = 0
while (count < 9):
    print 'the index is:', count
    count = count + 1
```

这个例子的子句由`print`和递增语句组成，在计数器`count`不小于9之前它们会反复不断地执行。在每一次循环遍历中，计数器`count`的当前值将被显示出来，然后按1递增。如果把这段代码带到Python的解释器里，代码的输入过程和执行过程会是下面的样子：

```
>>> count = 0
>>> while (count < 9):
...     print 'the index is:', count
...     count = count + 1
...
the index is: 0
the index is: 1
the index is: 2
the index is: 3
the index is: 4
the index is: 5
the index is: 6
the index is: 7
the index is: 8
```

8.4.3 无限循环

使用`while`循环的时候要谨慎，因为其条件表达式可能永远也得不出一个假值。在这种情况下，我们得到的就是一个永远也不会到头的循环。这些“无限”循环并不一定是坏事情，许多客户-服务器系统中的通信“服务器”就正是以这种方式工作的。这一切取决于是否需要该循环持续不断地运行；如果不需要，这个循环有没有到头的可能性；换句话说，它的条件表达式能否得出一个假值？

```
while 1:
    handle, indata = wait_for_client_connect()
    outdata = process_request(indata)
    ack_result_to_client(handle, outdata)
```

举例来说，上面这段代码就被明确地设置为永不停止，因为数值1永远也不能变成布尔假值。这段服务器代码的主要作用是守候并等待来自客户的连接，这种连接一般都来自网络上的链接。这些客户会发送出能够被服务器正确理解和处理的操作请求。该请求被处理之后，一个返回值或数据会发送到客户端，再由客户决定是结束这次连接还是继续发送另外一个请求。从服务器的角度看，在完成了这个客户请求的工作后，它返回到循环的开始去等待下一个客户的到来。在讲述网络和Web程序设计的有关章节（第16和19章）里读者可以找到更多关于客户-服务器方

面的计算处理。

8.4.4 单语句子句

类似于if语句的语法，如果一个while语句的子句只由一条语句构成，那就可以把它和while关键字部分写在同一行上。下面是一个只有一行的while语句的例子：

```
while not ready : ready = is_data_ready()
```

8.5 for语句

Python语言中另外一个循环机制是以for语句的面目出现在我们面前的。Python语言中的for语句与其他主流的第三代程序设计语言（third-generation languages，简称3GLs）如C、Fortran或者Pascal等中的不同，它更像某种脚本语言中的遍历型foreach循环。

8.5.1 一般语法

遍历型循环会依次对某个集合中的全体元素进行处理，处理完所有的数据项之后终止循环。Python语言中的for语句只能对序列进行遍历处理，如下面的一般语法所示：

```
for iter_var in sequence :
    suite_to_repeat
```

序列sequence将被遍历处理；在每一次循环里，iter_var遍历变量都会被设置为该序列的当前元素，然后会在suite_to_repeat部分对它进行某种形式的处理。

8.5.2 与序列类型一起使用

在这一小节里，我们将看到for循环是如何处理各种序列类型的。示例将包括字符串、列表和表列类型。先看看下面的例子：

```
>>> for eachLetter in 'Names':
...     print 'current letter:', eachLetter
...
current letter: N
current letter: a
current letter: m
current letter: e
current letter: s
```

在对一个字符串进行遍历的时候，遍历变量永远都是由一个单个字符（长度为1的字符串）构成的。这种构造的用处可能不太必要。如果是要在一个字符串里查找某个字符，程序员通常会使用in操作符来测试该字符是否是该字符串的一个成员，也可以使用某个字符串模块函数或字符串方法对子字符串进行检查。

查找单个字符的操作比较有用的地方之一是在一个应用程序里用一个for循环调试纠错一个序列，此时print语句预期显示的是字符串或者整个的对象；如果你看到的是单个的字符，这通常表示你接收到的是一个单个的字符而非一个对象序列。

遍历一个序列有两种基本的办法，它们是：

1. 用序列的数据项实现遍历

```
>>> nameList ['Walter', 'Nicole', 'Steven', 'Henry']
>>> for eachName in nameList:
...     print eachName, "Lim"
...
Walter Lim
Nicole Lim
Steven Lim
Henry Lim
```

在上面的例子里，我们遍历了一个列表。在每次遍历中，eachName遍历变量中包含的都是那次循环遍历中将要处理的列表元素

2. 用序列的下标实现遍历

另外一种遍历形式是通过序列本身每一个数据项的下标偏移值进行遍历：

```
>>> nameList = ['Shirley', 'Terry', 'Joe', 'Heather', 'Lucy']
>>> for nameIndex in range(len(nameList)):
...     print "Liu, ", nameList[nameIndex]
...
Liu, Shirley
Liu, Terry
Liu, Joe
Liu, Heather
Liu, Lucy
```

在这里，我们没有遍历序列元素本身而是通过列表的下标实现了遍历。

我们利用给出表列中元素总数的len()内建函数和range()内建函数（我们将在下面对它做进一步讲解）完成了对序列实际上的遍历：

```
>>> len(nameList)
5
>>> range(len(nameList))
[0, 1, 2, 3, 4]
```

range()内建函数使我们获得了一个由下标值构成的列表，再通过这个列表用nameIndex做遍历变量进行遍历；同时，我们用切片/下标操作符（[]）取出了与之对应的序列元素。

对执行效率有研究的读者肯定会发现通过序列的数据项实现的遍历要胜过通过下标实现的遍历。如果不是这样，倒是一个值得思考的问题了。（请参考练习8-13。）

8.5.3 switch/case语句的代理

我们早在8.3节就已经介绍过if-elif-else结构，并且指出Python语言不支持switch或case语句。在许多情况下，一大串if-elif-else语句可以用一个for循环来代替，后者包含着序列中需要被遍历的“case”项。我们在下面给出对8.3节中的例子进行了修改之后的版本，把全部elif语句替换为一个for循环，如下所示：

```
for cmd in ('add', 'delete', 'quit'):
    if cmd == user.cmd:
        action = cmd + " item"
        valid = 1
        break
else:
```

```

action = "invalid choice... try again!"
valid = 0

```

读者可能很高兴看到在Python语言里也有解决缺乏switch或case语句的措施, 但你是否意识到使用列表会给程序员更大的施展空间呢? 在其他程序设计语言里, 一个case语句中的元素都是常数, 并且是代码中不可改变的部分。但在使用列表的Python语言环境里, 这些元素不仅可以是变量, 它们还可以是动态的, 能够在运行过程中加以改动!

最后, 读者可能会对出现在代码尾部的else语句感到惊讶。是的, else语句可以和for循环一起使用。在这种情况下, else子句将在for循环全部执行完毕后被执行。8.9节里有更多关于else的内容。

8.5.4 range()内建函数

我们在前面介绍Python语言中的for循环时说它是一个遍历型的循环机制。Python语言里还有一个能帮助我们以传统的伪条件设置形式使用for语句的工具, 即从一个数开始计数到另一个数, 到达后一个数时或者某些条件不再满足时就立刻退出for循环。

内建函数range()可以把foreach类型的for循环转换为我们比较熟悉的传统形式, 比如从0开始计数到10, 或者从10开始以5为递增量计数到100。

1. range()的完整语法

range()在Python里有两种用法。它的完整语法要求给出两个或者全部三个整数参数, 如下所示:

```
range(start, end, step=1)
```

range()会返回一个列表, 对列表中任一元素k来说, $start \leq k < end$, 并且k是从start开始以步长step递增到end的。步长step不能是0, 否则会引起一个错误。

```

>>> range(2, 19, 3)
[2, 5, 8, 11, 14, 17]

```

如果省略了步长, 只给出了两个参数, 那么步长step的默认取值将是1。

```

>>> range(3, 7)
[3, 4, 5, 6]

```

我们来看看解释器环境中使用的一个例子:

```

>>> for eachVal in range(2, 19, 3):
...     print "value is:", eachVal
...
value is: 2
value is: 5
value is: 8
value is: 11
value is: 14
value is: 17

```

这个for循环现在是从2“计数”到19, 递增的步长是3。如果读者熟悉C语言, 就会注意到range()的参数和C语言for循环中的循环变量是一致的:

```
/* equivalent loop in C */
```

```
for (eachVal = 2; eachVal < 19; i += 3) {  
    printf("value is: %d\n", eachVal);  
}
```

虽然这里看起来像是一个条件循环（检查是否满足`eachVal < 19`的条件），但我们知道实际情况是`range()`根据这个条件表达式生成了一个符合要求的列表，然后由Python的`for`语句通过这个列表控制循环次数。

2. `range()`的简化语法

`range()`还有一个简化的格式，它只使用一个或两个整数参数，如下所示：

```
range(start=0, end)
```

这个简化格式只使用两个参数，它实际上就是只带两个参数的长格式`range()`——它的步长因为缺省而被认为是1。如果只给出了一个数字，就把`start`的值默认为0，`range()`返回的将从0开始、以1递增到`end`值的一个数字列表，如下所示：

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

我们现在把它带到Python解释器里并加上`for`和`print`语句，得到的结果是：

```
>>> for count in range(5):  
...     print count  
...  
0  
1  
2  
3  
4
```

执行`range()`产生其列表结果后，上面的条件表达式就是下面代码中的样子：

```
>>> for count in [0, 1, 2, 3, 4]:  
...     print count
```

编程提示：为什么`range()`的语法不止一个？

[CN]

现在你已经知道`range()`函数的两个语法了，那么，为什么不把这两个东西像下面这样合并起来呢？

```
range(start=0, end, step=1) # invalid
```

这个语法在只有一个或者全部三个参数时是能正常执行的，但如果只给出了两个参数就不行了。两个参数之所以是非法的原因是有`step`的时候也必须有`start`；换句话说，你给出的是`end`和`step`这两个参数，但它们会被（错误地）解释为`start`和`end`，产生了二义性。

3. 用于内存不足情况下的`xrange()`函数

`xrange()`和`range()`很相似，它们之间的区别主要是：在需要生成一个特别大的列表的时候，`xrange()`用起来更方便一些，因为它不需要真的在内存里产生那个大列表的拷贝。这个内建函数专门用在`for`循环里，在`for`循环以外也用不着它。此外，因为整个列表不在内存里，所以它的执行性能肯定要差一些。

至此我们已经介绍完Python语言里所有的循环语句了，现在来看看经常和循环语句用在一起的几个辅助命令，其中包括终止循环的break语句和立即开始下一循环的continue语句。

8.6 break语句

Python中的break语句的作用是终止当前的循环，从下一个语句开始继续执行程序，这和C语言中break命令的作用是一样的。break的常见用法是在某些外部条件被触发（它通常是用一个if语句来测试的）时立即退出当前的循环操作。break语句可以用在while和for循环中。

```
count = num / 2
while count > 0:
    if (num % count == 0):
        print count, 'is the largest factor of', num
        break
    count = count - 1
```

这段代码的作用是找出给定数字num的最大因子。我们遍历所有可能会是num因子的数字，用count做循环变量，按照递减的顺序剔除所有不能整除num的数字。这样，第一个能够整除num的数字就是我们要找的最大因子，而一旦找到了这个最大因子数，就不再需要继续循环，此时就可以用break终止循环。

```
phone2remove = '555-1212'
for eachPhone in phoneList:
    if eachPhone == phone2remove:
        print "found", phone2remove, '... deleting'
        deleteFromPhoneDB(phone2remove)
        break
```

这里的break语句用来中断对列表的循环处理。它的目的是找出列表中的目标元素，从数据库里删除它，然后退出循环。

8.7 continue语句

编程提示：continue语句

[CN]

在使用提供有continue语句的Python、C、Java或者其他结构化的程序设计语言时，某些程序员新手会有这样一个错误的概念，那就是传统的continue语句“将立刻开始循环的下一个遍历操作”。虽然这看上去是一个明显的程序动作，但我们要在这里澄清这个有些错误的看法。遇到continue语句后并不会立即开始循环的下一个遍历操作，这个continue语句会终止或者说放弃当前循环遍历操作中其余的语句，回到循环的顶部。

如果是在一个条件循环中，在开始循环的下一个遍历操作之前要先检查条件表达式是否仍然成立；只有在表达式仍然成立的前提下，下一个遍历操作才能开始执行。类似地，如果是在一个遍历循环中，就需要先确认是否还有用来遍历的参数；只有在确认了还有遍历参数的前提下才能开始下一个遍历操作。

Python语言中的continue语句与其他高级语言中传统的continue相比没有什么不同。这个continue语句可以用在while和for循环中。while循环是一个条件型循环，for循环是一个遍历型循

环，因此遇到continue语句后是否能够开始循环的下一个遍历操作还要看（上面编程注释中提到的）要求是否得到了满足；如果条件不满足，循环会正常结束。

```
valid = 0
count = 3
while count > 0:
    input = raw_input("enter password")
    # check for valid passwd
    for eachPasswd in passwdList:
        if input == eachPasswd:
            valid = 1
            break
    if not valid:      # (or valid == 0)
        print "invalid input"
        count = count - 1
        continue
    else:
        break
```

上面的例子里用到了while、for、if、break和continue语句，它的作用是检查用户输入的合法性。用户有三次机会输入正确的口令字；如果三次都失败了，valid变量会保持为一个代表假值的0不变，后面的程序代码将采取适当的行动对这种情况进行处理。

8.8 pass语句

Python语言里有一个C语言里没有的pass语句。Python语言不使用花括号来分隔代码段，但有时候却需要对代码段进行分隔。Python语言不像C语言那样有用来表示“什么也不做”的结束括号或者单个的分号。如果想在Python程序里的适当位置插入一条代表代码子段或子句段的语句，你是找不到的，而这样就会引发一个错误。此时，就需要使用pass语句，这是一个什么也不做的语句——它与汇编语言中的NOP（即“No OPeration”（没有操作））操作是完全一样的。从程序设计风格和程序看法的角度看，这个pass语句也很有用，你可以把它放在代码肯定会执行到、但目前还没有编写出来的位置上（如下例所示）：

```
def foo_func() :
    pass

或者

if user_choice == 'do_calc':
    pass
else:
    pass
```

这段代码在开发和调式阶段很有用，因为你想在编写代码的时候先把程序的结构安排好，并且暂时不想把它和已经完成的其他代码部分掺和在一起。如果你想表达什么也不想做的概念，pass语句就是一个好东西。

pass语句另外一个常见的用途是用在例外处理过程中，这一部分内容将在第10章论述；如果你想在一个错误发生后对它进行追踪，但如果它不是什么致命的错误又不必采取什么行动（比

如你只想在发生某个错误后记录一下有关的事件, 或者只是想随便执行一个什么操作), 就可以用上这条pass语句。

8.9 else语句之二

在C语言(以及其他程序设计语言)里, 你是不会在条件语句范围以外的地方看到else语句的, 但Python又一次出人意料地为while和for循环配上了else语句。它们的工作情况是怎样的呢? 当else语句和循环一起使用时, 它将在该循环完全执行完毕后被执行, 这就意味着它们不会因break而被跳过去。

在while语句里使用else的一个常见示例是用来找出一个数的最大因子。通过把else语句和while循环一起使用的办法, 我们在下面实现了一个能够完成这项任务的函数。程序示例8-1(maxFact.py)中的showMaxFactor()函数就把else语句用做while循环的组成部分了。

程序示例8-1 while-else循环示例(maxFact.py)

这个程序能够给出10到20之间的一个数字的最大因子。如果这个数字是一个素数, 这个脚本程序也会指示出来。

```
1  #!/usr/bin/env python
2
3  def showMaxFactor(num):
4      count = num / 2
5      while count > 1:
6          if (num % count == 0):
7              print 'largest factor of %d is %d' % \
8                  (num, count)
9              break
10         count = count - 1
11     else:
12         print num, "is prime"
13
14 for eachNum in range(10, 21):
15     showMaxFactor(eachNum)
```

循环从showMaxFactor()函数的第三行开始, 从给定数字的一半开始递减计算(先检查该数字能否被2整除, 如果可以, count中的数值就是那个最大的因子)。每次循环都从count中减去1(第10行)直到找到一个整除数为止(第6~9行)。如果循环递减为1的时候还没有找出一个整除数, 原来的那个数字必然是一个素数。第11、12行上的else语句负责处理素数的情况。第14、15行处是这个程序的主部分, 它调用showMaxFactor()函数对10到20之间的数字进行计算处理。

运行我们的程序会产生如下所示的输出结果:

```
largest factor of 10 is 5
11 is prime
largest factor of 12 is 6
13 is prime
largest factor of 14 is 7
largest factor of 15 is 5
largest factor of 16 is 8
17 is prime
largest factor of 18 is 9
```

```
19 is prime
largest factor of 20 is 10
```

类似地，for循环也可以有一个扫尾的else语句。它的执行情况和while循环中的是完全一样的。那个else子句会在for循环正常退出（不通过break语句）后立即执行。我们已经在8.5.3节里看到过一个这样的例子了。

表8-1总结了这些辅助语句都可以用在哪些条件或者循环语句中。

表8-1 循环语句和条件语句的辅助语句

辅助语句	循环语句和条件语句		
	if	while	for
elif	•		
else	•	•	•
break		•	•
continue		•	•
pass^a	•	•	•

① pass可以用在需要一个子句的任何地方（还包括elif、else、class、def、try、except、finally等）。

8.10 练习

8-1 条件语句。请看下面的代码：

```
# statement A
if x > 0:
    # statement B
    pass

elif x < 0:
    # statement C
    pass

else:
    # statement D
    pass

# statement E
```

- 如果 $x < 0$ ，上面的哪个语句（A、B、C、D、E）将被执行？
- 如果 $x == 0$ ，上面的哪个语句将被执行？
- 如果 $x > 0$ ，上面的哪个语句将被执行？

8-2 循环。编写一个程序，让用户输入三个数字：(f)rom、(t)o、和(i)ncrement。以i为步长从f计数到t，包括f和t。举例来说：如果输入的是 $f == 2$ 、 $t == 24$ 、 $i == 4$ ，程序将输出2、6、10、14、18、22。

8-3 range()。如果我们想生成下面这些列表，需要在range()内建函数里使用哪些参数？

a) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

b) [3, 6, 9, 12, 15, 18]

c) [-20, 200, 420, 640, 860]

8-4 素数。我们在本章里已经给出了一些代码来确定一个数字的最大因子或者它是否是一个素数。请把有关代码转换为一个名为isprime()的布尔函数。它的输入是一个整数，如果这个数字是一个素数，函数的返回值将是1；否则，返回值将是0。

8-5 约数。编写一个名为getfactors()的函数，它以一个整数作为参数，返回的是一个包括1和它本身在内的该整数所有约数的列表。

8-6 素数因子。以刚才练习中的isprime()和getfactors()为基础编写一个函数，它以一个整数作为输入，返回的是一个该整数的素数因子的列表。这个过程叫做求素数因子，它输出的那些因子乘在一起的时候应该得出原来的数字。请注意：里边可能会有重复的元素。比如说，如果你输入的是20，其输出应该是[2, 2, 5]。

8-7 完全数。完全数是这样的数字：它的约数（不包括它本身）加在一起等于它本身。举例来说：6的约数是1、2、3，因为1+2+3等于6，所以它（6）被认为是一个完全数。编写一个名为isperfect()的函数，它以一个整数为输入，如果这个数字是一个完全数，函数的返回值将是1；否则，返回值将是0。

8-8 阶乘。一个数的阶乘被定义为从1开始到该数字之间所有数字的乘积。数字N的阶乘被简写为“N!”， $N! = \text{factorial}(N) = 1 * 2 * 3 * \dots * (N-2) * (N-1) * N$ 。因此， $4! = 1 * 2 * 3 * 4$ 。编写一个函数，给定数字N后返回N!的值。

8-9 菲波那契数列。菲波那契数列是“1, 1, 2, 3, 5, 8, 13, 21, ...”等等。即数列中的下一个值是数列中前两个值的和。编写一个输出菲波那契数列的前30个数的函数。

8-10 文字处理。统计一句话里面元音、辅音和单词（以空格分隔）的个数。忽略特殊的元音和辅音情况，如“h”、“y”、和“qu”等。

8-11 文字处理。编写一个程序，让用户以“Last Name, First Name”——即名、逗号、姓——的格式输入一串人名。编写一个对输入进行处理的函数，让它在用户以错误的顺序——比如“First Name Last Name”——输入姓名时对这些错误加以纠正，并且通知用户。这个函数还要对输入出错的次数进行统计。用户把人名输入完毕后，对名单进行排序，最后以“Last Name, First Name”的顺序把排序后的名单显示出来。

输入输出示例（你不一定要完全按照这里的样子做）：

```
% nametrack.py
Enter total number of names: 5

Please enter name 0: Smith, Joe
Please enter name 1: Mary Wong
>> Wrong format... should be Last, First.
>> You have done this 1 time(s) already. Fixing input...
Please enter name 2: Hamilton, Gerald
Please enter name 3: Royce, Linda
Please enter name 4: Winston Salem
>> Wrong format... should be Last, First.
>> You have done this 2 time(s) already. Fixing input...
```



```
The sorted list (by last name) is:
Hamilton, Gerald
Royce, Linda
Salem, Winston
Smith, Joe
Wong, Mary
```

8-12 (整数) 位操作。编写一个程序, 它在由用户输入开始和结束数字后给出一个下面这样的表格, 分别显示出两个数字之间所有数字的十进制、二进制、八进制和十六进制表示形式。如果字符是可打印的ASCII字符, 也要把它们打印出来; 如果不是可打印ASCII字符, 就略掉ASCII那一栏的标题。请参考下面的输入输出格式:

SAMPLE OUTPUT 1

```
-----
Enter begin value: 9
Enter end value: 18
DEC    BIN    OCT    HEX
-----
  9    01001    11    9
 10    01010    12    a
 11    01011    13    b
 12    01100    14    c
 13    01101    15    d
 14    01110    16    e
 15    01111    17    f
 16    10000    20   10
 17    10001    21   11
 18    10010    22   12
```

SAMPLE OUTPUT 2

```
-----
Enter begin value: 26
Enter end value: 41
DEC    BIN    OCT    HEX    ASCII
-----
 26    011010    32    1a
 27    011011    33    1b
 28    011100    34    1c
 29    011101    35    1d
 30    011110    36    1e
 31    011111    37    1f
 32    100000    40    20
 33    100001    41    21    !
 34    100010    42    22    "
 35    100011    43    23    #
 36    100100    44    24    $
 37    100101    45    25    %
 38    100110    46    26    &
 39    100111    47    27    '
 40    101000    50    28    (
 41    101001    51    29    )
```

8-13 程序执行性能。在8.5.2节里, 我们介绍了两种遍历序列的方法: (1) 按序列数据项遍历, (2) 通过序列下标遍历。在该小节的末尾我们指出后一种方法对长序列的执行性能不佳 (在我自己的系统上测试时, 性能下降了两倍还多, 差了83%)。你对这种情况怎么看, 它的原因是什么?

第9章 文件和输入/输出操作

本章将深入介绍文件的使用方法和相关的Python输入/输出功能。我们将介绍文件对象（它的内建函数以及它的内建方法和属性），讲述标准文件，讨论文件系统的访问方法，论述文件的执行情况，并且简单涉及永久性存储和标准库中与文件有关的模块。

9.1 文件对象

文件对象不仅可以用来访问普通的磁盘文件，还可以用来访问任何符合“文件”抽象概念的其他类型。只要有了正确的“钩钩”，就可以通过类似于对文件进行访问的接口以访问普通文件的方式访问其他对象。

`open()`内建函数（请参考下一小节）返回一个文件对象，对该文件进行的所有后续操作都要用到它。有大量其他的函数也能够返回一个文件对象或者一个类似于文件的对象。出现这种情况的一个主要原因是人们在大量使用输入/输出数据结构时更喜欢坚持使用一种比较统一常见的操作接口，在程序行为和程序实现等方面能够保持些一致性。像UNIX这样的操作系统甚至把文件作为计算机通信中基础和体系结构方面的操作接口。不要忘记，文件只是一连串的字节序列而已。只要是需要传输数据的地方，就会涉及到某种形式排序的字节流，不管那个字节流只是一个字节还是一些数据块。

9.2 文件的内建函数

`open()`内建函数可以说是打开文件这扇大门的钥匙，它提供了初始化文件输入/输出操作的基本手段。如果成功地打开了一个文件，`open()`将返回一个文件对象，否则引发一个错误。如果它打开文件时失败，Python会产生一个IOError例外——我们将在下一章讨论对错误和例外的处理。`open()`内建函数的基本语法是：

```
file_object = open(file_name, access_mode='r', buffering=-1)
```

`file_name`是包含着的将要打开的文件名字的字符串，它可以是一个相对或者绝对/完整路径名。可选变量`access_mode`也是一个字符串，它是由指示文件打开方式的一串标志构成的。通常，在打开文件的时候可以把它的打开方式设置为“r”、“w”和“a”，这三个字母分别代表读、写和追加的意思。

以“r”方式打开的文件必须是已经存在的。以“w”方式打开的文件如果已经存在，其长度会先被截短为0字节，然后再（重新）创建。以“a”方式打开文件的目的是为了向文件中写入数据。如果该文件已经存在，文件（写）操作的初始位置将被设置为该文件的文件尾；如果该文件还不存在，就会像“w”方式那样创建该文件。如果你是一名C语言程序员，就会知道这些也是C语言库函数`fopen()`打开文件时使用的方式。

`fopen()`支持的打开文件的其他方式也能够用在Python语言的`open()`函数里。其中包括指示进

行读写方式访问的“+”和指示进行二进制方式访问的“b”。关于“b”方式有一点需要说明：对所有与POSIX兼容的UNIX系统（包括Linux）来说，“b”已经过时了，因为它们把包括文本文件在内的所有文件都当做是“二进制”文件。下面是从Linux的fopen()函数的使用手册中摘录的一段话，Python语言的open()函数就是从它衍生出来的：

指示文件打开方式的字符串中也可以包含字母“b”，但它只能作为最后一个字母，或者被夹在上面描述的由两个字符构成的字符串的字符的中间。这样做的目的是为了严格地满足与ANSI C3.159-1989（即“ANSI C”）标准的兼容性，并没有什么实际上的作用；“b”在所有与兼容POSIX的系统上都将忽略，包括Linux。（其他系统可能会区别对待文本文件和二进制文件，但如果你确实需要对一个二进制文件进行I/O操作，或者你的程序可能会被移植到非UNIX环境中去时，加上一个“b”会是一个好主意。）

读者可以在表9-1里找到一个关于文件访问方式的完整清单，包括“b”的用法——如果读者选择使用它的话。如果在打开文件时没有给出access_mode，它将自动默认为“r”。

另外一个可选择的参数buffering用来指明访问该文件时将要采用的数据缓冲方式。其中，0表示不使用缓冲；1表示只缓冲一行数据；任何大于1的数字表示对I/O进行缓冲，并把缓冲区长度设置为给出的数字；如果没有给出这个值或者给出了一个负数值，就表示将使用系统缺省的数据缓冲机制，即对tty串行类型的设备使用行缓冲，对其余设备使用正常缓冲。buffering的值在正常情况下是不必给出来的，使用系统缺省的方式就可以了。

表9-1 文件对象的访问方式

文件访问方式	操 作
r	以读方式打开
w	以写方式打开（如有必要，先进行截短）
a	以写方式打开（从EOF开始，如有必要，先创建它）
r+	以读和写方式打开
w+	以读和写方式打开（请参考上面的“w”）
a+	以读和写方式打开（请参考上面的“a”）
rb	以二进制读方式打开
wb	以二进制写方式打开（请参考上面的“w”）
ab	以二进制追加方式打开（请参考上面的“a”）
rb+	以二进制读和写方式打开（请参考上面的“r+”）
wb+	以二进制读和写方式打开（请参考上面的“w+”）
ab+	以二进制读和写方式打开（请参考上面的“a+”）

下面是几个打开文件的操作示例：

```
fp = open('/etc/motd')      #open file for read
fp = open('test', 'w')     #open file for write
fp = open('data', 'r+')    #open file for read/write
fp = open('c:\io.sys', 'rb') #open binary file for read
```

9.3 文件的内建方法

`open()`顺利执行并返回一个文件对象后，其他所有的文件后续访问操作都要用到那个“文件句柄”。文件的方法可以分为四大类，它们分别是：输入、输出、文件内移动和其他杂项。文件的所有方法可以在表9-3中找到。我们现在来依次介绍这几个大类。

9.3.1 输入

`read()`方法用来把字节直接读到一个字符串里，它会根据有关参数尽量读取最多个数的字节。如果没有给出读操作的字符串长度参数，其缺省值被设置为-1，表示读到文件尾。而`readline()`方法是读入已打开文件的一行数据（读入一个换行符前的所有字节）。换行符将保留在返回字符串中。`readlines()`方法与`readline()`方法很相似，但它是一次性把所有剩余的数据行读到字符串里去，返回的是一个包含着已读入数据的字符串的列表。`readinto()`方法把指定个数的字节读到一个“writable buffer”（可写缓冲区）类型的对象里去，这个类型的对象是由不被支持的`buffer()`内建函数所返回的。（因为`buffer()`不被支持，所以`readinto()`也不被支持。）

9.3.2 输出

`write()`内建方法的功能正好与`read()`和`readline()`相反。它以一个包含一行或者多行文本数据或字节块的字符串为参数，把它写到一个文件中去。`writelines()`像`readlines()`那样对一个列表进行操作，但它是把一些字符串写到文件里去。各行之间不插入换行符；如果需要加上换行符，就必须在调用`writelines()`之前把它们插入到每一行的末尾去。

这在Python 2.0里是很容易用一个列表操作实现的，如下所示： [2.0]

```
>>> output=['1stline', '2ndline', 'the end']
>>> [x + '\n' for x in output]
['1stline\n', '2ndline\n', 'the end\n']
```

请注意，不存在“`writeline()`”方法，因为它相当于调用`write()`对一个以换行符结尾的单行字符串进行操作。

9.3.3 文件内移动

`seek()`方法（功能类似C语言中的`fseek()`函数）的作用是在文件内部把文件指针移动到其他的位置。在以字节为单位给出位移量的时候还要加上一个叫做出发点`whence`的“相对偏移量”（relative offset）。如果出发点的值是0，就表示位移量从文件头开始计算（从文件头开始计算的位移量也叫做“绝对位移”（absolute offset））；如果出发点的值是1，就表示位移量从文件的当前位置开始计算；如果出发点的值是2，就表示位移量从文件尾开始计算。如果读者在使用C语言进行编程时使用过`fseek()`，就会知道出发点0、1、2分别对应着常数`SEEK_SET`、`SEEK_CUR`和`SEEK_END`。打开文件进行读写访问操作时经常会用到`seek()`方法。

`tell()`是对`seek()`的一个补充；它告诉你文件指针的当前位置——该位置是以字节为单位从文件头开始计算的。

9.3.4 其他

`close()`方法的作用是结束对一个文件的访问并关闭它。如果一个文件对象的引用线索减少为零(0), Python的废弃回收例程也会自动关闭这个文件。出现这种情况的一种可能是: 某个文件本来只有一个引用线索, 比如说是`fp = open()`, 但在明确关闭原来的文件之前`fp`又被另外赋值了一个文件对象。为了保持良好的程序设计风格, 建议大家在重新赋值为另外一个文件对象之前要先关闭原来的文件。

`fileno()`方法返回的是一个打开文件的文件描述符; 它是一个整数值参数, 主要用在`os`模块里那些底层的操作中, 比如`flush()`方法。`isatty()`是一个布尔型的内建方法, 如果文件是一个串行类的设备就返回1; 否则返回0。`truncate()`方法把文件长度截短为0字节或者给定的字节。

9.3.5 其他各种文件方法

我们现在来看看我们的第一个文件例子, 它来自第2章, 如下所示:

```
filename = raw_input('Enter file name: ')
file = open(filename, 'r')
allLines = file.readlines()
file.close()
for eachLine in allLines:
    print eachLine,
```

我们曾经介绍过这个程序。与大多数标准的文件访问方法相比, 它的不同之处在于它是在读取完全部的数据行之后才开始依次显示各行内容。很明显, 如果文件很大, 这个方法是不好的。如果是对大文件进行操作, 最好还是回到“摸着石头过河”的老路上来, 每次读入一行, 显示一行:

```
filename = raw_input('Enter file name: ')
file = open(filename, 'r')
done = 0
while not done:
    aLine = file.readline()
    if aLine != "":
        print aLine,
    else:
        done = 1
file.close()
```

在这个例子里, 我们并不知道什么时候才能到达文件的末尾, 因此我们使用了一个布尔标志`done`, 它在初始化状态时被设置为假(0)。当我们到达文件尾的时候, 要把这个标志重新设置为真(1), `while`循环就会因此而结束。我们把读取所有数据行的`readlines()`修改为`readline()`, 后者每次只读入一行。`readline()`在到达文件尾的时候会返回一个空行。因此, 如果没有遇到空行, 就把该行内容显示在屏幕上。

我想读者此时会迫不及待地问一个问题: “如果我的文件里真的有一个空白行该怎么办? Python会认为到达了文件的末尾而停止吗?”答案当然是不。文件中的空白行并不会返回为一个空行。大家应该还记得, 在每一行的末尾还有一个或者多个分隔字符, 因此“空白行”至少

会有一个换行符或者你系统使用的其他符号。所以，即使你的文件里真有一行是“空白的”，读入的行也不是空的，这就意味着在到达你文件的文件尾之前你的程序实际上是不会停止的。

编程提示：行分隔符和文件系统的其他差异

[CN]

操作系统之间的差异之一是它们的文件系统支持的行分隔字符是不同的。在UNIX上，行分隔字符是一个换行符（\n）；在Macintosh机器上，它是一个回车字符（\r）；而DOS和Windows同时使用这两个字符（\r\n）。请读者检查自己的操作系统使用的是哪一种行分隔字符。

其他差异包括文件路径名分隔符（UNIX使用“/”；DOS和Windows使用“\”；而Macintosh使用“:”），它是用来分隔文件路径名与当前子目录和父目录的记号。

当我们准备编写需要运行在这三种计算机平台上的应用程序时，就不得不考虑这些差异之处（如果想让应用程序支持更多的体系结构和操作系统，差异之处可能会更多）。但幸运的是Python的os模块的设计者已经替我们想到了这些问题。os模块里有五个很有用的属性参数，它们列在表9-2中。

表9-2 有助于多平台开发的os模块属性

os模块属性	说 明
linesep	用来分隔文件中数据行的字符串
sep	用来分隔文件路径名成分的字符串
pathsep	用来分隔一组文件路径名的字符串
curdir	当前工作子目录的字符串名字
pardir	（当前工作子目录的）父目录字符串名字

不管你使用的是哪一种计算机平台，只要你导入了os模块，这些变量就会被设置为正确的值。烦恼没有了。

还要提醒大家的是：放在print语句末尾的逗号会取消通常由print语句加在输出内容末尾的换行符。这样做的原因是文本文件中的每一行已经包含有一个换行符了。readline()和readlines()两个函数不对文本行里的空白字符做任何处理（请参考本章的练习）。因此，如果省略掉逗号，显示出来的文本文件的行间距将是两行，其中一个换行符是输入时带来的，另外一个换行符是print语句添加的。

在继续学习下一小节之前，我们来看两个示例。第一个例子的重点是输出到文件（不是输入）；第二个例子有输入输出，也有用seek()和tell()方法进行的文件定位：

```
filename = raw_input('Enter file name: ')
file = open(filename, 'w')
done = 0
while not done:
    aLine = raw_input("Enter a line ('.' to quit): ")
    if aLine != ".":
        file.write(aLine + '\n')
    else:
        done = 1
file.close()
```

这段代码从功能上来说正好与上一个例子相反：它不是每次读入一行显示一行，而是由用户输入一行文本，再把它写到文件里去。因为`raw_input()`不保留用户输入中的换行符，所以调用`write()`方法时要记得加上一个换行符。又因为从键盘上很难输入一个文件尾字符，所以在我们的程序里用一个句号（.）来做为文件尾字符：当用户单独输入一个句号时，就表示结束输入并关闭文件。

最后一个例子打开一个文件进行读写，创建了一个文件片段（可能是截短了一个现有的文件）。在向文件中写入数据之后，我们用`seek()`方法在文件内部移动，还用`tell()`方法显示了文件内移动的过程。

```
>>> f = open('/tmp/x', 'w+')
>>> f.tell()
0
>>> f.write('test line 1\n')      # add 12-char string [0-11]
>>> f.tell()
12
>>> f.write('test line 2\n')      # add 12-char string [12-23]
>>> f.tell()                      # tell us current file location (end)
24
>>> f.seek(-12, 1)                # move back 12 bytes
>>> f.tell()                      # to beginning of line 2
12
>>> f.readline()
'test line 2\n'
>>> f.seek(0, 0)                  # move back to beginning
>>> f.readline()
'test line 1\n'
>>> f.tell()                      # back to line 2 again
12
>>> f.readline()
'test line 2\n'
>>> f.tell()                      # at the end again
24
>>> f.close()                    # close file
```

表9-3列出了文件对象的全部内建方法。

表9-3 用于文件对象的方法

文件对象方法	操 作
<code>file.close()</code>	关闭文件file
<code>file.fileno()</code>	返回一个整数值，它是file的文件描述符（FD）
<code>file.flush()</code>	消除文件file的内部缓冲区
<code>file.isatty()</code>	如果file是一个串行类设备，返回1；否则返回0
<code>file.read(size = -1)</code>	把整个文件或size个字节长的文件数据读入一个字符串，返回值就是这个字符串
<code>file.readinto(buf, size)^a</code>	把size字节的文件数据读入一个缓冲区buf
<code>file.readline()</code>	从file中读入并返回一行数据（包括尾缀的\n字符）
<code>file.readlines()</code>	把file中全部的数据行读入到一个列表中后返回（包括所有尾缀的\n字符）
<code>file.seek(off, whence)</code>	在文件file内从出发点whence（0=文件头、1=当前位置、2=文件尾）开始移动off个字节
<code>file.tell()</code>	返回文件file里的当前位置

(续)

文件对象方法	操 作
<code>file.truncate(size = 0)</code>	把文件file截短为0字节或size字节
<code>file.write(str)</code>	把字符串str写入文件file里去
<code>file.writelines(list)</code>	把字符串的列表list写入文件file里去

① 从Python 1.5.2版本开始引入的不被支持方法（文件类对象的其他实现里不包括这个方法）。

9.4 文件的内建属性

除了方法以外，文件对象还有一些数据属性。这些属性容纳着与其对应文件有关的辅助性数据，其中包括文件名（`file.name`）、文件打开方式（`file.mode`）、文件是否已被关闭（`file.closed`）以及一个指示在用`print`语句显示下一个数据项之前是否需要加上一个额外的空格字符的标志（`file.softspace`）。表9-4给出了这些属性及其简单描述。

表9-4 文件对象的属性

文件对象属性	说 明
<code>file.closed</code>	1表示文件file已经被关闭；否则为0
<code>file.mode</code>	文件file被打开后的访问方式
<code>file.name</code>	文件file的名字
<code>file.softspace</code>	0表示需要在 <code>print</code> 语句里明确地加上空格；否则为1。程序员很少会用到它——通常只用于内部用途

9.5 标准文件

只要你的程序一开始运行，通常就会有三个标准文件可以被随时使用了，它们是：标准输入（通常就是键盘）、标准输出（到监视器或显示器的缓冲输出）和标准错误（到屏幕的非缓冲输出）——这里所说的“缓冲”输出和“非缓冲”输出指的是`open()`函数的第三个参数。这些文件沿用C语言中的用法，被分别命名为`stdin`、`stdout`和`stderr`。当我们说“这些文件可以在你的程序里随时使用”时，意思是这些文件已经预先被打开了，只要知道它们的文件句柄就可以随时访问这些文件。

Python通过`sys`模块向你提供这些文件的句柄。导入`sys`模块以后，就可以用`sys.stdin`、`sys.stdout`和`sys.stderr`来访问这几个文件。`print`语句通常是输出到`sys.stdout`；而内建函数`raw_input()`则通常是从`sys.stdin`那里接受输入数据的。

我们通过“Hello World!”程序再来看看使用`print/ raw_input()`和直接使用文件名两种情况之间的相似与差异之处：

print

```
print 'Hello World!'
```

sys.stdout.write()

```
import sys
sys.stdout.write('Hello World!' + '\n')
```


注意要在sys.stdout的write()方法中明确地添加上换行符；但在下面的输入示例里就不用这么做，因为在sys.stdin上执行的readline()会保留换行符。raw_input()不保留换行符，所有我们要在相应的print语句里加上它自己的换行符。如下所示：

```
raw_input()

aString = raw_input('Enter a string: ')
print aString

sys.stdin.readline()

import sys
sys.stdout.write('Enter a string: ')
aString = sys.stdin.readline()
sys.stdout.write(aString)
```

9.6 命令行参数

sys模块还通过sys.argv属性提供了对命令行参数的访问。命令行参数是调用某个脚本程序时脚本程序名以外的其他参数。这些参数之所以有这样的名称，其原因是：从历史沿革来看，在一个基于文本的环境里（比如UNIX操作系统的shell环境或者DOS的命令状态），这些参数都是和程序的名字一起在命令行上被输入的。但在IDE或者GUI环境中可能就不会是这样了，大多数IDE环境都另外提供一个用来输入“命令行参数”的窗口；可这些参数最终还是会象从命令行上直接运行那样被传递到应用程序中去。

熟悉C语言的读者可能会问：“argc哪儿去了？”字符串argc和argv分别代表“参数个数”（argument count）和“参数向量”（argument vector）。变量argv包含着一个由从命令行上输入的各个参数组成的字符串数组；而变量argc包含着输入的参数的个数。在Python语言中，argc的值其实就是sys.argv列表中数据项的个数，而该列表的第一个数据项sys.argv[0]永远是程序的名字。总结如下：

- sys.argv是命令行参数的列表。
- len(sys.argv)是命令行参数的个数（也就是sys argc）。

我们把下而这几行代码写到一个名为argv.py的测试程序中去：

```
import sys

print 'you entered', len(sys.argv), 'arguments...'
print 'they were:', str(sys.argv)
```

下面是这个脚本程序的一次运行和输出的情况：

```
% argv.py 76 tales 85 hawk
you entered 5 arguments...
they were: ['argv.py', '76', 'tales', '85', 'hawk']
```

命令行参数有用吗？UNIX操作系统中的命令通常都是这样一些程序：它们把程序的输入、所完成的函数功能和程序的输出都看做是一个数据流。这个数据通常还会被直接输入到下一个程序，在完成了一些其他类型处理后，再把新的输出送往再下面的一个程序，如此延伸下去。各个程序的输出一般是不保存的，这样可以节约大量的磁盘空间。各个程序的输出通常是“经

管道输送”到下一个程序，并且成为下一个程序的输入。这种效果是通过在命令行上提供数据或者通过标准输入提供数据而实现的。当一个程序把自己的输出显示出来或者送往标准输出文件时，其内容就会出现在屏幕上——除非该程序也“经管道连接”到下一个程序；如果真是这样，它的标准输出文件实际上就是下一个程序的标准输入文件。我想你现在应该明白了吧！

命令行参数使程序员或系统管理员在启动一个程序的时候能够有几种不同的程序行为方面的选择。在大多数情况下，这些执行操作都发生在午夜，并且是做为不需人工干预的批处理任务来运行的。命令行参数和程序选项使这种处理功能成为可能。因为有计算机在夜里有空闲，并且有大量需要处理的工作，所以就必要在这些昂贵的“计算器”的后台运行程序。

Python语言提供了一个getopt模块帮助大家分析命令行上的参数和选项。

9.7 文件系统

对文件系统的访问大多通过Python的os模块来实现。这个模块是到操作系统的功能和到Python所提供的服务的基本接口。os模块实际上只是加载到计算机中的那个“真正”模块的前端，而那个“真正”的模块很明显要依赖于具体的操作系统。这个“真正”的模块可能是以下几种中的一个：posix（适用于UNIX操作系统）、nt（适用于Windows系统）、mac（适用于Macintosh系统）、dos（适用于DOS系统）和os2（适用于OS/2操作系统）等。用户当然不必直接导入这几种模块；只要导入os就会加载上正确的模块，底层进行的工作是用户看不到的。根据你系统支持的功能的不同，你可能无法访问一些能够用在其他系统上的属性。

除了对进程进行管理和对程序运行环境进行维护之外，os模块还负责处理大部分文件系统方面的操作，应用程序的开发人员经常要用到这些操作。这些功能包括删除和重新命名文件、巡查目录树、管理文件的访问权限等。表9-5里列出了一些由os模块提供的常见的文件或目录操作。

如果需要完成某些特定的路径名操作，还有另外一个模块可供选用，它就是可以通过os模块访问的os.path模块。这个模块里包括主要用于管理和操作文件路径名成分、获取文件或子目录信息、进行文件路径查询等方面的函数。表9-6列出了os.path模块中比较常见的几个函数。

有了这两个模块，对不同计算机平台和操作系统上的文件系统的访问就能够保持一致性。程序示例9-1（ospathex.py）中的程序演示了os和os.path模块中部分函数的用法。

表9-5 os模块中与文件/子目录访问有关的函数

os模块中的文件/子目录函数	操 作
文件处理	
remove()/unlink()	删除文件
rename()	重命名文件
*stat()	返回文件的统计数据
symlink()	创建符号链接
utime()	刷新时间标签
目录/文件夹	
chdir()	切换当前工作目录
listdir()	列出目录中的文件清单
getcwd()	返回当前工作目录的路径名

(续)

os模块中的文件/子目录函数	操 作
mkdir()/makedirs()	创建目录 (一个或多个)
rmdir()/removedirs()	删除目录 (一个或多个)
访问/存取权限 (只适用于UNIX或Windows)	
access()	检验存取权限状态 (只适用于UNIX)
chmod()	修改存取权限状态 (适用于UNIX和Windows)
umask()	设置缺省的存取权限状态 (适用于UNIX和Windows)

① 包括stat()、lstat()和xstat()。

表9-6 os.path模块的路径名访问函数

os.path路径名函数	操 作
分隔	
basename()	去掉目录路径, 返回文件名
dirname()	去掉文件名, 返回目录路径
join()	把各个成分组合为一个路径名
split()	返回(dirname(), basename())表列
splitdrive()	返回(drivename, pathname)表列 (驱动器名, 路径名)
splitext()	返回(filename, extension)表列 (文件名, 文件后缀)
信息	
getatime()	返回文件上一次被访问的时间
getmtime()	返回文件上一次被修改的时间
getsize()	返回文件的长度 (以字节为单位)
查询	
exists()	路径名 (文件或子目录) 存在吗?
isdir()	路径名存在并且是一个目录吗?
isfile()	路径名存在并且是一个文件吗?
islink()	路径名存在并且是一个符号链接吗?
samefile()	两个路径名指向的是同一个文件吗?

程序示例9-1 os和os.path模块用法示例 (ospathex.py)

这段代码练习使用一些os和os.path模块中的功能。它建立一个测试文件;

在里面放上少量的数据; 重命名这个文件; 输出显示文件的内容。还进行了一些其他辅助性的文件操作, 主要是一些目录树巡查和文件路径名处理方面的操作内容。

```

1  #!/usr/bin/env python
2
3  import os
4  for tmpdir in ('/tmp', 'c:/windows/temp'):
5      if os.path.isdir(tmpdir):
6          break
7  else:
8      print 'no temp directory available'
9      tmpdir = ''
10
11 if tmpdir:
12     os.chdir(tmpdir)
13     cwd = os.getcwd()

```

```

14     print '*** current temporary directory'
15     print cwd
16
17     print '*** creating example directory...'
18     os.mkdir('example')
19     os.chdir('example')
20     cwd = os.getcwd()
21     print '*** new working directory:'
22     print cwd
23     print '*** original directory listing:'
24     print os.listdir(cwd)
25
26     print '*** creating test file...'
27     file = open('test', 'w')
28     file.write('foo\n')
29     file.write('bar\n')
30     file.close()
31     print '*** updated directory listing:'
32     print os.listdir(cwd)
33
34     print '*** renaming 'test' to 'filetest.txt'
35     os.rename('test', 'filetest.txt')
36     print '*** updated directory listing:'
37     print os.listdir(cwd)
38
39     path = os.path.join(cwd, os.listdir(cwd)[0])
40     print '*** full file pathname'
41     print path
42
43     print '*** (pathname, basename) =='
44     print os.path.split(path)
45     print '*** (filename, extension) =='
46     print os.path.splitext(os.path.basename(path))
47
48     print '*** displaying file contents:'
49     file = open(path)
50     allLines = file.readlines()
51     file.close()
52     for eachLine in allLines:
53         print eachLine,
54
55     print '*** deleting test file'
56     os.remove(path)
57     print '*** updated directory listing:'
58     print os.listdir(cwd)
59     os.chdir(os.pardir)
60     print '*** deleting test directory'
61     os.rmdir('example')
62     print '*** DONE'

```

在一个UNIX平台上运行这个程序得到的输出结果如下所示:

```

% ospathex.py
*** current temporary directory
/tmp
*** creating example directory...
*** new working directory:
/tmp/example
*** original directory listing:
[]
*** creating test file...
*** updated directory listing:
['test']
*** renaming 'test' to 'filetest.txt'

```

```

*** updated directory listing:
['filetest.txt']
*** full file pathname:
/tmp/example/filetest.txt
*** (pathname, basename) ==
('/tmp/example', 'filetest.txt')
*** (filename, extension) ==
('filetest', '.txt')
*** displaying file contents:
foo
bar
*** deleting test file
*** updated directory listing:
[]
*** deleting test directory
*** DONE

```

在一个DSO窗口里运行这个示例程序得到的输出结果和刚才的差不多:

```

C:\>python ospathex.py
*** current temporary directory
c:\windows\temp
*** creating example directory...
*** new working directory:
c:\windows\temp\example
*** original directory listing:
[]
*** creating test file...
*** updated directory listing:
['test']
*** renaming 'test' to 'filetest.txt'
*** updated directory listing:
['filetest.txt']
*** full file pathname:
c:\windows\temp\example\filetest.txt
*** (pathname, basename) ==
('c:\windows\temp\example', 'filetest.txt')
*** (filename, extension) ==
('filetest', '.txt')
*** displaying file contents:
foo
bar
*** deleting test file
*** updated directory listing:
[]
*** deleting test directory
*** DONE

```

在这里就不向大家做逐行的讲解了, 我们把这部分留给大家做为练习。但我们将带大家一起去看看下面那些交互式操作的例子(包括错误), 给大家一个一步一步地执行脚本程序的感觉。我们把代码分解成几个小段, 然后依次进行讲解。

```

>>> import os
>>> os.path.isdir('/tmp')
1
>>> os.chdir('/tmp')
>>> cwd = os.getcwd()
>>> cwd

```

```
 '/tmp'
```

代码的第一个部分导入了os模块（同时也包含了os.path模块）。我们检查并确认了'/tmp'是一个合法的目录，然后切换到这个临时目录开始我们的工作。切换到这个目录后，调用getcwd()方法给出当前位置。

```
>>> os.mkdir('example')
>>> cwd = os.getcwd()
>>> cwd
'/tmp/example'
>>>
>>> os.listdir()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: function requires at least one argument
>>>
>>> os.listdir(cwd)
[]
```

接下来，我们在这个临时目录里建立了一个下级子目录，然后用listdir()方法检查并确认了新子目录里面什么也没有（因为我们刚创建它）。第一次调用listdir()时出现了错误，原因是我们忘了给出准备列出文件清单的目录的名字。我们马上在第2次调用时改正了这个失误。

```
>>> file = open('test', 'w')
>>> file.write('foo\n')
>>> file.write('bar\n')
>>> file.close()
>>> os.listdir(cwd)
['test']
```

接下来，我们创建了一个有两行内容的测试文件，再用列目录文件清单的办法检查并确认那个文件确实被创建了。

```
>>> os.rename('test', 'filetest.txt')
>>> os.listdir(cwd)
['filetest.txt']
>>>
>>> path = os.path.join(cwd, os.listdir(cwd)[0])
>>> path
'/tmp/example/filetest.txt'
>>>
>>> os.path.isfile(path)
1
>>> os.path.isdir(path)
0
>>>
>>> os.path.split(path)
('/tmp/example', 'filetest.txt')
>>>
>>> os.path.splitext(os.path.basename(path))
('filetest', '.ext')
```

上面这段代码练习使用了os.path模块中的一些功能，包括join()、isfile()、isdir()、split()、basename()和splitext()等方法。我们还调用了os模块中的rename()函数。

```
>>> file = open(path)
>>> file.readlines()
>>> file.close()
>>>
>>> for eachLine in allLines:
...     print eachLine,
...
foo
bar
```

接下来的这段代码对大家来说应该是很熟悉的，因为这已经是我们第三次遇到它了。打开那个测试文件，读入所有的数据行，关闭文件，再每次一行地显示所有的行。

```
>>> os.remove(path)
>>> os.listdir(cwd)
[]
>>> os.chdir(os.pardir)
>>> os.rmdir('example')
```

最后一段删除了那个测试的文件，还试验了目录的退出操作。对chdir()方法的调用使我们返回到上一级目录里去，然后从那里删除了我们练习用的目录（os.pardir属性保存着父目录路径的字符串：UNIX和Windows用的是“..”；Macintosh用的是“..”）。删除自身所在的目录是没有道理的。

模块：os（以及os.path）

[CM]

从上面这些讨论里可以看出，os和os.path模块提供了访问计算机上文件系统的不同方法。虽然我们对本章的学习基本上是严格限制在文件访问方面，但os模块实际上能够完成更多的工作。它使我们大家能够管理自己的进程环境，提供了对文件进行底层访问的手段，使我们能够创建和管理新的进程，甚至使我们能够运行直接与另外一个运行中的程序进行“对话”的Python程序。你很快就会发现你离不开这个模块了。第14章里有对os模块进一步的论述。

9.8 文件的执行

不管你是简单地运行一个操作系统命令，还是调用一个二进制可执行文件或者调用执行其他类型（比如说是一个shell脚本程序、Perl或者Tcl/Tk）的脚本程序，都要涉及到执行保存在系统其他地方的另外一个文件。甚至在运行另外的Python代码时也有可能需要另外启动一个Python解释器，尽管这种情况不经常出现。我们将把这部分内容留到第14章去讨论。如果读者有兴趣了解如何启动其他程序——哪怕只是想与它们进行通信，或者有兴趣了解与Python的执行环境有关的一般知识，都可以在第14章里找到答案。

9.9 永久性存储模块

在本书的许多练习里，应用程序都要求由用户输入大量的数据。经过这么多次折腾之后，

我们想读者对翻来覆去地输入同样的数据也有点烦了。如果你正准备输入大批数据供今后使用，那肯定也会有同样的想法。这就是永久性存储大显身手的地方了，它的作用是把用户的数据归档保存起来以便今后的使用，使大家不必反复输入同样的资料了。在简单的磁盘文件已经不能满足我们的需要，而完整的关系数据库管理系统（relational database management systems，简称RDBMSs）又有些大材小用的情况下，简单的永久性存储就能够填补两者之间的沟壑。永久性存储的大部分内容都是用来对付字符串数据在存储方面的问题的，但它也有归档Python对象的办法。

9.9.1 pickle和marshal模块

Python语言中有许多模块可以实现基本的永久性存储。其中的一组模块（marshal和pickle）可以用来解决Python对象的存储转换问题。“存储转换”是这样—个过程：比基础类型要复杂的对象可以被转换为一个二进制字节的数据集合，这个数据集合可以被保存起来或者通过网络来传输，到那里再重新恢复这些对象的本来面目。“转换存储”也叫做数据的扁平化、数据的串行化，或者叫做数据顺序化。另外一组模块（包括dbhash/bsddb、dbm、gdbm、dumbdbm等）和它们的“管理器”（anydbm）只能对Python字符串提供永久性存储。而最后一个模块（shelve）两种功能都具备。

我们在前面已经提到过，marshal和pickle都可以对Python对象进行变换存储处理。这些模块本身并不提供“永久性存储”方面的功能，因为它们并没有为对象留出名字空间（namespace）；它们也不能在永久性对象上实现共发的写访问（concurrent write access）操作。它们能做的事情只是对Python对象进行存储转换好让它们能够被保存和传输。数据存储当然要有次序（对象的存储和传输是一个接着一个地进行的）。marshal和pickle两者之间的区别是：marshal只能处理简单的Python对象（包括数字、序列、映射和代码等类型）；而pickle则可以对递归对象、多地点多引用线索对象、用户定义的类（class）及其实例（instance）进行变换。pickle模块还有一个改进版本叫做cPickle，它实现了C语言中的全部相关功能。

9.9.2 DBM风格的模块

*db*系列模块以传统的DBM格式写入数据。这一系统的模块有许多种不同的实现，比如：dbhash/bsddb、dbm、gdbm和dumbdbm等。我们强烈推荐使用anydbm模块，它能够自动检测出用户系统上安装的是哪一种DBM兼容模块，然后判断出并加载上“最佳”的那个模块。dumbdbm模块是功能最少的，如果没有其他模块可用，这个模块就将是系统缺省使用的。这些模块倒是为用户的对象留出了名字空间，所使用的对象在行为方面同时具备字典对象和文件对象的特点。这些模块的一个不足之处是它们只能存储字符串。换句话说，它们不能对Python对象进行串行化转换。

9.9.3 shelve模块

最后，我们来看看一个比较复杂一些的解决方案，即shelve模块。shelve模块利用anydbm模块找出一个最合适的DBM模块，然后通过cPickle完成对数据的存储变换处理。shelve模块允许

对数据库文件进行共发的读访问操作，但不允许共享的读/写访问操作。这是我们在Python的标准库里所能找到的最接近于永久性存储的东西了，可能还有其他一些外部的扩展模块能够完全实现“真正”的永久性存储。图9-1中的示意图给出了存储转换模块和永久性存储模块之间的关系，同时也说明了shelve对象是怎样把这两大类模块的优点集中于自身的。

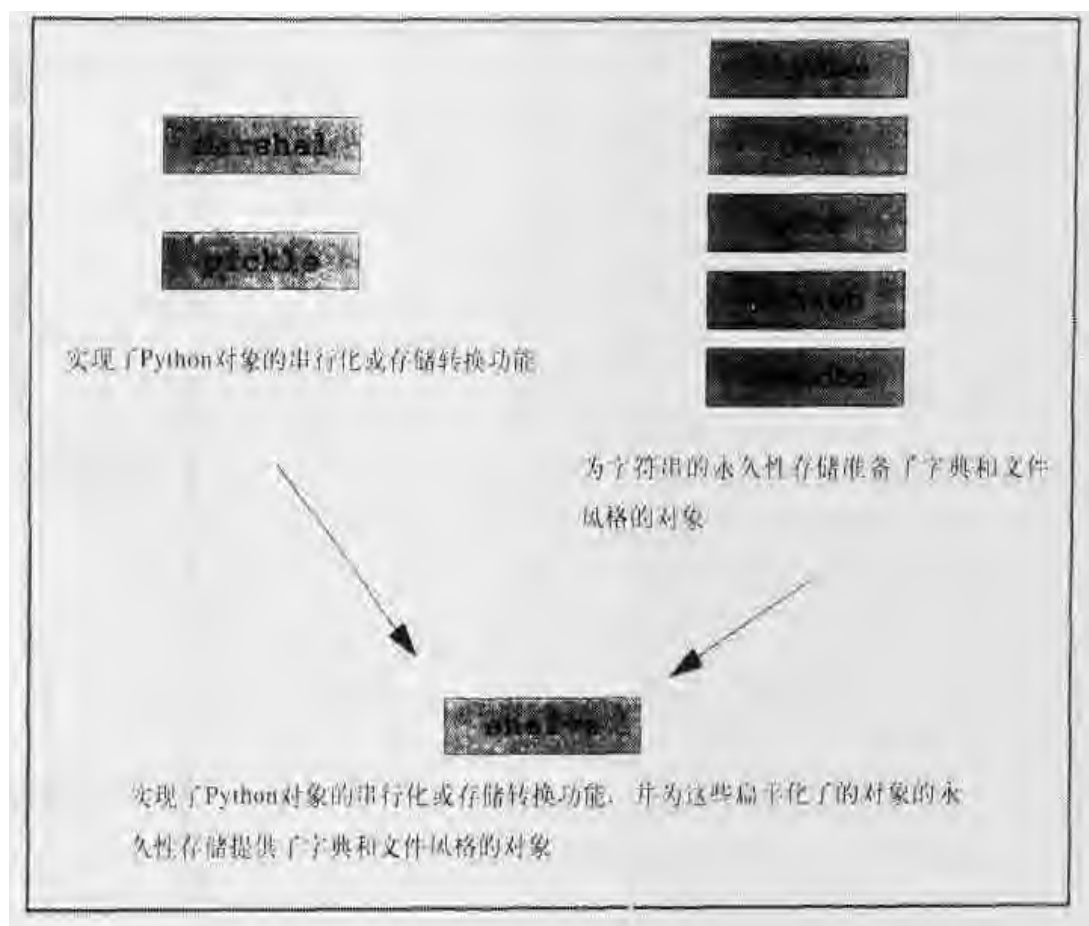


图9-1 Python语言中的串行化和永久性存储模块

模块：pickle

[CM]

`pickle`模块允许把Python对象直接保存到一个文件里去，不需要把它们转换为字符串，也不需要使用底层的文件访问操作把它们写到一个二进制文件中去。`pickle`模块会创建一个Python语言专用的二进制版本，使用户能够在不必考虑文件读写细节的情况下干净利索地读写完整的数据对象。只要有一个合法的文件句柄，你就可以在磁盘上读写数据对象了。

`pickle`模块中两个主要的函数是`dump()`和`load()`。`dump()`函数以一个文件句柄和一个数据对象为参数，把该对象以它自己明白的格式保存到指定的文件里去。当使用`load()`函数从磁盘上载入一个经过存储转换的对象时，它也知道应该如何把那个对象恢复为被保存到磁盘之前的原始配置形式。我们建议读者认真阅读一下`pickle`模块和它“更聪明”的兄弟`shelve`模块，后者提供了字典方式的文件访问功能，进一步减少了程序员在文件操作方面的负担。

9.10 相关模块

还有许多其他的模块与文件和输入/输出有关，它们中的大多数都能够工作在主要的计算机平台上。表9-7列出了部分与文件操作相关的模块。

表9-7 相关文件模块

模 块	内 容
fileinput	遍历多个输入文件的全部数据行
getopt	提供了对命令行参数的分析和处理功能
glob/fnmatch	提供了UNIX风格的通配符匹配功能
gzip/zlib/zipfile ^①	允许在访问文件时加上自动进行的压缩/解压缩功能
shutil	提供了高级的文件访问功能
c/StringIO	在字符串对象顶部实现了文件风格的操作接口
tempfile	生成临时文件名或临时文件

① 从Python 1.6版本开始出现。

fileinput模块遍历一组输入文件，每次读入它们的一行内容，使用户能够遍历每一行数据，与Perl语言中不带任何给定参数的“<>”操作符的工作方式很相似。没有明确给出的文件名被默认是从命令行上输入的。

glob和fnmatch模块允许以老式的UNIX操作系统的shell风格对文件名进行模式匹配，即用星号(*)通配符代表任意字符串，用问号(?)代表单个字符。

gzip和zlib模块提供了对zlib压缩库文件的直接访问手段。gzip模块是在zlib模块的基础上编写出来的，它允许进行标准的文件访问操作，但提供了自动的gzip兼容的压缩和解压缩功能。请注意：如果读者正在编译自己的Python解释器，千万不要忘记（通过编辑Modules/Setup文件）激活建立zlib模块的选项，因为在缺省状态下是不会自动打开这个设置开关的。新的zipfile模块也要求系统中先有zlib模块，它允许程序员创建、修改和读取zip档案文件。 [1.6]

shutil模块提供高级的文件访问功能，包括拷贝文件、拷贝文件的访问权限、递归地拷贝目录树等函数。

在前面介绍字符串的章节里，我们已经提到过StringIO模块（以及它的C语言编译版本CStringIO模块），并且介绍了它是在字符串对象的顶部又增加了一个文件操作接口。这个操作接口包括所有适用于普通文件对象的标准方法。

我们在前面永久性存储小节（9.9节）里提及的模块中还包括了带有文件和字典对象混合风格的例子。

其他产生类似于文件的对象的Python模块包括：网络 and 文件套接字（socket）对象（socket模块）、把应用程序连接到其他运行中进程的popen*()文件对象（os和popen2模块）、用在底层文件访问中的fdopen()文件对象（os模块），以及通过一个因特网Web服务器的统一资源定位器（Uniform Resource Locator，简称URL）地址建立一个到该服务器的网络连接（urllib模块）等。需要注意的是并非所有标准的文件方法都能在这些对象上全部实现；反而是这些模块额外增加了一些对普通文件进行处理的功能。

详细资料请参考这些与文件访问相关的模块的文档。

9.11 练习

9-1 文件过滤。显示一个文件中的全部行，但不显示那些以井字号（#）打头的行。这个字符是Python、Perl、Tel以及大多数其他脚本程序语言中使用的注释符号。

9-2 文件访问。提示输入一个数字N和文件F，然后显示F的前N行。

9-3 文件信息。提示输入一个文件名，显示该文本文件中的行数。

9-4 文件访问。编写一个逐页显示文本内容的程序。你的解决方案需要提示输入一个文件名，然后每次显示该文本文件的25行；显示一页后暂停下来，向用户提示“press a key to continue”（按任意键继续）。

9-5 考试成绩。改进考试成绩问题（练习5-3和6-4）的解决方案，要求从一个文件里读入一组考试成绩。文件的数据格式由你自己定。

9-6 文件比较。编写一个比较两个文本文件的程序。如果它们不同，给出第一个不同之处的行号和列号。

9-7 分析文件。Windows用户：编写一个对Windows.ini文件进行分析的程序。UNIX用户：编写一个对etc/services文件进行分析的程序。其他计算机平台用户：编写一个分析某个系统文件的程序，文件要有一定的结构。

9-8 模块研究。提取模块的属性资料。提示用户输入一个模块的名字（或者通过命令行接收它）。然后用dir()和其他内建函数提取出该模块的所有属性，把它们的名字、类型和值显示出来。

9-9 Python程序的文档字符串。进入保存着Python标准库模块的目录。依次检查每个.py文件，看看该模块是否有一个__doc__字符串。如果有，对其格式进行适当的整理和归类。当你的程序执行完毕时，应该生成一个漂亮的清单，里面列出哪些模块有文档字符串以及该字符串的内容。清单尾部要附上那些没有文档字符串的模块的名字。附加题：提取出标准库中各模块内全体类（class）和函数的文档。

9-10 家庭理财。编写一个家庭理财程序。你的解决方案需要能够处理储蓄、支票、汇兑、定期存款等多种帐户。为每种帐户提供一个菜单操作界面，要有存款、取款、借、贷等操作；还要有一个选项让用户能够取消本次交易。用户退出这个程序时有关数据应该保存到文件里去（出于备份目的，在程序执行过程中也要时不时地备份一下）。

9-11 Web站点地址。

a) 编写一个URL书签管理程序。编写一个使用文本菜单的应用程序，用户可以用它添加、修改或者删除书签数据项。书签数据项中包括站点的名称、站点的URL地址，还可以有一行简单的说明（可选）。要提供检索功能，检索操作要在站点名称和URL两部分中查找可能的匹配。你要在用户退出这个程序时把数据保存到一个磁盘文件里去；用户再次进入时加载保存的数据。

b) 改进a)部分的解决方案，把书签输出到一个合法并且语法正确的HTML文件（.htm或者.html）里去，这样用户就可以用他们的浏览器指向这个输出文件去查看自己的书签清单。该运行实现的另一个功能是创建“文件夹”，对相关的书签进行分组管理。附加题：请阅读Python

的re模块中关于规则表达式的文字材料。对用户输入到他们的数据库中的URL进行验证。

9-12 用户和口令字。

a) 做练习7-5，记录用户名和口令字。修改你的代码使之支持一个“上次登录时间”。请阅读time模块中的文档以了解如何才能获得用户“登录”到系统的时间标签。此外，创建一个“administrator”（系统管理员）用户的概念，他可以列出全体用户、他们的口令字（如果愿意还可以对口令字进行加密处理）和他们上次登录的时间等几个清单。这些数据库都要保存到磁盘上去，每个用户占用一行，各数据域之间用冒号(:)隔开，比如：“joe:boohoo:953176591.145”。文件中数据行的数目应该是你系统上的用户的数目。

b) 进一步改进你的程序，使它不是每次写入一行数据，而是对整个数据库对象进行存储转换后再写到文件里去。请阅读关于pickle模块的文档以了解如何才能“扁平化”或“串行化”你的对象，了解如何才能对存储转换后的对象进行I/O操作。利用这个模块提供的功能，你现在的解决方案应该比a)中解决方案的行数要少。

9-13 命令行参数。

a) 什么是命令行参数？它们有什么作用？

b) 编写一段代码显示输入的命令行参数。

9-14 记录结果。把你的计算器程序（练习5-6）修改为从命令行读取输入数据，即：

```
% calc.py 1+2
```

只输出计算结果。此外，把每一个表达式及其计算结果写到一个文件里去。当发出下面的命令时，

```
% calc.py print
```

会把“记录磁带”上的所有内容都显示到屏幕上去，再重置/截短记录文件。下面是程序某次执行的情况：

```
% calc.py 1 + 2
3
% calc.py 3 ^ 3
27
% calc.py print
1 + 2
3
3 ^ 3
27
% calc.py print
%
```

9-15 拷贝文件。提示输入两个文件名（更好的办法是使用命令行参数）。第一个文件中的内容将被拷贝到第二个文件中去。

9-16 文字处理。因为人们输入的文字经常超过你的邮件阅读器程序的行宽，所以你已经厌倦看到打包了的邮件文本。编写一个程序，在一个文本文件中查找那些长于80个字符的文本行。在这些超标的行里找到最靠近第80个字符的单词，从那里分断这一行，把剩余的文字插到下一行去（把原来的下一行向下压一行）。

程序执行完毕后，应该没有超过80个字符的文本行了。

9-17 文字处理。编写一个原始初级的文本文件编辑器。你的解决方案应该是菜单驱动的，并且要有以下几个选项：1) 创建文件（要提示输入文件名和任意行数的文本输入）；2) 显示文件内容（把它的内容显示到屏幕上）；3) 编辑文件（提示输入准备修改的行并允许用户对它进行修改）；4) 保存文件；5) 退出。

9-18 搜索文件。提示输入一个字节值（0到255）和一个文件名。显示该字节在那个文件里出现的次数。

9-19 生成文件。编写一个上一问题的姐妹程序。创建一个包含随机字节数据的文件，但那个特定的字符必须出现给定的次数。这个程序要求有三个参数：1) 一个字节值（0到255）；2) 该字节应该出现在文件中的次数；3) 构成数据文件的字节总数。你的任务是创建那个数据文件，把给定字节随机散布在文件里，并且要保证给定字符在文件里只出现指定的次数，而结果数据文件应该精确地达到要求的长度。

第10章 错误和例外处理

错误是每个程序员每天都会遇到的事情。在过去的一个时期，错误要么对程序（也许还有机器）是致命的；要么会产生出一大堆无用的输出，别的计算机或程序无法把他们辨认为合法的输入，提交该作业运行的人也搞不懂它的意义。只要出现了错误，程序的执行就会被中止，直到错误被纠正，程序再重新执行才算告一段落。随着时间的推移，人们要求有一个比较“柔和”的对付错误的办法而不是简单地停止程序的执行的呼声越来越高，而程序本身也在不断地进化。程序中的错误并不是每一个都是致命的；并且即使错误真的发生了，编译器或者执行中的程序也能够提供更多的诊断信息，这样就可以帮助程序员尽快地解决问题。但错误终究还是错误，其解决办法通常还是要在程序或者编译过程停止后才能实行。除了退出执行——也许还会给出一些错误潜在原因的模糊提示以外，一段代码实际能做的补救是很少的，但这一切都是例外和例外处理出现以前的事情了。

虽然我们目前还没有讨论到Python语言中的类（class）和面向对象的程序设计，但本章涉及的许多概念都是与类和类的实例有关的。（从Python 1.5版本开始，所有标准的例外都按类来实现和对待。如果读者不太了解类、实例或者其他面向对象的术语，请先到第14章把这些概念弄清楚。）在本章的结束部分，我们特意为读者准备了一小节如何自行创建例外类（exception class）的选读内容。老版本的Python使用的是字符串形式的例外，现在已不多见了。我们建议人家在今后的开发中尽量使用基于类的例外。

本章内容将先向大家介绍什么是例外、什么是例外处理，以及Python语言对它们有怎样的支持。我们还将向程序员介绍如何在自己的代码里引发例外。最后，我们将揭示如何创建自己的例外类。

10.1 什么是例外

10.1.1 错误

在深入介绍例外之前，我们先来看看什么是错误。从软件方面来说，错误本身或者是语法上的，或者是逻辑上的。语法错误表示在软件的结构方面出现了问题，它使解释器不能正确地执行，或者不能正确地编译程序。这些错误必须在程序执行之前加以纠正。

程序的语法正确性得到保证之后，剩下的就都是逻辑错误了。逻辑错误的原因可能是缺少输入或者是输入不正确；在某些情况下，还可能是根据输入无法按逻辑生成或者计算出预期结果等情况。这些错误通常被分别称为域错误和范围错误。

当Python检测到一个错误的时候，解释器会指出它已经到达这样一个地点：即按照当前的流程已经不可能继续执行下去了。这时就该例外登场了。

10.1.2 例外

对例外最好的描述是：它是因为程序出现了错误而在正常的控制流以外采取的动作。这个动作又分为两个阶段：首先是出现了一个引发了某种例外的错误；其次是检测（和采取可能的解决措施）阶段。

第一个阶段是在发生了一个“例外情况”（“exception condition”，有时也叫做“例外的情况”）。只要检测到一个错误并识别出一个例外情况，解释器就会立刻执行一个叫做“引发一个例外”的操作。引发也可以叫做触发、抛出或者生成等等，它就是解释器通知当前控制流有问题发生的那个过程。Python语言还向程序员提供并支持其引发例外的能力。不管是Python解释器触发的还是程序员触发的，例外就是出现了错误的信号。程序当前的执行流程将被中断去处理这个错误并采取相应的动作，这正好就是所谓的第二阶段。

第二个阶段就是对例外进行处理。例外引发之后，将调用一系列操作来对那个例外做出反应。这些操作涉及的范围很广，主要有以下几种：忽略该错误；记录下该错误但不采取任何措施；采取一些修正措施后终止程序运行；或者弱化那个问题并设法继续执行程序。所有这些操作都代表着一种“继续”，也就是例外一个控制分支。关键是程序员能够在错误发生时指示程序如何继续运行。

读者可能已经自己总结出这样一个结论：程序运行时发生的错误主要是由外部原因引起的，比如不良输入或某种类型的操作失败等等。这些因素并不处于程序员的直接控制之下，因为程序员只能预见一部分错误，也只能编写常见补救措施的代码。

象Python这样支持引发和处理例外（这一点更加重要）的程序设计语言使程序开发人员能够在错误发生时具备更直接的控制权力。程序员不仅能够检测到错误，还能够在错误发生时采取一些更可靠的补救措施。因为程序在运行时具有对错误进行管理的能力，所以就可以说应用程序的健壮性增强了。

例外和例外处理并不是什么新概念，早在Ada、Modula-3、C++、Eiffel和Java等程序设计语言里它们就已经出现了。例外这个概念可能出自处理系统错误和硬件中断等例外的操作系统代码。做为软件工具的例外处理手段最早出现在1960年代中期，而PL/1则是第一个具备例外概念的主要的程序设计语言。和其他一些支持例外处理的程序设计语言一样，Python也采取了用一个“try”代码块来“捕获”例外的办法，但它在例外的处理方面更有“纪律性”。这样，在对所谓意料之中的例外进行检测时，我们就能针对不同的例外编写出不同的处理代码，不再象只有一个通用性的“一把抓”代码时那么盲目。

10.2 Python语言中的例外

因为读者已经在上一章里看到过一些例子，所以大家可能已经注意到在自己的程序“崩溃”——即因为无法解决的错误而终止时会发生什么事情。你会看到一个traceback通知和由解释器尽其所能向你提供的一条诊断信息，其中包括该错误的名称、原因，也许还有尽量接近错误发生位置的程序行编号。不管是运行在Python解释器里还是做为标准的脚本程序来执行，一切错误都有相近的格式，这有助于使各种错误的程序界面保持一致。不管是逻辑上的还是语法上的

错误，都是因为与Python解释器的不相容而导致的，其后果就是引发例外来。

我们现在来看几个例外。

1. NameError: 试图访问一个未经初始化的变量

```
>>> foo
Traceback (innermost last):
  File "<interactive input>", line 0, in ?
NameError: foo
```

NameError例外表示我们对一个没有经过初始化的变量进行了存取。引起错误的那个变量在Python解释器的符号表里没有找到。我们会在后面的章节里对名字空间(namespace)进行讨论，现在大家只要把它们想象为联系名字和对象的“地址簿”就可以了。能够被存取的对象都会被列在某个名字空间里。存取某个变量的操作最终落实为解释器的一次检索操作，如果在全部的名字空间里都找不到要求的名字，就会产生一个NameError例外。

2. ZeroDivisionError: 一个数被零除

```
>>> 12.4/0.0
Traceback (innermost last):
  File "<interactive input>", line 0, in ?
ZeroDivisionError: float division
```

我们在上面的例子用的是浮点数，但一般来说，任何数值类型的零除法都会导致一个ZeroDivisionError例外。

3. SyntaxError: Python解释器语法错误

```
>>> for
      File "<string>", line 1
        for
          ^
SyntaxError: invalid syntax
```

SyntaxError是唯一的一个不会在程序运行中发生的例外。它们表示Python代码中有一个不正确的结构，在得到纠正之前无法运行程序。这些错误一般都发生在编译期间，即发生在解释器把用户的脚本程序调入内存并试图把它转换为Python的字节码的时候。它们也可能是因为导入了一个有缺陷的模块而产生的。

4. IndexError: 为序列请求一个超出范围的索引

```
>>> aList = []
>>> aList[0]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```

如果试图使用一个序列合法范围以外的下标，就会引发一个IndexError例外。

5. KeyError: 请求一个不存在的字典键字

```
>>> aDict = {'host': 'earth', 'port': 80}
>>> print aDict['server']
Traceback (innermost last):
  File "<stdin>", line 1, in ?
KeyError: server
```

象字典这样的映射类型必须要依靠键字来访问数据的值。如果用的键字不正确或不存在，

其映射值是检索不到的。此时就会引发一个KeyError例外表示出现了这样的问题。

6. IOError: 输入/输出错误

```
>>> f = open("blah")
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
IOError: [Errno 2] No such file or directory: 'blah'
```

试图打开一个并不存在的磁盘文件就是操作系统输入/输出(I/O)错误的一个例子。任何形式的I/O错误都会引发一个IOError例外。

7. AttributeError: 试图访问一个未知对象的属性

```
>>> class myClass:
...     pass
...
>>> myInst = myClass()
>>> myInst.bar = 'spam'
>>> myInst.bar
'spam'
>>> myInst.foo
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: foo
```

在上面的例子里，我们在myInst.bar——即实例myInst的bar属性里保存了一个值。一旦对一个属性做出了定义，我们就可以利用熟悉的点属性记号来访问它；可如果没有进行过定义，比如上面的foo情况，就会引发一个AttributeError例外。

10.3 检测和处理例外

把例外体现在try语句的组成部分里就可以把它们检测出来。系统会监控try语句的所有代码子句，检查其中有无例外发生。

try语句主要有两种形式，它们是try-except和try-finally。两种语句不能同时使用，只能选用其中的一种。一个try语句可以配对使用一个或者多个except从句，但只能配对使用一个finally从句。（绝对不允许出现try-except-finally形式的混合用法。）

try-except语句使人们能够对例外进行检测和处理。为了应付没有检测到例外但又需要执行一些代码的情况，Python语言甚至还为大家准备了一个可选的else从句。而try-finally语句只具备检测例外是否发生和强制进行扫尾处理（不管例外是否发生）两个功用，除此之外再没有别的对付例外的手段了。

10.3.1 try-except语句

try-except语句（以及这个语句更复杂的版本）允许定义一段代码进行例外监控，并且提供了执行例外处理程序的机制。

最普通的try-except语句的语法如下所示：

```
try:
    try_suite      # watch for exceptions here
```

```
except Exception:
    except_suite    # exception-handling code
```

我们用·一个例子来说明这一切是如何进行的。我们使用的是前一小节中IOError的例外做例子。把代码用try-except语句“打包起来”可以让代码更加健壮，如下所示：

```
>>> try:
...     f = open('blah')
... except IOError:
...     print 'could not open file'
...
could not open file
```

正如你所看到的，这段代码在运行时似乎没有发生什么错误。可实际情况是：当我们试图打开那个并不存在的文件时，同样的IOError例外依然发生了。有什么区别吗？有，我们添加了检测和处理这个错误的代码。当那个IOError例外被引发时，我们告诉解释器要它做的一切就是显示一条诊断性信息；程序将继续执行，不再象以前例子里那样被“轰出来”了——例外处理小小地显了一下身手。那么在代码方面到底发生了怎样的事情呢？

在程序运行期间，解释器会尝试执行try语句里的一切代码。如果在代码段执行完毕后没有发生例外，程序将跳过except语句继续执行；如果在except语句上指定的例外真的发生了，控制流会立刻跳到与之对应的处理器（即处理该例外的代码部分）继续执行（try从句中还没有执行到的代码都会被跳过去，不会再执行）。我们上面的例子只捕获IOError例外，其中的例外处理器不捕获其他例外。如果，比如说，你还想捕获一个OSError，就需要增加一个专门用于那个例外的处理器。随着我们对本章学习的深入，我们将逐渐揭开try-except语法的庐山真面目。

编程提示：跳过代码、继续执行和向上移交

[CN]

从例外发生位置开始，try子句中的剩余代码永远也到不了（因此也就执行不了）了。只要出现了例外，就必须决定控制流将何去何从。剩余的代码被跳过去，不再执行；查找一个处理器的工作开始了，如果找到了一个，程序就转到那个处理器内继续执行。

如果检索穷尽后还是没有找到一个对应的处理器，那个例外就会被移交给调用者去处理，这就意味着堆栈架构将立刻回到当前堆栈之前的那一个。如果在上一层调用者那里也没有对应的处理器，该例外会再被继续向上移交。如果一直到顶部都没有找到对应的处理器，这个例外就会被认为是未被处理的，最后就要由Python解释器出面，它会在显示一条traceback信息之后退出。

10.3.2 打包一个内建函数

[1.5]

我们现在给出一个交互操作的例子——先从最基本的错误检测编写起，然后逐步改进它，增强这段代码的健壮性。问题的核心是把一个用字符串形式表示的数值转换为正确（数值类型）的数值表示形式，在这个过程中要检测并处理可能的错误。

float()内建函数的基本作用是把任一数值类型转换为一个浮点数。从Python 1.5版本开始，float()的功能增加了，可以把一个用字符串形式表示的数值直接转换为真正的浮点数值，不再需

要使用string模块中的atof()函数了。如果读者现在用的是老版本的Python, 请把下面例子中的float()替换为string.atof()函数。

```
>>> float(12345)
12345.0
>>> float('12345')
12345.0
>>> float('123.45e67')
1.2345e+069
```

但要是float()遇到了不合适的输入, 它也挺挑剔的:

```
>>> float('abcde')
Traceback (innermost last):
  File "<stdin>", line 1, in ?
    float('abcde')
ValueError: invalid literal for float(): abcde
>>>
>>> float(['this is', 1, 'list'])
Traceback (innermost last):
  File "<stdin>", line 1, in ?
    float(['this is', 1, 'list'])
TypeError: object can't be converted to float
```

从上面出现的错误可以看出: float()对不代表数值或者非字符串数据的字符串很不客气。更仔细的观察可以看出: 如果参数的类型正确(是字符串类型)但该类型包含的值不合适, 被引发的例外将是ValueError——这是因为是数据的值而不是数据的类型有问题; 反之, 一个列表从总体上来说就是一个不良输入, 连类型都不对了, 所以它引发的是TypeError例外。

我们的目的是“安全地”或者说“更安全地”调用float()函数, 就是说我们打算忽略这类错误情况, 因为它们与这个把用字符串形式表示的数值转换为浮点数的任务没什么关系, 而且这些错误也没有严重到值得解释器放弃执行这个程序。为了实现这个目的, 就需要我们创建一个“打包”函数, 在try-except的帮助下制造出我们预想的程序执行环境——我们给它起名为safe_float()。在第一次改进里, 我们只检测和忽略ValueError, 因为这种例外情况是最常发生的。TypeError例外很少发生, 因为几乎人人都知道float()函数的参数必须是一个非字符串数据。

```
def safe_float(object):
    try:
        return float(object)
    except ValueError:
        pass
```

我们采取的第一个措施是“止血”。在上面的例子里, 我们把错误“吞”了下去, 让它不再影响程序的执行流程。换句话说, 错误将被捕获, 但因为我们没在except子句里放什么东西(除了一条占着地方但什么也不做的pass语句, 这个语句表示从语法上来讲那个位置应该出现程序代码), 所以不进行什么处理。我们只是不理睬那些个错误罢了。

这个解决方案有一个明显的不足, 那就是在出现错误的情况下也不向其函数调用者明确地返回什么信息。而如果返回的是None(如果一个函数没有明确地返回一个值, 比如说在没有遇上“return object”语句就结束了执行的时候, 它就会返回一个None), 我们也很难弄清楚到底是哪儿出了问题, 出了什么样的问题。我们至少应该明确返回一个None, 这样在出错和没出错

两种情况下我们的函数都能返回一个值，使这段代码多少容易理解了一些。如下所示：

```
def safe_float(object):
    try:
        retval = float(object)
    except ValueError:
        retval = None
    return retval
```

请注意，刚才的修改并不影响我们的代码，只是多用了—个局部变量而已。对—个编写风格良好的应用程序程序员接口（application programmer interface, API）来说，返回值可以更灵活—些。也许可以这样做：如果传递给safe_float()函数—个正确的参数，那就应该返回—个浮点数；如果出现了错误，我们可以返回—个字符串说明输入数据有什么问题。按照这个办法再改进—次代码，如下所示：

```
def safe_float(object):
    try:
        retval = float(object)
    except ValueError:
        retval = 'could not convert non-number to float'
    return retval
```

我们在例子里所做的唯一的修改是把None替换为—个错误信息字符串。我们现在来试试这个函数，看它现在的表现如何：

```
>>> safe_float('12.34')
12.34
>>> safe_float('bad input')
'could not convert non-number to float'
```

这是一个好的开头——我们现在已经能够对付非法的字符串输入了，可如果传递来的是一个非法的对象，还是会“受伤”，如下所示：

```
>>> safe_float({'a': 'Dict'})
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "safeflt.py", line 28, in safe_float
    retval = float(object)
TypeError: object can't be converted to float
```

这里只是暂时地指出这最后—个弱点，但在进一步改进我们的示例程序之前，最好是先去看看try-except灵活多变的语法，特别是except语句，它有好几种变化形式。

10.3.3 带多个except的try语句

我们在本章的前面已经介绍过except语句如下所示的通用语法：

```
except Exception :
    suite_for_exception_Exception
```

这种格式的except语句专门用来处理名为Exception的例外。我们可以把多个except语句串在—起来处理同—个try代码块产生的不同类型的例外，如下所示：

```
except Exception1:
    suite_for_exception_Exception1
except Exception2:
```

```
suite_for_exception_Exception2
    :
```

先尝试执行try从句，如果没有错误发生，程序将跳过全部的except从句；可要真的发生了一个例外，解释器就会在处理器清单里查找与那个例外对应的处理器（即except从句）。如果找到了，程序就会跳到那个except子句部分继续执行。

我们的safe_float()函数在检测某些特定的例外情况方面现在已经有一些头脑了。更聪明的代码能够把每一种例外都处理好。这就需要我们使用多个except语句，每一个except语句对应一种例外类型。这不会有什么问题，因为Python语言允许把except语句串在一起使用。那些熟悉流行的第三代程序设计语言（third-generation language, 3GL）的读者肯定会注意到这与Python语言所没有的switch/case语句的相似之处。我们现在就来编写每个错误类型分别对应的错误信息，为用户问题的起因提供更准确的情报，如下所示：

```
def safe_float(object):
    try:
        retval = float(object)
    except ValueError:
        retval = 'could not convert non-number to float'
    except TypeError:
        retval = 'object type cannot be converted to float'
    return retval
```

执行这段代码，给它一些错误的输入，我们将得到如下所示的结果：

```
>>> safe_float('xyz')
'could not convert non-number to float'
>>> safe_float()
'argument must be a string'
>>> safe_float(200L)
200.0
>>> safe_float(45.67000)
45.67
```

10.3.4 处理多个例外的except语句

我们也可以让一个except从句处理多个例外。对多个例外进行处理的except语句要求那组例外被包含在一个表列里，比如说：

```
except ( Exception1, Exception2 ) :
    suite_for_Exception1_and_Exception2
```

上面这个语法示例演示了如何用一段代码处理两个例外的情况。一般说来，一个except语句后面可以跟任意多个例外，但前提是它们都必须正确地被放在一个表列里，如下所示：

```
except ( Exception1[, Exception2[, . . . ExceptionN . . . ] ] ) :
    suite_for_exceptions_ Exception1_to_ExceptionN
```

如果因为某些原因（比如说内存不足或设计方面的因素）要求safe_float()函数的全部例外都必须在同一段代码里进行处理，我们现在就可以满足这个要求：

```
def safe_float(object):
    try:
```

```

    retval = float(object)
except (ValueError, TypeError):
    retval = 'argument must be a number or numeric string'
return retval

```

现在，遇到错误输入时返回的错误信息字符串都是一样的了：

```

>>> safe_float('Spanish Inquisition')
'argument must be a number or numeric string'
>>> safe_float([])
'argument must be a number or numeric string'
>>> safe_float('1.6')
1.6
>>> safe_float(1.6)
1.6
>>> safe_float(932)
932.0

```

10.3.5 不带例外名参数的try-except语句

try-except的最后一种语法形式是在except语句后面根本不定义例外：

```

try:
    try_suite      # watch for exceptions here
except:
    except_suite  # handles all exceptions

```

虽然这段代码捕获的例外最多，可这并不是Python语言良好的程序设计风格。一个主要的原因是它没有考虑到产生例外的问题根源。这类代码不去调查和发现可能会发生哪类错误以及如何避免其出现，它采取的是一种“眼不见为净”的做法，忽略了问题的原因（和补救措施）。请参考下面编程风格中的内容。

编程风格：把try-except和例外名一起使用

[CS]

Python语言中的try-except语句向程序员提供了一种功能强大的机制，使他们能够跟踪潜在的错误并在代码里准备好处理例外情况的逻辑；这类机制在其他语言（比如C语言）里是很难实现的。其宗旨是在减少错误出现次数的同时仍然保证程序的正确性。和其他工具一样，只有使用得当才能发挥其作用。

try-except语句的一个不正确的使用方法是把它当做一个大绷带“绑”在一大段程序上，也就是把一大段程序（如果还不是全部源代码的话）放在一个try语句里，再用一个大而化之的except语句忽略掉那些严重的错误，就像下面这样：

```

# this is really bad code
try:
    large_block_of_code # bandage of large piece of code
except:
    pass                # blind eye ignoring all errors

```

看不见错误不等于没有错误，而try-except语句的作用是提供这样一种机制：让不太严

重的问题可以通过补救措施得到适当的处理；不应该把它当作一个过滤器。上面这个结构能掩盖许多错误，但这是一种不负责任的行为，我们不主张这样做。

最低要求：避免用try-except语句打包一大段代码，然后用一个pass掩盖错误。正确的办法是在处理特定的例外时只把必要的代码放到try从句里去，就象本小节中safe_float()函数例子里我们用到那些结构一样。

10.3.6 例外参数

例外可以有参数，并且能够在例外被引发的时候一起传递到例外的处理器中去。例外被引发时，通常会向该例外的处理代码提供一个有辅助作用的参数。虽然例外的参数是可选的，但那些标准的内建例外都准备有至少一个参数，即一个给出该例外发生原因的错误信息字符串。

例外的参数可以在处理器里被忽略，但Python确实提供了保存这个值的语法。要想对例外的参数进行存取，就必须保留一个容纳该参数的变量。这个参数出现在except语句的标题行上，跟在将要处理的例外类型的后面。我们来看看except语句下面这个不同寻常的扩展语法：

```
# single exception
except Exception, Argument:
    suite_for_Exception_with_Argument

# multiple exceptions
except (Exception1, Exception2, ..., ExceptionN), Argument:
    suite_for_Exception1_to_ExceptionN_with_Argument
```

除非引发的是一个字符串形式的例外（请参考10.4节），Argument将是一个来自引发该例外的代码的包含着诊断信息的类实例。例外参数本身将被保存到一个表列中去，而这个表列又是那个类实例的一个属性。在上面第一个扩展语法里，Argument是Exception类（class）的一个实例（instance）。

对大多数标准的内建例外，即那些从StandardError推导出来的例外来说，做为其参数的表列里只包含一个指示错误原因的字符串。例外的名字已经可以做为一个比较令人满意的线索，加上错误信息字符串就说得更明白了。操作系统错误或其他与操作环境有关的错误，比如IOError，会在表列里错误信息字符串前多出一个操作系统错误编号。不管Argument是单独的一个字符串还是一个错误编号和一个字符串的组合，调用str(Argument)函数都能读到以可阅读方式显示的错误原因。

还有一个问题：某些第三方或其他外来模块所引发的例外并不都遵守这个标准的协议（即或者是错误信息字符串，或者是（错误编号，错误信息字符串）这两种例外参数格式）。我们建议大家在引发自己的例外时遵守这里介绍的标准（请参考下面的编程风格内容）。

编程风格：遵守有关例外参数的协议

[CS]

· 当你在自己的代码里引发内建的例外时，请尽量遵守现有的Python代码已经确立的协议，把错误信息做为那个例外的参数放在表列里传递出来。换句话说，如果你引发的是一

个ValueError例外，就要象解释器引发的ValueError例外那样提供同样的参数信息；以此类推。这有助于保持代码的一致性，并防止使用你模块的其他代码因此而崩溃。

下面是把一个非法的对象传递到float()内建函数，从而导致一个TypeError例外的例子：

```
>>> try:
...     float(['float() does not', 'like lists', 2])
... except TypeError, diag: # capture diagnostic info
...     pass
...
>>> type(diag)
<type 'instance'>
>>>
>>> print diag
object can't be converted to float
```

我们做的第一件事是在try语句里引发一个例外。忽略掉这个例外，但把错误信息保存起来。调用type()内建函数来确认这个例外确实是一个实例(instance)。最后，调用print语句和诊断性的例外参数显示出这个错误的原因。

利用特殊的__class__实例属性可以获得更多关于例外的资料，它标识出该实例是从哪个类(class)里初始化来的。这些类对象还有属性，比如一个文档字符串和一个进一步指明错误类型的字符串名字。如下所示：

```
>>> diag # exception instance object
<exceptions.TypeError instance at 8121378>
>>> diag.__class__ # exception class object
<class exceptions.TypeError at 80f6d50>
>>> diag.__class__.__doc__ # exception class documentation string
'Inappropriate argument type.'
>>> diag.__class__.__name__ # exception class name
'TypeError'
```

所有类实例都有一个特殊的实例属性__class__；如果还定义过文档字符串，那它们就还有类属性__doc__（我们将在第13章“类与OOP”里讨论这些内容）。

我们再来改进一次我们的safe_float()，当例外情况发生时，让它带上从解释器传递过来的来自float()内部的例外参数。在对safe_float()函数上一次的改进里，我们为了满足某些要求而把ValueError和TypeError两种例外的处理器合并在了一起。这个解决方案的不足是我们无法了解具体被引发的是哪一个例外，也不知道错误的原因是什么；唯一的返回值是一个指示出有非法输入参数的错误信息字符串。现在既然有了例外参数，情况当然会不同了。

因为每个例外都会产生它自己的例外参数，所以如果我们选择返回这个字符串而不是返回我们自己给出的那句套话，就可以为问题的根源提供一个更好的线索。在下面的代码块里，我们把当初自己编写的错误信息字符串替换为例外参数的字符串表示形式。

```
def safe_float(object):
    try:
        retval = float(object)
    except (ValueError, TypeError), diag:
        retval = str(diag)
    return retval
```


再运行这段新代码。当我们向`safe_float()`给出不正确的输入数据时，我们看到的是下面这些（不同的）信息，尽管两种例外是用同一个处理器解决的。如下所示：

```
>>> safe_float('xyz')
'invalid literal for float(): xyz'
>>> safe_float({})
'object can't be converted to float'
```

10.3.7 把打过包的函数用在一个应用程序里

我们现在来把`safe_float()`用到一个小应用程序里去，这个小程序要从小程序要从一个信用卡交易数据文件（`carddata.txt`）里读入所有的交易数据，包括说明文字在内。下面是`carddata.txt`文件例子的内容：

```
% cat carddata.txt
# carddata.txt
previous balance
25
debits
21.64
541.24
25
credits
-25
-541.24
finance charge/late fees
7.30
5
```

我们的程序，`cardrun.py`，在程序示例10-1中给出。

程序示例10-1 信用卡交易（`cardrun.py`）

我们用`safe_float()`函数来处理一组信用卡交易数据，这些数据保存在一个文件里，并将做为字符串读入到程序中。我们用一个记录文件跟踪处理过程。

```
1  #!/usr/bin/env python
2
3  import types
4
5  def safe_float(object):
6      'safe version of float()'
7      try:
8          retval = float(object)
9      except (ValueError, TypeError), diag:
10         retval = str(diag)
11     return retval
12
13 def main():
14     'handles all the data processing'
15     log = open('cardlog.txt', 'w')
16     try:
17         ccfile = open('carddata.txt', 'r')
18     except IOError:
19         log.write('no txns this month\n')
20         log.close()
21     return
22
```

```

23     txns = ccfile.readlines()
24     ccfile.close()
25     total = 0.00
26     log.write('account log:\n')
27
28     for eachTxn in txns:
29         result = safe_float(eachTxn)
30         if type(result) == types.FloatType:
31             total = total + result
32             log.write('data... processed\n')
33         else:
34             log.write('ignored: %s' % result)
35     print '$%.2f (new balance)' % (total)
36     log.close()
37
38 if __name__ == '__main__':
39     main()

```

1~3行

脚本程序启动，导入types模块。这个模块里包含着与各种Python类型对应的Type对象。这也是我们为什么把它们引导到标准错误的原因。

5~11行

这段代码就是我们的safe_float()函数。

13~36行

这个应用程序的核心部分将完成三项主要的工作：(1) 读信用卡数据文件；(2) 对输入数据进行处理；(3) 显示结果。第16~24行完成从文件里读出数据的工作。请注意，文件的打开操作是用try-except语句打包了的。

我们为数据的计算处理过程准备了一个记录文件。在这个例子里，我们假定这个记录文件可以毫无问题地打开并写入数据。我们做的每一项交易计算都保存在这个记录文件里。如果没有找到信用卡数据文件，就假定这个月里没有交易（第18~21行）。

数据被读入txns（交易）列表，然后在程序的第28~34行进行遍历处理。每调用一次safe_float()函数，就用types模块检查一下它的返回结果的类型。types模块内包含着与每一种类型对应的数据项，名字依次为typeType；因此我们可以把这个值与safe_float()函数的结果直接进行比较以判断一个对象的类型。这个例子需要检查safe_float()函数返回的是一个字符串还是一个浮点数。字符串表示因字符串输入无法被转换为一个数字而出现了错误，其余的返回值应该都是浮点数，把这些数值加在一起。计算出的总余额用main()函数的最后一行程序显示出来。

38~39行

这两行表示“如果不是被导入就启动运行”的功能。

运行这个程序后，我们看到的是下面这样的输出：

```

% cardrun.py
$58.94 ( new balance )

```

检查一下计算结果的记录文件（cardlog.txt）。在cardrun.py处理完carddata.txt文件里的交易数据后，我们会看到下面这样的记录项：

```
% cat cardlog.txt
account log;
ignored: invalid literal for float(): # carddata.txt
ignored: invalid literal for float(): previous balance
data... processed
ignored: invalid literal for float(): debits
data... processed
data... processed
data... processed
ignored: invalid literal for float(): credits
data... processed
data... processed
ignored: invalid literal for float(): finance charge/
late fees
data... processed
data... processed
```

10.3.8 else从句

我们已经在Python语言中的其他结构，比如条件和循环里见过else语句了。当else与try-except语句一起使用时，其作用和读者见过的其他情况相比也没有太大的不同：如果在前面的try子句里没有检测到例外，就执行else从句。

开始执行else子句中的代码的前提是try子句中的全部代码都已经成功地执行完了（即在没有任何引发例外的情况下结束了）。下面是用Python伪代码写的一个小例子：

```
import 3rd_party_module

log = open('logfile.txt', 'w')

try:
    3rd_party_module.function()
except:
    log.write("**** caught exception in module\n")
else:
    log.write("**** no exceptions caught\n")

log.close()
```

在上面的例子里，我们导入了一个外来的模块并测试它是否存在错误。我们用一个记录文件来确定这个第三方模块的代码里是否有缺陷。我们将根据外来模块中的函数在执行时有无引发例外而在记录里写下不同的信息。

10.3.9 try-except语句用法总结

我们现在把在这一章里讲过的try-except-else结构的语法以各种不同用法相结合的形式列在下面：

```
try:
    try_suite

except Exception1:
    suite_for_Exception1
```

```

except (Exception2, Exception3, Exception4):
    suite_for_Exceptions_2_3_and_4

except Exception5, Argument5:
    suite_for_Exception5_plus_argument

except (Exception6, Exception7), Argument67:
    suite_for_Exceptions6_and_7_plus_argument

except:
    suite_for_all_other_exceptions

else:
    no_exceptions_detected_suite

```

10.3.10 try-finally语句

try-finally语句与try-except语句不一样，它并不是用来处理例外的。不管try子句里有没有引发例外，引发的例外是哪一个，finally子句都要执行。也就是说，某些个操作肯定会被执行到。

```

try:
    try_suite
finally:
    finally_suite # executes regardless of

exceptions

```

如果try子句里真的引发了一个例外，程序将立刻跳到finally子句去执行。当finally子句中的所有代码都执行完以后，再重新引发刚才的那个例外让高一级的程序流程去处理。因此我们经常会看到try-finally被嵌在一个try-except子句里。

我们可以在cardrun.py程序中加上一个try-finally语句来改进我们的代码，这样就可以捕获从carddata.txt文件读取数据时可能出现的问题。在程序示例10-1给出的代码里，在读数据阶段（即readlines()操作时）没有预备任何错误检测功能，如下所示：

```

try:
    ccfile = open('carddata.txt')
except IOError:
    log.write('no txns this month\n')
    log.close()
    return

txns = ccfile.readlines()
ccfile.close()

```

有许多因素会让readlines()操作失败，其中之一是carddata.txt文件是一个无法访问的网络（或者一张软盘）上的文件。不管怎样，我们应该改进这一段代码，把整个数据输入部分包括在try从句中，就像下面这样：

```

try:
    ccfile = open('carddata.txt')
    txns = ccfile.readlines()
    ccfile.close()
except IOError:

```

```
log.write('no txns this month\n')
log.close()

return
```

我们只是把readlines()方法和close()方法移动到try子句中去。虽然我们的代码现在已经很健壮了,但它还有进一步改进的余地。请注意如果有错误发生会是怎样的情况。如果打开文件的操作成功但readlines()调用因为某些原因没有成功,那么例外处理会从except从句开始继续执行,这样就根本没机会关闭打开的文件。如果不论错误发生与否我们都先把文件关闭,不是很好吗?用try-finally就可以实现这一点,如下所示:

```
try:
    ccfile = open('carddata.txt')
    try:
        txns = ccfile.readlines()
    finally:
        ccfile.close()
except IOError:
    log.write('no txns this month\n')
    log.close()

return
```

我们的代码现在更健壮了。现在再来看看另一个熟悉的例子:用一个非法输入值调用float()函数。我们将用print语句演示try-except和try-finally从句中的程序执行流程。tryfin.py程序的代码请见程序示例10-2。

程序示例10-2 测试try-finally语句 (tryfin.py)

这个小脚本程序简单地演示出在一个try-except语句的try从句里嵌入一个try-finally从句时的控制流走向。

```
1  #!/usr/bin/env python
2
3  try:
4      print 'entering 1st try'
5      try:
6          print 'entering 2nd try'
7          float('abc')
8
9      finally:
10         print 'doing finally'
11
12 except ValueError:
13     print 'handling ValueError'
14
15 print 'finishing execution'
```

运行这段代码,我们将看到下面这样的输出:

```
% tryfin.py
entering 1st try
entering 2nd try
doing finally
handling ValueError
finishing execution
```

最后的提示：如果finally子句中的代码引发了另外一个例外，或者因为遇到了return、break和continue语句而半途退出了，那个原来的例外就会丢失，不能重新引发出来。快速总结：try-finally语句检测出错误后并不处理它而是忽略它，把这个例外传递到上一层去处理。

编程提示：在一个try语句里使用continue

[CN]

目前，在try子句里还不允许使用continue语句，因为现在使用的Python字节码发生器还不允许这样做（请参考FAQ 6.28）。但在JPython里这个性质已经有所松动。

一个比较好的解决办法是用一个if-else来代替continue。一个更有趣的办法是在一个特殊的例外处理器里面发出continue命令（因为continue语句允许用在except从句里），可以使用如下面这段Python伪代码所示：

```
# create our own exception (see section 10.9)
class Continue(Exception):
    pass

# begin our loop
some_loop:                # pseudocode for a loop

    # try clause inside some_loop
    try:
        if skip_rest_of_loop_expr:
            raise Continue

        ...code we do not want executed
        if skip_rest_of_loop_expr is true...

    except Continue:      # continue proxy (as except clause)
        continue         # start next some_loop iteration

    except SomeError:     # handle real exceptions
        :
```

我们将在本章后面的内容里介绍raise语句，但读者可能已经猜到raise语句就是程序员在Python语言里用来明确地引发一个例外的语句。

10.4 *例外的字符串形式

[1.5]

在Python 1.5版本之前，标准的例外都被实现为字符串。但这个办法局限性很大，因为它不允许各例外之间彼此有什么联系。但在引入例外类（exception class）之后，情况就不一样了。从Python 1.5版本开始到现在，所有标准的例外都已经是类（class）了。虽然还会有程序员仍然把自己引发的例外做为字符串，但我们建议从现在开始使用例外类。

[1.6]

为了兼容过去的做法，现在还允许使用字符串形式的例外。在启动Python解释器的时候加上命令行参数-X就能使用字符串形式的标准例外。这个功能从Python 1.6版本开始基本上就没有人再使用了。

如果不得不使用字符串形式的例外，可以采用我们下面介绍的办法做好它。下面的代码有可能在你的系统上无法应用：

```
# this may not work... risky!
try:
```

```

        :
        raise 'myexception'
        :
except 'myexception':
    suite_to_handle_my_string_exception
except:
    suite_for_other_exceptions

```

上面这段代码之所以可能无法使用是因为例外是基于对象的标识而不是基于对象的值的(请参考10.5.1节中的内容)。上面的代码里使用了两种字符串对象,但两者的值都是一样的。为了避免可能发生的意外,我们创建一个静态字符串对象来使用,如下所示:

```

# this is a little bit better
myexception = 'myexception'
try:
    :
    raise myexception
    :
except myexception:
    suite_to_handle_my_string_exception
except:
    suite_for_other_exceptions

```

虽然做了改进后,我们使用的字符串与原来相比还是一样的。如果你真的要使用这段代码,最好还是使用一个例外类,即把上面的myexception赋值语句替换为:

```

# this is the best choice
class MyException(Exception):
    pass
    :
try:
    :
    raise MyException
    :
except MyException:
    suite_to_handle_my_string_exception
except:
    suite_for_other_exceptions

```

可见,从现在起在创建自己的例外时,实在没理由不使用例外类。但这样做一定要很谨慎,因为你可能会用到一个外来的模块,而它还是使用着字符串形式的例外。

10.5 *例外的类形式

我们在前面已经说过,从Python 1.5版本开始所有标准的例外都已经统一使用类(class)了。用户定义的基于类的例外已经出现了很长时间了(从Python 1.2版本就有了!),但在Python 1.5版本之前,标准的例外还是被实现为字符串,其主要原因是为了与过去的版本兼容。但使用类确实带来了许多好处,并且正是这些理由最终导致了把所有的标准例外都从字符串转换为基于类的形式。

10.5.1 通过对象的标识符进行挑选

对例外处理器的检索查找（检查每一个except从句）是通过对象的标识而非对象的值来完成的。这就意味着如果你使用的是字符串形式的例外，那么在except从句中使用的字符串对象就必须和被引发的字符串例外完全一致。如果它们是两个不同的字符串对象，即使包含的是完全相同的字符串，代表的也是不同的例外！

而使用类就简化了这个挑选机制，因为例外类在大多数情况下都是静态的。当你引用一个例外的时候，你实际访问的那个类对象其实是一个在整个执行过程中保持不变的内置标识符。不管是在一个except从句还是在一个raise语句里，只要是使用了IndexError，就能够确定它们引用的都是同一个类对象，这样就能够找到与之对应的例外处理器。

10.5.2 例外之间的关系

使用类还允许例外之间的继承关系得以保留下来。这种结构有两种使用方法：

1. 鼓励把相关的例外分为一个组

在例外还是简单的字符串的年代，任意两个错误之间是没有内在联系的。虽然大多数错误本来就没有什么联系，但有些错误确实具有相当紧密的关系，比如表示出现非法序列下标的IndexError和表示出现非法映射键字的KeyError就有很紧密的联系。字符串形式的例外只允许这些例外在上下文或者定义描述里建立关系，除了从代码方面体现出来的关系之外再也没有其他的了。

基于类的例外就允许这种关系的存在。两种例外现在是从同一个祖先LookupError例外里推导出来的两个子类了。如果在用户的应用程序里定义了一个新的与LookupError例外有关系的类，就可以很容易地再创建一个相关的例外，只要再从LookupError例外，甚至再从IndexError或KeyError例外里简单地分离出一个子类就行了。

Python例外及其类继承关系的完整清单可以在表10-2里找到。

2. 简化了例外的检测操作

有了基于类的例外，处理器代码就能够对一个完整的例外类“树”（即一个祖先级别的例外类和从它推导出来的所有子类）进行检测。举个例子：假设你想在自己的程序里捕获所有普通的算术运算错误，那么有关代码就应该是如下所示的结构：

```
try:
    code_to_scan_for_math_errors
except FloatingPointError:
    print "math exception found"
except ZeroDivisionError:
    print "math exception found"
except OverflowError:
    print "math exception found"
```

因为每个例外的处理器都是相同的，所以我们可以把这段代码缩短为：

```
try:
    code_to_scan_for_math_errors
except (FloatingPointError, ZeroDivisionError, OverflowError):
    print "math exception found"
```


但这个解决方案并不象看上去那样全能，一下子列出三个例外多少有些混乱，并且它还没有考虑今后的扩展。如果下一版本的Python增加了新的算术例外，或者读者为自己的应用程序创建了一个新例外又该怎么办呢？我们刚才编写的代码将因为不够准确而被淘汰。

这个问题的解决办法是在你的except从句里引用一个基本的类。因为你的新例外（比如FloatingPointError、ZeroDivisionError以及OverflowError等）都是从ArithmeticError例外类里推导出来的子类，所以你最好的办法是引用那个能够检测全部的ArithmeticError例外以及从ArithmeticError例外类里推导出来的例外的ArithmeticError例外类。再改进一次我们的代码，下面是最具灵活型的解决方案了：

```
try:
    code_to_scan_for_math_errors
except ArithmeticError:
    print "math exception found"
```

现在，你的代码将能够处理所有已知的ArithmeticError例外和你可能会从ArithmeticError里推导出来的子类中的例外了。但在使用同一个try语句处理类及其母类两种事物的时候一定要很谨慎。请看下面的两个例子：

```
try:
    code_to_scan_for_math_errors
except ArithmeticError:
    print "math exception found"
except ZeroDivisionError:
    print "division by zero error"
```

```
try:
    code_to_scan_for_math_errors
except ZeroDivisionError:
    print "division by zero error"
except ArithmeticError:
    print "math exception found"
```

例外的处理器是排它性的，即一个例外只要找到了一个例外（或者一个基本的例外类）的处理器，就不会再继续查找了，找到的处理器将立即被执行。在第一个例子里，ZeroDivisionError只会被第一个except语句处理并产生一条“math exception found”（发现算术例外）输出信息。专门为ZeroDivisionError编写的except从句永远不会到达。

第二个例子会更实用一些，它会先处理一个特定的算术错误（ZeroDivisionError），让通用性的ArithmeticError处理器负责从ArithmeticError推导出来的其余例外。

10.6 引发例外

到目前为止，我们看到的所有例外都是由解释器负责引发的。这对那些因程序运行期间遇上一个错误而导致的错误结构是正确的。但编写API的程序员可能会希望能在遇到错误的输入数据的时候自己引发一个例外，Python语言特意准备了一个能够让程序员明确地引发一个例外的机制，即raise语句。

raise语句

raise语句和它携带的参数具有很好的灵活性，它的参数可以被转换为许多语法上支持的不同格式。raise语句的一般语法如下所示：

```
raise [ Exception [, args [, traceback ] ] ]
```

第一个参数，Exception，是将要引发的例外的名字。如果有的话，它必须是一个字符串、一个类或者一个实例（下面还有很多）。如果还需要给出其他的参数（变参或traceback参数），Exception就是必不可少的。Python标准例外的完整清单见表10-2。

第二个表达式args里包含着该例外可选的参数（包括该例外的各种参数和值）。这个值是一个单个的对象或者是一个对象的表列。只要检测到例外，它的参数就会以一个表列的形式被返回。如果args是一个表列，那么这个表列包含的就是将传递到该例外的处理器的那组参数。如果args是一个单个的对象，那么表列里也就只包含那一个对象了（即一个只有一个元素的表列）。在大多数情况下，单独的一个参数是由说明该错误原因的那个字符串构成的。如果返回的是一个表列，那它通常会由以下几个部分组成：一个错误信息字符串、一个错误编号，也许还会有发生错误的位置——比如一个文件等。

最后一个参数traceback也是可选的（实际工作中很少用到），如果有的话，它就是该例外用到的那个traceback对象——引发一个例外的时候通常会新建一个traceback对象。如果用户打算重新引发一个例外（比如说想从当前位置指向前一个位置），这第三个参数是很有用的。没有给出的参数用值None来表示。

最常用的语法是Exception做为一个类的时候，此时其他额外参数并不是必需的。但如果在这种情况下真的给出了它们，那它们可以是一个单个的对象、一个由参数组成的表列，或者是一个例外类的实例。如果这个参数是一个实例，那它可以是一个给定类或者一个推导类（从一个现有例外类里分离出来的子类）的实例。如果这个参数是一个实例，就不允许再有其他的参数（即例外参数）了。

如果这个参数是一个实例会发生什么样的事情呢？如果instance是一个给定的例外类的实例，就不会出现什么问题。但如果instance不是某个类的实例或者不是该类的某个子类的一个实例，就会创建出该例外类的一个新的实例来，而新实例的参数是从给定类那里复制来的。如果instance是一个例外类的子类的实例，那个新例外就会从该子类而不是从最初的母类里产生出来。

在raise语句里，如果和一个例外类一起使用的额外参数不是一个实例而是一个单个元素或表列，就将从那个类里产生一个实例，并将把args用做该例外的参数表。如果没有给出第二个参数或者第二个参数是None，参数表就将是空的。

如果Exception是一个实例，就不需要我们再实例化什么东西了。在这种情况下，额外参数不必给出或者是None。例外的类型（type）就是instance属于的那个类（class）；换句话说，它相当于用这个实例引发了那个类，即相当于“raise instance.__class__, instance”。

使用字符串形式的例外不如使用例外类形式的例外，可万一Exception是一个字符串，那它引发的就是由string标识的例外，并且会用可选参数args（如果有的话）做为它的参数。

最后，不带任何参数的raise语句本身是从Python 1.5版本里开始引入的一个新结构，它的作用是再引发一次本代码块里引发出来的上一个例外。如果前面没有引发过例外，它就引发一个TypeError例外，因为没有以前的例外可供再次引发。

因为raise有许多不同的语法格式（即Exception可以是一个类、一个实例或者一个字符串），我们在表10-1里列出了raise各种不同的使用方法。

表10-1 raise语句的使用方法

raise语法	说 明
<code>raise exclass</code>	引发一个例外，创建一个exclass（不带任何例外参数）的实例
<code>raise exclass()</code>	同上，因为现在把类用做例外；用函数调用操作符（即一对圆括号）调用一个类来产生exclass的一个实例，也不带参数
<code>raise exclass, args</code>	同上，但带例外参数args；args可以是单个的参数或一个表列
<code>raise exclass(args)</code>	同上。
<code>raise exclass, args, tb</code>	同上，但提供traceback对象tb供用户使用
<code>raise exclass, instance</code>	用instance（它通常是exclass的一个实例）引发一个例外；如果instance是exclass的某个子类的一个实例，那么这个新例外的类型就将是那个子类的类型（不是exclass类型）；如果instance既不是exclass的一个实例，也不是exclass的某个子类的一个实例，那将创建一个新的exclass实例，而这个新实例的例外参数都是从instance处复制来的
<code>raise instance</code>	用instance引发一个例外；该例外的类型就是推导出instance的那个类的类型；它相当于“raise instance.__class__.instance”（与刚才的情况相同）
<code>raise string</code>	（老办法）抛出一个string字符串例外
<code>raise string, args</code>	同上，但引发例外时带args参数
<code>raise string, args, tb</code>	同上，但提供traceback对象tb供用户使用
<code>raise</code>	（新出现于Python 1.5版本）重新引发一个曾经引发过的例外；如果此前没有引发过例外，就引发一个TypeError例外

10.7 确认

确认是诊断性的前提条件，它的值必须取为布尔真值；否则就会引发一个例外指出该表达式是不成立的。它与C语言中预处理器里的assert宏命令的作用是相似的，只是在Python语言里它们是程序运行期间的结构罢了（不再是预编译指令）。

如果读者是第一次接触确认这个概念，也没有什么大不了。你可以把确认操作想象为一个raise-if语句（或者更准确一些，想象为一个raise-if-not语句）。它对一个表达式进行测试，如果结果是一个假值，就会引发一个例外。 [1.5]

确认操作是由assert语句实现的，这是从Python 1.5版本开始新引入的一个关键字。

assert语句

assert语句对一个Python表达式进行求值操作，如果确认成功就不会采取什么动作（类似于一个pass语句），否则就会引发一个AssertionError例外。assert语句的语法如下所示：

```
assert expression [, arguments ]
```

下面是使用assert语句的几个例子:

```
assert 1 == 1
assert (2 + 2) == (2 * 2)
assert len(['my list', 12]) < 10
assert range(3) == [0, 1, 2]
```

AssertionError例外可以像其他各种例外那样被try-except语句捕获, 但如果不对它进行处理, 它们就会结束程序的执行并给出一个下面这样的traceback信息:

```
>>> assert 1 == 0
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AssertionError
```

和我们在前一小节里研究过的raise语句一样, 可以给assert命令加上一个例外参数, 如下所示:

```
>>> assert 1 == 0, 'One does not equal zero silly!'
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AssertionError: One does not equal zero silly!
```

下面是如何用try-except语句捕获一个AssertionError例外的代码:

```
try:
    assert 1 == 0, 'One does not equal zero silly!'
except AssertionError, args:
    print '%s: %s' % (args.__class__.__name__, args)
```

从命令行上执行上面的代码将得到如下所示的输出:

```
AssertionError: One does not equal zero silly!
```

为了让大家进一步了解assert语句的工作情况, 可以把assert语句想象为Python语言中的一个函数, 它的样子看上去应该是下面这个样子:

```
def assert(expr, args=None):
    if __debug__ and not expr:
        raise AssertionError, args
```

第一个if语句表达了确认操作的正确语法, 即expr应该是一个表达式。我们可以把expr的类型与一个真正的表达式进行比较来验证这一点。函数的第二个部分对表达式进行求值, 如有必要就引发AssertionError例外。内建变量__debug__在正常情况下是1, 在要求优化的情况下(即带有命令行参数-O的时候)是0。

10.8 标准例外

表10-2列出了Python语言现有的全部标准例外。这些例外都是以内建方式加载到解释器里去的, 所以说它们在用户开始运行自己的脚本程序之前, 或者说在用户看到解释器的提示符之前(如果是交互方式运行的话)就都已经准备好了。

所有标准或内建的例外都是从根类(root class) Exception里推导出来的。Exception现在有两个直接推导出来的子类, 它们是SystemExit和StandardError。所有其他的内建例外都是

StandardError的子类。在表10-2里，每个缩进一级的例外都表示它是从上一级的例外类里推导出来的。

表10-2 Python语言中的标准例外

例外名称	说 明
Exception ^a	一切例外的根类 (root class)
SystemExit	要求终止Python解释器的运行
StandardError ^a	一切标准的内建例外的基类 (base class)
ArithmeticError ^d	一切数值计算类错误的基类
FloatingpointError ^a	浮点计算过程中出现错误
OverflowError	计算超出了数值类型的所能表示的最大范围
ZeroDivisionError	用零做除数的除法 (或求除法余数) 错误 (全体数值类型)
AssertionError ^a	assert语句的确认操作失败
AttributeError	对象的属性不存在
EOFError	到达文件尾标记但仍然没有找到输入数据
EnvironmentError ^b	操作系统环境错误的基类
IOError	输入/输出操作失败
OSError ^b	操作系统错误
WindowsError ^c	MS Windows系统调用失败
ImportError	导入模块或对象时失败
KeyboardInterrupt	用户中止了程序的执行 (通常是用^C)
LookupError ^a	非法数据查找错误的基类
IndexError	序列中不存在这样的下标
KeyError	映射关系中不存在这样的键字
MemoryError	内存越界错误 (对Python解释器来说不是致命的)
NameError	未声明/未初始化的对象 (非属性)
UnboundLocalError ^c	试图访问一个未经初始化的局部变量
RuntimeError	程序运行期间的普通缺省错误
NotImplementedError ^b	不存在的方法
SyntaxError	Python语法错误
IndentationError ^d	不正确的缩进
TabError ^d	制表符 (TAB) 和空格搭配不正确
SystemError	普通的解释器系统错误
TypeError	对类型进行了非法操作
ValueError	给出的参数不正确
UnicodeError ^c	与Unicode有关的错误

① 在Python 1.5版本之前，带此记号的例外都是不存在的。所有早期的例外都是基于字符串的。

② 从Python 1.5.2版本开始出现。

③ 从Python 1.6版本开始出现。

④ 从Python 2.0版本开始出现。

10.9 *创建例外

虽然标准例外的覆盖面已经相当广泛了，但创建自己的例外还是有好处的。情形之一就是打算让一个标准的或者模块化的例外提供更多的信息。我们用两个例子来说明这个问题，它

们都与IOError有关。

IOError是输入/输出有问题时最常遇到的例外，它可能因非法的文件访问或者其他数据通信方面的问题而引发。假设我们想更准确地定位问题的来源。举例来说，如果出现了文件错误，我们想有一个行为类似于IOError的FileError例外，但它的名字从执行文件操作的角度来看应该更有意义。

我们准备说明的另一个例外与通过套接字(socket)进行的网络编程有关。socket模块引发的例外被称为socket.error，它不是一个内建的例外。它是从基本的Exception例外里推导出来的子类。但来自socket.error的例外参数与那些来自IOError例外的非常相似，所以我们将定义一个名为NetworkError的新例外；它将是一个从IOError里推导出来的子类，但至少要包含一些由socket.error提供的信息。

到目前为止，我们还没有正式介绍过类(class)和面向对象的程序设计方法，也没有正式介绍过网络方面的程序设计，如果读者需要了解这方面知识，请跳到第16章去学习。

我们现在就给出一个名为myexc.py的模块，在这个模块里将用到刚才定制的FileError和NetworkError例外。请看下面的程序示例10-3。

程序示例10-3 创建例外 (myexc.py)

这个模块定义了两个新的例外，它们分别是FileError和NetworkError，并用它们重新编写了带诊断功能的open() [myopen()]函数和socket.connect() [myconnect()]函数。此外还有一个将在直接执行此模块时运行的测试函数[test()]。

```

1  #!/usr/bin/env python
2
3  import os, socket, errno, types, tempfile
4
5  class NetworkError(IOError):
6      pass
7
8  class FileError(IOError):
9      pass
10
11 def updArgs(args, newarg=None):
12
13     if type(args) == types.InstanceType:
14         myargs = []
15         for eachArg in args:
16             myargs.append(eachArg)
17     else:
18         myargs = list(args)
19
20     if newarg:
21         myargs.append(newarg)
22
23     return tuple(myargs)
24
25 def fileArgs(file, mode, args):
26
27     if args[0] == errno.EACCES and \
28         'access' in dir(os):
29         perms = ''
30         permd = { 'r': os.R_OK, 'w': os.W_OK,
31                   'x': os.X_OK}
32         pkeys = permd.keys()
33         pkeys.sort()
34         pkeys.reverse()

```

```

35
36     for eachPerm in 'rwx':
37         if os.access(file, perm[d[eachPerm]]):
38             perms = perms + eachPerm
39         else:
40             perms = perms + '-'
41
42     if type(args) == types.InstanceType:
43         myargs = []
44         for eachArg in args:
45             myargs.append(eachArg)
46     else:
47         myargs = list(args)
48
49     myargs[1] = "%s" %s (perms: '%s')" % \
50         (mode, myargs[1], perms)
51
52     myargs.append(args.filename)
53
54 else:
55     myargs = args
56
57     return tuple(myargs)
58
59 def myconnect(sock, host, port):
60
61     try:
62         sock.connect((host, port))
63
64     except socket.error, args:
65         myargs = updArgs(args) # conv inst2tuple
66         if len(myargs) == 1:    # no #s on some errs
67             myargs = (errno.ENXIO, myargs[0])
68
69         raise NetworkError, \
70             updArgs(myargs, host + ':' + str(port))
71
72 def myopen(file, mode='r'):
73
74     try:
75         fo = open(file, mode)
76
77     except IOError, args:
78         raise FileError, fileArgs(file, mode, args)
79
80     return fo
81
82 def testfile():
83
84     file = mktemp()
85     f = open(file, 'w')
86     f.close()
87
88     for eachTest in ((0, 'r'), (0100, 'r'), \
89                     0400, 'w'), (0500, 'w')):
90         try:
91             os.chmod(file, eachTest[0])
92             f = myopen(file, eachTest[1])
93
94         except FileError, args:
95             print "%s: %s" % \
96                 (args.__class__.__name__, args)
97     else:
98         print file, "opened ok... perm ignored"
99         f.close()
100

```

```

101     os.chmod(file, 0777)# enable all perms
102     os.unlink(file)
103
104     def testnet():
105         s = socket.socket(socket.AF_INET, \
106                           socket.SOCK_STREAM)
107
108         for eachHost in ('deli', 'www'):
109             try:
110                 myconnect(s, 'deli', 8080)
111             except NetworkError, args:
112                 print "%s: %s" % \
113                       (args.__class__.__name__, args)
114
115     if __name__ == '__main__':
116         testfile()
117         testnet()

```

1~3行

UNIX操作系统下的程序启动命令和帮助我们启动此模块的socket、os、errno、types、tmpfile等模块的导入命令。

5~9行

不管你相信与否，就是这五行程序构造出了我们的新例外。不只是一个，而是两个都构造出来了。除非需要引入新功能，否则创建一个新例外也就是从一个现有的例外再推导出一个子类来。在这个模块里，现有的例外就是IOError。从IOError中推导出来的EnvironmentError也够用，但这样做是为了表明我们的例外确实是与I/O有关的。

选择IOError的原因是它能够两个参数，一个是错误编号，另一个是错误信息字符串。（因为使用了open()函数）而与文件有关的IOError例外甚至还支持第三个参数，这第三个参数并不是例外经常用到的东西，它是一个文件名。对这个存在于主表列对以外的第三个参数要采用特殊的处理，我们给它取名为filename。

11~23行

updArgs()函数的全部功能就是“刷新”例外的参数。原来的例外会向我们提供一组参数，而我们想把这些参数添加到新例外里去，比如说添上那个第三个参数（如果没有给出的话就不添加什么东西——None是一个缺省的参数，我们将在下一章研究缺省参数的问题）。我们的目的是向用户提供尽可能多的详细资料，这样在错误发生时就能够尽快地找到问题的根源。

25~57行

只有myopen()（请参考后面的内容）才会来调用fileArgs()函数。说白了，我们要找的是EACCESS错误，它表示“permission denied”（权限不够）；我们放过其他的IOError例外不去修改（第54~55行）。如果读者想进一步了解ENXIO、EACCESS或者其他系统错误的编号，在UNIX系统上可以查阅/usr/include/sys/errno.h文件，在Windows系统上使用C语言Visual C++时可以查阅C:\Msdev\include\Errno.h文件。

在第27行里，还需要确定我们正在使用的计算机支持os.access()函数，这个函数能够帮助用户查出自己对某个特定的文件拥有怎样的访问权限。在没有遇到访问权限错误和查出我们拥有的访问权限之前不采取进一步的行动。查出访问权限后，我们会创建一个字典来帮助我们建立

一个代表着我们对那个文件的访问权限的字符串。

UNIX操作系统的文件系统能够对用户、组（一个组里可以有不止一个用户）、其他用户（除文件属主和与该属主同组的其他人以外的所有用户）按读、写、执行（r、w、x）的顺序明确地分配文件的访问权限。Windows只支持这些权限中的一部分。

现在开始构造访问权限字符串。如果文件有一个权限，它的对应字母就会出现在一个字符串里，否则就会出现一个短划线字符（-）。举例来说，字符串“rw-”表示用户对文件拥有读和写的权限；而字符串“r-x”表示用户只有读和执行权限；“---”表示用户什么权限也没有。

构造好访问权限字符串之后，我们创建一个临时的参数列表。我们可以让错误信息字符串包含访问权限字符串的内容，这是一些没有在标准的IOError例外里提供的信息。如果你不知道自己对文件拥有怎样的权限，就无法改正错误；而在这种情况下，系统只给出了一条“permission denied”（权限不够）的信息当然是没有什么实际意义的。这种做法是出于安全性方面的考虑。在入侵者没有访问权限的时候，你最不想让他们看见的东西就是文件的访问权限了，而这就是症结所在。这个例子只是一个练习，所以我们允许临时性地出现“安全漏洞”。我们的目的是验证os.chmod()函数调用能否按照我们的意愿改变文件的访问权限。

最后一件事是在参数列表里加上文件名，并把这组参数以表列的形式返回给调用者。

59 ~ 70行

我们的新myconnect()函数只是简单地对标准的套接字方法connect()进行了打包，让它在网络连接失败的情况下给出一个IOError类型的例外。和正常socket.error不同的是我们还把主机名和端口号做为额外的信息提供给程序员。

对那些不熟悉网络编程的人们来说，主机名和端口号就象是你给别人打电话时要拨的长途区号和电话号码。在这个例子里，我们试图连接到远程主机上运行着的某个程序去，也就是连接到某个服务器程序上去；因此需要知道主机名和那个服务器正在监听着的端口号。

在发生错误的时候，错误编号和错误信息字符串是很有帮助的，要是再有准确的主机名和端口号就更好了，因为这两个信息通常都是动态生成或者从某个数据库或域名服务中查到的。这就是在改进后的connect()里将要实现的增值服务。如果主机没有找到还会出现另外一个问题，那就是socket.error例外将无法给出一个直接的错误编号；因此，要想让它也按照IOError的做法给出一组“错误编号-错误信息字符串”来，就需要找到一个与之最接近的错误编号——我们的选择是ENXIO。

72 ~ 80行

和myconnect()一样，myopen()也打包了一个现有的代码段，这里用的是open()函数。我们编写的例外处理器只捕获IOError例外，其他的都不加处理地传递到上一级去（因为找不到与之对应的例外处理器）。如果捕获到一个IOError，我们就引发自己的例外，并把修改后的参数从fileArgs()函数里返回给调用者。

82 ~ 102行

我们先要进行文件测试，这里使用的是testfile()函数。为此需要先创建一个测试用的文件，我们可以修改这个文件的访问权限以产生访问权限错误。tempfile模块里包含着产生临时文件名或临时文件本身的代码。我们的程序只需用到一个临时的文件名，然后用那个新的myopen()函

数创建一个空文件。需要注意的是：如果在这一阶段出现了错误是没有相应的例外处理器的，而程序也将会无疾而终——如果我们连一个测试用的文件也创建不出来的话，测试程序是不会再往下执行的。

我们的测试使用了四种不同的访问权限配置。0表示什么权限也没有；0100表示只有执行权限；0400表示只读；而0500表示只有读和执行权限（0400+0100）。无论哪种情况，我们都会尝试以一个非法的模式打开那个文件。os.chmod()函数将负责修改文件访问权限的模式。（请注意：这些访问权限的前面都有一个前缀的0，表示它们是一个八进制数字。）

如果出现了错误，我们希望能够比照Python解释器遇到未捕获例外时的处理办法，用同样的方式把自己的诊断信息显示出来，也就是要在例外名的后面给出它的参数。特殊变量__class__的作用是提供一个类对象(class object)，我们可以从它开始创建一个实例。我们不打算在程序里给出这个类的全名(myexc.FileError)，而是使用这个类对象的__name__变量只给出这个类的名字(FileError)，这也正是解释器遇到未处理错误情况时向用户显示的东西。紧接着是我们在自己的打包函数里放到一起去的那些例外参数。

如果文件成功地打开了，就意味着因为某种原因忽略了访问权限。在这种情况下，我们会给出一条错误信息后关闭文件。全部测试结束后，我们将设置好文件的全部访问权限，再用os.unlink()函数删除它。os.remove()函数的作用与os.unlink()是一样的。

104~113行

接下来的代码(testnet())测试我们的NetworkError例外。一个套接字(socket)就是一个通信结点，与另外一个主机的联系需要通过它才能建立。我们将创建一个这样的对象，然后用它去尝试连接一个没有服务器来接受我们连接请求的主机和一个不在我们网络上的主机。

115~117行

我们只想让这个脚本程序在被直接执行的时候才执行test*()函数，这也正是这段代码要做的工作。本书里面的大多数脚本程序都是以这种方式工作的。

在一台UNIX机器上运行这个程序，我们将得到如下所示的输出：

```
% myexc.py
FileError: [Errno 13] 'r' Permission denied (perms: '---'):
'/usr/tmp/@18908.1'
FileError: [Errno 13] 'r' Permission denied (perms: '--x'):
'/usr/tmp/@18908.1'
FileError: [Errno 13] 'w' Permission denied (perms: 'r--'):
'/usr/tmp/@18908.1'
FileError: [Errno 13] 'w' Permission denied (perms: 'r-x'):
'/usr/tmp/@18908.1'
NetworkError: [Errno 146] Connection refused: 'deli:8080'
NetworkError: [Errno 6] host not found: 'www:8080'
```

在一台Windows机器上运行这个程序，结果将稍有不同：

```
D:\python>python myexc.py
C:\WINDOWS\TEMP\~-195619-1 opened ok... perms ignored
C:\WINDOWS\TEMP\~-195619-1 opened ok... perms ignored
FileError: [Errno 13] 'w' Permission denied (perms: 'r-x'):
'C:\\WINDOWS\\TEMP\\~-195619-1'
FileError: [Errno 13] 'w' Permission denied (perms: 'r-x'):
```

```
'C:\\WINDOWS\\TEMP\\~-195619-1'  
NetworkError: [Errno 10061] winsock error: 'deli:8080'  
NetworkError: [Errno 6] host not found: 'www:8080'
```

读者可能会注意到Windows不支持文件上的读权限，这也是为什么前两个文件的打开操作会成功的原因。在读者自己的计算机和操作系统上，这段程序执行的结果可能会与我们这里给出的不一样。

10.10 为什么会发生例外

只要还有软件，就肯定还会有错误。在这个飞速发展的计算机世界里，我们程序执行的环境发生了变化，在让错误处理准确地反映软件的操作上下文环境方面也要随之发生变化。那些现代的应用程序一般都是以自我包容的图形化用户操作界面（graphical user interface，简称GUI）方式运行的，或者运行在像Web这样的客户-服务器体系结构中。

应用程序水平上的错误处理能力如今正变得越来越重要，因为直接运行应用程序的人已不仅仅是用户了。随着因特网和在线电子商务的发展，Web服务商将是应用软件主要的使用者。这就意味着应用程序不能动不动就失效或者崩溃了，因为一旦发生这样的情况，系统错误将引起浏览器错误，连锁反应下会引起用户们的不满。损失吸引力就意味着损失了广告收益和潜在的无法挽回的大把商业机会。

如果真的出了问题，一般也源于某些非法的用户输入。程序的执行环境必须健壮到足以处理应用程序级的错误和产生用户级错误信息的程度。对Web服务商来说，这句话必须理解为“无错误”，因为应用程序必须要顺利地执行完毕；即使应用程序的全部工作只是以一个正确的超文本标签语言（HTML）网页的形式向用户返回并显示一个错误信息，其自身也必须成功地执行完毕。

如果你不太明白我在说些什么，那只显示着大黑体字“Internal Server Error”（内部服务器出错）的浏览器白屏你见没见过？一个用弹出菜单告诉你“Document contains no data”的致命错误见没见过？做为一个用户，这些对你都意味着什么呢？没什么，确实也没什么（除非你本人就是一名因特网软件工程师）；但对中等水平的用户们来说，它们是无休无止的困惑和烦恼的源头。这些错误信息表示某个应用程序的执行失败了。应用程序或者返回无效的超文本传输协议（hypertext transfer protocol，HTTP）数据，或者致命地结束了运行，从而导致Web服务器举手说“我投降！”

只要有可能，这些错误的执行过程是不能被允许的。随着系统复杂性的提升和初学者用户的增加，必须进一步采取措施以保证用户能够获得一个顺利的应用程序经验。即使面对着一个错误情况，应用程序也必须成功地结束运行，这样做是为了防止给它的执行环境带来一种灾难性影响。Python的例外处理促进了成熟和正确的程序设计。

10.11 为什么要有例外

如果上一小节的论述还不够生动，可以想象一下如果没有了程序级的错误处理Python的程序设计会出现怎样的后果。首先想到的是客户端程序员在对他们代码控制方面的损失。举例来说，

如果你编写了一个涉及和处理大量资源的交互式应用程序，如果一位用户敲入了^C或者其他键盘中断命令，那你的应用程序可能根本没有机会完成清理工作，从而导致数据丢失或者数据崩溃。有时连采取缓冲措施（比如说询问用户他们是真的想退出还是无意中按下了控制键）的机会都没有。

另外一个缺陷是函数不得不重新编写以便在面对一个出错情况时能够返回一个“特殊的”值——比如说None。工程师必须负责检查来自某个函数调用的每一个返回值。这将是一个非常繁重的工作，因为你将不得不对那些与没有遇到错误时预期返回的对象的类型不一样的返回值进行检查。而且，如果需要在函数里把None做为一个合法的数据值时又该怎么办？即使解决了这个问题，又会遇到另外一个返回值，比如说一个负数。用不着我们提醒你就该知道负数在一个Python上下文里可能是合法的，比如序列里的下标就可以是一个负数。做为一个应用程序程序员接口（application programmer interface, API）的程序设计人员，你将不得不对你的用户根据其输入而可能会遇到的每一个错误的返回值加以说明。此外，要想在多层代码里向上移交错误（及其原因）是很困难（也是很繁琐）的。

没有像例外那样更简单的移交办法了。因为错误数据需要在调用继承关系里逐层向上移交，因此在整个过程中难免会对错误做出歧义性的解释。一个完全不沾边的错误可能会被当作问题的原因，可事实上它却可能与最初的问题一点关系也没有。在这个逐层移交的过程中，我们可能错失那些由例外提供给我们的对错误进行封闭和安全处理的机会，更不用说可能会完全失去我们最初关心的数据的踪影了！例外不仅简化了代码，还简化了整个的错误管理机制，其实这个机制并不应该在应用程序的开发中占太大的地位。而有了Python语言为大家提供的例外处理机制后，也确实没让它占很大的地位。

10.12 例外和sys模块

要想获取例外信息还有一种办法是使用sys模块中的exc_info()函数。这个函数返回三个表列的信息，比单纯使用例外参数要多一些。我们来看看用exc_info()函数能得到些什么东西吧：

```
>>> try:
...     float('abc123')
... except:
...     import sys
...     exc_tuple = sys.exc_info()
...
>>> print exc_tuple
(<class exceptions.ValueError at f9838>, <exceptions.ValueError instance at 122fa8>,
<traceback object at 10de18>)
>>>
>>> for eachItem in exc_tuple:
...     print eachItem
...
exceptions.ValueError
invalid literal for float(): abc123
<traceback object at 10de18>
```

我们从exc_info()函数得到的三个表列中的信息是:

- 例外类对象。
- (这个)例外类的实例对象。
- traceback对象。

前面两个我们比较熟悉, 它们分别是实际的例外类和这个特定的例外的实例(这与我们前一小节里介绍的例外参数是一样的)。第三项的traceback对象是新的。这个对象给出了例外发生地点的执行环境指示。它包括的信息有程序运行时的执行框架和出现例外的程序行编号。

在Python语言的早期版本里, 这三个值是以sys模块中的sys.exc_type、sys.exc_value和sys.exc_traceback的面目出现的。但这三个值被用做全局变量, 多线程化时的安全性比较差。因此我们建议用sys.exc_info()函数代替它。

10.13 相关模块

[2.0]

在模块Lib/exceptions.py中找到的类(class)会在启动时做为解释器的内建名字自动加载上, 所以对这个模块就不必明确地进行加载了。我们建议你阅读一下它的源代码使自己熟悉Python语言中的例外及它们彼此之间的内在联系和内部作用。从2.0版本开始, 例外已经被内建于解释器中了(请参考Python/exceptions.c文件)。

10.14 练习

10-1 引发例外。在程序执行期间, 下面这些个因素哪个能够引发例外? 注意这个问题问的不是引发例外的原因。

- a) 用户。
- b) 解释器。
- c) 程序。
- d) 以上所有因素。
- e) 只有b) 和c)。
- f) 只有a) 和c)。

10-2 引发例外。根据上一问题中的清单, 哪些因素会在交互式解释器中引发例外?

10-3 关键字。说出用来引发例外的关键字。

10-4 关键字。try-except和try-finally之间有什么区别?

10-5 例外。请说出下面这段Python代码如果在交互式解释器里执行会引发什么例外(请参考表10-2中给出的内建例外清单):

- a)

```
>>> if 3 < 4 then: print '3 IS less than 4!'
```
- b)

```
>>> aList = ['Hello', 'World!', 'Anyone', 'Home?']
>>> print 'the last string in aList is:', aList[len(aList)]
```
- c)

```
>>> x

d)
>>> x = 4 % 0

e)
>>> import math
>>> i = math.sqrt(-1)
```

10-6 改进open()。请编写一个open()函数的打包函数。如果某个程序成功地打开了一个文件，就返回一个文件句柄；如果打开文件的操作失败了，请向调用者返回一个None而不是产生一个错误，这样好让它们能够在没有例外处理器的情况下打开文件。

10-7 例外。下面两段Python伪代码a)和b)有什么区别？注意要考虑语句A和B的上下文环境，因为这两段代码就是由它们构成的（这么细致的区别要感谢Guido先生）。

```
a)
try:
    statement_A
except ...:
    ...
else:
    statement_B

b)
try:
    statement_A
    statement_B
except ...:
    ...
```

10-8 改进raw_input()函数。我们在本章的开始给出了一个“安全版本”float()内建函数，其作用是检测和处理两种由float()产生的例外。同样，raw_input()函数也会产生两种不同的例外，它们分别是由文件尾标记（即EOF标记）和中止输入操作所引发的EOFError或KeyboardInterrupt。请编写一个打包函数，比如说safe_input()，让这个函数在用户输入EOF（在UNIX里是^D，在DOS里是^Z）或者试图用^C中止输入时不引发一个例外，而是返回一个None供调用函数检查。

10-9 改进math.sqrt()函数。math模块里有许多与完成各种数学计算操作有关的函数和常数。但这个模块不能识别和处理复数，这也是为什么又开发了cmath模块的原因。与其忍受为复数而完整地导入一个原本不打算用在自己程序里的模块，不如使用本身就能够对复数进行操作的标准的数学操作符。但这就需要有一个能够求出一个给定负数的复数平方根结果的平方根函数。请编写出这样的一个函数来，比如说safe_sqrt()。让这个函数包住math.sqrt()函数，它必须能够对一个负数参数进行处理并向调用者返回一个正确的复数平方根结果。

第11章 函 数

我们在第2章介绍过函数之后已经看过不少它们的编写和调用情况了。我们将在本章里放眼全局，把关于函数的方方面面详细地介绍给大家。除预期行为外，Python语言中的函数还支持包括函数化程序设计接口在内的许多创新风格和参数类型。我们将以对Python变量的作用范围和递归函数的讨论结束本章的学习。

11.1 什么是函数

函数是程序中对逻辑进行组织的一种结构化和过程化的程序设计方法。大段的代码可以干净利索地分割为便于管理的小块，可能会经常重复使用的代码被保存到函数里而不是散布于各处的多个拷贝，这样能够节省大量的空间，也便于程序的维护，因为只修改一份拷贝总比到处寻找和修改重复代码的多个拷贝省时省力得多。Python语言中的函数从根本上讲与大家所熟悉的其他语言没有太大的差别。本章的前半部分主要是一些基础性的介绍，然后看看Python语言给我们带来了什么不一样的东西。

函数可以以许多种面目出现，下面是函数在创建、使用或者引用时大家可能会看到的一些情形：

<i>declaration/definition</i>	<code>def foo(): print 'bar'</code>
<i>function object/reference</i>	<code>foo</code>
<i>function call/invoke</i>	<code>foo()</code>

11.1.1 函数与过程的比较

人们经常会拿函数去与过程进行比较。这两者都是能够被程序的其他部分调用的实体，但传统意义上的函数或者说“黑箱”（不管它有没有输入参数）都会在完成某些处理工作后向调用者返回一个返回值；有些函数是布尔型的，返回一个“是”或者“否”的回答——也可以更准确地说是分别返回一个非零值或者零值。而与函数进行比较的过程是一些特殊类型的函数，它们没有返回值。大家将在后面的内容里看到，Python中的过程实际也是函数，因为解释器会隐舍地返回一个缺省值None。

11.1.2 返回值和函数类型

函数会向自己的调用者返回一个值，而那些更过程化的将不会明确地返回任何东西。把过程也看做是函数的程序设计语言通常都会给“不返回任何东西”的函数准备一个特殊的类型或者数据值名字。在C语言里，这类函数缺省的返回类型是“void”，表示没有返回值。在Python语言里，与之相当的返回对象类型是None。

下面代码中的hello()函数其行为就像是一个过程，没有任何返回值。如果真的保存了返回值，

大家就会看到它的值是None：

```
>>> def hello():
...     print 'hello world'
>>>
>>> res = hello()
hello world
>>> res
>>> print res
None
>>> type(res)
<type 'None'>
```

另外，与其他语言一样，在Python语言里一次只能返回一个值或者对象。但这里有一个区别：如果返回的是一个包容器类型，那它看上去就好像实际上返回了好几个单个的对象。换句话说，从商店离开时只能拿一样东西，但你可以把它们都放在一个购物袋里合理合法地带出商店。

```
def foo():
    return ['xyz', 1000000, -98.6]

def bar():
    return 'abc', [42, 'python'], "Guido"
```

foo()函数返回一个列表，而bar()函数返回一个表列。因为表列的语法并不要求必须带上圆括号，所以它看上去和返回多个数据项的情况完全一样。如果我们给表列中的数据项正确地加上括号，bar()函数的定义就是下面这个样子：

```
def bar():
    return ('abc', [4-2j, 'python'], "Guido")
```

从返回值的角度来看，有许多办法可以用来保存表列。下面三种保存返回值的办法其效果都是一样的：

```
>>> aTuple = bar()
>>> x, y, z = bar()
>>> (a, b, c) = bar()
>>>
>>> aTuple
('abc', [(4-2j), 'python'], 'Guido')
>>> x, y, z
('abc', [(4-2j), 'python'], 'Guido')
>>> (a, b, c)
('abc', [(4-2j), 'python'], 'Guido')
```

在对x, y, z和a, b, c进行赋值时，每个变量都会根据值返回的顺序得到与之对应的返回值。对aTuple的赋值是把从函数返回的整个表列隐含地全部赋值给这个变量。我们在以前的内容里曾经说过，表列可以被“分解”为一个一个的变量，也可以做为一个整体把它的引用线索直接单独赋值给一个变量（请参考6.17.3节中的内容）。

许多支持函数的程序设计语言都有这样一个规定：一个函数的类型就是它返回值的类型。但在Python语言里却无法给出一个相关的直接类型，这是因为Python语言中类型都是动态地确定的，并且函数可以返回各种类型的值。又因为Python也不允许函数名的重载，所以程序员可以把type()内建函数用做一个代理去处理带有多种“签名”的多重函数声明（即实际上被重载了的

函数的多模型类型，它会随自己的参数转变类型)。

11.2 函数的调用

11.2.1 函数操作符

函数是用大家比较熟悉的圆括号调用的。事实上，有些人把一对圆括号“()”看做是一个两个字符的操作符，即函数操作符。大家应该知道，一个函数的所有输入参数都必需放在两个括号之间。在声明函数时也要使用圆括号来定义那些参数。虽然我们还没有正式学习类(class)和面向对象的程序设计，但大家可能已经注意到函数操作符也可以用于Python语言的类的实例。

11.2.2 关键字参数

“关键字”参数只用于调用函数的时候。它的目的是让调用者在进行函数调用时可以通过参数的名字来标识参数。这一方法允许参数缺失或不按顺序排列，因为解释器能够通过给出的关键字找到参数的值。

下面给出一个简单的例子，假设函数foo()是用下面的伪代码定义的：

```
def foo(x):
    foo_suite    # presumably does so processing with 'x'
```

foo()的标准调用方法: foo(42) foo('bar') foo(y)

foo()的关键字调用方法: foo(x=42) foo(x='bar') foo(x=y)

下面是一个更实际的例子，假设你有一个名为net_conn()的函数，已经知道它需要两个参数，比如说是host和port，即：

```
def net_conn(host, port):
    net_conn_suite
```

很自然，我们可以按照函数定义中正确的参数位置顺序给出参数，如下所示：

```
net_conn('kappa', 8080)
```

host参数得到的是字符串'kapa'，而port得到的是8080。关键字参数允许参数不按顺序排列，但你必须把参数的名字做为一个“关键字”，以便让参数与它们的参数名对应起来，如下所示：

```
net_conn(port=8080, host='chino')
```

关键字参数还可被用在允许参数“缺失”的场合。这种情况出现在使用了缺省参数的函数中，请参考下一小节的内容。

11.2.3 缺省参数

缺省参数是那些在对函数进行定义时带有缺省值的参数。这样，在调用函数时就允许不把一些参数传递给函数，这些没有传递给函数的参数就会使用它们的缺省值。我们将在11.5.2节里进一步讨论缺省参数的问题。

11.3 函数的创建

11.3.1 def语句

函数是用def语句创建的，它的定义语法如下所示：

```
def function_name(arguments):
    "function_documentation_string"
    function_body_suite
```

构成标题行的有def关键字、函数的名字和函数的参数（如果有的话）。def子句的其余部分包括一个可选但我们强烈推荐使用的文档字符串和被定义函数的程序代码主题部分。我们会在书中看到大量的函数定义，下面又是一个：

```
def helloSomeone(who):
    'returns a salutatory string customized with the input'
    return "Hello " + str(who)
```

11.3.2 函数声明与函数定义的比较

在某些程序设计语言里，函数的声明和函数的定义是有区别的。一个函数声明包括函数的名字和函数各参数的名字（传统上还有其类型），但不必给出函数中的任何代码；给出函数具体代码的程序段将被视为是函数的定义部分。

在有这种区别的程序设计语言里，这样做的原因通常是因为函数定义与函数声明分别放在程序代码中不同的地点。Python语言对这两者不加区别，一个函数子句是由声明性质的标题行和紧跟在其后面的定义部分构成的。

11.3.3 向前引用

Python语言与其他一些高级程序设计语言一样，不允许在函数被定义之前引用或者调用它。请看下面这几个例子：

```
def foo():
    print 'in foo()'
    bar()
```

如果我们在这里调用foo()函数，就会因bar()还没有被定义而失败：

```
>>> foo()
in foo()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in foo
NameError: bar
```

现在来定义bar()，把对它的声明放在foo()函数声明的前面：

```
def bar():
    print 'in bar()'

def foo():
```

```
print 'in foo()'
bar()
```

此时再调用foo()就很安全，不会再出现问题了：

```
>>> foo()
in foo()
in bar()
```

事实上，我们甚至可以把foo()放在bar()的前面：

```
def foo():
    print 'in foo()'
    bar()

def bar():
    print 'in bar()'
```

这段代码不会引起向前引用的问题，完全可以顺利执行，如下所示：

```
>>> foo()
in foo()
in bar()
```

这段代码之所以没有出现问题是因为虽然（在foo()函数里）对bar()函数的一个调用出现在对bar()函数进行定义之前，可foo()本身在bar()被定义之前也没有被调用到。换句话说，我们先定义foo()，再定义bar()，然后再调用foo()，但此时bar()已经是存在着的了，所以这个调用会顺利执行。

请注意，foo()函数在没有引发错误的情况下成功地给出了输出。NameError是访问未初始化标识符时肯定会被引发的例外。

11.4 函数可以用做其他函数的参数

在学习C语言时，函数指针的概念是一个比较高级的论题，但在Python语言里就不会这样，因为Python语言中的函数和其他各种对象是完全一样的。它们可以被引用（被访问或者别名给另外一个变量）、可以做为参数被传递到函数中去、可以被当作列表和字典等包容器的元素。把函数与其他对象区别开来的唯一特性是它们是能够被调用的，也就是说，可以通过函数操作符调用它们。（Python语言中还有其他的可调用成分，详情请参考第14章中的内容。）

请注意上一段文字里函数可以被别名给另外一个变量的说法。因为一切对象都是通过引用来传递的，所以函数也不例外。赋值操作实际上是把同一个对象的引用赋值给另外一个变量；如果该对象是一个函数，此对象的所有别名就都是可调用的：

```
>>> def foo():
...     print 'in foo()'
...
>>> bar = foo
>>> bar()
in foo()
```

在我们把foo赋值给bar的时候，也就把同一个函数对象赋值给了bar，所以可以像调用foo()

函数那样调用bar()。一定要弄明白“foo”(函数对象的引用)和“foo()”(函数对象的调用方式)之间的区别。

再前进一步, 我们甚至可以把函数做为参数传递到另一个函数中去调用:

```
>>> def bar(argfunc):
...     argfunc()
...
>>> bar(foo)
in foo()
```

请注意, 函数对象foo被传递到bar()函数中去了。bar()实际上是一个调用了foo()函数的函数(它就像我们前面例子里把foo赋值给bar那样被别名给局部变量argfunc)。现在来看一个更实际的例子numconv.py, 它的代码在程序示例11-1中给出。

程序示例11-1 传递和调用(内建)函数(numconv.py)

这是把函数做为参数进行传递和在另外一个函数中调用它们的更实际一些的例子。这个脚本程序的作用是把一个数字序列用做为参数传递进来的转换函数转换为同一种类型。准确的说, test()函数中传递进了int()、long()和float()函数来完成转换。

```
1  #!/usr/bin/env python
2
3  def convert(func, seq):
4      'conv. sequence of numbers to same type'
5      newSeq = []
6      for eachNum in seq:
7          newSeq.append(func(eachNum))
8      return newSeq
9
10 def test():
11     'test function for numconv.py'
12     myseq = (123, 45.67, -6.2e8, 999999999L)
13     print convert(int, myseq)
14     print convert(long, myseq)
15     print convert(float, myseq)
16
17 if __name__ == '__main__':
18     test()
```

如果运行这个程序, 就会看到如下所示的输出结果:

```
% numconv.py
[123, 45, -620000000, 999999999]
[123L, 45L, -620000000L, 999999999L]
[123.0, 45.67, -620000000.0, 999999999.0]
```

11.5 正式参数

一个Python函数的正式参数的集合包括与该函数在声明时参数清单里的各个参数准确对应的传递到该函数调用形式中的全部参数。这些参数包括全部的固定参数(按正确位置顺序传递到函数中)、关键字参数(按顺序或不按顺序传递, 但带有把它们的值和它们在参数表里面的正确位置相匹配的关键字), 和出现或者不出现在函数调用中的有缺省值的参数。所有这些参数都会

在该函数的本地名字空间里为它们各自的值创建一个名字，函数一开始执行就能够访问这些局部名字。

11.5.1 位置参数

这是一些在形式上我们大家都很熟悉的标准化的参数。位置参数必须按照它们在被调用函数中所定义的精确顺序传递进来。此外，如果其中没有缺省参数（请参考下一小节），传递到一个函数（调用）中的参数个数必须与所定义的个数完全相同，如下所示：

```
>>> def foo(who):          # defined for only 1 argument
...     print 'Hello', who
...
>>> foo()                  # 0 arguments... BAD
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: not enough arguments; expected 1, got 0
>>>
>>> foo('World!')          # 1 argument... WORKS
Hello World!
>>>
>>> foo('Mr.', 'World!')   # 2 arguments... BAD
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: too many arguments; expected 1, got 2
```

foo()函数有一个位置参数，这就意味着对foo()函数的任何调用都必须精确的拥有一个参数，既不能多，也不能少；否则就会频频看到TypeError错误信息了。请注意Python语言中的错误提供了多么多的信息。做为一条基本原则，只要是调用一个函数，就必须给出它全部的位置参数。它们必须按顺序或不按顺序给出。不按顺序给出指的是有关键字参数的情况，它能够把自身与它自己在参数表中的正确位置匹配起来（请参考11.2.2节）。缺省参数因其自身的特性可以不必给出来。

11.5.2 缺省参数

缺省参数是这样一种参数：如果函数被调用时没有给出这个参数的值，它就会使用声明该函数时给它定义的缺省值。缺省参数的定义是在函数声明部分的标题行里给出的。C++和Java是另外两种支持缺省参数的程序设计语言，对函数进行定义的语法也差不多，即参数后面紧跟着一个赋值操作把缺省值分配给那个参数。这个赋值语句只在语法方面起这样一个作用：如果函数被调用时没有给那个参数传递来一个值，则该参数就取定义时给定的缺省值。

在Python语言里，定义带缺省值的参数要遵守一个规则，即全部位置参数必须在缺省参数之前进行定义，如下所示：

```
def function_name(posargs, defarg1=dval1, defarg2=dval2,...):
    "function_documentation_string"
    function_body_suite
```

每个缺省参数后面都跟着一个给出其缺省值的赋值语句。如果在调用函数时没有给出相应

的值，这个赋值语句就发挥作用了。

1. 为什么要使用缺省参数

缺省参数能够大幅度地提高应用程序的健壮性，因为它们能提供标准的位置参数所无法提供的灵活性，大大减轻了应用程序设计人员的担子。少几个需要操心的参数让生活轻松了许多。当人们新接触到一个API接口，不太有把握提供目的明确的值做为参数时，这个功能就更加有用了。

我们可以把使用缺省参数的概念比做在计算机上安装软件的过程。在安装软件的时候，有多少次是选择“default install”（缺省安装）而不是选择“custom install”（用户定制安装）的呢？我想总是选择前者的人要占绝大多数。这样做既方便又不必操心安装过程的细节，更不用说节省大量的时间了。如果读者是一位总是选择用户定制安装发烧友，请记住你可是少数人里面的一个。

另一个好处是对程序设计人员而言的，缺省参数使他们能够更好地控制为顾客编写的软件。在挑选缺省值的时候，他们能够有选择地选定一个可能是“最佳”的缺省参数，并通过这种做法加大用户做出取舍的自由度；用户可以在逐渐熟悉了新的API或者系统之后再自行给出参数的值，抛开“学步车”。

下面是一个使用了缺省参数的小例子，它用起来很方便，对日益发展的电子商务业多少有些帮助：

```
>>> def taxMe(cost, rate=0.0825):
...     return cost + (cost * rate)
...
>>> taxMe(100)
108.25
>>>
>>> taxMe(100, 0.05)
105.0
```

在上面的例子里，taxMe()函数以一件商品的价格为参数，给出加税之后的含税销售总价。商品的价格是一个固定参数，而销售税则是一个缺省参数（我们的例子里是8.25%）。也许读者就是一位网上零售商，生意上的大部分顾客都来自同一个洲或同一个国家。不同税率地区的顾客当然都希望按照他们当地的销售税率计算含税价格。如果不想让缺省参数值起作用，你只需把自己的参数值放到函数调用里去就可以了，比如上面例子里的taxMe(100, 0.05)。给出5%的税率其实就相当于给rate参数一个参数值，它覆盖了或者说绕过了那个8.25%的缺省值。

固定参数必须放在任何一个缺省参数的前面。为什么？因为固定参数有多少个就得是多少个，不能多也不能少；而缺省参数就不是这样。再从语法上来看，如果允许混合次序地使用这些不同种类的参数，解释器就不可能正确识别出哪个值对应着哪个参数。如果参数不是按照正确次序给出的，就会引发一个SyntaxError例外，如下所示：

```
>>> def taxMe2(rate=0.0825, cost):
...     return cost * (1.0 + rate)
...
SyntaxError: non-default argument follows default argument
```

我们再来看看关键字参数，还使用我们熟悉的net_conn()函数做例子。

```
def net_conn(host, port):
    net_conn_suite
```

读者可能还记得在这里可以通过给参数命名的办法使自己的参数能够不按其位置次序提供。这样，对刚才的函数定义来说，下面两种调用办法都是有效的：一是使用正常的位置参数；二是使用关键字参数：

- net_conn('kappa', 8000)
- net_conn(port=8080, host='chino')

如果在此引入缺省参数，将又是一个天地，当然上面这两个调用还是有效的。我们把函数net_conn()的定义修改为让port参数有一个缺省值80；再增加一个名为stype（意思是服务器类型）的参数，让它有一个缺省值'tcp'。如下所示：

```
def net_conn(host, port=80, stype='tcp'):
    net_conn_suite
```

这样就增加了调用net_conn()函数的方法的个数。下面这些全部都是net_conn()函数的合法调用方法：

- net_conn('phaze', 8000, 'udp') # no def args used
- net_conn('kappa') # both def args used
- net_conn('chino', stype='icmp') # use port def arg
- net_conn(stype='udp', host='solo') # use port def arg
- net_conn('deli', 8080) # use stype def arg
- net_conn(port=81, host='chino') # use stype def arg

上面这些例子里有一个固定的东西，是什么？对了，就是那个固定参数host。host没有缺省值，所以它必须出现在每一个对net_conn()函数的调用当中。

关键字参数已经被证明是非常有用的，因为有了它们就可以不按次序地给出参数；有了缺省参数以后，它们还可以用来“跳过”缺失的参数，正如上面的例子里那样。

2. 函数对象做缺省参数的例子

下面再给出一个证明缺省参数很有用的例子。在程序示例11-2中给出的grabweb.py是一个比较简单的脚本程序，它的主要作用是从因特网上下载一个Web网页并把它临时保存到一个本地文件里等过后再进行分析。这类应用程序可以用来测试某个Web站点上网页的完整性，或者用来监控某个服务器上的负载（通过统计连接个数和下载速度）。我们可以在process()函数里做任何事情，其用途可以有无数种。在这个练习里，我们要做的事情是把检索到的网页里面的第一个和最后一个非空白行显示出来。虽然这个例子本身可能没有多少实用价值，但读者完全可以在这段代码的基础上自由发挥。

程序示例11-2 抓取网页 (grabweb.py)

这个脚本程序下载一个网页（默认为本地的www服务器上的某一个）并把该HTML文件中第一个和最后一个非空白行显示出来。因为download()函数中的两个参数都是缺省参数，既允许使用不同的URL地址，也可以定义不同的处理函数，所以它非常灵活。

```
1  #!/usr/bin/env python
```

```

2
3 from urllib import urlretrieve
4 from string import strip
5
6 def firstnonblank(lines):
7     for eachLine in lines:
8         if strip(eachLine) == '':
9             continue
10        else:
11            return eachLine
12
13 def firstlast(webpage):
14     f = open(webpage)
15     lines = f.readlines()
16     f.close()
17     print firstnonblank(lines),
18     lines.reverse()
19     print firstnonblank(lines),
20
21 def download(url='http://www', \
22             process=firstlast):
23     try:
24         retval = urlretrieve(url)[0]
25     except IOError:
26         retval = None
27     if retval:      # do some processing
28         process(retval)
29
30 if __name__ == '__main__':
31     download()

```

在我们的操作环境中运行这个脚本程序将得到如下所示的输出结果，它与读者计算机上得到的结果肯定不一样，因为读者查看的肯定会是另外一个网页。

```

% grabweb.py
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final// EN">

</HTML>

```

11.6 可变长参数

有时候会出现这样的情况：需要程序处理的参数的个数是变化着的。它们被称为“可变长参数表”。函数的定义部分里是不会给可变长参数明确命名的，这是因为在程序开始执行之前它们的个数是未知的（甚至在执行过程中，上次调用和这次调用的参数个数都不见得一样），它们与正式参数（位置参数和缺省参数）一个明显的区别就是后者会在函数的定义部分里明确地得到一个名字。因为Python语言中的函数调用提供了关键字和非关键字这两种类型，所以它对可变长参数的支持情况也有两种。

11.6.1 非关键字可变长参数

在调用一个函数的时候，全体正式参数（固定参数和缺省参数）都被赋值到定义该函数时给出的与之对应的本地局部变量去；剩余部分中的非关键字可变长参数会被依次插入一个表列

以便访问。读者应该对C语言中的“变参”（比如va_list、va_arg等变量和枚举类型的参数[...]）都比较熟悉吧。Python语言也提供了与之相当的支持——遍历表列中的元素进行操作与在C语言里使用va_arg变量其作用是完全一样的。如果读者不熟悉C语言或者变参这个概念，那我在这里告诉大家，它们代表的是接受传递到一个函数调用中去的非固定个数的参数的语法。

可变长参数表列必须被放在所有位置参数和缺省参数之后，带表列或非关键字可变长参数的函数的一般语法如下所示：

```
def function_name([formal_args,] *vargs_tuple):
    "function_documentation_string"
    function_body_suite
```

把所有的正式参数都定义分配好以后，如果还有多出来的参数，就都属于可变长参数这一大类，在容纳剩余参数的变量名前面要加上星号（*）操作符；如果没有多出来的参数，那么那个表列就会是空的。

我们在前面的内容里已经看到过，只要在调用函数时给出的参数个数不正确，就会生产一个TypeError例外。可现在，只要在函数定义时的标题行尾部添上一个代表可变长参数表的变量，就可以应付传递给函数的参数过多时会引起的的问题，这是因为所有多出来的（非关键字）参数都会被添加到容纳可变长参数的表列里去（多出来的关键字参数要求有一个关键字变量参数做为参数，请参考下一小节）。

再重申一遍，全体正式参数必须放在非正式参数的前面，这和要把位置参数放在缺省参数的前面的道理是一样的。请看下面的例子：

```
def tupleVarArgs(arg1, arg2='defaultB', *theRest):
    'display regular args and non-keyword variable args'
    print 'formal arg 1:', arg1
    print 'formal arg 2:', arg2
    for eachXtrArg in theRest:
        print 'another arg:', eachXtrArg
```

现在调用这个函数，看看表列形式的可变长参数是如何工作的：

```
>>> tupleVarArgs('abc')
formal arg 1: abc
formal arg 2: defaultB
>>>
>>> tupleVarArgs(23, 4.56)
formal arg 1: 23
formal arg 2: 4.56
>>>
>>> tupleVarArgs('abc', 123, 'xyz', 456.789)
formal arg 1: abc
formal arg 2: 123
another arg: xyz
another arg: 456.789
```

11.6.2 关键字可变长参数

如果可变长参数是一些关键字参数，就要把它们放到一个字典里去，其中“关键字”参数的变量名将做为那个字典的键字，而那个参数本身就是与之对应的键值。为什么一定要用一个

字典来完成这一任务呢？因为每个可变长参数都有两个互相对应着的数据项，即该参数的名字和它的值，非常适合用字典来保存这些参数。下面就是利用可变长参数字典容纳多出来的关键字参数的函数定义语法：

```
def function_name([formal_args,] [*vargst,] **vargsd):
    function_documentation_string
    function_body_suite
```

为了区分关键字可变长参数和非关键字非正式参数，要在前者的前面加上一个双星号（**）。这个双星号在Python语言里还可以用做乘方运算的操作符，这是一个被重载的操作符，但不应该把它与乘方操作符混为一谈。关键字可变长参数的字典必须是函数定义中的最后一个参数，它前面必须有一个双星号（**）。我们现在通过一个例子来看看应该如何使用一个这样的字典：

```
def dictVarArgs(arg1, arg2='defaultB', **theRest):
    'display 2 regular args and keyword variable args'
    print 'formal arg1:', dictVarArgs
    print 'formal arg2:', arg2
    for eachXtrArg in theRest.keys():
        print 'Xtra arg %s: %s' % \
            (eachXtrArg, str(theRest[eachXtrArg]))
```

在解释器里执行这个函数，我们将得到如下所示的输出结果：

```
>>> dictVarArgs(1220, 740.0, c='grail')
formal arg1: 1220
formal arg2: 740.0
Xtra arg c: grail
>>>
>>> dictVarArgs(arg2='tales', c=123, d='poe',
a='mystery')
formal arg1: mystery
formal arg2: tales
Xtra arg c: 123
Xtra arg d: poe
>>>
>>> dictVarArgs('one', d=10, e='zoo', men=('freud',
'gaudi'))
formal arg1: one
formal arg2: defaultB
Xtra arg men: ('freud', 'gaudi')
Xtra arg d: 10
Xtra arg e: zoo
```

关键字可变长参数和非关键字可变长参数可以同时用在同一个函数里，但前提是关键字的字典必须是最后一个参数，而非关键字的表列必须放在它的前面。如下所示：

```
def newfoo(arg1, arg2, *nkw, **kw):
    'display regular args and all variable args'
    print 'arg1 is:', arg1
    print 'arg2 is:', arg2
    for eachNKW in nkw:
        print 'additional non-keyword arg:', eachNKW
    for eachKW in kw.keys():
        print "additional keyword arg '%s': %s" % \
```

```
(eachKW, kw[eachKW]))
```

在解释器里调用这个函数，我们将得到如下所示的输出结果：

```
>>> newfoo('wolf', 3, 'projects', freud=90, gamble=96)
arg1 is: wolf
arg2 is: 3
additional non-keyword arg: projects
additional keyword arg 'freud': 90
additional keyword arg 'gamble': 96
```

11.6.3 调用带有可变长参数对象的函数

[1.6]

从Python 1.6版本开始允许以成组的方式提供可变长参数，非关键字的表列和关键字的字典都可以这么做。在前面每一个使用了可变长参数的例子里，调用函数时那些可变长参数都是单个地给出来的（请参考前一小节里的每一个例子）。在1.6版本之前，这是调用带有可变个数的参数的函数的唯一方法。

函数调用允许与一个表列为参数，把它直接用做一个容纳非关键字可变长参数的表列；也允许以一个字典为参数，把其中的键字-键值对添加到容纳关键字可变长参数的字典里去。这个表列和字典可以在函数调用里面与按照正常方式给出的参数们分别列出，一起使用。从1.6版本开始，Python语言所支持的带可变长参数的函数调用的完整语法是：

```
function_name(formal_args, *nonKWtuple, **KWdict)
```

在前一小节里，我们已经看到单星号(*)和双星号(**)构造能够用在函数的定义部分里，但现在它们在函数调用里也可以用了！

我们现在用前一小节定义的新foo()函数测试一下新的调用语法。对新foo()函数的第一个调用使用的还是老办法，即一个一个地列出使用的参数，包括那些跟在正式参数后面的可变长参数，如下所示：

```
>>> newfoo(10, 20, 30, 40, foo=50, bar=60)
arg1 is: 10
arg2 is: 20
additional non-keyword arg: 30
additional non-keyword arg: 40
additional keyword arg 'foo': 50
additional keyword arg 'bar': 60
```

现在再做一次调用。但这次不是把参数一个一个地列出来，而是先把非关键字参数放在一个表列里，把关键字参数放在一个字典里去后再调用它：

```
>>> newfoo(2, 4, *(6, 8), **{'foo': 10, 'bar': 12})
arg1 is: 2
arg2 is: 4
additional non-keyword arg: 6
additional non-keyword arg: 8
additional keyword arg 'foo': 10
additional keyword arg 'bar': 12
```

最后，我们再调用一次这个函数，但这次我们是在函数外建立起表列和字典的，如下所示：

```
>>> aTuple = (6, 7, 8)
>>> aDict = {'z': 9}
```

```
>>> newfoo(1, 2, 3, x=4, y=5, *aTuple, **aDict)
arg1 is: 1
arg2 is: 2
additional non-keyword arg: 3
additional non-keyword arg: 6
additional non-keyword arg: 7
additional non-keyword arg: 8
additional keyword arg 'z': 9
additional keyword arg 'x': 4
additional keyword arg 'y': 5
```

请注意，我们的表列和字典只是函数调用最终接受的表列和字典的一个组成部分：一个非关键字的值“3”和两个关键字值“x”和“y”虽然不是“*”和“**”操作符定义的可变长参数表里的元素，可最终还是被包括在整个的参数表里了。在1.6版本之前，可变对象只能通过执行/调用的功能函数`apply()`来传递。

11.7 函数化的程序设计

Python不是一个函数化的程序设计语言，并且今后也不太可能这样声称自己；可是它确实支持一些很有价值的函数化的程序设计构造，也有一些类似于函数化程序机制的东西——但从传统角度考虑还可能不这样认为。Python语言提供的东西有四个内建函数和`lambda`表达式。

11.7.1 匿名函数和`lambda`

Python允许人们通过`lambda`关键字创建匿名的函数。说它们是“匿名的”是因为它们不是通过标准的方式（即通过`def`语句）声明的。（除非被赋值给一个本地的局部变量，否则这类对象也根本不会在任何名字空间里创建一个名字。）但它们像函数一样可以有自己的参数。一个完整的`lambda`“语句”表现为一个表达式，而这个`lambda`表达式又必须与它的定义标题行放在同一行上。下面就是用`lambda`关键字定义匿名函数的语法：

```
lambda [ arg1[, arg2, . . . , argN] ] : expression
```

其中的参数是可选的，如果有参数，通常也是哪个表达式的一个组成部分。

编程提示：`lambda`表达式返回的是可调用函数对象

[CN]

调用带有表达式的`lambda`命令会产生一个函数对象（function object），并且可以像其他函数一样用。它们可以被传递给其他函数做参数、用额外的引用建立别名、被当作容器对象的成员等，还可以做为一个可调用的对象被调用（如有必要，还可以带参数）。在被调用时，这些对象产生的结果与同一个表达式给定同样参数时求出来的结果是完全一样的。如果定义部分中的表达式彼此相当，就很难把它们与返回一个表达式结果的函数区分开。

在介绍使用`lambda`关键字的例子之前，先复习一下单行语句的内容和它们与`lambda`表达式的相似之处。

```
def true() :
    return 1
```

这个函数没有参数，并且返回值永远都是1。Python语言中的单行函数也可以和自己的标题行写在同一行上。这样，我们就可以把刚才的true()函数重新编写成如下所示的样子：

```
def true() : return 1
```

我们将在本章内容里用这种方式给出这个函数，因为这样可以帮助大家更好地认识与之对应的lambda参照物。就这个true()函数来说，与之对应的lambda表达式（无参数，永远返回1值）是：

```
lambda :1
```

被命名为true()的函数的用途是很明显的，可lambda表达式就不同了。是就这样用呢，还是需要我们在哪儿执行个赋值操作呢？一个lambda表达式本身无法用于任何目的，如下所示：

```
>>> lambda :1
<function <lambda> at f09ba0>
```

在上面的例子里，我们只是简单地创建了一个lambda函数，却没有把它保存起来或者调用它。这个函数对象引用的计数在创建该函数对象时被设置为1，但因为没有任何引用，所以又回到0，而该对象也被当作废弃物回收了。要想留住这个对象，可以把它保存到一个变量去，以后就可以随时调用它了。也许现在就是个好机会，如下所示：

```
>>> true = lambda :1
>>> true()
1
```

把它用在这里的赋值语句里就有用得多了。同样地，我们还可以把lambda表达式赋值给一个数据结构，如一个列表或者表列等；这样，根据对输入数据加以处理的措施，我们可以挑选即将执行的函数和对应的参数。（下一小节将向大家展示如何把lambda表达式用在函数化的程序设计结构中。）

现在来设计一个函数，它要使用两个数值或两个字符串做参数，返回数值的和以及字符串的合并结果。我们先让大家看看标准函数会怎样做，然后再看看这个无名之辈怎样做。

```
def add(x, y) : return x + y  ⇔  lambda x, y : x + y
```

缺省参数和可变长参数也允许使用，请看下面的例子：

```
def usuallyAdd2(x, y=2) : return x + y  ⇔  lambda x, y=2 : x + y
def showAllTuple(*z) : return z        ⇔  lambda *z : z
```

眼见为实，所以下面就来看看如何在解释器里使用它们：

```
>>> a = lambda x, y=2: x + y
>>> a(3)
5
>>> a(3,5)
8
>>> a(0)
2
>>> a(0,9)
9
>>>
>>> b = lambda *z: z
>>> b(23, 'zyx')
(23, 'zyx')
```

```
>>> b(42)
(42,)
```

关于lambda还有最后一句话。虽然lambda看起来像是个只有一行的函数，却并不等同于C++中的“行内”语句，后者的作用是在调用期间绕过函数堆栈的分配操作以提高程序的执行性能。lambda表达式的行为就像是一个函数，会在被调用的时候创建一个框架对象（frame object）。

11.7.2 内建函数：apply()、filter()、map()、reduce()

本小节主要介绍apply()、filter()、map()和reduce()这四个内建函数并给出一些这几个函数的使用例子。这几个函数为Python语言增添了函数化程序设计的色彩。对这几个函数的总结请参考表11-1，它们都要求有一个函数对象以供调用。

表11-1 函数化程序设计内建函数

内建函数	说 明
<code>apply(func[, nkw][, kw])</code>	用可选的参数调用函数func。nkw表示非关键字参数，kw表示关键字参数。它的返回值就是func函数的返回值
<code>filter(func, seq)</code>	调用布尔函数func遍历处理序列seq中的每一个元素。它的返回值是一个序列，其元素都是让func函数返回真值的原seq序列中的元素
<code>map(func, seq1[, seq2...])</code>	用函数func对给定序列中的每一个元素进行处理，并把结果放在一个列表里返回。如果func是None，则func的行为就好比一个恒等函数。如果此内建函数的参数表里有n个序列，那做为返回值的列表就由n个表列构成，每个表列是由各个序列对应元素组成的集合
<code>reduce(func, seq[, init])</code>	用二元函数func对序列seq中的元素进行处理，每次处理两个数据项（一个是前次处理的结果，一个是序列中的下一个元素），如此反复对当前结果和下一个序列元素进行运算以获得一个递进的结果，最后对整个序列求出一个单个的返回值。如果给出了初始值init，第一个运算就对init和序列中的第一个元素而不是对序列中的前两个元素来进行

可以想象，lambda表达式非常适合于用在使用了上面这些函数的应用程序里，因为这些函数都要求有一个函数对象供调用目的使用，而lambda正好提供了一个随时随地创建函数的机制。

1. *apply()内建函数

我们准备研究的第一个内建函数就是apply()。apply()函数是这四个函数中最基本的一个，它只被简单地用来传递一个函数对象和任何参数（如果有的话），然后由apply()调用那个函数对给出的参数进行处理。apply()没有丝毫的特殊之处，按部就班地完成自己的工作。下面两个调用在效果上是完全一致的：

```
foo(3, 'pyramid')    ⇔    apply( foo, (3, 'pyramid') )
```

那些参数也可以事先保存到一个表列里，然后再用apply()调用那个函数，如下所示：

```
args = (4, 'eve', 79)
apply(foo, args)
```

注意这和用一个单个的参数（即一个表列）调用foo()函数的foo(args)可不一样；apply()是用三个参数（即表列中的元素）调用了foo()函数的。

如果读者想从解释器调用内建函数`dir()`，可以直接执行它，也可以利用`apply()`函数调用它。因为不需要用到什么参数，所以这个例子中的两种调用方法效果完全一致。

```
dir()      ⇔      apply(dir)
```

下面是在解释器里执行这两种调用方法的结果，它们是完全相同的：

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>>
>>> apply(dir)
['__builtins__', '__doc__', '__name__']
```

读者可能会问：在可以进行函数调用的时候，还需要用到`apply()`吗？有必要做以下的事情吗——既要多打好几个字，语法还复杂了不少。

`apply()`函数在某些特定的情况下会是非常有用的工具。如果需要调用一个函数，可这个函数的参数是动态生成的，在这种情况下，`apply()`就非常方便了。这类情况通常都会涉及到拼凑一个参数清单的问题。在程序示例11-3里的算术游戏中（`matheasy.py`），我们要拼凑出一个有两个数据项的参数清单，然后再把它送到相应的算术函数中去。

`matheasy.py`程序是为孩子们编写的一个算术测验小游戏，让程序自己从加法、减法和乘法里挑出一个算术运算来。我们使用了这些算术操作符的等价函数，即`add()`、`sub()`和`mul()`，这几个函数都可以在`operator`模块里找到。接下来要生成一个参数清单来（两个参数的，因为算术运算都属于二元操作符/操作），这些随机数将做为运算数。因为我们不打算在程序的这个相当初级的版本里支持负数，所以还要把参数清单里的两个数字按从大到小的顺序排好序。最后，我们把这个参数清单和随机挑选出来的算术运算符做为参数送到`apply()`函数里并调用它以获得正确的测验答案。在这个应用程序里之所以选用`apply()`函数有两个原因：

- 需要动态生成参数清单。
- 具体使用的算术函数需要随机挑选。

因为我们无法知道用户看见的算术题里具体参加计算的参数是什么，也无法知道调用的是哪一个算术运算函数，所以用`apply()`使我们的解决方案灵活多能。

程序示例11-3 使用`apply()`函数的算术游戏（`matheasy.py`）

随机选择两个数字和一个算术函数，向用户显示该算术题，对用户输入的计算结果进行验证。三次答错后由程序给出正确答案并不再继续提问；此时用户只有输入了正确的答案后才能继续游戏。

```
1  #!/usr/bin/env python
2  from string import lower
3  from operator import add, sub, mul
4  from random import randint, choice
5
6  ops = { '+': add, '-': sub, '*': mul }
7  MAXTRIES = 2
8
9  def doprob():
10     op = choice('+-*')
11     nums = [randint(1,10), randint(1,10)]
12     nums.sort() ; nums.reverse()
13     ans = apply(ops[op], nums)
14     pr = '%d %s %d = ' % (nums[0], op, nums[1])
15     oops = 0
```

```

16     while 1:
17         try:
18             if int(raw_input(pr)) == ans:
19                 print 'correct'
20                 break
21             if oops == MAXTRIES:
22                 print 'answer\n%s%d'%(pr,ans)
23             else:
24                 print 'incorrect... try again'
25                 oops = oops + 1
26         except (KeyboardInterrupt, \
27                 EOFError, ValueError):
28             print 'invalid input... try again'
29
30 def main():
31     while 1:
32         doprob()
33         try:
34             opt = lower(raw_input('Again? '))
35         except (KeyboardInterrupt, EOFError):
36             print ; break
37         if opt and opt[0] == 'n':
38             break
39
40 if __name__ == '__main__':
41     main()

```

1~4行

代码的开始照例是UNIX的启动行，各种非UNIX的系统都能够毫无痛苦地忽略这一行语句。它后面是from-import语句，加载的string.lower()用来对用户的输入进行大小写不敏感处理后进行验证；random.randint()用来选择算术运算的运算数；random.choice()用来随机挑选出算术运算符；而我们需要的全体算术运算符都来自operator模块。

6~7行

这个应用程序里用到的全局变量是一组操作运算和与之对应的函数，以及一个用来指示回答次数的值（初始值是2，即用0、1、2代表有三次答题机会）。如果三次的答案都不正确，程序就会给出正确的答案。函数的字典（即ops）用操作符的符号做为字典中的键字，用它找出正确的算术函数来。

9~28行

doprob()函数是这个应用程序的核心引擎。它随机挑选出一个运算操作并生成两个运算数，把它们按从大到小的顺序排好以避免在做减法的时候出现负数。接下来，它用调用apply()函数的办法调用算术函数和运算数计算出正确的答案。然后把算术题显示给用户并给三次机会输入正确的答案。

30~41行

本应用程序的主程序main()部分，如果这个脚本程序是被直接调用的就从最顶层开始执行。如果是被导入的情况，导入它的那个函数可以通过调用doprob()对其操作执行进行管理，也可以通过调用main()来控制这个程序。main()很简单，先调用doprob()让用户进入这个脚本程序的主功能，再提问用户是退出还是继续回答下一个问题。

由于运算数和操作符都是随机挑选出来的，所以matheasy.py每次执行的情况都不会相同。这是某次执行的情况：


```
% matheasy.py
7 - 2 = 5
correct
Try another? ([y]/n)
7 * 6 = 42
correct
Try another? ([y]/n)
7 * 3 = 20
incorrect... try again
7 * 3 = 22
incorrect... try again
7 * 3 = 23
sorry... the answer is
7 * 3 = 21
7 * 3 = 21
correct
Try another? ([y]/n)
7 - 5 = 2
correct
Try another? ([y]/n) n
```

另外一个使用了`apply()`函数的有用的应用程序来自调试纠错和性能测试方面。假设你正在编写的函数需要通宵运行以便进行彻底的测试或者找出不足之处，或者需要对它的多次运行情况进行计时以便做进一步的改进。总之，需要编写一个诊断性函数来建立测试环境，然后调用准备对它进行测试的函数。因为这个系统的灵活性和适应性应该很好，你打算把被测试函数做为一个参数传递过去。所以两个满足这些要求的函数`timeit()`和`testit()`对今日的软件开发人员来说可能很有用。

我们在这里给出`testit()`函数的示范性源代码（请参考程序示例11-4），把`timeit()`函数留给读者做为一个练习（请参考本章后面的练习11-12）。

程序示例11-4 测试函数（`testit.py`）

`testit()`调用一个带参数的给定函数，如果成功就把一个返回值1和被测试函数的返回值打包在一起返回。如果失败就返回一个0值和产生例外的原因。

```
1  #!/usr/bin/env python
2
3  def testit(func, *nkwargs, **kwargs):
4
5      try:
6          retval = apply(func, nkargs, kwargs)
7          result = (1, retval)
8      except Exception, diag:
9          result = (0, str(diag))
10     return result
11
12 def test():
13     funcs = (int, long, float)
14     vals = (1234, 12.34, '1234', '12.34')
15
16     for eachFunc in funcs:
17         print '-' * 20
18         for eachVal in vals:
```

```

19         retval = testit(eachFunc, \
20                           eachVal)
21         if retval[0]:
22             print '%s(%s) = ' % \
23                   (eachFunc.__name__, `eachVal`), retval[1]
24         else:
25             print '%s(%s) = FAILED: ' % \
26                   (eachFunc.__name__, `eachVal`), retval[1]
27
28 if __name__ == '__main__':
29     test()

```

这个模块为函数准备了一个测试性的执行环境。testit()函数以一个被测试函数做为参数，在一个例外处理器的监视下用给定的参数调用该函数。如果被测试函数执行成功了，就把一个返回值1和被测试函数的返回值打包在一起返回给调用者；如果执行失败了，就返回一个0值和对例外的原因。(Exception是所有例外的根类 (root class)，详细情况请参考第10章内容。)

在程序示例11-4中，主测试函数test()用四个数的输入集合对一组数值转换函数进行了测试。在示例中的测试中出现了两个失败的情况，证明testit.py这个程序是具备我们所要求的功能性的。下面是运行这个脚本程序的输出：

```

% testit.py
-----
int(1234) = 1234
int(12.34) = 12
int('1234') = 1234
int('12.34') = FAILED: invalid literal for int(): 12.34
-----
long(1234) = 1234L
long(12.34) = 12L
long('1234') = 1234L
long('12.34') = FAILED: invalid literal for long(): 12.34
-----
float(1234) = 1234.0
float(12.34) = 12.34
float('1234') = 1234.0
float('12.34') = 12.34

```

2. filter()内建函数

我们在本章研究的第二个内建函数是filter()。假设你去到一个果园，离开时带着一袋从树上采摘下来的苹果。如果整个袋子通过一个过滤器后能够让留下来的都是好苹果，这样不是很好吗？这就是filter()函数的主要功能。

给定一个由许多对象组成的序列和一个“过滤”函数，让序列中的每一个数据项通过那个过滤器，只把那些让过滤函数返回真值的对象留下来。filter()函数调用给定的布尔函数对给定序列中的每一个数据项进行判定。使filter()返回一个非零（真）值的数据项被迫加到一个列表里去。最后返回的就是对原始序列“过滤后”得到的那个序列。

如果用纯Python来编写这个filter()函数，那它看起来应该是如下所示的样子：

```

def filter(bool_func, sequence):
    filtered_seq = []
    for eachItem in sequence:

```

```

if apply(bool_func,(eachItem,)):
    filtered_seq.append(eachItem)
return filtered_seq

```

图11-1中的示意图可以帮助大家更好地理解filter()函数的行为。

在图11-1中，放在最上面的是原始的序列，我们用seq[0]、seq[1]、... seq[N-1]来表示长度为N的序列中的各个元素。每调用一次bool_func()（即bool_func(seq[0])、bool_func(seq[1])等等），就会返回一个1或0的返回值（至少根据布尔函数的定义是如此的——要保证你的函数确实会返回0或1）。如果序列中有数据项使bool_func()返回一个真值，就把该元素插到返回序列里面去。完成对整个序列的遍历之后，filter()函数将返回那个新创建的序列。

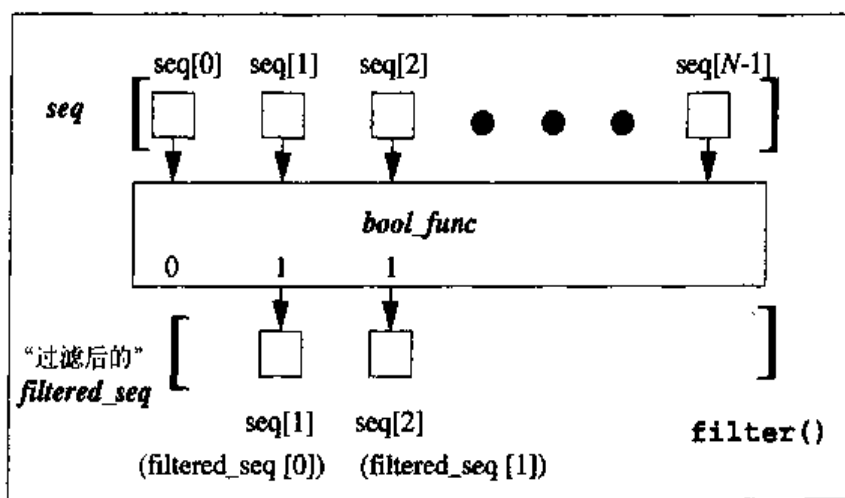


图11-1 filter()内建函数的工作原理

下面的脚本程序利用filter()得到一个随机奇数列表。这个脚本程序先要生成一个大的随机数集合，然后过滤掉全部的偶数，留下来的就是我们想要的数据集。刚开始为这个例子编写程序时，oddnogen.py差不多是下面这个样子：

```

from random import randint

def odd(n):
    return n % 2

def main():
    allNums = []
    for eachNum in range(10):
        allNums.append(randint(1, 101))
    oddNums = filter(odd, allNums)
    print len(oddNums), oddNums

if __name__ == '__main__':
    main()

```

这个脚本程序由两个函数组成：`odd()`是个用来判断一个整数是奇数（true）还是偶数（false）的布尔函数；`main()`是主程序部分。`main()`的作用是生成10个1到100之间的随机数，然后调用`filter()`函数过滤掉所有的偶数。最后，把那10个数字中的奇数显示出来，并先给出经过过滤的列

表的长度。

导入并执行几次这个模块，我们将得到如下所示的结果：

```
>>> import oddnogen
>>> oddnogen.main()
4 [9, 33, 55, 65]
>>>
>>> oddnogen.main()
5 [39, 77, 39, 71, 1]
>>>
>>> oddnogen.main()
6 [23, 39, 9, 1, 63, 91]
>>>
>>> oddnogen.main()
5 [41, 85, 93, 53, 3]
```

仔细研究一下这个模块可以发现，传递到filter()函数中去的odd()函数可以被替换为一个lambda表达式，而这将是我们修改后最终的oddnogen.py脚本程序。具体代码请参考程序示例11-5。

程序示例11-5 奇数发生器 (oddnogen.py)

这个简单的程序先在1到100之间生成10个随机数字，然后过滤掉所有的偶数。最后把过滤后剩余数字的个数和奇数列表结果显示出来。

```
1 #!/usr/bin/env python
2
3 from random import randint
4
5 def main():
6
7     allNums = []
8     for eachNum in range(10):
9         allNums.append(randint(1, 100))
10    oddNums = filter(lambda n: n % 2, allNums)
11    print len(oddNums), oddNums
12
13 if __name__ == '__main__':
14     main()
```

3. map()内建函数

map()内建函数有一点与filter()很相似，那就是它们都是用一个函数对一个序列进行处理。但与filter()不同的是：map()函数是对序列中各数据项的函数调用进行“映射”，它返回的是一个由全体返回值组成的列表。

形式最简单的map()函数以一个函数和一个序列做为参数，对序列中的每一个数据项调用那个函数，最后创建一个返回值列表，该列表由原序列中各数据项被那个函数调用后的返回值组成。所以，如果映射函数是给它的参数加上2，把这个函数和一列表数字送入map()后，结果列表将与原列表中的数字属同一集合，只不过每个数字都加上了2。用Python语言编写的这段简单形式的map()函数将是如下所示的样子，整个操作过程请参考图11-2中的示意图。

```
def map(func, seq):
    mapped_seq = []
    for eachItem in seq:
        mapped_seq.append(apply(func, eachItem))
    return mapped_seq
```

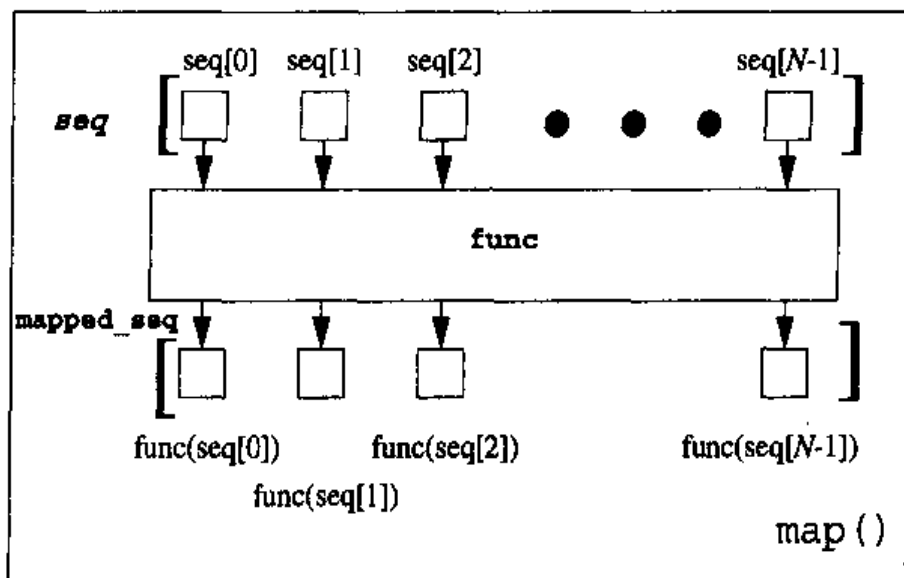


图11-2 map()内建函数的工作原理

下面这几个lambda表达式可以进一步说明map()内建函数的工作原理：

```
>>> map((lambda x: x+2), [0, 1, 2, 3, 4, 5])
[2, 3, 4, 5, 6, 7]
>>>
>>> map(lambda x: x**2, [0, 1, 2, 3, 4, 5])
[0, 1, 4, 9, 16, 25]
>>>
>>> map((lambda x: x**2), range(6))
[0, 1, 4, 9, 16, 25]
```

形式更一般化的map()函数可以把不止一个序列当作自己的输入。在这样的情况下，map()将同时遍历每一个序列。在map()第一次调用做为其参数的那个函数（即func函数）时，它会先把各序列中的第一个元素都归入一个表列，用func对这个表列进行处理，而map()返回的也是一个表列；它会把这个表列插入到mapped_seq代表的那个映射序列里去；当map()结束运行时，这个mapped_seq映射序列就是最终的返回值。

图11-2给出的是map()对单个序列进行操作的执行情况。如果是用map()对M个各自包含N个对象的序列进行操作，其执行情况就会是如图11-3所示的样子。

举例来说，我们来看看下面这个map()调用：

```
>>> map(lambda x, y: x + y, [1, 3, 5], [2, 4, 6])
[3, 7, 11]
```

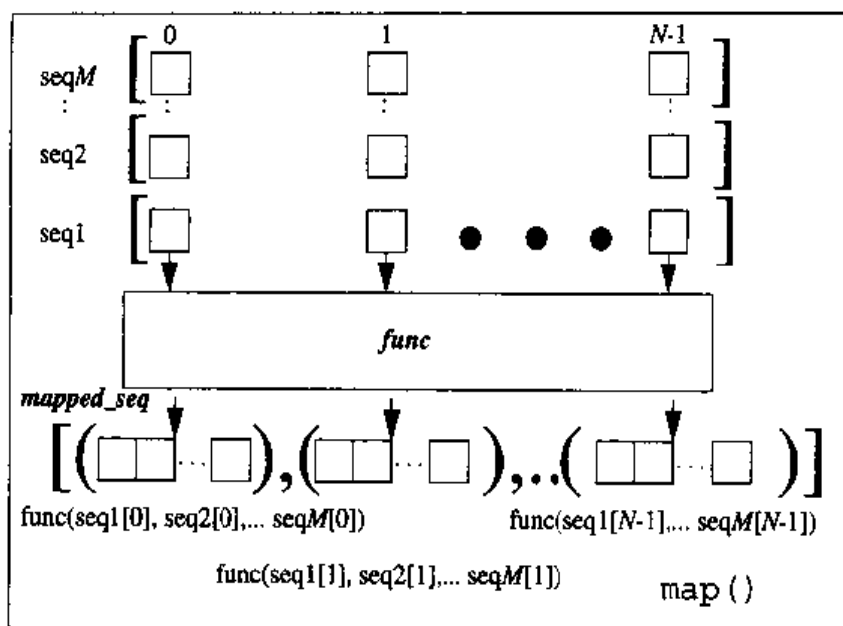


图11-3 map()内建函数对多个序列进行处理的执行情况示意图

在上面的例子里，序列的个数M是2，即列表[1, 3, 5]和[2, 4, 6]。这两个序列的元素个数或者说长度N都是3。最后的结果将是：

已知：

```
f(x, y) = x + y,
seq0 = [1, 3, 5], and
seq1 = [2, 4, 6]
```

则：

```
map(f, seq0, seq1)
= [ f(seq0[0], seq1[0]), f(seq0[1], seq1[1]), \
    f(seq0[2], seq1[2])
= [ 1 + 2, 3 + 4, 5 + 6 ]
= [ 3, 7, 11]
```

此外，map()还可以用None做为自己的函数参数。如果用None代替一个实际的函数对象做为参数，则map()会把这种情况默认为恒等函数，即结果映射图将与传递进去的序列是一样的。如果传递进去了多个序列，结果列表将由一组表列构成，每个表列由各序列相同位置上的元素组成。下面是用map()对多个序列进行处理的几个例子，其中一个是None做为map()中的函数参数的：

```
>>> map(lambda x, y: (x+y, x-y), [1,3,5], [2,4,6])
[(3, -1), (7, -1), (11, -1)]
>>> map(lambda x, y: (x+y, x*y), [1,3,5], [2,4,6])
[(3, 2), (7, 12), (11, 30)]
>>> map(None, [1,3,5], [2,4,6])
[(1, 2), (3, 4), (5, 6)]
```

[2.0]

因为经常会用到最后一个例子里的操作，在Python 2.0版本里特意新增了一个完成同样功能

(给出的序列其长度要相同)的内建函数zip() (见程序示例11-6)。

程序示例11-6 文本文件的处理 (strupper.py)

strupper.py以一个现有文本文件为参数,清除文本行前后所有的多余空格,再把所有文字都转换为大写字符。

```

1  #!/usr/bin/env python
2
3  from string import strip, upper
4
5  f = open('map.txt')
6  lines = f.readlines()
7  f.close()
8
9  print 'BEFORE:\n'
10 for eachLine in lines:
11     print ' [%s]' % eachLine[:-1]
12
13 print '\nAFTER:\n'
14 for eachLine in map(upper, \
15                     map(strip, lines)):
16     print ' [%s]' % eachLine

```

这些例子都是很明显的,但我们要在这里再给出一段更有实际意义的代码。在下一个例子里,我们将创建一个名为map.txt的文本文件,该文件里有几个前后都有多余空格字符的文本行。我们将利用程序示例11-6中给出的strupper.py脚本程序把该文件的每一行传递到string.strip()里以去掉它们前后多余的空格,并用string.upper()把所有文本都转换为大写字符。这个脚本程序会把开始处理之前和处理完成之后的文件内容显示给大家,如下所示:

```

% strupper.py
BEFORE:

[ Apply function to every item of list and return a ]
[list of the results. If additional list arguments are ]
[passed, function must take that many arguments and is ]
[applied to the items of all lists in parallel. ]

AFTER:

[APPLY FUNCTION TO EVERY ITEM OF LIST AND RETURN A]
[LIST OF THE RESULTS. IF ADDITIONAL LIST ARGUMENTS ARE]
[PASSED, FUNCTION MUST TAKE THAT MANY ARGUMENTS AND IS]
[APPLIED TO THE ITEMS OF ALL LISTS IN PARALLEL.]

```

注意:只有文本行前后多余的空格被清理干净了。字符串之间(比如最后一行中的情况)的多余空格不受影响。

本小节里的最后一个例子是对数字进行处理。具体来说,假设我们有一个文本文件,里面的内容全都是美元金额。可以假设这些金额来自你的收入税单,但你想把它们都四舍五入为最接近的美元整数金额。下面是测试用的文本文件round.txt中的内容:

```

98.76
90.69
51.36
50.89
28.34

```

49.64
6.87
36.95
59.25
55.96

请看程序示例11-7中脚本程序rounder.py的代码，它的作用是去掉尾缀的换行符并把所有的数字四舍五入为最接近的美元金额（当然要把数据从字符串转换为浮点数）。

程序示例11-7 处理文本文件中数字（rounder.py）

rounder.py从一个文本文件中读入一组浮点数字，把它们四舍五入为最接近不带小数部分的数字。这个练习模仿的是从收入税单中读出数字并把它们四舍五入为最接近的美元金额。

```
1  #!/usr/bin/env python
2
3  f = open('round.txt')
4  values = map(float, f.readlines())
5  f.close()
6
7  print 'original\trounded'
8  for eachVal in map(None, values, \
9      map(round, values)):
10     print '%6.02f\t\t%6.02f' % eachVal
```

这个脚本程序做的第一件事是调用map()函数，把每一行送到float()内建函数中去，在那里把字符串形式的数字转换为去掉了前后多余空格字符的数值。

最后，主代码部分把原来的数字和四舍五入后的数字都显示给大家，其具体处理方法是把这些数字都通过map()送入round()内建函数。同时，我们还以None为函数参数调用了map()——隐含为恒等操作，唯一的目的是把它的序列参数合并为一个由表列构成的单个列表里去，每个表列又是来自各序列的一个值组成的（我们从2.0版本开始可以用zip()内建函数直接完成这一工作，这在本小节前面已经提到过了）。这个例子中的表列是由一个原来的数字和一个四舍五入后的数字组成的。for循环对这个由表列构成的列表进行遍历，每个表列代表着一个原来的数字和一个四舍五入后的数字，最后再以整齐易读的格式显示给用户。

执行rounder.py脚本程序，我们得到如下所示的输出结果：

```
% rounder.py
original      rounded
98.76         99.00
90.69         91.00
51.36         51.00
50.89         51.00
28.34         28.00
49.64         50.00
6.87          7.00
36.95         37.00
59.25         59.00
55.96         56.00
```

4. reduce()内建函数

最后一块函数化程序设计的砖石是reduce()内建函数，它的输入参数包括一个二元函数（一个以两个值做为自己的输入参数，经过计算处理后返回输出一个值的函数）、一个序列和一个可选的初始计算值，按给定的办法把该序列“缩短”为一个单个的值，这也就是它名字的来历。

它的具体做法是这样的：先从序列中取出头两个元素并把它们传递到那个二元函数里去求出一个值；再把这个值和序列中的下一个元素做为那个二元函数的参数继续进行计算处理，如此循环直到计算到序列中的最后一个值为止。

读者可以把reduce()看做下面这个功能相当的例子：

```
reduce( func, [1, 2, 3] )    ==    func(func(1, 2), 3)
```

有些人认为reduce()函数“正确的函数性”用法应该是每次只要求取出一个数据项参加计算。在上面例子里，进行第一次计算时用到了两个数据，因为此时我们还没有来自上次计算的一个“结果”（还没有进行过计算，哪来的上次计算的结果呢）。而这就是为什么有一个可选的初始计算值的原因了。如果给出了一个初始计算值，就会用该初始计算值和序列中的第一个元素进行第一次计算，此后的过程就和刚才介绍的一样了。

如果要用纯Python语言实现reduce()函数，它看起来应该是如下所示的样子：

```
def reduce(bin_func, seq, init=None):

    lseq = list(seq)           # convert to list

    if init == None:           # initializer?
        res = lseq.pop(0)      # no
    else:
        res = init             # yes

    for item in lseq:           # reduce sequence
        res = bin_func(res, item) # apply function

    return res                  # return result
```

在本章要介绍的四个函数里，这个函数可能是从概念上最难理解的了，所以我们在下面用一个例子和一个示意图（请参考图11-4）来做进一步的说明。这第一个使用了reduce()函数的例子是一个简单的加法函数，与它相当的lambda表达式是大家在本章前面看到过的那个：

- `def sum(x, y) : return x + y`
- `lambda x, y : x + y`

给定一个列表后，要想得到其中所有值的总和是很简单的：建立一个循环，对该列表进行遍历，不断累加当前元素，最后在循环结束时给出累加的结果。如下所示：

```
allNums = range(5)           # [0, 1, 2, 3, 4]
total = 0
for eachNum in allNums:
    total = sum(total, eachNum) # total = total + eachNum
print 'the total is:', total
```

在解释器里敲进这段代码，如下所示：

```
>>> def sum(x,y): return x+y
>>> allNums = range(5)
```

```
>>> total = 0
>>> for eachNum in allNums:
...     total = sum(total, eachNum)
...
>>> print 'the total is:', total
the total is: 10
```

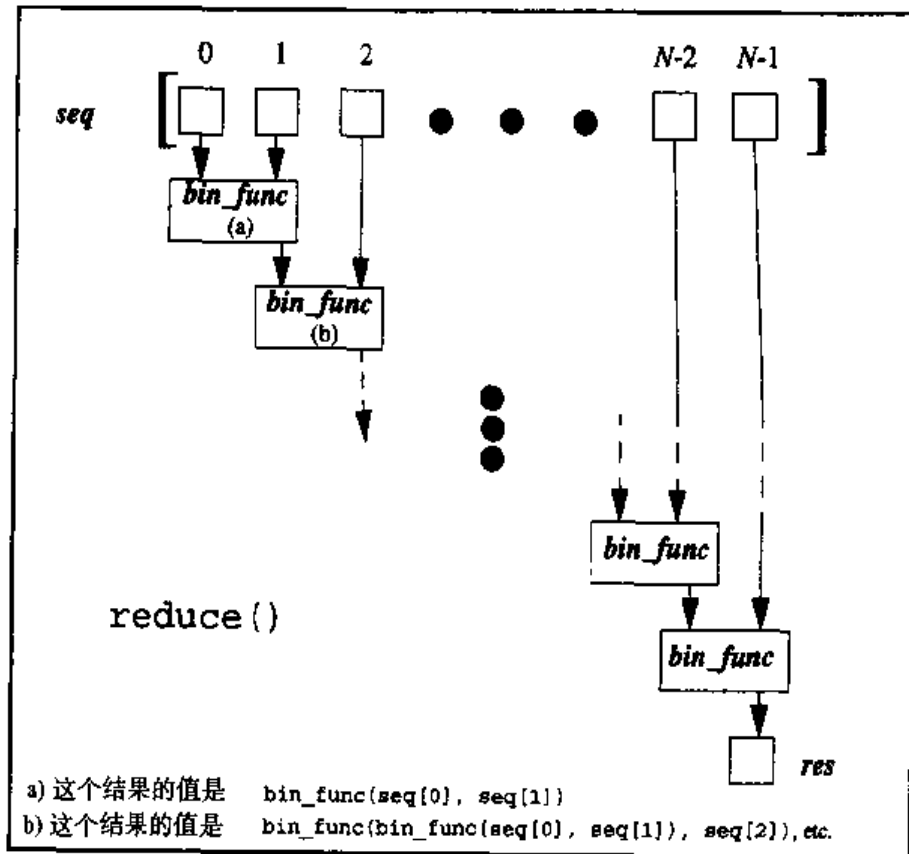


图11-4 reduce()内建函数的工作原理

利用lambda表达式，此项工作可以用一行调用reduce()的程序代码来完成：

```
>>> print 'the total is : ', reduce((lambda x, y : x + y), range(5) )
the total is : 10
```

reduce()函数对给定的输入数据执行了以下数学运算：

$((((0 + 1) + 2) + 3) + 4) \Rightarrow 10$

它先取出列表中的头两个元素（0和1），调用sum()得到它们的和1；再对这个值和第二个数据项2调用sum()，得到结果3；再继续把它与第三个数据项3用sum()累加；最后把4加到前面得到的部分和数上得到最终的返回值10。

11.8 变量的作用范围

标识符的作用范围是能够让该标识符的定义起有效作用的那部分程序，我们也经常把它叫做“变量的可见性”。换句话说，这就等于在问你能够在程序的哪个部分对某个特定的标识符进行访问。变量的作用范围或者是全局性的或者是局部性的。

11.8.1 全局变量和局部变量的比较

在一个函数内定义的变量其作用范围是局部性的，而一个模块中处于最高层的那些变量其作用范围是全局性或非局部性的。

在Aho、Sethi和Ullman等人关于编译理论的著名的长篇大论中是这样总结的：“一个定义起作用的那部分程序叫做该定义的作用范围。一个过程中的变量名如果它的定义的作用范围限于该过程之内，就叫做对该过程是局部的；否则就叫做非局部的。”

全局变量的一个特性是：除非被删除了，否则它的生命期和正在运行的那个脚本程序是一样长的，而它的值可以让所有函数来存取；而局部变量和它们驻留的堆栈架构一样是暂时性的，其生命期只有定义它们的函数的处于活跃期的时间那么长。当一个函数被调用的时候，它的局部变量从被定义的位置开始其作用范围，一旦函数结束堆栈框架被收回，该变量也就结束了它的作用范围。

```
global_str = 'foo'

def foo():
    local_str = 'bar'
    return global_str + local_str
```

在上面的例子里，`global_str`是一个全局性的变量，而`local_str`是一个局部变量。`foo()`函数能够对这两个变量进行访问，而代码的主程序块却只能对全局变量进行访问。

编程提示：检索标识符（包括变量、名字等）

[CN]

在需要检索一个标识符的时候，Python会先在局部作用范围内进行检索。如果在该局部作用范围里没有找到那个名字，则它必须能够在全局访问里被找到，否则就会引发一个`NameError`例外。一个变量的作用范围与它所驻留的名字空间（namespace）是相关联的。我们将在下一章里正式讨论名字空间方面的内容；现在只要明白名字空间就是把名字和对象映射到一起去的命名区域，或者变量名正在其中被使用的虚拟区域的集合就可以了。作用范围这个概念与用来查找某个变量的名字空间检索顺序相关联。当一个函数被调用的时候，其局部名字空间中的所有名字都处于局部作用范围内；如果要查找一个变量，第一个检索的就是这个名字空间。如果在那里没有找到，也许会找到一个那个名字的全局性变量。这些变量被保存在（并检索于）全局名字空间和内建名字空间里。

创建一个局部变量可能会“隐藏”或者覆盖一个全局变量。要知道局部名字空间在其局部作用范围内是第一个被检索的。如果在那里找到了那个名字，检索过程是不会继续去查找有无一个全局变量的，这就等于覆盖了全局名字空间和内建名字空间里任何与它同名的变量。

此外，在使用与全局变量同名的局部变量方面要持谨慎态度。在一个函数里，如果在给局部变量赋值之前使用了这样的变量，就会引发一个例外，根据所使用的Python版本的不同，它会是`NameError`例外或`UnboundLocalError`例外。

11.8.2 global语句

如果在函数里又定义了一个与某个全局变量同名的局部变量，那前者就会被后者覆盖。下面这个例子与第一个例子很相似，但其中全局和局部的性质不那么明显。

```
def foo():
    print "\ncalling foo()..."
    bar = 200
    print "in foo(), bar is", bar

bar = 100
print "in __main__, bar is", bar
foo()
print "\nin __main__, bar is (still)", bar
```

它会给出如下所示的输出：

```
in __main__, bar is 100
calling foo()...
in foo(), bar is 200
in __main__, bar is (still) 100
```

局部的bar把全局性的bar推出了局部作用范围。要想特意引用某个特定的全局变量，需要使用global语句。global语句的语法如下所示：

```
global var1[, var2[, . . . varN ] ]
```

我们可以把刚才例子中的代码修改为使用全局性的is_this_global变量而不是那个在函数中新创建的局部变量，如下所示：

```
>>> is_this_global = 'xyz'
>>> def foo():
...     global is_this_global
...     this_is_local = 'abc'
...     is_this_global = 'def'
...     print this_is_local + is_this_global
...
>>> foo()
abcdef
>>> print is_this_global
def
```

11.8.3 作用范围到底有几个

Python语言在语法上支持多层函数的嵌套；但它最多只允许同时有两个作用范围，即某个函数的局部作用范围和全局性的作用范围。即使存在多层的函数嵌套，用户也不能访问两个以上的作用范围。如下所示：

```
def foo():
    m = 3
    def bar():
        n = 4
        print m + n
    print m
```

```

    bar()

>>> foo()
Traceback (innermost last):
  File "<interactive input>", line 0, in ?
  File "<interactive input>", line 7, in foo
  File "<interactive input>", line 5, in bar
NameError: m

```

在函数bar()里访问foo()函数中的局部变量m是非法的,这是因为m被定义为foo()的局部变量。从bar()函数里能够访问的作用范围只有bar()的局部作用范围和全局作用范围两个。foo()的局部作用范围是不包括在这两个里面的。需要注意的是“print m”语句的输出操作是没有问题的,失败的只是对bar()函数的调用罢了。(注意:今后的Python版本可能会改变这种情况。)

11.8.4 作用范围的其他特性

1. 作用范围和lambda

Python的lambda表达式和标准函数一样遵守着同样的作用范围规则。一个lambda表达式将定义一个新的作用范围,就像一个函数定义一样;因此除那个局部的lambda/函数以外,程序中的其他部分是不能访问那个作用范围的。

在某个函数中被定义为是局部的那些lambda表达式只能在该函数内访问;而lambda语句中的表达式的访问范围也和函数一样。换句话说,它们能够访问全局变量,但不能访问其他的局部作用范围。读者可以把函数和lambda表达式看做是孪生兄弟。

```

>>> x = 10
>>> def foo():
...     y = 5
...     bar = lambda :x+y
...     print bar()
...     y = 8
...     print bar()
...
>>> foo()
Traceback (innermost last):
  File "<interactive input>", line 0, in ?
  File "<interactive input>", line 4, in foo
  File "<interactive input>", line 3, in <lambda>
NameError: y

```

在上面的例子里,虽然lambda表达式是在函数foo()的局部作用范围内创建的,但它只能访问两个作用范围,即它的局部作用范围和全局作用范围(请参考11.8.3节)。为了纠正这一点,我们可以在lambda表达式里放上一个局部变量z,用z来引用函数中的局部变量y。如下所示:

```

>>> x = 10
>>> def foo():
...     y = 5
...     bar = lambda z:x+z
...     print bar(y)
...
...     y = 8
...     print bar(y)
...
>>> foo()

```

15
18

2. 变量的作用范围和名字空间

根据我们对本章的学习，我们已经知道在任何给定的时刻都会有一个或两个有效的活跃作用范围——不多也不少。如果我们处于一个模块的顶层，就只能访问全局作用范围；如果我们在执行一个函数，就能够访问它的局部作用范围和全局作用范围。那名字空间和作用范围又是怎样联系着的呢？

从11.8.1节中的编程注释里还可以看出：在任何给定的时刻都会有两个或者三个名字空间。如果是在一个函数里，局部作用范围对应于局部名字空间，即第一个检索名字的地方。如果名字在那里，就会跳过对全局作用范围（涉及全局名字空间和内建名字空间）的检查。对于全局作用范围（即任何函数之外的地点）来说，检索一个名字将先从全局名字空间开始；如果没有找到，就会到内建名字空间里继续检索。

我们在这里给大家准备了一个混有各种作用范围的脚本程序，即程序示例11-8。至于这个程序的输出，我们留给读者做为一个练习。

程序示例11-8 变量的作用范围 (scope.py)

局部变量会隐藏全局变量，这一点可以从这个研究变量作用范围的程序里看出来。那么，这个程序的输出是什么呢？（为什么？）

```
1  #!/usr/bin/env python
2  j, k = 1, 2
3
4  def proc1():
5
6      j, k = 3, 4
7      print "j == %d and k == %d" % (j, k)
8      k = 5
9
10 def proc2():
11
12     j = 6
13     proc1()
14     print "j == %d and k == %d" % (j, k)
15
16
17 k = 7
18 proc1()
19 print "j == %d and k == %d" % (j, k)
20
21 j = 8
22 proc2()
23 print "j == %d and k == %d" % (j, k)
```

11.9 *递归

如果一个函数里面又包含了这个函数本身，我们就把它称做是递归的。Aho、Sethi和Ullman定义到：“如果同一个过程在前面一次执行过程结束之前又开始了一次新的执行过程，这个过程就是递归的。”换句话说，在一个函数结束执行之前，同一函数的一次新的执行过程又发生在这个函数的内部了。

递归在语言识别方面和使用递归函数的数学应用程序方面非常有用。我们在本章的前面向大家介绍了阶乘运算函数，它的定义如下所示：

$$N! = \text{factorial}(N) = 1 * 2 * 3 * \dots * N$$

我们也可以这样看待阶乘运算函数：

```
factorial(N) = N!
              = N * (N-1)!
              = N * (N-1) * (N-2)!
              :
              = N * (N-1) * (N-2) * ... * 3 * 2 * 1
```

现在就可以看出阶乘运算是递归的了，因为 $\text{factorial}(N) = N * \text{factorial}(N-1)$ 。换句话说，要想计算出 $\text{factorial}(N)$ ，必须先计算出 $\text{factorial}(N-1)$ ，而它又需要先计算出 $\text{factorial}(N-2)$ ，以此类推。

下面是采用递归算法的阶乘运算函数：

```
def factorial(n):
    if n == 0 or n == 1: # 0! = 1! = 1
        return 1
    else:
        return (n * factorial(n-1))
```

11.10 练习

11-1 参数。请比较下面三个函数：

```
def countToFour1():
    for eachNum in range(5):
        print eachNum,

def countToFour2(n):
    for eachNum in range(n, 5):
        print eachNum,

def countToFour3(n=1):
    for eachNum in range(n, 5):
        print eachNum,
```

从结果输出的角度看，给定下面这些输入值，这几个函数会是怎样的情况？请把输出情况填写到表11-2里去。如果给定的输入会引起错误，请填写“ERROR”，如果不会有输出，请填写“NONE”。

表11-2 练习11-2的输出情况表

输入	countToFour1	countToFour2	countToFour3
2			
4			
5			
(无输入)			

11-2 函数。把练习5-2中的两个解决方案组合在一起，即编写一个组合式的函数，它以同样的两个数作为输入，同时返回它们的和与积。

11-3 函数。在这个练习里，我们来编写max()和min()这两个内建函数。

a) 编写简单的max2()和min2()函数，它们以两个数据项为输入，分别返回两个数据项中那个比较大的和那个比较小的。它们应该能够处理任何Python对象。举例来说，max(4, 8)和min(4, 8)将分别返回8和4。

b) 利用练习a) 中max()和min()函数的解决方案重新编写出新的函数my_max()和my_min()。新函数将分别返回非空序列中最大和最小的数据项。请用数字和字符串检验你的解决方案。

11-4 返回值。为练习5-13的解决方案编写一个补充函数。编写一个函数把以分钟为单位的时间值作为输入数据，返回与之对应的以小时和分钟计的时间值。

11-5 缺省参数。修改练习5-7中的销售税脚本程序，使销售税率不再是函数必要的输入数据。在没有给出销售税率的情况下，请在程序里把你当地的税率用做缺省参数。

11-6 可变长参数。编写一个名为printf()的函数。要求有一个位置参数，即一个格式字符串；其余的是需要根据格式字符串中的值显示到标准输出的可变长参数，其中必须允许使用特殊的字符串格式操作符指令如%d、%f等。提示：不用弄得太复杂——不要求实现字符串操作符的功能，但你必须明确地用到字符串格式操作符(%)。

11-7 用map()进行函数化程序设计。给定两个长度相同的列表，比如说[1, 2, 3, ...]和['abc', 'def', 'ghi', ...]，把这两个列表合并为一个列表，新列表中的元素由表列组成，每个表列由各列表位置相同的两个元素构成，最后结果应该像[(1, 'abc'), (2, 'def'), (3, 'ghi'), ...]这样。（虽然这个练习题和第6章里的一个问题很相似，但它们的解决方案之间没有什么直接的内在联系。）

11-8 用filter()进行函数化程序设计。用练习5-4的解决方案的代码判定闰年。把该代码修改为一个函数。然后编写代码以一组年份为输入，返回一个只包括闰年年份的列表。

11-9 用reduce()进行函数化程序设计。复习11.7.2节里利用reduce()函数累加一组数字的代码。以它为基础编写一个名为average()的新函数计算一组数字的简单平均值。

11-10 用filter()进行函数化程序设计。在UNIX文件系统里，在每一个文件夹/目录里都有两个特殊的文件，它们一个是代表当前目录的“.”，一个是代表父目录的“..”。了解这一情况后，请阅读os.listdir()函数的有关文档并说出下面这段代码是干什么用的：

```
files = filter(lambda x : x and x[0] != '.', os.listdir(folder))
```

11-11 用map()进行函数化程序设计。编写一个程序，它以一个文件名为参数，用去掉各行前面多余空格的办法“整理”该文件。读入原始文件，再写到一个新文件里去，既可以创建一个新文件，也可以覆盖写入现有的那个文件。请让你的用户挑选到底要按两个办法中的哪一种进行处理。

11-12 把函数用做参数。按照本章中的说明编写一个testit()函数的姐妹函数。这次不是为查错目的而测试性地执行程序了，timeit()以一个函数对象（以及其他必要的参数）为输入参数，对执行该函数需要花费的时间进行计时。要求返回以下几个值：函数的返回值、花费的时间。你可以使用time.clock()和time.time()方法，它们都能向你提供更高的精度。

11-13 用reduce()和递归进行函数化程序设计。在第8章里，我们看到N阶乘即N!就是1到N

之间所有数字的乘积。

a) 请编写一个名为`mult(x, y)`的简单小函数，它以`x`和`y`为输入参数，返回的是这两个数字的积。

b) 请用练习a)里编写出来的`mult()`函数加上内建函数`reduce()`来计算阶乘。

c) 完全弃用`mult()`函数，用一个`lambda`表达式来完成这个任务。

d) 我们在本章里向大家提供了一个求阶乘 $N!$ 的递归算法。请用你在前面练习里编写的`timeit()`函数对这三个阶乘函数（遍历、`reduce()`和递归）进行计时。请对其执行性能做出解释，要包括预期情况和实际情况。

11-14 递归。我们在第8章里还介绍了菲波那契数列。请重新编写以前的求菲波那契数列的解决方案（练习8-9），要求使用递归的办法。

11-15 递归。重新编写练习6-5的解决方案，用递归的办法把一个字符串以反顺序打印出来。请用递归的办法正序和反序地打印一个字符串。

第12章 模 块

本章焦点集中在Python模块和数据如何从模块导入到程序的运行环境中等几个方面。我们还会涉及到软件包的概念。模块是组织Python代码的方法，软件包则帮助你组织模块。本章将以对与模块相关的其他方面的讨论结束。

12.1 什么是模块

模块允许用户逻辑化地组织自己的Python代码。当代码膨胀到足够大时，就应该在功能方面依然保持彼此交互作用的前提下把它分割成便于管理的小段。这些小段其属性之间有一些千丝万缕的联系，也许是一个带成员数据变量和方法的类（class），也许是一组相关但彼此独立的操作函数。这些小段是共享的，所以Python语言允许“调入”一个模块，允许使用其他模块的属性以利用以前的工作成果，尽可能让代码能够反复重复地使用。这个使其他模块中的属性与用户模块相互结合的过程叫做“导入”（import）。那些自我包容和自我组织管理的能够被共享的Python代码块（把它们打包在一起）就构成了一个模块（module）。

12.2 模块和文件

如果模块代表着组织Python代码的逻辑方法，那么文件就是物理性地组织模块的方法。从这一角度来看，每个文件都可以被看做是一个模块，每个模块也可以被当做是一个文件。一个模块的文件名是由模块的名字加上文件后缀.py构成的。什么样的文件结构意味着模块呢？在这一方面需要讨论的问题有好几个。在其他程序设计语言里用户导入的是类（class），但在Python语言里用户导入的是模块和模块的属性。

12.2.1 名字空间基本概念

我们将在本章的后面再详细讨论名字空间（namespace）的方方面面，但从基本概念来说，一个名字空间就是一个从名字到对象的映射关系集合。我们已经很明确地知道，模块名在给它们的属性命名时扮演的是一个非常重要的角色。属性的名字总是附着在模块名的后面。举例来说，string模块里的atoi()函数就被称为是string.atoi()。给定一个模块名，就只能有一个模块被加载到Python语言的解释器里去，因此不同模块中的名字是不会出现交叉现象的；因此我们就可以说每个模块都定义了它们自己独一无二的名字空间。如果我在自己的模块（比如说是mymodule模块）里也创建了一个名为atoi()的函数，那它的名字将是mymodule.atoi()。这样，即使某个的名字发生冲突，它的完整授权名字（fully-qualified name）（即引用某个对象时使用的属性名字的点记号形式）也不会一样，这就避免了真的发生这类的冲突。

12.2.2 搜索路径和路径搜索

导入一个模块的过程要求有一个叫做“路径搜索”的操作过程。这是在文件系统“预先指定的区域”里查找mymodule.py文件以加载mymodule模块的一个操作过程。这些预先指定的区域无非是Python搜索路径中的一组目录而已。路径搜索和搜索路径是有区别的两个概念，前者指的是查找某个文件的操作，后者指的是去查找文件的地方（一组目录）。

有时候，导入一个模块的操作会以失败而告终：

```
>>> import xxx
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ImportError: No module named xxx
```

发生错误时，解释器会告诉用户它无法访问用户指定的模块，其原因多半是那个模块没有在搜索路径里，从而导致了路径搜索的失败。

编译过程或者安装过程会自动定义一个缺省的搜索路径，这个搜索路径可以在一两个地方修改。

一个地方是在启动Python的shell即命令行解释器里设置的PYTHONPATH环境变量。这个变量的内容由一组用冒号分隔开的目录路径构成。如果读者打算让解释器使用这个变量的内容，就一定要在启动解释器或者运行一个Python脚本程序之前设置或者修改好这个变量。

解释器启动之后，用户就可以访问该路径本身，它是以sys.path变量的形式保存在sys模块里的。但这时它已不再是用冒号分隔的字符串了，该路径已经被分断为一个一个的目录字符串。下面例子里就是一台UNIX机器上的搜索路径。记住，搜索路径随系统的不同肯定不是一样的。

```
>>> sys.path
['', '/usr/local/lib/python1.5/', '/usr/local/lib/python1.5/plat-sunos5', '/usr/local/lib/python1.5/lib-tk', '/usr/local/lib/python1.5/lib-dynload']
```

注意，这只是一个列表，我们可以随时随地对它进行修改。如果用户知道自己需要某个模块，但它的目录没有在搜索路径上，只需用列表的append()方法把它追加到这个路径列表里就行了，如下所示：

```
sys.path.append( '/home/wesc/py/lib' )
```

上面这个操作完成后，用户就可以加载自己的模块了。只要搜索路径中的某个目录里包含着那个文件，就可以导入该模块。当然，这个办法只能把目录追加在搜索路径的尾部。如果用户想把它加在其他位置，比如搜索路径的开始或者中间，就需要使用列表的insert()方法来操作。在上面的例子里，我们对sys.path属性的修改是以交互方式进行的，用运行脚本程序的办法也可以达到这个目的。

下面是采用交互方式执行时遇到这个问题时的情况：

```
>>> import sys
>>> import mymodule
Traceback (innermost last):
  File "<stdin>", line 1, in ?
```

```

ImportError: No module named mymodule
>>>
>>> sys.path.append('/home/wesc/py/lib')
>>> sys.path

['', '/usr/local/lib/python1.5/', '/usr/local/lib/
python1.5/plat-sunos5', '/usr/local/lib/python1.5/lib-
tk', '/usr/local/lib/python1.5/lib-dynload', '/home/
wesc/py/lib']
>>>
>>> import mymodule
>>>

```

从反面看，用户可以保存了一个模块的许多拷贝。如果出现这样的情况，解释器加载的将是它以给定名字沿搜索路径的顺序找到的第一个模块。

12.3 名字空间

一个名字空间就是一个名字（标识符）到模块的映射关系。把一个名字添加到一个名字空间的操作过程涉及到把该标识符“绑定”到那个对象的动作（还要给那个对象的引用线索加上1）。Python Language Reference中有这样的定义：“改变一个名字的绑定叫做重新绑定，删除一个名字叫做解除绑定”。

我们在前一章里已经做过简单的介绍：在执行期间任何一个给定的时刻有两或三个活跃有效的名字空间。这三个名字空间分别是局部名字空间、全局名字空间和内建名字空间，但因为局部名字空间只在执行期间有效，所以说“有两或三个”活跃有效的名字空间。能够从这些名字空间中访问的名字依赖于它们的加载顺序，即把名字空间加载到系统里去的顺序。

Python解释器最先加载的是内建名字空间，它由__builtins__模块里的名字构成。随后加载的是那个执行模块的全局名字空间，它会在那个模块开始执行时成为活跃有效的名字空间。这样我们就有了两个活跃有效的名字空间。

编程提示：__builtins__模块和__builtin__模块的比较

[CN]

请不要把__builtins__模块与__builtin__模块弄混了。这两个模块的名字确实是很容易弄混的，对那些Python程序员新手来说更是如此。__builtins__模块由内建名字空间中那些内建的名字的集合构成。这些名字中的大多数（如果不是全部的话）都来自__builtin__模块，这是一个包含着内建函数、内建例外以及其他内建属性的模块。在标准化的Python执行过程中，__builtins__模块包含着所有来自__builtin__模块的名字；只有在限制执行模式下才有一些区别（我们将在第14章正式讨论限制执行模式）。在限制执行模式下，__builtins__模块只包含有在该限制执行模式内能够访问的来自__builtin__模块的名字的子集。

如果在执行期间又调用了一个函数，那将创建出第三个即局部名字空间。我们可以通过globals()和locals()内建函数判断出哪个名字属于哪个名字空间。这两个函数将在本章后面的内容里做详细讲解。

12.3.1 名字空间与变量作用范围的比较

好了，我们已经知道名字空间是什么东西了，那它们与变量的作用范围又有这样的关系呢？它们看起来很相似，而事实也确实如此。

名字空间是纯粹意义上的名字和对象之间的映射关系，作用范围却指出了从用户的代码里如何或者说从哪儿才能根据其物理保存位置访问到这些名字。我们用图12-1中的示意图来说明名字空间和作用范围之间的关系。

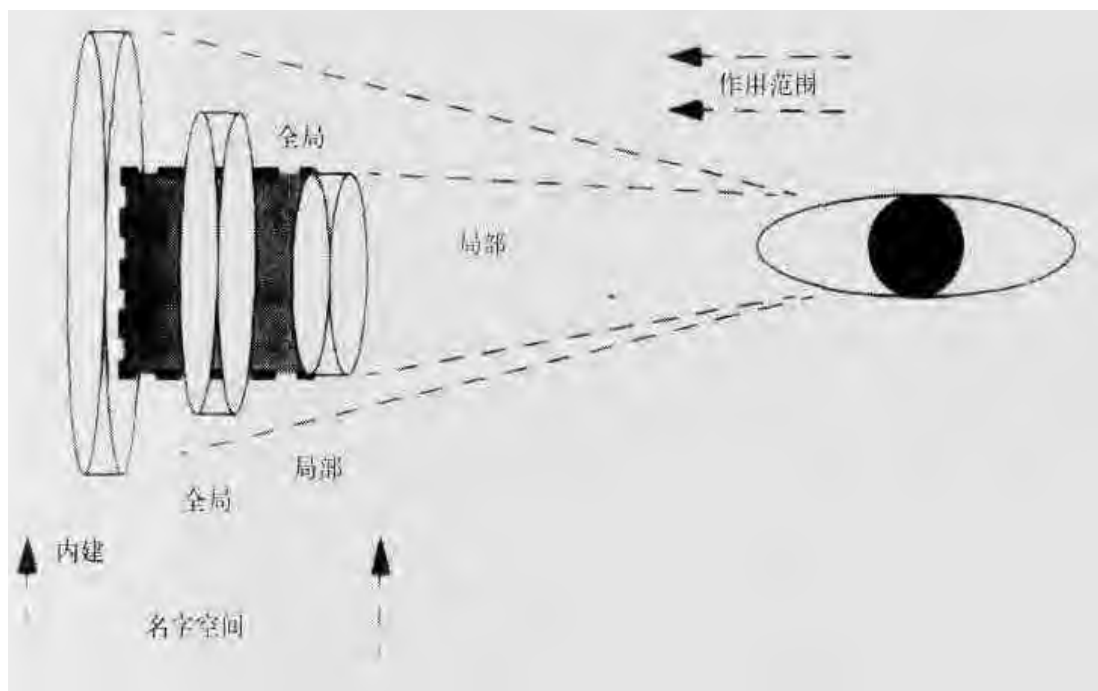


图12-1 名字空间与变量作用范围的比较

请注意，每个名字空间都是一个自我包容的单元；但从作用范围的角度来看名字空间就可以看出其中的奥妙。局部名字空间里的所有名字都在局部作用范围内；局部作用范围以外的所有名字都在全局作用范围内。

还要提醒大家的是在程序的执行期间，局部名字空间和局部作用范围会随着函数调用的开始结束而发生变化，而全局名字空间和内建名字空间则保持稳定不变。

学习完这一小节，我们建议读者在涉及到名字空间时问自己这样一个问题“它存在吗？”；而在涉及到变量作用范围时问自己“我能看见它吗？”

12.3.2 名字的查找、确定作用范围和覆盖

那么，确定变量作用范围的规则是如何作用于它与名字空间的关系上的呢？那就是通过名字查找操作。在对一个属性进行访问的时候，它的名字必须是解释器能够在这三个名字空间中的某个里找到的。检索从局部名字空间开始。如果属性没有在那里被找到，就将对全局名字空间进行检索。如果那里也没有成功，就要去内建名字空间里检索。如果这最后的检索也失败了，用户就会看到下面这个熟悉的错误：

```
>>> foo
```

```
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: foo
```

请注意，这个错误信息体现了最先检索的名字空间是如何“遮蔽”了随后检索的名字空间的。这是为了试图把后果看做是“名字的覆盖”。名字的覆盖指的是那些个名字可能来自当前作用范围以外，其原因是该名字出现在不止一个名字空间里。请看下面这段在前一章里介绍过的代码：

```
def foo():
    print "\ncalling foo()..."
    bar = 200
    print "in foo(), bar is", bar

bar = 100
print "in __main__, bar is", bar
foo()
```

执行这段代码，我们将得到如下所示的输出：

```
in __main__, bar is 100

calling foo()...
in foo(), bar is 200
```

foo()函数的局部名字空间中的bar变量覆盖了全局性的bar变量。虽然bar变量存在于全局名字空间里，可名字的查找操作先在局部名字空间里找到了一个后就不再去检索全局名字空间了，这就等于“覆盖”了那个全局性的变量。关于作用范围的细内容请参考11.8节。

12.4 导入模块

导入一个模块要使用import语句，它的语法如下所示：

```
import module1[, module2 [, ... moduleN ]]
```

当解释器遇到这条语句时，如果在搜索路径里找到了指定的模块，就会加载之。它要遵守作用范围规则，也就是说，如果是从一个模块的顶层执行的导入操作，它的作用范围就是全局性的；如果是从一个函数里执行的导入操作，它的作用范围就是局部性的。

如果一个模块是第一次被导入，它将被加载和执行。

12.4.1 模块加载时的执行情况

加载一个模块的后果之一是这个被导入的模块将要被执行，也就是说，被导入的模块的顶层代码部分将会直接执行。它通常包括对该模块全局变量进行设置和对该模块中的类（class）和函数进行定义等操作；如果还通过检查__name__变量准备好在该脚本程序被直接调用时多干一些事情，那些工作也会被执行。

当然，这类执行过程是也可能不是用户想要的效果。如果不想，用户就必须把尽可能多的代码放到函数里面去。我们必须指出：良好的模块化程序设计风格要求应该只把函数定义和/或类（class）定义放在一个模块的顶层。

12.4.2 导入与加载的比较

不管模块被导入多少次，它只会被加载一次。这样做是为了防止出现多次导入操作导致该模块多次被“执行”的现象。假设你的模块导入了sys模块，而你导入的其他五个模块也需要导入它，很明显多次导入sys模块（或者多次导入其他模块）是不明智的！所以，无论导入多少次，加载和执行只发生在第一次被导入时。

12.5 导入模块属性

只把某些个特定的模块元素导入到用户自己的模块里去也是可行的。这就等于是说把某些个特定的名字从某个模块里导入到当前的名字空间里去。这要用到from-import语句，它的语法如下所示：

```
from module import name1[, name2[, . . . nameN ] ]
```

12.5.1 把名字导入当前名字空间

调用from-import可以把名字导入当前的名字空间里去意味着用户可以不使用属性/点记号来访问模块的标识符。举例来说，如果想访问用下面的语句导入的模块module中的一个变量名var，可以直接使用var这个变量名本身。

```
from module import var
```

因为我们刚导入了module模块，所以就不必再引用其模块名了。

用下面的from-import语句可以把一个模块中的所有名字都导入到当前的名字空间里去：

```
from module import *
```

编程风格：尽量少使用“from module import *”语句

[CS]

在实践中，“from module import *”的用法被认为是不好的编程风格，因为它“污染”了当前的名字空间，并有可能覆盖当前名字空间中现有的名字；但如果某个模块里确实有大量需要经常访问的变量，或者该模块的名字很长，这个办法也确实是非常方便。

我们建议只在下面两种情况下使用这个办法。第一种情况是目标模块中的属性非常多，使得反复敲入模块名很不方便。两种最常见的这类例子是Tkinter（Python/Tk）和Numpy（Numeric Python）模块，也许还要算上socket模块。另外一个比较适合使用“from module import *”的地方是在交互式解释器中，因为这样可以少敲不少键盘。

12.5.2 被导入到导入者作用范围的名字

只从其他模块里导入名字的另一个副作用是被导入的那些名字现在成为了导入者模块的作用范围的组成部分。这就意味着对这些变量的修改将只影响到局部的这份拷贝而不会影响到被导入模块的名字空间中原来的那些变量。换句话说，绑定操作现在已经是局部性的了，不再跨越名字空间。

下面，我们给出了两个模块的代码，它们分别是导入者`impter.py`和被导入者`imptee.py`。`impter.py`里使用了只创建局部绑定关系的`from-import`语句。如下所示：

```
#####
# imptee.py #
#####
foo = 'abc'
def show():
    print 'foo from imptee:', foo

#####
# impter.py #
#####
from imptee import foo, show
show()
foo = 123
print 'foo from impter:', foo
show()
```

运行这个导入者脚本程序，我们发现即使在导入者代码里修改了被导入模块中的`foo`变量，在被导入模块眼里，它自己的`foo`变量并没有发生什么变化。如下所示：

```
foo from imptee: abc
foo from impter: 123
foo from imptee: abc
```

这个问题的解决办法是在导入操作中用上以属性/点记号方式表示的完全授权标识符名 (fully-qualified identifier name)。如下所示：

```
#####
# impter.py #
#####
import imptee
imptee.show()
imptee.foo = 123
print 'foo from impter:', imptee.foo
imptee.show()
```

修改完毕并相应地修改了引用线索之后，得到的结果就是我们预料中的了：

```
foo from imptee: abc
foo from impter: 123
foo from imptee: 123
```

12.6 模块的内建函数

系统为模块的导入提供了一些功能上的支持。我们现在就来看看它们。

12.6.1 `__import__()`

[1.5]

`__import__()`函数是从Python 1.5版本开始新增加的，它就是实际完成导入操作的那个函数；也就是说，`import`语句是靠调用`__import__()`函数完成自己的工作的。增加这样一个函数的目的是：如果需要自行开发导入操作算法的话，人们可以覆盖掉它。

`__import__()`函数的语法如下所示:

```
__import__(module_name[, globals[, locals[, fromlist : ] ]])
```

`module_name`变量是将被导入的模块的名字, `globals`是全局符号表里那些当前名字的字典, `locals`是局部符号表里那些当前名字的字典, 而`fromlist`是将用`from-import`语句导入的那些符号构成的一个列表。

`globals`、`locals`和`fromlist`参数都是可选的, 如果没有给出它们, 就分别以`globals()`、`locals()`和`[]`为缺省值。

调用“`import sys`”可以用下面的语句实现:

```
sys = __import__('sys')
```

12.6.2 `globals()`和`locals()`

`globals()`和`locals()`内建函数分别返回调用者的全局和局部名字空间的字典。如果是在一个函数内, 局部名字空间代表的就是为该函数的执行而定义的所有名字, 也就是`locals()`返回的东西。而`globals()`返回的当然就是对该函数来说是可全局性访问的那些名字了。

如果对全局名字空间同时调用`globals()`和`locals()`, 它们返回的字典将是完全一样的; 这是因为在全局名字空间里执行程序时, 它对该程序来说也就是局部的。下面是从两个名字空间里调用两个函数的一小段代码:

```
def foo():
    print '\ncalling foo()...'
    aString = 'bar'
    anInt = 42
    print "foo()'s globals:", globals().keys()
    print "foo()'s locals:", locals().keys()

print "__main__'s globals:", globals().keys()
print "__main__'s locals:", locals().keys()
foo()
```

我们只要求输出字典键字, 这是因为这里的讨论与键值并没有什么关系 (而加上它们也只会使输出行上出现更多的文字而已)。执行这个脚本程序, 我们得到如下所示的输出:

```
% namespaces.py
__main__'s globals: ['__doc__', 'foo', '__name__',
'__builtins__']
__main__'s locals: ['__doc__', 'foo', '__name__',
'__builtins__']

calling foo()...
foo()'s globals: ['__doc__', 'foo', '__name__',
'__builtins__']
foo()'s locals: ['anInt', 'aString']
```

12.6.3 `reload()`

`reload()`内建函数的作用是把一个以前导入过的模块再导入一次。`reload()`函数的语法如下所

示:

```
reload(module)
```

`module`是将要导入的模块的名字。使用`reload()`模块有一些限制条件。第一条是模块必须是曾经被完整地导入过的（不是用`from-import`语句导入的），并且必须已经成功地被加载了。接下来是第二条规则，说的是`reload()`的参数必须是模块本身而不是一个包含有该模块名字的字符串，也就是说，它必须是`reload(sys)`这样的而不能是`reload('sys')`这样的。

此外，模块中的代码将在被导入时执行，但只执行一次。后面的导入不会再次执行那些代码，它只绑定模块名字。这也就是`reload()`的意义所在，因为它将覆盖这个缺省行为。

12.7 软件包

[1.5]

一个软件包就是一个继承性的文件目录结构，它定义了一个由各种模块和子软件包组成的Python应用程序环境。软件包概念新出现于Python 1.5版本，用来帮助解决包括下列情况在内的问题：

- 为扁平性质的名字空间加上继承性（树状）的组织结构。
- 使开发人员能够把相关模块归到一个组里去。
- 使发行商能够利用目录有条理地组织大批文件。
- 帮助解决冲突的模块名字。

除类（`class`）和模块外，软件包还使用大家熟悉的属性/点属性记号来访问它们的元素。在软件包内部导入模块可以使用`import`和`from-import`语句。

12.7.1 目录结构

为了方便大家的学习，下面例子里的软件包目录结构已经包含在书后的CD-ROM光盘里了。大家直接浏览代码样本继续学习第12章内容即可。

```
Phone/
  __init__.py
  Voicedta/
    __init__.py
    Pots.py
    Isdn.py
  Fax/
    __init__.py
    G3.py
  Mobile/
    __init__.py
    Analog.py
    Digital.py
  Pager/
    __init__.py
    Numeric.py
```

`phone`是顶层的软件包，而`Voicedta`等是子软件包。导入子软件包要像下面这样使用`import`语句：

```
import Phone.Mobile.Analog
```

```
Phone.Mobile.Analog.dial()
```

用户还有多种from-import办法可以选择使用。

第一个办法是只导入最顶层的子软件包，再沿着子软件包树结构用属性/点记号引用需要的属性，如下所示：

```
from Phone import Mobile
Mobile.Analog.dial('555-1212')
```

我们还可以再深入一层子软件包进行引用：

```
from Phone.Mobile import Analog
Analog.dial('555-1212')
```

事实上，用户可以一直沿着子软件包的树结构走到底，如下所示：

```
from Phone.Mobile.Analog import dial
dial('555-1212')
```

在我们上面的目录树状结构中，我们看到有一系列的__init__.py文件。这是一些用from-import语句导入子软件包时所要求的初始化模块。虽然有的只是空文件，但还是需要有它们存在。

12.7.2 软件包的from-import语句操作

软件包也支持from-import语句的全导入操作，如下所示：

```
from package.module import *
```

对Python语言来说，这样的一条语句过于依赖于操作系统的文件系统了，因此不好确定到底需要导入哪一个文件。这就需要有__init__.py中的__all__变量。如果需要用上面的语句完成某个导入操作，这个变量里就包含着需要导入的全部模块的名字。它是一个由各模块名构成的字符串。

12.8 模块的其他特性

12.8.1 自动加载模块

当Python解释器以标准模式启动时，解释器会自动加载某些模块供系统使用。其中唯一会影响到用户的是__builtin__模块，它通常会以__builtins__模块的形式被加载。

sys.modules变量包含着解释器当前（完整且成功地）加载到解释器中的模块构成的一个字典。模块的名字被用做键字，它们从哪儿被加载的位置被用做键值。

举例来说，在Windows里，sys.modules变量包含着大量已经被加载的模块，为了缩短清单的长度，我们要求只列出模块的名字就行。我们可以用字典的keys()方法做到这一点，如下所示：

```
>>> import sys
>>> sys.modules.keys()
['os.path', 'os', 'exceptions', '__main__', 'ntpath',
```

```
'strop', 'nt', 'sys', '__builtin__', 'site',
'signal', 'UserDict', 'string', 'stat']
```

在UNIX中导入的模块与上面的结果也很相似:

```
>>> import sys
>>> sys.modules.keys()
['os.path', 'os', 'readline', 'exceptions',
 '__main__', 'posix', 'sys', '__builtin__', 'site',
 'signal', 'UserDict', 'posixpath', 'stat']
```

12.8.2 阻止某个属性的导入

当使用“from module import *”导入一个模块时, 如果不想导入模块的某个属性, 需要在模块里该名字的前面加上一个下划线字符(_)。被导入模块中以下划线字符打头的名字都不会被导入。如果导入的是整个的模块, 这个小小的数据隐蔽技巧就不管用了。

12.9 练习

12-1 路径搜索和搜索路径的比较。路径搜索和搜索路径之间有什么区别?

12-2 导入属性。假设你在自己的模块mymodule里有一个名为foo()的函数。要想把这个函数导入到你的名字空间里有哪两种办法?

12-3 导入。“import module”和“from module import *”有什么区别?

12-4 名字空间和变量作用范围的比较。名字空间和变量的作用范围有什么样的区别?

12-5 使用__import()函数。

a) 用__import()函数把一个模块导入到你的名字空间里去。能够达到目的的正确语法是什么?

b) 同上, 但这次使用__import()函数的目的是从模块里只导入某些特定的名字。

12-6 扩展的导入操作。编写一个新的名为importAs()的函数。这个函数可以把一个模块或者几个模块导入到你的名字空间里去, 但被导入模块的名字将被替代为你指定的名字, 不使用它们原来的名字。举例来说, 调用“newname = importAs('mymodule')”将导入模块mymodule, 但访问这个模块及其属性时都必须采用newname或newname.attr的形式。这正是从Python 2.0版本开始引进的新扩展的import语法所提供的功能。

[2.0]

第13章 类和 OOP

类 (class) 把面向对象的程序设计 (object-oriented programming, OOP) 最终带到了我们的眼前。我们将先给大家做一个全面的概述, 介绍在Python里使用类和OOP的各主要方面。然后对类、类实例 (class instance) 以及方法 (method) 的各个细节做详细的论述。我们还将描述如何在Python里推导或者说分离出子类, 以及它们的继承模型是什么。最后, Python语言提供有特殊的属性以帮助程序设计人员为类定制特殊的功能, 比如说对操作符进行重载的功能和模仿Python类型的功能等。我们将向大家演示如何实现并用这些特殊方法来定制你的类以获得与类型相类似的操作行为。

13.1 简介

开始进入错综复杂的OOP和类的世界之前, 我们先从全面的概述开始, 然后用几个简单的例子给大家“热热身”。如果你是第一次接触到面向对象的程序设计, 可以跳过这一小节, 从13.2节开始正式的学习。这一小节的内容主要是为那些已经熟悉有关概念、但想知道Python的具体工作原理的读者准备的。

面向对象的Python程序设计中最主要的两个概念是类 (class) 和类实例 (class instance) (请参考图13-1)。

1. 类和实例

类和实例彼此是相互关联的: 类提供了对一个对象的定义, 而实例则是根据类的定义实际产生出来的对象。

下面是如何创建一个类的例子:

```
class MyNewObjectType:
    'define MyNewObjectType class'
    class_suite
```

class是一个关键字, 它后面是类的名字。接下去是对这个类进行定义的代码子句, 通常由各种各样的定义和声明组成。创建一个实例的过程叫做实例化过程, 具体实现办法如下所示:

```
myFirstObject = MyNewObjectType()
```

类的名字加上函数操作符 (()) 就像大家熟悉的函数名那样被用来调用这个类。通常需要把这个新创建的实例赋值给一个变量。从语法角度来说这个赋值操作并不是必需的, 可如果你不把你的实例保存到一个变量中去, 它就不会有什么用处, 并且会因为这个实例上没有任何引用线索而自动被回收——你做的事情只是分配了内存, 然后又立刻被收回去了。

类可以很简单也可以根据需求变得很复杂。在最简单的情况下, 类可以被用做“名字空间的包容器”, 我们的意思是你可以把数据保存在变量里, 然后对这些变量加以组织, 使它们具备同样的关系——即使用标准的Python点属性记号表示的关系。举例来说, 你可能有一个没有任

何继承属性的类，并且只把这个类用来为数据提供名字空间，这使你的这个类类似于Pascal语言中的记录（record）或者C语言中的结构（structure）：或者，换句话说，把这个类简单地用做一个带有共享的命名操作的包容器对象。

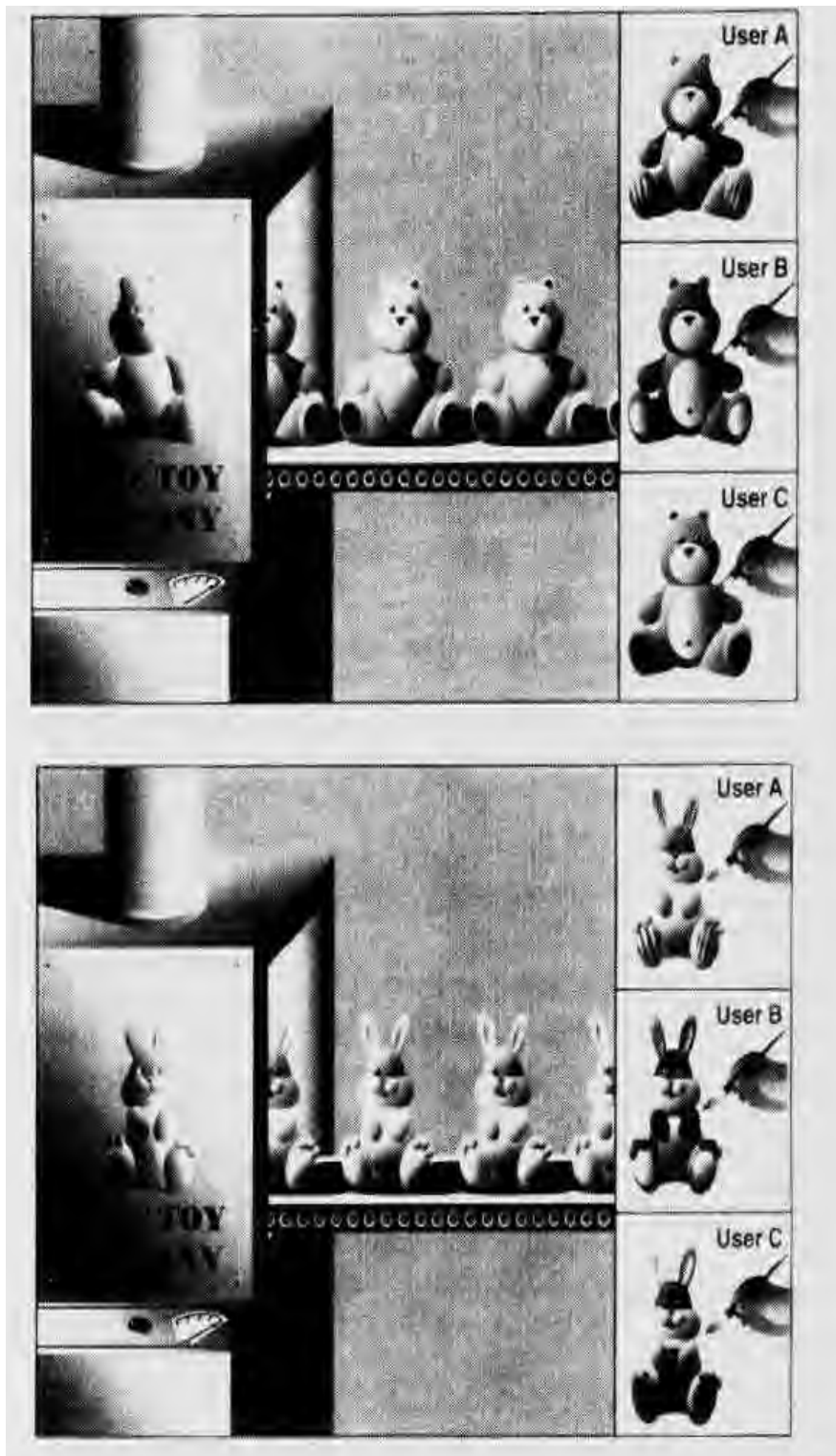


图13-1 图中左边的机器就好比是类，生产出的每个玩具就是其对应类的实例。

虽然每个实例的基本内在构造都一样，但个别属性如颜色和手感会有所变化——这些就类似于实例的属性。

下面是一个这样的类的例子:

```
class MyData :
    pass
```

大家应该还记得pass语句是被用在语法上要求出现一个语句的地方的。在这个例子里, 要求出现的代码是类定义子句, 但我们这里没有给出来。我们定义的这个类没有任何方法和其他属性。下面我们将创建一个实例, 把这个类简单地用做一个名字空间的包容器。如下所示:

```
>>> mathObj = MyData()
>>> mathObj.x = 4
>>> mathObj.y = 5
>>> mathObj.x + mathObj.y
9
>>> mathObj.x * mathObj.y
20
```

我们当然可以直接用变量“x”和“y”完成同样的事情, 但在我们的这个例子里, mathObj.x和mathObj.y通过实例的名字mathObj联系在一起了。这就是我们“把类用做名字空间的包容器”的含义。mathObj.x和mathObj.y叫做实例的属性(instance attribute), 因为它们是它们的实例对象(mathObj)仅有的属性, 不是类(MyData)的属性。

2. 方法 (method)

改进类的用途的办法之一是给它们加上函数。这些类的函数有一个更常用的名字叫做方法。在Python语言里, 方法的定义也是类定义的一部分, 但只能通过一个具体的实例进行调用。换句话说, 调用一个方法的必由之路是: (1) 定义那个类(和方法); (2) 创建一个实例; (3) 从该实例调用那个方法。下面是一个带有一个方法的类的例子:

```
class MyDataWithMethod:      # define the class
    def printFoo(self):      # define the method
        print 'You invoked printFoo()!'
```

请注意参数self必须出现在所有的方法调用中, 这个参数代表的是那个实例对象。当你通过一个实例调用某个方法的时候, self参数将由解释器隐含地传递到那个方法去; 所以你本人不必操心传递不传递参数了。现在把这个类实例化, 在获得实例后开始调用这个方法, 如下所示:

```
>>> myObj = MyDataWithMethod() # create the instance
>>> myObj.printFoo()           # now invoke the method
You invoked printFoo()!
```

在本小节的末尾我们准备了一个稍微复杂点的例子让大家了解通过类(和实例)都可以完成哪些工作, 让大家对特殊方法__init__()、分离子类的操作、类的继承性等概念有一个初步的了解。

那些比较熟悉面向对象的程序设计的读者应该知道__init__()就是类的构造器(class constructor)。刚接触OOP世界的读者请注意: 一个构造器简单来说就是一个在实例化操作期间被调用的特殊方法, 它定义了一个类在实例化时将要采取的额外操作行为, 比如说设置初始值或执行一些基本的诊断性代码等——基本上都是在一个实例被创建以后但在从实例化调用返回之前完成一些必要的特殊任务。

我们在我们的方法里加上了一些print语句，这样在某特定方法被调用的时候大家能够看得更清楚。一般说来，如果输出工作不在代码主体事先安排好的范畴内，在函数里加上输入或者输出语句是不常见的做法。

3. 创建一个类（类的定义）

```
class AddrBookEntry:                                # class definition
    'address book entry class'
    def __init__(self, nm, ph):                    # define constructor
        self.name = nm                            # set inst .attr .1
        self.phone = ph                           # set inst .attr .2
        print 'Created instance for:', self.name

    def updatePhone(self, newph):                  # define method
        self.phone = newph
        print 'Updated phone# for:', self.name
```

在对AddBookEntry类的定义中，我们定义了两个方法，即__init__()和updatePhone()。__init__()将在实例化期间，也就是调用AddBookEntry()的时候被调用。读者可以把这样一个实例化调用想象为对__init__()的一个隐含的调用，因为在调用AddBookEntry()里给出的参数和__init__()接收到的参数是完全一样的。

从一个实例调用一个方法的时候，self（实例对象）参数将由解释器自动传递过去，所以上面例子中的__init__()方法只要求nm和ph两个参数，这两个参数分别代表姓名和电话号码。__init__()方法会在实例化的时候对这两个实例属性进行设置，这样当该实例从实例化调用当中返回的时候，程序员就可以直接使用它们了。

读者可能已经看出updatePhone()方法是用来替换地址簿里某个数据项的电话号码属性的。

4. 创建实例（实例化）

```
>>> john = AddrBookEntry('John Doe', '408-555-1212')
Created instance for: John Doe
>>> jane = AddrBookEntry('Jane Doe', '650-555-1212')
Created instance for: Jane Doe
```

这就是实例化调用，再由它去调用__init__()方法。别忘了，有一个实例对象self会自动地传递进去。所以读者可以在脑子里把各方法里的self替换为实例的名字；因此，在第一个例子里，当对象john被实例化的时候，被设置的实际上是john.name，并且可以在下面得到验证。

另外，因为__init__()方法的两个参数都不是缺省参数，所以在实例化调用时必须明确地给出这两个参数。

5. 访问实例的属性

```
>>> john
<__main__.AddrBookEntry instance at 80ee610>
>>> john.name
'John Doe'
>>> john.phone
'408-555-1212'
>>> jane.name
'Jane Doe'
```



```
>>> jane.phone
'650-555-1212'
```

一旦创建了我们的实例，我们就可以验证出我们的实例属性实际是由__init__()方法在实例化期间设置的。在解释器内调用该实例会告诉我们它到底是哪一种对象。我们将在以后介绍如何定制我们的类才能看到一个更有意义的输出信息而不是看到缺省的Python对象字符串“<...>”。

6. 方法的调用（通过实例）

```
>>> john.updatePhone('415-555-1212')
Updated phone# for: John Doe
>>> john.phone
'415-555-1212'
```

updatePhone()方法明确要求有一个参数，即新的电话号码。在updatePhone()调用结束后我们立刻对实例的属性进行了检查以确认它完成了它应该完成的事情。

到现在为止，我们只是通过一个实例调用了一个方法，就像上面的例子里那样。在Python语言里这些方法被称为“绑定方法”。绑定是一个Python术语，它指出我们是否有一个实例来调用一个方法。

7. 创建一个子类

继承性地创建一个子类指的是这样一种操作：它创建和定制一个新的类的类型（class type），这个类类型具有一个现有类的全部特性，但不必修改原来那个类的定义。新的子类可以定制有只对新的类类型唯一有效的特殊功能。除了它与其父类（parent class或base class）的关系以外，一个子类具有正常的类应该具有的全部特性，并且也会像所有其他类一样被实例化。在下面的例子里，一个父类被用做其子类定义的一部分：

```
class AddrBookEntryWithEmail(AddrBookEntry): # define subclass
    'update address book entry class'
    def __init__(self, nm, ph, em):           # new __init__
        AddrBookEntry.__init__(self, nm, ph) # base class cons.
        self.email = em
    def updateEmail(self, newem):             # define method
        self.email = newem
        print 'Updated e-mail address for:', self.name
```

现在我们创建了我们的第一个子类AddrBookEntryWithEmail。在Python语言里，衍生子类时子类将继承父类的属性；所以，在上面的例子里，除我们定义的方法__init__()和updateEmail()以外，AddrBookEntryWithEmail还从AddrBookEntry那里继承了updatePhone()方法。

如有必要，每个子类必须定义它自己的构造器，否则就会调用它父类的构造器。但如果一个子类覆盖了一个父类的构造器，父类的构造器就不会被自动调用了——这一请求必须明确地做出，就像我们在上面做的那样。就我们的子类来说，在完成任何“局部”任务之前，我们最先调用了父类的构造器，即用AddrBookEntry.__init__()设置好姓名和电话号码。我们的子类设置了另外一个实例属性电子邮件地址，是用新构造器的其余代码行设置的。

需要注意的是我们明确地把self实例对象参数传递给了父类的构造器，这是因为我们并不是

在一个实例里调用那个方法的。我们是在某个子类的一个实例里调用那个方法的。因为我们不是通过一个实例调用的那个方法，所以这个非绑定方法的调用要求我们传递一个可接受的实例(self)到该方法中去。

我们将以一个具体的例子结束本小节的学习，它创建一个子类的实例、访问它的属性并且调用它的方法，包括从父类中继承得到的方法。

8. 使用一个子类

```
>>> john = AddrBookEntryWithEmail('John Doe', '408-555-1212', 'john@spam.doe')
Created instance for: John Doe
>>> john
<__main__.AddrBookEntryWithEmail instance at 80ef6f0>
>>> john.name
'John Doe'
>>> john.phone
'408-555-1212'
>>> john.email
'john@spam.doe'
>>> john.updatePhone('415-555-1212')
Updated phone# for: John Doe
>>> john.phone
'415-555-1212'
>>> john.updateEmail('john@doe.spam')
Updated e-mail address for: John Doe
>>> john.email
'john@doe.spam'
```

编程风格：给类、属性和方法命名

[CS]

类的名字传统上要以一个大写字母开始。这种标准化的做法可以帮助你记住那些标识符是类的标识符，这在实例化操作（它多少有些像一个函数调用）期间更有用。在实践中，数据属性的读音应该和数据值的名字相像，方法的名字应该指明对某个特定对象或值的操作动作。另一种办法是；数据值的名字用名词表示，而方法则用动词加上直接对象构成。数据项是你作为程序员对它进行操作的对象，而方法应该指明程序员打算在那个对象上准备采取的动作。

在我们上面定义的类里，我们尽量遵守这一原则，数据值的名字都是name、phone和email等；方法的名字都是updatePhone、updateEmail等有意义的词。其他一些好的数据值名字有data、amount或balance等；推荐使用的方法名字包括getValue、setValue和clearDataset等。类也该取个好名字；比较好的名字有AddBookEntry、RepairShop等。

我们希望读者对如何使用Python语言实现面向对象的程序设计已经有了一定的了解。本章的其他小节将带领大家深入面向对象的程序设计的方方面面和Python语言中的类和实例。

13.2 面向对象的程序设计

程序设计方法上的进步使我们摆脱了单控制流中逐条指令的顺序性而进入一个更有组织结构的境界；在这里，代码块可以从程序中“切割”下来用一个名字保存到子程序和新定义的功能

能里去。结构化或过程化的程序设计思想使我们能够把自己的程序组织为逻辑块，这些逻辑块能被反复和重复地利用。编写应用程序也成为更符合逻辑的过程：根据需要挑选操作动作，再创建出适合操作动作处理的数据。《Deitel & Deitel》根据逻辑必须在没有任何行为关联着的数据上“动作化”这一事实认为：结构化的程序设计其实是“面向操作动作的”。

那么，如果我们能够给数据加上操作行为又怎么样呢？如果我们能够对源自实际生活的某个数据模型进行创建和编程，把数据方面的特性和操作方面的行为结合到一起又会怎么样呢？如果我们能够通过一组事先定义好的操作接口（比如说一组访问器函数）对数据的属性进行访问，就好像一张ATM自动柜员机上使用的卡或者一张能够存取你银行户头的个人支票那样，我们就会拥有一个由各种“对象”构成的系统——那些“对象”不仅可以与其自身交互作用，还可以与更大范围内的其他对象交互作用。

面向对象的程序设计就迈出了这进化的一步，它对结构化程序设计进行了改进，在数据和操作行为之间建立起有机的联系：数据和逻辑被高度地抽象为一个描述，并且通过这个抽象本身来创建这些对象。现实世界中的问题和事物被揭露出它们的本来面目，呈现在人们面前的是抽象化的问题和事物。人们根据这些抽象对它们进行编程或在对象中实现它们，这些对象又能与系统中其他模块的其他对象交互作用，而最终的结果是这些问题得到了解决。类（class）就提供了对这些对象的定义，实例就是这些定义的具体实现。这两样是面向对象的设计（object-oriented design, OOD）中最关键的两件东西；OOD的含义很简单，它指的是以一种面向对象的方式来构筑系统的体系结构。

13.2.1 OOD和OOP之间的联系

面向对象的设计并不要求必须有一种面向对象的程序设计语言。事实上，OOD完全可以用纯粹的结构化程序设计语言如C语言来实现，但这种做法需要程序员花费更多的精力去建立精确的数据结构，让那些数据结构具有对象的结构和特性。而一种具有内建的OO特性的程序设计语言自然能够简化OOP工作，它会使OO程序的开发更流畅、更迅速。

反过来说，面向对象的程序设计语言并不是要强迫人们去编写OO程序。C++完全可以简单地被当作一种“更好的C语言”来使用。毫无疑问，用Python开发程序并不意味着每天都要用到类和OOP方法。虽然Python从其自身的设计之初就是一种面向对象的程序设计语言，具备支持OOP的构造，可它并没有规定或者要求用户必须用OO代码编写自己的应用程序。应该承认，OOP是工具箱里一件功能强大的工具，当你准备好前进到OOP时，不管你的目的是学习、使用、软件移植，还是其他什么东西，都可以随时取出并应用它。Python语言的创始人经常把这个现象称为“透过树木看森林”。

13.2.2 现实世界中的问题

考虑使用OOD进行工作的一个最重要的原因是：它提供了建立模型和解决现实世界里问题和情况的一种直接的手段。我们来尝试建立一个汽车修理厂的模型，人们将把自己的汽车送到这里来修理。我们需要创建两个事物：存在于这样一个“系统”的里面并且彼此打交道的“人”；用来完成修理厂中的工作活动的物理“地点”。“人”又分为各种各样不同类型的许多

“人”，所以我们先来把它说清楚，然后再说“地点”。

先创建一个名为Person（人）的类来代表这一活动涉及到的所有人员。Person的实例将包括顾客（Customer）、机修工（Mechanic），也许还有收银员（Cashier）。这些实例又有相似和各自独一无二的行为。比如说，它们都应该有一个用来与其他人交流的talk()方法和一个驾驶汽车的drive_car()方法；但只有机修工有修理汽车的repair_car()方法，只有收银员有计价收款的ring_sale()方法。机修工必须有一个repair_certification（修车证书）属性，全体Person都应该有一个drivers_license（驾驶执照）属性。

最后，所有这些实例都将被归纳到一个名为修理厂（RepairShop）的类里去，它要有一个用来确定什么时候顾客可以送车来修理、什么时候机修工和收银员等雇员（Employee）必须来上班的operating_hours（营业时间）数据属性。修理厂还可以有一个名为自动线（AutoBay）的类，这个类有诸如尾气区（SmogZone）、补胎区（TireBrakeZone）等实例，也许还会有一个大修（GeneralRepair）实例。

我们虚构出来的这个修理厂例子是为了让大家明白如何通过类、实例及其操作行为把实际生活中的问题抽象为一个模型。读者可以自己归纳出飞机场、饭店、商店、医院甚至邮购音乐公司等行业的模型来，里面要有相应的人员和功能。

13.2.3 抽象世界里的模型

介绍了与OOP有关的一些基本概念后，我们来看看Python都是这样做的：

1. 抽象/实现

抽象指的是针对真实世界中问题和事物的基本方面、行为和特性来建立模型的过程，这个过程综合问题或事物主要因素形成一个相互关联的子集，也就是对一个程序化的结构的定义，而这个程序化的结构就是用来识别这个模型的东西。抽象过程不仅包含了这样一个对象的数据属性，还定义了这些个数据的操作接口。经抽象过程得到的一个实现就是数据和伴随数据而存在的操作接口一起构成的一个真实体现。这个真实体现不涉及程序员个人，与他们也没有什么必然的联系。Python中的类（class）对象使我们能够建立起这样的抽象过程，而实现方面的细节由设计者负责解决。

2. 封装/操作界面

封装这个概念指的是隐藏数据/信息本身和提供数据属性的操作接口，也就是访问器函数这样一个过程。任何绕过操作接口直接访问数据的客户端操作都是与封装原则相抵触的，但它允许程序员进行这样的数据访问。确定并构造出数据的属性是实现过程的一部分工作，客户甚至不必去了解抽象过程是如何完成这些工作的。在Python语言里，所有的类属性都是公开的，但名字可能会“乱成一团”以尽量阻止未经授权的访问——但也不是完全禁止这样做。应该由设计者负责提供合适的数据操作接口，客户端程序员不必操心处理和封装数据属性的问题。

3. 构造

构造引申了我们对类的描述，它指的是把多个彼此互相区别的类组合在一起构成一个更大的实体以解决真实世界中的问题。构造过程描述了一个单一的复杂系统，比如一个由其他更小的组件（包括其他的类、数据属性和操作行为等）组成的一个类，所有这些组合在一起，体现

了一种“拥有”性的联系。举例来说，RepairShop“拥有”Mechnic（至少有一个），还“拥有”Customer（也至少有一个）。

构造这些组件的办法有两种，一是通过关联，即拥有子组件的访问权（就拿RepairShop来说吧，一个顾客可以进来要求做一次SmogCheck（尾气检查），也就是客户程序员要和RepairShop的组件打交道）；二是添加被封装的组件，而新添加的组件只能透过定义好的操作接口进行访问，具体情况对客户存在与否来说还是不可见的。还用刚才的例子，客户程序员可以通过Customer要求做一次SmogCheck，但不能和RepairShop的SmogZone部分打交道，SmogZone部分只能通过RepairShop的内部控制机制在调用SmogCheckCar()方法的时候才能访问到。Python语言支持这两种构造办法。

4. 推导/继承性/继承树

推导描述的是子类的创建过程。子类是这样一个新的类：它保留了一个现有的类类型中它自己需要的数据和操作行为，但又做了一些修改和定制；这些修改后定制不涉及或影响原来那个类的定义。继承性指的是在子类里保留并沿用祖先类中的属性的方法。在我们前面的例子里，一个Mechnic可以比一个Customer拥有更多汽车修理技术方面的属性；因为每个Mechnic又“是”一个Person，所以也可以合法地调用talk()方法；talk()是每个Person的实例都具备的操作。继承树描述的是多“世代”的推导，用“家谱”一样的树状图可以把后继子类与祖先类的关系清楚地表示出来。

5. 变身（或可变性）

不管是哪个类，其中的对象都可以用它们拥有的一般属性和操作行为去处理和访问，从而产生许多看起来完全不同、可实际上属于同一个类的对象；变身概念指的就是这样的情况。变身又引出了动态（也可以叫即时、运行期间）绑定的概念，即在代码的执行过程中允许名字的重载、允许动态确定和验证对象的类型。如果出现方法被重载的情况，不少OO语言都会使用“签字”来确定将要调用它的哪一个版本；但Python调用都是不进行类型确定的通用性调用，重载是不必要的，也不被这种语言支持。

6. 探查/反射

探查是为程序员提供的完成诸如“人工类型检查”这样的活动的的能力，它也叫做反射。它描述的是在代码执行期间某个特定的对象访问关于自身的信息。如果能够拥有把某个对象传递给你并确定它是用来干什么的这样一种能力，对你来说不是很好吗？这是读者将在本章经常会遇到的一种功能强大的特性。如果Python语言没有对探查能力提供某些支持，内建函数dir()和type()工作起来就会很困难。请留意这两个函数和__dict__、__name__、__doc__、__members__、__methods__等几个特殊属性的调用情况。对其中的几个你可能已经很熟悉了！

编程提示：Python语言里使用的术语“对象”和“类型”

[CN]

在其他面向对象的程序设计语言里，术语“对象”可能专门用来指类的实例，如果这种语言的所有数据类型都是类的话，情况就更是如此。但Python语言不这样做。因为Python语言中的数据类型不是类，所以并非所有的对象都是类的实例。还有些语言把定义一个类看做是创建一个新类型的同义词。但Python语言还是没有这样做，但它的做法也差

不多。Python语言里预先定义好的类型的个数是固定的，并且它们不会有什么变化。（在Python语言里创建一个新类型可不是件小事情，需要把它实现为一个扩展；具体做法超出了我们这里的讨论范围。）我们可以在创建类的时候给它加上类型的行为特征，可它们依然不能被看做是类型。

可无论如何，Python是一种面向对象的程序设计语言，并且把所有事物一视同仁地都看做是对象——尽管它们确实有一些共同的特征，但彼此之间的差异依然可以足以让我们把它们看做是不同类型的对象。总而言之，类、实例、类型之间彼此是没有联系的（例外情况是一个由类出发定义的对象将被认为是一个实例，实例是另外一种类型的对象）。

原则：（全部的）类是类（class）对象；（全部的）实例是实例（instance）对象；它们都不是类型，任何东西都是一个对象。请参考13.5.1节中的编程提示。

13.3 类

一个类就是一个可以用来定义同时具备值和操作行为两方面特征的对象的数据结构。类是把真实世界中的问题抽象化以后得到的程序化的表示形式；而实例则是这样一个对象的具体实现。打个比方，类就像是蓝图或者模子，实例就是用蓝图或者模子制造出来的产品。那么，为什么把它们称为“类”呢？我想大概是从生物学上借用来的：生物学用类来区分生物物种，即可以对生物进行归类，又可以进一步细分为相似但又有区别的子类。这些做法对程序设计方面的“类”概念也同样适用。

在Python语言里，对类的定义和对函数的定义是很相似的，都是一个带相应的关键字的标题行后面跟着一个做为其定义的子句，如下所示：

```
def functionName(args):
    'function documentation string'
    function_suite

class ClassName:
    'class documentation string'
    class_suite
```

这样的定义看上去当然要比对一个标准类型的定义要“大”，这个事实多少可以证明Python语言中的类所涉及到的东西要比标准类型多。一个类就好比是一个Python容器类型。它不仅能够容纳多个数据项，还可以支持一组自己的函数，这些函数通常被称为方法。那么，与标准的容器类型如列表和字典相比，类又有哪些其他的优点呢？

标准类型是固定的，不能进行定制，其属性都是一成不变的。数据类型不会为对象个别提供名字空间，也不能用来推导出“子类型”。包含在列表中的对象除了它们容器的名字相同之外，彼此之间毫无联系。列表成员只能通过下标在一个类似于数组的数据结构里进行访问。所有列表都有完全相同的方法。字典方面的情况也是一样，只不过另有一组常用的方法，并且通过键字来访问其成员（成员之间除了容器名字以外还是毫无联系）。

在这一小节，我们将仔细研究类和类拥有的属性。但要记住，虽然类也是对象（Python里的

每种东西都是一个对象)，可它们并不是它们定义的对象实现。我们将在后面对实例做详细论述，现在不必着急；让我们把注意力集中到类（class）对象身上。

当你创建类的时候，实际上是在创建你自己的数据实体。同一个类的所有实例都是相似的，但类与类之间就有区别了（不同的类的实例当然也就有区别了）。与其玩别人送给你的玩具，为什么不自己做一个玩呢？

13.3.1 类的创建

在Python语言里，创建类要使用class关键字。在简单格式的类定义里，类的名字紧跟在这个关键字的后面，如下所示：

```
class ClassName:
    'class documentation string'
    class_suite
```

class_suite由构成类定义的一切语句组成，它定义了类的成员、数据属性和函数。类通常是在一个模块的最顶层定义的，这样就可以在定义了类的源代码中的任何位置来创建类的实例。

13.3.2 声明和定义的比较

和Python语言中的函数一样，对类的声明和定义没有任何区别，这是因为它们是同时发生的，也就是说，定义（类的子句）是紧跟在声明（带class关键字的标题行）和文档字符串（虽然可选，但我们建议读者永远要加上这个字符串）后面的。同样地，所有的方法也必须在这个时候进行定义。如果读者熟悉OOP术语，就会知道Python语言不支持纯虚拟函数（pure virtual function，C++语言）或抽象方法（abstract method，Java语言），这二者允许程序员在某个子类里定义一个方法。

13.4 类的属性

什么是属性？属性是属于另外一个对象并且能够通过大家熟悉的点属性记号访问的数据或函数元素。有些Python类型如复数带有数据属性（即分别代表实数部分和虚数部分的real和imag），列表和字典带有方法（即函数属性）。

关于属性有一点要提醒大家注意：当你访问一个属性的时候，因为它也是一个对象，所以它可能会有自己的属性，这些属性也可以访问；这样就形成了一个属性链，比如myThing.subThing.subSubThing等。大家比较熟悉的例子有：

```
• sys.stdout.write('foo')
• print myModule.myClass.__doc__
• myList.extend(map(upper, open('x').readlines()))
```

类的属性只属于对它们进行定义的类。因为实例对象是使用OOP工作时每天最经常用到的对象，所以实例的数据属性是你将会用到的最基本的数据属性。类的数据属性只有在要求使用不依赖任何实例的更“静态的”数据类型时才有用，所以我们把下一小节定为选读的高级内容。

在随后的小节里，我们将简单介绍在Python语言里如何实现和调用方法。一般说来，Python语言中所有的方法都遵守同样的规定：在调用它们之前必须先有一个实例。

13.4.1 类的数据属性

类的数据属性简单说来就是类里面的变量。它们是在创建某个类时设置的变量，可以在创建了那个类的程序环境里象其他变量一样使用；但修改它们的值却只能在类的内部或者程序的主程序部分用方法来进行。

OO程序员通常都把这些属性称为静态成员、类变量或静态数据。它们代表的数只属于对它们进行定义的类，并且独立于任何类的实例。如果你是一位C++或Java程序员，这种数据就相当于在变量定义的前面加上一个static关键字。

静态成员通常都用来保存与类相关联的值。在大多数情况中，实例属性要比类属性用得更多一些；在正式介绍实例时我们将比较这二者的区别。

下面是使用了一个类的数据属性（foo）的例子：

```
>>> class C:
...     foo = 100

>>> print C.foo
100
>>> C.foo = C.foo + 1
>>> print C.foo
101
```

请注意，在上面的代码里看不到任何类的实例的引用线索。

13.4.2 方法

一个方法，比如下面例子里MyClass类里的myNoActionMethod方法，简单说来就是在类定义中定义的一个函数（这使方法成为了类的属性）。这意味着myNoActionMethod只能对MyClass类型的对象（实例）进行操作。请注意myNoActionMethod是怎样附着在它的实例上的，因为调用要求用户在点属性记号里给出这两个名字，如下所示：

```
>>> class MyClass:
...     def myNoActionMethod(self):
...         pass
```

```
>>> myInstance = MyClass()
>>> myInstance.myNoActionMethod()
```

myNoActionMethod把自己做为一个函数进行调用时会失败：

```
>>> myNoActionMethod()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
    myNoActionMethod()
NameError: myNoActionMethod
```

因为全局名字空间里没有一个这样的函数，所以这里引发了一个NameError例外。这样做是为了提醒大家myNoActionMethod是一个方法，也就是说它属于那个类，不是全局名字空间中的一个名字。如果myNoActionMethod在顶层被定义为一个函数，我们的调用就会成功了。

请看下面的例子，用一个class对象来调用方法是会失败的：


```
>>> MyClass.myNoActionMethod()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
    MyClass.myNoActionMethod()
TypeError: unbound method must be called with class
instance 1st argument
```

这个TypeError乍看上去有点让人摸不着头脑，因为我们知道这个方法是类的一个属性，所以它的调用失败让人搞不明白。下面就来解释这个问题。

1. 绑定操作（绑定和未绑定方法）

为了保持OOP传统，Python语言里有这样的规定：方法不能在不通过实例的情况下被调用。要想完成方法的调用，必须要有一个实例。这条规定也说明了Python语言中绑定操作的概念，即方法必须被绑定（到一个实例）后才能被直接调用。未绑定方法也可以调用，但为了保证调用的成功，必须明确给出一个实例对象。但即使不考虑绑定操作，方法也是定义它们的那个类的继承性属性，虽然它们几乎总是通过一个实例来调用的。我们将在13.7节对绑定和未绑定方法做进一步解释。

2. 静态方法

Python语言不支持静态方法（也就是静态成员函数）——即只与一个类相关联却没有与任何实例相关联的函数。它们或者是帮助管理类的静态数据的函数，或者是具有与定义它们的类有关的某些功能的全局函数。因为Python语言不支持静态方法，所以在需要有静态方法提供的功能时可以用一个全局函数来绕过这一限制。达到这一目的的具体细节请参考13.7.2节内容。

13.4.3 确定类的属性

有两种办法可以确定一个类里都有哪些属性。最简单的办法是使用dir()内建函数。另一个办法是访问类的字典属性__dict__，它是每个类都有的特殊属性之一。我们来看看下面的例子：

```
>>> class MyClass:
...     'MyClass class definition'
...     myVersion = '1.1'           # static data
...     def showMyVersion(self):    # method
...         print MyClass.myVersion
... 
```

用上面定义的类做例子，我们分别使用dir()内建函数和类的特殊属性__dict__查看一下这个类的属性：

```
>>> dir(MyClass)
['__doc__', '__module__', 'showMyVersion', 'myVersion']
>>>
>>> MyClass.__dict__
{'__doc__': None, 'myVersion': 1, 'showMyVersion':
<function showMyVersion at 950ed0>, '__module__':
'__main__'}
```

可见，dir()返回的是一个对象的属性的列表；而__dict__返回的是一个字典，其中属性的名字被用做键字，它的键值是相应属性的数据值。

输出结果里有两个大家比较熟悉的类MyClass的属性showMyVersion和myVersion，还有几个新面孔即__doc__和__module__，是每个类都具备的特殊属性（再加上__dict__）。如果给var()内建函数传递一个类（class）对象做为它的参数，它就会返回那个类的__dict__属性的内容。

13.4.4 类的特殊属性

对一个类C来说，表13-1列出了C的全部特殊属性：

表13-1 类的特殊属性

C.__name__	类C的字符串名字
C.__doc__	类C的文档字符串
C.__bases__	类C的父类的表列
C.__dict__	类C的属性
C.__module__	对类C进行定义的模块（从1.5版本开始的新功能）

用我们刚才定义的类MyClass做例子，我们得到如下所示的输出：

```
>>> MyClass.__name__
'MyClass'
>>> MyClass.__doc__
'MyClass class definition'
>>> MyClass.__bases__
()
>>> MyClass.__dict__
{'__doc__': None, 'myVersion': 1, 'showMyVersion':
<function showMyVersion at 950ed0>, '__module__':
'__main__'}
>>> MyClass.__module__
'__main__'
```

__name__是那个指定类的字符串名字。在需要一个字符串而不是一个（类）class对象时，它就会很有用处。甚至有的内建类型也具有这个属性，我们将以一个内建类型为例向大家演示一下__name__字符串的有用之处。

类型（type）对象是一个具有__name__属性的内建类型。type()函数在调用时会返回一个类型（type）对象。有时我们希望得到的是一个指示出类型而不是关于对象的字符串。我们可以用类型（type）对象的__name__属性来获得与之对应的字符串名字。请看下面的例子：

```
>>> stype = type('What is your quest?')
>>> stype
# stype is a type object
<type 'string'>
>>> stype.__name__
# get type as a string
'string'
>>>
>>> type(3.14159265)
# also a type object
<type 'float'>
>>> type(3.14159265).__name__
# get type as a string
'float'
```

__doc__是类的文档字符串，它类似于函数和模块的文档字符串，必须是标题行下面第一个

未赋值字符串。文档字符串不会被推导出来的子类继承，各个子类的文档字符串都是它们自己独有的。

`__bases__`涉及类的继承性，它是由类的父类组成的一个表列；我们将在本章后面的内容里详细讨论它。

前面已经介绍过的`__dict__`属性是一个包含着类的数据属性的字典。在需要访问类的某个属性的时候，就会到这个字典里检索有无那个属性。如果它没有出现在`__dict__`里，就会到这个类的父类的字典里继续检索，检索按“深度优先搜索”算法进行。对那些父类进行的检索操作将按它们在这个子类定义中出现的顺序进行。对一个类属性的修改只能影响它所在的当前类的字典；父类中的`__dict__`属性不能被子类所修改。

Python语言支持跨模块的类继承关系，所以为了更准确地描述类的特性，从1.5版本开始引入了`__module__`属性，在类的名字前加上它的模块名构成完整授权名。请看下面的例子：

```
>>> class C:
...     pass
...
>>> C
<class __main__.C at 81201f0>
>>> C.__module__
'__main__'
```

类C的完整授权名是“`__main__.C`”，也就是“`source_module.class_name`”的形式。如果类C位于一个被导入的模块里，比如说`mymod`模块吧，就会是下面这样的情况：

```
>>> from mymod import C
>>> C
# class C in Python 1.5.2
<class mymod.C at 8120be0>
>>> C.__module__
'mymod'
```

早期的Python版本没有提供特殊属性`__module__`，要想确定某个类的位置相当困难，因为类并不使用它们的完整授权名。举例来说，如果我们执行了同样的模块导入操作并且访问了那个类，你会看到类C上根本没有源模块的名字，就像下面这样子：

```
>>> from mymod import C
>>> C
# class C in Python 1.4
<class C at 8120be0>
```

13.5 实例

一个类是通过对一种数据结构进行定义而得到的类型，而一个实例就是对类定义确定的那个类型的一个变量做出的声明。换句话说，实例是有具体内容的类。蓝图有了，下一步就是把它变成现实了。实例是程序执行期间将被主要使用的对象，所有实例的类型都是一样的，即“instance”。

13.5.1 实例化：调用类对象创建实例

大多数其他的OO语言都为创建某个类的一个实例的操作准备了一个new关键字。但Python语言的办法要简单得多。一旦类被定义好了，创建一个实例的工作并不比调用一个函数要做的事情更困难——至少从字面上看是这样的。实例化可以利用函数操作符来实现，请看下面的例子：

```
>>> class MyClass:           # define class
...     pass
>>> myInstance = MyClass()    # instantiate class
>>> type(MyClass)             # class is of class type
<type 'class'>
>>> type(myInstance)          # instance is of
instance type
<type 'instance'>
```

编程提示：类和实例本身都是普通的类型

[CN]

Python语言使用的术语“类型”和“一个实例的类型就是创建它的类的类型”这句话里的“类型”两者的含义是有区别的。我们在第4章里讲过：所有的Python对象都可以被看做是带着三个特性的数据实体，即一个ID标识、一个类型和一个值。一个对象的类型指出了该种对象在Python系统里都有那些行为方面的属性，并且这些类型都是Python所支持的全体类型的一个子集。

象类这样用户定义的“类型”也是按照相同的办法归类的。类有相同的类型，但彼此有不同的ID和值（即有不同的类定义）。所有的类都是用同一个语法定义的，它们可以被实例化，它们都具有同样的核心属性，从以上这些事实可以得出结论：它们具有同样的特性，可以据此把它们归到同一个分类里去。类只是在定义部分彼此有区别的各个对象而已，在Python语言里它们都是同一个“类型”的事物。类的实例也遵从这个论断。

不要让Python的文字弄混了头；实例与自己从中实例化的那个类是百分之百有联系的，但与别的实例就没有其他任何联系了（除非它们是一个子类或父类）。为了避免混淆，请记住这里的提醒：当你创建一个类的时候，并没有创建出一个新的类型。你只是定义了一个新的类类型（class type），但它依然是一个类。当你对类进行实例化的时候，做为操作结果的对象永远是一个实例。即使实例可以从不同的类进行实例化，它们也依然是类的实例。

正如你所看到的，创建类MyClass的实例myInstance时“调用”了类MyClass()。调用的返回对象是一个实例（instance）对象。接着我们用type()内建函数验证了MyClass和myInstance的数据类型，得到的结果是：MyClass是一个class（类）对象，而myInstance是一个instance（实例）对象。进一步来说，我们要告诉读者：所有的类都是同一个类型（type class）的事物，所有的实例是另外一种同一个类型（type instance）的事物。（你可以用type()内建函数验证这一点。）

13.5.2 __init__() 构造器方法

[1.5]

在调用类的时候，实例化过程中的第一个步骤是创建实例对象。创建好对象后，检查类中

是否实现有构造器。如果类中没有定义新的构造器覆盖原来的__init__()特殊方法,就不对实例采取任何行动。如果想进行任何特殊操作,就需要由程序员自行编写一个__init__()覆盖掉它缺省的操作行为。如果类中没有实现__init__(),就返回新创建的对象,而实例化操作也就结束了。

如果真的实现有__init__(),就调用这个特殊方法,新创建的实例将做为它的第一个参数(self)被传递进去,整个过程就好像一个标准方法的调用一样。传递到类调用中的任何参数都将被传递到__init__()里去。从实践的角度看,你可以把创建类的调用看做是对构造器的一个调用。

__init__()是为类定义的众多特殊方法中的一种。这些特殊方法中有许多都预先把不采取任何操作做为自己的缺省操作(就像__init__()一样),在需要定制的情况下要由程序员覆盖掉;其他的特殊方法也需要根据具体情况另行实现。我们将在本章的13.13节里讨论有关特殊方法的问题。

13.5.3 __del__() 拆除器方法

类似地,还有一个名为拆除器__del__()的特殊方法。因为Python语言自己有一套管理废弃对象回收的办法(通过计算引用的个数),所以在一个实例对象的引用被删除之前是不会调用到这个函数的。Python语言中的拆除器方法将在回收给实例分配的内存之前进行某些特殊的处理;因为实例很少会被明确地删除,所以一般用不着实现这个方法。

编程提示: 对实例进行跟踪记录

[CN]

Python语言没有提供记录一个类创建了多少个实例的内部机制,也没有提供记录这些实例具体情况的内部机制。如果真的需要用到这样的功能,就必须由程序员(也就是读者本人)在类定义里(比如说在__init__()和__del__()里)加上一些代码。最好的办法是用一个静态变量来跟踪记录实例的个数。想通过保存它们的引用来跟踪实例对象的做法是危险的,这是因为这样做要求你(程序员)必须能够正确地管理好所有这些引用,否则就会出现你的实例永远也不能被收回的情况(因为你的实例上有多余的引用)!请看下面的例子:

```
class myClass:
    count = 0                # use static data for count
    def __init__(self):      # constructor, incr. count
        myClass.count = myClass.count + 1
    def __del__(self):       # destructor, decr. count
        myClass.count = myClass.count - 1
        assert myClass.count > 0 # cannot have < 0 instances
    def howMany(self):       # return count
        return myClass.count

>>> a = myClass()
>>> b = myClass()
>>> b.howMany()
2
>>> a.howMany()
2
>>> del b
>>> a.howMany()
```

```

1
>>> del a
>>> myClass.count
0

```

在下面的例子里，我们分别创建（并且覆盖）了构造器__init__()和拆除器__del__()两个函数，然后对类进行实例化并且给同一个对象赋值了多个别名。接着用id()内建函数确认那三个别名引用的都是同一个对象。最后用del语句删除了全部的别名，请注意在这个过程中拆除器被调用的时间和方式。

```

>>> class C:
    # class declaration
    def __init__(self):
        # constructor
        print 'initialized'
    def __del__(self):
        # destructor
        print 'deleted'

>>> c1 = C()
initialized
# instantiation
>>> c2 = c1
# create additional alias
>>> c3 = c1
# create a third alias
>>> id(c1), id(c2), id(c3)
(11938912, 11938912, 11938912)
# all refer to same object
>>> del c1
# remove one reference
>>> del c2
# remove another reference
>>> del c3
# remove final reference
deleted
# destructor finally invoked

```

请看，在上面的例子里，只有在类C的那个实例的全体引用都被删除后（也就是引用计数值被减为零的时候）才调用了拆除器方法。如果在预期会调用__del__()方法的时候因为某种原因没有调用它，就表明实例对象的引用还没有被减到零，还有一些你没有注意到的其他引用使那个对象还存在着。

再要注意的一点是拆除器只会被调用一次，那就是在引用第一次减到零并收回该对象的时候。这样做合情合理，因为相同中的任何对象都只会分配和回收一次。

13.6 实例的属性

实例只有数据属性（方法被规定为类的属性），简单说来，它们是一些与类的某个特定的实例相关联、并且能够通过熟悉的点属性记号访问的数据值。这些值不依赖于任何其他的实例，也不依赖于它本身从中实例化出来的那个类。当一个实例被回收时，它的属性也被回收了。

13.6.1 “实例化”实例的属性

在一个实例被创建出来以后，它的属性可以在能够访问该实例的代码的任意位置被设置任意次。但对这些属性进行设置的关键性位置之一是构造器__init__()。

1. 构造器：设置实例属性的第一地点

构造器是能够对实例属性进行设置的最早位置，这是因为__init__()是实例对象被创建后第一个被调用的方法，没有比这更早的对实例属性进行设置的场所了。一旦__init__()执行完毕，

实例对象就会被返回，从而结束了实例化操作这一过程。

2. 缺省参数提供了缺省的实例设置

人们可以把__init__()和缺省参数结合起来做为用一个实例来解决真实世界问题的有效办法。在许多时候，缺省参数代表着对实例属性进行设置的最常见的情况，而缺省值的这种用法意味着人们不必向构造器明确地给出有关的参数。在11.5.2节里我们已经简要地介绍了一些使用缺省参数的好处。

程序示例13-1演示了如何利用构造器缺省的操作行为帮助我们计算美国都市地区旅馆的房价成本。

程序示例13-1 在实例中使用缺省参数 (hotel.py)

一个虚构旅馆房间租金计算器的类定义。构造器方法__init__()用来设置实例的几个属性。另有一个calcTotal()方法用来计算出房间日租金总数或住宿总天数的总租金。

```
1 class HotelRoomCalc:
2     'Hotel room rate calculator'
3
4     def __init__(self, rt, sales=0.085, rm=0.1):
5         '''HotelRoomCalc default arguments:
6         sales tax == 8.5% and room tax == 10%'''
7         self.salesTax = sales
8         self.roomTax = rm
9         self.roomRate = rt
10
11     def calcTotal(self, days=1):
12         'Calculate total; default to daily rate'
13         daily = round((self.roomRate * \
14             (1 + self.roomTax + self.salesTax)), 2)
15         return float(days) * daily
```

这段代码的主要目的是帮助人们计算旅馆房间的日租金，包括各州的销售税和房屋税。缺省情况针对的是旧金山周围地区，这一地区的销售税是8.25%，房屋税是10%。房屋的日租金没有缺省值，我们必须在创建的所有实例里给出日租金数值。

初始化设置工作由第4到第8行的__init__()在实例化过程中完成，另一个核心代码部分是第10到第18行的calcTotal()方法。__init__()的工作是把确定一间旅馆房间基本租金（不包括房间服务、电话费或其他非固定收费项目）总数所需要的数值设置好。接着用calcTotal()计算出日租金总数；如果给出了住宿天数，就计算出这些天的总租金数。而round()内建函数用来把计算出来的房价结果四舍五入到最接近的分（即小数点后面两位）。下面就是这个类的用法示例：

```
>>> sfo = HotelRoomCalc(299)                # new instance
>>> sfo.calcTotal()                          # daily rate
354.32
>>> sfo.calcTotal(2)                         # 2-day rate
708.64
>>> sea = HotelRoomCalc(189, 0.086, 0.058)   # new instance
>>> sea.calcTotal()
216.22
>>> sea.calcTotal(4)
864.88
>>> wasWkDay = HotelRoomCalc(169, 0.045, 0.02) # new instance
```

```
>>> wasWkEnd = HotelRoomCalc(119, 0.045, 0.02) # new instance
>>> wasWkDay.calcTotal(5) + wasWkEnd.calcTotal()# 7-day rate
1026.69
```

头两个例子是旧金山地区，这里使用了缺省值；接下来是西雅图，这里需要给出不同的销售税率和房屋税率。最后一个例子是首都华盛顿，这里需要计算的是一个住宿天数较长的总租金数：5个工作日加上一个周末特价，星期日离店返家。

缺省参数给函数带来的灵活性也同样适用此方法。在实例中使用可变长参数是另一个好主意（当然要看应用程序的需要而定）。

3. 构造器的返回值必须是None

现在大家都已经知道：用函数操作符调用一个类对象会创建一个类实例，这个实例就是这个调用返回的那个对象，请看下面的例子：

```
>>> class MyClass:
...     pass
>>> myInstance = MyClass()
>>> myInstance
<__main__.MyClass instance at 95d390>
```

如果定义了构造器，它就不应该返回任何对象，这是因为实例化调用会自动返回一个实例对象。因此，`__init__()`不应该返回任何对象（也就是说只能返回None）；否则就是一个冲突，因为应该返回的是实例化操作过程所创建的那个实例。试图返回一个不是None的任何对象都会导致一个TypeError例外。如下所示：

```
>>> class MyClass:
...     def __init__(self):
...         print 'initialized'
...         return 1
...
>>> myInstance = MyClass()
initialized
Traceback (innermost last):
  File "<stdin>", line 1, in ?
    myInstance = MyClass()
TypeError: __init__() should return None
```

13.6.2 确定实例的属性

我们可以用`dir()`内建函数列出实例的全体属性，就像它列出类的属性一样：

```
>>> class C:
...     pass
>>> c = C()
>>> c.foo = 'roger'
>>> c.bar = 'shrubber'
>>> dir(c)
['bar', 'foo']
```

类似于类的情况，实例也有一个`__dict__`特殊属性（也可以通过调用以一个实例为参数的`vars()`来访问），这是一个代表了它的属性的字典：


```
>>> c.__dict__
{'foo': 'roger', 'bar': 'shrubber'}
```

13.6.3 实例的特殊属性

实例只有两个特殊属性（请参考表13-2）。对实例I来说：

表13-2 实例的特殊属性

I.__class__	实例化出I的那个类
I.__dict__	I的属性

我们以类C和它的实例c为例子来看看实例的这些特殊属性：

```
>>> class C:                # define class
...     pass
...
>>> c = C()                 # create instance
>>> dir(c)                   # instance has no attributes
[]
>>> c.__dict__               # yep, definitely no attributes
{}
>>> c.__class__              # class that instantiated us
<class '__main__.C' at 940230>
```

正如你所看到的，c目前还没有任何数据属性；但我们可以给它加上几个，然后再用特殊属性__dict__检查它们是否正确地添加上去了：

```
>>> c.foo = 1
>>> c.bar = 'SPAM'
>>> '%d can of %s please' % (c.foo, c.bar)
'1 can of SPAM please'
>>> dir(c)
['bar', 'foo']
>>> c.__dict__
{'foo': 1, 'bar': 'SPAM'}
```

__dict__属性包含着一个字典，其内容是某个实例的属性。字典的键值是属性的名字，键值是与之对应的属性的数据值。在这个字典里你只能找到实例的属性——里面既没有类的属性也没有特殊属性。

编程风格：修改__dict__

[CS]

虽然类和实例的__dict__属性实际上是属于可变类型的一个字典，但我们建议除非真的知道自己到底想干什么，否则就不要对这个字典进行修改。这样的修改会“污染”你的OOP操作环境，有可能引起预料不到的副作用。通过熟悉的点属性记号来访问和操作属性要更稳妥一些。虽然很少会遇到，但有的时候确实需要由你本人直接对__dict__属性进行修改，情形之一是你准备覆盖__setattr__()特殊方法的时候。实现__setattr__()方法本身就是另外一个冒险，到处是无限递归和实例对象崩溃等问题的陷阱——现在还不用考虑这件事情。

13.6.4 内建类型的属性

内建类型也有属性，虽然从技术的角度看它们不是类的实例的属性，但两者之间的相似之处有很多，所以我们也在这里对它们稍加说明。类型属性不象类和实例那样有一个属性字典（即`__dict__`），那么，我们如何才能知道内建类型都有哪些属性呢？Python语言为内建类型准备了两个特殊的属性，即`__methods__`和`__members__`，来查看其方法和/或数据属性。复数就是既有方法又有属性的内建类型的例子，所以我们这里就以它为例用`__methods__`和`__members__`来帮助我们找出它的属性：

```
>>> aComplex = (1+2j)      # create a complex number
>>> type(aComplex)         # display its type
<type 'complex'>
>>> aComplex.__members__   # reveal its data attributes
['imag', 'real']
>>> aComplex.__methods__   # reveal its methods
['conjugate']
```

知道一个复数都有哪些属性以后，我们来看看如何访问其数据属性和调用它的方法：

```
>>> aComplex.imag
2.0
>>> aComplex.real
1.0
>>> aComplex.conjugate()
(1-2j)
```

访问`__dict__`的尝试将会是失败的，因为内建类型没有这个属性，如下所示：

```
>>> aComplex.__dict__
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: __dict__
```

最后一个提示：特殊属性`__methods__`和`__members__`都只是Python语言本身提供的权宜之计。在外部模块或者第三方扩展模块中定义的新类型里就不见得实现有这两个方法，尽管这是我们大力推荐的做法。

13.6.5 实例属性和类属性的比较

类的数据属性我们已经在13.4.1节里介绍过了。现在再回顾一下：类的属性简单来说都是一些只与类相关联而不与某个特定的实例相关联的数据值，这和实例的属性是不一样的。这些值也叫做静态成员，因为它们的值即使在那个类经过多次实例化之后依然保持固定不变。不管在什么情况下，静态成员都是以不依赖于任何实例的方式保持它们自己的值的——除非直接地、明确地对它进行了修改。实例属性和类属性之间的比较与动态变量和静态变量之间的比较非常相似，如果读者从其他程序设计语言里弄懂了这些概念，就更容易理解这句话了。

类属性和实例属性的某些差异需要我们特别加以注意。最首要的是你可以通过类或者实例来访问类的某个属性，但前提是那个实例没有一个与它同名的属性。

1. 类属性的访问方法

类型的属性可以通过类或者实例进行访问。在下面的例子里，我们在创建类C时给它加上了

一个名为version的类属性，所以我们完全可以通过这个类对象C.version来访问它。而在创建实例c的时候，Python会向我们提供一个缺省的只读属性c.version，它就是那个类属性的别名。如下所示：

```
>>> class C:                # define class
...     version = 1.0        # static member
...
>>> c = C()                 # instantiation
>>> C.version                # access via class
1.0
>>> c.version                # access via instance
1.0
>>> C.version = C.version + .1 # update (only) via class
>>> C.version                # class access
1.1
>>> c.version                # instance access, which
1.1                          # also reflected change
```

请注意，通过一个实例属性来访问一个类属性严格说来只能以只读的方式进行（如果你试图对它进行修改，请参考下一小节的内容），因此我们只能通过那个类来修改那个属性的值，就像刚才对C.version进行加法运算那样。试图使用实例名来设置或者修改类属性是不允许的，那样会创建一个实例属性来。

2. 赋值操作创建局部实例属性

对一个局部属性的任何赋值操作都会导致创建和赋值一个实例属性，这和创建并赋值一个普通的Python变量的情况一样。即使存在一个同名的类属性，它也会在实例中被覆盖，如下所示：

```
>>> dir(C)
['__doc__', '__module__', 'version']
>>> dir(c)
[]
>>> c.version = 100         # attempt to update class attr
>>> c.version
100
>>> C.version                # nope, class attr unchanged
1.1
>>> dir(c)                   # confirm new instance attr created
['version']
```

上面的代码段创建了一个新的名为version的实例属性，覆盖了那个同名的类属性的引用线索。但那个类的属性并没有受到影响，依然存在于那个类的势力范围内，还可以做为一个类属性被访问，这也是我们在上面的例子里看到的情况。

为了核实增加了一个新的实例属性，我们在上述代码里赋值语句的执行前后分别调用了dir()内建函数。在赋值语句执行之前，实例c没有任何属性，而类C倒有三个属性（分别是__doc__、__modules__和version）。赋值语句执行之后再调用dir()内建函数就可以看到多出了一个新的实例属性version。

如果我们删除了这个新的引用线索会发生什么样的事情呢？我们来看看用del语句删除掉c.version会发生什么事情：

```
>>> del c.version          # delete instance attribute
>>> dir(c)
[]
>>> c.version              # can now access class attr again
1.1
```

现在再对这个类属性进行一次修改，但这次是做一个加法操作：

```
>>> c.version = c.version + 1.0
>>> c.version
2.1
>>> dir(c)
['version']
>>> C.version
1.1
```

好像还是不对劲。我们再创建一个实例属性，不要去动那个类属性。赋值语句右边的表达式取值为最初的那个类变量，给它加上1.0，然后赋值给新创建的实例属性。我们在下面给出的是一个同样功能的赋值语句，但可能更容易理解一些：

```
c.static = C.static + 1.0
```

3. 类属性的稳定性

静态变量就像它们的名字一样面对实例（及其属性）的来来往往岿然不动（完全独立于实例）。如果在对某个类属性进行修改后又创建了一个新的实例，这个修改就会在新实例里反映出来。如下所示：

```
>>> class C:
...     spam = 100          # class attribute
...
>>> c1 = C()               # create an instance
>>> c1.spam                 # access class attr thru inst.
100
>>> C.spam = C.spam + 100   # update class attribute
>>> C.spam                  # see change in attribute
200
>>> c1.spam                 # confirm change in attribute
200
>>> c2 = C()               # create another instance
>>> c2.spam                 # verify class attribute
200
>>> del c1                  # remove one instance
>>> C.spam = C.spam + 200   # update class attribute again
>>> c2.spam                 # verify that attribute changed
400
```

13.7 绑定和方法的调用

我们现在要对Python语言中的绑定概念做个细致的研究，这个概念只与方法的调用相关联。我们先来看看方法到底都是些什么东西。首先，一个方法就是做为一个类的组成部分而被定义出来的一个函数，这意味着方法是类的属性（而不是实例的属性）。第二，只有在对方法进行定义的那个类有了一个实例时才能对方法进行调用。当存在一个实例时，方法就会被认为是“绑定”了的；如果没有一个实例，方法就会被认为是“未绑定”的。第三，任何方法定义中的第

一个参数都是变量self，它代表的就是调用那个方法的实例对象本身。

编程提示：什么是“self”？

[CN]

用在类的实例方法中的变量self代表的是该方法绑定于其上的实例。在任何方法的调用过程中，它的第一个参数必须是那个方法所在的那个实例，而self就是被选出来代表那个实例的名字。在定义方法时必须把self放到有关代码部分里去（大家可能已经注意到这一点了），但在调用方法时却并不必使用实例（即self）。

如果你在自己的方法没有使用self，可以考虑把它替换为一个普通的函数，除非你有更好的理由不这样做。总之，因为你的代码不会用到实例对象，所以把它的概念和类“脱离”开来会使它更像是一个普通函数。

我们下面创建一个类C，在其中定义一个名为showSelf()的方法来显示我们新创建的实例对象的有关信息，如下所示：

```
>>> class C:
...     def showSelf(self):
...         self
...         type(self)
...         id(self)

>>> c = C()
>>> c.showSelf()
<__main__.C instance at 94abe0>
<type 'instance'>
9743328
```

现在来直接查看一下实例，看看信息是否一致——确实一致，如下所示：

```
>>> c
<__main__.C instance at 94abe0>
>>> type(c)
<type 'instance'>
>>> id(c)
9743328
```

在其他面向对象的程序设计语言里，self被取名为this。

13.7.1 调用绑定方法

不管方法绑定与否，它的代码都是一样的。两者唯一的区别在于是否有一个实例以便调用那个方法。虽然每一个方法的定义里都要求把self做为第一个参数，但从一个实例来调用它的时候却不必明确地传递它，解释器会自动做好这项工作。

下面是一个调用绑定方法的例子，我们在13.4.2节里已经见过它了：

```
>>> class MyClass:
...     def myNoActionMethod(self):
...         pass
...
>>>
>>> myInstance = MyClass()           # create instance
>>> myInstance.myNoActionMethod()    # invoke method
```

调用方法时要以点属性记号形式写出实例的名字和方法的名字，最后再加上函数操作符（即圆括号`()`）和参数。

在13.4.2节里，我们简单地提到如果用类的名字来调用方法是不能成功的；调用失败的原因是没有通过实例来调用方法。如果没有通过一个实例来调用方法（并把该实例做为`self`变量传递到方法去），解释器就会认为没有传递来那个必不可少的`self`参数。我们的错误在于调用了一个没有实例支持着的未绑定方法

13.7.2 调用未绑定方法

程序员通常会在两种情况下试图调用一个未绑定的方法。一是程序员试图编写静态的方法（但这在Python语言里是不支持的）；二是定义那个方法的类的某个实例不可用。我们先来说说怎样绕过Python语言缺乏静态方法的限制。

1. 迂回实现静态方法

一般说来，有两种场合需要用到静态方法。第一种情况是程序员需要保持全局或局部名字空间的“纯洁性”，不在相应的名字空间里添加其他的函数。第二种情况是那个只是一个不太重要的小函数，和程序员打算定义的类关系不太大；或者那只是一个帮助管理静态数据的函数。第一种情况争论的理由不太充分；但第二种情况——即静态成员管理函数——应该说还是有一定道理的，因为我们只是想按照函数的思路调用那个方法（即不依赖于实例）对静态数据进行修改而已。请看下面的例子：

```
>>> class C:
...     version = 1.0          # static data attribute
...     def updateVersion(self, newv):
...         C.version = newv  # update static data
... 
```

当然，如果没有一个实例就直接调用这个方法会引起`TypeError`例外，这我们在前面已经见过了：

```
>>> C.updateVersion(2.0)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
    C.updateVersion()
TypeError: unbound method must be called with class
instance 1st argument
```

有折中的办法吗？有，但可能不太让人高兴：放弃这样的尝试，把这个函数设计为一个能够访问类C（因此也就能够访问C的属性）的全局函数。下面就是我们的做法：

```
>>> def updateVersion(newv):
...     C.version = newv
...
>>> updateVersion(2.0)
>>> C.version
2.0
```

但像这样用全局函数来解决问题的做法也有一个问题，就是你会觉得`updateVersion()`不像是类的一个方法，而事实也确是如此，它的确不再是一个方法了。我们的工作改为调用C.

updateVersion()或者其他什么东西了。我要告诉大家的是，还有一些更绝的办法可以绕过这个限制，但做为程序员新手，最好避免这样做，请看Python的FAQ（常见问题答疑）中的论述：“如果你不了解自己为什么会想到要这样做，那是因为你头脑单纯，而且你可能永远也不会想到要这样做！这是一个很危险的技巧，只要能够不这样做，就尽量别这样做。”

2. 实例不可用

调用一个方法（比如说调用instance.method(x, y)方法吧）相当于让解释器执行函数method(instance, x, y)，请看下面的例子：

```
>>> class MyData:
...     def myMethod(self, arg):
...         print 'called myMethod with:', arg
...
>>>
>>> myInstance = MyData()
>>> myInstance.myMethod('grail')
called myMethod with: grail
```

调用myInstance.myMethod('grail')相当于调用myMethod(myInstance, 'grail')，也就是相当于函数签名myMethod(self, arg)。

但调用一个未绑定方法时情况就不太一样了。因为没有绑定的实例，对它的调用就不再是method(instance, x, y)而是method(x, y)，再看一遍出现的错误：

```
>>> MyData.myMethod(932)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
    MyData.myMethod(932)
TypeError: unbound method must be called with class
instance 1st argument
```

请注意TypeError例外给出的注释说明：“unbounded method must be called with class instance 1st argument”（这句话的意思是：未绑定方法，调用时必须把实例做为第一个参数）。这就是问题所在。传递给方法的第一个参数是一个整数，不是预料中的一个类的实例。函数签名不匹配，即在预期出现myMethod(self, arg)的地方出现的是myMethod(932)。那么，如果我们人为地把一个实例做为第一个参数传递过去（让函数签名匹配了）又会怎样呢？

```
>>> MyData.myMethod(myInstance, 932)
called myMethod with : 932
```

万岁！成功了。所以如果已经有了一个实例，就可以调用一个未绑定的方法，前提是你必须明确地给出这个实例，即采用method(myInstance, x, y)的形式调用它。

这个办法是很不错，可如果我们手边没有可用的实例又该怎么办呢？如果在上面的例子里，我们没有创建过myInstance又会发生怎样的情况呢？真要是这样，我们将无法调用myMethod()方法。会发生这样的情况吗？会发生，请看下面13.9节的内容。这种情况会引起从一个被推导出来的类去调用一个父类的方法。我们在13.1节“创建子类”那部分内容里已经见过这样的情况了。

13.8 构造

对类进行定义的目的是把它用在程序中做为一个模型，把这个对象嵌在你的代码里，让它

和其他数据类型以及程序执行的逻辑流交织在一起。类在操作代码里有两种用法。第一种用法是构造，就是把各种各样的类组织在一起，进一步构造出其他的类，增强功能性和代码的重复利用性。你也许会在一个更大的类的内部创建你自己的类的实例，让它具备其他属性和方法，增强了原来那个类对象的功能。第二种用法是推导，我们将在下一小节讨论。

让我们把本章开始时的那个电话号码本类改进一下。如果在我们的设计工作中又新创建了其他的姓名、地址等数据的类，当然希望能够把它们集成到AddrBookEntry类里面去而不是推翻原方案再重新设计。这样做可以节省大量的时间和精力，而代码也更完整统一。只要把同一代码段中的bug都除掉了，那所有用到这段代码的应用程序中相应的bug也就都除掉了。

一个这样的类可能包含Name（姓名）和Phone（电话号码）实例，也许还有其他一些诸如StreetAddress（门牌号）、Phone（住宅电话、单位电话、传真、呼机、手机等号码）、Email（住宅、单位等电子邮件地址），说不定还有几个Data（生日、婚礼、周年纪念日等日期）实例。下面的例子里就有我们刚才提到的几个类：

```
class NewAddrBookEntry:           # class definition
    'new address book entry class'
    def __init__(self, nm, ph):    # define constructor
        self.name = Name(nm)      # create Name instance
        self.phone = Phone(ph)    # create Phone instance
        print 'Created instance for:', self.name
```

这个NewAddrBookEntry类就是由它本身和另外几个类构造而成的。它使一个类和它参与构造出来的类之间建立了一种关系。比如说，现在的NewAddrBookEntry类里就有一个Name类的实例和一个Phone类的实例。

构造出来的复合对象具有综合性的功能，也比较符合现实生活中的情况：这些类每个都负责一摊，各自管理着自己的名字空间和操作行为。除了这种独立个体组合在一起的情况外，对象之间还可以有更紧密的关系，这就是推导。

13.9 子类的分离和推导

构造这种办法比较适合彼此区别明显的类组合出一个更大的类的情况，但如果你想要的是“同一个类，但有点小区别”，推导就是一个更合理的选择。

OOP的强大功能里包括这样一个能力：就是能够对一个现有的类进行功能扩展或者修改，但这些变化不会影响到系统中使用着这个现有的类的那些代码。OOD允许类的功能被它的“后代”类或“子类”“继承”。这些子类从“父”（也叫祖先、上级）类里推导出自己属性的核心部分。这种推导可以扩展许多代。一层推导所涉及的两个类（即类的树状谱系中上下两个直接联系着的类）是“父子”关系；从同一个父类里推导出来的类（即类的树状谱系中左右挨着的那些类）是“兄弟”关系。父类和更高层的那些类都被认为是“祖先”。

我们用前面内容中的例子来说明这些关系。假设需要我们创建不同种类的地址簿。这里说的可不是创建一个地址簿的多个实例——在这个例子里，所有的对象都有一些共性，彼此又有一些小的区别。比如说，BussinessAddressBook类可能更倾向于工作方面，里面主要记录着职务、单位电话号码和单位电子邮件地址等资料；PersonalAddressBook类则偏重于家庭方面，主要记

录着住址、亲戚关系、生日等资料。

我们并不打算从头开始设计这两个类，因为肯定会重复在设计AddressBook类时做过的大量工作。如果能够在保留AddressBook类中的所有功能和特点的前提下针对新类中数据项的特点增减一些定制功能不是更好吗？这就是类的推导概念的由来和目的。

创建子类

我们已经介绍过如何对一个基本类进行定义，它的通用语法是这样的：

```
class ClassName:
    'optional class documentation string'
    class_suite
```

子类的定义过程和它们的父类型差不多，只是要把它们继承的那些类列在它们名字的后面，如下所示：

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
    'optional class documentation string'
    class_suite
```

到目前为止，我们已经见过不少类和子类的例子了，请看下面这个简单的例子：

```
>>> class Parent:
...     # define parent class
...     def parentMethod(self):
...         print 'calling parent method'

>>> p = Parent()
>>> dir(Parent)
['_doc__', '__module__', 'parentMethod']
>>> p.parentMethod()
calling parent method

>>> class Child(Parent):
...     # define child class
...     def childMethod(self):
...         print 'calling child method'

>>> c = Child()
>>> dir(Child)
['_doc__', '__module__', 'childMethod']
>>> c.childMethod()
calling child method

>>> c.parentMethod()
calling parent method
```

13.10 继承性

继承指的是基类的属性被“延续”到一个推导出来的子类中。子类将继承它全体基类的所有属性，包括它们的各种数据属性和方法。

请看下面给出的这个例子。P是一个很简单的类，它没有属性。C是从P推导出来的一个类（因此它就是P的一个子类），也没属性：

```
>>> class P:
...     pass
>>> class C(P):
...     pass
```

```
>>>
>>> c = C()                # instantiate child
>>> c.__class__            # child "is a" parent
<class __main__.C at 8120c98>
>>> C.__bases__           # child's parent class(es)
(<class __main__.P at 811fc98>,)

```

因为P没有属性，所以C什么也没继承到。现在给P加上几个属性好让它更有用一些：

```
>>> class P:                # parent class
...     'P class'
...     def __init__(self):
...         print 'created an instance of', \
...             self.__class__.__name__
...
>>> class C(P):            # child class
...     pass

```

现在的P里有一个文档字符串（__doc__），还有了一个构造器，当我们对P进行实例化操作的时候就会执行那个构造器，请看交互方式下的执行情况：

```
>>> p = P()                # parent instance
created an instance of P
>>> p.__class__            # class that created us
<class __main__.P at 811f900>
>>> P.__bases__            # parent's parent class(es)
()
>>> P.__doc__              # parent's doc string
'P class'
>>> dir(P)                 # parent class attributes
['__doc__', '__init__', '__module__']

```

“created an instance”是直接从构造器__init__()中输出的。为了让读者了解类P的情况，我们另外提供了一些信息。因为P不是一个子类，所以它的__bases__属性是空的。现在对C进行实例化，请注意它是如何继承到__init__()（构造器）方法并执行它的：

```
>>> c = C()                # child instance
created an instance of C
>>> c.__class__            # class that created us
<class __main__.C at 812c1b0>
>>> C.__bases__            # child's parent class(es)
(<class __main__.P at 811f900>,)
>>> C.__doc__              # child's doc string
>>>
>>> dir(C)                 # child class attributes
['__doc__', '__module__']

```

C并没有定义自己的__init__()方法，但在创建类C的实例c时还是有相应的输出，这就是C从P继承到__init__()方法的结果。__bases__表列现在把P列为它的父类。

读者可能已经注意到有些特殊的数据属性没有继承下来，最引人注目的是__doc__。每个类都应该有它自己的文档字符串。继承特殊的类属性是没有道理的，因为它们的值一般都只与一个特定的类相关联。

13.10.1 类属性__bases__

在13.4.4节里我们已经简要地介绍过类属性__bases__，它是包含着（子）类的所有父类的一

个表列。这里所说的父类 (parent class) 不同于基类 (base class) 的概念, 后者包括了全体祖先类。如果一个类不是推导出来的, 它的 `__bases__` 属性就是一个空表列。我们来看看 `__bases__` 有什么用处, 如下所示:

```
>>> class A: pass                # define class A
...
>>> class B(A): pass             # subclass of A
...
>>> class C(B): pass             # subclass of B (and indirectly, A)
...
>>> class D(A,B): pass           # subclass of A and B
...
>>> C.__bases__
(<class __main__.B at 8120c90>,)
>>> D.__bases__
(<class __main__.A at 811fc90>, <class __main__.B at 8120c90>)
```

在上面的例子里, 虽然类C是从类A (通过B) 和类B中推导出来的一个子类, 但C的父类是B, 在它的定义部分明确地指出了这一点, 所以在C. `__bases__` 里只出现了B。D则同时继承了两个类, 即A和B。(多重继承在13.10.4节讨论。)

13.10.2 通过继承覆盖掉方法

我们在P里面创建一个函数, 然后在它的子类里覆盖掉这个函数:

```
>>> class P:
...     def foo(self):
...         print 'Hi, I am P-foo()'
...
>>> p = P()
>>> p.foo()
Hi, I am P-foo()
```

现在从父类P开始创建子类C:

```
>> class C(P):
...     def foo(self):
...         print 'Hi, I am C-foo()'
...
>>> c = C()
>>> c.foo()
Hi, I am C-foo()
```

虽然C继承了P的foo()方法, 但因为C又定义了自己的foo()方法, 所以继承的foo()方法被覆盖掉了。对方法进行覆盖处理的原因是: 在一般情况下, 子类自己应该有一些特殊的或者与基类不同的功能才有存在的必要。那么, 读者会问了: “我还能调用被覆盖掉的基类中的方法吗?”

答案是可以, 但此时你调用的是基类中一个未绑定的方法, 所以必须明确地给出子类的实例作为参数, 如下所示:

```
>>> P.foo(c)
Hi, I am P-foo()
```

请注意，在上面的例子里，我们实际已经有一个名为P的实例了，但在操作中并没有用到它。调用P中的方法并不需要P的一个实例，因为我们还已经有P的子类的一个实例c，用它就可以了。

编程提示：覆盖__init__()不调用基类的__init__属性

[CN]

从一个带有构造器__init__()的类开始推导子类时，如果不覆盖掉原来的__init__()，它就会被子类继承并自动执行。如果在子类里覆盖了__init__()，那么基类里面的__init__()方法就不会在对子类进行实例化操作时自动执行了。

```
>>> class P:
...     def __init__(self):
...         print "calling P's constructor"
...
>>> class C(P):
...     def __init__(self):
...         print "calling C's constructor"
...
>>> c = C()
created an instance of C
```

如果你真的想调用基类里面的__init__()方法，就必须按我们刚才介绍的方法用一个子类的实例明确地调用基类里面的这个(未绑定)方法。我们把类C的定义做如下所示的修改：

```
>>> class C(P):
...     def __init__(self):
...         P.__init__(self)
...         print "calling C's constructor"
...
>>> c = C()
calling C's constructor
```

在上面的例子里，我们把基类里面的__init__()方法放在我们自己的__init__()方法的最前面。在实践中我们经常会这样做，即先对基类做初始化，然后再进行子类内部的局部初始化。这种做法很合理，因为在子类的构造器的代码运行之前，应该让被继承对象都得到正确的初始化，因为构造器的主要用途是对属性进行初始化，其中当然也包括将要被继承的属性。

C++允许在子类的构造器定义中调用基类的构造器，具体做法是在声明标题尾部加上一个冒号，然后再跟上对任何基类的构造器的调用。Java程序员就没有选择的余地了——基类的构造器必须是子类构造器中第一个被调用的。Python语言中用基类的名字来调用一个基类方法的做法正好对应着Java语言中使用super关键字时的情况。

13.10.3 对标准类型进行推导

Python语言中的类型不是类(class)，这就意味着不能从它们开始推导出子类来。好在还有办法，我们可以通过多种缺省的特殊属性方法来仿真这些标准类型（请参考4.2节中的编程提示和6.14.2节、13.12节的内容）。

13.10.4 多重继承

Python语言允许从多个基类开始创建和定义子类，这就是人们常说的“多重继承”。Python语言对多重继承有一些规定，它采用深度优先搜索算法收集将要赋值给子类的属性。这个算法不像Python语言中的其他算法那样用后找到的名字覆盖掉先找到的名字，多重继承把找到的第一个名字（如果有重名的话）做为将被继承的属性的名字。

我们用下面的例子来说明深度优先算法的执行情况。例子里有两个父类、两个子类和一个“孙”类：

```
class P1:                                # parent class 1
    def foo(self):
        print 'called P1-foo()'

class P2:                                # parent class 2
    def foo(self):
        print 'called P2-foo()'
    def bar(self):
        print 'called P2-bar()'

class C1(P1,P2):                          # child 1 der. from P1, P2
    pass

class C2(P1,P2):                          # child 2 der. from P1, P2
    def foo(self):
        print 'called C2-foo()'
    def bar(self):
        print 'called C2-bar()'

class GC(C1,C2):                          # define grandchild class
    pass                                  # derived from C1 and C2
```

在交互式解释器里执行这段代码，就可以验证它只使用了第一个遇到的属性：

```
>> gc = GC()
>>> gc.foo()      # GC ⇒ C1 ⇒ P1
called P1-foo()
>>> gc.bar()      # GC ⇒ C1 ⇒ P1 ⇒ P2
called P2-bar()
```

再次提醒大家，只要在调用方法时用上完全授权名字并用一个合法的实例做参数，你就可以调用任何一个特定的方法。

```
>>> C2.foo(gc)
called C2-foo()
```

13.11 类、实例和其他对象的内建函数

13.11.1 issubclass()

布尔函数issubclass()的作用是确定某个类是否是另外一个类的子类或后代，它的语法如下所

示:

```
issubclass(sub, sup)
```

如果给出来的类sub确实是sup的一个子类, issubclass()就返回1。这个函数允许出现一个“非正常”子类, 即一个类是它自己的一个子类; 所以在sub就是sup和sub是sup的子类这两种情况下这个函数返回的都是1。(“正常”子类严格定义为从某个类开始推导出来的子类。)

如果让我们自己实现issubclass()内建函数, 它应该是如下所示的样子:

```
def my_issubclass(sub, sup):
    if sub is sup or sup in sub.__bases__:
        return 1
    for cls in sub.__bases__:
        if my_issubclass(cls, sup):
            return 1
    else:
        return 0
    return 0
```

我们先检查它们是否是同一个类。因为非正常子类也在许可范围之内, 所以两个类完全相同也会返回1。如果sup是sub的父类, 返回1; 这是通过检查sub的__bases__属性得到结果的。

如果这两个检查都没有成功, 就需要上溯子类的谱系去查看sup是否是sub的一个祖先。先检查sub的父类, 看它们是否是sup的子类; 如果不是, 就对“祖父”类进行检查; 依次向上看看祖先里是否有sup的子类。这是一个深度优先的递归检查, 所有祖先都会检查到。如果最终的回答是否定的, 就返回0值表示sub不是sup的后代。

13.11.2 isinstance()

[1.5]

isinstance()是一个布尔函数, 它的作用是确定某个对象是否是一个给定的类的实例, 它的语法如下所示:

```
isinstance(obj1, obj2)
```

如果obj1是类obj2的一个实例, 或者是obj2的某个子类的一个实例, isinstance()将返回真值1。

请看下面的例子:

```
>>> class C1: pass
...
>>> class C2: pass
...
>>> c1 = C1()
>>> c2 = C2()
>>> isinstance(c1, C1)
1
>>> isinstance(c2, C1)
0
>>> isinstance(c1, C2)
0
>>> isinstance(c2, C2)
1
>>> isinstance(C2, c2)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
```

```
isinstance(C2, c2)
TypeError: second argument must be a class
```

请注意，第二个参数必须是一个类，否则就会引起一个TypeError错误；这第二个参数还可以是一个type（类型）对象，因为以后完全可以使用isinstance()来查看对象obj1是否是obj2指定的类型，如下所示：

```
>>> isinstance(4, type(4))
1
>>> isinstance(4, type(''))
0
>>> isinstance('4', type(''))
1
```

如果打算自己实现isinstance()内建函数，它应该是下面这个样子：

```
def my_isinstance(obj1, obj2):
    if obj2 is type(type(0)):      # check if obj2 is type obj
        return type(obj1) is obj2
    if obj1.__class__ is obj2:     # check if obj1 inst of obj2
        return 1
    return my_issubclass(obj1.__class__, obj2)
```

isinstance()看起来要比issubclass()简单一些，但注意定义里用到了issubclass()函数。如果没有它，我们就不得不自己编写issubclass()，因此isinstance()实际要比issubclass()要长一些。

我们编写的这个isinstance()函数的工作情况是这样的：先检查obj2是否是一个类型对象（type对象）以确定我们是不是在与对象和类型打交道。如果obj2是类型对象，检验这二者的关系并返回结果；如果不是，说明摆在我们面前的是类和实例，所以开始检查实例obj1是否是类obj2的一个实例。如果是，这个函数也就该结束了。如果不是，就需要递归检查实例化出obj1实例的那个类是否是obj2的一个后代。是返回1；否返回0。

缺失功能的替代实现

[1.5]

前面介绍的这两个is*()族的内建函数（即issubclass()和isinstance()）都是从1.5版本开始新增到Python语言中的。在1.5版本之前，这两个函数都必须由用户自己来实现——就像我们刚才做过的那样，同时还必须给这些缺失功能安排好替代性定义。因为对Python的升级不可能包括所有新的函数，所以极有可能因为某些不可控制的原因使读者现在使用的还是不支持新函数的老Python版本。解决这类问题的办法就是自行编写出能够实现新函数功能的函数并把它们集成到自己的代码里去，这可以让你系统的行为至少看上去像是一个比较新版本的解释器。

下面是我们给出的一个解决方案示例，它试图导入__builtin__模块（我们想从该模块里加载上issubclass()和isinstance()函数），但如果不是这样，就把前两节内容里我们自己编写出来的代码定义为相应的函数，如下所示：

```
if '__builtin__' not in dir() and __import__ in dir():
    __builtin__ = __import__('__builtin__')

if 'issubclass' not in dir(__builtin__):
    issubclass = my_issubclass
```

```
if 'isinstance' not in dir(__builtins__):
    isinstance = my_isinstance
```

我们用前一小节里定义的类和实例来检验一下这些函数。注意P*代表的都是父类，C*代表的都是子类，GC代表的是一个“孙”类

```
>>> for eachCls in (P1, P2, C1, C2, GC):
...     print "is GC subclass of", eachCls.__name__, '?', \
...           isinstance(GC, eachCls)
...     print "is 'gc' an instance of", eachCls, '?', \
...           isinstance(gc, eachCls)

is GC subclass of P1 ? 1
is 'gc' an instance of __main__.P1 ? 1
is GC subclass of P2 ? 1
is 'gc' an instance of __main__.P2 ? 1
is GC subclass of C1 ? 1
is 'gc' an instance of __main__.C1 ? 1
is GC subclass of C2 ? 1
is 'gc' an instance of __main__.C2 ? 1
is GC subclass of GC ? 1
is 'gc' an instance of __main__.GC ? 1
```

13.11.3 hasattr()、getattr()、setattr()、delattr()

*attr()族的函数可以对各种对象进行操作，不仅仅限于类和实例。但因为它们经常用来对类和实例进行处理，所以把它们放在这里进行介绍。

hasattr(obj1, attr)是一个布尔函数，它的唯一用处就是检查一个对象是否具有某个特定的属性；用户程序代码在实际访问某个属性之前最好先用这个函数验证一下。getattr()和setattr()函数分别用来检索和设置对象的属性的值。如果准备读取的对象不具有函数中要求的属性，getattr()会引发一个AttributeError例外。setattr()会给对象添加一个属性或者替换掉一个现有的属性。delattr()从一个对象里去掉指定的那个属性。

下面是*attr()族各内建函数的使用示例：

```
>>> class myClass:
...     def __init__(self):
...         self.foo = 100
...
>>> myInst = myClass()
>>> dir(myInst)
['foo']
>>> hasattr(myInst, 'foo')
1
>>> getattr(myInst, 'foo')
100
>>> hasattr(myInst, 'bar')
0
>>> setattr(myInst, 'bar', 'my attr')
>>> dir(myInst)
['bar', 'foo']
>>> getattr(myInst, 'bar')
'my attr'
>>> delattr(myInst, 'foo')
>>> dir(myInst)
```



```
['bar']
>>> hasattr(myInst, 'foo')
0
```

13.11.4 dir()

我们最早是在练习2-12、练习2-13和练习4-7里接触到dir()的。在那些练习里，我们用dir()来获取关于一个方法的全部属性的资料。现在我们知道其实dir()是可以对任何具有属性的其他对象进行操作——这些对象里面包括有类、类的实例、文件、列表、复数数字，等等。只要一个对象具有__dict__属性字典和/或__members__和__methodds__列表，dir()函数就能够工作。内建类型不具备__dict__属性字典，所以dir()在对它们进行操作时就只能依靠__members__和__methods__列表了。请看下面的例子：

```
>>> dir(3+3j)                # complex number attributes
['conjugate', 'imag', 'real']
>>>
>>> (3+3j).__dict__
Traceback (innermost last):
File "<stdin>", line 1, in ?
AttributeError: __dict__
>>>
>>> (3+3j).__members__
['imag', 'real']
>>>
>>> (3+3j).__methods__
['conjugate']
>>>
>>> f = open('/etc/motd')
>>> dir(f)                   # file object attributes
['close', 'closed', 'fileno', 'flush', 'isatty', 'mode',
'name', 'read', 'readinto', 'readline', 'readlines',
'seek', 'softspace', 'tell', 'truncate', 'write',
'writelines']
>>> f.__dict__
Traceback (innermost last):
File "<stdin>", line 1, in ?
AttributeError: __dict__
>>> f.__members__
['closed', 'mode', 'name', 'softspace']
>>> f.__methods__
['close', 'fileno', 'flush', 'isatty', 'read',
'readinto', 'readline', 'readlines', 'seek', 'tell',
'truncate', 'write', 'writelines']
>>> f.close()
```

13.11.5 vars()

vars()内建函数的功能与dir()函数的很相似，只是前者要求做为其参数的对象必须有一个__dict__属性。vars()返回的是由指定对象的属性（用做键字）和属性值（用做键值）构成的一个字典，这个字典的根据就是该对象的__dict__属性。如果给出的对象不具有这个对象，就会引发一个TypeError例外。如果没有在vars()函数里给出任何对象做为参数，它就会把属性（用做键

字)和局部名字空间中的对应值构成的属性字典显示出来,也就是说,此时它相当于locals()函数的作用。下面是调用vars()对一个类实例进行操作的例子:

```
>>> class C:
...     pass

>>> c=C()
>>> c.foo = 100
>>> c.bar = 'Python'
>>> c.__dict__
{'foo': 100, 'bar': 'Python'}
>>> vars(c)
{'foo': 100, 'bar': 'Python'}
```

表13-3里总结了与类和类的实例有关的内建函数。

表13-3 类、实例、和其他对象的内建函数

内建函数	说 明
issubclass(sub, sup)	如果类sub是类sup的一个子类, 返回1; 否则返回0
isinstance(obj1, obj2)	如果实例obj1是类obj2的一个实例, 或者是类obj2的某个子类的一个实例返回1; 如果obj1是obj2类型的, 也返回1
hasattr(obj, attr)	如果obj有attr属性(以字符串方式给出), 返回1
getattr(obj, attr)	查找检索obj的属性attr; 和obj.attr的返回情况是一模一样的。如果attr不是obj的属性, 就会引发一个AttributeError例外
setattr(obj, attr, val)	把obj的属性attr设置为值val, 会覆盖掉现有属性的当前值; 如果没有当前属性, 将创建它; 相当于“obj.attr = val”
delattr(obj, attr)	从obj里删除属性attr(以字符串方式给出); 相当于“del obj.attr”的效果
dir(obj = None)	把obj的属性放在一个列表里返回; 如果没有给出obj, dir()就显示局部名字空间的属性, 即相当于locals().keys()
vars(obj = None)	返回由obj的属性和值构成的字典; 如果没有给出obj, vars()就显示局部名字空间的字典(属性和值), 即相当于locals()

13.12 类型和类/实例的比较

Python和Java等语言不同, 它的标准类型不是类, 它的变量也不是所谓类的实例。明确地说, Python语言中的标准类型和变量只是不允许直接进行子类推导的初级类型而已。(相对地, 我们可以把类看做是允许进行子类推导的特定的初级内建类型。)

Python所支持的内建类型的个数是固定的, 因此即使是不同种类的实例也都要归入同一个类型里去(“instance”), 而类也都要归入单一的“class”类型里去。(本章前面有一个编程提示里专门回答了为什么要把各种不同的实例归入是同一个类型。)

想对标准类型进行子类推导是不可能的, 虽然可能经常有这方面的需求。Python为此准备了一些非常精致的解决方案。办法之一是新创建一个具备某个标准类型操作行为的类(class)。这个方案的适用性很高, 用户对这个新创建的类型有完全的控制。另一个解决办法是把标准类型“打包”到一个类里, 利用现有的类型达到目的。(我们将在13.15节里讨论打包方面的问题。)。在这里, “打包”的意思是把那个名字类型用做那个类的数据对象, 再增加一些对它进行访问

和操作的方法，而这些方法的功用正好体现了原标准类型在操作方面的特性。在为一个应用程序设计和开发定制的数据类型时，这也是一个很完美的机制，而这就是我们后续各小节内容的焦点。

13.13 用特殊方法对类进行定制

在本章前面几小节里我们已经介绍过方法的两个重要方面：一是方法在能被调用之前必须绑定（到与之对应的类的一个实例上去）；二是有两个特殊的方法分别提供了构造器和拆除器的功能，这两个方法的名字是：__init__()和__del__()。

事实上，__init__()和__del__()只是可以由用户负责实现的那些特殊方法中的两种而已。在这些特殊方法中，有的把什么都不做预先定义为自己缺省的操作行为，还有的需要由用户根据其需要加以实现。这些特殊方法大大增强了类在Python语言中的地位，扩展了类的功能。具体来说，它们允许：

- 仿真标准类型
- 重载操作符

特殊方法使类能够通过重载标准操作符如“+”、“*”、甚至还有切片和映射操作符“[]”等来模仿标准类型的操作。和大多数特殊的保留标识符一样，这些方法的名字都是以双下划线(__)开始和结尾的。表13-4列出了所有的特殊方法及其介绍说明。

表13-4 对类进行定制时会用到的特殊方法

特殊方法	说 明
核心方法	
C.__init__(self[, arg1, ...])	构造器（可带任意个可选的参数）
C.__del__(self)	拆除器
C.__repr__(self)	可求值的字符串表示形式；相当于repr()内建函数和“”操作符
C.__str__(self)	可打印的字符串表示形式；相当于str()内建函数和print语句
C.__cmp__(self, obj)	对象的比较操作；相当于cmp()内建函数
C.__call__(self, *args)	定义可调实例
C.__nonzero__(self)	定义对象的假(false)值情况
C.__len__(self)	“长度”（适用于类）；相当于len()内建函数
属性	
C.__getattr__(self, attr)	获取属性；相当于getattr()内建函数
C.__setattr__(self, attr, val)	设置属性；相当于setattr()内建函数
C.__delattr__(self, attr)	删除属性；相当于del语句
对类进行定制/对类型进行仿真	
数值类型：二元操作符	
C.__add__(self, obj)	加法：+ 操作符
C.__sub__(self, obj)	减法：- 操作符
C.__mul__(self, obj)	乘法：* 操作符
C.__div__(self, obj)	除法：/ 操作符
C.__mod__(self, obj)	取除法余数：%操作符
C.__divmod__(self, obj)	除法和整除取余、作用相当于内建的divmod()函数
C.__pow__(self, obj[, mod])	幂运算：作用相当于内建的pow()函数；** 操作符

(续)

特殊方法	说 明
<code>C.__lshift__(self, obj)</code>	左移位: << 操作符
<code>C.__rshift__(self, obj)</code>	右移位: >> 操作符
<code>C.__and__(self, obj)</code>	逐位AND与操作; & 操作符
<code>C.__or__(self, obj)</code>	逐位OR或操作; 操作符
<code>C.__xor__(self, obj)</code>	逐位XOR异或操作; ^ 操作符
数值类型: 单元操作符	
<code>C.__neg__(self)</code>	单元求反操作
<code>C.__pos__(self)</code>	单元无变化操作
<code>C.__abs__(self)</code>	绝对值; 相当于内建的abs()函数
<code>C.__invert__(self)</code>	逐位求反; 相当于~操作符
数值类型: 数值转换	
<code>C.__complex__(self, com)</code>	转换为复数; 相当于内建的complex()函数
<code>C.__int__(self)</code>	转换为整数; 相当于内建的int()函数
<code>C.__long__(self)</code>	转换为长整数; 相当于内建的long()函数
<code>C.__float__(self)</code>	转换为实数; 相当于内建的float()函数
数值类型: 数制表示形式(字符串)	
<code>C.__oct__(self)</code>	八进制表示形式; 相当于内建的oct()函数
<code>C.__hex__(self)</code>	十六进制表示形式; 相当于内建的hex()函数
数值类型: 数值协调	
<code>C.__coerce__(self, num)</code>	协调为同一种数值类型, 相当于内建的coerce()函数
序列类型	
<code>C.__len__(self)</code>	序列中的数据项个数
<code>C.__getitem__(self, ind)</code>	截取序列中的一个元素
<code>C.__setitem__(self, ind, val)</code>	给序列中的一个元素赋值
<code>C.__delitem__(self, ind)</code>	删除序列中的一个元素
<code>C.__getslice__(self, ind1, ind2)</code>	截取序列切片
<code>C.__setslice__(self, i1, i2, val)</code>	给序列切片赋值
<code>C.__delslice__(self, ind1, ind2)</code>	删除序列切片
<code>C.__contains__(self, val)^b</code>	检查val是否是序列的一个成员; 相当于in关键字
<code>C.__*add__(self, obj)</code>	合并; 相当于+操作符
<code>C.__*mul__(self, obj)</code>	重复; 相当于*操作符
映射类型	
<code>C.__len__(self)</code>	映射关系中的数据项个数
<code>C.__hash__(self)</code>	哈希函数的计算结果值
<code>C.__getitem__(self, key)</code>	给出指定键字的对应键值
<code>C.__setitem__(self, key, val)</code>	对指定键字的对应键值进行赋值
<code>C.__delitem__(self, key)</code>	删除指定键字的对应键值

说明a: “*” 分别代表无 (self OP obj)、“r” (obj OP self) 或者运算加赋值操作的 “i” (新出现于Python 2.0版本)。

比如: `__add__`、`__radd__` 或 `__iadd__`。

说明b: 新出现于Python 1.6版本。

“核心方法”组中的特殊方法是特殊方法的基本构成, 不需要仿真任何特定类型就可以实现它们。“属性”组中的特殊方法可以帮助大家管好用好实例的属性。“数值类型”组中的特殊方

法可以用来仿真各种数值运算，包括标准（单元和二元）操作符、数值转换、数制转换、数值协调等。还有几个仿真序列和映射类型的特殊方法。自行编写实现某个特殊方法将重载与之对应的操作符，使它们能够对类（class）类型的实例进行操作。

表中列出的二元数值操作符名字里都有一个起通配符作用的星号，这表示该方法有几个稍有区别的版本。星号可以表示没有额外字符，也可以是一个字母“r”表示按右操作规则进行操作。在没有字母“r”的时候，操作是按照“self OP obj”（按左操作）的方式进行运算的；而加上“r”表示将按照“obj OP self”的方式进行运算。举例来说，__add__(self, obj)对应的是运算“self + obj”；而__radd__(self, obj)对应的是“obj + self”。 [2.0]

在Python 2.0版本开始出现的增量赋值语句引入了一个“运算并赋值”操作。出现在星号位置上的“i”表示按左计算规则完成运算后再执行一个赋值操作，也就是相当于“self = self OP obj”语句的效果。举例来说，__iadd__(self, obj)对应着“self = self + obj”。

13.13.1 对类进行简单定制的例子

我们的第一个例子就从创建一个由两个有序数字(x, y)组成的类开始好了。我们把这个数据在我们的类里面用一个两元素的表列来表示。下面的代码段对类进行了定义，并且还定义了一个构造器，它把给定的两个数值当做oPair类的数据属性保存起来，如下所示：

```
class oPair:
    def __init__(self, obj1, obj2):
        self.data = (obj1, obj2)
```

ordered pair
constructor
assign attribute

用这个类实例化一个对象出来：

```
>>> myPair = oPair(6, -4)
>>> myPair
<oPair instance at 92bb50>
>>> print myPair
<oPair instance at 92bb50>
```

create instance
calls repr()
calls str()

从上面可以看出，无论是print（使用了str()）还是那个对象本身的字符串表示形式（使用了repr()）都不能给我们以更多的信息。所以实现__str__()方法或者__repr__()方法应该是个好主意；要是两个函数都实现了就更好了，这样我们就可以“看到”我们的对象是什么样子的了。换句话说，大家在输出显示一个对象的时候，实际上是想看到一些有意义的东西而不是Python语言的对象字符串（<object type at id>）。我们想看到的是一个里面有两个数据值的有序数字对（表列）。说干就干，我们来编写显示有序数字对的__str__()方法，如下所示：

```
def __str__(self):
    return str(self.data)
```

str() string representation
convert tuple to string

__repr__ = __str__

repr() string representation

我们还想用这段代码实现__repr__()，与其一个字一个字的照抄代码，不如给这段代码来个再利用，给__str__()创建一个别名不就行了。现在，我们的输出已经大有改观了：

```
>>> myPair = oPair(-5, 9)
>>> myPair
(-5, 9)
>>> print myPair
(-5, 9)
```

create instance
repr() calls __repr__()
str() calls __str__()

下一步该做什么了？假设我们想给它加上一些运算方面的功能。举例来说，我们可以定义两个 oPair 对象 (x1, y1) 和 (x2, y2) 的加法操作是对两个元素分别求和。这样，两个 oPair 对象的“和”就被定义为一个新的对象，它的值是 (x1 + x2, y1 + y2)。我们这样来实现 __add__() 特殊方法：先计算出两组加法的结果，再调用这个类的构造器返回一个新对象。最后，我们再给 __add__() 加上一个别名 __radd__()，因为这里的顺序并不重要——换句话说，这是因为数值加法中两个操作数的次序是可以前后互换的。__add__() 和 __radd__() 的定义如下所示：

```
def __add__(self, other):      # add two oPair objects
    return self.__class__(self.data[0] + other.data[0],
                           self.data[1] + other.data[1])
```

创建新对象时和平常一样调用了那个类。唯一的差别是：在类的内部一般不直接调用这个类的名字。常见的做法是使用 self 的 __class__ 属性，它的值就是实例化出 self 并调用它的那个类。因为 self.__class__ 和 oPair 是一样的，所以调用 self.__class__() 和调用 oPair() 也是一样的。

现在用我们刚才重载的操作符来做加法运算。重新加载我们改进了的模块，创建两个 oPair 对象并把它们加起来，得到的是如下所示的和：

```
>>> pair1 = oPair(6, -4)
>>> pair2 = oPair(-5, 9)
>>> pair1 + pair2
(1, 5)
```

如果没有实现相应的特殊方法就使用与之对应的操作符进行运算会引发一个 TypeError 例外，如下所示：

```
>>> pair1 * pair2
Traceback (innermost last):
  File "<stdin>", line 1, in ?
    pair1 * pair2
TypeError: __mul__ nor __rmul__ defined for these operands
```

很明显，如果我们没有实现 __add__() 和 __radd__()，在进行加法运算时就会看到类似的错误信息了。最后一个例子对已经存在的数据进行处理。假设我们的系统里已经有一些由两个元素构成的表列了，在目前的情况下，如果我们想把它们创建为 oPair 对象，就必须先把它们分拆成单个的元素，再用两个单个的元素实例化出一个 oPair 对象来，请看：

```
aTuple = (-3, -1)
pair3 = oPair(aTuple[0], aTuple[1])
```

但与其像上面这样先分拆原来的表列才能创建我们的对象，不如直接把那个表列馈入我们的构造器，在那里对它进行处理不是更好吗？好是好，可不能像其他面向对象的程序设计语言那样对构造器进行重载。Python 语言不支持可调用对象的重载，因此解决这个问题的唯一办法只能是利用 type() 内建函数做一些人工探索。

在下面对构造器 __init__() 的改进中，我们先探索到手的是不是一个表列。如果是，就直接把它赋值给 data 属性；否则，这就是一个“正常”的实例化过程，即预期传递来两个数字。

```
def __init__(self, obj1, obj2=None): # constructor
    if type(obj1) == type(()):      # tuple type
        self.data = obj1
    else:
```

```

if obj2 == None:                # part of values
    raise TypeError, \
        'oPair() requires tuple or numeric pair'
self.data = (obj1, obj2)

```

请注意，上面的例子给参数obj2定义一个缺省值None。这样在参数是一个表列的时候，只要传递来一个对象就可以了。我们不允许发生的情况是在没有第二个参数的情况下去创建一个oPair对象，所以我们在else从句里又对obj2是否是None进行了检查。现在再进行调用就可以得到比较有意义的结果了：

```

aTuple = (-3, -1)
pair3 = oPair(aTuple)
>>> pair3
(-3, -1)
>>> pair3 + pair1
(3, -5)

```

我们希望读者对操作符的重载现在有了更进一步的认识，知道为什么要这样做，以及怎样实现特殊方法才能达成目的。如果读者有兴趣看一个复杂些的定制示例，请继续学习下面的选读章节。

13.13.2 *对类进行较复杂定制的例子

我们来创建一个新的类，它是个由一个数值“n”和一个字符串“s”组成的有序数据项，我们用整数做为那个数值的类型。一般情况下，一个有序数据项“正确”的记号形式应该是(n, s)，但为了与之有所区别，我们决定用[n : s]做为我们数据项的表示形式。不管记号方式如何选择，这两个数据元素在我们的模型里是不可分割的。我们的目的是创建一个新的类，名字就叫做NumStr，它应该具备以下特性：

1. 初始化

这个类在初始化时必须使用数值和字符串两个参数；如果其中一个（或者两个）缺失了，就把整数值0和空字符串用做缺省值，即 $n = 0$ 和 $s = ''$ 。

2. 加操作

我们把加法操作符的功能定义为把两个数值加到一起，把两个字符串合并到一起；注意字符串必须按照正确的顺序进行合并。举例来说，已知NumStr1=[n1 : s1]和NumStr2=[n2 : s2]，那么NumStr1 + NumStr2=[n1 + n2 : s1 + s2]，即“+”代表数值的加法操作和字符串的合并操作。

3. 乘操作

类似地，我们把乘法操作符的功能定义为把两个数值乘到一起，把两个字符串重复或者合并到一起，即NumStr1 * NumStr2=[n1 * n2 : s1 * s2]。

4. 假值

我们把数值为0且字符串为空的情况，即当NumStr1=[0 : '']时的情况，定义为这个数据项的假值。

5. 比较操作

两个NumStr对象进行比较会出现九种不同的组合情况（如 $n1 > n2$ and $s1 < s2$ 、 $n1 == n2$ and

$s1 > s2$ 等)。我们把数值和字符串的比较操作分别定义为按数值大小和字母表顺序的先后进行比较的正常情况。而`cmp(obj1, obj2)`的普通比较情况是：当`obj1 < obj2`时返回一个小于零的整数；当`obj1 > obj2`时返回一个大于零的整数；如果两个对象具有同样的值就返回一个整数零。

我们对这个类的解决方案是把数值和字符串分别进行比较后的结果取值加在一起做为整个比较操作的返回值。有一件事情很有意思，就是`cmp()`好像不愿意返回不是-1、0、1的值；也就是说，即使数值和字符串分别进行比较后的结果取值加在一起的和是-2或2，`cmp()`分别的仍将是-1和1。如果两个对象中对应的数值和字符串都相同、或者两个比较的结果正好相反（即出现 $(n1 < n2)$ and $(s1 > s2)$ 或者反过来的情况时）最终的返回值将是0。

根据以上各项规定，我们把编写好的`numstr.py`程序的代码列在下面（见程序示例13-2）。

程序示例13-2 用类仿真类型 (`numstr.py`)

```

1 #!/usr/bin/env python
2
3 class NumStr:
4
5     def __init__(self, num=0, string=''): # constr.
6         self.__num = num
7         self.__string = string
8
9     def __str__(self):                    # define for str()
10        return '{d} : {s}'.format(d=self.__num, s=self.__string)
11
12    __repr__ = __str__
13
14    def __add__(self, other):              # define for s+o
15        if isinstance(other, NumStr):
16            return self.__class__(self.__num + \
17                                   other.__num, \
18                                   self.__string + other.__string)
19        else:
20            raise TypeError, \
21            'illegal argument type for built-in operation'
22
23    def __radd__(self, other):             # define for o+s
24        if isinstance(other, NumStr):
25            return self.__class__(other.num + \
26                                   self.num, other.str + self.str)
27        else:
28            raise TypeError, \
29            'illegal argument type for built-in operation'
30
31    def __mul__(self, num):                # define for o*n
32        if type(num) == type(0):
33            return self.__class__(self.__num * num, \
34                                   self.__string * num)
35        else:
36            raise TypeError, \
37            'illegal argument type for built-in operation'
38
39    def __nonzero__(self):                 # reveal tautology
40        return self.__num or len(self.__string)
41
42    def __norm_cval(self, cmpres): # normalize cmp()
43        return cmp(cmpres, 0)
44
45    def __cmp__(self, other):              # define for cmp()
46        nres = self.__norm_cval(cmp(self.__num, \
47                                   other.__num))
48        sres = self.__norm_cval(cmp(self.__string, \
49                                   other.__string))
50
51        if not (nres or sres): return 0 # both 0
52        sum = nres + sres

```



```
53     if not sum: return None # one <,one>
54     return sum
```

下面是一次示范性执行的具体情况，请注意这个类是如何工作的：

```
>>> a = NumStr(3, 'foo')
>>> b = NumStr(3, 'goo')
>>> c = NumStr(2, 'foo')
>>> d = NumStr()
>>> e = NumStr(string='boo')
>>> f = NumStr(1)
>>> a
[3 :: 'foo']
>>> b
[3 :: 'goo']
>>> c
[2 :: 'foo']
>>> d
[0 :: '']
>>> e
[0 :: 'boo']
>>> f
[1 :: '']
>>> a < b
1
>>> b < c
0
>>> a == a
1
>>> b * 2
[6 :: 'googoo']
>>> a * 3
[9 :: 'foofoofoo']
>>> e + b
[3 :: 'boogoo']
>>> if d: 'not false'
...
>>> if e: 'not false'
...
'not false'
>>> cmp(a,b)
-1
>>> cmp(a,c)
1
>>> cmp(a,a)
0
```

6. 逐行解释

第3-7行

构造器 `__init__()` 函数对我们的实例进行初始化设置时要使用传递到这个类的实例发生器 `NumStr()` 中的参数值。如果有哪个值缺失了，与之对应的属性就取相应的假值（整数0或者是空字符串）做为它的缺省值。

需要特别留意的地方是在属性的名字里使用了双下划线。大家学习完下一小节内容就会明白这样做是为了增加一些私密性——虽然这些私密性也很初级。导入我们这个模块的程序员不能够直接访问我们的数据元素。采用只有通过访问器的函数功能才能对数据元素进行访问的办

法，我们试图强调的是OO设计中的“封装”概念。如果读者觉得这样做很奇怪或者不舒服，可以自己把实例属性上的那些双下划线都去掉，这样做不会影响这个程序示例的工作情况。

所有名字以双下划线(____)开始的属性都会被“编排”，这样在程序执行期里这些名字就不那么容易被访问了。但这种编排本身并不复杂，逆向解码工程没有多困难。事实上，这种编排操作的模式可以说是广为人知，也很容易被察觉。其实，这样做的主要目的是为了在被导入到一个可能会引起名字冲突的外部模块时出现意外的重名现象。名字被替换为一个包含着类的名字的新标识符以防止它被意外地“踩到”。详细内容请参考13.14节关于私密性的讨论。

第9~12行

我们把有序数据对的字符串表示形式选定为 “[num::'str']”。这样，只要出现把str()用到我们的实例身上或者我们的实例出现在print语句中的情况，就要靠__str__()特殊方法来提供相应的内容形式了。因为我们想强调第二个元素是一个字符串，所以用户看到这个字符串时它是被括在引号里的。为此，我们特意用单反引号调用了repr()给字符串的取值结果明确地加上了单引号，如下所示：

```
>>> print a
[3 :: 'foo']
```

如果不在self.__string上调用repr()（即去掉了self.__string两头的单反引号），就显示不出字符串两端的引号来。出于学习的目的我们这样演示一遍。去掉反引号，我们把那条return语句修改为如下所示的样子：

```
return '%d :: %s' % ( self.__num, self.__string )
```

现在用print语句查看一个实例的结果是：

```
>>> print a
[3 :: foo]
```

没有引号看上去的感觉如何？不如'foo'那样看上去更像是一个字符串，对吧？它看上去更像是一个变量。我都觉得不太可信呢。（我们赶紧把刚才的这个修改忘掉，就当它从来都没有发生过。）

跟接在__str__()函数后面的第一个语句把这个函数又赋值给另外一个特殊方法的名字，__repr__()。我们决定让实例的取值字符串表示形式和它的可打印字符串表示形式一致起来。我们没有用整个抄写一遍__str__()的办法定义这个新函数，我们简单地创建了一个别名，也就是复制了__str__()的引用线索。

对__str__()函数里的代码来说，如果用户把这个类的一个实例做为str()内建函数的参数，解释器就会调用这段代码对那个对象进行处理。__repr__()和repr()的关系也是如此。

如果选择不实现__repr__()，我们的执行情况又会怎样呢？如果去掉这个赋值语句，那就只有print语句（该语句将调用str()）能够用来查看对象的内容。字符串的可取值表示形式在缺省情况下是Python标准的<...some_object_information...>形式。如下所示：

```
>>> print a          # calls str(a)
[3 :: 'foo']
>>> a                # calls repr(a)
<NumStr.NumStr instance at 122640>
```

第14 ~ 29行

我们打算添加到这个类的功能是加操作，我们已经在前面具体介绍过对加操作的要求了。Python语言在对类进行定制这方面有一个很强大的功能，就是允许重载操作符以便让那些定制出来的操作更“真实”。调用一个“add(obj1, obj2)”模样的函数把对象obj1和obj2“加”起来看上去像是加法操作；可要是使用加号(+)（比如说像obj1 + obj2这样）就可以完成同样的操作，是不是更吸引人？

重载加号需要实现两个函数，它们是__add__()和__radd__()，这两个函数我们在前一小节里已经详细地讨论过了。__add__()函数负责处理SELF+OTHER（自身+其他）的情况，但我们需要定义一个__radd__()函数以便处理OTHER+SELF（其他+自身）的情况。这两个函数对数值的加法操作来说没有什么区别；但对字符串来说就不同了，因为字符串的合并操作与其先后次序有直接关系。

加操作把两个对应着的元素加起来，然后以加法结果做为数据项产生一个新的对象——把加法结果传递到以self.__class__()函数（这个函数也在前面解释过）形式调用的实例化调用中去就可以产生这个新的对象。如果对象的类型与规定的不符，就会引发一个TypeError例外，我们对这种情况也有所准备。

第31 ~ 37行

我们还重载了星号(*)操作符（通过实现__mul__()函数），它的作用是完成数值的乘法和字符串的重复操作，然后用操作的结果经实例化过程创建一个新的对象。因为重复操作只允许在操作符的右边出现一个整数，所以我们也必须严格遵守这一规定。因为同样的原因，我们也没有定义__rmul__()函数。

第39 ~ 40行

Python对象有这样一个概念上的特点，就是总有一个布尔值。对标准类型来说，如果对象是等于零的数值，或者是一个空序列和空映射，它就具有一个假值。对我们的类来说，我们选定的是只有在它的数值是零且字符串为空的情况下才让该实例具有一个假值。为此我们覆盖了__nonzero__()方法。其他严格仿真序列和映射类型的对象把长度为零的情况定义为假值。如果遇到那样的情况，就需要实现__len__()方法来影响此功能。

第42 ~ 54行

__norm_cval()不在特殊方法之列。它是一个帮助我们覆盖__cmp__()的辅助函数；它的主要作用是把cmp()方法所有大于零的返回值都转换为1，把所有小于零的返回值都转换为-1。cmp()会通常都会根据比较操作的结果返回一个正值或一个负值（还有零值）。但我们这里必须把返回值严格地限制为只有-1、0、1。用整数调用cmp()就可以给出我们希望的结果，它相当于下面这段代码：

```
def __norm_cval(self, cmpres):
    if cmpres < 0:
        return -1
    elif cmpres > 0:
        return 1
    else:
        return 0
```

两个这类的对象之间实际进行的比较操作由数值和字符串的比较操作组成，返回值是上述比较的和。读者可能已经注意到：在上面的代码里，我们在数据属性名字的前面都加上了双下划线。这样做的可以稍微增加一些私密性。

13.14 私密性

在缺省的情况下，Python语言中的属性一直都是“公共”的，能够被本模块内部和导入包含这个类（class）的本模块的模块中的代码访问。

许多OO语言都为数据提供了几个不同水平的私密性保护，只有通过访问器函数才能对数据的值进行访问。这叫做实现数据隐藏，这也是对象封装的一个关键步骤。大多数OO语言通过“访问限制器”来限制对成员函数的访问。 [1.5]

Python 1.5版本引入了一个针对类元素（属性和方法）的初级私密性处理办法。那些名字以一个双下划线（__）开始的属性在执行期间都会被“编排”，使之不容易被直接访问到。这个编排过程是这样：在名字前面加上一个下划线后，再放它到与之对应的那个类的名字后面。我们以程序示例13-2（numstr.py）中的self.__num属性为例来说明编排的具体情况。编排过程完成后，现在用来访问该数据的值的标识符变成了self._NumStr__num。把类的名字加到编排出来的新名字里去能够避免祖先或后代类里因为同名现象而发生崩溃。

虽然这提供了一定的私密性，但算法本身仍处于公共域内，破解起来很容易。它更像是一种保护性的措施，主要用途是导入那些不能直接访问其源代码的模块，或者用来导入同一模块中的其他代码。

防止访问源代码的办法之一是只允许访问经字节编译得到的.pyc文件。比如说，一家开发Python软件的公司可以选择只向公众提供.pyc文件。这样做可以增强安全控制，保证不会有人恶意地通过可执行程序窃取私密性变量和方法的访问权限。

我们在第12章里已经介绍过，Python语言提供有一种简单的模块级私密性保护措施，即在属性名字的前面加上一个下划线字符。这些名字里有下划线的模块属性将不会被“from mymodule import *”语句导入。

13.15 对类型进行打包

13.15.1 打包

“打包”是使用Python语言进行程序设计时经常会听到的术语。它描述的是对某个现有对象（不管它是一个数据类型还是一段代码）进行包装的过程，包括添加新内容、删除无用功能、对现有对象的现有功能进行各种改进等等。

在Python语言里，对一个标准类型进行子类分离或者推导是不允许的；但我们可以把某个类型打包为类的核心成员，这样新对象的操作行为就模仿着你想保留的原数据类型的操作行为，那些你不想保留的操作行为可以不要，而且还可以额外添些功能。这个过程叫做“对类型进行打包”。在本书后面的附录里，我们还会介绍另外一种形式的打包方案，即如何扩展Python语言的功能。

说到打包，就要定义新的类（class），让它的实例具备某个标准类型的核心操作行为。换句话说，它不仅能够唱歌和跳舞，就连走路和说话都和原始类型一模一样。图13-2给出了类型打包在一个类里面的情况。这个标准类型的核心操作行为位于图的正中央，并且通过一些新功能或升级功能得到了增强，这些新增功能也许会是一些能够访问实际数据的新方法。

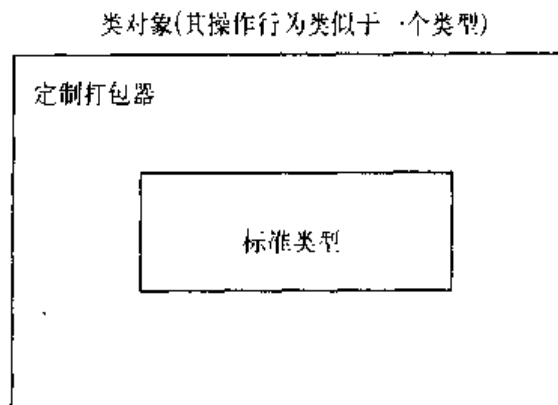


图13-2 对一个类型进行打包

对类也可以进行打包，但这样做似乎没什么道理，因为刚才描述的对标准类型进行打包的做法完全适用于对某个对象进行打包的情况，两种做法的原理是差不多的。而拿一个现有的类来模仿你自己需要的操作行为，去掉一些不需要的动作，再加上一些调整使那个打包出来的类和原来那个类完成的工作不一样，你又会怎样来完成这一系列操作呢？对了，这就是我们前面介绍过的子类推导过程。我们只对类型进行打包的原因是因为不能从它们开始分离出子类来。

13.15.2 实现对类型进行的打包

“对类型进行的打包”也叫做“代表”，它的意思是用打包后的对象做为原类型在操作上的代表。代表是打包操作的一个特性，通过它可以简化实现某些功能的处理过程。

代表是打包操作的一种形式，它利用已经存在的功能最大限度地重复使用过去编写出来的代码。对一个类型进行打包通常会涉及到要对现有类型进行一些定制处理。例如我们在前面已经提到过的，这些“调整”就是在原参照物的基础上新增、修改和删除某些功能；其余一切都保持原来的样子，即保留它原有的功能和操作行为。代表指的就是这样一种情况：所有经过修改的功能都被当做新类的一部分去处理，不用修改的原有功能则由新对象的缺省属性们去“代表”。

实现代表的关键之处是：必须用一段包含有内建`getattr()`函数的代码覆盖掉`__getattr__()`方法。说得更明确一些，调用`getattr()`的目的是为了获取对象的缺省属性（数据属性或方法）并返回它们以做进一步的访问或调用。`__getattr__()`特殊方法的工作原理是这样的：当需要检索某个属性的时候，系统会先在局部属性中查找（也就是先从定制属性中开始查找）；如果找不到，就会调用`__getattr__()`，再由它调用`getattr()`获取一个对象的缺省属性。

换句话说，在引用一个属性的时候，Python解释器会先尝试在局部名字空间里查找这个名字，比如说经过定制的方法或者局部实例的属性等。如果在局部字典里没有找到它，就会到那个类的名字空间里去查找，看看准备访问的是否是那个类的一个属性。最后，如果两次查找都失败

了，查找就会代表原始对象的操作请求，于是开始调用`__getattr__()`。

1. 对任意对象进行打包的简单例子

我们来看一个例子。下面例子中的类几乎可以对任何对象进行打包，其基本功能有用`repr()`和`str()`方式提供字符串表示形式等。其他定制功能包括一个`get()`方法，它的作用是去掉打包功能，返回未经修饰的对象。所有其他功能都由对象自身的固有的属性来代表，这些固有属性在必要时通过`__getattr__()`来检索。

下面就是这个示例性的打包类：

```
class WrapMe:

    def __init__(self, obj):
        self.__data = obj

    def get(self):
        return self.__data

    def __repr__(self):
        return 'self.__data'

    def __str__(self):
        return str(self.__data)

    def __getattr__(self, attr):
        return getattr(self.__data, attr)
```

我们把复数做为讲解的第一个例子，因为Python语言所有的数值类型中只有复数带有属性、数据属性及`conjugate()`内建方法。属性可以是数据属性，也可以是函数或者方法。此外，我们选择复数的原因还因为它也是同时具有两种属性的标准例子。下面是对一个复数进行操作的例子：

```
>>> wrappedComplex = WrapMe(3.5+4.2j)
>>> wrappedComplex          # wrapped object
[repr()]
(3.5+4.2j)
>>> wrappedComplex.real     # real attribute
3.5
>>> wrappedComplex.imag     # imaginary attribute
42.2
>>> wrappedComplex.conjugate() # conjugate() method
(3.5-4.2j)
>>> wrappedComplex.get()    # actual object
(3.5+4.2j)
```

在创建出被打包的对象类型后，我们通过交互式解释器调用`repr()`而获得了一个字符串表示形式。接下来我们访问了复数的三个属性，这些属性没有一个是我们的类里面定义的。对这三个属性的访问都需要通过`getattr()`方法用该对象的属性来代表。我们例子中的最后一个访问是检索一个确实是为我们的对象而定义的属性，这时候要用`get()`方法来返回经我们打包的那个数据对象本身。

接下来的例子在我们的打包类里使用了一个列表。我们先创建出那个对象，再执行了多个

操作，每个操作都是用列表本身的方法来代表的。如下所示：

```
>>> wrappedList = WrapMe([123, 'foo', 45.67])
>>> wrappedList.append('bar')
>>> wrappedList.append(123)
>>> wrappedList
[123, 'foo', 45.67, 'bar', 123]
>>> wrappedList.index(45.67)
2
>>> wrappedList.count(123)
2
>>> wrappedList.pop()
123
>>> wrappedList
[123, 'foo', 45.67, 'bar']
```

请注意，虽然在我们的例子里使用的是一个类的实例，但它们显示出来的操作行为与我们对它进行打包的数据类型是极其相似的。请大家记住，只有对确实存在的属性才能进行代表操作。

如果某种特殊的操作行为没有出现在原来那个类型的方法清单里，那它就是不可访问的，因为它们不是属性。比如说，列表的切片操作就是该类型的一个内建操作，不象append()方法那样是个属性。另外一个原因是切片操作符（[]）是序列类型的一个组成部分，不需要通过__getitem__()特殊方法来实现。

```
>>> wrappedList[3]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "wrapme.py", line 21, in __getattr__
    return getattr(self.data, attr)
AttributeError: __getitem__
```

出现AttributeError例外的原因是切片操作调用了__getitem__()方法，而__getitem__()既没有被定义为一个类的实例的方法，它也不是列表对象的方法。只有在对一个实例的字典或者一个类的字典穷尽检索之后仍然没有找到成功的匹配时才会调用getattr()。正如读者在上面看到的那样，对getattr()的调用是失败的，正是它引发了那个例外。

我们可以通过直接访问那个对象本身（通过我们的get()方法）和它的切片能力来瞒天过海，如下所示：

```
>>> realList = wrappedList.get()
>>> realList[3]
'bar'
```

读者现在可能已经明白我们为什么要实现get()方法了——就是为了处理需要直接访问原始对象本身这一情况。如果直接通过访问调用去访问对象的属性，我们就可以回避对局部变量(realList)进行赋值的问题，如下所示：

```
>>> wrappedList.get()[3]
'bar'
```

get()方法返回那个对象，然后立刻通过它的下标获得了切片子集。再看下面的例子：

```
>>> f = WrapMe(open('/etc/motd'))
>>> f
<open file '/etc/motd', mode 'r' at 80e95e0>
>>> f.readline()
```

```
'Have a lot of fun...\012'
>>> f.tell()
21
>>> f.seek(0)
>>> print f.readline(),
Have a lot of fun...
>>> f.close()
>>> f
<closed file '/etc/motd', mode 'r' at 80e95e0>
```

一旦读者熟悉了某个对象的属性，就会知道某个特定的信息是在哪个位置生成的，也就能够在新知识的基础上反复应用有关的功能，比如说：

```
>>> print "<%s file %s, mode %s at %x>" % \
... (f.closed and 'closed' or 'open', 'f.name',
'f.mode', id(f.get()))
<closed file '/etc/motd', mode 'r' at 80e95e0>
```

这是我们对类进行简单打包的最后一个例子。通过对类型的仿真，我们开始接触到对类进行定制的领域。你会发现可以通过对自己的代码进行各种各样的改进进一步扩大它的用途。比如说，你可以通过改进自己的代码给对象加上一个时间标签。在接下来的内容里，我们将给我们打包的类增加另外一个属性：时间。

2. 对我们这个简单的打包类进行改进

创建时间、修改时间和访问时间都是一些比较常见的文件属性，但这并不等于说不能给对象加上这种信息。不管怎样，肯定会有一些应用程序能够受益于这些额外提供的信息。

鉴于读者可能不太熟悉这三个与时间有关的数据，所以我们在这里先介绍些预备知识。创建时间（即“ctime”）是进行实例化操作时的时间；修改时间（即“mtime”）指的是核心数据被修改（通过调用新的set()方法来实现）的时间；而访问时间（即“atime”）是该对象的数据值最近一次被检索或者某个属性最近一次被访问的时间标签。

现在来改进我们前面定义的那个类，这就是列在程序示例13-3里面的twrapme.py模块。

我们是如何对代码进行改进的呢？首先，读者应该注意到多了三个新的方法，它们分别是：gettimeval()、gettimestr()和set()。我们还添加了一些代码，对有关时间标签的刷新就是由这些代码根据访问操作的具体类型完成的。

gettimeval()方法的参数是一个单独的字符，创建、修改或者访问时间分别对应于“c”、“m”和“a”；它返回的是一个以浮点数形式保存着的相应的时间值。gettimestr()返回的是一个用time.ctime()函数安排好时间格式后得到的可打印字符串版本。

我们来试试这个新模块。我们已经见过代表操作是如何进行的，所以我们在打包对象的时候就不增加什么属性了，这样可以突出我们刚添上的新功能。

程序示例13-3 对标准类型进行打包(twrapme.py)

可以对任何内建类型进行打包的类定义，加上时间属性、get()、set()和以字符串形式显示时间值的方法；对其他属性的访问都用原标准类型的属性来代表。

```
1 #!/usr/bin/env python
2
3 from time import time, ctime
```



```

4
5 class TimedWrapMe:
6
7     def __init__(self, obj):
8         self.__data = obj
9         self.__ctime = self.__mtime = \
10             self.__atime = time()
11
12     def get(self):
13         self.__atime = time()
14         return self.__data
15
16     def gettimeval(self, t_type):
17         if type(t_type) != type('') or \
18             t_type[0] not in 'cma':
19             raise TypeError, \
20                 "argument of 'c', 'm', or 'a' req'd"
21         return eval('self.__s__%stime' % \
22                     (self.__class__.__name__, t_type[0]))
23
24     def gettimestr(self, t_type):
25         return ctime(self.gettimeval(t_type))
26
27     def set(self, obj):
28         self.__data = obj
29         self.__mtime = self.__atime = time()
30
31     def __repr__(self):          # rep()
32         self.__atime = time()
33         return 'self.__data'
34
35     def __str__(self):          # str()
36         self.__atime = time()
37         return str(self.__data)
38
39     def __getattr__(self, attr): # delegate
40         self.__atime = time()
41         return getattr(self.__data, attr)

```

```

>>> timeWrappedObj = TimedWrapMe(932)
>>> timeWrappedObj.gettimestr('c')
'Wed Apr 26 20:47:41 2000'
>>> timeWrappedObj.gettimestr('m')
'Wed Apr 26 20:47:41 2000'
>>> timeWrappedObj.gettimestr('a')
'Wed Apr 26 20:47:41 2000'
>>> timeWrappedObj
932
>>> timeWrappedObj.gettimestr('c')
'Wed Apr 26 20:47:41 2000'
>>> timeWrappedObj.gettimestr('m')
'Wed Apr 26 20:47:41 2000'
>>> timeWrappedObj.gettimestr('a')
'Wed Apr 26 20:48:05 2000'

```

请注意，当对某个对象第一次进行打包时，它的创建、修改和最后一次访问的时间都是一样的值。访问那个对象以后，它的访问时间就会被刷新，但其他时间值不受影响。如果我们用 `set()` 替换了那个对象，修改时间和最后一次访问时间就都会发生变化。最后，对次对象的一个读操作结束了我们的例子。

```

>>> timeWrappedObj.set('time is up!')

```

```
>>> timeWrappedObj.gettimestr('m')
'Wed Apr 26 20:48:35 2000'
>>> timeWrappedObj
'time is up!'
>>> timeWrappedObj.gettimestr('c')
'Wed Apr 26 20:47:41 2000'
>>> timeWrappedObj.gettimestr('m')
'Wed Apr 26 20:48:35 2000'
>>> timeWrappedObj.gettimestr('a')
'Wed Apr 26 20:48:46 2000'
```

3. 用改进后的办法打包一个对象

下面是一个把文件对象打包为一个类的例子。我们的类在操作行为方面和普通的文件对象完全一致，只是在写模式下，只有全部是大写字母的字符串才能被写到文件里去。

需要我们在这里解决的问题是对一个文本文件执行写操作，而这个文本文件中的数据将被读入一台老式大型计算机中。许多老式的计算机都只能处理大写字母，所以我们希望实现这样一个文件对象：即把所有写入这个文件里去的文本都自动转换为大写字母，不让程序员担心在拿到老式计算机上去的时候会出现问题。事实上，比较值得注意的区别是我们不使用 `open()` 内建函数来打开一个这样的文件，我们将使用一个实例化 `CapOpen` 类的调用。这个新调用的参数和 `open()` 函数用的都完全一样。

请看程序示例13-4中的代码，这个程序的文件名是 `capOpen.py`。我们来看看使用这个类完成有关操作的一个例子：

程序示例13-4 打包一个文件对象 (`capOpen.py`)

这个类是在 Python FAQ4.48 的基础上扩展而来的，它提供了一个文件形式的对象，该对象对 `write()` 方法进行了定制。其余功能还是用原来的文件对象来代表。

```
1 #!/usr/bin/env python
2
3 from string import upper
4
5 class capOpen:
6     def __init__(self, fn, mode='r', buf=-1):
7         self.file = open(fn, mode, buf)
8
9     def __str__(self):
10         return str(self.file)
11
12     def __repr__(self):
13         return `self.file`
14
15     def write(self, line):
16         self.file.write(upper(line))
17
18     def __getattr__(self, attr):
19         return getattr(self.file, attr)
```

```
>>> f = capOpen('/tmp/xxx', 'w')
>>> f.write('delegation example\n')
>>> f.write('faye is good\n')
>>> f.write('at delegating\n')
>>> f.close()
>>> f
```

```
<closed file '/tmp/xxx', mode 'w' at 12c230>
```

如上所示，唯一与普通情况不同的调用是第一个调用，这里用的是`capOpen()`而不是`open()`。其他代码与你打算对一个真正的文件对象进行处理（而不是对一个在操作行为方面类似于一个文件对象的类的实例进行处理）时将要使用的代码是完全一致的。除`write()`方法外，所有其他的属性都用文件对象来代表。为了证明我们的代码是成功的，我们装入一个文件并显示出它的内容。（注意我们可以选用`open()`或者是`capOpen()`，但这个例子要求我们使用`capOpen()`完成有关操作。）如下所示：

```
>>> f = capOpen('/tmp/xxx')
>>> allLines = f.readlines()
>>> for eachLine in allLines:
...     print eachLine,
...
DELEGATION EXAMPLE
FAYE IS GOOD
AT DELEGATING
```

13.16 相关模块和文档

Python语言中有几个扩展了核心语言现有功能的类，我们在这一章里对它们都进行了论述。`User*`族的模块类似于方便食品，可以拿来就吃。我们提到了类可以有特殊的方法，如果实现了这些特殊的方法就可以对类进行定制，这在围绕一个标准类型进行打包时，给实例带来和类型一样的使用效果。

`UserList`和`UserDict`，再加上`UserString`（从Python 1.6版本开始引入的新功能）是几个对某些类进行定义的模块，这些类对列表、字典和字符串等对象分别打了包，模仿了这些对象的操作行为。这些模块的基本目的是向用户提供必要的功能，这样用户就不必自己动手去实现它们了；同时还能以它们为基本类来分离出子类以便做进一步的定制。Python语言已经为我们准备了大量有用的内建类型，但这种“由用户自己去创建”类型的额外能力更增添了这种语言的威力。

在第4章里，我们介绍了Python语言的标准类型和其他内建类型。`types`模块是进一步学习有关Python类型方面的知识的好地方，其中不少内容超出了我们这本书的讨论范围。`types`模块还定义了一些能够对之进行比较操作的类型（`type`）对象（这种比较在Python里是很常见的，因为它不支持方法的重载——这使这种语言比较简单，但它同时又提供了一些工具，可以在这种语言有欠缺的地方添加功能）。

下面的代码检查传递到`foo`函数的`data`对象是否是一个整数或字符串，不允许其他任何类型的出现（会引发例外）：

```
def foo(data):
    if type(data) == type(0):
        print 'you entered an integer'
    elif type(data) == type(''):
        print 'you entered a string'
    else:
        raise TypeError, 'only integers or strings!'
```

虽然上面的代码执行起来没有问题，但用types模块属性来做就更清晰：

```
from types import *
def foo(data):
    if type(data) == IntType:
        print 'you entered an integer'
    elif type(data) == StringType:
        print 'you entered a string'
    else:
        raise TypeError, 'only integers or strings!'
```

最后一个相关模块是operator模块。这个模块提供了大多数Python标准操作符的函数版本。在某些场合，这种形式的操作接口会比标准操作符的硬编码用法更灵活有效。

请看下面的例子。在阅读这些代码的时候请想象一下：如果在这个实现里使用的是一个的操作符，那会多写多少行代码：

```
>>> from operator import *          # import all operators
>>> vec1 = [12, 24]
>>> vec2 = [2, 3, 4]
>>> opvec = (add, sub, mul, div)    # using +, -, *, /
>>> for eachOp in opvec:           # loop thru operators
...     for i in vec1:
...         for j in vec2:
...             print '%s(%d, %d) = %d' % \
...                 (eachOp.__name__, i, j, eachOp(i, j))
...
add(12, 2) = 14
add(12, 3) = 15
add(12, 4) = 16
add(24, 2) = 26
add(24, 3) = 27
add(24, 4) = 28
sub(12, 2) = 10
sub(12, 3) = 9
sub(12, 4) = 8
sub(24, 2) = 22
sub(24, 3) = 21
sub(24, 4) = 20
mul(12, 2) = 24
mul(12, 3) = 36
mul(12, 4) = 48
mul(24, 2) = 48
mul(24, 3) = 72
mul(24, 4) = 96
div(12, 2) = 6
div(12, 3) = 4
div(12, 4) = 3
div(24, 2) = 12
div(24, 3) = 8
div(24, 4) = 6
```

上面这段代码定义了三个向量，前两个包含着操作数，最后一个给出了程序员打算对两个操作数进行的一系列操作。最外层的循环遍历每一个操作，而内层的两个循环用两个操作数向量中的元素组成各种可能的数据对。最后，print语句应用到当前操作符和组合出来的参数上。

我们前面介绍过的模块都列在表13-5中。

表13-5 与类相关的模块

模 块	说 明
UserList	提供了对列表对象的一个类打包器
UserDict	提供了对字典对象的一个类打包器
UserString	提供了对字符串对象的一个类打包器；它还有一个MutableString子类，如有必要，可以使用该子类提供的有关功能
types	为所有Python对象的类型定义了用在标准的Python解释器里的名字
operator	处理与站点有关的模块或软件包

① 新出现于Python 1.6版本。

在Python FAQ里有许许多多与类和面向对象的程序设计有关的问题。在13.5.3节里，我们提到用记录实例引用的办法来跟踪它们是危险的，这方面的进一步资料请参考Python FAQ 4.17。大多数与类和面向对象的程序设计有关的问题都可以在FAQ的第4和第6节里找到。

程序示例13-4的代码是根据Python FAQ 4.48里的代码实现编写的。这个FAQ问题的答案里包括一个简短的对文件对象进行打包和对write()方法进行修改的例子。我们给出的是一个完整的类，它可以像一个文件对象那样被使用，不再是简单地传递一个现有（即已经打开）的文件对象。我们的类有一个不足，就是它只对使用write()方法的文件起作用；所以我们在练习13-16里面要求由读者去实现一个writelines()方法。

最后，再提醒大家一次，《Python Library and Language Reference》（Python库和语言参考大全）是无价之宝。

13.17 练习

13-1 程序设计。请说出一些面向对象的程序设计比过去的程序设计先进的地方。

13-2 函数和方法的比较。函数和方法之间的区别是什么？

13-3 对类进行定制。创建一个把浮点数值转换为金额的类。我们在这个练习里使用的是美国货币，但读者可以自由发挥。

基本任务：编写一个dollarize()函数，它以一个浮点数值为输入，以字符串的形式返回一个金额，金额必须带有正确的符号，并进行了四舍五入。比如说：

```
dollarize(1234567.8901) ⇒ '$1,234,567.89'
```

dollarize()函数给出的金额里要有逗号——比如1,000,000，和美元的货币符号。如果有负号，那它必须出现在美元符号的左边。完成这项工作之后，就可以继续把它转换为一个有实际用处的类，我们称之为MoneyFmt类。

MoneyFmt类里只有一个数据值，即金额；有五个方法（你可以在这个练习外随意发挥）。__init__()构造器方法对那个数据值进行初始化；update()方法把那个数据值替换为一个新的金额；__nonzero__()方法是一个布尔函数，在数据值是一个非零值时返回1；__repr__()方法以一个浮点数的形式返回金额；而__str__()方法以与dollarize()同样的货币格式返回金额的字符串表示形式。

a) 请编写出update()方法的实现代码，让它能够对数据值进行修改。

b) 以前面dollarize()函数的成果为基础编写出__str__()方法的代码。

c) 纠正__nonzero__()方法中的漏洞, 这个漏洞会认为所有小于1的数值, 比如50美分(\$0.50), 有一个false (假) 值。

d) 附加题: 允许用户通过一个可选的参数指定是把负数数值显示在一对尖括号里还是显示一个负号。缺省参数应该是标准的负数符号 (-)。

我们在程序示例13-5中给出了moneyfmt.py程序的代码框架。在书后的CD-ROM光盘上还可以找到一个带注释(虽然不算完整)的moneyfmt.py程序。如果准备在解释器导入最终完成了的类, 执行过程应该和下面的情况差不多:

```
>>> import moneyfmt
>>>
>>> cash = moneyfmt.MoneyFmt(123.45)
>>> cash
123.45
>>> print cash
$123.45
>>>
>>> cash.update(100000.4567)
>>> cash
100000.4567
>>> print cash
$100,000.46
>>>
>>> cash.update(-0.3)
>>> cash
-0.3
>>> print cash
-$0.30
>>> repr(cash)
'-0.3'
>>> `cash`
'-0.3'
>>> str(cash)
'-$0.30'
```

程序示例13-5 金额转换程序 (moneyfmt.py)

这是一个字符串格式类, 设计目的是对浮点数值进行“打包”, 让它显示为带有正确符号的金额。

```
1  #!/usr/bin/env python
2
3  class MoneyFmt:
4
5      def __init__(self, value=0.) # constructor
6
7          self.value = float(value)
8
9      def update(self, value=None) # allow updates
10         ###
11         ### (a) complete this function
12         ###
13
14      def __repr__(self):          # display as a float
15          return `self.value`
```

```

16
17     def __str__(self):                # formatted display
18         val = ''
19
20         ###
21         ### (b) complete this function... do NOT
22         ###         forget about negative numbers!!
23         ###
24
25         return val
26
27     def __nonzero__(self):            # boolean test
28         ###
29         ### (c) find and fix the bug
30         ###
31
32         return int(self.value)

```

13-4 用户注册。创建一个用户数据库（包括登录名、口令字和上次登录时间标签）类（可以参考练习7-5和练习9-12）来管理一个系统，要求用户在访问允许访问的资源之前必须先登录。这个数据库类对系统的用户进行管理，在进行实例化操作时要加载上以前保存的用户资料，提供访问器函数来添加或者更新数据库中的信息。如果有所修改，数据库要把保存新资料到磁盘作为它回收操作（请参考__del__()特殊方法）的一部分。

13-5 几何。创建一个Point类，它由代表某个点的X和Y坐标的一个有序数值对构成。X和Y坐标在实例化时要传递给构造器；如果缺失某个坐标，就默认为坐标原点。

13-6 几何。创建一个直线/直线段的类，除主要的数据属性——即两个点坐标（请参考上一个练习）之外，它还具有长度和斜率属性。需要覆盖__repr__()方法（如果愿意，再算上str()）以使一条直线（或直线段）的字符串表示形式是由两个表列构成的表列，即((x1, y1), (x2, y2))。问题总结：

__repr__() 把直线的起止点显示为一对表列。

length 返回直线段的长度。不要使用“len”，因为“len”让人觉得长度值好象应该是个整数似的。

slope 返回这个直线段的斜率（必要时可以返回None）。

13-7 数据的类。提供一个time模块的操作接口，让用户可以用几种给定的形式（比如“MM/DD/YY”、“MM/DD/YYYY”、“DD/MM/YY”、“DD/MM/YYYY”、“Mon DD, YYYY”或者标准的UNIX日期格式“Day Mon DD, HH:MM:SS YYYY”）查看日期。你的类应该自己提供一个日期值，用给定的时间值创建一个实例。如果没有给出时间值，默认使用当前的系统时间。有关方法包括：

update() 用给定时间修改数据值，缺省值是当前的系统时间。

display() 以指示时间显示方式的字符串为参数按指定格式显示日期和时间：

‘MDY’ -> MM/DD/YY

‘MDYY’ -> MM/DD/YYYY

‘DMY’ -> DD/MM/YY

‘DMYY’ -> DD/MM/YYYY

‘MODYY’ -> Mon DD, YYYY

如果没有指定时间的显示格式，就默认使用系统时间或`ctime()`函数输出的格式。

附加题：把这个类和练习6-15结合起来。

13-8 Stack堆栈类。堆栈是一种具有后进先出（last-in-first-out, LIFO）特性的数据结构。我们可以把堆栈想象为一个碗架。最先摆上去的碗将是最后一个取下来的，而最后摆上去的碗将是最先一个取下来的。你的类需要有`push()`方法（往堆栈里压入一个数据项）和`pop()`方法（从堆栈里取出一个数据项）。再编写一个`isempty()`布尔方法，如果堆栈为空返回1，否则返回0；一个`peek()`方法，返回堆栈顶部的数据但不取出它。 [1.5.2]

注意：如果你使用一个列表来实现你的堆栈，那么从Python 1.5.2开始`pop()`方法就已经存在了。要是这样的话，请在你的新类里加上一些代码检查是否已经有了`pop()`方法。如果确实是已经有了`pop()`方法，就调用这个内建的方法；否则就执行你实现的`pop()`方法。你可能会使用一个列表对象来完成这个练习；如果真是这样，可以随意实现列表的其他功能（比如切片等）。必须保证你的Stack类能够正确完成上面提到的两项功能。可以参考13.16节内容和程序示例6-2。

13-9 Queue队列类。队列是一种具有先进先出（first-in-first-out, FIFO）特性的数据结构。一个队列就像是人们排的一个队伍，数据从前端取出，从后端加入。这个类必须支持下面几种方法：

`enqueue()`——在队列的尾部加入一个新元素。

`dequeue()`——在队列的头部取出一个元素，返回它并从队列里删除掉它。

可以参考上一个练习和程序示例6-3。

13-10 堆栈和队列。编写一个类型，定义一个同时具备队列（FIFO）和堆栈（LIFO）操作行为的数据结构，这个东西和Perl语言中数组的性质很接近。需要实现四个方法：

`shift()`——返回并删除序列中的第一个元素，类似于上一个练习中的`dequeue()`函数。

`unshift()`——在序列的头部“压入”一个新元素。

`push()`——在序列的尾部加上一个新元素，类似于前面练习中的`enqueue()`方法和`push()`方法。

`pop()`——返回并删除序列中的最后一个元素，和前面练习中的`pop()`方法完全一样。

请参考练习13-8和13-9。

13-11 电子商务。需要读者为一家B2C（商业到消费者）零售商编写一个电子商务引擎的基础。你需要有一个对应于顾客的类，我们给它取名为User；一个对应于库存商品的类，我们给它取名为Item；还要有一个对应于购物推车的类，我们给它取名为Cart。商品要放到购物推车里去，顾客可以有多个购物推车。购物推车里可以有多件商品，包括多件同样的商品。

13-12 聊天室。你对目前的聊天室应用程序感到很失望，并决心自己编写一个，创建一家新的因特网公司，接受风险投资，把广告集成到你的聊天程序里去，争取在6个月的时间里让投资翻两番，股票上市，然后退休。但是，如果你没有一个非常酷的聊天软件，这一切就都不会发生。

你需要用到三个类：一个Message类，它包含着一个消息字符串和诸如广播或私聊收件人等其他附加信息；一个User类，里面是进入你聊天室的某个人全部资料。为了能够真正从风险投资家那里赢得你的启动资金，你添加一个Room类：它体现了一个更加复杂的聊天系统，用户可以在聊天区里创建单独的“聊天屋”并邀请其他人加入。附加题：请为用户开发一个图形化用

户界面 (GUI) 应用程序。

13-13 股票档案类。你的数据库要记录每个公司的名字、股票代码、购买日期、购买价格和持股数量。需要实现的方法包括：添加股票代码、删除股票代码、根据一个给定价格（或者日期）计算任一或者全部股票的收益或年回报率。

13-14 DOS。为DOS机器编写一个UNIX操作界面的shell。你向用户提供一个命令行，用户在那里敲入UNIX命令，你对这些命令进行解释并给出与之对应的输出。比如说，ls命令将调用dir显示一个子目录中的文件清单；more调用同名命令（逐页显示一个文本文件）；cat调用type；cp调用copy；mv调用ren；rm调用del等。

13-15 代表。程序示例13-4的执行情况已经证明了capOpen类是能够成功地按要求完成写操作的。在最后的评论里，我们提到可以使用capOpen()或open()来读文件中的文本。为什么？这两者使用起来有什么差异吗？

13-16 代表和函数化程序设计。

a) 请为程序示例13-4中的capOpen类编写一个writelines()方法。这个新函数一次可以读入多行文本，然后在执行些操作之前把它们转换为大写的形式，它与程序示例13-4中的write()之间的差异与正常writelines()和write()方法之间的差异是一样的。注意，这个练习完成之后，writelines()就不再由文件对象来“代表”了。

b) 在writelines()方法里加上一个参数，用它来指示是否需要给各行文本加上一个换行符。这个参数的缺省值必须设置为0，表示不加换行符。

第14章 执行环境

在Python语言环境里执行一个命令或者执行磁盘上的一个文件可以有許多办法，这要看你到底想干什么。在执行过程中，会出现各种各样的可能性，比如说：

- 在当前脚本程序里继续执行。
- 创建并管理一个子进程。
- 执行一个外部命令或程序。
- 执行一个要求有输入的程序。
- 调用一个跨网络的命令。
- 执行一个命令，它产生的输出需要立刻处理。
- 执行另外一个Python脚本程序。
- 在一个安全环境里执行一个命令或者程序。
- 执行一组动态生成的Python语句。
- 导入一个Python模块（并执行它的顶层代码）。

能够提供上述功能的既有内建模块，也有外部模块。程序员要根据应用程序的特点选择适当的实现工具。本章内容将对Python执行环境的方方面面加以讨论；但我们不讨论如何启动Python解释器，也不讨论它的各种命令行参数。这方面内容请大家去复习第2章。

我们的Python执行环境之旅先去看看“可调用”对象，再从底层对code（代码）对象进行一些研究。然后，我们会去看看Python语言的语句和内建函数是如何支持我们需要实现的功能的。Python脚本程序能够执行其他程序，这一能力大大增强了Python脚本程序的实力，也大大节约了资源，因为重复实现所有这些代码不合乎逻辑，而且既费时又费力，绝不是明智的做法。Python提供了许多在当前脚本程序环境之外执行程序或命令的机制，我们将把其中一些最常见介绍给大家。接下来，我们会对Python的受限执行环境稍做介绍，最后是各种中止执行的办法（不让程序执行到正常完成）。我们就从Python执行环境的“可调用”对象开始吧。

14.1 可调用对象

Python对象中有一些是我们所说的“可调用的”，它指的是可以用函数操作符“()”进行调用的任何类型。要想调用可调用对象，必须把函数操作符紧贴着放在它名字的后面。比如说，函数“foo”就要用“foo()”来调用。读者早就知道这些事情了。可调用对象还可以通过函数化编程接口如apply()、filter()、map()和reduce()等被调用，而这些都是我们在第11章里介绍过的。Python语言有四种可调用对象，它们分别是：函数、方法、类和一些类示例。需要提醒大家的是，这些对象额外的引用或别名也是可调用的。

14.1.1 函数

我们介绍过的第一个可调用对象是函数。有三种不同类型的函数对象，其中的第一种是Python语言的内建函数。

1. 内建函数 (built-in function, BIF)

BIF一般都是用C或C++语言编写的扩展部件，它们被编译到Python解释器里并作为第一个（内建）名字空间的一部分被加载到系统中去。正如我们在前面章节里已经提到过的，这些函数都保存在__builtin__模块里，并且会以__builtins__模块的形式被导入到解释器里去。受限执行环境模式里只有这些函数的一个子集是可用的。（受限执行情况的细节请参考14.6节内容。）

所有BIF都有表14-1中列出的属性。

表14-1 内建函数的属性

BIF属性	说 明
bif.__doc__	文档字符串
bif.__name__	用字符串形式表示的函数名字
bif.__self__	设置为None（保留给内建方法使用）

用dir()内建函数能够查看到这些属性，请看下面对type()内建函数进行操作例子：

```
>>> dir(type)
['__doc__', '__name__', '__self__']
```

在Python语言的内部，内建函数和内建方法被归入同一个类型，所以对一个内建的函数或方法调用type()内建函数时会得到“builtin_function_or_method”这样的结果，请看下面的例子：

```
>>> type(type)
<type builtin_function_or_method>
```

2. 用户自定义函数 (user-defined function, UDF)

函数的第二种类型是用户自定义函数。这些函数通常都定义在模块最顶部的位置，并且会作为全局名字空间的一部分（在建立内建名字空间之后）被加载到系统中去。函数也可以定义在其他函数里面，此时内层函数就不能访问外层函数的局部作用范围。正如我们在前面章节里指出的，Python语言目前只支持两个作用范围，即全局作用范围和某个函数本身的局部作用范围。在一个函数里定义的所有名字，包括参数在内，都归入局部名字空间。

所有UDF都有表14-2中列出的属性。

表14-2 用户自定义函数的属性

UDF属性	说 明
udf.__doc__	文档字符串（udf.func_doc也是）
udf.__name__	用字符串形式表示的函数名字（udf.func_name也是）
udf.func_code	字节编译得到的代码（code）对象
udf.func_defaults	缺省参数的表列
udf.func_globals	全局名字空间字典；相对于在该函数内调用globals(x)

在Python语言的内部，用户自定义函数被定义为“function”类型，请看下面type()内建函数的使用示例：

```
>>> def foo(): pass
>>> type(foo)
<type 'function'>
```

3. lambda表达式（归入“<lambda>”类型的函数）

lambda表达式和用户自定义函数基本一样，两者之间只有很小的区别。虽然lambda表达式产生的也是函数对象，但它们是用lambda关键字而不是def语句创建的。

因为lambda表达式没有给与之对应的代码进行命名的内部结构，所以lambda表达式如果想要调用lambda表达式就只能通过函数化的程序设计接口或把它们的引用赋值给一个变量的办法来实现；它们可以通过函数化程序设计直接或者反复地调用。这个变量只是一个别名，不是函数对象的名字。

用lambda关键字创建的函数对象具有与用户自定义函数完全一样的属性，两者之间只有一个因前者没有名字而产生的区别，那就是前者的__name__或udf.func_doc属性是一个字符串“<lambda>”。

通过type()内建函数可以清楚地看出lambda表达式产生的函数对象和用户自定义函数是完全一样的：

```
>>> lambdaFunc = lambda x: x * 2
>>> lambdaFunc(100)
200
>>> type(lambdaFunc)
<type 'function'>
```

在上面的例子里，我们把表达式赋值给一个别名变量。我们也可以直接对一个lambda表达式调用type()内建函数，如下所示：

```
>>> type(lambda : 1)
<type 'function'>
```

我们以lambdaFunc和上一小节中的foo为例看看两者在UDF名字方面的区别：

```
>>> foo.__name__
'foo'
>>> lambdaFunc.__name__
'<lambda>'
```

14.1.2 方法

在上一章里，我们重点讨论了方法，即被定义为一个类的组成部分的函数——这是一些用户自定义方法。Python语言中的许多类型，比如列表和字典，都带有它们自己的方法，这些方法叫做内建方法。为了进一步明确地显示出这种“所有权”，在给方法命名的时候要通过点属性记号加上其所属对象的名字。

1. 内建方法（built-in method, BIM）

在上一小节里我们讨论了内建方法和内建函数的相似之处。只有内建类型（built-in type, BIT）才有BIM。从下面的内容可以看出，type()内建函数对内建方法和内建函数所给出的执行

结果都是一样的——请注意我们是如何通过一个内建类型（对象或引用）来访问BIM的：

```
>>> type(l.append)
<type 'builtin_function_or_method'>
```

再深入一些，BIM和BIF还都有相同的属性（参见表14-3）。两者之间唯一的区别是__self__属性现在指向的是一个（对应于BIM情况的）Python对象而不是（对应于BIF情况的）None。

表14-3 内建方法的属性

BIM属性	说 明
bim.__doc__	文档字符串
bim.__name__	用字符串形式表示的函数名字
bim.__self__	这个方法所绑定的对象

一般情况下，一个BIT有表14-4所示的BIM和（内建）属性。

表14-4 内建类型的属性

BIT属性	说 明
bit.__methods__	由（内建）方法构成的列表
bit.__members__	由数据属性构成的列表

对类和实例来说，如果把对象本身用做dir()的参数，就可以通过dir()内建函数获得它们的数据属性和方法属性。很明显，BIT有两个分别列出它们的数据属性和方法属性的属性。BIT的属性可以通过一个引用或者通过一个该对象本身来访问，请看下面的例子：

```
>>> aList = ['on', 'air']
>>> aList.append('velocity')
>>> aList
['on', 'air', 'velocity']
>>> aList.insert(2, 'speed')
>>> aList
['on', 'air', 'speed', 'velocity']
>>>
>>> [].__methods__
['append', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
>>> [3, 'headed', 'knight'].pop()
'knight'
```

从上面最后一个例子里可以看出，通过对象本身来访问它的方法并没有多大的功用。因为对象的引用没有被保存下来，所以它马上就被系统回收了。这种形式的访问只有一个用处，就是可以通过它查看一个BIT具有哪些方法（或成员）。

2. 用户自定义方法（user-defined method, UDM）

用户自定义方法都是在类里面定义的，它们只是一些对标准函数进行了“包装”修饰的“打包器”，也只能对定义它们的类起作用。如果没有在子类定义里被覆盖掉，它们还可以被子类的实例调用。

在上一章里我们解释过，UDM是与类（class）对象相关联的（此时它们是未绑定方法），但

只能和类的实例一起被调用（此时它们是绑定方法）。不管它们绑定与否，所有的UDM都被归入同一个类型“instance method”，就像下面调用type()的例子那样：

```
>>> class C:                                # define class
...     def foo(self): pass                  # define UDM
...
>>> c = C()                                # instantiation
>>> type(C.foo)                             # type of unbound method
<type 'instance method'>
>>> type(c.foo)                             # type of bound method
<type 'instance method'>
```

UDM具有表14-5所列的属性。

表14-5 用户自定义方法的属性

UDM属性	说 明
udm.__doc__	文档字符串
udm.__name__	以字符串形式表示的方法的名字
udm.im_class	与方法关联的类
udm.im_func	该方法的函数对象（参见关于UDF的内容）
udm.im_self	如果是绑定方法，这就是与之关联的实例；否则是None

通过直接访问对象本身可以了解你引用的是一个绑定的方法还是一个未绑定的方法。从下面的例子可以看出，绑定方法给出的是该方法绑定在其上的那个实例对象：

```
>>> C.foo                                # unbound method object
<unbound method C.foo>
>>>
>>> c.foo                                # bound method object
<method C.foo of C instance at 122c78>
>>> c                                    # instance foo()'s bound to
<__main__.c instance at 122c78>
```

14.1.3 类

类的可调用性使我们能够创建实例。“调用”一个类的后果就是创建一个实例来，也就是人们常说的实例化操作。类的构造器在缺省情况下什么动作也不执行，一般也就包含一条pass语句而已。程序员有权选择自行实现__init__()方法，从而对实例化过程进行定制。实例化调用的所有参数都会被传递到构造器去，如下所示：

```
>>> class C:
...     def __init__(self, *args):
...         print 'instantiated with these arguments:\n', args
...
>>> c1 = C()                             # invoking class to instantiate c1
instantiated with these arguments:
()
>>> c2 = C('The number of the counting shall be', 3)
instantiated with these arguments:
('The number of the counting shall be', 3)
```

我们对实例化过程和它的执行情况应该是很熟悉的了，所以就不多浪费笔墨了。但这里有

一个新问题，那就是怎样才能使实例成为可调用的。

14.1.4 类的实例

Python语言为类准备了一个名为`__call__()`的特殊方法，它允许程序员创建一个可调用的对象（实例）。在缺省的情况下，这个`__call__()`方法是没有实现的，也就是说，大多数实例都不是可调用的。但如果在类定义里覆盖了这个方法，就可以让那个类的实例成为可调用的，而调用一个这样的实例就相当于调用其`__call__()`方法。很自然，在实例调用中给出的一切参数都会被传递到`__call__()`去做参数。

不要忘记`__call__()`终究只是一个方法，所以那个实例对象本身必须作为第一个参数，即`self`参数，传递到`__call__()`里去。换句话说，如果`foo`是一个实例，那么`foo()`的作用和效果就完全等同于`foo.__call__(foo)`——括号中的`foo`是一个参数，就是调用任何一个方法时都会自动作为第一个参数的那个`self`。如果`__call__()`有参数，即`__call__(self, args)`，那么`foo(args)`就和调用`foo.__call__(foo, args)`的效果完全一样。我们在下面给出了一个可调用实例的示例；其中的例子和上一小节里用的差不多：

```
>>> class C:
...     def __call__(self, *args):
...         print "I'm callable! Called with args:\n", args
...

>>> c = C()                                # instantiation
>>> c                                       # our instance
<__main__.C instance at babb30>
>>> callable(c)                            # instance is callable
1
>>> c()                                    # instance invoked
I'm callable! Called with arguments:
()
>>> c(3)                                  # invoked with 1 arg
I'm callable! Called with arguments:
(3,)
>>> c(3, 'no more, no less')              # invoked with 2 args
I'm callable! Called with arguments:
(3, 'no more, no less')
```

在这一小节即将结束的时候我要提醒大家：除非对类进行定义时实现了`__call__()`方法，否则这个类的实例就不会是可调用的。

14.2 代码对象

虽然可调用对象是Python执行环境具有关键意义的组成因素，但它也只是棋盘上的一个棋子而已。Python语言中的语句、赋值、表达式还有模块等形成了更大的局面。但这些“可执行对象”与可调用对象之间有明显的区别，就是它们没有被调用的能力。这些对象是组成代码（code）对象（即能够执行的代码块）的小部件。

每一个可调用对象的核心都有一个由语句、赋值、表达和其他可调用对象构成的代码对象。我们可以把模块看做一个巨大的包含着该模块中所有代码的代码对象，它可以进一步分解为语句、赋值、表达和下一级的可调用对象，那些下级可调用对象又递归性地包含有它们自己的代

码对象。

一般说来，代码对象能够作为函数或方法调用的组成部分而被执行，我们还可以使用exec语句或eval()内建函数来执行它们。从居高临下的角度看，一个Python模块就是一个该模块全体代码行构成的代码对象。

任何Python代码，如果想执行，就必须先转换为字节编译代码（也叫做字节码（bytecode））。精确说来，这才是代码对象。它们不包含与其执行环境有关的任何信息，而这也正是可调用对象存在的原因——它的作用就是“打包”一个代码对象并提供额外的信息。

还记不记得上一小节里提到的UDF的udf.func_code属性？想不到吧，它就是一个代码对象。UDM的udm.im_func函数对象又会怎样呢？因为它也是一个函数对象，所以它也有自己的udm.im_func.func_code代码对象。现在就很清楚了，函数对象就是代码对象的“外包装”，而方法又是函数对象的“外包装”。从那里入手都可以，但当你到达终点时，看到的就是一个代码对象。

14.3 可执行对象语句和内建函数

Python语言为支持可调用对象和可执行对象准备了几个包括exec语句在内的内建函数。这些函数使程序员既能够执行代码对象，也能够利用compile()内建函数生成代码对象，它们都列在表14-6里。

表14-6 与可执行对象有关的语句和内建函数

内建函数或语句	说 明
callable(obj)	检查obj是否是可调用的：是就返回1，否则返回0
compile(string, file, type)	从type类型的字符串string创建一个代码对象；file是代码原来的存放位置（通常被设置为空字符串“”）
eval(obj, globals=globals(), locals=locals())	对obj求值，obj或者是一个被编译为代码对象的表达式，或者是一个字符串表达式；可以为它提供全局和/或局部的名字空间，如果没有提供，就使用当前环境中的缺省设置
exec obj	执行obj，obj可以是一条Python语句，也可以是一组语句；可以是代码对象，也可以是字符串格式。obj还可以是一个文件对象（在一个Python脚本程序里合法打开的文件）
input(prompt = ' ')	相当于eval(raw_input(prompt = ' '))函数
intern(string)	要求把string放到Python的内置字符串表里面去

14.3.1 callable()

callable()是判定一个对象类型是否可以用函数操作符（()）来调用的布尔函数。如果那个对象是可调用的，它返回1；否则返回0。下面是一些对象以及与之对应的callable()返回值：

```
>>> callable(dir)           # built-in function
1
>>> callable(1)             # integer
0
>>> def foo(): pass
```



```

...
>>> callable(foo)           # user-defined function
1
>>> callable('bar')         # string
0
>>> class C: pass
...
>>> callable(C)             # class
1

```

14.3.2 compile()

`compile()`这个函数允许程序员在程序执行期间就地生成一个代码对象，这些动态生成的代码对象马上就可以用`exec`语句来执行或者用`eval()`内建函数来求值。`exec`和`eval()`都可以接受字符串形式的Python代码并执行它，这一点非常重要。如果执行代码是以字符串形式给出的，那么每次执行时都要对这些代码进行字节编译。`compile()`函数提供了对代码的一次性字节编译操作，这样以后再执行这段代码时就不必进行预编译了。很明显，只有在同一段代码会被反复执行的时候这样的做法才有实际上的好处。在这种情况下，对代码进行预编译肯定会有好的效果。

`compile()`的三个参数都不能省略，必须明确地给出来。第一个参数是一个字符串，它就是准备进行编译的Python代码。第二个字符串虽然不能省掉，但经常被设置为一个空字符串。这个参数是保存这个代码对象的（字符串形式的）文件名。因为`compile()`的常见用法是根据动态生成的Python代码字符串来生成一个代码对象，而这些代码明显不是从一个现有文件里读出来的。

最后一个参数也是一个字符串，它指出了这个代码对象的类型。它可以取三种值：

- 'eval'可求值表达式（和`eval()`函数一起使用）
- 'single'一条可执行语句（和`exec`语句一起使用）
- 'exec'一组可执行语句（和`exec`语句一起使用）

1. 可求值表达式

```

>>> eval_code = compile('100 + 200', '', 'eval')
>>> eval(eval_code)
300

```

2. 一条可执行语句

```

>>> single_code = compile('print "hello world!"', '', 'single')
>>> single_code
<code object ? at 120998, file "", line 0>
>>> exec single_code
hello world!

```

3. 一组可执行语句

```

>>> exec_code = compile("""
... req = input('Count how many numbers? ')
... for eachNum in range(req):
...     print eachNum
... """, '', 'exec')

```

```
>>> exec exec_code
Count how many numbers? 6
0
1
2
3
4
5
```

14.3.3 eval()

eval()的作用是对一个表达式求值，表达式可以是字符串表示形式，也可以是由compile()内建函数编译而来的一个预编译代码对象。这是eval()的第一个参数。它的第二个和第三个参数都是可选的，分别给出了该代码对象的全局和局部名字空间。如果这后两个参数没有给出来，就分别默认为globals()和locals()返回的对象。请看下面的例子：

```
>>> eval('932')
932
>>> int('932')
932
```

在上面的例子里，eval()和int()给出了相同的结果：一个值为932的整数；但两个函数采取的办法是不同的。eval()把引号里的字符串看作是一个Python表达式并求出它的值来；而int()则认为那是一个整数的字符串表示形式，并把它转换为一个整数。上面例子里的字符串正好就是字符串'932'，所以它作为一个表达式求值得到的是932，作为一个整数的字符串表示形式'932'转换后得到的也是932。如果我们使用的是一个纯字符串形式的表达式，情况就有所区别了，如下所示：

```
>>> eval('100 + 200')
300
>>> int('100 + 200')
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ValueError: invalid literal for int(): 100 + 200
```

在这个例子里，eval()以字符串为参数把“100 + 200”当作一个表达式来求值，经过整数加法运算后得出结果是300。而int()的调用则失败了，因为字符串参数可不是一个整数的字符串表示形式——这个字符串里有非法字符，即空格和“+”字符。

要想把握eval()函数的执行情况可以这样做：对表达式两端的引号视而不见，然后问自己“如果我是Python解释器，应该怎样看这个表达式？”换句话说，如果这个表达式是以交互方式输入进来的，解释器会怎样反应？按下回车键后的输出应该和eval()的执行结果完全一样。

14.3.4 exec

类似于eval()，exec语句也是既可以执行一个代码对象，又可以执行Python代码的字符串表示形式。同样地，把可能会反复使用的代码用compile()预编译后会大大改进代码的执行性能，因为这样做了以后就不必在每次调用时都要经过字节码的编译过程了。exec语句只有一个参数，就象下面它的通用语法给出的这样：

```
exec obj
```

可执行对象 (obj) 既可以是一条语句, 也可以是一组语句; 两种情况都可以编译出一个代码对象 (编译时要分别使用single和exec), 另外它还可以就是字符串的形式。下面是一个把多条语句作为一个字符串发送给exec语句的例子:

```
>>> exec """
... x = 0
... print 'x is currently:', x
... while x < 5:
...     x = x + 1
...     print 'incrementing x to:', x
... """
x is currently: 0
incrementing x to: 1
incrementing x to: 2
incrementing x to: 3
incrementing x to: 4
incrementing x to: 5
```

最后, exec语句还能够接受一个合法的文件对象 (即一个合法打开的Python文件)。如果我们把刚才那多条语句保存到一个名为xcount.py的文件里去, 那么用下面的办法同样可以执行这些代码:

```
>>> f = open('xcount.py')          # open the file
>>> exec f                          # execute the file
x is currently: 0
incrementing x to: 1
incrementing x to: 2
incrementing x to: 3
incrementing x to: 4
incrementing x to: 5
>>> exec f                          # try execution again
>>>                                # oops, it failed... why?
```

请注意, 在一次成功的执行完成后, 后续的exec语句失败了。可事实上并没有什么失败, 它只是什么也没有做罢了, 也许就是这样才让你吃惊的吧。实际情况是这样的, exec语句已经读完了文件里的所有数据, 现在就待在文件尾 (EOF) 位置上。这样, 当我们使用同一个文件对象再次调用exec语句时, 已经没有可供执行的代码了, 所以它什么也没有做, 也就是我们看到的情况。怎样才能确定它确实是在EOF位置上呢?

我们用文件对象的tell()方法来报告我们在文件中的位置, 再使用os.path.getsize()函数报告xcount.py脚本程序有多长。你会看到, 两个操作的结果数值是完全一样的, 如下所示:

```
>>> f.tell()                        # where are we in the file?
116
>>> f.close()                       # close the file
>>> from os.path import getsize
>>> getsize('xcount.py')            # what is the file size?
116
```

1. 用Python语言生成和执行Python代码的例子

我们现在给出一个名为loopmake.py的脚本程序的代码, 它是一项简单的计算机辅助软件工

程 (computer-aided software engineering, CASE), 作用是就地生成循环并执行它。它提示用户输入各种参数 (比如循环的类型是while还是for, 被遍历数据的类型是数字还是序列, 等等), 生成相应的代码字符串, 然后立刻执行它 (见程序示例14-1)。

程序示例14-1 动态生成和执行Python代码 (loopmake.py)。

```

1  #!/usr/bin/env python
2
3  dashes = '\n' + '-' * 50          # dashed line
4  exec_dict = {
5
6      'f': """                        # for loop
7      for %s in %s:
8          print %s
9      """,
10
11     's': """                        # sequence while loop
12     %s = 0
13     %s = %s
14     while %s < len(%s):
15         print %s[%s]
16         %s = %s + 1
17     """,
18
19     'n': """                        # counting while loop
20     %s = %d
21     while %s < %d:
22         print %s
23         %s = %s + %d
24     """,
25 }
26
27 def main():
28
29     ltype = raw_input('Loop type? (For/While) ')
30     dtype = raw_input('Data type? (Number/Seq) ')
31
32     if dtype == 'n':
33         start = input('Starting value? ')
34         stop = input('Ending value (non-inclusive)? ')
35         step = input('Stepping value? ')
36         seq = str(range(start, stop, step))
37
38     else:
39         seq = raw_input('Enter sequence: ')
40
41     var = raw_input('Iterative variable name? ')
42
43     if ltype == 'f':
44         exec_str = exec_dict['f'] % (var, seq, var)
45
46     elif ltype == 'w':
47         if dtype == 's':
48             svar = raw_input('Enter sequence name? ')
49             exec_str = exec_dict['s'] % \
50             (var, svar, seq, var, svar, svar, var, var, var)
51
52         elif dtype == 'n':
53             exec_str = exec_dict['n'] % \
54             (var, start, var, stop, var, var, var, step)
55
56     print dashes
57     print 'Your custom-generated code:' + dashes
58     print exec_str + dashes
59     print 'Test execution of the code:' + dashes
60     exec exec_str
61     print dashes

```

```
62
63 if __name__ == '__main__':
64     main()
```

下面是这个脚本程序几次执行的情况:

```
% loopmake.py
Loop type? (For/While) f
Data type? (Number/Sequence) n
Starting value? 0
Ending value (non-inclusive)? 4

Stepping value? 1
Iterative variable name? counter
```

The custom-generated code for you is:

```
for counter in [0, 1, 2, 3]:
    print counter
```

Test execution of the code:

```
0
1
2
3
```

% loopmake.py
Loop type? (For/While) w
Data type? (Number/Sequence) n
Starting value? 0
Ending value (non-inclusive)? 4
Stepping value? 1
Iterative variable name? counter

Your custom-generated code:

```
counter = 0
while counter < 4:
    print counter
    counter = counter + 1
```

Test execution of the code:

```
0
1
2
3
```

% loopmake.py
Loop type? (For/While) f

```
Data type? (Number/Sequence) s
Enter sequence: [932, 'grail', 3.0, 'arrrghhh']
Iterative variable name? eachItem
```

```
-----
Your custom-generated code:
```

```
-----
for eachItem in [932, 'grail', 3.0, 'arrrghhh']:
    print eachItem
```

```
-----
Test execution of the code:
```

```
-----
932
grail
3.0
arrrghhh
```

```
-----
% loopmake.py
Loop type? (For/While) w
Data type? (Number/Sequence) s
Enter sequence: [932, 'grail', 3.0, 'arrrghhh']
Iterative variable name? eachIndex
Enter sequence name? myList
```

```
-----
Your custom-generated code:
```

```
-----
eachIndex = 0
myList = [932, 'grail', 3.0, 'arrrghhh']
while eachIndex < len(myList):
    print myList[eachIndex]
    eachIndex = eachIndex + 1
```

```
-----
Test execution of the code:
```

```
-----
932
grail
3.0
arrrghhh
```

2. 逐行解释

第1~25行

我们在脚本程序的开始部分设置了两个全局变量。它们一个是由一行短划线构成的静态字符串dashes，另一个是一个字典，字典里是我们准备生成的循环代码的骨架。字典中的键字“f”对应着for循环；“s”对应着对序列进行遍历的while循环；“n”对应着计数方式的while循环。

第27~30行

这里是给用户的使用方法提示，分别提示用户输入循环的类型和他或她打算使用的数据类型。

第32 ~ 36行

选择数字；这些数字分别对应着循环的起始值、终止值和循环增量值。在这部分代码里，我们第一次用到了input()内建函数。我们将在14.3.5节里看到，input()在提示用户进行字符串输入这方面与raw_input()是一样的，但与raw_input()不同的是input()会把用户的输入当作一个Python表达式进行求值，即使用户输入的是一个字符串，这个函数也会给出一个Python对象来。

第38 ~ 39行

选择一个序列；以一个字符串的形式输入序列。

第41行

获取用户准备用做循环遍历变量的变量名。

第43 ~ 44行

生成for循环，填写所有定制细节。

第46 ~ 50行

生成一个对序列进行遍历的while循环。

第52 ~ 54行

生成一个计数方式的while循环。

第56 ~ 61行

输出动态生成的源代码；同时输出的还有刚生成的代码执行得到的结果。

第63 ~ 64行

如果本模块被直接调用，就执行main()。

为了控制这个脚本程序长度，我们在这个脚本程序里面省略了对代码进行注释和对出错进行检查处理的部分。读者可以在书后所附的CD-ROM光盘上找到未加注释和出错检查以及加了注释和出错检查的两个程序版本。

扩展版本里包括的额外功能有：在字符串输入中允许缺少结束引号、输入数据的缺省值，以及检查是否出现非法数值范围和非法标识符等；它还禁止使用内建名字和关键字作为变量名。

14.3.5 input()

input()内建函数可以说是eval()和raw_input()的组合，它相当于eval(raw_input())。类似于raw_input()，input()也有一个可选的用做显示给用户作为字符串提示信息的参数。如果没有给出这个参数，这个字符串就会把空字符串作为自己的缺省值。

从功能上来说，input()与raw_input()是不同的，这是因为raw_input()总是原封不动地返回一个包含着用户输入的字符串。而input()虽然同样是完成获取用户输入的任务，但它会采取进一步的行动，把输入当作是一个Python表达式进行求值。这就意味着input()返回的数据是一个Python对象，即对输入表达式进行求值后得到的结果。

一个明显的例子是用户输入一个列表的时候。raw_input()返回的是列表的字符串表示形式，而input()返回的是那个列表本身。如下所示：

```
>>> aString = raw_input('Enter a list: ')
Enter a list: [ 123, 'xyz', 45.67 ]
```

```
>>> aString
"[ 123, 'xyz', 45.67 ]"
>>> type(aString)
<type 'string'>
```

上面的例子是用`raw_input()`完成的。正如你所看到的，全都是字符串。现在来看看使用`input()`会发生什么情况：

```
>>> aList = input('Enter a list: ')
Enter a list: [ 123, 'xyz', 45.67 ]
>>> aList
[123, 'xyz', 45.67]
>>> type(aList)
<type 'list'>
```

虽然用户输入的是一个字符串，但`input()`会把那个输入当作是一个Python对象来求值，最终返回的是该表达式的求值结果。

14.3.6 内置字符串和`intern()`

出于执行效率方面的考虑，Python保留着一个由静态字符串文字和标识符字符串构成的内置字符串表，它的目的是加快对字典进行检索的速度。把字符串放到内置字符串表里以后，无论何时，只要被引用的字符串文字或者标识符已经被内置到那个字符串表里了，就不再需要为那个字符串分配新的空间，这样就节省了对内存进行分配的时间。这个“内置的”字符串集合在退出解释器之前是不会被回收的。

下面是一个内置字符串的例子：

```
>>> id('hello world')
1040072
>>> foo = 'hello world'
>>> id(foo)
1040072
>>> del foo
>>> id('hello world')
1040072
```

在第一个语句里我们创建了一个字符串“hello world”。揭示该字符串标识的`id()`调用并不是必需的，因为不管是否有这个调用，这个字符串都会被创建出来。我们之所以在例子里这样做，是为了在创建这个字符串以后立刻把它的标识显示出来。把这个字符串赋值给一个标识符以后，我们又调用了一次`id()`，可以看到，`foo`引用的是同一个字符串对象，而这个字符串对象已经被内置了。即使我们删除了这个引用，也就是减少了引用计数，也可以看到这样的情况：这对已经内置的字符串是没有影响的。

更让人吃惊的是就连字符串“foo”本身也已经（作为一个标识符的字符串名字）被内置了。如果我们再另外创建两个新的字符串，就会看到“foo”字符串有一个比新字符串更早的ID值，这表示它是更早被创建的。如下所示：

```
>>> id('bar')
1053088
>>> id('foo')
1052968
```



```
>>> id('goo')
1053728
```

`intern()`是从Python 1.5版本开始出现的，它允许程序员明确地请求对一个字符串进行内置。读者可能已经猜到`intern()`的语法了，那就是：

```
intern(string)
```

给定的`string`参数就是将被内置的字符串。`intern()`会把这个字符串输入到（全局）内置字符串表里去。如果这个字符串还没有被放到过内置字符串表里去，它就会被内置并返回这个字符串。否则，如果这个字符串已经在表里了，简单地返回它就行了。

14.4 执行其他Python程序

当我们说起执行其他程序的时候，要把Python程序和所有其他的非Python程序区别对待，后者包括二进制可执行代码或其他脚本程序语言的源代码。我们先来讨论如何运行其他Python程序，然后再讨论如何利用`os`模块来调用外部的程序。

14.4.1 导入

要想在代码执行期间运行其他Python程序有好几种办法。正如我们在前面的章节里讨论过的，第一次导入一个模块会引起这个模块的顶层代码被执行。这是Python语言中的导入操作的行为准则，不管你愿不愿意都会如此。在这里要提醒大家的是：属于模块顶层的代码只有全局变量、类定义和函数定义。

这些顶层代码必需放在对`__name__`进行检查以确定是否调用了这个脚本程序的`if`语句（比如“`if __name__ == '__main__':`”语句）之前。这样，你的脚本程序才会执行代码的主程序部分；或者，如果这个脚本程序是准备被导入到其他模块里去的，它就会运行一个测试性子句对这个模块里的代码进行检查。

一个比较复杂的情况是这个被导入的模块本身也有`import`语句。如果这些`import`语句中的模块还没有被加载过，就会在此时加载它们并执行其顶层代码，从而形成一个递归形式的导入行为。我们在下面给出了一个简单的例子。我们有两个模块，分别是`import1`和`import2`，在它们的最外层都有`print`语句。`import1`将导入`import2`，所以当我们在Python里导入`import1`的时候，它就会导入并“执行”`import2`。

下面是`import1.py`的内容：

```
# import1.py
print 'loaded import1'
import import2
```

下面是`import2.py`的内容：

```
# import2.py
print 'loaded import2'
```

下面是从Python里导入`import1`时的输出：

```
>>> import import1
loaded import1
```

```
loaded import2
>>>
```

根据我们建议的先检查`__name__`值的迂回办法，我们对`import1.py`和`import2.py`中的代码做些修改，这样就不会发生这种情况了。

下面是修改后的`import1.py`的内容：

```
# import1.py
import import2
if __name__ == '__main__':
    print 'loaded import1'
```

下面是按同样办法对`import2.py`进行修改后的内容：

```
# import2.py
if __name__ == '__main__':
    print 'loaded import2'
```

现在再从Python导入`import1`时就不会看到原先的输出：

```
>>> import import1
>>>
```

不用说，你应该在任何情况下都这样编写自己的代码。也许会出现这样的情况，即你需要显示某些输出信息以确认确实导入了某个模块。这要看你自己所处的情形。我们的目的是通过实用性的程序设计例子来防止出现不需要的副作用。

14.4.2 `execfile()`

从上面的讨论可以看出，导入一个模块并不是从一个Python脚本程序执行另外一个Python脚本程序的好办法；要真是这样，这也就不是导入操作了。导入一个模块的副作用之一就是会引起顶层代码的执行。

我们在本章前面的内容里介绍过如何利用`exec`语句和一个文件对象做为参数来读入一个Python脚本程序的内容并执行它。这项工作可以用如下所示的代码段来完成：

```
f = open(filename, 'r')
exec f
f.close()
```

这三行代码可以被替换为一个`execfile()`调用：

```
execfile(filename)
```

虽然上面的代码确实会执行一个模块，但它是在它自己当前的执行环境（即它的全局名字空间和局部名字空间）中这样做的。有的时候，可能需要在另外一组不同的全局和局部字典而不是当前缺省的字典里执行一个模块。出于这个目的，我们可以使用`execfile()`内建函数，它的语法允许程序员指定名字空间，如下所示：

```
execfile(filename, globals = globals(), locals = locals())
```

14.5 执行其他非Python程序

我们可以在Python里执行非Python程序。这些非Python程序包括二进制可执行代码以及其他

shell脚本程序等。所有这一切都要求有一个合法的执行环境，也就是说，必须让它们拥有文件访问和执行的权限；shell脚本程序必须能够访问它们的解释器（Perl、bash，等等）；二进制代码必须是可访问的（同时还必须符合本地计算机的体系结构）。

最后，程序员还必须注意自己的Python脚本程序是否需要和将要执行的其他程序进行通信。有些程序需要有输入，其他程序会在运行完毕时返回一些输出或一个错误代码（也许两者都有）。根据不同的情况，Python准备了好几种执行非Python程序的办法。在这一小节介绍的函数都可以在os模块里找到。我们把它们集中起来放到表14-7里面去（如果某个函数只适用于某个特定的计算机平台，我们会做出相应的提示），并在本小节后面的内容里把它们介绍给大家。

表14-7 用于执行外部程序的os模块函数

os模块函数	说 明
<code>system(cmd)</code>	执行以字符串形式给出的程序cmd，等待程序的完成并返回退出代码（在Windows中，返回代码永远是0）
<code>fork()</code>	创建一个与父进程平行运行的子进程（通常和 <code>exec*()</code> 一起使用）；返回两次，一次是父进程的返回，一次是子进程的返回 # 只适用于UNIX
<code>execl(file, arg0, arg1...)</code>	以arg0、arg1等为参数执行file
<code>execv(file, arglist)</code>	相当于 <code>execl()</code> ，只是使用了参数列表（或表列）arglist
<code>execle(file, arg0, arg1..., env)</code>	相当于 <code>execl()</code> ，只是还提供了环境变量字典env
<code>execve(file, arglist, env)</code>	相当于 <code>execle()</code> ，只是使用了参数列表（或表列）arglist
<code>execlp(cmd, arg0, arg1...)</code>	相当于 <code>execl()</code> ，但要求在用户搜索路径中检索cmd完整的文件路径名
<code>execvp(cmd, arglist)</code>	相当于 <code>execlp()</code> ，只是使用了参数列表（或表列）arglist
<code>execvpe(cmd, arglist, env)</code>	相当于 <code>execvp()</code> ，只是还提供了环境变量字典env
<code>spawn*(mode, file, args[, env])</code>	根据mode的不同， <code>spawn*()</code> 函数可以复制 <code>fork()</code> 、 <code>exec*()</code> 、 <code>system()</code> 、 <code>wait*()</code> 函数的功能及这些函数的功能组合 # 只适用于Windows
<code>popen(cmd, mode = 'r', buffering = -1)</code>	执行cmd字符串，返回一个文件形式的对象作为通信句柄来运行程序。参数有缺失时mode会默认采用读模式；buffering会采用系统缺省的缓冲设置 # 只适用于UNIX
<code>wait()</code>	等待子进程的完成（通常和 <code>fork()</code> 和 <code>exec*()</code> 一起使用） # 只适用于UNIX
<code>waitpid(pid, options)</code>	等待指定子进程的完成（通常和 <code>fork()</code> 和 <code>exec*()</code> 一起使用） # 只适用于UNIX

随着我们逐步接近软件的操作系统层面，读者将会发现跨计算机平台执行程序（即使是Python脚本程序）的一致性也开始变得有些随机性了。我们刚刚讲过，在这一小节里介绍的函数都是os模块里的。真实的情况是：有好几个os模块。比如说，对应于UNIX的是posix模块；对应于Windows的是nt模块（不管你运行的是哪一个版本的Windows；DOS用户使用的是dos模块）；对应于Macintosh的是mac模块。但大家不必担心，当你调用“import os”时Python会自动加载上正确的模块。用户不必直接加载某个特定的操作系统模块。

14.5.1 os.system()

我们清单上的第一个函数就是system()，这是一个比较简单的函数，它以一个字符串名字的形式获得一个系统命令并执行它。Python自己的执行将在这个命令的执行期间被悬挂起来。当它的执行完成之后，其退出状态会以system()的返回值的形式给出来，Python的执行将继续进行。这个函数只对UNIX和Windows有效。

system()会保留包括标准输出在内的当前的标准文件，这就意味着执行有输出显示的任何程序或命令都会把它们的输出传送到标准输出去。请大家注意这一点，因为对某些应用程序如通用网关接口（common gateway interface, CGI）程序来说，如果通过标准输出回送给客户端的输出内容不是合法的超文本标签语言（hypertext markup language, HTML）字符串，就会引起Web浏览器出错。人们一般都在被调用命令不产生任何输出的场合才使用system()，这些命令包括文件压缩和文件转换程序、挂装磁盘到系统上，或者是其他一些用自己的退出状态代码而不是经由输入输出的通信来指示特定任务完成与否的程序。比较常见的做法是用退出状态代码0来表示成功；用非零整数来表示某种类型的失败。

为了让大家更清楚地理解这个函数，我们将执行两个命令。它们确实都有来自交互式解释器的程序输出，这样方便大家掌握system()的工作原理。

```
>>> import os
>>> result = os.system('cat /etc/motd')
Have a lot of fun...
>>> result
0
>>> result = os.system('uname -a')
Linux solo 2.2.13 #1 Mon Nov 8 15:08:22 CET 1999 i586 unknown
>>> result
0
```

请注意这两个命令的输出和保存在result变量中它们的执行退出状态代码。下面是一个执行一个DOS命令的例子；

```
>>> import os
>>> result = os.system('dir')

Volume in drive C has no label
Volume Serial Number is 43D1-6C8A
Directory of C:\WINDOWS\TEMP

.                <DIR>          01-08-98  8:39a .
..               <DIR>          01-08-98  8:39a ..
      0 file(s)                0 bytes
      2 dir(s)         572,588,032 bytes free
>>> result
0
```

14.5.2 os.popen() 只适用于UNIX和Windows

popen()函数是把一个文件对象和system()函数相结合的产物。它的工作情况和system()是一样的，但它多出这样一个功能：它能够与那个程序建立一个单向的连接然后对它进行访问，就

好象它是个文件一样。如果那个程序要求有输入，你就可以用'w'模式调用popen()对那个程序进行“写”操作。你发送给那个程序的数据会通过它的标准输入进行接收。同样地，'r'模式允许你接收一个命令的输出，当它向标准输出进行写操作的时候，你可以用自己熟悉的文件对象的read*()方法通过文件形式的句柄读到那些数据。并且，就象对待文件一样，当你完成操作时要记得用close()关闭这个连接。

在我们前面给出的一个system()示例中，我们调用了UNIX的uname程序来报告一些关于我们正在使用的计算机和操作系统的信息。这个命令产生的那行输出将直接送到屏幕上去。如果我们想把那个字符串读到一个变量里去进行一些内部处理或者把那个字符串保存到一个记录文件里去，就可以使用popen()。事实上，这段代码看起来就像是下面这个样子：

```
>>> import os
>>> f = os.popen('uname -a')
>>> data = f.readline()
>>> f.close()
>>> print data,
Linux solo 2.2.13 #1 Mon Nov 8 15:08:22 CET 1999 i586 unknown
```

正如你所看到的，popen()返回了一个文件形式的对象；另外请注意readline()象平常一样保留了输入文本每行后面的换行符。

14.5.3 os.fork(), os.exec*(), os.wait*() 只适用于UNIX

我们打算在这里详细介绍操作系统方面的理论，在本小节里只“轻微地”涉及到一些进程(process)方面的知识。fork()对用户单个的控制执行流(即人们常说的“进程”)进行处理，按照用户的意愿创建出一条“与原路线的岔路”。有趣的事情就发生了：用户系统现在是在两条路线上同时前进，也就是说这个函数调用的后果是使用户拥有两个并排运行的程序(不用说，运行的是同一个程序，这是因为两个进程都是从紧跟在fork()调用后面的那个代码行开始执行的。)

调用fork()的那个原始进程被称为“父”进程，而作为该调用的结果而创建出来的那个进程被称为“子”进程。当子进程返回的时候，它的返回值永远是零；而当父进程返回的时候，它的返回值永远是子进程的进程标识符(即进程ID，简称PID)；这样，父进程就可以监控它所有的子进程。而PID也是区分它们的唯一办法！

我们说过，两个进程都将从fork()调用的后面立刻开始执行。因为代码是一样的，所以如没有在此时采取其他措施，我们看到的将是完全一样的执行内容。这通常并不是我们想要的结果。创建另外一个进程的主要目的是为了运行另外一个程序，所以我们必须在父进程和子进程返回的时候采取分流措施。正如我们前面提到过的，它们的PID是不同的，而我们正是利用这一点把它们区分开来的。

有过进程管理方面经验的读者对下面这小段代码应该会比较熟悉的。但如果你是个新手，可能一开始会看不出它是如何工作的，可只要你认真体会，就会明白其中的奥妙。

```
ret = os.fork()           # spawn 2 processes, both return
if ret == 0:              # child returns with PID of 0
    child_suite            # child code
```

```

else:                                # parent returns with child's PID
    parent_suite                       # parent code

```

代码的第一行就是对fork()的调用。它创建出一个我们称之为子进程的新进程，而原来的进程被称为父进程。子进程对虚拟内存地址空间有一份它自己的拷贝，其中包含着它的父进程地址空间的精确复制——是的，两个进程几乎是一模一样的。fork()会返回两次，也就是说，子进程和父进程都会返回。你会问了：如果它们都会返回，那如何才能把它们区分开呢？当父进程返回的时候，它会带着子进程的PID；而当子进程返回的时候，它的返回值永远是0。我们就是通过这一点把二者区分开来的。

加上一个if-else语句后，我们就可以引导子进程（即if从句）和父进程（即else从句）去执行它们各自的代码。我们可以在为子进程准备的代码里对各种exec*()函数进行调用，让它去运行另外一个完全不同的程序或者去执行同一程序里另外一些函数（只要子进程和父进程还分别处于不同的执行岔路上，就可以一直这样做）。最普遍的做法是让子进程负责主要的工作，而父进程既可以耐心地等待子进程完成它的工作，也可以继续执行，过一会儿再去查看了进程是否已经正常结束了。

exec*()族函数加载一个文件或命令，加上必要的参数（可以单独给出，也可以放在一个参数列表里）后执行它。如有必要，还可以为那个命令准备一个环境变量字典。环境变量的作用是把用户的当前执行环境向程序做一个更精确的描述，程序可以通过访问这些变量了解自身所处的执行环境。大家经常见到的环境变量包括用户名、搜索路径、当前shell、终端类型、本地语言、计算机型号、操作系统名字等。

所有exec*()族的函数都会用给定文件作为即将执行的程序替代掉运行在当前（子）进程中的Python解释器。和system()不一样的地方是它不再返回到Python（因为Python已经被替换掉了）。如果因为某种原因使程序无法执行，就会使exec*()执行失败，进而引发一个例外。

下面的代码在子进程里启动了一个有趣的名为“xbill”的小游戏，而父进程则继续运行着Python解释器。因为子进程永远也不会返回，所以不必替子进程操心在调用exec*()之后还需不需要执行别的代码。需要大家注意的是这个命令本身也要做为参数列表中的第一个参数。

```

ret = os.fork()

if ret == 0:                          # child code
    execvp('xbill', ['xbill'])

else:                                  # parent code
    os.wait()

```

在这段代码里，读者还可以看到一个wait()调用。在子进程的工作完成时，需要它们的父进程来打扫战场。这项工作被称为“收获一个子进程”，可以用wait*()函数来完成。紧随着一个fork()调用，父进程可以等待子进程的结束并完成一些打扫战场的工作。父进程也可以继续自己的执行，过一会儿再用某个wait*()族的函数去收获那个子进程。

不管父进程选用的是哪一个方法，这项工作是必不可少的。如果一个子进程已经结束了自己的执行但还没有被收获，它就会进入一个什么也不干的状态，成为一个“哑”进程。我们建议大家尽量把自己系统中的哑进程数目控制为最少，因为处在这种状态里的子进程还全部占用

着在其运行期间分配给它们的系统资源，这些系统资源在它们被父进程收获之前是不会被释放的。

一个wait()调用会把执行悬挂起来（进入等待状态）直到一个子进程（任意一个子进程）结束——不管它是正常结束还是被某个信号终结的。子进程结束后，wait()会跟上去收获它，释放所有的资源。如果这个子进程已经完成了自己的工作任务，就只需wait()去完成收获过程。waitpid()和wait()在功能上是一样的，但它多出两个参数来：一个是PID，它给出的是某个特定的需要等待的子进程的进程标识符；另一个是options，它通常是零或者一组可选操作标志的逻辑OR（或操作）组合。更深入的细节请参考Python、你的操作系统的有关文档，或者任何操作系统方面的教科书——比如Sibershatz和Galvin、Tanenbaum和Woodhull或Stallings等人的著作。

14.5.4 os.spawn*()只适用于Windows

spawn*()族函数只对Windows起作用。根据选择的模式的不同，spawn*()函数可以复制fork()、exec*()、system()、wait*()等函数的功能及这些UNIX流行函数的功能组合。spawn()和spawnnve()都是从Python 1.5.2版本开始出现的新函数。进一步的资料请参考《Python Library Reference》(Python库函数大全)一书。 [1.5.2]

14.5.5 其他函数

表14-8列出了一些能够完成有关任务的函数（及其模块）。

表14-8 与文件执行有关的各种函数

文件对象属性	说 明
popen2.popen2()	执行一个文件并对这个新创建的运行文件进行读/写操作，用户把它要求的输入数据写到它的标准输入（stdin），从它的标准输出（stdout）读取它的输出数据 # 只适用于UNIX
popen2.popen3()	执行一个文件并对这个新创建的运行文件进行读/写操作，用户把它要求的输入数据写到它的标准输入（stdin），从它的标准输出和标准错误（stdout和stderr）读取它的输出数据 # 只适用于UNIX
commands.getoutput()	在一个子进程里执行一个文件，把所有输出返回为一个字符串 # 只适用于UNIX

14.6 受限执行环境

本书通篇使用的都是普通的非受限执行环境，非受限执行环境允许用户全面访问那些在Python解释器里可用的资源。其中包括（但不限于）磁盘文件和数据库的访问、建立网络连接、调用其他程序等。

但有的时候，你也许会考虑给在你的系统中运行的Python程序加上一些限制。需要你加上部分或者全部限制的情形包括：Python中的CGI应用程序、允许上传和执行Python脚本程序的环境、把Python解释器安装在/bin目录时的匿名FTP访问等等。

有两个基本模块可以帮助设置受限环境。头一个是Bastion模块，它提供了对读者数据的访问限制。利用Bastion模块的基本办法是实例化那个对你的对象进行了打包的Bastion类，它提供了一个属性过滤器，使程序只能有选择地访问你的对象的属性。

本书不讨论Bastion模块，读者可以在Python文档或者Beazley的著作里找到进一步的资料。我们将把注意力集中到rexec模块上来，它是用来建立受限环境的，你可以把信任度不高的Python代码放到受限环境里去执行。关于Bastion模块我们再多说一句：你可以把它和rexec模块结合在一起使用，提供一个既完整又安全的执行机制，同时对数据访问和代码运行环境加以限制。（这两个模块会做为标准库的一部分随Python一起安装。）

rexec模块的基本任务是限制一个Python脚本程序的执行环境。这个模块只允许使用一定数目的内建成员（函数属性和/或数据属性）；可以限制哪些模块能够被导入；可以设置sys模块和os模块中的哪些属性允许或者不允许被访问；同时还对一些关键性的内建属性（比如open()、reload()和__import__()等）用限制规则进行打包。

读者应该记得__builtins__模块是由__builtin__模块中的所有属性组成的。如果模块__builtins__就是__builtin__模块，得到的就是一个未加限制的环境，请看：

```
>>>__builtins__
<module '__builtin__' (builtin)>
```

当我们对执行环境加以限制的时候，事实上的__builtins__就会是__builtin__模块专门为受限环境挑选出来的一个子集，甚至会在该环境中变成“不可访问的”，如下所示：

```
>>>__builtins__
<module '?' (builtin)>
```

rexec模块实现了一个RExec类，你可以从中推导子类并建立自己的受限环境。在这个类里面有一些静态成员，对它们进行覆盖就等于设定了哪些能哪些不能在这个“笼子”一样的环境里使用。我们把RExec类里的静态数据属性列在表14-9里。

表14-9 RExec类的属性

属性名字	说 明
not_builtin_names	不适合包括在__builtins__中的属性
ok_builtin_modules	可以被导入的模块
ok_path	可以在受限环境中被访问的目录清单
ok_posix_names	适合从os模块导入的属性
ok_sys_names	适合从sys模块导入的属性

RExec子类的实例有几个执行受限代码的方法，我们把它列在表14-10里。

表14-10 RExec类的方法

方法名称	说 明
<code>r_eval()</code>	<code>eval()</code> 的受限版本
<code>r_exc_info()</code>	<code>sys.exc_info()</code> 的受限版本
<code>r_exec()</code>	<code>exec</code> 的受限版本
<code>r_execfile()</code>	<code>execfile()</code> 的受限版本
<code>r_import()</code>	<code>__import__()</code> 的受限版本
<code>r_open()</code>	<code>open()</code> 的受限版本
<code>r_reload()</code>	<code>reload()</code> 的受限版本
<code>r_unload()</code>	<code>del</code> 模块的受限版本

`r_*()`族中的所有方法(除`r_exc_info()`和`r_open()`以外)都有与之对应的`s_*()`方法,后者和前者的行为完全一样,只是多了能够对标准文件(标准输入、标准输出和标准错误)进行访问。在`rexec`模块里还有其他一些方法,我们建议大家去Python文档里查阅更详细的资料。

我们在下面给出了一个由两个文件组成的例子。`cager.py`是负责建立一个能够安全执行另外一个脚本程序`caged.py`的受限环境的程序。这个例子里采用的限制规则是这样的:除了少数几个以外,所有内建属性都不允许在我们的受限环境里访问。为了做到这一点,我们覆盖了`nok_builtin_names`属性,这个表列里面列出的是“不”允许在受限环境里使用的属性。

`cager.py`的代码见程序示例14-2。`caged.py`的代码见程序示例14-3。

程序示例14-2 创建一个受限环境(`cager.py`)

```

1  #!/usr/bin/env python
2
3  import rexec
4
5  class YourSandbox(rexec.RExec):
6      nok_builtin_names = dir(__builtins__)
7      nok_builtin_names.remove('dir')
8      nok_builtin_names.remove('str')
9      nok_builtin_names.remove('vars')
10
11  r = YourSandbox()
12  r.r_execfile("caged.py")

```

程序示例14-3 在一个受限环境中执行程序(`caged.py`)

```

1  #!/usr/bin/env python
2
3  print 'Restricted to these built-in attributes:'
4  for eachBI in dir(__builtin__):
5      print '\t', each BI:
6  print '\nAll others inaccessible, i.e. eval():\n'
7  eval(123)

```

请看`caged.py`执行时给出的输出,它先列出了一个自己确实有权使用的内建属性清单(请把这个清单与`cager.py`中的代码进行比较),然后是调用一个无权使用的函数时(例子里用的是`eval()`)会出现的出错情况:

```

% cager.py
Restricted to these built-in attributes:

```

```

__builtin__
__import__
dir
open
reload
str
vars
All others inaccessible, i.e. eval():

Traceback (most recent call last):
  File "cager.py", line 12, in ?
    r.r_execfile("caged.py")
  File "/usr/lib/python2.0/rexec.py", line 261, in r_execfile
    return execfile(file, m.__dict__)
  File "caged.py", line 7, in ?
    eval(123)
NameError: There is no variable named 'eval'

```

`r_*`()族函数如`r_open()`等（以及它们的`s_*`()对应函数）会自动把所有的调用导向遵守附加限制来执行这些函数的特殊的打包器去。比如说，一个`open()`调用将调用`r_open()`，而后者只允许读模式。如果试图用`open()`打开一个文件进行写操作，就会引发一个如下所示的`IOError`例外：

```
IOError : can't open files for writing in restricted mode
```

如果你覆盖掉`r_open()`，甚至可以连读操作都不允许。我们先把下面的方法定义加到脚本程序`cager.py`中`YourSandbox`类的定义部分里去：

```
def r_open(f, m = 'r', b = -1) :
    raise IOError, ' sorry, no file access period '
```

现在，当我们想打开一个文件进行读或写操作时，就会看到：

```
IOError : sorry, no file access period
```

14.7 中断程序的执行

如果一个程序模块中的顶层代码都正常地执行完毕，然后正常地退出，即该程序运行到工作完成，我们就称这次执行是“干净的”。但有的时候，比如出现了一个严重错误时，就需要在执行完成之前退出Python。另外的情形是继续执行的条件不充分等情况。

在Python语言里，对错误的响应办法可以说是多种多样，一种办法是通过例外和例外处理来解决问题。另外一个办法是以一种“更干净”的方式来执行代码，即把代码的主程序部分放在一个if语句里，只有在没有出错条件的情况下才执行它。也就是说，即使遇到出错条件，也让代码“正常”地终止运行。但有时需要返回到调用者程序里去，用一个出错代码指出发生了那样的状况。

14.7.1 sys.exit()和SystemExit

立刻退出一个程序并返回到调用者程序里去的基本方法是使用`sys`模块里面的`exit()`函数。`sys.exit()`的语法如下所示：

```
sys.exit(status=0)
```

调用`sys.exit()`的时候会引发一个`SystemExit`例外。如果用户没有（在一个`try`语句里用一个适当的`except`从句）对它进行监控，这个例外一般不会被捕获也不会被处理，而解释器会立刻带着给定的执行状态参数`status`退出；这个执行状态参数`status`在没有给出的情况下以0为缺省值。`SystemExit`是唯一不被看做是关于错误的例外。它只表达想退出Python的愿望。

`sys.exit()`经常用在调用命令的中途发现出现了一个错误的时候；比如参数不正确、不合法或者参数的个数不对等情况。下面的程序示例14-4（`args.py`）是一个测试用脚本程序，只有在给出特定个数的参数之后它才能继续正确执行。

程序示例14-4 立刻退出（`args.py`）

调用`sys.exit()`会使Python解释器立刻退出。`exit()`的任何整数参数都会做为退出时的执行状态代码返回给调用者，它的缺省值是0。

```
1  #!/usr/bin/env python
2
3  import sys
4
5  def usage():
6      print 'At least 2 arguments (incl. cmd name).'
7      print usage: args.py arg1 arg2 {arg3... }'
8      sys.exit(1)
9
10 argc = len(sys.argv)
11 if argc < 3:
12     usage()
13 print "number of args entered:", argc
14 print "args (incl. cmd name) were:", sys.argv
```

执行这个脚本程序，我们得到如下所示的输出：

```
% args.py
At least 2 arguments required (incl. cmd name).
usage: args.py arg1 arg2 {arg3... }

% args.py XXX
At least 2 arguments required (incl. cmd name).
usage: args.py arg1 arg2 {arg3... }

% args.py 123 abc
number of args entered: 3
args (incl. cmd name) were: ['args.py', '123', 'abc']

% args.py -x -2 foo bar
number of args entered: 5
args (incl. cmd name) were: ['args.py', '-x', '-2', 'foo',
'bar']
```

许多命令行驱动的程序都会在开始执行其核心功能之前对输入数据的合法性进行检查。如果合法性检查在某一点上失败了，就会调用一个`usage()`函数告诉用户什么问题引起了那个错误，并显示一个帮助性的用法“提示”，这样用户下次就可以正确地调用这个脚本程序了。

14.7.2 `sys.exitfunc()`

`sys.exitfunc()`在缺省的情况下是不可用的，但可以被覆盖。在被覆盖之后，它就可以在调用`sys.exit()`时在解释器退出之前提供一些额外的处理功能。不需要向这个函数传递任何参数，所以

读者在自己创建这个函数时不要给它加上参数。

根据Beazley的说法，如果`sys.exitfunc()`已经被一个以前定义的退出函数覆盖掉了，那么最好还是把那段代码做为你的退出函数的一部分。通常，退出函数被用来完成一些关机之类的动作，比如关闭文件或网络连接等；而且最好再完成一些系统维护方面的工作，比如释放先前占用的系统资源等。

下面是一个如何设置一个退出函数的例子，如果已经设置了另外一个，千万记得要把那个也执行了。请看：

```
import sys

prev_exit_func = getattr(sys, 'exitfunc', None)

def my_exit_func(old_exit = prev_exit_func):
    #      :
    # perform cleanup
    #      :
    if old_exit != None and callable(old_exit):
        old_exit()

sys.exitfunc = my_exit_func
```

在我们的扫尾工作完成之后，我们执行了以前的退出函数。`getattr()`调用的作用是检查以前是否定义过一个`exitfunc()`函数。如果没有，就把`None`赋值给`prev_exit_func`变量；否则`prev_exit_func`就会成为原有退出函数一个新的别名，这个新的别名将做为一个缺省参数被传递到我们新的退出函数`my_exit_func`去。

对`getattr()`的调用可以重新写成如下所示的样子：

```
if hasattr(sys, 'exitfunc'):
    prev_exit_func = sys.exitfunc    # getattr(sys, 'exitfunc')
else:
    prev_exit_func = None
```

14.7.3 `os._exit()`函数

`os`模块的`_exit()`函数一般不用在平常的程序设计中（它依赖于计算机平台，只在某些特定的计算机平台上才有（比如UNIX和Windows））。它的语法如下所示：

```
os._exit(status)
```

这个函数提供的功能正好与`sys.exit()`和`sys.exitfunc()`相反，它不进行任何（Python或者程序员定义的）扫尾工作，立刻退出Python。与`sys.exit()`不同的是，它的`status`参数是必不可少的。通过`sys.exit()`退出解释器是更好的办法。

14.8 相关模块

表14-11是一个没有包括`os`和`sys`模块的模块清单，这些模块都与我们本章对执行环境的论题有关。

表14-11 与执行环境有关的模块

模 块	说 明
popen2	在os.popen()的顶层提供一些额外的功能。提供了通过标准文件与其他进程进行通信的能力
commands	在os.system()的顶层提供一些额外的功能。把程序的所有输出保存在一个返回的字符串里面（与把输出内容全部显示到屏幕上的做法相反）
getopt	处理应用程序的可选项和命令行参数
site	对与站点有关的模块和软件包进行处理
findertools	提供了一个到Macintosh机器的finder功能的操作接口，比如启动一个应用程序等（或文档，即启动与文档对应的应用程序对它进行处理）

14.9 练习

14-1 可调用对象。请说出Python语言中的可调用对象。

14-2 exec和eval()的比较。exec语句和eval()内建函数有什么区别？

14-3 input()和raw_input()的比较。内建函数input()和raw_input()之间有什么区别？

14-4 执行环境。编写一个运行其他Python脚本程序的Python脚本程序。

14-5 os.system()函数。请选择一个比较熟悉的系统命令完成一项工作，这个命令不要求有输入，但可以输出到屏幕或根本没有输出。请使用os.system()调用来运行这个程序。

14-6 commands.getoutput()函数。用commands.getoutput()函数完成上一个练习。

14-7 popen()族函数。选择一个熟悉的系统命令，它应该从标准输入获取文本，进行处理或者输出数据。请使用os.popen()函数与这个程序进行通信。它的输出到哪里去了？试试用popen2.popen2()来代替它。

14-8 受限执行。建立一个受限环境，然后显示你的__builtins__模块的内容来确认它。

14-9 退出函数。设计一个你的程序退出时将调用的函数。把它安装为sys.exitfunc()，运行你的程序，确认这个退出函数确实被调用了。

14-10 shell。编写一个shell（即操作系统操作界面）程序：提供一个用来接收操作系统命令去执行的命令行操作界面。

附加题 1：支持管道（请参考os模块中的dup()、dup2()和pipe()函数）。这个管道过程允许一个进程的标准输出连接到另外一个进程的标准输入。

附加题 2：支持使用尖括号的逆向管道，让你的shell有一个函数化程序设计风格的界面。

第二部分 高级论题

第15章 规则表达式

15.1 介绍与动机

对文本和数据进行处理是一件大事情。如果你不相信我说的话，就请仔细看看现如今的计算机每天都在做些什么工作。字处理、“填表”、Web主页，来自数据库的信息流，新闻输入——这个清单还可以继续列下去。因为我们可能不知道我们编程让计算机处理的到底是什么样的文本和数据，所以如果能够把那些文字或者数据用某种计算机能够识别和采取行动的表达式表达出来是非常有必要的。

假设我经营着一家电子邮件（e-mail）档案服务公司，而你是我的一位顾客。那么，当你想要查看自己在去年二月收发的全部电子邮件时，如果我有一个计算机程序能够自动收集整理并把这些信息发送给你而不是让一个人通读你的电子邮件后再人工处理你的要求，那该有多好啊。我想，如果有人要看遍你的邮件，即使是那个人的眼睛只是扫过收发邮件的时间，你恐怕都不会答应。另一个例子是要求检查邮件是否有“I LOVE YOU”这类的标题行，这个标题表示邮件已经被病毒感染，需要把这些电子邮件消息从你的私人邮件档案里清除出去。这就引出一个问题，即怎样编程才能让计算机具备在文本里检索这类模式的能力。

规则表达式（regular expressions, RE）就提供了这样的内在结构，它能够对文本模式实现先进的匹配、抽取以及搜索与替换功能。RE简单说来不过是一些字符串，但这些字符串里会有一些特殊的符号和字符。这些特殊的符号和字符或者指示一段文字的重复出现，或者可以代表多个字符以“匹配”一组具有共同点的字符串——而这些“共同点”就是模式所要描述的东西（如图15-1所示）。换句话说，它们能够匹配多个字符串——如果一个RE模式只能匹配一个字符串，那可是既烦人又低能的事，你不这么看吗？

Python通过标准库的re模块支持RE。我们希望这个简介性质的小节确实能给你一个简单而又清晰的概念。限于篇幅，这一小节的讨论范围也只能限制为日常Python程序设计中常用的RE特点方面。每个人的经验当然是不一样的。我们建议大家去阅读一些正式的支持文档和关于这个主题的其他书籍。你对字符串的看法肯定不会是从前那样了！

编程提示：搜索和匹配

[CN]

这一章里到处都是“搜索”和“匹配”。在讨论规则表达式并涉及到字符串模式的时候，“匹配”的严格定义指的是模式匹配。在Python术语里，完成模式匹配有两个主要的办法：一是搜索，在字符串的任何位置查找一个模式的匹配；二是匹配，试图把一个模式匹配到

一个完整的字符串去（从字符串头部开始）。搜索是用`search()`函数或方法完成的，而匹配则是用`match()`函数或方法完成的。总之，在说到模式时我们保留“匹配”的广义含义；把“搜索”和“匹配”之间的差异限定在Python完成模式匹配的办法方面。



图15-1 你可以使用规则表达式，比如图中的这个来识别合法的Python标识符。

“`[A-Za-z]\w+`”表示第一个字符必须是字母A-Z或a-z，后面至少跟着一个（+）字母或数字字符（`\w`）。在我们的过滤器里，请注意有多少字符串进入了过滤器，但能通过的只有那些符合我们过滤器过滤标准的字符串

你的第一个规则表达式

正如我们前面提到过的，规则表达式是一个由文本和特殊字符构成的字符串，这个字符串描述的是一个用来识别多个字符串的模板。我们还简要介绍过一个适用于普通文本的规则表达式字母表，规则表达式使用的字符集包括全体大、小写字母和数字。有时也会用到特别一点的字符集，比如只包含字符“0”和“1”的情况——建立在这个字符集基础上的字符串描述的是

各种二进制数的字符串表示形式，如“0”、“1”、“00”、“01”、“10”、“11”、“100”等。

虽然规则表达式经常被视为一个“高级话题”，但它们在本质上也很简单。我们下面来看一些最基本的规则表达式。我们通过普通文本用的标准字母表写出几个简单的规则表达式，同时给出这些模板所描述的字符串。下面这些规则表达式都是最基本的，一点修饰都没有。它们是一些只能匹配一个字符串的字符串模板。我们把规则表达式放在前面，后面是它们匹配的字符串：

规则表达式模板	匹配的字符串
foo	foo
python	python
abc123	abc123

上表中的第一个规则表达式模板是“foo”。除这几个字符之外，这个模板没有使用任何用来匹配其他字符的特殊符号，所以这个模板唯一能够匹配的字符串就是字符串“foo”。“Python”和“abc123”的情况也都是如此。规则表达式的威力要靠特殊字符来体现，用它们对字符集、模板分组和模板重复等进行定义。正是这些特殊符号使RE不仅能够匹配单个字符，还能够匹配一组字符。

15.2 规则表达式使用的特殊符号和字符

我们现在向大家介绍一些以后经常会用到的特殊字符和符号——它们的学名是“元字符”，正是它们使规则表达式具有了强大的功能和灵活性。表15-1列出了其中最常用的符号和字符。

表15-1 常用规则表达式符号和特殊字符

记号	说 明	举 例
符 号		
re_string	匹配文字性字符串值re_string	foo
re1/re2	匹配文字性字符串值re1或re2	foolbar
.	匹配任意单个的字符（不包括换行符）	:::++:
^	对字符串的头部进行匹配	^Dcar
\$	对字符串的尾部进行匹配	/bin/^W*sh\$
*	此前规则表达式定义的情况可以匹配0次（即不出现）或多次	{A-Za-z}^w*
+	此前规则表达式定义的情况至少匹配1次，可以匹配多次	^d+\\.\\.\\.^d-
?	此前规则表达式定义的情况最多匹配1次，可以匹配0次（即不出现）	goo?
{N}	此前规则表达式定义的情况必须匹配N次	^d{3}
{M,N}	此前规则表达式定义的情况必须匹配M到N次	^d{5,9}
[...]	匹配此字符集中任意一个单个的字符	[aeiou]
[...x-y...]	匹配字符范围x到y中任意一个单个的字符	[0-9], [A-Za-z]
[^...]	不匹配此字符集中任意一个单个的字符，包括字符范围 （如果此字符集里有的话）	[^aeiou], [^A-Za-z0-9_]
(* + ? { }) ?	让此前的多次匹配和重复匹配符号（*、+、?、{ }）尽量匹配最少的字符	.*?^w
(...)	匹配括号中的规则表达式，并保存为一个子组	{^d(3)}?, f(oo u)bar

(续)

记 号	说 明	举 例
特殊字符		
\d	匹配任意数字, 相当于[0-9] (\D的作用与\d正好相反, 表示不匹配任何一个数字)	data\d+.txt
\w	匹配任意字母和数字, 相当于[A-Za-z0-9_] (\W的作用与\w正好相反)	[A-Za-z_]\w+
\s	匹配任意空格字符, 相当于[\n\t\r\v\f] (\S的作用与\s正好相反)	of\sthe
\b	匹配任何单词边界 (\B的作用与\b正好相反)	\bThe\b
\nn	匹配第nn个保存起来的子组 (请参考上面的(...))	price:\16
\c	匹配特殊字符的原义(即不使用它的特殊意义, 而把它动作一个普通的字符)	\\, \\\, *
\A(\Z)	匹配字符串的头部(尾部) (请参考上面的^和\$)	\ADear

1. 用垂直线 (|) 匹配多个规则表达式模板

垂直字符 (|) 表示多选一操作, 它的作用是选择使用被垂直线字符分隔的多个规则表达式中的一个。下面的例子是多选一情况的几个模板示例和它们匹配的字符串:

规则表达式模板	匹配的字符串
at home	at, home
r2d2 c3po	r2d2, c3po
bat bet bit	bat, bet, bit

有了这个符号, 规则表达式的灵活性增加了, 使它能够匹配不止一个字符串。“多选一”有时也叫做“联合”或“逻辑或”。

2. 匹配任意一个单个的字符 (.)

点字符或句号 (.) 用来匹配任意一个单个字符, 不包括换行符在内 (Python的规则表达式有一个编译标志[S或DOTALL]将取消这一限制, 包括换行符)。这个点字符可以匹配以下各种字符: 字母、数字、不包括“\n”的空格、可打印字符、非打印字符或者一个符号等。请看下面的例子:

规则表达式模板	匹配的字符串
f.o	“f”和“o”之间的任何字符, 如: fao、f9o、f#o等。
..	任意两个字符。
.end	字符串end前面的任意一个字符。

问: “如果我想匹配点字符或句号怎么办?”

答: 为了明确地匹配一个点字符, 你必须用一个反斜线即“\”对它的功能进行转义。

3. 从字符串的开头或结尾或单词边界开始匹配 (^/\$)

有几个符号和特殊字符是用来指定从字符串的开头或结尾对规则表达式模板开始搜索的。如果想从字符串的头部开始匹配一个模板, 你必须使用上箭头 (^) 或特殊字符\A (大写字母“A”前面加上一个反斜线); 后者是为那些没有上箭头符号的键盘准备的, 比如国际键盘。类似地, 美元符号 (\$) 或\Z用来从字符串的尾部开始对模板进行匹配。

使用了这些符号的模板与我们将在本章介绍的大多数其他字符是不一样的, 因为这些符号指定的是位置。在上面的编程提示里, 我们曾经说明过“匹配”是从头开始匹配整个字符串,

而“搜索”则可以从字符串里的任意位置开始匹配。因为这几个字符是专门用来处理与位置有关的情况，所以只有在与搜索一起使用时才有意义。

下面是几个“打擦边球”的规则表达式搜索模板：

规则表达式模板	匹配的字符串
<code>^From</code>	任何以From开始的字符串。
<code>/bin/tcsh\$</code>	任何以/bin/tcsh结束的字符串。
<code>^Subject: his</code>	由且仅由Subject: his组成的字符串

如果你想匹配这两个字符本身，就一定使用反斜线进行转义。比如说，如果你想匹配以美元符号结尾的字符串，就要使用“`.*\$$`”这样的规则表达式来解决问题。

`\b`和`\B`特殊字符用来匹配空字符串，这就意味着它们可以从任意位置开始进行匹配。两者的不同之处在于`\b`匹配的模板是一个单词边界，也就是说，与之对应的模板一定是在某个单词的开头，不管它前面有字符（比如说字符串中间的单词）还是没有字符（一行开始处的单词）。同样地，`\B`只匹配出现在字符串里某个单词中的模板（也就是说不在单词边界上的字符串）。下面是几个例子：

规则表达式模板	匹配的字符串
<code>the</code>	任何包含有the的字符串
<code>\bthe</code>	任何以the开始的字符串
<code>\bthe\b</code>	只匹配单词the
<code>\Bthe</code>	任何包含the但不以它开始的单词

4. 建立字符集合（[]）

点符号可以用来匹配任意一个单个的符号，但某些时候可能只需要匹配几个特定的字符。方括号（[]）就是为这个目的发明出来的。使用了方括号的规则表达式将匹配方括号中任何一个字符。下面是几个例子：

规则表达式模板	匹配的字符串
<code>b[aeiou]t</code>	bat, bet, bit, but
<code>[cr][23][dp][o2]</code>	开始是“r”或“c”，然后是“2”或“3”，再接着“d”和“p”，最后是“o”和“2”。比如c2do, r3p2, r2d2, c3po等。

对规则表达式“`[cr][23][dp][o2]`”有一点需要说明——如果只想让“r2d2”或“c3po”成为能够通过规则表达式的字符串，需要使用比上面例子里更有限制性的规则表达式才行；但因为方括号隐含的是“逻辑或”功能，所以使用方括号是达不到这一目标的。唯一的解决办法是使用“`r2d2lc3po`”。

如果准备匹配的是单个的字符，那多选一和方括号的效果是一样的。举个例子，我们从规则表达式“`ab`”开始，它只能匹配以“a”开头后面再跟一个“b”的字符串。如果我们只想要的是一个单字符的字符串，即“a”或者“b”，就可以使用下面这个规则表达式：“`[ab]`”。又因为“a”和“b”是不同的字符串，所以我们可以选择规则表达式“`alb`”。但如果我们想用模板“ab”或“cd”去匹配字符串，就不能再使用方括号了，因为刚才所说的做法只适用于单个字符的情况。此时唯一的正解是“`ablcd`”，它与我们前面讲过的那个“`r2d2lc3po`”例子道理是

一样的。

5. 表示范围 (-) 和否定 (^)

方括号除支持单个字符外,还支持字符范围。在方括号中,如果在两个符号之间加上一个连字符,就表示它是一个字符范围,比如A-Z、a-z或0-9等就分别对应于大写字母、小写字母和十进制数字。这是相对于字符集而言的一个范围,所以它不仅限于字母表中的字母和普通十进制数字。此外,如果上箭头符号 (^) 是方括号开始后的第一个字符,就表示“不”匹配给定字符集里的字符。请看下面的例子:

规则表达式模板	匹配的字符串
z.[0-9]	“z”后面跟一个任意的字符,然后是一个数字。
[r-u]env-y[us]	“r”、“s”、“t”或“u”后面跟一个“e”、“n”、“v”、“w”、“x”或“y”,再后面是“u”或“s”。
[^aeiou]*	零个以上(*星号将在下一小节介绍)的非元音。(练习:为什么我们说“非元音”而不说“辅音”?)
[^\\t\\n]+	从第一个TAB制表符或换行符开始,但不包括它们在内的一个以上的字符。
[^a]	在一个使用ASCII字符集的系统上,所有顺序值在“”和“a”之间的字符,即顺序号为34到97之间的字符。

6. 使用封闭操作符 (*、+、?、{ }) 的重复匹配

我们现在开始介绍最常用的规则表达式记号,即特殊符号“*”、“+”和“?”,这几个符号都可以用来匹配字符串模板出现一次、多次以及不出现的情况。星号操作符(*)匹配的是它紧左边的那个规则表达式出现零次或零次以上(即允许字符串模板不出现)的情况(在计算机语言和编译器理论里,这个操作符被称为Kleene封闭操作符)。加号(+)匹配一个规则表达式至少出现一次的情况(它也叫做正封闭操作符),而问号(?)则精确地匹配一个规则表达式出现零次或一次的情况。

另外还有一个花括号操作符({}),花括号里面可以是一个单个的值,也可以是用逗号分开的两个值。如果是一个值(即{N})的情况,就表示将匹配N次出现;如果是一个范围(即{M,N})的情况,就表示匹配M次到N次。这些符号可以用反斜线符号进行转义,也就是说,“*”将匹配星号字符本身。

下面是一些使用了封闭操作符的例子:

规则表达式模板	匹配的字符串
[dn]ot r	“d”或“n”,然后是一个“o”,最后是最多一个“t”,即do、no、dot和not。
0?[0-9]	任何一位数字,也许前面还有一个0。我们可以把它看做是一月到九月的数字形式,不管是一位数字还是两位数字都没问题。
[0-9]{15,16}	15到16个数字,比如信用卡号码
</?[^>]+>	能够匹配一切合法HTML标记的字符串。
[KQRBNP][a-h]{1-8}-[a-h]{1-8}	以“长埃尔吉布莱克”记号法表示的国际象棋棋步(只有移动,不

包括吃了和将军), 即“K”、“Q”、“R”、“B”、“N”或“P”等字母后面加上两个用连字符连在一起的“a1”到“h8”(之间)的棋盘格编号。前面的编号表示从哪个格开始走, 后面的编号表示走到哪一格去。

7. 代表字符集的特殊字符

我们还提到过有些特殊字符可以用来代表字符集合。比如说, 你可以不使用“0-9”这个范围来表示十进制数字, 只要简单地使用“\d”就可以匹配任何一个十进制数字了。特殊字符“\w”代表整个字母数字集合, 即相当于“A-Za-z0-9_”的简写形式; 而特殊字符“\s”用来代表任意空格符。这些字符串的大写形式代表的意义是“不匹配”, 比如说, “\D”匹配的是任何非十进制数字的字符(即相当于“[^0-9]”), 等等。

下面是几个使用了这种简写形式的例子:

规则表达式模板	匹配的字符串
<code>\w+-\d+</code>	用连字符连在一起的字母数字字符串和至少一个数字。
<code>[A-Za-z]\w*</code>	第一个字符是字母, 其余字符(如果有的话)是字母或数字(它几乎相当于Python语言中合法标识符的集合[请参考练习部分的内容])。
<code>\d{3}-\d{3}-\d{4}</code>	(美国)电话号码, 前面有一个长途区号, 比如: 800-555-1212。
<code>\w+@\w+\.</code>	简单的XXX@YYY.com形式的电子邮件地址。

需要注意的是: 所有这些特殊字符, 包括我们前面已经介绍过的“\A”、“\B”、“\d”等, 可能有也可能没有对应的ASCII字符。为了确保你所使用的是规则表达式所定义的字符含义, 最好的办法是用生字符串取消反斜线符号的功能(请参考本章后面内容里的编程提示)。

此外, “\w”和“\W”字母数字集合会受到从Python 1.6版本开始引入的Unicode编译标志“L”或“LOCALE”的影响。

8. 用圆括号(())组建字符集合

现在, 我们已经基本达到匹配一个字符串和剔除不匹配情况的目的了。但有的时候, 我们可能更关心我们匹配到的数据本身。我们不仅需要知道是否整个字符串都匹配上了我们的规则表达式, 还要在匹配成功时需要取出某个特定的字符串或子字符串。想这样做没问题, 具体办法是给一个规则表达式加上一对圆括号。

一对圆括号(())和规则表达式一起使用时可以实现两种功能:

- 1) 对规则表达式进行分组
- 2) 匹配子组

需要对规则表达式进行分组的情况有很多, 一个比较好的例子是你需要用两个不同的规则表达式去比较一个字符串。另外一个理由是准备给整个规则表达式加上一个重复操作符(即不仅仅是匹配单个的字符或单个字符的集合)。

使用圆括号还有一个附加效果, 就是匹配上模板的子字符串会被保存起来供今后使用。这些“子组”可以在同一次搜索或匹配操作中反复调用, 或者被取出来做进一步的处理。为什么子组的匹配很重要呢? 主要原因就是除进行匹配操作以外, 你还想取出匹配得到的字符串模板。

比如说, 如果我们决定使用“\w+-\d+”进行匹配操作, 但又想把第一个的字母部分和第二

个的数字部分分别保存，那该怎样做呢？这是经常会遇到的情况，因为我们想知道到底是什么样的字符串匹配上了我们的规则表达式模板。如果我们给两个子模板分别加上圆括号，即让它成为“`(\w+)-(\d+)`”的样子，我们就可以对这两个匹配到的子组分别进行访问了。你当然也可以先写一段用来检查是否有匹配的代码，再另外执行一段（也要我们自己编写）对整个匹配进行分析以分别取出这两个部分的代码来；两相比较，当然是使用对规则表达式进行分组的办法更好了。既然Python已经在它的re模块里支持了这种功能，为什么不让Python去完成这个工作而后再搞什么“新发明”呢？

规则表达式模板

`(\d+(\.\d+)?)`

`(Mr?|Ms?|Mrs?|Miss?)[A-Z][a-z]*`

`[A-Za-z-._-]+`

匹配的字符串

代表简单浮点数的字符串，即任意个十进制数字后面跟上一个可选的小数点，然后是零或多个十进制数字，比如“0.004”、“2”、“75.”等。

英文中的名和姓，对名字是有要求的（必须以大写开始，后面只能是小写字母——如果有的话）；人名的前面要加上一个称呼“Mr.”、“Mrs.”、“Ms.”，或“M.”；姓就比较灵活了，可以有多个单词，可以有连字符和大写字母。

15.3 规则表达式和Python语言

[1.5]

关于规则表达式本身也就是这么多东西，我们现在来看看Python是如何通过其re模块来支持规则表达式的。re模块是从版本1.5开始引入到Python语言中的。如果读者使用的是一个比较早期的Python版本，就不得不使用现在已经淘汰的regex和regsub模块——这些老式的模块有很浓的Emacs风味，而且功能也不太齐全，与现如今的re模块在很多地方都不兼容。

但不管怎么说，规则表达式还是规则表达式，所以本节中的大部分基本概念还是适用于老式的regex和regsub模块的。与之形成鲜明对比的是新的re模块支持功能更强大和更规则化的Perl风格（具体说来是Perl 5风格）的规则表达式，允许多个线程共享同一个经过编译的规则表达式对象，同时它还支持对规则表达式分组进行命名和按名字调用。另外有一个名为reconvert的转换模块是专门用来帮助开发人员从regex和regsub模块转向re模块的。总之，虽然规则表达式有不同的风格，但我们的注意力主要地还是要集中在现如今的Python语言上。

[1.6]

re引擎已经在版本1.6重新写过，改进了它的执行性能并添加了对Unicode的支持。操作接口倒还没有变，所以导致模块的名字也保持了原样。新的re引擎（它的内部称呼是sre）替换了原有的版本1.5的引擎（它的内部称呼是pcre）。

15.3.1 re模块的核心函数和方法

在表15-2里列出了re模块最常用的函数和方法。这些函数有许多也是适用于编译后得到的规则表达式对象（即“regex对象”和规则表达式“match对象”）的方法。在这一节里，我们将对两个主要的函数/方法进行介绍，它们是match()和search()；对compile()函数也多说几句。在下一节我们还会再多介绍几个，但是要想进一步了解我们这里介绍过的和没介绍过的函数/方法，我们建议读者去阅读Python的文档。

表15-2 常用规则表达式函数和方法

函数/方法	说 明
re模块的函数（没有与之对应的方法）	
<code>compile(pattern, flags=0)</code>	对规则表达式模板pattern进行编译，flags是可选的编译标志。返回一个regex对象
re模块的函数和regex对象的方法	
<code>match(pattern, string, flags=0)</code>	尝试把规则表达式pattern匹配到字符串string上去，flags是可选的编译标志。如果匹配成功，返回一个match对象；否则返回None
<code>search(pattern, string, flags=0)</code>	在字符串string里查找规则表达式pattern的第一次出现，flags是可选的编译标志。如果匹配成功，返回一个match对象；否则返回None
<code>findall(pattern, string)</code>	在字符串string里找出pattern的全部（不重复）出现。返回一个match对象构成的列表（Python 1.5.2版本新增功能）
<code>split(pattern, string, max=0)</code>	根据规则表达式pattern中的分隔符把字符串string分割为一个列表，返回成功匹配的列表；最多分割max次（缺省情况是分割出全部的匹配情况）
<code>sub(pattern, repl, string, max=0)</code>	把字符串string中所有匹配规则表达式pattern的地方替换为字符串repl；如果没有设定max的值，就对所有匹配的地方进行替换（另外请参考subn()，它还会返回一个表示替换次数的数值）
match对象的方法	
<code>group(num=0)</code>	返回整个匹配（或指定的编号为num的子组）
<code>groups()</code>	在一个表列里返回全部匹配到的子组（如果没有匹配，就返回一个空表列）

编程提示：RE编译（编译还是不编译？）

[CN]

我们曾经在上一章里说过：Python代码最终都要被编译为字节码以后才能被解释器执行。我们特别提到用`eval()`或`exec`调用一个代码对象与一个字符串相比会大幅提升代码的执行性能，因为这样可以省掉每次执行时必需的编译过程。换句话说，使用预编译过的代码对象要比使用字符串快很多，因为解释器在执行字符串形式的代码之前无论如何也要先把它编译成一个代码对象才行。

这个概念也同样适用于规则表达式——规则表达式在模板匹配操作开始进行之前必须先被编译成regex对象。如果规则表达式需要在整个程序的执行过程中被比较多次，我们强烈建议一定要先对它进行编译，道理还是这个：规则表达式总是要被编译的，所以提前弄好它对程序的执行性能有很重要的意义。`re.compile()`就是用来完成这一功能的。

其实那些模块函数也会缓冲保存一些编译过的对象，所以并不是每个使用同一个规则表达式的`search()`和`match()`都需要编译。因此，你可以保存对缓冲区的查找而不必用同样的字符串反复进行函数调用。在Python版本1.5.2里，这个缓冲区能够容纳20个编译过的规则表达式对象；但到了版本1.6，因为增加了对Unicode的支持，编译引擎的速度稍微慢了一些，所以缓冲区被扩大到能够容纳100个编译过的regex对象。

1. 使用`compile()`编译规则表达式

几乎所有我们马上就要讨论的re模块函数都可以用做regex对象的方法。记住，虽然有我们

强烈的建议，预编译过程也不是必不可少的。如果你编译了，使用的就是方法；如果你没编译，使用的就是函数。好消息是不管你采取哪一种办法，函数或方法的名字都是一样的。（这也是为什么有名字完全一样的函数和方法的原因，比如search()和match()等，就是为了不让你弄得头昏脑胀。）在后面的大多数例子里，我们主要采用字符串的方法，因为这样可以省下一个小步骤。但为了让你明白编译到底是怎样一回事，还是有几个例子给出了编译过程。

某些特定的编译操作需要使用一些可选的标志做为参数。这些参数允许匹配过程区分字母的大小写、使用系统的局部设置对字母和数字字符进行匹配，等等。详细情况请参考有关的文档。这些标志读者已经见过几个了（比如DOTALL和LOCAL），它们也可以用在模块版的search()和match()中以尝试匹配某些特定的模板。这些标志几乎都是为编译过程准备的，所以它们也可以被传递到模块版的search()和match()中去，因为它们至少要对规则表达式对象编译一次。如果你想在方法中使用这些标志，就必须事先把它们集成到经过编译的regex对象里去。

除下面将要介绍的方法外，regex对象还有几个数据属性，其中两个容纳着给定的编译标志和准备编译的规则表达式对象。

2. match对象和group()、groups()方法

在与规则表达式打交道的时候，除regex对象外还有另外一种对象类型，那就是match匹配对象。这些对象是从成功的search()和match()调用中返回而来的。匹配对象有两个基本的方法，它们就是group()和groups()。

group()或者返回一个整个匹配，或者按要求返回某个指定的子组。groups()则相对简单一点，它返回的是一个包含着某个或者全部匹配子组的表列。如果没有指定返回子组，groups()将返回一个空白表列，而group()依然会返回整个匹配。

Python语言中的规则表达式允许对匹配进行命名，这部分内容超出了这个规则表达式简介性质的小节的讨论范围。建议大家到完整的re模块文档里去进一步了解没有在这里提到的细节。

3. 使用match()匹配字符串

我们先来看看re模块的函数、规则表达式对象（即regex对象）的方法：match()。match()函数会从字符串的头部开始尝试对模板进行匹配。如果这个匹配是成功的，就返回一个匹配对象；如果匹配失败，就返回None。匹配对象的group()方法可以用来查看那个成功的匹配。下面是如何使用match()（以及group()）的一个例子：

```
>>> m = re.match('foo', 'foo') # pattern matches string
>>> if m != None:                # show match if successful
...     m.group()
...
'foo'
```

模板“foo”精确地匹配了字符串“foo”。在交互式解释器里我们可以确认m就是一个匹配对象，如下所示：

```
>>> m                                     # confirm match object returned
<re.MatchObject instance at 80ebf48>
```

下面是一个匹配失败的例子，它返回的是None：

```
>>> m = re.match('foo', 'bar') # pattern does not match string
```

```
>>> if m != None: m.group()    # (1-line version of if clause)
...
>>>
```

上例中的匹配失败了，所以把`m`赋值为`None`；因为我们没有在`if`语句里采取什么行动，所以也没有什么后续的处理。在以后的例子里，为了让例子更简洁，我们会在条件允许的情况下略去这个`if`语句；但在实际进行程序设计时，最好还是把它留在那里以防止出现`AttributeError`例外（因为匹配失败会返回`None`，可它却没有`group()`属性（方法））。

即使字符串比模板的长度要长，匹配依然有可能成功，这是因为匹配操作是从字符串的头部开始进行的。比如说，模板“foo”会在字符串“food on the table”里找到一个匹配，因为它从头部开始对模板进行匹配操作的，如下所示：

```
>>> m = re.match('foo', 'food on the table') # match succeeds
>>> m.group()
'foo'
```

正如你所看到的，虽然字符串比模板要长，但在字符串的开始处已经找到一个匹配了。子字符串“foo”代表的是从那个比较长的字符串里取出来的匹配部分。

在某些时候，我们可以省略保存中间结果的步骤，充分利用Python语言面向对象的特性，请看下面的例子：

```
>>> m = re.match('foo', 'food on the table').group()
'foo'
```

请注意：上面几个例子里如果发生匹配失败的情况将引发一个`AttributeError`例外。

4. 用`search()`在字符串里查找一个模板（搜索和匹配的比较）

一般说来，你要搜索的模板出现在字符串中间的机会要比出现在字符串开头的机会更大一些。这就是`search()`大显身手的地方了。它的工作方式与`match()`其实是完全一样的，只不过它可以查找到它的字符串参数任意位置处的规则表达式匹配情况。如果查找成功就返回一个匹配对象，否则返回`None`。

下面我们来看看`match()`和`search()`之间的差异。这次我们用一个比较长一点的例子来进行说明。下面把我们的“foo”匹配到“seafood”去：

```
>>> m = re.match('foo', 'seafood')          # no match
>>> if m != None: m.group()
...
>>>
```

正如你所看到的，这里没有匹配。`match()`在对模板进行匹配时一定要从字符串的头部开始，所以在这个例子里，模板中的“f”需要匹配到字符串中的“s”上，这当然是失败的。可事实上，`seafood`字符串里确实有“foo”出现（只是在别的位置而已），因此，怎样做才能让Python回答yes呢？答案是使用`search()`函数。`search()`所完成的操作并不是准确意义上的匹配，它的作用是在字符串里找到第一个出现的模板匹配情况。准确地说，`search()`是从左到右进行搜索，如下所示：

```
>>> m = re.search('foo', 'seafood')          # use search() instead
>>> if m != None: m.group()
```



```
...
'foo'          # search succeeds where match failed
>>>
```

在本节以后的内容里，我们将使用作为regex对象方法的match()和search()以及作为匹配对象方法的group()和groups()通过大量示例告诉大家如何在Python语言中使用规则表达式。我们将用到几乎全部的构成规则表达式语法的特殊字符和符号。

5. 匹配多个字符串 (|)

在15.2节里，我们在规则表达式“bat|bet|bit”里使用了多选一操作符。我们把这个规则表达式用在下面的Python代码里：

```
>>> bt = 'bat|bet|bit'          # RE pattern: bat, bet, bit
>>> m = re.match(bt, 'bat')     # 'bat' is a match
>>> if m != None: m.group()
...
'bat'
>>> m = re.match(bt, 'blt')     # no match for 'blt'
>>> if m != None: m.group()
...
>>> m = re.match(bt, 'He bit me!') # does not match string
>>> if m != None: m.group()
...
>>> m = re.search(bt, 'He bit me!') # found 'bit' via search
>>> if m != None: m.group()
...
'bit'
```

6. 匹配任意一个单个的字符 (.)

在下面的例子里，大家可以看到点字符是不能匹配换行符或“非字符”（即空字符串）的：

```
>>> anyend = '.end'
>>> m = re.match(anyend, 'bend') # dot matches 'b'
>>> if m != None: m.group()
...
'bend'
>>> m = re.match(anyend, 'end')  # no char to match
>>> if m != None: m.group()
...
>>> m = re.match(anyend, '\nend') # any char except \n
>>> if m != None: m.group()
...
>>> m = re.search('.end', 'The end.') # matches '.' in search
>>> if m != None: m.group()
...
'end'
```

下面是用来真正查找一个点字符（小数点）的规则表达式，我们用一个反斜线对它的功能进行了转义：

```
>>> patt314 = '3.14'          # RE dot
>>> pi_patt = '3\\.14'         # literal dot (dec. point)
>>> m = re.match(pi_patt, '3.14') # exact match
>>> if m != None: m.group()
...
'3.14'
>>> m = re.match(patt314, '3014') # dot matches '0'
```

```
>>> if m != None: m.group()
...
'3014'
>>> m = re.match(patt314, '3.14') # dot matches '.'
>>> if m != None: m.group()
...
'3.14'
```

7. 组建字符集合 ([])

我们在前面对 “[cr][23][dp][o2]” 以及它与 “r2d2lc3po” 的区别已经讲得不少了。从下面的例子可以看出 “r2d2lc3po” 比 “[cr][23][dp][o2]” 要严格得多:

```
>>> m = re.match('[cr][23][dp][o2]', 'c3po') # matches 'c3po'
>>> if m != None: m.group()
...
'c3po'
>>> m = re.match('[cr][23][dp][o2]', 'c2do') # matches 'c2do'
>>> if m != None: m.group()
...
'c2do'
>>> m = re.match('r2d2|c3po', 'c2do') # does not match 'c2do'
>>> if m != None: m.group()
...
>>> m = re.match('r2d2|c3po', 'r2d2') # matches 'r2d2'
>>> if m != None: m.group()
...
'r2d2'
```

8. 重复、特殊字符和规则表达式分组

规则表达式的常用情况包括特殊字符的使用、规则表达式模板的重复出现, 以及使用圆括号对规则表达式进行分组组合。我们曾经提到过一个用来表示普通电子邮件的规则表达式 (“\w+@\w+\.com”)。假定我们想匹配的电子邮件地址比这个规则表达式所允许的多, 比如说, 假设我们还想在域名的前面再加上一个主机名, 比如说 “www.xxx.com”, 而不仅仅是把 “xxx.com” 作为域名, 就必须修改现有的规则表达式。为了表示这个主机名是可选的、需要我们先建立一个匹配主机名 (后面还要再加上一个英文句号) 的模板, 然后在它的后面加上一个问号 “?” 表示该模板最多只出现一次, 最后再把这个可选的规则表达式插入到我们原来的那个规则表达式里去, 整个规则表达式将是这样的: “\w+@(\w+\.)?\w+\.com”。从下面的例子就可以看出, 现在在 “.com” 前有一或两个名字就都是可接受的了。如下所示:

```
>>> patt = '\w+@(\w+\. )?\w+\.com'
>>> re.match(patt, 'nobody@xxx.com').group()
'nobody@xxx.com'
>>> re.match(patt, 'nobody@www.xxx.com').group()
'nobody@www.xxx.com'
```

再进一步, 我们甚至可以用下面的模板把这个例子扩展为允许有任意个数的中间子域名存在的情况: “\w+@(\w+\.)*\w+\.com”:

```
>>> patt = '\w+@(\w+\.)*\w+\.com'
>>> re.match(patt,
'nobody@www.xxx.yyy.zzz.com').group()
'nobody@www.xxx.yyy.zzz.com'
```

但我们要在这里说明一下, 仅使用字母数字字符并不能完全匹配使用能够用来构成电子邮

件地址的字符。上面的规则表达式模板就匹配不了像“xxx-yyy.com”这样或本身带有“\W”字符的其他域名。

我们曾经在前面的内容里提到过：用圆括号来匹配和保存子组以便进一步处理要比另外再编写一段专门在确定存在有规则表达式匹配以后再人工分析一个字符串的程序要好。特别地，我们对规则表达式“\w+-\d+”进行了讨论，这个规则表达式描述的是一个用连字符（-）连在一起的一个字母数字字符串和一个数字；同时还讨论了如何通过对规则表达式进行分组组合而构成一个新的规则表达式“(\w+)-(\d+)”，这个新的规则表达式将圆满地达成我们的目的。下面是最初那个规则表达式执行时的情况：

```
>>> m = re.match('\w\w\w-\d\d\d', 'abc-123')
>>> if m != None: m.group()
...
'abc-123'

>>> m = re.match('\w\w\w-\d\d\d', 'abc-xyz')
>>> if m != None: m.group()
...
>>>
```

在上面的代码里，我们创建的规则表达式能够识别由连续三个字母数字字符再跟上三个数字字符所组成的字符串。用“abc-123”来测试我们的规则表达式成功了，但用“abc-xyz”测试这个规则表达式时失败了。现在按照前面的讨论对这个规则表达式做一些修改，使我们能够分别取出前一部分的字母数字字符串和后一部分的数字。请注意我们是如何通过group()方法分别访问那两个子组以及如何通过groups()方法获得一个由全体匹配到的子组构成的表列的：

```
>>> m = re.match('(\w\w\w)-(\d\d\d)', 'abc-123')
>>> m.group()           # entire match
'abc-123'
>>> m.group(1)          # subgroup 1
'abc'
>>> m.group(2)          # subgroup 2
'123'
>>> m.groups()          # all subgroups
('abc', '123')
```

正如你所看到的，group()的普通用法给出的是整个的匹配，但也可以用来分别取出匹配到的各个子组。同时，groups()给出的则是一个由全体匹配到的子组构成的表列。

下面这个例子也很简单，它给出了各子组的划分情况。看过这个例子，大家应该更明白了吧：

```
>>> m = re.match('ab', 'ab')      # no subgroups
>>> m.group()                     # entire match
'ab'
>>> m.groups()                   # all subgroups
()
>>>
>>> m = re.match('{ab}', 'ab')    # one subgroup
>>> m.group()                     # entire match
'ab'
>>> m.group(1)                   # subgroup 1
'ab'
>>> m.groups()                   # all subgroups
('ab',)
```

```

>>>
>>> m = re.match('(a)(b)', 'ab') # two subgroups
>>> m.group()                      # entire match
'ab'
>>> m.group(1)                     # subgroup 1
'a'
>>> m.group(2)                     # subgroup 2
'b'
>>> m.groups()                     # all subgroups
('a', 'b')
>>>
>>> m = re.match('(a(b))', 'ab') # two subgroups
>>> m.group()                      # entire match
'ab'
>>> m.group(1)                     # subgroup 1
'ab'
>>> m.group(2)                     # subgroup 2
'b'
>>> m.groups()                     # all subgroups
('ab', 'b')

```

9. 从字符串的开头或结尾以及在单词边界上的匹配

下面这个例子展示了规则表达式位置操作符的用法。就搜索和匹配两种操作来说，前者用到规则表达式位置操作符的情况要比后者更多一些，这是因为match()永远都是从字符串的头部开始进行的：

```

>>> m = re.search('^The', 'The end.') # match
>>> if m != None: m.group()
...
'The'
>>> m = re.search('^The', 'end. The') # not at beginning
>>> if m != None: m.group()
...
>>> m = re.search(r'\bthe', 'bite the dog') # at a boundary
>>> if m != None: m.group()
...
'the'
>>> m = re.search(r'\bthe', 'bitethe dog') # no boundary
>>> if m != None: m.group()
...
>>> m = re.search(r'\Bthe', 'bitethe dog') # no boundary
>>> if m != None: m.group()
...
'the'

```

读者在这里还应该注意到生字符串的出现。在这一章后面的内容里专门有一个编程提示用来说明为什么要在这里使用生字符串。一般来说，在规则表达式里使用生字符串是个很好的主意。

15.3.2 re模块的其他函数和方法

我们认为还应该让大家了解一下另外四个re模块的函数和regex对象的方法，它们是：findall()、sub()、subn()和split()。

1. 用findall()找出全部匹配

`findall()`是从Python版本1.5.2开始新增加的功能。它的作用是在一个字符串里不重叠地找出给定规则表达式的全部匹配情况。从对字符串的处理角度看它类似于`search()`所进行的搜索，但它与`match()`和`search()`的区别也很明显——`findall()`返回的永远都是一个列表。如果没有匹配，这个列表将会是空的；但要是匹配，这个列表里面就包含着找到的全部匹配（按匹配从左到右的顺序排列）。如下所示：

```
>>> re.findall('car', 'car')
['car']
>>> re.findall('car', 'scary')
['car']
>>> re.findall('car', 'carry the barcardi to the car')
['car', 'car', 'car']
```

要是规则表达式里还有子组，那`findall()`返回的列表就更复杂了；可这也很有道理，因为子组本身就是能够让通过单一的规则表达式而提取出各种特定模板的一种机制，比如从一个完整的电话号码里提取出长途区号，或者从一个完整的电子邮件地址里提取出用户名等。

如果成功的匹配只有一个，那每个子组匹配就将是`findall()`返回的结果列表中的一个元素；如果成功的匹配不止一个，那每个子组匹配就将是一个表列中的一个元素，再由这些表列（每个表列对应一个成功的匹配情况）构成最终的结果列表。这几句话乍读起来很不好懂，但多用一些例子做练习就会帮助你尽快地理解它们了。

2. 用`sub()`和`subn()`进行搜索与替换

在搜索与替换方面主要有两个函数/方法，它们是`sub()`和`subn()`。它们的功能几乎完全一样，都是把某个规则表达式模板在一个字符串里的全部匹配用另外的什么东西替换掉。这个用来替换原来内容的东西一般是一个字符串，但也可以是一个函数——该函数必须能够返回一个字符串（用来替换原来内容）。`subn()`和`sub()`在功能上是完全一样的，只是它还会多返回一个数字，这个数字指出对该字符串进行了多少次替换，替换后得到的新字符串和替换次数构成了一个有两个元素的表列。请看下面的例子：

```
>>> re.sub('X', 'Mr. Smith', 'attn: X\n\nDear X,\n')
'attn: Mr. Smith\n\nDear Mr. Smith,\n'
>>>
>>> re.subn('X', 'Mr. Smith', 'attn: X\n\nDear X,\n')
('attn: Mr. Smith\n\nDear Mr. Smith,\n', 2)
>>>
>>> print re.sub('X', 'Mr. Smith', 'attn: X\n\nDear X,\n')
attn: Mr. Smith

Dear Mr. Smith,
>>> re.sub('[ae]', 'X', 'abcdef')
'XbcdXf'
>>> re.subn('[ae]', 'X', 'abcdef')
('XbcdXf', 2)
```

3. 用`split()`（使用分隔模板）进行分割

字符串类型有一个`split()`方法，`re`模块和规则表达式对象也有一个`split()`方法，两个方法的功能都差不多；但前者用内容固定的字符串进行分割，而后者是在一个规则表达式模板的基础上进行分割，所以后者的字符串分割能力明显地强于前者。如果你不想在每个匹配了模板的地方

都对字符串进行分割，可以通过给max参数赋一个值（不能是0）的办法来限定最大分割次数。

如果在re.split()里面给出的分隔符不是一个使用了能够匹配多种模板的特殊字符的规则表达式，那它的执行情况就与string.split()完全一样，请看下面的例子（这个例子在每个冒号处对字符串进行分割）：

```
>>> re.split(':', 'str1:str2:str3 ')
['str1', 'str2', 'str3' ]
```

要是给出的是一个规则表达式，re.split()这个工具的功能可就大了。请看：UNIX操作系统的who命令将输出一个清单，这是某个系统里面全体用户的一个名单：

```
% who
wesc      console      Jun 20 20:33
wesc      pts/9          Jun 22 01:38      (192.168.0.6)
wesc      pts/1          Jun 20 20:33      (:0.0)
wesc      pts/2          Jun 20 20:33      (:0.0)
wesc      pts/4          Jun 20 20:33      (:0.0)
wesc      pts/3          Jun 20 20:33      (:0.0)
wesc      pts/5          Jun 20 20:33      (:0.0)
wesc      pts/6          Jun 20 20:33      (:0.0)
wesc      pts/7          Jun 20 20:33      (:0.0)
wesc      pts/8          Jun 20 20:33      (:0.0)
```

假设我们想把某些用户信息保存起来，比方说登录名、用户登录时使用的通信方式、他们的登录时间，以及从哪里登录的，等等。要是用string.split()来处理这些东西是很吃力的，因为分隔各数据项的那些空格的个数是不固定的；同时，月、日、时等数据域之间分别有一个空格，而我们想把这几个数据域保留为一个数据项。

我们的模板应该表达这样一个意思：“从有两个或两个以上空格的地方进行分割”。规则表达式很容易就能做到这一点。说干就干，至少两个空格字符，“\s\s+”，规则表达式模板就是它了。我们来编写一个名为rewho.py的程序，它要把who命令的输出（假设已经保存在一个名为whodata.txt的文件里了）读进来。原始的rewho.py程序应该是下面这个样子：

```
import re
f = open('whodata.txt', 'r')
for eachLine in f.readlines():
    print re.split('\s\s+', eachLine)
f.close()
```

现在执行who命令，把它的输出保存到whodata.txt文件里去，然后调用rewho.py程序看看是什么结果：

```
% who > whodata.txt
% rewho.py
['wesc', 'console', 'Jun 20 20:33\012']
['wesc', 'pts/9', 'Jun 22 01:38\011(192.168.0.6)\012']
['wesc', 'pts/1', 'Jun 20 20:33\011(:0.0)\012']
['wesc', 'pts/2', 'Jun 20 20:33\011(:0.0)\012']
['wesc', 'pts/4', 'Jun 20 20:33\011(:0.0)\012']
['wesc', 'pts/3', 'Jun 20 20:33\011(:0.0)\012']
['wesc', 'pts/5', 'Jun 20 20:33\011(:0.0)\012']
['wesc', 'pts/6', 'Jun 20 20:33\011(:0.0)\012']
['wesc', 'pts/7', 'Jun 20 20:33\011(:0.0)\012']
```

```
['wesc', 'pts/8', 'Jun 20 20:33\011(:0.0)\012']
```

看上去还行，但算不上完全正确。首先，我们没想到命令的输出里会夹杂有一个制表符TAB（ASCII字符011）——它看上去像是两个空格，对不对？其次，我们也不太想在最终结果里留着换行符（ASCII字符012）——它的作用只是结束每一行而已。我们给这个应用程序动些手术，把这些毛病去掉，全面改进它的质量。

首先，我们把who命令挪到脚本程序里面来运行，也就是说，不在脚本程序外完成把它的输出保存到whodata.txt文件这项工作——像这样一次又一次地操作很快就让人烦了。在脚本程序里调用另外一个程序的工作交给os.popen()去完成，os.popen()是在14.5.2节里介绍过的。虽然只有在UNIX系统上才能用os.popen()，但我们的目的是演示re.split()的功能和用法，它可是在所有计算机平台上都能用的。

我们还要把map()内建函数和string.strip()结合在一起去掉尾缀的换行符。最后，还要修改一下我们的规则表达式，让它能够检测出单个的制表符TAB并把它作为re.split()的另一个分隔符。在下面程序示例15-1里给出的就是最终的rewho.py脚本程序了。

程序示例15-1 对UNIX操作系统的who命令输出进行分割（rewho.py）。

这个脚本程序调用了who命令，并在各种空格间隔位置上对输入数据进行了分割。

```
1  #!/usr/bin/env python
2
3  from os import popen
4  from re import split
5  from string import strip
6
7  f = popen('who', 'r')
8  for eachLine in map(strip, f.readlines()):
9      print split('\s\s+|\t', eachLine)
10 f.close()
```

运行这个脚本程序，我们就得到如下所示的正确输出：

```
% rewho.py
['wesc', 'console', 'Jun 20 20:33']
['wesc', 'pts/9', 'Jun 22 01:38', '(192.168.0.6)']
['wesc', 'pts/1', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/2', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/4', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/3', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/5', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/6', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/7', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/8', 'Jun 20 20:33', '(:0.0)']
```

如果是在DOS或Windows环境里，做这个练习的时候要用dir命令代替who。

编程提示：Python生字符串的使用

[CN]

读者可能已经在前面的一些例子里见到过生字符串了。可以这样说：规则表达式是促使生字符串诞生的重要因素。其原因就在于ASCII字符和规则表达式特殊字符之间的使用

冲突上。“\b”是一个特殊字符，它既代表着ASCII的后退（backspace）字符，又是规则表达式里一个代表着“匹配一个单词边界”的特殊字符。为了不让规则表达式编译器把你规则表达式模板字符串中的“\b”字符解释为后退字符，就必须用反斜线字符对它进行转义，最终的结果是使它成为“\\b”的样子。

这样做很容易发生混淆，在规则表达式字符串里有许多个特殊字符的情况下就更是如此，可以说是乱上添乱。而我们在6.4.2节里介绍的生字符串可以用来帮助我们使规则表达式看上去更明晰，更有可管理性。事实上，许多Python程序员在定义规则表达式的时候只使用生字符串。下面几个例子有的使用了生字符串，有的没有使用生字符串，目的就是让大家看清后退字符“\b”和规则表达式字符“\\b”之间的区别：

```
>>> m = re.match( '\\bblow', 'blow') # backspace, no match
>>> if m != None: m.group()
...
>>> m = re.match( '\\\\bblow', 'blow') # escaped \, now it works
>>> if m != None: m.group()
...
'blow'
>>> m = re.match(r'\\bblow', 'blow') # or use raw string
instead
>>> if m != None: m.group()
...
'blow'
```

读者可能已经注意到我们在规则表达式里使用“\d”时即使没有使用生字符串也没有遇到什么麻烦。这是因为没有与之对应的ASCII特殊字符，所以规则表达式编译器知道你指的就是一个第十进制数字。

15.4 规则表达式的使用示例

我们现在要用一个比较深入的例子向大家展示一下用规则表达式对字符串进行处理的各种办法。第一步是编写一段代码来生成一些随机（当然也不是一点规律也没有）数据，我们将对这些“随机”数据进行处理。请看程序示例15-2里的gendata.py脚本程序，我们将用它来生成一个数据集。虽然这个程序只是简单地把它生成的那些字符串输出到标准输出去，但这个输出完全可以重定向到一个测试用文件里去。

编程提示：32位数据和“时间终点”

[CN]

UNIX系统，以及其他一些系统，都是使用体系结构尺寸的整数来表示以秒计时的当前时间的。因为现如今的大多数系统都是32位的，所以任何一种使用这一机制的计算机平台能够识别的时间总值是 2^{32} 秒。又因为这些整数都是带符号整数，所以我们实际上只有 $2^{31}-1$ 秒。系统的当前时间是以从零时间开始已经逝去的秒数来识别的，这个零时间被设定为1970年1月1日的0点0分0秒。与32位带符号正整数所能表示的最大值（ $2^{31}-1$ ）对应的就是所谓的“时间终点”，计算得出的结果是国际协调时间（格林威治时间）2038年1月19日的上午3点14分7秒。希望那时我们早已经不使用32位的计算机了。这个现象也被称为Y2038问

题（千年虫）。

下面这个办法可以帮助读者确定与“时间终点”这个特殊时间对应的本地时间是什么时候，使用Python语言：

```
>>> import sys, time
>>> time.asctime(time.localtime(sys.maxint))# Pacific Time
'Mon Jan 18 19:14:07 2038'
```

sys.maxint是用32位整数所能表示的最大的时间值。我们先把这个以秒计算的时间值送到time.localtime()里去转换为一个包含着本地对间值的表列（我们这里是太平洋时间）；最后，我们把表列送入time.asctime()得到用标准时间标签形式表示的“最后一秒”的时间。从上面的例子里可以看出，我们要比格林威治基准时晚8个小时。

这段内容不完全是个程序设计方面的提示，它更像是常识方面的讨论，因为它的涉及面很广，包括全体有C语言应用程序的32位系统——不管具体是哪一种计算机平台，即所有UNIX系统和使用UNIX计时办法的非UNIX系统。在后面的gendata.py脚本程序里，我们使用随机生成的整数来有效地生成供我们程序使用的随机时间。

这个脚本程序生成的字符串有三个部分组成，各部分之间用了两个冒号来分隔。头一个部分是一个随机整数（32位），它被转换为一个时间（请参考上面的编程提示）。接下来是一个随机生成的电子邮件地址，最后是用一个连字符隔开的两个整数。

程序示例15-2 练习规则表达式的数据生成器（gendata.py）

为练习规则表达式生成随机数据，并把生成的数据输出到屏幕上。

```
1  #!/usr/bin/env python
2
3  from random import randint, choice
4  from string import lowercase
5  from sys import maxint
6  from time import ctime
7
8  doms = ( 'com', 'edu', 'net', 'org', 'gov' )
9
10 for i in range(randint(5, 10)):
11     dtint = randint(0, maxint-1) # pick date
12     dtstr = ctime(dtint)         # date string
13
14     shorter = randint(4, 7)      # login shorter
15     em = ''
16     for j in range(shorter):     # generate login
17         em = em + choice(lowercase)
18
19     longer = randint(shorter, 12) # domain longer
20     dn = ''
21     for j in range(longer):      # create domain
22         dn = dn + choice(lowercase)
23
24     print '%s::%s@%s.%s::%d-%d-%d' % (dtstr, em,
25         dn, choice(doms), dtint, shorter, longer)
```

运行这段代码，我们得到如下所示的输出（读者们的输出肯定是不和这里的一样的），这些数据还被保存到redata.txt文件里：

```

Thu Jul 22 19:21:19 2004::izsp@dicqdhvtvhv.edu::1090549279-4-11
Sun Jul 13 22:42:11 2008::zqeu@dxaiobjgkniy.com::1216014131-4-11
Sat May 5 16:36:23 1990::fclihw@alwdbzpsdg.edu::641950583-6-10
Thu Feb 15 17:46:04 2007::uzifzf@dpyivihw.gov::1171590364-6-8
Thu Jun 26 19:08:59 2036::ugxfugt@jkhughs.net::2098145339-7-7
Tue Apr 10 01:04:45 2012::zkwaq@rpxwmtikse.com::1334045085-5-10

```

不管你相信还是不相信，这个程序的输出非常适合用规则表达式进行处理。在我们的逐行解释后面，我们将用几个规则表达式对这些数据进行操作；在本章后面的练习里也有许多与此有关的问题。

逐行解释

第1~6行

在这个示例性的脚本程序里我们将用到多个模块。但因为我们只会用到这些模块里的一两个函数，所以不需要把整个模块都导入进来，只要把需要的属性导入进来就足够了。因此，我们在这里选用的是from-import而不是import语句。脚本程序的第一行是UNIX操作系统下的启动代码，后面几行是必要的from-import代码。

第8行

doms很简单，它包含着一组顶级域名，我们将从中随机挑选一个来组成我们随机生成的电子邮件地址。

第10~12行

gendata.py每次执行都会随机生成5到10行输出。这个脚本程序在需要用到随机数字的地方使用的都是random.randint()函数。在每个输出数据行里，我们会在整个可能的范围内（0到 $2^{31}-1$ （即sys.maxint））随机选择一个整数，再把这个整数通过time.ctime()转换为一个日期。

第14~22行

我们把那些随机生成的假电子邮件地址的登录名取为4到7个字符长。我们随机选取4到7个小写字母，把他们依次合并到一个字符串里去。random.choice()函数的功能是从给定的一个序列里随机挑出一个元素来。在我们的这个例子里，给定的序列就是字母表中的26个小写字母string.lowercase。

我们决定把假电子邮件地址中的主域名取为4到12个字符长，但至少要和登录名一样长。我们还是随机挑出几个小写字母组成这个名字。

第24~25行

这是我们这个脚本程序的主程序部分，它把所有随机数据组成为一个输出行。各项数据之间用分隔符隔开。先是日期，然后是随机生成的电子邮件地址——这个电子邮件地址是把登录名、“@”符号、域名和随机挑出的顶级域名合并在一起得到的。在随后一个双冒号后面，我们还加上了一个随机整数字符串，这个字符串前面是与日期字符串对应的那个随机整数，后面跟着登录名和域名的长度，这几个整数之间用连字符隔开。

匹配一个字符串

在下面的练习里，我们将尽量给出宽松和严格两种版本的规则表达式。我们建议大家编写一个小程序，然后用上面给出的redata.txt（或读者自己运行gendata.py而生成的数据）对这些规则表达式进行测试。在本章后面的练习里你还会用到这些数据的。

在把规则表达式加到我们的小程序里之前需要先对它们进行一番测试，因此需要我们导入re模块把redata.txt文件中的某一行赋值给一个字符串变量data。这两个语句在以后的示范中是不会发生什么变化的。如下所示：

```
>>> import re
>>> data = 'Thu Feb 15 17:46:04 2007:uzifzf@pyivihw.gov:1171590364-6-8'
```

在第一个例子里，我们将创建一个规则表达式，用它从数据文件redata.txt里（唯一地）取出时间标签中的星期项。我们将使用如下所示的规则表达式：

```
"^Mon|^Tue|^Wed|^Thu|^Fri|^Sat|^Sun"
```

这个规则表达式描述的是以上面列出的七个字符串打头（规则表达式操作符“^”）的字符串。如果我们想把上面的这个规则表达式翻译过来，它的意思大概是：以Mon、Tue、... Sat、Sun打头的字符串。

如果像下面这样把星期字符串都归为一组，就可以只留下一个上箭头符号，其余的上箭头符号都可以不要了：

```
"^(Mon|Tue|Wed|Thu|Fri|Sat|Sun)"
```

这组字符串两头的圆括号表示一个成功的匹配必须遇到这些字符串中的某个。与前面那个没有加上圆括号的规则表达式相比，这个要“友好”一些。而这个改进了的规则表达式还有一个好处，就是能够让我们随后去访问作为被匹配字符串的那个组，如下所示：

```
>>> patt = '^ (Mon|Tue|Wed|Thu|Fri|Sat|Sun) '
>>> m = re.match(patt, data)
>>> m.group()                # entire match
'Thu'
>>> m.group(1)               # subgroup 1
'Thu'
>>> m.groups()               # all subgroups
('Thu',)
```

就我们现在这个例子来看，取子组字符串的功能似乎没有什么大用，但在下面的例子里就不一样了。如果需要在规则表达式里加上一些额外的数据来帮助字符串匹配过程，即使这些额外字符并非必须出现在想要找出来的字符串里，这个功能也很有用。

上面的这两个规则表达式都是限制性比较强的，专门对应于一组字符串。但如果是在一个国际化的环境里，各地区又使用着当地的日期表示形式和缩写，现在这种做法就不太得当。比较宽松的规则表达式是：“^\\w{3}”。这个规则表达式只要有任意三个连续的字母数字集合中的字符就能满足。要是把它翻译过来的话，上箭头表示“以...开始”；“\\w”表示任意一个字母数字字符；“{3}”表示“{3}”前面的那个规则表达式所描述的情况必须连续出现三次。如果还要对它进行分组，就还要加上圆括号，即“^(\\w{3})”：

```
>>> patt = '^ (\\w{3}) '
>>> m = re.match(patt, data)
>>> if m != None: m.group()
...
'Thu'
>>> m.group(1)
'Thu'
```

注意，要是把这个规则表达式写成了“`^(\w){3}`”就不正确了。当“`{3}`”在圆括号里面时，会先匹配三个连续的字母数字字符，再把它们作为一个子组来对待。而把“`{3}`”挪到括号外面去以后，就等于是再找三个连续的单个字母数字字符了，如下所示：

```
>>> patt = '^(\w){3}'
>>> m = re.match(patt, data)
>>> if m != None: m.group()
...
'Thu'
>>> m.group(1)
'u'
```

在访问子组1时我们只看到了一个“u”，这是为什么呢？原因在于子组1不停地被“下一个”字符替换了三次。换句话说，`m.group(1)`最开始的时候是“T”，然后被替换为“h”，最后又被替换为“u”。这是三个彼此没有关系的（同时又是重叠了的）单个字母数字字符，而不是一个由连续三个字母数字字符组成的子组。

在下一个（也是最后一个）例子里，我们编写的规则表达式将从`redata.txt`文件的数据行里把每一行最后面的那些数值提取出来。

搜索和匹配的比较

在开始写出规则表达式之前，先做些分析工作。我们注意到这些整数数据项都出现在数据字符串的尾部。这就意味着我们可以选择用搜索还是用匹配。用搜索更好一些，因为我们知道自己想要找的东西是什么（三个整数）、我们要找的东西不在字符串的开头，同时我们要找的东西只是整个字符串的一小部分。如果我们打算用匹配来完成这件事，就不得不写出一个能够匹配整个数据行的规则表达式，再通过子组来保存我们想要的数值。为了展示两者之间的差异，我们先用搜索做一遍，然后再用匹配做一遍，向大家证明搜索是更合适的。

因为我们要找的是用连字符（-）隔开的三个整数，所以我们给出下面这样的规则表达式来：“`\d+-\d+-\d+`”。这个规则表达式描述的是“任意位数（至少1个）的数字，后面跟一个连字符；再一个数字，又一个连字符；然后是最后一个数字。现在用`search()`来测试一下我们的规则表达式：

```
>>> patt = '\d+-\d+-\d+'
>>> re.search(patt, data).group() # entire match
'1171590364-6-8'
```

用这个规则表达式进行匹配操作肯定会失败。为什么？因为匹配是从字符串的头部开始进行的，而这些数值字符串出现在尾部。我们必须另外写出一个能够匹配整行字符串的规则表达式来。偷个懒，就用“`.*`”来表示任意个字符，然后加上我们真正想要的东西，现在请看下面这个例子：

```
patt = '.*\d+-\d+-\d+'
>>> re.match(patt, data).group() # entire match
'Thu Feb 15 17:46:04
2007::uzifzf@dpyivihw.gov::1171590364-6-8'
```

效果不错，但我们需要的是尾部的数字部分而不是整个字符串，所以需要用圆括号对它进行分组组合，如下所示：

```
>>> patt = '.*+(\d+-\d+-\d+)'
>>> re.match(patt, data).group(1) # subgroup 1
'4-6-8'
```

哎呀！出什么事情了？我们取出的应该是“1171590364-6-8”，不应该只有“4-6-8”呀。头一个数字的前半部分哪里去了？问题的根源在于：规则表达式的本性是比较“贪婪”的。这句话的意思：如果在规则表达式里使用了通配性的特殊字符，那么当这个规则表达式按从左向右的顺序进行求值的时候，会尽量“抓取”能够与模板匹配的最多的字符。在我们上面的例子里，“.”会从字符串的头部开始抓取每一个单个的字符，其中就包括了本来是我们想要的第一个整数部分里的大部分字符。“\d+”只需要一个数字就可以满足，所以它得到的是“4”，而“.”则匹配了从字符串的开始到这个数字“4”之间的所有东西：“Thu Feb 15 17:46:04 2007::uzifzf@dpyivihw.gov::117159036”，就像下面图15-2中所示的那样。

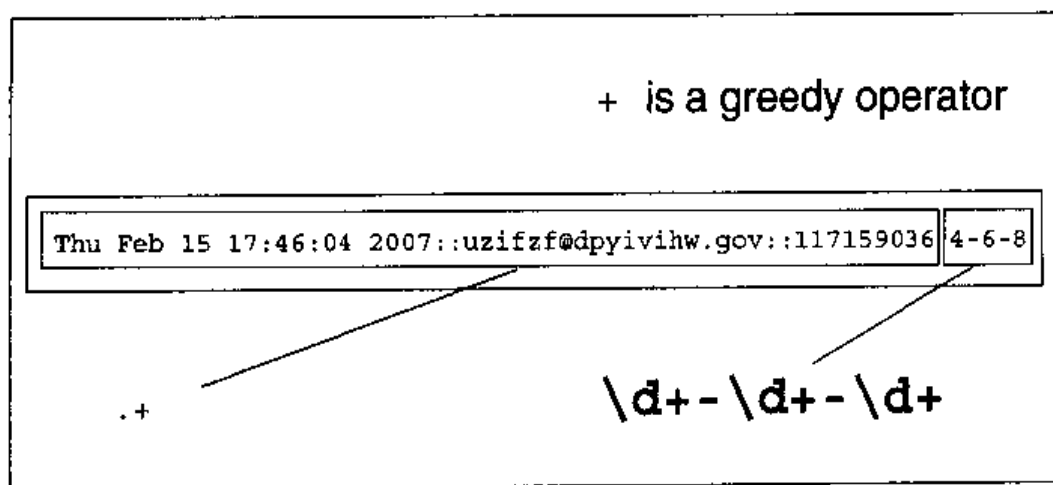


图15-2 为什么我们的匹配结果很怪：+是一个“贪婪”的操作符

解决这个问题的办法是使用一个“反贪”操作符“?”。这个操作符可以用在“*”、“+”或“?”的后面，其作用是通知规则表达式引擎匹配尽量少的字符。所以，如果我们在“.”后面加上一个“?”，就可以得到如图15-3所示的正确结果了。

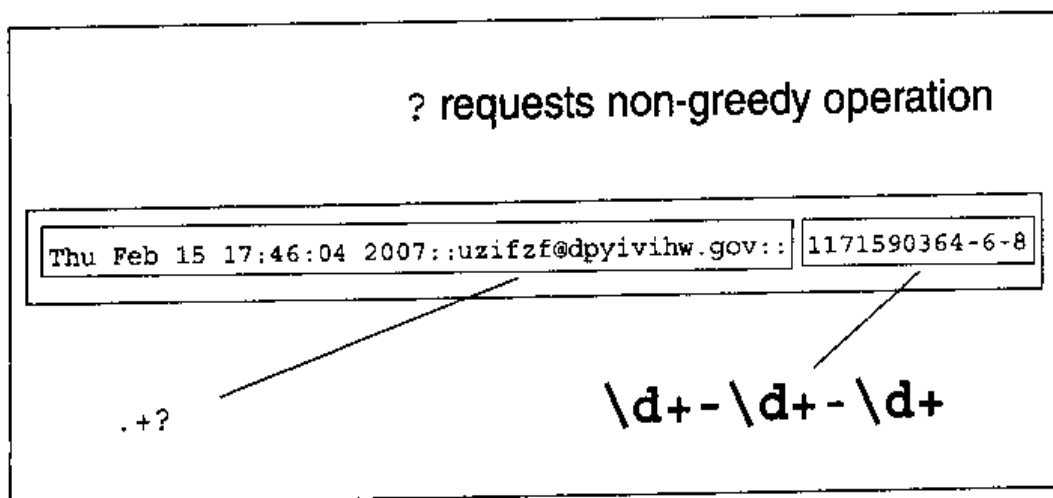


图15-3 解决“贪婪”问题：?指示进行不“贪婪”操作

```
>>> patt = '.*?(\d+--\d+--\d+)'
>>> re.match(patt, data).group(1)    # subgroup 1
'1171590364-6-8'
```

最后一个例子。假设我们只想取出三个整数数据里面中间的那个。这里是我们的解决办法（这里使用的是搜索操作，这样就不必匹配整个字符串了）：“-(\d+)-”。对这个模板进行测试，我们得到如下所示的结果：

```
>>> patt = '-(\d+)-'
>>> m = re.search(patt, data)
>>> m.group()                # entire match
'-6-'
>>> m.group(1)               # subgroup 1
'6'
```

在这一章里，我们初步接触到规则表达式的强大威力，但在这有限的篇幅里，我们无法逐个地对它们做详细的介绍。但我们希望已经给出了足够的介绍性资料，读者可以由此开始掌握这个程序设计技巧，把它作为自己进行程序设计的工具。我们建议读者参考有关文档，进一步学习如何在Python语言里使用规则表达式。在全面论述规则表达式领域各种问题的书籍里，我们推荐由Jeffrey E. F. Friedl编写的《Mastering Regular Expressions》（掌握规则表达式）一书。

15.5 练习

规则表达式。根据练习5-1到5-12中的要求写出正确的规则表达式来：

15-1 识别下列字符串：bat、bit、but、hat、hit或hut。

15-2 匹配用一个空格分开的任意两个单词，比如英文人名中的名和姓。

15-3 匹配用一个逗号和空格分开的的一个单词和一个字母，比如英文人名中的姓和名字的第一个字母。

15-4 匹配所有合法的Python标识符。

15-5 根据读者当地的地址写法匹配一个街道地址（要让你的规则表达式尽量通用一些，要求能够匹配街道单词中的任何数字，包括类型指示等）。比如说，美国街道地址使用的是这样的格式：“1180 Bordereaux Drive”。让你的规则表达式尽量通用，要求能够支持多单词的街道名称，如“3120 De la Cruz Boulevard”。

15-6 匹配简单的以“www.”打头并以“.com”结尾的Web域名，比如“www.yahoo.com”。

附加题：让规则表达式还支持其他顶级域名：.edu、.net等等，如“www.ucsc.edu”。

15-7 匹配全体Python整数的字符串表示形式。

15-8 匹配全体Python长整数的字符串表示形式。

15-9 匹配全体Python浮点数的字符串表示形式。

15-10 匹配全体Python复数的字符串表示形式。

15-11 匹配全体合法的电子邮件地址（先从一个比较宽松的的开始，然后尽最大可能收紧限制条件，但要保证功能的正确性）。

15-12 匹配全体合法的Web站点地址（先从一个比较宽松的的开始，然后尽最大可能收紧限制条件，但要保证功能的正确性）。

15-13 `type()`。`type()`内建函数返回的是一个类型（type）对象，这个对象被显示为如下所示的Python字符串形式：

```
>>> type(0)
<type 'int'>
>>> type(.34)
<type 'float'>
>>> type(dir)
<type 'builtin_function_or_method'>
```

请写出一个能够从这个字符串里取出类型名字的规则表达式来。要求你的函数以“<type int'>”这样的字符串为输入，返回'int'。（所有其他类型也都如此要求，比如'float'、'builtin_function_or_method'等等。）提示：你准备处理的这个数据保存在类或某些内建类型的__name__属性里。

15-14 规则表达式。在15.2节里，我们给出了用一或两位数字匹配一月到九月的字符串表示形式的一个模板（“0?[0-9]”）。请编写一个代表标准日历上其余三个月的规则表达式。

15-15 规则表达式。在15.2节里，我们给出了一个匹配信用卡卡号的规则表达式模板（“[0-9]{15,16}”）。但这个模板不允许用连字符分隔卡号中的数字组。请写出一个允许使用连字符的规则表达式，但连字符必须出现在正确的位置上。比如说，15位数信用卡编号的格式是4-6-5，即四个数字一个连字符，再六个数字一个连字符，最后又是五个数字。16位数信用卡编号的格式是4-4-4-4。整个字符串的长度必须是正确的，位数不足时要在前面加0补足之。

附加题：有一个用来确定某个信用卡卡号是否合法的标准算法。请编写一段代码，它不仅能够识别出一个格式正确的信用卡编号，还能检查出内容也合法的卡号。

下面几组问题（练习15-16到练习15-27）专门用来处理用gendata.py生成的数据。在开始着手解决练习15-17和15-18中的问题之前，可以先把练习15-16和所有的规则表达式做出来。

15-16 改进gendata.py脚本程序，使数据直接写入redata.txt文件而不是输出到屏幕上。

15-17 在生成一个redata.txt文件后，统计该文件里一个星期的每一天分别出现了多少次。（类似地，你也可以统计各个月份分别出现了多少次。）

15-18 验证redata.txt文件中的数据完整性。办法是检查各输出行中整数部分的第一个整数是否与该行最前面给出的时间标签相匹配。

按下面各个练习中的要求写出相应的规则表达式：

15-19 完整提取出各行中的时间标签。

15-20 完整提取出各行中的电子邮件地址。

15-21 只提取出时间标签里的月份。

15-22 只提取出时间标签里的年份。

15-23 只提取出时间标签里的时间值（HH:MM:SS.）。

15-24 只提取出电子邮件地址中的登录名和域名（主域名和顶级域名要连在一起）。

15-25 只提取出电子邮件地址中的登录名和域名（主域名和顶级域名要分别提取）。

15-26 把各数据行中的电子邮件地址替换为你自己的电子邮件地址。

15-27 提取出时间标签中的月、日、年并按“Mon Day, Year”（星期 日期，年）的格式显

示出来，要求每行只遍历一次。

在15.2节里，我们给出了一个匹配电话号码的规则表达式，其中的长途区号是固定的，即“\d{3}-\d{3}-\d{4}”。在练习15-28和练习15-29里，请修改这个规则表达式，使它满足：

15-28 长途区号（第一组的三个数字以及它后面的那个连字符）是可选的，即让你的规则表达式既能够匹配800-555-1212，也能够匹配555-1212。

15-29 长途区号既可以使用圆括号，也可以使用连字符，同时它们还是可选的。即让你的规则表达式匹配800-555-1212、555-1212和(800)555-1212。

第16章 网络程序设计

在本章里，我们将对使用套接字进行的网络程序设计做一个简单的介绍。我们先向大家介绍一些关于网络程序设计以及如何在Python语言里使用套接字等方面的背景知识，然后向大家展示如何使用几个Python模块来建立网络化的应用程序。

16.1 介绍

16.1.1 什么是客户-服务器体系结构

什么是客户-服务器体系结构？不同的人有不同的回答，这要看你问的是什么人以及你指的是软件系统还是硬件系统。不管是哪种情况，答案的核心都很简单：“服务器”是一个软件或者硬件部件，它向一个或多个“客户”提供一种“服务”；而“客户”就是使用该项服务的用户。服务器存在的唯一目的就是等待（客户）的请求，为那些客户服务，然后等待更多的请求。

做为另外一方，客户向一个（预先确定好了的）服务器发送一个特定的请求，提交必要的数据，然后等待服务器回答——或者完成了那个请求，或者指出了失败的原因。服务器永不停息地处理着请求，而客户请求一次性的服务，接受那项服务，然后结束它们之间的交易。一个客户可能会晚些时候再提出其他请求，但这将被视为另外一个交易了。

图16-1里给出的是现如今最常见的“客户-服务器”结构的示意图：一个用户或客户计算机正在通过因特网从一台服务器上检索资料。虽然一个这样的系统确实是客户-服务器体系结构的一个例子，但它并不是唯一的。广泛地说，客户-服务器体系结构可以用来描述计算机的硬件和软件等多方面。

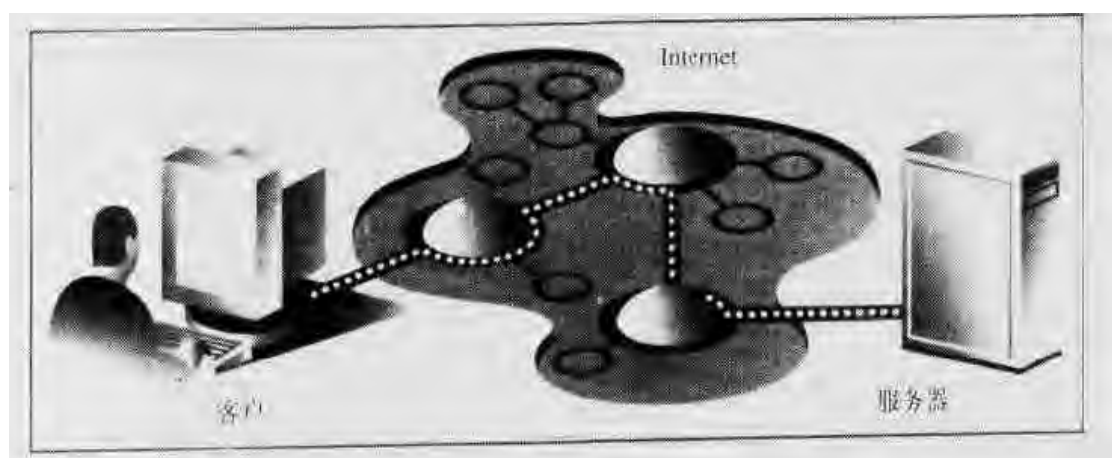


图16-1 因特网上一个客户-服务器系统的典型概念

1. 硬件客户-服务器体系结构

打印机服务器是硬件服务器的典型代表。它们对接收到的打印作业进行处理，把它们送到

一个与该系统相连接的打印机（或者其他打印设备）去。一台这样的计算机通常都可以通过网络来访问，客户机向它们发出打印请求。

硬件服务器的另外一个例子是文件服务器。它们通常会是一些能够被客户远程访问的具有巨大存储能力的计算机。客户机从服务器机器上把这些磁盘“挂装”到它们本地的机器上，就好像那些磁盘本身就放在本地机器上一样。最流行的一种支持文件服务器的网络操作系统是Sun微系统公司的“网络文件系统”（Network File System, NFS）。如果你正在访问一个网络上的硬盘驱动器却无法分辨出它是本地的还是网络上的，就说明客户服务器系统干得很好。从用户实际操作经验方面来看，这样做能够保证网络硬盘的操作与本地磁盘的操作方法完全一致——即“抽象”为一般的磁盘访问。而要想让它具备这样的行为方式，就需要有事先设计好的程序化“实现”来保证了。

2. 软件客户-服务器体系结构

软件服务器需要运行在一个硬件设备上，但它并不像硬件服务器那样必须有相应的外围设备如打印机、硬盘驱动器等等。软件服务器所能够提供的基本服务包括：程序执行、数据传输检索、资料积累、资料更新，以及其他程序化处理或数据处理等等。

现如今最常见的软件服务器之一是Web服务器。在一台计算机上设置好Web网页和/或Web应用程序，然后就可以启动Web服务器了。这类服务器的工作是接收客户请求、向（Web）客户（比如用户计算机上的浏览器）发送Web网页，然后等待下一个客户请求。这些服务器在启动时都抱着“永远运行”的想法，即使它们达不到这个目标，也会尽最大努力尽量运行得长久一些，直到受到某些外来的力量（比如明确的关闭动作和灾难性的硬件故障等）的影响才会停止。

数据库服务器是另外一种软件服务器。它们接收客户的存储或检索请求、完成相应的服务，然后等待更多的业务。它们的设计目标也是“永远”运行。

我们将要讨论的最后一种软件服务器是窗口服务器。这些服务器几乎可以看做是硬件服务器。它们运行在一台带有显示设备的计算机上，比如某种型号的监视器等。窗口客户实际上是一些要求运行在一个窗口化环境里的程序，通常被看做是一些图形化用户操作界面（GUI）下的应用程序。如果是在没有窗口服务器的情况下（比如在一个DOS窗口或一个UNIX中的shell等文本环境里）执行它们，它们根本就无法启动。只有在能够访问一个窗口服务器的前提下，它们才会行动起来。

当网络出现在舞台上以后，这类环境就变得更有意思了。一个窗口客户通常使用的显示设备就是同一台本地机器上的服务器，但在某些网络化的窗口环境中（比如X窗口系统）允许选择另外一台机器中的服务器做为其显示设备。在这种情况下，你可以在一台计算机上运行一个GUI程序，但在另外一台计算机上看到它的显示情况。

3. 服务器是银行出纳员？

理解客户-服务器体系结构工作原理的办法之一是在自己的脑子里勾画出一个银行出纳员的形象：他不吃不睡不休息，依次向排在一条看不到尽头的队伍里的顾客提供服务（如图16-2所示）。这个队伍时短时长，有的时候还会是空的；但任何时刻都可能会出现一位顾客。当然，这样一个出纳员几年前还是个天方夜谭，但现今的自动柜员机（ATM机）和这种模型就非常接近了。

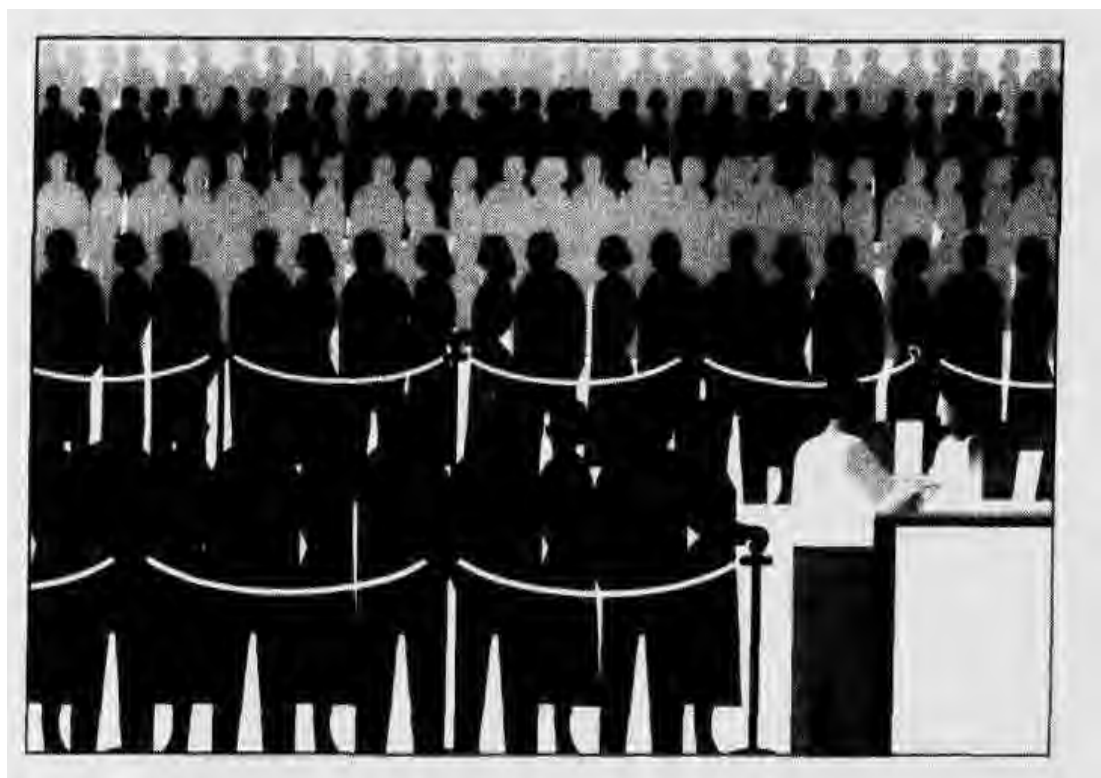


图16-2 这个示意图里的出纳员会“永不停息”地为客户请求提供服务。这个出纳员运行在一个无限循环里，接受请求、提供服务，然后再去服务或等待下一位顾客。

顾客队伍可能会很长，也可能是空的；但不论哪种情况，出纳员的工作是不会停止的

这个出纳员当然就是运行在一个无限循环里的服务器了。每个顾客都是一个有服务需求的客户。顾客是按照先来先服务的方式到达和被出纳员服务的。一旦某项交易结束，客户就立刻离开，而服务器将服务下一个客户或等待其他客户上门。

这为什么如此重要呢？因为这种执行方式就是客户-服务器体系结构正常工作原理。有了这个基本概念，我们再把它结合到网络程序设计中去，网络程序设计符合软件客户-服务器体系结构模型。

16.1.2 客户-服务器网络程序设计

在开始提供任何服务之前，服务器必须先完成一些基本的设置过程，为将要到来的工作做好准备。需要创建一个通信端点，这样服务器才能“监听”有无请求的到来。我们可以把服务器比做一个公司的前台接待员或者比做一个应答公司总线电话的总机接线员。一旦电话号码和有关设备安装好并且接线员也到岗之后，服务就可以开始了。

网络世界里也基本上是同样的过程——只要建立起通信端点，我们的监听服务器就可以进入它的无限循环以等待客户的连接并提供相应的服务了。当然，我们还必须记得要把电话号码印在公司的信笺上、广告上，或者其他传播媒介上；否则是没有人会打电话来的！

与此对应的是，必须通知潜在的客户这个服务器已经设置好，可以处理它们的请求了——否则这个服务器是收不到请求的。请想象一下建立一个全新的Web网站：它可能是最好的、最精彩的、最吸引人的、最有用和最酷的Web站点，但如果Web网站的地址或者统一资源定位器

(Universal Resource Locator, URL) 从来没有广告或者介绍, 那就不会有人知道它, 它的前途肯定会是暗淡无光的。同样的情况也会发生在一个新的公司总部电话号码身上。如果公众不知道这个电话号码, 它就不会接到任何呼叫。

现在大家应该对服务器有一个比较清楚的认识了。你已经度过了最困难的部分。客户那一头的东西要比服务器这头简单多了。客户要做的全部工作只不过是创建一个它自己的通信端点, 再建立一个到服务器的连接而已。客户现在就可以发出它们的请求, 其中还包括必要的数据交换。一旦请求被服务, 客户收到某种形式的结果或通知, 这次通信就结束了。

16.2 套接字: 通信端点

16.2.1 什么是套接字

套接字是计算机上网络数据结构中的一种, 它是上一节描述的“通信端点”概念的具体体现。网络化的应用程序必须先创建出套接字才能开始进行任何形式的通信。它们就像是电话线的插口, 如果不把电话线连接到插口上是不可能进行通话的。

套接字的概念最早起源于加利福尼亚州立大学伯克利分校于1970年推出的UNIX操作系统版本, 它也叫做BSD UNIX。因此, 有时候你会听到别人把套接字称为“伯克利套接字”或“BSD套接字”。创建套接字的初衷是为了让同一台主机上的应用程序能够让一个运行中的程序(即进程)与另外一个运行中的程序进行通信。这也叫做“进程间通信”或简称为“IPC”。

有趣的是, 套接字的发明早于网络的出现。不管你以前是如何听说的, 套接字其实并不是专门为网络应用程序而存在的。那些最早的套接字至今仍在使用, 它们被称为UNIX套接字, 它们的“家族名”叫做AF_UNIX, 意思是“地址家族: UNIX”(大多数流行的计算机平台, 包括Python在内, 使用的都是术语“地址家族”和缩写“AF”, 其他系统则可能把地址家族叫做“域”或者“协议家族”并使用“PF”代替“AF”)。因为两个进程都运行在同一个机器上, 所以这些套接字都是基于文件的; 也就是说, 它们内在的内部结构都是由文件系统支持的。这很有道理, 因为文件系统确实是同一台主机上不同进程之间能够共享的固定目标。

当网络(通过因特网协议Internet Protocol, IP)出现在人们面前时, 研究者们坚信进程间通信依然会发生, 但与其把通信局限于运行在同一台机器上的两个应用程序之间, 为什么不让一台机器上的某个进程能够与另一台机器上的某个进程通信呢? 这些新的、网络化的套接字有它们自己的家族名AF_INET, 即“地址家族: Internet”。还有其他的地址家族, 它们或者是非常专业化, 或者已经被淘汰, 或者很少使用, 或者还没有被实现。就全体地址家族来说, AF_INET现在是应用最广的。Python语言目前只支持AF_UNIX和AF_INET家族。因为我们的焦点是网络程序设计, 所以在本章的大部分内容里我们使用的都是AF_INET。

16.2.2 套接字地址: 主机加端口

如果把套接字比做是一个电话线插口——即一个实现通话的内部结构, 那一个主机名加端口号就好比是长途区号和电话号码的一个组合。如果不知道要给谁打电话、往哪儿打电话, 那么, 光有硬件设备和通信能力是没有实际用处的。一个因特网地址包括一个主机名和一个端口

号，而网络上的通信正需要一个这样的地址。在进行网络通信的时候对方必须有人监听；如果不是这样，打个比方说，你就会听见熟悉的电话提示音和“对不起，该电话已经不再使用，请核对您的号码后重新拨叫”的提示。而在网上冲浪时你也肯定见过这样的情形：屏幕提示“无法连接服务器。服务器未响应或连接不到。”

合法的端口号码其范围是0到65535，那些小于1024号的端口是保留给系统使用的。如果读者使用的是一台UNIX系统，那就可以在/etc/services文件里找到一个保留端口号的清单。常用端口号的清单可以在下面的Web站点上查到：

<http://www.isi.edu/in-notes/iana/assignments/port-numbers>

16.2.3 面向连接方式和无连接方式

1. 面向连接方式

不管你使用的是哪一种地址家族，都会有两种套接字连接方式。第一种是面向连接方式。它的基本含义是在通信发生之前必须先建立起一个连接，就像通过电话系统呼叫一个朋友那样。这种通信方式又叫做“虚拟线路”或“流式套接字”。

面向连接方式的通信在收发数据时具有顺序性、可靠性、无副本等特点，数据各部分之间也没有明显的记录边界。也就是说，每条消息都可以拆成多个小的信息块，而所有的小信息块都能够保证到达（“只到达一次”——也就是数据既不会丢失也不会多出副本来）它们的目的地，然后按原来的顺序拼接起来并送入等待着的应用程序去。

实现这种连接方式的基本协议是传输控制协议（Transmission Control Protocol, TCP）。要想建立TCP套接字，必须把准备创建的套接字的类型指定为SOCK_STREAM。之所以给TCP套接字起名为SOCK_STREAM是因为TCP套接字还有一个名字是“流式套接字”（stream socket）。又因为这些套接字使用因特网协议（Internet Protocol, IP）在网络中寻找主机，所以在提到这样的一个完整系统时一般都会把两个协议组合在一起而称之为“TCP/IP”系统。

2. 无连接方式

与虚拟线路类型的套接字相对立的是数据图（datagram）类型的套接字，它是一种无连接方式的套接字。这意味着在开始通信之前不需要先建立一个连接。此时，数据收发过程中将不保证顺序性、可靠性和无副本等特性。数据图的各部分之间有记录边界，也就是说，消息不会根据面向连接方式的协议中的原则被拆成小的信息块，它们是做为一个整体一起被发送和接收的。

使用数据图收发消息可以比做邮递服务。信件和包裹可能不会按它们投递时的先后顺序到达。事实上，它们还可能会到不了呢！而网络还增加了事情的复杂性，因为网络上的消息很容易就会多出几个副本来。

既然它有这么多的缺点，那为什么还要使用数据图呢？（肯定要有一些超过使用流式套接字的优点！）原因是这样的：虽然面向连接的套接字能够在许多方面做出很好的保证，但这些保证在安装设置时需要大量的工作，在维护虚拟线路的连接方面也需要大量的工作。而数据图就不需要这么复杂繁重的设置和维护工作，因而“成本比较低”一些，执行效率也会好一些，对某些类型的应用程序来说还是很适合的。

实现这种连接方式的基本协议是用户数据图协议（User Datagram Protocol, UDP）。要想建

立UDP套接字，必须把准备创建的套接字的类型指定为SOCK_DGRAM。读者可能已经猜到之所以给UDP套接字起名为SOCK_DGRAM就是因为“datagram”（数据图）这个单词。因为这个协议也使用因特网协议在网络中寻找主机，所以在提到这样的完整系统时也有一个更常用的名字，就是把两个协议组合在一起而称之为“UDP/IP”系统。

16.3 使用Python语言进行网络程序设计

在向大家介绍了客户-服务器体系结构、套接字和套接字的网络连接方式之后，我们现在来看看Python语言是如何运用这些概念的。这一节使用的基本模块是socket模块。在这个模块里有一个socket()函数，我们就要用它来创建套接字(socket)对象。套接字也有它们自己的方法，这些方法使我们能够应用基于套接字的网络通信。

16.3.1 socket()模块函数

要想创建一个套接字，必须使用socket.socket()函数，它的通用语法如下所示：

```
socket(socket_family, socket_type, protocol = 0)
```

其中的socket_family或者是AF_UNIX，或者是AF_INET，具体含义前面已经讲过了；socket_type或者是SOCK_STREAM，或者是SOCK_DGRAM，具体含义前面也已经讲过了。protocol一般都空着，就使用它的缺省值0。

因此，如果想创建一个TCP/IP套接字，就要像下面这样调用socket.socket()函数：

```
tcpSock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

同样地，创建一个UDP/IP套接字要像下面这样：

```
udpSock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

因为socket模块有太多的属性，所以在这里使用“from module import *”是能够被接受的。如果使用了“from socket import *”，就等于把套接字的属性都添加到我们的名字空间里来了，但我们的代码也因此能够简短一些，如下所示：

```
tcpSock = socket(AF_INET, SOCK_STREAM)
```

创建出一个套接字对象之后，以后的操作就都要通过使用这个套接字对象的方法来实现了。

16.3.2 套接字对象的内建方法

我们在表16-1里列出了一些最常用的套接字方法。在下一节里，我们将使用这里列出的方法分别创建出TCP和UDP客户和服务端来。虽然我们的注意力主要集中在因特网套接字上，但这些方法的含义在它们和UNIX套接字一起使用的时候是相似的。

表16-1 常用套接字对象的方法

方 法	说 明
服务器的套接字方法	
s.bind()	把地址（主机和端口号）绑定到套接字上去

(续)

方 法	说 明
<code>s.listen()</code>	设置和启动TCP监听程序
<code>s.accept()</code>	被动接收TCP客户的连接; 等待连接的到来 (或阻断)
客户套接字的方法	
<code>s.connect()</code>	主动对TCP服务器的连接进行初始化
通用套接字的方法	
<code>s.recv()</code>	接收TCP消息
<code>s.send()</code>	传输TCP消息
<code>s.recvfrom()</code>	接收UDP消息
<code>s.sendto()</code>	传输UDP消息
<code>s.close()</code>	关闭套接字

16.3.3 创建一个TCP服务器

我们先给出创建一个普通TCP服务器的通用伪代码, 然后对它的情况做一个基本说明。需要提醒大家的是这只是设计你服务器的办法之一。一旦读者熟悉了服务器的设计, 就可以对这段伪代码加以修改, 让它按照自己的意愿完成工作:

```

ss = socket()           # create server socket
ss.bind()               # bind socket to address
ss.listen()             # listen for connections
inf_loop:               # server infinite loop
    cs = ss.accept()     # accept client connection
    comm_loop:           # communication loop
        cs.recv()/cs.send() # dialog (receive/send)
    cs.close()           # close client socket
ss.close()              # close server socket

```

所有套接字都是用`socket.socket()`函数创建的。服务器需要“坐在一个端口上”等待请求的到来, 所以必须把它们“绑定”到一个本地地址上去。因为TCP是一个面向连接的通信系统, 所以在一个TCP服务器能够开始工作之前必须对其内部先做一些设置工作。特别地, TCP服务器必须“监听”着有无(进入的)请求。这个设置过程完成之后, 该服务器就可以开始它的无限循环了。

接下来, 一个简单的(单线程)服务器会通过`accept()`调用等待一个连接的到来。在缺省的情况下, `accept()`是处于“阻断”状态的; 也就是说, 它的执行在有连接到来之前是处于挂起状态的。套接字还有一个非阻断状态, 至于为什么要使用非阻断套接字以及如何使用非阻断套接字方面的内容, 请参考有关文档和操作系统方面的教科书。

如果`accept()`接收到一个连接, 它就会另外返回一个客户套接字供下一步交流消息使用。使用一个新的客户套接字类似于把一个顾客打来的电话转交给一位服务代表。当真的有一位顾客到来的时候, 总机接线员会把这个呼叫通过另外一条线路转接给处理顾客需求的那个适当的人。

这样就可以解放电话总线——即原来的服务器套接字, 接线员去继续等待新的呼叫(即客户请求), 让顾客和服务代表去继续他们之间的交谈。同样地, 当再有一个新的呼叫到达的时候,

会再创建一个新的通信端口与新的顾客直接交谈，总机线路还是要解放出来以便接收下一位客户的连接。

编程小技巧：“繁殖”线程以处理客户的请求

[CT]

我们没有在我们的例子里实现这一功能，但把一位客户的请求交给一个线程（即新的进程）去完成对客户处理也是很常见的做法。在socket模块顶部还有一个高级套接字通信模块SocketServer模块，它在处理客户请求的时候支持线程化和“繁殖”出来的进程。如果读者想进一步了解关于SocketServer模块和第17章多线程程序设计里面的练习方面的资料，请参考有关文档。

在那个临时性的套接字被创建出来以后，通信就可以开始了，客户和服务端两者一块进入一个对话过程，在这个过程里它们使用这个新的套接字收发数据直到连接的结束。连接结束通常发生在对话双方中的一方关闭了连接或者向对方发送了一个空字符串的时候。

在我们的代码里，在一个客户连接被关闭的时候，服务器就返回到自己的开始去等待下一个客户连接。我们代码的最后一行是关闭那个服务器套接字，但这一行是永远也执行不到的，因为我们设定它运行在一个无限循环里。我们在代码里加上这条语句是为了提醒读者在为服务器设计一个智能化的退出机制时（比如当处理器检测到迫使服务器关闭的一些外部条件）使用close()方法是推荐做法。而此时的最佳方案就是使用这个close()方法。

程序示例16-1里给出的是一个名为tsTserv.py的服务器程序的代码，它从一个客户那里接收到一个data字符串，加上时间标签后（格式为：“[timestamp] data”）再返回给那个客户。（tsTserv是timestamp（时间标签）、TCP、server（服务器）等几个单词的首字母组合。另外一个文件的命名方式也是如此。）

程序示例16-1 TCP时间标签服务器（tsTserv.py）

创建一个TCP服务器，它接收来自客户的消息，给消息加上一个时间标签前缀后再返回给它们。

```

1  #!/usr/bin/env python
2
3  from socket import *
4  from time import time, ctime
5
6  HOST = ''
7  PORT = 21567
8  BUFSIZ = 1024
9  ADDR = (HOST, PORT)
10
11 tcpSerSock = socket(AF_INET, SOCK_STREAM)
12 tcpSerSock.bind(ADDR)
13 tcpSerSock.listen(5)
14
15 while 1:
16     print 'waiting for connection...'
17     tcpCliSock, addr = tcpSerSock.accept()
18     print '...connected from:', addr
19
20     while 1:
21         data = tcpCliSock.recv(BUFSIZ)
22         if not data: break
23         tcpCliSock.send(['[ts] %s' % \
24             (ctime(time()), data)])
25
26     tcpCliSock.close()
27 tcpSerSock.close()

```


逐行解释

第1~4行

在UNIX操作系统的启动行后面，我们导入了time.time()、time.ctime()以及socket模块中的全部属性。

第6~13行

HOST变量是空的，表示bind()方法可以使用任何一个可用的地址。端口号是我们随机选择的，但它既不是系统使用的，也不是由系统保留的。我们在这个程序里把缓冲区的大小设置为1K。读者可以根据自己网络的能力和应用程序的要求改变这个值。listen()方法的参数是进入连接的最大数目；超过这个数字之后，新的连接请求将被拒绝。

TCP服务器套接字(tcpSerSock)是在第11行建立的，它后面是把这个套接字绑定到服务器地址并启动该TCP监听程序的函数调用。

第15~27行

进入服务器的无限循环之后，我们（被动地）等待连接的到来。当有连接到来的时候，我们进入对话循环等待客户发送它的消息给我们。如果消息是空的，就意味着客户已经退出了；所以我们也中断对话循环，关闭客户连接，回到开头去等待另外一个客户。如果我们确实接收到一条来自客户的消息，就在这条消息的前面加上当前时间标签后返回给那个客户，数据部分保持不变。最后一行（第27行）是永远也执行不到的，把它放在那里是为了提醒用户如果编写了一个更加体面的退出机制，就应该调用close()——就像我们前面讨论过的那样。

16.3.4 创建一个TCP客户

创建一个客户要比创建一个服务器简单多了。类似于我们对TCP服务器的描述，我们先给出的也是一段伪代码和对它的解释，然后再把真家伙拿出来。如下所示：

```
cs = socket()           # create client socket
cs.connect()            # attempt server connection
comm_loop:              # communication loop
    cs.send()/cs.recv()  # dialog (send/receive)
cs.close()               # close client socket
```

正如我们前面提到过的，所有的套接字都是用socket.socket()创建出来的。一旦某个客户拥有了一个套接字，它就可以立刻使用该套接字的connect()方法向某个服务器发出一个连接。在连接建立好以后，它就可以和服务器一起进入对话过程。当该客户结束了它的交易之后，它就可以关闭它的套接字，结束这次连接。

我们把tsTclnt.py的代码列在程序示例16-2里。这个脚本程序的作用是连接到服务器并提示用户一行一行地输入数据。服务器将返回加上了时间标签的数据，这些从服务器返回的数据再由这个客户程序的代码显示给用户。

程序示例16-2 TCP时间标签客户 (tsTclnt.py)

创建一个TCP客户，它提示用户输入消息并发送给服务器，收到加上时间标签前缀的消息，再把从服务器返回的结果显示给用户。

```
1  #!/usr/bin/env python
```

```
2
3  from socket import *
4
5  HOST = 'localhost'
6  PORT = 21567
7  BUFSIZ = 1024
8  ADDR = (HOST, PORT)
9
10 tcpCliSock = socket(AF_INET, SOCK_STREAM)
11 tcpCliSock.connect(ADDR)
12
13 while 1:
14     data = raw_input('> ')
15     if not data: break
16     tcpCliSock.send(data)
17     data = tcpCliSock.recv(1024)
18     if not data: break
19     print data
20
21 tcpCliSock.close()
```

逐行解释

第1~3行

在UNIX操作系统的启动行后面，我们导入了socket模块中的全部属性。

第5~11行

HOST和PORT变量指的是服务器的主机名和端口号。因为我们（在这个例子里）是在同一台机器上进行我们的测试，所以HOST变量里包含的是本地主机名（如果读者是在另外一台机器上运行着自己的服务器，请做相应的修改）。端口号PORT必须与我们为服务器设置的端口号完全一致（否则就不会有那么多通信了！）缓冲区我们选得和服务器的上的一样，都是1K。

TCP客户套接字（tcpCliSock）是在第10行建立的，它后面是连接服务器的调用。

第13~21行

这个客户也有一个无限循环，但它并不会像服务器循环那样永远运行。如果遇到下面两个条件之一，客户循环就会退出：如果用户不再有任何输入（第15行）；或者如果因为某些原因服务器退出了运行，从而使我们的recv()调用失败了（第18行）。如果两种情况都没有发生，那么在正常情况下，用户需要输入一些字符串数据，这些数据将被发送到服务器去进行处理。处理后的输入字符串将带着新添加上的时间标签返回并显示在屏幕上。

16.3.5 执行TCP客户-服务器应用程序

现在我们来运行服务器和客户程序，看看它们的工作情况是怎样的。我们应该先运行服务器还是先运行客户呢？很自然，如果我们先运行客户，就不可能有任何连接，因为没有服务器接收它发出的请求。服务器之所以被看做是“被动方”。就是因为它必须先把自己设置好，然后被动地等待连接的到来。而做为另一方的客户是一个“主动”方，因为是它主动开始一个连接的。换句话说：（在任何客户试图连接之前）必须先启动服务器。

在运行客户和服务器的这个例子里，我们使用的是同一台机器，但这并不是说我们不能使用另外一台主机做为我们的服务器。如果真的是在另外一台主机上运行的服务器，也只要修改

一下主机名就行了。(如果读者的第一个网络应用程序是在不同的机器上分别运行服务器和客户的话,那可够刺激的!)

下面是客户程序方面相应的(输入和)输出,要想退出客户程序,不输入任何数据直接按下回车键就行。请看:

```
% tsTelnt.py
> hi
[Sat Jun 17 17:27:21 2000] hi
> spanish inquisition
[Sat Jun 17 17:27:37 2000] spanish inquisition
>
%
```

服务器的输出主要具有诊断意义:

```
% tsTserv.py
waiting for connection...
...connected from: ('127.0.0.1', 1040)
waiting for connection...
```

当客户开始建立连接的时候,我们会看到“...connected from...”消息。在客户继续接受服务的同时,服务器已经去等待新的客户了。当我们从服务器退出的时候,我们必须从它的无限循环中挣脱出来,这样会导致一个例外。避免这种错误的最佳办法是建立一个更体面的退出机制,这一点我们前面已经讨论过了。

编程技巧: 体面地退出和调用服务器的close()方法。

[CT]

建立“友好”退出机制的办法之一是把服务器的while循环放到一个try-except语句的try从句里去,对EOFError和KeyboardInterrupt例外进行监控。然后,在相应的except从句里,你可以发出一个关闭服务器套接字的调用。

16.3.6 创建一个UDP服务器

UDP服务器要求的设置工作不像TCP服务器那么多,这是因为前者并不是面向连接的。事实上,除了等待进入的连接外,不需要再做什么其他的工作了。

```
ss = socket()                # create server socket
ss.bind()                    # bind server socket
inf_loop:                     # server infinite loop
    cs = ss.recvfrom()/ss.sendto() # dialog (receive/send)
ss.close()                    # close server socket
```

从上面的伪代码里可以看出,除了创建套接字并把它绑定到本地地址(主机名/端口号)以外没有什么额外的工作。它的无限循环里包括从某个客户那里接受消息、返回加上时间标签后的原消息和继续等待下一条消息。再说一次,因为是无限循环,所以close()方法是永远也执行不到的,把它放在那里是为了提醒大家它应该是我们前面提到过的体面或智能化的退出机制的组成部分。

UDP和TCP服务器之间的一个明显差异是: 因为数据图套接字是无连接类型的,所以成功的

通信并不需要把某个客户连接“转交”给另外一个服务器套接字。这些服务器只是接收消息并尽可能回复就行了。

tsUser.py的代码列在程序示例16-3里，它是前面那个TCP服务器程序的UDP版本。它接收来自客户的一条消息，加上时间标签后返回给那个客户。

程序示例16-3 UDP时间标签服务器 (tsUser.py)

创建一个UDP服务器，它接收来自客户的消息，给消息加上一个时间标签前缀后再返回给它们。

```

1  #!/usr/bin/env python
2
3  from socket import *
4  from time import time, ctime
5
6  HOST = ''
7  PORT = 21567
8  BUFSIZ = 1024
9  ADDR = (HOST, PORT)
10
11 udpSerSock = socket(AF_INET, SOCK_DGRAM)
12 udpSerSock.bind(ADDR)
13
14 while 1:
15     print 'waiting for message...'
16     data, addr = udpSerSock.recvfrom(BUFSIZ)
17     udpSerSock.sendto('%s [%s]' % (
18         ctime(time()), data), addr)
19     print '...received from, returned to:', addr
20
21 udpSerSock.close()

```

逐行解释

第1~4行

在UNIX操作系统的启动行后面，我们导入了time.time()、time.ctime()以及socket模块中的全部属性。这与TCP服务器的设置工作完全一样。

第6~12行

HOST和PORT与前面一样，原因也是一样。对socket()的调用只有一个小区别，就是现在申请的是一个数据图/UDP套接字类型；但bind()的调用就与TCP服务器版本中一样了。同样地，因为UDP是无连接类型的，所以这里就不再需要“调用listen()监听到来的连接”了。

第14~21行

进入服务器的无限循环后，我们（被动地）等待连接的到来。当有连接到来的时候，我们对它进行处理（给它加上一个时间标签），把它发送回去，再回到开头去等待另外一条消息。套接字的close()方法只在那里摆个样子，原因同上。

16.3.7 创建一个UDP客户

这一节介绍了四个程序，而这里的UDP客户其代码是最短的。请看如下所示的伪代码：

```

cs = socket()           # create client socket
comm_loop:              # communication loop
    cs.sendto()/cs.recvfrom() # dialog (send/receive)

```

```
cs.close() # close client socket
```

创建好套接字对象以后，我们进入与服务器交换消息的对话循环。当通信结束时，这个套接字也就被关闭了。

这个客户程序的实际代码tsUclnt.py列在程序示例16-4里。

程序示例16-4 UDP时间标签客户 (tsUclnt.py)

创建一个UDP客户，它提示用户输入消息并发送给服务器，收到加上时间标签前缀的消息，再把从服务器返回的结果显示给用户。

```
1  #!/usr/bin/env python
2
3  from socket import *
4
5  HOST = 'localhost'
6  PORT = 21567
7  BUFSIZ = 1024
8  ADDR = (HOST, PORT)
9
10 udpCliSock = socket(AF_INET, SOCK_DGRAM)
11
12 while 1:
13     data = raw_input('> ')
14     if not data: break
15     udpCliSock.sendto(data, ADDR)
16     data, ADDR = udpCliSock.recvfrom(BUFSIZ)
17     if not data: break
18     print data
19
20 udpCliSock.close()
```

逐行解释

第1~3行

在UNIX操作系统的启动行后面，我们导入了socket模块中的全部属性。这与TCP客户版本中的设置工作完全一样。

第5~10行

因为我们还是在同一台机器上运行的服务器，所以我们在客户端使用的仍然是“本地主机”和同样的端口号，大小为1K的缓冲区也是老样子。建立UDP套接字对象的办法与UDP服务器中是一样。

第12~20行

我们UDP客户的工作情况与TCP客户的情况几乎是完全一样的。两者之间唯一的差异是：我们不必事先建立一个到UDP服务器的连接，只要发送一条消息再等待服务器的应答就完事了。当加上时间标签的字符串返回以后，我们把它显示到屏幕上，然后继续输入更多的字符串。当所有输入都处理完之后，我们跳出循环并关闭那个套接字。

16.3.8 执行UDP客户-服务器应用程序

UDP客户的操作行为和TCP客户完全一样：

```
% tsUclnt.py
> hi
[Sat Jun 17 19:55:36 2000] hi
```

```
> spam! spam! spam!
(Sat Jun 17 19:55:40 2000) spam! spam! spam!
>
%
```

服务器的情况也差不多：

```
% tsUser.py
waiting for message...
...received from and returned to: ('127.0.0.1', 1025)
waiting for message...
```

事实上，之所以要让客户输出信息是因为我们可以从多个客户那里接收消息和返回应答，而客户的输出就可以告诉我们消息来自哪一个客户。对TCP服务器来说，我们知道消息是从哪里来的，因为每个客户都有它自己建立的连接。请注意服务器给出的输出说的是“waiting for message”（等待消息）而不是“waiting for connection”（等待连接）。

16.3.9 其他socket模块函数

除了创建套接字对象的`socket.socket()`函数以外，`socket`模块还有一整套辅助性的函数可以帮助程序员设计网络化的应用程序，请参考表16-2。

表16-2 其他socket模块函数

函数名称	说 明
<code>fromfd()</code>	从一个打开的文件描述符里创建一个套接字
<code>gethostname()</code>	返回当前的主机名
<code>gethostbyname()</code>	把一个主机名映射到它的IP编号上去
<code>gethostbyaddr()</code>	把一个IP编号或主机名映射到DNS资料上去
<code>getservbyname()</code>	把一项服务的名字和一个协议的名字映射到一个端口号上去
<code>getprotobyname()</code>	把一个协议的名字（如'tcp'）映射到一个数字上去
<code>ntohl()/ntohs()</code>	把来自网络的整数转换为主机字节顺序
<code>htonl()/htons()</code>	把来自主机的整数转换为网络字节顺序
<code>inet_aton()</code>	把IP地址的八进制字符串形式转换为32位紧凑格式
<code>inet_ntoa()</code>	把32位紧凑格式转换为IP地址字符串
<code>ssl()</code>	安全套接字层（Secure Socket Layer）支持（必须加以配置）；1.6版本新增功能

进一步资料请参考《Python Library Reference》（Python库大全）中的`socket`模块文档。

16.4 相关模块

表16-3列出了一些与网络和套接字程序设计有关的其他Python模块。在开发套接字底层应用程序的时候，通常要把`select`模块和`socket`模块一起使用。它提供的`select()`函数可以对一组套接字对象进行管理。它最有用的功能是能够在—组套接字上监听活跃的连接。除非至少有一个套接字处于通信就绪状态，否则`select()`函数将呈阻断状态；如果有处于就绪状态的套接字，`select()`能够通知你哪些个套接字已经处于读操作就绪状态（它也可以确定哪些个套接字处于写操作就绪状态，但这在普通操作情况下很少会用到）。

表16-3 与网络/套接字程序设计有关的模块

模块名称	说 明
asyncore	提供了创建能够以异步方式对客户进行处理的网络化应用程序的内在结构
select	在一个单线程网络服务器应用程序里管理多个套接字连接
SocketServer	高级模块，为网络化应用程序提供了服务器类（class），这些类是用进程分支（forking）或线程变异（threading variants）实现的

就服务器涉及到的范围来说，asyncore和SocketServer模块都提供了高级的功能。这两个模块都是在socket和/或select模块的顶部编写的，能够加快客户-服务器系统的开发进度，因为它们能够替程序员处理好底层的代码。需要程序员做的只是从相应的基类（base class）里构造或分离出适当的子类，然后就可以继续进行了。正如我们前面曾经提到过的，SocketServer模块甚至提供了把线程即新进程集成到服务器里去的能力，这样就可以对客户请求进行并行处理。

我们在这一章里讨论的问题包括在Python语言里使用套接字进行网络程序设计和如何使用TCP/IP和UDP/IP等底层协议创建定制的应用程序等两个方面。如果读者想要开发高水平的Web和因特网应用程序，我们强烈建议你学习第19章内容。

16.5 练习

16-1 套接字。面向连接的套接字和无连接套接字有什么区别？

16-2 套接字。TCP和UDP有什么区别？

16-3 套接字。在TCP和UDP系统里，哪种类型的服务器会接受连接并把它们转交给其他的套接字去与客户进行通信？

16-4 客户。改进TCP（tsTclnt.py）和UDP（tsUclnt.py）客户，使服务器的名字不再是应用程序中的硬代码。允许用户指定一个主机名和端口号；并且只有在用户没有给出或少给出了这两个参数时使用缺省值。

16-5 网络与套接字。请完成在《Python Library Reference》（Python库大全）7.2.2节里由Guido给出的TCP客户/服务器示范程序，让它们能够运行。先设置好服务器，再设置好客户。它们源代码的在线版本可以在下面的网页上找到：

http://www.python.org/doc/current/lib/Socket_Example.html

你觉得那个服务器功能少了点。请改进那个服务器，让它能多干些事情，能够识别出下面的命令：

date 服务器将返回它的当前日期/时间标签，即time.ctime(time.time())。

os 获取关于操作系统的信息（os.name）。

ls 列出当前目录中的文件清单（提示：os.listdir()列出一个目录里的文件清单；os.curdir是当前目录）。附加题：接受“ls dir”形式的命令并返回目录dir的文件清单。

16-6 报时服务。在UDP协议下，用socket.getservbyname()来确定“daytime”（报时服务）所使用的端口号。请查阅有关文档了解getservbyname()的准确语法（即查看文档字符串socket.getservbyname.__doc__）。编写一个程序，它发送一个哑消息并等待服务器的回复。当你从服务器接收到一个回复时，把它显示在屏幕上。

16-7 半双工聊天。编写一个简单的半双工聊天程序。“半双工”的意思是当建立起连接并开始服务之后，只有一个人可以打字。其他聊天者在提示他或她输入消息以前只能等待阅读一条消息。当消息送出之后，发消息的那个人必须等到有回音之后才能再被允许发送出另外一条消息。一个聊天者在服务器端，其他聊天者都在客户端。

16-8 全双工聊天。改进上一个练习中的解决方案，使你的聊天服务现在是全双工的，也就是聊天双方可以彼此独立地发送和接收聊天消息。

16-9 多用户全双工聊天。进一步改进你的解决方案，使你的聊天服务成为多用户的。

16-10 多用户多聊天室全双工聊天。现在把你的聊天服务改进为多用户和多聊天室的。

16-11 Web客户。编写一个TCP客户程序，它可以连接到你最喜欢的Web站点的80号端口（不要“http://”和其他尾缀信息，只使用主机名）。建立一个连接之后，发送HTTP命令字符串“GET / \n”并把服务器返回的数据写到一个文件里去。（GET命令的作用是检索一个Web主页；“/”给出的是将要检索的文件；而“\n”把命令发送给服务器。）

查看检索到的文件的内容。它是什么？如何检查确认你接收到的数据是正确的？（注意：你也许需要在命令行上给出一个或两个换行符，一般一个就够用。）

16-12 催眠服务器。编写一个“催眠”服务器。客户会要求把自己“催眠”几秒钟。这个服务器会应客户的请求发出“催眠”命令，然后返回一条消息给客户表示操作成功。客户将休眠并按请求的时间准确保持休眠状态。这是一个“远程过程调用”的简单例子，即一个客户的请求可以通过网络调用另外一台机器上的命令。

16-13 域名服务器。设计并实现一个域名字服务器。这个服务器将负责管理维护一个由主机名和端口号构成的数据库；也许还可以加上该服务器所提供的各项服务的字符串介绍。请对一两个现有的服务器进行测试，让它们把它们的服务“注册”到你的域名服务器上。（注意，在这个例子里，这些服务器是域名服务器的客户）。

每个启动了的客户都不知道它们要寻找的服务器在哪里。这些客户同时也是你域名服务器的客户，它们将向域名服务器发出一个请求，告诉你它们正在寻求哪一种服务。在你域名服务器的回复里，向这些客户返回一个主机名-端口号组合，客户再根据你服务器的回复连接到相应的服务器去处理它的请求。

附加题：（1）在你的服务器上为经常出现的请求提供缓冲功能；（2）给你的服务器加上登录记录功能，记录下哪些个服务器已经注册、客户都请求了哪些服务项目；（3）你的服务器应该定期“ping”一下已注册服务器的各个注册端口以确定确实有该项服务。如果“ping”操作多次失败，将导致一个服务器从服务清单被撤下来。

你可以把服务器注册在你域名服务器上的服务具体实现出来，也可以使用哑服务器（即只确认有一个请求）。

第17章 多线程程序设计

在本节里，我们将探索运用Python语言提供的多线程（multithreaded，简称MT）程序设计功能使你的代码具备更多并行运行能力的多种办法。在本章的前几节里，我们先从线程和进程之间的区别讲起。然后介绍多线程程序的编写方法。（那些熟悉MT程序设计的读者可以直接跳到17.3.5节。）本章的最后一节给出了一些如何运用threading和Queue模块在Python里实现MT程序设计的例子。

17.1 介绍

在多线程（multithreaded，简称MT）程序设计出现以前，计算机程序的执行是由一系列单个的步骤组成的，这些单个的步骤在主机中央处理器（CPU）上也是按照先后的顺序来执行的。不管任务本身就要求按先后顺序来执行还是整个程序实际是由多个子任务所组成，这样的执行情况都是很正常的。如果这些子任务是彼此独立的，没有任何相互影响的联系（即子任务的执行结果不影响其他子任务的输出），有没有需要改进的地方呢？如果真是这样的话，让这些彼此独立的任务同时运行不是更合理吗？这种并行处理能够大幅度提高整个任务的执行性能，而这也正是MT程序设计所关心的。

MT程序设计对某些程序任务很适用，这些程序任务或者本身就不需要按照先后顺序执行，要求同时有多个共发的活动，而每个活动的处理也是不确定的——即随机的和不可预见的。这些程序任务可以被组织或划分为多个执行流，每个执行流都有它特定要完成的任务。根据应用程序的不同，这些子任务可能会是一些计算中间结果的工作，最后它们再组合为一个最终的输出。

把与CPU直接有关的任务划分为子任务并按顺序或以多线程方式执行还是比较容易的，但管理一个具有多个外部输入资源的单线程进程就不那么简单了。如果想在不使用多线程的前提下完成一个这样的程序任务，一个顺序执行的程序就需要使用多个定时器并形成一种极其复杂的机制。

一个顺序执行的程序需要查询每一个I/O（输入/输出）终端通道以查看有没有用户的输入；同时，程序在读取I/O终端通道时不发生阻塞也是非常重要的——这是因为用户输入的到来与否是不可预见的，而阻塞将影响到对其他I/O通道的处理。这个顺序执行的程序必须使用非阻塞I/O或者在用到阻塞I/O时加上一个定时器（这样阻塞就会是暂时性的）。

因为顺序执行的程序是一个单执行线程，所以必须协调好自己需要完成的多个任务，保证在每个任务上都不会花费太多的时间，还必须保证用户的响应时间分配得很适当。对这一类型的程序任务使用顺序执行程序来完成通常都会导致一个复杂的程序控制流，而这个控制流是难于理解和维护的。

如果在一个多线程化的程序里使用了一种共享化的数据结构——比如一个Queue（这是一个

多线程的队列数据结构,我们将在本章后面的内容里讨论它),这个程序任务就可以被组织为几个分别完成特定函数的线程:

1) **UserRequestThread**: 用户请求线程,负责读取客户的输入——也许是来自I/O通道的输入。程序可以创建几个线程,每个线程对应于一个当前的客户,而客户输入的请求被输入到队列里去。

2) **RequestProcessor**: 对请求进行处理的程序部分,一个负责从队列里检索出请求并对它们进行处理的线程,这个线程向第三个线程提供输出。

3) **ReplyThread**: 回复线程,负责为用户取出输出——如果是在一个网络化的应用程序里,就把输出发送给相应的客户;或者把数据写入本地文件系统或数据库。

采用多线程来组织这个程序大大减少了程序的复杂性,使最终得到的程序本身条理、有效率、易于组织。各线程中的逻辑通常都不会再象以前那样复杂,因为它们每一个都有自己特定的任务。比如说, **UserRequestThread**只负责从某个用户那里读取输入并把它们放置到队列里以便其他线程的进一步处理。每个线程都有它自己要做的工作;程序员只需设计不同类型的线程去做某件事情并把那件事情做好。针对具体任务使用线程与亨利·福特制造汽车的组装线模型是不一样的。

17.2 线程和进程

17.2.1 什么是进程

计算机程序只不过是一些保存在磁盘上的可执行二进制代码,如果没有操作系统把它们加载到内存里并调用它们,其本身是没有什么存在价值的。一个“进程”(有时也被称为“重权进程”)就是一个执行中的程序。每个进程都有它自己的地址空间、内存、一个数据堆栈以及其他一些保持其执行状态的辅助数据。操作系统负责管理着系统上所有进程的执行情况,在全体进程之间公平地分配着处理时间。进程也可以分支或“繁殖”出完成其他任务的新进程,但每个新进程又都有其各自的内存、数据堆栈等事物;除非应用了进程间通信(IPC)功能,否则它们彼此之间一般不共享信息。

17.2.2 什么是线程

“线程”(有时也被称为“轻权进程”)与进程很相似,只不过它们都是在同一个进程的內部执行的,因此它们共享着同样的上下文环境。它们可以被看做是并行运行在一个主进程或叫“主线程”内的“小进程”。

每个线程都有它自己的开始、执行顺序和结尾。它有一个指令指针,里面保存着它在其上下文环境里当前运行到的位置。它可以被中断并临时性被阻塞(也叫做“休眠”),而其他线程可以继续运行——这个情况叫做“让位”。

一个进程内的多个线程与主线程共享着同样的数据空间,所以它们在彼此之间的信息和通信共享方面要比没有什么内在联系的各个进程要容易的多。线程一般都是以共发方式执行的,正是这种并行性和数据共享性使多个任务之间的协调工作易于进行。当然,在一个单CPU系统

里要想真正做到以并发方式运行是不可能的，所以线程们会被安排成这样的情况：向前运行一点，然后让位给其他的线程（它自己会排到队尾等待下一次在CPU上运行的机会）。在整个进程执行过程中，各线程完成它们各自的工作，并会在必要时把执行的结果与其他线程进行通信。

当然，这样的共享也不是没有危险。如果两个或者更多个线程要访问同一项数据，就有可能因数据访问的先后顺序而导致该数据的不稳定性。这种情况一般被称为“竞争情况”。幸运的是，大多数与线程有关的（函数/程序）库在线程间的同步和协调方面都具备某种形式的“内功”，允许线程管理器对执行和访问加以控制。

还要注意一件事情：线程执行时间的分配可能是不平等和不公平的。这是因为某些函数会在自己的执行完成之前对某些资源加以阻塞。如果在编写程序时没有特别考虑到线程的这些特点，那些比较“贪婪”的函数就会霸占过多的CPU时间。

17.3 线程和Python

17.3.1 全局性解释器锁

Python代码的执行是由Python虚拟机（即解释器的主循环）控制的，而Python被设计成在这个主循环里只允许执行一个控制线程，这种情况类似于一个系统里的多个进程共享一个CPU的情况。内存里可能有好几个程序，但任意给定时刻在CPU上被处理的只能是一个。同样地，虽然在Python的解释器里可以“运行”多个线程，但在任意给定的时刻，只有一个线程是正在被解释器执行着的。

对Python虚拟机进行的访问是由一个“全局性解释器锁”（global interpreter lock，简称GIL）控制着的。正是这个锁精确地保证了只有一个线程在运行中。在一个多线程环境里，Python虚拟机的执行方式是下面这个样子：

- 设置GIL。
- 把一个线程切换到运行状态。
- 执行一定数量的字节码指令。
- 把那个线程放回到休眠状态（把线程切换出执行状态）。
- 解开GIL，然后
- 从头再来（有点象洗衣服，洗一会儿，泡一会儿，反复进行）。

如果调用的是外部代码——比如任意一个扩展该语言功能的C/C++函数，那么，在它完成之前GIL锁会一直是锁上的（因为在此期间没有可供计数的Python字节码）。编写语言扩展的程序员们确实有可以打开GIL的办法，所以如果你是Python语言的开发程序员的话，可以不必担心你的Python代码会在这类情况下被锁住。

举个例子，对任何一个面向I/O的Python过程（它会调用内建的C语言操作系统代码）来说，在执行I/O调用之前将打开GIL，从而允许其他线程在I/O操作完成期间去执行。因此，如果代码中的I/O操作不太多，那么，在它让位之前，在分配给一个线程被执行的时间段里它对CPU（和GIL）的利用就更充分。换句话说，I/O操作受限型Python程序要比CPU受限型代码更适合多线程环境。

如果读者对源代码、解释器的主循环和GIL等感兴趣，可以阅读Python/ceval.c文件中的eval_code2()过程，它就是Python虚拟机。

17.3.2 退出线程

线程是为函数的执行而创建的，当这个函数执行完成时，这个线程也就要退出了。退出线程还可以采用调用一个退出函数如thread.exit()、调用退出Python进程的各种标准办法如sys.exit()或者引发一个SystemError例外等手段。

线程退出时的管理有许多种办法。在大多数系统中，当主线程退出时，所有其他的线程也会死亡，但不做任何扫尾工作；在另外一些系统上，它们会继续生存下去。如果想要了解这种情况下线程的具体行为，请参考你自己操作系统的线程程序设计文档。

主线程应该是一个好的管理者，它的主要责任是掌握各线程具体能够完成哪些任务、繁殖出来的线程各自都需要些什么样的数据或参数、它们什么时候能够完成其执行，以及它们将会给出怎样的结果等等。这样，主线程就可以把各线程零散的处理结果拼凑为一个最终的结论性输出。

17.3.3 从Python访问线程

Python支持多线程程序设计，但这也要看用户具体采用的是哪一种操作系统。它支持UNIX的大多数版本，包括Solaris和Linux，以及Windows。线程目前还不能应用在Macintosh平台上。Python使用的是POSIX兼容线程，它们通常被称为“pthreads”。

在缺省的情况下，从源代码开始建立Python时一般不激活线程功能。但在Windows平台上会由安装程序自动安装上有关的模块和功能。分辨是否安装有线程功能的办法是试着从交互式解释器导入thread模块。如果已经安装了线程功能，将不会有任何出错信息，如下所示：

```
>>> import thread
>>>
```

如果你的Python解释器在编译的时候没有激活多线程功能，导入这个模块的操作就会失败：

```
>>> import thread
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ImportError: No module named thread
```

如果真的是这种情况，你就必须重新编译你的Python解释器以添加线程功能——在调用配置脚本时加上“--with-thread”选项一般就能解决问题。请查阅你发行版本中的README文件，看看里面有没有关于如何为你的系统编译上线程功能的特殊指示。

限于本章的篇幅，我们只向大家简单地介绍了Python语言中的线程和多线程程序设计。我们建议读者去查阅正式的文档以全面了解Python所提供的对线程的所有支持。另外，我们建议大家阅读一些操作系统方面的教科书，进一步了解进程、进程间通信、多线程程序设计以及线程/进程同步等方面的知识（书后附录里列出了一些参考书）。

17.3.4 不使用线程时的程序设计情况

我们的第一个例子将用time.sleep()函数做例子说明线程的工作原理。time.sleep()以一个浮点

数为参数，其作用是“休眠”指定的秒数——即在给定的时间段里执行暂时停止。

我们创建两个计时循环，`loop0()`将休眠4秒钟，`loop1()`将休眠2秒钟。（名字“`loop0`”和“`loop1`”是提醒大家此处对应着一个循环语句。）如果我们在一个单进程或单线程程序里按先后顺序执行`loop0()`和`loop1()`，比如像程序示例17-1那样，那总的执行时间至少会是6秒钟。`loop0()`和`loop1()`的启动可能要花费不到一秒钟的时间，再加上其他执行开销可能会使总的执行时间接近7秒钟。

程序示例17-1 由一个单线程执行的循环（`onethr.py`）

在一个单线程程序里顺序执行两个循环，一个循环结束后另外一个循环才能开始执行。总的执行时间是两个循环各自执行时间的和。

```

1 #!/usr/bin/env python
2
3 from time import sleep, time, ctime
4
5 def loop0():
6     print 'start loop 0 at:', ctime(time())
7     sleep(4)
8     print 'loop 0 done at:', ctime(time())
9
10 def loop1():
11     print 'start loop 1 at:', ctime(time())
12     sleep(2)
13     print 'loop 1 done at:', ctime(time())
14
15 def main():
16     print 'starting...'
17     loop0()
18     loop1()
19     print 'all DONE at:', ctime(time())
20
21 if __name__ == '__main__':
22     main()

```

通过执行`onethr.py`可以验证这一点，我们得到如下所示的输出：

```

% onethr.py
starting...
start loop 0 at: Sun Aug 13 05:03:34 2000
loop 0 done at: Sun Aug 13 05:03:38 2000
start loop 1 at: Sun Aug 13 05:03:38 2000
loop 1 done at: Sun Aug 13 05:03:40 2000
all DONE at: Sun Aug 13 05:03:40 2000

```

好了，如果`loop0()`和`loop1()`的功用不是休眠而是互不干扰的两个函数，能够独立完成各自的计算任务并最终结合为一个完整的解决方案，那么，让它们以并行方式运行以节省总的运行时间不是很好吗？这就是我们马上就要向大家介绍的多线程背后的精华。

17.3.5 Python语言中的线程化模块

Python语言提供了几个支持多线程程序设计的模块，包括`thread`、`threading`和`Queue`模块在内。`thread`和`threading`模块允许程序员创建并管理线程。`thread`模块提供了对线程和线程锁的基本支持，`threading`模块提供了更高水平的全功能线程管理功能。`Queue`模块允许用户创建一个能

够被多个线程共享的队列数据结构。我们将对这几个模块分别进行介绍，举几个例子，另外还有几个中规模的应用程序。

编程技巧：避免使用thread模块

[CT]

我们建议大家尽量少使用thread模块的原因是多方面的。第一，因为高层的threading模块更先进，threading模块中的线程支持有相当大的改进，而thread模块中某些属性的用法还可能与使用threading模块相冲突。其次，thread模块中的同步因子比较少（实际上，只有一个），而threading模块中有许多个。

但从Python语言和线程学习总的方面来说，我们还是给出了一些使用thread模块的代码。这些代码仅供学习目的使用，同时也是为了让大家进一步理解为什么要避免使用thread模块。另外，当我们逐步转移到更合适的工具（比如threading和Queue模块提供的那些工具）上去的时候，这些例子也可以让大家看清楚如何改进我们的程序和线程化程序设计。

只有对那些打算进行底层线程访问的专家我们才推荐他们使用thread模块。如果读者在线程方面是一个新手，请把注意力集中在我们代码示例上，认真学习我们是如何把线程覆盖到我们的定时循环上去的，这样才能更好地理解这第一个程序示例是如何逐步改进为作为本章重点的程序示例的。你的第一个多线程应用程序应该尽量使用threading模块或其他必要的线程模块。

17.4 thread模块

我们先来看看thread模块都提供了哪些东西。除了能够繁殖线程以外，thread模块还提供了一个基本的同步数据结构，我们把它叫做线程锁对象（lock object）（它又叫做基本锁、简单锁、共用互斥锁、互斥子或二进制锚）。正如我们前面曾经提到过的，这个同步因子在线程管理中是轮流传递的。

表17-1里列出的是常用线程函数和LockType线程锁对象方法的一个清单：

表17-1 thread模块和线程锁对象

函数/方法	说 明
thread模块函数	
start_new_thread (function, args, kwargs=None)	繁殖出一个新的线程，并以给定的args和可选的kwargs为参数执行函数function
allocate_lock()	为LockType线程锁对象分配内存
exit()	指示一个线程退出执行
LockType锁对象方法	
acquire(wait=None)	尝试获取线程锁对象
locked()	如果得到了线程锁，返回1；否则返回0
release()	释放线程锁

thread模块中的重点函数是start_new_thread()。它的语法与apply()内建函数是完全一样的，

它的参数包括一个函数、该函数的参数和可选的关键字参数。两者的区别在于后者不是让主线程去执行那个函数，而是新繁殖出一个线程去调用那个函数。

我们把线程集成到onethr.py例子上去。只要对loop*()函数的调用稍加改动，我们就得到了如程序示例17-2所示的mtsleep1.py。

程序示例17-2 thread模块的用法 (mtsleep1.py)。

这里执行的是来自onethr.py的同样的循环，但这次使用了由thread模块提供的简单的多线程机制。两个循环同时执行（很明显，那个短一点的循环将先结束），而总的执行时间取决于那个最慢的线程而不再是它们各自用时的和。

```

1  #!/usr/bin/env python
2
3  import thread
4  from time import sleep, time, ctime
5
6  def loop0():
7      print 'start loop 0 at:', ctime(time())
8      sleep(4)
9      print 'loop 0 done at:', ctime(time())
10
11 def loop1():
12     print 'start loop 1 at:', ctime(time())
13     sleep(2)
14     print 'loop 1 done at:', ctime(time())
15
16 def main():
17     print 'starting threads...'
18     thread.start_new_thread(loop0, ())
19     thread.start_new_thread(loop1, ())
20     sleep(6)
21     print 'all DONE at:', ctime(time())
22
23 if __name__ == '__main__':
24     main()

```

start_new_thread()的头两个参数是必不可少的，所以即使执行函数不需要参数也要给它传递一个空表列过去。

再执行这个程序我们的输出可就大变样了。它不再需要用时6到7秒，我们的脚本程序现在只需要4秒就可以执行完毕——也就是执行那个最长的循环所需要花费的时间，再加上点开销。

```

% mtsleep1.py
starting threads...
start loop 0 at: Sun Aug 13 05:04:50 2000
start loop 1 at: Sun Aug 13 05:04:50 2000
loop 1 done at: Sun Aug 13 05:04:52 2000
loop 0 done at: Sun Aug 13 05:04:54 2000
all DONE at: Sun Aug 13 05:04:56 2000

```

分别休眠4秒和2秒的两个代码现在是同时执行的，总用时减少了。

这个程序里还有一个比较重大的修改，就是添加了一个“sleep(6)”调用。这是必需的吗，为什么？原因是这样的：如果我们不用这个调用延缓主线程的执行，它就会前进到下一条语句，显示“all Done...”后退出；而这样将会杀死运行着loop0()和loop1()的两个线程。

没有代码告诉主线程在继续执行前要等待子线程的完成。这就是我们所说的“线程需要某

种形式的同步”这句话的含义。在这个例子里，我们使用了另外一个sleep()调用作为我们的同步机制。我们使用6秒做它的参数，原因是我们知道在主线程数到6之前那两个线程（用时分别是4秒和2秒）都将已经结束了。

读者可能会想到：应该有一个比在主线程里创建一个额外的6秒延时更好的线程管理办法。因为这个延时，总的执行时间并没有比单线程版本好多少。象我们这样使用sleep()来同步线程的办法并不可靠。如果我们循环的执行时间独立且不固定又该怎样做呢？我们退出主线程的时机可能太早或太晚。在这种情况下，就需要使用线程锁。

再来改进我们的代码，加上线程锁，取消各自为战的循环定义，我们得到了程序示例17-3中的mtsleep2.py。运行它，我们会看到与mtsleep1.py差不多的输出。唯一的区别是我们不再需要象mtsleep1.py那样等待额外的时间了。使用了线程锁以后，我们就可以在两个线程完成其执行以后尽快地退出整个程序的执行。

```
% mtsleep2.py
starting threads...
start: loop 0 at: Sun Aug 13 16:34:41 2000
start: loop 1 at: Sun Aug 13 16:34:41 2000
loop 1 done at: Sun Aug 13 16:34:43 2000
loop 0 done at: Sun Aug 13 16:34:45 2000
all DONE at: Sun Aug 13 16:34:45 2000
```

程序示例17-3 thread模块和线程锁的用法 (mtsleep1.py)

这里没有象mtsleep1.py那样用一个sleep()调用来延缓主线程，使用线程锁更合理。

```
1 #!/usr/bin/env python
2
3 import thread
4 from time import sleep, time, ctime
5
6 loops = [ 4, 2 ]
7
8 def loop(nloop, nsec, lock):
9     print 'start loop', nloop, 'at:', ctime(time())
10    sleep(nsec)
11    print 'loop', nloop, 'done at:', ctime(time())
12    lock.release()
13
14 def main():
15     print 'starting threads...'
16     locks = []
17     nloops = range(len(loops))
18
19     for i in nloops:
20         lock = thread.allocate_lock()
21         lock.acquire()
22         locks.append(lock)
23
24     for i in nloops:
25         thread.start_new_thread(loop, \
26                                 (i, loops[i], locks[i]))
27
28     for i in nloops:
29         while locks[i].locked(): pass
30
31     print 'all DONE at:', ctime(time())
```



```
32
33 if __name__ == '__main__':
34     main()
```

那么，怎样才能通过线程锁完成我们的任务呢？我们来看看源代码：

逐行解释

第1~6行

在UNIX操作系统的启动行后面，我们导入了thread模块和time模块中的几个比较熟悉的属性。我们不再用硬编码函数去计时4秒和2秒，我们将使用一个loop()函数并把这些常数放到一个列表loops里去。

第8~12行

前面例子中的两个loop*()函数现在被一个loop()函数顶替了。我们必须对loop()函数进行些修饰，让它能够使用线程锁完成自己的工作。比较明显的改动是我们需要知道循环的编号和休眠期的时间长度；最后一个新加上去的参数是线程锁本身。每个线程都会分配到一个所要求的线程锁。我们将在sleep()时间结束时释放相应的线程锁，通知主线程这个线程已经完成了。

第14~34行

绝大部分工作都是在这个main()部分完成的，它使用了三个for循环。我们先创建了一个线程锁列表，先用thread.allocate_lock()函数为线程锁申请内存，再用acquire()方法取得各线程锁。取得一个线程锁的作用是“把那个锁给锁上”。当线程锁被锁上以后，我们把它添加到线程锁列表locks里去。下一个循环完成的操作有：繁殖线程，在每个线程里调用loop()函数——要向它提供循环的编号、休眠时间和该线程取得的线程锁。那么，我们为什么不在取得线程锁的那个循环里启动线程呢？有以下几个原因：（1）我们想让这两个线程同步执行，让“所有的赛马在同一时间”冲出起跑线；（2）线程锁的取得需要花费一点儿时间。如果你的线程这匹“马”跑得“太快”，就有可能在取得线程锁之前冲过了“终点”。

线程在完成它的执行以后必须负责解除它的线程锁对象。最后一个循环的作用是坐等（暂停主线程的执行）两个线程锁都打开之后再继续执行。因为我们是按先后顺序对每个线程锁进行检查的，所以我们的等待主要集中在那个比较慢的循环身上。在这种情况下，大部分等待时间是用在第一个loop()身上的。当第一个线程锁打开的时候，另外一个线程锁可能早已经打开了（即与另外一个线程锁对应的线程早就结束执行了）；这样就将导致主线程飞快地通过线程锁检查步骤，不做任何停留。最后请注意这个脚本程序的最后两行代码的作用是只有在直接调用这个脚本程序时才执行main()部分。

我们在前面的编程提示里已经提到过，我们之所以会使用thread模块的原因是为了向大家介绍线程化的程序设计方法。你的多线程应用程序应该使用高层模块如threading模块等，这将是下面要讨论的内容。

17.5 threading模块

我们现在开始介绍高层的threading模块，它不仅提供了一个Thread类，还提供了各种能够用在代码心脏部分的同步机制。表17-2列出了threading模块所提供的全部对象。

表17-2 threading模块对象

threading模块对象	说 明
Thread	代表一个执行线程的对象
Lock	基本锁对象（和thread模块中的线程锁对象是一样的东西）
RLock	可重入线程锁对象向一个单线程提供了（重新）取得一个已经保有了的线程锁的能力（即递归性的线程锁）
condition	条件变量对象会让一个线程在等到某个特定的“条件”被另外一个线程满足之后再继续执行，比如状态或描写数据值的改变等
Event	条件变量的通用版本：所有线程都会等待某个事件的发生；而在那个事件发生时，所有线程都将被唤醒
Semaphore	为等待某个线程锁的线程们提供一个类似于“抛锚区”的结构

在本节里，我们将研究如何使用Thread类来实现线程。因为我们已经讨论过线程锁加锁过程的基本原理，所以就不在这里重复关于加锁因子方面的内容了。Thread()类已经包含有一个同步方式，所以没有必要明确地使用加锁因子。

17.5.1 Thread类

利用Thread类来创建线程的办法可以有多种。我们将在这一节里介绍其中的三种，它们的具体做法都差不多。你可以随便挑一个自己觉得顺眼的办法来创建线程，最好让它最适合你的应用程序及其未来发展（我们最喜欢第三种办法）：

- 1) 创建Thread实例，传递进一个函数。
- 2) 创建Thread实例，传递进可调用类实例。
- 3) 从Thread推导子类并创建子类实例。

1. 创建Thread实例，传递进一个函数

在我们的第一个例子里，我们只对Thread类进行实例化，然后把我们的函数传递进去，就象我们前面例子里的做法那样。这个函数就是我们让那个线程开始执行时将要被执行的那个函数。在mtsleep2.py脚本程序的基础上做进一步的修改，添加上Thread类的使用，我们得到程序示例17-4所示的mtsleep3.py。

当我们运行它的时候，我们看到的是和前面例子类似的输出：

```
% mtsleep3.py
starting threads...
start loop 0 at: Sun Aug 13 18:16:38 2000
start loop 1 at: Sun Aug 13 18:16:38 2000
loop 1 done at: Sun Aug 13 18:16:40 2000
loop 0 done at: Sun Aug 13 18:16:42 2000
all DONE at: Sun Aug 13 18:16:42 2000
```

到底改了些什么？我们改动的是在使用thread模块时不得不实现的线程锁。我们创建了一系列Thread对象来代替线程锁。对Thread的每次实例化操作都必须把函数（target）和参数（args）传递给它，然后收到一个返回的Thread实例。对Thread进行实例化（调用Thread()）和调用thread.start_new_thread()最大的区别是新的线程不会立刻开始执行。这是一个非常有用的同步功

能，特别是在你不想让线程们立刻开始执行的时候更是如此。

程序示例17-4 threading模块的用法 (mtsleep3.py)

threading模块中的Thread类有一个名为join()的方法，它的作用是让主线程等待线程的完成。

```

1  #!/usr/bin/env python
2
3  import threading
4  from time import sleep, time, ctime
5
6  loops = [ 4, 2 ]
7
8  def loop(nloop, nsec):
9      print 'start loop', nloop, 'at:', ctime(time())
10     sleep(nsec)
11     print 'loop', nloop, 'done at:', ctime(time())
12
13 def main():
14     print 'starting threads...'
15     threads = []
16     nloops = range(len(loops))
17
18     for i in nloops:
19         t = threading.Thread(target=loop,
20                             args=(i, loops[i]))
21         threads.append(t)
22
23     for i in nloops:          # start threads
24         threads[i].start()
25
26     for i in nloops:          # wait for all
27         threads[i].join()     # threads to finish
28
29     print 'all DONE at:', ctime(time())
30
31 if __name__ == '__main__':
32     main()

```

当所有的线程都分配好以后，我们再通过调用每个线程的start()方法让它们一起冲出起跑线，但在此之前不让它们有所行动。同时，那些线程锁也不再需要我们的管理（分配内存、取得、释放、检查各个线程锁状态等）了，我们只要简单地对每个线程调用join()方法就行了。join()会等待一个线程的结束，如果进行了相应的设置，也可以等待一个线程超时情况的发生。使用join()明显要比一个等待线程锁被释放的无限循环（这使得这些线程锁有时也被称为“轮转锁”）要明晰得多。

join()另外一个重要的特点是其实完全可以不调用它。一旦某个线程启动以后，它们将一直执行到给定它们的函数的结束，而那时它们也就退出了。如果你的主线程不需要等待线程们的完成而是另有事做（比如还有其他处理工作或等待新客户的请求等），就完全可以去那样做。只有在你想等待线程们的完成时join()才有用。

2. 创建Thread实例，传递进可调用类实例

与在创建一个线程时传递进一个函数的做法相类似的另外一个做法是准备一个可调用的类（class）并传递进一个实例去执行——这是多线程程序设计中比较具有OO特点的做法。可调用类体现的是一个执行环境，这比一个函数或者比从一组函数进行挑选要更灵活。你的背后现在有了一个类对象的强有力的支持，不再是一个单独的函数或者是一个函数列表/表列了。

在mtslepp3.py的代码里添加上新类(class) ThreadFunc并做一些小的修改后, 我们得到了程序示例17-5中所示的mtslepp4.py。

如果我们运行mtslepp4.py, 就会得到预期的输出, 如下所示:

```
% mtslepp4.py
starting threads...
start loop 0 at: Sun Aug 13 18:49:17 2000
start loop 1 at: Sun Aug 13 18:49:17 2000
loop 1 done at: Sun Aug 13 18:49:19 2000
loop 0 done at: Sun Aug 13 18:49:21 2000
all DONE at: Sun Aug 13 18:49:21 2000
```

这次我们又改了些什么? 这次添加了ThreadFunc类(class), 并在实例化Thread对象时做了一个小改进——它也对我们的可调用类ThreadFunc进行实例化。其效果是: 我们在这里进行了双重的实例化操作。我们来仔细研究一下我们的ThreadFunc类。

我们想让这个类更通用一些, 让它还能使用我们的loop()函数以外的其他函数。因此, 我们在它的内部结构方面添加了一些新的东西, 比如把线程将要执行的那个函数的参数、那个函数本身以及一个函数名字字符串都放在这个类里。构造器__init__()将把所有这些值都设置好。

程序示例17-5 使用可调用类 (mtslepp4.py0)

在这个例子里, 我们传递进来的是一个可调用类(实例)而不仅仅是一个函数。与mtslepp3.py相比, 它更体现了一种OO手法。

```
1  #!/usr/bin/env python
2
3  import threading
4  from time import sleep, time, ctime
5
6  loops = [ 4, 2 ]
7
8  class ThreadFunc:
9
10     def __init__(self, func, args, name=''):
11         self.name = name
12         self.func = func
13         self.args = args
14
15     def __call__(self):
16         apply(self.func, self.args)
17
18 def loop(nloop, nsec):
19     print 'start loop', nloop, 'at:', ctime(time())
20     sleep(nsec)
21     print 'loop', nloop, 'done at:', ctime(time())
22
23 def main():
24     print 'starting threads...'
25     threads = []
26     nloops = range(len(loops))
27
28     for i in nloops: # create all threads
29         t = threading.Thread( \
30             target=ThreadFunc(loop, (i, loops[i]),
31                               loop.__name__))
32         threads.append(t)
33
```

```

34     for i in nloops: # start all threads
35         threads[i].start()
36
37     for i in nloops: # wait for completion
38         threads[i].join()
39
40     print 'all DONE at:', ctime(time())
41
42 if __name__ == '__main__':
43     main()

```

在创建一个新的线程的时候，当Thread代码调用我们的ThreadFunc对象时，它将调用__call__()特殊方法。因为我们的参数集合是已经准备好了的，所以我们就不再需要把它传递到Thread()构造器去了。但是，在我们的代码里现在必须要有apply()函数了，因为我们使用了一个参数表列。如果读者使用的是1.6或更高版本的Python，就可以用11.6.3节里介绍的函数调用新语法代替第16行处的apply()函数，如下所示：

```
self.res = self.func(*self.args)
```

3. 子类化Thread并创建子类实例

最后的这个介绍性例子引入了子类化Thread()的问题，它与前面例子里创建一个可调用类的做法是极其相似的。当你创建你的线程的时候，采用子类化做法的代码读起来比较容易理解一些（请参考第29到第30行）。我们在程序示例17-6里给出了mtsleep5.py的代码，下面是它的输出情况；mtsleep4.py和mtsleep5.py之间的差异我们留给读者做为一个练习。

下面是mtsleep5.py的输出，和我们预期的完全一样：

```

% mtsleep5.py
starting threads...
start loop 0 at: Sun Aug 13 19:14:26 2000
start loop 1 at: Sun Aug 13 19:14:26 2000
loop 1 done at: Sun Aug 13 19:14:28 2000
loop 0 done at: Sun Aug 13 19:14:30 2000
all DONE at: Sun Aug 13 19:14:30 2000

```

在读者比较mtsleep4和mtsleep5两个模块源代码的时候，我们想指出几个最明显的修改：（1）MyThread子类的构造器必须先调用其父类的构造器（第9行）；（2）原来的特殊方法__call__()在子类里必须叫做run()。

现在，我们在MyThread类里添加一些诊断性输出信息并把它保存到另外一个模块里，新模块的名字是myThread（请参考程序示例17-7）。这样，在以后的程序示例里直接导入这个类就可以了。我们不再简单地通过调用apply()来运行我们的函数，我们将把其操作结果保存到实例属性self.res里去；同时将编写一个新的方法来检索这个值，新方法的名字是getResult()。

程序示例17-6 使用Thread的子类（mtsleep5.py）

我们没有直接对Thread类进行实例化，我们使用了它的子类。这在定制我们的线程对象和简化线程创建调用方面增加了更多的灵活性。

```

1  #!/usr/bin/env python
2

```

```

3  import threading
4  from time import sleep, time, ctime
5
6  loops = ( 4, 2 )
7
8  class MyThread(threading.Thread):
9      def __init__(self, func, args, name=''):
10         threading.Thread.__init__(self)
11         self.name = name
12         self.func = func
13         self.args = args
14
15     def run(self):
16         apply(self.func, self.args)
17
18 def loop(nloop, nsec):
19     print 'start loop', nloop, 'at:', ctime(time())
20     sleep(nsec)
21     print 'loop', nloop, 'done at:', ctime(time())
22
23 def main():
24     print 'starting threads...'
25     threads = []
26     nloops = range(len(loops))
27
28     for i in nloops:
29         t = MyThread(loop, (i, loops[i]), \
30                     loop.__name__)
31         threads.append(t)
32
33     for i in nloops:
34         threads[i].start()
35
36     for i in nloops:
37         threads[i].join()
38
39     print 'all DONE at:', ctime(time())
40
41 if __name__ == '__main__':
42     main()

```

程序示例17-7 Thread类的MyThread子类 (myThread.py)

为了增加mtsleap5.py里我们子类的通用性，我们把这个子类转移到另一个模块里，并且为产生返回值的可调用对象增加了一个名为getResult()的新方法。

```

1  #!/usr/bin/env python
2
3  import threading
4  from time import time, ctime
5
6  class MyThread(threading.Thread):
7      def __init__(self, func, args, name=''):
8         threading.Thread.__init__(self)
9         self.name = name
10        self.func = func
11        self.args = args
12
13    def getResult(self):
14        return self.res
15
16    def run(self):

```

```

17         print 'starting', self.name, 'at:', \
18             ctime(time())
19         self.res = apply(self.func, self.args)
20         print self.name, 'finished at:', \
21             ctime(time())

```

17.5.2 菲波那契数列、阶乘、连加和

在程序示例17-8里给出的mtfacfib.py脚本程序将同时对菲波那契（Fibonacci）数列、阶乘、连加和等递归函数进行计算，并比较它们的执行情况。这个脚本程序先以单线程方式运行这三个函数，然后再用多线程来完成同样的任务。这样做的目的是向大家展示拥有一个线程化环境的优点。

程序示例17-8 菲波那契数列、阶乘、连加和（mtfacfib.py）

在这个多线程应用程序里，我们将同时执行三个递归函数——先以单线程方式执行，然后是一个采用多线程方式的解决方案。

```

1  #!/usr/bin/env python
2
3  from myThread import MyThread
4  from time import time, ctime, sleep
5
6  def fib(x):
7      sleep(0.005)
8      if x < 2: return 1
9      return (fib(x-2) + fib(x-1))
10
11 def fac(x):
12     sleep(0.1)
13     if x < 2: return 1
14     return (x * fac(x-1))
15
16 def sum(x):
17     sleep(0.1)
18     if x < 2: return 1
19     return (x + sum(x-1))
20
21 funcs = [fib, fac, sum]
22 n = 12
23
24 def main():
25     nfuncs = range(len(funcs))
26
27     print '*** SINGLE THREAD'
28     for i in nfuncs:
29         print 'starting', funcs[i].__name__, 'at:', \
30             ctime(time())
31         print funcs[i](n)
32         print funcs[i].__name__, 'finished at:', \
33             ctime(time())
34
35     print '\n*** MULTIPLE THREADS'
36     threads = []
37     for i in nfuncs:
38         t = MyThread(funcs[i], (n,))
39         funcs[i].__name__
40         threads.append(t)
41
42     for i in nfuncs:
43         threads[i].start()
44
45     for i in nfuncs:

```

```

46         threads[i].join()
47         print threads[i].getResult()
48
49     print 'all DONE'
50
51 if __name__ == '__main__':
52     main()

```

单线程模式下的运行很简单，依次调用这三个函数，在函数调用的后面显示与之对应的计算结果。

在多线程模式下运行时，我们不把计算结果立刻显示出来。为了保证MyThread类的通用性（能够执行任意的可调用对象，不管它是否会产生输出），我们将等到最后才调用getResult()方法把每个函数调用的返回值显示给大家。

因为这些函数执行起来都非常快（也许菲波那契数列函数要慢一点儿），所以读者应该注意到我们不得不在每个函数里都加上了sleep()调用以降低速度，只有这样才能让我们看清楚线程是如何改善了性能的。如果是在实际工作中，读者当然不必拘泥于sleep()调用以致于影响到自己的工作。请看下面的输出情况：

```

% mtfacfib.py
*** SINGLE THREAD
starting fib at: Sun Jun 18 19:52:20 2000
233
fib finished at: Sun Jun 18 19:52:24 2000
starting fac at: Sun Jun 18 19:52:24 2000
479001600
fac finished at: Sun Jun 18 19:52:26 2000
starting sum at: Sun Jun 18 19:52:26 2000
78
sum finished at: Sun Jun 18 19:52:27 2000

*** MULTIPLE THREADS
starting fib at: Sun Jun 18 19:52:27 2000
starting fac at: Sun Jun 18 19:52:27 2000
starting sum at: Sun Jun 18 19:52:27 2000
fac finished at: Sun Jun 18 19:52:28 2000
sum finished at: Sun Jun 18 19:52:28 2000
fib finished at: Sun Jun 18 19:52:31 2000
233
479001600
78
all DONE

```

17.5.3 制造商-消费者问题和Queue模块

最后一个例子模拟了制造商和消费者之间的关系：商品或服务的制造商生产商品并把它放到一个数据结构（比如一个队列）里去；生产商品的时间间隔是不确定的；同时，消费者消费着制造商生产的商品。

我们使用Queue模块提供线程间的内部通信机制，它允许线程们彼此共享数据。特别地，我们创建一个队列好让制造商（线程）把商品放进去，而消费者（线程）则从中取出商品进行消费。

特别地，我们将使用Queue模块的下列属性（如表17-3所示）。

表17-3 常用的Queue模块的属性

函数/方法	说 明
Queue模块的函数	
queue(size)	创建一个给定长度的Queue对象
Queue对象的方法	
qsize()	返回队列的长度（这是一个大概的数字，因为队列随时会被其他线程刷新）
empty()	如果队列是空的，返回1；否则返回0
full()	如果队列是满的，返回1；否则返回0
put(item, block=0)	把item放入队列中去。如果给出了block值（不是0），就会被阻塞直到队列里腾出地方为止
get(block=0)	从队列里取出item。如果给出了block值（不是0），就会被阻塞直到队列里至少有一项数据为止

说干就干，我们在程序示例17-9里给出了prodcons.py脚本程序的代码。

程序示例17-9 生产者-消费者问题 (prodcons.py)

这是一个制造商-消费者问题的模拟方案，它使用了Queue对象，生产（及消费）的商品个数用一个随机数来表示。制造商和消费者是独立（但又同时）执行的线程。

```

1  #!/usr/bin/env python
2
3  from random import randint
4  from time import time, ctime, sleep
5  from Queue import Queue
6  from myThread import MyThread
7
8  def writeQ(queue):
9      print 'producing object for Q...',
10     queue.put('xxx', 1)
11     print "size now", queue.qsize()
12
13 def readQ(queue):
14     val = queue.get(1)
15     print 'consumed object from Q... size now', \
16         queue.qsize()
17
18 def writer(queue, loops):
19     for i in range(loops):
20         writeQ(queue)
21         sleep(randint(1, 3))
22
23 def reader(queue, loops):
24     for i in range(loops):
25         readQ(queue)
26         sleep(randint(2, 5))
27
28 funcs = [writer, reader]
29 nfuncs = range(len(funcs))
30
31 def main():
32     nloops = randint(2, 5)
33     q = Queue(32)
34
35     threads = []
36     for i in nfuncs:
37         t = MyThread(funcs[i], (q, nloops), \
38             funcs[i].__name__)
39         threads.append(t)
40
41     for i in nfuncs:
42         threads[i].start()

```

```

43
44     for i in nfuncs:
45         threads[i].join()
46
47     print 'all DONE'
48
49 if __name__ == '__main__':
50     main()

```

下面是这个脚本程序某次执行的输出情况:

```

% prodcons.py
starting writer at: Sun Jun 18 20:27:07 2000
producing object for Q... size now 1
starting reader at: Sun Jun 18 20:27:07 2000
consumed object from Q... size now 0
producing object for Q... size now 1
consumed object from Q... size now 0
producing object for Q... size now 1
producing object for Q... size now 2
producing object for Q... size now 3
consumed object from Q... size now 2
consumed object from Q... size now 1
writer finished at: Sun Jun 18 20:27:17 2000
consumed object from Q... size now 0
reader finished at: Sun Jun 18 20:27:25 2000
all DONE

```

正如你所看到的, 不要求制造商和消费者轮流执行。(感谢上帝让我们有了随机数!) 更重要的是, 现实生活通常都是随机和不确定的。

逐行解释

第1~6行

在这个模块里, 我们使用了Queue.Queue对象和我们在程序示例17-7中给出的线程类myThread.MyThread。我们用random.randint()让生产和消费更富于变化, 另外还从time模块里导入了几个常用的属性。

第8~16行

writeQ()和readQ()函数各有其特定的用途, 分别是把一个对象放入到队列里和从队列里取出一个对象, 这个对象我们用字符串'xxx'做例子。需要注意的是我们每次只写入一个对象, 每次也只读出一个对象。

第18~26行

writer()将做为一个单线程来运行, 它唯一的用途就是为队列制造出一件商品, 过会儿再来一次, 如此反复直到达到指定次数为止, 其循环次数是脚本程序每次执行时随机选定的。reader()的工作情况与之相似, 只是它的用途是消费一件商品。

读者应该注意到, writer的休眠秒数和reader的休眠秒数都是随机数, 但一般情况下前者会比后者要短一些。这样做的目的是为了尽量避免reader从一个空队列里取东西。通过给writer一个比较短的等待时间的办法, 当轮到reader去取东西的时候差不多都会有一个对象能够让它取走了。

第28~29行

这几个赋值语句用来设置将被繁殖出来并被执行的线程总数。

第31~47行

最后，是我们的main()函数，它与我们在本章其他脚本程序里见过的main()看上去很相似。我们创建了相应的线程并把它们送上征程，当两个线程都结束了它们的执行时我们的工作也就完成了。

通过这个例子我们想说明的是：一个需要完成多项任务的程序可以使用多个线程去完成它的各项任务。与一个试图完成所有任务的单线程程序相比，这样做所得到的程序设计要合算得多。

在本章里，我们向大家展示了一个单线程进程是如何限制住一个应用程序的执行性能的。特别地，如果某些顺序执行的程序其任务是独立的、非确定性的以及非关键性的，那就能够通过把它分解为由各独立线程执行的独立任务的办法使之得到改善。并不是所有的应用程序都能够从多线程中获益并承受其开销。但在学习了本章内容之后，读者应该对Python语言的线程化能力有了足够的认识，应该能够用对用好这个工具了。

17.6 练习

17-1 进程和线程的比较。进程和线程之间有什么区别？

17-2 线程。如果多个线程运行在一个单CPU系统上，它们是怎样共享CPU的？

17-3 线程。如果在一台多CPU系统上使用了多个线程，你是否认为会发生重大的事情？你对这些系统上的多线程是怎样看的？

17-4 线程和文件。改进你对练习9-19的解决方案，那个练习是给定一个字节值和一个文件名，要求显示出该字节在该文件里出现过的次数。让我们假设这个文件确实是非常之大。一个文件里有多个阅读器是可以接受的，所以请创建多个线程分别对文件的不同部分进行统计，让每个线程负责那个文件的一个部分。请把各线程的统计结果合并到一起累加出一个总数来。请使用timeit()代码对单线程版本和你新的多线程版本分别计时，然后就性能改善方面发表些看法。

17-5 线程、文件和规则表达式。你有一个非常大的邮箱文件——如果你没有，请把你所有的电子邮件都放到一个文本文件里去。你的任务是这样的：用你在第15章里设计的识别电子邮件地址和Web站点URLs的规则表达式把所有的电子邮件地址和URL都转换为有效链接；这样，当把新文件保存为一个.html（或.htm）文件时，所有的电子邮件地址和URL都会在Web浏览器里呈有效的可点击状态。用线程把对这个大型文本文件的转换处理进行分割，并把它们的结果合并为一个单个的新.html文件。在你的Web浏览器上对结果文件进行测试，确保那些链接确实能够起作用。

17-6 线程和网络。你在上一章中聊天服务应用程序的解决方案（练习16-7和16-10）里使用的可能是重权线程即进程。请把它转换为多线程代码。

17-7 *线程和Web程序设计。程序示例19-1中的Crawler是一个单线程的Web网页下载程序，它应该能够从多线程程序设计中获益。请改进crawler.py（你可以叫它mtcrawler.py）使用多个独立的线程下载网页。记得要使用某种形式的加锁机制以避免链接队列的访问冲突。

第18章 使用Tkinter进行GUI程序设计

我们将在这一章里对使用Tkinter进行图形化用户操作界面（GUI）程序设计的方法做一个快速简单的介绍，Tkinter是Python语言中的Tk图形工具包。GUI开发方面的素材完全可以另外写成一本书（而且也确实如此！），但如果你还是一个新手或者想多学一些这方面的知识，或者你想知道在Python语言这一切都是如何做到的，这章的内容就很适合你学习。我们为大家准备了四个简单的小例子和一个中等规模的例子，对Tkinter更完整的学习还要靠Python语言GUI程序设计方面的专著才行。

18.1 介绍

18.1.1 什么是Tcl、Tk和Tkinter

Tkinter是Python语言缺省的图形化用户操作界面库。它是基于Tk工具包的，而Tk最初是为工具命令语言（Tool Command Language，简称TCL）设计的。因为Tk非常的流行，所以它又被移植到许许多多其他的脚本程序语言上，其中包括Perl（Perl/Tk）和Python（Tkinter）。Tk本身在GUI开发方面就具有很好的可移植性和灵活适应性，再加上与系统语言的强大功能相结合脚本程序语言的简单性，你得到的就是快速设计和实现各种具备商业版质量的GUI应用程序的工具。

如果你是GUI程序设计新手，就会既高兴又吃惊地看到这一切是多么的容易做到。你还将发现Python，再加上Tkinter，提供了一种快速而又激动人心的应用程序设计手段，用它建立的应用程序可以很有意思（可能还有用）；而如果直接使用C/C++语言及C语言的窗口系统库进行程序设计所花费的设计可能要长得多。一旦你规划好自己的应用程序以及这些程序的外观和感觉，就可以使用那些被称为“素材”的基本部件拼凑出想要的东西，最后连接上预定的功能使它“美梦成真”。

如果你在Tk应用方面是一位老手，那么不管你熟悉的是Tcl还是Perl，你都将发现Python是一条通往GUI的新路。在它的顶层提供了一个更加迅速的快速建立GUI模型的系统。别忘了，你还拥有Python语言的系统访问能力、网络处理功能、XML、数值和可视处理、数据库访问以及所有其他的标准库和第三方扩展库。

一旦Tkinter安装到你的系统上，让你的第一个GUI应用程序开始运行起来最多只需要花费15分钟。

18.1.2 安装Tkinter并使它工作

在你的系统上，Tkinter在缺省的情况下是没有被打开的，这与线程很相似。要想知道在自己的Python解释器里能否使用Tkinter，可以通过尝试导入Tkinter模块的办法来检查。如果已经

装上Tkinter了，在执行这个操作时就不会出错，如下所示：

```
>>> import Tkinter
>>>
```

如果你的Python解释器在编译的时候没有激活Tkinter，这个模块导入操作就会出错：

```
>>> import Tkinter
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "/usr/lib/python1.5/lib-tk/Tkinter.py", line 8, in ?
    import _tkinter # If this fails your Python may not
    be configured for Tk
ImportError: No module named _tkinter
```

如果出现这样的情况，你就必须重新编译你的Python解释器，在里面加上Tkinter。这通常需要编辑修改Modules/Setup文件，正确设置编译带Tkinter的Python解释器所要求的全部设置项或者选择把Tk安装到你的系统上。请查阅你Python发行版本中的README文件，看看在你的系统上编译Tkinter方面有没有特殊的指令。在编译完成之后，记得要启动你刚才新建立的Python解释器，否则，你的Python解释器就还是不带Tkinter的那个老版本。

18.1.3 再论客户-服务器体系结构

在前面介绍网络程序设计的那一章里，我们向大家介绍了客户-服务器计算的记号办法。窗口系统也是一种软件服务器。它们运行在一台带有显示设备的计算机上，比如某种型号的监视器等。窗口系统的客户实际上是一些要求运行在一个窗口化环境里的程序，人们通常称它们为GUI应用程序。在没有窗口服务器的情况下，这些应用程序是根本无法启动的。

当网络出现在舞台上以后，这类环境就变得更有趣了。通常，在一个GUI应用程序执行的时候，它会把自己的显示（通过窗口服务器）送到启动它的机器上去，但在某些网络化的窗口环境——比如UNIX操作系统中的X窗口系统里，选择另外一台机器的服务器来显示也能行。在这种情况下，你可以在一台计算机上运行一个GUI程序，但在另外一台计算机上看到它的显示情况！

18.2 Tkinter和Python程序设计

18.2.1 Tkinter模块：把Tk添加到应用程序中去

怎样才能把Tkinter添加到你的应用程序中去呢？先跟大家说明一点，有没有一个应用程序并不是必要的，你完全可以建立一个纯粹的图形化用户操作界面，但如果没有内涵的能够完成某些功能的软件做基础，它一般也不会有太大的用处。

要想建立和运行一个GUI应用程序一般需要经过五个基本步骤：

- 1) 导入Tkinter模块（或者使用“from Tkinter import *”语句）。
- 2) 创建一个顶层窗口对象，它包含着你整个的GUI应用程序。
- 3) 在你的这个顶层窗口对象的顶部（或者“内部”）制作你的GUI部件（和功能）。
- 4) 把这些GUI部件和内涵的应用程序代码结合起来。

5) 进入主事件循环。

第一个步骤就不用多说了：一切使用了Tkinter的GUI都必须导入Tkinter模块。能够访问Tkinter是最基本的要求（请参考前面18.1.2节中的内容）。

18.2.2 GUI程序设计简介

在给大家举例说明之前，我们先对GUI应用程序的开发做一个简单的介绍，让大家继续学习之前先有一个大概的了解。

建立一个GUI应用程序与画家的绘画工作有些相似。而画家先要有一块能让他往上画画的画布。具体过程大概会是这个样子：先从一张白纸开始，这张白纸就是我们的“顶层”窗口对象，你就将在这上面建造其余的部件。我们可以把它想象为一个房子的地基或一个画家的画布。换句话说，你必须先打好地基或支起画布才能开始盖房子或画画。用Tkinter的话来说，这个基础被称为顶层窗口对象。

在GUI程序设计中，一个顶层根窗口对象包含着一切小的窗口对象，这些小的窗口对象装配在一起就构成了完整的GUI应用程序。它们可以是文本标题、按钮、列表框等等。这些小的GUI部件也叫做“素材”。所以，当我们说创建一个顶层窗口对象的时候，我们的意思是需要有一块把所有素材放在它上面的地方。在Python语言里，这个过程通常用下面这样的语句来实现：

```
top = Tkinter.Tk() # 如果使用的是“from Tkinter import *”就只使用Tk()
```

由Tkinter.Tk()返回的对象通常被称为“根对象”（root object），所以有些应用程序在提起它的时候用的是root而不是top。顶层窗口是你的应用程序中能够不依靠其他窗口独立显示的窗口，你的GUI里当然可以有不止一个顶层窗口，但只能有一个是你的根窗口。你可以先把所有的素材都先设计出来，再添上实际功能；也可以一边设计一边实现有关功能——也就是把上表里的第3步和第4步混在一起完成。

素材可以是独立的，也可以是容器。如果一个素材还“包容”着其他素材，它将被认为是那些素材的“父亲”。相应地，如果一个素材被另外一个素材包容着，它就被认为是该“父亲”的一个“孩子”，它的父亲就是紧挨着它的上一级的容器素材。

通常，素材都会有一些关联的行为，比如某个按钮什么时候被按下、某个文本框里的文本已经填写满了，等等。这些用户行为叫做“事件”，而GUI对这些事件的反应叫做“应答”。

窗口里的动作包括按钮被按下（或释放）、鼠标移动、按下回车键，等等。所有这些都会被当作事件报告给系统。对一个GUI应用程序来说，从其开始到结束所发生的事件组成的整个集合就是驱动它运行的东西。这就是人们说的由事件驱动的处理过程。

很简单的鼠标移动就是一个带应答的事件例子。假设鼠标指针正处于你GUI应用程序上的某个位置。如果鼠标被移动到你应用程序的另外一个部分，就应该有些东西会引起鼠标在屏幕上的移动，使它看起来好象真的移动到另外一个位置上去了。这个“东西”就是鼠标移动事件，系统必须对这些事件进行处理，让你看到鼠标正在窗口上有移动。当你释放鼠标的时候，需要处理的事件就没有了，所以屏幕上的东西就又会一动不动地待在它们的位置上。

GUI在本质上是事件驱动的处理过程，这正好符合客户-服务器体系结构的概念。当你启动一个GUI应用程序的时候，它必须先完成一些过程，为核心功能的执行做好准备，这与一个网络

服务器分配一个套接字并把它绑定到一个本地地址的初始化过程道理是一样的。GUI应用程序必须建立起所有的GUI部件，然后把它们画到屏幕上去。Tk有几个几何方面的管理器，它们可以帮助我们吧素材放到正确的位置；最经常使用的那个叫做包装器packer。一旦packer确定了所有素材的尺寸和排列情况，它就会替你把它们都放到屏幕上去。

当包括顶层窗口在内的所有素材都出现在你的屏幕上以后，你的GUI应用程序就会进入一个类似于服务器工作方式的无限循环。这个无限循环等待一个GUI事件的发生，对它进行处理，然后再去等待下一个事件。

我们上面提到的最后一个步骤说的是在所有素材都准备好以后进入主事件循环。这就是我们刚才提到的“服务器”方式的无限循环。在Tkinter里，完成这项工作的代码是：

```
Tkinter.mainloop()
```

这条语句通常会你的程序按顺序将要执行的最后一条语句。在进入主循环以后，GUI就从那里开始对执行进行接管。所有其他的动作都将通过应答来完成，甚至退出应用程序的动作也是如此。当你拉下“File”菜单点击“Exit”菜单项或者直接关闭窗口的时候，将会通过一个应答的调用来退出你的GUI应用程序。

18.2.3 顶层窗口：Tkinter.Tk()

我们前面提到：所有的窗口素材都将建立在顶层窗口对象中。这个对象是由Tkinter中的Tk类（class）创建并通过正常的实例化操作产生出来的，如下所示：

```
>>> import Tkinter
>>> top = Tkinter.Tk()
```

你可以这个窗口里放入单个的素材，也可以把多个部件组合在一起构成你的GUI。那么，都有哪些素材呢？下面我们就来介绍Tk素材。

18.2.4 Tk素材

目前，TK里面有15种素材。我们把这些素材和对它们的简短说明列在表18-1里。

我们不打算对这些TK素材做详细的讲解，因为已经有大量优秀的文档在介绍它们了，其来源包括Python语言的主Web站点里的Tkinter论题主页和浩如烟海的Tcl/Tk书籍与在线资源（本书的附录里给出了一些资料来源）。但我们也准备了几个简单的例子以帮助大家开始学习。

编程提示：缺省参数是你的朋友

[CN]

Python语言的缺省参数确实能够帮上GUI开发的大忙，因为Tkinter素材有大量的缺省动作。除非你知道自己使用的每个素材的每个选项都是干什么用的，否则最好还是只把与你有关的那些参数设置好，剩下的让系统去负责处理。这些缺省值都是经过精心挑选的，即使你没有给出这些缺省参数对应的值，也不必担心你的应用程序会在屏幕上显示成奇怪的样子。它们将按一组经过优化的缺省参数来创建，这是一个基本原则；只有当你清楚地知道应该如何定制自己的素材时才应该使用缺省值以外的值。

表18-1 Tk素材

素材	说 明
Button	与Label的功能类似, 但为鼠标的指针停留、按钮的按下和释放以及键盘活动等事件提高了附加的功能
Canvas	提供了绘制形状(直线、椭圆、多边形、矩形等)的功能: 可以包含图象或位图
Checkbutton	任意个数的一组单选框, 允许选中多个选项(类似于HTML的checkbox输入情况)
Entry	单行的文本输入区, 用来收集键盘输入(类似于HTML的文本输入)
Frame	包含有其他素材的纯容器
Label	用来包含文本或图象
Listbox	向用户显示一个清单, 用户从中挑选出自己的选择
Menu	从一个Menubutton上“悬挂”下来的选择清单, 用户从中进行选择
Menubutton	提供了包含菜单(下拉菜单、层叠菜单等)的内在构造
Message	类似于Label, 但显示的是多行文本
Radiobutton	一组按钮, 每次只能“按下”其中的一个(类似于HTML的radio输入)
Scale	直线型“滑块”素材, 能够提供当前设置的精确值; 需要对它的开始值和结束值进行定义
Scrollbar	为它支持的素材(如Text、Canvas、Listbox、和Entry等)提供翻页卷屏功能
Text	多行的文本区, 用来收集用户输入(或向用户显示信息)(类似于HTML的textarea)
Toplevel	类似于Frame, 但提供了另外一个窗口容器

18.3 Tkinter程序示例

18.3.1 Label素材

在程序示例18-1中, 我们给出了Tkinter版本的“Hello World!”程序tkhello1.py。请注意它是如何对一个Tkinter应用程序进行设置的, 另外要注意的是Label素材的用法。

程序示例18-1 Label素材的使用方法(tkhello1.py)

我们的第一个Tkinter应用程序, “Hello World!”, 请注意我们介绍的第一个素材, Label。

```

1 #!/usr/bin/env python
2
3 import Tkinter
4
5 top = Tkinter.Tk()
6 label = Tkinter.Label(top, text='Hello World!')
7 label.pack()
8 Tkinter.mainloop()

```

我们在程序的第一行创建了我们的顶层窗口。后面是一个Label素材, 其内容就是那个著名的字符串。我们把管理和显示这个素材的工作交给packer去做, 最后调用mainloop()开始运行我们的GUI应用程序。图18-1是运行这个GUI应用程序时你将看到的东西。

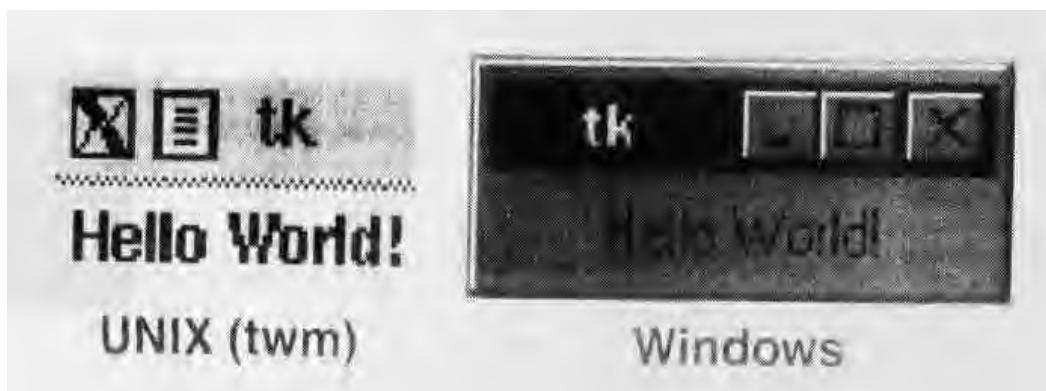


图18-1 Tkinter的Label素材 (tkhello1.py)

18.3.2 Button素材

接下来的例子其实和第一个差不多。但我们创建了一个按钮而不是继续使用那个简单的文本标题。程序示例18-2里给出的是tkhello2.py脚本程序的源代码。

程序示例18-2 Button素材的使用方法 (tkhello2.py)

这个例子和我们的第一个例子完成的是同样的工作，只是我们创建了一个Button素材来代替原来那个Label素材。

```
1 #!/usr/bin/env python
2
3 import Tkinter
4
5 top = Tkinter.Tk()
6 quit = Tkinter.Button(top, text='Hello World!',
7     command=top.quit)
8 quit.pack()
9 Tkinter.mainloop()
```

程序的头几行是完全一样的。从创建Button素材开始两者有了区别。我们的按钮有一个额外的参数，即一个Tkinter.quit()方法。它在我们的按钮上安装了一个应答，这样当它被按下（再被释放）的时候，整个应用程序将结束并退出。最后两行代码分别是普通的pack()语句和进入mainloop()的语句。这个简单按钮的运行情况如图18-2所示。

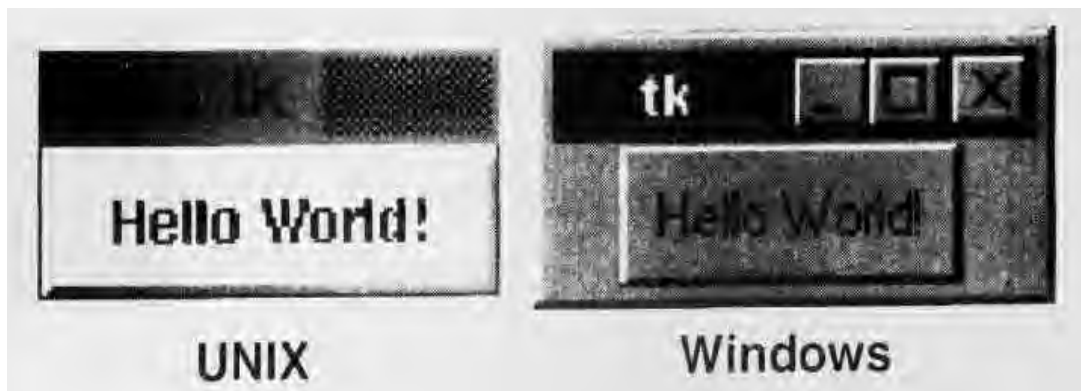


图18-2 Tkinter的Button素材 (tkhello2.py)

18.3.3 Label和Button素材

我们把tkhello1.py和tkhello2.py合并为tkhello3.py。这个脚本程序的素材包括一个标题和一个按钮。此外，与刚才全部使用自动设置的缺省参数不同的是，我们在这个例子里自己多设置了几个参数。tkhello3.py的源代码请见程序示例18-3。

程序示例18-3 Label和Button素材的使用方法 (tkhello3.py)

这个例子使用了两个素材，一个是Label素材，另外一个Button素材。在创建素材时也不再继续使用缺省参数了，对Button素材本身及其设置方法进一步了解使我们能够自行定义更多的参数。

```
1  #!/usr/bin/env python
2
3  import Tkinter
4  top = Tkinter.Tk()
5
6  hello = Tkinter.Label(top, text='Hello World!')
7  hello.pack()
8
9  quit = Tkinter.Button(top, text='QUIT',
10                        command=top.quit, bg='red', fg='white')
11  quit.pack(fill=Tkinter.X, expand=1)
12
13  Tkinter.mainloop()
```

除了在创建素材时多加了几个参数以外，我们还能看到在packer里多出了几个参数。fill参数告诉packer要让“QUIT”按钮占据窗口剩余的水平区域，expand参数指示packer填充整个水平区域，让那个按钮伸展到窗口的左右边界上去。

正如你在图18-3里看到的那样，如果没有给packer其他指令，素材将从上到下依次排列下来（一个压着一个）。如果想把素材沿水平方向放置，就必须新创建一个Frame对象，把按钮放到那里面去。那个框架将做为一个子对象出现在父对象原来出现的位置上（请参考18.3.5节里程序示例18-5的listdir.py模块的按钮布置办法）。



图18-3 Tkinter的Label和Button素材 (tkhello3.py)

18.3.4 Label、Button和Scale素材

最后这个小例子tkhello4.py又多出一个Scale素材。这个Scale素材用来对Label素材进行控制。Scale滑块控制着Label素材中的文本字体的尺寸。滑块的位置越“大”（即向右方移动滑块），字体的尺寸就越大；它的位置越“小”（即向左方移动滑块），字体也就越小。tkhello4.py的源代码在程序示例18-4里给出。

这个脚本程序的新特色包括一个resize()应答函数（见程序第5到第7行），它附着在那个Scale素材身上。当Scale素材中的滑块移动时，就会调用这个代码调整Label素材中文本的字体尺寸。

我们还定义了顶层窗口的尺寸（250 x 150）（见程序第10行）。这个脚本程序和前面三个最后一个差异是：我们用“from Tkinter import *”语句把Tkinter模块的全部属性都导入到我们的名字空间里来了。这主要是因为这个程序大了一些，引用Tkinter模块属性的地方多了一些；如果不这样做就必须在代码里使用这些属性的完整授权名字。而现在的情况是：代码稍微短了一些，也不会象没有把全部属性导入到本地名字空间时那样一条语句要延续好几行了。

程序示例18-4 Label、Button和Scale素材的使用方法（tkhello4.py）

我们最后的这个介绍性例子介绍了Scale素材的用法，并演示了如何通过应答调用（比如resize()）让素材彼此能够进行“通信”。Label素材中的文本会受Scale素材中采取的动作的影响。

```
1  #!/usr/bin/env python
2
3  from Tkinter import *
4
5  def resize(ev=None):
6      label.config(font='Helvetica -%d bold' % \
7                  scale.get())
8
9  top = Tk()
10 top.geometry('250x150')
11
12 label = Label(top, text='Hello World!',
13              font='Helvetica -12 bold')
14 label.pack(fill=Y, expand=1)
15
16 scale = Scale(top, from_=10, to=40,
17              orient=HORIZONTAL, command=resize)
18 scale.set(12)
19 scale.pack(fill=X, expand=1)
20
21 quit = Button(top, text='QUIT',
22              command=top.quit, activeforeground='white',
23              activebackground='red')
24 quit.pack()
25
26 mainloop()
```

如图18-4所示，在窗口的主要部分里显示有滑块机制和它的当前设置值。


```

13     self.label.pack()
14
15     self.cwd=StringVar(self.top)
16
17     self.dirl = Label(self.top, fg='blue',
18                       font=('Helvetica', 12, 'bold'))
19     self.dirl.pack()
20
21     self.dirfm = Frame(self.top)
22     self.dirsb = Scrollbar(self.dirfm)
23     self.dirsb.pack(side=RIGHT, fill=Y)
24     self.dirs = Listbox(self.dirfm, height=15,
25                          width=50, yscrollcommand=self.dirsb.set)
26     self.dirs.bind('<Double-1>', self.setDirAndGo)
27     self.dirsb.config(command=self.dirs.yview)
28     self.dirs.pack(side=LEFT, fill=BOTH)
29     self.dirfm.pack()
30
31     self.dirn = Entry(self.top, width=50,
32                       textvariable=self.cwd)
33     self.dirn.bind('<Return>', self.doLS)
34     self.dirn.pack()
35
36     self.bfm = Frame(self.top)
37     self.clr = Button(self.bfm, text='Clear',
38                       command=self.clrDir,
39                       activeforeground='white',
40                       activebackground='blue')
41     self.ls = Button(self.bfm,
42                      text='List Directory',
43                      command=self.doLS,
44                      activeforeground='white',
45                      activebackground='green')
46     self.quit = Button(self.bfm, text='Quit',
47                        command=self.top.quit,
48                        activeforeground='white',
49                        activebackground='red')
50     self.clr.pack(side=LEFT)
51     self.ls.pack(side=LEFT)
52     self.quit.pack(side=LEFT)
53     self.bfm.pack()
54
55     if initdir:
56         self.cwd.set(os.curdir)
57         self.doLS()
58
59     def clrDir(self, ev=None):
60         self.cwd.set('')
61
62     def setDirAndGo(self, ev=None):
63         self.last = self.cwd.get()
64         self.dirs.config(selectbackground='red')
65         check = self.dirs.get(self.dirs.curselection())
66         if not check:
67             check = os.curdir
68         self.cwd.set(check)
69         self.doLS()
70
71     def doLS(self, ev=None):
72         error = ''
73         tdir = self.cwd.get()
74         if not tdir: tdir = os.curdir
75
76         if not os.path.exists(tdir):
77             error = tdir + ': no such file'
78         elif not os.path.isdir(tdir):
79             error = tdir + ': not a directory'
80
81         if error:
82             self.cwd.set(error)

```

```

83         self.top.update()
84         sleep(2)
85         if not (hasattr(self, 'last') \
86                 and self.last):
87             self.last = os.curdir
88             self.cwd.set(self.last)
89             self.dirs.config(\
90                 selectbackground='LightSkyBlue')
91             self.top.update()
92             return
93
94         self.cwd.set(\
95             'FETCHING DIRECTORY CONTENTS...')
96         self.top.update()
97         dirlist = os.listdir(tdir)
98         dirlist.sort()
99         os.chdir(tdir)
100
101         self.dirl.config(text=os.getcwd())
102         self.dirs.delete(0, END)
103         self.dirs.insert(END, os.curdir)
104         self.dirs.insert(END, os.pardir)
105         for eachFile in dirlist:
106             self.dirs.insert(END, eachFile)
107         self.cwd.set(os.curdir)
108         self.dirs.config(\
109             selectbackground='LightSkyBlue')
110
111 def main():
112     d = DirList(os.curdir)
113     mainloop()
114
115 if __name__ == '__main__':
116     main()

```

这个应用程序的UNIX版本请看图18-6。

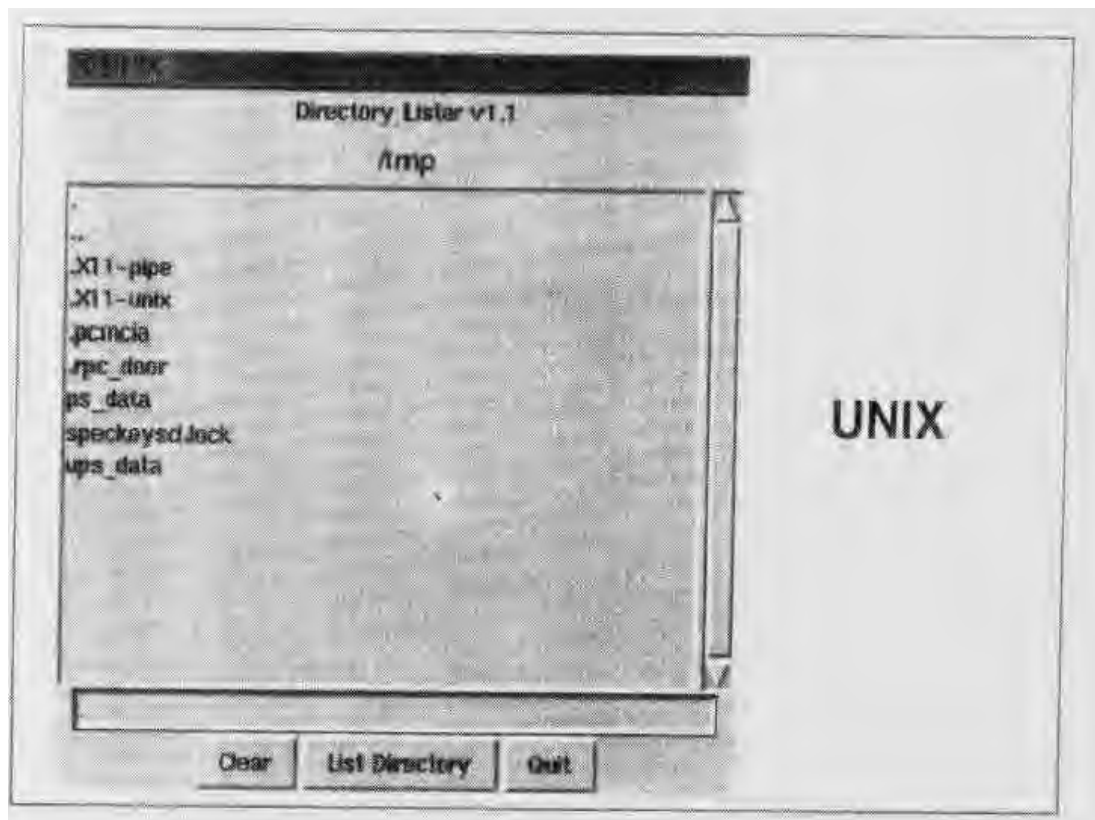


图18-6 UNIX中的列目录文件清单GUI应用程序 (listdir.py)

逐行解释

第1~5行

程序开始处的这儿行包括UNIX操作系统的启动行和导入os模块、time.sleep()函数以及Tkinter模块全部属性的代码。

第9~13行

这几行定义了DirList类的构造器，这个类是代表着我们这个应用程序的一个对象。我们创建的第一个Label其内容是这个应用程序的标题和版本号。

第15~19行

我们定义了一个名为cwd的Tk变量来保存我们身处其中的那个目录的名字——稍后将会看到这样做的方便之处。再另外创建一个Label显示当前目录的名字。

第21~30行

这一部分定义的是我们这个GUI的核心部分——名为dirs的Listbox素材，它包含着对它进行列文件清单操作的那个目录里的文件清单。这里用了一个Scrollbar素材让用户能够在文件个数超过Listbox素材长度的情况下在文件清单里移动。这两个素材都被包容在一个Frame框架素材里。Listbox素材有一个应答调用 (setdirandgo)，它是用Listbox素材的bind()方法绑定在它们身上的。

“绑定”在这里的意思是把一个键盘击键动作、鼠标动作或者其他事件与一个应答调用挂上钩，当用户产生相应的事件时就执行这个应答调用。如果在Listbox素材里双击某个文件，就会调用setdirandgo()。Scrollbar通过调用Scrollbar.config()方法绑定到Listbox素材上。

第32~35行

接下来创建了一个文本输入区Entry素材，用户在这里输入自己想要查看其文件清单的目录名字，文件清单将列在Listbox里面。我们增加了一个回车键绑定到这个文本输入区，这样，用户按下回车键就相当于按下了一个按钮。我们在上面Listbox里看到的使用绑定也是同样的道理。当用户在某个Listbox数据项上双击鼠标的时候，其作用就相当于用户把那个目录的名字手工输入到文本输入区里再按下“go”按钮。

第37~54行

接下来我们定义了一个用来保存那三个按钮的Button框架(bfm)，三个按钮分别是：一个“clear”按钮(clr)、一个“go”按钮(ls)、和一个“quit”按钮(quit)。每个按钮的配置情况各不相同，被按下时的应答调用也不一样。

第56~58行

在构造器的最后我们对这个GUI程序进行了初始化，让它从当前工作目录开始列出文件清单。

第60~61行

clrDir()方法清除Tk字符串变量cwd的内容，它保存着当前正在被列表的目录的名字。这个变量被用来记录我们正在哪个目录里；它还有一个更重要的用处，就是在出现错误的情况下帮助记录前一个被列处文件清单的目录。读者可能已经注意到在它的应答调用函数里有一个缺省值为None的ev变量。任何这样的值都会被窗口系统传递进去，在你的应答调用里可以用也可以不用它们。

第63~71行

setDirAndGo()方法先设置好将要查看的目录，然后发出一个对doLS()方法的调用，这个方法将完成列文件清单操作。

第73~108行

doLS()是整个GUI应用程序的关键所在。它完成所有的安全性检查（比如将要被列文件清单的目标是一个目录吗？它存在吗？）如果出现一个错误，就把刚才列过清单的最后一个目录重新设置为当前目录。如果一切正常，它就调用os.listdir()获得一个文件清单并用它替换掉Listbox里原来的文件清单。当后台工作正在提取新目录有关信息的时候，蓝色光标条将变成鲜红色。等到新目录安装好以后，它又恢复为蓝色。

第110~115行

listdir.py中最后这部分代码是整个代码的主程序部分。只有在直接调用这个脚本程序时才执行main()；当main()运行时，它会先创建这个GUI应用程序，再调用mainloop()启动这个GUI，而这个应用程序的控制权也就转交给它了。

我们这个应用程序的其他方面留给读者做为一个练习，把整个应用程序看做是一系列素材和功能的一个组合更容易读明白它。如果你每个部分都很明白，要弄明白整个脚本程序也就不困难了。

这一章的目的是向大家介绍如何使用Python和Tkinter进行GUI程序设计，我们希望已经较好地达到了这个目标。请记住，熟悉Tkinter程序设计的最佳办法是多实践，多从几个程序示例

里汲取他人的经验！Python语言的发行版本里有大量的演示程序（到Demo子目录里找找看），你可以学习它们的做法。我们在前面还曾经提到过，关于Tkinter程序设计有一个完整的教程。

给大家最后提个醒：你还怀疑Tkinter能够用来编写出商业化应用程序的能力吗？请认真研究一下IDLE吧。IDLE本身就是一个Tkinter应用程序（它是由Guido编写的）！

18.4 相关模块和其他GUI

还有其他一些GUI开发系统能够 and Python一起使用。我们把有关的模块和与它们对应的窗口系统列在表18-2里面。

表18-2 可供Python语言使用的GUI系统

GUI模块或系统	说 明
其他Tkinter模块	
Pmw	Python Mega Widget扩展素材库
开放源代码	
wxPython	wxWindows窗口系统
PyGTK	GTK+ / GNOME / Glade / GIMP 等窗口（或桌面）系统
PyQt/PyKDE	Qt / KDE 图象库和桌面环境
商业软件	
win32ui	微软公司的MFC
swing	Sun微系统公司的Java / Swing开发库

18.5 练习

18-1 客户-服务器体系结构。请说出窗口（或窗口化）服务器和窗口客户各自的特点。

18-2 面向对象的程序设计。请说出父窗口和子窗口之间的关系。

18-3 Label素材。改进tkhello1.py脚本程序，让它显示你自己的文字内容而不再是“Hello World！”。

18-4 Label和Button素材。改进tkhello3.py脚本程序，除“QUIT”按钮外再添加三个新按钮。要求按下这三个新按钮中的任何一个时会改变文本标题的内容，让它里面包含被按下的Button（素材）里的文字。

18-5 Label、Button、Radiobutton素材。改进你上一练习中的解决方案，通过三个单选按钮（Radiobutton）对Label里的文字进行选择。还要有两个按钮：“QUIT”按钮和一个“Update”（修改）按钮。当按下“Update”按钮时，文本标题将被修改为中选单选按钮所代表的文字。如果没有按下任何单选按钮，Label将保持不变。

18-6 Label、Button、Entry素材。改进你上一练习中的解决方案，用一个Entry文本输入区素材代替那三个单选按钮，要求这个Entry有一个缺省值“Hello World！”（这是Label中的初始文字内容）。用户可以对这个Entry区进行编辑，输入一个新的文本字符串；当按下“Update”按钮时，文本标题Label的内容将被刷新。

18-7 Label和Entry素材以及Python语言的I/O。编写一个GUI应用程序，向用户提供一个

Entry文本输入区，用户可以在这里给出一个文本文件的名称；打开这个文件并读它，把它的内容显示在Label里。

附加题（菜单）：把Entry素材替换为一个带“File Open”选项的菜单，这个选项会弹出一个窗口，用户在那个弹出窗口里指定将要读取的文件。另外要在菜单里添加一个“Exit”或“Quit”选项，不再沿用刚才的“QUIT”按钮。

18-8 简单的文本编辑器。以你上一个练习的解决方案为基础编写出一个简单的文本编辑器。文件可以从无到有地创建，也可以被读入并显示到一个Text素材里去，用户可以对这个Text素材进行编辑。当用户退出这个应用程序时（按下“QUIT”按钮或通过“Quit/Exit”选项），要提示用户是否保存所做的修改。

附加题：把你的脚本程序通过接口连接到一个拼写检查器，并增加一个按钮或菜单选项对文件进行拼写检查。拼写错误的单词要在Text素材中用另外的前景或背景颜色突出显示出来。

18-9 多线程聊天应用程序。第13、16和17章中的聊天程序需要进一步完善。请编写出一个全功能多线程的聊天服务器。对服务器来说，GUI并不是必需的，除非你打算创建一个配置前台来对它的端口号、名字、连接的域名服务器等进行配置。编写一个多线程的聊天客户，让它用一个线程处理用户的输入（并把消息发送到服务器以便广播），用另外一个线程接收到来的消息并显示给用户。这个客户的GUI前台需要有两块聊天窗口：较大的那个部分有多个文本行，用来容纳所有的对话；较小的那个文本输入区用来接受来自用户的输入。

第19章 Web程序设计

19.1 介绍

如果缺少对Web程序设计的讨论，任何Python参考书都将是不完整的；Web程序设计是人们认识Python语言的一条主要道路。事实上，最早出现的Python书籍里有一本就叫做《Internet Programming with Python》（用Python进行因特网程序设计）（可惜已经绝版了）。Python语言在因特网上可以做很多的事情，从网上冲浪到编写用户反馈表单、从识别统一资源定位器到生成动态Web网页输出，等等；本章是Web程序设计方面一个快速而又深入的介绍性论述，希望这一章能给大家一个比较全面的认识。

19.1.1 网上冲浪：客户-服务器计算

网上冲浪又一次落在我们前面已经反复多次地看到过的客户-服务器体系结构范畴当中。这一次，Web客户是浏览器，即允许用户在World Wide Web上检索文件的应用程序；另外一头是Web服务器，即运行在一个信息提供者的计算机主机上的进程。这些服务器等待着客户和它们的文档检索请求，对它们进行处理，再按要求返回有关的数据。和大多数客户-服务器系统中的服务器一样，Web服务器也被设计为“永远”运行。网络冲浪的过程如图19-1所示。图中的情况是：一名用户正在运行一个Web客户程序（比如一个浏览器）并与因特网上其他地方的某个Web服务器建立起一个连接以获取那里的资料。

客户可以向Web服务器发出各种各样的请求。这些请求里包括获取并浏览一个网页或提交一个数据表单要求对之进行处理。Web服务器对这些请求进行服务，它们的答案以一种方便显示的特殊格式再返回到客户那里。

Web客户和Web服务器使用的“语言”，即Web通信中使用的标准协议，叫做HTTP，也就是人们常说的超文本传输协议（HyperText Transfer Protocol）。HTTP位于TCP和IP协议集之上，即它需要依赖TCP和IP来完成它的底层通信功能。它（HTTP）的工作不是安排路由或收发消息——这些都是TCP和IP负责的东西，而是用来和/或传递HTTP消息。

HTTP是一个“无状态”协议，因为它不在一个客户请求到下一个客户请求之间记录信息，这与我们前面见过的客户-服务器体系结构非常相似。服务器一直在运行，来自客户的交互操作都是个体性的事件，一旦客户请求得到了服务，就会立刻退出。新的请求可以一直发送过来，但它们将被认为是另外的服务请求。因为各个请求缺乏上下文知识，所以有些URL地址会在其请求中带有一长串变量和变量值，这样做就是为了提供某种形式的状态信息。另一个办法是使用“cookie”——这是一些保存在客户端的静态信息，通常都包含着一些状态信息。在本章后面的内容里，我们将向大家介绍如何通过长URL地址和cookie来保留状态信息。

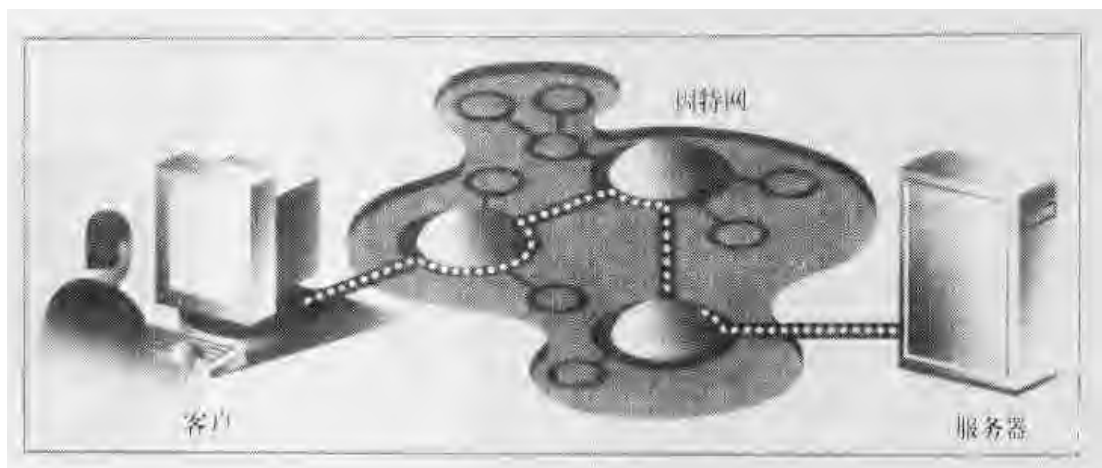


图19-1 因特网上的Web客户和Web服务器。一个客户通过因特网把一个请求发送给一个服务器，服务器响应这个请求并向客户返回所请求的数据

19.1.2 因特网

因特网是一个不断移动和变化中的网络，而这个网络是由散布在世界各地通过各种联网形式连接在一起的客户和服务服务器构成的。客户和服务服务器之间的通信由一系列接力方式的连接组成，由客户端开始，最后一步连接到服务器。作为一个客户端的用户，所有这些连接的细节都是不可见的。从抽象意义上来说，你（即客户）和你要访问的服务器之间好象是直接连接上的，但连接所内涵的HTTP、TCP和IP等协议都隐藏在你看不到的地方，默默地完成了所有的工作。对一般用户来说，不必关心与连接路途上的交接点有关的信息，对自己也没有什么太大的用处，所以具体实现的隐蔽是很好的做法。图19-2给出了一个宏观上的因特网示意图。

从这个示意图里可以看出，因特网是由交叉连接着的网络组成的，各种局部性的网络都以某种和谐（也许彼此并不兼容）的方式工作着。示意图的左侧集中着Web客户，既有在家通过调制解调器拨入其因特网接入服务商（ISP）上网的，也有通过本公司局域网（LAN）上网的。

示意图的右侧主要是Web服务器和它们创建的安放地点。拥有大型Web站点公司在它们的ISP那里通常都安装有一个完整的“服务器阵”。这样的一个物理场所通常被称为“集中地”，意思是一个公司的服务器们都集中安装在某个ISP那里，而该处还有其他顾客的机器。这些服务器或者全部用来向客户提供各种各样的数据，或者用来组成冗余系统对信息进行备份——这是针对大访问量（客户数量非常多）系统的设计。小公司的Web站点一般用不着这么多的硬件和网络引擎，因此只有不多的几个服务器集中放置在它们的ISP那里。

不管是哪一种情况，大多数集中设置的服务器都安装在大型的ISP那里，这些ISP位于一个网络的主干上，也就是说，它们一般拥有更宽和更快的因特网连接——也就是所谓的更接近因特网的“核心”。这样，客户就能够更快地对服务器进行访问。在网络主干上意味着客户不再需要跨越许多个网络才能到达某个服务器，因此在一个给定的时间段里就能有更多的客户得到服务。

另外要提醒大家的是，虽然Web冲浪是最常见的因特网应用，但它并不是因特网唯一的使用办法，而且肯定也不是最古老的使用办法。在出现Web之前，因特网几乎已经存在三十年了。在Web之前，因特网主要被用于教育和研究目的。因特网上大多数系统运行的都是UNIX，这是一

种多用户的操作系统，许多原始的因特网协议至今仍在使用中。

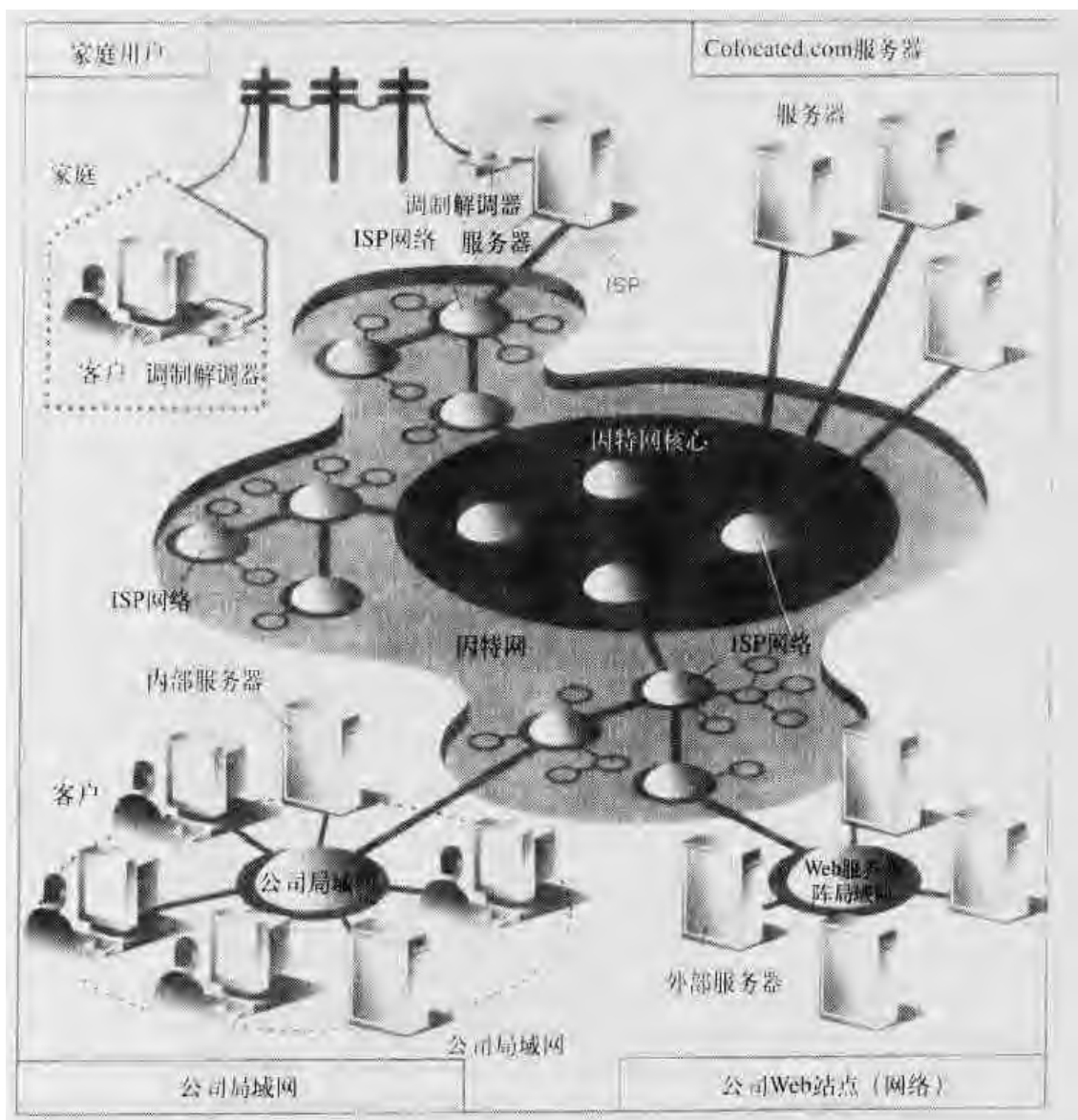


图19-2 因特网的宏观图。左侧是各种各样的Web客户，而右侧是典型的服务器安放地点

这些协议包括telnet（允许用户登录到因特网上的一个远程主机；至今仍在使用中）、FTP（File Transfer Protocol，文件传输协议；允许用户通过上传和下载实现文件和数据的共享；至今仍在使用中）、gopher（Web搜索引擎的前身，这是一个在因特网里搜索用户感兴趣的数据的软件）、SMTP（Simple Mail Transfer Protocol，简单邮件传输协议；这个协议用来实现一个古老而又广泛使用着的因特网应用：电子邮件）和NNTP（News-to-News Transfer Protocol，新闻传输协议）。

因为Python语言最初的发展动力就是因特网程序设计，所以你可以在其中找到对使用上述协议的支持，另外还能够支持许多我们刚才没有提到的协议。在此我们把“因特网程序设计”和“Web程序设计”这两个概念区分一下，用后者表示专门为Web应用而开发的应用程序，我们将在这一章重点介绍的就是Web客户和Web服务器。因特网程序设计覆盖的应用范围就更广了，它

包括我们刚才提到的一些因特网协议，比如FTP、SMTP，等等；一般还包括我们在前面章节里讨论过的网络和套接字程序设计。

19.2 用Python网上冲浪：编写简单的Web客户

请记住，浏览器只是Web客户范畴里的一种而已。应用程序只要能够向某个Web服务器请求数据，它就被看做是一个“客户”。是的，我们完全能够编写出其他的客户来从因特网上检索文档或数据。这样做的一个重要原因是浏览器的能力比较有限，它的基本用途是浏览Web站点并与之互动。而一个客户程序具备完成更多事情的能力——它不仅能够下载数据，还能够保存数据、处理数据，甚至还能够把数据传输到另外一个地方或另外一个应用程序去。

使用urllib模块来下载或访问Web上信息（通过urllib.urlopen()和urllib.urlretrieve()）的应用程序都可以被看做是简单的Web客户。需要你做的就是提供一个正确的Web地址那么简单。

19.2.1 统一资源定位器

简单的Web冲浪需要使用我们称之为统一资源定位器（Uniform Resource Locator，简称URL）的Web地址。一个这样的地址可以用来准确定位一个Web网上的文档或者调用一个CGI程序为你的客户生成一个文档。URL是涵盖范围更大的标识符集合URI（Uniform Resource Identifiers，统一资源标识符）的一个子集。建立URI的目的是为其他命名方案预先做准备，而那些方案仍在开发过程中。简单地说，一个URL就是在自己的地址方案里使用了现有协议或原理（比如http、ftp等）的一个URI。再补充一下，不是URL的URI有时也被称为URN（Uniform Resource Name，统一资源名字）；但是，因为URL是现时期唯一得到应用的URI，所以你很少会听人们说起URI或者URN。

类似于街道地址，Web地址也有自己的书写规则。美国的街道地址通常把门牌号放在前面，把街道名称放在后面，比如“123 Main Street”。它与其他国家中的地址写法不同，各国的习俗是不一样的。URL的格式如下所示：

```
prot_sch://net_loc/path;params?query#frag
```

各成分的含义请看表19-1。

表19-1 Web地址各成分的含义

URL成分	说 明
prot_sch	网络协议或下载方式
net_loc	服务器的位置（也许还有用户信息）
path	用斜线字符（/）分隔的路径名，文件或CGI应用程序就保存在这个路径下
params	可选的参数
query	用“&”字符分隔的一组“键字=键值”数据对
frag	在文档内定位某个特定位置的段落标记

net_loc还可以被分成几个成分，其中有的是必不可少的，有的是可选的，net_loc字符串的格式如下所示：

```
user:password@host:port
```

这些成分的含义请看表19-2。

表19-2 网络位置的各组成成分

net_loc成分	说 明
user	用户名或登录名
password	用户的口令字
host	运行着Web服务器的计算机的名字或地址（此参数必不可少）
port	端口号（如果不是缺省的80号端口，就必须给出来）

在这四个成分里面，主机名host是最重要的。端口号port只有当Web服务器运行在与其缺省设定不同的端口号上时才必须给出。（如果你还不明白端口号是什么，请复习第16章内容。）

用户名，也许还要算上口令字，只有在建立FTP连接时才会用到；但即使是在FTP中它们也不是必须的，因为大多数这类的连接都是“匿名的”。

Python提供了两个不同的模块，它们对URL的处理功能和能力是完全不同的。它们一个是urlparse，另一个是urllib。我们下面介绍几个这两个模块中的函数。

19.2.2 urlparse模块

urlparse模块提供了对URL字符串进行处理的基本功能（urlparse模块的核心函数见表19-3）。这些函数包括urlparse()、urlunparse()和urljoin()。

1. urlparse.urlparse()函数

urlparse()的作用是把一个URL字符串分断为前面介绍的各个成分，它的语法如下所示：

```
urlparse(urlstr, defProtSch=None, allowFrag=None)
```

urlparse()把urlstr分断为由6个元素组成的表列，即(prot_sch, net_loc, path, params, query, frag)。这些成分的说明请参考上一节内容。defProtSch定义了缺省的网络协议或下载方式，如果用户没有给出这个设置值，就会使用它定义的缺省设置。allowFrag是一个表示是否允许在URL中使用不完整成分的操作标志。下面是对一个URL进行处理后的urlparse()输出：

```
>>> urlparse.urlparse('http://www.python.org/doc/FAQ.html')
('http', 'www.python.org', '/doc/FAQ.html', '', '', '')
```

2. urlparse.urlunparse()函数

urlunparse()的作用正好与urlparse()相反，它把一个6个元素的表列(prot_sch, net_loc, path, params, query, frag)（我们通常称之为urltup）合并为一个URL字符串并返回它，这个urltup也许正是urlparse()的一个输出。也就是说，下面的恒等式是成立的：

```
urlunparse(urlparse(urlstr)) == urlstr
```

读者可以已经总结出urlunparse()的语法了，如下所示：

```
urlunparse(urltup)
```

3. urlparse.urljoin()函数

当需要用到许多有相互联系的URL地址——比如为某个Web站点而生成的 一组网页的URL地址时，urljoin()函数就很有用。urljoin()的语法如下所示：

```
urljoin(baseUrl, newurl, allowFrag= None)
```

urljoin()把baseUrl的基本路径（net_loc加上文件的完整路径，不包括最后的文件名）和newurl组合在一起。请看下面的例子：

```
>>> urlparse.urljoin('http://www.python.org/doc/FAQ.html', \
... 'current/lib/lib.htm')
'http://www.python.org/doc/current/lib/lib.html'
```

表19-3 urlparse模块的核心函数

urlparse函数	说 明
urlparse(urlstr, defprotSch=None, allowFrag=None)	把urlstr分断为各个成分。如果在urlstr里没有给出协议或者下载方式，就使用defProctSch的设置值；allowFrag用来确定是否允许该URL地址包含不完整成分
urlunparse(urltup)	把一个包含着URL数据（urltup）的表列组合为一个URL字符串
urljoin(baseUrl, newurl, allowFrag=None)	把URL地址baseUrl的基本路径与newurl组合在一起组成一个完整的URL。allowFrag的作用与urlparse()中的一样

19.2.3 urllib模块

编程风格：urllib

[CM]

除非你正打算编写一个更底层的网络客户程序，否则urllib模块所能够提供的功能就足以满足你的需要。urllib模块是一个高层Web通信库，支持各种基本的网络协议如HTTP、FTP和Gopher等，同时还提供了访问本地文件的能力。特别地，urllib模块中的函数的设计目的就是为能够使用前面提到的各种协议（从因特网、本地网络或本地主机上）下载数据。如果使用了这个urllib模块，一般就不再需要使用httplib、ftplib和gopherlib等模块了，除非你确实需要用到这些模块里面的底层功能。在这种情况下，可以考虑用底层模块代替这个模块。（注意：大多数名字是*lib形式的模块通常都是用来开发相关协议的网络客户程序的。但这并非总是如此；也许应该把urllib模块的名字改为“internetlib”或者其他差不多的名字！）

urllib模块提供的函数能够从一个给定的URL地址处下载数据，还能够对字符串进行编码或解码以便让它们适合做为一个合法URL字符串的组成成分。在这一章里我们将向大家重点介绍的函数包括：urlopen()、urlretrieve()、quote()、quote_plus()、unquote()、unquote_plus()和urlencode()。

urlopen()函数返回的是一个文件风格的对象，我们还将在这一节里介绍这个对象的几个方法。

1. urllib.urlopen()函数

urlopen()按照给定的URL字符串打开一个Web连接并返回一个文件风格的对象。它的语法如下所示：

```
urlopen(urlstr, postQueryData=None)
```


urlopen()将打开由urlstr字符串指定的URL。如果在调用时没有给出将要使用的协议或下载方式，或者传递到函数里去的是一个“文件”方式，urlopen()就将打开一个本地文件。

就各种HTTP请求来说，最普通的请求类型就是“GET”。在这种情况下，传递给Web服务器的查询字符串应该包括在urlstr字符串里；查询字符串是一些经过“编码”或者说是用quote*()函数处理过的键字-键值数据对，比如urlencode()函数的字符串输出等（请参考后面给出的例子）。

如果准备使用的请求方法是“POST”，那就必须要把查询字符串（别忘了编码）放在postQueryData变量里。（有关GET和POST请求方法的详细介绍可以在任何一本介绍CGI应用程序设计的通用文档或教科书里找到——我们也将下面的内容里做些讨论。GET和POST请求是向Web服务器上传下载数据的两种方法。

在成功地建立起一个连接之后，urlopen()将返回一个文件风格的对象，就好像服务器上的操作目标是一个以读模式打开的文件那样。举个例子，假设我们的文件对象是f，这个“句柄”将支持以下各种读操作的文件方法，如f.read()、f.readline()、f.readlines()、f.close()和f.fileno()等。

除此之外，还有一个名为f.info()的方法，它返回的是一个MIME（多用因特网邮件扩展）表头。这个表头向浏览器提供“哪个应用程序可以用来查看返回的文件类型”这样的信息。比如说，浏览器本身就可以用来查看HTML（Hypertext Markup Language，超文本标签语言）或纯文本类型的文件、GIF（Graphics Interchange Format，图形交换格式）和JPEG（Joint Photographic Experts Group，联合图像专家组）图形文件。其他类型的文件如多媒体或某些特定的文档类型需要使用外部应用程序才能查看。

最后，还有一个geturl()方法，它可以获取最终被打开的那个操作目标的真实URL，这样做的原因是考虑到可能会出现重定向的因素。我们把该文件风格对象的这些方法列在表19-4里。

表19-4 urllib.urlopen()返回的文件风格对象的方法

urlopen()对象的方法	说 明
f.read([bytes])	从f里读出它全部的字节或bytes字节
f.readline()	从f里读一行数据
f.readlines()	从f里把全部数据行读到一个列表里
f.close()	关闭与f的URL连接
f.fileno()	返回f的文件号
f.info()	取出f的MIME表头
f.geturl()	返回真正打开的URL

2. urllib.urlretrieve()函数

如果你想把一个URL文档作为一个整体进行处理，urlretrieve()可以替你完成某些短平快和繁重的工作。下面是urlretrieve()的语法：

```
urlretrieve(urlstr, localfile=None, downloadStatusHook=None)
```

urlretrieve()不是像urlopen()那样对URL进行读操作，它只是简单地把位于urlstr处的HTML文件整个地下载到你的本地硬盘上去。如果给出了localfile，就把下载到的数据保存到这个文件里去；如果没有给出localfile，它就会把下载到的数据保存到一个临时文件里去。如果已经从因特网上拷贝过这个文件、或者这个文件是一个本地文件的话，下载操作就不会被执行。

`downloadStatusHook`是一个函数，如果给出了这个函数，就会在每个数据块被下载和发送到本地计算机上之后调用它。在调用这个函数的时候需要有下面这三个参数：到目前为止已经读到了的数据块个数、以字节计算的数据块长度、被下载文件的（字节）长度。如果你打算以文本或者图形化的方式向用户显示“下载进度”信息，这几个参数所包含的信息是很有用的。

`urlretrieve()`将返回一个由2个元素组成的表列，即(`filename`, `mime_hdrs`)。`filename`是包含着下载数据的本地文件的名字；`mime_hdrs`是由Web服务器那一方返回的一组MIME表头。详细情况请查阅`mimetools`模块里的`Message`类。本地文件的`mime_hdrs`是`None`。

我们在程序示例11-2里给出了一个使用`urlretrieve()`的例子（`grabweb.py`）。

3. `urllib.quote()`和`urllib.quote_plus()`函数

`quote*()`函数对URL数据进行处理，它的作用是对它们进行“编码”使之能够被用做一个URL字符串的组成成分。明确地说，必须对某些非打印或不能用在能够被Web服务器接受的合法URL里面的特殊字符进行转换。这就是`quote*()`函数将为你做的事情。两个`quote*()`的语法都是下面这个样子：

```
quote(urldata, safe='/' )
```

永远不需要被转换的字符包括逗号、下划线、英文句号、连字符和字母数字字符；所有其他的字符都要求进行转换。准确地说，不允许用在URL中的字符都将被转换为用百分号（%）引导的十六进制对应值，即“%xx”形式；“xx”是与该字符ASCII值对应的十六进制表示形式。在调用`quote*()`的时候，`urldata`字符串会被转换为一个相应的能够被用在URL字符串里面的字符串。`safe`字符串里面包含的是一组不需要被转换为的字符，它的缺省值是斜线字符（/）。

`quote_plus()`与`quote()`的作用基本一致，只是前者会把空格字符转换为加号（+）。下面是`quote()`和`quote_plus()`的对照用法：

```
>>> name = 'joe mama'
>>> number = 6
>>> base = 'http://www/~foo/cgi-bin/s.py'
>>> final = '%s?name=%s&num=%d' % (base, name, number)
>>> final
'http://www/~foo/cgi-bin/s.py?name=joe mama&num=6'
>>>
>>> urllib.quote(final)
'http:%3a//www/%7efoo/cgi-bin/s.py%3fname%3djoe%20mama%26num%3d6'
>>>
>>> urllib.quote_plus(final)
'http%3a//www/%7efoo/cgi-bin/s.py%3fname%3djoe+mama%26num%3d6'
```

4. `urllib.unquote()`和`urllib.unquote_plus()`函数

读者可能已经猜到`unquote*()`函数的作用应该是正好与`quote*()`函数相反——它们把所有按“%xx”方式编码的字符转换为它们对应的ASCII值。`unquote*()`的语法如下所示：

```
unquote*(urldata)
```

调用`unquote()`会对`urldata`中所有按URL规则编码了的字符进行解码，并返回结果字符串。`unquote_plus()`会把加号转换为空格字符。

5. `urllib.urlencode()`函数

[1.5.2]

`urlencode()`是Python语言新增加的一个函数（从1.5.2版本开始），它对一个由键字-键值数据

对组成的字典进行编码处理，使它们能够被用在一个供CGI请求使用的URI字符串里。“键字=键值”格式的数据对之间用“&”符号隔开，然后键字和它们的对应键值被送入unquote_plus()函数里进行编码。下面是一个urlencode()函数的输出示例：

```
>>> aDict = { 'name': 'Georgina Garcia', 'hmdir': '~ggarcia' }
>>> urllib.urlencode(aDict)
'name=Georgina+Garcia&hmdir=~7eggarcia'
```

在urlparse和urllib模块里还有其他一些函数，但我们没有机会在这里逐个对它们进行介绍了。详细情况请参考有关的文档。

6. 安全套接字层支持

[1.6]

Python 1.6版本中的urllib模块是经过改进的，它现在已经能够支持使用安全套接字层(Secure Socket Layer, SSL)打开的HTTP连接。增加SSL支持所要求的修改主要是在socket模块里实现的。其结果是urllib和httplib模块也得到了升级，现在已经能够支持使用“https”方式的连接了。需要提醒大家注意的是：到本书付印的时候还只有使用SSL的HTTP请求被实现出来了。大家可能会在今后看到进一步的升级使urllib模块能够支持其他的协议，比如FTP等。

对本节讨论过的urllib模块中的函数的总结请看表19-5。

表19-5 urllib模块的核心函数

urllib函数	说 明
urlopen(urlstr, post-QueryData=None)	打开URL地址urlstr，如果是一个POST请求就送出postQueryData中的请求数据
urlretrieve(urlstr, localfile=None, download-StatusHook=None)	把URL地址urlstr处的文件下载到localfile文件里；如果没有给出localfile就下载到临时文件里去。如果调用时还给出了downloadStatusHook函数，就可以用它接收下载统计数字
quote(urldata, safe='/')	对urldata中的非法URL字符进行编码。safe字符串里面的字符不参加编码
quote_plus(urldata, safe='')	类似于quote()，但它还会把空格字符编码为加号
unquote(urldata)	对编码后的urldata进行解码
unquote_plus(urldata)	类似于unquote()，但它还会把加号转换为空格字符
urlencode(dict)	把字典dict中的键字-键值数据对编码为合法的CGI查询字符串；对键字-和键值的编码工作是用quote_plus()完成的

19.3 高级Web客户

Web浏览器是基本的Web客户程序，它们的主要作用是在网上搜索和下载文档。Web网高级客户程序指的是这样一些应用程序，它们完成的工作远不止从因特网上下载一个文档那么简单。

网页收集器（有时人们也叫它“蜘蛛”或“机器人”）就是这样的高级Web客户程序。它们为了不同的理由从因特网上搜索和下载主页，这些理由包括：

- 加入到一个大型搜索引擎（比如Google、Alta Vista和Yahoo!等）的分类索引里去。
- 离线浏览——把文档下载到一个本地硬盘上，重新编排超链接以创建一个能够在本地浏览的镜像。

- 为存档目的而进行的下载和存储。
- 对Web网页进行缓存，再次访问Web站点时可以节省大量的时间。

我们在下面给出的网页收集器crawl.py从某个Web地址（即URL）开始把链接在前一个网页上的其他全部主页都下载下来，但只下载与第一个主页在同一个域的那些主页。如果不加上后一条限制，你的硬盘肯定是不够用的！程序示例19-1是crawl.py的源代码。

程序示例19-1 一个高级Web客户程序：一个网页收集器（crawl.py）

这个网页收集器包括两个类，一个用来对整个收集过程进行管理（Crawler类），另外一个对下载到的每一个Web网页进行检索和分析（Retriever类）。

```

1  #!/usr/bin/env python
2
3  from sys import argv
4  from os import makedirs, unlink, sep
5  from os.path import dirname, exists, isdir, splitext
6  from string import replace, find, lower
7  from htmllib import HTMLParser
8  from urllib import urlretrieve
9  from urlparse import urlparse, urljoin
10 from formatter import DumbWriter, AbstractFormatter
11 from cStringIO import StringIO
12
13 class Retriever:  # download Web pages
14
15     def __init__(self, url):
16         self.url = url
17         self.file = self.filename(url)
18
19     def filename(self, url, deffile='index.htm'):
20         parsedurl = urlparse(url, 'http:', 0) # parse path
21         path = parsedurl[1] + parsedurl[2]
22         text = splitext(path)
23         if ext[1] == '': # no file, use default
24             if path[-1] == '/':
25                 path = path + deffile
26             else:
27                 path = path + '/' + deffile
28         dir = dirname(path)
29         if sep != '/': #os-indep.path separator
30             dir = replace(dir, '\\', sep)
31         if not isdir(dir): # create archive dir if nec.
32             if exists(dir): unlink(dir)
33             makedirs(dir)
34         return path
35
36     def download(self): # download Web page
37         try:
38             retval = urlretrieve(self.url, self.file)
39         except IOError:
40             retval = ('*** ERROR: invalid URL "%s" %\n' %\
41                       self.url,)
42         return retval
43
44     def parseAndGetLinks(self): # parse HTML, save links
45         self.parser = HTMLParser(AbstractFormatter(\
46             DumbWriter(StringIO()))
47         self.parser.feed(open(self.file).read())
48         self.parser.close()
49         return self.parser.anchorlist

```

```

48
49 class Crawler:      # manage entire crawling process
50
51     count = 0        # static downloaded page counter
52
53     def __init__(self, url):
54         self.q = [url]
55         self.seen = []
56         self.dom = urlparse(url)[1]
57
58     def getPage(self, url):
59         r = Retriever(url)
60         retval = r.download()
61         if retval[0] == '*': # error situation, do not parse
62             print retval, '... skipping parse'
63             return
64         Crawler.count = Crawler.count + 1
65         print '\n(', Crawler.count, ')'
66         print 'URL:', url
67         print 'FILE:', retval[0]
68         self.seen.append(url)
69
70         links = r.parseAndGetLinks() # get and process links
71         for eachLink in links:
72             if eachLink[:4] != 'http' and \
73                 find(eachLink, '://') == -1:
74                 eachLink = urljoin(url, eachLink)
75             print '* ', eachLink,
76
77             if find(lower(eachLink), 'mailto:') != -1:
78                 print '... discarded, mailto link'
79                 continue
80
81             if eachLink not in self.seen:
82                 if find(eachLink, self.dom) == -1:
83                     print '... discarded, not in domain'
84                 else:
85                     if eachLink not in self.q:
86                         self.q.append(eachLink)
87                         print '... new, added to Q'
88                     else:
89                         print '... discarded, already in Q'
90             else:
91                 print '... discarded, already processed'
92
93     def go(self): # process links in queue
94         while self.q:
95             url = self.q.pop()
96             self.getPage(url)
97
98     def main():
99         if len(argv) > 1:
100             url = argv[1]
101         else:
102             try:
103                 url = raw_input('Enter starting URL: ')
104             except (KeyboardInterrupt, EOFError):
105                 url = ''
106
107         if not url: return
108         robot = Crawler(url)
109         robot.go()
110
111 if __name__ == '__main__':
112     main()

```

逐行（逐类）解释

第1~11行

这个脚本程序的开始部分包括UNIX操作系统标准的Python启动行以及此应用程序将要用到的各种模块属性的导入操作。

第13~47行

Retriever类负责从Web上下载网页并对每一个文档里面的链接进行分析，如果符合我们的下载原则就把它添加到“待处理”队列里去。从网上下载到的每个主页都有一个与之对应的Retriever实例。Retriever有几个帮助实现其具体功能的方法，它们是：一个构造器（__init__）、filename()、download()和parseAndGetLinks()。

filename()方法把给定的URL对应转换为本地存储时将要使用的文件名。它的基本做法是：去掉URL中的“http://”前缀，用剩余部分作本地文件名，创建一切必要的目录路径。如果URL里面没有尾缀的文件名，就用缺省值“index.htm”做文件名（这个文件名可以在filename()调用中被覆盖掉）。

构造器实例化一个Retriever对象，并把那个URL字符串和从filename()返回的与之对应的文件名保存为本地属性。

download()方法就像它的名字表达的那样连接上网并按给定的链接下载主页。它以那个URL作为参数调用urllib.urlretrieve()函数实现操作，把下载到的东西保存到filename()方法返回的那个文件名里去。如果下载成功，就调用parse()方法对刚从网上拷贝来的主页进行分析；如果失败，返回一个表示出现错误的字符串。

如果在处理过程中Crawler没有发现任何出错，就会调用parseAndGetLinks()方法对新下载到的主页进行分析，确定对那个主页上的每一个链接应该采取什么样的行动。

第49~96行

Crawler类是这出戏里的“明星”，它对整个网页收集过程进行管理，因此我们这个脚本程序每次执行时只会创建一个Crawler实例。Crawler由三个数据项组成，这三个数据项是由构造器在实例化阶段中保存在这里的。第一个数据项是q，这是一个由下载链接组成的队列。这个清单在执行过程中是会变化的，每处理一个主页它就缩短一次，而在各下载主页里发现一个新的链接时就会被加长。

Crawler的另外两个数据项包括seen——这是我们已经下载过的全体链接所组成的一个列表；最后，我们把主链接的域名保存到dom里，用这个值核对后续链接是否属于这同一个域。

Crawler还有一个静态数据项叫做count。这个计数器的作用是记录我们已经从网上下载到的对象的个数。每成功地下载到一个主页，就让它增加一个数。

除了自己的构造器以外，Crawler还有两个其他的方法，它们是getPage()和go()。go()方法的作用很简单，它用来启动Crawler，由代码的主程序部分调用。go()方法里面有一个循环，只要队列里还有需要下载的新链接，它就会不停地执行。但这个类里真正干活的还是getPage()方法。

getPage()方法用第一个链接实例化出一个Retriever对象，从它开始进行后续的处理。如果这个网页下载成功，就增加计数器并把该链接添加到“已查看过”清单里。它递归地检查每个已下载主页上的全部链接，看看有没有需要添加到“待处理”队列里去的新链接。go()方法的主循

环将对队列里的链接进行处理直到队列变成空的为止，然后通知用户操作完成。

以下链接将被忽略，不会被添加到待处理队列里去：属于另一个域的链接、已经被下载过的链接、已经放入待处理队列里去了的链接或者是“mailto:”链接。

第98~102行

main()只有在这个脚本程序在直接被调用时才会执行，它是程序执行的出发点。其他导入了crawl.py的模块需要明确地调用main()才能开始处理。要让main()开始执行，需要给它一个URL。如果已经在命令行上给出了一个URL（比如我们直接调用这个脚本程序的时候），它就会从给定的URL起开始运行；否则，脚本程序将进入交互模式，提示用户输入一个起始URL。有了初始链接之后，程序将对Crawler类进行实例化并开始执行。

下面是某次调用crawl.py脚本程序的执行结果：

```
% crawl.py
Enter starting URL: http://www.null.com/home/index.html

( 1 )
URL: http://www.null.com/home/index.html
FILE: www.null.com/home/index.html
* http://www.null.com/home/overview.html ... new, added to Q
* http://www.null.com/home/synopsis.html ... new, added to Q
* http://www.null.com/home/order.html ... new, added to Q
* mailto:postmaster@null.com ... discarded, mailto link
* http://www.null.com/home/overview.html ... discarded, already in Q
* http://www.null.com/home/synopsis.html ... discarded, already in Q
* http://www.null.com/home/order.html ... discarded, already in Q
* mailto:postmaster@null.com ... discarded, mailto link
* http://bogus.com/index.html ... discarded, not in domain

( 2 )
URL: http://www.null.com/home/order.html
FILE: www.null.com/home/order.html
* mailto:postmaster@null.com ... discarded, mailto link
* http://www.null.com/home/index.html ... discarded, already processed
* http://www.null.com/home/synopsis.html ... discarded, already in Q
* http://www.null.com/home/overview.html ... discarded, already in Q

( 3 )
URL: http://www.null.com/home/synopsis.html
FILE: www.null.com/home/synopsis.html
* http://www.null.com/home/index.html ... discarded, already processed
* http://www.null.com/home/order.html ... discarded, already processed
* http://www.null.com/home/overview.html ... discarded, already in Q

( 4 )
URL: http://www.null.com/home/overview.html
FILE: www.null.com/home/overview.html
* http://www.null.com/home/synopsis.html ... discarded, already processed
* http://www.null.com/home/index.html ... discarded, already processed
* http://www.null.com/home/synopsis.html ... discarded, already processed
* http://www.null.com/home/order.html ... discarded, already processed
```

执行结束后，在本地文件系统里会创建一个www.null.com目录，这个目录还有一个名为

home的下级子目录。在那个home子目录里可以找到全部经过处理的HTML文件。

19.4 CGI: 帮助Web服务器处理客户数据

19.4.1 CGI简介

因特网最初被开发为一个全球性的在线文档库（大部分文档都是教育和科研方面的）。这些信息通常都被保存为HTML（HyperText Markup Language，超文本标签语言）静态文本的格式。还有一些文档被保存为普通文本格式、Adobe公司开发的可移植文档格式（Portable Document Format，简称PDF）或者扩展标签语言（Extended Markup Language，简称XML；这是一种通用性的标签语言）格式。

与其说HTML是一个语言，不如说它是一个文本格式排版器。它可以表示出字体类型、字号大小、字形变化等方面的设置情况。HTML的主要特点在于它的超文本链接能力，文档中的文本可以通过这样或那样的标记指向另外一个文档里与原始文档有关的内容。这些加上了标记的文本可以通过鼠标操作或其他用户选择机制进行访问。这些（静态的）HTML文档都保存在Web服务器上，当有用户请求时就会发送给客户。

提起因特网和Web服务，就需要有对用户输入的处理。在线零售商需要能够对每一张定单进行处理，在线银行和搜索引擎公司需要为每一个用户建立帐户。为此人们发明了“填表”的办法，而这也成为目前网络站点获取用户个人资料唯一的办法（但Java插件出现后情况已经发生了变化），这就要求为每个客户提交的用户个人数据动态地生成相应的HTML表单。

现今的Web服务器值得称道的只有一件事：接收要求下载某个文件的用户请求并向客户送出那个文件（一个HTML文件）。它们没有“大脑”去处理与特定客户有关的数据，比如那些现场采集的数据。因为这些工作不归Web服务器负责，所以它们会把这类请求转移到外部的应用程序，再由它们创建动态生成的HTML并发送回客户那里去。

这一切都是从Web服务器收到一个客户请求（即GET或POST请求）并调用相应的外部应用程序开始的。服务器做完这些事情以后，就会等待出来HTML结果——同时，客户也在等待。一旦应用程序执行完毕，它就会把动态生成的HTML传递回服务器，再由服务器（最终）传递给用户。服务器收到一个表单，与外部应用程序联系、接收并向客户返回新生成的HTML，这一过程中的所有事情都要通过网络服务器的通用网关接口（Common Gateway Interface，简称CGI）来完成。CGI工作原理的示意图见图19-3，图中给出了控制流和数据流的流向，从用户提交表单开始直到Web网页返回为止。

由用户输入并发送到网络服务器的表单可能包含有需要在某个后端数据库中进行的处理，还可能包含有某种形式的存储。请记住，只要有需要由用户填写的数据域和/或一个“Submit”（提交）按钮或图像，就极可能涉及到某种形式的CGI活动。

创建HTML的CGI应用程序通常是用某个高级程序设计语言编写的，这些高级语言必须有能力接收用户数据、对它们进行处理，再把HTML结果返回给服务器。现在可以使用的高级语言包括：Perl、C、C++和Python，等等。我们将在下一节里介绍如何在cgi模块的帮助下利用Python

语言编写CGI应用程序。

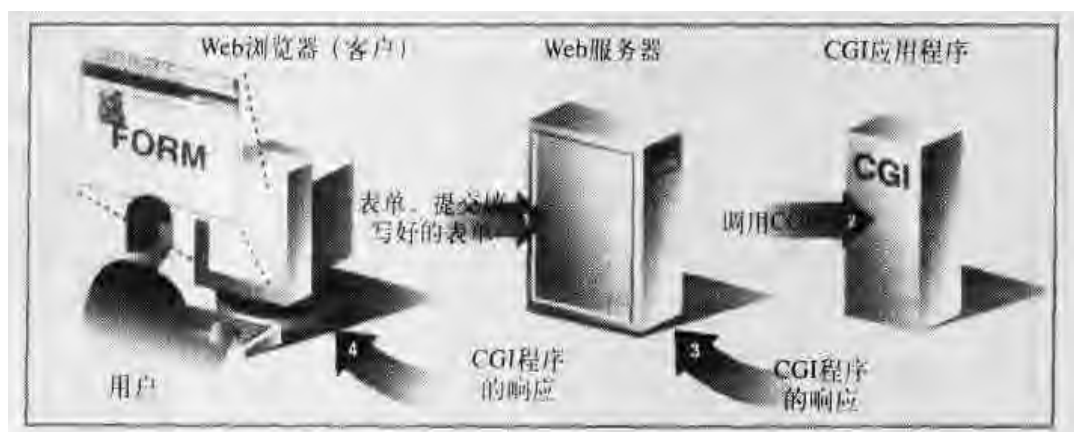


图19-3 CGI的工作原理。CGI代表了网络服务器和外部应用程序之间的交互关系，这些应用程序用来对用户的表单进行处理并生成将最终返回的动态HTML

19.4.2 CGI应用程序

CGI应用程序和正常的程序稍有不同，它们之间的基本差别体现在计算机程序的输入、输出和用户交互作用方面。

在一个CGI脚本程序启动的时候，它必须有对用户提供的数据进行检索的能力，这些程序处理的数据来自网络客户的输入而不是来自服务器机器上的用户或服务器上的磁盘文件。

输出方面的差别是：CGI应用程序输出到标准输出上的数据将被发送回与服务器连接着的网络客户而不是被发送到服务器机器的显示屏幕、GUI窗口或磁盘文件去。这些发送回网络客户的数据由表头及后续HTML语句组成，它们必须满足HTML语言的有关规定。如果不符合有关规定，并且网络客户方是一个浏览器，就会引起一个错误（准确地说是一个因特网服务器错误）；这是因为像浏览器这样的网络客户程序只能对合法的HTML数据（即MIME表头和HTML语句）进行处理。

最后，正如大家预想的那样，CGI脚本程序与用户之间没有任何交互。一切通信都发生在网络客户程序（它代表着一个用户）、网络服务器和CGI应用程序三者之间。

19.4.3 cgi模块

cgi模块里有一个可以完成全部工作的基本类（class），它就是FieldStorage类。一个使用Python语言编写的CGI脚本程序在开始执行时必须对这个类进行实例化，这样就可以（通过Web服务器）从web客户那里把与用户有关的所有信息都读进来。那个被实例化出来的对象是由一个字典形式的对象组成的，该对象中包含着一组键字-键值数据对。键字就是通过表单传递进来的表单数据项的名字，键值就是与之对应的数据。

这些数据值本身又有三种形式可以选择。它们可以是FieldStorage对象（实例）、与FieldStorage对象相似的MiniFieldStorage类的实例，或者由这两种对象组成的一个列表。MiniFieldStorage类用在没有文件上传或表单数据没有分成多个部分的情况里，它的实例只包含着分别对应于名字和数据的键字-键值对。如果在一个表单里相同的数据输入域名字包含了多个

输入数据，它就会是一个列表。

在简单的Web表单里通常只有MiniFieldStorage实例。我们后续内容里的所有例子都遵从这个普遍的规律。

19.5 建立CGI应用程序

19.5.1 制作结果网页

在程序示例19-2里，我们给出了一个简单的Web表单friends.htm的代码。

程序示例19-2 静态表单网页 (friends.htm)

这个HTML文件显示给用户看的表单里有一个空白的用户名输入域和一组单选按钮，用户通过这些单选按钮 (radio button) 做出自己的选择。

```

1  <HTML><HEAD><TITLE>
2  Friends CGI Demo (static screen)
3  </TITLE></HEAD>
4  <BODY><H3>Friends list for: <I>NEW USER</I></H3>
5  <FORM ACTION="/cgi-bin/friends1.py">
6  <B>Enter your Name:</B>
7  <INPUT TYPE=text NAME=person VALUE="NEW USER" SIZE=15>
8  <P><B>How many friends do you have?</B>
9  <INPUT TYPE=radio NAME=howmany VALUE="0" CHECKED> 0
10 <INPUT TYPE=radio NAME=howmany VALUE="10" > 10
11 <INPUT TYPE=radio NAME=howmany VALUE="25"> 25
12 <INPUT TYPE=radio NAME=howmany VALUE="50"> 50
13 <INPUT TYPE=radio NAME=howmany VALUE="100"> 100
14 <P><INPUT TYPE=submit></FORM></BODY></HTML>

```

从上面这段代码可以看出，这个表单是由两个输入变量组成的，它们分别是person和howmany。这两个数据输入域的值都将被传递到我们的CGI脚本程序friends1.py里面去。

从上面这段代码里还可以看出，我们把CGI脚本程序安装到本地主机缺省的cgi-bin目录里 (请看那个“Action”链接) 去了。如果这个信息与读者的开发环境不相符，请在对这个WEB主页和CGI脚本程序进行测试之前修改表单的动作。此外，因为表单动作里少了一个METHOD标记，所以一切请求都将是缺省的类型，即都是GET请求。我们之所以选择使用GET请求是因为我们不想让表单有过多的数据输入域，也是因为我们想让查询字符串能够出现在“Location” (在不同的机器上可能会是“Address”或“Go To”) 栏里，这样我们就能够清楚地看到发送给服务器的是哪个URL。

我们来看看friends.htm在Web浏览器里生成的画面。图19-4给出的是在UNIX环境中用Netscape Communicator 4看到的主页画面，图19-5给出的是在Windows环境中用微软公司的IE5看到的主页画面。

用户给出输入之后按下Submit (提交) 按钮就会把这个表单发送给服务器 (用户在文本输入域里按下回车键也将是同样的效果)。此时，程序示例19-3中的friends1.py脚本程序将通过CGI被调用执行。

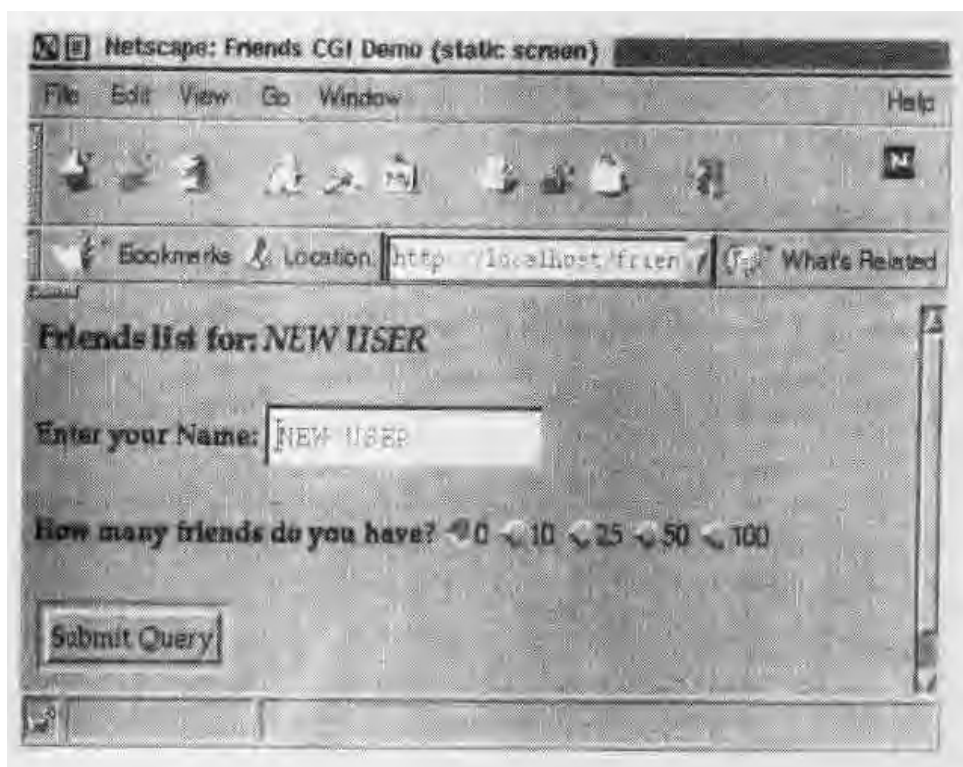


图19-4 UNIX环境中Netscape 4里看到的表单主页 (friends.htm)

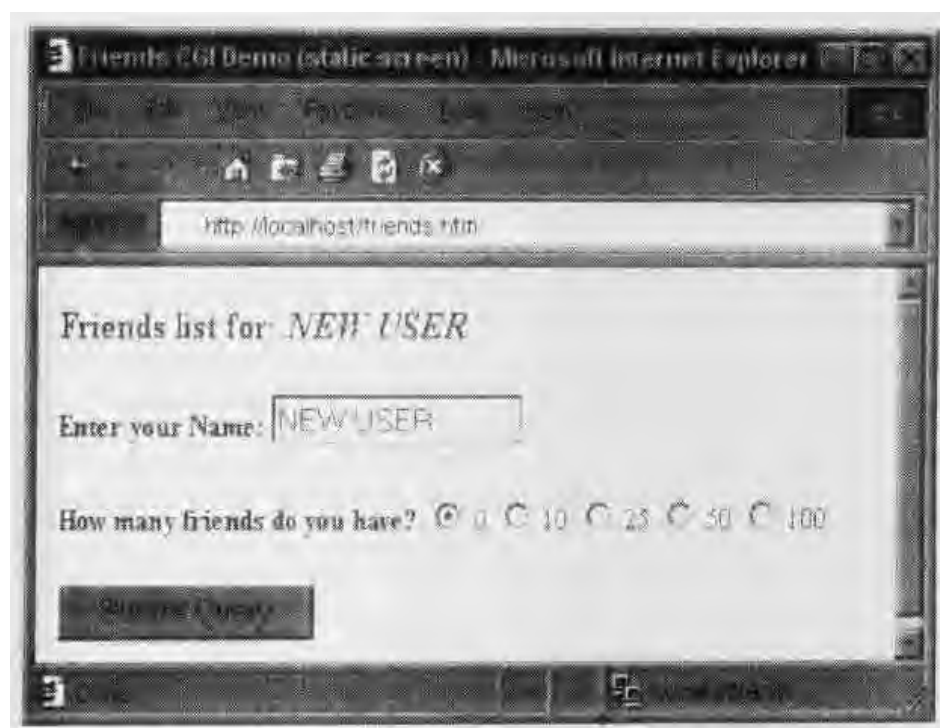


图19-5 Windows环境中IE 5里看到的表单主页 (friends.htm)

程序示例19-3 生成结果画面的CGI代码 (friends1.py)

这个CGI脚本程序从表单里取出用户在person和howmany域里输入的数据，然后用这些数据创建一个动态生成的结果画面

```
1 #!/usr/bin/env python
```

```

2
3 import cgi
4
5 reshtml = '''Content-Type: text/html\n
6 <HTML><HEAD><TITLE>
7 Friends CGI Demo (dynamic screen)
8 </TITLE></HEAD>
9 <BODY><H3>Friends list for: <I>%s</I></H3>
10 Your name is: <B>%s</B><P>
11 You have <B>%s</B> friends.
12 </BODY></HTML>'''
13
14 form = cgi.FieldStorage()
15 who = form['person'].value
16 howmany = form['howmany'].value
17 print reshtml % (who, who, howmany)

```

这个脚本程序具备读取表单输入并对它们进行处理的程序能力，并且能够把处理后得到的HTML结果主页返回给用户。这个脚本程序里全部“真正的”工作都发生在区区四行Python代码中（第14~17行）。

form变量是一个FieldStorage实例，它包含着用户在person和howmany域里输入的数据值。我们把这些值分别读到Python变量who和howmany里去。那个reshtml变量包含着准备返回给用户的HTML文本的主体，其中有几个需要动态填写的输出域，我们把从表单里读到的数据填上去就行了。

编程技巧：把HTTP表头和HTML隔离开

[CT]

让CGI初学者头疼不已的一件事是：当需要把处理结果从CGI脚本程序发送回用户时，在发送任何HTML之前必须先返回正确的HTTP表头。而为了把这些表头和结果HTML隔开，就需要在这两部分数据之间插入几个换行符。在friends1.py示例的第5行我们就是这样做的，本章后面内容里的代码也是如此。

图19-6给出了一个示例性的结果画面，这是用户输入“erick allen”做为姓名并点击了那个表示“有10个朋友”的单选按钮时得到的结果。这个屏幕画面是在Windows环境中由老一点的IE 3显示的。

如果你就是某个Web站点的创建者，你可能会说：“瞧，要是我能把人名中的第一个字母自动转换为大写的该有多好，特别是人们忘了这样做的时候，不就更好了吗？”这个工作是和易用Python语言的CGI完成的。（我们马上就要这么做了！）

请注意“Address”栏里的表单动作URL，注意在提交GET请求时表单变量和它们的值是怎样加到URL地址的后面去的。此外，你是否注意到在前面friends.htm主页的标题里有一个单词“static”（静态的），而friends1.py生成的输出画面上其标题里有一个单词“dynamic”（动态的）？我们这样做是有原因的，就是为了表明friends.htm文件是一个静态的文本文件，而这个结果主页是动态生成的。换句话说，结果主页的HTML并不是一个存在于磁盘上的文本文件；它是由我们的CGI脚本程序生成的并返回的——操作时可以把它看做是一个本地文件。

在我们的下一个例子里，我们将改进我们的CGI脚本程序，让它绕过静态文件，成为多面手。

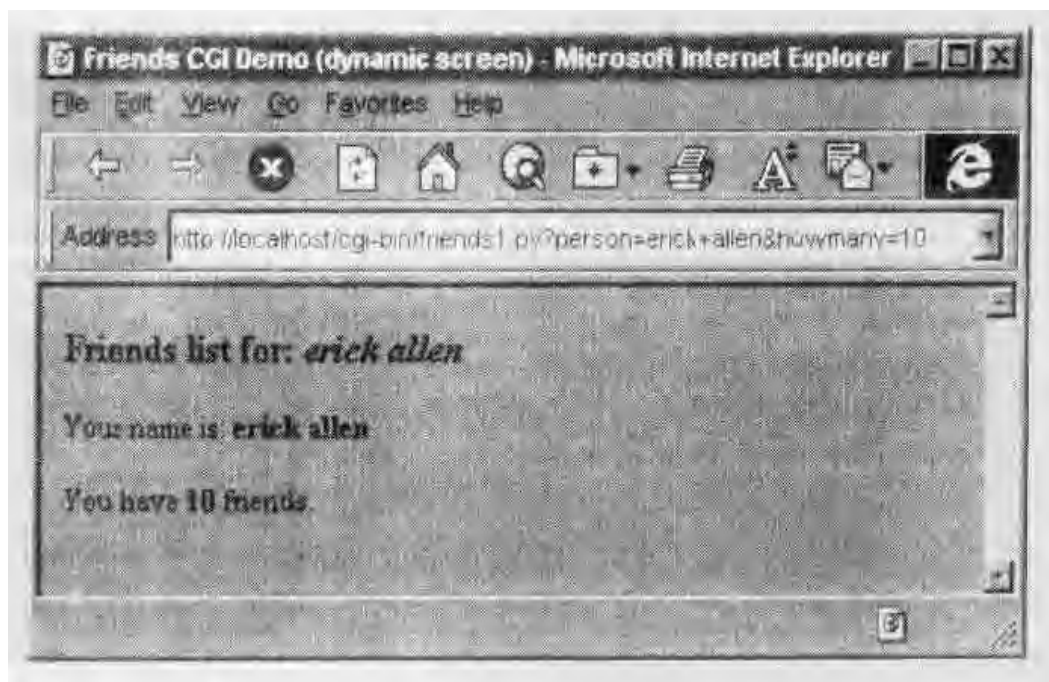


图19-6 Windows环境中IE 3里看到的结果主页

19.5.2 制作表单和结果主页

我们淘汰掉friends1.py，把它合并到friends2.py里去。现在，这个脚本程序可以生成表单主页和结果主页两样东西了。但我们怎么才能知道应该生成哪个主页呢？是这样的，如果有表单数据发送给我们，就表示我们应该生成一个结果主页。如果我们什么信息也没有收到，就表示我们应该生成一个表单主页让用户输入他或她的数据。

我们这个新的friends2.py脚本程序在程序示例19-4里给出。

程序示例19-4 生成表单和结果主页 (friends2.py)

我们把friends.html和friends1.py合并到一起得到friends2.py。这个新的脚本程序现在可以通过动态生成的HTML分别输出表单和结果两个主页，并且知道应该输出哪个主页。

```

1  #!/usr/bin/env python
2
3  import cgi
4
5  header = 'Content-Type: text/html\n\n'
6
7  formhtml = '''<HTML><HEAD><TITLE>
8  Friends CGI Demo</TITLE></HEAD>
9  <BODY><H3>Friends list for: <I>NEW USER</I></H3>
10 <FORM ACTION="/cgi-bin/friends2.py">
11 <B>Enter your Name:</B>
12 <INPUT TYPE=hidden NAME=action VALUE=edit>
13 <INPUT TYPE=text NAME=person VALUE="NEW USER" SIZE=15>
14 <P><B>How many friends do you have?</B>
15 %s
16 <P><INPUT TYPE=submit></FORM></BODY></HTML>'''
17
18 fradio = '<INPUT TYPE=radio NAME=howmany VALUE="%s" %s> %s\n'
19
20 def showForm():

```

```

21     friends = ''
22     for i in [0, 10, 25, 50, 100]:
23         checked = ''
24         if i == 0:
25             checked = 'CHECKED'
26         friends = friends + fradio % \
27             (str(i), checked, str(i))
28
29     print header + formhtml % (friends)
30
31 reshtml = '''<HTML><HEAD><TITLE>
32 Friends CGI Demo</TITLE></HEAD>
33 <BODY><H3>Friends list for: <I>%s</I></H3>
34 Your name is: <B>%s</B><P>
35 You have <B>%s</B> friends.
36 </BODY></HTML>'''
37
38 def doResults(who, howmany):
39     print header + reshtml % (who, who, howmany)
40
41 def process():
42     form = cgi.FieldStorage()
43     if form.has_key('person'):
44         who = form['person'].value
45     else:
46         who = 'NEW USER'
47
48     if form.has_key('howmany'):
49         howmany = form['howmany'].value
50     else:
51         howmany = 0
52
53     if form.has_key('action'):
54         doResults(who, howmany)
55     else:
56         showForm()
57
58 if __name__ == '__main__':
59     process()

```

我们在脚本程序里都进行了哪些修改？我们来看看这个脚本程序里面的几段代码。

逐行解释

第1~5行

在正常的启动行和模块导入语句行以外，我们把HTTP协议的MIME表头和其余的HTML主体隔离开了，这样做的原因是我们想把它用在由程序返回的两种类型的主页（表单主页和结果主页）上，同时又不想重复输入相同的文字。在需要输出某个主页时，我们将把这个表头字符串加到相应的HTML主体上去。

第7~29行

前面两个模块合并为friends2.py以后，这部分CGI脚本程序代码对应着原来的表单主页friends.htm。我们把表单主页里面的文本赋值给一个变量formhtml，再用一个字符串变量fradio代表原来的单选按钮。本来我们可以照抄friends.htm中与单选按钮对应的那部分HTML文本，但我们想在这里展示一下如何利用Python语言生成更加动态化的输出——请看22~27行的for循环。

showForm()函数负责生成一个表单,用户就在这个表单里进行输入。它为那些单选按钮建立了一系列文本,把所有的HTML输出行合并到一起后赋值给formhtml,再加上这个表单的表头,最后通过把整个字符串发送到标准输出的办法把数据块返回给客户。

这段代码里面有几个值得注意的地方。第一处是代码第12行表调用action处的“hidden”变量和所包含的值“edit”。这个域是选择显示哪个主页(即是显示表单主页还是显示结果主页)的唯一地点。在代码的第53到56行可以看到这个域发挥了作用。

此外,在制作全体按钮的循环里我们对按下了哪个按钮进行了检查;如果用户没有按下任何按钮,就使用“0”单选按钮做为缺省值。我们还可以只用一行语句就完成对单选按钮的摆放布局和/或它们的值所进行的修改(第18行),这样做不需要使用多行语句。这样做还能够给检查按下了哪个按钮的逻辑以更大的灵活性——请参考这个脚本程序的下一个改进版本friends3.py。

你可能会想了:“既然我可以检查用户是否输入了person或howmany,那为什么还要使用一个action变量呢?”这个问题问的很好,在我们这个例子里确实是只使用person或howmany就能完成预定的任务。

可以这样说,无论是它的名字还是它的作用——使代码更容易理解,action变量的出现肯定会多引起一些人们的注意。对person或howmany变量而言,我们用的是它们的值,对action变量来说,我们把它用做一个操作标志。

出现action的另外一个原因是我们还需要利用它来确定到底要生成哪一个主页。说得更明确些,我们将要显示的是一个带person变量(而不是一个结果主页)的表单——如果只有在“有一个person变量”的情况下才能显示一个表单(或主页)的话,代码会因此变得很脆弱。

第31~39行

显示结果主页的代码和friends1.py里的是完全一样的。

第41~56行

因为这个脚本程序会生成两个不同的主页,所以我们编写了一个总控性质的process()函数来获取表单数据并决定需要采取什么样的动作。process()的主程序部分和friends1.py的主程序部分的代码很相似。两者之间只有两个比较大的区别。

因为我们的脚本程序可能得到也可能得不到预期的输入域(比如说,第一次调用这个脚本程序来生成一个表单主页时是不会向服务器传送任何输入域的),所以我们必须把我们对表单上的数据输入域的检索操作“括”在if语句里,这样做的目的是为了检查它们是不是真的有这些输入域在那里。此外,我们在前面已经说过action域可以帮助我们确定需要显示哪一个主页,具体完成这个确认工作的代码是第53到第56行。

在图19-7和图19-8里,你先看到的是我们这个脚本程序生成的表单主页(用户已经输入了姓名,也按下了某个按钮),然后是结果主页——它也是我们这个脚本程序生成的。

读者在查看“Location”或“Go to”栏里面的URL地址时将不再会看到前面的图19-4或图19-5里引用了静态文件friends.htm的情况。

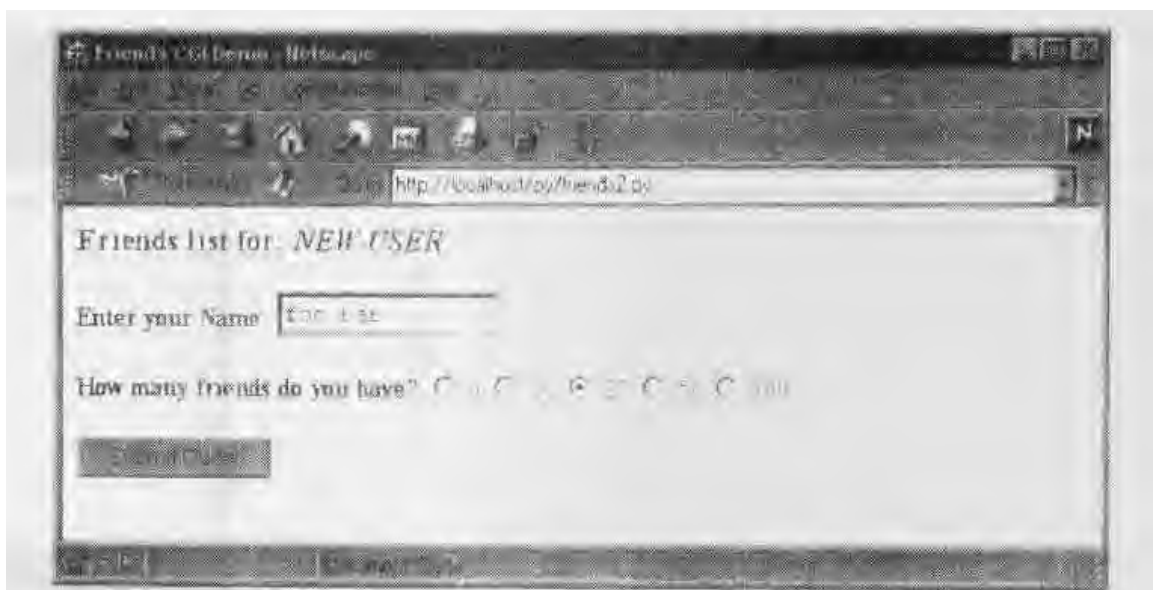


图19-7 Windows环境中Netscape 4里看到的表单主页

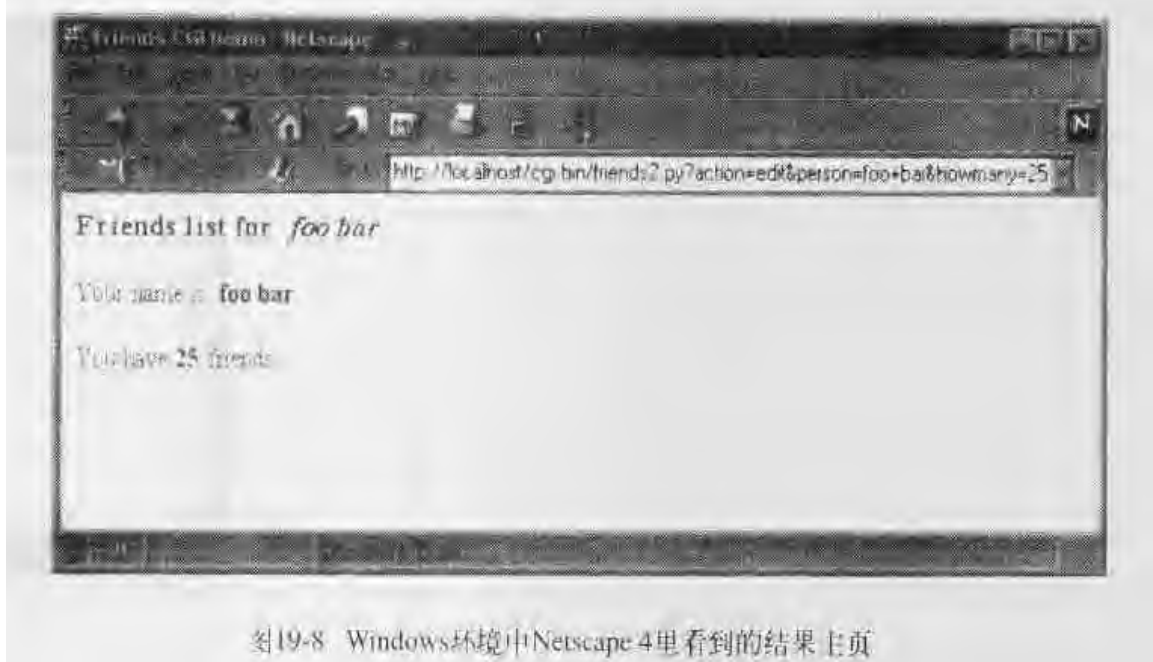


图19-8 Windows环境中Netscape 4里看到的结果主页

19.5.3 完全以交互方式运行的Web站点

本书的最后一个程序示例将使前面的开发工作到达循环的终点。在前面给出的几个程序示例里，用户要在一个表单主页里输入自己的资料，我们随后对这些数据进行处理并输出一个结果主页。现在，我们将在这个结果主页里加上一个链接，允许用户通过这个链接回退到表单主页去。但在用户回退到表单主页时我们给出的不是一个空白的表单而是要把用户刚才输入的数据填写好。我们还准备再添加一些出错处理，让大家对如何完成这些工作有一个认识。

下面的程序示例19-5是我们的升级终结版friends3.py。

程序示例19-5 完全交互方式的用户操作和出错处理 (friends3.py)

这个脚本程序给结果主页添加了一个用来返回表单主页的链接，并在返回到表单主页时把用户已经提供的资料都填写好。这样我们就最终完成了对这个CGI应用程序的开发，用户能够在这个站点上体会到完全以交互方式进行的网络冲浪经验。同时，我们的应用程序现在能够进行一些简单的出错检查——如果用户没有按下任何按钮就会通知用户出现了这样的错误。

```

1  #!/usr/bin/env python
2
3  import cgi
4  from urllib import quote_plus
5  from string import capwords
6
7  header = 'Content-Type: text/html\n\n'
8  url = '/cgi-bin/friends3.py'
9
10 errhtml = '''<HTML><HEAD><TITLE>
11 Friends CGI Demo</TITLE></HEAD>
12 <BODY><H3>ERROR</H3>
13 <B>%s</B><P>
14 <FORM><INPUT TYPE=button VALUE=Back
15 ONCLICK="window.history.back()"></FORM>
16 </BODY></HTML>'''
17
18 def showError(error_str):
19     print header + errhtml % (error_str)
20
21 formhtml = '''<HTML><HEAD><TITLE>
22 Friends CGI Demo</TITLE></HEAD>
23 <BODY><H3>Friends list for: <I>%s</I></H3>
24 <FORM ACTION="%s">
25 <B>Your Name:</B>
26 <INPUT TYPE=hidden NAME=action VALUE=edit>
27 <INPUT TYPE=text NAME=person VALUE="%s" SIZE=15>
28 <P><B>How many friends do you have?</B>
29 %s
30 <P><INPUT TYPE=submit></FORM></BODY></HTML>'''
31
32 fradio = '<INPUT TYPE=radio NAME=howmany VALUE="%s" %s> %s\n'
33
34 def showForm(who, howmany):
35     friends = ''
36     for i in [0, 10, 25, 50, 100]:
37         checked = ''
38         if str(i) == howmany:
39             checked = 'CHECKED'
40         friends = friends + fradio % \
41             (str(i), checked, str(i))
42     print header + formhtml % (who, url, who, friends)
43
44 reshtml = '''<HTML><HEAD><TITLE>
45 Friends CGI Demo</TITLE></HEAD>
46 <BODY><H3>Friends list for: <I>%s</I></H3>
47 Your name is: <B>%s</B><P>
48 You have <B>%s</B> friends.
49 <P>Click <A HREF="%s">here</A> to edit your data again.
50 </BODY></HTML>'''
51
52 def doResults(who, howmany):
53     newurl = url + '?action=reedit&person=%s&howmany=%s' % \
54         (quote_plus(who), howmany)
55     print header + reshtml % (who, who, howmany, newurl)
56
57 def process():

```

```

58     error = ''
59     form = cgi.FieldStorage()
60
61     if form.has_key('person'):
62         who = capwords(form['person'].value)
63     else:
64         who = 'NEW USER'
65
66     if form.has_key('howmany'):
67         howmany = form['howmany'].value
68     else:
69         if form.has_key('action') and \
70             form['action'].value == 'edit':
71             error = 'Please select number of friends.'
72         else:
73             howmany = 0
74
75     if not error:
76         if form.has_key('action') and \
77             form['action'].value != 'reedit':
78             doResults(who, howmany)
79         else:
80             showForm(who, howmany)
81     else:
82         showError(error)
83
84 if __name__ == '__main__':
85     process()

```

friends3.py和friends2.py的差异也不是很大。我们请读者来比较这两者的区别；下面是对几个比较大的修改的简单总结：

简洁版逐行解释

第8行

我们从表单里提取出URL地址，因为现在有两个地方需要用到它了——结果主页成为了我们的新主页。

第10～19、69～71、75～82行

与前面几个程序示例相比，这个脚本程序将多出一个出错画面。这些行是与这个新功能有关的代码。如果用户一个（代表朋友个数的）单选按钮也没选，就不会向服务器传递那个howmany域。如果真的出现这样的情况，showError()函数将会把出错主页返回给用户。

出错主页上还有一个用JavaScript实现的“Back”按钮。因为按钮属于input（输入）类型，所以我们需要一个表单，但不需要有任何动作，因为我们想做的只不过是简单地回退到浏览历史记录里的前一个主页而已。虽然我们的这个脚本程序现在只支持（检查、测试）一种错误，我们还是用了一个具有通用意义的error变量；如果我们想在今后对这个脚本程序做进一步的开发以增加检测更多的错误，这样做的好处是不言而喻的。

第27、38～41、49、52～55行

这个脚本程序的目的之一是创建一个能够让用户从结果主页回退到表单主页去的有意义的链接。实现这个链接的目的是为了让用户能够在输入有误的情况下返回到表单主页里对自己刚才输入的数据进行修改。那个新的表单主页只有在它包含着用户刚才输入过的数据的时候才有价值。（如果用户不得不重新从头输入资料的话，那他肯定会不高兴！）

为了实现这一目的，需要我们把当前数据嵌入到将被修改的表单里去。在程序的第27行里，我们给人名增加了一个值（VALUE）属性，如果给出了这个值，它就会被插入到人名输入域里去。很明显，在最初的表单主页上它应该是空的。在第38到第41行里，我们把单选按钮框设置为用户刚选的朋友个数。最后，在第49行以及第52到第55行的修改函数doResults()里，我们用现有资料创建了一个链接，它的作用是把用户“返回”到被我们修改过的表单主页去。

第62行

最后，我们增加了一个我们认为能够给应用程序带来一些专业感受的简单功能。在脚本程序friends1.py和friends2.py对应的画面上，用户输入的做为其姓名的文字是按原样照搬照抄的。读者应该已经在前面那些画面里注意到：如果用户没有把其姓名的第一个字母输入为大写字母，在结果主页里也将显示为他输入的样子。我们增加了一个把用户姓名的第一个字母转换为大写字母的string.capitalize()函数的调用。string.capitalize()函数会把传递给它的字符串里面所有单词的第一个字母转换为大写字母。不管它是不是我们想要的功能，我们愿意让大家都知道有一个这种功能的函数。

我们在下面给出四个屏幕画面，它们展示了用户与这个CGI表单和脚本程序交互的操作过程。

在图19-9所示的第一个画面里，我们调用friends3.py显示出我们已经很熟悉的表单主页。我们输入一个姓名“bar foo”，但故意不按任何单选按钮。提交这个主页后将导致出错，这可以在第二个画面（图19-10）里看到。

我们单击“Back”按钮，选取“50”单选按钮，再重新提交我们的表单。结果主页如图19-11所示，这也是一个熟悉的画面，但这次在其底部多了一个链接。这个链接将把我们带回到表单主页去。

新的表单主页与最初的那个表单主页唯一的区别是用户刚才填写的数据现在已经成为“缺省的”设置值了，即这些值已经在表单上填写好了。这一点可以从图19-12上看起来。



图19-9 Windows环境中Netscape 3里看到的初始主页

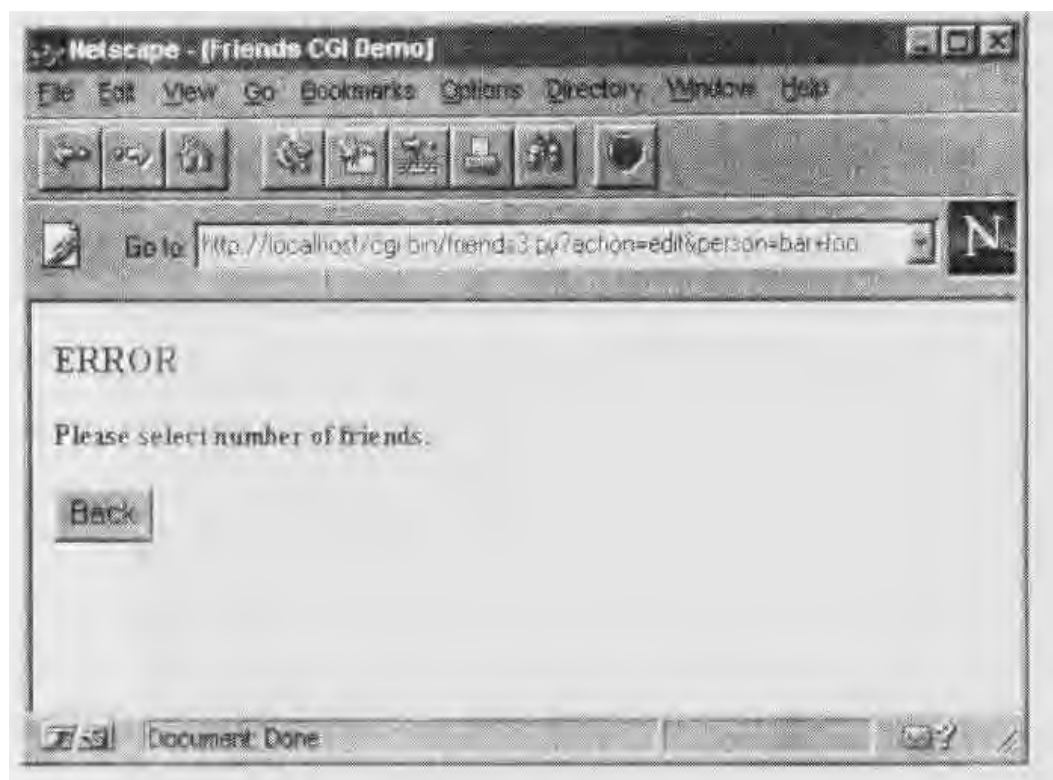


图19-10 出错提示主页（用户非法输入）

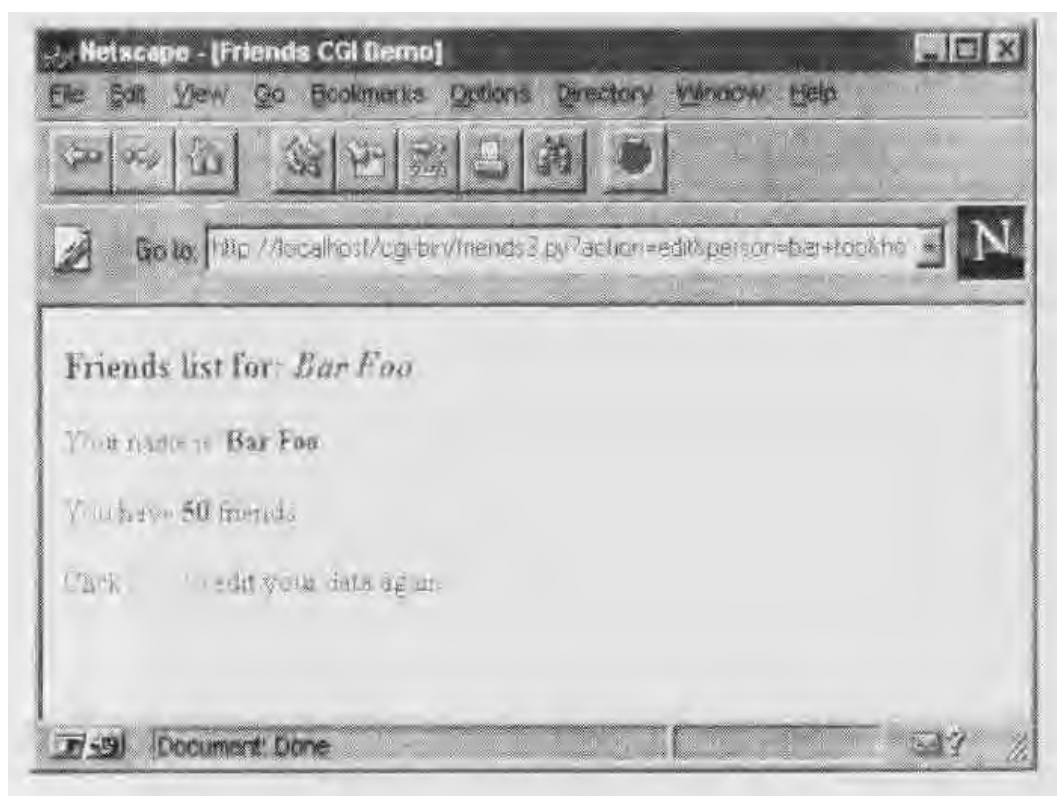


图19-11 结果主页（合法输入）

现在用户可以对数据输入域进行修改，也可以重新提交他们的表单。

读者肯定已经注意到随着我们的表单和数据变得越来越复杂，所生成的HTML也越来越复杂，对本来就比较复杂的结果主页来说就更是如此。如果遇到生成HTML文本与你的应用程序难以协

调的情况，可以考虑使用一个诸如HTMLgen之类的Python模块，这是一个专门用来生成HTML文本的外部模块。



图19-12 填写好当前资料的输入修改主页

19.6 高级CGI

从现在开始，我们将向大家介绍一些CGI程序设计方面的高级论题。这些论题包括：cookie的使用（cookie是被保存在客户方机器里的缓冲数据）、在同一个CGI数据输入域里使用多个值，以及通过multipart（包含多个组成部分的）表单的提交操作上传文件等。为了节约篇幅，我们将用一个程序示例同时向大家展示这三个功能。我们先来看看包含多个组成部分的表单的提交操作。

19.6.1 包含多个组成部分的表单的提交操作和文件的上传

目前，CGI技术只允许两种形式的表单编码，它们分别是“application/x-www-form-urlencoded”（应用程序/x-www-表单-经过编码的URL）和“multipart/form-data”（多组成部分/表单-数据）。因为“application/x-www-form-urlencoded”是缺省的设置情况，所以永远不必在FORM标记里像下面这样指定编码方案：

```
<FORM enctype="application/x-www-form-urlencoded" . . . >
```

但要是使用了multipart（包含多个组成部分的）表单，就必须像下面这样明确地给出编码方案：

```
<FORM enctype="multipart/form-data" . . . >
```

在提交表单时可以使用这两种编码方案的任意一种,但到目前为止,文件的上传还只能通过multipart编码方案来完成。multipart编码方案是由网景公司早年发明的,但已经被微软公司(从IE 4开始)和其他浏览器所采纳。

文件上传是用下面的文件输入类型完成的:

```
<INPUT type=file name= . . .>
```

这条指令会给出一个空白的文本输入域,旁边还有一个按钮;这个按钮使你能够浏览自己文件的目录结构,找出需要上传的文件。在大多数浏览器上,这个按钮上的文字是“Browse”(浏览),但你的具体情况可能会有变化。(比如说,我们将在例子里使用的Opera浏览器,它的这个按钮上面的符号就是省略号“...”。)

如果读者使用了multipart编码方案,那你的Web客户对服务器的表单提交操作就会与带附件的(multipart方式)的电子邮件消息极其相似。这需要使用另外一种编码方案,因为对一个文件进行“urlencode”编码是不明智的,对二进制文件来说就更是如此。信息还是会传送到服务器上去的,但它是用另外一种办法“打包”的。

不管你使用的是缺省的编码方案还是multipart编码方案,cgi模块对它们的处理还是一样的,在表单提交操作中依然使用着键字和与之对应的键值。你还是要像从前那样通过FieldStorage实例来访问数据。

19.6.2 多取值输入域

除文件上传外,我们还将向大家展示如何对具有多个取值的输入域进行处理。最常见的情况是你有一组复选框(checkbox)并允许用户对多种情况进行选择。每个复选框都被标记为相同的数据域名字,但为了把它们区分开来,每个选择都有一个与某个特定的复选框关联着的值。

大家都知道,从用户那里发送到服务器去的数据在表单提交过程中都是键字-键值形式的数据对。在提交多个复选框的时候,同一个键字就会对应着多个键值。在这样的情况下,cgi模块将不再用一个单个的MiniFieldStorage实例来容纳你的数据,它会创建一个由这些实例组成的列表,你需要对这个列表进行遍历以获取不同的键值。不太难办。

19.6.3 cookie

最后,我们还要在我们的例子里使用cookie。如果你对cookie还不很熟悉,那我要告诉你它们只是某个Web站点处的服务器申请放在客户方(比如浏览器)里的一小段数据信息。

因为HTTP是一个“无状态”协议,所以需要一个主页传递到另外一个主页去的信息可以通过在请求里使用键字-键值数据对的办法来完成,你已经在GET请求和本章前面例子的画面里看到过了。我们在前面还见过另外一种完成这一操作的办法,那就是使用hidden(“隐藏”)的表单域(form field),请参考前面几个friends*.py脚本程序中的action变量。这些变量以及它们的值是由服务器来管理的,因为在它们返回给客户的动态生成的主页里必须嵌入这些数据值。

在浏览多个主页时还有一种办法能够用来维持各主页状态之间的连续一致性,这个办法就是把数据保存到客户这一方。这就是我们平常所说的cookie。嵌在表单里的数据不再被保存在从

服务器返回的Web主页里；某个服务器会向客户发出一个请求，让客户保存一个cookie。这个cookie与发出它的服务器所在的网域是链接着的（所以服务器是不能设置和覆盖来自其他Web站点的cookie的），同时它还有一个有效期（这样浏览器才不会被cookie塞得满满的）。

这两个特性与cookie密切相关，除此之外，cookie还有一个代表有关数据项的键字-键值数据对。cookie还可以有一些其他的属性，这些属性可以是某个网域的子路径，也可以是一个“只有在安全环境中才允许收发cookie”的请求。

使用了cookie之后，我们就不必为了跟踪记录某个用户在网上的活动而把数据在主页之间传来传去。虽然反对使用cookie的观点也很多，认为它们在隐私方面有负面作用，但大多数Web站点在使用cookie方面还是很负责的。有关代码完成的工作是这样的：服务器通过在向客户发送该客户请求的文件之前立刻先发送一个表示需要“设置cookie”的表头的办法请求一个客户保存一个cookie。

当cookie在客户方设置好以后，再向服务器发出请求时就会利用HTTP_COOKIE环境变量把那些cookie也发送给服务器。cookie之间用分号分隔，它们是一些“键字=键值”形式的数据对。如果想在你的应用程序里访问这些数据值，就需要对这个字符串进行几次分断操作（既可以使用string.split()，也可以人工分断它）。cookie之间是用分号(;)分隔的，每个键字-键值数据对都要用一个等号(=)隔开。

类似于multipart编码方案，cookie最初也是由网景公司发明的，该公司实现了cookie并编写了关于cookie的第一份技术报告，如今的cookie仍遵守着这份技术报告中的规定。读者可以在下面的Web站点上找到这份文档：

http://www.netscape.com/newsref/std/cookie_spec.html

有关机构正在对cookie做进一步的标准化工作，如果最终推出了关于cookie的标准并淘汰了这份文档，你还可以从Request for Comment（审查申请报告，简称RFC）文档里获取更多最新的资料。在本书付印时，关于cookie的最新审查申请报告是“RFC 2109”号报告。

19.6.4 高级CGI实战

我们将在这一节里给出我们的CGI应用程序advcgi.py，它与我们在本章前面内容里看到的friends3.py脚本程序之间的差异并不是太大。缺省的第一个主页是需要用户填写的表单，这个表单由四个主要的部分组成，它们是：由用户设置的cookie字符串、姓名输入域、程序设计语言复选框清单和文件提交对话框。这个表单的屏幕显示画面请参考图19-13，我们这次使用的是Windows环境下的Opera 4浏览器。

在网景公司和微软公司的浏览器占主导地位的浏览器领域里，我们很少会听到人们提起其他的浏览器，比如Opera和Lynx等等，但它们确实是存在的！我们要告诉大家的是Opera在内存占用和速度方面做的都是非常好的。

我们来看看同一个表单在运行于Linux操作系统上的Netscape里是个什么样子，如图19-14所示；我们这样做的目的是想让大家感觉更自然亲切一些。从图上可以看出，Netscape在文件上传标签里使用的是“Browse”而不是省略号。（本节其他的屏幕显示画面使用的将都是Opera浏览器。）



图19-13 Windows环境下用Opera 4浏览器看到的带上传功能和multipart结构的表单主页

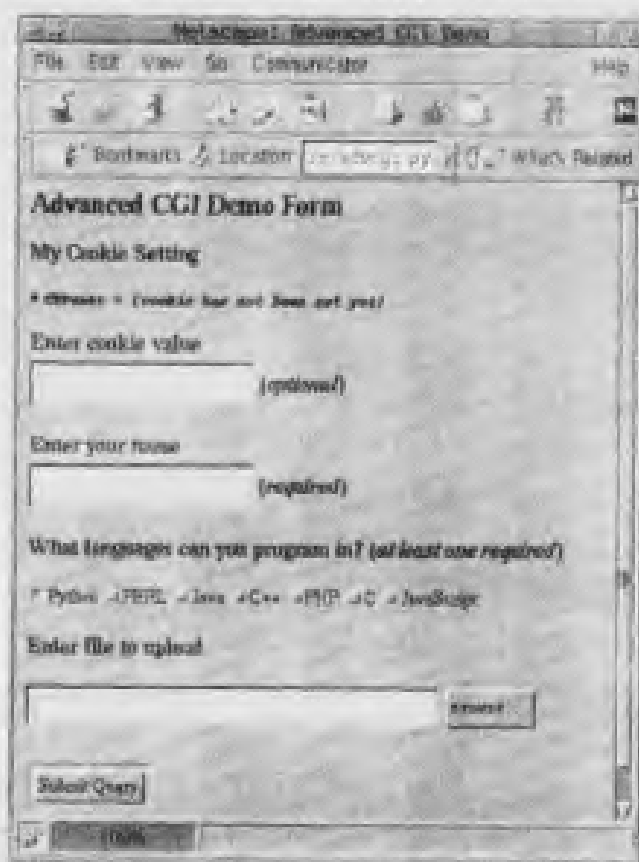


图19-14 Linux操作系统上Netscape-4里看到的同一个高级表单主页

我们把资料输入到这个表单里去，就像图19-15里给出的示范数据那样。



图19-15 我们这个高级CGI演示程序的示范性数据提交画面

数据在提交给服务器的时候使用了multipart编码方案，在服务器上使用FieldStorage实例对这些数据进行检索时也使用了同样的办法。只有一个部分需要我们稍加解释，即对上传文件所进行的检索操作。在我们的应用程序里，我们选择的做法是对文件进行遍历，逐行读入它的内容。如果你不关心文件的长度，一次性读入整个文件的内容也是可以的。

因为这是服务器第一次收到数据，所以它会在第一次把结果主页返回给客户的时候加上“设置cookie”表头，把我们的数据缓冲到浏览器cookie里去。

在图19-16里，读者将看到提交我们的表单数据之后从服务器返回的结果主页。用户输入过的数据域都显示在这个主页上。在最后一个对话框里给出的是一个文件名，它的内容实际上是被上传到服务器以后又送回来显示在图里的。

请注意结果主页底部的那个链接，它能让我们使用同一个CGI脚本程序返回到表单主页里去。

如果读者点击了底部的那个链接，虽然没有向我们的脚本程序提供任何表单数据，但它还是会显示一个表单主页。而且，正如你在图19-17里看到的那样，出现在你面前的可不是一个空白的表单！用户刚才输入的资料都在它里面！既然没有提交过表单数据（既没有被隐藏，也没有在URL里加上查询参数），那这些东西是哪儿来的？这些事情是如何完成的呢？秘密就在于数据已经以cookie的形式被保存在客户方了，事实上，一共有两个cookie。



图19-16 由Web服务器生成并返回的结果主页

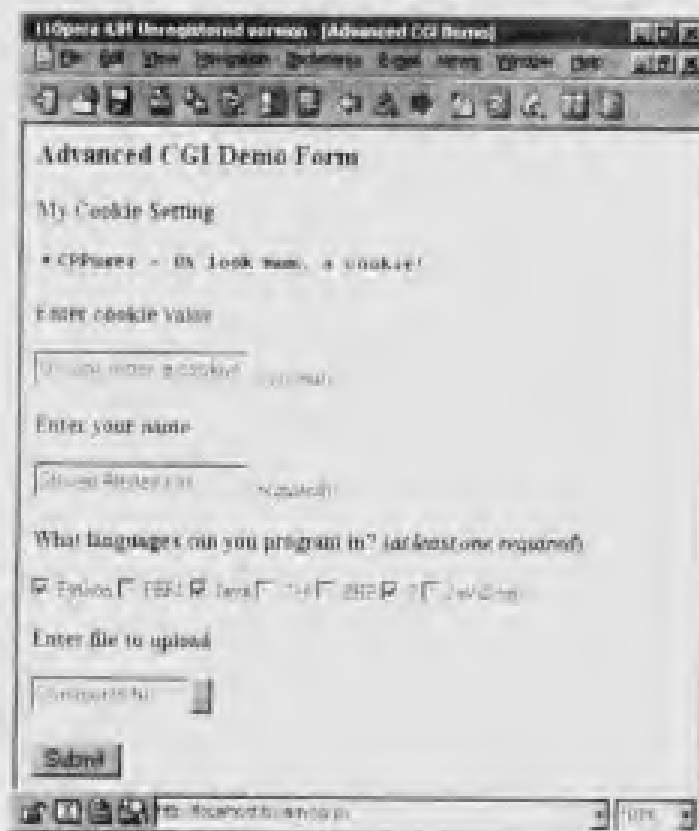


图19-17 数据加载自客户cookie的表单主页

名字是user的那个cookie容纳着用户在表单的“Enter cookie value”(输入cookie值)输入域里输入的字符串;用户的姓名、所熟悉的程序设计语言和被上传的文件等资料被保存在名字是info的cookie里。

脚本程序在检测出客户没有提交任何表单数据的时候就会显示一个表单主页,但在创建这个表单主页之前,它会从客户那里取得cookie(这些cookie是在用户点击那个链接时由客户程序自动传输给服务器的)并把那个表单相应地填写好。这样,当最后显示这个表单的时候,以前输入的所有资料都魔术般地出现在用户面前。

大家肯定都等不及要看看这个应用程序了,它就在后面的程序示例19-6里。

程序示例19-6 高级CGI应用程序 (advcgi.py)

这个脚本定义了一个能够完成所有工作的AdvCGI类(class)。这个类有用来显示表单主页、出错主页、结果主页的方法;还有从客户(一个Web浏览器)那里读出和向客户写入cookie的方法。

```

1  #!/usr/bin/env python
2
3  from cgi import FieldStorage
4  from os import environ
5  from cStringIO import StringIO
6  from urllib import quote, unquote
7  from string import capwords, strip, split, join
8
9  class AdvCGI:
10
11     header = 'Content-Type: text/html\n\n'
12     url = '/py/advcgi.py'
13
14     formhtml = '''<HTML><HEAD><TITLE>
15 Advanced CGI Demo</TITLE></HEAD>
16 <BODY><H2>Advanced CGI Demo Form</H2>
17 <FORM METHOD=post ACTION="%s" ENCTYPE="multipart/form-data">
18 <H3>My Cookie Setting</H3>
19 <LI> <CODE><B>CPPuser = %s</B></CODE>
20 <H3>Enter cookie value<BR>
21 <INPUT NAME=cookie value="%s"> (<I>optional</I></H3>
22 <H3>Enter your name<BR>
23 <INPUT NAME=person VALUE="%s"> (<I>required</I></H3>
24 <H3>What languages can you program in?
25 (<I>at least one required</I></H3>
26 %s
27 <H3>Enter file to upload</H3>
28 <INPUT TYPE=file NAME=upfile VALUE="%s" SIZE=45>
29 <P><INPUT TYPE=submit>
30 </FORM></BODY></HTML>'''
31
32     langSet = ('Python', 'PERL', 'Java', 'C++', 'PHP',
33               'C', 'JavaScript')
34     langItem = \
35         '<INPUT TYPE=checkbox NAME=lang VALUE="%s"%s> %s\n'
36
37     def getCPPCookies(self): # read cookies from client
38         if environ.has_key('HTTP_COOKIE'):
39             for eachCookie in map(strip, \
40                                   split(environ['HTTP_COOKIE'], ';')):
41                 if len(eachCookie) > 6 and \
42                     eachCookie[:3] == 'CPP':
43                     tag = eachCookie[3:7]
44                     try:
45                         self.cookies[tag] = \

```

```

46         eval(unquote(eachCookie[8:]))
47     except (NameError, SyntaxError):
48         self.cookies[tag] = \
49             unquote(eachCookie[8:])
50     else:
51         self.cookies['info'] = self.cookies['user'] = ''
52
53     if self.cookies['info'] != '':
54         self.who, langStr, self.fn = \
55             split(self.cookies['info'], ':')
56         self.langs = split(langStr, ',')
57     else:
58         self.who = self.fn = ''
59         self.langs = ['Python']
60
61     def showForm(self):          # show fill-out form
62         self.getCPPCookies()
63         langStr = ''
64         for eachLang in AdvCGI.langSet:
65             if eachLang in self.langs:
66                 langStr = langStr + AdvCGI.langItem % \
67                     (eachLang, ' CHECKED', eachLang)
68             else:
69                 langStr = langStr + AdvCGI.langItem % \
70                     (eachLang, '', eachLang)
71
72         if not self.cookies.has_key('user') or \
73             self.cookies['user'] == '':
74             cookStatus = '<I>(cookie has not been set yet)</I>'
75             userCook = ''
76         else:
77             userCook = cookStatus = self.cookies['user']
78
79         print AdvCGI.header + AdvCGI.formhtml % (AdvCGI.url,
80             cookStatus, userCook, self.who, langStr, self.fn)
81
82         errhtml = '''<HTML><HEAD><TITLE>
83 Advanced CGI Demo</TITLE></HEAD>
84 <BODY><H3>ERROR</H3>
85 <B>%s</B><P>
86 <FORM><INPUT TYPE=button VALUE=Back
87 ONCLICK="window.history.back()"></FORM>
88 </BODY></HTML>'''
89
90     def showError(self):
91         print AdvCGI.header + AdvCGI.errhtml % (self.error)
92
93         reshtml = '''<HTML><HEAD><TITLE>
94 Advanced CGI Demo</TITLE></HEAD>
95 <BODY><H2>Your Uploaded Data</H2>
96 <H3>Your cookie value is: <B>%s</B></H3>
97 <H3>Your name is: <B>%s</B></H3>
98 <H3>You can program in the following languages:</H3>
99 <UL>%s</UL>
100 <H3>Your uploaded file...<BR>
101 Name: <I>%s</I><BR>
102 Contents:</H3>
103 <PRE>%s</PRE>
104 Click <A HREF="%s"><B>here</B></A> to return to form.
105 </BODY></HTML>'''
106
107     def setCPPCookies(self): # tell client to store cookies
108         for eachCookie in self.cookies.keys():
109             print 'Set-Cookie: CPP%s=%s; path=/' % \
110                 (eachCookie, quote(self.cookies[eachCookie]))
111

```

```

112 def doResults(self):# display results page
113     MAXBYTES = 1024
114     langlist = ''
115     for eachLang in self.langs:
116         langlist = langlist + '<LI>%s<BR>' % eachLang
117
118     filedata = ''
119     while len(filedata) < MAXBYTES:# read file chunks
120         data = self.fp.readline()
121         if data == '': break
122         filedata = filedata + data
123     else: # truncate if too long
124         filedata = filedata + \
125             '... <B><I>(file truncated due to size)</I></B>'
126     self.fp.close()
127     if filedata == '':
128
129         filedata = \
130             '<B><I>(file upload error or file not given)</I></B>'
131     filename = self.fn
132
133     if not self.cookies.has_key('user') or \
134         self.cookies['user'] == '':
135         cookStatus = '<I>(cookie has not been set yet)</I>'
136         userCook = ''
137     else:
138         userCook = cookStatus = self.cookies['user']
139
140     self.cookies['info'] = join([self.who, \
141         join(self.langs, ','), filename], ':')
142     self.setCPPCookies()
143     print AdvCGI.header + AdvCGI.reshtml % \
144         (cookStatus, self.who, langlist,
145         filename, filedata, AdvCGI.url)
146
147 def go(self): # determine which page to return
148     self.cookies = {}
149     self.error = ''
150     form = FieldStorage()
151     if form.keys() == {}:
152         self.showForm()
153         return
154
155     if form.has_key('person'):
156         self.who = capwords(strip(form['person'].value))
157         if self.who == '':
158             self.error = 'Your name is required. (blank)'
159     else:
160         self.error = 'Your name is required. (missing)'
161
162     if form.has_key('cookie'):
163         self.cookies['user'] = unquote(strip(\
164             form['cookie'].value))
165     else:
166         self.cookies['user'] = ''
167
168     self.langs = []
169     if form.has_key('lang'):
170         langdata = form['lang']
171         if type(langdata) == type([]):
172             for eachLang in langdata:
173                 self.langs.append(eachLang.value)
174         else:
175             self.langs.append(langdata.value)
176     else:
177         self.error = 'At least one language required.'

```

```

178         if form.has_key('upfile'):
179             upfile = form["upfile"]
180             self.fn = upfile.filename or ''
181             if upfile.file:
182                 self.fp = upfile.file
183             else:
184                 self.fp = StringIO('(no data)')
185         else:
186             self.fp = StringIO('(no file)')
187             self.fn = ''
188
189         if not self.error:
190             self.doResults()
191         else:
192             self.showError()
193
194 if __name__ == '__main__':
195     page = AdvCGI()
196     page.go()

```

advcgi.py与我们在本章前面内容里看到的friends3.py脚本程序相似之处非常多。它有表单主页、结果主页和出错提示主页需要返回。在我们这个新的脚本程序里，除新增加的高级CGI功能外，我们还使用了更多面向对象的程序设计技巧，我们用 一个带有方法的类代替了一组函数。与那些主页相对应的HTML文本现在是我们这个类的静态数据，这就意味着它们的内容在所有实例里都将保持稳定不变——即使在我们的例子里实际上只用了 一个这样的实例。

逐行（逐代码块）解释

第1~7行

这里是平常的启动行和模块导入语句。大家可能对cStringIO模块不太熟悉，我们曾经在第10章的结尾部分简单介绍过这个模块，在程序示例19-1里还用 过这个模块。这个模块的cStringIO.StringIO()函数能够为字符串创建一个文件风格的对象，随后对那个字符串的访问就类似于打开一个文件并通过其“句柄”访问数据。

第9~12行

在对AdvCGI类进行声明的语句后面，我们定义了这个类里面的两个静态变量header和url，显示各种主页的方法们将要用到它们。

第14~80行

这部分里面的所有代码用来生成并显示表单主页。数据属性的作用从它们的 名字就能看出来，这里就不多说了。getCPPCookies()用来获取由Web客户发送出来的cookie信息；showForm()对全部资料进行组织并把表单主页发送回客户。

第82~91行

这段代码负责与出错主页有关 的处理操作。

第93~144行

结果主页是用这段代码生成的。setCPPCookies()方法用来请求一个客户为我们的应用程序保存cookie；doResult()方法把全体数据组织在一起再输出给客户。

第146~196行

这个脚本程序的主程序部分很简单，先实例化一个AdvCGI主页对象，再调用该对象的go()

方法，这与严格意义上的过程化程序设计过程形成了鲜明的对照。go()方法里面的逻辑将读取所有的输入数据，并决定向用户显示哪一个主页。

如果用户没有给出姓名或者没有选择自己熟悉的程序语言，就显示那个出错主页。如果没有接收到任何输入数据就调用showForm()方法输出那个表单；如果接收到数据就调用doResult()方法显示那个结果主页。

对person数据输入域的处理和我们前面看到的做法是一样的，都是一个键字-键值数据对；但收集程序设计语言的工作有些技巧，因为我们必须检查收到的是一个(Mini)FieldStorage实例还是一个由这种实例组成的列表。我们用熟悉的type()内建函数来完成这项工作。根据用户输入的选择，我们将得到一个由各程序设计语言的名字组成的列表。

我们使用cookie来保存用户输入的数据，这样做可以避免使用其他形式的CGI数据输入域的来回传递。大家可以在获取这些数据的代码部分看到我们并没有调用任何形式的CGI处理功能，这就意味着数据不来自FieldStorage对象。数据是由Web在每一个请求里传递给我们的，而数据的值（用户选择的数据、用来填写后续表单的“现有”资料等）都是从cookie那里获得的。

因为showResults()方法会从用户那里接收新的输入，所以它也有责任对cookie进行设置，也就是需要由它来调用setCPPCookies()方法。而showForm()方法将依次读入已经保存在cookie里面的数据值，并把用户当前的选择显示在表单主页里。这项工作是通过它对getCPPCookies()方法的调用而完成的。

最后是对文件上传工作的处理。不管是否会上传一个文件，我们都要给FieldStorage对象的文件属性分配一个文件句柄；该句柄的数据值属性就是这个文件的整个内容，访问句柄的值就等于把文件的内容放到对应的位置上去。做为另外一个比较好的解决办法，你可以访问其文件指针——这是文件的一个属性，这就可以每次仅读入一行数据，还可以采用其他比较慢一点儿的处理办法。

在我们的例子里，文件的上传只是用户提交操作的一个组成部分，所以我们只是简单地把文件指针传递给doResults()函数，让它把文件里面的数据提取出来。doResults()函数只显示文件开始处的1K字节，之所以这样做一是因为内存空间方面的原因，二是因为完整地显示出一个四兆字节的二进制文件没有什么必要（从效果和用途方面看）。

19.7 Web服务器

到目前为止，我们讨论的内容主要集中在两个方面：一是如何使用Python语言创建Web客户，二是怎样才能帮助Web服务器对CGI请求进行处理。我们知道（并已经在前面的19.2节和19.3节里看到）Python语言既可以用来创建简单的Web客户，也可以用来创建复杂的Web客户。不用说，CGI请求的复杂性也会有相应的变化。

我们还要探讨Web服务器创建方面的问题，而这将是本节的焦点。如果说Netscape、IE、Opera、Mozilla和Lynx等浏览器是一些最流行的Web客户的话，那最流行的Web服务器又是什么东西呢？它们是Apache、Netscape和IIS。如果你想用自己的应用程序取代这些服务器，我们愿意帮助你使用Python语言编写出简单但仍然很实用的Web服务器来。

用Python语言编写Web服务器

既然你已经决定要建立一个这样的应用程序，我自然会认为你应该已经把顾客方面的问题都弄好了，但你将要用到的基本代码其实都已经在Python语言的标准库里为你准备好了。要想建立一个Web服务器，至少要有一个基础服务器和一个“客户请求的处理器”。

基础（Web）服务器是各种服务器的基础，是一个必不可少的东西。它的作用是让客户与服务器之间能够完成必要的HTTP通信。这个基础服务器有一个很般配的名字叫做HTTPServer，你可以在BaseHTTPServer模块里找到它。

“客户请求的处理器”是完成“Web服务”的大部分工作的一个软件。它对客户的请求进行处理并返回相应的文件；这些返回文件既可以是静态的，也可以是由CGI动态生成的。这个“处理器”的复杂程度决定了你的Web服务器的复杂程度。Python语言的标准库能够提供三种不同的处理器。

最基本、最简单的处理器叫做BaseHTTPRequestHandler，在BaseHTTPServer模块里也可以找到它——别忘了这个模块里还有基础Web服务器呢。这个处理器除了接收客户请求以外什么事情也不做，所以你必须自己完成所有的工作，就象我们下面的myhttpd.py服务器里做的那样。

SimpleHTTPServer模块里的SimpleHTTPRequestHandler客户请求处理器是在BaseHTTPRequestHandler的基础上建立的，它以一种相当直白的方式实现了标准的GET和HEAD请求。虽然没有什么吸引人的地方，但它已经能够完成简单的工作。

最后出场的是CGIHTTPServer模块里的CGIHTTPRequestHandler客户请求处理器，它以SimpleHTTPRequestHandler为基础并增加了对POST请求的支持。它能调用CGI脚本程序完成相应的处理工作再把生成出来的HTML发送给客户。

对这三个模块和它们的类（class）的总结请看表19-6。

表19-6 Web服务器模块和类

模 块	说 明
BaseHTTPServer	分别提供了基础Web服务器类HTTPServer和基础的客户请求处理器类BaseHTTPRequestHandler
SimpleHTTPServer	包含着SimpleHTTPRequestHandler类，这个类能够对GET和HEAD请求进行处理
CGIHTTPServer	包含着CGIHTTPRequestHandler类，这个类能够对POST请求进行处理，还能调用执行CGI应用程序

为了更好地理解SimpleHTTPServer和CGIHTTPServer模块中那些更先进的处理器的工作原理，我们为BaseHTTPRequestHandler添上一个对GET请求的处理办法。我们在程序示例19-7里给出了一个完全能够实际运行的Web服务器的代码，myhttpd.py脚本程序。

程序示例19-7 简单的Web服务器（myhttpd.py）

这个简单的Web服务器可以读入GET请求，取出一个Web主页（即一个.html文件）并把它返回给做为调用者的客户。它使用了BaseHTTPServer模块里的BaseHTTPRequestHandler，并且实现了一个do_GET()方法使自己能够对GET请求进行处理。

```
1  #!/usr/bin/env python
2
```



```

3  from os import curdir, sep
4  from BaseHTTPServer import \
5      BaseHTTPRequestHandler, HTTPServer
6
7  class MyHandler(BaseHTTPRequestHandler):
8
9      def do_GET(self):
10         try:
11             f = open(curdir + sep + self.path)
12             self.send_response(200)
13             self.send_header('Content-type',
14                               'text/html')
15             self.end_headers()
16             self.wfile.write(f.read())
17             f.close()
18         except IOError:
19             self.send_error(404, \
20                             'File Not Found: %s' % self.path)
21
22  def main():
23      try:
24          server = HTTPServer(('', 80), MyHandler)
25          print 'Welcome to the machine...'
26          print 'Press ^C once or twice to quit.'
27          server.serve_forever()
28      except KeyboardInterrupt:
29          print '^C received, shutting down server'
30          server.socket.close()
31
32  if __name__ == '__main__':
33      main()

```

这个服务器是BaseHTTPRequestHandler的子类，其中只有一个do_GET()方法，当基础服务器接收到一个GET请求时就会调用这个方法。我们将尝试打开从客户那里传递过来的路径，如果文件存在，就返回一个“OK”执行状态（200）并把客户要求下载的文件发送过去；如果文件不存在，就返回一个404执行状态。

main()函数很简单，它对我们Web服务器的类进行实例化，调用它进入我们熟悉的服务器无限循环运行；在出现^C或类似的键盘输入时，就关闭它。如果读者拥有能够运行这个服务器所需要的足够的权限，就会在屏幕上看到它给出的下面这样的可记录输出：

```

# myhttpd.py
Welcome to the machine... Press ^C once or twice to quit
localhost - - [26/Aug/2000 03:01:35] "GET /index.html HTTP/1.0" 200 -

localhost - - [26/Aug/2000 03:01:29] code 404, message
File Not Found: /dummy.html
localhost - - [26/Aug/2000 03:01:29] "GET /dummy.html HTTP/1.0" 404 -

localhost - - [26/Aug/2000 03:02:03] "GET /hotlist.htm HTTP/1.0" 200 -
HTTP/1.0" 200 -

```

当然，我们这个简单的小Web服务器确实太简单了，它甚至不能对纯文本文件进行处理。我们把这个问题留给读者，它是本章最后一节中的一个练习题。

正如你所看到的，如果想用Python语言建立一个Web服务器并让它运行，这个过程并不复杂，具体做法也不太困难。要想改进这个处理器并把它定制为符合自己要求的应用程序，还需要做

大量的工作。《Library Reference》(库函数大全)里有关于本节介绍过的这些模块(和它们的类)的进一步资料。

19.8 相关模块

我们在表19-7里给出了一个模块清单,你会发现它们在Web和因特网开发方面都很有用。

- 分析用模块负责识别特定格式的文档。
- 使用相应的协议模块可以编写出符合POP或IMAP协议要求的电子邮件客户程序。
- Python语言有大量支持大多数二进制文件的电子邮件编码方案和其他面向MIME应用程序的模块。
- 你可以利用相应的模块编写出使用HTTP、FTP、Telnet和NNTP等常见因特网协议的客户程序来。请记住,对你的浏览器所支持的协议——如HTTP和FTP等协议来说,urllib模块提供了比较高层的程序接口,所以只有当你无法从urllib模块里找到你想要的东西时才有必要使用这些底层的协议模块。
- 最后,我们有HTMLgen外部模块和Digital Creations公司出品的商业化Zope(Z Object Publishing Environment, Z对象出版环境)系统。我们在19.5节的末尾简单介绍过HTMLgen模块。当你需要通过CGI脚本程序生成比较复杂的HTML文档时,这个模块确实是非常方便。

表19-7 与Web程序设计有关的模块

模 块	说 明
分析	
htmllib	分析简单的HTML文件
sgmlib	分析简单的SGML文件
xmllib	分析简单的XML文件
robotparse*	在对URL的“可载性”进行分析时分析robots.txt文件
电子邮件客户协议	
poplib	用来建立POP3客户
imaplib	用来建立IMAP4客户
电子邮件和MIME的处理以及数据的编码方案	
mailcap	分析mailcap文件以获取MIME应用程序的代理情况
mimetoole	提供了对经过MIME编码的消息进行处理的函数
rimetypes	提供了MIME类型关联
MimeWriter	生成经过MIME编码的multipart文件
multifile	分析经过MIME编码的multipart文件
quopri	用quoted-printable编码方案对数据进行编码或解码
rfc822	分析符合RFC822规定的电子邮件信头
smtplib	用来创建SMTP(Simple Mail Transfer Protocol, 简单邮件传输协议)客户程序
base64	用base64编码方案对数据进行编码或解码
binascii	用base64、binhex或uu等(模块)对数据进行编码或解码
binhex	用binhex4编码方案对数据进行编码或解码
uu	用uuencode编码方案对数据进行编码或解码

(续)

模 块	说 明
因特网协议	
http-lib ³	用来创建HTTP (HyperText Transfer Protocol, 超文本传输协议) 客户 (在Python 1.6版本里经过了修改以支持HTTP 1.1和SSL)
ftplib	用来创建FTP (File Transfer Protocol, 文件传输协议) 客户
gopherlib	用来创建Gopher客户
telnetlib	用来创建Telnet客户
nntplib	用来创建NNTP (Network News Transfer Protocol, 网络新闻传输协议: 即Usenet) 客户
外部模块/商业软件	
HTMLgen	与CGI一起使用, 用来生成复杂的HTML文档
Zope (不是一个模块)	网络对象出版物和Python语言Web应用程序的开发环境。网址为: http://www.zope.org

Zope是一个源代码开放的Web出版和应用程序开发平台, 它里面到处都是Python代码。它本身就有一部分是用Python语言编写的, 并且人们可以使用以Python语言编写的程序对Zope软件进行扩展。虽然把它列在相关模块这一节里, 但Zope并不是一个模块, 它是Web出版方面一个功能强大的软件系统。

Zope的功能非常强大, 如果简单的CGI和数据库访问能力不能够满足你正在编写的应用程序的需要, 就可以考虑把Zope用做它们的替代品。关于Zope的文字本身就可以构成一本书——也许你马上就能看见一本! 如果读者有意建立复杂的软件系统, 我们建议大家认真研究一下这个系统。

[1.612.0]

robotparser模块是Python 1.6版本新增加的, 同时, 1.6版本里的httplib和urllib模块都经过了修改以支持通过SSL (在19.2.2节里有一个相当简短的介绍) 建立的HTTP连接。此外, 2.0版本里又新增加了一个webbrowser模块, 它的作用是提供了一种与计算机平台无关的启动某个Web浏览器的办法。

19.9 练习

19-1 urllib模块和文件。改进friends3.py脚本程序, 要求把姓名和朋友个数分两栏保存到一个磁盘文件里; 以后再执行这个脚本程序时继续向文件里添加姓名等数据。

附加题: 增加代码, 把这个文件中的内容 (以HTML格式) 全部输出到一个Web浏览器里去。

附加题: 在上一个附加题的基础上再增加一个用来清除文件中姓名等数据的链接。

19-2 urllib模块。编写一个程序, 用户输入一个URL (Web主页或FTP文件均可, 比如: <http://www.python.org>或<ftp://ftp.python.org/pub/python/README>) 之后, 把它下载到你的计算机里, 还使用原来的文件名 (如果原来的文件名在你的系统上不合法, 可以把它修改为与之近似的)。Web主页 (HTTP) 要保存为.html或.htm文件; FTP文件要保留原来的扩展名。

19-3 urllib模块。重新改写程序示例11-2中的grabweb.py脚本程序，该程序下载一个Web主页并显示下载到的HTML文件中第一个和最后一个非空白行；要求使用urlopen()代替urlretrieve()直接对数据进行处理（不是先整个地下载文件以后再对它进行处理）。

19-4 URL和规则表达式。你的浏览器会把你最喜欢的Web站点的URL保存为一个“书签”HTML文件（网景公司的浏览器是这样做的）或保存为“favorites”子目录里的一组“.URL”文件（微软公司的浏览器是这样做的）。请找出你的浏览器使用哪一种办法记录你的“热点链接”，保存在什么地方，以及存储格式是怎样的。请在不影响原来这些文件的基础上提取出URL地址和与之对应的Web站点名称（如果有的话），生成一个两栏的清单做为输出，一栏是站点名称，另一栏是链接；同时要把这些数据保存到一个磁盘文件里去。必要时要对站点名称或URL地址进行截断使每一行输出保持在80列字符以内。

19-5 URL、urllib模块、例外和规则表达式。在前一个问题的基础上增加脚本程序的代码，对每一个你喜欢的链接进行检查。生成一份“死”链接（及其名称）清单的报告，“死”链接指的是不再存在的Web站点或一个已经被删除的Web主页。只输出和保存那些依然有效的链接。

19-6 出错检查。friends3.py脚本程序会在用户没有选择任何（代表朋友个数的）单选按钮时报告出现一个错误。改进这个CGI脚本程序，让它在用户没有输入姓名（即输入为空或空格）时也报告出现一个错误。

附加题：到目前为止，我们讨论的一直都是服务器端的出错检查。请学习JavaScript程序设计并实现客户端的出错检查，编写JavaScript代码对这两种出错情况进行检查，使这些错误在到达服务器之前就被制止。

下面的练习19-7到19-10与Web服务器的访问记录文件和规则表达式有关。Web服务器（以及它们的系统管理员）通常都要管理维护一个“访问记录文件”（通常是Web服务器主目录里的logs/access_log文件），它记录着与客户请求下载的文件有关的信息。经过一段时间之后，这些文件通常都会变得很大，因而需要被转储或截短。为什么不只保留一些统计信息并删除这些文件以节约硬盘空间呢？在下面几个练习里，请读者使用规则表达式帮助完成对Web服务器数据的转储归档和分析。

19-7 统计记录文件里每种类型的请求（GET和POST）各有多少。

19-8 统计成功的主页/数据下载情况：显示导致返回代码为“200”（表示成功，没有出错）的所有链接以及每个链接被访问的次数。

19-9 统计出错情况：显示导致出错（返回代码为4xx和5xx）的所有链接以及每个链接被访问的次数。

19-10 记录IP地址：针对每个IP地址输出一份它下载过的主页/数据的清单以及每个链接的访问次数。

19-11 简单CGI。为某个Web站点创建一个Comment（评论）或Feedback（反馈）主页。通过一个表单获取用户的反馈意见，在你的脚本程序里对数据进行处理，并且要返回一个“致谢”画面。

19-12 简单CGI。创建一个通信录。让用户输入一个姓名、一个电子邮件地址、一个通信地

址,把这些数据记录到一个文件里去(格式由读者自行设定)。类似于前一道练习题,返回一个“感谢填写通信录”主页。再提供一个让用户查看通信录的链接。

19-13 Web浏览器cookie和Web站点注册。改进练习13-4的解决方案,把你的用户-口令资料与Web站点的注册结合起来,使它不再是一个基于文本的菜单系统。

附加题:学习掌握设置Web浏览器cookie,把它们的失效期设定为上次成功登录之后的四个小时。

19-14 股票行情。有许多在线服务允许用户查询股票行情信息。其中有的站点,比如雅虎还允许用户下载这些信息。这个站点上股票行情各项数据之间是用逗号隔开的。请熟悉其中一个站点并且学习如何把股票行情资料下载到你的本地硬盘上。编写一个Python应用程序,要求它不仅能进行资料下载,还能对保存在本地硬盘上某些特定股票代码的数据进行读、分析和显示。

附加题:把你的解决方案与上一个习题结合起来,用你在练习13-13的解决方案中创建的类对用户进行注册和单独下载某只股票的资料。

19-15 股票行情。改进你对上一个习题的解决方案,不把股票行情下载到一个本地文件里。要求直接打开一个Web服务器连接,然后在股票数据下载到你的应用程序的同时对它们进行分析,再把这些信息显示到屏幕上。

编程提示: Python语言和微软公司的COM程序设计

[CN]

在Windows和32位平台上,Python语言能够连接上“组件对象模型”(Component Object Model,简称COM),这是一项微软公司的程序接口技术,它允许对象之间进行“交谈”,或者更高层一些,允许应用程序之间进行“交谈”,这些通信不依赖于任何语言,也不需使用某种规定的格式。在Hammond和Robinson出版的书籍中可以找到关于COM的全部知识。Python语言和COM的结合使人们得以编写出能够与Word或Excel“交谈”的Python脚本程序,多好的机会!

19-16 股票行情和Excel/COM程序设计(仅限于Windows环境)。请熟悉Python语言中的COM程序设计,然后在前一问题解决方案的基础上编写出一个新的应用程序来,要求它能够下载股票行情信息并且把那些数据直接传输到一个Excel电子表里去。你可以选择让用户人工调用这个Python脚本程序来刷新数据;但如果你有一条到因特网的直接连接,就要让这个应用程序在每个工作日里定期刷新那些数据。把你的解决方案和上一个问题结合起来,使之能够自动刷新多只股票各自对应的Excel电子表。

19-17 多线程COM程序设计(仅限于Windows环境)。改进上一问题的解决方案,通过多个线程使数据的下载工作“同时”进行。

19-18 Web数据库应用程序。为你的Web数据库应用程序设计一个数据库方案。在这个多用户的应用程序里,你想让每个人都拥有对数据库全部内容的读权限,但只对自己本人的数据有写权限。举个例子,你的家庭和亲戚“通信录”就是一个这样的数据库。家庭里的各个成员在

成功登录之后会看到一个主页，主页上提供的操作选项有：添加记录、查看本人记录、修改本人记录、删除本人记录、查看全部记录（即整个数据库）等。

设计一个UserEntry类，并且要为这个类的每一个实例创建一个数据库记录。用户注册工作可以用前面任一习题的解决方案来实现。最后，数据库的数据存储方式可以选用任何类型：既可以是MySQL这样的关系数据库，也可以是简单些的Python永久存储模块——比如anydbm或shelve等模块。

19-19 电子商务引擎。使用练习题13-11解决方案中创建的类再加上一些产品库存，我们得到的就是一个潜在的电子商务网站。记得还要让你的Web应用程序支持多顾客，并且要为每个用户提供注册服务。

19-20 字典和cgi模块。大家知道，cgi.FieldStorage()方法返回的是一个字典形式的对象，里面包含着用户提交的CGI变量的键字-键值数据对。这个对象可以用keys()和has_key()等字典方法进行操作。Python 1.5版本给字典增加了一个get()方法，它的作用是返回给定键字的对应键值——用户没有设定的键字要返回它们的缺省键值。FieldStorage对象没有这样的方法。假设我们象下面这样取回表单：

```
form = cgi.FieldStorage()
```

请在cgi.py（你可以把它改名为mycgi.py或其他类似的东西）模块的类定义部分增加一个get()方法，让下面这段代码：

```
if form.has_key('who'):
    who = form['who'].value
else:
    who = '(no name submitted)'
```

能够被替换为如下所示的这一行语句，它使表单看起来更象是一个字典了：

```
howmany = form.get('who', '(no name submitted)')
```

19-21 建立Web服务器。19.7节里myhttpd.py的代码只能读取HTML文件并把它们返回给调用客户。请增加对以“.txt”结尾的普通文本文件的支持。记得要返回正确的MIME类型“text/plain”。

附加题：继续增加对以“.jpg”或“.jpeg”结尾的JPEG文件的支持，它的MIME类型为“image/jpeg”。

19-22 高级Web客户。改进19.3节中的crawl.py脚本程序，让它也能下载“ftp:”形式的链接。原来的crawl.py只忽略使用的“mailto:”链接。现在要求请增加代码使它还忽略“telnet:”、“news:”、“gopher:”和“about:”等链接

19-23 高级Web客户。19.3节中的crawl.py脚本程序只下载出现在Web主页上并链接在同一网站内的.html文件，对同样是该主页合法“文件”的图象不做任何处理/保存。如果用来打开服务器的URL地址里缺少尾缀的斜线字符(/)，它是不会对这样的服务器进行处理的。请在crawl.py里添上几个解决这些问题的类(class)。

从urllib.FancyURLopener类里衍生出一个名为My404URLopener的子类，它只有一个名为http_error_404()的方法，这个方法的功能是：检查一个“404”错误是不是因为URL地址里缺少

尾缀的斜线而引起的。如果是这个原因，它就会添上一个斜线再请求一次（也只请求一次）。如果它还是失败了，就返回一个真正的“404”错误。你必须用这个类的一个实例对`urllib._urlopener`进行设置，只有这样`urllib`才能使用它。

另外还需要从`htmlib.HTMLParser`里推导出一个名为`LinkImageParser`的类。这个子类的构造器先调用它父类的构造器，再初始化一个由Web主页上的图象文件构成的列表。需要对`handle_image()`方法进行覆盖，让它把图象文件的文件名添加到图象文件列表里去（而不象其父类的方法那样丢弃它们）。

第20章 扩展Python语言

在这一章里，我们将讨论如何把在外部编写的代码的功能集成到你的Python环境里去。我们先把这样做的动机介绍给大家，然后一步一步地带领大家走过这项工作的各个步骤。需要提醒大家的是：因为这些扩展基本上都是用C语言编写的，所以读者将在这一章里看到的代码实例都是用纯粹的C语言编写的。

20.1 介绍

20.1.1 什么是扩展

从普遍意义上来说，只要你所写的代码能够被集成或导入到另外一个Python脚本程序里去，就可以把它看做是一个“扩展”。这个新代码既可以用纯粹的Python语言编写的，也可以用某种编译型语言如C和C++（如果你使用的是JPython，就要使用Java）语言编写的。但对扩展更“严格”的定义仅限于后一个范畴，也就是我们这一章将要讨论的话题。

Python语言的优异特性之一就是它的扩展与解释器之间的交互关系和普通的Python模块是完全一样的。Python语言在整体设计方面达到了这样的效果，就是用模块导入这个抽象概念把使用这些扩展的代码中的具体实现细节隐藏起来了。除非客户程序员特意去搜索文件系统，否则他或她是无法分辨出某个具体的模块到底是用Python语言还是用另外一种编译型语言编写的。

编程提示：在不同的计算机平台上创建扩展

[CN]

我们要在这里提醒大家，扩展一般都存在于一个开发环境里，你就在这个环境里对自己的Python解释器进行编译。在手动编译和直接获取二进制执行代码两者之间有一个明确的界限。虽然编译过程要比下载和安装二进制执行代码的过程稍微繁琐一些，但你在定制自己专用的Python版本方面具有最大的灵活性。

如果读者打算创建一个扩展，就必须在一个与之近似的环境里完成这项工作。本章的例子使用的是一个UNIX系统（它自己在缺省的情况下带有编译器），但如果读者有权使用C/C++（或者Java）编译器和一个用C/C++（或者Java）语言编写的Python开发环境，唯一的区别也就在于你编译的手法了。使你的扩展能够被Python世界接受并使用的实际代码在一切计算机平台上都是一样的。

20.1.2 为什么要扩展Python语言

纵观软件工程简短的发展历史，各种程序设计语言其本身总是一副固定不变的面孔。你看到什么就得到什么；要想在一种已经存在的语言里添加新的功能几乎是不可能的。但在现如今

的程序设计环境里，定制个人专用的程序设计环境的能力已经成为人们最需要的功能之一；而这也大大促进了代码的重复使用。TCL和Python等语言就属于最早一批具备了对基本语言进行扩展的能力的语言。可是，就拿Python语言来说吧，既然它已经具有很丰富的功能，为什么还要对它进行扩展呢？之所以这样做有下面几个好理由：

1. *增加额外（非Python语言）功能

扩展Python语言的原因之一是人们需要使用这个语言的内核无法提供的新功能。这项工作其实可以使用纯Python语言或使用某种编译型语言来完成，但要想这样做必须做很多事情——比如创建新的数据类型或者把Python代码嵌入到某个现有的应用程序中去等等。

2. *改进程序瓶颈的执行性能

大家都知道解释型语言在执行速度方面比不上编译型语言，其原因是语言翻译工作是在程序执行期间即时完成的。一般说来，把一段代码移到一个扩展里会提高整体的执行性能。问题在于如果资源占用方面的成本太高的话，这样做并没有什么好处。

比较明智一点的做法是对代码进行一些简单的分析，找出程序执行的瓶颈所在，然后只把那些代码取出来放到一个扩展里去。这样做的效果可以说是立杆见影，而在资源占用方面扩张的也不太多。

3. *保证关键性代码的私密性

编写扩展还基于一个重要的理由，也可以说它是脚本程序语言的一个弊病。就这类语言带给大家的易用性来说，其源代码毫无私密性可言，因为可执行代码就是源代码。

从Python移到某种编译型语言中去的代码可以帮助我们保证关键性代码的私密性，因为你将与一个二进制对象打交道。因为这些对象已经经过了编译，所以对它们的逆向工程也就不象以前那么容易进行；这样，源代码的私密性就得到了加强。这是特种算法、加密或者软件安全性等方面的关键所在。

要想保证代码的私密性还有其他的办法，那就是只传播经过预编译的.pyc文件。在公开真实的源代码（.py文件）和不得不把代码转换为扩展之间，这应该是一个不错的折中。

20.2 用编写扩展的办法扩展Python语言

为Python语言编写扩展需要以下三个步骤：

- 1) 编写应用程序代码。
- 2) 给代码加上程序接口，对它进行“包装”。
- 3) 编译。

在这一节里，我们将依次把这三个步骤详细地介绍给大家。

20.2.1 编写应用程序代码

首先，在把任何代码转变为一个扩展之前先要编写出一个独立的“扩展库”。换句话说，在编写你程序的代码时必须随时提醒自己它将被转换为一个Python模块。在设计函数和对象的时候必须想到其他Python代码将与你的C语言代码相互通信。

然后，编写出测试代码确保你的软件一切正常。你甚至可以把在Python语言里设计main()函

数的开发手段用在C语言里做为测试程序，这样做了以后，如果你的代码经过编译、链接并被加载到一个可执行成分（与共享对象相对的概念）里去了的话，一个这样的可执行部分将成为你的软件库里的测试程序储备。我们在下面给出的这个扩展示例就是这样做的。

我们的试验性示例包括两个C语言函数，我们想把它们添加到Python语言的程序设计环境中来。第一个是递归阶乘函数fac()；第二个叫做reverse()，它是一个简单的字符串倒置算法，它的主要作用是对一个字符串“原地”进行倒置——即返回的字符串里的所有字符其位置与在原来那个字符串里的位置正好头尾倒置，而且不必使用新分配的字符串按逆序对它进行拷贝。因为这需要用到指针，所以在把它们带到Python语言里去之前必须对我们的代码进行认真的设计和严格的调试。

请看程序示例20-1中的第一版Extest1.c扩展。

程序示例20-1 完全用C语言编写的扩展库 (Extest1.c)

下面的代码是我们准备“打包”的C语言函数库，我们的最终目的是想把这些代码用到Python解释器里而去，main()是我们的测试器函数。

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int fac(int n)
6  {
7      if (n < 2) return(1);
8      return((n)*fac(n-1));
9  }
10
11 char *reverse(char *s)
12 {
13     register char t,
14                 *p = s,
15                 *q = (s + (strlen(s) - 1));
16
17     while (s && (p < q))
18     {
19         t = *p;
20         *p++ = *q;
21         *q-- = t;
22     }
23     return s;
24 }
25
26 void main()
27 {
28     char s[BUFSIZ];
29     printf("4! == %d\n", fac(4));
30     printf("8! == %d\n", fac(8));
31     printf("12! == %d\n", fac(12));
32     strcpy(s, "abcdef");
33     printf("reversing 'abcdef', we get '%s'\n", \
34           reverse(s));
35     strcpy(s, "madam");
36     printf("reversing 'madam', we get '%s'\n", \
37           reverse(s));
38 }

```

这段代码由两个函数fac()和reverse()组成，它们分别实现了我们刚才描述的功能。fac()有一个整数参数，作用是递归地计算出它的阶乘结果，当它结束了自己最外层的调用以后，就会把阶乘结果返回给调用者。

程序最后一部分代码是测试用途的main()函数。我们把它用做我们的测试器，通过它把各种各样的参数送入fac()和reverse()。我们的目的是通过这个函数检查我们的代码能否正确工作。

现在对这段代码进行编译。因为许多版本的UNIX都带有gcc编译器，所以我们将使用如下所示的命令：

```
% gcc Extest1.c -o Extest
%
```

运行这个程序，我们给出下面的命令并看到如下所示的输出情况：

```
% Extest
4! == 24
8! == 40320
12! == 479001600
reversing 'abcdef', we get 'fedcba'
reversing 'madam', we get 'madam'
%
```

再强调一次：你必须尽最大努力完善你的代码，因为当把它结合到Python环境里去之后，你肯定不想出现不得不混合调试扩展库以纠正其潜在程序漏洞的局面。换句话说，要把你核心代码的调试与模块集成的调试区别开来。你编写的代码越是接近Python语言的程序接口，你的代码就会越快地被集成到Python环境中。

我们的每个函数都只使用了一个参数，也只返回一个值。它本身也很简短清晰，所以它们与Python的集成应该没有什么问题。请注意：一直到现在我们和Python语言还没有什么瓜葛，只是简单地编写了一个C或C++应用程序而已。

20.2.2 给代码加上程序接口

要想得到一个扩展，整个实现过程都是围绕着“打包”概念展开的，这个概念我们在13.15.1节里介绍过。在设计代码的时候必须遵循这样一条原则：即在Python世界和你用来实现扩展的程序设计语言之间必须能够平滑地过渡。负责处理两者之间过渡问题的接口代码经常被人们称为“烧锅”，意思是如果你的代码想与Python语言的解释器进行对话必须先“熬一熬”。

接口软件由以下四个主要部分组成：

- 1) 加上Python头文件 (header file)。
- 2) 给模块中的每个函数加上“PyObject * Module_func()” Python打包器。
- 3) 给模块中的每个函数加上“PyMethodDef ModuleMethods[]” 数组/表。
- 4) 加上“void initModule()” 模块初始化函数。

1. 加上Python头文件

你要做的第一件事是找出你的Python头文件（即用在include语句中的头文件）放在哪里，确保编译器能够访问到它所在的目录，这个目录一般是/usr/local/include/python1.x或/usr/include/python1.x，其中的“1.x”是你使用的Python版本号——它可能是1.5或2.0。如果你

曾经编译并安装过自己的Python解释器，就应该不会遇到什么问题，因为系统一般都会知道你的文件被安装到哪儿去了。

把Python.h头文件加到你的源代码里去，这条语句应该是下面这个样子：

```
# include "Python.h"
```

这是最简单的部分。现在你就要添加接口软件的剩余部分了。

2. 给模块中的每个函数加上“PyObject * Module_func()” Python打包器

这部分最不好弄。你必须为每个你想在Python环境里访问的函数创建一个“static PyObject*”函数，还要把模块名（以及一个下划线字符“_”）加在函数名的前面。

举例来说，我们想让fac()成为一个能够被Python导入语句（即import语句）识别的函数，而模块最终选定用Exttest做为自己的名字，那我们创建的“打包器”就是Exttest_fac()。而在Python脚本程序的某个地方就可以有一个“import Exttest”语句和一个“Exttest.fac()”调用了（如果使用了“from Exttest import fac”就可以只调用“fac()”）。

打包器的作用是接收Python参数值，把它们转换为C语言数据，然后调用相应的函数完成工作。函数执行结束时还应该返回到Python世界里去，此时又需要先由打包器接收函数的返回值，把它们转换为Python语言数据，再完成程序的返回操作，返回必要的数值。

就拿fac()来说吧，当客户调用Exttest.fac()时我们的打包器就会被调用。我们将接收到一个Python整数，把它转换为一个C语言整数，调用我们的C语言函数fac()得到另外一个整数结果。然后我们把这个返回值再转换为一个Python整数，再从调用中返回。（可以这样想象一下：你正在编写的代码实际上是一个“def fac(n)”函数声明的代理，而你从函数调用中的返回就好像假想中的Python语言函数fac()正在结束执行。）

读者要问了，这些转换都是怎样发生的呢？答案是这样的：从Python语言到C语言要用到PyArg_Parse*()函数；而从C语言返回到Python语言的时候要用到Py_BuildValue()函数。

PyArg_Parse*()函数的作用类似于C语言中的sscanf()函数。它对一个字节流进行处理，根据某些格式字符串把它们分断并组织为与之对应的包容器变量，而这些变量又都具有指针地址。如果变量的分断操作成功了，它们就返回“1”，否则就返回“0”。

Py_BuildValue()函数的作用类似于C语言中的sprintf()函数，它会根据格式字符串把全体参数按你指定的格式转换为一个包含着与之对应的数据值的返回对象。

这几个函数的总结请看表20-1。

表20-1 Python和C/C++之间的数据转换

函数	说 明
Python到C	
int PyArg_ParseTuple()	对从Python语言传递到C语言的（一个表列）参数进行转换
int PyArg_ParseTupleAndKeywords()	类似于PyArg_ParseTuple()，但它还能够分断关键字参数
C到Python	
PyObject* Py_BuildValue()	把C语言数据值转换为一个Python语言的返回对象，既可以是一个单个的对象，也可以是一个单个的表列——这个表列由多个对象构成

C语言和Python语言之间数据对象的转换需要用到一组转换为代码，它们都列在表20-2里面。

表20-2 Python和C/C++之间进行数据转换时的常用代码

格式代码	Python类型	C/C++类型
s	string	char*
Z	string, None	char* 'NULL'
i	int	int
l	long	long
c	string	char
d	float	double
D	complex	Py_Complex*
O	(任意)	PyObject *
S	string	PyStringObject

这些转换代码将出现在相应的格式字符串里，指出数据在两种语言之间传递时应该如何对它们进行转换。注意：转换类型与Java语言中的情况是不同的，因为它所有的数据都是类(class)。Java类型和Python对象之间对应关系请参考JPython文档中的介绍。

下面是我们最终的Extest_fac()打包器函数：

```
static PyObject *
Extest_fac(PyObject *self, PyObject *args) {

    int res;           // parse result
    int num;           // arg for fac()
    PyObject* retval;   // return value

    res = PyArg_ParseTuple(args, "i", &num);
    if (!res) {         // TypeError
        return NULL;
    }
    res = fac(num);
    retval = (PyObject*)Py_BuildValue("i", res);
    return retval;
}
```

第一步是分析从Python接收到的数据。它应该是一个普通整数，所以我们用转换代码“i”来指示这一点。如果这个值确实是一个整数，就把它保存在num变量里；如果这个值不是一个整数，PyArg_ParseTuple()将返回一个NULL，而出现这种情况时我们也会返回一个NULL。在我们的例子里，这将生成一个TypeError例外，通知用户我们预期的是一个整数。

接下来，我们调用fac()，它的参数就是保存在num中的值；计算结果放到res里去，再次使用了这个变量。现在开始构造我们的返回对象，一个Python整数，我们使用的还是转换代码“i”。

Py_BuildValue()创建一个Python语言的整数对象，我们随后返回了这个对象。大功告成！

事实上，如果你创建了嵌套的打包器，就可以缩短你的代码，避免使用过多的变量。但一定要保证你代码的正确性。我们对Extest_fac()函数进行处理，把它进一步缩短为如下所示的样子，注意此时只需要一个变量num：

```
static PyObject *
Exttest_fac(PyObject *self, PyObject *args) {
    int num;
    if (!PyArg_ParseTuple(args, "i", &num)) return NULL;
    return (PyObject*)Py_BuildValue("i", fac(num));
}
```

reverse()函数该怎么办？既然你已经了解如何返回一个单个的值，我们想把reverse()例子稍微变化一下，让它返回两个而不是一个值。我们将以一个表列的形式返回两个字符串，第一个是传递给我们的字符串，而第二个就是新倒置的字符串。

为了向大家展示一些灵活性，我们将把这个函数称为Exttest.doppel()以表示它的行为与reverse()函数有所不同。把代码打包到Exttest_doppel()函数里面去，我们得到如下所示的结果：

```
static PyObject *
Exttest_doppel(PyObject *self, PyObject *args) {
    char *orig_str;
    if (!PyArg_ParseTuple(args, "s", &orig_str)) return NULL;
    return (PyObject*)Py_BuildValue("ss", orig_str, \
        reverse(strdup(orig_str)));
}
```

类似于Exttest_fac()函数，我们只有一个输入值，这次是一个字符串，我们把它保存到orig_str变量里去。注意我们这次用的转换代码是“s”了。接下来调用strdup()得到该字符串的一个拷贝。（因为我们既想返回原来的字符串，又需要一个字符串来完成倒置操作，所以最好的办法就是拷贝一下这个字符串。）由strdup()函数返回的字符串拷贝被立刻送入reverse()，我们得到一个倒置了的字符串。

正如你所看到的，Py_BuildValue()按照格式字符串“ss”的指示把两个字符串放到一起，这就生成了一个由两个字符串构成的表列，就是原来的和倒置后的字符串。故事讲完了吗？对不起，还没有。

我们注意到C语言程序设计中的一个危险：内存流失，即被分配了的内存得不到释放回收。内存流失有点象从图书馆里借书，而且是只借不还。对你申请到的资源，只要你不需它们，就一定要释放它们；而且你应该永远都这样做。我们的代码是如何犯下这个错误的呢（看起来可是够无辜的）？

当Py_BuildValue()把应该返回的Python对象放到一起的时候，它会对传递给它的数据进行复制拷贝。在我们这儿的例子里，那就是两个字符串。问题是这样的：我们为第二个字符串分配了内存，但在结束时没有释放那部分内存，让它流失了。其实我们想做的事情是：先构造出返回对象，然后释放那些在我们的打包器里分配的内存。没有别的选择，只能象下面这样加长我们的代码：

```
static PyObject *
Exttest_doppel(PyObject *self, PyObject *args) {
    char *orig_str;           // original string
    char *dupe_str;           // reversed string
    PyObject* retval;

    if (!PyArg_ParseTuple(args, "s", &orig_str)) return NULL;
    retval = (PyObject*)Py_BuildValue("ss", orig_str, \
```

```

        dupestr=reverse(strdup(orig_str));
        free(dupe_str);
        return retval;
    }

```

我们用dupe_str变量指向新分配的字符串，创建返回对象并用retval来引用它。然后用free()释放了已经分配的内存，再返回到调用者。现在才是真正的结束。

3. 给模块中的每个函数加上“PyMethodDef ModuleMethods[]”数组/表

既然我们这两个打包器都弄好了，就需要把它们列在什么地方好让Python解释器知道如何导入和访问它们。这是由“ModuleMethods[]”数组负责的工作。

这是一个由数组组成的数组，其中的每个数组分别包含着每个函数的有关资料，最后是一个表示“到此为止”的NULL数组。具体到我们的Extest模块，需要创建如下所示的ExtestMethods[]数组：

```

static PyMethodDef
ExtestMethods[] = {
    { "fac", Extest_fac, METH_VARARGS },
    { "doppel", Extest_doppel, METH_VARARGS },
    { NULL, NULL },
};

```

各数组里的第一个元素是将在Python语言里访问的名字；随后是与之对应的打包函数；第三个是常数METH_VARARGS，它表示有一组以表列形式存在的参数。如果我们在打包器里使用的是PyArg_ParseTupleAndKeywords()和关键字参数，就等于用这个操作标志和METH_KEYWORDS常数进行逻辑或OR操作。最后，是一个由两个NULL组成的数组，它及时结束了我们这个只有两个函数的数组清单。

4. 加上“void initModule()”模块初始化函数

最后一个步骤是模块的初始化函数。这段代码将在解释器导入并准备使用我们的模块时被调用执行。在这段代码里，我们以模块名和“ModuleMethods[]”数组名为参数调用了一次Py_InitModule()，这样做了之后，解释器就可以访问我们的模块函数了。与我们的Extest模块对应的initExtest()过程如下所示：

```

void initExtest() {
    Py_InitModule("Extest", ExtestMethods);
}

```

现在，所有的打包工作都完成了。我们把这些代码和Extest1.c文件里的代码合并到一起组成一个我们称之为Extest2.c的新文件，这个例子的开发阶段就告一段落了。

建立扩展的另外一种做法是先用一些“伪”函数（也可以是一些测试函数或“哑”函数）把与打包函数有关的代码编写好，随着开发过程的推进，逐步替换为具有实际功能的代码实现。这个办法可以保证你Python语言和C语言之间的程序接口是正确的，然后要用Python来测试你的C语言代码。

20.2.3 编译

我们现在到达了编译阶段。为了把刚打好包的Python语言重新扩展建立起来，你需要把它和

Python库一起编译。各种计算机平台上的这项工作现在都已经标准化了，这使语言扩展方面的设计人员日子好过多了。

- 1) 拷贝Misc/Makefile.pre.in文件。
- 2) 建立Setup文件。
- 3) 建立Makefile文件。
- 4) 运行make命令，编译和链接你的代码。
- 5) 把你的模块导入到Python环境里去。
- 6) 对函数进行测试。

1. 拷贝Misc/Makefile.pre.in文件

第一步是把Makefile.pre.in文件从Python发行版本的Misc子目录拷贝到将在其中对你自己的扩展进行编译的本地目录里去。事实上，所有步骤都将在这个目录或文件夹里完成。

2. 建立Setup文件

接下来是建立一个Setup文件。它的第一行里必须有字符串“*shared*”。随后各行以模块名开始，然后是源代码文件名和编译器选项，编译器选项是建立该模块所必须的编译参数。如果你只有一个模块，那就只有一行。这些行的格式如下所示：

```
modName modFile[1, modFile2 ... ] [ compiler_opts ] [ linker_opts ]
```

就我们的Extest例子来说，它的Setup文件是由下面这两行组成的：

```
*shared*
Extest Extest2.c
```

Setup文件第一行里的“*shared*”字符串表示将要建立的是一个共享库（.so对象文件），比如Extest.so文件。这个文件可以被任何Python模块导入，就好像它完全都是用Python语言编写的一样。

3. 建立Makefile文件

现在需要建立Makefile文件。我们用下面的make命令完成这项工作：

```
% make -f Makefile.pre.in boot
```

这一步骤通常有非常多的输出，大多数都不太重要。这一步的作用简单点儿说就是根据Setup文件提供的资料，再加上它对所有Python文件都保存在什么地方的了解，生成一个Makefile文件。你将利用这个Makefile文件建立你的模块对象文件。

4. 运行make命令，代码的编译和链接

```
% make
gcc -fpic -O2 -m486 -fno-strength-reduce -I/usr/
include/python1.5 -I/usr/include/python1.5 -
DHAVE_CONFIG_H -c ./Extest2.c
gcc -shared Extest2.o -o Extestmodule.so
```

编程提示：Module.so与Modulemodule.so的对比

[CN]

如果你的模块仅由一个同名的文件组成，那你的共享对象文件也将使用同样的名称，只是文件扩展名会是“.so”。比如说，如果我们的模块名是Extest，文件名是Extest.c，那我

们的共享对象文件就将是Extest.so。如果你有不只一个文件，或者只有一个文件但文件名与模块名不相同，那你的模块就会在它的名字后面多出一个“module”后缀，比如说Extestmodule.so。不管是哪种情况，在导入这个模块时还是要使用它原始的名字（即不加上“module”后缀）。

5. 把模块导入到Python环境

现在可以在Python解释器里尝试导入我们的模块了，如下所示：

```
>>> import Extest
>>> Extest.fac(5)
120
>>> Extest.fac(9)
362880
>>> Extest.doppel('abcdefgh')
('abcdefgh', 'hgfedcba')
>>> Extest.doppel("Madam, I'm Adam.")
("Madam, I'm Adam.", ".madA m'I ,madaM")
```

6. 对函数进行测试

我们想做的最后一件事情是添加一个测试用函数。事实上，我们已经有一个了，就是我们前面内容里曾经介绍过的主函数main()。但我们代码里的这个main()函数现在已经有潜在的危险性了，因为系统里应该只有一个main()函数。消除这个危险的办法并不难：先把我们的main()函数改名为test()函数并对它进行打包，然后再增加一个Extest_test()函数并对ExtestMethods数组进行修改，从而得到如下所示的代码：

```
static PyObject *
Extest_test(PyObject *self, PyObject *args) {
    test();
    return (PyObject*)Py_BuildValue("");
}

static PyMethodDef
ExtestMethods[] = {
    { "fac", Extest_fac, METH_VARARGS },
    { "doppel", Extest_doppel, METH_VARARGS },
    { "test", Extest_test, METH_VARARGS },
    { NULL, NULL },
};
```

Extest_test()模块函数运行test()并返回一个空字符串，导致一个Python语言中的None值被返回给调用者。

现在就可以从Python环境里运行同样的测试了：

```
>>> Extest.test()
4! == 24
8! == 40320
12! == 479001600
reversing 'abcdef', we get 'fedcba'
reversing 'madam', we get 'madam'
>>>
```

下面给出的是最终版本的Extest2.c文件（程序示例20-2），我们就是用它生成刚才看到的输出内容的。

程序示例20-2 C扩展库的Python打包版本 (Extest2.c)

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int fac(int n)
6  {
7      if (n < 2) return(1);
8      return ((n)*fac(n-1));
9  }
10
11 char *reverse(char *s)
12 {
13     register char t,
14                 *p = s,
15                 *q = (s + (strlen(s) - 1));
16
17     while (s && (p < q))
18     {
19         t = *p;
20         *p++ = *q;
21         *q-- = t;
22     }
23     return(s);
24 }
25
26 void test()
27 {
28     char s[BUFSIZ];
29     printf("4! == %d\n", fac(4));
30     printf("8! == %d\n", fac(8));
31     printf("12! == %d\n", fac(12));
32     strcpy(s, "abcdef");
33     printf("reversing 'abcdef', we get '%s'\n", \
34           reverse(s));
35     strcpy(s, "madam");
36     printf("reversing 'madam', we get '%s'\n", \
37           reverse(s));
38 }
39
40 #include "Python.h"
41
42 static PyObject *
43 Extest_fac(PyObject *self, PyObject *args)
44 {
45     int num;
46     if (!PyArg_ParseTuple(args, "i", &num))
47         return NULL;
48     return (PyObject*)Py_BuildValue("i", fac(num));
49 }
50
51 static PyObject *
52 Extest_doppel(PyObject *self, PyObject *args)
53 {
54     char *orig_str;
55     char *dupe_str;
56     PyObject* retval;
57
58     if (!PyArg_ParseTuple(args, "s", &orig_str))
59         return NULL;
60     retval = (PyObject*)Py_BuildValue("ss", orig_str, \
61         dupe_str=reverse(strdup(orig_str)));
62     free(dupe_str);
63     return retval;
64 }
65
66 static PyObject *
67 Extest_test(PyObject *self, PyObject *args)

```

```

68 {
69     test();
70     return (PyObject*)Py_BuildValue("");
71 }
72
73 static PyMethodDef
74 ExtestMethods[] =
75 {
76     { "fac", Extest_fac, METH_VARARGS },
77     { "doppel", Extest_doppel, METH_VARARGS },
78     { "test", Extest_test, METH_VARARGS },
79     { NULL, NULL },
80 };
81
82 void initExtest()
83 {
84     Py_InitModule("Extest", ExtestMethods);
85 }

```

在这个例子里，我们选择了以Python代码书写方式对C语言代码进行分段。这样做的目的只是为了让程序更容易阅读，并不代表我们前面那个比较短的程序示例有什么问题。在现实生活里，这些源代码文件通常都很长，并且经常选择把它们的打包器完全放在另外一个文件里，比如ExtestWrappers.c或其他类似的文件里。

20.2.4 引用的计数

读者可以还记得Python语言使用引用计数做为跟踪记录对象的办法，引用为零的对象其已经分配的内存将被收回，这也是废弃物回收机制的组成部分。在建立扩展的时候，对Python对象的处理手段必须特别留心，因为你必须注意是否需要修改这类对象的引用。

一个对象可以有两种类型的引用。其中之一是所谓的“拥有”引用，即该对象的这个引用是用递增一个“1”的办法来表明你的拥有权的。如果某个Python对象是你亲自创建的，那你肯定会有一个“拥有”引用。

当你用完某个Python对象时，你必须抛弃你的拥有权，办法有很多：可以通过递减它的引用计数，可以通过参数传递转移你的拥有权，还可以把这个对象保存起来。如果没有抛弃“拥有”引用会导致出现一次内存流失现象。

你还可以拥有某个对象的“借用”引用，此时你的责任不象刚才那么重。如果某个对象的引用是别人传递给你的，可你既不对它进行任何形式的处理，也不关心它的引用计数，那这个引用就是你“借用”的；如果它的引用被减为零，你也就不能够再引用它了。如果想把你的“借用”引用转换为一个“拥有”引用，只需简单地递增该对象的引用计数就行。

Python语言提供了两个能够对某个Python对象的引用进行修改的C语言宏命令。它们被列在表20-3里面。

表20-3 修改Python对象引用计数的宏命令

函 数	说 明
Py_INCREF(obj)	递增对象obj的引用计数
Py_DECREF(obj)	递减对象obj的引用计数

在前面的Extest_test()函数里, 我们通过用空字符串创建PyObject的办法来返回None; 我们还可以通过变成None对象PyNone的“拥有者”来达到这一目的, 即增加你对它的引用计数, 然后明确地返回它, 就像如下所示的代码这样:

```
static PyObject *
Extest_test(PyObject *self, PyObject *args) {
    test();
    Py_INCREF(Py_None);
    return PyNone;
}
```

Py_INCREF()和Py_DECREF()函数还有专门对NULL对象进行检查的版本, 它们分别是Py_XINCRF()和Py_XDECREF()。

我们强烈建议读者查阅有关的Python文档以进一步了解扩展和嵌入Python语言方面与引用有关的各种细节(请参考附录A中的参考文档清单)。

20.2.5 线程化和GIL方面的考虑

语言扩展的开发人员必须想到他们的代码可能会在一个多线程Python环境里执行。在17.3.1节里, 我们介绍了Python虚拟机(Python Virtual Machine, PVM)和全局解释器线程锁(Global Interpreter Lock, GIL)等概念, 同时还说明了为什么在任一给定时刻在PVM里只能运行有一个执行线程的原因, 而GIL将负责禁止其他线程的运行。此外, 我们还指出调用外部函数(比如扩展代码中的函数)的代码在该调用返回之前会使GIL保持在加锁状态。

我们也提示到还有一个补救措施, 也就是有一个让扩展程序释放GIL的办法, 在开始执行一个系统调用之前可以采用这个办法让线程释放GIL。具体的做法是: 在线程能够(或者不能够)安全运行的地方把两个C语言宏命令分别加到该部分的首尾, 就好像用一对括号把这段代码“括起来”, 这两个C语言宏命令分别是Py_BEGIN_ALLOW_THREADS和Py_END_ALLOW_THREADS。这两个宏命令之间的代码将允许其他线程运行。

类似于引用计数宏命令的情况, 我们建议大家查阅扩展和嵌入Python语言方面的文档以及Python/C API参考手册。

20.3 相关论题

1. SWIG

有一个名为SWIG的外部工具软件, 它的意思是“Simplified Wrapper and Interface Generator”(简化打包器和接口生成器)。它是由David Beazley编写的, 他也是《Python Essential Reference》(Python基本参考手册)的作者。SWIG这个软件工具能够自动地使用相应的C/C++头文件生成经过打包的代码, 为编译Python、Tcl和Perl的工作做好准备。事实上, SWIG既简单又好用, 一旦你习惯了使用它, 就几乎可以完全绕过我们在这章里讨论过的所有步骤了! 在SWIG的主Web站点上可以找到更多关于它的资料, 它的Web站点的Web地址(URL)如下所示:

<http://www.swig.org>

2. 嵌入

嵌入是Python语言里的另一特色。它正好与扩展的概念相反：它不是把C语言代码打包到Python语言里去，它指的是对Python解释器打包后再把它放入一个C语言应用程序中去。对某些规模庞大、要求苛刻的关键性代码和/或重要任务代码来说，这样做的效果是可以让它们拥有一个嵌入式Python解释器的强大功能。一旦你拥有了Python，就是一个崭新的天地了。

做为结论，我们想告诉大家与本章内容有关的东西涉及到两个正式的Python文档，它们是《Embedding and Extending the Python Interpreter》（嵌入和扩展Python解释器）和《Python/C API Reference Manual》（Python/C语言API参考手册）。这两份文档可以补充我们在本章没有讲细的地方，它们都可以在Python的主页或直接从下面的链接处找到：

<http://www.python.org/doc/ext>

20.4 练习

20-1 扩展Python语言。Python语言的扩展都有哪些优点？

20-2 扩展Python语言。你能说出一些使用扩展的缺点或危险吗？

20-3 编写扩展。设法找到一个C/C++编译器并用它编写一个小程序，让自己熟悉使用C/C++语言进行的程序设计。找到你的Python发行版本目录，找出Misc/Makefile.pre.in文件存放的位置。在Python解释器里对你刚才编写的C语言程序进行打包。完成建立一个共享对象所必须的各个步骤。在Python解释器里访问这个模块并对它进行测试。

20-4 从Python到C的移植。把你在前面章节里做过的几个练习题以扩展模块的形式移植到C/C++语言中去。

20-5 打包C语言代码。找出一段你也许是很久以前编写的一段C/C++代码，把它移植到Python语言中去。要求不采用导入的方式，请把它转换为一个扩展模块。

20-6 编写扩展。在练习13-3里，你编写出一个dollarize()函数，它是一个把浮点数值转换为金额数值字符串的类（class）的组成部分，在它返回的金额字符串里还加上了美元符号和逗号。请编写一个扩展，要求对dollarize()函数进行打包，同时在模块里要集成有一个内部的测试性函数，即test()函数。

20-7 扩展和嵌入的对比。扩展和嵌入之间有什么区别？

第三部分 附 录

附录A 部分练习答案

第2章

5. 循环和数字

a)

```
i = 0
while i < 11:
    i = i + 1
```

b)

```
for i in range(11):
    pass
```

6. 条件语句

```
n = int(raw_input('enter a number: '))
if n < 0:
    print 'negative'
elif n > 0:
    print 'positive'
else:
    print 'zero'
```

7 s = raw_input('enter a string: ')

```
for eachChar in s:
    print eachChar
```

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

8.

```
subtot = 0
for i in range(5):
    subtot = subtot + int(raw_input('enter a number: '))
print subtot
```

第3章

7. 标识符

40XL	number
\$saving\$	symbol
print	kw
0x40L	number
big-daddy	symbol
2hot2touch	number
thisIsn'tAVar	symbol
if	kw
counter-1	symbol

第4章

6. `type(a) == type(b)` 和 `type(a) is type(b)` 之间的区别是:

`type(a) == type(b)` 判定 `type(a)` 的值和 `type(b)` 的值是否相等。== 是一个值比较。

`type(a) is type(b)` 判定 `type(a)` 和 `type(b)` 返回的对象是否相同。

第5章

8. `import math`

```
def sqcube():
    s = float(raw_input('enter length of one side: '))
    print 'the area is:', s ** 2., '(units squared)'
    print 'the volume is:', s ** 3., '(cubic units)'

def cirsph():
    r = float(raw_input('enter length of radius: '))
    print 'the area is:', math.pi * (r ** 2.),
    '(units squared)'
    print 'the volume is:', (4. / 3.) * math.pi * (r **
3.), '(cubic units)'
```

```
sqcube()
cirsph()
```

11.

a.

```
for i in range(0, 22, 2):    # range(0, 21, 2) okay too
    print i
```

OR

```
for i in range(22):        # range(21) okay too
```

```
if i % 2 == 0: print i
```

b.

```
for i in range(1, 20, 2):      # range(1, 21, 2) okay too
    print i
```

OR

```
for i in range(20):          # range(21) okay too
    if i % 2 != 0: print i
```

c. 当 $i \% 2$ 是0时，它就是偶数（可以被2整除）；否则它就是奇数。

第6章

1. find(), rfind(), index(), rindex()

2.

```
import string

alphas = string.letters + '_'
alnums = alphas + string.digits

iden = raw_input('Identifier to check? ')

if len(iden) > 0:
    if iden[0] not in alphas:
        print "invalid: first char must be alphabetic"
    else:
        if len(iden) > 1:
            for eachChar in iden[1:]:
                if eachChar not in alnums:
                    print "invalid: other chars must be alphanumeric"
                    break
            else:
                import keyword
                if iden not in keyword.kwlist:
                    print 'ok'
                else:
                    print 'invalid: keyword name'
        else:
            print 'no identifier entered'
```

第7章

4.

```
# assumes both list1 and list2 are of the same length
dict = {}
for i in range(len(list1)):
    dict[list1[i]] = list2[i]
```


如果使用map()内建函数, 解决方案会更精巧。

7.

```
list1 = oldDict.values()
list2 = oldDict.keys()
```

(本答案的其余部分请参考练习4的答案。)

第8章

3a.

```
range(10)
```

4.

```
def isprime(num):
    count = num / 2
    while count > 1:
        if num % count == 0: return 0
        count = count - 1
    return 1
```

第9章

2.

```
file = open(raw_input('enter file: '))
allLines = file.readlines()
file.close()
num = input('enter number of lines: ')
i = 0
while i < num:
    print allLines[i],
    i = i + 1
```

(HINT: 1., 2., 3.可以由outfile.py on p.38和p.245激发)

14b.

```
import sys

print "# of args", len(sys.argv)
print "args:", sys.argv
```

18.

一部分来自于creatext.py on

第10章

1. e)

2. **try-except**在**try**子句里对例外进行监控，如果出现例外，执行就跳到对应的**except**子句去。但**try-finally**中的**finally**子句不管例外是否发生都会被执行。

第11章

5.

```
def printf(string, *args):
    print string % args
```

第13章

2. 方法也是函数，但它们是跟特定的类对象类型捆绑在一起的。它们的定义是类定义的一部分，它们在执行时是那个类的实例的一部分。

15. 用**open()**还是用**capOpen()**来读我们的文件没有什么差别，因为我们在**capOpen.py**里把所有的读功能都用Python系统的缺省函数代表了，也就是说，对读操作不需要采取特殊的操作，代码可以像原来一样执行。**read()**、**readline()**或**readlines()**都没有被特殊功能覆盖。

第14章

1. 函数、方法、类、可调用类的实例。

3. **raw_input()**以字符串形式返回用户的输入；**input()**先对用户输入进行求值，返回的是一个Python对象。

第15章

1.

bat, hat, bit, etc.

[bh] [aiu] t

2. 名 姓

[A-Za-z-]+ [A-Za-z-]+

(任何两个用一个空格分隔的单词，比如名和姓，允许使用连字符)

3. 姓，名

[A-Za-z-]+, [A-Za-z]

(任何两个用一个空格和逗号分隔的单词或单个字母，比如姓和名字的首字母，允许使用连字符)

[A-Za-z-]+, [A-Za-z-]+

任何两个用逗号和空格分隔的单词，比如名和姓，允许使用连字符。

8. Python长整数

```
\d+[1L]
```

(只适用于十进制整数)

9. Python浮点数

```
[0-9]+(\.[0-9]*)?
```

(描述的是一个简单的浮点数，即由任意个数的数字组成的数，后面跟着一个可选的小数点；小数点后面又有零个或者多个数字。比如“0.004”、“2”、“75.”等)

第16章

3.

TCP

5.

```
>>> import socket
>>> socket.getservbyname('daytime', 'udp')
13
```

附录B 参 考 信 息

阅读参考书

1. Altom, Tim, with Mitch Chapman, *Programming with Python*, Prima, 0-7615-2334-0
2. Beazley, David M., *Python Essential Reference*, New Riders, 0-7357-0901-7
3. Brown, Martin C., *Python Annotated Archives*, McGraw Hill, 0-07-212104-1
4. Grayson, John E., *Python and Tkinter Programming*, Manning, 1-884777-81-3
5. Hammond, Mark and Andy Robinson, *Python Programming on Win32*, O'Reilly, 1-56592-621-8
6. Harms, Daryl, and Kenneth McDonald, *The Quick Python Book*, Manning, 1-884777-74-0
7. Himstedt, Tobias, and Klaus Mänzel, *Mit Python programmieren (Programming with Python)* [in German], dpunkt.verlag, 3-920993-85-3
8. Lundh, Fredrik, (the eff-bot guide to) *The Standard Python Library*, Fat-Brain.com, (Product#) EB00002582
9. Lutz, Mark, and David Ascher, *Learning Python*, O'Reilly, 1-56592-464-9
10. Lutz, Mark, *Programming Python*, O'Reilly, 1-56592-197-6
11. Lutz, Mark, *Python Pocket Reference*, O'Reilly, 1-56592-500-9
12. McGrath, Sean, *XML Processing with Python*, Prentice Hall, 0-13-021119-2
13. Van Laningham, Ivan, *Teach Yourself Python in 24 Hours*, Sams, 0-672-31735-4
14. Watters, Aaron, Guido van Rossum, and James C. Ahlstrom, *Internet Programming with Python*, Henry Holt & Co./M&T Books/MIS:Press/IDG Books, 1-55851-484- [out-of-print]
15. van Rossum, Guido, *Python Library Reference: Release 1.5.2*, iUniverse.com, 1-58348-373-x
16. van Rossum, Guido, *Python Reference Manual: Release 1.5.2*, iUniverse.com, 1-58348-374-8
17. van Rossum, Guido, *Python Tutorial: Release 1.5.2*, iUniverse.com, 1-58348-375-6

18. von Löwis, Martin, and Nils Fischbeck, *Das Python-Buch (The Python Book)* [in German], Addison Wesley Longman, 3-8273-1110-1 [out-of-print]

其他印刷品

1. Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley Longman, 0-201-10088-6
2. Brookshear, J. Glenn, *Computer Science, An Overview*, 6th Ed., Addison Wesley Longman, 0-201-35747-X
3. Eckel, Bruce, *Thinking in C++*, 2nd Ed., Prentice Hall, 0-13-979809-9
4. Eckel, Bruce, *Thinking in Java*, Prentice Hall, 0-13-027363-5
5. Friedl, Jeffrey, *Mastering Regular Expressions*, O'Reilly, 1-56592-257-3
6. Galvin, Peter, and Abraham Silberschatz, *Operating System Concepts*, 5th Ed., Addison Wesley Longman, 0-471-36414-2
7. McKusick, Marshall Kirk, Keith Bostic, Michael J. Karels, and John S. Quarterman, *Design and Implementation of the 4.4BSD Operating System*, Addison Wesley Longman, 0-201-54979-4
8. Ousterhout, John, *Tcl and Tk Toolkit*, Addison Wesley Longman, 0-201-63337-X
9. Tanenbaum, Andrew S., *Operating Systems: Design and Implementation*, Prentice Hall, 0-13-638677-6
10. Welch, Brent, *Practical Programming in Tcl and Tk*, 3rd Ed., Prentice Hall, 0-13-022028-0

在线参考

下面的清单列出了许多与Python语言有关的在线参考。最新清单请查阅书后所附的CD-ROM光盘上的Python热门站点表。更好的办法是直接访问Core Python Programming (Core Python程序设计) 网站:

<http://starship.python.net/crew/wesc/cpp>

Apache服务器模块

1. mod_python (前身是httpdapy和Insapy)

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

代码

1. Python的源代码 (SourceForge公司站点)
http://sourceforge.net/project/?group_id=5470
2. Snippets (SourceForge公司站点)
<http://sourceforge.net/snippet/browse.php?by=lang&lang=6>
3. SourceForge公司的Python项目站点
http://sourceforge.net/search/?type_of_search=soft&words=python
4. Vaults of Parnassus (Python共享软件)
<http://www.vex.net/parnassus/>

商业公司

1. **ActiveState Tool**
<http://www.activestate.com/>
2. **CyberWeb Consulting**
<http://www.roadkill.com/~wesc/cyberweb>
3. **O'Reilly Python DevCenter**
<http://www.oreillynet.com/python/>
4. **PythonLabs** (commercial home page)
<http://www.pythonlabs.com/>
5. **PythonWare**
<http://www.pythonware.com>
6. **ReportLab**
<http://www.reportlab.com/>
7. **UC Santa Cruz Extension Scripting Language Courses**
<http://www.ucsc-extension.edu/to/software/silang.html>

通信

1. 旧金山港区Python兴趣小组
<http://www.baypiggies.org/>
2. comp.lang.python新闻组周刊
<http://purl.org/thecliff/python/url.html>
3. Python协会
<http://www.python.org/workshops/>
4. Python链接 (热门站点链接, 名单长度随时变化)
http://www.cetus-links.de/oo_python.html

5. Python邮件表

<http://www.python.org/mailman/listinfo>

6. Python特殊兴趣小组

<http://www.python.org/sigs>

7. Starship的常见问题答疑FAQ

<http://starship.python.net/~tbryan/FAQ/Starship/>

8. Starship的Python站点

<http://starship.python.net>

核心站点

1. JPython

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

2. Python.org (社区主页)

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

数据库

1. 数据库

<http://www.python.org/topics/database/>

2. Python的数据库模块

<http://www.python.org/topics/database/modules.html>

3. Python的数据库应用程序接口DB API 2.0技术文件

<http://www.python.org/topics/database/DatabaseAPI-2.0.html>

扩展

1. 扩展和嵌入操作方面的参考资料

<http://www.python.org/doc/current/ext/ext.html>

2. Python的C语言应用程序接口

<http://www.python.org/doc/current/api/api.html>

3. SWIG (Simple Wrapper and Interface Generator, 简单打包器和操作界面生成器)

<http://www.swig.org>

图形化用户操作界面 (带有Python开发接口)

1. Gimp-Python

<http://www.daa.com.au/~james/pygimp>

2. Glade
<http://glade.pn.org>
3. GLC (Glade的Python代码生成器)
<http://glc.sourceforge.net>
4. GTK+ (GIMP工具箱)
<http://www.gtk.org>
5. KDE (K Desktop Environment, K桌面环境)
<http://www.kde.org>
6. PMW (帮助使用Python开发Tkinter应用程序Python MegaWidgets站点)
<http://www.dscpl.com.au/pmw/>
7. PyG工具 (PyGTK、PyGNOME等)
<http://www.bioinformatics.org/pygtools>
8. PyGTK模块
<http://www.daa.com.au/~james/pygtk>
9. PyQt-PyKDE
<http://www.thekompany.com/projects/pykde>
10. Python-KDE教程
<http://www.xs4all.nl/~bsaremp/python/tutorial.html>
11. TKinter (Python-Tk)
<http://www.python.org/topics/tkinter>
12. TKinter入门 (F. Lunch的个人站点)
<http://www.pythonware.com/library/tkinter/introduction>
13. Trolltech公司的Qt产品 (商业站点)
<http://www.trolltech.com/products>
14. wxPython
<http://www.wxpython.org/>

Macintosh

1. Macintosh库模块
<http://www.python.org/doc/current/mac/mac.html>
2. MacPython
<http://www.cwi.nl/~jack/macpython.html>
3. MacPython下载主页
http://www.python.org/download/download_mac.html

4. 打开目录的MacPython链接

<http://dmoz.org/computers/systems/macintosh/development/languages/python>

新闻

1. Python活动

<http://www.python.org/Events.html>

2. Python邮件表

<http://www.python.org/mailman/listinfo>

3. Python新闻

<http://www.python.org/News.html>

数学/科学计算

1. NumPy数学扩展

<http://www.python.org/topics/scicomp/numpy.html>

2. NumPy源代码 (SourceForge公司站点)

http://sourceforge.net/project/?group_id=1369

程序设计

1. Python与其他语言的比较

<http://www.python.org/doc/Comparisons.html>

2. 都来设计程序 (Computer Programming for Everybody, CP4E)

<http://www.python.org/cp4e/>

3. CP4E建议报告

<http://www.python.org/doc/essays/cp4e.html>

4. Empirical语言比较报告

<http://wwwipd.ira.uka.de/~prechelt/Biblio/jccpprtTR.pdf>

5. Guido的CP4E谈话 (Guido是Python的创始人)

<http://www.python.org/doc/essays/ppt/acm-cp4e/>

6. 说干就干: 学习Python编程

<http://www.idi.ntnu.no/~mlh/python/programming.html>

7. Python快速教程

<http://www.idi.ntnu.no/~mlh/python/instant.html>

8. 学习程序设计

http://members.xoom.com/alan_gauld/tutor/tutindex.htm

9. 普通用户教程

<http://www.honors.montana.edu/~jjc/easytut/easytut/>

参考资料

1. Python文档

<http://www.python.org/doc>

2. FAQ (常见问题答疑)

<http://www.python.org/doc/FAQ.html>

3. Python知识库FAQTS

<http://python.faqs.com>

4. Python标准库的全球性模块索引

<http://www.python.org/doc/current/modindex.html>

5. Python语言大全

<http://www.python.org/doc/current/ref/ref.html>

6. Python库大全

<http://www.python.org/doc/current/lib/lib.html>

7. Python-Perl用法指南

<http://starship.python.net/crew/da/jak/cookbook.html>

8. 快速参考指南

http://starship.python.net/quick-ref1_52.html

9. 规则表达式的HOWTO文档

<http://www.python.org/doc/howto/regex/regex.html>

发行版本

1. Python 1.5到1.5.2版本

<http://www.python.org/1.5>

2. Python 1.6版本

<http://www.python.org/1.6>

3. Python 2.0版本

<http://www.pythonlabs.com/products/python2.0>

4. Python下载

<http://www.python.org/download>

5. Python的FTP站点

<ftp://ftp.python.org>

6. 2.0版本的新增功能

<http://starship.python.net/crew/amk/python/writing/new-python>

Unicode

1. Python的Unicode集成 (M. A. Lemburg的个人站点)

<http://starship.python.net/crew/lemburg/unicode-proposal.txt>

2. Python的Unicode教程

http://www.reportlab.com/i18n/python_unicode_tutorial.html

3. 关于Unicode标准的主页

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

Web

1. 五分钟学会Python的CGI编程 (D. Mertz的个人站点, 网络版)

<http://webreview.com/pub/2000/07/07/feature/index02.html>

2. HTMLgen的主页

<http://starship.python.net/crew/friedrich/HTMLgen/html/>

3. Web程序设计

<http://www.python.org/topics/web/>

4. 用Python编写CGI程序

http://www.devshed.com/Server_Side/Python/CGI/print.html

XML

1. XML 1.0技术报告注释版

<http://www.xml.com/axml/axmlintro.html>

2. Python-XML的How-to文档

<http://www.python.org/doc/howto/xml/>

3. Python-XML大全

<http://www.python.org/doc/howto/xml-ref/>

4. XML

<http://www.python.org/topics/xml/>

5. XML索引

<http://www.oasis-open.org/cover/>

6. XML的FAQ

<http://www.ucc.ie/xml/>

7. Zope

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

附录C Python操作符汇总

表C-1列出了所有的Python操作符和它们能够处理的标准类型。操作符按优先级从高到低的顺序排列，优先级相同的归入一个组。

表C-1 Python操作符（⊥表示是单元操作符）

操作符	整数	长度	浮点数	复数	字符串	列表	表列	字典
[]					•	•	•	
[:]					•	•	•	
**	•	•	•	•				
+†	•	•	•	•				
-†	•	•	•	•				
~†	•	•						
*	•	•	•	•	•	•	•	
/	•	•	•	•				
%	•	•	•	•	•			
+	•	•	•	•	•	•	•	
-	•	•	•	•				
<<	•	•						
>>	•	•						
&	•	•						
^	•	•						
	•	•						
<	•	•	•	•	•	•	•	•
>	•	•	•	•	•	•	•	•
<=	•	•	•	•	•	•	•	•

(续)

操作符	整数	长度	浮点数	复数	字符串	列表	表列	字典
>=	•	•	•	•	•	•	•	•
=	•	•	•	•	•	•	•	•
!=	•	•	•	•	•	•	•	•
<>	•	•	•	•	•	•	•	•
is	•	•	•	•	•	•	•	•
is not	•	•	•	•	•	•	•	•
in					•	•	•	
not in					•	•	•	
not†	•	•	•	•	•	•	•	•
and	•	•	•	•	•	•	•	•
or	•	•	•	•	•	•	•	•

附录D Python版本2.0的新增功能

D.1 简介

在本书的编写过程中，Python的开发团队也在为Python 2.0努力工作着，到本书开始印刷的时候，它也和人家见面了。

书后所附的CD-ROM光盘里有三个最新的Python发行版本：1.5.2、1.6和2.0，还包括Python解释器最新的Java版本，JPython 1.1（也叫做Jython）。

做为一个稳定的版本，版本1.5.2已经面世几乎两年了，它是本书大部分内容的基础。版本1.6给Python带来了许多新的变化。增加了字符串方法、Unicode支持，并改进了规则表达式引擎。

一些更显著的修改出现在发行版本2.0中，这也是我们要在这里讨论的内容。我们同时推荐大家阅读Python版本2.0 Web站点上的“Python版本2.0的新增功能”文档（它的URL请参考附录B的在线资源一节）。

D.2 复习和预习

下面的内容是对从Python版本1.5.2到版本1.6改进的复习，另外还有一些漏洞修补和模块更新（新增、修订和弃用模块）。

- Unicode支持
- 字符串方法
- 升级了的规则表达式引擎（性能和Unicode增强版）
- 新的函数调用机制

版本2.0也有常见的漏洞修补和模块升级，此外还给这种语言增加了以下新特色：

- 增量赋值
- 智能列表
- 功能扩展了的**import**语句
- 功能扩展了的**print**语句

当你在自己的计算机上编译并安装了Python版本2.0之后，会在UNIX下看到熟悉的启动信息（如果你使用的是其他计算机平台，应该也是熟悉的信息）：

```
% python
Python 2.0 (#4, Oct  2 2000, 23:58:52)
[GCC 2.95.1 19990816 (release)] on sunos5
Type "copyright", "credits" or "license" for more
information.
>>>
```

现在来看看它的新功能吧!

D.3 增量赋值

增量赋值说的是操作符的用法,即同时完成算术运算和赋值两个操作。如果你是一名C、C++或Java程序员,就会知道下面的符号是做什么用的:

```
+=      -=      *=      /=      %=      **=
<<=     >>=     &=      ^=      |=
```

比如说,下面这个比较短的代码

```
A += B
```

就表示:

```
A = A + B
```

除了在语法方面的明显变化外,最显著的差别是第一个对象(我们例子中的A)只求值了一次。可变对象会马上被修改;而不可变对象还是“A = A + B”的老样子(新分配出一个对象),但对A的求值操作还是只进行一次,就象我们刚才说的那样。请看下面的例子:

```
>>> m = 12
>>> m %= 7
>>> m
5
>>> m **= 2
>>> m
25
>>> aList = [123, 'xyz']
>>> aList += [45.6e7]
>>> aList
[123, 'xyz', 456000000.0]
```

在创建仿真数值类型的类时,这些即时操作符都有与之对应的特殊方法。在实现即时特殊方法的时候,只要非即时操作符的加上一个“i”就可以了。比如说,与+操作符对应的是__add__(),而与+=操作符对应的就是__iadd__()了。

D.4 智能列表

我们曾经把lambda函数和map()及filter()一起用来对列表成员进行操作,也曾经用这个办法根据一个条件表达式的规则过滤掉列表成员,这些你还记得吧?智能列表简化了这项各种并改进了代码的执行性能,有了它,就不必非得使用lambda表达式,也不必非得对内建函数进行函数化的程序设计了。智能列表允许你直接遍历原来的列表序列进行操作。

我们先来看看简单些的智能列表语法:

```
[ expression for iterative_var in sequence ]
```

这个语句的核心是那个for循环,它将遍历序列sequence中的每一个数据项。最前面的表达式expression将作用于序列中的每一个成员并用其结果组成该表达式将要产生的列表。遍历变量不再需要是表达式的一部分了。

请看在书中（第11章，函数）出现过的下列代码，它是用一个lambda函数来求一个序列中各个成员的平方：

```
>>> map(lambda x : x ** 2, range(6))
[0, 1, 4, 9, 16, 25]
```

我们可以把这代码替换为下面的智能列表语句：

```
>>> [ x ** 2 for x in range(6) ]
[0, 1, 4, 9, 16, 25]
```

新语句里只需要一个函数调用（range()）而不是三个（range()、map()和那个lambda函数）。在表达式两头还可以加上括号让它看上去更清晰，即“[(x ** 2) for x in range(6)]”。智能列表的这个语法可以用于同时使用map()内建函数和lambda的情况代替原来的做法，而且更有效率。

智能列表还支持带if语句的扩展语法：

```
[ expression for iterative_var in sequence if cond_expression ]
```

这个语法会在遍历过程中根据cond_expression条件表达式给出的条件过滤或者“捕获”序列中符合该条件的成员。

请看下面的odd()函数，它的作用是确定一个数值参数是偶数还是奇数（如果是奇数返回1，是偶数返回0）：

```
def odd(n) :
    return n % 2
```

我们可以用filter()和lambda来实现这个函数的核心操作——找出序列里的奇数。如下所示：

```
>>> seq = [11, 10, 9, 9, 10, 10, 9, 8, 23, 9, 7, 18, 12, 11, 12]
>>> filter(lambda x: x % 2, seq)
[11, 9, 9, 9, 23, 9, 7, 11]
```

类似于前一个例子，我们可以用智能列表替代filter()和lambda来获取想要的数字集合，如下所示：

```
>>> [ x for x in seq if x % 2 ]
[11, 9, 9, 9, 23, 9, 7, 11]
```

智能列表还支持多重嵌套的for循环和多个if从句。进一步的资料请查阅包括“新增功能”在线文档在内的其他文档。

D.5 功能扩展了的import语句

Python程序员们另一个经常性的要求是希望在把模块和模块属性导入到自己的程序里去的时候能够使用简短一点的名字而不是它们原始给定的名字。一个常用的折中办法是把模块名赋值给一个变量，如下所示：

```
>>> import longmodulename
>>> short = longmodulename
>>> del longmodulename
```

在上面的例子里，程序员可以使用short.attribute代替longmodulename.attribute来访问同一个

对象。(另外一个情况是使用from-import导入模块属性的时候,请往下看。)但反复多次地这样做,模块一多,就会很乱,看上去也很浪费。新扩展的import语句现在支持下面这样的用法:

```
>>> import longmodule.name as short
```

同样地,你也可以把这个语法用在from-import语句里:

```
>>> from sys import stderr as err
>>> err.write("now using sys.stderr")
now using sys.stderr
```

我们注意到“as”并不是一个关键字,它只有在和import一起使用的时候才会被识别为特殊用途。其结果是,你还是可以在代码里把它用做一个合法的标识符:

```
>>> as = 14
>>> as += 3
>>> as
17
```

D.6 功能扩展了的print语句

Python版本2.0里面最后一个,也是最有争议的一个是功能扩展了的print语句。这个修改增加了两个大于号(>>),它允许你把print的输出导向一个不是标准输出的文件。

在下面的例子里,我们接着刚才把sys.stderr导入为err的情况继续往下说:

```
>>> print >> err, "using sys.stderr again"
using sys.stderr again
```

D.7 结论

虽然增量赋值和智能列表看上去好像是给Python语言原本简单的语法增加了一点复杂性,但并没有改变这种语言干净利索的特性。这些新功能带到桌子上的关键性增值实际被掩盖在桌布下面了。

增量赋值只对第一个对象求值一次——这在长途赛跑里可以节省时间并提高性能。同时,因为lambda所创建的函数对象和用def生成的函数对象在实际效果上完全一致,所以当它们执行的时候也会象一个真正的函数那样产生同样的负担。使用了智能列表之后,就不必额外再生成什么函数对象了,也不会因为额外的函数调用而产生额外的负担。从这个角度来说,智能列表让Python执行得更顺畅了。

功能扩展了的import和print语句与执行性能的关系不大,它们主要侧重于为程序员提供方便。

其他新增功能还包括一个可选用的废弃物回收器,它可以检测循环并改进了对XML的支持(xml.dom、xml.sax、xml.parsers和pyexpat模块等)。版本2.0值得一提的其他特色有范围显示(range display)、并行for循环以及向64位相同的移植等。

D.8 练习

D-1 请编写一个复合智能列表语句,(随机)创建一个由1到10个随机数构成的列表,每个数

都是1到100之间的随机数。取出其中的奇数。

答案:

我们的解决方案使用了智能列表和新扩展的import语法。

```
>>> from random import randint as ri
>>> [ y for y in [ ri(1, 100) for x in range(ri(1, 10)) ] if y % 2]
[47, 9, 85]
>>> [ y for y in [ ri(1, 100) for x in range(ri(1, 10)) ] if y % 2]
[45, 3]
>>> [ y for y in [ ri(1, 100) for x in range(ri(1, 10)) ] if y % 2]
[]
>>> [ y for y in [ ri(1, 100) for x in range(ri(1, 10)) ] if y % 2]
[47, 25, 95, 83, 15, 77]
```

都是1到100之间的随机数。取出其中的奇数。

答案:

我们的解决方案使用了智能列表和新扩展的import语法。

```
>>> from random import randint as ri
>>> [ y for y in [ ri(1, 100) for x in range(ri(1, 10)) ] if y % 2]
[47, 9, 85]
>>> [ y for y in [ ri(1, 100) for x in range(ri(1, 10)) ] if y % 2]
[45, 3]
>>> [ y for y in [ ri(1, 100) for x in range(ri(1, 10)) ] if y % 2]
[]
>>> [ y for y in [ ri(1, 100) for x in range(ri(1, 10)) ] if y % 2]
[47, 25, 95, 83, 15, 77]
```