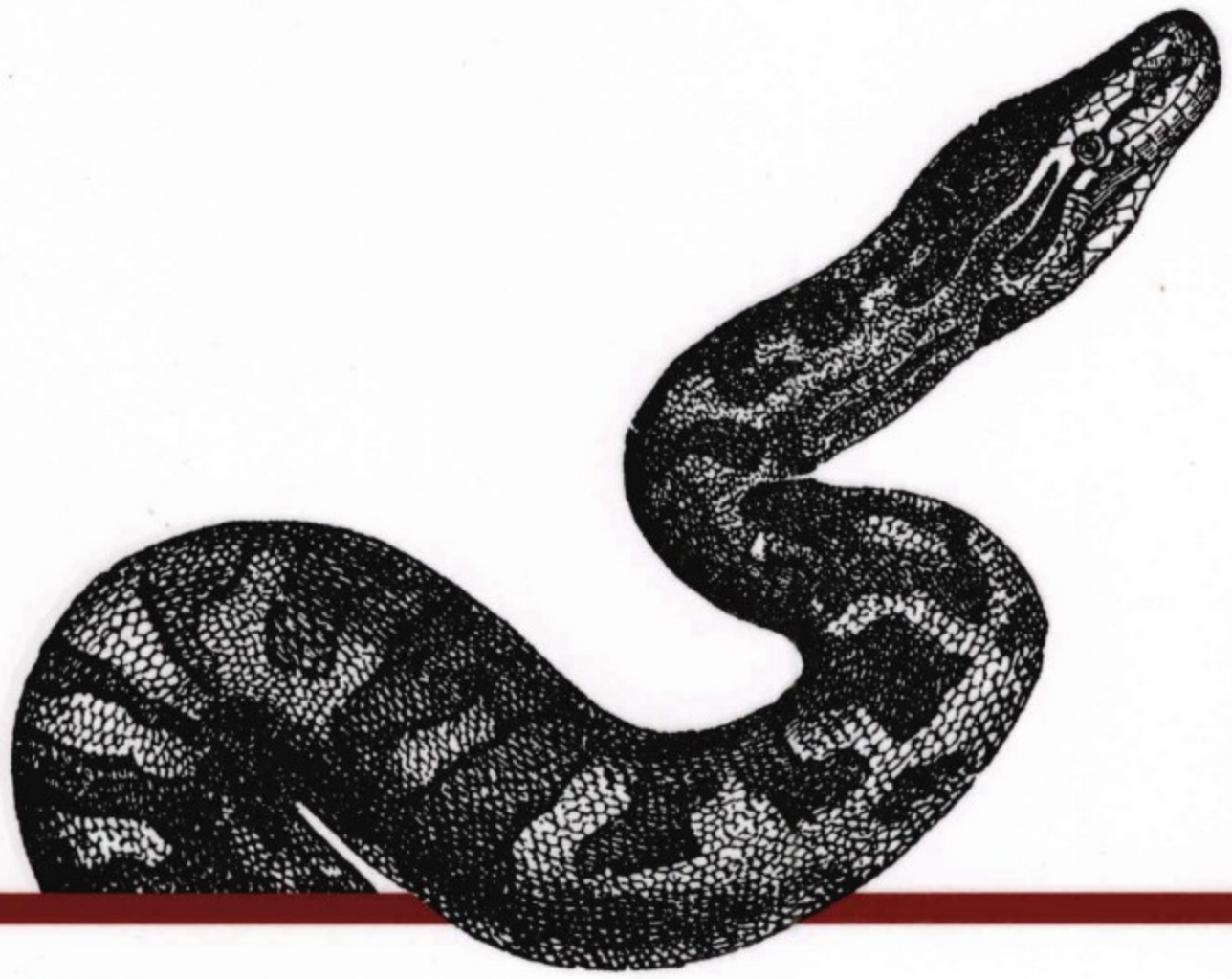
Python in a Nutshell



PYTHON

技术手册

（第2版）

[美] *Alex Martelli* 著

程胜 杨萍 译

O'REILLY®

人民邮电出版社
POSTS & TELECOM PRESS

O'REILLY®

PYTHON 技术手册（第 2 版）

[美] Alex Martelli 著

程胜 杨萍 译

人 民 邮 电 出 版 社

北 京

图书在版编目（CIP）数据

Python技术手册 : 第2版 / (美) 马特利(Martelli,A.) 著 ; 程胜, 杨萍译. -- 北京 : 人民邮电出版社, 2010.6
ISBN 978-7-115-22583-2

Ⅰ. ①P… Ⅱ. ①马… ②程… ③杨… Ⅲ. ①软件工具—程序设计—技术手册 Ⅳ. ①TP311.56-62

中国版本图书馆CIP数据核字(2010)第056707号

版权声明

Python 技术手册（第 2 版）

◆ 著 [美] Alex Martelli
译 程 胜 杨 萍
责任编辑 刘映欣

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷

◆ 开本：787×1000 1/16
印张：39.5
字数：829 千字 2010 年 6 月第 1 版
印数：1 – 3 500 册 2010 年 6 月北京第 1 次印刷

著作权合同登记号 图字：01-2009-1823 号
ISBN 978-7-115-22583-2

定价：89.00 元

读者服务热线：(010)67132705 印装质量热线：(010)67129223
反盗版热线：(010)67171154

内 容 提 要

本书是一本全面介绍有关 Python 语言和 Python 程序开发专业知识的参考手册。书中详细介绍了 Python 开发工具的安装和使用、Python 语言的语法结构、Python 内置对象、库和模块以及 Python 与其他语言的扩展和嵌入，并专门介绍了有关 Python 网络和 Web 编程的内容和实例。本书列举了 Python 对象和模块中提供的所有类型、方法和函数，并辅以适当的示例，系统地展示了 Python 包含的功能及其使用方法。

本书的内容面向 Python 语言的各层次用户，可以为不同层次的读者提供相应的信息。针对从其他编程语言转到使用 Python 语言的程序员，本书还重点介绍了 Python 的跨平台功能。

O'Reilly Media, Inc.介绍

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

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

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

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

前言

Python 编程语言可以很好地协调一些看起来似乎很明显的矛盾：Python 编程语言格式优雅并注重实效、简单而且功能强大、非常高层但是并不妨碍用户对底层的比特（bit）和字节（Byte）的处理，Python 编程语言适合于编程新手，对 Python 专家也非常适用。

本书的目标读者是已经对 Python 有一些了解的程序员，以及刚开始从其他编程语言转到使用 Python 的有经验的程序员。本书是一本有关 Python 的参考指南，包括 Python 本身、Python 庞大的标准库中最常用的部分，以及一些最流行和最有用的第三方模块和软件包，这些第三方模块和软件包涵盖了广泛的应用程序开发领域，包括 Web 和网络编程、图形用户界面（GUI）、XML 处理、数据库交互操作以及高速数值计算等。本书将主要关注 Python 的跨平台功能，还将介绍如何扩展 Python 模块，以及如何将 Python 嵌入使用 C 或 Java™ 编写的其他应用程序中的一些基础知识。

本书的组织结构

本书分为以下 5 个部分。

第 1 部分“Python 入门指南”

第 1 章“Python 简介”

本章介绍了 Python 语言及其实现的一般特性，并介绍在哪里可以获得有关 Python 的帮助和信息。

第 2 章“安装”

本章介绍了如何获得 Python 以及在开发者的计算机上安装 Python。

第 3 章“Python 解释器”

本章介绍了 Python 解释器程序、解释器的命令行选项，以及如何使用 Python 解释器在交互式会话中运行 Python 程序。本章还提到了一些特别适合于编辑 Python 源代码的文本编辑器和可以全面检查 Python 源代码的辅助程序，还介绍了一些已经发展得很完善的集成开发环境，包括 IDLE，IDLE 是标准 Python 发布版本附带的免费集成开发环境。

第 2 部分“核心 Python 语言和内置对象”

第 4 章“Python 语言”

本章介绍了 Python 的语法、内置数据类型、表达式、语句，以及如何编写和调用 Python 函数。

第 5 章“面向对象的 Python”

本章介绍了 Python 中的面向对象编程功能。

第 6 章“异常”

本章介绍了如何处理 Python 程序中的错误和异常情况。

第 7 章“模块”

本章介绍了如何使用 Python 软件将代码组合到模块和包中，如何定义和导入模块，以及如何安装以标准 Python 方式打包的第三方 Python 扩展模块。

第 8 章“核心内置”

本章介绍了 Python 的内置数据类型和内置函数，以及标准 Python 库中最基础的模块（粗略地讲，也就是在其他一些编程语言中被内置到语言本身以提供各种功能的模块）。

第 9 章“字符串和正则表达式”

本章介绍了 Python 强大的字符串处理功能，包括 Unicode 字符串和正则表达式。

第 3 部分“Python 库和扩展模块”

第 10 章“文件和文本操作”

本章介绍了如何使用内置 Python 文件对象、来自 Python 标准库中的一些模块和用于富文本（Rich text）I/O 的平台相关扩展模块对文件和文本进行处理。本章还介绍了有关国际化和本地化的问题，以及如何使用 Python 定义文字模式的交互式命令会话这样的特殊任务。

第 11 章“持久化和数据库”

本章介绍了 Python 的序列化机制和持久化机制，以及 Python 与 DBM 数据库、Berkeley 数据库和关系数据库（基于 SQL）之间的接口。

第 12 章“时间操作”

本章介绍了在 Python 中如何使用标准库和常用扩展模块来处理时间和日期。

第 13 章“控制执行”

本章介绍了如何在 Python 中完成高级执行控制，包括对动态生成的代码的执行控制和对垃圾收集操作的控制。本章还介绍了 Python 的一些内部类型，以及与注册“清理”

函数有关的特殊问题，“清理”函数是在程序终止时执行的函数。

第 14 章“线程和进程”

本章介绍了 Python 的并发执行功能，包括如何在一个进程中运行多个线程，以及如何在单台计算机上运行多个进程。本章还介绍了如何访问进程的环境，以及如何通过内存映射机制访问文件。

第 15 章“数值处理”

本章介绍了 Python 标准库模块和第三方扩展包中的数值计算功能。本章特别介绍了如何使用十进制浮点型数字，而不是默认的二进制浮点型数字。还介绍了如何获得和使用伪随机数和真正的随机数。

第 16 章“数组处理”

本章介绍了用于执行数组处理的内置对象和扩展包，主要针对传统的 Numeric 第三方扩展，并提到了其他一些最近开发的数组处理解决方案。

第 17 章“Tkinter GUI”

本章介绍了如何使用标准 Python 发布版本附带的 Tkinter 包开发 Python 图形用户界面，并简要提到了其他一些可选的 Python GUI 框架。

第 18 章“测试、调试和最优化”

本章介绍了如何使用 Python 工具和方法来确保开发者的程序的正确性（也就是说，程序完成了开发者想要其实现的功能），查找并纠正程序中的错误，以及检查并增强程序的性能。本章还介绍了“警告”的概念和用来处理“警告”的 Python 库模块。

第 4 部分“网络和 Web 编程”

第 19 章“客户端网络协议模块”

本章介绍了 Python 标准库中用来帮助程序员编写网络客户端程序的一些模块，并专门通过从客户端处理各种网络协议和处理 URL 来介绍这些模块。

第 20 章“套接字和服务器端网络协议模块”

本章介绍了 Python 与底层网络机制（套接字）的接口，用来帮助程序员编写网络服务器程序的标准 Python 库模块，以及如何使用标准模块和强大的 Twisted 扩展模块进行异步（事件驱动）网络编程。

第 21 章“CGI 脚本和其他解决方案”

本章介绍了 CGI 编程的基础知识，如何使用标准 Python 库模块在 Python 中实现 CGI 编程，以及如何使用“cookie”处理 HTTP 服务器端编程中的会话状态。本章还提到了许多 CGI 编程的可选方案，这些可选方案可以使用流行的 Python 扩展模块来实现服务

器端 Web 编程。

第 22 章“MIME 和网络编码方式”

本章介绍了如何在 Python 中处理电子邮件，以及如何处理其他网络数据结构和编码的文档。

第 23 章“结构化文本：HTML”

本章介绍了可以用来处理和生成 HTML 文档的 Python 库模块。

第 24 章“结构化文本：XML”

本章介绍了可以用来处理、修改和生成 XML 文档的 Python 库模块和流行的扩展模块。

第 5 部分“扩展和嵌入”

第 25 章“扩展和嵌入经典 Python”

本章介绍了如何使用 C 和其他经典编译语言来编写 Python 扩展模块、如何在用其他语言编写的应用程序中嵌入 Python，扩展 Python 以及访问现有 C、C++和 Fortran 库的其他可选方法。

第 26 章“扩展和嵌入 Jython”

本章介绍了如何使用 Python 的 Jython 实现中的 Java 类，以及如何在 Java 编写的应用程序中嵌入 Jython。

第 27 章“发布扩展和程序”

本章介绍了几个可以帮助开发者对 Python 扩展、模块和应用程序进行打包并进行发布的工具。

本书使用的惯例

本书使用了以下这些惯例。

引用惯例

在函数/方法引用输入参数中，只要有可能，每个可选参数都将被显示为一个使用 Python 语法 name=value 表示的默认值。内置函数不需要接受命名参数，因此参数的名称并不是很重要。有些可选参数可以按照这些参数的出现与否得到最好的诠释，而不是通过其默认值。在这些情况下，本书通过在方括号（[ ]）中包含这个参数以表示该参数是可选的。在有多于一个的可选参数时，将形成嵌套的方括号。

如何使用本书中的代码示例

本书的目的是帮助读者完成自己的 Python 开发工作。一般情况下，读者有可能会在自

己的程序和文档中使用本书中的代码。对此，不需要联系出版社以获得许可，除非打算利用本书中的主要代码进行二次商业开发。例如，使用本书中的一部分代码段编写程序是不要求获得许可的。但是，如果要销售或发布一张包含来自 O'Reilly 图书中的示例的 CD-ROM，则要求获得许可。如果要将本书中的大量示例代码复制到自己开发的产品的资料中，则必须获得许可。

我们感谢，但并不要求读者在使用本书的代码示例时注明引用源。引用源通常包括标题、作者、出版社和 ISBN。例如，*Python in a Nutshell*, Second Edition, by Alex Martelli. Copyright 2006 O'Reilly Media, Inc., 0-596-10046-9。

如何联系我们

作者已经尽自己最大努力测试并验证了本书中的所有内容，但是读者可能还是会发现有些功能被更改了（甚至会在书中找到一些错误!）。请将发现的任何错误告知出版社，还可以告诉出版社对本书将来版本的建议，通信地址是：

美国：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

100080 北京市西城区西直门成铭大厦 C 座 807 室

奥莱利技术咨询（北京）有限公司

出版社为本书提供了一个网页，网页上列出了本书的勘误表、示例和所有其他信息。可以通过下面的链接访问这个网页：

http://www.oreilly.com/catalog/pythonian2

要想询问技术问题或者对本书进行评论，请发送电子邮件到：

bookquestions@oreilly.com

info@mail.oreilly.com.cn

要想获得有关 O'Reilly 的图书、会议、资源中心和 O'Reilly Network 的更多信息，请访问 O'Reilly 的网站：

http://www.oreilly.com

致谢

衷心感谢曾经在这本书上帮助过我的所有人，包括本书第 1 版和现在的第 2 版。许多 Python 的初学者、专业人士和专家曾经阅读过本书的一部分初稿，并提供了一些反馈

意见，这些意见可以帮助我让这本书更加清楚、精确和正确，并具备更好的可读性。除了要感谢所有这些提供了大量高质量的反馈意见和其他帮助的人之外，我还必须单独感谢我在 Google 公司的同事们，特别是 Neal Norwitz 和 Mohsin Ahmed。

本书的第 1 版得到了来自于 Python 各个领域的专家（Aahz 是有关线程的专家、Itamar Shtull-Trauring 是有关 Twisted 的专家、Mike Orr 是有关 Cheetah 的专家、Eric Jones 和 Paul Dubois 是有关 Numeric 的专家，而 Tim Peters 是有关线程、测试和最优化的专家）、一个非常优秀的技术审校小组（包括 Fred Drake、Magnus Lie Hetland、Steve Holden 和 Sue Giller）以及本书的编辑 Paula Ferguson 不可或缺的帮助。本书的第 2 版也从编辑 Jonathan Gennick 和 Mary O'Brien、技术审校 Ryan Alexander、Jeffery Collins 和 Mary Gardiner 的辛勤工作中受益匪浅。我要对 O'Reilly 工具小组的优秀员工们致以特殊的感谢，他们（不管是直接和亲自的帮助，还是通过他们开发出来的那些非常有帮助的工具）帮助我解决了几个非常困难的技术问题。

一直以来，我非常思念我的家庭：我的孩子 Flavia 和 Lucio，我的姐姐 Elisabetta 和我的父亲 Lanfranco，他们已经回到了我的祖国意大利，而我在 Google 的工作让我来到了美国的加利福尼亚。

但是，在我内心深处，无比感谢，甚至不止是感谢的一个极其了不起的人，就是我的妻子 Anna Martelli Ravenscroft，我的 *Python Cookbook* 一书第 2 版的合作者，Python 软件基金会的会员之一，以及所有作者都可能会梦寐以求的最严格、最出色的技术审校。除了对本书数不清的直接贡献之外，在过去的一年中，Anna 还非常神奇地设法为我创造了足够多的和睦、安静和自由时间（尽管我同时还承担了 Google 公司的 Uber Tech Lead 这个非常奇妙和具有挑战性的工作），是她使得这本书成为可能。实际上，这也是一本属于她的书，她至少付出了和我同样多的努力。

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

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

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

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

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

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

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

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

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

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

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

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

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

目录

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

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

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

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

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

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

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

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

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

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

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

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

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

第 1 部分

Python 入门指南

第 1 章

Python 简介

Python 是一个通用目的编程语言。Python 已经发展了很多年：Python 的创造者 Guido van Rossum 从 1990 年就开始了 Python 的开发。这个稳定而成熟的语言是非常高层的、动态的、面向对象的和跨平台的，所有这些特征都非常吸引开发者。Python 可以在所有主流的硬件平台和操作系统上运行，因此 Python 不会限制开发者的平台选择。

Python 为软件的生命周期的每个阶段都提供了非常高的效率：分析、设计、原型、编码、测试、调试、优化、文档、部署和维护。几年来，Python 的流行已经日渐稳定，并且在不间断地增长。如今，熟悉 Python 对于每个程序员来说都是一种优势，因为 Python 已经渗透到了每一种开发环境中，并且 Python 具有一些非常有用的功能，可以作为任何一种软件解决方案的一部分。

Python 提供了一种非常独特的功能，可以将优雅、简单、实用性和强大功能融合在一起。使用 Python，开发者可以快速提高开发效率，这要感谢 Python 提供的一致性和规律性、丰富的标准库，以及许多可以很容易在 Python 中使用的第三方模块。Python 是很容易学习的，因此，如果读者刚开始学习编写程序，非常适合于从学习 Python 开始，与此同时，对于大多数非常有经验的专家而言，Python 的功能也是足够强大的。

1.1 Python 语言

Python 语言并不是一种追求最简单化的语言，但在提供好的实用性功能之外，Python 语言并没有什么多余之处。一旦一种语言提供了一种很好方法来表达一种设计思想，添加其他方法只能获得非常有限的好处，而且这样做的费用会根据语言的复杂性按照添加功能的数量以高于线性的速度增长。复杂的语言往往要比简单的语言更难学习和精通，也很难提高实现效率并且不出现错误。一种语言中的任何复杂和怪异的地方都会妨碍软件维护的效率，尤其是在大型项目中，因为在这些项目中，许多开发人员需

要相互合作，并且经常需要维护最初由其他人编写的代码。

Python 是简单的，但是并没有被过分单纯化。Python 遵循这样一种思想，那就是，如果一种语言在多个上下文环境下表现出一种特定的方式，那么这种语言在所有上下文环境下都应该可以很理想地以类似的方式工作。Python 还遵循这样一个原则，那就是，一种语言不应该具有为了便利而创建的特殊快捷方式、特殊用法、奇特的异常情况、过于细致的使用区别，或者比较神秘和不为人知的一些隐藏的优化方法。与任何其他人工设计的工艺品一样，一种好的语言必须能够在体验性、常识性和高度实用性等一般原则之间寻求一种平衡。

Python 是一种通用目的编程语言，因此 Python 最显著的特点就是几乎在软件开发的任何领域都可以使用。Python 可以作为任何领域的最佳解决方案的一部分。在这里，“一部分”是一个非常重要的词，尽管许多开发者发现 Python 可以满足他们的所有需求，但是 Python 是不应该被孤立使用的。Python 程序可以很容易地与其他各种语言组件一起使用，这使得 Python 成为一种理想的，可以将其他各种语言编写的组件结合在一起使用的语言。

Python 是一种非常高层的语言（Very-High-Level Language，VHLL）。这意味着 Python 使用了更高层的抽象，从概念上讲，Python 要比一些经典的编译语言更远离底层计算机，比如 C、C++和 Fortran，这些语言从传统上被称为高层语言。还有，Python 要比经典的高层语言更简单、处理更快，也更有规则。这样就为程序员提供了非常高的开发效率，并使得 Python 成为一个非常吸引人的开发工具。在经典编译语言中，好的编译器通常可以生成二进制机器代码，运行起来要比 Python 代码快得多。但是，在大多数情况下，使用 Python 编程的应用程序的性能已经被证明是足够的。如果仍然不满足开发者的性能要求，开发者可以应用本书第 18.4 节中介绍的最优化技术以提高程序的性能，同时保留 Python 高编程开发效率的好处。

一些比较新的编程语言，比如 Java 和 C#，都是比经典编程语言（比如 C 和 Fortran）稍微更高层（远离计算机）的语言，它们同时还具有经典编程语言的一些特性（比如需要使用声明）和 Python 这样的 VHLL 语言的一些特性（比如在典型实现中使用可移植的字节代码作为编辑目标，以及使用垃圾收集功能来减轻程序员管理内存的需要）。如果开发者发现自己使用 Java 或 C#的开发效率要比使用 C 或 Fortran 更高，可以试试使用 Python（可以使用 Jython 或 IronPython 实现，请参见第 3.1 节），这样会变得更有效率。

从语言级别来看，可以将 Python 与其他强大的 VHLL 语言比较，比如 Perl 或 Ruby。不过，Python 还具有简单性和规则性的优点。

Python 是一种面向对象编程语言，但是 Python 可以让开发者使用面向对象和传统程序风格，有时候还可以使用功能化程序设计风格来开发代码，并根据开发者的应用程序要求混和和匹配这些风格。Python 的面向对象特性与 C++的面向对象特性类似，但是

Python 的使用要简单得多。

1.2 Python 标准库和扩展模块

与 Python 编程有关的内容要比 Python 语言本身的内容多很多：从有效使用 Python 的角度，标准 Python 库和其他扩展模块几乎与 Python 语言本身具有相同的重要性。Python 标准库支持许多精心设计的、可靠的和 100%的纯 Python 模块，开发者可以很方便地重用这些模块。Python 标准库中包含一些可以用来执行像显示数据、字符串和文本处理、与操作系统和文件系统交互操作，以及 web 编程这样的任务的模块。因为这些模块都是用 Python 编写的，因此这些模块可以在 Python 支持的所有平台上使用。

来自于标准库或其他地方的扩展模块可以帮助 Python 代码访问底层操作系统或其他软件组件支持的功能，这些软件组件包括图形用户界面（GUI）、数据库和网络。扩展模块还可以为计算密集型任务提供最大的速度，比如 XML 解析和数值数组计算。不过，对于那些不是使用 Python 编写的扩展模块，则没有必要像纯 Python 代码那样具有相同的跨平台可移植性。

开发者可以使用底层语言编写特定目的扩展模块，这样可以为开发者最初使用 Python 进行原型设计的小的、计算密集型部分获得最大性能。开发者还可以使用像 SWIG 这样的工具将现有 C/C++库封装为 Python 扩展模块，参见本书第 25.2 节。最后，开发者还可以将 Python 嵌入以其他语言编程的应用程序中，通过专用 Python 扩展模块将现有应用程序的功能放到 Python 脚本中。

本书介绍了许多来自 Python 标准库和其他源的模块，这些模块的范围包括客户端和服务器端网络编程、图形用户界面（GUI）、数值数组处理、数据库、文本和二进制文件处理，以及与操作系统的交互操作。

1.3 Python 的实现

目前，Python 有三个产品级质量的实现，被称为 CPython、Jython 和 IronPython，以及几个其他实验性实现，比如 PyPy。本书将主要介绍 CPython，这是最广泛使用的一种 Python 实现，为了简化，本书将其称之为 Python。但是，语言及其实现之间的区别是一个非常重要的区别。

CPython

经典 Python（也就是 CPython，通常直接被称为 Python）是最快、最新、最稳定和最完整的 Python 实现。因此，CPython 可以被认为是 Python 语言的“参考实现”。CPython 是一个编译器、解释器和一组内置对象和可选扩展模块的集合，所有模块都是使用标

准 C 语言编写的。CPython 可以在安装的 C 编译器符合 ISO/IEC9899:1990 标准的任何平台上使用（例如，所有现代的、常用的平台）。在第 2 章中，将介绍如何下载和安装 CPython。除本书第 26 章和少数几节有明确说明的地方之外，本书中的所有内容使用的都是 CPython，因为 CPython 是最广泛使用的 Python 版本。

Jython

Jython 是一个可以在所有支持 Java 1.2 或更高版本的 Java 虚拟机（JVM）上使用的 Python 实现。这样的 JVM 在所有常用的、现代的平台上都可以使用。使用 Jython，开发者可以使用所有 Java 库和框架。为了最好地使用 Jython，开发者需要熟悉一些基本的 Java 类。开发者不必使用 Java 来编程，但是现有 Java 类的文档和示例都是以 Java 术语来说明的，因此开发者需要对 Java 有一些了解才能阅读和理解这些文档和示例。开发者还需要使用 Java 支持工具来完成像操作.jar 文件和签名 Java 程序（applet）这样的任务。但是，本书要介绍的是 Python，而不是 Java，对此不再赘述。要想使用 Jython，除了本书之外，读者还需要补充阅读 O'Reilly 出版，Noel Rappin 和 Samuele Pedroni 编著的 *Jython Essentials* 一书，或者 O'Reilly 出版，David Flanagan 编著的 *Java in a Nutshell* 一书，并且，如有必要，还可以进一步了解许多其他可用的 Java 资源。

IronPython

IronPython 是一个可以在 Microsoft 设计的公共语言运行时（Common Language Runtime, CLR）上使用的 Python 实现，CLR 更通俗的说法就是.NET。使用 IronPython，开发者可以使用所有的 CLR 库和框架。除了 Microsoft 自己的实现，还可以将 CLR 的一个跨平台实现（也被称为 Mono）与其他非 Microsoft 操作系统平台一起使用，就像在 Windows 上使用一样。为了最好地使用 IronPython，开发者需要熟悉一些基础的 CLR 库。开发者不必使用 C#来编程，但是现有 CLR 库的文档和示例通常都是以 C#术语来说明的，因此开发者需要对 C#有一些了解才能阅读和理解这些文档和示例。开发者还需要使用 CLR 支持工具来完成像创建 CLR 程序集这样的任务。但是，本书要介绍的是 Python，而不是 CLR，对此不再赘述。要想使用 IronPython，除了本书之外，开发者还需要补充阅读 IronPython 本身的在线文档，并且，如有必要，还可以了解其他一些有关.NET、CLR、C#、Mono 的可用资源。

在 CPython、Jython 和 IronPython 之间进行选择

如果开发者的平台可以运行 CPython、Jython 和 IronPython，那么如何选择使用这些 Python 实现呢？首先，不要进行选择，直接下载并安装所有这些 Python 实现。Python 实现之间是可以共存的，不会产生任何问题，并且它们都是免费的。将这些 Python 实现全部安装在计算机上只需要花费一些下载时间和很小的额外硬盘空间。

这些 Python 实现之间的主要区别在于它们的运行环境，以及它们可以使用的库和框架。

如果开发者需要在 JVM 环境下工作，则 Jython 就是最佳选择。如果开发者需要在 CLR 环境下工作，最好使用 IronPython。如果开发者主要在传统环境下工作，则 CPython 就是最适合的选择。如果开发者并没有对某一种 Python 实现有强烈的偏好，则应该从标准的 CPython 参考实现开始工作。

换句话讲，在开发者只是在体验、学习和试验的时候，最常见的是使用 CPython，因为 CPython 的发展更快一些。为了开发和部署，开发者的最佳选择取决于想要使用的扩展模块和想要如何发布完成的应用程序。CPython 应用程序通常会快一些，尤其是在可以使用适当的扩展模块时，比如 Numeric（参见本书第 16 章）。CPython 更成熟；CPython 已经经历了很长时间，而 Jython，尤其是 IronPython 都是新出现的，实用性验证比较少。CPython 版本的开发也趋向于继续比 Jython 和 IronPython 版本的开发更快一些：例如，在编写本书的过程中，可以使用的 Python 语言版本是：Jython 的版本为 2.2、IronPython 的版本为 2.4，而 CPython 的版本正快速从 2.4 向 2.5 前进（当读者阅读到本书的时候，新版本应该已经发布了）。

不过，正如读者在本书第 26 章中将了解的，Jython 可以将任何 Java 类作为一个扩展模块使用，不管这个类来自标准 Java 库、第三方的 Java 库还是读者自己开发的 Java 库。同样地，IronPython 可以使用任何 CLR 类，不管该类来自标准 CLR 库，还是使用 C#、Visual Basic .NET 或者其他 CLR 兼容语言编写的类。使用 Jython 编程的应用程序是一个 100%的纯 Java 应用程序，这样的应用程序具有使用 Java 开发部署应用程序的所有好处和问题，并且可以在任何安装了适当的 JVM 的目标机上运行。这些应用程序打包的方式也与 Java 完全相同。同样地，使用 IronPython 编程的应用程序也完全符合.NET 规范。

Jython、IronPython 和 CPython 都是非常好的和可靠的 Python 实现，并且在可用性和性能方面都有各自不同的特点。因为每个 JVM 和 CLR 平台都附带了许多软件包，而且提供了大量有用的库、框架和工具，因此每个 Python 实现都可以在特定的开发环境和应用场景下体现出具有决定意义的实用性优点。最明智的做法就是熟悉每个 Python 实现的优点和缺点，然后为不同的开发任务选择最佳的 Python 实现。

PyPy 和其他实验性版本

除上述 Python 实现之外，还有几个非常有趣的 Python 实现，尽管在编写本书的时候，这些实现还不适合于作为产品使用，但是，开发者还是很值得了解这些 Python 实现中的一些基本特性和将来的潜在发展的。人们正在积极工作的两个这样的实验性实现是 Pirate（http://pirate.tangentcode.com）和 PyPy（http://codespeck.net/pypy/）。Pirate 是一个基于 Parrot 虚拟机的 Python 实现，该实现还支持 Perl 6 和其他 VHL，PyPy 是使用 Python 自身编写的一个快速和灵活的 Python 实现，该实现是以几种底层语言和使用像类型推断这样的高级技术的虚拟机作为目标的。

许可证和价格问题

CPython 被包含在 CNRI 开源 GPL-兼容许可证（Open Source GPL-Compatible License）之中，并允许开发者免费使用 Python 进行商业和免费软件开发（参见 http://www.python.org/2.4.2/license.html）。Jython 和 IronPython 的许可证也具有相同的自由度。开发者从主要的 Python、Jython 和 IronPython 的网站上下载的任何内容都不会花费一分钱。这些许可证并没有限制开发者为自己使用 Python 提供的工具、库和文档开发出来的软件指定什么样的许可证和价格条款。

不过，并不是与 Python 有关的所有东西的许可证费用都是完全免费，或者不会引起任何争端的。开发者可以免费下载的许多第三方 Python 源代码、工具和扩展模块都拥有免费许可证，类似于 Python 本身。然而，GNU 公共许可证（GNU Public License, GPL）或者较宽松 GNU 公共许可证（Lesser GPL, LGPL）涵盖的其他源代码、工具和扩展模块对开发者可以赋予新开发的工作的许可条件进行了限制。某些商业开发的模块和工具可能会要求开发者支付费用，要么要求开发者无条件支付费用，要么要求开发者在使用这些模块和工具从事商业行为并获利时支付费用。

仔细检查 Python 相关的许可条件和价格是不可避免的。在开发者投入时间和精力开发任何软件组件之前，请检查自己是否可以接受其许可证。通常，尤其是在公司环境下，这样的法律事件可能需要引入咨询律师。除非本人明确说明，本书中涉及的模块和工具在编写本书的时候都可以认为是允许免费下载的开源代码，并且被涵盖在 Python 的免费许可证之下。但是，我并不是法律专家，并且许可证是会随时间而改变的，因此再次认真检查总是明智之举。

1.4 Python 的开发和版本

Python 是由以 Guido van Rossum 为首的一个核心开发小组开发、维护和发布的，Guido van Rossum 是 Python 的发明者、架构师和“Benevolent Dictator For Life（BDFL）”。这个称号意味着 Guido 有权最终决定哪些内容可以成为 Python 语言和标准库中的一部分。Python 的知识产权属于 Python 软件基金会（Python Software Foundation, PSF），一个致力于推广 Python 的非营利机构，这个基金会包括许多个人成员（根据人们对 Python 的贡献进行提名，其中包括所有 Python 核心开发团队）和企业赞助商。大多数 PSF 成员都有向 Python 的 SVN 知识库（http://svn.python.org/projects/）提交资料的权利，并且大多数向 Python SVN 提交资料的人都是 PSF 的成员。

对 Python 提议的更改被详细记录在名为 Python 增强提议（Python Enhancement Proposal, PEP）的公共文档中，由 Python 开发者和更广泛的 Python 社区进行讨论（有时候通过投票表决），最后由 Guido 接受或拒绝，Guido 将考虑讨论和表决的结果，但并不受此约束。现在，有好几百人正通过 PEP、讨论、bug 报告和对 Python 源代码、库和文档

提议的补丁活跃地对 Python 的开发做出贡献。

Python 核心团队发布了 Python 的次要版本（Minor release）（2.x，x 的值将会增加），目前大约每年或每两年增加一个次要版本。Python 2.2 是在 2001 年 10 月发布的，2003 年 7 月发布了 Python 2.3，2004 年 11 月发布了 Python 2.4。Python 2.5 计划在 2006 年夏天发布（在编写本书的时候，Python 2.5 的第一个 alpha 版本刚刚出现）。每个次要版本都会添加一些特性，这些特性使得 Python 功能更加强大，使用也更简单，同时也会考虑维持向后兼容性。但是，总有一天会发布 Python 3.0 版本，为了删除某些冗余的“旧”特性并进一步简化语言本身，这个版本从某种程度上将不再提供向后兼容性。不过，这个版本也会在未来几年才可能出现，现在并没有为这个版本制订专门的时间表；Guido 对 Python 3.0 的一些想法的当前状态可以通过 http://python.org/peps/pep-3000.html 进行了解。

每个次要版本 2.x 都是从 alpha 版本开始的，这些版本被标记为 2.xa0、2.xa1 等。在 alpha 版本之后，至少会有一个 beta 版本，如 2.xb1，而在 beta 版本之后，至少会有一个候选版本，如 2xrc1。在最终发布 2.x 版本的时候，这个版本应该是稳定和可靠的，并且应该在所有主流平台上经过了严格测试。任何 Python 程序员都可以通过下载 alpha 版本、beta 版本和候选版本，对其进行充分地测试，并将可能出现的任何问题整理成 bug 报告文件以帮助 Python 确保版本的稳定和可靠。

在发布了次要版本之后，核心开发团队的部分注意力将会转移到下一个次要版本。但是，次要版本通常也会有后续的维护版本（Point release）（例如，2.x.1、2.x.2 等），这些维护版本并没有添加新功能，但是可以修改错误，而且还可以将 Python 移植到新平台上、提供增强的文档，以及添加优化和工具。

本书主要针对 Python2.4（及其所有维护版本），这是在本书编写期间最稳定和广泛使用的版本。本书还介绍，或者至少是提到了 Python 2.5 中计划将要出现的更改，并且还说明了 2.4 版本首次引入了哪些语言和库的新内容，而这些内容在 2.3 版本中是不能使用的。在本书每次提到 2.4 版本中的某个特性时，表示 2.4 版本及其以后版本（换句话讲，这个说法意味着包含 Python 2.5 但是不包含 Python 2.3），除非本书立即继续解释这些是与 2.5 版本不同的地方。

在编写本书的时候，Jython 发布的版本只支持 Python 2.2（并且是部分支持，而不是全部支持 Python 2.3），但是不支持 Python 2.4 的全部规范。IronPython 1.0 支持 Python 2.4。

本书并不打算针对 Python 的老版本进行介绍，比如 1.5.2、2.0、2.1 和 2.2，这些版本都已经超过 4 年时间了，开发者不应该再使用这些版本进行任何新的开发。但是，如果这些版本被嵌入到一些需要开发者处理的应用程序中，开发者可能需要担心这些老版本的使用。幸运的是，Python 的向后兼容功能是很好的：Python 的当前版本可以正确地处理任何使用 Python1.5.2 及其以后版本编写的可用的 Python 程序。开发者可以从 http://python.org/doc/versions.html 找到 Python 的所有老版本的代码和文档。

1.5 Python 的资源

Python 最丰富的资源就是 Internet。最佳的起始点就是 Python 的网站 http://www.python.org，这个网站包含很多值得浏览的有趣链接。如果读者对 Jython 感兴趣，那么必须浏览 http://www.jython.org 网站。对于 IronPyton，在编写本书的时候与之最相关的网站是 http://workspaces.gotdotnet.com/ironpython，但是 IronPython 开发团队近期的计划包括恢复 http://ironpython.com 网站；当读者看到本书时，http://ironpython.com 网站应该已经回来了，该网站将作为主要的 IronPython 网站。

文档

Python、Jython 和 IronPython 都附带了非常好的文档。这些指南提供了各种文件格式，适合于读者浏览、搜索和打印。读者可以在网站 http://www.python.org/doc/current 上浏览这些指南。读者还可以从 http://www.python.org/doc/current/download.html 找到可供下载的各种文件格式的链接，并且 http://www.python.org/doc/上还有大量文档的链接。对于 Jython，与其他 Python 一样，http://www.jython.org/docs/上有专门针对 Jython 的文档的链接。Python FAQ（常见问题）文档位于 http://www.python.org/doc/FAQ.html，Jython 专用 FAQ 文档位于 http://www.jython.org/cgi-bin/faqw.py?req=index。

大多数 Python 文档（包括本书）都假定读者已经具备了一些软件开发的基础知识。不过，Python 是非常适合初学者的，因此对于这个假定也存在一些例外情况。下面为那些还不是程序员的读者列出了一些好的在线介绍网站：

- Josh Cogliati 编写的 *Non-Programmers Tutorial For Python*，参见 http://www.honors.montana.edu/~jjc/easytut/easytut/；
- Alan Gauld 编写的 *Learning to Program*，参见 http://www.freenetpages.co.uk/hp/alan.gauld/；
- Allen Downey 和 Jeffrey Elkner 编写的 *How to Think Like a Computer Scientist (Python Version)*，参见 http://www.ibiblio.org/obp/thinkCSpy/。

新闻组和邮件列表

网站 http://www.python.org/community/lists/中包含一些与 Python 相关的邮件列表和新闻组的链接。在读者向邮件列表或新闻组发送任何邮件时，请使用纯文本格式，而不是 HTML 格式。

用于 Python 讨论的 Usenet 新闻组是 comp.lang.python。这个新闻组也可以作为邮件列表使用。要想订阅该新闻组，可以向 python-list-request@python.org 发送一封正文为单词“subscriber”的邮件，每周都会有一个非常有趣的收集了每周大多数值得注意的新

闻和 Python 资源的“Dr. Dobb's Python URL!”发布到 comp.lang.python 上。读者还可以访问网址 http://groups.google.com/groups?q=+Python-URL!+group%3Acomp.lang.python&start=0&scoring=d&查看以时间顺序倒序（最近的放在最前面）排列的所有问题。

读者还可以在网站 http://www.pythonware.com/daily/上看到一个有些类似于 Python 每日新闻的列表。

与 Python 相关的公告将被发布在 comp.lang.python.announce 上。为了订阅其邮件列表，可以向 python-announce-list-request@python.org 发送一封正文为单词“subscribe”的邮件。要想订阅 Jython 邮件列表，可以访问 http://lists.sf.net/lists/listinfo/jython-users。要想获得有关 Python 的个人帮助，可以将问题通过电子邮件发送到 python- help@python.org。对于有关使用 Python 教学或学习编程的问题和讨论，可以发送电子邮件到 tutor@python.org。

特别兴趣（Special-Interest）组

人们可以在 Python 特别兴趣组（Special Interest Groups, SIG）的邮件列表中讨论与 Python 有关的一些特定问题。http://www.python.org/sigs/ 页面上包含了一个活跃的 SIG 的列表和有关这些 SIG 的一般和特殊信息。在编写本书的时候大约有十几个 SIG 是活跃的。下面是几个示例：

http://www.python.org/sigs/c++-sig/　C++和 Python 之间的绑定；

http://www.python.org/sigs/i18n-sig/　Python 程序的国际化和本地化；

http://www.python.org/sigs/image-sig/　Python 中的图像处理。

Python 商务论坛

Python 商务论坛（Python Business Forum，PBF）是一个由基于 Python 实施商务行为的公司组成的国际化社区，网站为 http://www.python-in-business.org/。PBF 的网站提供了有关 Python 的许多商务应用的一些有趣信息。

Python 期刊

Python 期刊（Journal）是一个关注于 Python 的免费在线出版物，主要用来介绍如何使用 Python 及其应用程序，其网站为 http://pythonjournal.cognizor.com。

扩展模块和 Python 源代码

探索可用的 Python 扩展模块和源代码的一个很好的起点就是“Python 奶酪店”（Python Cheese Shop），网站为 http://www.python.org/pypi，该网站当前包含了超过 1200 个包含说明和指导的软件包。另一个很好的资源就是“The Vaults of Parnassus”，网址为

http://www.vex.net/parnassus/，该网站包含超过 2000 个分类和带有评论的链接。通过这些链接，开发者可以最方便地查找和下载可用的 Python 模块和工具。

标准的 Python 源代码发布版本在其标准库，以及 Demos 和 Tools 目录中包含了最好的 Python 源代码和许多内置扩展模块的 C 源代码。即使开发者没有兴趣从这些源代码构建 Python 程序，出于学习的目的，还是可以下载和打开这些发布的 Python 源代码软件包的。

本书中介绍的许多 Python 模块和工具也是有其专门网站的。请参考本书的相应章节中介绍的网站。

Python 食谱

ActiveState 网站 http://www.activestate.com/ASPN/Python/Cookbook 主要用来动态收集与 Python 程序有关的食谱。每个食谱包含了 Python 代码、注释和讨论，这些信息都是在 David Ascher 的监督之下，由志愿者贡献并由读者进行丰富和完善的。所有这些代码都涵盖在一个类似于 Python 的许可证之下。每个人都可以被邀请为这个社区事业的作者和读者。经过 Python 专家的编辑、注释，并分组到包含介绍的章节，来自这个网站的数百个食谱由 O'Reilly 出版为 *Python Cookbook* 一书，这本书是由 Alex Martelli、Anna Martelli Ravenscroft 和 David Ascher 共同编著的。

书籍和杂志

尽管网络提供了非常丰富的信息资源，但是书籍和杂志仍然占据非常重要的位置（如果读者和作者本人不同意这一点，也就不会编写这本书了，而读者也将不会看到这本书）。在编写本书的时候，唯一的一本完全专注于 Python 的杂志是 *Py*（要想获得最新的信息，请访问 http://www.pyzine.com/）。

有关 Python 和 Jython 的书是非常多的。我推荐以下几本书，尽管其中的一些只涵盖了 Python 语言的老版本，而不是最新版本。

- 如果读者只是刚开始学习 Python（但是有一些编程经验），可以阅读 O'Reilly 出版，Mark Lutz 和 David Ascher 编著的 *Learning Python* 一书。这本书介绍了 Python 语言和核心库的一些基础知识，清晰并且深入地介绍了涉及的每个主题。
- New Riders 出版，Steve Holden 编著的 *Python Web Programming* 一书，这本书介绍了 Python 和许多用来帮助开发者构建动态网站的其他技术的基础知识，这些技术包括 TCP/IP、HTTP、HTML、XML 和关系数据库。这本书还提供了一些很实用的例子，包括一个完整的基于数据库的网站。
- Apress 出版，Mark Pilgrim 编著的 *Dive Into Python* 一书，这本书通过示例以快速和全面的方式介绍了 Python，非常适合于那些已经精通其他语言编程的读者。读者还可以从 http://diveintopython.org/免费下载这本书，该网站提供了几种可供下载

的文件格式。

- Apress 出版，Magnus Lie Hetland 编著的 *Beginning Python: From Novice to Professional* 一书，这本书通过完整地解释和完全开发的各种应用程序领域的 10 个完整程序来介绍 Python。
- O'Reilly 出版，Mark Hammond 和 Andy Robinson 编著的 *Python Programming on Win32* 一书，要想在 Windows 操作系统下最好地使用 Python，这本书是必不可少的。这本书详细介绍了用于 COM、ActiveScripting、Win32 API 调用，以及与 Windows 应用程序集成的 Python 特定平台扩展模块。这本书使用的是 Python 的老版本 1.5.2，但是所有内容都可以应用到 Python 的当前版本上。
- O'Reilly 出版，Samuele Pedroni 和 Noel Rappin 编著的 *Jython Essentials* 一书，这本书是一本内容丰富但简洁明了的有关 Jython 的书，适合于对 Java 已经有一些了解的读者。为了有效地使用 Jython，还推荐 O'Reilly 出版，David Flanagan 编著的 *Java in a Nutshell* 一书。
- New Riders 出版，David Beazley 编著的 *Python Essential Reference* 一书，这本书是一本完整介绍 Python 语言及其标准库的参考指南。
- O'Reilly 出版，Fredrik Lundh 编著的 *Python Standard Library* 一书，这本书提供了对标准 Python 库中所有模块的简明实用的介绍，并包含超过 300 个带注释的脚本，由此向开发者说明了如何使用每个模块。示例的数量和质量都是这本书非常杰出的显著特点。
- 要想获得有关 Python 基础知识的参考摘要和提示，请使用 O'Reilly 出版，Mark Lutz 编著的 *Python Pocket Reference* 一书。

第2章

安装

开发者可以在大多数操作系统平台上安装 Python 的经典（CPython）、JVM（Jython）和.NET（IronPython）版本。使用适当的开发系统（CPython 使用 C 开发、Jython 使用 Java 开发、IronPython 使用.NET 开发），开发者可以从发布的源代码安装 Python。在流行的操作系统平台上，开发者还可以选择从发布的预编译二进制文件安装 Python。如果开发者的操作系统平台上已经预装了 Python，开发者可能还需要安装另一个更丰富或更好的升级版本。如果是这样，本书推荐开发者不要删除，也不要覆盖平台上已经安装的 Python 版本，只需要在第一个版本“旁边”同时安装另一个 Python 版本即可。通过这种方式，可以确保开发者不会打乱已经作为操作系统平台的一部分而安装的任何其他软件，这些软件可能非常依赖于该操作系统平台本身附带的某个特定版本的 Python。

从发布的二进制文件安装 Cpython 会更快一些，可以节省开发者在某些操作系统平台上的操作时间，如果开发者没有合适的 C 编译器，则使用二进制文件安装 CPython 是唯一可行的方法。从发布的源代码安装 CPython 可以让开发者拥有更多控制，并更具灵活性，如果开发者无法找到一个适合于操作系统平台的二进制安装文件，则从源代码安装 Python 是唯一可行的方法。即使要从二进制文件安装 Python，本书仍建议开发者下载发布的源代码，因为其中包含一些示例和演示程序，而在预编译的二进制安装包中可能会缺少这些源代码。

2.1 从源代码安装 Python

要想从源代码安装 CPython，开发者需要一个包含 ISO 兼容的 C 编译器和辅助工具（比如 make）的操作系统平台。在 Windows 操作系统下，编译 Python 最常用的方法就是使用 Microsoft Visual Studio（版本为 7.1，也就是 VS2003，可以用于 Python 2.4 和 Python 2.5）。

要想下载 Python 源代码，可以访问 http://www.python.org，然后进入标记为 Download 的链接。本书编写时最新的 Python 版本是：http://www.python.org/ftp/python/2.4.3/Python-2.4.3.tgz。

.tgz 文件扩展名等同于.tar.gz（也就是，使用强大和流行的 gzip 压缩工具压缩的 tar 存档文件）。如果开发者可以处理 Bzip-2 压缩文件（现在大多数流行的解压缩工具都可以处理这种压缩文件），开发者还可以下载扩展名为.tar.bz2 而不是.tgz 的版本，这是使用更强大的 bzip2 压缩工具压缩的。

要想下载 Python 2.5 的源代码，可以访问 http://www.python.org/download/releases/2.5/。在相同的 URL 下，开发者还将看到 Python 2.5 的文档和二进制发布文件。在编写本书的时候，刚刚出现 Python 2.5 的第一个 alpha 版本，但是在读者看到本书的时候，Python 2.5 的最终版本应该已经可以使用了。

Windows

在 Windows 操作系统下，从源代码安装 Python 是非常麻烦的，除非开发者已经熟悉了 Microsoft Visual Studio 集成开发环境，并且经常在 Windows 命令行方式下工作（也就是说，在不同的 Windows 版本中被称为“MS-DOS 提示符”或“命令行提示符”的面向文本的窗口中工作）。

如果开发者觉得下面的安装指令很麻烦，建议跳过本节到第 2.2 节。无论如何，从二进制文件安装 Python 都是一个好主意，即使开发者还从源代码安装了 Python。这样，如果开发者在使用从源代码安装的版本时发现了任何奇怪的事情，都可以通过二进制安装的 Python 进行再次确认。如果在使用二进制文件安装的 Python 中奇怪的事情消失了，那肯定是因为从源代码安装的 Python 出现了问题，然后，开发者就知道应该仔细检查从源代码安装的 Python 了。

在下面的几节中，为了更清楚地介绍，假定开发者已经新建了一个名为 C:\Py 的目录，并且该目录下包含下载的 Python-2.4.3.tgz 文件。当然，开发者也可以选择适合于自己使用的目录名称和目录位置。

解压缩和解包 Python 源代码

开发者可以使用 tar 和 gunzip 程序来解包和解压缩一个.tgz 文件。如果开发者没有 tar 和 gunzip 程序，可以将 ftp://ftp.objectcentral.com/winutils.zip 上的工具集 winutils.zip 下载到 C:\Py 目录下。如果开发者没有方法解压缩 ZIP 文件，可以将 ftp://ftp.th-soft.com/UNZIP.EXE 文件下载到 C:\Py 目录下。打开“MS-DOS 提示符”窗口，并输入以下命令：

```
C:\> My Documents> cd \Py
C:\Py> unzip winutils
    [unzip 将列出其正在解包的文件——此处省略]
```

```
C:\Py> gunzip Python-2.4.3.tgz
C:\Py> tar xvf Python-2.4.3.tar
    [tar 将列出其正在解包的文件——此处省略]
C:\Py>
```

许多商用程序也可以用来解压缩和解包.tgz 存档文件（也可以处理.tar.bz2 文件），比如 WinZip（http://www.winzip.com)和 PowerArchiver (http://www.powerarchiver.com)。不管使用 gunzip 和 tar、商用程序，还是其他程序，现在开发者的计算机上应该有了一个目录 C:\Py\Python-2.4.3，这个目录是包含以源代码形式发布的整个标准 Python 版本的目录树的根目录。

使用 Microsoft Visual Studio 2003 编译 Python 源代码

使用 Microsoft Visual Studio 打开工程文件 C:\Py\Python-2.4.3\PCbuild\pcbuild.dsw，例如，打开“Windows 资源管理器”，进入目录 C:\Py\Python-2.4.3\PCbuild，然后双击 pcbuild.dsw 文件。

选择 Build→Set Active Configuration→pythonWin32 Release，然后选择 Build→Build python.exe。Visual Studio 将编译 pythoncore 和 python 这两个工程，在 C:\Py\ Python-2.4.3\PCbuild 目录下生成 python24.dll 和 python.exe 文件。开发者还可以编译另一个子工程（例如，使用 Build→Batch Build)。要想编译 subprojects _tkinter、bsddb、pyexpat 和 zlib，首先需要下载另外一些开源软件包，并将其安装在 C:\Py 目录下。按照 C:\Py\Python-2.4.3\PCbuild\readme.txt 文件中的指令编译发布的源代码中的所有 Python 软件包。

编译调试版本的 Python

开发者还可以选择编译 Python 软件包的调试版本，就像编译发布版本一样。

使用 Visual Studio，用于发布的可执行文件（.exe）只能与同样也是用于发布的动态加载库（DLL）文件完全互操作，而用于调试的可执行文件也只能与同样用于调试的 DLL 文件完全互操作。试图混和并随意匹配发布版本和调试版本文件的使用可能会导致程序崩溃和各种异常的发生。为了帮助开发者避免意外混和使用用于发布的文件和用于调试的其他文件，Python 工程在调试用可执行文件和 DLL 文件名称的后面添加了“_d”标识。例如，在编译用于调试的文件时，项目 pythoncore 将生成 python24_d.dll 文件，而项目 python 将生成 python24_d.exe 文件。

使得调试用和发布用 Visual Studio 编译文件不兼容的原因就是 C 运行时库的选择。可执行文件和 DLL 文件只有在使用相同的 C 运行时库时才能完全互操作，并且该运行时库必须也是一个 DLL 文件。可以选择“项目”→“设置”→“C/C++”→“代码生成”→“使用运行时库”，将所有项目设置为使用“多线程 DLL（MSVCRT.DLL)”（还需要删除“C/C++”→“代码生成”→“预处理器”中的_DEBUG 定义)。本书建议只有在开

发者对 Microsoft Visual Studio 非常有经验，并且有特殊的和高级需求的时候，才能按照上面的方法操作。否则，请保持发布版本和调试版本在两个相互独立和可区别的“空间”，这是到目前为止在 Windows 上使用的最简单方法。

编译后安装 Python

Python24.dll（或者是 python24_d.dll，如果开发者想要运行一个调试模式的 python_d.exe）必须被放置在一个适当的目录下，也就是说，Windows 可以在需要时从这个目录加载 DLL 文件。合适的目录取决于 Windows 的版本；例如，c:\windows\system 就是这样的目录之一。如果开发者没有将 python24.dll 复制到这样一个合适的目录中，则只有在当前目录中包含 python24.dll 文件时，才能当前目录下运行 Python。

同样地，python.exe 必须包含在 Windows 可以自动查找可执行文件的目录路径下，通常就是在 Windows 的 PATH 环境变量列出的一个目录中。如何设置 PATH 和其他环境变量取决于开发者的 Windows 版本，请参见本书第 3.1 节。Python 可以根据各种策略定位其他文件，比如标准库模块。请参见 Python 安装目录下的 C:\Py\Python-2.4.3\PC\readme.txt 文档了解各种可行的方案。

为 Cygwin 编译 Python

Python 2.4 还可以作为用于 Windows 的免费 Cygwin 类 UNIX 环境的一部分（要想了解更多信息，请参见 http://cygwin.com/）。Cygwin 运行在 Windows 之上。但是，Cygwin 在许多方面都非常类似于 Linux 和其他免费的类 UNIX 环境。特别是，Cygwin 使用了流行的免费 gcc C/C++编译器和相关工具，比如 make。因此，在 Cygwin 上从源代码编译 Python 与在类 UNIX 环境上从源代码编译 Python 相似，尽管 Cygwin 是运行在 Windows 之上的。

类 UNIX 平台

在类 UNIX 平台上，从源代码安装 Python 通常是很简单的。在下面几节中，为了更清楚地介绍，本书假定开发者已经创建了一个名为～/Py 的新目录并在该目录下下载了 Python-2.4.3.tgz 文件。当然，开发者也可以选择最适合自己使用的目录名称和目录位置。

解压缩和解包 Python 源代码

开发者可以使用 tar 和 gunzip 程序解压缩和解包.tgz 文件。如果已经有了最流行的 GNU 版本的 tar 程序，则只需要在 shell 提示符下输入以下命令：

```
$ cd ～/Py
$ tar xzf Python-2.4.3.tgz
```

与此类似，如果开发者选择下载更小的.tar.bz2 文件，仍然可以使用 GNU 版本的 tar 程序输入以下命令进行解包：

```
$ tar xjf Python-2.4.3.tar.bz2
```

不管使用哪种方式进行解包，现在都应该看到了一个目录～/Py/Python-2.4.3，这个目录是包含以源代码形式出现的整个标准 Python 发布目录树的根目录。

配置、编译和测试

开发者可以在～/Py/Python-2.4.3/README 文件的“Build instructions”标题下找到详细注解，同时，本书强烈建议读者阅读这些注解。不过，在最简单的情况下，开发者开始要做的所有事情就是在 shell 提示符下输入以下命令：

```
$ cd ～/Py/Python-2.4.3
$ ./configure
    [configure 将会打印许多信息，这里忽略这些信息]
$ make
    [make 需要花费一些时间，并打印许多信息]
```

如果没有运行./configure 就运行了 make，则 make 将隐式运行./configure。在 make 结束后，必须输入以下命令测试刚编译的 Python 是否可以正常工作：

```
$ make test
    [需要花费一些时间，并打印许多信息]
```

最有可能的情况是，make test 将确认编译结果是可用的，同时还将提示开发者，因为缺少一些可选模块，有些测试被跳过了。

因为有些模块用于专用平台（例如，有些只能在运行 SGI 的 Irix 操作系统的计算机上工作的模块），如果开发者的计算机并不支持这些模块，请不要为此担心。不过，在编译过程中跳过的其他一些模块的原因是，这些模块依赖于其他开源软件包，而开发者的计算机上没有安装这些软件包。例如，_tkinter 模块需要运行本书第 17 章介绍的 Tkinter GUI 软件包，还需要运行 IDLE 集成开发环境，IDLE 集成开发环境是 Python 附带的，只有在./configure 命令可以在开发者的计算机上找到 Tcl/Tk 8.0 或更新版本的安装时才能编译 IDLE。请参见～/Py/Python-2.4.3/README 了解更多详细信息，以及有关各种不同 UNIX 和类 UNIX 平台的特殊说明。

从源代码编译 Python 可以帮助开发者以几种非常有用的方式对 Python 进行配置。例如，在开发 C 语言编码的 Python 扩展时，开发者可以以一种特殊的方式编译 Python，这种方式可以帮助开发者跟踪内存泄漏，参见第 25.1 节。需要再次说明的是，～/Py/Python-2.4.3/README 是一个很好的信息源，其中包含有关开发者可以使用的配置选项的介绍。

编译后安装 Python

在默认情况下，./configure 将把 Python 安装到/usr/local/bin 和/usr/local/lib 目录下。开发者可以在运行 make 之前通过运行带--prefix 选项的./configure 来更改这个设置。例如，

如果想要在开发者的主目录下的子目录 py24 中安装私人使用的 Python，可以运行：

```
$ cd ~/Py/Python-2.4.3
$ ./configure --prefix=~/py24
```

然后按照上一节的说明接着运行 make。在完成了 Python 的编译和测试之后，可以执行所有文件的实际安装，运行：

```
$ make install
```

运行 make install 的用户必须拥有目标目录的写权限。根据选择的目标目录和这个目录的权限设置，在运行 make install 时，开发者可能需要使用 su 命令切换到 root、bin 或其他类型的特殊用户上。要想这样做，一个常用的方法就是使用 sudo make install 命令：如果 sudo 提示输入密码，可以输入当前用户的密码，而不是 root 用户的密码。

2.2 从二进制文件安装 Python

如果开发者使用的平台是当前流行的，应该会找到可以直接安装的预编译和打包好 Python 二进制安装文件。二进制软件包通常都是可以自安装的，既可以直接作为可执行程序安装，也可以通过相应的系统工具安装，比如 Linux 上的 RedHat Package Manager（RPM）和 Windows 上的 Microsoft Installer（MSI）。在开发者下载了软件包之后，可以运行该程序并交互式选择安装参数（例如，用来安装 Python 的目录）来安装 Python。

要想下载 Python 二进制文件，请访问 http://www.python.org 并进入标记为 Download 的链接。在编写本书的时候，主 Python 网站上直接可用的二进制安装文件是 Windows Installer（MSI）软件包：http://www.python.org/ftp/python/2.4.3/Python-2.4.3.msi 和适合于 Mac OS X 10.3.9 及其以后版本，并在 PowerPC 或 Intel 处理器（全球规格）运行的 Mac OS X Disk Image（.dmg）软件包：http://www.python.org/ftp/python/2.4.3/ Universal-MacPython-2.4.3.dmg。许多第三方组织还提供了可以用于其他平台的免费二进制 Python 安装软件。要想获得有关 Linux 的软件发布，可以参见 http://rpmfind.net 上基于 RPM（RedHat、Fedora、Mandriva、SUSE 等）的软件发布，对于 Debian 和 Ubuntu，可以参见 http://www.debian.org。网站 http://www.python.org/download/上提供了可以用于 OS/2、Amiga、RISC OS、QNX、VxWorks、IBM AS/400、Sony PlayStation 2、Sharp Zaurus 和 Windows CE（也被称为“Pocket PC”）等操作系统的二进制发布的链接。从 1.5.2 版本开始的一些老的 Python 版本也是可以使用的，尽管不如当前的 Python 2.4.3 版本那么强大和完美。该网站的下载页面上还提供了到 1.5.2 版本，以及用于老版本或不太流行的平台（MS-DOS、Windows 3.1、Psion、BeOS 等）的其他安装程序的链接。

要想获得可以在 Nokia S60 系列手机上使用的 Python 版本，可以参见 http://www.forum.nokia.com/python。

ActivePython（http://www.activestate.com/Products/ActivePython）是 Python 2.4 的一个二

进制软件包，包含几个第三方扩展模块，可以用于 AIX、HP-UX、Linux（只支持 x86 处理器）、Mac OS X、Solaris（SPARC、x64 和 x86 处理器）和 Windows（从 Windows 95 到 Windows XP 和 Windows Server 2003 的所有版本）。

Enthought（http://www.enthought.com/python/）提供了一个非常巨大而且丰富的二进制发布软件，其中包含 Python 本身（在编写本书的时候，其包含的稳定版本是 Python 2.3），以及特别适合于（但不仅限于）科学计算的大量预编译、测试和集成的扩展软件包和工具。在编写本书的时候，Enthought 只有可以用于 Windows 平台的自安装 EXE 文件，但是有计划为 Mac OS X 开发相似的软件包，非正式地称之为 MacEnthon。

Apple Macintosh

Apple 公司的 Mac OS X, 10.3（“Panther”）及其以后版本附带了 Python 2.3（只支持文本模式）。但是，本书强烈推荐开发者按照 http://www.python.org/download/releases/2.4.3/ 上的指示和链接安装 Python 的最新版本和增强功能；由于 Apple 的发布周期比较长，Mac OS 中包含的 Python 版本通常都是有些过时的，并且缺少一些功能，比如 bsddb 和 readline。Python 的最新版本安装了除 Apple 附带的 Python 支持的功能之外的更多功能，而不仅仅是相互替代的功能；Apple 使用了自己的 Python 版本和私有扩展模块，从而实现了将某些软件发布为 Mac OS X 的一部分，而冒险发布这样一个版本是不明智的。

2.3 安装 Jython

要想安装 Jython，需要开发者的 Java 虚拟机（Java Virtual Machine，JVM）兼容 Java 1.1 或更高版本。请参见 http://www.jython.org/platform.html 了解有关适用于开发者平台的 JVM 的说明。

要想下载 Jython，请访问 http://www.jython.org 并进入标记为 Download 的链接。在编写本书的时候，Jython 的最新版本（支持某些 Python 2.3 功能，并支持所有 Python 2.2 功能）是 http://prdownloads.sf.net/jython/jython-22.class。

在接下来的部分，为了更清楚地介绍，本书假定开发者已经创建了一个名为 C:\Jy 的新目录，并在该目录下下载了 jython-22.class。当然，开发者可以选择最适合自己的目录名称和位置。特别地，在类 UNIX 平台上，这个目录的名称可能会显示为~/Jy。

Jython 安装程序.class 文件是一个自安装程序。打开 MS-DOS 提示符窗口（或者类 UNIX 平台上的 shell 提示符），转移到目录 C:\Jy，并对 Jython 安装程序运行 Java 解释器。请确信目录 C:\Jy 包含在 Java 的 CLASSPATH 环境变量中。例如，在 Sun 公司发布的大多数 Java 开发工具包（Java Development Kit，JDK）上，可以运行：

```
C:\Jy> java -cp . jython-22
```

这个命令将会运行一个 GUI 安装程序，该程序让开发者可以选择目的目录和选项。如果开发者不想使用 GUI，可以在命令行中使用-o 选项开关。这个选项开关可以让开发者在命令行中指定安装目录和选项。例如：

```
C:\Jy> java -cp . jython-22 -o C:\Jython-2.2 demo lib source
```

这个命令可以在 C:\Jython-2.2 目录下安装带有所有可选组件（示例、库和源代码）的 Jython。Jython 安装程序将会创建两个很小，但是很有用的命令文件。第一个命令文件是 jython（在 Windows 下是 jython.bat），这个文件将会运行 Java 解释器。另一个命令文件是 jythonc，这个文件将 Python 源代码编译为 JVM 字节代码。开发者可以将 Jython 安装目录添加到操作系统的 PATH 环境变量中，或者将这些命令文件复制到操作系统 PATH 环境变量中的任何目录中。

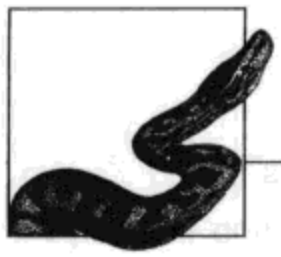
开发者可能想要在相同计算机上的不同 JDK 下使用 Jython。例如，尽管 JDK1.5 最适合于大多数的软件开发，但是开发者可能还需要偶尔使用 JDK 1.1 来编译一些可以在只支持 Java 1.1 的浏览器上运行的 Java 小程序。在这样的情况下，开发者可以在多个 JVM 之间共享单个 Jython 安装。不过，为了避免冲突和意外，本书建议开发者根据想要支持的每个 JVM，对下载的相同 Jython 安装程序执行独立的安装。例如，假定开发者将 JDK 1.5 安装在 C:\Jdk15 目录下，而将 JDK 1.1 安装在 C:\Jdk11 目录下。在这种情况下，可以使用以下命令：

```
C:\Jy> \Jdk15\java -cp . jython-22 -o C:\Jy22-15 demo lib source
C:\Jy> \Jdk11\java -cp . jython-22 -o C:\Jy22-11 demo lib source
```

在安装了这些 Jython 后，就可以选择大多数时候使用 C:\Jy22-15 目录下安装的 Jython 工作（例如，将该目录放到操作系统的 PATH 环境变量中），而在特殊情况下需要使用 JDK 1.1 编译 Java 小程序时 cd 到 C:\Jy22-11 目录下工作。

2.4 安装 IronPython

要想安装 IronPython，需要在计算机上安装当前的“公共语言运行时”（Common Language Runtime，CLR）实现。最新版本的 Mono（参见 http://www.mono-project.com/Main_Page）和 Microsoft .NET Framework 2.0 都非常适合于 IronPython。要想下载 IronPython，可以访问 http://workspaces.gotdotnet.com/ironpython（或者 http://ironpython.com，这个网站最终将成为 IronPython 的主网站，但是在编写本书的时候这个网站仍然是被废弃的）并按照该网页上的下载指示操作。在编写本书的时候，IronPython 的最新版本是 1.0。在相同的网站上还提供最新的安装指示。本书无法提供这些指示，因为在编写本书的时候这些指示还在不断变动中。



第3章 Python 解释器

为了在 Python 中开发软件系统，开发者首先需要编写包含 Python 源代码和文档的文本文件。开发者可以使用任何文本编辑器编写源文件，包括使用“集成开发环境”（Integrated Development Environment，IDE）中支持的文本编辑器。然后，可以使用 Python 编译器和解释器来处理这些源文件。开发者可以通过 IDE，或者其他嵌入了 Python 的程序在后台直接进行编译和解释。Python 解释器还可以让开发者交互式执行 Python 代码，包括在 IDE 中。

3.1 Python 程序

Python 解释器程序的文件名为 python（在 Windows 中名为 python.exe）。Python 文件包括解释器本身和 Python 编译器，在导入模块时，会在需要的时候隐式激活该编译器。根据不同的系统，Python 程序必须包含在该系统的 PATH 环境变量的目录列表中。另外，与所有其他程序一样，开发者也可以在命令（shell）提示符下，或者在 shell 脚本（或者.BAT 文件文件，快捷方式等）中给出 Python 程序的完整路径（根据不同的操作系统，如果该路径包含空格，则可能需要使用引号）。在 Windows 中，还可以使用“开始”→“程序”→“Python 2.4”→“Python（command line）”来运行 Python。

环境变量

除了 PATH，其他环境变量也会影响 Python 程序。有些环境变量与通过命令行传递给 Python 的选项具有相同的效果，第 3.1 节将对此进行介绍。一些环境变量还提供了命令行选项不能实现的设置：

PYTHONHOME

Python 安装目录。包含标准 Python 库模块的 lib 子模块必须被放在这个目录之下。

在类 UNIX 操作系统上，对于 Python 2.3，标准库模块必须被放在子目录 lib/python-2.3 下，对于 Python 2.4，则放在子目录 lib/python-2.4 下，依此类推。

PYTHONPATH

Python 的目录列表，在类 UNIX 系统中通过冒号分隔，在 Windows 系统中通过分号分隔。Python 模块都是从这些目录导入的。这个列表扩展了 Python 的 sys.path 变量的初始值。本书将在第 7 章中介绍模块、导入和 sys.path 变量。

PYTHONSTARTUP

每次启动一个交互式解释器会话时自动执行的 Python 源文件的名称。如果没有设置这个变量或者在这个变量设置的路径下找不到对应的文件，则不运行该文件。在开发者运行 Python 脚本时，不会使用这个 PYTHONSTARTUP 文件；这个文件只有在开发者启动一个新的交互式会话时才会被使用。

如何设置和检查环境变量取决于使用的操作系统：shell 命令、持久化启动 shell 文件（例如，Windows 下的 AUTOEXEC.BAT 文件），或者其他方法（例如，在 Windows XP 上运行“开始”→“设置”→“控制面板”→“经典视图”→“系统”→“高级”→“环境”）。除了环境设置，一些 Windows 上的 Python 版本还可以在“注册表”中查找这些信息。在 Macintosh 系统上，可以像在其他类 UNIX 系统上一样运行 Python 解释器，但是 Mac 系统还有其他一些选项，包括 MacPython 特定 IDE。要想获得有关 Mac 上的 Python 的信息，请参见 http://www.python.org/doc/current/mac/mac.html。

命令行语法和选项

Python 解释器命令行语法可以概括为以下命令：

```
[path]python {options} [-c command | -m module | file | -] {arguments}
```

在这个命令中，方括号（[ ]）包含可选项，花括号（{ }）包含可能会出现 0 个或多个选项，竖线符号则表示多个可替代的选项中的一个选择。

Options 使用的都是区分大小写的短字符串，以连字符（-）开始，这些选项可以指示 python 执行非默认的操作。与大多数 Windows 程序不一样，python 只接受以连字符（-）开始的选项，不接受以斜线（/）开始的选项。像在 UNIX 中一样，Python 总是使用斜线（/）来表示文件路径。表 3-1 列出了最常用的选项。每个选项的说明都给出了环境变量（如果存在的话），在设置这些选项时，可以产生与设置对应的环境变量相同的效果。

下表中的命令行选项的说明中给出了对应的环境变量（如果存在），当这些环境变量被设置为某个值时，则要求执行相应的操作。

表 3-1　Python 的常用命令行选项

选　　项	说明（以及对应的环境变量）
-c	将 Python 语句指定为命令行的一部分
-E	忽略所有环境变量
-h	打印选项和帮助摘要的完整列表，然后结束运行
-i	不管指定了什么样的（PYTHONINSPECT），都确保一个交互式会话
-m	指定一个 Python 模块作为主脚本运行
-O	优化生成的字节代码（PYTHONOPTIMIZE）
-OO	与-O 相似，并将从字节代码中删除存档字符串
-Q arg	控制整数的除法运算符/的行为
-S	忽略启动时的隐含的 import site （第 13.5 节中将介绍这个选项）
-t	在制表符（tab）和空格（blank space）的使用不一致时，产生警告消息
-tt	与-t 相似，但是将产生错误消息，而不是警告消息
-u	对标准输出和标准错误使用无缓冲的二进制文件（PYTHONUNBUFFERED）
-v	详细跟踪模块的导入和清除操作（PYTHONVERBOSE）
-V	打印 Python 的版本号，然后终止运行
-W arg	向警告筛选器添加一个条目（第 18.3 节中将介绍这个选项
-x	排除（跳过）主脚本源代码的第一行

在想要运行某些脚本之后立即进入一个交互式会话时，可以使用-i 选项，这样还可以保持变量不变，并可以进行检查。开发者不需要为正常的交互式会话使用-i 选项，尽管使用该选项也没有什么危害。-t 和-tt 可以确保 Python 源代码中制表符和空格的使用保持一致（参见第 4.1 节以了解 Python 中有关空格的更多信息）。

-O 和-OO 选项可以为导入的模块生成的字节代码节省时间和空间，特别是会将 assert 语句转变为空操作，参见第 6.6 节。使用-OO，不会再有存档字符串。-Q 决定了对两个整数操作数使用除法运算符/的行为（第 4.5 节中将介绍除法）。-W 将向警告筛选器添加一个条目（第 18.3 节中将介绍警告）。

-u 选项将对标准输出（和标准错误）使用二进制模式。有些平台，主要是 Windows，是可以区分二进制模式和文本模式的。例如在某些“公共网关接口”（Common Gateway Interface，CGI）脚本中，是需要使用二进制模式将二进制数据发送到标准输出的。-u 还可以确保立即执行输出操作，而不是进行缓存以增强性能。在缓存的延迟可能会导致出现问题的时候，这样做是非常需要的，例如在某些 UNIX 流水线操作中。

如果有的话，在上面这些选项之后，是一个指示选项，指定要运行哪个 Python 程序。这个选项是一个文件路径，也就是 Python 源文件或二进制文件运行的位置，以一个文

件扩展名（如果有的话）结束。在任何平台上，都可以使用斜线（/）符号作为路径中各个组件之间的定界符。当且仅当在 Windows 平台下，还可以选择使用反斜线（\）符号。除了文件路径之外，还可以使用-ccommand 直接执行 Python 代码字符串（command）。代码字符串通常包含空格，因此需要为该字符串加引号，以满足操作系统的 shell 或命令行处理器的使用要求。有些 shell（例如，bash）允许开发者输入多个行作为单个参数，这样的代码字符串可以是一连串的 Python 语句。而其他 shell（例如，Windows shell）将代码字符串限制为只支持单个行；在这种情况下，代码字符串可能是由分号（;）分隔的一个或多个简单语句，参见第 4.1 节中介绍的简单语句。在 Python 2.4 中，另一种方法就是使用-mmodule 来指定运行哪个 Python 程序。这个选项用来告诉 Python 从 Python 的 sys.path 中的某个目录中加载和运行一个名为 module 的模块。

连字符，或者在这个位置不指定任何标记，可以告诉 Python 解释器从标准输入，通常也就是从交互式会话中读取程序。只有在后面带参数的情况下才需要使用显式的连字符。arguments 可以是任意字符串；正在运行的 Python 应用程序可以访问作为列表 sys.argv 中的条目的字符串。

例如，在 Python 2.4 的标准 Windows 安装中，开发者可以在“MS-DOS 提示符”（也就是“命令提示符”）下输入以下命令让 Python 打印当前日期和时间：

```
C:\> c:\python24\python -c "import time; print time.asctime()"
```

在从源代码安装，并在 Cygwin、Linux、OpenBSD 或其他类 UNIX 系统上执行的 Python 默认安装中，开发者可以在 shell 提示符下输入以下命令，以启动一个带有详细导入和清理跟踪信息的交互式会话：

```
$ /usr/local/bin/python -v
```

如果 Python 可执行文件的路径被包含在系统的 PATH 环境变量中，则在以上任何一种情况下，开发者只需要使用 python 字符串（而不需要指定 Python 可执行文件的完整路径）来运行命令即可。

交互式会话

在运行不带脚本参数的 python 时，python 将进入一个交互式会话（session），并提示开发者输入 Python 语句或表达式。对于试验 Python 的功能、验证基本操作和将 Python 作为一个强大的、可扩展的交互式计算器而言，Python 交互式会话是非常有用的。

在输入一个完整的语句之后，Python 将执行这条语句。在输入一个完整的表达式之后，Python 将计算这个表达式。如果这个表达式有输出结果，则 Python 将输出一个显示了其结果的字符串，并将结果赋值给一个名为“_”（单个下划线）的变量，这样就可以很容易地在另一个表达式中使用这个结果了。在开始向 Python 输入语句或表达式时，显示的提示符字符串是>>>，而在已经开始输入一个语句或表达式，但是还没有输入完全时，出现的提示符字符串是…。例如，在前一行使用了一个圆括号，但是这一行没

有输入反括号时，Python 将会提示…。

开发者可以使用标准输入（Windows 操作系统下是 Ctrl-Z，类 UNIX 系统下是 Ctrl-D）中的文件尾来终止交互式会话。不管是在交互式会话中，还是在正在运行的代码中，使用 raise SystemExit 语句也可以终止会话，就像调用了 sys.exit()一样（第 6 章中将介绍 SystemExit 和 Python 异常处理）。

行编辑和历史功能取决于 Python 是如何编译的：如果包含了可选的 readline 模块，则可以使用 GNU readline 库的功能。Windows NT、2000 和 XP 都带有一个简单但是实用的历史工具，可供像 python 这样的交互式文本模式的程序使用。而 Windows 95、Windows 98 和 ME 则没有。开发者可以在 Windows 下安装 Alternative Readline 软件包（http://newcenturycomputers.net/projects/readline.html），或者在 UNIX 下安装 pyrepl（http://starship.python.net/crew/mwh/hacks/pyrepl.html）。

除了 Python 内置的交互式环境，以及第 3.2 节中将要介绍的一部分功能更丰富的开发环境，开发者还可以免费下载其他一些功能强大的交互式环境。最流行的一个是 IPython（http://ipython.scipy.org/），这个开发环境提供了非常丰富的功能。

3.2 Python 开发环境

Python 解释器的内置交互式模式是 Python 最简单的开发环境。这个开发环境有点原始，但是是轻量级的，具有较小的尺寸，并且启动很快。结合适当的文本编辑器（参见第 3.2 节中的介绍），以及行编辑和历史功能，这个交互式解释器（或者还可以选择 IPython）提供了可用的和流行的开发环境。不过，还有许多其他的开发环境可供开发者使用。

IDLE

标准 Python 软件发布版本中包含了 Python 的集成开发环境（Integrated DeveLopment Environment，IDLE）。IDLE 是跨平台的，是基于 Tkinter（参见第 17 章）的 100%的纯 Python 应用程序。IDLE 提供了一个类似于交互式 Python 解释器会话，但是功能上更强大的 Python shell。IDLE 还包含一个优化的文本编辑器用来编辑 Python 源代码、一个集成的交互式调试器和几个特定的浏览器/查看器。

其他免费的跨平台 Python IDE

IDLE 是成熟、稳定、容易使用，并且功能非常丰富的。不过，还有几个非常有前景的新 Python IDE 正在出现，他们共享了 IDLE 的免费和跨平台的自然特性。RedHat 的 Source Navigator（http://sources.redhat.com/sourcenav/）支持多种语言，可以在 Linux、Solaris、HPUX 和 Windows 下运行。BoaConstructor（http://boa-constructor.sf.net/）只支持 Python 语言，并且还是 beta 版本，但是非常值得开发者尝试使用。Boa Constructor 包含一个 wxWindows 跨平台 GUI 工具包中的 GUI 编译器。

eric3（http://www.die-offenbachs.de/detlev/eric3.html）是一个基于 PyQt 3.1 跨平台 GUI 工具包的全功能 Python 和 Ruby IDE。

流行的跨平台和跨语言模块化 IDE Eclipse 包含支持 CPython 和 Jython 的插件；请访问 http://pydev.sourceforge.net/ 以了解更多信息。

另一个很新但是非常流行的跨平台 Python 编辑器和 IDE 是 SPE，也就是“Stani's Python Editor”（参见 http://stani.be/python/spe/blog/）。

平台专用免费 Python IDE

Python 是跨平台的，而本书主要关注跨平台工具和组件。不过，Python 还为其支持的许多平台提供了很好的平台专用功能，包括 IDE。特别是，在 Windows 下，ActivePython 包含 PythonWin IDE。PythonWin 还可以作为 Windows 下的标准 Python 发布软件的免费扩展模块，也就是 MarkHammond 的 win32all 扩展模块的一部分（参见 http://starship.python.net/crew/mhammond）。

商用 Python IDE

还有几家公司销售商用 PythonIDE，包括跨平台和专用平台 IDE。如果开发者使用这些 IDE 进行商用软件开发，甚至在大多数情况下，即使开发免费软件，也必须为这些 IDE 支付费用。不过，这些商用 Python IDE 可以提供技术支持合同和丰富的工具。如果开发者有经费购买软件工具，值得详细考察这些软件工具，并尝试商用这些工具的免费测试和评估版本。这些工具大多数都可以在 Linux 和 Windows 下使用。

Archaeopterix 销售的一款优秀的 Python IDE 工具是 Wing，这个工具特别值得关注的是其强大的源代码浏览和远程调试工具（http://wingware.com）。theKompany 销售的一款 Python IDE 工具是 BlackAdder，这个工具还包含一个用于 PyQtGUI 工具包（http://www.thekompany.com/products/blackadder）的 GUI 编译器。

ActiveState（http://www.activestate.com）销售的 Python IDE 工具是 Komodo，这个工具是在 Mozilla（http://www.mozilla.org）之上编译的，并包含远程调试功能。

支持 Python 的免费文本编辑器

开发者可以使用任何文本编辑器编辑 Python 源代码，即使是最简单的编辑器也可以，比如 Windows 下的“记事本”或 Linux 下的 ed。一些强大的免费编辑器还支持 Python，并带有一些附加的功能，比如按语法着色和自动缩进。跨平台编辑器可以让开发者在不同平台下以相同的方式工作。一些优秀的程序员的文本编辑器还可以让开发者从该编辑器上运行对正在编辑的源代码选择的工具。开发者总是可以在 http://wiki.python.org/moin/PythonEditors 上找到 Python 编辑器的最新列表。

要想使用最纯粹的编辑功能，最好的工具就是经典的 Emacs（要想了解 Python 专用的附加功能，请参见 http://www.emacs.org 和 http://www.python.org/emacs）。不过，Emacs 并不是最容易学习的编辑器，也不是轻量级的。我个人最喜欢的是另一个经典的编辑器 vim（http://www.vim.org），是传统 UNIX 编辑器 vi 的现代改进版本，vi 并不像 Emacs 那样具有强大的功能，但是仍然是非常值得考虑的。与 vi 相同，vim 进行了形式上的设计，这种设计可以让开发者在命令模式下使用一些普通按键进行光标移动和文本更改。有些人很喜欢 vim，认为它符合人体工程学原理，可以最小化手指的行程。而其他一些人则发现这样做容易混淆，因此讨厌这样做。一些更新的编辑器向那些经典的编辑器提出了挑战。SciTE（http://www.scintilla.org）建立在 Scintilla 编程语言编辑器组件之上。FTE（http://fte.sf.net）也是非常值得尝试的。

其他高级的，支持 Python 语法的免费编辑器都是专用平台的。在 Windows 系统下，可以尝试 SynEdit（http://www.mkidesign.com/syneditinfo.html）。在类 UNIX 系统下，可以尝试 Glimmer（http://glimmer.sf.net）和 Cooledit（http://freshmeat.net/projects/cooledit/），这些编辑器与 vim 相似，也提供了 Python 的可编程性，但是不具备 vim 的模式架构。在 Mac OS X 系统下，TextWrangler（http://www.barebones.com/products/textwrangler/index.shtml）是非常强大的，也能很好地支持 Python。SubEthaEdit（http://www.codingmonkeys.de/subethaedit/）对于非商业用途是免费的，而对于商业用途是需要付费的（非常便宜），这个编辑器具有非常独特的功能并进行了优化，可以让多个程序员在相同局域网中的不同 Mac 计算机上同时编辑相同的文件以进行合作开发。

开发者可以在 http://wiki.python.org/moin/PythonEditors 上找到特别适合于 Python 的编辑器（免费的和付费的），还包括 IDE 的大量汇总信息。

Python 程序检查工具

Python 编译器并不能全面彻底地检查程序和模块：编译器只检查代码的语法。如果开发者想要更全面地检查 Python 代码，可以通过下载和安装以下几个工具来实现。PyChecker（可以从 http://pychecker.sourceforge.net/ 获得）是很容易安装和使用的：这个工具通过标准 Python 编译器将 Python 源代码转换为字节代码，然后导入该字节代码并检查所有代码以查找多种类型的错误和异常。Pyflakes（可以从 http://divmod.org/projects/pyflakes 获得）比 PyChecker 更快，尽管不如 PyChecker 全面，但是 Pyflakes 不需要导入要检查的模块，这样做使得 Pyflakes 的使用更安全。PyLint（可以从 http://www.logilab.org/projects/pylint 获得）的功能非常强大，并且具有很高的可配置性。PyLint 并不是轻量级的，也不像 PyChecker 和 Pyflakes 的安装那样简单，因为 PyLint 要求安装其他一些可以从 Logilab 免费下载的软件包；不过，PyLint 可以根据自定义配置文件以一种高可配置的方式检查多种类型的错误和异常，这样可以充分回报其复杂的安装和使用过程。

3.3 运行 Python 程序

不管使用什么工具创建 Python 应用程序，开发者都可以将该应用程序看作是一组 Python 源代码，这些源代码都是普通文本文件。脚本（Script）是开发者可以直接运行的文件。模块（Module）是开发者可以导入（参见第 7 章）以向其他文件或交互式会话提供功能性的文件。一个 Python 文件可以既是模块，又是脚本，并在导入的时候显示其功能性，但是也适合直接运行。一个非常有用并广泛使用的习惯用法就是，对于那些最初打算被导入为模块的 Python 文件，在直接运行时，必须执行一些简单的自我测试操作，第 18.1 节对此进行了介绍。

Python 解释器将根据需要自动地编译 Python 源文件。Python 源文件通常都有一个扩展名.py。Python 可以将每个模块的编译好的字节代码文件保存在相同的目录中，将这些文件作为该模块的源文件，并将其保存为相同的文件名称和扩展名.pyc（如果运行 Python 的时候包含了选项-O，则扩展名为.pyo）。在开发者直接运行脚本时，Python 并不会保存为一个脚本编译好的字节代码文件；相反，Python 将在每次运行该脚本的时候重新编译这个脚本。Python 只为导入的模块保存字节代码文件。只要有需要，Python 将自动重新编译每个模块的字节代码文件，例如，在开发者编辑模块的源代码时。最后，为了进行部署，开发者需要使用第 27 章中介绍的工具打包 Python 模块。

开发者可以使用 Python 解释器或集成开发环境交互式运行 Python 代码。不过，通常开发者需要运行一个顶级的脚本来初始化执行操作。要想运行一个脚本，可以按照第 3.1 节中介绍的方法将脚本的路径指定为 python 程序的一个参数。根据不同的操作系统，开发者可以从 shell 脚本或命令文件中直接运行 python。在类 UNIX 系统中，可以通过设置文件的权限比特 x 和 r 使得 Python 脚本可以直接执行，并让该脚本文件以一个名为 shebang 的行开始，也就是脚本文件的第一行文本，例如：

```
#!/usr/bin/env python {options}
```

或者是其他以#!开始，后面跟一个 python 解释器程序的路径的命令行。

在 Windows 下，开发者可以在 Windows 注册表中将扩展名为.py、.pyc 和.pyo 的文件与 Python 解释器进行关联。大多数用于 Windows 操作系统的 Python 版本都可以在安装的时候执行这种关联。然后，开发者就可以按照常规的 Windows 机制来运行 Python 脚本了，比如双击这些文件的图标。在 Windows 中，当开发者双击脚本的图标运行 Python 脚本时，Windows 将在脚本运行结束之后立即自动关闭与该脚本关联的文本模式的控制台。如果想要让控制台停留更长时间，以允许开发者阅读该脚本在屏幕上的输出结果，则必须保证该脚本不要太快就运行结束。例如，使用下面一行代码作为该脚本的最后一行语句。

```
raw_input('Press Enter to terminate')
```

在开发者从已有的控制台（也被称为命令提示窗口）上运行脚本时，是不需要这样做的。

在 Windows 操作系统下，开发者还可以使用扩展名.pyw 和解释器程序 pythonw.exe，而不是.py 和 python.exe。Python 的 w 变体可以运行不带文本模式控制台的 Python，因此并不会产生标准输入和输出。这些变体适合于依赖于 GUI 或者在后台运行不可见的脚本。只有在一个程序经过了完整的调试，并且要保留在开发过程中对一些信息、警告和错误消息可用的标准输出和错误时才使用这个解释器程序。在 Mac 操作系统下，在开发者想要运行一个需要访问任何 GUI 工具包，而不仅仅是文本模式的交互操作时，需要使用解释器程序 pythonw，而不是 python。

使用其他语言编写的应用程序可以嵌入 Python，这些代码可以根据自己的目的控制 Python 代码的执行。本书将在第 25.3 节中进一步讨论这个话题。

3.4 Jython 解释器

在安装过程中编译的 jython 解释器（参见第 2.3 节）的运行类似于 python 程序：

```
[path]jython {options} [ -j jar | -c command | file | - ] {arguments}
```

-j jar 选项告知 jython 要运行的主要脚本是.jar 文件中的_ _run_ _.py。选项-i、-S 和-v 与 Python 的选项相同。--help 选项与 python 的-h 选项相同，--version 选项与 python 的--V 选项相同。jython 使用安装目录中的一个名为 registry 的文本文件以结构化名称的方式记录各种属性，而不是使用环境变量。例如，Jython 中的属性 python.path 相当于 Python 的环境变量 PYTHONPATH。开发者还可以以-D name=value 的形式使用 jython 命令行选项来设置属性。

3.5 IronPython 解释器

IronPython 的运行与 python 程序相似：

```
[path]IronPythonConsole {options} [-c command | file | - ] {arguments}
```

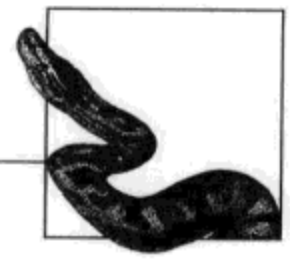
不幸地是，在编写本书的时候，这些参数的细节仍在不断变动中，因此本书无法提供这些信息。请参见 http://ironpython.com 以了解最新的信息。

PDG

第 2 部分

核心 Python 语言和内置对象

第 4 章



Python 语言

本章是 Python 语言的快速指南。要想从头开始学习 Python，本书建议读者从 O'Reilly 出版，Mark Lutz 和 David Ascher 编著的 *Learning Python* 一书开始。如果读者已经对其他编程语言有所了解，而只是想要学习 Python 的一些不同的特殊功能，本章是非常适合的。不过，本章并不打算教大家如何使用 Python，因此本章将非常快速地介绍许多基础知识。本章将主要介绍一些规则，其次会指出一些最好的经验和推荐的编程风格；要想了解标准的 Python 编程风格，请参见 http://python.org/doc/peps/pep-0008/。

4.1 词法结构

编程语言的词法结构（lexical structure）就是一组基本规则，这些规则可以规范开发者如何用这种语言编写程序。词法结构是语言的最低一级语法，同时指定了变量名是什么样子，哪些字符表示注释这样一些信息。与所有其他文本文件一样，每个 Python 源文件都是一个字符序列。开发者还可以将 Python 源文件看成由代码行、语言符号（token）或语句组成的序列。这些不同的对于词法的观点可以相互补充和完善。Python 在程序布局上是非常特别的，尤其是代码的行和缩进，因此，如果开发者是从其他语言转到 Python 的，需要注意这个信息。

行和缩进

Python 程序是由逻辑行（logical line）序列组成的，每个逻辑行由一个或多个物理行（physical line）组成。每个物理行的结尾可以是注释。没有被包含在一个字符串字面常量内的#号用于开始一段注释。#号之后到物理行结尾为止的所有字符都是注释部分，Python 解释器将忽略注释内容。一个只包含空格或者注释的行被称为空（blank）行，Python 将完全忽略这一行代码。在交互式解释器会话中，开发者必须输入一个空的物理行（不带任何空格或注释）以终止一个多行语句。

在 Python 中，物理行的末尾也就表示大多数语句的末尾。与其他的语言不一样，开发者通常不需要使用定界符来终止 Python 语句，比如分号（;）。如果一条语句太长，以至于单个物理行无法容纳时，可以将两个相邻的物理行连接成一个逻辑行，这需要确保第一个物理行没有注释，并且该行以反斜线（\）结束。不过，如果一个左边的圆括号（(）、方括号（[）或花括号（{）还没有对应的右括号，则 Python 将自动把多个物理行连接成一个逻辑行，利用这种机制，通常可以产生更具可读性的代码，而不是显式地在物理行的末尾插入反斜线符号。三重引用字符串字面常量也可以跨越物理行。逻辑行中第一行后面的物理行被称为后续行（continuation lines）。下面将要介绍的缩进问题并不会应用到后续行上，而只是应用到每个逻辑行的第一个物理行上。

Python 使用缩进（indentation）来表示程序的块结构。与其他语言不一样，Python 没有使用花括号或者其他开始/结束定界符来包含语句块；缩进是标识每个语句块的唯一方法。Python 程序中的每个逻辑行都在其左侧缩进了空格。一个块就是一个连续的逻辑行序列。块中的所有语句必须具有相同的缩进，因为所有的子句都必须在一个复合语句中。源文件中的第一个语句是不需要缩进的（例如，不允许以任何空格开始）。开发者在交互式解释器主提示符>>>（参见第 3.1 节）输入的语句也不需要缩进。

Python 可以从逻辑上将每个制表符（tab）替换为最多 8 个空格，因此制表符之后的下一个字符的逻辑列号为 9、17、25 等。标准 Python 风格是每个缩进级别使用 4 个空格（永远都不要使用制表符）。请不要混淆空格和制表符缩进，因为不同的工具（比如，编辑器、电子邮件系统和打印机）处理制表符的方式不同。Python 解释器的-t 和-tt 选项（参见第 3.1 节）可以确保开发者不受到 Python 源代码中制表符和空格的使用不一致而产生的影响。本书建议开发者将自己比较喜爱的文本编辑器配置为将制表符展开为空格，这样编写的所有 Python 源代码就会只包含空格，而没有制表符了。使用这种方法，所有工具，包括 Python 本身，都将在处理 Python 源文件中的缩进上取得完美的一致性。最优化的 Python 风格是精确地缩进 4 个空格。

字符集

通常，Python 源代码必须完全由 ASCII 集合（0～127 的字符代码）组成。不过，开发者可以选择告诉 Python，在特定的源文件中使用了作为 ASCII 超集的字符集（character set）。在这种情况下，Python 允许这个特定的源代码中包含 ASCII 集合之外的字符，但是这些字符只能出现在注释和字符串字面常量中。要想这么做，可以在 Python 源代码的起始部分加上一行注释，这行注释的格式必须与下面的示例保持严格一致：

```
# -*- coding: utf-8 -*-
```

在 coding 和-*-之间，输入 Python 已知的字符编码方式的名称，比如 utf-8 或 iso-8859-1。请注意，这个具有编码指令（coding directive）的注释只有在位于源文件的起始部分时（可能在第 3.3 节中介绍的 shebang 行的后面）才能起作用，并且该编码指令的唯一影响就是让开发者可以在字符串字面常量和注释中使用非 ASCII 字符。

语言符号

Python 可以将每个逻辑行分成一个由名为语言符号（token）的基本词法组件组成的序列。每个语言符号对应于逻辑行的一个子字符串。常用的语言符号类型包括标识符（identifier）、关键字（keyword）、运算符（operator）、定界符（delimiter）和字面常量（literal），下面几节将分别这几种类型。开发者可以很直接地使用语言符号之间的空格来分隔这些语言符号。逻辑相邻的标识符或关键字之间使用一些空格来分隔是必需的；否则，Python 会将其解析为单个很长的标识符。例如，printx 是一个单独的标识符；要想输入关键字 print，后面紧跟一个标识符 x，则需要插入一个空格（例如，print x）。

标识符（Identifier）

标识符（Identifier）是一个用来标识变量、函数、类、模块或其他对象的名称。标识符以字母（A～Z 或者 a～z）或者下划线（_），后面带零个或多个字母、下划线和数字（0 到 9）组成。Python 是区分大小写的：小写和大写字母是不同的。Python 不允许在标识符中使用某些标点符号，例如@、$和%。

通常情况下，Python 的风格是类名称以大写字母开始，而所有其他标识符都是小写字母。按习惯用法，以单个下划线开始的标识符表示该标识符是私有的。以两个下划线开始的标识符表示非常强的私有标识符；如果该标识符还以两个下划线结束，则表示该标识符是 Python 语言定义的特殊名称。标识符“_”（单个下划线）专门用于交互式解释器会话中：解释器将把“_”绑定到交互式计算的最后一个表达式语句的结果上，如果这个结果存在的话。

关键字（Keyword）

Python 有 30 个关键字，这些关键字都是 Python 保留的以供专门语法使用的标识符。关键字只包含小写字母。开发者不能将关键字用作常规标识符。某些关键字用来开始一些简单的语句或符合语句的子句，而其他关键字则是运算符。本书将在本章，还有第 5 章、第 6 章和第 7 章中详细介绍所有关键字。Python 中的关键字包括：

```
and       del       for       is        raise
assert    elif      from      lambda    return
break     else      global    not       try
class     except    if        or        while
continue  exec      import    pass      with (2.5)
def       finally   in        print     yield
```

标识符 with 是从 Python 2.5 开始引入的一个新关键词（到 Python 2.4 为止，这些是完整的常规标识符）。严格来说，标识符 as 并不是一个关键字，as 可以用作一部分语句（特别是，from 和 import 语句，以及 Python 2.5 中的 with 语句）的伪关键字。在 Python 2.5

中，使用 with 或 as 作为常规标识符将会产生警告。在 Python 2.5 中，要想让 with 可以用作关键字（并因此使能新的 with 语句），可以在源文件的起始部分使用以下语句：

```
from __future__ import with_statement
```

这个“import from the future”语句用来允许在这个模块中使用 with 语句。

运算符（Operator）

Python 使用一些非文字和数字的字符和字符组合作为运算符。Python 可以识别以下运算符，本书将在第 4.4 节中详细介绍这些运算符：

```
+     -     *     /     %     **     //     <<     >>     &
|     ^     ~     <     <=     >     >=     <>     !=     ==
```

定界符（Delimiter）

Python 使用以下符号和符号组合作为表达式、列表、字典、各种语句和字符串中的定界符，以及其他用途：

```
(          )          [          ]          {          }
,          :          .          '          =          ;
+=         -=         *=         /=         //=        %=
&=         |=         ^=         >>=         <<=       **=
```

句点符号（.）还可以出现在浮点字面常量（例如，2.3）和虚数字面常量（例如，2.3j）中。最后两行列出了扩展的赋值运算符，这些运算符从词法上讲是定界符，但是也可以执行相应的运算。本书将在介绍对象或语句时，再详细介绍使用的各种定界符的语法。

与部分其他语言符号一样，以下字符具有特殊涵义：

```
'          "          #          \
```

字符$和?、除空格之外的所有控制字符和 ISO 编码中高于 126 的所有字符（也就是，非 ASCII 字符，比如重音字母）永远都不能成为 Python 程序代码的一部分，除在注释或字符串字面常量之外（要想在注释或字面常量中使用非 ASCII 字符，必须按照第 4.1 节中介绍的，在 Python 源代码的起始部分使用“编码指令”语句）。这种情况也适用于 Python 2.3 中的字符“@”；不过，在 Python 2.4 中，“@”表示装饰器（decorator），参见第 5.3 节。

字面常量（Literal）

字面常量（literal）是在程序中直接显示的数字或字符串。下面是 Python 中的所有字面常量的示例：

```
42                              # 整数字面常量
3.14                            # 浮点型字面常量
```

```
1.0j                        # 虚数字面常量
'hello'                     # 字符串字面常量
"world"                     # 另一种字符串字面常量
"""Good night"""           # 三重引用字符串字面常量
```

使用字面常量和定界符，可以创建其他一些基本类型的数据值：

```
[ 42, 3.14, 'hello' ]     # 列表
( 100, 200, 300 )         # 元组
{ 'x':42, 'y':3.14 }      # 字典
```

本书将在第 4.2 节中介绍 Python 支持的各种数据类型，并详细说明字面常量和其他基本类型数据值的语法。

语句

开发者可以将 Python 源文件看作是由简单和复合语句组成的序列。与其他语言不一样，Python 没有声明或其他顶层语法元素，只有语句。

简单语句

简单语句（simple statement）是不包含其他语句的语句。一个简单语句完全包含在一个逻辑行中。在其他语言中，开发者可能可以将多于一个的简单语句放到单个逻辑行中，并使用分号（;）作为分隔符。但是，每行一条语句是 Python 最常用的风格，这样可以让程序更有可读性。

任何表达式都可以将其自身作为一个简单表达式（本书将在第 4.4 节中详细介绍表达式）。交互式解释器将会显示开发者在提示符（>>>）后输入的表达式语句的结果，并将该结果帮定到一个以“_”（单个下划线）命名的变量上。除了交互式会话，表达式语句只能用于调用具有负面影响（例如，执行输出、更改全局变量或引发异常的函数）的函数（以及其他可调用的内容）。

赋值语句是一个简单语句，可以为变量赋值，参见第 4.3 节。与其他一些语言不同，Python 中的赋值操作是一个语句，不能作为表达式的一部分。

复合语句

复合语句（compound statement）包含一个或多个其他语句，并可以控制这些语句的执行。复合语句具有一个或多个子句（clause），这些子句以相同的缩进排列。每个子句有一个以关键字开始，以冒号（:）结束的标题，然后是正文，正文就是由一个或多个语句组成的序列。当正文包含多个语句时，也被成为块（block），这些语句必须放到标题行之后的单独逻辑行中，向右缩进 4 个空格。当缩进返回到该子句的标题的缩进位置（或者到某些复杂语句更左边的缩进位置）时，块词法就结束了。另外，正文也可以是单个简单语句，作为标题直接放在相同逻辑行中的冒号（:）的后面。正文还可以由相同行上使用分号（;）分隔的几个简单语句组成，但是，正如本书已经指出的，这

并不是一种好的编程风格。

4.2 数据类型

Python 程序的运行取决于该程序要处理的数据。Python 中的所有数据值都是对象，并且每个对象或者值都有一个类型（type）。对象的类型确定了该对象支持哪些操作，或者换句话讲，确定了可以对该数据值执行哪些操作。类型还确定了该对象的属性（attribute）和项目（item）（如果存在的话），以及该对象是否可以被改变。一个可以被改变的对象被称为“可变对象”（mutable object），而不可以被改变的对象则被称为“不可变对象”（immutable object）。本书将在第 4.3 节中详细介绍对象的属性和项目。

内置的 type(obj)可以接受任何对象作为其参数，并返回 obj 类型的类型对象。如果对象 obj 具有类型 type（或者其任何子类），则内置函数 isinstance(obj, type)将返回 True；否则，该函数将返回 False。

对于一些基本数据类型，比如数字、字符串、元组、列表和字典，Python 都有内置类型，下面几节将会对这些类型进行介绍。开发者还可以创建用户自定义类型，这些类型也被称为类（class），参见第 5.1 节。

数字

Python 中的内置数字对象支持整数（普通整型和长整型）、浮点型数字和复数。在 Python 2.4 中，标准库还提供了十进制浮点型数字，参见第 15.4 节。Python 中的所有数字都是不可变对象，这意味着在对一个数字对象执行任何操作时，总是会产生一个新的数字对象。第 4.5 节将介绍数字运算，也被称为算术运算。

请注意，数字字面常量不包含符号：+或-，如果包含这两个符号，则表示该符号是一个分隔运算符，参见第 4.5 节。

整数

整数字面常量可以是十进制、八进制或十六进制。十进制字面常量。十进制字面常量可以由第一个数字为非零数字的数字序列表示。为了表示八进制字面常量，可以使用 0 后面带一个八进制数字（0～7）序列。为了表示十六进制字面常量，可以使用 0x 后面带一个十六进制数字序列（0～9 和 A～F，可以使用大写或小写字母）。例如：

```
1, 23, 3493                    # 十进制整数
01, 027, 06645                 # 八进制整数
0x1, 0x17, 0xDA5               # 十六进制整数
```

实际上，在 Python 中，开发者不需要担心普通整型和长整型之间的区别，因为在需要的时候，对普通整型的操作将生成长整型结果（也就是，在运算结果超出普通整型的数值范围之内时）。不过，开发者可以选择将字母 L（或 l）放在任何类型的整数字面常量的后面以明确地表示该整数是长整型。例如：

```
1L, 23L, 99999333493L                  # 长十进制整数
01L, 027L, 01351033136165L             # 长八进制整数
0x1L, 0x17L, 0x17486CBC75L             # 长十六进制整数
```

这里使用了大写字母 L，而不是小写字母 l，因为小写的 l 看起来像数字 1。长整型和普通整型之间的区别是一个实现。长整型没有预定义的大小限制；只要内存允许，长整型可以无限大。普通整型只占用了几个字节的内存，并且其最小和最大值是由计算机的架构决定的。Sys.maxint 是可以使用的最大正整数，而-sys.maxint-1 是最大的负整数。在 32 位计算机上，sys.maxint 是 2147483647。

浮点型数字

浮点型字面常量可以由包含小数点（.）、指数部分（e 或 E，可以选择后面为+或−，然后是一个或多个数字），或者包含这两个部分的十进制数字序列表示。浮点型字面常量的第一个字符不能是 e 或 E；但可以是任何数字或小数点（.）。例如：

```
0., 0.0, .0, 1., 1.0, 1e0, 1.e0, 1.0e0
```

Python 浮点型值对应于 C 语言的双精度型（double），并与其具有相同的范围和精度限制，在最新的平台上通常都是 53 位精度（Python 并没有提供方法以得到开发者平台上的浮点型值的确切范围和精度）。

复数

复数是由两个浮点值组成的，一个是实部，另一个是虚部。开发者可以通过访问只读属性 z.real 和 z.imag 访问复数对象 z 的两个部分。开发者可以将一个虚数字面常量指定为一个浮点或十进制字面常量，后面跟一个 j 或 J：

```
0j, 0.j, 0.0j, .0j, 1j, 1.j, 1.0j, 1e0j, 1.e0j, 1.0e0j
```

这些字面常量末尾的 j 表示−1 的平方根，复数通常在电气工程中使用（在有些其他语言中使用 i 来表示，但是 Python 选择使用 j）。除此之外没有其他的复杂字面常量。为了表示任何复数常数，可以使用一个浮点（或整数）字面常量加上或减去一个虚数。例如，要想表示数字 1 的复数，可以使用表达式 1+0j 或者 1.0+0.0j。

序列

序列（sequence）是一个排序的项目容器，按非负整数索引。Python 提供了一些内置的序列类型，包括字符串（普通和统一的字符编码标准 Unicode）、元组和列表。库和扩展模块提供了其他一些序列类型，开发者也可以编写自己的序列类型（参见第 5.2 节中

介绍的序列）。开发者可以按照各种方法操作序列，参见第 4.6 节。

可迭代对象

一个可以用来概括“序列”的 Python 概念就是可迭代对象（iterable），参见第 4.10 节中介绍的迭代器。所有的序列都是可迭代的：只要本书提到可以使用可迭代对象，则表示开发者可以使用序列（例如，列表）。

另外，当本书提到开发者可以使用可迭代的对象时，通常表示一个“有界”（bounded）的可迭代对象，也就是一个最终可以停止生成项目的可迭代对象。所有序列都是有界的。而通常，可迭代对象可以是无界的，但是，如果开发者试图使用一个无界的可迭代对象，而没有进行特别处理，可能会很容易引发一个永远都不能停止，或者是一个耗尽所有可用内存的程序。

字符串

内置字符串对象（普通或 Unicode）是一个字符序列，用来保存和显示基于文本的信息（普通字符串有时候也用来保存和显示由二进制字节组成的任意序列）。Python 中的字符串都是不变的（immutable），这意味着在对字符串执行操作时，总是会产生一个新的字符串对象，而不是修改一个现有的字符串。字符串对象提供了多个方法，本书第 9.1 节中将详细介绍这些方法。

字符串字面常量可以是被引用的或者被三重引用的。引用的字符串是一个由包含在单引号（'）或双引号（"）中的零个或多个字符组成的序列。例如：

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

这两种不同类型的引用功能是相同的；提供这两种功能可以让开发者在使用一种类型引用的字符串中包含另一种类型引用的字符串，这样就不需要使用反斜线字符（\）来将引号转义为字符：

```
'I\'m a Python fanatic'          # 引号可以被转义为字符
"I'm a Python fanatic"           # 这种方式可读性更好
```

除此之外，所有其他的使用都是相同的，使用单引号表示字符串字面常量是更常见的 Python 风格。要想让字符串常量跨越多个物理行，可以使用反斜线（\）作为一行文本的最后一个字符，以表示下一行文本是这一行的后续文本：

```
"A not very long string\
that spans two lines"            # 上一行不允许有注释
```

要想让输出的字符串跨越两行，可以在字符串中嵌入一个新行：

```
"A not very long string\n\
that prints on two lines"        # 上一行不允许有注释
```

一种更好的方法是使用三重引用字符串，该字符串被包含在三个引号字符（“或”）中：

```
"""An even bigger
string that spans
three lines"""                    # 前面几行不允许有注释
```

在三重引用字符串字面常量中，字面常量中的断行将被保留，并作为结果字符串对象中的新行的字符串。

尽管一般引用的字符串不能包含未加转义的反斜线，不能包含行的末尾，也不能包含引用字符本身，但是，唯一不能在三重引用字符串中使用的字符就是未加转义的反斜线（\）。反斜线字符用来开始一个转义序列，这样可以在各种类型字符串中放入任何字符。Python 中的字符串转义序列如列表 4-1 所示。

表 4-1　字符串转义序列

序　列	含　义	ASCII/ISO 码
\<newline>	忽略行尾	None
\\	反斜线	0x5c
\'	单引号	0x27
\"	双引号	0x22
\a	响铃（Bell）	0x07
\b	退格（Backspace）	0x08
\f	换页（Form feed）	0x0c
\n	换行（Newline）	0x0a
\r	回车（Carriage return）	0x0d
\t	制表符（Tab）	0x09
\v	垂直制表符（Vertical tab）	0x0b
DDD	八进制值 *DDD*	给定的值
\x*XX*	十六进制值 *XX*	给定的值
other	其他字符	0x5c + 给定的值

字符串字面常量的一个变体是原生字符串（raw string）。其语法与引用或三重引用字符串字面常量相同，区别在于其左引号之前有一个 r 或 R 字符。在原生字符串中，这些转义序列与表 4-1 中的解释有所区别，原生字符串中的转义序列将被直接从字面上复制到字符串中，包括反斜线和换行字符。原生字符串语法便于处理包含许多反斜线的字符串，例如在正则表达式中（参见第 9.7 节）。原生字符串不能以奇数个反斜线结束；最后一个反斜线将被作为转义右引号使用。

Unicode 字符串字面常量与其他字符串字面常量具有相同的语法，在其左引号的前面带

有字符 u 或 U。Unicode 字符串字面常量可以使用\u 加 4 个十六进制数字来表示 Unicode 字符，并且可以包含表 4-1 中列出的转义序列。Unicode 字面常量还可以包含转义序列\N{name}，其中 name 是一个标准 Unicode 名称，http://www.unicode.org/charts/列出了所有的 Unicode 名称。例如，\N{Copyright Sign}表示一个 Unicode 版权符号字符（©）。原始 Unicode 字符串以 ur，而不是 ru 开始。请注意，原生字符串并不是一种不同于普通字符串的字符类型：原生字符串只是两种常用的字符类型中可以替代字面常量的一种语法，这两种字符类型也就是普通字符串（又称为字节字符串）和 Unicode 字符串。

任意类型（引用的、三重引用的、原始的、Unicode）的多个字符串字面常量可以通过一些可选的空格邻接在一起。编译器将把这些邻接的字符串字面常量串联成单个字符串对象。如果串联的单个字符串对象中的任何字面常量是 Unicode 类型的，则整个对象的结果是 Unicode 类型。按这种方式定义一个长字符串字面常量可以让字符串可以跨越多个物理行进行显示，这样具有很好的可读性，同时还可以在该字符串的一部分物理行后面插入注释。例如：

```
marypop = ('supercalifragilistic'     # 打开括号 -> 逻辑行继续
           'expialidocious')          # 在后续行中忽略缩进
```

Marypop 将被赋值为一个字符串，这个字符串是一个由 34 个字符组成的单词。

元组

元组（tuple）是一个顺序不可变的项目序列。元组中的项目都是断言对象，并且可以是不同的类型。开发者可以使用一连串由逗号（,）分隔的表达式（元组中的项目）来指定一个元组。在最后一个项目的后面，还可以有选择地多放置一个逗号。开发者可以在圆括号中对元组中的项目进行分组，但是圆括号只有在逗号具有其他含义（例如，在函数调用中），或者表示空白的或嵌套的元组时才是必需的。包含两个项目的元组通常被称为“对”（pair）。要想创建一个由单个项目组成的元组（通常被称为“单例”，singleton），可以在表达式的末尾添加一个逗号。要想表示一个空白元组，可以使用一对空白的圆括号。下面列出了一些元组，所有元组都包含在可选的圆括号中：

```
(100, 200, 300)                    # 包含 3 个项目的元组
(3.14,)                            # 包含 1 个项目的元组
()                                 # 空白元组（圆括号不是可选的，而是必需的！）
```

开发者还可以调用内置的类型 tuple 来创建一个元组。例如：

```
tuple('wow')
```

这行代码将创建下面这个元组：

```
('w', 'o', 'w')
```

不带任何参数的 tuple()将创建并返回一个空白元组。当 x 可迭代时，tuple(x)将返回一个元组，其中的项目与 x 中的项目相同。

列表

列表（list）是一个顺序可变的项目序列。列表中的项目可以是不同类型的任意对象。开发者可以使用包含在方括号（[]）中的一连串由逗号（,）分隔的表达式（列表中的项目）来指定一个列表。在最后一个项目的后面，可以有选择地多放置一个逗号。要想表示空白列表，可以使用一个空白的方括号对。下面列出了一些示例：

```
[42, 3.14, 'hello']                    # 包含 3 个项目的列表
[100]                                  # 包含 1 个项目的列表
[]                                     # 空白列表
```

开发者还可以调用内置的类型 list 来创建一个列表。例如：

```
list('wow')
```

这行代码将创建下面这个列表：

```
['w', 'o', 'w']
```

不带任何参数的 list()将创建并返回一个空白列表。当 x 可迭代时，list(x)将返回一个列表，其中的项目与 x 中的项目相同。开发者还可以使用列表推导来构建列表，参见第 4.10 节“for 语句”中介绍的列表推导。

集合

Python 2.4 引入了两个内置的集合类型（set 和 frozenset）来表示由不重复的项目组成的任意顺序的集合。这些类型相当于标准库模块 sets 中的 Set 和 ImmutableSet 类。Python 2.3 中也存在这个标准库模块。要想确保开发者的模块使用最可用的集合，在 Python 2.3 以后的任何 Python 发布中，可以将下面这段代码放在模块的开始：

```
try:
  set
except NameError:
  from sets import Set as set, ImmutableSet as frozenset
```

集合中的项目可以是不同的类型，但是必须是可哈希的（hashable），参见第 8.2 节。set 类型的实例都是可变的，因此不是可哈希的；frozenset 类型的实例都是不可变的和可哈希的。因此开发者不能拥有一个项目为 set 的集合，但是可以拥有一个项目为 frozenset 的集合（或 frozenset）。集合和 frozenset 都不是排好序的。

要想创建一个集合，可以不带参数（这意味着一个空白集合）或者带一个可迭代参数（这意味着一个包含可迭代项目的集合）调用内置的类型 set。

字典

映射（mapping）是一个按照近乎是任意值排序的对象组成的任意集合，这些任意值被称为键（key）。与序列不一样，映射是可变的，并且不是排好序的。

Python 提供了单个内置映射类型，也就是字典类型。库和扩展模块提供了其他一些映射类型，并且开发者可以编写一些自己的类型（参见第 5.2 节中介绍的映射）。字典中的键可以是不同类型，但是这些键必须是可哈希（hashable）的（参见第 8.2 节）。字典中的值是任意对象，并有可能是不同类型。字典中的项目是一个键/值对。开发者可以将字典看作是联合的数组（在其他语言中常常被称为“映射”、“哈希表”或“哈希”）。

开发者可以使用在花括号（{}）中使用逗号分隔的一连串表达式对（这些对就是字典中的项目）来表示字典。开发者还可以选择在最后一个项目的后面多放置一个逗号。字典中的每个项目都可以写作 key:value 形式，其中 key 是一个给定项目的键的表达式，而 value 则是给定项目的值的表达式。如果某个键在字典字面常量中出现了多次，则只能在结果字典对象中保留该键的一个项目，字典不允许包含重复的键。要想表示空白字典，可以使用一个空白的花括号对。下面列出了一些字典示例：

```
{'x':42, 'y':3.14, 'z':7 }          # 包含 3 个项目和字符串键的字典
{1:2, 3:4 }                         # 包含 2 个项目和整数键的字典
{}                                  # 空白字典
```

开发者还可以调用内置的类型 dict 来创建一个字典，这种方法尽管并不简练，但是有时更具可读性。例如，上面这几个示例中的字典也可以分别写成：

```
dict(x=42, y=3.14, z=7)             # 包含 3 个项目和字符串键的字典
dict([[1, 2], [3, 4]])              # 包含 2 个项目和整数键的字典
dict()                              # 空白字典
```

不带任何参数的 dict()可以创建并返回一个空白目录。当 dict 的参数 x 是一个映射时，dict 将返回一个新的字典对象，具有与 x 相同的键和值。当 x 是可迭代的值时，x 中的项目必须是一对值，并且 dict(x)将返回一个字典，其中的项目（键/值对）与 x 中的项目相同。如果某个键在 x 中出现了多次，则结果字典中只保留该键的最后一个项目。

在开发者调用 dict 时，除了定位参数 x，可能还会传递一个命名参数（named argument），每个参数的语法都是 name=value，其中 name 是一个标识符，用作项目的键，而 value 是一个表达式，用作该项目的值。当开发者调用 dict 并传递一个定位参数和一个或多个命名参数时，如果某个键同时出现在位置参数中，并作为一个命名参数，则 Python 将把该命名参数（例如，命名参数“wins”）的值关联到这个键上。

开发者还可以通过调用 dict.fromkeys 来创建一个字典。第一个参数是一个可迭代的值，其项目将成为该字典的键；第二个参数是对应到每个键上的值（所有键最初都具有相同的对应值）。如果忽略第二个参数，则对应于每个键的值为 None。例如：

```
dict.fromkeys('hello', 2)           # 等同于 {'h':2, 'e':2, 'l':2, 'o':2}
dict.fromkeys([1, 2, 3])            # 等同于 {1:None, 2:None, 3:None}
```

None

内置的 None 表示一个空（null）对象。None 没有方法和其他属性。开发者可以在需要

一个引用，但是不介意引用什么对象的时候，或者在需要表示这里没有对象的时候，使用 None 作为一个占位符。函数将返回 None 作为其结果。除非某些函数具有特殊的 return 语句以返回其他值，否则这些函数将返回 None 作为其结果。

可调用

在 Python 中，可调用（callable）类型是那些类型的实例支持函数调用操作（参见第 4.11 节）的类型。函数是可调用的。Python 提供了几个内置函数（参见第 8.2 节），并支持用户自定义函数（参见第 4.11 节）。生成器也是可调用的（参见第 4.11 节）。

类型也是可调用的，正如本书已经在 dict、list 和 tuple 内置类型中介绍的那样（参见第 8.1 节以了解完整的内置类型列表）。正如本书将在第 5.1 节中介绍的，类对象（用户自定义类型）也是可调用的。调用一个类型通常会创建并返回一个该类型的新实例。

其他的可调用对象就是方法（method），方法是一些被限定为类属性和支持特殊方法命名_ _call_ _的类实例的函数。

布尔型值

Python 中的所有数据值都可以被看作是一个真值：真（True）或假（False）。任何非零数字或非空容器（例如，字符串、元组、列表、集合或字典）都为真。而 0（任何数字类型）、None 和空容器都为假。在使用浮点型数字作为真值的时候要注意：这种使用相当于比较一个数字是否确切等于零，而浮点型数字几乎永远都不能用来比较确切相等！

内置类型 bool 是 int 的一个子类。Bool 类型的唯一两个值是 True 和 False，可以使用字符串‘True’和‘False’来表示，还可以使用数字值 1 和 0 来分别表示。有些内置函数会返回 bool 类型的结果，例如比较运算符。开发者可以调用以任意 x 为参数的 bool(x) 函数。如果 x 为真，则结果为 Ture，如果 x 为假，则结果为 False。好的 Python 风格是不使用这样的重复调用：可以写成 if x，不要写成 if bool(x):、if x==True:和 if bool(x)==True 等。

4.3 变量和其他引用

Python 程序通过引用来访问数据的值。引用（reference）是表示一个值（对象）在内存中的位置的名称。引用可以表现为变量、属性和项目的形式。在 Python 中，一个变量或其他引用不具备固定的类型。在给定的时间被限定到某个对象的引用总是具有一种类型，但是给定的引用可能会在程序执行期间被指定为各种类型的对象。

变量

在 Python 中没有变量的声明。变量的表现形式是以一个绑定（bind）该变量的语句开始的，换句话讲，就是设置一个名称，将引用保存到某些对象上。开发者还可以解除

绑定（unbind）一个变量，复位该名称，使其不再保存引用。赋值语句是绑定变量和其他引用的最常用方法。Del 语句可以用来取消对引用的绑定。

绑定一个已经被绑定的引用也被称为重新绑定（rebinding）。本书中只要提到了绑定都包含重新绑定，除非明确说明不包含重新绑定。如果没有任何引用绑定到一个对象，该对象将会消失，只要不出现这种情况，重新绑定或解除绑定一个引用对该引用绑定的对象没有任何影响。自动清除无引用绑定的对象被称为垃圾收集（garbage collection）。

开发者可以使用除了 30 个 Python 保留的关键字（参见第 4.1 节中介绍的关键字）之外的任何标识符来命名一个变量。变量可以是全局（global）的或本地（local）的。全局变量是模块对象（第 7 章将介绍模块）的一个属性。本地变量存在于函数的本地命名空间中（参见第 4.11 节）。

对象属性和项目

对象的属性（attribute）和项目（item）之间的主要区别存在于开发者用来访问他们的语法之中。对象的属性是通过该对象的引用，后面带一个点号（.），再带一个被称为属性名（attribute name）的标识符来表示的（例如，x.y 表示被绑定到名称 x 上的对象的一个属性，这个属性的名称为 y）。

对象的项目（item）是通过该对象的引用，后面带一个包含在方括号（[ ]）中的表达式来表示的。方括号中的表达式被称为该项目的索引（index）或键（key），而该对象被称为这个项目的容器（container）（例如，x[y]表示被绑定为名称 y 的键或索引的项目，并且该项目被包含在绑定为名称 x 的容器对象中）。

可以被调用的属性也被称为方法（method）。Python 没有对可调用和不可调用的属性给出很强的区别，而其他一些语言对此是有区别的。有关属性的所有规则也应用于可调用的属性（方法）。

访问不存在的引用

一个很常见的编程错误就是视图访问一个并不存在的引用。例如，变量可能是未绑定的，或者要应用于某个对象的属性名或项目索引并不是有效的。在 Python 编译器分析和编译源代码时，将只诊断语法错误。编译器不会诊断语义错误，比如试图访问一个未绑定的属性、项目或变量。Python 只有在错误代码执行，也就是运行时才诊断语义错误。当某个操作是一个 Python 语义错误时，试图运行该操作将引发一个异常（参见第 6 章）。与任何其他语义错误一样，访问一个并不存在的变量、属性或项目都将引发一个异常。

赋值语句

赋值语句可以是简单的或增量的。对一个变量进行简单赋值（例如，name=value）也就

是如何创建一个新变量或者将一个已有变量重新绑定到一个新值。对一个对象的属性进行简单赋值（例如，x.attr=value）也就是请求一个对象 x 创建或重新绑定属性 attr。对容器中的一个项目进行简单赋值（例如，x[k]=value）也就是请求容器 x 创建或重新绑定索引为 k 的项目。

增量赋值（例如，name+=value）本身并不能创建新引用。增量赋值可以重新绑定一个变量、要求一个对象重新绑定其已有属性或项目中的一个，或者要求目标对象修改其自身（当然，一个对象可能会创建任何想要的内容以响应这些请求）。在对一个对象发起一个请求时，只有该对象才能决定是接受这个请求还是引发一个异常。

简单赋值

简单赋值语句的最简单形式具有以下语法：

```
target = expression
```

这里，目标对象（target）也被称为左边（LHS），而表达式就是右边（RHS）。在执行赋值语句时，Python 将计算右边表达式的值，然后将该表达式的值绑定到左边的目标对象上。绑定操作并不取决于这个值的类型。特别是，Python 并没有像其他语言一样，在可调用和不可调用的对象之间设定特别强的区别，因此开发者可以将函数、方法、类型和其他可调用对象绑定到变量上，就像数字、字符串、列表等对象一样。

不过，绑定操作的细节取决于目标对象的类型。赋值语句中的目标对象可以是一个标识符、属性引用、索引或者切片。

标识符（identifier）

标识符是一个变量的名称。为一个标识符赋值也就是使用赋值的名称绑定这个变量。

属性引用（attribute reference）

属性引用具有语法 obj.name。Obj 是任意表达式，name 是一个标识符，也被称为对象的属性名。为一个属性引用赋值也就是让对象 obj 绑定其名为 name 的属性。

索引（indexing）

索引具有语法 obj[expr]。Obj 和 expr 都是任意表达式。为一个索引赋值也就是让容器 obj 绑定一个由 expr 的值指定的项目，这个值也被称为容器中的项目的索引或键。

切片（slicing）

切片具有语法 obj[start:stop]或 obj[start:stop:stride]. Obj。其中 start、stop 和 stride 都是任意表达式。Start、stop 和 stride 都是可选的（也就是说，obj[:stop:]和 obj[:stop]也都是语法正确的切片，相当于 obj[None:stop:None]）。为一个切片赋值要求容器 obj 绑定或解除绑定某些项目。为像 obj[start:stop:stride]这样的切片赋值相当于为索引

obj[slice(start, stop, stride)]赋值，其中 slice 是一个 Python 内置类型（参见第 8.1 节），其实例表现为切片。

本书在第 4.6 节中介绍列表操作时和在第 4.8 节中介绍字典时，将回过来介绍索引和切片目标对象。

当赋值语句的目标对象是一个标识符时，该赋值语句表示要绑定一个变量。这是做是被允许的：当开发者请求这样做时，绑定操作就会发生。在所有其他情况下，赋值语句表示请求一个对象绑定其一个或多个属性或项目。一个对象可能会拒绝创建或重新绑定某些（或者全部）属性或项目，如果开发者尝试一个不被允许的创建或重新绑定操作（参见第 5.2 节中对__setattr__和__setitem__的介绍）则会引发异常。

开发者可以在简单赋值语句中给出多个目标对象和等号（=）。例如：

```
a = b = c = 0
```

这个赋值语句表示将变量 a、b 和 c 绑定为相同的值 0。每次执行该语句时，右边的表达式实际上只计算了一次，不管这个语句中包含多少个目标对象。然后，每个目标对象将被绑定到该表达式返回的单个对象上，就像一个接一个地执行几个简单赋值语句一样。

简单赋值语句中的目标对象可以列举两个或更多由逗号分隔的引用，可以选择包括在圆括号或方括号中。例如：

```
a, b, c = x
```

这个语句要求 x 成为一个具有确定的三个项目的可迭代对象，并绑定 a 为第一个项目，b 为第二个项目，c 为第三个项目。这种类型的赋值语句被称为拆包赋值（unpacking assignment）。右边的表达式必须是一个可迭代对象，并且具有与目标对象中的引用完全相同的项目数量；否则，Python 将引发一个异常。目标对象中的每个引用都将绑定为右边的相应项目上。一个拆包赋值语句还可以用来交换引用：

```
a, b = b, a
```

这个赋值语句将把名称 a 重新绑定为名称 b 所绑定的对象，反之亦然。

增量赋值

增量赋值与简单赋值的区别在于，增量赋值没有在目标对象和表达式之间使用等号（=），而是使用增量运算符（augmented operator），该运算符是一个二元运算符后面带一个等号（=）。增量运算符包括+=、-=、*=、/=、//=、%=、**=、|=、>>=、<<=、&=和^=。增量赋值语句的左边可以只有一个目标对象；增量赋值不支持多个目标对象。

与简单赋值一样，在增量赋值中，Python 首先计算赋值语句右边表达式的值。然后，如果赋值语句左边的目标对象具有一个特殊的方法，正好是这个运算符的一个适当的原地运算版本，Python 将调用该方法，并以右边表达式的值作为其参数。只有该方法

才能决定如何正确地修改左边的目标对象并返回修改后的对象（本书第 5.2 节介绍了这样一些特殊方法）。如果左边的对象没有适当的原地运算特殊方法，Python 将对增量赋值语句的左边和右边对象应用相应的二元运算符，然后将目标引用重新绑定为该运算符的结果。例如，x+=y 就像 x=x.__iadd__(y)一样，其中 x 具有特殊方法__iadd__。否则 x+=y 就像 x=x+y 一样。

增量赋值不能创建目标引用；在执行增量赋值时，该目标引用必须已经被绑定。增量赋值可以将目标引用重新绑定到一个新对象或者修改目标引用已经绑定的相同对象。相反，简单赋值可以创建或重新绑定左边的目标引用，但是简单赋值不能修改该对象，即使是该目标引用以前绑定的对象。在这里，对象和对象的引用之间的区别是至关重要的。例如，x=x+y 不会修改名称 x 最初绑定的对象。而是重新绑定名称 x，以表示一个新对象。相反，x+=y 则修改了名称 x 绑定的对象，如果该对象具有特殊方法__iadd__；否则，x+=y 将把名称 x 重新绑定到一个新对象上，就像赋值语句 x=x+y 一样。

del 语句

尽管使用了这个名称，del 语句并不删除对象；而是解除绑定引用。当一个对象没有任何引用绑定存在时，结果可能是通过垃圾收集自动删除该对象。

del 语句是由关键字 del，后面带一个或多个使用逗号（,）分隔的目标引用组成的。每个目标引用可以是一个变量、属性引用、索引或切片，就像在赋值语句中使用的那样，并且在执行 del 语句的时候，每个目标引用必须是绑定的。当 del 的目标引用是一个标识符时，del 语句就表示要解除绑定该变量。如果该标识符是被绑定的，则允许解除绑定；在请求该操作时，就会发生解除绑定。

在所有其他情况下，del 语句会向一个对象指定一个请求，以解除绑定该对象的一个或多个属性或项目。对象可能会拒绝解除绑定某些（或所有）属性或项目，如果开发者试图执行一个不被允许的解除绑定操作（参见第 5.2 节中对__delattr 和__delitem__的介绍），将会引发一个异常。解除绑定一个切片通常会产生与将一个空白序列赋值给该切片相同的效果，但是只有容器对象才能实现这种等价操作。

4.4 表达式和运算符

表达式（expression）是一个代码语句，Python 将计算这段代码以产生一个结果。最简单的表达式是字面常量和标识符。开发者可以使用表 4-2 中列出的运算符和/或定界符来连接子表达式以建立其他新的表达式。表 4-2 按优先级递减顺序列出了运算符，高优先级在前，低优先级在后。列在一起的运算符具有相同的优先级。第三列列出了运算符的结合规则：L（从左到右）、R（从右到左）或 NA（无结合规则）。

表 4-2　表达式中运算符的优先级

运　算　符	说　明	结 合 规 则
'expr,...'	字符串转换	NA
{key:expr,...}	创建字典	NA
[expr,...]	创建列表	NA
(expr,...)	创建元组或圆括号	NA
f(expr,...)	函数调用	L
x[index:index]	切片	L
x[index]	索引	L
x.attr	属性引用	L
x**y	幂（x 的 y 次幂）	R
~x	按位非	NA
+x, -x	一元正和负	NA
x*y, x/y, x//y, x%y	乘法、除法、截断除法、求余	L
x+y, x-y	加法、减法	L
x<<y, x>>y	左移位、右移位	L
x&y	按位与	L
x^y	按位异或	L
x|y	按位或	L
x<y, x<=y, x>y, x>=y, x<>y, x!=y, x==y	比较（小于、小于等于、大于、大于等于、不等于、等于）[a]	NA
x is y, x is not y	同一性测试	NA
x in y, x not in y	成员测试	NA
not x	布尔非	NA
x and y	布尔与	L
x or y	布尔或	L
lambda arg,...: expr	匿名简单函数	NA

[a] <>和!= 是不等于运算符的两种不同表现形式。!= 是首选版本；<>是旧版本。

在表 4-2 中，expr、key、f、index、x 和 y 表示任意表达式，而 attr 和 arg 表示任何标识符。除字符串转换之外，符号,...表示使用逗号分隔的零个或多个重复表达式，字符串转换需要一个或多个重复表达式。除了字符串转换运算符，在所有这些运算符中，拖尾逗号是允许使用和没有危害的，在字符串运算符中禁止使用拖尾逗号。因为这种奇怪的特性，本书不推荐使用字符串转换运算符；而建议使用内置函数 repr（参见第 8.2 节）。

比较链

开发者可以将比较运算符链接在一起，隐含逻辑与（and）操作。例如：

```
a < b <= c < d
```

与下面的表达式具有相同的含意：

```
a < b and b <= c and c < d
```

这种链接形式更有可读性，并且可以一次计算每个子表达式。

短路运算符

运算符 and 和 or 可以短路（short-circuit）其操作数的计算：只有其值是为了得到完整的 and 或 or 运算的真实值所必需的时候，右边的操作数才会真正进行计算。

换句话讲，x and y 将首先计算 x。如果 x 为假，则结果就是 x；否则，结果为 y。同样地，x or y 将首先计算 x。如果 x 为真，则结果为 x；否则，结果为 y。

and 和 or 并不强制其结果为真（True）或假（False），而是返回一个或另一个操作数。这可以让开发者更广泛地使用这些运算符，而不只是在布尔（Boolean）运算环境中。由于具备短路语义，and 和 or 不同于所有其他运算符，那些运算符在执行运算之前需要完全计算所有的操作数。and 和 or 可以让左边的操作数作为右边操作数的防护。

Python 2.5 三元运算符

Python 2.5 引入了另外一个短路运算符，也就是三元运算符 if/else：

```
whentrue if condition else whenfalse
```

whentrue、whenfalse 和 condition 都是任意表达式。该运算符将首先计算 condition。如果 condition 为真，则结果为 whentrue；否则，结果为 whenfalse。根据 condition 的真值，该运算符只计算两个子表达式 whentrue 和 whenfalse 中的一个。

这个新的三元运算符中的三个子表达式的顺序可能有点让人混淆。还有，一种推荐的风格就是总是将整个表达式放在圆括号中。

4.5 数值运算

Python 支持常用的数值运算，参见表 4-2 中显示的运算符。数字是不可变对象：当开发者对数字对象执行数值运算时，总是会产生一个新数字对象，并且永远都不能修改已有的数字。开发者可以通过只读属性 z.real 和 z.imag 访问复数对象 z 的实部和虚部。如果尝试重新绑定复数对象的这些属性，将会引发一个异常。

数字的可选+或-符号，以及将浮点型字面常量加入到复数的虚部以生成一个复数的+符

号都不是该字面常量语法的一部分。这些都是普通的运算符，遵循普通运算符的优先级规则（参见表 4-2）。例如-2**2 的计算结果为-4：幂运算比一元负运算具有更高的优先级，因此整个表达式应该解析为-(2**2)，而不是(-2)**2。

数值转换

开发者可以在任意两个 Python 内置类型的数字之间执行算术运算和比较。如果操作数的类型不同，将应用强制转换：Python 将把具有“较小”类型的操作数转换为“较大”类型的操作数。这些类型，按从最小到最大的顺序排列，分别是整型、长整型、浮点型数字和复数。

开发者可以通过向任何内置数字类型（包括 int、long、float 和 complex）传递一个非复数的数值参数来请求显式转换。如果这个参数存在小数，使用 int 和 long 将丢弃该参数的小数部分，例如，int(9.8)的转换结果为 9。开发者还可以通过两个数值参数来调用复数类型，向其提供实部和虚部，但是不能使用这种方法将一个复数转换为另一种数值类型，因为没有一种唯一确定的方法可以将一个复数转换为另一种类型，例如，浮点型。

使用适当的数值字面常量语法，并通过很小的扩展：作为参数的字符串可能会包含起始和/或结束空格、可能会以正负号开始，对于复数，还可能会加上或减去一个实部和虚部，这样，每个内置的数值类型还可以使用字符串作为参数。调用 int 和 long 需要用到两个参数：第一个参数是要转换的字符串，第二个参数是基数（radix），2～36 的任何整数可以作为数值转换的基数（例如，int('101', 2)）将会返回 5，也就是基数为 2 时‘101’的值）。

算术运算

Python 的算术运算采用了最显而易见的运算方式，并可能引发除法和幂运算异常。

除法

如果/、//或%右边的操作数为 0，Python 将引发一个运行时异常。//运算符将执行截断除，这意味着该运算符将返回一个整数结果（转换为与位数更长的操作数相同的类型）并忽略剩余的小数，如果该操作数包含小数部分。当两个操作数都是整数（普通整型或者长整型）时，如果 Python 命令行上使用了-Qold 开关（在 Python 2.3、2.4 和 2.5 中，默认使用-Qold），则/运算符的使用与//运算符相同。否则，/运算符将执行真正的除法，并返回浮点型结果（如果其中的一个操作数是复数，则返回复数结果）。在 Python 2.3、2.4 或 2.5 中，要想让/运算符对整型操作数执行真正的除法，可以在 Python 命令行上使用-Qnew 开关，或者在源文件的起始位置使用以下语句：

```
from __future__ import division
```

这条语句可以确保/运算符（只能在以这条语句开始的模块中）在运算时不对任何类型的操作数进行截断。

要想确保除法运算不依赖于-Q 开关，而且不依赖于正在使用的确切 Python 版本，在想要进行截断除时，应该总是使用//运算符。如果不想进行截断，则使用/运算符，但是还需要确保至少一个操作数不是整数。例如，开发者可以使用代码 1.0*a/b，而不是 a/b，这样可以避免对操作数 a 和 b 的类型进行任何假设。要想检查程序在使用除法时是否存在版本依赖性，可以在 Python 命令行上使用-Qwarn 开关，从而获得所有对整型操作数使用/运算符时产生的运行时警告。

内置的 divmod 函数需要用到两个数值参数，并返回由商和余数组成的数值对，因此不需要使用//求商，并使用%求余数。

幂运算

对于幂（“次方”）运算 a**b，如果 a 小于零，并且 b 是一个带有非零小数部分的浮点型值，则该幂运算将引发一个异常。内置的 pow(a, b)函数将返回与 a**b 相同的结果。带有三个参数的 pow(a, b, c)将返回与(a**b)%c 相同的结果，但是运算更快。

比较

包括数字在内的所有对象都可以进行相等（==）和不等（!=）比较。比较指令（<、<=、>、>=）可以在任意两个数字之间使用，除非其中的一个操作数是复数，在这种情况下将引发运行时异常。所有这些运算符将返回布尔型值（True 或 False）。

整数的按位运算

整型数值和长整型数值可以被看作是由多个位组成的字符串，还可以使用表 4-2 中列举的按位运算。按位运算符的优先级低于算术运算符。从概念上讲，可以在正整数的左边扩展无穷个显示为字符串 0 的位。负整数表现为 2 的补数，因此从概念上讲，可以在负整数的左边扩展无穷个显示为字符串 1 的位。

4.6 序列运算

Python 支持多种适用于所有序列的运算，这些序列包括字符串、列表和元组。有些序列运算可以应用于所有容器（包括不是序列的对象，例如，集合和字典），有些可以应用于所有可迭代对象（也就是“可以循环的任意对象”，参见第 4.2 节；所有容器，不管是序列还是其他内容，都是可迭代的。另外还包括许多不是容器的对象，比如文件，参见第 10.3 节，以及生成器，参见第 4.11 节）。本书后面将非常精确和具体地使用术语序列（sequence）、容器（container）和可迭代对象（iterable）来说明哪些运算可以应用于每个类别。

序列概述

序列就是其项目可以通过索引或切片访问的容器。内置的 len 函数可以将任意容器作为一个参数，并返回该容器中项目的数量。内置的 min 和 max 函数可以将其项目是可比较的一个非空迭代作为一个参数，并分别返回最小和最大的项目。开发者还可以在调用 min 和 max 函数时提供多个参数，在这种情况下将分别返回最小和最大的参数。内置的 sum 函数可以将其项目为数字的迭代作为一个参数，并返回这些数字的和。

序列转换

除了在需要的时候可以将普通字符串转换为 Unicode 字符串，不同的序列类型之间不存在隐式转换（第 9.6 节详细介绍了字符串转换）。开发者可以对单个参数（任意迭代）调用内置的 tuple 和 list 函数以获得正在调用的类型的一个新实例，新实例中包含与调用的参数相同的项目（并按照相同的顺序排列）。

串联和重复

开发者可以使用+运算符将相同类型的序列串联在一起。开发者还可以使用*运算符将一个序列 S 乘以整数 n。S*n 或者 n*S 表示将 n 个 S 的副本串联在一起。当 n<=0 时，S*n 是一个与 S 具有相同类型的空白序列。

成员测试

x in S 运算符可以通过测试检查对象 x 是否等于序列（或者其他类型的容器或迭代）S 中的任何项目。如果存在相等的项目，则返回 True，如果不存在，则返回 False。x not in 运算符与 not (x in S)一样。不过，在特殊情况下，比如字符串，x in S 具有更广泛的应用；在这种情况下，该运算符将测试 x 是否等于字符串 S 的任意子字符串，而不仅仅是任意单个字符。

索引序列

序列 S 的第 n 个项目可以表示为索引：S[n]。索引是从 0 开始的（S 的第一个项目是 S[0]）。如果序列 S 包含 L 个项目。则索引 n 可以是 0、1 … 最大到 L−1，并包括 L−1，但没有更大的索引了。索引 n 还可以是−1、−2 … 最小到−L，并包括−L，但没有更小的索引了。负的 n 表示序列 S 的第 L+n 个项目。也就是说，S[−1]相当于 S[L−1]，是序列 S 的最后一个项目，S[−2]是倒数第二个项目，依此类推。例如：

```
x = [1, 2, 3, 4]
x[1]                # 2
x[-1]               # 4
```

使用>=L 或<−L 的索引将会引发一个异常。为项目指定一个无效的索引也会引发异常。

开发者可以向一个列表添加一个或多个元素，但是这样做是对一个切片赋值，而不是对一个项目，下面将对此进行介绍。

对序列切片

要想表示序列 S 的子序列，可以使用切片（slicing），其语法是 S[i:j]，其中 i 和 j 都是整数。S[i:j]是 S 的子序列，从第 i 个项目开始（包括第 i 个项目），到第 j 个项目结束（但不包括第 j 个项目）。在 Python 中，范围总是包括下限，但不包括上限。如果 j 小于或等于 i，或者 i 大于或等于 L（也就是序列 S 的长度），则该切片是一个空白序列。如果 i 等于 0，则可以省略 i，这样切片就会从 S 的起始项目开始。如果 j 大于或等于 L，则可以省略 j，这样切片就会延伸到 S 的最后一个项目。还可以省略两个索引，这样就意味着整个序列的一个副本：S[:]。两个索引都可以小于 0。负索引表示 S 中的相同项目 L+n，就像在上一节的索引序列中介绍的那样。大于或等于 L 的索引表示 S 的结尾，小于或等于−L 的负索引则表示 S 的开始。下面是一些示例：

```
x = [1, 2, 3, 4]
x[1:3]                  # [2, 3]
x[1:]                   # [2, 3, 4]
x[:2]                   # [1, 2]
```

切片还可以使用扩展语法 S[i:j:k]。k 是切片的步长（stride），也就是连续的两个索引之间的距离。S[i:j]等于 S[i:j:1]，S[::2]是 S 的子序列，包括序列 S 中所有索引为偶数的项目，而 S[::−1]包含序列 S 中的所有项目，但是顺序相反。

字符串

字符串对象是不可变的，因此试图重新绑定或删除字符串的一个项目或切片将会引发一个异常。字符串对象的项目（对应于字符串中的每个字符）本身也都是字符串，每个字符的长度为 1。字符串对象的切片仍然是字符串。字符串对象可以使用几个方法，参见第 9.1 节中的介绍。

元组

元组对象是不可变的，因此试图重新绑定或删除元组中的一个项目或切片将会引发一个异常。元组中的项目可以是任意对象，并且可以具有不同类型。元组的切片也是元组。元组没有可供使用的普通（非特殊）方法，只能使用第 5.2 节中介绍的某些特殊方法。

列表

列表对象是可变的，因此可以重新绑定或删除列表中的项目和切片。列表中的项目可以是任意对象，并且可以具有不同类型。列表的切片也是列表。

修改列表

开发者可以通过为一个索引的项目赋值来修改列表。例如：

```
x = [1, 2, 3, 4]
x[1] = 42                    # x 现在是[1, 42, 3, 4]
```

修改列表对象 L 的另一种方法就是使用 L 的一个切片作为赋值语句的目标对象(左边)。赋值语句的右边必须是一个迭代。如果左边的切片是扩展格式，则右边必须具有与左边切片中的项目数量相同的项目。不过，如果左边切片是普通（非扩展）格式，则左边切片和右边切片可以是任意长度；为列表的普通（非扩展）切片赋值可以添加或删除列表的项目。例如：

```
x = [1, 2, 3, 4]
x[1:3] = [22, 33, 44]     # x 现在是[1, 22, 33, 44, 4]
x[1:4] = [8, 9]           # x 现在是[1, 8, 9, 4]
```

下面是为切片赋值的一些重要的特殊情况。

- 使用空白列表[ ]作为右边的表达式可以从列表 L 中删除目标切片。换句话讲，L[i:j]=[ ]具有与 del L[i:j]相同的效果。
- 使用 L 的空白切片作为左边的目标对象可以将右边的项目插入到列表 L 的适当位置。也就是说，L[i:i]=['a', 'b']将把项目'a'和'b'插入到赋值前的列表 L 中索引为 i 的项目的前面。
- 使用包含整个列表对象 L[:]的切片作为左边对象将完全替换列表 L 中的内容。

可以使用 del 从列表中删除一个项目或切片。例如：

```
x = [1, 2, 3, 4, 5]
del x[1]                  # x 现在是[1, 3, 4, 5]
del x[::2]                # x 现在是[3, 5]
```

列表的原地运算

列表对象定义了+ 和 * 运算符的原地（in-place）运算版本，可以通过增量赋值语句使用。增量赋值语句 L+=L1 具有与将迭代 L1 中的项目添加到 L 的末尾相同的效果。L*=n 具有将 L 的 n-1 个副本添加到 L 的末尾的效果；如果 n<=0，L*=n 将导致 L 中的内容为空白，与 L[:]=[ ]一样。

列表方法

列表对象提供了如表 4-3 所示的几种方法。非变异方法（nonmutating method）可以返回一个结果，并且不会改变该方法应用的对象，而变异方法（mutating method）可能会改变该方法应用的对象。许多变异方法的操作就像为列表的适当切片赋值一样。在表 4-3 中，L 表示任意列表对象，i 表示 L 中的任意有效索引，s 表示任意迭代，x 表示任

意对象。

表 4-3　列表对象的方法

方　法	说　明
非变异方法	
L.count(x)	返回列表 L 中等于 x 的项目的数量
L.index(x)	返回列表 L 中出现的第一个等于 x 的项目的索引，如果列表 L 中没有等于 x 的项目，则引发一个异常
变异方法	
L.append(x)	将项目 x 添加到列表 L 的末尾；例如，L[len(L):]=[x]
L.extend(s)	将迭代 s 中的所有项目添加到列表 L 的末尾；例如，L[len(L):]=s
L.insert(i, x)	将项目 x 插入到列表 L 中索引为 i 的项目之前，移动列表 L“右边”的其他项目（如果有的话）为插入的项目留出空间（len(L)增加 1，但是不替换任何项目，也不引发异常：就像 L[i:i]=[x]一样
L.remove(x)	从列表 L 中删除出现的第一个等于 x 的项目，如果列表 L 中没有等于 x 的项目，则引发一个异常
L.pop([i])	返回列表 L 中索引为 i 的项目的值，并从列表 L 中删除该项目；如果省略 i，则删除并返回最后一个项目；如果列表 L 为空，或者 i 是一个无效索引，则引发一个异常
L.reverse( )	在原地反转列表 L 中的项目
L.sort([f])　(2.3)	在原地对列表 L 中的项目排序，使用函数 f 成对地比较项目；如果省略 f，则使用内置函数 cmp 进行比较。要想获得更多详细信息，请参见第 4.6 节中介绍的列表排序
L.sort(cmp=cmp, key=None, reverse=False)　(2.4)	在原地对列表 L 中的项目排序，使用传递为 cmp 的函数（在默认情况下，使用内置函数 cmp）成对地比较项目。当参数 key 不是 None 时，比较的是每个项目 x 的 key(x)，而不是 x 本身。要想获得更多详细信息，请参见第 4.6 节中介绍的列表排序

除了 pop，列表对象的所有变异方法都返回 None。

列表排序

列表的 sort 方法可以让列表以一种保证恒定的方式（比较结果相等的元素不交换位置）在原地进行排序（对列表中的项目重新排序，按照递增顺序放置项目）。在实际应用中，sort 是极其快的，经常是超乎想象地快，因为 sort 可以充分利用可能会出现任何子列表中的任何顺序或者倒序（sort 使用的高级算法被称为 timsort 以对其开发者 Tim Peter 表示尊敬，从技术上讲，这个算法是一个“非-递归自适应恒定自然归并排序/二进制插入排序混合算法”）。

在 Python 2.3 中，列表的 sort 方法有一个可选参数。在使用时，该参数必须是一个函数，

在以任意两个列表项目作为参数调用该函数时，将会返回-1、0 或 1，返回值取决于第一个项目被判定为小于、等于或大于第二个项目，这个返回值被用来进行排序。尽管传递参数可以很容易地以非常灵活的方式对小的列表进行排序，但是会减缓排序的过程。装饰-排序-去除装饰（Decorate-Sort-Undecorate）的习惯用法（参见第 18.4 节中介绍的搜索和排序）要更快一些，至少同样灵活，并且通常要比向 sort 传递一个参数产生更少的错误。

在 Python 2.4 中，sort 方法有三个可选参数，可以向其传递位置或者命名参数语法。参数 cmp 与 Python 2.3 中唯一的（未命名）可选参数具有相同的角色。如果不为 None，则参数 key 必须是一个可以使用任何列表项目作为其唯一参数调用的函数。在这种情况下，要想比较任意两个项目 x 和 y，Python 将使用 cmp(key(x), key(y))，而不是 cmp(x, y)（在实际应用中，这与第 18.4 节中介绍搜索和排序时提到的装饰-排序-去除装饰是以相同的方式实现的，并且甚至可以更快）。如果参数 reverse 为 True，将导致每个比较的结果被反转；这与排序之后反转列表 L 并不相同，因为不管参数 reverse 为 True 还是 False，顺序都是恒定的（比较结果相等的元素不交换位置）。

Python 2.4 还提供了一个新的内置函数 sorted（参见第 8.2 节）以从任何输入迭代生成一个排序的列表。与列表的 sort 方法一样，sorted 函数可以接受相同的输入参数。此外，Python 2.4 还向模块 operator（参见第 15.2 节）添加了两个新的高阶函数 attrgetter 和 itemgetter，这使得函数特别适合于列表的 sort 方法和新的内置函数 sorted 的 key=可选参数。在 Python 2.5 中，一个相同的 key=可选参数也被添加到内置函数 min 和 max，以及标准库模块 heapq 中的 nsmallest 和 nlargest 函数中，参见第 8.8 节。

4.7 集合运算

Python 提供了多种适用于集合的运算。由于集合是容器，因此内置的 len 函数可以将一个集合作为其单个参数，并返回该集合对象中项目的数量。集合是可迭代的，因此开发者可以将其传递给包含一个可迭代参数的任何函数或方法。在这种情况下，集合中的项目是按照某些任意顺序进行迭代的。例如，对于任意集合 S，min(S)将返回 S 中的最小项目。

集合成员

k in S 运算符可以检查对象 k 是否是集合 S 中的一个项目。如果是，则返回 True，如果不是，则返回 False。与此类似，k not in S 相当于 not (k in S)。

集合方法

集合对象提供了如表 4-4 所示的一些方法。非变异方法将返回一个结果，并且不改变其

应用的对象，还可以对类型为 frozenset 的实例调用该方法，而变异方法可能会改变其应用的对象，并且只能对类型为 set 的实例调用该方法。在表 4-4 中，S 和 S1 表示任何集合对象，x 表示任何可哈希的对象。

表 4-4　集合对象的方法

方　法	说　明
非变异方法	
S.copy( )	返回集合的一个简化副本（该副本中的项目是集合 S 中的相同对象，但不是完全的副本）
S.difference (S1)	返回在集合 S 中，但是不在集合 S1 中的所有项目组成的集合
S.intersection(S1)	返回在集合 S 中，同时也集合 S1 中的所有项目组成的集合
S.issubset(S1)	如果集合 S 中的所有项目也都在集合 S1 中，则返回 True；否则，返回 False
S.issuperset(S1)	如果集合 S1 中的所有项目也都在集合 S 中，则返回 True；否则，返回 False（与 S1.issubset(S)）类似）
S.symmetric_difference(S1)	返回在集合 S 或 S1 中，但是不同时在两个集合中的所有项目组成的集合
S.union(S1)	返回在集合 S、S1 或同时在这两个集合中的所有项目组成的集合
变异方法	
S.add(x)	将 x 添加为集合 S 中的一个项目；如果 x 已经是集合 S 中的一个项目，则不对集合进行任何操作
S.clear( )	从集合 S 中删除所有项目，使得 S 为空白集合
S.discard(x)	删除集合 S 中的项目 x；如果 x 并不是集合 S 中的一个项目，则不对集合进行任何操作
S.pop( )	删除并返回集合 S 中的任意一个项目
S.remove(x)	删除集合 S 中的项目 x；如果 x 不是集合 S 中的项目，引发一个 KeyError 异常

除了 pop 之外，集合对象的所有变异方法都返回 None。

pop 方法可以用于对集合进行破坏性的迭代，并且几乎不消耗额外内存。当开发者想要的是在执行循环操作的同时“消耗掉”集合（减少集合中的项目）的话，内存的节省使得 pop 可以用来对巨大的集合执行循环操作。

集合还包含名为 different_update、intersection_update、symmetric_difference_update 和 update（对应于非变异方法 union）的变异方法。这些变异方法都可以执行与对应的非变异方法相同的操作，但是这些方法要在原地执行操作、改变所调用的集

合，并返回 None。这 4 个非变异方法也可以使用运算符语法进行访问：分别是，S-S1、S&S1、S^S1 和 S|S1；对应的变异方法也可以使用增量赋值语法进行访问：分别是 S-=S1、S&=S1、S^=S1 和 S|=S1。在使用运算符或增量赋值语法时，两个操作数都必须是集合或者固定集合（frozenset）；不过，在调用命名方法时，参数 S1 可以是由可哈希的项目组成的任意迭代，并且其语义就像传递的参数为 set(S1) 一样。

4.8 字典运算

Python 提供了多种适用于字典的运算。由于字典是容器，因此内置的 len 函数可以将一个字典作为其单个参数，并返回字典对象中项目（键/值对）的数量。字典是可迭代的，因此可以将其传递给任何可以接受可迭代参数的函数或方法。在这种情况下，只有字典的键是以某些任意顺序迭代的。例如，对于任意字典 D，min(D)将返回 D 中的最小键。

字典成员

k in D 运算符可以检查对象 k 是否是字典 D 的一个键。如果是，则返回 True，如果不是，则返回 False。k not in D 相当于 not (k in D)。

索引字典

字典 D 中与键 k 对应的值可以使用索引来表示：D[k]。对不在字典中的键进行索引将引发一个异常。例如：

```
d = { 'x':42, 'y':3.14, 'z':7 }
d['x']                          # 42
d['z']                          # 7
d['a']                          # 引发 KeyError 异常
```

使用还不在字典中的键对一个字典进行普通赋值（例如，D[newkey]=value）是一个有效操作，将把该键和值作为一个新项目添加到字典中。例如：

```
d = { 'x':42, 'y':3.14}
d['a'] = 16                      # d 现在是{'x':42, 'y':3.14,'a':16}
```

格式为 del D[k]的 del 语句将从字典中删除键为 k 的项目。如果 k 不是字典 D 中的一个键，则 del D[k]将引发一个异常。

字典方法

字典对象提供了如表 4-5 所示的一些方法。非变异方法将返回一个结果，并且不改变其应用的对象，而变异方法可能会改变其应用的对象。在表 4-5 中，D 和 D1 表示任何字典对象，k 表示任何可哈希对象，x 表示任何对象。

表 4-5　字典对象的方法

方　法	说　明
非变异方法	
D.copy( )	返回字典的一个简化副本（该副本中的项目是字典 D 中的相同对象，但不是完全的副本）
D.has_key(k)	如果 k 是字典 D 中的一个键，则返回 True；否则，返回 Flase，相当于 k in D
D.items( )	返回一个包含字典 D 中所有项目（键/值对）的新列表
D.keys( )	返回一个包含字典 D 中所有键的新列表
D.values( )	返回一个包含字典 D 中所有值的新列表
D.iteritems( )	返回一个基于字典 D 中所有项目（键/值对）的迭代器
D.iterkeys( )	返回一个基于字典 D 中所有键的迭代器
D.itemvalues( )	返回一个基于字典 D 中所有值的迭代器
D.get(k[, x])	如果 k 是字典 D 中的一个键，则返回 D[k]；否则，返回 x（如果没有给定 x，则返回 None）
变异方法	
D.clear( )	从字典 D 中删除所有项目，使得 D 为空白字典
D.update(D1)	对于字典 D1 中的每个 k，将 D[k]设置为等于 D1[k]
D.setdefault(k[, x])	如果 k 是字典 D 中的一个键，则返回 D[k]；否则，将 D[k]设置为等于 x 并返回 x
D.pop(k[, x])	如果 k 是字典 D 中的一个键，则删除并返回 D[k]；否则，返回 x（如果没有给定 x，则引发一个异常）
D.popitem( )	删除并返回一个任意项目（键/值对）

Items、keys 和 values 方法都可以按照任意顺序返回结果。如果开发者调用了这几个方法，但没有对字典本身进行任何更改，则这些方法返回结果的顺序都是相同的。Iteritems、iterkeys 和 itervalues 方法可以返回等同于列表的迭代器（参见第 4.10 节中介绍的迭代器）。迭代器比列表消耗较少的内存，但是在对任意一个该字典的迭代器执行迭代操作时，不能修改字典中键的集合（也就是说，不能添加或删除键）。对 items、keys 或 values 返回的列表进行迭代操作就没有这样的限制。直接对字典 D 进行迭代操作就像对 D.iterkeys( )进行迭代一样。

Popitem 方法可以用于对字典进行破坏性的迭代。Items 和 popitem 都以键/值对的方式返回字典项目，但是使用 popitem 可以消耗较少的内存，因为 popitem 不依赖于一个单独的项目列表。当开发者想要的是在执行循环操作的同时“消耗掉”字典（减少字典中的项目）的话，内存的节省使得该习惯用法可以用来对巨大的字典执行循环操作。

Setdefault 方法返回与 get 方法相同的结果，但是如果 k 不是字典 D 中的键，则 setdefault 也会对将 D[k]绑定为值 x 产生影响。同样地，pop 方法返回与 get 方法相同的结果，但是如果 k 是字典 D 中的键，则 pop 也会对删除 D[k]产生影响（在没有指定 x，并且 k 不是字典 D 中的键时，get 将返回 None，但是 pop 将引发一个异常）。

在 Python 2.4 中，update 方法也可以接受一个可迭代的键/值对作为可选择的参数，而不是映射，并且可以只接受关键字参数，而不是其唯一的位置参数，或者除了接受其唯一的位置参数之外，还接受关键字参数；update 方法的语义与在调用内置的 dict 类型时传递这样的参数的语义相同，参见第 4.2 节。

4.9 print 语句

打印（print）语句表示为关键字 print 后面带零个或多个使用逗号分隔的表达式。print 语句是一种以文本格式输出信息的便捷和简单的方法，大多数情况下用于调试目的。print 语句会自动在表达式之间输出一个空格，并在最后一个表达式后面输出\n，除非最后一个表达式后面带一个拖尾的逗号（,）。下面是 print 语句的一些示例：

```
letter = 'c'
print "give me a", letter, "..."       # 打印: give me a c...
answer = 42
print "the answer is:", answer          # 打印: the answer is: 42
```

print 语句的输出的目的地是文件和类文件对象，也就是 sys 模块的 stdout 属性的值（参见第 8.3 节。如果开发者想要将结果从特定的 print 语句输出到特殊的文件对象 f（该对象必须是打开可写的），可以使用下面的特殊语法：

```
print >>f, rest of print statement
```

（如果 f 是 None，则其目的地是 sys.stdout，就像没有带>>f 一样）。开发者还可以使用文件对象的 write 或 writelines 方法，参见第 10.3 节。不过，print 的使用非常简单，在通常情况下，所有要做的就是 print 特别支持的简单输出策略，因此简单是很重要的。最常见的情况就是，为了进行调试，开发者可能会临时将这种类型的简单输出语句添加到程序中。第 10.9 节给出了一些有关使用 print 语句的建议和例子。

4.10 控制流语句

程序的控制流（control flow）是程序的代码执行的顺序。Python 程序的控制流是通过条件语句、循环和函数调用来管控的（本节将介绍 if 语句，for 和 while 循环；第 4.11 节中将介绍函数）。引发和处理异常也会影响控制流；第 6 章将介绍异常。

if 语句

通常，只有在某些条件满足时才需要执行某些语句，或者根据几个互斥条件选择要执行的语句。Python 的符合语句 if 是由 if、elif 和 else 子句组成的，可以让开发者有条件地执行语句块。下面是 if 语句的语法：

```
if expression:
    statement(s)
elif expression:
    statement(s)
elif expression:
    statement(s)
...
else:
    statement(s)
```

elif 和 else 子句是可选的。请注意，与其他语言不一样，Python 没有 switch 语句。Python 使用 if、elif 和 else 执行所有的条件处理。

下面是一个包含所有这 3 种子句的典型 if 语句：

```
if x < 0: print "x is negative"
elif x % 2: print "x is positive and odd"
else: print "x is even and non-negative"
```

当子句中有多条语句时（也就是说，子句控制一个语句块），这些语句将被放在包含该子句的关键字（也被称为该子句的标题行）的代码行之后的多个单独的逻辑行中，并基于该标题行向右缩进。当缩进返回到该子句的标题（或者更左边）的缩进位置时，这个语句块将会终止。当只有单个简单语句时，这条语句可以直接放在与子句标题相同的逻辑行的冒号（:）之后，但是也可以将这条语句放在一个单独的逻辑行上，紧跟标题行之后，并基于该标题行向右缩进。大多数 Python 程序员更喜欢单独行的风格，并将这些语句缩进 4 个空格。这种风格被认为更通用和更具可读性。

```
if x < 0:
    print "x is negative"
elif x % 2:
    print "x is positive and odd"
else:
    print "x is even and non-negative"
```

开发者可以使用任何 Python 表达式作为 if 或 elif 子句中的条件。按这种方式使用表达式被称为“在布尔型场景下”使用表达式。在布尔型场景下，任何值都可以看作是 True 或 False。正如前面提到的，任何非零数字或非空容器（字符串、元组、列表、字典和集合）可以算作是 True；零（任何数字类型）、None 和空白容器可以算作是 False。在开发者想要在布尔型场景下测试值 x 时，可以使用以下编程风格：

```
if x:
```

这是最清楚和最 Python 的格式。请不要使用以下任何一种格式：

```
if x is True:
if x == True:
if bool(x):
```

说一个表达式返回 True（意味着该表达式返回 bool 类型的值 1）和说一个表达式计算为 True（意味着表达式返回布尔型场景下为 True 的任何结果）之间有至关重要的区别。在测试一个表达式时，开发者需要关心后面的条件，而不是前面的条件。

如果 if 子句的表达式计算为 True，将会执行 if 子句后面的语句，并且整个 if 语句将会结束。否则，Python 将按顺序计算每个 elif 子句的表达式。如果存在条件计算为 True 的 elif 子句，则执行第一个这样的 elif 子句后面的语句，并且整个 if 语句将会结束。否则，如果存在 else 语句，则执行该子句后面的语句。

while 语句

Python 中的 while 语句支持重复执行由一个条件语句控制的一条语句或一个语句块。下面是 while 语句的语法：

```
while expression:
    statement(s)
```

while 语句还可以包含一个 else 子句，参见第 4.10 节，以及 break 和 continue 语句，参见第 4.10 节。

下面是 while 语句的一个典型示例：

```
count = 0
while x > 0:
    x = x // 2              # 截断除
    count += 1
print "The approximate log2 is", count
```

首先，Python 将计算 expression，也被称为循环条件。如果该条件为 False，则 while 语句结束。如果循环条件是满足的，则执行构成循环体的语句或语句块。当循环体结束执行时，Python 将再次计算循环条件以检查是否应该再执行一次迭代操作。这个处理过程将继续执行，直到循环条件为 False，此时 while 语句将会结束。

循环体必须包含最终会使循环条件为 False 的代码；否则，该循环将永不结束（除非引发异常或者循环体执行 break 语句）。如果在一个循环体中执行 return 语句，则函数体中的循环也将结束，因为在这种情况下，整个函数都将结束。

for 语句

Python 中的 for 语句支持重复执行由一个可迭代表达式控制的一条语句或一个语句块。下面是 for 语句的语法：

```
for target in iterable:
    statement(s)
```

in 关键字是 for 语句的语法的一部分，并且与 in 运算符有明显的区别，in 运算符用来测试成员关系。for 语句还可以包含一个 else 子句，参见第 4.10 节，以及 break 和 continue 语句，参见第 4.10 节。

下面是 for 语句的一个典型示例：

```
for letter in "ciao":
    print "give me a", letter, "..."
```

iterable 可以是任何一个适合于作为内置函数 iter 的一个参数的 Python 表达式，iter 函数将返回一个迭代器对象（下一节将详细介绍迭代器对象）。特别需要说明的是，任何序列都是可迭代对象。target 通常是一个标识符，可以用来命名循环的控制变量；for 语句可以按顺序连续将这个变量重新绑定到循环器中的每个项目上。对于 iterable 中的每个项目，构成循环体的语句或语句块只执行一次（除非因为引发了异常或者执行了 break 或 return 语句而导致循环结束）。请注意，由于循环体中可能包含结束循环的 break 语句，在这种情况下，开发者可能想要使用一个无界的可迭代对象，该对象永远都不会停止生成项目。

开发者还可以使用一个包含多个标识符的 target，以及拆包赋值。在这种情况下，迭代器的项目必须是可迭代的，每个迭代与 target 中的标识符具有相同数量的项目。例如，当 d 为一个字典时，下面给出了一个对 d 中的项目（键/值对）进行循环的典型方法：

```
for key, value in d.items():
    if not key or not value:        # 只保留等于 True 的键和值
        del d[key]
```

items 方法返回一个由键/值对组成的列表，因此可以使用在 target 中带有两个标识符的 for 循环以将每个项目拆包为 key 和 value。

当一个迭代器具有可变的基础对象时，不能在对其使用 for 循环期间改变该对象。例如，上一个示例不能使用 iteritems，而不是 items。iteritems 可以返回一个基础对象为 d 的迭代器，因此循环体不能改变 d（通过执行 del d[key]）。items 将返回一个列表，这样 d 就不是该迭代器的基础对象了；因此，循环体可以改变 d。特别是：

- 在对列表循环时，不要插入、添加或删除项目（重新绑定现有索引的项目即可）；
- 在对字典循环时，不要添加或删除项目（重新绑定现有键的值即可）；
- 在对集合循环时，不要添加或删除项目（不允许任何改变）。

控制变量可以在循环体中被重新绑定，但是在循环的下一次迭代中将再次被重新绑定到迭代器中的下一个项目上。如果迭代器不生成项目，则循环体根本就不会执行。在这种情况下，控制变量不能以任何方式通过 for 语句绑定和重新绑定。如果迭代器生成

至少一个项目，则在循环语句终止时，控制变量仍然绑定到循环语句绑定的最后一个值上。只要 someseq 不是空白的，则下面的代码是正确的：

```
for x in someseq:
    process(x)
print "Last item processed was", x
```

迭代器

迭代器（iterator）是一个对象 i，可以不使用任何参数调用 i.next()。i.next 将返回迭代器 i 的下一个项目，或者在迭代器 i 没有更多项目时，引发一个 StopIteration 异常。在开发者编写一个类（参见第 5.1 节）时，可以通过定义这样一个 next 方法以允许类的实例成为迭代器。大多数迭代器是通过隐式或显式调用内置函数 iter 来建立的，参见第 8.2 节。调用一个生成器也会返回一个迭代器，参见第 4.11 节。

for 语句隐含调用 iter 以获得一个迭代器。下面的语句：

```
for x in c:
    statement(s)
```

等于：

```
_temporary_iterator = iter(c)
while True:
    try: x = _temporary_iterator.next()
    except StopIteration: break
    statement(s)
```

这里_temporary_iterator 可以是当前未被使用过的任意名称。

因此，如果 iter(c)返回一个迭代器 i，并且 i.next()永远都不引发 StopIteration（无界迭代器），则 x in c 循环将永不终止（除非循环体中的语句包含适当的 break 或 return 语句，或者引发了异常）。iter(c)将按顺序调用特殊的方法 c. _ _iter_ _( )以获得并返回一个 c 的迭代器。第 5.2 节将更多地介绍特殊方法_ _iter_ _。

开发者可以在标准库模块 itertools 中找到一些构建和使用迭代器的最好方法，参见第 8.11 节。

rang 和 xrange

对一个整数序列进行循环是一个很普通的任务，因此 Python 提供了内置函数 range 和 xrange 以生成和返回整数序列。在 Python 中循环 n 次的最简单方法是：

```
for i in xrange(n):
    statement(s)
```

range(x)可以返回一个项目是从 0（包含 0）到 x（不包含 x）的连续整数的列表。range(x, y)可以返回一个项目是从 x（包含 x）到 y（不包含 y）的连续整数的列表。如果 x 大于

或者等于 y，则结果为空白列表。range(x, y, step)可以返回一个从 x（包含 x）到 y（不包含 y）的整数的列表，列表中每两个邻近项目之间的差就是步长（step）。如果 step 小于 0，则 range 从 x 到 y 倒计数。在 x 大于或等于 y，并且 step 大于 0，或者在 x 小于或等于 y，并且 step 小于 0 时，range 将返回空白列表。当 step 等于 0 时，range 将引发一个异常。

range 可以返回一个可以用于所有目的普通列表对象，而 xrange 将返回一个特殊目的的对象，尤其适用于像前面显示的 for 语句一样的迭代操作（不幸地是，为了保持对 Python 老版本的向后兼容，xrange 并不返回一个迭代器，这在今天的 Python 中是很自然的事情；不过，开发者可以很容易获得这样一个迭代器，如果需要，可以调用 iter(xrange(...))）。xrange 返回的特殊目的对象比 range 对象返回的列表对象消耗较少的内存（对于比较大的范围而言，消耗的内存显得更少），但是对特殊目的对象执行循环操作的开销要略微高于对列表执行循环的开销。如果不考虑性能和内存消耗问题，在任何可以使用 xrange 的位置都可以使用 range，如果考虑这些问题，则不这样。例如：

```
>>> print range(1, 5)
[1, 2, 3, 4]
>>> print xrange(1, 5)
xrange(1, 5)
```

其中，range 将返回一个非常完美的普通列表，其显示结果非常正常，但是 xrange 将返回一个特殊目的对象，将显示为其自身的特殊方式。

列表推导

for 循环通常可以用来查看可迭代对象中的每个项目，并使用一个表达式对某些或所有项目进行计算，通过添加计算的结果来建立一个新列表。这种表达式的形式被称为列表推导（list comprehension），可以帮助开发者简明直接地编写这种通用习惯用法。由于列表推导是一个表达式（而不是一个语句块），开发者可以在需要一个表达式的位置（例如，作为函数调用和 return 语句中的一个参数，或者作为某些其他表达式的子表达式）使用列表推导。

列表推导具有以下语法：

```
[ expression for target in iterable lc-clauses ]
```

target 和 iterable 的使用与在普通 for 语句中的使用一样。如果 expression 指示一个元组，则必须将其包含在圆括号中。

lc-clauses 是一个由零个或更多子句组成的序列，每个子句都使用以下形式之一：

```
for target in iterable
if expression
```

列表推导的每个 for 子句中的 target 和 iterable 与普通 for 语句中的 target 和 iterable 具有

相同的语法和含义，并且列表推导的每个 if 子句中的 expression 也与普通 if 语句中的 expression 具有相同的语法和含义。

列表推导相当于通过重复调用结果列表的 append 方法建立相同列表的 for 循环。例如（为了更清楚，将列表推调的结果赋值给一个变量）：

```
result1 = [x+1 for x in some_sequence]
```

这个列表推导与 for 循环相同：

```
result2 = []
for x in some_sequence:
    result2.append(x+1)
```

下面是一个使用 if 子句的列表推导：

```
result3 = [x+1 for x in some_sequence if x>23]
```

这个列表推导与包含一个 if 语句的 for 循环相同：

```
result4 = []
for x in some_sequence:
    if x>23:
        result4.append(x+1)
```

下面是一个使用 for 子句的列表推导：

```
result5 = [x+y for x in alist for y in another]
```

这个列表推导与内部嵌套了另一个 for 循环的 for 循环相同：

```
result6 = []
for x in alist:
    for y in another:
        result6.append(x+y)
```

正如这些示例所示的，列表推导中的 for 和 if 的顺序与对等的循环中的顺序相同，但是在列表推导中，嵌套仍然是隐含的。

break 语句

break 语句只允许出现在循环体内。在执行 break 语句时，循环将会终止。如果一个循环被嵌套在另外的循环之内，这个循环中的 break 语句只会终止最内层嵌套的循环。在实际使用中，break 语句通常位于循环体的 if 语句的某些子句中，这样 break 才会有条件地执行。

break 语句的一个通常方法就是在循环中实现，break 语句用来决定该循环体是否只有在每个循环迭代的中间进行循环操作：

```
while True:                                    # 这个循环永远都不能自然终止
    x = get_next()
    y = preprocess(x)
```

```
    if not keep_looping(x, y): break
    process(x, y)
```

continue 语句

continue 语句只允许出现在循环体内。在执行 continue 时，将会终止循环体的当前迭代操作，并使用该循环的下一个迭代继续执行。在实际使用中，continue 语句通常位于循环体的 if 语句的某些子句中，这样 continue 才会有条件地执行。

有时候，continue 语句可以替换循环中嵌套的 if 语句。例如：

```
for x in some_container:
    if not seems_ok(x): continue
    lowbound, highbound = bounds_to_test()
    if x<lowbound or x>=highbound: continue
    if final_check(x):
        do_processing(x)
```

不使用 continue 进行有条件处理的相同代码：

```
for x in some_container:
    if seems_ok(x):
        lowbound, highbound = bounds_to_test()
        if lowbound <= x < highbound:
            if final_check(x):
                do_processing(x)
```

这两个版本的代码功能是相同的，因此，使用哪一个只是个人喜好和风格的问题。

循环语句中的 else 子句

while 和 for 语句可以有选择地使用拖尾的 else 子句，else 子句之后的语句或语句块将在循环自然终止时（在 for 迭代器结束时，或者在 while 的循环条件变为 False 时）执行，但是不能在循环提前终止时（因为执行 break 和 return，或者引发异常）执行。当一个循环包含一个或多个 break 语句时，通常需要检查该循环是自然终止还是提前终止。开发者可以在循环中使用 else 子句达到这个目的：

```
for x in some_container:
    if is_ok(x): break                    # 项目 x 有效，终止循环
else:
    print "Warning: no satisfactory item was found in container"
    x = None
```

pass 语句

Python 的复合语句体不能是空白的；至少必须包含一个语句。在程序从语法结构上要求包含一个语句，但是又没有什么要执行的操作时，开发者可以使用 pass 语句作为占位符，该语句不执行任何动作。下面是在条件语句中使用 pass 语句的一个示例，该条

件语句可以作为用来测试互斥条件的某种复杂逻辑的一部分：

```
if condition1(x):
    process1(x)
elif x>23 or condition2(x) and x<5:
    pass                                    # 在这种情况下，不执行任何操作
elif condition3(x):
    process3(x)
else:
    process_default(x)
```

请注意，在其他不同的空白 def 或 class 语句体中，可以使用 docstring，参见第 4.11 节中介绍的文档字符串；如果开发者编写了一个 docstring，则不需要添加一个 pass 语句，尽管如果开发者愿意，仍然可以这样做。

try 和 raise 语句

Python 支持使用 try 语句进行异常处理，该语句包含 try、except、finally 和 else 子句。一个程序可以使用 raise 语句显式引发一个异常。第 6.2 节中将详细介绍异常，在引发异常时，程序的正常控制流将会停止，同时 Python 将会寻找一个适当的异常处理程序。

with 语句

在 Python 2.5 中，已经添加了更具可读性的 with 语句作为 try/finally 语句的另一个选择。第 6.1 节中介绍的 with 语句将详细介绍 with 语句。

4.11 函数

典型的 Python 程序中的大多数语句都被分组和组织成了函数（放在函数体内的代码要比放在模块的顶级更快，参见第 18.4 节中介绍的避免使用 exec 和 from...import*，因此，这些极好的实践经验就是将大多数代码放到函数中的原因）。函数（function）是根据要求执行的一组语句。Python 提供了许多内置函数并允许程序员定义自己的函数。请求执行一个函数的操作被称为函数调用（function call）。在调用一个函数时，可以向函数传递指定数据的参数，函数将根据这些参数执行计算。在 Python 中，函数总是会返回一个结果值，可以是 None 或者表示计算结果的一个值。class 语句中定义的函数也被称为方法（method）。有关方法的一些问题请参见第 5.1 节；不过，本节中介绍的常用函数也可以应用于方法。

在 Python 中，函数是可以像其他对象那样进行处理的对象（值）。因此，开发者可以将函数作为另一个函数调用中的一个参数。同样地，函数可以返回另一个函数作为调用的结果。与任何其他对象一样，函数也可以绑定到变量、容器中的一个项目，或者对象的一个属性上。函数还可以是字典中的键。例如，如果需要快速找到一个函数的反函数，可以定义一个字典，其键和值也都是函数，然后让其成为双向字典即可。下面

是这个方法的一个小例子，使用了来自模块 math 的一些函数，参见第 15.1 节：

```
inverse = {sin:asin, cos:acos, tan:atan, log:exp}
for f in inverse.keys(): inverse[inverse[f]] = f
```

在 Python 中，说到函数都是普通对象，实际上是说函数是第一类（first-class）对象。

def 语句

def 语句是定义函数的最常用方法。def 是一个具有以下语法的单子句复合语句：

```
def function-name(parameters):
    statement(s)
```

function-name 是一个标识符。在执行 def 时，这个标识符就是一个绑定（或者重新绑定）到该函数对象的一个变量。

parameters 是一个可选的标识符列表，被称为形式参数（formal parameters）或者参数，在函数被调用时，这些参数将被绑定到实际输入的值上。在最简单的情况下，函数也可以没有任何形式参数，这意味着该函数在调用时不需要取得任何参数。在这种情况下，函数定义中的 function-name 后面为一个空白圆括号。

当函数需要取得参数时，parameters 包含一个或多个由逗号（,）分隔的标识符。在这种情况下，每次调用该函数时都需要提供值，也被称为实际参数（arguments），对应于函数定义中列出的形式参数。这些参数都是函数的本地变量（本节后面将详细介绍），并且每次调用函数时将把这些本地变量绑定到调用程序提供为实际参数的值上。

非空语句序列被称为函数体（function body），在执行 def 语句时并不执行函数体。而是在每次函数被调用时，再执行函数体。下面将简要介绍，函数体可以包含零个或多个 return 语句。

下面是一个简单函数示例，该函数将在每次被调用时返回一个两倍于其传入参数的值：

```
def double(x):
    return x*2
```

参数

只包含标识符的形式参数表示强制参数（mandatory parameter）。每次调用函数必须为每个强制参数提供对应的值（实际参数）。

在使用逗号（,）分隔的参数列表中，零个或多个强制参数后面可能会带零个或多个可选参数（optional parameter），每个可选参数具有以下语法：

```
identifier=expression
```

def 语句将计算每个这样的表达式（expression），并保存一个到该表达式的值的引用，这个值也被称为参数的默认值，是该函数对象的一个属性。当函数调用没有提供与可

选参数对应的实际参数时，该函数调用将把参数的标识符绑定到其默认值上以供该函数执行。请注意，在 def 语句执行时，将会计算每个默认值，但是在调用结果函数时，则不会计算这些默认值。特别是，这意味着只要调用程序不提供对应的实际参数，则相同的对象将以默认值绑定到可选参数上。在默认值是可变对象，并且函数体将改变该参数时，处理起来比较麻烦。例如：

```
def f(x, y=[]):
    y.append(x)
    return y
print f(23)                    # 打印: [23]
prinf f(42)                    # 打印: [23, 42]
```

第二个 print 语句将打印[23, 42]，因为第一次调用 f 时改变了 y 的默认值，y 的初始默认值是一个空白列表[]，调用后添加了一个值 23。如果开发者想要在每次使用单个参数调用 f 时，将 y 绑定到一个空白列表对象上，可以使用以下方式：

```
def f(x, y=None):
    if y is None: y = []
    y.append(x)
    return y
print f(23)                    # 打印: [23]
prinf f(42)                    # 打印: [42]
```

在参数的末尾，可以有选择地使用特殊形式*identifier1 或**identifier2，或者两者都用。如果两个形式都出现了，要将带有两个星号的形式放在后面。*identifier1 指定了对该函数的任何调用要提供任意数量的额外位置参数，而**identifier2 指定了对该函数的任何调用要提供任意数量的额外命名参数（参见第 4.11 节中对位置和命名参数的介绍）。对该函数的所有调用都将 identifier1 绑定到一个项目是额外位置参数的元组（或者是空白元组，如果没有任何项目）。同样地，identifier2 绑定到一个项目是额外命名参数的名称和值的字典（或者是空白字典，如果没有任何项目）。下面是一个可以接受任意数量的位置参数并返回其和的函数：

```
def sum_args(*numbers):
    return sum(numbers)
print sum_args(23, 42)         # 打印: 65
```

一个函数的参数数量，加上这些参数的名称、强制参数的数量，以及是否出现了单星号和/或双星号的特殊形式的信息（在参数的末尾），共同形成了被称为函数签名（signature）的规范。函数签名定义了调用函数的方法。

函数对象的属性

def 语句设置了函数对象的某些属性。属性 func_name 也可以通过__name__进行访问，是指 def 语句中作为函数名的标识符字符串。在 Python 2.3 中，这是一个只读属性（试图重新绑定或解除绑定该属性将引发一个运行时异常）；在 Python 2.4 中，可以将该属

性重新绑定到任何字符串值上，但是试图解除绑定该属性将引发一个异常。可以自由重新绑定或解除绑定的属性 func_defaults 是指由可选参数的默认值组成的元组（或者空白元组，如果该函数没有可选参数）。

文档字符串

另一个函数属性是文档字符串（documentation string），也被称为 docstring。开发者可以使用或重新绑定函数的 docstring 属性为 func_doc 或__doc__。如果函数体的第一个语句是字符串字面常量，编译器将把该字符串绑定为该函数的 docstring 属性。相同的原则也可以应用于类（参见第 5.1 节中介绍的类文档字符串）和模块（参见第 7.1 节中介绍的模块文档字符串）。在大多数情况下，Docstrings 会跨越多个物理行，因此通常以三重引用字符串字面常量的形式指定该属性。例如：

```
def sum_args(*numbers):
    '''Accept arbitrary numerical arguments and return their sum.
    The arguments are zero or more numbers.  The result is their sum.'''
    return sum(numbers)
```

文档字符串应该是编写的任何 Python 代码的一部分。文档字符串的角色类似于任何编程语言中的注释，但是其适用性更广泛，因为文档字符串还可以在运行时使用。开发环境和工具可以使用 docstrings 形式的函数、类和模块对象以提醒程序员如何使用这些对象。Doctest 模块（参见第 18.1 节）可以很容易地检查 docstrings 中显示的示例代码是精确和正确的。

要想让 docstrings 尽可能有用，必须遵循几个简单的习惯用法。docstring 的第一行应该是该函数目的的简要说明，以大写字母开始，以句点结束。这一行不需要提到函数的名称，除非这个名称恰好是可以作为函数操作的一个好的简要说明部分中出现的一个自然语言词组。如果 docstring 是多行的，则第二行应该是空白的，并且后面的行应该构成由空白行分隔的一个或多个段落，说明该函数的参数、前提条件、返回值和负面影响（如果有的话）。可以有选择地将进一步的解释、文献引用和用法示例（需要使用 doctest 进行检验）放在 docstring 的末尾。

函数对象的其他属性

除了预定义的属性，函数对象可以有其他任意属性。要想创建函数对象的属性，在 def 语句执行之后，可以将一个值绑定到赋值语句中适当的属性引用上。例如，计算一个函数被调用了多少次的函数：

```
def counter():
    counter.count += 1
    return counter.count
counter.count = 0
```

请注意，这不是最通常的用法。更常见的是，当开发者想要将一些状态（数据）和一些行为（代码）组合在一起时，必须使用第 5 章介绍的面向对象机制。但是，可以将

任意属性与一个函数关联的功能有时候也是很有用的。

return 语句

Python 中的 return 语句只允许在函数体内出现，并且可以有选择地带一个表达式。在执行 return 语句时，函数将会终止，并且表达式的值就是该函数的值。如果函数是达到了函数体的末尾，或者执行了一个没有表达式的 return 语句（或者，当然还包括执行 return None）而终止，则该函数将返回 None。

作为一种好的风格，开发者不应该在函数体的末尾编写一个不带表达式的 return 语句。如果函数中的某些 return 语句带有表达式，则所有 return 语句都应该带有表达式。只能显式地编写 return None 来满足这种风格的要求。Python 并不强制使用这些格式上的习惯用法，但是如果遵循这些习惯用法，代码会更清楚并更具可读性。

调用函数

函数调用是一个使用以下语法的表达式：

```
function-object(arguments)
```

function-object 可以是函数对象（或其他可调用对象）的任何引用；最常见的是，function-object 就是该函数的名称。圆括号表示函数调用操作本身。在最简单的情况下，arguments 是一连串逗号（,）分隔的零个或多个表达式，为该函数的对应参数提供值。在执行函数调用时，这些参数将绑定到参数的值上、执行函数体，并且这个函数调用的表达式的值就是该函数返回的值。

请注意，编写一个函数（或者其他可调用对象）并没有调用该函数。要想不带任何参数地调用一个函数（或者其他对象），必须在该函数名的后面使用（）。

参数传递的语义

从传统上讲，Python 中的所有参数都是通过值（value）来传递的。例如，如果传递一个变量作为参数，Python 将向该函数传递该变量当前对应的对象（值），而不是“变量本身”。因此，函数不能再重新绑定调用程序的变量。但是，如果开发者将一个可变对象传递为一个参数，该函数可能会对这个对象进行更改，因为 Python 传递的是该对象本身，而不是一个副本。重新绑定一个变量和改变一个对象的值是完全不同的概念。例如：

```
def f(x, y):
    x = 23
    y.append(42)
a = 77
b = [99]
f(a, b)
print a, b                    # 打印: 77 [99, 42]
```

print 语句显示，a 仍然绑定为 77。函数 f 中将参数 x 重新绑定为 23 对 f 的调用程序不产生任何影响，特别是，对调用程序变量的绑定也没有任何影响，该变量用来将 77 传递为该参数的值。不过，print 语句也显示了 b 现在绑定为[99, 42]。像在调用前一样，b 仍然绑定到相同的列表对象上，但是该对象已经被改变了，因为函数 f 将 42 添加到该列表对象上。在这两种情况中，函数 f 没有改变调用函数的绑定关系，函数 f 也不能改变数字 77，因为数字是不可变的。但是，f 可以更改一个列表对象，因为列表对象是可变的。在这个示例中，函数 f 通过调用列表对象的 append 方法更改了调用函数传递为函数 f 的第二个参数的列表对象。

参数的类别

只有表达式的参数被称为位置参数（positional argument）。每个位置参数按照其在函数定义中的位置（顺序）为对应的参数提供值。

在函数调用中，零个或多个位置参数后面可能会带有零个或多个命名参数（named argument），每个命名参数使用以下语法：

```
identifier=expression
```

identifier 必须是该函数的 def 语句中使用的一个参数名称。expression 为这个名称的参数提供了值。大多数内置函数不接受命名参数，开发者只能使用位置参数调用这样的函数。但是，所有使用 Python 编写的普通函数都可以接受命名参数和位置参数，因此，开发者可以使用不同的方法调用这些函数。

函数调用必须通过位置参数或命名参数为每个强制参数提供一个确切的值，为每个可选参数提供零个或一个值。例如：

```
def divide(divisor, dividend):
    return dividend // divisor
print divide(12, 94)                              # 打印: 7
print divide(dividend=94, divisor=12)             # 打印: 7
```

可以看出，对 divide 函数的两次调用的效果是相等的。在开发者认为标识每个参数的角色并控制参数的顺序可以增强代码的清晰度时，可以传递命名参数以获得更好的可读性。

命名参数通常可以用于将某些可选参数绑定到特定值上，并让其他可选参数使用默认值：

```
def f(middle, begin='init',end='finis'):
    return begin+middle+end
print f('tini', end='')                           # 打印: inittini
```

感谢命名参数 end=' '，调用函数可以指定一个值（也就是空白字符串 ' '）作为函数 f 的第三个参数 end，并仍让函数 f 的第二个参数 begin 使用其默认值，也就是字符串 'init'。

在函数调用的参数后面，开发者可以可选地使用一个或两个特殊形式*seq 和**dct。如果这两种形式都被使用了，带有双星的形式必须在后面。*seq 可以将 seq 中的参数作为位置参数（函数调用使用通常的语法给定的普通位置参数，如果有的话）传递给该函数。seq 可以是任何可迭代对象。**dct 可以将 dct 中的项目作为命名参数传递给该函数，其中 dct 必须是所有键都为字符串的字典。每个项目的键都是一个参数的名称，该项目的值则是这个参数的值。

有时候，在参数都使用类似的形式时，开发者可能想要传递一个形式为*seq 或**dct 的参数，参见第 4.11 节。例如，使用本节中定义的函数 sum_args（这里也显示了该函数），开发者可能想要打印字典 d 中所有值的和。使用*seq，这是很简单的：

```
def sum_args(*numbers):
    return sum(numbers)
print sum_args(*d.values())
```

（当然，在这种情况下，print sum(d.values( )) 可能会更简单和更直接！）

但是，开发者还可以在调用一个没有在其参数中使用对应形式的函数时，传递形式为*seq 或**dct 的参数。当然，在这种情况下，必须确认可迭代的 seq 包含正确数量的项目，或者字典 dct 分别使用正确的名称作为其键；否则，该调用操作将引发一个异常。

命名空间

函数的参数，加上函数体中绑定的所有变量（通过赋值或者其他绑定语句，比如 def），共同构成了该函数的本地命名空间（local namespace），也被称为本地范围（local scope）。所有这些变量也被称为该函数的本地变量（local variable）。

不是本地变量的变量也被称为全局变量（global variable）（在缺乏内嵌函数定义时，接下来将介绍内嵌函数）。全局变量都是模块对象的属性，参见第 7.1 节中介绍的模块对象的属性。在函数的本地变量与全局变量具有相同的名称时，则函数体内的这个名称指的是本地变量，而不是全局变量。我们可以将这称之为本地变量在整个函数体内隐藏了相同名称的全局变量。

global 语句

在默认情况下，函数体内绑定的任何变量都是该函数的本地变量。如果一个函数需要重新绑定某些全局变量，则该函数的第一个语句必须是：

```
global identifiers
```

这里，identifiers 是一个或多个使用逗号（,）分隔的标识符。global 语句中列出的标识符指的是该函数需要重新绑定的全局变量（也就是，模块对象的属性）。例如，第 4.11 节中介绍的其他函数 counter 可以使用 global 和全局变量，而不是该函数对象的属性来实现：

```
_count = 0
def counter():
    global _count
    _count += 1
    return _count
```

如果没有 global 语句，counter 函数将引发一个 UnboundLocal-Error 异常，因为_count 将成为一个未初始化（无界）的本地变量。尽管 global 语句实现了这种编程方式，但是这种风格通常并不优雅，也是不适当的。正如前面提到的，在开发者想要将某些状态和行为组合在一起时，通常最好使用第 5 章介绍的面向对象的机制。

如果函数体只是要使用一个全局变量（包括改变绑定到这个变量的对象，如果该对象是可变的），可以不使用 global。只有在函数体重新绑定一个全局变量（通常就是为变量的名称赋值）时，才使用 global 语句。作为一种编程风格，除非确实需要，请不要使用 global 语句，出现的 global 语句将导致程序的阅读者假定该语句用于某些特殊的目的。特别需要说明的是，除了作为函数体的第一个语句之外，永远都不要使用 global 语句。

内嵌函数和嵌套范围

函数体中的 def 语句定义了一个内嵌函数（nested function），并且函数体包含 def 的函数被称为内嵌函数的外函数（outer function）。嵌套的函数体中的代码可以访问（但不是重新绑定）一个外部函数的本地变量，也被称为这个内嵌函数的自由变量（free variable）。

让一个内嵌函数可以访问一个变量的最简单方法通常并不依赖于嵌套的范围，而是将这个值显式传递为该函数的一个参数。如有必要，也就是在内嵌函数是通过使用该值作为一个可选参数的默认值来定义时，该参数的值可以被绑定。例如：

```
def percent1(a, b, c):
    def pc(x, total=a+b+c): return (x*100.0) / total
    print "Percentages are:", pc(a), pc(b), pc(c)
```

下面是使用嵌套范围的相同功能：

```
def percent2(a, b, c):
    def pc(x): return (x*100.0) / (a+b+c)
    print "Percentages are:", pc(a), pc(b), pc(c)
```

在这种特殊情况下，percent1 有一个很小的优点：a+b+c 的计算只出现了一次，而 percent2 的内函数（inner function）pc 重复计算了三次。不过，如果外函数（outer function）。不过，如果外函数在对内嵌函数的调用中重新绑定其本地变量，则重复计算是必需的。因此需要对这两种方案都有所了解，并选择最适当的方案。

访问来自外部本地变量的值的内嵌函数也被称为闭包（closure）。下面的示例显式了如何建立一个闭包：

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

对于第 5 章中介绍的面向对象机制就是融合数据和代码的最佳方法的一般性原则而言，闭包是一个例外。在开发者需要特别构造可调用对象，并且在对象构造时固定某些参数时，闭包要比类更简单，也更有效率。例如，make_adder(7)的结果是一个接受单个参数并向该参数加 7 的函数。返回一个闭包的外函数是一个可以制造由某些参数（比如上一个示例中参数 augend 的值）区分的函数家族的成员函数的“工厂”，并且经常可以帮助开发者避免重复编程。

lambda 表达式

如果函数体是一个单独的 return expression 语句，开发者可以选择使用特殊的 lambda 表达式形式替换该函数：

```
lambda parameters: expression
```

lambda 表达式相当于函数体为单个 return 语句的普通函数的匿名函数。请注意，lambda 语法并没有使用 return 关键字。开发者可以在任何可以使用函数引用的位置使用 lambda 表达式。在开发者想要使用一个简单函数作为参数或者返回值时，使用 lambda 表达式是很方便的。下面是使用 lambda 表达式作为内置 filter 函数（参见第 8.2 节）的一个参数的示例：

```
aList = [1, 2, 3, 4, 5, 6, 7, 8, 9]
low = 3
high = 7
filter(lambda x, l=low, h=high: h>x>l, aList)    # 返回: [4, 5, 6]
```

作为另外一种选择，开发者还可以使用一个可以为函数变量命名的本地 def 语句。然后，开发者可以使用这个名称作为参数或返回值。下面是使用本地 def 语句的相同 filter 示例：

```
aList = [1, 2, 3, 4, 5, 6, 7, 8, 9]
low = 3
high = 7
def within_bounds(value, l=low, h=high):
    return h>value>l
filter(within_bounds, aList)                    # 返回: [4, 5, 6]
```

因为 lambda 表达式只是偶尔有用，许多 Python 用户更喜欢使用 def，def 要更通用一些，如果开发者为函数选择了一个比较合理的名称，会让代码具有更好的可读性。

生成器

在函数体中出现了一个或多个关键字 yield 时，可以将该函数称为生成器（generator）。

在调用一个生成器时，并不会执行该函数体。调用生成器将返回一个特殊的迭代器对象，该对象包含这个函数体、函数体的本地变量（包括函数体的参数），以及当前的执行位置，这个位置最初就是该函数的起始点。

在调用这个迭代器对象的 next 方法时，函数体将执行到下一个 yield 语句，该语句使用以下格式：

```
yield expression
```

在执行 yield 语句时，函数的执行将被“冻结”，保留执行的当前位置和未经使用的本地变量，并且 yield 后面的表达式将被返回为 next 方法的结果。当再次调用 next 方法时，函数体的执行将从其被冻结的位置开始继续执行，再次到达下一个 yield 语句。如果函数体运行结束，或者执行了 return 语句，该迭代器将引发一个 StopIteration 异常，用以表明迭代操作已经完成。生成器中的 return 语句不能包含表达式。

生成器是用来构建迭代器的一种非常方便的方法。由于使用迭代器的最常用方法是使用 for 语句对其执行循环操作，开发者通常可以使用下面的代码调用生成器：

```
for avariable in somegenerator(arguments):
```

例如，假定开发者想要获得一个由从 1 到 N，然后从 N 到 1 的数字组成的序列。可以使用生成器：

```
def updown(N):
    for x in xrange(1, N): yield x
    for x in xrange(N, 0, -1): yield x
for i in updown(3): print i                    # 打印: 1 2 3 2 1
```

下面是一个使用起来有些像内置 xrange 函数的生成器，但是该生成器将返回一个由浮点型值组成的序列，而不是一个由整数组成的序列：

```
def frange(start, stop, step=1.0):
    while start < stop:
        yield start
        start += step
```

这个 frange 函数示例只是有些类似于 xrange，因为为了简单，该函数强制使用 start 和 stop 参数，并假定 step 为正数。

生成器要比可以返回列表的函数更灵活。生成器可以构建无界的迭代器，这意味着可以返回一个无限的结果序列（只能在使用其他方法终止的循环操作中使用，例如，使用 break 语句终止循环操作）。更进一步，生成器构建的迭代器可以执行懒惰计算（lazy evaluation）：只有在需要时迭代器才计算每个连续项目，而相等函数需要提前进行所有计算，并且可能需要大量内存来保存结果列表。因此，如果开发者需要的是对一个计算的序列进行迭代的功能，通常最好的方法就是计算生成器中的序列，而不是可以返

回一个列表的函数中的序列。如果调用程序需要一个由某些有界的生成器 G(arguments) 生成的所有项目组成的列表，调用程序只需要使用以下代码：

```
resulting_list = list(G(arguments))
```

生成器表达式

Python 2.4 引入了一种更简单的方法来特别地编写简单的生成器：生成器表达式(generator expressions)，通常被称为 genexp。genexp 的语法就像列表推导（参见第 4.10 节中介绍的列表推导）一样，区别在于 genexp 被包含在圆括号（()）中，而不是方括号（[ ]）中；genexp 的语义与对应的列表推导的语义是相同的，区别在于 genexp 将生成一个迭代器，一次生成一个项目，而列表推导会在内存中产生由所有结果组成的列表（因此，在适当的时候使用 genexp 可以节省内存）。例如，在任何近代版本的 Python 中，要想计算所有一位整数的平方的和，可以使用代码 sum([x*x for x in xrange(10)])；在 Python 2.4 中，可以以更好的方法表达这种功能，代码可以编写为 sum(x*x for x in xrange(10))（基本上是相同的，只是省略了方括号），并且可以在消耗更少内存的情况下获得完全相同的结果。请注意，用来表示函数调用的圆括号还可以“有双重用途”并包含 genexp（不需使用额外的圆括号）。

Python 2.5 中的生成器

在 Python 2.5 中，生成器得到了进一步地增强，开发者可以在每个 yield 执行时，从调用函数接收一个返回的值（或者异常）。这些高级功能允许 Python 2.5 中的生成器实现完全合格的协同程序（co-routine），参见 http://www.python.org/peps/pep-0342.html。最主要的更改是，在 Python 2.5 中，yield 并不是一个语句，而是一个表达式，因此 yield 是有值的。在调用生成器的 next 方法继续执行该生成器时，对应的 yield 的值为 None。要想将一个值 x 传递到某些生成器 g 中（这样 g 可以将 x 接收为暂停的 yield 的值），而不是调用 g.next()，调用程序可以调用 g.send(x)（调用 g.send(None)就像调用 g.next()）一样。还有，在 Python 2.5 中，不带任何参数的 yield 变得合法了，相当于 yield None。

Python2.5 中生成器的其他增强功能与异常有关，参见第 6.1 节中介绍的生成器增强功能。

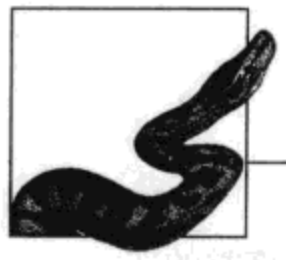
递归

Python 支持递归（也就是说，Python 函数可以调用其本身），但是对于可以递归多少层是有限制的。在默认情况下，当 Python 检测到其递归调用堆栈超过了 1000 层的深度时，将中断该递归调用并引发 RecursionLimitExceeded 异常（参见第 6.4 节）。开发者可用使用 sys 模块的 setrecursionlimit 函数更改迭代限制，参见第 8.3 节。

但是，更改递归限制也不能获得无限制的递归；绝对的最大限制取决于开发者的

程序运行的平台，特别取决于底层的操作系统和 C 运行时库，但是，通常最多也只有数千层。如果递归得太深，程序可能会崩溃。在调用了一个超过系统平台能力的 setrecursionlimit 函数之后，这种失控的递归是 Python 程序可能会导致崩溃的极少数情况之一，这种崩溃是真正的、完全的崩溃，不再有 Python 的异常机制这样的常用安全网。因此，要注意对程序进行修改，防止出现使用 setrecursionlimit 将递归限制设置得太高而引发 RecursionLimitExceed 异常的情况。最常见的情况是，开发者最好考虑查找一些方法来删除递归，更专业地讲，就是限制程序所需要的递归深度。

熟悉 Lisp、Scheme 或功能编程语言的读者必须特别注意，Python 没有实现“尾调用（tail-call）消除”的优化功能，这种功能在那些语言中是非常重要的。在 Python 中，不管递归与否，任何函数调用在时间和内存空间上都存在相同的开销，这只取决于参数的数量：不管该调用是“尾调用”还是任何其他的非尾调用，其开销都不会改变。



第5章 面向对象的 Python

Python 是一种面向对象的编程语言。与其他一些面向对象语言不同，Python 并不强迫开发者只使用面向对象程序。Python 还支持使用模块和函数进行面向过程的编程，因此开发者可以为程序的每个部分选择最适当的编程范式。通常，在开发者想要将状态（数据）和行为（代码）组合在一起，使之成为便于使用的功能包时，使用面向对象的范式是很合适的。在开发者想要使用本章介绍的 Python 的某些面向对象机制，比如继承或特殊方法时，使用面向对象的范式也是很有用的。基于模块和函数的面向过程的范式可能会更简单一些，因此，在开发者不需要用到面向对象编程的任何好处时，使用面向过程的范式更合适。使用 Python，开发者可以混和和搭配这两种编程范式。

今天的 Python 界乎这两种略微不同的对象模型之间。本章主要介绍一种新型模型，也就是新对象模型，这种模型更简单、更常见、功能更强大，并且本书建议开发者总是使用这种对象模型；在任何时候本书中说到类或者实例时，如果没有明确指定其对象模型，则表示这种新型的类或实例。但是，为了向后兼容，所有 Python 2.x 版本（对于每个版本 x）中的默认对象模型使用的都是传统的对象模型，也被称为经典或老式对象模型；几年之后，在 Python 3.0 出现之后，Python 的新型对象模型将成为默认对象模型（传统对象模型将会消失）。因此，在本章的每一节中，在介绍了新型对象模型的使用之后，还将介绍新型和传统对象模型之间小的区别，并介绍如何使用 Python 2.x 处理这两种对象模型。最后，本章还介绍了一些特殊方法（参见第 5.2 节），然后介绍了两种被称为装饰器（decorator）的高级概念（参见第 5.3 节），以及元类（metaclass），参见第 5.4 节。

5.1 类和实例

如果读者已经通过其他编程语言，比如 C++或 Java，熟悉了面向对象编程，那么读者可能已经对类和实例有了很直观的了解：类（class）是一个用户自定义类型，开发者

可以将其实例化以获得实例（instance），实例表示这种类型的对象。Python 通过其类和实例对象支持这些概念。

Python 类

Python 类是一个具有以下几个特征的 Python 对象。

- 可以像函数一样调用类对象。调用将返回另一个对象，也被称为该类的实例；这个类也被称为该实例的类型。
- 类中包含任意名称的属性，开发者绑定和引用这些属性。
- 类属性的值可以是描述符（包括函数）（参见第 5.1 节），或者普通数据对象。
- 绑定到函数的类属性也被称为该类的方法。
- 方法可以有一个特殊的、使用 Python 定义的名称，这个名称带有两个前导下划线和两个拖尾下划线。如果一个类提供了一些特殊方法，在对该类的实例进行各种类型的操作时，Python 可以隐式调用这些特殊方法。
- 可以从其他类继承类，这意味着这个类可以委托其他类对象查找该类本身没有的属性。

类的实例是一个带有任意名称的属性的 Python 对象，开发者可以绑定和引用这些属性。实例对象可以隐式地委托其类查找该实例本身没有的属性。反过来，如果有的话，一个类也可以委托查找继承该类的类。

在 Python 中，类就是对象（值），开发者可以像对其他对象那样处理类。这样，开发者可以在调用函数时传递一个类作为参数。同样地，函数可以返回一个类作为调用的结果。就像任何其他对象一样，类可以绑定到一个变量（本地或全局）、容器中的一个项目，或者对象的一个属性上。类还可以是字典的键。类就是 Python 中的普通对象，这个事实通常是通过说这些类是第一类（first-class）对象来表达的。

class 语句

class 语句是创建一个类对象最常用的方法。class 是一个使用以下语法的单子句符合语句：

```
class classname(base-classes):
    statement(s)
```

classname 是一个标识符。该标识符是一个在执行完 class 语句之后被绑定（重新绑定）到类对象的变量。

base-classes 是一个使用逗号分隔的表达式序列，这些表达式的值必须是类对象。在不同的编程语言中，这些类被命名为不同的名称；根据开发者的选择，开发者可以将这些类称之为要创建的类的基类（base）、超类（superclass）或者父类（parent）。根据开

发者熟悉的编程语言的类型，要创建的类可以被称为继承自或者派生自其基类，或者是其基类的扩展类或子类。这个类也被称之为其基类的直接子类或子孙类。

从语法上讲，base-classes 是可选的：要想表示正在创建一个没有基类的类，可以省略 base-classes 及其圆括号，将冒号直接放在类名称的右边（在 Python 2.5 中，还可以在类名称和冒号之间使用一个空白圆括号，这样具有相同的含义）。不过，由于向后兼容的原因，没有基类的类是一个老式类（除非定义了__metaclass__属性，参见第 5.4 节）。要想创建一个没有任何“真正”基类的新型类 C，可以将代码写成 class C(object):；由于所有的类型都是内置对象的子类，因此，将 object 指定为基类的值就意味着类 C 是一个新型类，而不是老式类。如果开发者的类具有的都是老式类的祖先，并且没有定义__metaclass__属性，则该类是老式类；否则，具有基类的类都是新型类（即使某些基类是新型类，而另外的一些是老式类）。

类之间的子类关系是可以传递的：如果 C1 是 C2 的子类，而 C2 是 C3 的子类，则 C1 也是 C3 的子类。内置函数 issubclass(C1, C2)可以接受两个类对象参数：如果 C1 是 C2 的子类，则返回 True；否则返回 False。任何类都可以看作是其自身的子类；因此，对于任意类 C，issubclass(C, C)将返回 True。要想了解一个类的基类是如何影响该类的功能的，参见第 5.1 节。

class 语句之后的非空语句序列被称为类体（class body）。作为 class 语句执行的一部分，类体将在 class 语句之后立即执行。在类体执行完之前，新的类对象还不存在，并且 classname 标识符也还没有被绑定（或者重新绑定）。本章第 5.4 节中提供了更多有关执行 class 语句时的情况的详细信息。

最后，请注意，class 语句并不直接创建新类的任何一个实例，而是定义了在以后调用这个新类创建实例时，所有实例共有的属性集。

类体

类的主体就是通常指定该类的属性的位置；这些属性可以是描述符对象（包括函数）或者任何类型的普通数据对象（类的属性也可以是另一个类，例如，开发者可以将一个 class 语句“嵌入”另一个 class 语句中）。

类对象的属性

通常可以通过将一个值绑定到类体中的一个标识符上来指定类对象的一个属性。例如：

```
class C1(object):
    x = 23
print C1.x                                  # 打印: 23
```

类对象 C1 包含一个名为 x 的属性，该属性被绑定为值 23，C1.x 则表示这个属性。

开发者还可以绑定或者解除绑定类体之外的类属性。例如：

```
class C2(object): pass
C2.x = 23
print C2.x                                  # 打印: 23
```

但是，如果开发者使用类体中的语句绑定，并因此创建类属性，开发者程序会更具可读性。下面将介绍，在实例被创建时，任何类属性都由该类的所有实例隐式共享。

class 语句将隐式设置某些类属性。属性__name__是 class 语句中使用的 classname 标识符字符串。属性__bases__是 class 语句中的基类的类对象的元组。例如，使用类 C1 可以创建：

```
print C1.__name__, C1.__bases__          # 打印: C1, (<type 'object'>,)
```

类还包含一个属性__dict__，这个属性是该类的字典对象，被用来保存所有其他属性。对于任何类对象 C、任何对象 x，以及任何标识符 S（除了__name__、__bases__和__dict__），C.S=x 等于 C.__dict__['S']=x。例如，再次用到刚才创建的类 C1：

```
C1.y = 45
C1.__dict__['z'] = 67
print C1.x, C1.y, C1.z                   # 打印: 23, 45, 67
```

在类体中创建的属性与通过在类体外为一个属性赋值，或者通过在类体外显式绑定 C.__dict__中的一个条目而创建的类属性之间并没有区别。

对于包含在类体中的语句，要想引用该类的属性，必须使用属性的简单名称，而不是完整名称。例如：

```
class C3(object):
    x = 23
    y = x + 22                                  # 必须只使用 x，而不是 C3.x
```

但是，在类体中定义的方法中的语句中，要想引用类的属性，必须使用完整名称，而不是简单名称。例如：

```
class C4(object):
    x = 23
    def amethod(self):
        print C4.x                              # 必须使用 C4.x，而不是 x
```

请注意，属性引用（也就是像 C.S 这样的表达式）具有比那些属性绑定更丰富的语法。本章第 5.1 节中详细介绍了这些引用。

类体中的函数定义

大多数类体包含 def 语句，由于函数（在当前的上下文环境下，称之为方法）对大所属类对象而言是非常重要的属性。类体中的 def 语句遵循第 4.11 节中给出的规则。另外，类体中定义的方法都有一个强制的第一参数，通常被命名为 self，该参数对应于要对其调用方法的实例。self 参数在方法调用中起着特殊的作用，参见第 5.1 节。

下面是一个包含方法定义的类的实例：

```
class C5(object):
    def hello(self):
        print "Hello"
```

一个类可以定义多个特殊方法（名称中包含两个前导和两个拖尾下划线的方法），这些方法指的是有关其实例的特殊操作。本章第 5.2 节中将详细介绍特殊方法。

类私有变量

当类体中的一个语句（或者类体中的一个方法）使用一个以两个下划线（但是不以下划线结束），比如__ident，Python 编译器将把该标识符隐式更改为_classname__ident，classname 是该类的名称。这样可以让一个类的属性、方法、全局变量和其他目标对象使用“私有”名称，减少意外在其他位置使用重复名称的风险。

按惯例，所有以单个下划线开始的标识符表示对于他们被绑定的范围而言是私有的，不管这个范围是不是一个类。Python 编译器并不强制执行这个私有惯例；而是取决于 Python 程序员是否遵循这个习惯用法。

类文档字符串

如果类体中的第一个语句是一个字符串字面常量，编译器将把该字符串绑定为这个类的文档字符串属性。这个属性被命名为__doc__，并且被称为这个类的 docstring。参见第 4.11 节中介绍的文档字符串以了解更多有关 docstrings 的信息。

描述器

描述器（descriptor）是其类可以提供一个名为__get__的特殊方法的任何新型对象。作为类属性的描述器可以控制对这个类的实例的属性进行访问和设置的语法。粗略地讲，就是在开发者访问一个实体属性时，Python 可以通过对相应的描述器调用__get__方法获得该属性的值，如果这个值确实存在。要想获得更多详细信息，参见第 5.1 节。

覆盖和非覆盖描述符

如果一个描述符的类还可以提供一个名为__set__的特殊方法，则这个描述符被称为覆盖描述符（overriding descriptor）（或者被称为数据描述器，一个老式和有些令人混淆的术语）；如果该描述器的类只提供__get__，而不提供__set__，则该描述器被称为非覆盖描述符（nonoverriding descriptor）（或者非数据描述器）。例如，函数对象的类可以提供__get__，而不提供__set__；因此，函数对象是非覆盖描述符。粗略地讲，在使用对应的覆盖描述符为一个实例属性赋值时，Python 可以通过对该描述符调用__set__方法来设置该属性的值。要想获得更详细的信息，参见第 5.1 节中介绍的实例对象的属性。

老式类可以包含描述器，但是老式类中的描述器总是作为非覆盖描述器使用（如果有

__set__方法，该方法将被忽略）。

实例

要想创建一个类的实例，可以调用类对象，就像该对象是一个函数一样。每个调用都将返回一个类型为该类的新实例：

```
anInstance = C5()
```

开发者可以调用内置函数 isinstance(I, C)，并使用一个类对象作为参数 C。如果对象 I 是类 C 或类 C 的任何子类的一个实例，则 isinstance 将返回 True。否则，isinstance 将返回 False。

__init__

当一个类定义或继承了一个名为__init__的方法时，调用该类对象将对新实例隐式执行__init__方法以执行任何需要的与实例相关的初始化。该调用中传递的参数必须对应于__init__的参数，除了参数 self。例如，考察下面的类：

```
class C6(object):
    def __init__(self, n):
        self.x = n
```

下面是如何创建类 C6 的一个实例：

```
anotherInstance = C6(42)
```

正如类 C6 所示的，__init__方法通常包含绑定实例属性的语句。__init__方法不能返回一个值；否则，Python 将引发一个 TypeError 异常。

__init__的主要目的就是绑定，并因此创建新创建的实例的属性。开发者还可以绑定或者解除绑定__init__之外的实例属性，接下来将会看到这一点。不过，如果开发者最初就使用__init__方法中的语句绑定一个类实例的所有属性，代码会更具可读性。

在缺少__init__时，开发者调用该类时不能带参数，并且新生成的实例没有特定属性。

实例对象的属性

在开发者创建了一个实例之后，就可以使用句点（.）操作符访问其属性（数据和方法）了。例如：

```
anInstance.hello()                          # 打印: Hello
print anotherInstance.x                     # 打印: 42
```

本章第 5.1 节中将详细介绍像 Python 中那些具有非常丰富的语法的属性引用。

开发者可以通过将一个值绑定到一个属性引用为实例对象提供任意一个属性。例如：

```
class C7(objeet): pass
z = C7()
```

```
z.x = 23
print z.x                                    # 打印: 23
```

现在，实例对象 z 有了一个名为 x 的属性，绑定到值 23 上，并且 z.x 表示这个属性。请注意，如果出现了__setattr__特殊方法，将会截断每个试图绑定一个属性的操作（本章第 5.2 节中将介绍__setattr__方法）。此外，对于一个新型实例，如果开发者试图绑定一个属性，并且该属性的名称对应于该实例的类中的一个覆盖描述符，则该描述符的__set__方法将截断这个绑定操作。在这种情况下，语句 z.x=23 将执行 type(z).x.__set__(z, 23)（老式实例将忽略其类中找到的描述符的覆盖特性，也就是说，这些实例不会调用其__set__方法）。

创建一个实例将隐式设置两个实例属性。对于任何实例 z，z.__class__是 z 所属的类对象，而 z.__dict__是 z 用来保存其他属性的字典。例如，对于实例 z，只需要创建：

```
print z.__class__.__name__, z.__dict__       # 打印: C7, {'x':23}
```

开发者可以重新绑定（而不是解除绑定）任意一个或者这两个属性，但是很少必须这样做。一个新型实例的__class__只能重新绑定到一个新型类上，而一个老式实例的__class__只能重新绑定到一个老式类上。

对于任何实例 z、任何对象 x 和任何标识符 S（除了__class__和__dict__），z.S=x 等于 z.__dict__['S']=x（除非__setattr__特殊方法，或者覆盖描述符的__set__特殊方法截断绑定操作）。例如，再次用到刚才创建的实例 z：

```
z.y = 45
z.__dict__['z'] = 67
print z.x, z.y, z.z                          # 打印: 23, 45, 67
```

通过为属性赋值或者通过显式绑定 z.__dict__中的一个条目在__init__中创建的实例属性之间并没有什么区别。

工厂函数的习惯用法

一个很常见的任务就是根据某些条件创建不同类的实例，或者如果已经有一个实例可供使用，则避免创建一个新实例。一个最常见的误解就是这样的需求可以通过让__init__返回一个特殊对象来满足，但是，这样的方法是绝对不可能实现的：当__init__返回一个不同于 None 的任何值时，Python 将引发一个异常。实现灵活创建对象的最好方法就是通过使用一个普通函数，而不是直接调用该类对象。用于这个目的的函数被称为工厂函数（Factory-function）。

调用工厂函数是一种灵活的方法：函数可能会返回一个已有的可重用实例，或者通过调用适当的任何类创建一个新实例。假定开发者有两个几乎可互换的类（SpecialCase 和 NormalCase），并且想要灵活生成任意一个类，这取决于一个参数。下面的 appropriateCase 工厂函数正好可以让开发者这样做（本章第 5.1 节中将介绍 self 参数的

功能)：

```
class SpecialCase(object):
    def amethod(self): print "special"
class NormalCase(object):
    def amethod(self): print "normal"
def appropriateCase(isnormal=True):
    if isnormal: return NormalCase()
    else: return SpecialCase()
aninstance = appropriateCase(isnormal=False)
aninstance.amethod()                    # 正如想要的那样，打印 "special"
```

__new__

每个新型类都有（或者继承了）一个名为__new__的静态方法（本章第 5.1 节中介绍了静态方法）。当开发者调用 C(*args,**kwds)来创建类 C 的一个新实例时，Python 将首先调用 C.__new__(C,*args,**kwds)。Python 使用__new__的返回值 x 作为新创建的实例。然后，Python 将调用 C.__init__(x,*args,**kwds)，但是只有在 x 确实是 C 的一个实例，或者 C 的任何一个子类时才会调调用该方法（否则，x 的状态仍然是__new__返回的状态）。例如，语句 x=C(23)等同于：

```
x = C.__new__(C, 23)
if isinstance(x, C): type(x).__init__(x, 23)
```

object.__new__可以创建其接收为第一个参数的类的一个新的和未初始化的实例。如果这个类包含一个__init__方法，则该方法将忽略其他参数，但是，如果除了第一个参数，该方法还接收了其他参数，并且第一个参数的类不包含__init__方法，则该方法将引发一个异常。当开发者覆盖类体中的__new__方法时，不需要像平常所作的那样，添加__new__=staticmethod(__new__)：Python 可以识别名称__new__，并在这个上下文环境下对其进行特别处理。只有在开发者以后会在类 C 的类体之外重新绑定 C.__new__的极少情况下，开发者才需要使用 C.__new__=staticmethod(whatever)。

__new__具有工厂函数的大多数灵活性，参见上一节中介绍的工厂函数的习惯用法。__new__可以适当地选择返回一个已有的实例或者创建一个新实例。当__new__确实需要创建一个新实例时，最常见的方法就是通过调用 object.__new__或者 C 的另一个超类的__new__方法来委托这个创建操作。下面的实例显示了如何为了实现 Sigleton 设计模式的一个版本而覆盖静态方法__new__：

```
class Singleton(object):
    _singletons = {}
    def __new__(cls, *args, **kwds):
        if cls not in cls._singletons:
            cls._singletons[cls] = super(Singleton, cls).__new__(cls)
        return cls._singletons[cls]
```

(参见第 5.1 节中介绍合作超类方法调用时说明的 super 内置函数)。Singleton 的任何子

类（不会进一步覆盖__new__）都只有一个实例。如果这个子类定义了一个__init__方法，该子类必须确保__init__方法在被这个子类唯一的类实例进行多次重复调用时（在每次请求创建时）是安全的。

老式类没有__new__方法。

属性引用基础知识

属性引用（attribute reference）是一个形式为 x.name 的表达式，其中 x 是任何表达式，name 是一个调用该属性名称的标识符。非常感谢 Python 对象具有很多属性，但是当 x 表示一个类或实例时，属性引用具有特殊的丰富语法。请记住，方法也是属性，因此，通常本书中提到有关属性的所有内容也可以应用于可调用的属性（也就是方法）。

假定 x 是类 C 的一个实例，类 C 继承自基类 B。这两个类和实例具有以下几个属性（数据和方法）：

```
class B(object):
    a = 23
    b = 45
    def f(self): print "method f in class B"
    def g(self): print "method g in class B"
class C(B):
    b = 67
    c = 89
    d = 123
    def g(self): print "method g in class C"
    def h(self): print "method h in class C"
x = C()
x.d = 77
x.e = 88
```

一些属性名称是很特殊的。例如，C.__name__是字符串‘C’和类的名称。C.__bases__是元组(B,)，C 的基类的元组。x.__class__是类 C，也就是 x 所属的类。当开发者通过这些特殊名称引用一个属性时，该属性引用将直接查找这个类或实例对象中的某个专门位置，并提取在这个位置找到的值。开发者不能解除绑定这些属性。而重新绑定这些属性是被允许的，因此开发者可以在运行时更改一个类的名称或者基类，或者一个实例的类，但是极少需要用到这种高级技术。

类 C 和实例 x 都有另外一个特殊属性：一个名为__dict__的字典。除了极少数特殊属性，一个类或实例的所有其他属性都被保存为该类或实例的__dict__属性中的项目。

从类获得属性

在开发者使用语法 C.name 引用类对象 C 的一个属性时，查询操作将执行以下两个步骤。

1．当‘name’是 C.__dict__中的一个键时，C.name 将从 C.__dict__['name']中提取值 v。

然后，如果 v 是一个描述器（也就是说，type(v)提供了一个名为__get__的方法），则 C.name 的值就是调用 type(v).__get__(v, None, C)的结果。否则，C.name 的值为 v。

2. 否则，C.name 将委托查找 C 的基类，这意味着将循环查找 C 的祖先类，并按照“方法解析顺序”（参见第 5.1.8 节中介绍的方法解析顺序）在每个祖先类中查找 name。

从实例获得属性

当开发者使用 x.name 语句引用类 C 的实例 x 的一个属性时，查询操作将执行以下两个步骤。

1. 当‘name’作为一个覆盖描述器 v 的名称（也就是说，type(v)提供了方法__get__和__set__）在类 C（或者 C 的某个祖先类中）中被找到时，C.name 的值就是调用 type(v).__get__(v, x, C)的结果（这个步骤不能应用于老式实例）。

2. 否则，当‘name’是 x.__dict__中的一个键时，x.name 将提取并返回 x.__dict__['name']的值。

3. 否则，x.name 将委托查找 x 的类（根据前面详细介绍的，用于 C.name 的两个相同查找步骤）。如果找到了一个描述器值 v，则属性查找的全部结果还是 type(v).__get__(v, x, C)；如果找到了一个非描述器值 v，则属性查找的全部结果就是 v。

当这些查找步骤没有找到任何一个属性时，Python 将引发一个 AttributeError 异常。但是，对于 x.name 的查找，如果 C 定义或继承了特殊方法__getattr__，Python 将调用 C.__getattr__(x,'name')，而不是引发该异常（然后根据__getattr__返回一个合适的值或者引发一个适当的异常，通常就是 AttributeError）。

考察以下属性引用：

```
print x.e, x.d, x.c, x.b, x.a          # 打印: 88, 77, 89, 67, 23
```

因为不涉及任何描述器，并且‘e’和‘d’都是 x.__dict__中的键，实例查找过程的第二个步骤将找到 x.e 和 x.d。因此，查找操作将不再继续执行，而是返回 88 和 77。其他三个引用必须继续执行到实例处理的第三个步骤，并在 x.__class__（也就是，类 C）中查找。因为‘c’和‘b’都是 C.__dict__中的键，因此类查找过程的第一个步骤将找到 x.c 和 x.b。因此，查找操作不再继续执行，而是返回 89 和 69。x.a 到达了类处理的第二个步骤，在 C.__bases__[0]（也就是，类 B）中查找。‘a’是 B.__dict__中的一个键；因此，x.a 最终将找到并返回 23。

设置属性

请注意，只有在开发者引用属性，而不是在绑定属性时，才会按照这种方式执行属性查找步骤。在绑定（对类或者实例）一个名称并不特殊的属性时（除非__setattr__方

法，或者覆盖描述器的__set__方法截断了实例属性的绑定），只会影响属性的__dict__条目（分别在类或实例中）。换句话讲，在属性绑定的情况下，除非检查覆盖描述符，不涉及任何查询过程。

绑定和解除绑定方法

函数对象的__get__方法可以返回一个包围该函数的解除绑定方法对象或者一个绑定方法对象。解除绑定和绑定方法之间的关键区别就是，解除绑定方法不与特殊实例相关联，而绑定方法则与之相关联。

在上一节的代码中，属性 f、g 和 h 都是函数；因此，对其中任何一个函数的属性引用将返回一个包围各自函数的方法对象。考察以下代码：

```
print x.h, x.g, x.f, C.h, C.g, C.f
```

这个语句将输出以字符串形式显式的 3 个绑定方法：

```
<bound method C.h of <__main__.C object at 0x8156d5c>>
```

以及以字符串形式显式的 3 个解除绑定方法：

```
<unbound method C.h>
```

在对实例 x 运行属性引用时将得到绑定方法，而在对类 C 运行属性引用时将得到解除绑定方法。

由于绑定的方法已经与特殊实例相关联，开发者可以像下面这样调用该方法：

```
x.h()                          # 打印：method h in class C
```

这里需要注意的一个关键事情就是，开发者不需要按照常规参数传递语法传递方法的第一个参数。而是，实例 x 的绑定方法可以将 self 参数隐式绑定到对象 x 上。这样，这个方法的主体可以访问该实例的属性，即使开发者没有向该方法传递一个显式参数。

但是，解除绑定方法并没有与特殊实例相关联，这样，在开发者调用一个解除绑定的方法时，必须指定一个适当的实例作为第一个参数。例如：

```
C.h(x)                         # 打印：method h in class C
```

开发者调用解除绑定方法的频率远远低于调用绑定方法。解除绑定方法最主要的用途就是访问覆盖方法，参见第 5.1 节；此外，即使对于这个任务，通常最好使用本章第 5.1 节中介绍合作超类方法调用时给出的 super 内置函数。

解除绑定方法详解

正如本书刚才介绍的，当对类的属性引用涉及一个函数时，对这个属性的引用将返回一个包围该函数的解除绑定方法。除了包围的那些函数对象，解除绑定方法还有 3 个属性：im_class 是提供该方法的类对象、im_func 是包围的函数，而 im_self 总是 None。这些属性都是只读的，这意味着试图重新绑定或者解除绑定任何一个属性都将引发一个

异常。

开发者可以就像调用其 im_func 函数那样调用一个解除绑定方法，但是任何调用中的第一个参数必须是一个 im_class 实例或者一个子孙实例。换句话讲，对解除绑定方法的调用必须至少有一个参数，这个参数对应于包围函数的第一个形式参数（按照惯例命名为 self）。

绑定方法详解

在查找过程中，在对一个实例的属性引用找到了一个作为该实例的类的属性的函数对象时，该查找过程将调用该函数的__get__方法以获得这个属性的值。在这种情况下，这个调用将创建并返回一个包围该函数的绑定方法。

请注意，在属性引用的查找过程在 x.__dict__中找到了一个函数对象时，该属性引用操作并不会创建一个绑定方法，因为在这种情况下，这个函数没有别看作是一个描述符，并且该函数的__get__方法没有被调用；而是，该函数对象本身就是这个属性的值。同样地，对于不是普通函数的可调用函数，也不会创建绑定方法，比如内置函数（与 Python 代码不同），因为这些函数不是描述符。

绑定方法与解除绑定方法类似之处在于，除了包围的那些函数对象之外，这两个方法还有 3 个只读属性。与在解除绑定方法中一样，im_class 是提供该方法的类对象，而 im_func 是包围的函数。但是，在绑定方法对象中，属性 im_self 指的就是 x，也就是获得绑定方法的实例。

绑定方法的使用与其 im_func 函数类似，但是对绑定方法的调用不会显式提供一个对应于第一个形式参数（按照惯例命名为 self）的参数。在开发者调用绑定方法时，绑定方法将在调用程序给定其他参数（如果存在）之前，把 im_self 作为第一个参数传递到 im_func 中。

下面将极其详细地从底层介绍与使用普通语法 x.name(arg)的方法调用有关的概念化步骤。在下面的上下文环境中：

```
def f(a, b):...              # 带有两个参数的函数 f

class C(object):
    name = f
x = C()
```

x 是类 C 的一个实例对象，name 是一个为 x 的方法命名的标识符（C 中的一个值为函数的属性，在这里就是函数 f），arg 是任意表达式。Python 将首先检查‘name’是否是覆盖描述符 C 中的属性名称，但是结果为否——函数就是描述符，因为他们的类定义了方法__get__，而不是覆盖的方法，也因为他们的类没有定义方法__set__。Python 接下来检查‘name’是否是 x.__dict__中的一个键，但是结果为否。因此，Python 在 C

中查找 name（如果通过继承，在 C 的一个__bases__中找到了这个 name，则所有的操作都按相同的方式执行）。Python 将会注意到这个属性的值，也就是函数对象 f，是一个描述符。因此，Python 将调用 f.__get__(x, C)，该函数将创建一个绑定方法对象，并将 im_func 设置为 f，将 im_class 设置为 C，将 im_self 设置为 x。然后，Python 将调用这个绑定方法对象，并使用 arg 作为唯一的实际参数。在调用绑定方法的 im_func（也就是函数 f）时，绑定方法将插入 im_self（也就是 x）作为第一个实际参数，而 arg 将成为第二个参数。总的效果就像调用：

```
x.__class__.__dict__['name'](x, arg)
```

在执行一个绑定方法的函数体时，没有特殊的命名空间与其 self 对象或任何类相关。引用的变量是本地的或者全局的，就像在任何其他函数中一样，参见第 4.11 节。变量不会隐式指示 self 中的属性，也不会指示任何类对象中的属性。在该方法需要引用、绑定，或者解除绑定其 self 对象的一个属性时，将通过标准属性-引用语法（例如，self.name）来实现。缺少隐式范围可能会用到某些习惯用法（因为 Python 在这方面与许多其他面向对象语言有所不同），但是其结果是清晰和简单的，并且消除了一些潜在的模糊。

绑定方法对象是第一类对象，并且开发者可以在任何可以使用可调用对象的位置使用该方法。由于绑定方法包含对其包围的函数的引用，还包含可供执行的 self 对象，使其成为闭包的一个强大和灵活的选择（参见第 4.11 节中介绍的内嵌函数和嵌套范围）。其类提供了特殊方法__call__的实例对象（参见第 5.2 节）提供了另一种可行的选择。所有这些构造可以让开发者将某些行为（代码）和某些状态（数据）捆绑成一个单独的可调用对象。闭包是最简单的，但是受限于其适用性。下面是来自于第 4.11 节的内嵌函数和嵌套范围部分介绍的闭包：

```
def make_adder_as_closure(augend):
    def add(addend, _augend=augend): return addend+_augend
    return add
```

绑定方法和可调用实例要比闭包更丰富，也更灵活。下面是如何使用绑定方法实现相同功能的示例：

```
def make_adder_as_bound_method(augend):
    class Adder(object):
        def __init__(self, augend): self.augend = augend
        def add(self, addend): return addend+self.augend
    return Adder(augend).add
```

下面是如何使用可调用实例（其类提供了特殊方法__call__的实例）实现相同功能的示例：

```
def make_adder_as_callable_instance(augend):
    class Adder(object):
        def __init__(self, augend): self.augend = augend
```

```
        def __call__(self, addend): return addend+self.augend
    return Adder(augend)
```

从调用函数的代码的角度来看，所有这些工厂函数都是可以相互转换的，因为他们都返回一个多态（也就是说，可以按相同的方式使用）的可调用对象。从实现的角度来看，闭包是最简单的；绑定方法和可调用实例使用了更加灵活、通用和强大的机制，但是，在这个简单示例中，实在不需要这些额外的强大功能。

继承

在对一个类对象 C 使用属性引用 C.name，并且‘name’不是 C.__dict__中的键时，将按照特定顺序（出于历史原因，这个顺序被称为方法解析顺序，也就是 MRO，尽管这个顺序可以用于所有属性，而不仅仅是方法）对 C.__bases__中的每个类对象隐式执行查找操作。查找操作将按 MRO 逐个检查直接和间接祖先，在‘name’被找到时停止查找。

方法解析顺序

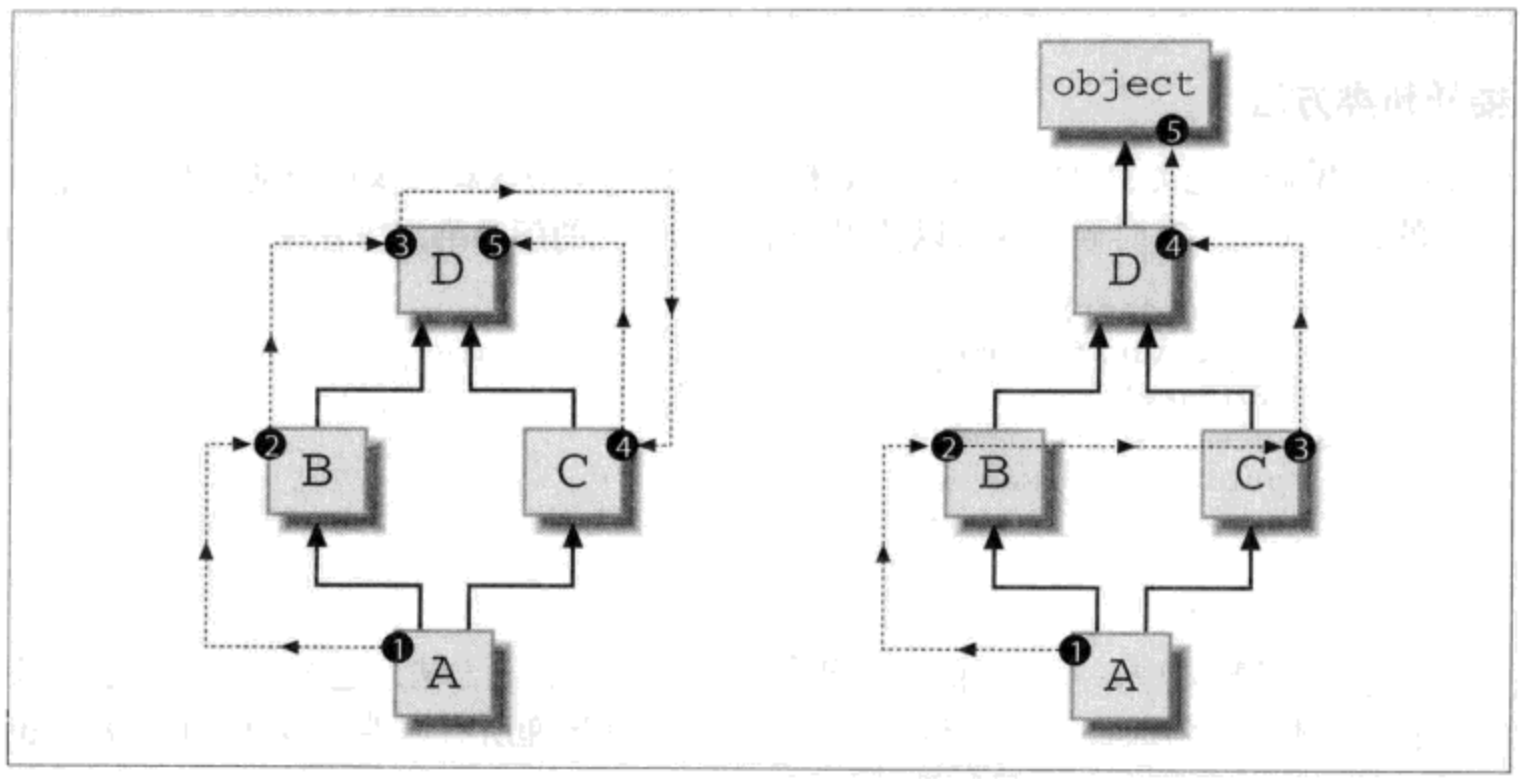
查找一个类中的属性名称本质上是按照从左到右，深度优先的顺序访问其祖先类来实现的。但是，在存在多重继承的情况下（这使得继承图成为一个普通“有向无环图”(Directed Acyclic Graph)，而不是确定的树形图），这个简单的方法可能会导致某些祖先类被访问两次。在这种情况下，通过在查询序列中只保留最右边出现的任何给定类，这样可以理清解析顺序。这个最终的、至关重要的简化并不是传统对象模型规范的一部分，这使得多重继承很难在传统对象模型中正确和有效地使用。在这一点上，新型对象模型要好得多。

在多重继承的情况下，纯粹的从左到右、深度优先的搜索问题可以非常简单地使用一个基于老式类的示例来演示：

```
class Base1:
    def amethod(self): print "Base1"
class Base2(Base1): pass
class Base3:
    def amethod(self): print "Base3"
class Derived(Base2, Base3): pass
aninstance = Derived()
aninstance.amethod()                    # 打印: "Base1"
```

在这个示例中，将从 Derived 类开始查找 amethod 属性。如果没有找到，则继续在 Base2 中查找。由于在 Base2 中也没有找到这个属性，这个传统型的查询将继续在 Base2 的祖先 Base1 中查找，并在这里找到了这个属性。因此，传统型查找将在这个位置停止查找，并且不再考虑在 Base3 中查找，尽管这里也能找到一个具有相同名称的属性。新型的方法解析顺序（Method Resolution Order，MRO）可以通过从类搜索序列中删除最左边出现的 Base1 来解决这个问题，这样，将会找到 Base3 中出现的 amethod 属性。

图 5-1 显示了在这种“菱形”继承图的情况下传统和新型 MRO。



经典方法解析顺序　　新型方法解析顺序

图 5-1　传统和新型 MRO

每个新型类和内置类型都包含一个名为__mro__的特殊只读类属性，这个属性是由按顺序排列的方法解析类型组成的元组。开发者可以只对类，而不对实例引用__mro__，并且，因为__mro__是一个只读属性，开发者不能重新绑定或者解除绑定该属性。要想获得 Python 的 MRO 各个方面的详细信息和高技术说明，可能需要学习 Michele Simionato 的论文“The Python 2.3 Method Resolution Order”，参见 http://www.python.org/2.3/mro.html。

覆盖属性

正如我们在前面看到的，搜索一个属性的操作将按照 MRO（通常沿继承树向上搜索）执行，在找到该属性之后将停止搜索。子孙类总是会在其祖先类之前被搜索到，这样，当一个子类使用超类中的一个相同名称定义了一个属性时，搜索操作将找到子类中的定义，并停止搜索。这被称为子类覆盖了超类中的定义。考察以下示例代码：

```
class B(object):
    a = 23
    b = 45
    def f(self): print "method f in class B"
    def g(self): print "method g in class B"
class C(B):
    b = 67
    c = 89
    d = 123
    def g(self): print "method g in class C"
    def h(self): print "method h in class C"
```

在这段代码中，类 C 覆盖了其超类 B 的属性 b 和 g。请注意，与其他一些语言不一样，在 Python 中，开发者可以覆盖数据的属性，就像覆盖可调用属性（方法）一样简单。

委托超类方法

当子类 C 覆盖其超类 B 的方法 f 时，C.f 的函数体通常应该将其操作的某些部分委托给超类的方法实现。有时候，这可以使用一个如下所示的解除绑定方法来实现：

```
class Base(object):
    def greet(self, name): print "Welcome ", name
class Sub(Base):
    def greet(self, name):
        print "Well Met and",
        Base.greet(self, name)
x = Sub()
x.greet('Alex')
```

在 Sub.greet 函数体中，对超类的委托使用了一个解除绑定方法，这个方法是通过对该超类执行属性引用 Base.greet 获得的，并因此正常地传递所有属性，包括 self。委托超类实现就是对解除绑定方法最频繁的使用。

委托的一个通常用法是与特殊方法__init__一起出现的。当 Python 创建一个实例时，基类的__init__方法都不会像在其他一些面向对象语言中那样被自动调用。这样，如有必要，将由子类使用委托来执行正确的初始化。例如：

```
class Base(object):
    def __init__(self):
        self.anattribute = 23
class Derived(Base):
    def __init__(self):
        Base.__init__(self)
        self.anotherattribute = 45
```

如果类 Derived 的__init__方法没有显式调用类 Base 的方法，Derived 的实例将省略其初始化操作的这个部分，因此这些实例将缺少属性 anattribute。

合作超类方法调用

不过，在基于菱形图的多重继承情况下，使用解除绑定语法调用一个方法的超类版本是非常有问题的。考察以下定义：

```
class A(object):
    def met(self):
        print 'A.met'
class B(A):
    def met(self):
        print 'B.met'
        A.met(self)
```

```
class C(A):
    def met(self):
        print 'C.met'
        A.met(self)
class D(B,C):
    def met(self):
        print 'D.met'
        B.met(self)
        C.met(self)
```

在这段代码中，当我们调用 D().met()时，A.met 在被调用两次后结束运行。怎么确保每个祖先的方法实现被调用了一次，并且仅此一次呢？解决方案就是使用内置类型 super.super(aclass, obj)，该类型将返回对象 obj 的一个特殊超对象。当我们在这个超类中查找一个属性（例如，一个方法）时，该查找操作将在 obj 的 MRO 中的类 aclass 之后开始。因此，可以将上一段代码重新编写为：

```
class A(object):
    def met(self):
        print 'A.met'
class B(A):
    def met(self):
        print 'B.met'
        super(B,self).met()
class C(A):
    def met(self):
        print 'C.met'
        super(C,self).met()
class D(B,C):
    def met(self):
        print 'D.met'
        super(D,self).met()
```

现在，D().met()将导致对每个类的 met 版本只执行一次调用。如果开发者养成了总是使用 super 编写超类调用函数的习惯，则开发者编写的类甚至可以非常平稳地适合复杂的继承结构。当然，只要开发者按照本书推荐的，只使用了新型对象模型，即使出现相反的情况，如果继承结构反而非常简单，也不会产生什么负面影响。

开发者可能会愿意使用通过解除绑定方法技术调用超类方法的粗略方法的唯一情况就是在各种类的相同方法具有不同和不兼容的签名时——这在许多方面都是令人不愉快的情况，但是，如果开发者必须处理这种情况，解除绑定方法技术有时候可能是麻烦最小的。多重继承的正常使用将受到严重地妨碍——但另一方面，即使是 OOP 最基础的属性，比如基类和子类实例之间的多态性，在对应的方法具有不同和不兼容的签名时，也将被严重削弱。

“删除”类属性

通过添加或覆盖子类中的属性，继承和覆盖提供了一种简单并有效的方法来无损地（也

就是说，不会修改定义这些属性的类本身）添加或删除类属性（特别是方法）。但是，继承并没有提供一种方法来无损地删除（隐藏）基类的属性。如果该子类只是在定义（覆盖）一个属性时失败，Python 将找到基类的定义。如果开发者需要执行这样的删除，可能要执行以下操作：

- 覆盖这个方法并在方法体中引发一个异常；
- 避免继承，在子类的_ _dict_ _之外的任何位置保存属性，并为选择委托定义_ _getattr_ _；
- 使用新型对象模型并将_ _getattribute_ _覆盖为类似的效果。

本章第 5.1 节给出了最后一个技术的演示。

内置 object 类型

内置 object 类型是所有内置类型和新型类的祖先。object 类型定义了一些实现了对象的默认语法的特殊方法（参见第 5.2 节）：

_ _new_ _

_ _init_ _

开发者可以通过调用不带任何参数的 object( )来创建对象的一个直接实例。该调用将隐式使用 object._ _new_ _ 和 object._ _init_ _ 以创建并返回一个不包含属性的实例对象（并且甚至不包含用来保存属性的_ _dict_ _）。这样一个实例对象可以被用作“标记”，用来保证与任何其他不同对象的不相等性比较。

_ _delattr_ _

_ _getattribute_ _

_ _setattr_ _

在默认情况下，对象可以使用这些方法处理属性引用（参见第 5.1 节）。

_ _hash_ _

_ _repr_ _

_ _str_ _

任何对象都可以被传递到函数 hash 和 repr 中，还可以被传递到类型 str 中。

object 的子类可以覆盖以上方法中的任何一个和/或添加其他方法。

类级方法

Python 提供了两种内置的非覆盖描述符类型，这为一个类带来了两种不同类型的“类

级方法”（class-level method）。

静态方法

静态方法是可以对类或者类的任何实例进行调用的方法，静态方法没有普通方法（绑定和解除绑定方法）的特殊行为和限制，还与第一个参数有关。静态方法可能会包含任何签名；该方法可能不带任何参数，并且其第一个参数（如果有）不具备任何特殊功能。开发者可以将静态方法看作是一个可以正常调用的普通函数，尽管事实上静态方法有时候会被绑定到一个类属性上。尽管定义静态方法并不是必需的（开发者总是可以定义一个普通函数来替代静态方法），但是，当一个函数的目的紧密绑定到某些特殊类时，有些程序员将静态方法看作是一个很优美的实现方法。

要想构建一个静态方法，可以调用内置类型 staticmethod，并将其结果绑定到一个类属性上。与所有的类属性绑定一样，这通常可以在类体中完成，但是开发者还可以选择在任何位置执行这个绑定操作。staticmethod 的唯一参数就是在 Python 调用静态方法时要调用的函数。下面的示例显示了如何定义和调用一个静态方法：

```
class AClass(object):
    def astatic(): print 'a static method'
    astatic = staticmethod(astatic)
anInstance = AClass()
AClass.astatic()                    # 打印: a static method
anInstance.astatic()                # 打印: a static method
```

这个实例为传递给 staticmethod 的函数和绑定到 staticmethod 的结果的属性使用了相同的名称。这种编程风格并不是强制的，但是是一个好主意，而且，本书推荐开发者总是使用这种风格。Python 2.4 提供了一个特殊的简化语法来支持这种风格，参见第 5.3 节。

类方法

类方法是一个可以对类或该类的任何实例进行调用的方法。Python 将把这个方法的第一个参数绑定到可以调用该方法的类，或者可以调用该方法的实例的类上；Python 不会像普通绑定方法那样，将第一个参数绑定到这个实例上。类方法并没有等同于解除绑定方法的方法。按照惯例，类方法的第一个参数被命名为 cls。尽管定义类方法并不是必需的（开发者总是可以有选择地定义一个普通函数，并类对象作为这个函数的第一个参数），有些程序员将类方法看作是普通函数的一个优美的实现选择。

要想构建一个类方法，可以调用内置类型 classmethod，并将其结果绑定到一个类属性上。与类属性的所有其他绑定一样，这通常是在类体内完成的，但是开发者还可以选择在任何其他位置执行进行绑定。classmethod 的唯一参数就是在 Python 调用该类方法时调用的函数。下面给出了如何定义和调用类方法的一个示例：

```
class ABase(object):
    def aclassmet(cls): print 'a class method for', cls.__name__
```

```
    aclassmet = classmethod(aclassmet)
class ADeriv(ABase): pass
bInstance = ABase()
dInstance = ADeriv()
ABase.aclassmet()                    # 打印: a class method for ABase
bInstance.aclassmet()                # 打印: a class method for ABase
ADeriv.aclassmet()                   # 打印: a class method for ADeriv
dInstance.aclassmet()                # 打印: a class method for ADeriv
```

这个实例为传递给 staticmethod 的函数和绑定到 staticmethod 的结果的属性使用了相同的名称。这种编程风格并不是强制的，但是是一个好主意，而且，本书推荐开发者总是使用这种风格。Python 2.4 提供了一个特殊的简化语法来支持这种风格，参见第 5.3 节。

属性

Python 提供了一个内置覆盖描述器类型，可以用来给定类的实例属性（property）。

属性是一个具有特殊功能的实例属性（attribute）。开发者可以使用普通语法引用、绑定或解除绑定该属性（例如，print x.prop, x.prop=23, del x.prop）。但是，这些访问属性可以对实例 x 调用指定为其内置类型 property 参数的各种方法，而不是遵循用于属性引用、绑定和解除绑定的常规语法。下面是一个如何定义只读属性的示例：

```
class Rectangle(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def getArea(self):
        return self.width * self.height
    area = property(getArea, doc='area of the rectangle')
```

类 Rectangle 的每个实例 r 都有一个合成的只读属性 r.area，是在 r.getArea( )方法中将矩形的两个边相乘计算出来的。文档字符串 Rectangle.area.__doc__是‘area of the rectangle’。因为在对属性的调用中只指定了一个 get 方法，而没有指定 set 或 del 方法，因此属性 r.area 是只读的（试图重新绑定或者解除绑定该属性将会失败）。

属性可以执行那些类似于特殊方法__getattr__、__setattr__和__delattr__（参见第 5.2 节）的任务，但是属性的执行更快也更简单。开发者可以通过调用内置类型 property 创建一个属性，并将其结果绑定到一个类属性上。与所有其他类属性的绑定一样，这通常是在类体中完成的，但是开发者还可以选择在任何位置执行绑定。在类 C 的类体中，可以使用以下语法：

```
attrib = property(fget=None, fset=None, fdel=None, doc=None)
```

当 x 是类 C 的一个实例，并且引用 x.attrib 时，Python 将对 x 调用被传递为属性构造函数的参数 fget 的方法，不带任何参数。在使用 x.attrib = value 赋值时，Python 将调用被传递为参数 fset 的方法，使用 value 作为唯一参数。在执行 del x.attrib 时，Python 将调

用被传递为参数 fdel 的函数，不带任何参数。Python 使用被传递为 doc 的参数作为该属性的文档字符串。property 的所有参数都是可选的。在缺少某个参数时，将禁止对应的操作（当有些代码试图执行这个操作时，Python 将引发一个异常）。例如，在 Rectangle 示例中，属性 area 被定义为只读，因为这个示例只为 fget 传递了一个参数，并没有为 fset 和 fdel 传递参数。

为什么属性很重要？

属性至关重要的地方在于，属性的存在使得开发者可以非常安全并且确实可行地将公共数据属性作为类的公共接口的一部分开放出来。在开发者自己的类或其他需要成为多态类的将来版本中，如果有必要让一些代码在属性被引用、重新绑定或解除绑定时得到执行，开发者知道自己将可以把普通属性（attribute）更改为一个属性，并且在不影响使用这个类（又名“客户代码”）的任何其他代码的情况下获得想要的效果。这可以让开发者避免遇到一些由缺少属性或同等机制的面向对象（OO）语言所要求的愚蠢的习惯用法，比如访问和变异方法。例如，客户代码可以简单地使用自然的习惯用法，比如：

```
someInstance.widgetCounter += 1
```

而不是被迫像下面这样使用嵌套的访问和变异方法：

```
someInstance.setWidgetCounter(someInstance.getWidgetCounter() + 1)
```

在任何时候，如果开发者有兴趣编写一些像 getThis 或者 setThat 这样的接近自然名称的方法，为了更清楚地使用，可以考虑将这些方法包装为属性。

属性和继承

属性通常是继承的，就像任何其他属性（attribute）一样。但是，不注意的话还是会落入一个小陷阱的：为了访问一个属性而调用的方法是在定义该属性本身的类中定义的那些方法，本质上并不会使用可能会出现在子类中的对这些方法的进一步覆盖。例如：

```
class B(object):
  def f(self): return 23
  g = property(f)
class C(B):
  def f(self): return 42
c = C()
print c.g                    # 打印 23，而不是 42
```

访问属性 c.g 时将调用 B.f，而不是像开发者可能直觉上认为的那样，调用 C.f。其原因非常简单：这个属性是通过传递函数对象 f 创建的（并且是在 B 的类语句执行的时候创建的，因此正在谈论的这个函数对象也被称为 B.f）。实际上名称 f 后来在子类 C 中被重新定义了，但这与属性是不相关的，因为该属性不执行对这个名称的查找，而是使用在创建时被传递的函数对象。如果开发者需要解决这个问题，可以总是使用一个

额外的间接层处理来实现：

```
class B(object):
  def f(self): return 23
  def _f_getter(self): return self.f()
  g = property(_f_getter)
class C(B):
  def f(self): return 42
c = C()
print c.g                    # 打印 42，符合预期
```

在这个示例中，属性得到的函数对象是 B._f_getter，这样会按顺序执行对名称 f（因为其调用 self.f()）的查找 ；因此，对 f 的覆盖会产生期望的效果。

_ _slots_ _

通常，任何类 C 的每个实例对象 x 包含一个字典 x._ _dict_ _，Python 使用该字典让开发者将任意属性（attribute）绑定到 x 上。要想节省一些内存（这是以让 x 只有一个预定义的属性名称集合为代价的），开发者可以在一个新型类 C 中定义一个名为_ _slots_ _的类属性，也就是一个由字符串（通常是一些标识符）组成的序列（通常是一个元组）。当新型类 C 包含一个属性_ _slots_ _时，类 C 的直接实例 x 不包含 x._ _dict_ _，并且任何试图在 x 上绑定任何名称不在 C._ _slots_ _中的属性（attribute）时将会引发一个异常。使用_ _slots_ _可以让开发者为小的实例对象减少内存消耗，这些对象不需要非常强大和便利的功能就可以获得任意命名的属性。_ _slots_ _只值得被添加到会引发很多实例的对象中，每个实例可以节省几十个字节是很重要的——特别是对于那些同时运行着几百万个，而不只有几千个实例的类而言。与其他大多数类属性（attribute）不一样，只有在类体中的一些语句将_ _slots_ _绑定为一个类属性时，_ _slots_ _才可以像刚才描述的那样使用。以后再对_ _slots_ _进行的任何变更、重新绑定或者解除绑定都不会产生任何影响，也不能从基类继承_ _slots_ _。下面给出了如何将_ _slots_ _添加到前面定义的 Rectangle 类中以获得更小（尽管灵活性也更少）的实例：

```
class OptimizedRectangle(Rectangle):
    __slots__ = 'width', 'height'
```

开发者不需要为 area 属性定义一个槽（slot）。_ _slots_ _并不限制属性，只有普通的实例属性（attribute）才是在没有定义_ _slots_ _时将会驻留在实例的_ _dict_ _中的属性（attribute）。

_ _getattribute_ _

所有对新型实例中实例属性的引用都是通过特殊方法_ _getattribute_ _来进行的。这个方法是由基类对象提供的，基类对象实现了对象属性引用语法的所有详细内容，参见第 5.1 节中的介绍。但是，开发者可能会覆盖_ _getattribute_ _用作特殊用途，比如为子类的实例隐藏继承的类属性（例如，方法）。下面的示例显示了一种在新型对象模型中实现不能添加项目的列表的方法：

```
class listNoAppend(list):
    def __getattribute__(self, name):
        if name == 'append': raise AttributeError, name
        return list.__getattribute__(self, name)
```

除了性能变得更差之外，类 listNoAppend 的一个实例 x 几乎无法从一个内置的列表对象中被识别出来，并且任何对 x.append 的引用都将引发一个异常。

按实例方法

传统和新型对象模型都允许一个实例的所有属性（attribute）都有实例专用的绑定方法，包括可调用属性（方法）。对于方法而言，就像任何其他属性（除了那些在新型类中绑定到覆盖描述器的属性之外），实例专用绑定隐藏了类级绑定：当属性查找直接在实例中找到了一个绑定，属性查找不考虑类本身。在两种对象模型中，可调用属性的实例专用绑定并不执行本章第 5.1 节中详细介绍的任何转换。换句话讲，属性引用正好返回以前直接绑定到该实例属性的相同可调用对象。

传统和新型对象模型对特殊方法的按实例绑定有着不同的效果，Python 将隐式调用这些特殊方法作为各种操作的结果，参见第 5.2 节。在经典对象模型中，实例可能会非常有用地覆盖一个特殊方法，并且 Python 甚至在隐式调用该方法时使用按实例绑定。在新型对象模型中，特殊方法的隐式使用总是依赖于特殊方法的类级绑定，如果存在的话。下面的代码显示了传统和新型对象模型之间的这个区别：

```
def fakeGetItem(idx): return idx
class Classic: pass
c = Classic()
c.__getitem__ = fakeGetItem
print c[23]                          # 打印: 23
class NewStyle(object): pass
n = NewStyle()
n.__getitem__ = fakeGetItem
print n[23]                          # 结果:
# Traceback (most recent call last):
#   File "<stdin>", line 1, in ?
# TypeError: unindexable object
```

在这个示例中经典对象模型的语法有时候适用于麻烦和有些隐晦的目的。但是，新型对象模型的方法更通用，并且使得类和元类之间的关系规范化和简单化，参见第 5.4 节。

从内置类型继承

新型类可以从内置类型继承。但是，只有在那些内置类型是专门设计用来允许这一级的互兼容性时，类才可以直接或间接作为多个内置类型的子类。Python 不支持从多个任意内置类型无约束地继承。通常，新型类最多只能作为一个实际内置类型的子类——这意味着除了对象，最多只有一个内置类型，该对象是所有内置类型和新型类的超类，

并且不对多重继承采取任何限制。例如：

```
class noway(dict, list): pass
```

将会引发一个 TypeError 异常，以及非常详细的解释“Error when calling the metaclass bases: multiple bases have instance lay-out conflict.”。如果开发者曾经看到过这样的错误信息，这就意味着开发者正在试图直接或间接从多个不是专门设计用来在这样一个深度进行互操作的内置类型进行继承。

5.2 特殊方法

类可以定义或继承特殊方法（也就是，名称以双下划线开始和结束的方法）。每个特殊方法都与一个特殊操作相关。任何时候对一个实例对象执行相关操作时，Python 将会隐式调用一个特殊方法。在大多数情况下，这个特殊方法的返回值就是操作的结果，并在其相关方法没有显示引发了一个异常的情况下尝试这个操作。本节将指出一些不能应用这种一般规则的情况。在本节中，要对类 C 执行特殊操作，x 是类 C 的实例，y 则是另一个操作数（如果有）。每个方法的形式参数 self 也引用实例对象 x。在下面的几节中，只要提到调用 x.__name__(...)，请记住，对于新型类，比较学术的说法就是，真正调用的是 x.__class__.__name__(x,...)。

通用目的特殊方法

某些特殊方法与通用目的操作有关。定义或继承了这些方法的类允许其实例控制这样的操作。这些操作可以分成以下几类：

初始化和终止化

类可以通过特殊方法__new__（只适用于新型类）和__init__控制其实例的初始化（initialization，经常需要）和/或通过特殊方法__del__控制其实例的终止化（finalization，很少需要）。

表现为字符串

类可以控制 Python 如何通过特殊方法__repr__、__str__和__unicode__将其实例表现为字符串。

布尔型环境中的比较、哈希和使用

类可以控制其实例如何与其他对象进行比较（使用方法__lt__、__le__、__gt__、__ge__、__eq__、__ne__和__cmp__），可以控制字典如何将其实例用作键和设置为成员（__hash__），以及在布尔型环境下这些实例的计算结果是 True 还是 False（__nonzero__）。

属性引用、绑定和解除绑定

类可以通过特殊函数（__getattribute__（只适用于新型类）、__getattr__、__setattr__和__delattr__）控制对其实例属性的访问（引用、绑定和解除绑定）。

可调用实例

如果实例的类包含特殊方法__call__，则该实例是可调用的，就像函数对象一样。

表 5-1 给出了通用目的特殊方法的详细说明。

表 5-1

__call__	__call__(self[,*args...*]) 在调用 x([args...])时，Python 将把这个操作翻译为调用 x.__call__([args...])。这个调用操作的参数与__call__方法的参数相同，只是减少了第一个参数。这个通常被称为 self 的第一个参数指的就是 x，Python 将自动地隐式提供该参数，就像对绑定方法的任何其他调用一样。
__cmp__	__cmp__(self,*other*) 任何比较运算符，在缺少其特定的特殊方法（__lt__、__gt__ 等）或者返回 NotImplemented 时，都将改为调用 x.__cmp__(y)，就像需要用到比较的内置函数一样，比如 cmp(x, y)和 max(x, y)，以及列表对象的 sort 方法。如果 x 小于 y，则__cmp__将返回-1，如果 x 等于 y，则返回 0，如果 x 大于 y，则返回 1。在__cmp__也缺少时，次序比较运算（<、<=、>和>=）将引发异常。在这种情况下，相等比较（= =、!=）将变成标识检查：x= =y 相当于计算 id(x)= =id(y)的值（也就是说，x is y）。
__del__	__del__(self) 在 x 因为垃圾收集而消失之前，Python 将调用 x.__del__( )让 x 终止其自身。如果没有__del__，Python 将不对垃圾收集 x 执行特殊的终止化操作（这是一种常见情况，因为极少的类需要定义__del__）。Python 将忽略__del__的返回值。Python 不执行对类 C 的超类的__del__方法的隐式调用。C.__del__必须隐式执行任何需要的终止化。 例如，在类 C 有一个基类 B 需要终止化时，C.__del__中的代码必须调用 B.__del__(self)（或者更好的情况是，对于新型类，可以调用 super(C, self).__del__()）。在开发者需要及时和有保证的终止化时，__del__通常不是最好的解决方案。对于这种需要，可以是用包含在“try/finally”中的 try/finally 语句（或者更好的情况是，使用 Python 2.5 中的新 with 语句，参见第 6.1 节中介绍的 with 语句）。 定义__del__的类实例不能参与循环垃圾收集，参见第 13.3 节“垃圾收集”。因此，开发者必须特别小心，避免在这些实例中出现引用循环，应该只在没有合理的选择时才定义__del__。
__delattr__	__delattr__(self, *name*) 在每次请求解除绑定属性 x.y（例如，del 语句 del x.y）时，Python 将调用 x.__delattr__('y')。后面将要介绍的有关__setattr__的所有事项也都可以应用于__delattr__。Python 将忽略__delattr__的返回值。如果缺少__delattr__，Python 通常将把 del x.y 翻译成 del x.__dict__['y']。

续表

__eq__, __ge__, __gt__, __le__, __lt__, __ne__	__eq__(self, *other*) __ge__(self, *other*) __gt__(self, *other*) __le__(self, *other*) __lt__(self, *other*) __ne__(self, *other*) 比较操作 x==y、x>=y、x>y、x<=y、x<y 和 x!=y 将分别调用上面列出的特殊方法，这些方法将返回 False 或 True。每个方法可能会返回 NotImplemented，告诉 Python 以另外的方法处理比较操作（例如，Python 可以尝试用 x<y 代替 y>x）。
__getattr__	__getattr__(self, *name*) 当属性 x.y 被访问，但是没有在通常的步骤中被发现时（也就是说，通常会引发 AttributeError 异常），Python 将改为调用 x.__getattr__('y')。Python 并不会为使用普通方法找到的属性调用__getattr__（也就是说，x.__dict__ 中的键或者通过 x.__class__）。如果开发者想要 Python 对每个属性引用调用__getattr__，可以将属性放在其他位置（例如，在一个具有私有名称的属性引用的另一个字典中），或者改为编写一个新型类和覆盖__getattribute__。如果__getattr__找不到 y，则引发 AttributeError 异常。
__getattribute__	__getattribute__(self, *name*) 在每次请求访问属性 x.y 时，如果 x 是新型类 C 的一个实例，Python 将调用 x.__getattribute__('y')，该方法必须获得并返回这个属性值，否则将引发 AttributeError 异常。属性可以访问的普通语法（使用 x.__dict__、C.__slots__、C 的类属性 x.__getattr__）都起因于 object.__getattribute__。 如果类 C 覆盖了__getattribute__，则必须实现该方法应该提供的所有属性访问语法。通常，实现属性访问语法最便利的方法是通过委托来实现的（例如，调用 object.__getattribute__(self,...)，作为覆盖__getattribute__操作的一部分）。请注意，当一个类覆盖__getattribute__时，对类实例的属性访问变得很慢，因为覆盖代码是对每个这样的属性访问调用的。
__hash__	__hash__(self) hash(x)内置函数调用，以及使用 x 作为一个字典键（比如 D[x]，这里 D 是一个字典）或者一个集合成员，调用 x.__hash__(). __hash__必须返回一个 32-位的 int 型值，这样，x==y 意味着 hash(x)==hash(y)，对于一个给定对象，还必须返回相同的值。 在缺少__hash__，同时还缺少__cmp__和__eq__时，hash(x)将使用 x 作为一个字典键或者一个集合成员调用 id(x)。 对于任意 x，如果 hash(x)都会返回一个结果，而不是引发一个异常，则可以将这个 x 称为可哈希对象（hashable object）。在缺少__hash__，但是包含__cmp__或__eq__时，hash(x)使用 x 作为一个字典键时将会引发一个异常。在这种情况下，x 不是可哈希的，并且不能是一个字典键。 通常定义__hash__只是为了用于不可变对象，而这些对象还定义了__cmp__和/或__eq__。请注意，如果存在任意 y，并且 x==y，即使 y 是一个不同的类型，如果 x 和 y 都是可哈希的，则开发者必须确保 hash(x)==hash(y)。

__init__	__init__(self[,*args...*]) 当一个函数调用 C([args...])创建类 C 的一个实例 x 时，Python 将调用 x.__init__([args...])让 x 初始化其本身。如果缺少__init__，开发者必须不带参数调用类 C，并且 C()和 x 在创建时是没有实例专用属性的。严格地讲，新型类 C 是不会缺少__init__的，因为新型类将从 object 继承__init__，除非其重新定义了__init__；但是，即使在这种情况下，开发者仍然必须不带参数调用类 C，C()，并且结果实例在创建时是没有实例专用属性的。 __init__必须返回 None。Python 不执行对类 C 的超类的__init__方法隐式调用。C.__init__必须显式执行任何需要的初始化。例如，在需要不带参数地初始化类 C 的一个基类 B 时，C.__init__中的代码必须显式调用 B.__init__(self)（或者更好的情况是，对于新型类，可以调用 super(C, self).__init__()。
__new__	__new__(cls[,*args...*]) 在调用 C([args...])，并且 C 是一个新型类时，Python 将获得一个通过调用 C.__new__(C,[args...])创建的新实例 x。__new__是每个新型类都包含的一个静态方法（通常简单地从 object 继承而来），并且该方法可以返回任意值 x。换句话讲，__new__没有被限制为返回 C 的一个新实例，尽管通常人们期望这个方法这样做。当且仅当__new__返回的值 x 确实是 C 的一个实例时（不管是一个新建的还是以前存在的一个实例），在通过对 x 隐式调用__init__（使用与最初传递给__new__的参数相同的[args...]）来调用__new__之后，Python 将继续运行。 由于开发者可以对任何一个特殊方法（__init__或__new__）中的新实例执行大多数类型的初始化，开发者可能会产生疑问，最好在哪里进行初始化呢？答案非常简单：将所有类型的初始化放到__init__中，除非有一些特殊的、高级的原因要将一些类型的初始化放到__new__中。在各种各样的环境下，这样做会让处理更简单一些，这是因为事实上__init__是一个实例方法，而__new__是一个非常特殊的静态方法。
__nonzero__	__nonzero__(self) 在 x 的计算结果为 True 或 False（参见第 4.2 节）——例如，在调用 bool(x)时——Python 将调用 x.__nonzero__()，这个调用将返回 True 或 False。在没有__nonzero__时，Python 将改为调用__len__，并在 x.__len__()返回 0 时将 x 认为是 False（因此，要想检查一个容器是否非空，不要使用代码 if len(container)>0:；而是使用 if container:）。在既没有__nonzero__，也没有__len__时，Python 总是将 x 认为是 True。
__repr__	__repr__(self) repr(x)内置函数调用、‘x’ 表达式形式，以及交互式解释器（在 x 是一个表达式语句的结果时）可以调用 x.__repr__()以获得 x 的一个“官方的”和完整的字符串来表现。如果缺少__repr__，Python 将使用一个默认字符串来表现。__repr__必须返回一个字符串，包含对 x 的无歧义信息。在理想情况下，在有可行性时，该字符串必须是一个像 eval(repr(x))==x 这样的表达式。

续表

__setattr__	__setattr__(self, *name*, *value*) 在每次请求绑定属性 x.y 时（通常使用赋值语句 x.y=value），Python 将调用 x.__setattr__('y', value)。Python 总是可以对绑定到 x 的任何属性调用 __setattr__，这是与__getattr__的一个主要区别（从这个角度来看，__setattr__更接近于新型类的__getattribute__）。要想避免递归，在 x.__setattr__绑定 x 的属性时，必须直接修改 x.__dict__（例如，使用 x.__dict__[name]=value 修改）；更好的情况是，对于新型类，__setattr__可以委托对超类的设置（通过调用 super(C, x).__setattr__('y', value)）。Python 将忽略__setattr__的返回值。如果缺少__setattr__，Python 通常将把 x.y=z 翻译为 x.__dict__['y']=z。
__str__	__str__(self) str(x)内置类型和 print x 语句可以调用 x.__str__()以获得 x 的一个非正式的、简洁的字符串表现形式。如果缺少__str__，Python 将改为调用 x.__repr__。__str__应该返回一个方便的、具有可读性的字符串，即使对该字符串进行了一些近似化。
__unicode__	__unicode__(self) unicode(x)内置类型将优先调用 x.__unicode__()（如果有），而不是 x.__str__()。如果一个类同时提供了这两种特殊方法（__unicode__和 __str__），这两种方法必须返回相等的字符串（分别是 Unicode 和纯文本字符串类型）。

容器的特殊方法

实例可以是一个容器（container）（这个容器可以是序列或映射，但是不能同时都是，因为这两个概念是互斥的）。为了得到最大程度的使用，容器不仅需要提供特殊方法__getitem__、__setitem__、__delitem__、__len__、__contains__和__iter__，还需要提供几个非特殊方法，下面几节将介绍这些方法。

序列

在每个项目访问特殊方法中，一个包含 L 个项目的序列必须接受任何一个整数 key，-L<=key<L。为了与内置序列兼容，负的索引 key（0>key>=-L）应该等于 key+L。在 key 的类型无效时，特殊方法将引发 TypeError。在 key 是一个有效类型的值，但是超出了范围时，特殊方法将引发 IndexError。对于没有定义__iter__的容器类，for 语句可以实现迭代需求，就像使用可迭代参数的内置函数一样。序列的每个项目访问特殊方法还应该可以将内置类型 slice 的一个实例接受为该方法的索引参数，且 slice 的 start、step 和 stop 属性是 int 或 None。切片（slicing）语法依赖于这一要求，参见第 5.2 节中介绍的容器切片。

序列还必须允许使用使用 + 运算符串联（与相同类型的另一个序列）和使用 * 运算符重复（乘以一个整数）。因此序列必须包含特殊方法__add__、__mul__、__radd__

和__rmul__，参见第 5.2 节。一个序列与另一个具有相同类型的序列必须是有可比性的，可以像列表和元组那样实现字典（lexicographic）比较。可变序列也必须包含__iadd__和__imul__，以及第 4.6 节中介绍的非特殊方法：append、count、index、insert、extend、pop、remove、reverse 和 sort，这些方法与对应的列表方法具有相同的签名和语法。如果序列中的所有项目都是可哈希的，则不可变序列也必须是可哈希的。序列的类型可能会在有些方面限制其项目（例如，只接受字符串项目），但是这并不是强制的。

映射

在映射的项目访问特殊方法接收到一个有效类型的无效 key 参数值时，将会引发 KeyError，而不是 IndexError。任何映射都必须定义第 4.8 节中介绍的非特殊方法：copy、get、has_key、items、keys、values、iteritems、iterkeys 和 itervalues。特殊方法__iter__必须等于 iterkeys。一个映射与另一个具有相同类型的映射必须是有可比性的。可变映射还必须定义 clear、popitem、setdefault 和 update 方法。如果映射中的所有项目都是可哈希的，则不可变映射也必须是可哈希的。映射的类型可能会在有些方面限制其键（例如，只接受可哈希的键，或者更特殊的是，只接受字符串键），但是这并不是强制的。

集合

集合可以被看作是一些非常特殊的容器，既不是序列也不是映射的容器，并且不能被索引，但是集合是有长度（元素的数量）和可迭代的。集合还支持许多运算符（&、|、^、-，以及成员测试和比较）和同等的非特殊方法（intersection、union 等）。如果开发者要实现一个类似于集合的容器，该容器对 Python 内置集合必须是多态的，参见第 4.2 节。如果类型中的所有元素都是可哈希的，则不可变的类似于集合的类型也必须是可哈希的。类似于集合的类型可能会在有些方面限制其元素（例如，只接受可哈希的元素，或者甚至更特殊，只接受整数元素），但是这并不是强制的。

容器切片

在对一个容器 x 引用、绑定或解除绑定一个像 x[i:j]或 x[i:j:k]这样的切片时，Python 将调用 x 的可用项目访问特殊方法，将一个内置类型的对象传递为 key，这个对象名为切片对象（slice object）。切片对象具有 start、stop 和 step 属性。如果这些属性对应的值在切片语法中被省略，则该属性为 None。例如，del x[:3]将调用 x.__delitem__(y)，y 是一个切片对象，在这种情况下，y.stop 为 3，y.start 为 None，y.step 为 None。容器对象 x 负责正确解释传递给 x 的特殊方法的切片对象参数。切片对象的 indices 方法是很有用的：调用该方法，并使用容器的长度作为其唯一参数，该方法将返回一个由 3 个非负索引组成的元组，分别相当于开始、结束和步长，这个元组用来对切片中的每个项目进行循环索引。为了完全支持切片，序列类的__getitem__特殊方法的一个通常的习惯用法是：

```
def __getitem__(self, index):
  # 循环特殊情况下的切片
  if isinstance(index, slice):
    return self.__class__(self[x]
                          for x in xrange(*index.indices(len(self)))
  # 检查索引，同时负的索引进行处理
  if not isinstance(index, int): raise TypeError
  if index<0: index+=len(self)
  if not (0<=index<len(self)): raise IndexError
  # 现在索引是一个包含在 range(len(self))范围之内的正确整数
 ... __getitem__的其余部分，通过整数索引处理单个项目访问...
```

这个习惯用法使用了 Python 2.4 生成器表达式（genexp）语法并假定这个类的__init__方法可以使用一个可迭代参数以创建该类的一个合适的新实例。

有些内置类型，比如 list 和 tuple，定义了（出于向后兼容的原因）一些现在不被推荐的特殊方法__getslice__、__setslice__和__delslice__。对于这种类型的实例 x，只使用一个冒号的切片 x，如 x[i:j]，将调用切片专用特殊方法。使用两个冒号的切片 x，如 x[i:j:k]，将以一个切片对象为参数调用项目访问特殊方法。例如：

```
class C:
    def __getslice__(self, i, j): print 'getslice', i, j
    def __getitem__(self, index): print 'getitem', index
x = C()
x[12:34]
x[56:78:9]
```

第一个切片调用 x.__getslice__(12,34)，第二个切片调用 x.__getitem__(slice(56,78,9))。最好通过不在类中定义切片专用特殊方法来避免这种复杂的情况；但是，如果开发者定义的类是 list 或 tuple 的子类，并且开发者想要在这个类的实例使用一个冒号进行切片的时候提供特殊功能，则可能需要覆盖这些方法。

容器方法

特殊方法__getitem__、__setitem__、__delitem__、__iter__、__len__和__contains__表现了容器的功能。

表 5-2

__contains__	__contains__(self,*item*)
	布尔型测试运算 y in x 将调用 x.__contains__(y)。当 x 是一个序列，并且 y 等于该序列中的一个项目的值时，__contains__将返回 True。当 x 是一个映射，并且 y 等于该映射中的一个键的值时，__contains__将返回 True。否则，__contains__将返回 False。如果缺少__contains__方法，Python 将执行下面的 y in x 运算，执行时间与 len(x)成正比：
	`for z in x:`
	`  if y==z: return True`
	`return False`

续表

__delitem__	__delitem__(self,*key*) 对于解除绑定 x 的一个项目或切片的请求（通常使用 del x[key]），Python 将调用 x.__delitem__(key)。只有在容器 x 是可变的情况下，容器 x 才应该包含__delitem__方法，这样才可以删除项目（也有可能是切片）。
__getitem__	__getitem__(self,*key*) 在可以访问 x[key]时（也就是说，在容器 x 是可以索引或者切片时），Python 将调用 x.__getitem__(key)。所有（不类似于集合）容器必须包含__getitem__方法。
__iter__	__iter__(self) 对于对 x 的所有项目进行循环的请求（通常使用 for item in x），Python 将调用 x.__iter__()以获得 x 的迭代器。内置函数 iter(x)也可以调用 x.__iter__()。在缺少__iter__，并且 x 是一个序列时，iter(x)将合成并返回一个迭代器对象，该对象将包装 x 并返回 x[0]、x[1]等，直到某一个索引引发 IndexError 以表示序列的结尾。但是，开发者最好确保编写的所有容器类都包含__iter__方法。
__len__	__len__(self) len(x)内置函数调用，以及需要知道容器 x 中有多少个项目的其他内置函数都可以调用 x.__len__()。__len__将返回一个整型值，也就是 x 中的项目数。在缺少__nonzero__方法时，Python 还可以调用 x.__len__()在布尔型上下文环境下计算 x。如果缺少__nonzero__方法，当且仅当一个容器为空（也就是说，该容器的长度为 0）时，这个容器被认为是 False。所有容器都必须包含__len__方法，除非该容器确定其当前包含多少个项目的开销极其昂贵。
__setitem__	__setitem__(self,*key*,*value*) 对于绑定 x 的一个项目或切片的请求（通常使用赋值语句 x[key]=value），Python 将调用 x.__setitem__(key, value)方法。只有在容器 x 是可变的情况下，容器 x 才应该包含__setitem__，这样才可以添加和/或解除绑定项目，也有可能是切片。

数值对象的特殊方法

实例可以使用许多特殊方法来提供数值运算。有些不是数字的类还支持下面这些特殊方法以重载像 + 和 * 这样的运算符。例如，正如本章的第 5.2 节中介绍的序列，序列必须具有特殊方法__add__、__mul__、__radd__和__rmul__。

表 5-3

__abs__, __invert__, __neg__, __pos__	__abs__(self) __invert__(self) __neg__ self) __pos__ self) 一元运算符 abs(x)、~x、-x 和+x 将分别调用以上这些方法。

续表

__add__, __div__, __floordiv__, __mod__, __mul__, __sub__, __truediv__	__add__(self,*other*) __div__(self,*other*) __floordiv__(self,*other*) __mod__(self,*other*) __mul__(self,*other*) __sub__(self,*other*) __truediv__(self,*other*) 运算符 x+y、x/y、x//y、x%y、x*y、x-y 和 x/y 将分别调用以上这些方法。在除法是非截断（参见第 4.5 节）的情况下，运算符 / 将调用__truediv__（如果有），而不是调用__div__。
__and__, __lshift__, __or__, __rshift__, __xor__	__and__(self,*other*) __lshift__(self,*other*) __or__(self,*other*) __rshift__(self,*other*) __xor__(self,*other*) 运算符 x&y、x<<y、x|y、x>>y 和 x^y 将分别调用以上这些方法。
__coerce__	__coerce__(self,*other*) 对于任何有两个操作数 x 和 y 的数值运算，Python 将调用 x.__coerce__(y).__coerce__，并返回一个由 x 和 y 转换成可接受类型组成的数值对。在不能执行转换操作时，__coerce__将返回 None。在这种情况下，Python 将调用 y.__coerce__(x)。现在并不推荐使用这个特殊方法；因此不要求开发者的类中实现这个方法，而是直接在相关数值运算的特殊方法中处理这些类可以接受的任何类型。但是，如果一个类提供了__coerce__方法，为了向后兼容，Python 仍将会调用这个方法。
__complex__, __float__, __int__, __long__	__complex__(self) __float__(self) __int__(self) __long__(self) 内置类型 complex(x)、float(x)、int(x)和 long(x)将分别调用以上这些方法。
__divmod__	__divmod__(self,*other*) 内置函数 divmod(x, y)将调用 x.__divmod__(y).__divmod__，并返回一个等于(x//y, x%y)的(quotient, remainder) （商，余数）对。
__hex__, __oct__	__hex__(self) __oct__(self) 内置函数 hex(x)将调用 x.__hex__()。内置函数 oct(x)将调用 x.__oct__()。这两个特殊函数将分别返回一个表示 x 的十六进制或八进制值的字符串。
__iadd__, __idiv__, __ifloordiv__, __imod__, __imul__, __isub__, __itruediv__	__iadd__(self,*other*) __idiv__(self,*other*) __ifloordiv__(self,*other*) __imod__(self,*other*) __imul__(self,*other*) __isub__(self,*other*) __itruediv__(self,other) 增量赋值运算 x+=y、x/=y、x//=y、x%=y、x*=y、x-=y 和 x/=y 将分别调用以上这些方法。每个方法都将在原地修改 x，并返回 self。当 x 是可变时定义这些方法（也就是说，当 x 可以在原地被更改时）。

续表

__iand__, __ilshift__, __ior__, __irshift__, __ixor__	__iand__(self,*other*) __ilshift__(self,*other*) __ior__(self,*other*) __irshift__(self,*other*) __ixor__(self,*other*) 增量赋值运算 x&=y, x<<=y, x|=y, x>>=y 和 x^=y 将分别调用以上这些方法。每个方法都将在原地修改 x，并返回 self。
__index__	__index__(self) 只能用于 Python 2.5。与__int__相似，但是__index__只能由整数的其他一些实现类型来提供（换句话讲，这个类型的所有实例都可以被确定地映射到整数上）。例如，除了所有内置类型之外，只有 int 和 long 提供__index__；而 float 和 str 则不提供，尽管他们提供了__int__方法。在 Python 2.5 中，序列索引和切片内部使用__index__来获得需要的整数切片（而直到 Python 2.4，还是只要求使用 int 或 long 类型）。使用新的特殊方法__index__，Python 2.5 为用户代码或第三方扩展，比如 gmpy（参见第 15.5 节），提供的其他一些可选择的整数实现提供了更好的支持。
__ipow__	__ipow__(self,*other*) 增量赋值运算 x**=y 将调用 x.__ipow__(y). __ipow__,这样将在原地修改 x，并返回 self。
__pow__	__pow__(self,*other*[,*modulo*]) x**y 和 pow(x,y)都将调用 x.__pow__(y)，而 pow(x, y, z)将调用 x.__pow__(y,z). x.__pow__(y,z)，这样将返回一个等于表达式 x.__pow__(y)%z 的值。
__radd__, __rdiv__, __rmod__, __rmul__, __rsub__	__radd__(self,*other*) __rdiv__(self,*other*) __rmod__(self,*other*) __rmul__(self,*other*) __rsub__(self,*other*) 运算符 y+x、y/x、y%x、y*x 和 y-x 将在 y 不具备所需的方法__add__、__div__等时，分别调用以上这些方法。
__rand__, __rlshift__, __ror__, __rrshift__, __rxor__	__rand__(self,*other*) __rlshift__(self,*other*) __ror__(self,*other*) __rrshift__(self,*other*) __rxor__(self,*other*) 运算符 y&x, y<<x, y|x, y>>x 和 y^x 将在 y 不具备所需的方法__and__、__lshift__等时，分别调用以上这些方法。
__rdivmod__	__rdivmod__(self, *other*) 内置函数 divmod(y, x)在 y 不具备__divmod__. __rdivmod__将调用 x.__rdivmod__(y)，并返回一个(remainder, quotient)数据对。
__rpow__	__rpow__(self, *other*) y**x 和 pow(y, x)将在 y 不具备__pow__方法时调用 x. __rpow__(y)。在这种情况下没有 3 个参数的形式。

5.3 装饰器

由于像 staticmethod 和 classmethod 这样的解释器类型的存在（参见第 5.1 节），这些类型将把一个函数对象作为其参数，Python 将在类体内比较频繁地使用以下的习惯用法：

```
def f(cls,...):
 ...此处省略 f 的定义...
f = classmethod(f)
```

把对 classmethod 进行调用的文本放在 def 语句的后面可能会减少代码的可读性，因为在读取 f 的定义时，代码的阅读者还没有意识到 f 将要成为一个类方法，而不是一个普通实例方法。如果将对 classmethod 的调用放在 def 语句之前，而不是之后，则代码会更具可读性。Python 2.4 允许通过下面这个被称为装饰的新语法形式实现这种放置顺序：

```
@classmethod
def f(cls,...):
 ...此处省略 f 的定义...
```

@classmethod 装饰必须紧跟在 def 语句的后面，这意味着 f=classmethod(f)将紧跟在 def 语句之后执行（不管 def 为 f 定义了什么名称）。更通用的是，@expression 将计算该表达式（这个表达式必须是一个名称，也有可能是一个调用）并将计算结果绑定到一个内部临时名称（假定，__aux）上；任何这样的装饰必须紧跟在 def 语句之后，表示 f=__aux(f)是在 def 语句之后立即执行的（不管 def 为 f 定义了什么名称）。绑定到__aux 的对象被称为装饰器（decorator），也就是说用来装饰函数 f。

装饰为有些高阶函数（以及其他一些使用类似于高阶函数的可调用函数）提供了一种很方便的简单表达方式。开发者可以对任意 def 语句应用装饰，而不仅仅是类体中出现的 def 语句。开发者还可以编写自己的装饰器，也就是高阶函数，装饰器将接受一个函数对象作为其参数，并将一个函数对象作为结果返回。例如，下面是一个不修改其装饰的函数，而是在函数定义时将函数的文档字符串发送到标准输出的装饰器：

```
def showdoc(f):
    if f.__doc__:
        print '%s: %s' % (f.__name__, f.__doc__)
    else:
        print '%s: No docstring!' % f.__name__
    return f

@showdoc
def f1(): "a docstring"
# 打印: f1: a docstring

@showdoc
def f2(): pass
# 打印: f2: No docstring!
```

5.4 元类

任何对象，即使是一个类对象，都有一个类型。在 Python 中，类型和类也都是第一类对象。类对象的类型也被称为该类的元类（metaclass）（严格来说，类 C 的类型只能被称为 C 的实例的元类，而不是 C 本身的元类，但是这个非常细微的术语上的区别在实际使用中是极少被察觉的）。对象的行为主要是由该对象的类型确定的。这也适用于类：类的行为也是主要由该类的元类确定的。元类是一个高级主题，在第一次阅读本书时，你可能会跳过本节的其余部分。但是，完全地掌握有关元类的知识可以帮助开发者获得对 Python 更深入的理解，有时候定义开发者自己的元类也是很有用的。

传统和新型类的区别依赖于这样一个事实：每个类的行为是由其元类确定的。换句话讲，传统类的行为有别于新型类的原因在于传统类和新型类属于不同类型（元类）的对象：

```
class Classic: pass
class Newstyle(object): pass
print type(Classic)                    # 打印: <type 'class'>
print type(Newstyle)                   # 打印: <type 'type'>
```

Classic 的类型是来自标准模型 types 的对象 types.ClassType，而 Newstyle 的类型是内置对象 type。type 也是所有 Python 内置类型的元类，包括其本身（也就是说，print type(type) 也会打印<type 'type'>）。

Python 如何确定一个类的元类

要想执行一个 class 语句，Python 首先将其基类收集到元组 t（如果没有基类，则该元组为空）中，并在一个临时字典 d 中执行该类体。然后，Python 将确定要为 class 语句正在创建的新类对象 C 使用的元类 M。

当‘__metaclass__’是 d 中的一个键时，M 则是 d['__metaclass__']。这样，开发者可以通过在 C 的类体中绑定属性__metaclass__来显式控制类 C 的元类。否则，在 t 为非空时（也就是说，当 C 有一个或多个基类时），M 是 C 的基类的所有元类中的叶子元类（如果 C 的基类的元类没有形成一个包括其底层范围的继承网格，也就是说，没有叶子元类——Python 将引发一个异常以诊断这个元类型冲突）。这就是为什么从 object 继承就表示 C 是一个新型类的原因。由于 type(object)就是 type，从 object（或其他内置类型）继承的类 C 将得到与 object 相同的元类（也就是说，C 的元类 type(C)也是 type）。因此，一个新型类与具有元类 type 是同义的。

在 C 没有基类，但是当前模块具有一个全局变量__metaclass__时，M 是这个全局变量的值。这使得开发者可以让没有基类的类在一个模块中被默认为新型类，而不是传统类。只需要将以下语句放在模块体的开始即可：

```
__metaclass__ = type
```

如果缺少以上这个语句，M 将默认为 types.ClassType。在__metaclass__没有在类体中被绑定或者作为模块的一个全局变量时，这个最后的“默认值中的默认值”子句就是为什么没有基类的类在默认情况下是传统类的原因。

元类如何创建类

在确定了 M 之后，Python 就可以通过 3 个参数来调用 M 了：类名称（一个字符串）、元类 t 的元组和字典 d。对 M 的调用将返回类对象 C，然后 Python 将把该类对象绑定到输入的类名称上，完成其 class 语句的执行。请注意，这实际上是类型 M 的一个实例，因此对 M 的调用将执行 M.__init__(C, namestring, t, d)，其中 C 是 M.__new__(M, namestring, t, d)的返回值，就像在一个新型类的任何其他类似实例中一样。

在创建了类对象 C 之后，类 C 及其类型（type(C)，通常是 M）之间的关系与任何对象及其类型之间的关系是相同的。例如，在调用类对象 C（创建 C 的一个实例）时，将会执行 M.__call__，并以类对象 C 作为第一个实际参数。

请注意本章第 5.1 节中介绍的新型方法的好处。调用 C 来进行实例化必须执行其元类的 M.__call__，不管 C 是否具有一个按实例属性（方法）__call__（也就是说，独立于 C 的实例是否可调用）。这个要求与传统对象模型是完全不兼容的，在传统对象模型中，按实例方法将覆盖按类方法——即使是被隐式调用的特殊方法。新型方法避免了必须在类及其元类之间建立关系，一种特别特殊的关系。避免特别特殊的情况是 Python 能够提供强大功能的关键：Python 只有极少的、简单的一般规则，并且可以始终如一地应用这些规则。

定义和使用自定义元类

自定义元类是很简单的：从 type 继承并覆盖其中的一些方法即可。开发者还可以使用__new__、__init__和__getattribute__等一些不涉及元类的方法执行其中的大多数任务。但是，自定义元类可以更快，因为只有在创建类的时候才进行特殊的处理，而创建类是一个并不常见的操作。自定义元类可以帮助开发者在一种框架内定义一套完整的类，这种框架可以奇迹般地获得已经编写的代码的任何有趣的行为，并且非常独立于这些类可能会选择定义的特殊方法。此外，类对象的某些行为只能在元类中自定义。下面的示例显示了如何使用元类来更改类对象的字符串格式：

```
class MyMeta(type):
    def __str__(cls): return "Beautiful class '%s'"%cls.__name__
class MyClass:
    __metaclass__ = MyMeta
x = MyClass()
print type(x)       # 打印: Beautiful class 'MyClass'
```

严格来说，实例化这样一个自定义元类的类既不是传统的，也不是新型的：类及其实例的语义是完全由其元类定义的。实际上，自定义元类几乎总是子类内置类型。因此，这种用来实例化这样一个自定义元类的类的语法是作为新型类的语法变体的最好方法。

自定义元类的一个真实示例

假定在 Python 中编程时，开发者不知道 C 的 struct 类型：就是一串具有固定名称的数据属性的对象。除了这些固定的名称之外，Python 还可以帮助开发者非常容易地定义一个适当的通用 Bunch 类：

```
class Bunch(object):
    def __init__(self, **fields): self.__dict__ = fields
p = Bunch(x=2.3, y=4.5)
print p                  # 打印: <__main__.Bunch object at 0x00AE8B10>
```

但是，自定义元类可以帮助开发者认识到这样一个事实，那就是属性名称在类被创建的时候是固定的。示例 5-1 中显示的代码定义了一个元类 metaMetaBunch 和一个类 MetaBunch，编写的代码显示如下：

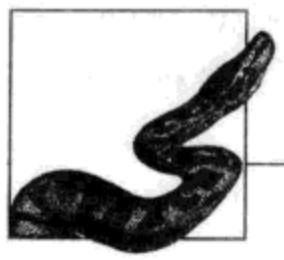
```
class Point(MetaBunch):
    """ A point has x and y coordinates, defaulting to 0.0, and a color,
        defaulting to 'gray' -- and nothing more, except what Python and
        the metaclass conspire to add, such as __init__ and __repr__
    """
    x = 0.0
    y = 0.0
    color = 'gray'
# Point 类的使用示例
q = Point()
print q                        # 打印: Point()
p = Point(x=1.2, y=3.4)
print p                        # 打印: Point(y=3.399999999, x=1.2)
```

在这段代码中，print 语句打印了 Point 实例的可读字符串表示。Point 实例是非常节省内存的，并且其性能与上一个示例中的简单类 Bunch 实例的性能基本上相同（由于隐式调用了特殊方法，因此没有额外开销）。请注意，示例 5-1 是一个非常实际的例子，所有的实现细节要求理解理解本书后面介绍的有关 Python 的各个方面的知识，比如字符串（参见本书第 9 章）和模块 warnings（参见第 18.3 节）。实例 5-1 中使用的标识符 mcl 表示“元类”，在这种特殊的高级情况下，这种使用方法要比 cls 表示“class”这种更习惯的用法更清晰。

示例 5-1　metaMetaBunch 元类

```
import warnings
class metaMetaBunch(type):
    """
    metaclass for new and improved "Bunch": implicitly defines __slots__,
    __init__ and __repr__ from variables bound in class scope.
    A class statement for an instance of metaMetaBunch (i.e., for a class
    whose metaclass is metaMetaBunch) must define only class-scope data
    attributes (and possibly special methods, but NOT __init__ and
    __repr__!).  metaMetaBunch removes the data attributes from class
    scope, snuggles them instead as items in a class-scope dict named
    __dflts__, and puts in the class a __slots__ with those attributes'
```

```
names, an __init__ that takes as optional keyword arguments each of
them (using the values in __dflts__ as defaults for missing ones), and
a __repr__ that shows the repr of each attribute that differs from its
default value (the output of __repr__ can be passed to __eval__ to
make an equal instance, as per the usual convention in the matter, if
each of the non-default-valued attributes respects the convention too)
"""
def __new__ (mcl, classname, bases, classdict):
    """ Everything needs to be done in __new__, since type.__new__ is
        where __slots__ are taken into account.
    """
    # 将要在新类中使用的__init__和__repr__定义为本地函数
    def __init__(self, **kw):
        """ Simplistic __init__: first set all attributes to default
            values, then override those explicitly passed in kw.
        """
        for k in self.__dflts__: setattr(self, k, self.__dflts__[k])
        for k in kw: setattr(self, k, kw[k])
    def __repr__(self):
        """ Clever __repr__: show only attributes that differ from the
            respective default values, for compactness.
        """
        rep = ['%s=%r' % (k, getattr(self, k)) for k in self. __dflts__
                 if getattr(self, k) != self.__dflts__[k]
             ]
        return '%s(%s)' % (classname, ', '.join(rep))
    # 构建将要用作新类的类字典的 newdict
    newdict = { '__slots__':[ ], '__dflts__':{ },
        '__init__': __init__, '__repr__': __repr__, }
    for k in classdict:
        if k.startswith('__') and k.endswith('__'):
            # 特殊方法：复制到 newdict，警告冲突
            if k in newdict:
                warnings.warn("Can't set attr %r in bunch-class %r"
                    % (k, classname))
            else:
                newdict[k] = classdict[k]
        else:
            # 类变量，将名称保存到__slots__中，并且名称和值
            # 作为__dflts__中的一个项目
            newdict['__slots__'].append(k)
            newdict['__dflts__'][k] = classdict[k]
    # 最后将其余的工作委托给类型.__new__
    return super(metaMetaBunch, mcl).__new__(
                 mcl, classname, bases, newdict)
class MetaBunch(object):
    """ For convenience: inheriting from MetaBunch can be used to get
        the new metaclass (same as defining __metaclass__ yourself).
    """
    __metaclass__ = metaMetaBunch
```



第 6 章

异常

Python 使用异常来通告错误和不寻常的情况。异常（exception）是一个对象，表示错误或不寻常的情况。在 Python 检测到一个错误时，将引发一个异常——也就是说，Python 可以通过向异常传播机制传递一个异常对象，发出一个异常情况出现的信号。开发者的代码可以通过执行 raise 语句显式引发一个异常。

处理（Handling）异常表示从传播机制接收异常对象，并执行所需的各种操作以处理这种异常情况。如果一个异常处理程序不处理收到的异常，则该程序将在输出错误跟踪消息之后终止运行。不过，程序可以对异常进行处理，并在出现错误或其他反常的情况下仍保持运行。

Python 还使用异常来表示某些不是错误，甚至不是异常情况的特殊情况。例如，参见第 4.10 节中介绍的迭代器，一个迭代器的 next 方法将在该迭代器没有更多的项目之后引发 StopIteration 异常。这个异常并不是一个错误，甚至不是一种异常情况，因为大多数迭代器最终都会用完所有的项目。因此，Python 中的这种用来检查和处理错误和其他特殊情况的最优策略不同于其他语言中可能是最好的其他策略，本章将在第 6.6 节中详细介绍这些策略。本章还将在第 6.6 节中介绍 Python 标准库中的 logging 模块，在第 6.6 节中介绍 Python 的 assert 语句。

6.1 try 语句

try 语句提供了 Python 的异常处理机制。这是一个复合语句，可以采用以下这两种不同的形式：

- try 子句后面带一个或多个 except 子句（和一个可选的 else 子句），
- try 子句后面只带一个 finally 子句。

在 Python 2.5 中，try 语句可以在 except 子句（和一个可选的 else 子句）后面带一个 finally 子句；但是，在 Python 的所有以前版本中，不能合并使用以上这两种形式，因此下面将分别介绍这两种形式。参见第 6.1 节中介绍的 try/except/finally 语句以了解 Python 2.5 中对 try 语句语法的这个小的增强功能。

try/except

下面是 try 语句的 try/except 形式的语法：

```
try:
    statement(s)
except [expression [, target]]:
    statement(s)
[else:
    statement(s)]
```

try 语句的这种形式具有一个或多个 except 子句，以及一个可选的 else 子句。

每个 except 子句中的代码被称为异常处理程序（exception handler）。如果 except 子句中的 expression 与从 try 子句传递的异常对象匹配，则执行这段代码。expression 是一个类或者一个由类组成的元组，并与这些类中的一个类或者任何一个子类的任何实例匹配。可选的 target 是一个标识符，用来为一个变量命名，Python 将这个变量绑定到执行异常处理程序之前的异常对象上。异常处理程序还可以通过调用 sys 模块的 exc_info 函数（参见第 8.3 节）获得当前的异常对象。

下面是 try 语句的 try/except 形式的一个示例：

```
try: 1/0
except ZeroDivisionError: print "caught divide-by-0 attempt"
```

如果一个 try 语句有多个 except 子句，异常传播机制将按顺序测试这些 except 子句；其表达式与异常对象匹配的第一个 except 子句将被用作异常处理程序。因此，在列出更多一般情况的处理程序之前，必须总是列出特殊情况的处理程序。如果首先列出了一般情况，则后面的更特殊情况的 except 子句将永远得不到执行。

最后的 except 子句可以不带表达式。这个子句可以处理传播过程中到达这条子句的任何异常。这种无条件处理程序是极少需要的，但也会发生，通常会发生在重新引发一个异常之前，必须执行某些额外任务的包装器函数中，参见第 6.3 节。请小心使用“裸 except”（不带表达式的 except 子句），除非要在其中重新引发一个异常：这种随便的风格可能会产生非常难以找到的错误，因为裸 except 通常覆盖过于宽泛，并且可以很容易屏蔽编程错误和其他类型的错误。

异常传播会在找到了一个其表达式与异常对象匹配的处理程序之后终止。因此，如果一个 try 语句被嵌套在另一个 try 语句的 try 子句中，在异常传播期间，将首先到达这个内部 try 创建的处理程序，因此，如果这个 try 匹配指定的表达式，则其就是用来处理

这个异常的异常处理程序。例如：

```
try:
    try: 1/0
    except: print "caught an exception"
except ZeroDivisionError:
    print "caught divide-by-0 attempt"
# 打印: caught an exception
```

在这种情况下，尽管外部 try 子句中的子句 except ZeroDivisionError: 建立的异常处理程序要比内部 try 子句中的所有 except: 子句更特殊，也更适合，但还是执行了内部 try 中的异常处理程序。外部 try 甚至得不到执行的机会，因为异常没有从内部 try 传播出去。要想获得更多有关异常传播的详细信息，参见第 6.2 节。

只有在 try 子句正常终止之后才会执行 try/except 的可选 else 子句。换句话讲，在从 try 子句传播异常时，或者在 try 子句使用 break、continue 或 return 语句退出时，都不会执行 else 子句。try/except 建立的异常处理程序只包含 try 子句，而不包括 else 子句。else 子句可以用来避免意外处理无法预料的异常。例如：

```
print repr(value), "is ",
try:
    value + 0
except TypeError:
    # 不是一个数字，可能是一个字符串、Unicode 和 UserString...?
    try:
        value + ''
    except TypeError:
        print "neither a number nor a string"
    else:
        print "a string or string-like value"
else:
    print "some kind of number"
```

try/finally

下面是 try 语句的 try/finally 形式的语法：

```
try:
    statement(s)
finally:
    statement(s)
```

这种形式只带有一个 finally 子句，并且不能带有 else 子句。

finally 子句建立了被称为清理处理程序（clean-up handler）的程序。这段代码总是在 try 子句以某种方式终止之后执行。在从 try 子句传播一个异常时，try 子句将会终止，而清理处理程序将会执行，并且异常将继续传播。在没有异常发生时，清理处理程序也会执行，不管 try 子句执行到了最后，还是由于执行了 break、continue 或 return 语

句而退出。

使用 try/finally 建立的清理处理程序提供了一种健壮和显式方法，以指定不管在什么情况下都必须总是执行的终止代码，这样可以确保程序状态和/或外部实体（例如，文件、数据库、网络连接）的一致性。下面是 try 语句的 try/finally 形式的一个示例：

```
f = open(someFile, "w")
try:
    do_something_with_file(f)
finally:
    f.close()
```

请注意，try/finally 形式与 try/except 形式有显著的区别：try 语句不能同时有 except 和 finally 子句，除非开发者使用的是 Python 2.5。如果开发者需要用到异常处理程序和清理处理程序，并且代码必须在 Python 2.3 或 2.4 下运行，可以在另一个 try 语句的 try 子句中嵌套一个 try 语句来显式和明确地定义执行的顺序。

finally 子句不能直接包含 continue 语句，但是该子句可以包含 break 或 return 语句。但是，这种用法会导致开发者的程序不太清楚，因为在 break 或 return 这样的语句执行时，异常传播将会停止。大多数程序员通常并不期望在 finally 子句中停止异常传播，因此，这种用法可能会让那些阅读代码的人感到混淆。在 Python 2.3 和 2.4 中，try/finally 语句不能包含 yield 语句（而在 Python 2.5 中是可以的）。

Python 2.5 中与异常相关的增强功能

除了允许 try 语句同时包含 except 子句和 finally 子句，Python 2.5 还引入了一个新的 with 语句，这个语句可以很好地将 try/except 的许多用法包装到一个简明和便捷的语法中。Python 2.5 还包含对生成器的一些增强功能，这使得生成器可以更好地与新的 with 语句和普通异常语句一起使用。

try/except/finally 语句

Python 2.5 独有的 try/except/finally 语句，例如：

```
try:
    ...guarded clause...
except...expression...:
    ...exception handler code...
finally:
    ...clean-up code...
```

这条语句等同于以下语句（可以在 Python 2.5 和任何以前版本中使用）：

```
try:
    try:
        ...guarded clause...
    except...expression...:
```

```
        ...exception handler code...
finally:
    ...clean-up code...
```

上面的语句中还可以在结束 finally 子句之前包含多个 except 子句和一个可选的 else 子句。在所有变体中，其效果总是与上面的语句相似——也就是说，就像将一个 try/except 语句，包括所有的 except 子句和 else 子句（如果有的话），嵌套到一个包含 try/finally 语句中。

with 语句

with 语句是 Python 2.5 中新引入的一条语句。要想在 Python 2.5 中的给定模块中使用这条语句，必须在该模块的起始位置添加以下指令以将 with 变成一个关键字：

```
from __future__ import with_statement
```

在 Python 2.6 及其以后版本中，with 直接就是一个关键字，这样，就不再需要用到上面的这条语句了（尽管这条语句也可以被接受，同时也没有什么害处）。

with 具有以下语法：

```
with expression [as varname]
    statement(s)
```

with 的语义定义相当于：

```
_normal_exit = True
_temporary = expression
varname = _temporary.__enter__()
try:
    statement(s)
except:
    _normal_exit = False
    if not _temporary.__exit__(*sys.exc_info()):
        raise
    # 如果__exit__最终返回一个真实值，将不传播这个异常：
    if _normal_exit:
        _temporary.__exit__(None, None, None)
```

其中_temporary 和_normal_exit 是当前范围的其他任何位置都没有使用过的任意内部名称。如果省略 with 子句的可选部分 as varname，Python 仍将调用_temporary.__enter__()，但是不将结果绑定到任何名称上，并且仍然会在程序结束时调用_temporary.__exit__()。

这个新的 with 语句具体化了著名的 C++习惯用法“资源获取即初始化”：开发者只需要使用两个新的特殊方法__enter__（可以不带参数调用）和__exit__（可以在调用时带 3 个参数；如果这段代码不需要传播异常，则这 3 个参数可以全部为 None，否则这 3 个参数分别是异常的类型、值和跟踪）来编写类，这样可以获得与 C++的 auto 变量中

使用的典型 ctor/dtor 对的行为相同的有保证的终止化行为（还将具有增加的功能，可以根据开发者的期望以不同的方式终止传播，还可以通过从__exit__返回一个真实值来可选地阻止一个正在传播的异常）。

例如，使用 Python 2.5 有一种方法可以包装文件打开功能，这样是为了确保在打开该文件时文件是关闭的：

```
class opened(object):
    def __init__(self, filename, mode='r'):
        self.f = open(filename, mode)
    def __enter__(self):
        return self.f
    def __exit__(self, etyp, einst, etb):
        self.f.close()
# 将被用作:
with opened('foo.txt') as f:
   ...使用已打开的文件对象 f 的语句...
```

这个示例代码并不是特别地有用，因为在 Python 2.5 内置文件对象中已经提供了所需的方法__enter__和__exit__；这个示例的目的纯粹只是为了演示。

建立这样的包装器还有一种更简单的方法，标准 Python 库的新 contextlib 模块最有可能提供这种方法：这个模块计划包含一个装饰器（可能会被命名为 contextmanager），该装饰器可以将生成器函数变成这样的包装器的工厂，以及包装器函数 closing（通过__exit__调用某些对象的关闭方法）和 nested（以嵌套多个这样的包装器以在单个 with 语句中使用）。

读者可以在位于 http://www.python.org/peps/pep-0343.html 的 PEP 343 中找到更多示例和详细介绍。

生成器增强功能

在 Python 2.5 中，try/finally 语句中允许使用 yield 语句（在以前的版本中，是不允许使用这种组合的）。此外，现在的生成器对象具有两个其他的新方法，throw 和 close（除了第 4.11 节中在介绍 Python 2.5 中的生成器时提到的另一个新方法 send）。给定一个通过调用生成器函数建立的生成器对象 g，throw 方法的语法如下：

```
g.throw(exc_type, exc_value=None, exc_traceback=None)
```

在该生成器的调用者调用 g.throw 时，其效果就像是在生成器 g 被暂停的 yield 位置执行带有相同参数的 raise 语句。close 方法没有参数；在生成器的调用者调用 g.close() 时，其效果就像调用 g.throw(GeneratorExit)。GeneratorExit 是在 Python 2.5 中新引入的一个内置异常类，这个类是直接从 Exception 继承的。生成器的 close 方法必须重新引发（或传播）GeneratorExit 异常，或者等同于在进行了生成器可能需要的任何清理操作之后引发 StopIteration。现在，生成器还具有一个完全等同于 close 方法的析构函数（特

殊方法_ _del_ _）。

要想获得有关 Python 2.5 的生成器增强功能的更多详细信息，以及如何使用这些增强功能的示例（例如，要使用生成器实现协同程序），参见 http://www.python.org/peps/pep-0342.html 上的 PEP 342。

6.2 异常传播

在引发一个异常时，异常传播机制将取得控制权。该程序的常规控制流将会停止，Python 将查找一个适当异常处理程序。Python 的 try 语句将通过其 except 子句建立异常处理程序。该处理程序可以处理 try 子句的程序体中引发的异常，以及直接或间接从这段代码调用的任何函数传播的异常。如果异常是在一个具有可用的 except 处理程序的 try 子句内引发的，这个 try 子句将会终止，并将执行异常处理程序。在异常处理程序结束时，将继续执行 try 语句后面的语句。

如果引发异常的语句不在一个具有可用的处理程序的 try 子句中，包含该语句的函数将会终止，而异常将沿着函数调用的堆栈“向上”传播到调用该函数的语句。如果对终止的函数的调用在具有可用的处理程序的 try 子句中，该 try 子句将会终止，而处理程序将会执行。否则，包含这个调用的函数将会终止，而这个传播过程将会重复，并将展开函数调用的堆栈，直到一个可用的异常处理程序被找到。

如果 Python 不能找到任何可用的处理程序，在默认情况下，该程序将向标准错误流（文件 sys.stderr）打印一条错误消息。这个错误消息包含一个跟踪，给出了有关传播期间终止的函数的详细信息。开发者可以通过设置 sys.excepthook（参见第 8.3 节）更改 Python 的默认错误报告行为。在报告错误之后，Python 将返回到交互式会话（如果有的话），如果没有交互式会话是活动的，Python 将会终止运行。在异常类是 SystemExit 时，终止行为是静默的，并将结束该交互式会话（如果有）。

下面是可以用来查看异常传播的一些函数：

```
def f():
    print "in f, before 1/0"
    1/0                          # 引发一个 ZeroDivisionError 异常
    print "in f, after 1/0"

def g():
    print "in g, before f()"
    f()
    print "in g, after f()"

def h():
    print "in h, before g()"
    try:
```

```
        g()
        print "in h, after g()"
    except ZeroDivisionError:
        print "ZD exception caught"
    print "function h ends"
```

调用 h 函数将产生以下结果：

```
>>> h()
in h, before g()
in g, before f()
in f, before 1/0
ZD exception caught
function h ends
```

函数 h 将建立一个 try 语句，并在 try 子句中调用函数 g。函数 g 将按顺序调用 f，该函数将执行一个除 0 运算，引发一个类 ZeroDivisionError 异常。这个异常将传播回到 h 中的 except 子句。在异常传播过程中，函数 f 和 h 将会终止，这就是为什么这些函数“之后的”消息没有打印的原因。在异常传播过程中，h 的 try 子句的执行也将终止，因此也不会打印该函数“之后的”消息。在异常处理程序执行完，也就是在 h 的 try/except 程序块结束之后，将继续执行该函数。

6.3 raise 语句

开发者可以使用 raise 语句显式引发一个异常。raise 是一个具有以下语法的简单语句：

```
raise [expression1[, expression2]]
```

只有异常处理程序（或者处理程序直接或间接调用的函数）可以不带任何参数使用 raise 语句。一个单纯的 raise 语句将重新引发该处理程序接收到的相同异常对象。处理程序终止后，异常传播机制将继续搜索其他可用的处理程序。在处理程序发现自己不能处理接收的异常，或者只能部分地处理这个异常时，使用不带参数的 raise 是很有用的，这样，该异常将继续传播，以允许调用堆栈上面的处理程序执行异常处理和清理。

在只有 expression1 时，raise 语句可以是一个传统型的实例对象或者类对象（Python 2.5 更改这个规则是为了允许新型类，只要这些类继承自新的内置新型类 BaseException 和 BaseException 的实例）。在这种情况下，如果 expression1 是一个实例对象，Python 将引发这个实例。在 expression1 是一个类对象时，raise 将不带参数地实例化这个类并引发结果实例。在给出了两个表达式时，expression1 必须是一个传统型的类对象。raise 将实例化该类，并使用 expression2 作为参数（如果 expression2 是一个元组，则为多个参数），并引发结果实例。请注意，raise 语句是 Python 2.3 和 2.4 中剩余的唯一构造函数，这两个 Python 版本中只允许使用传统类和实例，而不允许新型类和实例；另一方面，在 Python 2.5 中，内置异常类都是新型的，尽管为了向后兼容，在所

有的 Python 2.x 版本中，raise 语句中都仍将允许传统类，而只是在 Python 3.0 中将被删除。

下面是 raise 语句的一个典型使用的示例：

```
def crossProduct(seq1, seq2):
    if not seq1 or not seq2:
        raise ValueError, "Sequence arguments must be non-empty"
    return [(x1, x2) for x1 in seq1 for x2 in seq2]
```

crossProduct 函数将返回一个由所有值对组成的列表，每个项目都来自于其每个序列参数，在函数返回之前将首先测试这两个参数。如果任何一个参数为空，该函数将引发 ValueError，而不是像列表推导通常所作的那样，只返回一个空白列表。请注意，crossProduct 没有必要测试 seq1 和 seq2 是否是可迭代的：如果两个都不是，则列表推导本身将引发适当的异常，可能是一个 TypeError。一旦引发了一个异常，不管是由 Python 本身引发的还是在代码中使用一个显式 raise 语句引发，只能由调用者决定是否处理这个语句（使用合适的 try/except 语句）或者让其进一步在调用堆栈上传播。

只有在为通常被认为是正确的，但是开发者定义其为错误的情况下才可以使用 raise 语句来引发附加的异常。不要使用 raise 重复 Python 可以显式执行的错误检查操作。

6.4 异常对象

异常是内置的传统型 Exception 类的子类的实例（为了向后兼容，Python 还允许开发者使用字符串或者任何传统类的实例作为异常对象，但是这样的用法可能会在将来产生不兼容的风险，而且这样用也没有什么好处）。Exception 的任何子类的实例都有一个属性 args，这个属性是由用来创建该实例的参数组成的元组。args 保存了特定的错误信息，可以用于错误的诊断或恢复。在 Python 2.5 中，类 Exception 是一个新型类，因为该类是从新的内置新型类 BaseException 继承的；这样将依次使得 Python 2.5 中的所有内置异常类成为新型类。要想获得有关标准异常层次结构在 Python 2.5 中是怎样更改的，以及异常功能在 Python 3.0 中将如何进一步发展的详细说明，请参见 http://python.org/doc/peps/pep-0352/。

标准异常的层次结构

Python 自身引发的所有异常都是 Exception 的子类的实例。异常类的继承结构是很重要的，因为其确定了哪些 except 子句处理哪些异常。但是，在 Python 2.5 中，类 KeyboardInterrupt 和 SystemExit 都是直接从新类 BaseException 继承的，而不是 Exception 的子类：这种新的安排更像是代码为 except Exception:的普通处理器子句完成了想要的操作，因为开发者极少想要捕获 KeyboardInterrupt 和 SystemExit 异常（本章第 6.1 节中

介绍了异常处理程序）。

在 Python 2.3 和 Python 2.4 中，SystemExit、StopIteration 和 Warning 类是直接从 Exception 继承的。SystemExit 的实例通常是由 sys 模块（参见第 8.3 节）中的 exit 函数引发的。StopIteration 用在迭代协议中，参见第 4.10 节中介绍的迭代器。第 18.3 节中介绍了 warning。其他的标准异常是由 StandardError 派生的，StandardError 是 Exception 的一个直接子类。在 Python 2.5 中，类 GeneratorExit 也是直接从 Exception 继承的，参见第 6.1 节中介绍的生成器增强功能，而其他的标准异常层次结构与 Python 2.3 和 2.4 中的相似，下面将对此进行介绍。

与 StandardError 本身和 Exception 一样，StandardError 的 3 个子类不能被直接初始化（至少，不能由 Python 本身初始化）。这些子类的目的就是可以让开发者更简单地指定用来处理大量相关错误的 except 子句。StandardError 的这 3 个抽象子类是：

ArithmeticError

由于算术错误而引发的异常的基类（也就是，OverflowError、ZeroDivisionError 和 FloatingPointError）

LookupError

容器在接收到一个无效键或索引（也就是，IndexError 和 KeyError）时引发的异常的基类。

EnvironmentError

由于外部原因（也就是，IOError、OSError 和 WindowsError）而导致的异常的基类。

```
Exception
    SystemExit
    StandardError
          AssertionError, AttributeError, ImportError,
          KeyboardInterrupt, MemoryError,
          NotImplementedError, SystemError, TypeError,
          UnicodeError, ValueError
          NameError
                UnboundLocalError
          SyntaxError
                IndentationError
          ArithmeticError
                FloatingPointError, OverflowError, ZeroDivisionError
          LookupError
                IndexError, KeyError
          EnvironmentError
                IOError
                OSError
                      WindowsError
```

```
StopIteration
Warning
```

标准异常类

普通运行时错误将引发以下类型的异常：

AssertionError

断言语句失败。

AttributeError

属性引用或赋值失败。

FloatingPointError

浮点型运算失败。派生自 ArithmeticError。

IOError

I/O 操作失败（例如，硬盘满、文件没找到，或者没有所需的权限）。派生自 EnvironmentError。

ImportError

import 语句（参见第 7.1 节）不能找到要导入的模块，或者不能找到该模块特别请求的名称。

IndentationError

解析器遇到了一个由于错误的缩进而引发的语法错误。派生自 SyntaxError。

IndexError

用来索引序列的整数超出了范围（使用非整数作为序列索引将引发 TypeError）。派生自 LookupError。

KeyError

用来索引映射的键不在映射中。派生自 LookupError。

KeyboardInterrupt

用户按了中断键（Ctrl+C、Ctrl+Break 或 Delete 键，取决于系统平台）。

MemoryError

一个运算用光了内存。

NameError

引用了一个变量，但是没有找到该变量的名称。

NotImplementedError

由抽象基类引发的异常，用来指示一个具体的子类必须覆盖一个方法。

OSError

由模块 os（参见第 10.7 节和第 14.7 节）中的函数引发的异常，用来指示平台相关错误。派生自 EnvironmentError。

OverflowError

对一个整数进行运算的结果太大，以至于不能放到一个整数中（运算符 << 不引发这个异常，而是丢弃超出的位）。派生自 ArithmeticError。这个异常要回溯到 Python 2.1 版本；在今天的 Python 版本中，太大的整数结果将隐式变成长整型，而不是引发异常。这个标准异常保留了一个内置对象以供向后兼容（为了支持引发或试图捕获该异常的现有代码）。不要在任何新代码中使用这个异常类。

SyntaxError

解析器遇到了一个语法错误。

SystemError

Python 本身或某些扩展模块中的内部错误。开发者必须将这个异常报告给 Python 或这些扩展模块的作者和维护者，并提供所有可能的细节以让他们可以重现这个错误。

TypeError

应用于某个不适当类型的对象的操作或函数。

UnboundLocalError

对一个本地变量进行的引用，但是当前没有值被绑定到这个本地变量。派生自 NameError。

UnicodeError

在将 Unicode 转换为字符串，或者反向操作时出现的错误。

ValueError

应用于某个对象的操作或函数，这个对象具有正确类型但是不适当的值，没有任何特殊应用（例如，KeyError）。

WindowsError

模块 os 中的函数引发的异常（参见第 10.7 节和第 14.7 节），用来指示与 Windows 相关的错误。派生自 OsError。

ZeroDivisionError

除数为 0（/、// 或 % 运算符右边的操作数，或者内置函数 divmod 的第二个参数）。派生自 ArithmeticError。

6.5 自定义异常类

为了定义开发者自己的异常类，开发者可以定义任何标准异常类的子类。通常，这样的子类只需要添加文档字符串：

```
class InvalidAttribute(AttributeError):
    "Used to indicate attributes that could never be valid"
```

正如第 4.10 节中介绍的，开发者不需要使用 pass 语句来组成类体；文档字符串（开发者总是需要编写该字符串）足以让 Python 满意了。就像“空白”函数一样，这样的“空白”类的最好风格就是包含一个文档字符串，但是不包含 pass 语句。

根据给定的 try/except 语义，引发一个像 InvalidAttribute 这样的自定义异常类几乎与引发其标准异常超类 AttributeError 完全一样。任何可以处理 AttributeError 的 except 子句同样也可以处理 InvalidAttribute。此外，了解 InvalidAttribute 自定义异常类的特别之处的客户代码可以特别地处理该异常类，如果客户代码没有准备处理所有其他情况，则不必处理 AttributeError 的这些情况。例如：

```
class SomeFunkyClass(object):
    "much hypothetical functionality snipped"
    def __getattr__(self, name):
        "this __getattr__ only clarifies the kind of attribute error"
        if name.startswith('_'):
            raise InvalidAttribute, "Unknown private attribute "+name
        else:
            raise AttributeError, "Unknown attribute "+name
```

现在，客户代码可以更有选择地使用其处理程序。例如：

```
s = SomeFunkyClass()
try:
    value = getattr(s, thename)
except InvalidAttribute, err:
    warnings.warn(str(err))
    value = None
# AttributeError 的其他情况将以无法预料的方式进行传播
```

与普通标准异常相比，在模块中定义并引发自定义异常类是最好的方法：通过使用自定义异常，可以让模块代码的所有调用者更容易地处理来自于该模块的异常，并将其与其他异常区分开。

开发者有时候会发现非常有用的自定义异常类的一种特殊情况就是包装另一个异常并

添加进一步的信息。要想收集有关一个等待（pending）异常的信息，开发者可以使用来自 sys 模块的 exc_info 函数（参见第 8.3 节）。基于这一点，开发者的自定义异常类可以定义如下：

```
import sys
class CustomException(Exception):
    "Wrap arbitrary pending exception, if any, in addition to other info"
    def __init__(self, *args):
        Exception. __init__(self, *args)
        self.wrapped_exc = sys.exc_info()
```

然后，开发者可以非常典型地在下面这个包装器函数中使用这个类：

```
def call_wrapped(callable, *args, **kwds):
    try: return callable(*args, **kwds)
    except: raise CustomException, "Wrapped function propagated
exception"
```

自定义异常和多重继承

对于自定义异常，一种特别有效的方法（但是，这种方法在 Python 的实际应用中并不经常使用）就是从开发者的模块的特殊自定义异常类和一个标准异常类多重继承异常类，就像下面的代码片断一样：

```
class CustomAttributeError(CustomException, AttributeError): pass
```

现在，当开发者的代码引发一个 CustomAttributeError 实例时，这个异常可以被用来捕获 AttributeError 的所有情况的调用代码，以及用来捕获开发者模块引发的所有异常的代码捕获。在开发者必须决定是否需要引发一个特殊标准异常，比如 AttributeError，或者开发者在模块中自定义的异常类时，可以考虑这种多重继承方案，这种方案提供了两种方式中的最好选择。开发者要确认将模块的这个特性清楚地放到文档中；因为刚才介绍的这个技术并没有被广泛使用，除非开发者将所作的操作清楚显式地放到文档中，否则使用这个模块的用户不会想到这样使用。

标准库中使用的其他异常

Python 的标准库中的许多模块都定义了自己的异常类，这些类等同于开发者自己的模块可以定义的自定义异常类。通常，除了本章第 6.4 节中介绍的标准层次结构中的异常，这样的标准库模块中的所有函数都可能引发这些类型的异常。例如，模块 socket 提供了类 socket.error，这个类直接派生自内置类 Exception，以及 error 的几个子类 sslerror、timeout、gaierror 和 herror；除了标准异常，模块 socket 中的所有函数和方法都有可能引发类 socket.error 及其子类的异常。对应于提供这些异常类的标准库模块，本书将在后续章节介绍这些异常类的主要情况。

6.6 错误检查策略

大多数支持异常的编程语言都只适合于在极少的情况下才引发异常。Python 的重点有所不同。在 Python 中，只要异常可以让程序更简单和更健壮，都可以适当地考虑使用异常，即使这意味着异常的引发会非常频繁。

LBYL 对比 EAFP

在其他编程语言中使用的一个通用的习惯用法，有时候被称为“三思而后行”（Look Before You Leap，LBYL），也就是说，在尝试进行一个操作之前，提前检查可能会导致这个操作非法的所有情况。出于以下几个原因，这种方案并不是很理想：

- 这些检查可能会导致在所有情况都很正常的时候降低常规主要操作的可读性和清晰度；
- 检查所需的工作可能会重复操作本身已经完成的工作中的一部分具体操作；
- 程序员可能会因为忽略某些必要的检查而很容易产生错误；
- 在执行检查的时刻和执行操作的时刻之间，情况可能会发生改变。

Python 中首选的习惯用法通常都是在 try 子句中尝试某个操作，在 except 子句中处理可能会引发的异常。这个习惯用法被称为“要求谅解要比要求许可更容易”（it's easier to ask forgiveness than permission，EAFP），普遍认为这是海军上将 Grace Murray Hopper，COBOL 的合作发明人的座右铭，这种做法可以避免 LBYL 的所有缺点。下面是使用 LBYL 习惯用法编写的函数：

```
def safe_divide_1(x, y):
    if y==0:
        print "Divide-by-0 attempt detected"
        return None
    else:
        return x/y
```

使用 LBYL，将首先进行检查，而主要操作将出现在函数的末尾。下面是使用 EAFP 习惯用法编写的具有相同功能的函数：

```
def safe_divide_2(x, y):
    try:
        return x/y
    except ZeroDivisionError:
        print "Divide-by-0 attempt detected"
        return None
```

使用 EAFP，主要操作出现在前面的 try 子句中，而异常情况将在 except 子句中进行处理。

EAFP 的正确用法

在通常情况下，EAFP 是最可取的错误处理策略，但是 EAFP 也不是万能的。特别是，开发者必须非常小心，不要建立一个过于宽泛的网络以捕获开发者无法预料的错误，并因此导致毫无意义地捕获。下面是这种风险情况的一个典型实例（第 8.2 节中介绍的 getattr 函数）。

```
def trycalling(obj, attrib, default, *args, **kwds):
    try: return getattr(obj, attrib)(*args, **kwds)
    except AttributeError: return default
```

函数 trycalling 的目的是尝试调用对象 obj 的一个名为 attrib 的方法，但是，如果 obj 没有名为 attrib 的方法，则返回 default。不过，这段函数代码并不只是完成这个功能：这段代码还错误地隐藏了一些错误情况，就在这个广受欢迎的方法的执行过程中引发 AttributeError 的地方，在所有这些错误情况下，该方法将返回 default。这样可能会很容易隐藏其他代码中的错误。要想确切地实现这段代码的目标，必须更小心地编写这个函数：

```
def trycalling(obj, attrib, default, *args, **kwds):
    try: method = getattr(obj, attrib)
    except AttributeError: return default
    else: return method(*args, **kwds)
```

trycalling 函数的这种实现方法将 getattr 函数调用从整个方法的调用中分离出来，将其放在 try 字句中，从而可以通过 except 子句中的处理程序来查看这个函数调用，而整个方法的调用被放在 else 子句中，从而可以自由地传播可能需要的任何异常。以最有效的方式使用 EAFP 还包括在 try/except 语句中频繁地使用 else 子句。

在大程序中处理错误

在大程序中，特别容易因为将 try/except 语句定义得太宽泛而导致错误，尤其是在开发者很自信地将 EAFP 的强大功能作为通用错误检查策略的情况下。在可以捕获到太多的不同错误，或者一个错误可能会出现在太多不同的位置时，try/except 组合显得过于宽泛。如果开发者需要确切地区别在什么位置和发生了什么错误，并且跟踪信息不足以确定这些细节（或者在跟踪信息中丢弃了所有或部分信息）时，使用宽泛的 try/except 就会产生问题。要想有效地进行错误处理，必须清晰地区分错误、开发者预料到的异常情况（并确切地知道如何处理这种异常情况）、没有预料到的错误，以及用来指示程序中小错误的异常情况。

有些错误和异常情况并不真正都是错误的，甚至有可能并不都是反常的：只是一些特殊情况，可能极少出现但是是完全可以预料到的，因此，选择使用 EAFP 而不是使用 LBYL 进行处理可以避免使用 LBYL 的许多内在缺陷。在这些情况下，开发者只需要处理异常情况，通常甚至没有记录日志或报告异常。在这种环境下，要非常小心地将

相关的 try/except 设定到一个切实可行的窄范围之内。通常可以使用较小的 try 子句，在该子句中包含一小段不调用太多其他函数的代码，并在 except 子句中包含非常特殊的异常类元组。

从某种程度上讲，开发者总是可以确定地预料到一些取决于用户输入或其他外部条件，并且不受开发者控制的错误和异常，这是因为开发者并不能对这些错误和异常的潜在原因进行控制。在这种情况下，开发者必须将精力集中在对异常进行适当的处理上，通常就是报告并记录异常的准确特性和详细信息，并让程序中未受损害的内部功能以稳定的状态保持运行。在这些情况下，try/except 子句涉及的幅度也需要合理地收窄，尽管这与开发者使用 EAFP 构造自己的用来处理并非真正错误的特殊情况的处理程序，并不是同样地非常至关重要。

最后，完全无法预料的错误和异常表示开发者的程序设计或编程中出现了错误。在大多数情况下，对于这些错误，最好的策略就是要避免 try/except，并只是让程序以错误和跟踪消息结束（开发者可能想要将这些信息记入日志和/或使用 sys.excepthook 中的应用程序专用钩子函数更适当地显示这些信息，接下来将对此进行介绍）。在开发者必须不惜一切代价让程序保持运行这种未必可能的情况下，甚至程序的运行环境非常糟糕，那么采用非常宽泛的 try/except 语句可能是合适的，可以使用执行大粒度的程序功能的 try 子句守卫函数调用和多个 except 子句。

在长时间运行的程序的情况下，请确保异常或错误的所有详细信息都被记录到某个持久化位置，以供将来进行分析（并且还将显示这个问题的一些指示信息，这样可以知道将来的分析是必要的）。关键就是要确保程序的稳定状态可以被还原到某些未受损害的内部一致的位置。这种可以让长时间运行的程序幸免于其自身的一些错误的技术被称为“检查站”（checkpointing）和“事务行为”（transactional behavior），不过本书并没有进一步介绍这些技术。

日志错误

在 Python 将一个异常一路传播到堆栈的顶部，但是没有找到一个适用的处理程序时，Python 解释器通常会在程序结束之前将错误跟踪信息打印到这个处理过程（sys.stderr）的标准错误流上。为了将这个错误跟踪信息转移到更适合使用的目的地，开发者可以将 sys.stderr 重新绑定到可用于输出的任何类文件对象上。

在开发者有时想要更改信息的数量和类型时，重新绑定 sys.stderr 是不够的。在这些情况下，开发者可以将自己定义的函数赋值给 sys.excepthook，而 Python 将在由于未经处理的异常而结束程序之前调用这个函数。在开发者的异常报告函数中，可以输出开发者认为有助于诊断和调试错误的任何信息，并将这个信息发送到开发者喜欢的任何目的地。例如，开发者可以使用模块 traceback（参见第 18.2 节）来格式化跟踪堆栈。在异常报告函数结束时，程序也就结束了。

日志模块

Python 标准库提供了丰富和强大的 logging 软件包，可以让开发者以系统和灵活的方式管理来自开发者的应用程序的消息日志。开发者可以管理 Logger 类和子类的整个层次结构，再加上 Handler 的实例（以及其中的子类），可能还有插入到调优标准的类 Filter 的实例，这个标准确定了哪些消息将按哪种方式记录到日志中，发出的消息将被 Formatter 类的实例格式化——实际上，这些消息本身就是 LogRecord 类的实例。logging 软件包甚至还包括一个动态配置工具，由此可以通过从硬盘文件读取，甚至通过在一个专门的线程的专用套接字上接收这些日志配置文件对日志进行动态配置。

由于 logging 软件包具有极其复杂和强大的体系结构，适合于实现高度复杂的日志策略和原则，而这些策略在大型和复杂的编程系统和在许多应用程序中可能是需要的，在这些应用程序中，开发者可以通过 logging 模块本身提供的一些简单函数使用该软件包的一个微小的子集来回避这种复杂性。首先是 import logging。然后，通过将日志消息作为字符串传递到按严重程度递增的顺序排列的任何一个函数 debug、info、warning、error 或 critical 中来发送消息。如果传递的字符串包含像%s 这样的格式标识符（参见第 9.3 节），在这个字符串之后，必须将要在这个字符串中格式化的所有值作为进一步的参数进行传递。例如，不要调用：

```
logging.debug('foo is %r' % foo)
```

不管需不需要，这行代码都将执行格式化操作；而是应该调用：

```
logging.debug('foo is %r', foo)
```

这行代码只有在需要的时候才会执行格式化（也就是，当且仅当调用 debug 将会产生日志输出时，这取决于当前的阈值级别）。

在默认情况下，这个阈值级别是 WARNING，表示任何函数 warning、error 或 critical 都将导致日志输出，但是 debug 和 info 函数不能导致日志输出。要想在任何时候更改阈值级别，可以调用 logging.getLogger( ).setLevel，将模块 logging: DEBUG、INFO、WARNING、ERROR 或 CRITICAL 提供的一个对应常数作为唯一参数传递给这个函数。例如，在调用下面这个函数后：

```
logging.getLogger().setLevel(logging.DEBUG)
```

从 debug 到 critical 的所有函数都将导致日志输出，直到再次更改这个阈值级别；如果以后再调用下面的函数：

```
logging.getLogger().setLevel(logging.ERROR)
```

然后，只有函数 error 和 critical 会导致日志输出（debug、info 和 warning 将不会导致日志输出）；这种情况将持续到再次更改这个阈值级别，依此类推。

在默认情况下，日志输出将会发送到用户程序的标准错误流（sys.stderr，参见第 8.3 节

中介绍的 stdin、stdout 和 stderr），并使用一种非常简单的格式（例如，在输出的每个行上都不包含时间戳）。开发者可以通过实例化一个适当的处理程序实例，并使用一个适当的格式程序实例来控制这些设置，并创建和设置一个新日志程序实例来保存该设置。在开发者想要一次性地全部设置这些日志参数的简单而常见的情况下，设置好的参数将在程序运行的整个过程中一直保持，最简单的方法是调用 logging.basicConfig 函数，这个函数可以让开发者通过命名参数非常简单地设置参数。只有第一次调用 logging.basicConfig 函数，并且只有在函数 debug、info 等之前调用该函数，才会有效果。因此，最合适的使用方法就是在程序的最开始调用 logging.basicConfig。例如，程序最开始经常使用的一个习惯用法如下：

```
import logging
logging.basicConfig(format='%(asctime)s %(levelname)8s %(message)s',
                    filename='/tmp/logfile.txt', filemode='w')
```

这个设置将把所有日志消息发送到一个文件，并使用精确的、适合人们阅读的时间戳，后面是在一个 8 个字符的字段中右对齐放置的严重级别，然后是消息体这种格式进行发送。

要想了解有关日志软件包，以及开发者可以使用该软件包执行的操作的所有疑问的海量详细信息，可以在 http://docs.python.org/lib/module-logging.html 上参考 Python 有关这个主题的丰富的在线信息。

assert 语句

assert 语句允许开发者将调试代码引入到程序中。assert 是一个使用以下语法的简单语句：

```
assert condition[,expression]
```

在运行 Python，并使用优化标记（-O，参见第 3.1 节）选项时，assert 是一个空操作：编译器不为 assert 语句生成代码。如果不使用优化标记，assert 将对 condition 进行计算。如果 condition 满足条件，则 assert 不做任何操作。如果 condition 不满足条件，assert 将使用 expression 作为参数（如果没有指定 expression，则不带参数）实例化 AssertionError 并引发结果实例。

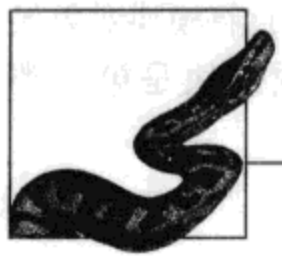
assert 语句是为程序提供文档化管理的有效方法。在开发者想要表述已知一个重要的条件 C 位于程序执行的某个特定位置时，assert C 要比一个只是用来表述 C 的注释更好一些。assert 语句的好处在于，在设定的条件实际上不被满足时，assert 将立即通过引发 AssertionError 异常发出警告。

__debug__内置变量

在运行 Python，不带选项-O 时，__debug__内置变量为 True。运行 Python，在带选项-O

时，__debug__为 False。另外，在使用选项-O 时，编译器将不为条件为__debug__的任何 if 语句生成代码。

要想利用这种优化功能，可以将开发者调用的函数的定义包含在使用 if __debug__:的 assert 语句中。在使用-O 选项运行 Python 时，这个技术可以让编译后的代码更小和更快，还可以通过显示这些函数的只是为了执行正确性检查来增强程序的清晰度。



第7章

模块

一个典型的 Python 程序是由几个源文件组成的。每个源文件对应于一个模块(module)，模块可以将程序代码和数据组合在一起以供重用。模块通常是相互独立的，因此其他程序可以重用所需的特定模块。一个模块可以使用 import 或 from 语句与另一个模块建立依赖关系。在其他一些编程语言中，全局变量可以提供一种隐藏的渠道以在模块之间建立连接。但是，在 Python 中，全局变量并不是对所有模块都是全局的，而只是单个模块对象的属性。因此，Python 的模块之间是以显式和可管理的方式进行通信的。

Python 还支持扩展（extension），也就是使用其他语言编写的组件，比如 C、C++、Java 或 C#，这些组件可以在 Python 中使用。使用扩展的 Python 代码（被称为客户代码）可以将扩展看作是模块。从客户代码的角度看，一个模块是 100%的纯 Python 代码还是扩展代码并不重要。开发者可以总是使用 Python 开始编写一个模块。然后，如果需要更好的性能，可以使用低层语言重新编写某些模块，这样不必更改使用这些模块的客户代码。第 25 章和第 26 章介绍了如何使用 C 和 Java 编写扩展。

本章介绍了模块的创建和加载，还介绍了如何将模块组合到包（package）中，包是以树形层次结构包含其他模块的模块。最后，本章还介绍了如何使用 Python 的发布工具（distutils）准备包的发布，以及如何安装发布的包。

7.1 模块对象

模块是一个 Python 对象，包含一些任意命名的属性以供开发者绑定和引用。名为 aname 的模块的 Python 代码通常被保存在名为 aname.py 的文件中，参见第 7.2 节。

在 Python 中，模块都是对象（值），可以像处理其他对象那样处理模块。这样，开发者可以将一个模块作为参数传递到函数调用中。同样地，函数可以返回一个模块作为函

数调用的结果。就像任何其他对象一样，模块可以被绑定到一个变量、容器中的一个项目，或者对象的一个属性上。例如，sys.modules 字典（参见第 7.2 节）将模块对象保存为其值。模块是 Python 中的普通对象，这个事实通常是通过说模块是第一类（first-class）对象来表达的。

import 语句

开发者可以将任何 Python 源文件用作一个模块，这是通过在其他一些 Python 源文件中执行一个 import 语句来实现的。import 具有以下语法：

```
import modname [as varname][,...]
```

import 关键字后面带有一个或多个使用逗号分隔的模块指定符（identifier）。在最简单，也最常见的情况下，模块指定符就是 modname，一个标识符——在 import 语句结束时，Python 用来绑定模块对象的一个变量。在这种情况下，Python 将查找具有相同名称的模块，以满足 import 的请求。例如：

```
import MyModule
```

这行代码将查找名为 MyModule 的模块，并将当前范围内名为 MyModule 的变量绑定到这个模块对象。modname 还可以是一个由句点（.）分隔的标识符组成的序列，用来命名包中的一个模块，参见第 7.3 节。

当模块指定符包含 as varname 时，Python 将查找一个名为 modname 的模块，然后将该模块对象绑定为变量 varname。例如：

```
import MyModule as Alias
```

这行代码将查找名为 MyModule 的模块，并将该模块对象绑定到当前范围中的变量 Alias 上。varname 就是一个简单的标识符。

模块体

模块体就是模块的源文件中的语句序列。指示一个源文件就是一个模块并不需要特殊的语法；任何合法的 Python 源文件都可以被用作一个模块。当运行的程序第一次导入一个模块时，将立即执行该模块的模块体。在执行模块体期间，模块对象已经存在，而且 sys.modules 中的一个条目已经被绑定到该模块对象上，并在模块体执行时逐步进行组装。

模块对象的属性

import 语句可以创建一个新的命名空间，其中包含一个模块的所有属性。要想访问这个命名空间中的一个属性，可以使用这个模块的名称作为前缀：

```
import MyModule
a = MyModule.f()
```

或

```
import MyModule as Alias
a = Alias.f()
```

模块对象的属性通常是使用该模块体中的语句来绑定的。当模块体中的一条语句绑定一个变量（全局变量）时，其实绑定的是该模块对象的一个属性。模块体的一般用途实际上就是创建模块的属性：def 语句可以用来创建和绑定函数，class 语句可以用来创建和绑定类，而赋值语句可以绑定任何类型的属性。

开发者还可以绑定和解除绑定模块体之外（也就是，在其他模块中）的模块属性，通常使用属性引用语法 M.name（其中 M 是任意值为该模块的表达式，而标识符 name 就是属性名称）。但是，为了让程序更清楚，通常最好将程序限定为只绑定该模块体内本身的模块属性。

只要 import 语句创建了模块对象，在模块体执行之前，该语句将隐式设置某些模块属性。__dict__属性是一个字典对象，该模块将把这个字典对象作为其属性的命名空间。与这个模块的所有其他属性不同，__dict__不可以作为该模块中的一个全局变量进行编程。这个模块中的所有其他属性都是该模块的__dict__中的条目，并且可以作为该模块中的一个全局变量进行编程。属性__name__是这个模块的名称，属性__file__则是用来加载该模块的文件的名称。

对于任意模块对象 M、任意对象 x，和任意标识符字符串 S（除__dict__）之外，绑定 M.S=x 等同于绑定 M.__dict__['S']=x。属性引用，比如 M.S，也完全等同于 M.__dict__['S']。唯一的区别在于，当 'S' 不是 M.__dict__中的一个键时，访问 M.__dict__['S'] 将引发 KeyError，而访问 M.S 将引发 AttributeError。模块属性也可以作为全局变量被模块体内的所有代码使用。换句话讲，在模块体内，作为全局变量使用的 S 等同于 M.S（也就是，M.__dict__['S']），可以用于绑定和引用。

Python 内置对象

Python 提供了几种内置对象（参见第 8 章）。所有内置对象都是名为__builtin__的预加载模块的属性。在 Python 加载一个模块时，该模块将自动获得一个名为__builtins__的额外属性，这个属性将引用模块__builtin__或者__builtin__的字典。Python 可以任选其一，不依赖于__builtins__。如果开发者需要直接访问模块__builtin__（极少有这种需要），可以使用一个 import__builtin__语句。请注意属性的名称与模块的名称之间的区别：属性名称多一个字母 s。在当前模块中没有找到全局变量时，Python 将在引发 NameError 之前查找当前模块的__builtins__中的标识符。

查找是 Python 用来使用代码访问内置对象的唯一机制。这个内置的名称不是保留的，在 Python 中也不是固定的。由于这种访问机制是简单和有文档说明的，开发者自己的代码也可以直接使用这个机制（但是要适度地使用这种机制，否则程序的清晰度和简

单性将受到损坏)。这样，开发者可以添加自己的内置函数，或者将自己的函数替代为普通内置函数。下面的示例显示了如何使用开发者自己的函数包装一个内置函数（参见第 7.2 节了解有关__import__的信息，参见第 7.2 节了解有关 reload 的信息）：

```
# reload 可以接受模块对象；下面让 reload 还可以接受字符串
import __builtin__
_reload = __builtin__.reload                          # 保存初始 built-in
def reload(mod_or_name):
    if isinstance(mod_or_name, str):                  # 如果参数是字符串
        mod_or_name = __import__(mod_or_name)         # 替换该模块
    return _reload(mod_or_name)                       # 调用真正的 built-in
__builtin__.reload = reload                           # 覆盖 built-in w/包装程序
```

模块文档字符串

如果模块体中的第一个语句是一个字符串字面常量，编译器将把这个字符串绑定为该模块的文档字符串属性，名为__doc__。文档字符串也被称为 docstrings，参见第 4.11 节中对文档字符串的介绍。

模块私有变量

模块中没有变量是真正私有的。但是，按惯例，以单个下划线（_）开始的标识符，比如_secret，表示这个标识符是私有的。换句话讲，这个开始的下划线可以告知客户代码程序员，他们不能直接访问这个标识符。

开发环境和其他工具依赖于起始下划线命名惯例来识别一个模块的哪些属性是公有的（也就是，部分模块的接口）和哪些属性是私有的（也就是，只能在该模块内使用）。将私有变量以下划线开始是区别私有和公有属性的一个很好的编程经验，这样可以更清晰，并从工具获得最大的好处。

在使用其他人编写的模块编写客户代码时，遵循这个惯例是特别重要的。换句话讲，就是在名称以下划线开始的这些模块中避免使用任何属性。毫无疑问，这些模块的将来版本将维护其公有接口，但是非常有可能更改私有实现的细节，并且私有属性可以确切地表明这些实现细节。

from 语句

Python 的 from 语句可以帮助开发者将特定属性从一个模块导入到当前的命名空间。from 具有两个语法变体：

```
from modname import attrname [as varname][,...]
from modname import *
```

from 语句指定了一个模块名，后面带有一个或多个使用逗号分隔的属性指定符。在最简单、也最常见的情况下，属性指定符只是一个标识符 attrname，这个标识符是 Python

绑定到名为 modname 的模块中相同名称的属性的一个变量。例如：

```
from MyModule import f
```

modname 还可以是一个由句点（.）分隔的标识符组成的序列，用来命名包中的一个模块，参见第 7.3 节“包”。

在属性指定符中包含 as varname 时，Python 将从模块获得属性 attrname 的值，然后将其绑定到变量 varname 上。例如：

```
from MyModule import f as foo
```

attrname 和 varname 总是非常简单的标识符。

在 Python 2.4 中，开发者可以选择将所有在 from 语句中带有关键字 import 的属性指定符包含在圆括号中。在开发者需要使用许多属性指定符时，有时候是非常有用的，这样做可以将 from 语句的单个逻辑行分割成多个逻辑行，这种方式要比使用反斜线（\）更优美。

```
from some_module_with_a_long_name import (another_name, and_another,
                                          one_more, and_yet_another)
```

from...import * 语句

直接位于模块体内（而不是在一个函数体或类体内）的代码可以在 from 语句中使用星号（*）。

```
from MyModule import *
```

* 要求模块 modname 的所有属性被绑定为导入模块中的全局变量。在模块 modname 具有一个名为__all__的属性时，该属性的值就是被这种类型的 from 语句绑定的属性列表。否则，这种类型的 from 语句将绑定模块 modname 中除那些以下划线开始的属性之外的所有属性。由于 from M import * 可以绑定任意集合的全局变量，因此可能经常产生无法预料和不想发生的负面影响，比如隐藏内置对象和重新绑定仍然需要使用的变量。总的来说，使用 from 语句的星号（*）形式要非常小心，而且这种形式只能用于有文档说明可以显式支持这种用法的模块。最可能的情况是，如果开发者永远都不使用这种形式，这样的程序可能会更好一些。

from 对比 import

一般来说，import 语句通常是比 from 语句更好的一个选择。只有在交互式 Python 会话中偶尔使用时，才将 from 语句，尤其是 from M import * ，看作是比较方便的方法。如果总是使用语句 import M 访问模块 M，并且总是使用显式语法 M.A 访问 M 的属性，代码可能会稍微不那么简洁，但是要清楚得多，并且更具可读性。from 的一个很好的用法就是从包导入特定模块，第 7.3 节将对此进行介绍。但是在绝大部分情况下，import 要比 from 更好。

7.2 模块加载

模块加载操作依赖于内置 sys 模块的属性（参见第 8.3 节）。本节介绍的模块加载操作是由内置函数_ _import_ _实现的。开发者可以在代码中使用模块名字符串作为一个参数直接调用_ _import_ _。_ _import_ _将返回这个模块对象，如果导入失败，则引发 ImportError。

为了导入一个名为 M 的模块，_ _import_ _将首先检查字典 sys.modules，并使用字符串 M 作为键。在键 M 存在于这个字典中时，_ _import_ _将返回对应值作为请求的模块对象。否则，_ _import_ _将使用 M 的一个_ _name_ _把 sys.modules[M]绑定到一个新的空白模块对象上，然后查找正确的方法来实例化（加载）该模块，参见第 7.2 节。

感谢这种机制，这种相对比较慢的加载操作只有在程序的给定运行中模块第一次被导入时才会发生。当一个模块被再次导入时，该模块将不被重新加载，因为_ _import_ _将快速查找并返回该模块在 sys.modules 中的条目。这样，在给定模块第一次被导入之后的所有导入操作都是非常快的：这些操作只是字典查找（要想强制重新加载操作，参见第 7.2 节）。

内置模块

在模块被加载后，_ _import_ _将首先检查该模块是否是内置模块。元组 sys.builtin_module_names 中列出了内置模块，但是重新绑定这个元组不会影响模块的加载。在 Python 加载一个内置模块时，并且在 Python 加载任何其他扩展时，Python 将调用该模块的初始化函数。搜索内置模块还会在平台的特定位置查找模块，比如 Mac 上的资源和框架，以及 Windows 的注册表。

在文件系统中搜索模块

如果模块 M 不是内置的，_ _import_ _将查找 M 的代码，就像查找文件系统中的文件。_ _import_ _将按顺序查看字符串，这些字符串是列表 sys.path 中的项目。每个项目都是一个目录的路径，或者是一个普通 ZIP 格式压缩文件的路径。sys.path 是在程序启动的时候使用环境变量 PYTHONPATH 初始化的（参见第 3.1 节）。sys.path 中的第一个项目永远是主程序（脚本）被加载的目录。sys.path 中的空白字符串表示当前字符串。

代码可以变异或重新绑定 sys.path，而这样的更改将影响_ _import_ _搜索哪些目录和 ZIP 压缩文件来加载模块。更改 sys.path 不会影响在更改 sys.path 时已经加载的模块（并且这些已经记录在 sys.modules 中了）。

如果系统启动时在 PYTHONHOME 目录中找到了扩展名为.pth 的文本文件，这个文件的内容将被添加到 sys.path 中，每行一个项目。.pth 文件可以包含空白行和以字符# 开

始的注释行；Python 将忽略所有这些行。.pth 文件还可以包含 import 语句，但是不包含其他类型的语句，Python 将在开发者的程序开始执行之前执行这些 import 语句。

在 sys.path 中的每个目录和 ZIP 压缩文件中的文件中查找模块 M 时，Python 将按下面列出的顺序查找以下扩展名。

1. .pyd 和.dll（Windows）或.so（大多数类 UNIX 平台），这表示 Python 扩展模块（其他一些 UNIX 使用了不同的扩展名；例如，.sl 是 HP-UX 上使用的扩展名）。在大多数平台上，不能从一个 ZIP 压缩文件加载扩展——只能从纯的源代码或编译的字节代码 Python 模块加载扩展。
2. .py，这表示纯的 Python 源模块。
3. .pyc（或者.pyo，如果 Python 运行时使用了选项-O），这表示编译的字节代码 Python 模块。

在文件中查找模块 M 时，Python 查找文件的最后一个路径是 M/__init__.py，这表示名为 M 的目录中的一个名为__init__.py 的文件，参见第 7.3 节。

在查找源文件 M.py 时，Python 将把这个文件编译成 M.pyc（或者 M.pyo），除非字节代码文件已经存在、要比 M.py 更新，并且是使用 Python 的相同版本编译的。如果 M.py 是从可写目录编译的，Python 将把该字节代码文件保存到相同目录的文件系统中，这样，在将来运行时就不再需要重新编译了。在字节代码文件要比源文件更新时（这是根据字节代码文件中的内部时间戳确定的，而不是根据文件系统中记录的日期），Python 将不重新编译这个模块。

在 Python 有了字节代码文件后，不管是通过编辑来构造的还是从文件系统读取的，Python 将执行这个模块体以初始化该模块对象。如果这个模块是一个扩展，Python 将调用该模块的初始化函数。

主程序

Python 应用程序的执行通常是从最顶层的脚本开始的（也被称为主程序（main program）），参见第 3.1 节。主程序的执行就像加载的任何其他模块一样，区别只是 Python 将在内存中保存该字节代码，而不是将其保存到硬盘中。主函数的模块名永远是__main__，同时作为__name__全局变量（模块属性）和 sys.modules 中的键。开发者通常不能导入被用作主程序的相同.py 文件。如果这么做了，这个模块将被再次加载，并且该模块体将在一个单独的模块对象中使用不同的__name__从顶部开始再次被执行。

Python 模块中的代码可以通过检查全局变量__name__是否等于‘__main__’来测试这个模块是否被用作主程序。这个习惯用法就是：

```
if __name__=='__main__':
```

这行代码通常用于守卫某些代码，因此，这行代码只有在模块作为主程序运行时才会

执行。如果一个模块被设计为只能被导入，则在该模块作为主程序运行时，通常必须执行单元测试，参见第 18.1 节。

reload 函数

在程序运行时，Python 只在第一次导入模块时加载该模块。在进行交互式开发时，开发者需要确保模块在每次编辑时都被重新加载（有些开发环境提供了自动加载功能）。

要想重新加载一个模块，可以将该模块对象（而不是模块名）作为内置函数 reload 的唯一参数。reload(M)可以确保 M 的重新加载版本可以被依赖于 import M，并使用语法 M.A 访问属性的客户代码使用。但是，reload(M)不会对绑定到 M 的属性的以前值的其他现有引用产生任何影响（例如，使用 from 语句）。换句话讲，已经被绑定的变量仍然是被绑定的，不会受到 reload 的影响。reload 不能重新绑定这些变量是避免使用 from，而更支持使用 import 的又一个明证。

reload 并不是递归的：在开发者加载一个模块 M 时，这并不意味着由 M 导入的其他模块将按顺序被重新加载。开发者必须通过显式调用 reload 函数进行特别安排以重新加载修改的所有模块。

循环导入

Python 可以指定循环导入。例如，开发者可以编写一个包含 import b 的模块 a.py，而模块 b.py 包含 import a。实际上，避免循环导入通常要更好一些，因为循环依赖关系总是容易出错和难以管理的。如果开发者因为某些原因决定使用循环导入，为了避免代码中的错误，必须理解循环导入是如何工作的。

假定主脚本执行 import a。正如前面介绍的，这个 import 语句创建了一个新的空白模块对象 sys.modules['a']，然后 a 的模块体将开始执行。在 a 执行 import b 时，将会创建一个新的空白对象 sys.modules['b']，然后 b 的模块体将开始执行。a 的模块体的执行将被挂起，直到 b 的模块体结束。

现在，在 b 执行 import a 时，import 语句将查找已经定义的 sys.modules['a']，并因此将模块 b 中的全局变量 a 绑定到模块 a 的模块对象上。因为 a 的模块体的执行当前是挂起的，模块 a 在这个时候只是部分运行。如果 b 的模块体中的代码立即尝试访问还没有被绑定的模块 a 的一些属性，将会产生一个错误。

如果开发者在某些情况下坚持要使用循环导入，则必须非常小心地管理每个模块定义其自身的全局变量、导入其他模块和访问其他模块的全局变量的顺序。开发者可以对这些操作发生的顺序进行更多地控制，开发者可以将语句组合到函数中，并按控制的顺序调用这些函数，而不是只依赖于模块体中顶层语句的连续执行。但是，删除循环依赖关系几乎总是要比在保持这样的循环依赖关系的同时确保安全的执行顺序更简单。这是因为循环依赖关系对于其他原因而言也是很坏的，本书建议尽量删除这些依

赖关系。

sys.modules 条目

内置__import__函数绝不会绑定除模块对象之外的任何对象作为 sys.modules 中的值。但是，如果__import__找到了一个已经在 sys.modules 中的条目，该函数将返回这个值，不管对象的类型是什么。import 和 from 语句依赖于__import__函数，因此这两条语句也可以使用不是模块的对象来结束。这种不进行类型检查的特性是在几个 Python 版本之前被引入的高级功能（非常老的 Python 版本通常才会进行类型检查，只允许模块对象作为 sys.modules 中的值）。为了使用像__getattr__和__setattr__特殊方法这样的功能，不进行类型检查的功能可以让开发者将类实例设置为 sys.modules 中的条目。这种高级技术可以帮助开发者导入类似于模块的对象，并且该对象的属性可以临时进行计算。下面是一个简单示例：

```
class TT(object):
    def __getattr__(self, name): return 23
import sys
sys.modules[__name__] = TT()
```

在将这段代码作为一个模块导入时，将会获得一个类似于模块的对象（实际上是类 TT 的一个实例），看起来这个对象中包含开发者尝试从该对象获得的任何属性名称；所有的属性名称对应于整数值 23。

自定义导入器

Python 提供的一个高级，但是极少需要的功能就是可以更改语义，或者某些或全部 import 和 from 语句。

重新绑定__import__

开发者可以将模块__builtin__的__import__属性重新绑定到自定义的导入器函数——例如，一个使用通用内置包装技术构建的函数，参见第 7.1 节中对 Python 内置对象的介绍。这样的重新绑定操作将影响重新绑定之后执行的所有 import 和 from 语句，因此可能会出现不可预料的全局影响。通过重新绑定__import__构建的自定义导入器必须实现与内置__import__相同的接口，特别是，该导入器负责支持 sys.modules 的正确使用。尽管重新绑定__import__最初看起来可能像一个非常吸引人的方案，但是在大多数必须使用自定义导入器的情况下，可以改为通过导入钩子（import hooks）实现导入器。

导入钩子

Python 为有选择地更改导入的细节提供了强大的支持，包括标准库模块 imp 和 ihooks。自定义导入器是一种高级，但是极少使用的技术，然而有些应用程序出于各种各样的

目的，可能需要使用这种技术，这些目的包括从不同于 ZIP 文件格式的压缩文件、从数据库和从网络服务等位置导入代码。最适合于这种非常高级需要的方案就是将导入器工厂（importer factory）可调用对象记录为属性 meta_path 和/或 sys 模块的 path_hooks 中的项目，http://www.python.org/peps/pep-0302.html 上给出了详细介绍。正如前面提到的，为了允许从 ZIP 文件无缝导入模块，这个方案也是 Python 用来钩住标准库模块 zipimport 的一种方法。对 PEP 302 的详细信息进行完整的学习对于 sys.path_hooks 的任何真正使用和用户都是不可缺少的，但是下面有一个简单的示例，在开发者需要的时候，可以帮助开发者理解这种方法提供的可能性。

假定在开发某些程序的第一个框架时，开发者想要可以为还没有编写的模块使用 import 语句，并且结果将只得到警告消息（和空白模块）。开发者可以通过编写一个下面这样的自定义导入器模块很容易地实现这个功能（将与包连接的复杂功能放在一边，只处理简单的模块）：

```
import sys, new
class ImporterAndLoader(object):
     '''importer and loader functionality is usually in a single class'''
     fake_path = '!dummy!'
     def __init__(self, path):
         '''we only handle our own fake-path marker'''
         if path != self.fake_path: raise ImportError
     def find_module(self, fullname):
         '''we don't even try to handle any qualified module-name'''
         if '.' in fullname: return None
         return self
     def load_module(self, fullname):
         # 打印某种类型的警告消息
         print 'NOTE: module %r not written yet' % fullname
         # 创建新的空白模块，也将其放在 sys.modules 中
         mod = sys.modules[fullname] = new.module(fullname)
         # 最低限度地组装新模块，然后返回该模块
         mod.__file__ = 'dummy<%s>' % fullname
         mod.__loader__ = self
         return mod
# 将该类添加到钩子，并将其假路径标记添加到路径中
sys.path_hooks.append(ImporterAndLoader)
sys.path.append(ImporterAndLoader.fake_path)

if __name__ == '__main__':          # 在作为主脚本运行时进行自测试
    import missing_module           # 导入一个简单的 missing 模块
    print sys.modules.get('missing_module')  #...应该会成功
    # 检查该程序不处理从包导入
    try: import missing_module.sub_module
    except ImportError:
    else: print 'Unexpected:',sys.modules.get('missing_module.sub_module')
```

7.3 包

包（package）是一个包含其他模块的模块。包中的某些或所有模块也有可能是子包，这样会产生一个树形层次结构。名为 P 的包被保存在 sys.path 中的某些目录的一个子目录中，这个子目录也叫做 P。包也可以存在于 ZIP 文件中；在后续章节，本书将介绍包存在于文件系统中的情况，因为包存在于一个 ZIP 文件中的情况是非常类似的，这依赖于 ZIP 文件中类似于文件系统层次结构。

P 的模块体在文件 P/__init__.py 中。开发者必须有一个名为 P/__init__.py 的文件，即使该文件是空白的（表示一个空白模块体），这是为了向 Python 指示目录 P 确实是一个包。包的模块体是在第一次导入该包（或者这个包的任何模块）的时候被加载的，这些操作从所有方面来看都像任何其他 Python 模块。目录 P 中的其他.py 文件都是包 P 的模块。包含__init__.py 文件的 P 的子目录都是 P 的子包（在 Python 2.5 中，P 的所有子目录都是 P 的子包，不管这些子目录是否包含__init__.py 文件）。嵌套可以继续到任何深度。

开发者可以在包 P 中导入一个名为 M 的模块，作为 P.M。更多的句点（.）表示可以导航到包层次结构中（包的模块体总是在包中的任何模块被加载之前被加载的）。如果开发者使用语法 import P.M，变量 P 将被绑定到包 P 的模块对象上，而对象 P 的属性 M 将被绑定到模块 P.M 上。如果使用语法 import P.M as V，变量 V 将被直接绑定到模块 P.M 上。

使用 from P import M 从包 P 导入特定模块 M 是一个完全可以接受的实现方案：在这种情况下，from 语句是特别好的。

包 P 中的模块 M 可以使用语句 import X 导入 P 的任何其他模块 X。Python 将在搜索 sys.path 中的项目之前搜索该模块自己的包目录。但是，这只能应用于兄弟模块，而不是祖先或其他更复杂的关系。在包 P 中的模块之间共享对象（比如，函数或常数）的最简单，也最清楚的方法就是将共享模块分组到名为 P/Common.py 的文件中。然后，就可以从需要访问这些对象的包的所有模块导入了（import Common），然后将这些对象称为 Common.f、Common.K 等。

包对象的特殊属性

包 P 的__file__属性是一个字符串，指示 P 的模块体的路径——也就是，文件 P/__init__.py 的路径。

包 P 的模块体——也就是，文件 P/__init__.py 中的 Python 源代码——可以有选择地设置一个名为__all__的全局变量（就像任何其他模块可以做的那样），用来控制在其他一些模块执行了语句 from P import * 时发生的操作。特别是，如果没有设置__all__，

from P import * 不会导入 P 的模块，但是只导入 P 的模块体中设置的其他名称。但是，在任何情况下，使用 from P import * 都不是一个值得推荐的用法。

包 P 的__path__属性是一个由目录的路径字符串组成的列表，P 的模块和子包都是从这些目录被加载的。最开始，Python 将把__path__设置为一个包含单个元素的列表：包含文件__init__.py 的目录的路径，这个文件是这个包的模块体。但是，开发者的代码可以修改这个列表以影响将来对这个包的模块和子包的搜索。这个高级技术是极少需要的，但是有时候也可以在开发者想要在几个分开的目录中放置一个包的模块时使用。

绝对导入对比相对导入

在 Python 2.5 中，如果在模块的开始使用以下代码：

```
from __future__ import absolute_import
```

然后规则将更改为：import X 表示从 sys.path 中的某个位置正常地（“绝对”）导入一个模块；相反，要想得到一个相对导入代码 from . import X，这意味着从当前包中导入一个模块。使用这种功能，开发者还可以编写更丰富和更复杂的相对导入，尽管太有想象力的相对导入可能会很容易损害代码的清晰度。在 Python 的将来版本中，这些规则将成为默认规则；参见 http://www.python.org/doc/peps/pep-0328/ 以了解详细信息，其中包括当前对于这些更改的进度的最佳预测。

7.4 发布工具（distutils）

Python 模块、扩展和应用程序可以按以下几种形式进行打包和发布的。

压缩文件

通常是 Windows 中的.zip 文件和类 UNIX 系统的.tar.gz（又被称作.tgz）文件，但是这两种形式是相互可移植的。

自动解包或自动安装可执行文件

通常是 Windows 中的.exe 文件。

自包含的，不要求安装的预备运行可执行程序

例如，Windows 中的.exe 文件、UNIX 上带有一个小的脚本前缀的 ZIP 压缩文件、Mac 的.app 文件等。

平台相关安装程序

例如，Windows 上的.msi 文件、大多数 Linux 版本上的.rpm 和.srpm 文件，以及 Debian GNU/Linux 和 Ubuntu 上的.deb 文件。

Python 蛋

一个非常流行的第三方扩展，参见第 7.4 节。

在开发者将一个包发布为自动安装可执行文件，或者平台相关压缩文件，用户只需要运行安装程序就可以安装这个包了。如何安装这样一个安装程序取决于平台，但是这个程序是使用哪种语言编写的并不是什么问题。参见第 27 章以了解如何在各种平台上编译自包含和预备运行的可执行文件。

在开发者将包发布为一个压缩文件，或者发布为一个可以解包但是不能安装其自身的可执行文件时，发布文件的使用与这个包是否使用 Python 编写有关。在这种情况下，用户必须首先将该压缩文件解包到某个适当的目录，也就是 Windows 计算机上的 C:\Temp\MyPack 目录或类 UNIX 计算机上的~/MyPack 目录。在解压缩的文件中应该有一个脚本，通常被命名为 setup.py，这里使用了被称为“发布工具”（标准库包 distutils）Python 工具。然后，这个发布的包安装起来几乎与自动安装可执行程序一样简单。用户可以打开一个命令行提示符窗口，然后将目录更改到该压缩文件被解包的目录中。然后用户可以运行下面的命令：

```
C:\Temp\MyPack> python setup.py install
```

根据这个包的作者在 setup.py 安装脚本中指定的选项，setup.py 脚本将使用 install 命令，将这个包安装为用户的 Python 安装程序的一部分（当然，该用户需要拥有适当的权限，可以写入 Python 安装程序的目录，因此可能需要使用像 su 或 sudo 这样的权限提升命令）。在默认情况下，distutils 提供了用户在运行 setup.py 时的跟踪信息。在 install 命令的右边放置选项--quiet，可以隐藏大多数详细信息（如果存在错误，用户仍将看到这些错误消息）。下面的命令可以给出 distutils 的详细帮助信息：

```
C:\> python setup.py --help
```

在安装使用 distutils 准备好的包时，如果开发者想要，还可以对 distutils 如何执行安装程序进行详细地控制。开发者可以使用扩展名.cfg（也被称为配置文件）在文本文件中记录安装选项，因此 distutils 将默认应用开发者喜欢的安装选项。开发者可以在系统范围的基础上为单个用户，甚至为单个包安装完成这样的自定义。例如，如果开发者想要具有最小输出的安装程序成为系统级的默认设置，可以创建下面这个名为 pydistutils.cfg 的文本文件：

```
[global]
quiet=1
```

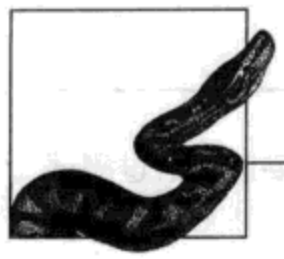
将这个文件放在保存 distutils 包的相同目录中。例如，在典型的 Windows 版本的 Python 2.4 安装上，这个文件是 C:\Python24\Lib\distutils\pydistutils.cfg。第 27.1 节提供了有关使用 distutils 准备模块、包、扩展和应用程序以进行发布的更多信息。

Python 蛋

一个新出现的用来发布 Python 包的标准是 Python 蛋（Python Eggs），也就是可以有选

择地包含结构化元数据和 Python 代码，并且文件扩展名为.egg 的 ZIP 文件。Eggs 的许多令人钟爱的特性之一就是，非常简单的 Eggs 实际上不需要真正的“安装”过程：只需要将.egg 文件放在开发者的 Python sys.path 中的任何位置，这样，Python 代码就可以立即开始使用这个包了。尽管使用很简单，但实际上 Eggs 还可以包含更丰富的元数据，从而提供许多其他令人激动的可能的功能。

不幸地是，Eggs 还没有被包含在 Python 2.5 的发布版本中，但这只是时间的原因（在决定将哪些新功能放到 Python 2.5 中的时候，Eggs 仍处于开发阶段）。不过，开发者仍然可以从 http://peak.telecommunity.com/DevCenter/EasyInstall 下载并运行一个单独的 Python 脚本，这样就可以在 Python 2.3 以后的所有 Python 版本上使用 Eggs 了。本书强烈推荐使用 Eggs 作为一种优秀的方法发布开发者自己的 Python 包；除了已经提到的这些好处，Eggs 还附带了一个包 setuptools，作为 distutils 的补充，这个包中包含了可以用来设置和安装的强大功能，参见第 27.1 节。可以从 http://peak.telecommunity.com/DevCenter/PythonEggs 阅读所有有关 Eggs 和 setuptools 的说明。



第8章

核心内置

在 Python 中，术语“内置”（built-in）具有多重含义。在大多数情况下，Python 内置表示一个不需要使用 import 语句就可以直接被 Python 代码访问的对象。第 7.1 节中介绍 Python 内置对象时介绍了 Python 用来实现这种直接访问对象的机制。Python 中的内置类型包括数字、序列、字典、集合、函数（参见第 4 章）、类（参见第 5.1 节）、标准异常类（参见第 6.4 节）和模块（参见第 7.1 节）。内置文件（file）对象参见第 10.3 节，第 13.2 节介绍了 Python 的内部操作内在的一些内置类型。本章提供了其他一些核心内置类型（参见第 8.1 节），还介绍了模块__builtin__中可用的内置函数（参见第 8.2 节）。

正如本书在第 7.1 节中介绍 Python 内置对象时提到的，有些模块被称为“内置”是因为这些模块是 Python 标准库不可或缺的一个部分（尽管需要使用 import 语句来访问这些模块），这一点区别于可以分开的和可选的扩展模块，扩展模块也被称为 Python 扩展（extension）。本章详细介绍了一些核心内置模块，基本上是那些可以提供在其他一些语言中被编译到语言本身的功能的模块：也就是，本章第 8.3 节中介绍的 sys 模块、第 8.4 节中介绍的 copy 模块、第 8.5 节中介绍的 collections 模块、第 8.6 节中介绍的 functional 模块（只支持 Python 2.5）、第 8.7 节中介绍的 bisect 模块、第 8.8 节中介绍的 heapq 模块、第 8.9 节中介绍的 UserDict 模块、第 8.10 节中介绍的 optparse 模块和第 8.11 节中介绍的 itertools 模块。第 9 章将介绍一些与字符串相关的核心内置模块（参见第 9.2 节中介绍的 string 模块、第 9.6 节中介绍的 codecs 模块、第 9.6 节中介绍的 unicodedata 模块和第 9.7 节中介绍的 re 模块）。第 3 部分和第 4 部分将分别介绍 Python 标准库中的其他一些模块。

8.1 内置类型

本节将详细介绍 Python 的核心内置类型，比如 int、float、dict 和其他一些类型。有关这些类型的更详细介绍，以及对其实例的操作可以参见第 4 章。在本节中，“数字”都

表示“非复数数字”。

表 8-1

basestring	`basestring` 类型 str 和 unicode 的非可实例化（抽象）公共基类型。主要用于通过测试 isinstance(x, basestring) 确定某个对象 x 是否是一个字符串 （不管是纯文本还是 Unicode）。
bool	`bool(x)` 如果参数 x 的计算结果为假，则返回 False；如果参数 x 的计算结果为真，则返回 True（参见第 4.2 节）。bool 是 int 的一个子类，内置名称 False 和 True 表示类型 bool 的唯一两个实例。这两个实例也是整数，分别等于 0 和 1，但是 str(True)是‘True’，而 str(False)是‘False’。
buffer	`buffer(obj,offset=0,size=-1)` 返回一个只读缓冲区对象，表示 obj 数据的压缩切片，从给定的 offset 开始，大小（size）已经给定（如果 size<0，或者 obj 的数据太短，以至于无法提供 offset 之后的 size 字节，则会一直到达 obj 数据的末尾）。obj 必须是支持缓冲区调用接口的类型，比如字符串或者数组。
classmethod	`classmethod(function)` 返回一个类方法对象。实际上，只能在类体中调用这个内置类型，并且在 Python 2.4 中，最常见的是将其作为装饰器使用。参见第 5.1 节中介绍的“类方法”。
complex	`complex(real,imag=0)` 将任何数字或适当的字符串转换为复数。只有在 real 是一个数字，并且是结果复数的虚部时，imag 才可能出现。参见第 4.2 节中介绍的复数。

dict	`dict(x={ })`
	返回一个新字典对象，包含与参数 x 相同的项目（参见第 4.2 节）。在 x 是一个字典时，dict(x)将返回 x 的一个副本，就像 x.copy( )一样。二者选一，x 可以是一个项目为成对数据的可迭代对象（每两个项目进行迭代）。在这种情况下，dict(x)将返回一个字典，其中的键是 x 中每个数据对的第一个项目，而值则是对应的第二个项目。换句话讲，在 x 是一个序列时，c=dict(x)具有与下面的代码相同的效果：
	`c = {}`
	`for key, value in x: c[key] = value`
	开发者还可以使用除了位置参数 x 之外，或者替代位置参数 x 的命名参数调用 dict。每个命名参数都将成为结果目录中的附加项目，使用该参数的名称作为键，并有可能覆盖一个来自 x 的项目。

enumerate	`enumerate(iterable)`
	返回一个项目为数据对的新迭代器对象。对于每个这样的数据对，第二个项目就是 iterable 中的对应项目，而第一个项目是一个整数：0、1、2…。例如，下面这段代码将对整数列表 L 进行循环，将每个偶数除以 2：
	`for i,num in enumerate(L):`
	`  if num % 2 == 0:`
	`  L[i] = num // 2`

续表

file, open	file(*path*,*mode*='r',*bufsize*=-1) open(*filename*,*mode*='r',*bufsize*=-1) 打开或创建一个文件并返回一个新文件对象。参见第 10.3 节。
float	float(*x*) 将任何数字或适当的字符串转换为浮点型数字。参见第 4.2 节中介绍的浮点型数字。
frozenset	frozenset(*seq*=[ ]) 返回一个新的固定（不可变）集合对象，包含与可迭代对象 seq 相同的项目。在 seq 是一个固定集合时，frozenset(seq)将返回 seq 自身，就像 seq.copy( )所做的一样。参见第 4.7 节。
int	int(*x*[,*radix*]) 将任何数字或适当的字符串转换为 int 型。当 x 是一个数字时，int 将截断到 0，丢弃所有的小数部分。只有在 x 是字符串时，需要给出进制数（radix）。radix 是数值转换的基础（进制数），在 2～36 时，默认值为 10。radix 可以被显式传递为 0: ，则进制数为 8、10 或 16，取决于字符串 x 的形式，就像整数字面常量一样，参见第 4.2 节中介绍的整数。
list	list(*seq*=[ ]) 返回一个新列表对象，具有与迭代对象 seq 相同的项目，并按相同的顺序。在 seq 是一个列表时，list(seq)将返回 seq 的一个副本，就像 seq [:] 所做的那样。参见第 4.2 节中介绍的列表。
long	long(*x*[,*radix*]) 将任何数字或适当的字符串转换为长整型（long）。关于 radix 参数的规则与 int 的规则是完全相同的。参见第 4.2 节中介绍的整数。
object	object() 返回最基本类型的一个新实例。类型 object 的这些直接实例没有任何功能，因此创建这样的实例的唯一实际原因就是要获得一个“哨兵”对象，也就是，一个可以保证不等于任何其他对象的对象。
property	property(*fget*=None,*fset*=None,*fdel*=None,*doc*=None) 返回一个属性指定符。实际上，开发者只能在类体中调用这个内置类型。参见第 5.1 节。
reversed	reversed(*seq*) 返回一个新迭代器对象，其中的项目是序列 seq（必须是一个固定序列，而不是任意一个可迭代序列）中以相反顺序排列的项目。
set	set(*seq*=[ ]) 返回一个新的可变集合对象，其中包含与可迭代对象 seq 相同的项目。在 seq 是一个集合时，set(seq)将返回 seq 的一个副本，就像 seq.copy( )所做的那样。参见第 4.2 节。
slice	slice([*start*,]*stop*[,*step*]) 返回一个包含只读属性 start、stop 和 step 的切片对象，这些属性绑定到各

续表

slice	自的参数值上，在不带这些参数时，每个值默认为 None。这样的切片实质上表示与 range(start,stop,step)相同的索引集合。切片语法 obj[start:stop:step]将把一个适当的切片对象传递为对象 obj 的方法_ _getitem_ _、_ _setitem_ _或_ _delitem_ _的参数。由 obj 负责解释这些方法接收到的切片对象。参见第 5.2 节中介绍的容器切片。
staticmethod	`staticmethod(`*function*`)` 返回一个静态方法对象。实际上，开发者只能在类体中调用这个内置类型，在 Python 2.4 中，最常见的是将其用作一个装饰器。参见第 5.1 节中介绍的静态方法。
str	`str(`*obj*`)` 返回 obj 对象的一个简洁和可读的字符串表示。如果 obj 是一个字符串，str 将返回 obj。参见第 8.2 节和第 5.2 节。
super	`super(`*cls*`,`*obj*`)` 返回适合于调用超类方法的对象 obj（该对象必须是类 cls 的实例或 cls 的子类）的一个超类。实际上，开发者只能在一个方法的代码中调用这个内置类型。参见 5.1 节中介绍的合作超类方法调用。
tuple	`tuple(`*seq*`)` 返回一个与可迭代对象 seq 具有相同的项目，并按相同顺序排列的元组。在 seq 是一个元组时，tuple 将返回 seq 本身，就像 seq[:]所做的那样。参见第 4.6 节。
type	`type(`*obj*`)` 返回类型为 obj 的类型对象（也就是，最底层的派生类（又名 leafmost）的类型对象，obj 是该类型对象的一个实例）。所有传统实例对象具有相同的类型（InstanceType），即使这些对象是不同类的实例：使用 isinstance 可以检查一个实例是否属于一个特殊类。但是，在新型对象模型中，type(x)也就是任何 x 的 x._ _class_ _。 检查 type(x)与其他一些类型对象是否相等或一致也被称为类型检查(type-checking)。类型检查甚至不适合在作为产品的 Python 代码中，因为接口具有多态性。Python 中的一个普通习惯用法就是尝试使用 x，就像 x 是开发者期望的类型一样，并使用 try/except 语句处理任何问题，正如第 6.6 节中介绍的那样。在只需要进行类型检查时，可以使用 isinstance 作为替代，这通常出于调试的目的。isinstance(x,atype)要比 type(x)更好一些的就是 atype，因为至少 isinstance 还可以接受作为 atype 的任何子类的一个实例的 x，而不只是 atype 本身的一个直接实例。
unicode	`unicode(`*string*`[,`*codec*`[,`*errors*`]])` 返回由解码 string 获得的 Unicode 字符串对象。codec 是要使用的编码方式的名称。如果不提供 codec 参数，unicode 将使用默认编码方式(通常是‘ascii’)。errors 是一个指定如何处理解码错误的字符串。参见第 9.6 节，尤其是有关编码方式和 errors 的信息，以及第 5.2 节中的_ _unicode_ _。

续表

xrange	xrange([*start*,]*stop*[,*step*=1]) 返回一个只读序列对象，其中的项目是等差数列中的整数。这些参数与 range 的参数相同。xrange 将返回一个特殊类型的序列对象，在大多数情况下用于 for 语句中（可以索引这个对象，但是不能对其进行切片），而 range 将创建并返回一个普通列表对象。对于这种特殊和经常的用法，xrange 要比 range 消耗更少的内存，要做的只是对一个等差数列进行循环操作。但是，对于循环操作本身，xrange 的开销要比 range 稍微大一些。总的来说，xrange 和 range 之间的性能差别通常是很小的，不值得担心这种差别。

8.2 内置函数

本节将按字母顺序详细列举模块__builtin__中可用的 Python 函数。请注意，这些内置函数的名称都不是保留字。因此，开发者的程序可以出于其自身的目的，在本地或全局范围内绑定一个与内置函数具有相同名称的标识符。在本地或全局范围内绑定的名称要比在内置范围内绑定的名称的优先级更高，因此本地和全局名称将隐藏内置名称。开发者还可以重新绑定内置范围内的名称，参见第 7.1 节中对 Python 内置对象的介绍。但是，要非常小心，避免不小心隐藏了开发者的代码可能需要的内置对象。为开发者自己的变量使用像 file、input、list 和 filter 这样的自然语言名称是很有诱惑力的，但是尽量不要这么做：这些都是内置 Python 类型或函数的名称，并且，除非开发者养成了永远都不使用自己的名称屏蔽这些内置名称，否则在进行这样的隐藏操作时，迟早会在代码中遇到一些神秘的错误。

与大多数内置函数和类型相似，本节介绍的函数通常不能使用命名参数进行调用，而只能使用位置参数；本节将特别提到不受这个限制的所有情况。

表 8-2

__import__	__import__(*module_name*[,*globals*[,*locals*[,*fromlist*]]]) 加载以字符串 module_name 命名的模块，并返回结果模块对象。在默认情况下，globals 是 globals( )的结果，而 locals 是 locals( )的结果，他们都被__import__作为只读字典，并且只能用来得到与包有关的导入的上下文环境，参见第 7.3 节。在默认情况下，fromlist 是一个空白列表，但是也可以是一个字符串列表，用来命名要在 from 语句中导入的模块属性的名称。参见第 7.2 节以了解有关模块加载的更多详细信息。 实际上，在开发者调用__import__时，通常只传递第一个参数，除了在开发者使用__import__进行与包相关的导入的极少和不确定的情况下。如果开发者为了提供特殊的导入功能，而必须使用自己的函数替换内置__import__函数，这样可能会很不幸地必须考虑 globals、locals 和 fromlist 参数。

续表

abs	abs(*x*) 返回数字 x 的绝对值。在 x 是复数时，abs 返回 x.imag**2+x.real**2 的平方根（也被称为复数的大小）。否则，如果 x 小于零，abs 返回-x，如果 x 大于或等于 0，abs 返回 x。参见第 5.2 节对__abs__、__invert__、__neg__和__pos__的介绍。

all	all(*seq*)
	seq 是任意一个可迭代的参数。只要 seq 中的任何一个项目为假，则返回 False；否则（包括 seq 为空白的情况），all 将返回 True。只适用于 Python 2.5。与第 4.4 节中介绍的 and 和 or 运算符一样，只要结果已知，all 将停止计算，并返回相应的结果；对于 all 函数而言，这意味着只要遇到一个项目为假，计算将立即停止，但是如果 seq 的所有项目都为真，则计算会遍历整个 seq。下面是使用 all 函数的一个典型的简单示例：
	`if all(x>0 for x in numbers):`
	`  print "all of the numbers are positive"`
	`else:`
	`  print "some of the numbers are not positive"`

any	any(*seq*)
	seq 是任意一个可迭代的参数。如果 seq 中的任何一个项目为真，则 any 返回 True；否则（包括 seq 为空的情况），any 将返回 Flase。只适用于 Python 2.5。与第 4.4 节中介绍的 and 和 or 运算符一样，只要结果已知，any 将停止计算，并返回相应的结果；对于 any 函数而言，这意味着只要遇到一个项目为真，计算将立即停止，但是如果 seq 的所有项目都为假，则计算会遍历整个 seq。下面是使用 any 函数的一个典型的简单示例：
	`if any(x<0 for x in numbers):`
	`  print "some of the numbers are negative"`
	`else:`
	`  print "none of the numbers are negative"`

callable	callable(*obj*) 如果 obj 可以被调用，则返回 True，否则返回 False。如果对象是一个具有__call__方法的函数、方法、类、类型或实例，则该对象可以被调用。参见第 5.2 节中介绍的__call__。
chr	chr(*code*) 返回一个长度为 1 的字符串，单个字符对应于 ASCII/ISO 编解码器中的一个整数 code。参见本节后面介绍的 ord 函数和 unichr 函数。
cmp	cmp(*x*,*y*) 在 x 等于 y 时返回 0，在 x 小于 y 时返回-1，在 x 大于 y 时返回 1。参见第 5.2 节中的__cmp__。
coerce	coerce(*x*,*y*) 返回一个数值对，其中的两个项目是被转换为普通类型的数字 x 和 y。参见第 4.5 节。

续表

compile	`compile(string,filename,kind)` 编译一个字符串，并返回一个可以被 exec 或 eval 使用的代码对象。在 string 不是一个语法正确的 Python 语句时，compile 将引发 SyntacError。在 string 是一个多行复合语句时，最后一个字符必须是‘\n’。在 string 是一个表达式，并且其结果将用于 eval 时，kind 必须是‘eval’；否则，kind 必须是‘exec’。filename 必须是一个字符串，并且只能用于错误消息（在错误发生时）。参见本节后面的 eval 函数和第 13.1 节。
delattr	`delattr(obj,name)` 从 obj 中删除属性 name。delattr(obj,'ident')就像 del obj.ident 一样。如果 obj 具有一个名为 name 的属性，只是因为该对象的类中具有该属性（这是通常的情况，例如，obj 的方法），开发者不能从 obj 本身删除这个属性。开发者可以从该对象的类删除这个属性，而这取决于该类的元类是否允许。如果开发者可以删除这个类属性，obj 将不再具有这个属性，该类的所有其他对象也都一样，不再具有这个属性。
dir	`dir([obj])` 如果调用 dir( )时不带任何参数，该函数将返回一个包含当前范围内绑定的所有变量名称的排序列表。而 dir(obj)将返回一个包含 obj 的所有属性名称的排序列表，包括来自 obj 的类型或者继承的属性。参见本节后面的 vars 函数。
divmod	`divmod(dividend,divisor)` 两个数字相除，并返回一个数值对，其中的两个项目分别是商和余数。参见第 5.2 节中介绍的__divmod__。
eval	`eval(expr,[globals[,locals]])` 返回一个表达式的结果。expr 可以是一个准备进行计算的代码对象，或者是一个字符串。在字符串的情况下，eval 将调用 compile(expr, 'string', 'eval')获得一个代码对象。eval 将把该代码对象作为一个表达式进行计算，并使用 globals 和 locals 字典作为命名空间。在不带这两个参数时，eval 将使用当前的命名空间。eval 不能执行语句；而只能计算表达式的值。参见第 13.1 节。

execfile	`execfile(filename,[globals[,locals]])`
	exefile 是下面这条语句的快捷方式：
	`exec open(filename).read() in globals, locals`
	参见第 13.1 节。

filter	`filter(func,seq)`
	构建一个由 seq 中的项目组成的列表，列表中的项目可以让 func 的结果为真。func 可以是任意可以接受单个参数或 None 的可调用对象。seq 可以是任意可迭代对象。在 func 是可调用对象时，filter 将对 seq 中的每个项目调用 func，并返回由 func 的结果为真的项目组成的列表，就像下面的字符串复合语句一样：
	`[item for item in seq if func(item)]`
	在 seq 是一个字符串或元组时，filter 的结果也将是字符串或元组，而不是列表。在 func 为 None 时，filter 将检查值为 True 的项目，就像：
	`[item for item in seq if item]`

续表

getattr	`getattr(obj,name[,default])` 返回由字符串 name 命名的 obj 的属性。name. getattr(obj, 'ident')就像 obj.ident 一样。在给出了 default，并且 obj 中找不到 name 时，getattr 将返回 default，而不是引发 AttributeError。参见第 5.1 节。
globals	`globals()` 返回调用模块的__dict__（也就是，在调用时用作全局命名空间的字典）。参见本节后面的 locals 函数。
hasattr	`hasattr(obj,name)` 如果 obj 没有属性 name（也就是，如果 getattr(obj, name)引发了 AttributeError），则返回 False。否则，hasattr 将返回 True。参见第 5.1 节。
hash	`hash(obj)` 返回 obj 的哈希值。只有在 obj 可以被哈希时，obj 可以是一个字典键，或者集合中的一个项目。所有相等的数字都具有相同的哈希值，即使这些数字是不同的类型。如果 obj 的类型没有定义相等比较，hash(obj)通常将返回 id(obj)。参见第 5.2 节中的__hash__。
hex	`hex(x)` 将整数 x 转换为十六进制字符串表示。参见第 5.2 节中的__hex__和__oct__。
id	`id(obj)` 返回一个用来表示 obj 的标识的整数值。在 obj 的生命周期中，obj 的 id 是唯一和恒定的，但是在 obj 被垃圾收集之后，这个 id 可以在任何时候被重用。在一个类型或类没有定义相等比较时，Python 将使用 id 来比较和哈希实例。对于任何对象 x 和 y，标识检查 x is y 具有与 id(x)==id(y)相同的结果。
input	`input(prompt='')` input(prompt)是 eval(raw_input(prompt))的快捷方式。换句话讲，input 可以提示用户输入行，将结果字符串计算为一个表达式，并返回这个表达式的结果。隐式 eval 可能会引发 SyntaxError 或其他异常。对于大多数程序而言，input 确实是用户不友好的和不适当的，但是对于小程序和开发者自己的小探测脚本而言，input 是非常便利的。参见本节前面的 eval 函数和本节后面的 raw_input 函数。
intern	`intern(string)` 确保 string 被保存在一个由内部化字符串组成的表中，并返回 string 本身或一个副本。内部化字符串进行相等比较要比其他字符串稍微快一些，因为开发者可以使用运算符 is，而不是运算符==进行这样的比较。但是，垃圾收集永远都不能收回用于内部化字符串的内存，因此内部化太多字符串可能会因为消耗太多的内存而让开发者的程序变慢。本书没有在其他位置介绍内部化字符串的概念。
isinstance	`isinstance(obj,cls)` 在 obj 是类 cls（或 cls 的任意子类）的一个实例，或者在 cls 是一个类型对象，而 obj 是这个类型的对象时，该函数将返回 True。否则，该函数将返回 False。cls 还可以是一个元组，其中的项目都是类或类型。在这种情况下，如果 obj 是元组 cls 中任何一个项目的实例，则 isinstance 将返回 True；否则，isinstance 将返回 False。

issubclass	issubclass(*cls1*,*cls2*) 如果 cls1 是 cls2 的一个直接或间接子类，将返回 True；否则，将返回 False。cls1 和 cls2 必须是类或类型。

iter	iter(*obj*)
	iter(*func*,*sentinel*)
	创建并返回一个迭代器，迭代器是一个具有 next 方法的对象，开发者可以重复调用这个方法，一次获得一个项目（参见第 4.10 节中介绍的迭代器）。在带一个参数调用这个函数时，iter(obj)通常将返回 obj.__iter__()。在 obj 是一个不带特殊方法__iter__的序列时，iter(obj)等同于下面的生成器：
	`def iterSequence(obj):`
	`  i = 0`
	`  while 1:`
	`  try: yield obj[i]`
	`  except IndexError: raise StopIteration`
	`  i += 1`
	参见第 4.2 节和第 5.2 节中的__iter__。
	在带两个参数调用这个函数时，第一个参数必须是不带参数的可调用函数，而 iter(func,sentinel)等同于下面的生成器：
	`def iterSentinel(func, sentinel):`
	`  while 1:`
	`  item = func()`
	`  if item == sentinel: raise StopIteration`
	`  yield item`
	正如第 4.10 节中介绍的，语句 for x in obj 完全等同于 for x in iter(obj)；因此，不要在这样一个 for 语句中调用 iter，因为这是多余的，因此也是一种比较坏的 Python 编程风格。iter 是等幂的（idempotent）。换句话讲，在 x 是一个迭代器时，iter(x)就是 x，只要 x 的类提供一个__iter__方法，其方法体中只是一条 return self 语句，就像迭代器的类那样。

len	len(*container*) 返回容器 container 中项目的数量，这个容器可以是序列、映射或集合。参见第 5.2 节中介绍的__len__方法。
locals	locals() 返回一个表示当前本地命名空间的字典。将返回的字典作为只读字典；尝试修改该字典可能会影响本地变量的值，甚至还有可能引发一个异常。参见本节前面的 globals 函数和本节后面的 vars 函数。
map	map(*func*,*seq*,**seqs*) 对 seq 中的每个项目应用 func 函数，并返回一个结果列表。在使用 n+1 个参数调用 map 时，第一个参数 func 可以是任何可以接受 n 个参数或 None 的可调用对象。map 的其余参数必须是可迭代的。在 func 是可调用函数时，map 将使用 n 个参数（来自于每个可迭代对象中的一个对应

续表

map	项目）重复调用 func 并返回结果列表。例如，map(func, seq)等同于以下代码： `[func(item) for item in seq]` 在 func 为 None 时，map 将返回一个元组列表，每个元组具有 n 个项目（每个可迭代对象提供一个项目）；这非常类似于“zip”。但是，在可迭代对象具有不同的长度时，从概念上讲，map 将使用 None 补足较短的迭代对象，而 zip 将截断较长的迭代对象。
max	`max(s,*args)` 返回唯一的参数 s（s 必须是可迭代的）或多个参数中最大的参数中的最大项目。在 Python 2.5 中，开发者还可以使用一个位置参数和一个与 sorted 函数具有相同语义的命名参数 key= 来调用 mmax，参见本节后面的 sorted 函数。
min	`min(s,*args)` 返回唯一的参数 s（s 必须是可迭代的）或多个参数中最小的参数中的最小项目。在 Python 2.5 中，开发者还可以使用一个位置参数和一个与 sorted 函数具有相同语义的命名参数 key= 来调用 min，参见本节后面的 sorted 函数。
oct	`oct(x)` 将整数 x 转换为八进制字符串表示。参见第 5.2 节中的_ _hex_ _和_ _oct_ _。
ord	`ord(ch)` 返回单字符字符串 ch 的 0 到 255（包含）之间的 ASCII/ISO 整数代码。在 ch 为 Unicode 时，ord 将返回一个 0～65534（包含）的整数代码。参见本节前面的 chr 函数和本节后面的 unichr 函数。
pow	`pow(x,y[,z])` 在给出了 z 时，pow(x,y,z)将返回 x**y%z。在没有给出 z 时，pow(x, y)将返回 x**y。参见第 5.2 节中的_ _pow_ _。

range	`range([start,]stop[,step=1])`
	按等差数列返回一个整数列表。
	`[start, start+step, start+2*step,...]`
	在没有给出 start 时，其默认值为 0。在没有给出 step 时，其默认值为 1。在 step 为 0 时，range 将引发 ValueError。在 step 大于 0 时，最大的项目是严格小于 stop 的最大 start+i*step。在 step 小于 0 时，最后一个项目是严格大于 stop 的最小 start+i*step。在 start 大于或等于 stop，而 step 大于 0 时，或者在 start 小于或等于 stop，而 step 小于 0 时，这个函数的结果为一个空白列表。否则，结果列表中的第一个项目始终是 start。参见第 8.1 节中的 xrange。

raw_input	`raw_input(prompt='')` 向标准输出写入 prompt，从标准输入读取一行输入，并以字符串返回该行（不带\n）。在文件结束（end-of-file）时，raw_input 将引发 EOFError。参见本节前面的 input 函数。

reduce	reduce(*func*,*seq*[,*init*])
	从左到右对 seq 中的项目应用 func，可以将可迭代对象减少到单个值。func 必须是可调用的，并带有两个参数。reduce 可以对 seq 的前两个项目调用 func，然后对第一个调用的结果和第三个项目调用 func，依此类推。reduce 将返回这样的最后一个调用的结果。在给出了 init 时，这个 init 用在 seq 的第一个项目之前。在没有给出 init，并且 seq 只有一个项目时，reduce 将返回 seq[0]。与此相似，在给出了 init，并且 seq 为空时，reduce 将返回 init。这样，reduce 大致上等同于：
	`def reduce_equivalent(func,seq,init=None):`
	`  seq = iter(seq)`
	`  if init is None: init = seq.next()`
	`  for item in seq: init = func(init,item)`
	`  return init`
	下面给出了 reduce 的一个示例，该示例用来计算一个数字序列的乘积：
	`theprod = reduce(operator.mul, seq, 1)`

reload	reload(*module*) 重新加载和重新实例化模块对象 module，并返回 module。参见第 7.2 节。
repr	repr(*obj*) 返回 obj 的一个完整和明确的字符串表示。只要有可能，repr 将返回一个字符串，开发者可以将其传递到 eval 以使用与 obj 相同的值创建一个新对象。参见第 10.12 节中的 str 和第 5.2 节中的__repr__。
round	round(*x*,*n*=0) 返回一个浮点型数字，其值为 x 舍入到小数点后 n 个数字（也就是，最接近 x 的 10**-n 的倍数）。在有两个这样的乘数接近于等于 x 时，round 将返回远离 0 的那个。因为当今的计算机都是按二进制表示浮点数字的，而不是十进制，大多数 round 的结果并不十分精确。参见第 15.4 节。
setattr	setattr(*obj*,*name*,*value*) 将 obj 的属性 name 绑定到 value。setattr(obj,'ident',val)就像 obj.ident=val 一样。参见第 4.3 节中介绍的对象属性和项目和第 5.1 节中介绍的设置属性。

sorted	sorted(*seq*,*cmp*=None,*key*=None,*reverse*=False)
	返回一个按排序顺序的，与可迭代对象 seq 具有相同项目的列表。等同于：
	`def sorted(seq,cmp=None,key=None,reverse=False):`
	`  result = list(seq)`
	`  result.sort(cmp,key,reverse)`
	`  return result`
	参见第 4.6 节中介绍的列表排序以了解这些参数的含义；sorted 是少数几个可以使用命名参数进行调用的内置函数之一。Python 2.4 中的新功能。

续表

sum	sum(*seq*,*start*=0) 返回可迭代对象 seq（seq 必须是数字，特别是，不能是字符串）中的项目，再加上 start 的值的和。在 seq 不包含任何项目时，该函数将返回 start。要想“求和”（连接）字符串序列，可以使用 ''.join(seqofstrs)，参见第 9.1 节和第 18.4 节中介绍的从片段构建字符串。
unichr	unichr(*code*) 返回一个 Unicode 字符串，其中的单个字符对应于 code，code 是 0～65535（包含）的一个整数。参见第 10.12 节和前面的 ord 函数。
vars	vars([*obj*]) 在不带参数调用该函数时，vars( )将返回一个表示当前范围内（就像 locals 一样，参见本节前面的 locals 函数）被绑定的所有变量的字典。这个字典必须被看作是只读的。vars(obj)将返回一个字典，该字典表示当前在 obj 中绑定的所有属性，参见本节前面的 dir 函数。这个字典可以是可修改的，这取决于 obj 的类型。
zip	zip(*seq*,**seqs*) 返回一个元组列表，其中的第 n 个元组包含来自于每个参数序列的第 n 个元素。zip 必须可以使用至少一个参数调用，并且所有参数必须是可迭代的。如果这些可迭代参数具有不同的长度，zip 将按最短的可迭代参数返回一个列表，并忽略其他可迭代对象中的拖尾项目。参见本节前面的 map 函数。

8.3 sys 模块

sys 模块的属性都被绑定到可以提供 Python 解释器的状态或直接影响 Python 解释器的运行的数据和函数上。本节按字母顺序详细列举了 sys 模块的大多数常用属性。

表 8-3

argv	传递给主脚本的命令行参数列表。argv[0]是这个主脚本的名称和全路径，如果要使用-c 选项，则这个参数是‘-c’。参见第 8.10 节了解一个使用 sys.argv 的好方法。

displayhook	displayhook(*value*)
	在交互式会话中，Python 解释器将调用 displayhook，向其传递输入的每个表达式语句的结果。如果 value 为 None，默认的 displayhook 不做任何操作；否则，将保存并显示 value：
	`if value is not None:`
	`    __builtin__._ = value`
	`print repr(value)`
	开发者可以重新绑定 sys.displayhook 来更改交互式行为，其初始值可以使用 sys.__displayhook__。

续表

excepthook	excepthook(*type*,*value*,*traceback*) 当一个异常没有被任何处理器捕获，并因此通过传播遍历了整个调用堆栈时，Python 将调用 excepthook，并向其传递异常类、异常对象和跟踪对象，参见第 6.2 节。默认的 excepthook 显示了错误和跟踪。开发者可以重新绑定 sys.excepthook 以更改在没有捕获到异常时的显示信息（在 Python 返回到交互式循环或者终止时）。初始值可以使用 sys.__excepthook__。
exc_info	exc_info() 如果当前线程正在处理一个异常，exc_info 将返回一个元组，其中包含的 3 个项目分别是异常的类、对象和跟踪。如果当前线程没有在处理任何异常，exc_info 将返回(None,None,None)。跟踪对象将间接保存对传播异常的所有函数的所有变量的引用。因此，如果要保存一个对跟踪对象的引用（例如，可以间接地将一个变量绑定到 exc_info 返回的整个元组上），Python 必须将其保存在内存数据中，而该内存有可能会被垃圾收集；因此要确定任何对跟踪对象的绑定都只持续一小段时间。要想确保这个绑定被解除，可以使用 try/finally 语句（参见第 6.1 节）。要想打印跟踪对象中的信息，参见第 18.2 节。
exit	exit(*arg*=0) 产生一个 SystemExit 异常，通常在执行完 try/finally 语句语句安装的清理处理程序之后终止执行。如果 arg 是一个整数，Python 将使用 arg 作为这个程序的退出码：0 表示成功地终止，而任何其他值表示程序失败地终止。大多数平台要求退出码范围为 0～127。如果 arg 不是一个整数，Python 将把 arg 打印到 sys.stderr，程序的返回码则为 1（也就是，一个通用的失败终止代码）
getdefaultencoding	getdefaultencoding() 返回用来编码和解码 Unicode 和字符串对象（通常是‘ascii’）的默认编解码器的名称。参见第 9.6 节了解有关 Unicode、编解码器、编码和解码的详细信息。
getrefcount	getrefcount(*object*) 返回对象 object 的引用次数。引用计数。参见第 13.3 节对引用计数的介绍。
getrecursionlimit	getrecursionlimit() 返回 Python 调用堆栈深度的当前限定值。参见第 4.11 节和本节后面的 setrecursionlimit 属性。
_getframe	_getframe(*depth*=0) 返回一个来自调用堆栈的框架对象。在 depth 等于 0 时，结果是_getframe 的调用者的框架。在 depth 等于 1 时，结果是这个调用者的调用者的结果，依此类推。_getframe 的前导下划线_用来提醒开发者这是一个只能用于内部专用目的的私有系统函数。调试（参见第 18.2 节）是使用_getframe 属性的一个典型的好原因。
maxint	当前版本的 Python 中最大的整数（至少为 2**31，也就是 2147483647）。由于二进制的补码表示原理，因此负整数不能小于-maxint-1。

续表

modules	一个字典，其中的项目是所有加载的模块的名称和模块对象。参见第 7.2 节了解有关 sys.modules 的详细信息。
path	一个字符串列表，指定了 Python 在查找要加载的模块时搜索的字典和 ZIP 文件。参见第 7.2 节以了解有关 sys.path 的更多信息。
platform	一个字符串，表示这个程序正在运行的平台的名称。其典型值是简化的操作系统名称，比如‘sunos5’、‘linux2’和‘win32’。

ps1, ps2	ps1 和 ps2 指定了主要和次要解释器提示符字符串，其初始值分别是‘>>>’和‘...’。这些属性只存在于交互式解释器会话中。如果开发者将其中任意一个属性绑定到一个非字符串对象 x，Python 将对每次提示符输出的对象调用 str(x)以显示 Python 提示符。这个功能可以通过编写一个定义了__str__的类，并将这个类的一个实例赋值给 sys.ps1 和/或 sys.ps2，让开发者创建动态提示符。例如，要想获得编号提示符：
	`>>> import sys`
	`>>> class Ps1(object):`
	`...  def __init__(self):`
	`...  self.p = 0`
	`...  def __str__(self):`
	`...  self.p += 1`
	`...  return '[%d]>>> ' % self.p`
	`...`
	`>>> class Ps2(object):`
	`...  def __str__(self):`
	`...  return '[%d]... ' % sys.ps1.p`
	`...`
	`>>> sys.ps1 = Ps1(); sys.ps2 = Ps2()`
	`[1]>>> (2 +`
	`[1]... 2)`
	`4`
	`[2]>>>`

setdefaultencoding	`setdefaultencoding(`*name*`)` 设置用来编码和解码 Unicode 和字符串对象（通常是‘ascii’）的默认编解码器。只有在程序启动时才能从 sitecustomize.py 中调用 setdefaultencoding 属性；然后，site 模块将从 sys 中删除这个属性。开发者可以调用 reload(sys)让这个属性再次可用，但是这不是一个良好的编程习惯。参见第 9.6 节中的 Unicode、编解码器、编码和解码。参见第 13.5 节中介绍的 site 和 sitecustomize 模块。
setprofile	`setprofile(`*profilefunc*`)` 设置一个全局配置文件函数，然后，Python 将在每次进入函数和在函数返回时调用一个可调用对象。参见第 18.4 节以了解如何操作配置文件。

续表

setrecursionlimit	`setrecursionlimit(`*`limit`*`)` 设置 Python 的调用堆栈的深度限定值（默认值为 1 000）。这个限定值可以防止无限递归导致 Python 崩溃。设置这个限定值对于依赖于深度递归的程序是必需的，但是大多数平台不能支持非常大的调用堆栈深度限制。更有用的是，降低这个限定值有助于开发者在测试和调试期间对程序进行检查，可以在程序几乎要崩溃的递归情况下很好地逐步降低递归深度，而不是突然崩溃。参见第 4.11 节和本节前面的 getrecursionlimit 属性。
settrace	`settrace(`*`tracefunc`*`)` 设置一个全局跟踪函数，然后 Python 将调用一个可调用对象，就像每个逻辑源代码行的执行一样。settrace 可以用于实现工具，比如配置文件、覆盖分析器和调试器。本书中没有进一步介绍跟踪。
stdin, stdout, stderr	stdin、stdout 和 stderr 都是预定义的文件对象，这些文件对象对应于 Python 的标准输入、输出和错误流。开发者可以将 stdout 和 stderr 重新绑定到类文件对象中（提供了一个可以接受字符串参数的 write 方法的对象）以重定向输入和错误消息的目的。开发者还可以将 stdin 重新绑定到一个打开以供读取的类文件对象中（提供了一个返回字符串的 readline 方法的对象）以重定向源，内置函数 raw_input 和 input 可以从该源读取信息。可用的初始值为__stdin__、__stdout__和__stderr__。第 10.3 节介绍了文件对象。
tracebacklimit	显示未经处理的跟踪的最大层数。在默认情况下，不设置这个属性（也就是，没有任何限制）。在 sys.tracebacklimit 为<=0 时，Python 将禁止跟踪，只打印异常类型和值。
version	一个字符串，表明了 Python 的版本、编译号和日期，以及使用的 C 编译器。对于 Python 2.3，version[:3] 等于 '2.3'，对于 Python 2.4，则等于 '2.4'，依此类推。

8.4 copy 模块

正如第 4.3 节中介绍的，Python 中的赋值语句并不复制被赋值的右边对象。而是，赋值语句将向右边对象添加一个引用。在开发者想要获得对象 x 的一个副本时，可以要求 x 创建 x 的一个副本，或者可以要求 x 的类型创建从 x 赋值的一个新实例。如果 x 是一个列表时，list(x)将返回 x 的一个副本，就像 x[:]一样。如果 x 是一个字典，dict(x)和 x.copy( )将返回 x 的一个副本。如果 x 是一个集合，set(x)和 x.copy( )将返回 x 的一个副本。在各种情况中，本书强烈推荐使用调用类型的统一和可读的习惯用法，但是在 Python 专家社区中并不一致认可使用这种风格。

copy 模块提供了一个 copy 函数以创建并返回多个对象类型的一个副本。普通副本（比如，列表 x 的 list(x)和 copy.copy(x)）也被称为浅（shallow）副本：在 x 具有到其他对象（比如项目或属性）的引用时，x 的一个普通（浅）副本也具有到相同其他对象的不

同引用。但是，有时候开发者需要一个深（deep）副本，其中引用的对象都被递归复制；幸运地是，这种需求极少，因为一个深入副本需要花费大量内存和时间。模块 copy 提供了一个 deepcopy(x)函数以创建并返回一个深入副本。

表 8-4

copy	copy(*x*) 为多种类型的 x 创建并返回 x 的一个浅副本（但是，不支持几种类型的副本，比如模块、类、文件、框架和其他内部类型）。如果 x 是不可变的，copy.copy(x)将返回 x 本身作为优化。一个类可以通过特殊方法__copy__(self)自定义 copy.copy 复制其实例的方法，这个特殊方法将返回一个新对象，也就是 self 的一个副本。

deepcopy	deepcopy(*x*,[*memo*])
	创建 x 的一个深入副本并返回该副本。深入复制意味着递归遍历引用的有向（不一定是无环的）图。必须进行特殊防护以保持图的形状：在遍历过程期间，对相同对象的引用遇到了很多次，必须创建不同的副本。正确的做法是，必须使用相同的复制对象的引用。考察以下简单示例：
	`sublist = [1,2]`
	`original = [sublist, sublist]`
	`thecopy = copy.deepcopy(original)`
	original[0] is original[1]为 True（也就是，引用相同对象的列表 original 的两个项目）。这是 original 的一个重要属性，因此必须被保持在被声明为该列表的一个副本的任何对象中。在这种情况下，copy.deepcopy 定义的语义用来确保 thecopy[0] is thecopy[1]也是 True，换句话讲，original 和 thecopy 的引用的图的形状是相同的。避免重复复制具有一个非常重要的好处：可以防止无限循环，否则会在引用的图有循环时出现无限循环。
	copy.deepcopy 可以接受第二个可选参数 memo，这个参数是一个字典，用来将已经复制的对象的 id( )映射到已经是其副本的新对象上。memo 是由 deepcopy 的所有对其自身的递归调用传递的，并且，如果需要在初始标识和对象副本之间维护这样一个对应的映射，开发者可能还要显式传递该参数（通常作为一个初始空白字典）。
	一个类可以通过包含一个可以返回新对象（self 的深入副本）的特殊方法__deepcopy__(self, memo)来自定义 copy.deepcopy 复制其实例的方式。在__deepcopy__需要深入复制某些引用的对象 subobject 时，必须通过调用 copy.deepcopy(subject, memo)来实现。在一个类中没有特殊方法__deepcopy__时，对这个类的实例调用的 copy.deepcopy 还将尝试调用特殊方法__getinitargs__、__getnewargs__、__getstate__和__setstate__，参见第 11.1 节中介绍的封装实例”。

8.5 collections 模块

collections 模块（Python 2.4 中引入）的目的就是最终提供几种有趣的集合（也就是，容器）类型。

deque

在 Python 2.4 中，collections 模块只提供了一种类型 deque，这个类型的实例就是“双端队列（Double-Ended Queue）”（也就是，适合于在两端添加和删除的类似于序列的容器）。使用单个参数（任意可迭代对象）调用 deque 可以获得一个新 deque 实例，该实例的项目是以相同顺序排列的可迭代对象，而不带任何参数调用 deque 可以获得一个新的空白 deque 实例。一个 deque 实例 d 是一个可变序列，并且因此可以被索引和迭代（但是，d 不能被切片，不管是访问、重新绑定或者删除，一次只能索引一个项目）。deque 的一个实例 d 提供了如表 8-5 所示的方法。

表 8-5

append	*d*.append(*item*) 在 d 的右边（末尾）添加项目 item。
appendleft	*d*.appendleft(*item*) 在 d 的左边（开始）添加项目 item。
clear	*d*.clear() 删除 d 中的所有项目。
extend	*d*.extend(*iterable*) 在 d 的右边（末尾）添加 iterable 中的所有项目。
extendleft	*d*.extendleft(*item*) 在 d 的左边（开始）添加 iterable 中的所有项目。
pop	*d*.pop() 删除并返回 d 中的最后一个（最右边的）项目。如果 d 为空，则引发 IndexError。
popleft	*d*.popleft() 删除并返回 d 中的第一个（最左边的）项目。如果 d 为空，则引发 IndexError。
rotate	*d*.rotate(*n*=1) 将 d 向右旋转 n 步（如果 n<0，则向左旋转）。

defaultdict

在 Python 2.5 中，collections 模块还提供了 defaultdict 类型。defaultdict 是 dict 的子类，并添加了一个名为 default_factory 的按实例属性。在 defaultdict 的实例 d 的 d.default_factory 值为 None 时，d 的工作方式与普通字典完全一样。否则，d.default_factory 必须是可调用的，并且不带任何参数，这样，除了在 d 是通过一个不在 d 中的键 k 索引时之外，d 的工作方式与普通字典相似。在这种特殊情况下，索引的 d[k]将调用 d.default_factory( )，将其结果赋值为 d[k]的值，并返回这个结果。换句话讲，类型 defaultdict 的工作方式就像下面这个使用 Python 编写的类：

```
class defaultdict(dict):
    def __init__(self, default_factory, *a, **k):
        dict.__init__(self, *a, **k)
        self.default_factory = default_factory
    def __getitem__(self, key):
        if key not in self and self.default_factory is not None:
            self[key] = self.default_factory()
        return dict.__getitem__(self, key)
```

正如这段等效的 Python 代码表示的，要想实例化 defaultdict，需要向其传递一个额外的第一个参数（在任何其他参数之前，位置参数和/或命名参数将被传递到普通 dict 上）。这个额外的第一个参数将成为 default_factory 的初始值。defaultdict 的其他工作方式基本上也可以通过这段等效的 Python 代码来表示（使用 str 和 repr 异常，这两个异常将返回不同于从普通 dict 返回的字符串）。开发者可以读取并重新绑定 default_factory，其他一些方法不会受到影响，比如 get 和 pop，与键（通过 in 等运算符测试方法 keys、迭代和成员关系）的集合有关的所有行为可以确切地反映当前在容器中的键（不管开发者是显式还是隐式通过索引调用 default_factory 来放置的）。

defaultdict 的一个典型使用就是可以作为包 bag（也被称为多重集合，multiset），也就是一个类似于字典并保存了对应于每个键的出现次数的对象；对于这种使用，default_factory 的一个自然的选择就是 int，因为调用 int( )将返回 0（还没有包含在 bag 中的键的正确出现次数）。例如：

```
import collections, operator
def sorted_histogram(seq):
    d = collections.defaultdict(int)
    for item in seq: d[item] += 1
    return sorted(d.iteritems, key=operator.itemgetter(1),
reverse=True)
```

在调用 sorted_histogram 函数，并以其中的项目都是可哈希项目的任何可迭代对象为参数时，这个函数将返回一个数据对列表，每个数据对的形式为（item, count），这个数据对给出了可迭代对象中的所有不同项目，以及对应的出现次数，按照出现次数的相反顺序排序（最常看到的项目最先开始）。defaultdict 的另外一个典型的使用就是可以将 default_factory 设置为 list，建立从键到由值组成的列表的映射：

```
def multi_dict(items):
    d = collections.defaultdict(list)
    for key, value in items: d[key].append(value)
    return d
```

在调用 multi_dict 函数，并以其中的项目是形式为（key, value）数据对的任何可迭代对象，且所有键都是可哈希的值为参数时，这个函数将返回一个映射，并将每个键关联到由多个值组成的列表上，键和值都出现在可迭代对象中（如果开发者想要得到一个纯粹的 dict 结果，可以将 last 语句更改为 return dict(d)语句；但是这通常

并不是必需的）。

8.6 functional 模块

functional 模块（在 Python 2.5 中引入）的目的是打算最终提供几个有趣的函数和类型以支持 Python 中的功能编程。在编写本书的时候，业界还在争论是将这个模块更名为 functools，还是再引入一个单独的 functools 模块，以保持高阶函数不会严格地与功能编程的习惯用法相关联（特别是，内置装饰器）。但是，还是在编写本书的时候，只有一个函数被确定性地接受为包含在 functional 模块中。

表 8-6

partial	`partial(func,*a,**k)`
	func 可以是任意一个可调用函数。partial 可以返回类似于 func 的另一个可调用函数 p，但是这个函数是以已经被绑定到 a 和 k 中给定的值的位置参数和/或命名参数为参数的。换句话讲，p 是 func 的分部应用程序，通常也被称为（其正确性是有争议的，但是很有趣）给定参数的 Curry（为纪念数学家 Haskell Curry 而命名）func。例如，假定开发者有一个由数字组成的列表 L，并且想要将其中的负数截短为 0。在 Python 2.5 中，有一个方法可以实现这个功能：
	`L = map(functional.partial(max, 0), L)`
	还可以选择使用下面这个使用 lambda 的代码段：
	`L = map(lambda x: max(0, x), L)`
	和列表推导：
	`L = [max(0, x) for x in L]`
	在要求回调函数的环境中，functional.partial 实际上得到了其自身，比如 GUI 的事件驱动编程（参见第 17 章）和网络互联应用程序（参见第 20.3 节）。

8.7 bisect 模块

bisect 模块可以在项目被插入到一个列表中时使用二分算法保持这个列表的排序顺序。在每次插入一个项目后，bisect 模块的操作要比调用一个列表的 sort 方法更快，本节将详细介绍 bisect 模块提供的主要函数。

表 8-7

bisect	`bisect(seq,item,lo=0,hi=None)` 返回 seq 序列中的索引 i，item 必须被插入到这个位置以保持 seq 序列的顺序。换句话讲，对于索引 i，seq[:i] 中的每个项目都小于或等于 item，而 seq[i:] 中的每个项目都大于 item。seq 必须是一个排序序列。对于任意排序序列 seq，seq[bisect(seq,y)-1]==y 等同于 y in seq，但是如果 len(seq)很大，则前者的运行速度更快。开发者可以向该函数提供可选参数 lo 和 hi，以对切片 seq[lo:hi] 进行操作。

续表

insort	insort(*seq*, *item*, *lo*=0, *hi*=None) 与 seq.insert(bisect(seq, item), item)相似。换句话讲，seq 必须是一个排序的可变序列（通常是一个排序列表），并且 insort 可以通过在右边插入一个 item 来修改 seq，这样可以保持 seq 的顺序。开发者可以向该函数提供可选参数 lo 和 hi，以对切片 seq[lo:hi]进行操作。

模块 bisect 还提供了 bisect_left、bisect_right、insort_left 和 insort_right 函数以显式控制包含重复单元的序列的搜索和插入策略。bisect 等同于 bisect_right，而 insort 等同于 insort_right。

8.8 heapq 模块

heapq 模块可以在项目被插入到列表和从列表中提取时使用堆（heap）算法保持这个列表的“近似排序（nearly sorted）”顺序。在每次执行插入操作之后，heapq 的运算要比调用列表的 sort 方法或使用 bisect 更快一些，并且，在许多情况下（比如，实现“优先级队列”），heapq 支持的近似排序顺序与完整排序顺序几乎一样有用。模块 heapq 提供了如表 8-8 所示的函数。

表 8-8

heapify	heapify(*alist*) 根据需求改变 alist 的顺序，使其满足堆条件：对于任意 i>=0，alist[i]<=alist[2*i+1]且 alist[i]<=alist[2*i+2]（如果所有待查的索引<len(alist)）。请注意，如果一个列表满足这个堆条件，该列表的第一个项目是最小的一个（或者是并列最小的）。排序列表都满足堆条件，但是，还是有其他一些列表的排列顺序满足堆条件，但是不要求该列表被完全排序。heapify 的时间复杂度为 O(len(alist))。
heappop	heappop(*alist*) 删除并返回 alist 的最小（第一个）项目，alist 是一个满足堆条件，并且可以改变 alist 中的剩余项目的顺序，以确保删除项目之后的 alist 仍然满足堆条件的列表。heappop 的时间复杂度为 O(len(alist))。
heappush	heappush(*alist*, *item*) 将项目 item 插入到 alist 中，alist 是一个满足堆条件，并且可以改变 alist 中的一些项目的顺序，以确保插入插入项目之后的 alist 仍然满足堆条件的列表。heappush 的时间复杂度为 O(len(alist))。

heapreplace	heapreplace(*alist*, *item*)
	逻辑上等于在运行 heappop 之后再运行 heappush，类似于：
	`def heapreplace(alist, item):`
	`  try: return heappop(alist)`
	`  finally: heappush(alist, item)`
	heapreplace 的时间复杂度为 O(len(alist))，并且通常要比上面显示的逻辑上相同的函数运行得更快。

续表

nlargest	nlargest(*n*,*seq*) 返回可迭代序列 seq 中 n 个最大项目组成的一个反转排序的列表（如果 seq 只有少于 n 个的项目，则列表中的项目数也小于 n）；就像 sorted(seq, reverse=True)[:n]函数一样，但是运行得更快。在 Python 2.5 中，开发者还可以指定一个 key=参数，就像在 sorted 中使用的那样。
nsmallest	nsmallest(*n*,*seq*) 返回可迭代序列 seq 中的 n 个最小项目组成的一个排序列表（如果 seq 只有少于 n 个的项目，则列表中的项目数也小于 n）；就像 sorted(seq)[:n] 函数一样，但是运行得更快。在 Python 2.5 中，开发者还可以指定一个 key= 参数，就像在 sorted 中使用的那样。

8.9 UserDict 模块

Python 的标准库的 UserDict 模块（类似于那些现在看起来已经有些过时的模块 UserList 和 UserString）的主要内容通常都是用来模仿标准内置容器类型的行为的类。这些类在传统的对象模型中曾经非常有用，这是因为开发者不能从内置类型继承类，并且我们现在所说的传统对象模型曾经是 Python 中唯一可用的对象模型。现在，开发者只需使用内置类型（比如 list 和 dict）的子类即可，因此，使用 Python 编码的类来模仿内置类型并没有太大的意义。不过，UserDict 模块确实还包含一个仍然极其有用的类。

实现被定义为“映射”的全部接口（也就是，一个字典的接口）需要进行大量的编程，因为字典具有很多有用和方便的方法。UserDict 模块提供了一个类 DictMixin，这个类可以让开发者很容易地编写提供全部映射接口的类，而只需要编写最小数量的方法：开发者只需要从 UserDict.DictMixin 继承（也有可能是多重继承）类即可。在最小情况下，开发者的类必须定义方法_ _getitem_ _、key 和 copy；如果这些类的实例是可变的，则这些类还必须定义方法_ _setitem_ _和_ _delitem_ _。

为了提高效率，开发者还可以有选择地定义方法_ _contains_ _、_ _iter_ _和/或 iteritems；如果开发者没有定义这些方法，则 DictMixin 超类将代替开发者定义这些方法，但是从 DictMixin 继承得到的方法版本可能要比开发者根据自己对类的特定具体结构的了解而自己定义的方法的效率更低一些（另一方面，映射接口的许多其他方法将会得到合理有效的实现，即使只是从 DictMixin 继承而来，因此通常并不特别需要在开发者自己的类中定义这些方法）。

8.10 optparse 模块

optparse 模块提供了非常丰富和强大的方法以解析用户传递的用来运行 Python 程序（通

过在命令行上使用语法元素来实现，比如-x 或- -foo=bar，并将其放在程序名之后，而在其他程序参数之前）的命令行选项（也叫作，标记）。实例化（不带参数）模块提供的 OptionParser 类并组装这个实例，这样，该实例就知道程序的选项了（通过调用其 add_option 方法），并最终调用该实例的 parse_args 方法以处理程序的命令行、正确地处理每个选项并返回选项值的集合和非选项参数列表。

optparse 支持许多高级方法以自定义和调整程序的选项解析行为。在大多数情况下，开发者可以接受 optparse 的合理默认值，并且只使用下面将要介绍的类 OptionParser 的两个方法来有效使用这个默认值（省略了这些方法的许多高级选项，开发者通常不需要用到这些选项）。要想了解这个模块的所有强大和复杂的详细信息，可以查看 Python 的在线文档。

OptionParser 的实例 p 提供了如表 8-9 所示的方法。

表 8-9

add_option	*p*.add_option(*opt_str*,**opt_strs*,***kw*) 位置参数指定了选项字符串，也就是，用户传递的，作为程序命令行的一部分的字符串，这个字符串用来设置程序的选项。每个参数都可以是一个短格式选项字符串（以单个破折号[连字号]开始，后面带单个字符或数字）或者长格式选项字符串（以两个破折号开始，后面带一个可能还会包含破折号的标识符）。通常需要确定性地传递一个短格式和一个长格式选项字符串，但是传递多个“同义字”、只传递短格式，或者只传递长格式也都是可以的。 从字面上来看，命名（可选）参数（kw）也就是这个操作的位置，因为最重要的参数就是命名的这个 action，具有一个默认值‘store’，这个值指定了与该选项字符串相关的值被绑定为选项对象的一个属性，也就是 parse_args 方法返回的值。 ‘store’暗示，如果运行程序的用户完全地提供了这个选项，则该用户还必须提供一个与这个选项相关的值。如果该用户完全没有传递这个选项，则这个选项对象的相关属性将被设置为 None。要想使用一个不同的默认值，要在对 add_option 的调用中包含一个像 default='foo' 的命名参数（这样将使用‘foo’作为该属性的默认值）。如果用户不传递这个选项，后面必须带一个与其相关的值。这个值可以是紧跟在程序后面的命令行参数，或者，对于短格式选项字符串，这个值可以与选项字符串并列放置，作为相同参数的一部分——只需要串联放置即可；对于长格式选项字符串，需要在这个值与字符串中间放置一个等号（=）进行连接。否则，parse_args 将发出一个错误消息，并引发一个 SystemExit 异常。 与一个选项相关的选项对象的属性的名称是一个位于其第一个长格式选项字符串中的标识符（如果有破折号，将被替换为下划线），如果该选项没有长格式，则这个名称是第一个短格式选项字符串中的单个字符。要想为这个属性使用一个不同的名称，可以在对 add_option 的调用中包含一个像 dest='foo' 这样的命名参数（这里使用了名称‘foo’来表示与这个选项相关的属性）。 选项的值通常是一个字符串，但是开发者可以在对 add_option 的调用中包含一个像 type='int' 这样的命名参数（这意味着与这个选项相关的值必须是一个整数的十进制表示法，并且被保存为 int 型）。命名参数 type 可以接受的

续表

add_option	值是 string、int、long、choice、float 和 complex。在传递 type='choice' 时，还必须传递另一个命名参数 choices=... ，这个参数的值是一个字符串列表，用户必须从其中选择一个作为与这个选项相关的值（如果用户传递这个选项）。 action 的其他有用的值不要求（也不允许）用户传递一个与该选项相关的值。‘store_const’ 可以在与这个选项相关的属性中保存一个常数值（如果用户传递这个选项）；在开发者传递 action='store_const' 时，还必须传递另一个命名参数 const=... ，这个参数的值是保存在属性中的一个常数。为了方便，action 还可以是 ‘store_true’ 或 ‘store_false’，分别相当于 const=True 或 Flase 的 ‘store_const’。action='count' 会导致与这个选项相关的属性递增（在用户每次传递这个选项时增加 1），这个属性本质上就是一个初始值为 0 的整数。 总而言之，add_option 最常用的命名参数是如下。 action 在用户传递这个选项时执行的动作；可以是 ‘store’、‘store_const’、‘store_true’、‘store_false’、‘count’，或者本书没有介绍的几个高级动作，比如 ‘callback’、‘append’、‘help’ 和 ‘version’。 choices 在 action='store' 且 type='choice' 是，允许作为选项的值的字符串列表。 const 在 action='store' 时，用户传递这个选项时要保存的常数。 default 在用户不传递这个选项是要保存的值（默认值为 None）。 dest 与这个选项相关的属性的名称（默认值是第一个长格式选项字符串的标识符，或者在没有长格式选项字符串时，是第一个短格式选项字符串的单个字符）。 help 如果用户传递选项 ‘-h’ 或 ‘--help’，则显示用来解释这个选项的文本。 type 与这个选项相关的值的类型，可以是 ‘string’、‘int’、‘long’、‘choice’、‘float’ 或 ‘complex’。 optparse 也允许许多高级的自定义动作，包括添加其他可能的动作和类型的功能，不过，本书没有介绍这些动作。
parse_args	`p.parse_args(args=sys.argv[1:])` 解析开发者传递为 args 的字符串列表（在默认情况下，也就是程序的命令行参数），并返回一个数值对 options 和 arguments。options 是一个带有属性集的选项对象，这个属性集是根据程序的可用选项和 parse_args 刚解析的参数列表建立的；arguments 是一个字符串列表（可能是一个空白列表），也就是 args 中的那些与选项无关的参数。 如果 parse_args 发现了任何解析错误（包括未知的选项、没有值与一个需要值的选项关联、有些值关联到一个不需要值的选项、无效类型等），parse_args 将向 sys.stderr 写入一个错误消息并引发 SystemExit。如果 parse_args 在选项中找到了一个 ‘-h’ 或 ‘--help’，将向 sys.stdout 写入一个帮助消息并引发 SystemExit。 parse_args 还提供了高级自定义功能，本书没有介绍这些功能。

parse_args

下面是一个使用 optparse 的简单和有趣的示例。将以下代码写入到一个名为 hello.py 的文件：

```
#!/usr/bin/python
import optparse

def main():
    p = optparse.OptionParser()
    p.add_option('--verbose', '-v', action='store_true')
    p.add_option('--name', '-n', default="world")
options, arguments = p.parse_args()
    if options.verbose: print "Greetings and
salutations,",
   else: print "Hello",
   print '%s!' % options.name

if __name__ == '__main__':
    main()
```

这个脚本（参见第 3.3 节）最开始的注释行主要用于类 UNIX 系统中（尽管这行注释对 Windows 并没有什么影响），这个注释行必须指示开发者安装的 Python 解释器的完整路径。正如第 3.3 节中介绍的，开发者可能还需要使用 chmod +x hello.py 命令（适用于类 UNIX 系统）让这个脚本可以直接执行（而在 Windows 中，如果开发者安装的 Python 版本在 Windows 注册表中将扩展名.py 与 Python 解释器进行了关联，则可以获得相同的效果，大多数 Python 版本都可以实现这个功能）。现在，开发者可以在任意命令行提示符上运行这个脚本，并且 optparse 将关注于读取开发者传递的命令行选项，并根据这些选项作出适当的反应。例如，在当前目录下包含 hello.py 文件时，类 UNIX 系统下的一个短命令行会话可以按以下方式执行：

```
$ ./hello.py
Hello world!
$ ./hello.py --help
usage: hello.py [options]

options:
  -h, --help          show this help message and exit
  -v, --verbose
  -nNAME, --name=NAME
$ ./hello.py -v --name=Alex
Greetings and salutations, Alex!
$ python hello.py -n
usage: hello.py [options]

hello.py: error: -n option requires an argument
$ python hello.py -nthere
Hello there!
```

最后两个示例显示（当然，假定 Python 解释器在路径 PATH 中），开发者还可以通过将.py 文件显式传递到 Python，并将该脚本的选项和参数放在脚本文件名称的后面来运行这个脚本。

8.11 itertools 模块

itertools 模块提供了许多强大的、高性能的功能模块来建立或操作迭代器对象。感谢迭代器本身的“惰性求值（lazy evaluation）”功能，操作迭代器通常要比操作列表更好一些：迭代器中的项目是根据需要一次产生一个，而列表（或其他序列）中的所有项目必须同时都存在于内存中（迭代器的这种“惰性”方法甚至适合于建立和操作无界的迭代器，而所有列表必须总是包含有限数量的项目）。

本节将详细介绍模块 itertools 最常使用的属性；每个属性都是一个迭代器类型，开发者可以调用这些属性以获得该类型的一个实例。

表 8-10

chain	`chain (*iterables )`
	建立并返回一个迭代器，该迭代器中的项目首先包括来自于传递的第一个可迭代对象中的所有项目，接着是来自于传递的第二个可迭代对象中的所有项目，依此类推，直到传递的最后一个可迭代对象中的所有项目，就像下面的生成器表达式一样：
	`(item for iterable in iterables for item in iterable)`

count	`count(firstval=0)`
	建立并返回一个无界的迭代器，其中的项目是以 firstval 开始的连续整数，就像下面的生成器一样：
	`def count(firstval=0):`
	`  while True:`
	`  yield firstval`
	`  firstval += 1`

cycle	`cycle(iterable)`
	建立并返回一个无界的迭代器，其中的项目都是 iterable 中的项目，在每次循环到末尾之后从头开始不断地重复项目，就像下面的生成器一样：
	`def cycle(iterable):`
	`  buffer = []`
	`  for item in iterable:`
	`  yield item`
	`  buffer.append(item)`
	`  while True:`
	`  for item in buffer: yield item`

ifilter	`ifilter(func,iterable)`
	建立并返回一个迭代器，其中的项目是 iterable 中可以使得 func 的值为 True 的项目，就像下面的生成器表达式一样：
	`(item for item in iterable if func(item))`
	func 可以是任何可以接受单个参数的可调用对象或者是 None。在 func 为 None 时，ifilter 将检查值为 True 的项目，就像下面的生成器表达式一样：
	`(item for item in iterable if item)`

续表

imap	imap(*func*,**iterables*)
	建立并返回一个迭代器，其中的项目是 func 的结果，在调用 func 时，其参数来自于每个 iterables 中的对应项目；在 iterables 中的最短可迭代参数被耗尽时停止操作。就像下面的生成器一样：
	`def imap(func,*iterables):`
	`  next_items = [iter(x).next for x in iterables]`
	`  while True: yield func(*(next() for next in next_items))`

islice	islice(*iterable*[,*start*],*stop*[,*step*])
	建立并返回一个迭代器，其中的项目来自 iterable 中的项目，跳过第一个 start 项目（默认值为 0）直到第 stop 个项目（除 stop 外），每次按 step 步长（默认为 1）前进。所有参数必须都是非负整数，并且 step 必须>0。除了要对参数进行检查和选择之外，就像下面的生成器一样：
	`def islice(iterable,start,stop,step=1):`
	`  en = enumerate(iterable)`
	`  for n, item in en:`
	`  if n>=start: break`
	`  while n<stop:`
	`  yield item`
	`  for x in range(step): n, item = en.next()`

izip	izip(**iterables*) 建立并返回一个迭代器，其中的项目是元组，元组中的每个项目来自于每个 iterables 中的对应项目；在 iterables 中的最短可迭代参数被耗尽时停止操作。就像 imap(tuple,*iterables) 一样。

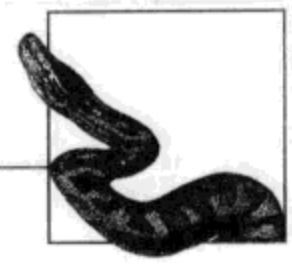
repeat	repeat(*item*[,*times*])
	建立并返回一个迭代器，其中的 times 个项目都是对象 item，就像下面的生成器表达式一样：
	`(item for x in xrange(times))`
	在没有提供 times 时，该迭代器是无界的，包含无限数量的项目，这些项目都是对象 item。就像下面的生成器一样：
	`def repeat_unbounded(item):`
	`  while True: yield item`

tee	tee(*iterable*,*n*=2) 建立并返回一个由 n 个独立迭代器组成的元组，每个迭代器中包含的项目与 iterable 中的项目相同。尽管返回的迭代器相互独立，但是这些迭代器并不独立于 iterable；因此，只要开发者仍要使用返回的任何一个迭代器，则必须避免以任何方式改变对象 iterable（这个警告大致上与 iter(iterable) 的结果相同）。这是 Python 2.4 中新引入的属性。 Python 的在线文档提供了模块 itertools 中包含的以上几个类型的丰富示例和解释，以及本书中没有介绍的 itertools 的其他一些类型。一个令人惊奇

续表

tee	的事实就是 itertools 类型的处理速度。要想举个简单的示例，可以考虑在开发者的计算机上重复某个动作 10 次：
	`for x in itertools.repeat(0, 10): pass`
	itertools 方法要比第二快的处理方法大约快 10%：
	`for x in xrange(10): pass`
	在上面的代码中，如果使用 range 而不是 xrange，则 itertools 方法要比这种方法快几乎两倍。

第 9 章



字符串和正则表达式

Python 在语句、运算符、内置函数、方法和专用模块中广泛支持普通和 Unicode 字符串。本章将在第 9.1 节中介绍字符串对象的方法，在第 9.3 节中介绍字符串的格式化，在第 9.2 节中介绍 string 模块，在第 9.4 节中介绍 pprint 模块，并在第 9.5 节中介绍 repr 模块，还将在第 9.6 节中介绍与 Unicode 字符串有关的内容。

正则表达式（Regular Expression）可以让开发者指定模式字符串并进行搜索和替换操作。正则表达式并不容易熟练掌握，但是正则表达式是用来处理文本的强大工具。Python 通过内置的 re 模块提供了丰富的正则表达式功能，参见第 9.7 节中的介绍。

9.1 字符串对象的方法

普通和 Unicode 字符串都是不可变序列，参见第 4.6 节。所有不可变序列的操作（重复、串联、索引和切片）都可以应用于字符串。字符串对象 s 还可以提供几个不可变方法，参见本节的介绍。除非另有说明，每个方法在 s 是一个普通字符串时将返回一个普通字符串，或者在 s 是一个 Unicode 字符串将返回一个 Unicode 字符串。像“字母”和“空白字符”之类的名词术语都用来表示 string 模块的对应属性，参见第 9.2 节和第 9.2 节（国家/地区）。

表 9-1

capitalize	`s.capitalize()` 返回 s 的一个副本，这个副本的第一个字符（也就是字母）是大写字母，而其他字符都是小写字母。
center	`s.center(n, fillchar=' ')` 返回一个长度为 max(len(s),n)的字符串，该字符串的最中间包含 s 的一个副本，前后包含的是相同数量的字符 fillchar 的副本（例如，'ciao'.center(2)的结果是‘ciao’，而 'x'.center(4,'_')的结果是 ‘_x_ _’。

续表

count	*s*.count(*sub*,*start*=0,*end*=sys.maxint) 返回 s[start:end]中子字符串 sub 不相互重叠的出现次数。
decode	*s*.decode(*codec*=None,*errors*='strict') 使用给定的编解码器和错误处理方式返回从 s 获得的一个字符串（通常是 Unicode 字符串）。参见第 9.6 节以了解详细介绍。
encode	*s*.encode(*codec*=None,*errors*='strict') 使用给定的编解码器和错误处理方式返回从 s 中获得的一个普通字符串。参见第 9.6 节以了解详细介绍。
endswith	*s*.endswith(*suffix*,*start*=0,*end*=sys.maxint) 在 s[start:end] 以 suffix 结束时，返回 True；否则，返回 False。
expandtabs	*s*.expandtabs(*tabsize*=8) 返回 s 的一个副本，其中的每个制表符被更改为一个或多个空格，制表位在每 tabsize 个字符之后结束。
find	*s*.find(*sub*,*start*=0,*end*=sys.maxint) 返回在 s 中找到的子字符串 sub 的最小索引号，这里 sub 要完整地包含在 s[start:end]中。例如，'banana'.find('na')的结果是 2，就像'banana'.find('na',1)一样，而'banana'.find('na',3)的结果是 4，就像'banana'.find('na',-2)一样。如果没有找到 sub，则 find 将返回-1。
index	*s*.index(*sub*,*start*=0,*end*=sys.maxint) 与 find 类似，但是在没有找到 sub 将引发 ValueError。
isalnum	*s*.isalnum() 在 len(s)大于 0，并且 s 中的所有字符都是字母或十进制数字时，返回 True。在 s 为空，或者在 s 中至少有一个字符既不是字母又不是十进制数字时，isalnum 将返回 False。
isalpha	*s*.isalpha() 在 len(s)大于 0，并且 s 中的所有字符都是字母时，返回 True。在 s 为空，或者在 s 中至少有一个字符不是字母时，isalpha 将返回 False。
isdigit	*s*.isdigit() 在 len(s)大于 0，并且 s 中的所有字符都是数字时，返回 True。在 s 为空，或者在 s 中至少有一个字符不是数字时，isdigit 将返回 False。
islower	*s*.islower() 在 s 中的所有字母都是小写时，返回 True。在 s 不包含任何字母，或者在 s 中至少有一个字母是大写时，islower 将返回 False。
isspace	*s*.isspace() 在 len(s)大于 0，并且 s 中的所有字符都是空白字符时，返回 True。在 s 为空，或者在 s 中至少有一个字符不是空白字符时，isspace 将返回 False。
istitle	*s*.istitle() 在 s 中的单词都是首字母大写时返回 True，首字母大写就是每个连续字符序列的开始是一个大写字母，其他字母都是小写字母（例如，'King Lear'.istitle()的结果是 True）。在 s 不包含任何字母，或者在 s 中至少有一个字母违反了首字母大写的条件时，istitle 将返回 False（例如，'1900'.istitle() 和 'troilus and Cressida'.istitle() 的结果都是 False）。
isupper	*s*.isupper() 在 s 中的所有字母都是大写字母时，返回 True。在 s 不包含任何字母，或者在 s 中至少有一个字母为小写字母时，isupper 将返回 False。

续表

join	*s*.join(*seq*) 返回包含串联的 seq 中的项目的字符串，seq 必须是一个序列或者其他项目都是字符串的可迭代对象，并且返回的字符串将在 seq 的每个项目对之间插入 s 的一个副本（例如，''.join(str(x) for x in range(7))的结果是‘0123456’，而'x'.join('aeiou') 的结果是‘axexixoxu’）。
ljust	*s*.ljust(*n*,*fillchar*=' ') 返回一个长度为 max(len(*s*),*n*)的字符串，这个字符串以 s 的一个副本开始，后面是以字符 fillchar 填充的零个或多个拖尾副本。
lower	*s*.lower() 返回 s 的一个副本，其中存在的所有字母都被转换成小写字母。
lstrip	*s*.lstrip(*x*=string.whitespace) 返回 s 的一个副本，删除了 s 中可以在字符串 x 中找到的相同前导字符。
replace	*s*.replace(*old*,*new*,*maxsplit*=sys.maxint) 返回 s 的一个副本，将 s 中的子字符串 old 的前 maxsplit（或者也可以更少）个不重叠的出现替换为字符串 new（例如，'banana'.replace('a','e',2) 的结果是‘benena’）。
rfind	*s*.rfind(*sub*,*start*=0,*end*=sys.maxint) 返回在 s 中可以找到的子字符串 sub 的最大索引，这里 sub 要完全包含在 s[start:end]中。如果没有找到 sub，rfind 将返回−1。
rindex	*s*.rindex(*sub*,*start*=0,*end*=sys.maxint) 与 rfind 相似，但是如果没有找到 sub，则引发 ValueError。
rjust	*s*.rjust(*n*,*fillchar*=' ') 返回一个长度为 max(len(s),n)的字符串，该字符串的末尾包含 s 的一个副本，前面是以字符串 fillchar 填充的零个或多个前导副本。
rstrip	*s*.rstrip(*x*=string.whitespace) 返回 s 的一个副本，删除了 s 中可以在字符串 x 中找到的相同拖尾字符。

split	*s*.split(*sep*=None,*maxsplit*=sys.maxint)
	返回一个最多包含 maxsplit+1 个字符串的列表 L。L 中的每个项目都来自于 s 中的一个“单词”或“词组”，字符串 sep 用来分割这些单词。在 s 中具有多于 maxsplit 个的 sep 单词时，L 的最后一个项目是 s 的一个子字符串，前面是 maxsplit 个 sep 单词。在 sep 为 None 时，则任何由空白字符组成的字符串用来分割单词（例如，'four score and seven years ago'.split(None,3) 的结果是['four','score','and','seven years ago']）。
	请注意，使用 None（任何由空白字符组成的字符串都是一个分隔符）进行分割和使用 ''（每个单独的空白字符是一个分隔符，不包括像制表符和换行符这样的空白字符，也不包括字符串）进行分割是不同的。例如：
	`>>> x = 'a  b'  # a 和 b 之间包含两个空格`
	`>>> x.split()`
	`['a', 'b']`
	`>>> x.split(' ')`
	`['a', '', 'b']`
	在第一种情况下，中间的双空格字符串是单个分隔符；而在第二种情况下，每个单独的空格都是一个分隔符，因此，返回的结果中包含一个空白字符串。

续表

splitlines	*s*.splitlines(*keepends*=False) 与 s.split('\n')相似。但是，在 keepends 为 True 时，拖尾的‘\n’被包含在结果列表的每个项目中。
startswith	*s*.startswith(*prefix*,*start*=0,*end*=sys.maxint) 在 s[start:end]以前缀 prefix 开始时，返回 True；否则，返回 False。
strip	*s*.strip(*x*=string.whitespace) 返回 s 的一个副本，删除了 s 中可以在字符串 x 中找到的相同前导字符和拖尾字符。
swapcase	*s*.swapcase() 返回 s 的一个副本，将 s 中的所有大写字母都转换为小写字母，或者相反。
title	*s*.title() 返回 s 的一个副本，将 s 转换为首字母大写：每个连续字母序列的开始是一个大写字母，而所有其他的字母都是小写字母。
translate	*s*.translate(*table*, *deletechars*='') 在 s 是一个普通字符串时 *s*.translate(*table*) 在 s 是一个 unicode 字符串时 在 s 是一个普通字符串时，返回 s 的一个副本，其中所有在字符串 deletechars 中出现过的字符都被删除，而剩下的字符都通过翻译表 table 进行映射。table 必须是一个长度为 256 的普通字符串，通常是使用 string.maketrans 函数建立的，参见第 9.2 节中对 maketrans 函数的介绍。 在 s 是一个 Unicode 字符串时，将返回 s 的一个副本，其中 table 中能找到的字符都被翻译或删除了。table 是一个键为 Unicode 序数的字典，值为 Unicode 序数、Unicode 字符串或 None（要删除）的字典，例如： u'banna'.translate({ord('a'):None,ord('n'):u'ze'})的结果就是 u'bzeze'。
upper	*s*.upper() 返回 s 的一个副本，将 s 中的所有字母转换为大写字母。

9.2 string 模块

string 模块提供了一些复制字符串对象的每个方法的函数，参见第 9.1 节，其中的字符串对象可以作为 string 模块提供的函数的第一个参数。string 模块还提供了以下几个有用的普通字符串的属性。

ascii_letters

字符串 ascii_lowercase+ascii_uppercase。

ascii_lowercase

字符串‘abcdefghijklmnopqrstuvwxyz’。

ascii_uppercase

字符串‘ABCDEFGHIJKLMNOPQRSTUVWXYZ’。

digits

字符串‘0123456789’。

hexdigits

字符串‘0123456789abcdefABCDEF’。

letters

字符串 lowercase+uppercase。

lowercase

包含所有被确认是小写字母的字符的字符串：至少包括‘abcdefghijklmnopqrstuvwxyz’，根据当前使用的本地语言环境，可能还包括更多字符（例如，重音符）。

octdigits

字符串‘01234567’。

punctuation

字符串‘!"#$%&\'( )*+, -./:;<=>?@[\\]^_'{|}～’（也就是，所有在‘C’语言环境下被确认是标点符号的 ASCII 字符；不取决于当前使用的本地语言环境）。

printable

由被确认为可打印的字符组成的字符串（也就是，数字、字母、标点符号和空白字符）。

uppercase

包含所有被确认为大写字母的字符的字符串：至少包括‘ABCDEFGHIJKLNOPQRSTUVWXYZ’，根据当前使用的本地语言环境，可能还包括更多字符（例如，重音符）。

whitespace

包含所有被确认为空白字符的字符的字符串：至少包括空格、制表符、换行符、回车符，根据当前使用的本地语言环境，还可能包含更多字符（例如，特定的控制字符）。

开发者必须重新绑定这些属性，因为 Python 库的其他部分可能依赖于这些属性，而且没有定义重新绑定这些属性时的效果。

string 模块还提供了 Template 类，参见第 9.3 节。

本地语言环境敏感性

参见第 10.12 节中对 locale 模块的介绍。本地语言环境设置会影响 string 模块的某些属

性（letters、lowercase、uppercase 和 whitespace）。通过这些属性，本地语言环境设置还将影响 string 模块的函数和按字母来处理字符分类的普通字符串对象的方法，以及大小写字母之间的转换，比如 capitalize、isalnum 和 isalpha。Unicode 字符串的对应方法不会受到本地语言环境设置的影响。

maketrans 函数

参见第 9.1 节中对 translate 的介绍，普通字符串的 translate 方法将一个长度为 256 的普通字符串作为其第一个参数，并将其用作一个翻译表。最简单的建立翻译表的方法就是使用 string 方法提供的 maketrans 函数。

表 9-2

maketrans	`maketrans(from,onto)`
	返回一个翻译表——也就是，一个长度为 256 的普通字符串，提供了从按升序排列的 ASCII 字符到另一种字符集合的映射。from 和 onto 都必须是普通字符串，并且 len(from)等于 len(onto)。from 中的每个字符都被映射到 onto 中对应位置的字符上。from 中没有列出的每个字符都映射为其本身。要想获得一个“标识(identity)”表，可以调用 maketrans("","")。
	使用 translate 字符串方法，开发者可以删除字符，也可以翻译字符。在只是为了删除字符而使用 translate 方法时，传递给 translate 的第一个参数必须是标识表，下面是使用 maketrans 方法和字符串方法 translate 删除元音字母的一个示例：
	`import string`
	`identity = string.maketrans('','')`
	`print 'some string'.translate(identity,'aeiou')`
	`# 打印: sm strng`
	等同于上面这段代码的 Unicode 代码是：
	`no_vowels = dict.fromkeys(ord(x) for x in 'aeiou')`
	`print u'some string'.translate(no_vowels)`
	`# 打印: sm strng`
	下面是将所有其他元音字母变成字母 a，并删除字母 s 的示例：
	`intoas = string.maketrans('eiou','aaaa')`
	`print 'some string'.translate(intoas)`
	`# 打印: sama strang`
	`print 'some string'.translate(intoas,'s')`
	`# 打印: ama trang`
	等同于上面这段代码的 Unicode 代码是：
	`intoas = dict.fromkeys((ord(x) for x in 'eiou'), 'a')`
	`print u'some string'.translate(intoas)`
	`# 打印: sama strang`
	`intoas_nos = dict(intoas, s='None')`
	`print u'some string'.translate(intoas_nos)`
	`# 打印: ama trang`

9.3 字符串格式化

在 Python 中，字符串格式化表达式具有以下语法：

```
format % values
```

其中 format 是一个包含格式指定符的普通或 Unicode 字符串，values 是一个元组或目录中的任何一个单独对象或对象集合。Python 的字符串格式化运算符具有与 C 语言的 printf 大致相同的功能集，并且以类似的方式操作。每个格式指定符都是 format 的一个子字符串，该子字符串以百分号（%）开始，以表 9-3 中显示的任何一个转换字符结束。

表 9-3 字符串格式化转换字符

字 符	输 出 格 式	注 释
d, i	带符号整数	值必须是数字
u	无符号整数	值必须是数字
o	无符号八进制整数	值必须是数字
x	无符号十六进制整数（小写字母）	值必须是数字
X	无符号十六进制整数（大写字母）	值必须是数字
e	以指数形式表示的浮点型值（小写 e 表示指数）	值必须是数字
E	以指数形式表示的浮点型值（大写 e 表示指数）	值必须是数字
f, F	以小数形式表示的浮点型值	值必须是数字
g, G	在 exp >=4 或者<精度时，与 e 或 E 一样；否则，与 f 或 F 一样	exp 是被转换的数字的指数
c	单个字符	值可以是整数或单个字符的字符串
r	字符串	使用 repr 转换任意值
s	字符串	使用 str 转换任意值
%	字面常量%字符	不使用任何值

在%和转换字符之间，可以指定一些可选的修改符，下面将介绍这些修改符。

格式化表达式的结果是 format 的一个字符串副本，其中的每个格式指定符将被替换为根据该指定符转换为字符串的 values 中的对应项目。下面是几个简单的示例：

```
x = 42
y = 3.14
z = "george"
print 'result = %d' % x                    # 打印: result = 42
print 'answers are: %d %f' % (x,y)         # 打印: answers are: 42 3.14
print 'hello %s' % z                       # 打印: hello george
```

格式指定符语法

格式指定符可以包含许多修改符，这些修改符用来控制如何将 values 中的对应项目转换为一个字符串。格式指定符的组件按顺序分别是：

- 强制的前导%字符，标记指定符的开始；
- 在圆括号中包含可选项目名称（例如，(name)）；
- 零个或更多个可选转换标记：

  #

  　　使用另一种格式进行转换（如果这种类型存在另一种格式）

  0

  　　转换是用 0 补足的

  –

  　　转换是左对齐的

  （一个空格）

  　　在正数前面放置一个空格

  +

  　　在任何数字转换之前放置数值符号（+或–）

- 转换的可选最小宽度：一个或多个数字，或者一个星号（*），表示这个宽度是从 values 中的下一个项目中取得的；
- 转换的可选精确度：点号（.）后面带零个或多个数字，或者带一个星号（*），表示这个精确度是从 values 中的下一个项目中取得的；
- 表 9-1 中的强制转换类型。

通常必须在 format 的所有格式指定符中给定项目的名称，或者不在任何指示符中给定项目的名称。在给定了项目名称时，values 必须是一个映射（通常是一个命名空间的字典，例如，vars( )），并且每个项目名称都是 values 中的一个键。换句话讲，每个格式指定符对应于 values 中以该指定符的项目名称作为键的项目。在给定了项目名称时，开发者不能在任意格式指定符中使用*。

在不提供项目名称时，values 必须是一个元组；在只有一个项目时，values 可能是该项目本身（除了 tuple 之外的任何项目），而不是一个单独的元组。每个格式指定符按位置对应到 values 中的一个项目上，并且 values 必须与 format 中包含的指定符具有相同数量的项目（加上使用 * 为每个宽度或精度给定的一个额外项目）。在使用 * 给定一

个指定符的宽度或精度组件时，* 将用掉 values 中的一个项目，这个项目必须是一个整数，这个整数可以被看作是字符的数量，用来表示转换的宽度或精度转换。

常用字符串格式化习惯用法

fromat 中包含几个%s，values 是一个元组，并且包含与 format 中包含%s 的个数完全相同数量的项目，这是非常常见的。其结果将产生 format 的一个副本，其中的每个%s 将被替换为将 str 应用于 values 中的对应项目而生成的字符串。例如：

```
'%s+%s is %s'%(23,45,23+45)                    # 结果是: '23+45 is 68'
```

开发者可以将%s 看作是一种快速和简明的方法，用来将几个被转换为字符串形式的值组合成一个较长的字符串。例如：

```
oneway = 'x' + str(j) + 'y' + str(j) + 'z'
another = 'x%sy%sz' % (j, j)
```

在执行这段代码之后，变量 oneway 和 another 将会相等，但是通过字符串格式化实现的 another 的计算要更快一些。哪种方法更清楚和更简单只是一个习惯的问题。本书鼓励开发者习惯于使用字符串格式化用法，而且这种方法很快会看起来更简单和清楚。

除了%s，其他适合使用的常用格式指定符是那些用于格式化浮点型值的指定符：%f 用于小数的格式化、%e 用于指数的格式化，而%g 用于小数或指数的格式化，如何选择取决于数字的量级。在格式化浮点型值时，通常需要指定宽度和/或精度修改符。宽度修改符是%右边的一个数字，给出了结果转换的最小宽度；如果开发者要格式化一个表格以按固定宽度字体显示表格中的值，通常需要使用宽度修改符。精度修改符是在转换类型字母之前加上一个数字和一个句点（.），通常，开发者使用精度修改符就是为了让一个数字显示固定数量的小数位，这样可以避免多余的精度造成一些误解，也可以避免浪费显示空间。例如：

```
'%.2f'%(1/3.0)                                 # 结果是: '0.33'
'%s'%(1/3.0)                                   # 结果是: '0.333333333333'
```

开发者不能使用%s 指定在小数点后显示多少位数字。在只需要几个数字就能精确表示数值结果时，重要的是避免高精度产生的错误印象。显示多个数字可能会误导人们通过看到的结果就相信这些结果要比其本身的值更精确。

模板字符串

Python 2.4 引入了 string.Template 类，在某些简单情况下，这个类比字符串的%操作符提供了更便利的格式化工具。

表 9-4

Template	`class Template(`*template*`)` 建立并返回 Template 的一个新实例 t，其只读属性 t.template 将被设置为字符串参数 template。 Template 的实例 t 提供了以下两种方法。

续表

safe_substitute	*t*.safe_substitute(*mapping*,**k) 返回 t.template 的一个字符串副本，其中： • 出现的每个$$都被更改为单个$； • 出现的每个$identifier 或${identifier }（其中 identifier 是任意一个有效的 Python 标识符，该标识符是 mapping 中的键和/或 k 中的一个参数名称）将被更改为 mapping.update(**k)[identifier]。 出现的$identifier or ${identifier }（其中的 identifier 不是有效的 Python 标识符，或者不是 mapping 中的键，也不是 k 中的一个参数名称）只是从 t.template 被复制到 safe_substitute 的字符串结果中。
substitute	*t*.substitute(*mapping*,**k) 与 safe_substitute 类似，除了出现的几个$identifier 或 ${identifier }（其中的 identifier 不是有效的 Python 标识符，或者不是 mapping 中的键，也不是 k 中的一个参数名称）之外，substitute 将引发一个 KeyError 异常。 要想获得有关 string.Template 的更高级（但是极少需要）用法，参见 http://docs.python.org/lib/module-string.html 了解所有详细信息。

文本包装和填充

textwrap 模块提供了一个类和几个函数以格式化字符串，将字符串分成给定最大长度的行。要想调整填充和包装，可以实例化 textwrap 模块提供的 TextWrapper 类，并应用详细控制。不过，在大多数时候，使用 textwrap 模块提供的两个主要函数就足够了。

表 9-5

wrap	wrap(*s*,*width* =70) 返回一个字符串列表（不带用于结束的换行符），每个字符串都不长于 width 个字符，并且这个字符串列表（后面填充了一些空白符号）等于 s。wrap 还提供了其他一些命名参数（等于类 TextWrapper 的实例的属性）；要想了解这些高级用法，参见 http://docs.python.org/lib/module-textwrap.html。
fill	fill(*s*,*width* =70) 返回单个多行字符串，完全相当于 '\n'.join(wrap(*s*,*width*))。

9.4 pprint 模块

pprint 模块可以非常漂亮地打印复杂的数据结构，其格式化要比内置函数 repr（参见第 8.2 节中介绍的 repr）提供的方式更有可读性。要想调整格式化，可以实例化 pprint 模块提供的 PrettyPrinter 类，并应用详细的控制，pprint 模块提供的辅助函数对此也有帮助。不过，在大多数时候，使用 pprint 模块提供的两个主要函数就足够了。

表 9-6

pformat	pformat(*obj*) 返回一个字符串以表示 obj 的漂亮打印输出。
pprint	pprint(*obj*,*stream*=sys.stdout) 将 obj 的漂亮打印输出到文件对象 stream 中，并加上一个用于结束的换行符。 下面的语句是相同的： print pprint.pformat(x) pprint.pprint(x) 在许多情况下，这两种方法中的任何一个都与 print x 大致相同，比如在 x 的字符串表示适合放在一行文本中时。不过，在像 x=range(30)这样的情况下，print x 会将 x 显示为两行文本，在任意位置断开，而用 pprint 模块会将 x 显示为超过 30 行，每个项目一行。在开发者更喜欢看到模块的特殊显示效果，而不是一种普通字符串表示时，可以使用 pprint 模块。

9.5 repr 模块

repr 模块提供了一个可以替代内置函数 repr（参见下面对 repr 的介绍）的函数，对表示字符串的长度进行了限制。要想调整长度限制，可以使用 repr 模块实例化或继承为 Repr 类的子类，并应用详细的控制。不过，在大多数时候，使用 repr 模块提供的主要函数就足够了。

表 9-7

repr	repr(*obj*) 返回一个字符串以表示 obj，并对其长度进行适当地限制。

9.6 Unicode

在开发者将一个普通字符串传递到一个期望得到 Unicode 的函数时，普通字符串将被显式（使用 unicode 内置）或隐式转换为 Unicode 字符串。在任何一种情况下，这种转换操作是通过一个被称为编解码器（用于编码-解码）的辅助对象来完成的。编解码器还可以将 Unicode 字符串转换为普通字符串，不管是显式地使用 Uncode 字符串的 encode 方法，还是隐式转换。

要想标识一个编解码器，可以将该编解码器的名称传递给 unicode 或 encode。在开发者不传递编解码器的名称，并进行隐式转换时，Python 将使用默认的编码方式，通常是“ascii”。开发者可以在 Python 程序的启动阶段更改默认的编码方式，参见第 13.5 节；还可以参见第 8.3 节。但是，这样的更改对于大多数“重要的”Python 代码而言并不是一个好主意：这种方式太容易妨碍标准 Python 库或第三方模块中的代码，这些代码可能会期望得到普通的“ascii”编码。

每个转换都有一个参数 errors，一个字符串指定了如何处理转换错误。默认的字符串就是‘strict’，表示任何错误都会引发一个异常。在 errors 是‘replace’ 时，转换操作将把每个导致错误的字符替换为普通字符串结果中的‘?’，替换为 Unicode 结果中的 u'\ufffd'。在 errors 为‘ignore’时，转换操作将跳过导致错误的字符。在 errors 是‘xmlcharrefreplace’时，转换操作将把每个导致错误的字符在结果中替换为这个字符的 XML 字符引用表示。开发者还可以编写自己的函数以实现一种转换错误处理机制，并通过调用 codecs.register_error 将该函数以一个适当的名称进行注册。

codecs 模块

编解码器名称到编解码器对象的映射是通过 codecs 模块来处理的。这个模块还允许用户开发自己的编解码器对象，并注册这些对象，这样就可以通过名称来查找这些对象了，就像使用内置编解码器一样。codecs 模块还可以让开发者显式查找任何编解码器，获得该编解码器用来编码和解码的函数，以及工厂函数，这些函数可以用来包装类文件对象。codecs 模块的这些高级功能是极少使用的，本书接下来也不对这些功能进行介绍。

codecs 模块，加上标准 Python 库的 encodings 包，可以提供一些内置编解码器，这些内置编解码器对于 Python 开发者处理国际化问题非常有用。Python 本身附带了 100 多个编解码器；读者可以在 http://docs.python.org/lib/standard-encodings.html 上看到这些编解码器的列表，以及每个编解码器的简要说明。提供的任意编解码器可以通过 sitecustomize 模块被安装为 site-wide 默认值，但是，首选的用法就是在任何时候需要在普通和 Unicode 字符串之间进行显式转换时，总是通过名称来指定编解码器。在默认情况下安装的编解码器是‘ascii’，该编解码器只能接受代码为 0～127 之间的字符，也就是对几乎所有的编码方式都通用的 7 比特值的“美国信息交换标准码”（ASCII）。一种流行的编解码器是‘latin-1’，这是 ISO 8859-1 编码方式的一种快速和内置实现，为西欧语言所需的所有特殊字符提供了一种每个字符一个字节的编码方式。

codecs 模块还为大多数 ISO 8859 编码方式提供了使用 Python 实现的编解码器，这些编解码器的名称从‘iso8859-1’～‘iso8859-15’。只有在 Windows 系统中，名为‘mbcs’的编解码器包装了该平台的多字节字符集转换程序。许多编解码器还专门支持亚洲语言。codecs 模块还提供了几种标准代码页（名称为‘cp038’～‘cp1258’的编解码器），Mac 专用编码方式（名称为‘mac-cyrillic’～‘mac-turkish’的编解码器），和 Unicode 标准编码方式‘utf-8’和‘utf-16’（后者还具有特殊的大端模式（big-endian）和小端模式（little-endian）变体：‘utf-16-be’和‘utf-16-le’）。在使用 UTF-16 时，codecs 模块还提供了属性 BOE_BE 和 BOE_LE，分别用于大端模式和小端模式计算机的字节序标记，还提供了 BOM，用于当前平台的字节序标记。

codecs 模块还提供了下面这个函数，让开发者可以注册自己的转换错误处理函数。

表 9-8

register_error	register_error(*name*,*func*) name 必须是一个字符串。func 必须可以对一个参数 e 进行调用，参数 e 是 UnicodeDecodeError 异常的一个实例，func 还必须返回一个包含两个项目的元组：要被插入到转换的字符串结果中的 Unicode 字符串和用于继续进行转换的索引（后者通常是 e.end）。这个函数体可以使用 e.encoding（这个转换操作的编解码器的名称）和 e.object[e.start:e.end]（导致转换错误的子字符串）。

codecs 模块还提供了两个函数以简单处理使用不同编码方式的文件。

表 9-9

EncodedFile	EncodedFile(*file*,*datacodec*,*filecodec*=None,*errors*='strict')
	包装类文件对象 file，返回一个类文件对象 ef，该对象可以隐式和明显地将给定编码方式应用到从文件读取或写入到文件的所有数据上。在向 ef 写入一个字符串 s 时，ef 将首先使用 datacodec 指定的编解码器对 s 进行解码，然后使用 filecodec 指定的编解码器对解码的结果进行编码，并将其写入到 file 中。在从 ef 中读取一个字符串时，ef 将首先应用 filecodec，然后应用 datacodec。在 filecodec 为 None 时，ef 将使用 datacodec 执行读取或写入操作中的编码和解码这两个步骤。
	例如，如果开发者想要将使用 latin-1 编码的字符串写入到 sys.stdout 中，并且输出编码方式为 utf-8 的字符串，可以使用以下代码：
	`import sys, codecs`
	`sys.stdout = codecs.EncodedFile(sys.stdout,'latin-1', utf-8')`

open	open(*filename*,*mode*='rb',*encoding*=None,*errors*='strict',*buffering*=1)
	使用内置模块 open（参见第 10.3 节）可以提供一个类文件对象，该对象可以从 Python 客户节点接受 Unicode 字符串和/或向 Python 客户节点提供 Unicode 字符串，而底层文件可以是使用 Unicode（在 encoding 为 None 时）或者使用 encoding 指定的编解码器编码的文件。例如，如果开发者想要将 Unicode 字符串写入到文件 uni.txt 中，并让该字符串在文件中被隐式编码为 latin-1，同时，对于任何不能以 latin-1 编码的字符，将其替换为‘?’，可以使用以下代码：
	`import codecs`
	`flout = codecs.open('uni.txt','w','latin-1','replace')`
	
	`# 现在可以直接向 flout 写入 Unicode 字符串了`
	`flout.write(u'élève')`
	`flout.close()`

unicodedata 模块

unicodedata 模块提供了一种访问 Unicode 字符数据库（Unicode Charater Database）的简单方式。给定任意 Unicode 字符，可以使用 unicodedata 模块提供的函数获得该字符

的 Unicode 类别、正式名称（如果有的话），以及其他一些外来信息。开发者还可以查找与给定正式名称对应的 Unicode 字符（如果有）。这种高级工具是极少需要的，因此本书并没有进一步介绍这些工具。

9.7 正则表达式和 re 模块

正则表达式（regular expression，RE）是一个用来表示一种模式的字符串。使用正则表达式功能，开发者可以根据定义的模式来检查任意字符串，并查看该字符串中是否有任意一部分匹配这种模式。

re 模块提供了 Python 的正则表达式功能。编译函数可以通过模式字符串和可选标记建立正则表达式对象。正则表达式对象的方法可以在字符串中查找匹配该正则表达式的内容，或者执行替换操作。re 模块还提供了一些等同于正则表达式的方法的函数，区别在于这些函数将使用正则表达式的模式字符串作为其第一个参数。

熟练掌握正则表达式是比较困难的，本书并不打算重点介绍正则表达式；本书只介绍在 Python 中使用正则表达式的一些方法。要想了解有关正则表达式的相关知识，推荐阅读 O'Reilly 出版，Jeffrey Friedl 编著的 *Mastering Regular Expressions* 一书。Friedl 的这本书提供了对正则表达式的从初级指南到高级应用的完整介绍。还可以在线查找有关正则表达式的许多指南和参考资料。

模式字符串语法

模式字符串可以按照以下特定语法表示一个正则表达式。

- 字母和数字字符代表其本身。这种正则表达式的模式就是由字母和数字组成的字符串匹配相同的普通字符串。
- 许多前面加上了反斜线符号（\）的字母和数字字符在模式中具有特殊的意义。
- 标点符号的使用正好相反：在被转义时可以匹配其自身，在没有被转义时具有特殊的意义。
- 使用重复的反斜线（也就是，模式\\）可以匹配反斜线字符。

因为正则表达式的模式通常包含反斜线符号，开发者通常可以使用原生字符串语法（参见第 4.2 节中介绍的字符串）来指定这些反斜线符号。模式元素（例如，r'\t' 等于非原生字符串字面常量 '\\t'）可以与对应的特殊字符（例如，制表符 '\t'）匹配。因此，开发者甚至可以在需要一个字面常量与某些这样的特殊字符匹配时使用原生字符串语法。

表 9-10 列出了正则表达式模式语法中的特殊元素。在开发者使用可选标记和模式字符串来建立正则表达式对象时，某些元素的确切含义可能会改变。本书的 9.7 节介绍了这

些可选标记。

表 9-10　正则表达式模式语法

元　素	含　义
.	匹配除\n 之外的任何字符（如果使用了 DOTALL 可选标记，还可以匹配\n）
^	匹配字符串的起始部分（如果使用了 MULTILINE 可选标记，还可以匹配\n 之后的部分）
$	匹配字符串的结束部分（如果使用了 MULTILINE 可选标记，还可以匹配\n 之前的部分）
*	匹配前面的正则表达式的零个或多个匹配项目；贪婪匹配（尽可能多地匹配）
+	匹配前面的正则表达式的一个或多个匹配项目；贪婪匹配（尽可能多地匹配）
?	匹配前面的正则表达式的零个或一个匹配项目；贪婪匹配（只匹配一个）
*?, +?, ??	*、+和？的非贪婪匹配版本（尽可能少地匹配）
{*m*,*n*}	匹配前面的正则表达式的 m 到 n 个匹配项目（贪婪匹配）
{*m*,*n*}?	匹配前面的正则表达式的 m 到 n 个匹配项目（非贪婪匹配）
[...]	匹配方括号中包含的字符集中的任意一个
|	匹配前面的表达式或后面的表达式
(...)	匹配圆括号中的正则表达式并指定一个组
(?iLmsux)	设置可选标记的另一种方法；对匹配没有影响
(?:...)	与(...)类似，但是不指定一个组
(?P<*id*>...)	与(...)类似，而且该组还将获得名称 id
(?P=*id*)	匹配前面按组名 id 匹配的任何匹配项目
(?#...)	圆括号中的内容只是注释；对匹配没有影响
(?=...)	向前看断言：如果正则表达式...匹配接下来的内容，则表示匹配，但是不消耗字符串的任何部分
(?!...)	相反的向前看断言：如果正则表达式...不匹配接下来的内容，则表示匹配，并且不消耗字符串的任何部分
(?<=...)	向后看断言：如果有一个以当前位置为结尾的项目匹配正则表达式...，则表示匹配（...必须匹配固定长度）
(?<!...)	相反的向后看断言：如果没有以当前位置为结尾的项目匹配正则表达式...，则表示匹配（...必须匹配固定长度）
number	按照编号 number 匹配以前匹配的任何项目（组是从 1 到 99 自动编号的）
\A	匹配一个空白字符串，但是只在整个字符串的起始位置
\b	匹配一个空白字符串，但是只在一个单词的起始和结束位置（由字母和数字组成的最大序列；参见\w）
\B	匹配一个空白字符串，但是不在一个单词的起始和结束位置

续表

元　素	含　义
\d	匹配一个数字，也就是集合[0-9]中的数字
\D	匹配一个非数字，也就是集合[^0-9]中的字符
\s	匹配一个空白字符，也就是集合[\t\n\r\f\v]中的字符
\S	匹配一个非空字符，也就是集合[^\t\n\r\f\v]中的字符
\w	匹配一个字母或数字字符；除非设置了 LOCALE 或 UNICODE，也就是集合[a-zA-Z0-9_]中的字符
\W	匹配一个非字母和数字字符，与\w 相反
\Z	匹配一个空白字符串，但是只在整个字符串的末尾匹配
\\	匹配一个反斜线字符

常用正则表达式习惯用法

作为正则表达式的模式字符串的子字符串，‘.*’表示“任意字符的任意的重复次数（零次或多次）”。换句话讲，‘.*’匹配一个目标字符串的任意子字符串，包括空白子字符串。‘.+’与此类似，但是只匹配非空子字符串。例如：

```
'pre.*post'
```

匹配一个包含子字符串‘pre’，及其后面带一个子字符串‘post’的字符串，即使后面的子字符串与前面的子字符串直接相连（例如，匹配‘prepost’和‘pre23post’）。另一方面：

```
'pre.+post'
```

只有在‘pre’和‘post’不是直接相连时才匹配（例如，匹配‘pre23post’，但是不匹配‘prepost’）。但是，以上这两种模式都匹配在‘post’之后还有字符的字符串。要想限制模式只匹配以‘post’结束的字符串，可以在模式的后面加上\Z。例如：

```
r'pre.*post\Z'
```

匹配‘prepost’，但是不匹配‘preposterous’。请注意，由于该模式中包含了反斜线符号 \ ，开发者需要使用原生字符串语法（或者使用双反斜线 \\ 来转义反斜线符号 \）以表述这个模式。建议对所有正则表达式模式字面常量使用原生字符串语法，这样可以确保不会忘记对反斜线符号执行转义操作。

正则表达式模式中另外一个经常使用的元素是 \b，该元素用来匹配一个单词的边界。如果开发者只想要全词匹配单词‘his’，而不是在‘this’和‘history’这样的单词中出现的子字符串，其正则表达式模式如下：

```
r'\bhis\b'
```

也就是在单词‘his’之前和之后都匹配单词的边界。要想匹配任何以‘her’起始的单

词的开始部分，比如‘her’本身，还有‘hermetic’，但是不匹配任何其他位置中简单包含‘her’的情况，比如，‘either’或‘there’，可以使用：

```
r'\bher'
```

也就是在单词‘her’之前匹配单词的边界，但是之后不用匹配。要想匹配任何以‘its’结束的单词的末尾部分，比如‘its’本身，还有‘fits’，但是不匹配任何其他位置中简单包含‘its’的情况，比如，‘itsy’或‘jujitsu’，可以使用：

```
r'its\b'
```

也就是在单词‘its’之后匹配单词的边界，但是之前不用匹配。要想匹配这种限制下的完整单词，而不是只匹配以这些单词的开始或者结束部分，可以添加一个模式元素\w* 以匹配零个或多个单词字符。要想匹配任意以‘her’开始的完整单词，可以使用：

```
r'\bher\w*'
```

要想匹配以‘its’结束的任意完整单词，可以使用：

```
r'\w*its\b'
```

字符集

可以通过在方括号（[ ]）中列举一些字符来表示一个模式中的字符集。出了列举字符之外，还可以通过给定某个范围，并使用连字号（-）分隔的第一个和最后一个单词来表示一个范围。这个范围的最后一个字符被包含在集合中，与其他的 Python 范围有所区别。在一个集合中，除了反斜线符号（\）、反方括号（]）和连字号（-），其他特殊符号都表示其本身，而在这几个符号没有被转义时，必须使用转义来表示（也就是在符号之前加上反斜线符号），这将形成集合语法的一部分。开发者可以通过转义字母表示法来表示一个集合中的字符种类，比如\d 或\S。集合中的\b 表示一个退格字符，而不是一个单词边界。如果集合的模式中的第一个字符，也就是 [右边的字符是一个脱字符号（^）时，表示这个集合是补集：这样的一个集合匹配除了集合模式表示法中那些在^后面的字符之外的任意字符。

字符集的一个常用功能就是通过定义哪些字符可以构成一个单词来匹配一个单词，单词字符集的定义不同于\w 的默认字符（字母和数字）。要想匹配一个包含单个或多个字符的单词，每个字符可以是字母、省略号或连字号，但不能是数字（例如，‘Finnegan-O'Hara’），可以使用：

```
r"[a-zA-Z'\-]+"
```

在上面这种情况下，并不严格要求使用反斜线符号来转义连字号，因为连字号的位置可以保证语法结构的明确性。但是，使用反斜线符号也是可取的，因为这使得该模式具有更好的可读性，可以将连字号作为字符集中的一个字符与那些用来指示范围的字符区分开来。

或操作

正则表达式模式中的竖线符号（|）用来指定可替代选择，具有较低的语法优先级。除非使用圆括号改变分组，否则 | 将应用于该竖线符号两边的整个模式，一直到字符串的起始和结束位置，或者到另一个竖线符号。模式可以由任何数量的使用 | 连接的子模式组成。要想匹配这样的正则表达式，首先尝试匹配第一个子模式，如果匹配，则其他子模式将被跳过。如果不匹配第一个子模式，则尝试匹配第二个子模式，依此类推。| 既不是贪婪的，也不是非贪婪的：该符号并没有考虑匹配的长度。

给定一个单词列表 L，则用来匹配任意单词的正则表达式模式如下：

```
'|'.join([r'\b%s\b' % word for word in L])
```

如果 L 中的项目是更普通的字符串，而不仅仅是单词，则需要使用 re.escape 函数对这些字符串进行转义（参见第 9.7 节中介绍的 escape 函数），并且，开发者可能并不想要在任何一边使用 \b 单词边界标记。在这种情况下，可以使用下面的正则表达式模式：

```
'|'.join(map(re.escape,L))
```

组

正则表达式可以包含任意数量的组，从 0 到 99（允许使用任何数字，但是只完全支持前 99 个组）。模式字符串中的圆括号表示一个组。元素(?P<id>...)也表示一个组，并将该组命名为 id，这个名称可以是任意 Python 标识符。不管是被命名的还是没有被命名的，所有组都被从左到右编号，从 1 到 99；组 0 表示整个正则表达式。

对于正则表达式与一个字符串的任何匹配项目，每个组都匹配一个子字符串（可能是一个空白字符串）。在正则表达式使用 | 时，某些组可能不匹配任意子字符串，也就是说该组不参与匹配。空白字符串（‘’）被用于不参与匹配的任意组的匹配子字符串，除了本章后面指出的另一种情况之外。

例如：

```
r'(.+)\1+\Z'
```

可以匹配一个由两个或多个任意非空子字符串的重复组成的字符串。模式的(.+)部分匹配任意非空子字符串（任意字符，一次或多次）并定义了一个组，这样感谢其中的圆括号。模式的\1+ 部分匹配该组的一次到多次重复，而\Z 将匹配锚定到字符串的末尾。

可选标记

在（? 和）之间包含一个或多个 iLmsux 的正则表达式模式元素可以在模式中设置正则表达式选项，而不是使用 re 模块的 compile 函数的 flags 参数。选项可以应用于整个正则表达式，不管选项元素出现在模式的什么位置。为了更清楚，总是将选项放在模式的起始位置。如果 x 出现在选项中，则必须强制将其放在起始位置，因为 x 可以改变 Python 解析模式的方式。

使用显式 flags 参数要比在模式中放置一个选项元素更具可读性。compile 函数的 flags 参数是一个通过按位与（使用 Python 的按位与运算符 |）操作 re 模块的一个或多个以下这些属性而建立的编码整数。为了方便，每个属性都有一个短名称（单个大写字母）和一个长名称（大写的多字母标识符），这些属性更具可读性，因此通常用起来更方便：

I 或 IGNORECASE

不区分大小写匹配；

L 或 LOCALE

根据当前语言环境中定义的字母和数字进行\w、\W、\b 和\B 的匹配；

M 或 MULTILINE

让特殊字符^和$匹配每个行的起始和结尾（也就是，换行符之后/之前），也匹配整个字符串的起始和结尾；

S 或 DOTALL

让特殊字符“.”匹配任意字符，包括换行符；

U 或 UNICODE

根据 Unicode 定义的字母和数字进行\w、\W、\b 和\B 的匹配；

X 或 VERBOSE

忽略模式中的空白字符，除了被转义的或字符集中的空白字符，并使得模式中的 # 字符开始一个注释行，直到该行的末尾。

例如，使用 compile 函数可以有 3 种方法定义相同的正则表达式，参见第 9.7 节中介绍的 compile 函数。这 3 个正则表达式可以按大写和小写字母的任意混合匹配单词“hello”：

```
import re
r1 = re.compile(r'(?i)hello')
r2 = re.compile(r'hello', re.I)
r3 = re.compile(r'hello', re.IGNORECASE)
```

很显然，第三种方案是最具可读性的，因此也是最可维护的，尽管这种方案稍微有些冗长。这里不需要原生字符串形式，因为这些模式不包括反斜线符号；但是，使用原生字符串并没有什么坏处，而且是一种推荐的风格，可以让代码更加清楚。

通过适当地使用空白符号和注释，选项 re.VERBOSE（或 re.X）可以让模式更可读和更好理解。复杂和冗长的正则表达式模式通常最好使用多行字符串来表示，因此，通常需要使用三重引用的原生字符串格式来表示这些模式字符串。例如：

```
repat_num1 = r'(0[0-7]*|0x[\da-fA-F]+|[1-9]\d*)L?\Z'
repat_num2 = r'''(?x)                    # 这个模式匹配整数数字
             (0 [0-7]*                 | # 八进制：以 0 开始，然后是 0+八进制数字
```

```
    0x [\da-fA-F]+      | # 十六进制：以 0x 开始，然后是 1+ 十六进制数字
    [1-9] \d*           ) # 10 进制：以非 0 数字开始，然后是 0+ 数字
    L?\Z                  # 可选的拖尾字符 L，然后是字符串结尾
    '''
```

上面示例中定义的这两种模式是相等的，但是，第二种模式的可读性要更好一些，因为这种模式给出了注释，并使用空白字符将模式的各个部分按照逻辑的方式组合在一起。

匹配对比搜索

到现在为止，本书已经介绍了如何使用正则表达式来匹配字符串。例如，模式为 r'box' 的正则表达式可以匹配像‘box’和‘boxes’这样的字符串，而不能匹配‘inbox’。换句话说，正则表达式匹配隐式锚定于目标字符串的起始位置，就像该正则表达式的模式以\A 开始一样。

通常，开发者会对在字符串的任意位置定位一个正则表达式可能的匹配感兴趣，而不只是锚定在某个位置（例如，在‘inbox’、‘box’和‘boxes’这样的字符串中查找 r'box' 匹配）。在这种情况下，Python 用于这个操作的术语是搜索（search），与匹配（match）相对。对于这样的搜索操作，可以使用正则表达式对象的 search 方法：match 方法只能从起始位置处理匹配。例如：

```
import re
r1 = re.compile(r'box')
if r1.match('inbox'): print 'match succeeds'
else print 'match fails'                          # 打印：match fails
if r1. search('inbox'): print 'search succeeds'   # 打印：search succeeds
else print 'search fails'
```

锚定字符串的起始和结尾

模式元素\A 和\Z 分别用来确保一个正则表达式的搜索（或匹配）锚定于字符串的起始和字符串的结尾。更传统的方式是，用于起始的元素^和用于结尾的元素$可以实现类似的功能。^ 相当于\A，而$相当于\Z，可以用于不是多行的正则表达式对象（也就是，不包含模式元素(?m)，也不能带标记 re.M 或 re.MULTILINE 进行编译）。不过，对于多行正则表达式，^可以锚定任意行的起始位置（也就是，在整个字符串的起始位置，或者在换行符\n 之后的任何位置）。与此类似，对于多行正则表达式，$ 可以锚定任何行的末尾（也就是，在整个字符串的结尾位置，或者在换行符\n 之前的任何位置）。另一方面，\A 和\Z 可以锚定在字符串的起始和结尾，不管该正则表达式对象是否为多行。例如，下面的示例给出了如何检查一个文件是否具有以数字结尾的任何行：

```
import re
digatend = re.compile(r'\d$', re.MULTILINE)
if digatend.search(open('afile.txt').read()):
```

```
        print "some lines end with digits"
    else: print "no lines end with digits"
```

上面的示例与 r'\d\n' 模式几乎是相等的，但是，在这种情况下，如果该文件的最后一个字符是一个后面没有带行尾字符的数字，则搜索将会失败。对于上面的示例，如果一个数字位于该文件的内容的末尾，并且在通常情况下，该数字的后面带一个行尾符号，则搜索将会成功。

正则表达式对象

正则表达式对象 r 具有以下只读属性，这些属性详细定义了如何构建 r（使用模块 re 的 compile 函数，参见第 9.7 节中介绍的 compile 函数）：

flags

传递到 compile 函数的 flags 参数，在省略 flags 时，默认值为 0；

groupindex

一个字典，其键是由元素(?P<id>)定义的组名；对应的值是指定的组的编号；

pattern

用来编译对象 r 的模式字符串。

这些属性可以很容易地将一个已编译的正则表达式对象还原到模式字符串和标记，因此开发者不需要分别存储这些内容。

正则表达式对象 r 还提供了一些方法以在一个字符串内定位 r 的匹配项目，还可以对这些匹配项目执行替换操作。匹配项目通常是由特殊对象表示的，参见第 9.7 节。

表 9-11

findall	*r*.findall(*s*)
	在 r 没有组时，findall 将返回一个字符串列表，每个字符串都是 s 的一个子字符串，而 s 是 r 的一个非覆盖匹配。例如，要想打印输出一个文件中的所有单词，每个单词一行：
	`import re`
	`reword = re.compile(r'\w+')`
	`for aword in reword.findall(open('afile.txt').read()):`
	`  print aword`
	在 r 具有一个组时，findall 也将返回一个字符串列表，每个字符串都是 s 的子字符串，但是 s 匹配 r 的这个组。例如，要想只打印后面带一个空白符号（不包括标点符号）的单词，只需要更改上面这个示例中的一条语句：
	`reword = re.compile('(\w+)\s')`
	在 r 具有 n 个组（n>1），findall 将返回一个元组列表，每个元组都与 r 非覆盖匹配。每个元组都有 n 个项目，r 的每个组一个项目，而 r 是匹配这个组的 s 的子字符串。例如，要想打印至少包含两个单词的每个行的第一个和最后一个单词：

续表

findall	import re first_last = re.compile(r'^\W*(\w+)\b.*\b(\w+)\W*$', re.MULTILINE) for first, last in \ first_last.findall(open('afile.txt').read()): print first, last
finditer	*r*.finditer(*s*) finditer 与 findall 类似，但是不返回字符串（或者元组）列表，而是返回一个迭代器，其中的项目与对象匹配。因此，在大多数情况下，finditer 要比 findall 更灵活，也更好执行。
match	*r*.match(*s*,*start*=0,*end*=sys.maxint) 在 s 的一个子字符串（从索引 start 开始，但是不要达到索引 end）匹配 r 时，将返回一个适当的匹配对象。否则，匹配将返回 None。请注意，这个匹配将隐式锚定 s 中的起始位置 start。要想从 start 开始在 s 中的任何位置搜索一个与 r 匹配的项目，可以调用 r.search，而不是 r.match。例如，下面是如何打印一个文件中以数字开始的所有行的例子： import re digs = re.compile(r'\d+') for line in open('afile.txt'): if digs.match(line): print line,
search	*r*.search(*s*,*start*=0,*end*=sys.maxint) 在 s 最左边的子字符串（从索引 start 开始，但是不要达到索引 end）匹配 r 时，返回一个适当的匹配对象。在不存在这样的子字符串时，搜索将返回 None。例如，要想打印所有包含数字的行，可以使用下面这个简单的方法： import re digs = re.compile(r'\d+') for line in open('afile.txt'): if digs.search(line): print line,
split	*r*.split(*s*,*maxsplit*=0) 返回 s 的一个按 r 分割的列表 L（也就是，按非覆盖、非空地匹配 r 而分开的 s 的子字符串）。例如，要想从一个字符串中删除出现的所有子字符串‘hello’（不区分大小写），一个方法就是： import re rehello = re.compile(r'hello', re.IGNORECASE) astring = ''.join(rehello.split(astring)) 在 r 具有 n 个组时，多于 n 个的项目在 L 中的每个分割对之间交织出现。每 n 个额外项目都是匹配 r 的对应组中 s 的子字符串，如果这个组不参与匹配操作，则这些项目为 None。例如，下面是一种删除只在冒号和数字之间出现的空白字符的方法： import re re_col_ws_dig = re.compile(r'(:)\s+(\d)') astring = ''.join(re_col_ws_dig.split(astring))

续表

split	如果 maxsplit 大于 0，则 L 中最多有 maxsplit 个分割，每个分割后面带 n 个上面提到的项目，而且在 r 的 maxsplit 个匹配之后的拖尾子字符串（如果有的话）就是 L 的最后一个项目。例如，要想只删除第一个出现的子字符串‘hello’，而不是所有出现的该字符串，可以将上面第一个示例中的最后一条语句更改为： `astring = ''.join(rehello.split(astring, 1))`

sub	*r*.sub(*repl*,*s*,*count*=0)
	返回 s 的一个副本，其中与 r 非覆盖匹配的项目将被替换为 repl，repl 可以是一个字符串或者是一个可调用对象，比如函数。只有在不与前一个匹配相邻时，一个空白匹配才会被替换。在 count 大于 0 时，在 s 中只有前面 count 个匹配 r 的项目将被替换。在 count 等于 0 时，s 中所有匹配 r 的项目都将被替换。例如，下面是另一种只删除第一次出现的不区分大小写的子字符串‘hello’的方法：
	`import re`
	`rehello = re.compile(r'hello', re.IGNORECASE)`
	`astring = rehello.sub('', astring, 1)`
	如果不提供 sub 方法的最后一个参数 1，则该示例将删除出现的所有‘hello’。
	在 repl 是一个可调用对象时，repl 必须接受一个参数（一个匹配对象），并返回一个字符串（或者 None，这相当于返回空白字符串‘’）以将其用作匹配项目的替换内容。在这种情况下，sub 将以一个适当的匹配对象作为参数，对每个与 r 匹配的项目调用 repl，sub 将替换这个项目。例如，要想将出现的所有以‘h’开始，以‘o’结束（不区分大小写）的单词更改为大写字母，可以使用以下方法：
	`import re`
	`h_word = re.compile(r'\bh\w+o\b', re.IGNORECASE)`
	`def up(mo): return mo.group(0).upper()`
	`astring = h_word.sub(up, astring)`
	在 repl 是一个字符串时，sub 将使用其本身作为替换内容，区别只是该字符串扩展了反向引用。反向引用就是形式为\g<id>的 repl 子字符串，其中 id 是 r 中一个组的名称（是由 r 的模式字符串中的语法(?P<id>)建立的），或者是\dd，其中 dd 是一个或两个作为组的编号的数字。不管是命名的还是编号的，每个反向引用都将被替换为 s 的子字符串，该子字符串匹配反向引用指定的 r 的组。例如，下面是一个将每个单词都包含在一个花括号中的方法：
	`import re`
	`grouped_word = re.compile('(\w+)')`
	`astring = grouped_word.sub(r'{\1}', astring)`

subn	*r*.subn(*repl*,*s*,*count*=0)
	subn 与 sub 类似，区别在于 subn 将返回一个数据对（new_string, n），其中 n 是 subn 已经执行的替代的数量。例如，下面是一个计算不区分大小写的子字符串‘hello’的出现次数的例子：
	`import re`
	`rehello = re.compile(r'hello', re.IGNORECASE)`
	`junk, count = rehello.subn('', astring)`
	`print 'Found', count, 'occurrences of "hello"'`

匹配对象

匹配对象是由正则表达式对象的 match 和 search 方法创建和返回的，也是 finditer 方法返回的迭代器中的项目。匹配对象还可以由 sub 和 subn 方法隐式创建（在参数 repl 是可调用对象时），因为在这种情况下，一个适当的匹配对象将被传递为每个 repl 调用的参数。匹配对象 m 提供了以下只读属性，这些属性可以详细说明如何创建 m：

pos

被传递为 search 或 match 的 start 参数（也就是，s 中开始搜索一个匹配项目的索引）；

endpos

被传递为 search 或 match 的 end 参数（也就是，s 中匹配子字符串结束之前的索引）；

lastgroup

最后匹配的组的名称（如果最后匹配的组没有名称，或者没有组参与匹配，则该属性为 None）；

lastindex

最后匹配的组的整数索引（1 和更多）（如果没有组参与匹配，则该属性为 None）；

re

正则表达式对象 r，其方法创建了 m；

string

传递给 match、search、sub 或 subn 的字符串 s。

匹配对象 m 还提供了如表 9-12 所示的几个方法。

表 9-12

end, span, start	*m*.end(*groupid*=0) *m*.span(*groupid*=0) *m*.start(*groupid*=0) 这些方法将在 m.string 中返回与 groupid（组编号或组名）标识的组匹配的子字符串的界限索引。在匹配子字符串是 m.string[i:j]时，m.start 将返回 i、m.end 将返回 j，而 m.span 将返回（i, j）。如果这个组没有参与匹配，则 i 和 j 都是−1。
expand	*m*.expand(*s*) 返回 s 的一个副本，其中的转义序列和反向引用都将按照与 r.sub 方法（参见第 9.7 节中介绍的 sub 方法）相同的方式被替换。
group	*m*.group(*groupid*=0,**groupids*) 在使用单个参数 groupid（一个组编号或组名称）调用 group 时，将返回与 groupid 标识的组匹配的子字符串，如果该组没有参与匹配，则返回 None。习惯用法 m.group( )还可以写作 m.group(0)，以返回整个匹配的子字符串，因为组编号为 0 表示整个正则表达式。

续表

group	在使用多个参数调用 group 时，每个参数必须是一个组编号或组名称。然后，group 将返回一个元组，每个参数一个项目，每个项目都是匹配对应的组的子字符串，如果这个组没有参与匹配，则返回 None。
groups	*m*.groups(*default*=None) 返回一个元组，r 中的每个组一个项目。每个项目都是匹配对应的组的子字符串，如果这个组没有参与匹配，则返回 default。
groupdict	*m*.groupdict(*default*=None) 返回一个字典，其中的键是 r 中所有被命名的组的名称。每个组的名称就是一个匹配对应组的子字符串，如果这个组没有参与匹配，则返回 default。

re 模块的函数

re 模块可以提供第 9.7 节中列出的属性，还为正则表达式对象（findall、finditer、match、search、split、sub 和 subn）的每个方法提供了一个函数，每个函数都有一个附加的第一个参数，也就是该函数隐式编译到正则表达式对象中的模式字符串。通常比较可取的做法就是将模式字符串显式编译到正则表达式对象中，并调用该正则表达式对象的方法，但是有时候，对于一次性地使用正则表达式模式而言，调用 re 模块的函数要稍微方便一些。例如，要想计算不区分大小写的子字符串‘hello’的出现次数，可以使用下面这个基于函数的方法：

```
import re
junk, count = re.subn(r'(?i)hello', '', astring)
print 'Found', count, 'occurrences of "hello"'
```

在这种情况下，正则表达式选项（例如，不区分大小写）必须被编码为正则表达式模式元素（也就是，(?i)），因为 re 模块的函数不能接受 flags 参数。re 模块将内部缓存从传递给该模块的函数的模式参数创建的正则表达式对象；要想清除这个缓存并收回内存，可以调用 re.purge( )。

re 模块还提供了 error、由错误引发的异常类（通常是指模式字符串中的语法错误）和表 9-13 列出的两个函数。

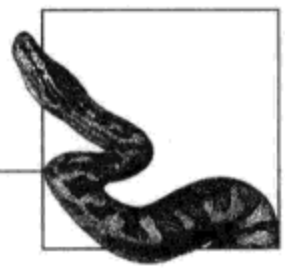
表 9-13

compile	compile(*pattern*,*flags*=0) 创建并返回一个正则表达式对象，按照 9.7 节中介绍的模式字符串语法对字符串 pattern 进行语法解析，并使用了整数 flags，参见第 9.7 节中介绍的可选标记。
escape	escape(*s*) 返回字符串 s 的一个副本，其中的每个非字母数字字符都被转义了（也就是，前面带一个反斜线符号 \），这个函数可以用于逐字逐句地将字符串 s 匹配为正则表达式模式字符串的一部分。

第 3 部分

Python 库和扩展模块

第 10 章



文件和文本操作

本章将介绍大多数与处理 Python 中的文件和文件系统有关的问题。文件（file）是一个程序可以读取和/或写入的字节流；文件系统（filesystem）是计算机系统上的文件的分层结构存储空间。

10.1 其他与处理文件有关的章节

因为文件是程序设计中的一个非常至关重要的概念，因此，尽管本章是这本书中最长的一章，但是其他几章还是包含了一些与处理特定类型的文件有关的资料。特别是，本书第 11 章中介绍了如何处理与持久化和数据库函数有关的多个类型的文件（参见第 11.1 节中介绍的 marshal 文件、pickle 文件、shelve 文件以及第 11.2 节中介绍的 DBM 和类 DBM 文件，以及第 11.3 节中介绍的 Berkeley 数据库文件），第 23 章中介绍了如何处理 HTML 格式的文件，第 24 章中介绍了如何处理 XML 格式的文件。

10.2 本章的组织结构

首先，本章在第 10.3 节中介绍了 Python 程序读取和写入数据的最典型方法，这种方法是通过内置 file 对象来实现的。在此之后，本章在第 10.3 节中介绍了类文件对象（不是文件，但是其行为从某种程度上类文件对象）的多态性概念。

本章接下来介绍了用来处理临时文件和类文件对象（参见第 10.3 节中介绍的 tempfile 模块和第 10.5 节中介绍的 StringIO 和 cStringIO 模块）。

然后，本章介绍了可以用来访问文本和二进制文件内容的模块（参见第 10.4 节中介绍的 fileinput 模块、linecache 模块和 struct 模块），这些模块还支持压缩文件和其他数据存

档文件（参见第 10.6 节中介绍的 gzip 模块、bz2 模块、tarfile 模块、zipfile 模块和 zlib 模块）。

在 Python 中，os 模块提供了许多可以对文件系统进行操作的函数，因此本章接下来在第 10.7 节中介绍了 os 模块。然后，在第 10.8 节中介绍了如何对 os、os.path 和其他模块（参见第 10.8 节中介绍的 listdir 函数和 dircache 模块、stat 模块、filecmp 模块和 shutil 模块）提供的文件系统进行操作（比较、复制和删除目录和文件、处理文件路径和访问底层文件描述符）。

尽管许多现代程序都依赖于图形用户界面（GUI）（参见第 17 章），但是基于文本的、非图形用户接口仍然十分有用，因为这种接口比较简单，编程也快，而且是轻量级的。本章通过一些资料在第 10.9 节中对 Python 中的文本输入和输出、在第 10.10 节中对富文本 I/O、在第 10.11 节中对交互式命令行会话，最后，在第 10.12 节中对有关如何显示不同用户可以理解的文本的信息（不管这些用户在哪里，也不管用户说哪种语言）进行了总结，这个主题通常被称为“国际化”（通常也被缩写为 i18n）。

10.3 文件对象

正如第 10.2 节中提到的，file 是 Python 中的一个内置类型，也是 Python 程序读取或写入输入的一个最常用的方法。通过文件对象，开发者可以通过底层操作系统从文件读取数据和/或将数据写入文件。Python 可以通过引发一个内置异常类 IOError 实例以对任何与文件对象有关的 I/O 错误作出反应。可能会导致这个异常的错误包括：打开或创建一个文件时的 open 错误、对一个不能应用某个方法的文件对象调用这个方法（例如，对只读文件对象调用 write 方法，或者对不可搜寻文件调用 seek），以及文件对象的方法诊断出来的其他 I/O 错误。本节将介绍文件对象，以及有关创建临时文件的重要问题。

使用 open 创建文件对象

要想创建一个 Python 文件对象，可以使用以下语法调用内置 open 函数：

```
open(filename, mode='r', bufsize=-1)
```

open 可以打开由普通字符串 filename 指定的文件，filename 还给出了文件的路径。open 将返回一个 Python 文件对象 f，该对象是内置类型 file 的一个实例。当前，直接调用 file 就像调用 open 一样，但是开发者必须调用 open 函数，在某些将来版本的 Python 中，open 函数可能会称为一个工厂函数。如果向 open 函数显示传递一个 mode 字符串，并且 filename 指定的文件还不存在，则 open 可以创建这个文件（取决于 mode 的值，下面将详细介绍）。换句话讲，尽管函数的名称为 open，但是 open 并不只是用来打开一个现有文件，还可以创建新文件。

文件模式

mode 是一个字符串，可以指定如何打开（或创建）一个文件。mode 可以是如下形式。

'r'

该文件必须已经存在，并且以只读模式打开。

'w'

以只写模式打开文件。如果该文件已经存在，则文件将被截短和改写，如果该文件还不存在，则创建这个文件。

'a'

以只写模式打开文件。如果该文件已经存在，则文件将仍保持完整，写入的数据将被添加到文件中已有数据的后面。如果该文件还不存在，则创建这个文件。对该文件调用 f.seek 并没有什么坏处，但是不会产生任何效果。

'r+'

该文件必须已经存在，并且将被打开以供读取和写入，因此可以调用 f 的所有方法。

'w+'

该文件将被打开以供读取和写入，因此可以调用 f 的所有方法。如果该文件已经存在，则文件被截短和改写，如果该文件还不存在，则创建这个文件。

'a+'

该文件将被打开以供读取和写入，因此可以调用 f 的所有方法。如果该文件已经存在，则文件将仍保持完整，写入的数据将被添加到文件中已有数据的后面。如果该文件还不存在，则创建这个文件。如果对 f 的下一个 I/O 操作是写入数据，则对该文件调用 f.seek 不会产生任何效果，但是如果对 f 的下一个 I/O 操作是读取数据，则 f.seek 可以正常工作。

二进制和文本模式

mode 字符串还可以在上一节介绍的任何模式值后面带一个字母 b 或 t。b 表示二进制模式，而 t 表示文本模式。在 mode 字符串既没有带 b，也没有带 t 时，默认值为文本模式（也就是，‘t’也就是‘rt’、‘w’也就是‘wt’，依此类推）。

在 UNIX 中，二进制和文本模式之间并没有什么区别。在 Windows 中，当以文本模式打开一个文件，并读取该文件的数据时，每次遇到值为 os.linesep（行终止字符串）的字符串，将会返回‘\n’。与此相反，每次向文件写入‘\n’时，还将写入 os.linesep 的一个副本。

这种已经广为人知的惯例，最初是在 C 语言中开发的，可以让开发者对任意平台上的文件进行读取和写入，而不必担心该平台的行分隔规定。但是，除了类 UNIX 平台，开发者不必知道（并通过向 open 传递适当的 mode 参数来告诉 Python）一个文件是二进制文件还是文本文件。在本章中，为了简化，使用了\n 来表示行终止字符串，但是，请记住该字符串实际上是文件系统上的文件中的 os.linesep，只有对于以文本模式打开的文件，才会在内存中与\n 互相翻译。

Python 还提供了通用换行符（universal newlines），在开发者不知道文件中的行分隔符是如何编码时，这个换行符可以让开发者在模式‘U’（或者‘rU’，二者相同）下打开这个文本文件以进行读取操作。这样做是很有用的，例如，在开发者通过网络在不同操作系统的计算机之间共享文本文件时。模式‘U’将把‘\n’、‘\r’和‘\r\n’中的任意一个都看作是行分隔符，并将任何行分隔符都翻译成‘\n’。

缓冲

bufsize 是一个整数，表示开发者为文件申请的缓冲区大小。在 bufsize 小于 0 时，将使用该操作系统的默认值。通常，这个默认值是对应于交互式控制台文件的行缓冲区大小和用于其他文件的一些合理的缓冲区大小，比如 8192 字节。在 bufsize 等于 0 时，该文件是不需要缓冲区的；其效果就像该文件的缓冲区在每次向文件写入任何内容时都被刷新一样。在 bufsize 等于 1 时，该文件使用的是行缓冲，这意味着文件的缓冲区在每次向文件写入\n 时将被刷新。在 bufsize 大于 1 时，该文件使用大约 bufsize 字节大小的缓冲区，会自动舍入到某个合理的值。在某些平台上，开发者可以更改已经打开的文件的缓冲区设置，但是没有一种跨平台的方法可以实现这个功能。

连续和非连续访问

文件对象 f 天生就具有连续性（也就是，字节流）。在从一个文件读取字节时，可以按照这些字节在文件中显示的连续顺序得到这些字节。在将字节写入一个文件时，将按照写入顺序将这些字节放到文件中。

要想允许非连续访问，每个内置文件对象必须跟踪其当前位置（也就是文件中下一次读取或写入操作将要开始传送数据的位置）。在打开一个文件时，初始位置在文件的开头。不管什么时候对以模式‘a’或‘a+’打开的文件对象 f 调用 f.write，在向 f 写入数据之前，都将把 f 的位置设置到该文件的末尾。在对文件对象 f 读取或写入 n 个字节之后，f 的位置将向前移动 n。开发者可以通过调用 f.tell 查询 f 的当前位置，调用 f.seek 更改 f 的当前位置，下一节将介绍这两个方法。

文件对象的属性和方法

文件对象 f 提供了本节中列举的属性和方法。

表 10-1

close	*f*.close() 关闭该文件。在执行了 f.close 之后，不能再对 f 调用任何其他方法。多次调用 f.close 是允许的，并没有什么坏处。
closed	closed f.closed 是一个只读属性，如果已经调用了 f.close()，则该属性的值为 True；否则，该属性的值为 False。
encoding	encoding f.encoding 是一个只读属性，如果 f 上的 I/O 使用了系统默认的编码方式，则该属性的值为 None，否则，该属性的值为实际使用的编码方式的名称（参见第 9.6 节中介绍的编码方式）。实际上，只有在 sys 模块的 stdin、stdout 和 stderr 属性（参见第 8.3 节中对 stdin、stdout 和 stderr 的介绍）涉及终端时，才会对这些属性设置 encoding 属性。
flush	*f*.flush() 请求将 f 的缓冲区写出到操作系统，这样，操作系统看到的文件与 Python 代码写入的文件具有完全相同的内容。根据平台和 f 的底层文件特性，f.flush 可能无法确保想要得到的效果。
isatty	*f*.isatty() 如果 f 的底层文件是一个交互式终端，则返回 True；否则返回 False。
fileno	*f*.fileno() 返回一个整数，这个整数是 f 表示的文件在操作系统级别的文件描述符。参见第 10.8 节中对文件描述符的介绍。
mode	mode f.mode 是一个只读属性，是在调用 open 函数创建 f 时使用的 mode 字符串的值。
name	name f.name 是一个只读属性，是在调用 open 函数创建 f 时使用的 filename 字符串的值。
newlines	newlines f.newlines 是一个只读属性，用于被打开以进行“通用换行符读取”的文本文件。f.newlines 可能是字符串‘\n’、‘\r’或‘\r\n’之一（当这个字符串是在读取 f 时已经遇到的唯一类型的行分隔符时），或者是一个元组，其中的项目是已经遇到的不同类型的行分隔符，或者是 None，在读取 f 时还没有遇到任何行分隔符，或者不是以模式‘U’打开 f 时。
read	*f*.read(*size*=-1) 从 f 的文件中读取最多 size 个字节，并将读取的字节返回为一个字符串。如果文件在读取 size 个字节之前就结束了，则 read 将读取并返回少于 size 个字节。在 size 小于 0 时，read 将读取并返回直到文件结束的所有字节。如果文件的当前位置就在文件的末尾，或者 size 等于 0，则 read 将返回一个空白字符串。
readline	*f*.readline(*size*=-1) 从 f 的文件中读取并返回一行，直到行尾（\n），并包括该行尾。如果 size 大于或等于 0，readline 将读取不超过 size 个字节。在这种情况下，返回的字符串可能不会以\n 结束。如果 readline 一直读取到了文件的末尾，但还是没有找到 \n，则返回的字符串中也有可能不包含\n。如果文件的当前位置就在文件的末尾，或者 size 等于 0，则 readline 将返回一个空白字符串。

续表

readlines	*f*.readlines(*size*=-1) 读取并返回由 f 的文件中的所有行（每个以\n 结束的字符串）组成的列表。如果 size 大于 0，则 readlines 将在收集到总数为大约 size 个字节的数据，而不是一直读取到文件的末尾之后，停止读取操作并返回这个列表。
seek	*f*.seek(*pos*, *how*=0) 将 f 的当前位置设置为从参考点偏移带符号整数个 pos 字节。how 表示这个参考点。在 how 为 0 时，参考点是文件的开头；在 how 为 1 时，参考点是文件的当前位置；在 how 为 2 时，参考点是文件的末尾。在 Python 2.5 中，os 模块具有名为 SEEK_SET、SEEK_CUR 和 SEEK_END 的属性，其值分别是 0、1 和 2。在调用这个方法时，使用这些属性，而不是纯粹的整数常数可以让程序获得更好的可读性。 在按文本模式打开 f 时，f.seek 可能会以无法预料的方式设置当前位置，由于 os.linesep 和\n 之间的隐式翻译。这种麻烦情况不会出现在 UNIX 平台上，也不会在以二进制模式打开 f 时和在使用 pos（前面调用 f.tell 的结果）来调用 f.seek，并且 how 为 0 时出现。在以模式‘a’或‘a+’打开 f 时，写入到 f 中的所有数据都被添加到已经在 f 中的数据之后，不管是否调用了 f.seek。
softspace	softspace f.softspace 是 print 语句（参见第 10.9 节）内部使用的一个读-写 bool 属性，用来跟踪其自身的状态。文件对象不会改变，也不会以任何方式解释 softspace：文件对象只会让这个属性可以自由地读取和写入，而 print 将处理其他属性。
tell	*f*.tell() 返回 f 的当前位置，从文件的起始位置开始，以字节为单位的整数偏移量。
truncate	*f*.truncate([*size*]) 截短 f 的文件。在给出了 size 时，将把该文件截短为最多 size 个字节。在没有给出 size 时，使用 f.tell( )作为该文件的新大小。
write	*f*.write(*s*) 将字符串 s 的所有字节写入到文件中。
writelines	*f*.writelines(*lst*) 相当于： for *line* in lst: f.write(*line*) 不管可迭代对象 lst 中的字符串是否是行：不管其名称是什么，writelines 方法只是将每个字符串写入到文件中，一个接着一个。

文件对象的迭代

文件对象 f（打开以进行文本模式的读取）也是一个迭代器，其中的项目就是文件的行。这样，下面的循环：

```
for line in f:
```

可以对文件的每个行进行迭代。由于缓冲的原因，提前打断这样一个循环（例如，使用 break），或者调用 f.next()而不是 f.readline()，可能会将该文件的位置设置为一个任意

值。如果开发者想要从使用 f 作为一个迭代器切换到调用 f 的其他读取方法，要确保通过适当地调用 f.seek 将该文件的位置设置为一个已知的值。在 plus side，直接对 f 进行循环具有非常好的性能，因为这些规范允许这个循环使用内部缓冲以最小化 I/O，即使是对巨大的文件，也不会占用过多的内存。

类文件对象和多态性

当一个对象 x 的行为表现出文件的多态性时，这个对象就是类文件的，表示一个函数（或者一个程序的其他一些部分）可以使用 x，就像 x 是一个文件一样。使用这样的对象的代码（被称为该对象的客户代码）通常会将该对象接收为一个参数，或者通过调用一个返回该对象作为结果的工厂函数以得到这个对象。例如，如果客户代码对 x 调用的唯一方法就是 x.read( )，并且不带任何参数，则为了成为这个客户代码的类文件，x 对象需要提供的所有方法就是一个不带任何参数，并返回一个字符串的可调用 read 方法。其他客户代码可能需要用到 x 对象以实现文件方法的一个更大的子集。类文件对象和多态性并不是绝对概念：他们与某些特定客户代码对一个对象提出的要求有关。

多态性是面向对象编程的一个强大的功能，而类文件对象是多态性的一个很好的例子。写入到文件或者从文件读取的客户代码模块可以自动重用于保持在其他位置的数据，只要该模块没有被类型测试的不确定性打破多态性。在介绍第 8.2 节中的内置 type 和 isinstance 时，本书提到过，通常最好避免进行类型测试，因为类型测试会妨碍 Python 提供的正常多态性。但有时候，开发者可能没有选择。例如，marshal 模块（参见第 11.1 节）要求真正的文件对象。因此，在开发者的客户代码需要使用 marshal 模块时，代码必须处理真正的文件对象，而不只是类文件对象。但是，这样的情况是及其少见的。最常见的是，要想在客户代码中支持多态性，开发者只需要避免类型测试即可。

开发者可以通过编写自己的类（参见第 5 章）和定义客户代码需要的特定方法（比如，read）来实现类文件对象。类文件对象 f1 不需要实现一个真正的文件对象 f 的所有属性和方法。如果开发者可以确定客户代码对 f1 调用了哪些方法，可以选择只实现这个子集。例如，在 f1 只准备被写入时，f1 不需要“读”方法，比如 read、readline 和 readlines。

在开发者实现一个可写的类文件对象 f1 时，要确信 f1.softspace 可以被读取和写入，并且不会改变，也不会以任何方法解释 softspace，如果开发者想要 f1 可以被 print 使用（参见第 10.9 节）。请注意，当用户在 Python 中编写 f1 的类时，这个行为是默认的。只有在 f1 的类覆盖了特殊方法__setattr__，或者相反，在控制了对其实例的属性的访问（例如，通过定义__slots__）时，开发者才需要特别注意，参见第 5 章。特别地，如果开发者的新型类定义了__slots__，则其中的一个槽（slot）必须被命名为 softspace，如果开发者想要类实例可以用作 print 语句的目的。

如果开发者需要一个类文件对象，而不是一个真正的文件对象的主要原因是将数据保留在内存中，可以使用 StringIO 和 cStringIO 模块，参见第 10.5 节。这些模块提供了一些类文

件对象，可以将数据保持在内存中，并且可以从很广泛的程度上让文件对象表现出多态性。

tempfile 模块

tempfile 模块可以帮助开发者以系统平台提供的最安全方式创建临时文件和目录。在处理大量可能不适合于放在内存中的数据，或者在程序需要写入一些数据，以供其他进程使用时，临时文件通常都是一个非常好的解决方案。

这个模块中的函数所使用的参数的顺序有些混乱：要想让代码更有可读性，可以总是使用命名参数语法调用这些函数。tempfile 模块提供了如表 10-2 所示的函数。

表 10-2

mkstemp	mkstemp(*suffix*=None, *prefix*=None, *dir*=None, *text*=False)
	安全地创建一个新临时文件，只能由当前用户读取和写入，但是不能执行，也不能被子进程继承；该函数将返回一个数据对（fd, path），其中 fd 是该临时文件的文件描述符（是由 os.open 返回的，参见第 10.6 节中介绍的 open 函数），并且字符串 path 是到该临时文件的绝对路径。开发者可以有选择地传递参数，以指定被用作临时文件的名称的起始（prefix）和结束（suffix）字符串，以及创建该临时文件的目录的路径（dir）；如果开发者想要将这个临时文件用作一个文本文件，必须显式传递参数 text=True。请确保在已经使用完了临时文件之后，删除该临时文件。下面是一个有关使用临时文件的典型示例，该示例将创建一个临时文本文件、关闭该文件、将其路径传递到另一个函数，并最终确保该文件被删除：
	`import tempfile, os`
	`fd, path = tempfile.mkstemp(suffix='.txt', text=True)`
	`try:`
	`  os.close(fd)`
	`  use_filepath(path)`
	`finally:`
	`  os.unlink(path)`

mkdtemp	mkdtemp(*suffix*=None, *prefix*=None, *dir*=None)
	安全地创建一个只能由当前用户读取、写入和搜索的新临时目录，并返回到该临时目录的绝对路径。可选的参数 prefix、suffix 和 dir 与 mkstemp 函数的参数一样。请确保在已经使用完了临时目录之后，删除该临时目录。下面是一个有关使用临时目录的典型示例，该示例将创建一个临时目录、将其路径传递到另一个函数，并最终确保该目录和目录中的所有内容都被删除：
	`import tempfile, shutil`
	`path = tempfile.mkdtemp()`
	`try:`
	`  use_dirpath(path)`
	`finally:`
	`  shutil.rmtree(path)`

续表

TemporaryFile	TemporaryFile(*mode*='w+b', *bufsize*=-1,*suffix*=None, *prefix*=None, *dir*=None) 使用 mkstemp 创建一个临时文件(向 mkstemp 传递可选参数 suffix、prefix 和 dir)，参见第 10.8 节中介绍的 fdopen 函数，使用 os.fdopen 创建一个文件对象（向 fdopen 传递可选参数 mode 和 bufsize)，并返回该文件对象（或者包含该文件对象的一个类文件包装器)。在文件对象被关闭（隐式或显式）后，该临时文件将立即被删除。为了获得更好的安全性，如果开发者的操作系统平台允许，在文件系统中，创建的临时文件可以没有名称（类 UNIX 平台可以这样，而 Windows 则不可以)。
NamedTemporaryFile	NamedTemporaryFile(*mode*='w+b', *bufsize*=-1,*suffix*=None, *prefix*=None, *dir*=None) 这个函数与 TemporaryFile 函数类似，区别只是在于创建的临时文件在文件系统中必须有一个名称，并且可以使用该文件或类文件对象的名称属性访问这个名称。某些操作系统平台（主要是 Windows）不允许文件被再次打开：因此，如果开发者想要确保程序可以跨平台使用，文件名称的用处有限。如果开发者需要将该临时文件的名称传递到另一个要打开该文件的程序，可以使用 mkstemp 函数，而不是 NamedTemporaryFile，这样可以保证正确地跨平台操作。当然，在选择使用 mkstemp 时，开发者必须非常小心，以确保在对临时文件的操作完成之后删除该文件。

10.4 文件 I/O 的辅助模块

文件对象提供了文件 I/O 所需的所有最小限度和必不可少的功能。但是，有些辅助 Python 库模块提供了方便的辅助函数，在一些重要的情况下，这可以使得 I/O 更简单和便捷。

fileinput 模块

fileinput 模块可以循环遍历文本文件列表中的文件的所有行。fileinput 模块的性能是很好的，可以与直接对每个文件进行迭代的性能相媲美，因为 fileinput 使用缓冲来最小化 I/O。因此，只要开发发现这个模块的丰富功能很便利，就可以使用 fileinput 模块进行面向行的文件输入，而不必担心性能问题。input 函数是 fileinput 模块的关键函数，并且该模块还提供了 FileInput 类，这个类的方法提供了与 fileinput 模块的函数相同的功能。

表 10-3

close	close() 关闭整个序列，这样，迭代将会停止，并且没有文件会继续保持打开。
FileInput	class FileInput(*files*=None, *inplace*=False, *backup*='', *bufsize*=0) 创建并返回 FileInput 类的一个实例 f。其参数与 fileinput.input 的参数相同，并且 f 的方法与 fileinput 模块的函数具有相同的名称、参数和语义。f 还提供了一个 readline 方法，该方法可以读取并返回下一行。在开发者想要嵌套或混合从多于一个的文件序列读取行的循环操作时，可以显式使用 FileInput 类。

续表

filelineno	`filelineno()` 返回从现在正在读取的文件中到目前为止已经读取的行数。例如，如果从当前的文件中只读取了第一行，则返回 1。
filename	`filename()` 返回正在读取的文件的名称，如果还没有读取任何行，则返回 None。
input	`input(files=None, inplace=False, backup='', bufsize=0)` 返回多个文件中的行序列，适合于在 for 循环中使用。files 是一个文件名序列，input 函数将按顺序打开这些文件并一个接一个读取这些文件。文件名‘-’表示标准输入（sys.stdin）。如果 files 是一个字符串，表示一个要打开和读取的单独的文件名。如果 files 为 None，则 input 将使用 sys.argv[1:]作为文件名列表。如果文件名序列为空，则 input 将读取 sys.stdin。 input 返回的序列对象是 FileInput 类的一个实例；这个实例也是 input 模块的全局状态，这样，fileinput 模块的所有其他函数可以基于相同的共享状态进行操作。fileinput 模块的每个函数都直接对应于 FileInput 类的一个方法。 在 inplace 为 False（默认值）时，input 只读取这些文件。在 inplace 为 True 时，input 将把每个正在被读取的文件（除了标准输入）移动到一个备份文件中，并重定向标准输出（sys.stdout）到一个新文件中，这个新文件的路径与正在被读取的文件最初所在路径相同。通过这种方法，开发者可以模拟原地覆盖文件。如果 backup 是一个以句点符号开始的字符串，input 将使用 backup 作为这些备份文件的扩展名，并且不会删除这些备份文件。如果 backup 是一个空白字符串（默认值），input 将使用.bak 作为扩展名，并在输入文件被关闭后删除每个备份文件。 bufsize 是内部缓冲区的大小，input 使用这个缓冲区从输入文件读取行。如果 bufsize 为 0，则 input 将使用大小为 8192 字节的缓冲区。
isfirstline	`isfirstline()` 返回 True 或 False，就像 filelineno( )==1。
isstdin	`isstdin()` 如果当前正在读取的文件是 sys.stdin，则返回 True；否则，返回 False。
lineno	`lineno()` 返回从调用 input 开始读取的行的总数。
nextfile	`nextfile()` 关闭当前正在读取的文件，这样下一个要读取的行就是下一个文件的第一行。

linecache 模块

linecache 模块可以从一个给定名称的文件中读取一个给定的行（由行的编号来指定），并将其保存在一个内部缓存中，这样，如果从文件中读取了几行，则要比每次都打开和查看文件更快一些。linecache 模块提供了如表 10-4 所示的函数。

表 10-4

checkcache	checkcache() 可以确保模块的缓存不保存陈旧的数据，并真实反应文件系统上的数据。在正在读取的文件可能在文件系统中被改变的时候，可以调用 checkcache 函数以确保将来调用 getline 时返回更新的信息。
clearcache	clearcache() 清除模块的缓存，这样可以让内存重用于其他目的。开发者可以在知道自己暂时不再需要执行读取操作时，调用 clearcache 函数。
getline	getline(*filename*, *lineno*) 从名为 filename 的文本文件中读取并返回第 lineno 行（第一行为 1，而不是像 Python 中常用的那样为 0），还包括每个行的拖尾字符\n。如果出现了任何错误，getline 都不会引发异常，而是返回空白字符串 ''。如果本地没有找到文件名为 filename 的文件，getline 将在 sys.path 列出的目录中查找这个文件。
getlines	getlines(*filename*) 从名为 filename 的文本文件中读取并以字符串列表的形式返回所有行，还包括每个行的拖尾字符\n。如果出现了任何错误，getlines 不会引发异常，而是返回空白列表[ ]。如果本地没有找到文件名为 filename 的文件，则 getlines 将在 sys. path 列出的目录中查找这个文件。

struct 模块

struct 模块可以将二进制数据包装到一个字符串中，并将这样的字符串以字节的形式解包为其表现的数据。这样的操作对于许多类型的底层编程而言是很有用的。最常见的是，开发者使用 struct 模块解释从具有某些特定格式的二进制文件中读取的数据记录，或者准备将数据记录写入到这样的二进制文件中。这个模块的名称来自于 C 语言的关键字 struct，这个关键字用于相关的目的。对于出现的任何错误，struct 模块的函数将引发异常，这些异常都是异常类 struct.error 的实例，struct.error 类也是 struct 模块支持的唯一类。

struct 模块依赖于结构格式字符串（struct format string），也就是具有特定语法的普通字符串。格式字符串的第一个字符指定了被打包的数据的字节序、字节长度和对齐方式。

@

当前平台本身的字节序、字节长度和对齐方式；如果第一个字符不是这里列出的任何一个字符，则@作为默认值使用；

=

当前平台本身的字节序，标准字节长度和对齐方式；

<

小端模式（Little-endian）字节序（类似于 Intel 平台）；标准字节长度和对齐方式；

>, !

大端模式（Big-endian）字节序（网络标准）；标准字节长度和对齐方式。

表 10-5 中给出了标准字节长度。标准对齐方式表示不强制对齐，如果需要，可以使用显式填充字节。平台本身的字节长度和对齐方式也就是平台的 C 编译器使用的方式。平台本身的字节序可以是小端模式或大端模式，由平台而定。

表 10-5　struct 的格式字符

字符	C 类型	Python 类型	标准字节长度
B	unsigned char（无符号字符型）	int（整型）	1 字节
b	signed char（带符号字符型）	int（整型）	1 字节
c	char（字符型）	str（长度为 1）	1 字节
d	double（双精度浮点型）	float（浮点型）	8 字节
f	float（浮点型）	float（浮点型）	4 字节
H	unsigned short（无符号短整型）	int（整型）	2 字节
h	signed short（带符号短整型）	int（整型）	2 字节
I	unsigned int（无符号整型）	long（长整型）	4 字节
i	signed int（带符号整型）	int（整型）	4 字节
L	unsigned long（无符号长整型）	long（长整型）	4 字节
l	signed long（带符号长整型）	int（整型）	4 字节
P	void*　（空类型）	int（整型）	N/A
p	char[ ]（字符型）	String（字符串）	N/A（未指定）
s	char[ ]（字符型）	String（字符串）	N/A（未指定）
x	填充字节	无值	1 字节

在可选的第一个字符之后是一个格式字符串，该字符串由一个或多个格式字符（参见表 10-1 中的格式字符），每个字符前面可以选择带一个计数值（使用十进制数字表示的整数）。对于大多数格式字符，这个计数值表示重复次数（例如，‘3h’与‘hhh’完全相同）。在格式字符为 s 或 p——也就是字符串时——这个计数值不表示重复次数，而是该字符串占用的字节数量。在格式之间可以随意使用空白字符，但是不要在计数值和其格式字符之间使用空白字符。

格式 s 表示固定长度的字符串，其长度与计数值相等（如有需要，Python 字符串可以是被截短或者使用空字符‘\0’的副本填充的）。格式 p 表示类 Pascal 字符串：第一个字节是真正的字符的数量，而真正的字符是从第二个字节开始的。计数值是字节的总数，包括长度字节本身。

struct 模块提供了如表 10-6 所示的函数。

表 10-6

calcsize	`calcsize(fmt)` 返回与数据结构格式字符串 fmt 对应的数据结构的字节长度。
pack	`pack(fmt, *values)` 按照数据结构格式字符串 fmt 打包给定的值，并返回结果字符串。values 必须匹配 fmt 要求的值的数量和类型。
unpack	`unpack(fmt, s)` 按照数据结构格式字符串 fmt 将二进制字符串 s 解包，并返回一个由值组成的元组。len(s)必须等于 struct.calcsize(fmt)。

10.5 StringIO 和 cStringIO 模块

开发者可以通过编写 Python 类以提供所需的方法，从而实现类文件对象。如果开发者想要的只是将数据保存在内存，而不是保存到从操作系统可以看到的文件中，可以使用 StringIO 或 cStringIO 模块。这两个模块几乎是相同的：每个模块都提供了一个工厂函数，可以调用该工厂函数在内存中创建类文件对象。这两个模块之间的区别在于，StringIO 模块创建的对象都是类 StringIO.StringIO 的实例。开发者可以继承这个类以创建自己的自定义类文件对象，覆盖需要指定的方法，并且可以对这个类的对象指定输入和输出方法。另一方面，cStringIO 模块创建的对象都是两种特殊目的类型（一个专用于输入，另一个专用于输出）的实例，而不是类的实例。在使用 cStringIO 时，可以获得较好的性能，但是不支持继承，并且这两个模块不能对相同的对象同时执行输入和输出。此外，cStringIO 不支持 Unicode。

每个模块都提供了一个返回类文件对象 fl 的工厂函数 StringIO。

表 10-7

StringIO	`StringIO([s])` 创建并返回一个内存中的类文件对象 fl，包括内置文件对象的所有方法和属性。fl 的数据内容将被初始化为参数 s 的一个副本，要想用于 cStringIO 中的 StringIO 工厂函数，这个参数的副本必须是一个普通字符串，而要想用于 StringIO 中的函数，这个参数的副本可以是普通或 Unicode 字符串。在给出了 s 时，cStringIO.StringIO 将生成一个适合于被读取的对象；在没有提供 s 时，cStringIO.StringIO 将生成一个适合于被写入的对象。 除了内置文件对象的所有方法和属性（参见第 10.3 节），fl 还提供了一个辅助方法，getvalue。
getvalue	`fl. getvalue()` 以字符串的形式返回 fl 的当前数据内容。开发者不能在调用了 fl.close 之后再调用 fl.getvalue；close 将释放 fl 内部保存的缓冲区，而 getvalue 需要访问这个缓冲区以生成其返回结果。

10.6 压缩文件

存储空间和传输带宽变得越来越便宜和丰富，但是，在许多情况下，开发者可以在消耗某些计算能力的情况下，使用压缩的方法来节约这些资源。计算能力也变得越来越便宜，甚至要比其他一些资源更加丰富，比如带宽，因此压缩方法的使用也越来越流行。Python 可以很容易地让开发者的程序支持压缩，因为 Python 标准库包含几个专门用于压缩的模块。

由于 Python 提供了很多方法来处理压缩，因此对这些方法进行一些说明是很有帮助的。除了那些使用 zipfile 模块（ZIP 文件档案的标准格式）创建的文件，如果文件中包含使用 zlib 模块压缩的数据，则该文件不能自动与其他程序转换。开发者可以使用任何能够访问 InfoZip 的免费 zlib 压缩库的编程语言来编写自定义程序，以读取 Python 程序使用 zlib 模块创建的文件。但是，如果开发者确实需要与使用其他语言编写的程序相互交换压缩的数据，但是有选择不同压缩方法的机会，本书建议开发者使用 bzip2 模块（最好）、gzip 模块或 zipfile 模块。不过，在开发者想要压缩一些数据文件，并且这些数据文件是以开发者自己的某种私有格式创建的，而且除了开发者自己的应用程序，也不需要与任何其他程序相互交换这些压缩数据时，可以使用 zlib 模块。

gzip 模块

gzip 模块可以用来读取和写入与那些使用强大的 GNU 压缩程序 gzip 和 gunzip 处理的压缩文件兼容的文件。GNU 程序支持许多压缩格式，但是 gzip 模块只支持具有高有效性的自有 gzip 格式，通常在文件名后面增加一个扩展名.gz 来表示。gzip 模块提供了 GzipFile 类和一个 open 工厂函数。

表 10-8

GzipFile	`class GzipFile(`*`filename`*`=None, `*`mode`*`=None, `*`compresslevel`*`=9, `*`fileobj`*`=None)` 创建并返回一个类文件对象 f，该对象包装了文件或类文件对象 fileobj。在 fileobj 为 None 时，filename 必须是一个文件名字符串；GzipFile 将使用给定模式 mode（默认值为‘rb’）打开这个文件，并且 f 将包装这个结果文件对象。模式 mode 必须是‘ab’、‘rb’、‘wb’或 None。在模式为 None 时，如果能找到 fileobj 的模式，则 f 将使用该模式；否则，f 将使用默认的‘rb’模式。在 filename 为 None 时，如果能找到 fileobj 的文件名，则 f 将使用该文件名；否则，f 将使用空白字符‘’。compresslevel 是 1～9 的一个整数：1 表示要求最小压缩，但是最快的操作速度；9 表示要求最大压缩，需要耗费更多的计算能力。 类文件对象 f 可以将大多数方法委托给底层类文件对象 fileobj 使用，按照需要透明进行压缩。但是，f 不允许非连续地访问，因此 f 不支持 seek 和 tell 方法。如果 f 是使用非 None 的 fileobj 创建的，则调用 f.close 不会关闭该 fileobj。在 fileobj 是 StringIO.StringIO 的一个实例时，这是一个问题：开发者可以在 f.close 之后调用 fileobj.getvalue 以获得压缩的数据字符串。但是，这同时也表示开发者必须总是在调用 f.close 之后显式调用 fileobj.close。

续表

open	open(*filename*, *mode*='rb', *compresslevel*=9)
	与 GzipFile(filename, mode, compresslevel)类似，但是其 filename 是强制要求的，并且不能传递一个已经打开的 fileobj。
	一个使用 gzip 的例子
	假定开发者有一个函数 f(x)可以通过调用 x.write 和/或 x.writelines 将数据写入一个传递为 f 的参数的文本文件对象 x 中。这样，很容易让 f 将数据写入一个使用 gzip 压缩的文件中：
	`import gzip`
	`underlying_file = open('x.txt.gz', 'wb')`
	`compressing_wrapper = gzip.\`
	`GzipFile(fileobj=underlying_file, mode='wt')`
	`f(compressing_wrapper)`
	`compressing_wrapper.close()`
	`underlying_file.close()`
	这个示例首先打开了底层二进制文件 x.txt.gz，并使用 gzip.GzipFile 显式包装这个文件，最后，需要分别关闭每个对象。分别关闭每个对象是有必要的，因为这个示例使用了两种不同的模式：底层文件必须以二进制模式打开（任何对行结束符号的翻译都会产生一个无效的压缩文件），但是压缩包装器必须以文本模式打开，因为需要将\n 隐式翻译为 os.linesep。逐行读取一个压缩的文本文件——例如，在标准输出上显式该文件——的示例代码如下：
	`import gzip`
	`underlying_file = open('x.txt.gz', 'rb')`
	`uncompressing_wrapper = gzip.GzipFile(fileobj=`
	`underlying_file, mode='rt')`
	`for line in uncompressing_wrapper:`
	`print line,`
	`uncompressing_wrapper.close()`
	`underlying_file.close()`

bz2 模块

bz2 模块可以用来读取和写入与使用压缩程序 bzip2 和 bunzip2 处理的那些文件兼容的文件，这两个压缩程序通常可以实现比 gzip 和 gunzip 更好的压缩功能。bz2 模块提供了 BZ2File 类来实现透明文件的压缩和解压缩，还提供了 compress 和 decompress 类以压缩和解压缩内存中的数据字符串。bz2 模块还提供了对象以增量压缩和解压缩数据，这使得开发者可以处理数据流，一般数据流都太大，以至于不适合于一次在内存中处理。对于这样的高级功能，可以参考 Python 库的在线参考资料。

表 10-9

BZ2File	class BZ2File(*filename*=None, *mode*='r', *buffering*=0, *compresslevel*=9) 创建并返回一个类文件对象 f，对应于以 filename 命名的 bzip2 压缩文件，这个文件名必须是一个指定了文件的路径的字符串。mode 可以是 'r'，表示读取；'w' 表示写入；或者 'rU'，表示使用通用换行符翻译进行读取。在 buffering 为 0，也就是为默认值时，这个文件是非缓冲的。在 buffering 大于 0 时，这个将文件使用长度为 buffering 字节的缓冲区，这个长度将被舍入到一个合理的值。compresslevel 是 1～9 的一个整数：1 表示要求最小压缩，但是最快的操作速度；9 表示要求最大压缩，需要耗费更多的计算能力。 f 提供了内置文件对象的所有方法，包括 seek 和 tell。这样，f 是可搜寻的；但是，这个 seek 操作是模拟的，并且，在保证语义正确的情况下，有时候可能会极其地慢。
compress	compress(*s*, *level*=9) 压缩字符串 s 并返回由压缩的数据组成的字符串。level 是 1～9 的一个整数：1 表示要求最小压缩，但是最快的操作速度；9 表示要求最大压缩，需要耗费更多的计算能力。
decompress	decompress(*s*) 解压缩压缩的数据字符串 s，并返回由压缩前的数据组成的字符串。

tarfile 模块

tarfile 模块可以用来读取和写入 TAR 文件（与普通存档程序处理的文件兼容的档案文件，比如 tar），可以选择使用 gzip 或 bzip2 进行压缩。对于无效的 TAR 文件错误，tarfile 模块的函数将引发异常，这些异常都是异常类 tarfile.TarError 的实例。tarfile 模块提供了以下类和函数。

表 10-10

is_tarfile	is_tarfile(*filename*) 如果字符串 filename 指定的文件看起来是一个有效的 TAR 文件（可能被压缩过），则返回 True，通过这个文件的前几个字节来判断，否则，将返回 False。
TarInfo	class TarInfo(*name*='') TarFile 实例的 getmember 和 getmembers 方法将返回 TarInfo 的实例，提供了有关该档案文件的成员的信息。开发者还可以使用 TarFile 实例的 gettarinfo 方法创建一个 TarInfo 实例。TarInfo 实例 t 提供的最有用的属性是： linkname　一个字符串，如果 t.type 为 LNKTYPE 或 SYMTYPE，该字符串表示目标文件的名称； mode 　　t 标识的文件的许可权限和其他模式比特； mtime 　　t 标识的文件的最后修改时间； name 　　t 标识的文件的档案文件中的名称； size 　　t 标识的文件的字节大小（压缩前的）；

续表

TarInfo	type 文件类型，tarfile 模块的属性的多个常数之一（SYMTYPE 表示符号链接、REGTYPE 表示常规文件、DIRTYPE 表示目录，依此类推）。 要想检查 t 的类型，而不是测试 t.type，可以调用 t 的方法。t 的最常用方法是： *t*.isdir( ) 如果该文件是一个目录，则返回 True； *t*.isfile( ) 如果该文件是一个常规文件，则返回 True； *t*.issym( ) 如果该文件是一个符号链接，则返回 True。
open	`open(filename, mode='r', fileobj=None, bufsize=10240)` 创建并返回一个 TarFile 实例 f 以通过类文件对象 fileobj 读取或创建一个 TAR 文件。在 fileobj 是 None 时，filename 必须是一个用来命名文件的字符串；open 将使用给定的 mode（默认值为‘r’）打开这个文件，并且 f 将包装该结果文件对象。如果 f 是使用 fileobj 而不是 None 打开的，则调用 f.close 不会关闭 fileobj。在 fileobj 是 StringIO.StringIO 的一个实例时，f.close 的行为是很重要的：开发者可以在调用 f.close 之后调用 fileobj.getvalue，从而以字符串的形式获得存档的数据，而且有可能是压缩的数据。这个行为也意味着开发者必须在调用 f.close 之后显式调用 fileobj.close。 mode 可以是‘r’，这样可以按任何一种压缩方式（如果有的话）读取一个已有的 TAR 文件；可以是‘w’，这样可以写入一个新的 TAR 文件，或者截短和重写一个已有文件，而不进行压缩，或者‘a’，以添加到一个已有的 TAR 文件的后面，而不进行压缩。这个模式不支持将数据添加到压缩的 TAR 文件的后面。要想写入一个压缩的 TAR 文件，mode 可以是‘w:gz’，以用于 gzip 压缩，或者是‘w:bz2’，以用于 bzip2 压缩。特殊模式字符串‘r|’或‘w|’可以用于读取或写入压缩前的、不可搜寻的 TAR 文件（使用大小为 bufsize 字节的缓冲区），‘r|gz’、‘r|bz2’、‘w|gz’和‘w|bz2’可以用来以压缩的方式读取或写入这样的文件。 TarFile 的实例 f 还提供了以下这些方法。
add	`f.add(filepath, arcname=None, recursive=True)` 向档案文件 f 添加 filepath（可以是一个常规文件、目录或者符号链接）指定的文件。在 arcname 不是 None 时，arcname 将被用作档案文件的成员名称，用来代替 filepath。在 filepath 是一个目录时，add 将递归添加以这个目录为根的整个文件系统子树， 除非开发者将 recursive 的值传递为 False。
addfile	`f.addfile(tarinfo, fileobj=None)` 向档案文件 f 添加一个 tarinfo 标识的成员，也就是一个 TarInfo 实例（如果 fileobj 不是 None，则该数据是类文件对象 fileobj 的第一个 tarinfo.size 字节）。
close	`f.close()` 关闭档案文件 f。开发者必须调用 close，否则就会有一个不完整和不可用的 TAR 文件被遗留在硬盘上。最好使用 try/finally 语句实现强制终止化，参见第 6.1 节中的介绍。
extract	`f.extract(member, path='.')` 将 member（一个名称或者一个 TarInfo 实例）标识的档案文件的一个成员提取到 path 路径指定的目录（默认为当前目录）中的一个对应文件中。
extractfile	`f.extractfile(member)` 提取 member（一个名称或者一个 TarInfo 实例）标识的档案文件成员，并返回一个只读类文件对象，该对象具有 read、readline、readlines、seek 和 tell 方法。
getmember	`f.getmember(name)` 返回一个 TarInfo 实例，包含字符串 name 指定的档案文件成员的有关信息。

续表

getmembers	*f*.getmembers() 返回 TarInfo 实例的一个列表，档案文件 f 中的每个成员一个 TarInfo 实例，并按照档案文件本身的条目顺序排列。
getnames	*f*.getnames() 返回一个字符串列表，包含档案文件 f 中的每个成员的名称，并按照档案文件本身的条目顺序排列。
gettarinfo	*f*.gettarinfo(*name*=None, *arcname*=None, *fileobj*=None) 在 fileobj 不为 None，或者已有文件的路径为字符串 name 时，返回一个 TarInfo 实例，包含有关打开的文件对象 fileobj 的信息。在 arcname 不为 None 时，arcname 将被用作结果 TarInfo 实例的 name 属性。
list	*f*.list(*verbose*=True) 将档案文件 f 的以文本形式表示的目录输出到文件 sys.stdout。如果可选参数 verbose 为 False，则只输出该档案文件的所有成员的名称。

zipfile 模块

zipfile 模块可以用来读取和写入 ZIP 文件（也就是，与那些使用流行的压缩程序 zip 和 unzip、pkzip 和 pkunzip、WinZip 等处理的文件兼容的档案文件）。可以在 http://www.pkware.com/appnote.html 和 http://www.info-zip.org/pub/infozip/ 中找到有关 ZIP 文件的格式和功能的详细信息。为了执行似乎用 zipfile 模块处理的高级 ZIP 文件，开发者需要学习这些详细信息。如果开发者并不特别需要与其他使用 ZIP 文件标准的程序进行交互式操作，gzip 和 bz2 模块通常是用来处理压缩文件需要的更可取方法。

zipfile 模块不能处理带有附加注释、多硬盘 ZIP 文件，或者某些特殊.zip 档案文件成员的 ZIP 文件。这些特殊的.zip 档案文件成员使用了除被称为存储（stored，不进行压缩，直接被复制到档案文件中的文件）和默认压缩（deflated，使用 ZIP 格式的默认算法压缩的文件）的常规压缩类型之外的压缩类型。对于与无效.zip 文件有关的错误，zipfile 模块的函数将引发一些异常，这些异常都是异常类 zipfile.error 的实例。zipfile 模块还提供了如表 10-11 所示的类和函数。

10-11

is_zipfile	is_zipfile(*filename*) 如果字符串 filename 指定的文件看起来像是有效的 ZIP 文件（通过这个文件的前几个和最后几个字节来判断），则返回 True；否则，返回 False。
ZipInfo	class ZipInfo(*filename*='NoName', *date_time*=(1980, 1, 1, 0, 0, 0)) ZipFile 实例的 getinfo 和 infolist 方法可以返回 ZipInfo 的实例以提供有关档案文件的成员的信息。ZipInfo 实例 z 提供的最有用的属性是： Comment 　　档案文件成员的注释字符串； compress_size 　　档案文件成员的压缩数据的字节大小；

续表

ZipInfo	compress_type 记录档案文件成员的压缩类型的整数代码； date_time 一个包含 6 个整数的元组，记录了这个文件的最后修改时间：其中的项目是年、月、日（大于等于 1）、小时、分钟、秒（大于等于 0）； file_size 档案文件成员压缩前的数据的字节大小； filename 档案文件中的文件的名称。
ZipFile	`class ZipFile(filename, mode='r',compression=zipfile.ZIP_STORED)` 打开字符串 filename 指定的一个 ZIP 文件。mode 可以是‘r’，用于读取一个已有的 ZIP 文件；也可以是‘w’，用于写入一个新 ZIP 文件，或者截短和重写一个已有文件；或者是‘a’，用来添加到一个已有文件的后面。 在 mode 为‘a’时，filename 可以指定一个已有 ZIP 文件（在这种情况下，新成员将被添加到已有的档案文件中）或者一个已有的非 ZIP 文件。在后面这种情况下，将会创建一个新的 ZIP 类文件档案文件，并将其添加到已有文件的后面。后面这种情况的主要目的是可以让开发者创建一个自解压.exe 文件（也就是，可以在运行时自解压的 Windows 可执行文件）。然后，这个已有文件必须是解压的.exe 前缀的一个原始副本，就像 www.info-zip.org 和其他 ZIP 文件压缩工具提供的那样。 compression 是一个整数代码，可以是 zipfile 模块的两个属性之一。zipfile.ZIP_STORED 要求该档案文件不使用压缩；zipfile.ZIP_DEFLATED 要求该档案文件使用压缩的标准模式（也就是，.zip 文件中使用的最普通和最有效的压缩方法）。 ZipFile 的实例 z 提供了以下这些方法。
close	`z.close()` 关闭档案文件 z。要确保调用 close 方法，否则就会有一个不完整和不可用的 ZIP 文件被遗留在硬盘上。通常最好使用 try/finally 语句来执行这样的强制终止化，参见第 6.1 节中的介绍。
getinfo	`z.getinfo(name)` 返回一个 ZipInfo 实例，该实例提供了字符串 name 指定的档案文件成员的相关信息。
infolist	`z.infolist()` 返回 ZipInfo 实例的一个列表，档案文件 z 中的每个成员一个 ZipInfo 实例，并按照档案文件本身的条目顺序排列。
namelist	`z.namelist()` 返回一个字符串列表，包含档案文件 z 中的每个成员的名称，并按照档案文件本身的条目顺序排列。
printdir	`z.printdir()` 将档案文件 z 的以文本形式表示的目录输出到文件 sys.stdout。
read	`z.read(name)` 返回一个字符串，包含档案文件 z 中以字符串 name 命名的文件压缩前的字节数。必须使用‘r’或‘a’打开档案文件 z。在该档案文件不包含一个名为 name 的文件时，读取操作将引发一个异常。

续表

testzip	*z*.testzip() 读取并检查档案文件 z 中的文件。返回一个字符串，包含损坏的第一个档案文件成员的名称，如果该档案文件是完好无缺的，则返回 None。
write	*z*.write(*filename*, *arcname*=None, *compress_type*=None) 将字符串 filename 命名的文件写入到档案文件 z 中，其档案文件成员的名称为 arcname。在 arcname 为 None 时，写入操作将使用 filename 作为档案文件成员的名称。在 compress_type 为 None 时，写入操作将使用 z 的压缩类型；否则，compress_type 为 zipfile.ZIP_STORED 或 zipfile.ZIP_DEFLATED，并且指定如何压缩该文件。必须使用‘w’或‘a’模式打开档案文件 z。

writestr	*z*.writestr(*zinfo*, *bytes*)
	zinfo 必须是一个 ZipInfo 实例，至少要指定 filename 和 date_time。bytes 是字符串的字节长度。writestr 将使用 zinfo 和 bytes 指定的元数据和 bytes 中的数据向档案文件 z 添加一个成员。必须使用模式‘w’或‘a’打开档案文件 z。在内存中有数据，并且需要将该数据写入到 ZIP 文件的档案文件 z 中，使用 z.writestr 要比使用 z.write 更简单和更快。后者要求开发者首先将数据写入到硬盘中，然后删除无用的硬盘文件。下面的示例显示了这两种方案，每个方案都被封装到一个函数中，并且彼此之间呈多态性。
	`import zipfile`
	`def data_to_zip_direct(z, data, name):`
	`  import time`
	`  zinfo = zipfile.ZipInfo(name, time.localtime() [:6])`
	`  zinfo.compress_type = zipfile.ZIP_DEFLATED`
	`  z.writestr(zinfo, data)`
	`def data_to_zip_indirect(z, data, name):`
	`  import os`
	`  flob = open(name, 'wb')`
	`  flob.write(data)`
	`  flob.close()`
	`  z.write(name)`
	`  os.unlink(name)`
	`zz = zipfile.ZipFile('z.zip', 'w', zipfile.ZIP_DEFLATED)`
	`data = 'four score\nand seven\nyears ago\n'`
	`data_to_zip_direct(zz, data, 'direct.txt')`
	`data_to_zip_indirect(zz, data, 'indirect.txt')`
	`zz.close()`
	除了更快和更简洁之外，data_to_zip_direct 函数还要更便利一些，因为这个函数是在内存中运行的，并且不要求当前的工作目录是可写的，而 data_to_zip_indirect 函数有这个要求。当然，在开发者已经有数据在硬盘上的一个文件中，并且只想将该文件添加到档案文件中时，write 方法也是可用的。
	下面的示例给出了如何打印上一个示例创建的 ZIP 文件档案文件中包含的所有文件的列表，并打印了每个文件的名称和内容：
	`import zipfile`
	`zz = zipfile.ZipFile('z.zip')`
	`zz.printdir()`
	`for name in zz.namelist():`
	`  print '%s: %r' % (name, zz.read(name))`
	`zz.close()`

zlib 模块

zlib 模块可以让 Python 程序使用免费的 InfoZip zlib 压缩库(http://www.info-zip.org/pub/infozip/zlib/) 的 1.1.3 版本，及其以后版本。zlib 模块用于 gzip 和 zipfile 模块，但是也可以直接满足任何特殊压缩操作的需要。zlib 模块提供的最常使用的函数如表 10-12 所示。

表 10-12

compress	`compress(s, level=6)` 压缩字符串 s，并返回压缩数据后的字符串。level 是 1～9 的一个整数；1 表示要求最小压缩，但是最快的操作速度；9 表示要求最大可能地压缩，需要耗费更多的计算能力。
decompress	`decompress(s)` 解压缩压缩数据字符串 s，并返回压缩前的数据字符串。 zlib 模块还提供了一些方法以计算循环冗余校验（Cyclic-Redundancy Check，CRC）的校验和以检测压缩数据中的损坏情况。该模块还提供了一些对象以增量压缩和解压缩数据，这样可以处理那些太大，以至于不适合于一次在内存中处理的数据流。对于这样的高级功能，可以参考 Python 库的在线参考资料。

10.7 os 模块

os 模块是一个庇护（umbrella）模块，提供了各种操作系统的不同功能的适当的统一跨平台视图。这个模块提供了各种方法以创建和处理文件和目录，并且创建、管理和破坏过程。本节介绍了 os 模块中与文件系统相关的功能；参见第 14.7 节中介绍的过程相关功能。

os 模块提供了一个 name 属性，该属性是一个字符串，这个字符串标识了 Python 运行的平台的类型。name 的常见值是 'posix'（所有类型的类 UNIX 平台，包括 Mac OS X）、'nt'（所有类型的 32-位 Windows 平台）、'max'（老式 Mac 系统）和 'java'（Jython）。使用 os 提供的功能，开发者通常可以开发平台的特有功能，至少是一部分。不过，本书将介绍跨平台编程，而不是平台专用功能，因此，本书没有介绍只存在于某一种类型的平台的一部分 os 功能，也没有介绍平台专用模块。本书中介绍的所有功能至少可以在 'posix' 和 'nt' 平台上可用。但是，本书还是介绍了不同平台上提供的给定功能之间的任何区别。

OSError 异常

在请求操作系统失败时，os 将引发一个异常，该异常是 OSError 的一个实例。os 还提供了名为 os.error 的内置异常类 OSError。OSError 的实例提供了以下 3 个非常有用的属性：

errno

操作系统错误的数字错误代码；

strerror

概要描述错误的字符串；

filename

导致操作失败的文件的名称（仅用于与文件相关的函数）。

os 函数还可以引发其他标准异常，通常是 TypeError 或 ValueError，在导致错误的原因是使用无效参数或值调用这些函数时，这样甚至不会触及底层操作系统功能。

errno 模块

errno 模块为错误代码编号提供了符号名称。要想基于错误代码有选择地处理可能的系统错误，使用 errno 可以增强程序的可移植性和可读性。例如，下面给出了如何处理“file not found”错误，并传播所有其他类型的错误的示例：

```
try: os.some_os_function_or_other()
except OSError, err:
    import errno
    # 检查“file not found”错误，重新引发其他错误情况
 if err.errno != errno.ENOENT: raise
    # 继续可以处理的特殊情况
    print "Warning: file", err.filename, "not found -- continuing"
```

errno 还提供了一个名为 errorcode 的字典：键是错误代码编号，而对应的名称就是错误名称，这些名称都是像‘ENOENT’这样的字符串。作为某个 os.error 实例 err 的诊断的一部分，显示 errno.errorcode[err.errno]通常可以让诊断更清晰，并且可以让要在特定平台上进行专门地诊断的读者更好理解。

10.8 文件系统操作

使用 os 模块，开发者可以用各种方法操作文件系统：创建、复制和删除文件和目录、比较文件，以及检查有关文件和目录的文件系统信息。本节将详细列举 os 模块中用于这些目的的属性和方法，并介绍一些对文件系统进行操作的相关模块。

os 模块的路径字符串属性

文件或目录是通过一个被称为路径（path）的字符串来标识的，路径的语法取决于平台。在类 UNIX 和 Windows 平台上，Python 采用了 UNIX 语法来表示路径，使用斜线符号（/）作为目录分隔符。在非类 UNIX 平台上，Python 还可以采用平台专用路径语法。特别是，在 Windows 平台上，可以使用反斜线符号（\）作为分隔符。但是，在字符串字面

常量中，需要将每个反斜线符号更改为双反斜线符号（\\），或者使用第 4.1 节中介绍的原生字符串语法；但是这样也就不必要地失去了可移植性。UNIX 路径语法是很便捷的，并且可以在任何位置使用，因此本书强烈建议开发者总是使用 UNIX 路径语法。在本章的其余部分，为了简短说明，本书假定在所有的解释和示例中都使用 UNIX 路径语法。

os 模块提供了一些属性，这些属性提供了有关当前平台上的路径字符串的详细信息。开发者通常必须使用第 10.8 节中介绍的高层路径处理操作，而不是基于这些属性的底层字符串操作。但是，这些属性有时候是非常有用的。

curdir

表示当前目录的字符串（在 UNIX 和 Windows 平台上都使用 '.'）；

defpath

程序的默认搜索路径，如果平台的环境中缺少 PATH 环境变量，则使用该默认搜索路径；

linesep

用来终止文本行的字符串（在 UNIX 平台上使用'\n'，在 Windows 平台上使用'\r\n'）；

extsep

将文件名的扩展部分从文件名的其余部分分隔开的字符串（在 UNIX 和 Windows 平台上都使用 '.'）；

pardir

表示父目录的字符串（在 UNIX 和 Windows 平台上都使用 '..'）；

pathsep

路径列表中的路径之间的分隔符，比如用于环境变量 PATH 的分隔符（在 UNIX 平台上使用 ':'；在 Windows 平台上使用 ';'）；

sep

路径元素的分隔符（在 UNIX 平台上使用 '/'；在 Windows 平台上使用 '\\'）。

权限

在类 UNIX 平台中，有 9 个比特与每个文件或目录的权限相关：文件的所有者、所有者所在的组和其他人各三个比特，这些比特用来指示该文件和目录是否可读、可写和可执行。这 9 个比特被称为文件的权限比特（permission bit），这些比特是该文件的模式（mode，一个比特字符串，还包括用来描述该文件的其他比特）的一部分。这些比特通常是按八进制符号显示的，每个数字 3 个比特。例如，模式 0664 表示一个可以被其所有者和所有者所在的组读取和写入，并且可以被其他人读取，但是不能写入的文件。在类 UNIX 系统上的任何过程创建一个文件或目录时，操作系统可以对指定的模

式应用一个被称为过程的 umask 的比特屏蔽，umask 可以用来删除某些权限比特。

非类 UNIX 平台是以非常不同的方式来处理文件和目录权限的。不过，Python 用来处理文件权限的标准库中的函数可以按照上一段落中介绍的类 UNIX 方案接受一个模式（mode）参数。每个平台上的实现都是按照适合于给定平台的方式来映射这 9 个权限比特的。例如，在只区分只读和读/写文件，而不区分文件的所有者的 Windows 版本中，文件的权限比特显示为 0666（读/写）或 0444（只读）。在这种平台上，在创建一个文件时，该实现操作只需要查看比特 0200，如果这个比特为 0，则该文件是可读和可写的，如果这个比特为 1，则该文件是只读的。

os 模块的文件和目录函数

os 模块提供了几个函数以查询和设置文件和目录的状态。

表 10-13

access	access(*path*, *mode*) 如果文件的路径 path 具有整数 mode 中编码的所有权限时，返回 True；否则，返回 False。mode 可以是 os.F_OK 以测试文件的存在，或者一个或多个名为 os.R_OK、os.W_OK 和 os.X_OK 的整数常数（如果用到多个，则使用按位或运算符 | 连接这些整数常数）以测试读取、写入和执行文件的权限。 access 并不使用标准方式来解释其 mode 参数，参见第 10.8 节。只有在这个特殊过程的真实用户和组标识符具有对该文件要求的权限时，才会使用 access 进行测试。如果开发者需要更详细地学习一个文件的权限比特，可以参见第 10.8 节中介绍的 stat 函数。
chdir	chdir(*path*) 将当前的工作目录设置为 path。
chmod	chmod(*path*, *mode*) 按照整数 mode 中的编码，更改文件 path 的权限。mode 可以是零个或多个 os.R_OK、os.W_OK 和 os.X_OK（如果是多于一个，则使用按位或运算符 | 连接这些模式比特）以设置读取、写入和执行的权限。在类 UNIX 平台上，mode 还可以具有更丰富的比特模式（参见第 10.8 节），这样可以为用户、组和其他人指定不同的权限。
getcwd	getcwd() 返回当前的工作目录的路径。
listdir	listdir(*path*) 返回一个列表，其中的项目是目录 path 中可以找到的所有文件和子目录的名称。返回的列表是按任意顺序排列的，并且不包含特殊目录名称‘.’（当前目录）和‘..’（父目录）。 dircache 模块还提供了一个名为 listdir 的函数，该函数的使用就像 os.listdir 一样，还包含两个增强功能。首先，dircache.listdir 将返回一个排序列表。此外，dircache 可以缓存该函数返回的列表，这样，对相同目录列表的重复请求就会得到更快的执行，如果在此期间这个目录的内容没有改变。dircache 将自动检测更改：只要调用了 dircache.listdir，同时就得到了一个可以反映这个目录中的内容的列表。

续表

<table>
<tr><td>makedirs,
mkdir</td><td>makedirs(path, mode=0777) mkdir(path, mode=0777)
makedirs 可以创建作为 path 的一部分，但是还不存在的所有目录。mkdir 只能创建 path 最右边的目录，如果 path 中前面的目录还不存在，则引发 OSError 异常。这两个函数都使用了 mode 作为其创建的目录的权限比特。如果创建失败，或者 path 指定的文件或目录已经存在，则这两个函数都将引发 OSError 异常。</td></tr>
<tr><td>remove,
unlink</td><td>remove(path) unlink(path)
删除 path 指定的文件（参见第 10.8 节中介绍的 rmdir 函数以删除一个目录）。unlink 与 remove 具有相同的功能。</td></tr>
<tr><td>removedirs</td><td>removedirs(path)
对 path 中包含的目录从右往左循环，并删除每个目录。当删除操作引发一个异常时，循环将会结束，这通常是因为某个目录非空。只要 removedirs 函数删除了至少一个异常，就不会传播异常。</td></tr>
<tr><td>rename</td><td>rename(source, dest)
将 source 指定的文件或目录重命名为 dest。</td></tr>
<tr><td>renames</td><td>renames(source, dest)
与 rename 函数类似，区别在于 renames 将尝试创建 dest 所需的所有中间目录。在重命名之后，renames 将使用 removedirs 尝试从路径 source 中删除空目录。该函数不传播任何结果异常；如果 source 的起始目录在进行重命名之后没有变成空目录，则该函数不会产生错误。</td></tr>
<tr><td>rmdir</td><td>rmdir(path)
删除 path 指定的空目录（如果删除失败，特别是，如果该目录不为空，则引发 OSError 异常）</td></tr>
<tr><td>stat</td><td>stat(path)
返回类型为 stat_result 的一个值 x，这个值提供了有关文件或子目录 path 的 10 个信息项目。通常并不建议通过项目的数字索引访问这些项目，因为这样会导致结果代码不具备很好的可读性。下表列出了 stat_result 实例的属性和对应项目的含义。
stat_result 实例的项目（属性）
<table>
<tr><th>项目索引</th><th>属性名称</th><th>含　义</th></tr>
<tr><td>0</td><td>st_mode</td><td>保护和其他模式比特</td></tr>
<tr><td>1</td><td>st_ino</td><td>Inode 编号</td></tr>
<tr><td>2</td><td>st_dev</td><td>设备 ID</td></tr>
<tr><td>3</td><td>st_nlink</td><td>硬链接数量</td></tr>
<tr><td>4</td><td>st_uid</td><td>所有者的用户 ID</td></tr>
<tr><td>5</td><td>st_gid</td><td>所有者的组 ID</td></tr>
<tr><td>6</td><td>st_size</td><td>字节大小</td></tr>
<tr><td>7</td><td>st_atime</td><td>上次访问时间</td></tr>
<tr><td>8</td><td>st_mtime</td><td>上次修改时间</td></tr>
<tr><td>9</td><td>st_ctime</td><td>上次状态改变时间</td></tr>
</table></td></tr>
</table>

stat	例如，要想打印文件 path 的字节大小，可以使用下面任意一种方法：
	`import os`
	`print os.path.getsize(path)`
	`print os.stat(path)[6]`
	`print os.stat(path).st_size`
	时间值是从新纪元时间（epoch）开始，以秒计算的，参见第 12 章（在大多数平台上使用的是整型 int；在非常老版本的 Macintosh 上使用的是浮点型 float）。平台不能为一个使用虚假（dummy）值的项目给出一个有意义的值。

tempnam, tmpnam	`tempnam(dir=None, prefix=None) tmpnam()` 返回一个可以用作新临时文件名称的绝对路径。注意：tempnam 和 tmpnam 在程序中的安全性是很弱的。要尽量避免使用这些函数，而是使用标准库模块 tempfile，参见第 10.3 节
utime	`utime(path, times=None)` 设置文件或目录 path 的访问时间（accessed）和修改时间（modified）。如果 times 为 None，则 utime 将使用当前时间。否则，times 必须是一个按顺序（访问时间，修改时间）排列的数值对（从新纪元时间开始，按秒计算，参见第 12 章）。

walk	`walk(top, topdown=TRue, onerror=None)`
	生成器将为根为目录 top 的树中的每个目录生成一个项目。在 topdown 为 True 时，也就是默认值时，walk 将访问从树的根向下的目录；在 topdown 为 False 时，walk 将访问从树的叶子向上的目录。在 onerror 为 None 时，walk 可以捕获并忽略遍历树期间引发的任何 OSError 异常。否则，onerror 必须是一个函数；walk 可以捕获遍历树期间引发的任何 OSError 异常，并将其传递为 onerror 函数调用的唯一参数，该函数可以处理这个异常、忽略这个异常，或者引发这个异常以终止树的遍历，并传播这个异常。
	每个项目 walk 生成的是一个包含 3 个子项目的元组：dirpath，一个字符串，表示该目录的路径；dirnames，一个由子目录的名称组成的列表，这些子目录是该目录的直接子目录（不包括特殊目录‘.’和‘..’）；还有 filenames，一个由文件的名称组成的列表，这些文件直接包含在该目录中。如果 topdown 为 True，则可以在原地改变列表 dirnames（删除某些项目和/或重新对其他项目进行排序）以影响根在 dirpath 中的子树的遍历；walk 将按照 dirnames 中剩下的子目录的顺序对这些子目录进行迭代。如果 topdown 为 False，则这样的迭代没有任何效果（在这种情况下，walk 已经在访问其当前目录并生成其中的项目时访问了所有子目录）。
	os.walk 的典型使用可能就是打印树中所有文件（不包括子目录）的路径，并跳过树中根目录的名称以‘.’开始的文件路径：
	`import os`
	`for dirpath, dirnames, filenames in os.`
	`walk(tree_root_dir):`
	`# 在原地改变 dirnames 以跳过名为‘.something’的子目录`
	`dirnames[:] = [d for d in dirnames if not d.`
	`startswith('.')]`
	`# 打印每个文件的路径`
	`for name in filenames:`
	`print os.path.join(dirpath, name)`
	如果参数 top 是一个相对路径，则对 os.walk 的结果进行循环操作的循环体不能更改工作目录，这样可能会导致未定义的行为。os.walk 本身不能更改工作目录。要想将任何名称 x（dirnames 或 filenames 中的一个项目）转换为一个路径，可以使用 os.path.join(top, dirpath, x)。

os.path 模块

os.path 模块提供了一些函数以分析和转换路径字符串。要想使用这个模块，可以导入该模块（import os.path）。但是，如果直接 import os，也可以访问 os.path 模块及其所有属性。

表 10-14

abspath	`abspath(path)`
	返回一个等于 path 的规范化的绝对路径字符串，就像下面的代码一样：
	`os.path.normpath(os.path.join(os.getcwd(), path))`
	例如，os.path.abspath(os.curdir)与 os.getcwd( )相同。

basename	`basename(path)` 返回 path 的基础名称部分，就像 os.path.split(path)[1]一样。例如，os.path.basename('b/c/d.e')将返回‘d.e’。
commonprefix	`commonprefix(list)` 接受一个字符串列表，并返回作为列表中所有项目的前缀的最长字符串。与 os.path 中的所有其他函数不一样，commonprefix 可以对任意字符串进行操作，而不仅仅是对路径。例如，os.path.commonprefix(['foobar', 'foolish'])将返回‘foo’。
dirname	`dirname(path)` 返回 path 的目录部分，就像 os.path.split(path)[0]一样。例如，os.path.dirname ('b/c/d.e')将返回‘b/c’。
exists	`exists(path)` 在 path 指定一个已有的文件或目录时，将返回 True；否则，返回 False。换句话讲，os.path.exists(x)与 os.access(x, os.F_OK)相同。

expandvars	`expandvars(path)`
	返回字符串 path 的一个副本，其中形式为$name 或${name}的每个子字符串将被替换为环境变量指定的值。例如，如果环境变量 HOME 被设置为/u/alex，则下面的代码：
	`import os`
	`print os.path.expandvars('$HOME/foo/')`
	将会输出/u/alex/foo/。

getatime, getmtime, getsize	`getatime(path) getmtime(path) getsize(path)` 这几个函数都可以返回 os.stat(path)的结果中的一个属性：分别是，st_atime、st_mtime 和 st_size。参见第 10.8 节中介绍的 stat 函数以了解有关这几个属性的更详细信息。
isabs	`isabs(path)` 在 path 为绝对路径时，返回 True。在一个路径是以斜线符号（/）开始，或者，在某些非类 UNIX 平台上，该路径是以一个驱动器盘符开始，后面带 os.sep 时，该路径为绝对路径。在 path 不是绝对路径时，isabs 将返回 False。
isfile	`isfile(path)` 在 path 指定到一个已有的常规文件时，将返回 True（但是，在 UNIX 中，isfile 还可以针对符号链接），否则，将返回 False。

续表

isdir	isdir(*path*) 在 path 制定到一个已有目录时，将返回 True（但是，在 UNIX 中，isdir 还可以针对符号链接）；否则，将返回 False。
islink	islink(*path*) 在 path 指定到一个符号链接时，将返回 True；否则（对于不支持符号链接的平台），islink 将返回 False。
ismount	ismount(*path*) 在 path 指定到一个挂接点（mount point）时，将返回 True；否则（对于不支持挂接点的平台），ismount 将返回 False。

join	join(*path*, **paths*)
	返回一个将参数字符串与当前平台上的相应路径分隔符连接在一起的字符串。例如，在 UNIX 平台上，可以使用单个斜线符号 / 分隔邻近的组成部分。如果任何一个参数是一个绝对路径，join 将忽略之前的所有组成部分。例如：
	`print os.path.join('a/b', 'c/d', 'e/f')`
	`# 在 Unix 平台上，打印：a/b/c/d/e/f`
	`print os.path.join('a/b', '/c/d', 'e/f')`
	`# 在 Unix 平台上，打印：/c/d/e/f`
	对 os.path.join 的第二次调用将忽略其第一个参数‘a/b’，因为其第二个参数‘/c/d’是一个绝对路径。

normcase	normcase(*path*) 返回 path 的一个副本，其大小写使用当前平台的规范定义。在区分大小写的文件系统中（通常在类 UNIX 系统中），返回未经改变的 path。在不区分大小写的文件系统中（通常在 Windows 系统中），返回的字符串中的所有字母都是小写的。在 Windows 中，normcase 还将把每个 / 转换为 \。
normpath	normpath(*path*) 返回一个等于 path 的规范化路径名称，删除冗余的分隔符和路径导航指示。例如，在 UNIX 平台上，当 path 为‘a//b’、‘a/./b’或‘a/c/../b’时，normpath 将返回‘a/b’。normpath 可以让路径分隔符适合于当前平台。例如，在 Windows 平台上，分隔符将变成 \。
split	split(*path*) 返回一个字符串对（dir, base），也就是说，join(dir, base)等于 path。base 是路径名称的最后一个组成部分，不会包含一个路径分隔符。如果 path 以一个分隔符结束，则 base 为‘ ’。dir 是 path 的前导部分，一直到最后一个路径分隔符，相当于剪切掉拖尾的分隔符。例如，os.path.split('a/b/c/d')将返回字符串对('a/b/c', 'd')。
splitdrive	splitdrive(*path*) 返回一个字符串对（drv, pth），也就是说，drv+pth 等于 path。drv 可以是一个驱动器盘符或‘ ’。在不支持驱动器盘符的平台上，drv 总是‘ ’，比如，所有的类 UNIX 系统。例如，在 Windows 平台上，os.path.splitdrive('c:d/e')将返回字符串对('c:', 'd/e')。

续表

splitext	splitext(*path*) 返回一个字符串对（root, ext），这样 root+ext 等于 path。ext 可以是‘’或者是一个以‘.’开始，并且不再有其他‘.’或路径分隔符的字符串。例如，os.path.splitext('a.a/b.c.d')将返回字符串对('a.a/b.c', '.d')。
walk	walk(*path, func, arg*) 为根为目录 path 的树中的每个目录调用函数 func(arg, dirpath, namelist)，从 path 本身开始。这个函数是很陈旧的，并且很难使用；因此可以改为使用生成器 os.walk，参见第 10.8 节中介绍的 walk 函数。

stat 模块

os.stat 函数（参见第 10.8 节中介绍的 stat 函数）可以返回一个 stat_result 实例，其中的项目为索引、属性名称，这些结果的含义请参见表 10-2。stat 模块提供了与 stat_result 的属性具有相同名称的属性，只是将这些名称改为大写，并且对应的值就是相应的项目索引。

stat 模块更有趣的内容就是包含一些函数，这些函数可以检查 stat_result 实例的 st_mode 属性以确定文件的类型。os.path 还提供了一些函数来完成这些任务，这些函数直接对文件的 path 进行操作。stat 提供的函数在对相同的文件执行几种测试的时候要更快一些：这些函数只要求在一系列测试的开始调用一次 os.stat 函数，而 os.path 中的函数将在每次测试时隐式要求操作系统提供相关信息。如果 mode 表示一个给定类型的文件，则每个函数都返回 True；否则，将返回 False。

S_ISDIR(*mode*)

这个文件是一个目录吗?

S_ISCHR(*mode*)

这个文件是一个字符类型的特殊设备文件吗?

S_ISBLK(*mode*)

这个文件是一个块类型的特殊设备文件吗?

S_ISREG(*mode*)

这个文件是一个普通文件（而不是一个目录、特殊设备文件等）吗?

S_ISFIFO(*mode*)

这个文件是一个 FIFO（也就是，一个“命名管道”）吗?

S_ISLNK(*mode*)

这个文件是一个符号链接吗?

S_ISSOCK(*mode*)

这个文件是一个 UNIX 域套接字吗?

除了 stat.S_ISDIR 和 stat.S_ISREG 之外，其他函数都只有在类 UNIX 系统上才有意义，因为其他平台不会将像设备和套接字这样的特殊文件像普通文件一样保存在相同的文件系统中，也不会像类 UNIX 系统所做的那样，直接提供符号链接。

stat 模块提供了另外两个函数，可以提取文件的 mode 中的相关部分（x.st_mode，为 os.stat 函数的某些结果 x）。

表 10-15

S_IFMT	S_IFMT(*mode*) 返回 mode 中描述文件类型的比特（也就是，这些比特是通过 S_ISDIR 和 S_ISREG 等函数检查的）。
S_IMODE	S_IMODE(*mode*) 返回 mode 中可以被函数 os.chmod 设置的那些比特(也就是,权限比特和类 UNIX 平台上的其他特殊比特，比如 set-user-id 标记)。

filecmp 模块

filecmp 模块提供了一些函数来比较文件和目录。

表 10-16

cmp	cmp(*f1*, *f2*, *shallow*=True) 比较路径字符串 f1 和 f2 指定的文件。如果这两个文件看起来是相同的，cmp 将返回 True；否则，将返回 False。如果 shallow 为 True，则如果文件的 stat 元组是相等的，可以认定文件是“相等”的。如果 shallow 是 False，则 cmp 将读取并比较 stat 元组相等的文件。
cmpfiles	cmpfiles(*dir1*, *dir2*, *common*, *shallow*=True) 对序列 common 进行循环操作。common 中的每个项目都是一个字符串，每个字符串指定了 dir1 和 dir2 这两个目录中都有的一个文件。cmpfiles 将返回一个元组，其中的项目是 3 个字符串列表：(equal, diff, errs)。equal 是由两个目录中相等的文件名组成的列表，diff 是由两个目录中不同的文件名组成的列表，而 errs 是由不能进行比较的文件名组成的列表（因为这个文件在两个目录中都不存在，或者没有读取这个文件的权限）。参数 shallow 与 cmp 函数中的参数具有相同的含义。
dircmp	class dircmp(*dir1*, *dir2*, *ignore*=('RCS','CVS','tags'), *hide*=('.','..')) 创建一个新目录比较实例对象，比较 dir1 和 dir2 指定的目录，忽略 ignore 中列出的名称，并隐藏 hide 中列出的名称。dircmp 实例 d 包含 3 个方法： *d*.report( ) 将 dir1 和 dir2 之间的比较结果输出到 sys.stdout； *d*.report_partial_closure( ) 将 dir1 和 dir2，及其直接子目录的比较结果输出到 sys.stdout；

续表

dircmp	*d*.report_full_closure( ) 将 dir1 和 dir2，及其所有递归子目录的比较结果输出到 sys.stdout。 dircmp 实例 d 提供了几个可以即时进行计算的属性（也就是，当且仅当在需要时进行计算，这要感谢_ _getattr_ _特殊方法），因此，使用 dircmp 实例不会造成不必要的开销。d 的属性包括： *d*.common dir1 和 dir2 中都有的文件和子目录； *d*.common_dirs dir1 和 dir2 中都有的子目录； *d*.common_files dir1 和 dir2 中都有的文件； *d*.common_funny dir1 和 dir2 中都有的名称，对于这个名称，os.stat 会对报告一个错误，或者对两个目录中的不同版本返回不同的类型； *d*.diff_files dir1 和 dir2 中都有，但是内容不同的文件； *d*.funny_files dir1 和 dir2 中都有，但是不能进行比较的文件； *d*.left_list dir1 中的文件和子目录； *d*.left_only 在 dir1 中，但是不在 dir2 中的文件和子目录； *d*.right_list dir2 中的文件和子目录； *d*.right_only 在 dir2 中，但是不在 dir1 中的文件和子目录； *d*.same_files dir1 和 dir2 中都有，并且具有相同内容的文件； *d*.subdirs 一个字典，其键是 common_dirs 中的字符串；对应的值是每个子目录的 dircmp 实例。

shutil 模块

shutil 模块（shell utilities 的缩写）提供了一些函数以复制和移动文件，并且可以删除整个目录树。除了提供一些直接有用的函数，标准 Python 库中的源文件 shutil.py 还是一个非常好的示例，可以告诉开发者如何使用许多 os 函数。

表 10-17

copy	copy(*src, dst*) 复制文件 src 的内容，创建或覆盖文件 dst。如果 dst 是一个目录，则该函数的目的文件就是目录 dst 中与 src 具有相同基础名称的文件。copy 还将复制权限比特，但是不包括上次访问时间和修改时间。
copy2	copy2(*src, dst*) 与 copy 类似，但是还复制上次访问和修改的时间。
copyfile	copyfile(*src, dst*) 只复制文件 src 的内容（不包括权限比特，也不包括上次访问和修改时间），创建或覆盖文件 dst。
copyfileobj	copyfileobj(*fsrc, fdst, bufsize*=16384) 将所有字节从文件对象 fsrc 复制到文件对象 fdst，fsrc 文件对象必须是以可读模式打开，而 fdst 必须是以可写模式打开的。如果 bufsize 大于 0，一次复制不超过 bufsize 个字节。参见第 10.3 节介绍的文件对象。
copymode	copymode(*src, dst*) 将文件或目录 src 的权限比特复制到文件或目录 dst 上。src 和 dst 都必须存在。不更改 dst 的内容，也不更改 dst 的文件或目录状态。
copystat	copystat(*src, dst*) 将文件或目录 src 的权限比特，以及上次访问和修改时间复制到文件或目录 dst 上。src 和 dst 必须都存在。不更改 dst 的内容，也不更改 dst 的文件或目录状态。
copytree	copytree(*src, dst, symlinks*=False) 将以 src 为根的目录树复制到 dst 指定的目的目录中。dst 不能是已经存在的：copytree 将创建 dst。copytree 将使用 copy2 函数复制每个文件。在 symlinks 为 True 时，copytree 在源目录树中找到符号链接时将在新目录树中创建符号链接。在 symlinks 为 False 时，copytree 将按照每个符号链接的定义，使用该链接的名称复制链接到的文件。在不具备符号链接概念的平台上，比如 Windows，copytree 将忽略参数 symlinks。
move	move(*src, dst*) 将文件或目录从 src 移动到 dst。首先尝试 os.rename。然后，如果操作失败（因为 src 和 dst 在各自的文件系统中，或者因为 src 和 dst 是已经存在的文件），则将 src 复制到 dst 中（使用 copy2 复制文件，使用 copytree 复制目录），然后删除 src（使用 os.unlink 删除文件，使用 rmtree 删除目录）。
rmtree	rmtree(*path, ignore_errors*=False, *onerror*=None) 删除以 path 为根的目录树。在 ignore_errors 为 True 时，rmtree 将忽略所有错误。在 ignore_errors 为 False，并且 onerror 为 None 时，任何错误都将引发一个异常。在 onerror 不为 None 时，onerror 必须是一个有 3 个参数的调用对象：func、path 和 excp。func 是引发异常的函数（os.remove 或 os.rmdir）、path 是传递给 func 的路径，而 excp 是 sys.exc_info( )返回的信息元组。如果 onerror 引发任何异常 x，rmtree 将会终止运行，并将传播异常 x。

文件描述符操作

os 模块提供了一些函数来处理文件描述符，文件描述符是操作系统用作不透明句柄的整

数，用来引用打开的文件。Python 文件对象（参见第 10.3 节）几乎总是要比输入/输出（I/O）任务更好使用一些，但是有时候在文件描述符层面进行操作可以让开发者更快或更优美地执行某些操作。请注意，文件对象和文件描述符不能以任何方式进行相互转换。

开发者可以通过调用 n=f.fileno( ) 获得 Python 文件对象 f 的文件描述符，还可以通过调用 f=os.fdopen(fd)在打开的文件描述符 fd 上包装一个新的 Python 文件对象 f。在类 UNIX 和 Windows 平台上，某些文件描述符是在处理开始时预先分配的：0 是用于程序的标准输入的文件描述符，1 是用于程序的标准输出的文件描述符，2 是用于程序的标准错误的文件描述符。

os 提供了如表 10-18 所示的函数来使用文件描述符。

表 10-18

close	`close(fd)` 关闭文件描述符 fd。
dup	`dup(fd)` 返回复制了文件描述符 fd 的文件描述符。
dup2	`dup2(fd, fd2)` 将文件描述符 fd 复制到文件描述符 fd2。如果文件描述符 f2 已经是打开的，dup2 将首先关闭 fd2。
fdopen	`fdopen(fd, mode='r', bufsize=-1)` 返回一个包装文件描述符 fd 的 Python 文件对象。mode 和 bufsize 与 Python 的内置 open 函数中的 mode 和 bufsize 具有相同的含义，参见第 10.3 节。
fstat	`fstat(fd)` 返回一个 stat_result 实例 x，其中包含使用文件描述符 fd 打开的文件的相关信息。表 10-13 介绍了 x 的内容。
lseek	`lseek(fd, pos, how)` 将文件描述符 fd 的当前位置设置为带符号整数字节偏移量 pos 并返回从文件起始位置到当前位置的结果字节偏移量。how 表示参考点（0 点）。在 how 为 0 时，参考点就是文件的起始位置；在 how 为 1 时，参考点就是当前位置；在 how 为 2 时，参考点是文件的结束位置。特别是，lseek(fd, 0, 1)将返回从文件的起始位置开始到当前位置的字节偏移量，而不会影响当前位置。普通硬盘文件支持搜寻功能；对一个不支持搜寻的文件（例如，一个打开用来输出到终端的文件）调用 lseek 将引发一个异常。在 Python 2.5 中，os 模块具有名为 SEEK_SET、SEEK_CUR 和 SEEK_END 的属性，对应的值分别为 0、1 和 2，为了具有更好的可读性，可以使用这些属性，而不是纯粹的整数常数。
open	`open(file, flags, mode=0777)` 返回一个文件描述符，打开或者创建一个由字符串 file 指定的文件。如果 open 将创建这个文件，则使用 mode 作为该文件的权限比特。flags 是一个整型值，并且通常是由一个或多个以下的 os 属性执行按位与操作来获得的： O_RDONLY O_WRONLY O_RDWR 分别表示按只读、只写或读/写方式打开文件（这几个属性是互斥的：flags 中只能包含其中的一个属性）；

续表

open	O_NDELAY O_NONBLOCK 　　如果平台支持无阻塞（无延时）模式，则按该模式打开文件； O_APPEND 　　将任何新数据都添加到文件的以前内容的后面； O_DSYNC O_RSYNC O_SYNC O_NOCTTY 　　如果平台支持同步模式，则设置相应的同步模式； O_CREAT 　　如果文件不存在，则创建该文件； O_EXCL 　　如果文件已经存在，则引发一个异常； O_TRUNC 　　丢弃文件的以前内容（与 O_RDONLY 不兼容）； O_BINARY 　　在非 UNIX 平台上按二进制模式打开文件，而不是按文本模式打开文件（对类 UNIX 平台没什么害处，也不会产生任何影响）。
pipe	`pipe()` 创建一个管道，并返回文件描述符对（r, w），可以分别打开以进行读取和写入
read	`read(fd, n)` 从文件描述符 fd 读取最多 n 个字节，并将读取的字节返回为一个普通字符串。在当前只能从该文件中读取 m 个字节，且 m<n 时，读取并返回 m 个字节。特别是，在当前不能从该文件读取任何比特时，将返回空白字符串，这通常是因为该文件已经结束。
write	`write(fd, s)` 将普通字符串中的所有字节写入到文件描述符 fd 中，并返回写入的字节数量（也就是 len(s)）。

10.9 文本输入和输出

Python 可以以文件对象的方式向开发者的程序提供非图形用户界面（GUI）的文本输入和输出通道，因此开发者可以使用文件对象的方法（参见第 10.3 节）来管理这些通道。

标准输出和标准错误

sys 模块（参见第 8.3 节）具有属性 stdout 和 stderr，这些属性都是可写的文件对象。除

非开发者使用的是 shell 重定向或管道，否则这些流将连接到运行脚本的终端上。当前，实际的终端是很少见的：所谓的“终端”通常就是一个支持文本 I/O 的屏幕窗口（例如，Windows 上的“命令提示符”控制台窗口或 UNIX 上的 xterm 窗口）。

sys.stdout 和 sys.stderr 之间的区别是一个习惯问题。sys.stdout 被称为脚本的标准输出，也就是程序输出结果的位置。sys.stderr 被称为脚本的标准错误，也就是输出错误消息的位置。将结果与错误消息分开可以帮助开发者有效地使用 shell 重定向。Python 遵循了这个惯例，可以使用 sys.stderr 输出错误和警告。

print 语句

要将结果输出到标准输出的程序通常需要写入到 sys.stdout。Python 的 print 语句（参见第 4.9 节）可以作为 sys.stdout.write 的一个很便利的可选方案。

print 可以很好地用于程序开发期间使用的非正式输出，这样可以帮助开发者调试代码。为了生成输出结果，通常需要用到比 print 所能提供的更多的格式化控制。开发者可能需要控制间距、字段宽度、浮点型值得小数位数等。在这种情况下，可以使用字符串格式化操作符%（参见第 9.3 节）将输出结果准备为字符串，然后输出该字符串，通常可以使用适当的文件对象的 write 方法来输出字符串（还可以将格式化的字符串传递到 print，但是 print 将添加空格和换行符，而 write 方法什么都不会添加，因此可以更容易地准确控制输出的结果）。

在开发者想要将多个 print 语句的输出结果直接输出到另一个文件中时，可以对每个 print 语句重复使用>>destination 子句，作为这种方法的一个替代方法，可以临时更改 sys.stdout 的值。下面的示例显示了一个通用重定向函数，开发者可以使用该函数对 sys.stdout 的值进行临时更改：

```
def redirect(func, *args, **kwds):
    """redirect(func,...) -> (output string result, func's return value)
    func must be a callable and may emit results to standard output.
    redirect captures those results as a string and returns a pair, with
    the results string as the first item and func's return value as the
    second one.
    """
    import sys, cStringIO
    save_out = sys.stdout
    sys.stdout = cStringIO.StringIO()
    try:
        retval = func(*args, **kwds)
        return sys.stdout.getvalue(), retval
    finally:
        sys.stdout.close()
        sys.stdout = save_out
```

要想将某些文本值输出到一个不遵循当前的 sys.stdout 值的文件对象 f，可以避免上面这样的复杂操作：对于这样的简单目的，只调用 f.write 通常就是最好的方法，并且 print>>f,...有时候也是一个非常便利的可选方案。

标准输出

sys 模块提供了 stdin 属性，这个属性是一个可读的文件对象。在开发者需要用户输入一行文本时，可以调用内置函数 raw_input（参见第 8.2 节中介绍的 raw_input 函数），并有选择地提供一个字符串参数，这样可以将这个字符串用作提示符。

在需要的输入不是一个字符串时（例如，在需要输入一个数字时），可以使用内置函数 input。但是，input 不适合用于大多数程序。更适合的方法是，使用 raw_input 从用户获得一个字符串，然后使用其他内置函数从字符串获得数字，比如，使用 int 或 float。只要开发者对用户完全信任，还可以使用 eval（为了更好地控制错误诊断，通常在前面使用 compile）让用户输入任何表达式。一个不怀好意的用户可以很容易地利用 eval 破坏安全性并引起损坏；对于这一点，并没有完全有效的防护方法，除非避免对来自不能完全信任的源的任何输入信息使用 eval（和 exec 语句）。不过，下面的函数使用了一些高级自测功能，可能会对这种情况有所帮助：

```
def moderately_secure_input(prompt):
  s = raw_input(prompt)
  c = compile(s, '<your input>', 'eval')
  if c.co_names: raise SyntaxError, 'names %r not allowed'%c.co_names
  return eval(c)
```

这个函数可能会引发一个 SyntaxError 异常（在这种情况下，如果需要，可以捕获 try/except 异常），该函数也不会让用户使用任何名称（也就是说，不是内置对象，也不是其他函数或变量），然而却可以接受多种表达式，并且适度安全，可以防止被滥用。

getpass 模块

有时候，开发者可能想要以下面这种方法输入一行文本：即使某人看着屏幕，也不知道用户键入的内容。这通常会出现在用户被要求输入密码的时候。getpass 模块提供了如表 10-19 所示的函数。

表 10-19

getpass	`getpass(prompt='Password: ')` 与 raw_input 一样，区别在于在用户输入时，输入的文本不会显示在屏幕上。getpass 的默认 prompt 不同于 raw_input。
getuser	`getuser()` 返回当前用户的用户名。getuser 将尝试按顺序从环境变量 LOGNAME、USER、LNAME 和 USERNAME 中的某一个变量的值中获取用户名。如果这些环境变量都不在 os.environ 中，getuser 将向操作系统询问这个信息。

10.10 富文本 I/O

到现在为止，本书介绍的工具只支持所有平台可以提供的文本 I/O 功能的最小子集。大多数平台还提供了富文本 I/O 功能，比如响应单次按键（而不是整行文本），还有在终端上的任何位置显示文本（而不是连续显示）。

Python 扩展和核心 Python 模块可以用来访问平台专用功能。不幸地是，各种平台是以不同的方式表示其功能。要想开发具有富文本 I/O 功能的跨平台 Python 程序，开发者需要统一包装不同的模块，根据情况导入平台专用模块（通常使用第 6.1 节中介绍的习惯用法）。

readline 模块

readline 模块可以用来包装 Readline 库。GUI Readline 库可以让用户在交互式输入时编辑文本行，并回放前面的输入行以进行编辑和重新输入。许多类 UNIX 平台上都安装了 Readline，可以在 http://cnswww.cns.cwru.edu/~chet/readline/rltop.html 上找到 Readline 库。移植的 Windows 版本（参见 http://starship.python.net/crew/kernr/）也是可用的，但是没有得到广泛使用。Chris Gonnerman 的模块是 Windows 上的 Alternative Readline 工具，这个模块实现了 Python 标准 readline 模块的子集（使用一个小的专用.pyd 文件，而不是 Readline 库），开发者可以在 http://newcenturycomputers.net/projects/ readline.html 找到这个模块。在 Windows 上使用 Readline 的一种方法就是安装 Gary Bishop 的 readline 版本（http://sourceforge.net/projects/uncpythontools）；但是，这个版本需要用到两个其他 Python 附加软件包（ctypes 和 PyWin32），因此不是非常轻量级地安装。

在 readline 可用时，Python 将使用 readline 实现所有面向行的输入，比如 raw_input。交互式 Python 解释器总是尝试加载 readline 以使能行编辑，并回放交互式会话。某些 readline 函数可以控制高级功能，特别是历史记录（history）功能，可以用来回放前面的会话中输入的行，还有自动完成（completion）功能，可以根据上下文对输入的单词进行自动完成（参见 http://cnswww.cns.cwru.edu/~chet/readline/rltop.html#Documentation 上的 GNU Readline 文档，其中包含了对配置命令的详细介绍）。Alternative Readline 工具也支持历史记录功能，但是其支持的与自动完成有关的函数是模拟的：在 Alternative Readline 工具中，这些函数不执行任何操作，其存在只是为了与 GNU Readline 功能兼容。

表 10-20

add_history	`add_history(s)` 将字符串 s 添加为历史记录缓冲区末尾的行。
clear_history	`clear_history(s)` 清除历史记录缓冲区。

续表

get_completer	get_completer() 返回当前的自动完成函数（set_completer 上次设置的函数），如果没有设置自动完成功能，则返回 None。
get_history_length	get_history_length() 返回被保存到历史记录文件中的历史记录的行数。在返回的值小于 0 时，历史记录中的行将被保存。

parse_and_bind	parse_and_bind(*readline_cmd*)
	为 Readline 设置一个配置命令。要想让用户点击 Tab 键来请求自动完成，可以调用 parse_and_bind('tab: complete')。参见 Readline 文档了解字符串 readline_cmd 的其他一些有用的值。
	rlcompleter 模块中有一个非常有用的自动完成函数。也就是，在交互式解释器会话中（或者，实际上是在解释器在每个交互式会话的开始运行的启动文件中，参见第 3.1 节）输入：
	`import readline, rlcompleter`
	`readline.parse_and_bind('tab: complete')`
	对于这个交互式会话的其余部分，开发者可以在进行行编辑时点击 Tab 键以获得完整的全局名称和对象属性。

read_history_file	read_history_file(*filename*=' /.history') 从名称或路径为 filename 的文本文件加载历史记录行。
read_init_file	read_init_file(*filename*=None) 让 Readline 加载一个文本文件，其中每行都是一个配置命令。在 filename 为 None 时，Readline 将加载与上次相同的文件。
set_completer	set_completer(*f*=None) 设置自动完成功能。在 f 为 None 时，Readline 将禁用自动完成。否则，在用户键入一个部分单词 start，然后按 Tab 键时，Readline 将调用 f(start, i)，且 i 被初始化为 0。f 将返回以 start 开始的第 i 个可能的单词，在没有这么多个单词时，将返回 None。Readline 将循环调用 f，i 将被设置为 0、1、2 等，直到 f 返回 None。
set_history_length	set_history_length(*x*) 设置被保存到历史记录文件中的历史记录行数。在 x 小于 0 时，该历史记录中的所有行都被保存。
write_history_file	write_history_file(*filename*=' /.history') 将历史记录中的所有行保存到名称或路径为 filename 的文本文件中。

控制台 I/O

现在的“终端”通常都是图形屏幕上的文本窗口，但是，开发者还是有可能会用到真正的终端或者个人计算机在字符模式下的控制台（主屏幕）。当前使用的所有类型的终

端都提供了高级文本 I/O 功能，但是开发者必须使用与平台相关的方式来访问这个功能。curses 软件包只能在类 UNIX 平台上使用（总是有谣言说 Windows 移植了该软件包，但是从来没有发现过一个可以使用的版本）。而 msvcrt、WConio 和 Console 模块只能在 Windows 平台上使用。

curses 软件包

出于某些并不十分明确的历史原因，传统的 UNIX 高级终端 I/O 实现方法被称为 curses（“Curses（诅咒）”很好地描述了程序员在面对这个丰富而复杂的方法时的典型态度）。Python 软件包 curses 提供了适当的简单使用方法，但是、如果需要，这个软件包也可以进行非常详细的控制。本书只介绍了 curses 的一个很小的子集，只是足以让开发者使用富文本 I/O 功能编写程序而已（参见 Eric Raymond 的教程“Curses Programming with Python”以了解更多信息，读者可以在 http://py-howto.sourceforge.net/curses/curses.html 上找到该教程）。本书在这一节中提到屏幕时，都表示终端的屏幕（例如，终端仿真器程序的文本窗口）。

最简单和最有效的方法就是通过 curses.wrapper 模块来使用 curses，这个模块只提供了一个函数。

表 10-21

wrapper	`wrapper(func, *args)` 执行 curses 的初始化，调用 func(stdscr, *args)，然后执行 curses 的终止化（将终端设置回普通模式），并最终返回该 func 的结果。wrapper 传递给 func 的第一个参数是 stdscr，这是一个类型为 curses.Window 的对象，表示整个终端屏幕。wrapper 可以将终端设置回普通模式，不管 func 是正常终止还是传播一个异常。 func 必须是一个函数，可以执行需要执行 curses 功能的程序中的所有任务。换句话讲，func 通常包含（或者更常用的说法就是，函数直接或者间接包含）程序的所有功能，但是除非交互式初始化和/或终止化任务之外。 curses 将字符的文本和背景色模型化为字符属性。终端上可用的颜色是从 0 到 curses.COLORS 编号的。color_content 函数可以将一个颜色编号 n 作为其参数，并返回一个由 0～1000 的整数组成的元组（r, g, b），参数 n 给出了三原色中每个颜色的量。color_pair 函数将使用一个颜色编号 n 作为其参数，并返回一个属性代码，为了按照这种颜色显示文本，开发者可以将该属性代码传递到 curses.Window 对象的各种方法中。 curses 可以创建类型为 curses.Window 的多个实例，每个实例对应于屏幕上的一个长方形。开发者还可以创建外来变体，比如 Panel 实例，该实例是 Window 的多态，但是不依赖于固定屏幕长方形。开发者不需要在简单 curses 程序中使用这样的高级功能：只需要使用 curses.wrapper 提供的 Window 对象 stdscr 即可。调用 w.refresh( ) 可以确保对任意 Window 实例 w（包括 stdscr）所作的更改都显示在屏幕上。curses 还可以缓冲所作的更改，直到开发者调用 refresh。除了提供许多其他方法之外，Window 的实例 w 还提供了以下这些常用方法。

续表

addstr	`w.addstr([y, x, ]s[, attr])` 使用属性 attr 将字符放到字符串 s 在窗口 w 的给定坐标（x, y）处，并改写以前的所有内容。所有 curses 函数和方法可以按相反的顺序接受坐标参数，将 y（行号）放在 x（列号）之前。如果不提供 y 和 x，则 addstr 将使用 w 的当前光标的坐标。如果不提供 attr，则 addstr 将使用 w 的当前默认属性。在任何情况下，addstr 在完成了添加字符串之后，将把 w 的当前光标坐标设置为其添加的字符串的末尾。
clrtobot, clrtoeol	`w.clrtobot() w.clrtoeol()` clrtoeol 将从 w 的当前光标坐标到行末尾写入空格。另外，clrtobot 也将用空格填充屏幕上当前位置下面的所有行。
delch	`w.delch([y, x])` 在坐标（x, y）处从 w 删除一个字符。如果不提供 y 和 x 参数，delch 将使用 w 的当前光标的坐标。在任何情况下，delch 不会更改 w 的当前光标的坐标。第 y 行中当前光标后面的所有字符（如果有的话）将会左移一位。
deleteln	`w.deleteln()` 从 w 中删除 w 的当前光标的坐标所在的整个行，并将屏幕中该光标下面的所有行向上滚动一行。
erase	`w.erase()` 向整个终端屏幕写入空格。
getch	`w.getch()` 返回一个对应于用户按的键的整数 c。0～255 的值表示一个普通字符，而大于 255 的值表示一个特殊键。curses 为特殊键提供了名称，因此开发者可以测试 c 与可读常数是否相等，比如 curses.KEY_HOME（Home 特殊键）、curses.KEY_LEFT（左箭头特殊键）等（Python 的免费文档中列出了所有的 curses 特殊键的名称（大约有 100 个），参见文档 *Python Library Reference* 中的“curses”节中的“Constants”子节）。如果通过调用 w.nodelay(True)函数将窗口 w 设置为无延时模式，那么，如果没有发生按键操作，则 w.getch 将引发一个异常。在默认情况下，w.getch 将会等待，直到用户按下一个键。
getyx	`w.getyx()` 以元组（y, x）的形式返回 w 的当前光标的坐标。
insstr	`w.insstr([y, x, ]s[, attr])` 使用属性 attr 将字符插入到字符串 s 在窗口 w 中的坐标（x, y）处，将这一行的剩余字符串向右移动。被移动超出行尾的任何字符都将丢失。如果不提供 y 和 x 参数的值，insstr 将使用 w 的当前光标的坐标。如果不提供 attr 参数的值，insstr 将使用 w 的当前默认属性。在任何情况下，在完成了插入字符串之后，insstr 将把 w 的当前光标的坐标设置为其插入的字符串的第一个字符所在的位置。
move	`w.move(y, x)` 将 w 的光标移动到给定坐标（x, y）。
nodelay	`w.nodelay(flag)` 在 flag 为 True 时，将 w 设置为无延时模式；在 flag 为 False 时，将 w 重置回普通模式。无延时模式会影响 w.getch 方法。

续表

refresh	`w.refresh()` 根据程序对窗口 w 所作的所有更改，更新屏幕上的窗口 w。 curses.textpad 模块提供了 Textpad 类，该类可以支持高级输入和文本编辑功能。

Textpad	`class Textpad(window)`
	创建并返回 Textpad 类的一个实例 t，Textpad 类包装了 curses 窗口实例 window。实例 t 有一个经常使用的方法：
	`t.edit()`
	让用户对 t 包装的窗口实例的内容执行交互式编辑。编辑会话支持简单的类 Emacs 键绑定：普通字符将改写窗口的以前内容、方向键可以移动光标，Ctrl+H 组合键可以删除光标左边的字符。在用户键入 Ctrl+G 组合键时，编辑会话将会终止，并且 edit 将把该窗口的内容返回为单个字符串，并使用换行符作为行分隔符。

msvcrt 模块

msvcrt 模块只能在 Windows 平台上使用，该模块提供的函数可以让 Python 程序访问 Microsoft Visual C++的运行时库 msvcrt.dll 提供的几个私有附加函数。某些 msvcrt 函数可以按字符读取用户输入的字符，而不是一次读取一整行。

表 10-22

getch, getche	`getch() getche()`
	读取并返回从键盘输入的一个字符，如果没有字符可供读取，则等待输入。getche 还可以将输入的字符显示在屏幕上（如果屏幕是可打印的），而 getch 则不能。在用户按下一个特殊键（方向键、功能键等）时，将被看作是两个字符：第一个字符是 chr(0)或 chr(224)，第二个字符与第一个字符一起，定义了用户键入的这个特殊键。要想知道 getch 为每个键返回的值，可以在 Windows 计算机上运行下面的这个小脚本：
	`import msvcrt`
	`print "press z to exit, or any other key to see the key's code:"`
	`while 1:`
	`  c = msvcrt.getch()`
	`  if c == 'z': break`
	`  print "%d (%r)" % (c, c)`

kbhit	`kbhit()` 在可以读取一个字符时，返回 True（如果调用了 getch，将立即返回）；否则，返回 False（如果调用了 getch，则等待键入）。
ungetch	`ungetch(c)` “不提取”字符 c；如果接下来调用 getch 或 getche，则返回 c。如果连续两次调用 ungetch，而不在其中插入调用 getch 或 getche 的操作，则会出现错误。

Wconio 和 Console 模块

这两个 Windows 专用扩展模块提供了单个字符键盘输入（与 msvcrt 类似）和将字符打

印在屏幕上的指定位置的功能。Chris Gonnerman 的 Windows 控制台 I/O 模块是很小、很简单，也很容易使用的；开发者可以从 http://newcenturycomputers.net/projects/wconio.html 免费下载这个模块，Fredrik Lundh 的控制台模块是非常完整，功能也非常丰富的；开发者可以从 http://www.effbot.org/efflib/console/ 免费下载这个模块。

10.11 交互式命令会话

cmd 模块提供了一种简单的方法，可以处理命令的交互式会话。每个命令都是一行文本。每个命令的第一个单词都是一个用来定义请求的动作的动词。命令行的其余部分将被传递为实现该动词定义的动作的方法的一个参数。

cmd 模块提供了 Cmd 类，并将其用作一个基类，开发者可以定义自己的 cmd.Cmd 子类。开发者定义的子类可以提供一些其名称以 do_和 help_开始的方法，还可以有选择地覆盖某些 Cmd 方法。在用户输入一个像 verb and the rest 这样的命令行时，只要开发者的子类定义了一个名为 do_verb 的方法，Cmd.onecmd 将调用：

```
self.do_verb('and the rest')
```

与此类似，只要开发者的子类定义了一个名为 help_verb 的方法，Cmd.do_help 将在命令行以‘help verb’或‘?verb’开始时调用该方法。在默认情况下，如果用户尝试使用某个动作或请求有关该动作的帮助，而用户的子类却没有定义所需的方法，则 Cmd 将显示适当的错误消息。

初始化 Cmd 实例

如果 cmd.Cmd 定义了自己的__init__特殊方法，则 cmd.Cmd 的子类必须调用基类的__init__，__init__的说明如表 10-23 所示。

表 10-23

__init__	Cmd.__init__(self, *completekey*='Tab', *stdin*=sys.stdin, *stdout*=sys.stdout) 使用指定或默认的 completekey 值(readline 模块中用于命令自动完成的键的名称；传递 None 则会禁用命令自动完成)、stdin(用于得到输入信息的文件对象)和 stdout(用于显示输出信息的文件对象）初始化实例 self。 如果开发者的子类没有定义__init__方法，则该子类将从基类 cmd.Cmd 继承这个方法。在这种情况下，可以按照上面介绍的方法，使用可选参数 completekey、stdin 和 stdout 来实例化开发者的子类，并调用该子类。

Cmd 实例的方法

类 Cmd 的子类的实例 c 提供了以下方法（其中的许多方法将被其子类覆盖）。

表 10-24

cmdloop	c.cmdloop(*intro*=None)
	执行一个面向命令行的交互式会话。cmdloop 从调用 c.preloop( )开始，然后显示字符串 intro（如果 intro 为 None，则显示 c.intro)。然后，c.cmdloop 进入一个循环。在循环的每次迭代中，cmdloop 将使用 s=raw_input(c.prompt)读取命令行 s。在标准输入到达行尾时，cmdloop 将设置 s='EOF'。如果 s 不是‘EOF’，cmdloop 将使用 s=c.precmd(s)对字符串 s 进行预处理，然后调用 flag=c.onecmd(s)。在 onecmd 返回一个为 True 的值时，表示将尝试请求终止命令循环。不管 flag 的值是什么，cmdloop 将调用 flag=c.postcmd(flag, s)来检查命令循环是否应该终止。如果现在 flag 的值仍为 True，则循环将会终止；否则，将再次重复循环。在循环终止时，cmdloop 将调用 c.postloop( )，然后终止。开发者可以非常容易地通过查看下面的等效 Python 代码来理解这里介绍的 cmdloop 方法：
	`def cmdloop(self, intro=None):`
	`  self.preloop()`
	`  if intro is None: intro = self.intro`
	`  print intro`
	`  finis_flag = False`
	`  while not finis_flag:`
	`  try: s = raw_input(self.prompt)`
	`  except EOFError: s = 'EOF'`
	`  else: s = self.precmd(s)`
	`  finis_flag = self.onecmd(s)`
	`  finis_flag = self.postcmd(finis_flag, s)`
	`  self.postloop()`
	cmdloop 是被称为模板方法（Template Method）的经典设计模式（Design Pattern）的一个很好的示例。这样的方法本身并不执行什么具体工作；而是构造和组织对其他方法的调用。子类可以覆盖某些或所有其他方法以定义由此建立的总体结构中的类行为的细节。在从 Cmd 继承时，几乎不会覆盖 cmdloop 方法，因为 cmdloop 的结构从 Cmd 继承为子类时得到的主要内容。

default	c.default(*s*) 在命令行 s 的第一个单词 verb 没有对应的方法 c.do_verb 时，c.onecom 将调用 c.defaults (s)。子类通常会覆盖这个默认方法。基类方法 Cmd.default 将打印一个错误消息。
do_help	c.do_help(*verb*) 在命令行 s 以‘help verb’或‘?verb’开始时，c.onecmd 将调用 c.do_help(verb)。子类极少会覆盖 do_help 方法。如果子类提供了 help_verb 方法，Cmd.do_help 方法将调用这个 help_verb 方法；否则，如果子类为 do_verb 方法提供了一个非空 docstring，Cmd.do_help 方法将显示 do_verb 方法的 docstring。如果子类没有提供其中的任何一个帮助方法，Cmd.do_help 将输出一个消息，以通知用户这个 verb 没有可用的帮助方法。
emptyline	c.emptyline() 在命令行 s 为空或者为空格时，c.onecmd 将调用 c.emptyline( )。除非子类覆盖了这个方法，否则基类方法 Cmd.emptyline 将重新执行看到的最后一个非空命令行，并将其保存在 c 的属性 c.lastcmd 中。

续表

onecmd	`c.onecmd(s)` c.cmdloop 可以为用户输入的每个命令行 s 调用 c.onecmd(s)。如果开发者独立获得了一个命令行 s 以将其作为一个命令来处理，还可以直接调用 onecmd 方法。通常，子类不覆盖 onecmd 方法。Cmd.onecmd 将设置 c.lastcmd=s。然后，在 s 是以单词 verb 开始，并且其子类提供了 do_verb 方法时，onecmd 将调用 do_verb；否则，正如前面介绍的，onecmd 方法将调用空行或者默认方法。在任何情况下，Cmd.onecmd 将返回其调用的任何其他方法的结果，postcmd 将把这个结果解释为一个终端请求标记。
postcmd	`c.postcmd(flag, s)` 在 c.onecmd(s)返回了 flag 值之后，c.cmdloop 将为每个命令行 s 调用 c.postcmd (flag, s)方法。如果 flag 为 True，则执行的命令就是引发一个试探性的请求以终止命令循环。如果 postcmd 返回一个为 True 的值，则 cmdloop 的循环将会终止。除非开发者定义的子类覆盖了基类方法 Cmd.postcmd，否则将调用该方法，并返回 flag 本身作为该方法的结果。
postloop	`c.postloop()` 在 cmdloop 的循环终止时，c.cmdloop 将调用 c.postloop( )。除非开发者自己的子类覆盖了基类方法 Cmd.postloop，否则该方法将不执行任何操作。
precmd	`c.precmd(s)` c.cmdloop 可以调用 s=c.precmd(s)以对每个命令行 s 进行预处理。循环的当前值将使得所有的进一步处理都基于 precmd 返回的字符串。除非开发者定义的子类覆盖了基类方法 Cmd.precmd，否则该方法将被调用，并返回 s 本身作为该方法的结果。
preloop	`c.preloop()` c.cmdloop 可以在循环开始之前调用 c.preloop( )。除非开发者定义的子类覆盖了基类方法 Cmd.preloop，否则该方法将不执行任何操作。

Cmd 实例的属性

Cmd 类的子类的实例 c 提供了以下属性：

identchars

其中的所有字符都是可以作为一个动作的一部分的字符串；在默认情况下，c.identchars 可以包含字母、数字和一个下划线（_）。

intro

在不带任何参数调用 cmdloop 时，该方法输出的第一个消息。

lastcmd

onecmd 看到的最后一个非空命令行。

prompt

cmdloop 用来提示用户进行交互式输入的字符串。开发者几乎总是会显式绑定 c.prompt，

或者将 prompt 改写为子类的类属性，因为 Cmd.prompt 的默认值只是‘(Cmd)’。

use_rawinput

在这个属性为 False 时（默认值为 True），cmdloop 将通过调用 sys.stdout 和 sys.stdin 的方法进行提示和输入，而不是通过 raw_input 方法。

本节没有介绍的其他一些 Cmd 实例还可以帮助开发者对帮助消息的许多格式化细节提供更加细致的控制。

Cmd 示例

下面的示例显示了如何使用 cmd.Cmd 提供动词 print（输出一行的其余部分）和 stop（结束循环）：

```
import cmd
class X(cmd.Cmd):
    def do_print(self, rest):
        print rest
    def help_print(self):
        print "print (any string): outputs (any string)"
    def do_stop(self, rest):
        return True
    def help_stop(self):
        print "stop: terminates the command loop"
if __name__ == '__main__':
    X().cmdloop()
```

使用这个示例的一个 Python 会话的流程如下：

```
C:\>\python22\python \examples\chapter10\CmdEx.py
(Cmd) help
Documented commands (type help <topic>):
= == == == == == == == == == == == == == == == == == =
print          stop
Undocumented commands:
= == == == == == == == == == =
help
(Cmd) help print
print (any string): outputs (any string)
(Cmd) print hi there
hi there
(Cmd) stop
```

10.12 国际化

大多数程序都会以文本的形式向用户显示一些信息。这些文本应该是读者可以理解和可以识别的。例如，在有些国家和有些文化习俗中，日期“3 月 7 日”也可以简明地表示为“3/7”。而在其他一些地方，“3/7”却表示“7 月 3 日”，而用来表示“3 月 7 日”的字符串是“7/3”。在 Python 中，这样的文化习俗都是通过标准模块 locale（本地语言

环境）来处理的。

与此类似，一个简单的问候可以按自然语言的方式，使用字符串“欢迎”来表示，而在其他语言中，使用的字符串是“Welcome”。在 Python 中，这样的翻译操作是使用标准模块 gettext 来处理的。

上述两个问题通常被称为国际化（internationalization），这个单词经常被缩写为 i18n，因为在完整的单词拼写中，i 到 n 之间有 18 个字符。但是这个单词也并不十分恰当，同样的问题也会出现在同一个国家中使用不同语言或具有不同文化习俗的用户上。

locale 模块

Python 对文化习俗的支持是模仿 C 语言的，但是进行了一些简化。在这个体系结构中，程序是在一个被称为 locale（本地语言环境）的文化习俗环境中运行的。本地语言环境设置渗透到程序的方方面面，并且通常是在程序启动时设置的。本地语言环境不是一个专用线程，并且 locale 模块也不是线程安全的。在多线程程序中，要在启动第二个线程时设置程序的本地语言环境。

如果程序没有调用 locale.setlocale，则该程序将在一种被称为 C 本地语言环境的自然本地语言环境下运行。C 本地语言环境得名于 C 语言中这种体系结构的原型，类似于，但是不等同于美国英语中的单词 locale。另一个选择是，程序可以找到并接受用户的默认本地语言环境设置。在这种情况下，locale 模块将与操作系统进行交互（通过环境变量或者以其他与系统相关的方法），以建立用户首选的本地语言环境。最后要讲的是，程序可以设置一个特定的本地语言环境，这是基于用户的交互式操作或者通过持久化的配置设置（比如，程序的初始化文件）进行推测，以决定设置哪种本地语言环境。

本地语言环境设置通常可以应用于一种文化习俗的所有相关类别。这种广谱化的设置是由 locale 模块的常量属性 LC_ALL 指定的。但是，locale 模块处理的文化习俗将被分组为类别，并且，在某些情况下，程序可以选择混合和匹配这些类别以建立一种合成的复合本地语言环境。这些类别是通过 locale 模块的以下常量属性来标识的：

LC_COLLATE

字符串排序；影响 locale 模块中的 strcoll 和 strxfrm 函数；

LC_CTYPE

字符类型；影响必须处理小写字母和大写字母的 string 模块（和字符串方法）的特性；

LC_MESSAGES

消息；可能会影响操作系统显示的消息——例如，os.strerror 函数和 gettext 模块；

LC_MONETARY

货币值的格式；影响 locale.localeconv 函数；

LC_NUMERIC

数字的格式；影响 locale 模块中的 atoi、atof、format、localeconv 和 str 函数；

LC_TIME

时间和日期的格式；影响 time.strftime 函数。

某些类别（由常量 LC_CTYPE、LC_TIME 和 LC_MESSAGES 表示）的设置会影响其他模块（可以指定为 string、time、os 和 gettext）中的行为。其他类别（由常量 LC_COLLATE、LC_MONETARY 和 LC_NUMERIC）的设置只影响 locale 模块本身的某些函数。

locale 模块提供了一些函数以查询、更改和操作本地语言环境，以及用来实现本地语言环境类别（LC_COLLATE、LC_MONETARY 和 LC_NUMERIC）的文化习俗的函数。

表 10-25

atof	atof(*s*) 使用当前的 LC_NUMERIC 设置将字符串 s 转换为一个浮点型数字。
atoi	atoi(*s*) 使用当前的 LC_NUMERIC 设置将字符串 s 转换为一个整数。

format	format(*fmt, num, grouping*=False)
	根据格式化字符串 fmt 和 LC_NUMERIC 设置，返回通过格式化数字 num 而获得的字符串。除了文化习俗的问题，其结果类似于 fmt%num。如果 grouping 为 True，format 还将根据 LC_NUMERIC 设置对结果字符串中的数字进行分组。例如：
	`>>> locale.setlocale(locale.LC_NUMERIC, 'en')`
	`'English_United States.1252'`
	`>>> locale.format('%s', 1000*1000)`
	`'1000000'`
	`>>> locale.format('%s', 1000*1000, True)`
	`'1,000,000'`
	在数字本地语言环境为美国英语，并且参数 grouping 为 True 时，format 支持将数字按 3 位分组，并使用逗号分隔的习俗。

getdefaultlocale	getdefaultlocale(*envvars*=['LANGUAGE','LC_ALL','LC_TYPE','LANG']) 按 envvars 中指定的名称的顺序检查环境变量。第一个环境变量确定了默认的本地语言环境。getdefaultlocale 将返回一个遵循 RFC 1766（除‘C’本地语言环境之外）的字符串对（lang, encoding），比如，['en_US', 'ISO8859-1']。如果 getdefaultlocale 不能找到这个项目应该具有什么值，则这个字符串对中的每个项目都为 None。
getlocale	getlocale(*category*=LC_CTYPE) 返回符合给定 category 的当前设置的字符串对（lang, encoding）。这个类别不能是 LC_ALL。

续表

localeconv	localeconv() 返回一个字典 d，其中包含由当前本地语言环境中的 LC_NUMERIC 和 LC_MONETARY 类别指定的文化习俗。LC_NUMERIC 最好通过 locale 模块的其他函数间接使用，而 LC_MONETARY 的详细信息只能通过 d 来访问。本地和国际上使用的货币格式是不同的。例如，美元符号‘$’仅用于本地使用，而在国际上使用时，‘$’会造成误解，因为相同的符号还用于其他被称为“dollar”的货币（比如，加拿大、澳大利亚等）。因此，在国际上使用时，明确表示美元符号的字符串是‘USD’。字典 d 中用于货币格式化的键是以下这些字符串： 'currency_symbol' 本地使用的货币符号； 'frac_digits' 本地使用的货币符号的小数点位数； 'int_curr_symbol' 国际上使用的货币符号； 'int_frac_digits' 国际上使用的货币符号的小数点位数； 'mon_decimal_point' 用作货币值的“小数点”的字符串； 'mon_grouping' 货币值的数字分组位数； 'mon_thousands_sep' 用作货币值的数字分组分隔符的字符串； 'negative_sign' 'positive_sign' 用作负的（正的）货币值的表示符号的字符串； 'n_cs_precedes' 'p_cs_precedes' 如果将货币符号放在负的（正的）货币值之前，则该字符串的值为 True； 'n_sep_by_space' 'p_sep_by_space' 如果在正负符号和负的（正的）货币值之间有一个空格，则该字符串的值为 True； 'n_sign_posn' 'p_sign_posn' 用于格式化负的（正的）货币值的数字代码： 0 货币值和货币符号都被放在圆括号中； 1 正负符号被放在货币值和货币符号之前； 2 正负符号被放在货币值和货币符号之后； 3 正负符号被放在货币值之前； 4 正负符号被放在货币值之后。 CHAR_MAX 当前的本地语言环境不为这个格式化指定任何习惯用法。 d['mon_grouping']是在格式化一个货币值时，要进行分组的一个数字列表。在 d['mon_grouping'][-1]为 0 时，不对指定的数字进行进一步地分组。在 d['mon_grouping'][-1]为 locale.CHAR_MAX 时，进行无限循环分组，d['mon_grouping'][-2]也是无限重复分组。locale.CHAR_MAX 是一个常数，用作字典 d 中货币本地语言环境没有指定任何习惯用法的所有条目的货币格式化的值。

续表

normalize	normalize(*localename*) 返回一个适合于作为 setlocale 的参数的字符串，这个字符串将被规范化为等于 localename。如果 normalize 不能规范化 localename 字符串，则 normalize 将返回未经更改的 localename。
resetlocale	resetlocale(*category*=LC_ALL) 将类别为 category 的本地语言环境设置为 getdefaultlocale 给定的默认值。
setlocale	setlocale(*category*, *locale*=None) 如果 locale 不为 None，则将类别为 category 的本地语言环境设置为给定的 locale，并返回其设置（在 locale 为 None 时，返回现有的设置；否则，返回新设置）。locale 可以是一个字符串或者一个字符串对（lang, encoding）。lang 字符串通常是一个基于 ISO 639 中的 2 字符代码（'en' 表示英语，'nl' 表示荷兰语等）的语言代码。在 locale 为空白字符串 '' 时，setlocale 将设置用户的默认本地语言环境。
str	str(*num*) 与 locale.format('%f', *num*)类似。
strcoll	strcoll(*str1*, *str2*) 与 cmp(str1, str2)类似，但是遵循 LC_COLLATE 的设置。

strxfrm	strxfrm(*s*)
	返回一个字符串 sx，该字符串是使用字符串的内置比较（例如，通过 cmp）函数对字符串 s 进行转换而来，等同于对原始字符串调用 locale.strcoll。strxfrm 可以使用装饰-排序-去除装饰（DSU）习惯用法进行包含本地语言环境一致性字符串比较的排序。但是，如果开发者需要的只是按本地语言环境一致性的方式对字符串列表进行排序，则 strcoll 的简易性会使得排序的速度更快。下面的示例显示了执行这样的排序的两种方法；在这种情况下，比较简单的方法通常要比 DSU 更快一些，对于包含 1000 个单词的列表，大约要快 10%：
	`import locale`
	`# 更简单，通常也更快的方法`
	`def locale_sort_simple(list_of_strings):`
	`    list_of_strings.sort(locale.strcoll)`
	`# 比较复杂，通常也更慢的方法`
	`def locale_sort_DSU(list_of_strings):`
	`    auxiliary_list = [(locale.strxfrm(s), s) for s in list_of_strings]`
	`    auxiliary_list.sort()`
	`    list_of_strings[:] = [s for junk, s in auxiliary_list]`
	在 Python 2.4 中，排序方法的 key= 参数同时提供了简单性和速度：
	`# 最简单，也最快的方法，但是只能在 Python 2.4 中使用：`
	`def locale_sort_2_4(list_of_strings):`
	`    list_of_strings.sort(key=locale.strxfrm)`

gettext 模块

国际化中的一个关键问题就是以不同的自然语言使用文本的能力，也就是一个被称为本地化（localization）的任务。Python 通过 gettext 模块提供了本地化功能，这个模块来源于 GNU gettext 工具。gettext 模块可以有选择地使用 GNU gettext 的体系结构和 API，但是该模块更简单，也更常用。开发者不需要安装或者学习 GNU gettext 就可以有效地使用 Python 的 gettext 模块了。

使用 gettext 进行本地化

gettext 并不处理自然语言之间的自动翻译。而是帮助开发者提取、组织和访问程序中使用的文本消息。将每个要翻译的字符串字面常量主题（也被称为消息）传递到一个以 _（下划线）命名的函数，而不是直接使用该主题。gettext 通常会在__builtin__模块中安装一个以 _ 命名的函数。要想确保开发者的程序在运行时使用或者不使用 gettext 模块，可以有条件地定义一个名为 _ 的不执行任何操作的函数，这个函数只是返回其未经改变的参数。然后，开发者就可以在通常使用一个必须要被翻译的字面常量‘message’的任何位置安全地使用_('message')了。下面的示例显示了如何运行一个有条件使用 gettext 的模块：

```
try: _
except NameError:
    def _(s): return s
def greet(): print _('Hello world')
```

如果在运行上面这个示例代码之前，其他一些模块已经安装了 gettext，则 greet 函数将输出一个正确的本地化祝贺词（greeting）。否则，greet 将输出未经改变的字符串‘Hello word’。

编辑源代码，使用函数 _ 装饰消息字面常量。然后，使用任何一种工具将消息提取到一个文本文件中（通常命名为 messages.pot），并将该文件发布到可以将该消息翻译为应用程序必须支持的各种自然语言的人。Python 提供了一个脚本 pygettext.py（在发布的 Python 源代码中的 Tools/i18n 目录下），这个脚本可以执行对 Python 源代码的消息提取。

每个翻译器都可以编辑 messages.pot 以生成一个包含已翻译的消息的文本文件，其扩展名为.po。将.po 文件编译为扩展名为.mo 的二进制文件，该文件适合于使用任何工具进行快速搜索。为了实现以上功能，Python 提供了脚本 Tools/i18n/msgfmt.py。最后，在适当的目录中以适当的名称安装每个.mo 文件。

从习惯上讲，哪些目录和名称适合于进行安装因平台和应用程序的不同而异。gettext 的默认目录是目录 sys.prefix 的子目录 share/locale/<lang>/LC_MESSAGES/，其中<lang>是语言的代码（两个字母）。每个文件都被命名为<name>.mo，其中<name>是应用程序或软件包的名称。

在开发者已经准备好，并安装了.mo 文件之后，通常会在应用程序启动的时候执行下面

这样的代码：

```
import os, gettext
os.environ.setdefault('LANG', 'en')        # 应用程序的默认语言
gettext.install('your_application_name')
```

这样可以确保调用像_('message')这样的函数将返回适当的已翻译的字符串。开发者可以选择不同的方法以访问程序中的 gettext 功能——例如，如果开发者还需要本地化 C 编码的扩展，或者要在运行期间在不同的语言之间来回切换。另一个非常重要的考虑就是，开发者是要本地化整个应用程序，还是只本地化单独发布的软件包。

主要的 gettext 函数

gettext 模块提供了许多函数；最常使用的函数如表 10-26 所示。

表 10-26

install	install(*domain*, *localedir*=None, *unicode*=False) 在 Python 的内置命名空间中安装一个名为 _ 的函数，并根据 getdefaultlocale 指定的语言代码<lang>执行目录 localedir 中的文件<lang>/LC_MESSAGES/<domain>.mo 中给出的翻译。在 localedir 为 None 时，install 将使用目录 os.path.join(sys.prefix, 'share', 'locale')。在 unicode 为 True 时，函数 _ 将接受并返回 Unicode 字符串，而不是普通字符串。

translation	translation(*domain*, *localedir*=None, *languages*=None)
	搜索一个类似于 install 函数的.mo 文件。在 languages 为 None 时，translation 将在环境中查找要使用的 lang，就像 install 一样。该函数将按顺序查看环境变量 LANGUAGE、LC_ALL、LC_MESSAGES、LANG；第一个非空环境变量将根据‘:’进行分割以给出一个语言名列表（例如，‘de:en’将被分割成['de', 'en']）。翻译操作将使用该列表中的第一个语言名称对找到的.mo 文件进行翻译。translation 函数将返回一个提供了 gettext（翻译普通字符串）、ugettext（翻译 Unicode 字符串）和 install（将 _ 名称下的 gettext 或 ugettext 方法安装到 Python 的内置命名空间中）方法的实例对象。
	translation 函数提供了比 install 函数更详细的控制，install 函数就像 translation(domain,localedir)一样。使用 translation 函数，开发者可以在按模块的基础上绑定名称 _ ，而不影响内置命名空间，例如，使用：
	`_ = translation(domain).ugettext`
	translation 还可以在几种语言之间全局切换，因为开发者可以传递一个显式 languages 参数，保留结果实例，并根据需要调用相应语言的 install 方法：
	`import gettext`
	`translators = {}`
	`def switch_to_language(lang, domain='my_app',`
	`use_unicode=True):`
	`  if not translators.has_key(lang):`
	`  translators[lang] = gettext.translation(domain,`
	`languages=[lang])`
	`  translators[lang].install(use_unicode)`

更多国际化资源

国际化是一个非常大的主题。要想了解一些有关国际化的基本介绍和有用的资源，可以参见 http://www.debian.org/doc/manuals/intro-i18n/ 和 http://www.i18ngurus.com/。关于国际化的一个最好的代码和信息软件包是 ICU（http://icu.sourceforge.net/），ICU 还包括 Unicode 协会提供的非常好的有关本地语言环境习俗的“公共本地语言环境数据库”（Common Locale Data Repository，CLDR），以及用来访问 CLDR 的代码。不幸地是，在编写本书的时候，ICU 只支持 Java、C 和 C++，而不（直接）支持 Python。开发者可以很容易地通过 Jython（参见第 26.1 节以了解更多有关如何通过 Jython 代码使用 Java 类的信息）使用 ICU 的 Java 版本；再多花一些功夫，开发者还可以使用像 SWIG 或 SIP（参见第 25 章）这样的工具包装 ICU 的 C/C++版本以从经典 Python 访问 ICU 功能。

第 11 章

持久化和数据库

Python 支持以各种方法实现数据的持久化。其中的一个方法被称为序列化（serialization），也就是将数据看作一个由 Python 对象组成的集合。这些对象可以被保存（也就是序列化，serialized）到一个字节流中，以后再从字节流加载和重建（也就是反序列化，deserialized）。从层次上来讲，对象持久化在序列化之上，并添加了像对象命名这样的功能。本章介绍了支持序列化和对象持久化的 Python 模块。

实现数据持久化的另一种方法就是将其保存在数据库中。一种简单类型的数据库就是一个文件格式，其中，使用键存取（keyed access）方式对数据的相关部分进行读取和更新。本章将介绍支持这种被称为 DBM 的文件格式的几种变体的 Python 标准库模块。

“关系数据库管理系统”（RDBMS），比如 MySQL 或 Oracle，都提供了一种更强大的方法以存储、搜索和提取持久化的数据。关系数据库依赖于各种“结构化查询语言”（SQL）创建和改变一个数据库的结构描述（schema），插入和更新数据库中的数据，并根据搜索规则查询数据库（本章不提供有关 SQL 的参考资料。要想了解相关信息，本书推荐读者阅读由 O'Reilly 出版，Kevin Kline 编著的 *SQL in a Nutshell* 一书）。不幸地是，尽管 SQL 标准是存在的，但是没有任何两个 RDBMS 实现完全使用相同的 SQL 语句。

Python 标准库没有附带 RDBMS 接口。但是，许多免费的第三方模块可以帮助 Python 程序访问一个特殊的 RDBMS。这样的模块大多数遵循 Python Database API 2.0 标准，也被称为 DBAPI。本章将介绍 DBAPI 标准，还将介绍实现了该标准的一些第三方模块。

11.1 序列化

Python 提供了许多用来处理 I/O 操作的模块，可以将整个 Python 对象序列化（保存）

到各种类型的字节流中，还可以从这样的字节流中反序列化（加载和重建）Python 对象。序列化也被称为排列（marshaling）。

marshal 模块

marshal 模块提供了保存和重新加载编译后的 Python 文件（.pyc 和.pyo）所需的特殊序列化任务。marshal 只能处理基本的内置数据类型：None、数字（整型、长整型、浮点型和复数）、字符串（普通字符串和 Unicode）、代码对象和内置容器（元组、列表和字典），其中的项目都是基本类型的实例。marshal 不能处理集合，也不能处理用户自定义的类型和类。marshal 要比其他序列化模块更快一些，并且是一个支持代码对象的模块。marshal 模块提供了如表 11-1 所示的函数。

表 11-1

dump, dumps	`dump(value,fileobj)`
	`dumps(value)`
	dumps 可以返回一个表示对象 value 的字符串。dump 将把相同的字符串写入到文件对象 fileobj 中，该文件对象必须以二进制可写模式打开，dump(v, f)就像 f.write(dumps(v))一样。fileobj 不能是任意类文件对象：而必须是一个确定的 file 类型实例。

load, loads	`load(fileobj)`
	`loads(str)`
	loads 可以创建并返回以前转出（dump）到字符串 str 的对象 v，这样，对于任意对象 v，如果 loads 支持该对象的类型，则 v = = loads(dumps(v))。如果 str 比 dumps(v)长，则 loads 将忽略多余的字节。而 load 将从文件对象 fileobj 读取正确的字节数，该文件对象必须以二进制可读模式打开，load 将创建并返回这些字节表示的对象 v。fileobj 不能是任意类文件对象：而必须是一个确定的 file 类型实例。
	load 和 dump 函数是互补的。换句话讲，如果 f 的内容是通过连续调用 dump(v, f)创建的，则连续调用 load(f)将反序列化以前通过 dump(v, f)序列化的相同值。

marshal 示例

假定开发者需要阅读几个文本文件，这些文件的名称被作为程序的参数，用来记录每个单词在文件中的出现位置。开发者需要为每个单词记录的是一个由（filename, line-number）数据对组成的列表。下面的示例使用 marshal 模块将（filename, line-number）数据对列表编码为字符串，并将其保存在一个类 DBM 文件（参见第 11.2 节）中。因为这些列表包含元组，每个元组包含一个字符串和一个数字，因此 marshal 模块可以序列化这些元组。

```
import fileinput, marshal, anydbm
wordPos = {}
for line in fileinput.input():
```

```
    pos = fileinput.filename(), fileinput.filelineno()
    for word in line.split():
        wordPos.setdefault(word,[]).append(pos)
dbmOut = anydbm.open('indexfilem', 'n')
for word in wordPos:
    dbmOut[word] = marshal.dumps(wordPos[word])
dbmOut.close()
```

开发者还需要使用 marshal 模块读取保存在类 DBM 文件 indexfilem 中的数据，如下面的示例所示：

```
import sys, marshal, anydbm, linecache
dbmIn = anydbm.open('indexfilem')
for word in sys.argv[1:]:
    if not dbmIn.has_key(word):
        sys.stderr.write('Word %r not found in index file\n' % word)
        continue
    places = marshal.loads(dbmIn[word])
    for fname, lineno in places:
        print "Word %r occurs in line %s of file %s:" % (word,lineno,fname)
        print linecache.getline(fname, lineno),
```

pickle 和 cPickle 模块

pickle 和 cPickle 模块提供了名为 Pickler 和 unpickler 的工厂函数，以生成可以包装类文件对象并提供序列化机制的对象。通过这些模块实现的序列化和反序列化也被称为封装（pickling）和拆封（unpickling）。这两个模块之间的区别在于，在 pickle 中，Pickler 和 Unpickler 是类，因此可以从这些类继承以创建自定义的序列化对象，根据需要覆盖其中的一些方法。另一方面，在 cPickle 中，Pickler 和 Unpickler 都是可以生成不能有子类的类型实例的工厂函数，而不是类。cPickle 的性能要好得多，但是不能实现继承。在本节的其余部分，主要将介绍 pickle 模块，但是该模块的所有内容也都可以应用到 cPickle 模块上。

请注意，从一个不被信任的数据源进行拆封是有安全风险的；攻击者可能会利用这一点执行任意代码。因此，请不要拆封不被信任的数据！

序列化与深入复制具有某些共同的问题，参见第 8.4 节中介绍的 deepcopy 函数。pickle 模块可以按照与 copy 模块非常相同的方式处理来这些问题。与深入复制一样，序列化隐含对由引用组成的有向图进行递归遍历操作。在多次遇到一个相同的对象时，pickle 将保存这个有向图的形状：只在第一次遇到该对象时才对其进行序列化，而出现的其他相同对象的序列化操作将直接引用已有的序列化副本。pickle 还可以正确地序列化包含循环引用的图。但是，这意味着，如果一个可变对象 O 被多次序列化为相同的 Pickler 实例 p，则在第一次将 O 序列化为 p 之后，任何对 O 的更改

都不会被保存。为了更清楚和更简单，要避免在对一个 Pickler 实例的实例化正在进行时改变正在被序列化的对象。

pickle 可以按 ASCII 格式或者两种压缩二进制格式进行序列化。为了向后兼容，默认值是 ASCII 格式，但是开发者通常也可以要求二进制格式 2，这种格式可以节省时间和存储空间。在重新加载对象时，pickle 将透明识别和使用任何格式。本书建议开发者总是指定二进制格式 2：可以大量节省内存大小和处理速度，并且，除了缺少与非常老的 Python 版本的兼容性之外，二进制格式基本上没有什么不好的地方。

pickle 可以按名称，而不是按值来序列化类和函数。因此，如果 pickle 序列化了某个模块中找到的一个类或函数，则 pickle 只需要从相同的模块中导入这个已序列化的类或函数的名称就可以反序列化该类或函数了。特别是，只有在类和函数是其模块的顶层名称（也就是，其模块的属性）时，pickle 才可以序列化和反序列化这些类和函数。例如，考察以下代码示例：

```
def adder(augend):
    def inner(addend, augend=augend): return addend+augend
    return inner
plus5 = adder(5)
```

这段代码将一个闭包绑定到名称 plus5 上（参见第 4.11 节中介绍的嵌套函数和嵌套范围），plus5 是一个嵌套函数 inner 加上一个适当的嵌套范围。因此，尝试封装 plus5 将会引发一个 pickle.PicklingError 异常：一个函数只有在其为顶层函数时才能被封装，而在这段代码中，inner 函数的闭包被绑定到名称 plus5 上，inner 函数不是一个顶层函数，而是 adder 函数中嵌套的函数。类似的问题也会出现在嵌套函数和嵌套类（即不是顶层的类）的所有封装中。

pickle 和 cPickle 的函数

pickle 和 cPickle 模块包含以下方法，如表 11-2 所示。

表 11-2

dump, dumps	dump(*value*,*fileobj*,*protocol*=None,*bin*=None) dumps(*value*,*protocol*=None,*bin*=None) dumps 可以返回一个表示对象 value 的字符串。dump 将把相同的字符串写入到类文件对象 fileobj 中，该类文件对象必须以可写模式打开。dump(v, f)就像 f.write(dumps(v))一样。不要传递 bin 参数，该参数的存在只是为了与老版本的 Python 兼容。protocol 参数可以为 0（出于兼容性的原因，默认值为 0，表示以 ASCII 字符输出，最慢也最消耗空间）、1（二进制输出，与老版本的 Python 兼容）或者 2（最快也最节省空间）。本书建议开发者总是传递参数值 2。除非 protocol 为 0，或者不提供该参数值，否则暗示以 ASCII 输出，dump 的 fileobj 参数必须以二进制写模式打开。

续表

load, loads	load(*fileobj*) loads(*str*) loads 可以创建并返回字符串 str 表示的对象 v，这样，对于一个支持的类型的任意对象 v，v==loads(dumps(v))。如果 str 比 dumps(v)长，则 loads 将忽略多余的字节。而 load 将从类文件对象 fileobj 读取正确的字节数，并创建和返回这些字节表示的对象 v。load 和 loads 都透明支持在任何二进制或 ASCII 模式下执行的封装。如果数据被封装为任意一种二进制格式，则为了进行 dump 和 load，必须以二进制方式打开该文件。load(f)就像 Unpickler(f).load( )一样。 load 和 dump 函数是互补的。换句话讲，如果 f 的内容是通过连续调用 dump(v, f)创建的，则连续调用 load(f)将反序列化以前通过 dump(v, f)序列化的相同值。
Pickler	Pickler(*fileobj protocol*=None,*bin*=None) 创建并返回一个对象 p，这样，在将 fileobj、protocol 和 bin 等参数传递到 Pickler 时，调用 p.dump 等同于调用 dump 函数。要想将多个对象序列化为一个文件，Pickler 要比重复调用 dump 更方便，也更快。开发者还可以创建 pickle.Pickler 的子类以覆盖 Pickler 方法（特别是 persistent_id 方法），并创建一个持久化框架。但是，这是一项高级技术，本书将不对其进行详细介绍。
Unpickler	Unpickler(*fileobj*) 创建并返回一个对象 u，这样，在将 fileobj 参数传递到 Unpickler 时，调用 u.load 等同于调用 load 函数。要想从一个文件反序列化多个对象，Unpickler 要比重复调用 load 函数更方便，也更快。开发者还可以创建 pickle.Unpickler 的子类以覆盖 Unpickler 方法（特别是 persistent_load 方法），并创建自己的持久化框架。但是，这是一项高级技术，本书将不对其进行详细介绍。

pickle 示例

下面的示例可以处理与前面介绍的 marshal 示例相同的任务，但是使用了 cPickle，而不是 marshal 以将（filename, line-number）数据对列表编码为字符串：

```
import fileinput, cPickle, anydbm
wordPos = {  }
for line in fileinput.input():
    pos = fileinput.filename(), fileinput.filelineno()
    for word in line.split():
        wordPos.setdefault(word,[  ]).append(pos)
dbmOut = anydbm.open('indexfilep','n')
for word in wordPos:
    dbmOut[word] = cPickle.dumps(wordPos[word], 1)
dbmOut.close()
```

开发者可以使用 cPickle 或 pickle 读取被保存在类 DBM 文件 indexfilep 中的数据，如下面的示例所示：

```
import sys, cPickle, anydbm, linecache
dbmIn = anydbm.open('indexfilep')
for word in sys.argv[1:]:
```

```
if not dbmIn.has_key(word):
    sys.stderr.write('Word %r not found in index file\n' % word)
    continue
places = cPickle.loads(dbmIn[word])
for fname, lineno in places:
   print "Word %r occurs in line %s of file %s:" % (word,lineno,fname)
   print linecache.getline(fname, lineno),
```

封装实例

为了让 pickle 可以重新加载一个实例 x，pickle 必须可以从其保存该实例时定义的类的相同模块中导入 x 的类。下面介绍了 pickle 是如何保存类 T 的实例对象 x 的状态，然后将保存的状态重新加载到 T 的一个新实例 y 中的（重新加载的第一步总是创建 T 的一个新的空白实例 y，除非本书在后面明确说明不这样做）。

- 在 T 提供了__getstate__方法时，pickle 将保存调用 T.__getstate__(x)的结果 d。
- 在 T 提供了__setstate__方法时，d 可以是任意类型，并且 pickle 将调用 T.__setstate__(y, d)以重新加载保存的状态。
- 否则，d 必须是一个字典，而 pickle 只需要设置 y.__dict__ = d。
- 否则，在 T 是一个新型类且提供了__getnewargs__方法，并且 pickle 使用值为 2 的 protocol 参数进行封装时，pickle 将保存调用 T.__getnewargs__(x)的结果；t 必须是一个元组。
- 在这种情况下，pickle 不会从一个空白 y 开始，而是通过执行 y = T.__new__(T, *t)创建 y，这样将结束重新加载。
- 否则，在 T 是一个老式类，并且提供了__getinitargs__方法时，pickle 将保存调用 T.__getinitargs__(x)的结果 t（t 必须是一个元组），然后将字典 x.__dict__保存为 d。
- Pickle 首先通过调用 T.__init__(y, *t)重新加载保存的状态，然后调用 y.__dict__.update(d)。
- 否则，在默认情况下，pickle 将把字典 x.__dict__保存为 d。
- 在 T 提供了__setstate__方法时，pickle 将调用 T.__setstate__(y, d)重新加载保存的状态。
- 否则，pickle 只设置 y.__dict__ = d。

pickle 保存和重新加载的对象 d 或 t 中的所有项目必须依次为适合于封装和拆封的类型（也就是，可封装对象）的实例，并且，如有必要，刚才概述的处理过程可能会递归重复执行，直到 pickle 到达最原始的可封装内置类型（字典、元组、列表、集合、数字和字符串等）。

正如第 8.4 节中提到的，特殊方法__getinitargs__、__getnewargs__、__getstate__和__setstate__也可以控制复制和深入复制实例对象的方法。如果一个新型类定义了__slots__方法，并因此其实例不包含__dict__方法，则 pickle 将尽力保存和还原一个等同于槽（slot）的名称和值的字典。但是，这样的新型类必须定义__getstate__和__setstate__方法；否则，其实例不能通过这种尽力而为的方式进行正确地封装和复制。

使用 copy_reg 模块自定义封装

开发者可以将工厂（factory）函数和归约（reduction）函数与 copy_reg 模块进行注册以控制如何使用 pickle 序列化和反序列化任意类型（或新型类）的对象。当开发者在 C 编码的 Python 扩展中定义一个类型时，这样做是特别有用的，尽管这不是唯一的方法。copy_reg 模块提供了以下函数，如表 11-3 所示。

表 11-3

constructor	constructor(*fcon*) 向构造函数表添加 fcon，fcon 列出了 pickle 可能会调用的所有工厂函数。fcon 必须是可调用的，并且通常是一个函数。

pickle	pickle(*type*,*fred*,*fcon*=None)
	将 fred 函数注册为类型为 type 的归约函数，其中 type 必须是一个类型对象（而不是一个老式类）。要想保存类型为 type 的任何对象 o，pickle 模块将调用 fred(o)并保存其结果。fred(o)必须返回一个数据对（fcon, t）或一个元组（fcon, t, d），其中 fcon 是一个构造函数，t 是一个元组。要想重新加载 o，pickle 可以调用 o=fcon(*t)。然后，如果 fred 返回一个 d，则 pickle 使用 d 来还原 o 的状态（如果 o 提供了__setstate__方法，则调用 o.__setstate__(d)；否则，调用 *o*.__dict__.update(*d*)），参见第 11.1 节中介绍的封装实例。如果 fcon 不为 None，则 pickle 还可以调用 constructor(fcon)以将 fcon 注册为一个构造函数。
	pickle 不支持封装代码对象，但是 marshal 支持。下面是一个如何通过将工作委托给 marshal，从而实现自定义封装以支持代码对象的例子（这要感谢 copy_reg 模块）：
	`>>> import pickle, copy_reg, marshal`
	`>>> def viaMarshal(x): return marshal.loads, (marshal.dumps(x),)`
	`...`
	`>>> c=compile('2+2','','eval')`
	`>>> copy_reg.pickle(type(c), viaMarshal)`
	`>>> s=pickle.dumps(c, 2)`
	`>>> cc=pickle.loads(s)`
	`>>> print eval(cc)`
	`4`

shelve 模块

为了提供一个简单和轻量级的持久化机制，shelve 模块可以将 cPickle 模块（或者 pickle 模块，在当前的 Python 安装中没有 cPickle 模块时）、cStringIO 模块（或者 StringIO 模块，在当前的 Python 安装中没有 cStringIO 模块时）和 anydbm 模块（及其用来访问类 DBM 档案文件的底层模块，正如第 11.2 节中介绍的那样）。

shelve 模块提供了一个 open 函数，这个函数是 anydbm.open 的一个多态函数。shelve.open 返回的映射对象 s 要比 anydbm.open 返回的映射对象 a 的限制更少。a 的键和值必须是字符串，s 的键也必须是字符串，而 s 的值可能是任意可封装的类型或类。pickle 模块的自定义函数（例如，copy_reg、__getinitargs__和__setstate__）也可以应用于 shelve 模块，因为 shelve 模块可以委托对 pickle 模块的序列化。

在使用 shelve 和可变对象时，要小心一个很微妙的陷阱。在开发者对“架子（shelf）”上保存的一个可变对象进行操作时，所作的更改是不会被“接受”的，除非开发者将这个被更改的对象赋值回相同的索引上。例如：

```
import shelve s = shelve.open('data')
s['akey'] = range(4)
print s['akey']                          # 打印: [0, 1, 2, 3]
s['akey'].append('moreover')             # 尝试直接更改
print s['akey']                          # 不会接受更改; 打印: [0, 1, 2, 3]

x = s['akey']                  # 提取该对象
x.append('moreover')           # 执行更改
s['akey'] = x                  # 将对象回存
print s['akey']                # 现在更改被接受了，打印: [0, 1, 2, 3, 'moreover']
```

在调用 shelve.open 时，开发者可以通过传递命名参数 writeback=True 来解决这个问题，但是要小心：如果不传递这个参数，则可能严重降低程序的性能。

shelve 示例

下面的示例可以处理与前面的封装和排列示例中相同的任务，但是这个示例使用了 shelve 模块来持久化（filename, line-number）数据对列表：

```
import fileinput, shelve
wordPos = { }
for line in fileinput.input():
   pos = fileinput.filename(), fileinput.filelineno()
for word in line.split():
       wordPos.setdefault(word,[ ]).append(pos)
shOut = shelve.open('indexfiles','n')
for word in wordPos:
   shOut[word] = wordPos[word]
shOut.close()
```

开发者必须使用 shelve 来读取被存储到类 DBM 文件 indexfiles 中的数据，如下面的示例所示：

```
import sys, shelve, linecache
shIn = shelve.open('indexfiles')

for word in sys.argv[1:]:
    if not shIn.has_key(word):
        sys.stderr.write('Word %r not found in index file\n' % word)
        continue
    places = shIn[word]
    for fname, lineno in places:
        print "Word %r occurs in line %s of file %s:" % (word,lineno,fname)
        print linecache.getline(fname, lineno),
```

在本节给出的多个具有相同功能的示例对中，上面这两个示例是最简单和最直接的。这也反映出这样一个事实，shelve 模块要比前面示例中使用的模块层次更高。

11.2 DBM 模块

类 DBM 文件是一个包含字符串对（key, data）的文件，支持根据给定的键提取或存储数据，因此也被称为键存取（keyed access）。类 DBM 文件都是在早期的 UNIX 系统上开发的，其功能大致上相当于那个时代的大型机和小型计算机上常用的访问方法，比如 ISAM，索引顺序访问方法。现在，许多平台上都提供了一些库，这些库可以让使用各种不同语言编写的程序创建、更新和读取类 DBM 文件。

尽管键存取方式不像关系数据库的数据访问功能那么强大，但是通常已经足够满足程序的需要了。如果类 DBM 文件是足够的，开发者可以使用类 DBM 文件实现一个比使用 RDBMS 更小，也更快的程序。

在许多年以前，经典的 dmb 库的第一个版本引入了类 DBM 文件，这个库只提供了有限的功能，但是打算要在许多 UNIX 平台上使用。GUN 版本的 gdbm 库要更丰富，流传也十分广泛。BSD 版本的 dbhash 库提供了非常高级的功能。Python 提供了一些模块，如果在系统上安装了相关的底层库，则这些模块与这几个库都有接口。Python 还提供了一个最小化的 DBM 模块 dumbdbm（可以在任何位置使用，因为其不依赖于安装的其他库）和通用 DBM 模块，这些模块可以自动标识、选择和包装适当的 DBM 库以处理一个已有或新建的 DBM 文件。根据开发者在计算机上安装的操作系统、Python 版本和类 dbm 库，默认的 Python 版本可能会安装这些模块的某个子集。通常，作为最小要求，开发者可以依赖于所有类 UNIX 平台上已有的模块 dbm、Windows 平台上的 dbhash 模块和任何平台上的 dumbdbm 模块。

anydbm 模块

anydbm 模块是一个到任何其他 DBM 模块的通用接口。anydbm 提供了一个单独的工厂函数。

表 11-4

open

open(*filename*,*flag*='r',*mode*=0666)

打开或创建 filename（一个字符串，可以是一个文件的全路径，而不只是一个文件名）指定的 DBM 文件，并返回一个对应于该 DBM 文件的映射对象。在 DBM 文件已经存在时，open 将使用 whichdb 模块确定哪个 DBM 库可以处理这个文件。在 open 创建一个新 DBM 文件时，open 将按下面的引用顺序选择第一个可用的 DBM 模块：dbhash、gdbm、dbm 或 dumbdbm。

flag 是一个单字符字符串，用来告诉 open 如何打开文件和是否创建文件，如下表所示。mode 是一个整数，如果 open 将要创建一个文件，则使用这个整数作为该文件的权限比特，参见第 10.3 节。并不是所有的 DBM 模块都使用 flags 和 mode 的，但是，为了提供可移植性，在开发者调用任何 anydbm.open 时，应该总是为这些参数提供适当的值。

anydbm.open 的 flag 值

Flag	只读	如果文件已经存在	如果文件不存在
'r'	是	open 将打开该文件	open 将引发错误
'w'	否	open 将打开该文件	open 将引发错误
'c'	否	open 将打开该文件	open 将创建该文件
'n'	否	open 将截短该文件	open 将创建该文件

anydbm.open 将返回一个具有字典的功能子集的映射对象 m（参见第 4.8 节）。m 只能以键和值的形式接受字符串，并且 m 提供的唯一映射方法就是 m.has_key 和 m.keys。开发者可以使用假定 m 是一个字典时使用的相同索引语法 m[key]以绑定、重新绑定、访问和解除绑定 m 中的项目。如果 flag 为 'r'，则 m 是只读的，因此开发者只能访问 m 的项目，而不能绑定、重新绑定或解除绑定这些项目。m 提供的另外一个方法就是 m.close，这个方法具有与文件对象的 close 方法相同的语义。就像在文件对象中使用的那样，开发者必须确保在完成对 m 的使用之后调用 m.close( )。try/finally 语句（参见第 6.1 节）是确保终止化的最好方法（但是，在 Python 2.5 中，with 语句甚至要比 try/finally 更好一些，参见第 6.1 节中介绍的 with 语句。

dumbdbm 模块

dumbdbm 模块提供了最小化的 DBM 功能和最普通的性能。dumbdbm 的好处在于可以在任何地方使用，因为 dumbdbm 不依赖于任何库。开发者通常都不需要 import dumbdbm；而是 import anydbm，并让 anydbm 向程序提供可用的最好 DBM 模块，如果当前的 Python 安装上没有更好的 DBM 可以使用，则默认使用 dumbdbm。开发者需

要直接导入 dumbdbm 的时候是极少的，唯一的情况是在需要创建一个类 DBM 文件，以便将来可以从任意 Python 安装读取时。dumbdbm 模块提供了一个 open 函数和一个异常类 error，该异常类与 anydbm 的异常类具有多态性。

dbm、gdbm 和 dbhash 模块

dbm 模块只存在于 UNIX 平台中，该模块可以包装任意 dbm、ndbm 和 gdbm 库，因为这些库都提供了 dbm 兼容接口。开发者几乎从来不直接 import dbm；而是 import anydbm，并让 anydbm 向程序提供可用的最好 DBM 模块，在适当情况下也包括 dbm。dbm 模块提供了一个 open 函数和一个异常类，该异常类与 anydbm 的异常类具有多态性。

gdbm 模块可以包装 GNU DBM 库 gdbm。gdbm.open 函数可以接受 flag 参数的其他值并返回一个包含其他几个额外方法的映射对象 m。开发者可以直接 import gdbm 以访问不可移植的功能。因为本书主要关注于跨平台的 Python，因此并没有特意介绍 gdbm。

dbhash 模块可以按 DBM 兼容方式包装 BSDDB 库。dbhash.open 函数可以接受 flag 参数的其他值并返回一个包含其他几个额外方法的映射对象 m。开发者可以直接 import dbhash 以访问不可移植的功能。不过，要想完全访问 BSD DB 的功能，则必须 import bsddb，参见第 11.3 节。

whichdb 模块

whichdb 模块可以尝试猜测几个 DBM 模块中的哪一个是适合使用的。whichdb 提供了一个单独的函数。

表 11-5

whichdb	`whichdb(`*filename*`)` 打开 filename 指定的文件以发现哪个类 DBM 包创建了该文件。如果该文件不存在或者不能打开和读取，则 whichdb 返回 None。如果该文件存在，并且可以被打开和读取，但是不能确定是哪个类 DBM 包创建了该文件（通常，这意味着这个文件不是一个 DBM 文件），则 whichdb 返回 ‘ ’。如果 whichdb 找到了哪个模块可以读取该类 DBM 文件，则返回这个模块的字符串名称，比如 ‘dbm’、‘dumbdbm’ 或 ‘dbhash’。

类 DBM 文件的使用示例

键存取适合于在程序需要持久地记录等同于 Python 字典的数据，其中包含作为键和值的字符串。例如，假定开发者需要分析几个文本文件，这些文件的名称被给定为程序的参数，并且记录了每个单词出现在这些文件的什么位置。在这种情况下，键就是这些单词，也就是基本的字符串。开发者需要为每个单词记录的是一个由（filename,

line-number）数据对组成的列表。不过，开发者需要以几种方式将这些数据编码为字符串——例如，通过利用路径分隔符字符串 os.pathsep（参见第 10.8 节）通常不会出现在文件名中的事实（请注意，有关将数据编码成字符串的常见问题的更具体、通用和可靠的解决方法，请参见第 11.1 节）。使用这种简化方法，记录文件中单词位置的程序代码如下：

```
import fileinput, os, anydbm
wordPos = {  }
sep = os.pathsep
for line in fileinput.input():
    pos = '%s%s%s'%(fileinput.filename(), sep, fileinput.filelineno())
    for word in line.split():
        wordPos.setdefault(word,[  ]).append(pos)
dbmOut = anydbm.open('indexfile','n')
sep2 = sep * 2
for word in wordPos:
    dbmOut[word] = sep2.join(wordPos[word])
dbmOut.close()
```

开发者可以使用几种方法读取被保存到类 DBM 文件 indexfile 中的数据。下面的示例可以接受单词作为程序的命令行参数，并打印请求的单词出现的行：

```
import sys, os, anydbm, linecache
dbmIn = anydbm.open('indexfile')
sep = os.pathsep
sep2 = sep * 2
for word in sys.argv[1:]:
    if not dbmIn.has_key(word):
        sys.stderr.write('Word %r not found in index file\n' % word)
        continue
    places = dbmIn[word].split(sep2)
    for place in places:
        fname, lineno = place.split(sep)
        print "Word %r occurs in line %s of file %s:" % (word,lineno,fname)
        print linecache.getline(fname, int(lineno)),
```

11.3 Berkeley DB 接口

Python 附带了 bsddb 包，如果系统中安装了 Berkeley 数据库（也被称为 BSD DB）库，并且编译的 Python 的安装版本支持该库，则 bsddb 包可以包装 BSD DB 库。使用 BSD DB 库，开发者可以创建哈希、二叉树和基于记录的文件，这些文件通常就像持久化的字典一样。在 Windows 上，Python 中包括一个 BSD DB 库的移植版本，这样可以确保 bsddb 模块总是可用的。要想下载可以用于其他平台的 BSD DB 源代码、二进制文件，以及有关 BSD DB 的详细文档，请参见 http://www.sleepycat.com。

简化的和完整的 BSD DB Python 接口

bsddb 模块本身提供了一个简化的、向后兼容的接口，以访问 BSD DB 功能的一个子集，参见 http://www.python.org/doc/2.4/lib/module-bsddb.html 上的 Python 在线文档。但是，标准 Python 库还在 bsddb 包中附带了许多模块，以 bsddb.db 开始。这个模块集合非常接近地模仿了 BSD DB 当前丰富、复杂的功能和接口，参见 http://pybsddb.sourceforge.net/bsddb3.html 上的文档。在这个链接上，可以看到这个包的文档有一个稍微不同的名称 bsddb3，这是一个可以单独下载，甚至可以在非常老的 Python 版本上安装的包的名称。但是，要想使用作为 Python 标准库的一部分的 bsddb 包版本，开发者需要导入的是名为 bsddb.db 和具有相似名称的模块，而不是 bsddb3.db 和具有相似名称的模块。除了这个命名上的细节，Sourceforge 文档完全可以应用于 Python 标准库中的 bsddb 包中的模块（db、dbshelve、dbtables、dbutil、dbobj 和 dbrecio）。

有关 BSD DB 的所有接口及其功能，可以（和已经）编写了一整本书，因此本书不打算介绍这个丰富、完整和复杂的接口（如果读者需要了解 BSD DB 的完整功能，本书建议除了学习前面提到的一些链接上的文档，还可以参考 New Riders 出版，Sleepcat Software 编著的 *Berkeley DB* 一书）。但是，在 Python 中，开发者还可以以一种简单得多的方式访问 BSD DB 功能的一个较小，但是十分重要的子集，这是通过 bsddb 模块提供的简化接口实现的，下面将介绍该模块。

bsddb 模块

bsddb 模块提供了 3 个工厂函数：btopen、hashopen 和 rnopen。

表 11-6

btopen, hashopen, rnopen	`btopen(filename,flag='r',*many_other_optional_arguments)`
	`hashopen(filename,flag='r',*many_other_optional_arguments)`
	`rnopen(filename,flag='r',*many_other_optional_arguments)`
	btopen 可以打开或创建 filename（一个字符串，包含文件的全部路径，而不只是文件名）指定的二叉树文件，并返回一个 BTree 对象以访问和操作该文件。flag 参数与 anydbm.open 的 flag 参数具有相同的值和含义。其他参数指定了一些用来进行细粒度控制的选项，但是极少使用。
	hashopen 与 rnopen 的工作方式相同，但是这两个函数分别用来打开或创建哈希格式和记录格式的文件，并分别返回 Hash 和 Record 类型的对象。hashopen 通常是最快的格式，并且在使用键来查找记录时是最有意义的。但是，如果开发者还需要按排序顺序访问记录，可以使用 btopen；如果开发者需要按照最初写入记录的相同顺序访问这些记录，可以使用 rnopen。而使用 hashopen 不会按记录在文件中的顺序保留这些记录。
	只要键和值都是字符串，任何类型为 BTree、Hash 和 Record 的对象 b 都可以被索引为一个映射。进一步讲，b 还支持通过当前记录（current record）的概念实现顺序访问。b 提供了以下这些方法。

close	`b.close()` 关闭 b。在执行 b.close( )之后，不能再调用其他方法。

续表

first	*b*.first() 将 b 的当前记录设置为第一个记录并返回对应于第一个记录的数据对（key, value）。除了 BTree 对象，记录的顺序是任意的，这可以确保记录按照键的字母顺序排序。如果 b 为空，则 b.first( )将引发 KeyError 异常。
has_key	*b*.has_key(*key*) 如果字符串 key 是 b 中的一个键，则返回 True；否则，返回 False。
keys	*b*.keys() 返回由 b 的键字符串组成的列表。除了 BTree 对象之外，返回的键的顺序是任意的，而 BTree 对象返回的键将按字母顺序排列。
last	*b*.last() 将 b 的当前记录设置为最后一个记录，并返回最后一个记录的数据对（key, value）。Hash 类型没有提供 last 方法。
next	*b*.next() 将 b 的当前记录设置为下一个记录，并返回下一个记录的数据对（key, value）。如果 b 没有下一个记录，则 b.next( )将引发 KeyError 异常。
previous	*b*.previous() 将 b 的当前记录设置为上一个记录，并返回上一个记录的数据对（key, value）。Hash 类型没有提供 previous 方法。
set_location	*b*.set_location(*key*) 将 b 的当前记录设置为键字符串为 key 的项目，并返回一个数据对 (key, value)。如果 key 不是 b 中的键，并且 b 不是 BTree 类型，b.set_location(key)将把 b 的当前记录设置为其键为大于 key 的最小键的项目，并返回这个键/值数据对。对于其他对象类型，如果 key 不是 b 中的一个键，则 set_location 将引发 KeyError 异常。

Berkeley DB 的使用示例

Berkeley DB 适合于执行的任务与类 DBM 文件适合于执行的任务相似。实际上，anydbm 使用了 dbhash（可以访问 BSD DB 的类 DBM 接口）来创建新的类 DBM 文件。此外，在直接使用 bsddb 时，BSD DB 还允许使用其他文件格式。对于键存取而言，二叉树格式不如哈希格式快，但是在开发者还需要按字母顺序访问键时，二叉树格式是非常好的。

下面的示例实现了与前面显示的 DBM 示例相同的任务，但是下面的示例使用的是 bsddb，而不是 anydbm：

```
import fileinput, os, bsddb
wordPos = {  }
sep = os.pathsep
for line in fileinput.input():
    pos = '%s%s%s'%(fileinput.filename(), sep, fileinput.filelineno())
    for word in line.split():
```

```
        wordPos.setdefault(word,[ ]).append(pos)
btOut = bsddb.btopen('btindex','n')
sep2 = sep * 2
for word in wordPos:
    btOut[word] = sep2.join(wordPos[word])
btOut.close()
```

这个示例与 DBM 的示例之间的区别很小：使用 bsddb 写入一个二叉树格式的文件与使用 anydbm 写入一个新的类 DBM 文件基本上是一个相同的任务。使用 bsddb.btopen ('btindex')，而不是 anydbm.open('indexfile')读取数据的任务也是相似的。为了描述二叉树的其他一些与按字母顺序访问键有关的功能，下面将解决一个更通用的任务。下面的示例将把其命令行参数看作是指定了单词开头，然后打印出现了这样的开头的任何单词所在的行：

```
import sys, os, bsddb, linecache
btIn = bsddb.btopen('btindex')
sep = os.pathsep
sep2 = sep * 2

for word in sys.argv[1:]:
    key, pos = btIn.set_location(word)
if not key.startswith(word):
    sys.stderr.write('Word-start %r not found in index file\n' % word)
    while key.startswith(word):
        places = pos.split(sep2)
        for place in places:
            fname, lineno = place.split(sep)
            print "%r occurs in line %s of file %s:" % (word,lineno,fname)
            print linecache.getline(fname, int(lineno)),
    try: key, pos = btIn.next()
    except IndexError: break
```

这个示例利用了这样一个事实，那就是，在 word 本身不是 btIn 中的一个键时，btIn.set_location 可以将 btIn 的当前位置设置为大于 word 的最小键。在 word 是一个单词的开头，并且这些键都是单词时，这意味着 set_location 将按字母顺序把当前位置设置为第一个以 word 开始的单词。使用 key.startswith(word)进行的 while 循环测试操作可以检查这个示例在继续扫描以 word 开头的单词，并将在不再出现以 word 开头的单词时终止 while 循环。上面的示例在 while 语句之前的一个 if 语句中第一次执行了这样的测试，因为该示例想要首先分离出没有任何单词以 word 开始的情况，并在这种特殊情况下输出一个错误消息。

11.4 Python 数据库 API（DBAPI）2.0

正如本书前面提到的，Python 标准库并没有附带一个 RDBMS 接口，但是有许多免费

的第三方模块可以帮助 Python 程序访问特殊的数据库。这些模块大多数遵循 Python 数据库 API 2.0 标准，也被称为 DBAPI。将来有可能会出现 DBAPI 的一个新版本（可能会被称为 3.0），但是，当前并没有对此制定严格的计划和时间表。如果 DPIAPI 3.0 到来，毫无疑问会为将来的程序提供进一步的增强功能，并由此给这些程序带来很大的好处，并且，使用 DBAPI 2.0 编写的程序只需要经过最小的更改，或者不经过更改就能适用于将来的 DBAPI 3.0 版本。

如果 Python 程序只在 Windows 上运行，开发者可能更愿意通过 COM 使用 Microsoft 的 ADO 软件包来访问数据库。要想了解更多有关在 Windows 上运行 Python 程序的信息，参见 O'Reilly 出版，Mark Hammond 和 Andy Robinson 编著的 *Python Programming on Win32*。因为 ADO 和 COM 都用于专门的平台，而本书主要关注 Python 的跨平台使用，因此本书没有进一步介绍 ADO 和 COM。不过，读者可以在 http://adodbapi.sourceforge.net/ 找到一个非常有用的 Python 扩展，该扩展可以通过 DBAPI 间接访问 ADO。

在导入了一个符合 DBAPI 标准的模块之后，可以使用适当的参数调用该模块的 connect 函数。connect 将返回一个 Connection 实例，该实例表示一个数据库连接。Connection 实例提供了 commit 和 rollback 方法以处理事务、提供了一个 close 方法以在完成了对数据库的操作之后调用该方法，还提供了一个 cursor 方法以返回一个 Cursor 实例。游标（cursor）对象提供了用于数据库操作的方法和属性。符合 DBAPI 标准的模块还提供了异常类、descriptive 属性、工厂函数和类型描述属性。

异常类

符合 DBAPI 标准的模块提供了异常类 Warning、Error 和 Error 的几个子类。Warning 指出了像插入时数据被切断这样的异常。Error 的子类指出了程序在处理数据库，以及与数据库接口的符合 DBAPI 标准的模块时遇到的各种类型的错误。通常，异常处理代码可以使用下面这种形式的语句：

```
try:
   ...
except module.Error, err:
   ...
```

这条语句可以捕捉需要处理的所有与数据库相关的错误，而不会终止程序。

线程安全

在符合 DBAPI 标准的模块具有一个大于 0 的属性 threadsafety 时，该模块将为数据库接口断言某种级别的线程安全。比依赖于这种方法更好的做法是，确保一个单线程可以排他性地访问任何给定的外部资源，比如数据库，这样做要更安全和更有可移植性，参见第 14.5 节中的概述。

参数格式

符合 DBAPI 标准的模块具有一个属性 paramstyle，这个属性可以用来标识用作参数的占位符的标记符号的格式。在传递到 Cursor 实例的方法（比如 execute 方法）中的 SQL 语句字符串中插入这样的标记符号可以使用运行时确定的参数值。例如，假定开发者需要提取 AFIELD 字段等于 Python 变量 x 的当前值的数据表 ATABLE 的行。假定游标实例名为 c，开发者可以像下面这样使用 Python 的字符串格式化操作符%来执行这个任务：

```
c.execute('SELECT * FROM ATABLE WHERE AFIELD=%r' %x)
```

但是，这并不是本书推荐的方法。这种方法将为 x 的每个值生成一个不同的语句字符串，并要求这样的语句每次都重新进行解析和准备。使用参数替换，开发者可以向 execute 传递单个语句字符串，这是使用占位符，而不是参数值来实现的。这可以让 execute 只执行一次解析和准备，从而潜在地获得更好的性能。例如，如果一个模块的 paramstyle 属性为‘qmark’，可以将上面的查询表示为：

```
c.execute('SELECT * FROM ATABLE WHERE AFIELD=?',[x])
```

只读字符串属性 paramstyle 可以告诉开发者的程序如何在这个模块中使用参数替换。paramstyle 的可能值为：

format

标记符号为%，就像在字符串格式化中一样。这样的查询可以写作：

```
c.execute('SELECT * FROM ATABLE WHERE AFIELD=%s', [x])
```

named

标记符号为:name，并且参数将被命名。这样的查询可以写作：

```
c.execute('SELECT * FROM ATABLE WHERE AFIELD=:x',{'x':x})
```

numeric

标记符号为：n，给定了参数的数量。这样的查询可以写作：c.execute('SELECT * FROM ATABLE WHERE AFIELD=1', [x])

```
c.execute('SELECT * FROM ATABLE WHERE AFIELD=:1', [x])
```

pyformat

标记符号为%(name)s，并且参数将被命名。这样的查询可以写作：

```
c.execute('SELECT * FROM ATABLE WHERE AFIELD=%(x)s', {'x':x})
```

qmark

标记符号为？。这样的查询可以写作：

```
c.execute('SELECT * FROM ATABLE WHERE AFIELD=?',[x])
```

在 paramstyle 既不是‘pyformat’，也不是‘nor’时，execute 方法的第二个参数是一个

序列。在参数被命名时（也就是，paramstyle 是‘pyformat’或者‘named’），execute 方法的第二个参数是一个字典。

工厂函数

通过占位符传递到数据库的参数通常必须是正确的类型：这意味着 Python 数字（整数或浮点型值）、字符串（普通或 Unicode）和 None 以表示 SQL NULL。没有一种类型可以普遍适用于表示日期、时间和二进制大对象（BLOB）。符合 DBAPI 标准的模块提供了工厂函数以建立这样的对象。大多数符合 DBAPI 标准的模块为了这个目而使用的类型是 datetime 和 mxDateTime 模块（参见第 12 章）提供的那些类型，以及用于 BLOB 的字符串或缓冲区类型。DBAPI 指定的工厂函数如表 11-7 所示。

表 11-7

Binary	`Binary(string)` 返回一个对象，这个对象将给定的字节字符串表示为一个 BLOB。
Date	`Date(year,month,day)` 返回一个对象，表示指定的日期。
DateFromTicks	`DateFromTicks(s)` 返回一个对象，表示 time 模块的从新纪元时间开始之后 s 秒对应的日期，参见第 12 章。例如，DateFromTicks(time.time( ))的结果是“today”。
Time	`Time(hour,minute,second)` 返回一个对象，表示指定的时间。
TimeFromTicks	TimeFromTicks(*s*) 返回一个对象，表示 time 模块的从新纪元时间开始之后 s 秒对应的时间，参见第 12 章。例如，TimeFromTicks(time.time( ))的结果是“now”。
Timestamp	`Timestamp(year,month,day,hour,minute,second)` 返回一个对象，表示指定的日期和时间。
TimestampFromTicks	`TimestampFromTicks(s)` 返回一个对象，表示 time 模块的从新纪元时间开始之后 s 秒对应的日期和时间，参见第 12 章。例如，TimestampFromTicks(time.time( ))的结果是当前日期和时间。

类型描述属性

Cursor 实例的属性 description 描述了一个查询结果的每个列的类型和其他特征。每个列的类型（描述该列的元组的第二个项目）应该等于下面列出的符合 DBAPI 标准的模块的属性中的某一个属性。

BINARY

描述包含 BLOB 的列；

DATETIME

描述包含日期、时间，或者同时包含日期和时间的列；

NUMBER

描述包含任何类型的数字的列；

ROWID

描述包含行标识号的列；

STRING

描述包含任何类型的文本的列。

游标的描述主要用于对程序正在处理的数据库进行自测，尤其是对每个列的类型。这样的自测可以帮助开发者编写通用模块，并使用不同的结构描述处理表，包括开发者在编写代码时可能还不知道的结构描述。

connect 函数

符合 DBAPI 标准的模块的 connect 函数可以接受各种取决于数据库的类型和包含的特殊模块的参数。DBAPI 标准推荐 connect 接受命名参数，但是并不强制要求。特别是，connect 必须至少可以接受具有以下名称的可选参数：

database

要连接的特定数据库的名称；

dsn

用于数据库连接的数据源的名称；

host

运行数据库的主机的名称；

password

用于数据库连接的密码；

user

用于数据库连接的用户名。

Connection 对象

符合 DBAPI 标准的模块的将返回一个对象 x，这个对象是 Connection 类的一个实例。x 提供了如表 11-8 所示的方法。

表 11-8

close	*x*.close() 终止数据库连接并释放所有相关资源。在完成了对数据库的操作之后，要立即调用 close 方法。不必要地始终保持打开数据库连接会造成系统资源的严重消耗。
commit	*x*.commit() 提交数据库中的当前会话。如果该数据库不支持会话，x.commit( )也只是一个没有什么坏处的空操作。
cursor	*x*.cursor() 返回 Cursor 类的一个新实例，参见第 11.4 节。
rollback	*x*.rollback() 重新运算数据库中的当前会话。如果该数据库不支持会话，x.rollback( )将引发一个异常。DBAPI 推荐（但是并不强制要求），对于不支持会话的数据库，Connection 类不提供 rollback 方法，因此 x.rollback( )将引发 AttributeError 异常。开发者可以使用 hasattr(*x*,'rollback')测试数据库是否支持会话。

Cursor 对象

Connection 实例提供了一个 cursor 方法，该方法可以返回一个对象 c，这个对象是 Cursor 类的一个实例。SQL 游标（cursor）表示查询结果的集合，可以帮助开发者按顺序使用这个集合中的记录，一次处理一个记录。DBAPI 建模的游标是一个很丰富的概念，因为游标表示程序首先执行 SQL 查询的唯一方法。另一方面，DBAPI 游标只允许开发者在结果序列中向前移动（某些关系数据库还提供了功能更丰富的游标，可以向后和向前移动游标，但不是所有数据库都提供了这种功能），并且不支持 SQL 子句 WHERE CURRENT OF CURSOR。DBAPI 游标的这些限制使得符合 DBAPI 标准的模块甚至可以在根本不提供真正的 SQL 游标的 RDBMS 数据库上提供游标。Cursor 类的实例 c 提供了许多属性和方法；表 11-9 列出了最常用的属性和方法。

表 11-9

close	*c*.close() 关闭游标并释放所有相关资源。
description	一个只读属性，是一个由 7 个项目的元组组成的序列，在上一次执行的查询结果中每个列包含一个项目： *name, typecode, displaysize, internalsize, precision, scale, nullable* 如果对 c 的上一次操作不是一个查询操作，或者返回了一个不能使用的列描述，则 c.description 为 None。游标的描述主要用于对程序正在处理的数据库进行自测。这样的自测可以帮助开发者编写通用模块，并使用不同的结构描述处理表，包括开发者在编写代码时可能还不知道的结构描述。
execute	*c*.execute(*statement*,*parameters*=None) 使用给定参数 parameters 对数据库执行一个 SQL 语句字符串 statement，在该模块的 paramstyle 为‘format’、‘numeric’或‘qmark’时，parameters 是一个序列，而在 paramstyle 为‘named’或‘pyformat’时，parameters 是一个字典。

续表

executemany	*c*.executemany(*statement*,**parameters*) 对数据库执行一个 SQL 语句 statement，按照给定参数 parameters 中的每个项目执行一次 SQL 语句。在该模块的 paramstyle 为 'format'、'numeric' 或 'qmark' 时，parameters 是一个由序列组成的序列，在 paramstyle 为 'named' 或 'pyformat' 时，parameters 是一个由字典组成的序列。例如下面的语句： c.executemany('UPDATE atable SET x=? WHERE y=?',(12,23),(23,34)) 在 paramstyle 为 'qmark' 时，上面的语句等同于下面的两条语句，但是执行更快： c.execute('UPDATE atable SET x=12 WHERE y=23') c.execute('UPDATE atable SET x=23 WHERE y=34')
fetchall	*c*.fetchall() 以元组序列的形式返回上次查询的所有剩余结果行。如果上次查询不是一个 SELECT 查询，则引发一个异常。
fetchmany	*c*.fetchmany(*n*) 以元组序列的形式返回上次查询的最多 n 个剩余结果行。如果上次查询不是一个 SELECT 查询，则引发一个异常。
fetchone	*c*.fetchone() 以元组的形式返回上次查询的下一个结果行。如果上次查询不是一个 SELECT 查询，则引发一个异常。
rowcount	一个只读属性，指定了上次操作读取或影响的行数，如果该模块不能确定这个值，则值为-1。

符合 DBAPI 标准的模块

不管开发者想要使用什么样的关系数据库，至少可以从 Internet 下载一个（通常多于一个）Python 的符合 DBAPI 标准的模块。除了 mxODBC（和 SAPDB，该模块使用了 GPL）之外，下面列出的所有模块都有类似于 Python 的自由许可证：开发者可以在开源程序或者不公开的源程序中免费使用。mxODBC 可以免费用于非商业用途，但是，要想用于商业用途，开发者必须购买许可证。目前可以使用的关系数据库是非常多的，本书无法一一列举，但是，一些最常用的关系数据库如下。

ODBC

开放数据库互联（Open DataBase Connectivity，ODBC）是一个可以连接到许多不同数据库的标准方法，包括其他一些符合 DBAPI 标准的模块不支持的数据库，比如 Microsoft Jet（也被称为 Access 数据库）。Python 的 Windows 版本包含一个 odbc 模块，但是这个模块是不被支持的，该模块遵循 DBAPI 的一个老版本，而不是当前的 2.0 版本。在 UNIX 或 Windows 上，可以使用 mxODBC（http://www.lemburg.com/files/Python/mxODBC.html）。mxODBC 的 paramstyle 为 'qmark'。其 connect 函数可以接受 3 个可选参数：dsn、user 和 password。

Oracle

Oracle 是一个广泛使用的商用 RDBMS。要想建立与 Oracle 的接口，可以使用

DCOracle2，参见 http://www.zope.org/Members/matt/dco2。DCOracle2 的 paramstyle 是‘numeric’。其 connect 函数可以接受单个可选的、未命名的参数字符串，该字符串使用以下语法：

```
'user/password@service'
```

cx_orcale（http://www.python.net/crew/atuining/cx_Oracle/index.html）是另一个选择。paramstyle 是‘named’；connect 函数可以接受一个与 DCOracle2 具有相同格式的字符串，或者多个可选参数，名为 dsn、user、passwrd 及其他。

Microsoft SQL Server

要想建立与 Microsoft SQL Server 的接口，本书推荐使用 mssqldb 模块，参见 http://www.object-craft.com.au/projects/mssql/。mssqldb 的 paramstyle 是‘qmark’。其 connect 函数可以接受 3 个参数——名为 dsn、user 和 passwd——以及一个可选参数 database。pymssql（http://pymssql.sourceforge.net/）是一个可选项。

DB/2

对于 IBM DB/2，可以使用 DB2 模块，参见 http://sourceforge.net/projects/pydb2。DB2 的 paramstyle 是‘format’。其 connect 函数可以接受 3 个可选参数：名为 dsn、uid 和 pwd。

MySQL

MySQL 是一个广泛使用的开源 RDBMS。要想建立与 MySQL 的接口，可以使用 MySQLdb，参见 http://sourceforge.net/projects/mysql-python。MySQLdb 的 paramstyle 是‘format’。其 connection 函数可以接受 4 个可选参数：名为 db、host、user 和 passwd。

PostgreSQL

PostgreSQL 是一个非常好的开源 RDBMS。要想建立与 PostgreSQL 的接口，本书推荐使用 psycopg，参见 http://initd.org/Software/psycopg。psycopg 的 paramstyle 是‘pyformat’。其 connect 函数可以接受单个强制字符串参数，名为 dsn，该参数使用下面的语法：

```
'host=host dbname=dbname user=username password=password'
```

SAP DB

SAP DB，也被称为 Adabas，是一个功能强大的 RDBMS，曾经是不公开的源代码，但是现在是开源代码。SAP DB 附带了 sapdbapi（参见 http://www.sapdb.org/sapdbapi.html）和其他一些有用的 Python 模块。sapdbapi 的 paramstyle 是‘pyformat’。其 connect 函数可以接受 3 个强制参数——名为 user、password 和 database——和一个名为 host 的可选参数。

Gadfly

Gadfly（http://gadfly.sf.net）不是一个用来连接到其他 RDBMS 的接口，而是一个使用 Python 编写的完整 RDBMS 引擎。Gadfly 支持标准 SQL 的一个很大的子集。例如，Gadfly

缺少 NULL，但是支持 VIEW。Gadfly 可以作为后台（daemon）服务器运行，这样，客户机可以通过 TCP/IP 连接到 Gadfly。另外，如果开发者不需要其他进程也可以并发访问相同的数据库，可以直接在应用程序的进程中运行 Gadfly 引擎。

gadfly 模块与 DBAPI 2.0(参见第 11.4 节)有几点区别，这是因为 Gadfly 实现的是老版本 DBAPI 1.0 的变体。两者在概念上是非常接近的，但是有一些细节上的区别。主要的区别如下。

- gadfly 不提供自定义异常类，因此 Gadfly 在操作失败时将引发普通 Python 异常，比如 IOError、NameError 等。
- gadfly 不提供 paramstyle 属性。但是，该模块的运行相当于提供了一个值为‘qmark’的 paramstyle。
- gadfly 不提供一个名为 connect 的函数；而是使用 gadfly.gadfly 或 gadfly.client.gfclient 函数。
- gadfly 不提供用于数据类型的工厂函数。
- Gadfly 游标不提供 executemany 方法。相反，在 SQL 语句为 INSERT 的特殊情况下，execute 方法将可选地接受一个元组列表作为其第二个参数，并插入所有数据。
- Gadfly 游标不提供 rowcount 方法。

gadfly 模块提供了如表 11-10 所示的函数。

表 11-10

gadfly	`gadfly.gadfly(dbname,dirpath)` 返回一个用于访问名为 dbname 的数据库的连接对象，这个数据库必须已经在 dirpath 字符串指定的目录中被创建。数据库引擎与应用程序在相同的进程中运行。
gfclient	`gadfly.client.gfclient(policyname, port, password, host)` 返回一个用于访问给定 host 和 port 上的 gfserve 进程服务的数据库的连接对象，policyname 标识了访问所要求的级别，通常使用‘admin’指定无限制访问。

SQLite

SQLite（http://www.sqlite.org）类似于 Gadfly，SQLite 不是一个用来连接到其他 RDBMS 的接口，而是一个完整和独立的 RDBMS 引擎。但是，SQLite 是使用 C 编写的，与 Gadfly 相比，SQLite 可能会提供更好的性能，并可以通过多种编程语言对其进行访问。可以连接到 SQLite 的最流行的 Python 接口是 PySQLite，参见 http://initd.org/tracker/pysqlite。PySQLite 与 DBAPI 2.0 非常兼容，区别只是 PySQLite 不支持类型（所有数据实际上都被 SQLite 保存为字符串）。PySQLite 的 paramstyle 为‘qmark’。另外一个可选的 Python 接口是 APSW，参见 http://www.rogerbinns.com/apsw.html。APSW 甚至没有试图符合 DBAPI 2.0 的规范，但是非常忠实地实现了 SQLite 自己的常用 API。

在 Python 2.5 中，PySQLite 以包 sqlite3 的形式被包含在 Python 标准库中。

第 12 章

时间操作

Python 程序可以使用几种方法来处理时间。时间间隔是以秒为单位（秒的小数部分就是时间间隔的小数部分）的浮点型数字。特定的时刻是以从被称为新纪元时间（epoch）的参考时刻开始到当前时刻的秒数来表示的，UNIX 和 Windows 平台上都使用一个通用的新纪元时间（epoch），也就是按“协调世界时”（Coordinated Universal Time，UTC）时间，1970 年 1 月 1 日的午夜。时刻通常还需要被表示为各种度量单位（例如，年、月、日、小时、分钟和秒）的混合表达式，特别是在用于输入/输出（I/O）的时候。当然，I/O 还需要具备可以将时间和日期格式化为人们可以理解的字符串，并从字符串格式中解析时间和日期的功能。

本章将介绍 time 模块，这个模块提供了 Python 的核心时间处理功能。从某种程度上讲，time 模块取决于底层系统的 C 语言库。本章还将展示来自标准 Python 库的 datatime、sched 和 calendar 模块、第三方模块 dateutil 和 pytz，以及常用扩展 mx.DateTime 的一些基本功能。mx.DateTime 已经存在了许多年，其跨平台行为要比 time 更统一，这有助于表明其普及程度，尤其对于数据库接口中日期和时间的表示。

12.1 time 模块

底层 C 语言库决定了 time 模块可以处理的日期的范围。在 UNIX 系统中，1970 年和 2038 年是典型的起止点，也就是 datetime 和 mx.DateTime 需要避免的限制。时刻通常是按 UTC（“协调世界时”，Coordinated Universal Time，曾经被称为 GMT，也就是格林尼治标准时间，“Greenwich Mean Time”）指定的。time 模块还支持本地时区和夏令时（Daylight Saving Time，DST），但是只能用于底层 C 语言系统库提供支持的扩展中。

作为从新纪元时间开始的秒数计时的另一种方法，时刻还可以使用由 9 个整数组成的元组来表示，这个元组被称为时间元组（Time-tuple，参见表 12-1 中介绍的时间元组）。

时间元组中的所有项目都是整数：时间元组不跟踪秒的小数位。时间元组是 struct_time 的一个实例。开发者可以将其用作一个元组，并按表 12-1 中列出的属性名，将该实例中的项目作为只读属性 x.tm_year、x.tm_mon 等进行访问。不管函数在哪里要求一个时间元组参数，开发者都可以传递 struct_time 的一个实例，或者其中的项目为 9 个正确范围内的整数的任何其他序列。

表 12-1　时间表示法的元组形式

项目	含　义	字 段 名	范　围	注　意
0	年	tm_year	1970-2038	在某些平台上范围更广。
1	月	tm_mon	1-12	1 是 1 月；12 是 12 月。
2	日	tm_mday	1-31	
3	小时	tm_hour	0-23	0 是午夜；12 是正午。
4	分钟	tm_min	0-59	
5	秒	tm_sec	0-61	60 和 61 用于闰秒。
6	一周的天数	tm_wday	0-6	0 是星期一；6 是星期日。
7	一年的天数	tm_yday	1-366	一年内的天数
8	DST 标记	tm_isdst	-1−1	-1 表示由库来确定是否使用夏令时。

为了将一个时刻从“新纪元时间之后的秒数”浮点型值翻译为时间元组，可以将该浮点型值传递到一个返回时间元组的函数（例如，localtime），该时间元组中包含所有 9 个有效项目。在向反方向转换时，mktime 将忽略该元组的第 6 个项目（tm_wday）和第 7 个项目（tw_yday）。在这种情况下，开发者通常需要将第 8 个项目（tm_isdst）设置为-1，这样，mktime 本身就可以确定是否应用夏令时了。

time 模块提供了如表 12-2 所示的函数和属性。

表 12-2

asctime	`asctime([tupletime])` 接受一个时间元组并返回一个包含 24 个字符的可读字符串，比如‘Tue Dec 10 18:07:14 2002’。不带任何参数的 asctime( )就像 asctime(localtime(time( )))一样（将当前时刻格式化为本地时间）。
clock	`clock()` 以浮点型秒数返回当前 CPU 的时间。要想测量不同程序的计算开销，time.clock 的值要比 time.time 的值更有用（标准库模块 timeit 甚至会更好，参见第 18.4 节中介绍的 timeit 模块）。其原因在于，在类 UNIX 平台上，time.clock 使用的是 CPU 时间，而不是实测时间，这样使得 time.clock 要比 time.time 较少受到由于计算机加载导致的不可预知的因素的影响。在 Windows 平台上并不存在这个原因，这是因为 Windows 没有 CPU 时间的概念，但是有另外一个原因：time.clock 使用了更高精度的性能计数器计算机时钟。新纪元时间（对应于 time.clock 的结果 0.0 的时间）是任意的，但是在相同进程中连续调用 time.clock 产生的结果之间的时间差是精确的。

续表

ctime	ctime([*secs*]) 与 asctime(localtime(secs))类似（可以接受从新纪元时间开始之后的秒数来表示的时刻，并返回这个本地时刻的一个可读的，包含 24 个字符的字符串格式）。不带任何参数的 ctime( )就像 asctime( )一样（格式化当前时刻）。
gmtime	gmtime([*secs*]) 接受一个从新纪元时间开始之后的秒数表示的时刻，并按 UTC 时间返回一个时间元组 t（t.tm_isdst 总是为 0）。不带任何参数的 gmtime( )就像 gmtime(time( ))一样（返回当前时刻的时间元组）。
localtime	localtime([*secs*]) 接受一个从新纪元时间开始之后的秒数表示的时刻，并按本地时间返回一个时间元组 t（t.tm_isdst 为 0 或 1，这取决于是否要按照当地的规定对时刻 secs 应用夏令时）。不带任何参数的 localtime( )就像 localtime(time( ))一样（返回当前时刻的时间元组）。
mktime	mktime(*tupletime*) 接受一个以本地时间的时间元组表示的时刻，并返回一个浮点型值，这个值以新纪元之后的秒数表示这个时刻。tupletime 中的最后一个项目 DST 有具体的含义：将其设置为 0 以获得太阳时，设置为 1 以获得夏令时，而设置为-1 可以让 mktime 计算夏令时对给定的时刻是否有效。
sleep	sleep(*secs*) 将正在调用的线程暂停 secs 秒（secs 是一个浮点型数字，可以表示以小数形式表示的秒数）。调用线程可能会在 secs 秒之前重新开始执行（如果其主线程和其他信号唤醒了该调用线程），或者需要暂停更长的时间（取决于系统的进程和线程调度）。
strftime	strftime(*fmt*[,*tupletime*]) 接受一个以本地时间中的时间元组的形式表示的时刻，并返回一个将这个时刻按字符串 fmt 指定的格式表示的字符串。如果省略 tupletime，strfttime 将使用 localtime(time( ))（以当前的时区格式化当前时刻）。fmt 字符串的语法类似于第 9.3 节中指定的语法。转换字符有所不同，参见下表。请参考 tupletime 中指定的时刻；另外，开发者不能指定格式中的宽度和精度。

strfttime 的转换字符

类型字符	含义	特殊提示
a	一个星期中某一天的名称，缩写	取决于本地语言环境
A	一个星期中某一天的名称，全称	取决于本地语言环境
b	月份的名称，缩写	取决于本地语言环境
B	月份的名称，全称	取决于本地语言环境
c	完整的日期和时间表示法	取决于本地语言环境
d	一个月中的第几天	1～31
H	小时（24-小时时钟）	0～23

续表

strftime

类型字符	含义	特殊提示
I	小时（12-小时时钟）	1～12
j	一年中的第几天	1～366
m	月份	1～12
M	分钟数	0～59
p	上午（AM）或下午（PM）	取决于本地语言环境
S	秒数	0～61
U	一年中的第几个星期（第一天是星期日）	0～53
w	一个星期中的第几天	0 表示星期日，最大值为 6
W	一年中的第几个星期（第一天是星期一）	0～53
x	完整的日期表示法	取决于本地语言环境
X	完整的时间表示法	取决于本地语言环境
y	一个世纪中的第几年	0～99
Y	年号	1970 到 2038，或者更大范围
Z	时区的名称	如果不存在时区，则为空
%	字面常量%字符	编码为%%

例如，开发者可以使用下面的格式化字符串获得像 asctime 格式化那样的日期（例如，‘Tue Dec 10 18:07:14 2002’）：

```
'%a %b %d %H:%M:%S %Y'
```

开发者可以使用下面的格式化字符串获得符合 RFC822 标准的日期（例如，‘Tue, 10 Dec 2002 18:07:14 EST’）：

```
'%a, %d %b %Y %H:%M:%S %Z'
```

strptime

`strptime(str,fmt='%a %b %d %H:%M:%S %Y')`

根据格式化字符串 fmt 解析 str，并按时间元组格式返回其时刻。该格式化字符串就像前面介绍的 strfttime 一样。

time

`time()`

返回当前时刻，也就是从新纪元时间开始之后的浮点型秒数。在某些平台上，时间度量的精度就是一秒。

timezone

time.timezone 属性是指从 UTC（美洲地区，>0；欧洲、亚洲和非洲的大部分，<=0）到本地时区（不包含夏令时 DST）偏移的秒数。

tzname

time.tzname 属性是一个与本地语言环境相关字符串对，分别是不包含 DST 和包含 DST 的本地时区的名称

12.2 datetime 模块

datetime 模块提供了一些类来表示日期和时间对象，这些类可以包含时区或者不包含时区（默认值）。tzinfo 类是一个抽象类：datetime 模块没有提供实现方法（要想了解所有的详细信息，参见 http://docs.python.org/lib/datetime-tzinfo.html）。参见第 12.3 节介绍的 pytz 模块可以了解 tzinfo 类的一个比较好的，简单的实现，该模块可以帮助开发者很容易地创建 aware 时区的对象。datetime 模块中的所有类型都有不可变的实例，因此这些实例的属性都是只读的，并且这些实例可以是字典 dict 中的键或者集合 set 中的一个项目。

date 类

date 类实例用来表示一个日期（不包含日期中的特定时间信息），并假定总是采用格里历（Gregorian calendar）。date 实例具有 3 个只读整数属性：year、month 和 day。

表 12-3

date	`date(year,month,day)` date 类还提供了一些类方法，可以用作可选的构造函数的。
fromordinal	`date.fromordinal(ordinal)` 返回一个对应于预期的阳历序数 ordinal 的 date 对象，其中值为 1 对应于第 1 年的第 1 天。
fromtimestamp	`date.fromtimestamp(timestamp)` 返回一个对应于 timestamp 时刻的日期对象，timestamp 时刻是从新纪元时间开始之后的秒数来表示的。
today	`date.today()` 返回一个表示今天的日期的 date 对象。 date 类的实例支持某些算术运算：date 实例之间的差是一个 timedelta 实例，并且，开发者可以加上或者减去一个 date 实例和一个 timedelta 实例以构造另一个 date 实例。还可以比较 date 类的任意两个实例（时间上比较靠后的实例被认为是较大的实例）。 date 类的实例 d 还提供了以下这些方法。
ctime	`d.ctime()` 按照与 time.ctime 相同的 24 字符格式，返回一个表示日期 d 的字符串。
isocalendar	`d.isocalendar()` 返回一个由 3 个整数（ISO 年、ISO 星期的第几天和 ISO 星期几）组成的元组。参见 http://www.phys.uu.nl/~vgent/calendar/isocalendar.htm 了解有关国际标准化组织（International Standards Organization，ISO）日历的详细信息。
isoformat	`d.isoformat()` 按照与 str(d)相同的格式‘YYYY-MM-DD’返回一个表示日期 d 的字符串。
isoweekday	`d.isoweekday()` 以整数的形式返回日期 d 是一个星期的第几天；1 表示星期一，7 表示星期日。

续表

replace	*d*.replace(*year*=None,*month*=None,*day*=None) 返回一个新的 date 对象，就像 d 一样，区别只是这些属性被显式指定为参数，而参数对应的属性将被替换为参数的值。例如： date(x,y,z).replace(month=m) == date(x,m,z)
strftime	*d*.strftime(fmt) 按照格式化字符串 fmt 的指定，返回一个表示日期 d 的字符串，例如： time.strftime(*fmt*, *d*.totuple())
timetuple	*d*.timetuple() 返回一个对应于日期 d 和时间 00:00:00（午夜）的时间元组。
toordinal	*d*.toordinal() 返回一个对应于日期 d 的预期阳历序数。例如： date(1,1,1).toordinal() == 1
weekday	*d*.weekday() 以整数的形式返回日期 d 是一个星期的第几天；0 表示星期一，6 表示星期日。

time 类

time 类的实例可以表示一天中的时间（非特定日期），可以是不包含时区或者包含时区，并总是忽略闰秒。这些实例具有 5 个属性：4 个只读整数（hour、minute、second 和 microsecond）和一个可选的 tzinfo（不包含时区的实例的这个属性值为 None）。

表 12-4

time	time(*hour*=0,*minute*=0,*second*=0,*microsecond*=0,*tzinfo*=None) time 类的实例不支持算术运算。开发者可以比较 time 类的两个实例（时间上比较靠后的实例被认为是较大的实例），但是只有在这两个实例都是包含时区的实例或者都是不包含时区的实例时。time 类的实例 t 还提供了以下方法。
isoformat	*t*.isoformat() 返回一个按格式‘HH:MM:SS’表示时间 t 的字符串；这个格式与 str(t)相同。如果 t.microsecond!=0，则结果字符串更长一些：‘HH:MM:SS.mmmmmm’。如果 t 是包含时区的，则还包括 6 个字符，‘+HH:MM’，这 6 个字符将被添加到返回的字符串的末尾以表示该时区到 UTC 的偏移量。换句话讲，这个格式化操作遵循 ISO1601 标准。
replace	*t*.replace(*hour*=None,*minute*=None,*second*=None,*microsecond*=None[,tzinfo]) 返回一个新的 time 对象，就像 t 一样，区别只是这些属性被显式指定为参数，而参数对应的属性将被替换为参数的值。例如： time(x,y,z).replace(minute=m) == time(x,m,z)
strftime	*t*.strftime() 按照格式化字符串 fmt 的指定，返回一个表示时间 t 的字符串。 time 类的实例 t 还提供了 dst、tzname 和 utcoffset 等方法，这些方法将委托到 t.tzinfo，但是，如果 t.tzinfo 为 None，则返回 None。

datetime 类

datetime 类的实例可以表示一个时刻（日期，以及这个日期中的特定时间），可以不包含时区或者包含时区，并总是忽略闰秒。datetime 类是 date 的子类，并添加了 time 的属性；datetime 类的实例具有只读整数属性 year、month、day、hour、minute、second 和 microsecond，以及一个可选的 tzinfo 属性（不包含时区的实例的这个属性值为 None）。

表 12-5

datetime	datetime(*year*,*month*,*day*,*hour*=0,*minute*=0,*second*=0,*microsecond*=0,*tzinfo*=None) datetime 类还提供了一些类方法，可以用作可选的构造函数的。

combine	datetime.combine(*date*,*time*)
	返回一个 datetime 对象，包含从 date 提取的日期属性和从 time 提取的时间属性（包括 tzinfo）。datetime.combine(d, t)相当于：
	`datetime(d.year, d.month, d.day,`
	`t.hour, t.minute, t.second, t.microsecond,`
	`t.tzinfo)`

fromordinal	datetime.fromordinal(*ordinal*) 返回一个对应于给定预期的阳历序数 ordinal 的日期的 datetime 对象，其中值为 1 对应于第 1 年的第 1 天的午夜。
fromtimestamp	datetime.fromtimestamp(*timestamp*,*tz*=None) 返回一个对应于 timestamp 时刻的 datetime 对象，timestampe 时刻是以从本地新纪元时间开始之后的秒数来表示的。在 tz 不为 None 时，将返回一个包含给定 tzinfo 实例 tz 的包含时区的 datetime 对象。
now	datetime.now(*tz*=None) 返回一个表示当前本地日期和时间的 datetime 对象。在 tz 不为 None 时，将返回一个包含给定 tzinfo 实例 tz 的包含时区的 datetime 对象。
today	datetime.today() 返回一个表示当前本地日期和时间的 datetime 对象。
utcfromtime-stamp	datetime.utcfromtimestamp(*timestamp*) 返回一个对应于 timestamp 时刻的不包含时区的 datetime 对象，timestamp 时刻是以从新纪元时间开始之后的秒数来表示的。
utcnow	datetime.utcnow() 返回一个按 UTC 表示当前日期和时间的 datetime 对象。 datetime 类的实例支持某些算术运算：datetime 实例之间的差是一个 timedelta 实例，并且，开发者可以加上或者减去一个 datetime 实例和一个 timedelta 实例以构造另一个 datetime 实例。还可以比较 datetime 类的任意两个实例（时间上比较靠后的实例被认为是较大的实例），但是只有在这两个实例都是包含时区或者都是不包含时区的。datetime 类的实例 d 还提供了以下这些方法。

续表

astimezone	*d*.astimezone(*tz*) 返回一个新的包含时区的 datetime 对象，与 d 相似（d 也必须是包含时区的），区别在于这个时区被转换为 tzinfo 对象 tz 中的一个属性。请注意，d.astimezone(tz)与 d.replace(tzinfo=tz)是有区别的：后者不进行转换，而是直接复制除 d.tzinfo 之外的所有 d 的属性。
ctime	*d*.ctime() 按照与 time.ctime 相同的 24 字符格式，返回一个表示日期 d 的字符串。
date	*d*.date() 返回一个表示与 d 的日期相同的 date 对象。
isocalendar	*d*.isocalendar() 返回一个由 3 个整数（ISO 年、ISO 星期的第几天和 ISO 星期几）组成的元组。参见 http://www.phys.uu.nl/～vgent/calendar/isocalendar.htm 了解有关国际标准化组织（International Standards Organization，ISO）日历的详细信息。
isoformat	*d*.isoformat(sep='T') 按照格式‘YYYY-MM-DDxHH:MM:SS’返回一个表示日期 d 的字符串，其中 x 是参数 sep 的值（必须是一个长度为 1 的字符串）。如果 d.microsecond!=0，则在该字符串的‘SS’部分之后添加 7 个字符‘.mmmmmm’。如果 t 是包含时区的，则在字符串的末尾再添加 6 个字符‘+HH:MM’，以表示从 UTC 到当前时区的偏移量。换句话讲，这个格式化操作遵循 ISO1601 标准。str(d)与 d.isoformat(' ')相同。
isoweekday	*d*.isoweekday() 以整数的形式返回日期 d 是一个星期的第几天；1 表示星期一，7 表示星期日。
replace	*d*.replace(*year*=None,*month*=None,*day*=None,*hour*=None,*minute*=None,*second*=None,*microsecond*=None[,*tzinfo*]) 返回一个新的 datetime 对象，就像 d 一样，区别只是这些属性被显式指定为参数，而参数对应的属性将被替换为参数的值。例如： datetime(x,y,z).replace(month=m) == datetime(x,m,z)
strftime	*d*.strftime(fmt) 按照格式化字符串 fmt 的指定，返回一个表示日期 d 的字符串，例如： time.strftime(*fmt*, *d*.totuple())
time	*d*.time() 返回一个不带时区的时间对象，表示与 d 的时间相同的时间。
timetz	*d*.timetz() 返回一个时间对象，表示与 d 的时间相同，并具有相同的 tzinfo 的时间。
timetuple	*d*.timetuple() 返回一个对应于时刻 d 的时间元组。
toordinal	*d*.toordinal() 返回一个对应于 d 的日期的预期阳历序数。例如： datetime(1,1,1).toordinal() == 1

续表

utctimetuple	*d*.utctimetuple() 返回一个对应于时刻 d 的时间元组；如果 d 是带时区的，则将该时间元组标准化为 UTC。
weekday	*d*.weekday() 以整数的形式返回日期 d 是一个星期的第几天；0 表示星期一，6 表示星期日。 datetime 类的实例 d 还提供了 dst、tzname 和 utcoffset 等方法，这些方法将委托到 d.tzinfo，但是，如果 d.tzinfo 为 None，则返回 None。

timedelta 类

timedelta 类的实例使用了 3 个只读整数属性来表示时间间隔：days、seconds 和 microseconds：

表 12-6

timedelta	timedelta(*days*=0, *seconds*=0, *microseconds*=0, *milliseconds*=0, *minutes*=0, *hours*=0, *weeks*=0)
	使用确定的因子（一个星期为 7 天，一个小时为 3 600 秒等）转换所有单位，并将所有单位标准化为 3 个整数属性，确保 0<=seconds<3600*24 和 0<=microseconds<1000000。例如：
	`print repr(timedelta(minutes=0.5))`
	`# 结果是 datetime.timedelta(0, 30)`
	`print repr(timedelta(hours=-1))`
	`# 结果是 datetime.timedelta(-1, 82800)`
	timedelta 类的实例支持算术运算（在这些实例之间，或者与整数和使用 date 和 datetime 类的实例）和比较运算。要想从一个 timedelta 实例 t 计算对应的以浮点型秒数的时间间隔，可以使用：
	`t.days*86400+t.seconds+t.microseconds/1000000.0`

12.3 pytz 模块

第三方的 pytz 模块提供了创建 tzinfo 实例的最简单方法，这种方法用来创建 time 和 datetime 类的带时区的实例（pytz 是基于时区 Olson 库的模块，参见 http://www.twinsun.com/tz/tz-link.htm 上对 Olson 库的介绍。可以通过 http://pytz. sourceforge.net/了解 pytz）。在程序中避免出现时区错误和陷阱的最好的常用方法是，总是在程序内部使用 UTC 时区，只有在用于显示的时候才转换到其他时区。

pytz 模块提供了一个包含超过 400 个字符串的列表 common_timezones，其中列出了开发者可能需要使用的最常见的时区的名称（大多数是以洲/城市（continent/city）的形式

命名的，还有其他一些像‘UTC’和‘US/Pacific’这样的名称），pytz 模块还提供了一个包含超过 500 个字符串的列表 all_timezones，其中也列出了 这些时区的某些同义词。例如，要想使用 Olson 库标准指定葡萄牙（Portugal）首都里斯本（Lisbon）的时区，通常可以使用字符串‘Europe/Lisbon’来指定，而且这也是可以从 common_timezone 列表中找到的名称；不过，开发者也可以使用‘Portugal’来指定，但是只能在 all_timezones 中找到这个名称。pytz 还提供了 utc 和 UTC（用于相同对象的两个不同名称），以及一个表示协调世界时（UTC）的 tzinfo 实例。

pytz 还提供了如表 12-7 所示的两个函数。

表 12-7

country_timezones	country_timezones(*code*) 返回一个由时区名称组成的列表，该列表对应于双字母 ISO 代码为 code 的国家的时区名称。例如，pytz.country_timezones('US')将返回一个由 22 个字符串组成的列表，从‘America/New_York’到‘Pacific/Honolulu’。

timezone	timezone(*name*)
	返回一个对应于时区名称 name 的 tzinfo 实例。
	例如，要想显示檀香山（Honolulu）时间 2005 年 12 月 31 日午夜在纽约（NewYork）的时间：
	`dt = datetime.datetime(2005,12,31,tzinfo=pytz.timezone ('America/ New_York'))`
	`print dt.astimezone(pytz.timezone('Pacific/Honolulu'))`
	`# 结果是 2005-12-30 19:00:00-10:00`

12.4 dateutil 模块

第三方软件包 dateutil（http://labix.org/python-dateutil）提供了一些模块以按各种方法操作日期，包括时间差、重复、时区和模糊解析（参见该模块的网站以了解其丰富功能的完整文档）。dateutil 包的主要模块如表 12-8 所示。

表 12-8

easter	easter.easter(*year*)
	返回给定年 year 的复活节（Easter）的 datetime.date 对象。例如：
	`from dateutil import easter`
	`print easter.easter(2006)`
	输出 **2006-04-16**.

parser	parser.parse(*s*) 根据非常宽容（又叫做“模糊”）的解析规则，返回字符串 s 指示的 datetime.datetime 对象。例如：

续表

parser	from dateutil import parser print parser.parse("Saturday, January 28, 2006, at 11:15pm") 输出 **2006-01-28 23:15:00.**
relativedelta	relativedelta.relativedelta(...) 开发者可以对两个 datetime.datetime 实例调用 relativedelta：结果 relativedelta 实例将捕获这两个参数之间的相对差。另外，还可以对一个表示绝对信息（year、month、day、hour、minute、second、microsecond）的关键字参数调用 relativedelta；对可能是正数或负数值的相对信息（years、months、weeks、days、hours、minutes、second、microseconds）调用 relativedelta；对可以是从数字 0（星期一）到数字 6（星期日）的特殊关键字 weekday 调用 relativedelta；或者对模块属性 MO、TU…、SU 之一调用 relativedelta，还可以在调用的时候带一个数字参数 n 以指定第 n 个星期几。在任何一种情况下，结果 relativedelta 实例将捕获函数调用中的信息。例如， from dateutil import relativedelta r = relativedelta.relativedelta(weekday=relativedelta.MO(1)) r 表示“下一个星期一”。开发者可以为 datetime.datetime 实例加上一个 relativedelta 实例以获得一个新的 datetime.datetime 实例，新实例比以前增加了一个相对的值： print datetime.datetime(2006,1,29)+r # 结果是：2006-01-30 datetime.datetime(2006,1,30)+r # 结果是：2006-01-30 datetime.datetime(2006,1,31)+r # 结果是：2006-02-06 请注意，按照 relativedelta 的阐述，如果给定的日期就是星期一，则“下一个星期一”是与给定日期相同的这一天（因此，更详细的说法应该是，“在给定的日期，或者给定日期之后的第一个星期一”）。dateutil 的网站包含一些用来定义 relativedelta 实例行为规则的非常详细的解释。
rrule	rrule.rrule(*freq*,...) rrule 模块实现了 RFC2445（也被称为 iCalendar RFC），完全适用于该文档的所有内容，参见 ftp://ftp.rfc-editor.org/in-notes/rfc2445.txt。freq 必须是 rrule 模块的以下几个常数属性之一：YEARLY、MONTHLY、WEEKLY、DAILY、HOURLY、MINUTELY 或 SECONDLY。在强制参数 freq 之后，可以有选择地带有某些可能的命名参数，比如 interval=2，以指定这个重复操作只按照指定的频率交替出现（例如，rrule.rrule(rrule.YEARLY)表示每年重复一次，而 rrule.rrule(rrule.YEARLY, interval=7)则表示每 7 年只重复一次。 类型 rrule.rrule 的实例 r 提供了以下几种方法。
after	r.after(*d*, *inc*=False) 返回最早出现的符合重复规则 r，并在日期 d 之后出现的 datetime.datetime 实例（在 inc 为 True 时，也可以接受在日期 d 当天出现的实例）。

续表

before	r.before(*d*, *inc*=False) 返回最晚出现的符合重复规则 r，并在日期 d 之前出现的 datetime.datetime 实例（在 inc 为 True 时，也可以接受在日期 d 当天出现的实例）。
between	r.between(*start*, *finish*, *inc*=False) 返回所有符合重复规则 r，并在日期 start 和 finish 之间出现的 datetime.datetime 实例（在 inc 为 True 时，也可以接受在日期 start 和 finish 当天出现的实例）。

count	r.count()
	返回符合重复规则 r 的实例的出现次数。例如，要想表示“2006 年 1 月中的每个星期一次”，可以使用下面的代码：
	`start=datetime.datetime(2006,1,1)`
	`r=rrule.rrule(rrule.WEEKLY, dtstart=start)`
	`for d in r.between(start, datetime.datetime(2006,2,1), True):`
	`  print d.date(),`
	结果是 **2006-01-01 2006-01-08 2006-01-15 2006-01-22 2006-01-29**.

12.5 sched 模块

sched 模块实现了一个事件调度程序，该调度程序可以通过单线程执行来简单地处理按照“真实”或“模拟”时间尺度进行调度的事件。sched 提供了一个 scheduler 类。

表 12-9

scheduler	class scheduler(*timefunc*,*delayfunc*) scheduler 的实例 s 包含两个函数，可以用于所有与时间有关的操作。timefunc 是一个可调用函数，不带任何参数，用来获得当前的时刻（以任何度量单位）；例如，开发者可以传递 time.time 函数。delayfunc 是一个可调用函数，带有一个参数（持续时间，与 timefunc 的度量单位相同），并按参数指定的时间延迟当前的线程；例如，开发者可以传递 time.sleep 函数。在每个事件结束之后，scheduler 将调用 delayfunc(0)以向其他线程提供运行的机会；这与 time.sleep 兼容。通过使用函数作为参数，scheduler 可以使用“模拟时间”或者“伪时间”以满足应用程序的需要。 scheduler 的实例 s 提供了以下方法。
cancel	*s*.cancel(*event_token*) 从 s 的队列中删除一个事件。event_token 必须是前一个对 s.enter 或 s.enterabs 调用的结果，并且该事件必须还没有发生过；否则，cancel 将引发 RuntimeError 异常。
empty	*s*.empty() 如果 s 的队列为空，则返回 True；否则，返回 False。

续表

enterabs	*s*.enterabs(*when*,*priority*,*func*,*args*) 将一个将来的事件（回调 func(*args)函数）调度到时间 when 执行。when 是以 s 的时间函数使用的时间单位为时间单位的。如果几个事件都被调度到相同的时间执行，s 将按 priority 的增序执行这几个事件。enterabs 将返回一个事件令牌 t，开发者可以将该事件令牌传递到 s.cancel 以取消这个事件。
enter	*s*.enter(*delay*,*priority*,*func*,*args*) 与 enterabs 相似，区别在于 delay 是一个相对时间（不同于当前时刻），而 enterabs 的参数 when 是一个绝对时间（将来时刻）。
run	*s*.run() 运行所有调度的事件。s.run 将循环执行，直到 s.empty()为空，使用被传递到 s 的初始化程序的 delayfunc 以等待每个调度的事件。如果回调函数 func 引发了一个异常，s 将传播该异常，但是 s 将保持其自己的状态，并从调度队列中删除这个事件。如果回调函数 func 运行的时间比下一个调度的事件到来之前的可用时间更长，s 将延长时间，但是保持按顺序执行调度的事件，不会丢弃任何事件。如果不再对某个事件感兴趣，可以调用 s.cancel 显式丢弃这个事件。

12.6 calendar 模块

calendar 模块提供了与日历相关的函数，包括用来打印给定月份或年的文本日历的函数。在默认情况下，calendar 将把星期一作为一个星期的第一天，而星期日作为最有一天。要想改变这个设置，可以调用 calendar.setfirstweekday。calendar 可以在 time 模块的范围之内处理年，通常是 1970～2038 年。calendar 模块提供了以下函数。

表 12-10

calendar	calendar(*year*,*w*=2,*l*=1,*c*=6) 返回一个多行字符串，显示 year 年的日历，该日历被格式化为使用 c 个空格分隔的 3 列。w 是每个日期字符的宽度；每个行的长度为 21*w+18+2*c。l 是每个星期的行数。
firstweekday	firstweekday() 返回每个星期的第一天的当前星期设置。在默认情况下，当日历是第一次导入时，这个值为 0，表示星期一。
isleap	isleap(*year*) 如果 year 是闰年，则返回 True；否则，返回 False。
leapdays	leapdays(*y1*,*y2*) 返回 range(y1, y2)之间的年中闰日的总数。
month	month(*year*,*month*,*w*=2,*l*=1) 返回一个多行字符串，显示 year 年 month 月的日历，每行一个星期，再加上两个标题行。w 是每个日期的字符宽度；每个行的长度为 7*w+6。l 是每个星期的行数。

续表

monthcalendar	monthcalendar(*year*,*month*) 返回一个由 int 型整数列表组成的列表。每个子列表表示一个星期。year 年和 month 月之外的日期被设置为 0；这个月中的日期被设置为月中的天数，从 1 往上计算。
monthrange	monthrange(*year*,*month*) 返回两个整数。第一个整数是 year 年的 month 月的第一天的星期代码；第二个整数是这个月的天数。星期代码为 0（星期一）～6（星期日）；month 的值为 1～12。
prcal	prcal(*year*,*w*=2,*l*=1,*c*=6) 与 print calendar.calendar(year,w,l,c)类似。
prmonth	prmonth(*year*,*month*,*w*=2,*l*=1) 与 print calendar.month(year,month,w,l)类似。
setfirstweekday	setfirstweekday(*weekday*) 将每个星期的第一天设置为星期代码 weekday。星期代码为 0（星期一）～6（星期日）。calendar 模块提供了 MONDAY、TUESDAY、WEDNESDAY、THURSDAY、FRIDAY、SATURDAY 和 SUNDAY 属性，这些属性的值是整数 0～6。在表示星期代码的使用使用这些属性（例如，使用 calendar.FRIDAY，而不是 4）可以让代码更加清楚，也更具可读性。
timegm	timegm(*tupletime*) time.gmtime 的反转：接受一个按时间元组形式表示的时刻，并返回与从新纪元时间开始之后浮点型秒数时刻对应的相同时刻。
weekday	weekday(*year*,*month*,*day*) 返回给定日期的星期代码。星期代码是 0（星期一）～6（星期日）；月份是 1（一月）～12（十二月）。

12.7 mx.DateTime 模块

DateTime 是 eGenix GmbH 公司创建的 mx 软件包中的一个模块。mx 大部分都是开源代码，在编著本书的时候，mx.DateTime 具有类似于 Python 本身的那些自由许可证条件。mx.DateTime 的流行得益于其丰富的功能和交叉平台可移植性。本书在这一节只介绍了 mx.DateTime 的丰富功能的一个基本子集；该模块还附带了有关其高级时间和日期处理功能的详细文档。

日期和时间类型

DateTime 包提供几个日期和时间类型，其实例是不可变的（这样，适合于作为字典的键）。DateTime 类表示一个时刻，并包含一个绝对日期，也就是根据格里历（阳历）（0001-01-01 是第 1 天），从第 1 年的 1 月 1 日的新纪元时间开始之后的天数，还包含一

个绝对时间，从午夜之后的浮点型秒数。DateTimeDelta 类型表示一个时间间隔，是一个浮点型秒数。RelativeDateTime 类可以按相对描述来指定日期，比如“下一个星期一”或者“下个月的第一天”。本节详细介绍了 DateTime 和 DateTimeDelta 类（分别在第 12.7 节和第 12.7 节中介绍），但是没有详细介绍 RelativeDateTime 类。

日期和时间类型提供了自定义的字符串变换，通过内置 str 进行调用，或者在隐式转换期间自动调用（例如，在 print 语句中）。结果字符串符合标准 ISO8601 格式，比如：

```
YYYY-MM-DD HH:MM:SS.ss
```

要想对字符串格式化进行更细粒度地控制，可以使用 strftime 方法。DateTimeFrom 函数将从字符串构造 DateTime 实例。mx.DateTime 包中的各种模块都提供了其他的格式化和解析函数。

DateTime 类型

DateTime 模块提供了一些工厂函数以创建 DateTime 类型的实例，这些工厂函数按顺序提供方法、属性和算术运算符。

DateTime 的工厂函数

表 12-11

DateTime, Date, Timestamp	`DateTime(year,month=1,day=1,hour=0,minute=0,second=0.0)`
	创建并返回一个 DateTime 实例以表示给定的绝对时间。Date 和 Timestamp 是 DateTime 的同义词。day 可以小于 0，以表示从月末开始计算的日期：-1 表示这个月的最后一天，-2 表示倒数第二天，依此类推。例如：
	`print mx.DateTime.DateTime(2002,12,-1)`
	`# 打印: 2002-12-31 00:00:00.00`
	second 是一个浮点型值，可以包含一秒的任意小数。

DateTimeFrom, TimestampFrom	`DateTimeFrom(*args,**kwds)` 创建并返回一个从给定参数建立的 DateTime 实例。TimestampFrom 是 DateTimeFrom 的同义词。DateTimeFrom 可以解析表示日期和/或时间的字符串。DateTimeFrom 还可以接受命名参数，并使用与 DateTime 函数的参数具有相同名称的参数。
DateTime-FromAbsDays	`DateTimeFromAbsDays(days)` 创建并返回一个表示新纪元时间开始之后的 days 天时刻的 DateTime 实例。days 是一个浮点型数字，可以包含一天的任意小数。
DateTime-FromCOMDate	`DateTimeFromCOMDate(comdate)` 创建并返回一个表示 COM 格式的日期 comdate 的 DateTime 实例。comdate 是一个浮点型数字，并且可以包含一天的任意小数。COM 日期的新纪元时间是 1900 年 1 月 1 日的午夜。
DateFromTicks	`DateFromTicks(secs)` 创建并返回一个表示时刻 secs 的日期的本地时间午夜。secs 是一个与 time 模块表示的时刻相同的时刻（也就是，从 time 的新纪元时间开始之后的秒数）。

续表

gmt, utc	gmt() 创建并返回一个表示当前 GMT 时间的 DateTime 实例。utc 是 gmt 的同义词。
gmtime, utctime	gmtime(*secs*=None) 创建并返回一个表示时刻 secs 的 GMT 时间的 DateTime 实例。secs 是一个像 time 模块表示的时刻那样的时刻（也就是，从 time 的新纪元时间开始之后的秒数）。在 secs 为 None 时，gmtime 将使用当前的时刻，就像 time.time 函数返回的时刻那样。utctime 是 gmtime 的同义词。
localtime	localtime(*secs*=None) 创建并返回一个表示时刻 secs 的本地时间的 DateTime 实例。secs 是一个像 time 模块表示的时刻那样的时刻（也就是，从 time 的新纪元时间开始之后的秒数）。在 secs 为 None 时，localtime 将使用当前时刻，就像 time.time 函数返回的时刻一样。
mktime	mktime(*timetuple*) 创建并返回一个使用 9 个项目的元组 timetuple 指定的时刻表示的 DateTime 实例，timetuple 元组遵循 time 模块使用的格式。
now	now() 创建并返回一个表示当前本地时间的 DateTime 实例。
Timestamp-FromTicks	TimestampFromTicks(*secs*) 创建并返回一个表示时刻 secs 的本地时间的 DateTime 实例。secs 是一个像 time 模块表示的时刻那样的时刻（从 time 的新纪元时间开始之后的秒数）。
today	today(*hour*=0,*minute*=0,*second*=0.0) 创建并返回一个表示今天（日期）的给定时间对应的本地时间的 DateTime 实例（默认值为午夜）。

DateTime 实例的方法

DateTime 实例 d 最常使用的方法如表 12-12 所示。

表 12-12

absvalues	*d*.absvalues() 返回一个数据对（ad, at），其中 ad 是一个整数，表示 d 的绝对日期，at 是一个浮点型数字，表示 d 的绝对时间。
COMDate	*d*.COMDate() 按 COM 格式（一个浮点型数字，从 1900 年 1 月 1 日午夜开始到现在的天数和天数的小数部分）返回 d 的时刻。
gmticks	*d*.gmticks() 返回一个浮点型值，按秒（和秒的小数部分）的形式表示从 time 模块的新纪元时间开始之后的时刻，假定 d 是按 GMT 来表示的。

续表

gmtime	*d*.gmtime() 按 GMT 返回一个表示 d 的时刻的 DateTime 实例 d1，假定 d 是按本地时间表示的。
gmtoffset	*d*.gmtoffset() 返回一个表示 d 的时区的 DateTimeDelta 实例，假定 d 是按本地时间表示的。如果是在美洲，gmtoffset 将返回负值，而在欧洲、亚洲和非洲的大多数地区，将返回正值。
localtime	*d*.localtime() 按本地时间返回一个表示 d 的时刻的 DateTime 实例 d1，假定 d 是按 GMT 表示的。
strftime, Format	*d*.strftime(*fmt*="%c") 返回一个按字符串 fmt 指定的格式表示 d 的字符串。fmt 的语法与 time.strftime 的语法相同。Format 是 strftime 的同义词。
ticks	*d*.ticks() 返回一个浮点型数字，以从 time 模块的新纪元时间开始之后的秒数（包括小数部分）表示 d 的时刻，假定 d 是按本地时间表示的。
tuple	*d*.tuple() 以 time 模块使用的格式中包含 9 个项目元组的形式返回 d 的实例。

DateTime 实例的属性

DateTime 实例 d 最常使用的属性如下（所有属性都是只读属性）：

absdate

d 的绝对日期；与 d.absvalues( )[0]一样；

absdays

一个浮点型数字，表示从新纪元时间开始之后的天数（包含一天的小数部分）；

abstime

d 的绝对时间；与 d.absvalues( )[1]一样；

date

格式为‘YYYY-MM-DD’的字符串；d 的日期的标准 ISO 格式；

day

1～31 的整数；d 的月份中的天数；

day_of_week

0～6 的整数；d 的星期中的星期序数（星期一为 0）；

day_of_year

1～366 的整数；d 的年中的天数（1 月 1 日为 1）；

dst

-1～1 的整数，表示在日期 d 这一天夏令时是否有效，假定 d 是按本地时间表示的（-1 表示未知、0 表示无效，而 1 表示有效）；

hour

0～23 的整数；d 的日期的小时数；

iso_week

d 的一个包含 3 个项目的元组（year, week, day），并带有 ISO 星期符号（week 是一年的星期序数；day 在 1（星期一）和 7（星期日）之间；

minute

0～59 的整数；d 的小时的分钟数；

month

1～12 的整数；d 的年的月份；

second

0.0～60.0 的浮点型数字；d 的分钟中的秒数（DateTime 实例不支持闰秒）；

year

一个整数；d 的年份（1 表示 1CE，0 表示 1BCE）。

DateTime 实例的算术运算

开发者可以在两个 DateTime 实例 d1 和 d2 之间使用二元运算符 -（减）。在这种情况下，d1-d2 是一个表示 d1 到 d2 之间经过的时间的 DateTimeDelta 实例，如果 d1 的时间比 d2 的时间晚，则实例 d1-d2 大于 0。开发者还可以在 DateTime 实例 d 和数字 n 之间使用二元运算符+和-。d+n、d-n 和 n+d 都是 DateTime 实例，与 d 相差 n（或者-n）天（如果 n 是一个浮点型数字，则还包含天数的小数部分），而 n-d 被武断地定义为等于 d-n。

DateTimeDelta 类型

DateTimeDelta 类型的实例表示时刻之间的时间差。内在地讲，DateTimeDelta 实例保存了一个表示秒数（和秒数的小数部分）的浮点型数字。

DateTimeDelta 的工厂函数

DateTime 模块提供了许多可以创建 DateTimeDelta 实例的工厂函数。某些工厂函数可

以通过一个或多个同义词进行调用。最常使用的同义词如表 12-13 所示。

表 12-13

DateTimeDelta	DateTimeDelta(*days*,*hours*=0.0,*minutes*=0.0,*seconds*=0.0) 按照下面的公式创建并返回一个 DateTimeDelta 实例： *seconds+60.0*(minutes+60.0*(hours+24.0*days))*
DateTimeDelta-From	DateTimeDeltaFrom(**args*,***kwds*) 从给定的参数创建并返回一个 DateTimeDelta 实例。参见第 12.7 节中介绍的 DateTimeFrom 和 TimestampFrom 函数。
DateTimeDelta-FromSeconds	DateTimeDeltaFromSeconds(*seconds*) 与 DateTimeDelta(0,0,0,seconds)一样。
TimeDelta, Time	TimeDelta(*hours*=0.0,*minutes*=0.0,*seconds*=0.0) 与 DateTimeDelta(0,hours,minutes,seconds)一样。TimeDelta 函数可以保证接受命名参数。Time 是 TimeDelta 的同义词。
TimeDeltaFrom, TimeFrom	TimeDeltaFrom(**args*,***kwds*) 与 DateTimeDeltaFrom 类似，区别在于，如果有位置参数，则该参数指示小时，而不是像 DateTimeDeltaFrom 那样指示天。TimeFrom 与 TimeDeltaFrom 相同。
TimeFromTicks	TimeFromTicks(*secs*) 为时刻 secs（按从新纪元时间开始之后的秒数）和时刻 secs 的相同天的午夜之间的时间差创建并返回一个 DateTimeDelta 实例。

DateTimeDelta 实例的方法

DateTimeDelta 实例 d 最常使用的方法如表 12-14 所示。

表 12-14

absvalues	*d*.absvalues() 返回一个数据对（ad, at），其中 ad 是一个整数（d 的天数），at 是一个浮点型数字（d 的秒数对 86400 取模），并且都带有相同的符号。

strftime, Format	*d*.strftime(*fmt*="%c")
	返回一个按字符串 fmt 指定的格式表示 d 的字符串。fmt 的语法与 time.strftime 的语法相同，参见第 12.1 节中介绍的 strftime 函数，但不是所有指定符都是有意义的。d.strftime 的结果并不反映 d 表示的时间间隔的符号；要想显示这个符号，可以通过字符串操作将符号放在字符串之前。
	`if d.seconds >= 0.0: return d.strftime(fmt)`
	`else: return '-' + d.strftime(fmt)`
	Format 是 strftime 的同义词。

tuple	*d*.tuple() 返回一个元组（day, hour, minute, second）；每个项目都是在各自范围内的一个带符号数字。second 是一个浮点型数字，而其他项目都是整数。

DateTimeDelta 实例的属性

DateTimeDelta 实例 d 提供了以下属性（所有属性都是只读的）：

day

hour

minute

second

与 d.tuple( )返回的元组中的 4 个项目一样；

days

hours

minutes

seconds

每个属性都是一个浮点型值，按给定的度量单位为 d 提供值，这样：

```
d.seconds == 60.0*d.minutes ==3600.0*d.hours == 86400.0*d.days
```

DateTimeDelta 实例的算术运算

开发者可以将两个 DateTimeDelta 实例 d1 和 d2 相加或相减，这样可以加上或者减去这些实例表示的带符号时间间隔。开发者可以在 DateTimeDelta 实例 d 和数字 n 之间使用二元运算符+和−；n 被看作是秒数（如果 n 是一个浮点型值，则还包括秒数的小数部分）。开发者还可以将实例 d 乘以 n 或除以 n 以按比例放大或缩小 d 表示的时间间隔。每个运算都将返回一个 DateTimeDelta 实例。开发者还可以从 DateTime 实例 d 加上或减去一个 DateTimeDelta 实例 dd 以得到另一个 DateTime 实例 d1，d1 与 d 之间相差 dd 所表示的带符号时间间隔。

其他属性

mx.DateTime 模块还提供了许多常数属性。最常使用的属性是：

oneWeek

oneDay

oneHour

oneMinute

oneSecond

DateTimeDelta 的实例，表示指定的持续时间；

Monday

Tuesday

Wednesday

Thursday

Friday

Saturday

Sunday

表示星期几的整数：星期一为 0、星期二为 1，依此类推；

Weekday

一个字典，将整数的星期几映射到其字符串名称，或者相反：0 映射为‘Monday’、‘Monday’映射为 0，依此类推；

January

February

March

April

May

June

July

August

September

October

November

December

表示月份的整数：一月为 1、二月为 2，依此类推；

Month

一个字典，将整数的月份映射到其字符串名称，或者相反：1 映射为‘January’、‘January’映射为 1，依此类推。

mx.DateTime 模块提供了另一个有用的函数。

表 12-15

cmp	cmp(*obj1,obj2,accuracy*=0.0) 比较两个 DateTime 或 DateTimeDelta 实例 obj1 和 obj2，并返回-1、0 或 1，就像内置函数 cmp 一样。如果两个时刻或持续时间的差小于 accuracy 秒，则该函数返回 0（表示 obj1 和 obj2 是“相等的”）。

子模块

mx.DateTime 包提供了几个用于特殊目的的子模块。mx.DateTime.ISO 模块提供了一些函数以解析和生成符合 ISO8601 格式的日期和时间字符串。mx.DateTime.ARPA 模块提供了一些函数以解析和生成符合 ARPA 格式的日期和时间字符串，这种格式广泛用于 Internet：

```
[Day, ]DD Mon YYYY HH:MM[:SS] [ZONE]
```

mx.DateTime.Feasts 模块提供了一些函数以计算任何给定年份的复活节（Easter Sunday）日期，以及基于复活节的其他可移动的节日。如果开发者的计算机连接到 Internet，可以使用 mx.DateTime.NIST 模块访问 NIST 原子钟（Atomic clock）提供的精确的世界标准时间。感谢 NIST 的原子钟，这个模块可以非常精确地计算当前日期和时间。该模块可以通过参考 NIST 的时钟，并补偿在访问 NIST 时导致的任何网络延时以校正计算机的近似时钟。

第 13 章

控制执行

Python 提供了（换句话讲，直接显示和文档支持）其使用的许多内部机制的文档。这些文档上的支持可以帮助开发者从一个更高的级别来理解 Python，并帮助开发者将自己的代码与这些文档介绍的 Python 机制挂钩，并从某种程度对这些机制进行控制。例如，第 7.1 节在介绍 Python 内置对象时说明了 Python 处理内置对象的方式是隐式可见的。本章将介绍其他一些高级 Python 控制技术，而第 18 章将介绍一些可以特别应用于测试、调试和最优化的技术。

13.1 动态执行和 exec 语句

Python 的 exec 语句可以执行开发者读取、生成，或者是在程序运行期间获得的代码。exec 可以动态执行一条语句或一组语句。exec 是一个具有以下语法的简单关键字语句：

```
exec code[ in globals [,locals]]
```

code 可以是一个字符串、一个打开的类文件对象或者一个代码对象。globals 和 locals 都是字典（在 Python 2.4 中，locals 可以是任意映射，但是 globals 必须确定是一个字典，在 Python 2.5 中，两者都可以是任意映射）。如果同时提供了这两个参数，则这两个参数分别表示 code 执行中的全局（global）或本地（local）命名空间。如果只提供了 globals，exec 将使用 globals 作为两种命名空间。如果既没有提供 globals，又没有提供 locals，code 将在当前作用域下执行。在当前作用域下运行 exec 是一个不好的主意，因为这样 code 可以绑定、重新绑定或解除绑定任意名称。要想让事情处在控制之下，只能对特定的和显式的字典使用 exec。

避免使用 exec

只有在 exec 真的必不可少时才能使用 exec。最常见的是，最好避免使用 exec，而是选

择更特殊和更好控制的机制：exec 会放宽开发者对代码的命名空间的控制，这样会损害程序的性能，并导致开发者遇到大量难以发现的错误（bug）。

例如，Python 的一个经常被问到的问题就是“我该如何设置一个其名称是我刚才读取或建立的变量呢？”严格来说，exec 可以帮助开发者实现这个操作。例如，如果开发者想要设置的变量的名称在 varname 中，可以使用：

```
exec varname+'=23'
```

但是，不要这样做！在当前的作用域下，像这样的 exec 语句会使得开发者失去对命名空间的控制，导致极其难以跟踪的错误，并且，通常会使得开发者的程序非常难以理解。一种改进的方案就是不要将开发者需要设置的“变量”作为变量，而是作为字典中的条目，也就是 mydict。然后，开发者就可以使用下面的变体了：

```
exec varname+'=23' in mydict
```

尽管这种用法不像前一个示例那样可怕，但仍然是一个很坏的做法。将这样的“变量”作为字典中的条目是简单而有效的，但是也意味着开发者完全不需要使用 exec 来设置这些变量。只需要使用下面的代码即可：

```
mydict[varname] = 23
```

使用这种方法，程序会变得更清楚、直接、优美，也更快。尽管也可以有效地使用 exec，但是 exec 的使用是极其稀少的，而是应该总是使用显式字典。

表达式

exec 可以执行一个表达式，因为任何表达式也是一个有效语句（被称为表达式语句）。但是，在这种情况下，Python 忽略了一个表达式语句返回的值。要想计算一个表达式并获得该表达式的值，可以使用内置函数 eval，参见第 8.2 节中介绍的 eval 函数。

编译和代码对象

要想获得一个可以与 exec 一起使用的代码对象，通常可以调用内置函数 compile，并将其最后一个参数设置为‘exec’（参见第 8.2 节中介绍的 compile 函数）。代码对象 c 提供了许多有趣的只读属性，这些属性的名称都是以‘co_’开始的，比如：

co_argcount

c 为代码的函数中的参数数量（在 c 不是一个函数的代码对象，而是由 compile 对一个字符串直接编译而来时，这个值为 0）；

co_code

由 c 的字节代码构成的字节字符串；

co_consts

c 中使用的常数组成的元组；

co_filename

文件 c 是从这个名称的文件编译而来的（在 c 是使用 compile 函数编译而来时，这个字符串是 compile 函数的第二个参数）；

co_firstlinenumber

源代码的初始行号（在 co_filename 指定的文件中），如果 c 是从一个文件编译而来，该源代码将被编译以生成 c；

co_name

c 是其中的代码的函数的名称（在 c 不是一个函数的代码对象，而是由 compile 对一个字符串直接编译而来时，这个名称为‘<module>’）；

co_names

c 中使用的所有标识符组成的元组；

co_varnames

c 中的本地变量的标识符组成的元组，从参数名开始。

这些属性中的大多数都只是用于调试目的，但是某些属性可能会有助于高级自测功能，本节后面会举例说明。

如果要运行一个包含多个语句的字符串，本书建议对这个字符串使用 compile，然后对结果代码对象调用 exec，而不是将这个字符串交给 exec 来编译和执行。这种分开操作的方法可以帮助开发者分别检查语法错误和赋值时的错误。通常还可以对操作进行安排，使得字符串只被编译一次，而代码对象可以被重复执行，这样可以加速操作的执行。eval 也可以从这样的分离中得到好处。此外，compile 步骤本质上是很安全的（而 exec 和 eval 都是非常有风险的，如果开发者对不被信任的代码执行这两个操作），并且开发者还可以对代码对象执行一些检查以降低风险。

代码对象具有一个只读属性 co_names，这个属性是由代码中使用的名称组成的元组。例如，假定开发者想要用户输入一个只包含字面常量常数和操作符的表达式——既不包含函数调用，也不包含其他名称。在计算这个表达式之前，开发者可以检查用户输入的字符串是否满足这些限制条件：

```
def safe_eval(s):
    code = compile(s, '<user-entered string>', 'eval')
    if code.co_names:
        raise ValueError, ('No names (%r) allowed in expression (%r)' %
```

```
                                    (code.co_names, s))
    return eval(code)
```

这个函数 safe_eval 只有在表达式字符串是一个语法正确的表达式（否则，compile 将引发 SyntaxError 异常），并且不包含任何名称（否则，safe_eval 将显式引发 ValueError 异常）时，才会计算作为参数 s 传递的表达式。

了解代码打算要访问的名称有时候可以帮助开发者优化字典的准备工作，开发者需要将这个字典作为命名空间传递到 exec 或 eval。由于开发者只需要为这些名称提供值，因此不需要准备其他条目，从而节省工作时间。例如，假定开发者的应用程序可以动态接受来自用户的代码，按常规，这些以 data_开始的变量名称是指保存在子目录 data 中的文件，并且用户编写的代码不需要显式读取这些文件。用户编写的代码将按顺序进行执行这些代码，并将结果保留在其名称以 result_开始的全局变量中，开发者的应用程序将把这些结果以文件的形式写回到子目录 data 中。感谢这种常规方法，开发者可以在以后将数据移动到任何位置（例如，移动到数据库中的 BLOB，而不是子目录中的文件），并且用户编写的代码不会受到任何影响。下面是一个如何有效实现这些常规方法的例子：

```
def exec_with_data(user_code_string):
    user_code = compile(user_code_string, '<user code>', 'exec')
    datadict = { }
    for name in user_code.co_names:
        if name.startswith('data_'):
            datafile = open('data/%s' % name[5:], 'rb')
            datadict[name] = datafile.read()
            datafile.close()
exec user_code in datadict
for name in datadict:
    if name.startswith('result_'):
        datafile = open('data/%s' % name[7:], 'wb')
        datafile.write(datadict[name])
        datafile.close()
```

永远不要 exec 和 eval 不被信任的代码

请注意，对于不被信任的代码，exec_with_data 函数也是完全不安全的：如果开发者将 user_code_string 作为参数传递到该函数，而 user_code_string 是一个以不能完全被信任的方式获得的字符串，从本质上讲，这样做可能会造成的损害程度是没有限制的。很不幸，在只是使用 exec 和 eval 的情况下，真的如此，除非在极少的情况下，开发者对将要执行或计算的代码设置了非常严格和可检查的限制，就像在使用 safe_eval 函数的情况一样。

老版本的 Python 试图提供一些工具以改善这种情况，也就是所谓的“受限执行”，但是这些工具从来无法完全防范那些有才干的电脑黑客的天才攻击，因此 Python 的当前版本将会放弃这些工具。如果开发者需要防卫这样的攻击，可以利用操作系统的保护机制：在一个单独的进程中运行不被信任的代码，尽可能严格地设置代码的权限（请学习操作系统为这个目的提供的一些机制，比如 chroot、setuid 和 jail）。要想防卫“拒绝服务”式攻击，可以让主进程监视单独的进程，如果出现了过多的资源消耗，则终止这个单独的进程。参见第 14.7 节中对进程的介绍。

13.2 内部类型

本节提到的某些内部 Python 对象是很难使用的。正确使用这样的对象并得到最佳的效果需要学习 Python 本身的 C（或者 Java，或者 C#）源代码。这样的要求是很少需要的，除非要建立通用目的开发框架和类似的向导任务。一旦开发者深入理解了这些内容，Python 可以帮助开发者在需要的时候对内部对象进行尽可能地控制。因为 Python 将许多类型的内部对象开放到开发者的 Python 代码中，因此开发者可以通过使用 Python 进行编码以实现对内部对象的控制，即使是一个对 C（或者 Java，或者 C#）略知一二的人也需要阅读 Python 的源代码，以理解这些源代码做了些什么。

类型对象

名为 type 的内置类型可以作为一个工厂对象，并返回也是类型的对象。除了相等比较和以字符串表示，类型对象不需要支持任何特殊操作。但是，大多数类型对象都是可调用的，并在调用时返回该类型的新实例。特别是，像 int、float、list、str、tuple、set 和 dict 这样的内置类型都是按这种方式操作的。type 模块的属性也都是内置类型，每个类型具有一个或多个名称。例如，type.DictType 和 types.DictionaryType 指的都是 type({})，也被称为 dict。除了可以被调用以生成实例之外，许多类型对象都是很有用的，因为开发者可以创建这些类型对象的子类，参见第 5.1 节。

代码对象类型

与使用内置函数 compile 一样，开发者还可以通过函数或方法对象的 func_code 属性得到代码对象（要想了解代码对象的属性，参见第 13.1 节）。代码对象是不可以调用的，但是可以将一个函数对象的 func_code 属性与正确数量的参数重新绑定，这样就可以将代码对象包装成可调用的形式了。例如：

```
def g(x): print 'g', x
code_object = g.func_code
def f(x): pass
```

```
f.func_code = code_object
f(23)     # 结果是:  g 23
```

不带任何参数的代码对象也可以与 exec 语句或内置函数 eval 一起使用。new 模块提供了一个函数来创建代码对象，还提供了其他一些函数以创建老式实例和类、函数、方法和模块。开发者还可以通过调用想要实例化的类型对象以创建一个新对象，而不是从 new 模块调用一个工厂函数。

frame 类型

sys 模块中的_getframe 函数可以从 Python 的调用堆栈中返回一个框架（frame）对象。框架对象具有一些属性，这些属性可以提供有关在该框架中执行的代码以及执行状态的信息。traceback 和 inspect 模块可以用来访问和显示信息，特别是在正在处理一个异常的时候。第 18 章提供了更多有关框架和跟踪的信息，还介绍了 inspect 模块，该模块通常是执行这样的自测操作的最佳方法。

13.3 垃圾收集

Python 的垃圾收集功能通常是透明和自动执行的，但是，开发者也可以尽力选择实行某些直接控制。一般的原则是，Python 将在每个对象 x 变得不可到达之后的某个时刻收集对象 x，也就是说，在没有引用链可以通过正在执行的函数实例的本地变量，也不能从一个加载的模块的全局变量访问 x 时。

通常，对象 x 在没有任何对 x 的引用时变得不可到达。此外，一组对象在他们之间相互引用，但是没有全局和本地变量引用这些对象时是不可到达的（这样的情况被称为“相互引用循环”）。

经典 Python 为每个对象 x 保存了一个计数，也被称为引用计数，这个计数就是对 x 的引用次数。在 x 的引用计数下降到 0 时，CPython 将立即收集 x。sys 模块的 getrefcount 函数可以接受任何对象作为参数，并返回其引用计数（至少为 1，因为 getrefcount 本身有一个对其检查的对象的引用）。其他版本的 Python，比如 Jython 或 IronPython，依赖于其运行的平台提供的其他垃圾收集机制（例如，JVM 或 MSCLR）。因此，gc 和 weakref 模块只能应用于 CPython。

在 Python 垃圾收集对象 x，并且没有任何对 x 的引用时，Python 将终止化 x（也就是说，调用 x._ _del_ _()）并使得 x 占用的内存可供其他对象使用。如果 x 保留了任何对其他对象的引用，Python 将删除这些引用，这样，通过让其他对象不可到达，可以按顺序让其他对象可收集。

gc 模块

gc 模块提供了 Python 的垃圾收集器功能。gc 只能处理以简单方法不能到达的对象，

这种方法是相互引用循环的一部分。在这样的循环中，循环中的每个对象都引用其他对象，并保持所有对象的引用计数为正数。但是，外部不再存在任何一个对相互引用的对象集合中的任意一个对象的引用。因此，整个组（也被称为循环垃圾）都是不可到达的，因此可以对其进行垃圾收集。查找这样的循环垃圾的循环需要花费一些时间，这就是 gc 模块存在的原因。在默认情况下，这种“循环垃圾收集”功能是被启用的，并带有一些合理的默认参数：但是，通过导入 gc 模块并调用其函数，开发者可以选择禁用这个功能，更改其参数，或者找到在这种情况下正确的做法。

gc 提供了一些可供使用的函数，可以对循环垃圾收集的时间进行控制。这些函数有时候可以用来向下跟踪内存泄漏——没有被收集的对象，即使没有任何对这些对象的引用，这是通过查找哪些其他对象实际上保持对这些对象的引用来实现的。

表 13-1

collect	`collect()` 强制立即运行完整的循环垃圾收集过程。
disable	`disable()` 暂停自动循环垃圾收集。
enable	`enable()` 重新使能以前使用 disable 暂停的自动循环垃圾收集。
garbage	一个只读属性，列出了不可到达，但是也不可收集的对象。如果循环垃圾的循环中的任何一个对象具有__del__特殊方法，这种情况将会发生，因为没有确定的安全顺序让 Python 终止化这样的对象。
get_debug	`get_debug()` 返回一个整型比特字符串，使用 set_debug 设置垃圾收集调试标记。
get_objects	`get_objects()` 返回一个由当前被循环垃圾收集器跟踪的所有对象组成的列表。
get_referrers	`get_referrers(*objs)` 返回一个由当前被循环垃圾收集器跟踪的所有容器对象组成的列表，这些容器对象引用了任何一个或多个参数。
get_threshold	`get_threshold()` 返回一个包含 3 个元素的元组（thresh0, thresh1, thresh2），使用 set_threshold 设置垃圾收集阈值。
isenabled	`isenabled()` 如果循环垃圾收集当前是启用的，则返回 True。在循环垃圾收集当前被禁用时，isenabled 将返回 False。

续表

set_debug	set_debug(*flags*) 为循环垃圾收集设置调试标记。flags 是通过对 gc 模块提供的零个或多个常数执行“或”操作（使用按位或运算符	）而生成的一个整型比特字符串： DEBUG_COLLECTABLE　打印收集期间找到的可收集对象的信息； DEBUG_INSTANCES　如果同时还设置了 DEBUG_COLLECTABLE 或 DEBUG_UNCOLLECTABLE 标记，则打印收集老式类的实例期间找到的对象的信息； DEBUG_LEAK　可以让循环垃圾收集器打印可以帮助开发者诊断内存泄漏的所有信息的调试标记的集合；相当于按位或所有其他常数（除了 DEBUG_STATS，这个标记用于其他目的）； DEBUG_OBJECTS 如果同时还设置了 DEBUG_COLLECTABLE 或 DEBUG_UNCOLLECTABLE，则打印收集不是老式类的实例期间找到的对象的信息； DEBUG_SAVEALL 将所有可收集对象保存到列表 gc.garbage（不可收集对象也总是被保存在这里）中，以帮助开发者诊断泄漏； DEBUG_STATS 打印收集期间的统计信息以帮助开发者调整阈值； DEBUG_UNCOLLECTABLE 打印收集期间找到的不可收集对象的信息。
set_threshold	set_threshold(*thresh0*[,*thresh1*[,*thresh2*]]) 设置阈值，该阈值用来控制循环垃圾收集以多长时间运行一次。thresh0 为 0 时禁用垃圾收集。垃圾收集是一个高级主题，Python 历来的各个版本中使用的垃圾收集方案的详细信息，以及由此导致阈值具有不同的详细含义都超出了本书要介绍的范围。

在开发者知道自己的程序中并没有与循环垃圾相关的循环操作，或者在一些紧要情况下，不能忍受循环垃圾收集的延时时，可以调用 gc.disable( ) 暂停自动垃圾收集。以后还可以调用 gc.enable( ) 再次启用垃圾收集操作。开发者可以调用 gc.isenable( ) 测试是否启用了自动垃圾收集，该函数将返回 True 或 False。要想控制什么时候需要花时间进行垃圾收集，可以调用 gc.collect( ) 以强制立即执行一个完整的垃圾收集操作。下面是一个包含了时间相关代码的习惯用法。

```
import gc
gc_was_enabled = gc.isenabled()
if gc_was_enabled:
    gc.collect()
    gc.disable()
# 在此插入一些时间相关代码
```

```
if gc_was_enabled:
    gc.enable()
```

gc 模块中的其他功能更加高级，也极少使用，这些功能可以被分成两个区域。get_threshold 函数和 set_threshold 函数，以及调试标记 DEBUG_STATS 可以用来调整垃圾收集的设置以优化程序的性能。gc 的其他功能可以用来诊断程序中的内存泄漏。尽管 gc 模块本身可以自动修复许多内存泄漏（只要开发者没有在类中定义__del__，因为__del__的存在可以阻碍循环垃圾收集），但是，如果开发者从一开始就能避免创造循环垃圾，则程序会运行得更快。

weakref 模块

仔细的设计通常可以避免引用的循环。但是，在需要两个对象相互知道对方时，避免相互引用会造成对设计的曲解，并把设计弄得很复杂。例如，一个容器包含对其项目的引用，然而，对一个项目而言，知道哪个容器包含自己通常也是非常有用的。其结果就是引用循环：由于相互引用，容器和项目相互让对方活着，即使所有其他对象都已经忘记了这两个对象的存在。弱引用可以通过让对象之间相互引用对方，但是不相互让对方活着来解决这个问题。

“弱引用”（week reference）是一个特殊对象 w，该对象引用其他对象 x，但是不增加 x 的引用计数。在 x 的引用计数下降到 0 时，Python 将终止化和收集 x，然后向 w 通知 x 的终结。然后，弱引用 w 可以消失，或者在可控的方式下被标记为无效。在任何时候，一个给定的弱引用 w 引用可以在 w 被创建时引用相同的目标对象 x，或者根本不引用任何对象；弱引用永远不会改变目标对象。并不是所有类型的对象都支持成为弱引用 w 的目标对象 x 的，但是类实例和函数是可以的。

weekref 模块提供了一些函数和类型以创建和管理弱引用。

表 13-2

getweakrefcount	`getweakrefcount(x)` 返回 len(getweakrefs(x))。
getweakrefs	`getweakrefs(x)` 返回一个由所有目标对象为 x 的弱引用组成的列表。
proxy	`proxy(x[,f])` 返回一个类型为 ProxyType（如果 x 是可调用的，则为 CallableProxyType 类型），对象 x 作为目标对象的弱代理 p。在大多数上下文环境中，使用 p 就像使用 x 一样，区别只是，如果在 x 已经被删除之后使用 p，Python 将引发 ReferenceError 错误。p 不是可哈希的（因此，不能使用 p 作为字典的键），即使在 x 是可哈希的情况下。如果给出了 f，f 必须是可调用的，并带有一个参数，并且 f 是 p 的终止化回调函数（也就是，在终止化 x 之前，Python 将调用 f(p)）。在 f 被调用时，x 不再能够从 p 到达。

续表

ref	`ref(x[,f])` 返回一个类型为 ReferenceType，对象 x 为目标对象的弱引用 w。w 是可调用的：如果 x 仍然是活着的，调用 w( )将返回 x；否则，调用 w( )将返回 None。如果 x 是可哈希的，w 也是可哈希的。开发者可以比较弱引用是否相等（使用==或!=），但是不能进行排序（使用<、>、<=或>=）。如果两个弱引用 x 和 y 的目标都是活着的，并且相等，或者 x is y，则这两个弱引用相等。如果给出了 f，f 必须是可调用的，并带有一个参数，并且 f 是 w 的终止化回调函数（也就是，在终止化 x 之前，Python 将调用 f(w)）。在 f 被调用时，x 不再能够从 w 到达。
WeakKeyDictionary	`class WeakKeyDictionary(adict={ })` WeakKeyDictionary 的实例 d 是一个映射，可以弱引用其键。在 d 中的一个键 k 的引用计数下降到 0 时，项目 d[k]将会消失。adict 用来初始化这个映射。
WeakValueDictionary	`class WeakValueDictionary(adict={ })` WeakValueDictionary 的实例 d 是一个映射，可以弱引用其值。在 d 中的一个值 v 的引用计数下降到 0 时，d 中 d[k]的值为 v 的所有项目都会消失。adict 用来初始化这个映射。

WeakKeyDictionary 可以用来非入侵地关联附加数据和某些可哈希对象，而不会更改这些对象。WeakValueDictionary 可以用来非入侵地记录对象和编译缓存之间的瞬时关联。在每种情况下，使用弱映射要比普通字典更好一些，弱映射可以确保一个可垃圾收集的对象不会像在映射中使用的那样一直保持活着。

弱引用的一个典型示例就是一个类将跟踪其实例，但是不会让这些实例保持活着，而只是为了跟踪这些实例：

```
import weakref
class Tracking(object):
    _instances_dict = weakref.WeakValueDictionary()
    def __init__(self):
        Tracking._instances_dict[id(self)] = self
    def instances(): return _instances_dict.values()
    instances = staticmethod(instances)
```

13.4 终止函数

atexit 模块可以注册终止函数（也就是，在程序终止时调用的函数："后进先出"）。终止函数类似于使用 try/finally 建立的清理处理程序。但是，终止函数是全局注册的，并在整个程序的末尾被调用，而清理处理程序是从词法上建立的，可以在特定的 try 子句的末尾调用。不管程序正常还是非正常终止，都可以调用终止函数和清理处理程序，

但是，不能在程序是通过调用 os._exit 而结束的特殊情况下调用终止函数或清理处理程序。atexit 模块提供了一个单独的被称为 register 的函数。

表 13-3

register	register(*func*,**args*,***kwds*) 确保 func (*args,**kwds) 在程序终止的时候被调用。

13.5 站点和用户自定义

Python 提供了一个特定的“钩子”以让每个站点（site）在每次开始运行 Python 时自定义 Python 在某些方面行为。在默认情况下，Python 没有启用每个单独用户的自定义功能，但是，对于想要在启动时运行用户提供的代码的程序，Python 指定了该程序如何显式请求这样的自定义功能。

site 和 sitecustomize 模块

Python 将在运行主脚本之前加载标准模块 site。如果在运行 Python 时使用了选项-S，Python 将不加载 site。-S 允许更快的启动，但是为主脚本增加了一些初始化事务。site 的任务如下。

- 将 sys.path 设定为标准格式（绝对路径，不重复）。
- 解释 Python 主目录中找到的每个.pth 文件，将条目添加到 sys.path 中，并按照每个.pth 文件的指示导入模块。
- 在交互式会话中添加用于显示信息（quit、exit、copyright、credits 和 license）的内置模块。
- 将默认 Unicode 编码方式设置为‘ascii’。站点的源代码包括两个代码块，每个代码块都是使用 if 0:来守卫的，一个用来将默认的编码方式设置为本地语言环境相关，另一个用来完全禁用 Unicode 和普通字符串之间的任何默认编码。开发者可以有选择地编辑 site.py 来选择任意一个代码块。
- 尝试导入 sitecustomize（使用 import sitecustomize 将引发一个 ImportError 异常，site 将捕获并忽略这个异常）。sitecustomize 是这样一个模块，每个站点的安装都可以有选择地用于超出站点任务的进一步的站点专用自定义功能。通常最好不要编辑 site.py，因为 Python 的任何一次升级或重新安装都会改写这些自定义设置。sitecustomize 的主要任务可以是为站点设置正确的默认编码方式。例如，对于一个西欧站点，可以选择调用 sitecustomize sys.setdefaultencoding('iso-8859-1')。
- 在完成了 sitecustomize 之后，从 sys 模块中删除 sys.setdefaultencoding 属性。

这样，Python 的默认 Unicode 编码方式可以只在开始运行 Python 的时候被设置，而不

能在运行过程中被更改。在紧急情况下，如果特定的主脚本万不得已需要打破这个原则，并设置一个不同于所有其他脚本使用的默认编码方式，可以将下面的代码片段放在主脚本的开始：

```
import sys                                      # 导入 sys 模块对象
reload(sys)                                     # 从硬盘还原 sys 模块
sys.setdefaultencoding('iso-8859-15')           # 或者是需要的其他编解码器
del sys.setdefaultencoding                      # 确保以后不出现意外
```

但是，这并不是一种好的风格。开发者应该重新修改脚本，这样，脚本可以接受站点已经选择的任何默认编码方式，并在需要特定编解码器的所有位置显式传递这个编码方式的名称。

用户自定义

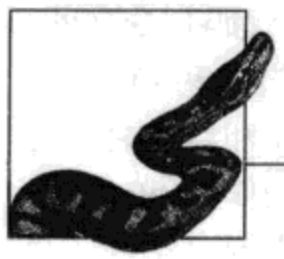
每个交互式 Python 解释器会话都会运行环境变量 PYTHONSTARTUP 指定的脚本。在交互式解释器会话之外，不再有任何自动地针对每个用户的自定义。要想请求针对每个用户的自定义，Python 主脚本可以显式使用 import user。在标准库模块 user 被加载时，将首先确定由环境变量 HOME（或者，如果访问环境变量 HOME 失败，可以访问 HOMEPATH，在 Windows 系统上，可能会在 HOMEPATH 的前面加上 HOMEDRIVE）指定的用户主目录。如果该环境变量没有指定一个主目录，则 user 模块将使用当前目录。如果 user 模块在指定的目录找到了一个名为.pythonrc.py 的文件，user 模块将在该模块自己的全局命名空间中使用内置函数 execfile 执行这个文件。

没有导入 user 模块的脚本是不会加载.pythonrc.py 的。当然，任何给定的脚本都可以自由安排其他特定方法，以加载其要求的任何启动文件或用户提供的插件文件。这样的应用程序专用安排要比导入 user 模块更通用一些。通过 import user 加载的通用.pythonrc.py 需要可以被加载该文件的任意应用程序使用。特别的应用程序专用启动文件和用户提供的插件文件只需要遵循特定应用程序所要求的任何规定。

例如，应用程序 MyApp.py 可以用来在用户的主目录（由环境变量 HOME 指定）中查找名为.myapprc.py 的文件，并在应用程序的主脚本的全局命名空间中加载该文件。要想实现这个功能，可以在主脚本中使用下面这段代码：

```
import os
homedir = os.environ.get('HOME')
if homedir is not None:
    userscript = os.path.join(homedir, '.myapprc.py')
    if os.path.isfile(userscript):
        execfile(userscript)
```

在这种情况下，如果主目录下存在.myapprc.py 用户自定义脚本，则该脚本只用来处理 MyApp 应用程序专用的用户自定义任务。



第 14 章

线程和进程

线程（thread）是一个可以与其他线程共享全局状态的控制流，所有线程看起来是同时执行的。熟练掌握线程的使用并不是很容易的，但是，一旦掌握了线程的使用，就可以使用更简单的体系结构来处理某些难题，与传统的“单线程”编程相比，使用线程往往会获得更好的性能。本章将介绍 Python 为处理线程而提供的一些工具，包括 thread、threading 和 Queue 模块。

进程（process）是正在运行的程序的一个实例。有时候，特别是在类 UNIX 操作系统或在多处理器计算机上，使用多进程要比使用多线程获得更好的结果。操作系统可以保护线程之间的相互操作。想要相互通信的线程必须使用本地进程间通信（IPC）来显式实现通信。进程可以通过文件（参见第 10 章）或者数据库（参见第 11 章）进行通信。在这两种情况下，使用这样的数据存储机制实现进程通信的常用方法就是，一个进程可以写入数据，而另一个进程以后再将这个数据读取出来。本章将介绍 Python 标准库中的 process 模块；os 模块中与进程相关的部分；还包括使用管道实现简单的 IPC；以及被称为内存映射文件的跨平台 IPC 机制，这种机制是由 mmap 模块提供给 Python 程序的。

网络机制非常适合于 IPC，因为这些机制是在网络的不同节点上运行的进程之间工作的，就像这些进程在相同节点上运行一样（本书第 20 章介绍了为 IPC 提供灵活性基础的底层网络机制）。其他被称为分布式计算的高层机制，比如 CORBA、DCOM/COM+、EJB、SOAP、XML-RPC 和.NET，可以让 IPC 更加简单，不管是本地 IPC 还是远程 IPC。但是，本书没有进一步介绍分布式计算。

14.1 Python 中的线程

Python 在支持线程的平台上提供了多线程功能，比如在 Win32、Linux 和大多数其

他 UNIX 变体上。经典的 Python 解释器不能自由切换线程。在当前的 Python 实现中，Python 使用了全局解释器锁（GIL）来确保线程之间的切换只发生在字节代码指令之间，或者在 C 语言代码故意释放 GIL 时（Python 自己的 C 语言代码可以通过阻塞 I/O 和睡眠（sleep）操作来释放 GIL）。如果一个动作可以保证，在这个动作开始和结束之间在 Python 的进程中不会发生线程切换操作，则可以将这个动作称为“原子操作”（atomic）。实际上，在当前的 Python 实现中，在对内置类型的对象执行操作是，看起来是原子操作的操作（比如，简单赋值和访问）实际上都是原子的（但是，参数赋值和多重赋值都不是原子的）。尽管如此，依赖于原子性通常并不是一个好主意。例如，开发者不会知道什么时候会处理一个派生类，而不是一个内置类型对象，这意味着要对 Python 代码进行回调，这样，任何对原子性的假定都会被证明为无法保证的。此外，依赖于原子性相关的实现会将开发者的代码锁定为一种特殊的实现，并有可能妨碍将来的升级。更好的策略就是使用本章其余部分介绍的同步工具。

Python 提供了两种不同风格的多线程功能。老式和底层模块 thread 提供了最小化的功能，不推荐在代码中直接使用该模块。建立在 thread 之上的高层模块 threading 得到了 Java 线程的启示，是推荐的工具。多线程系统中的关键设计问题通常就是如何最好地协调多个线程。threading 为此提供了几个同步对象。Queue 模块对于线程同步是非常有用的，因为该模块提供了一个同步 FIFO 队列类型，这个类型非常便于处理线程之间的通信和协调。

14.2 thread 模块

对于 thread 模块，开发者的代码必须直接使用的唯一一个部分就是 thread 模块提供的锁对象。锁是简单的线程同步原语。从技术上讲，thread 的锁是不能重入和没有所有者的：这些锁不跟踪哪个线程最后锁住了锁，因此，对于一个锁而言，没有特定的所有者线程。锁只能处于两种状态之一：锁定状态或未锁定状态。

要想获得一个新的锁对象（处于未锁定状态），可以调用名为 allocate_lock 的工厂函数，不用带任何参数。thread 和 threading 模块都提供了这个函数。锁对象 L 提供了如表 14-1 所示的 3 个方法。

表 14-1

acquire	`L.acquire(wait=True)` 在 wait 为 True 时，acquire 将锁定 L。如果 L 已经被锁定，调用线程将挂起并等待，直到 L 被解锁，然后该线程将锁定 L。即使调用线程就是最后锁定 L 的线程，该线程仍然要挂起并等待，直到另一个线程释放 L。在 wait 为 False 并且 L 未被锁定时，acquire 将锁定 L 并返回 True。在 wait 为 False 并且 L 被锁定时，acquire 不影响 L 的状态，并返回 False。
locked	`L.locked()` 如果 L 被锁定，则返回 True；否则，返回 False。

续表

release	*L*.release() 解除锁定 L，L 必须是被锁定的。在 L 被锁定时，任何线程都可以调用 L.release，而不仅仅是最后锁定 L 的线程。在多个线程都在等待 L（也就是，调用了 L.acquire，发现 L 被锁定，然后等待 L 被解锁）时，release 将唤醒任何一个正在等待的线程。调用 release 的线程不再是被挂起的：该线程将保持就绪状态，并将继续执行。

14.3 Queue 模块

Queue 模块提供了先进先出（FIFO）队列功能以支持多线程访问，该模块提供的一个主类和两个异常类如表 14-2 所示。

表 14-2

Queue	class Queue(*maxsize*=0) Queue 类是 Queue 模块的主类，参见第 14.3 节。在 maxsize 大于 0 时，一个新的 Queue 实例 q 在队列中包含 maxsize 个项目时被认为是满的。在 q 为满时，如果线程将使用 block 选项插入一个项目，则该线程将被挂起，直到另一个线程提取出一个项目。在 maxsize 小于或等于 0 时，可以认为 q 永远都不会满，并且，队列大小的限制只取决于可用的内存数量，就像普通 Python 容器一样。
Empty	Empty 是在 q 为空时，q.get(False)引发的异常类。
Full	Full 是在 q 为满时，q.put(x, False)引发的异常类。

队列实例的方法

Queue 类的实例 q 提供了如表 14-3 所示的方法。

表 14-3

empty	*q*.empty() 如果 q 为空，则返回 True；否则，返回 False。
full	*q*.full() 如果 q 为满，则返回 True；否则，返回 False。
get, get_nowait	*q*.get(*block*=true,*timeout*=None) 在 block 为 False 时，如果 q 中至少有一个项目，get 将从 q 中删除并返回一个项目；否则，get 将引发 Empty 异常。在 block 为 True 且 timeout 为 None 时，get 将从 q 中删除并返回一个项目，如果 q 为空，则 get 将挂起调用线程，直到有一个项目可用。在 block 为 True 且 timeout 不为 None 时，timeout 必须是一个大于等于 0 的数（这个数可能会包含小数部分以表示秒的分数），这样，get 将等待不超过 timeout 秒（如果到时候仍没有可用项目，get 将引发 Empty 异常）。g.get_nowait( )与 g.get(False)类似，也类似于 g.get(timeout=0.0)。get 将按照 put 插入项目的顺序删除并返回项目（FIFO）。

续表

put, put_nowait	*q*.put(*item*,*block*=true,*timeout*=None) 在 block 为 False 时，如果 q 不为满，put 将向 q 中添加项目 item；否则，put 将引发 Full 异常。在 block 为 True 且 timeout 为 None 时，put 将把 item 添加到 q 中，如果 q 为满，则 put 将挂起调用线程，直到 q 不为满。在 block 为 Ture 且 timeout 不为 None 时，timeout 必须是一个大于等于 0 的数（这个数可能会包含小数部分以表示秒的分数），并且 put 将等待不超过 timeout 秒（如果到时候 q 仍为满，则引发 Full 异常）。q.put_nowait(item)与 q.put(item, False)也类似于 q.put(item, timeout=0.0)。
qsize	*q*.qsize() 返回当前在 q 中的项目的数量。

Queue 提供了一个很好的示例来证实习惯用语“要求谅解要比要求许可更容易”（EAFP），参见第 6.6 节。由于有了多线程，每个不改变 q 的内容方法都只能进行查询。在其他一些线程执行和改变 q 时，在线程获得信息的时刻到线程接下要来对该信息进行处理的时刻之间，事情可能会发生变化。依赖于“三思而后行”（LBYL）的习惯用法是没有用的，并且胡乱使用锁来尝试和修改队列实际上就是浪费精力。要避免使用下面的 LBYL 代码：

```
if q.empty(): print "no work to perform"
else: x=q.get_nowait()
```

而是使用下面这个更简单，但是更健壮的 EAFP 解决方案：

```
try: x=q.get_nowait()
except Queue.Empty: print "no work to perform"
```

通过创建子类自定义 Queue 类

如果开发者要求使用 Queue.Queue 类本身的线程安全行为，但是不想使用 FIFO 排队规则，可以通过创建子类来自定义 Queue.Queue，并且需要覆盖 Queue.Queue 类为这个目的提供的某些或所有钩子（hook）方法：_qsize、_empty、_full、_put 和_get。每个方法都有对应于具有相应名称的公共方法的语义，但是这些方法要更简单一些，不用担心线程、超时或错误检查。在这些方法中，唯一一个带参数（除了常规的 self 参数之外）的方法就是_put（这个方法将要放入队列的元素作为其参数）。

Queue.Queue 可以确保这些钩子方法只在正确设置了线程安全的状态下被调用（也就是，Queue.Queue 自己的方法可以确保所有需要的锁定操作），并且，这些钩子方法不需要担心错误检查（例如，_get 只有在该队列为非空时被调用——在_empty 刚返回了一个为 False 的结果时）。例如，使用 LIFO 排队规则创建一个线程安全队列类要做的所有操作如下：

```
import Queue
class LIFOQueue(Queue.Queue):
    def _get(self): return self.queue.pop()
```

这段代码利用了 Queue.Queue 实例已经拥有的 self.queue 属性（一个类型为 collections.deque 的实例，参见第 8.5 节。

14.4 threading 模块

threading 模块建立在 thread 模块之上，并以更可用和更高层的形式提供了多线程功能。threading 的常规解决方案与 Java 类似，但是锁和条件都被模型化为分开的对象（在 Java 中，这样的功能是每个对象的一部分），并且线程不能直接从外部控制（这意味着没有优先级、分组、解构或停止）。threading 模块提供的对象的所有方法都是原子的。

threading 模块提供了许多类来处理线程，包括 Thread、Condition、Event、RLock 和 Semaphore。除了在下面几节中详细介绍的类的工厂函数之外，threading 模块还提供了 currentThread 工厂函数。

表 14-4

currentThread	`currentThread()` 为调用线程返回一个 Thread 对象。如果调用线程不是由 threading 模块创建的，currentThread 将创建并返回一个具有有限功能的半哑元（semi-dummy）Thread 对象。

线程对象

线程（Thread）对象 t 可以模型化一个线程。在开发者创建 t 时，可以将 t 的主函数传递为一个参数，或者也可以创建 Thread 的子类并覆盖 run 方法（还可以覆盖__init__，但是不能覆盖其他方法）。在创建 t 时，t 还没有准备运行；要想让 t 就绪（活动），可以调用 t.start( )。一旦 t 是活动的，t 将在其主函数结束时结束，不管是正常结束还是传播了一个异常。线程 t 可以是一个驻留程序，这意味着即使 t 仍然是活动的，Python 也可以终止运行，而一个普通（非驻留程序）线程将保持 Python 继续运行，直到该线程结束。Thread 类提供了如表 14-5 所示的构造函数和方法。

表 14-5

Thread	`class Thread(`*`name`*`=None,`*`target`*`=None,`*`args`*`=(),`*`kwargs`*`={ })` 总是使用具名参数调用 Thread。正式参数的数量和顺序将来有可能会更改，但是会保证现有参数的名称继续保留。在实例化 Thread 类本身时，必须指定 target：t.run 调用 target(*args,**kwargs)。在创建 Thread 的子类并覆盖 run 时，通常不需要指定 target。在任何一种情况下，只有在调用 t.start( )之后，线程才开始执行。如果 Thread 的子类 T 覆盖了__init__，T.__init__必须在调用任何其他 Thread 方法之前对 self 调用 Thread.__init__。
getName, setName	*t*`.getName()` *t*`.setName(`*`name`*`)` getName 将返回 t 的名称，而 setName 将重新绑定 t 的名称。name 可以是任意字符串，并且不要求线程的名称在所有线程中是唯一的。

续表

isAlive	`t.isAlive()` 如果 t 是活动的，将返回 True（也就是，如果 t.start 已经执行，而 t.run 还没有结束）。否则，isAlive 将返回 False。
isDaemon, setDaemon	`t.isDaemon()` `t.setDaemon(daemonic)` 如果 t 是一个驻留程序（也就是，即使 t 仍然是活动的，Python 可以终止整个处理过程；这样的终止操作还将结束 t），isDaemon 将返回 True；否则，isDaemon 将返回 False。最开始，当且仅当创建 t 的线程是一个驻留程序时，t 才是一个驻留程序。开发者只能在 t.start 之前调用 t.setDaemon；如果 daemonic 为 True，t.setDaemon 将把 t 设置为一个驻留程序。
join	`t.join(timeout=None)` 调用线程（不能是 t）将会挂起，直到 t 结束。参见第 14.4 节中介绍的 timeout。只能在 t.start 之后调用 t.join。
run	`t.run()` run 是用来执行 t 的主函数的方法。Thread 的子类通常会覆盖 run。除非被覆盖，run 将调用在创建 t 时传递的 target 可调用函数。不要直接调用 t.run；正确地调用 t.run 是 t.start 的工作！
start	`t.start()` start 可以让 t 活动，并让 t.run 在一个单独的线程中执行。对于给定的线程对象 t，只能调用一次 t.start 方法。

线程同步对象

threading 模块提供了几个同步原语，这些原语是用来实现线程通信和协调的对象。每个原语都有特定的用法。不过，只要开发者避免使用多个线程都可以访问的全局变量，Queue 通常可以提供所有需要的协调功能。本书第 14.5 节显示了如何使用 Queue 对象为多线程程序提供简单和有效的架构，通常不需要使用任何同步原语。

Timeout 参数

同步原语 Condition 和 Event 都提供了一个可以接受一个可选 timeout 参数的 wait 方法。Thread 对象的 join 方法也可以接受一个可选的 timeout 参数。timeout 参数可以为 None（默认值）以获得常规的阻塞行为（调用线程将会挂起并等待，直到满足需要的条件）。如果 timeout 参数不为 None，则 timeout 参数是一个浮点型值，表示以秒为单位的时间间隔（timeout 还可以有小数部分，因此可以表示任意长度的时间间隔，即使是非常短的时间）。在经过了 timeout 秒数之后，调用线程将再次变成就绪状态，即使还不满足需要的条件。timeout 可以用于设计可以解决一个或几个线程中出现偶然异常请求的系统，由此可以让这个系统更加健壮。但是，使用 timeout 也可能让程序的运行更慢。

Lock 和 RLock 对象

threading 模块提供的 Lock 对象与 thread 模块提供的 Lock 对象相同，参见第 14.2 节。RLock 对象提供了与 Lock 对象相同的方法。不过，RLock 对象 r 的语义通常要更方便一些。RLock 是一个“重入”锁，这意味着在 r 被锁定时，r 将跟踪其所有者线程（也就是，锁定 r 的那个线程）。所有者线程可以再次调用 r.acquire 而不会出现阻塞；r 只是将其内部计数器增加 1。对于 Lock 对象，在类似的情况下，线程将永远阻塞（直到锁被其他线程释放）。

RLock 对象 r 只有在 release 被调用了与 acquire 被调用的相同次数时才会被解锁。只有线程所有者 r 才能调用 r.release。RLock 可以用来确保在一个对象的方法相互调用时对这个对象的排他性访问；每个方法可以在开始时获得 RLock 实例，在结束后释放相同的 RLock 实例。try/finally（参见第 6.1 节是确保锁确实被释放的一个好方法（在 Python 2.5 中，新的 with 语句同样也很好，参见第 6.1 节中介绍的 with 语句）。

Condition 对象

Condition 对象 c 可以包装一个 Lock 或 RLock 对象 L。Condition 类提供了如表 14-6 所示的构造函数和方法。

表 14-6

Condition	`class Condition(`*lock*`=None)` Condition 可以创建并返回一个新 Condition 对象 c，并将锁对象 L 设置为 lock。如果 lock 为 None，L 将被设置为一个新创建的 RLock 对象。
acquire, release	*c*`.acquire(`*wait*`=1)` *c*`.release()` 这些方法将调用 L 的对应方法。除非一个线程拥有锁 L，否则该线程不能对 c 调用任何其他方法。
notify, notifyAll	*c*`.notify()` *c*`.notifyAll()` notify 可以唤醒正在等待 c 的线程中的某一个线程。调用线程在调用 c.notify( )之前必须拥有 L，并且 notify 不会释放 L。除非被唤醒的线程可以再次得到 L，否则该线程不会变为就绪状态。因此，调用线程通常会在调用 notify 之后调用 release。notifyAll 类似于 notify，区别在于 notifyAll 将唤醒所有正在等待的线程，而不只是其中的一个。
wait	*c*`.wait(`*timeout*`=None)` wait 将释放 L，然后挂起调用线程，直到其他线程对 c 调用 notify 或 notifyAll。调用线程在调用 c.wait( )之前必须拥有 L。第 14.4 节中介绍了 timeout。不管是由于通知（notify）还是超时，在一个线程被唤醒后，如果该线程再次获得 L，该线程将变为就绪状态。在 wait 返回时，调用线程总是会再次拥有 L。

在通常的使用中，Condition 对象 c 可以用来控制访问某些线程间共享的全局状态 s。在一个线程需要等待 s 来改变时，该线程可以按下面的方式循环执行：

```
c.acquire()
while not is_ok_state(s):
    c.wait()
do_some_work_using_state(s)
c.release()
```

同时，每个要对 s 进行修改的线程将在每次 s 改变时调用 notify（或者 notifyAll，如果该线程需要唤醒所有正在等待的线程，而不只是其中的一个）：

```
c.acquire()
do_something_that_modifies_state(s)
c.notify()    # 或者是 c.notifyAll()
c.release()
```

可以看出，开发者总是需要在每次使用 c 的方法之前和之后获得和释放 c，这使得 Condition 的使用有些容易出错。

Event 对象

事件（Event）对象可以让任意数量的线程挂起并等待。等待事件对象 e 的所有线程在任何其他线程调用 e.set( )时将变为就绪状态。事件对象 e 有一个标记，可以记录该事件是否已经发生；在 e 被创建时，这个标记的初始值为 False。Event 就是这样一个类似于简化的 Condition 的比特。事件对象可以用于对只有一次的改变发出信号，但是不适合于更多的常规使用；特别是，依赖于调用 e.clear( )是容易产生错误的。Event 类提供了如表 14-7 所示的方法。

表 14-7

Event	`class Event()` Event 可以创建并返回一个新事件对象 e，并且 e 的标记被设置为 False。
clear	`e.clear()` 将 e 的标记设置为 False。
isSet	`e.isSet()` 返回 e 的标记的值，True 或 False。
set	`e.set()` 将 e 的标记设置为 True。所有等待 e 的线程（如果有的话）将变为准备运行。
wait	`e.wait(`*timeout*`=None)` 如果 e 的标记为 True，wait 将立即返回。否则，wait 将挂起调用线程，直到其他一些线程调用 set。第 14.4 节中介绍了 timeout。

Semaphore 对象

信号量（也被称为计数信号量，counting semaphore）是广义上的锁。Lock 的状态可以被看作是 True 或 False；信号量对象 s 的状态时一个 0～n 的数字，n 是在 s 被创建时设置的。信号量可以用于管理固定的资源池（例如，4 个指针或 20 个套接字），尽管使用

队列来实现这个功能要更健壮一些。

表 14-8

Semaphore	`class Semaphore(n=1)` Semaphone 可以创建并返回一个状态被设置为 n 的信号量对象 s。信号量对象 s 提供了以下方法。
acquire	`s.acquire(wait=True)` 在 s 的状态大于 0 时，acquire 将把状态值减 1 并返回 True。在 s 的状态为 0，并且 wait 为 True 时，acquire 将挂起调用线程并等待，直到其他一些线程调用了 s.release。在 s 的状态为 0，并且 wait 为 Flase 时，acquire 将立即返回 False。
release	`s.release()` 在 s 的状态大于 0，或者在状态为 0，但是没有线程正在等待 s 时，release 将把状态增加 1。在 s 的状态为 0，并且有些线程正在等待 s 时，release 将把 s 的状态设为 0，并唤醒任意一个等待线程。调用了 release 的线程不再被挂起，该线程将保持就绪，并继续正常执行。

线程本地存储

在 Python 2.4 中，threading 模块提供了一个 local 类，线程可以使用这个类获得线程本地存储（thread-local storage，TLS），也被称为每线程数据（per-thread data）。local 类的实例 L 具有任意命名的属性，开发者可以设置（set）和获得（get）这些属性，并将其保存在自己可以访问的字典 L.__dict__中。L 是完全线程安全的，这意味着多个线程同时设置和获得 L 的属性是没有问题的。最重要的是，每个访问 L 的线程都可以看到一个完全不交叉的属性集合，并且，在一个线程中所作的任何更改都不会对其他线程产生影响。例如：

```
import threading
L = threading.local()
print 'in main thread, setting zop to 42'
L.zop = 42
def targ():
  print 'in subthread, setting zop to 23'
  L.zop = 23
  print 'in subthread, zop is now', L.zop
t = threading.Thread(target=targ)
t.start()
t.join()
print 'in main thread, zop is now', L.zop
# 结果是:
# in main thread, setting zop to 42
# in subthread, setting zop to 23
# in subthread, zop is now 23
# in main thread, zop is now 42
```

TLS 可以帮助开发者更容易地编写要在多个线程中运行的代码，因为开发者可以在多个线程中使用相同的命名空间（一个 threading.local 实例），而且线程之间不会相互干扰。

14.5 线程程序架构

一个线程程序必须总是安排单个（single）线程来处理任何给定的对象或该程序外部的子系统（比如文件、数据库、GUI 或网络连接）。让多个线程处理同一个外部对象通常可能会导致不可预料的问题。

只要线程程序必须处理某些外部对象，可以专门指定一个使用 Queue 对象的线程来实现这样的处理，通过这个 Queue 对象，外部接口线程可以通过这个 Queue 对象获得其他线程放入的工作请求。外部接口线程可以将结果放到一个或多个其他 Queue 对象来返回这些结果。下面的示例显示了如何将这种架构包装到一个通用的可重用类中，假定对外部子系统执行的每个单元的工作都可以通过一个可调用对象来表示：

```
import threading, Queue
class ExternalInterfacing(Threading.Thread):
    def __init__(self, externalCallable, **kwds):
        Threading.Thread._ _init__(self, **kwds)
        self.setDaemon(1)
        self.externalCallable = externalCallable
        self.workRequestQueue = Queue.Queue()
        self.resultQueue = Queue.Queue()
        self.start()
    def request(self, *args, **kwds):
        "called by other threads as externalCallable would be"
        self.workRequestQueue.put((args,kwds))
        return self.resultQueue.get()
    def run(self):
        while 1:
            args, kwds = self.workRequestQueue.get()
            self.resultQueue.put(self.externalCallable(*args, **kwds))
```

只要某些 ExternalInterfacing 对象 ei 已经被实例化，所有其他线程都可以调用 ei.request，就像调用不具备外部接口机制的 someExternalCallable 一样（根据实际情况，带有或不带参数）。ExternalInterfacing 机制的好处在于，所有对 someExternalCallable 的调用现在都是连续的。这意味着这些调用都是由一个线程（绑定到 ei 的线程对象）按照定义的连续顺序执行的，并且不会出现重叠和紊乱情况（由于不确定哪个线程首先调用而产生的难以调试的错误），或者由此产生的其他异常情况。

为了实现更好地通用性，如果几个可调用参数需要被连续放在一起，开发者可以将该可调用参数作为工作请求的一部分，而不是像平常那样，在 ExternalInterfacing 类进行

初始化的时候传递该参数。下面的示例显示了这种更通用的解决方案：

```
import threading, Queue
class Serializer(Threading.Thread):
    def __init__(self, **kwds):
        Threading.Thread.__init__(self, **kwds)
        self.setDaemon(1)
        self.workRequestQueue = Queue.Queue()
        self.resultQueue = Queue.Queue()
        self.start()
    def apply(self, callable, *args, **kwds):
        "called by other threads as callable would be"
        self.workRequestQueue.put((callable, args,kwds))
        return self.resultQueue.get()
    def run(self):
        while 1:
            callable, args, kwds = self.workRequestQueue.get()
            self.resultQueue.put(callable(*args, **kwds))
```

只要 Serializer 对象 ser 已经被实例化，其他线程都可以调用 ser.apply (someExternalCallable)，就像调用不具备连续化机制的 someExternalCallable 一样（根据实际情况，带有或不带更多参数）。Serializer 机制具有与 ExternalInterfacing 相同的好处，区别在于单个 ser 实例包装的相同或不同的可调用参数对应的所有调用现在都被连续化了。

整个程序的用户接口是一个外部子系统，因此必须使用单个线程来处理，特别是该程序的主线程（这对于某些用户接口工具包来说是强制性的，即使在不强制的时候，也建议这样处理）。因此，使用 Serializer 线程是不适当的。正确的做法是，程序的主线程必须只处理用户接口问题，而将实际工作交付给工作线程（worker thread），该线程可以接受一个 Queue 对象上的工作请求，并将结果返回到另一个 Queue 对象。一组工作线程也被称为线程池（thread pool）。正如下面的示例显示的，所有工作线程必须共享单个请求队列和单个结果队列，因为主线程就是唯一一个可以获得工作请求并产生结果的线程：

```
import threading
class Worker(Threading.Thread):
    requestID = 0
    def __init__(self, requestsQueue, resultsQueue, **kwds):
        Threading.Thread.__init__(self, **kwds)
        self.setDaemon(1)
        self.workRequestQueue = requestsQueue
        self.resultQueue = resultsQueue
        self.start()
    def performWork(self, callable, *args, **kwds):
        "called by the main thread as callable would be, but w/o return"
        Worker.requestID += 1
        self.workRequestQueue.put((Worker.requestID, callable, args,
        kwds))
```

```
            return Worker.requestID
        def run(self):
            while 1:
                requestID, callable, args, kwds = self.workRequestQueue.get()
                self.resultQueue.put((requestID, callable(*args, **kwds)))
```

主线程可以创建两个队列，然后像下面这样实例化工作线程：

```
import Queue
requestsQueue = Queue.Queue()
resultsQueue = Queue.Queue()
for i in range(numberOfWorkers):
    worker = Worker(requestsQueue, resultsQueue)
```

现在，不管主线程什么时候需要交付工作（执行某些可调用对象，执行这些对象可能会花费一些时间才能产生结果），主线程将调用 worker.performWork(callable)，就像调用没有这种机制的 callable 对象一样（根据实际情况，带有或不带更多参数）。但是，performWork 不返回调用的结果。除了结果之外，主线程还将得到一个用来标识这个工作请求的 id。如果主线程需要结果，可以跟踪这个 id，因为工作请求的结果在出现时将使用这个 id 来标记。这种机制的好处在于，主线程不会被锁定以等待可调用对象完成长时间的执行过程，而是立刻再次变为就绪状态，并且可以立即返回到其主要任务，也就是处理用户接口上。

在从队列中得到一个请求的工作线程完整了计算结果的操作之后，主线程必须安排检查 resultsQueue 的操作，因为每个工作请求的结果最终都会出现在 resultsQueue 中，并使用该请求的 id 来标记。主线程如何安排检查这两个用户接口事件，以及从工作线程返回到结果队列上的结果取决于使用了什么样的用户接口工具包，如果这个用户接口是基于文本的，还取决于程序运行的平台。

主线程可以使用的一种广泛可用的常用策略就是轮询（poll）（也就是，周期性地检查结果队列的状态）。在大多数类 UNIX 平台上，signal 模块的 alarm 函数允许轮询。Tkinter GUI 工具包提供了 after 方法，这个方法也可以用于轮询。有些工具包和平台还提供了一些更有效的策略，可以让工作线程在其将某些结果放置到结果队列中时提醒主线程，但是，目前还没有普遍可用的、跨平台和跨工具包的方法来处理这样的操作。因此，下面这个模拟的示例忽略了这样的用户接口事件，而只是通过在几个工作线程上使用随机延时计算随机表达式来模拟真实的工作，这样就对前一个示例进行了进一步的完善：

```
import random, time
def makeWork():
   return "%d %s %d"%(random.randrange(2,10),
        random.choice(('+', '-', '*', '/', '%', '**')),
        random.randrange(2,10))
def slowEvaluate(expressionString):
    time.sleep(random.randrange(1,5))
    return eval(expressionString)
workRequests = {  }
```

```
def showResults():
    while 1:
        try: id, results = resultsQueue.get_nowait()
        except Queue.Empty: return
        print 'Result %d: %s -> %s' % (id, workRequests[id], results)
        del workRequests[id]
for i in range(10):
    expressionString = makeWork()
    id = worker.performWork(slowEvaluate, expressionString)
    workRequests[id] = expressionString
    print 'Submitted request %d: %s' % (id, expressionString)
    time.sleep(1)
    showResults()
while workRequests:
    time.sleep(1)
    showResults()
```

14.6 进程环境

操作系统为每个进程 P 提供了一个环境（environment），环境是一组名为标识符的环境变量（按常规，通常是大写字母的标识符），并且其内容都是字符串。在第 3.1 节中，本书介绍了影响 Python 操作的环境变量。操作系统 Shell 命令行提供了一些方法，可以通过第 3.1 节中提到的 shell 命令和其他方法来检查和修改环境变量。

任意进程 P 的环境是在 P 启动的时候确定的。在启动之后，只有 P 本身可以更改 P 的环境。P 所作的操作不会对 P 的父进程的环境（启动 P 的进程）有任何影响，不会对以前从进程 P 中启动，并且现在正在运行的那些子进程造成影响，也不会影响与 P 无关的进程。对 P 的环境进行的更改只影响 P 本身：环境并不是一种进程间通信（IPC）的方式。P 的子进程通常会得到 P 的环境的一个副本以将其作为自己的启动环境。在这样的理解下，对 P 的环境进行的更改肯定会影响 P 在环境更改之后启动的子进程。

os 模块提供了 environ 属性，这个属性是一个映射，表示当前进程的环境。os.environ 是在 Python 启动时从进程环境中初始化的。如果平台支持对当前进程的环境进行更新，则对 os.environ 所作的更改将会更新当前进程的环境。os.environ 中的键和值必须是字符串。在 Windows 中，os.environ 中的键都是显式大写字母，而在类 UNIX 平台上不要求如此。例如，下面是一个如何确定 Python 程序是在什么 shell 或命令处理器下运行的例子：

```
import os
shell = os.environ.get('COMSPEC')
if shell is None: shell = os.environ.get('SHELL')
if shell is None: shell = 'an unknown command processor'
print 'Running under', shell
```

在 Python 程序更改其环境时（例如，通过 os.environ['X']='Y'），这样做不会影响启动该

程序的 shell 或命令处理器。正如已经介绍的，对于包括 Python 在内的所有编程语言，更改一个进程的环境只影响该进程本身，而不会影响其他程序的环境。

14.7 运行其他程序

开发者可以通过 os 模块中的函数运行其他程序，在 Python 2.4 中，还可以使用新的 subprocess 模块运行其他程序。

使用 os 模块运行其他程序

在 Python 2.4 中，程序运行其他进程的最好方法就是使用新的 subprocess 模块，参见第 14.7 节。但是，os 模块还提供了其他几种方法来实现这样的操作，在某些情况下，这些方法可能要更简单一些，或者可以让代码保持对老版本 Python 的向后兼容。

运行其他程序最简单的方法就是使用 os.system 函数，尽管这个函数不提供方法来控制外部程序。os 模块还提供了许多名称以 exec 开始的函数。这些函数提供了更细粒度的控制。任何一个 exec 函数运行的程序将替换相同进程中当前运行的程序（也就是，Python 解释器）。因此，实际上大多数时候是在可以让进程通过 fork 复制其本身的平台上（即类 UNIX 平台）使用 exec 函数。名称以 spawn 和 popen 开始的 os 函数提供了适中的简单性和功能：这两个函数都是跨平台的，并且不像 system 那样简单，但是，对于大多数应用而言是很简单的，也足够使用了。

exec 和 spawn 函数可以运行一个指定的可执行文件，需要给定这个可执行文件的路径、要传递的参数和可选的环境映射。system 和 popen 函数可以执行一个命令，这个命令是一个传递到平台的默认 shell（通常，在 UNIX 上是/bin/sh，在 Windows 上是 command.com 或 cmd.exe）的新实例上的字符串。命令是一个比可执行文件更一般的概念，因为命令可以包含使用当前平台上专用的常规 shell 语法的 shell 功能（管道、重定向和内置 shell 命令）。

表 14-9

execl, execle, execlp, execv, execve, execvp, execvpe	`execl(path,*args)` `execle(path,*args)` `execlp(path,*args)` `execv(path,args)` `execve(path,args,env)` `execvp(path,args)` `execvpe(path,args,env)` 这些函数可以运行字符串 path 指定的可执行文件（程序），并替换当前进程中的当前程序（也就是，Python 解释器）。这些函数的名称之间的区别（在前缀 exec 之后）用来控制与如何找到和运行新程序有关的 3 个问题。 • path 必须是程序的可执行文件的完整路径吗？或者这些函数可以像操作系统的 shell 工具所作的那样，接受一个文件名作为 path 参数，并在几个目录中搜索这个可执行文件吗？execlp、execvp 和 execvpe 可以接受一个只是文件名，而不是一个完整路径的 path 参数。在这种情况下，函数将在 os.environ['PATH']列出的目录中搜索具有这个名称的可执行文件。其他函数都要求 path 是一个到新程序的可执行文件的完整路径。

续表

execl, execle, execlp, execv, execve, execvp, execvpe	• 新程序的参数可以作为这些函数的单个序列参数 args，或者这些函数的分开的参数吗？名称以 execv 开始的函数可以接受单个参数 args，args 是一个在新程序中使用的参数序列。名称以 excel 开始的函数将把新程序的参数作为分开的参数（特别是，execle 将使用最后一个参数作为新程序的环境）。 • 新程序的环境可以被接受为这些函数的一个显式映射参数 env 吗？或者 os.environ 可以被显式使用吗？execle、execve 和 execvpe 可以接受一个为映射的参数 env，并将该参数用作新程序的环境（键和值必须是字符串），而其他的函数都使用 os.environ 来实现这个功能。 每个 exec 函数都使用 args 中的第一个参数作为正在运行的新程序的名称（例如，C 语言程序的 main 函数中的参数 argv[0]），只有 args[1:0]才被传递为新程序的正确参数。
popen	popen(*cmd*,*mode*='r',*bufsize*=-1) 在一个新进程 P 中运行字符串命令 cmd，并返回一个类文件对象 f，f 包装了一个到 P 的标准输入的管道和一个来自于 P 的标准输出的管道(取决于 mode)。mode 和 bufsize 与 Python 的内置 open 函数的 mode 和 bufsize 具有相同的含义，参见第 10.3 节。在 mode 为‘r’（或者‘rb’，用于二进制模式的读取）时，f 是只读的，并将包装 P 的标准输出。在模式为‘w’（或者‘wb’时，用于二进制模式的写入），f 是只写的，并将包装 P 的标准输入。 f 与其他类文件对象之间的关键区别在于 f.close 方法的行为。f.close 将等待 P 结束，在 P 成功地结束时，f.close 将返回 None，就像类文件对象的 close 方法通常所做的那样。但是，如果操作系统在 P 结束时给出了一个整数错误代码 c，则表示 P 没有成功地结束，f.close 还是会返回 c。并不是所有的操作系统都支持这种机制的：在某些平台上，f.close 总是会返回 None。在类 UNIX 平台上，如果 P 以系统调用 exit(n)结束（例如，如果 P 是一个 Python 程序，并调用 sys.exit(n)结束），f.close 将从操作系统接收到代码 256*n，并将该代码返回到 f.close 的调用者。
popen2, popen3, popen4	popen2(*cmd*,*mode*='t',*bufsize*=-1) popen3(*cmd*,*mode*='t',*bufsize*=-1) popen4(*cmd*,*mode*='t',*bufsize*=-1) 这些函数都将在一个新进程 P 中运行字符串命令 cmd，并返回一个由类文件对象组成的元组，这些类文件对象包装了到 P 的标准输入的管道，以及来自于 P 的标准输出和标准错误的管道。mode 可以是‘t’以获得文本模式的类文件对象，或者是‘b’以获得二进制模式的类文件对象。在 Windows 上，bufsize 必须是-1。在 UNIX 上，bufsize 具有与 Python 的内置 open 函数的 bufsize 相同的含义，参见第 10.3 节。 popen2 将返回一个数据对（fi, fo），其中 fi 将包装 P 的标准输入（这样，调用进程就可以写入到 fi 了），fo 将包装 P 的标准输出（这样，调用进程就可以从 fo 读取了）。popen3 将返回一个由 3 个项目组成的元组（fi, fo, fe），其中 fe 将包装 P 的标准错误（这样，调用进程就可以从 fe 读取了）。popen4 将返回一个数据对（fi, foe），其中 foe 将同时包装 P 的标准输出和错误（这样，调用线程就可以从 foe 读取了）。尽管 popen3 从某种意义上讲是这 3 个函数中最通用的，但是 popen3 很难协调从 fo 和 fe 读取的数据）。在 cmd 的标准错误与开发者自己的进程的标准错误去往相同的目的地时，popen2 的使用要比 popen3 更简单，

popen2, popen3, popen4	而在 cmd 的标准错误和输出有些任意地混合在一起时，popen4 更简单。
	文件对象 fi、fo、fe 和 foe 都是普通对象，不具备 popen 函数包含的 close 方法的特殊语义。换句话讲，popen2、popen3 或 popen4 的调用者没有办法了解 P 的结束代码。
	根据 cmd 命令的缓冲策略（这通常不受开发者的控制，除非是 cmd 的作者），在开发者的进程关闭文件 fi 之前，这些函数不能从文件 fo、fe 和/或 foe 中读取任何内容。因此，这种用法的常规模式如下：
	`import os`
	`def pipethrough(cmd, list_of_lines):`
	`fi, fo = os.popen2(cmd, 't')`
	`fi.writelines(list_of_lines)`
	`fi.close()`
	`result_lines = fo.readlines()`
	`fo.close()`
	`return result_lines`
	popen 组中的函数通常不适合于交互式推进另一个进程（也就是，写入一些内容，然后读取 cmd 对这些内容的响应，然后写入另外一些内容，依此类推）。在开发者的程序第一次尝试读取响应时，如果 cmd 遵循典型的缓冲策略，则所有内容都将阻塞。换句话讲，开发者的进程正在等待 cmd 的输出，但是 cmd 已经将其挂起的输出放到内存缓冲区中，而开发者的进程不能从该缓冲区中得到输出，然后，cmd 开始等待更多的输入。这是一种典型的死锁情况。
	如果开发者拥有对 cmd 的某些控制，可以通过确保 cmd 不使用缓冲机制运行来尝试解决这个问题。例如，如果 cmd.py 是一个 Python 程序，开发者可以像下面这样运行不进行缓冲的 cmd：
	`C:/> python -u cmd.py`
	其他可能的解决方案还包括 `telnetlib` 模块（参见第 19.5 节），如果开发者的平台支持 `telnet`，以及第三方的类 UNIX 扩展（比如 `expectpy.sf.net`）和包（比如 `pexpect.sf.net`）。没有通用的解决方案适用于所有平台和所有感兴趣的 `cmd`。

spawnv, spawnve	`spawnv(`*mode*`,`*path*`,`*args*`)` `spawnve(`*mode*`,`*path*`,`*args*`,`*env*`)` 这些函数可以在一个新进程 P 中运行 path 指定的程序，并使用传递为 args 序列的参数。spawnve 使用了映射 env 作为 P 的环境（键和值都必须是字符串），而 spawnv 使用了 os.environ 来实现同样的功能。在类 UNIX 平台上，还有 os.spawn 的其他变体，对应于 os.exec 的变体，但是 spawnv 和 spawnve 是 Windows 上唯一存在的两个函数。 mode 必须是 os 模块提供的两个属性之一：os.P_WAIT 表示调用进程将会等待，直到新进程结束，而 os.P_NOWAIT 表示调用进程继续与新进程同时执行。在 mode 为 os.P_WAIT 时，该函数将返回 P 的结束代码 c：0 表示成功地结束，c 小于 0 表示 P 因一个信号（singal）而终止，而 c 大于 0 表示正常，但是不成功地结束。在 mode 为 os.P_NOWAIT 时，该函数将返回 P 的进程 ID（在 Windows 上，P 的进程句柄）。现在没有跨平台的方法来使用 P 的 ID 或句柄；平台专用方法（本书没有进一步介绍）包括类 UNIX 平台上的 os.waitpid 函数和 Windows 平台上的 win32all 扩展（starship.python.net/crew/mhammond）。

续表

spawnv, spawnve	例如，开发者的交互式程序可以为用户提供一个机会以编辑程序将要读取和使用的文本文件。开发者必须提前确定到用户喜爱的文本编辑器的完整路径，比如 Windows 的 c:\\windows\\notepad.exe 或类 UNIX 平台上的/bin/vim。假定这个路径字符串绑定到 editor 变量，而开发者想要让用户编辑的文本文件的路径绑定到 textfile：
	`import os`
	`os.spawnv(os.P_WAIT, editor, [editor, textfile])`
	args 参数的第一个项目将被传递到作为“被调用的程序的名称”而运行的程序中。大多数程序并不查看这个参数，因此开发者可以在这里放置任何字符串。只有在编辑器程序不查看这个特殊的第一个参数时，传递被用作 os.spawnv 的第二个参数的相同的 editor 字符串是最简单和最有效的解决方案。

system	system(*cmd*) 在一个新进程中运行字符串命令 cmd，如果新进程成功地结束（或者如果 Python 不能确定新进程结束的成功状态，可能会发生在 Windows 95 和 98 中），则返回 0。如果新进程不能成功地结束（并且 Python 可以确定这个不成功的结束），system 将返回一个不等于 0 的整数错误代码。

子进程模块

子进程（subprocess）模块只能在 Python 2.4 及其以后版本中使用，子进程模块提供了一个功能非常丰富的类 Popen，这个类支持许多不同的方法让开发者的程序运行其他程序。

表 14-10

Popen	class Semaphore(*n*=1) class Popen(*args*, *bufsize*=0, *executable*=None, *stdin*=None, *stdout*=None, *stderr*=None, *preexec_fn*=None, *close_fds*=False, *shell*=False, *cwd*=None, *env*=None, *universal_newlines*=False, *startupinfo*=None, *creationflags*=0) Popen 将启动一个子进程以运行一个不同的程序，并创建和返回一个对象 p，p 代表了这个子进程。args 强制参数和许多可选（命名）参数可以控制如何运行子进程的所有详细信息。 如果在子进程创建期间和在不同的程序启动之前出现了任何异常，对 Popen 的调用将重新引发带有一个名为 child_trackback 的属性的调用进程中的这个异常，child_trackback 属性是这个子进程的 Python 跟踪对象。这样的异常通常可能是一个 OSError 实例（也有可能是一个 TypeError 或 ValueError 实例，以表示开发者向 Popen 传递了一个类型或值无效的参数）。

运行什么，如何运行：args、executable 和 shell

args 是一个字符串序列（通常是一个列表）：第一个项目是到要执行的程序的路径，如

果还有其他项目，则后面的项目是传递给程序的参数（在不需要传递参数时，args 还可以只是一个字符串）。在 executable 不为 None 时，将覆盖 args 以确定要执行哪个程序。在 shell 为 True 时，executable 将指定使用哪个 shell 来运行子进程；在 shell 为 True，executable 为 None 时，在类 UNIX 系统上使用的 shell 是/bin/sh（在 Windows 上，shell 是 os.environ['COMSPEC']）。

子进程文件：stdin、stdout、stderr、bufsize、universal_newlines 和 close_fds

stdin、stdout 和 stderr 分别指定了子进程的标准输入、输出和错误文件。每个文件都可能是管道（PIPE），这样将创建一个去往/来自于子进程的新管道；这些文件也可能为 None，表示这个子进程将使用与其（“父”）进程相同的文件；或者使用一个已经适当地打开（对于标准输入，表示以可读方式打开；对于标准输出和标准错误，表示以可写方式打开）的文件对象（或者文件描述符）。stderr 还可以是 STDOUT，表示子进程的标准错误将出现在与其标准输出相同的文件上。bufsize 可以控制这些文件的缓冲区（除非这些文件已经被打开），并具有与 open 函数的 bufsize 参数（默认值为 0，表示“无缓冲”）相同的语义，参见第 10.3 节。在 universal_newlines 为 True 时，stdout 和 stderr（除非已经被打开）将以“通用换行符”（‘rU’）模式打开，参见第 10.3 节中介绍的文件模式。在 close_fds 为 True 时，子进程中的所有其他文件（除标准输入、输出和错误之外）都将在子进程的程序或 shell 执行之前被关闭。

其他参数：preexec_fn、cwd、env、startupinfo 和 creationflags

在 preexec_fn 不为 None 时，必须是一个函数，或者其他可调用对象，在子进程的程序或 shell 被执行之前，子进程将调用这个函数。

在 cwd 不为 None 时，必须是一个字符串，这个字符串给出了到已有目录的路径；在子进程的程序或 shell 被执行之前，子进程的当前目录将被更改为 cwd。

在 env 不为 None 时，必须是一个映射（通常是一个字典），键和值都是字符串，并完整地定义了新进程的环境。

startupinfo 和 creationflags 都是只能用于 Windows 的参数，可以被传递到 CreateProcess Win32 API 调用以创建用于 Windows 专用目的的子进程（因为本书主要关注于 Python 的跨平台使用，因此没有进一步对此进行介绍）。

subprocess.Pn 实例的属性

Popen 类的实例 p 提供了以下属性：

pid

子进程的进程 ID

returncode

None 表示这个子进程还不存在；否则，这个属性为一个整数：0 表示成功地结束，大于 0 表示以一个错误代码结束；小于 0 表示该子进程被一个信号终止。

stderr、stdin 和 stdout

在 Popen 的相应参数是 subprocess.PIPE 时，这几个属性分别都是一个包装了对应管道的文件对象；否则，这些属性都为 None。

subprocess.Popen 实例的方法

Popen 类的实例 p 提供了如表 14-11 所示的方法。

表 14-11

communicate	`p.communicate(input=None)` 将字符串 input 发送为子进程的标准输入（在 input 不为 None 时），然后读取子进程的标准输出和错误文件，并将其放入内存字符串 so 和 se 中，直到这两个文件结束，最后等待子进程结束并返回一个数据对（so, se）。
poll	`p.poll()` 检查子进程是否已经结束，然后返回 p.returncode。
wait	`p.wait()` 等待子进程结束，然后返回 p.returncode。

14.8 mmap 模块

mmap 模块提供了一些内存映射文件对象。mmap 对象的行为类似于普通字符串（非 Unicode），因此，在需要一个普通字符串的位置，通常都可以传递一个 mmap 对象。但是，区别如下：

- mmap 对象不提供字符串对象的方法；
- mmap 对象是可变的，而字符串对象是不可变的；
- mmap 对象还可以对应于一个打开的文件，并且表现出对 Python 文件对象的多态性（参见第 10.3 节）。

mmap 对象 m 可以被索引或切片，还可以生成普通字符串。因为 m 是可变的，开发者还可以为 m 的索引或切片赋值。但是，在为 m 的切片赋值时，赋值语句的右边必须是一个与要赋值的切片的长度完全相同的字符串。因此，许多在列表切片赋值中非常有用的赋值方式（参见第 4.6 节中介绍的修改列表）不能应用于 mmap 切片赋值。

mmap 模块提供了一个工厂函数，这个函数在类 UNIX 系统和 Windows 系统上稍微

有些区别。

表 14-12

mmap	`mmap(filedesc,length,tagname='')# Windows`
	`mmap(filedesc,length,flags=MAP_SHARED,`
	`prot=PROT_READ|PROT_WRITE) # Unix`
	创建并返回一个 mmap 对象 m，m 将把文件描述符 filedesc 指定的文件的前 length 个字节映射到内存中。filedesc 通常必须是一个以可读和可写模式（除类 UNIX 平台之外，在类 UNIX 平台上，prot 参数要求只读或者只写）打开的文件描述符（参见第 10.8 节）。要想得到一个引用 Python 文件对象 f 的 mmap 对象 m，可以使用 m=mmap.mmap(f.fileno( ),length)。
	在 Windows 平台上，所有内存映射都是可读和可写，并且在进程之间共享的，因此，所有包含对文件的内存映射的进程都可以看到每个其他进程对内存映射所做的更改。只有在 Windows 平台上，开发者可以传递一个 tagname 字符串以向该内存映射提供一个显式标签名。这个标签名可以让开发者具有几个对相同文件的映射，但是这个功能是极少需要的。只使用两个参数调用 mmap 有这样一个好处，可以让开发者的代码在 Windows 和类 UNIX 平台之间具有可移植性。
	只有在类 UNIX 平台上，开发者可以将 mmap.MAP_PRIVATE 作为 flags 参数传递，这样可以获得一个进程私有，并且写入时复制（copy-on-write）的映射。在默认情况下，mmap.MAP_SHARED 将获得一个与其他进程共享的映射，这样，所有映射该文件的进程都可以看到某一个进程所做的更改（与在 Windows 上相同）。开发者可以将 mmap.PROT_READ 传递为 prot 参数以获得一个只能读，不能写的映射。传递 mmap.PORT_WRITE 将获得一个只能写，不能读的映射。在默认情况下，按位或 mmap.PROT_READ|mmap.PROT_WRITE 将获得一个可读也可写的映射（与在 Windows 上相同）。

mmap 对象的方法

mmap 对象 m 提供了如表 14-13 所示的方法。

表 14-13

close	`m.close()` 关闭 m 对应的文件。
find	`m.find(str,start=0)` 返回大于或等于 start，并使得 str==m[i:i+len(str)]的最小索引 i。如果不存在这样的索引 i，m.find 将返回-1。这个方法与字符串对象的 find 方法具有相同的功能，参见第 9.1 节中介绍的 find 方法。
flush	`m.flush([offset,n])` 确保对 m 所做的所有更改也存在于 m 的文件中。除非调用 m.flush，否则不能保证该文件是否能反映 m 的当前状态。可以传递一个开始字节偏移量 offset 和一个字节数 n 以将这个刷新操作的效果限制为只保证 m 的一个切片。

续表

move	*m*.move(*dstoff*,*srcoff*,*n*) 与切片 m[dstoff:dstoff+n]=m[srcoff:srcoff+n]类似，但是要更快一些。源和目的切片可以重叠。除了可能出现重叠之外，move 不会影响源切片（也就是，move 方法只复制字节，但是不移动字节，尽管 move 方法的名称表示移动）。
read	*m*.read(*n*) 读取并返回一个包含从 m 的文件指针开始最多 n 个字节的字符串 s，然后向前移动 m 的文件指针 len(s)个字节。如果 m 的文件指针到 m 的长度之间小于 n 个字节，则返回可用的字节。特别是，如果 m 的文件指针位于 m 的末尾，则返回空白字符串 ''。
read_byte	*m*.read_byte() 返回一个长度为 1 的字符串，其中包含 m 的文件指针所指的那个字符，然后将 m 的文件指针向前移动 1。m.read_byte( )就像 m.read(1)一样。但是，如果 m 的文件指针位于 m 的末尾，m.read(1)将返回一个空白字符串 ''，而 m.read_byte( )将引发一个 ValueError 异常。
readline	*m*.readline() 从 m 表示的文件中读取并返回一行文本，从 m 的当前文件指针到下一个 '\n'，包括 '\n'（如果没有 '\n'，则到 m 的末尾为止），然后将 m 的文件指针向前移动到指向刚才读取的字节之后那个字节。如果 m 的文件指针位于 m 的末尾，readline 将返回空白字符串 ''。
resize	*m*.resize(*n*) 更改 m 的长度，使得 len(m)变为 n。不影响 m 的文件的大小。m 的长度和文件的大小是相互独立的。要想将 m 的长度设置为等于文件的大小，可以调用 m.resize(m.size( ))。如果 m 的长度大于文件的大小，将使用 null 字节（\x00）填充 m。
seek	*m*.seek(*pos*,*how*=0) 将 m 的文件指针设置为整数字节偏移量 pos。how 表示参考点（0 点）：在 how 为 0 时，参考点就是文件的开始；在 how 为 1 时，参考点就是 m 的当前文件指针；在 how 为 2 时，参考点是 m 的末尾。试图将 m 的文件指针设置为一个负的字节偏移量，或者设置为一个超出 m 长度的偏移量的 seek 方法调用将引发一个 ValueError 异常。
size	*m*.size() 返回 m 的文件的长度（字节数），而不是 m 本身的长度。要想获得 m 的长度，可以使用 len(m)。
tell	*m*.tell() 返回 m 的文件指针的当前位置，也就是从 m 的文件的起始位置开始的字节偏移量。
write	*m*.write(*str*) 将 str 中的字节写入到 m 中，从 m 的文件指针的当前位置开始，改写已有的字节，然后将 m 的文件指针向前移动 len(str)字节。如果 m 的文件指针和 m 的长度之间不到 len(str)个字节，则 write 将引发一个 ValueError 异常。

续表

write_byte	*m*.write_byte(*byte*) 将 byte（byte 必须是一个单字符字符串）写入到映射 m 中 m 的文件指针的当前位置，改写已有的字节，然后将 m 的文件指针向前移动 1。在 x 是一个单字符字符串时，m.write_byte(x)类似于 m.write(x)。但是，如果 m 的文件指针位于 m 的末尾，m.write_byte(x)不做任何操作，而 m.write(x)将引发一个 ValueError 异常。请注意，这与对文件末尾进行操作的 read 和 read_byte 方法之间的关系正好是相反的：write 和 read_byte 可以引发 ValueError 异常，而 read 和 write_byte 则不。

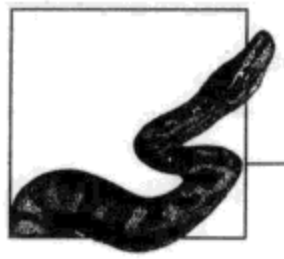
使用 mmap 对象实现进程间通信

使用 mmap 实现进程通信类似于进程间通信（IPC）使用文件进行进程通信的方法：一个进程可以写入数据，而另一个进程在此之后读取相同的数据。由于 mmap 对象基于一个底层文件，因此开发者还可以让某些进程直接对该文件执行 I/O 操作（参见第 10.3 节），而其他进程使用 mmap 访问相同的文件。开发者还可以基于便利的原则选择使用 mmap 和文件对象 I/O 操作：两者的功能是相同的，而且性能也大致相等。例如，下面是一个简单程序，这个程序使用文件 I/O 让一个文件的内容等于用户交互式键入的最后一行文本。

```
fileob = open('xxx','w')
while True:
    data = raw_input('Enter some text:')
    fileob.seek(0)
    fileob.write(data)
    fileob.truncate()
    fileob.flush()
```

接下来是另一个简单程序，与上一个程序在相同的目录下运行，这个程序使用 mmap（以及 time.sleep 函数，参见第 12.1 节中介绍的 sleep 函数）每秒检查一次对文件进行的更改，并打印文件的新内容：

```
import mmap, os, time
mx = mmap.mmap(os.open('xxx',os.O_RDWR), 1)
last = None
while True:
    mx.resize(mx.size())
    data = mx[:]
    if data != last:
        print data
        last = data
    time.sleep(1)
```



第 15 章

数值处理

开发者可以使用运算符（参见第 4.5 节）和内置函数（参见第 8.2 节）执行某些数值计算。Python 还提供了一些模块以支持附加的数值计算功能，本章对这些功能进行了介绍：第 15.1 节介绍了 math 和 cmath 模块，第 15.2 节介绍了 operator 模块，第 15.3 节介绍了 random 模块，第 15.4 节介绍了 decimal 模块。第 15.5 节还介绍了第三方的 gmpy 模块，这个模块进一步扩展了 Python 的数值计算功能。在更特殊的情况下，数值计算通常要求处理数组（array），参见第 16 章。

15.1 math 和 cmath 模块

math 模块提供了对浮点型数字进行操作的数学函数，而 cmath 模块提供了对复数进行计算的数学函数。例如，math.sqrt(−1)将引发一个异常，但是 cmath(−1)将返回 1j。

每个模块都提供了类型为 float，绑定到最基础的数学常数 pi 和 e 的两个属性和如表 15-1 所示的函数。

表 15-1

acos	`acos(x)` 返回 x 的反余弦值，以弧度表示。	math 和 cmath
acosh	`acosh(x)` 返回 x 的反双曲余弦值，以弧度表示。	cmath
asin	`asin(x)` 返回 x 的反正弦值，以弧度表示。	math 和 cmath
asinh	`asinh(x)` 返回 x 的反双曲正弦值，以弧度表示。	cmath
atan	`atan(x)` 返回 x 的反正切值，以弧度表示。	math 和 cmath

续表

atanh	atanh(*x*) 返回 x 的反双曲正切值，以弧度表示。	cmath

atan2	atan2(*y*,*x*)
	相当于 atan(y/x)，区别是 atan2 将正确地考虑两个参数的符号。例如：
	`>>> import math`
	`>>> math.atan(-1./-1.)`
	`0.78539816339744828`
	`>>> math.atan2(-1., -1.)`
	`-2.3561944901923448`
	还有，在 x 等于 0 时，atan2 将返回 pi/2，尽管除以 x 将引发 ZeroDivisionError。
		math

ceil	ceil(*x*) 返回 float(i)，其中 i 是满足 i 大于等于 x 的最小整数。	math
cos	cos(*x*) 返回 x 的余弦值，以弧度表示。	math 和 cmath
cosh	cosh(*x*) 返回 x 的双曲余弦值，以弧度表示。	math 和 cmath
e	数学常数 e。	math 和 cmath
exp	exp(*x*) 返回 e**x。	math 和 cmath
fabs	fabs(*x*) 返回 x 的绝对值。	math
floor	floor(*x*) 返回 float(i)，其中 i 是满足 i 小于等于 x 的最大整数。	math
fmod	fmod(*x*,*y*) 返回浮点型值 r，与 x 具有相同的符号，对于整数 n，满足 r==x-n*y，并且 abs(r)<abs(y)。与 x%y 类似，区别在于当 x 和 y 的符号不一样时，x%y 具有与 y 相同的符号，而不是与 x 相同的符号。	math
frexp	frexp(*x*) 返回一个数据对（m, e），其中包含所谓的“尾数”（实际上是有效数）和 x 的指数。m 是一个浮点型数，而 e 是一个满足 x==m*(2**e) 和 0.5<=abs(m)<1 的整数，除了 frexp(0)返回(0.0, 0）之外。	math
hypot	hypot(*x*,*y*) 返回 sqrt(x*x+y*y)。	math
ldexp	ldexp(*x*,*i*) 返回 x*(2**i) （i 必须是一个整数；如果 i 是一个浮点型数，i 将被截短，但是这种情况将产生一个警告）。	math

续表

log	log(*x*) 返回 x 的自然对数。	math 和 cmath
log10	log10(*x*) 返回 x 的以 10 为底的对数。	math 和 cmath
modf	modf(*x*) 返回一个数据对（f, i），包含 x 的小数和整数部分，表示两个与 x 具有相同符号的浮点型数，并满足 i==int(i) 和 x==f+i。	math
pi	数学常数 π。	math 和 cmath
pow	pow(*x*,*y*) 返回 x**y。	math
sin	sin(*x*) 返回 x 的正弦值，以弧度表示。	math 和 cmath
sinh	sinh(*x*) 返回 x 的双曲正弦值，以弧度表示。	math 和 cmath
sqrt	sqrt(*x*) 发挥 x 的平方根。	math 和 cmath
tan	tan(*x*) 返回 x 的正切值，以弧度表示。	math 和 cmath
tanh	tanh(*x*) 返回 x 的双曲正切值，以弧度表示。	math 和 cmath

15.2 operator 模块

operator 模块提供了等同于 Python 操作符的函数。这些函数在必须保存可调用对象、将可调用对象传递为参数，或者将可调用对象返回为函数结果时是非常方便的。operator 模块中的函数具有与对应的特殊方法相同的名称（参见第 5.2 节）。每个函数都可以使用两个名称，带有或者不带前导和拖尾的双下划线（例如，operator.add(a,b)和 operator._ _add_ _(a,b)都将返回 a+b）。表 15-2 列出了 operator 模块提供的函数。

表 15-2 operator 模块提供的函数

方 法	函 数	表 现 形 式
abs	abs(*a*)	abs(*a*)
add	add(*a*,*b*)	*a*+*b*
and_	and_(*a*,*b*)	*a*&*b*

续表

方　法	函　数	表 现 形 式
concat	concat(*a*,*b*)	*a*+*b*
contains	contains(*a*,*b*)	*b* in *a*
countOf	countOf(*a*,*b*)	*a*.count(*b*)
delitem	delitem(*a*,*b*)	del *a*[*b*]
delslice	delslice(*a*,*b*,*c*)	del *a*[*b*:*c*]
div	div(*a*,*b*)	*a*/*b*
eq	eq(*a*,*b*)	*a*==*b*
floordiv	floordiv(*a*,*b*)	*a*//*b*
ge	ge(*a*,*b*)	*a*>=*b*
getitem	getitem(*a*,*b*)	*a*[*b*]
getslice	getslice(*a*,*b*,*c*)	*a*[*b*:*c*]
gt	gt(*a*,*b*)	*a*>*b*
indexOf	indexOf(*a*,*b*)	*a*.index(*b*)
invert, inv	invert(*a*), inv(*a*)	~*a*
is	is(*a*,*b*)	*a* is *b*
is_not	is_not(*a*,*b*)	*a* is not *b*
le	le(*a*,*b*)	*a*<=*b*
lshift	lshift(*a*,*b*)	*a*<<*b*
lt	lt(*a*,*b*)	*a*<*b*
mod	mod(*a*,*b*)	*a*%*b*
mul	mul(*a*,*b*)	*a***b*
ne	ne(*a*,*b*)	*a*!=*b*
neg	neg(*a*)	−*a*
not_	not_(*a*)	not *a*
or_	or_(*a*,*b*)	*a*|*b*
pos	pos(*a*)	+*a*
repeat	repeat(*a*,*b*)	*a***b*
rshift	rshift(*a*,*b*)	*a*>>*b*
setitem	setitem(*a*,*b*,*c*)	*a*[*b*]=*c*
setslice	setslice(*a*,*b*,*c*,*d*)	*a*[*b*:*c*]=*d*
sub	sub(*a*,*b*)	*a*−*b*
truediv	truediv(*a*,*b*)	*a*/*b* # "true" div -> no truncation
truth	truth(*a*)	not not *a*, bool(*a*)
xor_	xor(*a*,*b*)	*a*^*b*

operator 模块还提供了两个高阶函数，这两个函数的结果是适合于传递为列表的 sort 方法、sorted 内置函数和（在 Python 2.5 中）其他内置函数（比如 mix 和 max）的命名参数 key=的函数。

表 15-3

attrgetter	attrgetter(*attr*) 返回一个可调用函数 f，这样，f(o)就与 getattr(o,attr)相同。

itemgetter	itemgetter(*key*)
	返回一个可调用函数 f，这样，f(o)就与 getitem(o,key)相同。
	例如，假定 L 是一个由列表组成的列表，每个子列表至少包含 3 个项目，如果开发者需要根据每个子列表的 3 个项目在原地对 L 进行排序。最简单的方法是：
	`import operator`
	`L.sort(key=operator.itemgetter(2))`

15.3 随机数和伪随机数

标准 Python 库的 random 模块可以生成满足各种分布的伪随机数。底层的统一伪随机数生成器使用了 Mersenne Twister 算法，以长度 2**19937-1 为周期（Python 的以前版本使用的是 Whichmann-Hill 算法，以长度 6953607871644 为周期。

物理随机数和密码强度的随机数

伪随机数是由 random 模块提供的，尽管非常好，但是不具备密码级的质量。如果开发者需要高质量的随机数（理想上讲，就是自然生成的随机数，而不是使用算法生成的伪随机数），在 Python 2.4 中，可以调用 os.urandom（来自于 os 模块，而不是 random 模块）。

表 15-4

urandom	urandom(*n*) 返回 n 个随机字节，在最新的 Linux 版本上，是从像/dev/urandom 这样的随机比特的物理源读取的，而在 Windows 上，是从像 CryptGenRandom API 这样的密码强度的源读取的。如果当前系统上不存在适当的源，urandom 将引发 NotImplementedError 错误。

要想了解物理随机数的其他源，参见 http://www.fourmilab.ch/hotbits。

random 模块

random 模块的所有函数都是 random.Random 类的一个隐藏的全局实例的方法。开发者可以显式实例化 Random 以获得不共享状态的多个随机数生成器。如果需要在多个线程（参见第 14 章）中使用随机数，显式实例化是明智的。本节将介绍 random 模块提

供的最常使用的函数。

表 15-5

choice	choice(*seq*) 返回一个来自于非空序列 seq 的随机项目。
getrandbits	geTRandbits(*k*) 返回一个具有 k 个随机比特的非负长整数，就像 choice(xrange(2**k))一样（但是要快得多，而且，对于很大的 k，也没有什么问题）。
getstate	getstate() 返回一个表示生成器的当前状态的可哈希、可封装和可排列的对象 S。开发者可以将 S 传递到 setstate 函数以恢复生成器的状态。
jumpahead	jumpahead(*n*) 向前移动生成器的状态，就像已经生成了 n 个随机数一样。计算新状态要比生成 n 个随机数速度更快一些。
random	random() 返回一个服从统一分布的随机浮点数 r，0<=r<1。
randrange	randrange([*start*,]*stop*[,*step*]) 与 choice(xrange(start,stop,step))类似，但是要快得多。
sample	sample(*seq*,*k*) 返回一个新列表，其中的 k 个项目都是从 seq 中随机取出的不同项目。这个列表按随机顺序排序，因此任意切片也相当于一个有效的随机样本。seq 可能会包含重复的项目。在这种情况下，项目的每一个出现都是样本中可供挑选的候选对象，这样，样本也有可能包含重复的项目。
seed	seed(*x*=None) 初始化生成器的状态。x 可以是任意可哈希的对象。在 x 为 None，并且 random 模块第一次自动加载时，seed 将使用当前系统时间（或者其他平台专用的随机源，如果有的话）以获得一个种子。x 通常是一个最大值为 27814431486575L 的长整型数。更大的 x 值也是可以被接受的，但是可能会产生与更小的数相同的生成器状态。
setstate	setstate(*S*) 还原生成器的状态。S 必须是上一个对 getstate 调用的结果（这样的调用可能会出现在另一个程序中，或者出现在这个程序的上一次运行中，只要对象 S 已经被正确地发送，或者被保存和还原）。
shuffle	shuffle(*alist*) 在原地倒换可变序列 alist。
uniform	uniform(*a*,*b*) 返回一个服从统一分布的浮点型随机数 r，a<=r<b。

random 模块还提供了一些可以生成伪随机浮点数的函数，这些伪随机浮点数服从其他概率分布（Beta、Gamma、指数、高斯、柏拉图等），这些函数将通过内部调用 random.random 作为这些分布的随机源。

15.4 decimal 模块

Python 的 float 是一个二进制浮点型数，通常符合被称为 IEEE 754 的标准，并在现代计算机中的硬件中得以实现。要想了解浮点型算术及其相关问题的简单而实际的介绍，可以参见 http://docs.sun.com/source/806-3568/ncg_goldberg.html 上 David Goldberg 写的短文“What Every Computer Scientist Should Know about Floating-Point Arithmetic”。通常，特别对于与钱有关的计算，更适合使用十进制浮点型数字；Python 2.4 在标准库模块 decimal 中为十进制提供了一个被称为 IEEE 854 标准的实现。在 http://docs.python.org/lib/module-decimal.html 上，可以找到完整的参考文档，包含可用的标准、指南和对 decimal 的宣传。本节只介绍了 decimal 功能的一个很小的子集，对应于该模块最常使用的部分。

decimal 模块提供了一个 Decimal 类，其中的不可变实例都是十进制数字、异常类，以及要处理算术环境（arithmetic context）的类和函数，算术环境指定了像精确度、舍入，以及出现时会引发异常的计算错误（比如，被 0 除、上溢出、下溢出等）。在默认环境下，精确度是 28 个十进制数字，舍入是“半-偶（half-even）”（将结果舍入到最近的可表示的十进制数字：在结果就是这两个数字中间的值时，将结果舍入到最后一个数字是偶数的数上），而引发异常的错误都是无效操作、被 0 除和溢出。

要想创建一个十进制数，可以带一个参数调用 decimal.Decimal：这个参数是一个整数或者一个字符串。如果从一个浮点型数 float 开始，则必须向 Decimal 传递该 float 的字符串形式以控制包含的所有数字。例如，decimal.Decimal(0.1)是一个错误；而应该使用 decimal.Decimal('0.1')（可以尝试使用 decimal.Decimal(repr(0.1))了解为什么要这样做，并参见 http://python.org/doc/2.4.2/tut/node16.html 以了解对这个问题的详细解释）。如果开发者愿意，可以很容易地使用 decimal 编写一个工厂函数进行简单的实验，特别是交互式实验。

```
import decimal
def d(x):
    return decimal.Decimal(str(x))
```

现在，d(0.1)与 decimal.Decimal('0.1')执行了相同的操作，但是要简洁得多，写代码时也更方便。

在有了 decimal 实例之后，可以对这些实例（也可以对整数，但是不能对浮点型数）执行算术操作、对这些实例进行封装和拆封、使用这些实例作为字典中的键和集合的成员，并使用与 float 中可用的相同格式化选择来格式化这些实例（使用字符串的%操作符，参见第 9.3 节）；但是，在后一种情况下（也就是说，就像将这些实例传递为 math 模块中的函数的参数一样），decimal 的实例将被转换为 float，这意味着要损失精确度。参见 http://docs.python.org/lib/decimal-recipes.html 了解一些很有用的方法以进行更精确

的格式化和三角法计算。

15.5 gmpy 模块

gmpy 模块（http://gmpy.sourceforge.net）包装了 GMP 库（http://www.swox.com/gmp/）以扩展和加快 Python 的多倍精度算术运算的功能，也就是说，算术运算涉及的精度位数只受到可用内存数量的限制。Python“out of the box”通过内置类型 long 为整数提供了多倍精度算术运算，参见第 4 章；gmpy 提供了另一种名为 mpz 的整数类型，mpz 提供了比 Python 的内置类型 long 甚至更快的操作，还提供了其他一些函数和方法以实现大量不同类型的快速数理计算（斐波纳契数列、阶乘、二项式系数和 probabilistic determination of primality 等）和比特串操作。gmpy 还提供了有理数类型（名为 mqp）、任意精度的浮点型数字类型（名为 mpf）和快速随机数生成器。

gmpy 的参考文档是 gmpy-sources 压缩包中的一部分，可以从 http://sourceforge.net/projects/gmpy/下载；在相同的网页上，还可以找到已经预编译的、可直接安装的 Windows 和 Mac OS X10.4 版本的 Python 下载文件（在编著本书的时候，是 Python 2.3 和 2.4 版本）。不管怎么样，开发者都应该下载并解压缩这个源文件软件包，即使已经安装了一个预编译的版本，因为这些源文件软件包是包含文档的唯一位置（在解开了下载的压缩文件之后，可以在 doc 子目录中找到这些文档）。此外，在 test 子目录中，大约有超过 1 000 个单元测试用例以验证安装是否完全正确（开发者可以在解压缩后的文件所在目录的命令行提示符下运行 python test/gmpy_test.py 以运行所有测试；这些测试只需要花费几秒钟），test 子目录中还有几个计时示例，可以用来显示开发者的特定计算机上 gmpy 类型与 Python 的内置类型之间的性能比较。例如，在作者自己的笔记本电脑上，通过运行 python test/timing2.py，可以看出使用 Python 的内置长整型 long 计算第 100 000 个斐波纳契数字（一个以 259741 开始，总共有 20898 位的数字）需要花费 1.5 秒，使用 gmpy.mpz 实例需要花费 0.5 秒，而调用 gmpy.fib 预定义函数，花费的时间小于 0.01 秒。

gmpy 力求提供启发式和有用的转换功能，可以使用 Stern-Brocot 树从 decimal.Decimal 实例和 float 构建“可感知的”有理数。

```
>>> import gmpy
>>> print gmpy.mpq(0.1)
1/10
```

在这个示例中，gmpy 设法补偿 float 的“表示错误”并生成确切的分数，这个分数是大致推测的，而不是数学上的确切分数，因为这样的确切分数是很少使用的：

```
>>> print gmpy.mpq(int(0.1*2**55),2**55)
3602879701896397/36028797018963968
```

第 16 章

数组处理

开发者可以使用列表（参见第 4.2 节中介绍的列表），以及 array 标准库模块（参见第 16.1 节）来表示数组。可以使用循环、列表推导、迭代器、生成器、生成器表达式（参见第 4 章）、像 map、reduce 和 filter 这样的内置函数（参见第 8.2 节）和像 itertools 这样的标准库模块（参见第 8.11 节）来操作数组。但是，要想处理大的数字数组，这些函数可能会比像 Numeric、numarray 和 numpy 这样的扩展（参见第 16.2 节）更慢，也不如这些扩展使用方便。

16.1 array 模块

array 模块提供了一个类型，也被称为 array，其实例都是可变序列，就像列表一样。array a 是一个一维序列，其中的项目可以只包含字符，或者只包含某种特定数值类型的数字，这是在创建 a 的时候选定的。

与列表相比，array.array 的主要好处是可以节省内存，因为可以保存所有相同类型的数据（数值或字符）。array 对象 a 有一个单字符只读属性 a.typecode，这个属性是在创建的时候设置的，给出了 a 的项目的类型。表 16-1 显示了 array 可用的类型代码。

表 16-1 array 模块的类型代码

类型代码	C 语言类型	Python 类型	最小大小
'c'	char	str (长度为 1)	1 字节
'b'	char	int	1 字节
'B'	unsigned char	int	1 字节
'U'	unicode char	unicode (长度为 1)	2 字节
'h'	Short	int	2 字节

续表

类型代码	C语言类型	Python类型	最小大小
'H'	unsigned short	int	2字节
'i'	int	int	2字节
'I'	unsigned	long	2字节
'l'	long	int	4字节
'L'	unsigned long	long	4字节
'f'	float	float	4字节
'd'	double	float	8字节

每个项目的字节大小可能会比最小大小更大，这取决于计算机的体系结构，字节大小可以用作只读属性a.itemsize。array模块只提供了名为array的类型对象。

表16-2

array	`array(typecode,init='')` 使用给定的typecode创建并返回一个array对象a。init可以是一个普通字符串，其长度是itemsize的倍数；被翻译为机器值的字符串字节可以直接初始化a的项目。另外，init可以是任何可迭代对象（在typecode为'c'时，这个可迭代对象是字符，否则就是数字）：可迭代对象中的每个项目都用来初始化a中的一个项目。 数组对象提供了可变序列（参见第4.6节）中除sort方法之外的所有方法和操作。使用+或+=进行串联和为切片复制的操作要求两个操作数都是具有相同类型代码的数组；与此相反，a.extend的参数可以是任意可迭代对象，只要其中的项目都可以被a接受。除了可变序列的方法，array对象a还提供了以下方法。
byteswap	`a.byteswap()` 交换a的每个项目的字节序。
fromfile	`a.fromfile(f,n)` 从文件对象f中读取n个项目，将其作为机器值，并将这些项目添加到a的末尾。请注意，f必须以二进制模式的可读模式打开——例如，似乎用模式'rb'。在f中只有少于n个项目可用时，fromfile将在添加了可用的项目之后引发EOFError错误。
fromlist	`a.fromlist(L)` 将列表L中的所有项目都添加到a的末尾。
fromstring	`a.fromstring(s)` 将字符串s的字节（被翻译为机器值）添加到a的末尾。len(s)必须是a.itemsize的确切倍数。
tofile	`a.tofile(f)` 将a中的所有项目以机器值的形式写入到文件对象f中。请注意，f必须以二进制模式的可读模式打开——例如，使用模式'wb'。

续表

tolist	*a*.tolist() 创建并返回一个与 a 具有相同项目的列表对象，与 list(a)类似。
tostring	*a*.tostring() 返回包含 a 中的所有项目的字节的字符串，这些字节被翻译为机器值。对于任意 a，len(a.tostring( ))== len(a)*a.itemsize. f.write(a.tostring( ))与 a.tofile(f)相同。

16.2 数值数组计算的扩展包

从网站 http://sourceforge.net/project/showfiles.php?group_id=1369 上，开发者可以免费下载 3 个相互兼容的扩展包：Numeric、Numarray 和 NumPy。每个包都可以作为源代码（很容易在许多平台上编译和安装）或者作为 Windows 上已经编译好的自安装.exe 文件使用；该网站上还有其他一些已经编译好的形式，比如 Linux 上的.rpm 文件，或者 Apple Mac OS X 上的.dmg 文件。从相同的网站上，还可以下载大量有关 Numeric 的指南，并找到到其他资源的链接，比如 bug 跟踪者、邮件列表和 Python 科学计算主页（http://numeric.scipy.org/）。

每个扩展包的重点都在于处理比较大的数字数组，这些数组通常都是多维的（比如矩阵）。许多辅助模块对高级计算提供了高性能地支持，比如线性代数、快速傅立叶变换和图像处理，其中的一部分模块是由扩展包本身附带的，而其他一些模块可以从其他网站单独下载。每个扩展包都是一个很大的，功能丰富的软件包。要想更完全地了解这些软件包，需要研究指南，演示示例和交互式实验。本章在假定开发者已经对数组运算和数值计算问题有所了解的基础上，提供了 Numeric 最基础的子集的一些参考资料。如果开发者还不熟悉这些主题，可以参考 Numeric 指南。

Numeric 不再处于活跃开发状态；对其用户而言，Numeric 被广泛认为是“稳定的”，而对于其批评者而言，认为 Numeric 是很“老”的。numarray 要更新一些，功能也更丰富，现在仍处于活跃开发状态，并在其主网站 http://www.stsci.edu/resources/software_hardware/numarray 上提供了很好的文档和技术支持，在这个网站上还可以找到大量优秀文档的链接。NumPy 是最新的，功能也最丰富的，并且处于非常活跃的开发状态（在编著本书的时候，还远未到达稳定的 1.0 版本）；在将来 NumPy 成熟后，开发者可以很有信心地期待 NumPy 将取代前两个扩展包，并成为最主要的数值数组计算的 Python 扩展包。在本书编著期间，http://numeric.scipy.org/ 上链接的 NumPy 文档是需要收费的，其收益用来支持 NumPy 的开发、展示和未来的发展。开发者可以期望这些文档最终将会成为免费的。

本书选择介绍 Numeric，这个软件包是稳定的，也很好执行，对于许多应用程序都是非常有用的。学习 Numeric 对于掌握其他可供选择的软件包也是非常有用的，Numeric 具

有高度的相互兼容性。要想了解 Python 中支持数值和科学计算的大量软件包，可以参见 http://scipy.org。在第 2.2 节中提到的 Python 的 Enthought 版本附带了大量可供选择的这种软件包。

16.3 Numeric 包

Numeric 包中的主要模块是 Numeric 模块，这个模块提供了 array 类型、可以作为数组实例的函数，以及可以对数组和其他序列进行操作的所谓“通用函数”。Numeric 是通常与习惯用法 from Numeric import * 一起使用的少数几个 Python 包之一，尽管在这种情况下，这种习惯用法甚至偶尔会出现一些问题。另外一种流行的方法就是使用一个短名称导入 Numeric（例如，import Numeric as N），并在每个名称前面加上 N 以让该名称有效，这可能是在简明和清晰之间最好的折衷方法。

16.4 数组对象

Numeric 提供了 array 类型以表示一个由项目组成的矩阵。array 对象 a 具有给定数量的维数，被称为数组对象的秩（rank），可以是某个任意上限（通常是 30，在使用默认选项编译 Numeric 时）。标量（也就是，单个数字）的秩为 0，向量的秩为 1，矩阵的秩为 2，依此类推。

类型代码

array 对象的网格单元中的值也被称为数组的元素（element），这些元素都是同类型的，表示所有元素都具有相同的类型，并且所有元素值都保持在一个内存区域。这是与列表相比较而言的，在列表中，项目可能是不同的类型，并且每个项目都被保存为单独的 Python 对象。这意味着 Numeric 数组要比具有相同数量项目的 Python 列表占用少得多的内存。a 的元素的类型被编码为 a 的类型代码，也就是一个单字符字符串，参见表 16-3。用于构建 array 实例的工厂函数（参见第 16.4 节）需要用到一个 typecode 参数，这个参数是表 16-3 中的一个值。

表 16-3 Numeric 数组的类型代码

类型代码	C 语言类型	Python 类型	同义字
'c'	char	str (长度为 1)	Character
'b'	unsigned char	int	UnsignedInt8
'1'	signed char	int	Int8
's'	short	int	Int16
'w'	unsigned short	int	UnsignedInt16

续表

类 型 代 码	C 语言类型	Python 类型	同 义 字
'i'	int	int	Int32
'u'	unsigned	int	UnsignedInt32
'l'	long	int	Int
'f'	float	float	Float32
'F'	Two floats	complex	Complex32
'd'	double	float	Float
'D'	Two doubles	complex	Complex
'O'	PyObject*	any	PyObject

Numeric 为每个类型代码提供了可读的属性名称，如表 16-3 的最后一列所示。Numeric 还在所有平台上提供了名称 Int0、Float0、Float8、Float16、Float64、Complex0、Complex8、Complex16 和 Complex64。在每种情况下，名称表示请求的种类的最小类型和最小位数。例如，Float8 是至少为 8 位的最小浮点类型（通常与 Float0 和 Float32 相同，但是可能在某些平台上，从理论上讲，可以提供非常小的浮点类型），而 Complex0 是最小的复数类型。在某些平台上，Numeric 还提供了名称 Int64、Int128、Float128 和 Complex128，这些名称具有类似的含义。并不是所有平台都能提供这些名称的，因为并非所有平台都能提供有这么多位的数字。类型代码为‘0’表示这些元素都是对 Python 对象的引用。在这种情况下，元素可以具有不同的类型。这样，开发者可以将 Numeric 数组对象用作 Python 容器，这些 Python 容器可以用于与数值处理无关的数组处理任务。

在使用 Numeric 的某个工厂函数建立数组 a 时，可以显式指定 d 的类型代码或者接受一个默认的数据相关类型代码。要想获得数组 a 的类型代码，可以调用 a.typecode( )。a 的类型代码可以确定 a 中的每个项目在内存中占用多少个字节。调用 a.itemsize( )可以获得这个信息。在类型代码为‘0’时，项目大小是最小的（例如，在 32-位平台上是 4 字节），但是这个大小计数只用于 a 的每个单元中保存的引用。这些引用指示的对象都以单独的 Python 对象的形式保存在其他位置；根据对象的类型，每个这样的对象可能会占用任意数量的额外内存，而这些并没有计算在类型代码为‘0’的数组的项目大小中。

形状和索引

每个 array 对象 a 都有一个属性 a.shape，这个属性是一个由整数组成的元组。len(a.shape)是 a 的秩；例如，由数字组成的一维数组（也被称为向量）的秩为 1，而 a.shape 只有一个项目。更常见的是，a.shape 中的每个项目都是 a 的对应维数的长度。a 的元素数也被称为 a 的大小，是 a.shape 中的所有项目的积。a 的每个维数也被称为一个轴。轴的索引从 0 开始增加，在 Python 中通常如此。Python 中还允许有负的轴索引，并从右边

开始计数，因此，-1 就是最后一个（最右边的）轴。

每个数组 a 都是一个 Python 序列。a 中的每个项目 a[i]都是 a 的一个子数组，这意味着 a[i]是一个秩比 a 的秩小 1 的数组：a[i].shape==a.shape[1:]。例如，如果 a 是一个二维矩阵（a 的秩为 2），对于任意有效索引 i，a[i]是 a 的一个一维子数组，对应于该矩阵的一行。在 a 的值为 1 或 0 时，a 中的项目就是 a 的元素（对于秩为 0 的数组，只包含一个元素）。由于 a 是一个序列，开发者可以使用常规索引语法来索引 a 以访问或更改 a 中的项目。请注意，a 中的项目也都是 a 的子数组；只是对于秩为 1 或 0 的数组，数组的项目也就是该数组中的元素。

开发者还可以在一个 for 语句中对 a 执行循环操作，就像可以对任何其他序列所作的操作那样。例如：

```
for x in a:
    process(x)
```

与下面的代码具有相同的功能：

```
for i in range(len(a)):
    x = a[i]
    process(x)
```

在这些示例中，for 循环中 a 的每个项目 x 都是 a 的一个子数组。例如，如果 a 是一个二维矩阵，这两个循环操作中的每个 x 都是 a 的一个一维子数组，对应于该矩阵的一行。

开发者还可以使用一个元组来索引 a。例如，如果 a 的秩至少为 2，对于任意有效的 i 和 j，可以将 a[i][j]写作 a[i,j]以进行重新绑定或者进行访问。元组索引要更快，也更便利。不要在方括号中放置圆括号以表示使用元组来索引 a：只需要一个接一个地列出索引号，并以逗号分隔即可。a[i, j]与 a[(i, j)]相同，但是不包含圆括号的形式更自然，也更具可读性。

如果索引的结果是单个数字，Numeric 有时候会将结果表示为一个秩为 0 的数组，而有时候将结果表示为一个具有适当 Python 类型的标量值。换句话讲，作为索引的结果，开发者有时候会得到一个只包含一个数字的数组，而有时候会得到其中包含的数字。例如，考察下面这个代码片段：

```
>>> for t in 'blswiufFdDO': print t, type(Numeric.array([0],t)[0])
```

有些令人惊奇的输出结果是：

```
b <type 'array'>
l <type 'int'>
s <type 'array'>
w <type 'array'>
i <type 'int'>
u <type 'array'>
f <type 'array'>
```

```
F <type 'array'>
d <type 'float'>
D <type 'complex'>
O <type 'int'>
```

这个输出结果显示，对于单个结果的索引，确切对应于一个 Python 数字类型的数组类型将产生 Python 数字，而其他数组类型将产生秩为 0 的数组。

存储

array 对象 a 通常被保存在一个连续的内存区域中，其中的项目按照传统上被称为“以行为主序（row-major order）”的顺序一个接一个地排列。例如，在 a 的秩为 2 时，a 的第一行 a[0]中的元素将首先开始，接着是 a 的第二行 a[1]中的元素，依此类推。

当一个数组共享一个更大的数组中的一部分存储空间时，这个数组可以是非邻近的，参见第 16.4 节。例如，在 a 的秩为 2 时，切片 b=a[:,0]是 a 的第一列，并被非连续地存储，因为该列占用了 a 的某些相同的存储空间。b[0]将占用与 a[0, 0]相同的存储空间，而 b[1]将占用与 a[1, 0]相同的存储空间，在 a 包含多于一个的列时，a[1, 0]不能邻接到 a[0, 0]占用的内存上。

在大多数情况下，Numeric 可以透明处理连续存储和非连续存储数组，这样就可以使用最自然的方案，而不会浪费内存，也不需要可以避免的副本了。本章的其余部分将介绍需要连续存储数组的位置产生的少数异常。在开发者想要将一个非连续存储数组 b 复制到一个新的连续存储数组 c 中时，可以使用 copy 方法，参见第 16.4 节。

切片

数组可以与其他数组共享某些或全部数据。只要有可能，Numeric 将在数组之间共享数据。如果开发者想要使用 Numeric 复制数据，可以显式请求一个副本。Numeric 的数据共享还可以应用于切片。对于内置 Python 列表和标准库数组对象，切片就是（浅的）副本，但对于 Numeric.array 对象，切片将与被切片的数组共享数据：

```
from Numeric import *
alist=range(10)
list_slice=alist[3:7]
list_slice[2]=22
print list_slice, alist        # 打印: [3,4,22,6] [0,1,2,3,4,5,6,7,8,9]
anarray=array(alist)
arr_slice=anarray[3:7]
arr_slice[2]=33
print arr_slice, anarray       # 打印: [3 4 33 6] [0 1 2 3 4 33 6 7 8 9]
```

重新绑定 list_slice 中的一个项目不会影响 alist 列表（list_slice 是 alist 列表的切片），因为对于内置列表而言，切片将执行一个复制操作。但是，因为对于 Numeric 数组而言，切片将共享数据，为 arr_slice 中的一个项目赋值确实会影响 array 对象 anarray，arr_slice

是 anarray 的切片。这种方式并不是初学者所期望的，但是这样做可以获得高性能。

切片示例

开发者可以使用元组对数组进行切片，就像可以使用元组来索引数组一样：对于数组来说，切片和索引是相互融合的。切片元组中的每个项目都可以是一个整数，并且，对于每个这样的项目而言，切片要比被切片的数组少一个轴：通过选择数组的指定平面，切片可以删除给出数字的那个轴。

切片元组的项目还可以是一个切片表达式；通用语法是 start:stop:step，还可以省略这 3 个部分中的一个或多个（要想详细了解切片语法和默认值，参见第 4.6 节和第 8.1 节中介绍的 slice 类型）。下面是切片的一些示例：

```
# a 是 [[ 0, 1, 2, 3, 4, 5],
#       [10,11,12,13,14,15],
#       [20,21,22,23,24,25],
#       [30,31,32,33,34,35],
#       [40,41,42,43,44,45],
#       [50,51,52,53,54,55]]
a[0,2:4]                        # array([2,3])
a[3:,3:]                        # array([[33,34,35],
                                #        [43,44,45],
                                #        [53,54,55]])
a[:,4]                          # array([4,14,24,34,44,54])
a[2::2,::2]                     # array([[20,22,24],
                                #        [40,42,44]])
```

切片元组项目还可以使用省略号（…）以指示切片元组中后面的项目将应用到切片数组的最后（最右边）的轴上。例如，考察对一个秩为 3 数组 b 进行切片：

```
b.shape                         # (4,2,3)
b[1].shape                      # (2,3)
b[...,1].shape                  # (4,2)
```

在使用 b[1]进行切片时（相当于索引），可以为轴 0 给定一个整数索引，这样就选择了一个沿着 b 的轴 0 开始的特定平面。通过选择一个特定平面，可以从结果的形状中删除这个轴。这样，结果的形状就是 b.shape[1:]。在使用 b[...,1]进行切片时，将沿着 b 的轴-1（b 的最右边的轴）选择一个特定平面。再次需要说明的是，通过选择一个特定平面，可以从结果的形状中删除这个轴。因此，结果的形状就是 b.shape[:−1]。

切片元组项目还可以是伪索引 NewAxis，这是 Numeric 模块提供的一个常数。结果切片在使用 NewAxis 的位置有一个附加轴，在形状元组的对应项目中，NewAxis 的值为 1。继续前一个示例：

```
b[Numeric.NewAxis,...,Numeric.NewAxis].shape      # (1,4,2,3,1)
```

上面的代码添加了两个新轴，一个在形状的开始，另一个在末尾，而不是进行选择并

因此删除了 b 的某些轴，这要感谢省略号。

轴的删除和添加可以同时发生在同一个切片操作中。例如：

```
b[Numeric.NewAxis,:,0,:,Numeric.NewAxis].shape    # (1,4,3,1)
```

上面的代码在形状的开始和末尾都添加了一个新轴，并通过为 b 的中间轴（轴 1）给定一个索引而为该轴选择了一个特定索引。因此，b 的轴 1 将从结果的形状中被删除。在这个实例中，被用作切片元组中的第二个和第四个项目的冒号是省略了 start 和 stop 的切片表达式，这表示所有相应的轴都被包含在切片中。在所有这些示例中，所有切片都共享 b 的某些或所有数据。切片只影响结果数组的形状，并没有复制数据，也没有对数据执行任何操作。

为数组切片赋值

为数组切片赋值不如为列表切片赋值那么灵活。通常，开发者可以为一个数组切片赋的唯一值就是与该切片具有相同形状的另一个数组。但是，如果赋值语句的右边（RHS）不是一个数组，Numeric 将为该 RHS 创建一个临时数组。RHS 中的每个元素都被强制转换为左边（LHS）类型。如果 RHS 数组与 LHS 切片的形状不同，将会应用广播（broadcasting），参见第 16.4 节。例如，开发者可以为一个数值数组的任意切片赋一个标量（表示单个数字）：RHS 数字是强制的，然后按照需要进行广播（复制）以使得赋值成功。

在将一个类型不同于 LHS 类型的 RHS 赋值给一个数组切片（或索引）时，Numeric 将把 RHS 的值强制为 LHS 的类型——例如，通过将浮点型数字截短为整型。如果 RHS 的值为复数，无法应用这种强制操作。不能对原地操作符应用完全的强制操作，原地操作符只能向上转换 RHS 的值（例如，整型 RHS 适用于使用浮点型 LHS 的原地操作，但是反之并不如此），参见第 16.4 节中介绍的原地操作。

真值和数组比较

尽管 array 对象 a 是一个 Python 序列，a 并不遵循 Python 为序列真值定义的常规规则（序列在空时真值为 False；否则序列的真值为 True）。而是在 a 没有元素或者在 a 的所有元素都为 0 时，a 的真值为 False。因为数组之间的比较也会产生数组（其中的项目为 0 或 1），Numeric 的规则对于按自然方式测试数组中元素的相等性而言是必不可少的：

```
if a==b:
```

如果没有这个限制条件，任何非空可比数组 a 和 b 都会满足这样的 if 条件。尽管有这个规则，数组比较仍然是非常麻烦的，因为只有在任何一个对应元素都相等时，两个数组的比较结果才为 True：

```
print bool(Numeric.array([1,2])==Numeric.array([1,9]))  # 打印 True (!)
```

Numeric 的 alltrue 和 sometrue 函数（参见表 16-4）提供了一种更好的方法来表达这样的比较操作；本书建议开发者不要依赖于 if a==b 这种容易令人混淆的操作，而是尽可能让自己的意图更清楚，并且使用 if Numeric.alltrue(a==b)或者 if Numeric.sometrue(a==b)这样的编码来显式比较。

请记住，不管怎么样，开发者在想要测试 a 中是否包含任何项目，或者测试 a 中是否包含任何元素时，必须进行显式处理，因为两者是不同的条件：

```
a = Numeric.array( [ [  ], [  ], [  ] ] )
if a: print 'a is true'
else: print 'a is false'                    # 打印: a is false
print bool(Numeric.alltrue(a))              # 打印: False
print bool(Numeric.sometrue(a))             # 打印: False
if len(a): print 'a has some items'
else: print 'a has no items'                # 打印: a has some items
if Numeric.size(a): print 'a has some elements'
else: print 'a has no elements'             # 打印: a has no elements
```

但是，在大多数情况下，比较数字数组的最好方法是使用 Numeric 的 allclose 函数进行约等于比较，参见第 16.4 节中 allclose 函数。

工厂函数

Numeric 提供了几个可以创建 array 对象的工厂函数。

表 16-4

array, asarray	array(*data*,*typecode*=None,*copy*=true,*savespace*=False) asarray(*data*,*typecode*=None,*savespace*=False) 返回一个新 array 对象 a。a 的形状取决于 data。在 data 是一个数字时，a 的秩为 0，而 a.shape 为空白元组（）。在 data 是一个数字序列时，a 的秩为 1，并且 a.shape 为单独的元组（len(data),）。在 data 是一个由数字序列组成的序列时，data 的所有项目（序列）必须具有相同的长度，a 的秩为 2，并且 a.shape 是数据对（len(data),len(data[0])）。这个方法可以推广到 data 的任意嵌套级别，也就是序列的序列，最大可以到达本章前面提到的秩的上限。如果 data 的嵌套级别超出了这个限制，array 将引发 TypeError 错误（实际上不太可能出现超出限制的问题：对于一个秩为 32 的数组，如果每个轴的长度为 2，该数组将包含超过 10 亿个元素）。 typecode 可以是表 16-3 中显示的任意值，或者为 None。在 typecode 为 None 时，array 将根据 data 中的元素的类型选择一个类型代码。在 data 中只有一个或者多个元素是长整型 long，或者既不是数字也不是普通字符串时（例如，None 或者 Unicode 字符串），typecode 为‘O’，也被称为 PyObject。在所有元素都是普通字符串时，类型代码为字符。在任何一个或者多个元素（但不是所有的）是普通字符串，所有其他元素都是数字（没有一个是长整型 long），并且 typecode 为 None 时，array 将引发 TypeError 错误。如果开发者想要使用 array 从某些普通字符串和 int 或 float 构建一个数组，必须将‘O’或 PyObject 显式传递为 typecode 参数的值。在所有元素都是数字（没有一个元素为长整型时），

array, asarray	类型代码取决于元素中“最宽的”数值类型。在任何一个元素为复数（complex）时，类型代码为 Complex。在没有元素为复数，但是某些或全部元素都是浮点型时，类型代码为 Float。在所有元素都是整型时，类型代码为 Int。
	在默认情况下，array 函数将返回一个 array 对象 a，该对象不与任何其他对象共享数据。如果 data 是一个 array 类型，并且开发者为参数 copy 显式传递了一个 False 值，array 将返回一个与 data 共享数据的 array 对象 a。asarray 函数就像 copy 参数的值被传递为 Flase 的 array 函数一样。
	在默认情况下，在与更宽的数值类型的数字进行运算时，数值 array 将被隐式转换为更宽的数值类型。在开发者不希望出现这样的隐式转换时，可以通过为 array 工厂函数的 savespace 参数显式传递一个 True 值来节省一些内存，因为这样可以将结果 array 对象 a 设置为节省空间模式。例如：
	`array(range(4),typecode='b')+2.0`
	`# array([2.,3.,4.,5.])`
	`array(range(4),typecode='b',savespace=True)+2.0`
	`# array([2,3,4,5])`
	`array(range(4),typecode='b',savespace=True)+258.7`
	`# array([2,3,4,5])`
	第一条语句创建了一个由浮点型值组成的数组；由于这条语句没有指定 savespace，因此在加上 2.0 时，数组中的每个元素将被隐式转换为浮点型。第二条和第三条语句分别创建了一个由 8 位整数组成的数组；由于这两条语句指定了 savespace，因此将把添加到每个元素的浮点型值隐式向下转换为 8 位整数，而不是将数组中的元素隐式向上转换为浮点型。258.7 将被向下转换为 2；由于要将 258.7 转换为一个整数，因此小数部分.7 将被丢弃，而结果 258 变成 2 是因为在转换为 8 位整数时，只有最低 8 位被保存。savespace 模式对大数组是非常有用的，但是在使用 savespace 模式时应该非常小心，以免遇到不可预料的精度损失。

arrayrange, arange	`arrayrange([start,]stop[,step=1],typecode=None)` 与 array(range(start,stop,step),typecode)类似，但是要更快一些（参见内置函数 range 以了解 start、stop 和 step 的详细信息）。aarayrangge 允许这些参数都是浮点型，而不仅仅是整型。在使用这种功能时，要非常小心，因为浮点型算术运算可能导致一个比开发者期望的多一个或者少一个项目的结果。arange 是 arrayrange 的同义字。
fromstring	`fromstring(data,count=None,typecode=Int)` 返回形状（count,）的一个一维数组 a，其中包含从 data 字符串的字节复制的数据。在 count 为 None 时，len(data)必须是 typecode 的项目大小的倍数，并且 a 的形状是（len(data)/a.itemsize( ),）。在 count 不为 None 时，len(data)必须大于或等于 count*a.itemsize( )，并且 formstring 将忽略 data 的拖尾字节，如果有的话。 连同 a.tostring 和 a.byteswapped 方法（参见第 16.4 节）一起，fromstring 函数允许使用 array 对象的二进制 I/O。在开发者需要保存数组并在以后重新加载这些数组，并且不需要在非 Python 程序中使用保存的形式时，使用 cPickle 模块要更简单，也更快一些，参见第 11.1 节。许多有经验的用户更愿意使用可移植的、自描述的文件格式，比如 netCDF（参见 http://met-www.cit.cornell.edu/noon/ncmodule.html）。

续表

identity	identity(*n*,*typecode*=Int) 返回形状为（n, n）的一个二维数组 a（矩形矩阵）。除了 a 的主对角线上那些值为 1 的元素（a[j, j]，j 属于 range(n)）之外，a 中其他元素的值为 0。
empty	empty(*shapetuple*,*typecode*=Int,*savespace*=False) 返回一个 a.shape==shapetuple 的数组 a。a 中的元素没有被初始化，因此元素的值完全是任意的（就像在允许“未初始化变量”的其他语言中一样，并且不同于 Python 中的所有其他情况）。
ones	ones(*shapetuple*,*typecode*=Int,*savespace*=False) 返回一个 a.shape==shapetuple 的数组 a。a 中的所有元素都为 1。

zeros	zeros(*shapetuple*,*typecode*=Int,*savespace*=False)
	返回一个 a.shape==shapetuple 的数组 a。a 中的所有元素都为 0。
	在默认情况下，identity、ones 和 zeros 都将返回类型为 Int 的数组。如果想要使用不同的类型代码，比如 Float，可以显式传递该类型代码。一个常见的错误是：
	`a = zeros(3)`
	`a[0] = 0.3  # a 为 array([0,0,0])`
	因为在上面这个代码片段中，a 是 Int 类型，在将 0.3 赋值给 a 中的一个项目时，这个值将被截短为整数 0。因此，通常需要使用类似于下面这样的代码：
	`a = zeros(3, Float)`
	`a[0] = 0.3  # a 为 array([0.3,0.,0.])`
	在上面这个代码片段中，显式指定了 Float 作为 a 的类型代码，因此在将 0.3 赋值给 a 中的一个项目时，不会出现截短操作。

属性和方法

对于大多数数组运算，Numeric 提供了一些可以对数组参数进行调用的函数，参见第 16.4 节。参数还可以是 Python 列表；这种多态性提供了比被包装为数组属性和方法的功能更多的灵活性。每个 array 对象 a 还提供了一些方法和属性以直接（稍微更快一些）访问可能不需要多态性的功能。

表 16-5

astype	*a*.astype(*typecode*) 返回一个与 a 具有相同的形状的新数组 b。b 中的元素就是将 a 中的元素强制转换为 typecode 指定的类型的元素。b 不会共享 a 中的数据，即使 typecode 等于 a.typecode( )。
byteswapped	*a*.byteswapped() 返回一个与 a 具有相同类型代码和形状的新数组对象 b。b 中的每个元素都是从对应的 a 中的元素复制而来，但是反转了元素值的字节顺序。这样的交换操作可以将每个元素的值从小端模式转换为大端模式，反之亦然。结合 fromstring 函数和 a.tostring 方法，这种交换操作可以在开发者从一种类型的计算机上获得了二进制数据，并需要将这些数据用于另一种类型的计算机时使用。

续表

byteswapped	例如，2005 年销售的全部 Apple Mac 计算机使用的都是 PowerPC 的 CPU，这种 CPU 使用的是大端模式，而新的 Mac 计算机使用了 Intel 的 CPU，而这种 CPU 使用的是小端模式；byteswapped 可以帮助开发者在一台新的 Mac 计算机上读取使用老式 Mac 计算机写入的二进制文件，反之亦然。
copy	*a*. copy() 返回一个新的连续存储数组对象 b，b 与 a 相同，但是不共享 a 中的数据。
flat	a.flat 是一个属性，表示秩为 1 的数组，具有与 a 相同的大小，并共享 a 中的数据。索引或切片 a.flat 可以帮助开发者通过 a 的这种间隔视图方式访问或更改 a 的元素。在 a 不是连续存储时，尝试访问 a.flat 将引发 TypeError 异常。在 a 为连续存储时，a.flat 将按以行为主序顺序排列。例如，在 a 的形状为 (7, 4) 时（也就是，a 是一个包含 7 个行和 4 个列的二维矩阵），对于所有 range(28) 范围内的所有 I，a.flat[I]与 a[divmod(I,4)]相同。
Imag, imaginary, real	尝试访问 a.imag 将引发 ValueError 异常，除非 a 的类型代码为复数；在这种情况下，a.real 是一个具有与 a 相同形状和类型代码的数组，并且与 a 共享数据。在 a 的类型代码为复数时，a.real 和 a.imag 是与 a 具有相同形状和浮点型类型代码的非连续存储数组，并与 a 共享数据。访问或修改 a.real 或 a.imag 可以访问或修改 a 中的复数元素的实部和虚部。Imaginary 是 imag 的同义字。
Iscontiguous	*a*. iscontiguous() 如果 a 的数据占用了连续的存储空间，则返回 True；否则，返回 False。在与 C 语言编码的扩展进行接口时，这种连续性是特别重要的。A.copy( )可以创建 a 的一个连续存储副本。在切片或置换数组时，将产生非连续存储的数组，而且具有复数类型代码的数组 a 的 a.real 和 a.imag 属性也是非连续存储的数组。
Itemsize	*a*. itemsize() 返回 a 中的每个元素使用的内存的字节数量（尽管这个方法的名称叫项目大小；但这里并不是指 a 的每个项目，而是元素，在通常情况下，项目表示 a 的子数组）。
savespace	*a*. savespace(*flag*=True) 根据 flag 设置或重新设置数组 a 的节省空间模式。在 flag 为 True 时，a.savespace(flag)将设置 a 的节省空间模式，因此在与更宽的数值类型运算时，a 中的元素并不会隐式向上转换（要想相关的详细信息，参见第 16.1 节中介绍的 array 函数的 savespace 参数）。在 flag 为 False 时，a.savespace(flag)将重新设置 a 的节省空间模式，这样，在需要时，a 中的元素将隐式向上转换。
shape	a.shape 是一个元组，a 的每个轴一个项目，每个项目给出了轴的长度。开发者可以将一个整数序列赋值给 a.shape 以更改 a 的形状，但是 a 的大小（元素的总数）必须保持相同。在开发者将序列 s 赋值给 a.shape 时，s 中的一个项目可以是-1，表示不管怎么样都需要这个轴的长度以保持 a 的大小不变。s 的其他项目的乘积必须平分 a 的大小，否则，改变形状将引发一个异常。在需要更改 a 中的元素的总数时，可以调用 resize 函数（参见第 16.4 节中介绍的 resize 函数）。
spacesaver	*a*. spacesaver() 如果数组 a 采用了节省空间模式，则返回 True；否则，返回 False。参见本节前面介绍的 savesapce 方法。

续表

tolist	*a*.tolist()
	返回一个等同于 a 的列表 L。例如，如果 a.shape 是（2, 3），a 的类型代码为‘d’，L 是两个由 3 个浮点型值组成的列表之一，在每个列表中，对于每个有效 i 和 j，L[i][j]==a[i,j]。list(a)只将数组 a 的顶层（轴 0）转换为一个列表，因此，如果 a 的秩为 2 或更大的值，则 list(a)不等于 a.tolist( )。例如：
	`a=array([[1,2,3],[4,5,6]],typecode='d')`
	`print a.shape  # 打印: (2,3)`
	`print a  # 打印: [[1. 2. 3.]`
	`#  [4. 5. 6.]]`
	`print list(a)`
	`# 打印: [array([1.,2.,3.]), array([4.,5.,6.])]`
	`print a.tolist()`
	`# 打印: [[1.0,2.0,3.0],[4.0,5.0,6.0]]`

toscalar	*a*.toscalar() 返回 a 的第一个元素作为一个适当类型的 Python 标量（通常是一个数字），这个类型取决于 a 的类型代码。
tostring	*a*.tostring() 返回一个二进制字符串 s，s 的字节是 a 中的元素组成的字节的一个副本。
typecode	*a*.typecode() 以单字符字符串的形式返回 a 的类型代码。

数组运算

算术运算符+、−、*、/、//、%和**；比较运算符>、>=、<、<=、==和!=；和按位运算符&、|、^和～（参见第 4.5 节）还可以应用于数组。如果操作数 a 和 b 都是具有相同形状和类型代码的数组，其结果是一个具有相同形状（和相同类型代码，除了比较运算符）的新数组 c。c 中的每个元素都是对 a 和 b 的对应元素执行运算的结果（按元素运算）。在类型代码为复数的数组之间是不允许进行顺序比较的，就像不允许对复数进行比较一样；在所有其他情况下，数组之间的比较运算符将返回具有整型类型代码的数组。

数组并不遵循*（复制）和+（串联）的序列语义：*和+执行按元素算术运算。与此类似，*并不意味着矩阵相乘，但是按元素相乘。Numeric 提供了一些函数以执行复制、串联和矩阵相乘；数组的所有运算符都是按元素运算的。

在 a 和 b 的类型代码不同时，更窄的数值类型将被转换为更宽的类型，就像其他 Python 数值运算一样。数值和非数值的值之间的运算是不允许的。在数组情况下，开发者可以使用 savespace 方法将数组设置为节省空间模式，这样可以限制类型转换。要非常小心地使用节省空间模式，因为这种模式可能会导致无意中丢失重要的数据。要想了解有关节省空间模式的详细信息，参见第 16.1 节中介绍的 array 函数的 savespace 参数。

广播

不同形状的数组之间的按元素运算通常是不可能的：试图执行这样的运算将引发一个异常。在可能的情况下，Numeric 允许执行某些这样的运算，也就是对较小的数组进行广播（复制）以将其扩大到较大的形状。要想让广播更有效率，复制只是概念性的：Numeric 不会实际复制数据（也就是说，开发者不需要担心因为某个运算包含了广播操作而导致性能会降低）。

广播最简单和最常见的情况就是在一个操作数 a 是一个标量（或者是一个秩为 0 的数组），而另一个操作数 b 是任意数组时。在这种情况下，Numeric 将从概念上建立一个形状为 b.shape 的临时数组 t，t 中的每个元素都等于 a。然后，Numeric 将执行 t 和 b 之间请求的运算操作。因此，实际上，在对一个包含标量 a 的数组进行运算时（比如 a+b 或 b-a），结果数组与 a 具有相同的形状，并且每个元素都是对 b 的相应元素与单个数字 a 应用运算符的结果。

更一般的情况是，还可以在操作数 a 和 b 都是数组时使用广播。从概念上讲，可以根据以下这些相当复杂的一般规则来使用广播：

- 在 a 和 b 的秩不同时，形状元组较短的数组可以通过添加前导轴（每个轴的长度为 1）来填充，以达到另一个元组的秩。
- 按照第一个规则填充到相同长度的 a.shape 和 b.shape 是从右边开始比较的（也就是从最后一个轴的长度开始）。
- 在 a 和 b 中正在比较的轴的轴长度相等时，表示这个轴是有效的，广播操作将向左移动到前一个轴以进行比较。
- 在轴的长度不同，并且都大于 1 时，Numeric 将引发一个异常。
- 在一个轴长度为 1 时，Numeric 可以通过沿着这个平面复制到另一个数组的轴长度以广播相应的数组。

因为这种一般性，广播的规则都是很复杂的，但是广播的大多数典型应用程序都是很简单的。例如，假定计算 a+b，并且 a.shape 为（5, 3）（一个包含 5 行和 3 列的矩阵）。b.shape 的典型值包括（）（一个标量）、(3,)（一个包含 3 个元素的一维向量）和（5, 1）（一个包含 5 行和 1 列的矩阵）。在每种情况下，b 将沿着需要的轴（在 b 为标量时，两个轴都需要）复制 b 的元素以从概念上广播到一个形状为（5, 3）的临时数组 t，并且 Numeric 将计算 a+t。当然，最简单和最常见的情况是在 b.shape 为（5, 3）时，也就是在与 a 具有相同的形状时。在这种情况下，不需要进行广播。

原地运算

数组可以通过增量赋值运算符（+=、-=等）来提供原地运算。运算符左边（LHS）的

数组或切片不能进行广播，但是右边（RHS）是可以的。类似的，LHS 不能向上转换，但是 RHS 可以。换句话讲，原地运算将把 LHS 看作是形状和类型都严格定义的，但是 RHS 服从常规的、更宽松的规则。

函数

Numeric 定义了几个可以对数组进行操作，或者以多态的形式对 Python 序列进行操作的函数，这些函数将从概念上从非数组操作数形成临时数组。

表 16-6

allclose	`allclose(x,y,rtol=1.e-5,atol=1.e-8)`
	返回单个数字：在 x 中的每个元素与 y 中的对应元素近似相等时，返回 0；否则，返回 1。如果满足以下条件，可以将 ex 和 ey 这两个元素定义为近似相等：
	`abs(ex-ey) < atol + rtol*abs(ey)`
	换句话讲，如果 ex 和 ey 都非常小（小于 atol），或者相对差非常小（小于 rtol），则 ex 和 ey 近似相等。通常，allclose 是一种比==更好的检查数组是否相等的方法，因为浮点型算术运算要求一定的比较容限。但是，allclose 不适合于复数数组，只能用于浮点型和整型数组。要想比较两个复数数组 x 和 y 是否近似相等，可以使用：
	`allclose(x.real, y.real) and allclose(x.imag, y.imag)`

argmax, argmin	`argmax(a,axis=-1) argmin(a,axis=-1)` argmax 将返回一个新整型数组 m，其形状元组就是 a.shape 减去指定的 axis。m 中的每个元素都是沿着 axis 的 a 中的最大元素的索引。argmin 与此类似，但是指示的是最小的元素，而不是最大的元素。

argsort	`argsort(a,axis=-1)`
	返回一个与 a 具有相同形状的新整型数组 m。沿着 axis 的 m 的每个向量都是对 a 的对应轴进行排序所需的索引序列。特别是，如果 a 的秩为 1，则 take(a,argsort(a))==sort(a)，这也是最常见的情况。例如：
	`x = [52, 115, 99, 111, 114, 101, 97, 110, 100, 55]`
	`print Numeric.argsort(x)`
	`# 打印: [0 9 6 2 8 5 7 3 4 1]`
	`print Numeric.sort(x)`
	`# 打印: [52 55 97 99 100 101 110 111 114 115]`
	`print Numeric.take(x, Numeric.argsort(x))`
	`# 打印: [52 55 97 99 100 101 110 111 114 115]`
	在上面的示例中，Numeric.argsort(x)的结果显示，x 的最小元素是 x[0]，第二小的元素是 x[9]，第三小的元素是 x[6]，依此类推。最后一个 print 语句中，在调用 Numeric.take 时将按这个顺序取得 x 中的元素，这样将产生与第二个 print 语句中调用 Numeric.sort 产生的相同排序数组。

around	`around(a,decimals=0)` 返回一个与 a 具有相同形状的新浮点型数组 m。m 中的每个元素与对 a 中的对应元素调用 Python 的内置函数 round 的结果相似。

array2string	array2string(*a*,*max_line_width*=77,*precision*=8,
	suppress_small=False,*separator*=' ', *array_output*=False)
	返回数组 a 的字符串表示法 s，其中的元素包含在方括号中，以字符串 separator 分隔。最后一维是水平的，倒数第二个是垂直的，并使用方括号嵌套其他的维。在 array_output 为 True 时，s 将以 'array(' 开始，以 ')' 结束，或者，在 X 是 a 的类型代码，并且 X 不是 Float、Complex 和 Int 时，以 ",'X')" 结束（因此，如果 separator 为 ','，以后还可以使用 eval(s)）。
	比 max_line_width 更长的行将被分开。precision 确定了每个元素显示多少个数字。如果 suppress_small 为 True，非常小的数字将被显示为 0。要想更改默认值，可以设置 sys 模块中名为 output_line_width、float_output_precision 和 float_output_suppress_small 的属性。例如：
	`>>> Numeric.array2string(Numeric.array([1e-20]*3))`
	`'[ 1.00000000e-20  1.00000000e-20  1.00000000e-20]'`
	`>>> import sys`
	`>>> sys.float_output_suppress_small = True`
	`>>> Numeric.array2string(Numeric.array([1e-20]*3))`
	`'[ 0.  0.  0.]'`
	str(a) 类似于 array2string(a)。repr(a) 类似于 array2string(*a*, separator=',', array_output=True)。开发者还可以使用 Numeric 模块中的名称 array_repr 和 array_str 访问这些格式化函数。

average	average(*a*,*axis*=0,*weights*=None,*returned*=False) 返回 a 的沿着 axis 的平均值。在 axis 为 None 时，返回 a 的所有元素的平均值。在 weights 不为 None 时，weights 必须是一个与 a 具有相同形状的数组，或者一个具有 a 的给定 axis 的长度的一维数组，而 average 将计算加权平均值。在 returned 为 True 时，将返回一个数据对：第一个项目是平均值；第二个项目是权重的和（在 weights 为 None 时，第二个项目是值的数量）。

choose	choose(*a*,*values*)
	返回一个与 a 具有相同形状的数组 c。values 是任意 Python 序列。a 中的元素是 0（包括 0）和 len(values)（不包括）之间的整数。c 中的每个元素都是 values 中的项目，这些项目的索引是 a 中的对应元素。例如：
	`print Numeric.choose(Numeric.identity(3),'ox')`
	`# 打印: [[x o o]`
	`#  [o x o]`
	`#  [o o x]]`

clip	clip(*a*,*min*,*max*)
	返回一个与 a 具有相同类型代码和形状的数组 c。c 中的每个元素 ec 都是 a 中的对应元素 ea，其中 min<=ea<=max。在 ea<min 时，ec 是 min；在 ea>max 时，ec 是 max。例如：
	`print Numeric.clip(Numeric.arange(10),2,7)`
	`# 打印: [2 2 2 3 4 5 6 7 7 7]`

续表

compress	compress(*condition,a,axis*=0)
	返回一个与 a 具有相同类型代码和秩的数组 c。c 只包含 a 中可以让 condition 为 True 的项目，并且这些项目对应于 a 中沿着给定的 axis 的项目。例如，compress((1,0,1),a) == take(a,(0,2),0)，因为 (1,0,1) 只在索引 0 和 2 上为 True 值。下面是一个如何从数组中只提取偶数的示例：
	`a = Numeric.arange(10)`
	`print Numeric.compress(a%2==0, a)  # 打印: [0 2 4 6 8]`

concatenate	concatenate(*arrays, axis*=0)
	arrays 是一个由数组组成的序列，所有数组具有相同的形状，除非有可能沿着给定的 axis。concatenate 将返回一个连接沿着给定 axis 的 arrays 的数组。如果 s 是一个常规 Python 序列，而不是一个数组，concatenate((s,)*n)具有与 s*n 相同的序列复制语义。例如：
	`print Numeric.concatenate([Numeric.arange(5), Numeric.arange(3)])`
	`# 打印: [0 1 2 3 4 0 1 2]`

convolve	convolve(*a,b,mode*=2)
	返回一个秩为 1 的数组 c，也就是秩为 1 的数组 a 和 b 的线性卷积。线性卷积是在无界序列的基础上定义的。convolve 可以通过填充 0 将 a 和 b 从概念上扩展到无限长度，然后将无限长度的结果剪切为其中央部分，从而生成 c。在 mode 为 2，也就是默认值时，convolve 只剪切填充值，因此 c 的形状为(len(a)+len(b)−1,)。否则，convolve 将剪切更多数据。假定 len(a)大于或等于 len(b)。在 mode 为 0 时，len(c)等于 len(a)-len(b)+1；在 mode 为 1 时，len(c)等于 len(a)。在 len(a)小于 len(b)时，将产生对称的效果。例如：
	`a = Numeric.arange(6)`
	`b = Numeric.arange(4)`
	`print Numeric.convolve(a, b)`
	`# 打印: [0 0 1 4 10 16 22 22 15]`
	`print Numeric.convolve(a, b, 1)`
	`# 打印: [0 1 4 10 16 22]`
	`print Numeric.convolve(a, b, 0)`
	`# 打印: [4 10 16]`

cross_correlate	cross_correlate(*a,b,mode*=0) 就像 convolve(a,b[::-1],mode)一样。

diagonal	diagonal(*a,k*=0,*axis1*=0,*axis2*=1)
	返回 a 中沿 axis1 轴和沿 axis2 轴的索引相差 k 的元素。在 a 的秩为 2 时，如果 k==0 时，就是主对角线，如果 k > 0，则是主对角线之上的子对角线，如果 k < 0，则是主对角线之下的子对角线。例如：
	`# a 为 [[ 0  1  2  3]`
	`#       [ 4  5  6  7]`
	`#       [ 8  9 10 11]`
	`#       [12 13 14 15]]`
	`print Numeric.diagonal(a)        # 打印: [0 5 10 15]`

续表

diagonal	`print Numeric.diagonal(a,1)` `# 打印: [1 6 11]`
	`print Numeric.diagonal(a,-1)` `# 打印: [4 9 14]`
	可以看出，diagonal(a)是主对角线、diagonal(a, 1)是主对角线上面一个子对角线，而 diagonal(a,-1)是主对角线下面一个子对角线。

dot	`dot(a,b)` 返回 a 和 b 按矩阵相乘，而不是按元素相乘的结果数组 m。a.shape[-1]必须等于 b.shape[-2]，并且 m.shape 等于元组 a.shape[:-1]+b.shape[:-2]+b.shape[-1:]。

indices	`indices(shapetuple,typecode=None)`
	返回一个由形状(len(shapetuple),)+shapetuple 组成的整型数组。子数组 x[i]中的每个元素都等于该子数组中该元素的索引 i。例如：
	`print Numeric.indices((2,4))` `# 打印: [[[0 0 0 0]`
	`#  [1 1 1 1]]`
	`#  [[0 1 2 3]`
	`#  [0 1 2 3]]]`

innerproduct	`innerproduct(a,b)` 返回一个数组 m，其中包含 a 和 b 的内积的结果，相当于 matrixmultiply(a,transpose(b))。a.shape[-1]必须等于 b.shape[-1]，并且 m.shape 等于元组 a.shape[:-1]+b.shape[0:-1:-1]。
matrixmultiply	`matrixmultiply(a,b)` 返回 a 和 b 按矩阵相乘，而不是按元素相乘的结果数组 m。a.shape[-1]必须等于 b.shape[0]，并且 m.shape 等于元组 a.shape[:-1]+b.shape[1:]。

nonzero	`nonzero(a)`
	返回 a 中不等于 0 的元素的索引，相当于下面的表达式：
	`array([i for i in range(len(a)) if a[i] != 0])`
	a 必须是一个序列或者一个向量（也就是一个一维数组）。

outerproduct	`outerproduct(a,b)` 返回一个数组 m，m 是向量 a 和 b 的外积（换句话讲，对于每个有效索引对 i 和 j，m[i,j]等于 a[i]*b[j]）。

put	`put(a,indices,values)`
	a 必须是一个连续数组。indices 是一个整数序列，被看作 a.flat 的索引。values 是一个由可以被转换为 a 的类型代码的值组成的序列（如果这个序列比 indices 短，values 可以根据需要进行重复）。a 中的每个由 indices 中的一个项目指定的元素将被替换为 values 中的对应项目。因此，put 类似于下面的循环操作（但是要快得多）：
	`for i,v in zip(indices,list(values)*len(indices)):`
	`  a.flat[i]=v`

putmask	`putmask(a,mask,values)` a 必须是一个连续数组。mask 是一个与 a.flat 具有相同长度的序列。values 是一个由可以被转换为 a 的类型代码的值组成的序列（如果这个

续表

putmask	序列比 mask 短，values 可以根据需要进行重复)。a 中的每个对应于 mask 中值为 True 的项目的元素都将被替换为 values 中的对应项目。因此，putmask 类似于下面的循环操作（但是更快）：
	`for i,v in zip(xrange(len(mask)),list(values)*len(mask)):`
	`  if mask[i]: a.flat[i]=v`

rank	`rank(a)` 返回 a 的秩，就像 len(array(a,copy=False).shape)一样。
ravel	`ravel(a)` 返回 a 的平面形式，就像 array(a, copy=not a.iscontiguous( ))).flat 一样。

repeat	`repeat(a,repeat,axis=0)`
	返回一个与 a 具有相同类型代码和秩的数组，其中 a 的每个元素都沿着 axis 重复 repeat 中的对应项目的值指定的次数。repeat 是一个整数，或者是一个长度为 a.shape[axis]的整型序列。例如：
	`>>>print N.repeat(range(4),range(4)) # 打印：[1 2 2 3 3 3]`

reshape	`reshape(a,shapetuple)`
	返回一个形状为 shapetuple 的数组 r，并共享 a 中的数据。r=reshape(a,shapetuple)类似于 r=a;r.shape=shapetuple。shapetuple 中的项目的乘积必须等于 a.shape 中的项目的乘积；shapetuple 中的一个项目还有可能为-1，这表示请求数组 r 适应轴的长度。例如：
	`print Numeric.reshape(range(12),(3,-1))`
	`# 打印: [[0 1 2 3]`
	`#  [4 5 6 7]`
	`#  [8 9 10 11]]`

resize	`resize(a,shapetuple)`
	返回一个形状为 shapetuple 的数组 r，并且 r 中的数据是从 a 复制的。如果 r 的大小小于 a 的大小，r.flat 将从 ravel(a)的开始复制；如果 r 的大小更大，则根据需要多次复制 ravel(a)中的数据。特别是，如果 s 是一个常规 Python 序列，而不是一个数组，则 resize(s,(n*len(s),))具有与 s*n 相同的序列复制语义。例如：
	`print Numeric.resize(range(5),(3,4))`
	`# 打印: [[0 1 2 3]`
	`#  [4 0 1 2]`
	`#  [3 4 0 1]]`

searchsorted	`searchsorted(a,values)`
	a 必须是一个排好序的秩为 1 的数组。searchsorted 可以返回一个与 values 具有相同形状的整型数组 s。s 中的每个元素都是 a 中的索引，且 values 的对应元素符合 a 的排序顺序。例如：
	`print Numeric.searchsorted([0,1],`
	`[0.2,-0.3, 0.5,1.3,1.0,0.0,0.3])`
	`# 打印: [1 0 1 2 1 0 1]`

searchsorted	这种特殊的习惯用法将返回一个数组，其中：0 对应于 values 中的每个小于或等于 0 的元素 x；1 对应于大于 0 且小于或等于 1 的元素 x；2 对应于大于 1 的元素 x。通过简单的一般化，并使用适当的阈值作为排序数组 a 中的元素，这种习惯用法可以对 values 的每个元素 x 归属的子范围进行非常快速的分类。
shape	`shape(a)` 返回 a 的形状，就像 array(a,copy=False).shape 一样。
size	`size(a,axis=None)` 在 axis 为 None 时，将返回 a 中的元素的总数。否则，返回 a 中沿着 axis 的元素的数量，与 array(a,copy=False).shape[axis]类似。

sort	`sort(a,axis=-1)`
	返回一个与 a 具有相同类型代码和形状的数组 s，沿着 axis 的每个平面中的元素都被重新排序，这样，该屏幕将按递增顺序排序。例如：
	`# x 为 [[0 1 2 3]`
	`#  [4 0 1 2]`
	`#  [3 4 0 1]]`
	`print Numeric.sort(x)  # 打印: [[0 1 2 3]`
	`#  [0 1 2 4]`
	`#  [0 1 3 4]]`
	`print Numeric.sort(x,0)  # 打印: [[0 0 0 1]`
	`#  [3 1 1 2]`
	`#  [4 4 2 3]]`
	在上面的代码中，sort(x)将对每个行进行排序，而 sort(x, 0)将对每个列进行排序。

swapaxes	`swapaxes(a,axis1,axis2)`
	返回一个与 a 具有相同类型代码、秩和大小的数组 s，并共享 a 中的数据。s 的形状与 a 相同，但是轴 axis1 和 axis2 的长度被交换了。换句话讲，s=swapaxes(a,axis1,axis2)相当于：
	`swapped_shape=range(length(a.shape))`
	`swapped_shape[axis1]= axis2`
	`swapped_shape[axis2]= axis1`
	`s=transpose(a,swapped_shape)`

take	`take(a,indices,axis=0)` 返回一个与 a 具有相同类型代码和秩的数组 t，t 是 a 的元素的一个子集，t 中的元素包含在沿着 axis 并由给定 indices 组成的切片中。例如，在执行 t=take(a,(1,3))之后，t.shape==(2,)+a.shape[1:]，并且 t 中的元素是 a 的第 2 行和第 4 行元素的副本。
trace	`trace(a,k=0)` 返回 a 的 k 对角线上的元素的总和，与 sum(diagonal(a,k))类似。

续表

transpose	transpose(*a*,*axes*=None) 返回一个与 a 具有相同类型代码、秩和大小的数组 t，并共享 a 中的数据。t 的轴将根据 a 的轴，并按照序列 axes 中的轴索引进行改变。在 axes 为 None 时，t 的轴转换 a 的轴的顺序，axes 就像是 reversed(a.shape)一样。
vdot	vdot(*a*,*b*) 返回一个标量，这个标量是向量 a 和 b 的点积。如果 a 为复数，这个运算将使用 a 的复数共轭。
where	where(*condition*,*x*,*y*) 返回一个与 condition 具有相同形状的数组 w。在 condition 为 True 的元素所在的位置上，w 中的对应元素就是 x 中的对应元素；否则就是 y 中的对应元素。例如，clip(a,min,max)相当于 where(greater(a,max),max,where(greater(a,min),a,min))。

16.5 通用函数（ufuncs）

Numeric 提供了与 Python 的算术、比较和按位运算符具有相同语义的命名函数，还提供了与内置模块 math 和 cmath（参见第 15.1 节）提供的类似的数学函数，比如 sin、cos、log 和 exp。

这些函数都是类型为 ufunc（代表“通用函数”）的对象，除了数组运算符中的常用函数（按元素运算、广播和强制）之外，这些函数还有几个共同特性。每个 ufunc 实例 u 都是可调用的，适用于序列和数组，并且可以接受一个可选的 output 参数。如果 u 是二元的（也就是，如果 u 可以接受两个操作数参数），u 还具有 4 个可调用属性，名为 u.accumulate、u.outer、u.reduce 和 u.reduceat。Numeric 提供的 ufunc 对象只能应用于具有数值类型代码的数组（也就是，不能用于类型代码为‘0’或‘c’的数组）和 Python 数字序列。

在对列表 L 开始操作时，直接对 L 调用 u 要比将 L 转换为一个数组更快一些。u 的返回值是一个数组 a；如有需要，开发者可以对 a 执行进一步计算；如果需要一个列表结果，可以最后调用 tolist 方法将结果数组转换为一个列表。例如，假定开发者需要计算一个列表中的每个项目的对数，并返回另一个列表。在作者的笔记本电脑上，在 N 被设置为 2222 时，像下面这样的列表推导：

```
def logsupto(N):
    return [math.log(x) for x in range(2,N)]
```

需要花费 5.2 毫秒。使用 Python 的内置 map：

```
def logsupto(N):
    return map(math.log, range(2,N))
```

速度要更快一些，大约 3.7 毫秒。使用 Numeric 的名为 log 的 ufunc：

```
def logsupto(N):
    return Numeric.log(Numeric.arange(2,N)).tolist()
```

将把时间减少到 2.1 毫秒。如果更细心地利用 log ufunc 的 output 参数：

```
def logsupto(N):
    temp = Numeric.arange(2, N, typecode=Numeric.Float)
    Numeric.log(temp, output=temp)
    return temp.tolist()
```

会进一步将时间减少到只要 2 毫秒。这样，几乎不费什么力就可以获得加速这种简单但是大计算量运算的能力，这是 Numeric 吸引人的一个好的方面，也是 Numeric 的 ufunc 对象的特殊之处。请一定要小心，不要随意编写下面这样的代码：

```
def logsupto(N):
    return Numeric.log(range(2,N)).tolist()
```

使用这样的代码，在作者的笔记本电脑上需要花费 18 毫秒；很清楚，在这种情况下，从列表到数组和从整数到浮点型的转换操作占用了主要的实际计算时间。

可选的 output 参数

任意 ufunc u 都可以接受一个可选参数 output，output 参数指定了一个输出数组。如果提供了这个参数，output 必须是一个数组，或者一个具有与 u 的结果相同形状和类型的数组切片（不强制，不广播）。u 将在 output 中保持结果，并且不会创建一个新数组。output 可以与 u 的输入数组参数 a 相同。实际上，指定 output 通常是为了使用更快的等式（比如 u(a, b, a)）来替代像 a=u(a, b)这样的通用习惯用法。但是，如果 output 不成为 a，是不能与 a 共享数据的（也就是说，output 不能是 a 的某些或所有数据的不同视图）。如果开发者传递了这样一个不被允许的 output 参数，Numeric 通常无法诊断错误并引发一个异常，因此，开发者可能会得到错误的结果。

不管开发者是否传递可选的 output 参数，ufunc u 将返回其结果以作为函数的返回值。在不传递 output 时，u 将在一个新 array 对象中保持其返回的结果，因此开发者通常可以将 u 的返回值绑定到某个引用上，这样可以以后再访问 u 的结果。在传递 output 参数时，u 将在 output 中保持这个结果，因此开发者不需要绑定 u 的返回值。开发者可以以后再访问 u 的结果，这个结果就是被传递为 output 的 array 对象的新内容。

可调用属性

每个二元 ufunc u 提供了 4 个也被称为可调用对象的属性。

表 16-7

accumulate	*u*.accumulate(*a*,*axis*=0)
	返回一个具有与 a 相同形状和类型代码的数组 r。r 中的每个元素都是使用 u 的底层函数或操作符对沿着给定 axis 的 a 中元素进行累加的结果。例如：
	`print add.accumulate(range(10))`
	`# 打印: [0 1 3 6 10 15 21 28 36 45]`
	因为 add 的底层运算符是+，并且 a 为序列 0,1,2,…,9，r 为序列 0,0+1,0+1+2,…,0+1+…+8+9。换句话讲，r[0]为 a[0]、r[1]为 r[0]+a[1]、r[2]为 r[1]+a[2]，依此类推（对于每个 i>0，r[i] 等于 r[i-1]+a[i]）。

outer	*u*.outer(*a*,*b*)
	返回一个形状元组为 a.shape+b.shape 的数组 r。对于每个索引 a 的元组 ta 和索引 b 的元组 tb，a[ta]将与 b[tb]进行运算（使用 u 的底层函数或运算符），并被放到 r[ta+tb]中（这里，+ 表示元组串联）。在 u 为 multiply 时，全部操作在数学上被称为外积。例如：
	`a = Numeric.arange(3, 5)`
	`b = Numeric.arange(1, 6)`
	`c = Numeric.multiply.outer(a, b)`
	`print a.shape, b.shape, c.shape`
	`# 打印: (2,) (5,) (2,5)`
	`print c`
	`# 打印: [[3 6 9 12 15]`
	`#  [4 8 12 16 20]]`
	c.shape 为（2, 5），也就是操作数 a 和 b 的形状元组的串联。c 的每个第 i 行都是整个 b 乘以 a 的对应第 i 个元素。

reduce	*u*.reduce(*a*,*axis*=0) 返回一个与 a 具有相同类型代码，并且秩比 a 的秩少 1 的数组 r。r 中的每个元素都是 a 中沿着给定 axis 的元素的缩减，并使用 u 的底层函数或运算符。因此，u.reduce 的功能接近于 Python 的内置 reduce 函数。例如，因为 0+1+2+…+9 的值为 45，add.reduce(range(10))为 45。使用内置 reduce 和 import operator，reduce(operator.add,range(10))的值也是 45，就像更简单和更快的表达式 sum(range(10))一样。

reduceat	*u*.reduceat(*a*,*indices*)
	返回一个与 a 具有相同类型代码，与 indices 具有相同类型的数组 r。r 中的每个元素都是从 indices 中的对应项目开始直到下一个项目，下一个项目除外（对于最后一个项目，直到最后一个元素）的缩减，并使用 u 的底层函数或运算符。例如：
	`print Numeric.add.reduceat(range(10),(2,6,8))`
	`# 输出: [14 13 17]`
	在这行代码中，r 中的元素是部分和 2+3+4+5、6+7 和 8+9。

Numeric 提供的 ufunc 对象

Numeric 提供了一些 ufunc 对象，如表 16-8 所示。

表 16-8　Numeric 提供的 ufunc 对象

ufunc	行　为
absolute	类似于 abs 内置函数
add	类似于 + 运算符
arccos	类似于 math 和 cmath 中的 acos 函数
arccosh	类似于 cmath 中的 acosh 函数
arcsin	类似于 math 和 cmath 中的 asin 函数
arcsinh	类似于 cmath 中的 asinh 函数
arctan	类似于 math 和 cmath 中的 atan 函数
arctanh	类似于 cmath 中的 atanh 函数
bitwise_and	类似于 & 运算符
bitwise_not	类似于～运算符
bitwise_or	类似于 | 运算符
bitwise_xor	类似于 ^ 运算符
ceil	类似于 math 中的 ceil 函数
conjugate	每个元素的复数共轭（一元）
cos	类似于 cos math 和 cmath 中的函数
cosh	类似于 cosh cmath 中的函数
divide	类似于 / 运算符 （在除以 0 时，结果为 inf）
divide_safe	类似于 / 运算符 （在除以 0 时，引发一个异常）
equal	类似于 == 运算符
exp	类似于 math 和 cmath 中的 exp 函数
fabs	类似于 math 中的 fabs 函数
floor	类似于 math 中的 floor 函数
fmod	类似于 math 中的 fmod 函数
greater	类似于 > 运算符
greater_equal	类似于>= 运算符
less	类似于< 运算符

续表

ufunc	行　为
less_equal	类似于<= 运算符
log	类似于 math 和 cmath 中的 log 函数
log10	类似于 math 和 cmath 中的 log10 函数
logical_and	类似于 & 运算符；返回由 0 和 1 组成的数组
logical_not	类似于～运算符；返回由 0 和 1 组成的数组
logical_or	类似于 | 运算符；返回由 0 和 1 组成的数组
logical_xor	类似于 ^ 运算符；返回由 0 和 1 组成的数组
maximum	按元素计算两个元素中较大的元素
minimum	按元素计算两个元素中较小的元素
multiply	类似于 * 运算符
not_equal	类似于 != 运算符
power	类似于 ** 运算符
remainder	类似于 % 运算符
sin	类似于 sin math 和 cmath 中的函数
sinh	类似于 sinh cmath 中的函数
sqrt	类似于 sqrt math 和 cmath 中的函数
subtract	类似于 - 运算符
tan	类似于 tan math 和 cmath 中的函数
tanh	类似于 tanh cmath 中的函数

下面是一个如何使用 ufunc 获得一个“斜面”数字，也就是先降后升的数字的例子：

```
print Numeric.maximum(range(1,20),range(20,1,-1))
# 打印: [20 19 18 17 16 15 14 13 12 11 11 12 13 14 15 16 17 18 19]
```

常用 ufunc 方法的简写

Numeric 为 ufunc 对象的某些常用方法定义了函数同义字，如表 16-9 所示。

表 16-9　ufunc 方法的同义字

同　义　字	代表的方法
alltrue	logical_and.reduce
cumproduct	multiply.accumulate

续表

同 义 字	代表的方法
cumsum	add.accumulate
product	multiply.reduce
sometrue	logical_or.reduce
sum	add.reduce

16.6 辅助 Numeric 模块

还有其他一些模块是建立在 Numeric 之上或者与其一起使用的。这些模块中的一部分包含在 Numeric 包中，模块的文档也是 Numeric 的文档的一部分。开发者可以在网站 http://www.scipy.org 上找到一个可以与 Numeric 一起使用的，功能强大的科学和工程计算工具集合；如果开发者打算使用 Python 实现任何类型的科学或工程计算，可以参考这个工具集合。

下面是 Numeric 附带的主要辅助模块。

MLab

MLab 提供了许多 Python 函数，是在 Numeric 的基础上编写的，但是其名称和操作类似于 Matlab 产品提供的函数。

FFT

FFT 提供了可以对 Numeric 数组中保持的数据执行的 Python 可调用“快速傅立叶变换”（Fast Fourier Transforms，FFT）。FFT 可以包装著名的 FFTPACK Fortran 语言编码的库或兼容的 C 语言编码的 fftpack 库，这个库是 FFT 附带的。

LinearAlgebra

LinearAlgebra 提供了 Python 可调用函数，这些函数可以对 Numeric 数组中保持的数据进行操作，可以包装 LAPACK Fortran 语言编码的库或兼容的 C 语言编码的 lapack_lite 库。LinearAlgebra 可以用来反转矩阵、求解线性系统、计算特征值和特征向量、执行奇异值分解和最小二乘法求解超定线性系统。

RandomArray

RandomArray 提供了快速、高质量的伪随机数生成器以建立具有各种随机分布的 Numeric 数组。

MA

MA 提供了屏蔽（masked）数组（也就是，可以包含缺少或者无效值的数组）。

MA 提供了 Numeric 的功能的一个很大子集，虽然有时候速度会降低。MA 还可以将一个可选的 mask 与每个数组关联，mask 是一个由布尔型值组成的辅助数组，其中 True 表示缺少、未知或无效的数组元素。计算可以传播屏蔽；开发者可以通过为无效元素提供一个填充值以将屏蔽的数组转变成普通的 Numeric 数组。MA 可以广泛使用，因为实际数据通常包含缺少或不适用的元素。如果开发者需要为自己的应用程序扩展或者特殊化 Numeric 某些方面的操作，最简单和最有效的方法就是从 MA 的源代码开始，而不是从 Numeric 的源代码开始。由于多年来对 Numeric 应用的优化程度不同，Numeric 通常是非常难以理解和修改的。

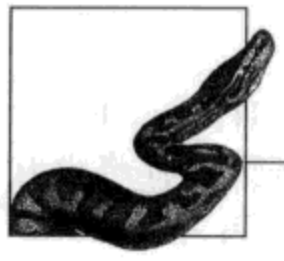
这些模块的性能通常是非常好的。例如：

```
import RandomArray
x = RandomArray.random((13,23))
```

在作者的笔记本电脑上，这个操作需要花费 109 毫秒。而纯的 Python 代码：

```
from random import random
x = [[random() for i in xrange(13)] for j in xrange(23)]
```

需要花费 230 毫秒，几乎两倍的时间长度。



第 17 章

Tkinter GUI

大多数专业的客户端应用程序与用户之间都是通过图形用户界面(GUI)进行互操作的。GUI 通过一个工具包（toolkit）进行编程，工具包是一个提供了控件（control）（也被称为部件（widget)），以及像按钮、标签、文本输入框和菜单这样的可视化对象的库。GUI 工具包可以将控件组合成一个有机的整体，在屏幕上显示这些控件，并与用户进行交互，通过键盘和鼠标接收输入信息。

Python 允许开发者在多个 GUI 工具包中进行选择。有些工具包是平台专用的，但是大多数是跨平台的，至少支持 Windows 和类 UNIX 平台，通常也支持 Mac。网站 http://wiki.python.org/moin/GuiProgramming 上列出了几十个可以用于 Python 的 GUI 工具包。当前最流行的 Python GUI 工具可能就是 wxPython（http://www.wxpython.org/）了，但是 Python 本身发布的一个工具包是 Tkinter。

Tkinter 是一个面向对象的 Python 包装器，包装了跨平台工具包 Tk，Tk 还可以与其他脚本语言一起使用，比如 Tcl（Tcl 最初就是为 Tk 而开发的）、Ruby 和 Perl。Tkinter 就像其底层的 Tcl/Tk，可以在 Windows、Macintosh 和类 UNIX 平台上运行。在 Windows 上，标准 Python 发布版本还包括了 Tkinter 本身和运行 Tkinter 所需的 Tcl/Tk 库。在其他平台上，开发者必须单独获取并安装 Tcl/Tk（根据不同 Python 发布版本的详细信息，开发者可能需要在安装了 Tcl/Tk 之后再安装或者重新安装 Python）。

本章介绍了 Tkinter 最基础的子集，不过，这些内容就足以让 Python 应用程序建立简单的图形前端了（更完整的文档可以参见 http://docs.python.org/lib/tkinter.html）。本章中的所有脚本都是可以单独运行的（也就是说，从一个命令行运行，或者按平台相关的方法运行，比如双击一个脚本的图标）。从另一个自身就带有 GUI 的程序（比如 Python 集成开发环境，例如 IDLE 或 PythonWin）内部运行 GUI 脚本通常可能会导致异常。这在 GUI 脚本尝试终止（并因此关闭 GUI）时特别成问题，因为脚本的 GUI 和开发环境的 GUI 可能会相互干扰。

17.1 Tkinter 基础知识

Tkinter 可以用来很容易地创建简单的 GUI 应用程序。开发者只需要导入 Tkinter、创建、配置和放置想要的部件，然后进入 Tkinter 主循环就可以了。开发者的应用程序将成为事件驱动的：用户与部件的交互操作会产生事件，应用程序将通过被安装为这些事件的处理程序的函数对事件进行响应。

下面的示例显示了一个可以展现这种通用结构的简单应用程序：

```
import sys, Tkinter
Tkinter.Label(text="Welcome!").pack()
Tkinter.Button(text="Exit", command=sys.exit).pack()
Tkinter.mainloop()
```

调用 Label 和 Button 可以分别创建部件，并将其返回为结果。因为没有指定父窗口，Tkinter 将把这些部件直接放在应用程序的主窗口上。命名参数指定了每个部件的配置。在这个简单的示例中，不需要将变量绑定到部件。开发者只需要对每个部件调用 pack 方法即可，这样将使用被称为包装器的布局管理器对象处理对部件布局的控制。布局管理器是一个不可视的组件，其作用就是在其他部件（也就是容器或父部件）中放置部件，并处理几何布局问题。上面这个示例没有传递参数来控制包装器的操作，这样会让包装器按默认方式操作。

在用户单击按钮时，Button 部件的 command 可调用函数将不带参数执行。上面的示例在创建 Button 时将 sys.exit 函数传递为名为 command 的参数。因此，当用户单击该按钮时，将会执行 sys.exit( )并终止该应用程序（参见第 8.3 节中介绍的 exit 函数）。

在创建并包装了部件之后，这个示例调用了 Tkinter 的 mainloop 函数，由此进入了 Tkinter 的主循环并成为事件驱动程序。由于这个示例中唯一安装了处理程序的事件就是单击按钮，因此，从应用程序的观点来看，在用户单击按钮之前，什么事情都没有发生。但是，在此期间，Tkinter 工具包还可以以期望的方式响应其他用户动作，比如移动 Tkinter 窗口，覆盖和显示窗口等。在用户调整窗口的大小时，包装器布局管理器将会工作，以更新部件的布局。在这个示例中，部件保持居中，接近窗口的上边缘，标签位于按钮之上。

所有去往 Tkinter 或者来自 Tkinter 的字符串都是 Unicode 字符串，因此，如果开发者需要显示或者接受 ASCII 编码之外的字符，请认真回顾第 9.6 节（然后使用适当的编码方式）。

本章使用了许多全部大写字母的多字母标识符（例如，LEFT、RAISE 和 ACTIVE）。所有这些标识符都是 Tkinter 模块的常数属性，并具有很广泛的用途。如果代码中使用的是 from Tkinter import *，则可以直接使用这些标识符。而如果代码中使用的是 import Tkinter，则需要确定这些标识符的资格，就像从 Tkinter 导入的所有其他对象一样，在对象的前面加上‘Tkinter.’。Tkinter 是极少的几个被设计为支持 from Tkinter import *

的 Python 模块之一，但是使用 import Tkinter 也是可取的，可以在牺牲一些便利的情况下获得更大的清晰度。在便利性和清晰度之间的一个好的折衷就是以更短的名称导入 Tkinter（例如，import Tkinter as Tk）。

对话框

Tkinter 附带了几个辅助模块以定义对话框（dialog），对话框都是模式框，在激活这些对话框后，只有在用户处理完对话框时才会将控制权返回到应用程序，在处理对话框时，用户可以提供一个要求的答案，或者通过单击某些按钮（比如 Cancel）拒绝回答。但是，这些很有用的模块都没有很好的文档，并缺少应用程序可能非常需要的功能，比如国际化（参见第 10.12 节）。要想更详细地了解对话框，并添加对话框这样的功能，本书建议开发者仔细查看 Python 标准库中这些模块的 Python 源代码：这些源代码也是一些如何使用 Tkinter 各个方面功能的好例子。本节只提供了这些模块的一个通用概述，以及这些模块最重要的功能中的一些突出点。

tkMessageBox 模块

tkMessageBox 提供了可以使用 Tk 的“消息框”的类、函数和常数。开发者可以指定要显示的按钮集，想要哪个按钮成为默认按钮，想要显示哪个图标，以及标题和消息字符串。通常，开发者只需要使用可选参数 title 和 message 调用下面的这些函数即可：

askokcancel

询问一个“是/否”问题；返回‘yes’或‘no’；

askquestion

询问一个“是/否”问题；返回‘yes’或‘no’；

askretrycancel

询问是否必须再次尝试一个操作；如果用户单击 RETRY，则返回 True；

askyesno

询问一个“是/否”问题；如果用户单击 YES，则返回 True；

showerror

显示一个错误消息；

showinfo

显示一个信息消息；

showwarning

显示一个警告消息。

tkSimpleDialog 模块

tkSimpleDialog 模块提供了一个 Dialog 类，这个类是一个基类，开发者可以通过创建该类的子类以创建自己的自定义对话框，tkSimpleDialog 模块还提供了 3 个很方便的函数，每个函数都有可选参数 title 和 prompt，并且每个函数都将返回一个用户响应（在用户单击 OK 时）或者 None（在用户单击 Cancel 时）：

askfloat

要求用户输入一个浮点型数字；返回一个 float 型值；

askinteger

要求用户输入一个整数；返回一个 int 型值；

askstring

要求用户输入一个字符串；返回一个普通 str 字符串（在用户只输入 ASCII 字符时）或一个 unicode 字符串（在所有其他情况下）。

tkFileDialog 模块

tkFileDialog 提供了一些类和函数，可以让用户选择一个文件或目录以进行加载（读取）或保存（写入）。这些类和函数都支持一些选项，比如，defaultextension 可以为文件指定一个默认扩展名，filetypes 可以指定接受哪些扩展名，而 initialdir 可以指定从哪里开始查找。这个模块中最常用的实用函数是：

askdirectory

返回一个目录的路径；

askopenfilename

返回一个或多个要打开的已有文件的路径；

askopenfilenames

返回一个要保存的文件的路径（如果该文件已经存在，需要进行确认）；

asksaveasfilename

返回一个要保存的文件的路径（如果该文件已经存在，需要进行确认）。

tkColorChooser 模块

tkColorChooser 提供了一个 askcolor 函数，开发者可以不带任何参数调用该函数，并返回用户选择的颜色，返回值是一个由 4 个项目组成的元组：red、green、blue，还有 Tkinter 颜色字符串。

17.2 部件基础知识

Tkinter 提供了多种类型的部件，大多数部件都有几个共同点。所有部件都是从 Widget 类继承的类的实例。Widget 类本身是抽象的；也就是说，不能实例化 Widget 本身。开发者只能实例化对应于某些特定类型的部件的具体子类。Widget 类的功能对于开发者实例化的所有部件都是通用的。

要想实例化任意类型的部件，都可以调用该部件的类。第一个参数是这个部件的父窗口，也被称为该部件的“主人”（master）。如果开发者忽略了这个位置参数，该部件的“主人”就是这个应用程序的主窗口。所有其他参数使用的都是 option=value 的命名形式。要想设置或更改一个已有部件 w 的选项，可以调用 w.config(option=value)。要想获得部件 w 的选项，可以调用 w.cget('option')，这样将返回该选项的值。每个部件 w 都是一个映射，因此开发者还可以使用 w['option']得到一个选项，并使用 w['option']=value 设置或更改这个选项。

通用部件选项

许多部件可以接受通用选项。有些选项会影响部件的颜色，而其他一些会影响部件的长度（通常按像素表示），还有各种其他类型的选项。本节将详细介绍最常使用的选项。

颜色选项

Tkinter 使用字符串来表示颜色。字符串可以是颜色的名称，比如‘red’或‘orange’，也可以是‘#RRGGBB’的形式，其中每个 R、G 和 B 都是一个 16 进制数字，这些数字表示红、绿、蓝三原色的值，这些值的范围从 0 到 255。如果开发者的屏幕不能显示数以百万的不同颜色，请不要担心，Python 将隐式使用这种解决方案：Tkinter 将把任何请求的颜色映射到开发者的屏幕上可以显示的最接近的颜色。通用的颜色选项是：

activebackground

在部件“活动”（active）时显示的背景颜色，“活动”表示将鼠标放在部件上并且单击该部件以让某些事情发生；

activeforeground

在部件活动时显示的前景颜色；

background（也可以是 bg）

部件的背景颜色；

disabledforeground

在部件被“禁用”（disable）时显示的前景颜色，“禁用”表示单击部件的操作将被忽略；

foreground（也可以是 fg）

部件的前景颜色；

highlightbackground

在部件有焦点时，突出显示的区域的背景颜色；

highlightcolor

在部件有焦点时，突出显示的区域的前景颜色；

selectbackground

部件的选定项目的背景颜色；用于具有可选项目的部件，比如 Listbox；

selectforeground

部件的选定项目的前景颜色。

长度选项

Tkinter 通常使用像素的整数值来表示长度；也可以使用其他度量单位，但是极少使用。通用长度选项是：

borderwidth

边框（如果有）的宽度，为部件提供了三维视图；

highlightthickness

在部件有焦点时，突出显示的矩形的宽度（在宽度为 0 时，该部件不绘制一个突出显示的矩形）；

padx

pady

部件在 x 和 y 方向上显示其内容所需的最小空间之外，从其布局管理器请求的额外空间。

selectborderwidth

包围部件的选定项目的三维边框（如果有的话）的宽度。

wraplength

可以执行自动换行的部件的最大行宽度（在小于或等于 0 时，不自动换行：部件只

是在出现‘\n’的位置将文本换行）。

表示字符数的选项

不是按像素，而是按字符数表示部件所要求的布局的一些选项，这些选项使用的是部件字体的平均宽度或高度：

height

部件的高度；必须大于或等于 1。

underline

在部件文本中带下划线的字符的索引（0 表示第一个字符，1 表示第二个字符，依此类推）；下划线字符是访问或激活该部件的快捷键。

width

部件的宽度（在小于或等于 0 时，这个宽度刚好足够放下该部件的当前内容）。

其他通用选项

各种部件都可以接受的其他一些选项是一个可以处理行为和外观的混合包：

anchor

在部件中的什么位置显示信息；必须是 N、NE、E、SE、S、SW、W、NW 或 CENTER（除了 CENTER 之外，所有的值都是指南针方向）。

command

可以不带任何参数执行的可调用函数；在用户单击部件的时候执行（仅用于按钮 Button、复选按钮 Checkbutton 和单选按钮 Radiobutton）。

font

部件中的文本的字体（参见第 17.6 节）。

image

部件中显示的图像，而不是文本；这个值必须是一个 Tkinter 图像对象（参见第 17.2 节）。

justify

在部件中显示多行文本时，文本行的对齐方式；必须是 LEFT（左对齐）、CENTER（居中）或 RIGHT（右对齐）。

relief

指定部件的内部相对于外部显示的三维效果；必须是 RAISED、SUNKEN、FLAT、

RIDGE、SOLID 或 GROOVE。

state

在鼠标和键盘单击时，部件的外观和行为；必须是 NORMAL、ACTIVE 或 DISABLED。

takefocus

如果这个值为 True，在用户按 Tab 或 Shift+Tab 组合键以在部件之间导航时，该部件可以接受焦点。

text

部件显示的文本字符串。

textvariable

与部件相关的 Tkinter 变量对象（参见第 17.2 节）。

通用部件方法

部件 w 提供了许多方法。除了事件相关方法（参见第 17.9 节）之外，常用的部件方法如表 17-1 所示。

表 17-1

cget	`w.cget(option)` 返回 w 中为 option 配置的值。
config	`w.config(**options)` 不带参数调用 w.config( )时，并返回一个字典，其中 w 的每个可能的选项将被映射到一个描述该选项的当前设置的元组上。带一个或多个命名参数调用 w.config( )时，config 将在 w 的配置中设置这些选项。
focus_set	`w.focus_set()` 将焦点设置到 w 上，这样，应用程序的所有键盘事件都被发送到 w 上。
grab_set, grab_release	`w.grab_set()` `w.grab_release()` grab_set 可以确保应用程序的所有事件都被发送到 w，直到调用了对应的 grab_release。
mainloop	`w.mainloop()` 输入 Tkinter 事件循环。事件循环是可以嵌套的；每次调用 mainloop 时将进入该事件循环的更深一级的嵌套。
quit	`w.quit()` 推出 Tkinter 事件循环。在事件循环被嵌套时，每次调用 quit 将退出该事件循环的一个嵌套级别。
update	`w.update()` 处理所有正在等待的事件。在正在处理一个事件时，不要调用这个方法。

续表

update_idletasks	`w.update_idletasks()` 处理那些正在等待的事件，这些事件通常是在事件循环空闲时处理的（比如，布局管理器更新和部件重绘），但是不执行任何回调。开发者可以在任何时候安全地调用这个方法。
wait_variable	`w.wait_variable(v)` v 必须是一个 Tkinter 变量对象（参见第 17.2 节）。wait_variable 只能在 v 的值改变时返回。在此期间，应用程序的其他部分仍保持活动。
wait_visibility	`w.wait_visibility(w1)` w1 必须是一个部件。wait_visibility 只能在 w1 变得可见时返回。在此期间，应用程序的其他部件仍保持活动。
wait_window	`w.wait_window(w1)` w1 必须是一个部件。wait_window 只能在 w1 被破坏时返回。在此期间，应用程序的其他部分仍保持活动。
winfo_height	`w.winfo_height()` 返回 w 的高度，以像素为单位。
winfo_width	`w.winfo_width()` 返回 w 的宽度，以像素为单位。

w 还提供了许多名称是以 winfo_开始的其他方法，但是上面的两个是最常调用的，通常是在调用 w.update_idletasks 方法之后调用。这些方法可以在用户调整了一个窗口的大小之后确定一个部件的维度，这将导致布局管理器重新安排部件的布局。

Tkinter 变量对象

Tkinter 模块提供了一些实例表示变量（variable）的类。每个类都可以处理特定的数据类型：DoubleVar 用于 float、IntVar 用于 int、StringVar 用于 str。不带任何参数实例化任何一个这样的类都可以获得一个被设置为 0 或 '' 的实例 x，在 Tkinter 中，这个实例也被称为变量对象（variable object）。x.set(datum)将把 x 的值设置为给定值；x.get( ) 将返回 x 的当前值。

开发者可以将 x 传递为部件的 textvariable 或 variable 配置选项。一旦这么做，该部件的文本将更改为跟踪 x 的值的任何更改，并且，对于某些类型的部件，x 的值将跟踪对部件所做的更改。Tkinter 变量可以控制多于一个的部件。Tkinter 变量可以比显式查询和设置部件属性更透明，有时候更方便地控制部件。下面的示例显示了如何使用 StringVar 自动连接一个 Entry 部件和一个 Label 部件：

```
import Tkinter

root = Tkinter.Tk()
tv = Tkinter.StringVar()
```

```
Tkinter.Label(textvariable=tv).pack()
Tkinter.Entry(textvariable=tv).pack()
tv.set('Welcome!')
Tkinter.Button(text="Exit", command=root.quit).pack()

Tkinter.mainloop()
print 'Final value is', tv.get()
```

在编辑 Entry 时，可以看到 Label 将自动更改。这个示例显式实例化了 Tkinter 的主窗口，将其绑定到名称 root，然后将绑定方法 root.quit 设置为 Button 的命令，这样将退出 Tkinter 的主循环，但是不会终止 Python 应用程序。这样，这个示例将以一个 print 语句结束，以在标准输出上显示 tv 变量对象最终的值。

Tkinter 图像

Tkinter 类 PhotoImage 可以处理“可交换的图像文件格式”（Graphical Interchange Format，GIF）和“可便携的像素映射”（Portable PixMap，PPM）图像。以一个命名参数 file = path 调用 PhotoImage 可以从给定 path 指定的文件加载图像数据并得到一个实例 x。

开发者可以将 x 设置为一个或多个部件的 image 配置选项。这些部件将显示图像，而不是文本。要想获得对许多图像格式（包括 JPEG、PNG 和 TIFF）的图像处理功能，可以使用 PIL，也就是 Python 图像库（http://www.pythonware.com/products/pil/），PIL 被设计用来与 Tkinter 一起工作。本书没有进一步介绍 PIL。

Tkinter 还提供了 BitmapImage 类，这个类的实例都可以在 PhotoImage 的实例所在的位置使用。BitmapImage 提供了一些被称为“位图”的文件格式。本书没有进一步介绍 BitmapImage。

被设置为部件的 image 配置选项也不足以让 PhotoImage 和 BitmapImage 的实例保持活动。可以将这样的实例保存在 Python 容器对象中，比如列表或字典，这样，该实例将不被垃圾收集。下面的示例给出了如何显示某些 GIF 图像：

```
import os
import Tkinter

root = Tkinter.Tk()
L = Tkinter.Listbox(selectmode=Tkinter.SINGLE)
gifsdict = {}

dirpath = 'imgs'
for gifname in os.listdir(dirpath):
    if not gifname[0].isdigit(): continue
    gifpath = os.path.join(dirpath, gifname)
    gif = Tkinter.PhotoImage(file=gifpath)
    gifsdict[gifname] = gif
```

```
        L.insert(Tkinter.END, gifname)

    L.pack()
    img = Tkinter.Label()
    img.pack()
    def list_entry_clicked(*ignore):
        imgname = L.get(L.curselection()[0])
        img.config(image=gifsdict[imgname])
    L.bind('<ButtonRelease-1>', list_entry_clicked)
    root.mainloop()
```

假定在某个目录（在这个示例中就是‘imgs’）中有几个 GIF 文件，文件名以数字开始，这个示例将把这些图像加载到内存中，在 Listbox 实例中显示这些文件的名称，然后在 Label 实例中显示用户单击的 GIF 文件的图像。为了简单，这个示例没有为 Listbox 部件设置滚动条 Scrollbar（第 17.3 节中将介绍如何为 Listbox 部件配置滚动条）。

17.3 常用的简单部件

Tkinter 模块提供了许多简单部件，这些部件可以满足简单 GUI 应用程序的大多数需求。本节将详细介绍 Button（按钮）、Checkbutton（复选按钮）、Entry（输入框）、Label（标签）、Listbox（列表框）、Radiobutton（单选按钮）、Scale（刻度）和 Scrollbar（滚动条）部件。

按钮

Button 类实现了一个按钮，用户可以单击按钮以执行一个动作。使用 text=somestring 选项实例化按钮可以让该按钮显示文本，而使用 image=imageobject 选项实例化按钮可以让该按钮显示一个图片。开发者通常还可以使用 command=callable，在用户单击该按钮时，这个选项可以让可调用对象（callable）不带参数执行。callable 可以是一个函数、一个对象的绑定方法、具有__call__方法的类的一个实例，或者是一个 lambda 表达式。

除了所有部件通用的方法之外，Button 类的实例 b 提供了两个按钮专用方法。

表 17-2

flash	*b*.flash() 通过多次重新绘制 b 吸引用户对按钮 b 的注意，也就是交替显示常规状态和活动状态。
invoke	*b*.invoke() 不带参数调用 b 的命令选项指定的可调用对象，就像 b.cget('command')( )一样。在某些动作中，如果开发者想要程序就像已经单击了按钮那样开始运行，这个方法是非常方便的。

复选按钮

Checkbutton 类实现了复选框（checkbox），这是一个小框，可以选择显示一个选择标记，用户可以单击这个选择标记来打开或关闭这个选项。可以通过两个选项来实例化 Checkbutton，选项 text=somestring 可以使用文本来标记复选框，选项 image=imageobject 可以使用图像来标记复选框。还可以选择使用 command=callable，这样可以在用户单击该复选框时让 callable 不带参数执行。callable 可以是一个函数、一个对象的绑定方法、具有__call__方法的类的一个实例，或者一个 lambda 表达式。

Checkbutton 的实例 c 必须与 Tkinter 变量对象 v 关联，也就是，使用配置 c 选项 variable=v。通常，v 是一个 IntVar 实例，在复选框未被选定时，v 的值为 0，在复选框被选定时，v 的值为 1。在复选框被选定或者未被选定时（用户单击复选框，或者使用代码调用 c 的 deselect、select 和 toggle 方法），v 的值会发生改变。反之亦然，当 v 的值更改时，c 也将适当地显示或隐藏复选标记。

除了所有部件通用的方法之外，Checkbutton 类的实例 c 还提供了 5 个复选框专用方法。

表 17-3

deselect	`c.deselect()` 删除 c 的复选标记，就像 c.cget('variable').set(0)一样。
flash	`c.flash()` 通过多次重绘 c 吸引用户对复选框 c 的注意，也就是交替显示常规状态和活动状态。
invoke	`c.invoke()` 不带参数调用 c 的命令选项指定的可调用对象，就像 c.cget('command')( )一样。
select	`c.select()` 显示 c 的复选标记，就像 c.cget('variable').set(1)一样。
toggle	`c.toggle()` 预先设定 c 的复选标记的状态，就像用户已经单击了 c 一样。

输入框

Entry 类实现了一个文本输入区域（text entry field），用户可以在这个部件中输入和编辑一行文本。Entry 的实例 e 提供了几个方法和配置选项，允许对部件操作和内容进行更细粒度的控制，但是，在大多数 GUI 程序中，开发者只需要使用以下 3 个输入框专用习惯用法即可：

```
e.delete(0, END)                # 清除部件的内容
e.insert(END, somestring)       # 将 somestring 添加到部件内容的末尾
somestring = e.get()            # 获得部件的内容
```

状态被设置为 DISABLED 的 Entry 实例（state=DISABLED）可以显示一行文本，并让

用户将其复制到剪贴板上。要想显示多于一行的文本，可以使用 Text 类，参见第 17.6 节。DISABLED 可以阻止程序和用户改变 e 的内容。要想执行任何改变，需要临时设置 state=NORMAL：

```
e.config(state=NORMAL)       # 允许改变 e 的内容
# 根据需要调用 e.delete 和/或 e.insert
e.config(state=DISABLED)     # 再次让 e 的内容不可改变
```

标签

Label 类实现了一个只显示文本或图像，而不与用户输入交互的部件。可以使用 text=somestring 选项实例化 Label 以让该部件显示文本，或者使用 image=imageobject 实例化 Label 以让该部件显示图像。

Label 类的实例 L 不能让用户从 L 将文本复制到剪贴板。因此，在用户可能想要将部件上显示的文本复制和粘贴到电子邮件或其他文档中时，L 并不是适当的部件。在使用 Entry 类的实例 e 时设置选项 state=DISABLED 可以避免对 e 进行更改，如上一节所述。

列表框

Listbox 类可以显示文本项目，并让用户选择一个或多个项目。要想为 Listbox 类的实例 L 设置文本项目，在大多数 GUI 程序中，只需要使用下面这两个列表框专用习惯用法即可：

```
L.delete(0, END)              # 清除列表框中的项目
L.insert(END, somestring)     # 将 somestring 添加到列表框的项目中
```

要想获得索引为 idx 的文本项目，可以调用 L.get(idx)。要想获得索引 idx1 到 idx2 之间的所有文本项目的列表，可以调用 L.get(idx1,idx2)。要想获得列表框中所有文本项目的列表，可以调用 L.get(0,END)。

selectmode 选项定义了 Listbox 实例 L 的选择模式（selection mode）。选择模式表示用户一次可以选定多少个项目：在 SINGLE 和 BROWSE 模式下可以选择一个；在 MULTIPLE 和 EXTENDED 模式下可以选择多个。selectmode 还定义了哪些用户动作会导致项目被选定或取消选定。默认值是 BROWSE；BROWSE 模式与 SINGLE 模式的区别在于，在 BROWSE 模式下，用户可以在按住鼠标左键的情况下，通过向上和向下移动更改选定的一个项目。在 MULTIPLE 模式下，每次单击一个列表项目将会选定或者取消选定该项目，而不会影响其他项目的选定状态。在 EXTENDED 模式下，单击一个列表项目将会选定该项目，并取消选定所有其他项目；但是，在按住 Ctrl 键的情况下单击以选定一个项目不会取消选定其他项目，而在按住 Shift 键的情况下单击可以选定一定范围内的项目。

Listbox 类的实例 L 提供了 3 个与选定项目相关的方法。

表 17-4

curselection	*L*.curselection() 返回一个由选定项目的 0 个或多个索引组成的序列，从 0 开始往上。根据底层使用的 Tk 版本，curselection 可能会返回整型索引的字符串表示，而不是整数。要想解决这种不确定性，可以使用： *indices* = [int(*x*) for *x* in *L*.curselection()] [L.get(x) for x in L.curselection( )] 总是由选定的 0 个或多个文本项目组成的列表，不管 curselection 返回哪种形式的索引。因此，如果开发者关心的只是选定的文本项目，而不是选定项目的索引，这种不确定性就不是一个问题。
select_clear	*L*.select_clear(*i*,*j*=None) 取消选定项目 i（如果 j 不为 None，则取消选定从 i 到 j 的所有项目，包括 j）。
select_set	*L*.select_set(*i*,*j*=None) 选定项目 i（如果 j 不为 None，则选定从 i 到 j 的所有项目，包括 j）。select_set 不会自动取消选定其他项目，即使 L 的选择模式为 SINGLE 或 BROWSE。

单选按钮

Radiobutton 类实现了一个可以被选定的小框。用户单击单选按钮可以将其打开或关闭。单选按钮组成一个组；选定一个单选按钮将自动取消选定相同组中的所有其他单选按钮。使用 text=somestring 选项调用 Radiobutton 可以使用文本来标记该按钮，使用 image=imageobject 可以使用图像来标记该按钮。还可以选择使用 command=callable，在用户单击单选按钮时，可以让 callable 不带参数执行。callable 可以是一个函数、一个对象的绑定方法、具有__call__方法的类的一个实例或者一个 lambda 表达式。

Radiobutton 的实例 r 必须与一个 Tkinter 变量对象 v 关联，也就是，使用 r 的配置选项 variable=v，对于指定值 X，使用 r 的选项 value=X。最常见的是，v 是一个 IntVar 实例。不管是通过用户单击 r，还是通过代码调用 r.select( )，在 r 被选定时，v 的值将更改为 X。反之亦然，在 v 的值更改时，当且仅当 v.get( )==X 时，r 将被选定。当且仅当单选按钮的几个实例具有相同的变量和不同的值时，这些单选按钮实例将形成一个组；选定一个实例将更改这个变量的值，这样还将自动取消选定以前被选定的任何一个其他实例。没有特殊的容器可以用来将 Radiobutton 实例组合成一个组，甚至也没有必要让单选按钮实例成为同一个部件的子部件。但是，如果开发者在不同的位置展开了一个单选按钮组，这样可能会让用户感到混淆。

除了所有部件都通用的方法，Radiobutton 类的实例 r 提供了 4 个单选按钮专用方法。

表 17-5

deselect	r.deselect() 取消选定 r，并将相关的变量对象设置为一个空白字符串，就像 r.cget('variable').set('')一样。

续表

flash	*c*.flash() 通过多次重绘 r 吸引用户对复选框 r 的注意，也就是交替显示常规状态和活动状态。
invoke	*c*.invoke() 不带参数调用 r 的命令选项指定的可调用对象，就像 r.cget('command')( )一样。
select	*r*.select() 选定 r 并将相关的变量对象设置为 r 的值，就像 r.cget('variable').set(r.cget('value'))一样。

刻度

Scale 类实现了一个部件，这个部件可以让用户通过沿着一条直线滑动游标来输入一个值。Scale 提供了一些配置选项来控制部件的外观和值的范围，但是，在大多数 GUI 程序中，开发者唯一指定的选项是 orient=HORIZONTAL，在开发者想要直线水平时（在默认情况下，直线是垂直的）使用。

除了所有部件通用的方法之外，Scale 类的实例 s 提供了两个刻度专用方法。

表 17-6

get	*s*.get() 返回 s 的游标的当前位置，通常是一个 0～100 的刻度值。
set	*s*.set(*p*) 设置 s 的游标的当前位置，通常是一个 0～100 的刻度值。

滚动条

Scrollbar 类实现了一个类似于 Scale 类的部件，这个部件用于为另一个部件（通常是列表框，参见第 17.3 节；文本，参见第 17.6 节；或者画布，参见第 17.7 节）提供滚动条功能，而不是让用户输入一个值。

Scrollbar 实例 s 将被连接到 s 控制的部件（例如，Listbox 实例 L）上，这是通过 s 和 L 上的配置选项实现的。出于这个目的，与滚动条相关的部件通常会提供一个名为 yview 的方法和一个名为 yscrollcommand 的配置选项以进行垂直滚动（要想进行水平滚动，Text、Cavas 和 Entry 这样的部件还提供了一个名为 xview 的方法和一个名为 xscrollcommand 的配置选项）。要想进行垂直滚动，可以使用 s 的选项 command=L.yview，这样，用户在 s 上的动作将调用 L 的绑定方法 yview 以控制 L 的滚动，另外，还需要使用 L 的选项 yscrollcommand=s.set，这样，通过调用 s 的绑定方法，L 的滚动而产生的更改将按顺序调整 s 显示的方式。下面的示例使用了滚动条来控制一个列表框的垂直滚动：

```
import Tkinter
s = Tkinter.Scrollbar()
```

```
L = Tkinter.Listbox()
s.pack(side=Tkinter.RIGHT, fill=Tkinter.Y)
L.pack(side=Tkinter.LEFT, fill=Tkinter.Y)
s.config(command=L.yview)
L.config(yscrollcommand=s.set)
for i in range(30): L.insert(Tkinter.END, str(i)*3)
Tkinter.mainloop()
```

因为 s 和 L 之间相互引用，因此两者都无法在构造实例的时候设置其各自的选项；为了一致性，可以在创建实例之后调用两个实例的 config 方法来设置这些选项。在这个示例中，需要将名称绑定到部件以调用该部件的 pack 和 config 方法，使用部件的绑定方法，并装配 Listbox. L=Tkinter.Listbox( ).pack( )不会将 L 绑定到 Listbox，而是绑定到 pack 的结果上（也就是，None）。可以通过下面这两条语句进行编码（就像在上面的示例中使用的那样）：

```
L = Tkinter.Listbox()
L.pack()
```

17.4 容器部件

Tkinter 模块提供了一些部件，其目的是包含其他部件。Frame 实例除了作为一个容器，不执行任何其他操作。Toplevel 实例（包括 Tkinter 的根窗口，也被称为应用程序的主窗口）是一个顶层窗口，因此开发者的窗口管理器需要与其进行交互（通常是通过提供适当的装饰和处理请求来实现的）。要想确保一个部件 parent（必须是一个 Frame 或 Toplevel 实例）是另一个部件 child 的父部件（也叫做主部件），可以在实例化 child 部件的时候，将 parent 部件传递为其第一个参数。

框架（Frame）

Frame 类是屏幕上的一个矩形区域，包含在其他框架或顶层窗口中。Frame 类的唯一目的就是包含其他部件。其 borderwidth 选项默认为 0，因此 Frame 的实例通常不显示边框。如果开发者想要看到框架边框的轮廓，可以将这个选项配置为 borderwidth=1。

顶层（Toplevel）

Toplevel 类表示顶层窗口屏幕上的一个矩形区域，因此可以从任何窗口管理器接收装饰以处理屏幕。Toplevel 类的每个实例都可以与窗口管理器进行交互，并包含其他部件。每个 Tkinter 程序至少有一个顶层窗口，也被称为根（root）窗口。开发者可以使用 root=Tkinter.Tk( )实例化 Tkinter 的根窗口；否则，Tkinter 将在第一次需要根窗口时隐式实例化其根窗口。如果开发者想要多于一个的顶层窗口，可以使用 root=Tkinter.Tk( )实例化最主要的一个。之后，按照需要实例化其他顶层窗口，比如调用 another_toplevel=

Tkinter.Toplevel( )。

Toplevel 类的实例 T 提供了许多方法，可以与窗口管理器进行交互操作。在这些方法中，许多都是平台专用的，只与 X Window 系统（大多在类 UNIX 系统中使用）的某些窗口管理器相关。其中，最常使用的跨平台方法如表 17-7 所示。

表 17-7

deiconify	*T*.deiconify() 让 T 正常显示，即使前一个 T 是图标化的或者看不见的。
geometry	*T*.geometry([*geometry_string*]) 在不带任何参数时，T.geometry( )方法将返回一个字符串，这个字符串对 T 的大小和位置进行了编码：widthxheight+x_offset+y_offset，其中 width、height、x_offset 和 y_offset 是十进制形式的像素值。在带有一个参数 S（一个相同形式的字符串）时，T.geometry(S)方法可以根据 S 设置 T 的大小和位置。
iconify	*T*.iconify() 将 T 显示为一个图标（在 Windows 中，显示为任务栏上的一个按钮）。
maxsize	*T*.maxsize([*width*,*height*]) 在不带任何参数时，T.maxsize( )方法将返回一个整数对，其中的两个项目是 T 的最大宽度和高度，以像素为单位。T.maxsize(w, h)方法带有两个整数参数 w 和 h，可以将 T 的最大宽度和高度分别设置为 w 和 h，以像素为单位。
minsize	*T*.minsize([*width*,*height*]) 在不带任何参数时，T.minsize( )方法将返回一个整数对，其中的两个项目是 T 的最小宽度和高度，以像素为单位。T.minsize(w, h)带有两个整数参数 w 和 h，可以将 T 的最小宽度和高度分别设置为 w 和 h，以像素为单位。
overrideredirect	*T*.overrideredirect([*avoid_decoration*]) 在不带任何参数时，对于普通窗口，T.overrideredirect( )将返回 False，而对于要求窗口管理器避免装饰的窗口，则返回 True。对于带有一个参数 x 的 T.overrideredirect(x)，当且仅当 x 为 True 时，该方法将要求窗口管理器避免装饰 T。不带装饰的顶层窗口 T 不包含标题，并且用户不能通过窗口管理器对 T 进行关闭、移动或调整大小的操作。
protocol	*T*.protocol(*protocol_name*,*callable*) 通过调用 protocol，并且第一个参数为‘WM_DELETE_WINDOW’（在大多数平台上，这是唯一有效的值），可以将 callable 安装为用户通过窗口管理器关闭 T（例如，单击 Windows 和 KDE 右上角的×按钮）时的处理程序。在用户进行这样的操作时，Python 将调用 callable，并且不带参数。callable 本身必须调用 T.destroy( )以关闭 T；否则，T 将保持打开。在默认情况下，如果 T.protocol 没有被调用，这样的关闭操作将隐式调用 T.destroy( )并无条件地关闭 T。

续表

resizable	*T*.resizable([*width*,*height*]) 在不带任何参数时，T.resizable( )将返回一个整数对（每个值为 0 或 1），这两个值分别指示用户在窗口管理器上的动作是否可以更改 T 的宽度和高度。在带有两个整数参数 w 和 h（每个值为 0 或 1）时，T.resizable(w, h)可以将用户动作设置为可以分别更改 T 的宽度和高度。在有些 Tk 版本中，在不带任何参数调用 resizable 时，将返回一个像 '1 1' 这样的字符串，而不是像（1, 1）这样的整数对。要想消除这种不确定性，可以使用： *resizable_wh* = *T*.resizable() if len(*resizable_wh*) != 2: *resizable_wh* = map(int, *resizable_wh*.split()) *resizable_w*, *resizable_h* = *resizable_wh*
state	*T*.state() 如果 T 为正常显示，则返回 'normal'，如果 T 看不见，则返回 'withdrawn'，如果 T 显示为一个图标（例如，在 Windows 中，只显示为任务栏上的一个按钮），则返回 'icon' 或 'iconic'（取决于窗口管理器）。
title	*T*.title([*title_string*]) 在不带任何参数时，T.title( )将返回一个字符串，该字符串是 T 的窗口标题。在带有一个参数 title_string 调用 T.title(title_string)时，将把 T 的窗口标题设置为字符串 title_string。
withdraw	*T*.withdraw() 让 T 不可见。

下面的示例显示了一个根窗口，其中包含一个 Entry 部件，可以让用户编辑这个窗口的标题，还包含一些按钮以执行各种根窗口操作：

```
import Tkinter
root = Tkinter.Tk()
var = Tkinter.StringVar()
entry = Tkinter.Entry(root, textvariable=var)
entry.focus_set()
entry.pack()
var.set(root.title())
def changeTitle(): root.title(var.get())
Tkinter.Button(root, text="Change Title", command=changeTitle).pack()
Tkinter.Button(root, text="Iconify", command=root.iconify).pack()
Tkinter.Button(root, text="Close", command=root.destroy).pack()
Tkinter.mainloop()
```

17.5 菜单部件

Menu 类实现了各种类型的菜单：顶层窗口的菜单栏、子菜单和弹出菜单。要想使用 Menu 实例 m 作为顶层窗口 w 的菜单栏，可以设置 w 的配置选项 menu = m。要想使用

m 作为 Menu 实例 x 的子菜单，可以使用 menu = m 调用 x.add_cascade。要想使用 m 作为弹出菜单，可以调用 m.post。

除了第 17.2 节中介绍的配置选项之外，Menu 实例 m 提供了 postcommand=callable 选项。Tkinter 可以在每次将要显示 m（因为调用 m.post，或者因为用户的动作）时不带任何参数调用 callable。使用这个选项可以在必要的时候非常及时地更新一个动态菜单。

在默认情况下，Tkinter 菜单显示了一个“虚线”菜单条目（其他菜单条目之前的一个虚线），这样可以让用户在一个单独的 Toplevel 窗口中获得该菜单的一个副本。因为这样的“虚线”菜单条目在流行的平台上并不是用户接口标准的一部分，开发者可能需要为这样的菜单使用配置选项 tearoff=0 来禁用虚线菜单条目功能。

菜单专用方法

除了所有部件通用的方法，Menu 类的实例 m 提供了几个菜单专用方法。

表 17-8

add,add_cascade, add_checkbutton, add_command, add_radiobutton, add_separator	*m*.add(*entry_kind, **entry_options*) 在 m 的已有条目后面添加一个新条目，其类型是由字符串 entry_kind 指定的，可以是‘cascade’、‘checkbutton’、‘command’、‘radiobutton’ 和 ‘separator’。第 17.5 节介绍了条目的类型和选项。 名称以 add_开始的方法就像 add 方法一样，但是不接受位置参数；每个方法添加的条目的类型都是由该方法的名称隐式指定的。
delete	*m*.delete(*i*[,*j*]) m.delete(i, j)可以删除 m 的第 i 个条目。m.delete(i, j)可以删除从 i 到 j（包含 j）的条目。第一个条目的索引为 0。
entryconfigure, entryconfig	*m*.entryconfigure(*i, **entry_options*) 更改 m 的第 i 个条目的条目选项。entryconfig 是一个完全同义的词。
insert, insert_cascade, insert_checkbutton, insert_command, insert_radiobutton, insert_separator	*m*.insert(*i,entry_kind, **entry_options*) 在 m 的条目 i 之前添加一个新条目，其类型是由字符串 entry_kind 指定的，可以是‘cascade’、‘checkbutton’、‘command’、‘radiobutton’ 和 ‘separator’。第 17.5 节介绍了条目的类型和选项。 名称以 insert_开始的方法就像 insert 方法一样，但是不接受位置参数；每个方法添加的条目的类型都是由该方法的名称隐式指定的。
invoke	*m*.invoke(*i*) 调用 m 的第 i 个条目，就像用户单击了该条目一样。
post	*m*.post(*x,y*) 将 m 显示为一个弹出菜单，m 的左上角位于坐标 x 和 y 上（从 Tkinter 的根窗口的左上角开始的偏移量，以像素为单位）。
unpost	*m*.unpost() 如果 m 显示为一个弹出菜单，将关闭 m；否则，不执行任何操作。

菜单条目

在菜单 m 显示时，将显示一个垂直（菜单栏是水平的）的项目列表。每个条目都可以是下面的几个类型之一。

cascade

一个子菜单；在选项 menu=x 中，x 必须是另一个 Menu 实例。

checkbutton

类似于 Checkbutton 部件；典型选项都是 variable（指定一个 Tkinter 变量对象）、onvalue、offvalue，还可以选择使用 command，就像 Checkbutton 实例中使用的一样。

command

类似于 Button 部件；一个典型的选项是 command=callable。

radiobutton

类似于 Radiobutton 部件；典型的选项是 variable（指定一个 Tkinter 变量对象）、value，还可以选择使用 command，就像 Radiobutton 实例中使用的一样。

separator

分隔由其他条目组成的组的线段。

菜单条目经常使用的其他项目是：

image

选项 image=x 可以使用 x（Tkinter 图像对象）以使用图像，而不是文本作为该条目的标签。

label

选项 label=somestring 可以使用一个文本字符串作为该条目的标签。

underline

选项 underline=x 可以将 x 作为该条目的标签中的下划线字符的索引（0 表示第一个字符，1 表示第二个字符，依此类推）。

菜单示例

下面的示例显示了如何将典型的 File 和 Edit 菜单添加到一个菜单栏上：

```
import Tkinter

root = Tkinter.Tk()
bar = Tkinter.Menu()
```

```
def show(menu, entry): print menu, entry

fil = Tkinter.Menu()
for x in 'New', 'Open', 'Close', 'Save':
    fil.add_command(label=x,command=lambda x=x:show('File',x))
bar.add_cascade(label='File',menu=fil)

edi = Tkinter.Menu()
for x in 'Cut', 'Copy', 'Paste', 'Clear':
    edi.add_command(label=x,command=lambda x=x:show('Edit',x))
bar.add_cascade(label='Edit',menu=edi)
root.config(menu=bar)
Tkinter.mainloop()
```

在这个示例中，出于演示的目的，每个菜单命令只是将信息输出到标准输出中（请注意，x=x 习惯用法可以在创建每个 lambda 时“快照”x 的值）。否则，在 lambda 执行时，x 的当前值‘Clear’将在每次选择菜单项目时显示出来。另一个比使用 x=x 习惯用法的 lambda 表达式更好的方法就是使用闭包。这样将使用下面的代码，而不是 def show：

```
def mkshow(menu, entry):
    def emit(): print menu, entry
    return emit
```

然后，在调用 fil 和 edi 菜单的 add_command 方法时，将分别使用 command=mkshow('File', x)和 command=mkshow('Edit', x)。

17.6 文本部件

Text 类实现了一个强大的多行文本编辑器，这个编辑器可以显示图像和嵌入的部件，以及多种字体和颜色的文本。Text 的实例 t 支持许多方法以处理 t 的特定内容。t 提供了一些方法和配置选项，这些方法和选项可以对文本操作、内容和表现形式进行细粒度的控制。本节介绍了这些丰富功能的一个比较大的、经常使用的功能子集。在某些非常简单的情况下，只需要用到以下 3 个文本专用的习惯用法：

```
t.delete('1.0', END)                    # 清除部件的内容
t.insert(END, astring)                  # 将 astring 添加到部件内容的末尾
somestring = t.get('1.0', END)          # 以字符串的形式得到部件的内容
```

END 是所有 Text 实例 t 上的一个索引，表示 t 的文本的末尾。‘1.0’也是一个索引，表示 t 的文本的起始位置（第一行，第一列）。要想了解更多有关索引的信息，参见第 17.6 节。

ScrolledText 模块

Python 标准库中的 ScrolledText 模块提供了一个名为 ScrolledText 的类。要想构造

ScrolledText 实例，可以按照与调用 Tkinter.Text 完全相同的方法调用 ScrolledText. ScrolledText。ScrolledText 实例 s 与 Text 实例几乎完全一样，区别只是 s 将自动为其环绕的 Text 实例提供一个滚动条。

文本部件的方法

Text 部件的实例 t 提供了许多方法（第 17.6 节介绍了用来处理标志和标签的方法）。许多方法可以接受一个或两个对 t 的内容的索引。最常使用的方法如表 17-9 所示。

表 17-9

delete	`t.delete(i[,j])` t.delete(i)可以删除 t 的内容中索引为 i 的字符。t.delete(i, j)将删除从索引 i 到索引 j 的所有字符，包括索引为 j 的字符。
get	`t.get(i[,j])` t.get(i)将返回 t 的内容中索引为 i 的字符。t.get(i, j)将返回一个字符串，该字符串连接了从索引 i 到索引 j 的所有字符，包括索引为 j 的字符。
image_create	`t.image_create(i,**window_options)` 在 t 的内容中索引为 i 的位置插入一个嵌入的图像。带选项 image=e 调用 image_create，其中 e 是 Tkinter 图像对象，参见第 17.2 节。
insert	`t.insert(i,s[,tags])` 在 t 的内容中索引为 i 的位置插入字符串 s。如果提供了 tags，则 tags 是一个字符串序列，这个字符串序列将被粘贴为新文本的标签，参见第 17.6 节。
search	`t.search(pattern,i,**search_options)` 在 t 的内容中索引 i 之后的位置查找第一次出现的字符串 pattern，并返回一个字符串，这个字符串是找到的字符串的索引，如果没有找到该字符串，则返回一个空白字符串 ‘’。选项 nocase=True 将搜索操作设置为不区分大小写；在默认情况下，或者使用显式选项 nocase=False 时，搜索操作是区分大小写的。选项 stop=j 可以让搜索操作在到达索引 j 时停止；在默认情况下，搜索操作将环回到 t 的内容的开始。在需要避免环回时，可以使用 stop=END。
see	`t.see(i)` 如果需要，滚动 t 以确信索引 i 处的内容是可见的。如果索引 i 处的内容已经可见，see 不执行任何操作。
window_create	`t.window_create(i,**window_options)` 在 t 的内容中索引为 i 的位置插入一个嵌入的部件。t 必须是插入的部件 w 的父部件。可以使用选项 window=w 或者使用选项 create=callable 调用 window_create 以插入一个已经存在的部件 w。如果使用选项 create，在第一次需要显示嵌入的部件的时候，Tkinter 将不带任何参数调用 callable，并且 callable 必须创建并返回一个父部件为 t 的部件 w。create 选项可以非常及时地安排嵌入的部件的创建，在一个非常长的文本中包含许多嵌入的部件时，这是一个非常有用的优化选项。

续表

xview, yview	*t*.xview([...]) *t*.yview([...]) xview 和 yview 可以分别处理水平和垂直方向上的滚动操作，并接受几个不同的参数模式。不带任何参数调用 t.xview( )时，将返回一个由 0.0～1.0 的两个浮点型值组成的元组，这两个值表示 t 中对应于第一个（最左边）和最后一个（最右边）当前可见的列的内容的小数比例。t.xview (MOVETO,frac)将 t 向左或向右滚动，这样，第一个（最左边）可见的列将变成对应于 t 的内容的小数比例 frac 的列，frac 在 0.0～1.0 之间。yview 支持相同模式的参数，但是操作的是行，而不是列，并且是上下滚动，而不是左右滚动。yview 还支持另一个模式的参数：对于任意索引 i，t.yview(i)将向上或向下滚动 t，这样第一个（最上面）的可见行将变成索引为 i 的行。

标志

Text 实例 t 上的标志（mark）是一个名称，用来指示 t 的内容中的一个位置。INSERT 和 CURRENT 都是任意 Text 实例 t 上的标志，具有预定义的含义。INSERT 指定了“插入光标”（也被称为文本的脱字符号^）在 t 中所在的位置。在默认情况下，当用户在键盘上输入文本，并且焦点在 t 上时，t 将在索引 INSERT 处插入文本。CURRENT 指定了在用户最后一次在 t 中移动鼠标时，t 中最接近于鼠标光标的位置。在默认情况下，当用户在 t 上单击鼠标时，t 将获得焦点，并将 INSERT 设置为 CURRENT。

要想在 t 上设置其他标志，可以调用 t.mark_set 方法。每个标志都是一个不包含空白字符的任意字符串。要想避免与其他形式的索引混淆，标志中不使用标点符号。标志是一个索引，参见第 17.6 节；在 t 的方法可以接受一个索引参数的任何位置都可以传递一个字符串，这个字符串是 t 上的一个标志。

在标志 m 之前插入或删除文本时，m 将相应地移动。删除 m 周围的一部分文本不会删除 m。要想删除 t 上的一个标志，可以调用 t.mark_unset 方法。在标志 m 所在的位置插入文本时发生的事情取决于 m 的引力设置，这个设置可以是 RIGHT（默认值）或 LEFT。在 m 的引力设置为 RIGHT 时，m 将移动到在 m 处插入的文本的末尾（也就是，到最右边），并保留在该位置。在 m 的引力设置为 LEFT 时，当用户在 m 处插入文本时，m 将不会移动：m 处插入的文本将跟着 m 的后面，并且 m 本身将保持位于这样插入的文本的起始位置（也就是，在最左边）。

Text 实例 t 提供了如表 17-10 所示与 t 上的标志有关的方法。

表 17-10

mark_gravity	*t*.mark_gravity(*mark*[,*gravity*]) mark 是 t 上的一个标志。t.mark_gravity(mark)将返回 mark 的引力设置，RIGHT 或 LEFT。t.mark_gravity(mark,gravity)将把 mark 的引力设置为 gravity，这个设置必须是 RIGHT 或 LEFT。

续表

mark_set	*t*.mark_set(*mark*,*i*) 如果 mark 还不是 t 上的一个标志，mark_set 可以在索引 i 处创建 mark。如果 mark 已经是 t 上的一个标志，则 mark_set 将把 mark 移动到索引 i 处。
mark_unset	*t*.mark_unset(*mark*) mark 是 t 上的一个用户自定义标志(而不是预定义标志 INSERT 或 CURRENT 之一)。mark_unset 将从 t 的标志中删除 mark。

标签

Text 实例 t 上的标签（tag）是一个符号名，表示 Text 实例 t 的内容中的零个或多个区域(范围)。SEL 是任意 Text 实例 t 上的一个预定义的标签，这个标签指定了 t 中被选定的单个范围，通常是由用户使用鼠标在文本上拖动而选定的。Tkinter 通常使用不同的背景和前景色来显示 SEL 范围。要想在 t 上创建其他标签，可以调用 t.tag_add 或 t.tag_config 方法，或者使用 t.insert 方法的可选参数 tags。t 上的各种标签的范围可能会重叠。t 将使用最上面的标签来为具有几个标签的文本着色，最上面的标签是根据 t.tag_raise 或 t.tag_lower 方法调用确定的。在默认情况下，最新创建的标签在以前创建的标签之上。

每个标签都是一个不包含空白字符的任意字符串。每个标签都有两个索引：first（标签的第一个范围的起始位置）和 last（标签的最后一个范围的结束位置）。在 t 的方法可以接受一个索引参数的任何位置都可以传递一个标签的索引。SEL_FIRST 和 SEL_LAST 可以表示预定义标签 SEL 的 first 和 last 索引。

Text 实例 t 提供了如表 17-11 所示与 t 上的标签相关的方法。

表 17-11

tag_add	*t*.tag_add(*tag*,*i*[,*j*]) t.tag_add(tag,i)将把标签 tag 添加到 t 中索引为 i 处的单个字符上。t.tag_add(tag,i,j)将把标签 tag 添加到从索引 i 到索引 j 的字符上。
tag_bind	*t*.tag_bind(*tag*,*event_name*,*callable*[,'+']) t.tag_bind(tag,event_name,callable)可以将 callable 设置为 tag 上的 event_name 的回调函数。t.tag_bind(tag,event_name,callable,'+')可以将 callable 添加到以前的绑定上。第 17.9 节中将介绍事件、回调函数和绑定。
tag_cget	*t*.tag_cget(*tag*,*tag_option*) 返回当前与标签 tag 的 tag_option 选项相关的值。例如，t.tag_cget(SEL,'background')可以返回 t 中用作 t 的选定范围的背景色的颜色。
tag_config	*t*.tag_config(*tag*,***tag_options*) 设置或更改与标签 tag 相关的标签选项，这些选项确定了 t 为该 tag 范围之内的字符着色的方式。最常使用的标签选项是： background foreground　背景色和前景色；

续表

tag_config	bgstipple fgstipple　背景点画和前景点画，通常为‘gray12’、‘gray25’、‘gray50’或‘gray75’；默认为纯色（无点画）； Borderwidth　文本边框的像素宽度；默认值为 0（无边框）； font　标签范围中的文本使用的字体（参见第 17.6 节）； justify　文本对齐方式：LEFT（默认值）、CENTER 或 RIGHT； lmargin1 lmargin2 rmargin　左边距（第一行，其他行）和右边距（所有行），以像素为单位；默认值为 0（无边距）； offset　基线偏移，以像素为单位（大于 0 表示上标，小于 0 表示下标）；默认值为 0（无基线——也就是，文本按基线水平对齐）； overstrike　如果为 True，在文本上绘制一条线； relief　文本风格：FLAT（默认值）、SUNKEN、RAISED、GROOVE 或 RIDGE； spacing1, spacing2, spacing3　行间距，以像素为单位（第一行之前、行之间或者最后一行之后）；默认值为 0（无行间距）； underline　如果为 True，在文本的下面绘制一条线； wrap　环绕模式：WORD（默认值）、CHAR 或 NONE。 例如： `t.tag_config(SEL,background='black',foreground='yellow')` 可以将 t 配置为以黑底黄字显示 t 的选定范围。
tag_delete	*t*.tag_delete(*tag*) 删除与 t 上的标签 tag 相关的所有信息。
tag_lower	*t*.tag_lower(*tag*) 将 tag 的选项设置为与其他标签覆盖的区域中的最小优先级。
tag_names	*t*.tag_names([*i*]) 返回一个字符串列表，其中的项目都是包含索引 i 的标签。在不带任何参数调用 tag_names 时，将返回一个字符串列表，其中的项目是当前存在于 t 中的所有标签。
tag_raise	*t*.tag_raise(*tag*) 将 tag 的选项设置为与其他标签覆盖的区域中的最大优先级。
tag_ranges	*t*.tag_ranges(*tag*) 返回一个由偶数个字符串组成的列表（如果 tag 不是 t 中的标签，或者没有范围，则返回 0），并交换 tags 的范围的起始和停止索引。
tag_remove	*t*.tag_remove(*tag,i*[,*j*]) t.tag_remove(tag,i)将从 t 中索引为 i 处的单个字符上删除标签 tag。t.tag_remove(tag,i,j)将从索引 i 到索引 j 的字符上删除标签 tag。从不具有这个标签的字符上删除一个标签不执行任何操作。
tag_unbind	*t*.tag_unbind(*tag,event*) t.tag_unbind(tag,event)将删除 tag 范围内的事件的任何绑定。第 17.9 节“Tkinter 事件”介绍了事件和绑定。

索引

在 Text 实例 t 的内容中指定一个位置的所有方法都被称为 t 上的索引。索引的基本形式是一个形式为'%d.%d'% (L,C) 的字符串，这个字符串表示文本中 L 行（第一行为 1），C 列（第一列为 0）所在的位置。例如，“1.0”是一个基本形式的索引，表示任意 t 的文本的起始位置。t.index(i)可以返回等同于任意形式的索引 i 的一个基本形式。

END 是一个用来表示任意 t 的文本结束位置的索引。对于任意行 L，'%d.end'%L 是一个表示第 L 行的结束位置的索引（也就是‘\n’行尾标志）。例如，‘1.end’表示第一行的末尾。要想获得 Text 实例 t 的第 L 行中的字符数，可以使用：

```
def line_length(t, L):
    return int(t.index('%d.end'%L).split('.')[-1])
```

@%d,%d'%(x,y)也是 t 上的一个索引，其中 x 和 y 是 t 的窗口中以像素为单位的坐标。

t 上的任意标签都与两个索引有关：字符串'%s.first'%tag（tag 的第一个范围的起始位置）和'%s.last'%tag（tag 的最后一个范围的结束位置）。例如，在 t.tag_add('mytag',i,j)之后，‘mytag.first’表示 t 中与索引 i 相同的位置，而‘mytag.last’表示 t 中与索引 j 相同的位置。在 t 中没有标签‘x’时，使用索引‘x.first’或‘x.last’将引发一个异常。

SEL_FIRST 和 SEL_LAST 都是索引（SEL 标签选定的范围的起始位置和结束位置）。在 t 中没有选定范围时使用 SEL_FIRST 和 SEL_LAST 将引发一个异常。

标志（参见第 17.6 节），包括预定义标志 INSERT 和 CURRENT，也都是索引。t 中嵌入的任何图像或部件也是 t 中的一个索引(参见第 17.6 节中介绍的 image_create 方法和 window_create 方法)。

索引表达式（index expression）是另一种形式的索引，索引表达式可以将下面的一个或多个修改符字符串字面常量连接到任何字符串形式的索引上：

'+ *n* chars' '- *n* chars'

文本的结尾或开始的 n 个字符（包括换行符）；

'+ *n* lines' '- *n* lines'

文本的结尾或开始的 n 个行；

'linestart' 'lineend'

索引的行的第 0 列，或者索引的行的换行符‘\n’；

'wordstart' 'wordend'

包含索引的单词的开始或结尾（在这种上下文环境下，一个单词表示一个由字母、数字和下划线组成的序列）。

开发者可以有选择地忽略空格，以及将关键字缩略为一个字符。例如，'%s-4c'%END 表示“t 的文本内容结束之前的 4 个字符”，'%s+1line linestart'%SEL_LAST 表示“t 的选定范围结束之后的第一行的开始”。

Text 实例 t 提供了与 t 上的索引相关的两个方法。

表 17-12

compare	*t*.compare(*i*,*op*,*j*) 返回 True 或 False，反映了索引 i 和 j 的比较结果，其中较低的数字表示更前面的索引，op 是‘<’、‘>’、‘<=’、‘>=’、‘==’和‘!=’运算符之一。例如，如果 t 包含多于 90 个字符（将行尾也计算为一个字符），则 t.compare('1.0+90c','<',END) 将返回 True。
index	*t*.index(*i*) 返回索引 i 的基本形式‘L.C’，其中 L 和 C 是 i 的行和列的十进制字符串形式（行从 1 开始；列从 0 开始）。

字体

开发者可以使用选项 font=font 更改任何 Tkinter 部件上的字体。在大多数情况下，更改部件的字体并没有什么意义。但是，在 Text 实例中，尤其是对于文本上的特殊标签，更改字体是非常有用的。

tkFont 模块提供了 Font 类；BOLD、ITALIC 和 NORMAL 属性以定义字体的特征；还有 families 函数（返回一个字符串序列，该序列指定了可用的字体系列）和 names 函数（返回一个字符串序列，该序列指定了所有用户自定义字体）。常用的字体选项是：

family

字体系列（例如，‘courier’或‘helvetica’）

size

字体大小（如果值为正，则以磅为单位，如果值为负，则以像素为单位）。

slant

NORMAL（默认值）或 ITALIC

weight

NORMAL（默认值）或 BOLD

Font 的实例 F 提供了如表 17-13 所示的常用方法。

表 17-13

actual	*F*.actual([*font_option*]) 在不带任何参数调用 F.actual( )时，将返回一个字典，其中包含 F 中实际使用的所有选项（与请求的字体最接近的字体）。F.actual(font_option)将根据 font_option 选项返回 F 中实际使用的值。
cget	*F*.cget(*font_option*) 返回 F 中为 font_option 选项配置的值（也就是，请求的值）。
config	*F*.config(***font_options*) 在不带任何参数调用 F.config( )时，将返回一个字典，其中包含 F 中配置的所有选项（也就是，请求的字体）。在带有一个或多个命名参数时，config 将设置 F 的配置中的字体选项。
copy	*F*.copy() 返回一个字体 G，该字体是 F 的一个副本。然后，开发者可以分别修改 F 和 G，并且对其中一个的任何修改都不影响另外一个。

文本示例

为了举例说明 Text 类中的一些功能，下面的示例显示了一种方法以突出显示文本中出现的与某个字符串相同的全部字符串。

```
from Tkinter import *

root = Tk()

# 在根窗口的顶部，从左到右放置一个标签、一个条目和一个按钮
fram = Frame(root)
Label(fram,text='Text to find:').pack(side=LEFT)
edit = Entry(fram)
edit.pack(side=LEFT, fill=BOTH, expand=1)
edit.focus_set()
butt = Button(fram, text='Find')
butt.pack(side=RIGHT)
fram.pack(side=TOP)

# 以 Text 填充根窗口的其余部分，并在其中放置一些文本。
text = Text(root)
text.insert('1.0',
'''Nel mezzo del cammin di nostra vita
mi ritrovai per una selva oscura
che la diritta via era smarrita
''')
text.pack(side=BOTTOM)

# 按钮的操作函数：突出显示找到的所有字符串
def find():
    # 删除前面使用的标签 'found'，如果有的话
```

```
        text.tag_remove('found', '1.0', END)
        # 获得要查找的字符串（如果为空，则不搜索）
        s = edit.get()
        if s:
            # 从起始点开始（并在到达末尾时停止）
            idx = '1.0'
            while 1:
                # 查找下一个字符串，如果没有找到，则退出循环
                idx = text.search(s, idx, nocase=1, stopindex=END)
                if not idx: break
                # 找到的字符串结束之后的索引
                lastidx = '%s+%dc' % (idx, len(s))
                # 标记整个查找字符串（包含起始位置，不包含结束位置）
                text.tag_add('found', idx, lastidx)
                # 准备搜索下一个查找字符串
                idx = lastidx
            # 为所有标签字符串使用红色前景
            text.tag_config('found', foreground='red')
    # 将焦点返回到 Entry 域
    edit.focus_set()

# 安装操作函数以在用户单击按钮时执行该函数
butt.config(command=find)

# 开始整个显示（进入事件驱动状态）
root.mainloop()
```

这个示例还显示了如何使用 Fram 来执行一个简单的部件布局任务（将 3 个部件并排放置，将 Text 部件放在这 3 个部件之下）。图 17-1 显示了这个示例的执行情况。

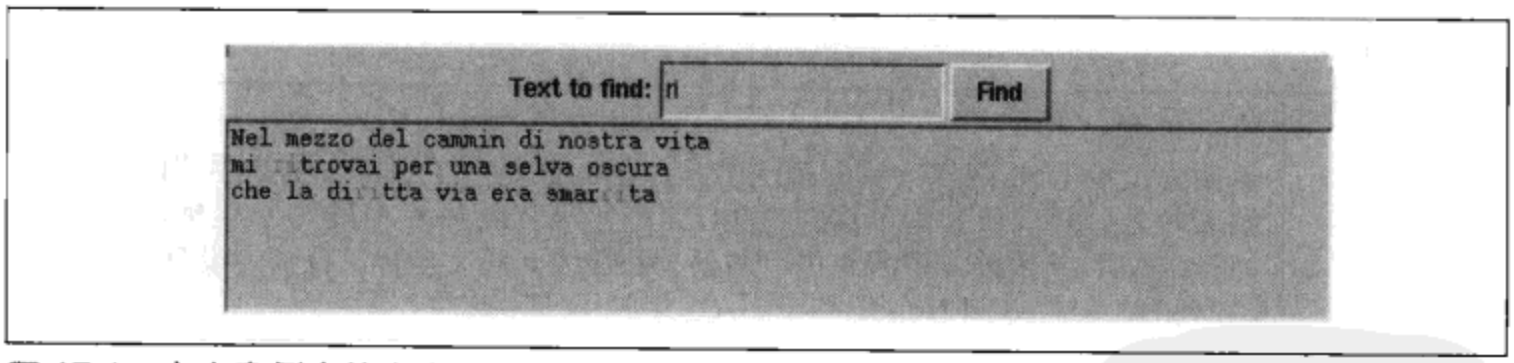


图 17-1　文本实例中的突出显示

17.7　画布部件

画布（Canvas）类是一个可以用于许多目的的，功能强大、灵活的部件，包括绘图，特别是构建自定义部件。构建自定义部件是一个高级主题，本书没有进一步对此进行介绍。本节只介绍了用于最简单的几种绘图的 Canvas 功能的一个子集。

Canvas 实例 c 中的坐标是以像素为单位的，坐标原点位于 c 的左上角，正坐标轴为向

右和向下。开发者可以使用一些高级方法来更改 c 的坐标系统，但是本书并没有对此进行介绍。

Canvas 实例 c 上绘制的内容叫做画布项目（canvas item）：线条、多边形、Tkinter 图像、弧线、椭圆、文本和其他一些项目。每个项目都有一个项目句柄（item handle），开发者可以使用项目句柄来引用该项目。开发者还可以将被称为标签的符号名称赋值给画布项目的集合（由包含可以重叠的不同标签的项目组成的集合）。ALL 是一个可以应用于所有项目的预定义标签；CURRENT 是一个预定义标签，可以应用于鼠标指针下的项目。

Canvas 上的标签不同于 Text 上的标签。画布标签只是一组项目集合，不包含独立存在的项目。在对作为项目标识符的 Canvas 标签执行任何操作时，这些操作将应用到作为该标签的当前集合的项目上。项目是以后从画布标签上删除的还是以后添加到画布标签上的，两者并没有什么区别。

要想创建一个画布项目，可以对画布实例 c 调用一个以 create_kindofitem 这种形式命名的方法，这个方法将返回新项目的句柄。c 的 itemcget 和 itemconfig 方法可以用来得到和更改项目的选项。

项目的画布方法

Canvas 实例 c 提供了一些可以对项目调用的方法。item 参数可以是一个项目的句柄——是由，例如 c.create_line 返回的——或者是一个标签，表示这个标签集合中的所有项目（如果这个标签的集合当前是空白的，则根本不包含任何项目），除非在该方法的描述中有其他指示。

表 17-14

bbox	*c*.bbox(*item*) 返回一个由 item 限定的近似范围的框，返回值是一个由 4 个整数组成的元组：按顺序分别是以像素为单位的最小 x、最小 y、最大 x 和最大 y 坐标。例如，c.bbox(ALL)将返回 c 中所有项目的最小和最大 x 和 y 坐标。在 c 中没有项目时，c.bbox(ALL)将返回 None。
coords	*c*.coords(*item*,**coordinates*) 更改 item 的坐标。只能对一个项目进行操作。如果 item 是一个标签，则 coords 将对当前在该标签的集合中的任意项目进行操作。如果 item 是一个包含空白项目集的标签，coords 则是一个无意义的无效操作。
delete	*c*.delete(*item*) 删除 item。例如，c.delete(ALL)将删除 c 上的所有项目。
gettags	*c*.gettags(*item*) 返回由所有项目集中包含 item 的标记组成的序列（但不包含 ALL 标签，尽管该标签包含所有项目，也不包含 CURRENT，不管该标签是否包含 item）。

续表

itemcget	*c*.itemcget(*item*,*option*) 返回 item 的 option 的值。只能对一个项目进行操作。如果 item 是一个标签，itemcget 将返回当前在标签的集合中的任意一个项目的 option 的值。如果 item 是一个包含空白项目集的标签，则 itemcget 将返回空白字符串 ‘’。
itemconfig	*c*.itemconfig(*item*,***options*) 设置或更改 item 的 options 的值。例如，c.itemconfig(ALL, fill='red')将把 c 上的所有项目的填充色设置为红色。
tag_bind	*c*.tag_bind(*tag*,*event_name*,*callable*[,'+']) c.tag_bind(tag,event_name,callable)可以将 callable 设置为当前在 tag 的集合中的项目上的 event_name 的回调函数。c.tag_bind(tag,event_name,callable,'+')可以将 callable 添加到前面的绑定上。第 17.9 节介绍了事件、回调函数和绑定。
tag_unbind	*c*.tag_unbind(*tag*,*event*) c.tag_unbind(tag, event)可以删除当前在 tag 的集合中的项目上的事件的绑定。第 17.9 节介绍了事件和绑定。

线条画布项目

Canvas 实例 c 提供了一个方法以创建一个线条（line）项目。

表 17-15

create_line	*c*.create_line(**coordinates*, ***line_options*) 使用给定的坐标 coordinates 指定的顶点创建一个线条项目，并返回该项目的句柄。coordinates 必须包含偶数个位置参数，x 坐标和 y 坐标交替出现以指定线条的每个顶点。坐标是以像素为单位的，原点（0,0）位于左上角，x 轴向右，y 轴向下。开发者可以对 c 设置不同的坐标系统，但是本书没有对此进行介绍。 line_options 可以包含： arrow　线条的哪一端包含箭头：NONE（默认值）、FIRST、LAST 或 BOTH； fill　线条的颜色（默认值为黑色）； smooth　如果为 True，该线条将被绘制成一个平滑曲线（B 样条曲线）；否则（默认值 False），线条将被绘制为一个多边形（线段序列）； tags　可以是一个字符串，用来在项目上设置一个标签，或者是一个由字符串组成的元组，用来在项目上设置多个标签； width　线条的宽度，以像素为单位（默认值为 1）。 例如： x=c.create_line(0,150,50,100,0,50,50,0 smooth=1) 可以在 c 上绘制一个 S 形曲线，将该曲线的句柄绑定到名称 x 上。然后，开发者可以使用下面这条语句将这条曲线的颜色更改为蓝色： c.itemconfig(x,fill='blue')

多边形画布项目

Canvas 实例 c 提供了一个创建多边形（polygon）项目的方法。

表 17-16

create_polygon	*c*.create_polygon(**coordinates*, ***poly_options*) 使用给定坐标 coordinates 指定的顶点创建一个多边形项目，并返回该项目的句柄。cordinates 必须包含偶数个位置参数，x 坐标和 y 坐标交替出现以指定多边形的每个顶点，并且至少必须有 6 个位置参数（也就是，至少 3 个顶点）。poly_option 可以包含： fill 多边形的内部颜色（默认值为黑色）； outline 多边形的边的颜色（默认值为黑色）； smooth 如果为 True，该多边形将被绘制成一个平滑曲线（B 样条曲线）；否则（默认值 False），该多边形将被绘制为一个普通多边形（边序列）； tags 可以是一个字符串，用来在项目上设置一个标签，或者是一个由字符串组成的元组，用来在项目上设置多个标签； width 线条的宽度，以像素为单位（默认值为 1）。 例如： *x*=c.create_polygon(0,150, 50,100, 0,50, 50,0, fill='', outline='red') 可以在 c 上绘制两个空的三角形以形成一个多边形，并将该多边形的句柄绑定到名称 x 上。然后，开发者可以使用下面这条语句将这个多边形填充为蓝色： *c*.itemconfig(x,fill='blue')

矩形画布项目

Canvas 实例 c 提供了一个创建矩形（rectangle）项目的方法。

表 17-17

create_rectangle	*c*.create_rectangle(*x0,y0,x1,y1,***rect_options*) 使用给定坐标 coordinates 指定的顶点创建一个矩形项目，并返回该项目的句柄。rect_option 可以包含： fill 矩形的内部颜色（默认值为空）； outline 矩形的边的颜色（默认值为黑色）； tags 可以是一个字符串，用来在项目上设置一个标签，或者是一个由字符串组成的元组，用来在项目上设置多个标签； width 线条的宽度，以像素为单位（默认值为 1）。

文本画布项目

Canvas 实例 c 提供了一个创建文本（text）项目的方法。

表 17-18

create_text	*c.create_text(x,y,**text_options)* 在给定的 x 和 y 坐标处创建一个文本项目，并返回该项目的句柄。text_option 可以包含： anchor 坐标 x 和 y 指定的文本限定框的确切位置；可以是 N、E、S、W、NE、NW、SE 或 SW——这些值都是指南针方向，用来指定限定框的角和边——或者 CENTER（默认值）； fill 文本的颜色（默认值为黑色）； font 文本使用的字体； tags 可以是一个字符串，用来在项目上设置一个标签，或者是一个由字符串组成的元组，用来在项目上设置多个标签； text 要显示的文本。

一个简单的绘图示例

下面的示例显示了如何使用 Canvas 执行一个基础绘图任务，也就是绘制一个用户指定的函数：

```
from Tkinter import *
import math

root = Tk()

# 首先，绘制一个行用于放置函数输入框和动作按钮
fram = Frame(root)
Label(fram,text='f(x):').pack(side=LEFT)
func = Entry(fram)
func.pack(side=LEFT, fill=BOTH, expand=1)
butt = Button(fram, text='Plot')
butt.pack(side=RIGHT)
fram.pack(side=TOP)

# 然后绘制另一个行以输入画布的范围
fram = Frame(root)
bounds = [ ]
for label in 'minX', 'maxX', 'minY', 'maxY':
    Label(fram,text=label+':').pack(side=LEFT)
    edit = Entry(fram, width=6)
```

```
        edit.pack(side=LEFT)
        bounds.append(edit)
    fram.pack(side=TOP)

    # 最后绘制画布
    c = Canvas(root)
    c.pack(side=TOP, fill=BOTH, expand=1)

    def minimax(values=[0.0, 1.0, 0.0, 1.0]):
        "Adjust and display X and Y bounds"
        for i in range(4):
            edit = bounds[i]
            try: values[i] = float(edit.get())
            except: pass
            edit.delete(0, END)
            edit.insert(END, '%.2f'%values[i])
        return values

    def plot():
        "Plot given function with given bounds"
        minx, maxx, miny, maxy = minimax()

        # 得到并编译函数
        f = func.get()
        f = compile(f, f, 'eval')

        # 获得画布的X和Y维
        CX = c.winfo_width()
        CY = c.winfo_height()

        # 计算线条的坐标
        coords = [  ]
        for i in range(0,CX,5):
            coords.append(i)
            x = minx + ((maxx-minx)*i)/CX
            y = eval(f, vars(math), {'x':x})
            j = CY - CY*(y-miny)/(maxy-miny)
            coords.append(j)

        # 绘制线条
        c.delete(ALL)
        c.create_line(*coords)

    butt.config(command=plot)

    # 给出一个初始函数示例，而不是一个文档
    f = 'sin(x) + cos(x)'
    func.insert(END, f)
    minimax([0.0, 10.0, -2.0, 2.0])

    root.mainloop()
```

图 17-2 显示了这个示例的输出结果。

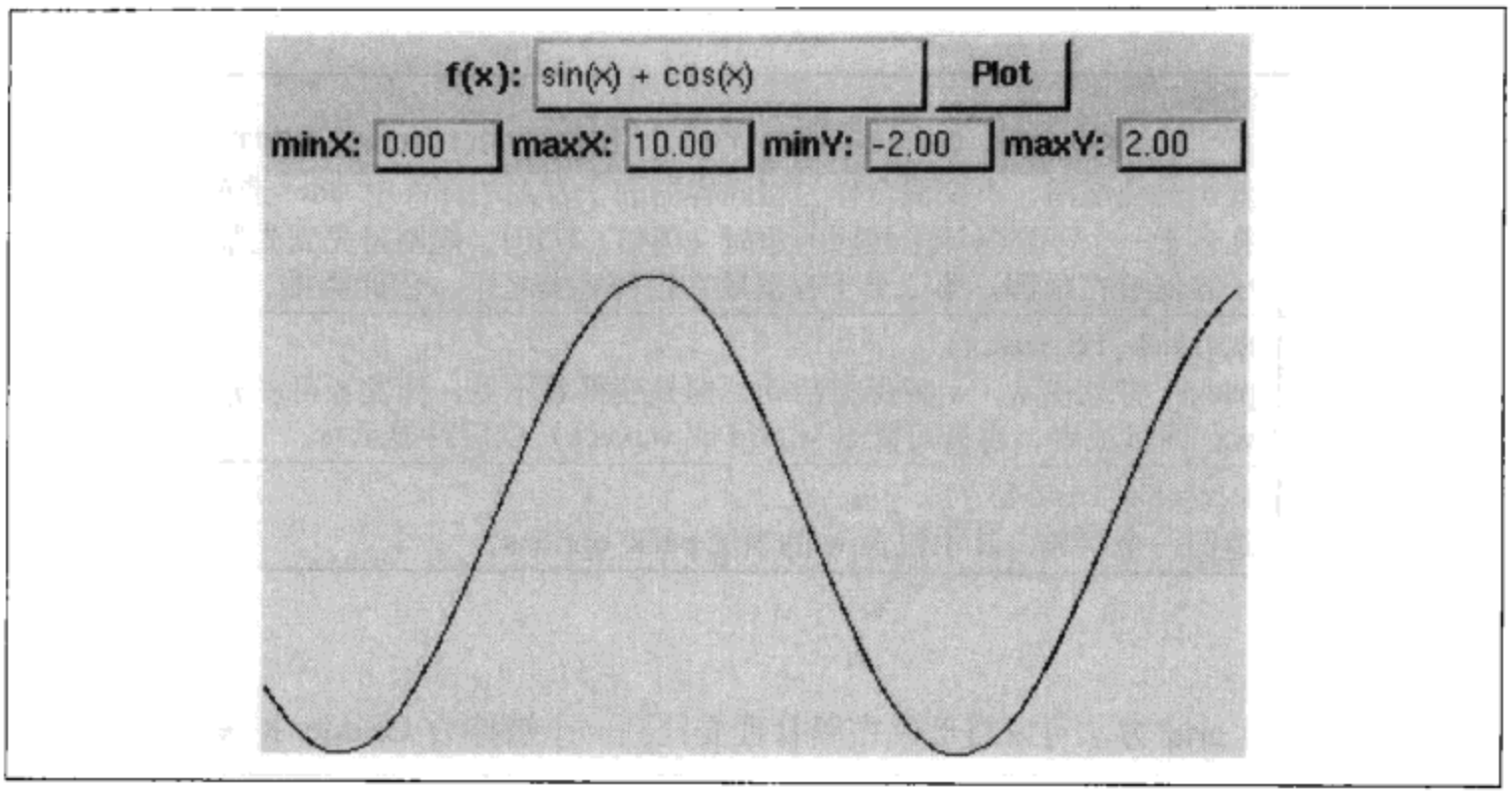


图 17-2　示例画布

17.8　布局管理

到现在为止，在所有示例中，大家已经知道可以通过 pack 方法让部件可见。这是典型的真实 Tkinter 用法。不过，有时候另外两个布局管理器也是很有用的。本节介绍了 Tkinter 提供的全部 3 个布局管理器：packer、gridder 和 placer。千万不要混合相同容器部件的布局管理器：给定容器部件的所有子部件必须由相同的布局管理器进行处理，否则将产生非常奇怪的效果（包括 Tkinter 进入无限循环）。

Packer

对一个部件调用 pack 方法可以将部件布局管理委托到一个被称为 Packer 的简单、灵活的布局管理器。Packer 可以根据每个部件的空间需要（包括 padx 和 pady）来调整容器部件（父部件）中的部件的大小和位置。每个部件 w 可以提供如表 17-19 所示这些与 Packer 相关的方法。

表 17-19

pack	`w.pack(**pack_options)` 将布局管理委托到 packer。pack_options 可以包括： expand 在为 True 时，w 将扩展为填充 w 的父部件中其他没有被使用的任意空格。 Fill 确定 w 是填充通过 packer 分配给 w 的任何额外空间，还是保持其自己的最小维度：NONE（默认值）、X（只水平填充）、Y（只垂直填充）或者 BOTH（同时水平和垂直填充）。

续表

pack	Side 确定父部件 w 靠近哪一边：TOP（默认值）、BOTTOM、LEFT 或 RIGHT。要想避免混淆，不要混合作为相同容器的子容器的部件中 side=选项的不同值。在多于一个子容器请求相同的边时（例如，TOP），规则是先到先服务：第一个子容器放在顶部，第二个子容器放在顶部容器之下，依此类推。
pack_forget	`w.pack_forget()` packer 将忽视 w。w 将保持活动，但是变得看不见；开发者可以通过再次调用 w.pack（或者，也有可能是 w.grid 或 w.place）以后再显示 w。
pack_info	`w.pack_info()` 返回一个字典，其中包含 w 的当前 pack_options。

Gridder

对一个部件调用 grid 方法可以将部件布局管理委托到一个被称为 Gridder 的专用布局管理器。Gridder 可以调整每个部件的大小和位置，并将每个部件放置到容器部件（父部件）中的一个表（网格）的单元格中。每个部件 w 提供了如表 17-20 所示的与 Gridder 相关的方法。

表 17-20

grid	`w.grid(**grid_options)`
	将布局管理委托到 gridder。grid_options 可以包括：
	column
	用来放置 w 的列；默认值为 0（最左边的列）；
	columnspan
	w 已经使用了多少列；默认值为 1；
	ipadx, ipady
	在 w 的边框之内水平和垂直填充 w 的像素数；
	padx, pady
	在 w 的边框之外水平和垂直填充 w 的像素数；
	row
	用来放置 w 的行；默认值是仍然保持空白的第一行；
	rowspan
	w 已经使用了多少行；默认值为 1；
	sticky
	如果单元格大于 w，需要怎么做？在默认情况下，如果 sticky=''，w 将在其单元格中居中。sticky 可能是零个或多个 N、E、S、W、NE、NW、SE 和 SW 的字符串串联，这些都是指南针方向，指定了 w 所在单元格的边和角。例如，sticky=N 表示 w 放在单元格的顶部，并且水平居中放置，而 sticky=N+S 表示 w 将垂直扩展到填充该单元格，并且水平居中。
	例如：
	`import Tkinter`
	`root = Tkinter.Tk()`
	`for r in range(3):`

续表

	for c in range(4): Tkinter.Label(root, text='R%s/C%s'%(r,c), borderwidth=1).grid(row=r,column=c) root.mainloop() 显示放置在 3×4 网格中的 12 个标签。
grid_forget	w.grid_forget() packer 将忽视 w。w 将保持活动，但是变得看不见；开发者可以通过再次调用 w.grid（或者，也有可能是 w.grid 或 w.place）以后再显示 w。
grid_info	w.grid_info() 返回一个字典，其中包含 w 的当前 grid_options。

Placer

对一个部件调用 place 方法可以显式处理部件布局管理，这要感谢被称为 Placer 的简单布局管理器。Placer 可以完全按照 w 显式要求的那样，调整容器部件（父部件）中的每个部件 w 的大小和位置。其他布局管理器通常更好使用，但是 Placer 可以帮助开发者实现自定义布局管理器。每个部件 w 提供了如表 17-21 所示的 Placer 相关方法。

表 17-21

place	w.place(**place_options) 将布局管理委托给 placer。place_options 可以包括： anchor w 的确切位置的其他选项包括：N、E、S、W、NE、NW、SE 或 SW，这些都是指南针方向，指示 w 的角和边；默认值为 NW（w 的左上角）。 bordermode 可以是 INSIDE（默认值），以指示与其父部件的内部相关的其他选项（忽略父部件的边）；否则为 OUTSIDE。 height width 高度和宽度，以像素为单位。 relheight relwidth 高度和宽度，可以是 0.0～1.0 的浮点型值，也可以是父部件的高度和宽度的分数值。 relx rely 水平和垂直偏移量，可以是 0.0～1.0 的浮点型值，也可以是父部件的高度和宽度的分数值。 x, y 水平和垂直偏移量，以像素为单位。
place_forget	w.place_forget() placer 将忽视 w。w 将保持活动，但是变得看不见；开发者可以通过再次调用 w.place（或者，也有可能是 w.pack 或 w.grid）以后再显示 w。
place_info	w.place_info() 返回一个字典，其中包含 w 的当前 place_options。

17.9 Tkinter 事件

到现在为止，我们只看到了一种类型的事件处理：使用按钮和菜单条目的 command= 选项对可调用集合执行回调函数。Tkinter 还可以将可调用对象设置为处理各种事件。Tkinter 不能创建自定义事件：开发者被限制为处理 Tkinter 本身预定义的事件。

事件对象

通用事件回调函数必须接受一个参数 event，这个参数是一个 Tkinter 事件对象。这样的事件对象包含以下几个用来描述该事件的属性：

char

表示一个键的代码的单字符字符串（只能用于键盘事件）；

keysym

一个字符串，表示一个键的符号名称（只能用于键盘事件）；

num

按钮编号（只能用于鼠标按钮事件）；从 1 开始编号；

x, y

相对于部件的左上角的鼠标位置，以像素为单位；

x_root

y_root

相对于屏幕的左上角的鼠标位置，以像素为单位；

widget

发生事件的部件。

将回调函数绑定到事件

要想将一个回调函数绑定到部件 w 中的一个事件上，调用 w.bind 并使用一个字符串来描述该事件，该事件通常被包含尖括号（'<...>'）中。下面的示例将在用户每次按 Enter 键的时候打印 'Hello World'：

```
from Tkinter import *

root = Tk()
def greet(*ignore): print 'Hello World'
root.bind('<Return>', greet)
root.mainloop()
```

Canvas 类和 Text 类的 tag_bind 方法（参见第 17.6 节中介绍的 tag_bind 方法和第 17.7 节中介绍的 tag_bind 方法）可以将事件回调函数绑定到由 Canvas 实例项目，或者由 Text 实例中的范围组成的集合上。

事件名称

几乎所有的通用事件名称都被包含在尖括号中，这些通用事件名称可以分为以下几类。

键盘事件

Key

用户单击了任意键。事件对象的属性 char 可以告诉开发者单击了哪个键，但仅限于普通键。字母和数字的 keysym 属性等于 char，标点符号的 keysym 属性等于该字符的名称，特殊键的 keysym 属性等于该键的名称。

“特殊键”

特殊键都有事件名称：F1、F2 以及后面的键都是功能键；Left、Right、Up 和 Down 都是方向键；Prior 和 Next 键是向上翻页和向下翻页键；Backspace、Delete、End、Home、Insert、Print 和 Tab 键都是按其名称标记的键；Escape 是标记为 Esc 的键；Return 是标记为 Enter 的键；Caps_Lock、Num_Lock 和 Scroll_Lock 都是锁请求键；而 Alt_L、Control_L、Shift_L 是修改符键 Alt、Ctrl 和 Shift（键盘的这些修改符键的多个实例之间没有什么区别）。所有这些名称都被放置在尖括号中，与几乎所有事件名称类似。

“普通键”

普通键都有不包含尖括号的事件名称——这些键也就是唯一缺少尖括号的键。每个普通键的事件名称都是该键的字符，比如‘w’、‘1’或‘+’。其中，例外的情况是空格键和小于符号键，空格键的事件名称为‘<space>’，小于符号键的事件名称为‘<less>’。

使用前缀‘Alt-’、‘Shift-’和‘Control-’可以对键的名称进行修改。在这种情况下，事件名称总是被包含在尖括号‘<...>’内；例如，‘<Control-Q>’和‘<Alt-Up>’。

鼠标事件

Button-1

Button-2

Button-3

用户单击了鼠标左键、中间键或右键。2-键鼠标只能产生事件 Button-1 和 Button-3，因为 2-键鼠标没有中间键。

B1-Motion

B2-Motion

B3-Motion

用户在按住鼠标左键、中间键或右键的同时移动了鼠标（不按住鼠标键的鼠标动作只能产生 Enter 和 Leave 功能）。

ButtonRelease-1

ButtonRelease-2

ButtonRelease-3

用户释放了鼠标左键、中间键或右键。

Double-Button-1

Double-Button-2

Double-Button-3

用户双击了鼠标左键、中间键或右键（在生成双击事件之前，这样的动作还将生成 Button-1、Button-2 或 Button-3 事件）

Enter

用户移动了鼠标，并由此进入了部件。

Leave

用于移动了鼠标，并由此离开了部件。

事件相关方法

每个部件 w 提供了如表 17-22 所示的事件相关方法。

表 17-22

bind	*w.bind(event_name,callable[,'+'])* w.bind(event_name,callable)可以将 callable 设置为 w 上的 event_name 的回调函数。w.bind(event_name,callable,'+')可以将 callable 添加到 w 的 event_name 以前的绑定上。
bind_all	*w.bind_all(event_name,callable[,'+'])* w.bind_all(event_name,callable)可以将 callable 设置为所有部件上的 event_name 的回调函数。 w.bind_all(event_name,callable,'+')可以将 callable 添加到所有部件上的 event_name 以前的绑定上。
unbind	*w.unbind(event_name)* 删除 w 上的 event_name 的所有回调函数。

续表

unbind_all	w.unbind_all(*event_name*) 删除任意部件上的 event_name 的所有回调函数，就像 bind_all 设置的那样。

事件示例

下面的示例可以使用 bind_all 检测键盘和鼠标事件：

```
import Tkinter
from Tkinter import *

root = Tk()
prompt='Click any button, or press a key'
L = Label(root, text=prompt, width=len(prompt))
L.pack()

def key(event):
    if event.char==event.keysym:
        msg ='Normal Key %r' % event.char
    elif len(event.char)==1:
        msg ='Punctuation Key %r (%r)' % (event.keysym, event.char)
    else:
        msg ='Special Key %r' % event.keysym
    L.config(text=msg)
L.bind_all('<Key>', key)

def do_mouse(eventname):
    def mouse_binding(event):
        L.config(text='Mouse event %s' % eventname)
    L.bind_all('<%s>'%eventname, mouse_binding)
for i in range(1,4):
    do_mouse('Button-%s'%i)
    do_mouse('ButtonRelease-%s'%i)
    do_mouse('Double-Button-%s'%i)

root.mainloop()
```

其他与回调函数相关的方法

每个部件 w 都提供了如表 17-23 所示其他与回调函数相关的方法。

表 17-23

after	w.after(*ms,callable,*args*) 启动一个定时器以从当前时间开始调用 callable(*args)函数 ms 毫秒。返回一个 ID，开发者可以将该 ID 传递到 after_cancel 以取消该定时器。定时器是一次性的：为了周期性地调用一个函数，函数本身必须调用 after 以再次调用该函数本身。

续表

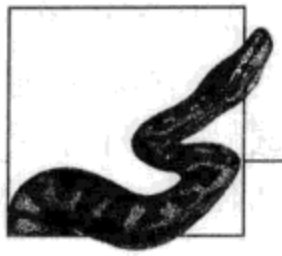
after_cancel	*w*.after_cancel(*id*) 取消 id 标识的定时器。
after_idle	*w*.after_idle(*callable*,**args*) 将一个回调函数注册到 callable(*args)，该回调函数将在事件循环空闲时执行（也就是，在所有等待事件已经被处理完之后）。

下面的示例使用了 after 以实现一个简单数字时钟：

```
import Tkinter
import time

curtime = ''
clock = Tkinter.Label()
clock.pack()

def tick():
    global curtime
    newtime = time.strftime('%H:%M:%S')
    if newtime != curtime:
        curtime = newtime
        clock.config(text=curtime)
    clock.after(200, tick)
tick()
clock.mainloop()
```

after 方法是至关重要的。许多部件都不包含回调函数以让开发者知道用户对这些部件的动作；要想跟踪这些动作，轮询是唯一的选择。例如，下面是如何使用 after 轮询以实时跟踪一个 Listbox 选择的例子：

```
import Tkinter

F1 = Tkinter.Frame()
s = Tkinter.Scrollbar(F1)
L = Tkinter.Listbox(F1)
s.pack(side=Tkinter.RIGHT, fill=Tkinter.Y)
L.pack(side=Tkinter.LEFT, fill=Tkinter.Y, yscrollcommand=s.set)
s['command'] = L.yview
for i in range(30): L.insert(Tkinter.END, str(i))
F1.pack(side=Tkinter.TOP)

F2 = Tkinter.Frame()
lab = Tkinter.Label(F2)
def poll():
    lab.after(200, poll)
    sel = L.curselection()
    lab.config(text=str(sel))
lab.pack()
F2.pack(side=Tkinter.TOP)

poll()
Tkinter.mainloop()
```



第 18 章

测试、调试和最优化

完成了编写代码并不意味着已经完成了编程任务；只有在代码运行正确，并且具有可以接受的性能时才算完成了任务。测试（参见第 18.1 节）表示通过在已知条件下运行代码，并检查结果是否为期望值来验证代码的运行是否正确。调试（参见第 18.2 节）表示查找错误行为的原因并修改错误（只要找到了错误的原因，修改通常是很简单的）。

最优化（参见第 18.4 节）通常可以作为用来确保可接受的性能的行动的总称。最优化可以分解为基准测试（benchmarking）（给定任务的可测量性能，可以检测该任务是否在可接受的范围之内）、性能测试（profiling）（使用工具检测程序以标识性能瓶颈）和正确地最优化（消除瓶颈让总的程序性能可以被接受）。很清楚，开发者只有在已经找到了性能瓶颈的位置之后（使用性能测试）才能消除性能瓶颈，而要想找到性能瓶颈，需要知道程序有哪些性能问题（使用基准测试）。

本章将按照以下 3 个主题在开发过程中出现的自然顺序对其进行介绍：最开始，也最重要的就是测试，然后是调试，最后是最优化。不过，大多数程序员的兴趣主要集中在最优化上：测试和调试通常被认为是繁杂琐事，而最优化被认为是非常有趣的（从本书的观点看，这样的认识是错误的）。因此，如果开发者只打算阅读本章的某一节，建议阅读第 18.4 节，这一节总结了 Python 化的最优化方案——接近于 Jackson 的经典名言“最优化规则：规则 1：不要这样做。规则 2（仅限于专家）：仍然不要这样做。”

所有这些任务都是巨大和重要的，并且每个任务本身至少需要一本书来介绍。本章甚至没有打算比较近距离地介绍每个相关的技术和涉及的知识；本章主要介绍了 Python 的专用技术、方法和工具。

18.1 测试

在本章中，要对两种完全不同类型的测试进行区分：单元测试（unit testing）和系统测

试（system testing）。测试是一个丰富和重要的领域，可以描绘出更多的区别，但是本节主要关注的是对软件开发人员而言当前最重要的一些问题。许多开发人员并不情愿在测试上花费时间，将其视作要从“真正”开发时间中除去的时间，但是，这种态度是目光短浅的：越早找到错误，就越容易修改，因此，花费在开发测试上的每个小时都可能由于尽快地找到了错误而得到充分的回报，从而节省大量的调试时间，如果不这样做，会大大增加软件开发周期的后面阶段所需的调试时间。

单元测试和系统测试

单元测试就是编写和运行测试以试验单个模块或甚至更小的单元，比如一个类或一个函数。系统测试（也被称为功能或集成测试）包括使用已知的输入运行整个程序。有些关于测试的经典书籍也描述了白盒（white-box）测试和黑盒（black-box）测试，白盒测试是指在了解程序内部结构情况下的测试，而黑盒测试时在不了解程序内部结构情况下的测试。这种经典的观点与现代意义上的单元测试和系统测试是平行的，但并不是完全地复制。

单元测试和系统测试服务于不同的目标。单元测试可以加快开发的进程；开发者可以并且必须在开发每个单元的时候对其进行测试。一种现代的方法被称为“测试驱动开发”（test-driven development，TDD）：对于程序必须具有的每个功能，首先应该编写单元测试，然后再开始编写实现该功能的代码。TDD 看起来好像是顺序颠倒的，但是这种方法具有几个好处。例如，TDD 可以确保开发者不会忽略某些功能的单元测试。进一步讲，首先开发测试代码也是很有用的，因为这样做可以激励开发者首先关注特定函数、类或方法必须完成的任务，然后才处理如何实现这些函数、类或方法。沿着 TDD 的开发思路，最近出现的一个非常重要的创新就是“行为驱动开发”（behavior-driven development）；参见 http://behavior-driven.org/以了解有关这种新的和富有前景的开发方法的所有详细信息。

为了测试一个单元（这个单元可能依赖于尚未完全开发完成的其他单元），开发者通常必须编写桩（stub），桩是各种单元接口的仿真实现，在需要这些单元接口测试其他单元时，这些接口可以给出已知的和正确的响应。Mock 模块（http://python-mock.sourceforge.net/）对于实现许多这样的桩是非常有帮助的（http://xper.org/wiki/seminar/PythonMockObjectTutorial 上提供了很好的指南，其中包含许多有关 Mock 对象的其他有用文档的链接）。

在单元测试之后就是系统测试，因为系统测试要求系统已经存在，并且可以相信系统功能的某些子集处于工作状态。系统测试提供了一种完整性的检测：在给定程序中的每个模块都工作正常时（通过了单元测试），整个程序工作正常吗？如果每个单元都是正常的，但是作为作为一个整体的系统运行不正常，这说明单元之间的集成有问题，也就是单元合作的方式有问题。出于这个原因，系统测试也被称为集成测试（integration test）。

系统测试类似于在产品使用情况下运行系统，区别在于提前确定了输入信息，因此系统运行中发现的任何问题都很容易重现。系统测试中失败的代价要小于产品使用中失败的代价，因为系统测试的输出没有用于做决定、服务于客户、控制外部系统等。更正确地讲，系统测试的输出是与系统在给定的已知输入下应该产生的输出之间的系统比较。其目的是要以一种廉价和可重复的方式找到程序应该要做的事情和程序实际所做的事情之间的任何差异。

系统测试发现的错误，就像产品使用中的系统错误一样，可能会暴露单元测试中的一些错误，以及代码中的错误。单元测试可能是不充分的：模块的单元测试可能无法测试该模块需要的所有功能。在这种情况下，单元测试显然需要被加强。开发者必须在更改代码以解决问题之前完成单元测试的功能增强，然后运行新增强的单元测试以确认单元测试现在仍可以显示存在的错误。然后解决问题并再次运行单元测试以确认单元测试不再显示任何错误。最后，重新运行系统测试以确认该问题已经真正被解决。

更常见的是，系统测试中的错误会暴露开发团队之间的信息交流问题：一个模块正确地实现了某个特定功能，但是另一个模块期望的是另一个功能。这种类型的问题（严格意义上讲，这是一个集成问题）在单元测试中难以查明。在好的开发经验中，必须经常运行单元测试，因此单元测试是否能快速运行是至关重要的。因此，本质上每个单元可以假定其他单元按照期望的那样正确工作。

如果系统架构是层次结构的（一种通用和合理的组织结构），在开发过程中适当往后的阶段运行的单元测试可以暴露集成的问题。在这样的体系架构中，底层模块不依赖于其他模块（除了有可能依赖库模块之外，可以假定库模块是正确的），因此如果完成了这样的底层模块的单元测试，足以确保底层模块的正确。高层模块依赖于底层模块，也依赖于正确理解每个模块期望和提供的功能。因此，开发者应该使用真正的底层模块而不是桩对高层模块运行完整的单元测试可以检测模块之间的接口，以及高层模块自身的代码。

因此，高层模块的单元测试以两种方式运行。开发者可以在开发的早期，也就是在底层模块还没有就绪的时候，或者在开发的晚期，也就是在只需要检查高层模块的正确性的时候使用底层模块的桩来运行单元测试。在开发的后期，开发者还需要使用真正的底层模块定期运行高层模块的单元测试。按照这种方法，开发者可以检查整个子系统的正确性，从高层到底层。然而，即使是在最顺利的情况下，开发者仍需要编写和运行系统测试以确信整个系统的功能是可试验和可检测的，并且不应该忽视模块之间的接口。

系统测试类似于按正常方式运行程序。开发者所需的特殊支持只是确保提供了已知的输入，并且可以捕获输出以和期望的输出进行比较。这对于可以对文件执行 I/O 的程序来讲是很简单的，但是对于 I/O 依赖于 GUI、网络或其他与独立外部实体的通信的程序而言，这是非常困难的。要想仿真这样的外部实体并让这些实体可预测和完全可见，

通常需要用到平台相关的体系结构。支持系统测试体系结构的另外一个非常有用的部分就是测试框架（testing framework），测试框架可以自动运行系统测试，包括记录成功和失败的日志。这样的框架还可以帮助测试人员准备已知输入和对应的期望输出的数据集合。

当前存在用于测试目的的免费和商用程序，但是这些程序并不依赖于被测系统中使用的编程语言。系统测试非常类似于经典的，被称为黑盒测试的测试方法，也就是独立于被测系统的实现的测试（特别是，因此也独立于用于实现被测系统的编程语言）。与此相反，测试框架通常取决于测试系统运行的操作系统平台，因为测试框架执行的任务都是平台相关的：使用给定的输入运行程序、捕获其输出，特别是仿真和捕获 GUI、网络和其他进程间通信 I/O。因为系统测试的框架取决于平台，而不是编程语言，本书没有进一步对此进行介绍。

doctest 模块

doctest 模块的主要目的是让开发者在代码的文档字符串中创建好的使用示例，这是通过检查这些示例实际产生的结果，并在文档字符串中显示该结果来实现的。doctest 可以通过在文档字符串中查找交互式 Python 提示符‘>>>’、然后是在相同行上的 Python 语句，以及下一行上该语句的期望输出来识别这些示例。

在开发一个模块时，需要保持文档字符串的更新，并且逐步使用示例来丰富文档字符串的内容。每当一个模块的部分（例如，一个函数）已经就绪，甚至只是部分就绪时，要习惯于将示例添加到文档字符串。将该模块导入到一个交互式会话中，然后交互式使用开发者刚开发的部分，这样可以提供由典型情况、限制情况和错误情况组成的混合示例。在只需要这种特殊目的时，可以使用 from module import *，这样，示例就不会将 module.作为该模块提供的每个名称的前缀了。将交互式会话的文本复制和粘贴到编辑器中的文本字符串中，修改所有错误，然后就基本上完成了。

现在，开发者的文档就增加了示例功能，读者就可以比较容易地按照示例进行开发了，假定开发者选择了一个好的混合示例，并使用非示例文本对其进行了很好的说明。请确信为整个模块，以及该模块导出的每个函数、类和方法准备了包含示例的文档字符串。开发者可以选择跳过一些名称以下划线（_）开始的函数、类和方法，因为，按照其名称指示的，这些对象是私有实现的；doctest 默认会忽略这些对象，而且模块的大多数读者也会这样做。

与代码工作的方式不匹配的示例要比无用的示例更坏。文档和注释只有在匹配实际情况时才有用。文档字符串和注释通常会随着代码的更改而过时，然后，过时的文档字符串和注释往往会误导和妨碍，而不是帮助读者了解模块的功能。不包含任何注释和文档字符串也要比包含错误的信息更好一些。doctest 可以帮助开发者测试文档字符串中的示例。doctest 运行失败会提示开发者检查包含错误示例的文档字符串，还可以提

醒开发者保持文档字符串的文本的更新。

在模块源代码的末尾，可以插入下面这段代码：

```
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

这段代码可以在开发者将该模块作为主程序运行时调用 doctest 模块的 testmod 函数。testmod 将检查所有相关文档字符串（docstring 模块，以及所有公共函数的文档字符串；公共类，以及公共类的公共函数）。在每个文档字符串中，testmod 将查找所有示例（通过查找出现的解释器提示符‘>>>’，再加上一个空格）并运行每个示例。testmod 可以检查每个示例的结果是否等于该示例之后的文档字符串中给出的结果。在出现异常的情况下，testmod 将忽略跟踪信息，而是只检查期望的错误消息与出现的错误消息是否相等。

在所有的运行都正确时，testmod 将终止运行。否则，testmod 将输出有关失败的示例的详细信息，显示期望的和实际的输出。示例 18-1 显示了一个在 mod.py 模块上执行 doctest 的典型示例。

示例 18-1　使用 doctest

```
"""
This module supplies a single function reverseWords that reverses a string
by words.

>>> reverseWords('four score and seven years')
'years seven and score four'
>>> reverseWords('justoneword')
'justoneword'
>>> reverseWords('')
''

You must call reverseWords with one argument, and it must be a string:

>>> reverseWords()
Traceback (most recent call last):
  ...
TypeError: reverseWords() takes exactly 1 argument (0 given)
>>> reverseWords('one', 'another')
Traceback (most recent call last):
  ...
TypeError: reverseWords() takes exactly 1 argument (2 given)
>>> reverseWords(1)
Traceback (most recent call last):
  ...
AttributeError: 'int' object has no attribute 'split'
>>> reverseWords(u'however, unicode is all right too')
```

```
u'too right all is unicode however,'

As a side effect, reverseWords eliminates any redundant spacing:

>>> reverseWords('with redundant spacing')
'spacing redundant with'

"""
def reverseWords(astring):
    words = astring.split()
    words.reverse()
    return ' '.join(words)

if __name__=='__main__':
    import doctest, sys
    doctest.testmod(sys.modules[__name__])
```

在这个模块的文档字符串中，截短了文档字符串中的跟踪信息，并将其替换为省略号：这是一种好的习惯做法，因为 doctest 可以忽略跟踪信息，并且不会向每个错误用例添加任何说明信息。除了这段代码之外，文档字符串就是交互式会话上的内容的复制和粘贴，还有就是为了可读性而添加的一些说明文本和空行。将这段代码源代码保存为 mod.py，然后使用 python mod.py 运行该代码。运行后不会产生任何错误输出，这表示所有的示例工作正常。尝试使用 python mod.py –v 以获得尝试的所有测试的数量，末尾还有一段冗长的总结。最后，可以在该模块的文档字符串中修改示例的结果，让结果错误，这样可以看到 doctest 为错误的示例提供的消息。

尽管 doctest 最初并没有打算用于通用目的单元测试，但是其仍然是用于该目的的一种非常方便的工具。Python 中推荐的用来执行单元测试的方法是使用 unittest 模块，参见第 18.1 节。但是，使用 doctest 的单元测试可以更简单和更快地建立，因为 doctest 只要求从交互式会话复制和粘贴即可。如果开发者需要维护一个缺少单元测试的模块，使用 doctest 将各种测试放到该模块中是一个合理的折中方案。包含基于 doctest 的单元测试显然要比完全没有任何单元测试更好一些，还有一种情况开发者可能会决定使用 doctest 单元测试，那就是使用 unitest“正确地”建立测试需要花费开发者太长的时间。

如果开发者已经决定了使用 doctest 进行单元测试，请不要将另外的测试添加到模块的文档字符串中。这样做可能会损害该文档字符串，使得其太长，并且难以读懂。请在文档字符串中保持适当数量和类型的示例，确实要用于说明该模块的目的，就像不是完全为了单元测试一样。此外，将另外的测试放到模块的全局变量中，也就是一个名为__test__的字典中。__test__中的键是被用作任意测试名称的字符串，而对应的值就是 doctest 将会提取，并按照使用文档字符串的相同方法使用的字符串。__test__中的值还可能是函数和类对象，在这种情况下，doctest 将检查其文档字符串以运行测试。后面的这种功能是在具有私有名称的对象上运行 doctest 的一种很便利的方法，而在默认情况下，doctest 将跳过具有私有名称的对象。

在 Python 2.4 中，doctest 模块还提供了两个函数，可以返回基于 doctest 的 unittest.TestSuite 类的实例，这样开发者就可以将这样的测试集成到基于 unittest 的测试框架中了。网站 http://docs.python.org/lib/doctest-unittest-api.html 提供了这种高级功能的文档。

unittest 模块

unittest 模块是 Python 版本的单元测试框架，最初是由 Kent Beck 为 Smalltalk 开发的。同样地，许多其他编程语言（例如，Java 的 JUnit 包）中也存在该框架的广泛传播的版本，这些版本通常被统称为 xUnit。

要想使用 unittest，开发者不需要将测试代码放到与测试模块相同的源代码文件中，而是为要测试的每个模块编写单独的测试模块。一个流行的习惯用法是将该测试模块的名称命名为被测模块的名称，并使用一个像‘test_’这样的前缀，然后将其放在用于保存源代码的目录下的一个名为 test 的子目录中。例如，mod.py 的测试模块可以是 test/test_mod.py。开发者需要使用一种简单和一致的命名惯例让自己可以简单地编写和维护辅助脚本，这些脚本可以用来查找和运行一个软件包的所有单元测试。

模块的源代码和其单元测试代码的分离可以帮助开发者更容易地重写该模块，包括使用 C 语言来重新编写其功能，而这样做不会影响单元测试代码。不管开发者对 mod.py 进行了哪些更改，都可以确信 test_mod.py 保持不变，这样可以增强开发者的信心，也就是，通过了 test_mod.py 中的测试表示 mod.py 在被更改之后仍然可以正确工作。

单元测试模块定义了 unittest 的 TestCase 类的一个或多个子类。每个子类通过定义“测试用例方法”（test-case method）指定了一个或多个测试用例，测试用例方法是可以不带任何参数调用的方法，并且方法的名称以 test 开始。子类也有可能覆盖 setUp 方法，框架将调用该方法以在调用每个测试用例方法之前准备一个新实例，还有可能覆盖 tearDown 方法，框架将调用该方法以在调用每个测试用例方法之后清理所有内容。为了表达测试必须满足的条件，每个测试用例方法都可以调用名称以 assert 开始的 TestCase 类的方法。unittest 可以按任意顺序在一个 TestCase 子类中运行测试用例方法，每个方法都是在该子类的一个新实例上运行的，在每个测试用例之前运行 setUp，在每个测试用例之后运行 tearDown。

unitest 还提供了其他一些工具（比如将测试用例组合成测试套件）和其他一些更高级的功能。除非开发者正在定义一个自定义单元测试框架，或者至少也是在为非常复杂的包构建同样复杂的测试流程，开发者不需要用到这些额外的功能。在几乎所有情况下，本节中介绍的概念和详细知识足够执行有效和系统级的单元测试了。示例 18-2 显示了如何使用 unittest 为示例 18-1 中的 mod.py 模块提供单元测试的例子。出于例证的目的，这个示例使用 unittest 来执行与示例 18-1 中使用 doctest 测试文档字符串中的示例完全相同的测试。

示例 18-2　使用 unittest

```
""" This module tests function reverseWords provided by module mod.py. """
import unittest
import mod

class ModTest(unittest.TestCase):

    def testNormalCase(self):
        self.assertEqual(mod.reverseWords('four score and seven years'),
            'years seven and score four')

    def testSingleWord(self):
        self.assertEqual(mod.reverseWords('justoneword'), 'justoneword')

    def testEmpty(self):
        self.assertEqual(mod.reverseWords(''), '')

    def testRedundantSpacing(self):
        self.assertEqual(mod.reverseWords('with redundant spacing'),
            'spacing redundant with')

    def testUnicode(self):
        self.assertEqual(mod.reverseWords(u'unicode is all right too'),
           u'too right all is unicode')

    def testExactlyOneArgument(self):
        self.assertRaises(TypeError, mod.reverseWords)
        self.assertRaises(TypeError, mod.reverseWords, 'one', 'another')

    def testMustBeString(self):
        self.assertRaises((AttributeError,TypeError), mod.reverseWords, 1)

if __name__=='__main__':
    unittest.main()
```

使用 python test_mod.py 运行这个脚本要比使用 python mod.py 运行 doctest（参见示例 18-1）稍微冗长一些。test_mod.py 将为其运行的每个测试用例方法输出一个点号符号（.），然后是一个破折号分隔行，最后是一个汇总信息行，比如“Ran 7 tests in 0.110s”，如果所有测试都通过，则最后一行为“OK”。

每个测试用例方法都会对名称以 assert 开始的方法（或者其名称以 fail 开始的同义词）进行一个或多个调用。这里，testExactlyArgument 方法是唯一一个包含两个这样的调用的方法。在更复杂的情况下，从单个测试用例方法对 assert 方法执行多个调用是非常普通的。

即使是在与这个示例一样简单的情况下，从小的方面来看，对于单元测试，unittest 要比 doctest 功能更强大，也更灵活。在 testMustBeString 方法中，该示例将一对异常类传递为 assertRaises 方法的第一个参数，表示该示例接受任何一种类型的异常。因此，

test_mod.py 可以被认为是 mod.py 的多个有效实现。unittest 可以接受示例 18-1 中的实现，该实现尝试对其参数调用 split 方法，并因此在调用一个不是字符串的参数时引发 AttributeError 错误。但是，unittest 还可以接受一种不同的假定实现，这种实现会在使用一个错误类型的参数调用时引发 TypeError 错误。开发者还可以使用 doctest 来编写这种测试功能，但是可能会很难使用或者被禁止的，而 unittest 可以让单元测试更简单，也更自然。

这种灵活性对于真实的单元测试而言是至关重要的，从某种程度上讲，真实的单元测试都是其模块的可执行规范。开发者可能会悲观地将对灵活性的需要看作是表示正在测试的代码并没有很好地定义接口。但是，最好将接口看作是已经被实现者可用的灵活性实现了：在环境 X 下（在这个示例中，具有无效类型的参数被传递到 reverseWords 函数中），这两种事情（引发 AttributeError 或 TypeError 错误）都允许发生。

这样，这两种方式的实现都是正确的，并且实现者可以在对性能和清晰度进行考虑的基础上在两者之间进行选择。通过将单元测试看作其模块的可执行规范（现代的观点，也是测试优先开发的基础），而不是将其看作严格限定为一种特定实现的白盒测试（在某些传统的测试分类中），测试成为软件开发过程的一个更至关重要的组件。

TestCase 类

使用 unittest，开发者可以通过创建 TestCase 类的子类并添加方法以编写测试用例，这些方法可以不带任何参数调用，并且其名称以 test 开始。这样的测试用例方法将按顺序轮流调用开发者的子类从 TestCase 继承的方法以指定测试继续执行必须满足的条件，继承的方法的名称是以 assert 开始的（或者其同义词，其名称以 fail 开始）。

TestCase 类还定义了另外两个方法，开发者的子类可以有选择地覆盖这两个方法以将动作组合在一起，并在每个测试用例方法运行之前或之后执行。这样做并没有完全用完 TestCase 的功能，但是，除非开发者在开发测试框架或者执行某些类似的高级任务，开发者并不需要用到 TestCase 的其余功能。TestCase 实例 t 中最常调用的方法如表 18-1 所示。

表 18-1

assert_, failUnless	`t.assert_(condition,msg=None)` 如果 condition 为 False，将会失败并输出 msg；否则，不执行任何操作。名称中的下划线是必需的，因为 assert 是 Python 关键字。failUnless 是一个同义词。在开发者可以使用更专用的方法时，比如 assertEqual，不要使用这些方法。
assertAlmostEqual, failUnlessAlmostEqual	`t.assertAlmostEqual(first,second,places=7,msg=None)` 如果 first!=second，将会失败并输出 msg，msg 不超过 places 位小数；否则，不执行任何操作。failUnlessAlmostEqual 是一个同义词。几乎总是这样，在开发者比较的是浮点型数字时，这些方法要比 assertEqual 更适合使用。

续表

assertEqual, failUnlessEqual	t.assertEqual(*first*,*second*,*msg*=None) 如果 first!=second，将会失败并输出 msg；否则，不执行任何操作。failUnlessEqual 是一个同义词。
assertNotAlmostEqual, failIfAlmost-Equal	t.assertNotAlmostEqual(*first*,*second*,*places*=7, *msg*=None) 如果 first==second 时，将会失败并输出 msg，msg 不超过 places 个小数；否则，不执行任何操作。failIfAlmostEqual 是一个同义词。
assertNotEqual, failIfEqual	t.assertNotEqual(*first*,*second*,*msg*=None) 如果 first==second，将会失败并输出 msg；否则，不执行任何操作。failIfEqual 是一个同义词。
assertRaises, failUnlessRaises	t.assertRaises(*exceptionSpec*,*callable*,**args*, ***kwargs*) 调用 callable(*args,**kwargs)。如果该调用不引发任何异常，则失败。如果该调用引发一个不满足 exceptionSpec 的异常，assertRaises 将传播该异常。如果该调用引发一个满足 exceptionSpec 的异常，assertRaises 将不执行任何操作。exceptionSpec 可以是一个异常类或者是一个由类组成的元组，就像 try/except 语句中的 except 子句的第一个参数一样。failUnlessRaises 是一个同义词。
fail	t.fail(*msg*=None) 无条件失败并输出 msg。
failIf	t.failIf(*condition*, *msg*=None) 如果 condition 为 True，将会失败并输出 msg；否则，不执行任何操作。
setUp	t.setUp() 框架将在调用一个测试用例方法之前调用 t.setUp( )。TestCase 中的实现不执行任何操作。提供这个方法的目的是，如果需要为每个测试执行某些准备，则可以让开发者的子类覆盖该方法。
tearDown	t.tearDown() 框架将在调用一个测试用例方法之后调用 t.tearDown( )。TestCase 中的实现不执行任何操作。提供这个方法的目的是，如果需要在每个测试之后执行某些清除操作，则可以让开发者的子类覆盖该方法。

单元测试处理大量数据

单元测试必须是很快的，因为在开发过程中需要经常运行单元测试。因此，在可能的时候，最好使用少量数据对模块的功能的每个方面进行单元测试。这样做可以让每个单元测试更快，并且可以让开发者很方便地将所有需要的数据嵌入到测试的源代码中。特别是，在测试一个从文件对象读取或写入到文件对象的函数时，开发者通常需要使

用 cStringIO 类的一个实例（参见第 10.5 节）来模仿一个文件对象，而将数据保存在内存中；这种方法要比写入到磁盘更快一些，并且可以避免在测试运行之后必须删除磁盘文件的麻烦。

但是，在少数情况下，如果不提供和/或比较大量数据（比可以适当地嵌入到一个测试的源代码更多的数据），则不可能完整地测试一个模块的功能。在这样的情况下，单元测试必须依赖于辅助的外部数据文件以保存需要提供给被测模块的数据和/或需要与被测模块的输出进行比较的数据。尽管这样，通常最好将数据读入到 cStringIO 的实例中，而不是直接让被测模块执行实际的磁盘 I/O。更重要的是，本书强烈建议开发者通常使用桩来测试打算与其他外部实体互操作的模块，比如数据库、GUI 或网络上的其他程序。在使用桩，而不是真实的外部实体时，开发者可以更容易控制测试的所有方面。还有，需要反复强调的是，单元测试的运行速度是很重要的，并且，在桩中执行仿真操作总是要比执行真实操作更快一些。

18.2 调试

因为 Python 的开发周期是如此之快，最有效的调试方法通常就是编辑代码，并让代码在关键位置输出相关信息。Python 具有很多方法让代码探测其自身的状态，从而提取可能与调试相关的信息。inspect 和 traceback 模块特别提供了这样的探测功能，这也被称为反射和自测。

一旦开发者已经获得了调试相关信息，print 语句通常就是显示信息的最简单方法。开发者还可以将调试信息记录到日志文件。日志对于长时间自动运行的程序来说是特别有用的，比如服务器程序。显示调试信息就像显示其他类型的信息一样，参见第 10 章和第 17 章。通过日志记录这样的信息非常类似于写入到文件（参见第 10 章），或者持久化信息，参见第 11 章；但是，要想有助于日志的特殊任务，Python 的标准库还提供了一个 logging 模块，参见第 6.6 节中介绍的 logging 模块。参见第 8.3 节中介绍的 excepthook 属性，重新绑定 sys 模块的 excepthook 属性可以让开发者的程序在被一个广播异常终止之前记录详细的错误信息。

Python 还提供了一些可以启用交互式调试的钩子函数。pdb 模块提供了一个简单的文本模块交互式调试器。Python 的其他交互式调试器都是集成开发环境（IDE）的一部分，比如 IDLE 和各种商用的集成开发环境。但是，本书没有介绍 IDE。

在调试之前

在开发者开始进行可能比较冗长的调试探测之前，请确信开发者已经使用第 3 章中提到的工具彻底检查了 Python 源代码。这些工具可以只捕获代码中的错误的一个子集，但是这些工具要比交互式调试快得多，因此，使用这些工具本身就已经足够了。

此外，再次需要说明的是，在开始一个调试会话之前，请确信包含的所有代码都在单元测试的覆盖之内，参见第 18.1 节。一旦开发者发现了一个错误，在修改该错误之前，应该将一个或两个测试添加到开发者的单元测试套件中，或者，如果需要，还可以添加到系统测试套件中（如果从一开始就提供了这些测试，这些测试可能已经发现了这个错误），并再次运行测试以确认找到并隔离了错误；只有在完成了这个操作之后，开发者可以开始修改这个错误。通过定期地遵循这个错误处理过程，开发者将很快拥有一个好得多的测试套件，学习编写更好的测试，并获得巨大的有关代码全局正确性的保证。

请记住，即使使用 Python 提供的所有工具，其标准库，以及开发者喜欢的任何 IDE，调试仍然是很困难的。甚至应该在开发者开始设计和编码之前考虑到这个事实：编写和运行大量的单元测试，并保持开发者的设计和代码简单，这样就可以将开发者可能需要的调试的数量减少到绝对的最小值！在这一点上，最经典的建议就是 Brian Kernighan 说过的下面这段话："首先，调试要比编写代码困难两倍。因此，即使你可以尽可能聪明地编写代码，但是，按照定义，你还是没有足够的聪明来调试代码的。"

inspect 模块

inspect 模块提供了一些函数以从所有类型的对象获得信息，包括 Python 函数调用栈（该堆栈将记录当前执行的所有函数调用）和源代码。inspect 最常使用的函数如表 18-2 所示。

表 18-2

getargspec, formatargspec	`getargspec(f)`
	f 是一个函数对象。getargspec 将返回一个由 4 个项目组成的元组：(arg_names, extra_args, extra_kwds, arg_defaults)。arg_names 是由 f 的参数的名称组成的序列。extra_args 是形式为*args 的特殊参数的名称，如果 f 没有这些参数，则 extra_args 为 None。extra_kwds 是形式为**kwds 的特殊参数的名称，如果 f 没有这些参数，则 extra_kwds 为 None。arg_defaults 是由 f 的参数的默认值组成的元组。开发者可以从 fgetargspec 的结果推断 f 的签名其他细节：f 具有 len(arg_names)-len(arg_defaults)个强制参数，并且 f 的可选参数的名称都是字符串，也就是列表切片 arg_names[-len(arg_defaults):]中的项目。
	formatargspec 可以接受 1～4 个与 getargspec 返回的元组中的项目相同的参数，还将返回一个包含这个信息的字符串。因此，formatargspec(*getargspec(f))将返回一个在圆括号中包含 f 的参数的字符串（也就是，f 的签名），正如创建 f 的 def 语句中使用的那样。例如：
	`import inspect`
	`def f(a,b=23,**c): pass print inspect.formatargspec`
	`(*inspect.getargspec(f))`
	`# 输出：(a, b=23, **c)`

续表

getargvalues, formatargvalues	getargvalues(*f*)
	f 是一个框架对象——例如，调用 sys 模块中的_getframe 函数（参见第 8.3 节中介绍的_getframe 函数）或调用 inspect 模块中的 currentframe 函数的结果。getargvalues 将返回一个由 4 个项目组成的元组：(arg_names, extra_args, extra_kwds, locals)。arg_names 是由 f 的函数的参数名称组成的序列。extra_args 是形式为*args 的特殊参数的名称，如果 f 的函数没有这样的参数，则 extra_args 为 None。extra_kwds 是形式为**kwds 的特殊参数的名称，如果 f 的函数没有这些参数，则 extra_kwds 为 None。locals 是一个由 f 的本地变量组成的字典。特别是，因为参数都是本地变量，因此可以通过使用该参数对应的参数名称来索引 locals 字典以从 locals 中获得每个参数的值。
	formatargspec 可以接受 1～4 个与 getargspec 返回的元组中的项目相同的参数，还将返回一个包含这个信息的字符串。因此，formatargspec (*getargspec(f))将返回一个在圆括号中包含 f 的参数的字符串（也就是，f 的签名），正如创建 f 的 def 语句中使用的那样。例如：
	`def f(x=23): return inspect.currentframe()`
	`print inspect.formatargvalues(inspect.getargvalues(f()))`
	`# 输出: (x=23)`

currentframe	currentframe() 返回当前函数(currentframe 的调用者)的框架对象。例如，formatargvalues (getargvalues(currentframe( ))) 将返回一个包含调用函数的参数的字符串。
getdoc	getdoc(*obj*) 返回 obj 的文档字符串，同时将制表符扩展为空格，并从每个行中去除多余的空白字符。
getfile, getsourcefile	getfile(*obj*) 返回定义了 obj 的文件的名称，在不能确定该文件时，将引发 TypeError 错误。例如，如果 obj 是内置对象，getfile 将引发 TypeError。getfile 可以返回一个二进制或源文件的名称。getsourcefile 可以返回一个源文件的名称，并在找到的所有文件都是二进制文件，而不是对应的源文件时引发 TypeError 错误。
getmembers	getmembers(*obj, filter*=None) 返回 obj 的所有属性（成员），包括数据和方法（包括特殊方法），返回值是一个由(name, value)数据对组成的排序列表。在 filter 不为 None 时，只返回在对属性的值调用可调用函数 filter 时，使得 filter 的值为 True 的属性，例如： `sorted((n, v) for n, v in getmembers(obj) if filter(v))`
getmodule	getmodule(*obj*) 返回定义了 obj 的模块对象，如果无法确定哪个模块对象定义了 obj，则返回 None。

续表

getmro	getmro(*c*)
	返回一个由类 c 中的基类和父类组成的元组，按方法解析顺序排列。c 是该元组中的第一个项目。每个类在该元组中只出现一次。例如：
	`class oldA: pass`
	`class oldB(oldA): pass`
	`class oldC(oldA): pass`
	`class oldD(oldB,oldC): pass`
	`for c in inspect.getmro(oldD): print c.__name__,`
	`# 输出: oldD oldB oldA oldC`
	`class newA(object): pass`
	`class newB(newA): pass`
	`class newC(newA): pass`
	`class newD(newB,newC): pass`
	`for c in inspect.getmro(newD): print c.__name__,`
	`# 输出: newD newB newC newA object`

getsource, getsourcelines	getsource(*obj*) 返回一个多行字符串，这个字符串是 obj 的源代码，如果不能确定或提取该源代码，则引发 IOError。getsourcelines 可以返回一个数据对：第一个项目是 obj 的源代码（一个代码行列表），而第二个项目是第一行的行号。
isbuiltin, isclass, iscode, isframe, isfunction, ismethod, ismodule, isroutine	isbuiltin(*obj*) 所有这些函数都可以接受单个参数 obj，并且，如果 obj 属于该函数名中指示的类型，则返回 True。可以接受的对象分别是：内置（C 语言编码）函数、类对象、代码对象、框架对象、Python 编码的函数（包括 lambda 表达式）、方法、模块，对于 isroutine，可接受的对象包括所有方法或函数，不管是 C 语言编码的还是 Python 编码的。这些函数通常被用作 getmembers 的 filter 参数。
stack	stack(*context*=1) 返回一个由 6 项目元组组成的列表。第一个元组是有关 stack 的调用者的信息，第二个元组是有关调用者的调用者的信息，依此类推。每个元组的项目按顺序分别是：框架对象、文件名、行号、函数名、由当前行前后的 context 源代码行组成的列表，以及该列表中当前行的索引。

一个使用 inspect 的示例

假定开发者在程序的某个位置执行了一行像下面这样的语句：

```
x.f()
```

但是出乎意料地收到了一个 AttributeError 错误以通知开发者对象 x 没有名为 f 的属性。这意味着对象 x 并不像开发者期望的那样，因此，开发者需要确定更多有关 x 的信息，并通过这些信息探知为什么 x 会引发错误，以及该如何对其进行处理。开发者可以将这行语句更改为：

```
try: x.f()
except AttributeError:
```

```
    import sys, inspect
    print>>sys.stderr, 'x is type %s, (%r)' % (type(x), x)
    print>>sys.stderr, "x's methods are:",
    for n, v in inspect.getmembers(x, callable):
        print>>sys.stderr, n,
    print>>sys.stderr
    raise
```

这个示例使用了 sys.stderr（参见第 8.3 节中介绍的 stdin、stdout 和 stderr），因为 sys 模块的这些属性显示了与错误有关的信息，而不是程序的结果。inspect 模块的 getmembers 函数获得了 X 上的所有可用的方法的名称，其目的是为了显示这些信息。如果开发者经常需要这种类型的诊断功能，可以将其打包到一个单独的函数中，比如：

```
import sys, inspect
def show_obj_methods(obj, name, show=sys.stderr.write):
    show('%s is type %s(%r)\n'%(name,obj,type(obj)))
    show("%s's methods are: "%name)
    for n, v in inspect.getmembers(obj, callable):
        show('%s '%n)
    show('\n')
```

然后，这个示例就可以简单地写成：

```
try: x.f()
except AttributeError:
    show_obj_methods(x, 'x')
    raise
```

不管是在打算用于诊断和调试目的的代码中，还是在实现程序功能的代码中，好的程序结构和组织形式都是必不可少的。参见第 6.6 节中介绍的__debug__内置变量以了解一个可以在定义诊断和调试函数时使用的好技术。

traceback 模块

traceback 模块可以用来提取、格式化和输出有关跟踪消息的信息，通常是由未被捕获的异常产生的。在默认情况下，traceback 模块可以重新构造 Python 用于跟踪信息的格式。但是，traceback 模块还可以让开发者对跟踪信息实现更细粒度地控制。traceback 模块提供了许多函数，但是，通常开发者只需要用到其中的一个。

表 18-3

print_exc	print_exc(*limit*=None, *file*=sys.stderr) 从一个异常处理程序或一个函数直接，或者由一个异常处理程序间接调用 print_exc。print_exc 可以将 Python 为未被捕获的异常输出到 stderr 的跟踪信息输出到类文件对象 file 中。在 limit 不为 None 时，print_exc 只输出 limit 个跟踪嵌套级别。例如，在一个异常处理程序中，当开发者想要捕获一个像传播的异常一样的诊断信息，但是实际上阻止了异常的进一步传播（这样，开发者的程序将保持运行，并且不会引入其他的处理程序）时，可以调用 traceback.print_exc( )。

pdb 模块

pdb 模块利用了 Python 解释器的调试和跟踪钩子函数以实现一个简单的面向命令行的交互式调试器。pdb 可以设置断点、源代码级单步调试、检查堆栈结构等。

要想在 pdb 的控制之下运行某些代码，可以导入 pdb，然后调用 pdb.run，并将代码字符串作为单个参数传递到 pdb.run 以执行该代码。要想使用 pdb 进行代码终止后的调试（也就是通过在交互式提示符上传播一个异常以调试刚刚终止的代码），可以不带任何参数调用 pdb.pm( )。在 pdb 开始运行时，将首先读取主目录和当前目录下名为.pdbrc 的文本文件。这样的文件可以包含任意 pdb 命令，但是，这些文件通常都使用别名命令，这是为了为其他命令定义有用的同义词和缩写。

在 pdb 开始控制代码的运行时，将使用字符串‘(Pdb)’作为提示符以供开发者输入 pdb 命令。help 命令（也可以输入该命令的缩写形式 h）列出了所有可用的命令。带一个参数调用 help（以一个空格隔开）可以获得有关任何特殊命令的帮助。开发者可以将大多数命令缩写为前一个或两个字母，但是，必须总是以小写字母输入命令：pdb，就像 Python 本身一样，是区分大小写的。输入一个空行将重复前一个命令。最常使用的 pdb 命令如下。

表 18-4

!	`! statement` 在当前调试环境中执行 Python 语句 statement。
alias, unalias	`alias [ name [ command ] ]` 不带任何参数 alias 可以列出当前定义的所有别名。alias name 可以输出别名 name 当前的定义。在 alias 命令的完整形式下，command 是任意带参数的 pdb 命令，还可以包含%1、%2 等参数以引用传递给被定义的新别名 name 的特殊参数，或者包含%*以一起引用所有这样的参数。unalias name 命令可以删除一个别名。
args, a	`args` 列出传递给当前正在调试的函数的所有实际参数。
break, b	`break [ location [ ,condition ] ]` 不带任何参数调用 break 可以列出当前定义的断点和每个断点已经触发的次数。在带一个参数调用 break 时，将在给定 location 设置一个断点。location 可以是一个行号或者一个函数名，还可以选择在前面带上 filename：在当前文件之外的另一个文件中设置一个断点，或者在一个不知道名称的函数的起始位置设置一个断点（也就是，一个存在于多个文件中的函数）。在给出了 condition 的值时，这个值是一个需要计算的表达式（在调试的上下文环境下），每次在给定行或函数将要执行时都需要对该表达式进行计算；只有在该表达式返回一个 True 值时，才执行断点操作。在设置一个新断点时，break 将返回一个断点编号，然后开发者可以使用该断点编号以在任意其他与断点相关的 pdb 命令中引用这个新断点。
clear, cl	`clear [ breakpoint-numbers ]` 清除（删除）一个或多个断点。不带参数调用 clear 时，在请求确认之后，clear 将删除所有断点。要想去活一个断点而不是删除该断点，参见下面介绍的 disable 命令。

续表

condition	`condition breakpoint-number [ expression ]` condition n expression 可以设置或更改断点 n 的条件。不带 expression 参数调用 condition n 可以让断点 n 无条件执行。
continue, c, cont	`continue` 继续执行正在被调试的代码，直到一个断点（如果有的话）。
disable	`disable [ breakpoint-numbers ]` 禁用一个或多个断点。不带任何参数调用 disable 将禁用所有断点（在请求确认之后）。这个命令不同于 clear 命令，区别在于调试器将记住被禁用的断点，开发者还可以使用 enable 重新激活这些断点。
down, d	`down` 在堆栈中向下移动一个位置（也就是，向着最近的函数调用移动）。通常，堆栈中的当前位置位于底部（也就是，位于最近被调用，并且当前正在被调试的函数）。因此，down 命令不能进一步向下移动了。但是，如果开发者之前执行了 up 命令（该命令可以向上移动当前位置），则 down 命令是很有用的。
enable	`enable [ breakpoint-numbers ]` 启用一个或多个断点。不带任何参数调用 enable 可以在请求确认之后启用所有断点。

ignore	`ignore breakpoint-number [ count ]`
	设置断点的忽略计数（如果没有提供 count，则默认值为 0）。触发一个忽略计数大于 0 的断点只会将次数减 1。在触发一个忽略计数为 0 的断点时，程序的执行将会停止，并显示一个交互式 pdb 提示符。例如，假定 fob.py 模块包含下面的代码：
	`def f():`
	`  for i in range(1000):`
	`  g(i)`
	`def g(i):`
	`  pass`
	现在，考察下面的交互式 pdb 会话（该会话在 Python 2.4 中运行；根据开发者运行的不同 Python 版本，少数细节有所不同）：
	`>>> import pdb`
	`>>> import fob`
	`>>> pdb.run('fob.f()')`
	`> <string>(1)?()`
	`(Pdb) break fob.g`
	`Breakpoint 1 at C:\mydir\fob.py:5`
	`(Pdb) ignore 1 500`
	`Will ignore next 500 crossings of breakpoint 1.`
	`(Pdb) continue`
	`> C:\mydir\fob.py(5)g()`
	`-> pass`
	`(Pdb) print i`
	`500`
	正如 pdb 所打印的，ignore 命令要求 pdb 忽略断点 1 上的接下来 500 个触发，该示例在前面的断点语句中在 fob.g 处设置了断点 1。因此，在执行最终停止时，函

续表

ignore	数 g 已经被调用了 500 次，正如该示例显示的，打印参数 i 时，这个值现在确实是 500。断点 1 的忽略计数现在是 0；如果再执行一次 continue 并打印 i，i 将显示为 501。换句话讲，一旦忽略计数减少到 0，将在每次触发断点时停止函数 g 的执行。如果想要跳过更多个断点触发，必须向 pdb 提供另一个 ignore 命令，并再次将断点 1 的忽略计数设置为某个大于 0 的值。
list, l	`list [ first [ , last ] ]` 不带任何参数调用 list 可以列出以当前行为中心的 11 行代码，如果前一个命令也是 list，则列出接下来的 11 行代码。list 命令的参数可以有选择地指定当前文件中要列出的第一行和最后一行。list 命令将列出物理行而不是逻辑行，也就是，包括注释行和空白行。
next, n	`next` 执行当前行，但是不单步进入从当前行调用的任何函数。但是，从当前行直接或间接调用的函数中触发的断点将停止执行。
print, p	`p expression` 在当前上下文环境中计算 expression，并显示其结果。
quit, q	`quit` 立即终止 pdb 和正在被调试的程序。
return, r	`return` 执行当前函数的其余部分，只在断点处停止（如果有的话）。
step, s	`step` 执行当前行，单步进入从当前行调用的任何函数。
tbreak	`tbreak [ location [ ,condition ] ]` 与 break 类似，但是 tbreak 断点是临时的（也就是，pdb 将在这个断点被触发之后自动删除该断点）。
up, u	`up` 在堆栈中向上移动一个位置（也就是，远离最近的函数调用，向着正在调用的函数）。
where, w	`where` 显示堆栈中的记录，并指示当前的位置（也就是，当前的上下文环境命令!执行语句、args 命令显示参数、print 命令计算表达式等所在的位置）。

Debugging in IDLE

IDLE 是 Python 附带的“交互式开发环境”（Interactive DeveLopment Environment），提供了类似于 pdb 的调试功能，但是不如 pdb 的功能那么强大。感谢 IDLE 的 GUI，这样可以很容易地访问 Python 调试功能。例如，开发者只需要激活 Debug Control 窗口中的 4 个复选框中的一个或多个，就可以看到，在程序每一步运行时，源代码、堆栈、本地变量和全局变量的信息总是会分别显示在相同的窗口中，开发者不必使用像 list 和 where 这样的 pdb 命令来显式请求源代码列表和堆栈列表。

要想运行 IDLE 的交互式调试器，可以使用 IDLE 的“Python Shell”窗口中的 Debug→Debugger。IDLE 将打开 Debug Control 窗口，在 shell 窗口中输出[DEBUG ON]，并在 shell 窗口中给出另一个>>>提示符。可以继续按照平常那样使用 shell 窗口；开发者现在在 shell 窗口的提示符下输入的任何命令都在调试器下运行。要想去活调试器，可以再次使用 Debug→Debugger；然后，IDLE 将切换调试状态，关闭 Debug Control 窗口，并在 shell 窗口中输出[DEBUG OFF]。要想在调试器活动时控制该调试器，可以在 Debug Control 窗口中使用 GUI 控制。开发者只能在调试器没有忙于活动地跟踪代码时切换调试器的状态；否则，IDLE 将禁用 Debug Control 窗口中的 Quit 按钮。

18.3 warnings 模块

警告是有关错误或异常的消息，这些错误或异常通常并没有严重到破坏程序的控制流的程度（否则会引发一个普通异常）。warnings 模块会对产生警告的操作，以及对警告执行什么操作提供细粒度的控制。开发者可以通过调用 warnings 模块中的 warn 函数有条件地输出一个警告。模块中的其他函数可以用来控制警告的格式、设置其目的地，并有条件地阻止某些警告（或者将某些警告转换为异常）。

类

warnings 模块提供了几个可以显示警告的异常类。Warning 类是 Exception 类的一个子类，也是所有警告的基类。开发者可以定义自己的警告类；这些类必须是 Warning 类的子类，可以是 Warning 类的直接子类，或者是其他已有子类的子类，这些已有的子类是：

DeprecationWarning

使用过时的功能，提供这个功能只是用于向后兼容；

RuntimeWarning

使用语义有错误趋势的功能；

SyntaxWarning

使用语法有错误趋势的功能；

UserWarning

其他不匹配以上这些情况的用户自定义警告。

对象

Python 没有提供具体的警告对象。警告是由 message（文本字符串）、category（Warning 类的子类）和两段可以标识引发警告的位置的信息：module（引发该警告的模块的名称）和 lineno（引发该警告的源代码行的行号）。从概念上讲，开发者可以将这些信息

看作是警告对象 w 的属性，为了更清楚，本书后面将把这些信息作为属性使用，但是，实际上并不存在特定的警告对象 w。

筛选器

在任何时候，warnings 模块为警告保持了一个活动筛选器列表。在程序运行中第一次导入 warnings 模块时，该模块将检查 sys.warnoptions 以确定筛选器的初始集合。开发者可以带选项-W 运行 Python 以为一个给定的程序运行操作设置 sys.warnoptions。不要依赖于被特别保存在 sys.warnoptions 中的初始筛选器集合，因为这是一个有关实现的问题，可能会在 Python 的将来版本中被更改。

在每次出现警告 w 时，warnings 将根据每个筛选器测试 w，直到匹配某个筛选器。第一个匹配的筛选器确定了警告 w 会发生什么事情。每个筛选器是一个由 5 个项目组成的元组。第一个项目 action 是一个字符串，定义了匹配筛选器的警告发生了什么事情。另外 4 个项目 message、category、module 和 linenon 可以控制 w 匹配筛选器的方式，并且必须满足一个匹配的所有条件。下面是这些项目的含义（使用属性记法来表示 w 的概念上的属性）。

message

一个正则表达式对象；匹配条件是 message.match(w.message)（匹配操作是不区分大小写的）。

category

Warning 或 Warning 的一个子类；匹配条件是 issubclass(w.category,category)。

module

一个正则表达式对象；匹配条件是 module.match(w.module)（匹配操作是区分大小写的）。

lineno

一个整数；匹配条件是 lineno in (0,w.lineno)。lineno 为 0，表示 w.lineno 没有关系，或者，w.lineno 必须等于 lineno。

在一个匹配中，筛选器的第一个字段 action 确定了要发生的事情：

'always'

不管 w 是否已经出现，都输出 w.message。

'default'

当且仅当 w 第一次从这个特定位置（也就是，特殊的 w.module、w.location 数据对）出现时，输出 w.message。

'error'

将 w.category(w.message)作为异常引发。

'ignore'

w 被忽略。

'module'

当前仅当 w 是第一次从 w.module 出现时，输出 w.message。

'once'

当且仅当 w 是第一次从任何位置出现时，输出 w.message。

函数

warnings 模块提供了如表 18-5 所示函数。

表 18-5

filterwarnings	`filterwarnings(action,message='.*',category=Warning, module='.*',lineno=0, append=False)` 向活动筛选器列表添加一个筛选器。在 append 为 True 时，filterwarnings 将在所有其他已有筛选器后面添加该筛选器（也就是，将该筛选器添加在已有的筛选器列表的后面）；否则，filterwarnings 将在任意其他已有筛选器之前插入该筛选器。除 action 之外的所有组件都有表示“匹配任何警告”的默认值。正如前面详细介绍的，message 和 module 都是正则表达式的模式字符串，category 是 Warning 的某个子类，lineno 是一个整数，而 action 是一个字符串，用来确定在一个消息匹配这个筛选器时将发生什么事情。
formatwarning	`formatwarning(message,category,filename,lineno)` 返回一个以标准格式表示给定警告的字符串。
resetwarnings	`resetwarnings()` 从筛选器列表中删除所有筛选器。resetwarnings 还将丢弃最初与-W 命令行选项一起添加的任何筛选器。
showwarning	`showwarning(message,category,filename,lineno,file=sys.stderr)` 将给定警告输出到给定文件对象。输出警告的筛选器动作将调用 showwarning，从而让 file 参数默认为 sys.stderr。要想更改在筛选器动作输出警告时发生的事情，可以使用这个签名编写开发者自己的函数，并将其绑定到 warnings.showwarning，这样将覆盖默认的实现。
warn	`warn(message,category=UserWarning,stacklevel=1)` 发送一个警告，这样，筛选器将检查该警告，并有可能输出该警告。如果 stacklevel 为 1，这个警告的位置是当前函数（warn 的调用者），如果 stacklevel 为 2，则警告的位置是当前函数的调用者。因此，将 2 传递为 stacklevel 的值，

续表

warn	可以让开发者编写根据其调用者的行为发送警告的函数，比如：
	`def toUnicode(astr):`
	`  try:`
	`  return unicode(astr)`
	`  except UnicodeError:`
	`  warnings.warn("Invalid characters in (%s)" %astr, stacklevel=2)`
	`  return unicode(astr, errors='ignore')`
	感谢参数 stacklevel=2，示例中的警告看起来来自于 toUnicode 的调用者，而不是来自于 toUnicode 本身。这在匹配这个警告的筛选器的 action 为 default 或 module 时是非常重要的，因为这些动作只有在该警告第一次从给定位置或模块出现时才输出一个警告。

18.4 最优化

“首先让程序工作。然后让程序正确。再然后让其更快”这句话被广泛认为是“程序设计的黄金准则”，这句话通常还有其他一些不同的说法。据本书考证，这句话是由 Kent Beck 提出的，但是他将其归功于他的父亲。被广泛所认知并没有减少这个原则的重要，特别是因为打破这个原则会比遵守这个原则得到更大的尊敬。与之相反，但是稍微有些言过其实的另一句话是 Don Knuth 所讲的（他将此归功于 Hoare）：“过早的最优化是程序中的所有错误之源”。

如果开发者的代码还不能工作，或者说，如果开发者还不确信代码应该确切做什么（因此，开发者也不能确信代码是否正常工作），最优化还言之过早。首先应该让代码可以工作。如果代码已经开始工作，但是开发者并不十分满意其总体结构和设计，则最优化也为时尚早。在考虑最优化之前，请先修正结构上的缺陷：首先让程序工作，然后让程序正确。前面的这两个步骤并不是可选项；可工作，结构化很好的代码永远都是必要选项。

相反，开发者并不总是需要让程序运行得更快。在前两个步骤之后，测试基准可能会显示开发者的代码的性能已经是可以接受的了。在性能不可接受时，性能测试（profiling）通常显示，所有的性能问题通常只在于一小部分代码，开发者的程序可能在 10%到 20%的代码上花费了 80%或 90%的时间。代码中的这些对性能起到关键性作用的区域通常被称为瓶颈（bottleneck）或者热点（hot spot）。最优化大部分代码而获得假定 10%的程序运行时间往往是浪费精力。即使开发者可以让这部分代码的运行再快 10 倍（极其了不起的成就），但是程序的总体运行时间可能只降低了 9%，甚至没有用户会注意到这样的速度加快。如果最优化是必需的，请将注意力集中在可能会产生问题的瓶颈上。只要保持代码 100%的纯 Python 化，就可以对瓶颈进行最优化，同时也不

会妨碍将来把代码移植到其他 Python 实现中。在某些情况下，开发者可以采取将某些与计算相关的瓶颈重新编码为 Python 扩展（参见第 25 章“扩展和嵌入经典 Python”），可能会获得甚至更好的性能（但是有可能会对将来的可移植性造成影响）。

开发一个足够快的 Python 应用程序

从在 Python 中设计、编码和测试应用程序开始，尽量使用可用的扩展模块，如果这些模块确实能节省开发者的时间。这些扩展模块要比使用经典的可编译语言花费少得多的时间。然后，通过对应用程序进行基准测试以确定结果代码是否足够快。通常的情况是，程序的运行已经达到了要求。恭喜！发布程序！

因为大量的 Python 程序本身都是使用高度优化的 C 语言编写的，例如许多 Python 标准和扩展模块，开发者的应用程序可能甚至已经要比典型的 C 语言代码更快一些。但是，如果应用程序还是太慢，开发者需要重新检查算法和数据结构。查找瓶颈需要了解应用程序体系结构、网络流量、数据库访问和操作系统的相互操作。对于典型的应用程序，这些因素中的每一个都要比语言的选择更有可能导致程序的运行速度降低。修正大尺度的结构性问题往往可以戏剧性地加快一个应用程序的运行，而 Python 是这种体验的一个完美的媒介。

如果开发者的程序仍然很慢，需要对其进行性能测试以找出时间都用在哪些操作上了。应用程序通常会表现出计算瓶颈：通常大约 10%到 20%的小范围的源代码花费了 80%或者更多的运行时间。然后，就可以应用本章的其余部分建议的技术再对瓶颈进行最优化了。

如果正规的 Python 级别的最优化仍然会留下一些显著的计算瓶颈，开发者可以使用 Python 扩展模块重新进行编码，参见第 25 章。最后，在大尺度测试帮助开发者找到了一个更好的体系结构时，应用程序的运行速度大致相当于全部使用 C、C++或 Fortran 编程的速度——甚至更快一些。使用这个流程产生的总的编程效率并不少于全部使用 Python 编码得到的效率。因为开发者使用 Python 来表示程序的总体结构，并使用底层的、难以维护的语言来处理少数特殊的计算瓶颈，因此将来的更改和维护都是很简单的。

只要开发者按照这个流程在给定区域构建应用程序，就可以积累一个可重用的 Python 扩展模块库。因此，开发者会在开发相同领域的其他快速运行 Python 应用程序方面变得越来越有效率。

即使外部限制最终迫使开发者使用底层语言来重新编写整个应用程序，开发者仍然最好从 Python 开始。长久以来，快速原型已经被公认为是获得正确的软件体系结构的最好方法。一个可以工作的原型能够帮助开发者确认已经标识了所有的问题，并确认采用了一条好的路径来实现其解决方案。原型还提供了对性能而言非常重要的大尺度体系结构试验的类型。使用 Python 开始原型可以使用扩展模块的方式逐步移植到其他语

言。这样，应用程序在每个阶段都可以保持完整的功能，并且可测。这样可以确保在编码阶段防止出现危害设计的体系结构完整性的风险。这样产生的软件要比从头开始的所有编码都采用底层语言的软件更快，也更健壮，并且，开发者的效率尽管不如纯的 Python 应用程序那样好，但仍然要比全部采用底层语言进行编码要更高。

基准测试

基准测试（Benchmarking）也被称为负荷测试（load testing）类似于系统测试：这两种测试都更像是为了产品目的而运行程序。在这两种测试情况下，至少需要程序想要实现的功能中的某些子集已经可以工作了，并且需要使用已知的、可重现的输入数据。对于基准测试，开发者不需要捕获和检查程序的输出：因为开发者需要在让程序更快之前让程序可以正确工作，因此在开发者对程序进行负荷测试时，应该已经对程序的正确性完全自信了。开发者需要输入有代表性的典型系统操作，比较理想的是那些会对程序的性能最有挑战的输入数据。如果程序可以执行几种类型的操作，请确定为每个不同类型的操作运行不同的基准测试。

使用手表来测量运行的时间可能就足够精确地基准测量大多数程序了。具有严格实时限制的程序显然是另一个问题，但是这些程序的需求非常不同于大多数情况下的普通程序。除了那些具有非常特殊限制的程序之外，5%或 10%的性能区别不会对程序的真实可用性产生实际的区别。

在为了帮助开发者选择一个算法或数据结构而对“玩具”程序或程序片段进行基准测试时，可能需要更高的精确度：Python 标准库中的 timeit 模块（参见第 18.4 节中介绍的 timeit 模块）非常适合于这样的任务。本节介绍的基准测试属于不同的类型：大致上相当于真实程序的操作，其唯一目的就是，在开始性能测试和其他最优化操作之前，检查一个程序的每个任务的性能都是可接受的。对于这样的系统基准测试，创造一个近似于程序正常运行条件的环境是最好的，而时间上的高度精确性并不是特别重要。

大尺度最优化

对于性能而言，程序中最重要的方面就是大尺度问题：算法的选择、总体结构的选择和数据结构的选择。

开发者必须经常考虑的性能问题就是那些与计算机科学中传统的大写 O 符号相关的问题。通俗地讲，如果将 N 作为一个算法的输入大小，并使用大写 O 符号表示算法的性能，对于 N 的较大值，可以将性能表示为 N 的某些函数的比例（按精确的计算机科学术语讲，可以将其称为大写 Theta，但是在实际使用中，大多数程序员将其称为大写 O，可能是因为希腊字母中的大写字母 Theta 看起来像中间有一个句点符号的字母 O!）。

O(1)算法（也被称为“常数时间”算法）是一种花费的时间不随 N 而增长的算法。O(N)算法（也被称为“线性时间”算法）表示，对于足够大的 N，处理 2 倍的数据需要花

费大约 2 倍的时间，而处理 3 倍的数据则需要花费 3 倍的时间，依此类推，按比例增加到 N 倍。O(N^2)算法（也被称为“平方时间”算法）表示，对于足够大的 N，处理 2 倍的数据需要花费大约 4 倍的时间，而处理 3 倍的数据需要花费 9 倍的时间，依此类推，按比例增加到 N 的平方。相同的概念和记法也被用于描述一个程序消耗的内存（“空间”）而不仅仅是时间。

在任何一本有关算法和数据结构的优秀书籍中，开发者会发现更多有关大 O 符号的信息。不幸的是，在编著本书的时候，还没有任何一本这样的书使用到 Python。但是，如果开发者熟悉 C 语言，本书建议开发者阅读 O'Reilly 出版，Kyle Loudon 编著的 *Mastering Algorithms with C* 一书。

要想理解大 O 符号在程序中的实际重要性，可以考察下面给出的从一个输入可迭代对象接受所有项目，并按相反的顺序将这些项目聚合到一个列表中的两种不同方法：

```
def slow(it):
    result = []
    for item in it: result.insert(0, item)
    return result

def fast(it):
    result = []
    for item in it: result.append(item)
    result.reverse()
    return result
```

实际上可以采用更简明的方式表示上面这两个函数，但是关键的区别最好通过展示以基本术语编写的函数来表现。slow 函数通过在所有以前接收到的项目之前插入每个输入项目来建立结果列表。fast 函数将每个输入项目添加到所有以前接收到的项目之后，最后再反转结果列表。直观地讲，大家可能会认为最后的反转操作表示一个额外的工作，因此，slow 应该要比 fast 更快一些。但是，这并不是计算速度的方法。

不管有多少个项目已经在 result 列表中，每次调用 result.append 都会花费大约相同的时间，因为在列表的末尾总有一个空闲位置可以用于一个额外的项目（比较学术的说法是，append 操作的时间复杂度为 O(1)，不过本书并没有介绍相关的原理）。fast 函数中的 for 循环执行了 N 次以接收 N 个项目。因为这个循环的每个迭代操作都花费一个固定长度的时间，因此总的循环时间为 O(N)。result.reverse 花费的时间也是 O(N)，因为该操作直接与项目的总数成正比。这样，fast 的总的运行时间为 O(N)（如果开发者不理解为什么两个数量 O(N)相加的结果还是 O(N)，可以这样认为，N 的任意两个线性函数的和仍然是一个 N 的线性函数——而且，“时间复杂度为 O(N)”与“消耗的时间量是 N 的线性函数”具有完全相同的含义）。

与此相反，每次调用 result.insert 时，必须将已经在 result 中的所有项目向后移动一个位置，从而为要插入的新项目创建一个空间。这个操作花费的时间与已经在列表中的项

目的数量成正比。因此，接收 N 个项目花费的时间总数与 1+2+3+…+N−1 之和（也就是，时间复杂度为 $O(N^2)$）成正比。因此，slow 函数总的运行时间为 $O(N^2)$。

将一个时间复杂度为 $O(N^2)$ 的解决方案替换为一个时间复杂度为 O(N) 的解决方案几乎总是有价值的，除非开发者出于某种原因为输入大小 N 指定了一个非常小的限定值。如果 N 可以不受非常严格的边界限制而增加，则对于足够大的值 N，$O(N^2)$ 解决方案将不可避免地要比 O(N) 解决方案慢得多，不管这两种情况下的比例常数为多少（不管使用什么样的性能测试工具，都会得到相同的结果）。除非在其他地方有不能消除的其他 $O(N^2)$，甚至更坏的瓶颈，程序中时间复杂度为 $O(N^2)$ 的一部分代码将不可避免地成为程序的瓶颈，对于足够大的值 N，这部分代码将占用主要的运行时间。开发者要帮自己一个忙，密切关注大 O 问题：与此相比，其他的所有性能问题几乎都是无关紧要的。

顺便提一下，通过使用更合乎 Python 习惯的表达式，fast 函数甚至可以变得更快。只需要将前两行代码替换为下面这行代码即可：

```
result = list(it)
```

这个更改不会影响 fast 的大 O 符号（在更改代码后，fast 的时间复杂度仍然是 O(N)），但是将 fast 的运行速度提高了一个很大的常数因子。在 Python 中，通常最简单、最清楚和最合乎语言习惯的表达方式往往也是最快的。

在 Python 和在任何其他语言中，选择具有好的大-O 特性的算法大致上都是相同的工作。实际上，开发者只需要了解一些 Python 基础模块的大-O 性能就可以了，本书在下面几节对此进行了介绍。

列表操作

Python 列表内部是使用向量（也被称为动态数组）实现的，而不是“链接的列表”。 从大-O 的角度而言，这种最基本的实现选择确定了 Python 列表的全部性能特性。

链接长度为 N1 和 N2 的两个列表 L1 和 L2（也就是，L1+L2）的时间复杂度为 O(N1+N2)。将一个长度为 N 的列表 L 乘以整数 M（也就是，L*M）的时间复杂度为 O(N*M)。访问或重新绑定任意列表项目的时间复杂度为 O(1)。对一个列表执行 len( ) 的时间复杂度也是 O(1)。将长度为 M1 的列表的切片绑定到一个具有不同长度 M2 的列表的时间复杂度为 O(M1+M2+N1)，其中 N1 是目标列表中的切片之后的项目数量（换句话讲，这样的长度改变切片重新绑定操作，如果发生在列表的末尾，其开销是很小的，而如果发生在一个长列表的开始或中间，则开销很大）。如果需要执行先入先出（FIFO）操作，列表可能并不是实现这个目的最快的数据结构：而是应该尝试使用 collections.deque 类型，参见第 8.5 节）。

如表 4-3 所示的大多数列表方法都相当于切片重新绑定，并且具有相同的大-O 性能。count、index、remove 和 reverse 方法和 in 操作符的时间复杂度都是 O(N)。sort 方法的时间复杂度通常为 O(N*log(N))，但是在某些极其特殊的情况下，该方法可以被高度最

优化为 O(N)，比如在列表已经被排序、反向排序或者除了少数项目之外都被排序的情况下。range(a, b, c)的时间复杂度为 O((b-a)/c)。xrange(a, b, c)的时间复杂度为 O(1)，但是对 xrange 的结果进行循环操作的时间复杂度为 O((b-a)/c)。

字符串操作

长度为 N 的字符串（可以是普通字符串，或者 Unicode 字符串）上的大多数方法的时间复杂度都是 O(N)。len(astring)的时间复杂度为 O(1)。生成一个音译和/或删除了特殊字符的字符串的副本的最快方法就是使用该字符串的 translate 方法。第 18.4 节中介绍了有关字符串的一个最实用，也最重要的大-O 因素。

字典操作

Python 字典内部是使用哈希表实现的。从大-O 的角度而言，这个基础的实现选择确定了 Python 字典全部性能特征。

访问、重新绑定、添加或删除一个字典项目的时间复杂度通常为 O(1)，同样地，has_key、get、setdefault 和 popitem 方法，以及 in 操作符的时间复杂度也为 O(1)。d1.update(d2)的时间复杂度为 O(len(d2))。len(addict)的时间复杂度为 O(1)。key、items 和 values 方法的时间复杂度为 O(N)。iterkeys、iteritems 和 itervalues 方法的时间复杂度为 O(1)，但是对这些方法返回的迭代器进行循环操作的时间复杂度为 O(1)（名称以 iter 开始的方法与其返回列表的类似方法相比可以节省内存，运行也更快），直接对 dict 进行循环与 iterkeys 具有相同的大-O 性能。不要使用 if x in d.keys() 进行测试，其时间复杂度为 O(N)，而与之相同的测试 if x in d: 的时间复杂度为 O(1)（if d.has_key(x): 的时间复杂度也是 O(1)，但是要比 if x in d: 更慢一些，并且没有补偿的好处。

在字典中的键都是定义了__hash__和相等比较方法的类的实例时，字典性能必然会受到这些方法的影响。本节提出的性能指标只有在哈希和相等比较的时间复杂度为 O(1)时才有效。

集合操作

与字典类似，Python 集合内部是使用哈希表实现的。从大-O 的角度而言，集合的全部性能特征与字典的性能特征相同。

添加或删除一个集合项目的时间复杂度通常是 O(1)，与 in 操作符相同。len(asset)的时间复杂度为 O(1)。对一个集合进行循环操作的时间复杂度为 O(N)。在集合中的项目都是定义了__hash__和相等比较方法的类的实例时，集合性能必然会受到这些方法的影响。本节提出的性能指标只有在哈希和相等比较的时间复杂度为 O(1)时才有效。

Python 内置类型操作的大-O 时间复杂度总结

假定 L 是任意列表，T 是任意字符串（普通字符串或 Unicode 字符串），D 是任意集合，

以（假定）数字作为项目（哈希和比较的时间复杂度为 O(1)），而 x 是任意数字：

O(1)

len(L)、len(T)、len(D)、len(S)、L[i]、T[i]、D[i]、del D[i]、if x in D、if x in S、S.add(x) 和 S.remove(x)，从 L 的最右边添加或删除项目。

O(N)

对 L、T、D 和 S 执行循环操作、向 L 添加或从 L 删除项目（不是在最右边）的一般操作、T 上的所有方法、if x in L、if x in T、L 上的大多数方法，以及所有的浅副本。

O(N log N)

通常包括 L.sort（如果 L 几乎已经按顺序或者按相反顺序排序，则时间复杂度为 O(N)）。

性能测试

大多数程序都有热点（也就是，在程序运行期间，源代码中占用了大多数运行时间的一部分代码）。不要尝试猜测程序的热点在哪里：众所周知，程序员在这个方面的直觉是相当不可靠的。使用 profile 模块通过已知输入信息，一次或多次地运行程序来收集性能测试数据。然后，使用 pstats 模块比较、解释和显示这些性能测试数据。要想获得精确性，可以校准计算机的 Python 性能测试器（也就是，确定性能测试会引起计算机在哪方面的开销）。然后，profile 模块可以从其度量的时间减去这方面的开销，这样，收集的性能测试数据就会更加接近于真实值。Python 2.5 引入了一个与 profile 具有类似功能的新标准库模块 cProfile；cProfile 要更好一些，因为 cProfile 的运行更快，并且会产生较少的开销。Python 标准库中的另一个性能测试模块是 hotshot（参见 http://docs.python.org/ lib/module-hotshot.html，该模块是从 Python 2.2 开始提供的）；不幸的是，hotshot 与线程不兼容。

profile 模块

profile 模块提供了一个经常使用的函数。

表 18-6

run	run(*code*,*filename*=None) code 是一个可以与 exec 语句一起使用的字符串，通常会调用正在进行性能测试的程序的主函数。filename 是 run 用来创建或重新写入性能测试数据的文件的路径。通常，为了按照一定的比例，根据开发者所期望的，将要在“真实生活”中使用的功能来测试程序的各个部分功能，开发者可以多次运行 run，指定不同的文件名，以及程序的主函数的不同参数。然后，开发者可以使用 pstats 模块来显示比较的结果。

续表

run	开发者可以不带 filename 参数调用 run 以在标准输出上获得一个汇总报告，类似于 pstats 模块可以提供的信息。但是，这种解决方案不能对输出信息的格式进行控制，也不能以任何方式将几次运行的结果集中到一个报告中。在实际使用中，开发者应该尽量不使用这个功能：最好将性能测试数据收集到文件中。 profile 模块还提供了 Profile 类（下一节将提到该类）。通过直接实例化 Profile，开发者可以访问高级功能，比如在一个指定的本地和全局目录中运行一个命令。本书中没有进一步介绍 profile.Profile 类的这些高级功能。

校准

要想校准计算机的 profile，需要使用 Profile 类，profile 模块提供了 Profile 类，并在 run 函数中内部使用。Profile 类的实例 p 提供了一个可以用于校准的方法。

表 18-7

calibrate	`p.calibrate(N)`
	循环 N 次，然后返回一个数字，这个数字是在开发者的计算机上进行每一次调用产生的性能测试开销。如果计算机的速度很快，N 的值必须很大。开发者可以调用几次 p.calibrate(10000)，然后检查调用该函数返回的几个数字相互之间是否很接近，然后选取其中最小的一个。如果这些数字的区别很大，可以尝试使用更大的值 N 再次进行测试。
	校准过程是非常消耗时间的。但是，开发者只需要执行一次校准过程，只有在开发者做出了可能会改变计算机的特性的更改时才需要重复校准过程，比如在对操作系统应用了补丁、添加了内存，或者更改了 Python 的版本时。只要开发者知道了计算机的开销，就可以在每次导入 profile 模块时告诉该模块这个开销值，也就是在使用 profile.run 之前。实现这个操作最简单的方法如下：
	`import profile`
	`profile.Profile.bias =... 开发者测量出来的开销值...`
	`profile.run('main()', 'somefile')`

pstats 模块

pstats 模块提供了单个类 Stats 以分析、合并和报告性能测试数据，该数据包含在使用 profile.run 函数写入的一个或多个文件中。

表 18-8

Stats	`class Stats(filename,*filenames)` 使用 profile.run 写入的测试数据文件的一个或多个文件名实例化 Stats。 Stats 类的实例 s 提供了一些方法以添加性能测试数据，然后对其进行排序并输出结果。每个方法都返回 s，因此可以在相同的表达式中链接几个调用。s 的主要方法如下。
add	`s.add(filename)` 将另一个性能测试数据文件添加到一个集合中，这个集合保存了 s 以进行分析。

续表

print_callees, print_callers	*s*.print_callees(**restrictions*) 输出 s 的性能测试数据中的函数列表，按照最后对 s.sort_stats 的调用进行排序，并遵循给定的约束条件（如果有的话）。开发者可以使用零个或多个 restrictions 约束条件以调用每个打印方法以减少输出的行数，该函数将按顺序一个接一个地应用这些约束条件。约束条件为 int n 可以将输出结果限制为前 n 行。约束条件为 0.0 和 1.0 之间的 float f 可以将输出结果限制为比例为 f 的总行数。以字符串表示的约束条件将被编译为一个正则表达式（参见第 9.7 节）；只有满足对正则表达式上的 search 方法调用的行将被输出。只有满足 search 方法调用中的正则表达式的行才会被输出。约束条件是累计的。例如，s.print_calls(10, 0.5)将输出前 5 行（10 的一半）。只能对汇总行和首部行之后的行应用约束条件：汇总行和首部行将无条件输出。 输出的每个函数 f 都根据该方法的名称附带了 f 的调用者（调用了 f 的函数）或者 f 的被调用者（f 调用的函数）的列表。
print_stats	*s*.print_stats(**restrictions*) 输出有关 s 的性能测试数据的统计信息，按照最后一次调用 s.sort_stats 进行排序，并遵循给定的约束条件（如果有的话），参见 print_callees 和 print_callers 方法。在几行汇总行（收集了性能测试文件的日期和时间、函数调用的数量以及使用的排序标准）、输出信息和缺少的约束条件之后，每个函数占用一行输出，每行包含 6 个字段，标题行中给出了每个字段的标签名。对于每个函数 f，print_stats 将输出 6 个字段： • 调用函数 f 的总次数； • 函数 f 花费的总时间，除 f 调用的其他函数花费的时间之外； • 每次调用 f 花费的时间（也就是，字段 2 除以字段 1）； • 函数 f 累计花费的时间，包括从 f 直接或间接调用所有函数花费的时间； • 每次调用 f 累计花费的时间（也就是，字段 4 除以字段 1）； • 函数 f 的名称。
sort_stats	s.sort_stats(*key*, **keys*) 给定一个或多个键，并根据这些键按优先级顺序对将来的输出信息进行排序。每个键都是一个字符串。按降序对表示时间或数字的键进行排序，并按字母顺序对键‘nfl’进行排序。在调用 sort_stats 时，最常使用的键是： 'calls' 函数的调用次数（相当于 print_stats 方法中介绍的字段 1）； 'cumulative' 某个函数和该函数调用的所有函数累计花费的时间（相当于 print_stats 方法中介绍的字段 4）； 'nfl' 函数的名称、模块，以及该函数在其源文件中的行号（相当于 print_stats 方法中介绍的字段 6）； 'time' 函数本身花费的总时间，除该函数调用的函数之外（相当于 print_stats 方法中介绍的字段 2）。

续表

strip_dirs	*s*.strip_dirs() 通过从所有模块名中去掉目录名称来改变 s，从而让将来的输出更简洁。在调用 s.strip_dirs( )之后，s 是未经排序的，因此，通常需要在 s.strip_dirs 之后立即调用 s.sort_stats。

小尺度最优化

对程序的操作进行细粒度地控制是极其重要的。这种微调可能会在程序的某些特殊热点上造成很小，但是非常有意义的区别，而这些细节往往是以前几乎不会考虑到，却是决定性的因素。然而，微调在大多数情况下追求的是与主要性能无关的一些微小的效率，而这往往是程序员的本能导致的。正是因为如此，大多数程序的最优化操作往往都为时过早，开发者最好避免这些最优化操作。微调最大的好处可以说是，如果一个习惯用法总是要比另外一个更快，而两者之间的区别可以被测量时，习惯上更有价值的做法总是使用前者而不是后者。

在 Python 中，更常见的是，如果开发者执行自然而然的操作，并且选择了简单性和优雅性，那么最终就会得到具有很好的性能，清晰，并具有可维护性的代码。只有在少数几种情况下，一种直观上不可取的解决方案依然还能提供性能上的好处，参见本节其余部分的介绍。

最简单的最优化可能就是使用 python -O 或-OO 来运行 Python 程序了。-OO 与-O 在性能上几乎没有什么直接的区别，但是-OO 可以节省内存，因为该选项从字节代码中删除了文档字符串，而内存可用性有时候（非直接）是一个性能瓶颈。在当前的 Python 发布版本中，优化器的功能并不十分强大，但是仍然会为开发者在性能上获得大约 5%的提高，有时候甚至会高达 10%（如果开发者按照第 6.6 节中的建议使用了 assert 语句和 if __debug__：守护语句，性能可能会提高得更多）。使用-O 最好的地方是，该选项没有任何开销——当然，只要没有太早进行最优化操作（不要在正在开发一个程序的时候使用-O，这样会干扰程序的开发）。

timeit 模块

标准库模块 timeit 非常便于测量特定代码片段的精确性能。开发者可以在代码通过 timeit 模块使用 timeit 的功能，但是最简单和最常用的方法是从命令行使用 timeit 功能：

```
python -mtimeit -s'setup statement(s)' 'statement(s) to be timed'
```

例如，假定开发者想知道 x=x+1 与 x+=1 在性能上有什么区别。在命令提示符上，可以简单地使用以下命令：

```
$ python -mtimeit -s'x=0' 'x=x+1'
1000000 loops, best of 3: 0.25 usec per loop
$ python -mtimeit -s'x=0' 'x+=1'
1000000 loops, best of 3: 0.258 usec per loop
```

这样可以发现在两种情况的所有目的和方法都相同时的性能区别。

从片段构建字符串

最有可能伤害程序性能的唯一一个 Python“反习惯用法”就是，使用像 big_string+=piece 这样的字符串串联语句进行循环操作，从多个字符串片段构建一个很大的字符串。Python 字符串都是不可变的，因此，每个这样的串联操作意味着 Python 必须释放以前为 big_string 分配的 M 个字节，然后为新的字符串分配 M+K 个字节，并填充这些字节。开发者可以在一个循环中重复执行这个操作，最终会获得大约为 $O(N^2)$的性能，其中 N 是字符的总数。很多时候，在可以获得 O(N)性能的地方采用 $O(N^2)$性能就是一个性能灾难。在某些平台上，情况甚至会更加糟糕，这是由于释放许多日益增多的，更大尺寸的内存区域而产生的内存碎片造成的影响。

为了实现 O(N)的性能，可以将中间字符串片段累积到一个列表中，而不是一个片段一个片段地构建字符串。与字符串不同，列表是可变的，因此，向列表的末尾添加一个字符串片度的性能为 O(1)。将所有的 big_string+=piece 语句更改为 temp_list. append(piece)。然后，在完成了字符串的累积之后，可以使用以下语句在 O(N)时间内建立想要的字符串结果：

```
big_string = ''.join(temp_list)
```

使用列表推导、生成器表达式或其他直接方法（比如调用 map，或者使用标准库模块 itertools）来构建 temp_list 通常可能会通过重复调用 temp_list.append 提供进一步的最优化。其他用来构建大的字符串的 O(N)方法就是使用数组的 extend 方法将字符串片段串联到一个 array.array('c')实例中，或者将字符串片段写入到一个 cStringIO.StringIO 实例中，Python 程序员可能会发现这些方法更具可读性。

在开发者想要输出结果字符串的特殊情况下，可以通过对 temp_list 使用 writelines 方法进一步增加一小片性能（不在内存中创建 big_string）。在可行的情况下（也就是，在循环操作中已经有了一个可用，并且打开的输出文件对象时），就像对每个 piece 执行一个 write 调用的效率一样，不存在任何累积操作。

尽管远不如对一个循环中的大字符串执行+=运算的性能影响那么大，但是另一种情况，删除字符串串联可能会带来细微的性能增强，就是在一个表达式中串联几个值时：

```
oneway = str(x)+' eggs and '+str(y)+' slices of '+k+' ham'
another = '%s eggs and %s slices of %s ham' % (x, y, k)
```

使用%运算符格式化字符串通常都是一个很好的性能选择。

搜索和排序

in 操作符是最自然的搜索工具，在右边操作数为一个集合或字典时，该操作符的时间复杂度为 O(1)，而在右边操作数为一个字符串、列表或元组时，时间复杂度为 O(N)。如果开发者需要对一个容器执行多个搜索操作，通常最好使用集合或字典，而不是列表或元

组作为容器。Python 集合和字典都高度最优化了通过键来搜索和提取项目的功能。

Python 列表的 sort 方法也是一个高度最优化和非常成熟的工具。开发者可以依赖于 sort 的性能。但是，如果开发者为了基于任何对象对一个列表进行排序而向 sort 传递一个自定义的可调用对象以执行比较，而不是使用内置比较工具，则性能会严重下降。要想满足这样的需求，可以考虑使用装饰-排序-去除装饰（DSU）习惯用法。更清楚地讲，这种习惯用法包含以下步骤。

Decorate

创建一个辅助列表 A，其中的每个项目都是一个由排序键组成的元组，以原始列表 L 的项目或者以该项目的索引结束。

Sort

不带任何参数调用 A.sort( )。

Undecorate

按照已经排序的列表 A 中的顺序提取项目。

例如，假定在 L 中有一个很大的字符串列表，每个字符串至少有两个单词，并且开发者想要通过每个字符串的第二个单词在原地对 L 进行排序：

```
A = [ (s.split()[1], s) for s in L ]
A.sort()
L[:] = [ t[1] for t in A ]
```

这段代码要比向 L.sort 传递一个按照第二个单词比较两个字符串的函数快得多，比如：

```
def cmp2ndword(a, b): return cmp(a.split()[1], b.split()[1])
L.sort(cmp2ndword)
```

通过使用 Python 2.4 对包含 10000 个字符串的列表执行一系列基准测试，可以测量出使用 DSU 的代码要比不使用 DSU 的代码快大约 5 倍。

使用 DSU 的一个特别快和特别有效的方法是为 sort 指定一个命名参数 key=，这是 Python 2.4 中引入的一种可行的方法。operator 模块提供了特别适合于这种用途的 attrgetter 和 itergetter 函数。执行上述任务最快的方法是：

```
def secondword(s):retum s split()[1]
L.sort(key = secondword)
```

在相同的基准测试中，这种方法要比普通的 DSU 代码又快 5 倍。

有时候，开发者可以使用堆来避免排序的需要，参见第 8.8 节。

避免使用 exec 和 from...import *

函数中的代码要比模块中的顶层代码运行得更快，这是因为访问一个函数的本地变量

被最优化为非常快地执行。但是，如果一个函数包含了一个不带任何显式字典的 exec 语句，整个函数将会慢下来。Python 编译器通常会对与访问本地变量有关的语句执行最优化操作，而出现这样的 exec 语句会迫使 Python 编译器不再执行这种适度的，但是很重要的最优化操作，这是因为 exec 可能会对函数的命名空间造成根本性的改变。下面这种形式的 from 语句：

```
from MyModule import *
```

也会浪费性能，因为这条语句也可能不可预料地改变一个函数的命名空间。

exec 本身也是非常慢的，如果开发者将其应用到一个源代码字符串，而不是一个代码对象上，exec 甚至会更慢。到现在为止，最好的解决方案——为了性能、为了正确性，也为了清晰度——就是完全避免使用 exec。通常总是可以找到一个更好的（更快、更健壮和更清楚）解决方案。如果开发者必须使用 exec，可以总是与显式字典一起使用。如果开发者需要多次执行（exec）一个动态获得的字符串，可以只编译（compile）一次该字符串，然后重复执行结果代码对象。

eval 可以对表达式进行操作，而不是语句；因此，尽管仍然很慢，eval 可以避免 exec 造成的某些最坏的性能影响。同样，在使用 eval 时，最好考虑使用显式字典。如果开发者需要对相同的动态获得的字符串进行几次计算，只需要编译一次该字符串，然后重复计算结果代码对象。

参见第 13.1 节以了解有关 exec、eval 和 compile 的更详细信息和建议。

最优化循环

程序的大多数瓶颈往往都出现在循环操作中，特别是嵌套的循环，因为循环体都是重复执行的。Python 不隐式执行任何代码提升：如果循环中有任何代码只需要执行一次，而这个循环是一个性能瓶颈，则需要开发者自己将该代码从循环中提升出来。有时候要提升的代码可能不会立即出现：

```
def slower(anobject, ahugenumber):
    for i in xrange(ahugenumber): anobject.amethod(i)
def faster(anobject, ahugenumber):
    themethod = anobject.amethod
    for i in xrange(ahugenumber): themethod(i)
```

在这段代码中，faster 从循环中提升出来的代码就是属性查找方法 anobject.amethod。slower 每次都须要重复查找操作，而 faster 只须要执行一次。这两个函数并不是 100%相等的：几乎很难察觉执行 amethod 可能会导致对 anobject 产生的这样一个更改，也就是，下次查找相同的命名属性将提取一个不同的方法对象。这就是为什么 Python 不自己执行这样的最优化的原因。实际上，这些微妙、隐晦和麻烦的情况极少发生；开发者可以非常安全地自己执行这样的最优化，这样可以从某些至关重要的瓶颈中解决最后的性能下降问题。

在 Python 中，使用本地变量要比使用全局变量更快。如果一个循环需要重复访问一个

在迭代操作不会发生改变的全局变量，可以将全局变量的值缓存在本地变量中，然后让循环操作访问这个本地变量。这种方法还可以应用于内置对象：

```
def slightly_slower(asequence, adict):
    for x in asequence: adict[x] = hex(x)
def slightly_faster(asequence, adict):
    myhex = hex
    for x in asequence: adict[x] = myhex(x)
```

在这段代码中，速度的增加非常有限，在5%左右。

不要缓存None。None是一个关键字，因此不需要进一步最优化。

列表推导可以比循环更快，因此可以map和filter。出于最优化的目的，可以尝试将循环更改为列表推导，或者在可行的位置调用map和filter。如果开发者必须使用lambda表达式或者特殊级别的函数调用，则map和filter在性能上的好处是无效的，只有在向map或filter传递一个内置函数，或者一个无论如何都需要调用的函数（即使是从一个显式循环中调用该函数）时，一定会获得性能上的好处。

开发者可以最自然地使用列表推导，或者map和filter调用来替换的循环操作，也就是那些通过重复对该列表调用append建立一个列表的循环操作。下面的示例在一个微观性能基准测试脚本中显示了这个最优化操作：

```
import time, operator

def slow(asequence):
    result = []
    for x in asequence: result.append(-x)
    return result

def middling(asequence):
   return map(operator.neg, asequence)

def fast(asequence):
    return [-x for x in asequence]

biggie = xrange(500*1000)
tentimes = [None]*10
def timit(afunc):
    lobi = biggie
    start = time.clock()
    for x in tentimes: afunc(lobi)
    stend = time.clock()
    return "%-10s: %.2f" % (afunc.__name__, stend-start)

for afunc in slow, middling, fast, fast, middling, slow:
    print timit(afunc)
```

在作者的笔记本计算机上使用Python 2.4运行上面这个示例显示，fast花费了3.62秒，middling花费了4.71秒，而slow花费了6.91秒。换句话讲，在这台计算机上，slow（append方法调用的循环）比middling（单个map调用）慢47%，而middling要比fast（列表推

导）慢大约 30%。列表推导是表达这个示例中被微基准测试的任务的最直接方法，因此，请不要惊奇，这也是最快的方法——几乎要比 append 方法调用的循环快大约 2 倍。

最优化 I/O

如果开发者的程序需要执行大量的 I/O 操作，很有可能性能瓶颈就在 I/O 上，而不是在计算上。这样的程序可以被称为 I/O 限制，而不是 CPU 限制。操作系统将尝试最优化 I/O 的性能，但是开发者也可以使用许多方法来对 I/O 进行最优化。其中的一个方法就是对大块的数据执行 I/O 操作，这对性能是最佳的，而不是简单地为了便于程序的操作。另一个方法就是使用线程。

从程序的便利性和简单性的角度来看，一次读取或写入的最理想的数据量通常是很小（一次一个字符或一行数据）或者非常大的（一次整个文件）。通常这样都是可以的：Python 和后台工作的操作系统可以让程序使用非常方便的逻辑数据块来执行 I/O 操作，而使用不同的块大小来处理物理 I/O 操作可以对性能进行更好的调节。只要一个文件不是非常大，一次读取和写入整个文件可以获得非常好的性能。特别是，只要文件中的数据非常适合于放到物理内存中，一次一个文件的 I/O 就是很好的，在这种情况下，需要为程序和操作系统保留充足的可用内存以执行同时执行的任何其他任务。I/O 限制在性能方面的最大难题往往来自于巨大的文件。

如果性能是一个问题，则不要使用文件的 readline 方法，这个方法受到了其可以执行的分块和缓冲数量的限制（另一方面，在 readline 方法适用于开发者的程序时，使用 writeline 不会有性能问题）。在读取一个文本文件时，直接对文件对象进行循环可以以最好的性能一次获得一行文本。如果这个文件并不是非常巨大，从而可以很方便地适合于放到内存中，对程序的这两个版本——一个直接对文件对象进行循环，另一个调用 readlines 方法以将整个文件读取到内存中——进行计时。不能证明哪个版本更快一些。

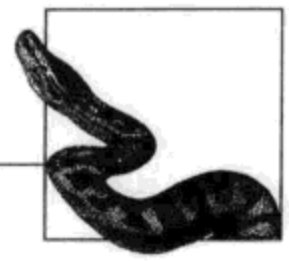
对于二进制文件，特别是很大的二进制文件，开发者在每次运行程序时只需要读取该文件的内容中的一部分，mmap 模块（参见第 14.8 节）通常可以为开发者提供很好的性能，也可以为程序提供简单性。

如果开发者可以按照多线程方式安排程序的结构，那么，让一个 I/O 限制程序按多线程方式运行有时候可以提供真正的性能增强。开始几个专门用于 I/O 的工作者线程，让计算线程通过 Queue 实例从 I/O 线程请求 I/O 操作，然后，只要开发者知道最终将需要这些数据，就为每个输入操作发送请求。只有在计算线程在 I/O 线程被阻塞以等待数据时还可以执行其他任务的情况下，性能才会增加。基本上讲，如果开发者可以设法实现并发计算和等待数据，并使用不同的线程来实现计算和等待操作（第 14.1 节中详细介绍了 Python 线程和一个建议的体系结构），就可以使用多线程方法获得更好的性能了。另一方面，如果一部分真实 I/O 在网络上，一个甚至更快，并且绝对更可伸缩的解决方案就是使用异步（事件驱动）体系结构来回避线程，参见第 20.3 节。

第 4 部分

网络和 Web 编程

第 19 章



客户端网络协议模块

在 Internet 上，一个程序可以作为客户（client，访问资源的程序）使用或者作为服务器（server，提供服务的程序）。这两种程序都需要处理协议问题，比如，如何访问和传输数据，以及数据格式问题。为了有条理和更清楚，Python 库将在几个不同的模块中处理这些问题。本书将在下面几章介绍这些主题。本章将介绍 Python 库中支持客户程序协议处理的模块。第 20 章将介绍客户和服务器程序中都要使用的底层模块，比如 socket，以及支持服务器程序协议处理的模块。有关数据格式的问题，参见第 22 章、第 23 章和第 24 章。第 21 章特别介绍了用来创建网页的服务器端程序，分别说明了如何使用服务器端程序单独创建网页或者与已有的 Web 服务器共同创建网页，比如 Apache 或 IIS。

数据访问通常可以非常简单地通过“统一资源定位器”（URL）来实现。Python 在 urlparse、urllib 和 urllib2 模块中都支持对 URL 的操作。在极少数情况下，比如，在需要对通常通过 URL 访问的数据访问协议进行细粒度的控制时，Python 还提供了 httplib 和 ftplib 模块。通常不仅仅需要处理 URL 的协议包括邮件（poplib 和 smtplib 模块）、网络新闻（nntplib 模块）和远程登录（telnetlib 模块）。Python 还通过 xmlrpclib 模块支持用于分布式计算的 XML-RPC 协议。

19.1 URL 访问

URL 标识了 Internet 上的一个资源。URL 是一个由几个被称为组件的可选部分组成的字符串，这些组件也被称为方案（协议）、位置、路径、查询和片段。一个包含所有部分的 URL 显示如下：

```
scheme://lo.ca.ti.on/pa/th?query#fragment
```

例如，在 http://www.python.org:80/faq.cgi?src = fie 中，协议方案是 http、位置是 www.python.org:80、路径是/faq.cgi、查询是 src = file，没有片段。某些标点符号字符用来将其分割的部分形成一个组件，而其他标点符号只是分隔符，是非组件的一部分。省略某些标

点符号就意味着失去了组件。例如，在 mailto:me@you.com 中，协议方案是 mailto、路径是 me@you.com，没有位置、查询或者片段。省略//表示该 URL 没有位置部分，省略?表示该 URL 没有查询部分，而省略#表示该 URL 没有片段部分。

urlparse 模块

urlparse 模块提供了一些函数以分析和综合 URL 字符串。urlparse 模块最常用的函数是 urljoin、urlsplit 和 urlunsplit。

表 19-1

urljoin	urljoin(*base_url_string*,*relative_url_string*) 返回一个 URL 字符串 u，该字符串是通过连接 relative_url_string 字符串和 base_url_string 字符串而获得的，relative_url_string 字符串可能是相对的。urljoin 执行的、用来获得其结果字符串 u 的连接过程可以总结如下。 • 在任意一个参数字符串为空时，u 就是另一个参数。 • 在 relative_url_string 显式指定一个不同于 base_url_string 的协议方案时，u 为 relative_url_string。否则，u 的协议方案为 base_url_string。 • 在协议方案不允许相对 URL（例如，mailto），或者 relative_url_string 显式指定一个位置（即使这个位置与 base_url_string 的位置相同）时，u 的所有其他组件都为 relative_url_string 的组件。否则，u 的位置为 base_url_string 的位置。 • u 的路径是根据绝对和相对 URL 路径的标准语法，通过连接 base_url_string 和 relative_url_string 的路径获得的。例如： importurlparseurlparse.urljoin('http://somehost.com/some/ path/here','../other/path') # 结果是: 'http://somehost.com/some/other/path'
urlsplit	urlsplit(*url_string*,*default_scheme*='',*allow_fragments*=True) 分析 url_string，并返回一个包含 5 个字符串项目的元组：协议方案、位置、路径、查询和片段。default_scheme 是在 url_string 缺少协议方案时默认的第一个项目。在 allow_fragments 为 False 时，该元组的最后一个项目总是‘ ’，不管 url_string 有没有片段。省略的部分对应的项目总是‘ ’。例如： urlparse.urlsplit('http://www.python.org:80/faq.cgi?src=fie') #结果是:('http','www.python.org:80','/faq.cgi','src=fie','')
urlunsplit	urlunsplit(*url_tuple*) url_tuple 是任意一个具有确定的 5 个项目的可迭代对象，其中的所有项目都是字符串。例如，urlsplit 函数调用返回的任意值都是 urlunsplit 可以接受的参数。urlunsplit 将返回一个包含给定组件和所需分隔符的 URL 字符串，但是不包含冗余分隔符（例如，在片段，也就是 url_tuple 的最后一个项目为‘ ’时，结果中不包含#）。例如： urlparse.urlunsplit(('http','www.python.org:80','/faq.cgi','src=fie','')) # 结果是: 'http://www.python.org:80/faq.cgi?src=fie' urlunsplit(urlsplit(x))可以返回标准化形式的 URL 字符串 x，返回的字符串不一定等于 x，因为 x 不需要被标准化。例如： urlparse.urlunsplit(urlparse.urlsplit('http://a.com/path/a?')) # 结果是: 'http://a.com/path/a' 在这种情况下，标准化可以确保冗余分隔符，比如 urlsplit 的参数中的拖尾字符?，不会出现在结果中。

urllib 模块

urllib 模块提供了一些简单的函数以从 URL 中读取数据。urllib 提供了以下协议(方案)：http、https、ftp、gopher 和 file。file 表示本地文件。urllib 使用 file 作为缺少显式协议方案的 URL 的默认协议方案。开发者可以在第 23 章找到使用 urllib 的简单和典型的示例，其中 urllib.urlopen 被用作提取 HTML 和 XML 页面，以供所有类型的示例进行解析和分析。

函数

urllib 模块提供了许多函数，urlopen 是使用最频繁的一个。

表 19-2

quote	quote(*str*,*safe*='/')
	返回 str 的一个副本，其中的特殊字符串都被更改为 Internet 标准引用形式 %xx。该函数不引用字母和数字字符、空格、_，.-，字符，也不引用字符串 safe 中的任何字符。例如：
	`print urllib.quote('zip&zap')`
	`# 结果是: zip%26zap`

quote_ _plus	quote_plus(*str*, *safe*='/') 与 quote 类似，但是还将空格更改为加号。

unquote	unquote(*str*)
	返回 str 的一个副本，其中的每个引用形式%xx 将被更改为对应的字符。例如：
	`print urllib.unquote('zip%26zap')`
	`# 结果是: zip&zap`

unquote_plus	unquote_plus(*str*) 与 unquote 类似，但是还将加号更改为空格。
urlcleanup	urlcleanup() 清除 urlretrieve 函数的缓存，参见后面介绍的 urlretrieve 函数。

urlencode	urlencode(*query*,*doseq*=False)
	以 query 的 URL 编码形式返回一个字符串。query 可以是一个由(name, value)数据对组成的序列，或者是一个映射，在这种情况下，结果字符串将解码该映射的（key, value）数据对。例如：
	`urllib.urlencode([('ans',42),('key','val')])`
	`# 'ans=42&key=val'`
	`urllib.urlencode({'ans':42, 'key':'val'})`
	`# 'key=val&ans=42'`
	字典中的项目的顺序是任意的：如果需要 URL 编码形式按特定顺序包含键/值数据对，可以使用序列作为 query 参数，就像上面的代码段中的第一个函数调用一样。
	在 doseq 为 True，且 query 为一个序列，而不是一个字符串时，query 中的任何 value 都被编码为一个单独的参数，value 中的每个项目一个参数。例如：
	`urllib.urlencode([('K',('x','y','z'))],True)`
	`# 'K=x&K=y&K=z'`
	在 doseq 为 False（默认值），每个值将被编码为内置 str 给定的该值的字符串形式 quote_plus，不管这个值是不是一个序列：
	`urllib.urlencode([('K',('x','y','z'))],False)`
	`# 'K=%28%27x%27%2C+%27y%27%2C+%27z%27%29'`

续表

urlopen	urlopen(*urlstring*,*data*=None,*proxies*=None) 访问给定的 URL 并返回一个只读的类文件对象 f。f 提供了类文件方法 read、readline、readlines 和 close，以及两个其他方法。 *f*.geturl ( ) 返回 f 的 URL。因为标准化（在前面介绍 urlunsplit 的时候提到过）和 HTTP 重定向（也就是，表示请求的数据位于其他位置）的原因，返回的 URL 可能与 urlstring 有所区别。urllib 支持透明重定向，如果需要，可以使用 geturl 方法进行检查。 f.info ( ) 返回 mimetools 模块的 Message 类的一个实例 m，参见第 22.2 节。m 的首部提供了有关 f 的元数据。例如，m['Content-Type']是 f 中的数据的 MIME 类型，并且 m 的方法 m.gettype( )、m.getmaintype( )和 m.getsubtype( )也提供了相同的信息。 在 data 为 None，并且 urlstring 的协议方案为 http 时，urlopen 将发送一个 GET 请求。在 data 不是 None 时，urlstring 的协议方案必须是 http， urlopen 将发送一个 POST 请求。然后，data 必须符合 URL 编码形式，开发者通常可以使用 urlencode 函数准备这个参数，参见前面介绍的 urlencode 函数。 urloepn 可以使用不要求身份认证的代理。可以在代理的 URL 中设置环境变量 http_proxy、ftp_proxy 和 gopher_proxy 以利用这个功能。在启动 Python 之前，开发者通常需要按平台相关方法在系统的环境中执行这样的设置。在 Macintosh 系统中，urlopen 将从 Internet 配置设置中透明和隐式提取代理 URL。另外，开发者还可以将一个映射传递为参数 proxies，映射的键为协议方案名称，对应的值就是代理 URL。例如： `f = urllib.urlopen('http://python.org',proxies={'http':'http://prox:999'})` urlopen 不支持要求身份认证的代理；对于这样的高级需求，可以使用更丰富的库 urllib2 模块，参见第 19.1 节。

urlretrieve	urlretrieve(*urlstring*,*filename*=None,*reporthook*=None,*data*=None)
	类似于 urlopen(urlstring, data)，但是返回一个数据对（f, m）。f 是一个指定到本地文件系统上的一个文件的路径的字符串。m 是 mimetools 模块的 Message 类的一个实例，就像对 urlopen 的返回值调用 info 方法所产生的结果一样，参见前面介绍的 urloepn 函数。
	在 filename 为 None 时，urlretrieve 将把提取的数据复制到一个临时的本地文件中，而 f 是该临时本地文件的路径。在 filename 不是 None 时，urlretrieve 将把提取的数据复制到名为 filename 的文件中，并且 f 是 filename。在 reporthook 不是 None 时，必须是一个带有 3 个参数的可调用函数，如下面的函数所示：
	`def reporthook(block_count, block_size, file_size):`
	`  print block_count`
	在提取数据时，urlretrieve 可能会不调用或多次调用 reporthook。在每次调用时，urlretrieve 都会向其传递 block_count，到目前为止已经提取的数据块数；block_size，每个块的字节大小；和 file_size，提取的文件的总的字节大小。在 urlretrieve 无法确定文件的大小时，将把 file_size 的值传递为 −1，这取决于当前执行的协议，以及服务器实现该协议的完整程度。reporthook 的目的是允许开发者的程序向用户提供一个有关 urlretrieve 执行文件提取操作的进展情况的图形化或文本化的反馈信息。

FancyURLopener 类

开发者通常都是通过使用 urllib 模块提供的函数（最常用的是 urlopen）来使用 urllib 模块的。但是，要想自定义 urllib 的功能，可以创建 urllib 的 FancyURLopener 类的一个子类，并将该子类的一个实例绑定到 urllib 模块的_urlopener 属性上。FancyURLopener 的子类的实例 f 的可自定义功能包括以下内容，如表 19-3 所示。

表 19-3

prompt_user_passwd	*f*.prompt_user_passwd(*host*,*realm*) 返回一个数据对（user, password），该数据对用来在安全区域（realm）下认证对 host 的访问。FancyURLopener 类中的默认实现将在交互式文本模式下提示用户输入这些数据。开发者的子类可以覆盖这个方法，以通过 GUI 与该用户进行交互，或者从持久化的存储空间提取身份认证数据。
version	*f*.version f 用来向服务器标识其本身的字符串，例如，通过 HTTP 中的 User-Agent 首部。开发者可以通过创建子类或者将这个属性直接重新绑定到 FancyURLopener 的一个实例上来覆盖这个属性。

urllib2 模块

urllib2 模块是 urllib 模块的一个功能丰富，高度可自定义的超集。urllib2 可以用来直接处理高层协议，比如 HTTP。例如，可以使用自定义的首部和 URL 编码的 POST 正文发送请求，并以 Basic 和 Digest 的形式直接或通过 HTTP 代理处理各种安全区域中的身份认证。

在本节的其余部分，将只介绍 urllib2 中的一些可以让开发者的程序自定义 URL 检索等高级功能的方法。本节不打算详细介绍 HTTP 和其他网络协议的高级知识，这些知识独立于 Python，但是开发者需要了解这些知识以完整地使用 urllib2 的丰富功能。本书推荐使用 New Riders 出版，Steve Holden 编著的 *Python Web Programming* 一书作为 HTTP 的教程：本书提供了有关 HTTP 基础知识的详细介绍和使用 Python 编程的例子，如果读者需要进一步了解网络协议的详细信息，这是一本很好的参考书。

函数

urllib2 提供了一个基本上等同于 urllib 的 urlopen 函数的 urlopen 函数。要想自定义 urllib2，需要在调用 urlopen 函数之前，使用 build_opener 和 install_opener 函数安装被组合到一个 opener 的任意数量的处理程序。

开发者还可以有选择地向 urlopen 传递 Request 类的一个实例，而不是一个 URL 字符串。这样的实例可能会包含一个 URL 字符串和有关如何访问该 URL 的补充信息，参见第 19.1 节中介绍的 Request 类。

表 19-4

build_opener	build_opener(*handlers) 创建并返回 OpenerDirector 类（参见第 19.1 节）的一个实例和给定的参数 handlers。每个处理程序都可以是 BaseHandler 类的一个子类，可以不带参数进行实例化，或者是这样的子类的一个实例，但是已经被实例化了。build_opener 将在被指定用来处理代理、未知的协议方案；http、file 和 https 方案；HTTP 错误和 HTTP 重定向等的处理程序之前添加 urllib2 模块提供的各种处理程序类的实例。但是，如果在 handlers 中存在刚才提到的类的实例或子类，这意味着开发者需要覆盖这些默认的实例或子类。
install_opener	install_opener(*opener*) 安装 opener 作为进一步调用 urlopen 的打开程序。opener 可以是 Request 类的一个实例，参见下一节的介绍。
urlopen	urlopen(*url*,*data*=None) 几乎与 urllib 模块中的 urlopen 函数完全相同。但是，自定义行为是通过 urllib2 的 opener 和 handler 类来实现的（参见下面介绍的 OpenerDirector 类和 Handler 类），而不是像在 urllib 中那样，是通过 FancyURLopener 类实现的。参数 url 可以是一个 URL 字符串，就像 urllib 模块中的 urlopen 使用的 url 参数一样。另外，url 可以是 Request 类的一个实例，参见下一节。

Request 类

开发者可以有选择地向 urlopen 函数传递 Request 类的一个实例，而不是一个 URL 字符串。这样的实例可以具体化一个 URL，也可以选择具体化其他有关如何访问目标 URL 的信息。

表 19-5

Request	class Request(*urlstring*,*data*=None,*headers*={}) urlstring 是 Request 类的这个实例具体化的一个 URL。例如，如果没有 data 和 headers 参数，可以调用： urllib2.urlopen(urllib2.Request(*urlstring*)) 就像调用下面的方法： urllib2.urlopen(*urlstring*) 在 data 不为 None 时，Request 构造函数将对新实例 r 隐式调用其方法 r.add_data(data)。headers 必须是一个从首部名称到首部值的映射。Request 构造函数可以执行等同于下面的代码的循环操作： for *k*,*v* in *headers*.items(): *r*.add_header(*k*,*v*) Request 构造函数还可以接受一些可选的参数，以允许对 HTTP Cookie 的行为进行更细粒度地控制，但是，这样的高级功能是极少需要的：类中对 cookies 的默认处理通常就已经足够了。要想对 cookies 进行更细粒度的客户端控制，参见 http://docs.python.org/lib/module-cookielib.html；本书没有介绍标准库的 cookielib 模块。 Request 类的实例 r 提供了以下方法。

续表

add_data	*r*.add_data(*data*) 将 data 设置为 r 的数据。然后，调用 urlopen(r)就像调用 urlopen(r, data)一样，也就是说，要求 r 的协议方案是 http，并使用包含正文 data 的 POST 请求，该请求必须是一个 URL 编码的字符串。 尽管 add_data 方法的名称表示添加数据，但是该方法并不只是添加 data 数据。如果 r 已经包含了数据，不管是在 r 的构造函数中设置的，还是在以前调用 r.add_data 时获得的，最后一次调用 r.add_data 时将使用新给定的值替换 r 以前的数据的值。特别是，r.add_data(None)将删除 r 以前的数据（如果该数据已经存在）。
add_header	*r*.add_header(*key*,*value*) 使用给定的 key 和 value 向 r 的首部添加一个首部。如果 r 的协议方案是 http，r 的首部将被用作请求的一部分。在使用相同的 key 添加多于一个的首部时，后面添加的首部将改写以前的首部，因此，在给定的 key 对应的所有首部中，只有最后一个给定的首部才是有效的。
add_unredirected_header	*r*.add_unredirected_header(*key*,*value*) 与 add_header 类似，区别在于只在第一次请求的时候添加首部，如果请求过程遇到或者跟随任何进一步的 HTTP 重定向，则不使用该首部。
get_data	*r*.get_data() 返回 r 的数据，结果为 None 或者一个 URL 编码的字符串。
get_full_url	*r*.get_full_url() 返回 r 的 URL，也就是构造函数中为 r 指定的 URL。
get_host	*r*.get_host() 返回 r 的 URL 的 host 组件。
get_method	*r*.get_method() 返回 r 的 HTTP 方法，也就是字符串‘GET’或‘POST’。
get_selector	*r*.get_selector() 返回 r 的 URL 的 selector 组件（路径和所有下面的组件）
get_type	*r*.get_type() 返回 r 的 URL 的协议方案组件（也就是，协议）
has_data	*r*.has_data() 相当于 r.get_data( ) is not None。
has_header	*r*.has_header(*key*) 如果 r 中包含带有给定 key 的首部，则返回 True；否则，返回 False。
set_proxy	*r*.set_proxy(*host*,*scheme*) 将 r 设置为使用 给定 host 和 scheme 的代理来访问 r 的 URL。

OpenerDirector 类

OpenerDirector 类的实例 d 可以收集 handler 类的实例并综合使用这些实例以打开各种协议方案的 URL，并处理错误。通常，可以通过调用 build_opener 函数创建实例 d，然后调用 install_opener 函数安装该实例。对于更高级的用法，还可以访问 d 的各种属性和方法，但是这是极少需要的，本书将不再进一步对此进行介绍。

Handler 类

urllib2 模块提供了一个 BaseHandler 类，这个类可以用作开发者编写的任何自定义 handler 类的超类。urllib2 还提供了 BaseHandler 类的许多具体子类，可以用来处理协议方案 gopher、ftp、http、https 和 file，以及身份认证、代理、重定向和错误。编写自定义处理程序是一个高级主题，本书将不再进一步对此进行介绍。

处理身份认证

urllib2 的默认打开程序（opener）并不执行身份认证操作。要想获得身份认证，可以调用 build_opener 以创建一个包含 HTTPBasicAuthHandler、ProxyBasicAuthHandler、HTTPDigestAuthHandler 和/或 ProxyDigestAuthHandler 实例的 opener，具体包含哪些实例取决于开发者想要直接在 HTTP 中还是在代理中进行身份认证，以及开发者需要 Basic 还是 Digest 认证。

要想实例化这些身份认证处理程序，可以使用 HTTPPasswordMgrWithDefaultRealm 类的实例 x 作为身份认证处理程序的构造函数的唯一参数。开发者通常可以使用相同的 x 来实例化所需的所有身份认证处理程序。要想记录为给定身份认证区域和 URL 指定的用户名和密码，可以调用一次或多次调用 x.add_password。

表 19-6

add_password	*x*.add_password(*realm*,*URLs*,*user*,*password*)
	将（user, password）数据对记录在 x 中，并将其作为 URLs 参数指定的 URL 的给定安全区域 realm 中的信任状。realm 是一个指定身份认证区域的字符串，或者为 None，None 表示为没有特别记录的任意区域提供默认的信任状。URLs 是一个 URL 字符串或者一个 URL 字符串序列。对于一个 URL u，如果 URLs 中有一个项目 u1，并且 u 和 u1 的 location 组件是相同的，并且 u1 的 path 组件是 u 的一个前缀，则 u 也适用于这些信任状。其他组件（协议方案、查询和片段）不影响身份认证功能的适用性。
	下面的示例显示了如何使用 urllib2 实现基本的 HTTP 身份认证功能：
	`import urllib2`
	
	`x = urllib2.HTTPPasswordMgrWithDefaultRealm()`
	`x.add_password(None, 'http://myhost.com/', 'auser', 'apassword')`
	`auth = urlib2.HTTPBasicAuthHandler(x)`
	`opener = urllib2.build_opener(auth)`
	`urllib2.install_opener(opener)`
	
	`flob = urllib2.urlopen('http://myhost.com/index.html')`
	`for line in flob.readlines(): print line,`

19.2 Email 协议

当前，大多数电子邮件（email）都是通过实现了“简单邮件传输协议”（Simple Mail Transport Protocol，SMTP）的服务器发送，通过实现了“邮局协议，第 3 版”（Post Office Protocol version 3，POP3）的服务器接收的。Python 的标准库模块 smtplib 和 poplib 支持这些协议。除了 POP3 之外，某些服务器还实现了更丰富和更高级的“Internet 消息访问协议，第 4 版”（Internet Message Access Protocol version 4，IMAP4），Python 的标准库模块 imaplib 支持 IMAP4 协议，不过，本书没有介绍 imaplib 模块。

poplib 模块

poplib 模块提供了一个 POP3 类来访问 POP 邮箱。有关 POP 协议的规范，请参见 http://www.ietf.org/rfc/rfc1939.txt。

表 19-7

POP3	`class POP3(host,port=110)` 返回 POP3 类的一个实例 p，该实例连接到给定的 host 和 port 的 POP 服务器上。实例 p 提供了许多方法，最常用的就是以下这些方法。
dele	`p.dele(msgnum)` 标记要删除的邮件 msgnum。在连接终止时，服务器将调用 p.quit 执行删除操作。dele 将返回服务器响应字符串。
list	`p.list(msgnum=None)` 返回一个数据对（response, messages），其中 response 是服务器响应字符串，messages 是字符串列表，每个字符串包含两个单词‘msgnum bytes’，给出了邮箱中的邮件编号和每个邮件的字节长度。在 msgnum 不为 None 时，messages 只包含一个项目：给定 msgnum 的邮件编号和字节长度‘msgnum bytes’。
pass_	`p.pass_(password)` 发送密码。必须在 p.user 之后被调用。这个名称中的拖尾下画线是必需的，因为 pass 是 Python 的关键字。该方法将返回服务器响应字符串。
quit	`p.quit()` 结束会话并告诉服务器执行调用 p.dele 时请求的删除操作。该方法将返回服务器响应字符串。
retr	`p.retr(msgnum)` 返回一个包含 3 个项目的元组（response, lines, bytes），其中 response 是服务器响应字符串、lines 是由邮件 msgnum 中的所有行组成的列表，而 bytes 是邮件中的字节总数。
set_debuglevel	`p.set_debuglevel(debug_level)` 将调试级别设置为整数 debug_level：0 是默认值，表示不输出调试信息；1 表示中等数量的调试输出；而 2 或更大的值表示完整地输出与服务器交互的所有控制信息。

续表

stat	*p*.stat() 返回一个数据对（num_messages, bytes），其中 num_messages 是邮箱中的邮件数量，而 bytes 是字节总数。
top	*p*.top(*msgnum*,*maxlines*). 与 retr 类似，但是只返回邮件主题之后的邮件中的不超过 maxlines 行文本。该方法可以用于查看长邮件的起始部分。
user	*p*.user(*username*) 发送用户名 username。后面必须带一个对 p.pass_方法的调用。

smtplib 模块

smtplib 模块提供了一个 SMTP 类以向任何 SMTP 服务器发送邮件。要想了解 SMTP 的规范，请参见 http://www.ietf.org/rfc/rfc2821.txt。

表 19-8

SMTP	class SMTP([*host*,*port*=25]) 返回 SMTP 类的一个实例 s。在给定了 host（还可以选择提供 port）时，将隐式调用 s.connect(host, port)。实例 s 提供了许多方法，下面是一些最常使用的方法。
connect	*s*.connect(*host*=127.0.0.1,*port*=25) 连接到给定 host（默认值是本地主机 localhost）和 port（25 端口是 SMTP 服务的默认端口）指定的 SMTP 服务器。
login	*s*.login(*user*,*password*) 使用给定的 user 和 password 登录到服务器。只有在 SMTP 服务器要求身份认证时才需要用到该方法。
quit	*s*.quit() 终止当前的 SMTP 会话。
sendmail	*s*.sendmail(*from_addr*,*to_addrs*,*msg_string*) 从邮件地址在 from_addr 字符串中的发件人将邮件 msg_string 发送到邮件地址是 to_addrs 列表中的项目的每个收件人。msg_string 必须是一个以单个多行字符串表示的完整 RFC 822 邮件：包括主题、用于分隔的一个空白行，然后是正文。from_addr 和 to_addrs 只指示邮件的传输，而不会在 msg_string 中添加或更改主题。要想准备符合 RFC 822 标准的邮件，可以使用 Python 的 email 包，参见第 22.2 节。

19.3 HTTP 和 FTP

urllib 和 urllib2 模块通常是用来访问 http、https 和 ftp 最便捷的方法。Python 标准库还提供了用于这些协议的特殊模块。这些协议的规范请参见 http://www.ietf.org/rfc/rfc2616.txt、http://www.ietf.org/rfc/rfc2818.txt 和 http://www.ietf.org/rfc/rfc959.txt。

httplib 模块

httplib 模块提供了一个 HTTPConnect 类以连接到 HTTP 服务器。

表 19-9

HTTPConnection	`class HTTPConnection(host,port=80)` 返回 HTTPConnection 类的一个实例 h，准备连接（但是还没有真正连接）到给定的 host 和 port。 实例 h 提供了几个方法，其中最常使用的方法如下。
close	`h.close()` 关闭到 HTTP 服务器的连接。
getresponse	`h.getresponse()` 返回 HTTPResponse 类的一个实例 r，表示从 HTTP 服务器接收到的响应消息。在 request 方法返回之后调用该方法。实例 r 提供了以下属性和方法。 *r*.getheadeeypr (*name*, *default*=None) 返回首部 name 的内容，如果不存在这样的首部，则返回 default。 *r*.msg mimetools 模块的 Message 类的一个实例，参见第 22.2 节。开发者可以使用 r.msg 以访问响应消息的首部和正文。 *r*.read( ) 返回一个字符串，该字符串是服务器响应消息的正文。 *r*.reason 服务器给出的字符串，指示错误或异常（如果有的话）的原因。如果请求成功，r.reason 通常是字符串‘OK’。 *r*.status 一个整数，表示服务器返回的状态代码。如果请求成功，根据 HTTP 标准，r.status 应该是 200～299 的值。400～599 的值表示 HTTP 错误代码：例如，404 就是在无法找到请求的网页时服务器发送的错误代码。 *r*.version 如果服务器只支持 HTTP 1.0，则返回 10；如果服务器支持 HTTP 1.1，则返回 11。
request	`h.request(command,URL,data=None,headers={})` 向 HTTP 服务器发送一个请求。command 是一个 HTTP 命令字符串，比如‘GET’或‘POST’。URL 是一个 HTTP 选择器（也就是，不带协议方案和位置组件的 URL 字符串——只包含路径，后面还可能带一个查询和/或片段）。如果 data 不为 None，则是一个作为请求的正文发送的字符串，并且通常只对‘POST’和‘PUT’命令有意义。request 将计算和发送 Content-Length 首部以给出 data 的长度。要想发送这样的首部，可以在字典参数 headers 中传递该首部，并使用首部的名称作为键，首部的内容作为对应的值。 httplib 模块还提供了 HTTPSConnection 类，这个类使用了与 HTTPConnection 类完全相同的方法，区别只是该类使用 https 进行连接，而不是 http。

ftplib 模块

ftplib 模块提供了一个 FTP 类以连接到 FTP 服务器。

表 19-10

FTP	class FTP([*host*[,*user*,*passwd*='']]) 返回 FTP 类的一个实例 f。在给出了 host 参数时，将隐式调用 f.connect(host)。如果同时还给出了 user 参数（和可选的 passwd），将在调用 f.connect 之后隐式调用 f.login(user, passwd)。 实例 f 提供了许多方法，其中最常使用的方法如下。
abort	*f*.abort() 通过立即以“带外”数据的形式发送 FTP 的‘ABOR’命令，尝试中断正在进行的文件传输。
connect	*f*.connect(*host*,*port*=21) 连接到给定 host 和 port 指定的 FTP 服务器。每个实例 f 调用一次，作为 f 的第一个方法调用。如果在创建时已经给定了 host，则不需要调用该方法。
cwd	*f*.cwd(*pathname*) 将 FTP 服务器上的当前目录设置为 pathname 指定的目录。
delete	*f*.delete(*filename*) 告诉 FTP 服务器删除一个文件并返回一个字符串，该字符串是服务器的相应消息。
getwelcome	*f*.getwelcome() 返回服务器的欢迎“welcome”响应信息字符串，该字符串是在调用 f.connect 时被保存的。
login	*f*.login(*user*='anonymous',*passwd*='') 登录到 FTP 服务器。在 user 为‘anonymous’且 passwd 为‘ ’时，login 将按照通用匿名 FTP 的常规要求，确定真实用户和主机，并发送 user@host 作为密码。每个实例 f 调用一次，作为建立连接后 f 的第一个方法调用。
mkd	*f*.mkd(*pathname*) 在 FTP 服务器上创建一个名为 pathname 的新目录。
pwd	*f*.pwd() 返回 FTP 服务器上的当前目录。
quit	*f*.quit() 关闭到 FTP 服务器的连接。作为 f 的最后一个方法调用。
rename	*f*.rename(*oldname*,*newname*) 告诉 FTP 服务器将一个文件从 oldname 重命名为 newname。
retrbinary	*f*.retrbinary(*command*,*callback*,*blocksize*=8192,*rest*=None) 按二进制模式提取数据。command 是一个适当的 FTP 命令字符串，通常为‘RETR filename’。callback 是 retrbinary 为返回的每个数据块调用的回调函数，可以将数据块（也就是一个字符串）传递为该回调函数的唯一参数。blocksize 是每个数据块的最大长度。在 rest 不为 None 时，如果 FTP 服务器支持‘REST’命名，rest 表示从开发者想要开始提取的文件的起始位置开始的字节偏移量。在 rest 不为 None，并且 FTP 服务器不支持‘REST’命令时，retrbinary 将引发一个异常。

续表

retrlines	*f*.retrlines(*command*,*callback*=None) 按文本模式提取数据。command 是一个适当的 FTP 命令字符串，通常为‘RETR filename’或‘LIST’。callback 是 retrlines 为返回的每行文本调用的回调函数，可以将文本行（也就是一个字符串）传递为该回调函数的唯一参数（不带行尾标记）。在 callback 为 None 时，retrlines 将把这些文本行写入到 sys.stdout 中。
rmd	*f*.rmd(*pathname*) 在 FTP 服务器上删除一个名为 pathname 的目录。
sendcmd	*f*.sendcmd(*command*) 将字符串 command 作为一个命令发送到服务器上，并返回该服务器的响应字符串。只适合于不需要打开数据连接的命令。
set_pasv	*f*.set_pasv(*pasv*) 如果 pasv 为 True，则将被动模式设置为打开，如果 pasv 为 False，则将被动模式设置为关闭。默认打开被动模式。
size	*f*.size(*filename*) 返回 FTP 服务器上 filename 指定的文件的字节大小，如果不能确定该文件的大小，则返回 None。
storbinary	*f*.storbinary(*command*,*file*,*blocksize*=8192) 按二进制模式保存数据。command 是一个适当的 FTP 命令字符串，通常为‘STOR filename’。file 是一个按二进制模式打开的文件，storbinary 将重复调用 file.read(blocksize)读取该文件以获得要传送到 FTP 服务器上的数据。
storlines	*f*.storlines(*command*,*file*) 按文本模式保存数据。command 是一个适当的 FTP 命令字符串，通常为‘STOR filename’。file 是一个按文本模式打开的文件，storlines 将重复调用 file.readline 读取该文件以获得要传送到 FTP 服务器上的数据

下面是在交互式解释器会话中使用 ftplib 的一个典型的简单示例：

```
>>> import ftplib
>>> f = ftplib.FTP('ftp.python.org')
>>> f.login()
'230 Anonymous access granted, restrictions apply.'
>>> f.retrlines('LIST')
drwxrwxr-x   4 webmaster webmaster    512 Oct 12  2001 pub
'226 Transfer complete.'
>>> f.cwd('pub')
'250 CWD command successful.'
>>> f.retrlines('LIST')
drwxrwsr-x   2 barry    webmaster     512 Oct 12  2001 jython
lrwx------   1 root     ftp           25 Aug  3  2001 python ->
www.python.org/ftp/python
drwxrwxr-x  43 webmaster webmaster    2560 Sep  3 17:22
www.python.org
'226 Transfer complete.'
>>> f.cwd('python')
'250 CWD command successful.'
```

```
>>> f.retrlines('LIST')
drwxrwxr-x   2 webmaster webmaster    512 Aug 23  2001 2.0
  [ many result lines snipped ]
drwxrwxr-x   2 webmaster webmaster    512 Aug  2  2001 wpy
'226 Transfer complete.'
>>> f.retrlines('RETR README')
Python Distribution
===================

Most subdirectories have a README or INDEX files explaining the
contents.
  [ many result lines snipped ]
gzipped version of this file, and 'get misc.tar.gz' will fetch a
gzipped tar archive of the misc subdir.
'226 Tran.fer complete.'
>>> f.close()
>>>
```

在这样的情况下，下面这行简单得多的代码的功能相当于上面的示例：

```
print urllib.urlopen('ftp://ftp.python.org/pub/python/README').read()
```

但是 ftplib 提供了比 urllib 更多的对 FTP 操作的细节控制。因此，在某些情况下，ftplib 在程序中是非常有用的。

19.4 网络新闻

网络新闻（Network News），也被称为新闻组（Usenet），主要是使用“网络新闻传输协议”（Network News Transport Protocol，NNTP）传输的。NNTP 协议的规范请参见 http://www.ietf.org/rfc/rfc977.txt 和 http://www.ietf.org/rfc/rfc2980.txt。Python 标准库在 nntplib 模块中支持这个协议。nntplib 模块提供了一个 NNTP 类以连接到 NNTP 服务器。

表 19-11

NNTP	classNNTP(*host*,*port*=119,*user*=None,*password*=None,*readermode*=False, *usenetrc*=True) 返回 NNTP 类的一个连接到 host 和 port 的实例 n，如果 user 不为 None，则可以选择使用给定的 user 和 password 进行身份认证。在 readermode 为 True 时，还将发送一个‘mode reader’命令；开发者可能需要这样做，这取决于 NNTP 服务器和开发者发送到该服务器的 NNTP 命令。在 usenetrc 为 True 时，如果没有显式指定用户名和密码，该类将尝试从当前用户的主目录中的一个名为.netrc 的文件获取用户名和密码以进行身份认证。

响应字符串

NNTP 的实例 n 提供了许多方法。n 的每个方法都将返回一个元组，其中的第一个项目是一个字符串（下文中被称为 response），这是一个从 NNTP 服务器到对应于某个方法

(post 方法只返回 response 字符串，而不是一个元组）的 NNTP 命令的响应信息。每个方法都将返回 NNTP 服务器提供的 response 字符串。该字符串以一个十进制形式的整数开始（这个整数被称为返回码），后面带一个空格，然后是解释文本。

对于某些命令，返回码之后的额外文本只是 NNTP 服务器提供的一个注释或解释。对于其他命令，NNTP 标准制定了响应行上的返回码后面的文本的格式。在这些情况下，相关方法还将进一步解析文本，生成该方法的结果元组中的其他项目，因此，开发者的代码不需要自己执行这样的解析；只需要访问该方法的结果元组中的其他项目即可，下面几节将对此进行介绍。

形式为 2xx（任意两个数字 xx）的返回码都是成功码（也就是，这些返回码表示对应的 NNTP 命令执行成功）。其他形式的返回码，比如 4xx 和 5xx，表示 NNTP 命令执行失败。在这些情况下，该方法不返回结果，而是引发异常类 nntplib.NNTPError 的一个实例，或者该异常类的某个子类，比如，NNTPTemporaryError 错误，这表示，如果开发者重试，这个错误可能（或者不可能）被自动解决，或者 NNTPPermanentError 错误，这表示，如果开发者重试，可以确定会再次发生这个错误。在 NNTP 实例的某个方法引发一个 NNTPError 实例 e 时，该服务器的响应字符串为 str(e)，这个字符串将以像 4xx 这样的返回码开始。

方法

NNTP 的实例 n 最常使用的方法如表 19-12 所示。

表 19-12

article	*n*.article(*id*) id 是一个字符串，可以是一个包含在尖括号（<>）中的文章 ID，或者是当前组中的一个文章编号。该方法将返回一个包含 3 个字符串和一个列表的元组（response, number, id, lines），其中 number 是当前组中的文章编号、id 是包含在尖括号中的文章 ID，而 lines 是一个字符串列表，也就是文章中的文本行（标题、正文和一个空白行分隔符，但是不带行尾字符）。
body	*n*.body(*id*,*file*) id 是一个字符串，可以是一个包含在尖括号（<>）中的文章 ID，或者是当前组中的一个文章编号。该方法将返回一个包含 3 个字符串和一个列表的元组（response, number, id, lines），其中 number 是当前组中的文章编号、id 是包含在尖括号中的文章 ID，而 lines 是一个字符串列表，也就是文章的正文，但是不带行尾字符。在 file 不为 None 时，可以是一个文件名字符串，body 将打开这个文件以供写入，或者是一个已经打开，可以写入的文件对象。在任何一种情况下，body 将把文章的正文写入到已打开的文件中，并且，在这些情况下，body 返回的元组中的 lines 是一个空白列表。
group	*n*.group(*group_name*) 将 group_name 作为当前的组，并返回一个由 5 个字符串组成的元组（response, count, first, last, group_name），其中 count 是组中文章的总数、last 是最新的文章的编号、first 是最早的文章的编号，而 group_name 是当前组的名称。group_name 通常与请求的组名相同（也就是，n.group 的参数）。但是，NNTP 服务器可以建立别名或者替代名；因此，开发者可能需要检查返回的元组的最后一个项目以确定哪个新闻组被设置为当前的组。

续表

head	*n*.head(*id*) 返回一篇文章的标题。id 是一个字符串，可以是一个包含在尖括号（<>）中的文章 ID，或者是当前组中的一个文章编号。该方法将返回一个包含 3 个字符串和一个列表的元组（response, number, id, lines），其中 number 是当前组中的文章编号、id 是包含在尖括号中的文章 ID，而 lines 是一个字符串列表，也就是文章标题中的行，但是不带行尾字符。
last	*n*.last() 返回一个由 3 个字符串组成的元组（response, number, id），其中 number 是当前组中最新的（最高的）文章编号，而 id 是被包含在尖括号中的文章 ID，表示当前组中的最后一篇文章。
list	*n*.list() 返回一个数据对（response, group_stats），其中 group_stats 是一个元组列表，包含有关服务器上的每个组的信息。group_stats 的每个项目都是一个由 4 个字符串组成的列表（group_name, last, first, group_flag），其中 group_name 是这个组的名称、last 是最新的文章的编号、first 是最早的文章的编号，而在开发者被允许发表文章时，group_flag 为 'y'，在开发者不被允许发表文章时，group_flag 为 'n'，在这个组为中间的组时，group_flag 为 'm'。
newgroups	*n*.newgroups(*date*,*time*) date 是一个格式为 'yymmdd' 的用来表示日期的字符串。time 是一个格式为 'hhmmss' 的用来表示时间的字符串。newgroups 将返回一个数据对（response, group_names），其中 group_name 是由给定日期和时间之后创建的组的名称组成的列表。
newnews	*n*.newnews(*group*,*date*,*time*) group 是一个字符串，可以是一个组名，表示开发者只需要这个组中的文章的数据，或者是一个 '*'，表示开发者需要该服务器上的任何新闻组中的文章的数据。date 是一个格式为 'yymmdd' 的用来表示日期的字符串。time 是一个格式为 'hhmmss' 的用来表示时间的字符串。newnews 将返回一个数据对（response, article_ids），其中 article_ids 是由给定日期和时间之后接收到的文章的标识符组成的列表。
next	*n*.next() 将返回一个由 3 个字符串组成的元组（response, number, id），其中 number 是当前组中的下一个文章编号，而 id 是当前组中的下一篇文章的文章 ID，ID 被包含在尖括号中。当前组是通过调用 n.group 设置的。每当开发者调用 n.next 时，将接收到有关另一篇文章的信息（也就是，n 将隐式管理一个指向这个组中的当前文章的指针，在每次调用 n.next 时向前移动这个指针）。在没有下一篇文章时（也就是，当前文章就是当前组中的最后一篇文章），n.next 将引发 NNTPTemporaryError 错误（这个错误可以被认为是“临时”的，因为，可以推测将来会有更多的文章）。
post	*n*.post(*file*) 向当前的组发表一篇文章，这篇文章是从 file 中读取的。file 是一个以可读模式打开的类文件对象；post 通过重复调用 file.readline 从该文件中读取这篇文章的标题和正文。file 包含所有标题、一个空白行分隔符，然后是正文。post 将返回一个字符串，这个字符串就是从服务器到发表文章的请求的 response 消息。
quit	*n*.quit() 关闭到 NNTP 服务器的连接。可以作为对 n 的最后一个方法调用。
stat	*n*.stat(*id*) id 是一个字符串，可以是一个包含在尖括号中的文章 ID，或者是当前组中的一个文章编号。返回一个由 3 个字符串（response, number, id）组成的元组，其中 number 是当前组中的文章编号，id 是包含在尖括号中的文章 ID。

示例

下面是在交互式解释器会话中使用 nntplib 的一个典型的简单示例，这个示例使用了免费的公共 NNTP 服务器 news.gmane.org：

```
>>> import nntplib
>>> n = nntplib.NNTP('news.gmane.org')
>>> response, groups = n.list()
>>> print response
215 Newsgroups in form "group high low flags".
>>> print 'gmane.org carries', len(groups), 'newsgroups'
gmane.org carries 8094 newsgroups
>>> pg_groups = [g for g in groups if 'postgresql' in g[0]]
>>> print 'gmane.org carries', len(pg_groups), 'groups about postgresql'
gmane.org carries 1162 groups about postgresql
>>> n.group('gmane.comp.db.postgresql.announce')
 ('211 699 1 699 gmane.comp.db.postgresql.announce', '699', '1',
'699', 'gmane.comp.db.postgresql.announce')
>>> response, artnum, artid, headers = n.head('699')
>>> len(headers)
71
>>> [h for h in headers if h.startswith('Subject:')]
['Subject: EMS SQL Manager 2005 for PostgreSQL ver. 3.4 released']
>>> n.quit()
'205 .'
```

19.5 Telnet

Telnet 是一个很老的协议，RFC 854 定义了该协议的规范（参见 http://www.ietf.org/rfc/rfc854.txt），Telnet 通常用于交互式用户会话。Python 标准库在其 telnetlib 模块中支持这个协议。telnetlib 模块提供了一个 Telnet 类以连接到一个 Telnet 服务器。

表 19-13

Telnet	`class Telnet(host=None,port=23)` 返回 Telnet 类的一个实例 t。在提供了 host（可以选择提供 port）时，将隐式调用 t.open(host, port)。 实例 t 提供了许多方法，其中最常使用的方法如下。
close	`t.close()` 关闭这个连接。
expect	`t.expect(res,timeout=None)` 读取来自这个连接的数据，直到读取的数据匹配列表 res 中的项目指定的任何一个正则表达式（参见第 9.7 节中介绍的正则表达式和匹配对象）时停止读取，或者在 timeout 不为 None 时，在经过了 timeout 秒数之后停止读取。这个方法将返回一个由 3 个项目组成的元组（i, mo, txt），其中 i 是 res 中匹配的正则表达式的索引、mo 是匹配对象，txt 是直到匹配时读取的所有文本，包括匹配的文本。在连接被关闭并且没有数据可用时，将引发 EOFError 错误；否则，在没有匹配值时，返回（－1, None, txt），其中 txt 是所有读取的文本，如果在超时之前没有读取任何文本，则 txt 为 ‘’。如果读取的数据匹配 res 中的多个项目，或者 res 中的任何一个项目包含贪婪匹配字符串（比如 ‘.*’），则结果是非确定的。

续表

interact	*t*.interact() 进入交互式模式，将标准输入和输出连接到这个连接的两个通道上，就像一个哑 Telnet 客户端。
open	*t*.open(*host*,*port*=23) 连接到一个给定 host 和 port 指定的 Telnet 服务器上。每个实例 t 都将调用一次这个方法，是 t 的第一个方法调用。如果在创建时给定了 host，则不需要调用该方法。
read_all	*t*.read_all() 从这个连接读取数据，直到该连接被关闭，然后返回所有可用数据。直到连接被关闭时才会阻塞。
read_eager	*t*.read_eager() 读取并返回所有可以从无阻塞的连接读取的数据；可能是空白字符串 ‘ ’。如果连接被关闭并且没有数据可用，则引发 EOFError 错误。
read_some	*t*.read_some() 从这个连接读取并返回至少一个字节的数据，除非该连接被关闭，在被关闭的情况下，这个方法将返回 ‘’。直到至少一个字节的数据是可用的时才会阻塞。
read_until	*t*.read_until(*expected*,*timeout*=None) 从这个连接读取数据，直到发现了期望的字符串，或者在 timeout 不为 None 时，在经过了 timeout 秒数之后停止读取数据。返回在停止读取时可用的数据，可能是一个空白字符串 ‘’。如果连接被关闭并且没有数据可用，则引发 EOFError 错误。
write	*t*.write(*astring*) 将字符串 astring 写入到这个连接中。

19.6 分布式计算

当前有许多用于分布式计算的标准，从简单的“远程过程调用”（Remote Procedure Call，RPC）到功能丰富的面向对象标准，比如 CORBA。开发者可以在 Internet 上找到许多支持这些标准的第三方 Python 模块。

Python 标准库支持一个被称为 XML-RPC 的简单但是功能强大的标准，可以在服务器端和客户端使用。要想深入了解 XML-RPC，本书推荐参考 O’Reilly 出版，Simon St.Laurent 和 Joe John 编著的 *Programming Web Services with XML-RPC* 一书。XML-RPC 使用了 HTTP 或 HTTPS 作为底层传输协议，并在 XML 中对请求和回复进行编码。要想了解对服务器端的支持，参见第 22.2 节。客户端的支持是由 xmlrpclib 模块提供的。

xmlrpclib 模块提供了一个 ServerProxy 类，可以实例化这个类以连接到一个 XML-RPC 服务器。ServerProxy 的一个实例 s 是开发者要连接到的服务器的一个代理：开发者可以调用 s 上的任何方法，而 s 将把方法的名称和参数包装为一个 XML-RPC 请求、将请求发送到 XML-RPC 服务器、接收服务器的响应，然后将响应解包以获得该方法的结果。这样的方法调用的参数可以是 XML-RPC 支持的以下这些类型：

布尔型

内置 bool 常数 True 和 False；

整型、浮点型数字、字符串、数组；

传递和返回 Python 的 int、float、Unicode 和 list 类型的值；

结构体

传递和返回其键必须是字符串的 dict 类型的值；

日期

传递 xmlrpclib.DateTime 类的实例，其值按照新纪元时间开始之后的秒数表示，参见第 12 章中对 time 模块的介绍；

二进制数据

传递和返回 xmlrpclib.Binary 类的实例，其值为一个任意字节字符串。

Xmlrpclib 模块提供了以下几个类，如表 19-14 所示。

表 19-14

Binary	`class Binary(x)` x 是一个任意字节的 Python 字符串。b 将这些字节包装为一个 XML-RPC 二进制对象。
DateTime	`class DateTime(x)` x 是从新纪元时间开始之后的秒数，就像 time 模块中使用的那样，参见第 12.1 节。
ServerProxy	`class ServerProxy(uri, transport=None, encoding='utf-8', verbose=False, allow_none=False)` uri 字符串通常是服务器的 URL，可以是‘protocol://user:pass@host/...’的形式，包含用于基本的身份认证的用户名和密码；protocol 是 http 或 https，通常还会传递一个可选参数 transport，这样可以让模块为给定的协议选取正确的传输方式。开发者还可以选择传递一个可选参数 encoding 作为要使用的 8 比特编码方式的名称，可以将 verbose 传递为 True 以在执行下面的 XML-RPC 操作时得到详尽的调试信息，还可以将 allow_none 传递为 True 以将 None 添加到支持的数据类型集合中（这要求服务器支持基本 XML-RPC 协议的一个流行，但是不普遍适用的扩展协议）。 如果给定 uri 上的服务器支持自测功能，则 s 可以提供一个属性 s.system，这个属性按顺序提供了 3 个方法。 *s*.system.listMethods() 返回一个字符串列表，每个字符串表示该服务器支持的一个方法。 *s*.system.methodSignature(*name*) 返回一个字符串列表，每个字符串列表表示该服务器上的一个名为 name 的方法的签名。签名字符串是由逗号分隔的类型名组成的：第一个是返回值的类型，然后是每个参数的类型。在 name 方法没有定义的签名时，s.server.methodSignature(name)将返回一个不是列表的对象。 *s*.system.methodHelp(*name*) 返回一个包含有关 name 方法的帮助信息的字符串。这个字符串可以是普通字符串或者 HTML。在 name 方法没有定义的帮助信息时，s.server.methodHelp(name)将返回一个空白字符串‘’。

19.7 其他协议

尽管标准 Python 库支持的协议是非常丰富的，但是 Net 上使用的协议集要丰富得多。开发者可以在许多第三方扩展中找到对这些协议的支持。例如，对于 RSS 协议（参见 http://blogs.law.harvard.edu/tech/rss），可以查看 http://wiki.python.org/moin/RssLibraries，在这里可以找到许多可用模块的详细汇总。对于 SSH（参见 http://www.snailbook.com/protocols.html），一个不要求第三方参与证书授权的非常安全的协议，最好的选择可能是 paramiko，可以在 http://www.lag.net/paramiko/ 找到相关信息。SSH 通常是 Telnet、FTP 和类似的传统协议的最安全、最便捷的替代协议，而 paramiko 则是 Python 中 SSH 的一个非常优秀的实现。

第 20 章

套接字和服务器端网络协议模块

要想与 Internet 进行通信，程序需要使用被称为套接字（socket）的对象。Python 库可以通过 socket 模块支持套接字，同时还可以将套接字包装到更高级的客户端模块中，参见第 19 章。为了帮助开发者编写服务器程序，Python 库还提供了更高级的模块，开发者可以将这些模块用作套接字服务器的框架结构。标准的和第三方的 Python 模块和扩展还支持异步套接字操作。本章将将在第 20.1 节中介绍 socket 模块；在第 20.2 节中介绍服务器端框架结构模块；在第 20.3 节中介绍使用标准 Python 库模块进行异步操作；并在第 20.3 节中介绍功能丰富和强大的 Twisted 第三方软件包的一些最基础的知识。

与 C 语言层次的套接字编程相比，本章介绍的模块提供了许多便利之处。但是，最终而言，这些模块依赖于底层操作系统提供的原始套接字功能。尽管开发者通常可以通过直接使用第 19 章中介绍的模块编写有效的网络客户程序，而不需要真正理解套接字的功能，但是，要想编写有效的网络服务器程序，通常确实要求对套接字有一些理解。因此，本章介绍了底层 socket 模块，而第 19 章中并没有介绍这个模块，尽管客户程序和服务器程序都使用了套接字。

但是，本章只介绍了开发者的程序如何使用 socket 模块访问套接字；本章没有打算单独详细介绍独立于 Python 的套接字、TCP/IP 和其他网络操作技术，而开发者需要了解这些知识才可以很好地使用 socket 功能。要想详细了解任何一种类型的平台上的套接字操作，本书建议学习 Prentice Hall 出版，W. Richad Stevens 编著的 *UNIX Network Programming, Volume 1* 一书。更高级的模块通常使用更简单，功能也更强大，但是，详细地理解底层技术总是非常有用的，而且这有时候被证明为是必不可少的。

20.1 socket 模块

socket 模块提供了一个工厂函数，也被称为 socket，开发者可以调用该函数以生成

一个套接字对象 s。要想执行网络层操作，可以调用 s 上的方法。在客户程序中，可以调用 s.connect 连接到一个服务器。在服务器程序中，可以调用 s.bind 和 s.listen 等待客户程序的连接。在客户程序请求连接时，服务器程序可以调用 s.accept 接受请求，该方法将返回连接到客户程序的另一个套接字对象 s1。在有了一个连接的套接字对象之后，就可以调用该对象的 send 方法传输数据，调用该对象的 recv 方法接收数据了。

Python 支持当前的互联网协议（Internet Protocol，IP）标准。IPv4 使用更广泛；IPv6 是一个新的协议。在 IPv4 中，网络地址是一个数据对（host, port）。host 是一个像‘www.python.org’的“域名系统”（Domain Name System，DNS）主机名，或者是一个像‘194.109.137.226’这样的点分 4 十进制 IP 地址。port 是一个整数，表示一个套接字的端口编号。在 IPv6 中，网络地址是一个元组（host, port, flowinfo, scopeid）。IPv6 网络体系结构还没有被广泛部署；因此本书并没有进一步介绍 IPv6。在 host 是一个 DNS 主机名时，Python 将在开发者系统平台的 DNS 体系结构中查找这个名称，然后使用对应于这个名称的 IP 地址。

socket 模块提供了一个异常类 error。socket 的函数和方法可以引发 error 错误以诊断与套接字相关的错误。socket 模块还提供了许多方法。其中的许多函数可以翻译主机的原始格式和网络标准格式之间翻译数据，比如整数。开发者的程序和套接字连接对端的程序中在套接字上使用的更高层协议确定了开发者必须执行哪些格式转换。

socket 函数

socket 模块最常使用的函数如表 20-1 所示。

表 20-1

getdefaulttimeout	`getdefaulttimeout()` 返回一个浮点型值，这个值是新创建的套接字对象当前默认设置的超时时间（按秒计算，可能带有小数部分），如果新创建的套接字对象当前没有超时设置，则这个值为 None。
getfqdn	`getfqdn(`*`host`*`='')` 返回给定 host（这个字符串通常是一个不完整的域名）的完整的域名字符串。在 host 为‘’时，将返回本地主机的完整的域名字符串。
gethostbyaddr	`gethostbyaddr(`*`ipaddr`*`)` 返回一个由 3 个项目组成的元组（hostname, alias_list, ipaddr_list）。hostname 是一个字符串，也就是 IP 地址为传递的字符串参数 ipaddr 的主机的主名称。alias_list 是这个主机的包含零个或多个别名的列表。ipaddr_list 是这个主机的包含一个或多个点分 4 十进制 IP 地址列表。

续表

gethostbyname_ex	gethostbyname_ex(*hostname*) 与 gethostbyaddr 返回相同的结果，但是 hostname 字符串参数可以是一个点分 4 十进制 IP 地址或者是一个 DNS 名称。
htonl	htonl(*i32*) 将 32 位整数 i32 从主机格式转换为网络格式。
htons	htons(*i16*) 将 16 位整数 i16 从主机格式转换为网络格式。
inet_aton	inet_aton(*ipaddr_string*) 将 IP 地址 ipaddr_string 转换为 32 位网络格式，返回一个 4 字节字符串。
inet_ntoa	inet_ntoa(*packed_string*) 将 4 字节网络格式字符串 packed_string 转换为 IP 地址格式；返回一个点分 4 十进制 IP 地址字符串。
ntohl	ntohl(*i32*) 将 32 位整数 i32 从网络格式转换为主机格式；返回整型值。
ntohs	ntohs(*i16*) 将 16 位整数 i16 从网络格式转换为主机格式；返回整型值。
setdefaulttimeout	setdefaulttimeout(*t*) 将浮点型值 t 设置为新创建的套接字对象的默认超时设置（按秒计算，可能带有小数部分）。如果 t 为 None，则新创建的套接字对象没有超时设置。
socket	socket(*family,type*) 创建并返回一个具有给定 family 和 type 的套接字对象。fimily 通常是 socket 模块的属性 AF_INET，表示开发者想要使用一个 Internet（TCP/IP）类型的套接字。根据不同的平台，family 还可能是 socket 模块的其他属性。AF_UNIX 只能用于类 UNIX 平台，表示开发者想要使用一个类 UNIX 套接字（本书没有介绍非-Internet 套接字，因为本书主要关注跨平台 Python 功能）。type 是 socket 模块的少数几个属性之一，通常，SOCK_STREAM 表示 TCP（连接）套接字，而 SOCK_DGRAM 表示 UDP（数据报）套接字。

socket 类

socket 对象 s 提供了许多方法。最常使用的方法如表 20-2 所示。

表 20-2

accept	*s*.accept() 接受一个连接请求并返回一个数据对（s1, (ipaddr, port)）。s1 是一个新连接的套接字；ipaddr 和 port 是这个连接对端的 IP 地址和端口号。s 必须是 SOCK_STREAM；在调用该方法之前，必须已经调用了 s.bind 和 s.listen 方法。如果没有客户端试图连接，accept 将会阻塞，直到某个客户端尝试进行连接。
bind	*s*.bind((*host*,*port*)) 绑定套接字 s 以接受来自主机 host 在端口号 port 上的连接。host 可以是空白字符串 ''，表示接受来自任何主机的连接。对任何一个套接字对象 s 调用两次 s.bind 是一个错误的操作。
close	*s*.close() 关闭该套接字，终止该套接字上的任何监听或连接。在调用 s.close 之后对 s 调用任何其他方法都是一个错误操作。
connect	*s*.connect((*host*,*port*)) 将套接字 s 连接到给定 host 和 port 指定的服务器上。该方法将会阻塞，直到服务器接受或拒绝请求尝试，并在错误的情况下引发一个异常。
getpeername	*s*.getpeername() 返回一个由对端的 IP 地址和端口号组成的数据对（ipaddr, port）。s 必须是已连接的，可以是因为调用 s.connect 而连接的，或者因为 s 是由另一个套接字对象的 accept 方法生成的。
getsockname	*s*.getsockname() 返回一个由本机上的这个套接字的 IP 地址和端口号组成的数据对(ipaddr, port)。

getsockopt	*s*.getsockopt(*level*,*optname*[,*bufsize*])
	返回 s 上的一个选项的当前值。level 可以是 socket 模块为这个目的提供的 4 个常数中的任意一个：SOL_SOCKET，表示与该套接字本身有关的选项，或者 SOL_IP、SOL_TCP 或 SOL_UDP，分别表示与 IP、TCP 和 UDP 相关的选项。optname 可以是 socket 模块提供的用来标识每个套接字选项的多个常数中的任意一个，这些选项的名称以 SO_开始。bufsize 通常不需要提供，getsockopt 将返回这个选项的整型值。但是，某些选项具有的值是结构体，而不是整型值。在这些情况下，可以将相应的结构体的字节大小传递给 bufsize——getsockopt 将返回一个适合于使用 struct 模块解包的二进制字节字符串，参见第 10.4 节）。
	例如，下面是一个如何找出在默认情况下套接字是否被允许重新使用地址的示例：
	`import socket`
	`s = socket.socket()`
	`print s.getsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR)`
	`# 结果是 0，表示在默认情况下，套接字不能重新使用地址。`

gettimeout	*s*.gettimeout() 返回一个浮点型值，这个值是 s 当前设置的超时时间（按秒计算，可能带有小数部分），如果 s 当前没有超时设置，则这个值为 None。

续表

listen	*s*.listen(*maxpending*) 监听到该套接字的连接尝试，任何时候允许最多 maxpending 个排队的连接尝试。maxpending 必须大于 0，并且小于或等于一个系统设定值，在所有当前使用的系统中，这个值至少为 5。
makefile	*s*.makefile(*mode*='r') 创建并返回一个文件对象 f（参见第 10.3 节），用于从套接字读取数据和/或将数据写入到套接字中。开发者可以独立地关闭 f 和 s；只有在 f 和 s 都被关闭时，Python 才会关闭底层套接字。
recv	*s*.recv(*bufsize*) 从套接字接收最多 bufsize 个字节的数据，并返回一个由接收的数据组成的字符串。在套接字被断开连接时，返回一个空白字符串。如果套接字中当前没有数据，该方法将会阻塞，直到套接字被断开连接或者一些数据到达。
recvfrom	*s*.recvfrom(*bufsize*) 从套接字接收最多 bufsize 个字节的数据，并返回一个元组（data, (ipaddr, port)）。data 是一个由接收的数据组成的字符串，ipaddr 和 port 则是发送方的 IP 地址和端口号。在使用数据报套接字时是非常有用的，这样可以接收来自多个发送方的数据。如果套接字中没有数据，该方法将会阻塞，直到数据到达。
send	*s*.send(*string*) 在套接字上按字节发送 string 字符串。返回发送的字节数 n。n 可能会小于 len(string)；开发者必须检查发送的字节数，如有必要，需要重新发送子字符串 string[n:]。如果该套接字的缓冲区没有剩余空间，该方法将会阻塞，直到出现剩余空间。
sendall	*s*.sendall(*string*) 在套接字上按字节发送 string 字符串，该方法将会阻塞，直到所有的字节都被发送。
sendto	*s*.sendto(*string*,(*host*,*port*)) 在套接字上将 string 字符串按字节发送到目的 host 和 port 上，并返回发送的字节数 n。对数据报套接字非常有用，可以将数据发送到多个目的地。开发者必须还没有调用 s.bind 方法。n 可能会小于 len(string)；开发者必须检查发送的字节数，如果 string[n:]非空，需要重新发送 string[n:]。

使用 TCP 套接字的回传服务器和客户机

示例 20-1 显示了一个监听端口 8881 上的连接的 TCP 服务器。在连接后，服务器将循环操作，将所有数据回传（echo）给客户机，并在该客户机完成操作后回来接受另一个连接。要想终止该服务器，需要在服务器的终端窗口（控制台）上键入终止键。终止键取决于开发者的平台和设置，可能是 Ctrl-Break（通常在 Windows 平台上）或者是 Ctrl-C。

示例 20-1　TCP 回传服务器

```
import socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind(('', 8881))
sock.listen(5)

# 循环等待连接
# 在 Win32 平台上，使用 Ctrl-Break 终止，在 UNIX 平台上，使用 Ctrl-C 终止：
try:
    while True:
        newSocket, address = sock.accept()
        print "Connected from", address
        while True:
            receivedData = newSocket.recv(8192)
            if not receivedData: break
            newSocket.sendall(receivedData)
        newSocket.close()
        print "Disconnected from", address
finally:
    sock.close()
```

在这个示例中，传递给 newSocket.recv 的参数为 8192，这是一次可以接收的最大字节数。一次接收最大几千个字节的数据是在性能和内存消耗之间的一个很好的折中，并且最大字节数通常是 2 的指数（例如，8192==2**13），因为内存分配总是会舍入到这样的 2 的指数值。非常重要的是要关闭 sock（以确保尽快地释放著名的端口 8881），因此，这个示例使用了 try/finally 语句来确保调用 sock.close。关闭 newSocket 并不是至关重要的，因为这是系统分配的任意一个适当的空闲端口，因此这个示例没有为其使用 try/finally，尽管这样做也是很好的。

示例 20-2 显示了一个简单的 TCP 客户机，这个客户机连接到本地主机的 8881 端口，发送几行数据，并打印从服务器接收到的数据。

示例 20-2　TCP 回传客户机

```
import socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('localhost', 8881))
print "Connected to server"
data = """A few lines of data
to test the operation
of both server and client."""
for line in data.splitlines():
    sock.sendall(line)
    print "Sent:", line
    response = sock.recv(8192)
    print "Received:", response
sock.close()
```

在一个终端窗口中运行示例 20-1，并在多个终端窗口中运行示例 20-2。

使用 UDP 套接字的回传服务器和客户机

示例 20-3 和示例 20-4 分别实现了一个使用 UDP 套接字（也就是，使用数据报套接字，而不是流套接字）的回传服务器和客户机。

示例 20-3　UDP 回传服务器

```
import socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('', 8881))

# 循环等待数据报
# 在 Win32 平台上，使用 Ctrl-Break 终止，在 UNIX 平台上，使用 Ctrl-C 终止：
try:
    while True:
        data, address = sock.recvfrom(8192)
        print "Datagram from", address
        sock.sendto(data, address)
finally:
    sock.close()
```

示例 20-4　UDP 回传客户机

```
import socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
data = """A few lines of data
to test the operation
of both server and client."""
for line in data.splitlines():
    sock.sendto(line, ('localhost', 8881))
    print "Sent:", line
    response = sock.recv(8192)
    print "Received:", response
sock.close()
```

在一个终端窗口中运行示例 20-3 的服务器，并在多个终端窗口中运行示例 20-4。示例 20-3 和示例 20-4，以及示例 20-1 和示例 20-2，可以同时独立运行。这样并不会出现冲突和交互操作，即使所有的程序都使用了本地主机上的端口号 8881，这是因为 TCP 和 UDP 端口是分开的。如果开发者在示例 20-3 的服务器没有运行时就运行了示例 20-4，不会接收到错误消息：示例 20-4 的客户机将永远挂起，等待永远不会到来的响应。不过，数据报套接字不如连接套接字健壮和可靠。

套接字超时功能

在大多数平台上，标准 C 语言级别的套接字都没有超时的概念。在默认情况下，每个套接字都是在阻塞模式下运行，直到数据传输成功或失败。有一些高级方法需要用到

无阻塞的套接字，并确保只有在套接字不被阻塞的情况下执行套接字操作（比如依赖于 select 模块，参见第 20.3 节）。但是，显式安排这样的功能，特别是以跨平台的方式，有时候可能是复杂和困难的。

处理具有超时功能的套接字对象通常要简单一些。如果对这样的套接字对象的操作在超时时间结束之后还没有成功，也没有失败，则该操作将会失败，并产生一个异常以指示不满足超时条件。这些对象是使用无阻塞套接字和选择器内部实现的，这样将开发者的程序从这种复杂性中隔离出来，只需要处理提供了简单和直观接口的对象即可。socket 模块中的 getdefaulttimeout 和 setdefaulttimeout 函数，以及套接字对象上的 gettimeout 和 settimeout 方法都可以帮助开发者设置套接字的超时功能：每个套接字的超时值可以是按秒计算的浮点型数值（这样，还可以使用带小数的秒数），也可以是 None，这样表示一个“常规”套接字，不支持超时功能。

在“常规”套接字（其超时值 t 为 None）中，许多方法都会阻塞并“永远”等待，比如 connect、accept、recv 和 send 方法。在对超时值 t 不为 None 的套接字 s 调用这些方法时，如果在调用这些方法 t 秒之后仍然还在等待，则 s 将停止等待并引发 socket.error 错误。

20.2 SocketServer 模块

Python 库提供了一个框架模块 SocketServer 以帮助开发者实现简单的 Internet 服务器。SocketServer 提供了服务器类 TCPServer，用于使用 TCP 的面向连接的服务器，还提供了服务器类 UDPServer，用于使用 UDP 的面向数据报的服务器，并使用了相同的接口。

TCPServer 或 UDPServer 的一个实例 s 提供了许多属性和方法，开发者可以创建这两个类的子类，并改写某些方法以构建开发者自己专用的服务器框架。但是，本书中没有介绍这些高级，但是极少使用的功能。

TCPServer 类和 UDPServer 类实现了可以一次服务一个请求的同步服务器。ThreadingTCPServer 类和 ThreadingUDPServer 类实现了线程服务器，每个请求将产生一个新线程。开发者根据需要负责同步结果线程。参见第 14.1 节中对线程的介绍。

BaseRequestHandler 类

SocketServer 的常规使用方法是，首先创建 SocketServer 提供的 BaseRequestHandler 类的子类并覆盖 handle 方法。然后，实例化一个服务器类，向其传递要服务的地址对和 BaseRequestHandler 的子类。最后，对服务器实例调用 serve_forever。

BaseRequestHandler 类的实例 h 提供了以下方法和属性，如表 20-3 所示。

表 20-3

client_address	h.client_address 属性是客户机的数据对（host, port），是在连接时由基类设置的。
handle	*h*.handle() 开发者的子类将覆盖这个方法，服务器将对每个输入请求的开发者子类的一个新实例调用这个方法。对于 TCP 服务器，开发者自己实现的 handle 方法可以管理与套接字 h.request 上的客户机的会话以服务输入的请求。对于 UDP 服务器，开发者自己实现的 handle 方法将检查 h.request[0]的数据报，并使用 h.request[1].sendto 发送一个回复字符串。
request	对于 TCP 服务器，h.request 属性是连接到客户机的套接字。对于 UDP 服务器，h.request 属性是一个数据对（data, sock），其中 data 是客户机作为请求发送的数据字符串（最多 8192 字节），而 sock 则是服务器套接字。开发者自己的 handle 方法可以对 sock 调用 sendto 方法以向客户机发送一个回复消息。
server	h.server 属性是用来实例化这个处理器对象的服务器实例。

示例 20-5 使用了 SocketServer 模块来重新实现示例 20-1 中的服务器，并添加了使用线程并发服务多个客户机的功能。

示例 20-5　使用 SocketServer 实现的线程 TCP 回传服务器

```
import SocketServer
class EchoHandler(SocketServer.BaseRequestHandler):
    def handle(self):
        print "Connected from", self.client_address
        while True:
            receivedData = self.request.recv(8192)
            if not receivedData: break
            self.request.sendall(receivedData)
        self.request.close()
        print "Disconnected from", self.client_address
srv = SocketServer.ThreadingTCPServer(('',8881),EchoHandler)
srv.serve_forever()
```

在终端窗口上运行示例 20-5 中的服务器，并尝试在几个终端窗口中运行示例 20-2。还可以尝试在另一个终端窗口（或者其他平台相关的类 Telnet 程序）中运行 telnet localhost 8881 以验证长久存在的连接的行为。

HTTP 服务器

BaseHTTPServer、SimpleHTTPServer、CGIHTTPServer 和 SimpleXMLRPCServer 模块基于 SocketServer 模块实现了不同完整程度和复杂程度的 HTTP 服务器。

BaseHTTPServer 模块

BaseHTTPServer 模块提供了一个服务器类 HTTPServer，这个类是 SocketServer.TCPServer 的子类，并且按照相同的方式使用。该模块还提供了一个请求处理器类 BaseHTTPRequestHandler，这个类是 SocketServer.BaseRe-questHandler 的子类，并添加了可以用于 HTTP 服务器的属性和方法，其中最常使用的属性和方法如表 20-4 所示。

表 20-4

command	h.command 属性是客户机的 HTTP 请求的操作，比如‘get’、‘head’或‘post’。
handle	*h*.handle() 覆盖其超类的 handle 方法并将请求处理委托给名称以‘do_’开始的方法，比如，do_get、do_head 和 do_post。BaseHTTPRequestHandler 类没有提供 do_方法：从该类创建子类并提供开发者想要实现的方法。
end_headers	*h*.end_headers() 通过发送一个空白行终止响应的 MIME 首部。
path	h.path 属性是客户机的 HTTP 请求的路径，比如‘/index.html’。
rfile	h.rfile 属性是一个以可读模式打开的类文件对象，开发者可以从这个类文件对象读取作为客户机的 HTTP 请求的正文发送的数据（例如，POST 协议中 URL 编码形式的数据）。
send_header	*h*.send_header(*keyword*,*value*) 使用给定的 keyword 和 value 向响应消息添加一个 MIME 首部。在每调用一次 send_header，就会向响应消息添加另一个首部。在使用相同的 keyword 重复调用 send_header 时，就会添加多个具有相同 keyword 的首部，每次调用 send_header 都会添加一个，并按照调用 send_header 的相同顺序添加。
send_error	*h*.send_error(*code*,*message*=None) 发送一个完整的错误回复消息，其中包含 HTTP 代码 code，在 message 不为 None 时，还包含 message 字符串中的文本。
send_response	*h*.send_response(*code*,*message*=None) 发送一个完整的响应消息首部，其中包含 HTTP 代码 code，在 message 不为 None 时，还包含 message 字符串中的文本。Server 和 Date 首部总是被自动发送。
wfile	h.wfile 属性是一个以可写模式打开的类文件对象，开发者可以在调用 send_response、send_header 和 end_headers 之后将响应消息的正文写入到该类文件对象中。

下面是一个简单 HTTP 服务器的示例，该服务器将使用错误代码 404 和相应的消息‘File not found’回答每个请求：

```
import BaseHTTPServer

classTrivialHTTPRequestHandler(BaseHTTPServer.BaseHTTPRequestHandler):
    """Trivial HTTP request handler, answers not found to every
```

```
request"""
    server_version = "TrivialHTTP/1.0"
    def do_GET(self):
        """Serve a GET request."""
        self.send_error(404, "File not found")
    do_HEAD = do_POST = do_GET

server = BaseHTTPServer.HTTPServer(('',80), TrivialHTTPRequestHandler)
server.serve_forever()
```

SimpleHTTPServer 模块

SimpleHTTPServer 模块是建立在 BaseHTTPServer 之上的，提供了服务 HTTP GET 请求以访问目录中的文件所需的功能。在如何使用 BaseHTTPServer 以实现一个简单的真实世界 HTTP 服务任务这样的示例中，这个模块是最有用的。

CGIHTTPServer 模块

CGIHTTPServer 模块是建立在 SimpleHTTPServer 之上的，提供了通过 CGI 脚本服务 HTTP GET 和 POST 请求这样的功能，参见第 19 章。例如，开发者可以使用这个模块在本地计算机上调试 CGI 脚本。

SimpleXMLRPCServer 模块

XML-RPC 是一个运行在 HTTP 之上的更高层协议。Python 通过 xmlrpclib 模块支持 XML-RPC 客户机，参见第 19.6 节。SimpleXMLRPCServer 模块提供了 SimpleXMLRPCServer 类，该类使用要服务的地址对进行实例化。

SimpleXMLRPCServer 类的实例 x 提供了两个在 x.serve_forever( )之前调用的方法。

表 20-5

register_function	`x.register_function(callable,name=None)` 注册只包含单个参数的可调用函数 callable 以响应 XML-RPC 对字符串 name 的请求。name 是一个标识符，或者是一个由句点符号连接的标识符序列。在 name 为 None 时，将使用名称 callable.__name__。Callable 的参数是 xmlrpclib.loads(payload)的结果，其中 payload 是请求消息的净荷。
register_instance	`x.register_instance(inst)` 注册 inst 以使用没有通过 register_function 方法注册的名称响应 XML-RPC 请求。在 inst 提供了一个方法_dispatch 时，将使用请求名称和参数作为输入参数调用 inst._dispatch。在 inst 没有提供_dispatch 时，请求名将被用作一个属性名称以在 inst 中进行搜索。在请求名包含句点时，将重复搜索每个组件。这次搜索操作找到的属性将以请求参数作为输入参数被调用。一次只能使用 register_instance 注册一个实例：如果开发者再次调用 x.register_instance，上一次调用中传递到 x.register_instance 中的实例将被替换为下一次调用中传递的实例。

SimpleXMLRPCServer.py 模块的文档字符串中给出了 SimpleXMLRPCServer 的所有典型用法模式的简单示例，开发者可以在 Python 安装目录 Lib 中找到这个模块。下面是一个使用_dispatch 方法的简单示例。在一个终端窗口中，运行下面的简单脚本：

```
import SimpleXMLRPCServer
class with_dispatch:
    def _dispatch(self, *args):
        print '_dispatch', args
        return args
server =SimpleXMLRPCServer.SimpleXMLRPCServer(('localhost',8888))
server.register_instance(with_dispatch())
server.serve_forever()
```

从相同计算机的另一个终端窗口上的 Python 交互式会话（或者相同计算机上的一个 IDLE 交互式会话）中，现在可以运行：

```
>>> import xmlrpclib
>>> proxy = xmlrpclib.ServerProxy('http://localhost:8888')
>>> print proxy.whatever.method('any', 'args')
['whatever.method', ['any', 'args']]
```

20.3 事件驱动套接字程序

套接字程序，特别是服务器，通常必须能够同时执行多个任务。示例 20-1 接受了一个连接请求，然后服务单个客户机，直到该客户机的处理结束——其他请求必须等待。这对于作为软件产品使用的服务器而言是不可接受的。客户机不能等待太长时间：服务器必须能够同时服务多个客户机。

一种可以让程序同时执行几个任务的方法就是线程，参见第 14.1 节。SocketServer 模块可以选择支持线程，参见第 20.2 节。除线程之外，另一种可以提供更好的性能和可伸缩性的方法就是事件驱动（也被称为异步）编程。

事件驱动（Event-driven）程序被放在一个事件循环程序中，并等待事件的发生。在网络互联中，典型的事件就是“客户机请求连接”、“数据到达套接字”和“套接字可以写入”。程序将响应每个事件，执行一小段工作以服务这个事件，然后返回到事件循环程序中以等待下一个事件。Python 库提供了对事件驱动网络编程的最小支持，包括底层 select 模块和高层 asyncore 和 asynchat 模块。Twisted 包（参见 http://www.twistedmatrix.com）中提供了对事件驱动编程更丰富的支持，特别是在 twisted.internet 子包中。

select 模块

select 模块提供了一种跨平台的底层函数以实现异步网络服务器和客户机。select 模块

在类 UNIX 平台上还有其他一些功能，但是本书只介绍其跨平台功能。

表 20-6

select	`select(inputs,outputs,excepts,timeout=None)` inputs、outputs 和 excepts 分别是由等待输入事件、输出事件和异常条件的套接字对象组成的列表。timeout 是一个浮点型值，这个值表示最大等待的秒数。在 timeout 为 None 时，该模块没有最大等待时间：select 将保持等待，直到有的套接字对象接收到事件。在 timeout 为 0 时，select 将立刻返回，不会等待，不管这些套接字对象是否已经收到了事件。 select 返回一个由 3 个项目组成的元组（i, o, e）。i 是一个由 inputs 中的零个或多个项目组成的列表，也就是接收到的输入事件。o 是一个由 outputs 中的零个或多个项目组成的列表，也就是接收到的输出事件。e 是一个由 excepts 中的零个或多个项目组成的列表，也就是接收到的异常条件（例如，带外数据）。i、o 和 e 都可以是空白列表，但是，如果 timeout 为 None，则至少必须有一个列表是非空的。 除了套接字之外，开发者还可以在 inputs、outputs 和 excepts 列表中包含其他一些对象，这些对象可以提供一个 fileno 方法，可以不带任何参数进行调用，并返回一个套接字的文件描述符。例如，SocketServer 模块的服务器类遵循了这个协议，参见第 20.2 节。因此，开发者可以在这些列表中包含这些类的实例。在类 UNIX 平台上，select.select 具有更广泛的适用性，因为 select.select 还可以接受与套接字无关的文件描述符。但是，在 Windows 平台上，select.select 只能接受与套接字相关的文件描述符。

示例 20-6 中使用了 select 模块来重新实现示例 20-1 中的服务器，并添加了功能以同时服务任意数量的客户机。

示例 20-6　使用 select 的异步 TCP 回传服务器

```
import socket
import select
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind(('', 8881))
sock.listen(5)

# 用于监视输入和输出事件的套接字列表
ins = [sock]
ous = []
# 映射套接字 -> 在可行情况下，这个套接字上发送的数据
data = {}
# 映射套接字 -> 正在运行的客户机的(主机，端口号)
adrs = {}

try:
    while True:
        i, o, e = select.select(ins, ous, [])  # 既没有异常，也没有超时参数
        for x in i:
            if x is sock:
```

```
            # 输入事件为 sock 表示一个客户机正在尝试连接
            newSocket, address = sock.accept()
            print "Connected from", address
            ins.append(newSocket)
            adrs[newSocket] = address
    else:
        # 其他输入事件表示数据到达，或者断开连接
        newdata = x.recv(8192)
        if newdata:
            # 数据到达，准备对该数据的响应，并将其放到队列中。
            print "%d bytes from %s" % (len(newdata), adrs[x])
            data[x] = data.get(x, '') + newdata
            if x not in ous: ous.append(x)
        else:
            # 断开连接事件，给出一个消息并清理现场
            print "disconnected from", adrs[x]
            del adrs[x]
            try: ous.remove(x)
            except ValueError: pass
            x.close()
            ins.remove(x)
    for x in o:
        # 输出事件总是表示可以发送某些数据
        tosend = data.get(x)
        if tosend:
            nsent = x.send(tosend)
            print "%d bytes to %s" % (nsent, adrs[x])
            # 请记住还需要发送的数据（如果有的话）
            tosend = tosend[nsent:]
        if tosend:
            print "%d bytes remain for %s" % (len(tosend), adrs[x])
            data[x] = tosend
        else:
            try: del data[x]
            except KeyError: pass
            ous.remove(x)
            print "No data currently remain for", adrs[x]
finally:
    sock.close()
```

进行这样的底层编程会造成相当的复杂性，示例 20-6 和其中的数据结构显示了这种复杂程度。在终端窗口中运行示例 20-6 的服务器，并尝试运行几个示例 20-2。还可以尝试在其他终端窗口中运行 telnet localhost 8881（或者其他平台相关的类 Telnet 程序）以验证长时间存在的连接的行为。

asyncore 和 asynchat 模块

asyncore 和 asynchat 模块可以实现比 select 模块可以提供的更高级，更有生产力的异步网络服务器和客户机。

asyncore 模块

asyncore 模块提供了一个函数。

表 20-7

loop	`loop()`
	这个函数实现了异步事件循环操作，可以将所有网络事件分发到以前被实例化的分发器对象。在所有分发器对象（也就是，所有的通信通道）都被关闭时，loop 循环将会终止。
	asyncore 模块还提供了 dispatcher 类，该类提供了套接字对象的所有方法，以及用于事件驱动编程的特殊方法，这些方法的名称都是以‘handle_’开始的。开发者可以创建 dispatcher 的子类并覆盖 handle_方法以处理所需的所有事件。要想实例化 dispatcher 类的一个实例 d，可以向其传递一个参数 s，s 是一个已经连接的套接字对象。否则，必须首先调用：
	`d.create_socket(socket.AF_INET,socket.SOCK_STREAM)`
	然后对 d 调用 connect 以连接到一个服务器，或者调用 bind 和 listen 以让 d 本身成为一个服务器。dispatcher 的子类 X 的实例 d 最常使用的方法如下。

create_socket	`d.create_socket(family,type)` 使用给定的 family 和 type 创建 d 的套接字。family 通常是 socket.AF_INET。type 通常是 socket.SOCK_STREAM，因为类分发器通常使用的是 TCP（也就是，基于连接）套接字。
handle_accept	`d.handle_accept()` 在一个新客户机已经连接了之后调用该方法。类 X 通常通过调用 self.accept 进行响应，然后使用作为结果的新套接字实例化 dispatcher 的另一个子类 Y，从而处理新的客户机连接。 开发者实现的 handle_accept 不需要返回 Y 的新实例：dispatcher 的子类的所有实例将使用 dispatcher._ _init_ _中的 asyncore 框架注册其本身，这样，以后 asyncore 将回调这些实例。
handle_close	`d.handle_close()` 在连接关闭时调用这个方法。
handle_connect	`d.handle_connect()` 在连接开始时调用这个方法。
handle_read	`d.handle_read()` 在套接字有了可以无阻塞读取的新数据时调用这个方法。
handle_write	`d.handle_write()` 在套接字有一些空闲缓冲区空间时调用这个方法，因此开发者可以无阻塞写入。 asyncore 模块还提供了 dispatcher_with_send 类，这个类是 dispatcher 的一个子类，覆盖了其中的一个方法。
send	`d.send(data)` 在 dispatcher_with_send 类中，d.send 方法等同于套接字对象的 send_all 方法，这两个方法都用来发送所有数据。但是，d.send 并不同时发送所有数据，而且也不会阻塞；而是，d 将以每 512 字节一个的小数据包发送数据以响应 handle_write 事件（回调）。在简单情况下，这种策略可以确保好的性能。

示例 20-7 可以使用 asyncore 模块来重新实现示例 20-1 中的服务器，并添加了的功能以同时服务任意数量的客户机。

示例 20-7　使用 asyncore 的异步 TCP 回传服务器

```
import asyncore
import socket

class MainServerSocket(asyncore.dispatcher):
    def __init__ (self, port):
        asyncore.dispatcher. __init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.bind(('',port))
        self.listen(5)
    def handle_accept(self):
        newSocket, address = self.accept()
        print "Connected from", address
        SecondaryServerSocket(newSocket)

class SecondaryServerSocket(asyncore.dispatcher_with_send):
    def handle_read(self):
        receivedData = self.recv(8192)
        if receivedData: self.send(receivedData)
        else: self.close()
    def handle_close(self):
        print "Disconnected from", self.getpeername()

MainServerSocket(8881)
asyncore.loop()
```

与示例 20-1 相比，示例 20-7 的复杂度是适中的。添加的服务多个客户机的功能具有异步事件驱动编程的高性能和高可伸缩性，并且实现简单，这要感谢 asyncore 的强大功能。

请注意，SecondaryServerSocket 的 handle_read 方法可以很自由地使用 self.send：SecondaryServerSocket 是 dispatcher_with_send 的子类，很好地覆盖了 send 方法。如果开发者已经选择了直接使用 asyncore.dispatcher，则不能这样做。

asynchat 模块

asynchat 模块提供了 async_chat 类，这个类是 asyncore.dispatcher 的子类，并添加了一些方法以支持缓冲和面向行（line）的协议。创建 async_chat 的子类并覆盖其中的某些方法。async_chat 的子类的实例 x 最常使用的附加方法如表 20-8 所示。

表 20-8

collect_incoming_data	*x*.collect_incoming_data(*data*) 只要一个字节字符串 data 数据到达，就调用这个方法。通常，x 将把 data 添加到 x 保存的某个缓冲区中；一般来说，缓冲区是一个列表，而 x 通过调用该列表的 append 方法添加数据。
found_terminator	*x*.found_terminator() 只要找到了 set_terminator 方法设置的终止符，就调用这个方法。通常，x 将处理其保存的缓冲区，然后清除该缓冲区。
push	*x*.push(*data*) 开发者的类通常不覆盖这个方法。该方法在其基类 async_chat 中的实现可以将字符串 data 添加到一个用于发送的输出缓冲区中。因此，push 方法非常类似于 asyncore.dispatcher_with_send 类的 send 方法，但是 push 方法具有更成熟复杂的实现，以在更多情况下确保好的性能。
set_terminator	*x*.set_terminator(*terminator*) 开发者的类通常不覆盖这个方法。terminator 通常是‘\r\n’，也就是大多数 Internet 协议指定的行终止符。terminator 还可以为 None，以禁用对 found_terminator 的调用。

示例 20-8 使用 asynchat 模块重新实现了示例 20-7 中的服务器，该示例使用了 asynchat.async_chat，而不是 asyncore.dispatcher_with_send。要想突出表现 async_chat 的典型使用，示例 20-8 只在已经收到了一个完整的行（也就是，一个以\n 结束的行）时才会响应（通过将接收的数据回传到客户机进行响应，就像本章中的所有其他服务器示例一样）。

示例 20-8 使用 asynchat 的异步 TCP 回传服务器

```
import asyncore, asynchat, socket

class MainServerSocket(asyncore.dispatcher):
    def __init__(self, port):
        print 'initing MSS'
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.bind(('',port))
        self.listen(5)
    def handle_accept(self):
        newSocket, address = self.accept()
        print "Connected from", address
        SecondaryServerSocket(newSocket)

class SecondaryServerSocket(asynchat.async_chat):
    def__init__(self, *args):
        print 'initing SSS'
        asynchat.async_chat__init__(self, *args)
```

```
        self.set_terminator('\n')
        self.data = []
    def collect_incoming_data(self, data):
        self.data.append(data)
    def found_terminator(self):
        self.push(''.join(self.data))
        self.data = []
    def handle_close(self):
        print "Disconnected from", self.getpeername()
        self.close()

MainServerSocket(8881)
asyncore.loop()
```

为了试验示例 20-8，这里不能使用示例 20-2 作为客户机，因为示例 20-2 的代码不能确保只发送以\n 结束的整行数据。不过，要修改这个功能也是很简单的。例如，下面的客户程序就非常适合于测试示例 20-8，以及本章中的任何其他服务器示例。

```
import socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('localhost', 8881))
print "Connected to server"
data = """A few lines of data
to test the operation
of both server and client."""
for line in data.splitlines():
    sock.sendall(line+'\n')
    print "Sent:", line
    response = sock.recv(8192)
    print "Received:", response
sock.close()
```

这段代码与示例 20-2 的唯一区别就是更改了循环体的第一行中的 sock.sendall 函数调用的参数。这段代码只是添加了一个行终止符‘\n’以确保该客户程序可以与示例 20-8 交互操作。

Twisted 框架

Twisted 包（参见 http://www.twistedmatrix.com）是一个可用的免费网络客户机和服务器框架。Twisted 包含像 Web 服务器、用户认证系统、电子邮件服务器和客户端、即时消息、SSH 客户端和服务器、DNS 服务器和客户端等这些强大、高级的组件，以及底层基础架构，所有这些高级组件都建立在该底层基础架构之上的。每个组件都是高可伸缩的，很容易定制，并且所有的组件都可以被平滑地集成以进行互操作。Twisted 包是为 Python 的强大功能和 Twisted 开发者的灵活性准备的礼物，这些功能是值得下载 2M 字节的数据来完成的。

twisted.internet 和 twisted.protocols 包

“Twisted Core”是 Twisted 的底层部分，支持事件驱动客户机和服务器，集中在 twisted.internet 模块和 twisted.protocols 模块上。twisted.protocols 模块提供了协议处理器和工厂。twisted.internet 模块提供了反应器（reactor）对象，该对象具体化了事件循环的概念。要想最佳使用 Twisted Core，需要很好地理解分布式计算中使用的涉及模式。华盛顿大学分布式对象计算中心的 Douglas Schmidt 在 http://www.cs.wustl.edu/~schmidt/patterns-ace.html 网站上给出了这些设计模式的说明。

twisted.protocols 提供了许多使用 twisted.internet 基础架构的协议，包括 FTP、HTTP、Finger、GPS-NMEA、IRC、Jabber、NNTP、POP3、IMAP4、SMTP、SIP、SocksV4 和 Telnet。

反应器

反应器（reactor）对象可以建立协议工厂作为给定 TCP/IP 端口（或其他传输协议，比如 SSL）上的接收程序（服务器），还可以建立连接协议处理器作为客户机。开发者可以根据导入的模块选择不同的反应器实现（反应器是在程序运行中第一次导入一个反应器模块时被实例化的：不需要调用任何工厂函数）。默认的反应器使用了第 20.3 节中介绍的 select 模块。其他的专用反应器是与许多 GUI 工具包的事件循环集成在一起的，或者使用了平台专用技术，比如 Windows 的事件循环，某些类 UNIX 系统上的 select 模块中支持的 poll 系统调用，以及一些更专用的系统调用，比如 FreeBSD 的 kqueue。默认的反应器通常就已经足够了，但是可以使用其他实现这种额外的灵活性可以帮助开发者集成 GUI 或其他平台专用功能，或者实现更高的性能和可伸缩性。

反应器对象 r 实现了许多接口，每个接口提供了一些方法。使用 twisted.internet 模块实现 TCP/IP 客户机和服务器的程序最常使用的反应器方法如表 20-9 所示。

表 20-9

callLater	*r*.callLater(*delay*,*callable*,**args*,***kwds*) 调度在现在时刻之后延时 delay 秒调用 callable(*args, **kwds)。delay 是一个浮点型值，因此可以表示小数形式的秒数。
callInThread	*r*.callInThread(*callable*,**args*,***kwds*) 在一个不同于反应器的工作者线程中调用 callable(*args, **kwds)。
callFromThread	*r*.callFromThread(*callable*,**args*,***kwds*) 在反应器的线程中调用 callable(*args, **kwds)。只能在一个单独的线程中调用 r.callFromThread，并且本质上并不同步。
callWhenRunning	*r*.callWhenRunning(*callable*,**args*,***kwds*) 调度在 r 运行时调用 callable(*args, **kwds)。
connectTCP	*r*.connectTCP(*host*,*port*,*factory*,*timeout*=30,*bindAddress*=None) 建立 factory，factory 必须是 ClientFactory 类（或者是 ClientFactory 的任何子类）的一个实例，并将 factory 作为连接到给定 host 和 port 上的 TCP 客户端的协议处理工厂。如果在 timeout 秒中没有发生连接操作，则表示连接尝试被认为失败。在 bindAddress 不为 None 时，就是一个由两个项目组成的元组（clienthost, clientport），客户机将绑定到本地主机和端口上。

续表

listenTCP	r.listenTCP(port,factory,backlog=50,interface='') 建立 factory，factory 必须是 ServerFactory 类的一个实例（或者 ServerFactory 的任何子类），并将 factory 作为在给定 port 上监听的 TCP 服务器的协议处理器工厂。在任何给定时间，可以保持不超过 backlog 个客户端在队列中等待连接。在 interface 不为 '' 时，将绑定到主机名 interface。
run	r.run() 运行反应器的事件循环操作，直到调用 r.stop()。
stop	r.stop() 停止反应器通过调用 r.run()开始的事件循环操作。

传输

传输对象具体化了一个网络连接。每个协议对象可以对 self.transport 调用方法以将数据写入到其对端并断开连接。传输对象 t 提供了以下方法。

getHost	t.getHost() 返回一个标识连接的这一端的对象 a。a 的属性是 type（一个像 'TCP' 或 'UDP' 这样的字符串）、host（一个使用点分 4 十进制表示的 IP 地址字符串）和 port（一个标识端口号的整数）。
getPeer	t.getPeer() 返回一个标识连接对端的对象（很容易与代理、伪装、NAT、防火墙等混淆），并且与 getHost 的结果具有相同的类型。
loseConnection	t.loseConnection() 告诉 t 在完成了写入所有等待数据之后立即断开连接。
write	t.write(data) 将字符串 data 传输到对端，或者将其放在队列中以进行传输。t 将尽力确保传递到 write 的所有数据最终都被发送。
writeSequence	t.writeSequence(seq) 将可迭代参数 seq 的每个字符串项目数据传输到对端，或者将其放到队列中，最后进行传输。t 将尽力确保传递到 writeSequence 的所有数据最终都被发送。 特殊的传输将把某些方法添加到这个小集合中；例如，TCP 传输还有一些方法可以设置和获得 SO_KEEPALIVE 和 TCP_NODELAY 属性，以及一个 SSL 传输，除了这些之外，还提供了一个更进一步的方法以获得对端（连接的另一端）的证书信息。

协议处理器和工厂

反应器可以使用一个工厂实例化协议处理器，并在事件发生时对协议处理器实例调用方法。协议处理器是 Protocol 类的子类，并覆盖了某些方法。协议处理器可能会使用其

工厂（可以是 self.factory）作为一个仓库以处理需要在处理器之间数据共享，或者在多个实例之间持久化数据的情况。协议工厂可以继承为 Factory 类的子类，但是这种子类继承通常是不必要的，因为在大多数情况下，Factory 提供了所需的所有类。只需要将 Factory 实例 f 的 protocol 属性设置为一个类对象，该类对象是 Protocol 的一个适当的子类，然后将 f 传递给该反应器。

Protocol 的一个子类的实例 p 提供了以下方法，如表 20-10 所示。

表 20-10

connectionLost	`p.connectionLost(reason)` 在到对端的连接被关闭时调用该方法。参数 reason 是一个对象，解释了为什么连接被关闭。reason 不是一个 Python 异常实例，但是 reason 有一个属性 reason.value，通常就是这样的一个实例。开发者可以使用 str(reason)获得一个解释字符串，包括简单的跟踪信息，或者 str(reason.value)以获得只包含解释信息，而不包含任何跟踪信息的字符串。
connectionMade	`p.connectionMade()` 在到对端的连接成功建立之后调用该方法。
dataReceived	`p.dataReceived(data)` 在从对端接收到字符串 data 之后调用该方法。
makeConnection	`p.makeConnection(transport)` 使用给定 transport 实例化一个连接，并在连接尝试成功时调用 p.connectionMade，或者在连接尝试失败时调用 p.connectionLost。

使用 Twisted 的回传服务器

示例 20-9 使用了 twisted.internet 以实现一个回传服务器，该服务器具有同时服务任意数量的客户端的功能。

示例 20-9　使用 twisted 的异步 TCP 回传服务器

```
from twisted.internet import protocol, reactor

class EchoProtocol(protocol.Protocol):
    def connectionMade(self):
        p = self.transport.getPeer()
        self.peer = '%s:%s' % (p.host, p.port)
        print "Connected from", self.peer
    def dataReceived(self, data):
        self.transport.write(data)
    def connectionLost(self, reason):
        print "Disconnected from %s: %s" % (self.peer, reason.value)

factory = protocol.Factory()
```

```
factory.protocol = EchoProtocol

reactor.listenTCP(8881, factory)
def hello(): print 'Listening on port', 8881
reactor.callWhenRunning(hello)
reactor.run()
```

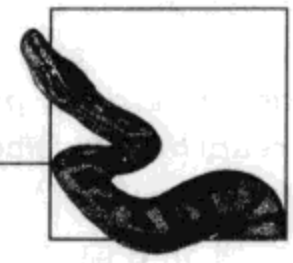
示例 20-9 展示了至少与示例 20-7 一样好的可伸缩性，然而这个示例是本章中回传服务器示例中最简单的一个——很好地表现了 Twisted 的强大功能和简单性，即使是在用于这样的底层任务时。请注意这条语句：

```
factory.protocol = EchoProtocol
```

这条语句将把类对象 EchoProtocol 绑定为 factory 对象的 protocol 属性。赋值语句的右边不能是 EchoProtocol( )，在类名之后带圆括号。这样的右边将调用，并因此实例化 EchoProtocol 类，因此，这条语句还将把一个协议实例对象，而不是一个协议类对象绑定到 factory.protocol。这样的错误可能会导致服务器非常快速地失败。

有一本面向任务的书，显示了如何使用 Twisted 实现各种底层和高层任务，请参考 O'Reilly 出版，Abe Fettig 编著的 *Twisted Network Programming Essentials* 一书。

第 21 章



CGI 脚本和其他解决方案

在 Web 浏览器（或任何其他 Web 客户端）从一个 Web 服务器请求一个网页时，服务器可能会返回静态或动态内容。要想服务动态内容，需要服务器端的 Web 程序生成并传送随时产生的内容，这些内容通常基于数据库中保持的信息。服务器端编程的长期存在的 Web 网络标准被称为 CGI，表示“公共网关接口”（Common Gateway Interface）：

1．Web 客户端（通常是 Web 浏览器）向 Web 服务器发送一个结构化的请求；

2．服务器执行另一个程序，传递请求的内容；

3．服务器捕获其他程序的标准输出；

4．服务器将输出作为对原始请求的响应发送到客户端。

换句话讲，服务器的角色就是客户端和其他程序之间的网关。这里的其他程序被称为 CGI 程序，或者 CGI 脚本。

CGI 充分体现了标准的典型好处。在使用 CGI 标准编程时，程序可以部署在所有 Web 服务器上，不管 Web 服务器有什么区别，程序都可以正常工作。本章主要介绍 Python 中的 CGI 脚本。本章还将介绍 CGI 的不足之处（基本上讲，就是高负载下的可伸缩性问题），并在第 21.3 节中介绍了其他一些不同于 CGI 的替代方案，开发者可以使用的非标准的服务器端体系结构。当前，非标准的替代方案通常都是很高级的，因为这些方案不会过多地限制开发者的部署，并且这些方案可以支持更高级的抽象以增强生产力，例如，使用第 21.2 节中介绍的 Cookie 模块隐式和透明使用 cookie 以提供“会话”抽象（在 CGI 中，如果开发者想要任何会话具有连续性，必须直接处理 cookie）。但是，至少对于底层的 CGI 替代方案（比如 FastCGI，参见第 21.3 节）而言，开发者从 CGI 编程学习到的大多数知识还是可以用得上的。

本章假定读者已经熟悉了 HTML 和 HTTP。对于这些标准的参考资料，可以参考 O’Reilly

出版，Stephen Spainhour 和 Robert Eckstein 编著的 *Webmaster in a Nutshell* 一书。要想详细了解 HTML，本书推荐 O'Reilly 出版，Chuck Musciano 和 Bill Kennedy 编著的 *HTML & XHTML: The Definitive Guide* 一书。要想了解 HTTP 的其他内容，请参考 O'Reilly 出版，Cliton Wong 编著的 *HTTP Pocket Reference* 一书。

21.1 Python 中的 CGI

CGI 标准可以让开发者使用任何语言编写 CGI 脚本。Python 是一个非常高层的、高生产力的语言，因此非常适合于 CGI 编码。Python 标准库提供了一些模块以处理典型的 CGI 相关任务。

表单提交方法

CGI 脚本通常用来处理提交的 HTML 表单。在这种情况下，form 标签的 action 属性指定了用来处理该表单的 CGI 脚本的 URL，并且 method 属性为 GET 或 POST，该属性指定了如何将表单数据发送到该脚本。根据 CGI 标准，GET 方法只能用于不会有负面作用的表单，比如请求服务器查询数据库并显示结果，而 POST 方法用于可能会有负面作用的表单，比如请求服务器更新数据库。但是，实际上 GET 通常也可以用于创建负面作用。在实际使用中，GET 和 POST 之间的区别就是，GET 将把表单的内容编码为一个连接到 action URL 的查询字符串以形成一个更长的 URL，而 POST 将把表单的内容作为编码的数据流进行传输，CGI 脚本将把这些数据流看作是标准输入。

GET 稍微快一些。在可以使用超链接的时候可以使用一个固定的 GET 形式的 URL。但是，GET 不能向服务器发送大量的数据，因为许多客户机和服务器都限制了 URL 的长度（安全值是最多大约 200 字节）。POST 方法没有大小限制。在表单包含 type=file 的 input 标签时，必须使用 POST 方法，然后，form 标签必须包含 enctype=multipart/form-data。

CGI 标准没有指定单个脚本是否可以访问查询字符串（用于 GET）和脚本的标准输入（用于 POST）。许多客户机和服务器都可以摆脱这个问题，但是，依赖于这种非标准经验实际上可能会消除 CGI 作为一个标准而带来的好处。Python 的标准模块 cgi（参见第 21.1 节）可以恢复只来自于查询字符串的表单数据，但是只能在给出了任意查询字符串的时候；否则，在没有给出查询字符串时，cgi 将恢复来自于标准输入的表单数据。

cgi 模块

cgi 模块提供了 CGI 脚本经常使用的一个方法和一个类。

表 21-1

escape	`escape(str,quote=False)` 返回字符串 str 的一个副本，使用适当的 HTML 实体（&, <, >）替换字符&、< 和 >。在 quote 为 True 时，escape 还可以使用&quot 替换双引号字符(")。escape 可以让脚本准备文本字符串以输出在一个 HTML 文档中，不管该字符串是否包含 HTML 特殊字符。
FieldStorage	`class FieldStorage(keep_blank_values=0)` 在开发者的脚本实例化一个 FieldStorage 实例 f 时，cgi 模块可以适当地解析查询字符串和/或标准输入。cgi 隐藏了 POST 和 GET 之间的区别。cgi 脚本只能实例化 FieldStorage 一次，因为该实例化操作可能会占用标准输入。 f 是一个映射：f 的键是表单的控件的名称属性。在 keep_blank_values 为 True 时，f 还包含值为空白字符串的控件。在默认情况下，f 将忽略这样的控件。f 提供了 dict 功能的一个子集：开发者可以对 f 进行迭代操作以获得每个键 n (for n in f)，检查键 n 是否存在（if n in f），并对 f 进行索引以获得键 n 的值 (f[n])。获得的值可以是： • 如果 name n 在表单中出现了不止一次，则值为 k 个 FieldStorage 实例组成的列表（k 是 n 出现的次数）； • 如果 name n 在表单中只出现了一次，则值为单个 FieldStorage 实例。 name 的出现计数遵循 HTML 表单规则。radio 或 checkbox 组控件共享一个 name，但是整个组只能算作这个名称的一次出现。 FieldStorage 实例中的值是按照处理嵌套表单的顺序排列的 FieldStorage 实例。实际上，开发者并不需要这样的复杂性。对于每个嵌套的实例，只需要访问值（偶尔访问其他属性），并忽略潜在的嵌套映射即可。要避免类型测试：cgi 模块可以使用 MiniFieldStorage 的实例最优化一个轻量级的签名兼容类，而不是 FieldStorage 实例。开发者可能会提前知道哪些 name 在表单中是重复的，从而知道 f 的哪些项目是列表。在开发者不知道时，可以使用 try/except 查找，而不是使用类型测试（参见第 6.6 节以了解详细信息）。更好的是，可以使用 f 的下面两个方法。
getfirst	`f.getfirst(key,default=None)` 在 key 在 f 中，并且 f[key].value 是单个值，而不是列表时，getfirst 将返回 f[key].value。在 key 在 f 中，并且 f[key].value 是一个列表时，getfirst 将返回 f[key].value[0]。在 key 不在 f 中时，getfirst 将返回 default。 在开发者知道表单（脚本的输入来自于这个表单）中最多只有一个名为 key 的输入字段时，可以使用 getfirst 方法。
getlist	`f.getlist(key)` 在 key 在 f 中，并且 f[key].value 是单个值，而不是一个列表，getlist 将返回 [f[key].value]，也就是，一个唯一的项目是 f[key].value 的列表。在 key 在 f 中，并且 f[key].value 是一个列表，getlist 将返回 f[key].value。在 key 不在 f 中时，getlist 将返回空白列表[ ]。

续表

getlist	在开发者知道表单（脚本的输入来自于这个表单）中可以有多于一个的名为 key 的输入字段时，可以使用 getlist 方法。 FieldStorage 类的实例 f 提供了以下属性： disposition Content-Disposition 首部，如果没有给出这样的首部，则为 None； disposition_options 使用 Content-Disposition 首部中的所有选项（如果有的话）建立的映射； headers 所有首部的映射：首部的名称作为映射的键，首部的值作为映射的值； file 一个类文件对象，可以从该对象读取控件的值；如果该值以字符串的形式保存在内存中，则为 None，大多数控件都是这种情况； filename 客户端指定的文件名，用于 file 控件；如果没有指定该文件名，则为 None； name 控件的 name 属性，如果没有这样的属性，则为 None； type Content-Type 首部，如果没有这样的首部，则为 None； type_options 使用 Content-Type 首部中的所有选项（如果有的话）建立的映射； value 控件的值是字符串；如果 f 在一个文件中保存该控件的值，则在访问 f.value 时，f 将把该文件隐式读取到内存中； 在大多数情况下，只需要 value 属性就可以了。其他属性都可以用于 file 控件，可能会包含像 Content-Type 和 Content-Disposition 首部这样的元数据。共享一个名称 name 的 checkbox 控件的值和多个选择的 select 控件都是字符串，以逗号分隔的选项列表表示。习惯用法是： `values=f.getfirst(n,'').split(',')` 将这样的组合值字符串分开，并放到一个由单独的组件字符串组成的列表中。

CGI 输出和错误

在服务器运行一个 CGI 脚本以满足一个请求时，对该请求的响应是这个脚本的标准输出。该脚本必须输出 HTTP 首部，然后是一个空白行，接着是响应正文。特别是，脚本必须总是输出一个 Content-Type 首部。最常见的是，脚本输出如下的 Content-Type 首部：

```
Content-Type: text/html
```

在这种情况下，响应正文必须是 HTML。但是，脚本还可以选择输出一个 text/plain 的内容类型（也就是，响应正文必须是普通文本），或者任何其他 MIME 类型，后面带一个符合该 MIME 类型的响应正文。MIME 类型必须与客户端发送的 Accept 首部（如果有的话）兼容。

下面是经典的“Hello World!”示例中可能出现的最简单的 Python CGI 脚本，忽略其输入，并只输出一行纯文本：

```
print "Content-Type: text/plain"
print
print "Hello, CGI World!"
```

通常，开发者可能还需要输出 HTML，而这几乎同样简单：

```
print "Content-Type: text/html"
print
print "<html><head><title>Hello, HTML</title></head>"
print "<body><p>Hello, CGI and HTML together!</p></body></html>"
```

浏览器将会非常宽容地解析 HTML：开发者可以得到这段代码的不带 HTML 结构标签的输出结果，但是，结果会完全正确，并且几乎没有任何开销。要想了解生成 HTML 输出的其他一些方法，参见第 23.5 节。

Web 服务器将从 CGI 脚本收集所有输出信息，然后将这些信息合并在一起发送到客户端浏览器上。因此，开发者不能向客户端发送处理信息，而只是最终结果。

如果开发者需要输出二进制数据（在二进制和文本文件有区别的平台上，也就是，Windows），必须确保在运行 Python 时使用了-u 开关，参见第 3.1 节。更健壮的方案就是使用第 22.1 节中介绍的编码模块（典型的情况是使用 Base 64 编码）和适当的 Content-Transfer-Encoding 首部，对输出结果进行文本编码。然后，符合标准的浏览器将根据 Content-Transfer-Encoding 首部对输出信息进行解码，并还原编码的二进制数据。编码会导致输出信息扩大 30%，这样有时候会带来性能问题。在这种情况下，最好确保脚本的标准输出流是一个二进制文件。要想确保 Windows 上的二进制输出，下面是一个可以替代-u 开关的用法：

```
try: import msvcrt, os
except ImportError: pass
else: msvcrt.setmode(1, os.OS_BINARY)
```

错误消息

如果异常从开发者的脚本开始传播，Python 将把跟踪诊断信息输出到标准错误上。对于大多数 Web 服务器而言，错误信息最终都会被放到错误日志中。客户端浏览器将接收到一个简明的通用错误消息。如果开发者可以访问该服务器的错误日志，这样做也是可以的。但是，在开发者调试 CGI 脚本时，能在客户端浏览器上看到详细的错误信息可以让事情变得更简单一些。在开发者认为脚本可能会有错误，并且需要看到错误跟踪信息以进行调试时，可以使用 text/plain 的内容类型，并将标准错误重定向到标准输出上，如下面的代码所示：

```
print "Content-Type: text/plain"
print
import sys
sys.stderr = sys.stdout
def witherror():
    return 1/0
print "Hello, CGI with an error!"
print "Trying to divide by 0 produces:",witherror()
print "The script does not reach this part..."
```

如果开发者的脚本只是偶尔失败，并且开发者想要查看一直到失败位置的 HTML 格式的输出信息时，还可以使用一个基于 traceback 模块（参见第 18.2 节）的更复杂的方案，如下面的代码所示：

```
import sys
sys.stderr = sys.stdout
import traceback
print "Content-Type: text/html"
print
try:
    def witherror():
        return 1/0
    print "<html><head><title>Hello, traceback</title></head>
<body>"
    print "<p>Hello, CGI with an error traceback!"
    print "<p>Trying to divide by 0 produces:",witherror()
    print "<p>The script does not reach this part..."
except ZeroDivisionError:
    print "<br><strong>ERROR detected:</strong><br><pre>"
    traceback.print_exc()
    sys.stderr = sys.__stderr__
    traceback.print_exc()
```

在导入、重定向和内容类型输出之后，这个示例在 try/except 语句的 try 子句中运行了这个脚本的最主要部分。在 except 子句中，该脚本输出了一个
标签以终止当前行，然后输出了一个<pre>标签以确保接下来的行分隔符都原封不动的。traceback 模块的 print_exc 函数输出所有错误信息。最后，该脚本将还原标准错误，并再次输出错误信息。这样，错误信息还会出现在错误日志中以供将来进行研究，而不仅仅是暂时显示在客户端浏览器上：在这个特殊示例中，这样做并不是非常有用，因为错误是可重复的，但是这样做对于跟踪真实世界中的错误而言是必需的。

cgitb 模块

在 CGI 脚本中提供好的错误报告的最简单方法就是使用 cgitb 模块，尽管这种方法并不像前一节刚介绍的方案那样具有非常好的灵活性。cgitb 模块提供了两个函数。

表 21-2

handle	handle(*exception*=None) 将一个异常的跟踪信息报告给浏览器。exception 是一个由 3 个项目组成的元组（type, value, k），就像调用 sys.exc_info( )的结果一样，参见第 8.3 节中介绍的 exc_info 函数。在 exception 为 None 时，在默认情况下，handle 将调用 exc_info 函数以获得有关要显示的异常的信息。

enable	enable(*display*=True, *logdir*=None, *context*=5)
	通过 sys.excepthook 安装一个异常钩子函数以诊断传播的异常。如果 display 为 True，这个钩子函数将在浏览器上显示这个异常的跟踪信息。如果 logdir 不为 None，这个钩子函数将把异常跟踪信息记录到 logdir 目录下的一个日志文件中。在跟踪信息中，这个钩子函数将显示源代码的 context 行。
	实际上，开发者可以使用下面的代码作为所有 CGI 脚本的开始：
	`import cgitb`
	`cgitb.enable()`
	同时，要确定尽最小的努力将好的错误报告发送到浏览器。在开发者不想自己的网页用户在他们的浏览器上看到来自开发者的脚本的 Python 跟踪信息时，可以调用 cgitb.enable(False,'/my/log/dir')并得到错误报告和跟踪信息，不同的是，跟踪信息在/my/log/dir 目录下的日志文件中。

安装 PythonCGI 脚本

CGI 脚本的安装取决于 Web 浏览器和主机的平台。从这个方面而言，Python 编码的脚本与其他语言编码的脚本并没有什么区别。当然，开发者必须确保已经安装并可以访问该 Python 解释器和标准库。在类 UNIX 平台上，必须为该脚本设置 x 权限位，并使用所谓的 shebang 行作为该脚本的第一行，例如：

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

这行代码取决于开发者的平台和 Python 的安装细节。如果在 UNIX 和 Windows 平台之间复制或共享文件，请确信 shebang 行不是以回车符（\r）结束的，这样可能会使得解析该 shebang 行的 shell 和 Web 服务器弄混淆，无法找出开发者的脚本使用的是哪个解释器。

Microsoft Web 服务器上的 Python CGI 脚本

如果开发者的 Web 服务器是 Microsoft IIS 或 Microsoft PWS（个人 Web 服务器），通过注册表路径 HKLM\System\CurrentControlSet\Services\W3Svc\Parameters\Script_Map 中的条目将文件扩展名指定为 CGI 脚本。这个路径中的每个值的名称都是一个文件扩展名，比如，.pyg（值的名称以一个句点符号开始）。而值是解释器命令（例如，C:\Python24\Python.exe -u %s %s）。出于这个目的，开发者还可以使用像.cgi 或.py 这样的文件扩展名，但是本书建议使用一个唯一的名称，也就是.pyg。将 Python 指定为所有名为.cgi 的脚本的解释器可能会干扰使用其他解释器操作 CGI 的功能。将所有使用.py

扩展名的模块解释为 CGI 脚本要比专门指定一个像.pyg 这样的唯一扩展名更容易出现问题，而且还有可能干扰 Python 编码的 CGI 脚本从相同的目录导入模块的功能。

使用 IIS5 及其以后版本，开发者可以使用“管理”工具→“计算机管理”程序以将一个文件扩展名与一个解释器命令行相关联。通过“服务和应用程序”→“Internet 信息服务”来执行这个操作。可以在[IISAdmin]（为全部网站）或者一个特定网站上单击鼠标右键，并选择“属性”→“配置”→“添加映射”→“添加”。在“扩展名”字段输入扩展名，比如，.pyg，并在“可执行程序”字段中输入解释器命令行，比如，C:\Python22\Python.Exe -u %s %s。

Apache 上的 Python CGI 脚本

流行的免费 Web 服务器 Apache 是通过文本文件（默认为 httpd.conf）中的指令来配置的。在配置文件中包含 ScriptAlias 条目时，比如：

```
ScriptAlias /cgi-bin/ /usr/local/apache/cgi-bin/
```

这个别名目录中的任何可执行脚本都可以作为 CGI 脚本运行。开发者可以通过对某个特定目录使用 Apache 指令以使能这个目录中的 CGI 执行：

```
Options +ExecCGI
```

在这种情况下，要想让一个带有特定扩展名的脚本作为 CGI 脚本运行，可能还需要添加一个全局 AddHandler 指令，比如：

```
AddHandler cgi-script pyg
```

这条指令可以使能扩展名为.pyg 的脚本作为 CGI 脚本运行。Apache 可以通过一个脚本开始位置的 shebang 行来确定为该脚本使用的解释器。在一个目录中使能 CGI 脚本的另一个方法（如果设置了全局指令 AllowOverride Options）就是在这个目录中的一个名为.htaccess 的文件中使用 Options +ExecCGI。

Xitami 上的 Python CGI 脚本

免费的、轻量级的和简单的 Web 服务器 Xitami（http://www.xitami.org）可以简单地安装 CGI 脚本。在 URL 的任何组件名为 cgi-bin 时，Xitami 将把该 URL 作为一个执行 CGI 的请求。Xitami 可以通过脚本开始位置的 shebang 行确定要为该脚本使用的解释器，即使是在 Windows 平台上。

21.2 Cookie

HTTP 本质上是一个无状态协议，这意味着 HTTP 不保留事务之间的会话状态。HTTP 1.1 标准中指定的 Cookie 可以帮助 Web 客户机和服务器相互合作，从 HTTP 事务组成的序列中建立一个有状态的会话。

每次服务器发送一个对客户请求的响应消息时，服务器可能会通过发送一个或多个 Set-Cookie 首部初始化或者继续一个会话，Set-Cookie 首部的内容都是被称为 cookies 的小数据项目。在客户机向服务器发送另一个请求时，会通过发送 Cookie 首部继续一个会话，这个 Cookie 首部中包含以前从这个服务器或者相同域内的其他服务器收到的 cookie。每个 cookie 都是一个字符串对，包括该 cookie 的名称和值，加上一些可选属性。max-age 属性是该 cookie 将被保持的最大秒数。客户机将在最大老化时间之后丢弃保持的 cookie。如果没有提供 max-age，则客户机将在该用户的交互式会话结束后丢弃该 cookie。

Cookie 不能从本质上提供隐私保护和身份认证。Cookie 将在 Internet 上透明传输，因此容易受到监听软件的攻击。一个怀有恶意的客户端可能会返回不同于以前收到的 cookie 的 cookie。要想使用 cookie 实现身份认证或标识，或者保护敏感的信息，服务器必须对发送到客户端的 cookie 进行加密和编码，并对从客户端收到的 cookie 进行解码、解密和验证。

对大量数据应用加密、编码、解码、解密和验证操作可能都是非常慢的。解密和验证要求服务器保存一部分服务器端的状态。在网络上来回发送大量的数据也是很慢的。因此，服务器必须在本地文件或数据库中持久化大量的状态数据。在大多数情况下，服务器只能将 cookie 用作小型的、加密的和可验证的键来确认用户或会话的标识，而使用 DBM 文件或关系数据库（参见第 11 章）来保存会话状态。HTTP 对 cookie 的大小设置了 2KB 的限制，但是，本书建议开发者使用甚至更小的 cookie。

Cookie 模块

Cookie 模块提供了几个类，大多数用于向后兼容。CGI 脚本通常使用来自于 Cookie 模块中的以下类，如表 21-3 所示。

表 21-3

Morsel	脚本并不直接实例化 Morsel 类。但是，cookie 类的实例可以保存 Morsel 的实例。Morsel 类的实例 m 表现了一个单独的 cookie 元素：键字符串、值字符串和可选属性。m 是一个映射。m 中唯一有效的键是 cookie 属性名称：'comment'、'domain'、'expires'、'max-age'、'path'、'secure' 和 'version'。m 中的键是不区分大小写的。m 中的值都是字符串，每个字符串保存了对应的 cookie 属性的值。

SimpleCookie	`class SimpleCookie(input=None)`
	SimpleCookie 实例 c 是一个映射。c 的键都是字符串。c 的值都是包装字符串的 Mosel 实例。c[k]=v 可以隐式扩展为：
	`c[k]=Morsel(); c[k].set(k,str(v),str(v))`
	如果 input 不为 None，实例化 c 时将隐式调用 c.load(input)。

续表

SmartCookie	`class SmartCookie(input=None)`
	SmartCookie 实例 c 是一个映射。c 的键都是字符串。c 的值都是 Morsel 实例，用来包装使用 pickle 序列化任意值。c[k]=v 具有以下语义：
	`c[k]=Morsel(); c[k].set(k,str(v),pickle.dumps(v))`
	（参见第 11.1 节中介绍的 pickle 模块）。因为开发者不能控制在使用 pickle.loads 反序列化期间执行的代码，SmartCookie 没有提供任何安全性。除非该脚本只在信任的内部互联网上使用，否则要避免使用 SmartCookie；而是使用 SimpleCookie。开发者可以使用任何密码方案以构建 SimpleCookie 实例中的 Morsel 实例值包装的字符串。第 22.1 节中介绍的模块可以很容易地将任意字节字符串编码为文本字符串，包括任意加密数据。
	SmartCookie 要比 SimpleCookie 加上加密、编码和解码使用更便利。但是，便利和安全性通常都是冲突的。选择在于开发者自己。不要苦于这样一个误解，认为系统是安全的，因为，“毕竟，没有人知道我在做什么”——引用一个著名的安全设计原则，“没有隐秘的安全性”。好的加密只是强安全性的必要条件（而不是充分条件）。要想在 Python 中使用加密，参见 http://www.amk.ca/python/code/crypto 上提供的 Python Cryptography Toolkit。

Cookie 方法

SimpleCookie 或 SmartCookie 的实例 c 提供了以下方法，如表 21-4 所示。

表 21-4

js_output	`c.js_output(attrs=None)` 返回一个字符串 s，这个字符串是一个 JavaScript 小程序，可以将 document.cookie 设置为 c 中保存的 cookie。开发者还可以将 s 嵌入到一个 HTML 响应中，如果客户浏览器支持 JavaScript，这样做可以模仿 cookie，而不必发送一个 HTTP Set-Cookie 首部。如果 attrs 不为 None，s 的 JavaScript 只能设置名称在 attrs 中的 cookie 属性。
load	`c.load(data)` 在 data 是一个字符串时，load 将解析该字符串并将每个解析的 cookie 添加到 c 中。在 data 是一个映射时，load 将为 data 中的每个项目添加一个新 Morsel 实例到 c 中。通常，data 是一个用来恢复客户端发送的 cookie 的字符串 os.environ.get('HTTP_COOKIE','')。
output	`c.output(attrs=None,header='Set-Cookie',sep='\n')` 返回一个被格式化为 HTTP 首部的字符串 s。开发者可以在响应消息的 HTTP 首部中 print c.output( )以将 c 中保存的 cookie 发送到客户端。每个首部的名称都是字符串 header，并且这些首部都是使用字符串 sep 分隔的。如果 attrs 不为 None，s 的首部只包含名称在 attrs 中的 cookie 属性。

Morsel 属性和方法

Morsel 类的实例 m 提供了 3 个读/写属性：

coded_value

cookie 的值，被编码为字符串；m 的输出方法使用 m.coded_value。

key

cookie 的名称。

value

cookie 的值，这个值是一个任意 Python 对象。

实例 m 还提供了以下方法，如表 21-5 所示。

表 21-5

js_output	*m*.js_output(*attrs*=None) 返回一个字符串 s，该字符串是一个 JavaScript 小程序，可以将 document.cookie 设置为 m 中保存的 cookie。还可以参见 cookie 实例的 js_output 方法。
output	*m*.output(*attrs*=None,*header*='Set-Cookie') 返回一个被格式化为 HTTP 首部的字符串 s，该首部设置了 m 中保存的 cookie。还可以参见 cookie 实例的输出方法。
OutputString	*m*.OutputString(*attrs*=['path','comment','domain','max-age','secure','version','expires']) 返回一个表示 m 中保存的 cookie 的字符串 s，不进行装饰。attrs 可以是适合于作为 in 操作符的右边操作数的任意容器，比如，列表、字典或集合；s 只包含名称在 attrs 中的属性。
set	*m*.set(*key*,*value*,*coded_value*) 设置 m 的属性。key 和 coded_value 必须是字符串。

使用 Cookie 模块

Cookie 模块支持客户端和服务器端脚本中的 cookie 处理。典型的用法就是在服务器端，通常在一个 CGI 脚本中（在开发者没有其他方法来管理会话状态，也没有其他方法直接处理 cookie 时）。下面的示例显示了一个使用 cookie 的简单 CGI 脚本：

```
import Cookie, time, os, sys, traceback

sys.stderr = sys.stdout

try:
```

```
# 首先，该脚本将生成 HTTP 首部
c = Cookie.SimpleCookie()
c["lastvisit"]=str(time.time())
print c.output()
print "Content-Type: text/html"
print
# 然后，脚本将生成响应消息的正文
print "<html><head><title>Hello, visitor!</title></head><body>"
# 响应消息的其余部分，脚本将得到并解码该 Cookie
c = Cookie.SimpleCookie(os.environ.get("HTTP_COOKIE"))
when = c.get("lastvisit")
if when is None:
    print "<p>Welcome to this site on your first visit!</p>"
    print "<p>Please click the 'Refresh' button to proceed</p>"
else:
    try: lastvisit = float(when.value)
    except:
        print "<p>Sorry, cannot decode cookie (%s)</p>"%when.value
        print "</br><pre>"
        traceback.print_exc()
    else:
        formwhen = time.asctime(time.localtime(lastvisit))
        print "<p>Welcome back to this site!</p>"
        print "<p>You last visited on %s</p>"%formwhen
print "</body></html>"
except:
print "Content-Type: text/html"
print
print "</br><pre>"
traceback.print_exc()
```

每当客户机访问这个脚本时，该脚本将设置一个 cookie 以编码当前时间。在连续的访问中，如果客户浏览器支持 cookie，该脚本将适当的问候访问者。参见第 12.1 节中介绍的 time 模块。这个示例没有使用加密或服务器端持久化，因为会话状态很小，并且不是要保密的。

21.3 其他服务器端方案

每当客户机请求一个 CGI 脚本时，该脚本将作为一个新进程运行。进程启动时间、解释器初始化、连接到数据库和脚本初始化共同构成了可测量的开销。在快速、现代化的服务器平台上，适度的加载开销是可以容忍的。在一个繁忙的服务器上，CGI 可能不会很好地按比例提高。Web 服务器支持许多服务器专用方法以减少开销，在可以服务几个点击的进程中运行脚本，而不是每个点击运行一个新 CGI 进程。

Microsoft 的 ASP（Active Server Pages）是一个可以平衡底层库、ISAPI 和 Microsoft 的

COM 计数的服务器扩展。大多数 ASP 网页都是使用 VBScript 语言编码的，但是 ASP 是语言独立的。正如爬虫连接建议的，Python 和 ASP 可以很好地结合在一起，只要 Python 安装了平台专用 win32all 扩展，特别是 ActiveScripting。许多其他服务器扩展都是跨平台的，而不依赖于特定的操作系统。

流行的应用程序服务器 Zope（http://www.zope.org）是一个 Python 应用程序。如果开发者需要高级管理功能，Zope（和建立在 Zope 之上的更高层内容管理系统 Pone，参见 http://phone.org）应该是开发者要考虑的解决方案之一。Zope 和 Pone 都是很大的、功能强大的系统，需要好几本书才能讲清楚这些应用程序本身。本书没有进一步介绍 Zope 和 Plone。

FastCGI

FastCGI 可以帮助开发者使用各种语言编写类似于 CGI 脚本的脚本，使用每个进程处理多个点击，可以在一个进程中顺序处理或者在不同的线程中同时处理。参见 http://www.fastcgi.com 了解 FastCGI 的概述和详细信息，还包括有关在各种类型的服务器上对 FastCGI 的支持的链接，以及 Python 对 FastCGI 的支持。FastCGI 的一个最新型的变体就是 SCGI（参见 http://www.mems-exchange.org/software/scgi/）。

WSGI

Python Web 服务器网关接口（Web Server Gateway Interface，WSGI）是正在出现的标准“中间件”解决方案，是从高层 Python Web 开发框架到底层 Web 服务器的接口，相关的文档请参见 http://www.python.org/peps/pep-0333.html。尽管 WSGI 的主要目的并不是让开发者的应用程序可以直接使用 WSGI（而是，使用几种更高抽象的框架中的任意一个来编写程序，这些框架可以按顺序使用 WSGI 与 Web 服务器进行对话），但是，也不是不可能直接使用 WSGI 的。为了简化这个任务，http://www.owlfish.com/software/wsgiutils/ 和 http://pythonpaste.org/上提供了开发者可能想要深入了解的几个模块。Mike Orr 在 Linux Gazette 上发表了一篇非常精彩的文章（http://linuxgazette.net/115/orr.html），文章介绍了 WSGI 的状态。要想了解更多有关如何最好地将 WSGI 与快速、轻量级的 lighttpd 服务器（http://www.lighttpd.org/）集成在一起的有趣的信息和讨论，可以参见 http://www.cleverdevil.org/computing/24/python-fastcgi-wsgi-and-lighttpd 上的一些博客文章和注解。

WSGI 的参考实现被称为 wsgiref，可能会被包含在 Python 2.5 的标准库中（在编写本书的时候，这一点还没有完全确认）。在任何情况下，可以参见 http://svn.eby-sarna.com/wsgiref/ 上的参考实现，并且必须使用 Python 2.3 以上的任何版本。

mod_python

Apache 的体系结构是高度模块化的。除了 CGI 和 FastCGI 可以提供的功能之外，mod_python（http://www.modpython.org）还提供了完整的 Python/Apache 集成，可以帮助 Python 访问所有需要的 Apache 内部模块，还包括编写身份认证脚本的功能。

自定义的纯 Python 服务器

在第 20.2 节中，可以看到标准 Python 库还包含实现了 Web 服务器的模块。开发者只需要进行很少的工作就可以创建 BaseHTTPServer 的子类，并实现专用目的 Web 服务器。这样的专用目的服务器在容量较低的应用程序中是非常有用的，但是无法很好地提高性能以处理中等到较高的服务器负载。

asyncore 和 asynchat 模块（参见第 20.3 节）展现了非常不同的性能特性。基于 asynchat 的应用程序的事件驱动体系结构提供了较高的可伸缩性和性能，要好于使用底层语言和传统体系结构（多进程或多线程）的应用程序。

Twisted 框架（参见第 20.3 节）具有甚至可以超过 asyncore 的性能优势，并且提供了丰富得多的功能。使用 Twisted，可以在高度抽象层的基础上编写一个 Web 网站，并仍将获得极好的可伸缩性和性能。特别是，Twisted Web2（http://twistedmatrix.com/projects/web2/）提供了一个高度抽象的 Web 编程框架，这个框架还可以用于不同的 Web 服务器（也就是，不是基于 Twisted 的服务器）；反之亦然，开发者还可以选择使用 Twisted 作为 Web 服务器，并使用一个不同的高度抽象的结构，因为使用 WSGI 作为中间件，因此可以让底层服务器层和高层应用程序层之间的接口相互透明。Nevow（http://divmod.org/trac/wiki/DivmodNevow）是另外一个强大的 Web 框架，为 Twisted 进行了优化，但是也可以与 WSGI 一起使用，除了一些非常著名的功能之外，Nevow 还提供了与“AJAX”风格的客户端 JavaScript 的非常完美和透明的集成功能。

其他的高级抽象框架

有这样一个说法“Python 是一种 Web 应用程序框架比关键字更多的语言”；Python 语言的强大功能和简单性结合在一起，吸引了许多开发人员编写他们自己的、唯一的框架。幸运的是，在这期间，WSGI 提供了某些互操作性。对一些这样的框架进行研究是非常值得的，也许其中的一个就能完美地匹配开发者自己的需要和期望（这样做的确要比开发另一个这样的框架更好地节省时间！）。开发者可以在 http://www.python.org/pycon/papers/framework/web.html 和 Wiki 网页 http://wiki.python.org/moin/WebProgramming 上找到对许多许多框架（包括许多不再活跃的框架开发）的介绍和讨论，通过网页上的 URL 可以找到更多框架，并下载这些框架。

从本书的观点来看，在所有当前活跃的开发和已经足够丰富和稳定的关键任务使用中，最有前途的框架就是 CherryPy（http://www.cherrypy.org/）、Django（http://www.djangoproject.com/）、Pylons（http://pylonshq.com/）和 TurboGears（http://www.turbogears.org/）。这些框架采用了非常不同的解决方案和哲学：有些框架集成甚至强调模板，或者数据库访问，而其他框架关注于网络部件，并可以帮助开发者通过单独的模块访问数据库或执行模板；有些框架要求（并利用）对 HTTP 和/或 SQL 相关知识有很好的理

解，而其他框架可以帮助开发者屏蔽这些层。通过给出这些区别，每个框架都可以证实对不同的程序员和工作组进行了优化。在本章的其余部分，本书挑选了 3 个框架（其中的两个是成熟的，但仍然在活跃开发中，而另一个是“出售中的新山羊”，并且，从个人观点看，这个框架是特别有前途和有趣的）。

Webware

用于 Python 的 Webware（http://www.webwareforpython.org/）是 Python 服务器端 Web 脚本的软件组件的模块化集合。开发者可以根据不同的编程模型编写 Python 脚本，比如，包含增值包装器、servlet 或 Python 服务器网页（PSP）的 CGI 脚本，并在 Webware 上运行所有这些脚本。Webware 可以使用很多方法按顺序与 Web 服务器接口，包括 CGI、FastCGI、mod_python、专门的 Apache 模块 mod_webkit 和用于 WSGI 接口的 Python Paste（http://pythonpaste.org/）。Webware 为开发者提供了许多体系结构、编码和部署服务器端 Python Web 脚本上的灵活性。

在 Webware 为生成网页提供的许多方法之中，一个经常让人感兴趣的方法就是模板（也就是，在接近成形的 HTML 脚本中自动插入 Python 计算的值和某些控制逻辑）。Webware 通过 PSP，还有 Cheetah 包（参见第 23.5 节）提供的在逻辑和表达部分之间更强大和更快速地分离来支持模板功能。

Quixote

Quixote（http://www.mems-exchange.org/software/quixote/）是可以通过通常使用的各种方法与 Web 服务器接口 Python Web 应用程序的另一个框架，这些方法包括 CGI、FastCGI、WSGI 和 mod_python。Quixote 定义了一个新语言，也就是 Python 模板语言（Python Template Language，PTL），还定义了一个导入钩子，可以让 Python 应用程序直接导入 PTL 编码的模块。

Quixote 的 PTL 基于与 Python 几乎是相同的，但是还提供了一些非常方便在 Web 应用程序中使用的额外功能。例如，PTL 关键字 template 定义了一些返回字符串结果的函数，这些函数将自动调用以响应 Web 请求；在这样的函数中，所有的表达式语句都被看作是该函数的返回值的附加字符串。例如，PTL 代码：

```
template hw():
    'hello'
    'world'
```

大致上与下面的 Python 代码相同：

```
def hw():
    _result = []
    _result.append('hello')
    _result.append('world')
    return ''.join(_result)
```

web.py

根据本书作者自己的技能、需求和口味，当前作者最喜欢的 Web 框架是 web.py（参见 http://webpy.org）。这是一个小的（web.py 本身当前是一个少于 2500 行的单个 Python 模块，包括注释）、简单的、可以与其他开源组件（比如，用于模板的 Cheetah，以及作为数据库的 PostgreSQL 或 MySQL）平稳工作的 Web 框架，并支持 WSGI 而不要求任何这样的组件（开发者的脚本还可以与 CGI、FastCGI 一起运行，或者作为一个单独的 Web 服务器运行，这对于开发期间的简单测试而言是特别有用的）。规范的“Hello world”示例是：

```
import web
urls = '/', 'greet'
class greet(object):
    def GET(self):
        print 'Hello, web.py world!'
if __name__ == '__main__':
    web.run(urls)
```

urls 列表将正则表达式（匹配 HTTP 路径）映射为类名，而这些类实现了 GET 和 POST 方法（可能还有其他方法）以服务 HTTP 请求。访问 http://webpy.org 以了解更多信息并下载 web.py。

第 22 章

MIME 和网络编码方式

网络上传播的都是字节或文本流。但是，开发者想要通过网络发送的数据通常还有更多的数据结构。“多用途 Internet 邮件扩展”（Multipurpose Internet Mail Extensions，MIME）和其他编码标准可以通过指定如何将结构化数据表示为字节或文本来弥补这个缺憾。Python 通过许多库模块，比如，base64、quopri 和 uu（参见第 22.1 节），和 email 包中的模块（参见第 22.2 节）来支持这些编码方式。

22.1 将二进制数据编码为文本

几种类型的媒体（例如，电子邮件）都只包含文本。在开发者想要通过这样的媒体传送任意二进制数据时，需要将这些数据编码为文本字符串。Python 标准库提供了一些模块以支持被称为 Base 64、Quoted Printable 和 UU 的标准编码方式。

base64 模块

base64 模块支持 RFC 1521 中定义的编码方式 Base 64。Base 64 编码方式是一种可以将任意二进制数据表示为文本的压缩方法，不可能产生任何可以人工识别的结果。base 64 模块提供了如表 22-1 所示的 4 个函数。

表 22-1

decode	`decode(`*`infile`*`,`*`outfile`*`)` 通过调用 infile.readline 读取类文本文件对象 infile，直到文件结束（也就是，直到调用 infile.readline 返回一个空白字符串），解码读取的 Base 64 编码的文本，并将解码的数据写入到类二进制文件对象 outfile 中。
decodestring	`decodestring(`*`s`*`)` 解码文本字符串 s，s 包含一个或多个完整的使用 Base 64 编码的文本行，并返回由相应的解码数据组成的字节字符串。

续表

encode	encode(*infile*,*outfile*) 通过调用 infile.read 读取类二进制文件对象 infile（每次 57 个字节，这个字节数是 Base 64 编码的每个输出行中被编码为 76 个字符的数据量），直到文件结束（也就是，直到调用 infile.read 返回一个空白字符串）。该函数将以 Base 64 编码方式对读取的数据进行编码，并按照一次一行的方式将编码后的文本写入到类文本文件对象 outfile 中，并在生成的每行文本后面添加\n，包括最后一行文本。
encodestring	encodestring(*s*) 对包含任意字节的二进制字符串 s 进行编码，并返回一个文本字符串，该字符串由一个或多个完整的基于 Base64 编码的数据行组成，数据行之间通过新行字符（\n）连接。encodestring 总是返回一个以\n 结束的文本字符串。

quopri 模块

quopri 模块支持 RFC 1521 中定义的编码方式 Quoted Printable（QP）。QP 可以将任何二进制数据表示为文本，但是该模块主要用来处理包含适当数量的高比特字符集（也就是，ASCII 范围之外的字符）字符的文本。对于这样的数据，QP 将生成压缩和可以人工识别的结果。quopri 模块提供了如表 22-2 所示的 4 个函数。

表 22-2

decode	decode(*infile*,*outfile*,*header*=False) 通过调用 infile.readline 读取类文件对象 infile，直到文件结束（也就是，直到调用 infile.readline 返回一个空白字符串），对读取的 QP 编码的 ASCII 文本进行解码，并将解码的数据写入到类文件对象 outfile 中。在 header 为 True 时，decode 还将把_（下划线）解码为空格。
decodestring	decodestring(*s*,*header*=False) 对包含 QP 编码的 ASCII 文本的字符串进行解码，并返回包含解码的数据的字节字符串。在 header 为 True 时，decodestring 还将把_（下划线）解码为空格。
encode	encode(*infile*,*outfile*,*quotetabs*,*header*=False) 通过调用 infile.readline 读取类文件对象 infile，直到文件结束（也就是，直到调用 infile.readline 返回一个空白字符串），将读取数据编码为 QP 编码方式，并将编码的 ASCII 文本写入到类文件对象 outfile 中。在 quotetabs 为 True 时，encode 还将对空格和制表符进行编码。在 header 为 True 时，encode 将把空格编码为_（下划线）。
encodestring	encodestring(*s*,*quotetabs* =False,*header*=False) 对包含任意字节的字符串 s 进行编码，并返回一个包含 QP 编码的 ASCII 文本的字符串。在 quotetabs 为 True 时，encodestring 还将对空格和制表符进行编码。在 header 为 True 时，encodestring 将把空格编码为_（下划线）。

uu 模块

uu 模块支持传统的 UNIX 到 UNIX(UU)编码，是通过 UNIX 程序 uuencode 和 uudecode 来实现的。UU 编码的数据是以一个开始（begin）行开始的，这个行还给出了正在编码的文件的文件名和权限，编码文件是以一个结束（end）行结束的。因此，UU 编码可以将编码后的数据嵌入到另外的非结构化文本中，而 Base 64 编码方式依赖于其他指示信息的存在，而指示信息指明了编码的数据开始和结束的位置。uu 模块提供了如表 22-3 所示的两个函数。

表 22-3

decode	decode(*infile*,*outfile*=None,*mode*=None) 通过调用 infile.readline 读取类文件对象 infile，直到文件结束（也就是，直到调用 infile.readline 返回一个空白字符串），或者直到出现一个终止符行（被任何数量的空白字符包围的字符串“end”）。decode 将对读取的 UU 编码的文本进行解码，并将解码后的数据写入到类文件对象 outfile 中。在 outfile 为 None 时，decode 将创建 UU 格式的开始行中指定的文件，并使用 mode 给定的权限比特（在 mode 为 None 时，则使用开始行中指定的权限比特）。在这种情况下，如果该文件已经存在，decode 将引发一个异常。
encode	encode(*infile*,*outfile*,*name*='-',*mode*=0666) 通过调用 infile.read 读取类文件对象 infile（每次 45 个字节，这个字节数是 UU 编码的每个输出行中被编码为 60 个字符的数据量）直到文件结束（也就是，直到调用 infile.read 返回一个空白字符串）。该函数将以 UU 编码方式对读取的数据进行编码，并将编码后的文本写入到类文本文件对象 outfile 中。encode 还将在已编码的文本之前写入一个 UU 开始行，在已编码的文本之后写入一个 UU 结束行。在开始行中，encode 将把文件名指定为 name，而模式指定为 mode。

22.2 MIME 和 Email 格式处理

Python 提供了 email 包来处理 MIME 文件的解析、生成和处理，比如电子邮件、网络新闻文章等。Python 标准库还包含其他一些可以处理这些任务的模块。但是，email 包提供了一个完整和系统的方案来实现这些重要的任务。本书建议开发者使用 email 包，而不是一些部分重叠了 email 功能的老式模块。email 包与接收或发送电子邮件无关；对于这样的任务，请参见第 19.2 节“Email 协议”中介绍的 poplib 和 smtplib 模块。email 包只是用来在接收到电子邮件之后，或者在发送电子邮件之前处理该电子邮件。

email 包中的函数

email 包提供了两个工厂函数，这两个函数可以返回 email.Message.Message 类的一个实例 m。这些函数依赖于 email.Parser.Parser 类，但是这些工厂函数都是很方便和简单的。因此，本书中没有进一步介绍 Parser 模块，如表 22-4 所示。

表 22-4

message_from_string	`message_from_string(s)` 通过解析字符串 s 构建 m。
message_from_file	`message_from_file(f)` 通过解析类文件对象 f 的内容构建 m，f 必须是以可读方式打开的。

email.Message 模块

email.Message 模块提供了 Message 类。email 包的所有部分都可以编译、修改或使用 Message 类的实例。Message 的实例 m 以 MIME 邮件为模型，该邮件包含多个首部和一个净荷（数据内容）。要想创建一个初始的空白实例 m，可以不带任何参数调用 Message 类。更常见的是，可以通过 email 模块的 message_from_string 和 message_from_file 函数进行解析，或者通过其他间接方法（比如，第 22.2 节中介绍的类）来创建 m。m 的净荷可以是一个字符串、单个其他的 Message 实例，或者一个用于多部分邮件的其他 Message 实例的列表。

开发者可以对正在创建的电子邮件设置任意首部。几个 Internet RFC 指定了用于各种不同目的的首部。主要应用的 RFC 是 RFC 2822（参见 http://www.faqs.org/rfcs/rfc2822.html）。Message 类的实例 m 可以保存多个首部和一个净荷。m 是一个映射，使用首部名称作为键，使用首部值字符串作为值。为了让 m 更便于使用，被用作映射的 m 的语义不同于字典的语义。m 的键是不区分大小写的。m 将按照首部被添加的顺序来保存这些首部，keys、values 和 items 方法将按这个顺序返回首部。m 可以有多个名为 key 的首部：m[key]可以返回任意一个首部，而 del m[key]将删除所有首部。len(m)将返回首部的总数，并计算重复的部分，而不仅仅是不同的首部名称的数量。如果没有名为 key 的首部，m[key]将返回 None，并且不会引发 KeyError 错误（也就是，其行为与 m.get(key)一样），在这种情况下，del m[key]不会执行任何操作。开发者不能直接对 m 进行循环操作；而是应该对 m.keys( )进行循环操作。

Message 的实例 m 提供了如表 22-5 所示的属性和方法以处理 m 的首部和净荷。

表 22-5

add_header	*m*.add_header(*_name*,*_value*,***_params*) 与 m[_name]=_value 类似，但是还可以提供首部参数作为命名参数。对于每个命名参数 pname=pvalue，add_header 将把下划线更改为破折号，然后将下面这种形式的参数添加到首部值的后面： ; *pname*="*pvalue*" 如果 pvalue 为 None，add_header 将只添加一个参数 '; pname'。
as_string	*m*.as_string(*unixfrom*=False) 以字符串的形式返回整个邮件（邮件的净荷必须是一个字符串）。在 unixfrom 为 True 时，还包含一个通常以 'From' 开始的第一行，这一行被认为是邮件的信封首部。

续表

attach	*m*.attach(*payload*) 将 payload 添加到 m 的净荷。如果 m 的净荷曾经为 None，则 m 的净荷现在是包含单个项目的列表[payload]。如果 m 的净荷是一个列表，可以将 payload 添加到该列表的后面。否则，m.attach(payload)将引发 MultipartConversionError 错误。
epilogue	m.epilogue 属性可以为 None，或者一个字符串，该字符串将成为最后一个边界行之后的邮件的字符串形式的一部分。epilogue 是 m 的一个普通属性：开发者的程序可以在处理一个不管用什么方法构建的 m 时访问该属性，并在构建或修改 m 的时候绑定该属性。
get_all	*m*.get_all(*name*,*default*=None) 按照首部被添加到 m 的顺序返回一个包含名为 name 的所有首部的值的列表。在 m 不包含名为 name 的首部时，get_all 将返回 default。
get_boundary	*m*.get_boundary(*default*=None) 返回 m 的 Content-Type 首部的 boundary 参数的字符串值。在 m 中不包含 Content-Type 首部，或者该首部中没有 boundary 参数时，get_boundary 将返回 default。
get_charsets	*m*.get_charsets(*default*=None) 返回 m 的 Content-Type 首部中由 charset 参数的字符串值组成的列表 L。在 m 包含多部分时，L 的每个部分都有一个项目；否则，L 的长度为 1。对于没有 Content-Type、没有 charset 参数，或者一个不同于‘text’的主类型的部分，L 中对应的项目就是 default。
get_content_maintype	*m*.get_content_maintype(*default*=None) 返回 m 的主内容类型：一个从 Content-Type 首部取出的小写字符串‘maintype’。在 m 不包含 Content-Type 首部时，get_content_maintype 将返回 default。
get_content_subtype	*m*.get_content_subtype(*default*=None) 返回 m 的内容子类型：一个从 Content-Type 首部取出的小写字符串‘subtype’。在 m 不包含 Content-Type 首部时，get_content_subtype 将返回 default。
get_content_type	*m*.get_content_type(*default*=None) 返回 m 的内容类型：一个从 Content-Type 首部取出的小写字符串‘maintype/subtype’。在 m 不包含 Content-Type 首部时，get_content_type 将返回 default。
get_filename	*m*.get_filename(*default*=None) 返回 m 的 Content-Disposition 首部的 filename 参数的字符串值。在 m 不包含 Content-Disposition，或者该首部不包含 filename 参数时，get_filename 将返回 default。

续表

get_param	*m*.get_param(*param*,*default*=None,*header*='Content-Type') 返回 m 的 header 首部的 param 参数的字符串值。对于只通过名称指定的参数，将返回空白字符串。在 m 不包含 header 首部，或者该首部没有名为 param 的参数时，get_param 将返回 default。 返回用于只通过名称指定的参数的空白字符串，
get_params	*m*.get_params(*default*=None,*header*='Content-Type') 返回 m 的 header 首部的参数，也就是一个由给出了每个参数的名称和值的字符串对组成的列表。使用空白字符串作为只通过名称指定的参数的值。在 m 不包含 header 首部时，get_params 将返回 default。
get_payload	*m*.get_payload(*i*=None,*decode*=False) 返回 m 的净荷。在 m.is_multipart( )为 False 时，i 必须为 None，并且 m.get_payload( )将返回 m 的整个净荷，一个字符串或邮件实例。如果 decode 为 True，并且 Content-Transfer-Encoding 首部的值为 'quoted-printable' 或 'base64'，m.get_payload 还将对净荷进行解码。如果 decode 为 False，或者没有 Content-Transfer-Encoding 首部或其他值时，m.get_payload 将返回未经更改的净荷。 在 m.is_multipart( )为 True 时，decode 必须为 False。在 i 为 None 时，m.get_payload( )将以列表的形式返回 m 的净荷。否则，m.get_payload( )将返回该净荷的第 i 个项目，如果 i<0 或者 i 太大，则引发 TypeError 错误。
get_unixfrom	*m*.get_unixfrom( ) 返回 m 的信封首部字符串，如果 m 不包含信封首部，则返回 None。
is_multipart	*m*.is_multipart( ) 在 m 的净荷为一个列表时，返回 True；否则，返回 False。
preamble	m.preamble 属性可以为 None 或者一个字符串，该字符串将成为第一个边界行之前的邮件的字符串形式的一部分。只有在邮件程序不支持多部分的邮件时，该程序才显示这个文本，因此开发者可以使用这个属性以警告邮件包含多部分的用户，并且需要一个不同的邮件程序来查看该文本。preamble 是 m 的一个普通属性：开发者的程序可以在处理一个不管用什么方法构建的 m 时访问该属性，并在构建或修改 m 的时候绑定该属性。
set_boundary	*m*.set_boundary(*boundary*) 将 m 的 Content-Type 首部的 boundary 参数设置为 boundary。在 m 不包含 Content-Type 首部时，将引发 HeaderParseError 错误。
set_payload	*m*.set_payload(*payload*) 将 m 的净荷设置为 payload，根据情况，payload 必须是一个字符串或者一个列表。
set_unixfrom	*m*.set_unixfrom(*unixfrom*) 设置 m 的信封首部字符串。unixfrom 是整个信封首部行，包含开头的 'From'，但是不包含结尾的 '\n'。
walk	*m*.walk( ) 返回一个迭代器，对 m 的所有部分和子部分进行迭代以深度优先方式遍历树的所有部分。

email.Generator 模块

email.Generator 模块提供了 Generator 类，开发者可以使用该类生成文本形式的邮件 m。m.as_string 和 str(m)可能就足够了，但是 Generator 类为开发者提供了更多的灵活性。开发者可以使用一个强制参数和两个可选参数实例化 Generator，如表 22-6 所示。

表 22-6

Generator	classGenerator(*outfp*,*mangle_from_*=False,*maxheaderlen*=78) outfp 是一个提供了 write 方法的文件或类文件对象。在 mangle_from_为 True 时，g 将在以‘From’开始的净荷中的任意行之前预先输入‘>’，以让该邮件的文本形式更容易解析。g 将把以分号分隔的每个首部行包装成不超过 maxheaderlen 个字符的物理行。要想使用 g，可以调用 g.flatten： *g*.flatten(*m*, *unixfrom*=False) 这样将生成 m 以作为要输出到 outfp 中的文本，就像使用 outfp.write(m.as_string(UNIXfrom))一样。

创建邮件

email 包提供了名称以‘MIME’开始的模块，每个模块都提供了一个以 Message 命名的子类。这些类可以很容易地创建各种 MIME 类型的 Message 实例。MIME 类如表 22-7 所示。

表 22-7

MIMEAudio	class MIMEAudio(*_audiodata*,*_subtype*=None,*_encoder*=None, ***_params*) _audiodata 是一个将音频数据包装到 MIME 类型‘audio/_subtype’的邮件中的字节字符串。在_subtype 为 None 时，_audiodata 必须可以使用标准 Python 模块 sndhdr 进行解析，以确定该子类；否则，MIMEAudio 将引发一个 TypeError 错误。在_encoder 为 None 时，MIMEAudio 可以将数据编码为 Base 64，这通常是最优的。否则，_encoder 必须可以带一个参数 m 进行调用，m 就是正在构造的邮件；然后，_encoder 必须调用 m.get_payload( )以获得净荷，还可以调用 m.set_payload 对净荷进行编码，并返回编码形式，然后适当地设置 m['Content-Transfer-Encoding']。MIMEAudio 将把包含命名参数的名称和值的_params 字典传递到 m.add_header 以构造 m 的 Content-Type。
MIMEBase	class MIMEBase(*_maintype*,*_subtype*,***_params*) 所有 MIME 类的基类；是 Message 的直接子类。实例化： *m* = MIMEBase(*main*,*sub*,***parms*) 等同于代码更长，也更不方便的习惯用法： *m* = Message() m.add_header('Content-Type','%s/%s'%(main,sub),**parms) *m*.add_header('Mime-Version','1.0')

续表

MIMEImage	class MIMEImage(*_imagedata*,*_subtype*=None,*_encoder*=None,***_params*) 与 MIMEAudio 类似，但是包含主类型‘image’；如有必要，可以使用标准 Python 模块 imghdr 以确定该子类。
MIMEMessage	class MIMEMessage(*msg*,*_subtype*='rfc822') 将 msg 包装为 MIME 类型‘message/_subtype’的邮件的净荷，msg 必须是 Message 的一个实例（或者一个子类）。
MIMEText	class MIMEText(*_text*,*_subtype*='plain',*_charset*='us-ascii', *_encoder*=None) 将文本字符串_text 包装为使用给定 charset 的 MIME 类型‘text/_subtype’的邮件的净荷。在_encoder 为 None 时，MIMEText 并不对文本进行编码，这通常是最优的。否则，_encoder 必须可以带一个参数 m 进行调用，m 就是正在构造的邮件；然后，_encoder 必须调用 m.get_payload( )以获得净荷，还可以调用 m.set_payload 对净荷进行编码，并返回编码形式，然后适当地设置 m['Content-Transfer-Encoding']。

email.Encoders 模块

email.Encoders 模块提供了几个将邮件 m 作为其唯一参数的函数，可以对 m 的净荷进行编码，并适当地设置 m 的首部，如表 22-8 所示。

表 22-8

encode_base64	encode_base64(*m*) 使用 Base 64 进行编码，这对于任意二进制数据是最优的。
encode_noop	encode_noop(*m*) 不对 m 的净荷和首部执行任何操作。
encode_quopri	encode_quopri(*m*) 使用 Quoted Printable 进行编码，这对于几乎全是，但是不完全是 ASCII 的文本是最优的。
encode_7or8bit	encode_7or8bit(*m*) 不对 m 的净荷执行任何操作，如果 m 的净荷的任何字节都设置了高比特，则将 Content-Transfer-Encoding 首部设置为‘8bit’；否则，将其设置为‘7bit’。

email.Utils 模块

email.Utils 模块提供了几个可以用于电子邮件处理的函数，如表 2-9 所示。

表 22-9

formataddr	formataddr(*pair*) pair 是一个字符串对（realname, email_address）。formataddr 将返回一个字符串 s，其中包含要插入到首部字段（比如 To 和 Cc）中的地址。在 name 为 False 时（例如，‘ ’），dump_address_pair 将返回 email_address。
formatdate	formatdate(*timeval*=None,*localtime*=False) timeval 是从新纪元时间开始之后的秒数。在 timeval 为 None 时，formatdate 将使用当前时间。在 localtime 为 True 时，formatdate 将使用本地时区；否则，将使用 UTC。formatdate 将返回一个包含给定时刻的字符串，该时刻是按照 RFC 2822 中指定的方法格式化的。
getaddresses	getaddresses(*L*) 解析 L 中的每个项目（L 是一个地址字符串列表，被用在像 To 和 Cc 这样的首部字段中），并返回一个由字符串对（name, email_address）组成的列表。在 getaddresses 不能将 L 中的某个项目解析为一个地址时，getaddresses 将使用（None, None）作为其返回的列表中的对应项目。
mktime_tz	mktime_tz(*t*) t 是一个包含 10 个项目的元组。t 的前 9 个项目都使用了与 time 模块中使用的相同格式，参见第 12.1 节。t[-1]是一个时区值，作为从 UTC 时间开始偏移的秒数（按照 RFC 2822 中的定义，还包含来自于 time.timezone 的相反符号）。在 t[-1]为 None 时，mktime_tz 将使用本地时区。mktime_tz 返回一个浮点型值，表示按 UTC 时间从新纪元时间开始之后的秒数，对应于 t 指示的时刻。
parseaddr	parseaddr(*s*) 解析字符串 s（s 包含一个地址，就像通常在 To 和 Cc 这样的首部字段中指定地址一样），并返回一个字符串对（realname, email_address）。在 parseaddr 不能将 s 解析为一个地址时，parseaddr 将返回（",")。
parsedate	parsedate(*s*) 按照 RFC 2822 中的规则解析字符串 s，并返回一个包含 9 个项目的元组 t，就像在 time 模块中使用的那样，参见第 12.1 节（项目 t[-3]没有任何意义）。parsedate 还将尝试解析广大寄信人使用的 RFC 2822 协议中的某些错误变体。在 parsedate 不能解析 s 时，parsedate 将返回 None。
parsedate_tz	parsedate_tz(*s*) 与 parsedate 类似，但是返回一个包含 10 个项目的元组 t，其中 t[-1]是 s 的时区值，表示从 UTC 时间开始的偏移秒数（按照 RFC 2822 中的定义，还包含来自于 time.timezone 的相反符号），就像在 mktime_tz 接受的参数中那样。项目[-4:-1]没有任何意义。在 s 不包含时区时，t[-1]为 None。
quote	quote(*s*) 返回字符串 s 的一个副本，其中每个双引号（"）都将变成‘\" ’，并且每个已有的反斜线符号都是重复的。
unquote	unquote(*s*) 返回字符串 s 的一个副本，包围字符串 s 的开头和末尾的双引号符号(")和尖括号（<>）都将被删除。

email 包的示例使用

email 包可以用来读取和撰写电子邮件和类电子邮件的邮件（另一方面，email 包绝对与接收和发送这样的邮件没有任何关系：这些任务属于完全不同和分开的模块，参见第 19 章和第 20 章）。下面是一个如何使用 email 包读取可能的多部分邮件，并将每个部分解包到一个给定目录中的文件的示例：

```
import os, email
def unpack_mail(mail_file, dest_dir):
    ''' Given file object mail_file, open for reading, and dest_dir, a
        string that is a path to an existing, writable directory, unpack
        each part of the mail message from mail_file to a file within
        dest_dir.
    '''
    try: msg = email.message_from_file(mail_file)
    finally: mail_file.close()
    for part_number, part in enumerate(msg.walk()):
        if part.get_content_maintype() == "multipart":
            continue
        dest = part.get_filename()
        if dest is None: dest = part.get_param("name")
        if dest is None: dest = "part-%i" % partCounter
        # 在实际使用中，要确信 dest 是开发者自己的 OS 上的一个合理的文件名；
        # 否则，将调整这个文件名，直到该文件名合理！
        f = open(os.path.join(dest_dir, dest), "wb")
        try: f.write(part.get_payload(decode=True))
        finally: f.close()
```

下面是另一个执行大致相反任务的示例，该示例将直接把给定源目录下的所有文件包装到一个适合于邮寄的文件中：

```
def pack_mail(source_dir, **headers):
    ''' Given source_dir, a string that is a path to an existing, readable
        directory, and arbitrary header name/value pairs passed in as
        namedarguments, packs all the files directly under source_dir
        (assumed to be plain text files) into a mail message returned
        as a string.
    '''
    msg = email.Message.Message()
    for name, value in headers.iteritems():
        msg[name] = value
    msg['Content-type'] = 'multipart/mixed'
    filenames = os.walk(source_dir).next() [-1]
    for filename in filenames:
        m = email.Message.Message()
        m.add_header("Content-type", 'text/plain', name=filename)
        f = open(os.path.join(source_dir, filename), "r")
        m.set_payload(f.read())
```

```
        f.close()
        msg.attach(m)
    return msg.as_string()
```

rfc822 和 mimetools 模块的 Message 类

处理类电子邮件的邮件的最好方法就是使用 email 包。但是，第 19 章和第 21 章中介绍的其他一些模块使用了 rfc822.Message 类或其子类（mimetools.Message）的实例。本节介绍了这些类的功能的子集，开发者需要通过这些功能有效地使用第 19 章和第 21 章中介绍的模块。

Message 类的实例 m 是一个映射，该映射使用首部的名称作为键，使用对应的首部值字符串作为值。键和值都是字符串，而键是不区分大小写的。m 支持除 clear、copy、popitem 和 update 之外的所有映射方法。get 和 setdefault 默认为 ''，而不是 None。实例 m 还提供了便利的方法（例如，结合获得一个首部的值和将该值解析为一个日期或一个地址）。出于这个目的，本书建议开发者使用 email.Utils 模块中的函数（参见第 22.2 节）并将 m 只作为一个映射使用。

在 m 是 mimetools.Message 的一个实例时，m 提供了其他一些方法，如表 22-10 所示。

表 22-10

getmaintype	*m*.getmaintype() 返回 m 的主内容类型，是从被转换为小写字符的 Content-Type 首部提取出来的。在 m 不包含 Content-Type 首部时，getmaintype 将返回 'text'。
getparam	*m*.getparam(*param*) 返回 m 的 Content-Type 首部中名为 param 的参数的字符串值。
getsubtype	*m*.getsubtype() 返回 m 的内容子类型，是从被转换为小写字符的 Content-Type 首部提取出来的。在 m 不包含 Content-Type 时，getsubtype 将返回 'plain'。
gettype	*m*.gettype() 返回 m 的内容类型，是从被转换为小写字符的 Content-Type 首部提取出来的。在 m 不包含 Content-Type 时，gettype 将返回 'text/plain'。

第 23 章

结构化文本：HTML

Web 上使用的大多数文档都使用了 HTML，也就是“超文本标记语言”（HyperText Markup Language，HTML）。标记就是在文本文档中插入被称为标签（tag）的特殊标记，为文本提供结构。理论上讲，HTML 是一个被称为“标准通用标记语言”（Standard General Markup Language，SGML）的大型通用标准的应用程序。实际上，许多 Web 文档都以很随便或错误的方式使用 HTML。浏览器已经经过多年的许多实际尝试。多年来，浏览器已经经历了许多发展以尝试和补偿这些缺点，但是，即使这样，有时候还是会发生浏览器以某种奇怪的方式显示错误的 Web 网页（不要责怪浏览器：在所有这些责怪中，10 次就有 9 次是因为 Web 网页作者自己的原因！）。

HTML 只适合于在屏幕上显示文档。从文档的显示结果反向操作以完整和精确地提取文档中的信息通常是不可能的。要想加强一些事情，HTML 已经发展成一个被称为 XHTML 的更严格的标准。XHTML 非常类似于传统 HTML，但是 XHTML 是根据 XML 和比 HTML 更精确的语言定义的。开发者可以使用第 24 章中介绍的工具来处理 XHTML。

尽管有这些困难，但是至少从 HTML 文档中提取某些有用的信息通常是可能的（这是一个通常被称为屏幕快照的任务，或者就称为快照）。Python 提供了 sgmllib、htmllib 和 HTMLParser 模块以实现解析 HTML 文档的任务，不管这样的解析是出于显示文档的目的，还是出于部分尝试提取（快照）信息的目的。在处理断开的网页时，第三方模块 BeautifulSoup 为开发者提供了最好的，也是最后的希望。生成 HTML 和在 HTML 中嵌入 Python 也是一个经常出现的任务。没有标准 Python 库模块直接支持 HTML 生成或嵌入，但是开发者可以使用普通 Python 字符串进行操作，而且第三方模块也可以对此有所帮助。

23.1 sgmllib 模块

sgmllib 模块的名称会令人误解：sgmllib 只能解析 SGML 的一个很小的子集，但是这仍

然是一个从 HTML 文件获取信息的好方法。sgmllib 只提供了一个类 SGMLParser，开发者可以创建这个类的子类，并覆盖其中的方法。SGMLParser 的子类 X 的实例 s 最常使用的方法如表 23-1 所示。

表 23-1

close	*s*.close() 告诉解析器已经没有输入数据了。在 X 覆盖 close 时，s.close 必须调用 SGMLParser.close 以确保缓冲区中的数据已经被处理了。
do_tag	*s*.do_tag(*attributes*) X 为想要处理的每个 tag 提供了一个以这个名称命名的方法，不包含对应的结束标签。在这个方法名称中，tag 必须是小写的，但是在解析的文本中可以是不区分大小写的（与 HTML 类似，SGML 标准是不区分大小写的，这一点与区分大小写的 XML 和 XHTML 是不同）。SGMLParser 的 handle_tag 方法会在适当的时候调用 do_tag。attributes 是一个由数据对（name, value）组成的列表，其中 name 是一个属性的名称，使用小写字母，而 value 是一个值，这个值经过了解析实体和字符引用的处理，并删除了围绕这个值的引号。
end_tag	*s*.end_tag() X 为每个结束标签 X 想要处理的 tag 提供了一个以这个名称命名的方法。在这个方法名中，tag 必须是小写的，但是在解析的文本中可以是不区分大小写的。X 还必须提供一个名为 start_tag 的方法；否则，end_tag 将被忽略。SGMLParser 的 handle_endtag 方法将在适当的时候调用 end_tag。
feed	*s*.feed(*data*) 向解析器传递一些要被解析的文本。解析器可以处理该文本的某些前缀，在缓冲区中保存其余的部分，直到下一次调用 s.feed 或 s.close。

handle_charref	*s*.handle_charref(*ref*)
	调用该方法以处理一个字符引用‘&#ref;’。handle_charref 的 SGMLParser 实现只能处理范围在 range(0, 256)之内的十进制数据，例如：
	`def handle_charref(self, ref):`
	`  try:`
	`  c = chr(int(ref[1:]))`
	`  except (TypeError, ValueError):`
	`  self.unknown_charref(ref)`
	`  else: self.handle_data(c)`
	开发者子集的子类 X 可能会覆盖 handle_charref 或 unknown_charref 以支持其他形式的字符引用 ‘&#...;’。

续表

handle_comment	*s*.handle_comment(*comment*) 调用该方法以处理注释。comment 是一个包含在‘<!--...-->’之内的字符串，不带定界符。SGMLParser 类中实现的 handle_comment 方法不执行任何操作。
handle_data	*s*.handle_data(*data*) 调用该方法以处理每个任意字符串 data。开发者自己创建的子类 X 通常会覆盖 handle_data 方法。SGMLParser 类中实现的 handle_data 方法不执行任何操作。
handle_endtag	*s*.handle_endtag(*tag*,*method*) 调用该方法以处理终止标签，为了实现这个功能，X 提供了名为 start_tag 和 end_tag 的方法。tag 是小写字母的标签字符串。method 是用于 end_tag 的方法。SGMLParser 类中实现的 handle_endtag 方法只需要调用 method( )即可，并且极少需要覆盖该方法。

handle_entityref	*s*.handle_entityref(*ref*)
	调用该方法以处理一个实体引用‘&ref;’。SGMLParser 类中实现的 handle_entityref 方法将在 s.entitydefs 中查找 ref。
	为了以不同的方式支持实体引用‘&...;’开发者的子类 X 可能会覆盖 handle_entityref 或 unknown_entityref 方法。SGMLParser 的 entitydefs 属性包括键‘amp’、‘apos’、‘gt’、‘lt’和‘quot’。假定子类 X 需要添加 htmlentitydefs 中定义的实体，请参见第 23.2 节。其中的一个解决方案是：
	`class X(sgmllib.SGMLParser):`
	`  entitydefs = dict(sgmllib.SGMLParser)`
	`  entitydefs.update((k, unichr(v))`
	`  for k, v inhtmlentitydefs. name2codepoint.`
	`  iteritems())`
	当然，X.handle_data 还必须准备处理 Unicode，而不只是普通字符串参数（在任何情况下，这种增强功能都是一个好主意）。

handle_starttag	*s*.handle_starttag(*tag*, *method*, *attributes*) 调用该方法以处理 X 为其提供了 start_tag 或 do_tag 方法的标签。tag 是小写字母的标签字符串。method 是 start_tag 或 do_tag 的边界方法。attributes 是一个由数据对（name, value）组成的列表，其中 name 是每个属性的名称，以小写字母表示，而 value 是值，处理该列表可以解决实体引用和字符引用，并可以删除引号。在 X 提供了 start_tag 和 do_tag 方法时，start_tag 有优先权，而 do_tag 将被忽略。SGMLParser 类中实现的 handle_starttag 方法只调用 method(attributes)，极少需要覆盖这个方法。
report_unbalanced	*s*.report_unbalanced(*tag*) 在没有开始标签就出现了终止标签时调用该方法。tag 是小写字母的标签字符串。SGMLParser 类中实现的 report_unbalanced 方法不执行任何操作。

续表

start_tag	*s*.start_*tag*(*attributes*) X 为想要处理的每个 tag（包括结束标签）提供了一个这样名称的方法。在方法名称中，tag 必须是小写字母，但是在解析的文本中，tag 是不区分大小写的。SGMLParser 的 handle_tag 方法可以在适当的时候调用 start_tag。attributes 是一个由数据对（name, value）组成的列表，其中 name 是每个属性的名称，以小写字母表示，而 value 是值，处理该列表可以解决实体引用和字符引用，并可以删除引号。
unknown_charref	*s*.unknown_charref(*ref*) 调用该方法可以处理无效或不可识别的字符引用。SGMLParser 类中实现的 unknown_charref 方法不执行任何操作。
unknown_endtag	*s*.unknown_endtag(*tag*) 调用该方法以处理 X 没有为其提供特殊方法的结束标签。SGMLParser 类中实现的 unknown_endtag 方法不执行任何操作。
unknown_entityref	*s*.unknown_entityref(*ref*) 调用该方法以处理未知的实体引用。SGMLParser 类中实现的 unknown_entityref 方法不执行任何操作。
unknown_starttag	*s*.unknown_starttag(*tag*, *attributes*) 调用该方法以处理 X 没有为其提供特殊方法的标签。tag 是以小写字母表示的标签字符串。attributes 是一个由数据对（name, value）组成的列表，其中 name 是每个属性的名称，以小写字母表示，而 value 是值，处理该列表可以解决实体引用和字符引用，并可以删除引号。SGMLParser 类中实现的 unknown_starttag 方法不执行任何操作。

使用 sgmllib 解析 HTML

下面的示例使用了 sgmllib 来执行典型的 HTML 相关任务，这个任务可以作为“网络蜘蛛”的核心代码：使用 urlllib 从 Web 抓取网页，解析该网页，然后输出其中的超链接的目标 URL。这个示例使用了 urlparse 来检查该网页的链接，并且只输出 URL 中包含显式‘http’协议的链接。

```
import sgmllib, urllib, urlparse

class LinksParser(sgmllib.SGMLParser):
    def __init__(self):
        sgmllib.SGMLParser.__init__(self)
        self.seen = set()
    def do_a(self, attributes):
        for name, value in attributes:
            if name == 'href' and value not in self.seen:
            self.seen.add(value)
            pieces = urlparse.urlparse(value)
            if pieces[0] != 'http': return
            print urlparse.urlunparse(pieces)
            return

p = LinksParser()
f = urllib.urlopen('http://www.python.org/index.html')
BUFSIZE = 8192
```

```
    while True:
        data = f.read(BUFSIZE)
        if not data: break
        p.feed(data)
    p.close()
```

LinksParser 类只需要定义 do_a 方法。超类将为所有<a>标签回调这个方法，并且该方法将循环处理属性，查找一个名为 'href' 的属性，然后处理对应的值（也就是，相应的 URL）。

23.2 htmllib 模块

htmllib 模块提供了一个名为 HTMLParser 的类，这个类是 SGMLParser 的子类，并定义了用来处理 HTML 2.0 标签的 start_tag、do_tag 和 end_tag 方法。HTMLParser 实现并覆盖了一些方法以执行对格式器对象的方法的调用，参见本节对 formatter 模块的介绍。开发者可以创建 HTMLParser 的子类并覆盖一些方法。除了 start_tag、do_tag 和 end_tag 方法，HTMLParser 的实例 h 还提供了如表 23-2 所示的属性和方法。

表 23-2

anchor_bgn	*h*.anchor_bgn(*href*,*name*,*type*) 为每个<a>标签调用该方法。href、name 和 type 都是具有相同名称的标签属性的字符串值。HTMLParser 类中实现的 anchor_bgn 方法可以在一个名为 s.anchorlist 的实例属性中维护一个由输出超链接对象组成的列表（也就是，s.anchor_bgn 方法的 href 参数）。
anchor_end	*h*.anchor_end() 为每个结束标签调用该方法。HTMLParser 类中实现的 anchor_end 方法将向格式器输出一个脚注引用，该引用是 s.anchorlist 中的一个索引。换句话讲，在默认情况下，HTMLParser 将要求格式器将一个<a>/标签对格式化为标签中的文本，文本的后面带一个指向<a>标签中的 URL 的脚注引用号。当然，如何处理这个格式化请求取决于格式器。
anchorlist	h.anchor_list 属性包含由输出超链接目标 URL 组成的列表，这些 URL 是由 h.anchor_bgn 方法构建的。
formatter	h.formatter 属性是与 h 相关的格式器对象 f，在实例化 HTMLParser(f)时，f 将被传递为 HTMLParser 的唯一参数。
handle_image	*h*.handle_image(*source*,*alt*,*ismap*='',*align*='',*width*='',*height*='') 为每个<img>标签调用该方法。每个参数都是具有相同名称的标签属性的字符串值。HTMLParser 类中实现的 handle_image 方法可以调用 h.handle_data(alt)（换句话讲，默认的实现将适当地忽略图像，并格式化替换文本）。
nofill	*h*.nofill 在通常情况下，在解析器是折叠的空白字符时，h.nofile 属性为 False。在解析器必须保留空白字符时（通常在<pre>标签中），h.nofile 属性为 True。
save_bgn	*h*.save_bgn() 将数据转移到内部缓冲区中，而不是将数据传递到格式器中，直到下次调用 h.save_end( )。h 只有一个缓冲区，因此不能嵌套 save_bgn 调用。
save_end	*h*.save_end() 返回一个包含内部缓冲区中所有数据的字符串，并将从现在开始的数据传递回格式器中。如果 save_bgn 的状态不是打开的，则引发 TypeError 错误。

formatter 模块定义了格式器和写入器类。可以通过向 formatter 类传递一个写入器实例以实例化一个格式器，然后将格式器实例传递到 htmllib 模块的 HTMLParser 类。开发者可以通过创建 formatter 类的子类并适当地覆盖其中的一些方法以定义开发者自己的格式器和写入器，但是本书没有介绍这个高级，但是极少使用的功能。有特殊输出要求的应用程序通常可以定义一个适当的写入器，创建 AbstractWriter 的子类并覆盖所有方法，然后使用 AbstractFormatter 类，而不需要使用该类的子类。formatter 模块提供了如表 23-3 所示的类。

表 23-3

AbstractFormatter	`class AbstractFormatter(`*writer*`)` 标准的格式器实现，该类适合于大多数任务。
AbstractWriter	`class AbstractWriter()` 一个写入器实现，在调用这个类时，将打印该类的每个方法名称，该类只适合于调试目的。
DumbWriter	`class DumbWriter(`*file*`=sys.stdout,`*maxcol*`=72)` 一个可以将文本输出到文本对象 file 的写入器实现，并使用自动换行以确保没有文本行长于 maxcol 个字符。
NullFormatter	`class NullFormatter(`*writer*`=None)` 一个格式器实现，其方法不执行任何操作。在 writer 为 None 时，将实例化 NullWriter。在创建 HTMLParser 的子类以分析 HTML 文档，但是不想产生任何输出时，非常适合使用这个类。
NullWriter	`class NullWriter()` 一个写入器实现，其方法都不执行任何操作。

htmlentitydefs 模块

htmlentitydefs 模块提供了 3 个属性。

codepoint2name

从 Unicode 编码点到 HTML 实体名称的映射。例如，htmlentitydefs.codepoint2name[229]的结果为‘auml’，因为 Unicode 字符 229（变元音的小写字母 a）在 HTML 中被编码为‘ä’。

entitydefs

从 HTML 实体名称到 Latin-1 字符或 HTML 字符引用的映射。例如，htmlentitydefs.entitydefs['auml']的结果为‘\xe4’，而 htmlentitydefs.entitydefs['sigma']的结果为‘σ’。

name2codepoint

从 HTML 实体名称到 Unicode 编码点的映射。例如，htmlentitydefs.name2codepoint

['auml']的结果为 228。

htmllib 模块内部使用了 htmlentitydefs 模块。

使用 htmllib 解析 HTML

下面的示例使用了 htmllib 来执行与前一个 sgmllib 示例相同的任务，使用 urllib 从 Web 抓取网页，解析该网页，然后输出其中的超链接：

```
import htmllib, formatter, urllib, urlparse

p = htmllib.HTMLParser(formatter.NullFormatter())
f = urllib.urlopen('http://www.python.org/index.html')
BUFSIZE = 8192
while True:
    data = f.read(BUFSIZE)
    if not data: break
    p.feed(data)
p.close()

seen = set()
for url in p.anchorlist:
  if url in seen: continue
  seen.add(url)
  pieces = urlparse.urlparse(url)
  if pieces[0] == 'http':
    print urlparse.urlunparse(pieces)
```

这个示例利用了 htmllib.HTMLParser 类的 anchorlist 属性，并因此不需要执行任何子类操作。htmllib.HTMLParser 创建了 anchorlist 属性以解析 HTML 网页，因此代码只需要对列表执行循环操作，并处理该列表中的项目即可，每个项目有一个对应的 URL。

23.3 HTMLParser 模块

HTMLParser 模块提供了一个类 HTMLParser，开发者可以创建子类以覆盖其中的方法。HTMLParser.HTMLParser 类似于 sgmllib.SGMLParser，但是要更简单一些，而且也可以解析 XHTML。HTMLParser 与 SGMLParser 之间的主要区别如下。

- HTMLParser 不调用名为 do_tag、start_tag 和 end_tag 方法。要想处理标签和结束标签，开发者创建的 HTMLParser 的子类必须覆盖 handle_starttag 和/或 handle_endtag 方法，并显式检查想要处理的标签。
- HTMLParser 既不跟踪，也不检查以任何方式嵌套的标签。
- 在默认情况下，HTMLParser 没有对解析字符和实体引用执行任何操作。如果需要执行对这些引用的处理，开发者创建的 HTMLParser 的子类 X 必须覆盖

handle_charref 和/或 handle_entityref 方法。

HTMLParser 的子类 X 的实例 h 通常使用的方法如表 23-4 所示。

表 23-4

close	*h*.close() 告诉解析器已经没有输入数据了。在 X 覆盖 close 方法时，h.close 还必须调用 HTMLParser.close 以确保所有缓冲的数据都已经被处理了。
feed	*h*.feed(*data*) 向解析器传递要被解析的一部分文本。解析器将处理该文本的一些前缀，并在缓冲区中保持其余部分，直到下次调用 h.feed 或 h.close。
handle_charref	*h*.handle_charref(*ref*) 调用该方法以处理一个字符引用‘&#ref;’。HTMLParser 类中实现的 handle_charref 不执行任何操作。
handle_comment	*h*.handle_comment(*comment*) 调用该方法以处理注释。comment 是一个包含在‘<!--...-->’之内的字符串，不带定界符。HTMLParser 类中实现的 handle_comment 方法不执行任何操作。
handle_data	*h*.handle_data(*data*) 调用该方法以处理每个任意字符串 data。开发者自己创建的子类 X 几乎总是会覆盖 handle_data 方法。HTMLParser 类中实现的 handle_data 方法不执行任何操作。
handle_endtag	*h*.handle_endtag(*tag*) 调用该方法以处理终止标签。tag 是小写字母的标签字符串。HTMLParser 类中实现的 handle_endtag 方法不执行任何操作。
handle_entityref	*h*.handle_entityref(*ref*) 调用该方法以处理一个实体引用‘&ref;’。HTMLParser 类中实现的 handle_entityref 方法不执行任何操作。
handle_starttag	*h*.handle_starttag(*tag*, *attributes*) 调用该方法以处理标签。tag 是小写字母的标签字符串。attributes 是一个由数据对（name, value）组成的列表：name 是每个属性的名称，以小写字母表示，而 value 是值，处理该列表可以解决实体引用和字符引用，并可以删除引号。HTMLParser 类中实现的 handle_starttag 方法不执行任何操作。

使用 HTMLParser

下面的示例使用了 HTMLParser 来执行与前一个示例相同的任务：使用 urllib 从 Web 抓取一个网页，解析该网页，然后输出其中的超链接：

```
import HTMLParser, urllib, urlparse

class LinksParser(HTMLParser.HTMLParser):
    def __init__(self):
        HTMLParser.HTMLParser.__init__(self)
        self.seen = set()
    def handle_starttag(self, tag, attributes):
        if tag != 'a': return
        for name, value in attributes:
            if name == 'href' and value not in self.seen:
                self.seen.add(value)
                pieces = urlparse.urlparse(value)
                if pieces[0] != 'http': return
                print urlparse.urlunparse(pieces)
                return

p = LinksParser()
f = urllib.urlopen('http://www.python.org/index.html')
BUFSIZE = 8192
while True:
    data = f.read(BUFSIZE)
    if not data: break
    p.feed(data)

p.close()
```

因为 HTMLParser.HTMLParser 没有对方法执行按标签的指派，LinksParser 覆盖了 handle_starttag 方法并检查 tag 是否为‘a’。

23.4 BeautifulSoup 扩展

BeautifulSoup（http://www.crummy.com/software/BeautifulSoup/）可以用来解析形式上比较差的 HTML，并使用简单的启发式方法来弥补可能的 HTML 缺陷（在这个困难的任务上，BeautifulSoup 取得了令人惊奇的成功）。BeautifulSoup 模块提供了一个类，其名称也是 BeautifulSoup，这个类可以使用一个类文件对象（读取该对象以给出要解析的 HTML 文本）或者一个字符串（要解析的文本）来实例化。该模块还提供了另外两个非常相似的类（BeautifulStoneSoup 和 ICantBelieveItsBeautifulSoup），但是这两个类适用于稍微不同的 XML 解析任务。BeautifulSoup 类的实例 b 提供了许多属性和方法以简化在已被解析的 HTML 输入数据中搜索信息的任务，并返回 Tag 和 NavigableText 类的实例，这样可以按顺序继续导航或者挖掘更多的信息。

使用 BeautifulSoup 解析 HTML

下面的示例使用了 BeautifulSoup 来执行与前一个示例相同的任务：使用 urllib 从 Web

抓取一个网页，解析该网页，然后输出其中的超链接：

```
import urllib, urlparse, BeautifulSoup

f = urllib.urlopen('http://www.python.org/index.html')
b = BeautifulSoup.BeautifulSoup(f)

seen = set()
for anchor in b.fetch('a'):
    url = anchor.get('href')
    if url is None or url in seen: continue
    seen.add(url)
    pieces = urlparse.urlparse(url)
    if pieces[0]=='http':
        print urlparse.urlunparse(pieces)
```

这个示例调用了 BeautifulSoup.BeautifulSoup 类的 fetch 方法来获得特定标签（这里是标签‘<a>’）的所有实例，然后调用 Tag 类的实例的 get 方法获得一个属性（这里是‘href’）的值，在没有这个属性时，属性值为 None。分析和生成输出超链接的目标 URL 的逻辑与前几个示例是完全相同的。

23.5 生成 HTML

Python 没有附带用来生成 HTML 的工具。如果开发者想要使用一个可以生成结构化 HTML 的高级框架，本书推荐使用 Robin Friedrich 开发的 HTMLGen 2.2 软件包（参见 http://starship.python.net/crew/friedrich/HTMLgen/html/main.html），但是本书中没有介绍这个软件包。要想生成 XHTML，可以使用第 24 章中介绍的解决方案。

嵌入

如果开发者喜欢的解决方案是按照 JSP、ASP 和 PHP 中流行的方式，将 Python 代码嵌入到 HTML 中使用，一种可能的方法是使用 Webware（参见第 21.3 节中对 Webware 的介绍）提供的 Python 服务器网页（Python Server Pages，PSP）。另一个特别关注于嵌入式解决方案的软件包是 Spyce（参见 http://spyce.sf.net/）。但是，对于全部但也是最简单的问题，也就是开发和维护，这些可以通过使用模板来分离逻辑和表示问题得以简化，参见下一节。Webware 和 Spyce 都可以选择支持模板以替代嵌入功能。

模板

要想生成 HTML，最好的方法是经常使用模板。要想使用模板功能，首先要创建一个模板（template），模板是一个有效 HTML 的文本字符串（通常是从文件、数据库中读取的），但是包含一些标签，也被称为占位符，用于插入动态生成的文本。程序将生成所需的文本并将其替换到模板中。在最简单的情况下，可以使用形式为‘%(name)s’

的标签。将动态生成的文本设置为某个字典 d 中名称为‘name’的键的值。Python 字符串格式化操作符%（参见第 9.3 节）就可以完成需要的所有操作：如果 t 是模板，t%d 就是所有值都被正确替换的模板的副本。

Cheetah 包

要想实现高级模板任务，本书推荐使用 Cheetah 包（参见 http://www.cheetahtemplate.org）。Cheetah 可以与 Webware 和其他 Python 服务器端 Web 框架进行特别好的互操作，第 21.3 节中提到过这一点。在已经安装了 Webware 之后，Cheetah 的模板对象就是 Webware 小程序，因此可以立即在 Webware 中部署这些小程序。开发者还可以在许多其他情况下使用 Cheetah：例如，Spyce 和 web.py（参见第 21.3 节中介绍的 web.py）都可以选择使用 Cheetah 进行模板操作，而且 TurboGears（参见第 21.3 节）也依赖于 Cheetah。Cheetah 可以处理任何目的的 HTML 模板。实际上，Cheetah 非常适合于对任何类型的结构化文本（HTML 或其他）进行模板化操作。

Cheetah 模板语言

在 Cheetah 模板中，可以使用$name 或${name}以请求插入名为 name 的变量的值。name 可以包含句点符号以请求查询对象属性或字典键。例如，$a.b.c 表示请求插入名为 a 的变量的 b 属性的 c 属性的值。在 b 是一个字典时，这个语句将被翻译为 Python 表达式 a.b['c']。如果在$替换期间遇到的对象是可调用的，Cheetah 将在查询期间调用该对象，并且不带任何参数。这种高度的多态性可以帮助非开发人员更容易地编写和管理 Cheetah 模板，因为这样可以让他们不必学习和理解各种区别。

Cheetah 模板可以包含一些指令（directive），这些指令都是以#开始的动词，可以用于注释、文件包含、控制流（条件、循环、异常处理）等；Cheetah 提供了一种建立在 Python 之上的功能非常丰富的模板语言。在简单 Cheetah 模板中，最常使用的动词如下（类似于 Python，但是在名称之前有‘$’符号，没有表示结束的冒号‘:’，不强制缩进，但是有#end 子句）。

comment text

以两个##字符开始的单行注释（还可以使用多行注释：以#*开始，以*#结束）；以 doc（在注释开始标记和 d 之间没有空格）开始的注释将被用于文档字符串（docstring）。

#break #continue #from #import #pass

类似于具有相同名称的 Python 语句。

#echo expression

计算一个 Python 表达式（名称前面带$字符）并输出结果。

#for $ *variable* in $ *container* ... #end for

类似于 Python 的 for 语句。

#include filename_expression

读取指定的文件并包含其文本，使用 Cheetah 语法对文本进行解析。

#include raw filename_expression

读取指定的文件并逐字包含其文本，不进行解析。

#if...#else if...#else...#end if

类似于 Python 的 if 语句（#else if 和#else 是可选的）。

#if...then...else...

单行的 if 语句（else 是必须有的；不允许出现 se if）。

#raw... #end raw

逐字输出文本，不检查出现的$name 或#directive。

#repeat $ *times* ... #end repeat

将某些文本重复$times 次。

#set $ variable = expression

为一个变量赋值（这个变量是模板的本地变量）。

#silent expression

计算 Python 表达式（名称前面带$字符），并隐藏其结果。

#slurp

消除下面的新行（也就是，将下一行连接到本行）。

#stop

停止处理当前的模板文件。

#while $ *condition* ... #end while

类似于 Python 的 while 语句。

请注意#echo、#silent 和$替换之间的区别。#echo $a(2)可以将使用参数 2 调用函数 a 的结果插入到模板的输出中。如果没有#echo，$a(2)将插入 a 的字符串形式（如果 a 是可调用的，则不带参数调用 a( )），后面带 3 个字符‘(2)’。#silent $a(2)使用参数 2 调用 a，并且不在模板的输出中插入任何内容。

Cheetah 包含许多其他动词，可以用来控制一些高级功能，比如变量缓存、过滤、设置和删除，以及方法定义。Cheetah 模板对象是一个类实例，可以使用继承、覆盖方法等。

但是，对于简单的模板，通常并不需要用到这样强大的机制。

模板类

Cheetah.Template 模块提供了一个类。

表 23-5

Template	`class Template(source=None,searchList=[],file=None)` 总是使用命名参数（除了可选的第一个参数）调用 Template；将来，参数的数量和顺序可能会改变。但是参数名称可以保证被保留。可以向该类传递 source 或 file 参数，但不能同时都传递。source 是一个模板字符串。file 是一个以可读模式打开的类文件对象，或者是一个以可读模式打开的文件的路径。 searchList 是一个用作顶层源以进行$name 插入的对象序列。Template 类的实例 t 将被显式添加到 t 的搜索列表的末尾（例如，如果搜索列表中没有其他对象具有属性 a 或键为‘a’的项目，模板中的$a 将插入 t.a 的值）。searchlist 默认为一个空白列表，因此，在默认情况下，t 的模板扩展只是用 t 的属性作为$替换的变量。 Template 类还允许其他关键字参数，但是上述这些是最常使用的。实例 t 提供了许多方法，但是通常只调用 str(t)，这样将返回扩展模板的字符串形式。

Cheetah 示例

下面的示例使用了 Cheetah.Template 输出包含动态内容的 HTML：

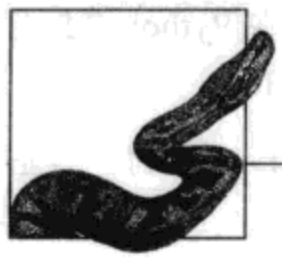
```
import Cheetah.Template
import os, time, socket

tt = Cheetah.Template.Template('''
<html><head><title>Report by $USER</title></head><body>
<h1>Report on host data</h1>
<p>Report written at $asctime:<br/>
#for $hostline in $uname
  $hostline<br/>
#end for
</p></body></html>
''', searchList=[time, os.environ])

try: tt.uname = os.uname
except AttributeError:
    tt.uname = [socket.gethostname()]

print tt
```

这个示例将实例化一个 Template 实例，并将其绑定到名称 tt 上，该实例的 source 是一个 HTML 文本字符串，其中包含一些 Cheetah 占位符（$USER, $asctime, $uname）和一个 Cheetah #for...#end for 指令。占位符$hostline 是#for 指令中的循环变量，因此在扩展时，该模板不会在搜索列表对象中搜索名称‘hostline’。这个示例使用了 searchList

参数来实例化 tt，将 time 模块和 os.environ 字典设置为搜索的一部分。对于不能在搜索列表上的对象中找到的名称，tt 的扩展将查看实例 tt 本身。因此，如果可能的话，该实例将把 tt.uname 属性绑定到 os.uname 函数（该函数将返回一个由主机描述数据组成的元组，但是只在特定平台上存在），或者绑定到一个列表，列表中的唯一项目就是 socket 模块的 gethostname 函数返回的主机名。

这个示例的最后一条语句就是 print tt。print 语句将把其参数转换为字符串，就像对每个参数调用 str 一样。因此，print tt 将扩展 tt。有些占位符的扩展将使用字典查找（$USER 将查找 os.environ['USER']），有些占位符将执行一个函数调用（$asctime 将调用 time.asctime( )），而其他一些占位符可能会以不同的方式执行（根据找到的 tt.uname，$uname 将调用这个属性——如果这个属性是可调用的，就像在该属性是 os.uname 时一样——或者在该属性已经是一个列表时，将其看作是 os.uname）。

应用于所有模板任务，而不只是 Cheetah 的一个非常重要的提示就是，模板几乎总是不变的，并不适合于保存程序逻辑。除了确实需要的程序逻辑，请不要将更多的程序逻辑放到模板中。模板引擎可以帮助开发者将计算结果的任务（最好在 Python 中完成，在任何模板之外）与将结果显示为 HTML 或其他类型的结构化文本的任务分开。模板应该只处理显示问题，并且尽可能少地包含程序逻辑。



第 24 章

结构化文本：XML

在过去的几年中，“可扩展标记语言”（eXtensible Markup Language，XML）的使用已经变得非常广泛。与 SGML（参见第 23.1 节）一样，XML 是一种元语言（metalanguage），用来描述标记语言的语言。在 XML 1.0 的基础上，XML 社区（主要在“万维网协会”[W3C]中）已经标准化了许多其他技术，比如结构描述（schema）语言、命名空间、XPath、XLink、XPointer 和 XSLT。

许多领域的工业协会已经定义了一些基于 XML 的工业专用标记语言，以推动这些领域中的应用程序之间的数据交换。这样的工业标准可以帮助应用程序交换数据，即使这些应用程序是使用不同的语言编写的，并且由不同的公司部署在不同的平台上。XML 相关技术和基于 XML 的标记语言都是在现代应用程序中实现应用程序之间、跨语言和跨平台数据交换的基础。

Python 可以非常完美地支持 XML。标准 Python 库提供了 xml 包，xml 包可以帮助开发者非常简单地使用基础的 XML 技术。第三方软件包 PyXML（http://pyxml.sf.net）扩展了标准库的 xml 功能，包括验证解析器、更丰富的 DOM 实现和像 XPath 和 XSLT 这样的高级技术。下载并安装 PyXML 以升级 Python 自己的 xml 包，因此，即使开发者并不使用 PyXML 专用功能，这样做也是一个好主意。

基于 PyXML，开发者还可以选择安装另一个免费的第三方软件包 4Suite（参见 http://4suite.org）。4Suite 提供了甚至更多用于特殊环境的 XML 解析器、像 XLink 和 XPointer 这样的高级技术，以及建立在 XML 之上的代码支持标准，比如“资源描述框架”（Resource Description Framework，RDF）。

ElementTree 软件包是可以用于 XML 处理的高度 Python 化的另一个选择（参见 http://effbot.org/zone/element-index.htm），ElementTree 的大多数功能已经被提名作为标准库模块 xml.etree（要想在 Python 2.3、Python 2.4，甚至在 Python 2.5 中使用更完整的功能，在任何时候都可以从 effbot.org 下载并安装完整的 ElementTree）在 Python 2.5 版

本中发布。ElementTree 的优美、快速和高度 Python 化的结构使其成为大多数 Python XML 应用程序选择使用的软件包，特别是在开发者可以限定应用程序在 Python 2.5 上运行时，但是也不排除其他情况。不过，本书没有进一步介绍 ElementTree。

本章只介绍标准库的 xml 包的基本功能，本书假定读者已经了解了 XML 本身的一些基础知识。

24.1 XML 解析概述

在应用程序必须解析 XML 文档时，首先，最基本的选择就是使用哪种类型的解析方法。开发者可以使用事件驱动（event-driven）解析，在这种方式下，解析器将连续读取文档，并在每次解析到文档的一个重要部分（比如一个元素）时回调应用程序，开发者也可以使用基于对象（object-based）的解析，在这种方式下，解析器将读取整个文档，并建立内存数据结构以表示该文档，然后，开发者就可以处理该文档了。SAX 是执行事件驱动解析的主要方法，DOM 是执行基于对象的解析的主要方法。对于任何一种方式，都有其他一些可供选择的方法，比如直接使用 expat 进行事件驱动解析，或者使用 ElementTree 进行基于对象的解析，但是本书在此将不介绍这些可选方法。另一个有趣的方法就是由 pulldom 支持的基于拉（pull-based）的解析，参见本章后面的章节（并且，从某种程度上讲，这种解析方法是由 ElementTree 通过 C 编码的模块 cElementTree 的 interparse 函数实现的）。

事件驱动解析只要求很少的资源，这使得该方法特别适合于解析非常大的文档。但是，事件驱动解析要求开发者构造相应的应用程序，以在解析器调用的方法中执行文档解析处理（通常需要构建一些辅助的数据结构）。面向对象的解析为开发者提供了更多的灵活性以构造开发者的应用程序，这更适合于在需要执行非常复杂的处理时使用，只要开发者可以提供基于对象的解析所需的额外资源即可（通常，这意味着只要开发者不是在处理非常大的文档即可）。基于对象的解决方案还支持需要修改或创建 XML 文档的程序，参见第 24.4 节。

作为一种通用的原则，在开发者综合考虑了各种情况之后仍无法做出决定时，本书建议开发者首先尝试使用事件驱动解析方式，只要开发者看到可以通过这种方式合理直接地执行程序的任务。事件驱动解析方式有更好的可伸缩性：如果程序可以通过事件驱动解析方式来执行任务，该程序要比使用其他方式的程序更适合于处理较大的文档。如果事件驱动解析方式有太多限制，则可以使用 pulldom（或 cElementTree.iterparse）尝试基于拉（pull）的解析方式。本书建议开发者只有在认为 DOM 是执行程序任务的唯一正确的方法时才考虑（非 pull）DOM 方式。只要开发者可以接受 DOM 方式的一些局限性：也就是程序可以支持的文档的最大长度，以及处理过程在时间和内存上的消耗，在这种情况下（而且假定开发者不能使用 ElementTree，ElementTree 提供了一个更 Python 化的 API，而且也是运行更快，消耗内存较少），DOM 可能是最好的。

24.2 使用 SAX 解析 XML

在大多数情况下，从 XML 文档提取信息的最好方法就是使用符合 SAX（XML 的简单 API）的事件驱动解析器来解析文档。SAX 定义了一个标准的 API，这个 API 可以在许多不同的底层解析器的基础上实现。SAX 的解析方式与第 23 章中介绍的大多数 HTML 解析器有些类似。由于解析器将在输入流中遇到 XML 元素、文本内容和其他重要的事件，解析器将回调类中的这些方法。基于在相关事件发生时回调类中的方法这种方式，这样的事件驱动解析也类似于 GUI 中和某些最好的、最有可伸缩性的网络架构（比如 Twisted，参见第 19 章）中普遍使用的事件驱动解决方案。各种编程领域中的事件驱动解决方案对于初学者而言显得并不自然，但是可以获得高性能和特别高的可伸缩性，这样将使得事件驱动解决方案非常适合于高负荷的情况。

要想使用 SAX，可以定义一个内容处理器类，该类是库类的一个子类并覆盖了其中的一些方法。然后，开发者可以创建一个解析器对象 p，将这个类的一个实例安装为 p 的处理器，并将输入流提供到 p 中以进行解析。p 将对该处理器调用方法以反映该文档的结构和内容。开发者的处理器的方法将执行应用程序专用处理。xml.sax 包提供了一个工厂函数以构建 p，还提供了一些很方便的函数以在典型情况下执行更简单的操作。xml.sax 还提供了一些异常类，在无效输入和其他错误下将引发这些异常。

除了内容处理器，开发者还可以选择向解析器 p 注册其他类型的处理器。开发者可以提供一个自定义的错误处理器以使用不同于普通异常引发的错误诊断策略，例如为了在解析期间诊断几个错误。开发者还可以提供一个自定义 DTD 处理器以接收有关符号和未解析的实体的信息，这些符号和未解析的实体来自于 XML 文档的“文档类型定义”（Document Type Definition，DTD）。开发者可以提供一个自定义实体解析器以高级和自定义的方式处理外部实体引用。这些高级方法极少使用，本书没有进一步介绍文档类型定义。

xml.sax 包

xml.sax 包提供了异常类 SAXException 及其子类以支持细粒度的异常处理。xml.sax 还提供了如表 24-1 所示的 3 个函数。

表 24-1

make_parser	`make_parser(`*`parsers_list`*`=[])` parsers_list 是一个字符串列表，其中是一些模块的名称，开发者可以从这些模块构建解析器。make_parser 将尝试序列中的每个模块，直到找到一个定义了 create_parser 的函数。在 parsers_list 中的模块之后，如果需要，make_parser 将继续尝试查找默认模块列表。只要可以生成一个解析器 p，make_parser 将立即终止，并返回 p。

续表

parse	parse(*file*,*handler*,*error_handler*=None)
	file 可以是一个文件名字符串或者一个以可读方式打开的类文件对象，该对象包含一个 XML 文档。handler 是 ContentHandler 类的开发者子类的一个实例，参见下面介绍的 ContentHandler。如果给出了 error_handler，则 error_handler 是 ErrorHandler 类的开发者子类的一个实例。开发者没有必要创建 ContentHandler 和/或 ErrorHandler 的子类；只需要提供与这些类相同的接口即可。归根结底，创建子类是一种很方便的方法。
	parse 函数等同于下面的代码：
	`p = make_parser()`
	`p.setContentHandler(handler)`
	`if error_handler is not None:`
	`  p.setErrorHandler(error_handler)`
	`p.parse(file)`
	这样的习惯用法在 SAX 解析中的使用是非常频繁的，因此将其包含在单个函数中使用是很方便的。在 error_handler 为 None 时，解析器将通过传播异常对错误作出反应，这个异常是 SAXException 的某个子类的一个实例。

parseString	parseString(*string*,*handler*,*error_handler*=None) 与 parse 类似，区别在于 string 是字符格式的 XML 文档。 xml.sax 还提供了一个类，开发者可以创建该类的子类以定义自己的内容处理器。
ContentHandler	class ContentHandler() ContentHandler 的子类（下面将把该类的实例命名为 h）可能会覆盖该类的几个方法，其中最常使用的方法如下。 *h*.characters(*data*)　在解析文本内容 data（一个 unicode 字符串）时调用这个方法。解析器会将文档中的每个范围的文本分割成任意数量的数据块，并使用 h.characters 分别调用这些数据块。因此，开发者实现的 characters 方法通常将缓存 data，通常将其添加到列表属性的后面。在开发者的类从其他事件知道所有相关数据都已经到达时，这个类将对该列表调用".join，并处理结果字符串。 *h*.endDocument( ) 在文档结束时调用一次这个方法。 *h*.endElement(*tag*) 在名为 tag 的元素结束时调用这个方法。 *h*.endElementNS(*name*, *qname*) 在一个元素结束，并且解析器正在处理命名空间时将调用这个方法。对于下面介绍的 startElementNS，name 和 qname 是相同的。 *h*.startDocument( ) 在文档开始时调用一次这个方法。 *h*.startElement(*tag*, *attrs*) 在名为 tag 的元素开始时，可以调用这个方法。attrs 是一个属性名到属性值的映射，参见下面一节介绍的属性。 *h*.startElementNS(*name*, *qname*, *attrs*)　在一个元素开始并且解析器正在处理命名空间时调用这个方法。name 是一个数据对（uri, localname），其中 uri 是该命名空间的 URI 或者 None，而 localname 是标签的名称。如果解析器不提供命名空间前缀功能，qname（表示当限名称）为 None，qname 还可以是文档的文本中为这个标签使用的字符串 prefix:name。attr 是一个属性名到属性值的映射，参见下面一节介绍的属性。

属性

startElement 和 startElementNS 方法的最后一个参数是一个属性对象 attr，也就是一个属性名到属性值的只读映射。对于 startElement 方法，属性名是标识符字符串。对于 startElementNS 方法，属性名是数据对（uri, localname），其中 uri 是命名空间的 URI 或者 None，而 localname 是标签的名称。除了一些映射方法之外，attr 还提供了一些方法，可以用来处理每个属性的 qname（当限名称）。

表 24-2

getValueByQName	`attr.getValueByQName(`*name*`)` 返回当限名称 name 的属性值。
etNameByQName	`attr.getNameByQName(`*name*`)` 返回一个当限名称 name 的数据对（namespace, localname）。
getQNameByName	`attr.getQNameByName(`*name*`)` 返回 name 的当限名称，该名称是一个（namespace, localname）数据对。
getQNames	`attr.getQNames()` 返回所有属性的当限名称的列表。

对于 startElement，每个 qname 都是与对应的名称相同的字符串。对于 startElementNS，qname 是与命名空间不相关的属性（也就是，uri 为 None 的属性）对应的本地名称；否则，qname 是文档的文本中为这个属性使用的字符串 prefix:name。

在以后的处理中，解析器还可以重复使用传递到 startElement 和 startElementNS 方法的 attr 对象。如果开发者需要保存一个元素的属性的副本，调用 attr.copy( )可以获得这个副本。

增量解析

所有解析器都支持 parse 方法，parse 方法可以将 XML 文档作为一个字符串或一个以可读方式打开的类文件对象进行调用。parse 直到 XML 文档结束才会返回。尽管不是全部，但是大多数 SAX 解析器还支持增量解析，这样可以在从网络连接或其他源接收文档时，每次只将一小部分 XML 文档输入到解析器中；好的增量解析器可以尽快地执行所有可能的回调函数以调用开发者的处理器类的方法，因此开发者不必在开始处理文档之前等待整个文档到达（这个处理过程就像解析操作本身所做的那样增量进行，这对于异步网络解决方案而言是一个非常好的主意，参见第 20.3 节）。具有增量解析功能的解析器 p 提供了另外 3 个方法。

表 24-3

close	*p*.close() 在 XML 文档解析结束时调用这个方法。
feed	*p*.feed(*data*) 向解析器传递一部分文档。该解析器将处理文本的某些前缀，并在缓冲区中保存其余的内容，直到下次调用 p.feed 或 p.close。
reset	*p*.reset() 在 XML 文档解析结束或被放弃之后，并在开发者开始将另一个 XML 文档输入到解析器之前调用这个方法。

xml.sax.saxutils 模块

xml.sax 包的 saxutils 模块提供了两个函数和一个类，这些函数和类提供了一些很方便的方法，可以根据一个输入 XML 文档生成 XML 输出。

表 24-4

escape	escape(*data*,*entities*={}) 返回字符串 data 的一个副本，其中字符<、>和&被更改为实体引用<、>和&。entities 是一个使用字符串作为键和值的字典；作为 entities 中的一个键的参数 data 的每个子字符串在 escape 的结果字符串中将被更改为字符串 entities[s]。例如，要想对单引号和双引号字符进行转义，除了使用尖括号和&号之外，还可以调用： `xml.sax.saxutils.escape(data, {'"': '&quot;', "'": '&apos;'})`
quoteattr	quoteattr(*data*,*entities*={}) 与 escape 类似，但是还将为结果字符串加引号以使其可以立即作为一个属性值使用，并对任意必须转义的引号字符进行转义。

XMLGenerator	class XMLGenerator(*out*=sys.stdout,*encoding*='iso-8859-1')
	这个类是 xml.sax.ContentHandler 类的子类，并实现了以特殊的编码方式 encoding 将输入的 XML 文档重新生成到给定的文件对象 out 所需的所有功能。在必须生成一个对输入的 XML 文档进行小的修改的 XML 文档时，可以创建 XMLGenerator 的子类，覆盖其中的一些方法，并将大部分的工作委托到 XMLGenerator 的方法实现中。例如，如果只需要根据字典重命名某些标签，XMLGenerator 可以让这个操作极其简单，参见下面的示例：
	`import xml.sax, xml.sax.saxutils`
	
	`def tagrenamer(infile, outfile, renaming_dict):`
	`  base = xml.sax.saxutils.XMLGenerator`
	
	`  class Renamer(base):`

续表

XMLGenerator	def rename(self, name):
	return renaming_dict.get(name, name)
	def startElement(self, name, attrs):
	base.startElement(self, self.rename(name), attrs)
	def endElement(self, name):
	base.endElement(self, self.rename(name))
	
	xml.sax.parse(infile, Renamer(outfile))

使用 xml.sax 解析 XHTML

下面的示例使用了 xml.sax 来执行一个典型的 XHTML 相关任务，这个任务非常类似于第 22 章中执行的任务。这个示例首先使用 urllib 从 Web 抓取一个 XHTML 网页，对其进行解析，然后输出从这个网页链接到其他网站的所有不同的超链接。这个示例使用了 urlparse 来检查给定网站的链接，并且只输出 URL 中包含显式‘http’方案的链接。

```
import xml.sax, urllib, urlparse

class LinksHandler(xml.sax.ContentHandler):
    def startDocument(self):
        self.seen = set()
    def startElement(self, tag, attributes):
        if tag != 'a': return
        value = attributes.get('href')
        if value is not None and value not in self.seen:
            self.seen.add(value)
            pieces = urlparse.urlparse(value)
            if pieces[0] != 'http': return
            print urlparse.urlunparse(pieces)

p = xml.sax.make_parser()
p.setContentHandler(LinksHandler())
f = urllib.urlopen('http://www.w3.org/MarkUp/')
BUFSIZE = 8192

while True:
    data = f.read(BUFSIZE)
    if not data: break
    p.feed(data)

p.close()
```

这个示例非常类似于第 22 章中的 HTMLParser 示例。在 xml.sax 模块中，解析器和处

理器是分开的对象（而在第 22 章的示例中，他们是合在一起的）。方法名称也是不同的（在这个示例中使用的是 startElement，而在 HTMLParser 示例中使用的是 handle_starttag）。在这个示例中，attributes 参数是一个映射，因此其 get 方法可以立即给出开发者感兴趣的属性值，而在第 22 章的示例中，属性是在由（name, value）数据对组成的序列中给出的，因此开发者必须对序列进行循环操作，直到找到正确的名称。尽管从细节上有这些区别，但是这两个示例的总体结构是非常接近的，并且都是简单事件驱动解析任务的典型示例。

24.3 使用 DOM 解析 XML

SAX 解析没有在内存中建立任何数据结构以表示 XML 文档。这使得 SAX 执行很快，并且具有很高的可伸缩性，因为应用程序将按照特定任务的需要建立确切的或多或少的内存数据结构。但是，如果要对相对较小的 XML 文档执行特别复杂的处理任务，开发者可能更愿意让库来建立内存数据结构以表示整个 XML 文档，然后遍历这个数据结构。XML 的标准中描述了 XML 的“文档对象模型”（Document Object Model，DOM）。DOM 对象可以将一个 XML 文档表示为树型结构，树的根就是该文档对象，而其他节点对应于元素、文本内容、元素属性等。本章中已经提到过的 ElementTree 模块提供了一种不同的，更 Python 化（并且更快）的解决方案以建立 XML 文档在内存中的表现方式，而 DOM 采用的是已有的 W3C 标准（请记住，大多数是使用其他语言开发的，比如 Java）。

Python 标准库提供了 XML DOM 标准的一个最小实现：xml.dom.minidom。minidom 将在内存中建立所有内容，具有使用 DOM 解决方案进行解析的典型优点和缺点。Python 标准库还在 xml.dom.pulldom 模块中提供了一种不同的类 DOM 解决方案。pulldom 处于 SAX 和 DOM 之间的一个非常有趣的中间地带，可以将解析事件流表示为一个 Python 迭代器对象，这样就不需要编写回调函数了，但是需要对事件进行循环操作，并检查每个事件以判断是否对该事件感兴趣。在开发者找到了一个应用程序感兴趣的事件时，可以要求 pulldom 调用 expendNode 方法以在这个事件的节点上建立 DOM 子树，然后就像在 minidom 中那样处理这个子树。Paul Prescod 是 pulldom 的作者，也是一位 XML 和 Python 专家，他认为 pulldom 的最终结果是“具有 SAX 80%的性能和 DOM 80%的便利性”。其他 DOM 解析器都是 PyXML 和 4Suite 扩展包的一部分，本章开头提到过这些扩展包。

xml.dom 包

xml.dom 包提供了异常类 DOMException 及其子类以支持细粒度的异常处理。xml.dom 还提供了一个 Node 类，通常用作 DOM 实现的所有节点的基类。Node 类本身只提供了常量属性，给出了节点类型的代码，比如，元素的 ELEMENT_NODE、属性的

ATTRIBUTE_NODE 等。xml.dom 还提供了包含某些重要的命名空间的 URI 的常数模块属性：XML_NAMESPACE、XMLNS_NAMESPACE、XHTML_NAMESPACE 和 EMPTY_NAMESPACE。

xml.dom.minidom 模块

xml.dom.minidom 模块提供了两个函数。

表 24-5

parse	parse(*file*,*parser*=None) file 是一个文件名字符串或者一个以可读模式打开的类文件对象，其中包含一个 XML 文档。parser 是 SAX 解析器类的一个实例；如果没有给出 parser，parse 将通过调用 xml.sax.make_parser( )生成一个默认 SAX 解析器。parse 将返回一个表示给定 XML 文档的 minidom 文档对象实例。
parseString	parseString(*string*,*parser*=None) 与 parse 类似，区别在于 string 是以字符形式表示的 XML 文档。

xml.dom.minidom 还提供了许多按照 XML DOM 标准定义的类。几乎所有这些类都是 Node 类的子类。Node 类提供了所有类型的节点都通用的方法和属性。xml.dom.minidom 模块中的一个著名类是 AttributeList 类，这个类不是 Node 类的子类，在 DOM 标准中被标识为 NamedNodeMap，是一个集合了 Element 类的单个节点的各种属性的映射。

要想了解与更改和创建 XML 文档相关的方法和属性，请参见第 24.4 节。这里，本书只介绍遍历 DOM 树最常使用的类、方法和属性，这些类、方法和属性通常是在通过解析一个 XML 文档建立了 DOM 树之后使用的。为了更具体和简单，本节提到了 Python 类。但是，DOM 规范可以处理抽象接口，从来不处理具体类。开发者的代码不能直接处理类对象，而只能处理这些类的实例。不要对节点进行类型测试（例如，不要对节点使用 isinstance），并且不要直接实例化节点类（而是使用第 24.4 节中介绍的工厂方法）。通常这是一种好的 Python 习惯用法，但是在这里并不是非常重要的。

节点对象

DOM 树中的每个节点 n 都是节点（Node）类的某个子类的一个实例；因此，n 可以提供 Node 类的所有属性和方法，如果需要，还可以提供适当的覆盖实现。n 最常使用的方法和属性如表 24-6 所示。

表 24-6

attributes	n.attributes 属性可以是 None，或者是包含 n 的所有属性的 AttributeList 实例。
childNodes	n.childNodes 属性是一个由 n 的所有子节点组成的列表，也可能是一个空白列表。

续表

firstChild	在 n.childNodes 为空时，n.firstChild 属性为 None；否则 n.firstChild 属性为 n.childNodes[0]。
hasChildNodes	*n*.hasChildNodes() 与 len(n.childNodes)!=0 一样，但是运行速度更快。
isSameNode	*n*.isSameNode(*other*) 在 n 和 other 引用相同的 DOM 节点时，返回 True；否则返回 False。不要使用普通的 Python 习惯用法 n is other：DOM 实现可以自由生成引用相同 DOM 节点的多个 Node 实例。因此，要想检查 DOM 节点引用的标识，应该总是使用专用的 isSameNode 方法。
lastChild	在 n.childNodes 为空时，n.lastChild 属性为 None；否则，该属性为 n.childNodes[-1]。
localName	n.localName 属性是 n 的当限名称的本地部分（在包含命名空间时相关）。
namespaceURI	在 n 的当限名称没有命名空间部分时，n.namespaceURI 属性为 None；否则，该属性为命名空间的 URI。
nextSibling	在 n 是 n 的父节点的最后一个子节点时，n.nextSibling 属性为 None；否则该属性为 n 的父节点的下一个子节点。
nodeName	n.nodeName 属性是 n 的名称字符串。这个字符串在对 n 的节点类型有意义时是一个节点专用名称（例如，在 n 是一个 Element 时，该字符串是标签名称）；否则，这个字符串是一个以‘#’开始的字符串。
nodeType	n.nodeType 属性是 n 的类型代码，一个表示 Node 类中的一个常量属性的整数。
nodeValue	在 n 不包含值时（例如，在 n 是一个元素时），n.nodeValue 属性为 None；否则，该属性为 n 的值（例如，在 n 是一个 Text 实例时，n 值为文本的内容）。
normalize	*n*.normalize() 规范化以 n 为根的整个子树，合并邻近的 Text 节点。解析操作将把 XML 文档中的文本的范围分开到任意主干中；只有在文本范围之间有区别时，规范化才能确保文本范围保持分开。
ownerDocument	n.ownerDocument 属性是包含 n 的 Document 实例。
parentNode	n.parentNode 属性是 DOM 树中 n 的父节点，如果属性节点和节点都不在 DOM 树中，则这个属性为 None。
prefix	在 n 的当限名称不包含命名空间前缀时，n.prefix 属性为 None；否则该属性为命名空间前缀。请注意，即使一个名称没有命名空间前缀，但是这个名称是有可能有命名空间的。
previousSibling	在 n 是 n 的父节点的第一个子节点时，n.previousSibling 属性为 None；否则，该属性为 n 的父节点的前一个子节点。

属性对象

属性（Attr）类是 Node 类的一个子类，表示 Element 的一个属性。除了 Node 类的属性和方法之外，Attr 的实例 a 还提供了如表 24-7 所示的属性。

表 24-7

ownerElement	a.ownerElement 属性是一个 Element 实例，并且 a 是该实例的一个属性。
specified	如果 a 是在文档中被显式指定的，则 a.specified 属性为 True，如果 a 是默认获得的，则 a.specified 属性为 False。

文档对象

文档（Document）类是 Node 类的一个子类，其实例是由 xml.dom.minidom 模块的 parse 和 parseString 函数返回的。文档中的所有节点都引用相同的 Document 节点作为其 ownerDocument 属性。但是，要想检查这一点，必须专门使用 isSameNode 方法，而不能使用 Python 的标识检查（也就是操作符 is）。除了 Node 节点的属性和方法之外，d 还提供了如表 24-8 所示的属性和方法。

表 24-8

doctype	d.doctype 属性是对应于 d 的 DTD 的 DocumentType 实例。这个属性直接来自于 d 的 XML 源文件中的!DOCTYPE 声明。
documentElement	d.documentElement 属性是作为 d 的根元素的 Element 实例。
getElementById	`d.getElementById(elementId)` 返回具有给定 ID 的文档中的 Element 实例（其中的元素属性是由 DTD 指定的 ID），如果没有这样的实例（或者底层解析器不提供 ID 信息），则返回 None。
getElementsBy-TagName	`d.getElementsByTagName(tagName)` 按照在 XML 文档中的相同顺序，返回文档中标签等于字符串 tagName 的 Element 实例组成的列表。也可能返回空白列表。在 name 为 '*' 时，将返回文档中的任何标签的所有 Element 实例组成的列表。
getElementsBy-TagNameNS	`d.getElementsByTagNameNS(namespaceURI,localName)` 按照在 XML 文档中的相同顺序，返回文档中具有给定 namespaceURI 和 localName 的 Element 实例组成的列表。也可能返回空白列表。如果 namespaceURI 和/或 localName 的值为 '*'，表示匹配相应字段的所有值。

元素对象

元素（Element）是 Node 节点的一个子类，表示标签元素。除了 Node 类提供的属性和方法之外，Element 类的实例 e 还提供了如表 24-9 所示的方法。

表 24-9

getAttribute	e.getAttribute(*name*) 返回实例 e 中名为 name 的属性的值。如果 e 中不包含名为 name 的属性，则返回空白字符串 ''。
getAttributeNS	e.getAttributeNS(*namespaceURI,localName*) 返回实例 e 中名为 namespaceURI 和 localName 的属性的值。
getAttributeNode	e.getAttributeNode(*name*) 返回一个 Attr 实例，该实例是实例 e 中名为 name 的属性，如果实例 e 中没有名为 name 的属性，则返回 None。
getAttributeNodeNS	e.getAttributeNodeNS(*namespaceURI,localName*) 返回一个 Attr 实例，该实例是实例 e 中名为 namespaceURI 和 localName 的属性，如果实例 e 中没有给定的命名空间和名称属性，则返回 None。
getElementsBy-TagName	e.getElementsByTagName(*tagName*) 按照在 XML 文档中的相同顺序，返回以 e 为根的子树中标签等于字符串 tagName 的 Element 实例列表。如果 e 的标签等于 tagName，则 e 也将被包含在该列表中。在以 e 为根的子树中没有节点包含一个等于 tagName 的标签时，getElementsbyTagName 将返回空白列表。在 tagName 为 '*' 时，getElementsbyTagName 将返回该子树中从 e 开始的所有 Element 实例组成的列表。
getElementsBy-TagNameNS	e.getElementsByTagNameNS(*namespaceURI,localName*) 按照在 XML 文档中的相同顺序，返回以 e 为根的子树中名为 namespaceURI 和 localName 属性的 Element 实例组成的列表。如果 namespaceURI 和/或 localName 的值为 '*'，则匹配相应字段的所有值。与 getElementsByTagName 方法一样，返回的列表可能会包含 e，也有可能为空白列表。
hasAttribute	e.hasAttribute(*name*) 当且仅当 e 具有一个名为 name 的属性时，返回 True。如果底层解析器从 DTD 提取相关信息，对于在 e 中具有默认值的属性，hasAttribute 还是为 True，即使没有显式指定这些属性。
hasAttributeNS	e.hasAttributeNS(*namespaceURI,localName*) 当且仅当 e 具有一个名为 namespaceURI 和 locaName 的属性时，返回 True。对于在 DTD 中具有默认值的属性，与 hasAttribute 方法相同。

使用 xml.dom.minidom 解析 XHTML

下面的示例使用了 xml.dom.minidom 来执行与前一个 xml.sax 示例相同的任务，使用 urllib 从 Web 抓取网页，对其进行解析，然后输出其中的超链接：

```
import xml.dom.minidom, urllib, urlparse

f = urllib.urlopen('http://www.w3.org/MarkUp/')
doc = xml.dom.minidom.parse(f)
```

```
links = doc.getElementsByTagName('a')
seen = set()
for a in links:
    value = a.getAttribute('href')
    if value and value not in seen:
        seen.add(value)
        pieces = urlparse.urlparse(value)
        if pieces[0] == 'http' and pieces[1]!='www.w3.org':
            print urlparse.urlunparse(pieces)
```

在这个示例中，可以得到标签为‘a’的所有元素组成的列表，如果有的话，还可以得到每个元素的相关属性。然后，开发者可以使用通常的方法处理属性的值。

xml.dom.pulldom 模块

xml.dom.pulldom 模块提供了两个函数。

表 24-10

parse	parse(*file*,*parser*=None) file 是一个文件名字符串或者一个以可读模式打开的类文件对象，其中包含一个 XML 文档。parser 是 SAX 解析器类的一个实例，如果没有给出 parser，parse 将通过调用 xml.sax.make_parser( )生成一个默认 SAX 解析器。parse 将返回一个表示给定 XML 文档的 pulldom 事件流实例。
parseString	parseString(*string*,*parser*=None) 与 parse 类似，区别在于 string 是按字符串形式表示的 XML 文档。 xml.dom.pulldom 还提供了 DOMEventStream 类，这个类是一个项目为数据对（event, node）的迭代器，其中 event 是提供事件类型的字符串，而 node 是 Node 类的适当子类的一个实例。event 可能的值是一些作为常量的大写字符串，这些值也可以作为 xml.dom.pulldom 中具有相同名称的常量属性：CHARACTERS、COMMENT、END_DOCUMENT、END_ELEMENT、IGNORABLE_ WHITESPACE 、 PROCESSING_INSTRUCTION 、 START_DOCUMENT 和 START_ELEMENT。 DOMEventStream 类的实例 d 还提供了另外一个重要的方法。
expandNode	*d*.expandNode(*node*) node 必须是到目前为止对 d 进行迭代返回的最后一个 Node 实例，也就是，最后一次调用 d.next( )返回的 Node 实例。expandNode 将对对应于以 node 为根的子树的那部分 XML 文档流进行处理，这样就可以使用常规的 minidom 解决方案来访问这个子树了。为了这个目的，d 将对其本身进行迭代，这样，在调用 expandNode 之后，下一次调用 d.next( )时将继续处理使用这种方法扩展的子树之后的节点。

使用 xml.dom.pulldom 解析 XHTML

下面的示例使用了 xml.dom.pulldom 来执行一个与前面的示例相同的任务，使用 urllib 从 Web 抓取一个网页，对其进行解析，然后输出其中的超链接：

```
import xml.dom.pulldom, urllib, urlparse

f = urllib.urlopen('http://www.w3.org/MarkUp/')
doc = xml.dom.pulldom.parse(f)
seen = set()
for event, node in doc:
    if event=='START_ELEMENT' and node.nodeName=='a':
        doc.expandNode(node)
        value = node.getAttribute('href')
        if value and value not in seen:
            seen.add(value)
            pieces = urlparse.urlparse(value)
            if pieces[0] == 'http' and pieces[1]!='www.w3.org':
                print urlparse.urlunparse(pieces)
```

这个示例只选择了标签为‘a’的元素。对于每个元素，要求完整地扩展，然后就像在minidom示例中一样进行处理（也就是，如果有的话，获得相关的属性，然后按常规方法对该属性的值进行处理）。在这种特殊情况下，实际上不需要进行扩展，因为不需要处理以每个 tag 为‘a’的元素为根的子树，只需要处理属性即可，而属性不需要调用expandNode就可以进行访问。因此，即使删除了调用doc.expandNode的代码，这个示例的执行还是相同的。不过，在这个示例中调用expandNode是为了显示通常是如何在这样的上下文环境中使用pulldom的这个关键方法的。

24.4 更改和生成 XML

就像对 HTML 和其他类型的结构化文本一样，输出一个 XML 文档最简单的方法通常就是使用 Python 的普通字符串和文件操作来准备和写入文件，参见第 9 章和第 10.3 节。模板（参见第 23.5 节）通常也是最好的解决方案之一。创建 XMLGenerator 类的子类（参见的第 24.2 节介绍的 xml.sax.saxutils 模块中的 XMLGenerator 类）也是生成一个与输入 XML 文档类似，有一些小的更改的 XML 文档的好方法。

xml.dom.minidom 模块还提供了另外一种可能的方法，因为该模块的类支持一些方法以生成、插入、删除和改变用来表示该文档的 DOM 树中的节点。开发者可以通过解析操作创建一个 DOM 树，然后改变该 DOM 树，或者也可以创建一个空白 DOM 树，并从头开始创建该 DOM 树。开发者还可以使用 Document 实例的 toxml、toprettyxml 或 writexml 方法输出结果 XML 文档。开发者也可以对作为子树的根的 Node 调用这些方法以输出一个子树。本章开始的介绍中提到的ElementTree模块也提供了类似的功能(但是提供了更 Python 化的 API 和高得多的性能)。

文档对象的工厂方法

Document 类提供了工厂方法以创建 Node 子类的实例。Document 实例 d 最常使用的工

厂方法如表 24-11 所示。

表 24-11

createComment	*d*.createComment(*data*) 使用文本 data 创建并返回 Comment 类的一个实例 c 以作为注释。
createElement	*d*.createElement(*tagname*) 使用给定标签创建并返回 Element 类的一个实例 e 以作为元素。
createTextNode	*d*.createTextNode(*data*) 使用文本 data 创建并返回 TextNode 类的一个实例 t 以作为文本节点。

元素对象的变异方法

Element 类的实例 e 提供了以下方法来删除和添加属性。

表 24-12

removeAttribute	*e*.removeAttribute(*name*) 删除 e 中名为 name 的属性。
setAttribute	*e*.setAttribute(*name*,*value*) 更改 e 中名为 name 的属性，让该属性的值为给定的 value，或者，如果 e 中没有名为 name 的属性，则使用给定的 name 和 value 向 e 中添加一个新属性。

节点对象的变异方法

Node 类的实例 n 提供了以下方法来删除、添加和替换子节点。

表 24-13

appendChild	*n*.appendChild(*child*) 将 child 作为 n 的最后一个子节点，不管该 child 的父节点是什么（包括 n 或 None）。
insertBefore	*n*.insertBefore(*child*,*nextChild*) 将 child 作为 n 的子节点，并放在 nextChild 之前，不管该 child 的父节点是什么（包括 n 或 None）。nextChild 必须是 n 的一个子节点。
removeChild	*n*.removeChild(*child*) 让 child 没有父节点并返回该 child。child 必须是 n 的一个子节点。
replaceChild	*n*.replaceChild(*child*,*oldChild*) 将 child 作为 n 的子节点，并将其放在 oldChild 的位置，不管该 child 的父节点是什么（包括 n 或 None）。oldChild 必须是 n 的一个子节点。返回 oldChild。

节点对象的输出方法

Node 类的实例 n 提供了如表 24-14 所示的方法来输出以 n 为根的子树。

表 24-14

toprettyxml	*n*.toprettyxml(*indent*='\t',*newl*='\n') 返回一个普通或 Unicode 字符串，该字符串为以 n 为根的子树的 XML 源，使用 indent 缩进嵌套的标签，并使用 newl 结束行。
toxml	*n*.toxml() 与 n.toprettyxml(",")类似，也就是，不插入无关的空白字符。
writexml	*n*.writexml(*file*,*encoding*='None') 使用指定的 encoding 将以 n 为根的子树的 XML 源写入到以可写模式打开的类文件对象 file 中。如果没有给出 encoding，则 file.write 必须接受一个 unicode 参数。

使用 xml.dom.minidom 更改和输出 XHTML

下面的示例使用了 xml.dom.minidom 来分析一个 XHTML 网页，并将其输出到标准输出中，在该超链接之前在三重圆括号中显示每个超链接的目的 URL。

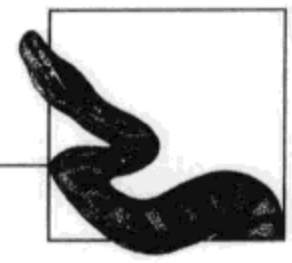
```
import xml.dom.minidom, urllib, sys

f = urllib.urlopen('http://www.w3.org/MarkUp/')
doc = xml.dom.minidom.parse(f)
as = doc.getElementsByTagName('a')
for a in as:
    value = a.getAttribute('href')
    if value:
        newtext = doc.createTextNode(' (((%s)))'%value)
        a.parentNode.insertBefore(newtext,a)

doc.writexml(sys.stdout, 'utf-8')
```

这个示例使用了‘utf-8’编码方式，因为这是 XML 标准指定的默认编码方式，但是，根据开发者的终端窗口所支持的编码方式，可能需要更改为适用的编码方式。

第 5 部分

扩展和嵌入

第 25 章



扩展和嵌入经典 Python

经典 Python 运行在一个可移植的，C 语言编码的虚拟机上。Python 的内置对象都是以 C 语言编码的，比如数字、序列、字典、集合和文件，这些也是 Python 的标准库中的几个模块。现代的平台支持动态加载库，在 Windows 平台上，该库的文件扩展名为.dll；在 Linux 和 Mac 平台上，库的文件扩展名为.so，而编译 Python 将生成这样的二进制文件。开发者可以使用 C 语言编写自己的扩展模块（使用本章介绍的 Python C API）来生成和部署动态库，而 Python 脚本和交互式会话可以在以后通过 import 语句使用这些动态库，参见第 7.1 节。

扩展 Python 意味着建立 Python 代码可以导入（import）的模块，从而访问这些模块提供的功能。嵌入 Python 意味着从开发者的应用程序执行 Python 代码。要想让这些代码的执行是有用的，Python 代码必须可以访问应用程序的某些功能。因此，实际上嵌入也隐含着某些扩展功能，以及少数专门的嵌入操作。扩展 Python 的 3 个主要原因可以总结如下：

- 以底层语言重新实现某些功能（最初是以 Python 编码的），期望获得更好的性能；
- 让 Python 代码访问以底层语言编码（或者，至少可以从底层语言调用）的库提供的某些已有功能；
- 让 Python 代码访问一个应用程序的某些已有功能，这个程序在将 Python 嵌入为该应用程序的脚本语言的进程中。

Python 的在线文档中也详细地介绍了 Python 的嵌入和扩展功能；读者可以在网站 http://www.python.org/doc/ext/ext.html 上找到比较深入的指南，在网站 http://www.python.org/doc/api/api.html 上找到一个参考指南。在 Python 的扩展文档源代码中也非常好地介绍了嵌入和扩展功能的详细信息。读者可以下载 Python 的源代码发布版本并学习 Python 的内核源代码、C 语言编码的扩展模块，以及为了学习目的而提供的示例扩展功能。

本章介绍了使用 C 语言扩展和嵌入 Python 的基础知识。本章还提到，但是没有深入介绍扩展 Python 的其他可用方法。

25.1 使用 Python 的 C API 扩展 Python

一个名为 x 的 Python 扩展模块将以适当目录（通常是 Python 库目录下的 site-pacakges 子目录）中的相同文件名（在 Windows 平台上是 x.pyd；在大多数类 Unix 平台上是 x.xo）存在于一个动态库中。开发者通常可以从一个 C 语言源代码文件 x.c 创建扩展模块 x，这个文件的总体结构为：

```
#include <Python.h>

/* 此处省略: x 模块的模块体*/

void
initx(void)
{
     /* 此处省略: 名为 x 的模块的初始化代码*/
}
```

在已经创建并安装了扩展模块之后，Python 语句 import x 可以加载该动态库，然后定位并调用名为 initx 的函数，这个函数必须执行初始化名为 x 的模块对象所需的所有操作。

编译和安装 C 语言编码的 Python 扩展

要想创建和安装一个 C 语言编码的扩展模块，最简单和有效的方法就是使用发布工具 distutils，参见第 7.4 节。在与 x.c 相同的目录中，放置一个包含以下语句的名为 setup.py 的文件。

```
from distutils.core import setup, Extension
setup(name='x', ext_modules=[ Extension('x', sources=['x.c']) ])
```

从这个目录的 shell 命令提示符下，现在可以运行：

```
C:\> python setup.py install
```

以创建并安装该模块，这样就可以在安装的 Python 程序中使用这个模块了。distutils 将使用正确的编译器和连接器命令和标记，执行所需的所有编译和链接步骤，然后将结果动态库复制到一个适当的目录中，这个目录取决于 Python 的安装目录（根据安装的详细情况，开发者可能需要拥有安装目录的管理员或超级用户权限；例如，在 Mac 或 Linux 平台上，开发者需要运行 sudo python setup.py install）。然后，开发者自己的 Python 代码就可以使用 import x 语句访问安装的模块了。

需要的 C 编译器

要想将 C 语言编码的扩展程序编译为 Python 中的扩展，通常需要使用与编译这个

Python 版本相同的 C 编译器。对于大多数平台，这通常意味着平台附带的，或者可以从平台的网站免费下载的免费 gcc 编译器。在 Macintosh 上，gcc 是由苹果公司的免费 XCode（也叫作开发工具）集成开发环境（IDE）附带的。

在 Windows 平台上，开发者通常需要用到 Microsoft 公司的产品 Visual Studio 7.1（也叫作，Visual Studio 2003）集成开发环境。但是，开发者也有可能不需要购买 Microsoft 公司的这个产品就在 Windows 平台上编译 C 语言编码的扩展。在网站 http://www.vrplumber.com/programming/mstoolkit/上，可以找到一些使用说明，这些文档显示了如何通过下载、安装和配置 5 个其他 Microsoft 组件来执行这项任务，这些 Microsoft 组件是不用支付许可证费用就可以下载的。不幸的是，在编写本书的时候，可以免费下载的 Microsoft Visual Studio 2005 还不能为 Windows 下的 Python 标准发布版本编译扩展，因为 Python 2.4 和 Python 2.5 版本都是使用 Visual Studio 2003 编译的。

C 语言编码的扩展在 Python 版本之间的兼容性

一般来说，被编译为在 Python 的某个版本下使用的 C 语言编码的扩展不能保证在 Python 的其他版本下运行。例如，为 Python 2.4 编译的一个扩展版本只能确定在 Python 2.4 版本下运行，既不能在 Python 2.3，也不能在 Python 2.5 下运行。在某些平台上，比如 Windows，甚至不能在 Python 的不同版本下运行一个扩展；而在其他平台上，比如 Linux 或 Mac OS X，一个给定的扩展可能碰巧可以在多个 Python 版本下运行，但是，在该模块被导入时，至少会看到一个警告消息，因此，最谨慎的做法就是注意这些警告信息，并适当地重新编译该扩展。

另一方面，从 C 语言源代码的角度看，几乎总是具有兼容性的。只是在新的 Python 2.5 中有一个例外，在 Python 2.5 中，许多常用的 C 语言 int 型值现在变成了 Py_ssize_t 类型——相当于 32-位平台上的 int 型，但是在 64-位平台上，就是一个 64-位带符号整型（明确的讲，相当于带符号的 size_t 类型）。这个 C 语言 API 的改变可以帮助开发者处理 64-位平台上一个 Python 2.5 序列中包含多于 20 亿个项目的情况，并且不会对 32-位平台产生任何影响。如果开发者的 C 语言编码的扩展最初是在 Python 的以前版本下开发和测试的，在为 Python 2.5 重新编译这些源代码时，C 编译器会生成错误和警告消息，其原因几乎可以确定是：开发者需要（在 C 编译器生成的错误和警告消息的帮助下）找到并更改必须变成 Py_ssize_t 类型的 int 型。可以从网站 http://svn.effbot.python-hosting.com/stuff/sandbox/python/ssizecheck.py 免费下载一个简单的检查工具，这个工具可以让这个处理过程更加简单。要想确保开发者自己的扩展仍能与 Python 2.4 和更早的版本兼容，并且在 64-位计算机上的 Python 2.5 下也可以正确执行，可以在源代码的前面插入以下代码行：

```
#if PY_VERSION_HEX < 0x02050000
typedef int Py_ssize_t;
#endif
```

C 语言编码的 Python 扩展模块概述

开发者自己的 C 语言函数 initx 通常具有以下的总体结构：

```
void
initx(void)
{
    PyObject* thismod = Py_InitModule3("x", x_methods, "docstring for x");
    /* 可选项：调用 PyModule_AddObject(thismod, "somename", someobj)
       和其他 Python C API 函数以完成对模块对象 thismod 及其类型（如果有的话）
       和其他对象的准备。
    */
}
```

参见第 25.1 节以了解更多详细信息。x_methods 是一个数据结构为 PyMethodDef 的数组。x_methods 数组中的每个 PyMethodDef 数据结构描述了一个 C 语言函数，可以让导入 x 的 Python 代码使用模块 x。每个这样的 C 语言函数具有如下总体结构：

```
static PyObject*
func_with_named_arguments(PyObject* self, PyObject* args,
PyObject* kwds)
{
    /*  省略：函数体，通过 Python C API 函数 PyArg_ParseTupleAndKeywords
        访问参数，并返回一个 PyObject* 结果，如果错误，则返回 NULL */
}
```

或者一个稍微简单一点的变体，比如：

```
static PyObject*
func_with_positional_args_only(PyObject* self, PyObject* args)
{
    /* 省略：函数体，通过 Python C API 函数 PyArg_ParseTuple
       访问参数，并返回一个 PyObject* 结果，如果错误，则返回 NULL */
}
```

第 25.1 节中介绍了 C 语言编码的函数如何访问 Python 代码传递的参数。第 25.1 节中介绍了这些函数是如何创建 Python 对象的，第 25.1 节中介绍了这些函数是如何引发异常，并将异常传播到调用这些函数的 Python 代码的。在开发者的模块定义了新的 Python 类型（和/或 Python 可调用函数）时，开发者的 C 代码将定义一个或多个 PyTypeObject 数据结构的实例。第 25.1 节中介绍了这个主题。

第 25.1 节中给出了一个简单示例，这个示例使用了以上所有概念。一个玩具级的“Hello World”示例可以像下面这样简单：

```
#include <Python.h>

static PyObject*
helloworld(PyObject* self)
{
    return Py_BuildValue("s", "Hello, C-coded Python extensions world!");
}
```

```
static char helloworld_docs[] =
    "helloworld(): return a popular greeting phrase\n";

static PyMethodDef helloworld_funcs[] = {
    {"helloworld", (PyCFunction)helloworld, METH_NOARGS, helloworld_docs},
    {NULL}
};

void
inithelloworld(void)
{
    Py_InitModule3("helloworld", helloworld_funcs,
                   "Toy-level extension module");
}
```

将上面这段代码保存为 helloworld.c，并使用 distutils 工具通过 setup.py 脚本对其进行编译。在运行了“python setup.py install”之后，就可以使用新安装的模块了——例如，从 Python 的交互式会话——比如：

```
>>> import helloworld
>>> print helloworld.helloworld()
Hello, C-coded Python extensions world!
>>>
```

Python 的 C API 函数的返回值

Python C API 中的所有函数可以返回一个 int 或 PyObject*类型的值。大多数返回 int 型值的函数在成功的情况下返回 0，返回-1 表示错误。有些函数返回 True 或 False 这样的结果：这些函数返回 0 来表示 False，返回一个不等于 0 的整数表示 True，不指示错误结果。返回 PyObject*类型值的函数在错误的情况下返回 NULL。参见第 25.1 节了解更多有关 C 语言编码的函数如何处理和引发错误的详细信息。

模块初始化

initx 函数最小必须包含一个对 C 语言 API 提供的模块初始化函数的调用。开发者可以总是使用 Py_InitModule3 函数。

表 25-1

Py_InitModule3	PyObject* Py_Init Module3 (char* *name*, PyMethod Def* *methods*, char* *doc*) name 是开发者正在初始化的模块的 C 语言字符串名称（例如，“name”）。methods 是一个数据结构为 PyMethodDef 的数组，参见下一节介绍的 PyMethodDef 数据结构。doc 是将会成为该模块的文档字符串的 C 语言字符串。Py_IntitMadule3 将返回一个 PyObject*类型的值，这个值是一个对新模块对象的借用引用，参见第 25.1 节。实际上，这意味着如果开发者不再需要对这个模块执行初始化操作，则可以忽略其返回值。否则，需

续表

Py_InitModule3	要将这个返回值赋值给一个类型为 PyObject*的 C 语言变量，并继续初始化。 Py_InitModule3 可以初始化该模块对象以包含表 methods 中描述的函数。如果需要进一步地初始化，可能需要添加其他模块属性，并且通常最好使用以下这些很方便的函数来执行。
PyModule_AddIntConstant	int PyModule_AddIntConstant(PyObject* *module*, char**name*,int *value*) 向 module 模块添加一个名为 name，值为整型值 value 的属性。
PyModule_AddObject	int PyModule_AddObject(PyObject* *module*,char* *name*,PyObject* *value*) 向 module 模块添加一个名为 name，值为 value 的属性，并偷用一个到该值的引用，参见第 25.1 节。
PyModule_AddStringCon-stant	int PyModule_AddStringConstant(PyObject* *module*, char* *name*,char* *value*) 向 module 模块添加一个名为 name，值为字符串值 value 的属性。 有些模块初始化操作可以很便利地通过执行包含 PyRun_String 函数（参见第 25.3 节中介绍的 PyRun_String 函数）的 Python 代码来执行，同时将该模块的字典同时作为 globals 和 locals 参数。如果开发者发现自己广泛地使用了 PyRun_String 函数，而不是偶尔为了方便而使用 PyRun_String，可以考虑将该扩展模块一分为二的可能性：一个是 C 语言编码的扩展模块，提供原始的快速功能；另一个是 Python 模块，该模块包装了 C 语言编码的扩展以提供更加便利的功能。 在开发者确实需要获得一个模块的字典时，可以使用 PyModule_GetDict 函数。
PyModule_GetDict	PyObject* PyModule_GetDict(PyObject* *module*) 返回一个对 module 模块的字典的借用引用。开发者不应该使用 PyModule_GetDict 来执行 PyModule_Add 函数（参见前面介绍的 PyModule_AddObject 方法）支持的特殊任务：只有在为了实现像支持 PyRun_String 的使用这样的目的时才使用 PyModule_GetDict。 如果开发者需要访问其他模块，可以调用 PyImport_Import 函数导入该模块。
PyImport_Import	PyObject* PyImport_Import(PyObject* *name*) 导入 Python 字符串对象 name 指定的模块，并返回一个到该模块对象的新引用，就像 Python 的__import__(name)一样。PyImport_Import 是用来导入一个模块的最高级、最简单，也最常用的方法。 要特别小心 PyImport_ImportModule 函数的使用，这个函数通常看起来非常方便，因为该函数可以接受一个 char*参数。PyImport_ImportModule 是在底层操作的，绕过了可能会有效的所有导入钩子函数，因此，使用该函数的扩展非常难以合并到其他一些软件包中，比如使用 py2exe 和 cx_Freeze 工具创建的包，参见第 27.2 节和第 27.4 节。开发者应该总是调用 PyImport_Import 来执行任何所需的导入，除非开发者有非常特殊的需要，并且确切地知道自己在干什么。

PyMethodDef 数据结构

要想向一个模块添加函数（或者向新类型添加非特殊的方法，参见第 25.1 节），开发者必须在一个数据结构为 PyMethodDef 的数组中描述函数或方法，并使用一个“哨兵”（sentinel）数据结构（也就是，其中的字段都是 0 或 Null 的数据结构）终止这个数组。PyMethodDef 数据结构的定义如下：

```
typedef struct {
    char* ml_name;                  /* 函数或方法的 Python 名称 */
    PyCFunction ml_meth;            /* 指向 C 函数 impl 的指针 */
    int ml_flags;                   /* 描述如何传递参数的标记 */
    char* ml_doc;                   /* 函数或方法的文档字符串 */
} PyMethodDef
```

开发者必须将第二个字段转换为（PyCFunction），除非这个 C 函数的签名恰好就是 PyObject* function(PyObject* self, PyObject* args)，这是 PyCFunction 的 typedef。这个签名在 ml_flags 为 METH_0 或 METH_VARARGS 时是正确的，METH_0 用来指示一个可以接受单个参数的函数，METH_VARARGS 用来指示一个可以接受位置参数的函数。对于 METH_0，args 是单个参数。对于 METH_VARARGS，args 是一个由所有参数组成的元组，这个元组可以使用 C API 函数 PyArg_ParseTuple 进行解析。但是，ml_flags 还可以是 METH_NOARGS 或 METH_KEYWORDS，METH_NOARGS 用来指示一个不接受任何参数的函数，而 METH_KEYWORDS 用来指示一个可以接受位置参数和命名参数的函数。对于 METH_NOARGS，函数的签名是不带任何参数的 PyObject* function(PyObject* self)。对于 METH_KEYWORDS，函数的签名是：

```
PyObject* function(PyObject* self, PyObject* args, PyObject* kwds)
```

args 是一个由位置参数组成的元组，而 kwds 是由命名参数组成的字典；这两个参数都可以使用 C API 函数 PyArg_ParseTupleAndKeywords 进行解析。在这些情况下，开发者需要将第二个字段转换为（PyCFunction）。

在一个 C 语言编码的函数实现了一个模块的函数时，不管 ml_flags 字段的值是什么，这个 C 函数的 self 参数总是 NULL。在一个 C 语言编码的函数实现了一个扩展类型的非特殊方法时，self 参数指向调用该方法的实例。

引用计数

Python 对象是在堆基础上实现的，而 C 代码将 Python 对象看作是类型为 PyObject*的指针。每个 PyObject 都会计算有多少对该对象的引用是尚未完成的，并将在引用的数量下降到 0 时删除其本身。要想这样做，开发者的代码必须使用 Python 提供的宏：使用 Py_INCREF 可以向 Python 对象添加一个引用，而使用 Py_DECREF 可以从 Python 对象删除一个引用。Py_XINCREF 和 Py_XDECREF 宏就像 Py_INCREF 和 Py_DECREF 一样，但是开发者还可以对一个空指针使用这两个宏。对非空指针的测试是在

Py_XINCREF 和 Py_XDECREF 宏内隐式执行的，这样，在开发者不知道指针是否可能为空时，就不必显式编写测试代码了。

对于开发者的代码通过调用其他函数或者被其他函数调用而接受到的 PyObject* p，如果提供 p 的代码已经调用了 Py_INCREF，则 PyObject* p 也被称为一个“新引用”（new reference）。开发者可以对一个借用引用调用 Py_INCREF，以将其放到开发者自己的一个引用中；如果开发者在可能会导致借用的引用计数减少的代码中需要跨函数使用该引用，则必须这样做。开发者必须总是在丢弃或改写自己的引用之前调用 Py_DECREF，但是永远都不要对不是自己的引用调用 Py_DECREF。因此，理解哪些交互操作可以转移引用的所有者，以及哪些操作依赖于引用借用绝对是至关重要的。对于 CAPI 中的大多数函数，以及开发者编写和 Python 调用的所有函数，可以应用下面的通用原则：

- PyObject*参数是借用引用；
- 作为函数的结果返回的一个 PyObject*将转移所有者。

对于这两个原则，C API 中的有些函数有几个例外情况。PyList_SetItem 和 PyTuple_SetItem 可以“偷取”一个其正在设置的项目的引用（但是不能对其正在设置的列表或元组对象执行相同的操作）。因此，应该使用这两个函数的更快版本，也就是作为 C 的预编译宏存在的 PyList_SET_ITEM 和 PyTuple_SET_ITEM。对于 PyModule_AddObject 也是如此，参见第 25.1 节中介绍的 PyModule_AddObject 函数。对于第一个原则，没有其他例外情况存在。这些例外情况的基本原理（可能会帮助开发者记住例外情况）是，开发者正在设置的对象通常都是出于这个偷用引用的目的而创建的，因此引用偷用的语义可以节省开发者的工作，不必在创建之后立即调用 Py_DECREF 了。

第二个原则的例外情况要比第一个原则多。在几种情况下，返回的 PyObject*是一个借用引用，而不是一个新引用。名称以 PyObject_、PySequence_、PyMapping_和 PyNumber 开始的抽象函数将返回新引用。这是因为开发者可以对许多类型的对象调用这些函数，但是可能不会有任何其他对这些函数返回的结果对象的引用（也就是，可能必须临时创建返回的对象）。名称以 PyList_、PyTuple、PyDict_等开始的具体函数，在其返回的对象的语义可以确信，在某些地方肯定有其他一些对返回的对象的引用时，将返回一个借用引用。

本章针对介绍的所有函数，指出了这些原则的所有例外情况（也就是，借用引用的返回，以及从参数偷用引用的极少情况）。在本章没有显式提到某个函数是一个例外时，这意味着该函数遵循这样的原则：该函数的 PyObject*参数（如果有的话）是借用引用，而其 PyObject*结果（如果有的话）是一个新引用。

访问参数

从 Python 中可以不带任何参数调用 PyMethodDef 数据结构中的 ml_flags 被设置为 METH_NOARGS 的函数。对应的 C 函数具有一个签名，其中包含一个参数 self。在

ml_flags 为 METH_0 时，Python 代码必须确定地使用一个参数调用该函数。C 函数的第二个参数是一个对象的借用引用，Python 调用者将把该对象传递为这个参数的值。

在 ml_flags 为 METH_VARARGS 时，Python 代码可以带任意数量的位置参数调用该函数，Python 解释器将隐式将这些参数收集到一个元组中。C 函数的第二个参数是一个对该元组的借用调用。然后，开发者的 C 代码就可以调用 PyArg_ParseTuple 函数了。

表 25-2

PyArg_ParseTuple

`int PyArg_ParseTuple(PyObject* tuple,char* format,...)`

返回 0 表示错误，返回一个不等于 0 的值表示成功。tuple 是作为 C 函数的第二个参数的 PyObject*。format 是一个描述强制和可选参数的 C 字符串。PyArg_ParseTuple 的以下参数都是 C 变量的地址，可以将从元组中提取的值放到这些变量中。C 变量中的任意 PyObject*变量都是借用引用。下表列出了常用的代码字符串，可以将其中的零个或多个连接在一起以形成字符串 format。

PyArg_ParseTuple 的格式代码

代码	C 语言类型	含　义
c	char	长度为 1 的 Python 字符串变成 C 语言中的 char
d	double	Python 中的 float 变成 C 语言中的 double
D	Py_Complex	Python 中的 complext 变成 C 语言中的 Py_Complex
f	float	Python 中的 float 变成 C 语言中的 float
i	int	Python 中的 int 变成 C 语言中的 int
l	long	Python 中的 int 变成 C 语言中的 long
L	long long	Python 中的 int 变成 C 语言中的 long long（Windows 上的_ _int64）
O	PyObject*	获得对 Python 参数的非 NULL 借用引用
O!	type+PyObject*	与代码 O 一样，加上了类型检查（见下文）
O&	convert+void*	任意转换（见下文）
s	char*	没有将 NULL 嵌入 C 的 char*中的字符串
s#	char*+int	对应于 C 的地址和长度的任意 Python 字符串
t#	char*+int	对应于 C 的地址和长度的只读单段缓冲区

续表

PyArgParseTuple	代码	C 语言类型	含　义
	u	Py_UNICODE*	没有将NULL嵌入到C的Python Unicode字符串
	u#	Py_UNICODE*+int	任意 Python Unicode C 地址和长度
	w#	char*+int	对应于 C 的地址和长度的读/写单段缓冲区
	z	char*	与 s 类似，还可以接受 None（可以将 C char*设置为 NULL）
	z#	char*+int	与 s#类似，还可以接受 None（可以将 C char*设置为 NULL）
	(...)	as per ...	一个 Python 序列被处理为每个项目一个参数
	|		后面的参数都是可选的
	:		格式化结束，后面是错误消息的函数名
	;		格式化结束，后面是整个错误消息文本

格式代码 d 到 L 可以从 Python 接受数值参数。Python 将强制对应的值。例如，代码 i 可以对应于 Python 的 float；小数部分将被截短，就像调用了内置函数 int 一样。Py_Complex 是一个 C 语言数据结构，其中包含两个名为 real 和 imag 的字段，都是 double 类型。

O 是最通用的格式代码，可以接受任意参数，开发者可以根据需要以后再检查和/或转换该参数。其变体 O!对应于变量参数中的两个参数：第一个是 Python 类型对象的地址，然后是 PyObject*的地址。在将该 PyObject*设置为指向一个值时，执行 O!将检查这个对应值是否属于给定的类型（或者这个类型的任意子类型）；否则，将引发 TypeError 错误（整个调用失败，而且该错误被设置为一个适当的 TypeError 实例，参见第 25.1 节）。变体 O&也对应于变量参数中的两个变量：第一个是开发者编码的转换器函数的地址，然后是一个 void*（也就是，任意地址）。转换器函数必须包含签名 int convert (PyObject*, void*)。Python 将调用开发者的转换函数，并将从 Python 传递的值作为第一个参数，将变量参数中的 void*作为第二个参数。转换函数必须返回 0 并引发一个异常（参见第 25.1 节）以指示一个错误，或者返回 1 并通过得到的 void*保存任何适当的信息。

格式代码 s 可以从 Python 接受一个字符串，从变量参数中接受一个 char*地址。s 将把 char*更改为指向该字符串的缓冲区，C 代码将把这个缓冲区看作是一个只读的、以 NULL 结尾的 char 数组（也就是，一个典型的 C 字符串；但是，开发者的代码不能修改该字符串）。Python 字符串必须不包含嵌入的空字符。s#与此类似，但是对应于变量参数中的两个参数：第一个是 char*的地址，然后是一个被设置为该字符串的长度的 int 的地址。Python 字符串可以包含嵌入的空字符，因此 char*可以被设置为指向这个缓冲

区。u 和 u#也是类似的，但是可以接受 Unicode 字符串，并且 C 指针必须是 Py_UNICODE*，而不是 char*。Py_UNICODE 是 Python.h 中定义的一个宏，并且对应于这个实现（通常是一个 C 的 wchar_t，但不总是如此）中的 Python Unicde 字符的类型。

t# 和 w# 都类似于 s#，但是对应的 Python 参数可以是一个有关缓冲区协议的类型的任意对象，t#和 w#分别是只读和度/写的。字符串是只读缓冲区的一个典型示例。mmap 和 array 实例都是读/写缓冲区的典型示例，就像所有读/写缓冲区一样，这两个实例在要求只读缓冲区的地方（也就是，t#）也是可接受的。

在其中的一个参数就是一个已知固定长度的 Python 序列，开发者可以对其中的每个项目和变量参数中对应的 C 地址使用格式代码（通过将格式代码组合到一个圆括号中）。例如，格式代码(ii)对应于由两个数字组成的 Python 序列，在其余的参数中，还对应于两个 int 地址。

格式字符串可能还会包含竖线（|）字符以指示后面的所有参数都是可选的。在这种情况下，在开发者调用 PyArg_ParseTuple 之前，必须将 C 变量的值初始化为适当的默认值，并在变量参数中传递后面的参数的地址。PyArg_ParseTuple 不会更改对应于没有在一个从 Python 到 C 语言编码的函数的给定调用中传递的可选参数的 C 变量。

格式字符串可以选择以:name 结束，以指示如果需要任意错误消息，则该 name 必须被用作函数名。此外，格式字符串还可以以;text 结束，以指示如果 PyArg_ParseTuple 检测到错误，则 text 必须被用作整个错误消息（这种形式是极少使用的）。

一个 PyMethodDef 数据结构中的 ml_flags 被设置为 METH_KEYWORDS 的函数可以接受位置和关键字参数。Python 代码可以带任意数量的位置参数和关键字参数调用该函数，Python 解释器将把位置参数集合到一个元组中，并将关键字参数集合到一个字典中。C 函数的第二个参数是一个到该元组的借用引用，第三个参数是一个到该字典的借用引用。然后，开发者的 C 代码可以调用 PyArg_ParseTupleAndKeywords 函数。

表 25-3

PyArg_ParseTupleAndKeywords	`int PyArg_ParseTupleAndKeywords(PyObject* tuple, PyObject* dict, char* format, char** kwlist,...)` 返回 0 表示错误，返回一个不等于 0 的值表示成功。tuple 是作为 C 函数的第二个参数的 PyObject*。dict 是作为 C 函数的第三个参数的 PyObject*。format 与 PyArg_ParseTuple 中的 format 类似，区别只是不能包含（...）格式代码以解析嵌套的序列。kwlist 是一个以 NULL 哨兵终止的 char*数组，其中包含所有参数的名称，一个接着一个放置。例如，下面的 C 代码：
	`static PyObject*`
	`func_c(PyObject* self, PyObject* args, PyObject* kwds)`
	`{`
	`  static char* argnames[] = {"x", "y", "z", NULL};`

PyArg_ParseTupleAndKeywords

```
    double x, y=0.0, z=0.0;
    if(!PyArg_ParseTupleAndKeywords(
        args,kwds,"d|dd",argnames,&x,&y,&z))
        return NULL;
    /* 省略了函数的其余部分*/
```

大致上等于下面的 Python 代码：

```
def func_py(x, y=0.0, z=0.0):
    x, y, z = map(float, (x,y,z))
    # 省略了函数的其余部分
```

创建 Python 值

与 Python 通信的 C 函数必须经常创建 Python 值，可以返回为其 PyObject*结果和用于其他目的，比如设置项目和属性。用来创建 Python 值的最简单和最便捷的方法通常就是使用 Py_BuildValue 函数。

表 25-4

_BuildValue

`PyObject* Py_BuildValue(char* format,...)`

format 是一个 C 语言字符串，用来描述要创建的 Python 对象。Py_BuildValue 的以下参数都是 C 语言值，返回的结果就是从这些值创建的。PyObject*结果是一个新引用。下表列出了常用的代码字符串，其中的零个或多个字符串将被连接成字符串 format。如果 format 包含两个或更多格式代码，或者如果 format 以(开始，以)结束，Py_BuildValue 将创建并返回一个元组。否则，结果不是一个元组。在以——例如，格式代码 s#——的形式传递缓冲区时，Py_BuildValue 将复制该数据。因此，在 Py_BuildValue 返回之后，开发者可以修改、放弃或 free( )该数据最初的副本。Py_BuildValue 总是会返回一个新引用（除了格式代码 N）。在使用一个空白 format 调用 Py_BuildValue 时，Py_BuildValue("")将返回一个到 None 的新引用。

Py_BuildValue 的格式代码

代码	C 语言类型	含　义
c	char	C 语言中的一个 Char 字符变成一个长度为 1 的 Python 字符串
D	double	C 语言中的 double 变成 Python 中的 float
d	Py_Complex	C 语言中的 Py_Complex 变成 Python 中的 complex
i	int	C 语言中的 int 变成 Python 中的 int
l	long	C 语言中的 long 变成 Python 中的 int
N	PyObject*	传递一个 Python 对象，并偷用一个引用

续表

	代码	C 语言类型	含　义
_BuildValue	O	PyObject*	传递一个 Python 对象，并 INCREF 其为正常值
	O&	convert+void*	任意转换（见下文）
	s	char*	将 C 语言中以 0 结束的 char*转换为 Python 字符串，或者将 NULL 转换为 None
	s#	char*+int	将 C 语言的 char*和长度转换为 Python 字符串，或者将 NULL 转换为 None
	u	Py_UNICODE*	将 C 语言的以 NULL 结尾的宽字符串转换为 Python Unicode，或者将 NULL 转换为 None
	u#	Py_UNICODE*+int	将 C 语言宽字符串和长度转换为 Python Unicode，或者将 NULL 转换为 None
	(...)	As per ...	从 C 语言值创建 Python 元组
	[...]	As per ...	从 C 语言值创建 Python 列表
	{...}	As per ...	从 C 语言值创建 Python 字典，交换键和值（必须是偶数个 C 语言值）

代码 O&对应于这些变量参数中的两个参数：第一个是开发者编码的转换器函数的地址，第二个是一个 void*（也就是，任意地址）。转换器函数必须具有签名 PyObject* convert(void*)。Python 将使用变量参数中的 void*作为唯一参数调用该转换函数。转换函数必须返回 NULL 并引发一个异常（参见第 25.1 节）以指示错误，或者返回一个新引用 PyObject*，该引用是从通过 void*获得的数据创建的。

代码{...}将从偶数个 C 语言值创建字典，键和值将交替出现。例如，Py_BuildValue ("{issi}",23,"zig","zag",42)将返回一个类似于 Python 的{23:'zig','zag':42}的字典。

请注意代码 N 和 O 之间的重要区别。N 将从变量参数中的对应 PyObject*值偷取一个引用，因此，使用 N 便于创建一个包含开发者自己的引用的对象，否则，必须使用 Py_DECREF。O 不会偷取引用，因此适合于创建一个包含开发者自己的引用的对象，或者包含一个还必须保存在其他位置的引用。

异常

要想传播从开发者调用的其他函数引发的异常，可以返回 NULL 作为从开发者的 C 函数返回的 PyObject*结果。要想引发开发者自己的异常，可以设置当前异常指示器并返回 NULL。Python 的内置异常类(参见第 6.4 节)都是全局可用的，其名称以 PyExc_开始，比如 PyExc_AttributeError、PyExc_KeyError 等。开发者的扩展模块还可以提供并使用其自己的异常类。与引发异常有关的最常使用的 C API 函数如下。

表 25-5

PyErr_Format	`PyObject* PyErr_Format(PyObject* type,char* format,...)`
	引发一个类为 type 的异常，该异常必须是一个内置对象，比如 PxExc_IndexError，或者是一个使用 PyErr_NewException 创建的异常类。从格式字符串 format 创建相关的值，该字符串具有类似于 printf 的语法，后面的 C 语言值是通过（...）变量参数来指示的。这个函数返回 NULL，因此开发者的代码可以像下面这样调用该函数：
	`return PyErr_Format(PyExc_KeyError,`
	`"Unknown key name (%s)", thekeystring);`

PyErr_NewException	`PyObject* PyErr_NewException(char* name,PyObject* base,PyObject* dict)`
	异常类 base 的子类，可以包含来自于字典 dict 的额外类属性和方法（通常为 NULL，表示没有额外的类属性或方法），创建一个名为 name 的新异常类（name 字符串必须是“modulename.classname”的形式），并返回一个到新类对象的新引用。在 base 为 NULL 时，可以使用 PyExc_Exception 作为基类。开发者通常在模块对象 module 的初始化期间调用这个函数。例如：
	`PyModule_AddObject(module, "error",`
	`PyErr_NewException("mymod.error", NULL, NULL));`

PyErr_NoMemory	`PyObject* PyErr_NoMemory()`
	引发一个内存耗尽错误，并返回 NULL，因此开发者的代码可以直接调用：
	`return PyErr_NoMemory();`

PyErr_SetObject	`void PyErr_SetObject(PyObject* type, PyObject* value)` 引发一个类为 type 的异常，这个类必须是一个内置对象，比如 PyExc_KeyError，或者使用 PyErr_NewException 创建的异常类，并使用 value 作为关联值（借用引用）。PyErr_SetObject 是一个 void 函数（也就是，不返回任何值）。

PyErr_SetFromErrno	`PyObject* PyErr_SetFromErrno(PyObject* type)`
	引发一个类为 type 的异常，该异常必须是一个内置对象，比如 PyExc_OSError，或者是一个使用 PyErr_NewException 创建的异常类。这个异常对象可以从全局变量 errno 获取所有详细信息，errno 是 C 标准库函数和系统调用为许多错误情况而设置的，另外，还可以从标准 C 库函数 strerror 获取所有详细信息，strerror 可以将这样的错误代码翻译为适当的字符串。这个函数返回 NULL，因此开发者的代码可以像下面这样调用：
	`return PyErr_SetFromErrno(PyExc_IOError);`

续表

PyErr_SetFromErrnoWithFilename	PyObject* PyErr_SetFromErrnoWithFilename(PyObject* *type*,char* *filename*) 与 PyErr_SetFromErron 类似，但还提供了字符串 filename 作为该异常的值的一部分。在 filename 为 NULL 时，其使用就像 PyErr_SetFromErrno 一样。 开发者的 C 语言代码可能需要处理一个异常并继续执行，就像 Python 代码中的 try/except 语句实现的功能。与捕获异常有关的最常使用的 C API 函数如下。
PyErr_Clear	void PyErr_Clear() 清除错误指示器。如果没有错误正在挂起，则不执行任何操作。
PyErr_ExceptionMatches	int PyErr_ExceptionMatches(PyObject* *type*) 只有在一个错误被挂起时，或者整个程序可能会崩溃时，才调用该函数。在挂起的异常是一个具有给定 type 或给定 type 的任意子类的实例时，该函数返回一个不等于 0 的值，而在挂起的异常不是这样的一个实例时，返回 0。
PyErr_Occurred	PyObject* PyErr_Occurred() 如果没有错误正在挂起，则返回 NULL；否则，返回一个与挂起的异常具有相同类型的借用引用（作为常规和期望的情况，同时也是为了捕获子类的异常，不要使用返回的值；而是调用 PyErr_ExceptionMatches）。
PyErr_Print	void PyErr_Print() 只有在一个错误被挂起，或者整个程序可能会崩溃时调用该函数。向 sys.stderr 输出一个标准跟踪，然后清除该错误指示器。 如果开发者需要以一种高度复杂的方法执行错误，可以学习 C API 的其他与错误相关的函数，比如 PyErr_Fetch、PyErr_Normalize、PyErr_GivenExceptionMatches 和 PyErr_Restore。但是，本书没有这些虽然高级，但是极少需要使用的功能。

抽象层函数

一个 C 语言扩展的代码通常需要使用某些 Python 功能。例如，开发者的代码可能需要查看或设置 Python 对象的属性和项目，调用以 Python 编码的内置函数和方法等。在大多数情况下，最好的方法就是让开发者的代码从 Python 的 C API 的抽象层调用函数。这些都是可以对任意 Python 对象调用的函数（名称以 PyObject_开始的函数），或者对很大类别的任意对象（比如映射、数字或序列）调用的函数（其名称分别以 PyMapping_、PyNumber_和 PySequence_开始）。

某些函数可以对这些类别中特定类型的对象进行调用，这些函数可以复制 PyObject_函数中也可以使用的功能。在这些情况下，开发者必须几乎总是使用更通用的 PyObject_

函数。本书没有介绍这种几乎是多余的函数。

如果开发者对不适用的对象调用了抽象层中的函数，可能会引发 Python 异常。所有这些函数都可以接受 PyObject*参数的借用引用，如果这些函数返回一个 PyObject*结果，则返回一个新引用（对于异常，则返回 NULL）。

最常使用的抽象类函数如表 25-6 所示。

表 25-6

PyCallable_Check	`int PyCallable_Check(PyObject* x)` 如果 x 是可调用的，则返回 True，就像 Python 的 callable(x)一样。
PyEval_CallObject	`PyObject* PyEval_CallObject(PyObject* x, PyObject* args)` 使用元组 args 中保存的位置参数调用可调用 Python 对象 x。返回调用的结果，就像 Python 的 return x(*args)一样。
PyEval_CallObjectWithKeywords	`PyObject* PyEval_CallObjectWithKeywords(PyObject* x, PyObject* args, PyObject* kwds)` 使用元组 args 中保存的位置参数和字典 kwds 中保存的命名参数调用可调用 Python 对象 x。返回调用的结果，就像 Python 的 return x(*args,**kwds)一样。
PyIter_Check	`int PyIter_Check(PyObject* x)` 如果 x 支持迭代器协议（也就是，如果 x 是一个迭代器），则返回 True。
PyIter_Next	`PyObject* PyIter_Next(PyObject* x)` 返回迭代器 x 的下一个项目。如果 x 的迭代结束（也就是，在 Python 的 x.next( )引发 StopIteration 时），则返回 NULL，不引发任何异常。
PyNumber_Check	`int PyNumber_Check(PyObject* x)` 如果 x 支持数字协议（也就是，如果 x 是一个数字），则返回 True。
PyObject_CallFunction	`PyObject* PyObject_CallFunction(PyObject* x, char* format,...)` 使用格式字符串 format 描述的位置参数，并使用与 Py_BuildValue 相同的格式代码调用可调用 Python 对象 x，参见第 25.1 节。在 format 为 NULL 时，可以不带任何参数调用 x。返回调用的结果。
PyObject_CallMethod	`PyObject* PyObject_CallMethod(PyObject* x, char* method,char* format,...)` 使用格式字符串 format 描述的位置参数调用 Python 对象 x 的名为 method 的方法，并使用与 Py_BuildValue 相同的格式代码。在 format 为 NULL 时，不带任何参数调用这个方法。返回调用的结果。

续表

PyObject_Cmp	int PyObject_Cmp(PyObject* *x1*,PyObject* *x2*, int* *result*) 比较对象 x1 和 x2，并将结果（-1, 0 或 1）放到*result 中，就像 Python 的 result=cmp(x1,x2)。
PyObject_DelAttrString	int PyObject_DelAttrString(PyObject* *x*,char**name*) 删除 x 中名为 name 的属性，就像 Python 的 del x.name 一样。
PyObject_DelItem	int PyObject_DelItem(PyObject* *x*,PyObject* *key*) 删除 x 中键(或索引)为 key 的项目，就像 Python 的 del x[key]一样。
PyObject_DelItemString	int PyObject_DelItemString(PyObject* *x*,char* *key*) 删除 x 中键为 key 的项目，就像 Python 的 del x[key]一样。
PyObject_GetAttrString	PyObject* PyObject_GetAttrString(PyObject* *x*, char* *name*) 返回 x 的 name 属性，就像 Python 的 x.name 一样。
PyObject_GetItem	PyObject* PyObject_GetItem(PyObject* *x*,PyObject* *key*) 返回 x 中键（或索引）为 key 的项目，就像 Python 的 x[key]一样。
PyObject_GetItemString	int PyObject_GetItemString(PyObject* *x*,char* *key*) 返回 x 中键为 key 的项目，就像 Python 的 x[key]一样。
PyObject_GetIter	PyObject* PyObject_GetIter(PyObject* *x*) 返回 x 上的一个迭代器，就像 Python 的 iter(x)一样。
PyObject_HasAttrString	int PyObject_HasAttrString(PyObject* *x*,char* *name*) 如果 x 具有 name 属性，则返回 True，就像 Python 的 hasattr(x, name)。
PyObject_IsTrue	int PyObject_IsTrue(PyObject* *x*) 如果 x 在 Python 中为 True，则返回 True，就像 Python 的 bool(x)一样。
PyObject_Length	int PyObject_Length(PyObject* *x*) 返回 x 的长度，就像 Python 的 len(x)一样。
PyObject_Repr	PyObject* PyObject_Repr(PyObject* *x*) 返回 x 的详细字符串表示，就像 Python 的 repr(x)。

续表

PyObject_RichCompare	PyObject* PyObject_RichCompare(PyObject* *x*, PyObject* *y*,int *op*) 执行 x 和 y 之间由 op 指定的比较操作，并将返回的结果作为一个 Python 对象。op 可以是 Py_EQ、Py_NE、Py_LT、Py_LE、Py_GT 或 Py_GE，对应于 Python 的比较运算 x==y、x!=y、x<y、x<=y、x>y 或 x>=y。
PyObject_RichCompareBool	int PyObject_RichCompareBool(PyObject* *x*,PyObject* *y*,int *op*) 就像 PyObject_RichCompare 一样，但是，对于 False，返回 0，对于 True，返回 1。
PyObject_SetAttrString	int PyObject_SetAttrString(PyObject* *x*,char* *name*,PyObject* *v*) 将 x 中名为 name 的属性设置为 v，就像 Python 的 x.name=v 一样。
PyObject_SetItem	int PyObject_SetItem(PyObject* *x*,PyObject* *k*,PyObject *v) 将 x 中键（或索引）为 key 的项目设置为 v，就像 Python 中的 x[key]=v 一样。
PyObject_SetItemString	int PyObject_SetItemString(PyObject* *x*,char* *key*,PyObject *v) 将 x 中键为 key 的项目设置为 v，就像 Python 的 x[key]=v 一样。
PyObject_Str	PyObject* PyObject_Str(PyObject* *x*) 返回 x 的可读字符串形式，就像 Python 的 str(x)一样。
PyObject_Type	PyObject* PyObject_Type(PyObject* *x*) 返回 x 的类型对象，就像 Python 的 type(x)一样。
PyObject_Unicode	PyObject* PyObject_Unicode(PyObject* *x*) 返回 x 的 Unicode 字符串形式，就像 Python 的 unicode(x)。
PySequence_Contains	int PySequence_Contains(PyObject* *x*,PyObject* *v*) 如果 v 是 x 中的一个项目，则返回 True，就像 Python 的 v in x 一样。
PySequence_DelSlice	int PySequence_DelSlice(PyObject* *x*,int *start*,int *stop*) 删除 x 中从 start 到 stop 的切片，就像 Python 的 del x[start:stop]一样。
PySequence_Fast	PyObject* PySequence_Fast(PyObject* *x*) 返回一个到元组的新引用，该元组具有与 x 相同的项目，除非 x 是一个列表，在 x 是列表的情况下将返回一个到 x 的新引用。在开发者需要获得任意序列 x 中的多个项目时，最快的方法是调用一次 t=PySequence_Fast(x)，然后根据需要，多次调用 PySequence_Fast_GET_ITEM(t,i)，最后调用 Py_DECREF(t)。

续表

<table>
<tr><td>PySequence_Fast_GET_ITEM</td><td>PyObject* PySequence_Fast_GET_ITEM(PyObject* x,int i)
返回 x 中的第 i 个项目，x 必须是 PySequence_Fast、x!=NULL 和 0<=i<PySequence_Fast_GET_SIZE(t)这样的结果。违反这些条件将导致程序崩溃。这种方法对速度，而不是安全性进行了最优化。</td></tr>
<tr><td>PySequence_Fast_GET_SIZE</td><td>int PySequence_Fast_GET_SIZE(PyObject* x)
返回 x 的长度。x 必须是 PySequence_fast 或 x!=NULL 这样的结果。</td></tr>
<tr><td>PySequence_GetSlice</td><td>PyObject* PySequence_GetSlice(PyObject* x,int start,int stop)
返回 x 中从 start 到 stop 的切片，就像 Python 的 x[start:stop]。</td></tr>
<tr><td>PySequence_List</td><td>PyObject* PySequence_List(PyObject* x)
返回一个新列表对象，具有与 x 相同的项目，就像 Python 的 list(x)一样。</td></tr>
<tr><td>PySequence_SetSlice</td><td>int PySequence_SetSlice(PyObject* x,int start,int stop,PyObject* v)
将 x 中从 start 到 stop 的切片设置为 v，就像 Python 的 x[start:stop]=v 一样。就像在与该函数对等的 Python 语句中一样，v 也必须是一个序列。</td></tr>
<tr><td>PySequence_Tuple</td><td>PyObject* PySequence_Tuple(PyObject* x)
返回一个到元组的新引用，该元组具有与 x 相同的项目，就像 Python 的 tuple(x)一样。
其他的名称以 PyNumber_开始的函数可以用来执行数值运算。下表列出了一元 PyNumber 函数及其对等的 Python 函数，一元 PyNumber 函数需要用到一个参数 PyObject* x，并返回一个 PyObject*值。
一元 PyNumber 函数
<table>
<tr><th>函　　数</th><th>对等的 Python 函数</th></tr>
<tr><td>PyNumber_Absolute</td><td>abs(x)</td></tr>
<tr><td>PyNumber_Float</td><td>float(x)</td></tr>
<tr><td>PyNumber_Int</td><td>int(x)</td></tr>
<tr><td>PyNumber_Invert</td><td>~x</td></tr>
<tr><td>PyNumber_Long</td><td>long(x)</td></tr>
<tr><td>PyNumber_Negative</td><td>-x</td></tr>
<tr><td>PyNumber_Positive</td><td>+x</td></tr>
</table></td></tr>
</table>

表 25-7 同样列出了二元 PyNumber 函数，这些函数需要用到两个 PyObject* 参数 x 和 y，

并返回一个 PyObject*值。

表 25-7　二元 PyNumber 函数

函　　数	对等的 Python 函数
PyNumber_Add	x + y
PyNumber_And	x & y
PyNumber_Divide	x / y
PyNumber_Divmod	divmod(x, y)
PyNumber_FloorDivide	x // y
PyNumber_Lshift	x << y
PyNumber_Multiply	x * y
PyNumber_Or	x | y
PyNumber_Remainder	x % y
PyNumber_Rshift	x >> y
PyNumber_Subtract	x – y
PyNumber_TrueDivide	x / y (非截短)
PyNumber_Xor	x ^ y

所有二元 PyNumber 函数都有名称以 PyNumber_InPlace 开始的原地对等函数，比如 PyNumber_InPlaceAdd 等。如有可能，这些原地函数版本将尝试在原地修改第一个参数，并且在任何情况下都将返回一个到结果的新引用，不管是第一个参数（被修改过）还是一个新对象。Python 的内置数字都是不可变的；因此，在第一个参数是一个内置类型的数字时，这个原地函数版本的操作与二元函数版本是完全相同的。PyNumber_Divmod 函数将返回一个由两个项目（商和余数）组成的元组，但是该函数没有对等的原地函数。

此外，还有一个三元 PyNumber 函数 PyNumber_Power。

表 25-8

PyNumber_Power	`PyObject* PyNumber_Power(PyObject* x, PyObject* y, PyObject* z)` 在 z 是 Py_None 时，将返回 x 的 y 次方，就像 Python 的 x**y，或者 pow(x, y)一样。否则，返回 x**y%z，就像 Python 的 pow(x, y, z)一样。其原地函数版本名为 PyNumber_InPlacePower。

具体层函数

每个特殊类型的 Python 内置对象都提供了一些具体函数，这些函数可以对这种类型的

实例进行操作，其名称以 Pytype_开始（例如，PyInt_表示 Python 的整型 int 相关的函数）。大多数这样的函数都复制了抽象层函数或本章前面介绍过的辅助函数的功能，比如 Py_BuildValue，该函数可以生成许多类型的对象。本节将介绍来自具体层的一些经常使用的函数，这些函数提供了唯一的功能，或者很好的便利性或速度。对于大多数类型，开发者可以通过调用 Pytype_Check 或者 Pytype_CheckExact 检查一个对象是否属于这个类型，Pytype_Check 函数还可以接受子类型的实例，而 Pytype_CheckExact 函数只能接受 type 实例，而不能接受子类型实例。这两个函数的签名与 PyIter_Check 函数的签名相同，参见第 25.1 节中介绍的 PyIter_Check 函数。

表 25-9

PyDict_GetItem	`PyObject* PyDict_GetItem(PyObject* x,PyObject* key)` 返回一个到字典 x 中键为 key 的项目的借用引用。
PyDict_GetItemString	`int PyDict_GetItemString(PyObject* x,char* key)` 返回一个到字典 x 中键为 key 的项目字符串的借用引用。
PyDict_Next	`int PyDict_Next(PyObject* x,int* pos,PyObject ** k,PyObject** v)` 对字典 x 中的项目进行迭代。开发者必须在迭代操作开始的时候将*pos 初始化为 0；PyDict_Next 使用并更新*pos 以跟踪其位置。对于每个成功的迭代步骤，返回 1；在没有项目时，返回 0。对于每个返回 1 的步骤，分别更新*k 和*v 以指向下一个键和值（借用引用）。如果开发者对这个键或值不感兴趣，可以将 k 或 v 的值传递为 NULL。在迭代期间，开发者不能以任何方式更改由 x 的键组成的集合，但是，只要键的集合保持相等，则开发者可以更改 x 的值。
PyDict_Merge	`int PyDict_Merge(PyObject* x,PyObject* y,int override)` 通过将字典 y 中的项目合并到字典 x 中来更新字典 x。override 确定了在字典 x 和 y 中都有键 k 时执行什么操作：如果 override 为 0，则 x[k]保持不变；否则，x[k]将被替换为 y[k]的值。
PyDict_MergeFromSeq2	`int PyDict_MergeFromSeq2(PyObject* x,PyObject* y,int override)` 类似于 PyDict_Merge，区别在于 y 不是一个字典，而是一个由序列组成的序列，其中每个子序列的长度为 2，每个子序列被用作一个（key, value）对。
PyFloat_AS_DOUBLE	`double PyFloat_AS_DOUBLE(PyObject* x)` 返回 Python 浮点型值 x 的 C 双精度值，非常快，不执行任何错误检查操作。
PyList_New	`PyObject* PyList_New(int length)` 返回一个给定长度 length 的新的、未被初始化的列表。然后，通常必须通过 length 次调用 PyList_SET_ITEM 来初始化该列表。

续表

PyList_GET_ITEM	PyObject* PyList_GET_ITEM(PyObject* *x*,int *pos*) 返回列表 x 的第 pos 个项目，不执行任何错误检查。
PyList_SET_ITEM	int PyList_SET_ITEM(PyObject* *x*,int *pos*,PyObject* *v*) 将列表 x 的第 pos 个项目设置为 v，不执行任何错误检查。偷取一个到 v 的引用。只能在使用 PyList_New 创建一个新列表 x 之后立即使用该函数。
PyString_AS_STRING	char* PyString_AS_STRING(PyObject* *x*) 返回一个指向字符串 x 的内部缓冲区的指针，非常快，不执行任何错误检查。开发者不能以任何方式修改该缓冲区，除非开发者刚通过调用 PyString_FromStringAnd-Size(NULL,size)分配了该缓冲区。
PyString_AsStringAndSize	int PyString_AsStringAndSize(PyObject* *x*,char** *buffer*,int* *length*) 将一个指向字符串 x 的内部缓冲区的指针放到*buffer 中，并将 x 的长度放到*length 中。开发者不能以任何方式修改该缓冲区，除非刚通过调用 PyString_FromStringAnd-Size(NULL,size)分配了该缓冲区。
PyString_FromFormat	PyObject* PyString_FromFormat(char* *format*,...) 返回一个从格式字符串 format 创建的 Python 字符串，该格式字符串具有类似于 printf 的语法，并且，后面的 C 语言值是通过上面的变量参数（...）指定的。
PyString_FromStringAnd-Size	PyObject* PyString_FromFormat(char* *data*,int *size*) 返回一个长度为 size 的 Python 字符串，从 data 复制 size 个字节。在 data 为 NULL 时，Python 字符串是未初始化的，开发者必须初始化该字符串。开发者可以通过调用 PyString_AS_STRING 获得指向该字符串的内部缓冲区的指针。
PyTuple_New	PyObject* PyTuple_New(int *length*) 返回一个给定长度的新的，未初始化的元组。然后，开发者必须初始化该元组，通常是通过调用 length 次 PyTuple_SET_ITEM 来实现的。
PyTuple_GET_ITEM	PyObject* PyTuple_GET_ITEM(PyObject* *x*,int *pos*) 返回元组 x 中的第 pos 个项目，不执行错误检查。
PyTuple_SET_ITEM	int PyTuple_SET_ITEM(PyObject* *x*,int *pos*,PyObject* *v*) 将元组 x 中的第 pos 个项目设置为 v，不执行错误检查。偷取一个到 v 的引用。只能在使用 PyTuple_New 创建一个新元组 x 之后立即使用。

一个简单的扩展示例

示例 25-1 显示了 Python C API 函数 PyDict_Merge 和 PyDict_MergeFromSeq2 用于 Python 的功能。字典的 update 方法的使用就像使用 override=1 的 PyDict_Merge 一样，但是示例 25-1 更通用。

示例 25-1　一个简单的 Python 扩展模块 merge.c

```
#include <Python.h>

static PyObject*
merge(PyObject* self, PyObject* args, PyObject* kwds)
{
    static char* argnames[] = {"x","y","override",NULL};
    PyObject *x, *y;
    int override = 0;
    if(!PyArg_ParseTupleAndKeywords(args, kwds, "O!O|i", argnames,
        &PyDict_Type, &x, &y, &override))
            return NULL;
    if(-1 == PyDict_Merge(x, y, override)) {
        if(!PyErr_ExceptionMatches(PyExc_TypeError))
            return NULL;
        PyErr_Clear( );
        if(-1 == PyDict_MergeFromSeq2(x, y, override))
            return NULL;
    }
    return Py_BuildValue("");
}

static char merge_docs[] = "\
merge(x,y,override=False): merge into dict x the items of dict y (or the
pairs\n\
    that are the items of y, if y is a sequence), with optional override.\n\
    Alters dict x directly, returns None.\n\
";

static PyObject*
mergenew(PyObject* self, PyObject* args, PyObject* kwds)
{
    static char* argnames[] = {"x","y","override",NULL};
    PyObject *x, *y, *result;
    int override = 0;
    if(!PyArg_ParseTupleAndKeywords(args, kwds, "O!O|i", argnames,
        &PyDict_Type, &x, &y, &override))
            return NULL;
    result = PyObject_CallMethod(x, "copy", "");
    if(!result)
        return NULL;
    if(-1 == PyDict_Merge(result, y, override)) {
```

```
        if(!PyErr_ExceptionMatches(PyExc_TypeError))
            return NULL;
        PyErr_Clear();
        if(-1 == PyDict_MergeFromSeq2(result, y, override))
            return NULL;
    }
    return result;
}

static char mergenew_docs[] = "\
mergenew(x,y,override=False): merge into dict x the items of dict y (or\n\
    the pairs that are the items of y, if y is a sequence), with optional\n\
    override.  Does NOT alter x, but rather returns the modified copy as\n\
    the function's result.\n\
";

static PyMethodDef funcs[] = {
    {"merge", (PyCFunction)merge, METH_KEYWORDS, merge_docs},
    {"mergenew", (PyCFunction)mergenew, METH_KEYWORDS, mergenew_docs},
    {NULL}
};

void
initmerge(void)
{
    Py_InitModule3("merge", funcs, "Example extension module");
}
```

这个示例将 C 源代码文件中除了 initmerge 之外的所有函数和全局变量声明为 static 的，initmerge 必须从该扩展模块外部可见，因此 Python 可以调用该函数。因为函数和变量都是通过 PyMethodDef 数据结构传递给 Python 的，Python 不需要直接使用这些函数和变量的名称。因此，将其声明为 static 是最好的：这样可以确保这些名称不会偶然在整个程序的全局命名空间中终止，否则在某些平台上是可能会发生这种情况的，并可能导致冲突和错误。

传递给 PyArg_ParseTupleAndKeywords 的格式字符串“"O!O|i”表示 merge 函数可以从 Python 接受 3 个参数：一个具有类型约束的对象、一个通用对象，以及一个可选的整数。同时，格式字符串表示 PyArg_ParseTupleAndKeywords 的参数中的变量部分必须包含按以下顺序排列的 4 个地址：Python 类型对象的地址、两个 PyObject*变量的地址和一个 int 变量的地址。因为对应的 Python 参数是可选的，因此必须在以前就将 int 变量初始化为打算使用的默认值。

实际上，在 argnames 参数之后，代码传递了&PyDict_Type（也就是，字典类型对象的地址）。然后传递了两个 PyObject*变量的地址。最后，传递了 override 变量的地址，override 是一个以前就被初始化为 0 的值，因为，在 override 参数不是从 Python 显式传递得到时，默认值为“不覆盖”。如果 PyArg_ParseTupleAndKeywords 的返回值为 0，则代码将立即返回 NULL 以传播该异常；这样做可以自动诊断大多数 Python 代码将错

误的参数传递到新函数 merge 的情况。

在参数看起来已经就绪时，代码将尝试 PyDict_Merge 函数，如果 y 是一个字典，则该函数将成功执行。在 PyDict_Merge 引发一个 TypeError 错误时，表示 y 不是一个字典，代码将清除这个错误并再次尝试，再次尝试时将使用 PyDict_MergeFromSeq2 函数，该函数在 y 是一个由数据对组成的序列时将成功执行。如果这样做还是失败，将返回 NULL 以传播该异常。否则，将以最简单的方法返回 None（也就是，使用 return Py_BuildValue("")）以指示成功。

mergenew 函数基本上复制了 merge 的功能；但是，mergenew 没有改变其参数，但是创建并返回了一个新字典作为该函数的结果。CAPI 函数 PyObject_CallMethod 可以让 mergenew 调用该函数的第一个由 Python 传递的参数（一个字典对象）的 copy 方法，并获得一个后来改变的新字典对象（使用与 merge 函数完全相同的逻辑）。然后，代码将返回这个被改变的字典作为函数的结果（因此，在这种情况下不需要调用 Py_BuildValue）。

示例 25-1 的代码必须被保存在名为 merge.c 的源文件中。在相同的目录下，创建下面这个名为 setup.py 的脚本：

```
from distutils.core import setup, Extension
setup(name='merge', ext_
modules=[ Extension('merge',sources=['merge.c']) ])
```

现在，在这个目录的 shell 提示符下运行 python setup.py（使用一个具有适当权限的用户 ID 以将这个模块写入到开发者的 Python 安装程序中，在类-Unix 系统上，如有必要，需要使用 sudo）。这个命令将为 merge 扩展模块创建动态加载库，并根据开发者的 Python 安装，将其复制到适当的目录中。现在，开发者的 Python 代码就可以使用这个模块了。例如：

```
import merge
x = {'a':1,'b':2 }
merge.merge(x,[['b',3],['c',4]])
print x                              # 打印: {'a':1, 'b':2, 'c':4 }
print merge.mergenew(x,{'a':5,'d':6},override=1)
# 打印: {'a':5, 'b':2, 'c':4, 'd':6 }
print x                              # 打印: {'a':1, 'b':2, 'c':4 }
```

这个示例显示了 merge（改变了第一个参数）和 mergenew（返回一个新对象，不改变其参数）之间的区别。该示例还显示第二个参数可以是一个字典或者一个由包含两个项目的子序列组成的序列。此外，该示例还举例介绍了默认操作（不管已经在第一个参数中的键是什么），以及 override 选项（其中来自于第二个参数的键被优先得到，就像在 Python 字典的 update 方法中一样）。

定义新类型

在开发者的扩展模块中，开发者通常想要定义新类型，并让类型可以在 Python 中使用。

类型的定义被保存在一个名为 PyTypeObject 的大型数据结构中。PyTypeObject 的大多数字段都是指向函数的指针。某些字段指向其他数据结构，这些数据结构依次为指向函数的指针组成的数据块。PyTypeObject 还包含一些字段，这些字段给出了类型的名称、大小和行为细节（选项标记）。如果开发者不想提供相关功能，可以让 PyTypeObject 的几乎所有字段都设置为 NULL。为了以标准方法提供一些特定的基础对象功能，开发者可以将某些字段指向 Python C API 中的函数。

实现一个类型的最好方法就是从 Python 源代码中复制文件 Modules/xxsubtype.c，并对其进行编辑，Python 提供这个文件就是为了这样的教学目的。这是一个包含两个类型的完整模块，这两个类型分别是从 list 和 dict 继承的子类。这个 Python 源代码中的另一个示例 Objects/xxobject.c 并不是一个完整模块，并且这个文件中的类型是最小和过时的，没有使用现代推荐的一些方法。参见 http://www.python.org/dev/doc/devel/api/type-structs.html 以了解有关 PyTypeObject 和其他相关数据结构的详细文档。Python 源文件中的 Include/object.h 文件包含这些类型的声明，以及开发者需要好好学习的几个重要注释。

按实例数据

要想表示开发者类型的每个实例，可以在左花括号的后面声明一个以宏 PyObject_HEAD 开始的 C 语言数据结构。为了成为一个 Python 对象，这个宏将展开到开发者的数据结构开始的数据字段中。这些字段包括引用计数和一个指向该实例的类型的指针。任何指向开发者的数据结构的指针都可以被正确地转换为一个 PyObject*。开发者可以选择将这种实际操作看作是一种 C 语言级别继承机制的实现。

定义了开发者的类型的特征和行为的 PyTypeObject 数据结构必须包含开发者的按实例（per-instance）数据结构，以及到开发者编写的对该数据结构进行操作的 C 语言函数的指针。因此，开发者通常要将 PyTypeObject 放置到 C 语言编码的模块的源代码的末尾，这些源代码在按实例数据结构的定义之后，也就是对按实例数据机构的实例进行操作的所有函数的源代码。每个指向一个以 PyObject_HEAD 开始的数据结构 x（特别是每个 PyObject* x）都有一个字段 x->ob_type，这个字段是作为 x 的 Python 类型对象的 PyTypeObject 数据结构的地址。

PyTypeObject 的定义

给定一个下面这样的按实例数据结构：

```
typedef struct {
    PyObject_HEAD
    /* 此处忽略这种类型的实例所需的其他数据*/
} mytype;
```

对应的 PyTypeObject 数据结构几乎总是以类似于下面这样的方式开始：

```
static PyTypeObject t_mytype = {
/* tp_head */        PyObject_HEAD_INIT(NULL)  /*对于 MSVC++，使用 NULL，*/
```

```
/* tp_internal */    0,                                  /* 必须为 0 */
/* tp_name */        "mymodule.mytype",                  /* 类型名称，包括模块 */
/* tp_basicsize */   sizeof(mytype),
/* tp_itemsize */    0,                                  /* 除了可变大小的类型,都为 0 */
/* tp_dealloc */     (destructor)mytype_dealloc,
/* tp_print */       0,                                  /* 通常为 0，否则使用 str */
/* tp_getattr */     0,                                  /* 通常为 0 (参见getattro) */
/* tp_setattr */     0,                                  /* 通常为 0 (参见setattro) */
/* tp_compare*/      0,                                  /* 参见 richcompare */
/* tp_repr */     (reprfunc)mytype_str,                  /* 类似于 Python 的__repr__ */
   /* 此处省略数据结构的其余部分 */
```

为了可以移植到 Microsoft Visual C++，位于 PyTypeObject 的起始位置的 PyObject_HEAD_INIT 宏必须包含一个值为 NULL 的参数。在模块初始化期间，开发者必须在其他任务中调用 PyType_Ready(&t_mytype)以将 t_mytype 类型的地址插入到 t_mytype 中（一个类型的类型也被称为元类型（metatype）），t_mytype 类型的地址通常是&PyType_Type。在指向另一个类型对象的 PyTypeObject 中的另一个槽是 tp_base，在数据结构的后面。在数据结构定义本身，必须有一个值为 NULL 的 tp_base，这仍然是为了与 Microsoft Visual C++兼容。但是，在开发者调用 PyType_Ready(&t_mytype)之前，可以有选择地将 t_mytype.tp_base 设置为另一个类型对象的地址。在这样做时，开发者的类型将从其他类型继承，就像一个以 Python 编码的类可以有选择地从一个内置类型继承一样。对于一个使用 C 语言编码的 Python 类型，继承意味着对于 PyTypeObject 中的大多数字段，如果将该字段设置为 NULL，则 PyType_Ready 将从其基类型复制相应的字段。一个类型必须在其 tp_flags 字段中特别声明该字段可以用作一个基类型，否则不能从这个基类型继承任何其他类型。

tp_itemsize 只与具有不同大小的实例的类型有关，比如元组，并且可以在创建时一次确定实例的大小并永久保持。大多数类型只是简单地将 tp_itemsize 设置为 0。tp_getattr 和 tp_setattr 字段通常被设置为 NULL，因为这两个字段的存在只是为了向后兼容；而现在的类型使用的是 tp_getattro 和 tp_setattro 字段。在后面省略的大多数字段中，tp_repr 字段是很典型的：这个字段保存了一个函数的地址，这个地址直接对应于一个 Python 特殊方法（在这里就是__repr__）。开发者可以将这个字段设置为 NULL，表示开发者的类型不支持这个特殊方法，或者也可以将这个字段设置为指向一个具有所需功能的函数。如果开发者不但将这个字段设置为 NULL，而且指向一个来自于 tp_base 槽的基类型，则可以从开发者的基类型继承这个特殊方法（如果有的话）。开发者通常需要将函数转换为一个字段需要的特殊 typedef 类型（这里，也就是 tp_repr 的 reprfunc 类型），因为 typedef 的第一个参数是 PyObject* self，而开发者的函数专门用于开发者的类型，通常需要使用更特殊的指针。例如：

```
static PyObject* mytype_str(mytype* self) {.../* 其余的部分被省略 */
```

另外，开发者还可以使用一个 PyObject* self 来声明 mytype_str，然后在该函数体中使

用一个强制转换(mytype*)self。这两种用法都是可以接受的风格，但是更通用的方法是在 PyTypeObject 声明中放置强制转换操作。

实例初始化和终止化

终止化实例的任务被分成了两个函数。除了恒定不变的类型（也就是，其实例永远都不被取消分配的类型）之外，tp_dealloc 槽一定不能为 NULL。Python 将对每个引用计数减少到 0 的实例 x 调用 x->ob_type->tp_dealloc(x)，并且被调用的函数必须释放对象 x 占用的所有资源，包括 x 的内存。在一个 mytype 实例没有保存其他必须被释放的资源时（特别是，没有到开发者可能必须 DECREF 的其他 Python 对象的被占用的引用），mytype 的析构函数可以极其简单：

```
static void mytype_dealloc(PyObject *x)
{
    x->ob_type->tp_free((PyObject*)x);
}
```

tp_free 槽中的函数包含释放 x 的内存的特殊任务。通常，开发者只需要将 C API 函数 _PyObject_Del 的地址放到 tp_free 槽中。

初始化开发者的实例的任务被分成 3 个函数。要想为开发者的类型的实例分配内存，可以将 C API 函数 PyType_GenericAlloc 放到槽 tp_alloc 中，这个函数可以执行最小化的初始化，将除了类型指针和引用计数之外新分配的内存字节清除为 0。类似地，开发者通常可以将 tp_new 字段设置为 C API 函数 PyType_GenericNew。在这种情况下，开发者可以在 tp_init 槽中放置的函数中执行所有的按实例初始化，tp_init 槽具有以下签名：

```
int init_name(PyObject *self,PyObject *args,PyObject *kwds)
```

tp_init 槽中的函数的位置和命名参数都是在调用该类型以创建新实例的时候传递的，就像在 Python 中那样，__init__的位置和命名参数都是在调用类对象的时候传递的。此外，对于 Python 中定义的类型（类），一般的原则是在 tp_new 中执行尽可能少的初始化，而在 tp_init 中执行尽可能多的初始化。使用 tp_new 的 PyType_GernericNew 来完成这样的操作。但是，开发者可以选择为特殊类型定义开发者自己的 tp_new，比如具有不可变实例的类型，其中必须更早进行初始化。其签名为：

```
PyObject* new_name(PyObject *subtype,PyObject *args,PyObject *kwds)
```

tp_new 中的函数必须返回新创建的实例，通常是一个 subtype 实例（subtype 可以是一个从开发者自己的类型继承的类型）。另一方面，tp_init 中的函数必须返回 0 以表示成功，或者返回-1 以表示异常。

如果开发者的类型是可以继承子类的，则重要的是要在 tp_new 中的函数返回之前建立所有不变的实例。例如，如果必须保证实例的某个特定字段永远都不为 NULL，则必须使用 tp_new 中的函数将这个字段设置为非-NULL 值。开发者的类型的子类型可能会在调用 tp_init 函数时失败；因此，这些必不可少的初始化操作是建立不变量所必需的，

必须总是在可以继承子类的类型的 tp_new 中。

属性访问

访问开发者的实例(包括方法)的属性(参见第 5.1 节)是通过放在开发者的 PyTypeObject 数据结构的 tp_getattro 和 tp_setattro 槽中的函数来实现的。通常，开发者可以将标准 C API 函数 PyObject_GenericGetAttr 和 PyObject_GenericSetAttr 放在那里，这两个函数实现了标准的语义。特别是，这些 API 函数可以通过将 tp_methods 槽指向一个由 PyMethodDef 数据结构组成的以哨兵终止的数组以访问开发者的类型的方法，并且通过 tp_members 槽访问开发者的实例的成员，tp_members 槽是一个类似于 PyMemberDef 数据结构的以哨兵终止的数组：

```
typedef struct {
    char* name;          /* Python 可见的成员名称 */
    int type;            /* 用来定义成员的数据类型的代码 */
    int offset;          /* 成员在按实例数据结构中的偏移量 */
    int flags;           /* READONLY 表示只读成员 */
    char* doc;           /* 成员的文档字符串 */
} PyMemberDef
```

包含 Python.h 就可以得到所需的所有声明是一个一般原则，作为这个一般原则的一个例外，为了让开发者的 C 语言源代码看到 PyMemberDef 的声明，必须显式包含 structmember.h。

对于 PyObject*成员，type 通常是 T_OBJECT，但是，对于开发者的实例保存为 C 语言数据的成员（例如 T_DOUBLE 表示 double，或者 T_STRING 表示 char*），Include/structmember.h 中定义了许多其他类型代码。例如，假定开发者定义了下面这样的按实例数据结构：

```
typedef struct {
    PyObject_HEAD
    double datum;
    char* name;
} mytype;
```

通过定义下面的数组并将开发者的 PyTypeObject 的 tp_members 指向该数组，可以显示 Python 的按实例属性 datum（读/写）和 name（只读）：

```
static PyMemberDef[] mytype_members = {
    {"datum", T_DOUBLE, offsetof(mytype, datum), 0, "The current datum"},
    {"name", T_STRING, offsetof(mytype, name), READONLY, "Datum name"},
    {NULL}
};
```

使用 tp_getattro 和 tp_setattro 的 PyObject_GenericGetAttr 和 PyObject_GenericSetAttr 属性还提供了更多的功能，不过本书没有对此进行详细说明。tp_getset 字段指向 PyGetSetDef 数据结构的一个以哨兵终止的数组，相当于在 Python 编码的类中包含

property 实例。如果开发者的 PyTypeObject 的 tp_dictoffset 字段不等于 0，则该字段的值必须是指向一个 Python 字典的 PyObject*在按实例数据结构中的偏移量。在这种情况下，通用的属性访问 API 函数将使用这个字典以允许 Python 代码对开发者的类型的实例设置任意属性，就像对 Python 编码的类的实例所作的那样。

另一个字典是按类型的，而不是按实例的；按类型字典的 PyObject*是 PyTypeObject 数据结构的 tp_dict 槽。开发者可以将 tp_dict 槽设置为 NULL，然后，PyType_Ready 将正确地初始化这个字典。此外，开发者还可以将 tp_dict 设置为一个由类型属性组成的字典，然后，除了开发者设置的类型属性之外，PyType_Ready 还将把其他条目添加到相同的字典中。通常，以将 tp_dict 槽设置为 NULL 开始操作要更容易一些，调用 PyType_Ready 以创建并初始化按类型字典，然后，如有需要，可以通过显式 C 代码将任何其他条目添加到这个字典中。

tp_flags 字段是一个长整型（long），该字段的比特确定了类型数据结构的确切布局，主要是为了向后兼容。通常，可以将这个字段设置为 Py_TPFLAGS_DEFAULT 以指示开发者定义了一个正规的现代类型。如果开发者的类型支持循环垃圾收集，则必须将 tp_flags 设置为 Py_TPFLAGS_DEFAULT|Py_TPFLAGS_HAVE_GC。如果该类型的实例包含可能指向任意对象的 PyObject*字段并成为引用循环的一部分，则开发者的类型必须支持循环垃圾收集。但是，要想支持循环垃圾收集，只是将 Py_TPFLAGS_HAVE_GC 添加到 tp_flags 字段中是不够的；开发者还必须提供适当的函数，通过 tp_traverse 和 tp_clear 来指示，并使用循环垃圾收集器正确地注册或者注销开发者的实例。支持循环垃圾收集是一个高级主题，本书没有对此进一步介绍。与此类似，本书也没有介绍有关支持弱引用的高级主题。

tp_doc 字段，也就是一个 char*，是一个以 NULL 结尾的字符字符串，该字符串是开发者的类型的文档字符串。其他字段都指向数据结构（其字段指向函数）；开发者可以将每个这样的字段设置为 NULL 以指示开发者不支持这种类型的函数。指向这样的函数块的字段是 tp_as_number，用于通常由数字提供的特殊方法；tp_as_sequence 用于通常由序列提供的特殊方法；tp_as_mapping 用于通常由映射提供的特殊方法；而 tp_as_buffer 用于缓冲区协议的特殊方法。

例如，不是序列的对象也可以支持 tp_as_sequence 指向的代码块中列出的一个或几个方法，在这种情况下，PyTypeObject 必须有一个非-NULL 字段 tp_as_sequence，即使其指向的函数块指针依次大多数都是 NULL。例如，字典支持__contains__特殊方法，因此在 d 是一个字典时，开发者可以检查 x 是否在 d 中。在 C 代码级别，方法是由 sq_contains 字段指向的一个函数，也是 tp_as_sequence 字段指向的 PySequenceMethods 数据结构的一部分。因此，一个名为 PyDict_Type 的 dict 类型的 PyTypeObject 数据结构包含一个用于 tp_as_sequence 的非-NULL 值，即使一个字典除了 sq_contains 之外不支持 PySequenceMethods 中的其他字段，因此*(PyDict_Type.tp_as_sequence)中的所有其他字段都是 NULL。

类型定义示例

实例 25-2 是一个完整的 Python 扩展模块，定义了非常简单的类型 intpair，这个类型的每个实例都包含两个名为 first 和 second 的整数。

```
#include "Python.h"
#include "structmember.h"

/* 按实例数据结构 */
typedef struct {
    PyObject_HEAD
    int first, second;
} intpair;

static int
intpair_init(PyObject *self, PyObject *args, PyObject *kwds)
{
    static char* nams[] = {"first","second",NULL};
    int first, second;
    if(!PyArg_ParseTupleAndKeywords(args, kwds, "ii", nams, &first,
&second))
        return -1;
    ((intpair*)self)->first = first;
    ((intpair*)self)->second = second;
    return 0;
}

static void
intpair_dealloc(PyObject *self)
{
    self->ob_type->tp_free(self);
}

static PyObject*
intpair_str(PyObject* self)
{
    return PyString_FromFormat("intpair(%d,%d)",
        ((intpair*)self)->first, ((intpair*)self)->second);
}

static PyMemberDef intpair_members[] = {
    {"first", T_INT, offsetof(intpair, first), 0, "first item" },
    {"second", T_INT, offsetof(intpair, second), 0, "second item" },
    {NULL}
};

static PyTypeObject t_intpair = {
    PyObject_HEAD_INIT(0)              /* tp_head */
    0,                                 /* tp_internal */
    "intpair.intpair",                 /* tp_name */
    sizeof(intpair),                   /* tp_basicsize */
    0,                                 /* tp_itemsize */
```

```
    intpair_dealloc,                /* tp_dealloc */
    0,                              /* tp_print */
    0,                              /* tp_getattr */
    0,                              /* tp_setattr */
    0,                              /* tp_compare */
    intpair_str,                    /* tp_repr */
    0,                              /* tp_as_number */
    0,                              /* tp_as_sequence */
    0,                              /* tp_as_mapping */
    0,                              /* tp_hash */
    0,                              /* tp_call */
    0,                              /* tp_str */
    PyObject_GenericGetAttr,        /* tp_getattro */
    PyObject_GenericSetAttr,        /* tp_setattro */
    0,                              /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT,
    "two ints (first,second)",
    0,                              /* tp_traverse */
    0,                              /* tp_clear */
    0,                              /* tp_richcompare */
    0,                              /* tp_weaklistoffset */
    0,                              /* tp_iter */
    0,                              /* tp_iternext */
    0,                              /* tp_methods */
    intpair_members,                /* tp_members */
    0,                              /* tp_getset */
    0,                              /* tp_base */
    0,                              /* tp_dict */
    0,                              /* tp_descr_get */
    0,                              /* tp_descr_set */
    0,                              /* tp_dictoffset */
    intpair_init,                   /* tp_init */
    PyType_GenericAlloc,            /* tp_alloc */
    PyType_GenericNew,              /* tp_new */
    _PyObject_Del,                  /* tp_free */
};

void
initintpair(void)
{
    static PyMethodDef no_methods[] = { {NULL} };
    PyObject* this_module = Py_InitModule("intpair", no_methods);
    PyType_Ready(&t_intpair);
    PyObject_SetAttrString(this_module, "intpair", (PyObject*)&t_intpair);
}
```

与 Python 中的一个对等的定义相比，示例 25-2 中定义的 intpair 类型并没有给出实际上的好处，比如：

```
class intpair(object):
    __slots__ = 'first', 'second'
```

```
    def __init__(self, first, second):
        self.first = first
        self.second = second
    def __repr__(self):
        return 'intpair(%s,%s)' % (self.first, self.second)
```

但是，C 语言编码的版本可以确保这两个属性都是整数，并根据需要截短浮点型或复数数字参数。例如：

```
import intpair
x=intpair.intpair(1.2,3.4)                    # x是: intpair(1,3)
```

在上面的示例中，intpair 的 C 语言编码版本的每个实例都要比 Python 版本的实例或多或少占用更少的内存。但是，示例 25-2 的目的纯粹是为了举例：只是用于表示一个定义了简单新类型的 C 语言编码的 Python 扩展。

25.2 不使用 Python 的 C API 扩展 Python

开发者可以使用 C 语言之外的其他经典编译语言来编写 Python 扩展代码，可供选择的有 Paul Dubois 的 Pyfort（参见 http://pyfortran.sf.net）和 Pearu Peterson 的 F2PY（参见 http://cens.ioc.ee/projects/f2py2e/）。这两个软件包都支持并要求 Numeric 软件包（参见第 16.3 节“Numeric 包”），因为数值处理是 Fortran 的典型应用范围。

对于 C++，开发者有许多选择。SCXX（参见 http://davidf.sjsoft.com/mirrors/mcmillan-inc/scxx.html）是一个简单的，轻量级的软件包，该软件包没有使用模板，因此适合于老的 C++编译器。PyCXX（参见 http://cxx.sf.net）使用了适量的模板，本质上来自于 C++标准库。SIP（参见 http://www.riverbankcomputing.co.uk/sip/index.php）也支持需要使用强大的 Qt 跨平台库的 C++扩展，但是，尽管 SIP 完全支持 Qt，但是并不需要 Qt。Boost Python 库（参见 http://www.boost.org/libs/python/doc）是 Boost 的一部分，是一个包含功能强大、模板丰富的 C++库的巨大宝库，并具有一致性的高质量，需要并支持可以很好地支持模板的现代 C++编译器。

当然，开发者还可以选择从 C++代码中使用 Python 的 C API，可以将 C++代码当作是 C 来使用，同时还可以获得 C++提供的额外好处。但是，如果开发者不管出于什么原因使用了 C++，而不是 C，则使用 SCXX、PyCXX、SIP 或 Boost 与使用 Python 的底层 C API 相比，可以充分地提高编程的效率。

如果开发者的 Python 扩展基本上是一个基于现有 C 或 C++库的包装程序（许多扩展都是如此），可以考虑使用“简化的包装程序和结构生成器”（SWIG），参见 http://www.swig.org。SWIG 可以基于库的头文件为开发者的扩展生成 C 源代码，通常还需要用到一个接口描述文件中的进一步说明。如果开发者特别需要包装一个已有的动态库（Windows 上的.dll 文件，和大多数类 Unix 系统上的.so 文件，包括 Linux 和 Mac OS X）

以从开发者的 Python 代码中使用该库，可以参考 ctypes 扩展（http://starship. python. net/crew/theller/ctypes/），这个扩展为这样的任务提供了完美的支持。在 Python 2.5 中，已经计划将 ctypes 包含在标准 Python 库中。

Greg Ewing 正在开发一个名为 Pyrex 的语言，这种语言是专门用于编写 Python 扩展的。Pyrex（参见 http://www.cosc.canterbury.ac.nz/～greg/python/Pyrex/）是 Python 和 C 概念的一个非常有趣的混合体，尽管还只是一个相对新的开发工具，但是已经非常有用了。Pyrex 适合于包装现有的 C 语言库，以及为被编译为机器语言的 Python 编写快速扩展模块，就像 C 语言编码的扩展那样。Pyrex 是通过先生成中间 C 代码，然后进行编译来工作的。参见第 25.4 节中的介绍。

weave 软件包（参见 http://www.scipy.org/site_content/weave）可以用来在 Python 中运行内联的 C/C++代码。特别是，blitz 函数可以从使用 Numeric 包的表达式来生成和运行 C++代码，因此需要用到 Numeric 包。

如果开发者的应用程序只在 Windows 上运行，用来扩展和嵌入 Python 的最实际的方法有时候就是通过 COM 来实现的。特别是，目前 COM 是从 Python 中使用 Visual Basic 模块（被包装为 ActiveX 类）的最好方法。COM 还是让 Python 编码的功能（被包装为 COM 服务器）可以用于 Visual Basic 程序的最好方法。标准的 Python Windows 发布版本（比如 Python 2.4）不直接支持 COM：开发者需要使用 ctypes（从 Python 2.5 开始，ctypes 成为标准 Python 库的一部分，但是也可以下载 ctypes，并在以前版本的 Python 上安装使用）。本书没有进一步介绍 Windows 专用功能，包括 COM。要想全面地了解平台专用 Python 在 Windows 上的使用，建议阅读 O'Reilly 出版，Mark Hammond 和 Robinson 编著的 *Python Programming on Win32* 一书。

25.3 嵌入 Python

如果开发者已经有了一个使用 C 或 C++（或者其他经典的编译语言）编写的应用程序，可能需要将 Python 嵌入为该应用程序的脚本语言。要想在除 C 语言之外的语言中嵌入 Python，这个其他的语言必须能够调用 C 函数。下面几节只是从 C 语言的角度进行了介绍，因为对于其他一些语言，要想从这些语言中调用 C 函数，要做的事情会有很大的区别。

安装驻留扩展模块

为了让 Python 脚本与开发者的应用程序进行通信，该应用程序必须提供包含 Python 可访问的函数和类的扩展模块，这些函数和类表现了开发者的应用程序的功能。如果这些模块被链接到开发者的应用程序，而不是驻留在动态库中，使得 Python 可以在需要的时候加载这些模块，可以通过调用 PyImport_AppendInittab C API 函数将这些模块作为附加的内置模块注册到 Python 中。

表 25-10

PyImport_AppendInittab	`int PyImport_AppendInittab(char* name,void (*initfunc)(void))` name 是模块的名称，Python 脚本将在 import 语句中使用这个模块名来访问该模块。initfunc 是模块初始化函数，不需要任何参数，也不返回任何结果，参见第 25.1 节（也就是说，initfunc 是该模块的函数，对于动态库中的一个普通扩展模块，这个函数将被命名为 initname）。PyImport_AppendInittab 必须在 Py_Initialize 之前被调用。

设置参数

开发者可能需要通过调用以下这两个 C API 函数，或者其中之一来设置程序的名称和参数，Python 脚本可以通过 sys.argv 访问程序的名称和参数。

表 25-11

Py_SetProgramName	`void Py_SetProgramName(char* name)` 设置程序的名称，Python 脚本可以通过 sys.argv[0]来访问这个程序名称。必须在 Py_Initialize 之前调用该函数。
PySys_SetArgv	`void PySys_SetArgv(int argc,char** argv)` 将程序的参数设置为 argv 中的 argc 个以 0 终止的字符串，Python 脚本可以通过 sys.argv[1:]来访问这些参数。必须在 Py_Initialize 之后调用该函数。

Python 初始化和终止化

在安装额外的内置模块并可选地设置程序的名称之后，开发者的应用程序就可以初始化 Python 了。最后，在不再需要 Python 时，开发者的应用程序将终止化 Python。C API 中的相关函数如下。

表 25-12

Py_Finalize	`void Py_Finalize(void)` 释放 Python 可以释放的所有内存和其他资源。在调用这个函数之后，开发者就不能再执行任何其他 Python C API 调用了。
Py_Initialize	`void Py_Initialize(void)` 初始化 Python 环境。在调用该函数之前，除了 PyImport_AppendInittab 和 Py_SetProgramName 之外（参见第 25.3 节中介绍的 PyImport_AppendInittab 函数和第 25.3 节中介绍的 Py_SetProgramName 函数），不能执行任何其他 Python C API 调用。

运行 Python 代码

开发者的应用程序可以从一个字符字符串，或者从一个文件运行 Python 源代码。要想运行或编译 Python 源代码，可以将执行模式选择为 Python.h 中定义的以下 3 个常量之一：

Py_eval_input

> 代码是一个要计算的表达式（就像将‘eval’传递为 Python 的内置函数 compile 一样）；

Py_file_input

> 代码是一个由一个或多个要执行的语句组成的程序块（与 compile 的‘exec’一样；就像使用一个拖尾的‘\n’来结束复合语句）；

Py_single_input

> 代码是一个用来交互式执行的单个语句（就像 compile 的‘single’一样；隐式输出表达式语句的结果）。

直接运行 Python 源代码类似于将一个源代码字符串传递到 Python 语句 exec 或内置函数 eval 上，或者将一个源代码文件传递到内置函数 execfile。开发者可以使用以下两个通用函数来完成这个任务。

表 25-13

PyRun_File	PyObject* PyRun_File(FILE* *fp*,char* *filename*,int *start*, PyObject* *globals*,PyObject* *locals*) fp 是一个以可读模式打开的源代码文件。filename 是这个文件的名称，用在错误消息中。start 是用来定义执行模式的常数 Py_..._input 之一。globals 和 locals 都是字典（可以是两个相同的字典），用作执行操作的全局和本地命名空间。在 start 是 Py_eval_input 时，该函数将返回表达式的结果，否则返回一个到 Py_None 的新引用，或者返回 NULL 以指示已经引发了一个异常（通常是由于语法错误而引发的，但并不总是如此）。
PyRun_String	PyObject* PyRun_String(char* *astring*,int *start*, PyObject* *globals*,PyObject* *locals*) 与 PyRun_File 类似，但是源代码在以 NULL 结尾的字符串 astring 中。 locals 和 globals 字典通常都是新的、空白的字典（大多数都是通过 Py_BuildValue("{}")很方便的创建的），或者是一个模块的字典。PyImport_Import 是一种可以获得一个已有的模块对象的便利方法；PyModule_GetDict 可以获得一个模块的字典。有时候，开发者可能需要临时创建一个新模块对象，并使用 PyRun_函数调用来装配该对象。要想创建一个新的空白模块，可以使用 PyModule_New C API 函数。

续表

PyModule_New	PyObject* PyModule_New(char* *name*)
	为名为 name 的模块返回一个新的空白模块对象。在这个新对象可用之前，开发者必须向该对象添加一个名为__file__的字符串属性。例如：
	`PyObject* newmod = PyModule_New("mymodule");`
	`PyModule_AddStringConstant(newmod, "__file__", "<synthetic>");`
	在这段代码运行之后，模块对象 newmod 将就绪；开发者可以使用 PyModule_GetDict(newmod)获得该模块的字典，并将这个字典作为 globals 参数（也有可能是 locals 参数）传递到像 PyRun_String 这样的函数中。
	要想重复运行 Python 代码，并将语法错误的诊断从代码运行时引发的运行时异常中分离出来，开发者可以将 Python 源代码编译为一个代码对象，然后保存该代码对象并重复运行该对象。这在使用 C API 和在 Python 中动态执行时都是一样的，参见第 13.1 节。下面是开发者可以用来实现这个任务的两个 C API 函数。

Py_CompileString	PyObject* Py_CompileString(char* *code*,char* *filename*, int *start*) code 是一个以 NULL 结尾的源代码字符串。filename 是错误消息中使用的文件的名称。start 是定义执行模式的常数之一。该函数将返回包含字节代码的 Python 代码对象，对于语法错误，则返回 NULL。
PyEval_EvalCode	PyObject* PyEval_EvalCode(PyObject* *co*,PyObject* *globals*, PyObject* *locals*) co 是一个 Python 代码对象，例如，是由 Py_CompileString 返回的。globals 和 locals 都是被用作执行操作的全局和本地命名空间的字典（可能是两个相同的字典）。在 co 是使用 Py_eval_input 编译时，该函数将返回表达式的结果，否则返回一个到 Py_None 的新引用，或者返回 NULL 以指示执行操作引发了一个异常。

25.4 Pyrex

Pyrex 语言（http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex/）通常是编写 Python 扩展的最便利方法。Pyrex 是 Python 的一个大的子集，其中还包含其他可选的类 C 变量类型：开发者可以将 Pyrex 程序（源文件的扩展名为.pyx）自动编译为机器代码（通过一个生成 C 代码的中间过程），从而生成 Python 可导入扩展。参见上面的 URL 以了解 Pyrex 变成的所有细节；本节只介绍一些可以帮助开发者初步了解 Pyrex 的基础知识。

与 Python 相比，Pyrex 语言的限制如下：

- 在其他语句中没有嵌套 def 和 class 语句（除了一级 class 中的一级 def 之外，并且确实是定义一个类的方法的正确或常规方法）；
- 没有 import*、生成器、列表推导、装饰器或增量赋值；
- 没有 globals 和 locals 内置；
- 要想为一个类提供一个 staticmethod 或 classmethod 方法，开发者必须在类语句之外首先定义（def）函数（在 Python 中，该函数通常是在类中定义的）。

正如开发者可以看到的，尽管并不像 Python 那样功能丰富，但 Pyrex 确实是一个非常大的子集。更重要的是，Pyrex 向 Python 添加了几个允许类 C 声明的语句，可以很容易地生成机器代码（通过一个中间的 C 代码生成步骤）。下面是一个简单示例：在一个新的空白目录中的源文件 hello.pyx 中编写代码：

```
def hello(char *name):
    return "Hello, " + name + "!"
```

这段代码几乎与 Python 代码完全相同——除了参数 name 前面带一个 char*，char*用来声明其类型必须总是一个 C 语言中以 0 结尾的字符串（但是，正如这段代码中可以看到的，在 Pyrex 中，开发者可以根据需要使用一个普通 Python 字符串作为 name 的值）。

在安装 Pyrex 时（通过常规的 python setup.py install 方法），还可以通过常规的 distutils 工具获得一种将 Pyrex 源文件编译到 Python 动态库扩展中的方法。在这个新目录中的 setup.py 文件中编写下面的代码：

```
from distutils.core import setup
from distutils.extension import Extension
from Pyrex.Distutils import build_ext

setup(name='hello',ext_modules=[Extension("hello",["hello.pyx"])],
    cmdclass = {'build_ext': build_ext})
```

现在，在新目录中运行 python setup.py install（忽略编译警告；警告是可以预料到的，但是是良性的）。现在就可以导入并使用这个新模块了——例如，从交互式 Python 会话中输入：

```
>>> import hello
>>> hello.hello("Alex")
'Hello, Alex!'
```

根据开发者编写这个 Pyrex 源代码的方式，用户必须向 hello.hello 传递一个字符串：不传递任何参数、传递多于一个的参数，或者传递不是字符串的参数都将引发一个异常：

```
>>> hello.hello()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function takes exactly 1 argument (0 given)
>>> hello.hello(23)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: argument 1 must be string, not int
```

cdef 语句和函数的参数

开发者可以在大多数可以使用 def 的位置使用关键字 cdef，但是 cdef 定义的函数是扩展模块的内部函数，对外不可见，而 def 函数还可以被导入该模块的 Python 代码调用。对于类型和函数而言，未指明类型的参数和返回值将变成生成的 C 代码中的 PyObject*指针（使用隐式和标准方法处理引用递增和递减）。cdef 函数还可以使用任意其他 C 语言类型的参数和返回值；除了无类型（也相当于 object）之外，def 函数只能接受 int、float 和 char*类型。例如，下面是一个 cdef 函数，专门用来将两个整数相加：

```
cdef int sum2i(int a, int b):
    return a + b
```

开发者还可以使用 cdef 来声明 C 变量：类似于 C 语言中的标量、数组和指针：

```
cdef int x, y[23], *z
```

以及 Python 化的语法 struct、union 和 enum（首部子句带冒号，接下来的语句缩进）：

```
cdef struct Ure:
    int x, y
    float z
```

（然后，可以只按照名称来引用新类型——例如，Ure。永远不要在声明这个类型的 cdef 之外的任何位置使用关键字 struct、union 和 enum。）

外部声明

要想与外部 C 代码接口，开发者可以声明 cdef extern 这样的变量，这个变量具有与 C 语言中 extern 语句相同的效果。更通用的是，开发者还可以在一个.h 的 C 语言头文件中包含有关想要使用的某些库的 C 语言声明；要想确保 Pyrex 生成的 C 代码包含这个头文件，可以使用以下形式的 cdef：

```
cdef external from "someheader.h":
```

后面是一段缩进的 cdef 风格的声明（不要在该代码段中重复关键字 cdef）。开发者只需要声明想要在 Pyrex 代码中使用的函数和变量即可；Pyrex 不会读取这个 C 头文件，而是信任这个代码段中的 Pyrex 声明，因此不为这些声明生成任意 C 代码。不要在 Pyrex 声明中使用 const，因为 Pyrex 并不认识这个关键字！

cdef class

cdef class 语句可以用来在 Pyrex 中定义一个新 Python 类。这条语句可以包含属性

的 cdef 声明（应用于每个实例，而不是总体上应用于类型），该声明通常不能从 Python 代码可见；但是，开发者可以将属性特别声明为 cdef public，这样可以让这些属性成为 Python 所认为的普通属性，或者将属性声明为 cdef readonly，这样可以让这些属性对 Python 可见，但是只读（这样的 Python 可见属性必须是数字、字符串或对象）。

cdef class 语句支持特殊方法（包含一些 caveat）、属性（具有特殊语法）和继承（只支持单个继承）。要想声明一个属性，可以在 cdef class 的类体中使用以下语句：

```
property name:
```

然后对__get__(self)、__set__(self, value)和__del__(self)方法使用缩进的 def 语句（如果属性不允许设置或删除，可以省略其中的一个或多个方法）。

cdef class 的__new__方法不同于普通 Python 类的相同方法：第一个参数是 self，也就是新实例，已经被分配，并且其内存以 0 填充。在对象析构时，Pyrex 将调用特殊方法__dealloc__(self)来取消__new__可能已经完成的任何分配操作（cdef class 没有__del__特殊方法）。

与 Python 中不一样，cdef class 中没有右边（righthand-side）版本的算术特殊方法，比如与__add__方法对应的__radd__方法；而是，如果（假定）a+b 不能找到或使用 type(a).__add__，接下来将调用 type(b).__add__(a, b)——请注意参数的顺序（不要交换位置！）。开发者可能需要尝试某些类型检查以确保在所有情况下都可以执行正确的操作。

要想将一个 cdef class 实例放到迭代器中，可以定义一个特殊方法__next__(self)，而不是像在 Python 中所做的那样调用一个名为 next 的方法。

下面是一个相当于示例 25-2 的 Pyrex 示例：

```
cdef class intpair:
    cdef public int first, second
    def __init__(self, first, second):
        self.first = first
        self.second = second
    def __repr__(self):
        return 'intpair(%s,%s)' % (self.first, self.second)
```

与示例 25-2 中的 C 语言编码的扩展类似，这个 Pyrex 编码的扩展也没有提供比对等的 Python 编码的扩展更实际的优点。但是，请注意，与 C 代码中所需的冗长和模板化语句不通，Pyrex 代码的简明扼要非常接近于 Python 代码，而且从 Pyrex 文件生成的机器代码也非常接近于从示例 25-2 中的 C 代码生成的机器代码。

ctypedef 语句

开发者可以使用关键字 ctypedef 为类型声明一个名称（同义词）：

```
ctypedef char* string
```

for...from 语句

除了常用的 Python 语句之外，Pyrex 还允许使用另外一种形式的 for 语句：

```
for variable from lower_expression<=variable<upper_expression:
```

这是一种最通用的形式，但是开发者可以在关键字 from 之后，在 variable 的任意一边使用<或<=；此外，还可以使用>和/或>=以实现一个反向循环（但是不能混合使用，在一边使用<或<=，而在另外一边使用>或>=）。只要变量和两个循环边界都是 C 语言类型的整数，这个语句要比 Python 中使用的常规 for variable in range(...): 语句快得多。

Pyrex 表达式

除了 Python 表达式语法之外，Pyrex 还可以使用 C 的某些附加表达式，但不是全部。要想获得变量 var 的地址，可以像在 C 语言中一样，使用&var。但是，要想解除对指针 p 的引用，可以使用 p[0]；C 的语法*p 不是有效的 Pyrex。类似地，在 C 中使用 p->q 的地方，在 Pyrex 中则使用 p.q。空指针使用 Pyrex 关键字 NULL。对于字符常量，可以使用语法 c'x'。对于强制转换，可以使用尖括号，比如<int>somefloat，而在 C 中将使用代码(int)somefloat，同样地，只能对 C 语言值和对 C 语言类型使用强制转换，而不能对 Python 值和类型进行强制转换（在 Python 值或类型出现时，可以让 Pyrex 自动执行类型转换）。

Pyrex 示例：最大公约数

求两个数字的最大公约数（GCD）的欧几里德（Euclid）算法在纯 Python 中是非常容易实现的：

```
def gcd(dividend, divisor):
    remainder = dividend % divisor
    while remainder:
        dividend = divisor
        divisor = remainder
        remainder = dividend % divisor
    return divisor
```

Pyrex 版本与此非常类似：

```
def gcd(int dividend, int divisor):
    cdef int remainder
    remainder = dividend % divisor
    while remainder:
        dividend = divisor
        divisor = remainder
```

```
        remainder = dividend % divisor
    return divisor
```

在作者的笔记本电脑上，gcd(454803, 278255)在 Python 版本下花费了大约 6 微秒，而 Pyrex 版本花费了 1.75 微秒。尽管纯的 Python 版本在实际使用中有一些好处（在 Jython 和 IronPython 中的运行就像在经典 Python 中运行一样，处理长整型就像处理整型一样，可以完美地跨平台使用，等等），但是，只需要很小的代价就可以得到 3～4 倍的速度提高是很值得的（当然，假定这个函数占用了程序的主要执行时间！）。

第 26 章

扩展和嵌入 Jython

Jython 可以在 Java 虚拟机（JVM）上实现 Python 语言。Jython 的内置对象都是使用 Java 编码的，比如数字、序列、字典和文件。要想使用 C 语言来扩展经典 Python，必须使用 Python C API 来编写 C 模块（参见第 25.1 节）。要想使用 Java 来扩展 Jython，不必用特殊的方法编写 Java 模块：开发者的 Jython 脚本和使用了 import 语句（参见第 7.1 节）的 Jython 交互式会话都可以自动使用 Java CLASSPATH（或者 Jython 的 sys.path）中的所有 Java 包。这种自动可用性可以应用于 Java 的标准库、开发者安装的第三方 Java 库和开发者自己开发的 Java 类上。开发者可以使用 Java 本地接口（Java Native Interface，JNI）将 Java 扩展到 C，并且这样的扩展可以用于 Jython 代码，就像这些代码是使用纯的 Java 编写的，而不是使用 JNI 兼容的 C 来编写的。

要想了解 Java 和 Jython 之间的互操作性的细节，本书推荐读者阅读 O'Reilly 出版，Samuele Pedroni 和 Noel Rappin 编著的 *Jython Essentials* 一书。本章提供了最简单的互操作性场景的一个简单概述，这已经足够满足大量的实际需要了。在大多数情况下，在 Jython 中导入、使用、扩展和实现 Java 类和接口直接就可以使用。但是，在某些情况下，开发者需要小心与可访问性、类型转换和过载相关的问题（参见本章的介绍）。在 Java 编码的应用程序中嵌入 Jython 解释器类似于在 C 语言编码的应用程序中嵌入 Python 解释器（参见第 25.3 节），但是 Jython 任务要更简单一些。Jython 还提供了另一种与 Java 进行互操作的功能，也就是使用 jythonc 编译器将 Python 源代码编译成经典的静态 JVM 字节代码.class 和.jar 文件。然后，可以在 Java 应用程序和框架中使用这些字节代码文件，就像其源代码已经被移植到 Java，而不是在 Python 中。

本书没有介绍如何使用 C# 或其他在 CLR（Microsoft .NET，或者 CLR 的 Mono 开源实现）上运行的语言来扩展和嵌入 IronPython 这种与前面介绍的非常类似的任务。但是，考虑到 C#和 Java 之间的类似之处，以及同一个程序员 Jim Hugunin 负责创建 Jython 和 IronPython 的事实，开发者在执行这些任务时遇到的大多数问题都非常类似于本章介绍

的问题。参见 http://ironpython.com 以了解有关 IronPyton 的详细信息，其中包含与扩展和嵌入任务有关的内容。在编写本书的时候，网站 http://ironpython.com 没有得到活跃维护，因此，有关 IronPython 的最新的网站是 http://workspaces.gotdotnet.com/ironpython。但是，IronPython 团队正计划在发布 IronPython 1.0 之后激活 http://ironpython.com 网站，使其再次成为 IronPython 的参考网站，在读者阅读到本书的时候，IronPython 1.0 应该已经发布了。

26.1 在 Jython 中导入 Java 包

与 Java 不一样，Jython 没有隐式和自动导入 java.lang。开发者的 Jython 代码可以显式 import java.lang，甚至只需要 import java，然后就可以使用像 java.lang.System 和 java.lang.String 这样的类了，就像这些类是 Python 类一样。特别地，开发者的 Jython 代码可以使用导入的 Java 类，就像这些 Java 类是具有__slots__类属性的 Python 类（也就是，开发者不能创建任意的新实例属性）一样。开发者可以创建一个 Java 类的子类，这个 Java 类中包含开发者自己的 Python 类，而开发者的类的实例可以让开发者像平常一样，只需要通过绑定这些实例就可以创建新属性了。

开发者可以选择导入一个顶层 Java 包（比如 java），而不是特定的子包（比如 java.lang）。在导入顶层包时，开发着的 Python 代码可以获得访问所有子包的能力。例如，在 import java 之后，开发者的代码就可以使用 java.lang.String 和 java.util.Vector 等类了。

Jython 运行时可以将开发者导入的所有 Java 类包装到一个透明代理中，这个透明代理用来管理 Python 和后台的 Java 代码之间的通信。除了第 7.1 节中提到的原因之外，又多了一个避免使用不确定的习惯用法 from somewhere import *的原因。在执行这样的一个大批量导入时，Jython 运行时必须为 somewhere 包中的所有 Java 类创建代理包装器，并且花费大量的内存和时间来包装开发者的代码可能不会使用的许多类。除了在交互式测试会话中为了方便偶尔使用 from … import *之外，开发者要尽量避免使用这条语句，并坚持使用 import 语句。另外，对于开发者已知的，并且在 Python 代码中专门需要使用的类，使用特定的显式 from 语句导入是可以的（例如，from java.lang import System）。

Jython 注册表

Jython 依赖于一个 Java 属性注册表（registry）作为这类设置的跨平台处理方法，这些设置通常使用 Windows 注册表或者类 Unix 系统上的环境变量。Jython 的注册表文件是一个名为 registry 的标准 Java 属性文件，位于被称为 Jython 根目录的目录中。Jython 根目录通常是 jython.jar 所在的目录，但是开发者可以通过设置 Java 属性 python.home 或 install.root 来改写这个目录。对于特殊需要，开发者可以通过主目录中的一个名

为.jython 的辅助 Java 属性文件，和/或通过用于 jython 解释器命令的命令行选项来调整 Jython 注册表设置。注册表选项 python.path 相当于经典 Python 的 PYTHONPATH 环境变量。这是开发者可能最感兴趣的选项，因为这个选项可以帮助开发者在 Jython 安装目录之外安装额外的 Python 软件包（例如，共享开发者已经为 CPython 的使用而安装的纯的 Python 软件包）。

可访问性

通常，开发者的 Jython 代码只能访问 Java 类的 public 功能（方法、字段和内部类）。开发者还可以选择在运行 Jython 之前在 Jython 注册表中设置一个选项以让 private 和 protected 功能可用：

```
python.security.respectJavaAccessibility=false
```

对于常规操作而言，这样弯曲常规 Java 规则的操作不是必需的。但是，访问私有和被保护功能的能力在用于完整测试一个 Java 包的 Jython 脚本中可能是有用的，这就是为什么 Jython 为开发者提供这个选项的原因。

类型转换

Jython 运行时可以在 Python 和 Java 之间透明转换数据。但是，在一个 Java 方法期望得到一个布尔型参数时，开发者必须传递一个整型或者一个 java.lang.Boolean 实例以从 Python 调用这个方法。在 Python 中，任何对象都可以被看作是 True 或 False，但是 Jython 不会对方法调用隐式执行到布尔型值的转换，从而避免混淆和出现错误的危险。处于这个目的，Jython 的新版本 2.2（在编写本书的时候，还只是 alpha 版本）也支持更自然的 Python 布尔类型的选择。

调用重载的 Java 方法

Java 类可以提供重载（overloaded）方法（也就是说，几个方法具有相同的名称，通过其参数的数量和类型来区分）。Jython 可以在运行时解析对重载方法的调用，这是根据 Python 代码在每个给定的调用中传递的参数数量和类型来实现的。如果 Jython 的隐式重载解析没有给出开发者想要的结果，则可以通过显式传递 Java 的 java.lang 包装器类的实例来实现，比如，在 Java 方法期望得到一个整型参数时传递 java.lang.Integer，在 Java 方法期望得到一个浮点型参数时传递 java.lang.Float，依此类推。例如，如果一个 Java 类 C 提供了具有两个重载版本的方法 M，也就是 M(long x)和 M(int x)，考察下面的代码：

```
import C, java.lang

c = C()
c.M(23)                          # 调用 M(long)
c.M(java.lang.Integer(23))       # 调用 M(int)
```

根据 Jython 重载解析原则，c.M(23)将调用 long 重载方法。但是，c.M(java.lang.Integer(23))将显式调用 int 重载方法。

jarray 模块

在开发者将 Python 序列传递到期望得到数组参数的 Java 方法时，Jython 将执行自动转换，将 Python 序列中的每个项目都复制到 Java 数组的一个元素中。在开发者调用一个可以接受和修改数组参数的 Java 方法时，开发者传递的 Python 序列不能影响 Java 方法对其数组参数执行的任何更改。要想有效地调用可以更改数组参数的方法，Jython 提供了 jarray 模块，该模块提供了两个工厂函数，可以帮助开发者直接创建 Java 数组。

表 26-1

array

`array(seq,typecode)`

seq 是任意 Python 序列。typecode 是一个 Java 类或者单个字符（按照下表指定一个原始的 Java 类型）。array 可以创建一个 Java 数组 a，该数组具有与 seq 相同的长度，其中的元素是 typecode 给定的类或类型的值。array 将根据 seq 的对应项目初始化 a 中的元素。

jarray 模块的类型代码

类型代码	Java 类型
'b'	Byte
'c'	Char
'd'	Double
'f'	Float
'h'	Short
'i'	Int
'l'	Long
'z'	Boolean

zeros

`zeros(length,typecode)`

创建一个 Java 数组 z，长度为 length，其中的元素是 typecode 给定的类或类型的值，这两个参数与 array 函数中的参数具有相同的含义。zeros 将把 z 中的每个元素初始化为 0、null 或 false，对应于给定的类型或类。当然，在从 Jython 代码访问这些元素时，可以将其看作是相等的 Pytohn 0 值（或者是 None，作为与 Java null 对应的 Jython 值），但是，在 Java 代码访问这些元素时，将以适当的 Java 类型和值来处理这些元素。

开发者可以将 array 和 zeros 函数创建的实例用作固定长度的 Python 序列。在将这样的一个实例传递到一个可以接受数组参数并修改该参数的 Java 方法时，所作的更改在传递的实例中是可见的，因此，开发者的 Python 代码可以有效地调用这样的方法。

java.util 集合类

Jython 不能使用上面介绍的两种方式在 Python 容器和 java.util 包的集合类（比如 java.util.Vector、java.util.Dictionary 等）之间执行自动转换。但是，Jython 向包装器添加了一个其为 Java 集合类创建的一个最小数量的支持功能，以帮助开发者根据实际情况将集合类实例作为 Python 序列、可迭代对象或映射来处理。

从 Java 类继承

Python 类可以从一个 Java 类继承（相当于 Java 中的 extends）和/或从 Java 接口继承（相当于 Java 中的 implements），以及从其他 Python 类继承。Jython 类不能直接或间接从多个 Java 类继承。对于从接口继承，没有任何限制。Jython 代码可以访问 Java 超类的 protected 方法，但是不能访问 protected 字段。开发者可以覆盖未完成的（non-final）超类方法。特别是，开发者必须总是覆盖继承的接口方法和抽象方法（如果有的话，在开发者的超类也是抽象类时）。如果一个方法在超类中被重载，在覆盖这个方法时必须支持重载方法的所有签名。要做到这一点，开发者可以定义自己的方法以接受可变数量的参数（通过让其最后一个形式参数使用特殊形式*args），并在运行时检查每次调用时接收到的参数的数量和类型，从而知道哪个重载变体被调用。

JavaBeans

Jython 为 JavaBeans 习惯用法提供了特殊的支持，这些习惯用法包括命名访问器方法 getSomeThing、isSomeThing 和 setSomeThing。在 Java 类中存在这样的方法时，Python 代码可以访问该 Java 类的实例，并在该实例上设置一个名为 someThing 的属性，然后使用 Python 语法进行属性的访问和绑定：Jython 运行时将把这个语法透明翻译为调用适当的访问器方法。

26.2 在 Java 中嵌入 Jython

开发者的 Java 编码的应用程序可以嵌入 Jython 解释器以使用 Jython 进行脚本处理。jython.jar 必须在 Java 的 CLASSPATH 路径中。开发者的 Java 代码必须导入 org.python.core.*和 org.python.util.*以访问 Jython 的类。为了初始化 Jython 的状态并实例化一个解释器，可以使用下面的 Java 语句：

```
PySystemState.initialize();
PythonInterpreter interp = new PythonInterpreter();
```

为了让开发者详细确定如何建立 PySystemState，并控制系统状态和每个解释器实例的全局范围，Jython 还提供了这个方法和构造函数的几个高级重载函数。但是，在比较典型和简单的情况下，开发者的应用程序只需要使用上面的 Java 代码即可。

PythonInterpreter 类

在有了 PythonInterpreter 类的一个实例 interp 之后，就可以调用 interp.eval 方法以让该解释器计算 Java 字符串中保存的 Python 表达式了。开发者还可以调用 interp.exec 和 interp.execfile 的任意一个重载函数以让该解释器执行 Java 字符串、预编译的 Jython 代码对象、文件或 Java InputStream 中保存的 Python 语句。

开发者执行的 Python 代码可以 import 开发者的 Java 类以访问开发者的应用程序的功能。开发者的 Java 代码可以通过调用 interp.set 的重载函数以在解释器命名空间中设置属性，并通过调用 interp.get 的重载函数以从解释器的命名空间获得属性。这些方法的重载函数给了开发者一个选择。开发者可以处理本地 Java 数据并让 Jython 执行类型转换，或者开发者可以直接处理 PyObject，也就是所有 Python 对象的基类，参见 6.2 节。PythonInterpreter 实例 interp 最常使用的方法和重载函数如表 26-2 所示。

表 26-2

eval	`PyObject interp.eval(String s)` 在 interp 的命名空间中计算 Java 字符串 s 中保存的 Python 表达式，并返回作为该表达式的结果的 PyObject 对象。
exec	`void interp.exec(String s) void interp.exec(PyObject code)` 在 interp 的命名空间中执行 Java 字符串 s 中，或者已编译的 PyObject code（是由 org.python.core 包的__buildin__.compile 函数生成的，参见第 26.2 节中介绍的__buildin__类）中保存的 Python 语句。
execfile	`void interp.execfile(String name) void interp.execfile(java.io.InputStream s) void interp.execfile(java.io.InputStream s,String name)` 在 interp 的命名空间中执行从流 s 或者从文件 name 读取的 Python 语句。在同时传递了 s 和 name 时，execfile 将从 s 中读取语句，并使用 name 作为错误消息中的文件名。

get	`PyObject interp.get(String name) Object interp.get(String name, Class javaclass)`
	从 interp 的命名空间中提取 name 属性的值。包含两个参数的重载函数也可以将属性的值转换为指定的javaclass，如果转换操作是不可行的，将抛出一个包装了 Python TypeError 错误的 Java PyException 异常。如果 name 是无界的，任何一个重载函数都将引发一个 NullPointerException 异常。两个参数形式的重载函数的典型使用可以是下面这样的 Java 语句：
	`String s = (String)interp.get("attname", String.class);`

set	`void interp.set(String name,PyObject value) void interp.set (String name,Object value)` 将 interp 的命名空间中名为 name 的属性绑定到 value 上。第二个重载函数也可以将这个值转换为一个 PyObject。

__build__类

org.python.core 包提供了一个__buildin__类，其静态方法可以让开发者的 Java 代码访问 Python 内置函数的功能。特别地，compile 方法非常类似于 Python 内置函数 compile，参见第 8.2 节中介绍的 compile 函数。开发者的 Java 代码可以带 3 个 String 参数（一个源代码字符串、一个在错误消息中使用的文件名和一个通常为“exec”的 kind）调用 compile，而 compile 将返回一个 PyObject 实例 p，该实例是一个预编译的 Python 字节代码对象。开发者可以重复调用 interp.exec(p)以在 p 中执行 Python 语句，而不会产生为每个执行编译 Python 源代码的开销。这个类的好处与第 13.1 节中介绍的好处相同。

PyObject 类

从 Java 的角度看，所有 Jython 对象都是扩展了 PyObject 的类的实例。PyObject 类提供了一些方法，这些方法的名称类似于 Python 对象的特殊方法，比如__len__、__str__等。PyObject 的具体子类覆盖了某些特殊方法以提供有意义的实现。例如，__len__对于 Python 序列和映射是有意义的，但是对数字是没有意义的；__add__对数字和序列是有意义的，但是对映射是没有意义的。在开发者的 Java 代码对一个 PyObject 实例调用特殊方法，但是该实例实际上并不提供这个方法时，调用操作将引发一个包装了 Python AttributeError 错误的 Java PyException 异常。

由于属性名称可以是一个 PyString 或一个 Java 字符串，因此用于设置、获取和删除属性的 PyObject 方法存在两个重载函数。由于键或索引可以是一个 PyObject、一个 Java 字符串、或者一个整数，因此用于设置、获取和删除项目的 PyObject 方法存在 3 个重载函数。开发者用作属性名称或项目的键的 Java 字符串实例必须是 Java 的内部化字符串（也就是说，是一个字符串字面常量或者对任意 Java 字符串实例 s 调用 s.intern( )的结果）。除了常规的 Python 特殊方法__getattr__和__getitem__之外，PyObject 类提供了类似的__findattr__和__finditem__方法，这两种特殊方法之间的区别在于，在没有找到属性或项目时，__find...方法将返回一个 Java 中的 null，而__get...方法将引发异常。

每个 PyObject 实例 p 都有一个__tojava__方法，这个方法只用到单个参数，一个 Java 类 c，并返回一个 Object，也就是将 p 转换为 c 后的值（如果转换操作是不可行的，将引发一个异常）。典型的使用可以是下面这样的 Java 语句：

```
String s = (String)mypyobj.__tojava__(String.class);
```

PyObject 的__call__方法具有几个很方便使用的重载函数，但是所有的重载函数的语义都归结为__call__的基础形式：

```
PyObject p.__call__(PyObject args[], String keywords[]);
```

在 keywords 数组的长度为 L 时，数组 args 必须满足长度 N>=L，并且 args 的最后 L 个项目将被看作是命名参数，其名称是 keywords 中的对应项目。在 args 的长度 N>L 时，args 的前 N−L 个项目将被看作是位置实际参数。因此，对等的 Python 代码类似于：

```
def docall(p, args, keywords):
    assert len(args) >= len(keywords)
    deltalen = len(args) - len(keywords)
    return p(*args[:deltalen], ** dict(zip(keywords, args[deltalen:])))
```

Jython 提供了表示所有内置 Python 类型的 PyObject 的具体子类。开发者可以实例化这样一个具体子类以创建一个 PyObject 以供将来使用。例如，PyList 类扩展了 PyObject，实现了一个 Python 列表，并且包含一些以一个数组或 PyObject 实例的一个 java.util.Vector 为参数的构造函数，以及一个创建空白列表[ ]的空构造函数。

Py 类

Py 类提供了几个有用的类属性和静态方法。Py.None 是 Python 的 None。Py.java2py 方法有一个 Java Object 参数，并将返回对应的 PyObject。对于指定了一个 Java 原始类型（boolean、byte、long、short 等）的 type 的所有值，Py.py2type 方法有一个 PyObject 参数，并将返回给定原始 Java 类型的对应值。

26.3 将 Python 编译到 Java 中

Jython 附带了 jythonc 编译器。开发者可以为 jythonc 提供.py 源文件，而 jythonc 将把这些源文件编译为常规的 JVM 字节代码，并将其打包到.class 和.jar 文件中。因为 jythonc 将生成传统的静态字节代码，因此不能非常好地处理 Python 允许的所有范围的动态功能。例如，jythonc 不能成功地编译在运行时才能动态确定其基类的 Python 类，而常规的 Python 解释器可以这样做。但是，除了这样的动态可变类结构的极端示例，jythonc 支持将几乎整个 Python 语言编译到 Java 字节代码中。

jythonc 命令

jythonc 被保存在 Jython 安装目录下的 Tools/jythonc 目录中。开发者可以使用下面的语法从一个 shell（控制台）命令行上调用 jythonc：

```
jythonc options modules
```

options 是以--开始的零个或多个选项标记。modules 是零个或多个要编译的 Python 源文件的名称，可以是保存在 Python 的 sys.path 路径中的模块的 Python 风格的名称，或者是到 Python 源文件的相对或绝对路径。开发者可以在每个到源代码的路径中包括.py 扩展，但是不能在一个模块名中包括.py 扩展。

有时候，开发者需要指定 jythonc 选项--jar jarfile 以创建一个已编译的字节代码.jar 文件，而不是单独的.class 文件。大多数其他选项用来处理要将什么放到.jar 文件中。开发者可以选择让文件本身包含所有内容（用于不支持使用多个.jar 文件的浏览器和其他 Java 运行时环境），其代价就是让文件变得更大。选项--all 可以确保所有 Jython 核心类都被复制到.jar 文件中，而--core 趋向于更保守的方案，复制尽可能少的核心类。选项--addpackages packages 可以列出（packages 中，一个逗号分隔的列表）其类被复制到.jar 文件中的那些外部 Java 包，只要 Python 类 jythonc 是依赖于这些包而编译的。--jar 的一个重要的可替代选项是--bean jarfile，这个选项还可以根据需要在.jar 文件中包含一个 Java bean 的清单，这个清单中列出了以 Python 编码的 JavaBeans 组件。

另一个非常有用的 jythonc 选项是--package package，这个选项可以指示 Jython 将其正在创建的所有新 Java 类放到给定的 package 中（还有反映 Python 侧包结构所需的 package 的所有子包）。

添加 Java 可见的方法

jythonc 创建的 Java 类通常是从 Java 库扩展已有类和/或实现已有接口的。其他 Java 编码的应用程序和框架通过构造函数重载实例化了 jythonc 创建的类，这些重载函数具有与其 Java 超类的构造函数相同的签名。Python 侧的__init__将在超类被初始化之后执行，并使用相同的参数（因此，不要在一个使用 jythonc 编译的 Python 类的__init__中__init__一个 Java 超类）。然后，Java 代码可以通过调用已知接口或超类中定义，以及 Python 代码覆盖的实例方法以访问 Python 编码的类的实例的功能。

Python 代码不能提供 Java 可见的 static 方法或属性，而只能提供实例方法。在默认情况下，每个 Python 类只提供其从扩展的 Java 类或者其实现的 Java 接口继承的实例方法。但是，Python 代码还可以通过@sig 指令提供其他 Java 可见的实例方法。

要想让开发者的 Python 类的方法在 Jythonc 编译该类是对 Java 可见，可以将这个方法的文档字符串编码为@sig，并在后面带一个 Java 方法的签名。例如：

```
class APythonClass:
    def__init__(self, greeting="Hello, %s!"):
        "@sig public APythonClass(String greeting)"
        self.greeting = greeting
    def hello(self, name):
        "@sig public String hello(String name)"
        return self.greeting % name
```

要想让一个构造函数对外公开，可以使用该类的@sig 签名，如上面示例中的__init__方法文档字符串所示。@sig 签名中的所有类名都必须是完全合格的，除了来自于

java.lang 的名称和正在被编译的 Python 编码的模块提供的名称之外。在带有@sig 的 Python 方法具有可选参数时，jythonc 将生成带有每个合法签名的 Java 可见方法的重载函数，并处理在需要的地方提供默认参数值。带有@sig 的__init__构造函数将使用其超类的空构造函数隐式初始化该超类。

由于 Python 类不能直接将数据属性对 Java 公开，因此开发者可能需要使用常用的 JavaBeans 方法编写访问器，并通过@sig 机制将其公开。例如，上面的示例中的 APythonClass 实例不允许 Java 代码直接访问或更改 greeting 属性。在需要这样的功能时，开发者可以像下面这样，在子类中提供这个功能：

```
class APythonBean(APythonClass):
    def getGreeting(self):
        "@sig public String getGreeting()"
        return self.greeting
    def setGreeting(self, greeting):
        "@sig public void setGreeting(String greeting)"
        self.greeting = greeting
```

Python Applet 和 Servlet

在已有的 Java 框架中使用 Jython 的两个简单示例是 applet 和 servlet。Applet 是 jythonc 使用的典型示例（带有特殊的警告信息），而 servlet 是专门由 Jython 提供的工具支持的。

Python applet

Jython applet 类必须导入 java.applet.Applet 并扩展该类，通常是覆盖 paint 和其他方法。开发者可以使用--jar somejar.jar 选项和--core 或--all 选项调用 jythonc 将 applet 编译到一个.jar 文件中。通常，Jython 被安装在现代的 Java 2 环境中，对于大多数使用而言，这样做是可以的。只要 applet 只在支持 Java 2 的浏览器（通常带有一个 Sun 提供的浏览器插件）中运行，applet 可以很好地运行。但是，如果开发者需要支持被限定为使用 Java 1.1 的浏览器，则必须确保使用的 JDK 是 1.1 版本，并且必须在 JDK 1.1 环境下使用 Jython 编译该 applet。在不同的 JDK 之间共享单个 Jython 安装是可能的，比如在 1.1 和 1.4 之间。但是，本书建议开发者在单独的目录中分别安装 Jython，在每个需要支持的 JDK 下安装一个 Jython，这样可以最小化出现混淆和意外的风险。

Python servlet

开发者可以使用 jythonc 创建并部署 servlet。但是，Jython 还提供了另一种方法，可以让开发者以.py 源文件的形式部署 Python 编码的 servlet。使用 Jython 提供的 servlet 类 org.python.util.PyServlet 和一个所有*.py URL 到 PyServlet 的 servlet 映射。每个 servlet .py

文件必须被保存在 web-app 顶层目录中，并且必须公开一个不带任何参数的可调用对象（通常是一个类），这个对象具有与该.py 文件相同的名称。PyServlet 将使用这个可调用对象作为该 servlet 的实例的工厂，并依照 Java Servlet API 对这些实例调用方法。开发者的 servlet 实例将轮流访问 Servlet API 对象（比如 request 和 response 对象，并将这些对象传递为方法的参数）和这些对象的属性和方法（比如 response.outputStream 和 request.getSession）。PyServlet 提供了一种完美的、快速转变的方法以试验 servlet，并快速部署 servlet。

第 27 章

发布扩展和程序

Python 的 distutils 允许开发者以几种不同的方式打包 Python 程序和扩展，并安装这些程序和扩展，以将其与 Python 安装程序一起使用。正如本书在第 25.1 节中提到的，distutils 还提供了最简单和最有效的方法以创建开发者自己编写的 C 语言编码的扩展，即使是在开发者对将这样的扩展发布给其他任何人都不感兴趣。本章将介绍 distutils，以及其他一些作为 distutils 补充的第三方工具，这些工具可以帮助开发者对 Python 程序进行打包，以将其作为单独的应用程序进行发布，并且可以在没有安装单独的 Python 安装程序的情况下，可以在具有特殊硬件和操作系统的计算机上进行安装。第 7.4 节中介绍的一种可以免费下载的第三方框架提供了一种更简单，功能也更强大的方法来打包 Python 程序和扩展。

27.1 Python 的 distutils

distutils 是一个功能丰富和灵活的工具集，可以用来打包 Python 程序和扩展以将其发布到第三方。本节将介绍能够满足最常用的打包需要的 distutils 的典型和简单使用。对于有关 distutils 的深入和非常详细的介绍，本书推荐两个指南，这两个指南是 Python 的在线文档的一部分：Distributing Python Modules（参见 http://www.python.org/doc/current/dist/）和 Installing Python Modules（参见 http://www.python.org/doc/current/inst/），这些文档都是由 distutils 的主要作者 Greg Ward 提供的。

发布版本及其根目录

发布（distribution）版本是打包到单个文件以供发布的一组文件。一个发布版本可能包含零个、一个或多个 Python 包和其他 Python 模块（参见第 7 章），以及可选的 Python 脚本、C 语言（和其他语言）编码的扩展、支持数据文件和包含有关发布本身的元数据的辅助文件。如果发布版本包含的所有代码都是以 Python 编写的，则称之为纯 Python

发布，如果发布版本包含非-Python 代码（最常见的情况是，包含 C 语言编码的扩展或 Pyrex 扩展），则称之为非纯 Python 发布。

开发者通常应该将一个发布版本的所有文件放到一个被称为“发布版本的根目录”的目录和这个根目录的子目录中。在大多数情况下，开发者可以调整以发布版本的根目录为根目录的文件和目录的子树以满足开发者自己的目录管理需要。但是，正如第 7.3 节中介绍的，Python 包必须保存在其自己的目录中，并且一个包的目录必须包含一个名为__init__.py 的文件（如果这个包有子包的话，子包子目录中也包含__init__.py 文件），以及属于这个包的其他模块。

setup.py 脚本

发布版本的根目录必须包含一个 Python 脚本，依惯例，这个 Python 脚本被命名为 setup.py。理论上讲，这个 setup.py 脚本可以包含任意 Python 代码。但是，实际上，setup.py 总是被归结为下面这段代码的变体之一：

```
from distutils.core import setup, Extension
setup( many named arguments go here )
```

所有动作都在调用 setup 时提供的参数中。如果开发者的 setup.py 用于处理一个纯 Python 发布，则不要导入 Extension。只有非纯 Python 发布才需要 Extension，并且开发者应该只在需要的时候才导入 Extension。当然，为了以更清晰和更可读的方式安排 setup 的参数，在调用 setup 之前有几条语句也是可以的，这种方式要比让所有代码都内联为 setup 调用的一部分更清晰和可读。

distutils.core.setup 函数只能接受命名参数，并且开发者可以潜在地提供大量这样的参数。有几个参数可以处理 distutils 本身的内部操作，除非开发者打算扩展或调试 distutils，否则不要提供这样的参数。本书没有介绍这个高级主题。setup 的其他命名参数可以分为两组：有关该发布版本的元数据和有关哪些文件在该发布版本中的信息。

有关发布版本的元数据

开发者必须提供有关发布版本的元数据，这是通过在调用 distutils.core.setup 函数时提供下面这些关键字参数实现的。与开发者提供的每个参数名称相关的值是一个字符串，主要目的是为了使参数更有可读性；有关该字符串格式的规范大多是建议性的。下面有关这些元数据字段的解释和推荐也都是非标准化的，只对应于通用原则而不是普遍的规定。只要下面的解释对应于开发者的“这个发布版本”，下面这些短语可以用于这个发布版本中包含的资料，而不是对应于发布版本中的包装。

author

这个发布版本中包含的资料的作者的名称。开发者必须总是提供这个信息，因为作

者应该为他们的工作而获得荣誉。

author_email

author 参数中指定的作者的电子邮件地址。只有在作者想要接收有关这个工作的电子邮件时，才应该提供这个信息。

classifiers

用来对开发者的包进行分类的 Trove 字符串列表；每个字符串必须是网站 http://www.python.org/pypi?%3Aaction=list_classifiers 上列出的字符串之一。

contact

这个发布版本的主要联系人或邮件列表的名称。如果还有人必须优先于 author 和 maintainer 参数中指定的人，则必须提供这个信息。

contact_email

contact 参数中指定的联系人的电子邮件地址。当且仅当开发者提供了 contact 参数时，必须提供这个信息。

description

这个发布版本的简要描述，最好适合放在一行 80 个字符中，或者更少。开发者应该总是提供这个信息。

fullname

这个发布版本的完整名称。如果被提供为 name 参数的名称是缩写或不完整的形式（例如，同义词），则必须提供这个信息。

keywords

查找这个发布版本提供的功能的人可能会搜索的关键字列表。如果可以证实这些关键字对某些人在搜索引擎中索引这个发布版本有用，则应该提供这个信息。

license

这个发布版本的简要形式的许可条款，可以参考这个发布版本中的一个文件或者一个 URL 以了解其详细信息。开发者应该总是提供这个信息。

maintainer

这个发布版本当前的维护人员的名称。如果这个维护人员不是作者，通常必须提供这个信息。

maintainer_email

maintainer 参数中指定的维护人员的电子邮件地址。只有在开发者提供了 maintainer

参数，并且这个维护人员打算接收有关这个工作的电子邮件时，必须提供这个信息。

name

这个发布版本的名称，可以作为一个有效 Python 标识符（通常要求缩写，例如，同义词）。开发者必须总是提供这个信息。

platforms

已知这个发布版本可以运行的平台列表。如果开发者有理由相信这个发布版本可能不能在所有平台上运行，则必须提供这个信息。这个信息必须适度简明，因此这个字段可以引用发布版本中的一个文件或一个 URL 中的详细信息。

url

一个 URL，可以从这个 URL 找到有关这个发布版本的更多信息。只要存在这样的 URL，都应该总是提供这个信息。

version

这个发布版本和/或其内容的版本信息，通常按照 major.minor 或者更细致的结构表示。开发者必须总是提供这个信息。

发布内容

一个发布版本可以包含 Python 源文件、C 语言编码的扩展和数据文件等的混合文件。setup 可以接受可选的关键字参数，这些参数详细说明了哪些文件可以放到这个发布版本中。只要开发者指定了文件的路径，这些路径必须与该发布版本的根目录相关，并使用“/”作为路径分隔符。在安装发布版本时，distutils 将适当地修改位置和分隔符。但是，请注意关键字参数 packages 和 py_modules 并没有列出文件路径，而是分别列出了 Python 包和模块的路径。因此，在这些关键字参数的值中，不要使用路径分隔符或文件扩展名。当开发者在 packages 参数中列出了子包名称时，可以改为使用 Python 语法（例如，top_package.sub_package）。

Python 源文件

在默认情况下，setup 将在发布版本的根目录下查找 Python 模块（关键字参数 py_modules 的值中列出了这些模块），并在发布版本根目录的子目录中查找 Python 包（关键字参数 packages 的值中列出了这些包）。开发者可以指定关键字参数 package_dir 以更改这些默认值。不过，按照 setup 的默认值来定位文件要更简单一些，因此本书没有进一步介绍 package_dir。

开发者最常使用的 setup 关键字参数（用来详细指定哪些 Python 源文件是发布版本的哪些部分）如表 27-1 所示。

表 27-1

packages	`packages=[ list of package name strings ]` 对于列表中的每个包名字符串 p，setup 将在发布版本的根目录中找到一个子目录 p，并在发布版本中包含一个文件 p/__init__.py，这个文件一定是与任意其他文件 p/*.py（也就是，p 包的所有模块）一起出现的。setup 不会搜索 p 的子包：开发者必须在关键字参数 packages 的值中显式列出所有的子包，以及顶层包。
py_modules	`py_modules=[ list of module name strings ]` 对于列表中的每个模块名字符串 m，setup 将在发布版本的根目录中找到一个文件 m.py，并将 m.py 包含在发布版本中。
scripts	`scripts=[ list of script file path strings ]` 脚本是将被作为主程序运行（通常从命令行运行）的 Python 源文件。scripts 关键字列表的值列出了这些文件相对于发布版本根目录的路径字符串，以.py 文件扩展名结束。 每个脚本文件都应该将其第一行作为 shebang 行，也就是一个以#!开始，并包含子字符串 python 的行。在 distutils 安装该发布版本中包含的脚本时，distutils 将把每个脚本的第一行改变为指向 Python 解释器。这在许多平台上是非常有用的，因为这个 shebang 行将被平台的 shell 或者其他可能会运行该脚本的程序所使用，比如 Web 服务器。

数据文件

要想将任意类型的数据文件放到发布版本中，需要提供如表 27-2 所示这个关键字参数。

表 27-2

data_files	`data_files=[ list of pairs (target_directory,[list of files]) ]`
	关键字参数 data_files 的值是一个数据对列表。每个数据对的第一个项目是一个字符串，命名了目标目录（也就是，在安装这个发布版本时 distutils 放置数据文件的目录）；第二个项目就是要被放到目标目录中的文件的文件路径字符串列表。在安装该发布版本时，对于纯发布版本，distutils 将把每个目标目录放置为 Python 的 sys.prefix 子目录；对于非纯发布版本，distutils 将把每个目标目录放置为 Python 的 sys.exec_prefix 子目录。distutils 将把给定文件直接放到各自的目标目录中，而不是目标目录的子目录中。例如，给定以下 data_files 用法：
	`data_files = [('miscdata', ['conf/config.txt', 'misc/sample.txt'])]`
	distutils 将在发布版本中包含来自发布版本根目录的 conf 子目录中的 config.txt 文件和来自 misc 子目录中的 sample.txt 文件。在安装该发布版本时，distutils 将在 Python 的 sys.prefix 目录中（如果该发布版本不是纯的 Python，则在 sys.exec_prefix 目录中）创建一个名为 miscdata 的子目录，并将这两个文件复制到 miscdata/config.txt 和 miscdata/sample.txt 文件中。

C 语言编码的扩展模块

要想将 C 语言编码的模块放到发布版本中，需要提供以下关键字参数。

表 27-3

ext_modules	`ext_modules=[ list of instances of class Extension ]` 在实例化 distutils.core.Extension 类时，有关每个扩展模块的所有详细信息都被提供为参数。 Extension 的构造函数可以接受下面介绍的两个强制参数和许多可选的关键字参数。
Extension	`class Extension(name, sources, **kwds)` name 是这个 C 语言编码的扩展模块的模块名字符串。name 可以包含句点符号以指示该扩展模块保持在一个包中。sources 是 distutils 为了编译这个扩展模块而必须编译和链接的 C 语言源文件列表。sources 中的每个项目都是一个字符串，这个字符串给出了一个源文件相对于该发布版本的根目录的文件目录，以.c 文件扩展名结束。kwds 可以用来将其他参数传递到 Extension 中，参见本节后面的介绍。

Extension 类还支持除.c 之外的其他文件扩展名，扩展名用来指示可能用于编写 Python 扩展的其他语言。在安装了 C++编译器的平台上，文件扩展名.cpp 表示 C++源文件。可能支持的其他文件扩展名取决于平台和 distutils 的各种附件，包括表示 Fortran 的.f、表示 SWIG 的.i 和表示 Pyrex 文件的.pyx。参见第 25.2 节以了解有关如何使用不同的语言扩展 Python 的信息。

在某些情况下，开发者的扩展模块不需要除了强制的 name 和 sources 参数之外的其他信息。distutils 将隐式执行让 Python 的头文件目录和 Python 的库文件可以用于开发者的扩展模块编译和链接所必需的所有操作，并提供在给定平台上编译该扩展模块所需要的所有编译器或链接器标记或选项。

在需要用到附加信息以正确编译和链接开发者的扩展模块时，开发者可以通过 Extension 类的关键字参数提供这样的信息。这样的参数可能会潜在地干扰开发者的发布版本的跨平台可移植性。特别是，在任何时候开发者指定了文件或目录路径作为这样的参数的值时，这些路径必须相对于该发布版本的根目录；使用绝对路径将严重影响开发者的发布版本的跨平台可移植性。

在开发者只是使用 distutils 作为一种很便利的方法来编译自己的扩展模块时（按照第 25.1 节中建议的方法），可移植性并不是什么问题。但是，当开发者计划将自己的扩展模块发布到其他平台时，必须检查是否真的需要通过 Extension 的关键字参数提供编译信息。有时候可以通过 C 语言级别的仔细编码来避免这样的需要的，前面已经提到的 *Distributing Python Modules* 指南提供了重要的示例。

在调用 Extension 时可以传递的关键字参数如下。

define_macros = [(*macro_name*, *macro_value*)...]

作为 define_macros 的值列出的数据对中的每个 macro_name 和 macro_value 项目都是一个字符串，分别表示 C 语言预处理器宏定义的名称和值，等效于 C 语言预处理器指令：

```
#define macro_name macro_value
```

macro_value 还可以是 None，以获得与下面的 C 语言预处理器指令相同的效果：

```
#define macro_name
```

extra_compile_args = [*list of compile_arg strings*]

作为 extra_compile_args 的值列出的每个 compile_arg 字符串都被放置到每次运行 C 编译器时的命令行参数中。

extra_link_args = [*list of link_arg strings*]

作为 extra_link_args 的值列出的每个 link_arg 字符串都被放到运行链接器时的命令行参数中。

extra_objects = [*list of object_name strings*]

作为 extra_objects 的值列出的每个 object_name 字符串都指定了一个要添加到链接器的运行中的对象文件。不要将文件扩展名指定为该对象名称的一部分：distutils 将添加平台适用的文件扩展名（比如类 UNIX 平台上的.o 和 Windows 平台上的.obj）以帮助开发者保持跨平台可移植性。

include_dirs = [*list of directory_path strings*]

作为 include_dirs 的值列出的每个 directory_path 字符串都标识一个为编译器提供的目录，可以在这个目录中找到头文件。

libraries = [*list of library_name strings*]

作为 libraries 的值列出的每个 library_name 字符串都指定了一个要在运行链接器的时候添加的库。不要指定文件扩展名或任何前缀作为这个库的名称的一部分：在与链接器互操作时，distutils 将添加平台适用的文件扩展名和前缀（比如类 UNIX 平台上的.a 和前缀 lib，Windows 平台上的.lib）以帮助开发者保持跨平台可兼容性。

library_dirs = [*list of directory_path strings*]

作为 library_dirs 的值列出的每个 directory_path 字符串都标识了一个提供给编译器的目录，可以在这个目录中找到头文件。

runtime_library_dirs = [*list of directory_path strings*]

作为 runtime_library_dirs 的值列出的每个 directory_path 字符串标识了一个目录，可

以在运行时从这个目录中找到动态加载库。

undef_macros = [*list of macro_name strings*]

作为 undef_macros 的值列出的每个 macro_name 字符串都是一个 C 语言预处理器宏定义的名称，等效于下面的 C 语言预处理器指令：

```
#undef macro_name
```

setup.cfg 文件

distutils 可以让一个正在安装开发者的发布版本的用户在安装时指定许多选项。通常这个用户只需要在命令行上输入：

```
C:\> python setup.py install
```

不过，已经提到的指南 *Installing Python Modules* 中还介绍了许多其他安装方法。要想为安装选项提供推荐值，可以在开发者的发布版本的根目录下放置一个 setup.cfg 文件。setup.cfg 还可以为提供给编译时命令的选项提供适当的默认值。要想了解有关 setup.cfg 文件的格式和内容的更多详细信息，请参见已经提到的指南 *Distributing Python Modules*。

MANIFEST.in 和 MANIFEST 文件

在运行下面这行命令时：

```
python setup.py sdist
```

将会生成一个打包的源发布版本（通常，在 Windows 平台上是一个.zip 文件，在 UNIX 平台上是一个.tgz 文件，也叫作 tarball），在默认情况下，distutils 将在发布版本中插入：

- 开发者的 setup.py 文件的选项显式提到或者直接隐含的所有 Python 和 C 语言源代码，以及数据文件，参见本章的前面部分；
- 测试文件，位于发布版本根目录下的 test/test*.py；
- README.txt（如果有的话）、setup.cfg（如果有）和 setup.py 文件。

为了在源发布版本的.zip 文件或 tarball 文件中添加更多文件，可以在发布版本的根目录下放置一个名为 MANIFEST.in 的“清单模板”文件，文件中的行就是一些有关如何从要被放到发布版本中的文件列表中添加（include）或删除（prune）文件的规则，这些规则将按顺序执行。distutils 的 sdist 命令将在发布版本根目录下的一个名为 MANIFEST 的文本文件中生成一个要被放到源发布版本中的文件的准确列表。

使用 distutils 创建预编译发布版本

开发者使用 python setup.py sdist 创建的打包后的源发布版本是可以使用 distutils 生成的最有用的文件。但是，开发者还可以使用 python setup.py bdist 命令创建发布版本的预

编译形式，这样可以让特定平台的用户更容易使用发布版本。

对于纯发布版本，提供预编译形式对用户而言只是一个方便性问题。只要在开发者的平台的路径上可以使用所需的命令（比如 zip、gzip、bzip2 和 tar），就可以为任何平台创建预编译的纯的发布版本，甚至包括不同于开发者正在工作的平台。对于所有类型的系统平台，这些命令都可以从 Internet 上免费获得，因此，为了向想要安装开发者的发布版本的用户提供最大的方便，开发者可以很容易地准备这些工具。

对于非纯发布版本，提供预编译的形式可能并不仅仅只是一个方便性问题。按照定义，非纯发布版本包括不是纯的 Python 代码——通常是 C 代码。除非开发者提供了一个预编译形式，否则用户需要安装正确的 C 编译器以编译并安装开发者的发布版本。这在该 C 编译器是免费的和有 gcc 编译器的平台（现在，这意味着大多数类 UNIX 平台，包括 Mac OS X，其中 gcc 是该操作系统附带的免费 XCode IDE 的一部分）上并不是一个十分麻烦的问题。但是，在其他平台上（主要是 Windows），用于 Python 扩展模块的常规编译所需的 C 编译器都是商用的，而且费用昂贵。例如，在 Windows 上，Python 及其 C 语言编码的扩展模块使用的常规 C 编译器是 Microsoft Visual Studio（VS2003，用于 Python 2.4 和 Python 2.5）。尽管有可能使用其他编译器来替代，比如像 mingw32 和 cygwin 版本的 gcc 这样的免费编译器，但是，使用这样的可替代编译器（Python 在线指南给出了这些编译器）是相当麻烦的，尤其是对那些并不是非常有经验的程序员的终端用户。

因此，如果开发者想要自己的非纯发布版本在像 Windows 这样的平台上被广泛使用，强烈推荐开发者还创建一个预编译形式的可用版本。但是，除非开发者已经开发或者购买了高级的跨编译环境，否则只能在目标平台上编译一个非纯发布版本并将其打包成一个预编译形式。开发者还需要安装必要的 C 编译器。不过，在所有这些条件都满足时，distutils 可以使得这个过程非常简单。特别是使用下面的命令：

```
python setup.py bdist_wininst
```

这个命令可以创建一个.exe 文件，该文件是开发者的发布版本的 Windows 安装程序。如果开发者的发布版本是非纯的，则预编译的发布版本取决于特定的 Python 版本。distutils 将在其为开发者创建的.exe 安装程序的名称中反映这个事实。例如，假定开发者的发布版本的 name 元数据是 mydist，发布版本的 version 元数据是 0.1，并且开发者使用的 Python 版本是 2.4。在这种情况下，distutils 将编译一个名为 mydist-0.1.win32-py2.4.exe 的 Windows 安装程序。

27.2 py2exe

distutils 可以用来对 Python 扩展和应用程序进行打包。但是，终端用户只有在安装了 Python 之后才可以安装结果打包形式。这在 Windows 中终端用户想要运行单个安装程序以获得一个可以在 Windows 计算机上工作的应用程序时尤其是个问题。首先安装

Python，然后运行开发者的应用程序的安装程序，这种方式可能引起这样的终端用户的太大争议。

Thomas Heller 开发了一个简单的解决方案，也就是一个名为 py2exe 的 distutils 扩展工具，开发者可以从 http://starship.python.net/crew/theller/py2exe/ 免费下载这个工具。这个 URL 还包含有关 py2exe 的详细文档，如果开发者打算用高级方法使用 py2exe，本书建议开发者学习这个文档。但是，本章在其余章节中介绍的一些最简单的用法还可以满足更实际的需要。

在下载和安装了 py2exe（在安装了 Microsoft VS 2003 的 Windows 计算机上）之后，只需要添加下面这行代码：

```
import py2exe
```

将上面这行代码添加到正规的 distutils 脚本 setup.py 的起始位置。现在，除了其他 distutils 命令，开发者还有另一个选择。运行：

```
python setup.py py2exe
```

将在开发者安装的 Python 版本的根目录的一个子目录中创建和收集一个.exe 文件和一个或多个.dll 文件。例如，如果开发者的 Python 版本的 name 元数据是 myapp，则用来收集.exe 和.dll 文件的目录的名称为 dist\myapp\。setup.py 脚本中的 data_files 选项指定的所有文件都被放置到 dist\myapp\子目录中。.exe 文件对应于开发者的应用程序的 scripts 关键字参数值中的第一个或唯一一个条目，并包含 setup.py 指定或隐式使用的所有字节代码编译形式的 Python 模块和包。这些.dll 文件中至少有一个 Python 动态加载库——例如，python24.dll，如果开发者使用的是 Python 2.4——和开发者的应用程序需要的任意其他.pyd 或.dll 文件，除 py2exe 已知的.dll 系统文件（也就是，保证可以在任意 Windows 安装软件上使用）之外。

py2exe 没有为简单的发布版本和安装提供直接的方法以收集 dist\myapp\目录下的内容。开发者有几种安装选择，其范围从.zip 文件（该文件也可能是一个扩展名为.exe 的文件，并可以进行自解压，根据开发者选择的 ZIP 文件处理工具的不同，处理方法也有所不同）一直到专业的 Windows 安装程序构造系统，比如像 Wise 和 InstallShield 这样的公司的安装软件。一个特别值得考虑的选择是 Inno Setup，这是一个免费的，具有专业软件质量的安装程序构造系统（参见 http://www.jrsoftware.org/isinfo.php）。因为，为了方便终端用户的安装，要被打包的文件是一个.exe 文件、一个或多个.dll 文件，可能还有子目录中的一些数据，现在，这个问题变得完全与 Python 无关。开发者可以对这些文件进行打包和重新发布，就像这些文件最初是从任何其他编程语言编写的源代码编译而来的。

27.3 py2app

py2app（http://undefined.org/python/py2app.html）是一个 distutil 扩展，可以为 Mac 编译

单独的 Python 应用程序。py2app 是与 PyObjC（http://pyobjc.sourceforge.net/）一起发布的，PyObjC 也就是 Python/Object C 桥接程序，PyObjC 提供了一种完美的方法以使用 Python 中的 Cocoa 接口创建 Mac 一次性；但是，py2app 也完全兼容 Python 的所有主流跨平台 GUI 工具包，包括 Tkinter、wxPython、pygame 和 PyQt。此外，py2app 还可以用来直接编译安装程序包（.mpkg 文件）。请参见上面的 URL 以了解所有实际使用细节。

27.4 cx_Freeze

cx_Freeze（http://starship.python.net/crew/atuining/cx_Freeze/）是一个单独的工具（而不是一个 distutils 扩展），可以为 Windows 和 Linux 编译单独的 Python 应用程序。请参见上面的 URL 以了解所有实际使用细节。

27.5 PyInstaller

PyInstaller（http://pyinstaller.hpcf.upr.edu/cgi-bin/trac.cgi）是另一个单独的工具，可以为 Windows、Linux 和 Irix 编译单独的 Python 应用程序。参见上面这个 URL 以了解所有实际使用细节。

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

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

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

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

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

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

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

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

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

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

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

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

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

Programming/Python

PYTHON技术手册(第2版)

“无论对于初学者、中级用户还是专家，本书正是他们寻觅的关于Python语言习惯用法的优秀参考书。本书根据Python标准库的用法对其进行了介绍，并辅以简单、直观的示例。清晰的写作风格使得阅读本书成为一种乐趣。”

——Jeffery D. Collins，微软公司开发主管

本书为Python程序员提供了丰富的参考信息，当Python程序员需要在回忆或解读这种开源语言的语法及其众多强大的功能模块时，可能会因缺少文档而需要获得帮助，这时可以参考本书中的内容。这本技术手册可以用来方便地查找经常需要使用的信息，这些信息不仅仅是关于Python语言本身，还包括最常使用的一部分标准库和最重要的第三方扩展。

本书内容包括：

- Python语言语法的快速指南；
- Python面向对象编程的说明；
- 迭代器、生成器、异常、模块、包、字符串和正则表达式；
- Python的内置类型和函数，以及关键模块的快速参考；
- 重要的第三方扩展的参考资料，比如Numeric和Tkinter；
- 有关扩展和嵌入Python的信息。

O'REILLY®

www.oreilly.com

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行

This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

分类建议：计算机/程序设计/Python

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-22583-2

ISBN 978-7-115-22583-2

定价：89.00 元