



Beginning Python: Using Python 2.6 and Python 3.1

# Python

# 编程入门经典

(美) James Payne 著  
张春晖 译



清华大学出版社

# Python 编程入门经典

(美) James Payne      著  
张春晖                译

清华大学出版社

北 京



James Payne

Beginning Python: Using Python 2.6 and Python 3.1

EISBN: 978-0-470-41463-7

Copyright © 2010 by Wiley Publishing, Inc., Indianapolis, Indiana

All Rights Reserved. This translation published under license.

本书中文简体字版由 Wiley Publishing, Inc. 授权清华大学出版社出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号 图字：01-2010-1678

本书封面贴有 Wiley 公司防伪标签，无标签者不得销售。

版权所有，侵权必究。举报电话：010-62782989 13701121933

图书在版编目(CIP)数据

Python 编程入门经典/(美) 佩恩(Payne, J.) 著；张春晖 译. —北京：清华大学出版社，2011.7

书名原文：Beginning Python: Using Python 2.6 and Python 3.1

ISBN 978-7-302-25709-7

I. P… II. ①佩… ②张… III. 软件工具—程序设计 IV. TP311.56

中国版本图书馆 CIP 数据核字(2011)第 106864 号

责任编辑：王 军 赵利通

装帧设计：孔祥丰

责任校对：胡雁翎

责任印制：王秀菊

出版发行：清华大学出版社

地 址：北京清华大学学研大厦 A 座

<http://www.tup.com.cn>

邮 编：100084

社 总 机：010-62770175

邮 购：010-62786544

投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈：010-62772015, zhiliang@tup.tsinghua.edu.cn

印 刷 者：清华大学印刷厂

装 订 者：三河市漂源装订厂

经 销：全国新华书店

开 本：185×260 印 张：34.75 字 数：846 千字

版 次：2011 年 7 月第 1 版 印 次：2011 年 7 月第 1 次印刷

印 数：1~4000

定 价：68.00 元

产品编号：033391-01

# 作者简介

James Payne 居住在佛罗里达州马盖特市，他是 Developer Shed 公司的主编，从 7 岁起就开始写作和编程。James 精通多门语言，已经撰写了 400 余篇文章，囊括了几乎每一种主流编程语言。James 的工作是使用 Python 为金融业开发专有软件，在业余时间他喜欢涉猎 Django 方面的知识。



# 致 谢

我感谢在 Developer Shed 的同事：Jack 和 Jay Kim 持续的敌意使我备受鼓舞；Charles Fagundes 使我成为一名编辑，而不仅仅是一个作者；Keith Lee 为我提供了编写代码的支持；特别要感谢 Jenny Ruggieri，他使我得到了这份工作，所以有了这本书的面世。最后，还要感谢所有为这本书的前期版本付出艰辛劳动的人们，他们的工作是本书的基础。还要感谢编辑 Carol Long、Jenny Watson、Ed Connor 以及 Chris McAvoy，他们帮助我在最终期限前完成本书。

我还想感谢 Guido Van Rossum，没有他，也就没有 Python 语言可写。



# 前言

欢迎使用 Python 3.1!

我使用 Python 已有 10 年左右，每一个新版本都使我越来越喜欢这门语言。版本 3.1 也不例外。如果您是 Python 的初学者，不必担心，我会一路引领着您。相反，如果您是一名希望尝试新版本的 Python 老手，可以不必在已掌握的知识上浪费时间，而是直接学习所需的知识，因为本书的结构十分清晰。

之所以写这本书，是因为我热爱 Python。我非常喜爱它！我想和您一起分享我对 Python 的热爱。您可能也会像我一样越来越喜爱它。

## 本书读者对象

如果您具备计算机知识，并且想学习一门有趣的程序语言来更好地控制计算机，那么本书适合您。

如果您是系统管理员，希望学习一门优秀的语言来更好地管理和配置系统和网络，那么本书适合您。

如果您已经了解 Python，但是想知道版本 3.1 中有哪些很酷的新功能，那么本书适合您。

总之，本书适合于所有热衷于使用功能最丰富且十分易用的最新版本 Python 3.1 进行编程的读者。

## 本书主要内容

本书主要介绍 Python 3.1。Python 3.1 发布于 2009 年，是 Python 程序语言的最新主版本。由于 Python 是一门跨平台的语言，本书中的内容和示例适用于任何平台(除非特别指出的例外情况)。在可能独立于平台的情况下，本书中的示例都尽量设计成跨平台的。

此外，由于 Python 3.1 相对较新，并非所有支持库都被更新到可以在 Python 3.x 下工作。在这种情形下，如果某些理论需要详细解释，本书将使用 Python 2.6 代替 Python 3.1。

## 本书结构

就像所有入门书籍一样，本书首先对语言进行了介绍。从这里开始，会接触到语言的

核心，进而接触到更加高级的专门主题。本书分为如下 4 个部分。

## 第 I 部分——初步了解 Python

如标题所示，第 I 部分让您初步了解 Python。

### 编程基础和字符串

本章首先介绍 Python，探讨 Python 是什么，为什么它如此有用和强大。另外，还探讨 Python 的开发历史和最新的版本，后者正是本书的讨论重点。您会了解到 Python 所涉及的范围，以及 Python 可能参与的应用开发领域。最后，您将开始操作自己的第一种数据类型：字符串。

### 数值与运算符

本章会介绍数值和运算符的基本知识。您将会学习不同类型的数值，如何执行简单的和复杂的公式，以及如何使用各种运算符。还将学到运算符的优先级以及数值的格式化。

### 变量

编程语言最终帮助程序员管理不同类型的信息，换言之，就是管理数据。理解数据类型并且知道 Python 如何表示它们对于 Python 编程非常必要。本章将帮助您理解在 Python 中表示不同数据类型的最佳方式。

## 第 II 部分——Python 语言和标准库

当然，使用一门语言最需要的是了解语言本身，熟悉它的语法和模块。本部分首先介绍数据类型和变量，并逐步介绍其他概念，直到获得开发功能完善的 Python 程序所需的全部知识。

注意，应该按顺序阅读本部分的章节，因为每一章都建立在前一章的基础之上。

### 做出决策

最终，程序将需要在某个位置做出决策：是选择这条路径，还是选择另一条路径？如果选择那条路径，将会发生什么？在本章中将学到如何比较数据，例如判断某个值是否比另一个值大，并且将学习如何利用循环来完成重复性的任务。

### 函数

本章将介绍函数编程，这将帮助您扩展 Python 知识。函数允许使用者利用强大的概念，例如参数传递和代码重用。本章将介绍如何利用函数使编写的代码更加高效和灵活。

### 类和对象

本章将介绍对象的概念。您将学习如何定义类和如何在类中创建对象和编写方法，还将学到有关对象作用域的知识。

## 组织程序

当程序变大时，需要把它们分成单独的几部分。本章将讨论 Python 模块，并探索包的概念。包并不复杂，它仅是若干模块的集合。

## 文件和目录

日常编程中一个重要的部分是学习如何处理文件和目录。本章着重介绍通常如何创建、修改和处理文件。另外，还将介绍如何从文件中获取数据，以及如何与各种各样的目录交互。

## Python 语言的其他特性

本章将介绍 Python 语言提供的其他特性，包括如何使用列表做出决策，如何使用字典进行字符串替换，以及一些有特色的模块。

## 创建模块

通过允许重用代码片段，模块能够帮助节省时间。它能减少发生错误的机率，因为当前用到的模块可能之前已经被测试和使用过很多次。本章将介绍如何创建自己的模块，以及如何导入并处理已存在的模块，这些已经存在的模块使得 Python 在应用中功能非常强大。

## 文本处理

在编程过程中，可以利用文本做很多事情，而且本质上，文本是与用户进行有效沟通的关键。毕竟，如果没有文本，就只能使用图片了。本章将介绍如何用多种方法处理文本，包括利用正则表达式、搜索文件以及搜索特定类型的文件等。

## 第III部分——开始使用 Python

现在您已经知道了 Python 是什么，以及如何利用这门语言进行工作，接下来应该做什么呢？本部分探究了您会遇到或者想了解的许多编程主题。可以顺序阅读本部分的各章，也可以按其他任意顺序阅读，因为它们是彼此独立的。

## 测试

在把程序交付给用户之前，对程序进行测试，这是保证它能正常工作的唯一途径。本章不仅会介绍正确测试程序的一些基本概念，还会介绍一些可用的测试工具和测试框架。

## 使用 Python 编写 GUI

目前为止，本书中的程序都是通过命令行方式工作。本章将介绍 GUI 编程的概念。您将使用 Tkinter 创建几个 GUI 程序，Tkinter 是 Python 程序员最常用的 GUI 工具包。

## 访问数据库

数据库存储了程序可以使用的各种信息。它也可以作为存储信息的地方，以便日后可

以从中检索出特定任务所需的数据。本章将介绍各种类型的数据库以及如何利用它们进行工作。

### 使用 Python 处理 XML

XML 是处理 Internet 数据的强大工具。本章将介绍 XML 的基础知识,包括模式与 DTD 之间的区别、基本的语法、如何创建和验证 XML 以及一些高级的主题(例如使用 lxml)。

### 网络编程

Internet 已经潜移默化地进入了我们的日常生活,并逐渐变成必需品,而不是一种特权。本章将介绍如何使用程序发送 E-mail, 以及如何允许用户通过 Web 进行交流。

### 用 C 扩展编程

本章深入介绍如何使用 C 语言编程,包括如何利用 C 的框架和模块, C 的基础知识, 以及如何从 Python 向 C 传递参数,并在之后将值返回给 Python。

### 数值编程

本书的开始部分简单介绍了数值,现在是进一步钻研的时候了。本章将深入介绍整型、浮点型、复数和数组的全部知识, 以及如何使用内置的数学函数和模块。

### Django 简介

Django 是用 Python 编写的 Web 应用程序框架,它利用了模型-视图-控制器模式。Django 最初被用来管理新闻 Web 站点,但由于简单易用而变得流行起来,它使得程序员可以用简单的方式创建复杂的 Web 站点,包括以数据库为中心的 Web 站点。本章将介绍 Django 的基础知识。

### Web 应用程序和 Web 服务

本章将介绍 Web 应用程序和 Web 服务的基础知识,例如 REST 架构, 以及如何处理 HTTP 请求和响应。

### 集成 Java 与 Python

本章将介绍 Java 的基础知识,这样可以在研究如何结合 Python 和 Java 之前,打好坚实的基础。还将介绍各种帮助在 Python 中使用 Java 的模块, 以及如何创建简单但有效的应用程序。

## 第IV部分：附录

本书结尾的附录可以进一步扩展您对 Python 的了解,激发您对 Python 的学习兴趣。

- 习题答案
- 在线资源
- Python 3.1 的新特性

- 术语表

## 使用本书的条件

使用本书有一些条件。在此给出下面列出的推荐，因为 Python 本身可在多种不同的平台上运行。然而，本书的前几章假定您可以使用 GUI，例如在 Windows、Mac OS X、UNIX 和 Linux 上的 X Window 等系统上可用的 GUI。自然地，一些章节(例如介绍 GUI 的那一章)也需要 GUI，而如果有网络连接，涉及网络的那些章节会更有意义。

下面是建议的最低配置需求：

- 一台 PC 机，运行 Linux、BSD UNIX 或 Windows 系统，频率在 500MHz 以上；或者运行 Mac OS X 版本 10.2 或更新版本的 G3 或更新的 Macintosh。
- 至少 256MB 内存。
- 所用平台具有图形用户界面。
- 对所用计算机有必要的权限，以安装必需的软件。
- 可以访问 TCP/IP 网络，例如 Internet 或者校园网。
- 可以通过 Internet 下载所需软件。

## 源代码

在阅读本书提供的代码时既可以亲自输入所有代码，也可以使用随书提供的代码文件。本书所有代码均可以从 <http://www.wrox.com>(或 <http://www.tupwk.com.cn/downpage>)网站下载。进入该网站后，请根据本书的书名查找本书(读者既可以使用搜索框进行查找，也可以使用书名列表进行查找)，然后单击本书详细内容页面上提供的 Download Code 链接，就可以下载本书提供的所有代码。

注意：

由于许多书籍名称与本书类似，因此也可以通过 ISBN 进行查找，本书英文版的 ISBN 为：978-0-470-41463-7。

下载代码后，可以利用一种压缩工具解压代码。此外，还可以通过访问网站 <http://www.wrox.com/dynamic/books/download.aspx> 中提供的 Wrox 代码下载页面来获取本书提供的代码，也可以下载 Wrox 出版的其他书籍提供的代码。

## 勘误表

为了避免本书文字和代码中存在错误，我们已经竭尽全力。然而，就如世界上不存在完美无缺的事物，本书仍然可能存在错误。如果您在我们编写的书籍中发现了诸如拼写错误或代码缺陷等问题，请告诉我们，我们对此表示感谢。利用勘误表反馈错误信息，可以为其他

读者节省大量时间，同时，我们也能够受益于您的帮助，编写出质量更高的专业著作。

如果需要参考本书的勘误表，请在网站 <http://www.wrox.com> 中用搜索框或书名列表查找本书书名。然后，在本书的详细内容页面上，单击 Book Errata 链接。在随后显示的页面中，可以看到与本书相关的所有勘误信息，这些信息是由读者提交、并由 Wrox 的编辑们加上的。通过访问 [www.wrox.com/misc-pages/booklist.shtml](http://www.wrox.com/misc-pages/booklist.shtml)，还可以看到 Wrox 出版的所有书籍的勘误表。

如果没有在 Book Errata 页面上找到自己发现的错误，那么请转到页面 [www.wrox.com/contact/techsupport.shtml](http://www.wrox.com/contact/techsupport.shtml)，针对您所发现的每一项错误填写表格，并将表格发给我们，我们将对表格内容进行认真审查，如果确实是我们书中的错误，那么我们将在该书的 Book Errata 页面上标明该错误信息，并在该书的后续版本中改正相关错误。

## 关于 p2p.wrox.com

如果您希望能够与作者进行讨论，或希望能够参与读者的共同讨论，那么请加入 p2p.wrox.com 的论坛。这个论坛是一个基于 Web 的系统，您可以在论坛发表与 Wrox 出版的书籍有关的技术信息，并与其他读者和技术用户进行讨论。论坛提供了订阅功能，可以将与您所选主题相关的新帖子定期发送到您的电子邮箱。Wrox 的作者、编辑、业界专家以及其他读者都会参与论坛中的讨论。

您可以在 <http://p2p.wrox.com> 参与多个论坛的讨论，这些论坛不仅能够帮助您更好地理解本书，还有助于您更好地开发应用程序。如果希望加入论坛，请按照以下步骤执行：

- (1) 进入 <http://p2p.wrox.com> 页面，单击 Register 链接。
- (2) 阅读使用条款，然后单击 Agree。
- (3) 填写必要的信息(以及想要提供的可选信息)，然后单击 Submit。
- (4) 随后会收到一封电子邮件，邮件中说明了如何验证账号并完成整个加入过程。

### 注意：

要阅读论坛信息，无须加入 P2P。但是如果需要发表主题或发表回复，那么必须加入论坛。

成功加入论坛后，就可以发表新主题了。此外，还可以回复其他主题。您在任何时间都可以阅读论坛信息。如果需要论坛将新的信息发送到自己的电子邮箱，那么可以单击论坛列表中论坛名称旁的 Subscribe to this Forum 图标完成该功能设置。

如果需要获得更多与 Wrox P2P 相关的信息，请阅读 P2P FAQs，这样可以获得大量与 P2P 和 Wrox 出版的书籍相关的具体信息。阅读 FAQs 时，请单击 P2P 页面上的 FAQs 链接。

# 目 录

第 I 部分 初步了解 Python	
第 1 章 编程基础和字符串 .....	3
1.1 编程与使用计算机的区别 .....	3
1.1.1 编程的一致性 .....	3
1.1.2 编程的可控性 .....	4
1.1.3 程序要应对变化 .....	4
1.1.4 小结 .....	4
1.2 准备工作 .....	4
1.2.1 在非 Windows 系统上安装 Python 3.1 .....	5
1.2.2 使用 Python Shell .....	5
1.3 开始使用 Python——字符串 .....	6
1.3.1 字符串概述 .....	6
1.3.2 为什么需要引号 .....	6
1.3.3 为什么有 3 种类型的引号 .....	7
1.3.4 使用 print() 函数 .....	7
1.3.5 理解不同的引号 .....	8
1.4 串联两个字符串 .....	10
1.5 用不同的方法串联字符串 .....	11
1.6 本章小结 .....	12
1.7 习题 .....	13
第 2 章 数值与运算符 .....	15
2.1 不同类型的数值 .....	15
2.2 程序文件 .....	17
2.2.1 使用不同的类型 .....	18
2.2.2 基本算术 .....	20
2.2.3 一些惊喜 .....	22
2.3 使用数值 .....	23
2.3.1 求值顺序 .....	23
2.3.2 数值格式 .....	24
2.3.3 可能会发生的错误 .....	25
2.3.4 几个不寻常的例子 .....	26
2.4 本章小结 .....	26
2.5 习题 .....	27
第 3 章 变量 .....	29
3.1 引用数据——使用数据的 名称 .....	29
3.1.1 使用名称修改数据 .....	30
3.1.2 复制数据 .....	31
3.1.3 禁用的名称以及一些规则 .....	31
3.2 使用更多的内置类型 .....	31
3.2.1 元组——不可更改的数据 序列 .....	32
3.2.2 列表——可以更改的数据 序列 .....	35
3.2.3 字典——以名称索引的分组 数据 .....	36
3.2.4 像列表一样处理字符串 .....	38
3.2.5 特殊类型 .....	39
3.3 序列的其他共有属性 .....	40
3.3.1 引用最后一个元素 .....	40
3.3.2 序列的范围 .....	41
3.3.3 通过附加序列增长列表 .....	41
3.3.4 使用列表临时存储数据 .....	42
3.3.5 处理集合 .....	43
3.4 本章小结 .....	44
3.5 习题 .....	44

## 第 II 部分 Python 语言和标准库

<b>第 4 章 做出决策</b>	<b>49</b>
4.1 比较两个值是否相等	49
4.2 比较两个值是否不相等	51
4.3 比较两个值的大小	51
4.4 对真值和假值取反	53
4.5 观察多个比较运算的结果	54
4.6 循环	57
4.6.1 重复执行操作	57
4.6.2 终止循环	59
4.7 处理错误	62
4.8 本章小结	64
4.9 习题	65
<b>第 5 章 函数</b>	<b>67</b>
5.1 将程序放在单独的文件中	67
5.2 函数：在一个名称下聚集代码	69
5.2.1 选择名称	70
5.2.2 在函数中描述函数	70
5.2.3 不同的位置相同的名称	72
5.2.4 添加注释	73
5.2.5 要求函数使用提供的值	73
5.2.6 检查参数	75
5.2.7 为参数设置默认值	77
5.2.8 在函数中调用其他函数	78
5.2.9 函数嵌套函数	80
5.2.10 用自己的词语标记错误	81
5.3 函数的层次	81
5.4 本章小结	82
5.5 习题	83
<b>第 6 章 类与对象</b>	<b>85</b>
6.1 考虑编程	85
6.1.1 对象的含义	85
6.1.2 已经了解的对象	85
6.1.3 展望：如何使用对象	87

6.2 定义类	87
6.2.1 如何创建对象	87
6.2.2 对象和它们的作用域	95
6.3 本章小结	98
6.4 习题	99
<b>第 7 章 组织程序</b>	<b>101</b>
7.1 模块	102
7.1.1 导入可用模块	102
7.1.2 通过已有模块创建新模块	102
7.1.3 从命令行开始使用模块	104
7.1.4 改变导入方式	106
7.2 包	106
7.3 模块和包	108
7.3.1 将所有内容引入当前作用域	108
7.3.2 重新导入模块和包	109
7.4 测试模块和包的基础知识	111
7.5 本章小结	112
7.6 习题	112
<b>第 8 章 文件和目录</b>	<b>115</b>
8.1 文件对象	115
8.1.1 编写文本文件	116
8.1.2 向文件中追加文本	116
8.1.3 读文本文件	117
8.1.4 文件异常	119
8.2 路径和目录	119
8.3 os 中的异常	119
8.3.1 路径	120
8.3.2 目录内容	122
8.3.3 获取文件信息	123
8.3.4 重命名、移动、复制和删除文件	125
8.3.5 示例：轮换文件	126
8.3.6 创建和删除目录	127
8.3.7 通配	128

8.4 本章小结 .....	129	10.5 创建一个完整的模块 .....	166
8.5 习题 .....	129	10.6 安装模块 .....	171
<b>第 9 章 Python 语言的其他特性 .....</b>	<b>131</b>	10.7 本章小结 .....	174
9.1 lambda 和 filter: 简单匿名 函数 .....	131	10.8 习题 .....	175
9.2 Map: 短路循环 .....	132	<b>第 11 章 文本处理 .....</b>	<b>177</b>
9.3 在列表中做出决策—— 列表解析 .....	133	11.1 文本处理的用途 .....	177
9.4 为循环生成迭代器 .....	133	11.1.1 搜索文件 .....	178
9.5 使用字典的特殊字符串 替换 .....	135	11.1.2 日志剪辑 .....	179
9.6 重要模块 .....	137	11.1.3 邮件筛选 .....	179
9.6.1 getopt——从命令行中得到 选项 .....	137	11.2 使用 os 模块导航文件 系统 .....	179
9.6.2 使用一个以上的进程 .....	139	11.3 使用正则表达式和 re 模块 .....	185
9.6.3 线程——在相同的进程中 完成多个工作 .....	141	11.4 本章小结 .....	189
9.7 本章小结 .....	143	11.5 习题 .....	189
9.8 习题 .....	144	<b>第 III 部分 开始使用 Python</b>	
<b>第 10 章 创建模块 .....</b>	<b>145</b>	<b>第 12 章 测试 .....</b>	<b>193</b>
10.1 研究模块 .....	145	12.1 断言 .....	193
10.1.1 导入模块 .....	147	12.2 测试用例和测试套件 .....	195
10.1.2 查找模块 .....	147	12.3 测试装置 .....	198
10.1.3 理解模块 .....	148	12.4 用极限编程整合 .....	201
10.2 创建模块和包 .....	149	12.4.1 用 Python 实现搜索实用 程序 .....	202
10.3 使用类 .....	150	12.4.2 一个更加强大的 Python 搜索 .....	207
10.3.1 定义面向对象编程 .....	151	12.5 软件生命周期中的正规 测试 .....	210
10.3.2 创建类 .....	151	12.6 本章小结 .....	210
10.3.3 扩展已有的类 .....	152	<b>第 13 章 使用 Python 编写 GUI .....</b>	<b>213</b>
10.4 完成模块 .....	153	13.1 Python 的 GUI 编程工具箱 .....	213
10.4.1 定义模块特定的错误 .....	154	13.2 Tkinter 简介 .....	215
10.4.2 选择导出哪些内容 .....	154	13.3 用 Tkinter 创建 GUI 小组件 .....	215
10.4.3 为模块建立文档 .....	155	13.3.1 改变小组件的尺寸 .....	215
10.4.4 测试模块 .....	164		
10.4.5 将模块作为程序运行 .....	165		

13.3.2	配置小组件选项	216	15.2	模式/DTD	251
13.3.3	使用小组件	217	15.2.1	文档模型的用途	251
13.3.4	创建布局	217	15.2.2	是否需要文档模型	252
13.3.5	填充顺序	218	15.3	文档类型定义	252
13.3.6	控制小组件的外观	219	15.3.1	DTD 示例	252
13.3.7	单选按钮和复选框	220	15.3.2	DTD 不完全是 XML	253
13.3.8	对话框	221	15.3.3	DTD 的局限性	253
13.3.9	其他小组件类型	222	15.4	模式	254
13.4	本章小结	222	15.4.1	示例模式	254
13.5	习题	223	15.4.2	模式是纯粹的 XML	254
第 14 章	访问数据库	225	15.4.3	模式具有层次	255
14.1	使用 dbm 持久字典	226	15.4.4	模式的其他优点	255
14.1.1	选择 dbm 模块	226	15.5	XPath	255
14.1.2	创建持久字典	227	15.6	HTML 是 XML 的子集	256
14.1.3	访问持久字典	228	15.6.1	HTML DTD	256
14.1.4	dbm 与关系数据库的适用 场合	230	15.6.2	HTMLParser	256
14.2	使用关系数据库	231	15.7	Python 中可用的 XML 库	257
14.2.1	编写 SQL 语句	232	15.8	SAX 的含义	257
14.2.2	定义表	234	15.8.1	基于流	258
14.2.3	建立数据库	235	15.8.2	事件驱动	258
14.3	使用 Python 的数据库 API	237	15.8.3	DOM 的含义	258
14.3.1	下载各个模块	237	15.8.4	内存中访问	258
14.3.2	创建连接	238	15.9	使用 SAX 或者 DOM 的 理由	259
14.3.3	使用游标	238	15.9.1	能力权衡	259
14.3.4	使用事务并提交结果	245	15.9.2	内存考虑	259
14.3.5	检查模块的功能和元 数据	246	15.9.3	速度考虑	259
14.3.6	处理错误	246	15.10	Python 中可用的 SAX 和 DOM 解析器	259
14.4	本章小结	247	15.10.1	xml.sax	259
14.5	习题	248	15.10.2	xml.dom.minidom	260
第 15 章	使用 Python 处理 XML	249	15.11	XSLT 简介	262
15.1	XML 的含义	249	15.11.1	XSLT 是 XML	262
15.1.1	层次标记语言	249	15.11.2	转换和格式语言	263
15.1.2	一组标准	251	15.11.3	函数式、模板驱动	263
			15.12	lxml 简介	263



15.13	元素类 .....	263	16.4.11	基于 select 的单线程 多任务 .....	312
15.14	使用 lxml 解析 .....	266	16.5	其他主题 .....	313
15.15	本章小结 .....	267	16.5.1	协议设计的多种考虑 .....	313
15.16	习题 .....	267	16.5.2	对等架构 .....	314
<b>第 16 章</b>	<b>网络编程 .....</b>	<b>269</b>	16.6	本章小结 .....	314
16.1	理解协议 .....	271	16.7	习题 .....	315
16.1.1	比较协议和程序语言 .....	271	<b>第 17 章</b>	<b>用 C 扩展编程 .....</b>	<b>317</b>
16.1.2	Internet 协议栈 .....	272	17.1	扩展模块概述 .....	318
16.1.3	Internet 协议简介 .....	273	17.2	构建并安装扩展模块 .....	320
16.2	发送电子邮件 .....	274	17.3	从 Python 向 C 传递参数 .....	322
16.2.1	E-mail 文件格式 .....	275	17.4	从 C 向 Python 返回值 .....	325
16.2.2	MIME 消息 .....	276	17.5	LAME 项目 .....	326
16.2.3	使用 SMTP 和 smtplib 发送 邮件 .....	284	17.6	LAME 扩展模块 .....	330
16.3	检索 Internet 邮件 .....	286	17.7	在 C 代码中使用 Python 对象 .....	342
16.3.1	使用 mailbox 解析本地 邮筒 .....	286	17.8	本章小结 .....	345
16.3.2	使用 poplib 从 POP3 服务器获取邮件 .....	288	17.9	习题 .....	346
16.3.3	使用 imaplib 从 IMAP 服务器获取邮件 .....	290	<b>第 18 章</b>	<b>数值编程 .....</b>	<b>347</b>
16.3.4	安全的 POP3 和 IMAP .....	294	18.1	Python 语言中的数值 .....	347
16.3.5	Webmail 应用程序不是 E-mail 应用程序 .....	294	18.1.1	整数 .....	348
16.4	套接字编程 .....	294	18.1.2	长整数 .....	348
16.4.1	套接字简介 .....	295	18.1.3	浮点数 .....	349
16.4.2	绑定到外部主机名 .....	297	18.1.4	格式化数值 .....	350
16.4.3	镜像服务器 .....	298	18.1.5	作为数值的字符 .....	352
16.4.4	镜像客户端 .....	299	18.2	数学 .....	353
16.4.5	套接字服务器 .....	300	18.2.1	算术运算 .....	354
16.4.6	多线程服务器 .....	302	18.2.2	内置数学函数 .....	355
16.4.7	Python 聊天服务器 .....	303	18.3	复数 .....	357
16.4.8	设计 Python 聊天 服务器 .....	303	18.4	数组 .....	359
16.4.9	Python 聊天服务器协议 .....	304	18.5	本章小结 .....	363
16.4.10	Python 聊天客户端 .....	309	18.6	习题 .....	363
			<b>第 19 章</b>	<b>Django 简介 .....</b>	<b>365</b>
			19.1	框架的定义以及使用框架的 理由 .....	365

19.2	Web 框架的其他功能	366
19.3	Django 发展史	367
19.4	理解 Django 的架构	368
19.4.1	项目初始设置	368
19.4.2	创建视图	371
19.5	使用模板	373
19.6	使用模板和视图	375
19.6.1	模型	378
19.6.2	创建模型的第一步—— 配置数据库设置	379
19.7	创建模型：创建一个应用 程序	380
19.8	本章小结	382
19.9	习题	383
第 20 章	Web 应用程序与 Web 服务	385
20.1	REST: Web 架构	386
20.1.1	REST 的特性	386
20.1.2	REST 操作	388
20.2	HTTP: REST 的实际应用	388
20.2.1	可见的 Web 服务器	390
20.2.2	HTTP 请求	393
20.2.3	HTTP 响应	393
20.3	CGI: 将脚本变为 Web 应用 程序	395
20.3.1	Web 服务器与 CGI 脚本的 协议	397
20.3.2	CGI 的特殊环境变量	397
20.3.3	通过 HTML 表单接收用户 输入	399
20.4	HTML 表单的有限词汇	400
20.5	访问表单值的安全性	401
20.6	构建 wiki	405
20.6.1	BittyWiki 核心库	406
20.6.2	BittyWiki 的 Web 接口	409
20.7	Web 服务	418
20.8	REST Web 服务	419
20.8.1	REST 快速入门: 在 Amazon.com 上寻找 廉价商品	419
20.8.2	WishListBargainFinder 简介	422
20.8.3	向 BittyWiki 提供 REST API	425
20.8.4	使用 REST Web 服务 实现搜索和替换功能	428
20.9	XML-RPC	432
20.9.1	XML-RPC 请求	434
20.9.2	XML-RPC 中的数据 表示	434
20.9.3	XML-RPC 响应	435
20.9.4	错误处理机制	436
20.9.5	通过 XML-RPC 展示 BittyWiki API	437
20.9.6	基于 XML-RPC Web 服务 的 wiki 搜索和替换	440
20.10	SOAP	442
20.10.1	SOAP 快速入门	442
20.10.2	SOAP 请求	443
20.10.3	SOAP 响应	444
20.10.4	错误处理机制	444
20.10.5	展示一个 BittyWiki 的 SOAP 接口	445
20.10.6	基于 SOAP Web 服务的 wiki 搜索和替换	447
20.11	为 Web 服务 API 创建 文档	449
20.11.1	人类可读的 API 文档	449
20.11.2	XML-RPC 自省 API	450
20.11.3	WSDL	451
20.12	选择 Web 服务标准	455
20.13	Web 服务礼仪	456

20.13.1 Web 服务的使用者 礼仪 .....	456	21.7 集成 Java 和 Jython .....	466
20.13.2 Web 服务的创建者 礼仪 .....	456	21.7.1 在 Jython 中使用 Java 类 .....	466
20.13.3 将 Web 应用程序作为 Web 服务使用 .....	456	21.7.2 从 Jython 访问数据库 .....	471
20.14 本章小结 .....	456	21.7.3 通过 Jython 编写 Java EE servlet .....	477
20.15 习题 .....	457	21.7.4 选择 Jython 开发工具 .....	483
<b>第 21 章 集成 Java 与 Python</b> .....	<b>459</b>	21.8 使用 Jython 进行测试 .....	483
21.1 在 Java 应用程序中编写 脚本 .....	460	21.9 嵌入 Jython 解释器 .....	484
21.2 比较各种 Python 实现 .....	461	21.10 处理 C-Python 和 Jython 之间的差异 .....	487
21.3 安装 Jython .....	461	21.11 本章小结 .....	488
21.4 运行 Jython .....	461	21.12 习题 .....	488
21.4.1 交互地运行 Jython .....	461		
21.4.2 运行 Jython 脚本 .....	463		
21.4.3 控制 jython 脚本 .....	463		
21.4.4 创建可执行命令 .....	464		
21.5 独立运行 Jython .....	465		
21.6 打包基于 Jython 的应用 程序 .....	466		

## 第IV部分 附 录

附录 A 习题答案 .....	491
附录 B 在线资源 .....	519
附录 C Python 3.1 的新特性 .....	521
附录 D 术语表 .....	527



# 创建健壮、可靠、可重用的Python应用程序

作为一门面向对象的开源编程语言，Python易于理解和扩展，并且使用起来非常方便。《Python编程入门经典》涵盖了Python的方方面面，通过学习本书，读者可以立即使用Python编写程序。作者James Payne从Python语言最基本的概念入手，重点介绍了Python 2.6 和3.1这两个版本的实际应用。通过对一些现有的Python程序进行深入分析，读者能够快速上手。本书涵盖的主题从字符串、列表和字典一直到类、对象和模块。掌握这些内容后，读者将学会如何迅速而自信地创建健壮、可靠而又可重用的Python应用程序。

## 主要内容

- ◆ 介绍用于存储和操作数据的变量的概念
- ◆ 讨论用于读写数据的文件和输入/输出
- ◆ 总结常被忽视的Python功能
- ◆ 深入研究如何为模块和程序编写测试
- ◆ 介绍如何使用Python编写GUI
- ◆ 着重介绍XML、HTML、XSL和相关技术
- ◆ 解释如何扩展Python
- ◆ 分享数值编程技术
- ◆ 介绍Jython的内部机理，Jython是一个用Java编写的Python版本

## 作者简介

James Payne是www.developershed.com的主编，这是一个由高科技网站组成的网络，每个月为几百万个寻求教程、建议、答案或论文的独立用户提供服务。

**Wrox Beginning guides** are crafted to make learning programming languages and technologies easier than you think, providing a structured, tutorial format that will guide you through all the techniques involved.

## 源代码下载及技术支持

<http://www.wrox.com>

<http://www.tupwk.com.cn/downpage>

**Wrox**  
An Imprint of  
**WILEY**

上架建议：软件开发/Python  
读者信箱：wkservice@vip.163.com  
投稿信箱：bookservice@263.net



# wrox.com

## Programmer Forums

Join our Programmer to Programmer forums to ask and answer programming questions about this book, join discussions on the hottest topics in the industry, and connect with fellow programmers from around the world.

## Code Downloads

Take advantage of free code samples from this book, as well as code samples from hundreds of other books, all ready to use.

## Read More

Find articles, ebooks, sample chapters and tables of contents for hundreds of books, and more reference resources on programming topics that matter to you.

ISBN 978-7-302-25709-7



9 787302 257097 >

定价：68.00元

## 第 I 部分

# 初步了解 Python

第 1 章：编程基础和字符串

第 2 章：数值与运算符

第 3 章：变量

数字资源  
PDG



# 第 1 章

## 编程基础和字符串

本章简单地介绍 Python 编程实践。Python 是一门丰富的语言，包含许多特性，因而学好基础知识是深入学习 Python 的前提条件。第 1 章～第 3 章通过易于理解的段落和简单的示例介绍了常见的编程思想。

对 Python 感兴趣的有经验的编程人员可以略读本章，并留意其中的示例，直到第 3 章为止，都可以借助于已有的对其他语言的知识阅读这些章节。

本章将介绍：

- 编程的一些指导原则。
- 关于初步接触程序语言 Python 的一些说明。

本章末的练习提供了对初步学到的基础知识进行实践的机会。

### 1.1 编程与使用计算机的区别

编程时第一件需要了解的事情就是控制计算机。有时，计算机并没有按照人们的意愿工作，当它第一次没能如您所愿后，在第二次、第三次仍然会做与第一次相同的事情，直到对它进行修改。

个人计算机的发展已背离了可靠性原则，并朝不可信的软件套软件的方向发展。这就导致人们认为计算机就像一头具有恶意且任性的野兽，当试图让它完成某些事的时候，它会不断地增加人们的额外的工作量并折磨着使用者。然而，学过编程之后，就会知道如何应对这种情况，也许还会发现自己比所用软件的某些编写者还要聪明。

使用 Python 这种解释型的语言编程，意味着并不需要了解太多的计算机内部细节，如硬件、内存或者长长的 0/1 串。这时编程就与平常阅读和写作一样，只不过是使用一种不同的简单的语言来书写。Python 是一门语言，就像所讲的英语或者其他语言一样，它对讲同种语言的人而言是有意义的。学习程序语言比学习人类语言甚至更简单，因为它并不是用于讨论、争论、打电话、表演、拍电影以及任何形式的日常交流活动。它们用来制订规则，并确保规则被执行。计算机已被塑造成异常灵活的工具，被广泛地用于人们日常生活中的各种商业活动和任务，但它们依旧是由可理解、可控制的基本部件组成的。

#### 1.1.1 编程的一致性

尽管计算机已悄然地囊括了各式各样的设计理念，并日趋复杂。但其基本原理仍然相

对简单。定义计算机如何工作的内部机制从上世纪 50 年代晶体管被第一次用于计算机起，就没有发生过太大的变化。

在那个时代，这种简单的核心意味着计算机可以而且应该具有高度一致性。对于程序员而言，这意味着，在任何时候让计算机跳高(比喻，并非真能跳高)，就必须告诉它要跳多高，落在哪里，这样它才能完成跳跃动作，并且在指定的时间内，它将会一次次地重复相同的动作。如果没有改动程序，程序就不应随意地停止或者改变工作方式。

### 1.1.2 编程的可控性

编制一台计算机与编制一个计划是截然不同的。现实生活中，当要求他人做事时，有时候不得不做一番斗争以确保他们按照自己的意愿行事。例如，假如您计划了一个 30 人的聚会，并指定其中的两个人负责带土豆条和蘸酱，但他们却带来了饮料，这就失去了控制。

计算机则不存在这种问题，它会严格按照指示工作。如您所料，这意味着必须注意一些细节，以确保计算机确实做了要求它做的事。

Python 语言的一个目标就是实现模块化编程，这就使得可以通过将每一个子项目作为一个部件，并通过确保部件以易于理解的方式运行来实现更大的项目。这也是面向对象编程的主要目标。这种编程模式的主要原理是通过编制可靠的部件，使这些部件组装之后依然可以运行，且易于理解和使用。这就给予了程序员对程序每一部分运行方式的控制力，使得当要解决的问题发生变化时，知道如何扩展已有的程序。

### 1.1.3 程序要应对变化

程序运行于计算机之上，并用于解决现实问题。现实世界中，计划和环境总是频繁地改变。这些不断变迁的环境使程序员很难有机会编写出技巧性、实用性和灵活性均趋于完美的程序。通常，只能达到其中的两个目标。这些不得不应对的变化应当教会程序员如何更加小心地编写程序。如果足够小心，就可以编写出这样的程序，当用户的要求超出它的能力范围，它可以很友好地退出，并且通知它的使用者自己已终止工作。最好可以编出能够指出失败位置并解释失败原因的程序。Python 提供了许多有用的特征，使得程序员可以描述出导致程序无法正常工作的原因。

### 1.1.4 小结

综上所述，这些初步的规则意味着您即将被引进编程的大门，在一个自己控制的环境里，告诉计算机要做的工作。还会意识到，意外时有发生，但这些错误是可以通过某些机制加以控制的，这些机制可以让您知道如何对导致错误的条件进行处理，包括如何让程序从问题中恢复并继续运行。

## 1.2 准备工作

在开始用 Python 编程之前，首先要下载并安装 Python 3.1。访问 [www.python.org/download](http://www.python.org/download) 并选择 Python 的最新版本。这会进入一个说明页面，告诉如何下载最适合本地计算机的 Python

版本。例如，如果正在运行 Windows 操作系统，该页面可能会建议下载 Windows x86 MSI Installer(3.0)。

程序是以源代码的形式编写的。源代码中包含了使用程序语言编写的指令，当计算机读到源代码并对其进行处理时，程序里的指令将会变成计算机执行的操作。

就像作家和编辑有编写杂志、书籍以及在线出版物的专用工具一样，程序员也需要专门的工具。作为 Python 的初学者，最合适的编程工具莫过于 Python IDLE 图形用户界面了。

下载结束后，双击以运行程序。最好接受 Python 的所有默认提示。这个过程大概需要几分钟，具体时间取决于所使用的系统。

安装完毕之后，也许想测试 Python 是否被正确地安装。单击 Windows 的 Start 菜单，找到 All programs，会看到 Python 3.0 的图标已经出现在菜单中，选择 IDLE(Python GUI)，等待程序加载。

一旦 IDLE 启动，输入 “Test, test, testing”，并按下回车键。如果 Python 运行正常，它会返回

```
'Test, test, testing'
```

这些字母都是蓝色的，并且带单引号(稍后将解释原因)。恭喜，您已成功安装了 Python，并且正式步入了成为编程大师的征途。

### 1.2.1 在非 Windows 系统上安装 Python 3.1

如果使用 Mac 并运行 Mac OS X 操作系统，很幸运，它已经预装了 Python。不过，它安装的可能不是最新版本的 Python。为了安全性以及兼容性的目的，建议访问 [www.python.org/download/mac](http://www.python.org/download/mac)，检查所用的 Mac OS X 版本是否适合正在安装的 Python 版本。

装有 Linux 系统的计算机可能已经安装好 Python，但同样可能不是最新版本。这里仍然建议去 Python 的官方 Web 站点找到最新版本(当然，要适合所用的系统)。Web 站点 [www.python.org/download](http://www.python.org/download) 上介绍了如何下载适合您的计算机的 Python 版本。

### 1.2.2 使用 Python Shell

在开始编写程序之前，需要学习如何使用 Python Shell 试验程序。现在，可以认为 Python Shell 是一种查看运行中的 Python 代码的方式。通过 Shell 可以进入 Python 的一个运行实例中，并可以向其中填写代码；同时，Python 会执行要求它做的任何工作，并会显示它是如何对程序的环境做出响应的。由于程序通常是在一定的上下文环境——由程序员根据自己的需求量身打造——中运行，使用 Shell 的好处是它可以让程序员在自己创建的上下文环境中进行实验。

成功安装 Python 3.1 后，可以开始使用 Shell 的基本功能做一些实验。例如，可以输入：

```
>>>"Hello World. You will never see this."
```

注意刚才输入 Shell 的语句实际上什么都不做，Python 的环境也没有任何变化。Python 会判断输入的语句让它做什么。在本例中，仅需要 Python 读取所输入的文本。

尽管从技术上说 Python 没有对输入的文本执行任何操作，但它会给出某种指示，表明自己已读过输入的语句。Python 通过显示输入的文本(字符串)表明它已经读过了。字符串是 Python 的一种数据类型，在 Python 中，每种数据类型都有不同的显示方式。在逐步深入地阅读本书的过程中，读者会看到 Python 是如何显示每种不同的数据类型的。

## 1.3 开始使用 Python——字符串

此时，您可以自由使用 Shell 的基本功能进行实验。输入一些文本，并用引号括住它们。初学者可以输入如下语句：

```
>>> "This text really won't do anything"
"This text really won't do anything"
```

您应当立刻注意到，当输入一个引号(")时，Python Shell 改变了之后输入的所有字符的颜色，直到输入第二个引号以结束当前输入的语句。当然，前面的解释是成立的。它什么都不做：它不会改变 Python 的环境，正在运行的 Python 实例仅判断输入的语句是否让 Python 做一些事情。本例中，仅仅让它读入所写的文本，但这些并不会改变环境。

然而，可以看到 Python 表示它看到了输入。它以字符串的形式显示了输入，即用引号把输入的文本括起来。当学习了其他数据类型后，会发现 Python 显示不同数据类型的方式是不同的。

### 1.3.1 字符串概述

字符串是 Python 语言的一种数据类型。顾名思义，数据类型是指某种特定类型的数据所适合的类别。输入到计算机中的数据会被划分为某种数据类型，不管它们是数值还是字母。定义数据类型使得计算机可以断定如何处理输入的数据。例如，如果想让程序在屏幕上显示数学公式  $1+1$ ，必须告诉它输入的是文本。否则，程序会将输入解释为数学公式，并进行相应的计算。

在后续的章节中，读者将会了解到越来越多的不同的数据类型，并理解定义数据类型的重要性。目前，知道字符串是由任意字符构成的数据类型即可，一个字符可能是一个字母、数值、符号或者标点符号。因此，下面所列都是字符串的示例。

```
"Hello, how are you?"
"1+1"
"I ate 4 bananas"
"!@#$%^&*()"
```

### 1.3.2 为什么需要引号

当向 Python 输入一个字符串时，首先要输入一个引号。至于应该使用单引号(')、双引

号(")还是三个引号(''), 要取决于想完成的目标。大多数情况下会使用单引号, 因为它最容易输入(不需要按下 Shift 键)。然而, 请记住, 单引号和双引号以及三层引号是等价的。

尝试输入一些字符串。每当输入一个语句后, 按下回车键让 Python 计算输入的语句。

### 试一试

### 用不同的引号输入字符串

输入如下字符串, 记住引号的类型(单引号或双引号)和行的末端(到达行的末端时使用回车键):

```
>>> "This is a string using a double quote"
'This a string using a double quote'
>>> 'This is a string with a single quote'
'This is a string with a single quote'
>>> """This string has three quotes
look at what it can do! """
'This string has three quotes\nlook at what it can do!'
>>>
```

在上面的示例中, 尽管每个句子看起来都不一样, 但计算机用同样的方式解释它们: 作为字符串。下面将介绍使用 3 种不同引号的真正目的。

### 1.3.3 为什么有 3 种类型的引号

有 3 种类型的引号的原因很简单。假定就像刚才所做的那样, 要在句子中用一个缩写。如果向 Shell 中输入语句"I can't believe it's not butter.", 不会有什么问题, 但是当实际让程序用到刚才输入的字符串时, 就会得到出错消息。为了解释这种现象, 下一节将介绍 print() 函数。

### 1.3.4 使用 print()函数

Python(以及其他程序语言)中的函数是开发人员用来节省时间以及使程序更有效率的工具。他们将可以重复使用的代码存储在函数中, 然后在需要的时候调用这些函数, 而不是一遍遍地编写相同的代码。目前不要太担心函数, 在后面会对它们进行更详细的介绍。现在, 了解函数的含义并且知道它与编程的关系就可以了。

print()函数用来在屏幕上打印文本。在 Python Shell 中尝试如下的示例:

```
>>> print("Hello World! ")
```

按下回车键后, 可得到如下输出:

```
Hello World!
```

这里有几个地方需要注意。首先, 当进入 print()函数时, 会弹出如图 1-1 所示的窗口, 显示函数中各种可用的选项。其次, 后续行中的文本再次显示为蓝色, 但这次, 文本周围

并无引号。这是因为与之前的示例有所不同，这一次 Python 实际上对数据做了一些操作。



图 1-1

恭喜，您已经编写了自己的第一个程序。

### 1.3.5 理解不同的引号

现在知道了如何使用 `print()` 函数，可以练习使用本章前面提到的各种不同类型的引号了。尝试之前的示例：

```
>>> print('This is a string using a single quote!')
This is a string using a single quote!
>>> print("This is a string using a double quote!")
This is a string using a double quote!
>>> print("""This string has three quotes!
Look at what it can do!""")
This string has three quotes
Look at what it can do!
```

可以看到，在示例所示的这些情况中单引号(')和双引号(")是可以互换的。然而，若想使用缩写，例如 `don't`，或者想引用某人的言论，观察会发生什么：

```
>>> print("I said, "Don't do it")
```

当按下回车键执行该函数时，会得到错误消息：`SyntaxError: invalid syntax(<pyshell#10>, line 1)`。您也许会感到奇怪，单引号和双引号不是可以互换吗？大部分情况下是这样的。然而，将它们混合在一起时，经常会出现语法错误，表示错误地输入了代码，并且 Python

不会知道您的意图何在。

实际上, Python 看到输入的第一个双引号后, 把它解释为字符串的开头。当它在单词 Don't 之前碰到双引号时, 会将该双引号看做字符串的结束符。因而, 后面的字母对 Python 是没有意义的, 因为它们不是当前字符串的一部分。直到遇到字母 t 之前的那个单引号时, 字符串才重新开始。

有一个简单的解决办法, 叫做转义。再试一下前面的代码, 在字符串中加入一个转义字符:

```
>>>print("I said, \"Don't do it")
I said, "Don't do it"
```

这一次, 代码可以工作了。当 Python 见到反斜杠(\), 也就是转义字符时, 知道应当把双引号看做一个字符, 而不是某种数据类型的指示符。但是, 也许您已经注意到了, 这行代码还存在最后一个问题。看到输出结果最后丢失了一个双引号吗? 为了使 Python 将语句末尾的双引号打印出来, 只需要再增加一个转义字符以及一个双引号即可, 如下所示:

```
>>>print("I said, \"Don't do it\"")
I said, "Don't do it"
```

最后, 花点时间讨论一下三引号。之前简单介绍过它的用法。从该示例可看出, 利用三引号可以输入多行文本, 在输入结束三引号之前这些文本不会被处理。这种技术在不想将大量数据打印到一行以及想在代码中换行时非常有用。接下来的示例使用这种方法写了一首诗:

```
>>>print("""Roses are red
Violets are blue
I just printed multiple lines
And you did too!""")
Roses are red
Violets are blue
I just printed multiple lines
And you did too!
```

还可以利用换行(\n)转义字符在多行中打印文本, 换行符是最常用到的转义字符。这里对其做一简单介绍, 在后续章节中会深入地讨论它。尝试如下代码:

```
>>>print("Roses are red \n Violets are blue \n
I just printed multiple
lines \n And you did too!")
Roses are red
Violets are blue
I just printed multiple lines
And you did too!
```

可以看到, 结果是一样的。采用哪种方法取决于自己, 但是换行符可能更有效率并且

更易于理解。

## 1.4 串联两个字符串

每个程序员都会遇到要把两个或多个字符串连接在一起的情况，这叫做串联 (concatenation)。例如，假设有一个包含了雇员的名和姓的数据库。某些时候可能希望将这些名字作为完整的记录打印，而不是将名和姓分开。Python 中，每个这样的项都会被看做一个单独的字符串，如下所示：

```
>>>"John"
'John'
>>>"Everyman"
'Everyman'
```

### 试一试

#### 用加号串联字符串

有许多将不同的字符串串联起来的方法。第一种是数学方法：

```
>>> "John" + "Everyman"
'JohnEveryman'
```

也可以不用+号，而采用下面这种方法：

```
>>>"John" "Everyman"
JohnEveryman
```

从这些示例可以看出，字符串都被串联起来了；然而，Python 逐字读这个语句，因而两个字符串之间没有空格(请记住：Python 现在将它们看做一个字符串，而不是两个！)。如何处理这种情况？有两种解决方法。第一种是在第一个字符串后加入一个空格，如下所示：

```
>>>"John " "Everyman"
John Everyman
```

然而，并不推荐使用这种方法，因为将来需要阅读这段代码时，很难确定是否在 John 的后面加了空格。另外一种方法是简单地使用一个分隔符，例如：

```
>>>"John" + " " + "Everyman"
John Everyman
```

使用这种方法而不是简单地输入一个空格的其他原因与数据库存储有关，第 14 章中将讨论这方面的内容。请注意，可以使用任意的分隔符，例如：

```
>>>"John" + "." + "Everyman"
John.Everyman
```

## 用 print() 函数连接字符串

默认情况下，当要在一句话中打印多个字符串时，`print()` 函数会非常周全地插入空格。这里没有必要使用空格分隔符。相反，只需要用逗号将不同的字符串隔开即可。

```
>>> print("John" , "Everyman")
John Everyman
```

## 1.5 用不同的方法串联字符串

另外一种指定字符串的方法是使用格式说明符。它通过插入一个特定的字符序列工作，这个字符序列被 Python 解释为占位符，并将由程序员提供的值替换。初看起来，这种方法很复杂，不是非常有用，但格式说明符可以控制要显示的信息的格式，还可以提供许多有用的技巧。

### 试一试

### 使用格式说明符构成字符串

在最简单的情形下，可以对朋友 John Q. 做同样的事情：

```
>>> "John Q. %s" % ("Public")
'John Q. Public'
```

### 示例说明

`%s` 是针对字符串的格式说明符。在本书对数据类型进行不断讨论的过程中，还会遇到更多的格式说明符，它们分别对应于特定的数据类型。在字符串中，每个格式说明符都担当相应类型的占位符。字符串外面的 `%` 符号表明它后面的所有值会被插入到字符串中相应的格式说明符处。

注意圆括号，它告诉字符串后面的序列包含用于填充格式说明符的值。

一种简单的理解方式是 `%s` 是一个存储器，存放着圆括号中的值。如果想处理更多的值，只需加入另外一个格式说明符即可，例如：

```
>>> "John %s%s" % ("Every" , "Man")
John Everyman
```

这些序列是 Python 编程的一部分，同样地，在本书的后续章节中会对其做更详细的介绍。目前，只需知道字符串中的每一个格式说明符在提供给它的序列中都必须有一个元素与之对应即可。序列中的每一项都是用逗号分隔开的字符串。

如果是向其中存放数据，为什么要把它们叫做格式说明符呢？原因是它们有多种功能，作为存储器只是其功能之一。下面的示例不仅说明了如何向格式说明符中存放数据，还说明了如何用它指定要显示的数据格式。

## 试一试

## 更多字符串格式

这个示例告诉格式说明符处理多少个字符。尝试如下代码，并观察结果：

```
>>> "%s %s %10s" % ("John" , "Every", "Man")
'John Every          Man'
>>> "%-5s %s %10s" % ("John" , "Every", "Man")
John    Every          Man
```

## 示例说明

在第一行代码中，单词 **Man** 出现时，离前面的单词很远，这是因为在最后一个格式说明符中添加了 10，表示一个长度为 10 的字符串。如果字符串没有 10 个字符(它仅包含 3 个字符，即 **M-a-n**)，它就会在前一个单词与 **Man** 中间加入 7 个空格。

在输入的第二行代码中，单词 **Every** 被空格隔开的方式与其他单词不同。原因与第一行代码相同，只不过这次空格是在左边，而不是右边。将负号放在格式说明符的右边，这个格式会出现在单词的右边。如果使用了一个非负的数值，它出现在左边。

## 1.6 本章小结

本章学习了如何安装 Python，以及如何使用 Python GUI(IDLE)，这个 GUI 是用 Python 编写的程序，专门用于编辑 Python 程序。除了编辑文件，这个 Shell 还允许用 Python 的简单编程语句进行实验。

本章通过使用 Shell 介绍了有关字符串处理的基本知识，包括字符串串联，如何使用格式说明符对字符串进行格式化，甚至如何在同样的%s 格式说明符中存储字符串。另外，还学习了如何使用多种类型的引号，包括单引号、双引号和三引号，并且了解了\n 换行转义字符的含义。

最后，学习了第一个函数 `print()`，并且编写了自己的第一个程序 **Hello World**，初学编程时第一个编写 **Hello World** 程序在程序员中间是一个历史悠久的传统，这与演奏吉他首先要学习“**Smoke on the Water**”一样——它们都是必学的第一课。

## 本章要点：

- 编程的一致性。所有程序都是为了特定的用途而创建的，程序的使用者不仅希望程序能够实现预定的用途，而且希望程序每次的工作方式都相同。如果用户单击一个按钮，有一个打印对话框弹出，那么这个按钮应当总以相同的方式工作。
- 编程的可控性。程序员控制着自己的应用程序可以执行以及不可以执行的操作。即使程序的某些方面对于普通的观察者而言是随机的，但实际上它们也是由程序员创建的参数控制的。
- 编程要应对变化。通过不断的测试，可以确保程序对用户做出适当的响应，即使用户让程序做一些其能力范围之外的事情。

- 字符串是一种数据类型，简单讲，是一类数据。这些字符串允许以多种方式与用户进行交互，例如，向窗口打印文本，接受用户输入的文本等等。一个字符串可由任意的字母、数值或者特殊字符组成。

## 1.7 习题

1. 在 Python Shell 中，输入字符串：“Rock a by baby, \n\ton the tree top, \t\when the windblows\n\t\t\t the cradle will drop”。可以使用不同数目的\n和\t转义序列进行实验，看看它们的数目对屏幕上的显示有什么影响。甚至可以尝试把它们放在不同的位置。可能的输出是什么？
2. 在 Python Shell 中，使用与练习 1 中相同的字符串，但是这一次使用 print()函数显示该字符串。再一次尝试不同数目的\n和\t转义序列。结果会有什么不同？





# 第 2 章

## 数值与运算符

人们天生使用数值。婴儿从开始爬行到最终站起来都使用数值估计距离。随着时间的推移，人们扩大了数值的使用范围，开始更加有意识地使用数值，例如购买饮料或制定每月的预算。不论是一岁还是九十岁，都对数值有一定程度的了解。实际上，数值是一个如此熟悉的概念，以至于很难注意到在不同的上下文中有多种使用数值的方式。

本章介绍数值以及 Python 处理数值的一些方法，包括基本算术以及针对不同类型数值的特殊的字符串格式说明符。

本章将介绍：

- Python 使用的各种基本数值类型。
- 使用数值的方法。
- 如何显示以及混合各种类型的数值。

### 2.1 不同类型的数值

如果使用过电子制表软件，会注意到制表软件并不把数值看做单纯的数值，而是看做不同类型的数值。电子制表软件根据不同的单元格格式，用不同的方法显示数值。例如，当处理美元数据时，电子制表软件将 1 美元显示为 1.00。然而，如果想知道自己的车行驶的英里数，很可能以十分之一英里为单位记载，例如 10.2 英里。当说出想为一个新房子付多少钱时，很可能只想到最接近的千位数。在数值的大端，电表账单以千瓦时为单位记录了读表数，千瓦时是指每小时 1000 瓦。

这对 Python 意味着，当想使用数值时，有时需要意识到并不是所有的数值都是相关联的(就像本章中讲到的虚数一样)，并且有时必须谨慎地决定使用什么样的数值以及要用这些数值做什么。然而，一般情况下，有两种使用数值的方式：第一种是告诉 Python 重复一个特定的动作，第二种是用数值代表现实世界中的事物(即在程序中尝试对现实世界中的事物建模)。在 Python 中进行计数时，很少需要将数值看做简单数值之外的其他事物。然而，当尝试解决现实世界中的问题，例如涉及货币、科学、汽车、电等问题时，会发现自己更加清楚如何使用数值。

#### Python 中的数值

Python 提供了三种类型的可用数值：整型、浮点型和虚数。

在之前的版本中, Python 用不同的方法处理大数。介于  $-2\,147\,483\,648$  和  $+2\,147\,483\,647$  之间的数被认为是整型。更大的数被提升为长整型。现在这两种类型已经合并, 所以整型用于表示整数, 无论这个整数是正数还是负数。

要确定数值的分类, 可以使用 Python 中内置的一个特殊函数 `type`。当使用 `type` 时, Python 会告诉正在使用的数据的类型。下面用几个示例进行说明。

### 试一试 对不同的数值使用 `type`

在 Python Shell 中可以输入不同数值, `type` 将指出 Python 如何看待这些数值:

```
>>> type(1)
<class 'int'>
>>> type(2000)
<class 'int'>
>>> type(9999999999999999)
<class 'int'>
>>> type(1.0)
<class 'float'>
```

#### 示例说明

尽管日常生活中, 1.0 与 1 相同, 但 Python 自动将 1.0 看做一个浮点数; 没有 .0, 数值 1 被当作整数 1 来处理(在小学学到的整数), 这是另一种数值类型。

本质上, 浮点数与整数的特殊区别在于浮点数有小数部分。像 1.01、2.34、0.02324 这样的数, 以及其他包含小数部分的任意数值都被当作浮点数(虚数例外, 它们有单独的规则)。这种数值类型可以用来处理货币或者具有可分性质的事物, 例如汽油或者一对袜子(经常发现只剩下一只袜子, 是不是这样?)。

#### 棘手的数值

工程领域、金融领域以及其他领域的专家需要处理非常大以及非常小的数(小到有一堆小数位), 他们需要比内置类型(例如浮点型)更高的精度和一致性。如果要在编程中探索这些学科, 应该使用可用的模块, 第 7 章将介绍模块, 它们用来处理与感兴趣的领域有关的问题。至少, 在需要使用高精度浮点数时, 有必要研究如何使用为处理这些数值编写的模块, 这些模块的使用方法不同于默认行为。

Python 提供的最后一种数值类型是面向工程师和数学家的。它就是虚数, 在学校中应该学习过这种数。虚数定义为  $-1$  的平方根。尽管叫做虚数, 它在对现实工程情形进行建模以及其他领域(例如物理和纯数学领域)都有许多实际的应用。虚数被内置到 Python 中, 因此可以被经常使用计算机解决问题的用户方便地使用。如果您恰巧是这类用户, 就会高兴地发现自己并不孤单, Python 可以帮助您完成自己的目标。

## 试一试

## 创建虚数

除了不能与浮点数混合之外，虚数与浮点数很相似。虚数尾部都有一个字母 `j`：

```
>>> 12j
12j
```

## 示例说明

当在数值之后使用字母 `j`，并且数值和字母都不是字符串(即不在引号中)时，Python 知道要将输入的数值看做虚数。出现在字符串以外的字母都必须有特殊的含义，例如，这个修饰符指定了数值的类型，或者是一个已命名变量(见第3章)，或者是另外一个特殊名称。否则，一个字母单独出现会导致出错！

可以将虚数与非虚数结合起来，创建一个复数：

```
>>> 12j + 1
(1+12j)
>>> 12j + 1.01
(1.01+12j)
>>> type (12j + 1)
<class 'complex'>
```

可以看到，当试图混合虚数以及其他数值时，它们没有相加(相减、相乘或相除)，而是以创建复数的方式保持独立。复数有实部和虚部，但是如何使用它们超出了本章的讨论范围，如果需要使用它们，可以在学习第6章以后探索复数模块(再一次出现这个词！)。模块的名称是 `cmath`，代表 `complex math`。复数将在第19章做进一步讨论。

## 2.2 程序文件

现在，您应该可以比较熟练地使用 Python Shell，并在其中编写不同代码。前面用它实现示例，但现在将以一种不同的方式使用它。不同于简单地输入那些在 GUI 关闭之后会消失的单行代码，现在将创建和保存实际的文件，这些文件可以被再次打开和使用。

本章后续部分鼓励结合使用 Python Shell 和记事本创建自己的文件。

## 试一试

## 在记事本中输入如下文本

向记事本中输入下面的文本：

```
print("This is a basic string")
print("We learned to join two strings using " + "the plus operation")
```

现在已经向编辑器中添加了代码，尝试保存它。选择 File 菜单的 Save As 命令(如图 2-1

所示)。

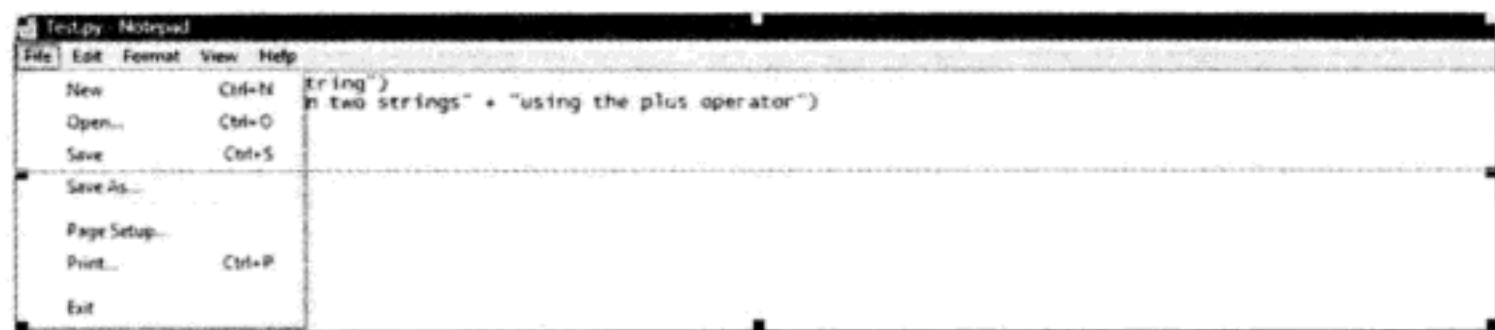


图 2-1

此时将出现一个弹出菜单，提示输入文件名和文件的存储目录。Python 文件使用.py 扩展名，因此要始终确保将.py 加到文件名之后，否则记事本会将文件保存为其默认类型.txt。将这个文件命名为 Test.py。接下来，定位到 Python 的安装目录。通常，安装路径以 C:/Python31/开头。单击 Save 按钮，就把文件保存到了指定位置。

提示：

在选择文件名并保存文件之后，可以重新打开它。要运行 Test.py 程序，在 Python Shell 中选择 File | Open，选定要运行的文件(此处是 Test.py)。Python 编辑器将打开。单击 Run，选择 Run Module(见图 2-2)，然后满怀期待地关注自己的第一个运行的程序。

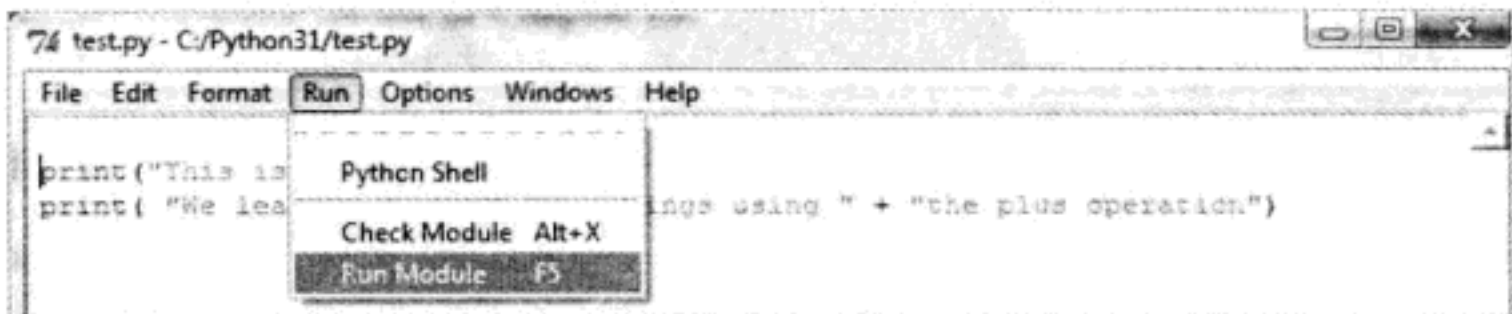


图 2-2

要注意一些事情。首先，初次打开 Test.py 文件时，Python 使用不同的颜色突出显示代码。这使得函数和数据类型(以及其他许多与编程有关的方面)更易于辨认。例如，print() 函数的颜色是紫色的，而包含值的字符串是绿色的。

运行这个模块时，不再看到代码，而是看到它的结果在屏幕上以蓝色文本写出：

```
This is a basic string
We learned to join two strings using the plus operator
```

用不同的字符串再多做几次这样的操作，把它们保存在不同的文件中。每个会话现在都是可用的，之后还可以引用它们。

### 2.2.1 使用不同的类型

除了基本的整型外，其他数值类型也可能增长到具有过多的数位，从而变得难以阅读和理解。所以，经常看到这些数值在生成之后，以类似于科学计数法的格式出现。Python 也允许以这种格式输入数值，所以这是一条双向路。使用非常大的整型数和浮点型数有许多障碍。这个主题涉及很多细节，并且不是学习 Python 必须理解的。如果想对一般意义上的浮点数有更多了解，并且了解它们对于计算机而言意味着什么，[http://docs.sun.com/source/806-3568/ncg\\_goldberg.html](http://docs.sun.com/source/806-3568/ncg_goldberg.html) 是一篇非常优秀的参考论文，尽管其中的解释只对了解

计算机和数值的人有意义。然而，不要因此就不去阅读，它也许是您将来某个时刻想了解的知识。

最常用的是整型和浮点型。从某些地方(例如日期、时间或者某人的年龄信息等)得到数值的情况很常见。当以数值的形式得到数据后，还需要显示数据。

通常的方法是将数值放到字符串中。可以使用第1章中用到的格式说明符方法。直觉上，可能认为应当能够使用+将数值包含到一个字符串中，但实际上这是行不通的，因为它们是不同的类型，而+运算符只能用于同类型数据：例如，两个字符串，两个数值，或者两个其他对象和类型等。有一个例外是浮点数和整型数可以相加。除此以外，应当认为不同类型的数据是不能用+运算符结合起来的。

您可能会感到奇怪，为什么字符串的格式说明符中可以有数值，但是+却不可以。原因是+运算符依赖于要相加的实际项中包含的信息。在 Python 中使用的任何东西都可以看做具有属性的对象，并且所有的属性结合在一起定义了这个对象。每个对象最重要的属性是它的类型，现在对于类型，最重要的是要理解，一些自然而然可以理解的操作(例如+操作)只能用于两个可兼容类型的对象。大部分情形下，除了数值以外，应该把可兼容类型看做同一类型。

#### 提示：

如果确实想对数值和字符串执行+操作(这经常是自己可以决定的风格问题)，可以使用一个内置函数 `str`，该函数在可行的情况下把数值转换为字符串。这样就可以完成将字符串和数值相加，从而形成一个单独的字符串这样的操作。可以对大多数对象使用 `str` 函数，因为大多数对象都有一个作为字符串显示的方法。然而，为了保持一致，现在仍然使用字符串格式说明符。

#### 试一试

#### 在字符串中包含不同的数字

在第1章中使用格式说明符将两个字符串连接在一起时，用到了格式说明符 `%s`，它的含义是“一个字符串”。由于数值和字符串有不同的类型，必须使用不同的说明符将数值包含到字符串中：

```
>>> "Including an integer works with %%d like this: %d" % 10
'Including an integer works with %d like this: 10'
>>> "An integer converted to a float with %%f: %f" % 5
'An integer converted to a float with %f: 5.000000'
>>> "A normal float with %%f: %f" % 1.2345
'A normal float with %f: 1.234500'
>>> "A really large number with %%E: %E" % 6.789E10
'A really large number with %E: 6.789000E+10'
>>> "Controlling the number of decimal places shown: %.02f" % 25.101010101
'Controlling the number of decimal places shown: 25.10'
```

如果不知道在哪里使用格式说明符，注意最后一个示例看起来与打印货币数值的方法很相似，实际上，任何处理元和分的程序都至少要有这种处理数值和字符串的能力。

### 示例说明

任何时候，当为字符串提供格式说明符时，可能会有一些选项可以用来控制该说明符如何显示与它相关联的值。在第 1 章中已经看到与%s 相关的选项，该选项可以控制显示的字符数。数值说明符也是一种约定，指出特定类型的数值应该如何显示。当使用任意数值格式说明符时，这些约定控制着显示的结果。

### 试一试

### 在字符串中将%符号转义

之前已经给出了另外一个小窍门。如果想在程序中打印字符串%d，可以在 Python 字符串中连续使用两个%符号。这仅当在同一字符串中还有其他 Python 可以替换的有效格式说明符时才是需要的。

```
>>> print("The %% behaves differently when combined with other letters, like  
this: %%d %%s %%f %d" % 10)  
The % behaves differently when combined with other letters, like this: %d %s  
%f 10
```

### 示例说明

注意，Python 注意字母的组合，它将在一个字符串中既有格式说明符又有双百分号时正确地工作。

## 2.2.2 基本算术

在程序中使用数值进行基本算术运算的情况是很常见的。加法、减法、除法和乘法都是内置的。加法和减法通过+和-号执行。

### 试一试

### 基本算术

在 Python Shell 提示中可以输入基本的算术表达式，将它当作一个计算器来使用。像计算器一样，Python 接受一组操作，当按下 Enter 键后，它将计算输入的表达式，并给出结果：

```
>>> 5 + 300  
305  
>>> 399 + 3020 + 1 + 3456  
6876  
>>> 300 - 59994 + 20  
-59674  
>>> 4023 - 22.46  
4000.54
```

### 示例说明

简单的算术运算看起来和预期的一样。除了+和-，乘法通过\*执行，除法通过/执行。由于浮点数和整数的区别，乘法和除法在 Python 中并没有想象的那么直截了当。

在 Python 早先的版本中,当数值变大时,将从整型被提升为长整型。然而,Python 3.1 中合并了这两种类型,所以不再需要做这样的提升。观察下面的数值以及当这些数值达到一定大小时 Python 是如何提升它们的:

```
>>> 2000403030 * 392381727
784921595607432810
>>> 2000403030 * 3923817273929
7849215963933911604870
>>> 2e304 * 3923817273929
inf
>>> 2e34 * 3923817273929
7.8476345478579995e+46
```

注意,尽管 Python 可以处理非常大的数,但是一些操作的结果仍会超过 Python 的容纳能力。Python 在遇到无法处理的大数时将返回 inf,这是 infinity(无穷大)的缩写。

在 Python 3.1 之前,除法更值得注意。在没有帮助的情况下,Python 不能够通过除法将一种数转化为另一种数。只有当至少有一个操作数是浮点数(数后有一个小数点)时才显示浮点型结果。如果两个普通的整型或者长整型数(无论是其中的哪种情况,都缺少小数部分,包括.0)相除,余数将被舍弃。这个问题已经被解决了,目前的 Python 将显示小数,除非告诉它不要这样做。观察如下输出:

```
>>> 44 / 11
4.0
>>> 5.0/2.5
2.0
>>> 324/101
3.2079207920792081
>>> 324.5/102.9
3.1535471331389697
```

可以看到,如果将一个整数与另外一个整数相除,将显示一个浮点数,即使在没有余数的情况下也是如此。同样地,一个整数除以一个浮点数将返回一个浮点数。然而要注意,即使整型被显示为浮点型,如前面示例中的 4.0 和 2.0,但实际上它仍旧是整型。不过,324/101 的结果被转换成一个浮点数。

### 试一试

### 使用取余运算

Python 还有一个基本运算应当注意:取余运算。Python 一个新增的功能是允许查看一则除法的完整结果(如前面的公式 324/101 中所示)。以前,如果想知道余数,必须要用取余运算符,因为 Python 只显示结果的整数部分。对于 324/101,Python 将显示 3。不过在某些情况下,仍然仅需要除法结果的余数部分。为了确定这部分结果,必须要用取余运算符,符号是%。不必感到困惑,%只有在用于数值的时候才代表取余数。当使用字符串时,它仍然表示格式说明符的含义。同样的操作在不同的上下文中有不同的含义,这叫做重载,

是一种非常有用的功能，但是当某些操作通过设计而具有不同行为时请不要惊讶。

```
>>> 5 / 3
1.6666666666666667
>>> 5 % 3
2
```

### 示例说明

上面的代码表示 5 除以 3 得 1.6666666666666667，而在第二个示例中，当 5 除以 3 时余数为 2。取余运算符一个非常有用的任务是确定某个数是否可以被另外一个数整除，例如，决定序列中的某些项是否可以均匀地填充到另外一个序列中(第 3 章会介绍更多关于序列的知识)。这里有一些示例可以尝试：

```
>>> 123 % 44
35
>>> 334 % 13
9
>>> 652 % 4
0
```

## 2.2.3 一些惊喜

当处理普通的浮点值时要小心，例如钱数。Python 中的一些地方很令人迷惑。例如，当以看似直接的数学方式操作某些数时，得到的结果后面将有一些多余的值，如下所示：

```
>>> 4023 - 22.4
4000.5999999999999
```

尾部的 9 会令人烦恼，但它们仅仅反映了 Python 的高精度。然而，当打印或者执行数学运算时，这个特性实际上将得到精确的结果。

### 试一试

### 打印结果

尝试实际打印出结果，将上面具有不同寻常结果的数学公式的结果显示给用户，例如，在某个程序内部就可能采取这种做法：

```
>>> print("%f" % (4023 - 22.4))
4000.600000
```

### 示例说明

前面介绍浮点数除法时提到，在 Python 3.0 中会输出整个公式。在计算公式 5/3 时，会得到 1.666666666667。但是也许并不想向用户显示这么长的一串数值。可以用 %f 格式说明符将答案截短。

## 试一试

## %f 格式说明符

尝试下面的代码，观察 Python 处理浮点运算的不同方法，以及如何利用格式化方法操作结果：

```
>>> print("%f" % (5/3))
1.666667
>>> print("%.2f" % (5/3))
1.67
>>> print("%f" % (415 * 20.2))
8383.000000
>>> print("%0.f" % (415 * 20.2))
8383
```

浮点数非常容易让人糊涂。对浮点数的完整讨论超出了本书的范围，但是如果熟悉计算机和数值并且想更多地了解浮点数，可以阅读 [http://docs.sun.com/source/806-3568/ncg\\_goldberg.html](http://docs.sun.com/source/806-3568/ncg_goldberg.html) 上的论文。论文里的解释应该可以帮助完善这里的讨论。

## 2.3 使用数值

如前面的示例所示，可以利用诸如格式说明符的方法将数值包含到字符串中，并用 `print()` 函数显示出来。重要的一点是必须确定如何显示这些数值，以便它们可以表示想让他们表示的含义，这要取决于对应用程序的了解程度。

### 2.3.1 求值顺序

进行数学运算时，可能会看到  $4*3+1/4-12$  这样的表达式。面临的一个问题是确定如何求这种表达式的值，以及自己求值的方法是否与 Python 的求值方法相同。最安全的做法总是把数学表达式用括号括起来，这就使得先求哪些数学运算的值变得清楚。

Python 以如下方式对这些基本的数学运算求值：先计算乘法和除法，后计算加法和减法，但即使这样也有令人困惑的时候。

## 试一试

## 使用数学运算

当考虑一组数学运算时，将它们写下来(或者用键盘输入)看起来最直接，然而以后查看这些运算时很容易感到迷惑。尝试如下示例，设想它们没有括号：

```
>>> (24 * 8)
192
>>> (24 * (8 + 3))
264
>>> (24 * (8 + 3 + 7.0))
432.0
```

```
>>> (24 * (8 + 3 + 7.0 + 9))
648.0
>>> (24 * (8 + 3 + 7.0 + 9))/19
34.10526315789474
>>> (24 * (8 + 3 + 7 + 9))/19
34.10526315789474
>>> (24 * (8 + 3 + 7 + 9))%19
2
```

在示例中要注意，使用浮点数时，整个公式会改为使用浮点数，而去掉所有浮点数会导致 Python 将所有数都看做整型，除非结果是浮点型的。

#### 示例说明

上面的示例以类似于正常求值顺序的方式组合在一起，但是使用圆括号可以肯定哪组代数运算将首先被求值。最里面的运算最先被求值，最外面的运算最后被求值。在一对圆括号内，以正常顺序求值。

### 2.3.2 数值格式

当准备在字符串中包含数值时，有很大的灵活性。下面的“试一试”部分给出了一些示例。

为了显示钱数，使用格式说明符将小数点后的位数限制为两位。

#### 试一试

#### 使用数字格式

例如，尝试下面的示例。这里把一个数值作为钱数打印：

```
>>> print("$%.02f" % 30.0)
$30.00
```

也可以用类似的格式表示小于 1 美分的值，例如当列出单独销售的小件商品时。如果要打印的数值多于即将打印的位数，看看 Python 将怎么做：

```
>>> print("$%.03f" % 30.00123)
$30.001
>>> print("$%.03f" % 30.00163)
$30.002
>>> print("%.03f" % 30.1777)
30.178
>>> print("%.03f" % 30.1113)
30.111
```

#### 示例说明

正如图所示，如果指定的格式比要求 Python 显示的精度更高，Python 不会截断当前的数。它将适当地进行数学上的四舍五入操作。

### 2.3.3 可能会发生的错误

输入这些示例时，可能会犯错。显然，输入一个不同的数时，Python 帮不了任何忙，得到的结果肯定会与本书不同。然而，对于作为格式说明符输入的字母对 Python 没有意义的情况，或者在提供给一个字符串格式说明符的序列中没有足够多的数值时，Python 会试图给出尽可能多的信息，指出发生了什么问题，开发人员可以根据这些信息修复错误。

#### 试一试

#### 出错

为了理解错误发生后，接下来会发生什么，这里给出一些可以尝试的示例。从第4章开始将介绍它们完整的含义，但此时应当知道：

```
>>> print("%.03f" % (30.1113, 12))
Traceback (most recent call last):
  File "<input>", line 1, in ?
TypeError: not all arguments converted during string formatting
```

#### 示例说明

在上述代码中，序列中的元素(总共两个)多于字符串中的格式说明符(仅一个)，因此 Python 给出一条帮助消息。这个错误将导致正在运行的程序停止运行，因此这通常是一个出错条件，或者称为异常。这里，词 **arguments** 指的是格式说明符，但通常用来代表使某个对象工作所需要的参数。当调用需要指定若干值的函数时，每个期望的值就叫做一个参数。

这是程序员认为理所当然的事；这种专门的技术语言可能不会立刻显得有意义，但是熟悉它之后就会感觉正确了。尽管为了减少困惑，在本书的前10章中，将把 **argument** 当作参数，因为没有人会有争议，它们仅仅是用于设置在特定时刻要用到的条件。编程时，**arguments** 和 **parameters** 是可以互换的。

下面列出了另外一个可能发生的错误：

```
>>> print("%.03f, %f %d" % (30.1113, 12))
Traceback (most recent call last):
  File "<input>", line 1, in ?
TypeError: not enough arguments for format string
```

现在知道了 Python 中 **argument** 所代表的含义，错误消息的含义就很明显了。代码中使用了一个格式说明符，但在伴随的序列中并没有一个值可以与之匹配，因此，没有足够多的参数。

如果尝试将字符串和数值相加，也将会出错：

```
>>> "This is a string" + 4
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

错误消息不难理解，因为前面已经介绍了执行这种操作的条件。不过下面给出了清晰的表述：Python 清楚地告诉您哪些事它被告知去做但却做不了，因此需要自己解决问题(提示：可以使用 `str` 函数)。

```
>>> "This is a string" + str(4)
'This is a string4'
```

### 2.3.4 几个不寻常的例子

Python 还为数值提供了另外一个值得了解的特性，以便在遇到时可以理解它。通常我们使用十进制，即基数为 10 的计数系统。它包含 0~9 共 10 个数字。大于 9 的数字由 0~9 的各种组合表示。然而，计算机通常用八进制(octal)以及十六进制(hexadecimal)表示它们实际处理的二进制数。这些系统给程序员提供了一种了解数据字节的简单方法，数据经常由一到两个 8 位组成。

另外，不管是八进制还是十六进制都不能显示负数。用这种方法描述的数值叫做无符号数，与有符号数对应。+号和-号是不同的。通常地，数值被认为是正数，但如果一个数是有符号的，它也可以是负数。如果一个数值是无符号的，它一定是正数。如果要求以无符号格式字符串显示负数，将得到预想不到的结果。

#### 试一试

#### 将数字格式化为八进制和十六进制

```
>>> print('Octal uses the letter "o" lowercase. %d %o' % (10,10))
Octal uses the letter "o" lowercase. 10 12
```

将数值 10 提供给字符串时，第二个数却打印为 12，看起来这像是个错误。然而八进制仅有 8 个数值(0~7)，因此 0~10 用八进制表示依次是 0、1、2、3、4、5、6、7、10、11、12。

```
>>> print('Hex uses the letter "x" or "X". %d %x %X' % (10, 10, 10))
Hex uses the letter "x" or "X". 10 a A
```

还有另外一个地方需要解释。十六进制使用 0~15 的数字，但是因为 9 之后就没有数字了，所以十六进制用了字母 a~f。如果使用格式说明符 `%x`，这些字母是小写的；如果使用 `%X`，它们则是大写的。因此，十进制的 0~19 用十六进制表示为：0、1、2、3、4、5、6、7、8、9、a、b、c、d、e、f、10、11、12、13。

## 2.4 本章小结

本章介绍了 Python 中的数值，但是没有涵盖数值的所有方面。我们看到并使用了 3 种 Python 内置的数字：整型数、浮点型数以及虚数。本章还介绍了如何用字符串格式说明符在字符串中包含数值，并且使用不同风格格式化了这些数值。

要重点记住，数值的格式(数值在字符串中的显示方式)并不会改变其值。即使打印成

整型，浮点数仍然是浮点数，反之亦然。

本章执行了主要的内置的数学运算：加、减、乘、除以及取余。还介绍了如果整型和浮点型混合，结果为浮点型，如果两个整型数相除，可能也将适当地返回一个浮点型数。如果将整型或者浮点型数与虚数结合起来做数学运算，结果将是一个复数，复数分别有实部和虚部。本章还示范了 `type` 函数，该函数可以确定数值的类型。

最后，人们通常使用十进制数，而 Python 很容易将数值转换为八进制或者十六进制。

#### 本章要点：

- Python 中有 3 种数值类型。它们分别是：整型数，即整数(正数和负数)；浮点型数，即带小数值的数；虚数，即  $-1$  的平方根，经常用于工程和物理计算。
- 当用于字符串时，`+` 操作符可以将两个或多个字符串串联。例如，语句 `print("Hello, "+"how are you?")` 的结果是一句话：“Hello, how are you?”。
- 可以使用 `str` 函数将数值转换为字符串。
- 将两个整型数相除有时可能得到一个浮点型数(例如， $3/2$ )。用一个浮点数除一个整型数，总会得到一个浮点数。
- 取余运算符(`%`)用来返回一则除法中的余数。例如，`5%2` 将返回 1。
- 计算中使用括号可以帮助确保以正确的顺序求值。

## 2.5 习题

在记事本中完成前 3 个习题，并将结果保存于文件 `ch2_exercises.py` 中。在 Python 中，可以打开文件，选择 `Run Module` 运行它们。

1. 在 Python Shell 中，将 5 和 10 相乘。并尝试其他数值。
2. 打印数字 6~14 的八进制形式。
3. 打印数字 9~19 的十六进制形式。
4. 尝试从 Python 解释器中引出其他错误，例如故意将 `print` 拼错为 `pinrt`。注意，在 PythonShell 的文件上工作时，它以不同方式显示 `print` 和 `pinrt`。





# 第 3 章

## 变 量

在前两章中，您学习了 Python 如何看待字符串、整型、浮点型和虚数，以及如何创建和显示它们。本章将给出更多的示例，说明这些数据类型的用法。

本章将介绍：

- 使用名称存储已经了解的以及即将介绍的类型。
- 如何处理还未学习的不同类型的对象，如变量以及不同的新类型，学习本章后您将更加熟悉列表、元组和字典。
- 引用的概念及示例。
- 学习本章的重点是，应当亲手输入示例代码并且更改它们，观察会发生什么。

### 3.1 引用数据——使用数据的名称

在程序中总是显式地写字符串和数值是非常困难的，因为这迫使程序员记住所有的字符和数值等。计算机的内存使得它可以比人记住更多的细节，利用计算机的这种能力是编程中一个很大的部分。然而，为了更灵活简单地使用数据，可以给数据命名，之后可以用名称引用这些数据。

#### 试一试

#### 给名称赋值

这些名称一般叫做变量，表明它们引用的数据可以变化，而名称保持不变。您会看到变量也叫做名称(name)，这是 Python 的叫法。

```
>>> first_string = "This is a string"
>>> second_string = "This is another string"
>>> first_number = 4
>>> second_number = 5
>>> print ("The first variables are %s, %s, %d, %d" % (first_string,
second_string, first_number, second_number))
The first variables are This is a string, This is another string, 4, 5
```

### 示例说明

可以用等号(=)将一个值(字符串或者整型数)与名称关联。所用的名称并不与所指的数据直接相关(如果将一个数据命名为“number”，并不意味着它实际保存了一个数值)。

```
>>> first_string = 245
>>> second_number = "This isn't a number"
>>> print(first_string)
245
>>> print(second_number)
"This isn't a number"
```

注意，仅希望将一个变量的值打印出来时并不需要使用引号。如果在 `print()` 函数中将一个变量用引号括起来，函数会打印出这个变量的名称，而不是它的内容，这是因为程序把它看做一个字符串而不是一个变量。命名数据的好处是可以赋予它一个有着某种含义的名称。给数据起一个有意义的名称非常有必要，它可以说明数据所包含的内容，或者数据在程序中的使用方式。如果打算清点家里所有的灯泡，您也许希望有一段程序包含壁橱中的灯泡数量，另外还有一段程序包含正在使用的灯泡的数量：

```
>>> lightbulbs_in_closet = 10
>>> lightbulbs_in_lamps = 12
```

使用灯泡时，把它们从壁橱中移到灯中，还可以给今年已经扔掉的灯泡数起一个名称，这样在年终，就会知道买了多少灯泡，现在有多少，以及使用过多少；当想知道还有多少灯泡时，只需要引用 `lightbulbs_in_closet` 或者 `lightbulbs_in_lamps` 即可。

当名称与其存储的值相关联时，就创建了一个非正式的索引，这样就可以查询并且记住信息存放的位置，从而便于在程序中使用它们。

### 3.1.1 使用名称修改数据

如果数据是一个数值或者是一个字符串，可以通过已经知道的一些可用的操作修改它们。

#### 试一试

#### 更名已命名的值

我们学过的关于数值和字符串的所有操作都可用于变量名，这样可以像处理它们所引用的数值那样处理它们：

```
>>> proverb = "A penny saved"
>>> proverb = proverb + " is a penny earned"
>>> print(proverb)
A penny saved is a penny earned
>>> pennies_saved = 0
>>> pennies_saved = pennies_saved + 1
>>> print(pennies_saved)
1
print(pennies_saved + 1)
2
```

### 示例说明

无论何时，在等号的右端使用已命名的值时，即使同样的名称出现在等号的左端，Python 仍将这些名称看做它们所引用的值。当 Python 遇到这样的情况时，它首先对右边的名称求值并计算结果，然后将结果赋给左边的名称。这样，名称同时出现在等号两端时就不再引起混乱，Python 会执行正确的操作。

### 3.1.2 复制数据

给数据起的名称仅仅是个名称，是表示要访问的数据的一种方法。这意味着可以用多个名称引用同一个数据：

```
>>> pennies_saved=1
>>> pennies_earned = pennies_saved
>>> print(pennies_earned)
1
```

当再次使用=号时，就给名称赋予了一个已创建的新值，而另一个名称仍然指向原来的值。

```
>>> pennies_saved = pennies_saved + 1
>>> print(pennies_saved)
2
>>> print(pennies_earned)
1
```

### 3.1.3 禁用的名称以及一些规则

Python 把一些名称作为特殊的内置词，它保留这些词用于特殊用途以防止出现多义性。下面列出了 Python 的保留词，不可以将它们用作数据的名称：

```
and, as, assert, break, class, continue, def, del, elif, else,
except, exec, False, finally, for, from, global, if, import, in,
is, lambda, not, None, or, pass, print, raise, return, try, True,
while, with, yield
```

另外，数据的名称不能以数值或者多数非字母的字符开头(例如逗号、加减号和斜杠等)，但下划线例外。下划线是合法的，甚至在某些情形下有特殊含义(特别是用于类和模块时，详见第 6 章以及后续章节)。

在本章后面的讨论中将会看到许多这样特殊的保留字。在使用 Python 完成各式各样的任务时，它们非常重要。

## 3.2 使用更多的内置类型

除了字符串和数值之外，Python 还提供了另外 4 种重要的基本类型：元组、列表、集

合和字典。这 4 种类型有许多共同点，因为它们都允许在一个名称下面组织多条数据。因为具有这种组织性，每个类型都允许在它们中查找元素。这些组织在一起的数据通过圆括号()、方括号[]和花括号{}表示。

在写程序或者读别人的程序时，如果看到一组元素，注意所使用的括号的类型是很重要的。{}、[]和()的区别非常重要。

### 3.2.1 元组——不可更改的数据序列

在第 1 章和第 2 章中为字符串中的多个格式说明符赋值时，看到了元组(tuple)的应用。元组是值的序列，其中每个值都可以被单独访问，元组是 Python 的基本类型。在创建时就可以识别出元组，因为它们被圆括号包围：

```
>>> print("A %s %s %s %s" % ("string", "filled", "by a", "tuple"))
A string filled by a tuple
```

#### 试一试

#### 创建和使用元组

元组包含对数据的引用，例如对字符串和数值的引用。然而，虽然它们引用数据，但是仍然可以像其他类型的数据一样为它们命名：

```
>>> filler = ("string", "filled", "by a", "tuple")
>>> print("A %s %s %s %s" % ("string", "filled", "by a", "tuple"))
A string filled by a tuple
```

注意，可以通过在 `print()` 函数中简单地调用元组，打印出其中的数据。尝试如下代码并观察结果：

```
>>> filler = ("string", "filled", "by a", "tuple")
>>> print(filler)
('string', 'filled', 'by a ', 'tuple')
```

可以看到，组成元组的 4 个部分被返回。这种技术在每次想查看组成元组的单个部分时非常有用。

#### 示例说明

从上面的示例可以看出，`filter`(包含字符串的元组)的处理方式好像是它的元素都呈现出来并且被字符串用来填充格式说明符，因为元组的处理方式就好像输入一个序列来满足给定的格式规范。

可以访问元组中的单个值。每个元素引用的值可以通过该语言的解除引用特性直接访问。通过在元组的名称后放置方括号并且从零起计算出要引用的元素的位置，可以解除元组中对值的引用。因此，元组中第一个元素的位置是 0，第二个元素的位置是 1，第三个元

素的位置是 2，以此类推直到最后一个元素：

```
>>> a = ("first", "second", "third")
>>> print("The first element of the tuple is %s" % a[0])
The first element of the tuple is first
>>> print("The second element of the tuple is %s" % a[1])
The second element of the tuple is second
>>> print("The third element of the tuple is %s" % a[2])
The third element of the tuple is third
```

元组知道它所包含的元素的数目，可以使用内置函数 `len` 得到这个数目：

```
>>> print("%d" % len(a))
3
```

这将返回元组中元素的数目(此处是 3)，因此需要记住 `len` 函数从 1 开始计数，但当访问元组时，由于元组从 0 开始计数，所以必须在比 `len` 返回的数小 1 的位置停止访问。

```
>>> print(a[len(a) - 1])
Third
```

元组中的一个元素也可以是对另外一个元组的引用。换句话说，可以创建嵌套元组：

```
>>> b = (a, "b's second element")
>>> print(b)
(('first', 'second', 'third'), "b's second element")
```

现在可以通过在第一组方括号后面加上另外一组方括号来访问元组 `a` 中的元素，访问 `a` 中的第二个元素与访问第一个元素的方法相同，仅需再添加一组方括号即可。

### 试一试

#### 通过一个元组访问另外一个元组

重新创建元组 `a` 和 `b`，以便观察如何通过一个元组访问另外一个元组。嵌套的序列有时也被称具有多维，因为它们有两层，可以想象纵向和横向延伸，就像图纸或者电子表格上的二维网格一样。再增加第三层元组可被认为是三维的，像一堆积木一样。可视化超过三维的多维数据会令人头痛，将它们看做多层的嵌套的数据会好一些。

```
>>> a = ("first", "second", "third")
>>> b = (a, "b's second element")
>>> print("%s" % b[1])
b's second element
>>> print("%s" % b[0][0])
first
>>> print("%s" % b[0][1])
second
>>> print("%s" % b[0][2])
third
```

### 示例说明

在每种情形下，代码的工作原理仿佛是先遵循元组 `b` 的第一个元素的引用，之后遵循第二层元组中的每个值的引用(第二层的元组最初来源于元组 `a`)。如同执行了如下操作一样：

```
>>> a = ("first", "second", "third")
>>> b = (a, "b's second element")
>>> layer2 = b[0]
>>> print(layer2[0])
'first'
>>> print(layer2[1])
'second'
>>> print(layer2[2])
'third'
```

注意，元组从被创建起就存在一个奇怪的地方。若要创建包含一个元素的元组，必须在该元素之后加一个逗号：

```
>>> single_element_tuple = ("the sole element",)
```

否则，将会创建一个字符串，在之后试图访问它时容易混淆。

元组可以包含各种类型的数据，但在创建之后，就不能再改变。元组是不可变的，Python 中还有几个类型是不可变的(例如，字符串在创建之后是不可变的，那些看起来改变它们的操作实际上创建了新的字符串)。

元组是不可变的，因为它们用来存储一组有序的事物，这些事物在使用时是不可以更改的。试图更改元组中的元素将导致 Python 报错，就像第 2 章末尾的错误一样。

```
>>> a[1] = 3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
>>> print("%s" % a[1])
second
```

在试图给元组中的元素赋值时，可以看到 Python 返回的错误是 `TypeError`，意思是该类型并不支持此操作(等号会引起元组执行一个动作)。在这个示例中，试图使元组 `a` 中的第二个元素引用一个整型数 3，但这个动作将不会发生。相反，`a` 保持不变。

如果试图引用元组中并不存在的元素，将发生一个不相关错误。如果试图引用 `a` 中的第四个元素，将得到一条错误(记住，由于元组从 0 开始对它的元素计数，因此要用数字 3 来引用第四个元素)。

```
>>> a[3]
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    a[3]
IndexError: tuple index out of range
```

注意，这是一个 `IndexError` 错误，并且对错误的解释也已给出(尽管它没有告诉超出范围的索引值，但确实可以知道自己试图用元组中并不存在的索引值访问其中的一个元素)。要在程序中修复该错误，必须找出试图访问的值以及元组中元素的数目。相对于不提供错误提示的语言，`Python` 使得找出这些错误变得容易。

### 3.2.2 列表——可以更改的数据序列

列表如同元组，是包含从 0 开始引用的元素的序列。列表用方括号创建：

```
>>> breakfast = [ "coffee", "tea", "toast", "egg" ]
```

#### 试一试

#### 查看列表中的元素

列表中的单个元素的访问方法与元组类似。像元组一样，列表中的元素从 0 开始引用，并且访问顺序也是从 0 开始直到末端。

```
>>> count = 0
>>> print("Today's breakfast is %s" % breakfast[count])
Today's breakfast is coffee
>>> count = 1
>>> print("Today's breakfast is %s" % breakfast[count])
Today's breakfast is tea
>>> count = 2
>>> print("Today's breakfast is %s" % breakfast[count])
Today's breakfast is toast
>>> count = 3
>>> print("Today's breakfast is %s" % breakfast[count])
Today's breakfast is egg
```

#### 示例说明

当顺序访问列表中的多个元素时，有必要使用一个名称来存储目前在列表中所处的位置。在简单的示例中，应当采取这种做法以形成一种习惯，实际中则总是会这样做。通常，用循环方法查看序列中的每一个元素(第 4 章将详细讨论循环)。

本例中手动完成了对 `count` 引用的值加 1 的工作，以遍历早餐列表中的每个元素，列出本周中 4 天的特色早餐。因为递增计数，不论 `count` 引用的数是什么，它都是早餐列表中被访问的元素索引。

使用列表与使用元组主要的区别在于，元组在创建后不可以修改，列表在任何时刻都可以被修改：

```
>>> breakfast[count] = "sausages"
>>> print("Today's breakfast is %s" % breakfast[count])
Today's breakfast is sausages
```

不仅可以修改列表中已经存在的元素，还可以向列表中添加需要的元素。可以用列表类型的内置方法 `append` 向列表末端添加元素。利用 `append` 方法每次只能向列表末端添加

一个元素：

```
>>> breakfast.append("waffles")
>>> count = 4
>>> print ("Today's breakfast is %s" % breakfast[count])
Today's breakfast is waffles
```

如果希望一次性地向列表末端添加多个元素，例如一个元组或者其他列表的内容，可以使用 `extend` 方法。添加的列表并不是整体地作为原有列表的一个元素，相反，其中的每个元素将被复制到原来的列表中。

```
>>> breakfast.extend(["juice", "decaf", "oatmeal"])
>>> print(breakfast)
['coffee', 'tea', 'toast', 'egg', 'waffle', 'juice', 'decaf', 'oatmeal']
```

同元组一样，不可以请求列表以外的元素，但是出错消息与元组略有不同，因为错误消息将指出是列表索引超出范围，而不是元组索引超出范围。

```
>>> count = 8
>>> print("Today's breakfast is %s" % breakfast[count])
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    print("Today's breakfast is %s" % breakfast[count])
IndexError: list index out of range
```

列表的长度也通过 `len` 函数确定。就像元组一样，长度从 1 开始，而列表的第一个元素的位置从 0 开始。总是记住这一点非常重要。

### 3.2.3 字典——以名称索引的分组数据

字典类似于列表和元组。它是包含一组数据的另外一种容器。然而，元组和列表以数字顺序索引，字典却用选择的名称索引。这些名称可以是字母、数值、字符串或者符号，自己认为合适即可。

#### 试一试

#### 创建字典

字典用花括号创建。刚开始，可以创建最简单的字典，即空字典，并通过逐行指定名称和值对它进行实例化。

```
>>> menus_specials = {}
>>> menus_specials["breakfast"] = "Canadian ham"
>>> menus_specials["lunch"] = "tuna surprise"
>>> menus_specials["dinner"] = "Cheeseburger Deluxe"
```

#### 示例说明

当初次为 `menus_specials` 赋值时，用花括号创建了一个空字典。一旦字典被定义并且

通过名称引用，将开始使用如下方式对其实例化：将希望作为索引的名称放在方括号内，将通过该索引引用的值放在等号的右端。因为值由所选择的名称索引，所以可以使用上述形式给已经定义的任意字典分配索引和值。

使用字典时，索引和值都有特殊的名称。字典中索引的名称叫做键，对应的值叫做值。为了创建一个完全指定(或者可以认为它是完全形成的)的字典(一开始就指定了键和值的字典)，必须在花括号之间指定每个键以及和它对应的值，并以冒号分隔它们。例如，另外一天的特色菜可以一次性定义：

```
>>> menu_specials = {"breakfast" : "sausage and eggs",
... "lunch" : "split pea soup and garlic bread",
... "dinner": "2 hot dogs and onion rings"}
```

为了打印出某个字典中所有的键与值，只需将该字典的名称作为 `print()` 函数的参数，如下面的代码所示。为了访问字典中的值，可以将键放在方括号中。如果键是字符串，需要将键放在引号中。如果键是数值(可以用数值创建字典，这样的字典十分类似于列表和元组)，仅使用数值就可以了。

```
>>> print(menu_specials)
{'lunch': 'split pea soup and garlic bread', 'breakfast': 'sausage and eggs',
 'dinner': '2 hot dogs and onion rings'}
>>> print("%s" % menu_specials["breakfast"])
sausage and eggs
>>> print("%s" % menu_specials["lunch"])
split pea soup and garlic bread
>>> print("%s" % menu_specials["dinner"])
2 hot dogs and onion rings
```

如果键是一个字符串，但是在方括号中意外地没有将键放在引号中，Python 就试图将它看做一个名称，需要解除对它的引用来找到键。大部分情况下，这将引起 `NameError` 异常，除非凑巧找到了一个与该字符串相同的名称，但这时又可能得到一个 `IndexError` 错误。

### 试一试

### 从字典中获取键

如果知道如何询问，字典可以告诉您它所有的键，或者所有的值。`keys` 方法要求字典以视图的方式返回所有键，以便用户查找所需的键，`values` 方法也将以视图的形式返回所有的值。

```
>>> hungry=menu_specials.keys()
>>> print(list(hungry))
lunch
breakfast
dinner
>>> starving=menu_specials.value()
>>> print(list(starving))
```

```
split pea soup and garlic bread
sausage and eggs
2 hot dogs and onion rings
```

### 示例说明

`keys` 和 `values` 方法都返回视图,可以像其他常规的视图一样使用这些视图并为其赋值。如果从 `keys` 方法得到视图中的项,可以使用视图中的项(也就是键),得到字典中与之匹配的值。注意,尽管一个特定键可以找出一个值,但是不能从一个值开始可靠地找到与该值关联的键。当根据已知的仅有的值试图寻找键时,需要彻底测试该值所有可能的键,尽管如此,仍可能存在两个不同的键与相同的值关联的情况。

字典的工作原理是每个键都是不同的(不可以有完全相同的两个键),但是可以有多个重复的值:

```
>>> menu = {"breakfast" : "spam", "lunch" : "spam", "dinner": "Spam with a side
of Spam"}
>>> print(menu)
{'lunch': 'spam', 'breakfast': 'spam', 'dinner': 'Spam with a side of Spam'}
>>> menu.get("lunch")
'spam'
>>> menu.get("breakfast")
'spam'
```

可见, Python 允许在不同的键下有多个值。然而,尝试如下代码并观察会发生什么,这段代码试图用同样的名称创建不同的键:

```
>>> menu2 = {"breakfast" : "spam", "breakfast" : "ham", "dinner": "Spam with
a side of Spam:"}
>>> menu2.get("breakfast")
'ham'
```

这里发生了什么呢? 尽管没有得到错误消息,代码中仍然有错误。当输入第二个叫做“breakfast”的键时, Python 替换了相同名称的第一个键的值。

### 3.2.4 像列表一样处理字符串

Python 给字符串提供了一个有趣的特性。有时,能够将字符串当作单个字符的列表那样处理很有帮助。在字符串的末端有无关字符并不罕见。人们也许没有识别出它们,但计算机遇到它们时会挂起。仅需要看字符串的首字符就知道如何处理该字符串的情况也很常见。例如,如果有一个姓与名的列表,您可以使用与列表相同的语法查看名与姓的第一个字符。这种看待字符串的方法叫做分片(slicing),是 Python 比较有趣的一个特性。

```
>>> last_names = [ "Douglass", "Jefferson", "Williams", "Frank", "Thomas" ]
>>> print("%s" % last_names[0])
Douglass
>>> print("%s" % last_names[0][0])
D
```

```
>>> print("%s" % last_names[1])
Jefferson
>>> print("%s" % last_names[1][0])
J
>>> print("%s" % last_names[2])
Williams
>>> print("%s" % last_names[2][0])
W
>>> print("%s" % last_names[3])
Frank
>>> print("%s" % last_names[3][0])
F
>>> print("%s" % last_names[4])
Thomas
>>> print("%s" % last_names[4][0])
T
```

例如，可以根据字符串的字母顺序将字典中的元素以姓的首字母分组。并不需要做任何复杂的操作，只需要检查包含名字的字符串以哪个字母开头，并以此对名字归档即可。

```
>>> by_letter = {}
>>> by_letter[last_names[0][0]] = last_names[0]
>>> by_letter[last_names[1][0]] = last_names[1]
>>> by_letter[last_names[2][0]] = last_names[2]
>>> by_letter[last_names[3][0]] = last_names[3]
>>> by_letter[last_names[4][0]] = last_names[4]
```

通过使用字符串的分片功能，字典 `by_letter` 仅包含所有姓的首字母。因此，`by_letter` 是根据每个姓的首字母索引的字典。也可以令 `by_letter` 中的每个键引用一个列表，并使用列表的 `append` 方法创建以相同字母开头的一系列名字(当然，前提是您希望拥有一个索引了更大的一组名字的字典，其中每个名字都以相同的字母开头)。

### 非常有用的字符串分片

如果刚刚接触编程，字符串分片最初看上去是一个不同寻常的特性。使用过 C 或者 C++ 这样的低级语言的程序员学习过如何将程序中的字符串看做特殊的列表(在 Python 中也可以对列表分片，如后面所示)，对他们而言，这个特性很自然。对于读者而言，一旦学习了如何在列表上控制元素的重复，它将是一个非常方便的工具。

### 3.2.5 特殊类型

Python 包含几个特殊类型。前面已经介绍过它们，但在此仍有必要单独介绍：None、True 和 False 都是特殊的内置值，可以在不同的时刻发挥作用。

None 较特殊，因为仅有一个 None。无论使用了多少次，它都只是一个名称，除了它本身以外没有任何其他对象可与之匹配。如果函数没有可以返回的值，即当函数没有可以

响应的动作时，它将返回 `None`。

`True` 和 `False` 是数值 0 和 1 的特殊表示。这就防止了在其他语言中比较常见的混乱，即一个语句的真值是任意的。例如，在 Unix Shell 中(Shell 不仅是与系统交互的平台，也是一门程序语言)，0 是真，其他值都是假。而在 C 和 Perl 中，0 是假，其余值是真。

然而，在这些情形中，没有内置的名称可以区分这些值。Python 通过显式地命名这些值，使得区分这些值变得更容易。名称 `True` 和 `False` 可用于基本的比较，您将看到许多这样的示例。在第 4 章中将介绍这些比较如何显著地影响程序，实际上，它们可以用来在程序中做决策。

```
>>> True
True
>>> False
False
>>> True == 1
True
>>> True == 0
False
>>> False == 1
False
>>> False == 0
True
>>> False > 0
False
>>> False < 1
True
```

## 3.3 序列的其他共有属性

元组和列表是两种类型的序列。前面也已介绍过，在某些情形下，可以像序列那样访问字符串。字符串是有意义的，因为可以将字符串中的字母看做一个序列。

尽管字典代表一组数据，但它们不是序列，因为它们没有一个从头至尾的特定顺序，而这是序列的一个特性。

### 3.3.1 引用最后一个元素

所有的序列类型都提供了一些捷径，使得它们用起来更加方便。经常需要知道一个序列最后一个元素的内容，有两种方法可以得到该信息。一种方法是获取列表中元素的个数，之后使用该数直接访问值：

```
>>> last_names = [ "Douglass", "Jefferson", "Williams", "Frank", "Thomas" ]
>>> len(last_names)
5
>>> last_element = len(last_names) - 1
>>> print("%s" % last_names[last_element])
Thomas
```

然而,这个方法分为两步。作为程序员,重复地在程序中输入这些代码非常耗时。Python 提供了一个捷径,使得可以通过使用数值 -1 访问一个序列的最后一个元素,使用数值 -2 访问倒数第二个元素,通过使用从数 -1 到负的列表长度(在 `last_name` 列表中是 -5),可以将列表的顺序颠倒过来。

```
>>> print("%s" % last_names[-1])
Thomas
>>> print("%s" % last_names[-2])
Frank
>>> print("%s" % last_names[-3])
Williams
```

### 3.3.2 序列的范围

您可以获取序列的一部分,从中提取一个片段,创建可以单独使用的一个副本。创建这些分组的方法叫做分片(与对字符串进行相同操作时所用的术语相同)。当从一个列表或者元组中创建一个片段时,得到的片段与创建该片段的数据类型一致,从字符串分片的示例中已经看到了这一点。例如,从列表中创建的片段是一个列表,从元组中创建的片段是一个元组,字符串的片段还是字符串。

#### 试一试

#### 将序列分片

前面已经学过如何将字符串分片,下面试着用同样的方法对元组、列表和字符串进行分片,并看看结果是什么:

```
>>> slice_me = ("The", "next", "time", "we", "meet", "drinks", "are", "on", "me")
>>> sliced_tuple = slice_me[5:9]
>>> print(sliced_tuple)
('drinks', 'are', 'on', 'me')
>>> slice_this_list = ["The", "next", "time", "we", "meet", "drinks",
"are", "on", "me"]
>>> sliced_list = slice_this_list[5:9]
>>> print(sliced_list)
['drinks', 'are', 'on', 'me']
>>> slice_this_string = "The next time we meet, drinks are on me"
>>> sliced_string = slice_this_string[5:9]
>>> print(sliced_string)
'ext '
```

#### 示例说明

在每种情形下,用冒号指定序列的一个片段指示 Python 创建一个新的序列,新序列恰好包含了片段中的元素。

### 3.3.3 通过附加序列增长列表

假设希望将两个列表连接在一起。我们还没看到专门用于这个方法。不能用

`append` 方法将一个序列附加到另一个序列的末端，这样得到的结果是向列表中增加了一个分层的序列。

```
>>> living_room = ("rug", "table", "chair", "TV", "dustbin", "shelf")
>>> apartment = []
>>> apartment.append(living_room)
>>> apartment
[('rug', 'table', 'chair', 'TV', 'dustbin', 'shelf')]
```

如果打算根据元组 `living_room` 的内容创建一个新的列表，并使用该列表创建包含 `apartment` 中所有元素的列表，以上所示并不是希望看到的结果。

为了复制一个序列中的所有元素，可以使用列表和元组的 `extend` 方法，而不是 `append` 方法，`extend` 方法将给定序列中的每个元素插入到调用它的列表中。

```
>>> apartment = []
>>> apartment.extend(living_room)
>>> apartment
['rug', 'table', 'chair', 'TV', 'dustbin', 'shelf']
```

### 3.3.4 使用列表临时存储数据

经常需要从其他来源获取数据，例如，用户输入的数据或者另外一台计算机上的信息。为此，最好将这些数据存放在一个列表中，这样就可以在随后按照数据进入列表的顺序对其进行处理。

然而，在处理完数据之后，不再需要将这些数据保存在列表中，因为不再需要它们。特定时间有效的信息(例如证券报价信息、天气预报或者新闻标题等)都属于这类信息。

为了防止列表变得笨重，可以使用 `pop` 方法在处理完列表的一个数据之后，将其引用从列表中删除。当删除引用之后，它原来在列表中占据的位置会填上后续元素，列表减少的元素个数等于已经弹出的元素个数。

#### 试一试

#### 从列表中弹出元素

需要告诉 `pop` 方法所要操作的元素。如果告知它处理元素 0，它将弹出列表中的第一项，向 `pop` 传递参数 1 告诉它使用位置 1 的项(列表中的第二个元素)，依此类推。`pop` 所操作的元素与使用方括号访问的列表元素具有相同的索引(记住，列表中第一个元素的位置为 0)：

```
>>> todays_temperatures = [23, 32, 33, 31]
>>> todays_temperatures.append(29)
>>> todays_temperatures
[23, 32, 33, 31, 29]
>>> morning = todays_temperatures.pop(0)
>>> print("This mornings temperature was %.02f" % morning)
This mornings temperature was 23.00
>>> late_morning = todays_temperatures.pop(0)
```

```
>>> print("Todays late morning temperature was %.02f" % late_morning)
Todays late morning temperature was 32.00
>>> noon = todays_temperatures.pop(0)
>>> print("Todays noon temperature was %.02f" % noon)
Todays noon temperature was 33.00
>>> todays_temperatures
[31, 29]
```

### 示例说明

当弹出一个值时，如果操作是在等号的右端，可以将删除的元素赋给左边的值，或者仅在合适的情形下使用该值。如果没有将弹出的值赋给其他变量，也没所有使用它，它将被丢弃。

通过只使用 `pop` 填充字符串格式，也可以避免使用中间名称，因为 `pop` 将返回列表中的指定元素，可以像已经指定了一个数值或者引用一个数值的名称那样使用该元素：

```
>>> print("Afternoon temperature was %.02f" % todays_temperatures.pop(0))
Afternoon temperature was 31.00
>>> todays_temperatures
[29]
```

如果没有告诉 `pop` 要删除列表中的哪个元素(在上面的示例中是 0)，它将删除列表中的最后一个元素，而不是这里所示的第一个元素。

### 3.3.5 处理集合

在 Python 中，集合与字典类似，只是它仅包含键，而没有与键相关联的值。本质上，集合是不包括重复数据的数据集。在从数据集删除重复数据时，集合非常有用。

有两种类型的集合：可变集合与不可变集合(`frozenset`)。两者的不同之处在于，对于可变集合，可以增加、删除、或者改变它的元素，而不可变集合的元素在它们被初始设定之后就不能再被更改。

#### 试一试

#### 删除重复的元素

本例中执行了赋值操作，并且通过把这些值赋给一个集合删除了重复的元素：

```
>>> alphabet = ['a','b', 'b', 'c', 'a', 'd', 'e']
>>> print(alphabet)
['a', 'b', 'b', 'c', 'a', 'd', 'e']
>>> alph2 = set(alphabet)
{'a', 'c', 'b', 'e', 'd'}
```

### 示例说明

该示例接受数据集 `alphabet` 作为输入，并将其转换为一个集合。因为集合不允许重复的值，多余的字符 `b` 和 `a` 被删除。之后 `alphabet` 被赋给 `alph2`，并且被打印以显示结果。

## 3.4 本章小结

本章介绍了如何操作 Python 提供的多种核心类型。这些类型是元组、列表、字典、集合以及三种特殊类型：None、True 和 False。本章还介绍了可以将字符串当作序列处理的特殊方法。元组和列表是另外两种序列类型。

元组是从 0 开始以固定数值顺序索引的一个数据序列。元组中的引用在元组被创建后不能再改变，也不能再增加或者删除元素。然而，如果元组包含可变元素的数据类型，例如列表，该数据类型的元素是可以改变的。元组在序列中的数据不会改变时很有用，例如，当希望显式地阻止数据被意外改变时。

列表是另一种序列，除了它的元素可被修改之外，它与元组类似。列表的长度可以改变，以容纳使用 `append` 方法新增加的元素，另外也可以通过 `pop` 方法缩减列表的长度。如果希望将一个序列中的数据附加到一个列表中，可以使用列表的 `extend` 方法一次性将序列中的所有元素添加到列表中。

字典是另外一种有索引的数据分组。列表和元组以数值为索引，而字典通过所选的值索引。如果想研究这些索引(叫做键)，可以调用 `keys` 方法。为了研究被引用的数据(叫做值)，可以使用 `values` 方法。这两个方法都返回列表。

集合是项集(0 个或者多个项)，不允许包含重复的值。理论上，它们与字典类似，不过它们只包含键，而没有与键相关联的值。集合的一个用途是从数据集中去除冗余的值。它们也很擅长模仿有穷的数学集合。

其他的数据类型还有 True、False 和 None。True 和 False 是看待 1 和 0 的特殊方式，但当希望测试某件事为真还是为假时，显式地使用名称 True 和 False 总是正确的做法。None 是 Python 内置的只与它自己相等的特殊值，当函数不能返回任何值(例如 True、False、字符串或者其他值)时得到的返回值就是 None。

### 本章要点：

- 变量是用来引用数据的名称。
- 利用语法：变量名称=“某值”，可以创建一个变量。
- 可以通过将一个变量赋给另一个变量来复制它的值：变量名称=要复制的变量名称。
- 元组存储多个数据并且不可以更改。
- 列表也是数据序列，与元组不同的是，可以改变它们的值。
- 字典与列表和元组类似。它是另外一种数据容器类型。元组和列表以数值顺序索引，而字典通过所选择的名称索引。这些名称可以是字母、数值、字符串或符号，适合自己需要即可。

## 3.5 习题

在 Python Shell 中完成如下任务：

1. 使用超市奶制品部的 4 个元素创建一个列表 `dairy_section`。

2. 打印由 `dairy_section` 列表的第一个和最后一个元素组成的字符串。
3. 创建元组 `milk_expiration`，它包含 3 个元素：最近一罐牛奶的过期日期的年、月和日。
4. 以字符串“This milk carton will expire on 12/10/2009”的形式输出元组 `milk_expiration` 中的值。
5. 创建空字典 `milk_carton`。添加如下的键/值对。可以虚构值，也可以使用一罐真正的牛奶：
  - `expiration_date`: 将它设为 `milk_expiration` 元组。
  - `fl_oz`: 将它设为您所依据的牛奶罐的尺寸。
  - `Cost`: 将它设为这罐牛奶的价格。
  - `brand_name`: 将它设为使用的牛奶的品牌名称。
6. 打印字典 `milk_carton` 中所有元素的值，使用字典中的值，而不要使用元组 `milk_expiration` 中的数据。
7. 根据 `milk_carton` 的价格，显示如何计算 6 罐牛奶的价格。
8. 创建列表 `cheeses`。列出可以想到的所有奶酪。将该列表附加到列表 `dairy_section` 中，并且查看 `dairy_section` 的内容。之后从中删除 `cheeses` 列表。
9. 如何计算 `cheeses` 列表中奶酪的个数？
10. 打印第一个奶酪名称的前 5 个字母。





## 第 II 部分

# Python 语言和标准库

第 4 章：做出决策

第 5 章：函数

第 6 章：类与对象

第 7 章：组织程序

第 8 章：文件和目录

第 9 章：Python 语言的其他特性

第 10 章：创建模块

第 11 章：文本处理

资源分享网  
PDG



# 第 4 章

## 做出决策

到目前为止，我们学习了如何直接操作数据，以及如何通过与数据绑定的名称操作它们。对如何手动操作这些数据类型有了基本的理解后，就可以开始实践所学到的关于数据类型的知识，并使用数据做出决策。

本章将介绍 Python 如何利用 True 和 False 做出决策，以及如何根据一个条件是 True 还是 False 做出更复杂的决策。

本章将介绍：

- 如何创建可以使用循环重复相同操作的情形，以自动逐步遍历列表、元组和字典。
- 如何协同使用字典、列表和元组，以探究字典中的内容。
- 如何使用异常处理编写程序，以应对可能在程序内部处理的问题。

### 4.1 比较两个值是否相等

第 3 章中出现了 True 和 False，但没有介绍如何使用它们。True 和 False 是比较两个值、问句以及执行其他一些操作的结果。然而，可以被赋值和命名的任何事物都可以用一组返回 True 或 False 的比较运算来比较。

#### 试一试

#### 比较两个值是否相等

用两个等号测试是否相等。记住，单个等号将数据绑定到一个名称，与这里希望执行的操作不同，这里将得出 True 或者 False：

```
>>> 1 == 1
True
>>> 1 == 2
False
```

#### 示例说明

当进行相等性比较时，Python 比较双等号两端的值。如果数值不同，结果为 False。如果数值相同，结果为 True。

即使数值类型不同, Python 仍然能够比较它们并且给出正确的答案:

```
>>> 1.23 == 1
False
>>> 1.0 == 1
True
```

也可以用双等号来测试两个字符串的内容是否相同, 甚至可以将该测试限定在字符串的某个局部范围内(回忆上一章中介绍的分片, 它复制了所引用的字符串的部分内容, 因此实际上比较的两个字符串代表了分片所涵盖的内容):

```
>>> a = "Mackintosh apples"
>>> b = "Black Berries"
>>> c = "Golden Delicious apples"
>>> a == b
False
>>> b == c
False
>>> a[-len("apples"):-1] == c[-len("apples"):-1]
True
```

在 Python 中, 序列也可以用双等号比较。Python 认为, 如果每个序列中同一位置的每个元素都相同, 那么两个序列相等。因此, 如果有两个序列中分别包含 4 项, 这 4 项的数据相同, 但顺序不同, 那么这两个序列不相等:

```
>>> apples = ["Mackintosh", "Golden Delicious", "Fuji", "Mitsu"]
>>> apple_trees = ["Golden Delicious", "Fuji", "Mitsu", "Mackintosh"]
>>> apples == apple_trees
False
>>> apple_trees = ["Mackintosh", "Golden Delicious", "Fuji", "Mitsu"]
>>> apples == apple_trees
True
```

另外, 字典也可以比较。像列表一样, 一个字典中的每个键与值(一组)必须与另外一个字典中的键与值一一对应, 其中第一个字典中的键与第二个字典中的键相等, 第一个字典中的值也与第二个字典中的值相等:

```
>>> tuesday_breakfast_sold = {"pancakes":10, "french toast": 4, "bagels":32,
"omelets":12, "eggs and sausages":13}
>>> wednesday_breakfast_sold = {"pancakes":8, "french toast": 5, "bagels":22,
"omelets":16, "eggs and sausages":22}
>>> tuesday_breakfast_sold == wednesday_breakfast_sold
False
>>> thursday_breakfast_sold = {"pancakes":10, "french toast": 4, "bagels":32,
"omelets":12, "eggs and sausages":13}
>>> tuesday_breakfast_sold == thursday_breakfast_sold
True
```

## 4.2 比较两个值是否不相等

有一个运算与相等比较相反。如果将感叹号和等号一起使用，可以要求 Python 比较两个值是否不相等(使用与相等比较同样的规则)，如果不相等，得到 `True` 值。

### 试一试

#### 比较两个值是否不相等

尝试如下示例，观察 Python 如何对这些比较求值：

```
>>> 3 == 3
True
>>> 3 != 3
False
>>> 5 != 4
True
```

#### 示例说明

如果一对数用 `==` 比较时得到 `True` 值，在这里将得到 `False`；如果两个数在用 `==` 比较时得到 `False`，在这里将得到 `True`。

这些规则同样适用于更加复杂的类型，如序列和字典：

```
>>> tuesday_breakfast_sold != wednesday_breakfast_sold
True
>>> tuesday_breakfast_sold != thursday_breakfast_sold
False
```

像数值一样，这些类型当用 `==` 比较为 `True` 时，在用 `!=` 比较时将为 `False`。

## 4.3 比较两个值的大小

相等性比较并不是找出希望知道的信息的唯一方法。有时，希望了解某物的数量是否大于另外一个，或者某个值是否小于另外一个值。Python 提供了大于和小于运算，可以用 `>` 和 `<` 进行调用。这些符号与数学书上介绍的符号相同，用于比较左边的值是否大于(`>`)或者小于(`<`)右边的值。

### 试一试

#### 比较大于和小于

```
>>> 5 < 3
False
>>> 10 > 2
True
```

### 示例说明

左边的值与右边的值进行比较。也可以比较字母。在某些条件下，这些比较运算也许会得出意外的结果，例如尝试比较字母和数值(这个问题在许多情形下不会发生，所以您的期望与 Python 的期望不尽相同)。字母表中字母的值大概以这样的方式排序：大写的“A”是最小的字母，之后是“B”，之后是“C”一直到“Z”。随后是小写字母，其中“a”是最小的小写字母，“z”是最大的小写字母。然而，“a”大于“Z”：

```
>>> "a" > "b"
False
>>> "A" > "b"
False
>>> "A" > "a"
False
>>> "b" > "A"
True
>>> "Z" > "a"
False
```

如果希望比较两个包含多个字符的字符串，Python 将观察每个字母，直到找到一个不同的字母为止。比较的结果将取决于不同的字母。如果两个字符串完全不同，第一个字母就将决定结果：

```
>>> "Zebra" > "aardvark"
False
>>> "Zebra" > "Zebrb"
False
>>> "Zebra" < "Zebrb"
True
```

通过使用字符串的特殊方法 `lower`，可以避免比较两个相似单词由于大小写不同而引起的问题，`lower` 方法可以将调用它的字符串中的所有字母都变为小写，然后返回一个新字符串。还有一个对应的 `upper` 方法。它们适用于 Python 中的所有字符串：

```
>>> "Pumpkin" == "pumpkin"
False
>>> "Pumpkin".lower() == "pumpkin".lower()
True
>>> "Pumpkin".lower()
'pumpkin'
>>> "Pumpkin".upper() == "pumpkin".upper()
True
>>> "pumpkin".upper()
'PUMPKIN'
```

注意，也可以用如下方式编写上述代码：

```
>>> "Pumpkin".lower() == "pumpkin"
True
```

由于“pumpkin”已经是小写的，就没有必要再改变它，尽管这样做可能更安全。例如，如果某人错误地输入了一个字符串，将字符串中的某个字母大写，将两者都进行大小写转换可以避免一些错误。观察如下代码：

```
>>> "Pumpkin".lower() == "puMpkin"
False
>>> "Pumpkin".lower() == "puMpkin".lower()
True
```

当通过名称引用字符串时，仍旧可以访问字符串通常具有的所有方法：

```
>>> gourd = "Calabash"
>>> gourd
'Calabash'
>>> gourd.lower()
'calabash'
>>> gourd.upper()
'CALABASH'
```

### 大于等于和小于等于

大于和小于各有一个有用的变体，即大于等于和小于等于。以某种方式将两个符号结合使用时显得很有意义：

```
>>> 1 > 1
False
>>> 1 >= 2
False
>>> 10 < 10
False
>>> 10 <= 10
True
```

## 4.4 对真值和假值取反

在创建一种情形并比较其结果时，有时希望知道结果是否为真，有时希望知道结果是否不为真。Python 提供了一个操作来实现相反的情形：使用单词 **not** 将会对它后面的值取反。

### 试一试

### 对测试的结果取反

```
>>> not True
False
>>> not 5
```

```
False
>>> not 0
True
>>> Not True
SyntaxError: invalid syntax (<pyshell#30>, line 1)
```

注意最后一行代码中的错误。一定不要将 `not` 运算符大写，否则将得到一条与此处类似的错误消息。

### 示例说明

`not` 运算符用于任何得出 `True` 或者 `False` 结果的测试中。然而，第3章中提到，任何非零值都被看做 `True`，因此可以在预期之外或者不一定有意义的情形下使用 `not`：

```
>>> not 5 > 2
False
>>> not "A" < 3
True
>>> not "A" < "z"
False
```

## 4.5 观察多个比较运算的结果

也可以将多个运算的结果合并，这使得程序可以通过对多个运算求真值，做出更加复杂的决策。

一种组合是 `and` 运算，它的含义是：“如果左边的运算、值或者对象为 `True`，接着对右边求值。如果左边不为 `True`，就停止运算并且输出结果 `False`，不再继续运算”。

```
>>> True and True
True
>>> False and True
False
>>> True and False
False
>>> False and False
False
```

另外一种组合运算是 `or` 运算。使用 `or` 告诉 Python 对左边的表达式求值，如果它为 `False`，Python 将继续对右边的表达式求值。如果它为 `True`，Python 将停止对更多的表达式求值：

```
>>> True or True
True
>>> True or False
True
>>> False or True
True
>>> False or False
False
```

您也许想基于希望发生的动作将多个这样的运算放在一起。这些情形下，从最左边的 `and` 或者 `or` 开始求值，根据之前的规则继续对后续运算求值——换言之，直到在 `and` 运算中求出 `False` 值，或者在 `or` 运算中求出 `True` 值。

### 如何做出决策

Python 有一种非常简单的做出决策的方法。用于做出决策的保留字是 `if`，它后面跟着一个判断条件是否为真的测试，该测试以冒号结束，所以在这里可以看到它被写作 `if...:`。它可用于求值为 `True` 和 `False` 的任何情形，表示“如果某条件为真，执行以下操作”：

```
>>> if 1 > 2:
...     print("No it is not!")
...
>>> if 2 > 1:
...     print("Yes it is!")
...
Yes, it is!
```

仅当 `if` 和冒号之间的语句的值为 `True` 时，缩进的语句才会被 Python 访问并求值。`if` 语句中的缩进表明接下来的代码虽然是程序的一部分，但只有在合适的条件发生时才被执行。对于 `if...:` 语句，合适的条件是指当比较运算的结果为 `True` 时。

您已经看到了 Python 在代码表现中最具有特色的一面，这也是大部分人在使用 Python 时容易发表评论的一个方面。

在 Python 程序中看到冒号时，它指出 Python 进入了程序中与其余部分相对独立的一部分。此时，缩进变得很重要。Python 通过缩进知道一个特定的代码块与周围的代码保持独立。所用的空格数很重要，面向 Python 的代码编辑器将总是帮助保持正在编写的代码具有正确的缩进。空格数很重要，因此应该使用编辑器确定缩进，并且不要手动改变空格数。

后面将看到更多的与冒号成对出现的关键字；在这些情形下，需要注意缩进。当程序改变了缩进，导致 Python 无法理解时，它会显示一个错误。

也可以将一个 `if...:` 语句放在另外一个 `if...:` 语句中，以做出比使用 `and` 和 `or` 更加复杂的决策，因为使用 `if...:` 可以在对缩进的 `if...:` 语句求值之前执行任意一组需要的语句。

### 试一试

#### 在测试中嵌套测试

尝试如下示例，其中一个 `if...:` 语句出现在另一个当中：

```
>>> omelet_ingredients = {"egg":2, "mushroom":5, "pepper":1, "cheese":1,
...                       "milk":1}
>>> fridge_contents = {"egg":10, "mushroom":20, "pepper":3, "cheese":2,
...                    "tomato":4, "milk":15}
>>> have_ingredients = [False]
>>> if fridge_contents["egg"] > omelet_ingredients["egg"]:
```

```

...     have_ingredients[0] = True
...     have_ingredients.append("egg")
...
>>> print(have_ingredients)
[True, 'egg']
>>> if fridge_contents["mushroom"] > omelet_ingredients["mushroom"]:
...     if have_ingredients[0] == False:
...         have_ingredients[0] = True
...         have_ingredients.append("mushroom")
...
>>> print(have_ingredients)
[True, 'egg', 'mushroom']

```

### 示例说明

当一个条件经 `if...` 测试之后，如果还有下一级缩进，Python 将继续对在该缩进中放置的代码求值。如果第一个 `if...` 不为真，它下面的任何代码都不会被求值，而将被完全跳过。

然而，如果第一个 `if...` 语句为真，同级的第二个条件将被计算。比较的结果仅决定了它下面缩进的代码是否即将被执行。同级别的以及上层代码不会停止，除非有一些特殊的事件发生，例如，一个错误或者一个阻止程序继续运行的条件。

如从 `print()` 函数看见的那样，当 Python 检查发现冰箱里的鸡蛋和蘑菇多于煎蛋卷配方所需的鸡蛋和蘑菇时，就会将这两项添加到 `have_ingredients` 变量中。

为了完成示例，可以输入如下代码(如果希望用计算机表示一个煎蛋卷的话)：

```

>>> if fridge_contents["pepper"] > omelet_ingredients["pepper"]:
...     if have_ingredients[0] == True:
...         have_ingredients[0] = False
...         have_ingredients.append("pepper")
...
>>> if fridge_contents["cheese"] > omelet_ingredients["cheese"]:
...     if have_ingredients[0] == False:
...         have_ingredients[0] = True
...         have_ingredients.append("cheese")
...
>>> if fridge_contents["milk"] > omelet_ingredients["milk"]:
...     if have_ingredients[0] == True:
...         have_ingredients[0] = False
...         have_ingredients.append("milk")
...
>>> if have_ingredients[0] == True:
...     print("I have the ingredients to make an omelet!")
...
I have the ingredients to make an omelet!

```

可以以 `if...` 开头，使用 `elif...` 创建一个测试链。`elif...` 仅当前面的条件不满足时，才允许测试多个条件。如果使用一系列的 `if...` 语句，它们将被执行。如果在 `if...` 后面使用一个 `elif...` 语句，那么 `elif...` 仅在 `if...` 语句返回 `False` 时被求值。

```
>>> milk_price = 1.50
>>> if milk_price < 1.25:
...     print("Buy two cartons of milk, they're on sale")
... elif milk_price < 2.00:
...     print("Buy one carton of milk, prices are normal")
... elif milk_price > 2.00:
...     print("Go somewhere else! Milk costs too much here")
...
Buy one carton of milk, prices are normal
```

还可以插入一个贯穿(fall-through)语句来处理之前的所有测试都没有返回 True 的情形: else:语句。如果 if...:和 elif...:语句都没有值为 True 的测试条件, else:语句将被调用:

```
>>> OJ_price = 2.50
>>> if OJ_price < 1.25:
...     print("Get one, I'm thirsty.")
... elif OJ_price <= 2.00:
...     print("Ummm... sure, but I'll drink it slowly.")
... else:
...     print("I don't have enough money. Never mind.")
...
I don't have enough money. Never mind.
```

## 4.6 循环

您已经看到有很多次都需要检查和比较序列或字典中的每个元素。手动完成这项工作异常乏味,而且难免出错,即使是一个打字飞快的打字员也不例外。而且,如果手动输入这些代码,将不可避免地发生输入错误,并且当正在计算的某些变量或者值在其他地方被更改时,手动输入的代码很难相应地做出改变。

为了执行重复的任务,Python 提供了两种循环运算。两者很相似,实际上,它们几乎是等同的,但是每一种运算都从不同的角度考虑问题,因此应该同时掌握这两种运算。

### 4.6.1 重复执行操作

可以启动和控制重复任务的两种运算是 while 和 for。while 运算测试一个条件的真值,因此表示为 while...:。for 运算使用一个列表中的所有值,因此表示为 for...in...:。

while...运算首先检查它要测试的条件(位于 while 与:之间的...),如果条件为 True,它将首次对其缩进的语句求值。当它到达缩进的代码块(该代码块也可以包含其他的缩进代码块)的末尾时,将再一次对测试条件求值,看其是否仍旧为 True。如果是,再一次重复动作;然而,如果为 False,Python 将离开缩进的部分,并继续计算程序中 while...:之后的部分。如果在测试条件中用到了名称,在第一次循环和下一次循环之间(再下一次,等等),这个名称引用的值可能会发生变化,直到某些可以让程序停止的因素出现。

## 试一试

## 使用 while 循环

```
>>> i = 10
>>> while i > 0:
print("Lift off in:")
print(i)
i=i - 1
```

```
Lift off in:
10
Lift off in:
9
Lift off in:
8
Lift off in:
7
Lift off in:
6
Lift off in:
5
Lift off in:
4
Lift off in:
3
Lift off in:
2
Lift off in:
1
```

## 示例说明

之前的代码创建了一个名为 `i` 的变量，并且给它赋值 10(注意，该变量可以使用任何合法的名称)。接下来，创建了 `while` 循环，表明“当 `i` 的值大于 0 时，执行如下操作”。之后让 Python 打印语句 `Lift off in:`，并输出 `i` 的当前值。最后，每次循环都将 `i` 减 1，导致该序列一次又一次地执行，直到 `i` 的值等于 0。

如之前所示，用 `for...in...` 形式的循环可以实现相同的结果，它和 `while...` 非常相似，但省略了一些步骤。在第一部分 `for...` 中，同样赋予一个变量名称(再一次使用 `i`，这种做法很常见，因为 `i` 是 `index` 的缩写)。在第二部分 `in...` 中，提供了一个序列，例如一个列表、元组或者这里所用的 `range`，可以取出 `range` 中的每个元素并将它的值赋给第一部分给定的变量名称：

```
>>> for i in range(10, 0, -1):
print("T-minus: ")
print(i)
...
T-minus:
10
```

```
T-minus:
9
T-minus:
8
T-minus:
7
T-minus:
6
T-minus:
5
T-minus:
4
T-minus:
3
T-minus:
2
T-minus:
1
```

可以看到，这个循环与 `while` 循环的工作方式类似，得到了几乎相同的结果(如果没有改变打印的文本的话，它将返回与 `while` 完全一致的结果)。这个版本的 `for` 循环稍微有点复杂。如下代码所示为一个简单的版本：

```
>>>for i in range(10):
print(i)
...
1
2
3
4
5
6
7
8
9
```

这里简单地告诉 `for` 循环不停地迭代以重复整个过程，直到 `i` 等于 10。因为使用了 `range`，它自动给变量加 1，导致程序运行一次之后又循环了 9 次。

### 4.6.2 终止循环

术语“无穷循环”指的是一段永远重复的代码。一个简单的例子是设置 `while...:` 语句，让它测试一个永远为 `True` 的条件。例如，仅使用 `True` 就可以实现这种效果。观察如下代码即可，不要直接输入它们：

```
>>> while True:
...     print ("You're going to get bored with this quickly")
... 
```

```
You're going to get bored with this quickly
You're going to get bored with this quickly
You're going to get bored with this quickly
You're going to get bored with this quickly
You're going to get bored with this quickly
```

上面的代码会一直继续下去，直到中断它。初看起来让某个操作一直重复不太方便，但有时可能希望如此，例如，在程序中使用一段循环代码等待用户输入，当用户输入完毕后，返回等待。

然而，有时希望知道某些特定的条件是否满足，例如，一天的某个恰当的时间是否到来，水是否用完，是否没有足够的鸡蛋做煎蛋卷等等，这时循环可以被打断，即使在 `while...` 顶部没有显式的测试，或者 `for...in...` 中正在使用的列表没有终点。

可以使用 `break` 语句退出无穷循环。尝试下面的代码，并确保缩进与这里显示的缩进匹配：

```
>>>age=0
>>> while True:
how_old=input("Enter your age: ")
if how_old=="No":
    print("Don't be ashamed of your age!")
    break
num=int(how_old)
age=age+num
print("Your age is :")
print(age)
print("That is old!")
...
Enter your age: 1
Your age is :
1
That is old!
Enter your age: 2
Your age is :
3
That is old!
Enter your age: -3
Your age is :
0
That is old!
Enter your age: 50
Your age is :
50
That is old!
Enter your age: No
Don't be ashamed of your age!
>>>
```

在上面的程序中，`while` 总是等于 `True`，因此，如果不加处理的话，`while True` 语句将

一直循环下去。为了解决这个问题，给用户提示了一些信息，这里提示的是年龄。只要他们输入一个数，程序就会将输入的数加到他们的年龄中，显示一个总的年龄，使得看起来他们都变老了(除非他们很机智，输入一个负数，这种情况下他们才会变年轻!)。如果他们仅仅输入数值，程序将一直运行下去。然而，如果他们输入一个字符串 No，程序将打印“Don't be ashamed of your age!”，然后退出循环。

注意 `print` 语句和 `break` 语句的位置。如果将 `print` 放在 `break` 之后而不是之前，那么程序每次循环时它都执行，而不是仅在退出循环的时候执行。因此要注意语句放置的位置!

如果使用 `break`，它仅从最近的一个循环中退出。如果有一个 `while...` 循环，它包含一个缩进的 `for...in...` 循环，`for...in...` 中的 `break` 将不会退出 `while...` 循环。

`while...` 和 `for...in...` 循环在末端都可以有一个 `else:` 语句，但它仅在循环不是由 `break` 语句退出时才会被运行。这种情形下，`else:` 叫做 `done` 或者 `on_completion` 之类的名称可能更合适一些，但是 `else:` 是一个较方便的名词，因为前面已经见到过它，并且它也不难记住。

### 试一试

#### 循环时使用 `else`

```
>>> for food in ("pate", "cheese", "crackers", "yogurt"):
...     if food == "yogurt":
...         break
...     else:
...         print("There is no yogurt!")
...
>>> for food in ("pate", "cheese", "crackers"):
...     if food == "yogurt":
...         break
...     else:
...         print("There is no yogurt!")
...
There is no yogurt!
```

#### 示例说明

每个示例中，都有一个测试判断是否还有酸奶。如果有，`while...` 会用一个 `break` 终止。然而，在第二个循环中，列表中没有酸奶，因此当循环到达列表尾部时将终止，`else:` 条件会被调用。

循环还有另外一个常用的特性：`continue` 语句。使用 `continue` 可以告诉 Python 并不希望循环终止，而是希望跳过当前循环的剩余部分，如果当前是在 `for...in...` 循环中，条件和列表要被重新求值以进行下一轮循环。

### 试一试

#### 用 `continue` 继续循环

```
>>> for food in ("pate", "cheese", "rotten apples", "crackers", "whip cream",
... "tomato soup"):
...     if food[0:6] == "rotten":
...         continue
```

```
...     print("Hey you can %s" % food)
...
Hey, you can eat pate
Hey, you can eat cheese
Hey, you can eat crackers
Hey, you can eat whip cream
Hey, you can eat tomato soup
```

#### 示例说明

因为使用了一个 `if...` 测试来判断 `food` 列表中每一项的第一部分是否包含字符串“rotten”，所以“rotten apples”元素会被 `continue` 跳过，而其余的内容将被打印出来，表示可以安全食用。

## 4.7 处理错误

第2章和第3章中已经介绍了 Python 如何报告错误。错误通常包含大量与发生的错误和失败的原因有关的信息：

```
>>> fridge_contents = {"egg":8, "mushroom":20, "pepper":3, "cheese":2,
"tomato":4, "milk":13}
>>> if fridge_contents["orange juice"] > 3:
...     print("Sure, let's have some juice!")
...
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    if fridge_contents["orange juice"] > 3:
KeyError: 'orange juice'
```

噢！目前冰箱中没有橘子汁，但如果不用终止程序就能知道这一信息会好一些。

前面已经介绍了一种找出字典中出现的所有键的方法，可以使用字典的 `keys` 方法，然后在返回的键列表中搜索，以判断某个希望的键是否出现。然而，没有理由不采用捷径。上述代码中的最后一行显示的错误是：

```
KeyError: 'orange juice'
```

这说明 Python 遇到的错误与字典 `fridge_contents` 中的键相关。可以利用 Python 指出的错误让程序防范这一类错误。可以用特殊词 `try:` 告诉 Python 要做好出错的准备。

#### 使用 `try:` 语句

`try:` 语句设立了这样一种情形，其中 `try:` 语句后面可以跟一个 `except:` 语句。每个 `except:` 语句都处理错误，错误也被正式地称作异常，当 Python 对 `try:` 语句中的代码求值时会抛出异常，而不是使程序失败。首先使用 `except:` 处理一种类型的错误，例如在试图检查冰箱时得到的 `KeyError` 错误。

有多种类型的异常，每个异常的名称都反映了发生的问题，并且也有可能反映出异常所发生的条件。因为字典有键与值，`KeyError` 表示正在向字典请求的键不存在。类似地，一个 `TypeError` 表明 Python 期待某种类型的数据(例如，一个字符串或者一个整型值)，但是提供给它的却是另外一种不能满足要求的类型。

另外，当异常发生时，可以访问原本在交互运行程序的时候程序停止时会看到的消息。学到更多知识后，在需要的条件下，您将能够定义自己的异常类型。

只有一行代码用于处理错误，这看起来很受限制，但第 5 章将介绍如何编写自己的函数，这样可以更加灵活地处理错误。

```
>>> fridge_contents = {"egg":8, "mushroom":20, "pepper":3, "cheese":2,
"tomato":4, "milk":13}
>>> try:
...     if fridge_contents["orange juice"] > 3:
...         print("Sure, let's have some juice!")
... except KeyError:
...     print("Awww, there is no juice. Let's go shopping!")
...
Aww, there's no juice. Lets go shopping
```

您也许发现需要打印关于错误本身更多的信息，这些信息是可以访问的。

### 试一试

### 创建异常以及对异常的说明

```
>>> fridge_contents = {"egg":8, "mushroom":20, "pepper":3, "cheese":2,
"tomato":4, "milk":13}
>>> try:
...     if fridge_contents["orange juice"] > 3:
...         print("Sure, let's have some juice")
... except (KeyError) as error:
...     print("Woah! There is no %s" % error)
...
Woah! There is no 'orange juice'
```

### 示例说明

因为在字典 `fridge_contents` 中没有键“orange juice”，Python 抛出了一个 `KeyError` 异常，说明没有这样的键。除此之外，还指定了名称 `error`，Python 将用它引用一个字符串，该字符串包含了 Python 可以提供的错误信息。我们通过 `as` 关键字将 `KeyError` 的值赋给 `error`。在这种情形下，字符串与已请求但未在字典 `fridge_contents` 中出现的键有关(这里是“orange juice”)。

有时也许会用相同的方式处理多种错误，在这种情况下，可以用一个元组包含有关的所有异常类型：

```
>>> fridge_contents = {"egg":8, "mushroom":20, "pepper":3, "cheese":2,
"tomato":4, "milk":13}
```

```
>>> try:
...     if fridge_contents["orange juice"] > 3:
...         print("Sure, let's have some juice")
... except (KeyError, TypeError) as error:
...     print("Woah! There is no %s" % error)
...
Woah! There is no 'orange juice'
```

如果需要处理一个异常，但是希望以什么都不做的方式处理它(原因可能是故障不严重)，Python 允许通过特殊词 `pass` 忽略这种情形：

```
>>> fridge_contents = {"egg":8, "mushroom":20, "pepper":3, "cheese":2,
"tomato":4, "milk":13}
>>> try:
...     if fridge_contents["orange juice"] > 3:
...         print("Sure, let's have some juice")
... except (KeyError) as error:
...     print("Woah! There is no %s" % error)
... except (TypeError):
...     pass
...
Woah! There is no 'orange juice'
```

还有一个 `else:` 语句可以放在 `try:` 代码块的末端。在没有捕获到任意异常时，它将被执行。如前所述，`else` 这个名称的描述性可能不如 “in case it all works” 或 “all\_clear” 这样的名称，但是到现在已经可以看到，`else:` 是可以处理各种情况的一种灵活的机制，表示“如果某事发生，则如何处理”。在需要时，随时可以使用它。

## 4.8 本章小结

本章介绍了 Python 提供的做出决策的方法。任意得出 `True` 或者 `False` 结果的运算，都可以用在 `if...:` 语句中来判断一个程序是否要对一段缩进的代码求值。

缩进在 Python 中扮演着重要角色。即使在交互的 Python Shell 中，缩进中的空格数也很重要。

本章讨论了在循环中使用序列和字典中元素的知识。通过使用循环，可以对一个列表的每个元素执行运算并且根据每个列表元素做出决策。

Python 提供了两种类型的循环，分别是 `while...:` 循环和 `for...in...:` 循环。它们持续执行相似的工作，直到某个条件导致它们结束。两者的区别在于允许它们对缩进的代码块求值的条件不同。`while...:` 循环在它的测试中只测试是 `True` 还是 `False`，而 `for...in...:` 循环将接受在 `in...:` 部分提供的序列，序列中的第一个到最后一个元素将被赋给 `for...:` 部分中提供的值。

这两种类型的重复循环在它们的测试条件满足之前，可以通过 `break` 语句退出。`Break` 语句导致正在执行的循环停止，不再对循环的代码块中其余的代码求值。然而，如果执行

一个 `break` 语句，循环中可选的 `else` 条件将不被执行。除了 `break` 之外，还有一个 `continue` 语句，它将跳过当前循环的剩余部分，然后返回到循环的顶部并对下一个测试条件求值。

本章还介绍了其他类型的做出决策的方法，即处理 Python 用来报错的异常。如果没有处理异常，错误会导致程序在出错的地方停下来。然而，如果将可能会导致错误的代码缩进并放到 `try` 的下面，就可以阻止程序退出，甚至可以处理错误并且使程序继续运行。在 `except...` 语句中指定预计会碰到的错误，其中，提供的第一个值定义了错误的类型(如果提供的是错误类型的元组，就是多个错误类型)；也可以选择在该值的后面提供词 `as`，以及用来引用包含错误信息的数据的名称。

#### 本章要点：

- 可以用两个等号(==)测试相等性。如果答案是 `True`，则返回 `True`。如果是 `False`，则返回 `False`。双等号也可以用来判断两个变量是否保存相同的数据。
- 如果希望知道两个值是否不等，可以使用不等运算符(!=)。在这种情形下，`True` 表示被比较的两个值不相等，`False` 表示被比较的值是相等的。
- 为了比较两个值的大小，可以分别使用大于(>)和小于(<)运算符。如果需要知道一个值是否大于等于或者小于等于另外一个值，可以使用大于等于运算符(>=)或者小于等于运算符(<=)。
- 如果希望程序根据一个值为 `True` 还是为 `False` 做出决策，可以使用 `if` 语句，它简单地表明，如果这个值为 `True`，就执行下面的操作。为了让程序在一个值为 `True` 时执行一种操作，而在该值为 `False` 时执行另一种操作，可以使用 `if...elif...` 语句。
- 有时需要将一个特定的动作循环很多次。使用 `while` 循环时，只要条件为 `True`，就会一直重复或者迭代某个动作，或者照字面意思讲，当该条件为 `True` 时执行如下操作。
- 如果需要将某个动作循环指定的次数，可以使用 `for` 循环，它用一个循环计数器指定要将一个给定的动作重复多少次。
- `try` 语句后面可以跟一个 `except` 语句。每个 `except` 语句都处理错误，错误的正式名称是“异常”。Python 在计算 `try` 语句中的代码时会抛出异常，而不是简单地失败。

## 4.9 习题

在 Python Shell 的代码编辑器中执行下列任务：

1. 用一系列的 `if...` 语句，通过创建 5 个单独的测试计算从 0~4 的数值是 `True` 还是 `False`。
2. 用一个 `if...` 语句创建一个测试，指出某个值是否包含在 0~9 之间(可以是 0 或者 9)，如果是，则打印一条信息。
3. 使用 `if...`、`elif...` 和 `else:`，创建一个测试，检查一个名称引用的值是否在一个序列的前两个元素中。用 `if...` 测试序列的第一个元素，用 `elif...` 测试序列中第二个元素的值，然后使用 `else:` 语句打印一条消息，指出正在查询的元素是否存在于序列中。
4. 创建一个字典，它包含一个虚构的电冰箱中的食物，将字典命名为 `fridge`。食物的

名称作为键，每个食物项的值是一个字符串，描述了该食物。创建一个名称，引用包含食物名称的字符串。将该名称叫做 `food_sought`。对于字典 `fridge` 中的每个键和值，用 `for...in...` 循环测试冰箱中包含的每个键，将习题 3 中的测试修改为简单的 `if...` 测试(这里不需要 `elif...` 或者 `else:`)。如果找到一个匹配，打印出包含这个键与值的消息，并使用 `break` 退出循环。在 `for` 循环的末端用一个 `else...` 语句打印当元素没有找到时的消息。

5. 修改习题 3，使用 `while...` 循环创建一个单独的叫做 `fridge_list` 的列表，这个列表将包含 `fridge.keys` 给出的值。同样地，使用叫做 `current_key` 的变量，引用循环中由 `fridge_list.pop` 得到的当前元素的值。记住将 `fridge_list.pop` 放到 `while...` 循环的最后一行，这样循环将正常结束。在 `while` 循环的末尾使用与习题 3 的末尾相同的 `else:` 语句。
6. 用一个不存在的键查询习题 3 中创建的字典 `fridge`，引起一个错误。在这样的情况下，`KeyError` 可被用来判断您希望的值是否存在于列表中。修改习题 3 的解决方法，使用 `try:` 代码块而不是 `for...in...`。



# 第 5 章

## 函 数

目前，每次希望完成一项任务时，需要输入所有的程序代码来完成工作。如果需要再次做相同的工作，可以将整个程序重新输入一遍或者将代码放在一个循环中。循环在重复操作时非常有用，但是只是略做了一些小的修改，就在程序的不同部分重复写相同的循环，这并不是明智的做法。

Python 的函数可以将代码段聚集到方便的分组中，在需要的时候可以再次调用它们。本章将介绍：

- 如何创建和使用函数。
- 给出一些指导原则，帮助思考如何创建和组织程序以使用函数。
- 如何编写函数，使随后可以询问它们的工作方式和实现的功能。

### 5.1 将程序放在单独的文件中

当本书中的示例变长时，输入整个代码块开始变成负担。一个错误将导致要重新输入整块代码。在遇到多于 40 行的代码要输入时，您不会再想输入相同的代码。

程序员将所写的程序作为源代码存入文件，这些文件可以被方便地打开、编辑和运行。

为了更加方便，从现在开始，您应当在 Python 的代码编辑器中输入正在使用的程序，并将本书的示例都放到一个文件中，以便以后引用和运行这些示例。一个可行的建议是将该目录命名为“Learning Python”，之后能够以程序出现的章节命名它们。

可以执行两种操作使程序易于运行。所有 Python 文件的第一行应该如此：

```
#!/usr/bin/env python 3.1
```

如果遵循本书附录中的说明，这行代码使得 UNIX 和 Linux 系统可以运行这些脚本。第二件要做的重要的事情是给所有的 Python 文件起一个以.py 为扩展名的名称。在 Windows 系统上，这将给操作系统提供信息，以便使操作系统将这些文件当作 Python 文件启动，而不是仅把它们当作普通的文本文件。例如，如果把前面章节的示例放到自己的文件中，会有一个包含如下文件的文件夹：

```
chapter_1.py
chapter_2.py
chapter_3.py
chapter_4.py
chapter_5.py
```

将第一个程序保存到文件后，代码编辑器开始以不同的颜色和样式显示程序中的某些部分以强调它们。内置的函数和保留字以一种方式处理，字符串以另外的方式处理，而一些关键字又会采用另外一种处理方式。然而，文件中的大部分文本都还是普通的黑色和白色，如图 5-1 所示。

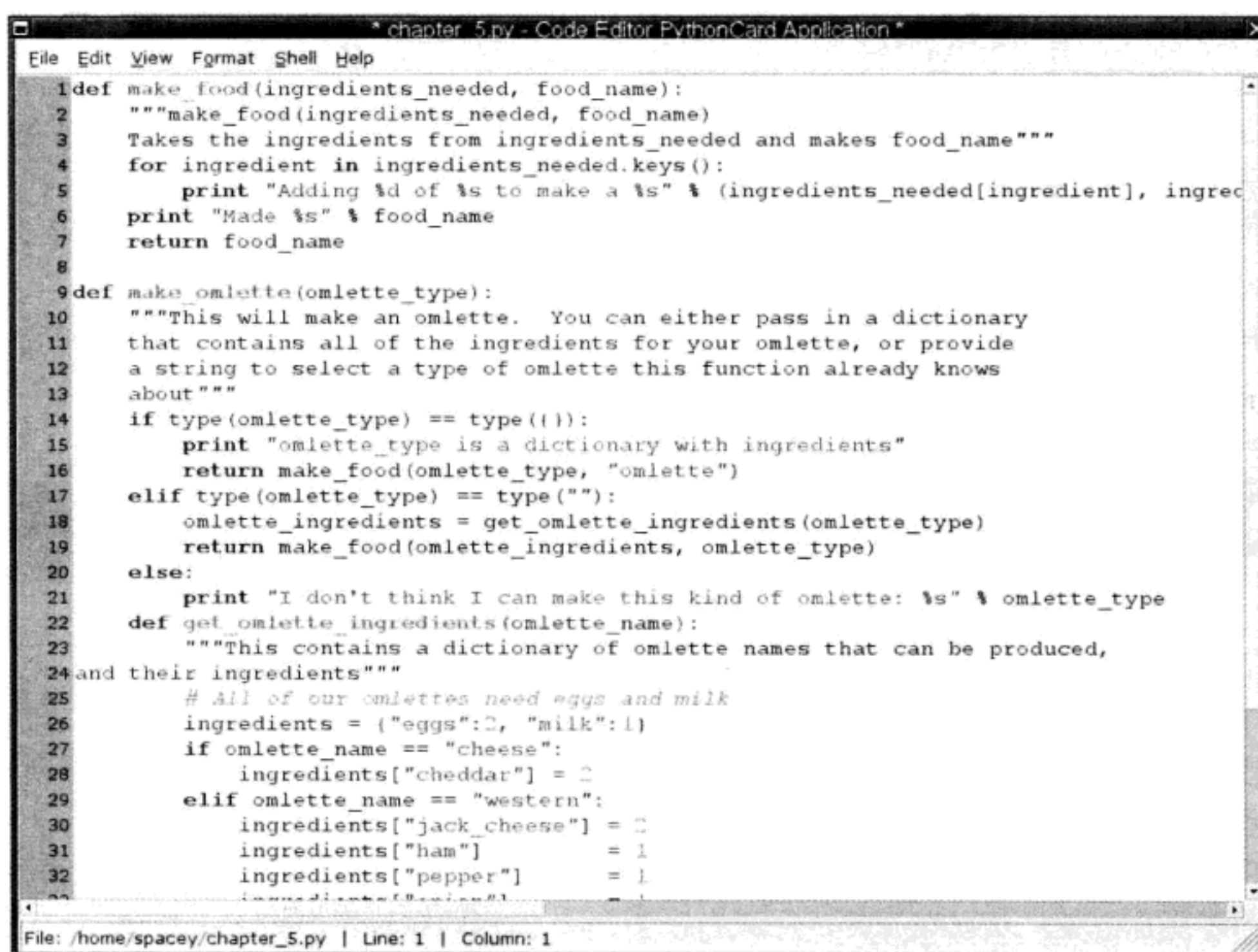


图 5-1

使用这些文件后，只输入一次示例即可。当输入一个例子并且保存它之后，可以用 `python -i <filename>` 运行它。`-i` 告诉 Python 读取程序文件，然后允许您继续与 Python 交互，而不是立刻退出(这是 Python 通常的做法)。在代码编辑器中，可以通过从 File 菜单选择 Run with Interpreter 自动完成这种设置。

### 试一试

### 利用-i 在 Python 中运行程序

为了说明如何利用 `python -i` 或者 Run with Interpreter，向名为 `ch5-demo.py` 的文件中输入如下代码：

```
#!/usr/bin/env python 3.1
a = 10
b = 20

print("A added to B is %d" % (a + b))
```

当用-i 选项调用 Python 时，将处于一个如下所示的 Python 交互会话中：

```
A added to B is 30
>>>
```

### 示例说明

文件 ch5-demo.py 中的代码被计算后，可以继续操作 a 与 b 的值，也可以对它们进行扩展，就像手动输入一样。当示例变长时，这将节省时间。后面的一些代码可能首先在 Shell 中展示，但是可以保存以便以后运行。另外一些出现在文件中的代码，可以保存和运行。之所以使用文件展示程序，原因可能是书中讨论的内容没有很好地表现一种思想，所以最好使您做一些额外的工作，一次又一次地输入相同的代码，或者实际与程序交互。也可能是因为程序太长，在需要对其测试时，使用文件就不用一次次地重复输入。

## 5.2 函数：在一个名称下聚集代码

大多数现代程序语言都提供了将代码聚集在一个名称下的功能，不论何时使用这个名称，它包含的代码将被调用和运算，从而不需要每次都重新输入它们。

为了创建一个包含代码的命名函数，可以使用 def，它可以定义一个代码功能块。

### 试一试

#### 定义一个函数

尝试将下面的代码保存到第 5 章的文件 ch5.py 中。

```
def in_fridge():

    try:
        count = fridge[wanted_food]
    except KeyError:
        count = 0
    return count
```

### 示例说明

当调用 ch5.py(在代码编辑器中按下 F5)时，若其中只定义了 in\_fridge 函数，则不会看到任何输出。然而，该函数将被定义，并且可以从已创建的交互式 Python 会话中调用。

为了使用 in\_fridge 函数，必须确保有一个包含各种食物名称的 fridge 字典。另外，必须有一个名为 wanted\_food 的字符串。使用 in\_fridge 时，可以通过该字符串询问是否有某

种食物。因此，在交互式的会话中，可以用该函数做如下的事情：

```
>>> fridge = {'apples':10, 'oranges':3, 'milk':2}
>>> wanted_food = 'apples'
>>> in_fridge()
10
>>> wanted_food = 'oranges'
>>> in_fridge()
3
>>> wanted_food = 'milk'
>>> in_fridge()
2
```

这不仅有用，而且十分有意义，可以减少工作量。在一个名称下聚集代码块意味着可以简化代码，从而以更快的速度完成更多的工作。输入量减少了，出错的机率也就降低了。

函数是任意现代程序语言的核心部分，它们也是利用 Python 解决问题的关键。

编写程序时，函数可以看做是一个问与答的过程。当它们被调用时，经常会问它们一个问题：“有多少个？”“什么时候？”“这个存在吗？”“这个可以改变吗？”等。作为响应，函数会返回包含答案的值，例如 True、一个序列、一个字典，或者其他类型的数据。如果不是这些值，返回的答案将是一个特殊值 None。

即使一个函数主要做一些简单的工作，通常会有一个隐含问题需要知道。当函数完成任务后，“它成功了吗？”或者“它是如何工作的？”之类的问题通常是您调用函数的方式。

### 5.2.1 选择名称

把函数写好的第一个指导原则是函数的名称能够反映其用途。这些名称应当指出函数的功能，如 Python 中的函数 print、type 和 len。

当确定一个名称时，应当考虑如何在程序中调用它。如果命名一个函数，日后您以及其他人员调用它时，都能很自然地理解它，这将非常好。几周之内忘记函数的具体细节是很正常的，因此日后返回来用它时，函数的名称就成为回忆它的功能的主要手段。

### 5.2.2 在函数中描述函数

在选定函数名称之后，应当再给函数添加一个描述。Python 可以用简单并且有意义的方式完成这件事情。

如果将一个字符串作为函数的第一部分内容，而没用名称引用它，Python 将它存储在函数中，以便以后可以引用它。这个字符串通常叫做 docstring，是 documentation string(文档字符串)的缩写。

函数的文档字符串用于描述函数。很少有计算机软件做了良好的说明。然而，通常情况下，由于 Python 提供了简单的文档字符串特性，相对于用缺乏友好和有约定的其他语言编写的程序，Python 程序中有更多可用的信息。

文档字符串中的文本并不必像其他代码那样遵循缩进规则，因为它仅仅是一个字符串。输入完文档字符串后，即使它可能看起来破坏了缩进，函数的后续部分也仍然必须保

持正确的缩进，记住这一点非常重要。

```
def in_fridge ():
    """This is a function to see if the fridge has a food.
    fridge has to be a dictionary defined outside of the function.
    the food to be searched for is in the string wanted_food"""
    try:
        count = fridge[wanted_food]
    except KeyError:
        count = 0
    return count
```

文档字符串通过函数中的名称引用，仿佛函数是字典一样。这个名称是`__doc__`，它的使用法是在函数名称之后加一个英文句点，再加上`__doc__`。注意，在`doc`前后各有两个下划线(`__`)。

### 试一试

### 显示`__doc__`

现在应当退出上个示例进入的交互式会话，然后重新调用`ch5.py`，因为现在`in_fridge`中加入了文档字符串。此后，可以执行如下操作：

```
>>> print("%s" % in_fridge.__doc__)
This is a function to see if the fridge has a food.
fridge has to be a dictionary defined outside of the function.
the food to be searched for is in the string wanted_food
```

### 示例说明

如其他类型一样，函数也有一些属性，可以通过在函数名称后加句点，再加属性名称来使用。`__doc__`是一个字符串，像其他字符串一样，在一个交互式会话中很容易把它打印出来以便引用。

函数还有其他信息(它保存的这组信息可以使用内置函数`dir`查看)。

`dir`显示您感兴趣的对象(例如函数)的所有属性，包括 Python 内部使用的属性：

```
>>> dir()
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
 '__defaults__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__',
 '__ge__', '__get__', '__getattr__', '__globals__', '__gt__', '__hash__',
 '__init__', '__kwdefaults__', '__le__', '__lt__', '__module__', '__name__',
 '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

可以使用获得`in_fridge.__doc__`引用的数据同样的表示法访问这些属性，但是通常并不需要直接使用其中的大部分属性，尽管使用`type`内置函数研究 Python 如何描述这些元素是很好的练习。

### 5.2.3 不同的位置相同的名称

函数的一个特殊属性在于它是您见到的对引用数据的名称进行划分的第一个示例。这意味着如果函数之外有一个名称，该名称引用一个特定值，这个值可以是字符串、数值、字典、序列或者函数。这些数据享有共同的空间。

例如，如果创建了一个字符串名称，然后在下一行创建了一个字典，并用相同的名称引用这个字典，那么这个名称将不再引用字符串，而是仅引用字典：

```
>>> fridge = "Chilly Ice Makers"
>>> print(fridge)
Chilly Ice Makers
>>> fridge = {'apples':10, 'oranges':3, 'milk':2}
>>> print("%s" % fridge)
{'apples': 10, 'oranges': 3, 'milk': 2}
```

这很合理，但是在函数中使用名称时会有所不同。函数创建了一个新的空间，其中名称可以被重新使用和创建，而不会影响到程序的其他作用域中的相同名称。所以在编写函数时，完全不用担心其他地方或者其他函数中已经用过同样的名称。

因此，编写函数时，函数有自己的一些名称，其他的函数也有自己的名称，它们是分开的。即使两个函数中某个名称完全相同，由于它们在不同的函数中，所以是完全独立实体，引用的是不同的值。

如果即将在一个已知的情形中使用一个函数，并确保它要用的名称已被定义，而且引用了正确的数据，函数就可以通过已经定义的名称访问这个全局数据。Python 的这种能力来源于它将一个名称的可见性分隔到不同的概念区域内。每一个这样的区域叫做一个作用域。

最上面一层作用域中的任何名称在一个低层次作用域中可再次被使用，而不影响上层名称引用的数据：

```
>>> special_sauce = ['ketchup', 'mayonnaise', 'french dressing']
>>> def make_new_sauce():
...     """This function makes a new special sauce all its own"""
...     special_sauce = ["mustard", "yogurt"]
...     return special_sauce
...
```

此处，最上层作用域中有一个 `special_sauce`，函数 `make_new_sauce` 中也有一个 `special_sauce`。当运行时，可以看到全局作用域中的名称没有改变：

```
>>> print("%s" % special_sauce)
['ketchup', 'mayonnaise', 'french dressing']
>>> new_sauce = make_new_sauce()
>>> print(special_sauce)
['ketchup', 'mayonnaise', 'french dressing']
>>> print(new_sauce)
['mustard', 'yogurt']
```

记住，不同的函数完全可以定义相同名称的变量，该名称在两个函数中都有意义，但却引用不同的值，彼此不会产生冲突。

### 5.2.4 添加注释

Python 有一个额外的特性可以帮助记录自己的程序。输入到程序中的全部内容，即使目前为止它还未改变程序的行为(如文档字符串)，Python 也会处理它。即使不被使用的字符串，Python 也会创建它，以便将来使用。

除了不需要的字符串，每门程序语言都有在代码中添加注释的能力，注释不会对程序有影响。它们是供代码使用者阅读的，而不是为 Python 准备的。

任何地方，一行里有#字符且它不出现在字符串中，Python 将忽略它后面的所有内容，而开始对下面一行语句求值，并从此处开始读取剩余部分的程序。

#### 试一试

#### 添加注释

如果交互式地测试注释，可以看到 Python 读它们时与读字符串不同：

```
>>> "This is a string"
'This is a string'
>>> # This is a comment
>>>
>>> "This is a string" # with a comment at the end
'This is a string'
>>> print("# Here is a pound sign within a string, being treated as a
string!")
# Here is a pound sign within a string, being treated as a string!
```

#### 示例说明

当注释单独出现时，Python 忽略它并且返回，询问下一个请求，试着提示输入一个它可以计算的语句。当注释与可以计算的代码出现在同一行时，即使仅仅是个字符串，Python 也知道您已经给了它指令。

注释通常出现在程序文件中。您不太可能不厌其烦地在交互式会话中输入注释，但这是在程序文件中使用它们的方法。

另外，如果想测试一个程序的变化，在会引起错误的一行(或多行)代码之前放置注释符号使其失效非常有用。然而要小心。注释会影响 Python 严格关注的缩进。必须非常仔细地将函数中的注释放在适当的缩进级别，因为如果不这样做，Python 将注释看做结束了该函数或 if...:代码块，或者其他引起缩进的单位，这肯定不是希望出现的情况！

将注释保持在同样的缩进级别，可以使注释更容易阅读，因为注释用于哪部分代码是显而易见的。

### 5.2.5 要求函数使用提供的值

在 in\_fridge 示例中，函数使用的是全局作用域的值。函数 in\_fridge 仅在已经定义的值

上做运算，这些值的名称对于整个程序都可见。这种行为只在较小的程序中有效。

当接触到包含成百上千或者更多行代码(程序的长度经常以它包含的代码行数来衡量)的大程序时，就不能再依靠特定名称的全局可用性，因为这些名称可能在您没参与时按照其他人的决定被更改掉了。相反，可以指定函数在每次被调用时处理提供給它的值。

这些值是规范或者参数，程序依靠它们完成工作。当函数被调用时，这些参数可以是引用了数据的名称，也可以是静态数据，例如像 5 一样的一个数或者一个字符串。所有情况下，实际数据将进入被调用函数的作用域，而不是全局作用域。

本书中的许多示例有不同的改进版本，可以把它们添加到同一个文件中，除非本书明确指示改变正在使用的函数。

不必总是将一个函数的前期版本删掉，因为下一个版本只是对它做了简单的修改。通过比较新旧两个函数，您有机会看到对该函数做的修改。

只要最新的版本在您加载时处于文件的最底部，这个最新版本将被使用。

当自己写程序时，这是一个有用的练习。没有什么会像乱动正在处理的一段代码，之后不记得如何将它改回以前可以工作的状态那样痛苦。

注意，在下面的代码中，`def`(函数的定义)已经被更改，现在它指定该函数需要两个参数，参数名称在函数名称后的元组中指定。这些参数将进入并且保持在 `in_fridge` 函数的作用域内，它们分别是 `some_fridge` 和 `desired_item`。

```
def in_fridge(some_fridge, desired_item):
    """This is a function to see if the fridge has a food.
    fridge has to be a dictionary defined outside of the function.
    the food to be searched for is in the string wanted_food"""
    try:
        count = some_fridge[desired_item]
    except KeyError:
        count = 0
    return count
```

当调用一个带参数的函数时，在 `in_fridge` 函数调用的圆括号中放置指定参数的值或者名称，参数值之间以逗号分隔。前面已经在使用函数 `len` 时看到了这种做法。

## 试一试

## 调用带参数的函数

再一次，通过运行 `python -i ch5.py` 或者使用 Run with Interpreter 重新调用一个交互式的 Python 会话，得到的交互式会话中定义了新的 `in_fridge` 函数。

```
>>> fridge = {'apples':10, 'oranges':3, 'milk':2}
>>> wanted_food = "oranges"
>>> in_fridge(fridge, wanted_food)
3
```

### 示例说明

字典 `fridge` 和字符串 `wanted_food` 作为参数传递给新的 `in_fridge` 函数。当进入该函数的作用域后，`fridge` 引用的字典现在被名称 `some_fridge` 引用。同时，`wanted_food` 引用的字符串“oranges”，在一进入 `in_fridge` 函数作用域后就与名称 `desired_item` 关联。做完该调整后，函数有了正常工作需要的信息。

为了进一步说明工作原理，可以使用没有命名的值，即不通过名称引用的数据：

```
>>> in_fridge({'cookies':10, 'broccoli':3, 'milk':2}, "cookies")
10
```

这些值被带入 `in_fridge` 函数的作用域，并根据函数定义赋值给函数中使用的名称。在函数内部将不再有一个对全局作用域名称的引用。

### 5.2.6 检查参数

打算使用的参数类型可能与函数被调用时提供的类型不同。例如，可能编写了一个函数，期待的参数是一个字典，但偶然传入一个列表，此时函数将一直运行，直到需要访问字典特有的运算。程序因为产生异常而退出。这与其他语言不同，它们尝试确保每个参数的类型是已知的，而且可以检查其正确性。

Python 并不检查哪种类型的数据与函数的参数名称关联。大多数情况下，这不是问题，因为在给定数据上的操作都是某个类型所特定的，如果名称引用的数据类型不正确，程序将不能正常地工作。

例如，如果向 `in_fridge` 传入数值而不是字典，Python 在尝试将数值当作字典访问时，将引发一个 `except` 捕捉不到的错误。产生的 `TypeError` 错误表明 Python 试图操作的类型做不了 Python 期望它做的事情：

```
>>> in_fridge(4, "cookies")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 7, in in_fridge
TypeError: unsubscriptable object
```

这个示例传递给函数一个数值，而函数期待对一个字典进行操作。不论怎样，数值没有将名称作为键查找值的属性。数值没有键，也没有与键对应的值。在任何上下文中，Python 中都不能找到 `4("cookies")`，因此将引起异常。

Python 用词 `unsubscriptable` 指出它不能像在字典中那样找到与一个键关联的值。下标 (`subscripting`) 是描述访问列表、元组或者字典中的元素时的术语，因此在其中任意一个上下文中都可能遇到该错误。

不需要明确定义期待的类型，而允许灵活确定希望如何处理类型的行为可给您带来好处。可以编写一个函数，处理任意期望的类型。还可以编写接受多种类型参数的函数，之后再依据传递给函数的类型确定函数应该如何运行。采取哪种方法取决于需要在程序中做什么。

为了确定某些数据的类型，记住可以使用内置函数 `type`，第2章对它做了介绍。使用该函数的输出，可以在函数的开头部分验证变量类型。

```
def make_omelet(omelet_type):
    """This will make an omelet. You can either pass in a dictionary
    that contains all of the ingredients for your omelet, or provide
    a string to select a type of omelet this function already knows
    about"""
    if type(omelet_type) == type({}):
        print("omelet_type is a dictionary with ingredients")
        return make_food(omelet_type, "omelet")
    elif type(omelet_type) == type(""):
        omelet_ingredients = get_omelet_ingredients(omelet_type)
        return make_food(omelet_ingredients, omelet_type)
    else:
        print("I don't think I can make this kind of omelet: %s" % omelet_type)
```

`make_omelet` 的定义并不能工作，因为它还依赖于其他几个未编写的函数。编程过程中有时会这么做，先创建一些函数的名称，随后再填充它们的内容。在本章后面将看到这些函数，那时上述代码就变得完全可用了。

### 试一试

### 使用 `type` 函数确定更多的类型

在用 `python -i` 或者 `Run with Interpreter` 加载 `ch5.py` 文件之后，输入如下代码：

```
>>> fridge = {'apples':10, 'oranges':3, 'milk':2}
>>> type(fridge)
<class 'dict'>
>>> type({})
<class 'dict'>
>>> type("Omelet")
<class 'str'>
>>> type("")
<class 'str'>
```

### 示例说明

要注意的第一件事情是 `type` 函数返回对象的类。在测试中可以使用这个类的对象，即它可与其他类对象进行比较。注意，Python 中，类和类型是同一事物的不同名称，即它们定义了变量和值的数据类型。

### 试一试

### 使用字符串比较类型

在这里还可以使用另外一个特性。前面已经看到，对于 `print` 函数，Python 中的许多对象可以表示成字符串。这是因为，许多对象都有内置的方法，在需要时可将它们转换成字符串。

例如，另外一种编写前面比较的方法如下：

```
>>> fridge = {'apples':10, 'oranges':3, 'milk':2}
>>> str(type(fridge))
"<class 'dict'>"
>>> if str(type(fridge))=="<class 'dict'>":
...     print("They match!")
...
They match!
```

### 示例说明

由于事先可以知道一个类型的字符串表示是什么样子，可以将那个字符串与一个通过 `str` 函数转换成字符串的对象进行比较。

## 5.2.7 为参数设置默认值

还有一个小窍门可确保函数易于使用。函数的每个参数都要有值。如果没有给所有必需参数赋值，函数将产生一个错误，或者更糟的是，它返回了错误的数据。

为了避免这种情形，Python 允许创建带默认值的函数，这些默认值赋给那些在函数调用时没有显式提供的参数。您已经见过这种行为。例如列表的 `pop` 方法，它既可以被告知处理列表中的一个特定元素，也可以在没有给定任何值的情况下，自动处理最后一个元素。

可以在定义函数时，在参数列表中使用赋值运算符(=)指定默认值。例如，如果希望 `make_omelet` 的变体可以默认地做出一个奶酪煎蛋卷，仅需改变它的定义，其余什么都不用作。

### 试一试

### 设置默认参数

剪切并粘贴整个 `make_omelet` 函数。之后，在函数的新副本中对它的定义做如下修改，它将默认地做奶酪煎蛋卷：

```
def make_omelet2(omelet_type="cheese"):
```

### 示例说明

该定义并不改变函数其余代码的行为。它仅在函数 `make_omelet2` 被调用、但没有定义 `omelet_type` 时设置它的值。

仍然可以用一个字典或者一个不同类型的煎蛋卷指定这个参数。然而，如果 `make_omelet` 用这样的方式定义，可以在不指定任何特定煎蛋卷的类型的情况下调用它，该函数将做出一个奶酪煎蛋卷。

对 `make_omelet` 做相同的操作是将 `make_omelet` 编写为一个以友好和直观的方式运行的函数的第一步。记住，仍然需要编写其他函数！目标是得到如下的输出：

```
>>> make_omelet()
Adding 2 of eggs to make a cheese
Adding 2 of cheddar to make a cheese
Adding 1 of milk to make a cheese
Made cheese
```

```
'cheese'
>>> make_omelet("western")
Adding 1 of pepper to make a western
Adding 1 of ham to make a western
Adding 1 of onion to make a western
Adding 2 of eggs to make a western
Adding 2 of jack_cheese to make a western
Adding 1 of milk to make a western
Made western
'western'
```

如果编写了一个有多个参数的函数，并且希望其中既有必需参数也有可选参数，必须将可选参数放在参数列表的末尾。因为一旦指定某个参数为可选参数，它可能出现也可能不出现。从第一个可选参数起，Python 不能保证那些在可选参数右边的参数都出现。换言之，第一个默认参数后面的所有参数都变成可选参数。这是自动发生的，因此要小心使用这个特性。

### 5.2.8 在函数中调用其他函数

在顶层或者全局作用域声明的函数可以被其他函数以及其他函数包含的函数使用。全局作用域的名称可在任何地方使用，因为最有用的函数需要在其他函数中可用。

为了使 `make_omelet` 函数能像之前看到的那样工作，它依赖的其他函数应当可用，以便被 `make_omelet` 使用。

它的工作方式是：首先，一个函数扮演了一本食谱的角色。它被给予一个指定煎蛋卷类型的字符串，并返回一个包含所有成分及其分量的字典。该函数将被叫做 `get_omelet_ingredients`，它需要一个参数，即煎蛋卷的名称：

```
def get_omelet_ingredients(omelet_name):
    """This contains a dictionary of omelet names that can be produced,
    and their ingredients"""
    # All of our omelets need eggs and milk
    ingredients = {"eggs":2, "milk":1}
    if omelet_name == "cheese":
        ingredients["cheddar"] = 2
    elif omelet_name == "western":
        ingredients["jack_cheese"] = 2
        ingredients["ham"] = 1
        ingredients["pepper"] = 1
        ingredients["onion"] = 1
    elif omelet_name == "greek":
        ingredients["feta_cheese"] = 2
        ingredients["spinach"] = 2
    else:
        print("That's not on the menu, sorry!")
        return None
    return ingredients
```

做煎蛋卷所需的第二个函数叫做 `make_food`，它需要两个参数。第一个参数是包含所需成分的列表，这些成分完全从 `get_omelet_ingredients` 函数得来。第二个参数是食物的名称，它应当是一种煎蛋卷的类型：

```
def make_food(ingredients_needed, food_name):
    """make_food(ingredients_needed, food_name)
    Takes the ingredients from ingredients_needed and makes food_name"""
    for ingredient in ingredients_needed.keys():
        print("Adding %d of %s to make a %s" %
              (ingredients_needed[ingredient], ingredient, food_name))
    print("Made %s" % food_name)
    return food_name
```

此时，所有部分都已就绪，可以使用 `make_omelet` 函数。它需要调用函数 `get_omelet_ingredients` 和 `make_food` 来完成工作。每个函数都提供了制作一个煎蛋卷的部分过程。`get_omelet_ingredients` 函数为特有的煎蛋卷类型提供了详细而精确的说明，而通过 `make_food` 函数提供的信息，可以知道在将合适数量的一些成分混合之后得到哪些结果(为了演示，这里采用了一种简单的方法)。

### 试一试

### 调用已完成的函数

现在已经有了能使 `make_omelet` 工作的所有函数，用 `python -i` 或者 `Run with Interpreter` 命令调用 `ch5.py` 文件，之后在 `Shell` 中尝试如下代码：

```
>>> omelet_type = make_omelet("cheese")
Adding 2 of eggs to make a cheese
Adding 2 of cheddar to make a cheese
Adding 1 of milk to make a cheese
Made cheese
>>> print omelet_type
cheese
>>> omelet_type = make_omelet({"eggs":2, "jack_cheese":2, "milk":1,
"mushrooms":2})
omelet_type is a dictionary with ingredients
Adding 2 of jack_cheese to make a omelet
Adding 2 of mushrooms to make a omelet
Adding 2 of eggs to make a omelet
Adding 1 of milk to make a omelet
Made omelet
>>> print omelet_type
Omelet
```

### 示例说明

所有函数已经就绪，可以被逐个调用，所以只需指定希望制作的煎蛋卷的名称，然后就可以使用 `make_omelet` 了。

## 5.2.9 函数嵌套函数

在工作中不太可能对煎蛋卷制作进行模拟，但对现实世界的情形进行部分模拟是有可能的，本小节将介绍如何改进已有的解决方案。

也许一个特定函数的工作太多，不便在一个地方定义，您希望将它分解成小的不同的片段。为此，可以将函数放在另外的函数内部，并且从那个函数中调用该函数。这样复杂函数变得更有意义。例如，`get_omelet_ingredients` 可以完全包含在 `make_omelet` 函数内部，并且不能用于程序的其余部分。

限制该函数的可见性是有意义的，因为该函数只能用于制作煎蛋卷。如果编写一个程序，其中也包含制作其他食物的指令，煎蛋卷的成分将不能用于制作其他类型的食物，即使像炒蛋或者梳芙厘这样与煎蛋卷类似的食物。每种新食物都需要自己的功能相同的函数。然而，`make_food` 函数仍然有意义，可用于任意类型的食物。

在另外一个函数中定义一个函数就像在顶层定义它一样。唯一的不同之处是它与包含它的函数中的其他代码在相同的级别缩进。这种情况下，所有代码看起来很类似：

```
def make_omelet(omelet_type):
    """This will make an omelet. You can either pass in a dictionary
    that contains all of the ingredients for your omelet, or provide
    a string to select a type of omelet this function already knows
    about"""
    def get_omelet_ingredients(omelet_name):
        """This contains a dictionary of omelet names that can be produced,
        and their ingredients"""
        ingredients = {"eggs":2, "milk":1}
        if omelet_name == "cheese":
            ingredients["cheddar"] = 2
        elif omelet_name == "western":
            ingredients["jack_cheese"] = 2

        ingredients["ham"] = 1
            ingredients["pepper"] = 1
            ingredients["onion"] = 1
        elif omelet_name == "greek":
            ingredients["feta_cheese"] = 2
        else:
            print("That's not on the menu, sorry!")
            return None
        return ingredients
    if type(omelet_type) == type({}):
        print("omelet_type is a dictionary with ingredients")
        return make_food(omelet_type, "omelet")
    elif type(omelet_type) == type(""):
        omelet_ingredients = get_omelet_ingredients(omelet_type)
        return make_food(omelet_ingredients, omelet_type)
    else:
        print("I don't think I can make this kind of omelet: %s" % omelet_type)
```

使用函数之前定义函数非常重要。如果试图在定义一个函数之前就调用它，那么在调用时 Python 不知道函数的存在，因此不能调用它！当然，这将导致一个错误并引起一个异常。因此，在文件的开始处定义函数，这样一直到最后都可以使用它们。

### 5.2.10 用自己的词语标记错误

如果需要指出发生了一个特定的错误，那么可能希望使用已经见过的一个错误指出正在调用的函数哪里出错了。

与 `try:` 和 `except:` 关键字对应的是 `raise...` 命令。使用 `raise...` 命令的好时机是，在编写了一个接受多个参数的函数，而其中一个参数类型错误时，非常适合使用 `raise...` 命令。

可以检查传递进来的参数，并使用 `raise...` 指出提供了错误类型的参数。当使用 `raise...` 时，还提供了一条 `except...:` 语句可以捕获并显示的错误信息，即关于该错误的解释。

下面的代码修改了 `make_omelet` 函数的结尾部分，将一个打印出的错误(适合运行程序的人阅读)替换为一个 `raise...` 语句，这样既可以通过函数处理错误，也可以将该错误打印出来。

```
if type(omelet_type) == type({}):
    print("omelet_type is a dictionary with ingredients")
    return make_food(omelet_type, "omelet")
elif type(omelet_type) == type(""):
    omelet_ingredients = get_omelet_ingredients(omelet_type)
    return make_food(omelet_ingredients, omelet_type)
else:
    raise TypeError("No such omelet type: %s" % omelet_type)
```

完成该修改后，`make_omelet` 在遇到问题时，可以显示关于该类错误的精确信息，同时仍旧为用户提供信息。

## 5.3 函数的层次

现在已经了解了函数是什么以及它们是如何工作的，考虑它们的调用方式以及 Python 如何记录调用层次非常有用。

当程序调用一个函数时，或者一个函数调用一个函数时，Python 在其内部创建了一个叫做栈的列表，有时也叫做调用栈。当调用一个函数时，Python 将停止片刻，记住程序调用函数时所处的位置，之后将该信息贮藏到它的内部列表。之后进入函数并且执行它。例如，如下代码阐明了 Python 如何记录它是如何进入以及离开函数的：

```
[{'top_level': 'line 1'}, {'make_omelet': 'line 64'}, {'make food': 'line 120'}]
```

在最上层，Python 从第一行开始记录。之后，当函数 `make_omelet` 在第 64 行被调用时，也对其进行了记录。再后来，`make_food` 被 `make_omelet` 调用。当 `make_food` 函数结束时，Python 确定它在第 64 行，于是返回到第 64 行并继续执行。这个示例中的行数是虚构的，

但是您可以了解其中的意思。

这个列表叫做栈，形象地表示出了进入函数的方式。可以想象直到退出时，一个函数位于栈的顶部，当去掉它时，栈的长度缩减了 1。

## 如何解读深层的错误

当程序中有一个错误发生时，如果引起一个未捕获的错误，您将看到一个比之前见过的更加复杂的错误。例如，假设传递了一个包含列表而不是一个数值的字典。这将导致一个如下所示的错误：

```
>>> make_omelet({"a":1, "b":2, "j":["c", "d", "e"]})
omelet_type is a dictionary with ingredients
Adding 1 of a to make a omelet
Adding 2 of b to make a omelet
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "ch5.py", line 96, in make_omelet
    return make_food(omelet_type, "omelet")
  File "ch5.py", line 45, in make_food
    Print("Adding %d of %s to make a %s" % (ingredients_needed[ingredient],
ingredient, food_name))
TypeError: int argument required
```

当从文件进入一个函数后，Python 将指出您在栈中的位置(这意味着错误发生时有多少层以及栈中的每层在程序的哪一行被调用)，因此可以打开有问题的文件来确定发生了什么。

当调用更多的函数或者使用函数调用其他函数，创建了深层的栈(可以将它想象为长一些的列表)时，就获得了使用栈跟踪(这是引发一个错误或者引起一个异常时，Python 对输出使用的通用名称)的经验。前面的栈跟踪深度为 3，可以看到在第 45 行，当调用 `make_food` 时，参数类型有问题。现在可以回去修复它。

如果认为这个问题会经常发生，可以将 `make_food` 调用括在一个 `try...:代码块` 中，这样 `TypeError` 错误将不会终止程序。然而，如果能在函数中处理掉这个错误，就更好了。

在使用明显错误的类型或者字典成员的情况下，Python 将引起一个 `TypeError`，我们通常没必要执行额外的操作。然而，希望如何处理特定的情形取决于您。

栈跟踪是栈的可读形式，可以检查它们来确定问题所在。栈跟踪显示了问题发生时的所有情况，只要引发异常，Python 就会生成栈跟踪。

## 5.4 本章小结

本章介绍了函数。函数将 Python 中的若干语句聚集在同一个名称下，以在需要的时候调用它。在定义函数时，可以指定该函数带有参数，这样在调用它时，函数就可以对指定的参数值进行操作。

函数的参数名称与该函数一起定义，它们在函数名称后面用括号括起来。不需要参数

时，括号仍然出现，只是为空而已。

当函数被调用时，每个函数创建了自己的作用域，在其中可以访问全局范围内的名称，还可以访问函数体内创建和赋值的名称。如果一个全局名称在一个特定函数内被赋值，当用全局名称引用它时，它的值不发生变化，只是在该函数体内发生变化。

如果一个函数在另外一个函数内定义，它可以访问定义它的函数内的所有名称，还有全局范围的所有名称。记住，这种可见性取决于函数在哪里定义，而不取决于它在哪里被调用。

函数可以在其他函数内被调用。这样可以使程序易于理解。函数将通用的任务用一个简短的名称表示，减少了重复输入。

定义了参数的函数通过值来调用，提供的每个值都要赋给函数参数列表中的名称。传递给函数的第一个参数赋值给第一个名称，第二个参数赋给第二个名称，依此类推。给函数传递参数时，每一个参数要么是必需的，要么是可选的。定义函数时，可选参数必须位于必需参数后面，可以赋予它们默认值。

可以使用 `raise...` 特征通知发生了错误，这些错误可以被 `except...` 接收和处理。这就可以从函数提供反馈，它不仅提供错误类型，还提供了对错误进行描述的字符串，从而使该错误可以被处理。

本章还介绍了栈。当 `raise...` 或者程序中的其他错误引发了一个错误条件时，错误的位置不仅通过发生错误的函数名称来描述，还通过涉及到的函数名称以及它们的调用位置来描述。如果一个函数足够有用，可在不同的地方使用它，且它只在其中一个地方出问题，可以通过栈跟踪缩小问题源。

#### 本章要点：

- 通过 `Python -i` (或者 `Run with Interpreter`) 运行 Python 程序，可以创建一些较长的程序，而不用直接向 Shell 中输入代码。
- 通过将代码片段另存为函数，使它们可在当前或者以后的程序中重复使用，可以节省时间。
- 文档字符串以三引号 (""" ) 开头，它可以定义函数的用途，为自己和将来的程序员留下注释。
- 可以在函数中用 `__doc__` 显示文档字符串。
- 通过 `dir()` 可以看到一个对象的所有属性。
- 可以通过 `#` 向代码中添加注释。Python 忽略同一行中该符号后面的内容。注释允许您留下记录，若干月后需要再次查看代码或者其他编程人员需要阅读代码时，可以更快地理解当初编写的代码。
- `type()` 函数可以显示一个对象的类型。

## 5.5 习题

1. 编写一个名为 `do_plus` 的函数，它接收两个参数，并且用 “+” 将它们相加。
2. 增加类型检查，以确认参数的类型要么是整型，要么是字符串。如果参数不正确，

抛出一个 `TypeError`。

3. 这个题目的工作量较大，请随意将它分为几部分。第 4 章中，编写一个循环制作煎蛋卷。它做了从检查成分到把它们从冰箱中取出并做成一个煎蛋卷的全部工作。以该循环为原型，更改 `make_omelet` 函数，编成一个名为 `make_omelet_q3` 的函数。它应该从下述方面对 `make_omelet` 进行修改，使得它更加贴切地描述一个真实的厨房：
  - a. 将冰箱作为第一个参数传递给新的 `make_omelet`。应当检查冰箱的类型，确保它是一个字典。
  - b. 增加一个函数检查冰箱，并且将即将用到的成分从冰箱中减去。将这个函数命名为 `remove_from_fridge`。该函数首先检查冰箱中是否有足够多的成分制作煎蛋卷，检查完毕后，再将这些成分从冰箱中取出以制作煎蛋卷。用错误类型 `LookupError` 作为要抛出的错误的类型。
  - c. 将从冰箱中取出的食物放到一个字典中，由函数 `remove_from_fridge` 函数返回，赋给一个名称，将该名称传递给 `make_food`。毕竟，并不希望将不打算使用的食物从冰箱中取出。
  - d. 制作一个不同的默认煎蛋卷，而不是奶酪煎蛋卷。向函数 `get_omelet_ingredients` 中添加这个煎蛋卷的成分。
4. 更改 `make_omelet`，在订购沙门氏菌煎蛋卷时，使 `get_omelet_ingredients` 函数抛出一个 `TypeError` 错误。尝试订购一个沙门氏菌煎蛋卷，并查看产生的栈跟踪。



# 第 6 章

## 类 与 对 象

到目前为止，已经介绍了编程所需的大多数构建模块。您已经使用了数据，使用名称引用数据(程序员通常将这些名称称作变量)，还在循环和函数中使用过数据。这三个元素的使用是编写程序以及用计算机解决问题的基础。可以使用名称来存储和引用值，并对它们进行操作。循环可以重复处理列表中的每个元素(或每隔一个元素、两个元素等的元素)。最后，函数可将一组代码整合到一个名称下，这样可在任何需要的时刻和位置调用它。

本章将介绍：

- Python 如何将函数和数据整合在一起，并通过一个对象的名称访问它们。
- 如何和为什么使用类与对象，以及它们如何使编程人员易于在多种情形下编写和使用程序。

### 6.1 考虑编程

到目前为止，本书已经介绍了 Python 的基础知识。现在要在 Python 中创建一个对对象的描述，您已有足够的知识获得两个视图。第一个是数据视图，除了顶层或者全局作用域的数据外，可以根据需要使用和清除它们。另一个是函数视图，它们没有固有的数据，而只操作提供给它们的数据。

#### 6.1.1 对象的含义

Python 中的任何一条数据都是对象。每个对象都由 3 部分组成：标识、类型和值。对象的标识代表该对象在内存中的存储位置(因此是不可更改的)，对象的类型表明它可以拥有的数据和值的类型。在对象中，可变类型的值可以更改，不可变类型的值不能更改。

简单些的解释是参考本书中已经介绍的(或者即将介绍)的对象。例如，整型、字符串、列表等都是对象。可以在程序中很方便地使用这些对象，但是将关系紧密的对象整合在一起岂不更有意义？这也是类的由来。类允许定义一组对象，并将它们封装到一个方便的空间中。

#### 6.1.2 已经了解的对象

下面即将介绍的方法使您可以考虑包含数据和函数的完整对象。字符串是前面已经接

触过的对象。一个字符串不仅包含文本，也有关联的方法，这使得字符串不仅代表文本，还提供了一些功能，例如将整个字符串变成大写或者小写。下面简单回顾已经学到的知识。一个字符串主要是指输入的文本：

```
>>> omelet_type = "Cheese"
```

除了经常处理的数据，即文本“Cheese”，字符串这种对象还拥有方法，方法经常被叫做行为。每一个字符串都有一些方法，如返回小写的字符串的 `lower` 方法，以及返回一个完全大写的字符串的 `upper` 方法：

```
>>> omelet_type.lower()
'cheese'
>>> omelet_type.upper()
'CHEESE'
```

其他可用的方法还有内置在元组、列表和字典对象中的方法，例如字典的 `keys` 方法，在前面已经使用过该方法：

```
>>> fridge = {"cheese":1, "tomato":2, "milk":4}
>>> for x in fridge.keys():
    print(x)
['tomato', 'cheese', 'milk']
```

使用 `dir` 函数可以列举出一个对象的所有属性和方法：

```
dir(fridge)
['__class__', '__contains__', '__delattr__', '__delitem__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__',
'__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__',
'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem',
'setdefault', 'update', 'values']
```

`dir` 函数可以显示字符串或者 Python 中其他任何对象的所有数据方法，简言之，它可以显示这些对象所包含的每个名称。Dir 函数能够以字母表的顺序列出正在检查的对象中所有可用的名称，它将以下划线开头的名称放在结果的最开始。按照约定，以下划线开头的名称是对象的私有属性，它们是不可见的。即，不能直接使用它们，但是 Python 将决定权留给了开发人员，所以您可以交互地查看这些属性以学习它们：

```
>>> type(omelet_type.__len__)
<class 'method-wrapper'>
```

这很有趣。可以调用该方法，并观察它输出的结果：

```
>>> omelet_type.__len__()  
6
```

这个方法返回与内置函数 `len` 一样的结果。当一个函数内置到一个对象中时，它叫做这个对象的“方法”。

实际上，函数 `len` 就是利用方法 `__len__` 工作的：函数 `len` 向这个内置方法询问对象的长度。对象的设计者可定义长度的计算方式，使得对于定义了 `__len__` 方法的任何对象，内置函数 `len` 都能够正确运行。

其余的以下划线开头的名称也都具有特殊含义。可以在 `Python Shell` 中探究它们。`Python Shell` 通过显示试图调用的对象内的名称，帮助研究一个字符串对象的方法，但它不显示以下划线开头的内部名称。您可以使用 `dir` 函数实现该功能。

### 6.1.3 展望：如何使用对象

我们希望能够以自然的方式使用对象。例如，一旦定义了 `Omelet` 类之后，它便可以产生对象，在查看源代码时，这些对象以自然的方式工作。稍后就会尝试编写实现该操作的代码(下一节中将看到如何操作)：

```
>>> o1 = Omelet()  
>>> o1.show_kind()  
'cheese'
```

我们希望将冰箱当做对象使用，而不仅仅把它看做一个字典。如果使用它就像使用一个真正的冰箱一样，就太合适不过了，我们可以向其中增加食物、从中取出食物、检查食物，以及一次增加或者取出多种食物等。

换言之，创建一个对现实世界的某物进行模拟的对象时，应该使熟悉该现实事物的人们能够比较轻松地认知和理解它们。

## 6.2 定义类

当思考包含几百行 `Python` 代码的小程序如何运行时，经常可以发现程序将数据组织成组的形式——当访问某个数据时，将影响与该数据一起协作的数据。经常会碰到有相互依赖关系的完整数据列表，如列表 1 的第一个元素与列表 2 和列表 3 中的第一个元素匹配，依此类推。有时，必须通过创造性地将这些列表组合起来才能解决这个问题。`Python` 运用了创建用作占位符的整个类的概念，类起了占位符的作用。当一个类被调用时，它创建了绑定到一个名称的对象。

### 6.2.1 如何创建对象

将对象绑定到一个名称后，使用该名称可以访问已定义的数据和函数。编写一个类的代码之前，首先要声明该类。通过 `class` 关键字声明类。

## 试一试

## 定义一个类

定义一个类比较简单，主要是使用关键字 `class`，并在后面紧跟一个名称来完成。定义类的方式与定义函数的方式类似，只不过不在类定义之后放置包含项目的元组(在类的定义之后放置包含项目的元组将创建一个要继承的类，第 10 章将介绍这方面的内容)。

```
class Fridge:
    """This class implements a fridge where ingredients can be
    added and removed individually, or in groups."""
```

## 示例说明

从类的定义开始，缩进于其中的所有代码在该类创建的对象中都可用。在第 5 章介绍函数时已经见过这一点，类似的规则同样适用于类。在类中可以自由选择内置文档字符串，就像在函数中一样。它们的表现方式相同，并且非常有用，提供了了解类信息的简单方法。

如前面的例子所示，应当试着创建 `Fridge` 类。注意此处使用了大写的“F”。类的名称以大写字母开头是 Python 程序员的惯例。当类的名称有多个单词时，另一种惯例是将这些词连在一起，但是每个词以大写字母开头以方便阅读。例如，模拟一个冰箱和制冷器的类可被命名为 `FridgeAndFreezer`。

## 试一试

## 由类创建对象

尝试将 `Fridge` 类输入到文件 `ch6.py` 中(或者类似的文件中)，之后用 `python -i` 或者命令 `Run with Interpreter` 调用该文件，如第 5 章所示。

通过使用一对圆括号调用 `Fridge` 类，可以创建一个 `Fridge` 对象：

```
>>> f=Fridge()
```

## 示例说明

此时还没有定义任何复杂的类。`Fridge` 类基本是空的，它将作为一个起点。然而，即使它是空的，也应当注意到已经创建了一个可用的空类。它几乎不进行任何操作，但在某些情形下，确实不需要进行任何操作。例如，可以将这个几乎为空的对象当作一个特殊类型的字典处理。可以通过在测试时交互地向类中添加名称来实现该操作。这有助于了解它的工作方式：

```
>>> f.items = {}
>>> f.items["mystery meat"] = 1
```

除此之外，在第 10 章将会介绍，异常实际上也是类，有时候只需要一个空类就可以创建一个有效的异常。当有一个简单的、像该类一样未定义的类时，应仅使用这种类型的直接访问。当有一个已做了许多定义的类时，访问该类作用域内的名称会妨碍类的编写方

式，因此将引起许多麻烦。

开始编写一个类的最佳方式是首先确定它的功能。此处，基于 Python 的电冰箱模型 `Fridge` 是第一个类，它比较基础。考虑这个类时，要专注于需要一个特定的 `Fridge` 对象完成哪些功能。您希望该对象有足够多的操作可以制造食物，但是不希望担心现实生活中冰箱的一些问题，如温度、冷藏室、除霜、电等，这些具体的问题没有包括在这个简单的示例中，因为它们都是不必要的细节，只会使目标变得复杂。现在，向 `Fridge` 类中增加文档字符串，定义即将为它创建的各种行为。

首先，您希望 `Fridge` 类有一个存储食物的方法，并打算用两种方式实现该功能：第一种方式是一次添加一种类型的单个项目，并向类中增加一个完整的字典，这种方式比较易于初始化；第二种方式是模拟冰箱被填充的情形，例如购物归来的时候。

其次，您希望有一个从 `Fridge` 类取出食物的方法。在取出食物和放入食物时做相同的事情：从冰箱中拿出一个或者许多食物。

还需要向这个对象中写入若干代码，使选择的这个模型成为一个 `Fridge`：一个是能确定某种食物是否在 `Fridge` 中存在的函数，另外一个检查一个包含食物成分的完整字典的函数。现在可以开始准备烹饪了。

为了建模的目的，在使用 `Fridge` 存储成分并在希望烹饪时从冰箱中取出它们时，这即是需要的全部方法。换言之，这将作为该特定情形的模型运行，并能够应付每种可能的情况。

对象提供给外界使用的方法叫做它的接口，因为这些方法是对对象之外的程序使用该对象的途径。对象通过提供接口变得实用。

接口是使对象可用的任何方法和名称。在 Python 中，这通常意味着没有以一个或多个下划线开头的方法和名称都是接口；当然，在类的文档字符串中明确指出哪些方法可用，以及如何使用它们，以与那些不希望被调用的函数区别，这是一个很好的习惯。

```
class Fridge:
    """This class implements a fridge where ingredients can be
    added and removed individually, or in groups.
    The fridge will retain a count of every ingredient added or removed,
    and will raise an error if a sufficient quantity of an ingredient
    isn't present.
    Methods:
    has(food_name [, quantity]) - checks if the string food_name is in the
    fridge. Quantity will be set to 1 if you don't specify a number.
    has_various(foods) - checks if enough of every food in the dictionary is in
    the fridge
    add_one(food_name) - adds a single food_name to the fridge
    add_many(food_dict) - adds a whole dictionary filled with food
    get_one(food_name) - takes out a single food_name from the fridge
    get_many(food_dict) - takes out a whole dictionary worth of food.
    get_ingredients(food) - If passed an object that has the __ingredients__
    method, get_many will invoke this to get the list of ingredients.
    """

    def __init__(self, items={}):
        """Optionally pass in an initial dictionary of items"""
```

```

    if type(items) != type({}):
        raise TypeError("Fridge requires a dictionary but was given %s" %
            type(items))
    self.items = items
    return

```

花几分钟看上面代码中的 `__init__` 和 `self` 部分。这是类的两个非常重要的特征。当 Python 创建对象时，`__init__` 方法传递给对象第一个参数。而 `self` 实际上是代表该实例本身的变量。

另外，当决定编写一个类时，为希望使用的方法做注释是一个好习惯。实际上，它是考虑类的完整性时，类要完成的功能的概述，这使得在编写程序的同时就可以测试程序(更详细的信息可参考第 12 章)。

当编写接口方法时，将会注意到，大多数情况下，简单的方法共享许多特征，例如“取出一种”、“取出两种”或者“取出多种”等，但是为了使它们调用起来比较简单，希望保留各种形式。这看起来好像要为其中每个函数重复大量源代码。然而，不用重复输入接口方法的共同部分，可以编写仅由每个方法内部使用的私有方法而节省输入。

这些私有方法可以执行部分或者全部接口方法的共同操作。当私有方法很复杂，或者包含用户在使用方法时不需要了解细节时，需要进行这样的操作。这样可以在类被调用时避免困惑，同时使它更易于编写。最好的情况下，这是个明显的双赢局面。

在 `Fridge` 类，以及即将编写的许多类中，通常有一个方法可以操作一组数据，有另外一个方法只处理单个元素。不论何时遇到这种情形，都可以使处理单个项目的方法简单地调用可处理任意多个项目的函数而减少工作量。实际上，有时将该方法当作私有方法，或者不作为接口的一部分很有用。这样不管它是否被使用，它的更改都不会影响类的使用，因为它所做的修改在对象外不可见，只在内部可见。

对于 `Fridge` 类，创建带有两个参数的内部方法 `__add_multi` 可将工作量减到最小，方法的两个参数分别是项的名称和项的数量。该方法可将项添加到每个对象的 `items` 字典中。

## 试一试

## 编写内部方法

当向本章的文件中添加新方法时，确保这些方法正确地缩进于 `Fridge` 类下，而不是单独出现在顶层。为了明确说明，在这里给出类的声明：

```

class Fridge:
    # the docstring and intervening portions of the class would be here, and
    # __add_multi should go afterwards.
    def __add_multi(self, food_name, quantity):
        """
        __add_multi(food_name, quantity) - adds more than one of a
        food item. Returns the number of items added

        This should only be used internally, after the type checking has been
        done
        """

```

```

if (not food_name in self.items):
    self.items[food_name] = 0

self.items[food_name] = self.items[food_name] + quantity

```

### 示例说明

现在已有方法可将任意数量的食物添加到 `Fridge` 对象中。然而，这是一个内部方法，不能判定当前传入的类型(`food_name` 或 `quantity`)是否有效。应当用接口函数做这个检测，因为作为一个认真的程序员，总是需要确定给私有方法传递了正确的值。在允许的每个地方做检查是一个好想法，但在当前这个例子中，不打算在此处检查，因为只会以非常简单的方式使用 `__add__multi` 方法。

现在 `Fridge` 类已经拥有了很好用的方法 `__add__multi`，之后 `add_one` 和 `add_many` 方法都可使用它，而不必将类似的函数写两遍。这就减少了工作量。

### 试一试

### 编写接口方法

为了更快捷，现在可以不输入文档字符串。此处的方法使您在遇到问题时，可以更好地理解代码的实际操作。

如前所示，这些方法需要缩进在 `Fridge` 类的定义中。看上去在每行的初始位置开始的任何代码实际上是前一行的延续，应当输入到同一行上：

```

def add_one(self, food_name):
    """
    add_one(food_name) - adds a single food_name to the fridge
    returns True
    Raises a TypeError if food_name is not a string.
    """
    if type(food_name) != type(""):
        raise TypeError, "add_one requires a string, given a %s" % type(
            food_name)
    else:
        self.__add_multi(food_name, 1)

    return True

def add_many(self, food_dict):
    """
    add_many(food_dict) - adds a whole dictionary filled with food as
    keys and
    quantities as values.
    returns a dictionary with the removed food.
    raises a TypeError if food_dict is not a dictionary
    returns False if there is not enough food in the fridge.
    """

```

```

    if type(food_dict) != type({}):
        raise TypeError("add_many requires a dictionary, got a %s" %
            food_dict)

    for item in food_dict.keys():
        self.__add_multi(item, food_dict[item])
    return

```

### 示例说明

`add_one` 和 `add_many` 的目的类似, 并且每个方法都有可以确保它们被正确使用的代码。同时, 它们都使用 `__add_multi` 来完成主要的工作。现在, 如果 `__add_multi` 的工作方式发生改变, 开发人员可以节省时间, 因为它将自动改变使用它的两个方法的行为方式。

现在已经编写了足够的代码, 可将食物放入 `Fridge` 对象中, 但是没有方法可以将放入冰箱的食物拿出来。可以直接访问 `object.items` 字典, 但除了测试的时候, 这从不是一个好主意。当然, 现在就是在测试, 为何不这么做呢?

```

>>> f = Fridge({"eggs":6, "milk":4, "cheese":3})
>>> f.items
{'cheese': 3, 'eggs': 6, 'milk': 4}
>>> f.add_one("grape")
True
>>> f.items
{'cheese': 3, 'eggs': 6, 'grape': 1, 'milk': 4}
>>> f.add_many({"mushroom":5, "tomato":3})
>>> f.items
{'tomato': 3, 'cheese': 3, 'grape': 1, 'mushroom': 5, 'eggs': 6, 'milk': 4}
>>>

```

目前为止输入的代码都能正常工作! 这是较简单的部分。接下来需要增加可以判断冰箱中是否存在某物的方法。

编写代码证实某物是否在冰箱中存在很重要, 因为它可用于取出食物的方法中, 如 `get_one`、`get_many` 和 `get_ingredients`, 从而使这些方法可以检查冰箱中是否有足够多所需的食物。这正是 `has` 与 `has_various` 方法的用途:

```

def has(self, food_name, quantity=1):
    """
    has(food_name, [quantity]) - checks if the string food_name is in the
    fridge. Quantity defaults to 1
    Returns True if there is enough, False otherwise.
    """

    return self.has_various({food_name:quantity})

def has_various(self, foods):
    """
    has_various(foods) determines if the dictionary food_name

```

```

        has enough of every element to satisfy a request.
    returns True if there's enough, False if there's not or if an element
does
    not exist.
    """

    try:
        for food in foods.keys():
            if self.items[food] < foods[food]:
                return False
        return True
    except KeyError:
        return False

```

在 `has` 和 `has_various` 方法完成后，可在测试中使用 `Fridge` 对象，当阅读代码时，它总是有意义的。

### 试一试

### 使用更多的方法

现在可以使用 `python -i` 或者 `Run with Interpreter` 命令调用 `ch6.py` 文件，这样可以使用添加到 `Fridge` 类的任何代码。如果出现错误而不是 `>>>` 提示符，注意抛出的异常，并试着修复缩进问题、拼写错误或者其他基本的错误。

`Fridge` 类可按下述方法使用：

```

>>> f = Fridge({"eggs":6, "milk":4, "cheese":3})
>>> if f.has("cheese", 2):
...     print("It's time to make an omelet!")
...
It's time to make an omelet!

```

### 示例说明

现在已经定义了新的方法，`f` 对象可以使用它们。当用鸡蛋、牛奶以及奶酪重新创建 `f` 对象时，就从新的 `Fridge` 类创建了对象，因此它拥有新添加的可用方法。

最后，我们应当讨论从冰箱中取食物的方法了。与向冰箱中添加食物的方法类似，由一个核心方法完成主要工作，所有接口方法都依赖于这个方法：

```

def __get_multi(self, food_name, quantity):
    """
    _get_multi(food_name, quantity) - removes more than one of a
    food item. Returns the number of items removed
    returns False if there isn't enough food_name in the fridge.
    This should only be used internally, after the type checking has been
    done
    """

```

```

try:
    if (self.items[food_name] is None):
        return False;

    if (quantity > self.items[food_name]):
        return False;
    self.items[food_name] = self.items[food_name] - quantity
except KeyError:
    return False
return quantity

```

当定义了上面的方法后，可以创建 `Fridge` 类的文档字符串指定的其他方法。其中每个方法都使用了 `__get_multi`，因此都可以用最少的额外代码从冰箱中取出食物：

```

def get_one(self, food_name):
    """
    get_one(food_name) - takes out a single food_name from the fridge
    returns a dictionary with the food:1 as a result, or False if there
    wasn't
    enough in the fridge.
    """
    if type(food_name) != type(""):
        raise TypeError("get_one requires a string, given a %s" %
            type(food_name))
    else:
        result = self.__get_multi(food_name, 1)
        return result
def get_many(self, food_dict):
    """
    get_many(food_dict) - takes out a whole dictionary worth of food.
    returns a dictionary with all of the ingredients
    returns False if there are not enough ingredients or if a dictionary
    isn't provided.
    """

    if self.has_various(food_dict):
        foods_removed = {}
        for item in food_dict.keys():
            foods_removed[item] = self.__get_multi(item, food_dict[item])
        return foods_removed

def get_ingredients(self, food):
    """
    get_ingredients(food) - If passed an object that has the __ingredients__
    method, get_many will invoke this to get the list of ingredients.
    """
    try:
        ingredients = self.get_many(food.__ingredients__)
    except AttributeError:
        return False

```

```

if ingredients != False:
    return ingredients

```

现在已经编写了一个完全可用的冰箱类。记住，可以将这些内容沿不同的方向引申。不只可以使用 `Fridge` 类制作煎蛋卷，也可以将它用于其他项目，对产品流程进行建模，例如，熟食品店有 10 个冰箱，每个冰箱中都装着不同的食品。

当有机会为已经编写的类(或者使用过的类)赋予新的用途时，可利用这个机会，在不影响类的已有功能的基础上向类中添加一些功能以支持新的需求。

例如，在需要若干个冰箱的应用程序中也许需要每个 `Fridge` 类有一些额外的属性，例如名称(像“奶制品冰箱”)、在商店中的位置、最佳温度设置和尺寸等。可以将这些属性添加到类中，并添加读取和设置这些值的方法，同时仍保持它对于本书中的煎蛋卷示例完全可用。接口的好处在此时就体现出来了。只要 `Fridge` 类中的接口没有改变，或者只要它们的运行方式保持不变，您就可以做任何修改。保持接口行为不变的能力叫做类的稳定性。

### 6.2.2 对象和它们的作用域

如在第 5 章所见，函数为它们使用的名称创建了自己的空间，也就是作用域。当函数被调用时，声明了名称并赋予值之后，只要函数还在使用，任何对该名称做出的修改会持续下去。然而，在函数结束运行，并再次被调用时，之前调用过程中所做的工作都丢失了，该函数必须重新开始执行。

对象中的值可被存储，并在对象内关联到 `self`(这里的 `self` 是指对象本身的名称，它与对象外指向对象的名称，例如 `f` 是一样的)。只要对象被一个仍在使用的名称引用，它包含的所有值也可用。如果一个对象在函数中被创建，并且没有被该函数返回，即该对象没有被一个具有更长生命周期的名称引用，那么与函数中的其他数据一样，它只在调用它的函数中可用。

如果多个对象先后被创建，它们就可被一起使用。例如，现已经在程序中实现了一个能够工作的 `Fridge` 的所有特性，还需要 `Omelet` 对象与其一起协作。

#### 试一试

#### 创建另外一个类

我们已经创建了一个类 `fridge`。用同样的形式，可以创建一个可用的 `Omelet` 类：

```

class Omelet:
    """This class creates an omelet object. An omelet can be in one of
    two states: ingredients, or cooked.
    An omelet object has the following interfaces:
    get_kind() - returns a string with the type of omelet
    set_kind(kind) - sets the omelet to be the type named
    set_new_kind(kind, ingredients) - lets you create an omelet
    mix() - gets called after all the ingredients are gathered from the fridge
    cook() - cooks the omelet
    """

```

```
def __init__(self, kind="cheese"):
    """ __init__(self, kind="cheese")
    This initializes the Omelet class to default to a cheese omelet.
    Other methods
    """
    self.set_kind(kind)
    return
```

### 示例说明

现在有一个类，它的目的很明确。在第5章的函数中，已经见过大多数这样的行为，但是现在有了一个结构，可以在其中整合所有这些行为。

Omelet 类有接口方法，使得它可与 Fridge 对象协作，并且与第5章一样，它仍具备创建定制的煎蛋卷的能力。

下面所有代码都必须在 Omelet 类定义下缩进一个级别：

```
def __ingredients__(self):
    """Internal method to be called on by a fridge or other objects
    that need to act on ingredients.
    """
    return self.needed_ingredients

def get_kind(self):
    return self.kind

def set_kind(self, kind):
    possible_ingredients = self.__known_kinds(kind)
    if possible_ingredients == False:
        return False
    else:
        self.kind = kind
        self.needed_ingredients = possible_ingredients

def set_new_kind(self, name, ingredients):
    self.kind = name
    self.needed_ingredients = ingredients
    return

def __known_kinds(self, kind):
    if kind == "cheese":
        return {"eggs":2, "milk":1, "cheese":1}
    elif kind == "mushroom":
        return {"eggs":2, "milk":1, "cheese":1, "mushroom":2}
    elif kind == "onion":
        return {"eggs":2, "milk":1, "cheese":1, "onion":1}
    else:
        return False

def get_ingredients(self, fridge):
```

```

        self.from_fridge = fridge.get_ingredients(self)

    def mix(self):
        for ingredient in self.from_fridge.keys():
            print("Mixing %d %s for the %s omelet" %
                  self.from_fridge[ingredient], ingredient, self.kind)
        self.mixed = True

    def make(self):
        if self.mixed == True:
            print("Cooking the %s omelet!" % self.kind)
            self.cooked = True

```

现在有一个 `Omelet` 类可以创建 `Omelet` 对象。`Omelet` 类与第 4 章和第 5 章中制作煎蛋卷的过程有相同的特性，但更简单易用，因为所有特性都整合在了一起，并且 `Omelet` 的外在表现集中到几个简单的接口。

现在有两个类，在用 `python -i` 或者 `Run with Interpreter` 命令加载它们后，可以制作一个煎蛋卷：

```

>>> o = Omelet("cheese")
>>> f = Fridge({"cheese":5, "milk":4, "eggs":12})
>>> o.get_ingredients(f)
>>> o.mix()
Mixing 1 cheese for the cheese omelet
Mixing 2 eggs for the cheese omelet
Mixing 1 milk for the cheese omelet
>>> o.make()
Cooking the cheese omelet!

```

这并不比在第 5 章中制作一个煎蛋卷更难或者更简单。然而，当同时要处理多件事情时，使用对象的好处变得很明显。例如，需要同时制作多个煎蛋卷：

```

>>> f = Fridge({"cheese":5, "milk":4, "eggs":12, "mushroom":6, "onion":6})
>>> o = Omelet("cheese")
>>> m = Omelet("mushroom")
>>> c = Omelet("onion")
>>> o.get_ingredients(f)
>>> o.mix()
Mixing 1 cheese for the cheese omelet
Mixing 2 eggs for the cheese omelet
Mixing 1 milk for the cheese omelet
>>> m.get_ingredients(f)
>>> m.mix()
Mixing 1 cheese for the mushroom omelet
Mixing 2 eggs for the mushroom omelet
Mixing 1 milk for the mushroom omelet
Mixing 2 mushroom for the mushroom omelet
>>> c.get_ingredients(f)

```

```
>>> c.mix()
Mixing 1 cheese for the onion omelet
Mixing 2 eggs for the onion omelet
Mixing 1 milk for the onion omelet
Mixing 1 onion for the onion omelet
>>> o.make()
Cooking the cheese omelet!
>>> m.make()
Cooking the mushroom omelet!
>>> c.make()
Cooking the onion omelet!
```

花几分钟时间将上面的代码与第 5 章中功能相同的函数进行比较，您将意识到为什么要用这种方式编写程序，以及这样的编程方式(叫做面向对象编程)为什么会用于编制大型系统。

只要冰箱里有需要的材料，制作不同类型的煎蛋卷变得异常容易：仅需调用该类创建一个新对象，之后调用每个 Omelet 对象的 3 个方法。当然，可以将这三个方法减少为一个。这个问题留作读者练习。

## 6.3 本章小结

本章介绍了 Python 中的类和对象，这是面向对象编程的基本概念。

在类中使用的函数叫做方法，每个类都有一个特殊的名称 `self`，当 `self` 被调用时，它包含对象中所有的数据和方法。

使用类的名称并在后面跟着圆括号“()”来创建该类的对象。此时可以给定初始化参数，当然不论是否提供参数，新创建的对象都将调用方法 `__init__`。与普通的函数一样，类中的方法(包括 `__init__`)可以接收参数，包括可选的和默认的参数。

创建一个类的过程包括决定创建哪些方法，以完成期望的功能。类中的方法通常可分为两类：一类是公共接口，应当在对象外部调用，另一类是私有方法，应当只能被对象内部的其他方法调用。接口应当尽量不变，而内部的私有方法可以改变，而不影响类的使用方式。当使用其他人编写的类时，记住这一点尤其重要。Python 认为在对象作用域内，以两个下划线开头的任何名称都是私有属性，因此您也应当遵守这个约定。其他名称一般被认为是公共的。

### 本章要点：

- 为了详细说明类的使用方法，应当为这个类创建文档字符串，即在类定义后的第一行输入一个字符串。文档字符串最好能够提供期望被使用的方法的名称和用途。在文档字符串中包含对整个类的解释也不是一个坏主意。
- 类中定义的所有名称(数据和方法)在每个创建的对象中都不同。当一个对象调用某个方法时，该方法改变那个对象中的数据，而同一类型的其他对象则不受影响。

Python 中内置的字符串对象就是一个示例，它们包含特殊的方法，可以帮助完成普通的文本处理任务。

- 为了使对象易用，通常提供行为类似的多个接口。为了节省工作，这些接口可以找到调用内部方法的一些方式，而内部方法比它们更复杂，接收更多的参数。这有两个好处。第一，调用这些方法的代码更易读，程序员不必记住参数的名称，因为方法的名称本身就提供了程序员所需的信息。第二，只改变内部方法就可以改变所有相关接口的行为。这在修复问题时尤其有用，因为一个修复就可改正所有接口的工作方式。另外，为其他方法提供这种支持的方法本身也可以是公共接口。像这样一个特别有用的方法应当是私有的还是公共的并没有严格的规则，这完全由开发人员决定。
- 编写对象的一个目的是尽量少编写重复代码，而是提供尽量多的特性。创建使用对象的类可以节省大量输入，因为它们通常比数据和函数分开更易于操作，原因在于同一个类中的方法总是可以确定它们使用的其他方法和数据是存在的。可以写一组相互依赖的类，对相互协作的事物建模。第 7 章将介绍如何组织这些相互依赖和合作的类。
- 本章最后介绍了 Python 的代码编辑器 Shell，它可以帮助研究对象，只要输入一个句点，它将显示该对象所有接口的名称。这比输入 `dir` 得到相同信息更简单，因为代码编辑器能够以便捷和易于使用的方式显示信息。

## 6.4 习题

下面每个习题都是建立在前面习题的基础之上。

1. 向 Omelet 类的 `mix` 方法中加入一个可以关闭创建消息的选项，方法是添加一个默认为 `True` 的参数，表示应当打印 “mixing...” 消息。
2. 在 Omelet 类中创建一个方法 `quick_cook`，该方法使用习题 1 中新的 `mix` 方法，它接受 3 个参数：煎蛋卷的类型、需要的数量，以及它们来自于哪个 `Fridge`。`quick_cook` 应当实现需要的功能，而不只是 3 个方法调用，但是它应当使用已有的方法完成这些功能，包括已修改的 `mix` 方法，它关掉了 `mix` 的消息。
3. 试着为 Omelet 类中所有没有文档字符串的方法创建一个文档字符串。确保在每个文档字符串中包含方法的名称，方法需要的参数，方法的功能，成功时返回的值，以及当发生错误时返回的结果(或者抛出什么样的异常)。
4. 创建一个 Omelet 对象，查看已创建的文档字符串。
5. 创建可以被 Omelet 类调用的 `Recipe` 类，它可以得到食物成分。`Recipe` 类应当拥有 Omelet 类包括的煎蛋卷成分列表，还可以包含其他任何喜欢的食物。`Recipe` 类应当有获得食谱的方法 `get(recipe_name)`，以及添加食谱并对它命名的方法 `create(recipe_name, ingredients)`，其中 `ingredients` 是一个字典，与 `Fridge` 类和 Omelet 类中用到的字典格式相同。
6. 更改 Omelet 类中的 `__init__` 方法，它可以接受 `Recipe` 类作为输入。为此，可执行如

下操作：

- a. 创建一个名称 `self.recipe`，这是每个 `Omelet` 对象都有的名称。
  - b. `Omelet` 类的内部方法 `__known_kinds` 存储处方。更改 `__known_kinds`，在期望的煎蛋卷类中调用 `self.recipe.get()` 来使用食谱。
  - c. 更改 `set_new_kind` 方法，将新的食谱添加 `self.recipe` 中，之后调用 `set_kind`，将当前煎蛋卷的类型设置为刚添加到食谱中的类型。
  - d. 修改 `__known_kinds`，使用 `recipe` 的 `get` 方法找出煎蛋卷的成分。
7. 尝试使用所有的新类和新方法，判断您是否已经理解了它们。



# 第 7 章

## 组 织 程 序

第 6 章中,您开始利用 Python 的特性创建不同的类,这些类可以创建完全自包含对象。创建的类和对象是一些工具,用于将数据和函数集中到自包含的空间,这样可将它们一个大实体的一部分。

到目前为止,都是在一个文件中定义类,而且并没有以通常的方式运行程序。相反,它们被交互式地调用,因此可以像从另外的程序中调用一样使用类。然而,如果希望用学到的知识使用编写的类,则需要使定义这些类的文件成为程序。这意味着使所有的类放在文件的开始,而将重要的做决策的代码放在文件尾部。通常需要花费大量时间在文件的尾部寻找希望找到的代码。

还有一点需要注意。类非常有用,但并不是所有的问题都要通过创建类来解决。有时,定义类是大材小用,有时则只需要函数,而并不需要像对象中的数据和方法那样拥有较长的生存期。

Python 提供了一些强大的特性使其更有用,其中之一就是允许创建模块,这些模块为函数和数据创建了一个已命名的作用域,但是比类和对象简单。模块是个工具,它将程序划分为不同的命名片段,而不必使用类来实现这一点。实际上,可在模块中定义类。

作为扩展,可以将这些模块划分到不同的文件中,Python 将这个特性叫做包。包将程序分散到几个文件中,甚至分散到不同的目录中,以对它们进行组织。

到目前为止,本书只介绍了 Python 语言的内部知识,即 Python 的工作原理。尽管 Python 只有一小部分核心特性,但它非常灵活,可在各种模块中扩展。可以使用一个叫做 os 的模块扩展 Python,使其能够利用操作系统的特性。Python 也具备网络特性,这是通过既能提供低级的网络协议(例如套接字),也能提供高级的网络协议(例如 http 和 ft 等)的模块实现的。Python 有许多自带模块。由于模块易于编写,所以有各式各样的由第三方提供的模块,其中既有免费的也有商业性的。

本章将介绍:

- 如何编写自己使用或者共享的简单模块。
- Python 中捆绑的模块。
- 导入模块的概念。
- 如何在包中包含独立于全局作用域的有效函数和名称。
- 如何利用作用域测试包。

## 7.1 模块

模块包含与一个公共主题相关联的一组函数、方法或者数据。这样的主题可能是网络组件(见第 16 章), 执行更加复杂的处理字符串和文本的操作(见第 12 章), 处理图形用户界面(见第 13 章), 以及其他服务。

在学习了如何使用一门语言编程之后, 您将发现经常需要使用一些组件, 这些组件最开始时并没有被捆绑到语言中。Python 也不例外。Python 是一个小巧简单的语言, 没有提供许多特性。然而, 因为它的简单性, 可以把它作为一个平台, 用一些任何人可用的函数和对象对它进行扩展。

### 7.1.1 导入可用模块

为了使用模块, 需要完成以下两个操作。首先要在操作系统上安装该模块。希望完成的大多数基本操作, 例如读写文件等(第 8 章将介绍更多这方面的内容), 以及其他一些重要的基础性操作(这些操作在不同的平台上是不同的), 都可以通过内置于 Python 的模块完成——即它们是免费的, 并且在语言中通用。

开始使用模块的最简单的方式是用 `import` 关键字导入它:

```
import sys
```

这条语句将导入名称为 `sys` 的模块, 它包含了 Python 提供的主要与各个系统有关的服务。这意味着它与系统的工作方式有关, 例如如何安装一个特定版本的 Python, 或者如何从命令行调用程序。

为了查看模块, 您将正在处理的文件编写为可以独立运行的程序。为此, 创建文件 `ch7.py`, 并输入如下代码:

```
#!/usr/bin/env python3.1
# Chapter 7 module demonstration
import sys
```

第一行用于在 Linux 和其他 Unix 系统中使用 `python`(或者是在基于 UNIX 的系统中使用 `python`, 例如 Cygwin)。在系统中还有其他 `python` 解释器时, 可以用上述方式运行 `python 3.1`。在 Windows 和 Macintosh 系统中, 文件的扩展名应该提供操作系统用来加载 Python 解释器所需的信息, 可能是 `python`、`python 3.1` 或者安装到系统上时使用的其他名称(尽管需要一些配置)。

### 7.1.2 通过已有模块创建新模块

为了创建模块, 首先要为该模块选择一个名称, 并在编辑器中打开以该名称命名的文件, 文件的扩展名是 `.py`。例如, 为了创建 `Foods` 模块, 仅需创建 `Foods.py` 文件。创建后, 可以通过使用“`Foods`”名称导入创建的模块, 而不用在模块名称尾部添加 `.py`。此时, 就已经成功导入了一个简单的模块!

## 试一试

## 创建模块

将第6章的所有源代码复制到文件 `Foods.py` 中。复制完成后，打开 `Python Shell`，导入 `Foods` 模块：

```
>>> import Foods
>>> dir(Foods)
['Fridge', 'Omelet', 'Recipe', '__builtins__', '__doc__', '__file__', '__name__']
>>>
```

## 示例说明

现在已经访问了 `Fridge` 类、`Omelet` 类，还有前面习题中的 `Recipe` 类。得到的文件就是一个模块，其中包含这些类，使得它们可以一起工作。不过现在需要通过名称 `Foods.Fridge`、`Foods.Omelet` 和 `Foods.Recipe` 访问这些模块，虽然有了一些新规则，它们仍旧完全可用。

这是您第一次直接运行本书中的示例！`Python` 维护了一个默认的目录列表，它在列表中的目录下寻找要加载的模块。这个列表包括了若干目录，它们的确切位置取决于安装 `Python` 解释器的方式。如果尝试导入 `Foods` 模块，但 `Python Shell` 却从一个未存放 `Foods.py` 文件的目录启动，就将得到一个错误(但是可以通过转换到正确的目录解决该问题)。

`Python` 自动搜寻的路径或者目录列表存储于 `sys` 模块的 `path` 变量中。为了访问该名称，要先导入 `sys` 模块。导入 `sys` 模块后，`sys.path` 才可以使用：

```
>>> import sys
>>> print(sys.path)
'C:/Python31/Chapter 6', 'C:\\Python30\\Lib\\idlelib',
'C:\\Windows\\system32\\python31.zip', 'C:\\Python31\\DLLs',
'C:\\Python31\\lib', 'C:\\Python31\\lib\\plat-win',
'C:\\Python31', 'C:\\Python31\\lib\\site-packages']
```

`sys.path` 是一个普通的列表，如果希望添加出现在 `sys.path` 之外的搜索目录，可以用普通方法更改系统路径——要么用 `append` 方法增加一个目录，要么用 `extend` 方法增加任意数量的目录。

导入 `Foods` 模块后，在输入一个名称时，它将弹出该模块作用域内所有名称的列表，代码编辑器的这种特性可以通过交互的方式提供帮助。每当输入一个句点时，如果输入的名称与一些名称关联，代码编辑器允许从该名称提供的接口做出选择。这有助于研究刚才创建的模块，但是在大型复杂模块中将更有用！

现在可以运行前面章节中的示例了，但是现在需要通过 `Foods` 模块访问类。例如，需要调用 `Foods.Fridge`，而不是 `Fridge`。如果希望单独访问 `Fridge`，您很快将看到如何完成该操作。

试一试 研究新模块

Python Shell 中的代码编辑器提供的特性可以在开发人员输入代码的过程中与其交互。您也许已经注意到，输入一个类或者一个模块的名称后，在名称后输入一个句点，Shell 将显示一个菜单，它包含模块或对象作用域中的名称。图 7-1 显示了这个特性，您也可以执行相同的操作。

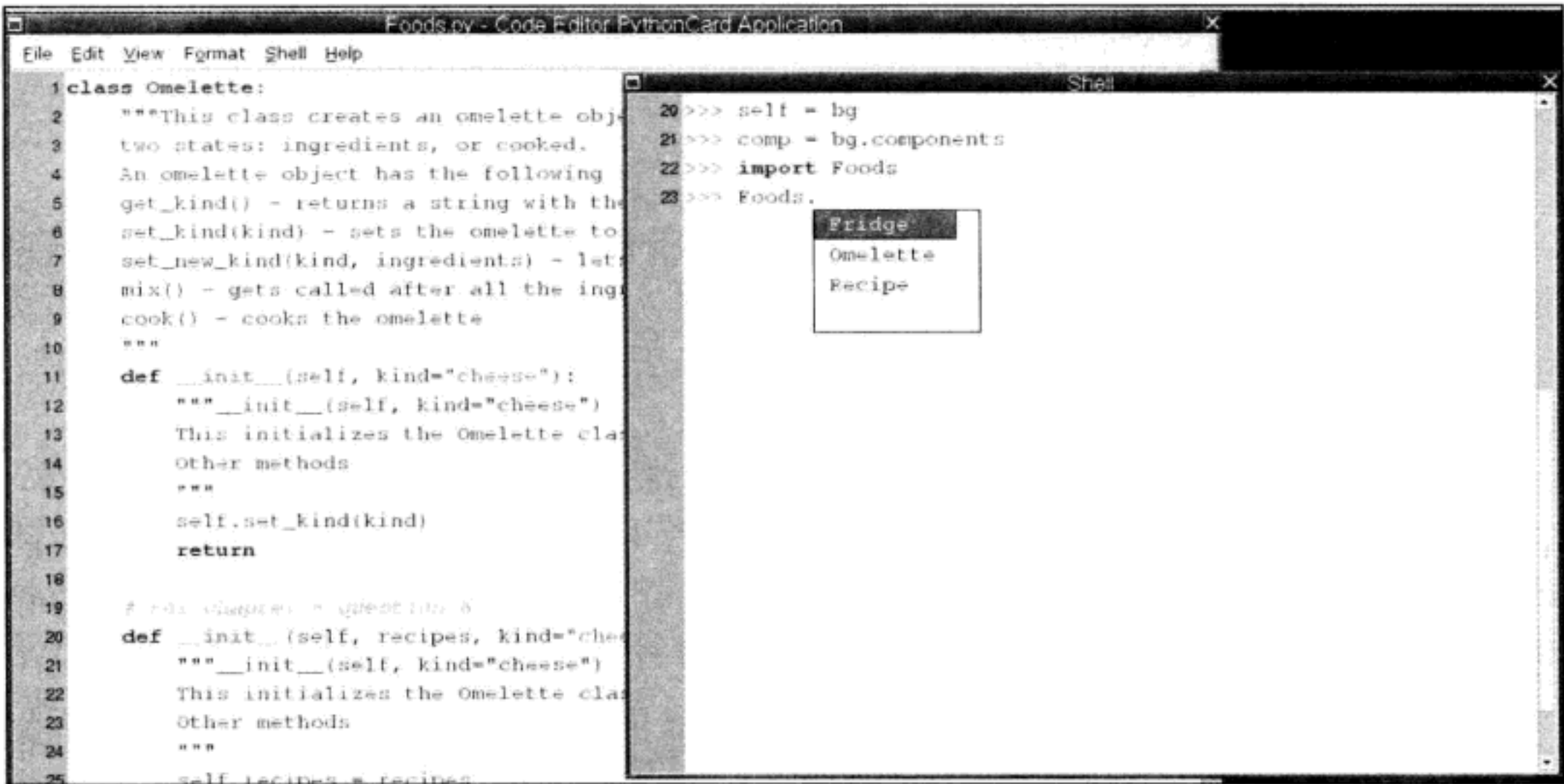


图 7-1

示例说明

在 Shell 中输入代码的同时，它将对输入的内容求值。当输入了特定字符时，Python 将对其执行操作。例如，无论输入哪种类型的引号，字符串都会改变颜色，还有就是在 Python 中关键字的颜色也不同。无论何时向 Python Shell 中输入一个句点后，它知道开发人员希望看到某个模块或者对象的内部，因此，它在后台查询该对象并显示结果。

7.1.3 从命令行开始使用模块

到目前为止，您已经开始使用 import，并在它后面跟着一个模块名称。当使用这种方式导入模块时，它包含的所有名称都被放到了该模块命名的作用域中，也就是说，作用域的名称就是 import 语句中使用的名称。

例如，在 sys 中，通过在 sys 后加上一个句点和内部名称，可以引用该模块内所有可用的内容，例如 sys.path 或者 sys.copyright，如字面意思所示，sys.copyright 指出了 Python 的版权。知道了模块的组织方式，就可以交互地使用代码编辑器 Python Shell，或者像第 6 章那样使用 dir 函数，研究 sys 模块(dir 甚至比代码编码器 Shell 的帮助对话框中显示的有效信息更多，它可以显示私有名称，这些私有名称不是模块接口。在类和对象中看到的概念依然适用于模块)。也可以研究模块以及模块中的函数和类中的文档字符串。

在 UNIX 和类 UNIX 环境中, 要求用户提供命令行参数是很正常的, 这是参数决定了整个程序的运行方式。从概念上讲, 这与在 Python 中函数参数的使用方式类似。这些命令行参数在 Python 程序中显示为 `sys` 模块中的一个特殊名称。这个名称是 `argv`。它初看起来不是非常有意义, 但需要重点了解它, 因为各个平台和语言都使用它。

`argv` 是 `argument vector` 的缩写。在程序语言 lingo 中, `argument` 是 `parameter` 的另外一个名称。当在命令行中运行带参数的程序时, `argument` 指命令行下函数的参数(命令行中参数的另外一个名称是 `flags`)。`vector` 是选项列表的别称。在某些语言中, 它有一个特定的与众不同的含义, 但 Python 不区分它们, 所以不必担心。

根据这些定义理解 `argv`, 可看出它的意思就是像列表那样访问的命令行参数。将 `argv` 转换成英语(或者任何其他非程序语言)中简短易理解的有意义的词非常困难, 因此词 `argv` 被一直保留下来。

为了从命令行中输出参数, 只需像使用其他列表那样使用 `sys.argv`:

```
>>>import sys
>>>print("This was given the command line parameters: %s" % sys.argv)
```

为了在其他平台上执行相同的过程, 需要从代码编辑器中启动。选择 `File|Run Options`, 并在 `Other argv` 域中输入希望的参数。前面从第 5 章开始也用这个功能, 但是使用 `Run Options` 对话框设置启动程序时的命令行, 这很新奇。

为了测试不断变化、并且不以交互式方式使用的程序, 一般来说最好使用 `python -i` 或者 `Run with Interpreter`, 这样就可以尝试从头开始重复运行程序。图 7-2 显示了一个列出了可选命令行选项的弹出窗口。

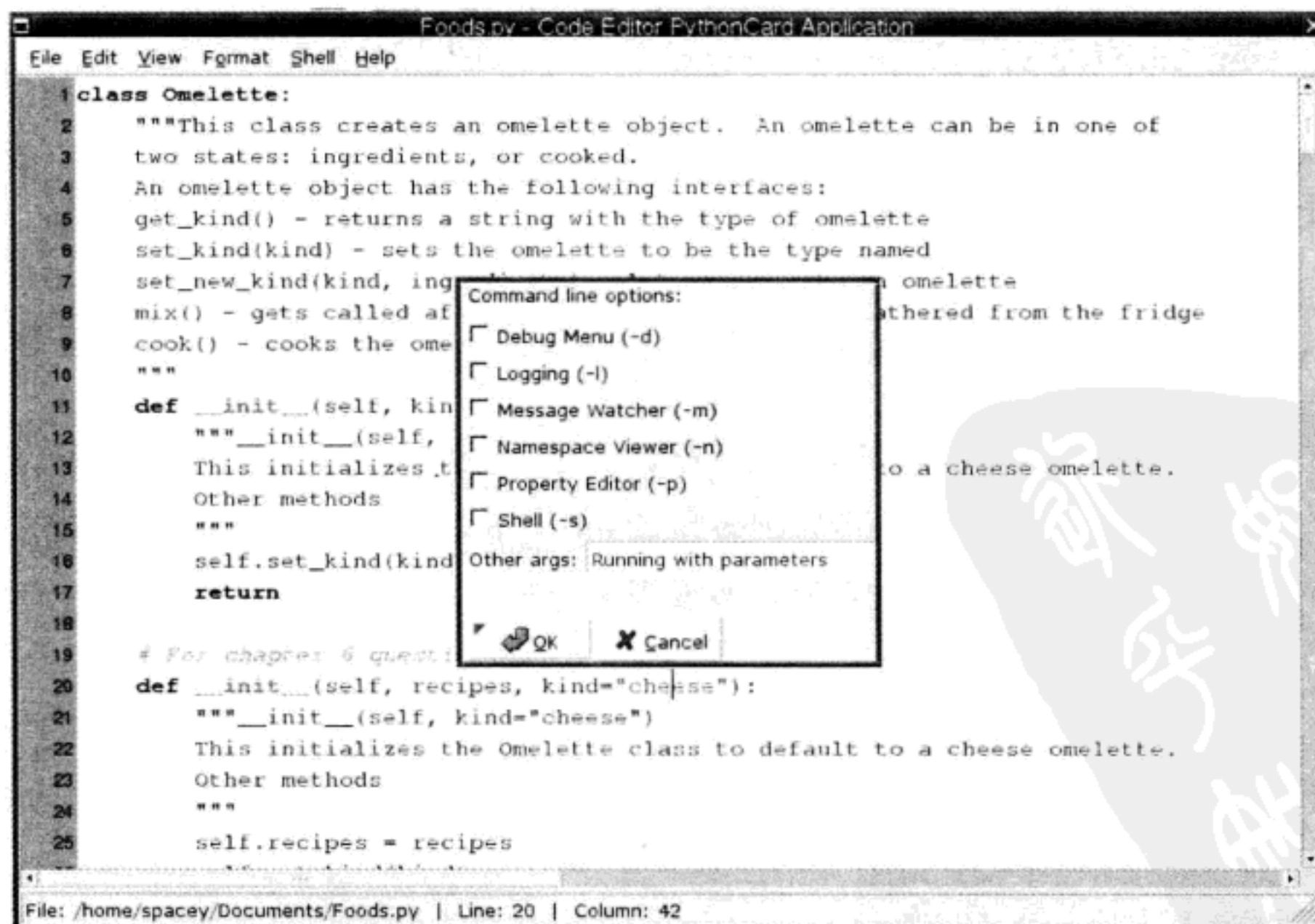


图 7-2

## 试一试

## 打印 sys.argv

任何时刻使用 File 菜单的 Run with Interpreter 选项运行程序，都可得到以列表形式输出的 sys.argv。例如，如果在 Other argv 中提供的命令行参数是“test 123 test”，程序将打印如下信息(这是 Windows 系统上的运行结果，UNIX Shell 下的 sys.path 看起来完全不同)：

```
This was given the command line parameters: ['D:\\Documents\\Chapter7.py',
'test', '123', 'test']
```

## 示例说明

sys.argv 列表的第一个元素永远都是程序的名称，其他元素都是 sys.argv 列表的元素，从位置 1 开始。

模块中的类的访问方式与其他任何名称的访问方式相同。对于提供了类的模块，其调用与预期一样，即在拼写出的完整路径后加上圆括号，例如调用 Foods.Recipe()。

## 7.1.4 改变导入方式

Import 可以单独使用，此时它会创建一个命名的作用域，模块中的所有内容都可从该作用域引用。有时，将模块的特定部分引入到程序顶层的全局作用域很有用。这样，在访问函数或者类之前将不必输入模块名称，从而可以减少输入量并使代码变得直接易读。利用下面的代码和 Foods 模块可创建一个洋葱煎蛋卷对象：

```
import Foods
r = Foods.Recipe()
onion_ingredients = Foods.Omelet(r, "onion")
```

该示例表明，希望调用或者访问模块中的某方法时，要拼写出完整的路径。您很快就会感到厌烦。然而，将 from 修饰符用于 import 命令，可以使名称与代码更接近：

```
from Foods import Omelet
from Foods import Recipe
r = Recipe()
onion_ingredients = Omelet(r, "onion")
```

如果必须深入多个级别，例如(制成的食物)Foods.Recipes.Breads.Muffins.Bran，并且要在当前作用域直接使用 Bran 中的名称，需要写类似的代码，如下所示：

```
from Foods.Recipes.Breads.Muffins import Bran
```

## 7.2 包

在单独的文件中创建一个模块后，该单独的文件经常会陷入组织问题。主要问题是，单独的类在单独使用时更有用，它比模块中其余的类有更多代码。这也是将它移到单独的

文件中的好理由，但这会毁坏使用该模块的代码！然而，有一个解决办法。

为了解决上述问题，Python 提供了包。包使用与操作系统中类似的目录(也叫做文件夹)结构，使得同一目录下的文件在一起使用时就像单独的模块。

可以从简单地创建目录开始。将 Foods 模块分解。首先，需要使用一个新名称，因为 Foods.py 已经存在，如果继续使用“Foods”会引起困惑。因此，首先创建一个新包 Kitchen(这个名称足够通用，为日后使用留下了足够的空间)。

创建 Kitchen 目录，并在其中创建 `__init__.py` 文件(这个名称必须与类中该方法的名称相同，并且名称前后各有两个下划线)。该文件提示 Python 这是个包目录，而不仅是包含 Python 文件的普通目录。这很重要，因为它确保开发人员知道要维护目录和控制它的行为。`__init__` 文件包含了控制包的用法的代码，因为与模块不同，当导入包时，目录中的每个文件并没有立即导入。相反，`__init__.py` 文件首先被计算，它指定了要使用的文件以及使用它们的方式。

### 试一试

#### 在 Kitchen 包中创建文件

为了将已经编写的 3 个类作为 Kitchen 包的一部分，在 Kitchen 目录下创建 4 个文件，将相应的类放到每个文件中，并以类的名称命名文件。记住，在 Windows 所有版本下，应当将看到的斜杠(/)替换为一个反斜杠(\)，因为 Windows 用它分隔目录。换言之，在 Kitchen 目录中创建文件 Kitchen/Fridge.py，并仅将 Fridge 类放到该文件中。

为每个类创建一个文件，并创建 `__init__.py` 文件：

- Kitchen/Fridge.py: 该文件包含 ch6.py 中从 Fridge 类的声明处开始的，Fridge 类的所有代码和注释。
- Kitchen/Omelet.py: 该文件包含 Omelet 类的所有代码和注释。使用在第 6 章的习题中修改后的 Omelet 类。
- Kitchen/Recipe.py: 该文件包含 Recipe 类的所有代码和注释。
- Kitchen/`__init__.py`(记住在文件名前后各有两个下划线): 本文件是空的。

#### 示例说明

现在，除了为每个类创建了文件，还创建了一个 `__init__.py` 文件，可以导入 Kitchen 包了。然而，导入 Kitchen 后，Python 只计算 `__init__.py`。这是个非常重要的细节，因为如果没有在 `__init__.py` 中放置代码，就将永远看不到代码。目前，没有导入任何东西，您默认地导入了 Kitchen。

导入 Kitchen 后，为了使所有类可用，需要将显式的 import 语句放入 `__init__.py` 文件：

```
from Fridge import Fridge
from Recipe import Recipe
from Omelet import Omelet
```

向 `__init__.py` 加入这些行后，导入 Kitchen 包后这些类都变得可用：

```
>>> import Kitchen
>>> r = Kitchen.Recipe()
>>> r.recipes
{'cheese': {'cheese': 1, 'eggs': 2, 'milk': 1}, 'onion':
{'cheese': 1, 'eggs': 2, 'milk': 1, 'onion': 1}, 'mushroom':
{'cheese': 1, 'eggs': 2, 'milk': 1, 'mushroom': 2}}
```

此处，它还没有带来很多好处，因为这只是一个非常小的项目，但是对于逐渐增长的项目，这个工具非常重要，它使得多个开发人员共同进行开发变得容易，通过将函数和类自然地分配到多个文件中，每个程序员可以在自己那组文件上工作。

## 7.3 模块和包

现在已经定义了模块和包，接下来将介绍如何使用它们。模块和包一般可以交换使用，对于两者在行为上存在区别的地方，后面的内容会特别指出。因为模块被命名为 `Foods`，包被命名为 `Kitchen`，所以在处理的是包的代码还是模块的代码方面您不会感到困惑。记住：`Kitchen` 是包，`Foods` 是模块。

### 7.3.1 将所有内容引入当前作用域

注意模块的一个特性：有时需要不用显式地指定模块中每个可用的名称而使用模块的所有内容。为此，Python 提供了一个特殊字符 `*`，它用于 `from...import...` 语句。记住，在导入到全局作用域时，只能使用 `*` 进行导入，理解这一点很重要：

```
From Foods import *
```

这将 `Omelet` 类以及 `recipe` 模块顶层的所有类都引入到当前作用域中。现在不必再输入 `Foods.Omelet()`，仅需输入 `Omelet()`，并且仅需输入一次，而不是每次在需要将某个名称变为局部名称时都要输入。

包与模块的工作方式类似，但它们内部的工作方式不同。对于包，当使用 `from...import *` 时，需要显式指定希望得到的名称。在 `__init__.py` 文件中使用 `__all__` 列表可使 `Kitchen` 包中的 3 个模块变得可用。出现在 `__all__` 列表中的所有名称也仅有这些名称可通过 `*` 导出。

`__all__` 列表中出现的元素可以是函数、类的名称或数据，它们都可以通过 `import *` 被自动导入到程序的全局作用域。

模块和类的使用者在程序中会自动使用 `from...import *` 语法。为了能够处理包，必须指定一个要导出的名称列表。然而，如果模块较大，也可以在模块文件的顶部创建 `__all__` 列表，它将像在包中那样把名称限制在模块中。

#### 试一试

#### 从包中导出模块

由于 `from...import *` 的经常使用，`__all__` 列表才得以存在。您将首先使用、然后再编

写包含很多层、函数、数据名称以及单独模块的包，用户看不见这些内容，因为它们不是公共接口的一部分。需要非常小心，避免给用户许多他们不需要的内容，`__all__`列表会强迫您确定接口的内容。

```
__all__=['Fridge','Recipe','Omelet']
```

### 示例说明

使用语句 `from Kitchen import *` 后，这些名称将进入程序的全局作用域。了解这点非常重要。如果 `__init__.py` 文件如下所示：

```
from Fridge import Fridge
from Omelet import Omelet
__all__ = ['Omelet', 'Recipe', 'Fridge']
```

如果删除 `from Recipe import Recipe` 语句，那么调用 `from Kitchen import *` 之后，需要调用 `Recipe.Recipe()` 创建一个新的食谱类。

## 7.3.2 重新导入模块和包

编程是一个试验并改正错误的过程。在交互式的 Shell 中，您经常意识到已经在模块中犯了一个错误。因为在加载问题模块前，为了正确地设置 Shell，您也许输入了大量的代码，所以您希望能够修复问题模块并让 Python 重新加载它，这样可以不用再次设置会话。目前还没介绍这种方法，但是可以实现这个操作。

第一件需要了解的事情是，多个模块共享一个公共模块，并且多次在同一个程序中调用它是很正常的现象。在这种情况下，为了不必每次花额外的时间重新加载、重新计算、重新编译该模块(见“编译和.pyc 文件”部分)，Python 将模块的名称以及模块的来源都隐藏在一个特殊的 `sys.modules` 下，该 `sys.modules` 包含了目前已经导入的全部模块。实际上，从代码编辑器中使用 Python Shell 时，它已经加载了 `sys` 以及其他许多模块，因此，每次调用 Python Shell 时，这些模块已经被加载！

### 编译和.pyc 文件

看看已经处理过的 `ch5.py` 文件、`ch6.py` 文件或者其他的 Python 文件，可以注意到在运行它们之后，会出现一个名称几乎一样的新文件，唯一不同之处是这个新文件的名称以 `.pyc` 为后缀。这是 Python 输出的一个特殊文件，它包含了某种形式的代码，这些代码比纯文本源代码更易于快速地加载和运行。如果对 `.py` 文件做了修改，下次调用它时(通过双击它，或者运行 `python -i`，或者在代码编辑器使用 `Run` 或 `Run with Interpreter` 菜单选项)，Python 将根据更新过的源代码重新创建 `.pyc` 文件。

### 试一试

### 检查 `sys.modules`

查看 `sys.modules.keys` 返回的列表，可以看到已加载过的每个模块的名称。即使在代码

编辑器以外启动了一个 Python Shell，在导入 sys 后，通过查看 sys.modules，可以发现系统在您不知情的情况下已经加载了许多模块。每个操作系统和不同 Python 版本的 sys.modules 字典的确切内容稍有不同，但它通常与下面的内容类似：

```
>>> list(sys.modules.keys())
['heapq', 'tkinter.filedialog', 'functools', 'random', '_bisect',
'idlelib.macosxSupport', 'ctypes._endian', 'builtins', 'struct',
'tempfile', 'imp', 'collections', 'idlelib.MultiCall',
'tkinter.simplesdialog', 'zipimport', 'string', 'encodings.utf_8',
'_bytesio', 'tkinter.constants', 'bisect', 'signal',
'idlelib.IOBinding', 'pydoc', 'threading', 'token', 'tkinter._fix',
'dis', 'locale', 'idlelib.TreeWidget', 'idlelib.rpc', 'encodings',
'idlelib.RemoteDebugger', 'abc', '_thread', '_tkinter', 'bdb', 're',
'idlelib.RemoteObjectBrowser', 'ntpath', 'math', 'idlelib.Debugger',
'inspect', '_ctypes', 'ctypes', 'codecs', '_functools', '_locale',
'idlelib.AutoComplete', 'tkinter', 'socket', 'traceback', '_stringio',
'queue', 'itertools', 'opcode', '_pickle', 'idlelib.StackViewer',
'idlelib.CallTipWindow', 'os', 'marshal', '__future__',
'idlelib.AutoCompleteWindow', '_collections', '_sre', 'operator',
'array', 'select', '_heapq', 'idlelib.ZoomHeight', 'pkgutil',
'errno', '_socket', 'binascii', 'sre_constants', 'encodings.latin_1',
'os.path', 'tokenize', '_warnings', 'idlelib.HyperParser',
'encodings.cp1252', '_struct', 'unicodedata', 'keyword',
'tkinter.commondialog', 'tkinter.messagebox', 'stringprep',
'encodings.aliases', 'fnmatch', 'sre_parse', 'pickle', '_fileio',
'reprlib', 'sre_compile', 'socketserver', '_random', 'site', 'io',
'__main__', 'copyreg', 'configparser', '_weakrefset', '_abcoll',
'_codecs', 'nt', 'idlelib.PyParse', 'genericpath', 'stat', 'warnings',
'sys', 'idlelib.CallTips', 'idlelib.configHandler', 'types',
'idlelib.ScrolledList', '_weakref', 'idlelib.ObjectBrowser',
'idlelib', 'tkinter.dialog', 'linecache', 'encodings.idna',
'time', 'idlelib.WindowList', 'idlelib.run']
```

### 示例说明

依赖于具体的操作系统，当调用 sys.modules 字典时，它将显示已调用过的所有模块。对于显示结果中没有显式地导入的模块，可以认为 Python 自动调用了它们，用来处理操作系统或者 Python 不要求您直接处理的任务。前面的示例来自于 Linux 系统，许多具体的操作显然与操作系统相关(例如 posix 和 posixpath，如果您用过 UNIX 就会了解)，而其他模块则不然。

可以利用这个机会查看与感兴趣的任何键所关联的值。可以看到一些模块被列作内置模块，一些被列作来自于文件，对于后者，模块文件的完整路径都列在了模块提供的信息中。不必担心 Python Shell 中显示的模块列表与前面的示例显示的模块列表不同。在加载了 Foods 模块后，它将出现在 sys.modules 字典中，此后，即使改变了 Foods.py，Python 也不再重新计算它。为了在交互式会话中修复这个问题，可以简单地将 Foods 模块从 sys.modules 字典中移除，然后再次导入它。由于 Python 在 sys.modules 中不再有该记录，它会按照您要求的方式工作，而不是采用像删除它的引用之前那样的方式工作：

```
>>> import Kitchen
>>> 'kitchen' in sys.modules
True
>>> sys.modules['Kitchen']
<module 'Kitchen' from 'Kitchen\__init__.py'>
>>> sys.modules.pop('Kitchen')
<module 'Kitchen' from 'Kitchen\__init__.py'>
>>> sys.modules['Kitchen']
Traceback (most recent call last):
  File "<input>", line 1, in ?
KeyError: 'Kitchen'
```

现在知道了 `sys.modules` 的底层工作方式, 还需要知道有一个简单的方法可以完成相同的功能。Python 提供了一个内置函数 `reload`, 它重新加载指定的模块, 效果与刚才手动重新加载它一样:

```
import Kitchen
reload(Kitchen)
<module 'Kitchen' from 'Kitchen\__init__.pyc'>
```

注意, 这不会影响已经存在的对象。即使您已经在刚才加载的模块里做了修改, 它们仍旧有可能关联到旧的定义。如果更改了 `Recipe` 和 `Omelet` 类, 需要重新调用这两个类, 并重新创建它们的对象, 对象的初始化如下所示:

```
>>> r = Omelet.Recipe()
>>> o = Omelet.Omelet(r, 'onion')
```

## 7.4 测试模块和包的基础知识

为模块创建的作用域有一个有趣的用法。程序中有一个特殊名称 `__name__`, 可以告诉您正在运行的作用域的名称。例如, 如果在 `Foods` 模块中检查 `__name__` 值, 它返回字符串“`Foods`”。

`__main__` 是一个特殊的保留字, 它是顶层全局作用域的名称。如果有一个从来不被直接使用的模块, 可以在其尾部粘贴一些代码, 这些代码只有一个用途, 即验证该模块可否正常工作。这使得对它的测试变得简单。

经常可以看到模块的尾部有如下的代码:

```
if __name__ == '__main__':
```

可以在自己的模块的尾部使用这个语句; 从此刻起, 可以测试类是否已经创建, 函数是否返回了期望的值, 或者执行其他任何您可以想到的测试。在程序中, 之前能够正常工作的代码突然崩溃是很常见的。在包中测试是否出现这种情况是个好主意, 这样将不再忘记会发生这种情况, 从而掌握主动权。第 12 章中将介绍更多有关测试的知识。

## 7.5 本章小结

前面的章节介绍了如何在交互的 Python Shell 中编写代码，以及将代码放到可以运行的单个文件中。本章介绍了如何将程序组织成模块和包。

Python 利用模块为局部名称保留一个作用域。模块内部的名称可被直接使用。然而，要在一个特定模块的外部(例如，名称实际上是 `__main__` 的全局作用域中)访问模块内的名称，首先需要指定模块名称，之后跟着一个句点，再后面是希望访问的名称。例如 `sys.path`。通过使用这种方式，可以在不同模块中将相同的名称用于不同的目的，而不会引起混乱。

为了使用一个模块，必须用 `import` 语句将它导入到程序中。`import` 将找到具有指定名称的模块文件，其扩展名为 `.py`，并使它可用。Python 检查 `sys.path` 列表中的每个目录，直到找到该文件为止。

通常希望在不输入完整路径的情况下，使用某个模块的特定部分。完整的路径以模块名称开始，后面跟着任何中间模块的名称(用句点分隔)，最后是实际需要的名称。此处，可以用 `from...import...` 导入频繁使用的名称。导入一个模块后，模块中不在任何函数或者类中出现的代码将被求值。

当要编写的代码量较大时，可以用包将代码组织起来，操作系统的文件系统提供了该功能。包结构以目录开始，将包导入程序后，该目录的名称就是包的名称。`__init__.py` 文件将普通的目录变为包。需要读 `__init__.py` 文件并对它解析，该文件包含对整个包有用的代码，例如包的所有部分共享的数据、版本信息和重要文件的存储位置等，文件中还包含必要的 `import` 语句，这些语句使包的其他部分能够正常工作。

当程序员用语句 `from...import *` 请求一个包时，包中的文件不会被自动导出，即使这些文件已经在 `__init__.py` 中被导入。应该在列表 `__all__` 中指定包中需要默认导出的名称。

### 本章要点：

- 使用 `import` 关键字导入模块。
- `argv` 是 `argument vector` 的缩写，它代表命令行参数。这些参数可以当作一个列表通过 `sys.argv` 访问。
- 每个操作系统都使用目录结构，包也使用目录结构，同一目录下的所有文件一起使用时，就像是一个单独的模块。
- 使用语句 `from modulename import *`，其中 `modulename` 是模块的名称，可以导入一个模块的全部内容。

## 7.6 习题

可以将代码直接移到模块和包中，而不必做出修改，这是 Python 比较好用之处。

本章的习题重点在于测试模块，因为测试实际上就是为自动化任务编写小程序。

1. 为 `Foods.Recipe` 模块写测试，创建包含一组食物的 `recipe` 对象，并且验证提供给它的关键与值都存在并且匹配。直接调用 `Recipe.py` 时运行该测试，而不是在导入它时

运行测试。

2. 测试 `Foods.Fridge` 模块, 向 `Fridge` 类添加食物, 因为 `get_ingredients` 需要一个 `Omelet` 对象, 练习使用除了 `get_ingredients` 之外的所有接口。
3. 对这些测试进行试验。从命令行直接运行它们。如果正确输入了测试代码, 不应当出现错误。试着引入错误, 以查看在测试中得到的错误消息。





# 第 8 章

## 文件和目录

本章将介绍 Python 中用来读写文件以及访问目录内容的函数和类型。这些函数很重要，因为几乎所有比较大的程序都用文件来读取输入或存储输出。

Python 提供了丰富的输入/输出函数，本章将介绍其中使用广泛的那些函数。首先介绍文件对象，它是 Python 中实现输入输出的基本方法。之后将介绍用于操作路径、获取文件信息和访问目录内容的函数。

本章将介绍：

- Python 中用于读写文件以及访问目录内容的类型和函数。这些函数很重要，因为几乎所有比较大的程序都用文件来读取输入或存储输出。
- Python 丰富的输入/输出函数，本章将介绍其中使用最广泛的那些函数。
- Python 中实现最基本的输入/输出的文件对象。
- 操作路径、获取文件信息和访问目录内容的函数。

### 8.1 文件对象

本章中的大部分示例使用 Windows 路径名称。如果您在其他平台上工作，用您的系统中的路径替换示例中的路径。

如果恰好使用 Windows，要记住反斜杠在 Python 字符串中是一个特殊字符，因此必须在路径中对反斜杠转义(即每次用两个反斜杠)。例如，路径 C:\Windows\Temp 用 Python 字符串"C:\\Windows\\Temp"代替。还可以通过在开头的引号前放置字母 r，取消字符串中对反斜杠的特殊处理，因此同样的路径可以写作 r"C:\Windows\Temp"。

我们将使用字符串对象来存储创建和访问的示例文件的路径。如果使用 Windows，输入如下代码(可以选择希望使用的任意路径)：

```
>>>path = "C:\\sample.txt"
```

如果使用 Linux，请输入下面的代码(或者自己选择一个路径)：

```
>>>path = "/tmp/sample.txt"
```

### 8.1.1 编写文本文件

下面从创建简单的文本文件开始介绍。要在系统上创建一个新文件，创建一个 `file` 对象，告诉 Python 希望向文件中写数据。`file` 对象代表对一个文件的连接，而不是文件本身，但是如果试图打开或者向一个不存在的文件写数据，Python 将自动创建该文件。输入如下代码：

```
>>> def make_text_file():
    a=open('test.txt','w')
    a.write("This is how you create a new text file")
    a.close()
```

可以从创建一个新的 `make_text_file()` 函数开始。告诉 Python 打开一个名为 `test.txt` 的文件。由于 Python 找不到该文件，就会创建该文件(注意，如果该文件存在，Python 会删除它并创建一个新文件，因此要小心使用这项技术！后面将学习如何在创建一个文件之前检查它是否已经存在)。“w”参数告诉 Python 打算向文件中写数据，如果没有指定该参数，Python 认为您打算从文件中读数据，若文件不存在，它将抛出异常。接下来，向文件中添加一行文本：“This is how you create a new text file”。

花点时间浏览 Python 安装目录，它应该类似于“C://Python31”。可以注意到一个新的 `test.txt` 的文件已经被创建。如果双击它，可以看到文件中已存在上面的示例中向其添加的文本。恭喜，您已经创建了第一个文件！

现在已经使用上面的技术创建了一个文件，接下来创建一个程序，首先检查某个文件是否已经存在。如果存在，将返回一个错误消息；如果不存在，程序将创建文件。输入下面的代码：

```
>>> import os
>>> def make_another_file():
    if os.path.isfile('test.txt'):
        print("You are trying to create a file that already exists!")
    else:
        f=open('test.txt','w')
        f.write("This is how you create a new text file")
    ...
>>> make_another_file()
"You are trying to create a file that already exists!"
```

打开一个文件时，利用本章讨论的其他文件处理函数，可以指定一个相对路径(相对于当前目录的路径，当前目录是指程序或者 Python 正在运行的地方)或者一个绝对路径(以磁盘或者文件系统的根目录开始的路径)。例如，`/tmp/sample.txt` 是一个绝对路径，而 `sample.txt` 由于没有指定上层目录，是相对路径。

### 8.1.2 向文件中追加文本

向文件中追加文本很容易实现。追加文本不是用参数(“w”)实现的，而是用追加参数

(“a”)。这样可以保证文件中已有的数据不被重写，而是将新文本追加到文件末尾。尝试下面的代码：

### 试一试

### 向文件中追加文本

```
>>> def add_some_text():
    a=open('test.txt','a')
    a.write("Here is some additional text!")
...
>>> add_some_text()
```

#### 示例说明

该示例创建了一个叫做 `add_some_text()` 的函数。之后使用 `open()` 方法打开文件 `test.txt`，告诉 Python 希望向文件中追加文本(使用参数“a”)。接下来，向文件中写入新文本。稍后调用该函数时，它增加了一行文本“Here is some additional text!”

可以到 Python 的安装目录打开 `test.txt` 文件查看结果。应当会看到新的文本被添加到了文件的末尾。

注意，`write` 函数不会自动写换行符，任何时候想向文件中增加换行符时，必须使用转义序列 `\n`。空格也一样。如果不自己添加空格、制表符或换行符，下一次向文件中添加的文本会紧跟在已有文本之后。

如果再次使用 `write`，新的文本又被追加到之前写的文本之后。如果传递的字符串超过一行，将有多行文本被添加到文件中：

```
>>> def even_more_text():
    a=open('test.txt','a')
    a.write("""
here is
more
text""")
...
>>> even_more_text()
```

此处使用了一个多行的、三引号的字符串。直到使用结束三引号，Python 会一直用“...”提示继续输入当前字符串。Python 在行之间加入了换行符。再次打开 `test.txt` 文件，看看有什么变化。

### 8.1.3 读文本文件

读文件是类似的。首先，通过创建一个 `file` 对象打开文件。这一次，使用“r”告诉 Python 打算读文件。“r”是默认参数，可以省略。

```
a=open("test.txt","r")
```

确保使用了前面创建的文件路径，或者使用希望读的某个文件的路径。如果文件不存

在, Python 将抛出一个异常。

可以使用 `readline` 方法从文件中读出一行文本。当在一个文件对象上第一次调用该方法时, 它将返回该文件的第一行文本:

```
>>> a.readline()
'This is how you create a new text file\n'
```

注意, `readline` 返回的字符串末尾是换行符。若要逐行读取文件内容, 可重复调用 `readline` 方法。

还可以使用 `read` 方法一次性地读取文件中的剩余内容。该方法返回文件中未读取的所有内容(如果在一打开文件时就调用 `read`, 它将该文件的完整内容作为一个长字符串返回)。

```
>>> f=open("test.txt","r")
>>> text=a.read()
>>> print(text)
This is how you create a new text file
Here is some additional text
here is
more
text
```

因为使用了 `print` 打印文本, Python 将换行符看做真正的换行符, 而不是 `\n`。当完成读操作时, 删除文件对象并关闭文件。

```
>>> del a
>>> a.close()
```

让 Python 将一个文本文件分解为多行是很方便的, 但是如果一次可以得到所有的行会更好, 例如, 这样就可以将其用在一个循环中。`readlines` 方法可以实现这个目的: 它将文件的剩余内容作为一个字符串列表返回。如果希望打印文件中每一行的长度, 可以使用下面的代码:

```
def print_line_lengths():
    a=open("test.txt","r")
    text=a.readlines()
    for line in text:
        print(len(line))
```

### 试一试

### 打印示例文件中行的长度

使用函数 `print_line_lengths`, 可以检查刚才创建的文件, 并显示每一行的长度:

```
>>> print_line_lengths()
38
28
```

7  
4  
4

### 示例说明

每一行作为一个字符串读入。在读入一行时，将读入的字符串作为 `len` 函数的参数可以显示该行的长度。记住，每一行都包含换行符，因此看起来为空的行的长度为 1。

### 8.1.4 文件异常

因为 Python 程序对计算机的文件系统没有独占控制，在访问文件时它必须能够处理预料之外的错误。当 Python 在执行文件操作并遇到问题时，它将抛出一个 `IOError` 异常(异常在第 4 章中介绍)。该异常的字符串表示将描述遇到的问题。

有许多情况都会引起 `IOError` 错误，包括：

- 试图打开并读取一个不存在的文件。
- 试图在一个不存在的目录中创建文件。
- 试图打开一个没有读访问权限的文件。
- 试图在一个没有写访问权限的目录下创建文件。
- 计算机遇到磁盘错误(或者网络错误，如果在访问一个网络磁盘上的文件)。

如果希望在错误发生时程序能够优雅地做出反应，必须能够处理这些异常。在接收到一个异常时采取什么操作取决于程序的行为。某些情形下，在打印了一个警告消息之后，也许希望尝试另一个文件。在另外的情形下，也许必须询问用户接下来想做什么，或者如果错误不能恢复，就直接退出程序。

## 8.2 路径和目录

Windows、Linux、UNIX 和 Mac OS/X 上的文件系统有许多共同点，但是它们的某些规则、约定和功能略有不同。例如，Windows 使用反斜杠将一个路径中的目录名称分隔开，而 Linux 和 UNIX(Mac OS/X 是 UNIX 的一种)使用正斜杠。除此之外，Windows 使用驱动器名称，而其他系统则不用。这些不同之处是编写跨平台程序的障碍。Python 将路径和目录操作的繁琐细节隐藏在 `os` 模块中，以方便程序员使用。然而，使用 `os` 并不能解决所有移植问题，`os` 模块中的一些函数并不是在所有平台上都可用。本小节仅描述在所有平台上都可用的函数。

即使打算仅在一个平台上使用程序，并且预计能够避免大多数这样的问题，但如果程序很有用，很可能某一天有人会在其他平台上尝试运行该程序。因此最好使用 `os` 模块，它提供了许多有用的服务。不要忘记首先要导入 `os`，然后再使用它。

## 8.3 os 中的异常

`os` 模块中的函数在失败时会抛出 `OSError` 异常。如果希望程序在出错时行为友好，那

么必须处理这个异常。与 `IOError` 一样，异常的字符串表示描述了遇到的问题。

### 8.3.1 路径

模块 `os` 包含另外一个模块 `os.path`，它提供了操作路径的函数。由于路径也是字符串，因此可以使用普通的字符串操作方法组合和分解文件路径。但是如果这样，代码可能不易移植，也不能处理 `os.path` 知道的一些特殊情形。使用 `os.path` 操作路径，可使程序易于移植，且可以处理一些特殊情形。

使用 `os.path.join` 可将目录名称组合成路径。Python 使用适合操作系统的路径分隔符。在使用之前不要忘记导入 `os.path` 模块。例如，在 Windows 系统上，输入如下代码：

```
>>> import os.path
>>> os.path.join("snakes", "Python")
'snakes\\Python'
```

在 Linux 系统上，在 `os.path.join` 中使用相同的参数会得到如下不同的结果：

```
>>> import os.path
>>> os.path.join("snakes", "Python")
'snakes/Python'
```

可以指定多于两个的名称。

函数 `os.path.split` 具有相反的功能，它将路径的最后一个组件提取出来。该函数返回包含两个项的元组：父目录的路径以及最后一个路径组件。这里有一个示例：

```
>>> os.path.split("C:\\Program Files\\Python30\\Lib")
('C:\\Program Files\\Python30', 'Lib')
```

在 UNIX 或者 Linux 系统上的结果如下：

```
>>> os.path.split("/usr/bin/python")
('/usr/bin', 'python')
```

自动分解序列在这里派上了用场。`os.path.split` 返回一个元组，该元组可分成几个部分，分别赋予等号左边的组件：

```
>>> parent_path, name = os.path.split("C:\\Program Files\\Python30\\Lib")
>>> print(parent_path)
C:\\Program Files\\Python30
>>> print(name)
Lib
```

尽管 `os.path.split` 仅将路径的最后部分分离开，但是有时候也许希望将一个路径完全分解为若干目录的名称。写一个这样的函数并不困难，只需对该路径调用 `os.path.split`，之后在父目录的路径上再次调用 `os.path.split`，依此类推，直到得到根目录。实现该功能的一种

得体的方式是使用递归函数，即自己调用自己的函数。它的形式如下：

```
def split_fully(path):
    parent_path, name = os.path.split(path)
    if name == "":
        return (parent_path, )
    else:
        return split_fully(parent_path) + (name, )
```

最关键的一行是最后一行，此处函数调用自身将父目录分解成各个组件。路径的最后一个组件 `name` 被添加到完全分解的父路径后面。`split_fully` 中间的行阻止函数无限次地调用自己。当 `os.path.split` 不能继续分解一个路径时，它返回的第二个组件为空，`split_fully` 注意到这一点并只返回父路径，而不再调用自身。

函数可以安全地调用自身，因为 Python 记录了函数每一个运行实例的参数和局部变量，即使运行实例是从另一个运行实例中调用的。在这种情形下，当 `split_fully` 调用自身时，即使内部(第二个)实例给 `name` 赋予了一个不同的值，外部(第一个)实例也不会丢失 `name` 的值，因为每个函数运行实例都有自己的变量 `name` 的副本。当内部实例返回后，外部实例继续使用它在进行递归调用时拥有的该变量的值。

编写递归函数时，要确保它不会无限次地调用自身。无限次调用自身是很糟糕的，因为它永远都不会返回结果(实际上，这种情形下，Python 会用完记录所有调用的空间，并抛出异常)。函数 `split_fully` 不会无限次调用自身，因为最终的 `path` 会足够短，并且 `name` 会变成一个空字符串，此时该函数不会再调用自身，而是直接返回。

该函数中有两处使用单个元素的元组，注意必须在圆括号中包含一个逗号。没有逗号，Python 会将圆括号解释为普通的分组圆括号，就像在数学表达式中那样：`(name,)` 是一个包含单个元素的元组，但 `(name)` 与 `name` 完全相同。

这里有一个可以运行的函数：

```
>>> split_fully("C:\\Program Files\\Python31\\Lib")
('C:\\', 'Program Files', 'Python31', 'Lib')
```

当有一个文件名称时，可以用 `os.path.splitext` 分解出它的扩展名：

```
>>> os.path.splitext("image.jpg")
('image', '.jpg')
```

对 `splitext` 的调用返回一个包含两个元素的元组，因此可以用上面的方式提取出扩展名：

```
>>> parts = os.path.splitext("image.jpg")
>>> extension = parts[1]
```

实际上，并不需要变量 `parts`。可以从 `splitext` 的返回值直接提取出第二个组件 `extension`：

```
>>> extension = os.path.splitext("image.jpg")[1]
```

`os.path.normpath` 也能派上用场，它可以规范化或“清理”路径：

```
>>> print(os.path.normpath(r"C:\\Program Files\\Perl\\..\\Python30"))
C:\\Program Files\\Python30
```

注意如何通过备份目录组件去掉“..”，以及如何修复双分隔符。函数 `os.path.abspath` 与 `os.path.normpath` 类似，它将一个相对路径(相对于当前目录的路径)转变为一个绝对路径(从驱动器或者文件系统的根目录开始)：

```
>>> print(os.path.abspath("other_stuff"))
C:\\Program Files\\Python30\\other_stuff
```

输出取决于调用 `abspath` 时的当前路径。您也许已经注意到，即使在 `Python` 目录下没有叫做 `other_stuff` 的文件或者目录存在，这个函数仍然可以正常工作。`os.path` 下面的所有路径操作函数都不检查正在操作的路径是否真正存在。

可以使用 `os.path.exists` 判断某个路径是否实际存在。它仅简单地返回 `True` 或者 `False`：

```
>>> os.path.exists("C:\\\\Windows")
True
>>> os.path.exists("C:\\\\Windows\\reptiles")
False
```

当然，如果使用的不是 `Windows`，或者 `Windows` 安装在另外一个目录(例如 `C:\\WinNT`)中，这两个调用都会返回 `False`！

### 8.3.2 目录内容

现在知道如何构造任意路径并且将它们分开。但是如何才能找出硬盘上实际存在哪些内容呢？`os.listdir` 模块会返回一个目录下所有名称条目，包括文件和子目录等内容。

#### 试一试

#### 获取目录的内容

下面的代码将得到一个目录下的条目列表。在 `Windows` 系统下，可以列出 `Python` 安装目录下的内容：

```
>>> os.listdir("C:\\\\Python31")
['Chapter 5', 'Chapter 6', 'Chapter 7', 'DLLs', 'Doc', 'ham', 'include',
'Lib', 'libs', 'LICENSE.txt', 'maybe', 'NEWS.txt', 'python.exe', 'pythonw.exe',
'README.txt', 'tcl', 'Test', 'Test.py', 'test.txt', 'test2.txt', 'test6.txt',
'tester.py', 'test.txt', 'Tools', 'w9xpopen.exe']
```

注意您的结果有可能不同，因为显示的结果取决于目录下的文件。

如果您使用其他的操作系统，或者在另外的目录下安装 `Python`，请将示例中的路径替换为其他路径。使用“.”可以列出当前目录。当然，如果列出一个不同的目录，将会得到

一个不同的名称列表。

无论哪种情况，都应该注意一些重要的事情。首先，返回的结果是目录条目的名称，而不是完整路径。如果需要某个条目的完整路径，必须使用 `os.path.join` 构造它。其次，结果中既有文件名称也有目录名称，从 `os.listdir` 的结果中无法区分两者。最后，注意结果中不包含“.”和“..”，这两个代表当前目录和其父目录的特殊目录名称。

编写一个函数，列出某个目录中的内容，但是需要打印出完整路径，而不是仅打印文件和子目录的名称，并且要求每行打印一个条目：

```
def print_dir(dir_path):
    for name in os.listdir(dir_path):
        print(os.path.join(dir_path, name))
```

该函数在 `os.listdir` 返回的列表上循环，并且对每个条目调用 `os.path.join`，在打印之前构造完整路径。尝试下面的代码：

```
>>> print_dir("C:\\Python30")
C:\\Python31\\DLLs
C:\\Python31\\Doc
C:\\Python31\\ham
C:\\Python31\\include
C:\\Python31\\Lib
C:\\Python31\\libs
C:\\Python31\\LICENSE.txt...
```

以上代码并不能保证 `os.listdir` 返回的条目列表以某种特定的方式排序，也就是说顺序是任意的。可能希望条目以某种特定顺序排序，以满足应用需求。由于它是字符串列表，因此可以使用 `sorted` 函数将它排序。

默认情况下，得到的结果按字母表排序，并区分大小写：

```
>>> sorted(os.listdir("C:\\Python31"))
['DLLs', 'Doc', 'LICENSE.txt', 'Lib', 'NEWS.txt', 'README.txt',
'Removepywin32.exe', 'Scripts', 'Tools', 'include', 'libs', 'py.ico',
'pyc.ico', 'python.exe', 'pythonw.exe', 'pywin32-wininst.log', 'tcl',
'w9xpopen.exe']
```

### 试一试

### 列出桌面或者主目录的内容

使用 `print_dir_by_ext` 列出桌面或者主目录的内容。Windows 的桌面是一个文件夹，它的典型路径是 `C:\\Documents and Settings\\username\\Desktop`，其中 `username` 是账户名称。在 GUN/Linux 或者 UNIX 上，主目录的典型路径是 `/home/username`。这是您期望的输出吗？

### 8.3.3 获取文件信息

可以很容易地判断出一个路径是指向一个文件还是指向一个目录。如果是指向文件，

`os.path.isfile` 将返回 `True`；如果是指向目录，`os.path.isdir` 将返回 `True`。如果路径不存在，这两个函数都返回 `False`：

```
>>> os.path.isfile("C:\\Windows")
False
>>> os.path.isdir("C:\\Windows")
True
```

### 其他类型的目录条目

在某些平台下，一个目录也许包含许多其他类型的条目，例如符号链接、套接字还有设备等。这些类型的条目的具体含义取决于特定的平台，相关内容比较复杂，这里不进行讨论。然而，`os` 模块为检查这些条目提供了支持，请查阅模块的文档了解与您的平台有关的细节。

### 递归目录列表

可以将 `os.path.isdir` 和 `os.listdir` 结合起来实现一些有用的操作：例如，递归地处理子目录。可以列出一个目录的内容，它的子目录，子目录的子目录，依此类推。对于这个目的，编写一个递归函数很有用。当函数找到一个子目录时，它调用自身，列出该子目录的内容：

```
def print_tree(dir_path):
    for name in os.listdir(dir_path):
        full_path = os.path.join(dir_path, name)
        print(full_path)
        if os.path.isdir(full_path):
            print_tree(full_path)
```

注意上面的函数与之前编写的 `print_dir` 函数很类似。然而，这个函数为每个条目构造了完整路径 `full_path`，因为这既符合打印的需求，也考虑到子目录的需求。最后两行代码检查它是否是子目录，如果是，该函数在继续运行之前通过调用自己列出子目录的内容。如果试用该函数，确保没有对一个非常大的目录树调用该函数，否则，不得不等待它打印出树中每一个子目录和文件的完整路径。

模块 `os.path` 中的其他函数提供了关于文件的信息。例如，`os.path.getsize` 在不必打开和扫描某个文件的情况下以字节为单位返回该文件的大小。使用 `os.path.getmtime` 可以得到文件上次被修改的时间。返回的值是 1970 年起到文件上次被修改的时间之间的秒数，而这并不是用户喜欢的日期的格式。必须调用另外一个函数 `time.ctime`，将该结果转换为易于理解的形式(不要忘了首先要导入 `time` 模块)。这里有一个示例输出了 Python 安装目录上次被修改的时间，这可能是在计算机上安装 Python 的日期和时间：

```
>>> import time
>>> mod_time = os.path.getmtime("C:\\Python30")
>>> print(time.ctime(mod_time))
Thu Mar 15 01:36:26 2009
```

现在知道了如何修改 `print_dir` 来打印目录的内容，包括每个文件的大小和修改时间。为了简单，下面的版本只打印条目名称，而不打印它们的完整路径：

```
def print_dir_info(dir_path):
    for name in os.listdir(dir_path):
        full_path = os.path.join(dir_path, name)
        file_size = os.path.getsize(full_path)
        mod_time = time.ctime(os.path.getmtime(full_path))
        print("%-32s: %8d bytes, modified %s" % (name, file_size, mod_time))
```

最后一条语句使用了第 1 章和第 2 章中介绍的 Python 内置的字符串格式化方法，以产生整洁的对齐的输出。如果希望输出其他文件信息，请浏览 `os.path` 模块的文档，学习如何获取这些信息。

### 8.3.4 重命名、移动、复制和删除文件

模块 `shutil` 中包含了操作文件的函数。可以使用函数 `shutil.move` 重命名一个文件：

```
>>> import shutil
>>> shutil.move("server.log", "server.log.backup")
```

或者，可以使用它将一个文件移动到另外一个目录下：

```
>>> shutil.move("old mail.txt", "C:\\data\\archive\\")
```

您也许已经注意到，`os` 也包含一个可以重命名和移动文件的函数 `os.rename`。一般应当使用 `shutil.move`，因为使用 `os.rename` 可能并不能指定一个目录名称作为目标，而且在某些系统上，`os.rename` 不能将一个文件移动到另外一个磁盘或者文件系统中。

`shutil` 模块还提供了 `copy` 函数，它将一个文件复制为具有一个新名称的文件，或者复制到一个新目录下。可以简单地使用如下代码：

```
>>> shutil.copy("important.dat", "C:\\backups")
```

删除文件是最简单的操作，只需调用 `os.remove` 即可：

```
>>> os.remove("junk.dat")
```

如果您是一位老派的 UNIX 黑客(或者希望自己冒充成一个黑客)，可能更喜欢 `os.unlink`，它能完成相同的操作。

#### 文件权限

文件权限在不同的平台上的工作方式是不同的，解释它们超出了本书的讨论范围。然而，如果需要改变一个文件或者目录的权限，可以使用 `os.chmod` 函数。它与 UNIX 或者 Linux 的系统调用 `chmod` 的工作方式相同。参考 `os` 模块的文档了解其细节。

### 8.3.5 示例：轮换文件

在这个示例中，将处理一个更困难的实际的文件管理任务。假设需要保留一个文件的多个老版本。例如，系统管理员要保留老版本的系统日志文件。通常，文件的老版本的名称有一个数字后缀，例如 `web.log.1`、`web.log.2` 等，其中较大的数字代表较老的版本。为了给文件的新版本预留空间，这些老版本被轮换：目前的版本 `web.log` 变成了 `web.log.1`，而 `web.log.1` 则变成了 `web.log.2`，依此类推。

手动实现该功能非常乏味，但是 Python 却可以很快地实现。有几个棘手的问题需要考虑。首先，文件当前版本与老版本的命名方式不同：老版本有一个数字后缀，而当前版本没有。解决这个问题一个方法是将当前版本作为版本 0。函数 `make_version_path` 为当前版本和老版本构造了正确的路径。

另外一个不易注意的地方是必须确保首先要重命名老版本。例如，如果在重命名 `web.log.2` 之前将 `web.log.1` 重命名为 `web.log.2`，后者将被重写，它之前的内容就会丢失，这并不是您所希望的。递归函数将再次伸出援手。该函数可以调用它自身，在重写下一个老版本的日志文件之前轮换它：

```
import os
import shutil

def make_version_path(path, version):
    if version == 0:
        # No suffix for version 0, the current version.
        return path
    else:
        # Append a suffix to indicate the older version.
        return path + "." + str(version)

def rotate(path, version=0):
    # Construct the name of the version we're rotating.
    old_path = make_version_path(path, version)
    if not os.path.exists(old_path):
        # It doesn't exist, so complain.
        raise IOError("'s' doesn't exist" % path)
    # Construct the new version name for this file.
    new_path = make_version_path(path, version + 1)
    # Is there already a version with this name?
    if os.path.exists(new_path):
        # Yes. Rotate it out of the way first!
        rotate(path, version + 1)
    # Now we can rename the version safely.
    shutil.move(old_path, new_path)
```

花几分钟时间研究一下上面的代码和注释。`rotate` 函数使用了递归函数通用的技术：第二个参数用于处理递归的情形，在这个示例中，即文件的版本号被轮换。该参数的默认值为 0，它表示文件的当前版本。当调用该函数时(与函数调用自己的情况不同)，不需要指

定该参数的值。例如，可以直接调用 `rotate("web.log")`。

该函数检查正在被轮换的文件是否确实存在，如果该文件不存在，则引发异常。假设希望轮换一个不确定是否存在的系统日志文件。解决问题的一种可能方法是在该日志文件不存在时，创建一个空的日志文件。回忆一下，当以写方式打开一个并不存在的文件时，Python 会自动创建它。如果没有向新文件中输入内容，它将为空。下面是一个轮换可能存在的日志文件的函数，如果不存在，首先要创建日志文件。它使用之前编写的 `rotate` 函数。

```
def rotate_log_file(path):
    if not os.path.exists(path):
        # The file is missing, so create it.
        new_file = file(path, "w")
        # Close the new file immediately, which leaves it empty.
        del new_file
    # Now rotate it.
    rotate(path)
```

### 8.3.6 创建和删除目录

创建一个空目录甚至比创建一个空文件更容易。仅需调用 `os.mkdir` 就可以实现该操作。然而，要创建的目录的父目录首先要存在。如果父目录 `C:\photos\zoo` 不存在，下面的代码将引发异常：

```
>>> os.mkdir("C:\\photos\\zoo\\snakes")
```

可以使用 `os.mkdir` 函数创建它的父目录，但一种更简单的方法是使用 `os.makedirs`，该函数可以创建不存在的父目录。例如，下面的代码将在必要的时候创建 `C:\photos` 和 `C:\photos\zoo`：

```
>>> os.makedirs("C:\\photos\\zoo\\snakes")
```

使用函数 `os.rmdir` 删除目录。该函数仅对空目录有效，如果要删除的目录不为空，首先需要删除该目录的内容。

```
>>> os.rmdir("C:\\photos\\zoo\\snakes")
```

上面的代码仅会删除子目录 `snakes`。

有一种方法可以在目录包含其他文件和子目录的情况下将该目录删除。函数 `shutil.rmtree` 可以实现该操作。然而，使用该函数时要谨慎。如果犯了一个编程或者输入错误，向该函数传入了一个错误的路径，它将删除一整组文件，您甚至不知道发生了什么情况！例如，下面的代码会删除完整的图片集，包括 `zoo`、`snakes` 等：

```
>>> shutil.rmtree("C:\\photos")
```

8.3.7 通配

如果您曾经使用过 Windows 系统的命令行提示符，或者 GUN/Linux、UNIX、Mac OS X 的命令行 shell，可能看到过通配符模式。通配符是一些特殊字符，例如\*和?，可以使用它们匹配许多名称类似的文件。例如，使用模式 P\*可以匹配名称以 P 开头的所有文件，使用\*.txt 可以匹配所有后缀名为.txt 的文件。

通配(globbing)是黑客们的行话，用来表示在文件名称模式中展开通配符。Python 在模块 glob 中提供了名称也为 glob 的函数，它实现了对目录内容进行通配的功能。glob.glob 函数接受模式作为输入，并返回所有匹配的文件名和路径名列表，这与 os.listdir 类似。

通配和大小写区分

在 Windows 操作系统下，模式 M\*可以匹配名称以 m 和 M 开头的所有文件，因为文件名称和文件名称通配是不区分大小写的。在大多数其他操作系统上，通配是区分大小写的。

例如，试着使用下面的命令，列出 C:\Program Files 目录下名称以 M 开头的所有条目：

```
>>> import glob
>>> glob.glob("C:\\Program Files\\M*")
['C:\\Program Files\\Messenger', 'C:\\Program Files\\Microsoft Office',
'C:\\Program Files\\Mozilla Firefox']
```

由于您的计算机可能安装了不同的软件，所以输出可能与上面不同。可以看到 glob.glob 返回了符合模式的包含磁盘驱动符和目录名称的路径，这与 os.listdir 不同，它只返回指定目录下的名称。

表 8-1 列出了通配模式中可以使用的通配符。这些通配符与操作系统的命令 shell 下的通配符并不一定完全一致，但是 Python 的 glob 模块在所有的平台上都使用相同的语法。注意，通配模式的语法与正则表达式的语法类似但不相同。

表 8-1

通 配 符	匹 配	示 例
*	0 个或多个任意字符	*.m*匹配扩展名以 m 开头的名称
?	任意单个字符	???匹配恰好包含 3 个字符的名称
[...]	方括号中列出的任意一个字符	[AEIOU]*匹配以大写的元音字母开头的名称
[!...]	不在方括号中出现的任意一个字符	*[!s]匹配不以 s 结尾的名称

也可以在方括号之间使用某个范围内的字符。例如，[m-p]匹配 m、n、o、p 中的任意一个字母，[!0-9]匹配数字以外的任意字符。

通配是为文件操作选择一组相似文件的较为便捷的方法。例如，要删除目录 C:\source\ 中所有扩展名为.bak 的备份文件，只需执行如下所示的两行代码：

```
>>> for path in glob.glob("C:\\source\\*.bak"):
...     os.remove(path)
```

通配比 `os.listdir` 的功能强大得多，因为可以在目录或者子目录名称中指定通配符。对于这样的模式，`glob.glob` 可以返回多个目录下的路径。例如，下面的代码返回当前目录的所有子目录中扩展名为 `.txt` 的文件：

```
>>> glob.glob("*\\*.txt")
```

## 8.4 本章小结

本章介绍了如何从磁盘上的文件读数据，以及向文件中写数据。使用 `file` 对象可以向文件中写字符串，并且一次性地或逐行读取文件中的内容。可以使用这些技术读取一个程序的输入，生成输出文件，或者存储中间结果。

本章还介绍了路径，路径指定了文件在磁盘上的位置，以及如何操作它们。使用 `os.listdir` 或者 `glob`，可以列出磁盘上的内容。

本章要点：

- 文件对象代表了对一个文件的连接，而不是文件本身，但是如果以写方式打开一个并不存在的文件，Python 会自动创建该文件。
- 使用 `append` 方法而不是 `write` 方法，可向文件中追加内容。这确保文件中的数据不被重写。
- 使用参数 “r” 从文件中读数据，例如 `a=open("test.txt","r")`
- `readline` 方法返回文件中的第一行文本。
- 当结束读文件时，确保要显式地关闭该文件，并删除文件对象。
- 模块 `os` 中的 `os.path` 子模块提供了操作路径的函数。
- 模块 `os.listdir` 可以列出一个目录下的文件、子目录等内容。
- “通配”是黑客的行话，用来表示展开文件名称模式中的通配符。Python 在模块 `glob` 中提供了同名函数 `glob`，它实现了对目录内容的通配操作。函数 `glob.glob` 接受 `glob` 模式作为输入，并返回匹配的文件名或路径名称列表，这与 `os.listdir` 函数功能类似。

## 8.5 习题

1. 创建非递归函数 `print_dir` 的另外一个版本，它首先列出所有子目录的名称，之后列出目录中文件的名称。子目录的名称按字母表顺序排列，文件名称也以该规则排列（为了获得更好的成绩，要使函数仅调用 `os.listdir` 一次。Python 操作字符串比执行 `os.listdir` 要快）。
2. 修改 `rotate` 函数，使其保留固定数目的旧版本文件。版本数目通过另外一个参数指定。超过该数值的额外旧版本应当被删除。



# 第 9 章

## Python 语言的其他特性

本章介绍 Python 中一些不常用的特性，以及一些常用的模块。每节描述至少一种使用特性的典型方式，并给出示例代码。

前几章介绍了一些普通的函数，并且还介绍了如何创建自己的函数。Python 最具吸引力之处在于它提供了广泛的内置函数和模块，可以用来处理许多困难的问题。在学习如何构建自己的模块之前，先看看 Python 提供的标准模块以更好地理解它们的用法。

本章将介绍：

- 如何使用 `lambda` 和 `filter` 函数。
- 如何使用 `map` 来避免循环。
- 字符串替换。
- `getopt` 模块。

### 9.1 `lambda` 和 `filter`：简单匿名函数

有时需要简单的函数调用，这可能是因为函数不是很通用，或者函数非常特殊，如果在代码中的另外一个位置调用，用法应该完全不同。有一个特殊的操作 `lambda`，可以处理这些情形。`lambda` 本身不是一个函数，而是一个关键字，它告诉 Python 创建一个函数并在适当的位置使用它，而不是通过一个名称引用它。

为了演示 `lambda` 的用法，下面的示例使用 `filter` 函数，它接受一个列表并基于函数中定义的标准移除元素。可以使用普通的函数实现该操作，但是在简单的情况下，比如只需要奇数(或者奇数的元素，或者以某些符号开头的字符串，等等)，一个完整定义的函数就有些大材小用了。

```
# use lambda with filter
filter_me = [1, 2, 3, 4, 6, 7, 8, 11, 12, 14, 15, 19, 22]
# This will only return true for even numbers (because x%2 is 0, or False,
# for odd numbers)
result = filter(lambda x: x%2 == 0, filter_me)
print(*result)
```

由于缺少名称，`lambda` 创建的函数被称为匿名函数。然而，可以使用 `lambda` 语句的

结果给函数绑定一个名称。该名称只在创建名称的作用域内可用，就像其他名称一样：

```
# use lambda with filter, but bind it to a name
filter_me = [1, 2, 3, 4, 6, 7, 8, 11, 12, 14, 15, 19, 22]
# This will only return true for even numbers (because x%2 is 0, or False,
# for odd numbers)
func = lambda x: x%2 == 0
result = filter(func, filter_me)
print(*result)
```

`lambda` 只能是一个简单函数，并且不能包含其他语句，比如为变量创建一个名称。在 `lambda` 内部，只能执行有限的操作，比如测试相等性、将两个数相乘或者以特定的方式使用其他已经存在的函数。但是不能使用像 `if ... : elif ... : else:` 这样的构造，甚至不能为变量创建新的名称！只能使用传递到 `lambda` 函数的参数。然而，可以通过使用 `and` 和 `or` 操作，执行比简单声明语句复杂一些的操作。但是仍然应该记住 `lambda` 只有非常有限的用法。

`lambda` 主要用在内置函数 `map` 和 `filter` 中。通过使用 `lambda`，这些函数能够以紧凑的方式执行一些大的操作，而不必使用循环。上面已经看到 `filter` 函数的应用，编写这个循环十分困难。

## 9.2 Map: 短路循环

匿名函数经常用在 `map` 函数中。`map` 是一个特殊的函数，它用于需要对列表中的每个元素执行一个指定的操作的情形。它在实现这种操作时不必编写一个循环。

### 试一试

### 使用 map

尝试如下基本测试：

```
# Now map gets to be run in the simple case
map_me = [ 'a', 'b', 'c', 'd', 'e', 'f', 'g' ]
result = map(lambda x: "The letter is %s" % x, map_me)
print(*result)
```

### 示例说明

就像在一个循环中那样，列表中的每个元素都将被访问。在 Python 先前的版本中，该测试同时返回一个列表，但是在 Python 3.1 中则返回了一个迭代器。以下就是它的结果：

```
>>> print(*result)
The letter is a The letter is b The letter is c The letter is d
The letter is e The letter is f The letter is g
```

需要知道一些关于 `map` 的特殊问题。如果向 `map` 中传递一个列表(或者元组，`map` 可以接受任何类型的序列作为参数)的列表，则它期待得到列表。主列表中的每个序列应该有

相同数目的元素：

```
# use map with a list of lists, to re-order the output.
map_me_again = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
result = map(lambda list: [ list[1], list[0], list[2]], map_me_again)
print(*result)
```

上面的代码得到一个列表的列表，其中每个列表的元素都被打乱了位置：

```
>>> print(*result)
[2, 1, 3] [5, 4, 6] [8, 7, 9]
```

`map` 总是返回一个迭代器。如果愿意，可以传递给 `map` 一个非匿名函数的名称，它仍然以相同的方式运行。

## 9.3 在列表中做出决策——列表解析

Python 2.0 中引入了称为列表解析的特性。它可以用来在列表解除引用操作符(方括号)中编写小的循环和判定，以此来定义用以限制被访问的元素范围的参数。

例如，要创建一个列表，并打印出列表中的偶数，可以使用列表解析：

```
# First, just print even numbers
everything = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 ]
print([ x for x in everything if x%2 == 0])
```

这是只将列表的一部分放入循环中的一种简洁实用的方式。列表中只有相关的部分，也就是在程序中希望得到的部分，才会被加入到循环中。

列表解析提供了与结合使用 `lambda` 和 `filter` 或 `map` 相同的功能，但是由于它可以包括循环和条件语句，因此能够提供更强大的做出决策的能力，而 `lambda` 只允许执行一个简单表达式。

在大多数情况中，列表解析将比其他方式运行得更快。

## 9.4 为循环生成迭代器

Python 有一个能够创建迭代器的特殊特性，即 `range` 函数：

```
f = range (10, 20)
print(*f)
```

这段代码会生成一个显而易见的结果：

```
>>> print(*f)
10 11 12 13 14 15 16 17 18 19
```

这段代码看起来并不深奥，但是它对于需要迭代特定次数并且不基于已有列表的 `for` 循环非常必要，这个数目可能在编写程序时不能确定，而只有在程序运行时才知道。

如果只给 `range` 传递一个数值，那么它将从 0 数到该数值。这个数值既可以是正数，也可以是负数：

```
for number in range(10):  
    print("Number is now %d" % number)
```

这将生成一个显而易见的输出：

```
Number is now 0  
Number is now 1  
Number is now 2  
Number is now 3  
Number is now 4  
Number is now 5  
Number is now 6  
Number is now 7  
Number is now 8  
Number is now 9
```

另外，如果只希望每隔一个数值或者每隔两个数值打印数值，可以使用一个可选的第三个参数 `step`，该参数描述 `range` 生成的每个数值之间的间隔：

```
for number in range(5, 55, 4):  
    print("Number from 5 to 55, by fours: %d" % number)
```

这会得到指定的一系列数值：

```
Number from 5 to 55, by fours: 5  
Number from 5 to 55, by fours: 9  
Number from 5 to 55, by fours: 13  
Number from 5 to 55, by fours: 17  
Number from 5 to 55, by fours: 21  
Number from 5 to 55, by fours: 25  
Number from 5 to 55, by fours: 29  
Number from 5 to 55, by fours: 33  
Number from 5 to 55, by fours: 37  
Number from 5 to 55, by fours: 41  
Number from 5 to 55, by fours: 45  
Number from 5 to 55, by fours: 49  
Number from 5 to 55, by fours: 53
```

在 Python 之前的版本中，一个程序可能会处理大量的元素——可能几十万甚至上百万个，在这种情况下，`range` 将创建一个包含要求的所有元素(例如，从 0 到 Internet 上所有可能的系统的个数)的数组。当要对序列中的全部元素进行检查时，每个元素都需要占用一些内存，结果是它可能会耗尽系统的所有内存。为了避免这类大型列表出现的各种问题，

Python 提供了一个称为 `xrange` 的特殊内置类，它只在内存中创建列表中的少量元素。在 Python 3.1 中，`range` 发生了变化，它不再创建一个列表，而是改为一个迭代器，本质上它可以以 `xrange` 的方式执行。`xrange` 也因此从语言中删除。

**试一试****检查 range 迭代器**

有趣的是，`range` 返回一个行为与列表类似的迭代器对象。注意这个对象没有公共接口。而仅有私有方法，这些私有方法看起来类似于大多数列表和元组所拥有的方法的子集：

```
>>> xr = range(0,10)
>>> dir(xr)
['__class__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattr__', '__getitem__', '__gt__', '__hash__', '__init__',
                                '__iter__',
 '__le__', '__len__', '__lt__', '__ne__', '__new__', '__reduce__',
                                '__reduce_ex__',
 '__repr__', '__reversed__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__']
```

直接调用它不返回列表，它生成一个对它的调用方法的表示：

```
>>> xr
range(0, 10)
```

然而，仍然可以通过同列表、序列以及字典相同的解除引用操作符(方括号)来访问它：

```
>>> xr[0]
0
>>> xr[1]
1
```

**示例说明**

`range` 生成一个没有任何公有方法的对象。它只有内置方法，这些方法使它能够像一个简单的序列那样使用。本质上，当使用方括号访问一个列表、元组或字典时，是在告诉 Python 要调用列表、元组或字典的 `__getitem__` 方法。`range` 对象具有这个私有方法，因此可以像序列那样被访问和解除引用。

当调用 `range` 对象时，它并不生成一个列表，而是告诉您它的生成方式，这样如果要知道它生成的数值，就可以查看参数。

需要注意，即使像一个序列一样执行，它与序列仍是不同的，这也是它比较酷的地方。

## 9.5 使用字典的特殊字符串替换

到目前为止还有一种特殊语法没有介绍，就是使用字典进行字符串替换。这在希望用

一种可配置的方式打印字符串时可以用到，例如一个格式化的报表或者类似的东西。

## 试一试

### 基于字典的字符串格式化

当这样做时，您希望从已知的元素集合中获取单个命名元素，如同字典中那样，并且用指定的顺序打印它们，打印的顺序可在程序外给定：

```
person = {"name": "James", "camera": "nikon", "handedness": "lefty",
          "baseball_team": "angels", "instrument": "guitar"}

print("%(name)s, %(camera)s, %(baseball_team)s" % person)
```

以上代码的输出如下：

```
>>> print("%(name)s, %(camera)s, %(baseball_team)s" % person)
James, nikon, angels
```

#### 示例说明

注意圆括号中的信息是键名，它的值将从字典中得到并替换为字符串。然而，为了能恰当地使用它，仍然需要在结束的圆括号后面指定插入的数据类型，以使字符串替换知道要做什么。在这里，所有的类型都是字符串，但是可以使用 *i*(表示整数)、*j*(表示虚数)、*l*(表示长整数)，以及其他所有的格式说明符。下面的示例中用到了其他格式。注意 `person` 应该出现在 `print` 语句的同一行中，而不能在下一行，它是一个长代码行的结尾：

```
person["height"] = 1.6
person["weight"] = 80
print("%(name)s, %(camera)s, %(baseball_team)s, %(height)2.2f,
      %(weight)2.2f" % person)
```

这将得到如下的简要输出：

```
>>> print("%(name)s, %(camera)s, %(baseball_team)s, %(height)2.2f,
          %(weight)2.2f" % person)
James, nikon, angels, 1.60, 80.00
```

这些例子的语法与前3章中学到的语法几乎相同。

Python 2.4 之后在 `string` 模块中新增加了另一种字符串替换的形式，这种形式使用一种新的语法作为替换语法。建立这种形式能够给程序的用户提供一个更有意义的格式：

```
import string
person = {"name": "James", "camera": "nikon", "handedness": "lefty",
          "baseball_team": "angels", "instrument": "guitar"}
person["height"] = 1.6
person["weight"] = 80
t = string.Template("$name is $height m high and $weight kilos")
```

```
print(t.substitute(person))
```

除了不能再控制格式信息，以上代码生成的输出和第一种方式没有好坏之分：

```
print t.substitute(person)
James is 1.6 m high and 80 kilos
```

当要求用户描述他们想要从一组数据中得到的信息时，可以考虑使用这个特性。对于让用户指定他们想要的信息而不需要您重写程序的情况，这个特性可以作为一种简单的支持方式。只需要让用户指定模板，并将他们给出的字符串提供给 `string.Template` 类，就可以创建一个执行想要的替换的模板对象。

## 9.6 重要模块

从第7章开始，我们看到了如何使用模块给 Python 添加功能。在第8章中，学到了如何通过模块同操作系统及其文件进行交互，这些模块为系统使用 `os` 模块提供了接口。

本节将介绍其他一些常用模块的例子，这将帮助您开始建立自己的程序。

### 9.6.1 getopt——从命令行中得到选项

在 UNIX 系统中指定一个程序的运行方式时，最常用的方法是在程序的命令行中加入参数。即使程序不从命令行运行，而是使用 `fork` 和 `exec` 运行(本章的后面详述)，当它被调用时也同样建立一个命令行。这种方法实现了一个控制程序行为的通用方法。

例如，您可能已经看到了能够在运行很多程序时可以使它们提供关于其运行方式的基本信息。在 Python 中可以通过使用 `-h` 来实现这种目的：

```
$ python -h
usage: python30 [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-c cmd : program passed in as string (terminates option list)
-d      : debug output from parser (also PYTHONDEBUG=x)
-E      : ignore environment variables (such as PYTHONPATH)
[ etc. ]
```

以前，在不同的 UNIX 平台上采用不同的方式来指定这些选项，而这在很大程度上导致了大部分项目采用了两种选项形式：简易形式(例如，生成帮助消息的 Python 选项)和长形式(例如，用于帮助的 `--help`)。

使用这两种选项是有意义的。理想情况下，我们希望同时提供一个简易的和长的命令形式，并且允许随意地指定其中的一个。所以如果想要编写一个具有用户可以指定的配置文件的程序，可能要为有经验的用户提供像 `-c` 这样的简易选项，同时也提供一个长的选项，比如 `--config-file`。在任何一种情况下，都希望它们在程序中具有相同的功能以节省时间，同时允许用户自由地使用这些选项。

`getopt` 模块为了使这个标准的约定易用提供了两个函数：`getopt.getopt` 和 `getopt.gnu-`

`getopt`。它们基本上是相同的。基本的 `getopt` 一直运行到遇到第一个非选项，剩下的其他部分都不检查。

为了让 `getopt` 更有用，必须知道想让用户使用什么选项。通常情况下，编写的程序最少要为用户提供关于如何运行程序的信息，就像 Python 通过 `-h` 选项打印信息一样。

另外，具有一个配置文件通常非常有用。将这些想法作为出发点，可以开始编写新程序，使得 `-h` 和 `--help` 都能产生关于程序如何使用的最少信息，并且使用 `-c` 或 `--config-file=file` 能够指定与默认配置不同的配置文件：

```
import sys
import getopt
# Remember, the first thing in the sys.argv list is the name of the command
# You don't need that.
cmdline_params = sys.argv[1:]

opts, args = getopt.getopt(cmdline_params, 'hc:', ['help', 'config='])

for option, parameter in opts:

    if option == '-h' or option == '--help':
        print("This program can be run with either -h or --help for this
message,")
        print("or with -c or --config=<file> to specify a different
configuration file")

    if option in ('-c', '--config'): # this means the same as the above
        print("Using configuration file %s" % parameter)
```

当使用长选项并要求一个参数(像前面例子中的 `--config`)时，必须使用等号连接选项和参数的值。然而，当使用简易选项时，可以用一个或多个空格或制表符分隔选项和它对应的值。这个区别是仿照一直沿用到现在的早期 UNIX 机器的选项的行为。它们能够延续是因为很多人希望那样做。您又能有什么办法？

前面的代码片段如果在一个使用参数 `-c test -h -config=secondtest` 的程序中运行，将产生下面的输出：

```
[('-c', 'test'), ('-h', ''), ('--config', 'secondtest')] []
Using configuration file test
This program can be run with either -h or --help for this message,
or with -c or --config=<file> to specify a different configuration file

Using configuration file secondtest
```

注意配置文件的第二个实例是默认接受的，当它出现时，重新访问设置 `config` 文件的相同代码，以使得能够使用第二个实例。

第二个列表(`args`数据)是一个空的列表，这是因为命令行中提供给程序的所有选项都是有效的选项，或者选项的有效参数。如果在选项中间插入其他的字符串，标准的 `getopt` 会

有不同的行为。如果将参数替换为 `-c test useless_information_here -h -- config=secondtest`, 输出内容将会很少, 而 `args` 数组中的内容将会很多。

```
[('-c', 'test')] ['useless_information_here', '-h', '--config=secondtest']
Using configuration file test
```

`gnu-getopt` 能够混合和匹配命令行, 使得非选项可以出现在选项中的任何地方, 后面的选项会得到解析, 而不是就此停止:

```
opts, args = getopt.gnu_getopt(cmdline_params, 'hc:', ['help', 'config='])

for option, parameter in opts:

    if option == '-h' or option == '--help':
        print("This program can be run with either -h or --help for this
              message,")
        print("or with -c or --config=<file> to specify a different
              configuration file")

    if option in ('-c', '--config'): # this means the same as the above
        print("Using configuration file %s" % parameter)
```

需要注意的重要一点是, 如果使用一些与选项的标准不匹配的选项(通过 `-` 或 `+` 开始, 或者放在具有参数的选项的后面), 这两种方式的行为是不同的。使用选项 `-c test useless_information_here -h -- config=secondtest`, `gnu_getopt` 函数产生如下的输出, 不符合标准的选项是命令行留在 `args` 数组中的唯一部分:

```
[('-c', 'test'), ('-h', ''), ('--config', 'secondtest')]
['useless_information_here']
Using configuration file test
This program can be run with either -h or --help for this message,
or with -c or --config=<file> to specify a different configuration file

Using configuration file secondtest
```

### 9.6.2 使用一个以上的进程

在 UNIX 和类 UNIX 的操作系统中, 执行某种类型的子任务的主要方式是运行一个新的程序以建立一个新的进程。在 UNIX 系统中, 这是使用一个系统调用完成的。在 Python 中, 可以使用 `os.fork` 进行这种系统调用。这实际上告诉计算机复制关于当前运行的程序的一切信息到一个新创建的单独程序中, 这个新程序同原来的程序几乎是完全相同的。唯一不同之处是, 在新创建的进程(子进程)中 `os.fork` 的返回值是 0, 并且这个返回值是新创建的进程在原始进程(父进程)中的进程 ID(PID)。这可能很难理解, 真正理解它的唯一方法是使用几次, 并且阅读一些在线的关于 `fork` 和 `exec` 的其他资料(或者请教身边的 UNIX 专家)。

基于这个重要的区别, 父进程和子进程可以执行不同的功能。父进程能够在子进程运

行中等待一个事件，反之亦然。其实现代码是简单并且常用的，但是它只能工作在 UNIX 和类 UNIX 的系统上。

```
import os
pid = os.fork()
if pid == 0: # This is the child
    print("this is the child")
else:
    print("the child is pid %d" % pid)
```

在调用 `os.fork` 之后最常做的一件事情是立即调用 `os.execl` 以运行其他的程序。`os.execl` 是一个用新程序代替正在运行的程序的指令，所以它使得调用程序中止，并且一个新的程序出现在它的位置(提示：UNIX 系统使用 `fork` 和 `exec` 方法运行所有的程序)：

```
import os
pid = os.fork()
# fork and exec together
print("second test")
if pid == 0: # This is the child
    print("this is the child")
    print("I'm going to exec another program now")
    os.execl('/bin/cat', 'cat', '/etc/motd')
else:
    print("the child is pid %d" % pid)
os.wait()
```

`os.wait` 函数通知 Python 让父进程什么都不做，一直等到子过程返回。了解这个函数的工作原理是很有用的，因为它只在像 Linux 这样的 UNIX 和类 UNIX 的平台上能够正常工作。Windows 也有启动新进程的机制。

为了使启动新程序这样普遍的任务变得更容易，Python 提供了单独的一族函数，合并了在类 UNIX 系统上的 `os.fork` 和 `os.exec`，并且能够在 Windows 上完成类似的工作。当只想要启动一个新程序时，可以使用 `os.spawn` 函数族。因为这些函数的名称很相似，所以被称作是一个族，但是每个函数的行为都稍有不同。

在类 UNIX 的系统上，`os.spawn` 族包括 `spawnl`、`spawnle`、`spawnlp`、`spawnlpe`、`spawnv`、`spawnve`、`spawnvp` 和 `spawnvpe`。在 Windows 系统上，`spawn` 族只包括 `spawnl`、`spawnle`、`spawnv` 和 `spawnve`。

在每一种情况下，词 `spawn` 后面的字母表示一些特定的情况。`v` 表示传递一个列表(实际上 `v` 代表 `vector`)作为参数。这允许在不同的实例中运行差别很大的命令，而不需要对程序进行任何修改。`l` 变量只需要一个简单的参数列表。

使用 `e` 时，需要传递一个包括名称和值的字典作为新创建的程序的环境，而不是使用当前的环境。

`p` 变体使用环境字典中 `PATH` 键的值来寻找程序。`p` 变体只能用在类 UNIX 平台上。这意味着在 Windows 中，程序必须有一个完全限定的路径，使其可以用于 `os.spawn` 调用，否

则必须自己搜索路径:

```
import os, sys
if sys.platform == 'win32':
    print("Running on a windows platform")
    command = "C:\\winnt\\system32\\cmd.exe"
    params = []

if sys.platform == 'linux2':
    print("Running on a Linux system, identified by %s" % sys.platform)
    command = '/bin/uname'
    params = ['uname', '-a']

print("Running %s" % command)
os.spawnv(os.P_WAIT, command, params)
```

当然, 这个例子只能在有限的系统上工作。如果使用的是其他 Unix 系统, 如 Solaris、Mac OS X、AIX 等, 可以使用自己计算机上的 `sys.platform` 的内容替换 `linux2`。

当工作时, 可以等待进程返回(也就是, 直到进程完成并退出), 或者可以告诉 Python 您希望让程序自己运行, 并且您会确认它成功地完成。这是由 `os.P_` 值族完成的。当 `os.spawn` 函数返回时, 将依赖于设置的值给出不同的行为。

如果只需要一个新命令的最基本的调用, 有时最简单的方式是使用 `os.system` 函数。如果正在运行一个程序并且只想等待它结束, 可以很简单地使用这个函数:

```
# Now system
if sys.platform == 'win32':
    print("Running on a windows platform")
    command = "cmd.exe"

if sys.platform == 'linux2':
    print("Running Linux")
    command = "uname -a"

os.system(command)
```

因为这段代码使用了操作系统提供的工具来查找想要运行的程序, 并且用户通常希望使用这些工具, 所以更简单, 并且它默认等待子进程结束。

### 9.6.3 线程——在相同的进程中完成多个工作

使用 `fork` 或 `spawn` 创建一个新的进程有时可能需要付出很大的代价, 但是不能提供足够的好处。具体来说, 当一个程序增长得很大时, `fork` 必须复制程序的全部信息到一个新程序中, 并且系统必须有足够的资源来处理这种操作。`fork` 的另一个不足之处是, 当有时需要程序同时做很多工作时, 一些工作可能需要等待其他工作处理完成。当这种情况发生时, 可能想要所有不同的组件将它们需要传递到程序的其他部分。

当使用多个进程时, 这样做就变得非常困难。因为子进程最初是使用父进程的数据创

建的，所以这些进程间共享很多信息。然而，它们是完全不同的实体。因此，使两个进程协同工作需要使用很多技巧。

所以，为了应付一些不适用于进程的复杂情况，用到了线程的概念。

程序执行时，很多合作的线程可以同时存在于相同的程序中。每一个线程内部都有不同的对象，具有不同的状态，但是它们都可以通信，同时也可以彼此半独立地运行。

这意味着在很多情况下，使用线程比使用单独的进程要方便得多。注意下面的例子使用了子类，第10章中将更详细地介绍相关概念。要想了解这段代码是如何运行的，试着使用一个相当大的参数运行它，例如2 000 000：

```
import math
from threading import Thread
import time

class SquareRootCalculator:

    """This class spawns a separate thread to calculate a bunch of square
    roots, and checks in it once a second until it finishes."""

    def __init__(self, target):
        """Turn on the calculator thread and, while waiting for it to
        finish, periodically monitor its progress."""
        self.results = []
        counter = self.CalculatorThread(self, target)
        print("Turning on the calculator thread...")
        counter.start()
        while len(self.results) < target:
            print("%d square roots calculated so far." % len(self.results))
            time.sleep(1)
        print("Calculated %s square root(s); the last one is sqrt(%d)=%f" %
              (target, len(self.results), self.results[-1]))

    class CalculatorThread(Thread):
        """A separate thread which actually does the calculations."""

        def __init__(self, controller, target):
            """Set up this thread, including making it a daemon thread
            so that the script can end without waiting for this thread to
            finish."""
            Thread.__init__(self)
            self.controller = controller
            self.target = target
            self.setDaemon(True)

        def run(self):
            """Calculate square roots for all numbers between 1 and the target,
            inclusive."""
            for i in range(1, self.target+1):
                self.controller.results.append(math.sqrt(i))
```

```
if __name__ == '__main__':
    import sys
    limit = None
    if len(sys.argv) > 1:
        limit = sys.argv[1]
        try:
            limit = int(limit)
        except ValueError:
            print("Usage: %s [number of square roots to calculate]"
                  % sys.argv[0])
    SquareRootCalculator(limit)
```

对于很多情形,例如网络服务器(见第16章)或者图形用户界面(见第13章),线程具有更重要的意义,因为它们只需要程序员做很少的工作,并且需要的系统资源很少。

注意,单独的线程能够很容易地访问彼此的名称和数据。这样很容易跟踪不同线程的运行,从而提供了很大的便利。

## 9.7 本章小结

本章介绍了 Python 提供的一些可利用的函数和模块。这些特性建立在您已经学习的内容之上,大多数特性将在本书剩下的章节中详述。

本章还介绍了如何使用一些基本特性,利用这些特性可以进行通常所说的函数式编程,Python 中通过函数 `lambda` 和 `map` 提供这种特性。`lambda` 允许编写一个简单的函数而不必在其他地方声明它。编写和运行这些函数不需要绑定一个名称,因此它们称为匿名函数。`map` 在列表上操作,当用于一个简单列表时,将针对列表从头到尾的每个元素运行一个函数。当提供给 `map` 多个列表,或者列表中包含列表时,它具有更复杂的行为。

### 本章要点:

- 列表解析是指在解除对序列的引用的方括号中运行有限量代码(例如,简单循环)的能力,只有满足中括号中标准的那些元素被返回。通过列表解析可以简单快速地访问一个序列的指定成员。
- `range` 操作能够提供起始于任意数值并且终止于任意数值的数值列表,所以可以用来生成一个通常用于 `for` 循环的迭代器。
- 除了简单的字符串替换以外,还可以提供一个具有格式说明符的字符串,这些格式说明符通过使用一种特殊的语法引用字典中的键名。这个形式允许继续使用格式说明符选项,例如,想要为替换保留多少个空格或者应该使用多少位小数。
- Python 2.4 中增加的 `string.Template` 模块中为这种简单的基于键名字符串格式化提供了一种替换形式。它提供了略微简单一点的格式,在允许用户指定模板时更适合(或者至少容易解释)。生成套信函是使用这个方法的一个例子。
- `getopt` 能够指定命令行选项,从而提供给用户选项来决定程序在运行时的行为。

- 您现在知道了如何在需要时建立多个进程，以及如何为需要并行处理很多工作的复杂程序创建线程。第 13 章和第 16 章将更详细地介绍如何使用线程。
- 这里介绍的特性和模块能够帮助您理解 Python 可以在不同的方向上扩展和使用，并且使用这些扩展是很简单的。在第 10 章将把已经介绍的大多数概念应用到一个示例程序中。

## 9.8 习题

第 9 章是一个不同特性的集合。在本章中，最好的习题是测试所有的示例代码，观察其产生的输出，并思索如何使用这里介绍的不同思想来解决您想要解决的或过去希望解决的问题。



# 第 10 章

## 创建模块

如第 7 章所述，模块提供了一种在应用程序之间共享 Python 代码的便捷方式。模块是一种非常简单的结构。在 Python 中，模块只是一个由 Python 语句组成的文件。在模块中可以定义函数和类，还可以包含简单的可执行代码，这些可执行代码不在函数或者类的内部。而最好的一点是，模块可以包含说明如何使用模块代码的文档。

Python 提供了一个包括几百个模块的库，您可以在脚本中调用它们，也可以创建自己的模块，在脚本间共享代码。本章将逐步演示如何创建模块。

本章将介绍：

- 研究模块的内部机制。
- 创建一个仅包含函数的模块。
- 在模块中定义类。
- 通过子类扩展类。
- 定义异常来报告错误状态。
- 为模块建立文档。
- 测试模块。
- 将模块作为程序运行。
- 安装模块。

首先研究什么是模块以及它们的工作方式。

### 10.1 研究模块

模块只是一个 Python 源文件。它可以包含变量、类、函数和 Python 脚本中可用到的其他任何元素。

通过使用 `dir` 函数可以更好地理解模块。给 `dir` 函数传递某个 Python 元素(例如模块)的名称，它将列出该元素的所有属性。例如，要查看 `__builtins__` 的属性，包括内置函数、类和变量，可以使用下面的语句：

```
dir(__builtins__)
```

这将得到类似于下面的输出：

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError',
'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FloatingPointError',
'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning',
'IndentationError', 'IndexError', 'KeyError',
'KeyboardInterrupt', 'LookupError', 'MemoryError',
'NameError', 'None', 'NotImplemented', 'NotImplementedError',
'OSError', 'OverflowError', 'PendingDeprecationWarning',
'ReferenceError', 'RuntimeError', 'RuntimeWarning', 'StopIteration',
'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit',
'TabError', 'True', 'TypeError', 'UnboundLocalError',
'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError',
'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
'ValueError', 'Warning', 'WindowsError', 'ZeroDivisionError',
'__build_class__', '__debug__', '__doc__', '__import__',
'__name__', '__package__', 'abs', 'all', 'any', 'ascii', 'bin',
'bool', 'bytearray', 'bytes', 'chr', 'classmethod', 'compile',
'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod',
'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format',
'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex',
'id', 'input', 'int', 'isinstance', 'issubclass', 'iter',
'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min',
'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit',
'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted',
'staticmethod', 'str', 'sum', 'super', 'tuple', 'type',
'vars', 'zip']
```

对于像 Python 这样具有许多特性的语言，它的内置元素非常少。同样可以对导入的模块运行 `dir` 函数。例如：

```
>>> import sys
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__',
'__stderr__', '__stdin__', '__stdout__', '_clear_type_cache',
'_current_frames', '_getframe', 'api_version', 'argv',
'builtin_module_names', 'byteorder', 'call_tracing', 'callstats',
'copyright', 'displayhook', 'dllhandle', 'dont_write_bytecode',
'exc_info', 'excepthook', 'exec_prefix', 'executable', 'exit', 'flags',
'float_info', 'getcheckinterval', 'getdefaultencoding',
'getfilesystemencoding', 'getprofile', 'getrecursionlimit', 'getrefcount',
'getsizeof', 'gettrace', 'getwindowsversion', 'hexversion',
'intern', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path',
'path_hooks', 'path_importer_cache', 'platform', 'prefix',
'setcheckinterval', 'setfilesystemencoding', 'setprofile',
'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout',
'subversion',
'version', 'version_info', 'warnoptions', 'winver']
```

使用 `dir` 有助于研究模块，包括自己创建的模块。

### 10.1.1 导入模块

在使用一个模块之前，需要先导入它。标准的导入语法如下：

```
import module
```

可以对 Python 自带的或您创建的模块使用这种语法。也可以使用如下所示的另一种语法：

```
from module import item
```

这种语法能够仅导入所需要的一个类或函数。

如果一个模块被修改了，可以使用 `imp.reload` 函数重新加载该模块的新定义。语法如下：

```
import module
import imp
imp.reload(module)
```

使用希望重新加载的模块替换 `module`。

注意：

使用 `imp.reload` 时，总是要使用圆括号；使用 `import` 时，不使用圆括号。

### 10.1.2 查找模块

为了导入一个模块，Python 解释器首先需要找到它。对于一个模块，Python 解释器首先查找一个称为 `module.py` 的文件，其中 `module` 是传递给 `import` 语句的模块的名称。当找到该模块时，Python 解释器将它编译成一个 `.pyc` 文件。当再次导入模块时，Python 解释器可以加载编译好的模块，加速 Python 脚本的运行。

在脚本中编写了一个 `import` 语句时，Python 解释器必须能够找到该模块。关键的问题是，Python 解释器只查找一定数量的目录。如果 Python 解释器不能找到您输入的名称，它将显示一个错误，如下面的例子所示：

```
>>> import foo
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    import foo
ImportError: No module named foo
```

Python 解释器查找模块搜索路径中的那部分目录。这些目录在 `sys` 模块的 `sys.path` 变量中列出。

要列出 Python 解释器查找模块的位置，可以在 Python 解释器中打印 `sys.path` 变量的值。

例如：

```
>>> import sys
>>> print(sys.path)
['C:\\Python31\\Lib\\idlelib', 'C:\\Windows\\system32\\python3.zip',
'C:\\Python31\\DLLs', 'C:\\Python31\\lib',
'C:\\Python31\\lib\\plat-win', 'C:\\Python31',
'C:\\Python31\\lib\\site-packages']
```

### 10.1.3 理解模块

因为 Python 是一个开源包，所以可以得到 Python 解释器和所有模块的源代码。实际上，即使在 Python 的二进制发布版本中，也可以找到用 Python 编写的模块的源代码。

首先在 `sys.path` 变量列出的所有目录中查找以 `.py` 结尾的文件。这些文件就是 Python 模块。一些模块只包含函数，而另外一些则包含类和函数。例如，Python 3.0 版本中的 `Parser` 模块定义了一个类，如下所示：

```
"""A parser of RFC 2822 and MIME email messages."""
__all__ = ['Parser', 'HeaderParser']
import warnings
from io import StringIO
from email.feedparser import FeedParser
from email.message import Message
class Parser:
    def __init__(self, *args, **kws):
        """Parser of RFC 2822 and MIME email messages.
        Creates an in-memory object tree representing the email message, which
        can then be manipulated and turned over to a Generator to return the
        textual representation of the message.
        The string must be formatted as a block of RFC 2822 headers and header
        continuation lines, optionally preceded by a `Unix-from' header. The
        header block is terminated either by the end of the string or by a
        blank line.
        _class is the class to instantiate for new message objects when they
        must be created. This class must have a constructor that can take
        zero arguments. Default is Message.Message.
        """
        if len(args) >= 1:
            if '_class' in kws:
                raise TypeError("Multiple values for keyword arg '_class'")
            kws['_class'] = args[0]
        if len(args) == 2:
            if 'strict' in kws:
                raise TypeError("Multiple values for keyword arg 'strict'")
            kws['strict'] = args[1]
        if len(args) > 2:
            raise TypeError('Too many arguments')
        if '_class' in kws:
            self._class = kws['_class']
```

```

        del kws['_class']
    else:
        self._class = Message
    if 'strict' in kws:
        warnings.warn("'strict' argument is deprecated (and ignored)",
                        DeprecationWarning, 2)
        del kws['strict']
    if kws:
        raise TypeError('Unexpected keyword arguments')
def parse(self, fp, headersonly=False):
    """Create a message structure from the data in a file.
    Reads all the data from the file and returns the root of the message
    structure. Optional headersonly is a flag specifying whether to stop
    parsing after reading the headers or not. The default is False,
    meaning it parses the entire contents of the file.
    """
    feedparser = FeedParser(self._class)
    if headersonly:
        feedparser._set_headersonly()
    while True:
        data = fp.read(8192)
        if not data:
            break
        feedparser.feed(data)
    return feedparser.close()

def parsestr(self, text, headersonly=False):
    """Create a message structure from a string.

    Returns the root of the message structure. Optional headersonly is a
    flag specifying whether to stop parsing after reading the headers or
    not. The default is False, meaning it parses the entire contents of
    the file.
    """
    return self.parse(StringIO(text), headersonly=headersonly)
class HeaderParser(Parser):
    def parse(self, fp, headersonly=True):
        return Parser.parse(self, fp, True)

    def parsestr(self, text, headersonly=True):
        return Parser.parsestr(self, text, True)

```

这个小模块主要由指导用户如何使用该模块的文档组成。文档很重要。

当浏览标准的 Python 模块时，可以理解模块是如何组成的。这也有助于理解如何创建自己的模块。

## 10.2 创建模块和包

创建一个模块要比想象中容易。模块只是一个 Python 源文件。实际上，任何时候您创

建一个 Python 文件，就已经在创建模块，只是自己不知道而已。

下面的例子将帮助您开始创建模块。

### 试一试

### 创建一个具有函数的模块

输入下面的 Python 代码，并将文件命名为 `food.py`：

```
def favoriteFood():  
    print("the only food worth eating is an omelet.")
```

这就是一个模块。接下来可以使用 Python 解释器导入该模块。例如：

```
>>> import food  
>>> dir(food)  
['__builtins__', '__doc__', '__file__', '__name__', '__package__',  
'favoriteFood']
```

#### 示例说明

Python 使用非常简单的方法定义模块。可以使用任何 Python 源文件作为一个模块，如这个简单的例子所示。`dir` 函数列出模块中定义的项，包括函数 `favoriteFood`。

一旦被导入，可以通过下面的命令执行模块中的代码：

```
>>> food.favoriteFood()  
The only food worth eating is an omelet.
```

如果不使用模块名前缀(这里是 `food`)，将出现一个错误，如下面的例子所示：

```
>>> favoriteFood()  
Traceback (most recent call last):  
  File "<pyshell#22>", line 1, in <module>  
    favoriteFood()  
NameError: name 'favoriteFood' is not defined
```

使用另一种语法导入模块能够解决这个问题：

```
>>> from food import favoriteFood  
>>> favoriteFood()  
The only food worth eating is an omelet.  
>>>
```

祝贺您！您现在已经是一个模块的创建者了。

## 10.3 使用类

大多数模块定义了一个相关的函数或类的集合。如第 6 章所述，类保存数据和操作数

据的方法。Python 比大多数编程语言(如 Java、C++或 C#)宽松,原因在于 Python 打破了其他语言的规则。例如,Python 默认能够访问类中的数据。这违反了一些面向对象编程的概念,但是好处在于:Python 首要的目标是实用。

### 10.3.1 定义面向对象编程

计算机程序员不断地争论什么是真正的面向对象编程(Object-Oriented Programming, OOP)。大多数专家同意如下的三个概念:

- 封装
- 继承
- 多态

封装的思想是,一个类能够隐藏执行某个任务所需的内部细节和数据。类保存需要的数据,并且在正常情况下不支持查看数据。而且,类提供了一些对数据进行操作的方法。这些方法能够隐藏内部细节,例如网络协议和磁盘存取操作等。封装是一种简化程序的技术。在创建程序的每一步,可以编写集中处理单个任务的代码。封装隐藏了复杂性。

继承意味着一个类可以继承或访问父类中定义的数据和方法。这正符合对一个问题领域进行分类的常识。例如,矩形和圆都是形状。这里,基类可以是 Shapes。Rectangle 类可以从 Shapes 继承, Circle 类也同样可以。继承使您可以将 Rectangle 类和 Circle 类对象看做 Shapes 类的子类和成员,这意味着可以在基类中编写通用的代码,而在子类中编写具体的代码(术语儿子、子类和类的成员是类似的,在这里可以交换使用)。对于大多数情况,基类应该是一般的,而子类应该是特殊的。继承通常被称为特化。

多态的意思是子类可以重写方法来完成更特殊的行为。例如,矩形和圆都是形状。可以定义一个可用于所有形状的共同操作的集合,例如 move 和 draw。然而, Circle 的 draw 方法显然和 Rectangle 的 draw 方法是不同的。多态使您可以将方法命名为 draw,然后调用这些方法,就好像 Circle 和 Rectangle 都是 Shapes,当然至少在这个例子中它们都是的。

### 10.3.2 创建类

就像第 6 章中描述的那样,创建一个类是很容易的(事实上,在 Python 中做大多数事情都很容易)。下面的例子演示了一个描述进餐的简单的类。

#### 试一试

#### 创建一个 Meal 类

下面的代码定义了 Meal 类。完整的源文件参见 10.5 节。

```
class Meal:
    '''Holds the food and drink used in a meal.
    In true object-oriented tradition, this class
    includes setter methods for the food and drink.

    Call printIt to pretty-print the values.
    '''
```

```

def __init__(self, food='omelet', drink='coffee'):
    '''Initialize to default values.'''
    self.name = 'generic meal'
    self.food = food
    self.drink = drink

def printIt(self, prefix=''):
    '''Print the data nicely.'''
    print(prefix, 'A fine', self.name, 'with', self.food, 'and', self.drink)

# Setter for the food.
def setFood(self, food='omelet'):
    self.food = food

# Setter for the drink.
def setDrink(self, drink='coffee'):
    self.drink = drink

# Setter for the name.
def setName(self, name=''):
    self.name = name

```

### 示例说明

Meal 类的每个实例保存 3 个数据值：Meal 类的名称、食物和饮料。Meal 类默认设置的名称为 generic meal，饮料为 coffee，食物为 omelet。

就像杜松子酒和滋补品一样，煎蛋卷不再只用作早餐。

`__init__` 方法为 Meal 初始化数据。`printIt` 方法以友好的方式打印内部数据。最后，为了支持开发人员使用更严格的编程语言，Meal 类定义了一组称为 setter 的方法。这些 setter 方法，例如 `setFood` 和 `setDrink`，可以设置类的数据。

### 注意：

因为在 Python 中可以直接设置数据，所以这些方法不是必需的。

关于类的更多信息可参见第 6 章。

## 10.3.3 扩展已有的类

在定义一个类之后，可以通过定义子类来扩展它。例如，可以创建一个描述一天中第一顿饭的 `Breakfast` 类：

```

class Breakfast(Meal):
    '''Holds the food and drink for breakfast.'''

    def __init__(self):
        '''Initialize with an omelet and coffee.'''
        Meal.__init__(self, 'omelet', 'coffee')
        self.setName('breakfast')

```

Breakfast 类通过如下所示的定义扩展 Meal 类:

```
class Breakfast(Meal):
```

另一个子类很自然应该是 Lunch:

```
class Lunch(Meal):
    '''Holds the food and drink for lunch.'''

    def __init__(self):
        '''Initialize with a sandwich and a gin and tonic.'''
        Meal.__init__(self, 'sandwich', 'gin and tonic')
        self.setName('midday meal')

    # Override setFood().
    def setFood(self, food='sandwich'):
        if food != 'sandwich' and food != 'omelet':
            raise AngryChefException
        Meal.setFood(self, food)
```

通过 Lunch 类, 可以看到 setter 方法的一些应用。在 Lunch 类中, setFood 方法只允许食物有两个值: sandwich 和 omelet。使用其他的值, 厨师会生气。

Dinner 类也重写了一个方法, 在这里是 printIt 方法:

```
class Dinner(Meal):
    '''Holds the food and drink for dinner.'''

    def __init__(self):
        '''Initialize with steak and merlot.'''
        Meal.__init__(self, 'steak', 'merlot')
        self.setName('dinner')

    def printIt(self, prefix=''):
        '''Print even more nicely.'''
        print(prefix, 'A gourmet', self.name, 'with', self.food, 'and', self.drink)
```

通常将所有这些类放入一个模块中。10.5 节给出了一个完整的模块的例子。

## 10.4 完成模块

在定义了模块的类和函数之后, 下一步是完成模块, 以使它更适合 Python 用户和 Python 解释器的约定。

完成模块包括很多工作, 但是至少需要完成以下操作:

- 定义应用于模块的错误和异常。
- 定义模块中要输出的项。这定义了模块的公共 API。

- 为模块编写文档。
- 测试模块。
- 在模块作为程序执行的情况下提供一个回退函数。

下面几节将描述如何完成模块。

### 10.4.1 定义模块特定的错误

Python 定义了一些标准的异常类，例如 `IOError` 和 `NotImplementedError`。如果这些类合适，一定要使用它们。否则，需要针对使用模块时可能发生的特定问题定义异常。例如，一个网络模块可能需要定义一组与网络故障相关的异常。

对于示例模块中使用的与食物相关的主题，可以定义一个 `AngryChefException`。为了使其更通用，并且允许其他模块重用它，将 `AngryChefException` 作为一个更一般的 `SensitiveArtistException` 类的子类来定义，这个父类表示由敏感的艺术派人士引起的问题。

在大多数情况下，异常类不需要定义任何方法或初始化任何数据。`Exception` 基类所提供的已经足够了。对于大多数异常，异常的存在已经指出了问题。

注意：

并非总是如此。例如，一个 XML 解析异常也许应该包括发生错误的行号，以及对错误的描述。

可以定义 `meal` 模块的异常如下：

```
class SensitiveArtistException(Exception):  
    pass  
  
class AngryChefException(SensitiveArtistException):  
    pass
```

当然这只是一个例子。在您的模块中，应该根据需要定义异常类。除了异常，应该认真决定模块的导出内容。

### 10.4.2 选择导出哪些内容

使用 `from` 形式导入模块时，可以指定导入模块中的哪一项。例如，下面的语句从 `meal` 模块导入 `AngryChefException`。

```
from meal import AngryChefException
```

要从模块导入所有的公有项，可以使用下面的格式：

```
from module_name import *
```

例如：

```
from meal import *
```

星号(\*)告诉 Python 解释器导入模块的全部公有方法。什么是公有的？作为模块的设计者，您可以选择定义哪些项作为能够导出的公有方法。

Python 解释器使用两种方法确定哪些项应该是公有的：

- 如果模块中定义了变量 `__all__`，解释器使用 `__all__` 确定哪些项是公有项。
- 如果模块中没有定义变量 `__all__`，解释器导入除了名称以下划线(`_`)开头的项目之外的所有项，所以 `printIt` 被认为是公有的，而 `_printIt` 则不会。

关于模块和 `import` 语句的更多信息，请参考第 7 章。

作为最佳实践，总是应该在模块中定义 `__all__`。这样可以显式控制其他 Python 脚本可以导入的项。为了完成这项工作，可以简单地创建一个文档字符串序列，每个字符串分别对应想要从模块导出的每个项的名称。例如，在 `meal` 模块中，可以用下面的方式定义 `__all__`：

```
__all__ = ['Meal', 'AngryChefException', 'makeBreakfast',  
          'makeLunch', 'makeDinner', 'Breakfast', 'Lunch', 'Dinner']
```

这个序列中的每个名称都是一个需从模块导出的类或函数。

选择要导出的内容很重要。当创建一个模块时，就创建了一个执行一些有用的功能的 API。从模块中导出的 API 定义了模块的用户能实现哪些功能。您希望导出模块的用户能够得到完成工作所需的全部功能，但是不需要导出一切。可能因为一些原因希望排除一些项，例如：

- 可能要修改的项应该保持私有性，直到确定这些项的 API。这允许您自由改变模块的内部实现而不影响模块用户。
- 模块时常可以有意隐藏复杂的代码。例如，一个 `e-mail` 模块可以隐藏 SMTP、POP3 和 IMAP 网络电子邮件协议的细节。`e-mail` 模块可以给出一个 API，使用户能发送邮件，看到可用邮件，下载邮件内容，等等。

注意：

隐藏代码实现的细节被称为封装。用这句话答复朋友：“完成你要求的修改将违反封装原则”，他一定会对您的专业程度留下深刻的印象。

总是应该明确地定义希望从模块中导出什么。也应该总是为模块建立文档。

### 10.4.3 为模块建立文档

为模块建立文档非常重要。否则，没有人(甚至程序员自己)会知道模块能做什么事情。想想 6 个月以后的情况。还能记得加入模块中的每部分内容吗？大概不能。解决的方法很简单：为模块建立文档。

Python 定义了一些为模块建立文档的简单约定。遵循这些约定后，模块可以使用户以标准的方式查看模块的文档。最基本的要求是，对于要写入文档的每一项，都要写一个文档字符串来描述这一项。在三引号内包含这个文档字符串，并将它放到项的后面，紧挨着它描述的项。

例如，为一个方法或函数写文档，可以使用下面的代码作为向导：

```
def makeLunch():
    ''' Creates a Breakfast object. '''
    return Lunch()
```

三引号中的行表明这是一个文档。函数后面的文档用 `def` 语句定义。同样可以为类建立文档：

```
class Meal:
    '''Holds the food and drink used in a meal.
    In true object-oriented tradition, this class
    includes setter methods for the food and drink.

    Call printIt to pretty-print the values.
    '''
```

将文档放到 `class` 语句的后面一行。

异常也是类。同样可以为它们建立文档：

```
class SensitiveArtistException(Exception):
    '''Exception raised by an overly-sensitive artist.

    Base class for artistic types.'''
    Pass
```

注意，即使类中没有加入新的功能，也应该描述每个异常或类的用途。

另外，要为模块本身建立文档。方法是，在模块开始位置使用特殊的三引号文档字符串，如下所示：

```
"""
Module for making meals in Python.

Import this module and then call
makeBreakfast(), makeDinner() or makeLunch().

"""
```

将该文档放到包含模块的文本文件的第一行。模块以概述模块用途的一行文本开始。使用前面所示的空白行，将这一行与文档的其余行分开。Python 的 `help` 函数将取出一行概述文本并对其做特殊处理(关于如何调用 `help` 函数的更多细节，见下面的“试一试”)。

通常，为每个类、方法或函数编写一至两行的文档就足够了。一般来说，文档应该告诉用户如下事情：

- 如何调用函数或方法，包括必要的参数和将要返回的数据类型。描述默认的参数值。
- 给定类的设计目的和用途。包括如何使用类的对象。
- 调用函数或方法的必备条件。

- 类的运行带来的副作用或者导致系统的哪些方面发生变化。例如，一个删除磁盘上所有文件的方法应该在文档中说明它的功能。
- 可能发生的异常和导致异常发生的原因。

注意：

一些人文档写得过多。过多的文档没有帮助，但是不要将此作为不写文档的借口。过多的文档也比一点都没有要好。

根据自己的兴趣可以确定一条经验法则。询问自己希望在别人的模块中看到什么样的文档，并用这个标准写文档。

可以使用 `help` 函数浏览编写的文档，如下面的例子所示。

### 试一试

### 浏览模块的文档

在交互模式下启动 Python 解释器，然后像下面这样运行 `import` 和 `help` 命令：

```
>>> import meal
>>> help(meal)
Help on module meal:
NAME
    meal - Module for making meals in Python.
FILE
    c:\python30\meal.py
DESCRIPTION
    Import this module and then call
    makeBreakfast(), makeDinner() or makeLunch().
CLASSES
    builtins.object
        Meal
            Breakfast
            Dinner
            Lunch
    SensitiveArtistException(builtins.Exception)
        AngryChefException

class AngryChefException(SensitiveArtistException)
| Exception that indicates the chef is unhappy.
|
| Method resolution order:
|   AngryChefException
|   SensitiveArtistException
|   builtins.Exception
|   builtins.BaseException
|   builtins.object
|
| Data descriptors inherited from SensitiveArtistException:
|
```

```

|  __weakref__
|     list of weak references to the object (if defined)
|
| -----
| Methods inherited from builtins.Exception:
|
|  __init__(...)
|     x.__init__(...) initializes x; see x.__class__.__doc__ for signature
|
| -----
| Data and other attributes inherited from builtins.Exception:
|
|  __new__ = <built-in method __new__ of type object at 0x1E1BCCC8>
|     T.__new__(S, ...) -> a new object with type S, a subtype of T
|
| -----
| Methods inherited from builtins.BaseException:
|
|  __delattr__(...)
|     x.__delattr__('name') <==> del x.name
|
|  __getattr__(...)
|     x.__getattr__('name') <==> x.name
|
|  __reduce__(...)
|
|  __repr__(...)
|     x.__repr__() <==> repr(x)
|
|  __setattr__(...)
|     x.__setattr__('name', value) <==> x.name = value
|
|  __setstate__(...)
|
|  __str__(...)
|     x.__str__() <==> str(x)
|
|  with_traceback(...)
|     Exception.with_traceback(tb) --
|     set self.__traceback__ to tb and return self.
|
| -----
| Data descriptors inherited from builtins.BaseException:
|
|  __cause__
|     exception cause
|
|  __context__
|     exception context

```

```

|  __dict__
|
|  __traceback__
|
|  args

class Breakfast(Meal)
|  Holds the food and drink for breakfast.
|
|  Method resolution order:
|      Breakfast
|      Meal
|      builtins.object
|
|  Methods defined here:
|
|      __init__(self)
|          Initialize with an omelet and coffee.
|
|  -----
|  Methods inherited from Meal:
|
|      printIt(self, prefix='')
|          Print the data nicely.
|
|      setDrink(self, drink='coffee')
|          # Setter for the drink.
|
|      setFood(self, food='omelet')
|          # Setter for the food.
|
|      setName(self, name='')
|          # Setter for the name.
|
|  -----
|  Data descriptors inherited from Meal:
|
|      __dict__
|          dictionary for instance variables (if defined)
|
|      __weakref__
|          list of weak references to the object (if defined)

class Dinner(Meal)
|  Holds the food and drink for dinner.
|
|  Method resolution order:
|      Dinner
|      Meal
|      builtins.object

```

```

|
| Methods defined here:
|
|     __init__(self)
|         Initialize with steak and merlot.
|
|     printIt(self, prefix='')
|         Print even more nicely.
|
| -----
| Methods inherited from Meal:
|
|     setDrink(self, drink='coffee')
|         # Setter for the drink.
|
|     setFood(self, food='omelet')
|         # Setter for the food.
|
|     setName(self, name='')
|         # Setter for the name.
|
| -----
| Data descriptors inherited from Meal:
|
|     __dict__
|         dictionary for instance variables (if defined)
|
|     __weakref__
|         list of weak references to the object (if defined)
|
class Lunch(Meal)
|     Holds the food and drink for lunch.
|
|     Method resolution order:
|         Lunch
|         Meal
|         builtins.object
|
|     Methods defined here:
|
|     __init__(self)
|         Initialize with a sandwich and a gin and tonic.
|
|     setFood(self, food='sandwich')
|         # Override setFood().
|
| -----
| Methods inherited from Meal:
|
|     printIt(self, prefix='')

```

```

|     Print the data nicely.
|
|     setDrink(self, drink='coffee')
|         # Setter for the drink.
|
|     setName(self, name='')
|         # Setter for the name.
|
|     -----
|     Data descriptors inherited from Meal:
|
|     __dict__
|         dictionary for instance variables (if defined)
|
|     __weakref__
|         list of weak references to the object (if defined)
|
class Meal(builtins.object)
|     Holds the food and drink used in a meal.
|     In true object-oriented tradition, this class
|     includes setter methods for the food and drink.
|
|     Call printIt to pretty-print the values.
|
|     Methods defined here:
|
|     __init__(self, food='omelet', drink='coffee')
|         Initialize to default values.
|
|     printIt(self, prefix='')
|         Print the data nicely.
|
|     setDrink(self, drink='coffee')
|         # Setter for the drink.
|
|     setFood(self, food='omelet')
|         # Setter for the food.
|
|     setName(self, name='')
|         # Setter for the name.
|
|     -----
|     Data descriptors defined here:
|
|     __dict__
|         dictionary for instance variables (if defined)
|
|     __weakref__
|         list of weak references to the object (if defined)

```

FUNCTIONS

```

makeBreakfast()
    Creates a Breakfast object.

makeDinner()
    Creates a Breakfast object.
makeLunch()
    Creates a Breakfast object.
DATA
__all__ = ['Meal', 'AngryChefException', 'makeBreakfast', 'makeLunch', ...

```

### 示例说明

**help** 函数是您的助手。它可以显示模块的文档，以及任何 Python 模块的文档。

### 注意：

在调用 **help** 函数读入模块的文档之前，必须向导入模块。

**help** 函数首先打印模块的文档：

```
Help on module meal:
```

#### NAME

```
meal - Module for making meals in Python.
```

#### FILE

```
c:\python30\meal.py
```

#### DESCRIPTION

```
Import this module and then call
makeBreakfast(), makeDinner() or makeLunch().
```

注意 **help** 函数如何将模块文档的第一个概述行同文档的其余行分开。下面显示了这个模块文档的原始字符串：

```

"""
Module for making meals in Python.

Import this module and then call
makeBreakfast(), makeDinner() or makeLunch().

"""

```

**help** 函数取出文档的第一行作为 **NAME** 项，剩下的作为 **DESCRIPTION** 项。

**help** 函数接下来概述了类的功能并显示每个类的文档：

#### CLASSES

```

exceptions.Exception
    AngryChefException
    SensitiveArtistException

```

```
Meal
    Breakfast
    Dinner
    Lunch
```

每个类根据继承关系缩进显示。在这个例子中，概述显示 `Breakfast` 类继承自 `Meal` 类。`help` 函数打印出每个函数和方法的文档：

```
| printIt(self, prefix='')
|     Print the data nicely.
```

如果只在函数或方法的定义附近编写注释，`help` 函数将试着把注释同函数或方法联系起来。然而，并不是所有情况下都奏效，因为 `help` 函数按字母顺序排列方法和函数。例如：

```
|
| setDrink(self, drink='coffee')
|     # Setter for the name.
|
| setFood(self, food='omelet')
|     # Setter for the drink.
|
| setName(self, name='')
|     # Setter for the name.
```

注意，注释与错误的方法相关联。以下是原始的代码：

```
# Setter for the food.
def setFood(self, food='omelet'):
    self.food = food

# Setter for the drink.
def setDrink(self, drink='coffee'):
    self.drink = drink

# Setter for the name.
def setName(self, name=''):
    self.name = name
```

这里的教训是要遵循 `Python` 中为方法建立文档的约定。为了修正这个错误，需要将每个方法上面出现的注释改为一个 `Python` 文档字符串。将 `Python` 文档字符串移动到紧跟在对应的 `def` 命令的下一行。

在开发模块的过程中，可以反复地调用 `help` 函数，观察代码的更改给文档带来怎样的改变。然而，如果已经改变了模块的 `Python` 源文件，需要重新加载之前的模块来调用 `help` 函数。`reload` 函数的参数是一个模块，就像 `help` 函数一样。语法如下：

```
import imp
imp.reload(module)
help(module)
```

例如，为了重新加载 `meal` 模块，可以使用下面的代码：

```
>>>import imp
>>> imp.reload(meal)
<module 'meal' from 'C:\Python30\meal.py'>
```

与文档一样，测试也很重要。对模块测试得越多，它就越适合 Python 应用程序。在程序中使用这些模块之前，就知道模块的功能有效。

#### 10.4.4 测试模块

测试很难而且令人讨厌，所以经常被忽略。虽然如此，测试一个模块可以验证它是否能够正常工作。更重要的是，创建测试使您可以在修改模块后验证功能仍然有效。

任何负责的模块都应该包含执行模块功能的测试函数。测试应该创建模块中定义的类的实例，并调用这些实例的方法。

例如，下面的方法提供了 `meal` 模块的一个测试(注意，如果现在运行它，它将不能工作，因为它还缺少本章后面定义的 `Dinner` 类)：

```
def test():
    '''Test function.'''

    print('Module meal test.')

    # Generic no arguments.
    print('Testing Meal class.')
    m = Meal()

    m.printIt("\t")

    m = Meal('green eggs and ham', 'tea')
    m.printIt("\t")

    # Test breakfast
    print('Testing Breakfast class.')
    b = Breakfast()
    b.printIt("\t")

    b.setName('breaking of the fast')
    b.printIt("\t")

    # Test dinner
    print('Testing Dinner class.')
    d = Dinner()
    d.printIt("\t")
```

```

# Test lunch
print('Testing Lunch class.')
l = Lunch()
l.printIt("\t")

print('Calling Lunch.setFood().')
try:
    l.setFood('hotdog')
except AngryChefException:
    print("\t", 'The chef is angry. Pick an omelet.')

```

将测试函数集成到模块中，这样测试将一直可用。第 12 章将介绍更多关于 Python 测试的内容。

**注意：**

测试永远都不会结束。总是可以加入更多的测试，所以做力所能及的事情即可。

### 10.4.5 将模块作为程序运行

通常，模块不应自己运行。其他 Python 脚本会从模块中导入项并使用它们。然而，由于任何包含 Python 代码的文件都可以看做一个模块，您可以运行一个模块。

因为不希望模块自己运行，Python 为模块定义了一个约定。当一个模块自己运行时，它应该执行模块测试。这提供了测试模块的一个简单方式：将 Python 模块作为一个脚本运行。

为了支持这个约定，Python 提供了一种简单的形式来检测模块是否作为一个程序运行。利用前面给出的 test 函数，可以使用下面的代码执行对模块的测试：

```

if __name__ == '__main__':
    test()

```

**注意：**

如果查看标准 Python 模块的源代码，将会反复碰到这种形式。

下面的示例运行 meal 模块，该模块将在 10.5 节创建。

#### 试一试

#### 运行模块

可以通过使用如下命令行，将模块(例如 meal 模块)作为程序运行：

```

$ python meal.py
Module meal test.
Testing Meal class.
    A fine generic meal with omelet and coffee
    A fine generic meal with green eggs and ham and tea
Testing Breakfast class.
    A fine breakfast with omelet and coffee

```

```

    A fine breaking of the fast with omelet and coffee
Testing Dinner class.
    A gourmet dinner with steak and merlot
Testing Lunch class.
    A fine midday meal with sandwich and gin and tonic
Calling Lunch.setFood().
    The chef is angry. Pick an omelet.

```

### 示例说明

该示例将模块作为一个 Python 程序运行。使用前面给出的形式检查这种情况，可以知道模块只运行 `test` 函数。您所看到的输出是测试的输出。

注意输出中运行了模块中定义的每个类的实例，并测试了 `AngryChefException` 异常的产生。

如果遵循本节中讲述的设计类的全部指导原则，模块将符合其他 Python 开发人员的期望。而且，模块将在脚本中工作得更好。在下一节的实际应用中可以看到这一切，它演示了一个完整的 Python 模块。

## 10.5 创建一个完整的模块

目前为止，本章讲述了创建模块时所需的全部要素。下面的例子演示了一个使用本章描述的技术的完整模块。

`meal` 模块的功能并不十分庞杂，它只是对一日三餐中的食物和饮料进行了建模。

这个模块中的代码刻意做了简化，目的不是完成一个有用的任务，而是演示如何组成一个模块。

### 试一试

### 完成一个模块

输入下面的代码，并将文件命名为 `meal.py`：

```

"""
Module for making meals in Python.
Import this module and then call
makeBreakfast(), makeDinner() or makeLunch().
"""
__all__ = ['Meal', 'AngryChefException', 'makeBreakfast',
           'makeLunch', 'makeDinner', 'Breakfast', 'Lunch', 'Dinner']
# Helper functions.
def makeBreakfast():
    ''' Creates a Breakfast object.'''
    return Breakfast()
def makeLunch():
    ''' Creates a Breakfast object.'''
    return Lunch()
def makeDinner():

```

```

    ''' Creates a Breakfast object.'''
    return Dinner()
# Exception classes.
class SensitiveArtistException(Exception):
    '''Exception raised by an overly-sensitive artist.
    Base class for artistic types.'''
    pass
class AngryChefException(SensitiveArtistException):
    '''Exception that indicates the chef is unhappy.'''
    pass
class Meal:
    '''Holds the food and drink used in a meal.
    In true object-oriented tradition, this class
    includes setter methods for the food and drink.

    Call printIt to pretty-print the values.
    '''
    def __init__(self, food='omelet', drink='coffee'):
        '''Initialize to default values.'''
        self.name = 'generic meal'
        self.food = food
        self.drink = drink
    def printIt(self, prefix=''):
        '''Print the data nicely.'''
        print(prefix, 'A fine', self.name, 'with', self.food, 'and', self.drink)
    # Setter for the food.
    def setFood(self, food='omelet'):
        self.food = food
    # Setter for the drink.
    def setDrink(self, drink='coffee'):
        self.drink = drink
    # Setter for the name.
    def setName(self, name=''):
        self.name = name
class Breakfast(Meal):
    '''Holds the food and drink for breakfast.'''
    def __init__(self):
        '''Initialize with an omelet and coffee.'''
        Meal.__init__(self, 'omelet', 'coffee')
        self.setName('breakfast')
class Lunch(Meal):
    '''Holds the food and drink for lunch.'''
    def __init__(self):
        '''Initialize with a sandwich and a gin and tonic.'''
        Meal.__init__(self, 'sandwich', 'gin and tonic')
        self.setName('midday meal')
    # Override setFood().
    def setFood(self, food='sandwich'):
        if food != 'sandwich' and food != 'omelet':
            raise AngryChefException

```

```

        Meal.setFood(self, food)
class Dinner(Meal):
    '''Holds the food and drink for dinner.'''
    def __init__(self):
        '''Initialize with steak and merlot.'''
        Meal.__init__(self, 'steak', 'merlot')
        self.setName('dinner')
    def printIt(self, prefix=''):
        '''Print even more nicely.'''
        print(prefix, 'A gourmet', self.name, 'with', self.food, 'and', self.drink)
def test():
    '''Test function.'''
    print('Module meal test.')
    # Generic no arguments.
    print('Testing Meal class.')
    m = Meal()
    m.printIt("\t")
    m = Meal('green eggs and ham', 'tea')
    m.printIt("\t")
    # Test breakfast
    print('Testing Breakfast class.')
    b = Breakfast()
    b.printIt("\t")

    b.setName('breaking of the fast')
    b.printIt("\t")
    # Test dinner
    print('Testing Dinner class.')
    d = Dinner()
    d.printIt("\t")
    # Test lunch
    print('Testing Lunch class.')
    l = Lunch()
    l.printIt("\t")
    print('Calling Lunch.setFood().')
    try:
        l.setFood('hotdog')
    except AngryChefException:
        print("\t", 'The chef is angry. Pick an omelet.')
# Run test if this module is run as a program.
if __name__ == '__main__':
    test()

```

### 示例说明

`meal` 模块是根据本章讲述的技术创建的一个完整的模块，它包含测试、文档、异常、类和函数。注意测试几乎与其他代码一样长。您以后将经常看到这种情况。

构建了模块之后，可以将它导入到 Python 脚本中。例如，下面的脚本调用了 `meal` 模块中的类和函数：

```
import meal

print('Making a Breakfast')
breakfast = meal.makeBreakfast()

breakfast.printIt("\t")

print('Making a Lunch')
lunch = meal.makeLunch()

try:
    lunch.setFood('pancakes')
except meal.AngryChefException:
    print("\t", 'Cannot make a lunch of pancakes.')
    print("\t", 'The chef is angry. Pick an omelet.')
```

这个例子使用标准方式导入模块:

```
import meal
```

运行这个脚本时, 将看到如下输出:

```
Making a Breakfast
    A fine breakfast with omelet and coffee
Making a Lunch
    Cannot make a lunch of pancakes.
    The chef is angry. Pick an omelet.
```

下一个脚本演示了导入模块的另一种方法:

```
from meal import *
```

全部脚本如下:

```
from meal import *

print('Making a Breakfast')
breakfast = makeBreakfast()

breakfast.printIt("\t")

print('Making a Lunch')
lunch = makeLunch()

try:
    lunch.setFood('pancakes')
except AngryChefException:
    print("\t", 'Cannot make a lunch of pancakes.')
```

```
print("\t", 'The chef is angry. Pick an omelet.')
```

注意，使用这种导入形式时，不必使用模块名称 `meal` 作为前缀就可以调用 `makeLunch` 和 `makeBreakfast` 函数。

这个脚本的输出与上个脚本的输出应该是类似的。

```
Making a Breakfast
    A fine breakfast with omelet and coffee
Making a Lunch
    Cannot make a lunch of pancakes.
    The chef is angry. Pick an omelet.
```

要十分小心使用的变量的名称。示例模块的名称是 `meal`，这意味着不能在其他任何上下文中使用这个名称，例如作为一个变量名称。如果那样做，就覆盖了作为模块的 `meal` 的定义。下面的例子演示了这个方法的缺陷。

### 试一试

### 破坏性导入

输入下面的脚本，并将文件命名为 `mealproblem.py`：

```
import meal

print('Making a Breakfast')
meal = meal.makeBreakfast()

meal.printIt("\t")

print('Making a Lunch')
lunch = meal.makeLunch()

try:
    lunch.setFood('pancakes')
except meal.AngryChefException:
    print("\t", 'Cannot make a lunch of pancakes.')
    print("\t", 'The chef is angry. Pick an omelet.')
```

运行这个脚本，将看到下面的错误：

```
Making a Breakfast
    A fine breakfast with omelet and coffee
Making a Lunch
Traceback (most recent call last):
  File "C:\Python30\mealproblem.py", line 9, in <module>
    lunch = meal.makeLunch()
AttributeError: 'Breakfast' object has no attribute 'makeLunch'
```

### 示例说明

这个脚本将 `meal` 作为模块，同时也把 `meal` 作为 `Breakfast` 类的一个实例。下面是出问题的代码行：

```
import meal
meal = meal.makeBreakfast()
```

运行上述代码后，`meal` 名称就成为一个变量，它是 `Breakfast` 类的一个实例。这时它改变了对下面的行的解释：

```
lunch = meal.makeLunch()
```

这一行的目的是调用 `meal` 模块中的 `makeLunch` 函数。然而，因为 `meal` 现在是一个对象，Python 解释器试图调用该对象(`Breakfast` 类的一个实例)的 `makeLunch` 方法，由于 `Breakfast` 类没有名为 `makeLunch` 的方法，因此 Python 解释器产生一个错误。

使用模块和调用模块中的函数的语法看起来很像调用一个对象中的方法的语法。一定要小心。

构建模块并对其进行测试之后，下一步是安装它。

## 10.6 安装模块

Python 解释器在 `sys.path` 变量中列出的目录中查找模块。`sys.path` 变量包括当前目录，所以总是可以使用当前路径中的模块。然而，如果希望在多个脚本或者多个系统中使用您编写的模块，需要将它安装到 `sys.path` 变量列出的某个目录下。

大多数情况下，需要将 Python 模块放到 `site-packages` 目录中。查看 `sys.path` 列出的目录，找到一个以 `site-packages` 结尾的目录名。这是一个用于从站点安装包的目录，这些包不是 Python 中包的标准库的一部分。

### 注意：

除了模块，还可以创建模块的包，包是一个安装到相同目录结构的相关模块的集合。更多关于该主题的内容，请参考位于 <http://docs.python.org> 的 Python 文档。

可以使用三种机制安装模块：

- 可以手动创建一个安装脚本或程序。
- 可以创建一个针对您的操作系统的安装程序，例如 Windows 上的 MSI 文件，Linux 上的 RPM 文件，或 Mac OS X 上的 DMG 文件。
- 可以使用方便的 Python `distutils`(代表 `distribution utilities`，分发实用程序)包，来创建基于 Python 的安装文件。

为使用 Python `distutils`，需要创建一个名为 `setup.py` 的安装脚本。最简单的安装脚本可以包括如下的内容：

```
from distutils.core import setup

setup(name='NameOfModule',
      version='1.0',
      py_modules=['NameOfModule'],
      )
```

需要两次包括模块的名称。用自己的模块的名称替换 `NameOfModule`，例如本章例子中的 `meal`。

*Name the script setup.py.*

创建了 `setup.py` 脚本之后，可以使用下面的命令创建一个模块的发布版本：

```
python setup.py sdist
```

参数 `sdist` 是 `software distribution`(软件分发)的缩写。可以试验下面的例子。

## 试一试

## 创建安装包

输入下面的脚本，并将文件命名为 `setup.py`：

```
from distutils.core import setup

setup(name='meal',
      version='1.0',
      py_modules=['meal'],
      )
```

Run the following command to create a Python module distribution:

```
$ python setup.py sdist
running sdist
warning: sdist: missing required meta-data: url
warning: sdist: missing meta-data: either (author and author_email) or
(maintainer and maintainer_email) must be supplied
warning: sdist: manifest template 'MANIFEST.in' does not exist (using default
file list)
warning: sdist: standard file not found: should have one of README, README.txt
writing manifest file 'MANIFEST'
creating meal-1.0
making hard links in meal-1.0...
hard linking meal.py -> meal-1.0
hard linking setup.py -> meal-1.0
creating dist
tar -cf dist/meal-1.0.tar meal-1.0
gzip -f9 dist/meal-1.0.tar
removing 'meal-1.0' (and everything under it)
```

### 示例说明

注意所有的警告。`setup.py` 脚本明显是不完善的。它包括了创建发布版本所需的足够内容，但是不足以满足 Python 的约定。当 `setup.py` 脚本运行完成，您应该在当前目录中看到如下的文件：

```
$ ls
MANIFEST      dist/          meal.py        setup.py
```

`setup.py` 脚本创建了 `dist` 目录和 `MANIFEST` 文件。`dist` 目录包括一个文件，也就是您的模块的压缩版本。

```
$ ls dist
meal-1.0.tar.gz
```

您现在拥有模块的单文件发布版本，它显得有点笨拙，因为模块本身就是一个文件。`distutils` 的好处是模块将会正确地安装。

可以将 `meal-1.0.tar.gz` 文件放到其他的系统上，并安装模块。首先，解压缩该模块，并对其进行扩展。在 Linux、UNIX 和 Mac OS X 下，使用下面的命令：

```
$ gunzip meal-1.0.tar.gz
$ tar xvf meal-1.0.tar
meal-1.0/
meal-1.0/meal.py
meal-1.0/PKG-INFO
meal-1.0/setup.py
```

在 Windows 上，使用类似于 WinZip 的可以处理 `.tar.gz` 文件的压缩程序。在模块扩展之后，可以用下面的命令安装它：

```
python setup.py install
```

例如：

```
$ python setup.py install
running install
running build
running build_py
creating build
creating build/lib
copying meal.py -> build/lib
running install_lib
copying build/lib/meal.py -> /System/Library/Frameworks/Python.framework/
Versions/30/lib/python30/site-packages
byte-compiling /System/Library/Frameworks/Python.framework/Versions/30/lib/
python30/site-packages/meal.py to meal.pyc
```

`distutils` 的好处是它能用于几乎任何 Python 模块。安装命令是相同的，所以无论使用哪一种系统，只需要知道一个安装 Python 模块的命令。

`distutils` 的另一个好处是安装程序会创建模块的文档，可用 `pydoc` 命令查看。例如，下面所示为 `meal` 模块的文档的第一页：

```
$ pydoc meal
Help on module meal:

NAME
    meal - Module for making meals in Python.

FILE
    /Users/ericfj/writing/python/inst2/meal-1.0/meal.py

DESCRIPTION
    Import this module and then call
    makeBreakfast(), makeDinner() or makeLunch().

CLASSES
    exceptions.Exception
        SensitiveArtistException
        AngryChefException
    Meal
        Breakfast
        Dinner
        Lunch

class AngryChefException(SensitiveArtistException)
    | Exception that indicates the chef is unhappy.
```

## 10.7 本章小结

本章将前几章的概念结合在一起，通过示例深入研究了如何创建模块。如果遵循本章描述的技术，模块将适合其他的模块并遵循 Python 的导入规则。

模块就是您选择作为模块处理的 Python 源文件。尽管听起来很简单，但是创建模块时需要遵循下面一些规则：

- 为模块和模块中的所有类、方法和函数建立文档。
- 测试模块并包含至少一个测试函数。
- 定义要导出模块中的哪些项，包括哪些类或函数等。
- 为使用模块时可能出现的问题创建需要的任何异常类。
- 处理模块本身作为一个 Python 脚本执行的情况。

Python 使得在模块内部定义类变得非常容易。

开发模块时，可以分别使用 `help` 和 `reload` 函数显示模块的文档以及重新加载改变的模块。在创建了一个模块之后，可以使用 `distutils` 创建一个模块的发布包。为此，需要创建

一个 `setup.py` 脚本。

第 11 章将介绍正则表达式，它是一个用来在海量数据中发现有关信息的重要概念。

#### 本章要点：

- 模块是 Python 源文件。像函数一样，模块是可重用的代码片段，它可以节省编程人员的时间。由于模块被反复使用并经过完善的测试，它可以使程序减少错误。
- 可以使用 `dir()` 函数浏览模块的属性，例如函数、类和变量。
- 为了使用程序中的模块，必须使用 `import` 语句导入它。也可以使用代码 `from module import item`，从模块导入一个类或函数。
- Python 在特定的位置搜索模块文件。要查看 Python 的搜索路径，导入 `sys` 并使用 `print(sys.path)` 函数进行查看。
- 面向对象编程由封装、继承和多态组成。
- 使用三引号(`"""`)为模块中的对象创建文档。第一个三引号开始注释，第二个三引号结束注释。
- 可以使用 `help()` 函数打印模块中的文档，即 `help(模块名)`。
- 应该总是在模块中生成测试函数，以备后需。

## 10.8 习题

1. 如何访问模块提供的功能？
2. 如何控制模块的哪些项是公有的(公有项在其他 Python 脚本中是可用的)？
3. 如何查看一个模块的文档？
4. 如何找出系统上安装了哪些模块？
5. 什么类型的 Python 命令能放到模块中？





# 第 11 章

## 文 本 处 理

像 Python 这样的脚本语言非常适合一类应用，实际上甚至可以说脚本语言就是专门为这类应用开发的：对目录树中的各种文件进行简单搜索和处理。总的来说，这类应用通常被称为文本处理。Python 是一个出色的脚本工具，通过使用清晰的面向对象的编码风格，可以快速编写文本处理脚本，并在以后将它们扩展为更通用的代码。

本章将介绍：

- 需要文本处理脚本的主要原因。
- 一些用于系统快速管理任务的简单脚本。
- 如何采用独立于平台的方法导航目录结构，以使脚本能够用于 Linux、Windows，甚至 Mac。
- 如何创建正则表达式来比较通过 `os` 和 `os.path` 模块找到的文件。
- 如何使用连续的细化来增强 Python 脚本对找到的数据的筛选。

文本处理脚本是任何计算机系统使用者的工具箱中最有用的工具之一，而 Python 是进行文本处理的一种出色的方法。您一定会喜欢本章介绍的内容。

### 11.1 文本处理的用途

总的来说，文本处理背后的全部思想是找到目标文本。当然，有的情况下数据以结构化的方式组织，这就是所谓的数据库，本章不讨论这方面的内容。数据库具有完善的索引和数据存储功能，如果知道希望寻找什么样的内容，就可以很快找到它。然而，一些数据资源包含的信息完全不是有序的和规整的，例如具有成百上千个文件的目录结构，由数千或数十万行组成的系统进程的事件日志，或者人们在几个月里交流的电子邮件存档。

当需要查找这种类型的数据，或者以某种方式处理它们，文本处理就非常适用。当然，没有理由不将文本处理同其他的数据访问方式结合起来。您可能会发现自己经常要编写一些脚本，它们遍历包含几千行的输出日志，并对查找到的某些数据执行 RDBMS(关系数据库管理系统——第 14 章将进行讨论)查找。这是正常的工作方式。

最后，这种脚本通常可以作为后端数据处理系统的一部分而使用多年。如果脚本采用像 Perl 这样的语言编写，当一些可怜的人被安排五年以后修改它时，它会非常不透明。使用 Python 编写的脚本可以很容易地转化成可重用的对象类，后面将给出一个演示示例。

文本处理领域的两个主要的工具是目录导航和一种称为正则表达式的神奇技术。目录导航是一个不同的操作系统真正给简单的程序带来大量麻烦的领域，因为三个主要的操作系统家族(UNIX、Windows 和 Mac)都以不同的方式组织它们的目录；而且，最棘手的是，它们使用不同的字符分隔子目录。不过 Python 考虑到了这种问题，提供了一系列的跨平台工具来执行目录和路径操作，当一直使用这些工具时，就可以完全消除这个困难。第 8 章中已经看到过这些问题，此处将介绍这些工具的更多用法。

正则表达式是指定一个非常简单的文本解析器的方法，对任意多行文本进行处理时，这种文本解析器的开销很低(意味着速度很快)。正则表达式在很多地方出现，您以前可能见过它们。然而，即使这是您第一次见到它们，也会对它们的功能感到非常满意。在本章的后续部分中，您将简单领略正则表达式的能力，但是这仍将给脚本带来很多强大的功能。

首先看一些需要编写文本处理脚本的原因，然后用新知识做一些试验。使用正则表达式最普遍的原因包括：

- 搜索文件
- 从程序日志(例如 Web 服务器日志)中提取有用的数据
- 搜索电子邮件

下面几节将介绍这些应用。

### 11.1.1 搜索文件

搜索文件，或者用一些文件完成某些工作，是文本处理的支柱。例如，假设您花了几个月的时间在整个 CD 集中提取 MP3 文件，而没有真正注意到如何组织这些放到任意一组目录里的上百个文件。几个月后，考虑按照艺术家将文件组织到不同的目录中时，就会发现目录结构令人异常困惑。文本处理可以解决这些问题。编写一个 Python 脚本来扫描毫无意义的目录结构，并将每个文件名称划分为可能是艺术家姓名的部分。然后在音乐数据库中查找该名称。结果是您可以遵循一定的依据(即使不是按照艺术家的姓名，也是按照一些很好的想法)重新将上百个文件整理到目录中，从而得到一个接近于合理的结构。然后就可以手动整理这些文件，并最终拥有一个有组织的音乐库。

上例中只需使用一次文本处理脚本，但是很容易想象到其他需要经常性地使用脚本的场景，例如处理来自一个不能控制的客户端或数据源的数据。当然，如果需要经常做这类整理，可以很容易地使用 Python 创建一些有组织的工具类来执行这些任务，以避免每次重复劳动。

注意：

当面对一个类似的任务，需要执行大量手工工作来处理计算机上的数据时，考虑使用 Python。写一两个脚本能够节省几小时的单调工作。

还有一个类似的情形是当今的大硬盘的附带结果。很多用户在硬盘上零散地存储文件，好像从来没有时间组织它们。当面对一个装满文件的硬盘，需要从中提取一些知道肯定存在、但无法确定其位置的信息时，情况更加糟糕。Apple、Google、Microsoft 和其他公司都提供了桌面搜索技术来帮助您在收集的文件中搜索数据，以提取有用的信息。

可以把 Python 看做加强版的桌面搜索，因为您可以创建脚本来更好地控制搜索，也可

以在找到的文件上执行操作。

### 11.1.2 日志剪辑

系统管理中另一个的普遍的文本处理任务是需要过滤日志得到各种信息。过滤日志的脚本可以是回答特定问题的临时想法(例如“电子邮件什么时间发送?”或者“最近一次我的程序记录一条特定的消息是什么时候?”),或者它们可能是数据处理系统的永久部分,随着时间变化可以管理正在进行的任务。例如,它们可能是系统管理和性能监视系统的一部分。脚本按规则过滤日志,得到信息的特定子集,这通常被称为日志剪辑,其思想是,就像可以裁剪多边形来适合屏幕,也可以裁剪日志来适合需要的任何系统视角。

无论决定如何使用这些技术,当基本熟悉它们后,这些脚本几乎成为一种习惯。这是一个广泛使用正则表达式的场合,原因有两个:首先,使用 UNIX Shell 命令(如 `grep`)做一级日志剪辑十分普遍;其次,如果用 Python 来完成,在做进一步工作之前,也许需要使用正则表达式将一行分割成一些可用的字段。在任何一种剪辑任务中,都可以很好地使用这两种技术。

下面的几节在对访问文件系统和创建正则表达式做简短介绍之后,将给出两个用于文本处理的脚本。

### 11.1.3 邮件筛选

最后的文本处理任务您大概已经发现是有很用的:通过邮箱文件的处理来找到正常的收件箱搜索不能找到的东西。需要这种强大功能的最常见的原因可能是邮箱文件是存档的,所以您能访问文件,但是不能使用邮件阅读器很方便地阅读它,或者它已经保存在服务器上,但是服务器上没有安装运行的邮件客户端。与移动它到收件箱树并像一个活动文件夹一样处理它相比,写一个脚本扫描它并得到所需要的内容将会更加容易。

然而,很容易想象到另一种情况,即搜索脚本要从外部数据源得到数据,例如网页或者其他数据源(如数据库,更多关于数据库的介绍参见第 14 章),来交叉引用数据或在搜索中完成其他不能使用普通邮件客户端完成的任务。对于这种情况,结合使用文本处理和其他技术可能是一个非常有用的方法,能够找到采用其他方法不容易找到的信息。

## 11.2 使用 os 模块导航文件系统

对于必须在许多不同的平台上执行的日常任务,os 模块和它的子模块 `os.path` 是最有用的工具之一。如果经常需要在 Windows 或 UNIX 上写脚本和程序,使其在另一种操作系统上仍然能工作,从第 8 章中可以知道,Python 在隐藏 Windows 和 UNIX 的不同上做了很多工作。

本章忽略 os 模块的许多功能(从进程控制到获得系统信息),只关注一些用于处理文件和目录的功能。其中一些已经作过介绍,其他的则是新的。

写跨平台脚本的其中一个困难和令人讨厌的地方是 Windows 下用反斜杠(\)分隔目录名,但是 UNIX 下用斜杠(/)。如果想要代码工作在两个操作系统下,甚至将整个路径分成它的

组成部分都是很复杂的。

而且，Python 像其他很多编程语言一样，使用反斜杠字符表示特殊文本，例如 `\n` 表示一个新行。这使得在 Windows 上创建文件路径的脚本变得很复杂。

然而，通过 Python 的 `os.path` 模块，可以得到一些现成的函数，它们可以使用正确的字符为您分割和连接路径名，并且它们能在任何正在运行 Python 的操作系统(包括 Mac)上正确地工作。可以调用单个函数来迭代目录结构，并对它在层次结构中找到的每个文件调用自己选择的另一函数。您将在下面的例子中看到许多这样的函数，但是首先看一个 `os` 和 `os.path` 模块中将被用到的一些有用的函数的综述，如表 11-1 所示。

表 11-1

调用的函数名	说 明
<code>os.getcwd()</code>	返回当前目录。可以将这个函数看做在任何语言中目录函数的基本坐标
<code>os.listdir(directory)</code>	返回一个存储在指定的 <i>directory</i> (目录)中的文件名和子目录列表。然后可以在单个文件上运行 <code>os.stat()</code> ——例如，确定哪个是文件，哪个是子目录
<code>os.stat(path)</code>	返回一个数值元组，该元组给出任何您可能需要的关于文件(或目录)的信息。这些数字从 ANSI C 的同名函数返回的结构中得到，它们的意义如下所示(一些在 Windows 中是虚拟值，但是它们在相同的位置！): st_mode: 文件的访问权限 st_ino: 节点数(UNIX) st_dev: 设备号 st_nlink: 链接号(UNIX) st_uid: 所有者的用户 ID st_gid: 所有者的组 ID st_size: 文件的大小 st_atime: 最后访问时间 st_mtime: 最后修改时间 st_ctime: 创建时间
<code>os.path.split(path)</code>	将路径分割为符合当前操作系统的组成名称。返回一个元组，不是一个列表(这总是使我感到很惊奇)
<code>os.path.join(components)</code>	将名称连接成为一个符合当前操作系统的路径
<code>os.path.normcase(path)</code>	规范化路径的大小写。在 UNIX 下没有影响，因为文件名是区分大小写的；但是在 Windows 下，操作系统在比较文件名的时候是忽略大小写的，所以在比较一个路径和其他路径之前运行 <code>normcase</code> 是有用的，这样如果一个路径包含大写字母，另一个路径不包含，Python 能够采用与操作系统相同的方法进行比较，也就是说，如果两个路径的区别仅在于字母的大小写，那么它们是相同的。在 Windows 下，函数返回一个全小写的路径并转换所有的斜杠为反斜杠

(续表)

调用的函数名	说 明
<code>os.walk(top, topdown=True, onerror=None, followlinks=False)</code>	这是一个出色的函数，它自上而下或自底而上迭代目录树。对于每个目录，它创建一个由 <code>dirpath</code> 、 <code>dirnames</code> 和 <code>filenames</code> 组成的三元组。 <code>dirpath</code> 部分是一个保存目录路径的字符串。 <code>dirnames</code> 部分是 <code>dirpath</code> 中子目录的列表，不包括 “.” 和 “..”。最后， <code>dirnames</code> 是 <code>dirpath</code> 中每个非目录文件的一个列表

还有很多相关的函数，但是以上这些函数是在下面的例子代码中用到的。同模块中的其他函数相比，您可能会更多地用到这些函数。在 Python 的 `os` 和 `os.path` 模块的文档中可以看到其他很多有用的函数。

试一试

列出文件并处理路径

熟悉 Python 函数最好的方法是在解释器中进行试验。试验前面的函数，看看有什么样的响应。

(1) 在 Python 解释器中导入 `os` 和 `os.path` 模块：

```
>>> import os, os.path
```

(2) 首先看您在哪个文件系统中运行。这个例子在 Windows 下完成，所以您的实际过程可能将会不同：

```
>>> os.getcwd()
'C:\\Python31'
```

(3) 如果要用程序进行处理，可以把它拆分成目录路径的元组(使用 `join` 可以把它们连接回去)：

```
>>> os.path.split (os.getcwd())
('C:\\', 'Python31')
```

(4) 为了找到一些关于目录或任何文件的感兴趣的信息，使用 `os.stat`：

```
>>> os.stat('.')
nt.stat_result(st_mode=16895, st_ino=0, st_dev=0, st_nlink=0,
st_uid=0, st_gid=0, st_size=8192, st_atime=1239767131,
st_mtime=1239767131, st_ctime=1234912369)
```

注意：  
名为 “.” 的目录是当前目录的简单表示。

(5) 如果实际上想要列出目录中的文件, 可以使用以下代码:

```
>>> os.listdir('.')
['.javaws', '.limewire', 'Application Data', 'Cookies',
'Desktop', 'Favorites', 'gsview32.ini', 'Local Settings',
'My Documents', 'myfile.txt', 'NetHood', 'NTUSER.DAT',
'ntuser.dat.LOG', 'ntuser.ini', 'PrintHood', 'PUTTY.RND',
'Recent', 'SendTo', 'Start Menu', 'Templates', 'UserData', 'WINDOWS']
```

### 示例说明

上面的大多数代码是很直接的, 而且很容易理解, 但是在继续写一两个完整的脚本之前要注意两点。

首先, 很容易看到如何使用 `listdir`、`split` 和 `stat` 构造一个迭代脚本, 但是并不需要这么做, 因为就像后面将会看到的那样, `os.path` 提供了 `walk` 函数来实现这种功能。`walk` 函数不仅节省编写和调试一个迭代算法的时间和精力(其中您要用自己的方法搜索每个东西), 而且它运行得更快, 因为它内置于 Python, 用 C 编写, 从而使这种情况下的工作变得简单。当已经有一些内置的功能能做相同的工作时, 一般不会想在 Python 中写迭代器。

其次, 注意来自系统调用的 `stat` 调用的输出是很不透明的。它返回的元组对应于 POSIX C 的同名库函数返回的结构, 它的成员值在表 11-1 中描述, 当然也在 Python 文档中描述。`stat` 函数真正告诉您几乎任何想要知道的关于文件和目录的信息, 所以当需要它时, 它是很值得去理解的函数, 即使它初看起来使人有点畏惧。

### 试一试

### 搜索特殊类型的文件

如果您使用过其他编程语言, 会感觉到使用 Python 搜索文件是多么简单。无论以前是否用过其他语言执行这种任务, 都会注意到示例脚本对于这类工作是非常短的。下面的例子使用 `os` 和 `os.path` 模块搜索您调用的函数所在的目录中(意味着当前目录)的 PDF 文件。在 UNIX 或 Linux 系统中, 可以使用命令行和像 UNIX 的 `find` 命令。然而, 如果不经常做这件工作, 将意味着每次想要搜索文件时, 都需要确定 `find` 的命令行语法(因为 `find` 的功能很多, 这是困难的, 并且这个困难是多方面的, 它还要求您已经熟悉它的工作原理)。在 Python 中完成这个工作的另一个好处是, 通过 Python 搜索文件, 您可以基于找到的结果来改进脚本, 以做一些特殊的工作, 在找到程序的新用法后, 可以添加新功能, 以自己需要的方式搜索文件。例如, 搜索文件时可能会得到过多的结果, 可以改进 Python 脚本来进一步筛选结果, 以找到想要的内容。

这是一个展示 `os.walk` 函数的好机会, 所以这个脚本以 `os.walk` 作为基础。这个函数是很出色的, 因为它完成了文件系统迭代的所有繁重工作, 您只需要写一个简单的函数, 对它找到的结果执行参数即可。

(1) 使用自己喜欢的文本编辑器, 在想要扫描 PDF 文件的目录中打开一个名为 `scan_pdf.py` 的脚本, 并输入以下代码:

```
import os, os.path
import re
```

```
def print_pdf (root, dirs, files):
    for file in files:
        path = os.path.join (root, file)
        path = os.path.normcase (path)
        if re.search (r"*\.*\.pdf", path):
            print(path)

for root, dirs, files in os.walk('.')
```

(2) 运行该脚本。显然，下面的输出和您的目录中的内容不一致。为了得到最好的结果，在这个目录中加入一组以.pdf 结尾的文件。

```
$ python scan_pdf.py
.\95-04.pdf
.\non-disclosure agreement 051702.pdf
.\word pro - dokument in lotus word pro 9 dokument45.pdf
.\101translations\2003121803\2003121803.pdf
.\101translations\2004101810\scan.pdf
.\bluemangos\purchase order - michael roberts smb-pt134.pdf
.\bluemangos\smb_pt134.pdf
.\businesssteam.hu\aok.pdf
.\businesssteam.hu\chn14300-2.pdf
.\businesssteam.hu\diplom_bardelmeier.pdf
.\businesssteam.hu\doktor_bardelmeier.pdf
.\businesssteam.hu\finanzamt_1.pdf
.\businesssteam.hu\zollbescheinigung.pdf
.\businesssteam.hu\monday\s3.pdf
.\businesssteam.hu\monday\s4.pdf
.\businesssteam.hu\monday\s5.pdf
.\gerard\done\tg82-20nc-md-04.07.pdf
.\gerard\polytronic\iau-reglement_2005.pdf
.\gerard\polytronic\tg82-20bes user manual\tg82-20bes-md-27.05.pdf
.\glossa\neumag\de_993_ba_s5.pdf
.\glossa\pepperl+fuchs\5626eng3con\vocab - 3522a_recom_flstd.pdf
.\glossa\pepperl+fuchs\5769eng4\5769eng4 - td4726_8400 d-e - 16.02.04.pdf
```

### 示例说明

这是一个不错的小脚本。Python 做了所有的工作，使您得到目录中的 PDF 文件的列表，包括它们的位置及全名，位置中甚至带有空格，在 UNIX 和 Linux 上这是很难处理的。

代码中对路径做了一些额外工作，所以很容易看出文件的位置：调用 `os.path.join`，从起始目录建立每个 PDF 的完整的(相对的)路径名，并调用 `os.path.normcase` 保证所有的文件名在 Windows 下是小写的。在 UNIX 下，`normcase` 将没有影响，因为在 UNIX 下大小写是有意义的，所以不希望改变大写(它也不会改变)；但是在 Windows 下，如果使文件名都以小写表示，将更容易看出它们是否以.pdf 结尾。

注意，代码中使用了一个非常简单的正则表达式来检查文件名结尾。也可以使用 `os.path.splitext` 得到包括文件的基本名和扩展名的元组，并和 pdf 比较，这样可能更清晰。

然而，因为这个脚本是作为过滤器而编写的，从正则表达式(也称为 `regex`)着手，一开始比较就有意义。采用这种方法意味着如果以后想要以某种方式限制输出，例如根据需要加入更多过滤器，可以加入更多的正则表达式比较并在文本表达式中采用好的、容易理解的代码。这主要是风格选择的问题(它也是提前看到正则表达式，并展示它们实际上并不难理解的一个好理由)。

形式 `r "<string constant>"` 简单地告诉 Python，字符串常量应该禁止对反斜杠做出任何特殊处理。这样，`"\n"` 是一个单个字母长度的、相当于一个新行的字符串，而 `r "\n"` 是一个两个字母长度的字符串，表示一个反斜杠字符后面跟着一个字母“n”。因为正则表达式总是包括很多反斜杠，能够通过这个开关禁止它们的特殊意义是很方便的。

### 试一试

### 改进搜索

示例搜索的结果中的 PDF 文件不是很多(大约 100 个)，所以能够通过浏览列表找到要寻找的文件；但是当进行这类搜索时，经常是首先看到进行第一次扫描的结果，然后根据最后需要的结果进行调整。调整的过程包括试验脚本，在看到它能返回更好的结果后，对脚本进行持续改进，以更好地找到需要的信息。

为了体验这类连续或迭代编程，假设不仅要显示所有的 PDF 文件，还想要排除名称中有一个空格的所有 PDF 文件。例如，因为正在搜索的文件是从网站上下载的，它们实际上没有空格，但是很多接收的电子邮件消息中包含的文件是别人文件系统的附件，所以经常包含空格。因此，这个改进是非常合适的，您将有机会用到。

(1) 再次使用喜欢的文本编辑器，打开 `scan_pdf.py` 并按如下代码修改它(修改的部分用斜体显示。如果您跳过了前一个例子，只需要完整输入下面的代码)：

```
import os, os.path
import re

def print_pdf (arg, dir, files):
    for file in files:
        path = os.path.join (dir, file)
        path = os.path.normcase (path)
        if not re.search (r"*.pdf", path): continue
        if re.search (r" ", path): continue

        print (path)

for root, dirs, files in os.walk('.')
```

(2) 现在运行修改后的脚本。同样，这个输出结果将同您的系统中的结果不一致：

```
$ python scan_pdf.py
.\95-04.pdf
.\101translations\2003121803\2003121803.pdf
```

```
.\101translations\2004101810\scan.pdf
.\bluemangos\smb_pt134.pdf
.\businesssteam.hu\aok.pdf
.\businesssteam.hu\chn14300-2.pdf
.\businesssteam.hu\diplom_bardelmeier.pdf
.\businesssteam.hu\doktor_bardelmeier.pdf
.\businesssteam.hu\finanzamt_1.pdf
.\businesssteam.hu\zollbescheinigung.pdf
.\businesssteam.hu\monday\s3.pdf
.\businesssteam.hu\monday\s4.pdf
.\businesssteam.hu\monday\s5.pdf
.\gerard\done\tg82-20nc-md-04.07.pdf
.\gerard\polytronic\iau-reglement_2005.pdf
.\glossa\neumag\de_993_ba_s5.pdf
```

### 示例说明

这个代码有一个格式上的修改，当作为这些面向快速文本处理的过滤器脚本时工作得很好。看看代码中的 `print_pdf` 函数，它首先建立和规范化路径名，然后对它们运行测试以保证得到的是想要的路径名。一个测试失败后，它将使用 `continue` 跳到列表中的下一个文件。这个技术能使全部的测试一个接一个地执行，同时保持代码容易阅读。

## 11.3 使用正则表达式和 re 模块

也许文本处理工具箱中最强大的工具就是正则表达式。尽管匹配简单字符串或子串是有用的，但其功能有限。正则表达式将很多功能包装到一些字符中，它们十分强大的，值得了解。在一些编程语言中，基本正则表达式语法是相同的，您可以找到至少一本单独介绍它们的用法的书，其他许多书(比如本书)中也有其用法的介绍。

就像以前提到的，一个正则表达式定义了一个简单的解析器，可以匹配文本中的字符串。当使用正则表达式在命令行上指定多个文件时，它们在本质上以和通配符一样的方式工作，因为通配符允许定义一个字符串来匹配很多不同的可能的文件名。如果您不知道它们是什么，像 `*` 和 `?` 都是通配符，当通过像 Windows 上的 `dir` 或 UNIX 上的 `ls` 命令使用它们时，可以选择一个以上的文件，但是可能比全部的文件少(就像 `dir win*` 将只打印 Windows 目录中以字符 `w`、`i` 和 `n` 开头，后面是任意字符的文件，这就是 `*` 被称为通配符的原因)。正则表达式和简单的通配符之间存在两个主要的不同：

- 正则表达式可以在长字符串的任何位置匹配多次。
- 以后将会看到，正则表达式比简单的通配符复杂得多，也丰富得多。

当开始学习正则表达式时，要注意的主要问题是：字符串总是与自身匹配。例如，模式 `"xxx"` 总是在 `"abcxxxabc"` 中匹配自己。其他的功能仅是锦上添花的，我们所做的核心只是在其他字符串中找到字符串。

可以加入特殊字符使模式匹配更有意义。最常用的一个特殊字符是一般的通配符 `"."` (一个句点)。句点匹配字符串中的任何一个字符，所以，`"x.x"` 将匹配字符串 `"xxx"` 或 `"xyx"`，甚至 `"x.x"`。

最后的例子提出了一个处理正则表达式的基本点。如果只想查找中间有句点的内容，如“x.x”，该怎么办？实际上，指定“x.x”作为一个模式是不行的，因为它也会匹配“x!x”和“xqx”。正则表达式允许在它们前面使用反斜杠来转义特殊字符。因此，为了匹配“x.x”并且只匹配“x.x”，应该使用模式“x\\.x”，其通过转义符取消句点的特殊意义。

然而，这里遇到了 Python 的字符串处理中的一个问题。Python 也使用反斜杠表示转义符序列，因为“\n”指定回车而“\t”是制表符。为了避免与这种正常处理产生冲突，正则表达式通常指定原生字符串，其实就是在字符串常量的前面加入一个“r”，然后 Python 特殊对待它们。

究竟怎样匹配“x.x”呢？很简单：指定模式 r“x.x”。幸运的是，如果您已经知道这些内容，就已经掌握在 Python 中理解正则表达式最难的部分。剩下的部分很容易。

在指定正则表达式使用的许多特殊字符之前，首先看一下用来匹配字符串的函数，然后在解释器中输入一些正则表达式，通过例子学习它们。

## 试一试

## 练习正则表达式

这个练习使用一些函数式编程工具，您以前可能已经见过它们，但是还没有机会使用。这个练习的想法是能够将正则表达式应用到一组不同的字符串来确定哪些与其匹配，哪些不匹配。要在输入的一行中完成这个工作，可以使用 filter 函数，但是因为 filter 函数，对于其输入列表的每个成员应用带有一个参数的函数，而 re.match 和 re.search 有两个参数，所以您不得不使用函数定义或者匿名 lambda 形式(如本例所示)。不要把它想得太难(可以回到第 9 章看看其工作原理)，因为它所完成的工作是显而易见的：

(1) 打开 Python 解释器并导入 re 模块：

```
$ python
>>> import re
```

(2) 现在，使用各种正则表达式定义一个要过滤的字符串列表：

```
>>> s = ('xxx', 'abcxxxabc', 'xyx', 'abc', 'x.x', 'axa', 'axxxxa', 'axxya')
```

(3) 首先执行最简单的正则表达式：

```
>>> a=filter ((lambda s: re.match(r"xxx", s)), s)
>>>print(*a)
xxx
```

(4) 等一下！为什么没有找到“axxxxa”？这是因为，虽然我们通常谈论的是字符串内的匹配，但在 Python 中，re.match 函数只从它的输入的开始搜索匹配。要在输入的任何地方找到字符串，需要使用 re.search(其拼写词为 research，所以比较容易记住)：

```
>>> b=filter ((lambda s: re.search(r"xxx", s)), s)
>>>print(*b)
xxx, abcxxxabc, axxxxa
```

(5) 搜索那个句点:

```
>>>c=filter ((lambda s: re.search(r"x.x", s)), s)
>>>print(*c)
xxx, abcxxxabc, xyx, x.x, axxxxa
```

(6) 下面的代码显示了如何只匹配句点(通过转义特殊字符):

```
>>> d=filter ((lambda s: re.search(r"x\.x", s)), s)
>>>print(*d)
x.x
```

(7) 也可以通过使用星号搜索任意数量的 x, 星号可以匹配它前面的一串任意字符:

```
>>> e=filter ((lambda s: re.search(r"x.*x", s)), s)
>>>print(*e)
xxx, abcxxxabc, xyx, x.x, axxxxa, axxya
```

(8) 等一等! 如果两个 x 之间什么都没有, “x.\*x” 如何匹配 “axxya”? 秘密在于, 星号会匹配两个 x 之间出现的 0 个或更多的字符。如果确实想要确保两个 x 之间有字符, 可以使用一个加号, 它匹配 1 个或更多个字符:

```
>>>f=filter ((lambda s: re.search(r"x.+x", s)), s)
>>>print(*f)
xxx, abcxxxabc, xyx, x.x, axxxxa
```

(9) 现在您知道如何匹配其中有一个 “c” 的任何字符串:

```
>>> g=filter ((lambda s: re.search(r"c+", s)), s)
>>>print(*g)
abcxxxabc, abc
```

(10) 这里是真正引起兴趣的地方: 如何匹配没有 “c” 的任何字符串? 正则表达式使用方括号表示要匹配的特殊字符集, 如果在列表的开头有一个 “^”, 它意味着不出现在集合中的所有字符, 所以首先想到的可能是:

```
>>>h=filter ((lambda s: re.search(r"[^c]*", s)), s)
>>>print(*h)
xxx, abcxxxabc, xyx, abc, x.x, axa, axxxxa, axxya
```

(11) 这匹配了整个列表。为什么? 因为它匹配具有一个不是 “c” 的字符的任何字符串。上面没有将 ^ 用在正确的地方。为了清楚地说明, 可以过滤一个具有更多 c 的列表:

```
>>>h=filter ((lambda s: re.search(r"[^c]*", s)), ('c', 'cc', 'ccx'))
>>>print(*h)
c, cc, ccx
```

注意:

以前版本的 Python 可能返回不同的元组('ccx')。

(12) 为了真正匹配其中没有一个“c”的任何字符串，必须在字符串的开始和结束使用^和\$特殊字符，然后告诉 re 您想要的是从头到尾不包含 c 字符的字符串：

```
>>>i=filter ((lambda s: re.search(r"^[^c]*$", s)), s)
>>>print(*i)
xxx, xyx, x.x, axa, axxxxa, axxya
```

从上个例子中看到，让 re 理解您的意思有时需要一点努力。最好在—组您理解的数据上试验新的正则表达式，然后仔细检查结果，确信得到了想要的结果，否则在以后会遇到很难理解的情况。

在下面的例子中使用这里展示的技术。通常能够以交互模式运行 Python 解释器，并用示例数据测试正则表达式，直到它匹配您想要的结果。

### 试一试

### 添加测试

至此，示例 scan\_pdf.py 脚本提供了一个测试文件的不错的格式化框架。如前所述，os.walk 函数解决了困难的工作。您编写的 print\_pdf 函数执行测试，在本示例中为搜索 PDF 文件。

这些例子的代码小于 20 行，展现了 Python 真正的能力。根据 print\_pdf 函数的结构，很容易加入测试来改进搜索，就像下面的例子所示。

(1) 再次使用喜欢的文本编辑器打开 scan\_pdf.py，并做如下修改。修改的部分用斜体表示(或者，如果跳过了上一个例子，只需要输入如下的整段代码)：

```
import os, os.path
import re

def print_pdf (arg, dir, files):
    for file in files:
        path = os.path.join (dir, file)
        path = os.path.normcase (path)
        if not re.search (r".*\.pdf", path): continue
        if re.search (r".\.hu", path): continue

        print(path)

for root, dirs, files in os.walk('.'):
    print_pdf(None, root, files)
```

(2) 现在运行修改后的脚本，这个输出将同您的系统中的输出不一致：

```
C:\projects\translation>python scan_pdf.py

.\businesssteam.hu\ak.pdf
```

```
.\businesssteam.hu\chn14300-2.pdf
.\businesssteam.hu\diplom_bardelmeier.pdf
.\businesssteam.hu\doktor_bardelmeier.pdf
.\businesssteam.hu\finanzamt_1.pdf
.\businesssteam.hu\zollbescheinigung.pdf
.\businesssteam.hu\monday\s3.pdf
.\businesssteam.hu\monday\s4.pdf
.\businesssteam.hu\monday\s5.pdf

...
```

### 示例说明

这个例子基于前面例子建立的结构，并加入了另外一个测试。可以不断添加测试，直到创建的脚本最好地满足要求。

在这个例子中，测试只搜索在名称中有一个.hu 的文件名(其包括完整的路径)。这里的假设是名称中有一个.hu(或者所在目录的名称中有一个.hu)的文件是从匈牙利语(hu 是匈牙利的两字母的国家代码)翻译而来的。因此，这个例子表明如何缩小搜索，仅搜索从匈牙利语翻译而来的文件(在实际使用时，显然需要不同的搜索标准。只需加入所需要的测试)。

可以持续改进脚本，以使用 Python 创建一个通用的搜索实用程序。第 12 章将更深入地介绍这一点。

## 11.4 本章小结

文本处理脚本一般是简短的、有用的、可重用的程序，它们或者为一次性和临时的应用编写，或者作为大型数据处理系统的组件使用。文本处理编程人员的主要工具是目录结构导航和正则表达式，本章简明扼要地对它们进行了讨论。

对于这种编程，Python 是很方便的，因为它能够保持平衡，既容易用于简单的、一次性的任务，也易于维护将被重用的代码。

本章展示的主要技术包括：

- 使用 os.walk 函数遍历文件系统。
- 在编写的函数中放入搜索标准并传递给 os.walk 函数。
- 正则表达式能够在每个通过 os.walk 函数找到的文件上执行测试。
- 在 Python 解释器中以交互方式试验正则表达式，以保证它们正确工作。

第 12 章将介绍一个重要的概念：测试。测试不仅能够确保脚本正常运行，而且能够确保当做出修改时脚本仍然能正常运行。

## 11.5 习题

1. 修改 scan\_pdf.py 脚本，从根目录或最顶层目录运行。在 Windows 系统中，这将是当前磁盘(C:、D:等)的最顶层目录。在共享网络上该函数可能很慢，所以如果您的

G:驱动器来自文件服务器,所花费的时间很长时也不要吃惊。在 UNIX 和 Linux 系统中,这应该是最顶层目录(根目录/。

2. 修改 `scan_pdf.py` 脚本,只匹配文件名中有文本 `boobah` 的 PDF 文件。
3. 修改 `scan_pdf.py` 脚本,不包括文件名中有文本 `boobah` 的所有文件。



## 第Ⅲ部分

# 开始使用 Python

第 12 章：测试

第 13 章：使用 Python 编写 GUI

第 14 章：访问数据库

第 15 章：使用 Python 处理 XML

第 16 章：网络编程

第 17 章：用 C 扩展编程

第 18 章：数值编程

第 19 章：Django 简介

第 20 章：Web 应用程序与 Web 服务

第 21 章：集成 Java 与 Python

资源分享网  
PDG



# 第 12 章

## 测 试

就像去看牙科医生一样，如果您希望避免对一个本以为已经处理完了的问题做出再次定位，那么就应该对任何程序都进行彻底的测试。这个教训通常需要一个程序员花费很多年去领会，而且坦白说，您将会在很多年中继续这样做。然而，最重要的是测试必须是有组织的；而且最有效的方法是，必须在开始编写程序时就认识到测试将会同时进行，并对编写和确认测试用例留出时间。

幸运的是，Python 提供了一种称作 PyUnit 的非常好的工具用于组织测试。它是 Java JUnit 程序包的一个 Python 版本，因此对于已经使用过 JUnit 的人来说，他们已经掌握了 Python 程序测试的稳固基础，但是如果没有使用过 JUnit，也不要担心。

本章将介绍：

- 断言的概念和使用。
- 单元测试与测试套件的基本概念。
- 用几个简单的测试示例演示如何组织一个测试套件。
- 对第 11 章中的搜索实用程序进行彻底测试。

PyUnit 的优点在于您可以在软件开发生命周期的早期就准备测试，并且在工作过程中随时可以进行测试。这样就可以在早期捕获到错误，从而可以避免返工，更不用说让其他人发现这些错误。也可以在编写代码之前准备测试用例，以便在编写代码的过程中，能够肯定代码的结果与期望的结果相符合。如果在开始编写代码之前就定义测试用例，就绝不会出现处理一个故障后，结果发现所做的修改已经失控，需要用好几天的时间进行处理的情况。

请注意 PyUnit 不是可以用来测试 Python 程序的唯一框架。实际上还存在很多其他的框架。在撰写本书时，其中绝大多数框架都还没有针对 Python 3.1 进行升级，但是在升级之后一定是非常值得探索的。

### 12.1 断言

Python 中的断言实际上类似于日常语言中的断言。当做出一个断言时，就说明了一些不一定经过证明、但是自己相信是真实的事项。当然，如果正设法表述一个观点，而所做出的断言是不正确的，那么整个论点将会分崩离析。

在 Python 中，断言是一种类似的概念。断言是在代码中使用的语句，在进行开发时，可以使用它们去测试代码的有效性，但是如果这个语言的结果不为真，将会引发一个 `AssertionError` 错误，如果这个错误未被捕获，那么程序将会停止(一般而言，不应捕获 `AssertionError` 错误，因为应该将它们看做一种警告，指出您之前没有考虑正确)。

断言使您能够通过一系列可测试的用例考虑代码。这样就可以保证在开发时，能够沿着“这个值不是 `None`”、“这个对象是字符串类型”或“这个数值的值大于 0”的路线进行测试。从思考如何开发程序的角度看，所有这些语句对于在开发时捕获错误很有帮助。

### 试一试

### 使用断言

创建一组简单的用例，可以看到断言语言特性的工作原理：

```
# Demonstrate the use of assert()
large = 1000
string = "This is a string"
float = 1.0
broken_int = "This should have been an int"

assert large > 500
assert type(string) == type("")
assert type(float) != type(1)
assert type(broken_int) == type(4)
```

尝试使用 `python -i` 运行上面的程序。

### 示例说明

这个简单的测试用例的输出如下所示：

```
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    assert type(broken_init)==type(4)
NameError: name 'broken_init' is not defined
```

从这个栈跟踪可以看到，这段代码只是简单地引发了错误。断言的实现非常简单。如果一个称为 `__debug__` 的特殊内部变量为 `True`，就检查断言。如果任何一个断言没有成功，就会引发一个 `AssertionError` 错误。因为断言实际上是一个 `if` 语句与 `raise` 的组合，当出现问题时，就会 `raise`(引发)一个异常，所以就像使用 `raise` 一样，可以指定一个自定义的消息。可以通过将最后一个断言替换为如下的代码并运行它来进行试验：

```
try:
    assert type(broken_int)==type(4),"broken_int is broken"
except AssertionError: print("Handle the error here.)
```

激活断言的变量 `__debug__` 是一个特殊的变量，在 Python 启动之后它便不可改变，因

此为了将其关闭，您需要对 Python 指定 `-O` (短划线，后面接着大写字母 `O`) 参数。`-O` 告诉 Python 对代码进行优化，意味着除了执行其他优化处理外，还将删除断言测试，因为它知道这些测试将会导致程序运行缓慢(不是很严重，但是类似这样的优化会尽力获得每一点性能)。在部署一个程序的时候将用到 `-O`，因此它删除被认为是开发阶段特性的断言。

如您所见，断言是非常有用的。甚至如果您认为自己可能已经犯了一个错误，并希望在今后的开发周期中捕获到它，那么可以放入一个断言去捕获此错误，并继续完成其他的工作，直到对该代码进行测试为止。当测试代码时，如果一个断言失败，它能显示发生了什么故障，而不是由您去想究竟发生了什么。此外，在部署中使用 `-O` 标志时，断言不会使程序减慢。

断言本身具有几个缺点。首先，断言不提供用于运行测试的结构。您需要创建一个这样的结构，这意味着只有理解了需要从测试中得到什么，测试才能帮助确认代码是否正确，否则它们可能会起阻碍开发的作用。

其次，断言只是停止程序并提供一个异常。能够提供一些总结的系统将会更有用处，以便对测试进行命名、增加测试、删除测试以及将若干测试编译为一个程序包，使您能够针对这个程序是否进行了充分的测试进行总结。这些想法以及一些更多的想法构成了单元测试和测试套件的概念。

## 12.2 测试用例和测试套件

单元测试以测试用例为主要内容，而测试用例是所测试情况的可测试代码块的最小结构单元。当使用 `PyUnit` 时，测试用例是一个简单对象，其中至少包含一个运行代码的测试方法；当测试完成时，它接着将测试的结果与您所做出的有关结果的各种断言进行比较。

`PyUnit` 是其开发者为这个程序包起的名称，但是导入的模块具有一个听起来更加通用的名称 `unittest`。

每个测试用例都是 `TestCase` 类的子类，`TestCase` 是一个恰当而易于记忆的名称。只要重写 `TestCase` 类的 `runTest` 方法，就可以定义一个基本的测试，这是可以写出的最简单的测试用例，但是也可以在单个测试用例类中定义若干不同的测试方法，这样就可以定义许多测试共有的部分，例如安装和清除过程。

测试套件是同时运行于一个特定项目的一系列测试用例。可以找到一些用于组织测试套件的简单工具，但是它们都基于相同的概念，即运行一组测试用例，并对哪些通过、哪些失败以及为何失败进行记录，以便知道如何处理。

因为可能的、最简单的测试套件只包含一个测试用例，而我们已经描述了这种可能的、最简单的测试用例，所以在下面的“试一试”中将编写一个快速的测试示例，以便可以看到这一切是如何组合到一起的。另外，为了避免分散您的注意力，这里对算术运算进行测试，它对系统、文件系统等没有外部的需求。

## 试一试

## 测试加法

(1) 使用最喜欢的编辑器在名为 ch12 的目录下创建一个 test1.py 的文件。使用程序编辑器编辑此文件，使其包含如下代码：

```
import unittest

class ArithTest (unittest.TestCase):
    def runTest (self):
        """ Test addition and succeed. """
        self.failUnless (1+1==2, 'one plus one fails!')
        self.failIf (1+1 != 2, 'one plus one fails again!')
        self.failUnlessEqual (1+1, 2, 'more trouble with one plus one!')

    def suite():
        suite = unittest.TestSuite()
        suite.addTest (ArithTest())
        return suite

if __name__ == '__main__':
    runner = unittest.TextTestRunner()
    test_suite = suite()
    runner.run (test_suite)
```

(2) 现在使用 python 命令运行此代码：

```
.
-----
Ran 1 tests in 0.026s
```

## 示例说明

在第(1)步中，导入 unittest(即包含 PyUnit 框架的模块)之后，定义 ArithTest 类，它是 unittest 中的类 TestCase 的子类。ArithTest 只定义了 runTest 方法，该方法执行实际的测试工作。注意 runTest 方法中定义了文档字符串。对测试进行文档注释至少与对代码进行文档注释是一样重要的。最后，在 runTest 方法中使用了 3 个断言。

以 fail 开始的 TestCase 类，例如 failUnless、failIf 和 failUnlessEqual，是一些变体，它们达到简化测试中条件设置的目的。在编程时，您可能会发现自己对写测试有抵抗情绪(它们会令人分心，有时候甚至很无聊，而且它们很少引人注目，这使得激励自己去编写测试变得更加困难)。PyUnit 试图为您尽可能地简化工作。

在定义了 ArithTest 中的单元测试之后，可能想要在一个可调用的函数中定义测试套件本身，这正是 PyUnit 开发者 Steve Purcell 在模块的文档中所推荐的。这样可以简单地定义正在执行的操作(测试)和在哪里执行操作(在您命名的函数中)。因此，在定义了 ArithTest 之后，创建了 suite 函数，简单地实例化一个寻常的、未修改的测试套件。它将一个单元测试加入其中并返回它。请记住，为了创建一个能返回的对象，suite 函数调用了 TestCase 类。

实际的测试是由返回的 `TestCase` 对象执行的。

如第 6 章所述，只有在作为主程序运行时，Python 才会调用 `TextTestRunner` 类去创建 `runner` 对象。`runner` 对象有一个 `run` 方法，接受 `unittests.TestSuite` 类的一个对象作为参数。`suite` 函数创建这样的一个对象，因此 `test_suite` 被赋值为 `TestSuite` 对象的一个引用。当完成上述操作时，`runner.run` 方法被调用，它使用 `test_suite` 中的 `suite` 函数，以完成在 `test_suite` 中定义的单元测试。

本例中的实际输出是枯燥乏味的，但是因为它意味着一切都已经成功了，所以您会逐渐喜欢这种输出。该单个句点表明已经成功地运行了一个单元测试。相反，如果看到一个 `F` 而不是一个句点，那么意味着测试失败了。在这两种情况中，`PyUnit` 都以一个报告结束一次运行。请注意算术测试的速度非常快。

现在来看看测试失败时的情形。

### 试一试

### 测试有错误的加法

(1) 使用最喜欢的文本编辑器向 `test1.py` 中添加第二组测试。这些测试以第一个示例为基础。将如下代码添加到文件中：

```
class ArithTestFail (unittest.TestCase):
    def runTest (self):
        """ Test addition and fail. """
        self.failUnless (1+1==2, 'one plus one fails!')
        self.failIf (1+1 != 2, 'one plus one fails again!')
        self.failUnlessEqual (1+1, 2, 'more trouble with one plus one!')
        self.failIfEqual (1+1, 2, 'expected failure here')
        self.failIfEqual (1+1, 2, 'second failure')

def suite_2():
    suite = unittest.TestSuite()
    suite.addTest (ArithTest())
    suite.addTest (ArithTestFail())
    return suite
```

还需要改变启动测试的 `if` 语句，并且需要保证它出现在文件的末尾，使两个类对它都是可见的：

```
if __name__ == '__main__':
    runner = unittest.TextTestRunner()
    test_suite = suite_2()
    runner.run (test_suite)
```

(2) 现在运行修改后的文件(在已经保存该文件后)。对于第二组测试，将得到与之前的测试非常不同的结果：

```
.F
=====
FAIL: Test addition and fail.
```

```

-----
Traceback (most recent call last):
  File "C:\Python30\ch12\test1.py", line 22, in runTest
    self.failIfEqual(1+1,2, 'expected failure here')
AssertionError: expected failure here
-----

```

```

Ran 2 tests in 0.062s

```

```

FAILED (failures=1)
>>>

```

### 示例说明

这里保留了第一个示例中的成功测试，并添加了第二个测试，而且知道它会失败。现在的结果是一个句点(第一个测试的结果)，后面是第二个测试的结果 F，代表“Failed”，所有这些都显示在测试运行的第一行输出中。

在测试运行之后，结果报告将被打印出来，以便您可以对所发生的情况进行仔细检查。成功的测试在报告中没有输出，这是合情合理的：设想您进行了一百个测试，其中只有两个失败，您将不得不在比现在多得多的输出中艰难地寻找那些失败的测试。这看起来像是把重点放在事物的消极方面，但您会习惯的。

因为存在一个失败的测试，所以失败测试的栈跟踪信息被显示出来。另外，从 `runTest` 方法中产生了几条不同的消息。第一个应该查看的是 FAIL 消息。它实际上使用了 `runTest` 方法的文档字符串，并将其打印在最上部，以便查阅那些失败的测试。因此，从这里学到的第一个经验便是应该在文档字符串中记录测试。同时，您会发现在 `runTest` 中为失败的测试指定的消息与 `PyUnit` 产生的异常一同显示出来了。

该报告最后列出了实际运行的测试用例数和失败的测试用例数。

## 12.3 测试装置

一切都很好，而且进行得很顺利，但是实际的测试通常还包括在运行测试之前准备测试(创建文件、创建合适的目录结构、保证一切都处于良好的状态，以及为保证对正确的对象进行测试所需要做的其他工作)。另外，在测试结束时经常需要做一些清理工作。

在 `PyUnit` 中，测试用例所运行的环境被称为测试装置(test fixture)。基本的 `TestCase` 类定义了两个方法：`setUp` 在运行测试前调用，`tearDown` 在测试用例完成后调用。这两个方法用于处理测试装置的创建和清理中所涉及的工作。

您应该知道如果 `setUp` 失败，`tearDown` 将不被调用。然而，即使测试用例本身失败，`tearDown` 也仍将被调用。

记住当创建测试时，每个测试的初始状态不应该依赖于之前的测试成功还是失败。每个测试用例应该为自身创建一个原始的测试装置。如果不能保证这一点，那么将会得到不

一致的测试结果，使工作变得更加复杂。

为了节省时间，当在一个相同配置的测试装置上重复运行一些相似的测试时，可以创建 `TestCase` 类的子类来定义设置方法和清除方法。这将得到一个可以作为起点使用的单个类。一旦完成了上述工作，建立该类的子类以定义每个测试用例。也可以选择单元测试用例类中定义若干个测试用例方法，然后为每个方法实例化测试用例对象。下面的示例中演示了这两种方法。

## 试一试

## 进行测试

(1) 使用最喜欢的文本编辑器添加一个新文件 `test2.py`。使其类似于如下的示例。注意这个示例是建立在之前的几个示例的基础上的。

```
import unittest
class ArithTestSuper (unittest.TestCase):
    def setUp (self):
        print("Setting up ArithTest cases")
    def tearDown (self):
        print("Cleaning up ArithTest cases")
class ArithTest (ArithTestSuper):
    def runTest (self):
        """ Test addition and succeed. """
        print("Running ArithTest")
        self.failUnless (1+1==2, 'one plus one fails!')
        self.failIf (1+1 != 2, 'one plus one fails again!')
        self.failUnlessEqual (1+1, 2, 'more trouble with one plus one!')

class ArithTestFail (ArithTestSuper):
    def runTest (self):
        """ Test addition and fail. """
        print("Running ArithTestFail")
        self.failUnless (1+1==2, 'one plus one fails!')
        self.failIf (1+1 != 2, 'one plus one fails again!')
        self.failUnlessEqual (1+1, 2, 'more trouble with one plus one!')
        self.failIfEqual (1+1, 2, 'expected failure here')
        self.failIfEqual (1+1, 2, 'second failure')

class ArithTest2 (unittest.TestCase):
    def setUp (self):
        print("Setting up ArithTest2 cases")
    def tearDown (self):
        print("Cleaning up ArithTest2 cases")
    def runArithTest (self):
        """ Test addition and succeed, in one class. """
        print("Running ArithTest in ArithTest2")
        self.failUnless (1+1==2, 'one plus one fails!')
        self.failIf (1+1 != 2, 'one plus one fails again!')
        self.failUnlessEqual (1+1, 2, 'more trouble with one plus one!')
```

```

def runArithTestFail (self):
    """ Test addition and fail, in one class. """
    print("Running ArithTestFail in ArithTest2")
    self.failUnless (1+1==2, 'one plus one fails!')
    self.failIf (1+1 != 2, 'one plus one fails again!')
    self.failUnlessEqual (1+1, 2, 'more trouble with one plus one!')
    self.failIfEqual (1+1, 2, 'expected failure here')
    self.failIfEqual (1+1, 2, 'second failure')

def suite():
    suite = unittest.TestSuite()
    # First style:
    suite.addTest (ArithTest())
    suite.addTest (ArithTestFail())
    # Second style:
    suite.addTest (ArithTest2("runArithTest"))
    suite.addTest (ArithTest2("runArithTestFail"))

    return suite

if __name__ == '__main__':
    runner = unittest.TextTestRunner()
    test_suite = suite()
    runner.run (test_suite)

```

## (2) 运行上面的代码:

```

Setting up ArithTest cases
Running ArithTest
Cleaning up ArithTest cases
Setting up ArithTest cases
Running ArithTestFail
Cleaning up ArithTest cases
Setting up ArithTest2 cases
Running ArithTest in ArithTest2
Cleaning up ArithTest2 cases
Setting up ArithTest2 cases
Running ArithTestFail in ArithTest2
Cleaning up ArithTest2 cases

```

```

=====
FAIL: Test addition and fail.
-----
Traceback (most recent call last):
  File "C:/Python31/test2.py", line 25, in runTest
    self.failIfEqual (1+1, 2, 'expected failure here')
AssertionError: expected failure here
=====
FAIL: Test addition and fail, in one class.

```

```
-----
Traceback (most recent call last):
  File "C:/Python31/test2.py", line 48, in runArithTestFail.
    self.failIfEqual (1+1, 2, 'expected failure here')
AssertionError: expected failure here
-----

Ran 4 tests in 0.396s

FAILED (failures=2)
>>>
```

### 示例说明

在继续后面的内容之前先看一看上面的代码。需要注意的第一点是所做的测试与之前的测试相同。一个测试成功，另一个测试失败，但是所做的是两组测试，每组测试都利用同一个测试装置，但却以不同的方式实现多个单元测试用例。

使用哪种方式完全由您决定，它实际上取决于您认为哪种方法的可读性和可维护性比较好。

代码中的第一组类(ArithTestSuper、ArithTest 和 ArithTestFail)与 test1.py 中的第二组示例在本质上是相同的测试，但是这次创建了一个名为 ArithTestSuper 的类。ArithTestSuper 实现 setUp 和 tearDown 方法。这两个方法没有完成太多功能，但是它们确实显示了将要引入条件语句的位置。每个单元测试类是新的 ArithTestSuper 类的子类，因此现在它们将对测试装置进行相同的设置。如果需要改变测试装置，那么现在可以在 ArithTestSuper 的类中进行更改，而且此更改在其所有的子类中生效。

实际的测试用例 ArithTest 和 ArithTestFail 除了添加了一些打印调用之外，与之前示例中的测试用例是相同的。

最后的测试用例类 ArithTest2 与之前已经定义的三个类完成同样的功能。唯一不同的是它合并了测试装置的方法与测试用例的方法，而且它没有重写 runTest 函数。相反，ArithTest2 定义了两个测试用例方法：runArithTest 和 runArithTestFail。在测试运行期间，创建测试用例实例时，这两个方法被显式地调用，从 suite 被更改的定义中可以看到这一点。

一旦此测试被实际运行，可以立即看到一个变化：因为设置、测试和清理函数都向 stdout 中写入信息，所以可以看到所有函数被调用的顺序。请注意，即使是在一个失败的测试之后，也会调用清理函数。最后，请注意那些失败测试的跟踪信息已经被收集起来，并一起显示在报告的最后。

## 12.4 用极限编程整合

考虑将上面所有这些技术组合在一起的一种很好的方法是在一个不断扩展的编码项目的开发过程中使用一个测试套件。这种策略是极限编程(Extreme Programming, 简写 XP)方法的基础。极限编程是编程的一种流行趋势：首先，对编码进行计划；然后将测试用例集合编写为一种框架；之后才编写实际的代码。在完成编码任务后的任何时候，可以重新

运行该测试套件以查看接近设计目标的程度，该测试套件具体体现了这个设计目标(当然，同时也在调试测试套件，这样做非常好！)，这种技术是一种在软件过程早期发现编程错误的非常好的方法，可以定位低层次代码中的故障，在进一步开展高层次工作之前使代码稳定下来，而且在 Python 中使用 PyUnit 建立这个过程是极其容易的，在下一个示例中可以看到这一点。

这个示例还包含了测试装置的一个实际可行的使用，创建一个包含几个文件的测试目录，然后在测试用例完成之后清理该测试目录。这个示例也说明了所有测试用例方法的命名约定，即使用“test”加上方法名，例如 testMyFunction，这样使 unittest.main 过程能够对它们进行自动地识别和运行。

### 12.4.1 用 Python 实现搜索实用程序

这种编程方法的第一步与其他的方法一样是定义目标，在此示例中，就是可以在工作中使用的一种通用的、可重用的搜索功能。显然，期待一个单一的搜索实用程序实现所有可能的文本处理功能将会是一种时间上的浪费，但是某些特定的搜索任务往往会多次重复。因此，如果希望实现一个通用的搜索实用程序，您会如何做呢？UNIX 的 find 命令是一个能从中寻找有用功能的好对象，因为它不仅能够遍历目录树并在找到的每个文件上执行动作，而且能够指定某些可以跳过的目录、指定一些相当复杂的命令行逻辑组合以及很多其他的操作，例如通过文件的修改日期和大小进行搜索。

另外，find 命令不包括任何关于文件内容的搜索(在 UNIX 下实现该操作的标准方法是在 find 内部调用 grep)，而且它的许多功能需要调用后处理程序，但是在一个通用的 Python 搜索实用程序中是不需要这些调用的。

在 Python 中，当搜索文件时，可能需要的功能包括如下几点：

- 在 Python 中可以方便地使用的一些返回值：一个包括完整路径、文件名、扩展名和文件大小的元组是一个很好的开始。
- 搜索文件名的正则表达式和搜索文件内容的正则表达式(如果不指定内容搜索，为了节省开销，不应该打开文件)。
- 其他可选的搜索项：文件大小、存在时间、最近一次修改信息等都是有用的。

一个真正通用的搜索实用程序可能包含一个用一些文件的参数调用的函数，从而可以指定更高级的逻辑。UNIX 的 find 命令允许在命令行中使用非常通用的逻辑组合，但是坦率地说，命令行中的复杂逻辑是很难让人理解的。这种行为在一个实际的编程语言(比如 Python)中可以产生更好的效果，因此可以包含一种可选的逻辑功能来缩小搜索的范围。

通常，完成此类任务的一个很好的想法是首先关注核心功能，在最初的代码已经处于良好状态之后再增加更多的一些功能。如下示例就采用了这种构造方法：首先从一个基本的搜索框架开始，这个框架封装了 os 和 re 模块的示例中包含的功能，在完成最初的部分后，就可以添加更多的功能。这种软件开发的增量式方法可以帮助您防止在进行一些根本性的工作之前就陷入到细节中，而且类似这种通用实用程序的软件的功能非常复杂，您很容易就会失去思路。

因为该程序也将演示极限编程方法，所以将遵照此方法，首先编写调用 find 实用程序的代码，将这些代码构建为测试套件，然后再编写 find 实用程序。当然，这里稍有欺骗的

成分。通常，在开发过程中需要不断修改测试套件，但是在这个例子中，此测试套件已经保证对于测试代码的最终版本是适用的。虽然如此，仍然可以使用这个示例进行学习。

## 试一试

## 编写测试套件

(1) 使用最喜欢的文本编辑器创建 `test_find.py` 文件。输入如下代码：

```
import unittest
import find
import os, os.path

def filename(ret):
    return ret[1]

class FindTest (unittest.TestCase):
    def setUp (self):
        os.mkdir ("_test")
        os.mkdir (os.path.join("_test", "subdir"))
        f = open (os.path.join("_test", "file1.txt"), "w")
        f.write ("""first line
second line
third line
fourth line""")
        f.close()

        f = open (os.path.join("_test", "file2.py"), "w")
        f.write ("""This is a test file.
It has many words in it.
This is the final line.""")
        f.close()

    def tearDown (self):
        os.unlink (os.path.join ("_test", "file1.txt"))
        os.unlink (os.path.join ("_test", "file2.py"))
        os.rmdir (os.path.join ("_test", "subdir"))
        os.rmdir ("_test")

    def test_01_SearchAll (self):
        """ 1: Test searching for all files. """
        res = find.find (r".*", start="_test")
        self.failUnless (map(filename,res) == ['file1.txt', 'file2.py'],
                          'wrong results')

    def test_02_SearchFileName (self):
        """ 2: Test searching for specific file by regexp. """
        res = find.find (r"file", start="_test")
        self.failUnless (map(filename,res) == ['file1.txt', 'file2.py'],
                          'wrong results')
```

```

    res = find.find (r"py$", start="_test")
    self.failUnless (map(filename,res) == ['file2.py'],
                      'Python file search incorrect')

def test_03_SearchByContent (self):
    """ 3: Test searching by content. """
    res = find.find (start="_test", content="first")
    self.failUnless (map(filename,res) == ['file1.txt'],
                      "didn't find file1.txt")
    res = find.find (where="py$", start="_test", content="line")
    self.failUnless (map(filename,res) == ['file2.py'],
                      "didn't find file2.py")
    res = find.find (where="py$", start="_test", content="second")
    self.failUnless (len(res) == 0,
                      "found something that didn't exist")

def test_04_SearchByExtension (self):
    """ 4: Test searching by file extension. """
    res = find.find (start="_test", ext='py')
    self.failUnless (map(filename,res) == ['file2.py'],
                      "didn't find file2.py")
    res = find.find (start="_test", ext='txt')
    self.failUnless (map(filename,res) == ['file1.txt'],
                      "didn't find file1.txt")

def test_05_SearchByLogic (self):
    """ 5: Test searching using a logical combination callback. """
    res = find.find (start="_test", logic=lambda x: (x['size'] < 50))
    self.failUnless (map(filename,res) == ['file1.txt'],
                      "failed to find by size")

if __name__ == '__main__':
    unittest.main()

```

(2) 现在创建另一个被命名为 `find.py` 的代码文件。注意这只是实际的 `find` 实用程序的框架，运行它将会失败。不过这没有关系，在测试和在极限编程中，失败是一件好事，因为它们使您明白还需要做些什么工作：

```

import os, os.path
import re
from stat import *

def find (where='.*', content=None, start='.', ext=None, logic=None):
    return ([])

```

(3) 从命令行运行 `test_find.py` 测试套件。如下代码显示了一个摘录：

```

C:\projects\articles\python_book\ch12_testing>python test_find.py
FFFFF

```

```
=====
FAIL: 1: Test searching for all files.
-----
```

```
[a lot more information]
```

```
Ran 5 tests in 0.421s
```

```
FAILED (failures=5)
```

### 示例说明

测试套件最开始的三行代码导入了 `PyUnit` 模块和要测试的 `find` 模块(此模块实际上还没有编写), 以及在创建和清理测试装置时用于文件和目录处理的 `os` 和 `os.path` 模块。接着定义了一个简单的帮助函数, 用于从搜索结果中提取出文件名, 使对结果正确性的检查变得更加简单。

之后是测试套件本身。这个示例中的所有测试用例都是基类 `FindTest` 的实例。`FindTest` 类首先用 `setUp` 和 `tearDown` 方法定义了测试用例中使用的测试装置, 接着定义了 5 个测试用例。

所有测试用例中的测试装置都包含一个测试目录; 此主目录下的一个子目录, 用于保证在扫描时子目录不会被作为文件处理; 以及两个以 `.txt` 和 `.py` 为扩展名的测试文件。测试文件的内容是相当随意的, 但是它们包含不同的文字, 以便测试套件可以包含一些通过使用内容搜索对它们进行区分的测试。

测试用例本身用一个序数和一个描述性名称命名, 并且每个测试用例以字符串 “test” 开头。因此在运行测试套件时, `unittest.main` 函数可以自动发现它们。因为在测试时使用一个简单的字符排序对测试进行排序, 所以该序数保证测试将按照定义的适当的次序运行。每个文档字符串引用测试的编号, 然后简单地描述测试类型。这有助于快速、容易地理解失败测试的结果, 从而可以准确地跟踪错误发生的位置。

最后, 在定义了这些测试用例之后, 有两行代码对脚本是直接运行的、而不是作为一个模块被调用进行检测, 而且, 如果脚本是直接运行的, 那么将使用该测试用例中的 `unittest.main` 创建一个默认的测试运行程序。然后 `unittest.main` 调用找到所有的测试用例, 按照序数对它们进行排序, 并按次序运行这些测试用例。

第二个文件是 `find` 实用程序本身的框架。除了决定它的功能是什么以及如何对它进行调用之外, 还没有编写任何代码, 因此编码便是下一步的任务了。

### 试一试

#### 一个通用的搜索框架

(1) 使用最喜欢的文本编辑器打开 `find.py`, 并按照如下代码对其进行修改:

```
import os, os.path
import re
from stat import *
```

```

def find (where='.*', content=None, start='.', ext=None, logic=None):
    context = {}
    context['where'] = where
    context['content'] = content
    context['return'] = []

    os.walk (start, find_file, context)

    return context['return']

def find_file (context, dir, files):
    for file in files:
        # Find out things about this file.
        path = os.path.join (dir, file)
        path = os.path.normcase (path)
        try:
            ext = os.path.splitext (file)[1][1:]
        except:
            ext = ''
        stat = os.stat(path)
        size = stat[ST_SIZE]

        # Don't treat directories like files
        if S_ISDIR(stat[ST_MODE]): continue

        # Do filtration based on the original parameters of find()
        if not re.search (context['where'], file): continue

        # Do content filtration last, to avoid it as much as possible
        if context['content']:
            f = open (path, 'r')
            match = 0
            for l in f.readlines():
                if re.search(context['content'], l):
                    match = 1
                    break
            f.close()
            if not match: continue

        # Build the return value for any files that passed the filtration tests.
        file_return = (path, file, ext, size)
        context['return'].append (file_return)

```

(2) 现在，例如要去查找包含“find”的 Python 文件，可以通过启动 Python 并执行下列操作实现：

```

>>> import find
>>> find.find(r"py$", content='find')
[('.\\find.py', 'find.py', 'py', 1297), ('.\\test_find.py',
'test_find.py', 'py', 1696)]

```

### 示例说明

除了使用一个更加通用的 `find_file` 函数扫描每个目录下的文件，而不是使用一个特定于任务的 `print_pdf` 函数之外，这个示例实际上与第 11 章中的第一个示例完成相同的功能。因为此代码比其他示例的脚本更加复杂，所以可以看到，如果提前有一个测试框架可用，就可以大大简化对最初各个版本的调试。第一个版本满足了测试套件中的前三个测试用例。

因为 `find_file` 函数完成大部分的过滤工作，所以很显然它需要访问各个搜索参数。另外，因为在搜索的过程中它还需要一个位置用于保存构建的命中记录的列表，考虑到字典是可以修改的，并可以包含任意数量的命名值，所以字典结构是其参数的一个很好的选择。因此，主 `find` 函数首先构建这个字典，并将搜索参数放入其中。之后，正如在本章开始的 PDF 搜索代码示例中所做的一样，它调用 `os.walk` 遍历目录结构。一旦遍历完成，它返回在搜索期间构建的返回值(找到的文件列表及各个文件的信息)。

在搜索期间，`os.walk` 在它找到的每个目录上调用 `find_file`，传递在搜索开始时构建的字典参数、当前目录的名称和目录中所有文件的一个列表。该 `find_file` 函数首先扫描这个文件列表，并通过在每个文件上运行 `os.stat` 以确定该文件的一些基本信息。如果该“文件”实际上是一个子目录，那么函数继续运行；因为所有搜索参数适用于文件名称，而不适用于目录树中的各个点(而且因为除非文件被打开，否则内容搜索将会导致一个错误！)，所以该函数通过使用从 `os.stat` 调用中收集的信息跳过那些子目录。

当搜索结束时，该函数使用在字典参数中存储的搜索参数去尽可能地排除一些文件。如果指定了内容参数，那么它将打开每个文件并对其进行读取，否则对文件不做任何操作。

如果一个文件已经通过了所有搜索参数(在这个初始版本中只有两个)的测试，那么将为此文件构建一个条目并将其附加到命中记录的列表中，该条目包含此文件相对于搜索起点的完整路径名、文件名自身、文件的扩展名及文件大小。很自然，可以将认为有用的与文件相关的任何值的集合返回，但是这些值是一个良好的基本集，可以使用它们构造一个类似于目录的命中记录的列表，或者使用它们对文件执行某种操作。

### 12.4.2 一个更加强大的 Python 搜索

请记住，这个程序的目的是示范增量编程方法，所以在第一个示例后，停下来给出一些解释很合适，但是在这个通用的搜索实用程序中还可以包含很多不错的搜索参数，而且当然，在最开始编写的测试套件中仍然有两个单元测试用例需要处理。因为 Python 提供一种关键字参数机制，所以将新命名的参数添加到函数定义中，并将它们加入到搜索上下文字典中将是非常简单的，之后按照需要在 `find_file` 中使用它们，而不是采用一种麻烦的做法，对 `find` 函数做一些单独的调用。

下一个示例说明了添加一个关于文件扩展名的搜索参数是多么容易，并且还加入了一个逻辑组合的回调函数。您可以添加更多的搜索参数，下面的代码只是显示了如何开始自己的扩展(本章有个练习需要您添加一些搜索参数，例如文件最近一次修改的日期参数等)。

虽然文件扩展名参数作为一个简单的值是很容易构想和实现的，只需要在搜索上下文中添加该参数，以及在 `find_file` 中添加一个过滤测试，但是计划一个逻辑组合回调参数需要一点思考。指定回调的一般策略是定义一个参数的集合(比如文件名、大小和修改时间)，之后在对回调的每次调用中将那些值传进来。如果要添加一个新的搜索参数，将面临一个

选择：可以随意指定新参数不能包含到逻辑组合中；也可以改变回调规范并使所有存在的回调无效，从而使用新的代码；或者还可以定义多种逻辑回调，每种有不同的参数集合。所有这些选择都不是非常令人满意，但终究需要做出一个选择。

然而，在 Python 中，字典结构提供了克服这个问题的一种方便的方法。如果定义一个字典参数，此参数传递指定的值以在逻辑组合中使用，那么未使用的参数将被简单忽略。于是，较新的代码定义了更多搜索参数，而较早的回调依然能与较新的代码一同使用，不需要对已有的代码做任何修改。下一个“试一试”中更新了搜索代码，将回调函数定义为以一个字典为参数并返回一个标志的函数，这是一个真正的过滤函数。在示例部分以及第 13 章中，可以看到它如何用在搜索测试套件的测试用例 5 中。

添加一个逻辑组合回调也使对数值参数(例如文件大小或更改时间)的处理变得简单。调用者不太可能用文件的精确大小进行搜索，相反，人们通常针对大于或小于给定值的文件进行搜索，或者在一个给定大小的区间进行搜索。换言之，大多数关于数值的搜索早已变成逻辑组合。因此，逻辑组合回调还应该得到文件的大小和日期，以便可以编写过滤函数来对它们进行搜索。幸运的是，这很简单——`os.stat` 的结果已经可用并可以复制到字典中。

### 试一试

### 扩展搜索框架

(1) 再一次使用最喜欢的文本编辑器，打开上一个示例中的 `find.py` 文件，并对它做如下修改：

```
import os, os.path
import re
from stat import *

def find (where='.*', content=None, start='.', ext=None, logic=None):
    context = {}
    context['where'] = where
    context['content'] = content
    context['return'] = []
    context['ext'] = ext
    context['logic'] = logic

    for root, dirs, files in os.walk(start):
        find_file(context, root, files)

    return context['return']

def find_file (context, dir, files):
    for file in files:
        # Find out things about this file.
        path = os.path.join (dir, file)
        path = os.path.normcase (path)
        stat = os.stat(path)
        size = stat[ST_SIZE]
        try:
```

```

        ext = os.path.splitext (file)[1][1:]
    except:
        ext = ''

    # Don't treat directories like files
    if S_ISDIR(stat[ST_MODE]): continue

    # Do filtration based on passed logic
    if context['logic'] and not context['logic'](locals()): continue

    # Do filtration based on extension
    if context['ext'] and ext != context['ext']: continue

    # Do filtration based on the original parameters of find()
    if not re.search (context['where'], file): continue

    # Do content filtration last, to avoid it as much as possible
    if context['content']:
        f = open (path, 'r')
        match = 0
        for l in f.readlines():
            if re.search(context['content'], l):
                match = 1
                break
        f.close()
        if not match: continue

    # Build the return value for any files that passed the filtration tests.
    file_return = (path, file, ext, size)
    context['return'].append (file_return)

```

(2) 现在查找大于 1000 个字节并早于昨天的文件:

```

>>> import find
>>> find.find(r"py$", content='find')
[( './\find.py', 'find.py', 'py', 1297), ( './\test_find.py',
'test_find.py', 'py', 1696)]

```

(3) 也可以从命令行运行 test\_find.py 测试套件:

```

C:\projects\python_book\ch11_regexp>python test_find.py
.....
-----
Ran 5 tests in 0.370s

```

**注意:**

在开发过程中, 这种运行并不会如此顺利!

## 12.5 软件生命周期中的正规测试

之前示例中显示的测试套件的结果是在一个稍微复杂的编程示例中得到的比较整洁和稳定的代码，以及一些定义良好的测试用例，通过实际运行证明它们可以正确地工作。在一个只有 30 行代码左右的软件“产品”的情形中，这是一个很迅速、很简单过程，虽然在 30 行代码中出现的编程错误的数量可能是令人震惊的！

当然，在一个实际的软件生命周期中，将有数千行代码。在具有这种规模的项目中，没有人可以在发布代码前就定义所有可能的测试用例。开发阶段中的正规测试将极大地改进代码，并增强您对它的信心，这是无疑的，但是当发布代码后，一些错误将仍然存在。

在软件生命周期的维护阶段，在目标代码处于生产阶段之后，就会把故障报告进行归档。如果在开发过程中采用一种集成测试的方法，那么可以将故障报告看做是对测试用例中的故障和代码本身的故障的一种突出强调，这是符合逻辑的。因此，对于故障报告，应该做的第一件事是使用它修改已有的用例，或从零开始定义一个新的用例，然后才应该开始修改目标代码本身。

执行这些操作后，您完成了几件事情。第一，您给报告的故障下了一个正规的定义。这样就可以就实际中修复故障与其他人达成一致，并且使关于故障是否已经被正确理解的进一步讨论成为可能。第二，通过定义测试装置和测试用例，确保了可以随意重现故障。如果您曾经需要重现那些不可琢磨的故障，那么就会知道仅此一点就可以节省大量时间。最后是这种方法的第三个，可能也是最重要的结果：如果从不修改未被任何测试用例覆盖的代码，那么总是可以确信后来的修改将不会破坏已有的修复。结果就是，用户更加满意，开发人员(包括您)更加轻松。这一切都归功于单元测试。

## 12.6 本章小结

测试最好在开发生命周期的一开始就着手进行。一般而言，如果能够写出问题的一个测试，证明您对问题有了足够的理解，能够对它进行强有力的控制了。

最基本的一类测试是断言。断言就是条件，它们被放置在程序中，用于确认应该存在的条件确实存在。当开发一个程序时，它们用于保证期望的条件得到满足。

如果 Python 以 -O 选项运行，则断言将被关闭。-O 指出您希望 Python 以一种更高效的模式运行，这种模式通常也是运行产品程序的正常方式。这意味着在一个运行的系统中，不应该依赖使用 `assert` 来捕获错误。

在 Python 中，PyUnit 是进行全面测试的一种默认的方法，它使得对测试过程的管理变得非常简单。PyUnit 在 `unittest` 模块中实现。

当使用 PyUnit 创建自己的测试时，PyUnit 提供一些函数和方法，它们对基于一些问题的特定条件进行测试，例如“A 的值是否大于 B 的值”就是一个问题。可以使用 `TestCase` 类中的很多方法，当方法的名称体现的条件失败时，这些方法也会失败。这些方法的名称都以“fail”开头，可以用于建立需要测试的大多数条件。

应该创建 `TestCase` 类的子类——运行测试正是通过调用 `run` 方法来实现的，需要为自

己的测试定制这个方法。另外，如果 `TestCase` 类的 `setUp` 和 `tearDown` 方法被重写，而且代码是为它们指定的，那么就可以在每个测试之前建立测试装置，或测试应该运行的环境。

本章介绍了建立一个测试框架的两种方法。一种是创建自定义类的一个子类，另一种是使用单独的函数实现相同的功能，但是不需要创建子类。应该尝试这两种方法，确定哪一种更符合自己的处理方法。这些测试不需要与模块或程序处于相同的文件中，它们应该被放到其他地方，以使代码不那么臃肿。

当阅读本书剩余部分时，试着考虑对所看到的函数或类编写相应的测试，甚至可以实际编写这些测试。这是一个很好的练习，所以本章没有必要提供习题。

#### 本章要点：

- 断言是在代码中所做的陈述，允许对代码的有效性进行测试。如果测试失败，那么将产生 `AssertionError` 错误。可以使用断言创建一些测试。
- `PyUnit` 是其开发者为这个程序包起的名称，但是导入的模块具有一个听起来更加通用的名称 `unittest`。
- 一个测试套件是一系列测试用例，它们是针对一个特定的项目共同运行的。
- 在 `PyUnit` 中，一个测试用例的运行环境被称为测试装置，基类 `TestCase` 定义了两个方法：`setUp` 在测试运行前被调用，`tearDown` 在测试用例完成后被调用。这两个方法用于处理在测试装置的创建和清理中所涉及的任何工作。

第 13 章将讨论图形用户界面(Graphical User Interface, GUI)编程，并讨论如何创建一些比较简单的、交互的程序。





# 第 13 章

## 使用 Python 编写 GUI

Python 不仅在一些世界上最大、最重要的服务器端应用程序的后台扮演了重要的角色，而且对一些终端用户应用程序也产生了很大的影响。用 C、C++ 以至 Java 或 C# 编写 GUI 是一种非常昂贵、非常痛苦的项目，但是用 Python 却可以非常快速地、简单地完成。即使您只编写一些简单的 Python 脚本，也能够快速完成一个 GUI，它可以成为一个力量倍增器，使不怎么懂技术的人也能使用您的脚本，从而增加它的价值。作为跨平台且真正面向对象的语言，Python 具有的优点使 Visual Basic 程序员都喜欢在他们的快速应用程序开发工具箱中使用它。

Python 允许像使用其他编程语言一样每次布局一个组件。然而，现在程序员都不再以手动方式编写 GUI 代码了。如果已经习惯那种做法，那么请准备好通过 Python 语言的强大能力利用 Delphi 所有的快速应用程序开发工具。当然，其他很多语言中也具有这种能力，例如 C#。

本章将介绍 Python 中 GUI 编程的基本知识。对创建 GUI 应用程序的全面介绍需要整本书的内容，本书仅简单地对 GUI 编程进行了介绍。然而，这已经足以使您编写出一些优雅的、交互的、用户友好的应用程序了。

本章将介绍：

- 创建一些小组件，例如标签、命令按钮、单选按钮和复选框。
- 布局图形用户界面。
- 定制小组件的外观。
- 创建自定义对话框。
- 理解填充顺序。
- 在小组件中插入一些函数。

### 13.1 Python 的 GUI 编程工具箱

很多不同的工具箱都支持用 Python 编写 GUI 程序：可以在 [www.python.org/moin/GuiProgramming](http://www.python.org/moin/GuiProgramming) 上找到很多这样的工具并进行尝试。这些工具箱是与使用 C/C++ 编写的本地 GUI 代码进行协作的 Python 二进制模块，它们具有不同的 API，提供不同的功能集。其中只有一个是 Python 默认提供的工具箱，即 TK GUI 工具箱。如果正在使用 Windows，那

么总是可以直接安装 win32all 并使用 Win32 API。真正的勇敢者将使用 pyGame 编写整个 GUI，并将声音添加到每个滑块中。

其他的选择包括 wxPython、PyQT 和 pyGTK。它们在很多方面是有区别的，其中许可证是一个非常重要的方面。PyQT 的网页中说明了在试图创建某类应用程序或库时，许可证问题将会对您所做出的决定施加那些限制。可以从下面这段话中看到：

“PyQt 是在 GNU GPL 的授权下(适用于 UNIX/Linux 和 MacOS/X)，在 QT 非商用许可证的授权下(适用于 Windows 下的 QT v2.3.0 非商用版本)，在 QT 教育许可证的授权下(适用于 Windows 下的 QT 教育版)，以及在商用许可证的授权下(适用于 Windows、UNIX/Linux 和 MacOS/X)……”

接着是：

“当部署商业版 PyQt 应用程序时，不鼓励用户自己访问底层的 PyQt 模块是必要的。一个使用您的应用程序中的模块去开发新的应用程序的用户，将被认为是开发者，他们需要拥有自己的商用 QT 和 PyQt 许可证。”

“这个问题的一个解决方法是 VenderID 程序包。它使您能够创建一些 Python 扩展模块，这些扩展模块只能被数字签名的自定义解释程序导入。VenderID 程序包使您能够创建这样的一个解释程序，并将您的应用程序包含其中。结果得到一个只能运行您的应用程序的解释程序，和只能被该解释程序导入的 PyQt 模块。可以使用这个程序包以类似方式限制对任何扩展模块的访问。”

可以看到，除非有一个非常好的理由，否则仅仅因为这一部分许可规定，您就很可能将整个 QT 工具集从考虑中排除。任何一个正常人都不愿意处理那种令人困惑的许可证问题。QT 开发者会声称他们工具箱的优点要远远超过那些使用 Windows 的少数人所需要支付的授权费用。如果您同意这种说法，那么就小心翼翼地踏入他们的授权“雷区”中吧。大多数人则是非常简单地选择不考虑它。

一个开源选择是 wxPython。WxPython 基于 wxWidgets，是一个可移植的(Windows、Linux 和 Mac OS X)图形工具箱，它具有悠久的历史，并具有本地代码相同的外观和运行方式。在 <http://wiki.wxpython.org/index.cgi/FrontPage> 上可以找到关于 wxPython 的最详细的信息。

创建 GUI 的新手对 wxPython 可能会不知所措。尽管邮件列表和专业组织都提供了良好的用户支持，wxPython 库依然令人畏惧。然而，对于愿意挑战困难的人来说它是一个很好的选择。在撰写本书时，并不是所有这些程序都支持 Python 3.1，不过大多数程序都承诺会支持。

本章的剩余部分将使用 Tkinter。Tkinter 是 Python 的标准 GUI 程序包，它与 Python 一起安装。它可能是使用最广泛的 GUI 编程工具箱，而且可移植到几乎所有的平台上。关于它的更多信息，可以访问 <http://tkinter.unpythonic.net/wiki/>。也可以访问它的主页 <http://wiki.python.org/moin/TkInter>。

## 13.2 Tkinter 简介

GUI 并不像它们看起来那么简单。然而，一旦理解了一些基本的概念，您将会发现它们是可理解的，而且适当的程序设计有助于绕过主要的障碍。

并不是所有的 Tkinter 应用程序都必须很复杂。您的应用程序可能是一个简单的对话框，用于自动化一个经常执行的业务过程。使用 Tkinter 可以快速地、方便地编写出一些像 CANVAS、DashBoard 和 PythonCAD 一样的大型应用程序，它使编写那些简单的应用程序变得微不足道。Tkinter 本身已经用于制作许多流行的程序，其中最著名的就是 IDLE 本身。

## 13.3 用 Tkinter 创建 GUI 小组件

首先需要理解的是，大多数 GUI 框架(包括 Tkinter)是基于一个小组件(widget)模型的。小组件是 GUI 的一个组件，如按钮、标签和文本框都是小组件。大多数小组件在屏幕上都有图形显示，但是有些小组件只是用来容纳其他小组件并将它们布置到屏幕上的，例如表和框。GUI 是对若干小组件进行布置而构成的。在下面的部分中，您通过定义一些小组件并对它们进行排列，创建一个简单的 GUI。

### 试一试

#### 编写一个简单的 Tkinter 程序

有了 Tkinter，就可以开始编写一个实际的 GUI 应用程序。这个 MyFirstGUI 脚本创建了包含一个简单的窗口和一个标签的 GUI。窗口中的标签显示一条消息：

```
import tkinter
from tkinter import *
widget = Label(None, text='This is my first Gui!!')      # create a label
widget.pack()
widget.mainloop()
```

运行这个程序，则会在窗口中看到 “This is my first Gui!!” 标签，如图 13-1 所示。

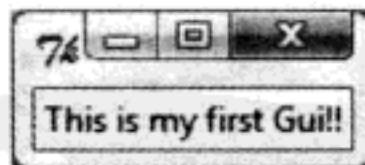


图 13-1

### 示例说明

要做的第一件事是导入 Tkinter 模块。然后，可以选择从 Tkinter 导入 Label 类，或者像示例中那样，简单地从 Tkinter 导入所有的类(用\*表示)。之后，为每个小组件(在这里是 Label)创建一个对象。然后将标签布置到父窗口中。最后，将小组件显示出来。

这个脚本的一个问题是，与大多数 GUI 应用程序不同，它实际上不完成任何操作。其原因很简单：这段脚本不处理任何 GUI 事件。但是别担心，本章的后面会解决这个问题。

### 13.3.1 改变小组件的尺寸

关于所创建的窗口，有一些地方值得注意。首先，它有一个内置的最小化和最大化按

钮，和一个关闭窗口的“X”按钮。另外，用户可以拉伸窗口。请按住窗口的右边并将其向右拉几英寸。可以看到标签仍留在顶部附近的中央位置。这种情况很好。

现在，请按住窗口的底部并将其拉伸。看出问题了吗？无论把窗口向下拉多远，标签依然留在顶部。理想情况下，当用户改变一个窗口的大小或形状时，希望窗口内的小组件会具有适当的行为。为此，对代码进行修改，使得当窗口改变大小时标签仍然可以居中：

```
import tkinter
from tkinter import *
Label(text='My first GUI!').pack(expand=YES, fill=BOTH)
mainloop()
```

当运行上面的程序时，试着改变窗口的大小，可以看到无论窗口变成什么样子，“My first GUI!” 标签保持在中心位置，如图 13-2 所示。

**注意：**

在这个示例中，再次从 Tkinter 导入所有的类，然后创建了一个带有一些文本的标签。之后，将值 Yes 赋值给 expand，将值 Both 赋值给 fill。这告诉 Python，当父窗口扩展时小组件也扩展。默认情况下，这个选项是关闭的。



图 13-2

### 13.3.2 配置小组件选项

您不只看到了如何给标签指派文本，还看到了如何配置小组件的某些选项，例如扩展和填充。尽管使用的方法在这个示例中很方便，但是您可能发现自己希望先创建一些小组件，然后在以后配置它们的选项。

#### 试一试

#### 配置小组件

在这个示例中将创建相同的父窗口和相同的标签。然而，不是在创建它们的同时设置选项，而是等到它们已经被创建之后再行设置：

```
import tkinter
from tkinter import *
root = Tk()

widget = Label(root)
widget.config(text='My first GUI!')
widget.pack(side=TOP, expand=YES, fill=BOTH)
root.mainloop()
```

在此示例中，通过调用配置方法实现了与前一个示例相同的结果。如果愿意，可以在程序中较晚地改变小组件的外观。例如，可能用户希望改变窗口的显示方式。可以通过插入一个按钮来触发该配置方法，而后该方法将改变小组件的选项。

### 13.3.3 使用小组件

现在您已经看到了如何创建一个基本的标签，以及如何格式化小组件。但是对于一个成功的程序来说，它不仅要具有良好的外观，还要实现一些实际操作。下面几个示例不仅演示如何向 GUI 中添加多个小组件，而且教您在那些小组件上执行操作。更加重要的是，您将学到如何使程序对用户的各种操作做出反应。

到目前为止，您已经学会了如何创建一种小组件，即标签。在这个示例中将创建一个按钮。试着将如下代码输入到一个名为 `MyFirstButton.py` 的文件中：

```
import sys
from tkinter import *
widget = Button(None, text='Click Me', command=sys.exit)
widget.pack()
widget.mainloop()
```

这段代码简短而易懂，它与前面的代码看起来非常类似。然而需要注意，除了改变了小组件的类型之外，也添加了一个新的选项，即 `command`。赋给 `command` 的值 `sys.exit` 向 Python 表明，当用户单击此按钮小组件时，程序将完全退出。

但是，如果要添加退出程序以外的选项，应该怎么办？要实现这一点，需要向窗口中添加第二个小组件。创建一个新文件，并将它命名为 `MultipleWidget.py`：

```
from tkinter import *

def result():
    print('The sum of 2+2 is ',2+2)

win = Frame()
win.pack()
Label(win, text='Click Add to get the sum or Quit to Exit').pack(side=TOP)
Button(win, text='Add', command=result).pack(side=LEFT)
Button(win, text='Quit', command=win.quit).pack(side=RIGHT)

win.mainloop()
```

这段代码引入了几个尚未介绍的新概念。首先，创建一个用户自定义函数 `result()`，而且将打印一些文本并返回 `2+2` 之和的任务分配给它。然后，创建 3 个小组件。第一个小组件是保存一些文本的标签，对它设置 `side=TOP` 选项。然后，创建一个调用用户自定义函数的按钮，并将其放置于窗口的左边。最后，在右侧创建一个按钮，并将 `win.quit` 赋值给其 `command` 选项。

如果用户单击 `Add` 按钮，标准输出中将会显示文本 “The sum of 2+2 is 4”。如果用户单击 `Quit` 按钮，窗口将被关闭。

这里需要注意的另外一个问题是，我们已经首次介绍了框架小组件的使用。

### 13.3.4 创建布局

在创建 GUI 时，考虑各个小组件间的层次结构是很重要的。这种层次结构一般称为父

子关系。在前面的示例中创建了很多小组件。第一个小组件是顶级的窗口，它起父小组件的作用。然后，创建了一个称作 `win` 的小组件，它自己拥有一个子小组件，是一个框架小组件。此时，`win` 小组件是顶级窗口的子小组件。

接下来，创建了一个标签和两个按钮，它们都是框架小组件的子小组件。框架小组件是一种以保存其他小组件为目的的小组件，它们给程序员提供了灵活性，使他们能够创建一个布局，并决定每个小组件应该出现在窗口中的具体位置。随着对 GUI 编程的深入，您将会使用很多不同的框架小组件，每个框架小组件在顶级的窗口中占据一个特定的位置，并且每个框架都有自己的一组小组件。属于每个框架的这些组件是它们各自框架的子小组件，将被限制在其父框架提供给它们的空间中。

例如，如果有两个大小相同的框架，每个占据窗口的一半，并给每个框架指派一个按钮小组件，指派给左侧框架的按钮将只能放置于窗口的左边。同样，指派给右侧框架的按钮将被约束在右边。如果将左侧框架中的按钮填充到右边的框架中，显示给用户的结果将是它位于顶级窗口的中央。

### 13.3.5 填充顺序

布局的另外一个重要的方面是填充顺序。当创建每个小组件并对它们进行填充时，它们便获得了其所在区域的所有空间。例如，如果在左边填充一个按钮，它将占据左边的所有空间。当创建第二个小组件并将它也填充到左边时，最初的按钮会收缩，但是仍占据最左侧的空间。这个过程一直继续，每个按钮都通过收缩来为其他小组件提供空间。然而，这些按钮绝不会移出它们最原始的空间。填充到左侧的第一个按钮将一直在最左边，同样地，填充到左侧的第二个按钮将一直是最靠近左边的第二个按钮。

虽然这听起来难以理解，但是对以前的代码进行简单的重新安排将会使这个问题变得清楚一些：

```
from tkinter import *

def result():
    print('The sum of 2+2 is ',2+2)

win = Frame()
win.pack()
Button(win, text='Add', command=result).pack(side=LEFT)
Label(win, text='Click Add to get the sum or Quit to Exit').pack(side=TOP)
Button(win, text='Quit', command=win.quit).pack(side=RIGHT)

win.mainloop()
```

图 13-3 显示了在修改代码之前程序的外观。图 13-4 显示了修改代码之后程序的外观。

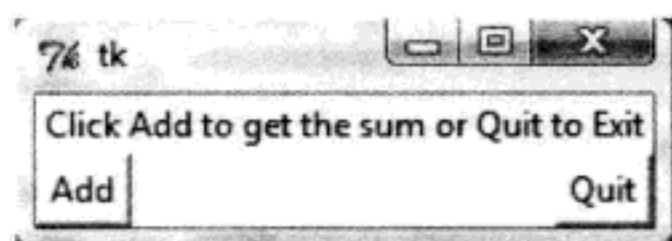


图 13-3

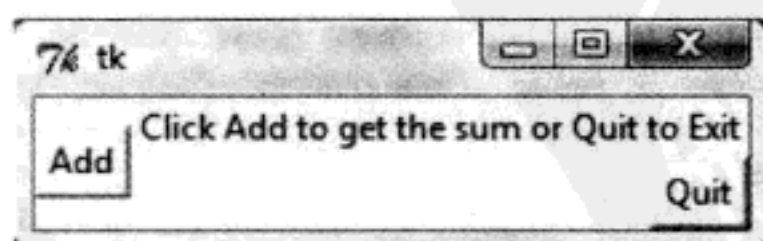


图 13-4

13.3.6 控制小组件的外观

窗体和函数是创建一个匀称的 GUI 的关键。前面已经学习了如何向界面中添加一些代码，以及布局的基本知识(还有很多知识需要了解，但是这超出了本章的讨论范围)。在本节中将使程序更进一步，并且将介绍如何控制小组件的外观。

试一试

配置小组件

直到现在，一直使用小组件的默认外观，它们是非常单调的。为了保持用户的注意力并创建在视觉上吸引人的程序，需要调整小组件的外观。请尝试如下代码：

```
import tkinter
from tkinter import *
root = Tk( )
labelfont = ('times', 24, 'italic')           # setting the family, size, and
                                              style
widget = Label(root, text='Eat At JOES')      # setting label text
widget.config(bg='black', fg='red')           # setting the back and foreground
                                              colors
widget.pack(expand=YES, fill=BOTH)
root.mainloop( )
```

运行该程序并观察其结果。虽然这个程序实际上不做任何操作，但是它的设计确实是引人注意的。

表 13-1 中列出了定制小组件的不同方法：

表 13-1

属 性	说 明
border、relief	Border 设置边框宽度(如 bd=1)。 Relief 用于确定边框样式(如 relief=raised)
color	bg 设置背景颜色，fg 设置前景颜色。可以使用简单的颜色名称或使用十六进制的颜色代码
cursor	设置光标悬浮在小组件之上时显示的形状(如 cursor=cross)
font	设置字体系列、尺寸和样式(如 Times、24、italic bold underline)
padding	设置一个小组件周围的空距
pack 、expand 和 fill	如之前的示例所示
state	设置对象的状态(如 state=DISABLED)
size	控制小组件尺寸的高度和宽度设置，允许它比 Tkinter 几何管理器设置的尺寸更大

### 13.3.7 单选按钮和复选框

现在，您已经使用了如下的小组件：顶级窗口、框架、标签和按钮。它们都是功能强大的小组件，但是有时候可能希望给用户提供更多的选项。这时候单选按钮和复选框便发挥作用了。

除了外观之外，单选按钮和复选框还有一个重要的区别：单选按钮为用户提供一个选项列表，但是只允许选择其中的一个；复选框为用户提供很多选项，并且允许他们选择任意多个选项。

#### 试一试

#### 创建单选按钮

创建一个名为 `MyRadio.py` 的新文件，并在其中包含如下代码：

```
import tkinter
from tkinter import *
state = ''
buttons = []

def choose(i):
    global state
    state = i
    for btn in buttons:
        btn.deselect()
    buttons[i].select()

root = Tk()
for i in range(4):
    radio = Radiobutton(root, text=str(i),
                        value=str(i), command=(lambda i=i: choose(i)))
    radio.pack(side=BOTTOM)
    buttons.append(radio)
root.mainloop()
print("You chose the following number: ",state)
```

这个程序创建了一系列按钮，其编号范围是 0~3(总共 4 个)，1、2、3 号按钮默认为突出显示。用户可以选择这些按钮中的任何一个。一旦一个按钮被选择，其他任何按钮的状态就变成 `False`，意味着它们未被选中。当用户关闭程序时，会向用户显示一条语句，说明他们选择了哪个号码。

可以使用复选框实现类似的功能。然而在下面这个示例中，只是简单地返回每个复选框的值是 `True` 还是 `False`：

```
from tkinter import *
states = []
def check(i):
    states[i] = not states[i]
```

```

root = Tk( )
for i in range(4):
    test = Checkbutton(root, text=str(i), command=(lambda i=i: check(i)) )
    test.pack(side=TOP)
    states.append(0)
root.mainloop( )
print(states)

```

如果运行上面的程序并选中复选框 2 和 3，那么关闭窗口后，结果应该是：

```

>>>
[0, True, True, 0]

```

### 13.3.8 对话框

有时候希望为用户提供一些额外的信息。您一定很熟悉对话框。当有错误出现时，或者当程序希望确认一些事情，例如您确实希望卸载一个程序，或者您希望要显示一些敏感信息时，它们就会弹出。

Tkinter 提供两类对话框：模态和非模态。模态对话框在用户执行某些操作前一直显示，并暂停程序的执行。非模态对话框不中断程序的流程。

#### 试一试

#### 创建自定义对话框

可以使用几种方法创建对话框。在这个“试一试”中，使用 Toplevel 创建了一个自定义对话框。创建一个名为 MyPopUp.py 的新文件，并输入如下文本：

```

import sys
from tkinter import *
popupper = (len(sys.argv) > 1)

def dialog( ):
    win = Toplevel( )
    Label(win, text='Do You Always Do What You Are Told?').pack( )
    Button(win, text='Now click this one', command=win.destroy).pack( )
    if popupper:
        win.focus_set( )
        win.grab_set( )
        win.wait_window( )
    print('You better obey me...')

root = Tk( )
Button(root, text='Click Me', command=dialog).pack( )
root.mainloop( )

```

当运行该程序时，首先得到一个弹出窗口，其中简单地显示“Click Me”。单击该窗口后，会发生两件事。首先，向标准输出写入一些文本(也就是“You better obey me...”)。其

次，会出现另外一个弹出窗口，它得到焦点并要求您也单击它。单击后，它关闭了自己，原来的弹出窗口则重新得到了焦点。

当谈到创建对话框时，这仅仅是冰山一角。除了警告、错误消息和确认之外，对话框也可以用于加载文件(考虑一下在大多数应用程序中使用的打开命令)、从色轮中选择一个颜色等很多种用途。

13.3.9 其他小组件类型

本章介绍了很多小组件类型，但是还有很多小组件需要探索。对它们进行完整地介绍超出了本章范围，实际上也超出了本书的范围，但至少应该知道它们的存在。表 13-2 列出了一些可用的、不同的小组件类，并对每一个小组件类做出了相应的解释。

表 13-2

小 组 件 类	说 明
BitMapImage	允许在其他小组件上部展现位图图像的一种小组件对象
Button	一种“可单击”的按钮
Canvas	用于显示一些图形的对象，图形包括圆形、线条、图片、文本等
Checkbutton	包含两种状态 True 和 False 的按钮小组件。它给用户提供多种选择
Entry	由一行文本组成的文本输入字段
Frame	保存其他小组件的一种容器
Label	可以在其上写文本或“标签”的小组件
Listbox	保存可选择的数据的列表框
Menu	选项集合
Menubutton	用于显示另一个选项菜单和子菜单的菜单
Message	类似于一个标签，只用于多行文本
PhotoImage	与 BitMapImage 相同，只用于全色图像
Radiobutton	状态为 True 或 False 的按钮，用于多选择情况
Scale	向上和向下滑动的小组件，并允许用户以这种方式选择选项
Scrollbar	允许滚动其他小组件的一种小组件
Text	用于处理多行文本的一种浏览/编辑小组件
Toplevel, Tk	一个新窗口

13.4 本章小结

使用 Tkinter 可以实现与 GUI 相关的任何操作。例如，可以抓取屏幕截屏、显示图形、处理大规模窗口中的复杂信息集合、在一个空白的画布上绘制或者简单地为自定义命令行实用程序弹出若干快速的 GUI，将它们显示给没有技术基础的用户。

当然，每个编程项目都有其自身的风格。很多人已经开发出可以自动进行应用程序开发的工具。Python 的众多自省和面向对象(Object-Oriented, OO)的特性使程序员能够对 GUI 中的各种更改进行动态地处理。在更加熟悉 Tkinter 后，您将发现这些技术都是很自然的。

即使不使用 Tkinter，理解 Tkinter 的工作原理也将对您的编程提供极大的帮助。而且，在有 15 分钟的空闲时间时，您可能希望为朋友们写一个自定义 GUI 的聊天客户端程序。

## 13.5 习题

1. 使用不同的填充顺序试验不同的布局。
2. 练习通过修改小组件的每个属性改变其外观。





# 第 14 章

## 访问数据库

几乎所有的大规模商业系统都使用数据库来存储数据。例如，在线零售商 Amazon.com 就需要数据库来存储销售的每件产品的信息。Python 要想证明自己能够处理这一类企业应用程序，就必须能够访问数据库。

幸运的是，Python 提供了一套数据库 API(实现数据库编程的方法)，它是一种通用的 API，使您可以访问大多数数据库，而不必考虑这些数据本身的那些互不相同的本地 API。这些数据库(DB)的 API 并没有定义使用数据库的所有方面，因此它们存在一些细微的差别。然而，在大多数情形下，可以从 Python 脚本访问一些数据库，例如 Oracle 或者 MySQL，而不必过分担心特定数据库的细节。

拥有一个通用的数据库 API 是非常有用的，因为您可能需要切换数据库或者让应用程序使用多种数据库，同时并不希望对代码的主要部分进行修改。一般来说，可以使用 Python 实现这个目的，并不需要对程序进行太多修改。

即使您并不是在编写下一个 amazon.com 网站，数据库也提供了一种便捷的方法，使数据的存在时间比正在运行的程序更久(这样如果希望重新启动该程序，那么用户已经输入的数据将不会丢失)，并且使查询数据项和以一种安全的方式修改数据变得更简单。

本章将介绍 Python 支持的两个主要数据库系统：dbm 持久字典和使用 DB API 的关系数据库。另外，本章还描述了如何安装一个数据库。

本章将介绍：

- 使用 dbm 库创建持久字典。
- 关于关系数据库。
- 安装 Sqlite 数据库。
- 安装 MySQL 数据库。
- 使用 Python DB API。
- 创建连接。
- 使用游标访问数据。
- 连接到数据库。
- 查询及修改数据。
- 使用事务。
- 错误处理。
- 使用其他数据库工具。



在很多情况下，并不需要一个成熟的关系数据库。在这种情况下，使用 dbm 文件创建一个持久字典就足够了。

### 14.1 使用 dbm 持久字典

一个持久字典的行为正如您所预期的一样。可以用持久字典来存储名称/值对，它们被保存在磁盘上，因此在程序的多次运行中，它们的数据得以保持。因此如果将数据存储到一个 dbm 支持的字典中，那么当下一次启动该程序时，一旦加载了该 dbm 文件，就可以再次读取存储在指定键下的值。这些字典和第 3 章中讲到的标准 Python 字典非常类似。主要的区别在于数据是在磁盘上写入和读取的。

**注意：**  
另一个不同之处是键和值都必须是字符串类型。

DBM 是 database manager 的缩写，它是原来在 UNIX 系统中创建的许多 C 语言库的一个通用名称。这些库的名称有 dbm、gdbm、ndbm、sdbm 等等。这些名称与在 Python 中已有的、提供必要功能的模块密切地对应。

#### 14.1.1 选择 dbm 模块

Python 支持很多 dbm 模块。每个 dbm 模块支持一个相似的接口，并使用一个特定的 C 语言库向磁盘存储数据。主要的差别在于磁盘上各个数据文件的底层二进制格式。遗憾的是，每个 dbm 模块创建的文件是不兼容的。也就是说，如果使用某个 dbm 模块创建一个 dbm 持久字典，那么必须使用相同的模块来读取数据。其他任何模块都不能处理该数据文件。

表 14-1 中列出了这些 dbm 模块。

表 14-1	
模 块	说 明
dbm	选择最好的 dbm 模块
dbm.dumb	使用 dbm 库的一个简单但可移植的实现
dbm.gnu	使用 GNU dbm 库

由于 dbm 库的历史原因，产生了这些 dbm 模块。起初，这个库只在商用版本的 UNIX 上提供。UNIX 免费版和之后的 Linux、Windows 等系统都不能使用 dbm 库。这导致了各种替代库的产生，例如 Berkeley UNIX 库和 GNU gdbm 库。

由于这些文件格式不兼容，过多的库是非常令人痛苦的。然而，dbm 模块是一种方便的选择，可以避免自己选择特定的 dbm 模块。您可以让这个 dbm 模块替自己做出选择。一般而言，在创建一个新的持久字典时，dbm 模块将选择系统上已有的最好的实现方式。当读取文件时，dbm 模块使用 whichdb 函数针对哪个库创建了该数据文件做出有根据的猜测。

注意：  
除非需要某个 dbm 库的一个特定的高级功能，否则就使用 dbm 模块。

14.1.2 创建持久字典

所有的 dbm 模块都支持使用 open 函数创建一个新的 dbm 对象。一旦成功地打开，便可以在字典中存储数据、读取数据、关闭 dbm 对象(以及相关的数据文件)、移除项和检查字典中是否存在某个键等。

如果要打开一个 dbm 持久字典，使用所选择模块的 open 函数。例如，可以使用 dbm 模块创建持久字典。

试一试

创建持久字典

输入如下代码，并将文件命名为 dbmcreate.py:

```
import dbm

db = dbm.open('websites', 'c')

# Add an item.
db['www.python.org'] = 'Python home page'

print(db['www.python.org'])

# Close and save to disk.
db.close()
```

当执行这段脚本时，将会看到如下输出：

```
$ python dbmcreate.py
b 'Python home page'
```

示例说明

这个示例中使用了推荐的 dbm 模块。

open 函数需要字典的名称来创建新的字典。这个名称被转换为可能已经存在于磁盘上的数据文件的名称(dbm 模块可能——虽然不总是——创建多个文件，通常一个文件用于数据，另一个文件用于键的索引)。该字典的名称被视为一个基本的、包括路径的文件名称。通常，底层的 dbm 库将为数据附加一个后缀，如.dat。您自己可以通过查找以 websites 命名的文件来找到此文件，它很可能就位于当前的工作目录中。

您还应该传递可选的标志。表 14-2 中列出了一些可用的标志。

表 14-2

标 志	用 法
C	打开数据文件以便对其进行读写，必要时创建该文件

(续表)

标 志	用 法
N	打开文件以便对其进行读写,但总是创建一个新的空文件。如果该文件已经存在,则将被覆盖,已有的内容将会丢失
W	打开文件以便对其进行读写,但是如果该文件不存在,那么不会创建它

也可以传递另一个表示模式的可选参数。该模式保存了一组 UNIX 文件权限。更多关于打开文件的内容请参考第 8 章。

dbm 模块的 open 方法返回一个新的 dbm 对象,之后便可以使用此对象存储和检索数据。在打开一个持久字典之后,可以像写入普通的 Python 字典那样写入值,如下面的示例所示:

```
db['www.python.org'] = 'Python home page'
```

键和值都必须是字符串,而不能是其他对象,比如数值或 python 对象。然而,请记住,如果希望保存一个对象,那么可以使用 pickle 模块对它进行串行化。

close 方法关闭文件,并将数据保存到磁盘中。

14.1.3 访问持久字典

使用 dbm 模块时,可以将从 open 函数返回的对象视作一个字典对象。使用如下所示的代码可以实现对值的获取和设置:

```
db['key'] = 'value'
value = db['key']
```

请记住,键和值都必须是文本字符串。

可以使用 del 删除字典中的值:

```
del db['key']
```

keys 方法返回包含所有键的一个列表,方式与处理一个标准字典时相同:

```
for key in db.keys():
    # do something...
```

注意:

如果在文件中包含大量的键,那么 keys 方法的执行可能会占用很长时间。另外,对于较大的文件,此方法可能需要大量的内存来存储它为这些文件创建的较大的列表。

可以将如下的脚本作为一个指导如何使用 dbm 持久字典进行编程的向导。

## 试一试

## 访问持久字典

输入如下的脚本，并将文件命名为 `dbmaccess.py`：

```
import dbm

# Open existing file.
db = dbm.open('websites', 'w')

# Add another item.
db['www.wrox.com'] = 'Wrox home page'

# Verify the previous item remains.
if db['www.python.org'] != None:
    print('Found www.python.org')
else:
    print('Error: Missing item')

# Iterate over the keys. May be slow.
# May use a lot of memory.
for key in db.keys():
    print("Key =",key," value =",db[key])

del db['www.wrox.com']
print("After deleting www.wrox.com, we have:")

for key in db.keys():
    print("Key =",key," value =",db[key])

# Close and save to disk.
db.close()
```

当运行上面的脚本时，将会看到类似于如下所示的输出：

```
$ python dbmaccess.py
Type of dbmfile = dbhash
Found www.python.org
Key = www.wrox.com value = Wrox home page
Key = www.python.org value = Python home page
After deleting www.wrox.com, we have:
Key = www.python.org value = Python home page
```

## 示例说明

这段脚本处理一个包含一些网站的 URL 及其描述的小型数据库。首先要运行前面的 `dbmcreate.py` 示例。该示例创建了 `dbm` 文件，并在文件中存储数据。`dbmaccess.py` 脚本随后会打开已经存在的 `dbm` 文件。

`dbmaccess.py` 脚本以读/写模式打开持久字典 `websites`。如果磁盘上的当前目录中没有

包含必要的数据库文件，那么对 `open` 函数的调用将产生一个错误。

在前面的示例 `dbmcreate.py` 中，字典中会有一个值，以 `www.python.org` 为键。本示例添加 Wrox 的网站 `www.wrox.com` 作为另一个键。

该脚本使用如下代码验证键 `www.python.org` 是否存在于字典中。

```
if db['www.python.org'] != None:
    print('Found www.python.org')
else:
    print('Error: Missing item')
```

然后，该脚本打印出字典中的所有键与值：

```
for key in db.keys():
    print("Key =",key," value =",db[key])
```

请注意，字典中只包含这两个条目。

在打印出所有条目之后，该脚本使用 `del` 方法删除一个条目：

```
del db['www.wrox.com']
```

然后该脚本再次打印出所有的键与值，如输出所示，其结果只包含一个条目。

最后，`close` 方法关闭了这个字典，将对字典的所有修改保存到硬盘上，因此在下次打开此文件时，它将处于关闭它时所处的状态。

可以看到，处理持久字典的 API 极其简单，因为它的工作原理与文件和字典非常类似，而这些您早已非常熟悉了。

#### 14.1.4 dbm 与关系数据库的适用场合

当数据需求可以通过存储一些键/值对来满足时，`dbm` 模块十分有用。可以使用一些想象在键/值对中存储更复杂的数据，例如，对于字典中的键和值部分，都通过创建格式化的字符串，使用逗号或其他字符来划定字符串中各个项的边界。这可能是有用的，但是可能非常难维护，并且因为数据是以一种不灵活的方式存储的，所以对相关的处理会产生一些限制。可能产生的另一种限制是技术方面的：一些 `dbm` 库对可以用于值的空间总数进行了限制(有时达到最大值即 1024 字节，但这仍是非常、非常小的)。

可以使用如下的准则帮助决定这两类数据存储中的哪一种更能满足需求：

- 如果数据需求很简单，那么使用 `dbm` 持久字典。
- 如果计划只存储少量的数据，那么使用 `dbm` 持久字典。
- 如果，需要支持事务，那么使用关系数据库(事务是指多件事同时发生时，它们保证不会发生数据在某一处改变、但未在另一处改变的情况。可以使用事务确定并发生的事件)。
- 如果需要一些复杂的数据结构或多个连接数据的表，那么使用关系数据库。
- 如果需要与一个已有的系统协作，那么很显然使用该系统。这种系统是关系数据库的可能性很大。

不像那些简单的 dbm 模块，关系数据库提供了更加丰富和更加复杂的 API。

其实还有另一种数据库可供使用，虽然这超出了本章的范围。这种数据库是 ORM 或对象-关系数据库，它允许数据在不与关系型数据库兼容的各个类型系统间相互转换。

如果希望使用 ORM，Python 提供了几种选择，例如 SQL Object、SQLAlchemy 甚至 Django ORM。对于更多的信息，请访问 Python Wiki，网址为：<http://wiki.python.org/moin/HigherLevelDatabaseProgramming>。

## 14.2 使用关系数据库

关系数据库已经产生几十年了，因此它们是一种非常成熟和知名的技术。使用关系数据库的人们知道它们的功能和工作方式，因此关系数据库是一种进行复杂数据存储时首选的技术。

在一个关系数据库中，数据存储在各个表中，表可以被看做二维数据结构。每个列，或者说二维矩阵中垂直的部分，都具有相同的数据类型，比如字符串、数值、日期等。表的每个水平部分由一些行组成，这些行也被称作记录。每行又由列组成。通常，每个记录保存与一个条目有关的信息，例如一个音频 CD、一个人、一个订购单、一辆汽车等。

例如，表 14-3 显示了一个简单的雇员表。

表 14-3

empid	firstname	lastname	department	manager	phone
105	Peter	Tosh	2	45	555-5555
201	Bob	Marley	1	36	555-5551

这个表包含 6 列：

- **empid**: 保存雇员的 ID 号。关系数据库广泛使用 ID 号，由数据库管理唯一号码的分配，以便可以通过这些号码对每行进行引用，以保证每行都是唯一的(即使它们具有相同的数据)。然后便可以通过 ID 号引用每一个雇员。仅仅 ID 号就提供了用于查找雇员的足够信息。
- **firstname**: 保存雇员的名。
- **lastname**: 保存雇员的姓。
- **department**: 保存雇员所在工作部门的 ID 号。它可能是某个部门的一个数值 ID，而各个部门在一个单独的表中定义，每个部门具有唯一的 ID 号。
- **manager**: 保存给定雇员的经理的雇员 ID 号。因为在此示例中，经理实际上也是一个雇员，所以这是一种自我引用。
- **phone**: 保存办公电话号码。

在现实生活中，一个公司可能会保存关于雇员的一些更多的信息，例如税务机关识别号码(在美国是社会安全号码)、家庭住址等，但是原理上与您已经看到的没有什么区别。

在此示例中，列 empid 是雇员的 ID 号，用作主键。一个主键是对一个表的一个唯一

索引，其中每个元素必须是唯一的，因为数据库可能使用那个元素作为给定行的键以及作为引用该行中数据的方式，这是一种与 Python 字典中的键和值类似的方式。因此，每个雇员需要有一个唯一的 ID 号，而且一旦有了 ID 号，便可以查找任何雇员。因此，`empid` 将作为此表的内容的键。

`department` 列保存一个部门的 ID 号，这是另一个表中一行的 ID。因为这个 ID 作为另一个表的一个键，所以可以将这个 ID 看做一个外键(在数据库中，外键具有更加严格的定义，所以这样想是没关系的)。

例如，表 14-4 中显示了部门表的一个可能的布局。

表 14-4		
department id	name	manager
1	Development	47
2	QA	32

在这些示例中，雇员 Peter Tosh 在部门 2 工作，即在一个有实力的世界级高质量软件开发公司中的质量保证(Quality Assurance, QA)部门工作。Bob Marley 在部门 1 工作，即在开发部门工作。

在一个大公司中，数据库中可能有数百个表，某些表中有数千个甚至数百万个记录。

14.2.1 编写 SQL 语句

结构化查询语言(Structured Query Language, SQL)定义了用于查询和修改数据库的一种标准语言。

注意：  
可以将 SQL 读做 “sequel” 或 “s-q-l”。

SQL 支持在表 14-5 中列出的一些基本操作。

表 14-5	
操 作	用 法
Select	执行一个查询，以在数据库中搜索指定的数据
Update	修改一行或若干行，通常是根据特定的条件进行修改
Insert	在数据库中创建新行
Delete	从数据库中删除行

一般将这些基本的操作称为 QUID(Query、Update、Insert 和 Delete 缩写)或 CRUD(Create、Read、Update 和 Delete 的缩写)。SQL 还提供了其他操作，但是在大多数情况下，它们是在编写应用程序的过程中使用最多的操作。

**注意：**

如果不熟悉 SQL，请查阅 SQL 书籍或在 Internet 上进行相关搜索。您会找到大量的教程资料。也可以访问本书的网站来获得关于 SQL 资源的更多参考。

SQL 是非常重要的，因为当使用 Python DB API 访问数据库时，必须首先创建一些 SQL 语句，然后通过让数据库对它们进行评估来执行这些语句。接着可以检索结果并使用它们。这样，您会发现自己处于一种尴尬的境地，要使用一种语言(Python)创建另一种语言(SQL)的命令。

CRUD 操作的基本 SQL 语法如下：

```
SELECT columns FROM tables WHERE condition ORDER BY columns  
ascending_or_descending
```

```
UPDATE table SET new values WHERE condition
```

```
INSERT INTO table (columns) VALUES (values)
```

```
DELETE FROM table WHERE condition
```

除了现有语法的这种基本写法之外，每个操作还有其他很多可选的参数和说明符。如果熟悉 SQL，仍然可以通过 Python 的 DB API 使用它们。

使用之前的雇员示例，为了在雇员表中插入一个新行，可以使用类似于如下代码的 SQL 查询(虽然它是添加数据而不是获取数据，但是根据约定，所有的 SQL 命令或语句都被称为查询)：

```
insert into employee (empid, firstname, lastname, manager, dept, phone)  
values (3, 'Bunny', 'Wailer', 2, 2, '555-5553')
```

在此示例中，第一个元组(用 Python 术语考虑是很有用的，即使 SQL 会赋予它们不同的名称)以插入数据的次序保存这些列的名称。第二个元组在关键字 `values` 之后，以相同的次序保存数据项。注意 SQL 使用单引号限定字符串，并且在数值周围没有引号(电话号码是不同的，它实际上是一个字符串，因为它需要能够包含一些非数值，比如不同的数据输入方式中用到的短线、句点和加号)。

对于查询，可以使用一些捷径，例如使用 `*` 表示希望使用一个表中的所有列执行一个操作。例如，要查询部门表中的所有行，并显示每行的所有列，可以使用类似下面的查询语句：

```
select * from department
```

**注意：**

SQL 对于其关键字是不区分大小写的，例如 `SELECT` 和 `FROM`。但是，有些数据库要求表和列的名称都是大写的。因此，开发人员经常使用大写字母表示操作，例如 `SELECT` 和 `FROM`，以将它们与查询语句的其他部分方便地区分开。

此 SQL 语句省略了要读取的列的名称以及可以缩减返回的数据量的任何条件。因此，此查询将会返回所有的列(通过\*)和所有的行(因为没有 where 子句)。

对于 select 命令，可以执行一个联接来实现从多个表中查询数据，但是将数据显示在一个响应中。因为将会返回来自两个表的数据，就仿佛是从单个表中查询到的，所以将它称为联接。例如，对于每个雇员，要提取其所在部门的名称，可以执行一个类似下面的查询语句(作为一个单一的查询，此语句的所有部分需要包含在一个字符串中)：

```
select employee.firstname, employee.lastname, department.name
from employee, department
where employee.dept = department.departmentid
order by lastname desc
```

在此示例中，select 语句请求来自雇员表的两列(firstname 和 lastname 列，但是按照指定表中的表名称和列名称的约定，这些列被指定为来自于 employee)和来自部门表的一列(department.name)。语句的 order by 部分向数据库表明按照 lastname 列中的值对结果集进行降序排序。

为了简化这些查询，可以使用表名称的别名，从而简化输入和阅读(但是不要改变查询语句的逻辑或语法)。例如，要使用 e 作为雇员表的别名，可以启动一个如下的查询语句：

```
select e.firstname, e.lastname
from employee e
...
```

在本例中，在 from 子句部分，必须将别名 e 放置于表的名称之后。也可以使用可选关键字 as 以采用如下的格式，这样更有助于理解。

```
select e.firstname, e.lastname
from employee as e
...
```

使用如下的一条 SQL 语句修改(或更新)一行：

```
update employee set manager=55 where empid=3
```

此示例对 ID 为 3 的雇员进行修改，将此雇员的经理设置为 ID 为 55 的雇员。与其他查询一样，数值的周围不需要有引号，然而，字符串需要放在单引号内。

使用如下的一条 SQL 语句删除一行：

```
delete employee where empid=42
```

此示例删除了 ID 为 42 的雇员，但是不影响数据库中的其他数据。

### 14.2.2 定义表

当第一次建立一个数据库时，需要定义一些表和它们之间的关系。为此，可以使用 SQL

中称为数据定义语言(Data Definition Language, DDL)的部分(它定义了各个表的结构)。DDL 的基本原理是相当简单的, 即使用一个操作创建表, 使用另一个操作删除它们:

```
CREATE TABLE tablename (column, type column type, ... )  
DROP TABLE tablename
```

还存在一个 ALTER TABLE 命令用于修改一个已有的表, 但是现在还不需要这样做。当希望使用这个命令时, 可以查阅专门介绍 SQL 的书籍或网页。

遗憾的是, SQL 并不是完全标准的语言, 而且每个数据库对它的某些部分的处理是不同的。DDL 就属于还没有被标准化的 SQL 部分。因此, 当定义表时, 尽管基本的概念是相同的, 您仍将会发现不同的数据库所支持的各个 SQL 间的区别。

### 14.2.3 建立数据库

在大多数情况下, 程序员已经有了一个已启动和运行的数据库, 甚至可能是一个由其他某个组织选择的数据库。例如, 如果在一个提供诸如数据库等附属配件的网站托管公司托管一个网站, 托管程序包可能包括了对 MySQL 数据库的访问。如果您受雇于某一个大型组织, 那么 IT 部门可能已经对使用的数据库进行了标准化, 例如统一采用 Oracle、DB/2、Sybase 或 Informix 等。如果使用 Python 创建企业级的应用程序, 那么在工作区中后面的这些程序包可能已经有了。

如果还没有任何数据库, 但仍然希望继续运行本章的示例, 那么使用 Sqlite 数据库将是一个很好的开端。Sqlite 的主要优点是它随 Python 一起安装, 简单、轻便, 但实用。这使得它成为在学习中进行试验的一个出色的帮手, 即使可以选择使用另外一个数据库。请记住, 每个数据库都有它自己的特殊之处。

注意:

编写本章这些示例的目的是使用 Sqlite, 以便您不需要任何外部基础结构就可以运行它们。然而, 可以很容易地对这些示例进行修改, 以使用一个不同的数据库。这正是 Python DB API 的一个重要特点。

使用 Sqlite 与导入模块一样简单。如下示例说明了创建一个数据库所需要完成的所有工作。

如果使用另一个数据库, 例如 SQL Server, 那么很可能已经创建了数据库。如果没有, 那么请按照数据库提供商的指示(很多时候, 可以从数据库管理员或者叫 DBA 那里得到类似这种任务的帮助, 他们更希望您在一个测试数据库而不是一个产品数据库上进行工作)。

使用 Sqlite 创建一个数据库是非常简单的。

#### 试一试

#### 创建一个 Sqlite3 数据库

输入如下脚本, 并将此文件命名为 createdb.py:

```

import os
import sqlite3
conn=sqlite3.connect('sample_database')
cursor=conn.cursor()
# Create tables.
cursor.execute("""
create table employee
(empid integer,
firstname varchar,
lastname varchar,
dept integer,
manager integer,
phone varchar)
""")
cursor.execute("""
create table department
(departmentid integer,
name varchar,
manager integer)
""")
cursor.execute("""
create table user
(userid integer,
username varchar,
employeeid integer)
""")
# Create indices.
cursor.execute("""create index userid on user (userid)""")
cursor.execute("""create index empid on employee (empid)""")
cursor.execute("""create index deptid on department (departmentid)""")
cursor.execute("""create index deptfk on employee (dept)""")
cursor.execute("""create index mgr on employee (manager)""")
cursor.execute("""create index emplid on user (employeeid)""")
cursor.execute("""create index deptmgr on department (manager)""")
conn.commit()
cursor.close()
conn.close()

```

当运行这段脚本时，除非脚本产生一个错误，否则将不会看到任何输出。

### 示例说明

除了标准的 Python DB API 外，Sqlite 还具有其自身的 API。此脚本使用 Sqlite API，但是您将会注意到这个 API 与下一节“使用 Python 的数据库 API”中介绍的 DB API 是非常类似的。本节主要描述了在 `createdb.py` 脚本中的 Sqlite 特有代码。

如下所示是使用 Sqlite 模块创建一个 `connection` 对象的代码：

```
conn=sqlite3.connect('sample_database')
```

脚本从中得到一个 `cursor` 对象(将在 14.3.3 节介绍)。Cursor 对象用于创建 3 个表，并

定义这些表上的索引。

此脚本调用该连接的 `commit` 方法以将所有修改保存到磁盘上。

Sqlite 将其所有数据保存到文件 `sample_database` 中。在运行 `createdb.py` 脚本之后，您将在 `Python30` 目录中看到该文件。

现在您已经准备好开始使用 Python 的数据库 API 了。

## 14.3 使用 Python 的数据库 API

首先，介绍一下关于 Python 和关系数据库的一些历史。Python 对关系数据库的支持始于一些特定的解决方案，每种解决方案与每个特定的数据库(例如 Oracle)建立连接。每个数据库模块创建自身的 API，这些 API 对该数据库而言是高度特有的，因为每个数据库提供商都根据自己的需要发展自身的 API。把为一个数据库编写的代码移植到另一个数据库上时，需要完全重新编写以及重新测试全部代码，这给程序员带来了极大的痛苦，也使对每个数据库的支持变得更加困难。

但是，多年来 Python 已经成熟到可以提供一个通用数据库(DB)的 API，称为 DB API。一些特有的模块使 Python 脚本能够与不同的数据库进行通信，例如 DB/2、PostgreSQL 等。然而，所有这些模块都支持通用的 API，当编写一些访问数据库的脚本时，它便使得工作变得更加容易。本节将介绍这种通用的 DB API。

DB API 提供了使用数据库的最低标准，并尽可能使用 Python 的结构和语法。这种 API 包括如下几方面：

- 连接，包含如何连接到数据库的指导原则。
- 执行语句和存储过程，使用游标查询、更新、插入和删除数据。
- 事务，支持提交和回滚事务。
- 检查数据库模块以及数据库和表结构的元数据。
- 定义一些错误的类型。

以下各节将带领您逐步学习 Python 的数据库 API。

### 14.3.1 下载各个模块

针对每个需要访问的数据库，必须下载一个相应的单独的 DB API 模块。例如，如果需要访问 Oracle 数据库和 MySQL 数据库，那么必须先下载 Oracle 和 MySQL 的数据库模块。

注意：

请访问 <http://wiki.python.org/moin/DatabaseInterfaces> 以获得数据库的列表。

除了 Microsoft 的 SQL Server 之外，大多数主流数据库的相应模块都是存在的。然而，也可以使用 ODBC 模块访问 SQL Server。实际上，通过在 Windows 上使用 ODBC 或在 UNIX(包括 Mac OS X)或 Linux 上使用 ODBC 桥，mxODBC 模块可以与大多数数据库进行通信。如果需要这样做，可以通过在线搜索关于这些术语的更多信息来看一看其他人的做法。

请下载需要的模块。按照各个模块自带的操作指南安装它们。

注意：

可能需要有一个 C 编译器和构建环境来安装某些数据库模块。如果确实需要，那么需要阅读模块自身的文档，其中将会描述到这一点。

对于某些数据库，例如 Oracle，需要从很多彼此间稍微有些差别的模块中做出选择。应该选择那种看起来最能满足需求的模块，或者如果不能确定，那么请到本书的官方网站上向作者询问建议。

一旦确认安装了所有必要的模块，便可以开始使用连接了。

14.3.2 创建连接

connection 对象提供了一种脚本与数据库程序进行通信的方法。注意，这里假设数据库在一个单独的进程(或若干进程)中运行。各个 Python 数据库模块连接到数据库。但不包括数据库应用程序本身。

每个数据库模块需要提供一个连接函数，它返回一个 connection 对象。传递给连接函数的各个参数随着模块和与数据库通信的需要而变化。表 14-6 列出了最常见的各种参数。

表 14-6

参 数	用 法
Dsn	数据源名称，来自于 ODBC 术语。它通常包括数据库的名称及其运行所在的服务器名称
Host	运行数据库的主机或网络系统名称
Database	数据库的名称
User	连接到数据库的用户名
Password	给定用户名的密码

例如，可以使用如下代码作为一个指导：

```
conn = dbmodule.connect(dsn='localhost:MYDB',user='tiger',password='scott')
```

请根据数据库模块的文档决定需要哪些参数。

有了 Connection 对象之后，就可以使用事务了，本章稍后会介绍这方面的内容；关闭连接以释放系统资源，特别是数据库资源；以及获取一个游标。

14.3.3 使用游标

游标是一种 Python 对象，它使您能够使用数据库。在数据库术语中，游标位于数据库中一个表或若干表中的特定位置，有点类似于编辑文档时屏幕上的光标，不同的是光标被定位于一个像素位置。

为了获取一个游标，需要调用 connection 对象的 cursor 方法：

```
cursor = conn.cursor()
```

一旦获取了一个游标，就可以执行数据库上的各种操作了，例如插入一些记录。

### 试一试

### 插入记录

输入如下脚本，并将文件命名为 `insertdata.py`：

```
import os
import sqlite3

conn=sqlite3.connect('sample_database')
cursor = conn.cursor()

# Create employees.
cursor.execute("""
insert into employee (empid,firstname,lastname,manager,dept,phone)
values (1,'Eric','Foster-Johnson',1,1,'555-5555')""")

cursor.execute("""
insert into employee (empid,firstname,lastname,manager,dept,phone)
values (2,'Peter','Tosh',2,3,'555-5554')""")

cursor.execute("""
insert into employee (empid,firstname,lastname,manager,dept,phone)
values (3,'Bunny','Wailer',2,2,'555-5553')""")

# Create departments.
cursor.execute("""
insert into department (departmentid,name,manager)
values (1,'development',1)""")

cursor.execute("""
insert into department (departmentid,name,manager)
values (2,'qa',2)""")

cursor.execute("""
insert into department (departmentid,name,manager)
values (3,'operations',2)""")

# Create users.
cursor.execute("""
insert into user (userid,username,employeeid)
values (1,'ericfj',1)""")

cursor.execute("""
insert into user (userid,username,employeeid)
values (2,'tosh',2)""")
```

```

cursor.execute("""
insert into user (userid,username,employeeid)
values (3,'bunny',3) """)

conn.commit()

cursor.close()

conn.close()

```

当运行此脚本时，除非它产生一个错误，否则将不会看到任何输出。

### 示例说明

此脚本的前几行建立了数据库连接，并创建了一个 `cursor` 对象：

```

import os
import sqlite3
conn=sqlite3.connect('sample_database')
cursor = conn.cursor()

```

注意连接到 `Sqlite` 数据库的方式。如果要连接到一个不同的数据库，那么根据需要，用数据库特有的模块代替此模块，并修改调用，使其使用该数据库模块的 `connect` 函数。

接下来的几行执行一些 `SQL` 语句，将几行数据插入到之前创建的 3 个表中：即雇员表、部门表和用户表。`cursor` 对象的 `execute` 方法执行了 `SQL` 语句：

```

cursor.execute("""
insert into employee (empid,firstname,lastname,manager,dept,phone)
values (2,'Peter','Tosh',2,3,'555-5554') """)

```

此示例根据需要使用一个三引号字符串实现了对一些行的跨越。您将会发现，如果可以在多行中格式化命令，一些 `SQL` 命令，特别是那些嵌入在 `Python` 脚本中的命令，变得更加容易理解。对于本章稍后的几个示例中包含的复杂查询，这一点更加重要。

必须提交该事务，以将所有的修改保存到数据库中：

```

conn.commit()

```

注意本方法是对 `connection` 而不是 `cursor` 调用的。

当脚本完成时，关闭游标和连接以释放资源。在像这样的很简短的脚本中，这一点可能看起来并不重要，但是它帮助数据库程序和 `Python` 脚本释放它们的资源。

```

cursor.close()

conn.close()

```

现在已经具备了一些少量的样本数据，可以通过 `DB API` 的其他部分(如数据查询)使用它们了。

## 试一试

## 编写简单的查询

下面的脚本实现了一个简单的查询，它在雇员表和部门表上执行一个联接操作。

```
import os
import sqlite3
conn=sqlite3.connect('sample_database')
cursor = conn.cursor()
cursor.execute("""
select employee.firstname, employee.lastname, department.name
from employee, department
where employee.dept = department.departmentid
order by employee.lastname desc
""")
for row in cursor.fetchall():
    print(row)
cursor.close()
conn.close()
```

将这段脚本保存为 `simplequery.py`。当运行此脚本时，可以看到类似下面的输出：

```
('Bunny', 'Wailer', 'qa')
('Peter', 'Tosh', 'operations')
('Eric', 'Foster-Johnson', 'development')
```

## 示例说明

此脚本采用与之前的脚本相同的方式对 `connection` 和 `cursor` 进行初始化。然而，此脚本向 `cursor` 的 `execute` 方法传递了一个简单的联接查询。此查询从雇员表中选择两列，从部门表中选择一列。

## 注意：

这确实是一个简单的查询，但是即使如此，也应该像此处显示的一样，对查询语句进行格式化，以便它们是可读的。

当处理一些用户界面时，经常需要将数据库中存储的各个 ID 扩展成人类可读的值。例如，在本例中，该查询语句扩展了部门的 ID 来查询部门名称。不能期望人们能记住这些数值 ID 的意义。

该查询语句还按照雇员的姓对结果集进行了降序排序(这意味着以字母表的起点开始，正符合人们的习惯。然而，也可以使它们以升序次序排列)。

在调用了 `execute` 方法之后，对于能够找到的那些数据，将它们存储到 `cursor` 对象中。可以使用 `fetchall` 方法提取这些数据。

## 注意：

也可以使用 `fetchone` 方法从结果集中一次获取一行。

注意，这些数据显示为 Python 的元组：

```
('Bunny', 'Wailer', 'qa')
('Peter', 'Tosh', 'operations')
('Eric', 'Foster-Johnson', 'development')
```

可以将此示例作为创建其他查询的一个模板，例如在如下的“试一试”中创建一个更加复杂的联接查询。

### 试一试

### 编写复杂的联接查询

输入此脚本，并将文件命名为 `finduser.py`：

```
import sqlite3
conn=sqlite3.connect('sample_database')
cursor = conn.cursor()
username = 'bunny'
query = """
select u.username,e.firstname,e.lastname,m.firstname,m.lastname, d.name
from user u, employee e, employee m, department d where username=?
and u.employeeid = e.empid
and e.manager = m.empid
and e.dept = d.departmentid
"""
cursor.execute(query, (username,))
for row in cursor.fetchall():
    (username,firstname,lastname,mgr_firstname,mgr_lastname,dept) = row
    name=firstname + " " + lastname
    manager=mgr_firstname + " " + mgr_lastname
    print(username,":",name,"managed by",manager,"in",dept)
cursor.close()
conn.close()
```

当运行此脚本时，将会看到类似下面的结果：

```
bunny : Bunny Wailer managed by Peter Tosh in qa
```

需要传递一个人的用户名以实现从数据库中的查询。它必须是在数据库中一个人的有效的用户名。在此示例中，`bunny` 是一个之前插入到数据库中的用户名。

#### 示例说明

此脚本执行了在所有三个示例表上的一个联接，并通过表名的别名创建了短查询。其目的是，通过查找用户名，找出数据库中的给定用户。此脚本也显示了如何将经理的 ID 扩展为经理的姓名，以及如何将部门的 ID 扩展为部门的名称。所有这些都使输出的可读性变得更高。

这个示例也显示了如何从每行中提取出一些数据并放入 Python 变量。例如：

```
(username,firstname,lastname,mgr_firstname,mgr_lastname,dept) = row
```

请注意，这并不是什么新内容。请到第 3 章查看更多关于 Python 元组的信息，一行就是一个元组。

然而，此脚本的一个很重要的新特征是问号的使用，使您可以使用动态的数据创建查询。当调用 `Cursor` 类的 `execute` 方法时，可以传递一个包含动态数据的元组，`execute` 方法将会对 SQL 语句中的那些问号进行填充(此示例使用了包含一个元素的元组)。使用元组中的每个元素去按顺序代替问号。因此，动态值的数量与 SQL 语句中问号的数量相同是非常重要的，如下例所示：

```
query = """
select u.username,e.firstname,e.lastname,m.firstname,m.lastname, d.name
from user u, employee e, employee m, department d where username=?
and u.employeeid = e.empid
and e.manager = m.empid
and e.dept = d.departmentid
"""

cursor.execute(query, (username,))
```

当希望开始对各个表中的行进行更新时，此示例中使用的查询非常有用，因为用户希望输入一些有意义的值。您需要使用一些 SQL 语句将用户输入转换为必要的 ID。

例如，如下的脚本能够对一个雇员的经理进行修改：

### 试一试

### 更新雇员的经理

输入如下脚本，并将文件命名为 `updatemgr.py`：

```
import sqlite3
import sys
conn=sqlite3.connect('sample_database')
cursor = conn.cursor()
newmgr = sys.argv[2]
employee = sys.argv[1]
# Query to find the employee ID.
query = """
select e.empid
from user u, employee e
where username=? and u.employeeid = e.empid
"""
cursor.execute(query, (newmgr,));
for row in cursor.fetchone():
    if (row != None):
        mgrid = row
# Note how we use the same query, but with a different name.
cursor.execute(query, (employee,));
```

### 第III部分 开始使用 Python

```
for row in cursor.fetchone():
    if (row != None):
        empid = row
# Now, modify the employee.
cursor.execute("update employee set manager=? where empid=?", (mgrid, empid))
conn.commit()
cursor.close()
conn.close()
```

当运行此脚本时，需要传递用户的姓名及其经理的姓名以进行更新。这两个姓名都是用户表中的用户名称。例如：

```
$ python finduser.py bunny
bunny : Bunny Wailer managed by Peter Tosh in qa
$ python updatemgr.py bunny ericfj
$ python finduser.py bunny
bunny : Bunny Wailer managed by Eric Foster-Johnson in qa
```

#### 示例说明

此示例的输出分别显示了雇员行在更新之前和更新之后的情况，证实 `updatemgr.py` 脚本起到了作用。

`updatemgr.py` 脚本需要用户提供两个值：将更新的雇员的用户名和新经理的用户名。这两个名称必须是已经存储在数据库中的用户名，它们通过使用一个简单的查询被转换为两个 ID。这种做法的效率不是很高，因为它包括对数据库的两次额外双程访问。一种效率更高的做法是在 `update` 语句中执行一个内部 `select` 语句。然而，为简单起见，这两个分开的查询更容易令人理解。

本示例还显示了 `Cursor` 类的 `fetchone` 方法的使用。然后，最后的 SQL 语句对给定用户的雇员行进行了更新，使其有一个新的经理。

下一个示例使用一种类似的技术解雇一个雇员。您也许会感到十分好玩(解雇您的朋友、您的敌人等)。

#### 试一试

#### 删除雇员

输入如下脚本，并将文件命名为 `terminate.py`：

```
import sqlite3
import sys
conn=sqlite3.connect('sample_database')
cursor = conn.cursor()
employee=sys.argv[1]
# Query to find the employee ID.
query = """
select e.empid
from user u, employee e
```

```

where username=? and u.employeeid = e.empid
"""
cursor.execute(query, (employee,));
for row in cursor.fetchone():
    if (row != None):
        empid = row
# Now, modify the employee.
cursor.execute("delete from employee where empid=?", (empid,))
conn.commit()
cursor.close()
conn.close()

```

当运行此脚本时，需要传递要解雇的那个人的用户名。除非此脚本产生一个错误，否则不会看到任何输出：

```

$ python finduser.py bunny
bunny : Bunny Wailer managed by Eric Foster-Johnson in qa
$ python terminate.py bunny
$ python finduser.py bunny

```

### 示例说明

此脚本使用与 `updatemgr.py` 脚本相同的技术，执行一个最初的查询得到给定用户名的雇员 ID，然后在后面的 SQL 语句中使用此 ID。在最后的 SQL 语句中，此脚本从雇员表中删除该雇员。

### 注意：

此脚本没有删除用户表中的相应记录。本章的习题 3 会解决这一问题。

## 14.3.4 使用事务并提交结果

对于每个连接对象，当它正进行某个活动时，会同时管理一个事务。使用 SQL 时，只有当提交了事务时，数据才被更改。数据库保证它将要么执行所有的修改，要么不做任何修改。因此，数据库将不会处于一种不确定的和可能不正确的状态。

调用连接对象的 `commit` 方法以提交一个事务：

```
conn.commit()
```

注意，有关事务的各个方法是 `connection` 类的一部分，而不是 `cursor` 类的一部分。

如果有错误发生，比如抛出一个可以处理的异常，那么应该调用 `rollback` 方法以撤消此不完全事务的效果，此操作将保证使数据库恢复到启动此事务之前的状态：

```
conn.rollback()
```

可以对一个事务进行回滚的能力是非常重要的，因为可以在保证数据库不被改变的情况下处理一些错误。另外，回滚对于测试也是非常有用的。可以将对大量行的插入、修改

和删除作为一个单元测试的一部分，然后回滚事务以撤消所有更改的效果。这使得单元测试的运行不会对数据库产生任何永久更改。它还允许重复运行单元测试，因为每次运行都重置了数据。

**提示：**  
关于测试的更多信息，请参看第 12 章。

14.3.5 检查模块的功能和元数据

DB API 定义了几个需要在模块级别定义的全局变量。可以使用这些全局变量确定关于数据库模块的信息以及它所支持的一些特征。表 14-7 中列出了这些全局变量。

表 14-7

全 局 变 量	保 存 值
Apilevel	对于 DB API 2.0 应该保存 “2.0”，对于 API 1.0 保存 “1.0”
Paramstyle	定义了 SQL 语句中，如何显示动态数据的占位符。包括的值如下： “qmark” ——使用问号，如本章中的示例所示 “numeric” ——使用一种位置数字的方式，带有 “:1”、“:2” 等 “named” ——使用冒号和每个参数的名称，如:name “format” ——使用 ANSI C Sprintf 格式代码，例如，对于字符串使用%s，对于整数使用%d “pyformat” ——使用 Python 的扩展格式代码，如%(name)s

**注意：**  
另外，切记 pydoc 对您是非常有益的。可以使用 pydoc 显示关于模块(例如数据库模块)的信息。

可以利用 cursor 对象的 definition 属性查看返回数据的信息。此信息应该是一个序列的集合，每个序列包含 7 个元素，对应于结果数据中的一列。这些序列包含如下的各项：

```
(name, type_code, display_size, internal_size, precision, scale, null_ok)
```

除了前两项之外，其余各项都可以使用 None。

14.3.6 处理错误

错误总会发生。对于数据库，经常会发生很多错误。DB API 定义了大量存在于每个数据库模块中的错误。表 14-8 列出了这些异常。

表 14-8

异 常	用 法
Warning	用于非致命的问题。必须定义为 StandardError 的子类
Error	所有错误的基类。必须定义为 StandardError 的子类
InterfaceError	用于数据库模块中的错误，而不是数据库本身的错误。必须定义为 Error 的子类
DatabaseError	用于数据库中的错误。必须定义为 Error 的子类
DataError	DatabaseError 的子类，指数据中的错误
OperationalError	DatabaseError 的子类，指那些类似于丢失数据库连接的错误。这些错误一般在 Python 脚本设计者的控制之外
IntegrityError	DatabaseError 的子类，用于可能破坏关系完整性的情况，例如唯一性约束或外键约束
InternalError	DatabaseError 的子类，指数据库模块内部的错误，例如游标未被激活
ProgrammingError	DatabaseError 的子类，指那些类似于错误的表名称等由程序员造成的问题
NotSupportedError	DatabaseError 的子类，指试图调用不支持的功能

Python 脚本应该处理这些错误。通过阅读 DB API 的规范可以了解更多信息。详细信息请访问 [www.python.org/topics/database/](http://www.python.org/topics/database/)和 [www.python.org/peps/pep-0249.html](http://www.python.org/peps/pep-0249.html)。

### 14.4 本章小结

数据库为数据的存储提供了一种方便的手段。可以使用一些附加的模块编写能够访问所有主流数据库的 Python 脚本。本章提供了 SQL 的一个简短介绍，并讨论了 Python 的数据库 API。

您还了解了各种 dbm 模块，它们允许使用各种 dbm 库持久化一个字典。这些模块使您能够使用各个字典并透明地对数据进行持久化。

另外，本章介绍了 Python 数据库 API，它们定义了一些方法和函数的标准集合，所有的数据库模块都应该提供它们。要点如下：

- connection 对象封装了到数据库的一个连接。使用数据库模块的 connect 函数以得到一个新的连接。传递给 connect 函数的各个参数可能会因各个模块的不同而不同。
- 游标提供了用于与数据库进行交互的主要对象。使用 connection 对象以获得一个游标。游标使您可以执行一些 SQL 语句。
- 可以将动态数据作为一个包含若干值的元组传递给游标的 execute 方法。这些值被放置于各个 SQL 语句中，从而允许创建一些可重用的 SQL 语句。
- 在执行了一个查询操作之后，cursor 对象对数据进行保存。使用 fetchone 或 fetchall 方法都可以提取数据。
- 在对数据库进行修改之后，调用 connection 对象的 commit 方法以提交事务并保存更改。使用 rollback 方法可以撤消更改。

- 当操作完成后，调用每个游标的 `close` 方法。当不需要连接时，调用 `connection` 对象的 `close` 方法。
- DB API 定义了一个异常集合。Python 脚本应该对这些异常进行检查，以处理可能出现的各种问题。

第 15 章将讨论 XML、HTML 和 XSL 样式表，这些是进行 Web 开发时经常用到的技术。

## 14.5 习题

1. 假设需要编写一个 Python 脚本来存储所在部门的员工喜欢吃的比萨饼。需要存储每个人的姓名以及他最喜欢的比萨饼配料。要实现这个脚本，下面的哪种技术最合适？
  - a. 建立一个关系数据库，如 MySQL 或 Sqlite。
  - b. 使用 dbm 模块，如 dbm。
  - c. 实现一个基于 Web 服务的丰富的 Web 应用程序，以创建一个涉及大量术语应用程序。
2. 使用表名称的别名重新编写下列中的查询：

```
select employee.firstname, employee.lastname, department.name
from employee, department
where employee.dept = department.departmentid
order by employee.lastname desc
```

3. 如前面所示，`terminate.py` 脚本从雇员表中删除一个雇员行，但是此脚本并不完善。对于同一个人，在用户表中会遗留一行。修改 `terminate.py` 脚本以删除雇员及该雇员在用户表中的行。



# 第 15 章

## 使用 Python 处理 XML

在过去的几年里，XML 作为存储和传输结构化数据的媒介，迅速地流行起来。Python 通过标准库和众多的第三方库，对 XML 极其丰富的各种标准给予了很好的支持。

本章将介绍：

- 创建和操作 XML。
- 验证 XML。
- 学习使用 Python 自带的标准库。

### 15.1 XML 的含义

XML 在会议中经常可以听到。它以自己的灵活性和可扩展性激发人们的雄心，他们把 XML 用于实现各种功能，例如创建新的格式独立的语义代码存储机制，乃至代替对象序列化。但是除了这些时髦词语和夸张的宣传，XML 究竟是什么？它能解决所有问题吗？也许不能。但它确实是功能强大、灵活、基于开放标准的数据存储方法。XML 语言是无限可定制的，适合希望存储的任何类型的数据。它的格式易于阅读，而且也易于程序解析。它鼓励语义标记，而不是基于格式的标记，通过分离不同标记的内容和表示，使一条数据可被多次赋予新的用途，并以多种不同的方式显示。

#### 15.1.1 层次标记语言

XML 本质上是一个简单的层次标记语言。标签用来标记内容中具有不同语义的部分，属性用来向内容添加元数据。

下面是一个简单的 XML 文档示例，它描述了一个图书馆：

```
<?xml version="1.0"?>
<library>
  <book>
    <title>Sandman Volume 1: Preludes and Nocturnes</title>
    <author>Neil Gaiman</author>
  </book>
  <book>
    <title>Good Omens</title>
```

```

    <author>Neil Gaiman</author>
    <author>Terry Pratchett</author>
</book>
<book>
    <title>"Repent, Harlequin!" Said the Tick-Tock Man</title>
    <author>Harlan Ellison</author>
</book>
</library>

```

注意，每条数据都被封装在标签中，这些标签嵌套于层次中，该层次包含了它封装的数据的详细信息。基于前面的文档，可以总结出<author>是<book>的一条子信息，<title>也是，每个图书馆都有一个属性 owner。

与语义标记语言(如Latex)不同，XML 中每条数据必须括在标签中。顶层标签是文档的根，它封装了文档的全部内容。每个 XML 文档只能有一个根。

文档的根节点前面是 XML 声明：<?xml version="1.0"?>。这个强制的元素告诉处理器这是一个 XML 文档。在写本书的时候，一共有两个 XML 版本：1.0 和 1.1。因为 Python 还未完全支持 XML 版本 1.1，所以本书的示例主要侧重于版本 1.0。

语义标记的一个问题是在数据的上下文发生变化时可能引起困惑。例如，您也许希望将一系列图书的标题转移到一个关于作者的数据库中。然而，如果没有人查看的话，数据库没有办法知道<title>指书的题目、编辑的业务头衔或者作者的荣誉头衔。这是名称空间发挥作用的地方。名称空间提供了对标签的引用框架，并且被赋值了一个 URL 形式的唯一 ID，外加用于应用该名称空间中的标签的一个前缀。例如，您可能创建了一个 Library 名称空间，它的标识符是 <http://server.domain.tld/NameSpaces/Library>，前缀是 lib:，您可以使用该名称空间创建标签的引用框架。在定义了名称空间后，XML 文档如下所示：

```

<?xml version="1.0"?>
<lib:library
    xmlns:lib="http://server.domain.tld/NameSpaces/Library">
    <lib:book>
        <lib:title>Sandman Volume 1: Preludes and Nocturnes</lib:title>
        <lib:author>Neil Gaiman</lib:author>
    </lib:book>
    <lib:book>
        <lib:title>Good Omens</lib:title>
        <lib:author>Neil Gaiman</lib:author>
        <lib:author>Terry Pratchett</lib:author>
    </lib:book>
    <lib:book>
        <lib:title>"Repent, Harlequin!" Said the Tick-Tock Man</lib:title>
        <lib:author>Harlan Ellison</lib:author>
    </lib:book>
</lib:library>

```

现在很明显，标题元素来自于图书馆名称空间中定义的一组元素，可以按照该名称空间的规范相应地处理它们。

名称空间的声明可以添加到文档的任意节点，并且此名称空间对该节点的所有子节点都有效。在大多数文档中，所有的名称空间声明都应用于文档的根节点，即使名称空间只在文档中更深层的部分使用。名称空间的声明必须用于根节点，才能对文档的每个标签都有效。

一个文档可以拥有和使用多个名称空间。例如，前面的示例中，图书馆可以对图书馆信息使用一个名称空间，对出版社信息使用另一个名称空间。

注意名称空间声明中的 `xmlns:` 前缀。有些名称空间前缀是为 XML 和与其相关的语言预留的，例如 `xml:`、`xsl:` 和 `xmlns:`。名称空间的声明可以添加到文档中的任意节点，这样该名称空间对此节点的所有子节点都有效。

上面的示例是一个很简单的文档。更加复杂的文档可能会包含用于存储未处理的数据的 CDATA 部分、注释，以及用于存储针对特定 XML 处理器的信息的处理指令。关于该主题的详尽介绍，可以访问 <http://w3cschools.org>，或者参考 Wrox Press 的 *Beginning XML, 3rd Edition*(David Hunter) 一书。

### 15.1.2 一组标准

XML 不只是一个存储层次数据的工具。如果这是它的全部功能，那么 XML 很快就会被已有的许多轻量级的数据存储方法所淘汰。XML 的强大之处在于它的可扩展性，还有它附带的一些标准，例如 XSLT、XPath、模式和 DTD 语言，以及用于查询、链接、描述、显示和操作数据的一系列其他标准。模式和 DTD 可用于描述 XML 词汇和验证文档的有效性。XSLT 是一强大的转换引擎，可以将 XML 词汇转换到其他语言，例如 HTML、纯文本、PDF 或其他格式。XPath 是描述 XML 节点集合的查询语言。XSL-FO 提供了创建 XML 文档的一种途径，它对要转换为 PDF 或其他可视格式的文档的格式和布局进行了描述。

XML 的另外一个优点是，大多数 XML 工具也是用 XML 编写的，并且可被相同的工具处理。例如，XSLT 和模式都是用 XML 编写的。这一特点的实际意义在于，用 XSLT 编写其他 XSLT 或者模式会非常简单，用模式验证 XSLT 和其他模式也比较容易。

## 15.2 模式/DTD

模式和 DTD(Document Type Definition, 文档类型定义)都是实现文档模型的方法。文档模型用以描述词汇和文档结构。它与创建数据库时的 DBA 的工作有些类似。利用它可以定义文档中将要出现的数据元素，元素之间的关系，以及元素的数量等。用日常用语解释前面的 XML 示例的文档模型：“一个图书馆是某个人拥有的一些书的集合。每本书都有标题和至少一位作者。”

DTD 和模式表达文档模型的方式不同，但是描述了文档的基本原则。后面将会看到，这两者有一些细微的差别，但是它们大部分的功能都相同。

### 15.2.1 文档模型的用途

文档模型用于在处理文档之前，验证它的内容是否符合标准。文档模型在与一个可能

会突然改变数据模型的应用程序交换数据时非常有用，而当希望限制用户输入的内容时，例如在一个基于 XML 的文档系统中，您将处理手动创建的 XML 文档，而不是处理一个应用程序中产生的 XML 文档，文档模型也将提供极大的帮助。

15.2.2 是否需要文档模型

一些应用程序并不需要文档模型。如果能够控制数据交换的两端，并且可以预测即将接收到的数据内容，那么此应用程序并不需要文档模型。

15.3 文档类型定义

DTD 是文档类型定义。它们是表示文档模型的最原始的方法，在 Internet 上相当普及。DTD 最初是为了描述 SGML 而创建的，由于 SGML 的语法从那时起就没怎么变化，因此 DTD 有足够的时间不断扩展。W3C(万维网联盟，给 Internet 制定标准的组织之一)，仍旧使用 DTD 表示文档类型，因此每一种 HTML 标准、可缩放矢量图形(Scalable Vector Graphics, SVG)、MathML 和其他许多有用的 XML 都有相应的 DTD。

15.3.1 DTD 示例

如果将对示例中的图书馆 XML 文档的描述转换成 DTD 格式，结果如下所示：

```
<?xml version="1.0"?>
<!ELEMENT library (book+)>
<!ATTLIST library
    owner CDATA #REQUIRED
>
<!ELEMENT book (title, author+)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
```

为了向前面讨论的图书馆文件中添加对该 DTD 的引用，需要在文件顶部的 XML 声明后插入一行：<!DOCTYPE config SYSTEM "library.dtd">，其中 library.dtd 是系统上的 DTD 路径。

接下来分别解释每个步骤。第一行，< ?xml version="1.0" ? >，告诉您这是一个 XML 文档。从技术上讲，这一行可有可无，因为 DTD 与其他 XML 文档的行为方式不同，本书稍后再对此进行详细解释。下一行，<!ELEMENT library (book+)>，告诉您有一个元素是 library，它可能有一个或者多个类型为 book 的子元素。DTD 中的元素频率和分组的语法非常简洁，与正则表达式类似。表 15-1 列出了 DTD 中元素频率和元素分组操作符。

表 15-1

运 算 符	定 义
?	指定 0 个或者 1 个前面出现的元素。例如，editor?意味着一本书可能有可选的 editor 元素

(续表)

运 算 符	定 义
+	指定一个或者多个前面出现的元素。类似于上一个示例，author+意味着一本书有一个或者多个作者
,	指定一系列元素必须以此特定的顺序出现。(title,author+)意味着书必须有一个标题，随后是一个或者多个作者，必须以该顺序出现
(list)	将元素组织在一起。应用于圆括号后的运算符适用于组中的所有元素。例如，(author,editor)+的含义是一篇文档可能有多个作者与多个编辑
!	“或”运算符。该运算符允许在多个选项之间选择。例如，(author editor)允许一本书有一个作者或者一个编辑，但是不能同时具有两者
*	指定前面的元素或组出现 0 次或多次。(book, CD)*允许图书馆中有任意数目的书或者 CD，或者什么都没有，是空的

下面所示的部分稍微有点复杂：

```
<!ATTLIST library
    owner CDATA #REQUIRED
>
```

第一行指定图书馆元素有一个属性列表。注意，属性列表是与图书馆中元素的声明分开的，它与元素的名称链接起来。如果元素名称发生了改变，属性列表必须更新，以指向新的元素名称。下面是元素的属性列表。在这个例子中，library 只有一个属性，但是属性列表可以包含任意多个属性。属性的声明有三个强制的元素：属性名称、属性类型和属性描述。一个属性类型要么是 DTD 规范指定的数据类型，要么是一组允许的值。属性的描述指定了属性的行为。这里可以给出属性的一个默认值，不论该属性是可选的还是必需的。

15.3.2 DTD 不完全是 XML

DTD 作为 SGML 的延续，在技术上与 XML 不完全相同。DTD 与模式不同，用 XML 的那些工具很难操作和验证它们。如果在 DTD 的开头声明一个文档类型，解析器可能会忽略它，更可能会产生一个语法错误。尽管创建 DTD 是有规范的，但是没有 DTD 格式的文档模型可用来验证一个 DTD 文档结构的有效性。有一些可以验证 DTD 的工具，但是它们与验证 XML 的工具不同。另一方面，存在模式形式的文档模型，可以用标准的 XML 工具针对它们验证模式。

15.3.3 DTD 的局限性

DTD 有很多局限性。尽管可以利用 DTD 表达复杂的结构，但是它们维护起来很困难。DTD 很难在文档模型中清楚地表达数值范围。如果希望指定一个图书馆包含不超过 100 本书，可以使用<!ELEMENT library (book, book, book, book etc etc)>，但是这很快会变为不可读的一堆代码。DTD 很难表达任意顺序的一些元素。如果可以按任意顺序接收 3 个元素，

必须写<!ELEMENT book ( ( (author, ((title, publisher) | (publisher, title))) | (title, ((author, publisher) | (publisher, author))) | (publisher, ((author, title) | (title, publisher))))>, 这看起来更像 LISP(一门包含许多括号的语言)而不是 XML, 而且过于复杂了。最后, DTD 不允许为数据指定模式, 因此不能表达这样的构造: “一个电话号码应当由数值、短线和加号组成。”好在, W3C 发布了一个稍微复杂的用于描述文档的语言规范, 它就是模式。

## 15.4 模式

模式用来弥补 DTD 的不足, 它可以提供基于 XML 的复杂语言, 用以描述文档模型。利用它可以清晰地指定内容的数值模型, 使用正则表达式描述字符数据模式, 以及表达例如序列、选择和不受限制的模型的内容模型。

### 15.4.1 示例模式

如果希望将假设的图书馆模型转换成模式, 并使其包含与 DTD 相同的信息, 可以用下面的代码实现该功能:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="library">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="book" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="title" type="xs:string"/>
              <xs:element name="author" type="xs:string" maxOccurs="unbounded"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="owner" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

上述模式表达的数据模型与 DTD 几乎相同, 但是不同之处也比较明显。

### 15.4.2 模式是纯粹的 XML

这篇文档的顶层节点包含一个名称空间声明, 指定了以 xs:开头的所有标签属于由 URI “http://www.w3.org/2001/XMLSchema” 标识的名称空间。出于实际考虑, 这表明您现在有一个可用于验证模式的文档模型, 使用的工具与验证其他 XML 文档的工具相同。

15.4.3 模式具有层次

接下来，注意前面的文档的层次与模式描述的文档层次非常类似。文档模型没有创建单独的元素，并使用引用将它们链接起来，而是尽可能近似地模仿文档结构。可以创建全局元素，并且在一个结构中引用它们，但是并非必须使用引用，也就是说，引用是可选的。这将创建一个更加直观的结构，可以看到由该模型创建的可能的文档形式。

15.4.4 模式的其他优点

模式支持一些属性，例如 `maxOccurs`，它接收一个无限制的数值作为参数，数值范围从 1 开始到无穷大，这表明可以出现任意多个元素或者分组。尽管这个模式没有表现出来，但是实际上模式可以利用 `pattern` 属性表示一个元素是否满足特定的正则表达式，而且通过将选择和序列内容模型混合在一起，模式还可以表示更加灵活的内容模型。

15.5 XPath

XPath 是在 XML 文档中描述位置和节点集合的语言。有一些关于 XPath 的完整教材。然而，它的基础知识相当简单。XPath 表达式包含对某个节点必须匹配的模式描述。如果该节点匹配此模式，它将被选择；否则，它将被忽略。模式由一系列的步骤组成，要么相对于一个上下文节点，要么由文档的根节点绝对定义。绝对路径以斜杠开始，而相对路径则不是，路径的每一步之间由斜杠分隔开。

路径中的每一步包含 3 个部分：描述移动方向的轴，沿着该轴选择节点的测试，还有可选的谓词，它是节点必须满足的 Boolean 型测试(取真值或者假值)。一个示例步骤是 `ancestor-or-self::book[1]`，其中 `ancestor-or-self` 是轴，`book` 是节点测试，`[1]` 是谓词，指定选择满足所有其他条件的第一个节点。如果省略了轴，则假定引用当前节点的子轴，因此 `library/book[1]/author[1]` 选择图书馆中第一本书的第一位作者。

节点测试既可以是一个函数，也可以是一个节点名称。例如，`book/node()` 将返回选择的书节点下的所有子节点，不管它们是文本还是元素。

表 15-2 描述了轴的一些快捷方法。

表 15-2	
快捷方法	含 义
@	指定属性轴。这是 <code>attribute::</code> 的缩写
*	指定当前节点的所有子节点
//	指定当前节点的所有后代节点。这是 <code>descendant-or-self::*//</code> 的缩写。如果在 XPath 开头使用，它将匹配文档中任意地方的元素

想获得该主题更加全面的知识，可以访问网址 <http://w3cschools.org> 或者挑选一本关于 XPath 的书籍。

## 15.6 HTML 是 XML 的子集

XML 与 HTML 有许多类似之处。这并不完全是偶然。XML 和 HTML 都出自于 SGML，因此共享许多语法特性。HTML 的早期版本并不直接兼容 XML，因为 XML 要求每个标签都是闭合的，但是有些 HTML 标签并不要求有结束标签，例如<br>和<img>。然而，为了努力将两个标准融合在一起，W3C 声明了 XHTML 模式。XHTML 像纯粹的 XML 一样可被相同的工具操作。然而，Python 也包含了专门为处理 HTML 而设计的特殊库。

### 15.6.1 HTML DTD

HTML 的当前版本是 4.01，它包括 4.01 Transitional、4.01 Strict 和 4.01 Frameset，其中 4.01 Frameset 是专门用于处理框架的。

### 15.6.2 HTMLParser

与 `htmlib` 类不同，`HTMLParser` 类并不基于 SGML 解析器，它可用于 XHTML 和 HTML 的早期版本。

#### 试一试

#### 使用 HTMLParser

(1) 创建包含至少一个 `h1` 标签的示例 HTML 文件 `headings.html`，并将该文件保存于 `Python30` 目录。

(2) 将如下代码剪切并粘贴到一个文件中：

```
from html.parser import HTMLParser
class HeadingParser(HTMLParser):
    inHeading = False
    def handle_starttag(self, tag, attrs):
        if tag == "h1":
            self.inHeading = True
            print("Found a Heading 1")
    def handle_data(self, data):
        if self.inHeading:
            print(data)
    def handle_endtag(self, tag):
        if tag == "h1":
            self.inHeading = False
hParser = HeadingParser()
file = open("headings.html", "r")
html = file.read()
file.close()
hParser.feed(html)
```

(3) 运行上述代码。

### 示例说明

HTMLParser 定义了一些方法，当解析器找到特定类型的内容，例如开始标签、结束标签或者处理指令时，就会调用这些方法。默认情况下，这些方法不执行任何操作。为了解析一个 HTML 文档，必须创建一个从 HTMLParser 继承而来的类，并实现必要的方法。在创建和实例化 parse 类之后，使用 feed 方法向解析器提供数据。可以每次提供一行数据，或者一次提供所有数据。

该示例类仅处理<h1>类型的标签。当 HTMLParser 遇到一个标签时，将调用 handle\_starttag 方法，并向其传递标签名称和任何附加的属性。

该 handle\_starttag 方法判断某个标签是否是<h1>类型的标签。如果是，它打印一条消息，说明它遇到了一个 h1 标签，并且设置一个标志以表明它现在在<h1>中。

如果找到了文本数据，handle\_data 函数将被调用，它根据标志判断当前是否位于<h1>标签中。如果标志为 true，该方法将打印出文本数据。

如果遇到了结束标签，handle\_endtag 方法将被调用，它判断刚结束的标签是否是一个<h1>标签。如果是，它将打印一条消息，并将标志设为 false。

## 15.7 Python 中可用的 XML 库

Python 有许多可以帮助处理 XML 的库。可以选择若干个 DOM(Document Object Model, 文档对象模型)实现，一个非验证的 Expat XML 解析器的接口，还可以选择使用 SAX(Simple API for XML, XML 简单 API)的若干库。

可用的 DOM 实现如下：

- xml.dom: 一个完全兼容的 DOM 处理器。
- xml.dom.minidom: 一个轻量级的并且更快的 DOM 规范的实现，但是与 Python 并不完全兼容。

## 15.8 SAX 的含义

在解析 XML 时，可以选择两种不同类型的解析器：SAX 和 DOM。SAX 代表 XML 的简单 API，最初只为 Java 实现，在 2.0 版本时被添加到 Python 中。它是基于流的事件驱动的解析器。这些事件称作“文档事件”，文档事件在元素开始之处、元素结尾处、遇到文本节点或者遇到一个注释时都有可能发生。例如，下面所示的简单文档：

```
<?xml version="1.0"?>
<author>
  <name>Ursula K. LeGuin</name>
</author>
```

可能会引发下列事件：

```
start document
start element: author
start element: name
characters: Ursula K. LeGuin
end element: name
end element: author
end document
```

无论何时发生文档事件时，解析器为调用应用程序触发一个事件让它处理。更精确地说，解析器为调用应用程序的 **Content Handler** 对象触发事件让其处理。**Content Handler** 对象实现了 **SAX API** 指定的已知接口，解析器可以通过该接口调用方法。在前面的示例中，解析器可以调用内容处理程序的 **startDocument** 方法，接在再调用两次 **startElement** 方法，等等。

### 15.8.1 基于流

当用 **SAX** 解析文档时，文档以其出现的顺序被读入和解析。解析器以数据流的方式(这表明不必一次读入所有数据)打开该文件或者其他数据源(例如 **URL**)，之后无论何时遇到任何元素都将引发事件。

由于解析器不必等到整个文档加载完毕之后才开始解析，**SAX** 可以在开始读文档后马上就解析它。然而，因为 **SAX** 没有读入整个文档，所以它在发现文档的格式不正确之前，也许已经处理了一部分文档。基于 **SAX** 的应用程序应当实现这些条件下的错误检查。

### 15.8.2 事件驱动

与 **GUI** 类似，当使用 **SAX** 工作时，文档事件由事件处理程序处理。程序员为特定类型的文档事件声明回调函数，这些函数被传递给解析器，当与回调函数匹配的文档事件发生时，它们就被调用。

### 15.8.3 DOM 的含义

**DOM** 的核心在于文档对象。它是 **XML** 文档的基于树的表示形式。基于树的模型很适合 **XML** 的层次结构，使得 **DOM** 成为处理 **XML** 的直观方法。树中的元素称作节点对象，节点拥有属性、子节点和文本等，它们以对象的形式存储在树中。**DOM** 对象有许多方法，可以用于创建和添加节点，寻找特定类型或者名称的节点，以及重新排序或者删除节点。

### 15.8.4 内存中访问

**SAX** 与 **DOM** 最大的不同在于，后者能够在内存中存储整个文档，并且以树的形式操作和搜索其中的元素，而不是强迫您重复地解析文档，或者强迫您在内存中自己构建文档的表示。文档被解析一次，随后即可在内存中添加、删除或者改变文档中的节点，并在程序结束时，将内存中的文档表示写回到文件中。

## 15.9 使用 SAX 或者 DOM 的理由

尽管 SAX 或者 DOM 可以实现与 XML 几乎相同的功能，在给定的任务中由于若干原因，您也许希望只使用其中的一个。例如，如果在一个应用程序中需要根据用户输入不断地修改一个 XML 文档，可能就希望利用 DOM 方便的随机访问功能。另一方面，如果正在构建一个应用程序，需要能够以最小的开销快速地处理 XML 流，SAX 也许是更好的选择。下面是架构使用 XML 的应用程序时两种方法的一些好处和坏处。

### 15.9.1 能力权衡

DOM 的架构思想是随机访问。它提供了一棵可以在运行时操作的树，并且只需将其加载到内存中一次。SAX 是基于流的，因此数据以流的形式传入，一个字符接着一个字符，但是只有在处理文档时才能看到完整的文档；因此，如果希望随机地访问数据，必须要么在内存中基于文档事件创建文档的一个部分树，要么在每次需要一条不同的数据时重新解析文档。

大部分人都认为 DOM 的面向对象的行为直观而易于理解。与之相比，SAX 的事件驱动模型更类似于函数式编程，掌握起来比较困难。

### 15.9.2 内存考虑

如果工作环境中内存有限，DOM 也许并不是一个好的选择。即使在一个相当高端的系统中为一个 2M 或者 3M 的 XML 文档构建一个 DOM 树，也会使计算机在处理时暂停运行。由于 SAX 将文档看做流，它从不将整个文档加载到内存中，因此在内存有限或者处理较大的文档时，最好使用 SAX。

### 15.9.3 速度考虑

使用 DOM 创建文档树时，需要大量预先处理的时间，但是一旦文档树被创建，DOM 搜索和操作节点的速度会变得非常快，因为整个文档都存储在内存中。SAX 搜索文档的速度还算快，但是对它们的操作却不够高效。然而，对于文档转换，应该选择 SAX 作为解析器，因为事件驱动模型速度很快，并且与 XSLT 的工作方式的兼容性很好。

## 15.10 Python 中可用的 SAX 和 DOM 解析器

在 Python 中可以使用如下的 SAX 和 DOM 解析器：`xml.sax` 和 `xml.dom.minidom`。它们的行为方式略有不同，因此这里分别概述它们。

### 15.10.1 `xml.sax`

`xml.sax` 是 Python 内置的 SAX 包。它默认使用 Expat 非验证解析器，但是可以向其传入一组解析器实例，以改变这种行为。

## 15.10.2 xml.dom.minidom

xml.dom.minidom 是轻量级的 DOM 实现，它比完整的 DOM 实现更简单，更小。

## 试一试

## 使用 DOM 处理 XML

- (1) 将本章开头的 XML 示例文件保存到 library.xml 中。
- (2) 要么手动输入，要么从本书的合作站点得到如下代码，并将其保存到文件 xml\_minidom.py 中：

```
from xml.dom.minidom import parse
import xml.dom.minidom
def printLibrary(library):
    books = myLibrary.getElementsByTagName("book")
    for book in books:
        print("*****Book*****")
        print("Title: %s" % book.getElementsByTagName("title")[0].childNodes[0].data)
        for author in book.getElementsByTagName("author"):
            print("Author: %s" % author.childNodes[0].data)
# open an XML file and parse it into a DOM
myDoc = parse('library.xml')
myLibrary = myDoc.getElementsByTagName("lib:library")[0]
#Get all the book elements in the library
books = myLibrary.getElementsByTagName("book")
#Print each book's title and author(s)
printLibrary(myLibrary)
#Insert a new book in the library
newBook = myDoc.createElement("book")
newBookTitle = myDoc.createElement("title")
titleText = myDoc.createTextNode("Beginning Python")
newBookTitle.appendChild(titleText)
newBook.appendChild(newBookTitle)
newBookAuthor = myDoc.createElement("author")
authorName = myDoc.createTextNode("Peter Norton, et al")
newBookAuthor.appendChild(authorName)
newBook.appendChild(newBookAuthor)
myLibrary.appendChild(newBook)
print("Added a new book!")
printLibrary(myLibrary)
#Remove a book from the library
#Find ellison book
for book in myLibrary.getElementsByTagName("book"):
    for author in book.getElementsByTagName("author"):
        if author.childNodes[0].data.find("Ellison") != -1:
            removedBook= myLibrary.removeChild(book)
            removedBook.unlink()
print("Removed a book.")
printLibrary(myLibrary)
```

```
#Write back to the library file
lib = open("library.xml", 'w')
lib.write(myDoc.toprettyxml(" "))
lib.close()
```

(3) 使用命令 `python xml_minidom.py` 运行该文件。

### 示例说明

为了创建 DOM，文档需要被解析成一棵文档树。可以通过调用 `xml.dom.minidom` 中的 `parse` 方法完成该操作。该方法返回一个 `Document` 对象，它包括查询子节点、得到文档中具有特定名称的所有节点、创建新节点等方法。方法 `getElementsByTagName` 返回一个名称与给定参数匹配的节点对象的列表，可用它提取文档的根节点 `<library>`。打印方法再次使用了 `getElementsByTagName`，它打印出每个书节点的题目和作者。包含文本的节点被认为有一个子节点，文本存储于节点的 `data` 属性，因此，`book.getElementsByTagName("title")[0].childNodes[0].data` 仅检索出元素 `<title>` 下的文本节点，并以字符串形式返回其数据。

在 DOM 中构造一个新节点需要创建一个 `Document` 对象的新节点，向它添加所有必需的属性和子节点，之后将其插入到文档树中正确的节点下。`Document` 对象的 `createElement(tagName)` 方法可以创建一个新节点，新节点的名称是传入的任意参数。添加文本节点的完成方式几乎与此完全相同，只是需要调用 `createTextNode(string)`。当所有子节点都创建完毕之后，调用节点的 `appendChild` 方法可以创建该节点的结构，新创建的子节点将被添加到该节点。节点的方法 `insertBefore(newChild, refChild)` 可以在节点的子节点列表中的任意位置插入新的子节点，方法 `replaceChild(newChild, oldChild)` 可以将一个子节点替换为另一个子节点。

删除节点首先需要得到要删除的节点的引用，随后再调用 `removeChild(childNode)` 方法。在删除子节点之后，建议调用 `unlink()` 方法强制对被删除的节点及它可能连接的子节点进行垃圾回收。该方法是 `minidom` 实现的特定方法，在 `xml.dom` 中不可用。

最后，对文档做出所有修改后，将 DOM 写回它的文件中将非常有用。`xml.dom.minidom` 有一个实用的方法 `toprettyxml`，它接收两个可选的参数：一个是缩进字符串，另外一个换行符。如果没有指定参数值，这两个参数分别默认为 `tabulator` 和 `\n`。该实用方法将 DOM 打印为包含良好缩进的 XML，正适合将 DOM 写回文件中。

### 试一试

### 使用 SAX 处理 XML

该示例表明如何利用 SAX 探究一个文档。

```
#!/usr/bin/python
from xml.sax      import make_parser
from xml.sax.handler import ContentHandler
#begin bookHandler
class bookHandler(ContentHandler):
    inAuthor = False
    inTitle = False
```

```

def startElement(self, name, attributes):
    if name == "book":
        print( "*****Book*****")
    if name == "title":
        self.inTitle = True
        print("Title: ",)
    if name == "author":
        self.inAuthor = True
        print("Author: ",)
def endElement(self, name):
    if name == "title":
        self.inTitle = False
    if name == "author":
        self.inAuthor = False
def characters(self, content):
    if self.inTitle or self.inAuthor:
        print(content)
#end bookHandler
parser = make_parser()
parser.setContentHandler(bookHandler())
parser.parse("library.xml")

```

### 示例说明

解析器 `xml.sax` 使用 `Handler` 对象处理解析文档过程中发生的事件。`Handler` 可能是 `ContentHandler`、`DTDHandler`、用于处理实体引用的 `EntityResolver` 或者 `ErrorHandler`。一个 SAX 应用程序必须实现符合这些接口的处理程序类，并为解析器设置处理程序。

接口 `ContentHandler` 包含了被文档事件触发的方法，例如元素和字符数据的开始和结束等。在解析字符数据时，解析器可以选择将结果作为一整块数据返回，或者作为若干小的以空白分隔的数据块返回，所以在处理一块文本的过程中需要反复调用 `characters` 方法。

`make_parser` 方法创建一个新的解析器对象并将它返回。它创建的解析器对象的类型是系统要第一个寻找的解析器类型。`make_parser` 方法也可以有一个可选参数，它包含一个可用解析器列表，这些解析器都必须实现 `make_parser` 方法。如果提供了该列表，`make_parser` 会首先尝试创建这些解析器，而不是默认列表中的解析器。

## 15.11 XSLT 简介

XSLT 代表可扩展样式表语言转换(Extensible Stylesheet Language Transformation)，用于将 XML 转换成输出格式，例如 HTML。XSLT 是一个过程化、模板驱动的语言。

### 15.11.1 XSLT 是 XML

像模式一样，XSLT 是用 XML 语言定义的，它用来补充 XML 的功能。XSLT 的名称空间是“<http://www.w3.org/1999/XSL/Transform>”，它指定了语言的结构和语法。像其他所有 XML 一样，XSLT 也可被验证。

### 15.11.2 转换和格式语言

XSLT 可以将一个 XML 语法转换为另一个语法, 或者转换为其他任意的基于文本的格式。XSLT 经常用来将 XML 转换为 HTML, 以便为 Web 表示做准备, 或者将一个自定义文档模型转换为 XSL-FO, 以便将其转换为 PDF 格式。

### 15.11.3 函数式、模板驱动

像 LISP 一样, XSLT 是一门函数式语言。XSLT 程序员声明了一系列模板, 这些模板是在文档中的节点匹配给定的 Xpath 表达式时触发的函数。程序员并不能保证执行顺序, 因此每个函数必须依靠自己, 不能假定其他函数的结果。

遗憾的是, Python 并不直接支持创建 XSLT。为了转换 XML 文档, 首先要创建 XSLT, 然后通过 Python 将其应用到 XML 文档。

此外, Python 的核心库并不提供通过 XSLT 转换 XML 的方法, 但是在其他库中有一些可用的方法。Fourthought 公司在其免费的 4Suite 包中提供了一个 XSLT 引擎, 但是在写这本书的时候, 该包还不支持 Python 3.0。然而, Python 也可以绑定到广泛使用的 libxslt C 库功能, 尤其是 lxml。

## 15.12 lxml 简介

下面的示例使用了 lxml 的最新版本, 在编写本书时, 最新版本是 2.2。如果还没有安装 lxml, 可以从 <http://pypi.python.org/pypi/lxml/> 下载最新版本。在本章后面的习题中需要用到它。

lxml 是 Python 利用 libxml2 和 libxslt 库的快速、丰富特性的唯一绑定, 并且它同时还通过一个简单的 API 允许既能处理 HTML 又能处理 XML。包 lxml 使用了略作修改的 ElementTree API, 试图使得用 libxml2 编程时尽量少出错。

导入 lxml 非常简单:

```
>>>import lxml
>>>from lxml import etree
```

## 15.13 元素类

元素是 ElementTree API 的主要容器对象, 提供了 XML 树功能的核心。它们的行为与列表类似, 实际上, 从技术上来说它们就是列表。它们拥有属性并且包含文本, 我们稍后将简单地讨论这一点。首先学习如何创建一个元素类。输入如下代码:

```
>>>author = etree.Element("Horror")
>>>print(author.tag)
Horror
```

在本示例中，创建了一个新的元素类 `author`，并给它赋予一个标签名称：`Horror`。随后使用 `print()` 函数和元素类的 `tag` 属性打印出它的标签名称。元素类遵守标准的 XML 树层次，因此既能支持父元素也能支持子元素。

假设希望 `author` 是根节点。`author` 元素的标签名称为“`horror`”，现在我们希望向元素类 `author` 中添加一组写恐怖小说的作者。这些新的元素现在变成了 `author` 元素的子元素。

### 试一试

### 创建子类

```
>>>author=etree.Element("Horror")
>>>writer1=etree.SubElement(author, "NeilGaiman")
>>>writer2=etree.SubElement(author, "StephenKing")
>>>writer3=etree.SubElement(author, "CliveBarker")
>>>print(etree.tostring(author))
b'<Horror><NeilGaiman/><StephenKing/><CliveBarker/></Horror>'
>>>writer=author[0]
>>>print(writer.tag)
NeilGaiman
>>>writer=author[1]
>>>print(writer.tag)
StephenKing
>>>for writer in author:
    print(writer.tag)

NeilGaiman
StephenKing
CliveBarker
```

### 示例说明

`SubElement` 的运行方式与您期望的类似，它就是一个元素的子元素。代码 `etree.SubElement(author,"NeilGaiman")` 告诉 Python 创建一个新的子元素 `writer1`，它的标签是 `NeilGaiman`，它的父元素是 `author`。添加 `writer2` 和 `writer3` 与添加 `writer1` 完全相同，使用 `etree.tostring` 方法打印 `author` 的子元素时可以看到这一点。

前面讲过元素类也是列表，因此，元素类中也可以使用列表函数。语句 `writer=author[0]` 将第一个子元素的值赋给变量 `writer`。同样，使用 `for writer in author` 代码可以处理 `author` 类中的每个子元素，并打印出它们的标签。

本章开始部分讨论了属性，元素可以包含属性，它们帮助进一步描述元素。

```
>>>author=etree.Element("author", audience="Adult")
>>>print(author.get("audience"))
Adult
>>>author.etree.Element("author", type="fiction", bestseller="Yes")
>>>print(author.get("type"))
Fiction
>>>print(author.get("bestseller"))
Yes
```

```
>>>print(author.get("audience"))
None
```

可以使用 `etree.Element` 向元素添加属性，如前面的代码所示。可以添加一个属性，或者 1000 个属性，但是要记住必须同时声明它们。在上面的示例中，首先给予 `author` 一个属性 `audience`，并且赋给该属性的值为 `Adult`。当使用 `author.get` 提取 `audience` 属性中的值(并用 `print` 函数显示该值)时，程序正常运行，打印出词 `Adult`。然而，我们试图再向 `author` 元素赋予两个属性(`type` 和 `bestseller`)，随后试图打印出 `audience`，但是得到结果 `None`。这是因为每次向元素赋予新的属性时，它会重写已经存在的属性。

有一个办法可以解决该问题。就像使用 `get()` 方法可以从元素中提取数据一样，我们还可以使用 `set()` 方法设置属性或者添加属性：

```
>>>author.etree.Element("author", type="fiction", bestseller="Yes")
>>>etree.tostring(author)
b'<author type="fiction" bestseller="Yes"/>'
>>>author.set("audience", "Adult")
>>> etree.tostring(author)
b'<author type="fiction" bestseller="Yes" audience="Adult"/>'
```

## 向元素添加文本

除了属性，还可以向元素中添加文本。当处理主要由数据驱动的 XML 文档时，唯一可以放置文本的地方就是在元素内部。实现此操作非常简单。在下面的示例中，我们创建了一棵类似于基本 HTML 文档的树：

```
>>>html=etree.Element("html")
>>>body=etree.SubElement(html,"body")
>>>h1=etree.SubElement(body, "h1")
>>>h1.text="Introduction"
>>>paragraph=etree.SubElement(body, "p")
>>>paragraph.text="Here is some text representing our paragraph"
>>>etree.tostring(html)
b'<html><body><h1>Introduction</h1>
<p>Here is some text</p></body></html>'
>>>etree.tostring(paragraph)
b'<p>Here is some text</p>'
```

上面的示例是展示父子关系的一种很好的方法。即使 `etree.tostring` 可以在一行上打印出结果，但是也许以实际的树结构浏览它会更好：

```
<html>
  <body>
    <h1>Introduction</h1>
    <p>Here is some text</p>
  </body>
</html>
```

可以看到，<html>是父节点，<body>是<html>的子节点，<h1>和<p>是兄弟节点，它们的父节点都是<body>。

在该情形下，文本属性允许我们显示用户可以看到的内容，而我们添加的任意属性只是提供关于元素本身的数据。

再讨论前面代码的最后一点。可以注意到我们不只使用 `etree.tostring` 打印出 HTML 的完整内容，还使用它专注于特定段落的内容。这是查看给定元素的内容的好方法，但是有时我们并不希望看到标签，只希望看到元素的文本。可以输入如下代码实现该功能：

```
>>>etree.tostring(paragraph, method="text")
b'Here is some text'
```

## 15.14 使用 lxml 解析

使用 lxml 解析非常直接。有三个可以选择的解析函数，每个都有自身的优缺点。每个函数都支持解析特定类型的对象，例如文件、字符串和 URL(HTTP 和 FTP)，其中最简单的是字符串解析函数 `fromstring()`。

在 lxml 提供的所有解析函数中，`fromstring()`是最简单易用的：

```
>>>sentence("<info>Here is a sentence</info>")
>>>info=etree.fromstring(sentence)
>>>print(info.tag)
info
>>>print(info.text)
Here is a sentence
```

在上面的代码中，通过使用开始和结束的<info>标签将其中嵌套的文本赋值给变量 `sentence`。我们随后创建了另一个变量 `info`，并使用 `etree.fromstring()`函数解析句子中的数据。

另一种解析方法是使用 `XML()`函数，它与 `fromstring()`函数类似，不同之处在于它直接将 XML 字面值写入源数据，如下所示：

```
>>>info=etree.XML("<info>Here is a sentence</info>")
>>>print(info.tag)
info
>>>print(info.text)
Here is a sentence
>>>etree.tostring(info)
b'<info>Here is a sentence</info>'
```

此处，我们跳过了创建 `sentence` 变量和给其赋值的初始步骤，而是使用 `XML()`函数直接将数据赋给了 `info`，节省了一个步骤。当打印 `info` 中的标签和文本时，得到了与前面一样的结果。

## 解析文件

到目前为止，本章已经介绍了如何解析简单的字符串。为了真正地理解 lxml 解析的强大功能，需要学习处理文件和与文件类似的对象，包括 URL 具有.read 方法的对象以及文件名称字符串。

与其他两个解析函数不同，函数 parse() 返回了一个 ElementTree 对象，而不是一个 Element 对象。这允许我们解析整个文档，而不仅仅是简单的 XML 片段：

```
>>>import io
>>>newsentence=io.StringIO("<info>This is another sentence</info>")
>>>somesentence=etree.parse(newsentence)
>>>etree.tostring(somesentence)
b'<info>This is another sentence</info>'
```

如果希望使用 print() 函数访问 somesentence 中的值，可以以如下方式实现该功能：

```
>>>printit=somesentence.getroot()
>>> print(printit.tag)
info
>>> print(printit.text)
This is another sentence
```

这只是涉及了可用 lxml 完成的功能的皮毛。关于该主题的完整介绍需要用两本书，这里没有足够的篇幅介绍它们。要获得更多信息，可以访问该模块的文档，网址为：<http://codespeak.net/lxml/index.html>。

## 15.15 本章小结

本章要点：

- 如何使用 SAX 和 DOM 解析 XML
- 如何使用 xmlproc 验证 XML
- 如何使用 HTMLParser 解析 HTML
- 如何使用 lxml

第16章将介绍网络编程以及电子邮件。在继续学习之前，尝试完成下面的习题，以测试您是否已经理解了本章所介绍的内容。在附录 A 中可以找到这些习题的答案。

## 15.16 习题

1. 给定 Python 应用程序的如下配置文件，编写利用 DOM 解析器提取配置信息的代码：

```
<?xml version="1.0"?>
<!DOCTYPE config SYSTEM "configfile.dtd">
<config>
```

```
<utilitydirectory>/usr/bin</utilitydirectory>
<utility>grep</utility>
<mode>recursive</mode>
</config>
```

2. 给定名为 `configfile.dtd` 的如下 DTD，编写一个 Python 脚本，验证前面的配置文件：

```
<!ELEMENT config (utilitydirectory, utility, mode)>
<!ELEMENT utilitydirectory (#PCDATA)*>
<!ELEMENT utility (#PCDATA)*>
<!ELEMENT mode (#PCDATA)*>
```

3. 使用 SAX 而不是 DOM，从前面的配置文件中提取配置信息。



# 第 16 章

## 网 络 编 程

最近十几年来,用户购买个人计算机的主要原因之一在于想满足上网的愿望,也就是以各种方式连接到全世界的其他计算机。网络连接,尤其是 Internet 连接,是个人计算机的一种招人喜爱的应用,这个特性使得大量不太懂计算机的人开始学习和购买个人计算机。

没有网络,也可以使用计算机做许多令人惊叹的事情,但是受众只局限于能够看到您的屏幕或读到您的输出的人,或者可以加载您发布的 CD 和 DVD 的那些人。如果将同一台计算机连接到 Internet,就可以与全世界范围内的人交流。

Internet 架构能够支持不计其数的应用,但是它本身有两个招人喜爱的应用,人们上网的主要目的就是使用它们。一个当然就是异常流行的互联网,第 20 章将介绍相关内容。

Internet 的另一个招人喜爱的应用是电子邮件,本章将详细介绍它。

本章将介绍:

- 如何使用标准库编写可以创建、发送以及接收电子邮件的应用程序。
- 如何创建一个能够以定制的格式发送和接收数据的程序。
- 套接字编程的基础知识。

### 试一试

### 发送电子邮件

Jamie Zawinski 是原来的 Netscape 公司的一名程序员,他曾经说过一句很有名的话:“每个程序都努力尝试扩展自己,以便能够阅读电子邮件。”这也许是真的(这也是 Netscape 浏览器的目标,甚至早在 Jamie 在那里工作时,他们就已经开始这样做了),但是在程序变为一个邮件阅读器之前,您可能发现需要它发几封邮件。邮件阅读器是典型的终端用户应用程序,但是几乎任何种类的应用程序都有理由发送邮件:监视软件、自动脚本、Web 应用程序,甚至游戏。E-mail 是经过时间验证的发送自动通知的方法,而自动通知在大量环境下都会发生。

Python 提供了一组复杂的类,用来构造 E-mail 消息,后面将对它们进行简单的介绍。实际上,一条 E-mail 消息仅仅是一个有着预定义格式的字符串。发送 E-mail 消息时,只需要一个符合格式的字符串、邮件要发送到的地址和 Python 的 `smtplib` 模块。下面这个简单的 Python 会话可以发送出一条最基本的 E-mail 消息:

```
>>> fromAddress = 'sender@example.com'
>>> toAddress = 'me@my.domain'
>>> msg = "Subject: Hello\n\nThis is the body of the message."
>>> import smtplib
>>> server = smtplib.SMTP("localhost", 25)
>>> server.sendmail(fromAddress, toAddress, msg)
{}
```

**注意:**

Smtplib 得名于 SMTP(Simple Mail Transport Protocol, 简单邮件传输协议)。SMTP 是为发送 Internet 邮件而定义的协议(或叫标准)。Python 打包了许多您可以轻松说出 Internet 协议名称的模块, 这些模块总是根据协议的名称命名, 例如 imaplib、poplib、httplib 和 ftplib 等。

在 me@mydomain 中放置自己的邮件地址, 如果您的计算机上有正在运行的邮件服务器, 就应当能够向自己发送邮件, 如图 16-1 所示。

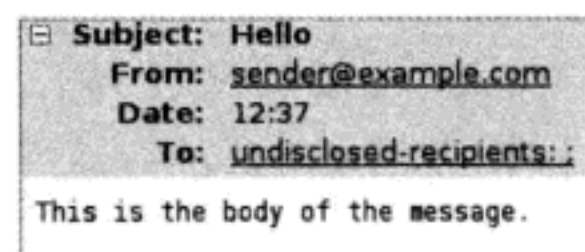


图 16-1

然而, 也许您的计算机上并没有运行邮件服务器(如果您正在一台共享的计算机上运行这些脚本, 可能计算机上已经有邮件服务器, 或者如果您自己设置了邮件服务器, 那么可能对网络已经有所了解, 正迫不及待地想阅读本章更高级的部分)。如果在运行脚本的计算机上没有邮件服务器, 那么在尝试实例化远程 SMTP 邮件服务器对象的时候会得到一个与下面类似的异常:

```
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    server=smtplib.SMTP("localhost",25)
  File "C:\Python31\lib\smtplib.py", line 239, in __init__
    (code, msg) = self.connect(host, port)
  File "C:\Python31\lib\smtplib.py", line 295, in connect
    self.sock = self._get_socket(host, port, self.timeout)
  File "C:\Python31\lib\smtplib.py", line 273, in _get_socket
    return socket.create_connection((host, port), timeout)
  File "C:\Python31\lib\socket.py", line 307, in create_connection
    raise error(msg)
socket.error: [Errno 10061] No connection could be made because the target
machine actively refused it)
```

这里发生了什么? 看一下引起这个异常的那行代码:

```
>>>server=smtplib.SMTP("localhost",25)
```

smtplib 类的构造函数尝试使用 Internet 协议(IP)启动一个网络连接。字符串“localhost”和数值 25 标识了假定邮件服务器的网络位置。由于您没有运行邮件服务器, 连接的另一端什么都没有, 当 Python 发现这个事实时, 它将无法继续。

为了理解“localhost”和 25 的含义, 有必要了解关于协议、尤其是 Internet 协议的一些知识。

## 16.1 理解协议

协议是组织网络上的两方或者多方之间发送数据的约定。这与在人与人之间礼仪或礼节扮演的角色类似。例如,假设您希望和朋友出去聚餐或者想与某人结婚。每种文化都定义了描述这些情形下的法律和社会行为的规范。当您外出就餐时,有关于如何在餐馆用餐、如何使用餐具和如何付账的礼仪。结婚过程则要符合仪式和婚约的规范,这些规范非常详细。

这两种活动很不同,但在它们背后都存在底层的社会协议。这些协议为一些事情设置了标准,例如礼仪和用语。打个比方,在最低层次上只是以某种方式振动自己的声带,但在较高层面上却是在通过说“我愿意”定下来这桩婚姻。如果违反了底层协议(比如,在餐馆就餐时表现粗鲁),那么达成高层目标的可能性就会降低。人类言行的协议的这些方面与计算机网络的协议有着一定的对应关系。

### 16.1.1 比较协议和程序语言

在过去的数十年里,人们为了各种可以想象的目的,发明了数以千计的网络协议,可以说网络发展的历史也是协议设计的历史。为什么会有如此多的协议?为了回答这个问题,考虑另外一个网络协议世界的对照物:为什么会有如此多的程序语言?网络协议之间的关系与程序语言之间的关系类似,人们创建新协议的原因与他们设计新的程序语言的原因相同。

不同程序语言的设计目的不同。例如,用 FORTRAN 语言写文本处理软件是件很疯狂的事情,并不是因为 FORTRAN 不好,而是因为它是为了数学和科学研究而设计的,并不适合终端用户的 GUI 应用程序。

类似地,不同协议的目的也不同。刚才简单看到的 SMTP 协议除发送邮件之外,还可以用于其他各种用途。但是并没有人这样做,因为使用 SMTP 完成最初设计它的目的,使用其他协议完成其他目的更有意义。

也有一些程序语言的设计目的是与其他程序语言竞争。新语言的创建者可能看到了已有语言在技术和优雅性上存在缺点,并希望通过新语言使他们自己的任务更简单。语言的创建者也许是贪求作为一个新语言的创建者所享有的荣誉和财富。而人们则会因为一种新类型的应用程序需要一个新协议而设计它。

一些程序语言是专门为了教学生学习编程而设计的,而在编程文化的另一端,是为了教他们编写编译器而设计的。一些语言为了探究新的思想而设计,而不是为了实际用途,但是另外一些语言则可能是一个公司将其作为一个与对手公司进行对抗的工具而设计的。

协议设计中也存在这些因素。一些公司有时会发明新的不可兼容的协议,试图从竞争对手那里抢生意。一些协议则是为了单纯的教学目的。例如,本章在教授网络编程的掩盖下,也会介绍如何为在线聊天室设计协议。已经有许多完美的实现该功能的协议,但是它们太过复杂,很难在这里详细介绍它们。

ADA 程序语言是由美国国防部设计,用于所有军事编程项目的公共语言。Internet 协议是为了使多个之前不兼容的网络可以相互通信而设计的,这也是协议称作“Internet”协议的原因。

目前,即使内部网络(内网)也通常运行在 Internet 协议上,但是原有的一些动机(解决

新问题、竞争等)仍旧在高层和底层的协议设计中起作用，这也是程序语言和协议快速增多的很有趣的一些原因。

### 16.1.2 Internet 协议栈

不同程序语言的抽象层次不同。Python 是一门非常高级的语言，可以完成任何任务，但是 Python 解释器并不是用 Python 编写的，而是用低级语言 C 编写的。C 程序则编译成符合您的计算机架构的机器语言。无论何时在 Python 解释器中输入一条语句，都存在一条从抽象代码到计算机代码的路径，甚至可以到达底层的驱动计算机的数字电路的操作。

注意：

有一个 Python 解释器用 Java 编写而成(Jython)，但是 Java 本身由 C 编写。PyPy 是用 Python 语言实现 Python 解释器的一个项目，但是 PyPy 运行在 C 或者 Java 实现上。您不可能脱离 C！

在某种意义上，当向 Python 解释器输入一条语句，计算机简单地完成您要求它做的事情。在另一种意义上，它运行了您输入的 Python 语句。第三层含义是，它执行了一长串 C 语句，这些 C 语句由 Python 的编写者编写，由您的 Python 语句激活。第四层含义是，计算机运行了非常长的一串几乎让人无法理解的机器码。第五层含义是，它实际上并没有“运行”任何程序：您只是引起了一系列发送到硬件的定时电子脉冲。我们需要高级程序语言的原因在于它们比低级语言更易使用。但这并不会使低级语言变得多余。

英语是一门高级的人类语言，可以胜任所有的任务，但是仅仅“说英语”还不是讲英语。为了讲英语，一个人必须实际发出一些声音，但是说话者不仅要发出声音，我们还得从脑部发送电波，将肺内的空气压迫到外边，并且持续地重新晃动舌头和嘴唇。这是个非常复杂的过程，但是我们甚至还没有考虑更低的层次，而只考虑了我们要说的词和试图传达的概念。

协议池可以根据抽象的层次，组织成一个类似的层次结构。在物理层次，也就是最底层，只有电子脉冲和电磁辐射。在物理层之上，每种类型的网络硬件都需要自己的协议，这些协议以软件实现(例如，在 LAN 上运行的网络的以太网协议)。物理层的电磁现象可看做设备之间发送和接收比特。这称作“数据链路层”。在协议栈中向上走时，这些原始的比特逐渐变得有意义：它们成为路由指令、命令、响应、图片、网页等。

由于不同的硬件以不同的方式通信，连接以太网和无线网时，需要比数据链路层更高层的协议。如前面提到的，当前大多数网络的公共协议是 Internet 协议(IP)，它实现了网络层，并将所有的网络连接在一起。IP 工作于“网络层”。

网络层的上一层是“传输层”，它使得通过 IP 发送的信息能够以正确的顺序可靠地到达目的地，并且不出错。IP 协议并不关心可靠性和错误检测：它仅接收数据和目的地址，在网络之间传送数据，并且假设数据能够完整地到达目的地址。

而 TCP(Transmission Control Protocol, 传输控制协议)则关心这些事情。TCP 实现了协议栈的传输层，使得网络上两点之间可靠的、有序的通信成为可能。通常，TCP 位于 IP 之上，这两个协议经常被当作一个协议处理，并被统称为 TCP/IP。

本章中您将学到和设计的所有网络协议都基于 TCP/IP。这些协议位于“应用层”，用来解决特定的用户问题。即使是非程序员，也可能知道这些协议：您也许听说过 HTTP、FTP 和 BitTorrent 等。

人们通常会考虑在应用层设计协议，这是最适合 Python 实现的层。目前另一个颇受关注的地方是在数据链路层的另一端：将新类型的设备连接到 Internet 的嵌入式系统编程。由于 Internet 如此流行，TCP/IP 或多或少地占据了协议栈的中间位置。

### 16.1.3 Internet 协议简介

现在您已经理解了 Internet 协议适合于计算机使用的协议栈中的哪一层，还需要了解它的另外两方面的知识：地址和端口。

#### 1. Internet 地址

Internet(或者私有的 TCP/IP 网络)上的每台计算机都有一个或多个 IP 地址，通常表示为一个用句点分开的 4 个数值的序列，如“208.215.179.178”。同一台计算机也许有多个像“wrox.com”一样的主机名称。

为了连接到某个计算机上运行的服务，需要知道它的 IP 地址，或者它的一个主机名(主机名由 DNS 管理。DNS 是运行在 TCP/IP 协议之上的协议，自动将主机名转换为 IP 地址)。回忆本章之初发送邮件的脚本。当它尝试连接到一个邮件服务器时，它用到了看起来很神奇的字符串“localhost”：

```
>>> server = smtplib.SMTP("localhost", 25)
```

“localhost”是特殊的主机名，在引用它时，它指向您正在使用的计算机(每个计算机也有一个特殊的 IP 地址指向自己：127.0.0.1)。主机名告诉 Python 在 Internet 上的什么位置寻找邮件服务器。

注意：

通常，使用主机名比使用 IP 地址要好，即使前者立即被转换为后者。主机名一般比 IP 地址更稳定。在运作的另一个协议栈的示例是，DNS 协议可以将 IP 寻址机制的细节隐藏起来。

当然，如果计算机上没有运行邮件服务器，“localhost”将不能工作。给您提供 Internet 访问的组织应该会允许您使用它的邮件服务器，可能位于 mail.[您的 ISP].com 或者 smtp.[您的 ISP].com。不论使用什么样的邮件客户端，它的配置中的某处会有邮件服务器的主机名称，因此可以使用它发送邮件。使用邮件服务器的主机名称替换前面的示例代码中的“localhost”，您应当可以从 Python 中发送邮件：

```
>>> fromAddress = 'sender@example.com'
>>> toAddress = '[your e-mail address]'
>>> msg = "Subject: Hello\n\nThis is the body of the message."
```

```
>>> import smtplib
>>> server = smtplib.SMTP("mail.[your ISP].com", 25)
>>> server.sendmail(fromAddress, toAddress, msg)
{}
```

**注意：**

遗憾的是，您也许由于某些原因仍然无法发送邮件。SMTP 服务器也许要求身份验证，但该示例会话没有提供。它也许不接受从运行脚本的计算机发出的邮件(尝试您通常用来发送邮件的计算机)。它也许运行于一个非标准的端口之上(见下节)。服务器也许并不喜欢这个简化的信息格式，而希望收到“真正”的 E-mail 消息；如果是这样，下节描述的 e-mail 模块也许可以提供帮助。如果都行不通，就要向系统管理员寻求帮助。

## 2. 网络接口

字符串“localhost”被解释为掩盖 IP 地址的 DNS 主机名。现在只剩下了一个神秘的数字 25。这意味着什么？考虑这样一个事实：一个计算机可能托管多个服务。拥有一个 IP 地址的计算机可能拥有 Web 服务器、邮件服务器、数据库服务器和其他一些服务器。客户端如何区分到不同服务器的连接，例如到 Web 服务器的连接和到数据库服务器的连接？

实现 Internet 协议的计算机有 65 536 个已编号的端口。当启动 Internet 服务器(比如一个 Web 服务器)时，服务器进程将自己“绑定”到计算机的一个或者多个端口(例如端口 80，这是 Web 服务器的常规端口)，并开始监听该端口的外部连接。如果您见到过与“http://www.example.com:8000”类似的网络地址，这个数字是 Web 服务器的端口号，不过这里列出的端口号违反了约定。约定的实施者为互联网数字分配机构(Internet Assigned Numberd Authority, IANA)。

**注意：**

IANA 的协议列表和常规端口号公布于网址 [www.iana.org/assignments/port-numbers](http://www.iana.org/assignments/port-numbers)。

根据 IANA，SMTP 的常规端口号为 25。所以前例中 SMTP 对象的构造函数接收 25 作为它的第二个参数(如果没有指定任何端口号，SMTP 构造函数将默认它为 25)：

```
>>> server = smtplib.SMTP("localhost", 25)
```

IANA 将端口划分为“公认端口”(端口号为 0~1023)、“注册端口”(端口号为 1024~49 151)和“动态端口”(端口号为 49 152~65 535)。在大多数操作系统上，必须有管理员权限才能将服务器绑定到一个公开端口，因为绑定到那些端口的进程通常有管理员权限。任何人都可以将服务器绑定到注册范围内的端口，对于本章中编写的定制服务器就将执行这种操作。动态范围通常用于客户端，而不是服务器，后面在讨论套接字时会进行介绍。

## 16.2 发送电子邮件

基本理解了 TCP/IP 的工作原理后，本章开始处的 Python 会话看起来会更有意义：

```
>>> fromAddress = 'sender@example.com'
>>> toAddress = 'recipient@example.com'
>>> msg = "Subject: Hello\n\nThis is the body of the message."
>>> import smtplib
>>> server = smtplib.SMTP("localhost", 25)
>>> server.sendmail(fromAddress, toAddress, msg)
{}

```

如果计算机上并没有运行 SMTP 服务器，您现在就能够找到可以使用的主机名和端口号。对于前面的代码，唯一没有作出解释的方面是 E-mail 消息为什么看起来是这个样子。

### 16.2.1 E-mail 文件格式

除了创建了大量的 E-mail 相关的协议之外，网络工程师还设计了一组对 E-mail 消息各部分打包的文件格式。这些协议和文件格式公布于几个称作 RFC 的编号文档中。

您在本章中开始编写自己的协议之前，一直都在使用其他人设计并在 RFC 中指定的协议和格式。这些文档通常包含正式的语言规范和说明，但是文档中的大部分内容非常易读。

当前定义 E-mail 消息格式的标准是 RFC 2822。它公布于 2001 年，对令人尊敬的 RFC 822 做了更新，RFC822 可以追溯到 1982 年。您也许仍然能够看到“RFC822”作为“E-mail 格式”被引用，例如 Python 弃用的 rfc822 模块。

**注意：**

为了找到特定的 RFC，可以在网上搜索“RFC x”，或者参考位于 [www.ietf.org/rfc.html](http://www.ietf.org/rfc.html) 的官方网址。RFC 2822 位于 [www.ietf.org/rfc/rfc2822.txt](http://www.ietf.org/rfc/rfc2822.txt)。

一条 E-mail 消息包含一组头(描述消息的元数据)和主体(消息本身)。邮件头以键-值对的形式发送，其中冒号和空格将键与值分开(例如，“Subject: Hello”)。邮件主体就是消息的文本部分。

您可以利用 Python 的 e-mail 模块中的 Message 类创建符合 RFC 2822 的信息。Message 对象与字典的行为类似，它将消息头的名称映射到它们的值。它也有一个“负载”，就是消息的主体文本：

```
>>>import os
>>>import sys
>>>import smtplib
>>>import mimetypes
>>>from optparse import OptionParser
>>>from e-mail import encoders
>>>from e-mail.message import Message
>>>message=Message()
>>>message['Subject']='Hello'
>>>message.set_payload('This is the body of the message')
>>>print(str(message))

Subject: Hello
This is the body of the message

```

上面的代码并不仅仅指定了 E-mail 字符串，这样可以减少犯错误的机率，尤其是对于复杂的消息更是如此。还可以注意到您得到了没有输入到消息中的信息。这是因为 `smtplib` 在您发送消息时向其中添加了必要的头信息。

表 16-1 列出了 RFC 2822 定义的一些标准的消息头。它还是一些消息头的值定义了数据表示标准(例如，它定义了 E-mail 地址和日期的表示方式)。标准还给您提供了空间，允许在发送和接收 E-mail 的程序中定制消息头。

表 16-1

头	示 例	目 的
To	To: Leonard Richardson<leonardr@example.com>	消息接收人的地址
From:	From: Peter C. Norton<peter@example.com>	消息发送者的地址
Date	Date: Wed, 16 Mar 2009 14:36:07-0500 (EST)	消息的发送日期
Subject	Subject: Python book	消息的摘要或者标题，以便于阅读者理解
Cc	Cc:Michael@example.com, Jason Diamond<Jason@example.com>	消息的抄送者地址，即使邮件并不是直接写给他们的

注意 RFC 2822 对消息主体内容有几个限制。RFC 2822 要求消息主体中每行文本不多于 1000 个字符。一个更加繁重的限制是消息的头和主体只能包含 U.S.ASCII 字符(即 ASCII 中的前 127 个字符)，而不允许包含国际字符或者二进制字符。这个限制看起来并没有道理，因为您也许已经见到过用其他语言写的 E-mail 消息。下面将对此进行解释。

16.2.2 MIME 消息

如果 RFC 2822 要求 E-mail 消息仅包含 U.S.ASCII 字符，如何使人们发送带有图片和其他二进制文件附件的邮件？可以利用 RFC 2822 的扩展标准 MIME(Multi-purpose Internet Mail Extension，多用途 Internet 邮件扩展协议)实现该目的。

MIME 是用来将非 U.S.ASCII 数据融入到由 127 个 7 位字符组成的 U.S.ASCII 中的一系列标准。多亏了 MIME，您可以向电子邮件消息添加二进制附件，用非英语字符写消息甚至消息头(例如您的姓名)，并使得消息内容在另一端正确地显示(假设另一端理解 MIME，目前几乎所有的邮件终端都能理解 MIME)。

主要的 MIME 标准是 RFC 1521，它描述了如何将二进制数据写入 E-mail 消息中。RFC 1522 描述了如何在 E-mail 消息的头部实现该功能。

1. MIME 编码：quoted-printable 编码和 Base64 编码

MIME 最重要的部分是它的编码，它提供了将 8 位字符编码为 7 位字符的机制。MIME 定义了两种编码方式：quoted-printable 编码和 Base64 编码。Python 提供了将字符串转换为每种编码或者从每种编码转换为字符串的模块。

quoted-printable 编码用于仅包含一些 8 位字符的文本，其中大多数字符是 U.S.ASCII

字符。quoted-printable 编码的优点是文本在编码后仍旧可读，这使它非常适合表示用西欧语言(语言中的大部分字符都可以表示为 U.S.ASCII 字符，但有少数使用变音符号的字符是例外)编写的文本。即使消息的接收者不能对 quoted-printable 消息进行解码，他们也能够阅读消息。只是他们将看到词中间夹杂一些奇怪的等号和十六进制数值。

Python 中用于编码和解码的模块是 quopri:

```
>>> import quopri

>>> encoded = quopri.encodestring(bytes("I will have just a
soupçon of soup.", 'utf-8'))
>>> print(encoded)
I will have just a soup=E7on of soup.
>>> print(quopri.decodestring(encoded))
I will have just a soup\xe7on of soup.
```

取决于终端设置，您也许看到了最后一行的“ç”字符，也许看到的是“\xe7”。“\xe7”是 Python 中“ç”的字符串表示，就像“\E7”是 quoted-printable 表示一样。在前面的代码重现的会话中，字符串被编码为 Python 字符串，之后再次编码为 Python 特定的显示格式(注意，str 对象封装在一个 bytes 对象中，因为 encodestring 方法需要一个字符串或者缓冲区对象。一个 str 实际是一个字符列表，与字节列表不同)。

Base64 编码则用于二进制数据。它不应当用于人类可读的文本，因为它完全掩盖了文本：

```
>>> import base64

>>> encoded = base64.encodestring(bytes("I will have just a
soupçon of soup.", 'utf-8'))
>>> print(encoded)
SSB3aWxsIGhhdmUganVzdCBhIHNVdXBvbiBvZiBzb3VwLg==
>>> print(base64.decodestring(encoded))
I will have just a soupçon of soup.
```

既然 quoted-printable 可以处理所有数据，并且不破坏人类可读的文本，为什么还要使用 Base64 编码？原因有二：有些数据并不是“printable”，而硬将它们编码为“quoted-printable”格式会引起误导；Base64 编码比 quoted-printable 编码在表示二进制数据时更有效率。下面使用一长串随机的二进制字符对这两种编码方式进行了比较。

```
>>> import random
>>> import quopri
>>> import base64
>>> length = 10000
>>> randomBinary = ''.join([chr(random.randint(0,255)) for x in range(0,
length)])
>>> len(quopri.encodestring(bytes(randomBinary, 'utf-8'))) / float(length)
2.0663999999999998
>>> len(base64.encodestring(randomBinary)) / float(length)
1.3512
```

这些数值在不同的运行过程中会有所变化，因为这里的字符串是随机产生的，但是如果您尝试做该试验，每次都得到相似的结果。将一个二进制字符串编码为 quoted-printable 在邮件中发送是安全的，但是它(平均)是原始的不可发送的数据的两倍长。用 Base64 编码的同一个二进制字符串也同样安全，但是只有原始数据的 1.35 倍的长度。使用 Base64 对大多数的二进制数据进行编码将节省空间和带宽。

同时，用 Base64 编码 ASCII 字符串就有些过犹不及，因为它包含几个在 U.S.ASCII 范围之外的字符。这里使用随机的几乎完全由 U.S.ASCII 字符构成的字符串对这两种编码方式进行了比较：

```
>>> import random
>>> import quopri
>>> import base64
>>> length = 10000
>>> randomBinary = ''.join([chr(random.randint(0,128)) for x in range(0,
length)])
>>> len(quopri.encodestring(bytes(randomBinary,'utf-8'))) / float(length)
1.0661
>>> len(base64.encodestring(bytes(randomBinary,'utf-8'))) / float(length)
1.3512
```

此处，quoted-printable 编码只比原来的文本长一点(几乎与原来的文本同样大小)，但是 Base64 与前例一样，是原来文本的 1.35 倍长。这就表明为什么 MIME 支持两种不同类型的编码方式：按照 RFC1521 的说法，“一种‘可读的’编码[quoted-printable]和一种‘密集型’编码[Base64]”。

注意：

MIME 比它名称揭示的意义还要更“多用途”。E-mail 之外的一些应用程序中也使用了 MIME 的许多特征。使用 Base64 或者 quoted-printable 编码将非 ASCII 字符转换成 ASCII 字符的思想在其他领域也有出现。Base64 编码有时也用来在不对文本做加密的前提下使文本变得隐晦而不易阅读。

## 2. MIME 内容类型

MIME 的另外一个重要之处是关于内容类型的思想。假设您向朋友发送了一条 E-mail 消息：“Here's that picture I took of you”，并且添加了一个图片附件。通过使用 Base64 编码，接收者将得到发送时进行编码的数据，但是邮件阅读器如何才能知道这是一幅图片，而不是其他形式的二进制数据？

MIME 通过定义一个定制的 RFC 2822 格式的邮件头解决该问题。这个头名为 Content-Type，它描述了邮件主体的文件类型，因此接收者的邮件客户端可以知道如何显示它。内容类型包括 text/plain(如果将一封普通的 E-mail 消息封装在 MIME 中，将得到该类型)、text/html、image/jpeg、video/mpeg 和 audio/mp3 等。每种内容类型都有一个“主类型”和“子类型”，用斜杠分隔。多数类型比较通用，且只有 7 个，定义在 MIME 标准中。子类型通常指定特定类型的文件格式。

注意:

有“内容类型”的字符串能够告诉接收者如何处理它,这是 MIME 用在 E-mail 世界之外的又一发明。最常见的用途是用于 HTTP 中,第 20 章将会介绍,HTTP 是万维网使用的协议。每一个 HTTP 响应都应当有一个“Content-type”头(就像 MIME E-mail 消息一样),它告诉 Web 浏览器如何显示该响应。

### 试一试

### 创建带附件的 MIME 消息

到目前为止一切顺利。Python 为 e-mail 模块提供了许多子模块,用于构建 MIME 消息,其中包括针对每种主内容类型的模块。使用这些模块可以很容易地制作一个包含已编码的图片文件的 MIME 消息。

```
import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.image import MIMEImage
msg=MIMEMultipart()
filename=('C:\Python30\photos.jpg')
msg['To']='jamesrobertpayne@yahoo.com'
msg['From']='james@developershed.com'
msg['Subject']='Some picture'
pic = open('C:\Python30\photos.jpg', 'rb')
img = MIMEImage(pic.read())
pic.close()
print(str(msg))
msg.attach(img)
sendit = smtplib.SMTP()
sendit.connect()
sendit.sendmail(me, family, msg.as_string())
sendit.close()
```

注意:

当然,您应当将“photos.jpg”替换为您手头的任何图片文件的名称。只需将该文件放到调用 Python 会话的目录即可。

因为您告诉 MIMEImage 的构造函数该图片的名称是“photo.jpg”,另一端的邮件客户端将以这个名称存储该图片。

### 3. MIME 多部分消息

还有一个问题。这并不完全是之前描述的 E-mail 消息。之前的 E-mail 消息是一个短文本(“Here's that picture I took of you”)和一个图片附件。而这个消息仅仅包含图片。消息主体没有用于容纳文本的部分,在文件主体内放置文本将破坏图片文件。邮件消息的 Content-Type 头可以是 text/plain 或者 image/jpeg,但不能两者都是。那么邮件客户端如何创建带附件的消息?

除了分类其他标准定义的文件格式(例如,针对图片文件格式的 image),MIME 还定义

了一个称作 **multipart** 的特殊主类型。主内容类型为 **multipart** 的消息可以在它的主体内包含其他 **MIME** 消息，每个 **MIME** 消息都有自己的邮件头和自己的内容类型。

观察 **multipart** 内容类型的工作原理的最好的方法是使用 **e-mail.mime.multipart** 模块创建一个多部分消息，并使用 **e-mail.mime\*** 模块创建作为附件的文件。这里有一个称作 **FormatMimeMultipartMessage.py** 的脚本，是比前面示例略微复杂的一个版本：

```
#!/usr/bin/python
from e-mail.mime.multipart import MIMEMultipart
import os
import sys
filename='C:\Python30\photos.jpg'
msg = MIMEMultipart()
msg['From'] = 'Me <me@example.com>'
msg['To'] = 'You <you@example.com>'
msg['Subject'] = 'Your picture'
from e-mail.mime.text import MIMEText
text = MIMEText("Here's that picture I took of you.")
msg.attach(text)
from e-mail.mime.image import MIMEImage
image = MIMEImage(open(filename, 'rb').read(),
name=os.path.split(filename)[1])
msg.attach(image)
```

运行该脚本，传入一个图片文件的路径，然后将看到一个 **MIME** 多部分 **E-mail** 消息，它包含了一个简洁的文本消息以及一幅用 **Base64** 编码的图片文件：

```
# python FormatMimeMultipartMessage.py ./photo.jpg
From nobody Sun Jun 20 15:41:23 2009
Content-Type: multipart/mixed; boundary="=====1011273258=="
MIME-Version: 1.0
From: Me <me@example.com>
To: You <you@example.com>
Subject: Your picture

=====1011273258==
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

Here's that picture I took of you.
=====1011273258==
Content-Type: image/jpeg; name="photo.jpg"
MIME-Version: 1.0
Content-Transfer-Encoding: base64

/4AAQSkZJRgABAQEASABIAAD//gAXQ3JlYXRlZCB3aXRoIFRoZSBHSU1Q/9sAQwAIBgYHBgUI
...
[As before, much base64 encoded text omitted.]
```

```
...
3f7kklh4dg+UTZ1TsAAv1F69UklmZ9hrzogZibOqSSA8gZySSSJI/9k=
-----1011273258==
```

发送该消息时，它将在另一端显示为一个符合预期的带附件的消息。发送带附件的消息时，您的 E-mail 客户端创建的就是这一类 E-mail。

这个 E-mail 有几个特性值得一提：

- 仅有内容类型(multipart/mixed)还不足以使邮件主体有意义。MIME 还需要对“边界”进行定义，边界是由 Python 半随机地生成的字符串，用来在邮件中标记一部分结束和另一部分开始的地方。
- 该消息作为一个整体拥有您关联到 E-mail 消息的所有头：Subject、From、To，还有 MIME 特定的 Content-header 头。除此之外，消息的每个部分都有各自的头部。这些并不是消息头，尽管它们的格式也符合 RFC 2822 头格式；还有一些头(MIME-Version 和 Content-Type)既在消息头部也在消息主体中出现。这些是 MIME 消息主体的头部，由 MIME 解析器解释。就 RFC 2822 而言，它们是消息体的一部分，类似于它们描述的文件，将 MIME 的不同部分进行区分的边界，以及文本“Here's that picture I took of you”。
- 包含消息主体的 MIME 部分的编码方式为 7bit。这意味着该部分根本就没有编码。该部分中的所有字符都是 U.S.ASCII 字符，因此没有必要对它们编码。

只要知道希望构建哪种类型的邮件，Python 中关于邮件的类将起到非常大的作用：对于只有文本的消息，使用简单的 e-mail.message 类即可。如果希望向消息中添加文件附件，需要使用 e-mail.mime\* 中的类。如果希望发送多个文件，或者发送文本和文件的组合，结合使用类 e-mail.mime.multipart 和其他的 e-mail.mime\* 类。

当事先不能确定要使用哪个类表示邮件消息时，就会出现这个问题。这里有一个称作 SmartMessage 的类，可以创建 E-mail 消息，它将主体文本保存在简单的 Message 表示中，但是如果添加了一个附件，就改为使用 MimeTypeMultipart。这个策略将产生与典型的终端用户邮件应用程序相同范围的 E-mail 消息主体：针对简单消息的简单的 RFC 2822 主体，以及针对带附件的消息的复杂的 MIME 主体。将这个类放在文件 SendMail.py 中：

```
from e-mail import encoders as Encoders
from e-mail.message import Message
from e-mail.mime.text import MIMEText
from e-mail.mime.multipart import MIMEMultipart
from e-mail.mime.nonmultipart import MIMENonMultipart
import mimetypes

class SmartMessage:
    """A simplified interface to Python's library for creating e-mail
    messages, with and without MIME attachments."""
    def __init__(self, fromAddr, toAddrs, subject, body):
        """Start off on the assumption that the message will be a simple RFC
        2822 message with no MIME."""
        self.msg = Message()
        self.msg.set_payload(body)
```

```

self['Subject'] = subject
self.setFrom(fromAddr)
self.setTo(toAddrs)
self.hasAttachments = False
def setFrom(self, fromAddr):
    "Sets the address of the sender of the message."
    if not fromAddr or not type(fromAddr)==type(''):
        raise Exception ('A message must have one and only one sender.')
    self['From'] = fromAddr
def setTo(self, to):
    "Sets the address or addresses that will receive this message."
    if not to:
        raise Exception ('A message must have at least one recipient.')
    self._addresses(to, 'To')
    #Also store the addresses as a list, for the benefit of future
    #code that will actually send this message.
    self.to = to
def setCc(self, cc):
    """Sets the address or addresses that should receive this message,
    even though it's not addressed directly to them ("carbon-copy")."""
    self._addresses(cc, 'Cc')
def addAttachment(self, attachment, filename, mimetype=None):
    "Attaches the given file to this message."
    #Figure out the major and minor MIME type of this attachment,
    #given its filename.
    if not mimetype:
        mimetype = mimetypes.guess_type(filename)[0]
    if not mimetype:
        raise Exception ("Couldn't determine MIME type for ", filename)
    if '/' in mimetype:
        major, minor = mimetype.split('/')
    else:
        major = mimetype
        minor = None
    #The message was constructed under the assumption that it was
    #a single-part message. Now that we know there's to be at
    #least one attachment, we need to change it into a multi-part
    #message, with the first part being the body of the message.
    if not self.hasAttachments:
        body = self.msg.get_payload()
        newMsg = MIMEMultipart()
        newMsg.attach(MIMEText(body))
        #Copy over the old headers to the new object.
        for header, value in self.msg.items():
            newMsg[header] = value
        self.msg = newMsg
        self.hasAttachments = True
    subMessage = MIMENonMultipart(major, minor, name=filename)
    subMessage.set_payload(attachment)
    #Encode text attachments as quoted-printable, and all other

```

```

#types as base64.
if major == 'text':
    encoder = Encoders.encode_quopri
else:
    encoder = Encoders.encode_base64
encoder(subMessage)
#Link the MIME message part with its parent message.
self.msg.attach(subMessage)
def _addresses(self, addresses, key):
    """Sets the given header to a string representation of the given
    list of addresses."""
    if hasattr(addresses, '__iter__'):
        addresses = ', '.join(addresses)
    self[key] = addresses
#A few methods to let scripts treat this object more or less like
#a Message or MultipartMessage, by delegating to the real Message
#or MultipartMessage this object holds.
def __getitem__(self, key):
    "Return a header of the underlying message."
    return self.msg[key]
def __setitem__(self, key, value):
    "Set a header of the underlying message."
    self.msg[key] = value
def __getattr__(self, key):
    return getattr(self.msg, key)
def __str__(self):
    "Returns a string representation of this message."
    return self.msg.as_string()

```

**试一试****使用 SmartMessage 创建 E-mail 消息**

为了测试 SmartMessage, 将其放入文件 SendMail.py 中, 并且运行如下所示的 Python 会话:

```

>>> from SendMail import SmartMessage
>>> msg = SmartMessage("Me <me@example.com>", "You <you@example.com>", "Your
picture",
"Here's that picture I took of you.")
>>> print (str(msg))
Subject: Your picture
From: Me <me@example.com>
To: You <you@example.com>

Here's that picture I took of you.
>>> msg.addAttachment(open("photo.jpg").read(), "photo.jpg")
>>> print (str(msg))

Content-Type: multipart/mixed; boundary="=====1077328303=="
MIME-Version: 1.0

```

```

Subject: Your picture
From: Me <me@example.com>
To: You <you@example.com>

-----1077328303==
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

Here's that picture I took of you.
-----1077328303==
Content-Type: image/jpeg
MIME-Version: 1.0
Content-Transfer-Encoding: base64

/9j/4AAQSkZJRgABAQEASABIAAD//gAXQ3JlYXRlZCB3aXRoIFRoZSBHSU1Q/9sAQwAIBgYHBgUI
...
[Once again, much base64 text omitted.]
...
3f7kklh4dg+UTZlTsAAv1F69UklmZ9hrzogZibOqSSA8gZySSSJI/9k=
-----0855656444=--

```

### 示例说明

`SmartMessage` 类封装于 Python 的 `e-mail` 模块中的类。当首次创建 `SmartMessage` 对象时，它在 `Message` 对象中保存它的内部表示。该消息拥有一个简单的字符串表示。

但是，当向 `SmartMessage` 附加文件时，`Message` 对象将不再能满足要求。`Message` 对象只了解 RFC 2822，而对 MIME 扩展一无所知。此处，`SmartMessage` 用相同的头部和负载以透明的方式交换出了 `MimeMultipart` 对象中的 `Message` 对象。

这种透明的交换避免了强迫用户事先决定一个消息是否为 MIME 编码。它还避免了 MIME 编码每条消息的低耦合策略，这对只包含文本部分的消息是多余的操作。

### 16.2.3 使用 SMTP 和 `smtplib` 发送邮件

现在您已经知道了如何构建 E-mail 消息，是时候稍微回顾一下用来发送邮件的协议的更多细节了。该协议是 SMTP，它是另外一个基于 TCP/IP 的协议，在 RFC 2821 中定义。

再次查看原来的示例：

```

>>> fromAddress = 'sender@example.com'
>>> toAddress = [your e-mail address]
>>> msg = "Subject: Hello\n\nThis is the body of the message."
>>> import smtplib
>>> server = smtplib.SMTP("localhost", 25)
>>> server.sendmail(fromAddress, toAddress, msg)
{}

```

您连接到一个 SMTP 服务器(`localhost` 上的端口 25)，并且从一个地址向另外一个地址

发送了一条字符串消息。当然，不应当对 SMTP 服务器的位置进行硬编码，而且由于一些服务器需要身份验证，因此如果在创建 SMTP 对象时能够接收身份验证信息会更好。为了能够更简单地发送邮件，下面的类使用了前一节定义的 `SmartMessage` 类。由于这两个类在一起协同工作，将该类添加到文件 `SendMail.py` 中，该文件也包含了 `SmartMessage` 类：

```
from smtplib import SMTP
class MailServer(SMTP):

    "A more user-friendly interface to the default SMTP class."

    def __init__(self, server, serverUser=None, serverPassword=None, port=25):
        "Connect to the given SMTP server."
        SMTP.__init__(self, server, port)
        self.user = serverUser
        self.password = serverPassword
        # Uncomment this line to see the SMTP exchange in detail.
        #self.set_debuglevel(True)

    def sendMessage(self, message):
        "Sends the given message through the SMTP server."
        #Some SMTP servers require authentication.
        if self.user:
            self.login(self.user, self.password)

        #The message contains a list of destination addresses that
        #might have names associated with them. For instance,
        #"J. Random Hacker <jhacker@example.com>". Some mail servers
        #will only accept bare e-mail addresses, so we need to create a
        #version of this list that doesn't have any names associated
        #with it.
        destinations = message.to
        if hasattr(destinations, '__iter__'):
            destinations = map(self._cleanAddress, destinations)
        else:
            destinations = self._cleanAddress(destinations)
        self.sendmail(message['From'], destinations, str(message))

    def _cleanAddress(self, address):
        "Transforms 'Name <e-mail@domain>' into 'e-mail@domain'."
        parts = address.split('<', 1)
        if len(parts) > 1:
            #This address is actually a real name plus an address:
            newAddress = parts[1]
            endAddress = newAddress.find('>')
            if endAddress != -1:
                address = newAddress[:endAddress]
        return address
```

**试一试****使用 MailServer 发送邮件**

本章的第一个示例构造了一条字符串消息,并通过 SMTPlib 发送它。通过 SmartMessage 类和 MailServer 类,可以使用较为简单的 Python 代码发送一条更加复杂的消息:

```
>>> from SendMail import SmartMessage, MailServer
>>> msg = SmartMessage("Me <me@example.com>",
                        "You <you@example.com>",
                        "Your picture",
                        "Here's that picture I took of you.")
>>> msg.addAttachment(open("photo.jpg").read(), "photo.jpg")
>>> MailServer("localhost").sendMessage(msg)
>>>
```

运行上面的代码(替换为合适的 E-mail 地址和服务主机名称),您将可以向任何人发送带 MIME 附件的邮件。

**示例说明**

SmartMessage 类封装于 Python 的 e-mail 模块。同之前一样,底层的表示最初是简单的 Message 对象,但是一旦添加了图片附件 photo.jpg,底层表示立即变为了 MimeMultipart 对象。

这一次,消息通过 SMTP 服务器发送。MailServer 类隐藏了 smtpplib 希望您两次指定 To 和 From 头的事实:一次是在调用 sendmail 方法时,第二次是在邮件消息主体中。它还对目的地址进行处理,将它们转换成所有 SMTP 服务器都可以处理的形式。在这两个包装器类中,从 Python 脚本发送复杂的 E-mail 消息就像从邮件客户端发送一样简单。

## 16.3 检索 Internet 邮件

现在,您已经知道如何发送邮件,接下来就满足 Jamie Zawinski 的预言,对您的程序进行扩展,使它们可以阅读邮件。有三种方法可以实现该目的,但是选择权可能不在您。检索邮件的方式取决于您与提供 Internet 接入的组织之间的关系。

### 16.3.1 使用 mailbox 解析本地邮筒

如果邮件服务器上有 UNIX shell 的账户(例如,在自己的计算机上运行邮件服务器时),您新收到的邮件会追加到一个文件末尾(可能是/var/spool/mail/[your username])。如果这是您为邮件设置的工作方式,现在的邮件客户端可能被设置为解析该文件。邮件客户端可能还会被设置为将新收到的邮件从邮筒文件中移动到主目录下。

目录/var/spool/mail/中的收件箱用一种特定的格式(mbox 格式)保存。您可以利用 mailbox 模块中的类解析这些文件(以及其他格式的邮件箱,例如 MH 或者 Maildir)。

这里有一个简单的脚本 MailboxSubjectLister.py,它遍历邮箱文件中的消息,打印出每

条消息的主题:

```
#!/usr/bin/python
import e-mail
import mailbox
import sys
if len(sys.argv) <2:
    print("Usage: %s [path to mailbox file]" % sys.argv[0])
    sys.exit([1])
path = sys.argv[1]
fp = open(path, 'rb')
subjects = []
for message in mailbox.PortableUnixMailbox(fp, e-mail.message_from_file):
    subjects.append(message['Subject'])
print('s message(s) in mailbox "%s":' % (len(subjects), path))
for subject in subjects:
    print('', subject)
```

UnixMailbox(以及 mailbox 模块中的其他 Mailbox 类)用一个文件对象(邮箱文件)作为它们的构造函数,并接受一个从文件类型对象中读取下一条消息的函数。在本例中,该函数是 e-mail 模块的 message\_from\_file。这个函数的输出是一个 Message 对象,或者是其某个 MIME \*子类,例如 MIMEMultipart。这个对象与 e-mail.message\_from\_string 函数是创建接收到的消息的 Python 表示最通用的方式。

前面的示例中从头创建了 Message 对象,其目的是展示如何发送邮件消息,可以像处理那些 Message 对象一样处理上面提到的 Message 对象。Python 使用相同的类代表进入的和发出的消息。

### 试一试

### 打印邮箱摘要

如果在 E-mail 服务器上拥有一个 UNIX 账户,就可以为邮筒文件运行邮箱主题列表,得到一系列主题。如果在 E-mail 服务器上没有 UNIX 账户,或者如果使用了基于 Web 的邮件服务,就不能够以该种方式得到邮件:

```
$ python MailboxSubjectLister.py /var/spool/mail/leonardr
4 message(s) in mailbox "/var/spool/mail/leonardr":
DON'T DELETE THIS MESSAGE -- FOLDER INTERNAL DATA
This is a test message #1
This is a test message #2
This is a test message #3
```

第一个消息不是一个真正的消息,使用邮件客户端读取邮筒文件时有时会创建这种假消息。如果应用程序工作于邮筒文件之上,并且有时可以通过其他方式访问这些文件,那么就需要能够识别和处理这类消息。

### 16.3.2 使用 poplib 从 POP3 服务器获取邮件

解析本地邮箱不需要访问网络，因为您是在邮箱所在的计算机上运行脚本。没有必要包含网络协议，仅需包含文件格式即可(UNIX 邮箱的格式，主要从 RFC 2822 派生得到)。

然而，大多数人在邮件服务器上没有 UNIX shell 账户(或者即使有账户，他们也希望在自己的计算机上而不是在服务器上读邮件)。您需要通过网络从邮件服务器获取文件，这意味着必须使用协议。实现该功能的协议主要有两个。第一个协议是曾经很通用，但现在已经不太流行的 POP3 协议，它是 Post Office Protocol(邮局协议)的第三个版本。

POP3 定义于 RFC 1939 中，但是与大多数流行的 Internet 协议一样，不必过多关注它的细节，因为 Python 包含了一个模块，它用 Python 接口包装了该协议。

下面是 POP3 SubjectLister，一个基于 POP3 的邮箱解析脚本。这个脚本打印出服务器上每条消息的主题行：

```
#!/usr/bin/python
from poplib import POP3
import e-mail
class SubjectLister(Pop3):
    """Connect to a POP3 mailbox and list the subject of every message
    in the mailbox."""
    def __init__(self, server, username, password):
        "Connect to the POP3 server."
        Pop3.__init__(self, server, 110)
        #Uncomment this line to see the details of the POP3 protocol.
        #self.set_debuglevel(2)
        self.user(username)
        response = self.pass_(password)
        if response[:3] != '+OK':
            #There was a problem connecting to the server.
            raise Exception(response)
    def summarize(self):
        "Retrieve each message, parse it, and print the subject."
        numMessages = self.stat()[0]
        print('%d message(s) in this mailbox.' % numMessages)
        parser = e-mail.Parser.Parser()
        for messageNum in range(1, numMessages+1):
            messageString = '\n'.join(self.top(messageNum, 0)[1])
            message = parser.parsestr(messageString)
            #message = parser.parsestr(messageString, True)
            print('', message['Subject'])
```

当数据已经传递到网络的这一端时，使用以上脚本和使用基于 UnixMailbox 类的脚本处理邮件在方法上已经不存在本质区别。与 UnixMailbox 脚本一样，您使用 e-mail 模块将每条消息解析为 Python 中的数据结构(尽管此处使用了定义于 e-mail.Parser 模块中的 Parser 类，而不是便捷函数 message\_from\_file)。

使用 POP3 实现这一目的的缺点是 POP3.retr 方法有副作用。当对邮件服务器上的消息调用 retr 方法时，服务器将该消息标记为已读。如果使用邮件客户端或者类似于 fetchmail

的程序从 POP3 服务器检索新邮件，那么运行该脚本可能会引起困惑。消息依然位于服务器之上，但是客户端也许并不下载该消息，因为它认为该消息已经被阅读过。

POP3 还定义了 `top` 命令，它不标记已经读过的消息，而只检索消息的头部。`top` 和 `retr` 方法是实现这个脚本的目的的理想方法，利用这种方法将节省带宽(不必检索整条消息，只检索主题)，而且脚本对使用相同 POP3 邮箱的程序没有影响。

遗憾的是，并不是所有的 POP3 服务器都正确地实现了 `top` 命令。如果能够正确实现了它将会非常有用，这里有一个 `SubjectLister` 类的子类，它使用 `top` 命令检索消息头部而不是检索整条消息。如果知道自己的服务器可以正确地支持 `top` 命令，下面是一个更好的实现：

```
class TopBasedSubjectLister(SubjectLister):

    def summarize(self):
        """Retrieve the first part of the message and find the 'Subject:'
        header."""
        numMessages = self.stat()[0]
        print('%d message(s) in this mailbox.' % numMessages)
        for messageNum in range(1, numMessages+1):
            #Just get the headers of each message. Scan the headers
            #looking for the subject.
            for header in self.top(messageNum, 0)[1]:
                if header.find('Subject:') == 0:
                    print(header[len('Subject:'):])
                    break
```

`SubjectLister` 和 `TopBasedSubjectLister` 将得到相同的输出，但是您将发现 `TopBasedSubjectLister` 运行地更快(假设 POP3 服务器正确地实现了 `top` 命令)。

最后，您将为基于 POP3 的 `SubjectLister` 类创建一个简单的命令行接口，就像在文件 `MailboxSubjectLister.py` 中所做的那样。然而这一次，您需要在命令行提供一个 POP3 服务器和凭据，而不是文件在磁盘上的路径：

```
if __name__ == '__main__':
    import sys
    if len(sys.argv) < 4:
        print('Usage: %s [POP3 hostname] [POP3 user] [POP3 password]' %
              sys.argv[0])
        sys.exit(0)
    lister = TopBasedSubjectLister(sys.argv[1], sys.argv[2], sys.argv[3])
    lister.summarize()
```

## 试一试

## 打印 POP3 邮箱的摘要

用 POP 服务器的凭据运行 `POP3SubjectLister.py`，将得到如下的主题列表：

```
$ python POP3SubjectLister.py pop.example.com [username] [password]
```

```
3 message(s) in this mailbox.  
This is a test message #1  
This is a test message #2  
This is a test message #3
```

当访问 POP3 服务器时,不会得到在前面的示例中解析一个原始的 UNIX mailbox 文件时得到的假消息。邮件服务器知道那条消息不是真正的消息,而 UNIX 邮箱则将它当作一个真正的消息处理。

### 示例说明

SubjectLister 对象(或者它的 TopBasedSubjectLister 子类)连接到 POP3 服务器,并且发送“stat”命令获得邮箱中的消息数目。对 stat 的调用返回一个元组,该元组包含邮箱中的消息数目以及邮箱以字节为单位计算的整体大小。随后该 lister 迭代相应的次数,检索每条消息(或者仅仅是每个消息的头部)。

如果正在使用 SubjectLister,消息用 e-mail 模块的 Parser 实用程序类解析,并且 Subject 头是从得到的 Message 或者 MIMEMultipart 对象中检索出来。如果正在使用的是 TopBasedSubjectLister,它不做任何解析:邮件的头部以列表形式从服务器检索而来,并且将被扫描以寻找 Subject 头。

### 16.3.3 使用 imaplib 从 IMAP 服务器获取邮件

另一个访问远程服务器上的邮件箱的协议是 IMAP,它是 Internet Message Access Protocol(Internet 消息访问协议)的缩写。最新版本的 IMAP 定义于 RFC 3501,它拥有比 POP3 更多的特性,逐渐变得比 POP3 更加流行。

POP3 与 IMAP 的主要区别在于,POP3 更像一个邮箱:它在您收取邮件之前暂时保存邮件。IMAP 则将邮件永久地存储于服务器之上。这样做的好处之一是,可以在服务器上创建文件夹,将邮件分类到不同的文件夹中,并且搜索它们。这些是终端用户邮件客户端通常具有的比较复杂的特性。利用 IMAP,邮件客户端仅需要展示 IMAP 的这些特性,而不必自己实现它们。

在服务器上保存邮件使得在换用不同计算机时保持相同的邮件设置变得更容易。当然,仍旧可以向自己的计算机下载邮件,并从服务器中删除它们,如同利用 POP3 协议所做的那样。

这里有一个 IMAPSubjectLister.py,它是您刚才写过两遍的脚本的 IMAP 版本,打印出服务器上的所有邮件的主题行。由于 IMAP 比 POP3 拥有更多特性,因此这个脚本只用到了其中较少的特性。然而,即使实现相同的功能,它也比该脚本的 POP3 版本有了很大改进。IMAP 通过只检索邮件主题而节省带宽:每条消息一个主题。即使当 POP3 的 top 命令被正确地实现,也不能比将所有头部作为一组获取更好。

那么困难是什么呢?就像 imaplib 模块所陈述的那样,“为了使用这个模块,必须阅读 IMAP4 协议的 RFC”。模块 imaplib 为每个 IMAP 命令提供了一个函数,但是它并没有在您经常创建的 Python 数据结构和 IMAP 协议使用的格式化字符串之间做太多转换。需要在手头保留一个 RFC 3501 的副本,否则将不知道要向 imaplib 的方法中传递什么参数。

例如，要向 `imaplib` 中传递一个消息 ID 的列表，需要传递一个像 1、2、3 一样的字符串，而不是 Python 列表(1,2,3)。为了确保只从服务器中提取出主题，`IMAPSubjectLister.py` 将字符串“(BODY[HEADER.FIELDS(SUBJECT)])”作为参数传递给 `imaplib` 方法。该命令的结果是一个格式化字符串的嵌套列表，其中只有一些是对脚本真正有用的。

这还并不完全是人们期望 Python 所拥有的那种直观性。`imaplib` 当然很有用，但是它并不能很好地向程序员隐藏 IMAP 的细节：

```
#!/usr/bin/python
from imaplib import IMAP4
class SubjectLister(IMAP4):
    """Connect to an IMAP4 mailbox and list the subject of every message
    in the mailbox."""
    def __init__(self, server, username, password):
        "Connect to the IMAP server."
        IMAP4.__init__(self, server)
        #Uncomment this line to see the details of the IMAP4 protocol.
        #self.debug = 4
        self.login(username, password)
    def summarize(self, mailbox='Inbox'):
        "Retrieve the subject of each message in the given mailbox."
        #The SELECT command makes the given mailbox the 'current' one,
        #and returns the number of messages in that mailbox. Each message
        #is accessible via its message number. If there are 10 messages
        #in the mailbox, the messages are numbered from 1 to 10.
        numberOfMessages = int(self._result(self.select(mailbox)))

        print('%s message(s) in mailbox "%s":' % (numberOfMessages, mailbox))
        #The FETCH command takes a comma-separated list of message
        #numbers, and a string designating what parts of the
        #message you want. In this case, we want only the
        #'Subject' header of the message, so we'll use an argument
        #string of '(BODY[HEADER.FIELDS (SUBJECT)])'.
        #
        #See section 6.4.5 of RFC3501 for more information on the
        #format of the string used to designate which part of the
        #message you want. To get the entire message, in a form
        #acceptable to the e-mail parser, ask for '(RFC822)'.
        subjects = self._result(self.fetch('1:%d' % numberOfMessages,
                                           '(BODY[HEADER.FIELDS (SUBJECT)])'))
        for subject in subjects:
            if hasattr(subject, '__iter__'):
                subject = subject[1]
                print('', subject[:subject.find('\n')])
    def _result(self, result):
        """Every method of imaplib returns a list containing a status
        code and a set of the actual result data. This convenience
        method throws an exception if the status code is other than
        "OK", and returns the result data if everything went all
```

```

        right."""
        status, result = result
        if status != 'OK':
            raise status (result)
        if len(result) == 1:
            result = result[0]
        return result
if __name__ == '__main__':
    import sys
    if len(sys.argv) < 4:
        print('Usage: %s [IMAP hostname] [IMAP user] [IMAP password]' % sys.
argv[0])
        sys.exit(0)
    lister = SubjectLister(sys.argv[1], sys.argv[2], sys.argv[3])
    lister.summarize()

```

**试一试****打印 IMAP 邮箱的摘要**

使用 IMAP 凭据(就像用 POP3SubjectLister 一样)执行 IMAPSubjectLister.py, 将得到与本章前面所示的类似的摘要:

```

$ python IMAPSubjectLister.py imap.example.com [username] [password]
3 message(s) in mailbox "Inbox":
  This is a test message #1
  This is a test message #2
  This is a test message #3

```

**示例说明**

就像在 POP3 示例中一样, 第一件要完成的事是连接服务器。POP3 服务器仅为每个用户提供一个邮箱, 但是 IMAP 允许每个用户有任意多个邮箱, 因此下一步便是选择一个邮箱。

默认邮箱称作“收件箱”, 选择一个邮箱会得到那个邮箱中的消息数目(一些 POP3 服务器在连接到服务器时返回邮箱中的消息数目, 但并非全部 POP3 服务器都是如此)。

与 POP3 不同, IMAP 允许一次检索多条消息。它还允许灵活地定义到底需要消息中的哪个部分。基于 IMAP 的 SubjectLister 仅调用 IMAP 一次, 来检索邮箱中每条消息的主题(仅仅是主题)。随后在列表上进行迭代, 并且打印出每个主题。真正的诀窍在于知道向 imaplib 传递什么参数以及如何解释结果。

**IMAP 的唯一消息 ID**

除了对 imaplib 用户友好性的抱怨之外, 如果假设消息数目不会随着时间变化, 也许在写 IMAP 脚本时会遇到问题。如果另外一个 IMAP 客户端从该脚本正在处理的邮箱中删除消息(假设正在运行邮件客户端, 并且在该脚本运行时, 您用它删除一些垃圾邮件), 消息数目从此处起将会不同步地发生变化。

在选择了邮箱后, 基于 IMAP 的 SubjectLister 类立即通过在一个操作中得到每条消息的主题来最小化这种风险:

```
self.fetch('1:%d' % numberOfMessages, '(BODY[HEADER.FIELDS (SUBJECT)])')
```

如果收件箱中有 10 条消息，fetch 的第一个参数将是 1:10。这是邮箱的一个片段，与一个 Python 列表片段类似，它返回所有消息：从第一条消息到第十条消息(IMAP 和 POP3 消息是从 1 开始计数的)。

一连接到服务器就尽可能快地得到所需数据将最小化向服务器传递一个不再有效的消息数目的风险，但是不能总是这样操作。您也许写了一个脚本来删除邮箱中的消息，或者将这些消息备份到第二个邮箱中。在更换了邮箱之后，您也许不能再继续信任最初得到的消息数目。

### 试一试

### 通过唯一的 ID 获取消息

为了帮助您避免该问题，IMAP 为自己控制的每个消息保持一个唯一的 ID(UID)。可以从服务器获取唯一 ID，并且在后面的调用中利用 imaplib 的 uid 方法使用这些 ID。但是这将使您更接近 IMAP 协议的细节。IMAP4 类为每个 IMAP 命令定义了一个单独的方法(例如，IMAP4.fetch、IMAP4.search 等)，但是当处理 ID 时，不能使用这些方法。可以仅用 IMAP4.uid 方法，而且必须将希望的 IMAP 命令作为要传递的第一个参数。例如，不能调用 IMAP4.fetch([arguments])，而必须调用 IMAP4.uid('FETCH',[arguments])。

```
>>> import imaplib
>>> import e-mail
>>> imap = imaplib.IMAP4('imap.example.com')
>>> imap.login('[username]', '[password]')
('OK', ['Logged in.'])
>>> imap.select('Inbox')[1][0]
'3'
>>>
>>> #Get the unique IDs for the messages in this folder.
... uids = imap.uid('SEARCH', 'ALL')
>>> print(uids)
('OK', ['49532 49541 49563'])
>>>
>>> #Get the first message.
... uids = uids[1][0].split(' ')
>>> messageText = imap.uid('FETCH', uids[0], "(RFC822)") [1][0][1]
>>> message = e-mail.message_from_string(messageText)
>>> print(message['Subject'])
This is a test message #1
```

### 示例说明

利用唯一 ID 提取消息需要 4 个 IMAP 命令。第一个和第二个是，客户端必须连接到服务器并且选择一个邮箱，如前面的 IMAP 示例所示。第三，客户端需要运行 SEARCH 命令，返回一个消息 UID 列表。最后，客户端可以向 FETCH 命令传递一个 UID，并且得到

实际的消息。

最后两步都贯穿了 IMAP4.uid 方法。如果没有使用 UID，它们可以分别使用 search 和 fetch 方法。

使用 imaplib 模块与 IMAP 服务器交互是件痛苦的事情，但是比直接与服务器通信要方便一些。

**注意：**

POP3 服务器也支持 UID，不过多个客户端同时访问一个 POP3 邮箱的情况并不常见。POP3 对象的 uidl 方法将检索它的邮箱中的消息的 UID。之后，可以向 POP3 对象的其他可以接收消息 ID 的方法传递 UID，例如 retr 和 top。IMAP 的 UID 是数值，而 POP3 的是“消息摘要”：从每个消息的内容中推导出的十六进制签名。

### 16.3.4 安全的 POP3 和 IMAP

本章前面介绍的 POP3 和 IMAP 示例都存在安全问题：它们在网络上发送未经加密的用户名和密码。这也是 POP 和 IMAP 经常运行在安全套接字协议层(Secure Socket Layer, SSL)之上的原因。这是一个通用的加密层，也用来对万维网上的 HTTP 连接进行加密。支持 SSL 的 POP 和 IMAP 服务器运行的端口与不支持 SSL 的服务器运行的端口不同。运行于 SSL 之上的 POP 服务器的标准端口号是 995，而不是 23，而 SSL 上的 IMAP 端口号为 993，而不是 143。

如果 POP3 和 IMAP 服务器支持 SSL，可以得到与服务器的加密连接，方法是使用 POP3\_SSL 或者 IMAP4\_SSL 类交换 POP3 或者 IMAP4 类。不同的 SSL 类都位于同一个模块中，它们与非安全的对应类具有相同的接口，不同之处在于它们在通过网络发送数据之前要对数据进行加密。

### 16.3.5 Webmail 应用程序不是 E-mail 应用程序

如果使用的是 Webmail 系统，例如 Yahoo! Mail 或者 Gmail，从技术上说您并没有使用邮件应用程序：您正在使用 Web 应用程序，只是它在另一端恰巧拥有邮件应用程序而已。本节中的脚本并不能帮助您从这些服务中获取邮件，或者通过这些服务发送邮件，因为它们实现的是 HTTP 协议，而不是 E-mail 协议(然而，Yahoo! 邮件提供了收费的 POP3 访问)。您应当参考第 20 章中有关 Web 应用程序工作方式的内容。

**注意：**

libgmail 工程旨在创建一个 Gmail 的 Python 接口，它可以把 Gmail 作为 SMTP、POP3 或者 IMAP 服务器处理。libgmail 的主页位于 <http://libgmail.sourceforge.net/>。

## 16.4 套接字编程

到目前为止，您已经了解与一种 Internet 应用程序(email)相关的协议和文件格式。E-mail

当然是一种通用而且有用的应用程序，但是 E-mail 相关的协议只是在 Internet 协议上实现的众多协议中的极少数几个。Python 通过提供包装库使得使用 E-mail 相关的协议(还有一些本章中未涉及的其他协议)变得容易，但是 Python 并没有为每个网络协议提供一个库。对于您为自己的 Internet 应用程序创建的新协议，它肯定没有对应的库。

为了编写自己的协议，或者实现与 `imaplib` 或者 `poplib` 相似的 Python 库，您需要向下走一层，并且学习基于 IP 协议的编程接口的工作方式。幸好编写这样的代码并不困难：利用 `smtplib`、`poplib` 和其他一些模块可以较为容易地实现它。秘诀在于套接字库，它使得读写网络接口就如同读写磁盘上的文件一样。

### 16.4.1 套接字简介

在前面的许多示例中，您通过特定计算机的特定端口连接到服务器上(例如，可以通过本地主机的 25 号端口连接到本地 SMTP 服务器)。当告诉 `imaplib` 或者 `smtplib` 要连接到特定主机的端口时，Python 在后台打开一个到那台主机的端口的连接。一旦创建了连接，服务器会打开一个到您的计算机的逆向连接。Python 的“套接字”对象在一个单独的接口下隐藏了出去和进入的连接。套接字就像一个可以同时读写的文件。

为了实现基于 TCP/IP 协议的客户端，需要打开到一台合适的服务器的套接字。您不仅向套接字中写入数据以将它发送给服务器，还要从服务器发送给您的套接字中读取数据。为了实现服务器端，所做的操作与上述操作相反：将一个套接字绑定到主机的一个端口，并等待客户端连接它。一旦服务器发现有一个连接上的客户端，它便从套接字中读取客户端发送过来的数据，并且向套接字中写入数据并传送回客户端。

在网络上实际发送一个单字节要耗费巨大的劳动量，而使用 TCP/IP 和套接字库，就可以避免这些繁琐的劳动。您不必想办法确定如何将数据传递到目的地，因为 TCP/IP 会处理这项工作。也不需要担心如何将数据转换为 TCP/IP 包，因为套接字库会处理这项工作。

#### 注意：

就像 E-mail 和 Web 是使用 Internet 的关键应用程序一样，套接字也可以被认为是采用 TCP/IP 协议的关键应用程序。套接字在 BSD UNIX 的早期版本中引入，从那时起几乎所有的 TCP/IP 实现都使用套接字来描述如何编写网络程序。套接字使得 TCP/IP 协议变得易于使用(至少，比其他可选的协议更易于使用)，这也是 TCP/IP 流行起来的主要原因。

作为第一个套接字示例，这里有一个超级简单的套接字服务器 `SuperSimpleSocketServer.py`：

```
#!/usr/bin/python
import socket
import sys
if len(sys.argv) < 3:
    print('Usage: %s [hostname] [port number]' % sys.argv[0])
    sys.exit(1)
hostname = sys.argv[1]
port = int(sys.argv[2])
#Set up a standard Internet socket. The setsockopt call lets this
#server use the given port even if it was recently used by another
```

## 第III部分 开始使用 Python

```

#server (for instance, an earlier incarnation of
#SuperSimpleSocketServer).
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
#Bind the socket to a port, and bid it listen for connections:
sock.bind((hostname, port))
sock.listen(1)
print("Waiting for a request.")
#Handle a single request.
request, clientAddress = sock.accept()
print("Received request from", clientAddress)
request.send(bytes('--SuperSimpleSocketServer 3000--\n', 'utf-8'))
request.send(bytes('Go away!\n', 'utf-8'))
request.shutdown(2) #Stop the client from reading or writing anything.
print("Have handled request, stopping server.")
sock.close()

```

这个服务器只能够接受单个请求。只要有客户端连接到它绑定的端口，它便会要求该客户端离开、关闭连接、停止服务请求并退出。

## 试一试

## 使用 Telnet 连接到 SuperSimpleSocketServer

`telnet` 程序是 TCP/IP 应用程序的一个简单的客户端。可以通过主机名称和端口号调用它，它会连接到指定的端口，随后您就可以自主操作。任何输入的数据都会通过套接字发送到服务器，服务器通过套接字发送的任何数据都会显示在终端上。Windows、Mac OS X 和 UNIX 等系统都以命令行程程序的形式安装了 Telnet，因此不必担心得不到它。

由于示例套接字服务器不做任何操作，没有必要再为它编写一个自定义客户端。为了测试它，只需启动服务器：

```

$ python SuperSimpleSocketServer.py localhost 2000
Waiting for a request.

```

之后，在单独的终端里通过 `telnet` 登录到服务器：

```

$ telnet localhost 2000
Trying 127.0.0.1...
Connected to rubberfish.
Escape character is '^]'.
--SuperSimpleSocketServer 3000--
Go away!
Connection closed by foreign host.

```

返回到运行服务器的终端，您会看到类似于下面的输出：

```

Received request from ('127.0.0.1', 32958)
Have handled request, stopping server.

```

### 示例说明

当启动 SuperSimpleSocketServer 时，您将进程绑定到“localhost”的端口 2000。当该脚本调用 socket.accept 时，它停止运行并且开始阻塞套接字输入，等待某人连接到服务器。

当 telnet 命令打开一个到 SuperSimpleSocketServer 的 TCP/IP 连接时，socket.accept 方法从等待状态返回。终于，某人连接到这个服务器。socket.accept 返回服务器与客户端通信的必备工具：套接字对象与描述客户端网络地址的元组。服务器通过套接字向客户端发送数据。多余的套接字连接将不再被接受。

这里唯一不易理解的地方是客户端地址元组：('127.0.0.1', 32958)。您已经看到过地址 127.0.0.1，它是指向本机的特殊 IP 地址，与主机名 localhost 等价。服务器上来自于 127.0.0.1 的连接的含义是客户端位于运行服务器的计算机上。如果从其他计算机上通过 telnet 登录到服务器，将会显示那台计算机的 IP 地址。

端口号 32958 是那个客户端临时的端口号。看上去是一个双向的“套接字”对象，实际上包含两个单向连接：一个是从客户端到服务器的连接，另一个是从服务器到客户端的连接。服务器在启动时绑定到 localhost 上的端口 2000，它也是所有客户端数据的目的地址（并不是说该客户端有机会发送数据）。服务器发送的数据也必须有目的主机名称和端口号，但并不是事先定义的一个地址。服务器管理员通常选定服务器端口，而临时端口则是由客户端的操作系统选择的。再次运行这个练习，您将看到每个单独的 TCP/IP 连接都被给定一个不同的临时端口号。

### 16.4.2 绑定到外部主机名

如果按照前面的建议尝试从其他计算机通过 telnet 登录到 SuperSimpleSocketServer，您也许会注意到不能连接到服务器。如果出现这种情况，原因可能是您启动服务器时将它绑定到 localhost 上。“localhost”是特殊的内部主机名，其他计算机无法访问它。毕竟，从其他人的角度看，“localhost”指他们自己的计算机，而不是您的。

这非常有用，因为它使得您可以测试本章(以及第 20 章)的服务器，而不必承担将您的计算机暴露给 Internet 上的大量连接的风险(当然，如果在一个多用户计算机上运行这些服务器，您不得不担心同一台计算机上的其他用户，因此尽量在自己的系统上运行这些服务器)。然而，当需要运行一个真正的服务器时，真正需要的是外部连接，此时需要将服务器绑定到一个外部主机名称。

如果可以通过 SSH 远程登录到计算机，或者已经运行了一个 Web 服务器，又或者曾经从其他计算机引用过自己的计算机，那么您已经知道自己的计算机的外部主机名。另一方面，如果拥有一个拨号或者宽带连接，无论何时连接到自己的 ISP(Internet 服务提供商)，都可能被分配了一个主机名和一个 IP 地址。找到自己的计算机的 IP 地址，并在它上面执行 DNS 查找，以找到自己的计算机的外部主机名。如果其他方法都失败，可以直接将服务器绑定到您的外部 IP 地址(不是 127.0.0.1，因为这样会产生与绑定到“localhost”相同的问题)。

#### 注意：

如果将服务器绑定到一个外部主机名称，但是依然不能够从外部连接它，原因可能是

存在防火墙。修复该问题超出了本书的讨论范围。您应该咨询当地的计算机专家，以解决该问题。

### 16.4.3 镜像服务器

下面介绍的服务器要复杂一些(尽管并不一定更加有用)，它将显示 Python 如何使您可以像处理文件那样处理套接字连接。这个服务器从套接字接收文本，就像脚本可以从标准输入接收文本一样。它反转这些文本，并且通过套接字将反转后的版本写回，如同脚本可以向标准输出写文本那样。当它接收到一行空文本时，就会终止连接：

```
#!/usr/bin/python
import socket

class MirrorServer:
    """Receives text on a line-by-line basis and sends back a reversed
    version of the same text."""

    def __init__(self, port):
        "Binds the server to the given port."
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.socket.bind(port)
        #Queue up to five requests before turning clients away.
        self.socket.listen(5)

    def run(self):
        "Handles incoming requests forever."
        while True:
            request, client_address = self.socket.accept()
            #Turn the incoming and outgoing connections into files.
            input = request.makefile('rb', 0)
            output = request.makefile('wb', 0)
            l = True
            try:
                while l:
                    l = input.readline().strip()
                    if l:
                        output.write(l[::-1] + bytes('\r\n', 'utf-8'))
                    else:
                        #A blank line indicates a desire to terminate the
                        #connection.
                        request.shutdown(2) #Shut down both reads and writes.
            except socket.error:
                #Most likely the client disconnected.
                pass
if __name__ == '__main__':
    import sys
    if len(sys.argv) < 3:
```

```

    print('Usage: %s [hostname] [port number]' % sys.argv[0])
    sys.exit(1)
    hostname = sys.argv[1]
    port = int(sys.argv[2])
    MirrorServer((hostname, port)).run()

```

**试一试****使用镜像服务器映像文本**

像使用 `SuperSimpleSocketServer` 那样，不用编写一个特殊的客户端就能使用镜像服务器。可以仅通过 `telnet` 登录到 `MirrorServer`，并输入一些文本。如果输入空行，服务器将断开连接。在一个终端中启动服务器：

```
$ python MirrorServer.py localhost 2000
```

在另外一个终端中，通过 `telnet` 作为客户端登录到服务器：

```

$ telnet localhost 2000
Trying 127.0.0.1...
Connected to rubberfish.
Escape character is '^]'.
Hello.
.olleH
Mirror this text!
!txet siht rorriM

Connection closed by foreign host.
$

```

**16.4.4 镜像客户端**

尽管您已经看到镜像服务器通过 `telnet` 完全可用，但并不是每个人都喜欢使用 `telnet`。您需要一个具有许多辅助功能的镜像服务器，使得即使是网络的初学者，也可以感受到输入文本并看到它在反转之后显示出来的兴奋感。这里有一个简单的客户端，它接受指定的服务器地址和要反转的文本作为命令行参数。客户端连接到服务器，发送数据，并打印反转后的文本：

```

#!/usr/bin/python
import socket

class MirrorClient:
    "A client for the mirror server."

    def __init__(self, server, port):
        "Connect to the given mirror server."
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.socket.connect((server, port))

```

```

def mirror(self, s):
    "Sends the given string to the server, and prints the response."
    if s[-1] != '\n':
        s += '\r\n'
    self.socket.send(bytes(s, 'utf-8'))

    #Read server response in chunks until we get a newline; that
    #indicates the end of the response.
    buf = []
    input = ''
    while not '\n' in input:
        try:
            input = self.socket.recv(1024)
            buf.append(input)
        except socket.error:
            break
    return ''.join(buf)[:-1]

def close(self):
    self.socket.send(bytes('\r\n', 'utf-8'))
    #We don't want to mirror anything else.
    self.socket.close()

if __name__ == '__main__':
    import sys
    if len(sys.argv) < 4:
        print('Usage: %s [host] [port] [text to be mirrored]' % sys.argv[0])
        sys.exit(1)
    hostname = sys.argv[1]
    port = int(sys.argv[2])
    toMirror = sys.argv[3]

    m = MirrorClient(hostname, port)
    print (m.mirror(toMirror))
    m.close()

```

镜像服务器将套接字连接转换为一对文件，但是该客户端直接从套接字读写。这么做并没有令人信服的理由，我只是感觉本章至少应该包含一个使用底层套接字 API 的示例。注意，服务器的响应以块状读取，并且在每一个块中扫描换行符，它指出响应的结束。如果该示例为进入的套接字连接创建一个文件，那么这些代码只需要调用 `input.readline`。

知道响应何时结束非常重要，因为调用 `socket.recv`(或者 `input.readline`)将阻塞您的进程，直到服务器发送更多的数据。如果服务器继续等待客户端输入更多数据，进程将永远被阻塞。

### 16.4.5 套接字服务器

套接字非常有用，但是 Python 并不满足于提供同样的基于 C 语言的套接字接口，在大多数操作系统以及大多数语言中都可以得到这些接口。Python 更进一步，提供了

socketserver, 它是一个包含大量类的模块, 允许用较少的代码编写复杂的基于套接字的服务器。

创建一个 socketserver 的主要工作在于定义一个请求处理程序类。这是 socketserver 模块的 BaseRequestHandler 类的一个子类, 请求处理程序对象的目的是, 只要客户端连接到服务器, 它便处理该客户端的请求。该功能通过处理程序的 handle 方法实现。通过重写 setup 和 finish, 处理程序还可以定义每个请求的创建和销毁代码。

BaseRequestHandler 的子类的方法可以访问如下三个成员:

- request: 套接字对象, 代表客户端请求: 与 MirrorServer 示例中 socket.accept 方法返回的对象相同。
- client\_address: 一个二元组, 它包含主机名和服务器输出的数据将要发送到的端口。它是 MirrorServer 示例中从 socket.accept 得到的另外一个对象。
- server: 对创建请求处理程序对象的 socketserver 的引用。

通过创建 streamRequestHandler 而非 BaseRequestHandler 的子类, 可以访问类似于文件的对象, 它允许您从套接字连接读取和发送数据。BaseRequestHandler 允许访问另外两个成员:

- rfile: 对应于通过套接字进入的数据的文件(如果正在向服务器写数据, 那么它来自于客户端, 如果正在向客户端发送数据, 它来自于服务器)。等价于调用 request.makefile('rb')。
- wfile: 对应于通过套接字发送的数据的文件(如果正在写服务器, 那么是发送到客户端的数据, 如果正在客户端写数据, 那么是发送到服务器的数据)。等价于调用 request.makefile('wb')。

通过将 MirrorServer 重写为一个 socketserver(具体来说, 一个 TCPServer), 可以不必编写创建和删除套接字的大量代码, 只要关注于反转文本这一任务。下面是 MirrorSocketServer.py 的代码:

```
#!/usr/bin/python
import socketserver

class RequestHandler(socketserver.StreamRequestHandler):
    """Handles one request to mirror some text."""

    def handle(self):
        """Read from StreamRequestHandler's provided rfile member,
        which contains the input from the client. Mirror the text
        and write it to the wfile member, which contains the output
        to be sent to the client."""
        l = True
        while l:
            l = self.rfile.readline().strip()
            if l:
                self.wfile.write(l[::-1] + bytes('\n', 'utf-8'))

if __name__ == '__main__':
```

```

import sys
if len(sys.argv) < 3:
    print('Usage: %s [hostname] [port number]' % sys.argv[0])
    sys.exit(1)
hostname = sys.argv[1]
port = int(sys.argv[2])

socketserver.TCPServer((hostname, port), RequestHandler).serve_forever()

```

几乎所有套接字特定的代码都不见了。不论何时有人连接到服务器，TCPServer 类都将创建一个拥有合适成员的新的 RequestHandler，并且调用其 handle 方法处理请求。

之前编写的 MirrorClient 同样也可以很好地与这个服务器一起工作，因为在网络上这两个服务器在输入相同的情况下都得到相同的输出。同样的原则也适用于改变模块中的函数以避免重复代码，但却保留相同的接口的情况。

### 16.4.6 多线程服务器

镜像服务器的这两种实现的一个问题是，每次只有一个客户端可以连接到正在运行的服务器。如果打开两个 telnet 会话，登录到一个运行的服务器，第二个会话将不能连接，除非关闭第一个连接。如果真实的服务器以这种方式运行，那么它将不能完成任何操作。所以大多数真正的服务器都派生出线程和子进程来处理多个连接。

SocketServer 模块定义了可以一次处理多个连接的两个有用的类：ThreadingMixIn 和 ForkingMixIn。ThreadingMixIn 的子类 SocketServer 将自动派生一个新线程来处理每个进入的请求。ForkingMixIn 的子类会自动生成一个新的子进程处理每个进入的请求。ThreadingMixIn 更好一些，因为线程比子进程更有效率并更易于移植。创建子线程与父线程通信的代码也比创建子进程与父进程通信的代码更容易。

**提示：**

参考第 9 章中对线程和子进程的介绍。

下面给出了 MultithreadedMirrorServer.py，它是 MirrorSocketServer 的一个多线程版本。注意它使用与 MirrorSocketServer.py 中完全相同的 RequestHandler 的定义。不同之处在于，此处的 MultithreadedMirrorServer 没有运行 TCPServer，而是运行了 ThreadingTCPServer，它是一个从 ThreadingMixIn 和 TCPServer 继承的标准类。

```

#!/usr/bin/python
import socketserver

class RequestHandler(socketserver.StreamRequestHandler):
    """Handles one request to mirror some text."""

    def handle(self):
        """Read from StreamRequestHandler's provided rfile member,
        which contains the input from the client. Mirror the text
        and write it to the wfile member, which contains the output

```

```

        to be sent to the client."""
        l = True
        while l:
            l = self.rfile.readline().strip()
            if l:
                self.wfile.write(l[:-1] + bytes('\n', 'utf-8'))

if __name__ == '__main__':
    import sys
    if len(sys.argv) < 3:
        print('Usage: %s [hostname] [port number]' % sys.argv[0])
        sys.exit(1)
    hostname = sys.argv[1]
    port = int(sys.argv[2])
    server = socketserver.ThreadingTCPServer((hostname, port),
        RequestHandler)
    server.serve_forever()

```

运行这个服务器时,可以并行运行大量的 telnet 会话和 MirrorClient 会话。ThreadingMixIn 隐藏了生成线程的细节,如同 TCPServer 隐藏套接字的细节一样。所有这些辅助类的目的都是让您关注于在网络上发送和接收数据。

### 16.4.7 Python 聊天服务器

对于镜像服务器,支持多个并行连接的能力很有用,但是它并不改变服务器的实际操作。每个客户端只与服务器交互,甚至都不间接地与其他客户端交互。这个模型非常流行,Web 服务器和邮件服务器等都使用这个模型。

还有另外一种类型的服务器,它允许客户端之间相互连接。许多应用程序并不关心服务器,而是关心其他哪些客户端连接到了服务器。最流行的这种类型的应用程序包括在线聊天室和在线游戏。本节将设计和构建一个简单的聊天服务器和客户端。

可能最原始的聊天室是(未联网的)UNIX 的 wall 命令,可以利用它向登录到 UNIX 系统的每个人广播一条消息。发明于 1988 年的 Internet 中继聊天(Internet Relay chat, IRC)在 RFC 1459 中有描述,是最流行的基于 TCP/IP 的聊天室软件。这里编写的聊天软件与 IRC 有一些相同的功能,尽管它与 IRC 不兼容。

### 16.4.8 设计 Python 聊天服务器

在 IRC 中,每个连接到服务器的客户端都必须提供一个昵称:一个标识出希望聊天的人的短字符串。服务器上的所有昵称都应该不同,这样用户才不会混淆。您的服务器也要遵守这个传统。

一个 IRC 服务器可以提供无限多个已命名的房间,并且每个用户可以加入任意多个房间。您的服务器将提供单个未命名的房间,所有连接的用户将在该房间中进行聊天。

在 IRC 客户端中输入一行文本,该消息将被广播给当前房间中的其他人,除非它以斜杠开头。以斜杠开头的一行文本将被服务器当作一条命令。您的服务器也如此工作。

IRC 实现了很多种类的服务器命令:例如,可以使用一个服务器命令改变自己的昵称,

加入另外一个房间，向其他用户发送一条私人消息，或者尝试向另外一个用户发送文件。

如果在 IRC 服务器上执行 `/nick leonardr` 命令，您可能正在试图将自己当前的昵称改为 leonardr。尝试可能成功，也可能不成功，这要取决于 IRC 服务器上是否已经存在 leonardr。

服务器将支持如下 3 条命令，它们来自于 IRC 并且经过简化：

- `/nick [nickname]`：如前所述，该命令尝试修改昵称。如果昵称有效，并且还没有被使用过，昵称将被改变，并且此更改将通告给同房间中的所有人。否则，您将得到一条私人错误消息。
- `/quit [farewell message]`：这条命令将用户从聊天服务器断开。如果指定了告别消息，它将广播给整个聊天室。
- `/names`：这条命令提取出聊天室中所有用户的名称，并以空格分隔的字符串的形式返回。

### 16.4.9 Python 聊天服务器协议

在确定了功能集合和设计方案之后，必须为自己的 Python 聊天服务器定义一个特定于应用程序的协议。该协议将与 SMTP、HTTP 和 IRC 协议类似，它运行于 TCP/IP 协议之上，为特定类型的应用程序提供结构。然而，它比这些协议中的任何一个都要简单得多。

该镜像服务器还定义了一个协议，尽管它很简单，不引人注目。镜像服务器协议包含如下所示的 3 个简单的规则：

- (1) 向服务器发送多行文本。
- (2) 每次发送一行新文本时，服务器将返回反转后的这行文本，并在最后放置一个换行符。
- (3) 发送空行终止连接。

Python 聊天服务器协议将比上面的协议略微复杂一些，但是根据协议设计标准，它仍然是一个非常简单的协议。下面的描述或多或少应该写进该协议的 RFC 中。如果您正在写一个 RFC，则需要包含更多的细节并提供该协议的正式定义。这里并不需要这么做，因为协议定义后面直接跟着 Python 实现。

**注意：**

当然，如果您为该协议编写了一个 RFC，它将不会被接受。IRC 协议已经有 RFC，它是比这个示例更加有用的协议。

#### 1. 假想协议

确定协议设计过程中可能遇到的问题的一个好方法是写一个示例会话，观察客户端和服务端都需要跟对方沟通哪些内容。这里有一个 Python 聊天服务器的示例会话。在下面的对话记录中，一个昵称为 jamesp 的用户连接到一个聊天室，一个昵称为 nrini 的阴影角色已经存在了。表 16-2 展示了 jamesp 可能发送给服务器的内容，服务器对他的回应，以及在 jamesp 输入后，服务器发送给其他客户端(nrini)的内容。

表 16-2

jamesp 到服务器	服务器到 jamesp	服务器到 nrini
	您是谁	
jamesp		
	您好, jamesp, 欢迎来到 Python 聊天室服务器	jamesp 加入了聊天
/names		
	nrini jamesp	
您好!		
	<jamesp> 您好!	<jamesp> 您好!
/nick nrini		
	这里已经有一个叫做 nrini 的用户	
/nick james		
	jamesp 现在叫做 james	jamesp 现在叫做 james
您好!		
	<james>您好!	<james>您好!
/quit 再见!		
		james 已离开: 再见!

2. 初始连接

客户端和服务端之间建立连接后，协议的第一个阶段是为客户端获取一个昵称。客户端如果没有昵称，将无法加入一个聊天室，因为这会让其他用户感到困惑。因此，服务器将询问每个新的客户端：“您是谁”？并期待客户端回应一个以换行符终止的昵称。如果发送的是无效的昵称或者昵称已经在聊天室中存在，服务器将发送一条错误消息并终止连接。否则，服务器将欢迎客户端来到聊天室，并且将某人已经加入聊天的消息广播给其他用户。

3. 聊天文本

当允许客户端加入聊天室时，他发送的任何一条文本都将广播给聊天室中的每个用户，除非文本是服务器命令。当广播一条聊天消息时，消息前面将加上发送者的昵称，并以尖括号括住昵称(例如，“<jamesp> Hello, all.”)。这就可以清楚地知道谁发送了什么消息，并且从视觉上区分系统消息和聊天消息。

4. 服务器命令

如果客户端发送一条服务器可以识别的命令，该命令将被执行，并且会有一条私人系统消息发送给此客户端。如果命令的执行改变了聊天室的状态(例如，一个用户改变了他的昵称或者退出)，所有用户都将收到一条聊天室的状态更改的系统消息(例如，“jamesp 现在叫做 james”)。一条无法识别的服务器命令将导致服务器给命令的发送者传递一条错误消息。

## 5. 通用原则

为了方便性和易读性，聊天协议将被设计为基于行的易读的格式。这使得聊天应用程序在没有特殊客户端的情况下仍然可用(尽管您将写一个特殊的客户端使聊天变得容易一些)。许多 TCP/IP 协议的工作方式都类似，但并不要求必须这么做。为了节省带宽或者在传输数据之前对它们进行加密，一些协议只发送二进制数据。

下面给出了 `PythonChatServer.py` 中的服务器代码。像 `MultithreadedMirrorServer` 一样，它的实际服务器类为 `ThreadingTCPServer`。它保留了一个从用户昵称指向 `wfile` 成员的持久映射。这使得服务器可以发送这些用户的数据。用户的输入以这种方式广播给聊天室中的所有人：

```
#!/usr/bin/python
import socketserver
import re
import socket

class ClientError(Exception):
    "An exception thrown because the client gave bad input to the server."
    pass

class PythonChatServer(socketserver.ThreadingTCPServer):
    "The server class."

    def __init__(self, server_address, RequestHandlerClass):
        """Set up an initially empty mapping between a user's nickname
        and the file-like object used to send data to that user."""
        SocketServer.ThreadingTCPServer.__init__(self, server_address,
                                                RequestHandlerClass)
        self.users = {}

class RequestHandler(SocketServer.StreamRequestHandler):
    """Handles the life cycle of a user's connection to the chat
    server: connecting, chatting, running server commands, and
    disconnecting."""

    NICKNAME = re.compile('^[A-Za-z0-9_-]+$') #Regex for a valid nickname

    def handle(self):
        """Handles a connection: gets the user's nickname, then
        processes input from the user until they quit or drop the
        connection."""
        self.nickname = None

        self.privateMessage('Who are you?')
        nickname = self._readline()
        done = False
        try:
            self.nickCommand(nickname)
```

```

        self.privateMessage('Hello %s, welcome to the Python Chat
Server.'\
                               % nickname)
        self.broadcast('%s has joined the chat.' % nickname, False)
    except ClientError (error):
        self.privateMessage(error.args[0])
        done = True
    except socket.error:
        done = True

    #Now they're logged in; let them chat.
    while not done:
        try:
            done = self.processInput()
        except ClientError (error):
            self.privateMessage(str(error))
        except socket.error (e):
            done = True

def finish(self):
    "Automatically called when handle() is done."
    if self.nickname:
        #The user successfully connected before disconnecting.
        #Broadcast that they're quitting to everyone else.
        message = '%s has quit.' % self.nickname
        if hasattr(self, 'partingWords'):
            message = '%s has quit: %s' % (self.nickname,
                                           self.partingWords)
        self.broadcast(message, False)

        #Remove the user from the list so we don't keep trying to
        #send them messages.
        if self.server.users.get(self.nickname):
            del(self.server.users[self.nickname])
    self.request.shutdown(2)
    self.request.close()

def processInput(self):
    """Reads a line from the socket input and either runs it as a
    command, or broadcasts it as chat text."""
    done = False
    l = self._readline()
    command, arg = self._parseCommand(l)
    if command:
        done = command(arg)
    else:
        l = '<%s> %s\n' % (self.nickname, l)
        self.broadcast(l)
    return done

```

Each server command is implemented as a method. The `_parseCommand` method,

## 第III部分 开始使用 Python

defined later, takes a line that looks like /nick and calls the corresponding method (in this case, nickCommand):

#Below are implementations of the server commands.

```
def nickCommand(self, nickname):
    "Attempts to change a user's nickname."
    if not nickname:
        raise ClientError ('No nickname provided.')
    if not self.NICKNAME.match(nickname):
        raise ClientError (Invalid nickname: %s' % nickname)
    if nickname == self.nickname:
        raise ClientError ('You are already known as %s.' % nickname)
    if self.server.users.get(nickname, None):
        raise ClientError ('There\'s already a user named "%s" here.' %
nickname)
    oldNickname = None
    if self.nickname:
        oldNickname = self.nickname
        del(self.server.users[self.nickname])
    self.server.users[nickname] = self.wfile
    self.nickname = nickname
    if oldNickname:
        self.broadcast('%s is now known as %s' % (oldNickname, self.
nickname))

def quitCommand(self, partingWords):
    """Tells the other users that this user has quit, then makes
sure the handler will close this connection."""
    if partingWords:
        self.partingWords = partingWords
    #Returning True makes sure the user will be disconnected.
    return True

def namesCommand(self, ignored):
    "Returns a list of the users in this chat room."
    self.privateMessage(', '.join(self.server.users.keys()))

# Below are helper methods.

def broadcast(self, message, includeThisUser=True):
    """Send a message to every connected user, possibly exempting the
user who's the cause of the message."""
    message = self._ensureNewline(message)
    for user, output in self.server.users.items():
        if includeThisUser or user != self.nickname:
            output.write(message)

def privateMessage(self, message):
    "Send a private message to this user."
    self.wfile.write(self._ensureNewline(message))
```

```

def _readline(self):
    "Reads a line, removing any whitespace."
    return self.rfile.readline().strip()

def _ensureNewline(self, s):
    "Makes sure a string ends in a newline."
    if s and s[-1] != '\n':
        s += '\r\n'
    return s

def _parseCommand(self, input):
    """Try to parse a string as a command to the server. If it's an
    implemented command, run the corresponding method."""
    commandMethod, arg = None, None
    if input and input[0] == '/':
        if len(input) < 2:
            raise ClientError, 'Invalid command: "%s"' % input
        commandAndArg = input[1:].split(' ', 1)
        if len(commandAndArg) == 2:
            command, arg = commandAndArg
        else:
            command, = commandAndArg
        commandMethod = getattr(self, command + 'Command', None)
        if not commandMethod:
            raise ClientError, 'No such command: "%s"' % command
    return commandMethod, arg

if __name__ == '__main__':
    import sys
    if len(sys.argv) < 3:
        print('Usage: %s [hostname] [port number]' % sys.argv[0])
        sys.exit(1)
    hostname = sys.argv[1]
    port = int(sys.argv[2])
    PythonChatServer((hostname, port), RequestHandler).serve_forever()

```

#### 16.4.10 Python 聊天客户端

像镜像服务器那样，该聊天服务器定义了一个简单的、人类可读的协议。通过 telnet 可以使用聊天服务器，但是大多数人还是喜欢使用定制的客户终端。

文件 PythonChatClient.py 是用于 Python 聊天服务器的一个简单的基于文本的客户终端。当使用 telnet 连接服务器时，客户端的几个高级特性将会无法使用。首先，它自己处理身份验证阶段：如果在一个类似于 UNIX 的系统上运行它，甚至不必指定一个昵称，因为它将默认地使用您的账户名称。连接建立之后，Python 聊天客户端运行 /names 命令，向用户展示聊天室中的成员列表。

完成连接后，客户端或多或少地像 telnet 客户端那样运行。它生成一个单独的线程，

处理来自于键盘的用户输入，就像它从网络读取服务器输出一样：

```
#!/usr/bin/python
import socket
import select
import sys
import os
from threading import Thread

class ChatClient:

    def __init__(self, host, port, nickname):
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.socket.connect((host, port))
        self.input = self.socket.makefile('rb', 0)
        self.output = self.socket.makefile('wb', 0)

        #Send the given nickname to the server.
        authenticationDemand = self.input.readline()
        if not authenticationDemand.startswith("Who are you?"):
            raise Exception ("This doesn't seem to be a Python Chat Server.")
        self.output.write(nickname + '\r\n')
        response = self.input.readline().strip()
        if not response.startswith("Hello"):
            raise Exception (response)
        print(response)

        #Start out by printing out the list of members.
        self.output.write('/names\r\n')
        print("Currently in the chat room:", self.input.readline().strip())

        self.run()

    def run(self):
        """Start a separate thread to gather the input from the
        keyboard even as we wait for messages to come over the
        network. This makes it possible for the user to simultaneously
        send and receive chat text."""

        propagateStandardInput = self.PropagateStandardInput(self.output)
        propagateStandardInput.start()

        #Read from the network and print everything received to standard
        #output. Once data stops coming in from the network, it means
        #we've disconnected.
        inputText = True
        while inputText:
            inputText = self.input.readline()
            if inputText:
                print inputText.strip()
```

```

propagateStandardInput.done = True

class PropagateStandardInput(Thread):
    """A class that mirrors standard input to the chat server
    until it's told to stop."""

    def __init__(self, output):
        """Make this thread a daemon thread, so that if the Python
        interpreter needs to quit it won't be held up waiting for this
        thread to die."""
        Thread.__init__(self)
        self.setDaemon(True)
        self.output = output
        self.done = False

    def run(self):
        "Echo standard input to the chat server until told to stop."
        while not self.done:
            inputText = sys.stdin.readline().strip()
            if inputText:
                self.output.write(inputText + '\r\n')

if __name__ == '__main__':
    import sys
    #See if the user has an OS-provided 'username' we can use as a default
    #chat nickname. If not, they have to specify a nickname.
    try:
        import pwd
        defaultNickname = pwd.getpuid(os.getuid())[0]
    except ImportError:
        defaultNickname = None

    if len(sys.argv) < 3 or not defaultNickname and len(sys.argv) < 4:
        print('Usage: %s [hostname] [port number] [username]' % sys.argv[0])
        sys.exit(1)

    hostname = sys.argv[1]
    port = int(sys.argv[2])

    if len(sys.argv) > 3:
        nickname = sys.argv[3]
    else:
        #We must be on a system with usernames, or we would have
        #exited earlier.
        nickname = defaultNickname

    ChatClient(hostname, port, nickname)

```

更加高级的聊天客户端可能会有一个 GUI，为了防止输入在视觉上与输出混淆，它将传入的文本放在一个单独的窗口中。在一个繁忙的聊天室中，您在打字时也许会被传入的

消息打断，并且丢失了您目前所处的位置。

### 16.4.11 基于 select 的单线程多任务

PythonChatClient 生成单独的线程来收集用户输入的原因是，对 `sys.stdin.readline` 的调用直到用户输入一条聊天消息或者服务器命令后才会返回。一个朴素的聊天客户端可能会调用 `sys.stdin.readline` 并且等待用户输入一些内容，但是当它在等待时，其他用户依然继续聊天，来自于服务器的套接字连接将会充满一大堆聊天消息。任何聊天消息都不会显示，直到用户按下 Enter 键(导致 `sys.stdin.readline` 返回)，此时未阅读的消息将全部显示到屏幕上。试图从套接字连接中读数据会引起相反的问题：在聊天室中的其他人说话之前，用户将不能输入任何聊天文本。使用两个线程可以避免这些问题：一个线程监视标准输入，而另一个线程则监视套接字连接。

然而，也可以不使用线程实现聊天客户端(毕竟，telnet 与 PythonChatClient 的工作方式很类似，而且 telnet 程序要比线程的概念出现得早)。秘诀是仅观察标准输入和套接字连接，不是试图从它们中读取数据，而是只判断是否有要读的数据。可以使用 `select` 函数实现该操作，Python 的 `select` 模块提供了这个函数。

`select` 函数的输入为 3 个嵌套列表，每个二级列表包含文件类型对象：一个是读取的对象(如 `sys.stdin`)，一个是写入的对象(如 `sys.stdout`)，还有一个是输出错误时使用的对象(如 `sys.stdout`)。默认情况下，对 `select` 的调用将阻塞(等待输入)，但是只要至少有一个传入的文件类型对象可用，阻塞将解除。它将返回 3 个嵌套列表，其中包含传入的对象的子集：仅包含那些已经就绪并且拥有程序应当关注的数据的对象。可以把 `select` 看做与 Python 内置的 `filter` 函数类似，可以过滤掉那些还未就绪的对象。通过使用 `select`，可以避免在没有可读的数据的文件类型对象上调用 `read` 函数。

下面所示为 ChatClient 的一个子类，它在 `select` 上使用循环，检查标准输入或者服务器输入中是否有未读的数据：

```
class SelectBasedChatClient(ChatClient):

    def run(self):
        """In a tight loop, see whether the user has entered any input
        or whether there's any from the network. Keep doing this until
        the network connection returns EOF."""
        socketClosed = False
        while not socketClosed:
            toRead, ignore, ignore = select.select([self.input, sys.stdin],
                                                    [], [])
            #We're not disconnected yet.
            for input in toRead:
                if input == self.input:
                    inputText = self.input.readline()
                    if inputText:
                        print(inputText.strip())
                else:
                    #The attempt to read failed. The socket is closed.
```

```
        socketClosed = True
    elif input == sys.stdin:
        input = sys.stdin.readline().strip()
        if input:
            self.output.write(input + '\r\n')
```

必须为 `select` 函数传入 3 个列表，而您传入的是输出文件和错误文件的空列表。您只关心两个输入来源(来自于键盘的输入和来自于网络的输入)，因为这些对象在您试图读它们时，可能会阻塞而且引起问题。

在某个意义上，这段代码比原来的 `ChatClient` 更难理解，因为它使用一个小技巧在两件工作之间快速切换，而不是同时做两件工作。在另外一层意义上，它没有原来的 `ChatClient` 复杂，因为它的代码更少，并且不包含多线程，多线程是很难调试的。

在不派生新线程的情况下，也可以使用 `select` 写服务器，但是并不推荐自己编写这样的代码。

## 16.5 其他主题

网络编程的许多方面都未在本章中介绍。最明显的遗漏(驱动万维网的技术和思想)在第 20 章中将会介绍。下面的小节列出了 Python 程序员可能会感兴趣或者认为重要的其他有关网络的主题。

### 16.5.1 协议设计的多种考虑

学习设计协议最好的方法是学习已有的成功的协议。协议通常都具备丰富且编写规范的文档，您可以从这些文档以及 RFC 中学到很多知识。下面列出了一些本章前面没有介绍的在设计协议时常见的一些考虑事项。

#### 1. 可靠的服务器

Python 聊天服务器被一个客户端用来向其他所有客户端广播信息。然而有时，服务器的角色是仲裁连接到它的客户端。在这种情形下，客户端愿意信任服务器提供的信息，而不信任其他的客户端提供的信息。

这在将人们聚集在一起的网站中经常发生，例如拍卖网站和在线支付系统。这个模型在许多在线游戏的协议层也有所实现，其中服务器扮演了调解人的角色。

考虑一个游戏，其中所有玩家都在地图上相互追逐。如果一个玩家知道其他人在地图上的位置，该玩家将得到不公正的优势。同时，如果允许玩家保密他们的位置，就可以作弊，在其他追逐者靠近自己时转移到地图的另外一个部分。所有玩家愿意放弃自己能够作弊，的能力而换来游戏中的其他玩家都不允许作弊的承诺。一个可靠的服务器创建了一片公平的游戏领地。

#### 2. 简洁的协议

客户端可以拼接得到的信息一般不放在协议中。让运行国际象棋的服务器在每次成功

移动一个棋子后，将整个棋盘的表示传递给两个玩家是非常浪费的。向移动棋子的玩家发送“成功移动”，并且对另一个玩家描述此次移动就足够了。基于状态的协议通常传输状态的变化，而不是在状态每次更改时，都要发送当前完整的状态。

Python 聊天服务器的协议用完整的句子发送服务器的状态信息。这使得代码易于理解，并且通过 telnet 使用应用程序也变得容易。客户端行为取决于那些状态信息，例如，PythonChatClient 一连接到服务器就期待字符串“您是谁？”。以这种方式设计协议将使服务器定制状态消息变得困难，或者使得客户端将状态转换成其他语言变得困难。许多协议都为状态消息和命令定义了数值编码或者简写，并在协议的 RFC 中或者其他定义文档中解释这些数值编码和简写的含义。

### 16.5.2 对等架构

本章开发的所有协议都是根据客户端-服务器架构设计的。这个架构将网络上的工作划分到两个不同的软件中：请求数据或服务的客户端软件，以及提供数据或执行服务的服务器软件。该架构假设几个强大的计算机作为服务器，一大群计算机作为客户端。此架构中，信息倾向于集中在服务器端，目的是允许中央控制，确保公平性(例如，在一个有隐私信息的游戏)，使客户端之间不必相互验证，或者使信息更容易找到。

另外一个比较流行的架构是对等架构。在该架构中，每个客户端也是一台服务器。一个对等协议可能会定义“客户端”动作和“服务器”动作，但是每个发出请求的进程也可以满足它们。

尽管基于 TCP/IP 实现的大多数协议都使用客户端-服务器架构，但是 TCP/IP 本身却是一个对等协议。一个套接字连接实际上包含两个单向的 TCP/IP 连接：一个从您的当前地址到您的目的地址，另外一个连接的方向则相反。您不可能在不成为 TCP/IP 服务器的情况下，成为 TCP/IP 客户端，因为这样的话在发送数据后将无法收到应答。

应用程序层上最流行的对等协议是 BitTorrent。BitTorrent 通过在某个大文件的所有下载者之间分担带宽开销使得发布一个文件变得容易。在客户端-服务器架构下，文件的持有者需要将文件放在自己的服务器上，并承受每次下载所需的全部带宽开销。原始的 BitTorrent 由 Python 实现，第一个版本发布于 2002 年。BitTorrent 证明了还有设计新的巧妙的 TCP/IP 协议的空间，也证明了使用 Python 实现高性能协议是可能的。

## 16.6 本章小结

Python 提供了可以使用现有的基于 TCP/IP 的协议的高层工具，使得编写自定义客户端变得容易。它还打包了可以帮助您设计自己的网络应用程序的工具。不论您希望从一个脚本发送邮件，还是想实现 Internet 的下一个关键应用程序，Python 都可以满足您的任何需要。

#### 本章要点：

- smtpplib 模块的名称来源于 SMTP(Simple Mail Transport Protocol, 简单邮件传输协议)。它是发送 Internet 邮件的协议或者标准。

- 协议是组织网络上多方之间发送的数据的约定。
- localhost 是一个特殊的主机名称,它总是指向正在使用的计算机。邮件服务器的主机名称告诉 Python 在 Internet 上的哪个位置寻找它。
- MIME 是一系列标准,它们用来将非 U.S.ASCII 数据融入到组成 U.S.ASCII 的 127 个 7 位字符中。
- 可以使用 mailbox 模块解析 mbox 类型的文件。
- POP3 代表 Post Office Protocol(邮局协议),数字 3 代表版本号。
- IMAP 代表 Internet Message Access Protocol(Internet 消息访问协议)。

## 16.7 习题

1. 区别如下与 E-mail 相关的标准: RFC 2822、SMTP、IMAP、MIME 和 POP。
2. 编写一个连接到 POP 服务器的脚本,下载所有消息,并且将所有消息整理到以消息的发送者命名的文件中(例如,如果收到两封来自于 user@example.com 的邮件,它们都应该被放到称作 user@example.com 的文件中)。如果您有一个 IMAP 服务器,对应的行为会如何?也编写那个脚本(使用 RFC 3501 作为参考)。
3. 假设您正在为低带宽的嵌入式设备(如移动电话)设计一个 IRC 样式的协议。需要对 Python 聊天服务器协议做哪些修改?
4. Python 聊天服务器没有复制的 IRC 的一个特征是/msg 命令,它使得一个用户可以向另外一个用户发送一条私人消息,而不是向整个聊天室的人广播该条消息。如何在 Python 聊天服务器中实现/msg 命令?
5. 什么时候适合设计使用对等架构的协议?





# 第 17 章

## 用 C 扩展编程

不要让任何人误导了您：用 C 语言编写的良好代码将总是比用 Python 编写的代码运行得更快。说到这里，您也不要再次被误导：用 Python 开发代码将总是比用 C 语言开发代码更快些。

初看起来，这可能是件令人左右为难的事。您希望拥有快速的代码，而且也希望迅速地生成代码。使两者保持平衡并解决出现的问题实际上是非常容易的，即选择用 Python 开发代码。毕竟，开发人员的时间比计算机的时间更加宝贵。另外，人类在预测一个系统中将要发生瓶颈的位置方面表现得非常糟糕。通过类似于花大量的时间用 C 语言编写一个全新的程序，而预先在优化代码上花费时间，通常是一种时间上的浪费。这正是受人尊敬的计算机科学家 C. A. R. Hoare 提出“过早的优化是一切罪恶的根源”的原因。当然，他只是在谈论计算机程序，但是道理是类似的。

如果已经编写完代码并优化了算法，但仍然发现性能无法令人满意，就应该通过找出程序将大量时间花费在哪里来分析应用程序、确定瓶颈的位置并用 C 语言重新实现那些微小的部分，将它们作为 Python 的扩展模块。这也是本章将介绍的内容之一。

或者如果您已经拥有了一个用 C 语言编写的代码，并希望在 Python 中利用它们，就可以通过创建一个很小的 Python 扩展模块，使那段 C 代码对 Python 代码可见，以实现 Python 代码对它的调用，就像它是用 Python 编写的一样。这或许是实现扩展模块(使用非 Python 语言编写的模块)的最常见的原因了。

本章将介绍：

- 如何用 C 语言为标准的 Python 解释器创建一个扩展模块(但是只有在完全没有其他选择时才应该这样做)。本章假设您已经对 C 语言很熟悉。如果还不熟悉，就需要找一个熟悉 C 语言的人帮助您摆脱困境。
- 在基本的实际示例中，用 C 语言定义一个类，将原始的声音数据编码为 MP3-编码的数据。可以在 Python 中使用此类，而且此类可以对那些纯粹的 Python 对象进行方法调用，说明如何进行双向的通信。
- 如何在 C 语言中使用 Python API。

本章只是对在 C 语言中使用 Python API 的一个介绍，绝不能替代 <http://docs.python.org/> 上的 API 文档。对于一些在几乎所有示例中都会涉及的函数，您应该先去查看一下它们的函数定义。

## 17.1 扩展模块概述

首先，一个 Python 扩展模块只不过是一个普通的 C 语言库。对于 UNIX 计算机，这些库通常以 .so(表示共享对象)结尾。而对于 Windows 计算机，通常看到的是 .dll 文件(表示动态链接库)。

在开始之前，您需要一些 Python 头文件。对于那些 UNIX 计算机，通常需要安装一个开发人员特有的程序包。而 Windows 用户在使用二进制的 Python 安装程序时，这些头文件将作为程序包的一部分安装。

当第一次查看 Python 扩展模块时，会把代码分成 3 个部分：希望作为模块接口呈现的 C 函数；将 Python 开发人员所看到的那些函数的名称映射为扩展模块中的 C 函数的一个表；以及一个初始化函数。

大多数扩展模块可以包含在一个单独的 C 源文件中，这个文件有时被称为胶水文件 (glue)。启动包含 Python.h 的文件，它允许访问内部的 Python API，这些 API 将模块与解释器相关联。在包含任何可能需要的其他头文件之前，应该先包含 Python.h。在包含的头文件之后是希望在 Python 中调用的函数。

有趣的是，函数的 C 语言实现的签名总是采用如下三种形式之一：

```
PyObject *MyFunction(PyObject *self, PyObject *args);
```

```
PyObject *MyFunctionWithKeywords(PyObject *self,
                                   PyObject *args,
                                   PyObject *kw);
```

```
PyObject *MyFunctionWithNoArgs(PyObject *self);
```

通常，C 函数会采取第一种形式。传递到这些函数中的参数被包装为一个元组，为了使用那些参数，必须分解它们，这解释了为什么可以用 C 语言实现一个只包含两个参数的函数，但是当在 Python 中调用时此函数可以接受任意数量的参数。

请注意之前的那些声明都返回一个 Python 对象。与 C 语言不同，Python 中不存在 void 函数。如果不希望函数返回值，那么可以返回 Python 的 None 值在 C 语言中对应的值。Python 的头文件定义了宏 Py\_RETURN\_NONE，用于实现这个目的。

在看到这些声明之后，Python 是一种怎样的面向对象语言应该很明显了。任何事物都是对象。在 C 语言中将使用 Python API 处理这些对象，但是从 Python 中了解的那些概念仍将有效。

C 函数的名称可以是您喜欢的任何名称，因为在所在的扩展模块外，它们将是不可见的。实际上，这些函数通常使用 static 关键字(在 C 语言中它表示函数在当前的源文件外部是不可见的)声明。在示例代码中，函数通常通过将 Python 模块和函数名称结合到一起来命名，如下所示：

```
static PyObject *foo_bar(PyObject *self, PyObject *args) {
    /* Do something interesting here. */
```

```
Py_RETURN_NONE;
}
```

此函数是在 `foo` 模块内被命名为 `bar` 的 Python 函数。您会将指向 C 函数的指针放入到模块的方法表中，此方法表通常会随后出现在源代码中。

此方法表是 `PyMethodDef` 结构的一个简单数组。该结构如下所示：

```
struct PyMethodDef {
    char            *ml_name;
    PyCFunction      ml_meth;
    int              ml_flags;
    char            *ml_doc;
};
```

第一个成员是 `ml_name`，当在一些 Python 程序中使用它时，它是 Python 解释器将呈现的函数的名称。`PyCFunction` 成员必须是一个函数的地址，此函数具有之前描述的那些签名之一。`ml_flags` 向解释器表明 `ml_meth` 正使用三个签名中的哪一个。`ml_flags` 的值通常是 `METH_VARARGS`。如果希望将关键字参数引入到函数中，那么这个值可以与 `METH_KEYWORDS` 按位或。它的值也可以是 `METH_NOARGS`，表示不希望接受任何参数。最后，`PyMethodDef` 结构中的最后一个成员 `ml_doc` 是函数的文档字符串，如果您不喜欢写一个文档字符串，它可以是 `NULL`（但是建议为函数写文档字符串）。

此方法表必须以一个守护结构结尾，对于合适的成员，它包含了一些 `NULL` 和 0 值。如下所示是包含了 `foo_bar` 函数的相应条目的一个方法表：

```
static PyMethodDef foo_methods[] = {
    { "bar", (PyCFunction)foo_bar, METH_NOARGS, "My first function." },
    { NULL, NULL, 0, NULL }
};
```

为了避免编译器发出关于不匹配指针类型的警告，将 `foo_bar` 的地址强制转换为 `PyCFunction` 是必要的。这么做是安全的，因为 `ml_flags` 成员的值是 `METH_NOARGS` 标志，该标志向 Python 解释器表明它应该只使用一个 `PyObject *` 作为参数去调用 C 函数（而不是使用 `METH_VARARGS` 时的两个参数，或者是使用 `METH_VARARGS|METH_KEYWORDS` 时的三个）。

扩展模块的最后一部分是初始化函数。当模块被加载时，Python 解释器调用此函数。需要将该函数命名为 `initfoo`，其中 `foo` 是模块的名称。

初始化函数必须从您构建的库中导出。针对在特殊环境中进行编译会发生的一些情况，Python 头文件定义了 `PyMODINIT_FUNC`，其中包含了一些适当的特殊方法。所需要做的只是当定义函数的时候使用它。

将所有这些内容放在一起如下所示：

```
#include <Python.h>

static PyObject *foo_bar(PyObject *self, PyObject *args) {
    /* Do something interesting here. */
    Py_RETURN_NONE;
}

static PyMethodDef foo_methods[] = {
    { "bar", (PyCFunction)foo_bar, METH_NOARGS, NULL },
    { NULL, NULL, 0, NULL }
};

PyMODINIT_FUNC inifoo() {
    Py_InitModule3("foo", foo_methods, "My first extension module.");
}
```

因为 `Py_InitModule3` 函数为一个模块定义了一个文档字符串，而定义文档字符串总是一件不错的事情，所以此函数通常用于定义模块。

## 17.2 构建并安装扩展模块

可以使用两种不同的方法构建扩展模块。比较明显的方法是使用在平台上构建所有库的方法去构建扩展模块。将之前的示例保存为 `foo.c`。然后，要在 Linux 上编译扩展模块，可以使用如下命令：

```
gcc -shared -I/usr/include/python3.1 foo.c -o foo.so
```

在 Windows 上构建扩展模块的命令如下所示：

```
cl /LD /IC:\Python31\include foo.c C:\Python31\libs\python31.lib
```

为了使这两个命令正常工作，需要安装一个 C 编译器并使它在您的路径上可用(如果您正在阅读本章，那么可能已经完成了这些工作)。需要安装 Python 头文件并使编译器可以访问它们。在这两个示例中，包含 Python 头文件的目录是通过命令行指定的(Windows 编译器的 Python 库的路径也是一样)。如果头文件和库位于不同的位置，就需要相应地改变命令。

实际共享对象(或 Windows 上的 DDL)的名称需要与传递给 `Py_InitModule3` 的字符串相同(去掉 `.so` 或 `.dll` 扩展名)。可以选择在库的基本文件名后面加上 `module` 后缀。所以 `foo` 扩展模块可以命名为 `foo.so` 或 `foomodule.so`。

这种方法十分有效，但并不是唯一的方法。构建扩展模块的一种新的、改进的方法是使用 `distutils` 程序包，它包含在 Python 的所有近期版本中。

`distutils` 程序包允许以一种标准的方式发布 Python 模块，包括纯粹的 Python 模块和扩展模块。模块以源代码的形式发布，并通过一个安装脚本(通常称为 `setup.py`)构建和安装。

只要用户拥有那些安装所必需的编译程序包和 Python 头文件，这种方法通常是有效的。

安装脚本非常简洁：

```
from distutils.core import setup, Extension
setup(name='foo', version='1.0', ext_modules=[Extension('foo',
['foo.c'])])
```

通过 Python 解释器运行此脚本，您将看到两行代码得到的输出远多于预期：

```
$ python setup.py
usage: setup.py [global_opts] cmd1 [cmd1_opts] [cmd2 [cmd2_opts] ...]
   or: setup.py --help [cmd1 cmd2 ...]
   or: setup.py --help-commands
   or: setup.py cmd --help

error: no commands supplied
```

用 `--help-commands` 参数再试一次，将显示安装脚本可以响应的所有命令：

```
$ python setup.py --help-commands
Standard commands:
  build          build everything needed to install
  build_py       "build" pure Python modules (copy to build
directory)
  build_ext      build C/C++ extensions (compile/link to build
directory)
  build_clib     build C/C++ libraries used by Python extensions
  build_scripts  "build" scripts (copy and fixup #! line)
  clean          clean up output of 'build' command
  install        install everything from build directory
  install_lib    install all Python modules (extensions and pure
Python)
  install_headers install C/C++ header files
  install_scripts install scripts (Python or otherwise)
  install_data   install data files
  sdist          create a source distribution (tarball, zip file,
etc.)
  register       register the distribution with the Python package
index
  bdist          create a built (binary) distribution
  bdist_dumb     create a "dumb" built distribution
  bdist_rpm      create an RPM distribution
  bdist_wininst  create an executable installer for MS Windows

usage: setup.py [global_opts] cmd1 [cmd1_opts] [cmd2 [cmd2_opts] ...]
   or: setup.py --help [cmd1 cmd2 ...]
   or: setup.py --help-commands
   or: setup.py cmd --help
```

这里需要解释的问题很多，但是现在只需要解释 `build` 命令。执行此命令将把 `foo.c` 编译为 `foo.so`(在 Linux 上)或 `foo.dll`(在 Windows 上)。除非通过命令行选项改变存储的目录位置，否则此文件最终将位于当前目录下的 `build` 目录的子目录中。

对于可被 Python 解释器导入的模块，它必须位于当前目录下、`PYTHONPATH` 环境变量中列出的目录下或 `sys.path` 列表中列出的目录下，可以在运行时修改此列表，但是并不推荐这样做。

最简单的实现方法是使用另一个安装脚本命令：

```
$ python setup.py install
```

因为构建是安装的先决条件(非常类似于 `make` 文件)，所以如果还没有构建模块，此命令将完成模块的构建。`install` 命令还会为 Python 安装程序将该模块复制到站点-程序包(`site-package`)目录下。此站点-程序包目录在 `sys.path` 中列出，因此当完成此操作之后，就可以开始使用该模块了。

在基于 UNIX 的系统中，为了得到写入站点-程序包目录的权限，很可能需要作为根用户(`root`)运行此命令。在 Windows 下这通常没有问题。还可以通过使用 `--home` 或 `--prefix` 命令行选项在其他位置安装模块，但是这样做的话，您要负责保证将它们放入一个在运行时 Python 解释器可以找到的目录中。

## 17.3 从 Python 向 C 传递参数

在完成了构建和安装之后，导入新的扩展模块并调用它的函数将十分容易：

```
>>> import foo
>>> dir(foo)
['__doc__', '__file__', '__name__', 'bar']
>>> foo.bar()
```

如果试图向某个函数传递任何参数，解释器将会发出出错信息：

```
>>> foo.bar(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: bar() takes no arguments (1 given)
```

因为很可能希望定义一些确实接受若干个参数的函数，对于这些 C 函数可以使用另外一个签名。例如，可以如下所示定义一个“标准的”函数(接受一些参数的函数)：

```
static PyObject *foo_baz(PyObject *self, PyObject *args) {
    /* Parse args and do something interesting here. */
    Py_RETURN_NONE;
}
```

包含此新函数相应条目的方法表将如下所示：

```
static PyMethodDef foo_methods[] = {
    { "bar", (PyCFunction)foo_bar, METH_NOARGS, NULL },
    { "baz", foo_baz, METH_VARARGS, NULL },
    { NULL, NULL, 0, NULL }
};
```

在对 `foo.c` 完成修改并进行保存之后，您将希望关闭任何打开的、导入了扩展模块的旧版本的解释器，以便可以重新编译源文件，启动一个新的解释器，并导入该扩展模块的新版本。如果在一个窗口中编译而在另一个窗口中调用 `Python`，很容易忘记这么做。

编译该模块的新版本并将其导入，您便能够使用任意数量、任何类型的参数去调用新函数了：

```
>>> foo.baz()
>>> foo.baz(1)
>>> foo.baz(1, 2.0)
>>> foo.baz(1, 2.0, "three")
```

任何参数都可以使用的原因是您还没有编写 C 语言代码来对参数的数目和类型做出限制。

Python API 提供 `PyArg_ParseTuple` 函数，从传入到 C 函数中的 `PyObject` 指针中提取出参数。这是一个 `variadic` 函数，它与标准的 `sscanf` 函数非常类似。

`PyArg_ParseTuple` 的第一个参数是 `args` 参数。它是您将进行“解析”的对象。第二个参数是一个格式字符串，描述了期望显示这些参数的方式。在格式字符串中每个参数用一个或多个字符表示。`i` 表示期望该参数是一个类似整型的对象，`PyArg_ParseTuple` 将其转换为一个 C 语言中的 `int` 类型。在格式字符串中指定 `d` 将提供一个 `double` 类型，而 `s` 将提供一个字符串(`char *`)类型。例如，如果您期望向 `baz` 函数传递一个整数、一个双精度数和一个字符串，那么格式字符串将是“`ids`”。<http://docs.python.org/api/arg-parsing.html> 上包含了一个完整的列表，其中列出了可以在格式字符串中包含的指示器。

`PyArg_ParseTuple` 的其余参数是一些指针，它们指向参数的适当类型的存储空间，就像 `sscanf` 一样。在了解了这一点之后，可以将 `baz` 改写成如下形式：

```
static PyObject *foo_baz(PyObject *self, PyObject *args) {
    int    i;
    double d;
    char   *s;
    if (!PyArg_ParseTuple(args, "ids", &i, &d, &s)) {
        return NULL;
    }
    /* Do something interesting here. */
    Py_RETURN_NONE;
}
```

如果 `PyArg_ParseTuple` 不能正确提取格式字符串中指定的参数，它将返回 0。当这种情况发生时，从函数中返回 `NULL` 将是非常重要的，这样解释器可以为调用者生成一个异常。

如何处理可选参数呢？如果在格式字符串中包含一个 “|” (竖线) 字符，“|” 左边的指示器是必需的，“|” 右边的指示器是可选的。对于可选参数，将希望为其本地存储提供一个默认值，因为如果调用者不指定必要的参数，`PyArg_ParseTuple` 将不会向那些变量中写入任何值。

例如，如果 `baz` 需要一个整数、一个双精度数和一个字符串，但是也允许一个可选的整数、双精度数和字符串，那么可以将其改写为如下形式：

```
static PyObject *foo_baz(PyObject *self, PyObject *args) {
    int    i;
    double d;
    char   *s;
    int    i2 = 4;
    double d2 = 5.0;
    char   *s2 = "six";
    if (!PyArg_ParseTuple(args, "ids|ids", &i, &d, &s, &i2, &d2,
&s2)) {
        return NULL;
    }
    /* Do something interesting here. */
    Py_RETURN_NONE;
}
```

最后，C 函数可能采用的下一种也是最后一种形式仅在函数接受关键字参数时才是必要的。在这种情况下，使用接受三个 `PyObject *` 参数的签名，并将方法表条目中的 `ml_flags` 成员设置为 `METH_VARARGS|METH_KEYWORDS`。使用 `PyArg_ParseTupleAndKeywords` 函数提取参数，而不是使用 `PyArg_ParseTuple`。

该函数可能如下所示：

```
static PyObject *foo_quux(PyObject *self, PyObject *args, PyObject
*kw) {
    char *kwlist[] = { "i", "d", "s", NULL };
    int    i;
    double d = 2.0;
    char   *s = "three";
    if (!PyArg_ParseTupleAndKeywords(args, kw, "i|ds", kwlist, &i,
&d, &s)) {
        return NULL;
    }
    /* Do something interesting here. */
    Py_RETURN_NONE;
}
```

如下所示是此函数在方法表中的条目，它在 `baz` 函数的条目之后，守护条目之前：

```
{ "quux", (PyCFunction)foo_quux, METH_VARARGS|METH_KEYWORDS, NULL },
```

除了具有两个额外的参数，`PyArg_ParseTupleAndKeywords` 函数的工作方式与 `PyArg_ParseTuple` 函数类似。首先，需要传递指向包含关键字参数的 Python 对象的指针。其次，需要指出感兴趣的關鍵字。这是通过一个以 `NULL` 结束的字符串列表实现的。在之前的示例中，关键字是 `i`、`d` 和 `s`。

即使不打算让调用者对某些参数使用关键字，每个关键字也需要与格式字符串中的一个指示器相对应。注意之前的示例在格式字符串中包含了三个指示器。第一个指示器 `i` 是必需的，而 `d` 和 `s` 是可选的。可以用如下的任何方式从 Python 中调用此函数：

```
>>> foo.quux(1)
>>> foo.quux(i=1)
>>> foo.quux(1, 2.0)
>>> foo.quux(1, 2.0, "three")
>>> foo.quux(1, 2.0, s="three")
>>> foo.quux(1, d=2.0)
>>> foo.quux(1, s="three")
>>> foo.quux(1, d=2.0, s="three")
>>> foo.quux(1, s="three", d=2.0)
>>> foo.quux(i=1, d=2.0, s="three")
>>> foo.quux(s="three", d=2.0, i=1)
```

您也许能够想出更多调用此函数的方式。

## 17.4 从 C 向 Python 返回值

`PyArg_ParseTuple` 和 `PyArg_ParseTupleAndKeywords` 将一些 Python 对象转换为 C 语言中的值，但是反过来呢？如何从 C 语言实现的函数中向 Python 返回一个值？

之前所见的所有函数签名都返回一个 `PyObject *`，因此为了将一个 C 语言中的值转换成一个 Python 对象，需要使用与 `PyArg_ParseTuple` 相反的函数。这个函数称作 `Py_BuildValue`。

同 `PyArg_ParseTuple` 一样，`Py_BuildValue` 接受一个格式字符串。向它传递一些实际的值，而不是传递正在构建的值的地址。下面的示例显示如何实现一个加法函数：

```
static PyObject *foo_add(PyObject *self, PyObject *args) {
    int a;
    int b;
    if (!PyArg_ParseTuple(args, "ii", &a, &b)) {
        return NULL;
    }
    return Py_BuildValue("i", a + b);
}
```

与其对应的 Python 函数如下所示：

```
def add(a, b):
    return a + b
```

如果希望从函数中返回多个值，那么该怎么办？在 Python 中，通过返回一个元组来实现此功能。在 C 语言中，通过使用 `Py_BuildValue` 构建一个元组来实现。如果格式字符串包含多个指示器，那么将会获得一个元组。也可以直接用圆括号将指示器括起来：

```
static PyObject *foo_add_and_subtract(PyObject *self, PyObject *args)
{
    int a;
    int b;
    if (!PyArg_ParseTuple(args, "ii", &a, &b)) {
        return NULL;
    }
    return Py_BuildValue("(ii)", a + b, a - b);
}
```

为了帮助想象此函数的功能，使用 Python 实现的相应函数如下所示：

```
def add_and_subtract(a, b):
    return (a + b, a - b)
```

有了这些知识以后，您就可以创建出各种各样的扩展模块。下面合理利用这些知识，用一个实际的示例进行练习。

## 17.5 LAME 项目

LAME 是一个只取首字母的缩写词，原来代表 “LAME Ain't an MP3 Encoder”。它是否是官方认可的 MP3 编码器对您并不重要，因为它的功能是作为一个(非常优秀的)免费的、能够编码 MP3 的开源库。

有很多软件项目使用 LAME，但是用 Python 实现的并不多，因此我们使用它作为一个示例，以说明利用已有的 C 代码库为 Python 创建扩展模块是多么简单，即使此 C 代码的编码与 Python 没有接口。

它同时是一个非常实用的示例。开发 LAME 代码库用了许多年时间。您真的希望通过使用 Python 重新实现它来重复那些工作吗？现在考虑一下，如果告诉您在只有一个 Python 编码器时，它的运行异常缓慢，您的答案将是什么。顺便说一下，这不是针对 Python。对于比 C 语言高级的任何语言，这都是事实。Java、Perl 等语言都有相同的限制。这是您不希望使用 Python 进行开发的一个典型的示例(这样的示例非常少)。

在创建一个包装了 LAME 库的扩展模块之前，需要学会如何使用该库提供的 API。LAME API 的核心部分非常小，以至于只用大约一页的 C 代码就可以创建一个快速演示。

当然，在可以使用 LAME 的 API 编写任何代码之前，需要将 LAME 头文件和库安装到计算机上。LAME 项目在 SourceForge 上的地址是 <http://lame.sourceforge.net/>。可以从那

里下载源代码。虽然可以从那里下载并编译和安装 LAME 程序包的任何部分的库,但是在此站点上找不到任何预构建的二进制文件(大概是为了避免发布 MP3 编码器可能带来的法律问题)。然而,可以通过在 LAME 项目的网站上进行搜索,找到一些提供可下载的二进制文件的站点的链接(如果不愿从源代码中构建)。

对于大多数 Linux 发行版本来说,可以在 Web 上找到相应的程序包。这些程序包可能以 lame、liblame 或 liblame-dev 程序包等名称列出。如果找不到程序包或者希望从源码进行构建, ./configure、make 和 make install 可以构建一个完整有效的 LAME 安装,正如在 Linux 上几乎所有的其他项目都可以使用它们从源代码构建一样。

Windows 用户可以使用任何预构建的二进制文件,但是它们通常不提供头文件,因此需要从主站点上下载它们。与其这样做,不如自己去构建这些库。LAME 源代码包含一个 Visual Studio 工作区,对于读完本章的其余部分所需要的任何程序,它都可以构建。可能会存在一些错误(这是作者的原因),但是此构建过程完全可以构建出需要的东西,因此忽略那些错误是没问题的。

用 LAME 创建 MP3 文件的过程如下:

- (1) 初始化库。
- (2) 设置一些编码参数。
- (3) 每次向库提供一个音频数据的缓冲区(返回此数据的 MP3 编码字节的另一个缓冲区)。
- (4) 清空编码器(可能返回更多的 MP3 数据)。
- (5) 关闭库。

如下所示是一个用 C 语言编写的、使用 LAME API 的示例。它可以将任何原始的音频文件编码为一个 MP3 编码的音频文件。如果希望对它进行编译以保证它是有效的,那么将其保存在一个名为 clame.c 的文件中:

```
#include <stdio.h>
#include <stdlib.h>

#include <lame.h>

#define INBUFSIZE 4096
#define MP3BUFSIZE (int)(1.25 * INBUFSIZE) + 7200

int encode(char *inpath, char *outpath) {
    int status = 0;
    lame_global_flags *gfp;
    int ret_code;
    FILE *infp;
    FILE *outfp;
    short *input_buffer;
    int input_samples;
    char *mp3_buffer;
    int mp3_bytes;

    /* Initialize the library. */
    gfp = lame_init();
```

```

if (gfp == NULL) {
    printf("lame_init returned NULL\n");
    status = -1;
    goto exit;
}

/* Set the encoding parameters. */
ret_code = lame_init_params(gfp);
if (ret_code < 0) {
    printf("lame_init_params returned %d\n", ret_code);
    status = -1;
    goto close_lame;
}

/* Open our input and output files. */
infp = fopen(inpath, "rb");
outfp = fopen(outpath, "wb");

/* Allocate some buffers. */
input_buffer = (short*)malloc(INBUFSIZE*2);
mp3_buffer = (char*)malloc(MP3BUFSIZE);

/* Read from the input file, encode, and write to the output
file. */
do {
    input_samples = fread(input_buffer, 2, INBUFSIZE, infp);
    if (input_samples > 0) {
        mp3_bytes = lame_encode_buffer_interleaved(
            gfp,
            input_buffer,
            input_samples / 2,
            mp3_buffer,
            MP3BUFSIZE
        );
        if (mp3_bytes < 0) {
            printf("lame_encode_buffer_interleaved returned
%d\n", mp3_bytes);
            status = -1;
            goto free_buffers;
        } else if (mp3_bytes > 0) {
            fwrite(mp3_buffer, 1, mp3_bytes, outfp);
        }
    }
} while (input_samples == INBUFSIZE);

/* Flush the encoder of any remaining bytes. */
mp3_bytes = lame_encode_flush(gfp, mp3_buffer,
sizeof(mp3_buffer));
if (mp3_bytes > 0) {
    printf("writing %d mp3 bytes\n", mp3_bytes);
}

```

```

        fwrite(mp3_buffer, 1, mp3_bytes, outfp);
    }

    /* Clean up. */

free_buffers:
    free(mp3_buffer);
    free(input_buffer);

    fclose(outfp);
    fclose(infp);

close_lame:
    lame_close(gfp);

exit:
    return status;
}

int main(int argc, char *argv[]) {
    if (argc < 3) {
        printf("usage: clame rawinfile mp3outfile\n");
        exit(1);
    }
    encode(argv[1], argv[2]);
    return 0;
}

```

如果要在 Linux 上编译此文件，应该使用如下命令(假设已经安装了一个类似于 liblame-dev 的程序包，或者 lame 开发组件已经在/usr/include/lame 中安装了合适的头文件)：

```
gcc -I/usr/include/lame clame.c -lmp3lame -o clame
```

在 Windows 上，可能需要使用如下所示的命令(假设从源代码进行构建)：

```
cl /IC:\lame-3.98.2\include clame.c \
C:\lame-3.98.2\libmp3lame\Release\libmp3lame.lib \
C:\lame-3.98.2\mpglib\Release\mpglib.lib
```

这些命令行参数告知编译器查找 LAME 头文件和必要的库的位置。可能需要对它们进行调整以指向正确的目录。

还不错，是吧？当然，此代码不能从一个 WAV 或任何其他类型的音频文件中提取数据。这里假设输入文件只包含原始的、16 位的、44.1kHz 的有符号样本。在大多数基于 UNIX 的计算机上，将一个 WAV 文件转换为这样的原始文件只需要一个很简单的命令(假设有 sox 程序，而且它作为一个程序包应该是可用的)：

```
sox test.wav -t raw test.raw
```

## 17.6 LAME 扩展模块

创建一个能够将原始的音频文件编码为 MP3 文件的扩展模块只需要创建一个简单的函数，此函数调用在之前的示例中定义的 `encode` 函数：

```
#include <Python.h>

#include <lame.h>

/* defined in clame.c */
int encode(char *, char *);

static PyObject *pylame1_encode(PyObject *self, PyObject *args) {
    int status;
    char *inpath;
    char *outpath;
    if (!PyArg_ParseTuple(args, "ss", &inpath, &outpath)) {
        return NULL;
    }
    status = encode(inpath, outpath);
    return Py_BuildValue("i", status);
}

static PyMethodDef pylame1_methods[] = {
    { "encode", pylame1_encode, METH_VARARGS, NULL },
    { NULL, NULL, 0, NULL }
};

PyMODINIT_FUNC initspylame1() {
    Py_InitModule3("pylame1", pylame1_methods, "My first LAME
module.");
}
```

在这里，`encode` 函数接受两个字符串参数，即输入路径和输出路径。

试着将之前的代码保存在一个以 `pylame1.c` 命名的文件中，并使用如下命令对它进行编译：

```
gcc -shared -I/usr/include/python3.1 -I/usr/include/lame \
    pylame1.c clame.c \
    -lmp3lame -o pylame1.so
```

在 Windows 上，需要的命令如下所示：

```
cl /LD /IC:\Python31\include /IC:\lame-3.96.1\include \
    pylame1.c clame.c \
    C:\Python31\libs\python31.lib \
    C:\lame-3.98.2\libmp3lame\Release\libmp3lame.lib \
    C:\lamexs-3.98.2\mpglib\Release\mpglib.lib
```

注意，通过将前一节中使用的同一个 `clame.c` 示例包含到命令行中，将它编译为一个 DLL 文件。

这样做是可行的，但是并不理想，因为除了传递两个字符串之外，没有其他方式能够影响 `encode` 函数的工作方式。如果希望对原始音频文件之外的某种格式进行编码，那该怎么办？如果是 WAV 文件或者网络上的音频数据流，那该怎么办？没有理由不能使用 Python 实现这种功能，实际上在 Python 中这是更容易实现的。

您有两个选择：就像在 C 函数中所做的那样，可以使用 Python 代码向 `encode` 函数传递音频数据，每次一大块。或者，可以将带有 `read` 方法的某个对象传递到 `encode` 中，`encode` 再从该对象中读出数据。

虽然第二种选择可能听起来更加面向对象，但是第一种是更好的选择，因为它提供了更大的灵活性。总是可以定义某类对象，它从某个数据源中读取数据并将其传递给编码器，但是反过来则困难得多。

使用这种设计需要在扩展模块中做出一些修改。目前只有一个函数，不过没关系，因为该函数实现了所有的工作。然而，对于新的方法，将多次调用把音频数据编码为 MP3 数据的函数。不能在每次调用该函数时让它重新打开文件，因此需要在其他某个位置保存一些关于在文件中的位置的状态信息。可以让调用者保存那个状态，或者可以将其封装到模块定义的某个对象中，这里将采用第二种方法。

新版本的扩展模块需要将一个类对外公开，以便客户可以创建此类的实例并调用它们的方法。我们将在那些实例中隐藏少量的状态信息，因此在两个方法调用之间它们能够记住正在写入哪个文件。

在介绍对这个新模块所需完成的工作的同时，将给出与正被解释的内容有关的一些代码片段。稍后会给出 `pylame2.c` 的全部代码，那时您便可以看到所有这些代码片段的整体效果了。

C 语言语法不直接支持定义一个新类，但是它包含结构类型，而且 C 语言中的结构可以包含一些函数指针，这足以实现接下来要实现的功能。当 Python 解释器创建类的一个新实例时，它实际上将分配足够的空间来存储相应结构的一个新实例。这个结构中会包含每个对象的所有状态。

Python 解释器还需要在对象中存储一些信息。每个对象包含一个引用计数和一个类型，所以该结构的第一部分需要保存这些信息，以使 Python 解释器能找到它们：

```
typedef struct {
    PyObject_HEAD
    /* State goes here. */
} pylame2_EncoderObject;
```

`PyObject_HEAD` 宏将一些适当的成员添加到该结构中，您只需要保证最先添加它就行了。

需要提供一个函数来创建此结构的新实例：

```
static PyObject *Encoder_new(PyTypeObject *type, PyObject *args,
    PyObject *kw) {
```

```

    pylame2_EncoderObject *self = (pylame2_EncoderObject *)
type->tp_alloc(type, 0);
    /* Initialize object here. */
    return (PyObject *)self;
}

```

可以认为此函数等同于 Python 的 `__new__` 方法。当解释器需要创建该类型的一个新实例时，将调用此函数。注意您并不直接调用 `malloc`，相反，调用了被传入的 `PyTypeObject` 的 `tp_alloc` 成员所指定的其他某个函数。稍后将讨论函数。

还需要一个释放实例的函数：

```

static void Encoder_dealloc(PyObject *self) {
    self->ob_type->tp_free(self);
}

```

可以认为此函数等同于 Python 的 `__del__` 方法，它与 `Encoder_new` 是配对的。因为此处是调用对象的类型对象的 `tp_free` 函数，所以您可能认为 `tp_free` 函数与 `tp_alloc` 函数是配对的，这样想是正确的。

对象应该支持的其他那些方法怎么办？会向结构中添加一些函数指针来表示它们吗？如果这样做，那么每个实例将以相同的指针集合占用内存，这将是一种浪费。相反，可以在另一个单独的结构中存储那些方法的函数指针，然后对象再引用那个结构。

请记住每个对象都知道它的类型，因为存在一个指向类型对象的指针，它隐藏在 `PyObject_HEAD` 宏的内部。因此，需要用另一个结构表示它：

```

static PyTypeObject pylame2_EncoderType = {
    PyObject_HEAD_INIT(NULL)
    0, /* ob_size */
    "pylame2.Encoder", /* tp_name */
    sizeof(pylame2_EncoderObject), /* tp_basicsize */
    0, /* tp_itemsize */
    Encoder_dealloc, /* tp_dealloc */
    0, /* tp_print */
    0, /* tp_getattr */
    0, /* tp_setattr */
    0, /* tp_compare */
    0, /* tp_repr */
    0, /* tp_as_number */
    0, /* tp_as_sequence */
    0, /* tp_as_mapping */
    0, /* tp_hash */
    0, /* tp_call */
    0, /* tp_str */
    0, /* tp_getattro */
    0, /* tp_setattro */
    0, /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT, /* tp_flags */
    "My first encoder object.", /* tp_doc */

```

```

0,          /* tp_traverse */
0,          /* tp_clear */
0,          /* tp_richcompare */
0,          /* tp_weaklistoffset */
0,          /* tp_iter */
0,          /* tp_iternext */
0,          /* tp_methods */
0,          /* tp_members */
0,          /* tp_getset */
0,          /* tp_base */
0,          /* tp_dict */
0,          /* tp_descr_get */
0,          /* tp_descr_set */
0,          /* tp_dictoffset */
0,          /* tp_init */
0,          /* tp_alloc */
Encoder_new, /* tp_new */
0,          /* tp_free */
};

```

当调用 `Encoder_new` 函数时，将得到一个指向此结构的指针。该结构涉及很多方面(甚至有些现在还看不到)，但是目前大多数成员的值都默认为 `NULL`。在继续进行之前，先看看那些最重要的成员。

`PyObject_HEAD_INIT` 宏添加所有类型公有的一些成员。它必须是该结构中的第一个成员。这个成员看起来像 `PyObject_HEAD`，只是它将传入的任何类型指针初始化为一个参数。

请记住：在 Python 中，类型也是对象，因此它们也具有类型。可以将一个类型的类型称为“类型类型”。Python API 将其称为 `PyType_Type`。它是类型对象的类型。您实际上很希望能够向这个宏中传入 `&PyType_Type`，但是某些编译器不允许使用在其他某个模块中定义的符号静态初始化一个结构的成员，因此稍后将必须对它进行填充。

下一个成员是 `ob_size`，它看起来可能很重要，但实际上它是 Python API 的一个旧版本的遗留部分，所以应该被忽略。类型名称之后的成员，即 `tp_basicsize`，表示所有对象实例的大小。当解释器需要为一个新实例分配存储空间时，它将请求 `tp_basicsize` 个字节。

目前，其余的大多数成员都是 `NULL`，但是稍后将对它们进行填充。它们为一些比较常见的操作保存函数指针，这些操作是很多对象都支持的。

`tp_flags` 成员指定类型对象的一些默认标志，它是所有类型对象都需要的。`tp_doc` 成员保存一个指向该类型的文档字符串的指针，作为一个优秀的 Python 程序员，总是应该提供文档字符串。

请注意 `tp_alloc` 和 `tp_free` 成员都被设置为 `NULL`。它们就是从 `Encoder_new` 和 `Encoder_dealloc` 中调用的那些成员，但是稍后将使用一个 Python API 函数使用合适的地址填充它们，因为一些平台不支持使用其他库中一些函数的地址去静态初始化结构的成员。

此处，已经定义了两种结构。为了通过扩展模块使它们实际可用，需要向模块的初始化函数中添加一些代码：

```
PyMODINIT_FUNC initspylame2() {
    PyObject *m;
    if (PyType_Ready(&pylame2_EncoderType) < 0) {
        return;
    }
    m = Py_InitModule3("pylame2", pylame2_methods, "My second LAME
module.");
    Py_INCREF(&pylame2_EncoderType);
    PyModule_AddObject(m, "Encoder", (PyObject *)
&pylame2_EncoderType);
}
```

`PyType_Ready` 为解释器的使用准备好一个类型对象。它将对象的类型设置为 `PyType_Type`，并设置了许多之前已经设置为 `NULL` 的函数指针成员，以及其他很多将所有任务适当地关联到一起的必要的记录任务，包括将 `tp_alloc` 和 `tp_free` 成员设置为适当的函数。

在准备好类型对象之后，便可以正常地创建模块，但是这次对返回值(一个指向模块对象的指针)进行保存，以便可以将新的类型对象添加到模块中。以前忽略了返回值，并让方法表定义模块的所有成员。因为不可能将一个 `PyObject` 指针填充到方法表中，所以需要使用 `PyModule_AddObject` 函数将类型对象添加到模块中。此函数接受指向从 `Py_InitModule3` 返回的模块的指针、新对象在模块内的名称以及指向新对象自身的指针。

如果对目前的所有代码进行编译，可以创建一些新的 `Encoder` 实例：

```
>>> import pylame2
>>> e = pylame2.Encoder()
```

然而那个对象并不起多大作用，因为它还不具有任何有用的行为。

为了使这些对象变得有用，需要将一些信息传递到它们的初始化函数中。这些信息可能只是希望向其写入的文件的名称。初始化函数可以使用该名称打开一个文件句柄，以便向其中写入一些信息，但是直到有人调用了对象的 `encode` 方法才会发生写操作。因此，对象需要保持它打开的文件的句柄。

还需要调用在 `LAME` 库中定义的一些函数，因此对象还需要记住 `lame_init` 返回的指向 `lame_global_flags` 结构的指针。

如下所示是这个包含状态的结构，以及一个对其进行初始化的改进的 `Encoder_new` 函数：

```
typedef struct {
    PyObject_HEAD
    FILE *outfp;
    lame_global_flags *gfp;
} pylame2_EncoderObject;

static PyObject *Encoder_new(PyTypeObject *type, PyObject *args,
PyObject *kw) {
    pylame2_EncoderObject *self = (pylame2_EncoderObject *)
type->tp_alloc(type, 0);
    self->outfp = NULL;
```

```

    self->gfp = NULL;
    return (PyObject *)self;
}

```

因为此函数等同于 Python 的 `__new__` 方法，而是不等同于 `__init__`，所以此处并不对 `args` 和 `kw` 进行检查。打开文件并对 LAME 库进行初始化是在 `__init__` 的 C 语言实现中完成的：

```

static int Encoder_init(pylame2_EncoderObject *self,
                        PyObject *args, PyObject *kw) {
    char *outpath;
    if (!PyArg_ParseTuple(args, "s", &outpath)) {
        return -1;
    }
    if (self->outfp || self->gfp) {
        PyErr_SetString(PyExc_Exception, "__init__ already called");
        return -1;
    }
    self->outfp = fopen(outpath, "wb");
    self->gfp = lame_init();
    lame_init_params(self->gfp);
    return 0;
}

```

`__init__` 的实现检查了两点。第一点您已经看到了，即使用 `PyArg_ParseTuple` 保证传入了一个字符串参数。第二点检查是保证实例的 `outfp` 和 `gfp` 成员是 `NULL`。如果它们不为 `NULL`，那么此对象必定已经调用过该函数，因此在使用 `PyErr_SetString` 函数设置了一个异常之后，将为该函数返回相应的错误代码。在返回到 Python 解释器之后，将产生一个异常，则调用者将必须捕获它或者承担后果。因为总是可能两次调用同一个对象的 `__init__`，所以这样做是必需的。有了这段代码，如果对对象的 `__init__` 进行两次调用，那么结果可能如下所示：

```

>>> import pylame2
>>> e = pylame2.Encoder("foo.mp3")
>>> e.__init__("bar.mp3")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
Exception: __init__ already called

```

当然，可以对该对象进行重新初始化，但是对于目前所希望完成的任务，并不是必须这么做。不过，还是应该检查那些错误。

为了表明希望对该类的每个新实例调用此初始化函数，需要在类型对象中添加该初始化函数需要的地址。

```

(initproc)Encoder_init,      /* tp_init */

```

因为之前耍了个小聪明，声明 `Encoder_init` 接受 `pylame2_EncoderObject *` 作为其第一

个参数，而不是使用更通用的 `PyObject *`，所以在此处要对它进行强制转换。在 C 语言中，这种做法是有效的，但是必须绝对确定知道自己在做什么。

因为现在实例包含了对一些资源进行引用的状态，所以需要保证当释放对象时，那些资源被适当地处理。为此，对 `Encoder_dealloc` 函数进行修改：

```
static void Encoder_dealloc(pylame2_EncoderObject *self) {
    if (self->gfp) {
        lame_close(self->gfp);
    }
    if (self->outfp) {
        fclose(self->outfp);
    }
    self->ob_type->tp_free(self);
}
```

如果使用迄今为止已有的代码去构建模块，然后导入该模块，创建一个编码器对象，之后再将其删除(使用 `del` 关键字或者将引用对象的变量与 `None` 或者其他某个对象重新进行绑定)，那么在当前目录下最终将得到一个空文件，因为所做的所有工作无非是打开文件然后关闭它，并没有进行任何写入操作。但是很快就会进行写入文件的操作。

现在需要向类型中增加对 `encode` 和 `close` 方法的支持。之前，已经创建了一种方法表，但是它实际上只定义了一些模块级的函数。为类定义一些方法同样简单，但是却有一些不同之处。定义与模块级函数类似的一些方法，然后创建一个表将它们列出：

```
static PyObject *Encoder_encode(PyObject *self, PyObject *args) {
    Py_RETURN_NONE;
}

static PyObject *Encoder_close(PyObject *self) {
    Py_RETURN_NONE;
}

static PyMethodDef Encoder_methods[] = {
    { "encode", Encoder_encode, METH_VARARGS,
      "Encodes and writes data to the output file." },
    { "close", (PyCFunction)Encoder_close, METH_NOARGS,
      "Closes the output file." },
    { NULL, NULL, 0, NULL }
};
```

然后，该表的地址用于初始化类型对象的 `tp_methods` 成员：

```
Encoder_methods,          /* tp_methods */
```

随着这些占位程序(stub)的就绪，可以构建模块并查看那些方法，甚至可以调用对象的方法：

```

>>> import pylame2
>>> e = pylame2.Encoder('foo.mp3')
>>> dir(e)
['__class__', '__delattr__', '__doc__', '__getattribute__',
 '__hash__', '__init__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__str__', 'close', 'encode']
>>> e.encode()
>>> e.close()

```

现在所需要做的是实现这些函数。如下所示是 `Encoder_encode` 的实现(没有进行完整的错误检查):

```

static PyObject *Encoder_encode(pylame2_EncoderObject *self,
PyObject *args) {
    char *in_buffer;
    int in_length;
    int mp3_length;
    char *mp3_buffer;
    int mp3_bytes;
    if (!(self->outfp && self->gfp)) {
        PyErr_SetString(PyExc_Exception, "encoder not open");
        return NULL;
    }
    if (!PyArg_ParseTuple(args, "s#", &in_buffer, &in_length)) {
        return NULL;
    }
    in_length /= 2;
    mp3_length = (int)(1.25 * in_length) + 7200;
    mp3_buffer = (char *)malloc(mp3_length);
    if (in_length > 0) {
        mp3_bytes = lame_encode_buffer_interleaved(
            self->gfp,
            (short *)in_buffer,
            in_length / 2,
            mp3_buffer,
            mp3_length
        );
        if (mp3_bytes > 0) {
            fwrite(mp3_buffer, 1, mp3_bytes, self->outfp);
        }
    }
    free(mp3_buffer);
    Py_RETURN_NONE;
}

```

您期望向此参数传递一个字符串。C 语言中的字符串是简单的以 NULL 结束的字符数组, 而与之不同, 这里的字符串可以包含一些内嵌的 NULL 字符(在 C 语言中, NULL 字符只是字符串的结束标志, 其值是 '\0'。请注意单引号, 记住在 C 语言中不同类型的引号具有不同的意义。在 C 语言中 NULL 也可以表示为 " ")。因此, 当对这些参数进行解析时,

不使用"s"指示器,而是使用"s#",它允许内嵌NULL字符。PyArg\_ParseTuple 将返回缓冲区中的各个字节和缓冲区的长度,而不在末尾附加一个NULL字符。除此之外,这个函数是非常简单的。

如下所示是 Encoder\_close 的实现:

```
static PyObject *Encoder_close(pylame2_EncoderObject *self) {
    int mp3_length;
    char *mp3_buffer;
    int mp3_bytes;
    if (!(self->outfp && self->gfp)) {
        PyErr_SetString(PyExc_Exception, "encoder not open");
        return NULL;
    }
    mp3_length = 7200;
    mp3_buffer = (char *)malloc(mp3_length);
    mp3_bytes = lame_encode_flush(self->gfp, mp3_buffer,
sizeof(mp3_buffer));
    if (mp3_bytes > 0) {
        fwrite(mp3_buffer, 1, mp3_bytes, self->outfp);
    }
    free(mp3_buffer);
    lame_close(self->gfp);
    self->gfp = NULL;
    fclose(self->outfp);
    self->outfp = NULL;
    Py_RETURN_NONE;
}
```

此处需要保证将 outfp 和 gfp 设置为 NULL,以防止 Encoder\_dealloc 试图再次关闭它们。

对于 Encoder\_encode 和 Encoder\_close,需要对它们进行检查,以保证对象处于一个可以进行编码和关闭的有效状态。有些人可能会在调用 close 后再次调用 close,甚至调用 encode。产生一个异常要比使托管扩展模块的进程崩溃好得多。

到目前为止,您已经完成了很多工作,如果能将整个扩展模块放在一个大的示例中进行查看,将是非常有帮助的:

```
#include <Python.h>

#include <lame.h>

typedef struct {
    PyObject_HEAD
    FILE *outfp;
    lame_global_flags *gfp;
} pylame2_EncoderObject;

static PyObject *Encoder_new(PyTypeObject *type, PyObject *args,
PyObject *kw) {
```

```

    pylame2_EncoderObject *self = (pylame2_EncoderObject *)
type->tp_alloc(type, 0);
    self->outfp = NULL;
    self->gfp = NULL;
    return (PyObject *)self;
}

static void Encoder_dealloc(pylame2_EncoderObject *self) {
    if (self->gfp) {
        lame_close(self->gfp);
    }
    if (self->outfp) {
        fclose(self->outfp);
    }
    self->ob_type->tp_free(self);
}

static int Encoder_init(pylame2_EncoderObject *self, PyObject *args,
PyObject *kw) {
    char *outpath;
    if (!PyArg_ParseTuple(args, "s", &outpath)) {
        return -1;
    }
    if (self->outfp || self->gfp) {
        PyErr_SetString(PyExc_Exception, "__init__ already called");
        return -1;
    }
    self->outfp = fopen(outpath, "wb");
    self->gfp = lame_init();
    lame_init_params(self->gfp);
    return 0;
}

static PyObject *Encoder_encode(pylame2_EncoderObject *self,
PyObject *args) {
    char *in_buffer;
    int in_length;
    int mp3_length;
    char *mp3_buffer;
    int mp3_bytes;
    if (!(self->outfp && self->gfp)) {
        PyErr_SetString(PyExc_Exception, "encoder not open");
        return NULL;
    }
    if (!PyArg_ParseTuple(args, "s#", &in_buffer, &in_length)) {
        return NULL;
    }
    in_length /= 2;
    mp3_length = (int)(1.25 * in_length) + 7200;
    mp3_buffer = (char *)malloc(mp3_length);
    if (in_length > 0) {

```

## 第III部分 开始使用 Python

```

        mp3_bytes = lame_encode_buffer_interleaved(
            self->gfp,
            (short *)in_buffer,
            in_length / 2,
            mp3_buffer,
            mp3_length
        );
        if (mp3_bytes > 0) {
            fwrite(mp3_buffer, 1, mp3_bytes, self->outfp);
        }
    }
    free(mp3_buffer);
    Py_RETURN_NONE;
}

static PyObject *Encoder_close(pylame2_EncoderObject *self) {
    int mp3_length;
    char *mp3_buffer;
    int mp3_bytes;
    if (!(self->outfp && self->gfp)) {
        PyErr_SetString(PyExc_Exception, "encoder not open");
        return NULL;
    }
    mp3_length = 7200;
    mp3_buffer = (char *)malloc(mp3_length);
    mp3_bytes = lame_encode_flush(self->gfp, mp3_buffer,
sizeof(mp3_buffer));
    if (mp3_bytes > 0) {
        fwrite(mp3_buffer, 1, mp3_bytes, self->outfp);
    }
    free(mp3_buffer);
    lame_close(self->gfp);
    self->gfp = NULL;
    fclose(self->outfp);
    self->outfp = NULL;
    Py_RETURN_NONE;
}

static PyMethodDef Encoder_methods[] = {
    { "encode", (PyCFunction)Encoder_encode, METH_VARARGS,
      "Encodes and writes data to the output file." },
    { "close", (PyCFunction)Encoder_close, METH_NOARGS,
      "Closes the output file." },
    { NULL, NULL, 0, NULL }
};

static PyTypeObject pylame2_EncoderType = {
    PyObject_HEAD_INIT(NULL)
    0, /* ob_size */
    "pylame2.Encoder", /* tp_name */

```

```

sizeof(pylame2_EncoderObject), /* tp_basicsize */
0, /* tp_itemsize */
(destructor)Encoder_dealloc, /* tp_dealloc */
0, /* tp_print */
0, /* tp_getattr */
0, /* tp_setattr */
0, /* tp_compare */
0, /* tp_repr */
0, /* tp_as_number */
0, /* tp_as_sequence */
0, /* tp_as_mapping */
0, /* tp_hash */
0, /* tp_call */
0, /* tp_str */
0, /* tp_getattro */
0, /* tp_setattro */
0, /* tp_as_buffer */
Py_TPFLAGS_DEFAULT, /* tp_flags */
"My first encoder object.", /* tp_doc */
0, /* tp_traverse */
0, /* tp_clear */
0, /* tp_richcompare */
0, /* tp_weaklistoffset */
0, /* tp_iter */
0, /* tp_iternext */
Encoder_methods, /* tp_methods */
0, /* tp_members */
0, /* tp_getset */
0, /* tp_base */
0, /* tp_dict */
0, /* tp_descr_get */
0, /* tp_descr_set */
0, /* tp_dictoffset */
(initproc)Encoder_init, /* tp_init */
0, /* tp_alloc */
Encoder_new, /* tp_new */
0, /* tp_free */
};

static PyMethodDef pylame2_methods[] = {
    { NULL, NULL, 0, NULL }
};

PyMODINIT_FUNC initypylame2() {
    PyObject *m;
    if (PyType_Ready(&pylame2_EncoderType) < 0) {
        return;
    }
    m = Py_InitModule3("pylame2", pylame2_methods, "My second LAME
module.");
    Py_INCREF(&pylame2_EncoderType);

```

```
PyModule_AddObject(m, "Encoder", (PyObject *)
&pylame2_EncoderType);
}
```

现在，可以将此文件保存为 `pylame2.c`，并对其进行编译。

在 Linux 上：

```
gcc -shared -I/usr/include/python3.1 -I/usr/include/lame pylame2.c \
    -lmp3lame -o pylame2.so
```

在 Windows 上：

```
cl /LD /IC:\Python31\include /IC:\lame-3.98.2\include pylame2.c \
    C:\Python31\libs\python31.lib \
    C:\lame-3.98.2\libmp3lame\Release\libmp3lame.lib \
    C:\lame-3.98.2\mpglib\Release\mpglib.lib
```

一旦完成了编译过程，就可以通过完全用 Python 编写的一个简单的驱动脚本来使用这个新的扩展模块了。

```
import pylame2

INBUFSIZE = 4096

encoder = pylame2.Encoder('test.mp3')
input = file('test.raw', 'rb')
data = input.read(INBUFSIZE)

while data != '':
    encoder.encode(data)
    data = input.read(INBUFSIZE)

input.close()
encoder.close()
```

这就完成了扩展模块的第二个版本。可以从任何位置读取数据。示例驱动脚本仍然从以前创建的原始输入文件中读取数据，但是要从一个 WAV 文件中提取信息或者从一个套接字中读取数据，也是可以的。

此版本模块的唯一不足是不能定制编码数据的写入方式。在该模块的下次修改中将通过向一个对象中“写入”、而不是直接向文件系统中写入来解决这个问题。接下来就介绍这方面的内容。

## 17.7 在 C 代码中使用 Python 对象

Python 是一种动态类型语言，因此虽然我们一直在使用接口，但是它并没有接口的正

式概念。最常见的接口是“文件”接口。类似于“像文件一样的对象”的措辞描述了这种接口。它只不过是“看起来像”文件对象的一个对象。通常，它可以只具有 `read` 或 `write` 方法。

对于扩展模块的下一个版本，当建立一些新的编码器对象时，将允许用户传递任何类似于文件的对象(支持一种 `write` 方法)。编码器对象用一些 MP3 编码的字节调用 `write` 方法。您不必关心它是一个真正的文件对象、一个套接字还是用户能想象出来的其他任何对象。这是多态性的最好体现。

在该模块的上一个版本中，对象保存一个 `FILE *` 指针。需要通过添加一个对 `PyObject` 的引用并删除 `FILE *` 来修改它。

```
typedef struct {
    PyObject_HEAD
    PyObject *outfp;
    lame_global_flags *gfp;
} pylame3_EncoderObject;
```

因为 `Encoder_new` 所完成的所有功能只是将 `outfp` 设置为 `NULL`，所以它可以保持不变。然而需要修改 `Encoder_dealloc`：

```
static void Encoder_dealloc(pylame3_EncoderObject *self) {
    if (self->gfp) {
        lame_close(self->gfp);
    }
    Py_XDECREF(self->outfp);
    self->ob_type->tp_free(self);
}
```

使用 `Py_XDECREF` 宏将引用计数减 1，而不是调用 `fclose`。因为可能有其他对象引用该对象，所以不能删除它。实际上，因为对象来自于此模块的外部，所以很有可能有其他对象引用此对象。不是您创建它，而是别人创建它并将它传递给您。他们可能仍然拥有一个与此对象绑定的变量。

如果这里在 `Encoder_dealloc` 中对引用计数减 1，那么在其他某处必然对其进行加 1。这个操作在 `Encoder_init` 中实现：

```
static int Encoder_init(pylame3_EncoderObject *self,
                        PyObject *args, PyObject *kw) {
    PyObject *outfp;
    if (!PyArg_ParseTuple(args, "O", &outfp)) {
        return -1;
    }
    if (self->outfp || self->gfp) {
        PyErr_SetString(PyExc_Exception, "__init__ already called");
        return -1;
    }
    self->outfp = outfp;
```

```

    Py_INCREF(self->outfp);
    self->gfp = lame_init();
    lame_init_params(self->gfp);
    return 0;
}

```

您已经修改了 `PyArg_ParseTuple` 的格式字符串，使其包含“O”而不是“s”。“O”表示希望接受一个对象指针。对象是什么类型并不重要，我们只是不希望 `PyArg_ParseTuple` 将该对象转换为 C 语言的任何一种基本数据类型。

在确定传入了正确数量的参数而且之前未调用 `__init__` 后，可以存储该对象的参数以供日后使用。这里使用 `Py_INCREF` 宏对引用计数加 1。这样，在对计数减 1 之前，该对象保持活跃。

为什么之前的宏 `Py_XDECREF` 中包含一个 X，而这个宏却没有呢？实际上，这些宏有两种形式。带“X”的版本在调整引用计数之前进行检查，保证指针不为 NULL。不带“X”的版本不做检查，它们更快，但是您必须知道自己在做什么，以便能正确地使用它们。`PyArg_ParseTuple` 的文档表明如果它成功，那么输出指针将是有效的，因此此处使用 `Py_INCREF` 是安全的，但是使用 `Encoder_dealloc` 可能就不那么安全了。

在扩展模块的实现中，保证加 1 与减 1 的完全平衡是最棘手的部分，因此一定要小心。如果它们不平衡，就可能造成内存泄漏，或者可能访问一个已经被删除的对象，这些都绝不是什么好事。

就引用而言，注意所使用的不同 API 函数的文档也是非常重要的。一些函数在返回前将增加引用计数，另一些函数则不会。因为 `PyArg_ParseTuple` 的文档说明它不增加引用计数，所以如果希望它在需要的时间内保持存在，就要对它加 1。

既然已经有了一个对象(也希望有一个 `write` 方法)，就需要使用它。在 `Encoder_encode` 和 `Encoder_close` 中，您希望调用对象的 `write` 方法，而不是调用 `fwrite`。Python API 中有一个 `PyObject_CallMethod` 函数，它将正确地满足这种需要。如下所示是在 `Encoder_encode` 和 `Encoder_close` 中调用对象的 `write` 方法时可以使用的代码片段：

```

PyObject* write_result = PyObject_CallMethod(
    self->outfp, "write", "(s#)",
    mp3_buffer, mp3_bytes);
if (!write_result) {
    free(mp3_buffer);
    return NULL;
}
Py_DECREF(write_result);

```

`PyObject_CallMethod` 需要三个参数。第一个是正调用方法的对象。此对象将是传入该方法的第一个参数，通常被称为 `self`。`PyObject_CallMethod` 的第二个参数是方法的名称。第三个参数是描述参数的格式字符串。如果没有参数，那么它可以是 NULL。当它不为 NULL 时，除了总是用圆括号括起来之外，它看起来非常类似于一个 `PyArg_ParseTuple` 格式字符串。`PyObject_CallMethod` 本质上使用这些参数调用 `Py_BuildValue`，产生的元组将

被传入方法中。

`PyObject_CallMethod` 返回一个 `PyObject *`。所有 `write` 方法的实现都可能返回 `None`，但是您仍然要负责对引用计数进行减 1。

因为 `pylame3.c` 的大部分代码与 `pylame2.c` 相比没有变化，所以这里不会包含整个文件。将本节中描述的这些修改插入到之前的代码中应该不是太难。

一旦对该模块的新版本进行了编译，就可以使用任何一个类似于文件的对象作为 `Encoder` 对象的一个参数。下面的示例演示了这一点：

```
import pylame3

INBUFSIZE = 4096

class MyFile(file):

    def __init__(self, path, mode):
        file.__init__(self, path, mode)
        self.n = 0

    def write(self, s):
        file.write(self, s)
        self.n += 1

output = MyFile('test3.mp3', 'wb')
encoder = pylame3.Encoder(output)
input = file('test.raw', 'rb')

data = input.read(INBUFSIZE)
while data != '':
    encoder.encode(data)
    data = input.read(INBUFSIZE)

input.close()
encoder.close()
output.close()

print('output.write was called %d times' % output.n)
```

此示例包含了一个由内置的文件对象派生的类，以突出您可以完成的那些工作。虽然它没有展示很多功能，但是至少说明了新的扩展模块可以非常灵活。只要传递一个带有 `write` 方法的对象，扩展模块就可以正常工作了。

## 17.8 本章小结

在本章中，您学习了如何创建扩展模块和定义方法表，使 Python 开发人员可以使用 C 语言实现的简单函数。使用 `PyArg_ParseTuple` 实现从 Python 对象到 C 语言的值的转换。

相反，使用 `Py_BuildValue` 将一个 C 语言的值转换成一个 Python 对象。

您还了解到在一个扩展模块中如何通过定义对象和一些类型结构来定义新类型。先创建类型对象，以便它可以创建类型的一些新实例，随后销毁它们。务必对所使用对象的引用计数进行正确地加 1 和减 1，这需要慎重考虑。

当然，关于扩展模块的编写还有其他很多内容，但在一章中没有足够的篇幅逐一介绍。请一定要访问网站 <http://docs.python.org/ext/ext.html> 和 <http://docs.python.org/api/api.html>，查询相关的一些文档。

#### 本章要点：

- 用 C 语言编写的代码可能比用 Python 编写的代码运行快，而用 Python 编写代码比用 C 语言编写代码要快得多，注意这一点很重要。
- Python 扩展模块是一些普通的 C 语言库。在 UNIX 计算机上，这些库通常以 `.so` (表示共享对象) 结尾。在 Windows 计算机上，它们通常以 `.dll` (表示动态链接库) 结尾。
- `distutils` 包允许以一种标准的方式发布 Python 模块。
- `--help-commands` 参数显示一个设置脚本所能响应的所有命令。
- 可以使用 `PyArg_ParseTuple` 和 `PyArg_ParseTupleAndKeywords` 实现从 Python 对象到 C 语言中的值的转换。反向操作需要使用 `Py_BuildValue` 来实现。

## 17.9 习题

1. 向本章前面创建的 `foo` 模块中添加一个新的模块级函数。将该函数命名为 `reverse_tuple` 并实现它，使它接受一个元组作为参数，并返回一个同样大小的元组，其中的元素以倒序排列。完成此题需要进行一些研究，因为您需要了解如何对一个元组进行分解。您已经知道了创建元组的一种方法(使用 `Py_BuildValue`)，但是因为您希望使用此函数处理任意大小的元组，所以本题中不能使用那种方法。Python/C API 文档关于元组的部分(请访问 <http://docs.python.org/api/tupleObjects.html>)列出了完成本题所需要的所有函数。切记一定要小心处理引用计数！
2. 对于几乎所有 Python 应用程序来说，列表和字典对象是重要组成部分，因此学习如何在 C 中操作这些对象将是非常有用的。向 `foo` 模块中添加另一个函数，将其命名为 `dict2list`，它接受一个字典作为参数并返回一个列表。列表的成员应该是字典中的键和值的交替形式。次序并不重要，只要每个键后面跟着其值就可以。您需要找到迭代字典中的所有项的方法(提示：请查阅 `PyDict_Next`)以及创建一个列表并向其末尾追加各个项的方法(提示：请查阅 `PyList_New` 和 `PyList_Append`)。

# 第 18 章

## 数值编程

本章将介绍如何使用 Python 来处理数值。您已经看到过一些算术运算的例子，但是读完本章之后，您将更深刻地了解 Python 语言表示数值的各种方法、进行数学运算的方法以及处理大数值数据集的有效方法。

处理数值的代码在技术软件中处于核心地位，被广泛地应用于科学、工程、金融以及其他相关领域。几乎所有大型程序都会包含大量的数值计算，所以即使您没有在上述相关领域工作，熟悉本章中的一些内容仍会对您有所帮助。例如，如果编写一个分析 Web 日志的脚本，可能需要计算关于 Web 服务器的点击率的统计数据；如果编写一个图形用户界面相关的程序，可能需要一些数学函数去计算图形在图形用户界面中的位置。

本章的部分内容需要事先了解比简单算术更高级的数学知识，如果不知道这些数学知识，可以略过这些内容。本章最后一部分介绍数值数组，这部分内容比本书的大部分内容更加高级，但是如果打算使用 Python 语言处理大量数值，阅读这部分内容将非常重要。

设计用来进行复杂数值计算的软件，既是一门科学，也是一门艺术。复杂数值计算通常被称为数值分析。通常，有多种方法可以进行数值计算，数值分析能够告诉您哪种方法可以得到最接近正确结果的答案。有些情况可能较难处理，特别是涉及到浮点数时，因为浮点数是一个实数的近似值。本章提到了数值精度，但是未深入介绍，因此如果您打算编写进行大量浮点运算的程序，需要考虑学习一本关于数值分析的书来帮助了解可能遇到的问题。

本章将介绍：

- 表示数值的不同数据类型。
- 一些基本的和高级的数学运算符。
- 如何对数组进行数学运算。
- 如何使用数学模型和数组模型。

### 18.1 Python 语言中的数值

像 Python 语言中的其他任何对象一样，数值也有自己的数据类型。Python 语言中有 3 种基本的数值类型。其中 `int` 类型表示整数，`float` 类型表示浮点数。第三种数据类型表示复数，本章稍后将会介绍这种类型。

### 18.1.1 整数

您已经学习过 `int` 类型。如果在程序中写一个类似于“42”这样的普通数值(称为字面值), Python 中将其创建一个 `int` 类型的对象:

```
>>> x = 42
>>> type(x)
<class 'int'>
```

不一定非要显式地构造 `int` 类型, 但如果希望这样做, 可以使用如下代码:

```
>>> x = int(42)
```

您还可以使用 `int` 构造函数来将其他类型的对象(例如字符串或者其他数值类型)转换为 `int` 类型:

```
>>> x = int("17")
>>> y = int(4.8)
>>> print(x, y, x - y)
17 4 13
```

在第一行代码中, Python 把一个表示数值的字符串转换为了数值本身; 您不能使用“17”(字符串)来进行数学运算, 但是使用 17(`int` 类型)则可以。在第二行代码中, Python 把浮点数 4.8 转换为了整数 4, 使用的是截取浮点数的方法, 即把浮点数小数点右边的部分去掉, 使其成为一个整数。

将一个字符串转换为一个整数时, Python 语言假定该数值使用十进制表示。可以为该数值指定另外一种进制。例如, 如果在程序中传递 16, 则认为这个数值使用十六进制表示:

```
>>> hex_number = "a1"
>>> print(int(hex_number, 16))
161
```

可以通过在数值前面加前缀 `0x` 指定一个十六进制数。例如, 十六进制数 `0xa1` 等于十进制数 161。类似地, 以 0 开头的字面值被认为是八进制数, 所以八进制数 `0105` 等于十进制数 69。其他许多程序语言中也使用这种约定。

### 18.1.2 长整数

Python 的 `int` 类型中能存储的最大数值是多少呢? 在 Python 3.0 之前, Python 语言使用 32 位来存储整数, 这意味着使用 `int` 类型存储的最大整数为  $2^{31} - 1$ , 最小整数为  $-2^{31}$ 。如果需要存储更大的整数, Python 语言提供了 `long` 类型, 用于存储非常大的整数。

例如, 在搜索引擎 Google 出现之前很久, 数学家定义了一个 googol, 表示的数值为 1 后面有 100 个 0。要使用 Python 语言表示这个数值, 或者输入 100 个 0, 或者使用指数运算符 `**` 来省去输入 100 个 0 的麻烦。



### 浮点数的精度

浮点数是一个近似值，它只能保证有限数量的精确位数。

Python 语言没有正式保证 float 变量的精确位数。但是在内部，Python 使用 C 语言中的 double 类型存储 float 类型的变量，因此如果知道 C 语言中的 double 类型在一个平台上的精度，就可以知道在相同平台上运行的 Python 语言中的 float 类型的精度。

大部分系统使用 64 位存储 C 语言中的 double 类型，并提供 16 位的精确位数。

### 18.1.4 格式化数值

可以使用 str 构造函数将 Python 语言中的任何数值转换为一个字符串。这样得到的文本可以被 print 语句作为字符串对象打印出来。对于简单应用程序，这样就够了。

如果希望更好地控制输出格式，可以使用 Python 语言内置的字符串格式化运算符：%。

注意：

这和取余运算符没有任何关系。如果在一个字符串后使用%，%为字符串格式化运算符。如果在两个数值中间使用%，%为取余运算符。

以下内容是格式化数值的一些细节。如果熟悉 C 语言中的 printf 函数，您就已经了解了在 Python 语言中格式化数值的大部分语法。

如果要格式化整数，在格式化字符串中使用转换符号%d。如果要格式化浮点数，请使用%f。如果对一个浮点数使用%d 或者对一个整数使用%f，Python 语言会将数值转换为转换符号表示的类型。例如：

```
>>> print("%d" % 100)
100
>>> print("%d" % 101.6)
101
```

您很可能没有注意到，Python 语言使用十进制格式化这些数值，因为这太明显了。对于有些应用程序，可能更希望用十六进制显示输出内容。使用%x 转换符号可以实现这一目的。如果使用 %#x，Python 语言会在输出内容之前加上 0x，使它看起来像一个十六进制字面值，例如：

```
>>> print("%#x" % 100)
0x64
```

类似地，%o(字母 o，不是数值零)可以产生八进制的输出，%#o 可以产生前面带 0 的八进制输出。

对于整数，可以通过在格式化字符串%后面增加一个数值来指定输出内容的宽度(数值的位数)。如果数值以 0 开头，输出内容左边的空余位数会用 0 填满，否则的话会用空格填满。在下面的例子中，输出内容被一对圆括号包围起来，因此您能准确地看到 Python 语言对于%d 转换符号会产生什么结果：



该函数的一个优点在于它首先检查被除数是否为 0，从而避免了除零的错误。仔细观察格式化字符串。第一个%与f一起构成了浮点数的转换符号。最后的%%在输出内容中会被转换为一个%：因为%用于指示转换，如果希望在输出内容中显示%这个字符，Python 语言要求在格式化字符串中连写两个%。

不需要将宽度或者小数位数硬编码到格式化字符串中。如果在转换符号中使用\*替代一个数值，Python 语言从参数元组中一个额外的整型参数(位于被格式化的数值前面)得到\*的值。使用这个特性，可以编写一个格式化美元的函数。它的参数包括美元的数量以及表示美元的数字位数，不包括表示美分的两位数字：

```
>>> def format_dollars(dollars, places):
...     return "$%.2f" % (places + 3, dollars)
...
>>> print(format_dollars(499.98, 5))
$ 499.98
```

在格式化字符串中的浮点转换符号中用\*替代了总宽度。Python 查看参数元组，然后使用第一个参数的值作为转换符号中的总宽度。在这个例子中，指定了比用于表示美元的数字位数多 3 位的数字位数，从而为小数点和用于表示美分的两位数字留下了空间。

还有其他许多使用字符串格式化运算符控制数值输出的方式。详细内容可以参考 Python 文档中位于 Python 库参考下的序列类型部分(因为字符串是一种序列)。

### 18.1.5 作为数值的字符

关于字符有哪些内容呢？使用 C 语言和 C++ 语言的程序员习惯于将字符作为数值进行处理，因为 C 语言中的 char 类型只是另一种整数数值类型。但是 Python 语言并不是这样处理字符的。在 Python 语言中，一个字符只是一个长度为 1 的字符串，不能作为一个数值使用。

有些时候，可能需要在字符和它们代表的数值之间进行转换。Python 提供了内置函数 ord 将一个字符转换为它的数值代码，还提供了内置函数 asc 来将一个数值代码转换为一个字符。数值代码的范围必须在 0 到 255 之间。

**注意：**

严格地说，上述数值代码并非 ASCII 代码，因为 ASCII 代码最高只到 127。尽管如此，ord 函数和 asc 函数转换的前 127 个数字值是 ASCII 代码值。

如果您经常使用 Usenet，那么很可能熟悉 rot13 加密。这种加密方法不是特别安全，它只是将字母从字母表中的位置向前转 13 位，到“z”时下一个转为“a”。使用 Python 中的 chr 函数和 ord 函数实现这一加密方法并不困难：

```
def rot13_character(character):
    # Look up codes for ends of the alphabet.
    a = ord('a')
    z = ord('z')
    A = ord('A')
```

```

Z = ord('Z')

code = ord(character)
# Rotate lower-case characters.
if a <= code <= z:
    code = a + (code - a + 13) % 26
# Rotate upper-case characters.
elif A <= code <= Z:
    code = A + (code - A + 13) % 26
# Leave other characters alone.
else:
    pass
return chr(code)

def rot13(plaintext):
    # Loop over letters in the text.
    ciphertext = ""
    for character in plaintext:
        ciphertext += rot13_character(character)
    return ciphertext

```

这段程序由两个函数组成。第一个函数 `rot13_character` 将 `rot13` 加密方法应用到单个字符上。如果字符是大写或小写字母，则该字符在字母表中向前转 13 个位置得到新值；否则的话，字符保持不变(如果您对取余运算符`%`还不熟悉的话，可以参考下一部分的讲解)。主函数 `rot13` 接受要被加密的消息(程序中的 `plaintext`)，然后旋转消息中的每个字母创造出加密消息(程序中的 `ciphertext`)。

将前面的代码输入到一个模块文件中，将文件命名为 `rot13.py`。在 Python 中，导入该模块，然后尝试如下代码：

```

>>> import rot13
>>> message = rot13.rot13("This is a TOP-SECRET encoded message.")
>>> print(message)
Guvf vf n GBC-FRPERG rapbqrq zrffntr.

```

`rot13` 加密方法的一个优点在于：要解密一条使用 `rot13` 加密的消息，只需要对加密的消息再应用一次 `rot13` 加密即可：

```

>>> print(rot13.rot13(message))
This is a TOP-SECRET encoded message.

```

## 18.2 数学

除了提供一般的算术运算实现，Python 还提供了一些实用的内置数学函数，以及一个提供其他常用函数的 `math` 模块。对算术运算的讲解可能看起来比较简单，但是您也需要理

解一些 Python 语言在处理特定数值类型时的细节。

### 18.2.1 算术运算

Python 语言为数值类型的对象提供了标准的算术运算符，包括+(加法运算符)、-(减法运算符)、\*(乘法运算符)和/(除法运算符)。当使用上述运算符时可以混用不同的数值类型，Python 语言会为运算结果自动选择最具灵活性的类型：

```
>>> i = 10
>>> f = 6.54
>>> print(i + f)
16.54
```

当一个整数 *i* 和一个浮点数 *f* 相加时，Python 语言选择 `float` 类型作为计算结果的类型。这些运算符都有用来更新变量值的特殊形式，即在运算符之后加一个等号。如下代码：

```
>>> total = total + 6
>>> coefficient = coefficient / 2
```

可以简单地写为：

```
>>> total += 6
>>> coefficient /= 2
```

其他运算符参照上述例子即可。

当两个整数相除时，Python 一般使用 `int` 类型表示运算结果。但是当运算结果含有小数部分时，情况就不同了，如下例所示：

```
>>> print 10 / 3
3.33333333333
```

#### 下界除法

Python 语言提供了另外一个除法运算符，称为下界除法，它显式地将商四舍五入为一个整数。下界除法运算符使用 `//` 表示。也可以在 `float` 类型的对象上应用它，例如 `6.6//3.0` 的结果为 `2.0`。

指数运算符 `**` 在前面的例子中已经使用过。它同样也适用于整数和浮点数。下面的函数使用指数运算符计算复利。如果您将 `starting_balance` 这么多的钱存入一个年利率为 `annual_rate` 的银行账户，并等待 `years` 这么多年，下面的函数将返回您能得到的钱数：

```
>>> def compound(starting_balance, annual_rate, years):
...     return starting_balance * ((1 + annual_rate) ** years)
... 
```

将一万美元存在银行一个世纪，每年利率为 4%，最终得到的钱数为：

```
>>> print(compound(10000, 0.04, 100))
505049.481843
```

这是 50 万美元。从现在开始存钱吧。

另外一个有用的运算符是取余运算符%。它与下界除法运算符类似，不同的是它不返回除法的商，而是返回除法的余数。使用取余运算符，可以将一个月份数转换为整年和剩余的月份数：

```
>>> def format_months(months):
...     print("%d years, %d months" % (months // 12, months % 12))
...
>>> format_months(100)
8 years, 4 months
```

### 18.2.2 内置数学函数

一些非常常用的数学函数被作为内置函数提供。最简单的内置数学函数是 `abs` 函数，它返回数值的绝对值。`abs` 函数返回的数值的类型和传递给它的数值的类型完全一致：

```
>>> print(abs(-6.5))
6.5
```

`min` 函数和 `max` 函数也是有用的内置数学函数，它们分别返回一组值中的最小值和最大值。可以使用一组数值参数或者一个序列参数(例如一个列表或者元组)来调用这些函数。这些数值参数不必为同一类型：

```
>>> print(min(6, 7, 2, 8, 5))
2
>>> print(max([0, 43.5, 19, 5, -6]))
43.5
```

`round` 函数可以将一个浮点数四舍五入为指定小数位数的数值。这和之前提到的%f 转换符号的行为类似，不同点在于 `round` 函数的结果不是一个字符串，而是另外一个浮点数，您可以对它继续进行计算。使用该函数时，指定希望四舍五入的数值，以及希望保留的小数位数：

```
>>> print(round(1234.56789, 2))
1234.57
```

甚至可以指定保留的小数位数为负数，这样将从小数点开始向左边四舍五入指定的位数：

```
>>> print(round(1234.56789, -2))
1200.0
```

最后，`sum` 对一个序列中的数值求和。结合 `range` 函数，可以计算出前 100 个正整数的和：

```
>>> print(sum(range(1, 101)))
5050
```

假设您在 Python 程序课的两次家庭作业中分别得到了 96 分和 90 分，在最终项目中得到 100 分，在最终考试中得到 88 分。您的平均分是多少呢？当然，可以写一个 Python 函数来计算它。这个函数使用了 `sum` 函数，计算出一个数值序列的平均值：

```
>>> def mean(numbers):
...     if numbers:
...         return float(sum(numbers)) / len(numbers)
...     else:
...         raise ValueError("no numbers specified")
...
>>> print(mean([96, 90, 100, 88]))
93.5
```

检查数值序列是否为空是个好主意，这样可以避免除零错误。在这个例子中，如果数值序列为空，函数将引发一个异常。

`math` 模块中包括了标准的超越函数。所有这些函数都使用 `float` 类型的参数，并返回 `float` 类型的值：

- 平方根函数：`sqrt`
- 指数函数：`exp`
- 对数函数：`log`(自然对数)，`log10`(以 10 为底的对数)
- 三角函数：`sin`，`cos`，`tan`。参数使用弧度表示
- 反三角函数：`asin`，`acos`，`atan`。函数返回的结果为弧度
- 双曲函数：`sinh`，`cosh`，`tanh`

其他一些有用的数学函数也被包含在内：

- `hypot(x,y)` 等同于 `sqrt(x ** 2 + y ** 2)`。
- `atan2(x,y)` 与 `atan(x,y)` 类似，但是结果取值区间为  $[-\pi/2, \pi/2]$ ，并可以处理分母为 0 的情形。
- `floor` 函数和 `ceil` 函数是标准的下界函数和上界函数，它们返回的结果为整数，但却以 `float` 类型的数值表示。

`math` 程序包也包括数学常量 `pi` 和 `e`。

下面展示一段使用 `math` 模块的示例代码。它会让您想起大一物理课的一些内容。这是一个函数，计算发射到空中的抛射物(如炮弹)的飞行时间和距离，计算时忽略摩擦力。仔细查看这段代码，以理解这段 Python 代码如何执行。注意代码从 `math` 模块中导入了 `sin` 函数、`cos` 函数和数学常量 `pi`，从而使您不必使用 `math.sin` 等来引用它们。这是一个使用常用函数的实用技巧。还要注意在参数中使用的单位以及函数结果被仔细记录了下来。许多火箭发射的失败都和这些实践的重要性有关。

```
from math import sin, cos, pi

def trajectory(velocity, angle):
    """Compute time of flight and range of a projectile.

    For a projectile with initial 'velocity' in meters/sec launched at
    'angle' from horizontal in degrees, returns time of flight in sec
    and range in meters, neglecting friction."""

    # Gravitational acceleration in meters/sec^2.
    g = 9.8
    # Convert 'angle' to radians.
    angle = angle * pi / 180
    # Compute horizontal and vertical components of velocity.
    v_h = velocity * cos(angle)
    v_v = velocity * sin(angle)
    # Compute the time of flight and range.
    tof = 2 * v_v / g
    range = tof * v_h
    return tof, range
```

假设您把一个球按照 45 度的角度和 40 米每秒(大概 90 英里每小时)的速度抛向空中。它会在空中停留多长时间? 它飞多远才会落地? 将前面那段代码保存到一个名为 `ballistic.py` 的文件中, 然后如下面所示调用该函数:

```
>>> from ballistic import trajectory
>>> tof, range = trajectory(40, 45)
>>> print("time of flight: %.1f sec, range: %.0f meters" % (tof, range))
time of flight: 5.8 sec, range: 163 meters
```

## 18.3 复数

复数是实数与虚数的和。回忆一下, 虚数是虚数单位的倍数, 虚数单位是指  $-1$  的平方根。数学家(以及数学教师)通常使用  $i$  表示虚数部分, 而工程师通常使用  $j$ 。

Python 中, 虚数用一个数值以及数值后面的  $j$  来表示(数值和  $j$  之间不能有空格):

```
>>> imaginary_number = 16j
```

要创建一个复数, 将一个实数和一个虚数相加(或者相减):

```
>>> complex_number = 6 + 4j
```

Python 将复数作为单个对象来存储, 使用的类型为 `complex` 类型:

```
>>> print(complex_number)
(6+4j)
```

### 第III部分 开始使用 Python

---

```
>>> print(type(complex_number))
<class 'complex'>
```

如果愿意，可以使用 `complex` 构造函数来构造复数对象。下面的赋值操作与上面的赋值操作效果完全一样：

```
>>> complex_number = complex(6, 4)
```

让我们确定一下 `1j` 就是虚数单位：

```
>>> print(1j ** 2)
(-1+0j)
```

上面的结果证实了 `j` 的平方实际上是 `-1`，而且也表明了涉及到复数的算术运算的结果也一定是复数，即使运算结果刚好是一个实数(即，有一个为 `0` 的虚数部分)。

**注意：**

不能直接用 `j` 来表示虚数单位，而必须使用 `1j`。单独的 `j` 表示名称为 `j` 的变量。

复数对象的实数部分和虚数部分都被存储为浮点数，即使您指定它们为整数。还记得在 Python 中 `1/3` 的返回结果为 `0` 么？对于复数可不是这样：

```
>>> print((1+0j)/3)
(0.333333333333+0j)
```

正如您所期望的，算术运算对于复数同样适用，而且您可以在同一个表达式中混用 `int` 类型、`float` 类型和 `complex` 类型：

```
>>> print(2 * (10 + 3j) * (6.5 - 4j) / (1 - 1j) + 30)
(127.5+56.5j)
```

Python 在处理复数时还提供了其他一些操作。首先，数学运算 `Re` 和 `Im` 分别返回一个复数的实数部分和虚数部分。在 Python 中可以使用每个 `complex` 类型的对象都具有的 `real` 属性和 `imag` 属性得到这两种运算的结果。每个结果都是 `float` 类型：

```
>>> x = 5 - 6j
>>> print(x.real)
5.0
>>> print(x.imag)
-6.0
```

之前提到，内置的 `abs` 函数返回 `int` 类型、`long` 类型或 `double` 类型的对象的绝对值。对于复数，`abs` 函数返回量值(magnitude)，即实数部分与虚数部分的平方和的平方根。可以用之前讨论过的 `hypot` 函数验证这一点：

```
>>> print(abs(x))
7.81024967591
>>> import math
>>> print(math.hypot(x.real, x.imag))
7.81024967591
```

最后，每个 `complex` 类型的对象都有一个名为 `conjugate` 的方法，这个方法返回复数的共轭值。共轭复数指拥有相同实数部分和相反虚数部分的两个复数。`real` 和 `image` 是对象的属性(不调用它们)，而 `conjugate` 是对象的一个方法(必须调用它)：

```
>>> print(x.conjugate())
(5+6j)
```

`math` 程序包中的超越函数只适用于 `float` 类型的对象，返回数据的类型也是 `float` 类型。例如，并不能通过计算  $-1$  的平方根得到 `1j`：

```
>>> print(math.sqrt(-1))
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
ValueError: math domain error
```

这很遗憾，因为在 `math` 模块中的平方根函数以及绝大部分其他函数都是可以被定义在复数上的。幸运的是，Python 语言提供了另外一个称作 `cmath` 的模块，它包含了相同函数的复数版本，即它们应用在复数上并返回复数。这个模块中的平方根函数可以处理  $-1$ ：

```
>>> import cmath
>>> print(cmath.sqrt(-1))
1j
```

### 复数的精度

让我们来验证著名的并且非常基本的数学等式  $e^{i\pi} + 1 = 0$ ：

```
>>> print(cmath.exp(1j * cmath.pi) + 1)
1.22460635382e-016j
```

这是一个实数部分为 0，虚数部分大概是  $1.225 \times 10^{-16}$  的复数。它很接近于 0，但不完全等于 0。

Python 语言将复数的实数部分和虚数部分都存储为相同精度的浮点值，在绝大多数系统上是 16 位的精度。这意味着虽然在 Python 中  $e^{i\pi}$  的表示等于  $-1$ ，但是这个表示只有大概 16 位是精确的。因此，看到  $e^{i\pi}$  加 1 的结果差了大概  $10^{-16}$  也不必惊讶。

## 18.4 数组

您已经学习了如何对单个数值进行数学运算，不管它们是整数、浮点数还是复数。如

果希望对多个数值执行计算呢？一组数值一般被组织成一个数组。本节将介绍在 Python 语言中实现数组的不同方法。

需要记住的是，数组可能是多维的。如果把数值排成一排，将得到一个一维数组。一维数组的一个例子是线性代数中的一个向量，另外一个例子是由某个您喜欢的股票每天的收盘价格构成的列表。也可以将数值放到一个矩形网格中，得到一个二维数组。一个灰阶图像经常用一个二维数组来表示，数组中的每个值表示图像中一个像素的亮度。在一些应用程序中，可能会希望把数值放到更高维度的数组中。

在之前编写 `mean` 函数的时候，已经看到了如何使用 Python 语言构造数组。`mean` 函数接受一个数值序列(序列长度是随意的)，然后计算出这些数值的平均数。可以认为这个数值序列是一个数组，而 `mean` 函数则是一个操作数组的函数。可以使用一组数值来调用这个函数，但是此函数能够处理任意序列类型，包括元组。这些内置类型是创建数组最简单的方式。

### 列表还是元组？

应当使用列表还是元组来创建数组呢？记住，列表可以被修改，而元组不能被修改。因此，如果需要对数组执行增加、删除或改变等操作，应该使用列表。尽管也可以在元组上通过增加、删除或改变数值创建新的元组来实现上述操作，但是这种实现方式的编码要比使用列表困难，而且运行得也更慢。对于固定的数值序列，可以使用元组。

让我们再看一个应用在数组上的函数的例子。前面已经写过一个计算数组平均数的函数。现在来编写一个计算标准差的函数。标准差用于表示数值之间的差别大小。如果所有数值都差不多一样，标准差会比较小，而当数值分布的范围较广时，标准差会比较大。计算标准差的公式如下所示：

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N x_i^2 - \mu^2}$$

其中， $x_1, \dots, x_n$  是数组中的数值， $\mu$  是它们的平均数， $N$  是数组的长度。

可以使用多种方法实现一个计算标准差的函数，这是其中的一种：

```
from math import sqrt

def stddev(numbers):
    n = len(numbers)
    sum = 0
    sum_of_squares = 0
    for number in numbers:
        sum += number
        sum_of_squares += number * number
    return sqrt(sum_of_squares / n - (sum / n) ** 2)
```

这个函数遍历数组中的所有数值，并计算出这些数值的平方和。同时，它也计算出数组中所有数值之和，因为需要这些数值之和来计算它们的平均数。最后一行按照之前的公式计

算出了标准差。注意这个函数在计算数值的平方和时使用了 `number * number` 而不是 `number ** 2`，这是因为使用乘以数值自身的方法计算它的平方比使用通用的指数运算符要快。

请查看 `stddev` 的实际运行情况。记住这个函数只需要一个数值序列(不是多个数值参数)作为参数：

```
>>> print(stddev((5.6, 3.2, -1.0, 0.7)))
2.50137462208
```

现在花点时间来思考使用数值列表作为数组的优点和缺点：

- Python 中列表的元素不必是同一类型。可以创建这样的一个列表，它的元素可能是 `int` 类型、`float` 类型、`long` 类型或者 `double` 类型，也可能是字符串，乃至其他序列类型。对于一些应用程序来说，这一点非常实用。例如，可能要在一个序列中存储 `None` 来表示一个未知的值。对于其他一些应用程序，保证一个数组中的所有值都是同一类型是非常重要的。在这种情况下，需要写额外的代码来保证这一点。
- 列表是一维的，因此利用列表来表示一维数组非常自然。可以使用列表的列表来创建二维数组，以类似方法创建更高维的数组，但这将变得复杂。
- 列表是 Python 的标准部分。它们总是可用(您甚至不需要导入它)，而且您已经知道如何使用它们。
- 列表可以被保存。很容易就可以将列表存储到一个文件中供以后使用。
- 在内部，Python 将列表中的每个元素都表示为一个对象。因此，如果有一个包括 100 万个数值的列表(在许多领域这都很常见)，您就让 Python 记录 1 000 001 个对象：列表本身和它的所有元素。这不但浪费了大量内存，而且当访问或修改这个数组时，Python 也要做很多工作。

最后一点对许多数值类型的相关工作都是一个巨大的限制。为了解决该问题，可以使用其他两种实现数组的方式，这两种方式存储数值的效率更高。

## array 模块

Python 标准库中有一个 `array` 模块，用来表示一维数值数组。这个模块中的 `array` 类型将数组中的所有数值作为一个对象存储在内存中，并且要求所有数值必须是同一数值类型。`array` 模块支持的数值类型和 Python 中的数值类型不一样(实际上，`array` 模块支持的数值类型和 C 语言中的数值类型相同)。`array` 类型的对象可以存储等同于 Python 中的 `int` 类型或 `float` 类型的数值，还可以存储其他大小的整数以及其他精度的浮点数(`array` 类型的对象可以存储 `long` 类型的值，但不能存储非常大的值，而且完全不能存储复数)。

当创建一个数组时，需要指定存储在数组中的数值的数值类型。数值类型可用一个字符指定。如果要存储 Python 中的 `int` 类型的对象，使用 `i`；对于 `float` 类型的对象，使用 `d`(还有其他一些选项，请参考 `array` 模块的文档)。如果不做其他任何指定，则将得到一个空数组：

```
>>> import array
>>> a = array.array("i")
```

## 第III部分 开始使用 Python

```
>>> print(a)
array('l')
```

一般来说，可以像使用一般列表那样使用数组对象。可以插入、追加或删除其中的元素，而且索引的语法也一样(注意在 Python 2.4 之前的版本中，数组对象比列表对象受到更多的限制)。例如：

```
>>> a.append(15)
>>> a.extend([20, 17, 0])
>>> print(a)
array('l', [15, 20, 17, 0])
>>> a[1] = 42
>>> print(a)
array('l', [15, 42, 17, 0])
>>> del a[2]
>>> print(a)
array('l', [15, 42, 0])
```

通过将一个列表或者元组作为参数传递给数组类型的构造函数，可以把它们转换为一个数组对象：

```
>>> t = (5.6, 3.2, -1.0, 0.7)
>>> a = array.array("d", t)
>>> print(a)
array('d', [5.5999999999999996, 3.2000000000000002, -1.0, 0.6999999999999996])
```

这里再次看到了浮点数的近似本质。

实际上，由于数组对象与列表非常类似，可以将这个数组对象作为参数传递给之前编写的 `stddev` 函数，可以发现 `stddev` 运行得很好：

```
>>> print(stddev(a))
2.50137462208
```

如果希望把一个数组类型的对象转换回一个普通的元组或者列表，只需要将这个数组类型的对象传递给 `tuple` 或 `list` 类型的构造函数。

```
>>> back_again = tuple(a)
>>> print(back_again)
(5.5999999999999996, 3.2000000000000002, -1.0, 0.6999999999999996)
```

与列表相比，数组对象有如下优点和缺点：

- 数组对象的所有元素必须是相同类型。
- 与列表相同，数组对象是一维的。
- `array` 模块是 Python 标准库的一部分(但要记着导入这个模块)。
- 数组对象不能自动被存储。

- 数组对象存储值的效率比数值列表高。但是，计算对象里的数值时却并不快多少，这是因为数值计算使用的是 Python 中的标准数值对象。

## 18.5 本章小结

本章介绍了如何使用 Python 实现各种数值运算。您首先了解了 Python 语言的内置整数类型和浮点数类型，并且知道了如何使用 Python 语言的内置算术运算符。然后接触了更高级的数学知识，学习了如何使用 math 模块中的特殊函数和 Python 语言中的复数类型。

最后，本章介绍了两种表示数值数组的方法。最简单的方法是使用数值列表或数值元组。为使存储更有效率，需要使用 Python 语言中的 array 模块。

### 本章要点：

- 与 Python 语言中的其他对象一样，每个数值都有类型。Python 有 3 种基本的数值类型。int 类型表示整数，float 类型表示浮点数。第三种数值类型是复数。
- 可以使用 str 构造函数将 Python 中的任何数值转换为一个字符串。这产生了字符串类型的文本，可以被 print 语句打印出来。
- 要格式化一个整数，在格式化字符串中使用 %d 转换符号。对于浮点数，使用 %f。如果对浮点数使用 %d，或者对整数使用 %f，Python 会将数值转换为转换符号表示的类型。
- Python 为各种数值类型都提供了标准的算术运算符：+(加)、-(减)、\*(乘)、/(除)。
- array 模块中的 array 类型将数组中的数值在内存中存储为一个对象(注意这些数值必须具有相同的类型)。
- math 模块包含这些函数：sqrt(平方根函数)，exp(指数函数)，log/log10(自然对数函数和以 10 为底的对数函数)，sin、cos 和 tan(三角函数)，asin、acos 和 atan(反三角函数)，以及 sinh、cosh 和 tanh(双曲函数)。

## 18.6 习题

1. 编写一个函数，将字节数用千兆字节、兆字节、千字节和字节之和来表示。记住，一个千字节是 1024 个字节，一个兆字节是 1024 个千字节，以此类推。每部分的数值不能超过 1023。函数的输出应当如下所示：

```
>>> print(format_bytes(9876543210))
9 GB + 203 MB + 5 KB + 746 bytes
```

2. 编写一个函数，将 RGB 颜色格式化为 HTML 的颜色语法。该函数接受 3 个参数，分别是红色分量、绿色分量和蓝色分量，每个参数值在 0~1 之间。函数的输出是一个形式为 #RRGGBB 的字符串，其中 RR 是红色分量的值，取值范围为 0~255，用一个两位的十六进制数表示。同样地，GG 和 BB 分别是绿色分量的值和蓝色分

量的值。

例如：

```
>>> print(rgb_to_html(0.0, 0.0, 0.0) # black)
#000000
>>> print(rgb_to_html(1.0, 1.0, 1.0) # white)
#ffffff
>>> print(rgb_to_html(0.8, 0.5, 0.9) # purple)
#cc80e6
```

3. 编写一个名为 **normalize** 的函数，它接受一个浮点数数组作为参数，并返回该数组的一个副本，其中的元素已经按比例进行了缩放，使得所有元素的平方和的平方根为 1。这在线性代数和其他领域中都是一个重要的操作。

下面是一个测试用例：

```
>>> for n in normalize((2.2, 5.6, 4.3, 3.0, 0.5)):
...     print("%.5f" % n,)
...
0.27513 0.70033 0.53775 0.37518 0.06253
```



# 第 19 章

## Django 简介

如果您曾经开发过 Web 应用程序，那么可能知道有时候这是一个非常繁琐的工作。为了缓解这个问题，有一些程序设计语言借助于 Web 应用程序框架(例如 Ruby on Rails 或 Java 的 Spring 框架)来处理所有 Web 应用程序通用的部分，而程序员只需要关注更重要的部分就可以了。

几乎每一种语言都有至少一种(经常有好几种)框架，Python 也不例外。Django 就是基于 Python 构建的标准 Web 应用程序框架，很多 Python 开发人员通过 Django 来快速开发 Web 应用程序。

阅读本章之前，您需要了解一些 Web 开发的基本知识，这样可以更好地理解本章的内容。至少，您需要很熟悉过程式设计，熟悉程序语言中的决策(if 语句和循环)，以及通过列表和哈希表保存数据。如果还不熟悉这些，最好返回本书前面的章节复习一下这些概念。

本章将介绍：

- 框架的定义，以及使用框架的理由。
- 安装最新版本的 Django。
- 理解什么是 MVC/MTV 架构。
- 在 Django 中创建视图和模板。
- 在 Django Web 应用程序中使用数据库。

### 19.1 框架的定义以及使用框架的理由

前面简单提到了 Web 应用程序框架这个词，但是没有解释这个词到底是什么意思。要想完全理解什么是框架，以及为什么要使用框架，必须理解几乎所有 Web 应用程序和使用数据库的 Web 站点的核心基础。

第一个基础是数据库连接。这是几乎所有 Web 应用程序的关键部分。数据库包含很多您甚至都想不到的记录。诸如用户名、用户权限、设置、评论和个人资料等许许多多的信息都存储在数据库中。Django 支持好几种数据库(本章后面详述)，有一些需要使用 SQL，而另一些不需要。

第二个基础是管理面板(administrative panel)。通过管理面板，管理员和其他人可以使

用所有存储在数据库中的数据。例如，如果需要将一个用户的权限从注册用户提升到超级管理员，就需要使用管理面板。

下一项功能是留言。尽管看上去大量无用的用户留言正在侵蚀 Web 的领地，但是留言功能仍然是任何设计良好的 Web 站点的重要功能。留言功能使用户觉得他们是社区的一部分，而不是孤零零地自说自话。

这一类 Web 站点的另外一项重要功能是用身份验证。这种功能控制用户登录的方式，确保安全的登录，并决定用户对 Web 站点拥有的权限等。

这些只是设计良好的 Web 应用程序的一小部分功能而已。可以看出，设计 Web 应用程序需要考虑很多方面，而且需要很多编程工作。而且，事实上这些工作并没有什么意思。我们甚至还没有开始考虑用户界面和设计的问题，而它们才应该是设计 Web 站点时主要关注的方面。

在没有框架的时候，程序员必须手工编写上面列出的所有功能和其他功能的代码，这会耗费大量时间，从而增加产品的成本。我们很幸运，因为框架可以帮我们完成那些无聊的工作。使用 Django 时，以上那些功能，甚至更多的功能，都可以迅速实现。

## 19.2 Web 框架的其他功能

除了前一小节列出的功能之外，框架还提供了以下功能：

- **URL 映射和一些框架(特别是 Django)**——可以解释 URL，这样 URL 对用户更加友好且直观，更重要的是，方便了搜索引擎进行索引。例如，有一个 URL 为 `/mypage.cgi?cat=comic&topic=superman`。URL 映射功能可以将这个 URL 转换为更简单的地址，如 `/mypage/comic/superman`。如果 Web 站点的访问者还想来访问这个页面，他们更有可能记住这个 URL，而且搜索引擎也能更好地理解 Web 站点的底层结构。
- **缓存(caching)**——Web 缓存指的是保存文档副本的过程。当重新访问一个页面的时候，如果满足某种条件，这个页面可以直接从内存中加载，而不用请求一个新页面。因此，缓存可以增加页面加载的速度，提高 Web 站点的整体可用性。
- **模板(templating)**——通过模板可以使整个 Web 站点的风格统一，还能达到其他很多目的。首先，Web 站点看上去很专业。其次，可以确保用户不会感到疑惑，误认为他们已经离开了您的站点。第三，模板可以创造出无缝的体验，这样 Web 站点每一部分的行为都符合预期。例如，如果在一个页面上单击打印按钮可以打印文档，那么在其他所有页面上单击打印按钮都应该有一致的功能，而且打印按钮应该都在同一个位置。模板更重要的功能是可以减少 Web 站点内页面的数目。通过使用“平页面(flat page)”，站点可以访问数据库并获取特定数据，并将这些数据显示在页面上。假设 Web 站点的功能是列出作者的简历，这个 Web 站点可以列出 10 000 个作者的简历。如果采用老办法，可能必须得编写 10 000 个页面，每一个作者一个页面。而使用模板功能，只要创建一个平页面即可，这个页面可以读取数据库，并且填入每个作者的信息。因此，实质上只有一个页面，但是这个页面可以动态变化为很多页面。

Web 框架及其功能的完整介绍超出了本书的讨论范围，但是有了上面的介绍，您应该已经牢固地理解框架能够实现多种可能性。这里提供的信息已经足以让您开始使用框架了。

## 19.3 Django 发展史

Django 中的“d”不发音，后面的发音和 Bang-o 押韵。这个名称来源于吉卜赛爵士吉他手 Django Reinhardt。Django 是由一个称为 World Online 的小部门的程序员们创建的，World Online 是 Kansas 的 Lawrence Journal-World 报纸的 Web 设计部门。

由于经常被迫在最后期限前匆忙交付项目，而且经常被要求迅速编写新的 Web 应用程序，两位开发人员 Adrian Holovaty 和 Simon Willison 开发了一个新的 Web 框架来应付编辑部的要求。两年后，也就是 2005 年，这个小组和 Jacob Kaplan-Moss 一起向开源社区发布了这个框架，并命名为 Django。

由于 Django 最初建立的目的是处理一些在线报刊的需求，所以被贴上了“内容”框架的标签，很多人认为它只适合于出版业而不适合其他行业。当然，事实并不是这样。您很快将发现，Django 和其他框架一样强大和灵活。

有关 Django 的历史以及用途的更多信息，请访问 <http://www.djangoproject.com/>。

### 安装 Django

编写本书的时候，Django 的最新版本为 1.1。与很多库和组件一样，Django 并没有升级到兼容 Python 3.1。不过 Django 可用于 Python 2.6，因此本章剩余部分使用的都是 Python 2.6。如果您还没有安装 Python 2.6，则需要安装该版本。安装方法和 3.1 版本一样。可以在 <http://www.python.org/download/> 上下载安装软件。

为安装 Django，打开 <http://www.djangoproject.com/download/> 页面。建议下载正式版本，避免使用最新的开发版本。如果确实决定使用开发版本，那么可能您知道如何进行安装，因此本章为介绍有关内容，将使用最新的版本。

对于 Windows 用户来说，安装相当容易。下载 .tar.gz 文件，解压到 Python26 文件夹下(位置一般是 C:/Python26)。解压文件后，打开命令提示符(Start Menu | All Programs | Accessories | Command Prompt)，输入下面的命令，将目录切换到 Python26 文件夹：

```
cd C:\Python26\Django-1.1
```

然后输入：

```
setup.py install
```

可以看到程序安装和配置一些包的过程。完成之后，Django 就正式安装好了。安装路径应该类似于 C:\Python26\Lib\site-packages。

下面测试安装。打开 IDLE，输入以下内容：

```
>>>import django
>>>django.VERSION
(1, 1, 0, 'final', 0)
```

要在非 Windows 计算机(例如 Linux、Mac OS X 或其他基于 UNIX 的系统)上安装 Django, 首先下载 .tar.gz 文件, 使用下面的命令进行解压操作:

```
tar xzvf Django-1.0.0-final.tar.gz
```

注意, 需要把上面的 1.0.0 替换为您下载的那个版本。下一步, 切换到文件解压的目录:

```
cd Django-1.0.0
```

其中 1.0.0 是版本号。最后, 在命令提示符下输入 `sudo python setup.py install`, 等待安装结束。测试安装的方法和上面介绍的一样。

## 19.4 理解 Django 的架构

在开始开发第一个 Django 项目之前, 首先简单讲解一下 Web 框架的一个重要方面——架构。大部分框架都采用 MVC 架构, 即模型-视图-控制器(Model-View-Controller)架构。Django 也是如此, 不过有时候会采用 MTV 架构, 即模型-模板-视图(Model-Template- View)架构。笔者认为这两种架构只是在意思上存在区别。

从 Web 应用程序的角度来说, MVC 架构最好按以下方式分解:

- 模型——页面内显示的实际数据或内容, 可以存储在数据库、XML 节点或其他地方。
- 视图——Web 页面, 可以是 HTML、XHTML 或其他标记语言。
- 控制器——代码部分, 负责将数据从模型(数据库/XML)中收集起来并传递给视图(Web 页面)。

尽管示例视图很简单, 但是对于继续学习的目的, 这已经足够了。

### 19.4.1 项目初始设置

由于这是第一次做 Django 项目, 因此首先必须完成一些设置任务。要注意的是, Django 术语中的项目和我们日常生活中的项目并不一样。在 Django 中, 项目指的是一个 Django 实例的设置, 由 Django、特定于应用程序的设置和数据库选项组成。

首先, 创建一个新目录。为此, 切换到 `django-admin.py` 文件所在的目录, 通常这个文件都在 `site-packages\django\bin` 目录下。输入下面的命令可以切换到该目录:

```
cd C:\Python26\Lib\site-packages\django\bin
```

下一步, 运行 `django-admin.py` 文件, 让这个脚本创建一个新的目录, 您的项目和代码都将存放在这个目录内。这里将新目录命名为 `newsite`。在命令提示符下输入以下命令:

```
django-admin.py startproject newsite
```

注意，可以将 `newsite` 随意修改为想要的名称。

`startproject` 是一个命令，不仅创建了新的目录，还在目录中存储 4 个重要的文件。这些文件是：

- `__init__.py`——这个空文件使得创建的目录(在本例中为 `newsite`)成为了一个包。回忆一下，包只不过是一组 Python 模块。
- `manage.py`——这是与 Django 项目交互的命令行实用工具。
- `settings.py`——这个文件存储项目的设置和通用配置。通过 IDLE 打开这个文件时，可以看到如下代码：

```
# Django settings for newsite project.
DEBUG = True
TEMPLATE_DEBUG = DEBUG
ADMINS = (
    # ('Your Name', 'your_email@domain.com'),
)
MANAGERS = ADMINS
DATABASE_ENGINE = 'postgresql_psycopg2', 'postgresql',
'mysql', 'sqlite3' or 'oracle'.
DATABASE_NAME = ''           # Or path to database file if using sqlite3.
DATABASE_USER = ''          # Not used with sqlite3.
DATABASE_PASSWORD = ''      # Not used with sqlite3.
DATABASE_HOST = ''          # Set to empty string for localhost. Not used
with sqlite3.
DATABASE_PORT = ''          # Set to empty string for default. Not used with
sqlite3.

# Local time zone for this installation. Choices can be found here:
# http://en.wikipedia.org/wiki/List_of_tz_zones_by_name
# although not all choices may be available on all operating systems.
# If running in a Windows environment this must be set to the same as your
# system time zone.

TIME_ZONE = 'America/Chicago'

# Language code for this installation. All choices can be found here:
# http://www.i18nguy.com/unicode/language-identifiers.html

LANGUAGE_CODE = 'en-us'

SITE_ID = 1
```

## 第III部分 开始使用 Python

```

# If you set this to False, Django will make some optimizations so as not
# to load the internationalization machinery.
USE_I18N = True

# Absolute path to the directory that holds media.
# Example: "/home/media/media.lawrence.com/"
MEDIA_ROOT = ''

# URL that handles the media served from MEDIA_ROOT. Make sure to use a
# trailing slash if there is a path component (optional in other cases).
# Examples: "http://media.lawrence.com", "http://example.com/media/"
MEDIA_URL = ''

# URL prefix for admin media -- CSS, JavaScript and images. Make sure to use a
# trailing slash.
# Examples: "http://foo.com/media/", "/media/".
ADMIN_MEDIA_PREFIX = '/media/'

# Make this unique, and don't share it with anybody.
SECRET_KEY = 'ms159e^bu=@m&grkl06cl4kq&b058)*b4#0lp69z@xoply4zas'

# List of callables that know how to import templates from various sources.
TEMPLATE_LOADERS = (
    'django.template.loaders.filesystem.load_template_source',
    'django.template.loaders.app_directories.load_template_source',
    # 'django.template.loaders.eggs.load_template_source',
)
MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
)

ROOT_URLCONF = 'newsite.urls'
TEMPLATE_DIRS = (
    # Put strings here, like "/home/html/django_templates" or "C:/www/django/
    # templates".
    # Always use forward slashes, even on Windows.
    # Don't forget to use absolute paths, not relative paths.
)

INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
)

```

稍后会讨论如何修改这个文件。

- `urls.py`——这是存储当前 Django 项目的所有 URL 的文件。

在开始进行第一个项目之前的最后一个步骤是建立开发服务器。通过开发服务器可以测试 Web 站点，观察开发进度，而不需要建立真正的 Web 服务器(向公众开放 Web 站点的时候需要建立真正的 Web 服务器)。

建立开发服务器第一步是切换到 `newsite` 目录：

```
cd C:\Python26\Lib\site-packages\django\bin\newsite
```

接着，运行 `manage.py` 脚本，为其提供 `runserver` 命令。

```
manage.py runserver
```

几秒钟之后，命令窗口内显示如下内容：

```
Django version 1.1, using settings 'newsite.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
...
```

开发服务器已经建立好了，并且在本地运行。可以在浏览器地址栏里输入 `http://127.0.0.1:8000/` 观察运行结果。注意，只能在您自己的计算机上看到这个页面，所以不要指望其他人也可以访问您的页面。

到此准备工作已经全部完成，下面可以开始实际的开发工作了！

### 19.4.2 创建视图

在前面描述模型—视图—控制器架构的时候，您已经了解了什么是视图。简单说来，视图只不过是 HTML 或 XHTML 页面而已。本章的前几个视图例子将在 Python 代码内创建 HTML。这只是出于演示的目的。实际中应该将 HTML 放在模板页面中。不过不用担心，很快就会讨论到相关内容。

#### 试一试

#### 创建一个视图

在这个例子中，您将创建一个非常简单的视图，在用户的浏览器中显示一些文本。新建一个文本文件，或者在 IDLE 中打开一个新窗口，输入下列代码。一定要将文件保存为 `myfirstview.py` 开发到 `newsite` 目录下：

```
from django.http import HttpResponse
def sometext(request):
    mypage= "<html><body><H1>Welcome to My First Page!</H1></body></html>"
    return HttpResponse(mypage)
```

现在，打开浏览器，并输入地址 `http://127.0.0.1:8000/sometext/`。注意，现在应该将 Web

开发服务器运行起来。如果关闭了开发服务器，按照前一小节的步骤重启它。

并没有看到您期待的结果吧？不用担心，您并没有哪一步操作错误。创建 `myfirstview.py` 文件只是第一步。这个文件只是创建了一个函数(即例子中的 `sometext`)，函数中保存一个简单的 Web 页面，其标题为“Welcome to My First Page!”。

为了让页面可以实际显示在 Web 浏览器中，需要让 Django 激活这个页面。这项工作可以通过之前运行 `startproject` 命令的时候自动创建的 `__urls.py__` 文件来完成。

在文本编辑器或 IDLE 中打开 `__urls.py__` 文件，可以看到下面的代码：

```
from django.conf.urls.defaults import *

# Uncomment the next two lines to enable the admin:
# from django.contrib import admin
# admin.autodiscover()

urlpatterns = patterns('',
    # Example:
    # (r'^mysite/', include('mysite.foo.urls')),

    # Uncomment the admin/doc line below and add 'django.contrib.admindocs'
    # to INSTALLED_APPS to enable admin documentation:
    # (r'^admin/doc/', include('django.contrib.admindocs.urls')),

    # Uncomment the next line to enable the admin:
    # (r'^admin/', include(admin.site.urls)),
)
```

现在，这个文件没有执行任何操作。然而对这个文件编辑完成之后，它可以加速显示 `myfirstview.py` 文件。

### 试一试

### 创建一个 URLconf

URLconf 是一种指示 Django 为哪些 URL 使用哪些代码的简单方法。例如，假设有一个 URL 为 `www.mysite.com/purple`，那么 URLconf 可能明确告诉 Django 应该使用文件 `purple.py`。如果 URL 为 `www.mysite.com/spaghetti`，那么 URLconf 可能告诉 Django 使用文件 `spaghetdivview.py` 中的代码。

在继续下去之前，首先删除 `__urls.py__` 文件中的所有内容，并添加以下内容：

```
from django.conf.urls.defaults import *
from newsite.myfirstview import sometext

urlpatterns = patterns('',
    (r'^sometext/$', sometext),
)
```

现在，刷新浏览器(如果关闭了浏览器，只要访问 <http://127.0.0.1:8000/sometext/>即可)，浏览器应该显示一个空白的页面，它只有一个<H1>标题为“Welcome to My First Page!”。恭喜，您成功地创建了第一个视图！

### 示例说明

首先解释视图是如何工作的。首先为了保持一致，应该使用带 `view` 的名称命名视图文件，例如 `myfirstview.py`。这样就可以很方便地在目录中找出哪些文件是视图，哪些不是。

第一步是导入 `django.http` 模块以及模块中的一个类 `HttpResponse`。下一步，通过定义一个名为 `sometext` 的视图函数创建 `HttpResponse` 类的一个实例。这个视图函数接受参数 `request`，这个参数必须是每个视图函数的第一个参数。

最后，通过 `return` 返回 `HttpResponse` 对象，其中填充了一些 HTML 代码。

与此同时，`__urls.py__` 文件首先导入了 `django.conf.urls.defaults` 模块中的所有内容。在这个模块中有一个函数 `patterns`，它的作用是将值保存在 `urlpatterns` 变量中。

接下来，从您编写的模块 `myfirstview` 中导入了函数 `sometext`(`from newsite.myfirstview import sometext`)。之后，向 `urlpatterns` 添加了模式的值，也就是一个元组。这个元组的第一个元素是一个正则表达式，第二个元素为 `sometext` 视图函数。

前面提到，`URLconf` 的作用是告诉 Django 为哪些 URL 执行哪些代码。在前面的例子中，是由下面的代码完成的：

```
urlpatterns = patterns('',
                        ('^sometext/$', sometext),
)
```

这段代码告诉 Django：看到 `/sometext` 这个 URL 的时候，应该使用 `myfirstview` 模块中的 `sometext` 视图函数。

这个例子只是简单介绍了视图和 `URLconf` 的功能，但是作为入门应该是足够了。有关视图的完整描述超出了本书的讨论范围。

## 19.5 使用模板

您现在已经理解了视图和 `URLconf` 的工作方式，可以开始使用模板了。如前所述，在 Python 代码中嵌入 HTML 代码并不是创建视图的最好方式。在这一节，您将学会如何使用模板。

简单地说，模板的作用是保证全站的一致性。使用模板还可以节省时间，还记得前面 10 000 个作者简历的例子吧？

和 HTML 文档类似，模板由很多不同的标签和过滤器组成，标签的数量非常多，以至于不能在这里完全列出。接下来会尽量介绍最重要的标签和过滤器。这些过滤器和标签控制页面的外观，而不管其中到底显示什么数据。可以把它们想象为校服，不同属性的不同学生都穿同样的衣服，因此产生了一种一致性。

下面演示如何使用模板。打开命令提示符，切换到 `newsite` 目录：

```
cd C:\Python26\Lib\site-packages\django\bin\newsite
```

接着，运行下面的命令：

```
manage.py shell
```

这个命令打开了一个交互式解释器。在这里您可能会觉得奇怪，为什么不使用 IDLE 编辑器？这个问题问得好。使用交互式解释器(至少在这里)的原因在于它可以自动设置好 Django 设置文件。如果试图在 IDLE 中运行下面部分中的代码，一定会出错。本章稍后会讲解如何编辑 Django 设置，这样就可以在任何地方编写代码了。不过现在，让 `manage.py Shell` 来完成这项工作吧。

首先，应该导入模板系统。如下所示：

```
>>>from django import template
```

接下来创建一个模板对象。在这里，创建一个保存书名的对象：

```
>>>btitle = template.Template('Your book is titled {{ book }}.')
```

下面的代码将向模板对象提供上下文。

```
>>> a = template.Context({'book': 'American Gods'})
```

在模板对象内有一个方法为 `render()`，这个方法计算合适的上下文内的每一个变量和模板标签，以字符串的方式返回已渲染的模板。代码如下：

```
>>>print btitle.render(a)
Your book is titled American Gods.
```

## 试一试

## 创建一个模板

现在您已经学习了模板系统的基础知识，下面可以开始实践了。在交互式解释器中输入下列代码：

```
>>>from django.template import Template, Context
>>>myfirsttemplate = """<H1>Welcome to {{owner}}'s Library</H1>
...<p>Below you will find a list of {{owner}}'s favorite books</p>
...<br />
...<H3> Book Title: {{books}} Author: {{author}} </H3>"""
>>>a=Template(myfirsttemplate)
>>>b=Context({'owner': 'James Payne', 'books': 'American Gods', 'author':
'Neil Gaimen'})
>>>print a.render(b)
```

```
<H1>Welcome to James Payne's Library</H1>
<p>Below you will find a list of James Payne's favorite books. </p>
<br />
<H3>Book Title: American Gods Author: Neil Gaimen </H3>
```

现在就得到了一个简单的模板。

### 示例说明

尽管代码很少，但是完成的事情却非常多。当然，这也是 Django 的优点之一。为了解释这个过程，我们逐行对代码进行解释。

首先，从 `django.template` 模块导入 `Template` 和 `Context` 类。然后，创建一个名为 `myfirsttemplate` 的变量，用于保存模板的文本。注意，这个变量保存，从第一个 `<H1>` 标签一直到结束的 `</H3>` 标签的所有数据。

接着创建了模板对象，简单地将其命名为 `a`，将 `myfirsttemplate` 的值赋值给 `a`。接下来创建了上下文对象，这个对象将变量名映射到它们的值(即，将 `owner` 映射到 `James Payne`)。最后，打印渲染结果，展示模板内的标签的执行结果。

## 19.6 使用模板和视图

现在您已经学会了如何使用视图、如何修改 `URLconf` 以及如何创建模板，可以将视图和模板结合使用，而不用直接在代码中嵌入 HTML。

在继续学习之前提醒一下：Django 模板并不是 Django 系统的必选部分。Python 本身也提供了一套模板。不过 Django 内置的模板系统非常灵活强大，因此强烈建议使用 Django 内置的模板系统。

### 试一试

### 使用模板和视图

为了保证模板和视图能够在一起正确地运行，需要采取一些步骤。首先需要创建模板。如果还没有创建模板，那么请在 `newsite` 目录下创建一个新文件夹，并且将其命名为 `templates`(注意，并没有强制要求必须将模板目录放在这里。这里这么做只是为了进行演示)。

接下来，新建一个文本文件，输入以下代码，然后将文件保存为 `myfirsttemplate.html`：

```
<H1>Welcome to {{owner}}'s Library</H1><p>Below you will find a list of
{{owner}}'s favorite books</p>
<br />
<H3> Book Title: {{books}} Author: {{author}} </H3>
```

您可能注意到了还没有定义变量(在 Django 中，任何包含在双花括号 `{{}}` 内的标识符都表示变量)。不用担心，很快就会定义变量。

现在已经定义了模板，并且将模板保存在 `newsite/templates` 文件夹中，下一步就是创建一个新的视图(如果已经定义了 `myfirstview.py` 视图，那么只要将其修改为包含以下代码

即可)。将视图命名为 `myfirstview.py`，并输入以下代码：

```
from django.shortcuts import render_to_response
def sometext(request):
    return render_to_response('myfirsttemplate.html',{'owner': 'James Payne',
                                                    'books': 'American Gods',
                                                    'author': 'Neil Gaimen'})
```

这是您初次在这个文件中见到这么多代码。对于初学者，90%的时间都会使用这个方法将模板视图化。当然还有一些其他的方法，但是它们都不够高效。

这段代码首先从 `django.shortcuts` 模块中导入 `render_to_response` 方法。`render_to_response` 是一个很好用的方法，可以创建模板对象、填充上下文(向模板对象中填入值)并进行渲染，使用这个方法可以避免输入前面例子中的很多代码。

导入之后，定义一个新的函数，并且为其传递必需的 `request` 参数。然后返回 `render_to_response` 方法，传入模板的名称(`myfirsttemplate.html`)。传入 `render_to_response` 方法的第二个参数是上下文，在这里向 `owner`、`books` 和 `author` 赋值。第二个参数并不是必需的，但是如果不为其赋值，页面上不会显示那些变量。

现在已经完成了模板和视图的创建，下面还剩下最后一步。现在必须修改 `__settings.py__`，这个文件位于 `newsite` 文件夹中。在 IDLE 或记事本中打开这个文件，应该可以看到以下内容：

```
# Django settings for newsite project.

DEBUG = True
TEMPLATE_DEBUG = DEBUG

ADMINS = (
    # ('Your Name', 'your_email@domain.com'),
)

MANAGERS = ADMINS

DATABASE_ENGINE = ''          # 'postgresql_psycopg2', 'postgresql',
                              # 'mysql', 'sqlite3' or 'oracle'.
DATABASE_NAME = ''           # Or path to database file if using sqlite3.
DATABASE_USER = ''           # Not used with sqlite3.
DATABASE_PASSWORD = ''       # Not used with sqlite3.
DATABASE_HOST = ''           # Set to empty string for localhost. Not used
                              # with sqlite3.
DATABASE_PORT = ''           # Set to empty string for default. Not used with
                              # sqlite3.

# Local time zone for this installation. Choices can be found here:
# http://en.wikipedia.org/wiki/List_of_tz_zones_by_name
# although not all choices may be available on all operating systems.
# If running in a Windows environment this must be set to the same as your
```

```
# system time zone.
TIME_ZONE = 'America/Chicago'

# Language code for this installation. All choices can be found here:
# http://www.i18nguy.com/unicode/language-identifiers.html
LANGUAGE_CODE = 'en-us'

SITE_ID = 1

# If you set this to False, Django will make some optimizations so as not
# to load the internationalization machinery.
USE_I18N = True

# Absolute path to the directory that holds media.
# Example: "/home/media/media.lawrence.com/"
MEDIA_ROOT = ''

# URL that handles the media served from MEDIA_ROOT. Make sure to use a
# trailing slash if there is a path component (optional in other cases).
# Examples: "http://media.lawrence.com", "http://example.com/media/"
MEDIA_URL = ''

# URL prefix for admin media -- CSS, JavaScript and images. Make sure to use a
# trailing slash.
# Examples: "http://foo.com/media/", "/media/".
ADMIN_MEDIA_PREFIX = '/media/'

# Make this unique, and don't share it with anybody.
SECRET_KEY = 'ms159e^bu=@m&grkl06cl4kq&b058)*b4#01p69z@xoply4zas'

# List of callables that know how to import templates from various sources.
TEMPLATE_LOADERS = (
    'django.template.loaders.filesystem.load_template_source',
    'django.template.loaders.app_directories.load_template_source',
    # 'django.template.loaders.eggs.load_template_source',
)

MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
)

ROOT_URLCONF = 'newsite.urls'

TEMPLATE_DIRS = (
    # Put strings here, like "/home/html/django_templates" or
    "C:/www/django/templates".
    # Always use forward slashes, even on Windows.
    # Don't forget to use absolute paths, not relative paths.
```

```
)

INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
)
```

现在只需要关注下面这一小段代码:

```
TEMPLATE_DIRS = (
    # Put strings here, like "/home/html/django_templates" or
    "C:/www/django/templates".
    # Always use forward slashes, even on Windows.
    # Don't forget to use absolute paths, not relative paths.
)
```

`__settings.py` 代码的这一部分告诉 Django 在哪里找到模板。将下列代码添加进文件, 一定要包含插入行末尾的逗号(,), 然后保存文件:

```
TEMPLATE_DIRS = (
    'C:/Python26/Lib/site-packages/django/bin/newsite/templates',
    # Put strings here, like "/home/html/django_templates" or
    "C:/www/django/templates".
    # Always use forward slashes, even on Windows.
    # Don't forget to use absolute paths, not relative paths.
)
```

这一行代码告诉 Django 模板文件在 `newsite/templates` 目录下。

好了, 大功告成了。在交互式解释器中输入 `manage.py runserver`, 再次启动 Django 开发服务器(如果还没有运行的话), 然后就可以查看 Web 页面了。接着, 打开浏览器, 输入地址: `http://127.0.0.1:8000/sometext/`。应该可以看到前面工作的结果了。

这只是 Django 模板的冰山一角。还有很多标签没有涉及到, 而且标签的数量还在不断地增长。不过这是一个好的开头, 您可以以此为基础继续深入了解 Django 的模板机制。

### 19.6.1 模型

前面指出 Django 采用 MVC(模型—视图—控制器)架构。同时也提到, Django 的架构又称为 MTV(模型—模板—视图)架构。Django 社区中大部分人都认为这两种叫法都没有错。不管采用哪种叫法, 现在您已经了解了 Django 架构中的视图和模板部分, 所以只剩下最后一个部分: 模型。

简言之, Django 中的模型描述了数据库中存储的数据。从这个描述可以知道如何访问数据、数据库中的各种关系以及如何验证数据。从另一个角度看, 可以把模型理解为 SQL Create 语句, 只是多了一些定义。这种定义源于 Django 创建表的方式, 它通过 Python 代码而不是 SQL 创建数据表(注意: Django 本身确实使用了 SQL 代码, 但是它返回的数据结

构却是 Python 的数据结构)。

有关这种额外定义的一个例子是使用 SQL 的大部分数据库处理 URL 的方式。SQL 中并没有特殊的数据类型来处理 URL。但是在 Django 中却有。通过这种更高层次的定义能力可以对数据进行更好的控制。

### 19.6.2 创建模型的第一步——配置数据库设置

第 14 章的例子使用的是 sqlite3，所以这里也将使用该数据库。本章使用 Python 2.6，所以不需要安装其他组件。

在前面讲述模板的时候提到，需要修改 `__settings.py__` 文件的部分设置。对于数据库来说，也需要使用这个文件。如果已经关闭了这个文件夹，也没有关系，只要重新切换到 `newsite` 目录即可。

通过记事本或 IDLE 打开这个文件，在代码顶部找到与下面类似的代码：

```
DATABASE_ENGINE = ''          # 'postgresql_psycopg2', 'postgresql',
                              # 'mysql', 'sqlite3' or 'oracle'.
DATABASE_NAME = ''           # Or path to database file if using sqlite3.
DATABASE_USER = ''           # Not used with sqlite3.
DATABASE_PASSWORD = ''       # Not used with sqlite3.
DATABASE_HOST = ''           # Set to empty string for localhost. Not used
                              # with sqlite3.
DATABASE_PORT = ''           # Set to empty string for default. Not used
                              # with sqlite3.
```

这是 `__settings.py__` 文件中配置数据库的代码部分。从每个选项的注释可以看出，如果使用 `sqlite3` 的话，大部分选项都不用修改，这是使用这个数据库的另一个原因。

不过，虽然只需要设置其中的两个选项，但是理解其他选项的作用很有帮助，这样在使用其他数据库程序的时候就知道该怎么做了。

列表中的第一项是 `DATABASE_ENGINE`。通过这个选项告诉 Django 将要使用的数据库引擎。可以从 4 个选项中选择，每一个选项都表示一个需要单独安装的引擎(再次说明，如果使用 `sqlite3`，就不需要去下载引擎)。这 4 个选项分别为 PostgreSQL(版本 1.x 和 2.x)、MySQL、SQLite3 和 Oracle。不管选择哪一个数据库，都需要将这个数据库的名称放在单引号中，例如：

```
DATABASE_ENGINE = 'sqlite3'
```

上面的代码告诉 Django 应该使用 SQLite3 作为数据库引擎。如果想使用 MySQL，那么应该将 `sqlite3` 替换为 `mysql`。如果想使用 PostgreSQL 版本 1.x，应该将其替换为 `postgresql`，以此类推(注意，如果使用版本 2.x 的 PostgreSQL，那么应该替换为 `postgresql_psycopg2`)。

下一个选项是 `DATABASE_NAME`，通过这个选项告诉 Django 将要使用的数据库的名称。因为这里使用的是 SQLite3，因此应该在这里包含您使用的数据库文件的路径，例如：

```
DATABASE_NAME = 'C:/Python26/Lib/site-packages/django/bin/newsite/sample_
database.db'
```

如果不是使用 SQLite3，只要输入数据库文件的名称即可，而不是输入路径：

```
DATABASE_NAME = 'sample_database'
```

下面设置的是 DATABASE\_USER。这一选项告诉 Django 连接数据库的用户。这一设置非常重要，因为不同的数据库用户可以拥有不同的权限。对于 SQLite3 用户来说，这一项留空。对于其他数据库系统，应该这样配置：

```
DATABASE_USER = 'chew_bacca'
```

如果通过用户名登录，可能还需要密码，因此下一项设置为 DATABASE\_PASSWORD。如果使用的是 SQLite3，这一项留空。同样，如果没有使用密码，这一项也留空。如果有密码，按照下面的方式设置：

```
DATABASE_PASSWORD = 'chewie'
```

最后一项设置为 DATABASE\_HOST，这一选项告诉 Django 连接数据库时应该连接哪一台主机。和大多数选项一样，如果使用的是 SQLite3，这一项应该留空。同样，如果数据库主机就是自己的计算机，这一项也应该留空。否则，输入使用的主机名。

这就是配置数据库的全部内容。如果要确认是否正常工作，通过运行 `manage.py Shell` 打开交互式解释器，然后输入下列代码：

```
>>>from django.db import connection
>>>cursor=connection.cursor()
```

如果输入代码时什么都没有发生，不用担心，因为这说明配置正确。如果得到错误消息，说明在 `settings.py` 文件中有一些配置不正确。阅读错误消息，然后返回文件，确保所有选项都设置正确。

## 19.7 创建模型：创建一个应用程序

应用程序(或称为 Django App)是一个 Python 包，包中包含了模型、视图以及其他各种代码。应用程序和 Django 项目不同，Django 项目可以包含一个或多个应用程序，以及那些应用程序的设置。

应用程序和项目的另一个区别在于：并不总是需要创建应用程序。事实上，唯一确实需要创建应用程序的情形(也就是说要求必须创建应用程序)是使用模型或创建数据库驱动的站点的时候。

现在您已经理解了什么是应用程序、应用程序和项目的区别以及何时必须使用应用程序，下面可以创建第一个应用程序了。代码非常简单。只要在命令提示符下输入下面的命

令即可(确认在 `newsite` 目录下):

```
manage.py startapp employees
```

如果什么都没有发生,那么恭喜您,您已经创建了一个应用程序。如果得到了错误消息,则很有可能没有在正确的目录下,或者存在拼写错误。要查看创建的结果,打开 `newsite` 文件夹,应该可以看见新创建的 `employees` 目录。

进入该目录后,可以看到 4 个文件: `__init.py__`、`__models.py__`、`__tests.py__` 和 `__views.py__`。

Django 自动创建了空的模型和视图文件。在接下来的例子中将使用这些文件。

首先打开 `__models.py__` 文件,将其修改为包含如下代码:

```
from django.db import models

class Employer(models.Model):
    name = models.CharField(max_length=50)
    website = models.URLField()
    industry = models.CharField(max_length=50)

class Employee(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    address = models.CharField(max_length=50)
    hire_date = models.DateField()
    email = models.EmailField()
```

这段代码首先从 `django.db` 中导入了 `models`。然后创建了第一个类 `Employer`,这个类继承于 `models` 类。接着,开始创建 `Employer` 类中包含的字段,并分别定义它们的数据类型。现在一定要了解,`Employer` 类实际上相当于数据库中的一个表。

`Employer` 类的第一个字段为 `name`,定义为 `CharField` 类型。`max_length=50` 是参数,告诉 Django 这个字段包含的内容为字符,而且最多包含 50 个字符。

接下来创建了第二个类 `Employee`,其中保存职员信息。一旦完成了所有的字段,这个模型也就创建完成了。

## 使用模型: 安装

刚才创建了一个描述数据形式以及数据类型的数据模型。到现在为止,还没有真正创建数据表。为了创建数据表,需要首先安装它们。

要安装数据表,首先回到 `newsite` 目录,然后打开 `__settings.py__` 文件。找到 `INSTALLED_APPS` 部分。可以看到这里已经列出了 4 个文件。先不要管它们,因为这些文件是默认文件。按照如下代码修改这一部分:

```
INSTALLED_APPS = (
    'newsite.employees',
)
```

下一步，修改 `MIDDLEWARE_CLASSES` 部分，删除其中的默认类(删除的 `InstalledApp` 依赖于它们，因此在创建数据库的时候会出错)：

```
MIDDLEWARE_CLASSES = (
)
```

现在保存文件，并在交互式解释器中运行下面的命令：

```
manage.py validate
```

从命令中可以很明显地看出，这个命令验证模型，并确保正确定义了每个部分。如果得到的消息指出找到 0 个错误，那么一切正常。否则在模型中肯定有定义错误的地方。

接下来让 Django 通过执行一些 SQL 代码来创建数据表。输入下列命令：

```
manage.py syncdb employees
```

这段代码在数据库中创建了数据表(假设它们还不存在)。命令运行的结果如下所示：

```
Creating table employees_employer
Creating table employees_employee
```

输出表明已经在 `employees.db` 数据库中创建了 `employer` 和 `employee` 表。模型安装也就大功告成了！

## 19.8 本章小结

本章简单介绍了 Django。除了可以迅速创建强大的交互式 Web 站点之外，Django 真正的能力在于驱动数据库的能力。尽管本章没有全面地进行讲解(这需要一本与本书差不多规模的书)，但是本章的内容应该足以让您沿着正确地方向前进了。

本章首先讲述了 Django 的背景和历史，这些内容是非常重要的，因为了解了这些之后，就知道了为什么要使用 Django 以及 Django 的优缺点。然后本章从整体上概述了框架，并对 MVC 和 MTV 架构做了基本介绍，这些架构是很多框架的构建基础。

然后，本章深入讲解了安装 Django 和配置安装文件的过程。接着讲解了如何创建 `URLConf`、视图、模板和模型。

### 本章要点：

- Django 是一个在线出版行业的强大工具，但也很好地适用于几乎每一类 Web 应用程序。
- Django 的运行架构为 MVC，即模型—视图—控制器架构；也常常被认为实际上是 MTV 架构，即模型—模板—视图架构。
- 模型是页面中显示的实际数据或内容，存储在数据库、XML 节点或其他地方。

- 视图是 Web 页面，可以是 HTML、XHTML 或其他标记语言。
- 控制器是一部分代码，这些代码从模型(数据库或 XML)中收集数据，并将数据传递给视图(Web 页面)。
- Django 的设置是在 `__settings.py__` 文件中进行配置。在这里可以定义使用的数据库和应用程序等许多内容。
- 应用程序(Django App)是 Python 包，其中包含了模型、视图以及各种其他代码。应用程序和 Django 项目不同，Django 项目可以包含一个或多个应用程序，以及那些应用程序的设置。

## 19.9 习题

1. 配置 `__settings.py__` 文件，尝试使用 Django 支持的各类数据库。
2. 解释 MVC 和 MTV 架构，说明两者的不同。
3. 创建一个模板，展示餐馆的菜单，并将其显示出来。
4. 使用习题 3 中的数据字段，创建一个显示餐馆菜单的模型，并且通过 Django 创建数据库。





# 第20章

## Web 应用程序与 Web 服务

如果您曾经在网上冲浪过，那么可能已经使用过 Web 应用程序进行研究、付账、发送 E-mail，或者从网上商店购物。作为一个程序员，您也许用其他语言编写过 Web 应用程序。如果是这样，您会发现用 Python 编写 Web 应用程序的方式很熟悉，而且更简单。对于初学者，没有比利用 Python 入门更简单的方式了。

当万维网在 1990 年初被发明的时候，它的主要使用者是学生、研究者和科技公司的员工等。在短短的数年里，Web 使得 Internet 变得流行起来并且将它带入到人们的文化中，指定私有在线服务，或者使它们破产。它获得了如此巨大的成功，以至于大家都认为 Web 是 Internet 的同义词，而实际上 Internet 技术比 Web 早出现 20 多年。

我们的文化快速地依赖于 Web，看起来没有必要为用户解释 Web 应用程序与传统的客户端-服务器应用程序或独立的应用程序相比具有极大的优势。Web 应用程序在世界各地几乎都可以访问。在计算机上安装 Web 浏览器后便可以访问全部网络。Web 应用程序使用有限个小组件(widget)向人们展示了一个简单的用户界面。它们(通常)是平台无关的，可用于任何操作系统上的任何 Web 浏览器，包括那些在编写应用程序时还没有被创建出来的 Web 浏览器。

如果还没有编写过自己的 Web 应用程序，那么可能还不了解在 Web 平台上进行开发的好处。在许多方面，对开发人员的好处是对用户的好处的另一面。Web 应用程序不需要发布，用户会主动使用它们。也没有必要发布更新：当更新了服务器上的程序版本时，所有用户都将开始使用新版本。Web 应用程序通常都易于使用，而且其他人可以从自己的 Web 站点链接到一个 Web 应用程序，从而给该应用程序带来流量和快速传播的口碑。作为开发人员，您可以更自由地实验并且控制软件的运行环境。

Web 的长处也是 Python 的长处：灵活性、易用性和包容性。Python 应用程序是在网络时代编写的；一个 Python 爱好者可能在某个晚上产生了一个好想法，第二天就可以将它实现为一个 Web 时尚。

Python 打包了简单实用的模块，用于与 Web 客户端和服务端交互，它们包括 `urllib.parse`、`urllib`、`urllib.request`、`urllib.error`、`cgi`，甚至 `http.server`。许多(一部分人可能会说太多)开源框架也使得构建复杂的 Python Web 应用程序变得简单。Django(第 19 章介绍)、Zope 以及一些其他框架都提供了模板、身份验证、访问控制等功能，使得开发人员可以把主要精力放在编写与众不同的应用程序上。

这是一个宽广的领域，可能是 Python 社区中最活跃的领域，本章将帮助您入门。本章

将介绍如何使用基本的、标准的 Python 模块来创建实用的 Web 应用程序。还将介绍如何创建 Web 服务接口使 Web 应用程序更加有用，这使得用户可以将 Web 应用程序用在他们的程序中。另外，本章也将介绍如何编写脚本来使用流行的 Web 服务，并使自己受益于这些知识。

如果正在阅读本章，您很可能已经使用过 Web 应用程序，或者已经编写过一两个 Web 页面，但是可能并不知道 Web 的设计原理和 Web 应用程序在后台的运行方式。经验丰富的 Web 开发人员可以略过下面的内容，但您可能会发现下面的内容很有吸引力。如果您一直在编写 Web 应用程序，可能还没意识到 Web 实际上实现了一个特殊的架构，牢记该架构会帮助您编写出更加优秀、更加简单的应用程序。

本章将介绍：

- Web 架构的相关知识，包括 REST 的定义以及其背后的深层概念。
- 如何创建和运行简单的 Web 服务器。
- 如何使用 HTTP 请求和响应。
- 如何使用其他 Web 服务。
- 合适的“Web 服务礼仪”。

## 20.1 REST: Web 架构

认为 Web 有架构，这看起来比较奇怪，特别是对于那些在网络刚刚或者已经流行起来的时候开始学习编程的人们来讲更是如此。因为它如此紧密地融入到了人们的日常生活中，所以导致驱动 Web 的设想被人们无视。然而它们确实存在，与之前的任何事物都不同，并且已经融入到一个一致的架构中。Web 架构由它的创始人之一 Roy Fielding 于 2000 年定义。他将 Web 架构称为 REST(Representational State Transfer, 表示状态转移)。本节简单地总结了 REST 最重要的概念，将它们与 HTTP(实现 REST 的协议)的工作机制联系起来，并提供了有关架构的示例，这些架构以不同的方式做出相同的决策。

### REST 资源

Fielding 关于架构风格以及 REST 的论文，可在网址 [www.ics.uci.edu/~fielding/pubs/dissertation/top.htm](http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm) 找到。Chapte 5 描述了 REST。关于 REST 的非正式介绍，可以访问 REST Wiki，网址为 <http://rest.blueoxen.net/>；或者 Wikipedia 的 REST 词条，网址为 <http://en.wikipedia.org/wiki/REST>。

### 20.1.1 REST 的特性

本章的大部分篇幅都在介绍如何最大限度地利用 REST 架构的特性编写应用程序。作为学习这些特性的第一步，这里首先简单地介绍了 REST 的主要特性。

#### 1. 包含相互链接的文档的分布式网络

架构最基础的特性是其用途。如果不以用途为指导，将没有办法区分好的架构与坏的

架构。因此，REST 的第一个特性是它解决的问题：创建一个“分布式超媒体系统”，这是 Fielding 的论文中的叫法。REST 驱动着 Web：一个相互链接的文档组成的网络，这些文档分散在大量位于不同地域、归不同人所有的计算机上。REST 的其他所有特性必须以该用途作为评判标准。

## 2. 客户端-服务器架构

REST 的第二个特性是 REST 架构中参与者的性质。REST 在参与者中定义了客户端-服务器关系，而不是 BT 或者其他文件共享程序定义的对等关系。Web 上的一个文档存储在一个特定的服务器上或者由该特定的服务器生成，并且传递给请求它的客户端。客户端只与服务器通信，服务器也只与它的客户端通信。在 HTTP 协议中，服务器是 Web 服务器，客户端通常是 Web 浏览器。

## 3. 无状态的服务器

REST 的第三个特性是服务器不保存会话状态。客户端发出的每个请求必须包括执行该请求所需要的全部信息。Web 服务器没有必要知道客户端之前发出的请求。由于这个原因，Web 浏览器在发出的每个请求中都要向站点传递 cookie 和身份验证凭据，而不是在每个会话开始时只传递一次。

HTTP 会话的生存时间就是客户端与服务器之间一次来回的事务处理时间：客户端向服务器请求一个文档，服务器发送响应，响应中要么包含请求的文档，要么解释服务器为什么不能够传送该文档。在 FTP 和 SSH 等协议中，客户端与服务器在每个会话中要多次通信，所以服务器端必须保存通信的状态信息，这样可以根据上一次通信的状态理解下一次通信。而 REST 则将这类信息放在客户端上。

许多框架和应用程序都使用 cookie、特殊的 URL 或者其他方法在 HTTP 上构建会话。这样做并没有错，它既不是非法操作，也不是不道德的操作，而且还有一定的好处，但是这样做，应用程序将丧失无状态服务器的优点。用户可能发现无法回退到一个特定的文档，或者在陷于一个错误的状态时无能为力，因为问题出在服务器上。

## 4. 资源

由于 REST 解决的问题是如何管理分布式文档网络，它的存储单元是文档，或者按 REST 的术语讲，是“资源”。根据 REST，一个静态的 Web 页面是资源，由 Web 应用程序动态生成的页面也是资源。Web 上的可以用 Web 浏览器阅读的任何事物都是资源。

每个资源都拥有至少一个唯一标识符，即命名它的字符串，没有任何一个其他资源会有同样的名称。在 HTTP 中，这些标识符就是资源的 URL。资源标识符 <http://www.python.org/> 标识了一个著名的有关 Python 的资源。<http://python.org/> 是同一资源的另外一个标识符。<http://www.google.com/search?q5Python> 是描述一个动态资源的标识符，这个资源由 Web 应用程序根据请求创建。该定制的资源包含了对其他资源的索引，这些资源都在某个方面与 Python(语言或蛇)有关。然而，也不必非要这样处理：例如 WAIS，一项被 Web 取代的技术，将搜索和搜索结果当做一级对象。在 REST 架构中，这些对象只存在于资源和它们的标识符中。

一个不能通过输入地址而到达的 Web 对象在技术上并不是一个 REST 资源，因为它没有标识符。如果只能通过在 Web 浏览器中提交一个表单而得到一个 Web 页面，那么该页面不是资源，它只是表单提交动作的附属品。将 Web 页面变成真正的资源通常是一个好主意。包含同样信息的资源比非资源要更加有用：因为可以为资源添加书签，将资源传递给其他人，自动访问资源，并可以将资源作为输入传递给能够操作资源的脚本。

5. 表示

当利用 Web 浏览器请求资源时，实际得到的是对该资源的表示。在最常见的情形下，每个资源只有一个表示：资源是 Web 服务器上的一个文件，该文件的表示与文件自身完全一致。然而，单个资源也许会有多个表示。一个新闻 Web 站点可能会将每个新闻故事用以下格式提供：HTML 文件、精简后适合打印的 HTML 文件、纯文本文件或 PDF 文件等。

Web 客户端可能会在资源的标识符中选择资源的一种表示(例如，`story.html` 或者 `story.html?printable`)，或者它会简单地告诉服务器它更喜欢什么格式，并让服务器决定哪种表示方法最合适。

20.1.2 REST 操作

我们通常认为 Web 页面就是可以阅读的那些页面，但是我们也在 Web 上执行操作，使用检索页面的工具创建和修改页面。如果拥有 Web 日志，很可能您已经熟悉了如何使用 Web 浏览器创建新的 Web 资源，但是在其他上下文中也会创建资源。当通过 Webmail 应用程序发送邮件时，一个包含了发送的消息的档案页面将被创建。当在网上商店购买了某件商品时，会收到一个收款确认页面，该 Web 站点的其他页面会更改为显示未完成的订单。

检索资源的操作应当是等价的：所发出的请求不会更改资源的内容。资源修改是另外一个操作。除了检索资源，REST 还允许客户端创建、修改和删除服务器的资源(当然是在拥有相应权限的情况下)。客户端通过以某种格式指定资源的表示来创建新资源，通过指定期望的新的表示修改已经存在的资源。该确切的表示格式由 Web 应用程序提供。

HTTP 中的 4 个基本操作通过表 20-1 所示的 4 个命令(动词)实现。

表 20-1

HTTP 动词	用 途
GET	检索资源的表示形式
POST	修改资源使其符合新的表示形式
PUT	依据提供的表示形式创建新资源
DELETE	删除某个已经存在的新资源

20.2 HTTP：REST 的实际应用

尽管 REST 的原理是通用的，但它主要用 HTTP 实现，而 HTTP 协议驱动着互联网。理解 HTTP 工作原理的最好方法是观察它的行为。为此，接下来将编写一个 Web 服务器。

利用 Python 编写 Web 服务器其实很简单。实现最简单的服务器只需要几行代码，因为 Python 带有 Web 服务器，\* 需要做的全部工作就是激活它。

### 试一试

### Python 的 3 行 Web 服务器

将如下代码输入一个名为 EasyWebServer.py 的文件中：

```
#!/usr/bin/python
import http.server
from http.server import HTTPServer
from http.server import BaseHTTPRequestHandler

def run(server_class=HTTPServer, handler_class=BaseHTTPRequestHandler):
    server_address=('',8000)
    httpd=server_class(server_address, handler_class)
    httpd.serve_forever()
if __name__ == '__main__':
    run()
```

运行以上脚本，您将能够通过 <http://localhost:8000> 这个 URL 访问刚刚编写的 Web 服务器。

#### 注意：

如果计算机上已经有另外一个服务器运行于端口 8000 之上，那么在检查端口号时仅需改变脚本中和 URL 中的端口号。

#### 示例说明

该脚本驱动了一个 HTTPServer 对象，它在端口号 8000 监听 HTTP 请求。每次使用 Web 浏览器访问 Web 服务器时，它将会派生一个对象来处理您的请求。该服务器将一直展示页面，直到终止脚本对应的进程而打断它。

当运行该脚本时，运行脚本的目录将变为 REST 可以访问的资源，它的所有文件和子目录也都如此。当使用 Web 浏览器向服务器发送 HTTP 请求来请求这些资源时，服务器将在它的磁盘上查找与请求的资源所对应的文件，并且将它作为 HTTP 响应的一部分传递给您。

将 Web 服务器绑定到特殊的主机名 localhost 将阻止 Internet 上的人或者本地网络的其他人使用您的 Web 服务器(详细信息请参考第 16 章)。然而，您正在使用的计算机上的其他用户可以访问 <http://localhost:8000/> 并看到您提供的任何页面。如果在一台共享机器上运行该脚本，要确保运行脚本的目录不包含您不希望共享的文档。

#### 注意：

当准备好要为 Internet 上的每个人提供 Web 页面和应用程序时，需要将 Web 服务器绑定到一个对外的主机名称或者 IP 地址上。第 16 章包含有关此主题更详细的信息。

## 20.2.1 可见的 Web 服务器

因为您已经编写了自己的 Web 服务器，再编写一个服务器来查看自己的示例 HTTP 请求和响应不会太困难。以下所示是一个称作 `VisibleWebServer.py` 的脚本。它包含 `SimpleHTTPRequestHandler` 的一个子类，它不仅实现了父类的全部功能，还能够获取 HTTP 请求和响应的文本信息，并将它们打印到标准输出。当发出请求时，该脚本向服务器的标准输出打印少量的日志消息。当访问可见的 Web 服务器时，得到所有输出：

```
#!/usr/bin/python
import http.server
from http.server import SimpleHTTPRequestHandler
from http.server import HTTPServer

#The port of your local machine on which you want to run this web
#server. You'll access the web server by visiting,
#e.g. "http://localhost:8000/"

PORT = 8000

class VisibleHTTPRequestHandler(SimpleHTTPRequestHandler):
    """This class acts just like SimpleHTTPRequestHandler, but instead
    of logging only a summary of each hit to standard output, it logs
    the full HTTP request and response."""

    def log_request(self, code='-', size='-'):
        """Logs a request in great detail. This method is called by
        SimpleHTTPRequestHandler.do_GET()."""
        print(self._heading("HTTP Request"))
        #First, print the resource identifier and desired operation.
        print(self.raw_requestline,)
        #Second, print the request metadata
        for header, value in self.headers.items():
            print(header + ":", value)

    def do_GET(self, method='GET'):
        """Handles a GET request the same way as
        SimpleHTTPRequestHandler, but also prints the full text of the
        response to standard output."""
        #Replace the file object being used to output response with a
        #shim that copies all outgoing data into a place we can see
        #later. Then, give the actual work of handling the request to
        #SimpleHTTPRequestHandler.
        self.wfile = FileWrapper(self.wfile)
        SimpleHTTPRequestHandler.do_GET(self)
        #By this time, the shim file object we created previously is
        #full of the response data, and is ready to be displayed. The
        #request has also been displayed, since it was logged by
        #log_request() (called by SimpleHTTPRequestHandler's do_GET)
        print("")
```

```

        print(self._heading("HTTP Response"))
        print(self.wfile)

    def _heading(self, s):
        """This helper method formats a header string so it stands out
        from the data beneath it."""
        line = '=' * len(s)
        return line + '\n' + s + '\n' + line

class FileWrapper:
    """This class wraps a file object, such that everything written to
    the file is also silently appended to a buffer that can be printed
    out later."""

    def __init__(self, wfile):
        """wfile is the file object to which the response is being
        written, and which this class silently replaces."""
        self.wfile = wfile
        self.contents = []

    def __getattr__(self, key):
        """If someone tries and fails to get an attribute of this
        object, they're probably trying to use it as the file object
        it replaces. Delegate to that object."""
        return getattr(self.wfile, key)

    def write(self, s):
        """Write a string to the 'real' file and also append it to the
        list of strings intended for later viewing."""
        self.contents.append(s)
        self.wfile.write(s)

    def __str__(self):
        """Returns the output so far as a string."""
        return ''.join(self.contents)

if __name__ == '__main__':
    httpd = HTTPServer(('localhost', PORT), VisibleHTTPRequestHandler)
    httpd.serve_forever()

```

注意，尽管 `SimpleHTTPRequestHandler` 并没有被设计为输出可被截获，将它的输出文件替换为可以满足需求的另外一个输出文件并不是非常困难的事情。Python 的运算符重载功能使得一个对象模拟另外一个对象变得容易。在下面的练习中，您使用了上述脚本，并且考虑了如何处理一个示例请求和响应。

### 试一试

### 查看 HTTP 请求和响应

在放置 `VisibleWebServer.py` 文件的目录中创建文件 `hello.html`。将下面的 `html` 代码输

入文件：

```
<html>
  <body>Hello, world!</body>
</html>
```

启动 VisibleWebServer.py，并且使用 Web 浏览器浏览这个 URL：<http://localhost:8000/hello.html>。在 VisibleWebServer.py 进程的标准输出中，应当会看到类似于下面的输出：

```
=====
HTTP Request
=====
b'GET /testpage.html HTTP/1.1\r\n'
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
application/x-ms-application, application/vnd.ms-xpsdocument,
application/xaml+xml, application/x-ms-xbap, application/msword,
application/vnd.ms-excel, application/vnd.ms-powerpoint,
application/x-shockwave-flash, */*
Accept-Language: en-us
UA-CPU: x86
Accept-Encoding: gzip, deflate
User-Agent: Mozilla (compatible; MSIE 7.0; Windows NT 6.0; GTB6; SLCC1; .NET
CLR 2.0.50727; Media Center PC 5.0; InfoPath.1; InfoPath.2; .NET CLR
3.5.30729; .NET CLR 3.0.30618)
Host: localhost:8000
Connection: Keep-Alive

=====
HTTP Response
=====
HTTP/1.0 200 OK
Server: SimpleHTTP/0.6 Python/3.1.1
Date: Thu, 24 Sep 2009 00:47:25 GMT
Content-type: text/html
Content-Length: 42

<html>
  <body>Hello, world!</body>
</html>
```

### 示例说明

请求 hello.html 时，脚本 VisibleWebServer.py 创建的 HTTPServer 对象将派生出一个 VisibleHTTPRequestHandler 对象来处理您的请求。它不仅实现了 EasyWebServer.py 派生的 SimpleHTTPRequestHandler 对象的全部功能，还能确保将 HTTP 请求和响应的全部文本打印到标准输出上。SimpleHTTPRequestHandler 只打印请求的摘要。

如果使用 FireFox 或者 Mozilla Web 浏览器，可以安装一个扩展，以便查看每个 HTTP

请求和得到的响应。这个扩展称作 LiveHTTPHeaders, 可以从 <http://livehttpheaders.mozdev.org/> 上下载。这个扩展对于调试 Web 应用程序非常有用, 但是您只能看到请求和响应的头部信息, 而不是实际的请求或者响应数据。

存在若干 Web 应用程序, 可以帮助发送 HTTP 请求并显示请求和响应。这种类型的应用程序中功能最全的是 Web-Sniffer, 其网址为 <http://web-sniffer.net/>。

### 20.2.2 HTTP 请求

一个 HTTP 请求包括两部分。请求的第一行是条命令, 它包含 HTTP 动词、一个资源标识符以及(可选的)正在使用的 HTTP 的版本号:

```
GET /hello.html HTTP/1.1
```

此处的动词是 GET, 资源标识符是 /hello.html。

HTTP 请求的第二部分是一系列头: 描述客户端及提供关于请求的额外信息的键值对:

```
host: localhost:8000
accept-language: en
accept-encoding: gzip, compress
accept: text/*, */*;q=0.01
```

在 REST 架构中, 标识资源的所有必要信息都放置在标识符中。由于 SimpleHTTPServer 只提供静态文件, 所以使用 /foo.html 唯一标识磁盘中的一个文件。另外一个 Web 服务器可以动态地为 /foo.html 生成一个表示, 而不是只在磁盘上查找文件, 但是 /foo.html 仍将标识一个特定的资源。

尽管标识符应当完全标识资源, 键值对可用来小范围决策应该显示资源的哪种表示形式, 例如, 作为对 Accept-Language 头的响应, 发送文档的本地化版本。HTTP 头还用来控制缓存, 以及发送持续的客户端状态(即 cookie)和身份验证信息。

注意:

Web 浏览器通常以首字母大写的名称, 例如 “User-Agent”, 发送 HTTP 头部, 这也是本章引用特定头部的方式。SimpleHTTPRequestHandler 类的另外一个奇怪的特点是可见的 Web 服务器以小写的方式打印头部名称, 即使在接收时它们并不是小写的, 但是这并没有关系: HTTP 头部是不区分大小写的。User-Agent 和 user-agent 代表同一个头。

### 20.2.3 HTTP 响应

HTTP 响应说明 Web 服务器如何试图满足对应的请求。它以状态代码开始, 状态代码指出了响应情况:

```
HTTP/1.1 200 OK
```

在这种情形下, 响应代码是 200(OK), 即一切正常, 资源也被包含了进来。在 Web 浏

览过程中可能会看到下面所示的一些状态编码，尽管您不会希望看到它们：

- 403(禁止)。即资源也许存在也许不存在，但是无论哪种情况，您都无法接收它。
- 404(文件未找到)。最著名的 HTTP 状态编码，在浏览器中会看到这样的状态编码，意味着资源被移动到其他地方，并且没留下可以链接过去的地址，或者资源从未在此地址出现过。
- 500(内容服务器错误)。Web 应用程序中的错误通常会引起该状态。

注意：

RFC 2616 定义了全部 40 个标准错误代码，并对它们进行了分类，在 [www.w3.org/Protocols/rfc2616/rfc2616-sec10.html](http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html) 上可以获取 RFC 2616。其中的一些编码非常令人费解，但是花点时间了解它们非常值得。例如，响应编码 204 表示“没有内容”，可用于 Web 应用程序，当用户单击一个链接时，它做出响应，而不会使用户的浏览器加载另外一个页面。

下面的状态编码是一些 HTTP 响应头，其格式与 HTTP 请求的头部的键值对格式相同：

```
Server: SimpleHTTP/0.6 Python/3.1.0
Date: Thu, 24 Sep 2009 00:47:25 GMT
Content-type: text/html
Content-Length: 42
```

如同请求的头部包含可能对 Web 服务器有用的信息，响应的头部包含可能对 Web 浏览器有用的信息。到目前为止，最重要的 HTTP 响应头是 Content-Type。如果没有这个头信息，Web 浏览器将不知道如何显示发送给它的文档。/foo.html 的内容类型是 text/html，它告诉浏览器将接收到的文档显示为 HTML 格式。如果客户端请求的是/foo.jpg，内容类型将是 image/jpeg，浏览器就知道要将该文档表示为一个图片。

响应的头部后面有一个空行，其余部分包含收到的文档(如果有的话)。对于一个成功的 GET 请求而言，文档即是被请求的资源。对于一个成功的 POST、PUT 或者 DELETE 请求而言，结果文档通常是经过修改的资源的新版本，或者是描述操作成功的状态消息。一个失败的操作通常会返回一个 HTTP 响应，该响应包含一个描述错误信息并且提供帮助的文档。

注意：

Web 应用程序被认为或多或少地具备 REST 的特点，具体程序取决于它们使用 HTTP 特性的程度。在这一方面不存在必须遵守的规则，有时候方便比满足 REST 特性还要好，但是 HTTP 有一些约定，您可以使用它们，而不必自己创建它们。下面列出了一些设计满足 REST 特性的接口的经验：

保持资源标识符透明。用户应该能够一眼就看出某个资源标识符标识的资源的种类。实现该目标最大的挑战在于如何设计出包含可以唯一标识资源的全部信息的标识符。

另一方面，不要在资源标识符中放置不能帮助识别该资源的额外信息。要求用户在 HTTP 头部，或者在 POST、DELETE 或 PUT 请求的数据中提供这些信息。

如果某条信息出现在标准的 HTTP 头中是有意义的，那么不要将它放在 POST、DELETE

或者 PUT 请求的数据中。例如，身份验证信息可以通过 HTTP 身份验证头提交。如果一个资源有多种可用的格式，可以让客户端使用 HTTP 头 Accept 指定他们需要的格式。

除非确实没有 HTTP 错误代码可以表示问题，否则不要在发生错误时返回状态代码 200(OK)。500(服务器端问题)和 400(用户端问题)是通用的错误代码。这个规则的一个问题是，类似于 IE 的浏览器在收到 200 之外的错误代码时，会显示它们自己的通用的错误界面，这会阻止显示您生成的帮助用户解决特定问题的文档。

## 20.3 CGI：将脚本变为 Web 应用程序

使用不同的 Web 浏览器和资源，对可见的 Web 服务器进行实验，直到感觉厌烦为止。除非您觉得整个主题都很乏味，这种厌倦感可能说明您将从检查 HTTP 请求和响应学到的知识发挥到了极致。幸好很快就会变得有趣：下一个阶段将是关于 Web 应用程序的充满活力的世界。

当仅提供硬盘上的文件时，REST 很容易实现，但是这只涉及 REST 中关于资源请求的部分。资源的表示方法，即创建、修改和删除资源的方式，并没有包括进来。尽管一组静态的 HTML 文件在技术上也叫 Web 应用程序，但是并不是一个非常有趣的 Web 应用程序。

尽管可以以多种方式处理资源表示方式之间的转移以及动态应用程序的创建(记起来 Django 那一章的内容了吗？)，但是公共网关接口(Common Gateway Interface, CGI)是令人尊敬的标准。CGI 开发于上世纪 90 年代初，并且自从创建以来几乎没有改变。CGI 的目标是使得人们可以编写从 HTTP 请求调用的脚本，而不必事先了解 Web 服务器编程的任何知识。支持 CGI 的 Web 服务器可以将一定的 HTTP 请求转换为脚本调用。

注意：

CGI 标准位于 <http://hoohoo.ncsa.uiuc.edu/cgi/> 上。该页面从 1996 年起就没有更新过，这是因为 CGI 也同样没有改变。

由于 CGI 在 Web 服务器内部实现，必须通过配置 Web 服务器使其可用。CGI 的设置极度依赖于 Web 服务器的类型，以及系统管理员关于如何管理系统的想法。Linux 的不同发布版本设置 CGI 的方式都不尽相同。这里既没有对所有可能的情况都给予详尽的介绍，也没有回避所有问题而简单地假设您知道如何进行配置。下面几行 Python 代码实现了一个简单的 CGI 服务器，将这些代码存入文件 EasyCGIServer.py 中。该服务器可用于本章中的所有 CGI 示例。同样，Python 内置的模块使其变得更简单。

```
#!/usr/bin/python
import http.server
from http.server import HTTPServer
from http.server import CGIHTTPRequestHandler

def run(server_class=HTTPServer, handler_class=CGIHTTPRequestHandler):
    server_address=('',8001)
    httpd=server_class(server_address, handler_class)
```

```

    httpd.serve_forever()
if __name__ == '__main__':
    run()

```

上述代码与 EasyWebServer 的代码同样简单, 实际上, 它们几乎是相同的。EasyCGIServer 提供了一个新特性, 该特性对 cgi-bin 目录进行特殊处理, CGI 脚本就保存在该目录中。

### 试一试

### 运行 CGI 脚本

在文件 EasyWebServer.py 和 EasyCGIServer.py 所在的目录下创建子目录 cgi-bin。将下面的代码输入到文件 cgi-bin/hello.cgi 中:

```

#!/usr/bin/python
Print("Content-type: text/plain\n")
Print("Hello, world!")

```

#### 注意:

本章中所有 CGI 脚本的名称都以.cgi 作为扩展名。这将显式地把 CGI 脚本和常规的 Python 脚本区分开来, 并使得在只执行以.cgi 为扩展名的 CGI 脚本的 Web 服务器上运行它们成为可能。

如果位于类 UNIX 系统上, 还需要利用 chmod 命令使 hello.cgi 变得可被编辑:

```
#chmod u+x ./cgi-bin/hello.cgi
```

从命令行运行 hello.cgi 以确保脚本可以正常工作:

```

# ./cgi-bin/hello.cgi
Content-type: text/plain

Hello, world!

```

启动 EasyWebServer.py, 并使用 Web 浏览器访问 <http://localhost:8001/cgi-bin/hello.cgi>。Web 浏览器要么会请求您将 hello.cgi 当做普通的 python 脚本下载, 要么会将 hello.cgi 的源代码以纯文本格式显示:

```

#!/usr/bin/python
print "Content-type: text/plain\n"
print "Hello, world!"

```

终止 EasyWebServer.py, 并启动 EasyCGIServer.py。在 Web 浏览器中, 重新加载 <http://localhost:8001/cgi-bin/hello.cgi>。您将在 Web 浏览器中看到以纯文本格式显示的字符串“Hello,world!”:

```
Hello, world!
```

### 示例说明

当通过 EasyWebServer 请求/cgi-bin/hello.cgi 时，服务器以 EasyWebServer 解释每个请求的方式解释该请求，即认为它请求磁盘上的静态文件。您接收到的是静态文件/cgi-bin/hello.cgi 的内容。

当通过 EasyCGIServer 请求同一个资源时，服务器会以不同的方式解释它。EasyCGIServer 不是将 hello.cgi 当做一个要读取的文件，而是将它当做要运行的一个脚本。该脚本如同是从命令行被执行一样，它的输出被用来创建 HTTP 响应。您在 Web 浏览器中看到的是 HTTP 响应的内容部分，这些内容根据脚本提供的 Content-Type 头显示。任何放置到 cgi-bin/目录中的可执行的.py 或者.cgi 脚本在被请求时，都会被 EasyWebServer 运行，它们的输出被用来创建 HTTP 响应。

### 20.3.1 Web 服务器与 CGI 脚本的协议

CGI 标准指定了一个协议，它是支持 CGI 标准的 Web 服务器与服务器解释为 CGI 脚本的文件之间的协议。Web 服务器负责接收和解析 HTTP 请求，将请求定位到正确的脚本，然后执行该脚本，如同从命令行执行 Python 脚本一样。它还负责修改脚本的运行时环境，将 CGI 特定的变量包括进来，这些变量的值与运行时环境信息以及 HTTP 请求中可以找到的信息对应。例如，User-Agent 头变成了环境变量 HTTP\_USER\_AGENT，该请求调用的 HTTP 动词变为环境变量 HTTP\_METHOD。如同其他的环境变量，这些特殊的变量可通过字典 os.environ 访问，脚本可以使用它们评估 HTTP 请求。

在该服务中，CGI 脚本应该在 HTTP 会话期间接管 Web 服务器的职责。脚本写到标准输出的任何内容被当做 HTTP 响应的一部分输出。这意味着除了产生某种类型的文档外，脚本还需要将任何必要的 HTTP 头部作为文档的引语部分输出。至少，每个 CGI 脚本必须输出 Content-Type HTTP 头部。

#### 注意：

如果让脚本通过 Web 浏览器工作存在困难，可以尝试手动设置合适的 CGI 环境变量，并且从命令行执行该脚本。

### 20.3.2 CGI 的特殊环境变量

您的脚本可能会发现在它的运行环境中 20 多个特殊的 CGI 变量。一些重要的变量稍后会做介绍，现在先看一个简单的 CGI 脚本，它提供了探索环境变量所需的工具。该脚本称作 PrintEnvironment.cgi:

```
#!/usr/bin/python

import os
import cgitb
cgitb.enable()
```

模块 cgitb 会在 Web 浏览器中显示异常报告和栈跟踪，与命令行 Python 脚本抛出异常时显示的内容类似。这可以避免让您看到令人困惑的错误代码 500，并且也不需要查看 Web

服务器日志来找到实际的错误消息:

```
#Following is a list of the environment variables defined by the CGI
#standard. In addition to these 17 predefined variables, each HTTP
#header in the request has a corresponding variable whose name begins
#with "HTTP_". For instance, the value of the "User-Agent" header is
#kept in "HTTP_USER_AGENT".
CGI_ENVIRONMENT_KEYS = [ 'SERVER_SOFTWARE',
                          'SERVER_NAME',
                          'GATEWAY_INTERFACE',
                          'SERVER_PROTOCOL',
                          'SERVER_PORT',
                          'REQUEST_METHOD',
                          'PATH_INFO',
                          'PATH_TRANSLATED',
                          'SCRIPT_NAME',
                          'QUERY_STRING',
                          'REMOTE_HOST',
                          'REMOTE_ADDR',
                          'AUTH_TYPE',
                          'REMOTE_USER',
                          'REMOTE_IDENT',
                          'CONTENT_TYPE',
                          'CONTENT_LENGTH' ]

#First print the response headers. The only one we need is Content-type.
print("Content-type: text/plain\n")

#Next, print the environment variables and their values.
print("Here are the headers for the request you just made:")
for key, value in os.environ.items():
    if key.find('HTTP_') == 0 or key in CGI_ENVIRONMENT_KEYS:
        print(key, "=", value)
```

将文件放入目录 `cgi-bin/` 中, 使它可以执行, 并且访问 `http://localhost:8000/cgi-bin/PrintEnvironment.cgi`。您会看到类似于下面的输出:

```
Here are the headers for the request you just made:
SERVER_SOFTWARE => SimpleHTTP/0.6 Python/3.1.0
REQUEST_METHOD => GET
PATH_INFO =>
SERVER_PROTOCOL => HTTP/1.1
QUERY_STRING =>
CONTENT_LENGTH =>
SERVER_NAME => rubberfish
PATH_TRANSLATED => /home/jamesp/LearningPython/listings
SERVER_PORT => 8001
CONTENT_TYPE => text/plain
HTTP_USER_AGENT =>
```

```
HTTP_ACCEPT => text/html, text/plain, text/rtf, text/*, */*;q=0.01

GATEWAY_INTERFACE => CGI/1.1
SCRIPT_NAME => /cgi-bin/PrintEnvironment.py
REMOTE_ADDR => 127.0.0.1
REMOTE_HOST => rubberfish
```

文件 `PrintEnvironment.py` 定义了一个标识符为 `http://localhost:8000/cgi-bin/PrintEnvironment.cgi` 的资源。当运行 `EasyCGIServer` 时，该资源定义于运行 `PrintEnvironment.cgi` 中的 Python 代码时得到的输出中，并且取决于请求的内容，每次访问该 URL 时得到的内容都不同。

`PrintEnvironment.cgi` 对已定义的 CGI 环境变量进行枚举，并且只打印出这些变量的值。这样做的目的有两个：一是将信息放到您可以看到的地方；二是避免泄露可能包含在其他不相关环境变量中的信息。

`EasyCGIServer` 继承了运行它的 Shell 的环境，意味着如果运行 `EasyCGIServer`，而不是 Apache 或者其他 Web 服务器，负责打印整个环境的 `PrintEnvironment.cgi` 版本将打印出 `PATH` 以及 Shell 中其他所有的环境变量。该信息可能会淹没合法的 CGI 变量，并且可能会暴露账户的敏感信息。记住，在自己的计算机上设置的任何 Web 服务器都可以被同一台计算机上的其他用户访问，并且还可能被 Internet 上的其他人访问。在没有必要的情况下不要暴露自己的私人信息。

这里有必要进一步研究几个特定的 CGI 环境变量：

- `REQUEST_METHOD` 是 HTTP 动词，对应于对该资源使用的 REST 方法。由于正在尝试检索资源的表示方式，您需要 HTTP 动词 `GET`。
- `QUERY_STRING` 和 `PATH_INFO` 是资源标识符在 CGI 脚本中的两种主要的出现方式。可以以不同的方式访问 `PrintEnvironment.cgi`，并对这两个变量进行实验。例如，使用 `GET` 获取资源标识符 `/cgi-bin/PrintEnvironment.cgi/pathInfo/?queryString` 将 `PATH_INFO` 设置为 `pathInfo/`，将 `QUERY_STRING` 设置为 `queryString`。在使用 Web 应用程序时经常看到的奇怪而难以理解的 URL 通常是长的 `QUERY_STRING`。
- `HTTP_USER_AGENT` 是 Web 浏览器提供的字符串，用于访问页面，它对应于“User-Agent”HTTP 头，并且应该标识出正在使用的 Web 浏览器。“User-Agent”是 HTTP 头被转换为 CGI 环境变量的一个有趣的例子。另外一个类似的变量是 `HTTP_REFERER`，它由 HTTP 头 `Referer` 而来。每次单击从一个页面跳转到下一个页面的链接时，HTTP 都会提供 `Referer` 头，这样第二个页面就会了解您是如何访问它的。

### 20.3.3 通过 HTML 表单接收用户输入

充分控制 `PrintEnvironment.cgi` 的输出可以证明它能够提供动态资源，但是连接到它的接口并不十分友好。为了获取不同的文本，必须使用不同的 Web 浏览器，不停地篡改 URL（即请求不同的资源），甚至做更加奇怪的事情。大多数 Web 应用程序都避免使用这种类型的接口，而倾向于选择基于 HTML 表单的接口。可以通过编写能够打印 HTML 表单和读取

QUERY\_STRING 与 PATH\_INFO 变量的简单的 CGI 来实现大量有用的 Web 应用程序。

在此处对 HTML 表单进行简单的概括再合适不过了，因为表单只与 Web 应用程序相关。即使您已经了解了 HTML，将 HTML 表单放到 REST 架构的上下文中仍然非常有用。

HTML 表单由<FORM>标签围住。开始的<FORM>标签有两个主要的属性：第一个属性是 action，它包含要调用的 CGI 脚本的标识符或者要操作的资源；第二个属性是 method，它包含提交表单时要使用的 HTTP 动词。

## 20.4 HTML 表单的有限词汇

HTTP 表单支持的 HTTP 动词只有用于读取资源的 GET 和用于向资源中写数据的 POST。表单动作 PUT 或者 DELETE 在 HTML 中是无效的，大多数 Web 浏览器会提交 POST 请求。您将看到，这给基于 REST 的 Web 应用程序的实现增加了一点难度，但是还不是太糟。

在开始的<FORM>标签和结束的</FORM>标签之间，可以使用特殊的 HTML 标签，Web 浏览器将它们显示为 GUI 控件。可用的 GUI 控件包括文本框、复选框、单选按钮组、激活表单提交的按钮(都可以通过 INPUT 标签实现)，大的文本输入域(TEXTAREA 标签)，以及下拉选择框(SELECT 标签)。

如果在 EasyCGIServer 的安装目录中创建文件 SimpleHTMLForm.html 来放置这些 HTML 标签，可以通过 URL <http://localhost:8001/SimpleHTMLForm.html> 检索它。由于它不是一个 CGI 脚本，EasyCGIServer 将它作为一个静态文件，就像 EasyWebServer 那样。如果之后您单击提交按钮，Web 浏览器便将表单数据编码到一个 GET 请求中，并且将它提交给以/cgi-bin/PrintFormSubmission.cgi 开头的一个长标识符所代表的资源。由于磁盘上没有对应于资源标识符的文件和脚本，Web 服务器不做任何有用的事，而只返回一个“页面未找到”的错误文档(状态代码：著名的 404)。然而，利用 Python 提供的 cgi 模块，编写可以接受表单提交并对表单进行处理的脚本很容易。

### cgi 模块：解析 HTML 表单

当单击页面 SimpleHTMLForm.html 上的某个提交按钮时，注意得到的并不是标签<FORM>的 action 属性指定的资源/cgi-bin/PrintFormSubmission.cgi，而是一个略微不同的资源，该资源的标识符是很长很笨重的/cgi-bin/PrintFormSubmission.cgi?textField=Some+text&radioButton=2&button=Submit。

这就是 GET 的表单提交的工作方式：Web 浏览器收集提交的表单中各个域的值，并对这些值进行编码，使它们不会包含任何在 URL 中无效的字符(例如，空格用加号代替)。之后它将输入域中的值附加到目的表单后面，并获取要检索的真正的资源。假设另一端有一个拦截请求的 CGI，该 CGI 将在其环境变量 QUERY\_STRING 中看到已编码的表单信息。当使用 POST 动词提交一个表单时，类似的编码过程也会发生，但在这种情形下，表单数据作为数据的一部分发送，而不是作为资源标识符的一部分发送。POST 的数据在标准输入中可用，而不是通过环境变量使其对脚本可用。

cgi 模块知道如何对 HTTP 请求中的表单数据进行解码，不管请求使用的是 GET 还是

POST。cgi 模块可以从环境变量中获取数据(GET)，也可以从标准输入中获取数据(POST)，并用这些数据在类 FieldStorage 中重新构造原来的 HTML 表单。

可以像访问字典那样访问 FieldStorage，但是最安全的使用方式是调用它的 `getfirst()` 方法，并传入希望得到其值的域的名称。

## 20.5 访问表单值的安全性

为什么 `form.getfirst('fieldName')` 比 `form['fieldName']` 更安全？问题的根本在于有时表单的一次提交中会合法地为同一个域提供两个或者多个值(例如，当用户在一个允许多选的选择框中选择多个值时，就会发生这种情况)。在这种情形下，`form['fieldName']` 将返回一列值(例如，在多选框中选择的所有值)，而不是一个值。如果脚本预计到了这种情形，那么没有问题，但是由于用户完全掌控提交给 CGI 脚本的数据，一个恶意用户就可以向一个只期待一个输入值的域提交多个值。

如果某人对使用 `form['fieldName']` 的脚本耍这种花招，在期望单个对象之处就会得到一个列表。如果按照处理单个对象的方式处理一个列表，脚本将崩溃。这就是使用 `getfirst` 更安全的原因：它保证总是返回第一个提交的值，即使用户试图使用恶意数据破坏脚本。

`getfirst` 方法在 Python 2.2 之前的版本中不可用。为了安全性考虑，需要用如下的代码模拟 `getfirst`：

```
fieldVal = form.getValue("field")
if isinstance(fieldVal, list): #More than one "field" was submitted.
    fieldVal = fieldVal[0]
```

如果一个 CGI 变量期待多个值，使用 `__getlist__` 方法而不是 `getfirst` 方法可以得到所有的设置值。

现在您已经了解了 FieldStorage 对象，编写 SimpleHTMLForm.html:PrintFormSubmission.cgi 的另一部分变得很容易。这一部分是一个 CGI 脚本，可以打印出表单输入域中的值：

```
#!/usr/bin/python
import cgi
import cgitb
cgitb.enable()

form = cgi.FieldStorage()
textField = form.getfirst("textField")
radioButton = form.getfirst("radioButton")
submitButton = form.getfirst("button")

print('Content-type: text/html\n')
print('<html>')
print('<body>')
print('<p>Here are the values of your form submission:</p>')
print('<ul>')
```

```
print('<li>In the text field, you entered "%s".</li>' % textField)
print('<li>Of the radio buttons, you selected "%s".' % radioButton)
print('<li>The name of the submit button you clicked is "%s".' % submitButton)
print('</ul>')
print('</body>')
print('</html>')
```

现在，在 SimpleHTMLForm.html 上单击提交按钮时，您不会看到错误代码 404，即“未找到”，而是将看到如图 20-1 所示的输出：

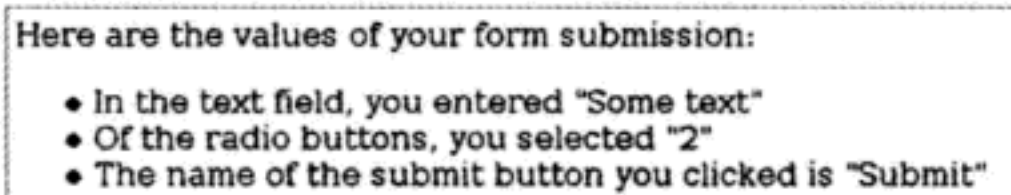


图 20-1

目前为止，一切都很顺利。还可以再深入一步，创建一个可以打印任何表单提交的脚本。您可以通过这种方式对不同类型的 HTML 表单进行实验。当您开始运行脚本但未向其提交一个表单时，可以让这个新脚本打印出一个相当复杂的 HTML 表单。将下面的脚本命令为 PrintAnyFormSubmission.cgi:

```
#!/usr/bin/python
import cgi
import cgitb
import os

cgitb.enable()
form = cgi.FieldStorage()

print('Content-type: text/html\n')
print('<html>')
print('<body>')
if form.keys():
    verb = os.environ['REQUEST_METHOD']
    print('<p>Here are the values of your %s form submission:' % verb)
    print('<ul>')
    for field in form.keys():
        valueObject = form[field]
        if isinstance(valueObject, list):
            #More than one value was submitted. We therefore have a
            #whole list of ValueObjects. getlist() would have given us
            #the string values directly.
            values = [v.value for v in valueObject]
            if len(values) == 2:
                connector = '" and "' #'"Foo" and "bar"'
            else:
                connector = '" , and "' #'"Foo", "bar", and "baz"'
            value = '" , "'.join(values[:-1]) + connector + values[-1]
        else:
```

```

        #Only one value was submitted. We therefore have only one
        #ValueObject. getfirst() would have given us the string
        #value directly.
        value = valueObject.value
        print('<li>For <var>%s</var>, I got "%s"</li>' % (field, value))
    else:
        print(''<form method="GET" action="%s">

<p>Here's a sample HTML form.</p>

<p><input type="text" name="textField" value="Some text" /><br />
<input type="password" name="passwordField" value="A password" />
<input type="hidden" name="hiddenField" value="A hidden field" /></p>

<p>Checkboxes:
<input type="checkbox" name="checkboxField1" checked="checked" /> 1
<input type="checkbox" name="checkboxField2" selected="selected" /> 2
</p>

<p>Choose one:<br />
<input type="radio" name="radioButton" value="1" /> 1<br />
<input type="radio" name="radioButtons" value="2" checked="checked" /> 2<br />
<input type="radio" name="radioButtons" value="3" /> 3<br /></p>

<textarea name="largeTextEntry">A lot of text</textarea>

<p>Choose one or more: <select name="selection" size="4" multiple="multiple">
<option value="Option 1">Option 1</option>
<option value="Option 2" selected="selected">Option 2</option>
<option value="Option 3" selected="selected">Option 3</option>
<option value="Option 4" selected="selected">Option 4</option>
</select></p>

<p><input type="Submit" name="button" value="Submit this form" />
<p><input type="Submit" name="button" value="Submit this form (Button #2)" />

</form>' ' % os.environ['SCRIPT_NAME'])

print('</body>')
print('</html>')
```

**试一试****打印任何 HTML 窗体提交**

将脚本 `PrintAnyFormSubmission.cgi` 放入目录 `cgi-bin/` 中，并启动 `EasyCGIServer`。访问 `http://localhost:8001/cgi-bin/PrintAnyFormSubmission.cgi`。您将得到一个类似于图 20-2 所示的 HTML 表单。

更改希望改变的任意表单数据，并且单击任意一个 `Submit` 按钮。您将看到如图 20-3

所示的屏幕。

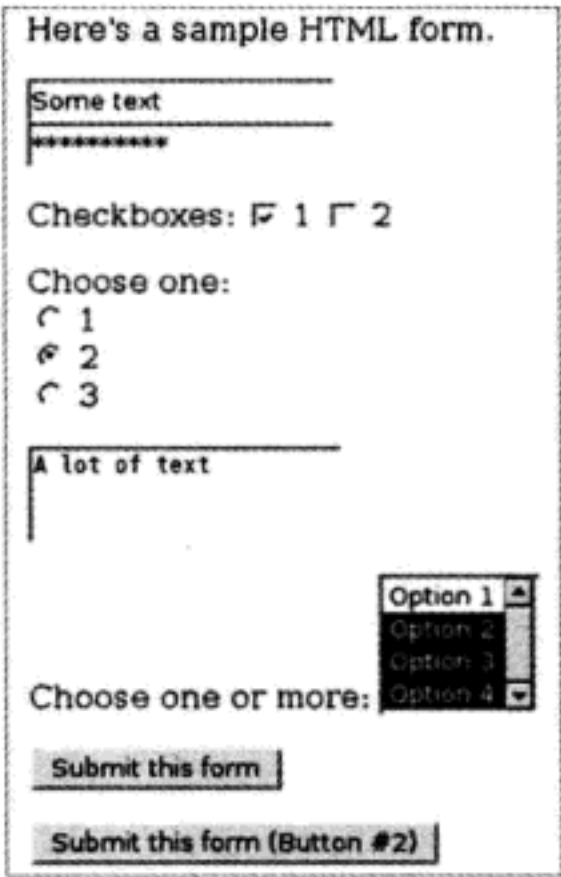


图 20-2

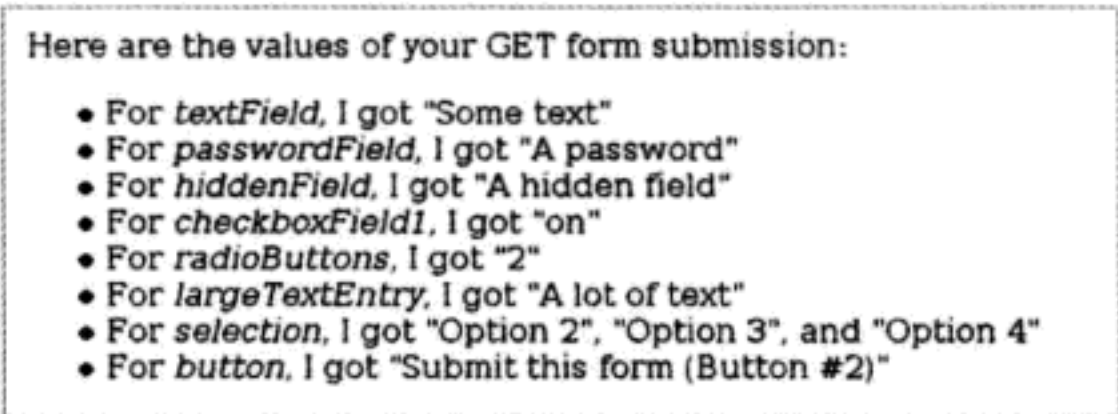


图 20-3

示例说明

初次请求被/cgi-bin/PrintAnyFormSubmission.cgi 标识的资源时，上述脚本使用 cgi 模块寻找表单提交。由于没有表单变量，它假设您根本没有提交表单，并展示默认资源：一个相当复杂的 HTML 表单。

当单击某个 Submit 按钮时，您请求一个非常不同的资源：类似于/cgi-bin/PrintAnyFormSubmission.cgi?textField=Some+text&passwordField=A+password&hiddenField=A+hidden+field&checkBoxField1=on&radioButtons=2&largeTextEntry=A+lot+of+text&selection=Option+2&selection=Option+3&selection=Option+4&button=Submit+this+form+%28Button+%232%29。这一次，cgi 模块使用大量的表单变量，并且输出动态生成的资源，它在提交的表单变量上迭代以描述提交的表单。如果再次用不同的值提交表单，就在请求一个略微不同的资源，脚本的 HTML 输出也会相应地不同。

注意：

如果您刚接触 Web 编程，要特别注意，尽管在提交的表单中有一个 checkBoxField2 域，但是在表单提交描述中并没有提到它。Web 浏览器不会将未选中的复选框放入表单提交信息中，因此它们不会显示在 FieldStorage 对象中。这一点不是太好。

可以在上面的脚本中使用 SimpleHTMLForm.html，也可以在 PrintFormSubmission.cgi 中使用它。实际上，可以在上述脚本中使用任意表单，包括为其他 Web 应用程序设计的表单，只要将表单的 action 属性修改为指向/cgi-bin/PrintFormSubmission.cgi 脚本即可。然而，如果没有提供任何输入(即 GET(获得)基本资源/cgi-bin/PrintFormSubmission.cgi)，那么将得到默认的 HTML 表单。当不带任何参数调用一个 CGI 脚本时，它将打印自己的表单，这种模式是构建自包含应用程序的一个强大工具。注意脚本使用特殊的 CGI 提供的环境变量 SCRIPT\_NAME 指向自己。即使将该脚本重命名，或者将它放到另一个目录中，它产生的表单仍然指向自己。

就像 EasyHTTPServer, PrintAnyFormSubmission.cgi 是试验新概念的好方法, 但是很快您就会感到厌烦。是时候学习一些更加有趣的内容了。接下来就介绍一个真正的 Web 应用程序。

## 20.6 构建 wiki

了解了 REST 这种 Web 架构和将程序与 Web 架构关联起来的主要方法 CGI 的基础知识后, 您已经可以设计和构建基本的应用程序了。下面详细描述了一个称作 wiki 的简单内容管理系统的构建过程。

Wiki 由 Ward Cunningham 于 1995 年设计, 是现在著名的 Wikipedia([www.wikipedia.org](http://www.wikipedia.org)) 的前身, Wikipedia 一个免费的在线百科全书(见图 20-4)。Cunningham 设计的最原始的 wiki (<http://c2.com/cgi/wiki/>) 目前在程序员中依然很流行, 它包含了有关技术讨论和最佳专业实践的信息。当然, 还包括之前提到的 REST wiki。

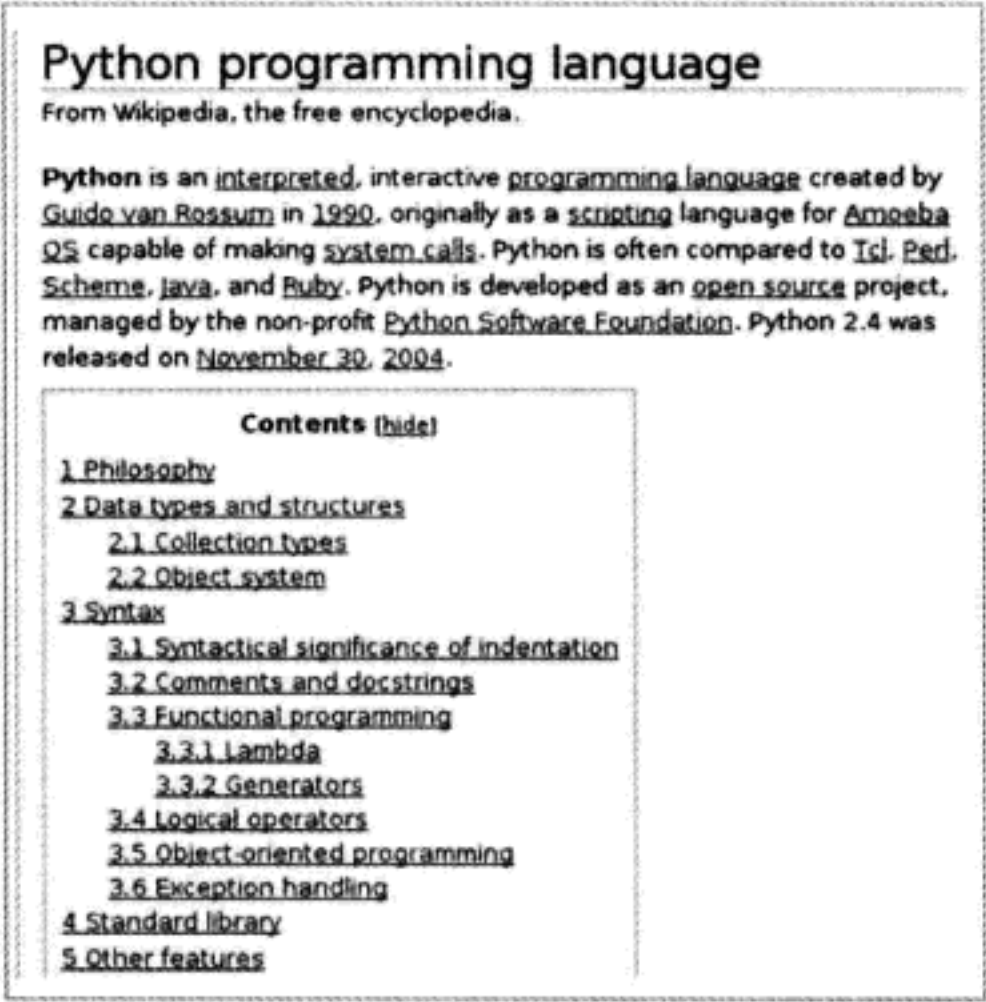


图 20-4

wiki 最与众不同的特性如下:

- 开放性, 基于 Web 的可编辑性——一些内容管理系统需要特殊的软件或者用户账户才能使用, 但是 wiki 页面允许通过任意的 Web 浏览器进行编辑。在大多数 wiki 上, 任何人都可以打开并编辑任意页面。由于存在垃圾邮件和破坏行为, 一些 wiki 开始要求用户账户。然而, 即使有些 wiki 区分成员和非成员, 基本的准则是任意成员均可以编辑任意页面。这使 wiki 看起来有点不正式, 但是由于 wiki 对输入不做要求, 这可以鼓励人们多为 wiki 贡献资源。
- 页面的平面名称空间——wiki 中的每个页面都有一个唯一的名称。页面的名称通常是 WikiWord, 即包含几个首字母大写的单词(页面的标题), 这几个单词合在一起构成了 wiki 的页面名称。wiki 页面的名称通常类似于 WikiPageNamesOftenLook-

LikeThis。wiki 中没有目录结构，所有页面都直接属于顶层节点。所有页面通过创建额外的索引和门户页面组织在一起。

- **通过引用进行链接**——一个 wiki 页面可以通过在其正文中引用另一个页面的 WikiWord 名称直接链接到该页面。当提交一个页面时，其引用的所有 WikiWord 都链接到相应的页面。一个页面可能会引用一个对应的页面还不存在的 WikiWord，提交该页面时，这个引用链接到一个创建还不存在的页面的表单。不使用 WikiWord 命名页面的 wiki 必须定义其他一些规则，使得可以链接到同一 wiki 中的其他页面。
- **简单的基于文本的标记**——wiki 并不要求用户输入 HTML，它使用一些简单的规则将 ASCII 文本转换成提交页面时显示的 HTML。示例规则包括使用空行表明一个新段落的开始，以及使用\*星号\*加粗一个选项。遗憾的是，这些约定都是非正式的，并不存在严格的规则。因此，不同 wiki 的实现规则千差万别。

**注意：**

要想查看 Cunningham 最初的 wiki 设计准则，请参考 <http://c2.com/cgi/wiki?WikiDesign-Principles>。

示例应用程序通常都缺乏一些必要特征，从而不能成为可以实际使用的应用程序。在本章中展示一个网上商店应用程序的话会太过复杂而不易理解，同时又不够完整，不能用来实际运营网上商店。由于定义 wiki 的特征很少而且很简单，因此在几页中就可以设计、创建和解释一个完全成熟的 wiki。BittyWiki 是根据刚才描述的准则设计和创建的应用程序，它的大小还不足 10 KB，然而它还不是使用 Python 编写的最短的 wiki。

**注意：**

<http://infomesh.net/2003/wypy/wypy.txt> 上的 wiki 由 11 行 Python 代码构成，仅包含 814 个字符。看这个 wiki 的代码是件非常痛苦的事情。

### 20.6.1 BittyWiki 核心库

在开始编写代码之前，需要首先决定希望创建的 wiki 具有哪些特性。下面的示例所做的设计决策产生了一个最简单的 wiki 后端，毕竟此处讨论的目的主要在于 BittyWiki 最重要的部分，即它展示给 Web 的接口，而不是它的后端。

#### 1. 后端存储

各种 wiki 以不同的方式存储它们的页面。一些 wiki 在磁盘上存储文件，另外一些 wiki 在数据库中存储页面，还有一些 wiki 在版本控制库中存储文件，这样可以很容易地抵制破坏。出于简单性考虑，一个 BittyWiki 安装将页面保存在同名的磁盘文件中。给定 wiki 的所有页面保存在同一个目录下。由于 wiki 的名称空间是平面名称空间，因此不需要任何子目录。

#### 2. WikiWord

每个使用 WikiWord 的 wiki 实现都必须判断哪些字符串是有效的 wiki 页面名称，以便

可以自动链接这些页面的引用。BittyWiki 使用最简单的 WikiWord 定义：任何以大写字母开头的且至少包含两个大写字母的字符串(包含字母或数字)都是一个 WikiWord。字符串 WikiWord 本身就是一个 WikiWord、WikiWord2、WikiworD、WWW 以及 AI 也都是 WikiWord。

任何 wiki 页面都可以通过名称检索，但是还需要一个默认页面以应对没有指定名称的情况。该默认页面就是称作 HomePage 的页面。

### 3. 编写 BittyWiki 核心库

基于这些设计决策，现在可以编写 BittyWiki 的核心库了。这个核心库就是从后端读数据和向后端写数据的代码，它还要处理 WikiWord 链接。将下面的代码放入目录 cgi-bin/或者您的 PYTHON\_PATH 下的 BittyWiki.py 文件中：

```
"""This module implements the BittyWiki core code: that which is not
bound to any particular interface."""

import re
import os

class Wiki:
    "A class representing a wiki as a whole."
    HOME_PAGE_NAME = "HomePage"

    def __init__(self, base):
        "Initializes a wiki that uses the provided base directory."
        self.base = base

        if not os.path.exists(self.base):
            os.makedirs(self.base)
        elif not os.path.isdir(self.base):
            raise IOError('Wiki base "%s" is not a directory!' % self.base)

    def getPage(self, name=None):
        """Retrieves the given page for this wiki, which may or may not
        currently exist."""
        if not name:
            name = self.HOME_PAGE_NAME
        return Page(self, name)

class Page:
    """A class representing one page of a wiki, containing all the
    logic necessary to manipulate that page and to determine which other
    pages it references."""

    #We consider a WikiWord any word beginning with a capital letter,
    #containing at least one other capital letter, and containing only
    #alphanumerics.
    WIKI_WORD_MATCH = "(([A-Z][a-z0-9]*){2,})"
    WIKI_WORD = re.compile(WIKI_WORD_MATCH)
```

```
WIKI_WORD_ALONE = re.compile('^%s$' % WIKI_WORD_MATCH)

def __init__(self, wiki, name):
    """Initializes the page for the given wiki with the given
    name, making sure the name is valid. The page may or may not
    actually exist right now in the wiki."""

    #WIKI_WORD matches a WikiWord anywhere in the string. We want to make
    #sure the page is a WikiWord and nothing else.
    if not self.WIKI_WORD_ALONE.match(name):
        raise(NotWikiWord, name)
    self.wiki = wiki
    self.name = name
    self.path = os.path.join(self.wiki.base, name)

def exists(self):
    "Returns true if there's a page for the wiki with this name."
    return os.path.isfile(self.path)

def load(self):
    "Loads this page from disk, if it exists."
    if not hasattr(self, 'text'):
        self.text = ''
        if self.exists():
            self.text = open(self.path, 'r').read()

def save(self):
    "Saves this page. If it didn't exist before, it does now."
    if not hasattr(self, 'text'):
        self.text = ''
    out = open(self.path, 'w')
    out.write(self.text)
    out.close()

def delete(self):
    "Deletes this page, assuming it currently exists."
    if self.exists():
        os.remove(self.path)

def getText(self):
    "Returns the raw text of this page."
    self.load()
    return self.text

class NotWikiWord(Exception):
    """Exception thrown when someone tries to pass off a non-WikiWord
    as a WikiWord."""
    Pass
```

## 试一试

## 从交互的 Python 会话中创建 Wiki 页面

稍后将为 BittyWiki 编写一个 web 接口，本章后面的内容主要讨论如何通过 HTTP 访问它。熟悉基本 API 最简单的方法是在一个交互的 Python 会话中使用 BittyWiki，不需要任何 Web 接口：

```
>>> from BittyWiki import Wiki
>>> wiki = Wiki("localwiki")
>>> homePage = wiki.getPage()
>>> homePage.text = "Here's the home page.\n\nIt links to PageTwo and
PageThree."
>>> homePage.save()
```

目录 localwiki 现在包含您的 wiki 文件：

```
>>> #The "localwiki" directory now contains your wiki's files.
>>> import os
>>> open(os.path.join("localwiki", "HomePage")).read()
"Here's the home page.\n\nIt links to PageTwo and PageThree."
```

HomePage 引用了 wiki 中的其他页面，但是现在这些页面都不存在：

```
>>> page2 = wiki.getPage("PageTwo")
>>> page2.exists()
False
```

当然，可以创建一个 HomePage 引用的页面：

```
>>> page2.text = "Here's page 2.\n\nIt links back to HomePage."
>>> page2.save()
>>> page2.exists()
True
```

最后看一下 NotWikiWord 异常：

```
>>> wiki.getPage("Wiki")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "BittyWiki.py", line 25, in getPage
    return Page(self, name)
  File "BittyWiki.py", line 47, in __init__
    raise NotWikiWord, name
BittyWiki.NotWikiWord: Wiki
```

## 20.6.2 BittyWiki 的 Web 接口

BittyWiki 库提供了操作 wiki 的方法，但是没有提供任何用户接口。可以编写单独的脚步

本操作该库，或者从一个交互的提示中创建页面，但是 wiki 原本是为了在 Web 上使用而设计的。还存在另外一组设计决策，与 BittyWiki 如何将 wiki 页面以及操作作为 REST 资源进行呈现有关。

### 1. 资源

由于 REST 是基于资源的，设计 Web 应用程序时，首先要考虑的是打算提供的资源的性质。一个 wiki 只能提供一种类型的资源：平面名称空间中的页面。URL 路径中的信息比放在字符串中更容易读取，因此 wiki 页面应当通过向 CGI 发送 GET 请求并将页面名称附加在 CGI 路径后面来检索。得到的资源标识符类似于 `/bittywiki.cgi/PageName`。为了修改页面，应当向它的资源标识符发送 POST 请求。

wiki 页面上允许的操作包括：创建、读取、更新和删除一个页面。不同类型的资源都有这 4 种操作，所以使用专门的缩写词 CRUD 表示这 4 种操作，以描述执行这些操作的一些应用程序。wiki 是针对平面名称空间中已命名文本页面的基于 Web 的 CRUD 应用程序。

#### 注意：

大多数 wiki 要么将页面删除操作作为一个特殊的管理员命令实现，要么根本不实现该操作，这是因为删除命令很容易造成破坏。实现删除命令是 BittyWiki。

### 2. 请求的结构

CRUD 操作对应于 4 个主要的 HTTP 动词并不是出于偶然。回忆一下重复出现的这 4 个操作，无论主题是数据库、文件系统访问还是 Web 资源。理想状态下，一个 CRUD 操作对应一个 HTTP 动词。

当用户请求读一个页面时，他们只需提供页面名称。因此，对于 read 操作，没有必要向前一节定义的资源标识符附加额外信息。向资源标识符发送简单的 GET 请求就足够了。

当修改页面时，除了发送页面的名称外，还要发送页面的新内容。将新数据 POST 到资源标识符即可。

现在遇到了一个问题：还有另外两个操作(创建和删除)，但是只有一个 HTTP 方法(POST)既能够适用于这两个操作，又能够得到构成界面的 HTML 表单的支持。这些操作必须通过某种方式合并起来。

创建一个已经存在的页面或者编辑一个不存在的页面是没有意义的，因此这两个操作可以合并为一个单独的写操作。然而还有两个操作(写和删除)需要通过 POST 实现，所以问题依然存在。

一个解决办法是让用户在 POST 的数据中放入一个标记，指示他们希望执行的操作，而不是仅仅传递他们希望操作的数据。该标记的关键在于指示具体的操作，它有两个可选值，分别是 write 和 delete。

### 3. 更多资源

目前为止，设计假设写操作和删除操作是由对 HTML 表单提交的响应而引发的。那些 HTML 表单来自哪里？由于表单需要根据它们修改的页面的名称动态生成，所以它们必须

由 wiki 程序生成。这使得表单成为一种新的资源类型。与之前讲述的不同，BittyWiki 实际上提供两类资源。它的一项主要工作是提供页面，但是它还必须同时提供 HTML 表单来操作那些页面。

与页面不同，表单不能由用户创建、更新或者删除，他们只能读表单(表单被读取之后，可被用来创建、更新或者删除页面)。表单也可以通过 GET URL 访问。

由于用户需要请求一个表单来写或者删除一个特定的页面，因此使表单的资源标识符基于页面的资源标识符是有意义的。这有两种实现方式。第一种方式是继续将操作附加到标识符的 PATH\_INFO 后面，因此删除页面/bittywiki.cgi/MyPage 的表单的位置应该是/wiki.cgi/MyPage/delete。另外一种方式是使用 QUERY\_STRING，此时删除以上页面的表单应该位于/wiki.cgi/MyPage?operation=delete。

没有完全正确或者完全错误的解决方法。然而，由于 operation 关键字已经用于 POST 表单提交，而且页面(而不是表单)才是 wiki 的关键点，所以 BittyWiki 实现了第二个策略。它允许的值与 POST 命令允许的值相同：即 write 和 delete。

总结一下：BittyWiki 中的每个 wiki 页面都有 3 个关联资源。每个资源对 GET 和 POST 的响应方式都不同，如表 20-2 所示。

表 20-2

资 源	GET 操 作	POST 操 作
/bittywiki.cgi/PageName	如果页面存在，则显示页面，否则显示创建页面的表单	无
/bittywiki.cgi/PageName?operation=write	显示编辑表单	写页面，提供状态
/bittywiki.cgi/PageName?operation=delete	显示删除表单	删除页面，提供状态

如果没有指定页面名称(即，某人请求基本资源/bittywiki.cgi/)，CGI将要求核心wiki 代码检索默认页面。

当设计资源标识符并且衡量应该使用 PATH\_INFO 还是 QUERY\_STRING 时，需要考虑折中。/foo.cgi/clients/MegaCorp 和/foo.cgi?client=MegaCorp 是同一资源的两种合法的 REST 标识符。第一个标识符的优点是，它看起来更友好，更像一个“真正的”资源。如果希望使数据结构具有层次，没有比基于 PATH\_INFO 的标识符更好的资源标识策略了。

问题在于不能够与可以让您从一组客户端中选择 MegaCorp 的 HTML 表单一起使用该策略。HTML 表单的目的地需要在打印它时定义好，因此提前可以做的是确定/foo.cgi/，当用户提交表单时，让 Web 浏览器添加“?client=MegaCorp”。如果应用程序有这个问题，可以考虑为每个资源定义两个资源标识符：一个标识符使用 PATH\_INFO，另一个标识符使用 QUERY\_STRING。

4. wiki 标记

最后一个问题是思考如何将用户输入的纯文本转换为显示给用户的 HTML 页面。一些 wiki 没有限制，允许用户画表格和上传图片。BittyWiki 仅支持少数几种基本的文本到 HTML 的标记：

- 为了确保 HTML 的有效性，所有页面都放置在段落(<p>)标签中。
- 两个连续的换行符被当做一个段落分隔符。
- 手动输入到 wiki 页面的任意 HTML 都将被转义，以便显示给用户，而不是被 Web 浏览器解释。

由于标记的规则较少，BittyWiki 页面看起来有点平淡，不过禁止原始的 HTML 标记可以限制破坏者的能力。

做出这些设计决策后，现在可以为 BittyWiki 创建 CGI Web 接口了。这些代码应该放入脚本 `bittywiki.cgi` 中，该文件所在的目录与 `BittyWiki.py` 所在的目录 `cgi-bin/` 相同：

```
#!/usr/bin/python
import cgi
import cgitb
import os
import re
from BittyWiki import Wiki, Page, NotWikiWord
cgitb.enable()

#First, some HTML templates.
MAIN_TEMPLATE = '''<html>
<head><title>%(title)s</title>
<body>%(body)s<hr />%(navLinks)s</body>
</html>'''

VIEW_TEMPLATE = '''%(banner)s
<h1>%(name)s</h1>
%(processedText)s'''

WRITE_TEMPLATE = '''%(banner)s
<h1>%(title)s</h1>
<form method="POST" action="%(pageURL)s">
  <input type="hidden" name="operation" value="write">
  <textarea rows="15" cols="80" name="data">%(text)s</textarea><br />
  <input type="submit" value="Save">
</form>'''

DELETE_TEMPLATE = '''<h1>%(title)s</h1>
<p>Are you sure %(name)s is the page you want to delete?</p>

<form method="POST" action="%(pageURL)s">
  <input type="hidden" name="operation" value="delete">
  <input type="submit" value="Delete %(name)s!">
</form>'''

ERROR_TEMPLATE = '<h1>Error: %(error)s</h1>'
BANNER_TEMPLATE = '<p style="color:red;">%s</p><hr />'

#A snippet for linking a WikiWord to the corresponding wiki page.
VIEW_LINK = '<a href="%s">%(wikiword)s</a>'
```

```
#A snippet for linking a WikiWord with not corresponding page to a
#form for creating that page.
ADD_LINK = '%%(wikiword)s<a href="%s">?</a>'
```

不是从 CGI 脚本中打印 HTML 页面，而是提前将 HTML 模板定义为字符串，并使用 Python 的字符串插值功能在它们中添加动态值。这帮助将表示和内容分开，使得定制 HTML 变得容易。将 HTML 从 Python 代码中分开，使得将模板移交给不懂 Python 的 Web 设计人员成为可能。

Python 有一个应当得到广泛认可的特性，即它利用映射而不是元组进行字符串插值的能力。如果有一个字符串“A %(foo)s string”，还有一个映射，它包含以 foo 为键的一项，利用该映射进行字符串插值将会把字符串中的“%(foo)s”替换为键 foo 对应的字符串值。

```
class WikiCGI:

    #The possible operations on a wiki page.
    VIEW = ''
    WRITE = 'write'
    DELETE = "delete"

    def __init__(self, wikiRoot):
        self.wiki = Wiki(wikiRoot)

    def run(self):
        toDisplay = None
        try:
            #Retrieve the wiki page the user wants.
            page = os.environ.get('PATH_INFO', '')
            if page:
                page = page[1:]
                page = self.wiki.getPage(page)
            except NotWikiWord, badName:
                page = None
                error = '"%s" is not a valid wiki page name.' % badName
                toDisplay = self.makeError(error)

        if page:
            #Determine what the user wants to do with the page they
            #requested.
            makeChange = os.environ['REQUEST_METHOD'] == 'POST'
            if makeChange:
                defaultOperation = self.WRITE
            else:
                defaultOperation = ''
            form = cgi.FieldStorage()
            operation = form.getfirst('operation', defaultOperation)

            #We now know which resource the user was trying to access
```

```

#("page" in conjunction with "operation"), and "form"
#contains any representation they were submitting. Now we
#delegate to the appropriate method to handle the operation
#they requested.
operationMethod = self.OPERATION_METHODS.get(operation)
if not operationMethod:
    error = '"%s" is not a valid operation.' % operation
    toDisplay = self.makeError(error)

if not page.exists() and operation and not \
(makeChange and operation == self.WRITE):
    #It's okay to request a resource based on a page that
    #doesn't exist, but only if you're asking for the form to
    #create it, or actually trying to create it.
    toDisplay = self.makeError('No such page: "%s"' % page.name)

if operationMethod:
    toDisplay = operationMethod(self, page, makeChange, form)

#All the operation methods, as well as makeError, are expected
#to return a set of values that can be used to render the HTML
#response: the title of the page, the body template to use, a
#map of variables to interpolate into the body template, and a
#set of navigation links to put at the bottom of the page.
title, bodyTemplate, bodyArgs, navLinks = toDisplay
if page and page.name != Wiki.HOME_PAGE_NAME:
    backLink = '<a href="%s">Back to wiki homepage</a>'
    navLinks.append(backLink % self.makeURL())
print("Content-type: text/html\n")
print(MAIN_TEMPLATE % {'title' : title,
                       'body' : bodyTemplate % bodyArgs,
                       'navLinks' : ' | '.join(navLinks)})

```

当 WikiCGI 被实例化后, 它找出正在请求的资源, 以及用户希望在资源上所做的操作。它委托可以处理各种可能操作的那些方法(需要被定义)中的某个方法来完成该操作。

每个这样的方法都应该返回 Web 页面的框架: 标题, 模板字符串(之前定义的某个模板: VIEW\_TEMPLATE、WRITE\_TEMPLATE 等), 在对模板进行插值时使用的变量映射, 以及帮助用户浏览 wiki 的一组链接。

WikiCGI 实例化的最后一项工作是填充该框架: 对提供的变量映射进行插值, 将其修改为页面特定的模板字符串, 之后再对模板字符串进行插值, 将其修改为主模板。返回结果是一个完整的 HTML 页面, 它简单地被打印到标准输出。

CGI 的下一部分定义了 3 个特定的操作方法, 它们接受页面和表单数据中存储的(可能的)资源表示作为参数, 进行适当的修改, 并返回文档的原始材料:

```

def viewOperation(self, page, makeChange, form=None, banner=None):
    """Renders a page as HTML, either as the result of a request
    for it as a resource, or as a side effect of some other

```

```

operation."""
if banner:
    banner = BANNER_TEMPLATE % banner
else:
    banner = ''
if not page.exists():
    title = 'Creating %s' % page.name
    toDisplay = (title, WRITE_TEMPLATE,
                 {'title' : title,
                  'banner' : banner,
                  'pageURL' : self.makeURL(page),
                  'text' : ''},
                 [])
else:
    writeLink = '<a href="%s">Edit this page</a>' \
                % self.makeURL(page, self.WRITE)
    deleteLink = '<a href="%s">Delete this page</a>' \
                  % self.makeURL(page, self.DELETE)
    toDisplay = (page.name, VIEW_TEMPLATE,
                 {'name' : page.name,
                  'banner' : banner,
                  'processedText' : self.renderPage(page)},
                 [writeLink, deleteLink])
return toDisplay

def writeOperation(self, page, makeChange, form):
    "Saves a page, or displays its create or edit form."
    if makeChange:
        data = form.getfirst('data')
        page.text = data
        page.save()
        #The operation is done, but we still need a document to
        #return to the user. Display the new version of this page,
        #with a banner.
        toDisplay = self.viewOperation(page, 0, banner='Page saved.')
    else:
        navLinks = []
        pageURL = self.makeURL(page)
        if page.exists():
            title = 'Editing ' + page.name
            navLinks.append('<a href="%s">Back to %s</a>' % (pageURL,
                                                            page.name))
        else:
            title = 'Creating ' + page.name
            toDisplay = (title, WRITE_TEMPLATE, {'title' : title,
                                                  'banner' : '',
                                                  'pageURL' : pageURL,
                                                  'text' : page.getText()},
                        navLinks)
    return toDisplay

```

```

def deleteOperation(self, page, makeChange, form=None):
    "Deletes a page, or displays its delete form."
    if makeChange:
        page.delete()
        banner = 'Page "%s" deleted.' % page.name
        #The page is deleted, but we still need a document to
        #return to the user. Display the wiki homepage, with a banner.
        toDisplay = self.viewOperation(self.wiki.getPage(), 0,
                                       banner=banner)
    else:
        if page.exists():
            title = 'Deleting ' + page.name
            pageURL = self.makeURL(page)
            backLink = '<a href="%s">Back to %s</a>'
            toDisplay = (title, DELETE_TEMPLATE, {'title' : title,
                                                  'name' : page.name,
                                                  'pageURL' : pageURL},
                        [backLink % (pageURL, page.name)])
        else:
            error = "You can't delete a page that doesn't exist."
            toDisplay = self.makeError(error)
    return toDisplay

#A registry mapping 'operation' keys to methods that perform the
operations.
OPERATION_METHODS = { VIEW : viewOperation,
                      WRITE: writeOperation,
                      DELETE: deleteOperation }

def makeError(self, errorMessage):
    "Creates a set of return values indicating an error."
    return (ERROR_TEMPLATE, "Error", {'error' : errorMessage,
                                       'mainURL' : self.makeURL("")}, [])

def makeURL(self, page="", operation=None):
    "Creates a URL to the resource defined by the given page and resource."
    if hasattr(page, 'name'):
        #A Page object was passed in, instead of a page name.
        page = page.name
    url = os.environ['SCRIPT_NAME'] + '/' + page
    if operation:
        url += '?operation=' + operation
    return url

```

CGI 的最后一个主要部分是将原始的 wiki 文本转换为 HTML 的代码, 它们将 WikiWord 与 BittyWiki 资源链接起来, 并创建段落分隔符:

```

#A regular expression for use in turning multiple newlines
#into paragraph breaks.

```

```

MULTIPLE_NEWLINES = re.compile("(\r?\n){2,}")

def renderPage(self, page):
    """Returns the text of the given page, with transforms applied
    to turn BittyWiki markup into HTML: WikiWords linked to the
    appropriate page or add form, and double newlines turned into
    paragraph breaks."""

    #First, escape any HTML present in the bare text so that it is
    #shown instead of interpreted.
    text = page.getText()
    for find, replace in (('<', '&lt;'), ('>', '&gt;'), ('&', '&amp;')):
        text = text.replace(find, replace)

    #Link all WikiWords in the text to their view or add resources.
    html = '<p>' + page.WIKI_WORD.sub(self._linkWikiWord, text) \
        + '</p>'

    #Turn multiple newlines into paragraph breaks.
    html = self.MULTIPLE_NEWLINES.sub('</p>\n<p>', html)
    return html

def _linkWikiWord(self, match):
    """A helper method used to replace a WikiWord with a link to view
    the corresponding page (if it exists), or a link to create the
    corresponding page (if it doesn't)."""
    linkedPage = self.wiki.getPage(match.group(0))
    link = ADD_LINK
    if linkedPage.exists():
        link = VIEW_LINK
    link = link % self.makeURL("%(wikiword)s")
    #The link now looks something like:
    # <a href="/cgi-bin/bittywiki.cgi/%(wikiword)s">%(wikiword)s</a>
    #We'll interpolate 'wikiword' to fill in the actual page name.
    return link % {'wikiword': linkedPage.name}

```

最后，当该文件作为脚本运行时，下面的代码针对特定的 wiki 调用 WikiCGI:

```

If __name__=='__main__':
    WikiCGI("wiki/").run()

```

如果已经准备好，就可以开始编辑自己的页面了。

使这段代码可执行，并且尝试与 EasyCGIServer 或者自己的 Web 主机的 CGI 设置一起使用它。访问 <http://localhost:8001/cgi-bin/bittywiki.cgi>(或者自己的 Web 主机上的对应地址)，它会发送给您创建 wiki 主页的表单。可以创建一个主页，引用其他暂时还不存在的页面，之后单击这些页面名称旁边的问号创建它们。从此处开始就可以构建自己的 wiki 了，实际的 wiki 就是以这种方式增长的。wiki 是管理开发团队中人员之间相互协作的出色工具，也可以记录自己的笔记。构建 wiki 很容易也很有趣，这也是有如此多的 wiki 实现存在的原因。

BittyWiki 是一个简单但功能完备的 wiki，它的设计简单而灵活。HTML 表示与逻辑分离，标识资源的任务通过某个方法完成，之后发送给某个处理方法。处理方法识别提供的表示(如果存在的话)，执行适当的操作，并返回即将显示的资源表示或者其他文档。考虑到可能会出现各种问题，资源和操作根据 REST 原则设计。这种设计和架构是构建独立的 Web 应用程序的有效方法。

## 20.7 Web 服务

到目前为止，本章开发的 Web 应用程序都做了一个没有明确指出的假设：它们的目标观众是人。Web 上可用的大多数应用程序也是如此。典型的 Web 应用程序(刚才编写的 wiki 也不例外)提供的资源表示是数据、响应消息、页面布局代码和导航等的结合，它们都放在一个 HTML 文件中，以便 Web 浏览器以用户喜爱的方式显示它们。当需要交互时，应用程序展示 GUI 表单，您可以通过人机界面填写它。提交表单后，将得到一个精美的 HTML 页面。简言之，Web 应用程序是由人编写且为人服务的。

尽管大多数 Web 应用程序以人为中心，但它们仍然拥有非人类用户：不直接受人指导的软件客户端，或者给它们起一个容易记住的名称：机器人。从搜索引擎蜘蛛到自动的拍卖投标脚本，再到实时天气显示客户端，各种脚本客户端都使用 Web 应用程序，但通常最初编写那些应用程序的人并不知道这一点。如果某个 Web 应用程序被证明很有用，肯定会有人编写使用该 Web 应用程序的机器人。

以前，机器人除了模仿 Web 浏览器之外别无选择。它们需要像 Web 浏览器一样发出 HTTP 请求，并解析返回的 HTML 页面以找到感兴趣的部分。尽管这项技术在现在仍然通用，但是越来越多的 Web 应用程序都为机器人提供专门的接口，这样会使编写机器人变得容易，并且服务器不必再占用带宽发送无用数据。这些接口称作 Web 服务。一些著名的公司，例如 Google、Yahoo!、Amazon 以及 eBay 都为它们的 Web 应用程序提供了 Web 服务 API，一些不知名的公司也如此。

围绕 Web 服务已经创建了许多出色的标准，本章后面将介绍其中的几个，但是基本事实是 Web 服务就是针对机器人的 Web 应用程序。一个 Web 服务通常对应一个 Web 应用程序，并使该 Web 应用程序的一些功能方便机器人使用。这些标准存在的唯一理由就是使编写机器人或者向机器人呈现应用程序变得简单。

机器人与人的需求不同。人可以扫一眼页面的 HTML，就区分重要的特定于页面的数据和导航、图标及杂乱的数据。机器人没有这样的能力，所以必须为它们编程序，使它们解析出需要的数据。如果做了重新设计，改变了某个站点生成的 HTML，就必须重新编写读和解析该 HTML 的任意机器人。当 Web 应用程序需要时，人可以回忆或者编造输入，而机器人则需要事先编程序以提供正确的输入。因此，Web 服务通常比它对应的 Web 应用程序拥有更好的使用文档，而且能提供更加结构化的资源表示，这一点都不奇怪。

Web 服务和使用它们的脚本可以以共生的关系存在。如果提供了人们希望使用的 Web 服务，就围绕产品形成了一个社区，并且从中获得了知名度。可以给用户自由，让他们在您的应用程序上构建新的应用程序，而不必亲自实现他们的功能请求。记住，如果应用程

序确实很有用，无论您怎么做，都会有人编写机器人来使用它。您倒不如支持这种使用，并对其进行监视和跟踪。

使用其他人的 Web 服务的好处更加明显：不必自己实现数据集和算法。也不必获得使用这些数据集的许可，因为 Web 服务都是事先打包许可的。

即使既能控制数据的生产者，也能控制数据的使用者，使用 Web 服务协调两者依然很有好处。Web 服务使您可以在不同的计算机和程序语言间共享代码，如同 Web 应用程序可以从任意浏览器和操作系统访问一样。

Python 非常适合使用和提供 Web 服务。它对输入没有严格要求，所以可以匹配各种 Web 服务标准，这些标准通常只提供有限的或者不提供输入。由于 Python 允许重载类方法的调用操作符，这使得 Web 服务调用看起来非常像一个普通的方法调用。最后，Python 的标准库对 Web 提供了基本的、不错的支持。如果高层协议不能满足需求或者它的库有错误，可以深入到下一层，并且仍然能完成任务。

## Web 服务的工作方式

Web 服务是为机器人提供的 Web 应用程序，因此它们的运行方式与普通的 Web 应用程序类似：发送一个 HTTP 请求，然后在响应中得到结构化的数据。Web 服务应当被脚本使用，因此需要更正式地定义请求和响应的格式。Web 应用程序通常返回一个完整页面，它被浏览器显示并被人脑解析，而 Web 服务则仅以易于解析的格式(通常是 XML)返回“重要的”数据。Web 服务提供的方法通常都有人类可读的或者可被计算机解析的描述，这使得用户编写满足他们需求的脚本变得容易。

Web 服务有 3 个主要标准：REST，XML-RPC 和 SOAP。对于每个标准，本章将详细介绍如何使用已经存在的公共的 Web 服务来完成有意义的工作，如何将 BittyWiki API 作为 Web 服务提供，以及如何让机器人通过该 Web 服务对 wiki 进行操作。

## 20.8 REST Web 服务

REST 不只对人们使用的 Web 非常重要，对机器人也一样可用。为人类用户设计的超文本链接和 HTML 表单是 REST API 的接入点，可以很容易地被正确编码的机器人使用。您仅需为自己的资源添加对机器人友好的表示，以及允许机器人得到那些表示的方法。

如果从头设计一个 Web 应用程序，要谨记人类用户和机器人的需求。最终可以为 HTML 表单和外部脚本提供类似的 API。不太可能为的人和机器人提供完全相同的特性，但是可以重用许多架构和代码。

在某些情形下，可能希望创建一个新的、简单的 API，并且将它作为 Web 服务提供。例如，当应用程序具有不会被外人看到的、不易用的 API 时，当 Web 应用程序非常复杂时，或者人们希望使用部分 API 编写机器人时，可能想采取这种做法。

### 20.8.1 REST 快速入门：在 Amazon.com 上寻找廉价商品

Amazon.com 是一个流行的网上购物 Web 站点，通过使用称作 Amazon Web Services

(AWS)的 REST Web 服务使它的数据可用。可能该 Web 服务最有趣的特性是它提供的查询书或者其他商品，以及检索商品的元数据、图片和评论的能力。Amazon 允许程序访问它的产品数据库，通过其他方法复制或者得到这些数据是很困难的。

注意：

Amazon Web Services 的主页位于 <http://aws.amazon.com/>。

为了使用 Amazon Web Services，需要一个订阅 ID。这是一个包含 13 个字符的字符串，用来标识账户。可以在 [www.amazon.com/gp/aws/registration/registration-form.html](http://www.amazon.com/gp/aws/registration/registration-form.html) 上注册一个免费的 ID。得到 API 密钥之后，可以用它查询 Amazon Web Services。由于 AWS 接口是基于 REST 的，所以可以通过向一个特定的资源发送 GET 请求调用该接口，结果在一个 XML 文档中返回。它是等价于 Amazon 搜索引擎 Web 应用程序的 Web 服务。AWS 不是基于 HTML 表单的用户接口，它拥有构建资源的规则。它返回结果的结构化 XML 表示，而不是一个包含搜索结果的漂亮的 HTML 文档。

Amazon Web Services 实际上并不完全符合 REST。AWS 的大部分设计是基于 REST 的，但是它定义了若干操作，当通过 GET 获取服务器端的资源时，这些操作在服务器端进行修改。例如，AWS 的 CartModify 操作使您可以仅通过发送 GET 请求就可以向自己的 Amazon 购物车添加商品或者从购物车中删除商品。回忆一下，GET 请求不应当修改服务器端的任何资源，您应该使用 POST、PUT 或者 DELETE 完成修改操作。AWS 的设计者们很可能在权衡之后选择了一致性(即使用 GET 完成所有操作)，而不是完全符合 REST。

由于 AWS API 并不完全符合 REST 架构，因此随意传递 AWS 给您的资源标识符是不安全的。其他人也许会不小心向您的购物车中添加了书籍！在设计自己的 REST API 时，需要避免这些问题。

## 试一试

## Amazon Web Services 响应一瞥

可以使用下载 Web 页面的 urllib 模块调用 Amazon Web Services。下面是个交互式 Python 会话，它搜索 James Joyce 写的书(为了简洁性，稍微对它进行了重新调整和编辑)：

```
>>> import urllib
>>> author = "Joyce, James"
>>> subscriptionID = [your subscription id]
>>> url = "http://xml.amazon.com/onca/xml3?f=xml&t=webservices-20&dev-t=%s&type=lite&mode=books&AuthorSearch=%s" % (subscriptionID, urllib.quote(author))
>>> print(urllib.urlopen(url).read())
<?xml version="1.1" encoding="UTF-8"?>
<ProductInfo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noName
spaceSchemaLocation="http://xml.amazon.com/schemas3/dev-lite.xsd">
...
<Details url="http://www.amazon.com/exec/obidos/ASIN/0142437344/webservices-
20?dev-t=D801OTR10IMN7%26camp=2025%26link_code=xm2">
    <Asin>0142437344</Asin>
```

```

    <ProductName>A Portrait of the Artist As a Young Man (Penguin Classics)
</ProductName>
    <Catalog>Book</Catalog>
    <Authors>
        <Author>James Joyce</Author>
    </Authors>
    <ReleaseDate>25 March, 2003</ReleaseDate>
    <Manufacturer>Penguin Books</Manufacturer>

<ImageUrlSmall>http://images.amazon.com/images/P/0142437344.01.THUMBZZZ.jpg
</ImageUrlSmall>

<ImageUrlMedium>http://images.amazon.com/images/P/0142437344.01.MZZZZZZZ.jpg
</ImageUrlMedium>

<ImageUrlLarge>http://images.amazon.com/images/P/0142437344.01.LZZZZZZZ.jpg
</ImageUrlLarge>
    <Availability>Usually ships in 24 hours</Availability>
    <ListPrice>$9.00</ListPrice>
    <OurPrice>$8.10</OurPrice>
    <UsedPrice>$1.95</UsedPrice>
</Details>
...
</ProductInfo>

```

### 示例说明

上述示例打开一个 URL 并读取它。可以在 Web 浏览器中访问同一个 URL(将 Web 服务当做 Web 应用程序处理), 并且得到与交互式 Python 会话完全一致的数据。Web 应用程序与 Web 服务之间的不同与架构无关, 它们都使用 Web 架构。唯一不同之处在于请求和响应的格式。

然而在打开和读取资源时存在两个问题(不论是从脚本还是从 Web 浏览器打开资源), 在会话日志中这一点很明显。第一个问题是, 搜索用的 AWS URL 很长, 非常不容易记住。即使有一个参考指南, 使所有的 URL 保持正确还是很困难。第二个问题是, 响应包含大量的 XML 数据。解析这些数据或者将它们转变为用户友好的形式需要大量工作。幸运的是, 这项工作已经完成了。

一个流行的 Web 服务最终会产生由各种主要的程序语言编写的客户端。Amazon Web Services 的标准的 Python 客户端是 PyAmazon, 最初由 Mark Pilgrim 编写, 现在由 Michael Josephson 维护。该模块对 Amazon Web Services 的 REST API 进行了抽象。它使您可以通过一个方法调用请求某个复杂的资源, 并检索一个 Python 对象列表, 而不是一块 XML 数据。在后端, 该模块使用 urllib 检索资源(如同我们所做的那样), 之后将 XML 响应解析为 Python 数据结构。有了 PyAmazon, 用 Amazon Web Services 编写 Python 函数变得容易。

### 注意:

从 [www.josephson.org/projects/pyamazon/](http://www.josephson.org/projects/pyamazon/) 下载 PyAmazon, 并且将它安装到 PYTHON\_PATH 目录, 或者将它安装到保存使用 AWS 的脚本的目录中。还需要从该站点下载

OnDemandAmazonList, 这个类允许将 AWS 搜索结果分页列表当作普通的 Python 列表并在其上迭代。下面的示例应用程序使用 OnDemandAmazonList, 这样使代码看起来更自然。

## 20.8.2 WishListBargainFinder 简介

Amazon 允许个人和书商推广用过的书, Amazon 会展示此书用过的版本的最低价格以及自己销售的新书的价格。如果再次查看 James Joyce 的 XML 搜索结果, 可以看到 *A Portrait of the Artist as a Young Man* 在 Amazon 上的新书价格为 8.10 美元(OurPrice), 然而用过的版本的最低价格为 1.95 美元(UsedPrice)。即使算上运费, 这个价格仍然非常具有诱惑力。Amazon 上的许多二手书只卖 1 美分。Amazon 会展示每本书的用过版本的最低价格, 但是在整个列表中找到最低价格并不容易。

Amazon 用户可以保存自己的“购物意愿列表”。如果您也拥有该列表, 这表明您已经从 Amazon 上数以百万的商品中挑选出了少数几个极其感兴趣、希望低价购买的商品。Amazon Web Services 提供了一个购物意愿列表搜索功能, 因此可以编写一个使用 AWS 的脚本, 搜索购物意愿列表, 找出低价商品。如果不介意购买二手物品, 这会节省大量的金钱。

下面是类 BargainFinder, 它接受来自 AWS 查询的列表, 并且从中查找二手低价商品。低价商品可以被定义为价格低于一定量(例如, 3 美元)的商品, 或者是比 Amazon 中对应的新货价格低一定量(比如低 75%)的商品。类 BargainFinder 以及它后面的代码片段是文件 WishListBargainFinder.py 的一部分:

```
import copy
import re
import amazon

class BargainFinder:
    """A class that, given a list of Amazon items, finds out which
    items in the list are available used at a bargain price."""

    def __init__(self, bargainCoefficient=.25, bargainCutoff=3.00):
        """The bargainCoefficient is how little an item must cost
        used, versus its new price, to be considered a bargain. The
        default bargain coefficient is .25, meaning that an item
        available used for less than 25% of its Amazon price is
        considered a bargain.

        The bargainCutoff is for finding bargains among items that are
        cheap to begin with. The default bargainCutoff is 5, meaning
        that any item available used for less than $3.00 is considered
        a bargain, even if it's available new for only a little more
        than $3.00."""
        if bargainCoefficient >= 1:
            raise Exception, 'It makes no sense to look for "bargains" that ' \
                + 'cost more used than new!'
        self.coefficient = bargainCoefficient
        self.cutoff = bargainCutoff
```

```

def printBargains(self, items):
    """Find the bargains in the given list and present them in a
    textual list."""
    bargains = self.getBargains(items)
    printedHeader = 0
    if bargains:
        print ('Here are items available used for less than $%.2f, ' + \
              'or for less than %.2d%% of their Amazon price:') \
              % (self.cutoff, self.coefficient*100)
        prices = bargains.keys()
        prices.sort()
        for usedPrice in prices:
            for bargain, amazonPrice in bargains[usedPrice]:
                savings = ''
                if amazonPrice:
                    percentageSavings = (1-(usedPrice/amazonPrice)) * 100
                    savings = '(Save %.2d%% off $%.2f) ' \
                              % (percentageSavings, amazonPrice)
                Print(' $%.2f %s%s' % (usedPrice, savings,
                                       bargain.ProductName))
    else:
        print("Sorry, I couldn't find any bargains in that list.")

def getBargains(self, items):
    "Scan the given list, looking for bargains."
    bargains = {}
    for item in items:
        bargain = False
        amazonPrice = self.getPrice(item, "OurPrice")
        usedPrice = self.getPrice(item, "UsedPrice")
        if usedPrice:
            if usedPrice < self.cutoff:
                bargain = True
            if amazonPrice:
                if (amazonPrice * self.coefficient) > usedPrice:
                    bargain = True
        if bargain:
            #We sort the bargains by the used price, so the
            #cheapest items are displayed first.
            bargainsForPrice = bargains.get(usedPrice, None)
            if not bargainsForPrice:
                bargainsForPrice = []
                bargains[usedPrice] = bargainsForPrice
            bargainsForPrice.append((item, amazonPrice))
    return bargains

def getPrice(self, item, priceField):
    """Retrieves the named price field (eg. "OurPrice",
    "UsedPrice", and attempts to parse its currency string into a

```

```

        number."""
        price = getattr(item, priceField, None)
        if price:
            price = self._parseCurrency(price)
        return price

    def _parseCurrency(self, currency):
        """A cheap attempt to parse an amount of currency into a
        floating-point number: Strip out everything but numbers,
        decimal point, and negative sign."""
        return float(self.IRRELEVANT_CURRENCY_CHARACTERS.sub('', currency))
    IRRELEVANT_CURRENCY_CHARACTERS = re.compile("[^0-9.-]")

```

该类并不能正常运行，因为它假设从 `PyAmazon` 得到的查询结果列表(传递给 `getBargains` 的 `items` 参数)与 `Python` 列表的工作方式类似。实际上，AWS 查询结果一共返回 10 页。单个 AWS 查询只返回请求的单个页面，需要一些额外的逻辑在第一页的最后一个项目到第二页的第一个项目之间迭代。

这就是 `OnDemandAmazonList` 出现的原因。该类可以从获取 `PyAmazon` 的同一个网站得到，它将检索后继 AWS 结果页面的复杂性隐藏于类似于 `Python` 列表的接口后面。可以像处理普通列表那样，在 `OnDemandAmazonList` 上迭代，它在后台执行必要的 Web 服务调用，以得到想要的结果。该示例再次证明了 `Python` 在 Web 服务方面的优势：它使隐藏不方便的细节变得简单。

利用 `OnDemandAmazonList`，可以非常简单地在类 `BargainFinder` 中添加一个接口，该接口可以将购物意愿表作为 `OnDemandAmazonList` 进行检索，并通过 `BargainFinder` 运行它，以查找意愿表中有低价二手货的商品。也可以同样简单地使用 `BargainFinder` 查找其他任意 AWS 查询的结果集合中的低价商品，只要确定已经将查询封装在 `OnDemandAmazonList` 中即可：

```

from OnDemandAmazonList import OnDemandAmazonList
def getWishList(subscriptionID, wishListID):
    "Returns an iterable version of the given wish list."
    kwds = {'license_key' : subscriptionID,
            'wishlistID' : wishListID,
            'type' : 'lite'}
    return OnDemandAmazonList(amazon.searchByWishlist, kwds)

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 3:
        print 'Usage: %s [AWS subscription ID] [wish list id]' % sys.argv[0]
        sys.exit(1)
    subscriptionID, wishListID = sys.argv[1:]
    wishList = getWishList(subscriptionID, wishListID)
    BargainFinder().printBargains(wishList)

```

下面是 `WishListBarginFinder` 在我母亲的购物意愿表上运行的结果：

```
# python WishListBargainFinder.py [My subscription ID] 1KT0ATF9MM4FT
Here are items available used for less than $3.00, or for less than 25% of
their Amazon price:
$0.29 (Save 94% off $4.99) Clockwork : Or All Wound Up
$1.99 (Save 68% off $6.29) The Fifth Elephant: A Novel of Discworld
$2.95 (Save 57% off $6.99) Interesting Times (Discworld Novels (Paperback))
$2.96 (Save 52% off $6.29) Jingo: A Novel of Discworld
```

注意:

关于 Amazon 意愿表 ID, 需要注意 WishListBargainFinder 将意愿表 ID 作为命令行输入, 但是意愿表 ID 隐藏于 Amazon Web 应用程序中。为了查找某个人的意愿表 ID, 需要到他或者她的意愿表中, 查看 URL 的 id 域。意愿表 ID 是一个由字母和数字组成的 12 个字符的序列, 例如 BUWBWH9K2H77。

还可以在程序中通过 AWS 调用(使用 ListSearch 操作)查找某个用户的意愿表, 但是由于该方法还未被 PyAmazon 支持, 您必须自己构建 URL 并解析 XML。可以参考 Amazon 网站 <http://aws.amazon.com/resources/> 上的相关示例。

### 20.8.3 向 BittyWiki 提供 REST API

再次回顾一下 BittyWiki, 这是前一节中作为示例 Web 应用程序创建的简单的 wiki 应用程序。根据设计, BittyWiki 已经提供了一个非常简单的 REST API。回忆一下, 除了页面的名称(它总是资源标识符的一部分), 只有两个变量需要考虑: operation 和 data。operation 告诉 BittyWiki 希望对指定的页面所做的操作, data 包含想要放到页面中的数据。现在从机器人的角度考虑该 API。

第一件需要考虑的事是如何确定某个给定的请求来自于人类用户(更准确地说, 一个 Web 浏览器)还是机器人。您也许认为这很简单, 毕竟, 之前所见的 User-Agent HTTP 头应当标识出发出请求的软件。问题是不存在确定的 Web 浏览器列表。新的浏览器和机器人随时都被创建, 而且有些还使用相同的底层库(Python 编写的 Web 浏览器和机器人也许都用到了 urllib)。User-Agent 字符串不能可靠地用作该决策的基础。

大多数 Web 服务通过创建另外一组资源标识符解决此问题, 该组资源标识符对 Web 应用程序使用的资源标识符进行映像, 但是却提供对机器人友好的资源表示。应用程序的“机器人入口”也许是一个完全独立的脚本(app-api.cgi 而不是 app.cgi), 或者是添加在资源标识符(app.cgi/api/foo 而不是 app.cgi/foo)的 PATH\_INFO 之前的一个标准字符串。PATH\_INFO 解决方法得到了一个更易读的资源标识符, 但是 BittyWiki 的 REST Web 服务将作为一个单独的 CGI 实现, 这是因为它易于表示。

下面是与 PUT 和 DELETE 有关的最后一个说明。Web 服务独立于 HTML 表单。尽管 Web 浏览器不支持 HTTP 动词 PUT 和 DELETE, 但是它们被许多(并不是全部)可编程的客户端支持。可以通过引入 PUT 和 DELETE 稍微简化之前的 BittyWiki 接口。这样可以避免使用 operation 参数, 因为它只用来区分 PUT 或者 POST 类型的 POST 请求和 DELETE 类型的 POST 请求。然而, 为了保持与 Web 应用程序的一致性, 而且因为并不是所有可编程的客户端都支持 PUT 和 DELETE, BittyWiki REST Web 服务不采取这种方法。

第二件要考虑的事情是通过外部 API 展示 Web 应用程序的哪些特性比较合理。为什么

有些人希望通过程序访问某个 wiki 的内容？一个 wiki 用户可能会创建两种类型的机器人：

- 一种机器人修改或者创建 wiki 页面。例如，自动化的测试系统向特定的 wiki 页面发送每天的状态报告。
- 另一种机器人检索 wiki 页面，以归档或者镜像一个 wiki，或者将 wiki 页面以 HTML 之外的格式发送到一个终端用户。

第一种类型的机器人需要创建、编辑和删除 wiki 页面。该功能可以保持自身的完整性，并不像 Web 应用程序那样需要在执行请求的操作后，展示一个美观的文档。机器人只需要了解它的请求是否被执行。POST 操作返回的文档只需包含一条状态消息。

这两种机器人都需要从 wiki 中检索页面。它们真正需要的不是页面的 HTML 显示(当使用 GET /bittywiki.cgi/PageName 时得到的对象)，而是原始页面数据(当使用 GET /bittywiki.cgi/PageName?operation=write 时，在编辑框中显示的数据)。第一种类型的机器人需要这种格式的数据，因为它会自己完成显示过程，而且显示原始数据比显示 HTML 更容易。由于类似的原因，第二种类型的机器人也需要这种格式的数据，因为这就是数据在编辑框中显示的格式，也是它在后端存储的方式。

因此，针对机器人的 BittyWiki 的 REST API 与针对 Web 浏览器的 REST API 基本类似。唯一不同的是响应的格式：REST Web 服务输出纯文本文档，而不是易读的 HTML 文档。更加复杂的 REST Web 服务，如 Amazon 的 Web 服务，可能会输出 XML 或者稀疏的 HTML 格式的文档，并期待客户端来解析它们。下面是使用 GET 请求 <http://localhost:8001/cgi-bin/bittywiki-rest.cgi> 得到的纯文本，将它与使用 GET 请求 <http://localhost:8001/cgi-bin/bittywiki.cgi> 得到的 HTML 输出进行比较：

```
This is the home page for my BittyWiki installation.
```

```
Here you can learn about the philosophy and technologies that drive web
applications: REST, CGI, and the PythonLanguage.
```

bittywiki-rest.cgi 的结构与 bittywiki.cgi 类似：

```
#!/usr/bin/python
import cgi
import cgitb
cgitb.enable()
import os
import re
from BittyWiki import Wiki, Page, NotWikiWord
```

```
class WikiRestApiCGI:
```

```
    #The possible operations on a wiki page.
    VIEW = ''
    WRITE = 'write'
    DELETE = 'delete'
```

```
    #The possible response codes this application might return.
```

```

RESPONSE_codes = { 200 : 'OK',
                    400 : 'Bad Request',
                    404 : 'Not Found'}

def __init__(self, wikiBase):
    "Initialize with the given wiki."
    self.wiki = Wiki(wikiBase)

def run(self):
    """Determine the command, dispatch to the appropriate handler,
    and print the results as an XML document."""
    toDisplay = None
    try:
        page = os.environ.get('PATH_INFO', '')
        if page:
            page = page[1:]
            page = self.wiki.getPage(page)
    except NotWikiWord, badName:
        toDisplay = 400, '"%s" is not a valid wiki page name.' % badName

    if not toDisplay:
        form = cgi.FieldStorage()
        operation = form.getfirst('operation', self.VIEW)
        operationMethod = self.OPERATION_METHODS.get(operation)
        if operationMethod:
            if not page.exists() and operation != self.WRITE:
                toDisplay = 404, 'No such page: "%s"' % page.name
            else:
                toDisplay = operationMethod(self, page, form)
        else:
            toDisplay = 400, '"%s" is not a valid operation.' % operation

    #Print the response.
    responseCode, payload = toDisplay
    print('Status: %s %s' % (responseCode,
                            self.RESPONSE_codes.get(responseCode)))
    print('Content-type: text/plain\n')
    print(payload)

```

上面的代码描述了资源和期望的操作，并且将它们(以及提供的所有表示)传递给处理方法。之后显示结果，这一次结果显示为纯文本：

```

def viewOperation(self, page, form=None):
    "Returns the raw text of the given wiki page."
    return 200, page.getText()

def writeOperation(self, page, form):
    "Writes the specified page."
    page.text = form.getfirst('data')
    page.save()

```

```

        return 200, "Page saved."

    def deleteOperation(self, page, format, form=None):
        "Deletes the specified page."
        if not page.exists():
            toDisplay = 404, "You can't delete a page that doesn't exist."
        else:
            page.delete()
            toDisplay = 200, "Page deleted."
        return toDisplay

    #A registry mapping 'operation' keys to methods that perform the
    operations.
    OPERATION_METHODS = { VIEW : viewOperation,
                           WRITE: writeOperation,
                           DELETE: deleteOperation }
```

这 3 个操作处理方法与 `bittywiki.cgi` 中的对应方法类似，但是因为它们产生的数据较少，所以更简单一些。

#### 20.8.4 使用 REST Web 服务实现搜索和替换功能

该 Web 服务对 BittyWiki 有何好处？这里有一个做了一些假设的示例：假设您要求某人为您编写的一个开源项目 Foo 托管 BittyWiki 安装。您创建了大量的 wiki 页面，页面内容中(“Foo is a triphasic degausser for semantic defibrillation”)页面标题中(BenefitsOfFoo、FooDesign 等等)都会反复出现项目名称。一切都很好，直到某天您决定将项目的名称改为 Bar。手动修改 wiki 页面中的这些名称(包括重命名许多页面)会花费大量的时间，而且您没有权力访问 wiki 所在的服务器，因此编写一个爬取整个文件系统的脚本不太可行。那么可以做什么？

此处的 Python 脚本 `WikiSpiderREST.py` 是一个 wiki 搜索和替换蜘蛛。从 wiki 的主页(主页名称是一个 WikiWord)开始，它沿着 WikiWord 链接爬取整个 wiki，并将遇到的某个字符串的所有实例(如“Foo”)替换为另外一个字符串(如“Bar”)。

名称中包含旧字符串的页面(如“FooDesign”)将被删除，并用另外一个名称(如“BarDesign”)的页面将被重新创建。`WikiSpiderREST.py` 记录了它处理过的所有页面，以避免因为陷入循环而浪费时间：

```

#!/usr/bin/python
import re
import urllib

class WikiReplaceSpider:
    "A class for running search-and-replace against a web of wiki pages."

    WIKI_WORD = re.compile('([A-Z][a-z0-9]*){2,}')
```

```

    "Accepts a URL to a BittyWiki REST API."
    self.api = BittyWikiRestAPI(restURL)

def replace(self, find, replace):
    """Spider wiki pages starting at the front page, accessing them
    and changing them via the provided API."""

    processed = {} #Keep track of the pages already processed.
    todo = ['HomePage'] #Start at the front page of the wiki.
    while todo:
        for pageName in todo:
            print('Checking "%s"; % pageName)
            try:
                pageText = self.api.getPage(pageName)
            except RemoteApplicationException, message:
                if str(message).find("No such page") == 0:
                    #Some page mentioned a WikiWord that doesn't exist
                    #yet; not a big deal.
                    pass
                else:
                    #Some other problem; pass it on up.
                    raise RemoteApplicationException, message
            else:
                #This page actually exists; process it.
                #First, find any WikiWords in this page: they may
                #reference other existing pages.
                for wikiWord in self.WIKI_WORD.findall(pageText):
                    linkPage = wikiWord[0]
                    if not processed.get(linkPage) and linkPage not in todo:
                        #We haven't processed this page yet: put it on
                        #the to-do list.
                        todo.append(linkPage)

                #Run the search-and-replace on the page text to get the
                #new text of the page.
                newText = pageText.replace(find, replace)

                #Check to see if this page name matches
                #search and replace. If it does, delete it and
                #recreate it with the new text; otherwise, just
                #save the new text.
                newPageName = pageName.replace(find, replace)
                if newPageName != pageName:
                    print(' Deleting "%s", will recreate as "%s" ' \
                          % (pageName, newPageName))
                    self.api.delete(pageName)
                if newPageName != pageName or newText != pageText:
                    print(' Saving "%s" ' % newPageName
                          self.api.save(newPageName, newText))
                #Mark the new page as processed so we don't go through

```

```

        #it a second time.
        if newPageName != pageName:
            processed[newPageName] = True
        processed[pageName] = True
        todo.remove(pageName)

```

到目前为止，除了对类 `BittyWikiRestAPI` 的引用外，还没有专门针对 REST 的操作。在后续处理中，例如定义类 `BittyWikiRestAPI`，以及其他实现一个到 BittyWiki REST API 的通用 Python 接口的类时，情形将发生变化：

```

class BittyWikiRestAPI:

    "A Python interface to the BittyWiki REST API."

    def __init__(self, restURL):
        "Do all the work starting from the base URL of the REST interface."
        self.base = restURL

    def getPage(self, pageName):
        "Returns the raw markup of the named wiki page."
        return self._doGet(pageName)

    def save(self, pageName, data):
        "Saves the given data to the named wiki page."
        return self._doPost(pageName, { 'operation' : 'write',
                                         'data' : data })

    def delete(self, pageName):
        "Deletes the named wiki page."
        return self._doPost(pageName, { 'operation' : 'delete' })

    def _doGet(self, pageName):
        """Does a generic HTTP GET. Returns the response body, or
        throws an exception if the response code indicates an error."""
        url = self._makeURL(pageName)
        return self.Response(urllib.urlopen(url)).body

    def _doPost(self, pageName, data):
        """Does a generic HTTP POST. Returns the response body, or
        throws an exception if the response code indicates an error."""
        url = self._makeURL(pageName)
        return self.Response(urllib.urlopen(url, urllib.urlencode(data))).body

    def _makeURL(self, pageName):
        "Returns the URL to the named wiki page."
        url = self.base
        if url[-1] != '/':
            url += '/'
        return url + pageName

```

```

class Response:
    """This class handles the HTTP response returned by the REST
    web service."""

    def __init__(self, inHandle):
        self.body = None
        statusCode = None

        info = inHandle.info()
        #The status has automatically been read into an object
        #that also contains all the HTTP headers. The status
        #string looks like '200 OK'
        statusHeader = info['status']
        statusCode = int(statusHeader.split(' ')[0])

        #The remaining data is the plain-text response. In a more
        #complex application, this might be structured text or
        #XML, and at this point it would need to be parsed.
        self.body = inHandle.read()

        #The response codes in the 2xx range are the only good
        #ones. Getting any other response code should result in
        #an exception.
        if statusCode / 100 != 2:
            raise RemoteApplicationException, self.body

class RemoteApplicationException(Exception):
    """A simple exception class for use when the REST API returns an
    error condition."""
    pass

```

BittyWikiRestAPI 类使用 `urllib` 库向 BittyWiki 的 REST 接口 CGI 请求和传递数据。它将响应解释为一条状态消息，一条异常消息，或者请求页面的文本。可以在一个单独的模块中发布该类，以鼓励开发人员使用 Python 编写 BittyWiki 扩展。

注意，`Response` 类定义于类 `BittyWikiRestAPI` 的内部：由于没有其他类要使用它，因此将它放到此处可以保证它在类外部不可见。这是一种可选操作，但是它能使顶层视图更整洁。

最后给出一段实现了蜘蛛的命令行接口的代码：

```

if __name__ == '__main__':
    import sys
    if len(sys.argv) == 4:
        restURL, find, replace = sys.argv[1:]
    else:
        print('Usage: %s [URL to BittyWiki REST API] [find] [replace]' \
              % sys.argv[0])
        sys.exit(1)

```

```
WikiReplaceSpider(restURL).replace(find, replace)
```

**试一试****wiki 搜索和替换**

使用安装的 BittyWiki 创建与某个主题相关的几个 wiki 页面。本例创建了与设想的 Foo 项目相关的几个页面。

运行 WikiSpiderREST.py 命令修改主题。应当会看到类似于下面的输出：

```
$ python WikiSpiderREST.py http://localhost:8001/cgi-bin/bittywiki-rest.cgi
Foo Bar
Checking "HomePage"
  Saving "HomePage"
Checking "FooCaseStudies"
  Deleting "FooCaseStudies", will recreate as "BarCaseStudies"
  Saving "BarCaseStudies"
Checking "CVSRepository"
  Saving "CVSRepository"
Checking "CaseStudy2"
Checking "BenefitsOfFoo"
  Deleting "BenefitsOfFoo", will recreate as "BenefitsOfBar"
  Saving "BenefitsOfBar"
Checking "CaseStudy1"
  Saving "CaseStudy1"
Checking "FooDesign"
  Deleting "FooDesign", will recreate as "BarDesign"
  Saving "BarDesign"
```

注意观察，这些 wiki 页面已经被更改，并且在必要的情况下被重命名。

**示例说明**

WikiSpiderREST.py 保留了一组 WikiWord 用于检查、搜索和替换。为了处理其中的某个 WikiWord，它利用 BittyWiki Web 服务 API 检索相应的页面。如果要检索的页面确实存在，则扫描它的文本，并将它的全部 WikiWord 放入以后要检查的项目列表中。然后使用字符串的搜索和替换功能来修改该页面，并通过 Web 服务 API 保存修改后的内容。如果页面名称包含要被替换的字符串，则删除该页面，并通过 Web 服务 API 创建包含同样内容的新页面。接着将检查列表中的下一个 WikiWord，以此类推。

**注意：**

由于 WikiSpiderREST.py 并不知道不能从主页访问的 wiki 页面，因此它并不一定能找到 wiki 上的所有页面。它只能得到人们从主页开始，单击所有链接后可以看到的页面。

## 20.9 XML-RPC

XML-RPC 协议与 REST 协议的功能相同：利用它编写机器人程序更简单，这些机器

人通过发送 HTTP 请求来访问和/或修改远程应用程序。然而它们也存在一些显著的不同。例如, REST 调用看起来与文档库的操纵类似, 而 XML-RPC 调用则更像函数调用(实际上, 在 Python 实现中, 对 Web 服务的调用被当做函数调用来处理)。REST 向希望检索或者修改的资源发送 GET 或者 POST, 而 XML-RPC 一般要求将用到的调用传递给一个特殊的“服务器”资源。传送的数据包含希望调用的函数的 XML 表示, 以及该函数所需的参数。与 REST 类似, 该调用的响应可能是包含请求的信息、任意状态信息等文档。

BittyWiki 非常简单, 它的输入或者输出都是字符串。在这一方面我们很幸运, 因为字符串是 REST 支持的唯一一种数据类型。如果需要向 REST 应用程序传递一个整型数, 需要将它编码为字符串, 并且要相信资源处理程序知道将它转换回整型数。如果需要传递一个有序列表, 需要知道服务器首选的将一个有序列表表示为一个字符串的方法。一个 REST 应用程序也许会把列表表示为 “item1, item2, item3”; 另外一个应用程序也许会将它们表示为 “item1|item2|item3”; 第三个应用程序还可能把它们表示为一个自定义的 XML 数据结构。REST 最主要的缺点是缺乏将不同的数据类型整理为字符串的标准方式, 也缺乏将字符串整理为某个类型的数据的标准方式。您需要重新学习要使用的 REST Web 服务的请求和响应格式。

下面是一个规范的 XML-RPC 客户端应用程序示例。公共的 XML-RPC 服务器 `betty.userland.com` 提供了一些示例方法, 其中一个方法根据给定的索引返回美国的州名, 该方法将州名组织为以字母表顺序排序的列表:

```
>>> import xmlrpc.client
>>> from xmlrpc.client import ServerProxy
>>> server=xmlrpc.client.ServerProxy("http://betty.userland.com")
>>> server.examples.getStateName(41)
'South Dakota'
```

如果这是一个 REST Web 服务, 列表中的第 41 个州可被当作独立的资源进行访问, 例如 `http://betty.userland.com/StateNames/41`。通过请求适当的资源可以得到某个州的州名。可能有一个 Python 库能够处理请求和响应的细节(就像 PyAmazon 库处理 Amazon Web 服务的细节), 但是类似的库需要为每个 REST Web 服务重写, 因为缺乏数据结构表示相关的 REST 标准。

XML-RPC 与 REST 相比最主要的优点是它提供了一个标准方法, 可以将简单的数据结构编码为请求和响应数据。XML-RPC 指定了不同的 XML 字符串, 可以编码整型数 4、浮点值 4.0、字符串 “4”, 以及仅包含字符串 “4” 的列表。XML-RPC 调用得到的不是一个需要解析的文档, 而是对 `xmlrpc.client` 自动创建的数据结构的描述, `xmlrpc.client` 是 Python 自带的 XML-RPC 库。只使用一个库(`xmlrpc.client`)进行任何类型的 XML-RPC 调用是可能的。

到目前为止, 您也许已经注意到 Python 对于类型并不是过分严格, 它可以将一个类型转换为另一个类型。虽然如此, 它内置的类型涵盖了 XML-RPC 定义的各种类型的表示: 布尔型、整型、浮点型数、字符串、数组和字典。`xmlrpc.client` 为二进制数据和日期提供了包装类。

注意：

有关 XML-RPC 的一个简短和友好的规范位于 [www.xml-rpc.com/spec](http://www.xml-rpc.com/spec)。

### 20.9.1 XML-RPC 请求

XML-RPC 请求的主体是 HTTP POST 请求的主体。它是一个包含 `methodCall` 元素的 XML 文档。`methodCall` 元素又包含两个元素：一个是 `methodName`，它指定了要调用的方法；另一个元素是 `params`，它包含即将传递到该方法的参数列表。

下面是一个 XML-RPC 示例请求，该请求调用一个假想的可以将一系列数以升序或者降序排列的方法：

```
<?xml version="1.1"?>
<methodCall>
  <methodName>searchsort.sortList</methodName>
  <params>
    <param>
      <value>
        <array>
          <data>
            <value><i4>10</i4></value>
            <value><i4>2</i4></value>
          </data>
        </array>
      </param>
      <param><value><boolean>1</boolean></param>
    </params>
  </methodCall>
```

用下面的代码调用假想的本地方法与上述示例 XML-RPC 请求的结果相同：

```
import searchsort
searchsort.sortList([10, 2], True)
```

理解了 `xmlrpc.client` 之后，运行如下代码便可以产生和传递方法请求。

```
import xmlrpc.client
xmlrpc.client.ServerProxy("http://sortserver/RPC").searchsort.sortList
([10, 2], True)
```

### 20.9.2 XML-RPC 中的数据表示

XML-RPC 的 `methodName` 可以是任意字符串，但是 XML-RPC 方法一般被分组到若干个已命名的包中，例如前面示例中的 `searchsort`。在 Python 实现中，`searchsort` 看起来像个模块，包含了用于展示的各种函数，如 `sortList`。XML-RPC 参数可以是表 20-3 中所示的任意类型：

表 20-3

数据类型	示例 XML-RPC 表示
布尔值(True 或者 False)	<boolean>1</boolean>
字符串	<string>James Joyce</string>
整型数	<i4>10</i4>
浮点数	<double>5.1</double>
数组(数组中的元素可以是任意类型,也可以是混合类型)	<pre>&lt; array &gt;   &lt; data &gt;     &lt; value &gt; &lt; i4 &gt; 10 &lt;/i4&gt; &lt;/value&gt;     &lt; value &gt; &lt; i4 &gt; 2 &lt;/i4&gt; &lt;/value&gt;   &lt;/data&gt; &lt;/array&gt;</pre>
字典(键必须是字符串,值可以是任意类型)	<pre>&lt; struct &gt;   &lt; member &gt;     &lt; name &gt; search &lt;/name&gt;   &lt; value &gt; &lt; string &gt; James Joyce &lt;/string&gt; &lt;/value&gt;   &lt;/member&gt;   &lt; member &gt;     &lt; name &gt; channels &lt;/name&gt;     &lt; value &gt; &lt; boolean &gt; 1 &lt;/boolean&gt; &lt;/value&gt;   &lt;/member&gt; &lt;/struct&gt;</pre>
日期和时间	<pre>&lt; dateTime.iso8601 &gt; 20090914T19:11:20 &lt;/dateTime.iso8601&gt;</pre> <p>(使用 xmlrpc.client 的 DateTime 包装类,它可以通过时间元组、从纪元开始的秒数等对象实例化)</p>
二进制数	< base64 > AVRoaXMgaXMgYmluYXJ5IGRhGEu </base64>

强类型语言在处理其中一些数据类型可能会有问题,例如混合类型的数组。动态语言(如 Python)能够很容易地处理这些问题。

20.9.3 XML-RPC 响应

XML-RPC 响应的主体是一个 XML 文档,它描述了 XML-RPC 请求调用的函数的返回值。

假设想象中的 searchsort.sortList 方法的功能与它描述的一样,如果用之前给出的示例主体调用它,它将返回类似于下面的响应:

```
<?xml version="1.1"?>
<methodResponse>
  <params>
    <param>
      <value>
        <array>
          <data>
```

```

    <value><i4>2</i4></value>
    <value><i4>10</i4></value>
  </data>
</array>
</value>
</param>
</params>
</methodResponse>

```

响应与请求的基本结构相同，但是响应更稀疏。它缺少一个 `methodName` 元素，因为响应假定您知道刚才调用了哪个方法。与请求相同，响应也有一个 `params` 元素；请求的 `params` 元素可能包含任意数目的子参数，响应列表则只允许包含一个子参数：返回值。

#### 20.9.4 错误处理机制

REST Web 服务使用 HTTP 状态代码标记错误条件，并结合错误文档描述问题。如您期待的那样，XML-RPC 以更加结构化的方式达到相同的目的。

如果 XML-RPC 服务器不能满足某个请求，它会返回一个包含 `fault` 的响应，而不是仅在 `params` 中包含一个返回值。下面是一个示例错误响应：

```

<?xml version="1.1"?>
<methodResponse>
  <fault>
    <value>
      <struct>
        <member>
          <name>faultCode</name>
          <value><int>4</int></value>
        </member>
        <member>
          <name>faultString</name>
          <value><string>No such method: "searchSort.sortList".</string></value>
        </member>
      </struct>
    </value>
  </fault>
</methodResponse>

```

`fault` 元素描述了一个包含两个成员的 XML-RPC 结构(即 Python 字典)：第一个成员是 `faultString`，它包含对错误的可读描述；第二个成员是 `faultCode`，它等价于用于在 REST 环境下表示失败的 HTTP 状态代码(即使返回一个错误响应的 XML-RPC 调用也将拥有一个 HTTP 代码 200)。`faultCode` 的优点是您可以随心所欲地为自己的应用程序定义特定的错误代码。缺点是，不同于 HTTP 状态代码，关于 `faultCode` 的含义并没有公认的看法。您和用户需要对服务的 `faultCode` 的含义达成一致。

在 Python 中，包含 `fault` 的响应对应于 `xmlrpc.client.Fault` 对象，它是 `Error` 的一个子类。如果使用 Python 的 XML-RPC 库，可以正常地抛出和捕获异常，而不必担心如何创建和解

析 XML-RPC 错误。

### 20.9.5 通过 XML-RPC 展示 BittyWiki API

Python 不仅绑定了一个库，使得编写 XML-RPC 客户端变得容易，还绑定了一个 XML-RPC 服务器。与其他 server 类一样，xmlrpc.server 在自己的端口上作为一个单独的 Web 服务器运行。然而，作为接受 XML-RPC 格式的 HTTP 请求的 CGI 程序，XML-RPC 的功能仍然可用。这在类 CGIXMLRPCRequestHandler 中得以实现，该类的名称可能比 Python 标准库中其他类的名称拥有更多连续的大写字母。

下面的脚本 bittywiki-xmlrpc.cgi 要么通过 XML-RPC CGI(如果没有通过命令行参数调用它，这是支持 CGI 的 Web 服务器采用的调用方式)展示 BittyWiki API，要么通过单独的 XML-RPC 服务器(如果通过命令行中要使用的端口传递它)展示：

注意：

如果正在使用之前所述的 EasyCGIServer，或者另外一个基于 Python 的 CGI HTTPServer 的服务器，将这个脚本作为 CGI 使用并不能正常运行。如果在使用 CGI 的过程中遇到问题，尝试使用另外一个 Web 服务器，例如 Apache，或者运行一个独立的 XML-RPC 服务器，而不是使用 CGI。

```
import sys
import xmlrpc.server
from BittyWiki import Wiki

class BittyWikiAPI:
    """A simple wrapper around the basic BittyWiki functionality we
    want to expose to the API."""

    def __init__(self, wikiBase):
        """Initialize a wiki located in the given directory."""
        self.wiki = Wiki(wikiBase)

    def getPage(self, pageName):
        """Returns the text of the given page."""
        page = self.wiki.getPage(pageName)
        if not page.exists():
            raise NoSuchPage, page.name
        return page.getText()

    def save(self, pageName, newText):
        """Saves a page of the wiki."""
        page = self.wiki.getPage(pageName)
        page.text = newText
        page.save()
        return "Page saved."

    def delete(self, pageName):
```

```

        "Deletes a page of the wiki."
        page = self.wiki.getPage(pageName)
        if not page.exists():
            raise NoSuchPage, pageName
        page.delete()
        return "Page deleted."

class NoSuchPage(Exception):
    pass

```

目前为止，还没有 XML-RPC 特有的地方，而只是对 BittyWiki API 的 3 个基本函数有一个打包好的接口。接下来，试着编写一个函数，向 XML-RPC 展示这 3 个基本的 BittyWiki API 函数。可以用两种方法实现：每次注册一个函数；或者注册一个对象实例，该对象实例可以一次性注册对象的所有方法。该示例提供了这两种注册方法的代码，但是实例注册被注释掉了，因为在 Python 之前的版本中，实例注册方法存在安全隐患：

```

def handlerSetup(handler, api):
    """This function registers the methods of the BittyWiki API
    as functions of an XML-RPC handler."""

    #Register the standard functions used by XML-RPC to advertise which methods
    #are available on a given server.
    handler.register_introspection_functions()

    #Register the BittyWiki API methods as XML-RPC functions in the
    #'bittywiki' namespace.
    handler.register_function(api.getPage, 'bittywiki.getPage')
    handler.register_function(api.save, 'bittywiki.save')
    handler.register_function(api.delete, 'bittywiki.delete')

```

最后是脚本部分，脚本要么启动一个独立的 XML-RPC 服务器来处理任意数量的请求，要么启动一个基于 CGI 的 XML-RPC 脚本以便只处理当前的请求：

```

if __name__ == '__main__':
    WIKI_BASE = 'wiki/'
    api = BittyWikiAPI(WIKI_BASE)
    standalonePort = None
    if len(sys.argv) > 1:
        #The user provided a port number; that means they want to
        #run a standalone server.
        standalonePort = sys.argv[1]
        try:
            standalonePort = int(standalonePort)
        except ValueError:
            #Oops, that wasn't a port number. Chide the user and exit.
            scriptName = sys.argv[0]
            print('Usage:')
            print(' "%s [port number]" to start a standalone server.' \

```

```

        % scriptName)
    print(' "%s" to invoke as a CGI.' % scriptName)
    sys.exit(1)
    isStandalone = 1
    print("Starting up standalone XML-RPC server on port %s." \
        % standalonePort)
    handler = xmlrpc.server.SimpleXMLRPCServer\
        (('localhost', standalonePort))
else:
    #No port number specified; this is a CGI invocation.
    handler = xmlrpc.server.CGIXMLRPCRequestHandler()

    handlerSetup(handler, api)

    if standalonePort:
        handler.serve_forever()
    else:
        handler.handle_request()

```

**试一试****通过 XML-RPC 操作 BittyWiki**

现在可以从其他计算机、甚至其他程序语言对 BittyWiki 进行 XML-RPC 调用，如同以前向 Meerkat(用 PHP 编写)发送 XML-RPC 调用那样。

在一个窗口中，启动一个独立的 XML-RPC 服务器(或者，确保提供 XML-RPC CGI 的 Web 服务器正在运行)：

```

# python BittyWiki-XMLRPC.py 8001
Starting up standalone XML-RPC server on port 8001.

```

在另外一个窗口中，启动一个交互的 Python 会话：

```

>>> import xmlrpc.server
>>> server = xmlrpc.server.ServerProxy("http://localhost:8001/")
>>> bittywiki = server.bittywiki
>>> bittywiki.getPage("CreatedByXMLRPC")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
...
    raise Fault(**self._stack[0])
xmlrpc.server.Fault: <Fault 1: 'No such page:CreatedByXMLRPC'>
>>> bittywiki.save("CreatedByXMLRPC", "This page was created through the XML-
RPC interface.")
'Page saved.'
>>> bittywiki.getPage("CreatedByXMLRPC")
'This page was created through the XML-RPC interface.'

```

您当前正在使用 Web 服务，但是不必编写特殊的客户端代码(除了最开始需要编写连接

到服务器的代码), 甚至不必意识到自己正在使用 Web 服务。当然, 通过接口对 wiki 做出的改变也会显示给正在使用 Web 应用程序或者 BittyWiki 的基于 REST 的 Web 服务的用户。

### 20.9.6 基于 XML-RPC Web 服务的 wiki 搜索和替换

还记得 WikiSpiderREST.py 那个脚本吗? 它爬行 BittyWiki 页面, 并使用 REST API 来执行搜索和替换操作。必须编写一个定制类(BittyWikiRESTAPI)来构造访问 REST 接口要使用的正确的 URL, 还需编写一个定制的 XML 解析器来处理返回的响应文档。当然, 一旦编写了上述代码, 就可以在使用 BittyWiki 的 REST API 的任意应用程序中重用它, 但是 XML-RPC 最主要的优点是不需要编写这些类: xmlrpc.client 能够处理所有事情。按照下面的代码重新编写 WikiSpiderREST.py, 将其另存为 WikiSpiderXMLRPC.py:

```
#!/usr/bin/python
import re
import xmlrpc.client

class WikiReplaceSpider:
    "A class for running search-and-replace against a web of wiki pages."

    WIKI_WORD = re.compile('([A-Z][a-z0-9]*){2,}')
```

```
def __init__(self, rpcURL):
    "Accepts a URL to a BittyWiki XML-RPC API."
    server = xmlrpc.client.ServerProxy(rpcURL)
    self.api = server.bittywiki

def replace(self, find, replace):
    """Spider wiki pages starting at the front page, accessing them
    and changing them via the XML-RPC API."""

    processed = {} #Keep track of the pages already processed.
    todo = ['HomePage'] #Start at the front page of the wiki.
    while todo:
        for pageName in todo:
            print('Checking "%s"' % pageName)
            try:
                pageText = self.api.getPage(pageName)
            except xmlrpc.client.Fault, fault:
                if fault.faultString.find("No such page") == 0:
                    #We tried to access a page that doesn't exist;
                    #not a big deal.
                    pass
                else:
                    #Some other problem; pass it on up.
                    raise xmlrpc.client.Fault, fault
            else:
                #This page actually exists; process it.
```

```

#First, find any WikiWords in this page: they may
#reference other pages.
for wikiWord in self.WIKI_WORD.findall(pageText):
    linkPage = wikiWord[0]
    if not processed.get(linkPage) and linkPage not in todo:
        #We haven't processed this page yet: put it on
        #the to-do list.
        todo.append(linkPage)

#Run the search-and-replace on the page text to get the
#new text of the page.
newText = pageText.replace(find, replace)

#Check to see if this page name matches the search
#string. If it does, delete it and recreate it
#with the new text; otherwise, just save the new
#text in the existing page.
newPageName = pageName.replace(find, replace)
if newPageName != pageName:
    print(' Deleting "%s", will recreate as "%s"' \
          % (pageName, newPageName))
    self.api.delete(pageName)
if newPageName != pageName or newText != pageText:
    print(' Saving "%s"' % newPageName)
    saveResponse = self.api.save(newPageName, newText)
#Mark the new page as processed so we don't go through
#it a second time.
if newPageName != pageName:
    processed[newPageName] = True
processed[pageName] = True
todo.remove(pageName)

```

WikiReplaceSpider 类看起来与之前几乎完全一样。唯一主要的不同是，之前 `api.getPage` 方法位于自己编写的 REST 代码中，而现在它位于已经存在的 `xmlrpclib` 代码中。由于不必实现那些特定的 API 类，WikiReplaceSpider 类几乎就是全部代码：

```

if __name__ == '__main__':
    import sys
    if len(sys.argv) == 4:
        rpcURL, find, replace = sys.argv[1:]
    else:
        print('Usage: %s [URL to BittyWiki XML-RPC API] [find] [replace]' \
              % sys.argv[0])
        sys.exit(1)
    WikiReplaceSpider(rpcURL).replace(find, replace)

```

这个蜘蛛与之前 REST 版本的蜘蛛的工作方式类似，但是它的代码量变少了，因为没有那些专门处理 REST API 的细节的代码了。该脚本可以像 REST 版本的蜘蛛一样运行，但是传入的 URL 是 XML-RPC 接口的 URL，而不是 REST 接口的 URL：

```
$ python WikiSpiderXMLRPC.py http://localhost:8000/cgi-bin/bittywiki-xmlrpc.
cgi Foo Bar
Checking "HomePage"
  Saving "HomePage"
Checking "FooCaseStudies"
...
```

## 20.10 SOAP

XML-RPC 通过定义数据类型(例如整型、日期、列表等)的标准表示方式,解决了 REST 的主要问题。然而,当定义 XML-RPC 时,W3C 的 XML 工作组正在考虑他们自己对那些数据类型以及其他一些数据类型的表示方法。XML-RPC 流行起来之后,W3C 将注意力转向它,并开始重新设计 XML-RPC,使其可以使用 W3C 之前的一些标准。在这个过程中,远大的抱负扩展了该项目的范围,使其包含了任意种类的消息交换,而不仅仅是过程调用和它们的返回值。这就导致了 SOAP 的出现。这个缩略语最开始代表简单对象访问协议 Simple Object Access Protocol,但是由于该标准的范围已经远远超出了简单的远程过程调用,此缩略语已经不再适用。

SOAP 与 COM 或者 CORBA 比起来仍然很简单,但是它比 XML-RPC 要复杂得多。幸运的是,在将 Web 应用程序展示为一个 Web 服务时,并不需要用到 SOAP 的全部特性。确实需要的那部分看起来与 XML-RPC 类似,只不过有一个更加通用的 XML 编码机制。SOAP 允许访问的数据类型多于 XML-RPC 允许访问的数据类型,甚至它还允许自定义数据类型。

遗憾的是,在写本书时,Python 3.1 还没有广泛地支持 SOAP,而有用的第三方模块(例如 SOAPpy)还没有更新到支持当前版本(甚至不支持 Python 2.6)。但是有理由期待在不久的将来,这个局面将会改变。本节展示如何在 Python 2.4 中使用 SOAP(特别是 SOAPpy 模块)。如果希望试一试本节的示例,推荐下载和安装 Python 2.4。否则,继续下面的内容。这些示例与之前的 XML-RPC 示例紧密对应,因此它们不是很复杂。

注意,使用 Python 2.6 及更高版本编写后面的代码时,代码不能正常运行。

### 20.10.1 SOAP 快速入门

与 REST 和 XML-RPC 一样,SOAP 消息一般作为 HTTP POST 请求的一部分发送。因此与其他协议一样,不用任何 SOAP 特定的工具就使用 SOAP Web 服务在技术上是可能的:手动构造一条消息,使用 urllib 将它发送出去,然后利用 xml.sax 模块解析响应。在现实中,如果希望在 Python 中使用 SOAP,首先需要一个 SOAP 库。SOAP 库可以在 Python 的数据结构和 SOAP 的 XML 表示之间进行转换,就像 xmlrpc.client 对于 XML-RPC 那样。

Python 没有自带的“soaplib”,但是您可以自己下载一个。Python 可以使用两个 SOAP 库。本章使用 SOAPpy,它提供了一个类似于 xmlrpc.client 的 SOAP 客户端和服务端版本。

注意:

如果正在运行 Debian GNU/Linux,可以仅安装 soappy 包;否则,可以从 <http://pywebsvcs>.

sourceforge.net/上下载一个发布版本。该网站还提供了 ZSI, 另一个 Python SOAP 包。SOAPpy 还需要其他两个包: 浮点型库 fpconst, 以及 PyXML, 后者是一组 XML 实用工具。关于这两个包的详细信息和链接, SOAPpy 的 README 文件中进行了说明。

### 20.10.2 SOAP 请求

下面是假设的 SOAP RPC 调用的一个副本, 它试图将一个列表以升序排列。将它与之前调用相同方法的 XML-RPC 版本的 XML-RPC 副本进行比较:

```
<?xml version="1.1" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/
encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
  <ns1:sortList xmlns:ns1="urn:SearchSort" SOAP-ENC:root="1">
    <v1 SOAP-ENC:arrayType="xsd:int[2]" xsi:type="SOAP-ENC:Array">
      <item>10</item>
      <item>2</item>
    </v1>
    <v2 xsi:type="xsd:boolean">True</v2>
  </ns1:sortList>
</SOAP-ENV:Envelope>
```

首先要注意的是所有的 xmlns 声明。SOAP 对 XML 名称空间的要求很细致, 而 XML-RPC 不太正式, 只提供单独的 XML 文档。SOAP 使用 XML 名称空间定义 SOAP 消息的格式(SOAP-ENV)、数据类型(例如 xsd:boolean 以及 SOAP 特有的 SOAP-ENC:Array)以及数据类型的概念(xsi:type)。这在数据编码方面给予了 SOAP 极大的灵活性, 但是 XML 模式(xsd)和 SOAP 数据编码机制(SOAP-ENC)的大多数基本数据类型已经定义了。只有在更加复杂的情形下, 才需要自定义数据类型。

这条消息中提到的另一个名称空间是 urn:SearchSort。这是想要调用的方法的名称空间。如同前面提到的, 这与 XML-RPC 版本的请求将方法命名为 searchsort.sortList 而不仅是 sortList 类似。SOAP 对 XML-RPC 约定进行了正式定义, 并使用 XML 名称空间来区分拥有相同名称的不同方法。SOAP 调用必须在一个特定的 XML 名称空间中执行。如果使用 Python SOAP 库进行 SOAP 调用, 这可能是唯一需要关心的名称空间。

如果忽略名称空间, 该条消息看起来与之前的 XML-RPC 消息很类似。这里有一个方法调用标签包含了要传入该方法的所有参数的标签列表。方法调用标签没有包含一个子标签来指示方法名称, 此处方法调用标签根据即将调用的方法命名。在 XML-RPC 中, 参数列在一个单独的 params 标签中, 它们是方法调用标签的直接子标签。SOAP 消息略微简洁一些, 但是(同样没有考虑名称空间声明)同样易读。

比较之前看到的将要排序的数组的 XML-RPC 表示, 与同一个数组的 SOAP 表示:

```

<array>
  <data>
    <value><i4>2</i4></value>
    <value><i4>10</i4></value>
  </data>
</array>
<v1 SOAP-ENC:arrayType="xsd:int[2]" xsi:type="SOAP-ENC:Array">
  <item>10</item>
  <item>2</item>
</v1>

```

这两个协议之间的差异很有典型性。SOAP 中有更多预先作出的定义，还有更多对正式定义数据类型的外部文档的引用。好处是，一旦定义完成，SOAP 需要较少的字节数即可实际定义数据结构。在如上面代码所示的小数组中，这种好处并不明显，但是考虑包含几千个或者几百万个元素的数组。SOAP 在大规模数据结构的表示方面比 XML-RPC 更有效率。

### 20.10.3 SOAP 响应

在向 SOAP 服务器发送了 sortList 请求后，可能得到如下所示的响应：

```

<?xml version="1.1" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
  <ns1:sortList xmlns:ns1="urn:SearchSort" SOAP-ENC:root="1">
    <return SOAP-ENC:arrayType="xsd:int[2]" xsi:type="SOAP-ENC:Array">
      <item>2</item>
      <item>10</item>
    </return>
  </ns1:sortList>
</SOAP-ENV:Envelope>

```

像 XML-RPC 一样，SOAP 响应与 SOAP 请求的基本结构相同。但是 SOAP 请求包含一个参数列表，而 SOAP 响应只有一个返回值。这也与 XML-RPC 相似：XML-RPC 响应包含一个 params 列表，它只允许包含一个 param，即返回值。SOAP 通过删除 params 标签并且只返回一个返回值，使得这条规则更加自然。

### 20.10.4 错误处理机制

如果发送的 SOAP 请求使得服务器代码抛出一个异常，那么返回的响应的主体则会包含一个 Fault 元素。它与下面的响应类似：

```

</SOAP-ENV:Body>
  <SOAP-ENV:Fault SOAP-ENC:root="1">
    <faultcode>SOAP-ENV:Client</faultcode>
    <faultstring>No method urn:SearchSort:sortList found</faultstring>
    <detail xsi:type="xsd:string">
      There's no method "sortList" in the urn:SearchSort namespace.
    </detail>
  </SOAP-ENV:Fault>
</SOAP-ENV:Body>

```

Fault 的子元素 `faultstring` 和 `detail` 是人类可读的描述, `faultcode` 元素则描述了错误类型。XML-RPC 除了要求错误代码必须是整型外, 没有对错误代码作出其他要求, 而 SOAP 定义了 4 个标准字符串作为错误代码。其中的两个错误(`mustUnderstand` 和 `VersionMismatch`)也许不会在基础的 SOAP 应用中碰到。另外两个错误代码标识错误的原因。如果正在编写一个 SOAP 客户端, 并且得到的 `faultcode` 是 `Client`, 这意味着是您引起了该错误(例如, 前面调用指定的名称空间中并不存在的方法而引起错误)。如果 `faultcode` 是 `Server`, 这意味着客户端请求没有任何错误, 但是服务器暂时不能满足该请求, 可能是因为服务器不能访问某个数据库或者其他一些必要的资源。

与 XML-RPC 类似, Python 接口隐藏了包含 Fault 的响应的细节。如果通过 SOAP 展示的 Python 方法抛出一个异常, SOAP 服务器会将异常自动转换为一个包含 Fault 元素的 SOAP 响应。如果正在使用 SOAPpy, 并且调用一个远程方法的响应返回了 Fault, 那么它将被转换为 `Error:SOAPpy.Types.faultType` 的子类。

### 20.10.5 展示一个 BittyWiki 的 SOAP 接口

原则上, 没有理由不可以从一个 CGI 脚本运行 SOAP 服务器: 记住尽管 SOAP 具有额外的复杂性和神秘性, 但是它与 REST 和 XML-RPC 一样, 因为它仅是一个要 POST 到 URL 的文档或者返回的文档。遗憾的是, SOAPpy 并没有提供一个可以处理 SOAP 请求的 CGI 脚本, 而只是提供了一个单独的服务器 `SOAPServer`。

**注意:**

ZSI 是 Python 的另外一个 SOAP 实现, 它提供了基于 CGI 的服务器。

下面的示例脚本 `BittyWiki-SOAPServer.py` 使用单独的服务器提供了 BittyWiki 的 SOAP 接口。该文件应当与 `BittyWiki.py` 放入同一个目录下, 以便可以使用核心的 BittyWiki 引擎。当然, 还可以将 `BittyWiki.py` 放入 `PYTHON_PATH` 中的某个目录下, 这样可以从其他地方使用它:

```

#!/usr/bin/python
import sys
import SOAPpy
from BittyWiki import Wiki

class BittyWikiAPI:

```

## 第III部分 开始使用 Python

```

"""A simple wrapper around the basic BittyWiki functionality we
want to expose to the API."""

def __init__(self, wikiBase):
    "Initialize a wiki located in the given directory."
    self.wiki = Wiki(wikiBase)

def getPage(self, pageName):
    "Returns the text of the given page."
    page = self.wiki.getPage(pageName)
    if not page.exists():
        raise NoSuchPage, page.name
    return page.getText()

def save(self, pageName, newText):
    "Saves a page of the wiki."
    page = self.wiki.getPage(pageName)
    page.text = newText
    page.save()
    return "Page saved."

def delete(self, pageName):
    "Deletes a page of the wiki."
    page = self.wiki.getPage(pageName)
    if not page.exists():
        raise NoSuchPage, page.name
    page.delete()
    return "Page deleted."

class NoSuchPage(Exception):
    """An exception thrown when a caller tries to access a page that
    doesn't exist."""
    pass

```

实际的 API 代码与为 XML-RPC 服务器编写的 API 完全相同，它甚至可以放入一个公共的库中。唯一不同之处在于现在向 SOAPServer 而不是 SimpleXMLRPCServer 注册 API：

```

DEFAULT_PORT = 8002
NAMESPACE = 'urn:BittyWiki'
WIKI_BASE = 'wiki/'
if __name__ == '__main__':
    api = BittyWikiAPI(WIKI_BASE)
    port = DEFAULT_PORT
    if len(sys.argv) > 1:
        port = sys.argv[1]
        try:
            port = int(port)
        except ValueError:
            #Oops, that wasn't a port number. Chide the user and exit.
            print 'Usage: "%s [optional port number]"' % sys.argv[0]

```

```

        sys.exit(1)
    print "Starting up standalone SOAP server on port %s." % port
    handler = SOAPpy.SOAPServer(('localhost', port))
    handler.registerObject(api, NAMESPACE)
    handler.serve_forever()

```

**试一试****通过 SOAP 操作 BittyWiki**

在一个窗口中启动单独的 SOAP 服务器：

```

$ python BittyWiki-SOAPServer.py 8002
Starting up standalone XML-RPC server on port 8002.

```

在另外一个窗口中启动一个交互式 Python 会话：

```

>>> import SOAPpy
>>> bittywiki = SOAPpy.SOAPProxy("http://localhost:8002/", "urn:BittyWiki")
>>> bittywiki.getPage("CreatedBySOAP")
<Fault SOAP-ENV:Server: Method urn:BittyWiki:getPage failed.: __main__.
NoSuchPage CreatedBySOAP>
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
...
SOAPpy.Types.faultType: <Fault SOAP-ENV:Server: Method urn:BittyWiki:getPage
failed.: __main__.NoSuchPage CreatedBySOAP>
>>> bittywiki.save("CreatedBySOAP", "This page was created through the SOAP
interface.")
'Page saved.'
>>> bittywiki.getPage("CreatedBySOAP")
'This page was created through the SOAP interface.'

```

使用隐藏在 SOAPpy 中的 SOAP 与使用隐藏在 xmlrpclib 中的 XML-RPC 的体验类似。可以进行方法调用，传入标准的 Python 对象，并且让库处理所有细节。

### 20.10.6 基于 SOAP Web 服务的 wiki 搜索和替换

下面的 WikiSpiderSOAP.py 是另外一个 wiki 搜索和替换客户端，与之前描述的 BittyWiki 的 REST 和 XML-RPC 接口类似。现在您对这段代码已经很熟悉了。模式总是相同：为基本的 BittyWiki API 创建引用，运行基本的使用 SOAP Web 服务的搜索和替换蜘蛛算法。该版本与 XML-RPC 版本唯一主要的不同在于异常处理的方法不同：xmlrpclib 和 SOAPpy 在服务器端出问题时的处理方法不同，因此异常处理的代码也不同。除此之外，基于 SOAP 的搜索和替换蜘蛛看起来与基于 XML-RPC 的搜索和替换蜘蛛很相似：

```

#!/usr/bin/python
import re
import SOAPpy

```

```

class WikiReplaceSpider:
    "A class for running search-and-replace against a web of wiki pages."

    WIKI_WORD = re.compile('([A-Z][a-z0-9]*){2,})')

    def __init__(self, rpcURL):
        "Accepts a URL to a BittyWiki SOAP API."
        self.api = SOAPpy.SOAPProxy(rpcURL, "urn:BittyWiki")
        self.api.config.dumpSOAPIn=1

    def replace(self, find, replace):
        """Spider wiki pages starting at the front page, accessing them
        and changing them via the XML-RPC API."""

        processed = {} #Keep track of the pages already processed.
        todo = ['HomePage'] #Start at the front page of the wiki.
        while todo:
            for pageName in todo:
                print 'Checking "%s"' % pageName
                try:
                    pageText = self.api.getPage(pageName)
                except SOAPpy.Types.faultType, fault:
                    if fault.detail.find("NoSuchPage") != -1:
                        #Some page mentioned a WikiWord that doesn't exist
                        #yet; not a big deal.
                        pass
                    else:
                        #Some other problem; pass it on up.
                        raise SOAPpy.Types.faultType, fault
                else:
                    #This page actually exists; process it.
                    #First, find any WikiWords in this page: they may
                    #reference other existing pages.
                    for wikiWord in self.WIKI_WORD.findall(pageText):
                        linkPage = wikiWord[0]
                        if not processed.get(linkPage) and linkPage not in todo:
                            #We haven't processed this page yet: put it on
                            #the to-do list.
                            todo.append(linkPage)

                    #Run the search-and-replace on the page text to get the
                    #new text of the page.
                    newText = pageText.replace(find, replace)

                    #Check to see if this page name matches the search
                    #string. If it does, delete it and recreate it
                    #with the new text; otherwise, just save the new
                    #text in the existing page.
                    newPageName = pageName.replace(find, replace)
                    if newPageName != pageName:

```

```

        print ' Deleting "%s", will recreate as "%s"' \
              % (pageName, newPageName)
        self.api.delete(pageName)
    if newPageName != pageName or newText != pageText:
        print ' Saving "%s"' % newPageName
        self.api.save(newPageName, newText)
    #Mark the new page as processed so we don't go through
    #it a second time.
    if newPageName != pageName:
        processed[newPageName] = True
    processed[pageName] = True
    todo.remove(pageName)

if __name__ == '__main__':
    import sys
    if len(sys.argv) == 4:
        rpcURL, find, replace = sys.argv[1:]
    else:
        print 'Usage: %s [URL to BittyWiki SOAP API] [find] [replace]' \
              % sys.argv[0]
        sys.exit(1)
    WikiReplaceSpider(rpcURL).replace(find, replace)

```

这个蜘蛛与本章前面描述的 REST 和 XML-RPC 版本的蜘蛛的工作原理类似:

```

$ python WikiSpiderSOAP.py http://localhost:8002/ Foo Bar
Checking "HomePage"
Saving "HomePage"
Checking "FooCaseStudies"
...

```

由于 BittyWiki-SOAPServer.py 运行自己的 Web 服务器, 所以没有必要指向 Web 服务器上可以处理 SOAP 接口的某个脚本。整个 Web 服务器都是 SOAP 接口。

对 Python 2.4 的介绍至此告一段落, 我们在后面介绍 WSDL 的一节中将继续使用它。

## 20.11 为 Web 服务 API 创建文档

公开一个 Web 服务 API 不会带来任何好处, 除非打算编写机器人的程序员能够知道如何使用该 API。如果发布一个没有足够文档的 Python 模块(这不是一种好做法), 一个有毅力的用户可能会查看源代码, 并在必要时做一些修改, 在不断地尝试中学习, 通过这种方法确定 API 的用法。公开一个 Web 服务时, 这不太可能发生, 因此真正有办法为用户展现 API 信息特别重要。

### 20.11.1 人类可读的 API 文档

不论使用什么样的 Web 服务协议, 保持最新的人类可读的 API 描述都是最重要的。可

以手动编写描述，也可以通过自省和使用 Python 的文档字符串自动生成它。接下来是 3 个示例文档，它们描述了本章创建的 BittyWiki 应用程序的 3 个 Web 服务 API。它们都很短，但是包含用户使用其中任何一种 API 编写应用程序所需的全部信息。

### 1. BittyWiki REST API 文档

为了得到页面“WikiPage”的原始 wiki 标记，请求 URL `http://localhost:8000/cgi-bin/bittywiki-rest.cgi/WikiPage`。您将得到一个 XML 数据结构，其中<data>标签包含了 WikiPage 页面的 wiki 标记。如果 WikiPage 页面不存在，将得到一条错误消息。

为了修改页面 WikiPage 的内容，发送新内容到 URL `http://localhost:8000/cgi-bin/bittywiki-rest.cgi/WikiPage`。将 data 设置为希望向页面中写入的 wiki 标记，并将 operation 设置为字符串 write。您将收到一个 XML 数据结构，其中<message>标签包含一条状态消息。如果 WikiPage 页面不存在，将自动创建该页面。

为了删除页面 WikiPage，发送请求到 URL `http://localhost:8000/cgi-bin/bittywiki-rest.cgi/WikiPage`。将 operation 设置为字符串 delete。您将收到一个 XML 数据结构，其中<message>标签包含一条状态消息。如果 WikiPage 页面不存在，将得到一条错误消息。

### 2. BittyWiki XML-RPC API 文档

BittyWiki API 服务器位于 `http://localhost:8001/`。它公开了以下 3 个方法：

- `bittywiki.getPage(string pageName)`——返回指定页面的文本。传入一个空字符串将指定 wiki 主页。请求一个不存在的页面时，将抛出错误。
- `bittywiki.save(string pageName, string text)`——设置指定页面的文本。如果页面不存在，它将被自动创建。
- `bittywiki.delete(string pageName)`——删除指定页面。试图删除一个不存在的页面会引起错误。

### 3. BittyWiki SOAP API 文档

BittyWiki SOAP 服务器位于 `http://localhost:8002/`。它公开了名称空间 `urn:BittyWiki` 中的 3 个方法：

- `getPage(string pageName)`——返回指定页面的文本。传递一个空字符串将指定 wiki 主页。请求一个不存在的页面将引起一个错误。
- `save(string pageName, string text)`——设置指定页面的文本。如果页面不存在，它将被自动创建。
- `delete(string pageName)`——删除指定页面。试图删除一个并不存在的页面将引起一个错误。

## 20.11.2 XML-RPC 自省 API

非官方的 XML-RPC 规范补充里定义了 `system` 名称空间中的 3 个特殊函数(如表 20-4 中所示)，以方便那些不知道 XML-RPC 服务器支持哪些函数，或者不清楚函数功能的用户。这些特殊函数是 Web 服务中等价于 Python 的 `dir` 和 `help` 命令的函数。`xmlrpc.server` 和

CGIXMLRPCRequestHandler 支持 3 个自省函数中的两个，假设您调用服务器上的 register\_introspection\_functions 方法或者在实例化之后调用处理程序对象。

```
handler=xmlrpc.server.SimpleXMLRPCServer((host,port))
handler.register_introspection_functions()
```

表 20-4

方法名称	方法功能
System.listMethods()	返回服务器支持的所有函数的名称
System.methodHelp(string funcName)	返回包含函数的帮助文档的一个字符串。在 Python 中通过返回函数的文档字符串实现
System.methodSignature(string funcName)	返回指定函数的签名和返回类型。Python 实现并不能自动支持该功能，因为 Python 函数的定义中并不包括类型信息

试一试

使用 XML-RPC 自省 API

启动并连接到 BittyWiki XML-RPC 服务器(或者 CGI)。除了之前所示的 BittyWiki 方法，还可以使用 XML-RPC 自省方法：

```
>>> import xmlrpc.client
>>> server=xmlrpc.client.ServerProxy("http://localhost:8001/")
>>> server.system.listMethods()
['bittywiki.delete', 'bittywiki.getPage', 'bittywiki.save', 'system.
listMethods', 'system.methodHelp', 'system.methodSignature']
>>> server.system.methodHelp("bittywiki.save")
'Saves a page of the wiki.'
>>> server.system.methodSignature("bittywiki.save")
'signatures not supported'
```

XML-RPC 自省并不是用来代替人类可读的 API 文档。一方面，如果使用 API 的用户必须调用 XML-RPC 方法才能知道它们的功能，则很难让他们有兴趣使用 API。然而，自省 API 确定使得从交互的 Python Shell 中实验 XML-RPC Web 服务变得容易许多。

20.11.3 WSDL

许多基于 SOAP 的 Web 服务都在 WSDL 文件中定义它们的接口。WSDL 基本上是本节前面所示的人类可读的 API 文档可被计算机解析的版本。

XML-RPC 定义了一组规则，可以在基本的数据结构与 XML 文档间进行转换。WSDL 允许动态地构造这些规则。它或多或少是用来描述函数的独立于程序语言的机制：函数的名称，参数的数据类型，返回值的数据类型等。尽管 WSDL 与 SOAP 相关，但是不用 WSDL 也可以使用 SOAP(实际上，在本章有关 SOAP 的小节中就是这么做的)。

WSDL 文件也是 XML 文档，它在 definitions 元素中定义了 Web 服务的如下几个方面：

- Web 服务定义的任何自定义数据类型。这些数据类型位于 `types` 列表的 `complexType` 元素中。
- Web 服务发送和接收的消息的格式，即 Web 服务定义的函数的签名和返回值。这在一系列 `message` 元素中定义，还可能引用之前定义的任意自定义数据类型。
- Web 服务提供的函数名称，以及这些函数的输入和输出消息。这在 `portType` 元素中定义，该元素包含每个 Web 服务函数的 `operation` 元素。
- Web 服务函数到特定协议(即 HTTP)的绑定。对于简单的 SOAP 应用程序，本节略显冗余：您最终只是再次列出所有的函数。它之所以存在，是因为 SOAP 是独立于协议的，您需要显式地表明要通过 HTTP 公开这些方法。这些内容都位于 `binding` 元素中。
- 最后是 Web 服务的 URL。这定义在 `service` 元素中。

注意，这里再次使用 SOAP，而 SOAP 库(在编写本书时)还没有更新到可以在 Python 2.6 或者 3.0 中使用，所以仍然需要 Python 2.4 来运行下面的示例。下面是 BittyWiki.wsdl，它是描述 BittyWiki 公开的 SOAP API 的 WSDL 文件：

```
<?xml version="1.1"?>
<definitions name="BittyWiki"
    targetNamespace="urn:BittyWiki"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">

  <!--Descriptions of the functions exposed by the BittyWiki API. The
  definitions of the functions reference message elements which will be
  defined afterwards.-->
  <portType name="BittyWikiPortType">
    <operation name="getPage">
      <input message="sendPageName"/>
      <output message="getPageText"/>
    </operation>

    <operation name="save">
      <input message="sendPageNameAndText"/>
      <output message="getStatusMessage"/>
    </operation>

    <operation name="delete">
      <input message="sendPageName"/>
      <output message="getStatusMessage"/>
    </operation>
  </portType>
```

WSDL 解析器现在知道 BittyWiki 公开了哪些函数，但是不知道这些函数的签名和返回类型。下面列出了这些内容：

```

<!--Descriptions of the method signatures used by the BittyWiki API.
For instance, this first one is for a method where you send in a page name.
This method signature is common to getPage() and delete().-->
<message name="sendPageName">
  <part name="pageName" type="xsd:string"/>
</message>

<message name="sendPageNameAndText">
  <part name="pageName" type="xsd:string"/>
  <part name="pageText" type="xsd:string"/>
</message>

<!--Descriptions of the possible return values obtained from the
BittyWiki API. The first one is for a return value that contains
a wiki page's markup: that is, the return value of getPage().-->
<message name="getPageText">
  <part name="pageText" type="xsd:string"/>
</message>

<message name="getStatusMessage">
  <part name="message" type="xsd:string"/>
</message>

```

下面的代码有些冗余，因为 4 个 SOAP 函数通过 HTTP 绑定到了 SOAP:

```

<!--A binding of the BittyWiki API functions (previously defined only
in the abstract) to the specific "SOAP-over-HTTP" protocol.-->
<binding type="BittyWikiPortType" name="BittyWikiSOAPBinding">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
  <operation name="getPage">
    <input><soap:body use="literal" namespace="urn:BittyWiki" /></input>
    <output><soap:body use="literal" namespace="urn:BittyWiki" /></output>
  </operation>

  <operation name="save">
    <input><soap:body use="literal" namespace="urn:BittyWiki" /></input>
    <output><soap:body use="literal" namespace="urn:BittyWiki" /></output>
  </operation>

  <operation name="delete">
    <input><soap:body use="literal" namespace="urn:BittyWiki" /></input>
    <output><soap:body use="literal" namespace="urn:BittyWiki" /></output>
  </operation>
</binding>

```

最后一段代码告知 WSDL 在哪里寻找 BittyWiki Web 服务代码:

```

<!--A link to the BittyWiki web service on the web. It uses the
BittyWiki API defined in BittyWikiPortType, as realized by its
SOAP-over-HTTP binding, BittyWikiSOAPBinding.-->

```

```
<service name="BittyWiki">
  <port name="BittyWikiPort" binding="BittyWikiSOAPBinding">
    <soap:address location="http://localhost:8002/" />
  </port>
</service>
</definitions>
```

**注意:**

BittyWiki API 没有定义任何自定义数据类型, 因此它的 WSDL 文件中没有任何 types 元素。如果您希望看到拥有 complexTypes 的 types 元素, 请查看 Google Web API 的 WSDL 文件。

WSDL 非常复杂: WSDL 文件比实现它描述的 Web 服务的 Python 脚本还要庞大。WSDL 文件通常由相应的 Web 服务源代码自动生成, 因此人们不必指定它们。但是 Python 代码却不能自动生成 WSDL, 因为 WSDL 的一部分重要内容都是在定义数据类型, 而 Python 函数却没有预定义的数据类型。SOAPpy 和 ZSI 库都可以解析 WSDL(实际上, 它们共享一个 WSDL 库: wstools), 但是 Python 特定的资源却没有办法可以生成 WSDL。

**试一试****通过 WSDL 代理操作 BittyWiki**

下面的代码与之前直接通过 SOAP 调用操作 BittyWiki 的示例很类似:

```
>>> import SOAPpy
>>> proxy = SOAPpy.WSDL.Proxy(open("BittyWiki.wsdl"))
>>> proxy.getPage("SOAPViaWSDL")
<Fault SOAP-ENV:Server: Method urn:BittyWiki:getPage failed.: __main__.
NoSuchPage SOAPViaWSDL>
Traceback (most recent call last):
...
SOAPpy.Types.faultType: <Fault SOAP-ENV:Server: Method urn:BittyWiki:getPage
failed.: __main__.NoSuchPage SOAPViaWSDL>
>>> proxy.save("SOAPViaWSDL", "This page created through SOAP via WSDL.")
'Page saved.'
>>> proxy.getPage("SOAPViaWSDL")
'This page created through SOAP via WSDL.'
```

主要的不同在于 WSDL 会阻止调用并不存在的 Web 服务方法:

```
>>> proxy.noSuchMethod()
Traceback (most recent call last):
...
AttributeError: noSuchMethod
>>>
>>> server = SOAPpy.SOAPProxy("http://localhost:8002/", "urn:BittyWiki")
>>> server.noSuchMethod()
<Fault SOAP-ENV:Client: No method urn:BittyWiki:noSuchMethod found:
exceptions.AttributeError BittyWikiAPI instance has no attribute
```

```
'noSuchMethod'>
Traceback (most recent call last):
...
SOAPpy.Types.faultType: <Fault SOAP-ENV:Client: No method urn:BittyWiki:
noSuchMethod found: exceptions.AttributeError BittyWikiAPI instance has no
attribute 'noSuchMethod'>
```

在这两种操作 BittyWiki 的方法中，调用 `noSuchMethod` 都会引起异常，但是 WSDL 可以在本地计算机上捕获该异常，而不是在服务器上捕获它。这种能力对于编译语言尤其有用：WSDL 可以像对本地函数调用那样，对 Web 服务调用执行相同的编译时检查。

本章中对 Python 2.4 的使用至此结束。

## 20.12 选择 Web 服务标准

本章描述了 Web 服务的 3 个标准，每个标准都有不同的原理，也都有各自的优缺点。REST 致力于尽量挖掘 HTTP 提供的实用程序的最大潜能，但是它缺乏数据类型的编码标准。XML-RPC 提供了编码标准，但是它有些冗长，而且只处理简单的数据类型以及简单数据类型的组合。SOAP 既提供了 XML-RPC 的结构化的数据类型，也提供了 REST 的灵活性，但是它的复杂性使得比较难处理的情形变得比仅用 REST 实现更难以理解。

业界更倾向于 REST 和 SOAP，而不是 XML-RPC。SOAP 受到了大型软件公司(例如 IBM 和 Microsoft)的支持，REST 有独立的 Web 服务用户和开发人员的支持。这是因为基于 REST 和 XML-RPC 的 API 更易于学习和使用。每当 Web 服务通过不同的协议公开相同的 API 时，最简单的那个都将胜出。例如，除了本章中介绍的 REST API 之外，Amazon 还公开了一个 SOAP API，但是 80% 的用户都选择 REST 而不是 SOAP。

应该选择哪个标准？如果是大软件公司(例如 IBM 和 Microsoft)的忠实拥护者，您可能从一开始就不会选择 Python。您可能会选择使用 Java 或者 .NET：这两种强类型语言对 SOAP 工具有良好的支持。在大多数情形下，并不需要 SOAP 的额外功能，而且 Python 对于 SOAP 的支持由于增加的复杂性而显得不很完善，所以既然没有必要，为什么还要选择它呢？

应当从展示一个设计良好的 REST 或者 XML-RPC API 开始。如果在设计或者实现阶段，自己的选择遇到问题，试着使用 SOAP(一旦 SOAP 库被更新)。除非您正在编写任务繁重的业务流程自动管理软件，或者结合静态类型语言，例如 Java 或者 .NET，您可能会始终使用 REST 或者 XML-RPC API。用户会感谢您提供的简单的接口。

理想的 Web 服务应当拥有符合 REST 的接口，其中每个资源都可以接收具备 XML-RPC (或者 SOAP 的简单子集)定义的格式的 POST 数据。然后可以按照 REST 的原则设计 Web 服务，但是 `xmlrpc.client` 或者 `SOAPpy` 的一些变体可被用于整理或者打乱数据，而不要求创建定制的解析器。

不论选择哪个标准，请在一开始设计时就将在 Web 服务牢记于心：Web 服务是为机器人设计的 Web 应用程序。如果希望应用程序能够激发创造力，而不仅仅是满足事先定义的需求，必须将一些控制权移交给用户。

## 20.13 Web 服务礼仪

Web 服务的有些用户可能会绕开规则，管理员也可能感觉用户对提供给他们的服务不领情。为了保持和谐性，这里有几点关于管理 Web 服务的建议。

### 20.13.1 Web 服务的使用者礼仪

如果编写了一个机器人来使用其他人的 Web 服务，遵守规则很重要。特别是，不要尝试回避一些限制，例如许可证或者每天对于访问 API 的次数的限制。访问 Web 服务是一种特殊荣幸，而不是权利。用完 API 调用，并在以后完成任务，哪怕比原计划的时间晚，这也比被完全禁止访问 API 好。

### 20.13.2 Web 服务的创建者礼仪

如果打算通过 Web 服务展示 Web 应用程序，需要考虑这些问题的另一面。如果受众已经脚本化您的应用程序，您已经有了优势，因为不必再去猜测人们可以用它做什么。在设计 Web 服务之前，调查机器人的编写者们，看看他们到底使用应用程序中的哪部分。尽量使您的 Web 服务方便用户转移到新系统，否则他们将没有动力做此转换。

作为公共的 Web 服务的创建者，您可能认为 Web 礼仪的重担完全落到了用户身上。毕竟，您为他们提供了服务，而并没有期待任何回报。然而，让您的应用程序的使用条件变得让大家认可很重要，因为编写机器人的用户具有最终的优势：只要您提供了一个 Web 应用程序，并且其功能与 Web 服务相同，有决心的用户总是可以通过各种方式编写使用 Web 应用程序的机器人。并没有万无一失的方法可以区分访问站点的机器人和用户访问站点所用的浏览器。它们都是运行在其他人的计算机上发送 HTTP 请求的软件。所有的 HTTP 头，包括 User-Agent 以及身份验证头，机器人都可以伪造。

虽然如此，如果一个特定的机器人带来了麻烦，可以用对付麻烦的人类用户那样解决机器人带来的问题。

### 20.13.3 将 Web 应用程序作为 Web 服务使用

可以编写将 Web 应用程序作为 Web 服务使用的脚本。毕竟，这也是 Web 服务的设计思想。一些 Web 站点仍旧没有领会 Web 服务的精神，或者提供了一些没有展示您需要的功能的 Web 服务。为了编写设想的机器人，需要借助 Web 应用程序。

本章并没有介绍如何编写这样的脚本，但是编写它的一般原则与 Web 服务类似。如果这个主题让您感兴趣，最终您将会这么做。此时，不要违反站点的任何服务条款。另外，不要过度访问站点。如果可以，在空闲时间运行您的脚本，这样不会给系统增加负担。最后，请求站点的管理员提供 Web 服务的接口，以便可以使用一个占用较少带宽的更加稳定的接口。

## 20.14 本章小结

Web 应用程序非常强大，也很流行，利用 Python 可以轻松地编写 Web 应用程序。REST

架构使 Web 变得易于使用和成功：利用 REST 架构设计应用程序可以让您在一开始就具有优势。Web 应用程序是为人类设计的，而 Web 服务则是为软件脚本设计的 Web 应用程序。为了简单性和易用性，可以公开 REST 和 XML-RPC Web 服务；如果要设计高负荷的任务，或者结合使用 Java 或者 .NET 应用程序，则需要使用 SOAP。还可以使用他人提供的 Web 服务：他们为了大家的利益，开放了自己的数据集和算法。

## 20.15 习题

1. 如何利用 REST 方式修改 BittyWiki，使其可以支持多个 wiki？
2. 为 WishListBargainFinder.py 编写一个 Web 应用程序接口(即委托给 Amazon Web Services 的 Web 应用程序)。
3. wiki 搜索和替换蜘蛛查找它遇到的每个新的 WikiWord，以查看它是否对应于某个 wiki 页面。如果它通过名称找到了页面，该页面将被处理。否则，什么都不会发生，蜘蛛则会浪费一次 Web 服务请求。如何修改 Web 服务 API，以避免蜘蛛对不存在的页面所做的多余的 Web 请求？
4. 假设为了避免破坏行为，您修改了 BittyWiki，使 wiki 上的页面不能被删除。遗憾的是，这会破坏 wiki 的搜索和替换蜘蛛，因为有时它需要先删除页面，然后使用新名称重新创建页面。如何才能找到既能满足您的需求又能满足蜘蛛需求的解决方案？





# 第 21 章

## 集成 Java 与 Python

Java 是一种面向对象的程序语言。Java 程序从源代码编译成字节码。Java 运行时引擎称为 Java 虚拟机(Java virtual machine, JVM), 它负责运行编译的字节码。是不是感觉很熟悉? 至少在抽象层次上, Java 和 Python 非常类似。与 Java 类似, Python 程序也编译为字节码, 尽管这个编译的过程也可以在运行时完成。

尽管有这些相似的地方, 但这两种语言还是有区别的:

- 在 Python 中, 可以直接从源代码运行脚本。编译是可选的。如果不提前编译 Python 代码, `python` 命令将完成这个任务。
- Java 的语法是基于 C 和 C++ 这两种非常流行的程序语言的, 这使得使用 C++ 的开发人员可以很方便地迁移到 Java。因此, 人们认为 Java 比 Python 更严肃且更适合商用。
- Python 的语法简单易学, 但语法已远远偏离了 C 语言。
- 由于语法简单, 而且内置了对列表、字典和元组的支持, 因此使用 Python 编写代码比使用 Java 简单得多。一般来说, Python 程序的代码量比相应的 Java 程序代码量少得多。
- 在标准 API 上, Java 比 Python 有优势。基本的 Java 语言包含成熟的数据库 API、解析 XML 文档的 API、远程通信的 API, 甚至包含访问 LDAP 目录服务器的 API。尽管也可以在 Python 中完成这些, 但是 Python 缺乏众多 Java API 的丰富性和标准性。通过 Python 编写企业级应用程序的时候这一点更加明显。Java 的企业级 API 称为 Java EE, 它使得 Java 在企业市场中占据了一席之地。而遗憾的是, Python 在企业市场中的地位却很低。

在编写企业级应用程序时, 很可能需要通过 Java 来编写。尽管 Python 可以在这个领域工作得很好, 但是 Java 还是控制了企业市场的关注。幸运的是, 通过 Jython, 可以同时获得这两种语言的好处。Jython 是用 Java 实现的 Python。

通过 Jython, 可以在 Java 虚拟机中运行 Python 代码, 也就是说可以从任何 Java 应用程序中运行 Python 代码。

本章将介绍:

- 为什么要在 Java 应用程序中使用脚本。
- 比较 Jython 与常规的基于 C 的 Python 实现。
- Jython 的安装。

- 在 Jython 中运行 Python 脚本。
- 在 Python 脚本中调用 Java 代码。
- 通过 Python 类扩展 Java 类。
- 通过 Python 编写 Java EE servlet。
- 在 Java 应用程序中嵌入 Jython 解释器。

请注意，熟悉 Java 和 Python 才能集成 Python 和 Java。

## 21.1 在 Java 应用程序中编写脚本

大多数软件开发人员认为 Java 是一个大型的系统编程语言，一个适合严肃任务的严肃语言。以同样的方式思考的话，Python 来自脚本语言(例如 Perl 和 Tcl)的阵营。因此，很多开发人员都不尊重 Python 语言，因为脚本语言当然都是为了不会编程的人创建的。您当然知道其实并不是这样，但是编程语言和脚本语言之间的差距依然存在，尽管 Python 能够很优雅地缩小两者之间的这种差距。

尽管不受尊重，但是很多脚本语言都被证明具有非常高的生产效率，而且在很多小公司和大公司的关键部门都有部署。通常，与使用 Java 等系统语言相比，使用脚本语言可以用更少的代码在更短的时间内完成更多的任务。

在 Java 应用程序中，脚本语言也越来越派上用场了，原因如下：

- 脚本语言可以作为一个宏扩展语言。就像可以通过 Visual Basic for Application (VBA)编写脚本扩展 Microsoft Office 的功能一样，也可以通过 Jython 扩展 Java 应用程序的功能。有一些复杂的文本编辑器，例如 jEdit([www.jedit.org](http://www.jedit.org))，可以以同样的方式编写脚本。
- 通过 Jython 加快 Java 应用程序的开发。作为一种高级的脚本语言，Python 具有高效率的优势。
- 探索和调试运行的系统。使用 Jython 的交互式功能，可以探索运行着的 Java 应用程序。可以按照需要交互地执行代码。在 Python 中您已经对这种交互性习以为常了，但是 Java 并没有这种能力。
- 使用脚本编写单元测试的速度大大高于用 Java 编写的速度。很多机构不习惯引入脚本语言，特别是开源的脚本语言。测试的时候使用脚本语言可以发挥脚本语言的优势，避免在应用程序中发布脚本语言包，还可以避免在生产中使用脚本语言包。
- 除了可以进行单元测试之外，脚本语言也可以很好地用于全面的系统测试。有一个叫做 Grinder 的系统语言包就使用 Jython 来创建测试脚本。有关 Grinder 的详细信息请访问 <http://grinder.sourceforge.net/>。
- 可以为类似数据迁移这样的任务创建一次性的脚本。如果只需要更新数据库表中的某一行，或者修复某个设置，使用脚本就可以快很多。
- 可以扩展企业级应用程序，而不必重新部署应用程序。脚本语言可以方便地保证系统持续运行。此外，开发人员可以扩展系统的功能，而不需要安全权限来对应用程序进行重新部署。

通过使用基于非常流行的 Python 语言的 Jython, 可以完成所有这些任务, 甚至更多任务。

## 21.2 比较各种 Python 实现

传统的 Python 实现通常称为 C-Python, 可以在大量平台上编译执行, 包括 Windows、Linux 和 Mac OS X。C-Python 使用 C 语言编写。基于 Java 的 Jython 可以在任何支持 Java 虚拟机的平台上编译和执行, 包括 Windows、Linux 和 Mac OS X。从这个角度说, 两种 Python 实现的跨平台性非常类似。

然而, 比起传统的 C-Python 实现, Jython 并没有实现最新功能。C-Python 实现支持还未在 Java 实现中编写的新功能。这是可以理解的, 因为 C-Python 是新功能最初开发的地方, 而 Jython 开发人员必须用 Java 重新实现每一项 Python 功能。

使用 C-Python 还是 Jython 编写 Python 脚本都不重要, 因为两者都支持 Python 语言。在一般情况下, 除非有在 Java 平台下工作的特别需要, 否则都应该使用 C-Python。当需要在 Java 平台下工作时, 很显然, 使用 Jython!

本章的其余部分将介绍如何使用 Jython。

## 21.3 安装 Jython

作为一个开源项目, Jython 没有遵守固定的发布周期。最好的选择是从 [www.jython.org](http://www.jython.org) 下载最新版本。然后, 遵循该 Web 站点上的说明安装 Jython。

较旧版本的 Jython, 如 2.1 版, 被打包为安装程序的 Java .class 文件。运行这个文件的时候, 安装程序将 Jython 安装在硬盘上。较新的预发布版本被打包成一个 Zip 文件。解压缩这个文件以安装 Jython。

安装 Jython 后, 在 Jython 安装目录下可以得到两个可执行脚本: `jython` 和 `jythonc`, 作用类似于 `python` 和 `pythonc`。`jythonc` 脚本的作用是将 Python 代码编译为 Java 的 .class 文件。需要将 `jython` 脚本放在路径中, 或者其他可以调用它的位置。

注意:

在 Windows 上, 对应的是 DOS 批处理文件 `jython.bat` 和 `jythonc.bat`。

## 21.4 运行 Jython

`jython` 脚本运行 Jython 解释器。`jythonc` 脚本运行 Jython 编译器, 将 Python 代码编译为 Java 的 .class 文件。在大多数情况下, 应该使用 `jython` 脚本来运行 Jython。

### 21.4.1 交互地运行 Jython

就像 Python 的 `python` 命令一样, Jython 也支持交互模式。在这种模式下, 可以输入 Jython 的表达式。Jython 的表达式基本上与 Python 的表达式相同, 但是还可以在其中集成 Java。

要运行 Jython 解释器，运行 jython 脚本(或者 Windows 上的 jython.bat)。

### 试一试

### 运行 Jython 解释器

运行解释器，输入下列表达式：

```
>>> 44 / 11
4
>>> 324 / 101
3
>>> 324.0 / 101.0
3.207920792079208
>>> 324.0 / 101
3.207920792079208
>>> import sys
>>> sys.executable
'C:\\jython2.5.0\\jython.bat'
>>> sys.platform
'java1.6.0_03'
>>> sys.version_info
(2, 5, 0, 'final', 0)
>>>
```

### 示例说明

正如这个例子所示，Jython 解释器的行为看上去与 Python 解释器很类似。这正符合我们的期望，因为 Jython 本来就是在 Java 平台上实现的 Python 语言。

数学运算应该和 Python 基本一致(“基本一致”是指一些浮点运算会产生略有不同的结果)。另外请注意，这个例子使用的是 Jython 2.5。

在同一平台上，您可以观察在 C-Python 解释器的 python 命令上运行相同表达式的区别。例如：

```
>>> 44 / 11
4
>>> 324 / 101
3
>>> 324.0 / 101.0
3.2079207920792081
>>> 324.0 / 101
3.2079207920792081
>>> import sys
>>> sys.executable
'C:\\Python31\\pythonw.exe'
>>> sys.platform
win32
>>> sys.version_info
(3, 1, 0, 'final', 0)
>>>
```



## 21.4.2 运行 Jython 脚本

和 python 命令一样, jython 也可以运行脚本, 如下例所示。

### 试一试

### 运行一个 Python 脚本

输入以下简单的脚本, 并命名该文件为 jsys.py:

```
import sys

print('Python sys.path:')
print(sys.path)

print('Script arguments are:')
print(sys.argv)
```

当通过 jython 运行此脚本时, 应该可以看到类似下面的输出:

```
Python sys.path:
['', ' C:\\jython2.5.0\\LIB', '__classpath__', '__pyclasspath__/',
'C:\\jython2.5.0\\LIB\\site-packages']
Script arguments are:
['']
```

如果将 Jython 安装在其他目录, 文件路径会有所不同。

### 示例说明

相比标准的 C-Python, sys.path 属性保存的目录很少。例如, 可以通过 python 运行同一脚本, 如下所示:

```
Python sys.path:
['C:\\Python31\\Lib\\idlelib', C:\\Windows\\system32\\python31.zip',
'C:\\Python31\\DLLs', 'C:\\Python31\\lib', 'C:\\Python31\\lib\\plat-win',
'C:\\Python31', 'C:\\Python31\\lib\\site-packages']
```

在这种情况下, 注意 sys.path 属性保存的目录很多。

### 注意:

这些例子都运行在 Windows Vista 上。运行在其他操作系统上时, 路径会有不同。

您还会注意到, jython 运行的脚本的启动时间比 python 运行的脚本的启动时间长得多。这是因为启动 java 命令并加载整个 Java 环境需要的时间较长。

## 21.4.3 控制 jython 脚本

jython 脚本本身只是 java 命令的简单包装器。jython 脚本设置 Java 的 classpath 和

`python.home` 属性。也可以传递参数到 `jython` 脚本来控制 `Jython` 的运行方式，还可以向自己的脚本传递参数。`jython` 命令行的基本格式如下：

```
jython jython_arguments what_to_run arguments_for_your_script
```

`jython_arguments` 可以为 `-S`，表示 `Jython` 启动的时候不导入 `site`，还可以为 `-i`，表示以交互式的方式运行 `Jython`。还可以传入 Java 系统属性，这些属性会传入您自己的 `Jython` 脚本。传递 Java 系统属性的格式为 `-Dproperty=value`，这是在命令行传递属性设置的标准 Java 格式。

通常将 `Jython` 脚本文件的名称放在命令的 `what_to_run` 位置。当然，`jython` 还有很多选项，如表 21-1 所示。

表 21-1

选 项	意 义
<code>filename.py</code>	运行给定的 <code>Jython</code> 脚本文件
<code>-c command</code>	运行命令行传递的命令字符串
<code>-jar jarfile</code>	运行指定的 <code>jar</code> 文件中的 <code>Jython</code> 脚本 <code>__run__.py</code>
<code>-</code>	从标准输入读取命令执行。这样可以通过管道向 <code>Jython</code> 解释器传送 <code>Jython</code> 命令

可以选择表 21-1 中列出的任意一种方法。

此外，`arguments_for_your_script` 表示要传递给脚本的参数。这些参数将被设置为 `sys.argv[1:]`。

21.4.4 创建可执行命令

注意，由于 `jython` 是一个脚本，因此不可以使用传统的 `shebang` 注释行来运行 `Jython` 脚本(在 UNIX 和 Linux 系统中，`shebang` 指的是以 `#` 开头、后面跟着感叹号的行。这一行告诉系统这是运行这个程序需要调用的命令)。例如，使用 `Python` 脚本的时候，可以将下面的一行作为脚本的第一行：

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

如果这是脚本的第一行，而且这个脚本具有执行权限，那么操作系统可以将 `Python` 脚本作为命令运行。

注意：

Windows 是唯一的例外。Windows 采用不同的方法将 `Python` 解释器和以 `.py` 结尾的文件关联起来。

然而使用 `Jython` 脚本的时候，不能利用这种机制。这是因为很多操作系统要求运行脚本的程序本身是一个二进制可执行文件，而不是一个脚本文件。也就是说，您编写了一个脚本，想要通过 `jython` 脚本来执行。

为了解决这个问题，使用 `env` 命令。例如，修改 `shebang` 行如下：

```
#!/usr/bin/env jython
```

为了让这一行生效，必须将 `jython` 脚本加入路径。

### 试一试

### 创建一个可执行脚本

将下面的行插入前面的 `jysys.py` 脚本。插入的新行用粗体表示。

```
#!/usr/bin/env jython

import sys

print('Python sys.path:')
print(sys.path)

print('Script arguments are:')
print(sys.argv)
```

将这个新文件保存为 `jysys`，不带扩展名。通过 `chmod` 命令添加执行权限，如下例所示：

```
$ chmod a+x jysys
```

然后，可以运行这个新的命令：

```
$ ./jysys 1 2 3 4
Python sys.path:
['', 'C:\\jython2.5.0\\LIB', '__classpath__', '__pyclasspath__/',
'C:\\jython2.5.0\\LIB\\site-packages']
Script arguments are:
['./jysys', '1', '2', '3', '4']
```

#### 示例说明

这样，`shebang` 注释也可以用于 `Jython` 了，就像适用于其他脚本语言一样。`Jython` 唯一的麻烦之处就在于 `jython` 命令本身是一个调用 `java` 命令的脚本。

在下一节中将更详细地介绍 `java` 命令是如何运行 `Jython` 脚本的。

## 21.5 独立运行 Jython

不需要使用 `jython` 脚本来执行 `Jython` 脚本。可以像调用其他任何 `Java` 应用程序一样调用 `Jython` 解释器。

`jython` 脚本本身非常短小。大部分操作都是通过使用一大堆参数调用 `java` 命令实现的，为了清晰说明，将其分解如下：

```
java -Dpython.home="C:\\jython2.5.0\\" \
    -classpath C:\\jython2.5.0\\jython.jar:$CLASSPATH" \
    "org.python.util.jython" "$@"
```

安装 Jython 的目录不同，文件路径也会有所不同。jython 脚本只不过是运行 jar 文件 jython.jar(脚本将这个文件加入 Java 的 classpath)中的 org.python.util.jython 类。这个脚本还设置了 python.home 系统属性，帮助 Jython 找到所需的支持文件。

要独立运行 Jython，只需要确保 jython.jar 在 classpath 中即可。执行解释器类，如 org.python.util.jython。此外，还需要设置 python.home 系统属性。

还需要确保在每个将要运行 Jython 脚本的系统上都正确安装了 Jython。

## 21.6 打包基于 Jython 的应用程序

Jython 不是独立的系统。它需要组成 Jython 库的大量 Python 脚本才能运作。因此，需要包含 jython.jar 文件和 Jython 的库文件。至少，需要包含 Jython 发布版中的 Lib 和 cachedir 目录。

**注意：**

Jython 需要有权限写入 cachedir 目录。

Java 应用程序，尤其是 Java EE 企业级应用程序，通常不要求在文件系统中的已知位置存储一组文件。但是，如果包含了 Jython，还需要打包这些文件。

到现在为止，您应该可以看出来，Jython 实际上就是 Python，尽管是老一点版本的 Python。Jython 真正的优势在于能够集成 Java 和 Python，为您提供两全其美的方案。

## 21.7 集成 Java 和 Jython

将 Jython 解释器集成进 Java 应用程序的时候，Jython 的威力开始得到发挥。通过这样的组合，既可以利用脚本语言的优势，又可以使用丰富的 Java API。通过 Jython，可以实例化 Java 类的对象，并且可以像 Python 对象一样操作这些对象。甚至可以在 Jython 脚本内扩展 Java 类。

Jython 积极地将 Java 数据类型映射为 Python 数据类型，同时也将 Python 数据类型映射为 Java 数据类型。这种映射并不总是完整的，因为这些功能正在积极开发中。然而，在大多数情况下，在和 Python 类型进行相互转换的时候，Jython 都能胜任。

### 21.7.1 在 Jython 中使用 Java 类

一般情况下，在 Jython 脚本中把 Java 类当成 Python 类进行处理。Jython 使用 Python 的语法导入 Java 类。可以把 Java 包想象为 Python 模块和类的组合。例如，要把 java.util.Map 导入一个 Jython 脚本，可以使用下面的代码：

```
from java.util import Map
```

注意这看上去就好像 Python 的 `import` 一样。您可以在自己的脚本中进行尝试，如下例所示。

### 试一试

### 调用 Java 类

输入下面的脚本，并命名该文件为 `jystring.py`：

```
import sys
from java.lang import StringBuffer, System

sb = StringBuffer(100)    # Preallocate StringBuffer size for performance.

sb.append('The platform is: ')
sb.append(sys.platform)   # Python property
sb.append(' time for an omelette.')

sb.append('\n')          # Newline
sb.append('Home directory: ')
sb.append( System.getProperty('user.home') )

sb.append('\n')          # Newline
sb.append('Some numbers: ')
sb.append(44.1)
sb.append(', ')
sb.append(42)
sb.append(' ')

# Try appending a tuple.
tup=( 'Red', 'Green', 'Blue', 255, 204, 127 )
sb.append(tup)

print(sb.toString())

# Treat java.util.Properties as Python dictionary.
props = System.getProperties()

print('User home directory:', props['user.home'])
```

运行此脚本时，应该可以看到下面的输出：

```
$ jython jystring.py
The platform is: java1.6.0_03 time for an omelette.
Home directory: /Users/James
```

### 第III部分 开始使用 Python

```
Some numbers: 44.1, 42 ('Red', 'Green', 'Blue', 255, 204, 127)
User home directory: /Users/James
```

请注意，输出取决于 home 目录的位置以及安装的 Java 版本。

#### 示例说明

这段脚本导入了 Java 的 StringBuffer 类，然后调用这个类的某个构造函数：

```
from java.lang import StringBuffer
```

```
sb = StringBuffer(100)
```

Jython 解释器将值 100 从 Python 数值转换为 Java 数值。

#### 注意：

在 Java 程序中，不必导入 java.lang 包中的类。在 Jython 中，需要导入每一个用到的 Java 类。

可以将文本字符串和 Python 属性传递给 StringBuffer 的 append 方法：

```
sb.append('The platform is: ')
sb.append(sys.platform) # Python property
```

这个例子展示了 Jython 可以正确地将 Python 属性转换为 Java 字符串，以用在 Java 对象中。还可以传递 Java 方法返回的数据：

```
sb.append( System.getProperty('user.home') )
```

在这个例子中，System.getProperty 方法返回一个 Object Java 类型的对象。同样，Jython 的妥善地处理了这种情况，就好像 Jython 处理数值一样：

```
sb.append(44.1)
```

```
sb.append(42)
```

甚至还可以追加一个 Python 元组：

```
tup=( 'Red', 'Green', 'Blue', 255, 204, 127 )
sb.append(tup)
```

前面的例子表明，Jython 可以正确地将元组转换为 Java 文本字符串。

除了能将 Python 类型转换为 Java 类型以外，Jython 也可以将 Java 类型转换为 Python 类型。可以将 toString 方法返回的 Java String 对象传递给 Python 的 print 函数：

```
print sb.toString()
```

这表明可以将 Java 字符串视为 Python 字符串。还可以将 Java 的哈希映射和哈希表视为 Python 的字典，如下面的例子所示：

```
props = System.getProperties()

print('User home directory:', props['user.home'])
```

Java 的 `System.getProperties` 方法返回一个类型为 `java.util.Properties` 的对象，Jython 自动将其转换为 Python 的字典。

这个例子中展示的数据类型转换正是将 Java 和 Python 集成在一起的时候期望看到的结果。Jython 在幕后做了很多工作。Java 有一个类层次结构，Python 也有。Jython 很大一部分工作就是将两者巨大的层次结构集成在一起。最终，您能得到两种语言的好处。

例如，Python 可以向构造函数传递命名的属性。这种特性在使用类似于 Swing 这样的用户界面 API 的时候特别有用。Swing 的 API 有很多类。每一个类的对象都具有大量的属性。如果只使用 Java，只能调用已经定义好的构造函数，而且参数必须按照特定的顺序传递。而使用 Python 的时候，可以向对象的构造函数传递命名的属性，而且可以在一个调用内按需要设置任意数量的属性。

下面的例子说明了这种技术。

### 试一试

### 通过 Jython 创建一个用户界面

输入以下简单的脚本，并命名该文件为 `jyswing.py`：

```
from java.lang import System
from javax.swing import JFrame, JButton, JLabel
from java.awt import BorderLayout

# Exit application
def exitApp(event):
    System.exit(0)

# Use a tuple for size
frame = JFrame(size=(500,100))

# Use a tuple for RGB color values.
frame.background = 127,255,127

button = JButton(label='Push to Exit', actionPerformed=exitApp)
label = JLabel(text='A Pythonic Swing Application',
               horizontalAlignment=JLabel.CENTER)

frame.contentPane.add(label, BorderLayout.CENTER)
frame.contentPane.add(button, BorderLayout.WEST)
```

```
frame.setVisible(1)
```

运行此脚本时，应该可以看到如图 21-1 所示的窗口。

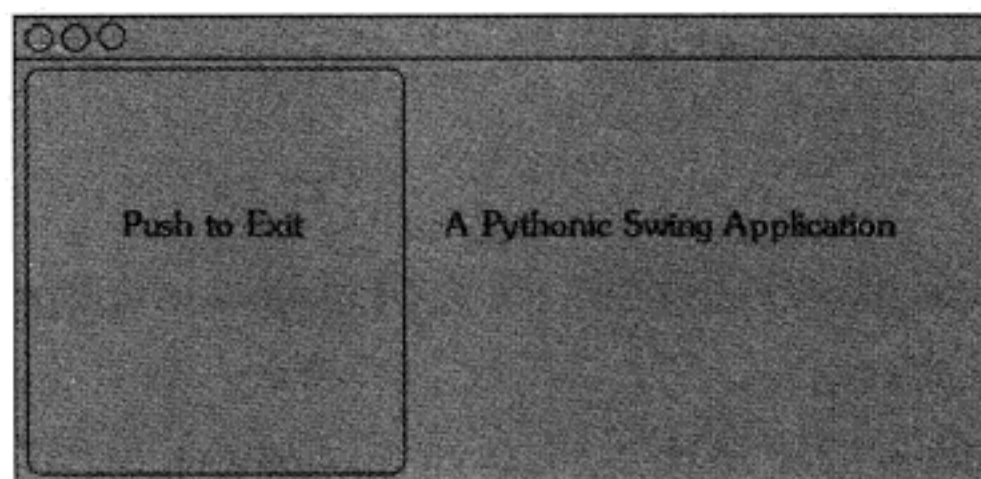


图 21-1

单击按钮退出应用程序。

### 示例说明

这个脚本展示了可以通过 Jython 使用复杂的 Swing API。尽管这个例子几乎完全是对 Java API 的调用，但是比相应的 Java 程序短得多。这是因为 Python 内置了很多方便的功能，例如元组和属性设置。

这个脚本首先导入了 AWT 和 Swing API 的几个类。JFrame 类作为应用程序的顶层窗口。可以通过下面的语句创建一个 JFrame 小组件：

```
frame = JFrame(size=(500,100))
```

JFrame 小组件的 size 属性是另一个 Java 类 java.awt.Dimension 的实例。在这个例子中，可以从元组构建一个 Dimension 对象，并传递这个对象以设置 JFrame 的 size 属性。

### 注意：

这里展示了 Jython 如何使得使用 Swing API 也变得轻松起来。使用 Swing 创建一个用户界面通常会涉及很多繁琐的编码。Jython 可以大大降低这些工作量。

可以通过元组以及 Jython 集成的 Java API 来设置颜色：

```
frame.background = 127,255,127
```

这段代码将背景颜色设置为浅绿色。

### 注意：

这个例子使用一个 8 位的颜色定义，分别用一个 0~255 的值来描述颜色的红、绿、蓝分量。因此，上面的代码将绿色设置为最高，将红色和蓝色的值设置为一半。

通过 Jython，可以很方便地在屏幕上创建交互式组件。例如，下面的代码创建了一个 JButton 小组件，并且将小组件设置为当用户单击按钮时调用函数 exitApp：

```
def exitApp(event):  
    System.exit(0)
```

```
button = JButton(label='Push to Exit', actionPerformed=exitApp)
```

在这里，`exitApp` 函数调用 Java 方法 `System.exit` 来退出 Java 引擎，因此退出了该应用程序。在 Jython 中，可以将 Java 属性设置为 Python 函数，例如本例中的 `exitApp`。在 Java 中，需要构建一个类，这个类要实现 `java.awt.event.ActionListener` 接口中的方法，然后传入这个类的一个实例来作为 `JButton` 动作监听器。使用 Jython 的方法大大减少了工作量。

该示例还创建了一个 `JLabel` 小组件，这个小组件可以显示文本消息、图片或两者兼而有之。`jyswing.py` 脚本设置了水平对齐方式，这样文本显示在小组件的中心：

```
label = JLabel(text='A Pythonic Swing Application',
               horizontalAlignment=JLabel.CENTER)
```

在这个例子中，值 `JLabel.CENTER` 是 `JLabel` 类的一个常量。

注意：

以 Java 的术语说，`JLabel.CENTER` 是类的一个公共静态 `final` 值。

创建小组件之后，需要将其放入一个容器。在这个示例脚本中，需要将 `JButton` 和 `JLabel` 小组件放入封闭的 `JFrame` 小组件中，如下面代码所示：

```
frame.contentPane.add(label, BorderLayout.CENTER)
frame.contentPane.add(button, BorderLayout.WEST)
```

注意：

在 Swing 应用程序中，将小组件添加到 `JFrame` 的内容窗格中，而不是直接添加到 `JFrame` 本身。

最后，该脚本设置 `JFrame` 小组件可见：

```
frame.setVisible(1)
```

请注意，Java 的 `setVisible` 方法接受一个 Java `boolean` 值，使用 Python 的 `True` 会产生一个语法错误，因为不同于 Python，Java 的 `boolean` 对象不是 0 和 1；Java 的 `boolean` 是一个类，在 Java 中，有时候使用 `boolean` 类，有时候又使用 0 和 1。这是 Python 还没有和 Java 进行映射的部分数据类型和常量。

### 21.7.2 从 Jython 访问数据库

Java 数据库连接(Java Database Connectivity, JDBC)提供了一套以一致的方式访问数据库的 API。使用 JDBC 的时候，数据库之间的大部分(但并非全部)差别都可以忽略。

Python 也有一套数据库 API，如第 14 章所述。Python API 和 Java API 的一大不同在于，Java 的 JDBC 驱动程序几乎是完全用 Java 编写的。此外，几乎所有 JDBC 驱动程序都由数据库供应商编写。大部分 Python 数据库驱动程序，例如 Oracle 的数据库驱动程序，都是用 C 语言编写的，并用 Python 做了包装。而且这些驱动程序大部分是由第三方编写

的，而不是由数据库供应商编写。Java 的 JDBC 驱动程序可以用于任何支持 Java 的平台。而 Python 的数据库驱动程序必须在每个平台上重新编译，而且可能无法在所有支持 Python 的平台上运行。

Jython 的 zxJDBC 包提供一个和 Python DB 兼容的驱动程序，这个驱动程序可以兼容任何 JDBC 驱动程序。也就是说，zxJDBC 连接了 Python 和 Java 的数据库 API，这样 Jython 脚本就可以利用很多 JDBC 驱动程序的优势，同时只要使用更简单的 Python DB API 就可以了。

使用 JDBC 驱动程序时，需要 4 个属性值来描述到数据库的连接，如表 21-2 所示。

表 21-2

属 性	意 义
JDBC URL	以驱动程序定义的格式描述到数据库的连接
User name	拥有数据库访问权限的用户的名称
Password	该用户的密码。这是数据库的密码，而不是操作系统的密码
Driver	提供 JDBC 驱动程序的 Java 类名

对于需要使用 JDBC 建立的任何数据库连接，都需要提供这 4 个值。使用 zxJDBC 模块时也需要提供这些值。通过 zxJDBC 驱动程序连接数据库时，可以使用下面的代码：

```
from com.ziclix.python.sql import zxJDBC

url='jdbc:hsqldb:hsqldb://localhost/xdm'
user='sa'
pw=''
driver='org.hsqldb.jdbcDriver'

db = zxJDBC.connect(url, user, pw, driver)
```

这里建立 JDBC 连接使用的属性值来自 HSqlDB 数据库的默认值，详见本章后面的“建立数据库”小节。

1. 使用 Python DB API

一旦建立了连接，就可以使用第 14 章描述的那些技术。zxJDBC 模块提供了一个和 DB 2.0 API 兼容的驱动程序(大部分兼容)。例如，可以使用下面的代码创建一个数据库表：

```
cursor = db.cursor()

cursor.execute("""
create table user
    (userid integer,
    username varchar,
    firstname varchar,
    lastname varchar,
```

```
        phone varchar)
    """)

cursor.execute("""create index userid on user (userid)""")
```

创建一个表之后，可以通过下面的代码插入行：

```
cursor.execute("""
insert into user (userid,username,firstname,lastname,phone)
values (4,'scientist','Hopeton','Brown','555-5552')
""")
```

一定要将任何修改提交至数据库：

```
db.commit()
```

可以通过下面的代码查询数据：

```
cursor.execute("select * from user")
for row in cursor.fetchall():
    print(row)

cursor.close()
```

**注意：**

有关 Python DB API 的更多信息，请参阅第 14 章。

## 2. 建立数据库

如果已经有一个包含 JDBC 驱动程序的数据库，就可以使用该数据库了。例如，Oracle、SQL Server、Informix 和 DB2 都为各自的数据库提供了 JDBC 驱动程序。

如果已经建立了一个数据库，尝试使用这个数据库。如果没有数据库，HSQLDB 是一个很方便的选择，它提供了小型、快速的数据库。HSQLDB 的一个主要优点是，由于它是用 Java 编写的，所以可以在任何运行 Java 的平台上运行。

**注意：**

有关 HSQLDB 的详细信息，请参阅 [https://sourceforge.net/projects/hsqldb/files/hsqldb/hsqldb\\_1\\_8\\_1/](https://sourceforge.net/projects/hsqldb/files/hsqldb/hsqldb_1_8_1/)，可以从此 Web 站点下载这个免费的开源包。

HSQLDB 的安装非常简单。只要解压缩下载的文件，并切换到新的 hsqldb 目录。要以默认参数在服务器模式下运行这个数据库，可以运行下面的命令：

```
$ java -cp ./lib/hsqldb.jar org.hsqldb.Server -database.0 mydb -dbname.0 xdb
[Server@922804]: [Thread[main,5,main]]: checkRunning(false) entered
[Server@922804]: [Thread[main,5,main]]: checkRunning(false) exited
[Server@922804]: Startup sequence initiated from main() method
[Server@922804]: Loaded properties from
```

```

[/Users/James/writing/python/chap22/server.properties]
[Server@922804]: Initiating startup sequence...
[Server@922804]: Server socket opened successfully in 160 ms.
[Server@922804]: Database [index=0, id=0, db=file:mydb, alias=xdb]
opened successfully in 1168 ms.
[Server@922804]: Startup sequence completed in 1444 ms.
[Server@922804]: 2009-08-22 20:09:33.417 HSQLDB server 1.8.1 is online
[Server@922804]: To close normally, connect and execute SHUTDOWN SQL
[Server@922804]: From command line, use [Ctrl]+[C] to abort abruptly

```

要停止这个数据库，只需要在启动 HSQLDB 数据库的 Shell 窗口中按 Ctrl+C 组合键即可。现在就有了一个可以通过表 21-3 中所示的默认属性连接的数据库。

表 21-3

属 性	值
JDBC URL	driver:jdbc:hsqldb:hsqldb://localhost/xdb
User name	sa
Password	''(两个单引号，即空字符串)
Driver	org.hsqldb.jdbcDriver

使用 JDBC 驱动程序的时候，需要将 JDBC 的 jar 文件加入到 Java classpath 中。jython 脚本没有处理这种情况，因此需要对脚本进行修改。例如，要使用 HSQLDB 数据库，修改脚本并添加 hsqldb.jar 文件：

```

#!/bin/sh
#####
# This file generated by Jython installer

java -Dpython.home=" C:\\jython2.5.0" \
    -classpath \
    "C:\\jython2.5.0jython.jar:$CLASSPATH:./hsqldb.jar" \
    "org.python.util.jython" "$@"

```

加粗的部分表示添加的 jar 文件。这个例子假定文件 hsqldb.jar 位于当前目录下。如果没有位于当前目录下，则需要输入这个 jar 文件的完整路径。

为把上面讲述的知识结合起来，尝试下面的例子，用一用 HSQLDB 数据库。

### 试一试

### 创建表

输入以下脚本，并命名该文件为 jyjdbc.py：

```
from com.ziclix.python.sql import zxJDBC
```

```
# Modify as needed for your database.
url='jdbc:hsqldb:hsqldb://localhost/xdb'
user='sa'
pw=''
driver='org.hsqldb.jdbcDriver'

db = zxJDBC.connect(url, user, pw, driver)

cursor = db.cursor()

cursor.execute("""
create table user
    (userid integer,
    username varchar,
    firstname varchar,
    lastname varchar,
    phone varchar)
""")

cursor.execute("""create index userid on user (userid)""")

cursor.execute("""
insert into user (userid,username,firstname,lastname,phone)
values (1,'ericfj','Eric','Foster-Johnson','555-5555')
""")

cursor.execute("""
insert into user (userid,username,firstname,lastname,phone)
values (2,'tosh','Peter','Tosh','555-5554')
""")

cursor.execute("""
insert into user (userid,username,firstname,lastname,phone)
values (3,'bob','Bob','Marley','555-5553')
""")

cursor.execute("""
insert into user (userid,username,firstname,lastname,phone)
values (4,'scientist','Hopeton','Brown','555-5552')
""")

db.commit()

cursor.execute("select * from user")
for row in cursor.fetchall():
    print(row)

cursor.close()
db.close()
```

运行此脚本时，应该可以看到类似下面的输出：

```
$ jython jyjdbc.py
(1, 'ericfj', 'Eric', 'Foster-Johnson', '555-5555')
(2, 'tosh', 'Peter', 'Tosh', '555-5554')
(3, 'bob', 'Bob', 'Marley', '555-5553')
(4, 'scientist', 'Hopeton', 'Brown', '555-5552')
```

### 示例说明

这个脚本几乎和第 14 章的 `createtable.py` 脚本完全一致。这充分说明了 Python DB API 给予您的自由，因为您没有被束缚在某一个数据库提供商上。除了建立数据库连接的代码之外，数据库代码可以使用多个数据库。

要建立到 **HSqlDB** 的连接，可以采用如下代码：

```
from com.ziclix.python.sql import zxJDBC

# Modify as needed for your database.
url='jdbc:hsqldb:hsqldb://localhost/xdb'
user='sa'
pw=''
driver='org.hsqldb.jdbcDriver'

db = zxJDBC.connect(url, user, pw, driver)
```

为了简单起见，这段代码使用了 **HSqlDB** 的默认连接属性。在现实世界情况下，绝对不能使用默认的用户名和密码。一定要更改数据库管理员用户名和密码。此外，**HSqlDB** 的管理员用户 **sa**(**system administrator** 的缩写)默认没有密码。这当然是一个很大的安全漏洞。

下面的代码取自第 14 章，创建了一个新的数据库表：

```
cursor = db.cursor()

cursor.execute("""
create table user
    (userid integer,
    username varchar,
    firstname varchar,
    lastname varchar,
    phone varchar)
""")

cursor.execute("""create index userid on user (userid)""")
```

尽管 SQL 并没有对创建数据库和数据库表的命令进行标准化，但是这个表本身的布局非常简单，因此这些命令应该适合大部分 SQL 数据库。

插入行的代码和查询代码也来自第 14 章。多亏了 Python 和 DB 2.0 API，使用数据库的代码才有了这些共性。**Jython** 的 **zxJDBC** 模块遵循此 API。例如，从用户表中查询所有行的代码如下：

```
cursor = db.cursor()

cursor.execute("select * from user")
for row in cursor.fetchall():
    print(row)

cursor.close()
```

不过, zxJDBC 模块扩展了 Python DB API, 提供了静态和动态游标的概念(对应 java.sql.ResultSet API 中的概念)。在 Python 的标准 API 中, 应该可以访问 Cursor 对象的 rowcount 属性。在 Java 中, ResultSet 不知道一个给定查询的完整行数, 而这个查询可能返回了数百万行。相反, JDBC 标准允许 ResultSet 按需获取数据, 根据数据库供应商或 JDBC 驱动程序供应商确定的方式进行缓冲。然后大多数读取数据库数据的 Java 代码都会遍历 ResultSet 提供的每一行。

为了支持 Python 的标准, zxJDBC 模块需要读取所有的行才能确定 rowcount 的值。这在查询大型表的时候可能会消耗大量内存。这就是 zxJDBC 文档所称的静态数据库游标。

为了避免占用太多内存的问题, 可以选择获得动态游标。动态游标不设置 rowcount 值。相反, 动态游标按需获得数据。如果请求一个动态游标, 那么将不能访问 rowcount 值, 但是可以遍历游标来处理查询返回的所有行。要请求一个动态游标, 将值 1 传递给 cursor 方法:

```
cursor = db.cursor(1)
```

动态游标不是 Python DB API 的一部分, 因此使用这种技术的代码只能使用 Jython 的 zxJDBC 驱动程序, 而不能使用其他数据库驱动程序。

数据库访问对企业级应用程序是必不可少的。您还需要数据库访问来创建健壮的 Web 应用程序。

### 21.7.3 通过 Jython 编写 Java EE servlet

大多数 Java 开发都围绕着企业级应用程序进行。为了提供帮助(或带来阻碍, 这取决于您的看法), Java 定义了一套称为 Java EE 的标准, 即 Java 平台企业版(Java Platform Enterprise Edition)。Java EE 标准定义了一个应用服务器, 以及一套应用服务器必须支持的 API。然后, 企业可以选择不同供应商提供的应用服务器, 例如 IBM 的 WebSphere、Bea 的 WebLogic、JBoss Group 的 JBoss 以及 Apache Jakarta 项目的 Tomcat。Java 开发人员编写的企业级应用程序都运行在这些应用服务器之上。

servlet 被定义为小型的、基于服务器的应用程序。servlet 这个术语受 applet 的启发, applet 描述的是小型应用程序。由于在 Java 的领域内, applet 总是运行在客户端, 因此服务端对应的小型应用程序应该取一个新名称, 这就是 servlet 的来历。每一个 servlet 都提供了整个应用程序的一小部分, 当然这里的“小”和您习惯的“小”不会一样, 因为大部分企业级应用程序都是巨大的。

在 Java EE 应用服务器中, servlet 是被动的请求-响应应用程序。客户端通常是 Web

浏览器，例如 Internet Explorer 或 Firefox。浏览器发送请求到应用服务器。应用服务器将请求传递给 servlet。然后 servlet 生成响应，通常都是服务器发送回客户端的 HTTP 文档。几乎在所有情况下，发送回客户端的 HTML 文档都是动态创建的。例如，在一个网络订购系统中，发送回客户端的 HTML 文档可能是搜索的结果或者一组产品的当前价格。

编写 servlet 的好处在于，Java EE 为编写 servlet 提供了一套定义良好的 API，而且多个供应商都支持这套 API。而在 Python 的情形中，有很多种 Python Web API 可供选择，但是却不像在 Java EE 中那样有众多供应商的支持。

在 Jython 中，可以用 Python 编写 Java servlet，这极大地简化了工作。当然，需要一个支持 servlet 的应用服务器才能完成这种工作。

### 1. 建立应用服务器

如果已经有一个 Java EE 应用服务器，就使用该服务器。如果没有，尝试 Tomcat。Tomcat 来自 Apache Jakarta 项目，提供了一个免费的开源 servlet 引擎(按照 Java EE 的术语，就是 servlet 容器)。

从 <http://jakarta.apache.org/tomcat/> 下载 Tomcat。安装时，将下载的文件解压到一个目录。应该可以看到一个反映下载的版本的目录，例如 jakarta-tomcat-6.0.20。进入这个目录。在这个目录中，您将看到一些文件和子目录。最重要的两个子目录分别是 bin 目录，其中包含启动和停止 Tomcat 的脚本；以及 webapps 目录，在这里放置自己创建的 Jython 脚本(放置在一个特殊的子目录中，下一节将详细介绍相关内容)。

要运行 Tomcat，切换到 bin 子目录，运行 startup.sh 脚本(或者 Windows 上的 startup.bat)。例如：

```
$ ./startup.sh
Using CATALINA_BASE:  /Users/jamesp/servers/jakarta-tomcat-5.0.28
Using CATALINA_HOME:  /Users/jamesp/servers/jakarta-tomcat-5.0.28
Using CATALINA_TMPDIR: /Users/jamesp/servers/jakarta-tomcat-5.0.28/temp
Using JAVA_HOME:      /Library/Java/Home
```

必须确保已经设置了 JAVA\_HOME 环境变量，否则 Tomcat 将无法启动。要验证 Tomcat 正在运行，在 Web 浏览器中输入下面的 URL: <http://localhost:8080/>。应该可以看到如图 21-2 所示的文档。

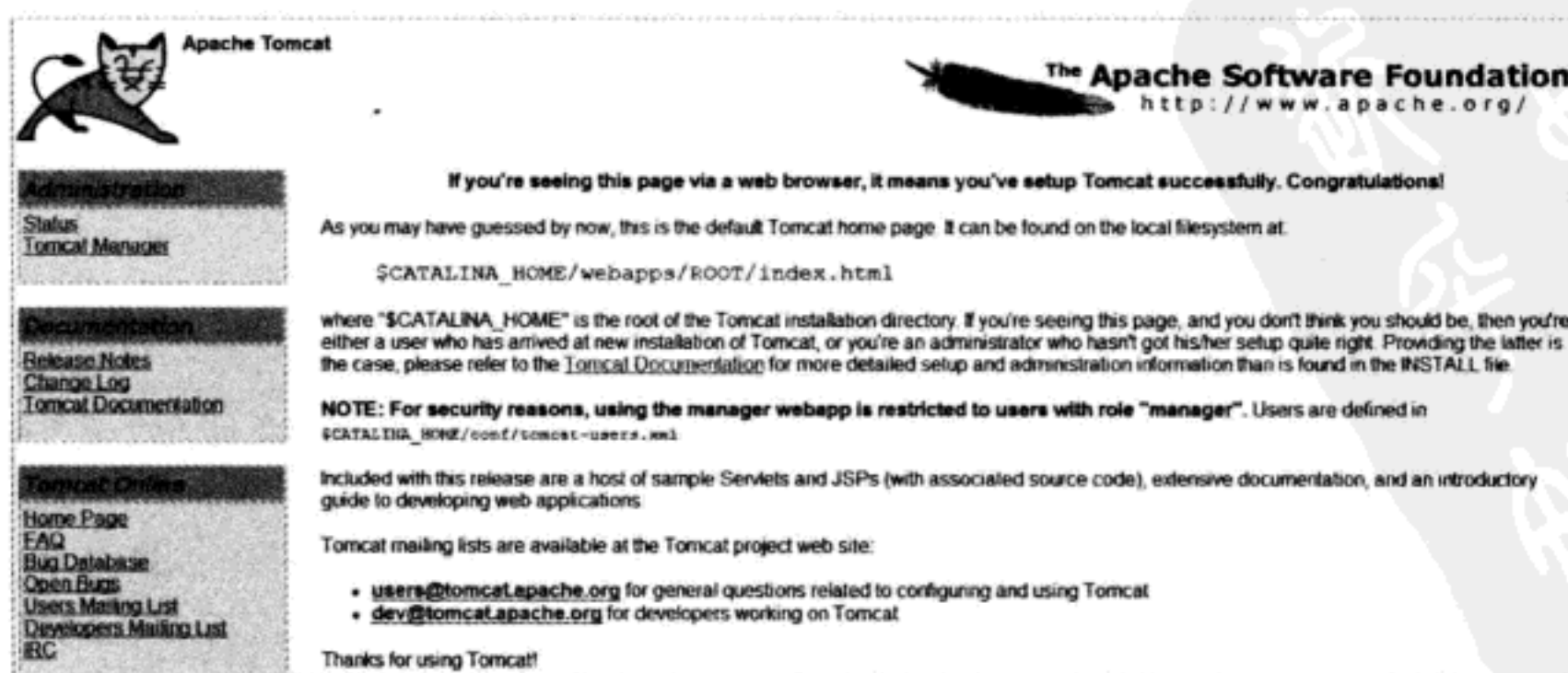


图 21-2

一旦运行了一个应用服务器，例如 Tomcat，下一步就是部署应用程序，在这里将部署一个称为 PyServlet 的特殊的 Python servlet。

## 2. 将 PyServlet 添加到应用服务器

Jython 包含一个称为 `org.python.util.PyServlet` 的类，这个类是 Python 脚本的前端。PyServlet 类负责加载 Python 脚本，编译脚本，并且就像运行 Java servlet 一样运行这些脚本(实际上它们也就是 Java servlet，参见下面的小节“扩展 HttpServlet”)。

必须创建一个真正的 Java EE Web 应用程序，上面介绍的行为才会发生。幸好，这项工作并不难。切换到安装 Tomcat 的目录，并运行下面的命令，这条命令将创建一个目录：

```
$ cd webapps
$ mkdir jython
```

这条命令在 `webapps` 目录下创建了一个名为 `jython` 的目录，表明将要创建一个名为 `jython` 的 Web 应用程序。

```
$ mkdir webapps/jython/WEB-INF
```

这条命令创建了一个名为 `WEB-INF` 的目录。这个目录的名称和大小写非常重要。在 Java EE 中，`WEB-INF` 目录包含 Web 应用程序的库和部署信息。

```
$ mkdir webapps/jython/WEB-INF/lib
```

`lib` 子目录包含 Web 应用程序需要的所有 jar 文件。您需要一个 Jython 安装中的 jar 文件：`jython.jar`。将这个文件复制到刚才创建的 `webapps/jython/WEB-INF/lib` 目录下。

下一步，需要修改 `tomcat 6.0/conf` 目录下的 `web.xml` 文件。将下列文本输入 `web.xml` 文件。

```
<web-app>
  <servlet>
    <servlet-name>PyServlet</servlet-name>
    <servlet-class>
      org.python.util.PyServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
    <init-param>
      <param-name>python.home</param-name>
      <param-value>c:\jython-2.5</param-value>
    </init-param>
    <init-param>
      <param-name>python.path</param-name>
      <param-value>
        c:\jython-2.5\lib\site-packages
      </param-value>
    </init-param>
  </servlet>
</web-app>
```

```
</servlet>
<servlet-mapping>
  <servlet-name>PyServlet</servlet-name>
  <url-pattern>*.py</url-pattern>
</servlet-mapping>
</web-app>
```

将加粗显示的路径改为 Jython 安装目录的完整路径。  
下一步，需要在新的 Web 应用程序中创建一些 Python 脚本。

**注意：**  
本章简要介绍了 Java EE，Java EE 本身是一个相当复杂的主题。如果您对 Java EE 不熟悉，请查阅 Java EE 教程，或访问 <http://java.sun.com/javaee/>。

3. 扩展 HttpServlet

javax.servlet.http.HttpServlet 类是 Java EE 开发人员创建 servlet 的主要动因。Java EE 开发人员用自己的类扩展 HttpServlet 类以创建 servlet。而通过 PyServlet 类，就可以在 Jython 中实现同样的操作。而且在 Jython 中，完成这项任务比在 Java 中简单得多。

使用下面的代码作为在 Jython 中创建 servlet 类的模板：

```
from javax.servlet.http import HttpServlet

class verify(HttpServlet):
    def doGet(self, request, response):
        self.handleRequest(request, response)

    def doPost(self, request, response):
        self.handleRequest(request, response)

    def handleRequest(self, request, response):
        response.setContentType('text/html');
        out = response.getOutputStream()
        print >>out, "YOUR OUTPUT HERE"
        out.close()
    return
```

自定义的类必须继承 HttpServlet 类。另外，还要定义两个方法：doGet 和 doPost，如表 21-4 所示。

表 21-4

方 法	作 用
DoGet	处理 GET 请求。GET 请求将所有的参数放在 URL 中
DoPost	处理 POST 请求。POST 请求通常带有表单数据

几乎所有情况下，都需要实现两种方法来完成相同的任务。这两种方法之间的任何区

别都只会导致 Web 应用程序的调试更加困难。因此,再写一个让这两个方法都调用的方法,例如上面模板中的 `handleRequest` 方法。

`handleRequest` 方法必须完成几项任务,每一项任务都必须正确完成,否则就会出现错误,或得不到输出。这几项任务包括:

- 设置 `response` 对象的正确内容类型。大部分情况下都是 `text/html`。
- 从 `response` 对象获得一个输出流。
- 将所有的输出写入这个流。
- 关闭流。

下面的例子展示了如何从这个代码模板创建真正的 `servlet`。

### 试一试

### 编写一个 Python `servlet`

输入以下代码,并将文件保存为 `webapps/jython/verify.py`:

```
import sys
from javax.servlet.http import HttpServlet

class verify(HttpServlet):
    def doGet(self, request, response):
        self.handleRequest(request, response)

    def doPost(self, request, response):
        self.handleRequest(request, response)

    def handleRequest(self, request, response):
        response.setContentType("text/html");

        out = response.getOutputStream()
        print >>out, "<html><head><title>"
        print >>out, "Jython Is Running</title></head>"
        print >>out, "<body>"
        print >>out, "<h2>Jython is running</h2>"
        print >>out, "<p>"
        print >>out, "Version:", sys.version, " verified."
        print >>out, "</p>"
        print >>out, "</body></html>"
        out.close()
        return
```

必须将 Web 应用程序中的这个文件保存在 `webapps/jython` 目录下。保存文件之后,停止并重新启动 Tomcat,以保证 Tomcat 识别了所做的修改。

在 Web 浏览器中输入下面的 URL 以测试这个新创建的 `servlet`: `http://localhost:8080/jython/verify.py`。图 21-3 展示了得到的结果。



图 21-3

示例说明

这个servlet的正确工作依赖于3个关键部分：

- Tomcat 必须运行。
- Web 应用程序必须拥有正确的目录结构和内容。
- URL 必须指向 Web 应用程序中的 Python 脚本。脚本必须以.py 作为文件扩展名。

在前面修改的 web.xml 文件中，已经为所有以.py 作为扩展名的文件注册了 PyServlet。因此，Tomcat 看到 `http://localhost:8080/jython/verify.py` 这样的 URL 之后，就会定向到 PyServlet 来处理请求。表 21-5 将这个 URL 分解为几个重要的组成部分。

表 21-5

组 成 部 分	作 用
http: //	定义使用的协议，在这个例子中为 HTTP
jython	这是 Web 应用程序的名称(可以根据需要任意命名)。使用 Tomcat 时，必须有一个 webapps/jython 目录
verify.py	Web 应用程序中的文件名。.py 后缀表示请求应该由 PyServlet 来处理

实际的 servlet 类本身非常小，完全遵循前面展示的代码模板。这个 servlet 的主要动作定义在 `handleRequest` 方法中：

```
def handleRequest(self, request, response):
    response.setContentType("text/html");

    out = response.getOutputStream()
    print >>out, "<html><head><title>"
    print >>out, "Jython Is Running</title></head>"
    print >>out, "<body>"
    print >>out, "<h2>Jython is running</h2>"
    print >>out, "<p>"
    print >>out, "Version:", sys.version, " verified."
    print >>out, "</p>"
    print >>out, "</body></html>"
    out.close()
    return
```

这个方法的大部分内容都是 `print` 语句，将 HTML 格式的文本发送给输出流。将这种创建 Web 应用程序的方法和第 20 章介绍的技术进行对比。

可以看到，确实至少需要了解一点 Python 和 Java 的知识才能够使用 Jython。因此，选择正确的工具很重要。

### 21.7.4 选择 Jython 开发工具

由于 Jython 主要关注于使用 Java 和 Python，因此最好的 Jython 工具来自 Java 领域。下面的工具可以帮助使用 Jython：

- JEdit 文本编辑器([www.jedit.org](http://www.jedit.org))包含很多和 Python 相关的插件。不管是使用 Python 还是 Jython，这个编辑器可以突出显示 Python 的语法。另外，JythonInterpreter 插件嵌入了一个 Jython 解释器。有关 JEdit 插件的更多信息，请访问 <http://plugins.jedit.org/>。
- Eclipse 集成开发环境(Integrated Development Environment, IDE)为 Java 开发提供了非常好的支持。另外，有一个 Eclipse 插件特别适合在 Jython 中使用 Python，这个插件是 PyDev，网址为 <http://sourceforge.net/projects/pydev/>。Eclipse 的下载地址为 [www.eclipse.org](http://www.eclipse.org)。

不管选择什么工具，真正需要的就是一个文本编辑器和一个命令行 Shell。另外，选择的工具还有助于测试，特别是 Java 应用程序的测试。

## 21.8 使用 Jython 进行测试

由于 Jython 在 Java 平台之上提供了一个交互式环境，因此 Jython 是一个非常好的交互式测试工具。下面的例子展示了如何通过 Jython 的交互式模式来探索 Java 环境。

### 试一试

### 通过 Jython 探索 Java 环境

输入下面的命令，查看 Java 的 Map 接口：

```
$ jython
Jython 2.1 on java1.4.2_05 (JIT: null)
Type "copyright", "credits" or "license" for more information.
>>> from java.util import Map
>>> print(dir(Map))
['__Entry__', '__class__', '__contains__', '__delattr__', '__delitem__', '__doc__',
 '__eq__', '__getattr__', '__getitem__', '__hash__', '__init__', '__iter__',
 '__len__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__setitem__', '__str__', 'class', 'clear', 'containsKey',
 'containsValue', 'empty', 'entrySet', 'equals', 'get', 'getClass', 'hashCode',
 'isEmpty', 'keySet', 'notify', 'notifyAll', 'put', 'putAll', 'remove', 'size',
 'toString', 'values', 'wait']
>>>
```

### 示例说明

这个例子通过 Python 的 `dir` 函数来显示 Java 中的 `java.util.Map` 接口的信息。可以列出任何 Java 类或接口的信息。

下面是另一个例子，查看 JNDI(Java Naming and Directory Interface, Java 命名和目录接口)的类，例如 InitialContext 类，如下所示：

```
$ jython
Jython 2.1 on java1.4.2_05 (JIT: null)
Type "copyright", "credits" or "license" for more information.
>>> from javax.naming import InitialContext
>>> print(dir(InitialContext))
['APPLET', 'AUTHORITATIVE', 'BATCHSIZE', 'DNS_URL', 'INITIAL_CONTEXT_
FACTORY', '
LANGUAGE', 'OBJECT_FACTORIES', 'PROVIDER_URL', 'REFERRAL', 'SECURITY_
AUTHENTICAT
ION', 'SECURITY_CREDENTIALS', 'SECURITY_PRINCIPAL', 'SECURITY_PROTOCOL',
'STATE_
FACTORIES', 'URL_PKG_PREFIXES', '__class__', '__delattr__', '__doc__', '__eq__',
'__getattr__', '__hash__', '__init__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__str__', 'addToEnvironment', 'bind',
'class', 'close', 'composeName', 'createSubcontext', 'destroySubcontext', 'do
Lookup', 'environment', 'equals', 'getClass', 'getEnvironment', 'getNameInNamesp
ace', 'getNameParser', 'hashCode', 'list', 'listBindings', 'lookup', 'lookupLink',
'nameInNamespace', 'notify', 'notifyAll', 'rebind', 'removeFromEnvironment',
'rename', 'toString', 'unbind', 'wait']
>>>
```

**注意：**

将这种技术和嵌入的 Jython 解释器结合起来可以查看正在运行的应用程序。有关嵌入 Jython 解释器的详细信息，请参阅下一节“嵌入 Jython 解释器”。

除了可以使用 Jython 的交互式模式之外，还可以用 Jython 编写测试。

**注意：**

很多公司都回避使用开源软件，例如 Jython。可能在编写测试的时候更容易引入 Jython，因为测试不会进入最终的产品。一旦公司有了一些使用 Jython 的经验，人们可能更容易接受在更多领域使用 Jython。

到目前为止，所有的例子都通过 jython 脚本来运行 Jython 脚本，但是 PyServlet servlet 的例子是一个例外。使用 PyServlet 类的时候，实际上用的是带有 Jython 解释器的 Java 类。也可以将 Jython 解释器嵌入到自己的类中。

## 21.9 嵌入 Jython 解释器

将 Jython 解释器嵌入自己编写的 Java 类之后，就可以在应用程序内部运行脚本，获得对整个环境的控制。这一点非常重要，因为很少有 Java 应用程序是从命令行执行的。

Jython 解释器在 org.python.util.PythonInterpreter 类中可以找到。

通过下列代码初始化 Jython 解释器：

```
Properties props = new Properties();

props.put("python.home", pythonHome);

PythonInterpreter.initialize(
    System.getProperties(),
    props,
    new String[0]);

interp = new PythonInterpreter(null, new PySystemState());
```

注意，这是 Java 代码而不是 Python 代码。  
必须设置 `python.home` 系统属性。

### 通过 Java 调用 Jython 脚本

初始化解释器之后，就可以通过调用 `execfile` 方法来执行 Jython 脚本了。例如：

```
interp.execfile(fileName);
```

需要传入要执行的文件的完整名称。下面的示例演示了具体操作。

#### 试一试

#### 嵌入 Jython 解释器

输入下面的 Java 程序，并将文件命名为 `JyScriptRunner.java`：

```
package jython;

import java.util.Properties;

import org.python.util.PythonInterpreter;
import org.python.core.PySystemState;

/**
 * Runs Jython scripts.
 */
public class JyScriptRunner {

    private PythonInterpreter interp;

    /**
     * Initializes the Jython interpreter.
```

```

    */
    public void initialize(String pythonHome) {
        Properties props = new Properties();

        props.put("python.home", pythonHome);

        PythonInterpreter.initialize(
            System.getProperties(),
            props,
            new String[0]);

        interp = new PythonInterpreter(null, new PySystemState());
    }

    /**
     * Runs the given script.
     */
    public void run(String fileName) {
        interp.execfile(fileName);
    }

    public static void main(String[] args) {
        String fileName = args[0];

        JyScriptRunner runner = new JyScriptRunner();

        String pythonHome = System.getProperty("python.home");

        runner.initialize(pythonHome);

        runner.run(fileName);
    }
}

```

由于这是一个 **Java** 程序，因此需要通过下面的命令来编译这个程序：

```
$ javac -classpath ./jython.jar JyScriptRunner.java
```

运行这个 **Java** 程序将得到如下的输出：

```

$ java -cp ./jython.jar:. \
    -Dpython.home="c:/jython2.5" \
    jython.JyScriptRunner jystring.py
The platform is: java1.6.0_03 time for an omelette.
Home directory: c:/jython2.5
Some numbers: 44.1, 42 ('Red', 'Green', 'Blue', 255, 204, 127)
User home directory: /Users/jamesp

```

这个例子运行了 `jystring.py` 示例脚本。注意，将需要 `-Dpython.home` 设置为安装 Jython 的位置。还要将 `.jython.jar` 改为文件 `jython.jar` 的位置。

示例说明

这个程序需要调用者传入两个值：`python.home` 系统属性的设置和需要执行的脚本的名称。必须将 `jython.jar` 放置在当前目录下(或者修改命令行，指向 `jython.jar` 文件的位置)。

`JyScriptRunner` 类包含一个 `main` 方法，当运行程序的时候被调用。`main` 方法从命令行(保存在 `args` 数组中)读取系统属性 `python.home` 和文件名。`main` 方法然后实例化 `JyScriptRunner` 对象。

`main` 方法初始化 `JyScriptRunner` 对象，然后调用 `run` 方法来执行脚本。Jython 脚本中的任何错误都会导致异常并且停止程序的执行。

这可能是您可以创建出来的最简单的 Jython 解释器了。在应用程序中，可能还需要控制 Python home 目录的位置，例如把这个目录放在应用程序目录下。

21.10 处理 C-Python 和 Jython 之间的差异

C-Python 平台根据 Python 的标准和约定创建了一个完整的 Python 环境。而 Jython 试图基于 Java 平台创建完整的 Python 环境。由于这个原因，这两种实现之间一定会有一些差异。在 Jython 脚本中混入 Java 代码的时候，这些差异就会加剧。

Jython 解释器试图将 Python 类型转换为适当的 Java 类型，这样就可以调用 Java 类的方法。只要有可能，Jython 解释器就试图做正确的转换，因此在大部分情形下都不用太关注这些类型转换。如果不确定调用某个 Java 方法需要什么 Python 类型，请查看表 21-6 中列出的类型。

表 21-6

Python 类 型	Java 类 型
None	Null
Integer(任何非零值都为 true)	Boolean
Integer	short, int, long, byte
String	byte[], char[], java.lang.string
长度为 1 的 String	Char
Float	float, double
String	java.lang.Object, 转换为 java.lang.String
任何类型	java.lang.Object
Class 或 JavaClass	java.lang.Class
Array(包含的对象必须为指定类型或指定类型的子类)	特定类型的 Array

例如，如果一个 Java 方法期待传入 `java.lang.Object` 类型的对象，而实际传入的是 Python

的 String, 那么 Jython 会将 Python 的 String 转换为 java.lang.String 对象。Jython 在传递其他任何 Python 对象类型时都不会进行修改。

除了本章介绍的内容, Jython 还可以实现其他很多功能。例如可以在 Jython 中创建类, 然后从 Java 中调用这些类(相关示例可以查看 PyServlet 类的源代码)。

## 21.11 本章小结

Jython 提供了集成 Python 的脚本功能和 Java 的企业基础结构的能力。使用 Jython 可以让您成为更高效的 Java 开发人员, 当您工作在没有接受 Python、但是接受了 Java 的企业中时更是如此。

通过 Jython, 可以完成下列任务:

- 在 Java 平台上运行 Python 脚本。由于这些脚本和 Python 脚本不同, 因此通常被称为 Jython 脚本。
- 在脚本中调用 Java 代码和类, 从而可以在 Jython 脚本中使用丰富的 Java 库。
- 通过 Java Swing API 创建用户界面。通过利用 Python 对元组和属性的支持, Jython 脚本可以显著地减少创建基于 Swing 的用户界面所需的代码量。
- 访问任何提供了 JDBC 驱动程序的数据库。zxJDBC 驱动程序连接了 Python DB API 和 Java JDBC API。
- 通过使用 Java EE Web 应用程序中方便的 PyServlet 类, 将 Jython 脚本作为 Java servlet 运行。
- 以交互式方式收集 Java 类的信息, 并执行这些类的方法。这种功能对于测试非常有用。
- 在自己编写的 Java 类中嵌入 Jython 解释器, 从而可以在 Java 代码中执行 Jython 脚本和表达式。

对 Python 的介绍至此结束。接下来的附录提供了各章习题的答案以及 Python 资源的链接。

## 21.12 习题

1. 既然 Python 这么强大, 为什么还有人使用其他程序设计语言, 如 Java、C++、C#、Basic 和 Perl?
2. Jython 解释器是使用什么语言编写的? python 命令又是使用什么语言编写的?
3. 如果要打包 Jython 应用程序以便在其他系统上运行, 应该打包哪些内容?
4. 能否在 Jython 脚本中使用 Python DB 驱动程序模块, 例如第 14 章描述的那些?
5. 编写一个 Jython 脚本, 通过 Swing API 创建一个背景为红色的窗口。

## 第Ⅳ部分

# 附 录

附录 A: 习题答案

附录 B: 在线资源

附录 C: Python 3.1 的新特性

附录 D: 术语表

数字资源  
PDG



# A

## A.1 第1章

## A.2 第2章

```
>>> print("%± % 6)
6
>>> print("%± % 7)
7
>>> print("%± % 8)
```

```
10
>>> print("%"± % 9)
11
>>> print("%"± % 10)
12
>>> print("%"± % 11)
13
>>> print("%"± % 12)
14
>>> print("%"± % 13)
15
>>> print("%"± % 14)
16
```

### 习题 3 解答

```
>>> print("%x" % 9)
9
>>> print("%x" % 10)
a
>>> print("%x" % 11)
b
>>> print("%x" % 12)
c
>>> print("%x" % 13)
d
>>> print("%x" % 14)
e
>>> print("%x" % 15)
f
>>> print("%x" % 16)
10
>>> print("%x" % 17)
11
>>> print("%x" % 18)
12
>>> print("%x" % 19)
13
```

### 习题 4 解答

当调用一个未知函数时，Python 不知道刚才输入的名称是一个函数名，所以它仅标记一个通用的语法错误：

```
>>> pintr("%x" & x)
File "<input>", line 1
  Pintr("%x" & x)

SyntaxError: invalid syntax
```

然而请注意，当输入 `print` 时，Python Shell 将以粗体显示它。这是因为 `print` 是 Python

中的一个关键字，Python Shell 也知道这一点。关注编辑器对您的输入所做出的反应有助于捕获错误。

## A.3 第 3 章

### 习题 1 解答

```
>>> dairy_section = ["milk", "cottage cheese", "butter", "yogurt"]
```

### 习题 2 解答

```
>>> print("First: %s and Last %s" % (dairy_section[0], dairy_section[1]))
First: milk and Last cottage cheese
```

### 习题 3 解答

```
>>> milk_expiration = (10, 10, 2009)
```

### 习题 4 解答

```
>>> print("This milk will expire on %d/%d/%d" % (milk_expiration[0],
milk_expiration[1], milk_expiration[2]))
This milk will expire in 10/10/2009
```

### 习题 5 解答

```
>>> milk_carton = {}
>>> milk_carton["expiration_date"] = milk_expiration
>>> milk_carton["fl_oz"] = 32
>>> milk_carton["cost"] = 1.50
>>> milk_carton["brand_name"] = "Milk"
```

### 习题 6 解答

```
>>> print("The expiration date is %d/%d/%d" %
(milk_carton["expiration_date"][0], milk_carton["expiration_date"][1],
milk_carton["expiration_date"][2]))
The expiration date is 10/10/2009
```

### 习题 7 解答

```
>>> print("The cost for 6 cartons of milk is %.02f" % (6*
milk_carton["cost"]))
The cost for 6 cartons of milk is 9.00
```

## 习题 8 解答

```
>>> cheeses = ["cheddar", "american", "mozzarella"]
>>> dairy_section.append(cheeses)
>>> dairy_section
['milk', 'cottage cheese', 'butter', 'yogurt', ['cheddar', 'american',
'mozzarella']]
>>> dairy_section.pop()
['cheddar', 'american', 'mozzarella']
```

## 习题 9 解答

```
>>> len(dairy_section)
4
```

## 习题 10 解答

```
>>> print("Part of some cheese is %s" % cheeses[0][0:5])
Part of some cheese is chedd
```

# A.4 第 4 章

## 习题 1 解答

这里要展示的是 0 是 False，而其他值被认为不是 False，也就是说它们等同于 True：

```
>>> if 0:
...     print("0 is True")
...
>>> if 1:
...     print("1 is True")
...
1 is True
>>> if 2:
...     print("2 is True")
...
2 is True
>>> if 3:
...     print("3 is True")
...
3 is True
>>> if 4:
...     print("4 is True")
...
4 is True
>>> if 5:
...     print("5 is True")
...
5 is True
```



## 习题 2 解答

```
>>> number = 3
>>> if number >= 0 and number <= 9:
...     print("The number is between 0 and 9: %d" % number)
...
The number is between 0 and 9: 3
```

## 习题 3 解答

```
>>> test_tuple = ("this", "little", "piggie", "went", "to", "market")
>>> search_string = "toes"
>>> if test_tuple[0] == search_string:
...     print("The first element matches")
... elif test_tuple[1] == search_string:
...     print("the second element matches")
... else:
...     print("%s wasn't found in the first two elements" % search_string)
...
toes wasn't found in the first two elements
```

## 习题 4 解答

```
>>> fridge = {"butter": "Dairy spread", "peanut butter": "non-dairy spread",
"cola": "fizzy water"}
>>> food_sought = "chicken"
>>> for food_key in fridge.keys():
...     if food_key == food_sought:
...         print("Found what I was looking for: %s is %s" % (food_sought,
fridge[food_key]))
...         break
...     else:
...         print("%s wasn't found in the fridge" % food_sought)
...
chicken wasn't found in the fridge
```

## 习题 5 解答

```
>>> fridge = {"butter": "Dairy spread", "peanut butter": "non-dairy spread",
"cola": "fizzy water"}
>>> fridge_list = fridge.keys()
>>> current_key = fridge_list.pop()
>>> food_sought = "cola"
>>> while len(fridge_list) > 0:
...     if current_key == food_sought:
...         print("Found what I was looking for: %s is %s" % (food_sought,
fridge[current_key]))
...         break
...     current_key = fridge_list.pop()
```

```
... else:
...     print("%s wasn't found in the fridge" % food_sought)
...
Found what I was looking for: cola is fizzy water
```

## 习题 6 解答

```
>>> fridge = {"butter":"Dairy spread", "peanut butter":"non-dairy spread",
"cola":"fizzy water"}
>>> food_sought = "chocolate milk"
>>> try:
...     fridge[food_sought]
... except KeyError:
...     print("%s wasn't found in the fridge" % food_sought)
... else:
...     print("Found what I was looking for: %s is %s" % (food_sought,
fridge[food_key]))
...
chocolate milk wasn't found in the fridge
```

## A.5 第 5 章

### 习题 1 解答

```
def do_plus(first, second):
    return first + second
```

### 习题 2 解答

```
def do_plus(first, second):
    for param in (first, second):
        if (type(param) != type("")) and (type(param) != type(1)):
            raise TypeError("This function needs a string or an integer")
    return first + second
```

### 习题 3 解答

```
# Part 1 - fridge has to go before the omelet_type. omelet_type is an
# optional parameter with a default parameter, so it has to go at the end.
# This can be used with a fridge such as:
# f = {'eggs':12, 'mozzarella cheese':6,
#      'milk':20, 'roast red pepper':4, 'mushrooms':3}
# or other ingredients, as you like.
def make_omelet_q3(fridge, omelet_type = "mozzarella"):
    """This will make an omelet. You can either pass in a dictionary
    that contains all of the ingredients for your omelet, or provide
```

a string to select a type of omelet this function already knows about

The default omelet is a mozzarella omelet"""

```
def get_omelet_ingredients(omelet_name):
    """This contains a dictionary of omelet names that can be produced,
    and their ingredients"""
    # All of our omelets need eggs and milk
    ingredients = {"eggs":2, "milk":1}
    if omelet_name == "cheese":
        ingredients["cheddar"] = 2
    elif omelet_name == "western":
        ingredients["jack_cheese"] = 2
        ingredients["ham"] = 1
        ingredients["pepper"] = 1
        ingredients["onion"] = 1
    elif omelet_name == "greek":
        ingredients["feta_cheese"] = 2
        ingredients["spinach"] = 2
    # Part 5
    elif omelet_name == "mozzarella":
        ingredients["mozzarella cheese"] = 2
        ingredients["roast red pepper"] = 2
        ingredients["mushrooms"] = 1
    else:
        print("That's not on the menu, sorry!")
        return None
    return ingredients
# part 2 - this version will use the fridge that is available
# to the make_omelet function.
def remove_from_fridge(needed):
    recipe_ingredients = {}
    # First check to ensure we have enough
    for ingredient in needed.keys():
        if needed[ingredient] > fridge[ingredient]:
            raise LookupError("not enough %s to continue" % ingredient)
    # Then transfer the ingredients.
    for ingredient in needed.keys():
        # Remove it from the fridge
        fridge[ingredient] = fridge[ingredient] - needed[ingredient]
        # and add it to the dictionary that will be returned
        recipe_ingredients[ingredient] = needed[ingredient]
    # Part 3 - recipe_ingredients now has all the needed ingredients
    return recipe_ingredients

# Part 1, continued - check the type of the fridge
if type(fridge) != type({}):
    raise TypeError("The fridge isn't a dictionary!")

if type(omelet_type) == type({}):
```

```

    print("omelet_type is a dictionary with ingredients")
    return make_food(omelet_type, "omelet")
elif type(omelet_type) == type(""):
    needed_ingredients = get_omelet_ingredients(omelet_type)
    omelet_ingredients = remove_from_fridge(needed_ingredients)
    return make_food(omelet_ingredients, omelet_type)
else:
    print("I don't think I can make this kind of omelet: %s" %
          omelet_type)

```

## 习题 4 解答

可以按照如下方式修改 `make_omelet_q3` 的 `get_omelet_ingredient`:

```

def get_omelet_ingredients(omelet_name):
    """This contains a dictionary of omelet names that can be produced,
    and their ingredients"""
    # All of our omelets need eggs and milk
    ingredients = {"eggs":2, "milk":1}
    if omelet_name == "cheese":
        ingredients["cheddar"] = 2
    elif omelet_name == "western":
        ingredients["jack_cheese"] = 2
        ingredients["ham"] = 1
        ingredients["pepper"] = 1
        ingredients["onion"] = 1
    elif omelet_name == "greek":
        ingredients["feta_cheese"] = 2
        ingredients["spinach"] = 2
    # Part 5
    elif omelet_name == "mozzarella ":
        ingredients["mozzarella cheese"] = 2
        ingredients["roast red pepper"] = 2
        ingredients["mushrooms"] = 1
    # Question 4 - we don't want anyone hurt in our kitchen!
    elif omelet_name == "salmonella":
        raise TypeError("We run a clean kitchen, you won't get this here")
    else:
        print("That's not on the menu, sorry!")
        return None
    return ingredients

```

运行它时，试图得到沙门氏菌煎蛋卷将导致下面的错误：

```

>>> make_omelet_q3({'mozzarella cheese':5, 'eggs':5, 'milk':4, 'roast red
pepper':6, 'mushrooms':4}, "salmonella")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "ch5.py", line 209, in make_omelet_q3

```

```

    omelet_ingredients = get_omelet_ingredients(omelet_type)
File "ch5.py", line 179, in get_omelet_ingredients
    raise TypeError, "We run a clean kitchen, you won't get this here"
TypeError: We run a clean kitchen, you won't get this here
>>>

```

注意, 取决于文件 `ch5.py` 的内容, 您的栈跟踪中显示的确切行数将会和这里的显示有所出入。

然后, 可以看到第 179 行是 `get_omelet_ingredients` 产生错误的地方(在您的文件中可能是其他的行)。

如果在其他函数中调用该函数, 调用栈将更深一层, 您也会看到与这个层相关的信息。

## A.6 第 6 章

### 习题 1 解答

```

def mix(self, display_progress = True):
    """
    mix(display_progress = True) - Once the ingredients have been
    obtained from a fridge call this
    to prepare the ingredients. If display_progress is False do not print
    messages.
    """
    for ingredient in self.from_fridge.keys():
        if display_progress == True:
            print("Mixing %d %s for the %s omelet" %
                  (self.from_fridge[ingredient], ingredient, self.kind))
        self.mixed = True

```

### 习题 2 解答

注意, 您也可以更进一步, 使混合函数的自动设置成为一个选项。这样, 您将无法得到关于正在进行的工作的大量反馈, 因此当您测试它时, 看起来可能会有点奇怪。

```

def quick_cook(self, fridge, kind = "cheese", quantity = 1):
    """
    quick_cook(fridge, kind = "cheese", quantity = 1) -
    performs all the cooking steps needed. Turns out an omelet fast.
    """

    self.set_kind(kind)
    self.get_ingredients(fridge)
    self.mix(False)
    self.make()

```

## 习题 3 解答

下面列出了几个只包含文档而不包括函数的类。然而，您应该找到一个适合自己的格式。注意，此处只有那些文档中没有列出的函数才拥有文档字符串。

```
class Omelet:
    """This class creates an omelet object. An omelet can be in one of
    two states: ingredients, or cooked.
    An omelet object has the following interfaces:
    get_kind() - returns a string with the type of omelet
    set_kind(kind) - sets the omelet to be the type named
    set_new_kind(kind, ingredients) - lets you create an omelet
    mix() - gets called after all the ingredients are gathered from the fridge
    cook() - cooks the omelet
    """
    def __init__(self, kind="cheese"):
        """__init__(self, kind="cheese")
        This initializes the Omelet class to default to a cheese omelet.
        Other methods
        """
        self.set_kind(kind)
        return

    def set_kind(self, kind):
        """
        set_kind(self, kind) - changes the kind of omelet that will be created
        if the type of omelet requested is not known then return False
        """

    def get_kind(self):
        """
        get_kind() - returns the kind of omelet that this object is making
        """

    def set_kind(self, kind):
        """
        set_kind(self, kind) - changes the kind of omelet that will be created
        if the type of omelet requested is not known then return False
        """

    def set_new_kind(self, name, ingredients):
        """
        set_new_kind(name, ingredients) - create a new type of omelet that is
        called "name" and that has the ingredients listed in "ingredients"
        """

    def __known_kinds(self, kind):
        """
        __known_kinds(kind) - checks for the ingredients of "kind" and returns them
        returns False if the omelet is unknown.
        """
```

```

def get_ingredients(self, fridge):
    """
    get_ingredients(fridge) - takes food out of the fridge provided
    """

def mix(self):
    """
    mix() - Once the ingredients have been obtained from a fridge call this
    to prepare the ingredients.
    """

def make(self):
    """
    make() - once the ingredients are mixed, this cooks them
    """

```

## 习题 4 解答

```

>>> print("%s" % o.__doc__)
This class creates an omelet object. An omelet can be in one of
two states: ingredients, or cooked.
An omelet object has the following interfaces:
get_kind() - returns a string with the type of omelet
set_kind(kind) - sets the omelet to be the type named
set_new_kind(kind, ingredients) - lets you create an omelet
mix() - gets called after all the ingredients are gathered from the fridge
cook() - cooks the omelet

>>> print("%s" % o.set_new_kind.__doc__)

    set_new_kind(name, ingredients) - create a new type of omelet that is
    called "name" and that has the ingredients listed in "ingredients"

```

可以以相同的方式显示其余的文档字符串。

## 习题 5 解答

```

class Recipe:
    """
    This class houses recipes for use by the Omelet class
    """

    def __init__(self):
        self.set_default_recipes()
        return

    def set_default_recipes(self):
        self.recipes = {"cheese" : {"eggs":2, "milk":1, "cheese":1},
                        "mushroom" : {"eggs":2, "milk":1, "cheese":1,

```

```

    "mushroom":2},
        "onion" : {"eggs":2, "milk":1, "cheese":1,
    "onion":1}}

    def get(self, name):
        """
        get(name) - returns a dictionary that contains the ingredients needed to
        make the omelet in name.
        When name isn't known, returns False
        """
        try:
            recipe = self.recipes[name]
            return recipe
        except KeyError:
            return False

    def create(self, name, ingredients):
        """
        create(name, ingredients) - adds the omelet named "name" with the
                                   ingredients
        "ingredients" which is a dictionary.
        """

        self.recipes[name] = ingredients

```

## 习题 6 解答

注意类的接口中参数的顺序现在已经改变了，因为不能在拥有默认值的可选参数后面放置一个必需参数。

当进行测试时，记住现在创建了一个以食谱作为必需参数的煎蛋卷类。

```

def __init__(self, recipes, kind="cheese"):
    """__init__(self, recipes, kind="cheese")
    This initializes the omelet class to default to a cheese omelet.

    """
    self.recipes = recipes
    self.set_kind(kind)

    return

def set_new_kind(self, name, ingredients):
    """
    set_new_kind(name, ingredients) - create a new type of omelet that is
    called "name" and that has the ingredients listed in "ingredients"
    """
    self.recipes.create(name, ingredients)
    self.set_kind(name)
    return

```

```
def __known_kinds(self, kind):
    """
    __known_kinds(kind) - checks for the ingredients of "kind" and returns them
    returns False if the omelet is unknown.
    """
    return self.recipes.get(kind)
```

## A.7 第 7 章

### 习题 1 解答

当编写测试时，记住您不是类的普通用户。在需要的情况下，您可以自由地访问类的内部名称。

```
if __name__ == '__main__':
    r = Recipe()
    if r.recipes != {"cheese" : {"eggs":2, "milk":1, "cheese":1},
                    "mushroom" : {"eggs":2, "milk":1, "cheese":1,
                                   "mushroom":2},
                    "onion" : {"eggs":2, "milk":1, "cheese":1,
                               "onion":1}}:
        Print("Failed: the default recipes is not the correct list")
    cheese_omelet = r.get("cheese")
    if cheese_omelet != {"eggs":2, "milk":1, "cheese":1}:
        print("Failed: the ingredients for a cheese omelet are wrong")
    western_ingredients = {"eggs":2, "milk":1, "cheese":1, "ham":1,
                           "peppers":1, "onion":1}
    r.create("western", western_ingredients)
    if r.get("western") != western_ingredients:
        print("Failed to set the ingredients for the western")
    else:
        print("Succeeded in getting the ingredients for the western.")
```

### 习题 2 解答

在 Fridge 模块的结尾，插入下面的代码。注意关于修改 `add_many` 函数使其返回 `True` 的注释。如果不那样做，`add_many` 将返回 `None`，并且这个测试将总是失败。

```
if __name__ == '__main__':
    f = Fridge({"eggs":10, "soda":9, "nutella":2})
    if f.has("eggs") != True:
        print("Failed test f.has('eggs')")
    else:
        print("Passed test f.has('eggs')")
    if f.has("eggs", 5) != True:
        print("Failed test f.has('eggs', 5)")
```

```

else:
    print("Passed test f.has('eggs', 5)")
if f.has_various({"eggs":4, "soda":2, "nutella":1}) != True:
    print('Failed test f.has_various({"eggs":4, "soda":2, "nutella":1})')
else:
    print('Passed test f.has_various({"eggs":4, "soda":2, "nutella":1})')
# Check to see that when we add items, that the number of items in the fridge
# is increased!
item_count = f.items["eggs"]
if f.add_one("eggs") != True:
    print('Failed test f.add_one("eggs")')
else:
    print('Passed test f.add_one("eggs")')
if f.items["eggs"] != (item_count + 1):
    print('Failed f.add_one() did not add one')
else:
    print('Passed f.add_one() added one')
item_count = {}
item_count["eggs"] = f.items["eggs"]
item_count["soda"] = f.items["soda"]
# Note that the following means you have to change add_many to return True!
if f.add_many({"eggs":3, "soda":3}) != True:
    print('Failed test f.add_many({"eggs":3, "soda":3})')
else:
    print('Passed test f.add_many({"eggs":3, "soda":3})')
if f.items["eggs"] != (item_count["eggs"] + 3):
    print("Failed f.add_many did not add eggs")
else:
    print("Passed f.add_many added eggs")
if f.items["soda"] != (item_count["soda"] + 3):
    print("Failed f.add_many did not add soda")
else:
    print("Passed f.add_many added soda")

item_count = f.items["eggs"]
if f.get_one("eggs") != True:
    print('Failed test f.get_one("eggs")')
else:
    print('Passed test f.get_one("eggs")')
if f.items["eggs"] != (item_count - 1):
    print("Failed get_one did not remove an eggs")
else:
    print("Passed get_one removed an eggs")

item_count = {}
item_count["eggs"] = f.items["eggs"]
item_count["soda"] = f.items["soda"]
eats = f.get_many({"eggs":3, "soda":3})
if eats["eggs"] != 3 or eats["soda"] != 3:
    print('Failed test f.get_many({"eggs":3, "soda":3})')

```

```

else:
    print('Passed test f.get_many({"eggs":3, "soda":3})')

if f.items["eggs"] != (item_count["eggs"] - 3):
    print("Failed get many didn't remove eggs")
else:
    print("Passed get many removed eggs")

if f.items["soda"] != (item_count["soda"] - 3):
    print("Failed get many didn't remove soda")
else:
    print("Passed get many removed soda")

```

### 习题 3 解答

可以试着在模块中的某个地方输入错误的键名以产生错误，并且确认测试会给出警告。如果您发现在某些情形下，这些测试不能捕获异常，应该试着为那些情形编写测试，以免自己陷入错误中。

## A.8 第 8 章

### 习题 1 解答

下面的代码是解决该问题的一个简单但是低效的方法。

```

import os

def print_dir(dir_path):
    # Loop through directory entries, and print directory names.
    for name in sorted(os.listdir(dir_path)):
        full_path = os.path.join(dir_path, name)
        if os.path.isdir(full_path):
            print(full_path)

    # Loop again, this time printing files.
    for name in sorted(os.listdir(dir_path)):
        full_path = os.path.join(dir_path, name)
        if os.path.isfile(full_path):
            print(full_path)

```

下面是一个更好的解决方法，它只扫描一次目录，并只对目录进行一次排序。

```

import os

def print_dir(dir_path):
    # Loop through directory entries. Since we sort the combined
    # directory entries first, the subdirectory names and file names

```

```

# will each be sorted, too.
file_names = []
for name in sorted(os.listdir(dir_path)):
    full_path = os.path.join(dir_path, name)
    if os.path.isdir(full_path):
        # Print subdirectory names now.
        print(full_path)
    elif os.path.isfile(full_path):
        # Store file names for later.
        file_names.append(full_path)

# Now print the file names.
for name in file_names:
    print(name)

```

## 习题 2 解答

```

import os
import shutil

def make_version_path(path, version):
    if version == 0:
        return path
    else:
        return path + "." + str(version)

def rotate(path, max_keep, version=0):
    """Rotate old versions of file 'path'.

    Keep up to 'max_keep' old versions with suffixes .1, .2, etc.
    Larger numbers indicate older versions."""

    src_path = make_version_path(path, version)
    if not os.path.exists(src_path):
        # The file doesn't exist, so there's nothing to do.
        return

    dst_path = make_version_path(path, version + 1)
    if os.path.exists(dst_path):
        # There already is an old version with this number. What to do?
        if version < max_keep - 1:
            # Renumber the old version.
            rotate(path, max_keep, version + 1)
        else:
            # Too many old versions, so remove it.
            os.remove(dst_path)

    shutil.move(src_path, dst_path)

```

## A.9 第 10 章

### 习题 1 解答

通过导入模块或者模块中的项可以访问模块的功能。

### 习题 2 解答

如果定义了 `__all__` 变量，可以列出组成模块的公有 API 的那些项。例如：

```
__all__ = ['Meal', 'AngryChefException', 'makeBreakfast',  
          'makeLunch', 'makeDinner', 'Breakfast', 'Lunch', 'Dinner']
```

如果没有定义 `__all__` 变量(尽管应该定义)，Python 解释器将查找名称不以下划线开始的所有项。

### 习题 3 解答

`help` 函数显示导入的任何模块的帮助信息。基本语法如下：

```
help(module)
```

### 习题 4 解答

在变量 `sys.path` 列出的目录中查找您的系统中模块的位置。需要首先导入 `sys` 模块。

### 习题 5 解答

模块中可以放置任何 Python 命令。模块可以包含 Python 命令、Python 函数、Python 变量、Python 类等。但是在大多数情况下，应该避免在模块中运行命令。相反，模块应该定义一些函数和类，让调用者决定调用什么。

## A.10 第 11 章

### 习题 1 解答

```
import os, os.path  
import re  
  
def print_pdf (arg, dir, files):  
    for file in files:
```

```

    path = os.path.join (dir, file)
    path = os.path.normcase (path)
    if not re.search (r".*\..pdf", path): continue
    if re.search (r" ", path): continue

    print(path)

os.path.walk ('/', print_pdf, 0)

```

注意这个示例修改了目录名称，并使用 `os.path.walk` 函数开始进行处理。

## 习题 2 解答

```

import os, os.path
import re

def print_pdf (arg, dir, files):
    for file in files:
        path = os.path.join (dir, file)
        path = os.path.normcase (path)
        if not re.search (r".*\..pdf", path): continue
        if not re.search (r"boobah", path): continue

        print(path)

os.path.walk ('.', print_pdf, 0)

```

该示例仅在 `print_pdf` 函数中包含了一个附加测试。

## 习题 3 解答

```

import os, os.path
import re

def print_pdf (arg, dir, files):
    for file in files:
        path = os.path.join (dir, file)
        path = os.path.normcase (path)
        if not re.search (r".*\..pdf", path): continue
        if re.search (r"boobah", path): continue

        print(path)

os.path.walk ('.', print_pdf, 0)

```

注意，这个例子从第二个测试中删除了 `not`。

## A.11 第 14 章

### 习题 1 解答

最合适的选项是 b，人名作为键，比萨饼成分作为值，不同成分之间可以用逗号进行分隔。

### 习题 2 解答

可以使用您喜欢的任何别名。下面是一个例子：

```
select employee.firstname, employee.lastname, department.name
from employee, department
where employee.dept = department.departmentid
order by employee.lastname desc
```

### 习题 3 解答

不必修改很多内容。

```
import sys
import sqlite3

conn=sqlite3.connect('sample_database')
cursor = connection.cursor()

employee = sys.argv[1]

# Query to find the employee ID.
query = """
select e.empid
from user u, employee e
where username=? and u.employeeid = e.empid
"""
cursor.execute(query, (employee,));
for row in cursor.fetchone():
    if (row != None):
        empid = row

# Now, modify the employee.
cursor.execute("delete from employee where empid=?", (empid,))
cursor.execute("delete from user where employeeid=?", (empid,))

connection.commit()
cursor.close()
connection.close()
```

## A.12 第 15 章

### 习题 1 解答

```
from xml.dom.minidom import parse
import xml.dom.minidom

# open an XML file and parse it into a DOM
myDoc = parse('config.xml')
myConfig = myDoc.getElementsByTagName("config")[0]

#Get utility directory
myConfig.getElementsByTagName("utilitydirectory")[0].childNodes[0].data

#Get utility
myConfig.getElementsByTagName("utility")[0].childNodes[0].data

#get mode
myConfig.getElementsByTagName("mode")[0].childNodes[0].data

#.....Do something with data.....
```

### 习题 2 解答

```
#!/usr/bin/python

from xml.parsers.xmlproc import xmlval

class docErrorHandler(xmlval.ErrorHandler):
    def warning(self, message):
        print(message)
    def error(self, message):
        print(message)
    def fatal(self, message):
        print(message)

parser=xmlval.XMLValidator()
parser.set_error_handler(docErrorHandler(parser))
parser.parse_resource("configfile.xml")
```

### 习题 3 解答

```
#!/usr/bin/python

from xml.sax          import make_parser
from xml.sax.handler import ContentHandler

#begin configHandler
```

```

class configHandler(ContentHandler):
    inUtildir = False
    utildir = ''
    inUtil = False
    util = ''
    inMode = False
    mode = ''

    def startElement(self, name, attributes):

        if name == "utilitydirectory":
            self.inUtildir = True

        elif name == "utility":
            self.inUtil = True

        elif name == "mode":
            self.inMode = True

    def endElement(self, name):
        if name == "utilitydirectory":
            self.inTitle = False

        elif name == "utility":
            self.inUtil = False

        elif name == "mode":
            self.inMode = False

    def characters(self, content):
        if self.inUtildir:
            utildir = utildir + content
        elif self.inUtil:
            util = util + content
        elif self.inMode:
            mode = mode + content
    #end configHandler

parser = make_parser()
parser.setContentHandler(configHandler())
parser.parse("configfile.xml")

#....Do stuff with config information here

```

## A.13 第 16 章

### 习题 1 解答

RFC 2822 是描述 E-mail 消息格式的文件格式标准。

MIME 是描述如何创建包括二进制数据和多个部分的 E-mail 消息的文件格式标准，它仍然遵循 RFC 2822。

SMTP 协议用来将 E-mail 消息传递给其他人。

POP 协议用来从邮件服务器获取 E-mail。

IMAP 协议和 POP 协议的功能类似，但比后者更新。IMAP 协议在服务器端永久保存 E-mail，而不是仅保存到您获取它为止。

## 习题 2 解答

下面是一个使用 POP 的脚本：

```
#!/usr/bin/python
from poplib import POP3
from email import parser

#Connect to the server and parse the response to see how many messages there
#are, as in this chapter's previous POP example.
server = POP3("pop.example.com")
server.user("[user]")
response = server.pass_("[password]")
numMessages = response[response.rfind(',')+2:]
numMessages = int(numMessages[:numMessages.find(' ')])

#Parse each email and put it in a file named after the From: header of
#the mail.
parser = parser()
openFiles = {}
for messageNum in range(1, numMessages+1):
    messageString = '\n'.join(server.retr(messageNum)[1])
    message = email.parsestr(messageString, True)
    fromHeader = message['From']
    mailFile = openFiles.get(fromHeader)
    if not mailFile:
        mailFile = open(fromHeader, 'w')
        openFiles[fromHeader] = mailFile
    mailFile.write(messageString)
    mailFile.write('\n')
#Close all the files to which we wrote mail.
for openFile in openFiles.values():
    openFile.close()
```

因为 IMAP 使您能够将消息分类整理到服务器上的不同文件夹中，所以这个脚本的 IMAP 版本可以简单地创建新的邮箱，并将消息移入它们。这里是实现该功能的脚本：

```
#!/usr/bin/python
from imaplib import IMAP4
import email
```

```

import re

#Used to parse the IMAP responses.
FROM_HEADER = 'From: '
IMAP_UID = re.compile('UID ([0-9]+)')

#Connect to the server.
server = IMAP4('imap.example.com')
server.login('[username]', '[password]')
server.select('Inbox')

#Get the unique IDs for every message.
uids = server.uid('SEARCH', 'ALL')[1][0].split(' ')
uidString = ','.join(uids)

#Get the From: header for each message
headers = server.uid('FETCH', '%s' % uidString,
                     '(BODY[HEADER.FIELDS (FROM)])')
for header in headers[1]:
    if len(header) > 1:
        uid, header = header
        #Parse the IMAP response into a real UID and the value of the
        #'From' header.
        match = IMAP_UID.search(uid)
        uid = match.groups(1)[0]

        fromHeader = header[len(FROM_HEADER):].strip()

        #Create the mailbox corresponding to the person who sent this
        #message. If it already exists the server will throw an error,
        #but we'll just ignore it.
        server.create(fromHeader)

        #Copy this message into the mailbox.
        server.uid('COPY', uid, fromHeader)

#Delete the messages from the inbox now that they've been filed.
server.uid('STORE', uidString, '+FLAGS.SILENT', '(\\Deleted)')
server.expunge()

```

### 习题3 解答

通常，要将尽可能多的文本移出协议，移入客户端软件，后者只需要下载一次。下面列出了一些具体的建议：

- 发送尽可能短的状态代码，而不是英文句子。例如，发送“HELLO”而不是“Hello [nickname], welcome to the Python Chat Server!”。
- 为聊天室的每个用户分配一个数字，在他们执行任何操作时，发送数字而不是他们的昵称。例如，当用户发送一条消息时，广播“4 Hello”而不是“<user> Hello”。

- 使用压缩技术使聊天文本占用更少的带宽。

## 习题 4 解答

最简单的方法是定义一个 `msgCommand` 方法，并让 `_parseCommand` 分发它。下面是 `msgCommand` 的一个简单实现：

```
def msgCommand(self, nicknameAndMsg):
    "Send a private message to another user."
    if not ' ' in nicknameAndMsg:
        raise ClientError('No message specified.')
    nickname, msg = nicknameAndMsg.split(' ', 1)
    if nickname == self.nickname:
        raise ClientError('What, send a private message to yourself?')
    user = self.server.users.get(nickname)
    if not user:
        raise ClientError('No such user: %s' % nickname)
    msg = '[Private from %s] %s' % (self.nickname, msg)
    user.write(self._ensureNewline(msg))
```

## 习题 5 解答

- 对等架构比客户端-服务器架构更通用。TCP/IP 的对等设计使它成为一个灵活通用的协议。在 TCP/IP 上实现一个客户端-服务器协议比在客户端-服务器上实现对等设计更容易。如果希望得到一个通用的协议，设法保持 TCP/IP 的对等本质。
- 当客户端从服务器下载一些数据，然后立即向其他客户端提供该数据时，可以考虑使用对等架构。对等架构对于 E-mail 分发没有意义，因为大多数 E-mail 只发给一个人。一旦那个人下载了 E-mail，它不应该自动地进一步分发。对等架构对于新闻通讯的分发更有意义。
- 当有一些方法可以搜索网络时，对等是最有用的。当某个网络资源没有单一的、明确的位置(与文件存放在一个 Web 服务器上的情况相反)时，很难找到想要的资源，此时搜索工具变得很重要。

## A.14 第 18 章

### 习题 1 解答

```
def format_bytes(bytes):
    units = (
        ("GB", 1024 ** 3),
        ("MB", 1024 ** 2),
        ("KB", 1024 ** 1),
        ("bytes", 1),
    )
```

```

terms = []
for name, scale in units:
    if scale > bytes:
        continue
    # Show how many of this unit.
    count = bytes // scale
    terms.append("%d %s" % (count, name))
    # Compute the leftover bytes.
    bytes = bytes % scale
# Construct the full output from the terms.
return " + ".join(terms)

```

## 习题 2 解答

```

def rgb_to_html(red, green, blue):
    # Convert floats between zero and one to ints between 0 and 255.
    red = int(round(red * 255))
    green = int(round(green * 255))
    blue = int(round(blue * 255))
    # Write out HTML color syntax.
    return "#%02x%02x%02x" % (red, green, blue)

```

## 习题 3 解答

解决方法使用了一个数值列表：

```

from math import sqrt

def normalize(numbers):
    # Compute the sum of squares of the numbers.
    sum_of_squares = 0
    for number in numbers:
        sum_of_squares += number * number
    # Copy the list of numbers.
    result = list(numbers)
    # Scale each element in the list.
    scale = 1 / sqrt(sum_of_squares)
    for i in xrange(len(result)):
        result[i] *= scale
    return result

```

这个很简洁的 `numarray` 版本仅在用 `numarray.array` 对象调用时才能正常工作。可以利用 `numbers = numarray.array(numbers)` 转换一个不同的数组类型：

```

from math import sqrt
import numarray

def normalize(numbers):
    return numbers / sqrt(numarray.sum(numbers * numbers))

```

## A.15 第 20 章

### 习题 1 解答

将 wiki 的名称放入资源标识符中的页面名称前，所以是 “/Wikiname/PageName”，而不是 “/PageName”。这是符合 REST 的方式，因为它将数据放入资源标识符中，使其保持透明。这个标识符模式也对应于 wiki 文件存储在磁盘上的方式。

### 习题 2 解答

```
#!/usr/bin/python
import cgi
import cgitb
import os
from WishListBargainFinder import BargainFinder, getWishList
cgitb.enable()

SUBSCRIPTION_ID = '[Insert your subscription ID here.]'
SUBSCRIPTION_ID = 'D8010TR10IMN7'

form = cgi.FieldStorage()
wishListID = form.getfirst('wishlist', '')

args = {'title' : 'Amazon Wish List Bargain Finder',
        'action' : os.environ['SCRIPT_NAME'],
        'wishListID' : wishListID}

print('Content-type: text/html\n')
print('''<html><head><title>%(title)s</title></head>
<form method="get" action="%(action)s">
<h1>%(title)s</h1>
Enter an Amazon wish list ID:
<input name="wishlist" length="13" maxlength="13" value="%(wishListID)s" />
<input type="submit" value="Find bargains"/>
</form>''' % args

if wishListID:
    print('<pre>')
    BargainFinder().printBargains(getWishList(SUBSCRIPTION_ID, wishListID))
    Print('</pre>')

print('</body></html>')
```

注意 BargainFinder 中的改进：创建一个在数据结构中返回低价商品信息的方法，可以使用纯文本、HTML 或其他任何方式格式化它，而不是只打印出低价商品的纯文本。

### 习题 3 解答

对于 REST: BittyWiki Web 应用程序已经输出渲染后的 HTML, 因为 Web 浏览器知道如何解析它。然而, 一个由 Web 应用程序提供的 BittyWiki 页面除了包括页面文本以外, 还包括导航链接和其他元素。如果 Web 服务用户不愿意费力删除无关系的 HTML 来得到实际的页面文本, 或者如果您一开始就希望通过不发送 HTML 来节省带宽, 有其他两个解决方法。第一个解决方法是让 Web 服务客户端在 GET 请求中提供 HTML Accept 头, 以表达它们究竟希望得到资源的 “text/plain” 格式还是 “text/html” 格式。第二个方法是通过不同的资源提供相同文档的不同格式。例如, `/bittywiki-rest.py/PageName.txt` 能提供页面的纯文本版本, 而 `/bittywiki-rest.py/PageName.html` 能提供相同页面的 HTML 版本。

对于 XML-RPC 和 SOAP 而言, 决定很简单。让客户端向 `getPage` 传入一个参数, 指定它们想要的页面格式即可。

### 习题 4 解答

可以通过改变 GET 资源或 `getPage` API 调用来修复此问题, 使其不仅返回页面的原始文本, 还返回页面上与已存在的页面对应的 WikiWord 的表示。这可以是一个拥有相关页面的 WikiWord 列表, 也可以是一个将页面上所有被引用的 WikiWord 映射为 True(如果某个 WikiWord 有相关的页面)或 False(如果某个 WikiWord 没有相关页面)的字典。第二种解决方法的优点是, 它们使机器人不必自己保持对 WikiWord 的定义。

### 习题 5 解答

创建一个专门用于重命名页面的新 API 调用。在 XML-RPC 或 SOAP 中, 这只需要创建 `rename` 函数并删除 `delete` 函数。对于 REST API, 可以给 POST 请求添加一项创建新的 wiki 页面的能力: 不是提供数据, 而是让它指定 wiki 中作为数据源使用的另一个页面, 并知道这个页面之后将被删除。

## A.16 第 21 章

### 习题 1 解答

很多组织都在另外一门程序设计语言上进行投入。然而, Jython 允许在 Java 环境中使用 Python。

### 习题 2 解答

Jython 用 Java 编写。python 解释器用 C 编写。

### 习题 3 解答

除了需要包含自己的 Jython 中脚本，还需要包含下面的资源：

- jython.jar Java 库
- Jython Lib 目录
- Jython cachedir 目录。必须拥有这个目录的写权限。

### 习题 4 解答

不是，除非 DB 驱动程序用 Python 或 Java 编写。大多数 Python DB 驱动程序用 C 和 Python 编写，因此不能从 Jython 中运行，除非使用 Java 本地接口(Java Native Interface, JNI)做大量工作。幸运的是，Jython zxJDBC 模块使您能够从 Jython 脚本调用任何 JDBC 驱动程序。通过这种方法能够访问的数据库比可以得到其 Python DB 驱动程序的数据库更多。

### 习题 5 解答

这可能是创建这样一个窗口的最简单的方法：

```
from javax.swing import JFrame

frame = JFrame(size=(500,100))

# Use a tuple for RGB color values.
frame.background = 255,0,0

frame.setVisible(1)
```

为了美观，还可以加入像按钮和标签这样的小组件。



## 附录

## B

## 在线资源

Python 是从 Internet 上可以得到的软件, 而 Python 最好的日常资源也都可以从 Internet 上找到。本附录描述了本书中使用的软件以及如何安装它们。

大多数 Python 相关的软件可以免费下载, 并且大多数这样的软件的源代码都可以下载并编译。对于那些从本书第 II 部分开始阅读的读者, 这可能是他们正在寻找的挑战。然而, 本书的大部分读者将非常乐意知道, 运行本书的示例所需的全部资源都可以在支持它们的操作系统上作为包来安装。

### B.1 软件

本书中的示例要求计算机上装有额外的软件, 以及一个合适的操作系统, 例如 Windows 2000、XP、XP Pro、2003 或 Vista; Linux(Red Hat 的 Fedora RC3 版本或更新版本; Debian 测试版或不稳定版; 或类似的当前版本)、Ubuntu 或 Mac OSX。

下面是所需软件的一个简单列表, 其中包含对软件的描述和可以下载到软件的 URL:

- **Python:** [www.python.org/](http://www.python.org/) 是 Python 语言的主页。可以在这里找到 Python 的所有资源, 包括在线教程、对语言的介绍和帮助解决问题的邮件列表等。编写、维护、修改和使用 Python 的人也经常在线。在那里也可以找到一个完整的、简明的文档集合。本书使用的软件版本是 Python 3.1.1, 可以单击 Python 主页顶部的 Download 链接, 或者直接到 [www.python.org/download/](http://www.python.org/download/) 下载它。如果足够幸运, 您可能会找到可以使用的更新版本的 Python。在本书出版时, Python 3.1.1 已经发布了。  
对于 Windows 用户, 使用最新的 Python 3.1.1 安装程序的 Windows .msi 版本进行安装。  
对于 Linux 系统用户, 安装系统维护者提供的 Python 包(例如, 来自 [debian.org](http://debian.org) 的 .deb 包, 或者来自 [redhat.com](http://redhat.com) 的 .rpm, 请参考 [www.python.org/download/releases/3.1.1/](http://www.python.org/download/releases/3.1.1/) 上的信息)。  
对于 Mac 用户, 可以在 [www.python.org/download/mac/](http://www.python.org/download/mac/) 上找到关于 Mac 上的 Python 3.1.1 的信息。
- **Tkinter:** 本书中的 GUI 编程一章使用 tkinter 接口编写, 它使您可以在 Python 内部访问 Tcl/Tk 图形用户界面工具包。它是跨平台的, 可以移植到每个系统上。

更多信息可以访问 <http://wiki.python.org/moin/TkInter>。

- **PyUnit**: 用于 Python 的单元测试框架。这个模块提供了一个在源代码内编写测试的系统化方法,使您可以验证自己的代码是否按照期望的方式工作。PyUnit 现在是标准 Python 库的一部分,它也称为 unittest。

PyUnit 的主页位于 <http://pyunit.sourceforge.net/>。

- **MySQL**: 流行的、快速的开源关系数据库系统。Python 有坚固的 MySQL 支持。
  - [www.mysql.com/](http://www.mysql.com/)——mysql.com 的主页。mysql.com 是维护 MySQL 数据库的公司。
  - <http://sourceforge.net/projects/mysql-python>——mysql-python 模块的主页,但是只有少量的在线文档。
- **Jython**: Python 语言的纯 Java 实现。Jython 可以访问商用 Java 产品空间的所有可用工具,但是它允许使用 Python 语言编程。详细信息可以访问 [www.jython.org/](http://www.jython.org/)。
- **Sqlite3**: 本书的数据库部分使用 Sqlite3 创建简单的数据库结构。它是一个使用 C 语言编写的轻量级库,与 DB-API 2.0 兼容。可以在 <http://docs.python.org/library/sqlite3.html> 找到更多关于 Sqlite3 的信息。
- **Django**: 用于 Python 的高层 Web 框架。Django 是建立和立即运行站点的重要工具。Django 非常适合数据库驱动的站点和 Web 应用程序,它通过为开发者建立基本“框架”,帮助他们节省时间。要下载该框架和阅读更多关于它的信息,请访问 <http://www.djangoproject.com/>。

## B.2 更多信息

可以从 Internet 上找到很多与 Python 相关的信息。另外,还可以在 Internet 上找到与本书用到的特殊组件相关的信息。由于 Python 及其模块经常改变,所以请访问 [www.wrox.com](http://www.wrox.com) 上本书的 Web 页面。可以在这里查找希望安装的软件的帮助信息,下载示例,提供反馈,并接受有关本书中任何问题的帮助。另外,还可以在本书的网站上找到这里提到的一些资源的安装包和信息。



## 附录

## C

## Python 3.1 的新特性

Python 以很小的方式不断地改变着。Python 3.1 从 2.6 版演化而来，但是它包含了一些重要变化。本附录将介绍同本书的主题相关的变化。这意味着本附录没有完整介绍 Python 3.1 的全部新特性，而只是介绍了本书中涉及到的主题，Python 初学者应该了解这些主题。

在 <http://docs.python.org/3.1/whatsnew/3.1.html> 上可以找到介绍 Python 3.1 中的变化的官方列表。如果在阅读本书时有更新的 Python 版本，也可以在 Python 网站上找到关于那个版本的变化列表。

### C.1 print 现在是一个函数

在过去，`print` 是一个语句。在 Python 3.1 中，它已经变成一个真正的函数，明确地说是 `print()`。

### C.2 某些 API 返回视图和迭代器

下面的 API 不再返回列表，而是返回视图和迭代器：

- 字典方法 `dict.keys()`、`dict.items()` 和 `dict.values()`。也要注意，Python 3.1 中不再支持 `dict.iterkeys()`、`dict.iteritems()` 和 `dict.itervalues()`。
- `map()` 和 `filter()` 都返回迭代器而不是列表。
- `range()` 方法代替了 `xrange()` 方法，其用法相同。
- `zip()` 方法现在返回一个迭代器。

### C.3 整型

`long` 数据类型已经被重命名为 `int` (现在 `int` 基本上 is 唯一的整数类型)。它和 `long` 类型的工作方式基本一致。整数不再有限制，因此，`sys.maxint` 已经被废弃了。另外，在做除法时，例如 `2/4`，将得到浮点数。如果想要截断后的结果，可以使用 `2//4`。

## C.4 Unicode 和 8 位编码

Unicode 和 8 位的字符串已经被文本和二进制数据代替。所有文本都被认为是 Unicode 编码，但是编码的 Unicode 现在被表示为严格的二进制数据。因此，文本存储在 `str` 中，而数据存储在 `bytes` 中。如果试图混合这两种数据类型，将导致 `TypeError` 发生。如果希望混合使用 `str` 和 `bytes`，必须首先转换它们。例如，如果希望将 `byte` 转换为 `str`，可以使用 `bytes.decode()` 实现该功能。要将 `str` 转换为 `byte`，可以使用 `str.encode()`。

另一个变化使用字面值的方法。对于 Unicode 文本，`u "..."` 字面值已经不再使用，而对于二进制数据，`b "..."` 字面值仍然可用。

Unicode 和 8 位编码发生了很多变化，这里无法一一介绍。在 <http://docs.python.org/dev/py3k/whatsnew/3.1.html> 可以查看 Unicode 和 8 位编码的新变化。

## C.5 异常

抛出异常的用法已经被替换。不再将它写为：`raise Exception, "I take exception to that!"`，而是改用下面的方式：

```
exception("I take exception to that!")
```

同样，如果想捕获一个异常，可以用下面的方式编码：

```
try:
    a=int("hotdog")
except ValueError as oops:
    print("ValueError has occurred ", oops)
```

这将返回：

```
ValueError has occurred invalid literal for int() with base 10: 'hotdog'
```

关于异常，还有一些其他的改变。例如，所有异常对象都使用 `__traceback__` 属性存储栈跟踪。另外，`StandardError` 被删除了。

## C.6 类

旧风格的类已经从 Python 3.1 中彻底删除了。这留给我们一个基于新风格的类的简单的单个对象模型。这些类的定义和它们以前的版本类似，然而，对象现在隐式地是一个超类。

## C.7 比较、操作符和方法

Python 3.1 中，比较操作符的工作方式有了一些改变。对于初学者，比较现在必须符合逻辑。例如，不能使用 `0>none`。在过去的版本中这将返回 `False`，但是因为不能比较 0 和空，它现在返回一个错误。

函数 `cmp()` 和方法 `__cmp__()` 都被删除了。

至于操作符，它们发生了下面的变化：

- `Unbound` 方法已经被删除。
- 目前操作符 `!=` 的返回值与 `==` 的返回值完全相反。
- `Next()` 已经被重命名为 `__next__()`。
- 下面的方法都已经被删除：`__delitem__()`，`__getslice__()`，`__hex__()`，`__members__`，`__methods__`，`__oct__()` 和 `__setslice__()`。

## C.8 语法变化

Python 3.1 中有很多语法都发生了变化。同样，这个变化的语法列表太大，以至于不能在这里有限的篇幅中都涵盖到。下面描述了一些比较重要的变化。

关键字 `as`、`with`、`True`、`False` 和 `None` 成为保留字。

使用元类时要注意，旧方法

```
Class Example:
    __metaclass__ = Apple
...
```

不再合法。要改写为：

```
Class Example(metaclass=Apple):
...
```

另外，模块级全局变量 `__metaclass__` 被删除。

编写列表解析的旧方法是：

```
[for var in example1, example2, example 3]
```

现在改为：

```
[for var in (example1, example2, example3)]
```

旧的 `<>` 已删除，并被 `!=` 替换。

字符串字面值和整数字面值也发生了改变。字符串字面值不再接受以 `u` 和 `U` 开头的值，而整数字面值不再接受以 `l` 或 `L` 开头的值。

关键字 `exec()` 已经被删除，尽管它仍然是一个函数。

## C.9 包和模块

Python 3.1 中删除了下面的模块。注意这不是一个完整的列表：

- `cfmfile`
- `cl`
- `md5` 和 `sha` (由 `hashlib` 代替)
- `mimetools`, `MimeWriter`, `mimify`, `multifile` 和 `rfc822` (由 `e-mail` 包代替)
- `posixfile`
- `sv`
- `timing` (用 `time.clock` 代替)
- `Canvas`
- `commands` 和 `popen2` (由 `subprocess` 代替)
- `compiler`
- `dircache`
- `dl`
- `fpformat`
- `htmlib` (由 `HTMLParser` 代替)
- `mhlib` (由 `mailbox` 代替)
- `stat` (改为 `os.stat`)
- `urllib` (由 `urllib2` 代替)

另外，下面的模块被重新命名：

- `_winreg` 现在是 `winreg`。
- `ConfigParser` 现在是 `configparser`。
- `copy_reg` 现在是 `copyreg`。
- `Queue` 现在是 `queue`。
- `SocketServer` 现在是 `socketserver`。
- `markupbase` 现在是 `_markupbase`。
- `repr` 现在是 `reprlib`。
- `test.test_support` 现在是 `test.support`。

为了简化操作，Python 3.1 中把一些相似的模块组织到一个单独的包中。例如：

- `dbm` 现在包括：`anydbm`, `dbhash`, `dbm`, `dumbdbm`, `gdbm`, `whichdb`。
- `html` 现在包括：`HTMLParser` 和 `htmlentitydefs`。
- `http` 现在包括：`httplib`, `BaseHTTPServer`, `CGIHTTPServer`, `SimpleHTTPServer`, `Cookie` 和 `cookielib`。
- `tkinter` 现在包括除 `turtle` 以外的全部 Tkinter 相关的模块。
- `urllib` 现在包括 `urllib`, `urllib2`, `urlparse` 和 `robotparse`。

- xmlrpc 现在包括 xmlrpclib, DocXMLRPCServer 和 SimpleXMLRPCServer。

## C.10 内置函数与类型

Python 3.1 中删除了下面的内置函数和类型：

- apply()
- callable()
- coerce()
- execfile()
- file 类型
- reduce()
- reload()
- dict.has\_key()

另外, raw\_input() 改为 input()。

## C.11 2to3 工具

虽然 2to3 工具不能将全部 Python 2x 代码转换为 3x 代码, 但它确实可以在很多领域提供帮助。基本上, 这个程序所做的是接受已经存在的代码并对它应用一组修复器, 将旧的代码转换为新的代码。例如, 如果在下面的代码上运行 2to3:

```
print "Hi, my name is James and I am a Pythonaholic"
```

它将被转换为:

```
print("Hi, my name is James and I am a Pythonaholic")
```

很不错, 不是吗? 当然, 使用这个工具时有许多地方需要注意。首先, 运行的代码必须可以正确工作, 所以要严格测试 Python 2x 代码, 以保证它们的正确性。其次, 必须注意 2to3 不会修复全部代码, 有一些代码它也没有办法转换。对于这些代码, 2to3 将打印一条警告消息, 您需要手动修改它们。

要浏览更多关于使用 2to3 的信息和它的修复器的文档, 请访问位于 <http://docs.python.org/dev/py3k/library/2to3.html#to3-reference> 上的 Python 文档。



## 附录

## D

## 术 语 表

本书用到了下面的术语。

**127.0.0.1** 一个用来表示“本机”的特殊的 IP 地址。参见“本地主机”。

**匿名(Anonymous)** 匿名函数和变量没有绑定名称。匿名函数的例子包括由 `lambda` 函数创建的函数，以及被创建的没有关联名称的列表或元组。

**Base64** MIME 定义的编码策略，它将整个字符串作为一个整体进行转义。对于二进制数据，Base64 编码比 quoted-printable 效率更高。

**BitTorrent** 一个对等协议，在所有下载它的人中间分配托管文件的开销。

**调用栈(Call Stack)** 当代码执行时，调用栈是代码到达程序中某处时执行的函数的列表。当进入一个函数或方法时，文件当前所处的位置同函数调用的参数一起被记录，并且函数的入口也被标记在调用栈中。函数退出时，需要删除它在调用栈中的入口。当发生异常时，打印栈跟踪信息，以指出问题发生在程序的哪个地方。

**CGI** 公共网关接口(Common Gateway Interface)。一个 Web 服务器标准，使得将 Web 接口展现给脚本变得容易。

**类** ①类是一个可以用来创建对象的定义。类定义包括数据和方法的声明，该类的对象可以使用这些数据和方法。在 Python 中，将出现在类中的函数看做方法。② 对象保存数据和操作数据的方法。类定义了存储什么数据和采用什么方法。Python 与大多数程序设计语言(例如 Java、C++或 C#)相比较为宽松，因为 Python 打破了其他语言的强制规则。例如，Python 默认允许访问类中的数据。这的确违反了面向对象编程的一些概念，但是这是因为 Python 首要的目标是实用。

**客户端-服务器(Client-Server)** 描述了一个架构，其中一个参与者(服务器)存储其他参与者(客户端)请求并操作的信息。

**注释(Comment)** 注释是程序中 Python 并不关注的文本。对于没有包含在字符串内的 #号，Python 解释器会忽略从#开始一直到行尾所包含的所有文本。

**内容类型(Content Type)** MIME 概念，用来表示在 E-mail 中编码并发送的文件的类型。Web 服务器也用它来表示正在提供的文件的类型。

**DB API** 用于访问数据库的 Python API。这个 API 的好处是，可以使用相同的 Python 代码处理任何具有兼容驱动程序的数据库。这包括 Oracle、DB2 等。代码中唯一不同之处可能是连接数据库的代码，这些代码随供应商不同而发生变化。

**DBM** 数据库管理器(Database Manager)的缩写。DBM 库提供了持久化 Python 字典的

方法。

**字典(Dictionary)** Python 中的数据类型，它通过编程人员设定的任意值进行索引。值可以是任何 Python 对象。索引被称为“键”，键引用的对象作为键的值。

**DNS 域名系统(Domain Name System)**。运行于 TCP 之上的服务，能够将主机名(wrox.com)解析为 IP 地址(208.215.179.178)。

**文档模型(Document Model)** 描述文档的词汇和结构的一种方法。它定义文档中将要出现的数据元素，元素之间的关系以及元素的数量。

**DOM 文档对象模型(Document Object Model)**，W3C 推荐的、用于处理 XML 文档的、基于树的 API。

**DTD 文档类型定义(Document Type Definition)**。用于产生文档模型的规范。

**动态端口(Dynamic Port)** 见“临时端口”。

**封装(Encapsulation)** 封装是一种思想，它使得类可以隐藏内部细节和执行某个任务所需要的数据。类保存需要的数据，并且正常情况下外部不应该看到这些数据。而且，类提供了多种操作数据的方法。这些方法可以隐藏内部的细节，例如网络协议、磁盘访问等。封装是一项简化程序的技术。在创建程序的每一步中，可以编写集中处理单个任务的代码。封装隐藏了实现的复杂性。

**加密(Encryption)** 隐藏信息的行为，使得如果没有密钥将很难或不可能恢复它。数据如果可以恢复，被认为是加密的数据。数据在混乱和不可恢复时则被认为丢失。

**临时端口(Ephemeral Port)** 端口号较高的 IP 端口，经常用来通过 TCP/IP 接受作为特定套接字连接一部分的数据。临时端口由操作系统管理，生存时间为单个套接字连接的存在时间。

**转义序列(Escape Sequence)** 以反斜杠开始的特殊字符，例如\n表示一个新行。

**故障(Fault)** Web 服务中使用的术语，表示一个错误条件。与 Python 的异常类似，并且一般与 Python 中的异常具有相同的实现方法。

**浮点(Float)** 浮点数是拥有小数部分的数。分数可以用一个浮点值表示为小数值。当浮点数和整数做算术运算时，整数将被提升为浮点数。

**函数(Function)** 函数是使用一个名称定义的代码集合，并且这些代码通过该名称调用。

**头(Header)** 在 E-mail 消息及 HTTP 请求和响应中可以找到的元数据项。头行由通过一个冒号和一个空格分隔的键与值组成。例如：Subject: Hello。

**十六进制(Hexadecimal)** 以 16 为基数的计数法，它包含的数字为 0~15，9 以上的值使用字母 A~F 表示。所以在十六进制中，数字 11 表示为 B。

**主机名(Hostname)** 表示 IP 网络中的计算机的人类可读的标识符，例如 wrox.com。主机名通过 DNS 进行管理。

**HTTP 主体(HTTP Body)** HTTP 请求或响应的数据部分。

**HTTP 头(HTTP Header)** 一个 HTTP 请求或响应的元数据部分，为一系列键-值对。HTTP 定义了一些标准的头，CGI 定义的标准头更多：应用程序可以定义它们自己的 HTTP 头。

**HTTP 超文本传输协议(HyperText Transfer Protocol)**，为 Web 浏览器和 Web 服务器之间的通信设计的协议。

**HTTP 请求(HTTP Request)** HTTP 客户端发送给服务器的字符串，请求对某些资源

的一些操作。

**HTTP 响应(HTTP Response)** HTTP 服务器发送给客户端的字符串, 作为对 HTTP 请求的响应。在 REST 术语中, 它包括资源的表示或描述对资源执行的操作的文档。

**HTTP 状态代码(HTTP Status Code)** HTTP 响应中使用的数字编码, 表示相应请求的状态。HTTP 标准中定义了 40 个状态代码。

**HTTP 动词(HTTP Verb)** HTTP 请求中使用的字符串, 描述客户端希望对资源执行的操作(例如, 获得资源的表示或修改资源)。

**幂等(Idempotent)** 幂等操作没有任何附加作用。它来自数学术语: 用 1 乘以一个数是一个幂等操作。所以它应该调用一个对象的访问方法或(在 REST 中)发送一个 HTTP GET 请求。

**虚数(Imaginary Number)** 一个与浮点数类似的特殊的数, 但是它不能自由地同浮点数或整数混合。如果虚数与它们混合, 结果是一个复数, 而不是一个虚数。

**IMAP** Internet 消息访问协议(Internet Message Access Protocol)。也称为 IMAP4。这是一个用于取回或管理邮件的协议。IMAP4 允许在服务器上存储邮件。参见 POP。

**无限循环(Infinite Loop)** 没有终止语句的循环, 例如“while True:”。通常, 无限循环是一个意外的情况, 但是只要有动作即将发生或者有代码正在执行, 无限循环可能是有用的。一个例子是服务器在等待连接。

**继承(Inheritance)** 继承意味着一个类可以继承或访问父类中定义的数据和方法。这正遵循了对一个问题领域进行分类的常识。例如, 矩形和圆都是形状。这里, 基类可以称为 Shapes。Rectangle 类和 Circle 类可以从 Shapes 继承。继承允许将 Rectangle 类和 Circle 类的对象视为 Shapes 类, 这意味着可以在基类中编写更通用的代码, 而在子类中使代码变得更具体。对于大多数情况, 基类应该是通用的, 而子类应该是特殊的。继承也通常被称为特化。

**输入/输出(Input/Output)** 包括任何类型的读/写操作的术语。写到屏幕、从键盘输入和网络连接都是输入/输出的例子。

**整型(Integer)** 不含小数部分的数。

**I/O** 见“输入/输出”。

**IP 地址(IP Address)** 一个计算机在 IP 网络上所处的位置。例如, 208.215.179.178。

**IP** Internet 协议(Internet Protocol)。利用不同的技术(例如, 以太网或无线技术)将多个网络连接为一个网络。

**IRC** Internet 中继聊天(Internet Relay Chat)协议。一个用于在线聊天室的协议。

**迭代器(Iterator)** 在某些情形下, 可以使用迭代器对象生成一个输出序列。与序列对象不同, 像 xrange 这样的迭代器不需要返回一个有限长度的列表。当它的下一个方法被调用时, 这个对象可以继续创建返回值。迭代器可以用在 for 循环中。

**J2EE** Java 2 企业版(Java 2 Enterprise Edition), 编写企业级 Java 应用程序的一组标准。它们不存在真正对应的 Python 标准, 但是 Twisted 等框架为 Python 提供了企业级特性。

**JVM** Java 虚拟机(Java Virtual Machine), Java 平台的运行时引擎。java 命令运行 Java 应用程序的方法与 Python 命令运行 Python 应用程序的方法相似。

**Jython** 一个用 Java 语言编写的运行于 Java 平台之上的 Python 实现。

**列表(List)** 列表是一个序列类型，也是一个迭代器。它和元组相似，不过在创建后可被修改。列表使用方括号([])创建。

**本地主机(Localhost)** 一个特殊的主机名，用来表示“本机”。参见“127.0.0.1”。

**循环(Loop)** 循环是一种重复的形式，它重复执行一组操作，直到满足事先设置的一组条件。

**方法(Method)** 方法是对象内部的函数(在类内编写函数时也称这个函数为方法)。它自动可以访问调用它的对象内部的所有数据。

**MIME** 多用途 Internet 邮件扩展(Multipurpose Internet Mail Extension)协议。在遵循 RFC 2822 的同时，这组标准允许通过 E-mail 发送多个文件以及国际数据和二进制数据。

**模块(Module)** 模块是文件内代码的集合。模块可以包括函数、命名变量和类。当在一个程序中使用模块时，使用 `import` 关键字可以导入它，它在以模块名称命名的作用域内都有效。所以一个 `mymodule` 模块中的函数 `myfunction` 应该通过 `mymodule.myfunction()` 来调用。这可以通过改变模块导入方式来更改：导入修饰符 `from` 和 `as` 可以修改导入行为，使模块看起来拥有不同的名称。当前模块可以通过查看变量名 `__name__` 找到，它在每个模块的作用域内创建。如果 `__name__` 是 `__main__`，那么当前的作用域是顶层模块，即程序开始运行的地方。

**多部分消息(Multipart Message)** 一条 MIME 消息，包括多个“文档”(例如，一条文本消息和一个图像)。

**对象(Object)** 对象是类的实例。对象包括类中定义的数据和方法。相同类的多个对象可以使用不同的名称同时存在于一个程序中。每个对象拥有的数据都与相同类型的其他对象不同。对象被创建时与一个名称绑定。

**八进制(Octal)** 以 8 为基数的计数法，其中数字范围为 0~7。

**包(Package)** 包是目录中的一组模块，其包含一个称为 `__init__.py` 的文件。目录中的所有文件可以一起运行以实现一个看起来组合在一起的包，包使用起来与单个模块类似。模块包括子目录，子目录中也可以包括模块。包为发布更复杂的程序结构提供了一个有组织结构，并且它也允许有条件地包含只能在一个平台上运行的代码(例如，如果一个文件只能运行在 Mac OS X 系统上，那么可以将它放入单独的文件，并且只在验证了正确的平台后调用它)。

**对等(Peer-to-Peer)** 描述所有参与者拥有平等身份的一种架构。

**多态(Polymorphism)** 多态的含义是子类可以重写父类的方法来完成更特殊的行为。例如，`Rectangle` 和 `Circle` 都是 `Shapes`。可以定义一个共同的操作集合，例如 `move` 和 `draw`，它们适用于所有的形状。然而，`Circle` 的 `draw` 方法显然和 `Rectangle` 的 `draw` 方法是不同的。多态允许将这两个方法都命名为 `draw`，并且让 `Circle` 和 `Rectangle` 都使用这些方法，仿佛它们都是 `Shapes`(至少在这个例子中是这样)。

**POP** Post Office Protocol(邮件协议)，也称为 POP3。一个从服务器下载 E-mail 的协议。POP 规定从服务器上下载邮件后，便将它从服务器上删除。参见 IMAP。

**端口(Port)** 同 IP 地址一起使用，端口号标识 Internet 上的一个特殊的服务。

**协议(Protocol)** 关于如何构造网络参与者之间发送的数据的约定。HTTP 和 TCP/IP 都是协议的例子。

**协议栈(Protocol Stack)** 一组协议，其中高层协议委托给低层协议。

**quoted-printable** 由 MIME 定义的编码策略，它分别转义每个非 U.S.ASCII 字符。对于主要包含 U.S. ASCII 字符的文本而言，它比 Base64 编码效率更高。

**引号(Quote)** 在 Python 中，字符串由引号中的文本定义。引号可以是单引号(')、双引号(")或三引号(" " "或' ' ')。如果一个字符串以单引号开始，就必须以单引号结束。一个以双引号开始的字符串必须以双引号结束。一个以三引号开始的字符串必须以相同类型的三引号结束(' ' '必须与' ' '匹配，" " "必须与" " "匹配)。单引号和双引号以相同的方式工作。三引号比较特殊，因为它们可以包含多行字符串(包含换行符的字符串)。

**Range** Range 产生一个数值的列表，默认从 0 到参数给定的数，步长为 1。它也可以开始于一个非 0 数，并以不是 1 的步长增加。

**RDBMS** 关系数据库管理系统(Relational Database Management System。见“关系数据库”。

**关系数据库(Relational Database)** 在关系数据库中，数据存储在表中。表是一个二维数据结构。每个表由行组成，行也称为记录。每行由列组成。通常，每个记录保存一个关于项的信息，例如一张音乐 CD、一个人、一张订购单、一辆汽车，等等。

**表示(Representation)** 在 REST 术语中，表示是对资源的描述。当请求一个资源时，得到的是它的一个表示。一个资源可能有多个表示。例如，一个文档资源可以有 HTML、PostScript 和纯文本等不同的表示。

**资源(Resource)** 在 REST 术语中，可以通过 Web 访问和/或操作的对象就是资源。资源可以有多种形式：服务器上的一个文档，数据库中的一行，甚至是一个物理对象(例如从网上商店订购的一个产品)。

**资源标识符(Resource Identifier)** 唯一标识某个资源的字符串。通常情况下，相当于一个 URL。一个资源可以有多个标识符。

**REST** 表示状态转移(REpresentational State Transfer)。万维网的架构。

**REST 符合程度(RESTfulness)** Web 应用程序符合 REST 设计的程度的非正式度量。

**RFC 2822** Internet E-mail 消息的标准格式。要求 E-mail 消息是 U.S. ASCII 格式。

**机器人(Robot)** 在没有人的直接控制下产生 HTTP 请求的脚本。

**RSS** 丰富站点摘要(Rich Site Summary)，或 RDF 站点摘要(RDF Site Summary)。用于联合内容的一种基于 XML 的格式。

**SAX** 用于 XML 的简单 API(Simple API for XML)一个基于流的 XML 解析器。

**作用域(Scope)** 数据和代码的名称；具有不同可见级别的变量名、类名、函数名等。在一个函数或方法内可见的名称或者位于它的作用域内，或者来自于访问方法或函数的操作所在的作用域之上的作用域。

**序列(Sequence)** 序列是一种数据类型。序列可以引用任何类型的对象，但是要求对象中包括一个从 0 开始的顺序数值索引，其包含值的引用。索引可以引用的每个值是通过变量名引用的任意 Python 对象。序列中的元素通过在序列名后面使用方括号解除引用。因此对于名为“seq”的序列，“seq[3]”解除引用第 4 个元素。中括号中的数字是 3 而不是 4，这是因为序列的第一个索引为 0。

**SMTP** 简单邮件传输协议(Simple Mail Transport Protocol)。发送 Internet E-mail 的标

准协议。

**SOAP** 最初代表简单对象访问协议(Simple Object Access Protocol)。用于 Web 服务调用的标准，同 XML-RPC 相似，但是定义更正式。

**套接字(Socket)** IP 网络上的双向连接。套接字允许编程人员像处理文件一样处理网络连接。

**蜘蛛(Spider)** 给定一个起始 Web 页面，沿着链接找到其他 Web 页面以进行操作的机器人。

**SQL** 结构化查询语言(Structured Query Language)，发音为 sequel 或 S-Q-L。用于访问关系数据库的语言。

**SSL** 安全套接字协议层(Secure Socket Layer)。运行于 TCP/IP 和其他一些网络协议(例如 SMTP 或 HTTP)之间的协议，提供端到端的加密。

**栈跟踪(Stack Trace)** 见“调用栈”。

**字符串(String)** 引号(单引号、双引号或一串连在一起的三个单或双引号)包围的任何字母或数值的组合。字符串由字符的多个实例组成(字符是一个在引号中包括单个字母或数值的数据类型)。在 Python 3.1 中，有两种字符串类型：`str` 和 `bytes`。`str` 类型保存文本，而 `bytes` 类型保存数据。如果希望混合两种类型，必须明确地在两者之间进行转换。如果希望将一个字符串转换为字节，可以使用 `str.encode()`；反过来，从字节转换为字符串，可以使用 `bytes.decode()`。

**TCP/IP** 描述一个非常常见的协议栈的术语：TCP 运行于 IP 之上。

**TCP** 传输控制协议(Transport Control Protocol)，保证在 IP 网络的两点之间可靠地、有序地通信。

**元组(Tuple)** 元组是一个序列类型，也是一个迭代器。元组同列表类似，只不过在定义一个列表后，它包含的元素个数和对它的元素的引用将不能改变(然而，如果它引用一个可以修改的对象，例如一个列表或字典，该对象的数据仍然可以改变)。元组由圆括号()创建。当创建一个只包含一个元素的元组时，必须在该元素的后面放上逗号。否则将建立一个字符串。

**UID** 唯一的 ID(Unique ID)。用于各种上下文，表示一个唯一的和随时间稳定的 ID。

**Unicode** Unicode 是一个用于编码字符串的系统，使得即使某些人使用不同的字符编码来读那个字符串，也可以确定字符串中原来的字母(假设某人使用区域设置为俄罗斯的计算机来读一个用希伯来语写的文档。在内部，字符可以被认为是查找表中的数字，对于不同的语言和字符集，字符#100 在各个字符集中可能是不同的)。

**用户代理(User Agent)** Web 浏览器或支持 HTTP 的脚本。

**变量(Variable)** 绑定到名称的数据称为变量。“变量”通常指基本类型和不太复杂的对象。即使整数、浮点数、虚数和字符串在 Python 中都是对象，这一点也成立。这种思考方法是从其他存在差别的语言延续下来的约定。

**Web 应用程序(Web Application)** 通过 HTTP 公开它的接口的应用程序，而不是通过命令行或 GUI 界面。

**Web 服务(Web Service)** 一个为支持 HTTP 的脚本而不是为使用 Web 浏览器的人设计的 Web 应用程序。

**公认端口(Well-Known Port)** IP 端口号在 0~1023 之间的端口称为公认端口。像 Web 服务这样流行的服务一般运行在公认端口上，运行在公认端口上的服务通常以管理员权限运行。

**空白(Whitespace)** 空白指在输入或阅读时看不见的字符。换行符、空格或制表符都是空白字符。Python 关注行的起始处的空白，也知道行尾的换行符，但是在列表或元组内定义和三引号字符串内部的空白除外。

**wiki** 一个 Web 应用程序，允许用户通过一个 Web 接口创建和编辑 web 页面。

**WSDL** Web 服务描述语言(Web Services Description Language)。使用 XML 表示方法调用的一种方式。

**XML** 可扩展标记语言(eXtensible Markup Language)。一个使用自定义词汇创建结构化标记语言的规范。

**XML-RPC** RPC 代表远程过程调用(Remote Procedure Call)。XML-RPC 是 Web 服务调用的标准。它定义了一种方法，可以使用 XML 表示简单数据结构、通过 HTTP 将数据结构作为一个函数调用的参数发送、并将得到的其他数据结构作为返回值。

**XML 模式(XML Schema)** 一个生成文档模型的规范。

**XML 验证(XML Validation)** 检查一个 XML 文档格式正确并符合它的文档模型的过程。

**XML 格式正确性(XML Wellformedness)** 检查一个 XML 文档是否符合 XML 规范的过程。

**xrange** xrange 产生一个 xrange 对象，它是一个行为与列表相似的可迭代对象，但是因为没有创建列表，当需要更大范围的数值时，没有使用额外的内存。

**XSL-FO** 可扩展样式表语言格式化对象(eXtensible Stylesheet Language Formatting Objects)。用于图形显示的标记语言。通常用于产生做最终介绍的文档。

**XSLT** 可扩展样式表语言转换(eXtensible Stylesheet Language for Transformation)，一种用于转换 XML 的编程语言。

