更多IT类电子书，视频教程了请访问： www.guma.pw

包含完整的模块指南和应用Pyth
作为RAD工具的详细介绍

技术参考大全

Python: The Complete Reference

Python

Martin C.Brown 著
康 博 译

包含丰富的Python编程的内容

包括应用程序功能的扩展库

涉及线程，Web发布，和使用Python跨平台开发

清华大学出版社
http://www.tup.tsinghua.edu.cn
麦格劳-希尔教育出版集团
http://www.mheducation.com

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

Python 技术参考大全

包含完整的模块指南和应用Python作为RAD工具的详细介绍

最权威的 Python 参考书

探索使用 Python 编程的各方面内容。资深程序员 Martin C. Brown 讲述了使用模块的基础知识到面向对象的高级应用。您可以学习到如何开发复杂的应用程序，如何创建多媒体软件以及如何开发网站的内容。本书由经验丰富的编程人员编写，是初学编程人员的宝贵资料。本书也涉及到如何去使用 Python 扩展库，包括读取和解析 SGML，HTML 和 XML 文件。

- 掌握 Python 语言编程
- 如何编写模块并嵌入到脚本中
- 使用 Python 扩展库进行信息处理，文件系统操作，数据管理，网络通信，多媒体及其他方面
- 应用 Python 作为 RAD 工具进行快速应用程序开发
- 理解 Perl 和其他语言的程序员的专业介绍
- 使用 SGML，HTML 和 XML 用于高级网络应用
- 使用 TK 接口建立基于 GUI 的应用程序

Martin C. Brown 的编程生涯长达 15 年。目前，他写作的主题十分丰富，包括编程、CD 编写和跨平台集成。

McGraw-Hill
全球智慧中文化

ISBN 7-302-05337-5
9 787302 053378 >
定价：54.00 元

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

TP312PY
B97

Python 技术参考大全

Martin C.Brown 著

康 博 译

A0995062

清 华 大 学 出 版 社

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(京) 新登字 158 号
北京市版权局著作权合同登记号：01-2001-3174

内 容 简 介

本书详细讲述了 Python 语言的各个方面，是一本极具参考价值的 Python 编程手册。在简单介绍了 Python 的基本原理和组成之后，本书给出大量示例，循序渐进、深入浅出地讲述了 Python 的核心内容、应用开发及相关细节。本书由经验丰富的专家编写，作者由浅入深地阐述了 Python 语言。全书共分为六个部分和两个附录，讲述了 Python 语言的基本原理、标准库、快速开发应用程序的方法、Web 开发方法、跨平台开发方法以及其内部机制，在附录中还提供了 Python 的库指南和资源信息。同时，本书说明了使用 Python 作为快速应用程序开发工具的原因和方法，并阐述了如何阅读和编写优秀文档与信息、如何利用 C 扩展和嵌入 Python 的方法。

Martin C.Brown : The Complete Reference Python
EISBN：0-07-212718-X
Copyright© 2001 by McGraw-Hill,Inc.
Authorized translation from the English language edition published by McGraw-Hill,Inc.
All rights reserved. For sale in the People's Republic of China only.
Chinese simplified language edition published by Tsinghua University Press.

本书中文简体字版由清华大学出版社和美国麦格劳-希尔国际公司合作出版。未经出版者书面许可，不得以任何方式复制或抄袭本书的任何部分。

图书在版编目(CIP)数据

Python 技术参考大全/(美)布朗(Martin C.Brown)著；康博公司译。-北京：清华大学出版社，2002

ISBN 7-302-05337-5

Ⅰ.P… Ⅱ.①布… ②康… Ⅲ.PYTHON 语言-程序设计 Ⅳ.TP312

中国版本图书馆 CIP 数据核字(2002)第 013525 号

版权所有，翻印必究。
本书封面贴有 McGraw-Hill 激光防伪标签，无标签者不得销售。

出 版 者：清华大学出版社(北京清华大学学研大厦，邮编 100084)
http://www.tup.tsinghua.edu.cn
责任编辑：于平
印 刷 者：北京密云胶印厂
发 行 者：新华书店总店北京发行所
开　　本：787×1092 1/16　**印张**：30　**字数**：768 千字
版　　次：2002 年 4 月第 1 版　2002 年 4 月第 1 次印刷
书　　号：ISBN 7-302-05337-5/TP・3136
印　　数：0001～4000
定　　价：54.00 元

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

前　言

本书的主要目的是：为想了解 Python 各个方面知识的读者提供全面的指南。实际上这并不像听起来那样容易。写作这类《技术参考大全》书籍，有点像问“字符串有多长”这样的问题。虽然本书可以涵盖许多信息，但总有一些不适合本书的内容，或者没有涵盖读者所希望的所有内容。

这并不表示本书缺少信息。在本书的前面几页包含使用基本 Python 解释器所需的所有参考信息，包括内置数据类型、循环和语句，以及 Python 错误处理系统(也称为异常)。

本书详细讨论了如何使用 Python 标准库中的模块，如何使用 Tk 和进行 Web 编程来利用和开发用户接口。同时，本书说明了使用 Python 作为快速应用程序开发工具的原因和方法，并阐述了如何阅读和编写优秀文档，以及如何利用 C 语言扩展和嵌入 Python。

本书适用对象

本书是一本参考书，作者尽量使本书易于使用和理解。如果您以前从未使用过 Python，本书不会教您如何使用。如果您有一定的 Python 基础，想深入学习，那么应该能使用本书提供的参考材料和编程/样式指南以提高应用 Python 的水平。

如果您不熟悉 Python，但熟悉其他脚本语言，尤其是 Perl，应该能很快学会 Python 语言。虽然本书主题不是从 Perl 到 Python 的迁移，但是包含了这两门语言区别的基本信息(要得到 Perl 编程指导，可查阅作者写的 Perl 方面的书籍)。

如何使用本书

本书的脚本和脚本框架不作修改就能在用户计算机上运行。Python 对 95%的平台兼容，因此在支持 Python 的机器上运行这些脚本应该没有任何问题(要得到 Python 的拷贝，请访问 Python 站点 www.python.org)。

第 1 部分介绍 Python 的基础知识。首先讨论 Python 的构成，然后讨论 Python 与其他语言(如 Perl)的不同点和 Python 的本质。在介绍使用 Python 面向对象方法时，还讨论了基本的 Python 组件的详细信息、创建函数和模块的方法，以及异常处理，并讨论了 Python 的用法。

第 2 部分着重讨论 Python 标准库—— 即作为标准组件随 Python 发行版一起发行的模块。标准库提供许多功能。这些功能涵盖的范围从内置函数到可以存储信息、使用文件、处理文件系统和基本网络功能的扩展功能。

第 3 部分讨论利用 Python 开发应用程序。首先从使用 Python 作为快速应用程序开发的工具开始，接着介绍简化开发过程的工具和资源。这部分的最后一章讨论了分布式应用程序的开发过程。

第 4 部分论述利用 Python 进行 Web 开发的内容。着重讨论创建 Python 的 Web 脚本和 CGI 接口的基本过程。接着讨论在基本过程中用到的一些特定库。在继续介绍 Web 工具和编写 Python 应用程序之前，着重论述用 Python 处理*ML 的技术，如处理 HTML 和 XML。

第 5 部分是使用 Python 跨平台开发的快速指南。Python 隐藏了这个过程的复杂性，但还存在一些使粗心的程序员感到迷惑的陷阱。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第 6 部分深入讨论 Python 语言的内幕。内容包括 Python 解释器内幕概览、用 C 语言编写模块以扩展 Python 库的方法、在 C 应用程序中嵌入 Python 解释器。这部分着重讨论了如何调试 Python 脚本、如何进行优化而得到最佳运行速度、如何为代码编制文档和说明以使代码更具可读性。

作者联系方式

欢迎对本书提出评论和建议。尤其感激对各种主题(特别是 Python)的指导和建议。作者会尽最大努力处理，但新事物总是层出不穷。与作者联系的最佳方式是电子邮件。可使用 books@mcwords.com。另外可访问作者的站点 http://www.mcwords.com，该站点的资源包括本书的脚本和内容、更新信息、后续文章和勘误等。本书主页为 http://www.mcwords.com/projects/book/pytcr/。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

目　　录

第 1 部分　Python 语言的基础知识

第 1 章　Python 简介 …… 1
1.1　什么是 Python …… 1
1.1.1　Python 是免费的 …… 1
1.1.2　Python 是可移植的 …… 2
1.1.3　Python 的强大功能 …… 2
1.1.4　Python 是可扩展的 …… 2
1.1.5　Python 的简单性 …… 3
1.2　Python 的适用范围 …… 3
1.2.1　数学 …… 3
1.2.2　文本处理 …… 3
1.2.3　快速应用程序开发 …… 3
1.2.4　跨平台开发 …… 4
1.2.5　系统实用工具 …… 4
1.2.6　互联网编程 …… 4
1.2.7　数据库编程 …… 4
1.2.8　其他 …… 4
1.3　Python 不适用的范围 …… 5
1.4　Python 用户 …… 5
1.5　Python 的历史 …… 5
1.5.1　Python 1.5.2 …… 6
1.5.2　Python 1.6 (2000 年 9 月) …… 6
1.5.3　Python 2.0 (2000 年 9 月 5 日) …… 6
1.5.4　Python 2.0(2000 年 10 月 28 日) …… 6
1.5.5　Python 3000 …… 6
1.6　其他相似的语言 …… 6
1.6.1　Perl …… 7
1.6.2　Java …… 8
1.6.3　JavaScript …… 8
1.6.4　Tcl …… 9
1.6.5　Rebol …… 9
1.6.6　Visual Basic …… 9

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

1.6.7 Awk/Gawk ····· 9
1.6.8 C/C++ ····· 10
1.6.9 Unix/DOS Shells ····· 10

第 2 章 Python 基本原理 ····· 12
2.1 执行 Python 程序 ····· 12
2.1.1 交互式 ····· 12
2.1.2 从文件执行 ····· 14
2.1.3 其他方法 ····· 19
2.2 脚本、程序或模块 ····· 20

第 3 章 Python 程序的组成 ····· 21
3.1 内置对象类型 ····· 21
3.1.1 Python 对象和其他语言 ····· 22
3.1.2 基本运算符 ····· 23
3.1.3 数字 ····· 24
3.1.4 字符串 ····· 28
3.1.5 列表 ····· 34
3.1.6 Tuples ····· 37
3.1.7 序列的使用 ····· 38
3.1.8 字典 ····· 39
3.1.9 文件 ····· 42
3.1.10 对象存储 ····· 42
3.1.11 类型转换 ····· 44
3.1.12 类型比较 ····· 45
3.2 语句 ····· 47
3.2.1 语句格式 ····· 47
3.2.2 注释 ····· 48
3.2.3 赋值 ····· 48
3.2.4 打印 ····· 50
3.2.5 控制语句 ····· 51
3.2.6 普通陷阱 ····· 55

第 4 章 函数 ····· 57
4.1 函数定义与执行 ····· 57
4.2 作用域 ····· 59
4.2.1 创建全局对象 ····· 59
4.2.2 LGB 规则 ····· 60
4.2.3 陷阱作用域 ····· 61

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

4.3 参数 …… 62
4.3.1 参数是对象 …… 63
4.3.2 关键字的参数调用 …… 64
4.3.3 默认参数 …… 64
4.3.4 参数 Tuples …… 65
4.3.5 参数字典 …… 65
4.3.6 函数规则 …… 66
4.4 返回值 …… 67
4.5 高级函数调用 …… 67
4.5.1 apply 语句 …… 67
4.5.2 map 语句 …… 68
4.5.3 间接函数调用 …… 69
4.5.4 匿名函数 …… 69

第 5 章 模块 …… 71
5.1 输入模块 …… 71
5.1.1 输入完整模块 …… 71
5.1.2 用别名输入模块 …… 72
5.1.3 输入特定模块实体 …… 72
5.1.4 重新加载(reloading)模块 …… 72
5.1.5 模块搜索路径 …… 73
5.1.6 模块装载与编译 …… 74
5.2 模块输入的技巧 …… 74
5.2.1 在脚本中使用 import …… 74
5.2.2 追踪 import 语句 …… 75
5.2.3 标识模块或脚本 …… 75
5.3 包 …… 76
5.4 创建模块 …… 78

第 6 章 面向对象 …… 79
6.1 创建类 …… 79
6.1.1 类的方法 …… 80
6.1.2 类的继承 …… 86

第 7 章 异常和错误的捕获 …… 88
7.1 异常的概念 …… 89
7.2 引发异常的结果 …… 90
7.3 异常的处理 …… 92
7.3.1 try… except… else …… 92

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

7.3.2 try… finally …… 95
7.3.3 异常嵌套 …… 96
7.3.4 引发异常 …… 97
7.3.5 assert 语句 …… 97
7.4 内置异常 …… 97
7.4.1 Exception …… 97
7.4.2 StandardError …… 98
7.4.3 ArithmeticError …… 98
7.4.4 AssertionError …… 98
7.4.5 AttributeError …… 98
7.4.6 EnvironmentError …… 99
7.4.7 EOFError …… 99
7.4.8 FloatingPointError …… 99
7.4.9 ImportError …… 99
7.4.10 IndexError …… 99
7.4.11 IOError …… 99
7.4.12 KeyError …… 99
7.4.13 KeyboardInterrupt …… 100
7.4.14 LookupError …… 100
7.4.15 MemoryError …… 100
7.4.16 NameError …… 100
7.4.17 NotImplementedError …… 100
7.4.18 OSError …… 100
7.4.19 OverflowError …… 100
7.4.20 RuntimeError …… 100
7.4.21 SyntaxError …… 100
7.4.22 SystemError …… 101
7.4.23 SystemExit …… 101
7.4.24 TypeError …… 101
7.4.25 UnboundLocalError …… 101
7.4.26 UnicodeError …… 101
7.4.27 ValueError …… 102
7.4.28 WindowsError …… 102
7.4.29 ZeroDivisionError …… 102
7.5 自定义异常 …… 102

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第 2 部分　应用 Python 库

第 8 章　Python 内置函数 …… 104

8.1　_import_(name[,globals[,locals[,formlist]]]) …… 104

8.2　abs(x) …… 105

8.3　apply(function,args[,keywords]) …… 105

8.4　buffer(object[,offset[,size]]) …… 105

8.5　callable(object) …… 106

8.6　chr(i) …… 106

8.7　cmp(x,y) …… 106

8.8　coerce(x,y) …… 106

8.9　compile(string,filename,kind) …… 107

8.10　complex(real[,imag]) …… 107

8.11　delattr(object, name) …… 107

8.12　dir([object]) …… 107

8.13　divmod(a, b) …… 108

8.14　eval(expression[,globals[,locals]]) …… 108

8.15　execfile(file [, globals [, locals]]) …… 109

8.16　filter(function, list) …… 109

8.17　float(x) …… 109

8.18　getattr(object, name [, default]) …… 110

8.19　globals() …… 110

8.20　hasattr(object, name) …… 110

8.21　hash(object) …… 110

8.22　hex(x) …… 110

8.23　id(object) …… 110

8.24　input([prompt]) …… 111

8.25　int(x [, radix]) …… 111

8.26　intern(string) …… 111

8.27　isinstance(object, class) …… 111

8.28　issubclass(class1, class2) …… 111

8.29　len(s) …… 111

8.30　list(sequence) …… 112

8.31　locals() …… 112

8.32　long(x) …… 112

8.33　map(function, list, ...) …… 112

8.34　max(s [, args...]) …… 112

8.35　min(s [, args...]) …… 113

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

8.36 oct(x) …… 113
8.37 open(filename [, mode [, bufsize]]) …… 113
8.38 ord(c) …… 114
8.39 pow(x, y [, z]) …… 114
8.40 range([start,] stop [, step]) …… 115
8.41 raw_input([prompt]) …… 115
8.42 reduce(function, sequence [, initializer]) …… 116
8.43 reload(module) …… 116
8.44 repr(object) …… 116
8.45 round(x[, n]) …… 117
8.46 setattr(object, name, value) …… 117
8.47 slice([start,] stop [, step]) …… 117
8.48 str(object) …… 117
8.49 tuple(sequence) …… 118
8.50 type(object) …… 118
8.51 unichr(i) …… 118
8.52 unicode(string [, encoding [, errors]]) …… 118
8.53 vars([object]) …… 119
8.54 xrange([start,] stop [, step]) …… 119
8.55 zip(seq1, ...) …… 119
8.56 执行任意语句 …… 119
8.57 exec 语句 …… 119
8.58 execfile()函数 …… 120
8.59 eval()函数 …… 120

第 9 章　与操作系统的接口 …… 121
9.1 使用系统(sys 模块) …… 121
9.1.1 获取命令行参数 …… 121
9.1.2 标准文件句柄 …… 122
9.1.3 终止执行 …… 123
9.1.4 跟踪终止 …… 124
9.1.5 解释器信息 …… 124
9.1.6 模块搜索路径 …… 125
9.2 使用操作系统(os 模块) …… 126
9.2.1 操作环境变量 …… 126
9.2.2 行终止 …… 127
9.2.3 进程环境 …… 127
9.2.4 进程执行与管理 …… 128

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

9.2.5 信号 …… 134
9.2.6 用户/组信息 …… 136
9.3 多线程 …… 137
9.3.1 多任务工作原理 …… 138
9.3.2 从多任务到多线程 …… 139
9.3.3 线程与多进程的比较 …… 140
9.3.4 线程和 select()的比较 …… 140
9.3.5 线程和 Python …… 141
9.3.6 基本线程 …… 141
9.3.7 高级线程 …… 145
9.3.8 队列 …… 149

第 10 章 信息处理 …… 151
10.1 操作数字 …… 151
10.1.1 math …… 151
10.1.2 cmath …… 153
10.1.3 随机数字 …… 153
10.2 文本操作 …… 156
10.2.1 基本字符串操作 …… 156
10.2.2 正则表达式 …… 160
10.3 时间 …… 168
10.3.1 抽取时间值 …… 169
10.3.2 时间格式化 …… 170
10.3.3 创建新纪元时间值 …… 171
10.3.4 时间值的比较 …… 172
10.3.5 暂停进程 …… 173
10.4 数据类型与运算符 …… 173
10.4.1 类型验证 …… 173
10.4.2 运算符 …… 175
10.4.3 数组建造 …… 176
10.4.4 二进制结构 …… 178
10.5 Unicode 字符串 …… 179
10.5.1 创建 Unicode 字符串 …… 180
10.5.2 转换 Unicode …… 181
10.5.3 编码为 Unicode 格式 …… 182
10.5.4 解码为 Unicode 格式 …… 183
10.5.5 编写自己的编码解码器 …… 184
10.5.6 访问 Unicode 数据库 …… 187

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第 11 章 文件处理 …… 188
11.1 文件处理 …… 188
11.1.1 读文件 …… 189
11.1.2 写入文件 …… 192
11.1.3 改变位置 …… 193
11.2 控制文件 I/O …… 194
11.2.1 文件控制 …… 194
11.2.2 I/O 控制 …… 195
11.2.3 文件锁定 …… 195
11.3 获取文件列表 …… 196
11.4 基本文件/目录管理 …… 197
11.5 访问和所有权 …… 198
11.5.1 检查访问 …… 198
11.5.2 获取文件信息 …… 199
11.5.3 设置文件权限 …… 200
11.6 操作文件路径 …… 201

第 12 章 数据管理和存储 …… 203
12.1 管理内在结构 …… 203
12.1.1 序列排序 …… 203
12.1.2 复制对象 …… 205
12.2 对象持续期 …… 206
12.2.1 对象存储 …… 206
12.2.2 DBM 数据库 …… 208
12.2.3 商业数据库 …… 211

第 13 章 网络通信 …… 213
13.1 Networking 101 …… 213
13.1.1 逻辑连接类型 …… 213
13.1.2 网络名称和数字 …… 214
13.1.3 网络端口 …… 214
13.1.4 网络通信 …… 215
13.1.5 BSD Socket 接口 …… 215
13.2 获取网络信息 …… 216
13.3 基本套接字函数 …… 217
13.4 创建网络服务器 …… 222
13.4.1 利用 SocketServer 模块 …… 223
13.4.2 运行 HTTP 服务 …… 225
13.5 客户机模块 …… 226

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

13.5.1 使用 SMTP ····· 226
13.5.2 使用 FTP ····· 227
13.5.3 使用 HTTP ····· 227
13.5.4 使用 IMAP ····· 229
13.6 处理因特网数据 ····· 232
13.6.1 base64 ····· 232
13.6.2 binascii ····· 232
13.6.3 binhex ····· 233
13.6.4 mailcap ····· 233
13.6.5 mimetools ····· 234
13.6.6 mimetypes ····· 235
13.6.7 MimeWriter ····· 235
13.6.8 multifile ····· 236
13.6.9 quopri ····· 237
13.6.10 rfc822 ····· 237
13.6.11 uu ····· 239
13.6.12 xdrlib ····· 240

第 14 章 多媒体中使用 Python ····· 241
14.1 音频模块 ····· 241
14.1.1 sndhdr ····· 242
14.1.2 aifc ····· 243
14.1.3 audioop ····· 246
14.1.4 chunk ····· 248
14.1.5 sunau ····· 249
14.1.6 wave ····· 251
14.2 图形模块 ····· 251
14.2.1 imghdr ····· 252
14.2.2 colorsys ····· 252
14.2.3 imageop ····· 253
14.2.4 rgbimg ····· 253

第 15 章 用 Tk 创建接口 ····· 255
15.1 Unix 下安装 Python/Tk ····· 256
15.2 Windows 下安装 Python/Tk ····· 257
15.3 MasOS 下安装 Python/Tk ····· 257
15.4 Tk 简介 ····· 257
15.4.1 窗口 ····· 259
15.4.2 窗口小部件(Widgets) ····· 259

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

15.4.3 嵌套 …… 259
15.4.4 几何管理 …… 260
15.4.5 回调 …… 260
15.4.6 事件循环 …… 260
15.5 使用窗口小部件 …… 263
15.5.1 核心部件 …… 263
15.5.2 普通部件属性 …… 264
15.5.3 标签 …… 267
15.5.4 按钮 …… 267
15.5.5 单选按钮 …… 268
15.5.6 复选按钮 …… 269
15.5.7 文本框 …… 270
15.5.8 输入框 …… 272
15.5.9 列表框 …… 273
15.5.10 菜单 …… 274
15.5.11 帧 …… 277
15.5.12 滚动条 …… 277
15.5.13 刻度条 …… 278
15.6 控制窗口几何 …… 281
15.6.1 包容器 …… 282
15.6.2 栅格 …… 283
15.6.3 定位器 …… 283

第 3 部分　应用程序开发

第 16 章 Python 作为 RAD 工具使用 …… 286
16.1 何为 RAD …… 286
16.1.1 RAD 需求 …… 287
16.1.2 可选的 RAD 解决方案 …… 288
16.2 为何选用 Python …… 289
16.2.1 开发生命周期 …… 289
16.2.2 高层编程 …… 291
16.2.3 方便的跨平台兼容性 …… 291
16.2.4 Python 和 OOP …… 292
16.2.5 Python 的嵌入扩展性 …… 292
16.2.6 Python: 在 Steroids 层次的 RAD …… 292

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第 17 章　使用 Python 开发应用程序 …… 293
17.1　集成开发环境 …… 293
17.1.1　IDLE …… 294
17.1.2　PythonWin …… 294
17.1.3　MacPython IDE …… 295
17.1.4　Komodo …… 296
17.1.5　Visual Studio/VisualPython …… 297
17.1.6　BlackAdder …… 298
17.1.7　WingIDE …… 298
17.2　Python 标准库 …… 299
17.2.1　演示、范例和样本 …… 299
17.2.2　标准模块 …… 300
17.2.3　配置标准扩展 …… 301
17.3　Vaults of Parnassus …… 303
17.4　Zope 与 Jython …… 303

第 18 章　发布 Python 模块 …… 304
18.1　使用 distutils …… 305
18.1.1　可支持的模块 …… 305
18.1.2　编写 setup.py …… 305
18.2　未来的特征 …… 306

第 4 部分　Web 开发

第 19 章　Web 开发基础 …… 307
19.1　编写 HTML …… 307
19.2　统一资源定位符(URL，Uniform Resource Locator) …… 309
19.3　使用 CGI 的动态网站 …… 310
19.3.1　Web 环境 …… 311
19.3.2　提取表单数据 …… 314
19.3.3　发送信息 …… 316
19.3.4　转义特殊字符 …… 321
19.3.5　调试 …… 322
19.4　Cookie …… 322
19.5　安全性 …… 324

第 20 章　标准标记语言处理 …… 326
20.1　处理 SGML …… 327
20.2　处理 HTML …… 327

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

20.3 处理 XML …… 331
20.3.1 XML 分析器 …… 333
20.3.2 使用 Expat …… 333
20.3.3 使用 DOM(minidom) …… 336

第 21 章 Python 的其他 Web 工具 …… 341
21.1 Zope(Z-对象发布环境) …… 341
21.1.1 Zope 系统 …… 342
21.1.2 Zope ORB 的工作方式 …… 342
21.1.3 Zope 的特性 …… 343
21.2 Jython …… 344
21.2.1 Jython 的工作原理 …… 344
21.2.2 Jython 的局限 …… 345
21.3 Python.NET …… 346
21.4 Python 服务器页面(Python Server Page) …… 346
21.5 Python 与 ActiveScript …… 347
21.6 Mailman …… 347
21.7 Grail …… 348
21.8 Apache 与 Python …… 348
21.9 SocketServer 与 BaseHTTPServer …… 349
21.10 Medusa …… 349

第 5 部分　跨平台开发

第 22 章 跨平台开发的路径 …… 350
22.1 基本平台支持 …… 350
22.2 运行环境 …… 351
22.3 行终止 …… 356
22.4 字符集 …… 356
22.5 文件和路径名 …… 357
22.6 数据不一致性 …… 357
22.7 性能和资源 …… 358

第 6 部分　深入 Python

第 23 章 Python 体系结构 …… 359
23.1 名称空间、代码块和帧 …… 360
23.1.1 代码块 …… 360

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

23.1.2 帧 …… 361
23.1.3 名称空间 …… 361
23.1.4 回跟踪(tracebacks) …… 362
23.1.5 综合 …… 362
23.2 内置类型 …… 363
23.2.1 可调用对象类型 …… 364
23.2.2 模块 …… 365
23.2.3 类 …… 366
23.2.4 类实例 …… 366
23.2.5 内部类型 …… 367
23.3 字节码 …… 369
23.3.1 Python 字节码 …… 369
23.3.2 字节码的分解 …… 370
23.3.3 字节码指令(Opcode) …… 372

第 24 章 调试和调整 …… 379
24.1 调试简介 …… 379
24.1.1 bug 类型 …… 379
24.1.2 基本调试原则 …… 381
24.1.3 预防 bug …… 382
24.2 调试技术 …… 385
24.2.1 利用 print …… 385
24.2.2 保存日志 …… 389
24.2.3 交互使用 Python …… 389
24.2.4 使用 Python 调试器 …… 390
24.3 优化 Python 应用程序 …… 395
24.3.1 手工优化 …… 396
24.3.2 Python 配置器 …… 398

第 25 章 文档编制和文档 …… 404
25.1 注释 …… 404
25.1.1 写注释 …… 405
25.1.2 编写好的注释 …… 405
25.2 嵌入文档字符串 …… 406
25.3 把嵌入字符串翻译为文档 …… 408
25.3.1 pydoc 工具 …… 408
25.3.2 结构化的文本格式化规则 …… 411

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第 26 章　Python 扩展 …… 413
26.1　基本接口 …… 413
26.1.1　编写包装器 …… 414
26.1.2　编译扩展 …… 417
26.1.3　测试结果 …… 418
26.2　数据转换 …… 419
26.3　引用计数管理 …… 420
26.3.1　引用计数 …… 421
26.3.2　引用类型 …… 421
26.3.3　更新引用计数 …… 421
26.4　异常 …… 422
26.5　低层对象访问 …… 426
26.6　下一章的内容 …… 432

第 27 章　Python 嵌入 …… 434
27.1　嵌入原则 …… 434
27.1.1　用 Python 嵌入 API …… 434
27.1.2　编译与链接 …… 437
27.2　用 Python 嵌入类型 …… 440
27.2.1　执行 Python 字符串 …… 440
27.2.2　用 Python 对象工作 …… 441
27.2.3　利用 Python 类 …… 443
27.3　下一步的工作 …… 446

附录 A　Python 库指南 …… 447

附录 B　Python 资源 …… 456
B.1　Web 资源 …… 456
B.2　邮件、新闻组和发邮件列表的资源 …… 458
B.3　在线文档资料 …… 459
B.4　资源提示 …… 460

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第 1 部分　Python 语言的基础知识

第1章　Python简介

在深入讨论 Python 语言之前，先花点时间了解 Python 语言遵循的原则以及它的适用范围。同时了解哪些人会使用 Python 以及 Python 与其他编程语言的不同。

1.1　什么是 Python

Python 是一种面向对象的解释性语言。它是一种高级编程语言，也就是说它尽可能将用户与底层操作系统隔离。然而，与其他解释性语言不同的是 Python 支持对操作系统的底层访问。因而，通常将 Python 划分在处于 Visual Basic 或 Perl 到系统级的 C 语言之间的位置。

虽然像 Perl ，Tcl 和其他一些语言一样，Python 被认为是一种解释性语言，但是它有编译的过程，用编码器将原始 Python 脚本翻译成一系列字节码(bytecode)字节码在 Python 虚拟机(Python Virtual Machine)上运行。编码器和字节码的使用提高了程序的性能，使 Python 比纯解释性语言(如 BASIC)更快，但是与 C 和 Pascal 这类编译语言比起来还是较慢。然而，与许多其他语言不同的是，模块的字节代码可以保存，在需要使用时，不需要重新编译就能运行，因此通过排除编译阶段可提高 Python 的性能。注意，产生的字节码与平台和操作系统完全独立，类似于由 Java 产生的字节码。

在 MacOS ，Windows (95/98/NT) 和 Unix 平台上，Python 还可以支持快速应用程序开发。有连接 Tk 接口库的模块提供给 Python，并且在任意一个平台上编写的应用程序可不做任何修正而在所有三个平台上运行。除了这些核心平台外，Python 也可以运行在 MS-DOS，Amiga，BeOS,OS/2,VMS,QNX 和许多其他操作系统上。甚至可以在 Psion organizer 上运行 Python!

在深入介绍 Python 前，应该先解释名字的由来。Python 源于喜剧团体 Monty Python，该团体因许多天才而闻名，如 Eric Idle，John Cleese ，Terry Jones，Terry Gilliam，Michael Palin 和 Graham Chapman。

1.1.1　Python 是免费的

虽然这不是独一无二的——许多编程语言可免费获取——但是这意味着人们可以不用购买任何软件，而且不用担心版权问题就可以编写、发布 Python 程序。如果想深入了解 Python 语言工作原理，甚至可以下载该软件的源代码。

免费通常意味着缺少甚至没有技术支持，Python 却不同。大量的 Python 程序员和 Python 开发人员都乐于帮助学习 Python 的新手。还有许多商业公司和个人在人们需要并愿意付款时，

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

为他们提供定制程序开发和高级技术支持。

1.1.2 Python 是可移植的

许多操作系统平台都支持 Python。Python 对 Windows 和 MacOS 提供编译好的格式，还包含 Tk 扩展，因此人们可以开发用户接口。在 Unix 和其他所有平台上，都可获得 Python 源代码自行编译。也可访问提供预编译成二进制格式的 Python 站点(参见附录 A)。在任何情况下，Python 都有无形的兼容性。在 Unix 平台编写的脚本，95%的情况下可不做修改在 Mac 或 PC 机上运行。因为提供了 Tk 支持，不做较大修改就可在三种平台下运行相同的基于 GUI(图形设备接口)的应用程序，并拥有一致的用户接口。

另外对于本地跨平台兼容能力，Python 也支持使用一些本地平台的辅助设备，减少与其他语言和环境的移植过程和连接间隙。例如，SunOS/Solaris 工具包括对 Sun 音频设备的驱动器，SGI 版提供了一种工具，实现与 SGI 工作站上的声频和视频设备(包括 OpenGL)的接口，Windows Python 解释器提供了工具箱，实现与 VC++库及 Windows 声频驱动设备的接口。甚至可以实现与组件对象模型(Component Object Model,COM)对象的通信。

1.1.3 Python 的强大功能

几乎没有 Python 办不到的事情。Python 的核心很小，但它提供足够的基本构建块，允许程序员设计大多数的应用程序。另外，可使用 C、C++，甚至特定环境下可以用 Java 对 Python 语言进行扩展，因此可开发任何类型的程序。Python 解释器提供大量的各种附加的模块构成的库来扩展 Pythno 程序设计的功能，如网络通信、文本处理(Python 2.0 包括广泛的 XML 支持)和正则表达式匹配。

虽然 Python 的主要目标是向程序员屏蔽底层复杂性，但它也支持一些必要挂钩(hook)、扩展和函数，以允许对操作系统的特定区域进行底层访问。通过提供高层和底层功能的支持，Python 和 C 工作在同一层，也可和 Visual Basic 工作在同一层，或者工作在两者之间的任何层。甚至可以将 Python 嵌入到应用程序用作宏或应用程序扩展，就像 Visual Basic 在 Microsoft Office 中被用作宏语言。

1.1.4 Python 是可扩展的

因为 Python 是用 C 编写的(有些扩展用 C++编写)，而且可以访问其源代码，因此可以对 Python 进行写扩展。Python 提供的许多标准模块支持 C 或 C++接口。包括诸如网络和 DBM 数据库访问的基本工具以及类似 Tk 的高级工具包。

另外，Python 可嵌入到 C 或 C++应用程序中，因此可用 Python 语言为应用程序提供脚本接口。由于支持跨语言开发，可用 Python 设计和概念化应用程序，并逐步移植到 C。使用前不必用 C 重写应用程序；Python 和 C 可以一起工作。

最后，Jython 是用 Java 编写的完整的 Python 解释器。这意味着可以编写连接 Java 对象的 Python 程序，或编写使用 Python 对象的 Java 应用程序。更妙的是，由于解释器完全用 Java 编写，因此可以在支持 Java 的任何平台上部署 Python 应用程序，甚至 Web 浏览器也可直接执行 Python 脚本。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

1.2.4 跨平台开发

现在人们知道Python支持多种平台。如果应用程序通过网络部署且用于多种不同的平台，可以采用Python开发应用程序。如果确定开发的系统未来将应用于跨平台操作，也可采用Python。许多公司开始使用特定平台，后来因性能关系转向别的平台。采用Python后可不重写软件而在平台间移植。

当然，提供面向终端用户的软件时，也可以用Python作为开发语言。Python可直接进行开发，而不必开发三套独立的应用程序，分别用于特定操作系统编译处理、测试系统和接口，以节省时间和资金。

1.2.5 系统实用工具

虽然Python的目标是隐藏操作系统的底层部分，但是如果需要访问系统最底层，也可使用工具和扩展。和操作系统一样，Python可访问同样的一组函数，因此可使用它复制和扩展操作系统的功能，同时保持Python支持的兼容性和接口。

1.2.6 互联网编程

Python提供一套标准模块支持通过网络套接字(Sockets)在基本层和协议层间进行通信。例如，要从POP服务器读取电子邮件，Python提供相应的库模块让用户使用。另外，Python也支持XML，HTML和CGI库，可分析用户输入并通过Web服务器产生高质量格式的输出。

实际上，Python的高层模块支持和RAD能力的组合使之成为庞大、快速的开发工具包。多数Internet模块使用简单的对象类和不同方法就可直接与Internet服务器通信。程序员花两个下午的时间就可以用Python编写新闻阅读器。在作者的家用网络中，有一个到IMAP服务器的基于Web的电子邮件接口，是作者用Python在一小时内编写而成的。

人们可以为Apache，Unix和Windows Web服务器编译一个嵌入了Python解释器的模块。这时要执行Python脚本，不必每次加载解释器，这样可从CGI脚本获取最好的性能。

1.2.7 数据库编程

有许多扩展模块提供到普通数据库系统的接口，从Oracle到Informix和免费系统如mSQL和mySQL。Python内有一个称为Gadfly的工具包，该工具包提供完全的SQL环境——无需任何扩展模块或扩展。因为Python有强大的文本和数据处理能力，可用它编写数据库间的接口或作为更好的汇总、报表工具，以替代数据库系统自带的接口。Python支持多种操作系统，可对任何数据库使用相同的接口。甚至可以用Tk开发前端，再移植到提供Python支持的任何平台——这样，可得到快速跨平台、独立于数据库的查询工具。

1.2.8 其他

Python可完成任何任务，没有限制。通过支持一小套核心函数、数据类型和功能，Python提供了优秀的开发基础。可用C和C++进行扩展功能，因此能得到无限可能和无约束的扩展。可以用结构化和可管理模式完成人们想完成的任务。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

1.3 Python 不适用的范围

很难列举出 Python 不能解决的问题的列表。Python 的扩展模块提供大部分功能，这说明很容易向 Python 添加功能。如果 Python 不能完成某项任务，也很容易用 C 或 C++编写扩展以完成任务。

有些人批评 Python，不是因为 Python 不能完成某些任务，而是因为这些人不懂得如何完成特定任务。经常有人抱怨 Python 明显缺乏对正则表达式的支持。实际上有两个模块(re 和老一点的 regex)可以处理正则表达式；regex 甚至支持和 Perl 的相同语法。正则表达式处理功能没有内置到 Python。虽然经过近几年大力的优化，但是处理速度可能还是不如 Perl。

相对于其他语言如 Perl，Rebol 或 Java，Python 的优点是其核心非常小。这减少了其执行时间——每次运行脚本时加载的代码少——有助于学习语言的其他部分，提高了 Python 的灵活性。

只要熟悉了 Python 支持的基本编程风格，人们就会发现无需其他帮助，他们就有很强的编程能力。而且可以随时阅读他们编写的代码。

1.4 Python 用户

许多人用 Python 解决各种问题。其中的多数问题不为人知或者并未公开，因为相关公司不会泄漏这类信息。但也有一些大公司在商业环境使用 Python，并乐于公开和褒扬这个事实。

- Red Hat(www.redhat.com)：发行备受欢迎的 Red Hat Linux。联合使用 Python 和 Tk 为配置和管理 Linux 操作系统提供可视化接口。配置系统提供对 Linux 操作系统的各种特性的完全控制，并根据所作选择自动更新配置文件。
- Infoseek(www.infoseek.com)：公开宣布搜索引擎的某些部分使用 Python 编写。也用 Python 定制 Infoseek 软件，该软件可从该公司的站点下载并用于终端用户。
- NASA(www.nasa.gov)：在许多不同区域使用 Python。Python 最重要的应用为：安排任务控制中心的规划任务的系统的某些部件。在其他应用中，由于 Python 强大的数字处理能力，使之成为计算太空对象的位置和绘制人造卫星轨迹的理想工具。
- Industrial Light and Magic(www.ilm.com)：因制作电影特效而闻名，如 Star Wars、The Abyss、Star Trek 和 Indiana Jones。他们用 Python 制作了具有商业水准的动画。事实上，如果访问他们的 Web 站点，可以发现有许多空缺的 Python 程序员职位。

1.5 Python 的历史

Python 的历史比多数人知道的要长。在 2000 年，Python 开发小组经历了多次重组。Python 的设计者和初期开发者 Guido van Rossum 和 Python 小组的其他成员(包括 Tim Peters、Barry Warsaw、Jeremy Hylton 和 Fred Drake)从 CNRI(Certre for National Research Initiatives 国家研究创新中心)转到 BeOpen，最终到 Digital Creations。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

1.5.1 Python 1.5.2

直到2000年9月5日，Python在获得CNRI公共许可证后才开发并发布。Python 1.5.2包含大家今天非常熟悉的大部分功能。最终的1.5.2发行版于1999年4月13日发布。

1.5.2 Python 1.6 (2000年9月)

2000年9月，在Guido离开CNRI后，Python 1.6的两个版本发行了。第一版来自于CNRI，这也是CNRI发行的最后一个Python语言的官方发行版。Python的beta版Python 1.6b1已经开发、测试了一段时间，因此Guido van Rossum和BeOpen开发队伍发行Python 1.6b1就不足为奇了。

1.6版有些小的改进，包括对象列表工作方式的改变、套接字和字符串到数字转化工具的改进。

1.5.3 Python 2.0 (2000年9月5日)

在CNRI发布通告的24小时内，Guido和开发小组的其他成员就发布了Python的2.0b1版(beta版)以对抗CNRI的1.6版。2000年5月5日，Python小组已经完全脱离CNRI、并从CNRI转到BeOpen(www.beopen.com)继续在开放性源代码协议下开发Python。

2.0版对1.6版进行了一些重要更新，包括新运算符、新列表语法和更好的模块输入法。2.0版还包含了一年多来对标准Python库所做的最重要的更新，修正了许多bug、增加了几项新功能，并重写了XML工具套件。Python 2.0最终版于2000年10月16日发布。

除了对Python语言的改进外，加入BeOpen也使开发小组在其他方面也得到改进，如将Python源代码转移到SourceForge(www.sourceforge.net)、完成了从基于Java的Python解释器、JPython到新的Jython混合体的转变。

1.5.4 Python 2.0(2000年10月28日)

2000年10月28日，Guido宣布Python小组的新闻组和Python的主Web站点又改变了，这次是转到了Digital Creations。Digital Creations是Zope(Z-Objects Publishing Environment，Z对象出版环境，最负盛名的Python项目)的开发者。

1.5.5 Python 3000

在BeOpen发布Python 2.0后不久，Python 3000的开发计划就已展开，并定于2002年发布。希望在Digital Creations的大力支持下，Python小组不必再迁移了。

Python 3000会对语言进行很大更新，如同Perl 6.0对Perl 5.6的更新。Python 3000也可能不是Python的最终版本名；而仅是新版本的代号。

1.6 其他相似的语言

现在可获得的脚本语言不断增多。虽然很多人听说过Perl、VBScript和BASIC，和一些有

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

用的脚本语言可解决许多问题。但是 Python 适合哪些领域？与其他语言比较起来如何呢？

如果考虑编程语言的使用环境的话，这个问题将更加复杂。Perl 的文本处理能力极强，接下来是 Python、Awk，但应选择哪种语言呢？本书作者的选择是：如果仅仅处理文本文件，用 Perl；要转化、翻译或封装数据到新的数据存储格式时用 Python；需要在 shell 中过滤文本时用 Awk。

1.6.1　Perl

Perl 是目前和 Python 最相似的语言。虽然他们的目标不同，语法也不一样，但它们都是顶尖的解释性脚本语言。

Perl 由 Larry Wall 开发，最初用作总结和汇报文本的处理工具。Perl 表示 Practical Extraction and Report Language(实用提取和报表语言)。与 Python 不同的是，Perl 来自于 Unix 环境，并采用了许多其他语言的原理和语法。可以把其他高级特性，如正则表达式组合到 Unix shell，并可以绑定灵活而功能强大的变量。

但 Perl 也有一些缺陷。虽然面向对象(OO)功能易于使用，但是它是在完成语言开发工作后加入的。虽然这样并不会引起任何问题，但面向对象成为附加功能而不是一个集成的功能。紧密联系的 OO 元素的缺乏使开发者可选择是否编写 OO 代码。不幸的是，即使在开发简单项目时也需要处理对象和非对象实体，这增加了编程的困难。

大多数 Perl 和 Python 提供的功能是难以比较的。他们主要的差异在于其基本特性。Perl 从设计到至今还主要用于系统文本处理和二进制数据处理，使得 Perl 的许多应用都很闻名，包括用于 Internet 中处理 CGI(Common Gateway Interface 通用网关接口)脚本。其他优点有：Perl 具有强大的系统级支持，使之成为 Unix 下复制和提高许多核心系统函数的理想工具。但 Perl 的根本应用领域还是文本处理。

另一方面，Python 是作为一种应用程序编程语言而开发的。用 Python 可开发成熟的、跨平台的应用程序，范围从实用工具到完整的应用程序开发集成环境。

谈到性能时，可能需要在它们之间进行选择。Perl 和 Python 使用相同的编译系统，即将源码编译为字节码，并在虚拟机上运行。但比较语法及其他语言功能时，毫无疑问存在差异。Python 有难以置信的灵活的数据处理能力；因为所有数据存储容器均为对象、并用相同方法处理多数信息，使 Python 的性能大为提高。

Perl 在处理内置数据类型时有一定的局限性。它使用引用机制处理对象，这就需要时间进行访问和处理，而且学习起来很难。但 Perl 的文本处理和正则表达式分析能力仅次于 Gawk。

Python 的内置函数集比 Perl 小，但具有很大的扩展库，可解决多数所谓的“缺少”元素(请参阅本章“Python 不适用的范围”部分)。面向对象是 Python 的最大特征，它应用于各处，包括提高灵活性和扩展性的扩展模块。如果想在完全面向对象环境下工作并利用 OO 特征尤其是继承性和多态性，应选择 Python。

最后一部分的比较不太明显。Python 整洁、易于阅读。使用复杂标点而非文本字符时，Perl 容易使人混淆。虽然 Perl 的某些方面使代码易于阅读，但是，还必须小心处理诸如变量前缀、大量使用括号的引用等元素。不仅如此，有些不可思议的变量应用于信息处理，不同标准和无标准快捷方式使 Perl 的调试变得很有挑战性。简而言之，许多 Perl 代码看上去很混乱。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

1.6.2 Java

与准解释性语言 Python、Perl 不同，Java 是真正的基于编译器的语言：在宿主计算机的 Java 虚拟机(Java Virtual Machine JVM)上执行前，必须将 Java 源代码编译为字节码。Python 虽然将程序编译成字节码，但编译过程由解释器完成，因此可直接从原始文本源代码执行 Python 程序。

已编译的 Java 脚本具有更紧密的字节码格式，因此 Java 脚本执行速度比 Python 脚本快。但是 Java 脚本开发时间长，因为在执行前需要把 Java 应用程序编译为字节码格式。Python 脚本可直接运行，Java 脚本在运行前需要编译。

除了可直接使用 Python 外，还可向 Python 解释器交互式输入 Python 语句。可以测试语句和序列，以确保它们正常工作并在下步工作前进行验证。这通常意味着 Python 程序加入源文件前就曾执行过，因为程序员在解释器中对每个语句进行了检查。

Java 与 Python 的真正差别在于语言的编程思想不同。Java 是种底层语言，类似 C 和 C++，并继承了它们的设计和方法学。Python 是种高层语言，这有助于向程序员隐藏底层操作系统的复杂性，同时又提供完成任务时所需的所有工具和特征。因此 Python 更适合结构化框架中的应用程序的快速开发。

对任何面向对象的任务来说，Python 都是优秀的原型语言。因此在用 Java 设计和开发前，可用 Python 作初始开发。Python 和 Java 可混合使用；甚至有用 Java 编写的 Python 解释器(Jython)。Jython 使 Java 程序可使用 Python 对象，又使 Python 程序可使用 Java 对象。Jython 还可使用基于 C 的解释器编译的 Python 字节码模块，而无需转换。

Python 与 Tk 或 Web 接口方面是否比 Java 好呢？这在于个人的选择。本书的作者倾向于使用 Tk，因为 Tk 更简洁，而简洁通常意味着易于编程，尤其是小型工作。Tk 的 OO 类型非常适合 Python 语言。他认为 Tk 在开发新接口部件时更易于定制。Python 的跨平台兼容性使它也具有这个优点，而 Java 则在产品级别缺乏这个优点。另一方面，目前只有 Unix、Windows 和 MacOS 支持 Tk，而支持 Java 的平台则更多。在这种情况下，显然 Java 是更好的选择。

1.6.3 JavaScript

JavaScript 和 Python 在设计方面很相似，但它们的目标却不一样。两者都面向对象，都允许不使用对象特征而应用它们的函数和功能。但 JavaScript 不是真正的脚本语言，就 JavaScript 的设计思想而言，它不是 Python 的竞争者。

JavaScript 是控制用户和 Web 页面间交互作用的嵌入式语言。JavaScript 作为 HTML 文本的一部分存储，并在访问页面时被传送，因此在用户浏览器和 Web 服务器间没有通信，这一点和基于 CGI 的接口的情况是一样的。JavaScript 不是单机语言；在 Web 页面外 JavaScript 不起作用。另外，JavaScript 没有真正的跨平台兼容性和安全性。

熟悉 JavaScript 编程的程序员会非常适应 Python 环境，因为许多术语和结构是相同的。程序布局和语义也很相似。只要了解 JavaScript 的原理，很容易转到 Python 的学习。Python 具有的某些扩展能力是 JavaScript 所不具备的，Python 为 JavaScript 程序员提供了更加开放的环境。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

1.6.4　Tcl

经过几年发展，Tcl 语言成为成熟的编程语言，但它是作为宏语言开发并用于扩展已存在的应用程序。多数人在使用 Tk 接口构造器开发应用程序时会想到 Tcl。Tk 为创建基于窗口的应用程序提供了统一和强大的接口。Tk 是跨平台的，它支持 Unix 的 X Windows 系统及其他支持 X Windows 的操作系统，如 QNX；支持 Windows 95/98/NT 和 MacOS 的本地环境。但 Tk 仅限于这三类平台；若使用 Tk 作为跨平台解决方案，将无法在某些较少使用的操作系统如 BeOS 上使用。

Tcl 的缺点在于它在数据结构方面较弱。而 Perl 等语言则不然，本章曾讨论 Python 较其他语言的长处在于数据结构和处理能力。Tcl 的最近版本，特别是 8.1.1 版已经增强了对数据结构的支持，并提供字节编译器以提升 Tcl 应用程序的性能。但与 Python 相比，Tcl 还是显得笨拙而缓慢。由于拥有与 Tk 的优秀接口，Python 提供了开发应用程序简单、快速的解决方案。

1.6.5　Rebol

Rebol 是种很新的语言。和 Python 一样，Rebol 是采用面向对象的思想开发出来的。Rebol 作为消息样式语言设计，重点支持使用传输和交流信息的消息。这使得 Rebol 适合于基于 Internet 的服务，因为 Rebol 可以与电子邮件、Usenet、Web 和 FTP 服务器直接会话，而且知道如何访问和处理这些服务，因此可用 Rebol 开发高复杂度的网络识别的应用程序，而不像 Perl 和 Python 需要许多工具包。

Rebol 功能很集中，致使开发普通程序对它不适合，而通常用 Perl 或 Python。不过随着这门语言的成熟，情况可望改观。

1.6.6　Visual Basic

Visual Basic(VB)是 Microsoft 提供的开发环境。它作为早期 DOS 版本下 BASIC 的扩展。和 JavaScript 一样，VB 有特定的市场目标。与彻底的编程语言如 Python 比较，它的局限性在于特定领域。

虽然经常被定位于普通编程语言，但实际上 Visual Basic 最适合数据库接口的开发。虽然可以用 Microsoft Access 或 Visual FoxPro 等应用程序开发复杂的数据库环境，但要开发完全定制的数据库接口时要用到 VB。

VB 提供的是一种完全定制的语言，它拥有访问 ODBC(Open Database Connectivity 开放数据库连接)兼容数据库的数据的直接挂钩。使用 VB，只要访问相同结构，就可决定显示和输入的内容。但最终还是与用户接口语言打交道。

Python 提供许多访问不同数据库系统的扩展，包括 Oracle、Informix、mSQL/mySQL，当然还包括 ODBC。学会用 Python 编制其他元素后，不仅能向用户显示信息或获取信息，还可以通过网络进行数据通信或把 Python 嵌入 Web 页面。可能更有用的是，因为 Python 支持这么多的数据库系统，所以可以用 Python 将一个系统的数据翻译并转移到另一个系统。

1.6.7　Awk/Gawk

Awk 是种很古老的编程语言。Unix 操作系统的早期版本就提供 Awk。虽然 Awk 拥有固定

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

的支持者(包括作者本人)，Awk 和它的 GNU 版本 Gawk 还是被定位成文本处理语言。Awk 和 Gawk 的内置功能可能比 Perl 支持的功能还多。在总结和汇报大量数据时，Gawk 的性能远远超过 Python 或 Perl，甚至 C 的性能。

Gawk 有局限性，但它不限于文本处理。最近的 Gawk 版本虽然还没有真正的网络套接字工具，但也包含了有限的通信工具。Gawk 还缺乏文件管理能力。

相对于 Python 和 Perl，Gawk 的最大优点是：易于编写快速脚本来处理 Unix shell 中命令的输出。一到两行的 Gawk 代码实现的功能用 Python 脚本要好几行才能实现；而在 Unix shell 中，需要更多管道函数通过 cut、paste 和 expr 实现。

Awk 是多数 Unix 发行版的标准配置。虽然有许多现代 Linux 发行版包含了 Perl 和 Python，但多数商业发行版还不包含 Perl 和 Python。Python 与所有版本兼容，但需要源代码，并需进行编译和安装。

最后，Awk 的正则表达式语法比 Perl 或 Python 先进。可以在置换表达式的搜索和替代部分使用正则表达式语法。甚至可以在正则表达式中进行嵌入式搜索，而在多数其他语言中需要进行循环。

1.6.8　C/C++

C 是多数操作系统的源代码语言，也经常用来开发新语言，包括本章提到的其他脚本语言。原因很简单：C 是一种优秀的低层编程语言，C 提供对操作系统多数部件访问的功能，开发新语言时易于模仿和提供开发其他语言所需的函数及便利工具。Python 本身用 C 编写，这使 Python 具有跨平台兼容性的功能——对多数操作系统而言，C 是系统级语言。

但相对于 C 和 C 的面向对象版本 C++而言，Python 提供更多便利功能。首先，Python 是一种较高层的语言，在函数和结构范围内，它隐藏了 C 的低层复杂性。这并不意味着不能用 Python 进行低层开发；只要人们愿意，就可利用便利的工具，不过用 Python 编程时不需这些工具。

C 和 C++都是复杂的语言，它们使用了许多不同术语和结构以支持语言的函数和功能。尤其是 C++的面向对象的特征非常复杂，使编程样式很多。多数情况下在同一程序中混合、匹配 C++和 C 函数是十分困难的。开发 C++应用程序需要与 C++完全兼容。采用 Python 后，在需要时可选择使用面向对象方式。另外，无法将 C++程序完全从一个平台移植到另一个新平台。两个平台要使用兼容编译器和运行时库，而这并不是跨平台的标准配置。

但应记住，Python 毕竟是用 C 开发的，Python 的所有功能均依赖于 C 源代码。当然，这种对 C 的依赖不是限制；事实上，可用 C 或 C++对 Python 语言进行扩展。在 C 程序中可使用 Python，C 程序允许嵌入式 Python 语句。嵌入特性在下列情况下非常有用：即程序需要支持脚本，但又不想开发一种新的语言时。

1.6.9　Unix/DOS Shells

虽然 Unix shells 和 DOS 命令行不能真正是编程语言，但它们提供基本的脚本处理能力。Unix 下主要的 shell 有 Bourne、Korn 和 C shells，以及一些变种如 bash 和 tcsh shells。很多高级 shells 提供数据处理和存储功能，但非常有限。任何情况下，在 shells 中处理或整理信息需要使

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

用 sort 或 cut 之类的外部工具；许多程序实际上使用 Awk 或 Gawk 来实现更复杂的功能。

高级 shell 编程大量使用数据管道，通常用到 10 或 15 个进程，并由数据管道进行组合以完成简单操作。而 Python 只需一到两行代码就可完成。因此，对多数方案来说，shell 脚本编程是可能但不现实的。

在 DOS 和 Windows 下，脚本处理能力通过批处理文件实现。类似于 shells，批处理文件只适合执行简单命令序列；而不适合真正的编程。数据存储仅限于环境变量，只能在特定命令中使用数据管道或重定向。尽管许多 DOS 产品进行了努力，但与本身环境相比，它更多地受到了可用到的命令的限制。DOS 提供的工具甚至不如 Unix 标准版提供的 1%。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第2章 Python基本原理

在解释 Python 语言的语义前，需要了解如何执行 Python 程序。与许多其他脚本语言有点不同的是：有多种执行 Python 语句的方法，包括直接与 Python 解释器交互执行。

许多编程语言在执行应用程序时遵循下列步骤：

(1) 在一个或多个源文件中编写应用程序。

(2) 将源文件编译进目标文件。

(3) 将目标文件与应用程序链接。

(4) 执行应用程序。

脚本语言的解释器直接使用源文件，因此步骤要简单一些。可能使用的步骤如下：

(1) 在一个或多个源文件中编写应用程序。

(2) 提供主源文件，运行解释器。

对于 Perl、Python 和其他脚本语言，第一个步骤的第 2、3 步实际上已内置到解释器。执行应用程序时，解释器将源代码编译为中间字节码格式，接着用虚拟机执行已编译的程序。虚拟机执行字节码的方式和已编译的应用程序执行本地机器代码的方式相同。

2.1 执行 Python 程序

和其他编程语言一样，用 Python 编写应用程序也遵循相同的基本过程。Python 解释器读取原始文本源代码并执行每条语句。不同的是，可直接在 Python 解释器中执行语句——而无需在执行前将 Python 语句放进文件中。

另外，Python 允许在文件中记录已编译的字节码指令；这样就无需首先编译源代码而直接可以执行字节码。虽然在应用程序中通常不采用这种技术，但它用在 Python 语言的模块和扩展中。

还有一些执行 Python 应用程序的其他方法。下面逐个介绍。

2.1.1 交互式

Python 解释器与其他解释器稍微不同的一点在于可直接向解释器输入 Python 语句，而无需创建文件，然后再执行该文件。这在测试特定语句或短小应用程序时非常有用。

1. Unix

Unix 环境下(Windows 和 MacOS 稍后介绍)，用命令行方式执行 Python 解释器，并输入 Python 语句，如下例所示：

```
$ python
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
Python 2.0   (#4, Dec 16 2000, 07:30:29)
[GCC 2.95.2 19991024 (release) ] on sunos5
Type "copyright", "credits" or "license" for more information.
>>>
```

输入 Python 语句时，解释器自动编译、执行该语句——无需通过编译过程。交互式方式使人们可以在一次会话过程使用不同语句。在执行语句前可在命令行中进行修改。按下回车键后，就会执行该语句(无法返回去修改)。

在 Python 命令行输入下列语句：

```
>>> print 63*56
3528
>>>
```

正如人们所看到的，程序作出响应并立即显示正确结果——这表明已编译、执行了该语句。

交互式接口有多种形式——可以一次执行多行语句，就像把单个语句输入到每个文件中执行。这意味着可直接向解释器输入小程序并观察结果。考虑下面的示例：

```
Python 2.0   (#4, Dec 16 2000, 07:30:29)
[GCC 2.95.2 19991024 (release) ] on sunos5
Type "copyright", "credits" or "license" for more information.
>>> pi=3.141592654
>>> radius=4
>>> area=pi*(radius**2)
>>> print 'Area of circle is:',area
Area of circle is: 18.849555924
>>>
```

目前不考虑语法问题——本书将大量使用交互式接口以便在 Python 中快速演示语句和函数的操作及结果。现在只需知道如何使用语句。

要离开 Python 的交互式解释器，只需按下结束文件(end-of-file)组合键(在多数 Unix 终端中为 CTRL-D)。

2. Windows

一旦安装 Python 解释器后，就可以像 Unix 中一样交互式执行 Python 语句：

```
C:\> python
Python 2.0 (#8, Oct 16 2000, 17:27:58)   [MSC 32 bit (Intel) ] on win32
Type "copyright", "credits" or   "license" for more information.
>>>
```

可能需要把包含 Python 解释器的目录添加到您的%PATH%上。在 Windows 95/98 中，需向 AUTOEXEC.BAT 文件添加目录信息。在 Windows NT 中，需更改系统控制面板的 PATH 设置。当然，也可在开始菜单找到 Python 解释器的命令行。

进入交互式解释器后，工作方式和 Unix 版本一样。不过，退出解释器需要使用 CTRL-Z

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

组合键。

3. MacOS

在 MacOS 中，需要开始运行 Python 应用程序。这样会打开简单的、终端样式的窗口，即 Python 解释器的交互式接口。图 2-1 显示了 MacOS 下运行中的 Python 应用程序。

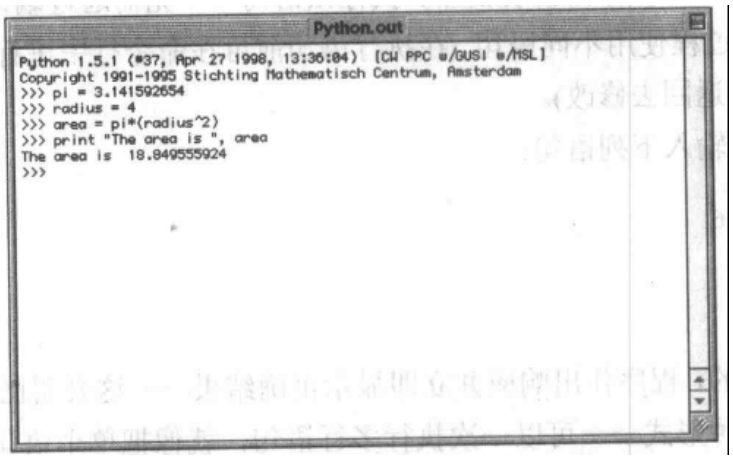

图 2-1　在 MacOS 中使用 Python 解释器

2.1.2　从文件执行

交互式输入语句的难点在于不能存储输入的语句——执行完语句后立即删除该语句。和其他脚本语言一样，可以将 Python 语句输入到文件中，并作为一个完整应用程序执行。在其他脚本语言中，将这种文件称为脚本文件、批处理文件或直接称为应用程序。

Python 中的脚本文件称为模块。Python 模块没有特殊格式：它是简单的文本文件。因此可用任何文本编辑器进行编辑，如 Emacs、Notepad、BBEdit 等等。采用模块的主要好处是：无需重复输入语句，而可以多次使用模块中的这些语句。

习惯上，给 Python 模块加上扩展名 .py 表示它们是 Python 脚本。在 Unix 和 MacOS 下，扩展名的意义不太明显，但 Windows 中可直接双击文件执行带扩展名的模块。本书将简要解释不同平台的特定选项。

相比于其他语言，Python 模块的另一优点是：Python 模块可无需修改而输入到另一模块——不必修改原始模块或改变模块名。甚至不必用专用方式编写模块，只要将文件以扩展名 .py 存盘即可。这使得共享函数和对象更加容易，也提高了已创建的对象的重用率。

1. 作为 Unix 脚本

在 Unix 下创建 Python 模块时，如果要在模块中执行语句，需要为 Python 解释器提供模块名。以一个简单脚本为例：

```
import sys
print 'Hello ', sys.argv[1]
```

把这些命令保存为 test.py 文件。把文件名作为第一个参数提供给 Python 解释器：

```
$ python test.py Martin
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
Hello Martin
```

把下面一行作为首行插入到文件中。该行被称为 shebang line:

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

还必须把文件模式改为可执行文件；这样 Unix shell 将检查 shebang line 以决定哪个应用程序执行该文本式脚本：

```
$ chmod 755 test.py
```

现在可直接运行脚本：

```
$ test.py Martin
Hello Martin
```

Shebang line 告诉 Unix 用什么应用程序执行该文件。上例中的 shebang line 直接指定 Python 解释器的位置。更兼容的方法是使脚本搜索环境变量 PATH 的值。使用如下 shebang line：

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

要确保实用程序 env 的位置正确，否则 shell 将出现错误信号。可以在 /bin、/usr/bin、/usr/local/bin 或 /usr/sbin 中找到 env。

配置 Python 解释器

无论采用哪种方式执行 Python 脚本，都可能需要向 Python 解释器提供额外的命令行开关。在 Python 解释器中手工执行脚本时，必须在提供脚本名之前给出那些命令行开关选项。例如，要在脚本执行后使解释器进入交互模式，按如下方式指定：

```
$ python –i sample.py
```

或者使用如下 shebang line：

```
# ! /usr/local/bin/python –i
```

表 2-1 包含命令行选项和环境变量的完整列表。

表 2-1　Python 命令行选项和环境变量

选　项	环 境 变 量	描　述
-d	PYTHONDEBUG	脚本编译后从解释器产生调试信息
-i	PYTHONINSPECT	脚本执行后使解释器进入交互模式
-O	PYTHONOPTIMIZE	在执行前对解释器产生的字节码进行优化
-OO		在执行前对字节码进行优化并删除优化代码中的嵌入式文档字符串
-S		运行解释器时不自动输入 site.py 模块，该模块包含特定站点的 Python 语句

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

选　项	环 境 变 量	描　述
-t		当脚本的 tab 缩排格式不一致时产生警告，参见第 3 章关于 Python 块的叙述
-tt		当脚本的 tab 缩排格式不一致时产生错误(并停止分析)
-u	PYTHONUNBUFFERED	强制标准输出和错误文件句柄无缓冲操作，如未指定，采用缓冲输出
-v	PYTHONVERBOSE	脚本执行时产生输入模块的信息
-x		忽略源文件的首行，在不同平台上执行脚本，并要忽略 shebang line 时有用
-X		使解释器中基于类的异常无效(详见第 6 章)
-c cmd		用 cmd 替代源文件作为脚本源码
-		从标准输入读取源文件

配置选项时，根据该选项所访问的变量和变量的值决定是否设置该选项。例如，和下例一样仅创建变量是不够的：

```
$ export set PYTHONINSPECT
$ sample.py
Hello World!
```

还必须给变量赋值：

```
$ export set PYTHONINSPECT=1
$ sample.py
Hello World!
>>>
```

另外，Python 支持表 2-2 所列举的环境变量。

表 2-2　Python 环境变量

变　量	描　述
PYTHONSTARTUP	交互模式下运行解释器时执行的文件名
PYTHONPATH	输入模块时搜索的目录列表(Unix 下用冒号分隔，Windows 下用分号分隔)，列表结果为 sys.path
PYTHONHOME	核心 Python 库所在目录。默认为 $PYTHONHOME/python2.0

2. 在 Windows 主机上

在 Windows 环境下有两种选择。若从 DOS 提示符下执行 Python 脚本，只需向应用程序提

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

供文件名：

```
C:\> python test.py
```

另一方法为在 Windows 资源管理器中(文件夹选项的文件类型 tab)将文件扩展名定义为 .py。安装解释器后，Python 安装程序会自动完成这些工作。在 Windows 资源管理器窗口双击文件时，Python 解释器将打开并显示文件，如同在命令行输入指令。

双击文件方式存在的惟一问题是：不能在命令行为脚本提供参数；当脚本执行时只有添加必要代码从用户请求信息，脚本才能交互式工作。解决方法为：创建批处理文件以执行 Python 解释器、脚本和提供给脚本的参数。

最后一种方案不用打开命令行提示符。如果文件扩展名定为 .pyw，则脚本运行时不打开 DOS 命令提示符，脚本和普通 Windows 应用程序一样。但是采用这种方式后，必须在脚本中定义，否则没有交互式的表单——需要打开控制台窗口，或者在脚本中开发基于 Tk 的接口以允许用户交互式操作。第 14 章叙述了在 Python 中用 Tk 开发接口的方法。

Windows 下 Python 不支持 shebang line。Unix 下的 shebang line 定义运行解释器时使用的选项。这意味着运行脚本时如要向解释器提供特定选项，必须在批处理文件中嵌入调用。这实际上是操作系统的限制，而不是 Python 的限制。Windows 中可用的命令行选项与 Unix 中可用的选项相同。请参阅表 2-1。

3. 在 MacOS 主机上

MacOS 下无法通过文件名识别文件类型。MacOS 不使用扩展名识别文件类型。它采用一种特殊系统，该系统使用特殊的 4 字符类型和创建者代码。例如，MacOS 下，Python applet 的类型为 TEXT，创建者为 Pyth。必须用特殊工具如 FileTyper 或 Snitch 来修改此信息。

在 MacOS 机器上执行 Python 脚本需要创建普通文本文件。可使用 SimpleText 应用程序，或者使用编辑器如 BBEdit、Emacs 创建文件。甚至可使用 Microsoft Word 或 AppleWorks——只要保证将文件存为普通文本格式，而非 Word 或 AppleWorks 文档。

脚本文件生成后，需要把它拖放到 Python 应用程序，以便使 Python 执行脚本。这时双击文件只会在编辑器中再次打开该文件。注意：执行文件时，将打开 Python 应用程序，并运行脚本，然后退出应用程序。要向用户显示信息，可通过等待输入的方式暂停输出，或者延迟程序退出的过程。处理其他执行选项的方法请参阅下节内容，“在 MacOS 中配置 Python”。

若希望创建通过双击就可执行的 Python 应用程序，需使用 BuildApplet 应用程序。将 Python 脚本拖放到 BuildApplet 中，则创建了新应用程序。双击新建的应用程序，将加载 Python 库并执行脚本，如同把脚本拖放到 Python 解释器。

但需要注意，应用程序不完全独立——在最终用户安装 Python 解释器后才可向它们提供应用程序。事实上创建的应用程序只是一个使 Python 脚本更易于执行的占位符。

修改源码脚本后，需要重新创建 applet 以保持与此次修改同步—— 修改不会自动应用到 applet 中。

在 MacOS 中配置 Python

MacOS 不支持环境变量和命令行选项。为修改解释器的操作或 Python applet，必须使用

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

EditPythonPrefs 应用程序。如果要修改主 Python 应用程序和拖放到解释器的脚本，只需双击 Python 应用程序。首先显示图 2-2，可以为解释器指定库搜索路径。也可指定解释器执行的 HOME 位置。在寻找输入模块时要用到这些信息。修改此信息和修改 Windows 和 Unix 下的 PYTHONPATH 和 PYTHONHOME 环境变量相似。

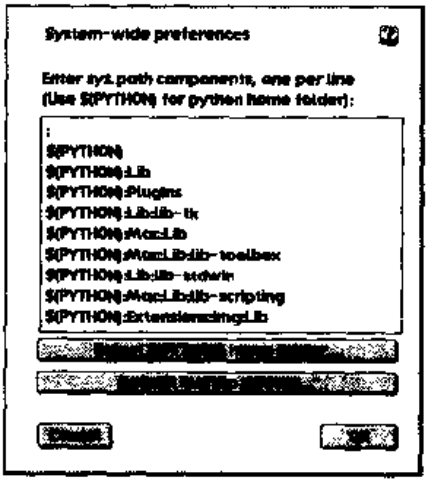

图 2-2　在 MacOS 下编辑 Python 的 PATH 选项

单击 Default Startup Options(默认开始选项)，可以配置许多 Unix 和 Windows 用户可用的选项，有些选项只出现在 MacOS 平台。图 2-3 显示了该选项窗口。

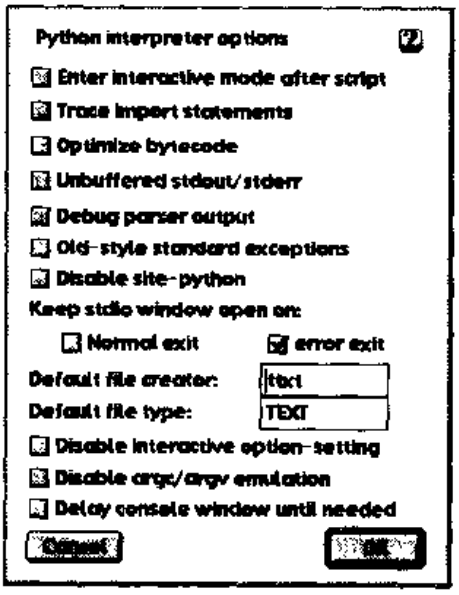

图 2-3　为 Python 解释器编辑其他 MacOS 选项

表 2-3 为图 2-3 的窗口中与命令行匹配的选项列表。

表 2-3　Python 解释器中等价于命令行的 MacOS 选项

MacOS　选项	等价的命令行
Enter Interactive Mode After Script (执行脚本后进入交互模式)	-I
Trace Import Statement (跟踪输入语句)	-v
Optimize Bytecode (优化字节码)	-O

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

MacOS　选项	等价的命令行
Unbuffered stdout/stderr (无缓冲标准输出/错误)	-u
Debug Parser Output (调试分析器输出)	-d
Old-Style Standard Exceptions (旧标准异常)	-X

表 2-4 描述了 MacOS 中 Python 解释器的其他选项。

表 2-4　Python 解释器中特定于 MacOS 的选项

MacOS 选项	描　　述
Keep stdio Window Open on Normal Exit	即使脚本成功执行，也使 Python 应用程序创建的控制台窗口在脚本退出后保持打开状态
Keep stdio Window Open on Error Exit (default)	仅在发生错误或异常时，使 Python 应用程序创建的控制台窗口在脚本退出后保持打开状态
Default File Creator	当 Python 脚本创建文件时指定 4 字母代码作为文件创建者。创建者代码用来定义双击文件时用哪个应用程序打开。TeachText/SimpleText 使用代码 ttxt，Microsoft Word 使用代码 MSWD，BBEdit 使用代码 R*ch
Default File Type	用 Python 脚本创建的文件指定 4 字母代码作为文件类型代码。TEXT 为普通文本文件，bina 为二进制文件。其他由应用程序/操作系统指定。如提供公用应用程序且使用自己的代码，需要在 Apple(苹果机)中注册该代码
Disable Interactive Option-Setting	启动解释器时选择 OPTION 键，可以动态指定某些选项 设置该选项则禁用该功能
Disable argc/argv Emulation	启动解释器时选择 OPTION 键，可以提供命令行选项 设置该选项则禁用该功能
Delay Console Window Until Needed	如果应用程序不需要或不使用控制台窗口，可设置该选项以避免显示空白窗口

注意，也可将 applet 拖放到 EditPythonPrefs 应用程序中，从而在单个 Python applet(但是不是脚本)中指定这些选项。

2.1.3　其他方法

当然还有其他方法可以执行 Python 语句。Python 语言不仅仅编写成一种简单的解释器。还可以把 Python 解释器嵌入 C 或 C++应用程序中，在其他应用程序中执行 Python 语句甚至整个模块。对本书来说该论题过于高级，因此不进行深入讨论。但这也是另一种实用且很有效的执行 Python 应用程序的方式。

把 Python 嵌入应用程序的其他信息可参阅第 28 章。或参阅 Python 提供的文档。标准文档集包括手册 Extending and Embedding the Python Interpreter(《扩展和嵌入 Python 解释器》)，它

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

包含所需的全部信息。附录 B 包含获取和使用 Python 文档的信息。

2.2　脚本、程序或模块

Python 新用户有时会对 Python 应用程序的不同名字之间的区别感到迷惑。Python 应用程序是脚本、程序、模块或者是其他什么东西？

严格地说，包含 Python 语句的文本文件应该指模块。Python 中的模块有特殊意义。可以执行它们，也可将它们输入到其他模块。有些用户可能更加迷惑了，因为他们常根据不同文件类型区分术语。Perl 中把可输入脚本称为模块而非脚本。Python 对两者不作区分，创建可输入 Python 模块也没有特殊技巧。

但是，Python 是种脚本语言，因此称 Python 模块为脚本也是正确的。另外，完成某一任务的 Python 模块也可称为程序或应用程序。这两个术语适合于那些指示计算机完成特定任务的文件。

最后，各术语间并无本质差异，但对于包含任何 Python 语句的文件，广为接受且推荐使用的术语为模块。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第3章 Python程序的组成

现在大家应该已经知道如何编写简单的 Python 程序，并会用不同方式执行程序，下面进一步深入讨论 Python 语言。本章首先叙述 Python 程序的基本组成。大家以前可能进行过某种形式的编程，那么不妨将编程描述为操作变量，因为无论何种语言编写的程序都可这样描述。

通过本章的学习，应该记住 Python 是对对象进行处理。实际上，任何 Python 编程练习都是这样进行的。虽然 Python 中出现的对象与其他语言使用的变量很相似(本章将进行比较)，但事实上，所有 Python 变量都是对象。

和许多其他语言一样，Python 遵循一些相同的基本原理：

- 独立的应用程序由若干文件组成，Python 中称为模块。
- 每个模块由一些语句组成。
- 语句创建、使用并修改变量；Python 创建对象变量。

本章讲述组成 Python 程序的两个主要部分——内置对象类型和语句。逻辑流程为从最低级的普通对象到最高级的模块。大家首先考虑对象，接下来考虑语句。在掌握了 Python 语言的基本组成后，在第 6 章深入讨论模块。

3.1 内置对象类型

所有语言均支持一些内置变量类型。它们用于创建语言中的所有其他可用变量类型的块。通过基本变量的组合或扩展，可以创建出非常复杂的变量。例如，C 语言中的基本变量类型 char 即单个字符。通过定义 char 数组，可以创建文本字符串；通过定义二维数组，可以创建数组或字符串。

Python 支持许多内置对象类型。C 语言通过提供最低级的命名来处理变量问题，而 Python 支持许多对多数程序更加实用的高级变量。与其他语言不同的是，Python 变量实际上是对象。Python 的基本存储单元为对象，因此 Python 是完全基于对象的。这有许多好处。可以用相同方法和函数操作不同类型的对象。而且，新对象继承了对象方法，在自建的复杂对象中可继续使用相同方法。本章稍后将给出一些示例。

和其他脚本语言类似，Python 处理对象的创建、内存分配和允许您使用对象的访问例程。这与 Visual Basic(VB)和 Perl 的模式相同，但不同于 C。在 C 中，程序员负责为变量和需要存储的信息分配内存。

Python 支持许多内置对象类型，使程序编写更加容易。Python 提供的许多基本类型需要单独库，或者在能使用对象前需进行开发处理。提供这些内置、优化的对象类型后，可不必担心对象的复杂性，而直接开始编程工作。

使用内置对象类型还有另一个好处。使用内置对象类型的代码经过高度优化(和调试)，并

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

经过多年开发而证明是高效的。虽然为了在应用程序中也可用其他方法来实现相同对象而编写您自己的例程，但要在 Python 解释器提供的内置对象基础上提升性能是很困难的。

3.1.1　Python 对象和其他语言

Python 支持五种主要的内置对象类型和一种类似内置对象可被访问的外部数据类型。内置对象是 Number(数字)、String(字符串)、List(列表)、Dictionary(字典)和 Tuple(元组)。

Python 支持的基本外部数据类型是文件。把文件当作一种数据类型看上去很奇怪，不过 Python 支持像访问内置对象类型一样访问文件。本章将讲述其工作机理并对文件和对象类型进行比较。

人们可能对有些内置对象类型比较熟悉，如数字型和字符串型，甚至列表型。不过 Python 支持这些对象的一些不同选项。而人们对其他对象类型的名字可能很陌生。表 3-1 对 Python 内置对象类型和其他语言的数据类型进行了比较。

表 3-1　Python 和其他语言的数据类型

Python	C/C++	Perl	Visual Basic
Number	int, long, float, double	Scalar	double, integer, long, single, currency
String	char[] 或 char *	scalar	string
List	int[] 或 char[]	array	任何 array 类型
Dictionary	N/A	hash 或 associative array	N/A
Tuple	参见正文	参见正文	参见正文
File	FILE * 或 fd	FILE	无内置类型

Tuple 元组是一种不能修改的特殊列表类型。因为它与列表相似，在 C/C++、Perl 或 VB 中也可用普通数组替代。关于 tuple 与普通列表的区别请参阅本章的“Tuples”部分。

与其他语言不同的是，Python 在为对象分配信息的方式时定义用来存储信息的对象类型。在 C 中必须明确定义创建的变量类型。在有些语言(包括 Perl)中，变量名的首字符告诉解释器对象的类型是什么。Python 解释器检查对象的信息，并创建或修改对象以与新类型相匹配。

观察下面的例子，代码显示了创建对象的不同方式。因为给变量分配了值，所以该操作称为赋值(assignment)：

```
integer = 12345
float = 123.45
string = 'hello'
list = [1, 'Two', 3]
dictionary = { 'one' : 1, 'two' : 2, 'three' : 3 }
tuple = (1, 'Two', 3)
```

在每种情况下，Python 都识别赋予变量的信息，并为需要存储的数据选用正确的对象类型。在单独讨论每种对象类型时，将学习更多关于赋值过程的语法知识。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

3.1.2　基本运算符

只讨论对象而不讨论运算符是远远不够的。运算符用于对对象进行操作。Python 用对象而非变量来存储信息，因此可以用相同运算符对不同对象进行操作。对每个讨论的对象，Python 自动调用相应方法执行操作。(请参阅本章稍后部分“运算符重载”)。

和其他语言一样，对于独立对象，Python 也采用相同的基本数学符号和格式的运算符。表 3-2 列出了运算符和它们的优先级。列在表尾的运算符优先级最高。

很多 C/C++、Perl 和 VB 程序员对这些运算符非常熟悉。例如，表达式 a+b 表示将变量 a 和变量 b 的值相加。其他运算符，如 lambda 和 in，是 Python 特有的。在学习每个对象类型的语法时将进行具体讨论。

表 3-2　Python 运算符和表达式优先级

运　算　符	描　　述
X or y	逻辑或(仅在 x 为 false 时，对 y 求值)
lambda args: 表达式	匿名函数
X and y	逻辑与(仅在 x 为 true 时，对 y 求值)
not x	逻辑非
<, <=, >, >=, = = , <>, !=	比较测试
Is，is not	识别测试
In，not in	是否为成员测试
X | y	按位或
X^y	按位异或
X&y	按位与
X<<y, x>>y	X 向左或向右移 y 位
X+y, x-y	相加/连接，相减
X*y, x/y, x%y	相乘/重复，相除，求余/格式化
-x, +x, ~x	一元否，同一，按位取补
X[i], x[i:j], x.y, x(...)	索引，分片，限定，函数调用
(...), [...], {...}, '...'	Tuple, list, dictionary, 转化为 string

1. 圆括号

尽管表 3-2 提到圆括号()，但其使用方法并不自动用于 tuple 的创建。可以用圆括号改变表达式的优先级。类似其他语言，首先对包含在圆括号中的表达式进行求值——圆括号的优先级在表 3-2 中是在其他运算符之前。

例如，表达式

```
result = 3*4+5
```

值为 17——与表 3-2 所示优先级一致。但注意表达式。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
result = 3*(4+5)
```

返回值为 27。

2. 表达式中的混合类型

应用混合类型时 Python 和 C/C++一样遵循相同的基本规则。如果表达式包含不同的混合类型，则返回值为表达式中最复杂的类型。例如，计算式：

```
result = 3*2.5
```

则 result 值为 7.5，且为浮点数字型，虽然该值为整数与浮点小数的值。对数字计算一般应用这项规则，但其他对象类型有些特殊情况。本书将逐个讨论。

3. 运算符重载

运算符重载指对不同对象类型使用相同运算符时执行不同的操作。从编程观点来看，语言支持运算符重载的能力可简化编程过程。例如，运算符“+”用于两个数字型对象时，会使两个数相加。而应用于两个字符串时，它们也相加在一起。也就是说，将两个字符串合并成单个字符串。运算符重载也适用于其他运算符。例如，运算符[]从字符串、列表和元组中提取字段。

理解这里使用的原则很重要——对每个对象运算符执行相同操作，即使对象用不同方式存储和使用它们的信息。

3.1.3　数字

Python 用一个简单对象来处理数字。在数字对象中存储哪种类型的数字没有限制。而 C 中存储整数和浮点小数使用不同的数据类型。除了这些基本类型外，Python 还支持复杂数字和任意大的整数。

大家已经知道，Python 根据创建对象时分配给它的值来决定所建对象的类型，并用这种对象类型创建对象。对于数字对象而言，数字格式决定存储信息的方式。因此应理解如何向 Python 程序加入数字常量。

1. 整数常量

可通过提供数字序列来创建整数对象，如下例所示：

```
number = 1234
number = -1234
```

所建对象为整数，内部存储方式和 C 中的 long 数据类型相同。它至少占 32 位，根据使用的 C 编译器和处理器不同可能更长。注意 0 也是数字：

```
zero = 0
```

这也是人们所希望的。在确定表达式的逻辑值时，Python 使用整数值。和其他语言一样，0 等价于 false，其他任何数字等价于 true。这样可以用整数表示 Boolean(布尔)值，而无需额外数据类型。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

在表达式中只使用整数常量(或对象值)时，Python 采用整数，而非浮点数字计算该例程。对于重复进行的计算，使用整数可提升程序性能。如前文所述，在表达式中使用浮点数值会使返回值成为浮点数字。本章“浮点数常量”部分将给出一些示例。

2. 十六进制和八进制数常量

用与 Perl 和 C/C++中相同的符号可指定十六进制和八进制数常量。即给数字加 0x 或 0X 的前缀会使 Python 把数字解释为十六进制数，加前缀 0 会成为八进制数。例如，设置值为 255 的十进制数，可采用下列方法中的任何一种：

```
decimal = 255
hexadecimal = 0xff
octal = 0377
```

因为它们都是简单整数，所以 Python 用存储十进制整数的方法存储它们。如果用八进制或十六进制格式打印整数，可采用格式化字符串对象的内插法。本章稍后将进行介绍。

3. 长整数

内置基本整数类型限于存储 32 位长的数。因此可表示的最大数为 2^{31}-1，另一位用于表示负数。多数情况下这就足够了，但有时需要使用长整数。Python 支持任意长整数——可创建数字长为一千的数并和其他数字一样使用。

为创建长整数，需在数字常量尾追加 l 或 L，如下例所示：

```
long = 1234567891234567891234567891234567891234567891234567891L
```

创建完长整数后，可将它当作普通数字用于表达式中。Python 解释器负责处理复杂的大型数字。例如，语句

```
long = 1234567891234567891234567891234567891234567891234567891L
print long +1
```

产生输出：

```
1234567891234567891234567891234567891234567891234567890L
```

或者继续做长整数计算：

```
long = 1234567891234567891234567891234567891234567891234567890L
print long+87654321987654321987654321987654321987654321987654321L
```

结果为：

```
2111111111111111111111111111111111111111111111111111110L
```

虽然在 Python 中可以和使用普通整数值一样使用长整数，但解释器需做大量额外工作以支持这种选择。在支持长整数的 C 中也是如此。因此，应优先使用内置整数或浮点小数类型。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

4. 浮点数常量

Python 支持用普通十进制和科学计数法表示浮点数。例如，下列常量均合法：

```
number = 1234.5678
number = 12.34E10
number = -12.34E-56
```

浮点数值在 Python 内部的存储格式与 C 中的双精度数相同，其精度尽可能高。注意没有 long(长)浮点变量。

记住浮点数或者浮点数与整数混合的表达式返回浮点数值。揭示这种现象的最简单方法是显示相同计算式的输出——一个采用整数常量，另一个采用浮点数常量：

```
>>> print 5/12
0
>>> print 5.0/12
0.416666666667
```

从前面的示例可以看出：第一个表达式返回 0——5/12 取整后的值，第二个表达式输出了所期待的小数部分。

在没有使用常量并因此无法控制计算式中的对象类型时，可使用内置 int 函数强制返回整数值。这会强制 Python 将整数表达式提供给函数。请参阅本节结尾部分“类型转换”。

5. 复数常量

Python 用普通符号支持复数。实部和虚部用加号分开。虚数使用前缀 j 或 J。例如，下例为复数常量：

```
cplx = 1+2j
cplx = 1.2+3.4j
```

Python 用两个浮点数存储复数，而不管数的原始精度。复数在 Python 内部是单独实体。对于包含复数的表达式，解释器自动执行复数计算。

6. 数值运算符

表 3-2 中的多数运算符适合于数值类型。表 3-3 列出更清楚的用于计算的运算符列表。所列出的都是常见的数值操作。

表 3-3　所有数字类型的数值运算符

操　作　符	描　　述
x+y	X 与 y 相加
x-y	X 减去 y
x*y	X 乘以 y
x/y	X 除以 y
x**y	X 的 y 次幂

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

操　作　符	描　　述
X%y	取模(返回 x/y 的余数)
−x	元减
+x	元加

有一系列移位和位运算符用于二进制运算和位运算；见表 3-4。注意这些运算符仅适用于整数；用于浮点数时将引起异常。

表 3-4　用于整数数值的位/移位运算符

运　算　符	描　　述
X<<y	左移(将二进制数 x 左移 y 位)如 1<<2=4
x>>y	右移(将二进制数 x 右移 y 位)如 16>>2=4
X & y	按位与
x | y	按位或
x ^ y	按位异或(xor)
~x	按位取反

除了这些基本运算符外，Python 2.0 引入了一系列增强的赋值运算符。例如，可将下式

```
a = a + 5
```

用增加赋值运算符表示为：

```
a += 5
```

即，表达式

```
x = x + y
```

可重写为：

```
x += y
```

下面是增加赋值运算符的完整列表。这些运算符的效果和表 3-4 中的基本运算符相同：

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

7. 数值函数

除了本章前面的表 3-2 提到的运算符外，Python 还有一组内置函数用于直接操作数值对象。表 3-5 是这些函数的列表。注意该列表不包含用于对象类型转换的函数。本章“类型转换”部分将解释类型转换函数。

在 math 模块中定义了更多数值函数和一些普通常量。第 10 章将讨论它们。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

表 3-5　数 值 函 数

函　　数	描　　述
abs(x)	返回数的绝对值，忽略正负。若 x 为复数，返回复数模
coerce(x, y)	用普通表达式规则将 x 和 y 变成常用类型，返回 tuple 值。例如，语句 coerce(2, 3.5)返回(2.0, 3.5)
divmod(x, y)	用 x 除以 y，返回包含商和余数的 tuple 值。该函数等价于(a/b, a%b)
pow(x, y [, z])	x 的 y 次幂。注意返回值的类型同 x 的类型。这意味着整数不能做指数为负数的幂运算，任何数的幂不能超出对象类型的范围。例如，表达式 pow(2, -1)和 pow(2, 250)都将引起异常。类似，表达式 pow(256, (1/2))返回 1，因为 1/2 舍入为 0。如果有参数 z，返回值为 pow(x, y)%z，但此时计算效率更高
round(x [, y])	将浮点数 x 舍入成小数位为 0 位。加上参数 y 后，舍入成小数位为 y 位。注意返回值依然为浮点数。用 int 将浮点数转换为整数时无舍入过程

3.1.4　字符串

Python 中的字符串与其他脚本语言不同。Python 字符串工作方式与 C 中的字符数组相同。即字符串是单个字符的序列。

序列这个术语非常重要，因为 Python 赋予基于序列的对象特殊的功能。其他序列对象包括列表(对象序列)和 tuple(对象的固定序列)。字符串也是固定的，即它们的位置无法改变。大家很快会明白它的实际意义。Python 字符串是本书首先介绍的 Python 支持的复杂对象。它们是许多 Python 支持的其他对象类型的基础。

用单引号或双引号定义字符串常量：

```
string = 'Hello World! '
string = "Hi there! You're looking great today! "
```

注意这里使用的两种格式。在 Python 字符串常量中使用单引号或双引号并没有区别。由于允许使用这两种格式，使得可以在双引号表示的常量中包含单引号，也可在单引号表示的常量中包含双引号，而无需对个别字符进行转义处理。

Python 还支持 3 引号块：

```
helptext = """This is the helptext for an application
that I haven 't written yet. It 's highly likely that the
help text will incorporate some form of instruction as
to how to use the application which I haven 't yet
written. Still, it's good to be prepared! """
```

Python 一直向字符串添加文本直到出现三个连续双引号。可以使用三个单引号或三个双引号来定义常量的起始与结束，只需保证使用相同数量的引号。另外，没必要遵循普通的多行表达式规则。除非在各行使用反斜杠符号，否则 Python 会合并文本中的行尾字符。打印前面的示例时，结果将会分成相同的五行。要创建完整的文本段落，需要在每行结尾添加反斜杠符号，

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

如下所示：

```
helptext = """This is the helptext for an application \
that I haven 't written yet. It 's highly likely that the \
help text will incorporate some form of instruction as \
to how to use the application which I haven't yet \
written. Still, it's good to be prepared! """
```

将多个置于引号中的常量放在一行会导致常量的连接。例如：

```
string = ' I ' "am" 'the' "walrus"
print string
```

产生输出为“Iamthewalrus”。注意空格被忽略了。

也可用运算符“+”连接字符串对象和/或常量，就像操作数字一样：

```
greeting = 'Hello'
name = 'Martin'
print greeting + name
```

但要注意不能将字符串与其他对象进行连接。表达式：

```
print 'Johnny'  + 5
```

会引起异常，因为数字对象不会自动转换为字符串。当然有多种方法解决这个问题。很快大家会看到一些示例。

可以用运算符“*”表示多行(重复)字符串。例如，表达式：

```
'Coo coo ca choo' * 5
```

生成字符串：

```
'Coo coo ca chooCoo coo ca chooCoo coo ca chooCoo coo ca chooCoo coo ca choo'
```

最后，Python 提供内置函数 len 用于返回字符串的字符数。表达式：

```
len('I am the walrus')
```

返回值为 15。

实际上，函数 len 可计算具有大小(size) (列表、字典型等)属性的任何对象的长度。

1. 字符串只是数组

就像前面看到的，字符串也是序列。这意味着既可以用数组符号访问字符串中的单个字符，也可像列表一样访问字符串。

和其他语言一样，用于字符串的数组符号遵循相同的基本格式，可为变量名追加方括号。例如：

```
string = 'I returned a bag of groceries'
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
print string[0]
```

显示字符串的第一个字符。

类似 C 的地方在于字符串中的第一个字符的偏移量为 0。与 C 不同的是，Python 允许指定索引范围，称为分片(slicing)，Python 不从字符串的首部，而是从字符串的尾部定位字符位置。例如，脚本：

```
string = 'I returned a bag of groceries'
print string[0]
print string[2:10]
print string[-1]
print string[13:]
print string[-9:]
print string[:-9]
```

产生以下输出：

```
I
returned
s
bag of groceries
groceries
I returned a bag of
```

分片的格式为：

```
string[start : end]
```

索引和分片的过程遵循以下规则：

- 返回的字符串包含从 start 起始到 end 但不包括 end 结束的所有字符。也不包含 end 指定的字符。
- 若指定了 start 但未指定 end，则一直向后分片，直至字符串结束。
- 若指定了 end 但未指定 start，则从 0 开始分片直至 end，但不包含 end 指定的字符。
- 若 start 和 end 均指定为负数，则索引从字符串尾部开始，而不是从首部开始。最后一个字符为–1。

当这些规则与图 3-1 所示的图表组合使用时意义就更明确了。

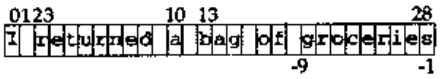

图 3-1　字符串的索引和分片

现在知道语句

```
print string[0]
```

打印第一个字符，

```
print string[2:10]
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

打印第 3 到第 9 个字符，

```
print string[-1]
```

打印最后一个字符，

```
print string[13:]
```

打印第 14 个字符以后的所有字符，

```
print string[-9:]
```

打印最后 9 个字符，而

```
print string[:-9]
```

打印除最后 9 个字符外的所有字符。

虽然 Python 允许对字符串进行分片及其他操作，但必须记住字符串是固定序列——不能改变字符位置。因此表达式

```
string[:-9] = 'toffees'
```

将引起异常。改变字符串内容的惟一方法是新建一个字符串。这意味着要进行某些简单操作，必须按下列顺序：

```
newstring = string[:-9] + 'toffees'
string = newstring
```

当然有改变字符串内容的简单方法。一种方法是使用外部模块(string)提供的函数来完成这项工作。另一种方法是使用字符串格式化，下面进行讨论。

2. 格式化

就像 C 中 sprintf 函数支持的符号一样，可以用 Python 运算符%来格式化字符串和其他使用相同基本符号的对象。但是因为使用的是运算符而非函数，其用法还是有些不同。将格式化的字符串置于运算符%的左边，需内插到返回表达式的对象(或 tuple 对象)置于右边。例如，语句

```
'And the sum was $%d, %s' % (1, 'which wasn't nice')
```

返回

```
And the sum was $1, which wasn't nice
```

返回值总是字符串。运算符%和 C 中 sprintf 函数一样接受相同的选项列表：请参阅表 3-6 所示的转换格式的完整列表。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

表 3-6　运算符%的转换格式

格　式	结　果
%%	百分号标志
%c	字符及其 ASCII 码
%s	字符串。实际上在打印前 Python 将相应对象转换为字符串，因此%s 可用于任何对象(其他细节请参阅“类型转换”部分。)
%d	有符号整数(十进制)
%u	无符号整数(十进制)
%o	无符号整数(八进制)
%x	无符号整数(十六进制)
%X	无符号整数(十六进制大写字符)
%e	浮点数字(科学计数法)
%E	浮点数字(科学计数法，用 E 代替 e)
%f	浮点数字(用小数点符号)
%g	浮点数字(根据值的大小采用%e 或%f)
%G	浮点数字(类似%g，在合适位置用 E 代替 e)
%p	指针(用十六进制打印值的内存地址)
%n	存储输出字符的数量放进参数列表的下一个变量中

Python 也支持调整输出格式的标记。它们在%和转换字母之间指定，请参阅表 3-7。

表 3-7　可选格式化标记

标　记	结　果
空格	用空格作正数的前缀
+	用加号作正数的前缀
-	区域左对齐
0	用 0 而非空格进行右对齐
#	用 0 做非零八进制数前缀，0x 做十六进制数前缀
数字	最小域宽度
.数字	指定浮点数精度(小数点后的位数)

运算符%也可用于字典型对象。要从字典抽取元素，需用以下格式引用字典名：%(NAME)后加格式化代码。格式化代码参见表 3-6 和表 3-7。例如，语句：

```
album = { 'title' : 'Flood', 'id' : 56 }
print "Catalog Number %(id)05d is %(title)s " % album
```

产生字符串“Catalog Number 00056 is Flood”。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

3. 转义字符

大家已经知道，在字符串常量中，如果只用双引号或单引号定义整个字符串，则可相应地嵌入单引号或双引号。如果用 3 个引号定义文本块，则可直接使用单引号或双引号。在需要向字符串插入引号或其他特殊字符的情况下，Python 支持反斜杠(\)转义字符。表 3-8 列出了 Python 支持的转义字符。

表 3-8　Python 支持的转义字符

转 义 字 符	描　　述
\(在行尾)	继续(在分析前追加下一行内容)
\\	反斜线符号
\'	单引号
\"	双引号
\a	Bell
\b	退格(Backspace)
\e	转义
\000	空。Python 字符串不以空结束
\n	新起一行或换行
\v	纵向制表符
\t	水平制表符
\r	回车
\f	换页
\0yy	八进制数 yy 代表的字符(例如，\012 等价于新起一行)
\xyy	十六进制数 yy 代表的字符(例如，\x0a 等价于新起一行)
\y	以上未列出的任何字符 y 以普通格式输出

4. 原始字符串

如果不想任何转义字符生效时，可用 r"和 R"定义原始字符串。例如，语句：

```
print r'\a\n\x99'
```

实际输出字符串"\a\n\x99"。

原始字符串主要用于正则表达式，其中反斜线符号用于对正则表达式进行转义操作。第 10 章详细讨论原始字符串和正则表达式。

5. 其他字符串操作

Python 有许多用于序列的方便的工具，这里并没有包含。它们是迭代、隶属关系(membership)和 min、max 函数。虽然这些函数可用于字符串在内的任何类型的序列，但通常用于列表或 tuple。其他细节请参阅本章后面的“使用序列”部分。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

3.1.5　列表

列表是另一种序列对象的形式，因此它继承了许多字符串的操作参数。字符串只是字符的列表，但与字符串不同的是，Python 列表可包含任何对象列表。可在列表中存储数字、字符串、其他列表、字典和自己创建的任何其他对象类型。因为它是对象列表，并且所有 Python 数据都存储为对象，所以列表可以是所选信息的任意组合。

Python 存储对象列表(或者指向对象的名字)，而不存储字符串或数字的列表。当需要信息列表但又不想仅限于存储信息的类型时，就可使用 Python 提供的列表。有关 Python 如何存储对象的其他信息，请参阅本章“对象”部分。

1. 使用列表

Python 中通过用方括号把一系列对象或常量封装起来从而创建列表，如下列语句：

```
list = [1, 2, 3, 4]
songs = ['I should be allowed to think', 'Birdhouse in your Soul' ]
```

使用方括号后会自动暗示方括号中的对象为列表。Perl 用户注意到：在 tuple 中用圆括号代替方括号，本章稍后将讲述。也可用下面的方式嵌套列表，实际上该列表包含两个列表：

```
hex = [ [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ], ['A', 'B', 'C', 'D', 'E', 'F' ] ]
```

类似于字符串，可根据表项的索引引用列表。索引从 0 开始。采用前面的示例，第一首歌曲的歌名可从下式得到：

```
print songs[0]
```

要访问完整的嵌套列表，需指定列表对象的索引：

```
print 'Numbers: ' , hex[0]
```

要访问嵌套列表的指定对象，用下列方法：

```
print 'D: ' , hex[1][3]
```

同理，相同的分片操作：

```
print 'A-C', hex[1][0:3]
```

要得到列表长度，用 len 函数：

```
print len( hex )
```

也可用运算符“+”连接列表，其工作方式同字符串和数字对象：

```
>>>hex[1] + list
['A', 'B', 'C', 'D', 'E', 'F', 0, 1, 2, 3 ]
```

另外，可以用增加赋值符号向列表中添加表项，但必须指定待添加的列表对象。如下例

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

所示：

```
>>> songs += ['AKA Driver' ]
```

最后，还可用数字对象与列表相乘，在列表内重复某元素：

```
>>>list * 2
[0, 1, 2, 3, 0, 1, 2, 3]
```

注意不能用列表与列表相乘，这样做没有意义。

但可创建基于简单表达式的新列表。例如，给定列表：

```
>>> list = [1, 2, 3, 4]
```

要得到每个元素的立方值的列表，可使用函数 map(第 9 章讲述)与匿名 lambda 函数(第 4 章讲述)的组合：

```
>>> cubes = map(lambda(x) : x**3, list)
```

在 Python 2.0 中，只需重写以上语句，得到下列语句：

```
>>> cubes = [x**3 for x in list]
```

2. 列表是可改变的

本书的前面部分曾经提到：字符串是固定对象，而列表是可变对象。其重要意义在于可在适当位置修改列表。对于字符串，下列操作会引起异常：

```
string = 'Narrow your Eyes'
string[7:11] = 'her'
```

但列表可采用下列操作：

```
list = [0, 1, 2, 4, 16]
list [3:4] = [4, 8]
```

结果为 6 元素的列表 [0, 1, 2, 4, 8, 16]。注意上式使用了分片操作。上例第二行若用 list[3]，则用嵌入式二元素列表替代第 4 个元素。另外，被替代的元素数目无需与插入的元素数目匹配。

从列表中删除表项，需要使用函数 del。该函数可以删除列表中的元素或分片：

```
del list [3]
```

删除第 4 个元素。而

```
del list [1:4]
```

删除列表中间 3 个元素。

3. 列表方法

熟悉对象的程序员知道许多对象都提供一组方法。方法是特殊函数，是对象类型定义的一

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

部分。方法实际上是对特定类型对象有效的函数。列表支持许多控制列表内容的默认方法。要使用方法，必须用对象名限定该方法。例如，调用方法 sort：

```
number = [3, 5, 2, 0, 4]
number.sort()
```

注意多数方法不会返回新列表，而是在适当位置对列表进行修改。在下面的例子中，对 numbers 对象进行了排序。如果要输出排序后的列表，那么操作就复杂一点，因为下列语句不会输出希望的结果：

```
print numbers.sort()
```

可以用以下两行替代：

```
numbers.sort()
print numbers
```

另一方面，对列表的操作也清楚显示了一点，即修改列表后再访问该列表，则返回排序后的列表。

表 3-9 列举了列表支持的方法。

表 3-9　列表对象支持的方法

方　法	描　述
append(x)	在列表尾部追加单个对象 x。使用多个参数会引起异常。追加 tuple 对象，则提供 tuple 参数，例如，list.append((1,2,3))。返回 None
count(x)	返回对象 x 在列表中出现的次数
extend(L)	将列表 L 中的表项添加到列表中。返回 None
Index(x)	返回列表中匹配对象 x 的第一个表项的索引。无匹配元素时产生异常
insert(i, x)	在索引为 i 的元素前插入对象 x。如 list.insert(0,x)在第一项前插入对象。返回 None
pop(x)	删除列表中索引为 x 的表项，并返回该表项的值。若未指定索引，pop 返回列表最后一项
remove(x)	删除列表中匹配对象 x 的第一个元素。无匹配元素时产生异常。返回 None
reverse()	颠倒列表元素的顺序。返回 None
sort()	对列表排序。返回 None

下面的例子使用了表 3-9 描述的方法：

```
list = [1, 2, 3]
more = [11, 12, 13]
list . append (4)              #  [1, 2, 3, 4]
list . append ( ('5a', '5b') )  #  [1, 2, 3, 4, (5a, 5b) ]
list . extend (more)           #  [1, 2, 3, 4, (5a, 5b), 11, 12, 13]
list . index (11)              #  返回 5
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
list . insert (5, 'Six')        #  [1, 2, 3, ('5a', '5b'), 'Six', 11, 12, 13]
list . pop ()                   #  返回(并移去)13
list . pop (4)                  #  返回 tuple ('5a', '5b')
list . remove ('Six')           #  [1, 2, 3, 11, 12]
list . reverse ()               #  [12, 11, 3, 2, 1]
list . sort ()                  #  [1, 2, 3, 11, 12]
```

虽然 None 适用于任何对象，但是，在前面大家还没遇到值 None。实际上 None 意味着无——它并不代表空列表或长度为 0 的字符串，而是没有值的对象。None 很重要，因为它可用来标识无值对象。类似 C 中的 NULL 值和 Perl 中的 undef。

4. 其他列表操作

因为列表是另一种序列，有些列表对象的功能这一部分没有提及。具体情况请参阅本章“使用序列”部分。

3.1.6 Tuples

不管目的是什么，tuple 基本等价于列表。只有一种例外：tuple 是固定不变的。对于列表，可根据需要对其进行截断、改变或修改，而 tuple 是不可修改的列表。一旦创建了新的 tuple，就不能进行修改了，除非新建 tuple。

创建列表可用方括号，而创建 tuple 时用圆括号定义 tuple 内容。如下例所示：

```
month = { 'Jan', 'Feb', 'Mar', 'Apr', 'May' 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec' }
```

以上语句很好地说明了在 Python 中使用 tuple 的方法。因为 tuple 不可修改，所以可用它们存储不希望改变的信息。上例中，人们可能会关心元素的顺序。如果用列表替代 tuple，并意外地对其进行了排序，则列举的月份就没有作用了。没有正确顺序的序列，其信息就没有价值。

注意，当人们试图修改 tuple 时，系统将产生异常以提示该错误操作。在开发和调试时这点非常有用，因为它有助于发现错误的使用对象类型的地方。

下面均为使用 tuple 的例子：

```
one = {1, }
four = {1, 2, 3, 4}
five = 1, 2, 3, 4, 5
nest = ('Hex', (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), ('A', 'B', 'C', 'D', 'E', 'F') )
```

理解以上语句的第一和第三行很重要。在第一行的情况下，创建一个元素的 tuple 必须使用看上去多余的逗号；否则 Python 解释器将(1)当作常量表达式。采用逗号后强制 Python 将语句解释为 tuple。

第三行语句没有采用圆括号创建 tuple。在不使用圆括号而不会导致混淆时，Python 允许不使用圆括号创建 tuple。该语句很明显创建了 tuple。一般规则是应该使用圆括号，以提示人们和阅读代码的程序员：这里引入的是 tuple。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

tuples 的使用

tuple 和列表一样支持相同的基本操作。可以对 tuple 进行索引、分片、连接和重复操作。因为 tuple 是一种序列，也可使用函数 len。列表中可用而 tuple 不能执行的惟一操作是：对 tuple 进行修改或使用修改对象的方法。

应该说明的是：tuple 操作返回 tuples。语句：

```
three = five [: 3]
```

使 three 成为 tuple，而非列表。

有些函数返回 tuple 而不是列表，有些函数甚至要求 tuple 作为参数。有关 Python 对象类型间信息转换的内容请参阅“类型转换”部分。

3.1.7　序列的使用

字符串、列表和 tuples 的对象类型均属于称为序列的 Python 对象。序列对象是可使用数字化索引进行访问其中元素的任何对象。字符串的每个字符，列表的每个对象通过索引均可利用。因为序列有顺序且由一个或多个元素组成，因此存在这种情况，即希望以序列或传统的值列表的方式将列表作为引用指针来使用。

Python 提供简单运算符 in 用于以逻辑方式访问列表内容。

1. 隶属

使用特殊结构的可能取值的列表时，如果要验证列表中某个元素是否存在，可选择两种方法。可以编写函数用于在列表的内容中寻找对象。或者使用字典之类的优先级高的数据结构存储表项，并用对象内置方法确定对象是否为伪列表的成员。

在 Python 中，可使用运算符 in 来确定当前对象是否为列表的成员。若在列表中找到该元素则返回 1，如下例所示：

```
>>> list = [1, 2, 3]
>>> 1 in list
1
>>> 4 in list
0
```

也可将 in 应用于字符串：

```
>>> word = 'supercalifragilisticexpialidocious'
>>> 'x' in word
1
```

或应用于 tuples：

```
>>> days = ('Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun' )
>>> 'Mon' in days
1
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

2. 迭代

大家还没有学习 Python 支持的循环运算符。Python 支持多数 while 和 for 操作，可以对序列的元素进行迭代操作。而且，Python 可用 in 运算符将序列中的每个元素返回到循环的每次迭代操作中：

```
for day in days
print day
```

本章后面的部分和本书其他章节有更多的相关示例。

3.1.8　字典

字典是一种联合数组。与列表采用数字引用独立元素不同，字典用其他对象(一般是字符串)引用元素。字典元素分成两部分：键(key)和值。通过指定键从字典访问值。

类似其他对象和常量，Python 用严格的格式创建字典：

```
monthdays = { 'Jan' : 31, 'Feb' : 28, 'Mar' : 31,
              'Apr' : 30, 'May' : 31, 'Jun' : 30,
              'Jul', : 31, 'Aug' : 31, 'Sep' : 30,
              'Oct' : 31, 'Nov' : 30, 'Dec' : 31 }
```

也可嵌入字典，如下例所示：

```
albums = { 'Flood' : { 1 : 'Birdhouse in your Soul' } }
```

要访问字典的条目，用方括号定义需返回的元素的索引条目。类似于字符串或列表：

```
print 'January has' , monthdays ['Jan'], 'days'
print 'First track is', albums ['Flood'] [1]
```

但是，如果没有找到指定的键，则解释器引起异常。关于 get 和 has_key 方法的细节请参阅本章稍后的“字典方法”部分。另外，和列表一样，可在适当位置重新定义条目来修改已存在的键/值对(key/value pair)：

```
monthdays [Fed'] = 29
```

如果要指定一个不同的键，则直接添加而不是更新条目。下例：

```
monthdays ['fed'] = 29
```

将在字典加入第 13 个元素。

用 del 函数删除字典的条目：

```
del monthdays ['Feb']
```

字典不是序列，因此不能进行连接和相乘操作。如果试图使用这两项操作将引发异常。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

1. 字典的使用

字典和列表看上去很相似，但实际上它们区别很大。字典没有顺序。和列表相似，索引(键)必须惟一，但字典的键没有逻辑顺序。意即字典不是序列。不能用访问字符串、列表或 tuple 中条目一样的方法去访问字典的条目。这带来了一个坏处：当分别两次访问相同字典时，由字典返回信息的顺序并不是一致的。

但字典对象提供两个方法 keys 和 values，用来返回字典中定义的所有键和值的列表。例如，要返回前面示例中月份字典的月份列表，使用以下语句：

```
months = monthdays.keys( )
```

现在 months 对象是一个月份字符串的列表，这些月份字符串构成了字典 monthdays 的键。函数 len 返回存储于字典中的元素，键/值对的数目。语句：

```
len (monthdays)
```

返回的正确值为 12。

2. 字典方法

表 3-10 是字典对象类型支持的方法的完整列表。

表 3-10　字 典 方 法

方　法	描　述
Has_key(x)	若字典中有键 x，则返回真(true)
Keys()	返回键的列表
values()	返回值的列表
dict.items()	返回 tuples 的列表。每个 tuple 由字典 dict 的键和相应值组成
clear()	删除字典的所有条目
copy()	返回字典高层结构的拷贝。但不复制嵌入结构，而只复制对那些结构的引用。参见“嵌入对象”部分
update(x)	用字典 x 中的键/值对更新字典内容。注意字典被合并而非被连接，因为字典的键不能复制
get(x [,y])	返回键 x。若未找到该键返回 None。可用于 dict[x]。若提供 y，则未找到值 x 时返回 y

3. 字典排序

字典对象的 keys()和 values()方法返回字典所有键或值的列表。这在字典上进行迭代或分离字典值时很有用。但不能在一次调用过程中嵌套 keys()或 values()和相应的列表或 tuple 的 sort()方法。要得到排序列表，需完成两个步骤：

```
months = monthdays . keys ( )
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
months . sort ( )
```

许多用户试图使用下面的语句：

```
months = monthdays . keys ( ) .sort ( )
```

但无法得到预期效果。原因很简单—— sort 方法在适当位置修改列表，但不返回新列表。虽然已经对 keys 方法产生的对象进行了排序，但没有返回信息。语句结束后，已排序的临时列表对象消失，months 只包含特殊值 None。

用顺序方式列举字典内容时，这个方法带来的问题就明显了。必须和第一个例子一样用两步对字典排序：

```
keys = monthdays . keys ( )
keys . sort ( )
for key in keys :
...
```

如果基于值而不是键对列表进行排序，过程更加复杂。不能使用值访问字典的信息，而必须用键。需要向列表的 sort 方法提供自定义的比较函数，从而给 tuple 对的列表进行排序。

使用 items 方法可获得 tuple 对的列表。过程类似下列语句：

```
monthdays =   { 'Jan' : 31, 'Feb' : 28, 'Mar' : 31,
              'Apr' : 30, 'May' : 31, 'Jun' : 30,
              'Jul', : 31, 'Aug' : 31, 'Sep' : 30,
              'Oct' : 31, 'Nov' : 30, 'Dec' : 31 }
months = monthdays . items ( )
months . sort (lambda f, s : cmp (f[1], s[1] ) )
for month, days in months:
    print 'There are', days, 'days in', month
```

上例使用了许多技术，现在解释其工作原理。第 5 行得到作为 tuple 的键/值对列表。第 6 行通过比较每个 tuple 组件的值对列表进行排序。lambda 是个匿名函数(lambda 函数的其他信息请参阅第 4 章，自定义元素的 sort 方法的说明请参阅第 7 章) 。接下来，for 循环用于从 tuple 列表的每个单独 tuple 中提取月份及该月份的天数，然后进行打印。

执行时，脚本输出如下：

```
There are 28 days in Feb
There are 30 days in Jun
There are 30 days in Nov
There are 30 days in Apr
There are 30 days in Sep
There are 31 days in Aug
There are 31 days in May
There are 31 days in Oct
There are 31 days in Jul
There are 31 days in Jan
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
There are 31 days in Dec
There are 31 days in Mar
```

字典是最有效存储元素的方式之一，可用任何对象作键访问字典中的信息。Python 允许用任何对象作为键/值对中的键。第 10 章将深入讨论 Python 中的字典。

3.1.9　文件

和见到过的其他对象类型一样，Python 也内置了对文件的访问方法。可以用内置 open 函数创建文件对象。接着用新建对象中的方法从文件读取信息或写信息到文件中。这减少了用于 C、Perl 和 VB 中的文件句柄模形的复杂性。Python 提供了访问主机上存储形式完全不同的对象的一致接口。

现在暂不深入讨论使用文件的细节，而把它留到第 11 章讨论，那时大家将学习使用 Python 处理和管理文件的方法。为使大家得到感性认识，先观察下面的脚本，该脚本用于在显示前打开并读取文件 myfile.txt 的每一行：

```
input = open ('myfile.txt' )
for line in input . readlines ( )
    print line
input . close ( )
```

3.1.10　对象存储

虽然创建变量时 Python 总是创建对象，但重要的一点是：应记住名称、对象和变量间的联系。虽然听起来有些重复，这里还是要重申本章开始的声明：Python 将信息存储为对象。实际上 Python 存储一个内部结构的引用或指针，该结构存储特定对象类型的信息。这等价于 C 中的指针或 Perl 中的引用。Python 对象只是存储于内存中的结构的地址。

在 C 中，要在应用程序中使用特殊变量，需要用指向结构的指针或在使用数据类型前预定义结构。使用系统级的指针比较困难，且增加了不必要的复杂性。另外，程序员必须明白他们处理的变量是指针并需要进行特殊处理。

在 Perl 中，程序员使用引用时，必须把引用当作特殊数据类型对待。在访问信息前要间接引用 Perl 引用。如果不进行间接引用或间接引用错误类型(例如以标量方式访问散列)，则会引起变量的物理地址或编译错误。这个问题容易发现，但它增加了编程过程的额外复杂性。实际上可以避免这些问题。

与 Perl 和 C 不同，Python 程序员不必理会指针的存在。访问对象时实际访问其存储的信息，而不管对象如何定义或对象信息的存储位置及方式。

1. 对象、名字和变量

Python 中的 name 是提供给指向对象的指针的字母数字混合名字。注意下例：

```
greeting = 'Hello World'
```

上面的语句中，名字为 greeting。对象没有用于标识的名字。它只是指向 greeting 的一块已

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

分配内存。但对象类型是字符串。两者的组合可作为典型的变量。

指针的意义非常重要。考虑对变量重赋值时发生了什么：

```
greeting = ['Hello', 'World']
```

名字依然未变，但所指的对象在内存中处于不同物理位置，是完全不同的类型。

2. 变量命名规则

命名变量时 Python 遵循下列基本规则：

- 变量名以下划线或字母开头，可包含任何字母、数字或下划线的组合。不能使用任何其他标点符号。
- 变量名区分大小写。string 和 String 是两个不同的变量名。
- 保留字不能做变量名。下面是 Python 中的所有保留字：

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

因为 Python 区分大小写，因此可以使用 Return 和 RETURN 做变量名，虽然这样做不是好主意。

3. Python 复制引用而不是数据

如果给对象分配信息时通过名字引用 Python 对象，那么，Python 存储的是引用而不是数据。例如，下列代码看上去创建了两个对象：

```
objecta = [1, 2, 3]
objectb = objecta
```

实际上，只创建了一个对象，但创建了两个对象名字(指针)。可以交互式使用下列语句进行证明：

```
>>> objecta = [1, 2, 3]
>>> objectb = objecta
>>> objecta [1] = 4
>>> print objectb
[1, 4, 3]
```

要创建对象的拷贝，必须强制 Python 明确分配对象值，或者用简短叙述过的类型转换符。对列表和 tuple 来说，前一种方法更容易些：

```
objectb = objecta [ : ]
```

复制对象指针而非对象内容会产生一些有趣的现象。最有用的一点在于这个事实：因为复

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

制对象指针是 Python 的标准行为，因此构造复杂结构就成为自然过程而非强制过程。例如，可用对象指定用于字典的键。也可为 contacts 数据库的表构造列表，如下所示：

```
tables = { contacts : 'A list of contacts',
        addresses : 'The address list' }
```

如果改变了 contacts 的内容也没关系，因为在字典 tables 中指定的键只是 contacts 对象的指针，而不是字典 contacts 的拷贝。

4. 嵌套

因为 Python 复制对象指针，所以可以在任何地方使用对象，而不必局限于其他语言中变量的普通边界和限制。例如，大家已经看到列表可以是字符串、数字甚至其他列表的组合。下例使用的结构描述了 contacts 列表，它使用描述信息的列表和字典的组合。

```
Contacts = [ { 'Name' : 'Martin',
               'Email' : 'mc@mcwords.com' },
               { 'Name' : 'Bob',
               'Email' : 'bob@bob.com'} ]
```

本书第 3 部分包含了有关嵌套的其他示例，深入学习嵌套后，可用 Python 解决特殊问题。

3.1.11 类型转换

除了针对单个对象类型讨论过的方法和运算符外，还有一些内置函数可将数据从一种类型转化为另一种类型。表 3-11 是这类函数的完整列表。

Python 还支持一个运算符``，其工作方式与内置函数 str()相同。运算符``(反勾号)对其包含的语句求值并返回字符串表示。如下例：

```
>>> print ' ( 34*56.0 ) '
1904
>>> album = ('TMBG', 'Flood', 'Theme from Flood', 'Birdhouse in your Soul ' )
>>> album
('TMBG', 'Flood', 'Theme from Flood', 'Birdhouse in your Soul ' )
>>> 'album[ :2] '
" ('TMBG', 'Flood' )"
>>> str (album[ :2] )
" ('TMBG', 'Flood' ) "
>>> 'list (album) '
" ['TMBG', 'Flood', 'Theme from Flood', 'Birdhuse in your Soul ' ]"
```

注意，返回信息时 str 和"所用的格式和构造对象时所需的一样。因此要显示构造前面的 album 列表所需的 Python 语句，可使用如下语句：

```
contacts = [ { 'Name' : 'Martin',
               'Email' : 'mc@mcwords.com' },
               { 'Name' : 'Bob',
               'Email' : 'bob@bob.com' } ]
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
>>> >>> 'contacts'
" [ { 'Email' : 'mc@mcwords.com', 'Name', : 'Martin' }, { 'Email' : 'bob@bob.com', 'Name' : 'Bob' } ]"
```

表 3-11　内置类型转换函数

函　数	描　述
str(x)	将对象 x 翻译成字符串
list(x)	将对象序列 x 作为列表返回。例如，字符串“hello”作为['h', 'e', 'l', 'l', 'o']返回。将列表和 tuple 转换为列表
tuple(x)	将对象序列 x 作为 tuple 返回
int(x)	将字符串和数字转换为整数。注意，对浮点数进行舍位而非舍入。 int(3.6)变成 3
long(x)	将字符串和数字转换为长整数。转换方式同 int
float(x)	将字符串和数字转换为浮点数对象
complex(x,y)	用 x 作实部，y 作虚部创建复数
hex(x)	将整数或长整数转换为十六进制字符串
oct(x)	将整数或长整数转换为八进制字符串
ord(x)	返回字符 x 的 ASCII 值
chr(x)	返回 ASCII 码 x 代表的字符(以字符串返回)
min(x [, ...])	返回序列中的最小元素
max(x [, ...])	返回序列中的最大元素

3.1.12　类型比较

在 Python 中比较两个对象时，解释器在返回比较的值之前会比较对象的值。这意味着比较两个列表时，解释器会检查两个列表的内容以确定列表是否相同。其他对象的比较也是这样——进行比较时，解释器检查每个数据结构的内容。

表 3-12 列出了 Python 支持的进行类型比较的运算符。

表 3-12　比较运算符

运　算　符	描　述
x<y	小于
x<=y	小于或等于
x>y	大于
x>=y	大于或等于
x==y	比较
x!=y, x<>y	不等于
x is y	指向同一对象
x is not y	不同对象
not y	取反——x 为假则返回真，x 为真则返回假

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

运　算　符	描　　述
x or y	若 x 为真则返回 x，或者若 x 为假则返回 y
x and y	若 x 为假则返回 x，或者若 x 为真则返回 y
x<y<z	链式比较：所有运算符均为真时返回真

在表 3-12 中有些特殊情况。如果创建两个相同列表并比较它们的值时，得到的返回值为 1，表示两个列表的值相等：

```
>>> lista = [1, 2, 3]
>>> listb = [1, 2, 3]
>>> lista == listb
1
```

注意值的顺序同样重要：

```
>>> listb = [2, 3, 1]
>>> lista == listb
0
```

要比较两个对象是否相等，例如，它们均指向相同物理对象，则需用运算符 is 进行比较：

```
>>> lista is listb
0
```

需要使用 is 运算符，这是因为有两个单独对象。但是若从 lista 复制对象引用给 listb：

```
>>> listb = lista
>>> lista is listb
1
```

则得到期待的结果。

检查相对于某个源或控制对象的另一个对象的源和有效性时，is 运算符很有用。

Python 用以下规则进行比较：

- 对数字进行数值比较。
- 对字符串比较字符。
- 对列表和 tuple，按索引从低到高比较其元素。
- 对字典比较排序后的键/值对。

在 Python 中，比较结果为真则返回 1，比较失败则返回 0。另外，Python 把字符串在内的非零值作为真。惟一例外为：把 None 和任何空对象(列表、tuple 或序列)作为假。表 3-13 对不同真/假值作了总结。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

表 3-13　对象和常量的真/假值

对象/常量	值
''	假
'string'	真
0	假
>1	真
<-1	真
() (空 tuple)	假
[] (空列表)	假
{} (空字典)	假
None	假

3.2　语句

语句是 Python 程序中可执行元素的最基本形式。变量自身不会做任何事，只有作为语句的一部分进行创建、修改和操作。在这一部分大家将学习一些普通的语句类型，包括基本语句、赋值函数调用和控制及循环语句。

3.2.1　语句格式

Python 使用非常简单的方法分析组成典型程序的单独语句。Python 中的每一行标识为单个语句；通常的回车和换行的行结束标记作为语句结束标记。例如，下面两行程序是合法的：

```
print "Hello World! "
print "I am a test program"
```

对于类似下例的很长的行，根据行的内容可采用多种方法：

```
print   "Hello, I am a test program and I am printing an extra long
line so I can demonstrate how to split me up"
```

对多数行来说，比较简单的方法是在需要分离的行尾加反斜线符号，如下例：

```
print   "Hello, I am a test program and I am printing an extra \
long line so I can demonstrate how to split me up"
```

当 Python 解释器发现某行最后一个字符为反斜线符号时，将在分析整行前自动把后面一行追加到本行中。这个过程是循环的，因此只需在每行结尾加上反斜线符号，就可将一行分成所希望的任意多行：

```
print   "Hello, I am a test program \
and I am printing an extra \
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
long line so I can demonstrate \
how to split me up"
```

对于由一对圆括号合并的语句，无需使用反斜线符号。Python 自动搜寻下一行直至找到标志结束的圆括号。可以修改上例的 print 语句以打印一列消息，其中需要使用圆括号做结束字符：

```
print  ("Hello, I am a test program "
"and I am printing an extra "
"long line so I can demonstrate "
"how to split me up")
```

最后，在既没有反斜线符号又没有圆括号的情况下，Python 自动检查下一行以获取更多信息。

对于 C 和 Perl 程序员来说，如果觉得更方便，也可用分号作为行结束符号。但分号的使用是完全随意的，并且使用分号对 Python 分析行没有任何影响—— 不管是否使用分号，都要使用前面阐述的几个方法之一。

3.2.2　注释

Python 允许在代码中插入“#”号以加入注释，如下例所示：

```
Version = 1.0   # This is the current version number
```

“#”号后的所有语句均作为注释而被 Python 解释器忽略。注意，引号里面的“#”号不会被视作注释符号，这正是人们所期望的。

3.2.3　赋值

赋值是 Python 中最基本的语句：给变量名分配数据和对象类型。在学习不同的 Python 对象类型时已经看到一些这方面的不同示例。与 C 不同，但类似 Perl(非严格模式)，在给 Python 变量赋值前不必预先声明。但是，变量在用于表达式或其他形式的语句中时必须已经存在。

有 4 种基本的赋值方法：基本赋值、tuple 赋值、列表赋值和多目标赋值。

1. 基本赋值

本章的多数示例均使用基本赋值法。采用基本赋值法时会根据分配给对象的值使用正确类型创建新对象。接着把指定名字指向新对象，如下例：

```
number = 45
message = 'Hello World'
mylist = ['I Palindrome I', 'Mammal']
mytuple = ('Mon', 'Tue', 'Wed', 'Thu', 'Fri')
mydict = { 'Twisting': Flood, 'Mammal' : 'Apollo 18'}
```

2. tuple 和列表赋值

可用不同的赋值运算符从 tuple 或列表中提取单个元素。对于 tuple，只需指定待赋值的变量的列表，每个变量间用逗号隔开：

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
title, name = 'Mr', 'Martin'
album = ('TMBG', 'Flood', 'Theme from Flood', 'Birdhouse in your Soul')
artist, title = album[0 : 2]
track1, track2 = album[-2 :]
```

在前面的语句中，实际上创建了赋值运算符左边所示的新 tuple 对象。但是不同变量名的 tuple 对象是匿名的，而且不记录整个 tuple 的信息。因为不必使用圆括号说明 tuple，所以赋值语句显得更自然。当然也可用下列表示方法：

```
album = ('TMBG', 'Flood', 'Theme from Flood', 'Birdhouse in your Soul')
(artist, title) = album[0 : 2]
(track1, track2) = album[-2 :]
```

每种情况下均以指向赋值符右边所示信息的新变量结束。注意：也可用分片和索引的方法从 tuple 中提取多个元素。

对于列表，必须用创建匿名 tuple 的方法创建一个新的匿名列表：

```
album = ['TMBG', 'Lincoln', 'Ana Ng', 'Cowtown' ]
[artist, title]   = album[0 : 2]
```

tuple 和列表赋值还有其他的方法。因为名字只是指针，可以通过执行下面的 tuple 或列表赋值来交换两个对象/名字的组合：

```
artist, title = title, artist
```

在下面的交互式会话中可看得更清楚：

```
>>> title = 'Flood'
>>> artist = 'TMBG'
>>> artist, title = title, artist
>>> artist, title
('Flood', 'TMBG')
```

3. 多目标赋值

使用多目标赋值语句并用多个指针来创建单个对象：

```
group = title = 'They Might Be Giants'
```

在前面的语句中，用两个不同的指针 group 和 title 创建单个字符串对象。记住，因为两个指针指向相同对象，所以用一个名字修改对象内容也会改变用另一名字获取的信息。其效果和下面的语句相同：

```
group = 'They Might Be Giants'
title = group
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

3.2.4 打印

与其他语言不同，Python 与用户通信的默认方法是通过语句而不是函数。语句有相同的基本名字(print)，但被嵌入到解释器中。而与 print 函数不同的是：Python 中的 print 语句实际输出字符串，该字符串代表提供给解释器的标准输出的对象。目的文件和 C 的 stdout 或 Perl 的 STDOUT 文件句柄一样，而且一般不能指向其他地方。

提示：

实际上，可修改指向标准输出设备的对象来重定向标准输出。实际上 Python 使用对象与外部进行通信。这并不奇怪。下面的代码显示了这一点：

```
import sys
fp = open('somefile.txt', 'w' )
sys.stdout = fp
print ' this goes in the file, not on your stdout'
```

因为没有把标准输出指向原始路径，而是把对象数据复制到新的 fp 对象，因此上式有效。现在由 print 函数使用的 stdout 对象将所有输出发送到 somefile.txt 文件。

打印时，语句遵循这些基本规则：

- 打印时，Python 用空格分隔由逗号分隔的对象和常量。要避免这种现象，可采用字符串连接或%格式化运算符。
- Python 向每个输出行追加换行符。要避免这种现象，可在每行行尾添加逗号。

例如，执行下列脚本将显示 Python 支持的不同格式：

```
print 'Hello', 'World'
print 'This is a' +' concatenated', string
print 'The first line',
print 'Plus some continuation'
print 'I ordered %d dozen %s today' % (6, 'eggs')
contacts = [ { 'Name' : 'Martin',.
'Email' : 'mc@mcwords.com'},
            { 'Name' : 'Bob',
            'Email' : 'bob@bob.com'}]
print contacts
```

上面的脚本产生下列输出：

```
Hello World
This is a concatenated string
The first line Plus some continuation
I ordered 6 dozen eggs today
[{'Email': 'mc@mcwords.com', 'Name': 'Martin'}, {'Email': 'bob@bob.com',
'Name': 'Bob'}]
```

注意，在最后一行，print 显示复合对象的格式与 str 和``运算符一样。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

这种打印对象内容的的方法非常有用，既可作为编程工具，也可作为调试工具。例如，可以在一个或多个变量里存储应用程序的配置参数。如果要保存信息，只需用 print 或 str 函数写出语句。要取回信息，所需做的只是用其他模块中所用的方法输入文件内容。

3.2.5　控制语句

Python 根据程序规定的顺序处理程序语句，除非使用控制语句或函数调用。Python 支持三种不同的控制语句：if、for 和 while。if 语句是最基本的控制语句。它根据一个和多个表达式的结果选择执行语句块。for 和 while 均为循环语句。

1. if 语句

if 语句接受表达式。如果返回真，则执行指定语句。if 语句的一般格式为：

```
if EXPRESSION:
    BLOCK
elif EXPRESSION2:
    BLOCK2
else:
    BLOCK3
```

EXPRESSION 是需要执行的测试条件(参见表 3-12)；若返回真，则执行 BLOCK 中的语句。当 EXPRESSION 返回值为假时，可选的 elif 语句执行进一步的测试；若 EXPRESSION2 返回真，则执行 BLOCK2。否则执行 BLOCK3。在单个 if 语句中可使用多个 elif 语句。如果没有匹配的表达式，则执行 else 语句的可选块。

注意每种情况下，语句块前面的行均以冒号结尾。这会告诉 Python：下面是新的语句块。与许多其他语言不同，Python 对代码块定义的要求稍微宽松。C 和 Perl 使用花括号定义代码块的开始和结束。Python 使用缩排的方式：只要愿意，可把冒号后任何行都当作相同的逻辑代码块。

图 3-2 显示了缩排和块的工作机制。注意，在该图中，Block 2 和 Block 3 结束，而 Block 1 还继续。详细内容请参阅块定义和缩排脚本的其余部分。

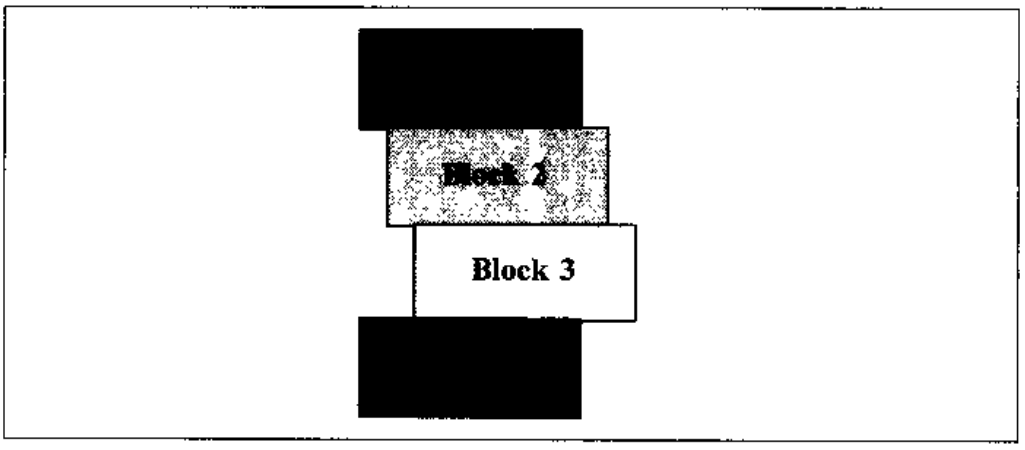

图 3-2　Python 中的块缩排

考虑下面的示例，该示例组合应用了 if 和 elif 语句：

```
if (result == 1):
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
        print 'Got a one'
    elif (result >1):
        print 'Got more than one'
    else:
        print 'Got something else'
```

在简单测试中，可以把语句和块组合到单行中：

```
if (1): print 'Hello World'
```

这实际上作为单行块(single-line block)。

注意：

如果要在 Python 中执行 swith 或 case 语句，必须使用带有多重 elif 表达式的 if 语句。

2. While

while 循环接受单个表达式，并且只要测试结果为真，则循环持续执行。在每次循环前对表达式重新求值。基本格式为：

```
while EXPRESSION:
    BLOCK
else:
    BLOCK
```

例如，要浏览字符串中的字符，可使用：

```
string = 'I Palindrome I'
while(len(string)):
    char = string[0]
    string = string[1:]
    print 'Give me a',char
```

输出为：

```
Give me a I
Give me a
…
Give me a
Give me a I
```

注意，在这个示例中，使用 for 循环可能更简单。

在循环退出时，如果 EXPRESSION 返回假，则执行可选的 else 块。如果执行过程被 break 语句中断则不执行 else 块。请参阅下面的 for 循环部分所列举的示例。

3. for

for 循环等价于 Perl 中 for 循环的列表形式。for 循环的基本格式为：

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
for TARGET in OBJECT:
    BLOCK
else:
    BLOCK
```

在循环中指定迭代的对象，接着提供任何形式的序列对象来迭代。例如：

```
for number in [1,2,3,4,5,6,7,8,9]:
    print 'I can count to',number
```

每次在循环中进行迭代操作时，都将 number 设置为数组中的下一个值。产生输出：

```
I can count to 1
I can count to 2
...
I can count to 8
I can count to 9
```

因为 for 可使用任何序列，所以也可采用字符串，如下例：

```
for letter in 'Martin':
print 'Give me a', letter
```

输出为：

```
Give me a M
Give me a a
Give me a r
Give me a t
Give me a i
Give me a n
```

另外，类似于 while 循环，在循环正常退出时执行 else 语句块，如下例：

```
for number in [1,3,5,7]:
    if number > 8:
            print "I Can't work with numbers that are higher than 8!"
            break
        print '%d squared is %d' % (number, pow(number,2))
else:
        print 'Made it!'
```

输出为：

```
1 squared is 1
3 squared is 9
5 squared is 25
7 squared is 49
Made it!
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

4. 范围

因为 for 循环不支持 Perl 和 C 中的循环计数格式，因此需要使用不同方法指定某个数字范围内的迭代。有两种可能的方法。其一是使用 while 循环：

```
while(x <10):
    print x
    x = x+1
```

另一种是使用 range 函数产生值列表：

```
for x in range(10):
...
```

range 函数返回列表。该函数的基本格式为：

```
range([start, ] stop [, step])
```

若只带有一个参数，则 range 返回由小于该参数的值组成的列表(列表中的元素不包括参数本身)：

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

若有两个参数，则 range 函数返回两个参数之间的值组成的列表(包括第一个参数，但不包括第二个参数)：

```
>>> range(4,9)
[4, 5, 6, 7, 8]
```

最后的格式为使用三个参数，用于定义步长：

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

因为 range 函数返回一个列表，所以可用该函数创建需要很大内存来存储的非常大范围的非常大的列表。要达到此目的，也可使用 xrange 函数。xrange 函数具有相同格式，也执行和 range 一样的功能。但 xrange 不能创建间隔型列表，因此对于大型列表它需要的内存少。但是另一方面，它产生一个很大的列表，因此当创建新对象并加到列表结构中的时候，会引起一些不必要的处理过程。

5. 循环控制语句

Python 支持 3 种循环控制语句用来修改循环的一般执行过程。它们是 break、continue 和 pass。

break 语句退出当前循环，并忽略任何 else 语句，而继续执行循环语句块的最后一行后面的行。

continue 语句强制循环立即进入下一次循环，忽略当前块中的其他语句。对循环语句表达示重新求值。例如：

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
x=20
while (x):
    x = x-1
    if not x % 3: continue
    print x,'is not a multiple of 3'
```

产生下列输出：

```
19 is not a multiple of 3
17 is not a multiple of 3
16 is not a multiple of 3
...
4 is not a multiple of 3
2 is not a multiple of 3
1 is not a multiple of 3
```

pass 是空操作语句——它不做任何事情。语句：

```
if (1) :  pass
```

不做任何操作。

虽然 pass 语句表面上没有意义，但在需要识别但忽略特定事件时则需使用 pass 语句。对于 Python 的错误处理系统(异常)而言，更是如此。第 5 章将讨论 pass 和异常的应用。

3.2.6　普通陷阱

对于转向 Python 的程序员，有许多不易察觉的普通陷阱。其中多数陷阱与 Python 和其他语言的不同点有关，尤其是在曾使用 Perl 和 Shellscript 的情况下。

1. 变量名

与其他语言不同的是，Python 变量遵循下列基本规则：

- 使用前不需事先声明变量；Python 根据分配的值设置对象类型。
- 变量被赋予简单的字母与数字混排的名字。
- 不需用特殊字符对变量进行限定。
- 有些变量有操作对象内容的内嵌方法。

2. 块和缩排

块以前面一行的冒号开始。例如：

```
if (1):
    print 'I am a new block'
```

在创建新的块以前，或者出现标志块结束的缩排格式以前，该块一直继续。

当用 tab 或空格缩进时，不要混合使用这两种方法，因为这可能使 Python 解释器和维护代码的人感到迷惑。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

3. 方法调用

调用对象的方法时最常见的错误是：假设该调用返回信息。语句：

```
sorted = list . sort ()
```

对 list 对象进行排序，但却将 sorted 的值设为特殊值 None。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第4章 函 数

任何编程语言都有一个基本特征：在程序中经常出现重复的执行语句。虽然可以将一块代码剪切并粘贴到另一块代码中，但是这样做比较凌乱。需要更新程序的特定部分时会发生什么情况呢？程序员必须逐个修改每个重复部分。对于非常小的程序，可能不会产生明显问题。但是随着程序规模和复杂度的增加,即使修改和更新相同代码，程序员花费的时间也将大量增加。

重复也增加了引入额外的语法错误、逻辑错误和印刷错误的危险。想像一下，如果复制了包含简单打字错误的序列，那么就必须追踪每个实例以排除这个问题!所有这些问题都可用函数解决。通过把重复的代码放进函数内部,可以分离代码序列，而且提供比较容易的访问方法以优化代码或分离程序 bug。

把重复代码置于一个函数之内的方法称为抽象(abstraction)。一般来说，某种层次的抽象总是有用处的——加快了程序编制过程，减少了引入错误的危险，而且使维护复杂程序更加容易。对于空间意识强的程序员，使用函数可以减少代码行数和程序执行时程序占用的内存。

大家应该已经了解Python 有三种层次的抽象:

(1) 程序可分成多个模块。

(2) 每个模块包含多条语句。

(3) 每条语句对对象进行操作。

函数大致处于第2层——它们提供一种方法，用于将重复的语句序列合并放到一个位置。

Python 也为代码重用提供了最好的系统之一。一旦人们编写了 Python 函数，就可立即在其他程序中应用该函数。惟一要做的是从源文件输入这个函数，而不必和 C/C++一样要创建特定库和头文件，也不必和 Perl 一样要创建特殊模块。在 Python 中，代码重用就像分辨函数存储在哪个文件中一样简单。

在函数实现方面，Python 除了具有其他的语言提供的一般功能外,还允许对提供给函数的参数及其处理方式进行高层控制和整合。同时，调用函数的返回值也很灵活——不必像 C 或 C++一样限制使用特定数据类型。

Python 中的函数构成了方法的基础，函数操作对象和对象类并执行特定动作。关于代码重用的对象类创建和模块使用,将在第 5、6 章中介绍。现在集中讨论函数的基本机制和单个 Python 模块文件中使用函数的方法。

4.1 函数定义与执行

创建新函数的一般格式如下:

```
def NAME(ARG1 [, ...]):
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
    STATEMENT BLOCK
    [return VALUE]
```

NAME 应该和对象名一样遵循相同规则:

- 函数名必须以下划线或字母开头，可以包含任何字母、数字或下划线的组合。但不能使用任何其他标点符号。
- 函数名区分大小写—— combine 和 Combine 是两个不同的函数名。 然而,应该避免创建仅仅大小写不同的相同的函数名。
- 不能使用保留字作为函数名，不过，应记住上面提到的区分大小写的规则。

定义函数时，即使函数不接受任何参数，也一定要使用括号。如果函数接受参数,则参数必须在括号里面命名(命名方法不同于其他语言)。本书将会简要地讨论参数传递机制。

用作控制语句的函数语句块前必须有冒号。结束语句return是可选的；如果不使用return语句,那么函数不会向调用者返回值。没有return语句的Python函数总是返回特殊值None。

在下列的示例中，函数将根据提供的半径计算圆的面积：

```
def area(radius):
    area = 3.141*(pow(radius,2))
    return area
```

注意：

上例给出的 π 值是一个近似值。 更精确值可在 Python 数学库中找到，表示为 pi。

要使用area函数，只需用参数进行调用:

```
print area(2)
```

返回值 12.564。 注意，返回值的精确度和所提供的 π 值的精确度相同。如果提供更高精度的 π 值，则返回值的精度也更高。语句：

```
print area (3.141592654)
```

返回 31.0004274319。

调用函数时，必须使用括号；Python 不允许用下列形式调用函数：

```
area 2
```

这是 Perl 程序员常犯的错误，他们经常不使用括号以使程序更易读。Print是语句而非函数，可以不用括号而调用print。这使得情况更加复杂。如果使用括号，则print将输出 tuple!

最后，与其他语言不同的是：Python中的函数可以置空定义——可以在正常执行的程序的任何地方创建函数。下面的代码完全有效:

```
if (message):
    def function():
      print "Hello!\n";
else:
    def function():
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
print "Goodbye!\n";
```

这样就产生了许多可能性,尤其当这种功能用于对象和方法的开发时。

在Python中，作用域、参数定义、返回值和函数调用的方法都具有扩展功能，下面将逐个描述。

4.2 作用域

Python 使用名称空间(namespaces) 的概念在应用程序中储存关于对象的信息和位置。名称空间依赖于特定级别的抽象——函数和模块有单独的名称空间，内置函数、语句和对象有特殊的名称空间。在每个级别上访问对象或函数时,Python 都将搜寻名称空间表以获得可引用的特定名字，并确定处理存储在该位置的信息的方法。在 Python 中，作用域遵循以下基本规则:

- 每个模块都有自己的作用域。这意味着多重模块有它们自己的作用域, 因此形成自己的名称空间。 第 5 章将讨论这方面的细节。
- 函数中封装的模块称为全局作用域(global scope)。这是在模块文件的最高级别中创建的对象的位置。
- 定义新函数时创建新的作用域；作用域仅限于该函数内。
- 除非特别声明，否则新对象都被赋予局部作用域。除非用关键字 global 声明的对象为全局类型，其他任何变量或函数都是局部作用域的成员。
- 函数的参数在局部函数作用域内定义。

作用域规则意味着在函数中创建的对象不会影响模块(全局类型)或内置名称空间中的命名对象。Area 就是个很好的例子：

```
def area(radius):
    area = 3.141*(pow(radius,2))
    return area
```

因为area函数定义在最高级别的模块中，所以它是全局的。area变量在area函数的局部作用域里定义。这意味着虽然它们有相同的名字，但是不会互相影响。如果想要创建递归函数,很明显必须为变量使用不同的名字，因为Python标识函数前就能发现area变量，并引发异常。(异常的细节请参阅第 7 章)。

4.2.1 创建全局对象

有时，需要在函数中对变量赋值，但是希望修改全局变量而不是创建新的变量。要做到这一点一般的方法是：在调用函数前预先声明变量：

```
name = 'Unknown'
def set_defaults():
name = 'Martin'
set_defaults()
print name
```

但在 Python 中，这样做是无效的。因为在函数中对 name 变量赋值时，Python 将在局部作

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

用域内创建新变量。要避免这种情况，需要使用 global 关键字来定义采用这种方法的变量。关键字 global 通知 Python 给全局变量创建局部别名。例如，如果修改前面的脚本如下：

```
name = 'Unknown'
def set_defaults():
global name
name = 'Martin'
set_defaults()
print name
```

这样该函数就如所期待地一样工作。打印出名字“Martin”。

关键字 global 也在全局名称空间创建以前不存在的对象，如下例：

```
def set_defaults():
    global name,address
    name = 'Martin'
    address = 'mc@mcwords.com'
set_defaults()
print "Name:",name,"Address:",address
```

执行前面一段代码时，产生如下输出：

```
Name: Martin Address: mc@mcwords.com
```

4.2.2　LGB 规则

LGB 规则是记忆 Python 如何在作用域里寻找名字的简易方法：

- 名字引用依次搜寻 3 种作用域：局部(Local)、全局(Global)、内置(Built-in)(LGB)。
- 在局部作用域内对名字赋值时会创建新对象或更新对象。如果想在局部作用域对全局对象赋值，必须使用关键字 global。
- 全局声明会把赋值名字映射到封装的模块的作用域。这意味着：要么显示地从输入的模块中输入对象，要么使用完全有效的模块/对象名。

学习了第 5 章关于模块机制部分的内容后，就会理解上面列举的最后一条。但要注意，所谓的全局定义只是相对于当前模块而言。

考虑下列代码段：

```
radius = 2
pi = 3.141592654
def area(radius):
area = pi *(pow(radius,2))
return area
print area(4)
```

该例中，定义了两个全局变量：radius 和 pi。执行 area 函数时，该函数会寻找名字 radius，并在局部范围中找到 radius，而忽略全局变量。而在局部范围不能找到 pi 变量，因此 Python 在

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

全局范围寻找并找到 pi。

在图 4-1 中可更清晰地看到 LGB 规则的布局

警告：

细心的读者可能已经注意到 LGB 规则意味着局部变量可以重写全局变量，局部变量和全局变量都可重写内置变量。例如，局部变量可称为 open，但这样会使相同作用域的函数 open 无法调用，除非显式识别。因此使用与全局或内置变量同名的局部变量不是好主意。见图 4-1。

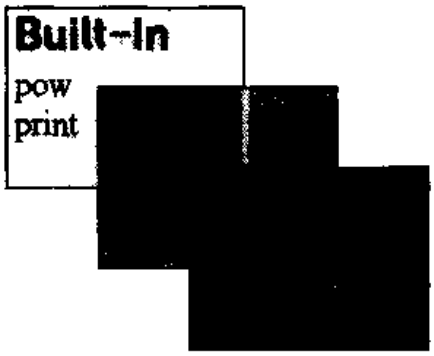

图 4-1　LGB 规则

4.2.3　陷阱作用域

很多人注意到一个重要的陷阱。名字解析过程只搜索三个抽象层——局部、全局和内置(请参阅前面的“LGB 规则”部分)。在 Python 中，可以在一个函数内部定义另外的函数。如果需要创建一组外部无法公共使用的函数，这项功能就会很有用。问题是作用域规则依然适用。考虑下面的示例：

```
def funca():
    value = 59
    funcb()
    def funcb():
      print (value*2)
```

上面的代码不能正确执行，因为 funcb 试图访问变量 value。但是 value 在局部、全局或内置作用域均不存在——即违反了 LGB 规则。函数不会自动继承父作用域，Python 名称空间管理器也不会搜寻函数的父作用域。

传递 value 的方法是使 value 作为嵌入函数的参数：

```
def funca():
    value = 59
    funcb()
    def funcb(value):
      print (value*2)
```

对于 funcb 作用域来说，value 还是局部变量，因此可以使用相同的名字，而且 LBG 规则有效。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

4.3　参数

在函数定义中指定的参数按照其定义的顺序依次被接收，例如：

```
def message(to, text):
    print "This is a message for",to,":\n",text
message('Martin','Your dinner is ready!')
```

通过赋值的方式将参数传递给函数。这种赋值规则也适用于函数定义里包含的参数。因此参数传递过程遵循如下规则：

- 通过引用将参数复制到局部作用域的对象中。这意味着被用来访问函数参数的变量与提供给函数的对象无关。而且修改局部对象不会改变原始参数。
- 可以在适当位置修改可变对象。当人们复制列表或字典时，就复制了对象列表的引用。如果改变引用的值，则修改了原始参数。

第一条规则意味着可以安全地使用局部类型对象，而不必担心会改变原始值。例如，代码：

```
def modifier(number, string, list):
number = 5
string = 'Goodbye'
list = [4,5,6]
print "Inside:", number, string, list

number = 1
string = 'Hello'
list = [1,2,3]

print "Before:", number, string, list
modifier(number,string,list)
print "After:", number, string, list
```

产生下列输出：

```
Before: 1 Hello [1, 2, 3]
Inside: 5 Goodbye [4, 5, 6]
After: 1 Hello [1, 2, 3]
```

局部类型的 number、string 和 list 变量已经修改，但是最初的变量保持原始值不变。

第二条规则说明任何可变对象都可在适当位置修改。这允许改变指向可变列表或字典的特定元素的信息。考虑下面的例子：

```
def modifier(list):
list[0:3] = [4,5,6]
print "Inside:", list
list = [1,2,3]
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
print "Before:", list
modifier(list)
print "After:", list
```

执行这段代码，将产生下列输出：

```
Before: [1, 2, 3]
Inside: [4, 5, 6]
After: [4, 5, 6]
```

要复制列表或字典对象而不是复制引用，必须显式地使用复制操作。在下例中，将把下面的代码行插入 modifier 函数并作为该函数的第一行：

```
list = list [ : ]
```

执行脚本，得到：

```
Before: [1, 2, 3]
Inside: [4, 5, 6]
After: [1, 2, 3]
```

4.3.1　参数是对象

虽然现在这一点看上去很明显，但参数只是用于对象的名字——对象类型的问题依然相同。另外，因为参数只是个指针，所以无需定义传递的对象类型。例如：

```
def multiply (x, y)
  return x * y
```

使用数字对象时，返回数字：

```
>>> multiply (2, 3)
6
```

或者，也可返回字符串：

```
>>> multiply ('Ha', 5)
'HaHaHaHaHa'
```

还可返回列表：

```
>>> multiply([1,2,3],5)
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

但应注意：当序列的对象类型不是数字序列时，将会引发异常。原因是类型不兼容(请参阅第 7 章中的异常部分)。通过使用对象而非严格变量类型，可以创建独立的函数，他们接受不同类型对象作为参数，从而完成许多不同任务。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

4.3.2　关键字的参数调用

参数调用的一般顺序为：按照函数定义参数的顺序为每个参数赋值。除此之外，Python 还支持许多其他函数/参数调用格式。传统的顺序赋值方法的问题是：只能按特定顺序接受特定数量的参数。在可读性方面，这种技术也不是永远实用。在为函数提供信息的同时，如果又可用来完成许多不同任务，而不必和 C，C++或 Visual Basic 那样支持单一功能的函数，那么这种灵活性则很有用。

除了顺序赋值外，最基本的扩展为用关键字为指定参数赋值。例如，可以这样调用 multiply 函数：

```
multiply (x=5, y=10)
```

这时，参数顺序并不重要，也可这样：

```
multiply (y=10, x=5)
```

4.3.3　默认参数

在定义函数时，Python 允许为每个参数指定默认值。通常，必须提供与函数定义中相同数量的参数。使用默认值后，可以省略参数而调用函数，而函数还会执行。定义默认值的格式为：

```
def function (ARG=VALUE)
```

未提供参数时使用 VALUE 做默认值。

例如，创建计算锥体体积的函数。函数定义如下：

```
def cone_volume(height,radius=1,pi=3.141592654):
    cone = (1.0)/(3.0)*height*pi*pow(radius,2)
    return cone
```

现在，可以用下列任意方式调用该函数：

```
cone_volume(4,3,(22.0/7.0))
cone_volume(4,3)
cone_volume(4)
cone_volume(radius=3, height=4)
cone_volume(pi = (22.0/7.0),
radius=4,
height=4)
```

注意，不能省略 height 的值——该参数没有默认值，因此 Python 中必须为它赋值。

另外应注意，理想中默认值应该是定义中的最后几个参数。虽然下列定义也有效：

```
def cone_volume(radius=1,pi=3.141592654,height):
```

如果只用一个参数调用该函数，编译器将引发异常。因为 Python 不能决定所提供的值应该应用于哪个参数：

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
>>> cone_volume(4)
Traceback (innermost last):
File "<stdin>", line 1, in ?
File "<stdin>", line 2, in cone_volume
TypeError: number coercion failed
```

注意：

因为传递给 height 参数的值为 None，因此得到 TypeError 异常。这意味着没有提供足够的参数值。当 Python 进行计算时，它试图将所提供的值转换为数字，但是不能将 None 的值转换为数字，因此出现类型不匹配。

4.3.4　参数 Tuples

若要接受变长参数，如果不想使用默认值，可采用 tuple 参数：

```
def function(* ARG)
```

所有参数均置于 ARG 指定的 tuple 中。例如，下面的示例接受参数的变量列表，这些参数放在第一个参数定义的列表中：

```
def multi_insert(list, *items):
  for item in list:
    list.append(item)
```

参数 tuples 挑出函数调用中的所有剩下还未定位的参数。同样，这些参数应该最后指定。很明显，在函数定义中只能使用一个参数 tuple。

4.3.5　参数字典

参数字典和关键字参数传递方式相似。例外情况是：参数字典不是为单独变量赋值，而是为每个关键字/值对在指定字典中创建一个条目：

```
def function(** ARG)
```

例如，函数：

```
def message(text, **header):
for field in data.keys():
print field+":",data[field]
print
print text
```

经调用后可创建标准电子邮件消息：

```
message('I pulled the chain but nothing happened!',
        From='mc@mcwords.com',
        To='mc@mcwords.com',
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
    Subject='Hello')
```

调用该函数产生下列输出：

```
Subject: Hello
To: mc@mcwords.com
From: mc@mcwords.com

I pulled the chain but nothing happened!
```

类似于 tuple 参数，字典参数必须在任何连续或默认参数后作为最后一个参数。也可组合使用 tuple 和字典参数，如修改后的 message 函数：

```
def message(text, *lines, **header):
    for field in header.keys():
        print field+":",header[field]
    print
    print text
    for line in lines:
        print line

message('I pulled the chain but nothing happened!',
        'Another line',
        'Second line',
        From='mc@mcwords.com',
        To='mc@mcwords.com',
        Subject='Hello')
```

调用该函数产生多行电子邮件消息：

```
Subject: Hello
To: mc@mcwords.com
From: mc@mcwords.com

I pulled the chain but nothing happened!
Another line
Second line
```

4.3.6　函数规则

Python 严格地规定如何分析函数调用和定义。定义函数时，Python 遵循如下规则：

- 默认值必须在非默认参数之后。
- 在单个函数定义中只能使用一个 tuple 参数(* arg)和一个字典参数(** arg)。
- tuple 参数(* arg)必须在连续参数和默认参数之后。
- 字典参数必须是最后的参数。

调用函数时，Python 遵循如下规则：

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

- 函数调用必须包括圆括号。func 和 func()不同。func 访问函数引用，而 func()则执行函数。
- 关键字参数必须出现在所有非关键字参数之后。

另外，Python 确定函数中参数的赋值时，采用下列步骤：

- 按顺序为非关键字参数赋值。
- 按参数名为关键字参数赋值。
- 将额外非关键字参数赋予 tuple 参数。
- 将额外关键字参数赋予字典参数。
- 将默认值赋予任何未赋值的参数。

4.4　返回值

return 语句返回调用程序指定的对象。基本格式为：

```
return OBJECT [, OBJECT, ...]
```

return 语句不限于返回单个对象；它能返回 tuple、列表或字典。可能 tuple 最有用，因为它可以把多个对象赋予返回值。

例如，这里有一个转换函数，它接受两个参数并返回第一个参数的平方和第二个参数的立方：

```
def sqsq(first,second):
    first = first*first
    second = second**3
```

现在可以调用 sqsq 函数并立即得到结果：

```
squared, cubed = sqsq(2,3)
```

其他信息请参阅第 3 章关于“Tuple 和列表赋值”的讨论。

4.5　高级函数调用

Python 除了支持前面叙述的基本函数功能外，还支持一些高级函数处理。在事先不知道函数名和要应用到函数的参数时，如果要动态调用该函数，需要使用 apply 语句。Python 中的 map 语句和 Perl 中的 map 函数具有相同功能。map 语句允许将相同函数应用于任意列表。

最后两个特征是联系在一起的。毫无疑问，函数也是 Python 对象。可以将函数赋给变量名，接着间接地动态调用这些函数。作为这种功能的扩展，可以创建变量名，该变量名指向无实际名字的函数——匿名函数(用 lambda 语句创建匿名函数)。

4.5.1　apply 语句

有时需要在应用程序中调用函数，却不知道函数名和要应用到该函数的参数数量。使用

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

apply 函数后，可将多个参数应用于函数，执行该函数时就如同显式调用该函数一样。apply 语句的格式为：

```
apply(FUNCTION, TUPLE)
```

下面是调用 apply 函数的示例：

```
apply(multiple, (2, 4) )
```

这样调用的好处来源于使用 tuple 作为参数。而 tuple 是一种数据结构，因此可以在运行时计算该函数调用，而不用开发时计算。本书还有许多 apply 语句的例子。

4.5.2　map 语句

想像一下，现在有一个数字列表，并希望把该列表转化为这些数字的立方的列表。普通的过程可能为：

```
for index in range(len(list)):
    list[index] = pow(list[index],3)
```

这个简单函数容易理解。但是对于复杂函数，必须使用循环为列表的每一项手工嵌入函数调用，这会产生不必要的负担。

比较简单的方法是使用 map 语句。map 语句接受函数名和该函数使用的列表或 tuple。因此，可以将前面的代码段重写为：

```
def cube(x):
    return pow(x,3)
map(cube, list)
```

虽然上述代码没有变得更短，但是更加实用。map 函数替程序员完成循环操作和重赋值操作。可以用任何函数作为 map 语句的第一个参数。更常见的情况是：map 语句与 lambda 语句一起使用。lambda 可创建匿名函数。前面的语句可重写为：

```
map(lambda x:pow(x,3), list)
```

本章稍后部分叙述如何创建匿名函数。其他细节请参阅“匿名函数”部分。

这两种调用命名过的函数和使用匿名函数的格式与 Perl 使用表达式和块的格式在语法上很相似。Perl 语句：

```
map EXPR, LIST
```

大体上和下面的 Python 语句等价：

```
map (FUNC, LIST)
```

同样，Perl 语句：

```
map BLOCK LIST
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

大体上和下面的 Python 语句等价：

```
map ( lambda ARG: EXPRESSION, LIST)
```

4.5.3　间接函数调用

函数是一种 Python 对象。因此可将函数赋值给变量名，并作为任何其他对象传递给其他函数。例如，可以和下例一样创建函数别名：

```
def hello():
    print "Hello World"
x = hello
x()
```

注意上例用到的符号。示例中将对象 hello 赋值给名为 x 的变量。用对象引用 hello 指定该函数，而没有用 hello()调用该函数。这样就复制了对函数的引用，而不是执行该函数并用结果赋值。

通过使用函数引用，可以将函数对象作为参数提供给另外的函数：

```
def call(function, *args):
    function(args)
call(multiply,2,2)
```

最后，使用相同方法，可以将函数对象置于其他数据结构中：

```
obj_dict = { 'function' : multiply, 'args' : (2,2) }
apply(obj_dict['function'],obj_dict['args'])
```

注意上例中 apply 的用法。apply 接受第一个参数作为函数引用，并接受 tuple 作为参数，这些参数是被应用到函数的参数。

4.5.4　匿名函数

匿名函数是使 Perl 和 Python 等语言具有灵活性的关键组件。本书已经演示了几个 Python 匿名函数的例子。在 C/C++中，与匿名函数最相似的是内联函数。在内联函数中，可以像真正的函数调用一样定义要执行的块或表达式，而不必做与预声明的典型函数相关的事情。

在 Python 中创建匿名函数需要使用 lambda 表达式。因为 lambda 是表达式，而不是语句，因此，可以在不能使用 def 语句的地方调用 lambda 表达式。另外，lambda 表达式返回函数对象，但不提供函数名。返回对象的用法类似间接函数调用。

lambda 表达式的一般格式为：

```
lambda ARG [, ARG, ...]: EXPR
```

ARG 是参数，它类似于典型的 def 语句。ARG 参数工作机理类似于普通函数的连续参数。在 lambda 定义中，可使用默认值，但不能使用其他参数特征。

EXPR 是个调用函数时执行的单行表达式。创建和使用 lambda 函数时，需要记住下面列出

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

的几个 lambda 函数与普通函数的重要区别：

- 用 lambda 创建的函数只接受单个表达式——不能使用多行匿名函数。
- 在 lambda 函数中，表达式的结果总是返回给调用者。

例如，可以使用 lambda 创建匿名函数，该函数计算数字的立方，并把结果赋值给变量名：

```
f = lambda x: x*x*x
f(2)
```

前面的语句实际上采用了典型的 def 语句的工作方式。差别在于：在 lambda 中，赋值给变量名的过程是自动完成的。

lambda 表达式的用法与 Perl 中的 sub{}匿名函数语句的用法相同，但是 Python 只是给该过程提供单独变量名，而不是把该过程组合到单个语句中。本质上，lambda 表达式与 def 语句没有差别。

lambda 函数多用于类似排序和在创建用户接口时的内联函数。本书有很多这方面的例子。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第5章 模 块

随着程序规模越来越大，如何把一个大型文件分割成几个小文件的问题也变得越来越重要了。代码看起来更优美、更紧凑，且易于修改。尤其是许多成员同时开发一个项目的时候，上述优点的重要性就更明显。

在 Python 中，可以把程序分割成许多单个文件，这些单个的文件称为模块。接着可以将模块输入到现存脚本中。对模块的访问可通过访问模块内定义的函数、对象和类进行。在 Python 中，模块的概念和 Perl 中的模块相似。也类似于 Visual Basic、C/C++及许多其他语言中的库和类的定义。

本章讨论如何从标准库和自建库中输入模块。还解释了创建模块包的方法。模块包是一套嵌套的模块组。根据系统配置，可以装载模块包中的单个元素，也可将模块包作为一个完整包进行装载。

5.1 输入模块

在 Python 中向脚本输入模块时，实际上输入的是包含在模块中的使用函数、对象和类的信息，而不是输入各种函数、对象和类的实体本身。理解了输入模块的不同方法和使用 import 语句时发生的情况以后，就会更清楚上句话的含义了。

5.1.1 输入完整模块

输入模块也就是给出一种方法以访问模块提供的函数、对象和类。使用 import 语句输入模块：

```
import ftplib
```

Python 中的 import 语句完成下面 3 件事：

- 创建新的名称空间(namespace)。该名称空间拥有给定模块中定义的所有对象。
- 执行该模块中限定在给定名称空间里的代码。
- 在调用者中创建引用模块名称空间的变量名。

上例中创建了新名称空间 ftplib。在 Python 库所在目录可找到 ftplib.py 文件。接着，在名称空间 ftplib 中执行 ftplib.py 文件。执行结果为：在当前模块(此时为主脚本)的名称空间里创建对象 ftplib。要获取有关 Python 如何搜寻模块的信息，请参阅本章后面的“模块装载和编译”部分。

同时也可使用下述方法输入多个模块：在 import 语句中用逗号分隔每个模块名。

```
import os, sys, getopt, ftplib
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

Python 对每个模块名逐个解释，因此上例与下面的语句等价：

```
import os
import sys
import getopt
import ftplib
```

为了清楚起见，可为每个需要输入的模块编写 import 语句。多数情况下，在一个 import 语句中输入多个模块也无妨。而且对多数程序员而言，阅读该行代码其余部分就能发现已经输入的模块。

5.1.2　用别名输入模块

Python 2.0 对基本的 import 方法做了如下扩展：可以在另一名称空间中输入模块，该名称空间不同于根据模块名选择的默认名称空间。例如，使用下列语句就可如同在 ftp 中一样输入 ftplib 模块：

```
import ftplib as ftp
```

现在，系统将所有对 ftplib 模块的调用都必须在 ftp 名称空间中进行标识。

使用别名是测试模块新版本而不影响原始版本的绝好方法。例如，想像一下，现在有两个模块：一个为稳定版 mylib，另一个为开发版 newmylib。在测试脚本中可以改变引用：

```
import newmylib as mylib
```

如果新模块没有改变任何 API 函数，就不需改变测试脚本的其他部分。

5.1.3　输入特定模块实体

要从给定模块输入特定函数和对象，可使用 from … import … 语句。例如，要从模块 foobar 输入对象 foo 和 bar，可使用下列语句：

```
from foobar import foo, bar
```

上面的语句与 import … 语句的主要差别在于：现在可在当前名称空间中使用输入的对象。因此，不必再显式指定这些对象。例如，下面的脚本是有效的：

```
from Foobar import foo, bar
foo()
bar()
```

要把给定模块的任何对象都输入到当前名称空间，可使用*：

```
from Foobar import *
```

5.1.4　重新加载(reloading)模块

通过调用内置函数 reload ()，可以重新加载前面已经装载的模块。如下例所示，reload ()语

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

句强制 Python 对模块重新解释和再输入。

```
Import foo
...
reload ( foo )
```

但是，reload 函数只能重新加载 Python 模块；不能重新加载依赖于 C/C++动态库的模块。重新加载操作不能修改已存在对象；因为这些对象使用了模块中定义的方法，除非删除旧的方法，否则这些对象将一直使用旧的方法。其他信息请参阅第 8 章。

5.1.5 模块搜索路径

装载模块时，Python 解释器搜索标准目录组以寻找指定的模块。搜索路径特定于系统和安装路径。可使用 sys.path 找到当前的目录列表。下面是在 Solaris 8/x86 中安装 Python 2.0 后得到的目录列表：

```
['', '/usr/local/lib/python2.0',
'/usr/local/lib/python2.0/plat-sunos5',
'/usr/local/lib/python2.0/lib-tk',
'/usr/local/lib/python2.0/lib-dynload',
'/usr/local/lib/python2.0/site-packages']
```

在 MacOS 9.1 中的等价目录为：

```
['', 'development:applications:python 2.0',
'development:applications:python 2.0:mac:plugins',
'development:applications:python 2.0:mac:lib',
'development:applications:python 2.0:mac:lib:lib-toolbox',
'development:applications:python 2.0:mac:lib:lib-scriptpackages',
'development:applications:python 2.0:lib',
'development:applications:python 2.0:extensions:img:mac',
'development:applications:python 2.0:extensions:img:lib',
'development:applications:python 2.0:extensions:numerical:lib',
'development:applications:python 2.0:extensions:numerical:lib:packages',
'development:applications:python 2.0:extensions:imaging:pil',
'development:applications:python 2.0:lib:lib-tk',
'development:applications:python 2.0:lib:site-packages']
```

上面两个目录列表中''表示当前目录。

在 sys.path 中增加一些语句就可以向模块搜索路径添加新目录。例如，可以向搜索列表中添加 ./lib/python：

```
import sys
sys.path.append('./lib/python')
```

现在 import 语句不仅搜索标准库，还搜索刚才添加的目录。

要向标准搜索路径中插入新目录，可使用 insert ()方法在列表开头添加目录(不是在列表

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

结尾)：

```
sys.path.insert(0,'./lib/perl')
```

相应地，在支持环境变量的平台上(Windows、Unix、BeOS 等)，可以在执行模块前向 PYTHONPATH 变量添加目录。

5.1.6　模块装载与编译

遇到装载特定模块的请求时，Python 解释器在下列位置寻找需要装载的模块：

- 用 Python 编写的程序/模块。
- 编译成共享库或 DLL(动态链接库)的 C 或 C++扩展库。
- 包含模块集合的包(请参阅本章后面的“包”部分)。
- 用 C 编写且链接到 Python 解释器的内置模块。

实际上，Python 搜索的是 Python 库搜索路径中的文件(假设模块为 foo，则搜索顺序如下)：

(1) 定义包 foo 的目录。

(2) 名为 foo.so、foomodule.so、foomodule.sl 或 foomodule.dll 的已编译扩展或库。当然，具体的解释过程依赖于主机的操作系统。有些操作系统不支持动态可装载模块。

(3) 文件 foo.pyo(假设使用了-O 选项)。

(4) 文件 foo.pyc(foo.py 的预编译字节码)。

(5) 文件 foo.py。

(6) 内置模块 foo。

Python 找到文件 foo.pyc 后，将对照 foo.py 来校验 foo.pyc 的时间戳。如果 foo.py 比较新，则编译文件，而且把该模块的已编译的字节码写入 foo.pyc。因此，Python 总是试图装载用 import 装载的模块的预编译字节码。作为脚本执行的模块不会以任何形式被预编译和存储。

如果命令行选项 –O 有效，则 Python 装载存储在 .pyo 文件里的预编译字节码的优化格式。除了.pyc 文件中没有行数、声明和其他调试信息(用于跟踪模块的执行过程)以外，其他内容和 .pyc 文件一样。

如果装载上述不同组件后，还没有找到模块，则引发异常 ImportError。

5.2　模块输入的技巧

除了已经叙述的基本方法外，Python 还支持许多不同技巧。用这些技巧可以在 Python 脚本中输入模块。多数技巧供专家级程序员使用。这些技巧可提高模块使用方法的灵活性。

5.2.1　在脚本中使用 import

一般在脚本头部输入需要使用的模块。例如：

```
import os,sys,getopt

# Start processing
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
def start():
...
```

只有在脚本执行过程中发现了 import 语句，Python 解释器才在 import 语句所在位置执行该语句。因此，可以像其他 Python 代码一样执行 import 语句，包括使用 if 或其他控制语句：

```
if (module == 'os'):
  import os
else:
  import sys
```

实际上，这种技巧应用在类似于 os 的模块中。这类模块根据 Python 解释器运行的主机来装载特定于平台的模块，如 posix 或 mac。

只在函数实际调用时才装载模块的做法也是有效的，例如：

```
def sendmymail():
  import smtplib
```

虽然输入模块的方式很灵活，但是，在未使用 exec 语句的情况下，不能将变量或字符串作为模块名来输入该模块。如下例所示：

```
module = 'os'
exec 'import '+module
```

有关 exec 语句的其他信息请参阅第 8 章。

5.2.2 追踪 import 语句

如果未能正确装载模块，则 import 语句会引发异常 ImportError。在脚本中可安全地直接捕获模块装载错误。因此可以轻松地退出该异常。如下例所示：

```
try:
    import mymodule
except ImportError:
    print "Whoa! We seem to be missing a module we require here"
    import sys
    sys.exit()
```

5.2.3 标识模块或脚本

每个模块均定义变量 __name__，该变量包含模块的名字。可以使用该变量来决定在哪个模块中执行特定代码段。而且，使用该变量还可方便地决定是否将给定模块作为脚本运行、或者是否已被输入。作为脚本运行的模块将__name__设置为__main__。可以用下例进行测试：

```
if __name__ == '__main__':
    # Work as a script
else:
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
# Work as a module
```

注意，__name__返回模块原始名，而不是使用 import...as 输入时所指定的别名。当使用上面所提到的技巧并不会引起任何不同。但是在输入模块时应小心使用标识该模块的内部模块名。

虽然这种特殊技巧(__name__变量使用技巧)不应影响模块中定义的函数和类，但可将该技巧作为一种测试模块的便利手段。把模块作为脚本运行时，__name__变量将进行自我测试。输入模块时，__name__变量则定义函数。正因为模块可当作脚本和附加库使用，因此，程序员就无需对脚本进行单独测试，而只需关注 Python 代码的重用。

检查 Python 中的任何标准模块，都可能发现一个问题。例如，将 smtplib 模块作为脚本运行，可以发送电子邮件消息：

```
$ python smtplib.py
From: mc@mcslp.com
To: mc@mcwords.com
Enter message, end with ^D:
Hello Doppleganger!
Message length is 20
send: 'ehlo twinsol\015\012'
reply: '250-twinsol.mchome.com Hello localhost [127.0.0.1], pleased
to meet you\015\012'
reply: '250-ENHANCEDSTATUSCODES\015\012'
reply: '250-EXPN\015\012'
...
```

为清楚起见，对上例的输出进行了整理，不过还是可以观察到效果。将 smtplib 作为模块输入则没有这种效果——输入模块后，该模块按普通方式执行。

5.3　包

使用 Python 中的包可以把一套模块用普通的包名进行分组。具有 Perl 背景的程序员通常希望用下面的语句：

```
import package.module
```

来自动搜索文件 package/module.py 的不同目录，但是并不能达到这个目的。采用这种方式使用 import 语句时，需要事先设置包和包的对应目录结构。

要定义包，首先需创建与包同名的目录，接着在该目录下创建__init__.py 文件。该文件包含一些 Python 解释器需要的指令。使用这些指令，可以在包中输入模块和模块组。例如，下面的目录结构显示了名为 MediaWeb 的工程的结构布局。该工程是种网络管理工具，且置于 Net 目录下：

```
Net/
__init__.py
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
MediaWeb/
    __init__.py
    Weather.py
    Weblog.py
    Systemlog.py
```

现在，在 Python 中可使用多种方法从该结构输入模块。语句：

```
import Net.MediaWeb.Weather
```

将从 Net/MediaWeb 目录输入子模块 Weather。和其他 import 语句相似，必须显式地指出该模块中的函数，也就是这种形式：Net.MediaWeb.Weather.report()。

语句：

```
from Net.MediaWeb import Weather
```

输入了相同模块 Weather。但是，现在可以不用包做前缀，而直接使用 Weather.report()。

语句：

```
from Net.MediaWeb.Weather import report
```

把 report 输入到局部名称空间。这样就可以用 report()调用该函数。

在每种情况下，都执行__init__.py 中的代码，从而完成特定包的初始化工作。在输入过程中，Python 一旦发现__init__.py 文件就会对其进行处理。例如，输入 Net.MediaWeb.Weather，则执行 Net/__init__.py 和 Net/MediaWeb/__init__.py 两个文件。

每个__init__.py 文件中的内容对于程序员是完全可见的。这些文件可能是空的，此时，除了输入所选模块外不会发生任何事情。但是，就像在 Perl 中一样，这些文件必须存在用于目录嵌套。

另一方面，可能需要执行特定选项。例如，语句：

```
import Net.MediaWeb
```

不会自动强制 Python 输入 Net/MediaWeb 目录中的内容。下面的语句也不会自动强制 Python 输入 Net/MediaWeb 目录中的内容。

```
From Net.MediaWeb import *
```

在这两个例子中，可选择下面两种方案之一。第一个例子中，可以把下列语句输入到 Net/MediaWeb/__init__.py 文件中：

```
# Net/MediaWeb/__init__.py
import Weather,Weblog
```

第二个例子中，可以使用__init__.py 文件的__all__属性。下面的语句将对需要输入的模块的列表进行初始化：

```
# Net/MediaWeb/__init__.py
__all__ = ['Weather', 'Weblog']
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

5.4 创建模块

在 Python 中创建新模块就像编写原始脚本一样容易。在 Python 中创建的任何文件或脚本可立即当作模块使用，而且不必对代码做任何修改。实际上，除了要把文件复制到标准位置使文件易于使用外，不必做任何工作。

例如，下面的示例创建了定义简单函数 add ( )的小型的脚本 mymath.py：

```
def add(a, b)
    return a+b

print add(1, 1)
```

可以输入 mymath.py 文件并在其他脚本中使用 add()函数，惟一要做的就是输入 mymath 文件并调用该函数：

```
import mymath
print mymath.add(2,2)
```

import 语句自动寻找 mymath.py 文件，并创建新的名称空间，接着把 add()函数输入到该名称空间。

而且，可以使用下面的语句显式地输入 add()到当前名称空间：

```
from mymath import add
```

确实非常简单！

因为这种灵活性，所以在 Python 中编写和创建的所有代码和函数都可不经修改而供其他脚本和模块使用。所有代码可直接重用，不必考虑特意创建模块或做其他特殊操作以使代码可以重用。

与 Python 比较，要在 Perl 中创建灵活性好的模块 MyModule，必须在文件 MyModule.pm 中添加下面所示的序言(preamble)：

```
package MyModule;

require Exporter;
use vars qw/@ISA @EXPORT/;
@ISA = qw/Exporter/;
@EXPORT = qw/add/;
```

对于每个需显式输出的实体，都必须更新@EXPORT；而且输入文件时，需添加该文件的返回值——“true”。与 Python 相比较，Perl 显得繁琐，而且序言部分出错时，模块可能完全停止执行。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第6章 面 向 对 象

面向对象编程是一种编程机制。使用面向对象方法编程可以创建智能变量——对象。对象可用来存储复杂的结构。这种结构不仅可以保存数据，同时还支持作用于数据的函数和操作。面向对象系统贯穿于类的创建过程。类定义单独对象(称为实例)的格式和结构，还定义对这些对象进行操作的函数。

可以将类组织成树形结构，以便存在通用类。在通用类中，有些特殊类用来模拟其他结构。类也可从其父类中继承方法。使用父类以后，无论有多少不同的类使用某种方法，都可以只在父类中定义该方法(只需定义一次)，从而提高了代码重用率。

例如，要创建用于管理银行和信用卡账户的系统。可以创建基类 Account，用来保存两种信息：账户名和该账户的现存余额。在类定义中还包含两种方法：一种用于向账户中存款(自动更新账户余额)，另一种相似方法用于从账户中提款。

基类 Account 自身不足以保存某些账户的特定信息。因此也可继续创建子类 BankAccount，该子类继承了 Account 的属性和方法，而且还包含了新的属性用来保存银行账户号、账户序号和银行名称。现在，在 BankAccount 类中可以使用 Account 类所定义的方法进行存款和提款，而不必重新定义。

另外，还可创建继承 Account 类的子类 CreditCard。CreditCard 类包含一些属性：账户号、有效期、信贷限额和利率。add_interest()方法通过计算前一段时间的账户利息来更新该账户的余额。

现在考虑创建这些类的细节情况，同时，将学习使用 Python 创建类、对象和方法。

Python 将对象作为内部数据类型使用。因此，大家应该已经比较熟悉如何创建不同对象，并能够使用应用于这些对象的信息和方法。本章将介绍创建新类的方法。大家可以创建自己的类和对象，并学习对类进行扩展的方法。扩展的类可以操作 Python 中的内置函数和运算符。

6.1 创建类

在 Python 中创建新类需使用 class 语句。class 语句类似于 Python 中的其他块定义语句。在 class 块中的所有内容都成为所定义类的一部分。

创建新类的格式如下所示：

```
class CLASSNAME([CLASS_PARENT, ...]):
    [ STD_ATTRIBUTES]
...
    def METHOD(self, [METHODARGS]):
...
```

CLASSNAME 是要创建的类的名字。Python 中类的基本规则和其他对象的规则相同。但是，

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

一般用户自定义类名采用标题格式以区别于 Python 提供的标准类库。类名后面的括号是可选的。括号里的内容用于指定从哪些类继承属性。详细信息请参阅本章后面的“类继承”部分。

STD_ATTRIBUTES 是应用于该类所有实例的默认属性。这些值是静态值，它们不同于类初始化过程中设置的属性值。现在对类方法的细节进行简要讨论。

例如，可以用如下方式创建类 Account：

```
class Account:
account_type = 'Basic'
def __init__ (self, name, balance):
self.name = name
self.balance = balance
def deposit(self, value):
self.balance += value
def withdraw(self, value):
self.balance – = value
```

要创建基于新类的新对象，可以像使用函数一样调用该类：

```
bank = Account ('HSBC', 2000)
```

6.1.1　类的方法

类的方法实际上就是在给定类的范围内定义的函数。类的方法与普通函数之间的惟一区别在于：在任何类的方法中，第一个参数都是该方法所操作的对象。例如，在实例中调用 deposit()方法：

```
bank . deposit (1000)
```

Python 实际调用了 Account 类的函数 deposit()。调用时，对象作为第一个参数(典型情况是 self)，提供给函数 deposit()的参数作为第二个参数：

```
Account . deposit (bank, 1000)
```

这样，就可以访问对象属性，并利用函数对属性值进行更新。如果不使用 self 参数，就无法修改对象。从下面的 Account 类的简化类和方法定义中可以更清楚地看到这一点：

```
class Account:
account_type = 'Basic'
def __init__ (self, name, balance):
self.name = name
self.balance = balance
def deposit(self, value):
self.balance += value
```

1. 对象构造函数

__init__()函数是类中用于构造函数的特殊函数名。创建基于类的新对象时，调用该类的__init__ ()函数。该方法接受第 4 章中描述的任何形式的参数(请参阅第 4 章)。下例中：

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
def __init__ (self, name, balance):
self.name = name
self.balance = balance
```

__init__ ()方法接受两个参数：创建的账户名和账户余额。它们用于初始化对象属性值。上述方法实际上做了以下调用：

```
bank = Account . __init__ ('HSBC', 2000)
```

新变量 bank 是类 Account 的对象和实例。访问 balance 属性可得到账户余额。

2. 对象析构函数

所有类的实例都有一个引用数，用于计算实例被引用的次数。该引用数包括：每个引用该实例的对象名，以及实例作为列表、元组或字典的一部分的次数等等。当引用数变为 0 时，Python 自动删除实例，并释放实例用于保存对象数据的内存。

如果要定义定制删除对象操作(析构函数)的顺序(可能还要间接引用其他对象或者记录实例删除过程)，那么需要定义方法__del__ ()。需要删除对象时，Python 自动调用__del__ ()方法。如果类没有自定义__del__ ()方法，则所有对象继承内置的__del__ ()方法。因此，多数基本对象(如本部分创建的示例)都不要求自定义__del__ ()方法。

但是，需要注意的是：有些对象在删除时要求关闭文件和网络连接或释放其他系统资源，此时，不能调用__del__ ()删除这些对象。

3. 特殊方法

Python 支持许多特殊方法。这些方法提供必要的挂钩(hook)以连接到 Python 解释器的内置函数和运算符。这些方法的格式类似于__init__ ()和__del__ ()，函数名的开头和结尾都是双下划线。例如，在 Account 类中定义__add__ ()函数后，可以使用标准运算符+把两个账户连接起来。

表 6-1 是 Python 支持的特殊方法的完整列表。该表列举了一些基本方法。用这些基本方法可以在给定类中定义大多数基本运算。程序员应该在对象类中定义该表所列出的多数方法。尤其是计划把这些类向使用它们的公众发布时，应对表中的方法进行定义。

表 6-1　特殊的类方法

方　　法	描　　述
__init__ (self [,args])	创建类的新实例时调用
__del__ (self)	删除实例时调用
__repr__ (self)	使用 repr()函数或反勾号运算符时调用。返回值为与 eval()兼容的对象字符串表达式(用于重建对象)
__str__ (self)	调用 str()函数时调用。返回值为字符串的非正式表达式
__cmp__ (self,other)	比较两个对象时调用。Self 的逻辑值小于 other 则返回负数，两者逻辑相等则返回 0，self 的逻辑值大于 other 则返回正数
__hash__ (self)	计算散列值时调用。返回 32 位散列值索引

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

方　法	描　述
__nonzero__ (self)	self 的逻辑值为假，则返回 0；self 的逻辑值为真，则返回 1
__getattr__ (self, name)	使用 self.name 时调用。返回属性 name 的值
__setattr__ (self, name, value)	使用 self.name=value 时调用。将属性 name 的值设为 value
__delattr__ (self, name)	调用 del self.name 时调用。删除属性 name

应设置__str__ ()和__repr__ ()方法以使它们返回代表给定对象的合适的字符串。在__str__ ()中，可返回基本字符串：

```
def __str__ (self)
    return "%s: %g" % (self.name, self.balance)
```

__repr__ ()方法的返回值经 eval()分析后可用于重建对象。即返回的字符串代表重建对象时所需的调用。例如，对于 Account 类，需要返回产生下列语句的字符串：

```
Account (name, balance)
```

实际定义方法时，采用下列形式的语句：

```
def __repr__ (self)
    return "Account ('%s', %g) " % (self . name, self . balance)
```

4. 模仿序列或字典对象

如果创建了对象类型(可通过序列或字典接口访问其中的信息)，那么必须定义表 6-2 中列举的方法。使用 len()函数或直接访问对象片断或元素时，调用该表中的方法。

表 6-2　模仿序列和映射对象的方法

方　法	描　述
__len__ (self)	返回 self 的长度。由内置函数 len()调用
__getitem__ (self, key)	返回 self[key]
__setitem__ (self, key, value)	将 self[key]的值设置为 value
__delitem__ (self, key)	删除 self[key]
__getslice__ (self, i, j)	返回 self[i : j]
__setslice__ (self, i, j, value)	将 self[i : j]的值设置为 value
__delslice__ (self, I, j)	删除 self[i : j]

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

5. 重载数学运算符

任何对象(甚至内置对象类型)使用运算符时，Python 实际上调用一些特殊方法。可以在用户自定义对象中添加这些特殊方法的实现，使用户自定义对象可以使用 Python 中的内置运算符。这个过程重载了运算符，而且使这些运算符可以处理不同对象类型，因此，该过程称为运算符重载(operator overloading)。

但应注意，重载运算符时，应该使得该运算符在上下文环境中有意义。例如，重载__add__ ()方法，用于在两个旧账户的基础上创建一个新类型的银行账户，这种重载是有意义的。如果创建 HTTP 类，并重载了__add__ ()方法，那么，无论用+运算符实现的操作有多花哨，这种重载都可能是没有意义的。

重载运算符类处理自定义的类和对象的方法非常简单。需要做的就是定义特定方法用于处理操作过程。例如，__add__ ()方法定义了用+运算符将两个相同类型的对象加在一起时所应完成的操作。如下例所示，可以在 Account 类中定义方法来合并两个银行账户：

```
def __add__ (self, other):
    return Account(self.name + ' and ' + other.name,
                   self.balance + other.balance)
```

在这种情况下，__add__ ()返回两个账户的账户余额，就如下例所示：

```
bank = Account('HSBC', 2000)
creditcard = Account('MBNA', -1000)
assets = bank + creditcard
```

现在，assets 包含一个新对象。该对象的 name 属性为“HSBC and MBNA”、balance 属性为 1 000。

虽然这些运算符是以数学运算符的形式列举出来的，但是，如果希望使用运算符来对对象进行操作，那么，也可以在这些对象中使用运算符。例如，字符串和其他序列对象定义了__add__ ()和__mul__ ()方法。表 6-3 列举了模仿数字对象时需要定义的所有方法。

表 6-3 重载标准 Python 运算符的方法

方 法	结 果
__add__ (self, other)	self + other
__sub__ (self, other)	self – other
__mul__ (self, other)	self * other
__div__ (self, other)	self / other
__mod__ (self, other)	self % other
__divmod__ (self, other)	divmod (self, other)
__pow__ (self, other [, modulo])	self ** other, pow(self, other, modulo)
__lshift__ (self, other)	self << other
__rshift__ (self, other)	self >> other

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

方　法	结　果
__and__(self, other)	self & other
__or__(self, other)	self | other
__xor__(self, other)	self ^ other
__radd__(self, other)	other + self
__rsub__(self, other)	other – self
__rmul__(self, other)	other * self
__rdiv__(self, other)	other / self
__rmod__(self, other)	other % self
__rdivmod__(self, other)	divmod(other, self)
__rpow__(self, other [, modulo])	other ** self, pow(other, self, modulo)
__rlshift__(self, other)	other << self
__rrshift__(self, other)	other >> self
__rand__(self, other)	other & self
__ror__(self, other)	other | self
__rxor__(self, other)	other ^ self
__neg__(self)	-self
__pos__(self)	+self
__abs__(self)	abs(self)
__invert__(self)	!self or ~self
__int__(self)	init(self)
__long__(self)	long(self)
__float__(self)	float(self)
__complex__(self)	complex(self)
__oct__(self)	oct(self)
__hex__(self)	hex(self)
__coerce__(self, other)	coerce(self, other)

6. 附加方法

在给定类中自定义的任何其他函数都称为该类的附加方法。每种方法的格式为：

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
def METHOD(self [, args])
```

正如前面所提到的，self 参数被传递给每个类方法。因为 self 提供到对象自身的连接，以便使用“and/or”来修改给定对象的属性。在前面的例子中，deposit()和 withdraw()方法是 Account 类的其他方法。

Python 没有限定在类中可以创建的方法的数量，也没有限定这些方法可以完成的任务。可以通过调用方法名来激活该方法。例如，在 Python 中可使用下列语句创建 Account 变量的新实例：

```
bank = Account ('HSBC', 2000)
```

用下面的语句可向账户存款：

```
bank . deposit(1000)
```

注意，在上例中，deposit()方法不直接修改对象的 balance 属性，因此，该方法不会返回任何值。如果要返回值，例如，要得到账户余额，可使用 return 语句(如同在其他函数中得到返回值)：

```
def accbalance(self):
  return self . balance
```

注意，使用 return 方法只是重复了直接访问属性的过程。给定了上面的 accbalance()方法后，下面两条语句会得到相同结果：

```
print "Balance: ", bank . accbalance()
print "Balance: ", bank . balance
```

除非对输出格式重新格式化(使用__str__()和__repr__()等方法)，否则，重复访问对象的属性是没有意义的。

7. 使对象可调用

可以使对象与函数一样能被 Python 调用。实现手段为定义__call__()方法。这意味着语句

```
object (arg1, arg2, ...)
```

实际调用了

```
object . __call__(self, arg1, arg2, ...)
```

在需要对某对象快速完成特定操作时，__call__()方法非常有用。例如，可指定调用类实例自动返回账户余额。

可调用对象还用于只定义了一种方法的类的实例。例如，可创建一个方法对象：在被系统调用时，该对象会返回已格式化版本从而适用于在 Web 上显示。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

6.1.2 类的继承

Python 根据类名后的括号来确定该类从哪些类继承了属性和方法。作为本章引言中示例的扩展，可以定义两个类：CreditCard 和 BankAccount。这两个类都继承自基类 Account。如下所示：

```
class BankAccount(Account):
    account_type = 'Bank Account'
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance
  def __init__(self, name, balance, accno, sortcode, bankname):
        Account.__init__(self, name, balance)
        self.accno = accno
        self.sortcode = sortcode
        self.bankname = bankname
class CreditCard(Account):
  account_type = 'Credit Card'
  def __init__(self, name, balance, accno, expiry, limit, rate):
        Account.__init__(self, name, balance)
        self.accno = accno
        self.expiry = expiry
        self.limit = limit
        self.rate = rate
def withdraw( self, amount ):
  if abs( self.balance – amount ) < self.limit:
        self.balance  - = amount
  else:
        raise ValueError, "You're past your limit!"
def add_interest(self):
  self.balance  - = ((abs(self.balance)*(self.rate/100))/12)
```

注意，CreditCard 类中的 withdraw()方法重写了 Account 类中定义的默认方法。并且，在试图超限度提款时会引发异常。

如果现在创建 CreditCard 的实例，那么就可以使用 withdraw()和 deposit()方法：

```
>>> visa = CreditCard('HSBC',  - 1000, '12345678', '02/99', 8000, 18)
>>> visa.deposit(500)
>>> visa.balance
 - 500
>>> visa.withdraw(2000)
>>> visa.balance
 - 2500
```

需注意的是：在创建继承自另一类的新类的实例时，新类的构造函数不会自动调用基类的

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

构造方法。因此，在 BankAccount 和 CreditCard 类中的构造函数都需要使用下面一行代码：

```
Account.__init__(self, name, balance)
```

该行代码直接调用基类的构造函数。它把 name 和 balance 提供给基类的构造函数，从而在当前类中对实例进行了初始化。当然，也可不必立即调用基类的构造函数，但立即调用基类的构造函数会更好些。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第7章　异常和错误的捕获

到目前为止，本书一直没有讨论捕获错误的问题。原因是先集中讨论了 Python 语言的基础知识，如变量类型和类。多数语言采用标准方法捕获错误。许多用户都接触过脚本语言 Perl，因此，本章以 Perl 作为示例语言。例如，考虑下面的用于打开文件的 Perl 代码：

```
open (FILE, "file.txt" ) || die"Couldn't open file file.txt : $!");
```

捕获错误的方法为：解释器寻找函数的返回值；如果正常调用 open 函数，那么返回值为代表真值的正整数。如果调用 open 函数失败，那么 die 函数将终止 Perl 解释器(或封装的 eval 块)并用来自特殊变量$!中的字符串来报告该错误。

用这种方法捕获错误时存在几个问题：

- 只能检测成功或失败两种状态——不能发现中间状态。
- 如果要检测失败原因，必须检查$!变量。
- 除非把每次调用都嵌入到 eval 块中，否则只能处理失败的调用，而不能处理引起失败的原因。

这种限制不仅出现在 Perl 中，也出现在使用相似的错误检查方法的 C/C++中。虽然 C 的头文件通常包含了可以捕获的指定错误数，但只是用在多重 if 或 switch 语句中。C++支持异常系统；但是，由于编译器对异常支持的某些历史问题，这种异常系统也未得到充分利用。Pascal、Visual Basic(VB)和其他多数语言采用相同的方法——即检查返回值的真或假。

现在考虑另外一种常见的需要捕获错误的情况：

```
$den = 25;
$num = 0;
print $den/$num, "\n";
```

在 Perl 中，由于除以零是未定义的，因此，这段代码在运行时将引起致命错误(解释器终止)。除非检查被除数以确保该数不为零或者将语句嵌入到eval块中(实质上与嵌入Perl解释器相同)，否则，无法在 Perl 的普通执行过程中捕获这种错误。现在的问题是：无法将所有代码都嵌入到 eval 块中，而且，虽然在计算前检查每个参与计算的值是种良好的编程习惯，但对于优化脚本而言，代码就显得冗余。

在 Perl、C 或 C++中，都没有其他更好的方法用于在执行时捕获单独语句的执行错误。但是，Python 提供了一种不同的方法。在 Python 中发生错误时会引发异常，可使用内置语句引发自身存在的异常并捕获异常，从而安全地进行处理。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

7.1　异常的概念

异常(exception)是种特殊类型的 Python 对象，它可以向模块调用者传递调用失败信息。异常发生后，可以捕获该异常，异常信息则传递到异常处理器中。与 Perl 和 C 不同的是：在 Python 中，可以获得问题的准确描述，甚至可以获得已发生错误的额外信息。

考虑异常的最佳方式为：把异常当作可配置和可扩展的信号处理器(请参阅“异常和 goto”部分)。使用 try 语句并执行需要监控的语句就可创建异常处理器。一旦发生异常，程序控制权将返回到 try 语句。不必监控所有的函数调用，也不必依赖于简单的返回值而决定代码块是否成功执行。只需做到在异常发生时能够处理异常。

异常与 goto

人们可能认为 Python 中的异常类似于 C、VB 和 Perl 中以不同形式出现的 goto 函数。或者认为 Python 中的异常提供了与 C 中的 setjmp()/longjmp()相似的功能。但是，goto 函数、setjmp()和 longjmp()函数都存在一些限制。任何基于 goto 语句的系统都存在弊端，即使 C 语言具有由 setjmp()函数提供的高级状态记录功能，也存在同样的弊端。

goto 语句存在的问题为：如果未对程序进行清理松散末端(loose end)就从程序当前位置跳转到新的位置，那么无法知道在代码块的上下文中打开的文件和变量会发生什么情况。如果没有全局变量，那么在跳转过程中不可能传递任何有用信息。因为 goto 语句会破坏程序的正常流程和安全执行，所以采用任何形式的 goto 语句都是不好的编程习惯。

Python 中的异常处理系统更加高级。如果在调用者中的任何位置都可能引起异常，那么可以用处理链上的函数进行处理。其他情况下，无论异常处理器的位置和异常发生的位置在何处，都可类似于 return 语句，将异常用于函数中。

因为异常是语言的内置功能，所以处理过程比其他方法更加清晰—— 不必依赖于信号或其他功能。异常也是对象，因此，可以对异常赋予值和方法。引发异常后，可以返回对象。这意味着不仅可用异常进行错误检查，还可完成更多任务。异常的其他用途包括：

- 多重错误处理：无需检查单个返回值并与已知列表进行比较。调用函数或执行块时，只需知道如何处理特定异常。可以忽略任何其他错误，或者用通用异常处理器捕获其他错误。
- 特殊控制流程：通过使用异常处理器，可以改变基于特殊功能的应用程序的流程。例如，考虑一个打开网络连接或使用本地文件的程序。使用异常处理器后，可以在网络连接失败时使用默认文件。
- 事件监控：不必使用独立的全局变量，也不必显式检查返回值，只要使用异常系统就可返回事件结果。Perl 和 C 处理事件的普通方法为检查返回值。为了验证是否返回了信息，需检查列表或散列表中返回的条目数量。在 C 中，必须使用整数返回值和全局值，或者为即将写入的信息提供函数的指针参数。

异常基于对象或对象类。在独立的异常之间存在一定程度的继承。所有异常都是基类 Exception 的成员。从每个类继承某些功能直至得到特定异常，这样就继续分层。例如，ArithmeticError 异常处理计算错误，而 ZeroDivisionError 异常更具体地处理前面的示例中出现

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

的计算式中除以零的错误。

使用对象和类，可以创建基于一般错误或特定错误的异常处理器。例如，如果使用来源不明的数据进行计算，那么可能需要捕获基本的算术错误；如果使用内部数据，可能需要知道引起了哪类计算错误。

7.2 引发异常的结果

发生异常时，Python解释器的默认操作为：识别错误并产生堆栈用于跟踪错误的发生地和如何到达发生地捕获该错误的方法。这比Perl或C产生的典型错误输出更加有用(在Perl或C产生的典型错误输出中，只能看到发生了错误的行，这也会使跟踪错误变得非常困难)。

为示范该过程，观察一个试图进行除零操作的简单脚本：

```
print 24/0
```

因为不能除以零，所以上面一条语句会引发异常：

```
Traceback (innermost last):
  File "t.py", line 1, in ?
  print 24/0
ZeroDivisionError: integer division or modulo
```

如果把这个计算式嵌入到函数调用中，会得到关于如何发生错误的更详细描述：

```
def divide(x,y):
    return x/y

def calc(x,y):
    return x*(divide(x,y))

print calc(24,0)
```

执行该脚本，将会看到除了简单的行号外，还有错误发生地的详细描述：

```
Traceback (innermost last):
    File "t.py", line 7, in ?
    print calc(24,0)
    File "t.py", line 5, in calc
    return x*(divide(x,y))
    File "t.py", line 2, in divide
     return x/y
ZeroDivisionError: integer division or modulo
```

从前面的示例可以明白Python是如何用堆栈跟踪的方式进行错误报告的。可以从实际错误发生的地方(第二行的计算式)返回到触发错误的原始调用(第七行的调用，通过第五行进行调用)。注意，堆栈跟踪的范围包括函数名和发生错误的源文件名。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

这里用到的方法是尝试和捕获这些错误，因此，可以向用户提供关于已发生错误的更有用的信息。为此，可以把调用嵌入到由 try 语句和处理任何错误的代码组成的异常处理器中。重写除法脚本并包含异常处理器，代码如下：

```
def divide(x,y):
    return x/y
def calc(x,y):
   return x*(divide(x,y))

try:
   print calc(24,0)
except:
   print "Whoa!: Those numbers don't seem to work!"
```

这里创建了简单的异常处理器，它可以跟踪每种类型的异常并报告错误。因为没有指定跟踪哪类异常，所以，如果存在语法问题或者为 calc 函数提供字符串作参数，该脚本也会报告错误。实际上，Python 定义了许多不同的异常和异常类用于处理特定的错误类型。在本章后面的部分将进行详细讨论。

Python 异常系统与其他多数语言的处理方式不同。在与前面的脚本类似的情况下，通常在计算前检查值，以确保信息正确且在执行时不会产生错误。例如，在 Perl 中可能会这样处理：

```
print calc(24,0);

sub divide
{
      my ($x,$y) = @_;
      die "Cannot divide by zero with $y" if ($y == 0);
      $x/$y;
}

sub calc
{
      my ($x,$y) = @_;
      $x*(divide($x,$y));
}
```

这里存在的问题是：需要在计算结束时使用很多代码来跟踪不同类型的错误。如果使用字符串而非数字来调用 calc 函数，会发生什么情况呢？在 Perl 中调用 calc('Hello',1)，返回值为 0。原因是：Perl 试图将字符串转化为数字但没有成功，因此，Perl 假设字符串的值为 0。人们实际上希望把无效参数识别为错误。为此，需要插入另一个 if 语句测试参数内容并确定参数类型。

而在 Python 中，如果使用的数字存在问题，那么，Python 会向用户报告错误。下面的示例利用了 Python 识别特定错误类型的能力：

```
def divide(x,y):
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
    return x/y

def calc(x,y):
    return x*(divide(x,y))

try:
    print calc('Hello',1)
except ZeroDivisionError:
    print "Whoa!: Your trying to divide by zero and you can't!"
except TypeError:
    print "Whoa!: That doesn't look like a number!"
except:
    print "Whoa!: Some other kind of error occurred!"
```

运行该脚本，得到下列输出：

```
Whoa! : That doesn't look like a number!
```

现在可以识别除以零和无效类型错误，但是，并没有修改该函数，也不必预先知道可能发生的错误；而只需处理调用函数时发生的错误，并捕获任何可能发生的异常。可以指定捕获的错误类型，也可不指定。在前例中，捕获了特定错误(ZeroDivisionError 和 TypeError)和一般错误 Exception。该脚本并不关心语句失败的原因。程序员可跟踪语句并按照该语句进行操作。

7.3　异常的处理

用于异常处理的 try 语句有两种不同形式：try…except…else 语句和 try…finally 语句。前面已经列举了几个第一种形式的示例。如果需要跟踪异常，而并不作实际处理，那么，第二种形式提供了更简单的操作方法。观察一些示例后将会对这一点更加清楚。

7.3.1　try… except… else

try 语句的第一种形式的作用与 if 语句相反——嵌入需要执行的代码块，而用一些 except 语句处理发生的异常。这种形式的基本格式为：

```
try:
    BLOCK
except [EXCEPTION [, DATA...]]:
    BLOCK
else:
    BLOCK
```

当 Python 解释器遇到 try 语句时，遵循下述基本过程：

(1) 执行 try 语句下的 BLOCK。如果引发异常，则执行过程立即返回到第一个 except 语句。

(2) 如果引发的异常与指定的 EXCEPTION 相匹配，或者如果没有定义 EXCEPTION，则匹

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

配通用的异常，那么，执行相应的 BLOCK，同时异常处理器可以访问 DATA 中定义的对象。

(3) 如果引发的异常不匹配，那么执行过程跳转到下一个 except 语句。可以编写任意数量的 except 语句。

(4) 如果没有 except 语句相匹配，则异常传递到下一个调用本代码块的最高层 try 代码块中。

(5) 如果没有发生异常，则执行 else 块。

在这个过程中，有几个关键部分需格外注意。

1. except 语句支持多重格式

关于这一点，前面的过程表现地还不够清楚，但是可以用许多不同方法捕获错误并引入许多异常处理器。except 语句接受五种不同格式，参见表 7-1。

表 7-1　except 语句接受的格式

格　式	描　述
except:	捕获所有异常(或其他所有异常)
except name:	捕获 name 指定的异常
except (name1, name2):	捕获列出的所有异常
except name, data:	捕获异常 name 和任何额外的返回值
except (name1, name2), data:	捕获列出的所有异常和任何额外的返回值

2. 顺序检查 except 语句

在异常处理过程的第 2 和第 3 步中，可以看到：引发异常后，Python 检查每个 except 语句以确定是否有指定的异常与发生的异常相匹配。检查过程按顺序进行。Python 不会在一次过程中检查所有的 except 子句。这意味着程序员既可以识别个别异常，也可以对异常分类(直至到最高层)。前文已经有这样的示例。

下面的脚本会寻找 ZeroDivisionError 和 TypeError 异常，并捕获其他任何错误：

```
def divide(x,y):
    return x/y

def calc(x,y):
    return x*(divide(x,y))

try:
    print calc('Hello',1)
except ZeroDivisionError:
    print "Whoa!: Your trying to divide by zero and you can't!"
except TypeError:
    print "Whoa!: That doesn't look like a number!"
except:
    print "Whoa!: Some other kind of error occurred!"
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

需要记住的重要一点为：应把最特殊的异常放在最前面，最一般的异常放在最后面。如果把所有异常类型的基类 Exception 放在第一个 except 语句中，那么 Python 不会检查其他的任何异常处理器。

3. 异常处理器只运行一次

执行前面的脚本，可以发现报告了 TypeError 异常。虽然通用的 Exception 可捕获所有异常，但是 try...except...else 语句只允许一次匹配过程。一旦执行了异常块，控制过程将返回到整个 try 语句后的语句中。当然，调用其他函数或 exit 函数后就另当别论。

4. 没有异常时才运行 else

根据异常处理过程的第 5 步，只在没发生异常时才执行 else 语句。通常 else 的用法与 if 块中的 else 相同：

```
try:
    file = open('file')
except EnvironmentError:
    print "Whoops: Can't seem to open the file"
else:
    lines = file.readline()
    file.close()
```

这条规则适用于所有情况。如果要在异常处理器中运行相似语句，可以复制该语句，或创建处理它们的函数。

5. 捕获数据

多数异常除了包含错误本身的信息外，还包含引起异常的额外信息。例如，考虑下面的脚本：

```
def parsefile(filename):
    file = open(filename,'rw')
    print file.readline()
    file.write('Hello World!')
    file.close()

try:
    parsefile('strings')
except EnvironmentError,(errno,msg):
    print "Error: %s (%d) whilst parsing the file" % (msg,errno)
```

该脚本包含处理 EnvironmentError 的异常处理器。EnvironmentError 是个基类，它用于处理访问 Python 解释器外部的信息(如文件内容)时 Python 遇到的问题。不必捕获所有可能发生的不同错误，如文件不存在、文件系统满或一个文件结束条件。而只需捕获一般异常，并用该异常提供的附加信息向用户详细描述该错误。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

EnvionmentError 异常会自动返回 C 库的错误号和相关消息(如果存在的话)。例如，调用 parsefile 函数时，下列消息都是有效的错误消息：

```
Error: No such file or directory (2) whilst parsing the file
Error: Bad file descriptor (9) whilst parsing the file
Error: No space left on device (28) whilst parsing the file
```

如果指定了单个对象用于获取数据，那么，内置异常将把这些数据信息转化为字符串格式，并把字符串放到该对象变量指向的引用中。如果如前例所示，要捕获指定信息，需要提供 tuple。返回信息的格式由每个异常定义(关于标准异常返回数据的其他信息，请参阅本章后面的“内置异常”部分)。

7.3.2　try... finally

用于代替 try...except...else 语句的另一种方法为：使用 try...finally 语句。try...finally 语句的格式如下：

```
try:
  BLOCK
finally:
  BLOCK
```

执行 try...finally 语句的规则如下：

- 执行 try BLOCK 中的语句。
- 发生异常时，在该异常传递到下级前，执行 finally BLOCK。
- 如果没有发生异常，则执行 finally BLOCK，控制过程按普通方式进行，直至整个 try 语句之后。

如果只是要运行一段代码，而不管是否发生异常，那么，try...finally 语句是非常有用的。例如，当通过网络与远程服务器通信时，需要捕获错误；但是还要保证不管是否有错误，通信信道都是关闭的。可使用 try...finally 语句完成这项任务：

```
try:
  remote.send_data(destination,datastream)
finally:
  remote.close_connection()
```

当然，对于 try...finally 语句自身，它只会把错误传播到下一级，因此，可能需要在另外的 try 语句中嵌入该调用：

```
try:
  try:
      remote.send_data(destination,datastream)
  finally:
      remote.close_connection()
except NetworkError,errorstr:
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
    print "Error: Couldn't send data,",errorstr
else:
    print "Data sent successfully"
```

现在，不管是否发生异常，都将关闭连接，而外部 try 语句捕获并处理执行过程中引发的实际异常。

7.3.3　异常嵌套

异常总是由封装的异常处理器处理，这一点已经很清楚了。但是，如果异常处理器没有显式处理每一个可能的异常，会发生什么情况呢？

答案很简单：将异常传递到下一个最高层的异常处理器。例如，前面列举的第二个 try…finally 示例中，由 send_data 方法引发的异常将传递到下一个异常处理器(该例中为 try…except…else 语句)。

在脚本执行过程中，逻辑上异常会进入堆栈。因此，如果异常处理器捕获了异常，或者当前异常处理器无法处理该异常而将异常传递到下一级异常处理器，那么将会从堆栈中去除这些异常。考虑下面的脚本，该脚本对本章前面部分所列举的计算脚本做了修改：

```
def divide(x,y):
    try:
        result = x/y
    except ZeroDivisionError:
        print "Whoa!: You're trying to divide by zero and you
can't!"
        raise
    return result

def calc(x,y):
    return x*(divide(x,y))

try:
    print calc(1,0)
except TypeError:
    print "Whoa!: That doesn't look like a number!"
except:
    print "Whoa!: Couldn't complete the calculation"
```

运行该脚本将得到两个错误消息——其中一个来自于 divide 函数的异常处理器，该处理器会识别除以零的错误。处理器也会引发它自身的异常，然后将该异常传递给主异常处理器：

```
Whoa!: You're trying to divide by zero and you can't!
Whoa!: Couldn't complete the calculation
```

也可用下面的代码代替上例的主程序部分：

```
try:
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
    print calc('Hello',0)
    except TypeError:
    print "Whoa!: That doesn't look like a number!"
except:
    print "Whoa!: Couldn't complete the calculation"
```

现在本地异常处理器捕获该错误。注意顺序——在执行引发 ZeroDivisionError 异常的 divide 函数前，Python 就识别出 calc 函数引发的 TypeError 异常。

7.3.4　引发异常

可以使用 raise 语句显示地引发异常。其效果如同内部错误引发的异常。raise 语句的格式为：

```
raise [EXCEPTION [, DATA] ]
```

其中：EXCEPTION 为引发的异常名，DATA 为提供给异常处理器的额外数据。

EXCEPTION 也可以是内建异常中的一个，您可以使用已经定义的一个异常或异常类。DATA 按正常方式传递到异常处理器。

7.3.5　assert 语句

assert 语句是 raise 语句的简化形式，它类似于 C/C++中的 ASSERT()宏。assert 语句的工作机制类似于 raise 语句，但是，只有在解释器未优化代码时才能用 assert 语句引发异常(也就是说，用户未指定-O 优化标志)。assert 语句的一般格式为：

```
assert TEST, DATA
```

实质上等价于：

```
if __debug__:
    if not TEST:
        raise AssertionError, DATA
```

注意，在解释器优化代码时，无法使用__debug__符号。

7.4　内置异常

正如前面所见到的，Python 提供了许多基异常和异常类。在脚本中可使用并捕获这些异常以显示或识别错误。如果创建新异常(本章后面的“自定义异常”部分进行了描述)，应考虑使用内置异常作为基类。图 7-1 说明了异常系统的类结构。

7.4.1　Exception

Exception 是所有异常的根(root)类。注意，对从任何异常返回的参数上的字符串操作，都应该提供已发生错误的字符串表达式，而不管提供的参数数量或类型。要获得任何异常的独立的参数，可使用 except 语句的数据格式将参数名组成的元组传递到信息存放地：

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
Exception
    StandardError
        ArithmeticError
            FloatingPointError
            OverflowError
            ZeroDivisionError
        AssertionError
        AttributeError
        EnvironmentError
            IOError
            OSError
                WindowsError
        EOFError
        ImportError
        KeyboardInterrupt
        LookupError
            IndexError
            KeyError
        MemoryError
        NameError
            UnboundLocalError
        RuntimeError
            NotImplementedError
        SyntaxError
        SystemError
        SystemExit
        TypeError
        ValueError
            UnicodeError
```

图 7-1　Python 异常系统的类结构

```
try:
  pow(2,262)
except Exception,(args,):
      print args
```

另外，如果没有提供显式地由 tuple 成员组成的元组，如：

```
except Exception . args
    print args
```

那么，args 保存了异常返回值组成的元组。

7.4.2　StandardError

StandardError 用作所有内置异常的基类。它继承了 Exception 根类所提供的功能。

7.4.3　ArithmeticError

用于算术错误引起的异常，它是特殊的算术异常(OverflowError、ZeroDivisionError 或 FloatingPointError)中的一种。ArithmeticError 是这三种异常的基类，它显示一般的算术错误。由于它是基类，可使用此异常捕获上述 3 种特殊的算术错误。

7.4.4　AssertionError

assert 语句失败时引发 AssertionError 异常。

7.4.5　AttributeError

当属性引用或赋值失败时引发 AttributeError 异常。注意，如果对象类型不支持属性，则引

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

发 TypeError 异常。

7.4.6　EnvironmentError

EnvironmentError 类用于处理 Python 控制过程外部发生的错误，也可跟踪 Python 的操作环境。EnvironmentError 是 IOError 和 OSError 异常的基类。

该异常返回的标准参数是两元素或三元素的 tuple。在两元素的 tuple 中，第一个元素为操作系统返回的错误号(errno)，第二个元素为相关错误字符串。在三元素的 tuple 中，第三个元素为异常发生时使用的文件名。

例如：

```
try:
   open('nosuchfile')
except EnvironmentError,(errno,string):
   print "Whoops!: %s (%d)" % (string, errno)
```

7.4.7　EOFError

当内置数据处理函数检测文件结束(eof)条件时，引发 EOFError 异常。注意，只有在未从源文件读取任何数据就检测文件结束条件时，才引发 EOFError 异常。还需注意，内置的 read 和 readline 方法检测文件结束条件时返回空字符串。

7.4.8　FloatingPointError

当浮点数操作失败时，引发 FloatingPointError 异常。当解释器在启用浮点信号处理功能的情况下进行编译时，才可使用 FloatingPointError 异常。如果在没有启用浮点信号处理功能时进行编译，引发的异常为 ArithmeticError。

7.4.9　ImportError

如果 import 语句没有找到指定模块，或者 from 语句没有在模块中找到指定符号，那么将引起 ImportError 异常。关于 import 方法和语法的其他信息，请参阅第 5 章。

7.4.10　IndexError

试图在序列大小范围之外访问序列元素时引发 IndexError 异常。注意，如果通过普通的下标符号访问无序列对象中的元素，那么返回 TypeError 异常。

7.4.11　IOError

I/O 操作失败时引发 IOError 异常。例如，试图打开不存在的文件或者对没有空闲空间的设备进行写操作。该异常提供的信息和由基于 EnvironmentError 类的异常提供的信息相同。

7.4.12　KeyError

在映射对象中不存在所请求的字典或其他映射键时，引发 KeyError 异常。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

7.4.13　KeyboardInterrupt

按下中断组合键(PC 中为 Ctrl-C，Mac 中为 Command-)时，引发 KeyboardInterrupt 异常。甚至在调用了内置的 input 或 raw_input 函数后，还会引发 KeyboardInterrupt 异常。

7.4.14　LookupError

LookupError 异常是内置的 IndexError 和 KeyError 异常的基类。该异常用于显示从序列(字符串、列表或 tuple)或映射(字典)访问信息时出现的错误。

7.4.15　MemoryError

如果解释器执行特定选项时耗尽所有内存，但依旧认为删除一些对象来释放内存后可以恢复系统，则引发 MemoryError 异常。可能无法从该状态进行恢复，但引发异常后会触发程序的堆栈跟踪。异常传递的数据描述了触发该异常的内部操作类型(存在例外情况)。

7.4.16　NameError

在局部或全局作用域中无法找到指定对象时，引发 NameError 异常。该异常传递的数据显示了无法找到的对象的名称。

7.4.17　NotImplementedError

当无法找到抽象的、用户自定义的错误所需要的方法时，引发 NotImplementedError 异常。该异常从 RuntimeError 异常派生。

7.4.18　OSError

操作系统发生错误时，引发 OSError 异常。通常通过 os 模块接口引发 OSError 异常。该异常从 EnvironmentError 异常派生。

7.4.19　OverflowError

当算术操作超出了 Python 解释器的限定范围时，引发 OverflowError 异常。注意，当进行长整数计算时解释器引发的异常为 MemoryError，而非 OverflowError。

7.4.20　RuntimeError

当出现用其他异常类型无法表示的运行时(run-time)错误时，引发 RuntimeError 异常。对于多数错误，均有对应的异常类，因此，RuntimeError 异常只是为了兼容的目的而存在。该异常传递的数据为：描述已发生错误的字符串。

7.4.21　SyntaxError

不管是在 import 或 exec 语句的原始脚本中，还是在内置的 eval 函数中，出现语法错误后都将引发 SyntaxError 异常。

SyntaxError 异常返回的信息为：包含错误消息的简单字符串。如果直接访问该异常对象，

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

则对象的字符串行中将包含 filename、lineno 和 offset 属性及字符串行的实际 text(文本)。

从细节来看，SyntaxError 异常传递的数据可以作为(message, (filename, lineno, offset, text))格式的 tuple 进行访问。例如，代码：

```
try:
    eval("print :")
except SyntaxError,(message,(filename,lineno,offset,text)):
    print "Error in line %d, from file %s: \n" % (lineno, filename),\
    text,"\n",\
            ' ' * offset+"^", message
```

执行后产生下列输出：

```
Error in line 1, from file None:
print :
      ^ invalid syntax
```

7.4.22　SystemError

发生系统错误时，引发 SystemError 异常。注意，该异常适用于这种类型的 Python 内部错误：可以安全地捕获这类错误并存在恢复系统的可能。SystemError 异常传递的数据为表示已发生错误的字符串。

注意：

因为 SystemError 异常表示了解释器中可能存在的错误，所以可以将 SystemError 异常信息发送给 Python 维护器(请参阅附录 B)。

7.4.23　SystemExit

调用 sys.exit()函数时，引发 SystemExit 异常。通常，Python 解释器不会输出任何错误或堆栈跟踪信息就直接退出系统。

7.4.24　TypeError

将内置操作或函数应用于类型不匹配的对象时，引发 TypeError 异常。返回值为字符串。该字符串提供了关于不匹配类型的细节情况。

7.4.25　UnboundLocalError

如果将引用变为函数或方法作用域内的局部变量，而引用值不在该变量范围内时，引发 UnboundLocalError 异常。

7.4.26　UnicodeError

发生与 Unicode 相关的编码或解码错误时，引发 UnicodeError 异常。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

7.4.27　ValueError

如果内置操作或函数接收了类型正确但值不正确的参数，而且又没有更加准确的异常(如IndexError)来描述这种情况时，引发 ValueError 异常。

7.4.28　WindowsError

发生特定于 Windows 的错误时，引发 WindowsError 异常。如果无有效的 errno 值与返回的错误号相匹配，也引发 WindowsError 异常。该返回的实际值使用 Windows API 调用 GetLastError 和 FormatMessage 生成。

7.4 29　ZeroDivisionError

当除法或求模操作的第二个参数为零时，引发 ZeroDivisionError 异常。返回值为字符串。该字符串描述操作数和操作的类型。

7.5　自定义异常

通过创建从系统异常类继承的新异常类，可得到自定义异常。为保持反向兼容性，也可创建基于字符串的异常，如下例所示：

```
CustomError = 'Error'

raise CustomError
```

注意，上例中，异常与值不匹配，因此，引发了字符串异常。但是，检查对象后发现对象依然有效：

```
CustomError = 'Error'

def test():
  raise 'Error'

try:
  test()
except CustomError:
  print "Error!"
```

但是，这样做不好。应该使错误引起一个预定义的错误字符串。

要创建基于类的异常，可以创建包含初始化信息的新类。然后，引发带有额外信息的异常，以便异常处理器进行分析。except 语句中的额外数据可从异常对象的 args 属性中提取。如下例所示：

```
class MyNameException(Exception):
    def __init__(self, name, msg):
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
        self.args = (name, msg)

try:
    raise MyNameException('Marvin', 'Not my real name')
except MyNameException,(name, msg):
    print 'Sorry, but',name,'is',msg
```

本书其他部分还有许多示例，这些示例描述了基于对象/类的异常。

注意：

虽然 Python 没有强制规定必须将自定义异常类以内置异常类为基类，但还是应该这样做。因为这样做以后，使用自定义代码和异常的程序员就可使用一般的、更高层的类中的方法捕获问题，而不需显式指定使用自定义异常类。基于字符串的异常不在 Python 的异常继承系统范围内，因此，应避免使用基于字符串的异常。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第 2 部分　应用 Python 库

第8章　Python内置函数

Python 的各种功能都以其标准形式由一系列外部模块实现。这些外部模块囊括了从获取命令行参数、获取系统配置和信息的基本机制到与 SMTP 和 HTTP 服务器通话、与其他系统和 API 通话的复杂模块等所有应用。

虽然已有一个构成 Python 标准函数的大库，但仍需一些内置函数来创建、处理或确定有关构成 Python 语言的对象、类型和类的各种信息。本章集中讨论这些内置函数，而本部分的其余章节则全力以赴地研究那些 Python 标准库内的特殊模块。

注意

本章信息以 Python 2.0 有效文档为基础。随 Python 提供的全部文档的更新版本可在 Python 和 Mcwords 网站上找到。详见附录 A。

本部分的函数是__builin__模块的一部分。

8.1　__import__(name[,globals[,locals[,formlist]]])

__import__()函数由 import 语句自动激活。例如语句：

```
import module
```

产生如下对__import__()的调用：

```
__import__ ('module',globals(),locals(),[])
```

而下列调用

```
from moduide import calss
```

产生如下对__import__()的调用：

```
__import__ ('module',globals(),locals(),['class'])
```

之所以提供__import__()这个函数，是为了便于有可能用自己编写的导入函数替换它。更多的举例，参见 ihooks 和 rexec 模块。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

8.2　abs(x)

abs()函数返回一数字(可为普通整型、长整型或浮点型)的绝对值。如果给出复数，返回值就是该复数的模。例如：

```
>>> print abs(-2.4)
2.4
>>> print abs(4+2j)
4.472135955
```

8.3　apply(function,args[,keywords])

apply()函数将 args 参数应用到 function 上。function 参数必须是可调用对象(函数、方法或其他可调用对象)。args 参数必须以序列形式给出。列表在应用之前被转换为元组。function 对象在被调用时，将 args 列表的内容分别作为独立的参数看待。例如：

```
apply(add,(1,3,4))
```

等价于

```
add(1,3,4)
```

在以列表或元组形式定义了一列参数，且需将此列参数分别作为个个独立参数使用的情况下，必须使用 apply()函数。在要把变长参数列应用到一函数上时，apply()函数非常有用。

可选项 keywords 参数应是个字典，字典的关键字是字符串，这些字符串在 apply()函数的参数列末尾处给出，它们将被用作关键字参数。

8.4　buffer(object[,offset[,size]])

如果 object 对象支持缓存调用接口，buffer()函数就为 object 对象创建一个新缓存。这样的对象包括字符串、数组和缓存。该新缓存通过使用从 offset 参数值开始直到该对象末尾的存储片断或从 offset 参数值开始直到 size 参数给出的尺寸为长度的存储片段来引用 object 对象。如果没给出任何选项参数，缓存区域就覆盖整个序列。最终得到的缓存对象是 object 对象数据的只读拷贝。

缓存对象用于给某个对象类型创建一个更友好的接口。比如，字符串对象类型通用缓存对象而变得可用，允许逐个字节地访问字符串中的信息。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

8.5　callable(object)

callable()函数在 object 对象是可调用对象的情况下，返回真(true);否则，返回假(false)。可调用对象包括函数、方法、代码对象、类(在调用时返回新的实例)和已经定义了“调用”方法的类实例。

8.6　chr(i)

chr()函数返回与 ASCII 码 i 相匹配的一个单一字符字符串，如下例所示：

```
>>>print chr(72)+chr(101)+chr(108)+chr(111)
Hello
```

chr()函数是 ord()函数的反函数，其中 ord()函数将字符转换回 ASCII 整数码。参数 i 的取值应在 0~255 范围内。如果参数 i 的取值在此范围之外，将引发 ValueError 异常。

8.7　cmp(x,y)

cmp()函数比较 x 和 y 这两个对象，且根据比较结果返回一个整数。如果 x<y，则返回负数；如果 x= =y，则返回值为 0；如果 x>y，则返回正数。请注意，此函数特别用来比较数值大小，而不是比较任何引用关系，因而有下面的结果：

```
>>> a=99
>>> b=int('99')
>>> cmp(a,b)
0
```

8.8　coerce(x,y)

coerce()函数返回一个元组，该元组由两个数值型参数组成。此函数将两个数值型参数转换为同一类型数字，其转换规则与算术运算转换规则一样。以下是两个例子：

```
>>> a=1
>>> b=1.2

>>> coerce(a,b)
(1.0,1.2)
>> a=1+2j
>> b=4.3e10
>>> coerce(a,b)
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
((1+2j),(43000000000+0j))
```

8.9　compile(string,filename,kind)

compile()函数将 string 编译为代码对象，编译生成的代码对象接下来能被 exec 语句执行，接着能利用 eval()函数对其进行求值。filename 参数应是代码从其中读出的文件名。如果内部生成文件名，filename 参数值应是相对应的标识符。Kind 参数指定 string 参数中所含代码的类别。有关 kind 可能取值的详细信息，请参见表 8-1。

举例如下：

```
>>> a=compile('print "Hello World" ', '<string> ', 'single ')
>>> exec(a)
Hello World
>>> eval(a)
Hello World
```

表 8-1　由 compile()函数编译的代码的类别

Kind 取值	编译生成的代码
exec	语句序列
eval	简单表达式
single	简单交互语句

8.10　complex(real[,imag])

complex()函数返回一个复数，其实部为 real 参数值。如果给出 imag 参数的值，则虚部就为 imag；如果默认 imag 参数，则虚部为 0j。

8.11　delattr(object, name)

delattr()函数在 object 对象许可时，删除 object 对象的 name 属性，此函数等价于如下语句：

```
del object.attr
```

而 delattr()函数允许利用编程方法来定义 object 和 name 参数，并不是在代码中显式指定。

8.12　dir([object])

当没有提供参数时，dir()函数列出在当前局部符号表中保存的名字，如下例所示：

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
>>> import smtplib, sys,  os
>>> dir()
['__builtins__', '__doc__', '__name__', 'os', 'smtplib', 'sys']
```

当给出一个参数时，dir()函数返回那个对象的一列属性。此函数对确定定义在模块内的对象和方法非常有用，如下例所示：

```
>>> import sys
>>> dir(sys)
['__doc__', '__name__', '__stderr__', '__stdin__', '__stdout__',
'argv', 'builtin_module_names', 'byteorder', 'copyright',
'exc_info', 'exc_type', 'exec_prefix', 'executable', 'exit',
'getdefaultencoding', 'getrecursionlimit', 'getrefcount',
'hexversion', 'maxint', 'modules', 'path', 'platform', 'prefix',
'ps1', 'ps2', 'setcheckinterval', 'setprofile',
'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout',
'version', 'version_info']
```

这些信息是来自于给定对象的__dict__、__methods__和__members__属性。这些信息可能是不完整的，比如从其他类继承过来的方法和属性一般就不包含于其中。

8.13　divmod(a, b)

divmod()函数返回一个元组，该元组包含 a 除以 b 的商和余数，如下例所示：

```
>>> divmod(7,4)
(1,3)
```

对整数而言，返回值与 a/b 和 a%b 相同。如果给出的参数值是浮点数，则结果就是(q,a%b)，其中：q 通常是 math.floor(a /b)，但也可能比这小 1，不管在什么情况下，q*b+a%b 都非常逼近 a；如果 a%b 是个非零值，则其正负符号与 b 相同，并且有 0<=abs(a%b)<abs(b)成立。下面的例子显示了 divmod()函数是如何处理浮点数的：

```
>>> divmod(3.75,1.125)
(3.0,0.375)
>>> divmod(4.99,1.001)
(4.0,0.98600000000000065)
>>> divmod(-3.5,1.1)
(-4.0,0.90000000000000036)
```

8.14　eval(expression[,globals[,locals]])

eval()函数将 expression 字符串作为 Python 标准表达式进行分析并求值，返回 expression 字

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

符串的值。当不调用其他可选参数时，expression 访问调用该函数的程序段的全局和局部对象。另一个选择是：以字典形式给出全局和局部符号表(参见本章后面部分对 globals()和 locals()函数的论述)。

eval()函数的返回值是被求值表达式的值，如下例所示：

```
>>> a=99
>>> eval('divmod(a,7)')
(14,1)
```

任何求值操作的语法错误，都将引发成异常。

eval()函数还能用来编译诸如由 compile()函数创建的代码对象，但仅当该代码对象用“eval”模式编译过后才可用 eval()函数编译。

要执行混合了语句和表达式的 Python 任意代码，请使用 exec 语句或使用 execfile()函数来动态地执行含有任意代码的文件。详见本章末尾处 “执行任意语句(arbitrary statement)”小节。

8.15　execfile(file [, globals [, locals]])

execfile()函数与 exec 语句等价，不同之处在于：execfile()函数执行文件中的语句，而 exec 语句处理字符串。其中 globals 和 locals 参数应是字典，该字典包含文件在执行期间有效的符号表；如果 locals 参数省略，则所有的引用都使用 globals 名称空间。如果两个可选参数都省略，文件就访问运行期间的当前符号表。

8.16　filter(function, list)

filter()函数根据 function 参数返回的结果是否为真(true)来过滤 list 参数中的项，最后返回一个新列表，如下例所示：

```
a=[1,2,3,4,5,6,7,8,9]
b=filter(lambda x: x > 6, a)
print b
[7,8,9]
```

如果 function 参数值为 None，就使用 identity 函数，list 参数中的所有为假(false)的元素都被删除。

8.17　float(x)

float()函数将 x 参数转换为浮点数，其中：x 可以是字符串，也可以是数字。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

8.18　getattr(object, name [, default])

getattr()函数返回 object 的 name 属性值。在语法上，以下语句：

```
getattr(x,'myvalue')
```

等价于

```
x.myvalue
```

如果 name 参数不存在，但给出 default 参数的值，则此函数就返回 default 参数值；否则引发 AttributeError 异常。

8.19　globals()

globals()函数返回一个表示当前全局符号表的字典。这个字典通常就是当前模块的字典。如果 globals()函数是在一函数或方法中被调用，它就返回定义该函数或方法的模块的符号表，而不是调用此函数的模块的符号表。

8.20　hasattr(object, name)

如果 object 对象具有与 name 字符串相匹配的属性，hasattr()函数返回真(true)；否则返回 0 。

8.21　hash(object)

hash()函数返回关于 object 对象的整数散列值。如任何两个对象比较起来是等价的，则它们的散列值是一样的。此函数不应用于可变对象上。

8.22　hex(x)

hex()函数将一整数转换为十六进制字符串，该字符串是个有效的 Python 表达式。

8.23　id(object)

id()函数返回值为一个整数(或长整型整数)—— 该对象的“标识”—— 该标识在其对应对象的生命期内，确保是惟一的和恒定不变的。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

8.24　input([prompt])

input()函数与 eval(raw_input(prompt))等价。详细信息参见本章后面部分对 raw_input()函数的论述。

8.25　int(x [, radix])

int()函数将使数字或字符串 x 转换为“普通”整数。如果给出 radix 参数的值，则 radix 参数值用作转换的基数，该参数应是 2~36 范围内的一个整数。

8.26　intern(string)

intern()函数将 string 加入到保留字符串的表，返回值为保留的版本号。“保留字符串”通过指针可用，而不是一个纯的字符串；因此允许利用指针比较代替字符串比较来进行字典关键字的查找，这比通常的字符串比较方法功能有所改善。

在 Python 名称空间表和用于保存模块、类或实例属性的字典中使用的名字通常被保留用以加速脚本执行。

保留字符串定义后不能被作为无用单元收集，所以必须注意在大字典关键字集上使用保留字符串将大大增加内存需求，即使字典关键字已经超出了作用域。

8.27　isinstance(object, class)

isinstance()函数在 object 参数是 class 参数的一个实例时，返回真(true)。函数值的确定服从普通继承法则和子类。如果 object 参数是在 types 模块中利用类型类定义的特殊类型的实例，也能用 isinstance()函数来识别。如果 class 参数不是类，也不是类型对象，就引发 TypeError 异常。

8.28　issubclass(class1, class2)

如果 class1 参数是 class2 参数的子类，issubclass()函数则返回真。类通常被认为是其自身的子类。若两个参数中任一个都不是类对象，则引发 TypeError 异常。

8.29　len(s)

len()函数返回一序列(字符串、元组或列表)或字典对象的长度。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

8.30　list(sequence)

list()函数返回一列表。该列表的项及顺序与 sequence 参数的项及顺序相同，如下例所示：

```
>>> list('abc')
['a', 'b', 'c']
>>> list([1,2,3])
[1, 2, 3]
```

8.31　locals()

locals()函数返回表示当前局部符号表的字典。

8.32　long(x)

long()函数将字符串或数字转换为长整形数。对浮点数的转换遵循与 int()相同的规则。

8.33　map(function, list, ...)

map()函数将 function 运用到 list 中的每一项上，并返回新列表，如下例所示：

```
>>> a=[1,2,3,4]
>>> map(lambda x: pow(x,2), a)
[1,4,9,16]
```

若提供附加的列表，则它们就被并行地提供给 function。在后续无元素的列表增加 None，直到所有参数列表达到相同的长度为止。

如果 function 参数值为 None，则假定为 identity 函数，将使 map()函数返回删除所有值为假的参数的 list。如果 function 参数值为 None，且给定多个列表参数，返回的列表由一个个元组组成，这些元组由函数中的每个参数列表内相同对应位置上的参数组成，如下例所示：

```
>>> map(None, [1,2,3,4], [4,5,6,7])
[(1, 4), (2, 5), (3, 6), (4, 7)]
```

上例的结果与 zip()函数产生的结果等价。

8.34　max(s [, args...])

当仅给定一个参数时，max()函数返回序列 s 的最大值。当给定一列参数时，max()函数返

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

回给定参数的最大参数。详细内容参见 min()函数。

8.35　min(s [, args...])

当仅给定一个参数时，min()函数返回序列 s 的最小值。当给定一列参数时，min()函数返回给定参数中的最小值。记住：多参数调用的序列不被遍历，每个列表参数作为一个整体进行比较，如：

```
min([1,2,3],[4,5,6])
```

返回

```
[1, 2, 3]
```

而并不是通常所想的结果为 1。要得到一个或多个列表中元素的最小值，可将所有列表连成一串。如下所示：

```
min([1,2,3]+[4,5,6])
```

8.36　oct(x)

oct()函数将整数转换为八进制字符串。其结果是个有效的 Python 表达式，如下例所示：

```
>>>oct(2001)
'03721'
```

请注意，返回值通常是无符号数。这样致使 oct(－1)在 32 位机器上产生'037777777777'的结果。

8.37　open(filename [, mode [, bufsize]])

open()函数通过使用 mode 和缓存 bufsize 类型来打开 filename 标识的文件。此函数返回一文件对象。(详见有关文件对象内容的第 3 章和第 6 章)。

其中，mode 与系统函数 fopen()使用的模式相同。表 8-2 列出了有效的打开模式。如果 mode 参数省略，其默认取值为 r。

表 8-2　open()函数的文件打开模式

模　式	含　义
R	打开用于读
w	打开用于写

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

模　式	含　义
a	打开用于附加(在打开期间，文件位置自动移到文件末尾处)
r+	打开用于更新(读和写)
w+	截断(或清空)文件，接着打开文件用于读和写
a+	打开文件用于读和写,并自动改变当前文件位置到文件尾
b	当附加于任何模式选项时，以二进制模式而不是文本模式，打开文件(这种模式仅对 Windows、DOS 和其他一些操作系统有效；而 Unix、MacOS 和 BeOS 则不管此选项为何值，以二进制模式对待所有的文件)

open()函数的 bufsize 选项参数决定从文件中读取数据时所使用的缓存的大小。表 8-3 列出了所支持的 bufsize 值。如果参数 bufsize 省略，就使用系统默认的缓存容量。

表 8-3　open()函数支持的 bufsize

bufsize 值	说　明
0	禁用缓存
1	行缓存
>1	使用大小近似为 bufsize 字符长度的缓存
<0	使用系统默认(对于 tty 终端设备而言就是行缓存；对其他所有文件而言就是全缓存)

8.38　ord(c)

ord()函数返回由一个字符 c 组成的字符串的 ASCII 码值或 Unicode 数字码。ord()函数是 chr()函数和 unichr()函数的反函数。

8.39　pow(x, y [, z])

pow()函数返回以 x 为底数以 y 为指数的幂值。如果给出 z，该函数就计算 x 的 y 次幂值被 z 取模的值。这样的计算比利用：

```
pow(x,y) % z
```

的效率更高。

提供给 pow()函数的参数应当是数值类型，并且给定数值的类型决定返回值的类型。如果计算得出的数值不能用给定参数值的类型表示，则引发异常。比如，以下对 pow()的调用将失败：

```
pow(2, -1)
```

但是

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
pow(2.0, - 1)
```

是有效的。

8.40　range([start,] stop [, step])

range()函数返回数值列表，该数值列表从 start 开始，以 step 为步长，于 stop 之前结束。所有的数字都应列出，并且以普通整型数返回。如果 step 省略，则 step 的默认取值为 1。如果 start 省略，则返回列表序列从 0 开始求值。请注意，若以两个参数的形式调用此函数，则认作给定的参数是 start 和 stop；如果要定义步长 step，就必须给出全部三个参数。下面对 range()函数的调用使用了值为正数的步长 step：

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5,10)
[5, 6, 7, 8, 9]
>>> range(5,25,5)
[5, 10, 15, 20]
```

请注意，最后的数值是 stop 减去 step，range()函数的返回值从小递增到大，趋近 stop 的值，但不包含 stop 这个值。

如果 step 的给定值是负数，range()函数的返回值从大递减到小的，而不是递增。stop 必须比 start 小；否则返回的列表为空。下例说明了 step 取值为负数的运用情况：

```
>>> range(10,0,-1)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> range (25,0,-5)
[25, 20, 15, 10, 5]
>>> range(0,10,-1)
[]
```

8.41　raw_input([prompt])

raw_input() 函数从 sys.stdin 接受原始输入并返回字符串。输入以换行符为结束，其中换行符在输入字符串返回给调用者之前被去除。如果给出 prompt，末尾不含换行符的 prompt 就被写到 sys.stdout 中，并用作输入的提示，如下例所示：

```
>>> name=raw_input('Name? ')
Name? Martin
```

如果已加载 readline 模块，则诸如行编辑和历史记录的特性在输入期间就得到支持。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

8.42　reduce(function, sequence [, initializer])

reduce()函数依次应用 function (支持两个参数)到 sequence 中的每个元素上，逐渐缩短整个语句直到为一个单一的值。举例，下面的语句模拟了算术运算符“!”：

```
reduce(lambda x,y: x*y, [1,2,3,4,5])
```

其结果如同执行以下计算一样：

```
((((1*2)*3)*4)*5)
```

结果等于 120。

如果给出 initializer 参数的值，则 initializer 参数值就被用作序列的第一个元素，如下例所示：

```
>>> reduce(lambda x,y: x*y, [1,2,3,4,5],10)
1200
```

8.43　reload(module)

reload()函数将以前导入过的模块再加载一次。重新加载(reload)包括最初导入模块时应用的分析过程和初始化过程。这样就允许在不退出解释器的情况下重新加载已更改的 Python 模块。

使用 reload()函数的若干注意事项如下：

- 如果模块在语法上是正确的，但在初始化过程中失败，则导入过程不能正确地将模块的名字绑定到符号表中。这时，必须在模块能被重新加载之前使用 import()函数加载该模块。
- 重新加载的模块不删除最初旧版本模块在符号表中的登记项。对于有恒定名字的对象和函数，这当然不是个问题；但是，若对一模块实体更改了名字，模块名在重新加载后仍保持在符号表中。
- 支持扩展模块(它依赖于内置的或所支持的动态加载的函数库)的重新加载，但可能是无目标的，并且确实可能导致失败，这完全依赖于动态加载的函数库的行为。
- 如果一模块利用 from...import...方式从另一模块导入对象，reload()函数不重定义导入的对象。可利用 import...形式避免这个问题。
- 提供类的重新加载模块不影响所提供类的任何已存实例——已存实例将继续使用原来的方法定义；只有该类的新实例使用新格式。这个原则对派生类同样适用。

8.44　repr(object)

repr()函数返回对象的字符串表示。这与将对象或属性使用单反引号(`)的结果是一致的。返回的字符串产生一个对象，该对象的值与将 object 传递给 eval()函数产生的值一样，如下例所示：

```
>>> dict = {'One':1, 'Two':2, 'Many': {'Many':4, 'ManyMany':8}}
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
>>> repr(dict)
"{'One': 1, 'Many': {'Many': 4, 'ManyMany': 8}, 'Two': 2}"
```

8.45　round(x[, n])

round()函数返回浮点型参数 x 舍入到十进制小数点后 n 位的值，如下例所示：

```
>>> round(0.4)
0.0
>>> round(0.5)
1.0
>>> round(-0.5)
-1.0
>>> round(1985,-2)
2000.0
```

8.46　setattr(object, name, value)

setattr()函数将 object 参数的 name 属性设置为 value 参数值。setattr()函数是 getattr()函数的反函数，后者仅获取信息。以下语句：

```
setattr(myobj, 'myattr', 'new value')
```

等价于

```
myobj.myattr = 'new value'
```

setattr()函数能用在这样的情况下：属性是通过 name 参数以编程方式命名，而不是显式地命名属性。

8.47　slice([start,] stop [, step])

slice()函数返回一序列切片(slice)对象，该对象表示由 range(start, stop, step)指定的索引集。如果给出一个参数，此参数就作为 stop 参数值；如果给出两个参数，它们就作为 start 和 stop 的参数值；任何未给出参数值的参数默认取值为 None。序列切片对象有 3 个属性(start、stop 和 step)，这 3 个属性仅仅返回要提供给 slice()函数的参数。

8.48　str(object)

str()函数返回对象的一个字符串表示。这与 repr()函数相似，惟一不同之处在于：此函数返

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

回值设计为可打印字符串而不是与 eval()函数兼容的字符串。

8.49　tuple(sequence)

tuple()函数返回一个元组，该元组的项及项的顺序与 sequence 参数完全一样。以下就是 tuple()函数的举例：

```
>>> tuple('abc')
('a', 'b', 'c')
>>> tuple([1,2,3])
(1, 2, 3)
```

8.50　type(object)

type()函数返回 object 参数的类型。返回值是个如类型模块所描述一样的类型对象。举例如下：

```
>>> import types
>>> if type(string) == types.StringType:
print "This is a string"
```

8.51　unichr(i)

unichr()函数返回代码是一个整型参数 i 的 Unicode 字符的 Unicode 字符串。此函数等价于本章前面部分论述的 chr()函数。请注意，要将 Unicode 字符转换回其整数格式，可使用 ord()函数；没有 uniord()函数。如果给出的整数超出 0~65535 这个范围，则引发 ValueError 异常。

8.52　unicode(string [, encoding [, errors]])

unicode()函数利用编码格式编码解码器将给定的字符串从一种格式解码为另一种格式。编码的任何错误都用 errors 参数定义的字符串标记。

此函数特别用于在字符串和 Unicode 编码格式之间转换。默认(当不给出 encoding 参数的值)操作是以严格方式将字符串解码为 UTF-8 格式，发生 errors 错误时就引发 ValueError 异常。有关合适的解码列表，请见 codecs 模块。详见有关“在 Python 中 Unicode 是如何起作用”的第 10 章。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

8.53　vars([object])

vars()函数返回对应于当前局部符号表的字典。当给出模块、类或类实例时，vars()函数返回对应那个对象的符号表的字典。因为结果是非定义的，所以一定不要修改返回的字典。

8.54　xrange([start,] stop [, step])

xrange()函数的作用与 range()函数一样，惟一的区别是：xrange()函数返回一个 xrange 对象。xrange()对象是个不透明对象类型，此类型返回的信息与所请求的参数列表是一致的，但是它不必存储列表中每个独立的元素。在创建非常巨大列表的情况下，此函数特别有用；利用 xrange()函数节省下来的内存比起使用 range()函数是相当可观的。

8.55　zip(seq1, ...)

zip()函数处理一系列序列，将这些序列返回为一个元组列表，其中，每个元组包含了给定的每个序列的第 n 个元素。以下是个例子：

```
>>> a=[1,2,3,4]
>>> b=[5,6,7,8]
>>> zip(a,b)
[(1, 5), (2, 6), (3, 7), (4, 8)]
```

8.56　执行任意语句

Python 支持 3 条指令，这 3 条指令允许执行一些任意文件或 Python 代码的字符串。这对执行不论是通过脚本动态建立的语句，还是由用户提供的语句都是有益的。这 3 条指令是 exec 语句、execfile()和 eval()函数。因为每条语句被设计为执行特定的功能，所以不管如何都要确保使用正确的指令来执行代码或程序段。

8.57　exec 语句

exec 语句被设计为执行能使用函数和语句的任意组合的 Python 的任何代码片段。执行的代码访问相同的全局定义和局部定义的对象、类和方法或函数。以下是使用 exec 语句的简单例子：

```
exec "print 'Hello World'"
```

也能通过提供一个包含对象及其取值的列表的字典来限定对 exec 语句有效的资源，如下例这样：

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
exec "print message" in mynamespace
```

其中：mynamespace 是要使用的字典。

也能利用如下的语句显式地提供全局和局部字典的名称空间：

```
exec "print message" in myglobals, mylocals
```

能用 globals()和 locals()函数来获得当前表的字典。

请注意，exec 语句执行表达式和语句、或者对表达式和语句求值，但是 exec 语句不返回任何值。因为 exec 是语句不是函数，所以任何获取返回值的试图都将导致语法错误，如下例所示：

```
>>> a=exec '3+4'
File "<stdin>", line 1
a=exec '3+4'
^
SyntaxError: invalid syntax
```

8.58　execfile()函数

execfile()函数执行与 exec 语句同样的操作，正如本章前面部分所描述的那样，它们的不同之处在于：execfile()函数从文件中读取被执行的语句，执行的对象不是字符串，不是代码对象；execfile()函数的其他所有方面都与 exec 语句等价。

8.59　eval()函数

eval()函数不允许执行任意的 Python 语句。eval()函数被设计为：执行一个 Python 表达式，并返回值，如下例中一样：

```
result=eval(userexpression)
```

或者在语句中更显式地给出表达式，如下例所示：

```
result=eval("99+45")
```

不能使用 eval()函数去执行语句，如下例所示：

```
>>> a=eval('print "Hello world"')
Traceback (most recent call last):
File "<stdin>", line 1, in ?
File "<string>", line 1
print "Hello world"
^
SyntaxError: invalid syntax
```

根据经验，通常使用 eval()函数来将一表达式求值并返回一个值，而在其他所有情况下则使用 exec 语句。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第9章　与操作系统的接口

任何编程语言的最基本操作是与它所运行的操作系统交互操作。对像 Unix 和 Windows 这类基于命令行的操作系统来说，与其交互操作的功能必须包括从命令行中提取所提供的其他元素和文本。对 Python 而言，要完成上述交互功能，实际地分析命令行中的内容也许是个好主意。

除了上述基本操作外，还有其他更复杂的操作。例如开始新进程、执行外部程序及在支持多线程的系统上处理多线程操作。

Python 在 sys 和 os 模块中为上述绝大部分操作提供了一个统一的入口点，还有一些模块用于处理其他系统，例如 time 模块和 getopt 模块。本章将讨论上述所有模块，并对本部分中提供类似系统操作的其他章节给予提示。

9.1　使用系统(sys 模块)

从严格意义上讲，sys 模块提供了与 Python 所运行的系统直接通信的工具。虽然 sys 模块的确提供了一些通用功能(包括访问命令行变量)，但它的初衷还是帮助程序员确定其脚本所运行的系统环境。

9.1.1　获取命令行参数

在 Python 中可通过 sys.argv 数组获得命令行参数。像其他数组一样，下标从零开始。第一个元素保存命令行脚本的名称，从下标为 1 的元素开始依次保存提供给脚本的所有参数。

例如，如下所示的脚本：

```
import sys
count=0
for argument sys.argv:
    print "Argument %d is %s" % (count,argument)
    count += 1
```

执行后，输出如下结果：

```
$ python argv.py hello this is a test
Argument 0 is argv.py
Argument 1 is hello
Argument 2 is this
Argument 3 is is
Argument 4 is a
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
Argument 5 is test
```

识别出命令行脚本的名称是很有用的。特别是在要将错误信息报告给程序用户的情况下，就更需要脚本名了。

分析命令行参数

像其他编程语言一样，有时可能需要给命令行脚本提供可供分析的参数，例如：

```
Process.py –y –g --debug-- output=process.out myfile.txt
```

在这种情况下，需要使用 getopt 模块。这个模块提供了一个单一的名为 getopt 的函数来处理命令行中的参数，以更方便的形式来存放信息，此函数的基本格式如下：

```
getopt(args,options [,long_options])
```

其中：args 参数是要分析的参数列表；而 options 参数是个字符串，它所包含的内容是要解释的单字母参数。如果在 options 字符串中的字母后插入一个冒号，那么这个参数将接受一个附加参数作为冒号前参数的数据。如果给出 long_options 选项参数，它应该是定义单词的字符串列表，而不是需要被标识和分析的单个字母。请注意，在参数列表中，基于单词的参数必须是以双连字符而不是单连字符作为前缀。在 long_options 参数列表中对其中一个参数加上“=”后缀导致 getopt 函数将后续的参数解释为附加数据。

getopt 函数返回两个对象。第一个对象是一个由元组组成的列表，每个元组包含已分析的参数和此参数的值(假设参数只有一个值)；第二个对象是保持未被分析参数的列表。

例如：

```
>>> import getopt
>>> args = ['-a','-x','.bak','--debuglevel','99','file1','file2']
>>> opts, remargs = getopt.getopt(args,'ax:', ['debuglevel='])
>>> opts
[('-a', ''), ('-x', '.bak'), ('--debuglevel', '99')]
>>> remargs
['file1', 'file2']
```

getopt 函数在以下情况出现时就引发异常(GetoptError)：在参数列表中有不能识别的选项出现，或者参数列表中某个参数需要附加数据却未接收到附加数据。

9.1.2　标准文件句柄

所有平台上的任何进程都隐含支持 3 个标准文件句柄，这些文件句柄用于与标准输入(键盘或终端)、标准输出(监控器或终端)及标准错误(通常是监控器 / 终端)通信。对于 Unix 用户而言，能将这些句柄当作<、 >和 2 个>重定向运算符；在 Mac OS 系统下， 这些文件句柄由 Python 解释器模拟。

能利用 sys.stdin、sys.stdout 和 sys.stderr 对象访问 Python 的标准文件句柄。 它们都是文件对象，因而必须使用第 10 章表 10-1 定义的文件方法。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

print 语句将其输出发送到与 sys.stdout 等价的对象上，所以语句：

```
print 'Busy doin nothin'
```

和

```
sys.stdout.write('Busy doin nothin\n')
```

产生的结果相同。

1. 重定向输出

经常用到：将标准文件句柄的输出重新定向到另一个新位置上。例如，如果想跟踪一脚本的执行情况，且假定正使用 sys.stderr 来将消息发送到常用错误信道上，则应在脚本内重定向输出。

要重定向任何文件句柄，必须按如下步骤打开文件：首先利用内置 open()函数创建一个新的文件对象，接着将其分配给新对象，再将新对象分配给一个标准文件句柄，或者像下例一样直接分配对象：

```
import sys
sys.stderr = open('error.log','w')
sys.stderr.write('Error log!\n')
sys.stderr.close()
```

2. 初始文件句柄

若重新分配 Python 的所有标准文件句柄，在脚本开始执行时的初始对象通过 sys.__stdin__对象、sys.__stdout__对象和 sys.__stderr__对象就可获得。

假设，已利用上例方法将输出重定向到另一个文件上，那么接着就可以利用如下语句将标准错误输出放回到标准输出上：

```
sys.stderr = sys.__stderr__
```

9.1.3　终止执行

当 Python 解释器发现脚本结束——再没有语句要执行，脚本自然终止，然后执行操作停止。也能用 sys.exit()函数强制终止脚本的执行，如下例所示：

```
Python 2.1 (#2, Apr 29 2001, 14:36:04)
[GCC 2.95.3 20010315 (release)] on sunos5
Type "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.exit()
$
```

可以给 sys.exit()函数提供一个可选参数，此参数既可以是整数，也可以是对象。如果是整数，按惯例，值为零表明执行成功，而非零整数值则指示一个错误。若此参数给定为对象，

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

则该对象被写入(利用 str()函数)sys.stderr 文件句柄中，且返回给调用者一个为 1 的数值。如果该对象是特殊值 None，则返回值为 0。

sys.exit()函数的实际用途是导致 Python 引发一个 SystemExit 异常。这强制兑现和执行所有未完成的 try 语句的所有 finally 子句，并为有可能在脚本最终终止之前执行清除操作(如关闭文件、关闭网络及关闭数据库连接等)而捕获异常。

注意:

实现终止执行脚本的另一个可选方法是实际地引发一个 SystemExit 异常，这将具有与调用 sys.exit()函数相同的效果。

9.1.4　跟踪终止

尽管能用 SystemExit 异常来捕获终止请求并执行一些清除功能。但最好还是使用 sys.exitfunc()函数。使用 sys.exitfunc()函数来定义 sys.exit()函数被调用时要调用的函数名。如下例所示：

```
import sys
def cleanup():
    print "Had enough, deciding to leave\n"
sys.exitfunc = cleanup
```

对于扩展系统，请使用 atexit 模块。详情见在线文档。

9.1.5　解释器信息

除了已论述的函数和结构以外，sys 模块还提供了到许多变量的一个接口，这些变量保存有关 Python 解释器的当前实例的各种信息。这些变量概括如表 9-1。

表 9-1　sys 模块所支持的解释器和其他变量

Python sys 模块变量	说　明
builtin_module_names	一个元组，它包含内置在 Python 可执行文件中的所有模块的名字
copyright	当前 Python 解释器的版权信息
exec_prefix	存放依赖平台的 Python 库文件的目录
Executable	返回保存当前 Python 解释器的可执行文件的路径
Maxint	Integer 对象类型所支持的最大整数
Platform	在安装／配置期间确定下来的平台标识字符串，如：'sunos5'和 'mac'
Prefix	存放与平台无关的 Python 库文件的目录
ps1, ps2	包含交互使用解释器时的主提示符和第二提示符的字符串。默认设置分别为>>>和...
Version	运行当前脚本的解释器的版本，返回值是诸如"2.0 (#71, Oct 22 2000, 22:09:24) [CW PPCw/GUSI2 w/THREADS]"格式的字符串。此字符串包含主版本号、版本修正号、建立日期及平台和解释器建立时配置的选项

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

9.1.6　模块搜索路径

Python 模块能在其编译期间确定下来的标准搜索路径下查到。例如，在 Solaris 8 下安装 Python 2.0 就产生如下清单：

```
['', '/usr/local/lib/python2.0',
'/usr/local/lib/python2.0/plat-sunos5',
'/usr/local/lib/python2.0/lib-tk',
'/usr/local/lib/python2.0/lib-dynload',
'/usr/local/lib/python2.0/site-packages']
```

此目录清单存储在 sys.path 数组对象中。如果想将其他目录添加到用 import 语句就能搜索到的目录清单中，则只需修改 sys.path 数组，如下例所示：

```
import sys
sys.path.append('/home/mc/lib')
import smtplib
```

要插入路径，使得插入的路径优先于标准路径，以使之首先被搜索到，请使用 insert() 方法，如下所示：

```
import sys
sys.path.insert(0,'/home/mc/lib')
import mymodule
```

一旦加载了 sys.path 数组，就能确认哪个模块是当前利用 sys.modules 对象加载的。sys.path 数组是个字典，此字典列出两项内容：模块名和对应模块是从哪个目录加载而来的。例如，在所有用户模块均被导入之前，在 Linux 系统上安装了 Python，其目录清单如下所示：

```
{'os.path': <module 'posixpath' from
'/usr/local/lib/python2.1/posixpath.pyc'>,
'os': <module 'os' from '/usr/local/lib/python2.1/os.pyc'>,
'readline': <module 'readline' (built-in)>,
'exceptions': <module 'exceptions' (built-in)>,
'__main__': <module '__main__' (built-in)>,
'posix': <module 'posix' (built-in)>,
'sys': <module 'sys' (built-in)>,
'__builtin__': <module '__builtin__' (built-in)>,
'site': <module 'site' from '/usr/local/lib/python2.1/site.pyc'>,
'signal': <module 'signal' (built-in)>,
'UserDict': <module 'UserDict' from
'/usr/local/lib/python2.1/UserDict.pyc'>,
'posixpath': <module 'posixpath' from
'/usr/local/lib/python2.1/posixpath.pyc'>,
'stat': <module 'stat' from '/usr/local/lib/python2.1/stat.pyc'>}
```

请注意，每对字典词条的格式是特定的：键和键所对应的值。其中，键包含模块的名字，

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

键所对应的值包含有关模块的实名以及模块是内置模块还是从外部文件加载而来的信息。

9.2　使用操作系统(os 模块)

与操作系统的交互包括如下种种事务：确认解释器和脚本运行的环境；用户和进程环境；还有对诸如文件和文件系统这样的外部系统的控制和与其进行通信。

在 Python 中上述功能大部分由 os 模块处理，尽管一些其他模块提供了与其他系统的通用接口。os 模块自身不是上述功能的提供者，它只是创建必要的链接，该链接是 os 模块名称空间与诸如 posix 模块（用于 Unix/Windows NT 和 2000)及 mac 内置模块之间的链接，其中：mac 内置模块提供依赖系统为基础的功能。这样的设计就允许 os 模块担当起跨平台模块的角色，跨平台模块提供用于与主机系统通信的所有核心 API。

通过检查 os.name 变量，就能确认种种依赖操作系统的模块中有哪个模块已加载，如下例所示：

```
Python 2.0 (#71, Oct 22 2000, 22:09:24) [CW PPC w/GUSI2 w/THREADS] on mac
Type "copyright", "credits" or "license" for more information.
>>> import os
>>> os.name
'mac'
```

os.path 变量保存用来处理与平台无关的路径名操作的模块名。能使用如下命令直接导入所需模块：

```
import os.path
```

9.2.1　操作环境变量

os.environ 字典提供对当前进程(Python 解释器）的环境变量的访问。os.environ 变量实际上是个映射对象，这就意味着 os.environ 能像字典一样访问其中信息，还能利用标准字典语句和表达式修改实际环境值。环境的改变将影响 Python 解释器和通过 os.system()函数或 os.exec()函数启动的所有程序；并且此环境还用来作为利用 os.fork()函数创建的子进程的环境。

在启动期间，os.environ 字典中的内容被填充到当前环境中。os.environ 字典能像标准字典一样被访问其中信息，如下所示：

```
print os.environ['PATH']
```

还能在 os.environ 字典中将值分配给特定的键来创建和 / 或设置环境变量值。例如，要修改环境变量 PATH 的值，其中 PATH 定义用于搜索外部程序的目录列表，就可以使用如下设置：

```
os.environ['PATH'] = '/bin:/sbin:/usr/sbin:/usr/bin:/usr/local/bin'
```

另一可选方法是：利用 os.putenv()函数设置一环境变量的值，如下所示：

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
os.putenv('PATH', 'C:\\Python;C:\\WINNT\system32;C:\\WINNT')
```

请注意，尽管将一个值分配给 os.environ 字典提供的映射对象就自动调用 os.putenv()函数，但是 os.putenv()函数不自动更新 os.environ 字典。这可能意味着利用 os.putenv()函数对环境的改变在 os.environ 字典中得不到反映，即使是在实际环境已确实被更改的情况下也是如此。

9.2.2　行终止

因为 os 模块是加载在依赖系统的基础之上的，所以 os 模块还提供用于设置和提供依赖系统的信息的便捷方法。此功能大部分是通过 os 模块加载的不同函数和其他依赖系统的模块处理的。

行终止一直是所有跨平台编译语言的老大难问题。Python 的解决之道是：自动设置 os.linesep 变量的值为当前平台使用的正确行终止。行终止的实际值是个字符串。对于 POSIX 系统而言，行终止是单个换行字符('\n')；对于 Mac OS 机器而言，行终止是单个回车字符('\r')；而 Windows 系统使用的则是回车符 / 换行符序列('\r\n')。

在输出信息到屏幕上时，os.linesep 变量值就被 print 语句使用；而在从文件中读取信息时，os.linesep 变量值则被 readline()方法使用。在大多数情况下，不必引用 os.linesep 变量值。但是，如果正编写跨平台脚本且使用文件对象和 write()方法来输出文本数据，就必须在每行语句的末尾处使用 os.linesep 变量值，如下例所示：

```
file.write('Some other text' + os.linesep)
```

这样就确保当前无论是在什么操作系统下运行，都将使用正确的行终止。

9.2.3　进程环境

要获取用户 ID 和其他特定于进程的信息，必须使用 os 模块的一系列函数中的一个函数。表 9-2 是这些函数的一个清单。

表 9-2　os 模块的进程信息函数

Python 函数	说　　明
chdir(path)	更改当前工作目录为 path
getcwd()	返回当前工作目录的路径
getegid()	返回有效组 ID，仅适用于 Unix 操作系统
geteuid()	返回有效用户 ID，仅适用于 Unix 操作系统
getgid()	返回实组 ID，仅适用于 Unix 操作系统
getpgrp()	返回当前进程组的 ID，仅适用于 Unix 操作系统
getpid()	返回当前进程的进程的 ID，仅适用于 Unix 和 Windows 操作系统
getppid()	返回父进程的 ID，仅适用于 Unix 操作系统

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

Python 函数	说　明
getuid()	返回实用户组的 ID，仅适用于 Unix 操作系统
putenv(var, value)	设置环境变量名为 var 的变量值为 value，详见本章前面部分的“操作环境变量”小节，仅适用于 Unix 和 Windows 操作系统
setgid(gid)	设置组 ID 为 gid 参数，仅适用于 Unix 操作系统，需要超级用户的特许
setpgrp()	为当前进程创建一个新的进程组，返回这个新进程组的 ID，仅适用于 Unix 操作系统，需要超级用户权限
setpgid(pid, pgrp)	分配进程 ID 值为 pid 的进程作为 pgrp 组的一个成员，仅适用于 Unix 操作系统，需要超级用户权限
setsid()	创建一个新的会话(session)，并返回新建会话的 ID。会话与终端和支持会话的 shell 一起使用，就允许在不使用窗口系统的情况下运行多应用程序，仅适用于 Unix 操作系统
setuid(uid)	设置当前进程的用户 ID，仅适用于 Unix 操作系统，需要超级用户权限
strerror(errno)	返回与 code 中错误号相关联的错误消息，仅适用于 Unix 和 Windows 操作系统
umask(mask)	设置当前进程的 umask 为 mask 参数值，仅适用于 Unix/Windows 操作系统
uname()	返回一个字符串元组，此元组包含的内容有系统名、节点名、版本修正号、版本号和当前系统的机器，仅适用于 Unix 操作系统

9.2.4　进程执行与管理

在 Python 内，能执行和管理进程。进程执行包含的操作：既可以是替代 Python 解释器，启动一个新进程，也可以是与从中读出或往里写入的进程，或者对其进行读写的进程进行通信。还能安装信号处理器(signal handler)来控制将一信号发送给进程时进程该如何反应。在 Python 中，上述功能大部分是在 os 模块中处理的。

1. 运行外部命令

尽管在 Python 中利用一些模块或扩展模块能处理许多事务，但还是有必须通过运行一些外部命令与外面系统通信的时候。

(1) os.exec*()系列函数　Python 用于执行外部命令的基本函数是 execv。在同一个函数上，根据最终要实现的功能还有一些变化：execve()函数及 execvpe()函数允许提供一个环境变量字典；搜索 os.environ['PATH'] 中用于特定应用程序的环境路径则执行 execvp()函数或 execvpe()函数。每个函数的格式如下：

```
os.execv(path, args)
os.execve(path, args, env)
os.execvp(path, args)
os.execvpe(path, args, env)
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

其中：path 参数应当是要执行文件的完整路径。当使用 os.execvp()函数或 os.execvpe()函数时，搜索的就是环境路径目录。另一个选择是：利用 os.defpath 变量设置一个不同的路径。args 参数应是提供给正调用程序的参数列表(或元组)。env 参数应是环境变量的字典：这些环境变量将被用来替代 Python 解释器所用的环境。运行上述所有函数的一些举例如下：

```
os.execv('/bin/ls', ('-la'))
os.execve('/usr/local/bin/cvs', ('commit'), {'CVSROOT':'/export/cvs'})
os.execvp('ls',('-la'))
os.execvpe('cvs', ('commit'), {'CVSROOT':'/export/cvs'})
```

请注意，在任何情况下，exec*()系列命令完全替代 Python 解释器。如果执行 Python 脚本中的如下语句(不使用 os.fork()函数)，则 Python 解释器和脚本将终止，而 ls 程序则替代它们开始执行。如下所示：

```
os.execv('/bin/ls', ('-la'))
```

如果想启动一个附加进程，必须使用 os.fork()函数、os.spawn*()函数(仅适用于 Unix/Windows 系统)和 os.system()命令之中的任何一个函数或命令，其中：os.fork()函数用来启动一个新子进程。

除了基本版本外，还有多参数版本，多参数版本能使用数目可变的参数，而不是使用固定数目的元组或列表参数来调用 os.exec*()系列函数。即：

```
os.execl(path, arg0, arg1, ...)          等价于 os.execv(path, (arg0, arg1, ...)
os.execle(path, arg0, arg1, ..., env)    等价于 os.execve(path, (arg0, arg1, ...),env)
os.execlp(path, arg0, arg1, ...)         等价于 os.execvp(path, (arg0, arg1, ...))
```

在各种情况下，如果在 path 路径下查不到程序，就会引发 OSError 异常，OSError 异常具有包含精确错误的附加参数，如下例所示：

```
>>> import os
>>> os.execv('nothing',('',))
Traceback (most recent call last):
File "<stdin>", line 1, in ?
OSError: [Errno 2] No such file or directory
```

(2) 启动新进程　要启动新进程，而不是替代当前进程，必须使用 system()函数，如下所示：

```
import os
os.system('emacs')
```

os.system()函数的返回值是被调用程序的退出状态。在 Windows 系统下，不管情况如何，其返回值总是零。如果想阅读程序的输出，请使用 Python 的 os.popen()函数来替代 os.system()函数(参见“与外部进程的通信”部分)。

另一个可选方式是：在 Unix 和 Windows 系统下，使用 os.spawn*()函数。os.spawn*()系

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

列函数与 os.exec*()系列函数的工作方式相近，只不过 os.spawn*()系列函数创建一个新的进程，而 os.exec*()系列函数是替代现存进程。os.spawn*()系列函数的两种格式如下：

```
os.spawnv(mode, path, args)
os.spawnvw(mode, path, args, env)
```

其中：path 参数是要创建的应用程序的位置；args 参数是一个列表或元组，此列表或元组包含应用程序所用的参数。当使用 os.spawnve()函数时，env 参数是个字典，该字典用于填充已创建的应用程序的环境。

在两种格式中，mode 参数都是常量，如 os 模块所定义的那样，其值列于表 9-3 中。

表 9-3　用于创建新应用程序的模式

常　量	说　明
P_WAIT	执行程序，并一直等待，直到该程序终止，将控制权返回给调用程序为止
P_NOWAIT	执行程序，并立即返回一个进程句柄
P_NOWAITO	等价于 P_NOWAIT
P_OVERLAY	执行程序，此程序替代当前进程，也就是说，像 exec*()系列函数那样工作
P_DETACH	执行程序，且从调用程序中分离出来，新进程继续运行，但不能使用 wait()函数来等待进程的终止

2. 与外部进程的通信

要阅读外部命令的输出，必须使用 os.popen()函数。os.popen()函数打开一个通向程序调用的管道，使得能够阅读从文件输出的或是往文件里写入的信息。例如，下面的代码打开一个与获取文件清单的 ls 外部命令的连接，通过这个连接，就能读出和打印相关的信息(关于较好的选择，请参见第 11 章中论述的 glob 模块)。

```
import os
dir = os.popen('ls -al','r')
while(1):
    line = dir.readline()
    if line:
    print line,
else:
    break
```

确切地说，os.popen()函数的工作情况与内置 open()函数一样。os.popen()函数返回一个文件对象，对文件对象能以字节或行为单位进行读取。os.popen()函数的实际格式如下：

```
os.popen(command [, mode [, bufsize]])
```

其中：command 参数应是被 shell 执行的命令字符串，该命令在调用 os.popen 函数时启动。mode 参数应是用于只读的“r”或者是用于只写的“w”，在默认 mode 参数时，其默认

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

取值为“r”。bufsize 参数是当对 command 参数所指命令进行读取时使用的缓存大小，就像 Python 的 open()函数的 bufsize 参数一样。os.popen()函数仅限于使用在 Unix 和 Windows 系统中。

有所缺憾的是：os.popen()函数是单向工作的，只能从所调用的程序读取或是往所调用的程序里写入，而不能对进程既读又写。要实现既读又写的功能，必须使用 popen2 模块中的 popen2()函数。

popen.popen2()函数创建一个新进程，并返回子进程的标准输出和标准输入文件句柄，如下所示：

```
(child_stdout, child_stdin) = popen2(cmd [, bufsize [, mode]])
```

popen3()函数返回标准输出、标准输入和标准错误文件句柄，如下所示：

```
(child_stdout, child_stdin, child_stderr) = popen3(cmd [, bufsize, [, mode]])
```

最后，popen4()函数返回一个组合的 stdout/stderr 文件句柄，如下所示：

```
(child_stdout_and_error, child_stdin) = popen4(cmd[, bufsize[, mode ] ])
```

一旦打开 child_stdout、child_stdin 和 child_stderr，就能利用与普通文件相同的方法对它们进行读写。

popen2.Popen3()函数返回 Popen3 类的一个实例，如下所示：

```
child = Popen3(cmd [, capturestderr [, bufsize]])
```

其中：capturestderr 参数如取值为真(true)，就强制此函数返回的实例与捕获标准输入和输出一样去捕获标准错误。默认设置是不捕获标准错误。此函数返回的新实例具有如下的方法和属性：

child.poll()	返回子进程的退出码，或返回–1；如果返回值是–1，就说明子进程仍在运行
Child.wait()	等待子进程终止并返回退出码
Child.fromchild	捕获子进程的标准输出的文件对象
Child.tochild	将输入发送给子进程的标准输入的文件对象
Child.childerr	捕获子进程的标准错误的文件对象

最后，还有 Popen4 类，它的工作方式与 popen4()函数相同，返回一个组合的标准输出/标准错误的文件句柄，如下所示：

```
Popen4(cmd[, bufsize ])
```

3. 创建子进程

大多服务器和其他进程都在运行期间创建“子进程”。父进程通常是几种监听程序及服务分配程序——例如 Web 服务器——当 Web 服务器接收到一个请求或网络连接时，它就创建一个子进程对请求提供服务。使用这种方法，处理大量客户请求就比较容易了，而无需求

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

助于循环法(round robin)或循环服务请求的手段。

操作系统的 fork()函数创建一个新的子进程。在 Python 中，os.fork()函数只是操作系统版本的别名。当 os.fork()函数被调用时，实际所发生的就是当前进程被复制。os.fork()函数在子进程中返回零，而在父进程中就返回子进程的 ID 值。在 Python 中，要检查当前进程是父进程还是子进程，只检查一下 os.fork()函数的返回值就行，如下例所示：

```
pid = os.fork()
if not pid:
        # Start of child process
else:
        # Continuation of parent process
```

例如，下面的脚本使一个新进程休眠一段时间并发送一个消息：

```
import os,time

pid = os.fork()
if not pid:
      for step in range(10):
            print 'Hello from the child!'
            time.sleep(1)
else:
       for step in range(10):
            print 'Hello from the parent!'
            time.sleep(1)
```

当上面的脚本执行时，产生的输出如下：

```
Hello from the parent!
Hello from the child!
Hello from the child!
Hello from the parent!
...
Hello from the parent!
Hello from the parent!
Hello from the child!
Hello from the child!
Hello from the parent!
```

如上所述，在同一个脚本中，我们有效地执行了两个不同的进程。实质上，在一个单进程内，我们最终实现了以“准多线程”系统的方法来执行多个进程。请注意，任何打开的文件句柄——包括网络套接字在内——在 os.fork()调用时都被复制。这在以下情况下有用：想接受父进程中打开的文件句柄上的数据，并将此信息提供给子进程。一旦子进程被创建，就关闭父进程的文件句柄而让子进程处理这些文件句柄。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

4. 等待子进程

当激活新进程时，到新进程最终死亡，必须等待进程完整退出以确保这些进程在进程表中不保持为“zombies”，这个进程称作 reaping。等待进程结束的函数有两个，分别是 os.wait()函数和 os.waitpid()函数，如下所示：

```
(pid, exitcode) = os.wait([pid])
(pid, exitcode) = os.waitpid(pid, options)
```

其中：pid 参数是所等待进程的进程 ID。如果不指定 wait()函数的 pid，则此进程就一直等待直到所有的子进程全部死亡为止。例如，在脚本的结尾处要等待子进程终止，可能只需使用如下的语句：

```
os.wait()
```

请注意，父进程冻结(或暂停)一直到所等待的进程完成为止。

如果想避免这样的冻结，就必须使用 waitpid()函数。options 参数可接受零和 os.WNOHANG 两个值中的任一个值。当取值为零时，就强制为等价于调用 os.wait()函数的正常操作；当取值为 os.WNOHANG 时，调用就立即返回。利用这种方式的 waitpid()函数，就允许像父进程的标准循环的一部分那样调用一个 reaping 进程。举例如下：

```
import os

while 1:
    # wait for a connection
    if accepted:
        pid = os.fork()
    if not pid:
        # do the child stuff
    else:
        os.waitpid(pid, os.WNOHANG)
```

在一个真正的多进程环境下，很可能将进程 ID 添加到一个数组中，然后再一个一个依次处理。

比较好的解决方法是创建一个信号处理程序。每个子进程在终止时，都给其父进程发送一个 SIGCHLD 信号。利用一个信号处理程序就能阻止数组的使用。接下来信号处理程序调用 wait()函数或 waitpid()函数。例如，以下信号处理程序是回收旧进程的最容易的途径。

```
def child_handler(signum, frame):
    os.wait()
```

详见本章后面关于如何创建和安装不同的信号处理程序内容的“信号”部分。

5. 获取退出状态

wait()函数和 waitpid()函数两者都返回一个元组，该元组由终止的子进程的进程 ID 和子

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

进程的退出状态组成。退出状态是一个 16 位数字，其低字节是中止进程的信号数字，而高字节是退出状态，如下所示：

```
(newpid, exitcode) = os.waitpid(pid, os.WNOHANG)
```

对于子进程自身而言，exitcode 不是很有用，但是，os 模块定义了一个函数序号，用来检查 exitcode，根据 exitcode 的值来确定子进程是如何终止的。这些函数是 WIFSTOPPED()、WIFSIGNALED()、WIFEXITED()、WEXITSTATUS()、WSTOPSIG()和 WTERMSIG()函数，并且在 exitcode 参数与给定的条件相匹配时，这些函数的返回值均为真(true)。

例如，因为发送给一个子进程另外的信号，所以要检查这个子进程是否已终止。就可以使用如下的代码来完成这样的检查：

```
import os

while(1):
        (newpid, exitcode) = os.waitpid(pid, os.WNOHANG)
        if (WIFSIGNALED(exitcode)): break
```

9.2.5　信号

信号提供了一个用于将特定的事件发送给进程的方法。信号并不存在一个通信机制：不能传递信息——您所能做的就是给特定事件发送信号。信号被用在各种操作系统上，使用信号的主要操作系统有 Unix 和 POSIX 1003.1 可兼容操作系统(包括 Windows NT/2000)。并且信号通常被用作管理进程和进程执行的一个方法。比如，KILL 信号就通告操作系统终止进程。其他信号能用来初始化特定函数。例如，将 HUP 信号发送给命名的 DNS(域名系统) 服务器就强制服务器重新加载其配置文件。

系统所支持信号的实际清单完全依赖于操作系统或操作系统的版本。表 9-4 包括了 POSIX 信号内容，它们应是绝大多数操作系统都支持的。

表 9-4　POSIX 信号

POSIX 1003.1 名称	说　明
SIGABRT	非正常终止
SIGALRM	alarm 函数设置的计时器到期
SIGFPE	算术异常，如，除法溢出或被零除
SIGHUP	在控制终端检测到的挂起或是控制进程的死亡
SIGILL	表明程序错误的非法指令
SIGINT	中断信号(来自于键盘的特殊字符或来自于另外应用程序的信号)
SIGKILL	终止信号，不能捕捉或忽略
SIGPIPE	试图往一个管道写入，该管道没有任何应用程序从中读取
SIGQUIT	解除信号(quit signal)(来自于键盘的特殊字符或来自于另外应用程序的信号)
SIGSEGV	试图访问一个无效的内存地址

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

POSIX 1003.1 名称	说　　明
SIGTERM	终止信号(来自于其他应用程序或来自于操作系统)
SIGUSR1	应用程序定义的(或用户定义的)信号
SIGUSR2	应用程序定义的(或用户定义的)信号
SIGCHLD	一个被终止的或被停止的子进程
SIGCONT	如果进程当前已停止，继续执行此进程
SIGSTOP	停止信号，停止特定进程
SIGTSTP	来自于键盘的特定字符对应的停止信号
SIGTTIN	来自于控制终端的后台进程尝试进行的一个读取
SIGTTOU	来自于控制终端的后台进程尝试进行的一个写入

所有的信号都具有默认操作：SIGSEGV 信号通常产生 Unix 操作系统下的进程核心转储，就如上面关于 named 的例子一样。关于捕获信号的详细内容请参见本章后面“信号处理程序”部分。在非 Unix 操作系统下，受到支持的信号就非常有限了，例如，Windows 系统就只支持 ABRT、FPE、ILL、INT、SEGV 和 TERM 信号。

1. 发送信号

Python 的 os.kill()函数的功能是将一个信号发送给一个现存进程，如下所示：

```
kill(pid, signal)
```

signal 模块包含了不同信号的符号式常量，这些信号是带有 SIG*这样形式的常量。比如，可以利用如下语句给当前进程发送警告信号：

```
kill(os.getpid(), signal.SIGALRM)
```

有关所在系统支持哪个信号的信息，请参见表 9-4(关于 POSIX 信号的完全清单)，或参见 Unix 系统下 signal(5)的联机帮助或 Windows 系统下的 MSDN 库。

2. 信号处理程序

要捕获信号必须安装所谓的“信号处理程序”。信号处理程序是当信号被进程接收到时将调用的函数。在 Python 中，利用 signal.signal()函数来安装一个信号处理程序。signal.signal()函数接受两个参数：一个是要捕捉的信号；另一个参数是将要调用的函数，如下例所示：

```
signal.signal(signal, handler)
```

例如，要安装 SIGALRM 信号的系统处理程序，其代码如下所示：

```
import signal,time,sys

def alarm_handler(signum, frame):
    print "Wake Up!"
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
    sys.exit()

signal.signal(signal.SIGALRM, alarm_handler)

signal.alarm(5)
print "Going to sleep... "
time.sleep(10)
print "Now I'm up. "
```

其中：signal.alarm()函数以给定秒数为计时的警告计时器。一旦计时到期，操作系统就立即给进程发送 SIGALRM 信号。在本例中，我们设置了一个 5 秒长为计时的警告计时器，而所需等待的时间是 10 秒。如果运行此脚本，则在大约 5 秒钟后应获得“Wake Up!”消息，而不应在脚本于信号处理程序函数末尾终止时看到“Now I'm up”消息。

在任何情况下，信号处理程序必须接受两个参数。第一个参数是 signum，它是信号处理程序被调用时产生的信号；第二个参数是 frame，它是个框架对象，该框架对象描述在信号产生点的 Python 执行堆栈。在第 24 章，我们将关注框架的细节。框架被 Python 解释器用来管理脚本的执行。

如果在安装信号处理程序时，不提供接受这两个参数的函数，则将引发 TypeError 异常。

(1) 获取当前信号处理程序　利用以下语句能获取当前安装的一个特定信号的信号处理程序名：

```
handler = signal.getsignal(signal.SIGALRM)
```

其返回对象是可调用的，所以能利用下面的语句立即调用此信号处理程序：

```
handler()
```

(2) 禁用信号处理程序　signal 模块定义了两个标准处理程序，这两个处理程序能用于修改信号处理进程的行为。signal.SIG_IGN 信号处理程序强制 Python 忽略指定的信号，而 signal.SIG_DFL 信号处理程序则调用默认信号处理程序。能像下面这样忽略一个信号：

```
signal.signal(SIGALRM, signal.SIG_IGN)
```

或者利用下面的语句来设置信号进行其默认操作(就是操作系统定义的操作)：

```
signal.signal(SIGQUIT, signal.SIG_DFL)
```

请记住，不能使用诸如 KILL 这样的方法来忽略信号，信号一般是由操作系统处理的。

9.2.6　用户/组信息

在 Unix /etc/passwd 和/etc/group{s}文件中存储的用户和组的信息通过 grp 和 pwd 模块可获取。无论在哪种情况下，如果组不存在，就引发 KeyError 异常。请参见表 9-5，表 9-5 是有关 Python 所支持的函数的清单。

举例，要获得当前用户的主目录和 shell，就可以使用如下代码：

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
import pwd,os

pwinfo = pwd.getpwuid(os.getuid())
name = pwinfo[0]
fullname, homedir, shell = pwinfo[4:]
print "You are %s (%s)\nHome Directory is %s\nShell is %s" \
                % (name, fullname, homedir, shell)
```

表 9-5　　Python 获取用户和组的入口

函　　数	说　　明
grp.getgrgid(gid)	返回包含与 gid 组 id 参数相匹配的组信息的元组。此元组由组名、组口令、组 id 和组的成员列表组成。如，grp.getgrgid(0)返回('root', '', 0, ['root'])
grp.getgrnam(name)	等价于 getgrgid()函数，但它所关注的是与 name 名字参数相匹配的组
grp.getgrall()	返回一个由元组组成的列表，其中，每个元组包含 getgrgid()函数返回的信息
pwd.getpwuid(uid)	返回 uid 用户 ID 参数所对应用户的信息，返回一个元组，该元组包含用户名、口令、用户 ID、组 ID、gecos(完整名字和／或有联系的细节)、主目录和 shell。如，pwd.getpwuid(0)返回 ('root','x', 0, 1, 'Super-User', '/', '/usr/bin/bash')
pwd.getpwnam(name)	等价于 getpwuid()函数，但它所关注的是与 name 名字参数相匹配的用户
pwd.getpwall()	返回一个由元组组成的列表，其中，每个元组包含 getpwuid()函数返回的信息

可能还想查询 getpass 模块和 crypt 模块，这两个模块提供一个安全途径，用来从终端获取口令，用来将一文本口令加密成为存储在/etc/passwd 文件中的加密版本。例如，下面的脚本提示并检查当前用户的口令：

```
import getpass,pwd,crypt
password = getpass.getpass()
realpw = pwd.getpwnam(getpass.getuser())[1]
if realpw == crypt.crypt(password,password[:2]):
        print "Password validated"
else:
        print "Password invalid
```

9.3　多线程

线程是对于许多平台和编程语言都比较新的模式。线程是在一给定进程中同时执行大量不同函数的重要方法，而无需借助于利用 fork()函数或利用诸如 select()函数(参见第 10 章)这样效率很低的方法(其中，因为进程被复制，所以 fork()函数意味着愈加严重的系统负荷)。在关注线程的特性之前，值得去研究一下现代多任务操作系统是如何工作的。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

9.3.1　多任务工作原理

如果研究过现代典型的操作系统，就会发现此操作系统被设计为能同时处理大量进程的执行。达到上述功能所用的方法或是“多任务协作”或是“多任务预占”。在这两种情况中，同时执行大量进程的实际方法是一样的——操作系统以几分之一秒为周期在应用程序间逐个切换，使用循环方式暂停前一个应用程序，接着恢复下一个应用程序。所以，如果操作系统有 20 个并发进程。每一个进程执行了几分之一秒之后，就被挂起，在再次获得执行的机会之前不得不等待其他 19 个进程执行它们各自的工作。

各个进程一般对这样的切换并不识别，这样的切换对应用程序的影响是微不足道的——大多数应用程序并不在意它们是像单进程那样工作还是作为多进程环境中的一部分那样工作。其原因是操作系统在如此低的层上控制应用程序的执行。

两种类型的多任务，协作和预占，描述了操作系统如何控制正在执行的应用程序。使用多任务协作类型，所有的进程隐含地给定了与其他所有进程一样数量的执行时间。实际所发生的是给定的进程一直执行直到该进程调用一系统函数为止，在这时，操作系统就能够切换到别的进程上。这就意味着一个应用程序永不放弃对处理器的控制是可能的。一些操作系统是比较具体的，将大部分处理器时间(一般为百分之八十)提供给“主”应用程序，而将剩余的处理器时间(百分之二十)提供给“后台”应用程序。这就是 Mac OS(但不是 MacOS X)使用的模式，并且 Mac OS 允许 GUI 环境将大多数时间提供给用户当前正使用的应用程序。

多任务预占更是复杂。不是采用所有执行进程之间任意共享处理器时间的方法，多任务预占类型的操作系统将处理器的大部分时间提供给提出请求的进程。操作系统通过监控每个运行的进程来实现上述分配处理器时间的策略，操作系统还管理每个进程的执行。具有高优先权的进程得到处理器较多的时间；而具有最低优先权的进程就得到最少的时间。因为能控制进程的优先权，我们就能较好地控制如何执行不同的进程。例如，在一个数据库服务器上，要想给数据库进程最高优先权以确保数据库的速度。多任务预占是许多面向服务器的操作系统的一个主要特性，面向服务器的操作系统包括 Unix、Linux 和包括 Windows 2000 和 NT 自身的以 NT 为基础的 Windows 兼容操作系统。多任务预占还普遍地用在面向客户的操作系统中，面向客户的操作系统包括 Windows 95/98/NT、Workstation/2000 专业版、Mac OS X 和所有支持多进程预占的 BeOS。

不同的多任务解决方法还决定了操作系统能使用的硬件类型。多任务协作仅在单处理器系统上真正有实效。这是因为循环法(round-robin approach)要求进程在其整个运行周期上都存在于同一个处理器上。尽管可能(并且事实上也时常地普遍地)在单处理器系统上发现多任务预占，但多任务预占并不是像所预料的那样处理进程执行。事实上，在某些方面，预占系统的要求对推动处理器的设计已有所帮助——Sun 公司使用的 SPARC、SuperSPARC 和 UltraSPARC 处理器已特别注意为实现多任务而设计。这些处理器拥有多个寄存器组用来允许处理器在多任务之间容易地切换。

对于多任务预占而言，多处理器解决方案是最佳硬件平台。因为操作系统知道每个进程需求的时间是多少，操作系统就能根据每个处理器的忙闲程度将各个进程分配给不同的处理器。这有助于充分发挥可利用的处理器的能力，并高效地分散其上的负荷。总之，工作的划

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

分是以一个进程 一个进程为基础的，所以如果有一个任务高度密集的进程，即使系统的其他处理器有空闲能力，它也仅能在单个处理器上执行。

9.3.2　从多任务到多线程

就多任务操作系统而言，似乎有大量进程在并发执行。事实上，每个进程运行的时间只是几分之一秒，并且可能在一秒钟内运行数次，这样自然就给出一个真实的多任务环境，在这个多任务环境里，许多个独立的处理器处理各自的应用程序。

对每个进程而言，都有一个内存的分配问题，就是在操作系统支持的寻址空间内的内存分配。内存分配需要跟踪，而对于像 Unix 这样的多用户操作系统而言，还有许可和安全属性，当然还有应用程序自身的实际代码。跟踪所有这样的信息是个专职工作——在 Unix 操作系统下，有大量进程密切注视所有这样的信息，除了 kernel 核进程之外，此进程实际处理许多请求。

如果一个独立进程想并发地执行大量任务，有两个可行的解决方法。第一个方法是循环法，就像主要操作系统所采用的那样，但是循环法不能在同一个控制层上执行多个任务。需要执行的每个函数都是在一个循环中依次被调用，但是，因为在函数执行中间不能终止其执行，所以就常常有个“延迟时间”的难题——如果一个函数有海量的信息要处理，则此函数的执行将阻塞整个循环的执行。

对于文件处理而言，能通过使用 select 函数和分析每个文件固定大小的块信息来化解此难题。在这个实例中，我们仅处理提供(或需要)较多数据的文件的信息，并假定只读取单行数据或固定长度的数据块，那么处理每个请求的时间应当是相对小的。

对于解决更复杂的多任务处理机制要求的方法，惟一的另一个可选方法是“创建”一个新进程来处理其要处理的事件。因为 fork()函数创建一个新进程，其执行和优先权处理都能被父操作系统控制，所以这常常是诸如 Apache、IMAP 及 POP3 后台程序的网络服务使用的解决方法，当客户连接到服务器时，网络服务进程创建一个设计为处理客户端请求的新进程。

创建新进程的难题是此进程是消耗时间的和资源紧张的进程，而且实际上它并不解决问题：新进程需要操作系统和另一个进程管理，并且还要给它分配时间。创建新进程就意味着分配新的内存块和创建进程表的新入口，进程表被操作系统的调度表用来控制每个进程的执行。为了对资源消耗情况有个大致了解，举个例子，典型的 Apache 进程占用的空间大约为 500KB，如果 20 个客户都在同一时刻连接，就要给 20 个新进程中的每一个进程分配 10MB 大小的内存，并复制主图像，由此可见资源消耗之大。

在大多数情况下，实际上并不需要大量与新进程相关联的元包。就 Apache 而言，创建的新进程不需要读取配置文件——Apache 已经做了这些事务，并且也不需要处理任何复杂套接字处理程序。仅仅需要的是与所服务的客户端套接字进行通信的能力。

创建新进程的资源需求对能在同一时刻并发连接的客户数目加了无益的限制——并发连接的客户数目依赖于可用内存空间的大小和操作系统能处理的进程的最大数目。处理客户请求所需的实际代码可能非常小，一般为 20KB　在一个具有 128MB 内存的系统上利用多进程处理可能限制客户数目为 200 个左右——对一个繁忙的站点而言，这不是个特别大的数字。要处理比这更多的客户请求，就需要更多的内存和更多的处理器：200 个进程在单 CPU 上进

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

行切换不是我们所推荐的，其原因是在一个遍历周期(将每个进程执行一次)内分配给每个进程的时间数量可能太小了，以致于即使是一个非常小的请求，对于单进程而言，也要花费几分钟才能完成请求处理。

上述就是线程思想引入的原因所在。线程就像个变轻型的进程——事实上，线程也常常被称作为“轻型进程”。线程在父进程的范围内运行，并且通常就执行父进程的某一个函数。创建新线程并不意味着要分配大区域的内存(大概就分配父进程内存空间中的一个内存块)，也不意味着需要在操作系统的调度表中增加额外的入口。在我们的 Web 服务器举例中，不是创建一个新进程来处理客户端的请求，而是利用处理客户端请求的函数创建一个新的线程。

通过使用多线程，能获得父操作系统提供的多进程处理能力，但多线程是在单进程的范围内。现在一独立进程能同时执行大量函数，或着执行同一函数多次，就像我们的 Web 服务器所遇到的情况一样。

在一个支持多任务预占和多线程的操作系统上，能获得基于主进程的优先系统和一个内部“预处理”多任务的环境。在一个多处理器系统上，大多数操作系统还将线程从单个处理器上分布到所有处理器上。所以，如果具有一个任务特别密集的进程，通过将该进程的操作划分为大量独立的线程，该进程就能使用所有可用资源和所有处理器。就不再陷入这样的情况：一个进程被紧紧地限制在一个处理器上，或是不得不创建大量的拷贝来对大量客户的请求提供服务。

线程是操作系统特有的。即使是在现在，也只有为数不多的操作系统能提供达到一合理水平上的线程功能，并且还需要附加的或不同的函数库来使此功能得以实现。大多数支持线程的操作系统不是以 Unix 为基础就是以 Windows 为基础，但也有例外，如 BeOS 系统和 Mac OS 系统(其中，以 Unix 为基础的操作系统有 Solaris、AIX、HP-UX、一些 Linux 版本、BSD 及 Mac OS X，而以 Windows 为基础的操作系统有 Windows 98/NT/2000/Me)。

9.3.3　线程与多进程的比较

多线程应用与多进程应用的主要区别与相对应的资源消耗直接相关，这已经被讨论过了。利用 fork()函数创建同一个进程的拷贝实例需要大量的内存和处理器时间。而新线程的总开销只比所执行函数所需的开销稍微大些，并且除非正在线程间传送巨大数据块，否则，要在单进程上创建上百个甚至上千个线程(对某些系统而言)是难以置信的。

另外一个惟一区别是能在线程上实行控制和通信所在的层。当创建一个进程，它与其子进程间的通信就受到限制。要相互交换信息，就必须打开通向与其子进程通信的管道，这样的通信对于大量子进程而言就变得太不方便了。如果仅是想控制子进程，在利用信号中止或挂起进程这方面就受到制约——没有办法将线程重新集成到主进程中去，也没办法不使用信号而任意地控制子进程的执行。

9.3.4　线程和 select()的比较

select 函数为并发地从大量文件句柄中进行数据输入和数据输出提供了一条极佳的途径，但这也是 select 函数能和线程相提并论的惟一之处。总之，select 函数不可能用于与文件句柄进行通信之外的任何事务上，并且使用 select 函数处理所有事务也限制并发处理进

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

程的效率。

另一方面，利用线程，能创建一个新的线程来处理进程执行的各方面事务，包括，但不仅限于，与文件句柄通信。举例，对于多协议计算，就可以创建新线程来分别处理计算的各个不同部分。

9.3.5　线程和 Python

Python 在 Windows、Solaris、BeOS、一些 Linux 变体和大多数支持 POSIX 线程库(pthread)的其他 Unix 变种操作系统上支持线程。Python 的初始配置默认：线程是被禁用的。这就意味着：必须通过手动修改 Python 扩展目录下的 Setup 文件来使系统增加对线程的支持。详细信息请查阅 Python 的安装文件。

线程调度和线程切换是由控制线程执行的解释器锁控制的。这样就确保在任何时刻仅仅一个单线程能被解释器当前实例执行。还仅能在执行独立字节码的线程间进行切换(详见关于字节码和它是如何影响执行内容的第 23 章)。字节码指令的数目是能控制的，字节码指令是在利用 sys.setcheckinterval()函数将执行操作切换到下个线程之前被每个线程执行的指令。系统默认字节码指令数目是 10 字节码指令。

在以下情况还要多加注意：利用 C/C++语言编写的 Python 扩展库，这些扩展库可能或不可能兼容线程。如果这些扩展库不兼容线程，则进程将死锁一直到函数调用全部完成为止，而不考虑 sys.setcheckinterval()函数的设置。

还应当意识到在应用线程中信号使用带来的影响。信号处理程序仅能被主线程捕捉和执行，而不能被脚本中的任何附加线程捕捉和执行。显然这还意味着您不能给独立线程发送信号。

9.3.6　基本线程

thread 模块为创建线程和管理数据结构上的锁提供了一个简化方法。Python 中的所有线程都是通过创建一个新线程来工作的，该新线程的惟一职责是执行在执行期间提供给它们的函数。这意味着如果有一个代码片段要在另外的线程上执行，就必须在创建线程来执行此段代码之前将这段代码放置到一个函数里去。

1. 创建新线程

核心函数是 start_new_thread()函数，其格式如下：

```
start_new_thread(func, args [,keywordargs])
```

此函数创建一个新线程，该新线程启动 func 参数给出的函数，这个函数通过 apply()函数被调用，而 apply()函数使用了 args 参数和 keywordargs 关键字参数。新建线程立即就启动。如果想延迟给定线程的执行，利用 threading 模块能实现，其中，threading 模块将在本章后面的“高级线程”部分中论述。如果函数有错且产生的异常未被处理，则堆栈轨迹被打印出来，执行此函数的线程退出，但包括父线程在内的其他线程继续执行。子线程的异常不会向上传播给父线程。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

总之，子线程在父线程终止时是否继续执行完全依赖操作系统的线程。一些操作系统允许子线程继续执行一直到线程达到所执行函数的末尾为止；而其他操作系统则会中止所有子线程并终止整个程序。检查系统的文档来看看您所在的系统是如何对待非父线程的。

举个例子，有一个显示时间的脚本，每 10 秒钟更新一次时间显示，此脚本利用一个线程来处理更新时间显示过程，如下所示：

```
import thread, time

def display_time(interval, prefix=''):
        while 1:
                print prefix,time.ctime(time.time())
                time.sleep(interval)
thread.start_new_thread(display_time, (10, 'The is now'))
while 1:
        pass
```

函数本身就是个循环，该循环在打印出时间(使用一个前缀选项)后暂停指定间隔的时间。请注意，提供给线程函数的参数必须以元组形式给出。还有，必须在主线程中设置一个什么事也不做的循环以阻止父线程死掉，如果不设置此循环，就可能中止我们所创建的更新时间显示线程。

2. 线程控制

要终止一个线程，可以让函数运行完成返回而使线程终止，也可以调用 thread.exit()函数来强制线程终止。还要意识到：在线程内调用 sys.exit()函数时，sys.exit()函数调用 thread.exit()函数，而 thread.exit()函数只终止线程本身，而不终止全部脚本。

所有的线程都给定了一个惟一的整数标识数字，这个数字能利用线程内的 get_ident()函数确定。如果想记录线程执行，这个整数标识数字就特别有用了(当关注对象访问经过线程时，就将看到关于这方面的一个例子)。请注意，线程 ID 不是连续的，并且进程中的线程和进程 ID 之间没有任何联系。

3. 对象锁定

子线程共享并访问父线程的所有全局对象和数据结构，所以从多个线程对这些结构更新时，必须要谨慎、谨慎再谨慎。同时在不同的两个线程中更新同一数据结构不可能达到希望的结果。尽管有不同的控制访问线程系统独立变量的方法，但是这些方法都依赖于同一个基本前提—— 一个“锁(lock)”。锁是一个线程保护对象，该对象能用于控制对多线程中另一个对象的访问。thread 模块仅提供了一个非常简单用于相互排斥的对象锁定机制，称作为 mutex。有关更高级的方法，请参见本章后面的“高级线程”部分。

allocate_lock()函数创建一个锁对象(类型为 LockType)，锁对象能用于设置或终止给定对象的锁。锁和对象间的关系不是自动的。在使用系统之前，必须知道与每个变量相关的锁对象是哪一个。比如，如果创建一个变量 counter，就可能要创建一个 counter_lock 对象，使用该对象来控制对该变量的访问，其代码如下所示：

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
counter = 0
counter_lock = thread.allocate_lock()
```

如果要改变 counter 变量，必须首先获取内 counter_lock 处理的锁。acquire()方法设法去获取锁，如下所示：

```
lock.acquire([waitflag])
```

其中：waitflag 参数的默认取值为零，这表明不管是否获取到锁，acquire()方法都应立即返回。如果 waitflag 参数是个非零值，则线程将被阻塞一直到获取到锁为止。不论是在哪种情况下，如果想要检查其返回状态，必须给 waitflag 参数提供一个数值。如果锁不能被获取到，acquire()方法就返回零，而如果锁能被获得，就返回 1 。 例如，要先获取锁，然后接着改变 counter 的值，就可使用如下代码：

```
if (counter_lock.acquire(0)):
    counter += 1
```

现在 counter_lock 就被锁定了，并且其他线程想获取此锁的请求总导致失败。虽然有可能其他线程仍能修改 counter 对象，但是其他线程在锁的权限之外进行修改就可能破坏其锁定对象的信息。从根本上而言，互斥(Mutexe)仅仅是个协议，所有线程都必须遵循系统的协议进行工作。要实实在在地释放锁，必须在 counter_lock 对象上使用 release()方法，以便其他线程能获取到锁并能改变对象的值，如下所示：

```
counter_lock.release()
```

如果想检查锁的状态，除了使用 acquire()方法外，还能使用 locked()方法。如果对象是锁定的，locked()方法返回 1，否则 locked()方法返回零。

将以上所述集中到一起，以下脚本就是一个具有三个线程的简单计数器。第一个线程是个显示线程，它简单地输出当前计数器数值而不做任何修改。还有两个线程，A 和 B，它们首先获取一个变量的锁，接着递增锁的值，然后在延迟一段短时期后释放锁。代码如下所示：

```
import thread, time

counter = 0

counter_lock = thread.allocate_lock()

def thread_incra():
    while 1:
        print "Thread A(%d): Executing" % (thread.get_ident(),)
        if (counter_lock.acquire(0)):
            global counter
            print "Thread A: Acquired lock...pausing..."
            counter += 1
            time.sleep(20)
            counter_lock.release()
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
            else:
                print "Thread A: Couldn't get lock"
            time.sleep(5)

def thread_incrb():
    while 1:
        print "Thread B(%d): Executing" % (thread.get_ident(),)
        if (counter_lock.acquire(0)):
            global counter
            print "Thread B: Acquired lock...pausing..."
            counter += 5
            time.sleep(10)
            counter_lock.release()
        else
            print "Thread B: Couldn't get lock"
            time.sleep(5)

def thread_display():
    while 1:
        print "Display thread(%d): Counter is %d" % (thread.get_ident(), counter)
        time.sleep(5)
thread.start_new_thread(thread_display, ())
thread.start_new_thread(thread_incra, ())
thread.start_new_thread(thread_incrb, ())
while 1:
    pass
```

如果执行此脚本，将得到如下所示的输出结果：

```
Display thread(1026): Counter is 0
Thread A(2051): Executing
Thread A: Acquired lock...pausing...
Thread B(3076): Executing
Thread B: Couldn't get lock
Thread B(3076): Executing
Thread B: Couldn't get lock
Display thread(1026): Counter is 1
Thread B(3076): Executing
Thread B: Couldn't get lock
Display thread(1026): Counter is 1
Thread B(3076): Executing
Display thread(1026): Counter is 1
Thread B: Couldn't get lock
Display thread(1026): Counter is 1
Thread B(3076): Executing
Thread B: Acquired lock...pausing...
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
Thread A(2051): Executing
Thread A: Couldn't get lock
Display thread(1026): Counter is 6
Display thread(1026): Counter is 6
Thread A(2051): Executing
Thread A: Acquired lock...pausing...
Display thread(1026): Counter is 7
Thread B(3076): Executing
Thread B: Couldn't get lock
Display thread(1026): Counter is 7
Thread B(3076): Executing
Thread B: Couldn't get lock
```

从输出的结果能看出第一个线程 A 是怎样获取锁和更新 counter 计数器及持有锁的：在第一个线程持有锁时，第二个线程 B 继续尝试执行并获取锁，接下来这个过程完全颠倒过来，在线程 B 得到锁的时候，线程 A 获取锁失败。与此同时，显示线程继续输出当前 counter 计数器的值。

9.3.7　高级线程

虽然 thread 模块为大多数线程实现提供了足够的工具，但它还是不适合于所有情况。比如，在 thread 模块中就没有办法控制父线程的子线程的操作或执行。在线程间通信和控制访问变量的方法上也受到限制，其中线程间通信不使用全局变量、资源或父线程的名称空间。

threading 模块通过提供以下机制来解决上述问题：更高级的新线程初始化方法、大量的块系统、信号量和其他线程间通信系统。其中，线程间(interthread)通信系统对父线程和子线程之间的通信有较大帮助。

我们简短地看一看此类灵丹妙药；所有的解决方案都以大量新对象类为基础，但是在模块内有一般的实用函数，这些实用函数允许对当前环境进行监控。这些函数就是：

activeCount()函数	返回当前活动的 Thread 对象的数目。
CurrentThread()函数	返回当前执行线程的 Thread 对象。这在线程内是有用的，能作为确定关于线程自身内部的信息的方法。
enumerate()函数	返回一个当前所有活动线程对象的清单，而不管这些线程对象处于什么样执行状态。

实际上，threading 模块和它的类建立在 thread 主模块之上。虽然 threading 模块不支持所有关于线程创建及线程锁定机制的不同解决方案，但实际上它所做的是增添基本方法来支持许多操作系统/语言接口提供的功能。

1. 线程对象

系统的核心是 Thread 类。Thread 类定义了一条单独的线程并用此线程来执行函数，执行函数所用的方法与 thread.start_new_thread()函数一样。创建一个新的类实例的基本格式如下所示：

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

Thread(group=None，target=None，name=None，args=()，kwargs())

其中，target、args 和 kwargs 参数是与 thread.start_new_thread()函数的 func、args 和 keywordargs 参数等价的参数。group 参数为将来的扩展而保留，且暂时能安全地被忽略。name 参数是想给线程起的名字。线程名字仅仅是符号，并且对线程的执行及控制没有任何影响。默认的名字是“Thread-N”，其中：N 是顺序排下来的数字。

例如，为上述计数器线程例子中的计数器线程创建一个新的 Thread 对象，就可以使用如下语句：

my_display_thread = Thread(None, thread_display, None, (), {})

其结果是个支持下列方法的 Thread 对象：

t.start()方法　当线程创建时，它不立即执行；必须调用此函数的 start()方法来使线程确实被执行。实际所发生的是 start()调用 run()方法，run()方法本身调用在 Thread 对象创建时指定的函数。

t.run()方法　当线程启动时，t.run()方法被调用。默认情况下，t.run()方法调用在对象创建时指定的函数。但是，还能用您想在以 Thread 线程为基础的新类中执行的语句重载 t.run()方法。

t.getName()方法　获取线程的名字。

t.setName()方法　设置线程的名字。

t.isAlive()方法　如果线程是活动的(即，当前正执行)，则返回 1，否则就返回零。对于 start()方尚未调用的或已终止的线程而言，将返回零。

t.setDaemon()方法　将一个线程做后台处理，所用方法是：在调用 start()方法之前调用 setDaemon()方法。被后台处理的线程采用与被后台处理程序相同的方式继续执行，被后台处理程序(如被 os.setpgprp()控制的那样)在主程序完成以后继续执行。因此，利用 setDaemon()方法，能启动一个 Unix 下诸如 named 及 httpd 后台程序的多线程化服务器，或者以相似的方式在 Windows NT/2000 系统下提供服务。

2、锁对象

threading 模块用两个新的锁对象来增添基本锁机制。它们是：原始锁和可重入锁。

(1) 原始锁　原始锁采用与 thread 模块内的基本锁机制相同的方式工作。原始锁具有两种状态：锁定或未锁定。能使用 acquire()和 release()方法获取或者释放锁。要创建一个新的原始锁，可使用如下代码：

mylock=Lock()

acquire()方法以与 thread 锁(线程锁)的 acquire()方法类似的方式工作，默认设置是线程将被阻塞一直到获取到锁为止(与不管是否获取到原始锁都立即返回正相反)。要改变这种行为，给 acquire()方法提供一个为零的单个参数。如以前所述，如果获取成功，此方法就返回 1；如果不能获取原始锁，就返回零。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
mylock.acquire(1)
```

要释放原始锁，请使用 release()方法，如下所示：

```
mylock.release()
```

(2) 可重入锁　可重入锁是用于“锁定”类的，但同一个锁能在同一个线程内被多次获取和释放。就一般情况而言，一旦锁被一个线程获取，它就不能被再次获取，直到锁被释放为止。利用可重入锁，能在一组嵌套语句内释放和获取同一个锁。最外层的 release()语句仅把锁复位到其未锁状态。

系统采取的策略是调用 acquire()就递增可重入锁状态。如果可重入锁已被当前线程获取，则再次调用 acquire()只是递增计数，就立即返回，且返回值为真(true)。在调用 release()时，计数器是递减的——只有当计数器达到零时，可重入锁才真正地被放弃。

要创建可重入锁，请使用 RLock 类，如下所示：

```
myrlock=RLock()
```

Lock 类上的 acquire()和 release()方法与前面部分所述 Lock 类的工作方式一样。

3. 条件变量

条件变量是前面描述的锁机制的更加高级形式，条件变量不是简单地提供锁机制，条件变量能被用来表明线程状态或者事件到线程的特殊变化。例如，在一个线程产生用于另一个线程处理的输出情况下，处理输出的线程可能正等待访问一个共享变量的数据。同时，产生输出的线程不得不等候处理输出数据的线程完成之后再添加新的数据。

在这种情况下，使用一个简单的 Lock 类或者 RLock 类都使程序很复杂。条件变量通过简化以下过程来解决上述问题：确定状态和通知一条等待线程某个变量可以使用了。

利用 Condition 指令能创建一个新的条件变量，如下所示：

```
mycondition=Condition([lock])
```

选项参数 lock 应是 Lock 类或 RLock 类的一个实例。如果不指定参数的值，那么一个新的 RLock 实例就被创建。新的对象支持以下方法：

mycondition.acquire(*args)	获取基本锁，调用与基本锁相对应的正确的 acquire()方法。如果给出 args 参数的值，args 参数就被逐字逐句地传递给基本锁的 acquire()方法。
mycondition.release()	释放基本锁。
mycondition.wait([timeout])	等待来自于另一个线程的通知，等待的时间为 timeout 参数定义的长度，或者就不确定地等待。此方法一旦被调用，基本锁就被释放，线程就暂停，一直到收到另一个线程对给定锁的 notify()事件或 notifyAll()事件。一旦收到事件，给定锁就被重新获取，此方法返回。如果操作时间超出 timeout，给定锁就被重新获取，执行将正常继续下去。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

mycondition.notify()	将一个通知事件发送给到当前正在等待的线程。请注意，此操作实际上不释放锁；必须调用 notify()和 release()两个方法来允许其他线程继续使用该锁。
mycondition.notifyAll()	通知所有等待的线程。

作为一个举例，下面简单的脚本将一条线程的消息传递到另一个线程去打印。在发送消息的线程中使用 notify()方法指示等待线程一条新消息已准备好，可以将消息打印出来了。很显然，这是一个简化了的例子，但是它演示了在两条线程之间传递一个变量的状态是多么容易。

```
import threading, time

mycondition = threading.Condition()
mymessage = ''

def thread_send():
    global mymessage
    counter = 0
    while 1:
        mycondition.acquire()
        mymessage = 'New Message: '+str(counter)
        counter += 1
        mycondition.notify()
        mycondition.release()
        time.sleep(5)

def thread_receive():
    global mymessage
    while 1:
        mycondition.acquire()
        if mymessage:
            print "Display:",mymessage
            mymessage = ''
            mycondition.wait()
        mycondition.release()

threada = threading.Thread(None, thread_send, None)
threadb = threading.Thread(None, thread_receive, None)

threada.start()
threadb.start()

while 1:
    pass
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

4. 信号量

信号量(semaphore)是一种以计数器为基础的基本锁机制。调用 acquire()方法则递减计数器值，而调用 release()方法则递增计数器值。 如果计数器值达到零，那么 acquire()就休眠一直到另一个线程调用 release()方法为止。信号量能被用来表明给定线程或其他对象的一种特殊状态。比如，可以使用与数组连在一起的信号量来表明数组中有多少个项——这样，一旦数组项数目变为零，就能阻塞执行和删除数组信息的线程。

要创建信号量，请使用 Semaphore 类，如下所示：

```
mysemaphore=threading.Semaphore([value])
```

选项参数 value 的值指明信号量初始值；其默认取值为 1。value 参数值应是一个正整数。有关信号量的 acquire()方法和 release()方法的格式简要说明如下：

mysemaphore.acquire([blocking])	acquire()方法试图获取信号量。 如果信号量的计数器值比零大，acquire()方法就递减计数器的值，并且立即返回。 如果计数器值已经为零，则此方法就休眠一直到另一个线程调用 release()方法为止。blocking 参数与其他锁的 blocking 参数一样。
mysemaphore.release()	递增信号量的内部计数器值。 如果计数器值为零，调用 release()方法就释放一条等待线程。

5. 事件

事件用于通过标记其他线程可能等候的特定事件来在线程之间通信。事件对象是个内部标志，它能随意地设置及清除。此外，wait()方法允许线程等待一直到标记被设置为止。要创建一个事件对象，其代码如下所示：

```
myevent=Event()
```

内部标志初始被设置为假(false)。事件实例所支持的方法如下：

myevent.isSet()	如果内部标志为真，myevent.isSet()方法就返回真。
myevent.set()	设置内部标志为真。
myevent.clear()	清除内部标志，设置内部标志为假。
myevent.wait([timeout])	等候 myevent 的内部标志变为真。此方法不是无限地等待，就是等到 timeout (指定为一个浮点数，以秒为单位)延迟期满。很明显，如果内部标志已经设置为真，则此方法立即返回。

9.3.8　队列

虽然上述处理消息的不同方法在大多数情况下能充分地工作，但这些方法还不能解决所有问题，尤其在处理相互之间需要进行海量数据通信的线程时。虽然能使用上面所示的方法(有效地共享信息)共享对象，但特别是，如果供应者和用户线程以不同的频率工作，或处理

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

不同的数据块，这就要在变得很难解决的冲突形势中结束。Queue 模块提供了一个解决方案，它提供一个类，该类建立一个多供应者和多用户的 FIFO (先进先出)队列。利用此对象，能把新的请求添加到队列中，并处理来自于队列的请求，而无须担忧线程的处理频率、工作量大小及其他限制，也不会遇到手动处理对象锁所遇到的难题，可以无忧无虑地继续处理排队的请求。

要创建一个新的队列，必须创建一个 Queue 类的实例，如下所示：

```
myqueue=Queue(maxsize)
```

其中：maxsize 参数定义能放到队列中的项的最大数目。如果 maxsize 小于或者等于零，那么就没有最大数目的上限。选择一个适合的队列尺寸是重要的，因为不合适尺寸的选定将意味着只能提供极少的空间可供新对象使用；如果队列尺寸被设置得太高，或是无穷大，那么就要冒允许线程消耗数量越来越大的资源的风险。

虽然有办法控制提供给队列的项的数目，但最好根据预期处理的最大数目，将最大数目的一半左右设定为队列的大小。例如，要处理大约 10 000 个元素的程序应该设置 maxsize 为 5000 左右。如果队列元素的尺寸特别大，那么根据想要分配给应用程序的最大空间的数量来乘上系数来求出 maxsize 的大小。

队列的实例支持下列方法：

myqueue.qsize()	返回队列的近似尺寸；因为其他线程可能正在更新此队列，所以很难确定一个确切的数目。
myqueue.empty()	如果队列为空，就返回 1；否则，返回零。
myqueue.full()	如果队列满，就返回 1；否则，返回零。
myqueue.put(item [, block])	将 item 对象加入到队列中去。如果给出 block 参数的值，并且是个正数，则调用者就休眠一直到队列中有一个空闲位置可用为止。如果 block 参数未给出或为零，则在队列满时，产生 Full 异常。
myqueue.put_nowait(item)	等价于 myqueue.put(item, 0)。
myqueue.get([block])	获取队列的一项对象。如果 block 参数给出，并且是个正数，则调用者就休眠一直到队列中有一项对象有效为止。如果没给出 block 参数的值，或者 block 参数的值为零，则在队列为空时，产生 Empty 异常。
myqueue.get_nowait()	等价于 myqueue.get(0)。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第10章 信 息 处 理

大多数应用包括几种不同形式的数据操作。数据操作应用可能是将几个数字简单地加在一起，也可能是从日志文件中抽取独立的字段生成一个报表。不管是哪种情况，都需要将信息从一种格式或者数值处理成为另一种格式或数值。

Python 的数据操作依赖内置运算符和提供附加功能的外部模块组合起来。例如，math 模块提供三角函数，因此能在仅给出顶角及其对边边长的条件下计算出三角形的侧边边长。

对文本操作而言，有大量有效解决方法，这包括内置运算符和字符串函数。对于更高级的文本操作来说，可以使用正则表达式来匹配元素，它能达到惊人的匹配控制程度。

本章将关注以下两个方面内容：首先是数据操作，接下来解释如何处理包括二进制结构和 Python 的 Unicode 系统在内的特殊数据类型。

10.1 操作数字

大多数数字操作能利用内置运算符范畴内的运算符完成，内置运算符直接对数字对象进行操作。所有基本算术操作，诸如加、减、求以一数值为底数另一数值为幂的值等等，都被Python运算符内部支持。

有关其他算术函数和三角函数，需要 math 或 cmath(关于复数)模块。而 random 和 whrandom 模块则提供用于产生随机数的不同方法。 所有这些模块都在本节中进行讨论。

10.1.1 math

math 模块提供基本的算术函数和三角函数。请注意，所有的三角操作函数都是以弧度为单位，而不是以度为单位。可以利用公式 d*((2*pi)/360 可以将角度转换为弧度，其中：d 是以度为单位的值；要将弧度数值转换回角度值，请使用公式 r*(360/(2*pi))。

math 模块提供两个常量：其一为 pi，圆周长对其直径的比率；其二为 e，自然对数。

表 10-1 列出了 math 模块支持的函数。

表 10-1 math 模块所支持的函数

函 数	说 明
acos(x)	返回 x 的反余弦值
asin(x)	返回 x 的反正弦值
atan(x)	返回 x 的反正切值
atan2(y，x)	返回 atan(y/x)的值
ceil(x)	返回 x 的上限(下一个整数)值

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

（续表）

函　数	说　明
cos(x)	返回 x 的余弦值
cosh(x)	返回 x 的双曲余弦值
exp(x)	返回 e 的 x 次幂(e**x)
fabs(x)	返回 x 的绝对值
floor(x)	返回 x 的下限(整数部分)值
fmod(x，y)	返回 x%y 的值
frexp(x)	返回一个元组，此元组包含 x 的正尾数和 x 的指数值
hypot(x，y)	返回勾股定理(Pythagoras's theorem)的结果，即，sqrt(x**2+y**2)
ldexp(x，i)	返回 x*(2**i)的值
log(x)	返回 x 的自然对数值
log10(x)	返回以 10 为底 x 的对数值
modf(x)	返回一个元组，此元组包含 x 的小数部分和 x 的整数部分。这两个值的正负号与 x 的正负号相同
pow(x，y)	返回 x**y 的值
sin(x)	返回 x 的正弦值
sinh(x)	返回 x 的双曲正弦值
sqrt(x)	返回 x 的平方根值
tan(x)	返回 x 的正切值
tanh(x)	返回 x 的双曲正切值

应该对ceil()和floor()函数更深一步地加以解释一下。这两个函数都返回整数，表示为float类型。这个返回值或者是给定表达式的整数部分，或者是给定表达式的下个整数。这与四舍五入的原则相似，所不同的是：ceil()函数总是向上舍入而为其最邻近的整数，也就是大于或等于给定参数的最小整数；而floor()函数总是向下舍掉，取其最邻近的小于或等于给定参数的整数。当交互地使用ceil()和floor()函数时，就能比较出ceil()和floor()结果的区别，请见下例：

```
>>> from math import *
>>> ceil(9)
9.0
>>> ceil(8.9)
9.0
>>> ceil(8.1)
9.0
>>> floor(9.9)
9.0
>>>
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

10.1.2 cmath

cmath 模块等价于 math 模块，惟一不同的是：cmath 模块对复数进行操作。cmath 模块还提供了 pi 和 e 这两个常量的复数版本。表 10-2 列出了 cmath 模块支持的函数。

表 10-2 cmath 模块所支持的函数

函 数	说 明
acos(x)	返回 x 的反余弦值
acosh(x)	返回 x 的反双曲余弦值
asin(x)	返回 x 的反正弦值
asinh(x)	返回 x 的反双曲正弦值
atan(x)	返回 x 的反正切值
atanh(x)	返回 x 的反双曲正切值
cos(x)	返回 x 的余弦值
cosh(x)	返回 x 的双曲余弦值
exp(x)	返回 e 的 x 次幂(e**x)
log(x)	返回 x 的自然对数值
log10(x)	返回以 10 为底 x 的对数值
sin(x)	返回 x 的正弦值
sinh(x)	返回 x 的双曲正弦值
sqrt(x)	返回 x 的平方根值
tan(x)	返回 x 的正切值
tanh(x)	返回 x 的双曲正切值

10.1.3 随机数字

计算随机数在许多编程领域中都有用。引入一个随机元素是用来给 ID 或引用号(reference number)添加一个惟一或不能猜测到的数值的主要方法；随机元素也能在诸如需要一种不能预料结果或者无法预言结果的游戏应用中使用。

生成真正的随机数是很难的，对计算机而言，就更不可能了，因为计算机被设计为对精确和可预言的数字进行处理。人吗?可能，在产生随机数这方面还好些，但不能总是带个便携式的人为您生成随机数吧！

大多数随机数生成器被正式地列为准随机数生成器。而且它们大多依赖于计算中不完善的或微小的差值来产生随机结果。不能单独依赖这些随机数生成器中的任何一个生成器来为临时 ID 产生随机数。有关利用随机和静态的元素创建惟一 ID 的方法，请参见下面“创建随机 ID”部分。

Python 的标准系统配置包含两个有关随机数的模块，分别为 random 模块和 whrandom 模块。random 模块提供了许多计算随机数的不同函数。此外，random 模块导出 whrandom 模

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

块里的函数。whrandom 模块为创建随机数(利用 Wichmann-Hull 算法) 提供了一个更常见的接口，使用的仅仅就是这 4 个基本函数：randint()、random()、seed()和 choice()函数。

seed()函数给随机数生成器种子值。它接受 3 个参数，如下例所示：

```
seed(1,2,3)
```

创建随机标识符(ID)

如果想创建一个会话 ID，而此会话 ID 又必须是惟一的，就不能单独使用随机数。虽然就随机的基本定义而言，所产生的数字一定是随机的，但现实是：由它们的本性决定最终产生的将是一个自然序列。

要避免此类问题，可把当前时间和随机数的一个组合用作 ID。将当前时间精确计算到最近的秒值(实际上，能更精确地计算时间，但是精确到秒值的时间计算在所有平台上得到支持)。因此，如果每次会话都被最新创建，就需要实际上在同一时间的同一时刻上依次提供极大数量的请求，此请求是关于要复制的整个 ID 字符串。虽然有许多方法能实现上述所需，而作者多年使用的一种方法如下所示：

```
def make_session_id():
 from whrandom import randint
 from time import gmtime
 (a,b,c) = (randint(0,9999),
 randint(0,9999),
 randint(0,9999))
 (year, month, day, hour, minute, second) = gmtime()[0:6]
 session = "%02d%04d%02d-%02d%02d%04d-%d%d%d" % \
 (second,a,hour,month,minute,b,c,day,year)
 return session
```

在上述函数调用后，make_session_id 的结果，是个看起来像 52854910-08398569-891732001 一样的字符串，这在不能保证结果是完全随机和完全惟一的情况下，大概已足够接近于当前给定的 CPU 极限。作者已检查了利用相同物理时间产生的 10 000 000 个结果，却没有发现一个完全相同，因此这一定是相当可靠的方法了。

如果这三个参数都默认或者三个参数给定为同一个值，则 seed()函数就把当前时间用作 seed 数值。如果第一次没先调用 seed()函数，而使用了其他三个函数之中的任一个函数，则 seed()函数被自动调用。

random()函数返回 0.0~1.0 范围内的一个随机数，如下所示：

```
>>> whrandom.random()
0.44718597724016607
    >>> whrandom.random()
0.93284180701215091
    >>> whrandom.random()
    0.03717170343673537
```

randint()函数接受两个参数，这两个参数表明将要产生随机数的整数范围。例如，要产

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

生一个1~256(包括256在内)范围内的随机数，如下所示：

```
>>> whrandom.randint(1,256)
24
```

要产生一个位于100~1000范围内的随机数，所用命令如下所示：

```
>>> whrandom.randint(100,1000)
291
```

最后，choice()函数的功能是从给定序列中随机地选取一个元素。请见下例：

```
>>> whrandom.choice([1,3,5,7,11,13,17,19,23,29])
23
```

请注意，随机选取的项不是从给定序列中删除，它仅仅是被选取和返回的项。

random模块导出函数来产生随机数，这些函数利用实数上的不同分布算法来产生随机数。所有这些函数可能用随机偏差产生随机数，这要好于whrandom模块的randint()和random()函数，但需要一点性能开支。表10-3列出了random模块导出的函数。

表10-3 random模块导出的函数

函数	说明
betavariate(alpha,beta)	返回0~1范围内的一个值，所用分布是Beta分布。其中，alpha和beta应大于-1
cunifvariate(mean,arc)	返回(mean-arc/2) ~ (mean+arc/2) 范围内的一个值，所用分布是circular平均分布
expovariate(lambda)	返回0~∞范围内的一个值，所用分布是指数分布
gamma(alpha, beta)	返回一个值，所用的分布是Gamma分布。其中：alpha参数值应大于-1，而beta参数值应大于0
gauss(mu, sigma)	返回一个值，所用分布是高斯分布。高斯分布的均值(mean)为mu参数值、标准偏差(standard deviation)为sigma参数值
lognormvariate(mu, sigma)	返回一个值，所用的分布是均值为mu参数值、标准偏差为sigma参数值的正态对数分布
normalvariate(mu, sigma)	返回一个值，所用的分布是均值为mu参数值、标准偏差为sigma.参数值的正态分布
paretovariate(alpha)	返回一个值，所用的分布是形状参数为alpha参数值的paretovariate分布
vonmisesvariate(mu,kappa)	返回一个数值，所用的是von Mises变分。其中，mu是0~2*pi间的平均角弧度(mean angle radians)，而kappa是个正数的集中因子(concentration factor)
weibullvariate(alpha, beta)	返回一个值，所用的是Weibull变分。其中，变分的形状(shape)参数是beta，而标量(scalar)参数是alpha

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

10.2 文本操作

Python 中的字符串是对象。并且由于 Python 强大的运算符支持，诸如像把两个字符串合并成一个字符串这样简单的任务，就能用运算符+很轻松地完成。字符串还是个序列，那就意味着能对单个字符进行寻址，或者对一个字符序列(一个切片)进行序列化，而无需借助于额外的函数来抽取所要的字符。

但是其他操作呢？像改变字符串的大小写或把一个文本记录分离成为不同的字段，其中文本记录是由一个单字符限定界限的。对这样的操作，如何进行处理呢？

在这里举个例子：假如说想用文本的另一个部分替换Python字符串的文本的一部分。例如，在表达式“the cat on the mat”中，将“cat”替换为“dog”。尽管能使用index()方法查找到“cat”这个单词的位置，却不能替换它。字符串是不可改变的——不能直接替换序列中的字符，即使替换字符与您要替换掉的字符长度一样，也不能替换。可以通过以下方法实现字符串中某个单词的替换：先查到要替换掉单词在字符串中的位置，切分出该单词之前的字符串序列和该单词之后的字符串序列，再重新组装字符串。

这样仍很杂乱，不是吗？如果原来的表达式是“the Cat sat on the mat”或是“The Cat Sat on the Mat”，将会怎样？情况仍然很糟！如果想把“Cat”和“Mouse”都替换为“Elephant”，会怎样呢？就必须统计单词的所有不同大小写和所有可能的不同单词，为的是实现正确的替换。

另一个选择是，使用正则表达式。正则表达式实际上是另一种语言，一种允许指定什么是要替换掉的语言。这种语言能应付以下从简到难的各种问题：简单的如，用 dog 替换 cat，将 cat 和 mouse 都替换为 elephant，一直到复杂的如，用 dog 替换 cat，但是仅当“the cat is sat on the mat”时，并且仅当这些英文单词全部是小写时才进行相应的替换。

对于大多数基本字符串操作，除了能依赖 string 对象类型提供的内置函数外，还能使用 string 模块。对于正则表达式，就要用 re 模块。本部分就集中讨论 string 和 re 这两个模块。

10.2.1 基本字符串操作

string 模块定义了许多有用函数，这些函数主要用于字符串的操作、字符串的抽取和字符串的替换。本节对 string 模块加以详细的介绍，包括用于字符串操作的内置方法和运算符。

关于本节，自始至终，重要的是要记住：必须关注所有这些函数的返回值——除非在具体的字符串对象上使用某个方法，否则 Python 不改变字符串也不修改字符串。

1. 查找字符串段

要在一个字符串内查找某个特殊串的位置，请使用 index()函数。例如，要在“the cat sat on the mat”字符串内查找“cat”串的位置，就请使用如下命令：

```
loc = string.index('the cat sat on the mat','cat')
```

其中：loc 变量现在应包含一个为 4 的值，由于所查找的单词在串(一定记住：字符串的索引值从 0 开始)中的第 5 个字符上出现。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

index 函数还接受第 3 个参数——在字符串中开始搜索的索引位置。例如，如果要查找“at”，但不是对“cat”中的“at”感兴趣，就可以使用如下语句：

```
lastat = string.index('the cat sat on the mat', 'at', 8)
```

最后，还能使用 rindex 函数从字符串的末尾处开始从后向前反向查找，而不是从字符串的开头开始向后查找。请注意，字符串的顺序不会受到影响，只是开始查找文本的开始位置有所不同而已。例如，要抽出单词“on”，从字符串的末尾处开始查找，就可以使用如下命令：

```
rloc = rindex('The cat sat on the mat', 'on')
```

2. 替换字符串片段

在 Python 中，string 对象不支持赋值，像下面的命令就无效：

```
text[4:7] = 'tyrannosaurus rex'
```

如果试着执行上述命令，就引发 TypeError 异常。替代上述命令，最直接了当的解决方法是使用切片对象来抽取要替换文本之前和之后的两个部分，然后重新连接成新的字符串，如下所示：

```
text = text[:4] + 'tyrannosaurus rex' + text[7:]
```

另一个选择是，如果能确保能在不略过任何元素的情况下找到所搜寻的字符串，请使用 string 模块的 replace()函数，如下所示：

```
replace(text, old, new [, max])
```

本例，在字符串 text 中，使用 new 替换 old，替换的次数为找到 old 字符串的次数，或者在给出 max 参数时，替换的次数就为 max 参数值。因此，使用 replace()函数对前例重新编写代码如下所示：

```
text = string.replace(text, 'cat', 'tyrannosaurus rex')
```

也能直接在字符串上使用 replace()方法，如下所示：

```
text.replace('cat', 'tyrannosaurus rex')
```

请注意：不管怎样，查找总是从字符串的开头开始。一旦替换，“at”就改变为“cat”。不能从字符串的开头之外的任何其他位置处开始查找。如果想替换在字符串末尾处出现的单词，既可以使用正则表达式，也可以使用切分对象仅改变原字符串中要修改的部分。例如：

```
text = text[:-3] + string.replace(text[-3:], 'at', 'oon')
```

text 变量就被改变为“the cat sat on the moon”。

3. 分离

Python 的 split()函数是作为 string 模块一部分出现的，它允许用单字符或多字符序列分

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

离一个字符串。例如，读者可能想将一个由逗号分界字段的列表分离为一个不同字段组成的列表。split()函数分离一个字符串的各个单元，最后返回一个列表。split()函数的一般格式如下：

```
split(text, [, expr [, max]])
```

其中：text 参数是字符串，这个字符串是要分离的字符串；expr 参数是个分隔字符。如果不给出 expr 参数，Python 就假设用空格字符对它进行分离。例如，能利用如下命令将一个文本语句转换为一个个单词的列表：

```
words = string.split('I pushed the button but nothing happened')
```

还能使用 split()函数从一个包含分界字符的字符串中提取独立的字段出来，诸如：

```
fields = split('rod:IA266:$23.99',':')
```

其中：如果提供max参数，其作用就同Perl版本一样，限制分离操作发生的次数。

如果想执行一个基于正则表达式的分离，请使用re模块的split()函数，如下所示：

```
import re
fields = re.split(r'[\s,:;]+', text)
```

上面的语句，不论 text 字符串是由一个还是多个空格字符、逗号、冒号和半冒号分界，切分 text 字符串中的字符。参见本章后面的“正则表达式”部分。

4. 连接

虽然在 Python 中能使用运算符+将两个字符串连接在一起，但如果有许多字符串要使用相同的字符或者序列连在一起，那这个过程就十分乏味了。例如，试想一下，要把以前分离的句子重新连接起来，由于不知道此句子当中的单词列的长度，通常需要使用循环，如下所示：

```
s = ''
for word in words:
    s += word + ' '
```

利用上述循环的结果是在合并起来的字符串末尾添加一个无用空格字符。

解决此问题的方法是：使用 string 模块的 join()函数。join()函数利用同一个分隔字符把一个列表中的所有元素合并在一起，并且将分隔字符放在串起来的两个元素中间，如下例所示：

```
names = string.join(['martin', 'sharon', 'wendy', 'rikke'], ', ')
```

如您所见：join()函数的第一个参数是要连接在一起的列表或元组；第二个参数是要放置在每个单元对中间的分隔符。names 变量的最终结果是：

```
'martin, sharon, wendy, rikke'
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

5. 截断

在 Python 中，利用 lstrip()、rstrip()和 strip()函数能截掉一个字符串前导和 / 或尾随空格字符。这儿的助记符号是：lstrip()函数移去左面的空格字符，而 rstrip()函数移去右面的空格字符，如下所示：

```
string = ' leading space trailing '
lstrip(string) # returns 'leading space trailing '
rstrip(string) # returns ' leading space trailing'
strip(string) # returns 'leading space trailing'
```

6. 改变大小写

Python 通过 string 模块支持一批大小写转换函数。这些函数改变字符串中字符的大小写然后返回一个新版本的字符串。表 10-4 列出了这些函数。例如，要把一个单词改变为小写，可以使用如下命令：

```
lctext = string.lower(text)
```

表 10-4　Python 的大小写转化函数

函　　数	说　　明
string.lower(text)	将 text 参数字符串中的所有字符改变为小写
string.upper(text)	将 text 参数字符串中的所有字符改变为大写
string.capitalize(text)	将 text 参数字符串中的第一个字符改变为大写
string.capwords(text)	将 text 参数字符串中每个单词的第一个字符改变为大写
string.swapcase(text)	将 text 参数字符串中的所有字符的大小写进行交换(即小写的字符改变为大写字符/大写的字符改变为小写字符)

7. 转换字符

虽然有时所想做的就是改变一下字符串的大小写，但也有想转换单个字符的时候。例如，试想一下，当需要一个“aaaaaa”串，具有的却是“ffffff”串时，可以手工转换，但有一种更轻松的方法——使用 string 模块的 translate 函数。

translate 函数接受两个参数：一个字符串和一个转换表。转换表映射要查找的字符和用来进行替换的字符。这个映射关系需要由 maketrans()函数生成。maketrans()函数本身接受两个参数：一个要被转化的字符和要转化成为相应字符的列表。例如，利用下面的命令就能将所有小写字符改变为大写字符：

```
string.translate(text, maketrans('abcdefghijklmnopqrstuvwxyz',
                                 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'))
```

这两个字符串依次按顺序配对使用。在上例中，第一个字符串中的第一个字符是“a”，第二个字符串中的第一个字符是“A”。那么，当字符串被转换时，每一个出现的“a”都被

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

替换为“A”。

让我们看另一个例子，将小写字符转换为其互补字符，即：将“a”转换为“z”，而将“z”转换为“a”，如下所示：

```
string.translate(text, maketrans('abcdefghijklmnopqrstuvwxyz',
                                 'zyxwvutsrqponmlkjihgfedcba'))
```

对于前面的例子使用此映射表，“the cat sat on the mat”的转换结果就是“gsv xzg hzg lm gsv nzg”！

8. 标准字符定义

除了前面所述的函数外，string 模块还定义了一些字符串序列常量，它们能用在本节所述的许多函数中。表 10-5 列出了能用在像 maketrans()这样的函数中指定具体字符组的常量。

表 10-5　字符串序列

Perl 表达式	Python 等价表达式
[0-9]	Digits
[0-9A-F]	Hexdigits
[a-zA-Z]	Letters
[a-z]	Lowercase
[A-Z]	Uppercase
[0-7]	Octdigits
\s	Whitespace

10.2.2　正则表达式

正则表达式是个简单方法，它描述函数中用于搜索正文的查找条件。在最简单的情况下，正则表达式仅由原始文本组成。但是正则表达式的威力还在于能指定通配符元素，因此能对任何字符或字符序列、重复的和更复杂的序列进行匹配。总之，最好是把正则表达式当作一种语言来对待。

有两种基本的正则表达式操作：

- 匹配，就是在一个文本字符串中搜索并希望匹配一特殊表达式；
- 置换，就是在一个字符串中查找和替换信息。

例如，返回头来再看看“cat”这个例子，不用正则表达式而将“cat”替换为“dog”是很难的，但不是不可能的。利用正则表达式置换，能很容易地做到这一点，如下所示：

```
text = 'the cat sat on the mat'
text = re.sub(r'cat', 'slug', text)
```

此外，能在不知道要替换的单词的情况下在文本上进行置换。例如，试想一下，要改变句子当中的动物名，但未必知道是什么动物，能用正则表达式通过鉴别“the”和“sat”之间

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

的单词来完成这个置换，如下所示：

```
text = 'the elephant sat on the mat'
text = re.sub(r'the.*?sat', 'the slug sat', text)
```

其中：在上例中，正则表达式是“the.*?sat”。“the”和“sat”这两个单词，大概认识吧。其中间的几位，“.*?”，是个正则表达式片段。其中，点字符表示与任何一个字符进行匹配；星号(*)表明此前的正则表达式元素(即点字符)应被匹配零次或多次；而问号(？)表明通配符匹配“.*”应该只“吞没”直到下个句子("sat")的所有字符。其结果是用“he slug sat”换了“the <something> sat”，其中<something>就是正则表达式片段“.*?”匹配的。

1. 正则表达式组成元素

在了解 re 模块提供的正则表达式机制的特性之前，需要先看看正则表达式引擎支持的正则表达式元素。

首先，所有生文本确实如它所见一样被匹配——如果包括了单词“cat”，实际上，正则表达式引擎匹配其后为字母“a”的字母“c”，而字母“a”后面又紧跟着字母“t”。如果包括一个句点(.)，正则表达式引擎就匹配表达式中任何一个单个字符，如“c.t”就匹配“cat”、“cot”、还有“c!t”和其他任何组合。

最后，有两个特殊的运算符，用来匹配字符串的开头或结尾。例如，在“the cat sat on the mat”这个字符串中，用“a”对“the”这个单词做置换可能达不到所要结果。如果只是想替换第一个“the”，由于“^”字符只能匹配字符串的开头，所以能通过规定搜索表达式为“^the”来实现这样的置换。还能用特殊字符“$”匹配字符串的结尾，因此能用“at$”这个正则表达式对上述字符序列最后的“at”作替换。表 10-6 概括了正则表达式的基本字符。

表 10-6 正则表达式的基本字符

字符序列	说明
text	匹配 text 字符串
.	匹配除换行符之外的任何一个单个字符(除非 s 标志在使用)
^	匹配一个字符串的开头
$	匹配一个字符串的末尾

任何正则表达式还能通过指定此表达式匹配的次数来更深一步地加以约束。例如，要匹配所有以“d”开始以“g”结尾的单词，使用正则表达式“d.*g”就能实现。点字符表明与任何一个字符匹配，而“*”表明匹配点字符零次或多次。

遗憾的是，在与“debug”匹配的同时，上述正则表达式也与“debugging”相匹配。事实上，上述正则表达式与任何以“d”开始以“g”结尾的字符序列都匹配，因此，它还与“do the right thing”匹配。

“.*”被称作最大匹配(maximal match)，因为此表达式能匹配源字符串中所有能匹配的那么多字符。由于这常常不是所要的，所以还能使用最小匹配。最小匹配只“吞没”一直到其随后项第一次出现的所有字符。因此，在“debugging”字符串中，“.*?”匹配到“debug”

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

就结束；而在“do the right thing”字符串中，匹配到“do the rig”就结束。

除了有匹配零次或多次这个匹配限定符之外，还有一次或多次、零次或一次及具体的限定符。表10-7包含了得到支持的限定符的完全清单。

表 10-7　匹配限定符

最　大	最　小	说　明
*	*?	重复匹配前表达式零次或多次，尽可能多或尽可能少地重复匹配
+	+?	重复匹配前表达式一次或多次，尽可能多或尽可能少地重复匹配
?	??	重复匹配前表达式零次或一次，尽可能多或尽可能少地重复匹配
{m}	{m}?	精确重复匹配前表达式 m 次，尽可能多或尽可能少地重复匹配
{m,}	{m,}?	至少重复匹配前表达式 m 次，尽可能多或尽可能少地重复匹配
{m,n}	{m,n}?	至少重复匹配前表达式 m 次，至多重复匹配前表达式 n 次，尽可能多或尽可能少地重复匹配

对于更复杂项的匹配，可使用一些字符序列来选取具体字符。例如，能利用“[]”序列匹配一个具体的字符集。使用“[]”序列能匹配一个字符范围，即“[a-z]”匹配仅为小写的字符，而“[!$?%]”匹配其中列出的符号，“[^0-9]”就匹配除数字 0~9 之外的所有字符。

还能进行匹配选择。例如，能使用“cat|mouse”与“cat”或“mouse”进行匹配。看起来像|一样的垂直线运算符在两个项之间扮演逻辑或(OR)的角色。

最后，还能使用组来匹配并抽取具体单元。例如，给定句子“the cat sat on the mat”，试想一下要确定猫是如何呆在垫子上的。利用正则表达式,必然能识别出这个单元，但必须了解实际的单词是什么。

利用组，能在不知道此单词是什么的前提下抽取出想要的单词。例如，正则表达式“the cat (.*?) on the mat”，其中，圆括号创建了一个新的组，在正则表达式执行之后，就进行抽取。组也能用在其他领域。例如，能使用正则表达式“([0-9]+)/([0-9]+)/([0-9]+)”从“23/6/2001”日期字符串中抽取元素。

表10-8列出了组和集运算符。

表 10-8　组和集运算符

组	说　明
[...]	匹配集合内的字符，例如，[a-zA-Z]或 [,./;']
[^...]	匹配集合之外的字符
A|B	既匹配表达式 A 又匹配表达式 B
(...)	表达式组
\number	匹配在 number 表达式组内的文本

除了已见过的特殊字符之外，还有能用来对具体字符类型或字符环境进行匹配的特殊字符序列。例如，“\b”匹配单词的边界，因此，正则表达式“food\b”和“food”、“zoofood”

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

匹配，而和“foodies”就不匹配。其他序列能用来匹配字符的具体类型。能使用正则表达式“\s”与任何空格字符匹配，并且能通过在这个表达式后添加限定符来匹配任何数量的空格字符。

例如，正则表达式“^\S+\s+\S+$”，基本上，匹配包含由一个或多个空格字符分开的两个单词组成的任何字符串，包括如下所有字符片段：

```
the cat
the cat
the          cat
the\ncat
```

表10-9列出了Python支持的特殊字符。还请注意，正则表达式引擎对诸如“\n”和“\t”的所有反斜线限定的字符按其通常解释进行匹配。

表 10-9 特殊字符序列

字 符	说 明
\A	只匹配字符串的开始
\b	匹配一个单词边界
\B	匹配一个单词的非边界
\d	匹配任何十进制数字字符，等价于 r'[0-9]'
\D	匹配任何非十进制数字字符，等价于 r'[^0-9]'
\s	匹配任何空格字符(空格符、tab 制表符、换行符、回车、换页符、垂直线符号)
\S	匹配任何非空格字符
\w	匹配任何字母数字字符
\W	匹配任何非字母非数字字符
\Z	仅匹配字符串的尾部
\\	匹配反斜线字符

最后一套正则表达式元素含有声明(assertion)，它能用在正则表达式中对具体事件声明，而实际又不影响表达式其余部分的处理。例如，有一列被冒号分开的数字，想把该列数字分成一个个含有 6 个数字的块，即，把“1:2:3:4:5:6:1:2:3:4:5:6”转换为["1:2:3:4:5:6: ","1:2:3:4:5:6"]——做到这一点，没有简易的方法。可以使用 string.split 把整个字符串分离成一个一个独立的单元，然后重新组装这些独立的单元成为六元素的块，但这可是太杂乱了！

对于上述要求，完全可以使用声明创建一个正则表达式组来匹配尺寸字段——即，匹配 6 次“#:”的表达式——接下来创建一个组来保存这个 6 元素字段序列。

实际声明看起来就像这样：“((?:[^:]*?:){.6})”。如果要重新再构造一下，重要的部分就是“(?:[^:]*?:)”，这部分创建一个新的正则表达式组，但这个组不填充正常被“()”填充的组表内，其结果是与“#:”匹配，然后计数这个字符序列六次，并把“#:#:#:#:#:#:”放到一个标准正则表达式组中。

表10-10包含了正则表达式系统所支持的声明的完全清单。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

表 10-10　正则表达式的声明

声　明	说　明
(?iLmsux)	此组匹配空字符串，iLmsux 字符匹配表 10-11 中所列相对应的正则表达式限定符。请注意，这些选项影响正则表达式紧随声明之后的部分，而不是整个正则表达式。此组还能用在：想要由搜索表达式所定义的效果，而不是由正则表达式函数所规定的时候
(?:...)	匹配圆括号内定义的表达式，但不填充字符组表
(?P<name>)	匹配圆括号内定义的表达式，但匹配的表达式还可用作 name 标识的符号组。请注意，这个组仍然填充正常的组匹配变量。要通过名字查阅一个字符组，就直接将它提供给 match.end()方法、match.group()方法或使用\g<name>
(?P=name)	匹配所有与前面命名的字符组相匹配的文本
(?#...)	引入注释，忽略圆括号内的内容
(?=...)	如果所提供的文本与下一个正则表达式元素匹配，这之间没有任何多余文本就匹配。这允许在一个表达式中进行超前操作，而不影响正则表达式其余部分的分析。例如，如果“Martin”其后紧随“Brown”，则“Martin (?=Brown)”就只与“Martin”匹配
(?!...)	仅当指定表达式与下一个正则表达式元素不匹配时匹配(就是(?=...)的反操作)
(?<=...)	如果字符串当前位置的前缀字符串是给定文本，就匹配，整个表达式就在当前位置终止。例如，(?<=abc)def 表达式与“abcdef”匹配。这种匹配是对前缀字符数量的精确匹配，因此，abc 和 a|b 匹配，但和 a*不匹配
(?<!...)	如果字符串当前位置的前缀字符串不是给定正文，就匹配(是(?<=...)的反操作)

实际上，在这儿我们只讲了有关正则表达式功能的一点皮毛而已，对于这个主题的论述的确能写成一本厚书。有关这方面的最好著作是 Jeffrey E. F. Friedl 编著的《掌握正则表达式》一书(由 O'Reilly 出版社于 1997 年出版)。

2. 正则表达式修饰符

大多数正则表达式还支持一些修饰符(也称作标志)。修饰符修改正则表达式的执行方式。作为一个参考，表 10-11 列出了这些修饰符。

表 10-11　正则表达式处理所支持的标志

修　饰　符	说　明
I 或 IGNORECASE	忽略表达式的大小写来匹配文本
L 或 LOCALE	使用用于确定\b、\B 和\w、\W 序列的本地设置
M 或 MULTILINE	使^和$应用于行的开头和结尾，而不是多行字符串的字符串上
S 或 DOTALL	强制.匹配所有字符，包括换行字符在内
X 或 VERBOSE	允许正则表达式忽略未被转义的空格字符和注释

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

3. 基本搜索/匹配

利用re.search()函数，能执行一个基本的搜索(或匹配)，re.search()函数的格式如下所示：

```
search(pattern, string [, flags])
```

例如，要确定字符串“cat”是否存在于 string 中，使用如下形式的命令即可：

```
if re.search(r'cat', string): ...
```

如果正则表达式不能被所给出的文本匹配，search()函数返回None，为假。如果匹配成功，此函数就返回一个MatchObject对象，这个对象能用来从字符组中提取文本。详见本章后面“使用MatchObject对象”部分的内容。

re 模块中的 match()函数是 search()任何函数限制更严格的版本，前者只匹配给出字符串的开头部分的表达式，而后者匹配字符串的任何部分。match()函数能用来强制脚本检查从字符串头部开始的匹配，而忽略用户给出的所有正则表达式。从本质上讲，此技术没有提供优于带有\A 前缀的搜索表达式的性能。

4. 抽取匹配的元素

如果想找到并匹配一具体表达式，最简单的方法是使用findall()函数。findall函数返回匹配给定表达式的列表，而不是匹配MatchObject对象的列表。findall()函数，在没使用字符组时，返回匹配的文本；如果使用一个组，findall()函数就返回所有匹配项的列表；如果使用多个组，返回列表的每个项就是元组，每个元组包含对应于每个组的文本。请见下面这个例子：

```
>>> import re
>>> string = 'the cat sat on the mat at ten'
>>> print re.findall(r'at',string)
['at', 'at', 'at']
>>> print re.findall(r'm.*?\b',string)
['mat']
>>> print re.findall(r'(at)',string)
['at', 'at', 'at']
>>> print re.findall(r'(cat)(.*?)(mat)',string)
[('cat', ' sat on the ', 'mat')]
```

5. 使用 MatchObjects 对象

search()和 match()函数返回 MatchObjects 对象，MatchObjects 对象包含发生匹配时有关匹配的字符组的内容和初始字符串的匹配位置这两方面信息。表 10-12 列出了由 m 表示的 MatchObject 对象的有效方法。

例如，利用如下脚本，能从字符串提取日期和时间：

```
datetime = 'The date and time is 11/2/01 16:12:01 from MET';
dtmatch = re.match(r'((\d+)/(\d+)/(\d+)) ((\d+):(\d+):(\d+))',
datetime)
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
date = dtmatch.group(1)
time = dtmatch.group(5)
(day, month, year) = dtmatch.group(2,3,4)
```

请注意，在上述代码中组是如何工作的，组序号实际上被指定为所看到的每个开始圆括号。在前面的例子中，第一个圆括号代表表达式的第 1 组，依次包含第 2 组、第 3 组和第 4 组。(请注意正则表达式的索引是从 1 开始的，而不是从 0 开始)。其结果就是既能抽取出整个日期字符串，又能抽取出日期字符串的独立元素。

表 10-12　有关给定 MatchObject 对象的方法

MatchObject 对象的方法	说　明
m.group([group, ...])	返回匹配的文本，是个元组。此文本是与给定 group 或由其索引数字定义的组匹配的文本。如果没给出组的名字，则返回所有匹配项
m.groups([default])	返回一个元组，该元组包含模式中与所有组匹配的文本。如果给出 default 参数，default 参数值就是与给定表达式不匹配的组的返回值。default 参数的默认取值为 None
m.groupdict([default])	返回一个字典，该字典包含匹配的所有子组。如果给出 default 参数，其值就是那些不匹配组的返回值。default 参数的默认取值为 None
m.start([group])	返回指定 group 的开始位置，或返回全部匹配的开始位置
m.end([group])	返回指定 group 的结束位置，或返回全部匹配的结束位置
m.span([group])	返回两元素元组，此元组等价于关于一给定组或一个完整匹配表达式的(m.start(group),m.end(group))列表
m.pos	传递给 match()或 search()函数的 pos 值
m.endpos	传递给 match()或 search()函数的 endpos 值
m.re	创建这个 MatchObject 对象的正则表达式对象
m.string	提供给 match()或 search()函数的字符串

6. 置换

sub()函数在一个字符串上执行置换。实际上，置换与典型的匹配工作方式完全一样，除了有个正则表达式搜索字符串之外，还有一个替换字符串。此替换字符串通常就是基本文本，但也能涉及搜索表达式中与任何组匹配的文本。

sub()函数的基本格式如下：

```
sub(pattern, replace, string [, count])
```

利用sub()函数，能使用如下所示的命令将“cat”置换为“slug”：

```
text = 'the cat sat on the mat'
text = re.sub(r'cat', 'slug', text)
```

请再次注意，在其上执行置换操作的文本或变量没有被相应修改，必须将此函数的结果

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

重新分配给原始变量以实现改变。

替代操作还能包含以\n 形式引用的组，其中，n 是组的序号。例如，将一个国际日期(yyyymmdd)转换为英国日期，所用命令如下所示：

```
date = '20010416';
date = re.sub(r'(\d{4})(\d{2})(\d{2})', '\3.\2.\1', date)
```

replace参数还接受函数，提供给此函数一个单MatchObject对象参数。例如，要用单字母等价符分析URL地址中使用的%xx格式的序列，就能使用如下命令：

```
value = re.sub(r'%([a-fA-F0-9][a-fA-F0-9])',
                lambda x: chr(eval('0x'+x.group(1))), value)
```

要获取置换发生的次数，请使用 subn()函数。subn()函数返回一个元组，此元组包含置换了的文本和置换次数，如下例所示：

```
text = 'the cat sat on the mat'
(text, subs) = re.subn(r'at', 'ow', text)
```

在这个例子中，subn()函数设置subs参数为3。

请注意，默认情况下，Python的所有置换函数对所有匹配的位置都替换。要限制修改的次数，使用可选参数项，即第四个参数count。例如，要将第一次查到的“at”字符串改变为“oelacanth”，使用如下命令：

```
text = 'the cat sat on the mat'
text = re.sub(r'at', 'oelacanth', text, 1)
```

上述命令产生“the coelacanth sat on the mat”这样的结果。

7. 使用编译的正则表达式

在使用正则表达式时，正则表达式被编译为内部语言，这种内部语言实际上用来执行匹配或置换操作。如果用同一个正则表达式进行许多次匹配或置换，而处理的却是不同的文本元素——也就是说，在处理同一文件中各个独立的字符行时——就在浪费，在重复编译一个始终不变的正则表达式。

通过对此正则表达式对象进行预编译就可避免这样的浪费。对正则表达式对象进行预编译利用 compile()函数就能实现。compile()函数把正则表达式编译为一个正则表达式对象。compile()函数的基本格式如下所示：

```
compile(str [, flags])
```

其中，str 参数是要使用的正则表达式；而 flags 参数，如表 10-11 所列，指定想在新对象上使用的修饰符。这个新对象具有与 re 模块中主要函数名字和功能都相同的方法。例如，可以重新编写以下 Python 代码段：

```
text = 'the cat sat on the mat'
text = re.sub(r'at', 'oelacanth', text, 1)
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

为

```
text = 'the cat sat on the mat'
cvanimal = compile(r'at')
text = cvanimal.sub('oelacanth', text)
```

表 10-13 列出了正则表达式对象支持的其他方法。

表 10-13　正则表达式对象支持的方法/属性

正则表达式对象支持的方法	说　　明
r.search(string [, pos [, endpos]])	等价于 search()函数。但此函数允许指定搜索的起点和终点
r.match(string [, pos [, endpos]])	等价于 match() 函数，但允许指定搜索的起点和终点
r.split(string [, max])	等价于 split() 函数
r.findall(string)	等价于 findall() 函数
r.sub(replace, string [, count])	等价于 sub() 函数
r.subn(replace, string [, count])	等价于 subn() 函数
r.flags	在对象创建时给出的标志
r.groupindex	将 r'(?Pid)'定义的符号组名字映射为组序号的字典
r.pattern	在创建对象时使用的模式

8. 转义字符串

有时，要利用变量元素来建立一个正则表达式。那难题就是正使用的变量可能含有可能要被解释为正则表达式元素的字符。例如，在一个正则表达式中使用“Martin (Brown)”，而在特别想对圆括号进行匹配时，因为已经知道圆括号是用来指示一个组的，所以就产生难题了。如果手工指定匹配，能利用反斜线字符(\)来转义那些圆括号。

如果信息在其他地方被终止，就没有办法在它被用于表达式当中之前修改文本。

然而，re 模块定义了 escape()函数。escape()函数转义一个字符串中所有这样的字符：将被辨认为正则表达式元素的字符，使得这些字符被解释为生字符，如下所示：

```
string = re.escape(expr)
```

10.3　时间

在使用机器的时间值时，必须对所有机器是怎样对待自身的时间数值有所了解。几乎对所有机器和操作系统而言，时间信息是以数字的格式存储的。此数字表示从新纪元计起的秒值——新纪元是用作参考点的一个过去时间点。在 Windows 和 Unix 操作系统下，新纪元是 1970 年 1 月 1 日 0 点 0 分 0 秒那个时刻。在 Mac OS (但不是 Mac OS X)系统上，则是 1904 年 1 月 1 日 0 点 0 分 0 秒。

例如，在编写本书时的一个时刻，时间值是 996853682.48452997，翻译过来的时间就是

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

2001年8月3日星期五下午4点48分2秒。能通过使用time.time() 函数来获得当前新纪元时间值，如下所示：

```
>>> import time
>>> time.time()
996853682.48452997
>>> time.ctime(996853682.48452997)
'Fri Aug 3 16:48:02 2001'
```

ctime()函数将新纪元时间值转换为一个字符串。详见本章后面有关如何格式化时间值内容的“时间格式化”部分。

time模块剩余部分提供了保存有关当前主机时间信息的函数和变量。表10-14列出了time模块定义的变量。

表10-14 time模块中定义的变量

变 量	说 明
accept2dyear	一个布尔数值；如果设定为真，时间函数除了接受4位数字表示的年份还接受两位数字表示的年份
altzone	在当前时区经济时(即夏令时间，Daylight Savings Time)(DST)启动的时候，所使用的时区
daylight	如果一个DST时区是在实行中，就设置为非零值
timezone	本地非DST时区
tzname	一个元组，此元组列包含本地时区的名字和在DST实施时所使用的时区

10.3.1 抽取时间值

能将一个新纪元时间值转化为独立的时间单位——年、月、日、小时、分钟、等等——利用gmtime()和localtime()函数。这两个函数都接受一个单一参数，即一个新纪元时间值，并返回一个元组，该元组包含表10-15列出的独立时间值。

这两个函数的惟一不同之处在于：localtime()函数返回相对于用户时区的时间值，而gmtime()函数返回值则对应于GMT时间值(GMT就是格林威治时间(Greenwich Mean Time)，另外还称作UTC，世界坐标时间(Universal Coordinated Time))。如果参数值默认，这两个函数都使用当前新纪元时间值。

例如，要抽取今天的日期，就请使用如下命令：

```
(year, month, day) = localtime()[0:3]
 print "%d/%d/%d" % day, month, year
```

就一般而言，在想显示一个时间值时，使用localtime()函数，但是在要比较或者存储一个稍后要检索的时间值时，请使用gmtime()函数。localtime()函数受本地时区影响，它在不同位置上返回的时间值不同，而最重要的是，在夏令时启动时(夏令时，在英国叫作BST，英国的

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

夏季时间)返回的时间值不同。如果存储两个用localtime()函数恢复的时间值，然后比较这两个值，由于时区和夏令时的不同，最终得到的结果将不合逻辑。

表 10-15　localtime 和 gmtime 函数返回的元组元素

元　素	内　容	值 范 围	说　明
0	年	0–9999	整个年份，即 2001
1	月	1–12	月份数字
2	日	1–31	本月的日期
3	小时	0–23	一天里的小时数字(24 小时格式)
4	分钟	0–59	小时里的分钟数
5	秒	0–59	分钟里的秒数
6	星期	0–6	对应于一星期的星期几(星期天=0)
7	天	1–366	一年里的哪一天
8	夏令时	－1, 0, 1	如果夏令时(DST)启动了，就为 1；如果夏令时没启动，就为 0；如果无信息可用，就为-1

10.3.2　时间格式化

尽管自己能抽取并格式化时间信息，但实际上 time 模块提供了许多实用函数，这些实用函数用于由未经处理的数值产生日期和时间字符串，而不需要任何手动干涉或手动格式化。两个基本的函数，asctime()和 ctime()函数，都返回一个标准字符串，此标准字符串使用当前本机定义的数字和日期格式。asctime()函数产生基于由 localtime()函数或 gmtime()函数返回的元组的字符串，如下所示：

```
print time.asctime(time.localtime(epoch))
```

ctime(epoch)函数仅仅只是time.asctime(time.localtime(epoch))函数的简写形式，其中epoch参数是以秒为单位的时间值。在一个英国操作系统上，上述命令产生如下结果：

```
'Fri Aug 3 17:30:52 2001'
```

对于更具结构化的输出，可使用 strftime()函数。此函数接受两个参数：第一个参数是个格式字符串，其格式与 Python 字符串格式化运算符(%)的风格大体相同，所不同的是格式化选项返回一个时间值。第二个参数应是个元组，此元组是由 localtime 或 gmtime 函数返回的元组。表 10-16 列出了所支持的格式化字符串。

表 10-16　time.strftime()函数使用的格式字符串

格式字符串	说　明
%a	本地简化的星期名称
%A	本地的完整星期名称
%b	本地简化的月份名称

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

格式字符串	说　　明
%B	本地的完整月份名称
%c	本地相应的日期表示和时间表示
%d	用十进制表示一个月内的一天(0–31)
%H	用十进制表示一天里的小时(0–23, 24 小时时钟)
%I	用十进制表示一天里的小时(01–12, 12 小时时钟)
%j	用十进制表示一年之中的一天(001–366)
%m	用十进制表示的月份(01–12)
%M	用十进制表示的分钟(00–59)
%p	本地 A.M.或 P.M.的等价符
%S	用十进制表示的秒(00–59)
%U	一年里的星期数(00–53, 星期天作为一个星期的第一天)
%w	用十进制数字表示的星期(0–6, 星期天作为一个星期的第一天)
%W	一年里的星期数(00–53, 星期一作为一个星期的第一天)
%x	本地相应的日期表示
%X	本地相应的时间表示
%y	用十进制表示的不带世纪前缀的年份(00–99)
%Y	用十进制表示的带世纪前缀的年份
%Z	当前时区的名称
%%	%字符

例如，要把日期写成日/月/年这种特定格式，请使用如下命令：

```
print time.strftime('%d/%m/%Y', time.localtime())
```

要把字符串写成与asctime()或ctime()函数生成字符串的格式一样，就使用%c格式，如下所示：

```
print time.strftime('%c', time.localtime())
```

10.3.3 创建新纪元时间值

能用 mktime()函数把单独的时间值转换回新纪元时间值，其中 mktime()函数接受一个元组，该元组值的顺序和范围与 localtime()和 gmtime()函数的返回值相同。

因此，能利用如下命令把一个新纪元时间值转换回其本身，如下所示：

```
current = time.time()
new = time.mktime(time.localtime(current))
if ( new != current):
    print "The time stasis has leaked!!"]
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

如果有个更复杂的时间字符串，并想抽取出其中信息，就必须使用strptime()函数。strptime()函数等价于strftime()函数，所不同的是互为反操作。strptime()函数分析给定的日期/时间字符串，返回一个localtime()或gmtime()样式的元组。例如，要把日/月/年格式的字符串转换为更易管理的内容，请见下面这个例子：

```
import time
datestring = '23/3/2001'
datetuple = time.strptime(datestring, '%d/%m/%Y')
```

秒参数的默认取值——要分析的日期／时间字符串的格式——是“%a %b %d %H:%M:%S %Y”。

10.3.4　时间值的比较

虽然在独立基础上进行时间值的比较是可能的，但这确实引发了一些问题。例如，如果想在月／年格式的两个组合体间找出具体差距时，就不得不先比较月份的数值再比较年份的数值来求得一个差值，如下所示：

```
montha,yeara = 11,1999
monthb,yearb = 04,2001
monthsdiff = ((yearb-yeara)*12)-montha+monthb
```

要找出日期的差别，就变得更复杂了。因为现在还必须计数每个月份内的天数(其原因是每个月的天数不是个常数)，并且还要为确定二月份具体天数而执行闰年的计算。想想要这样一直比下去，直到算出最后一秒之差，多复杂！

解决上述难题的方法是：将日期／时间格式的时间值转换为新纪元时间值，然后用一个新纪元时间值减去另一个值，接下来用 gmtime()函数来报告出具体细节。gmtime()函数报告的内容是个日期，但是因为它是基于提供给它的新纪元时间值来计算日期的，实际上对两个值的比较最终总结是年、月和日的差距。

例如，让我们来比较一下著名数字间的差距，找出 2001 年 8 月 1 日和 2001 年 8 月 31 日的天数差距。利用 mktime()函数,能把上述日期转换为新纪元时间值，比较这两个新纪元时间值，得到差值，如下所示：

```
import time

epocha = time.mktime((2001,8,1,0,0,0,0,0,0))
epochb = time.mktime((2001,8,31,0,0,0,0,0,0))
(year, month, day) = time.gmtime(epochb-epocha)[0:3]
print "Years: %d, Months: %d, days: %d" % (year,month,day)
```

预期的结果是 30 天。但如果运行上面的脚本，其结果是：

```
Years: 1970, Months: 1, days: 31
```

得到上面结果的原因是：上述脚本比较两个时间值，然后返回以新纪元时间值为参考的

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

差值，所知道的新纪元时间值为1970年1月1日。因为将一个为0的数值提供给gmtime()函数，它就返回元组(1970,1,1,0,0,0,0,0,0)，所以必须将纪元的基值删除来获得真正的差值，如下所示：

```
import time

epocha = time.mktime((2001,8,1,0,0,0,0,0,0))
epochb = time.mktime((2001,8,31,0,0,0,0,0,0))
(year, month, day) = time.gmtime(epochb-epocha)[0:3]
year-=1970
month-=1
day-=1
print "Years: %d, Months: %d, days: %d" % (year,month,day)
```

现在，如果执行这个脚本，就得到如下所示的结果：

```
Years: 0, Months: 0, days: 30
```

如果用不同于上述时间的两个值执行上述思想的脚本，比如说，1967年2月22号和今天2001年8月3号，将得到如下结果：

```
Years: 34, Months: 5, days: 11
```

10.3.5 暂停进程

time.sleep()函数暂停当前进程，暂停时间长度为指定的秒数。例如，要暂停一进程1.5秒钟，就用以下命令：

```
time.sleep(1.5)
```

请注意，机器不同使用sleep基本函数的粒度不同。尽管Python允许休眠小数点后面的数量长度的时间(提供给time.sleep函数的参数应是个浮点数)，但底层的操作系统可能仅支持休眠时间值的整数值那么长时间。进而，整个系统可能不精确，而实际的休眠时间可能比所请求的时间要长，最多超出一秒钟。

10.4 数据类型与运算符

除了已述不同数据类型的操作和处理的基本方法外，还有一大堆模块，这些模块既能用来标识特殊的对象类型，又能操作二进制结构，特殊的对象类型用来创建基于特殊的数据类型之上的结构。

10.4.1 类型验证

如果想标识特殊对象的类型，可使用type()内置函数，如下所示：

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
>>> import time
>>> type(time)
<type 'module'>
```

有些时候，总想检查一下对象类型。在许多实例当中，可以只提供一个空对象，如下例所示：

```
if (type(list) == type([])):
...
```

但是，有些类型是不能用这种方法轻松地生成。要避免此类问题，请使用types模块。types模块定义了所有不同类型的变量，这些变量允许通过名字来指定一个对象。例如，下例中reverse()函数就标识给定对象的类型并将其转换为相应的对象类型，如下所示：

```
def reverse(x):
    import types
    if (type(x) == types.ListType):
        copy = x
        copy.reverse()
        return copy
    if (type(x) == types.StringType):
        y = ''
        i = len(x)-1
        while(i >= 0):
            y += x[i]
            i -= 1
        return y
    if (type(x) == types.DictionaryType):
        y = {}
        for key in x.keys():
            y[x[key]] = key
        return y
    raise(TypeError)
```

现在就能用列表、字符串或字典来调用reverse()函数，得到给定数据类型的互补版本数据类型的返回，如下所示：

```
>>> reverse('hello')
'olleh'
>>> reverse([1,2,3])
[3, 2, 1]
>>> reverse({'a': 1,'b':2})
{2: 'b', 1: 'a'}
>>>
```

表10-17包含了对象类型的完全清单。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

表 10-17 types 模块定义的对象类型

变 量	别 名
BufferType	缓存对象，如 buffer()函数所创建的对象一样
BuiltInFunctionType	内置函数
BuiltinMethodType	内置方法
ClassType	类定义
CodeType	编译的字节码
ComplexType	复数
DictionaryType	DictType 的另一个可选名字
DictType	字典对象
EllipsisType	Ellipsis 对象(扩展的切片对象)
FileType	文件对象
FloatType	浮点类型对象
FrameType	执行框架对象
FunctionType	用户定义的函数
InstanceType	类对象实例
IntType	整型对象
LambdaType	FunctionType 的另一个可选名字
ListType	数组和列表
LongType	长(任意)整型
MethodType	约束(Bound)类方法
ModuleType	模块
NoneType	None 值
SliceType	Slice()函数返回的对象
StringType	字符串对象
TracebackType	一个异常的堆栈跟踪轨迹
TupleType	元组对象
TypeType	类型对象(如 type()函数返回对象那样)
UnboundMethodType	无约束类方法(class.method)
UnicodeType	Unicode 字符串
XRangeType	xrange()内置函数所创建的对象

10.4.2 运算符

operator模块提供访问内置运算符和解释器的特殊方法的函数，其中有关访问解释器的特殊方法要回头看第3章。表10-18列出了导出的函数、与它们等价的对象方法和等价的运算符。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

表 10-18　operator 模块的函数 / 方法和其等价运算符

函　数	对 象 方 法	等价运算符
add(a, b)	__add__(a, b)	a + b (数字的)
sub(a, b)	__sub__(a, b)	a - b
mul(a, b)	__mul__(a, b)	a * b (数字的)
div(a, b)	__div__(a, b)	a / b
mod(a, b)	__mod__(a, b)	a % b
neg(a)	__neg__(a)	- a
pos(a)	__pos__(a)	+a
abs(a)	__abs__(a)	a 的绝对值
inv(a)	__inv__(a)	a 的逆
invert(a)	__invert__(a)	
lshift(a, b)	__lshift__(a, b)	a << b
rshift(a, b)	__rshift__(a, b)	a >> b
and (a, b)	__and__(a, b)	a & b
or (a, b)	__or__(a, b)	a | b
xor(a, b)	__xor__(a, b)	a ^ b
not (a)	__not__(a)	Not a
truth(a)		如果 a 为真(true)就返回 1，否则返回 0
concat(a, b)	__concat__(a, b)	a + b (序列)
repeat(a, b)	__repeat__(a, b)	a * b (序列 a，数字 b)
contains(a, b)	__contains__(a, b)	b in a
countOf(a, b)		返回 b 在 a 内出现的次数
indexOf(a, b)		返回 b 在 a 内第一次出现的下标值
getitem(a, b)	__getitem__(a, b)	a[b]
setitem(a, b, c)	__setitem__(a, b, c)	a[b] = c
delitem(a, b)	__delitem__(a, b)	del a[b]
getslice(a, b, c)	__getslice__(a, b, c)	a[b:c]
setslice(a, b, c, v)	__setslice__(a, b, c, v)	a[b:c] = v
delslice(a, b, c)	__delslice__(a, b, c)	del a[b:c]

10.4.3　数组建造

从根本上而言，标准 Python 数组或列表类型是一列对其他对象的引用。在大多数情况下，这确实是所要的，因为很有可能需要在一典型列表中存储字符串、整数、和浮点数值。对于

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

复杂结构，还可以添加序列、元组和字典。然而，这样自由形式的数组和列表在存储诸如字符或数字这样的海量简单数据时却不能奏效。例如，在一个标准列表中，每一个 Integer 数据类型最少占据 12 个字节。假如说被 200 000 个单元相乘，仅对存储一个列表就要用掉 2.3MB 空间。

解决的方法是使用 array 模块。array 模块能让您创建一个列表，这个列表仅接受和存储单一类型的数据元素。例如，如果知道要存储浮点数值，就使用 array()函数来创建一个新的数组，array()函数所用参数为'f'，它表明要创建一个浮点类型的数组，如下所示：

```
myfparray = array.arry('f')
```

array()函数的第二个参数是个选项参数，它包含分配给新创建数组的初始数值。表10-19列出了所支持的其他数组类型。

表 10-19　有关创建固定类型数组的类型代码

数组类型代码	说　　明	等价的 C 语言类型
c	8 位字符型	Char
b	8 位整型	signed char
B	9 位无符号整型	Unsigned char
h	16 位整型	Short
H	16 位无符号整型	Unsigned short
i	整型	Int
I	无符号整型	unsigned int
l	长整型	Long
L	无符号长整型	unsigned long
f	双精度浮点型	Float
d	双精度浮点型	Double

数组一旦被创建，就能对这个列表操作并能像其他列表一样地使用。除此之外，这个新对象具有表10-20列出的方法和属性。

表 10-20　固定类型数组的方法 / 属性

方法 / 属性	说　　明
typecode	用来创建数组的类型代码字符
itemsize	每个数据元素的尺寸大小
append(x)	将 x 参数追加到数组上
buffer_info()	返回一个元组，该元组包含用来存储数组的内存地址和缓存长度
byteswap()	从高位到低位或相反的顺序交换数组中所有数据项的字节顺序。仅受整型数组支持
fromfile(f, n)	从 f 文件对象加载 n 个数据单元到当前数组中

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

（续表）

方法 / 属性	说　明
fromlist(list)	将 list 列表中的数据项追加到数组上。如果 list 列表有一个数据项与数组的类型不匹配，将引发 TypeError 异常
fromstring(s)	好像 s 是个二进制值的字符串附加字符串 s 的数据项
insert(i,x)	将 x 数据项插入到元素 i 处
reverse()	将数组的顺序反转
tofile(f)	将数组作为一个二进制流写入到文件对象 f 中
tolist()	将数组转换为一个普通数值列表
tostring()	将数组转换为一个二进制字符串

10.4.4　二进制结构

C 语言使用 struct 系统来构造由许多性质不同的基本类型构成一个单个实体。要读取这些二进制结构，就需要使用 struct 模块。struct 模块提供了两个主要函数：pack()函数和 unpack()函数。pack()函数将一系列值封装到一个二进制结构中去；而 unpack()函数则将一个二进制结构转换回一列值。

这两个函数都依赖于格式字符串，格式字符串定义所封装的结构的格式。格式字符串由一个字母组成的字符串构成，其中每个字母定义一个特殊数据类型。例如，“i”这个字母就表示整型(即 C 语言的 int 类型)。还能在每个字母前加上数字前缀来规定该值的重复个数。例如，要抽取 4 个整数，就使用格式字符串‘4i’。在定义了一个字符串的格式字符‘s’的情况下，其前缀数字就确定要抽取的字符串的最大长度。

例如，能读取下面的 C 语言结构体：

```
struct clockin {
    short id;
    char name[20];
    unsigned long timestamp;
};
```

读取这个结构体的封装格式应为‘h20sL’。要将数据封装到此结构体中，使用如下命令：

```
import struct, time
binary = struct.pack('h20sL',239,'Martin',time.time())
```

要将二进制字符串解封(拆包)回不同的变量，请使用如下命令：

```
(id, name, timestamp) = struct.unpack('h20sL', binary)
```

表10-21包含了所有受支持的格式字符的完整清单。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

表 10-21 用于封装 / 拆包 C 语言结构体的字符

格式字符	C 语言类型	Python 类型
x	pad byte	无数值(No value)
c	字符型(char)	长度为 1 的字符串
b	有符号字符型(signed char)	整型(Integer)
B	无符号字符型(unsigned char)	整型(Integer)
h	短整型(short)	整型(Integer)
H	无符号短整型(unsigned short)	整型(Integer)
i	整型(int)	整型(Integer)
I	无符号整型(unsigned int)	整型(Integer)
l	长整型(long)	整型(Integer)
L	无符号长整型(unsigned long)	整型(Integer)
f	单精度浮点型(float)	浮点型(Float)
d	双精度浮点型(double)	浮点型(Float)
s	char[]	字符串(String)
p	char[]	第一个字节为字符串长度的字符串
P	void *	整型(Integer)

还能加在格式字符串前作为前缀来指定二进制结构结果的字节顺序和对齐方式。表10-22所列字符指明所支持的前缀。请注意，这些前缀字符必须仅指定一次，并且必须作为格式字符串的第一个字符。如果不给定前缀，默认选择是使用当前平台所使用的本地字节顺序和标准对齐方式——对任何类型没有对齐要求。

表 10-22 所封装结构体的字节顺序 / 对齐前缀

格式前缀	字节顺序	尺寸和对齐方式
@	本机(Native)	本机(Native)
=	本机(Native)	标准(Standard)
<	低位在先(Little endian)	标准(Standard)
>	高位在先(Big endian)	标准(Standard)
!	网络(big endian)	标准(Standard)

10.5 Unicode 字符串

有些读者可能在其他地方无意中看见过 Unicode；如果你还没有见过 Unicode，那么很快就要见识到它了。Unicode 是 Windows 操作系统的一分子已有很长时间了。虽然 Windows 95 和 NT 系统已支持存在于其中的 Unicode，但是直到 Windows 98 才正式地引入 Unicode。Mac OS 对 Unicode 的支持也有许多年了。虽然在 Unix 下要得到对 Unicode 的支持当前还是个系

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

统范畴的论题，但大多数需要使用 Unicode 的软件自身实现对它的支持。

Unicode 解决了一个老大难问题，该问题与屏幕上的字符表示有关；与屏幕上的字符作为对象与外部文件的数据是怎样在物理介质上存储起来有关。大多数人知道的格式是 ASCII 码，ASCII 码列出主要拉丁字母、数字、语法标记和它们的大小写版本。一些 ASCII 扩展码也被广泛地接受，并且这些扩展码考虑到支持大多数南方欧洲语言的古老字符。实际字符是用数字表示的，利用的数值范围是 0~255，这使得能够把字母的等价数字存储到一个 8 位单字节中。

然而，在把使用 ASCII 码或者说拉丁字符集的系统迁移到实际上不使用拉丁字符集的环境时，会发生什么呢？比如说，在希腊，人们使用希腊字母表，而希腊字母不是 ASCII 码标准的一部分。就传统而言，程序员和设计者早已躲开此类问题，所用的方法是开发一个字符集来实现正常的拉丁字符到外国等价字符间的映射。在本质上，仍是使用一个单字节表示每个字符，但是因为软件使用希腊字符而不是拉丁字符，所以显示希腊字符。

当使用 ASCII 码的系统迁移到诸如汉语和日语这样更复杂的图形语言时，这种方法就不能再次奏效了。传统汉语书写系统有 30 000 左右个汉字——不能直接由限制为最多 256 字符的单字节可表示。对于这些语言，需要使用多字节字符，多字节允许表示大到 0~(2^{32}–1)范围的字母表中的一个字符（或者是对于 64 位计算机的 0~(2^{64} –1)范围）。

在这种情况下，Unicode 就非常适合。Unicode 的标准被设计为允许在全世界范围内交换文本信息的标准集合。由于 Unicode 标准包括有关字符集和所交换数据的多字节格式的信息，所以能保证所读信息使用正确的格式和语言。

Python 2.0 引入了 Python　Unicode 的支持。除允许引入 Unicode 和原始(raw)Unicode 字符串的基本能力之外，现在 Python 还包括编码和解码 Unicode 及不同编码格式间翻译 Unicode 字符的机制。

此外，大多数核心模块也应允Unicode，因此能像对正常字符串一样在Unicode对象上执行正则表达式。还能使用Unicode字符串作为正则表达式自身的识别和搜索项。

10.5.1　创建 Unicode 字符串

Python 并非天生就支持 Unicode 字符串——Python 中使用的所有字符串不自动使用 Unicode 格式——而是 Python 支持一个新的数据类型：Unicode 字符串。通过给字符串加前缀字母 u 就能创建一个新的 Unicode 对象，与引入原始字符串的方法相同，如下例所示：

```
>>> u'Hello World'
u'Hello World'
>>> u'Hello\u0020World'
u'Hello World'
```

要将特殊字符(非本机)包含到一个Unicode对象当中，请使用Unicode转义字符，即\u。这样就引入对应于给定十六进制数值的字符。在上例中，用十六进制数值20——空格符——相应的解释如上例中所见。这儿还有一个例子，这次是把一个带有一划或一个斜线的小写o(也就是说，ø)插入到一个Unicode字符串中，如下所示：

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
message = u'J\u00f8rgensen'
```

其他所有字符都是根据Latin-1编码进行转换。有关把一个Unicode 字符串转换为另一种格式的信息本章后面参见“转换Unicode”部分。请注意，在支持Unicode 的平台和系统上，Unicode字符串转换程序还能把非ASCII码字符转换成为Unicode字符。例如，在Mac上，引入古老的和其他外国语字符是操作系统的内置部分，因此能直接将这些字符插入到一个带字母u前缀的字符串中，如下所示：

```
>>> u'øåé'
u'\xbf\x8c\x8e'
```

要引入一个原始Unicode字符串——不嵌入任何Unicode转换的一个Unicode字符串——使用创建Unicode对象的ur前缀，请见下例所示：

```
>>> ur'Rikke\u0020J\u00f8rgensen'
u'Rikke J\u00f8rgensen'
```

原始Unicode字符串的工作方式与它们的原始字符串一样——原始字符串的存在使得能够引入字符串，引入的字符串可以包含不想被翻译或不想被解释的信息。如同原始字符串一样，在利用正则表达式内的Unicode字符串时，这尤其有用。在这些实例中，仅当小写u字母之前只有一个反斜线时，Unicode转义序列才被解释。在下例中能更清楚地看到这一点：

```
>>> ur'\\u0020'
u'\\\\u0020'
>>> ur'\u0020'
u' '
>>> u'\\\u0020'
u'\\\\ '
```

10.5.2 转换 Unicode

在大多数基本层上，Unicode 能和正常的 Python 字符串序列一起混合和匹配，但是其结果同下例一样，总是另一个 Unicode 对象：

```
>>> 'Hello ' + u'Miss J\u00f8rgensen'
u'Hello Miss J\xf8rgensen'
```

要将一个 Unicode 对象转换回正常的 ASCII 码(7 位)字符串，请使用 str()内置函数，如下所示：

```
>>> str(u'Hello World')
'Hello World'
```

总之，要特别注意，Unicode 字符串与 ASCII 码是不兼容的。如果试着转换一个包含非 ASCII 码字符的字符串，str()函数将引发错误，如下例所示：

```
>>> greet=u'Miss J\u00f8rgensen'
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
>>> str(greet)
Traceback (most recent call last):
    File "<stdin>", line 1, in ?
UnicodeError: ASCII encoding error: ordinal not in range(128)
```

请注意，这个原则不管怎样访问字符串，即使是从一个 Unicode 字符串中独立地抽取字符时，也一样奏效。例如，下面的代码仍然会产生一个错误：

```
for char in u'Miss J\u00f8rgensen':
    print char,
```

注意：

在编码和／或解码字符串过程中产生的任何错误都引发一个 UnicodeError 异常，该异常能用捕获其他任何异常同样的方法捕获。UnicodeError 异常提供错误消息，此消息就是其惟一的参数值。

10.5.3　编码为 Unicode 格式

ASCII 码不是最有用的格式。能使用 encode()函数把一个 Unicode 字符串转换为许多不同的格式。encode()函数改变用于将一个 Unicode 对象直接表示为诸如 Latin-1 或 UTF-8 的其他字符集的编码。事实上，encode()方法是在一个 Unicode 对象上使用 str()内置函数时调用的方法，encode()方法以 ASCII 码形式提供编码类型。

Latin-1 编码，支持 8 位 ASCII 码表中提供的前面 256 个字符，如前面的例子一样，能用来表示大多数字符串，如下所示：

```
>>> greet = u'Rikke J\u00f8rgensen'
>>> greet.encode('latin-1')
'Rikke Jørgensen'
```

这些字符在屏幕上的再现当然要依赖：您所具有的遵守Unicode标准的字体、应用程序和操作系统！

这儿有个典型的例子，就是Mac OS系统，此系统不直接支持Unicode标准。要想在编写一个标准的Mac文档或屏幕上显示时获得同样的效果，就需要使用Mac Roman编码。

encode()方法还能被用来将Unicode对象编码为本地的一种Unicode编码格式，如UTF-8或UTF-16格式。例如要编码上例的样本字符串，请使用如下代码：

```
>>> greet.encode('utf-8')
'Rikke J\xc3\xb8rgensen'
>>> greet.encode('utf-16')
'\xfe\xff\x00R\x00i\x00k\x00k\x00e\x00
\x00J\x00\x18\x00r\x00g\x00e\x00n\x00s\x00e\x00n'
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

10.5.4 解码为 Unicode 格式

要将一个编了码的字符串转换回其 Unicode 格式(就是说，求 encode()函数的反)，请使用 unicode()内置函数。这个函数是随 Python 2.0 引入的。此函数接受两个主参数：一个参数是要解码的字节流，另一个参数是要将字节流解码成为的格式。例如，能像下例代码一样将 Rikke Jørgensen 字符串从 Mac Roman 格式解码为一个 Unicode 字符串：

```
>>> unicode('Jørgensen','mac-roman')
u'J\xf8rgensen'
```

返回的类型是 Unicode 对象。一定要认识到：unicode() 函数利用给定格式将一个字符串对象解码为其 Unicode 版本——使用的格式要是不正确的话，结果就是不正确的 Unicode 对象。例如，利用 Latin-1 编码格式将源自于 Mac 文档的 Jørgensen 字符串进行解码，则将得到一个不同的 Unicode 字符串，如下所示：

```
>>> unicode('Jørgensen','latin-1')
u'J\xbfrgensen'
```

还能利用 unicode()函数从一个编码格式直接转化为另一种编码格式。例如，UTF-8 编码流 Jørgensen，能使用如下命令直接转化为 UTF-16 格式：

```
>>> unicode('Miss J\xc3\xb8rgensen','utf-16')
```

unicode() 函数还接受第三个参数，此参数告知解释器如何处理在编码过程中产生的错误。例如，使用‘strict’这样的参数值就强制编码机制执行严格的编码，不能被编码的字符将引发 ValueError 异常，如下所示：

```
>>> unicode('Jørgensen','utf-8','strict')
Traceback (most recent call last):
    File "<stdin>", line 1, in ?
UnicodeError: UTF-8 decoding error: unexpected code byte
```

错误字符串被编码解码器(codec)用来翻译 Unicode 字符以确定如何编码及应如何处理错误。实际上，错误字符串及其影响依赖于所使用的编码解码器，但是也有一些翻译系统支持的标准字符串。

已经见过‘strict’选项；使用‘ignore’则允许翻译过程继续，把编了码的字符串中的所有特殊字符都删除，如下例一样：

```
>>> unicode('Jørgensen','utf-8','ignore')
u'Jrgensen'
```

要用编码解码器认为可能合适的字符替换一个不认识的字符，请使用一个错误字符串‘replace’。Python 使用编码解码器所定义的正式替换字符\uFFFD。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

10.5.5　编写自己的编码解码器

unicode()函数和 encode()方法使用 codecs 模块，此模块是标准函数库的一部分。codecs 模块提供不同格式间转换所需的基本类，但实际上，是 Python 标准库的 encodings 目录下的一组不同模块在工作。例如，当指明转换为 Mac Roman 格式，那么就是 encodings 目录下的 mac_roman 模块做实际工作。

能通过创建一个新模块来编写自己的编码解码器。这就需要导入 codecs 模块，接着要定义一个 Codec 类，Codec 类应是从编码解码器继承而来的。Codec 类应当包含两个方法：encode()和 decode()方法。实现这两个方法的最简单方法是使用 codecs 模块的 charmap_encode()函数和 charmap_decode()函数。

encode() 和 decode()方法两者都接受一个字符映射——字典，此字典映射去编码或去解码或者来解码的字符。例如，从 mac_roman.py 模块中抽取出来的内容，如下所示：

```
{
0x0080: 0x00c4, # LATIN CAPITAL LETTER A WITH DIAERESIS
0x0081: 0x00c5, # LATIN CAPITAL LETTER A WITH RING ABOVE
0x0082: 0x00c7, # LATIN CAPITAL LETTER C WITH CEDILLA
0x0083: 0x00c9, # LATIN CAPITAL LETTER E WITH ACUTE
0x0084: 0x00d1, # LATIN CAPITAL LETTER N WITH TILDE
0x0085: 0x00d6, # LATIN CAPITAL LETTER O WITH DIAERESIS
0x0086: 0x00dc, # LATIN CAPITAL LETTER U WITH DIAERESIS
0x0087: 0x00e1, # LATIN SMALL LETTER A WITH ACUTE
...
}
```

如果更新一个现存字典，请使用 codecs 模块的 make_identity_dict()函数。此函数根据给出的范围创建一个基础字典。例如，要匹配标准的 256 字符 8 位 ASCII 码映射表，可使用如下命令：

```
decoding_map = codecs.make_identity_dict(range(256))
```

接下来能使用update()方法合并更新过的字典映射表，如下所示：

```
decoding_map.update({
0x0080: 0x00c4, # LATIN CAPITAL LETTER A WITH DIAERESIS
0x0081: 0x00c5, # LATIN CAPITAL LETTER A WITH RING ABOVE
...
})
```

请记住，需要两个映射表：一个用于编码，而另一个用于解码。假定两种翻译互为反操作(就是说，传递给一个字符串的编码/解码应该返回初始字符串)，能利用以下命令创建一个反向映射：

```
encoding_map = {}
for k,v in decoding_map.items():
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
        encoding_map[v] = k
```

再回过头来看看encode()和decode()方法，利用刚创建的映射，定义那些方法，就像下面的代码一样：

```
class Codec(codecs.Codec):
    def encode(self,input,errors='strict'):
        return codecs.charmap_encode(input,
                                     errors,
                                     encoding_map)
    def decode(self,input,errors='strict'):
        return codecs.charmap_decode(input,
                                     errors,
                                     decoding_map)
```

您的编码解码器还需要定义 StreamWriter 和 StreamReader 类。这些类被 codecs 模块使用读写具体的数据流类型并将数据流类型转换为合适的字符格式。对于简单的 Unicode 转换，可能不需要这个，因此能轻描淡写地加以定义，如下所示：

```
class StreamWriter(Codec,codecs.StreamWriter):
    pass
class StreamReader(Codec,codecs.StreamReader):
    pass
```

创建自己编码解码器的最后一步是用 codecs 模块寄存您的代码，通过定义 getregentry()函数来实现这一点。getregentry()函数返回一个具有 4 个元素的元组，此表列包含所定义类的 encode()方法和 decode()方法及 StreamReader 和 StreamWriter 类。在您的事例中，这就产生如下的定义：

```
def getregentry():
    return (Codec().encode,
            Codec().decode,
            StreamReader,
            StreamWriter)
```

在创建完自己的编码解码器之后，把此模块放到 encodings 目录下。编码解码器已准备好，可以使用了。

这儿有个完整的编码解码器示例，该例执行一个相对用处不大的操作，将字符 a 转换为字符 e 和相反转换，其代码如下所示：

```
import codecs

# Create our Codec class

class Codec(codecs.Codec):
    def encode(self,input,errors='strict'):
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
        return codecs.charmap_encode(input,
                                     errors,
                                     encoding_map)
    def decode(self,input,errors='strict'):
        return codecs.charmap_decode(input,
                                     errors,
                                     decoding_map)

class StreamWriter(Codec,codecs.StreamWriter):
    pass

class StreamReader(Codec,codecs.StreamReader):
    pass

# Register ourselves with the codec module:

def getregentry():
    return (Codec().encode,
            Codec().decode,
            StreamReader,
            StreamWriter)

# Create our decode and encoding maps

decoding_map = codecs.make_identity_dict(range(256))
decoding_map.update({
        0x0041: 0x0045,
        0x0061: 0x0065,
        0x0045: 0x0041,
        0x0065: 0x0061,
})

encoding_map = {}
for k,v in decoding_map.items():
    encoding_map[v] = k
```

用'mcb'来测试一下这个编码解码器。启动Python，其结果如下所示：

```
>>> unicode('ae','mcb')
u'ea'
>>> u'ae'.encode('mcb')
'ea'
```

转换字符数字

ord()内置函数返回表示一个特殊字符的数字。此函数识别Unicode，因此，能使用如下命

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

令获得一个字符的Unicode数字：

```
>>> ord('ø')
248
```

要将数字转换回Unicode字符，无论如何，需要使用unichr()函数，而不是chr()函数，如下所示：

```
>>> unichr(248)
u'\xbf'
```

此函数返回一个单字符的Unicode对象。

10.5.6 访问 Unicode 数据库

unicodedata 模块提供对 Unicode 数据库的一个直接接口，Unicode 数据库就如 Unicode 国际协议发行的数据文件所定义的那样。

要检查 unicodedata 模块描述的 Unicode 字符，请使用 lookup()函数。例如，这就是如何确定希腊字母 pi 的 Unicode 字符的例子，请看下面的代码：

```
>>> import unicodedata
>>> unicodedata.lookup('Greek capital letter pi')
u'\u03a0'
```

请注意，在字符的名字中，大小写不是重要的，但是名字——不能略去您认为是可选的单词，并且单词间的说明性空格字符是需要的。如果不能查到单词，将引发KeyError异常。

要获得一个具体Unicode字符的Unicode名字，请使用name()函数，如下所示：

```
>>> unicodedata.name(u'\u03a0')
'GREEK CAPITAL LETTER PI'
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第11章 文 件 处 理

不能总是依赖应用程序中的信息。有时需要从文件里把信息读出来或者把信息写入到文件里。而其他情况，可能需要打开一个文件，能对相应位置的现存文件进行更新。编辑文件的内容常常只是问题的一小部分。有时候，不知道文件的名称，而需要给用户提供一个文件名称的列表；或者有时，想处理一列与特定名称或扩展名相匹配的文件。围绕上面所述，有待解决的是如下操作：产生文件和目录、删除文件和目录、获得文件和目录、修改文件和目录、设置文件和目录的所有权及附带设置。

本章将讲述上面涉及到的所有操作。

11.1 文件处理

打开文件的基本技术已经在第 3 章和第 8 章中讨论过了。利用 open()内置函数能打开文件。open()内置函数至少接受一个参数，此参数就是要打开文件的名称。open()函数的基本格式如下所示：

```
file = open(filename [, mode [, bufsize]])
```

默认情况下，文件以只读模式打开。但是第二个选项参数能用来指定打开文件的模式，文件打开的模式可以是从基本的读到用于更新的读，也可以是二进制模式。表 11-1 列出了 open()函数所支持的文件打开模式。

表 11-1 open()函数的文件打开模式

模 式	含 义
r	打开用于读
w	打开用于写
a	打开用于追加(在打开期间，文件位置自动移到文件末尾处)
r+	打开用于更新(读和写)
w+	截断(或清空)文件，然后打开文件用于读和写
a+	打开文件用于读和写,并自动改变当前文件位置为文件的结尾处
b	当附加于任何模式选项时，以二进制模式而不是文本模式，打开文件(这种模式仅对 Windows、DOS 和其他一些操作系统有效；而 Unix、MacOS 和 BeOS 则不管此选项为何值，而是以二进制模式对待所有的文件)

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

下一个选项参数，即第三个参数，定义文件是否使用缓存技术，以及如果使用缓存技术，缓存的尺寸大小又为多少。表11-2列出了其有效设置，默认设置是<0。

表11-2 open()函数支持的bufsize

bufsize值	说明
0	禁用缓存
1	行缓存
>1	使用大小近似为bufsize字符长度的缓存
<0	使用系统默认(对于tty终端设备而言就是行缓存；对其他所有文件而言就是全缓存)

此函数返回一个新的文件对象。通过这个对象能从文件里读出信息并把信息写入到文件中去。表11-3列出了文件对象所支持的方法。本章后面部分将陆续讲解每个方法的具体细节。

表11-3 文件对象的方法

方法	说明
f.close	关闭文件
f.fileno	返回文件描述符的整数值
f.flush	刷新输出缓存
f.isatty	如果文件是个交互终端，就返回1
f.read([count])	从文件中读出count个字节
f.readline	读出一个单行，以字符串的形式将其返回
f.readlines	读出所有行，以字符串列表的形式将其返回
f.seek(offset [, where])	把文件指针移动到相对于where的offset处
f.tell	获得当前文件指针
f.truncate([size])	截断文件，使文件的大小为size
f.write(string)	把string字符串写入到文件中
f.writelines(list)	把list中的字符行(一列字符串)写入到文件中去。list中的信息是不经任何处理写到文件里的，其中的每个元素都不做任何处理。writelines()方法不会含添加在输出当中的换行字符及回车字符

11.1.1 读文件

从文件往外读信息有两种方式：一个字节一个字节地读，或一行一行地读。一个字节一个字节地读这种方式在读二进制数据的时候或在不想一行一行地处理信息的时候，非常有用。一行一行地读这种方式最好用在想一行一行地处理信息的时候——也就是说，在处理日志文件的时候。一行一行地读方式又分两种不同模式：一种模式是从文件中读出一个单行数据；另一种模式是一次把文件中所有行全部读出。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

1. 逐行地读

readline()方法从文件中读出一个单行数据，如下所示：

```
line = file.readline()
```

不同的平台使用不同的行终止字符。在读文件时，Python 能识别出所在平台的正确的行终止字符。然而，要谨记：Python 不识别用于当前平台之外的其他平台的行终止字符。例如，在 Unix 系统上，能一行一行地读出 Unix 文件当中的每一行，但是在 Mac 系统上，同样的一个 Unix 文件在 Mac 版本的 Python 中就呈现为很长的一整行数据。因为 Windows 系统使用两个字符作为行终止符，所以 Windows 文件能在 Mac 系统及 Unix 系统下读成一个个独立的数据行，但是在每行的开头或结尾处增加了一个伪字符，详见第 23 章。

readline 方法返回的数据行中包括尾随的行终止字符。因此，在大多数情况下，需要使用 string.rstrip()方法把这个新行删掉。例如，使用如下所示的循环把一个文件的内容显示在屏幕上具有一定的诱惑力：

```
    line = myfile.readline()
if line:
        print line
    else:
        break
```

上述代码的问题是，显示输出包含两个行终止字符，因此在输出中有许多不应有的空行。取而代之的是，把上述代码中的 print 命令行更改为如下所示的命令：

```
print line,
```

或者使用 sys.stdout.write()以确保行终止字符不被自动地追加到输出中。

2. 获得所有行

能在活动文件对象上利用 readlines()方法读出文件的所有行。例如，利用如下所示的命令能把整个文件读到一个列表中：

```
lines = myfile.lines()
```

要把一个文件的全部内容读到一个单字符串对象里，请使用默认参数的 read()方法，请见下例：

```
myfiledata = myfile.read()
```

3. 逐个字节地读

还能使用 read()方法从文件中读出具体字节数目的数据。例如，要从一个文件中读出一个 512 字节的记录，就使用如下所示的命令：

```
record = file.read(512)
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

4. 文件结束

Python 不支持一个打开文件对象的“文件结束”状态这样的概念。尽管确实存在 EOFError 异常，但实际上此异常是被 input()和 raw_input()内置函数使用，在键盘输入一个被证明为文件结束字符的时候引发，以确认文件的结束。

而 read()和 readline()方法在读到文件结束时返回一个空字符串。因此，要结束文件的处理，就需要检查从文件里读出的信息，进而相应地中断循环，代码如下所示：

```
while 1:
line = myfile.readline()
    if !line: break
    print line,
myfile.close()
```

其中，if 语句在查到空行时，就中断循环。

另一个选择是，使用一个 for 循环语句和 readlines()方法一步一步地处理每个独立行，如下所示：

```
for line in myfile.readlines():
print line,
myfile.close()
```

此脚本在调用 readlines()方法返回的行列表到头时，自动终止。

5. 文件处理范例

处理 Rython 文件比读出行或数据这些基本能力要求的要多得多。大多数时候，需要能够识别出所读行或数据不同的部分，事实上，在第 10 章中已经给出了许多这类范例。在一行一行地处理数据时，显而易见的解决方法是使用 string.split()把各个元素抽取出来或是使用 re 模块提供的正则表达式系统把所要元素识别出来。

如果读出的是二进制数据，请使用 struct 模块来抽取二进制信息。另一个方法是，利用 pickle 模块和第 12 章论述的类似模块提供的服务来存储和加载文件中的永久对象。

举个例子来演示一下 Python 能对文件做什么。这儿是个简单的 Python 脚本，此脚本使用 string.split()通过访问标准 Web 日志对主机和统一资源定位符(URL)进行计数，如下所示：

```
import sys

def cmpval(tuple1, tuple2):
    return cmp(tuple2[1],tuple1[1])

hostaccess = {}
urlaccess = {}

if len(sys.argv) < 2:
    print "Usage:",sys.argv[0],"logfile"
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
        sys.exit(1)

    try:
        file = open(sys.argv[1])
    except:
        print "Whoa!","Couldn't open the file",sys.argv[1]
        sys.exit(1)

    while 1:
        line = file.readline()
        if line:
            splitline = string.split(line)
            if len(splitline) < 10:
                print splitline
                continue
            (host,ident,user,time,offset,req,
            loc,httpver,success,bytes) = splitline
            try:
                urlaccess[loc] = urlaccess[loc] + 1
            except:
                urlaccess[loc] = 1
        else:
            break

    hosts = hostaccess.items()
    hosts.sort(lambda f, s: cmp(s[1], f[1]))

    for host, count in hosts:
        print host, ": ", count

    urls = urlaccess.items()
    urls.sort(cmpval)

    for url, count in urls:
        print url, ": ", count
```

11.1.2　写入文件

往一个文件里写入信息通常就是调用 write()或 writelines()方法的一个示例。

1. 使用 write()或 writelines()

write()和 writelines()方法是 Python 中往文件里写入信息的最显著方法。这两种方法都能用于写入二进制数据，而不像 print 那样，它们不能自动往写入文件的每个字符中添加新行。write()方法写入一个单字符串，不要只看方法的名称，实际上，writelines()方法却是往文件里写入一

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

个字符串列表。例如，要把一个字符串写入到已打开文件里，请使用如下所示的语句：

```
file.write('Some text')
```

能写入一个字符串列表，所用语句如下所示：

```
file.writelines(lines)
```

2. 使用 print

Python 2.0 中最新增加的功能是能够直接往已打开文件里写入数据，而无需使用 write()或 writelines()方法。代替 write()或 writelines()方法的是个特殊运算符，此运算符与您想写入的文件对象名组合在一起使用，放置信息在关键字 print 和要写的字符串之间，如下所示：

```
print >>file 'Some kind of error occurred'
```

因为使用了 print 命令，所以换行字符被自动追加于写入字符串的末尾处，因此在使用 print 代替 write()时，一定要谨慎——在 print 命令末尾处附加一个逗号以使 Python 停止行终止序列的追加。还要当心的是，在把列表或者字典提供给 print 命令的时候，因为 print 是把列表、元组或字典的字符串表示写入到文件里，而不效仿 writelines()方法。

11.1.3　改变位置

所有文件对象都持有一个关于其在文件内位置的记录。这个位置是以文件中读出的或者往文件里写入的字节数为基础。位置 0，就意味着往文件的开始位置处写入，即位于文件的第一个字节之前；位置 1 定义在文件的第一个字节上；位置 2 定义在文件的第二个字节上；而位置 512 则定义在文件的第 512 个字节上。如果往文件里添加信息，信息就被添加于当前位置之后的位置上。所以，在位置 1 的时候，信息实际上是从第 2 字节位置上开始写到文件里。请注意，当然，不管信息写到文件的哪个位置上，都是重写存在的数据，而不是插入其中。

利用在已打开文件对象上使用 tell()方法能确定当前位置。要改变位置——当然是读或写文件的当前位置之外——使用 seek()方法。seek()方法只有一个单一参数，此参数以字节为单位指定在文件内要移动到的位置。例如，要移动到文件的开始位置上，请使用如下命令：

```
file.seek(0)
```

要移动到文件的结尾处是比较困难的，因为为了移动到文件最后一个字节的绝对位置上，所以需要知道文件的长度。总之，是有方法解决的。seek()方法的第二个参数，是个选项参数，它定义从文件的哪个位置开始移动。其默认取值是 0——假定所有位置都是相对于文件的开始位置。

如果 seek()方法的第二个参数值被设置为 1，其第一个参数的给定值就作为相对于当前位置的附加位移。因此，如果文件对象指针正处在第 512 字节上，调用了如下命令：

```
file.seek(512,1)
```

则文件对象指针自动移动到第 1024 (512+512)字节上。如果 seek()方法的第二个参数值被设

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

置为 2，其第一个参数就作为相对于文件末端的位移量。因此，能使用如下所示的语句直接把文件对象指针移动到文件的最后字节上：

```
file.seek(0,2)
```

还能利用如下所示的语句把指针往回移动 512 个字节：

```
file.seek(-512,2)
```

表 11-4 概括了这些取值。

表 11-4　seek 定位取值

seek 取值	说　明
0	相对于文件的开始处(即绝对的)。这是默认取值
1	相对于当前位置
2	相对于文件的结尾

11.2　控制文件 I/O

一旦打开了一个文件，可能就想控制文件访问和共享的方式。在 Unix 系统下，利用 fcntl 模块能达到这个目的，此模块提供了到基础 fcntl.h 和 ioctl.h 头文件的接口和到 C 语言接口函数库的一个接口。

11.2.1　文件控制

要使用文件控制，第一步是获得有关要控制的文件对象的文件描述符的数字。fileno 方法就是用来实现这个功能的。

下一步，利用 fcntl 模块的 fcntl()函数设置或获取有关文件的配置信息。fcntl()函数的基本格式如下所示：

```
fcntl(fd, command [, args])
```

其中：fd 参数是 fileno()方法返回的文件描述符数字；command 参数是个常量，它指定要发送给文件控制系统的命令。这个命令或是设置或是获取文件的状态和共享信息。表 11-5 包含了有效命令的清单，而表 11-6 则包含了 F_SETFL 命令所支持的选项的清单。请注意，FCNTL 模块导出这两个表的所有常量。

表 11-5　fcntl 命令

命　令	说　明
F_DUPFD	复制一个文件描述符。如果给定 args 的值，它就用作文件描述符的最小可能数值，此数用于复制文件描述符。实际上，此值依赖于打开文件描述符。返回值是新建文件描述符的数值

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

命　令	说　明
F_SETFD	设置 close-on-exec 标志。如果设置 close-on-exec 标志为真(args 值为 1)，在 exec*()调用执行时，关闭文件描述符；如果设置 close-on-exec 标志为假(args 值为 0)，文件描述符保持打开状态，并被调用复制(默认情况下)
F_GETFD	返回 close-on-exec 标志的当前值
F_SETFL	设置文件状态标志为 args 的值。这个文件状态标志应当是表 11-6 定义的一个或多个常量的逻辑位或。一定要保证使用对应于所用系统类型的正确标志
F_GETFL	获得文件状态标志
F_GETOWN	获取接收 SIGIO 信号的进程 ID 或进程组 ID(仅适用于 BSD)
F_SETOWN	设置接收 SIGIO 信号的进程 ID(仅适用于 BSD)
F_GETLK	获取文件锁结构。不为所有平台支持
F_SETLK	锁住一个文件。如果文件已锁住，此命令就返回-1。不为所有平台支持
F_SETLKW	锁住一个文件，但休眠当前进程的执行直到此锁能被获取到为止。不为所有平台支持

表 11-6　F_SETFL fcntl 命令的状态 Flag 常量

System V	BSD	说　明
O_NDELAY	FNDELAY	不锁住输入／输出——在对一个文件进行读和写操作的时候，执行不停止。对大多数平台而言，这不是必须的
O_APPEND	FAPPEND	追加模式
O_SYNC	FASYNC	同步输入／输出——不使用缓存技术。因此，写文件操作自动地把信息写入到磁盘上。在 BSD 系统下，当输入／输出可行的时候，这导致把一个 SIGIO 信号提交给进程组

11.2.2　I/O 控制

ioctl()函数等价于 fcntl()函数，不同之处在于：ioctl()函数提供了关于 ioctl 子系统的一个接口，ioctl 子系统控制所有有效文件描述符的输入／输出。请检查一下所在系统的文档，看一看所支持的配置选项都是什么。

11.2.3　文件锁定

在处理文件时，特别是在往文件里写入的时候，可能要保证当前进程是惟一具有往此文件里写入数据能力的进程。尽管有许多可能的解决方法，但处理锁定进程的最好办法是使用操作系统的文件锁定机制。有两个函数是关于锁定一个文件的，如下所示：

- flock()　提供一个锁定整个文件的简单机制
- lockf()　允许锁定文件的某个特定部分

这两个函数支持同样的操作。这些操作列于表 11-7 中，并且得到 fcntl 模块中所定义的许

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

多常量的支持。请注意，对 fcntl 选项的支持程度完全依赖于操作系统。

flock()函数的定义如下所示：

```
flock(fd, op)
```

其中：fd 参数是个文件描述符数，而 op 参数是表 11-7 中所描述的文件锁定选项的逻辑位或。fd 函数锁定整个文件直到对其解锁或进程终止为止。

表 11-7　文件锁定选项

操　作	说　明
LOCK_EX	排斥锁定。甚至，其他进程不能获得一个关于此文件的共享锁
LOCK_NB	不能休眠正被锁定的进程。此操作导致函数立即返回，返回锁的状态。如果不指定此操作，进程就休眠(或等待)一直到特定的锁能被成功地获取到为止
LOCK_SH	获取一个共享锁。这允许仅仅是您的进程能读写一个文件，而其他进程只能读此文件
LOCK_UN	删除所有锁

lockf()函数的定义如下所示：

```
lockf(fd, op [, len [, start [, whence]]])
```

其中：fd 和 op 参数如上述定义；而 len、start 和 whence 参数定义了锁定的长度和期限。如果默认 len、start 和 whence 这几个参数，就锁定整个文件；如果给定 len 参数值，lockf()函数就只锁定文件的开头 len 个字节；start 和 whence 这两个参数的作用与 seek 函数中的对应参数一样，指定锁定的开始位置和参考点。例如，能用如下所示的命令锁定文件的开头 1 024 个字节：

```
fnctl.lockf(myfd, FCNTL.LOCK_EX, 1024)
```

或者用如下所示命令锁定文件的最后 1 024 个字节：

```
fnctl.lockf(myfd, FCNTL.LOCK_EX, 1024, -1024, 2)
```

11.3　获取文件列表

常常是想处理一大堆文件，而不是只单单处理一个文件。例如，假设想处理当前目录下的所有.html 文件。解决问题的方法是利用 shell—— 在 Unix 下，这就是利用 shell(sh、ksh、bash、等等)，实际上是把诸如*.html 这样的文件填充到一个文件列表里，接下来把这个文件列表直接提供给应用程序，并且此文件列表在 Python 中可用，就是 sys.argv。

如果想得到一个文件列表，而不使用目录，请使用 glob 模块。glob 模块的 glob()函数支持大多数 Unix shell 下都使用的通配符选项。表 11-8 列出了所支持的通配符规范。

例如，要得到包含所有 HTML 文件的一个列表，请使用如下命令：

```
files = glob.glob('*.html')
```

返回值是个字符串列表，每个元素指定一个单一文件。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

要执行更复杂的匹配，请使用第 10 章论述的正则表达式检查每个文件名。例如，要找到所有由大写字母和一个单个数字构成文件名称的文件，比如说，THEFILE1.txt，应使用如下所示的代码：

```
filelist = []
for file in glob.glob('./*'):
    if (re.search(r'[A-Z]*?[0-9].[a-z]*', file)):
        filelist.append(file)
```

表 11-8　查找文件的通配符规范

规　　范	说　　明
*	匹配任何数量的任何字符
?	匹配一个单一字符
[seq]	匹配 seq 序列中的任何一个字符
[!seq]	匹配不是 seq 序列中的任何一个字符

11.4　基本文件/目录管理

大多数用于移动、重命名和其他操纵目录的函数能在 os 模块中找到。这些函数大多数都是自解释的，概括于表 11-9 中。

表 11-9　Python 生成、重命名和删除文件 / 目录的函数

函　　数	说　　明
chdir(name)	改变当前工作目录为 name 目录
getcwd()	返回路径到当前工作目录
link(source, dest)	创建 source 和 dest 之间的硬链接。此函数为所有平台 / 操作系统支持
mkdir(path [, mode])	利用 mode 模式生成目录 path。如果不给出 mode 参数的值，其默认设置为 0777
makedirs(path [, mode])	等价于 mkdir()函数。不同之处在于 makedirs()函数生成 path 定义的所有目录，诸如，makedirs('/a/b/c')命令在这些目录不存在的情况下，创建所有目录，即'/a'、'/a/b'和'/a/b/c'
remove(path)或 unlink(path)	删除给定文件
removedirs(path)	删除在 path 中指定的目录，包括所有子目录及文件(等价于 rm -r)。如果目录不能被删除掉，将引发 OSError 异常
rename(source, dest)	把 source 重新命名为 dest
renames(source, dest)	等价于 rename()。不同之处在于，所有 dest 参数指定的不存在的目录以 makedirs()函数所用的相同方式被创建
rmdir(path)	删除目录 path
symlink(source, dest)	创建 source 和 dest 之间的符号链接。此函数不为所有平台 / 操作系统支持

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

对于诸如拷贝文件和树形目录这样的更高级选项，请使用 shutil 模块。shutil 模块包含实现如下功能的函数：基本文件拷贝、模式和权限拷贝。此外，shutil 模块还包含复制和删除完整树形目录的实用函数。表 11-10 包含 shutil 模块所支持函数的清单。这种支持是所有操作系统都提供的。

表 11-10　shutil 模块用于复制和删除目录的函数

函　　数	说　　明
copyfile(src, dst)	把 src 参数所指定目录下的文件复制到 dst 参数指定的目录下
copymode(src, dst)	把 src 的权限位复制到 dst 上
copystat(src, dst)	把 src 的权限、修改时间和访问时间复制到 dst 上。请注意，此函数不复制文件的内容，也不改变文件的拥有者和组
copy(src, dst)	把 src 的文件复制到 dst 目录或文件中
copy2(src,dst)	把 src 的文件复制到 dst 的目录或文件去，还复制文件的修改时间和访问时间
copytree(src, dst [,symlinks])	把 src 指定的整个树形目录复制到 dst 目录下。文件用 copy2()复制。如果 symlinks 参数值是真，符号链接就在目的地重新创建；如果 symlinks 值是假或者默认，每个文件内容都作为普通文件被复制
rmtree(path [, ignore_errors [, onerror]])	删除 path 目录下的整个树形目录。如果 ignore_errors 参数值是真，所有错误都被忽略；如果 ignore_errors 参数值是假，错误就被 onerror 参数指定的函数处理，所指定的函数必须接受 3 个参数：func、path 和 excinfo。其中，func 参数指的是产生错误的函数(如，rmdir()函数或是 remove()函数)；而 path 参数指定正被删除的目录或文件；excinfo 参数是 sys.exc_info()引发的一组异常信息

11.5　访问和所有权

常常需要确定或设置给定文件的文件选项和权限。os 模块提供了一系列函数获取和设置文件权限和其他信息。还能使用输出与 stat 模块的一个组合来提供有关返回信息的一个更清晰的接口，或者是与有关一些统计数字的 os.path 模块组合在一起。

11.5.1　检查访问

要检查访问文件的能力，请使用 os.access()函数，如下所示：

```
os.access(path, accessmode)
```

其中，path 参数应是要检查的文件或目录的路径；而 accessmode 参数是个常量，此常量指定要检查的访问方式的类型。accessmode 参数的有效取值如下所示：

- R_OK　　(能读)
- W_OK　　(能写)

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

- X_OK　　(能执行)
- F_OK　　(检测存在性)

例如，要检查是否具有读取一个文件的权限，可以使用如下所示的命令：

```
os.access('myfile.txt', os.R_OK)
```

11.5.2　获取文件信息

要获得有关一个文件的信息(即，其权限、拥有者和访问时间)，请使用 os.stat 函数(关于文件的)或使用 os.lstat 函数(链接的)。这两个函数都以值的元组的形式返回相同信息。该值包含关于此文件的信息。表 11-11 列出了返回值及其元素的引用和 stat 模块导出的常量，stat 模块通过名称而不是通过索引来访问信息。

表 11-11　stat()模块返回的元素

元 素 引 用	stat 常量	说　　明
0	ST_MODE	文件模式(类型和权限)
1	ST_INO	Inode 数字
2	ST_DEV	文件系统的设备数字
3	ST_NLINK	文件(硬)链接的数字
4	ST_UID	文件拥有者的数值用户 ID
5	ST_GID	文件拥有者的数值组 ID
6	ST_SIZE	文件的尺寸大小，以字节为单位
7	ST_ATIME	从新纪元计起的最后一次访问时间
8	ST_MTIME	从新纪元计起的最后一次修改时间
9	ST_CTIME	从新纪元计起的 Inode 改变时间(不是创建时间)

例如，要获得文件的尺寸大小，请使用如下所示的命令：

```
filesize = os.stat(file)[6]
```

为获得、设置或确定文件类型和其他信息，可以使用 stat 模块的 stat()命令的输出，stat 模块导出常量来访问 stat()元组的不同元素，这些元素如表 11-11 中的 stat 常量列所示。此外，stat 模块还提供表 11-12 所列出的函数，这些函数返回文件某一给定方面的摘要信息。请注意，所有函数都接受一个单一参数：即由 os.stat(path)[stat.ST_MODE]返回的模式。

表 11-12　stat 模块导出的检测函数

函　　数	说　　明
S_ISDIR(mode)	如果路径是个目录，就返回真

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

函　　数	说　　明
S_ISCHR(mode)	如果文件是个特殊的字符设备文件，就返回真
S_ISBLK(mode)	如果文件是个特殊的块设备文件，就返回真
S_ISREG(mode)	如果文件是个规则文件，就返回真
S_ISFIFO(mode)	如果路径是个先进先出(FIFO)，就返回真
S_ISLNK(mode)	如果路径是个符号链接，就返回真
S_ISSOCK(mode)	如果路径是个 Unix 套接字，就返回真
S_IMODE(mode)	返回能被 os.chmod()函数设置的文件模式部分
S_IFMT(mode)	返回描述文件类型的文件模式部分

通过 os.path 模块，一个快速接口有效，此接口提供一些实用函数。能使用其中某单个函数调用获得有关文件的大多数信息。表 11-13 列出了确定文件信息的 os.path 模块所支持的函数。

表 11-13　Python 获取文件信息的可选方法

函　　数	说　　明
getatime(path)	返回从新纪元计起以秒数表示的最近一次访问时间
getmtime(path)	返回从新纪元计起以秒数表示的最近一次修改时间
getsize(path)	以字节为单位返回文件的尺寸

11.5.3　设置文件权限

实际上，在 Python 中要设置文件模式和所有权，请相应地使用 os.chmod()和 os.chown()函数。请注意，这些函数仅应用于 Unix 样式的操作系统(包括 QNX、BeOS 和 MacOS X)下，而不能应用于 Windows 或 MacOS 系统下。例如，要设置一个文件的权限，可以使用如下所示的命令：

```
os.chmod('file.txt', 0666)
```

请注意，权限必须指定为八进制数字，正如在 Unix 命令行上指定模式时一样。os.chown()函数接受 3 个参数：路径、用户标识符和组标识符。例如，要改变 file.txt 为用户标识符 1001、组标识符 1000 所拥有，请使用如下所示的语句：

```
os.chown('file.txt', 1001, 1000)
```

想使用用户名而不想使用数字，请使用 pwd 和 grp 模块来确定给定用户或组的标识符数值。关于跨引用目的，表 11-14 包含了 os.chmod()函数和 os.chown()函数的格式。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

表 11-14 Python 访问模式的设置

Python 函数	说 明
os.chmod(path, mode)	改变 path 的权限模式为 mode
os.chown(path, uid, gid)	改变 path 的用户标识符和组标识符

11.6 操作文件路径

在处理文件的时候，一定需要操作目录和文件或操作要打开文件和目录的路径。操作这类信息不是件易事，特别是想支持不同的平台，就更不容易了。例如，Unix 平台使用正斜杠字符(/)分离目录的各个组成部分；而 MacOS 平台则使用冒号字符(:)；Windows 平台使用反斜杠字符(\)，当然 Windows 95 及其新版本平台也支持正斜杠字符(/)。

要使此类处理变得容易些，os 模块导出定义当前平台设置的变量。表 11-15 列出了保存这类信息的变量。

表 11-15 os 模块的目录元素变量

变 量	说 明
sep	用于分离路径元素的字符：在 Unix 平台上为“/”；在 MacOS 平台上为“:”；在 Windows 平台上为“\”
pathsep	用于分离$PATH 环境变量中每个路径的字符：“:”用于 Unix 平台、“;”用于 Windows/DOS 平台
pardir	用于描述父目录的字符序列：在 Unix/Windows 平台上为“..”；在 MacOS 平台上为“::”
curdir	用于描述当前目录的字符序列：在 Unix 和 Windows 平台上为“.”；在 MacOS 平台上为“:”
altsep	用作可选分离符的字符。这实际上就应用于 Windows 平台上，Windows 平台能使用正斜杠字符(/)

在这些信息的武装下，就能利用如下命令组成一个文件的路径：

```
fullpath = basedir + os.sep + dir + os.sep + file
```

但是，这太杂乱了。而实际上也计算不到这么精确，能删除最后路径上的重复分隔符。比较好的解决方法是由 os.path 模块提供的，os.path 模块提供了一些函数，这些函数能把任何平台上一给定路径组合成为或分解成为正确的格式。这些函数就列在表 11-16 中。

basename()函数和 dirname()函数两者都是对一给定路径进行分解，并返回此路径的目录和/或文件名，请见下例：

```
>>> os.path.basename('/usr/local/lib/python2.1/os.py')
'os.py'
>>> os.path.dirname('/usr/local/lib/python2.1/os.py')
'/usr/local/lib/python2.1'
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

表 11-16　Python 中的路径操作函数

Python 函数	说　明
os.path.join(patha [, pathb ...])	利用当前平台合适的路径分隔符把 patha 和 pathb 合并成一个有效路径
os.path.abspath(path)	返回 path 的清空版本，删除对当前目录和父目录的引用
os.path.basename(path)	抽取给定路径的基本名称(文件名或最后一级目录)
os.path.dirname(path)	抽取给定路径的目录名

abspath()函数清除路径，得到对当前目录和父目录的引用，其目的是抵达最后一级清除的路径。如下所示：

```
'/var/adm/su.log'
```

最后，join()函数把路径各个组成部分合并在一起，包括当前平台合适的分隔符。返回一个新的有效路径，如下所示：

```
'/usr/local/python2.1/os.py'
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第12章　数据管理和存储

前面两章已经研究了信息处理和从文件往外读信息及往文件里写信息这些操作。但是如果信息比较复杂，不是简单的一组字符串或一组单词，那怎么办呢？怎样才能实现以高效的方式存储和处理列表和字典里的信息呢？

我们所面临的问题有一些与如何管理和处理保存信息的内部对象有关。所要关注的两个特定领域是：序列排序(在其他章节中已简要论及)和如何恰当地复制对象而不是引用对象。

本章的第二部分将全力研究用于存储和加载对象和源文件中的数据(不是纯文本文件的数据)的方法。

12.1　管理内在结构

创建变量，存储一些信息于变量之中，以及稍后利用其带回的返回值，这些统统都是变量管理范畴的论题。Python 拥有强大的排序能力，但是为了更好地使用它们需要通盘考虑以下这两个方面：排序信息的类型——列表、元组或字典——包含于要排序的序列内的数据类型。例如，在排序一个简单文本或者数字时所进行的比较是一目了然的；那么在对日期或者对象排序时，所要进行的比较就难了。

还有需要注意的是，在复制和使用对象时。因为 Python 使用对象的引用，所以增加或者更新对象的引用是可能的，而复制对象就是不可能的了。通过强制 Python 把一个对象复制为新对象(及其相关的引用)能解决上述难题。

12.1.1　序列排序

在 Python 中进行信息排序要求要有一点构思和计划，才能得到正确的排序。列表内置对象类型支持 sort()方法，sort()方法把内容排序到适当的位置上。在排序 Python 列表时，要谨防下列三件事情：

- 永远是通过调用 list.sort()来排序列表。除非已经给列表做了个备份，否则没有办法保存列表的初始顺序；
- 字典是不能进行排序的，只能抽取出字典的键列表或值列表，然后对列表进行排序；
- 元组不能进行排序。如果接收一个作为函数返回值的元组(许多函数都返回元组而不返回列表)，需要先把元组的内容复制到一个新列表里，接着再对此列表进行排序。

所幸的是，Python 的这种方法也有一些优势，比如说，对数字和字符串构成的列表而言，会自动按数字和字母表的顺序进行排序；而且因为使用对象的备份版本进行信息的排序，所以还能单独对备份版本进行排序，以产生一个按自定义顺序排序的结果。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

1. 元组排序

因为元组是不可改变的，所以不能对其进行排序——元组内建对象甚至不支持 sort()方法。解决元组排序问题的方法是：用内置函数 list()把元组备份为列表，然后对这个备份列表进行排序。如下例所示：

```
tuplecopyaslist.sort()
for item in tuplecopyaslist:
        print item
```

2. 字典排序

在重点讲述字典的第二章里，已讲过这方面的一个例子，但是做个快速温习还值得的。从根本上而言，不能在其他语句内直接得到字典的键排序列表。例如，下面的 for 语句就不正确：

```
for item in dict.keys().sort():
```

取代上述方法，需要把字典的键列表存到一个新变量里，然后再对新变量进行排序，如下所示：

```
keylist.sort()
for item in keylist:
        print '%s is %s' % (item, dict[item])
```

3. 排序函数

在对非标准数据进行排序的时候，或者在想把数据正规化而不影响列表本身内容进行排序的时候，就需要使用这样的函数——匿名函数或命名函数——来做排序操作。

不管使用哪一个函数，给定函数两个参数，即要进行比较的两个元素。对列表而言，这很简单，就是要比较的两个元素值；对字典而言，就是先前从字典里抽取出来的两个键或两个值。而在排序文本信息时，就有问题了。因为实际的结果是依照 ASCII 表对数据进行了排序，遗憾的是，这导致 *A* 排在了 *a* 的前面，而 Z 也排在了 *a* 的前面。要避免此类问题发生，用一个单独的排序函数在比较期间把元素改变为小写字母，因而把所有内容都改变为相同大小写，所以就能对各个数据项进行正确的排序。

能利用一个匿名函数进行上述排序工作，如下所示：

```
list.sort(lambda x,y: return cmp(string.lower(x), string.lower(y)))
```

或者，利用一个单独的函数进行上述排序工作，如下所示：

```
return cmp(string.lower(x), string.lower(y))
list.sort(noncasesort)
```

请注意，在调用排序函数的时候，提供的是关于 noncasesort()函数的代码对象，而不是实际调用 noncasesort()函数。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

12.1.2　复制对象

Python 始终利用的是对数据的引用，这就引发了一个问题，对数据结构造成严重的破坏。其中，这些数据结构使用引用，而不是使用对其他变量的数据备份。举例，如下所示的命令就创建了两个变量，alist 和 blist：

```
>>> blist = alist
```

其中：blist 变量是对 alist 变量同一个数组对象的一个引用，这就意味着改变变量 alist 中的一个值就修改了 blist 变量也指向的同一个列表，如下所示：

```
>>> alist
[25, 50, 75, 2000]
>>> blist
[25, 50, 75, 2000]
>>>
```

得到完全一样的结果。alist 变量和 blist 变量两者指向同一个数组，因此，修改其中一个变量所指向数组中的数值，其后从另一个变量取得的数值也被同样修改。

要对列表或其他序列做个备份，就能抽取出一个全长切片对象，举例如下所示：

```
>>> blist = alist[:]
```

通过指定列表切片的取值，从而能把独立对象从 alist 复制到 blist 里，如下所示：

```
>>> blist
[25, 50, 75, 2000]
```

上述操作创建的被称作为浅拷贝，不拷贝其嵌套结构，如下所示：

```
>>> blist = alist[:]
>>> alist.append(5)
>>> blist
[1, 2, [3, 4]]
>>> alist[2][1] = 5
>>> blist
[1, 2, [3, 5]]
```

要执行深拷贝，深拷贝把所有元素(包括其嵌套结构)拷贝为一个全新的变量，需要使用 copy 模块的 deepcopy()函数。举例如下所示：

```
>>> alist = [1,2 [3,4]]
>>> blist = copy.deepcopy(alist)
>>> alist[2][1] = 5
>>> blist
[1, 2, [3, 4]]
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

12.2　对象持续期

往往，想要的不仅仅是存储和读取基本的文本文件。如果处理诸如列表、字典或嵌套结构的任何形式的复杂数据结构，那就需要更加具体的对象来存储信息。

12.2.1　对象存储

如果在 Python 应用程序中创建了一个结构，并且想把这个结构永久地记录在文件里，Python 有一些可选的有效方法。最适合的方法很大程度上依赖于所创建对象的复杂性。就最简单的情况而言，也不可能单单是想存储单一的数字或者字符串。即使是想存储单一的数字或者字符串，前面几章讲述的文件读取和写入方法就能轻松地解决这个问题。对列表和元组来说，假定存储的是其他诸如字符串和数字这样的基本类型，能很轻松地用多个数据行把信息存到文件里，其中每个数据行表示数组的一个元素。因此把信息读回来，也就是把每个数据行读出来，再把读出的数据添加到列表里这么一个简单的操作。必须确保把字符串中所有的值都转换回其数字格式——string 模块包含有把字符串转换成为整数和浮点数值的函数。当然，在理想情况下，最好把这类操作给诸如 mySQL 这样适当的数据库系统去处理。像 mySQL 这样的数据库系统还提供了能轻松地搜索和获取所要信息的方法，而无需装载或者分析整个文件。详见本章后面的“商业数据库”部分。

对字典而言，这就更复杂了，但其信息仍能容易地用文本形式表示出来——再次假定处理字典的简单的字符串和/或数字键和值。利用标准文本，只能把信息作为独立的数据行写入到文件里，其中每一个键/值对存在于同一个数据行上，用一个单个字符(冒号或者等号)把值分开。字典经常用于存储配置数值，如果读者见到过大多数配置文件的格式，那将发现使用的都是相对标准的格式，就像这样：

```
SaveLines = True
```

在把文件读进来的时候，需要的只是再次把键/值对分离而已，这个工作用 string.split()函数就能实现。再容易点的方法，就是用 DBM 数据库把字典存储在其整体里：那就不需要加载值或分析文本文件了。详见本章后面的“DBM 数据库”部分。再好一点的方法是把信息写成 Python 有效语句，就可以直接导入文件。

迄今为止，当然一直是在处理相当简单的结构——字符串、数字和简单的序列。在存储含有嵌套结构的复杂结构时——比如说，以字典为元素构成的字典或是由列表组成的列表，如果只是使用简单的文件，处理过程就变得更复杂了。当着手把自己的对象或者二进制信息存储于一典型文件里时，处理过程就变得更加困难了。要解决此类问题，那就采用像 XML 这样的构造数据格式，但仍然需要解决把原来的结构转变成为 XML 的这个难题。

Python 提供了许多解决这个问题的方法，这些解决方法通过一些标准模块给出。这些标准模块差不多把所有对象串行化成为一个文件或者成为一个字符串，然后再把所生成信息解串回去成为原来的对象。根据所使用的模块不同，实际的串行化过程完全不同。但大多数时候都使用与在此论述的原则相同的基本原则，虽然是个更具结构化的格式，但其目的是把内部对象转

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

化成为永久存储的文件或受字符串支持的格式。

在 Python 的标准配置情况下，有两个模块支持把对象串行化为文件或字符串的功能，它们是 marshal 模块和 pickle 模块(或者是实现 C 语言的 cPickle 模块)。

1. 利用 marshal

marshal 模块仅支持简单对象类型——即数字、字符串、列表、元组、字典和代码对象。序列对象仅能使用所支持的子类型。因为 marshal 模块不如 pickle 模块灵活性强，所以 marshal 模块可能不能应用于生产系统。如果试图存储一种不兼容的类型，那要引起争议。

然而，如果上述所有问题都没吓倒您，使用 marshal 模块本身是件简单的事。要把一个对象存储到文件里，请使用 dump()函数。dump()函数接受两个参数：一个参数是要转储的对象，另一个参数是要把对象存储于其中的已打开文件对象。举例如下所示：

```
names = { 'Martin' : 29,
          'Rikke' : '30',
          'Andy' : '33', }
namesdb = open('names','w')
marshal.dump(names, namesdb)
```

如果给定对象不是个所支持的类型，将引发 ValueError 异常。信息就像写其他数据一样被写到文件里，因此，可在最终关闭文件之前向此文件添加新的数据和附加对象。总之，在读取信息的时候，必须具备要加载对象的首地址。

要再次加载对象，请使用 load()函数。此函数接受一个单一参数：一个对象和可读文件对象，如下所示：

```
namesdb = open('names','r')
names = marshal.load(namesdb)
```

如果不能从给定的文件加载对象，此函数将引发 EOFError、ValueError 或 TypeError 异常。

除了这两个以文件为基础的函数之外，marshal 模块还支持两个以字符串为基础的函数：dumps()函数和 loads()函数。dumps()函数串行化一个对象，但不是把此对象写入到文件里，dumps()函数以字符串的形式返回串行化了的对象。接下来，loads()函数将从一个字符串加载回对象。

2. 利用 pickle/cPickle

pickle 模块在 pickle 模块和 cPickle 模块这两者中功能更强。pickle 模块能对任何对象进行串行化，甚至于对类也进行串行化。此模块把信息写到一个文件里或一个字符串里，就像 marshal 模块一样处理信息。cPickle 模块和 cPickle 扩展一起使用对对象进行串行化，其速度很显著，快于 pickle 模块。cPickle 模块惟一的限制是不能继承于 cPickle 类。

cPickle 模块提供两个基类：一个是 Pickler 类，另一个是 UnPickler 类。Pickler 类用于对对象进行串行化(封藏)；而 UnPickler 类用于对对象进行解串(解封)。要使用 Pickler 类，请创建 Pickler 对象的一个新实例。Pickler 类的指令接受两个参数：一个参数是文件对象，另一个参数是选项的二进制参数。如果第二个参数是假或未给定数值，对象就被串行化为能读取的文本格

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

式。但是空间利用率很低，原因是其格式化和结构过分地依赖于文本。如果第二个参数是真，串行化操作就采用二进制格式，而二进制格式所使用的空间很小并且二进制格式是不可读的。

产生的类实例结果支持两种方法。dump()方法，在类实例创建时把对象转储到指定的文件里。例如，要存储早期的字典，所用命令如下所示：

```
names = { 'Martin' : 29,
          'Rikke' : '30',
          'Andy' : '33', }
namesdb = open('names','w')
pickler = pickle.Pickler(namesdb)
pickler.dump(names)
```

Unpickler 类用于加载一个封藏的对象并把它转换回一个内部对象。解封程序能自动识别出信息是利用文本还是用二进制格式封藏的。解封程序只接受一个参数：打开从其中读取数据的文件对象。例如，要把字典读取回来，其命令如下所示：

```
namesdb = open('names','r')
unpickler = Unpickler(namesdb)
names = unpickler.load()
```

请注意，在用 marshal 模块时，能使用所有文件并能尽可能多地调用 dump()和 load()，只要以相同的顺序调用这些函数就行。

12.2.2　DBM 数据库

DBM 通常就是数据库管理(Database Management)的缩写，然而确实再没有任何人相信这个缩写。数据库管理使用键 / 值对，并且把数据存储在文件系统上的一个或者两个文件中。DBM 的初始规范说明应该能够抽取一个磁盘入口键所附加的值。这导致 DBM 是个十分快速的可能有些简化了的数据库系统。

这些年来，虽然基本规范仍保持相同，但已经不完全是原来的 DBM 系统，已有很大改进，并且其改进历经了几个不同版本。大多数不同的 DBM 系统互相兼容，根据所涉及平台和设备的不同，兼容的程度也不同，或达较大的程度，或达较小的程度。还要注意的是，DBM 文件不可移植。对于特定的硬件平台和操作系统，所使用的存储格式是一定的。在有些情况下，甚至于同一个操作系统的不同版本间都不具备相互兼容的 DBM 系统。

信息存储方式不同使得很难用标准工具进行文件拷贝。一个标准的 DBM 数据库由两个文件组成。其中之一是包含一个位图文件且以.dir 为后缀的目录；第二个文件包含所有的数据，且以.pag 为后缀。数据文件常常充满着空洞，这些空洞的存储空间已被分配。数据文件当中这些空洞存储空间真正所包含的都是些无用信息。这种方法的不利趋势是设备分配过多的存储空间，进而生成这样的文件：报告的占用空间是其所存有用信息空间的 10 倍、100 倍、甚至高达 1 000 倍。而其他设备，如 GDBM 和 Berkeley DB，则使用一个单个文件。仍然存在空间分配浪费这个问题，浪费的空间只不过是存在于一个文件里，而不是两个文件中。

DBM 老式工具，对存储每个键/值对的尺寸大小有所限制，存储每个键/值对的尺寸被称作

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

为存储桶尺寸(bucket size)。尺寸比存储桶尺寸大的记录，根据数据库在 C 语言层上实现方式的不同，导致数据库返回一个异常或者把试图存储的信息截断。下面几节将陆续论述常见的 DBM 工具。表 12-1 对有关常见 DBM 工具的信息进行了概括总结。

表 12-1　DBM 设备

设　　备	DBM/ODBM	NDBM	SDBM	GDBM	Berkeley DB
存储桶尺寸限制	1~2K	1~4K	1K (无)	无	无
磁盘使用	可变的	可变的	小	大	大
速度	慢	慢	慢	不错	快
数据文件是否可分配	否	否	是	是	是
字节顺序是否独立	否	否	否	否	是
用户定义排序顺序	否	否	否	否	是
通配符查找	否	否	否	否	是

- **DBM**

DBM 是 DBM 工具包的早期版本。虽然它被大多数 UNIX 变体系统作为标准工具包含于其中，但几乎完全被 DBM 设备 NDBM 选项取代。大多数系统支持的存储桶尺寸为 1K，但也有些系统比这小。

- **NDBM**

NDBM 是早期 DBM 的“新”版工具，其速度和存储分配机制都有一些改进。NDBM 已替代了标准 DBM 函数库。在有些情况下，它就是惟一的有效设备。根据操作系统的不同，存储桶的尺寸也不同，其取值范围是 1K~4K。

- **SDBM**

SDBM 代表 DBM 的替代/简化版本，是 DBM 的速度和稳定性均加强了的版本。SDBM 只对 Perl 有效。而 Python 并不存在对 SDBM 的标准接口，即使是遇到使用 SDBM 这类情况，这个接口也没有存在的必要。能够用 NDBM 或 GDBM 打开一个 SDBM 文件，当然不能保证其兼容性(或不被推荐的)。如果必须使用 SDBM 文件的信息，请使用 Perl 先把 SDBM 文件转换为 NDBM 或 GDBM。所用脚本与如下命令相似：

```
use GDBM_File;
use Fcntl;

die "Usage:$0 old new\n" if (@ARGV<2);

my($old,$new) = @ARGV;

tie (%oldhash, 'SDBM_File', $old, O_RDONLY, 0444)
    || die "$0: Error opening source $old: $!\n";
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
tie (%newhash, 'GDBM_File', $new, O_CREAT|O_RDWR|O_EXCL, 0666)
        || die "$0: Error opening dest $new: $!\n";

while(($key, $value) = each(%oldhash))
{
        $newhash{$key} = $value;
}

untie %oldhash || die "$0: Error closing old DBM file, $!\n";
untie %newhash || die"$0: Error closing new DBM file, $!\n";
```

SDBM 支持大小为 1K 的存储桶，但能在编译的时候修改存储桶的尺寸。

- GDBM

GDBM 是 DBM 的 GNU/FSF 设备。GDBM 的速度比所有设备都快，惟独比 Berkeley DB 慢。GDBM 设备已被输入到许多平台，而其他设备就没被输入到这么多平台当中去。GDBM 支持有限的文件，支持 DBM 文件里的记录锁定机制。记录锁定机制在多用户环境下是非常有用的。对 GDBM 数据库而言，没有存储桶尺寸这个概念——键 / 值对可为任意长度。

- Berkeley DB

Berkeley DB 是数据库访问方法的公共域 C 语言库，包括 B+树、扩展线性散列和定长 / 不定长记录。bsddb 模块提供对在 Python 内利用 Berkeley DB 文件的扩展支持，对 B-树和散列设备两者也加以了支持。进而能像个 DBM 替代版设备一样使用。也可以直接利用 dbhash 模块访问 Berkeley DB 散列数据库接口。

在下列脚本中，使用 anydbm 模块。dbhash 模块利用最有效的设备打开(或创建)DBM 数据库。依次，dbhash 模块试着使用 Berkeley DB 散列、GDBM、DBM 或 dumpdbm 数据库。这就意味着将打开那些数据库(假定系统支持它们)中的任何一个数据库，或者创建一个基于最有效设备基础之上的数据库。现代系统可能不具备 NDBM 设备了，而使用 GDBM 或 Berkeley DB 设备则要求安装一个单独模块。dumpdbm 数据库仅仅是个兼容性接口，它真正是为大型开发所使用。dumpdbm 数据库速度很慢，效率低，并且常常不一致。

- 利用 DBM

用 DBM 数据库能做什么完全取决于开发者。在最简单的情况下，所能做到的就是用来存储信息的磁盘字典；使用它存储配置信息；或者如果具有一个相对简单的数据库，用来存储其所有信息。

如果想处理大型内部字典，使用 DBM 数据库无疑是个好方法，DBM 数据库通过把字典存储转移到一个外部文件里去降低了应用程序所覆盖的内存空间。

如果要打开一个 GDBM 数据库，命令如下所示：

```
import gdbm
dict = gdbm.open('mydbmdatabase')
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

其中，进程创建的 dict 对象只是个普通的字典。除了存储数值是把此数值写到外部文件里和访问文件是从文件往外读取信息之外，在所有其他方面，此对象的表现与普通字典别无两样。

总之，要意识到不能用数字或字符串之外的其他对象来组成键或值：不能使用嵌套对象、列表或字典，除非使用了 pickle 模块或类似的串行化嵌套信息的模块。一个更显而易见的解决方法是通过 shelve 模块实现的。shelve 模块的工作方式与 DBM 模块相似——shelve 模块提供一个像字典一样的对象，但不像 DBM 系统那样，shelve 模块支持嵌套结构。

其他 DBM 模块的工作方式相同。每个模块都提供同样的 open()函数，open()函数的一般格式如下所示：

```
open(file [, flag [, mode]])
```

其中，选项参数 flag 允许指定以什么模式打开数据库，数据库的打开模式使用通用文件打开模式，就是在第 3 章和第 11 章中讲过的 open()内置函数所使用的通用文件打开模式。其默认设置为只读模式。选项参数 mode 是个八进制数值，用来在数据库文件不存在的情况下创建数据库文件。其默认设置为 0666。

例如，要打开一个 GDBM 数据库，所用命令如下所示：

```
dict = gdbm.open('mydbmdatabase, 'r', 0666)
```

如果不知道要打开的 DBM 文件的类型，而使用了不正确的模块打开文件，导致的结果可能是数据毁损，也可能是读内容失败这样的简单错误。要避免这类错误的发生，可以使用 Python 的 anydbm 模块。anydbm 模块对 DBM 数据库的类型进行识别，然后用相应的模块打开文件、读文件。因此，能重新编写上例，命令如下所示：

```
dict = anydbm.open('mydbmdatabase, 'r', 0666)
```

12.2.3　商业数据库

虽然使用的是“商业”这个术语，但其真正所指的却是所有大型数据库系统，如自由的 mySQL 和 PostgreSQL 系统，或像 Oracle 和 Sybase 这样的商业系统。

有所遗憾的是，现在对所有不同数据库系统还没有一个一致的单一的接口。尽管数据库系统研究小组(db-sig)正努力打造一个数据库接口，此接口支持对几乎所有数据库系统一致的接口。但是直到现在为止，还是需要到 Parnassus 的 Vaults 网站(http://www.vex.net/parnassus/)上搜寻一番，以找到对应于所要使用数据库系统的接口模块。比如说，MySQL-Python 模块允许访问 mySQL 数据库。

下面的脚本演示了怎样运行一个 SQL 查询并把得到的结果打印出来，代码如下所示：

```
import MySQLdb

dbh = MySQLdb.Connect(db='clients')
mycursor = mydb.cursor()

stmt = "select firstname, lastname from clients"
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
cursor.execute(stmt)
results = cursor.fetchall()

for firstname, lastname in results:
        print firstname, lastname

mydb.close()
```

其他模块可用于与其他数据库系统进行对话。查看 Python 网站和 Parnassus 的 Vaults 网站(详见附录 B)。在这两个网址上，能找到有用的模块。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第13章 网 络 通 信

一定有这样的时候：在不同的机器间进行文件拷贝。这意味着要把文件放在一个磁盘上再在上述机器间传送此文件。显然这对大文件不实用，特别是在利用 Jaz 或其他大存储格式的时候，就更不奏效了。然而，我们拥有网络。在网络上，各台机器间是彼此相互连接的。现代 LAN(局域网)是个相对简单的网络：线缆从一台机器连到另一台机器上，或者，对于大多数使用以太网的局域网来讲，线缆就连接到中央集线器或路由器上。有了互连网这个新事物，进程甚至于可以在比较远的距离上处理。利用路由器、网桥和其他信息的一个组合，能在放置于地球两个极上的两台机器之间通信。

本章，我们将关注网络编程的基本知识，涵盖从寻址机制和交换信息一直具体到打开网络连接的所有内容。Python 还提供了一个客户端协议模块的扩展函数库，客户端协议模块是直接与 Web 服务器、邮件服务器和其他各种各样服务的通信。本章就将一一论述所有这些论题。

13.1 Networking 101

大多数网络系统一步一步地继承下来，以 ISO/OSI(国际标准化组织/开放式系统互联)七层模型为基础。每层定义网络处理过程中的一个独立单元，从物理连接一直到使用网络的应用程序。每层根据其在七层模型中所处位置的不同，提供所要求的不同服务。

大多数人上现在都赞同这样一个观点：七层模型已不再实用。对所有适应七层模型协议的法则，有太多的异常和模糊不清之处。许多协议处于 OSI 模型的两个结构层之间，而不是很合适地处于一单个结构层上。实际上，七层模型仍然有其有武之地：本章所关注的协议大部分仍然直插七层模型上——而其他像 NetBEUI(在 PC 机上使用，用于局域网的通信)和 AppleTalk/EtherTalk(为 Mac 机使用，用在局域网上)这样普通的局域协议就不适合七层模型。

13.1.1 逻辑连接类型

不管特殊的理论模型是什么样，应用于所有网络技术之上的是同样的基本法则。能赋予网络以逻辑连接类型这个特性。一个网络既可以是面向连接的也可以是无连接的。面向连接的网络依赖于这样一个事实：相互间要对话的两台机器必须进行一些形式的连接处理，通常称作握手信号。这个握手信号与使用电话相似：呼叫方拨号，而接听方摘机。以这样的方式，因为接听方要对呼叫应答，所以呼叫方能马上知道对方是否接收到消息。此类型的连接为 TCP/IP 协议(传输控制协议 / 因特网协议)支持，并且是因特网和局域网(LAN)主要的通信形式。

在无连接网络上，信息发送给接听方，而无需先建立连接。由于数据以不连续的数据包发送，此类网络也是个数据报网络或是面向数据包网络。每个数据包由发送方地址、接听方

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

地址和信息组成，而且一旦消息被接收到，也不提供应答。因此，无连接网络更像个邮递服务——编写并发送一封书信，却不能保证此封书信一定抵达目的地，也不能保证信息被准确无误地接收到。无连接网络为 UDP/IP 协议(用户数据报协议 / 因特网协议)支持。

不管是在哪一种情况下，两台机器间的“线路”并非永久打开。数据是以分散的数据包发送的，分散的数据包可以经由不同的路径和不同的路线到达目的地。路线可以包括局域网、拨号连接、ISDN 路由器，甚至于还可以包括卫星链接。在 UDP 协议中，数据包可以以任意顺序到达，数据包被提交给客户端程序，由它来把数据包重新组装成为正确的数据序列，就是这样。对 TCP 协议而言，数据包被自动重新组装成为正确的序列，然后再以一个单个数据流提交给客户。

这两种类型的网络各有优缺点。无连接网络速度快，因为不要求对数据进行应答也不涉及任何为接收数据而建立连接的对话。总之，无连接网络还是不可靠的，原因在于没有办法确保信息一定到达其目的地。面向连接网络速度慢(对比无连接网络而言)，因为附加了额外的对话，所以保证了数据序列头尾衔接的可靠性。

无连接系统大多时候用于广播数据或送出不期待返回数据的信息。例如，UDP 协议能用来检查网络上客户端的可靠性——一个 UDP 数据包被发送到网络上，所有看到此数据包的机器(因为此数据包没有固定目的地址)，接下来应答说它们是终止的和还是正在运行。因为数据是基于 UDP 协议广播的，任何机器都能监听得到，而鉴于多媒体流是客户端永远不需要对服务端应答的惟一途径，UDP 还被一些因特网视频数据流服务程序所使用。事实上，最好的办法大概就是把 UDP 当作是个收音机或电视广播来理解。

面向连接系统用于要求在两台机器之间进行准确无误的双向通信时的所有情况下。因为客户端连接到一特定服务器上，发送一些信息，并期待一些返回信息，所以电子邮件服务、Web 服务和文件传送服务全部都使用面向连接系统。面向连接系统与一个电话呼叫相似：想利用一台特定的电话机呼叫某个人或者和某个人说话，即使连线另一端提供的是诸如Web 服务这样的普通服务。

13.1.2　网络名称和数字

TCP/IP 协议和 UDP/IP 协议当中的 IP 指的是因特网协议，因特网协议是指定网络中各个机器独立地址的一组标准。位于网络世界当中的每台机器都拥有一个惟一的 IP 地址。IP 地址由 4 个字节的序列构成，在典型写法上加以句点注释，例如，192.168.1.1。这些数字与网络中的独立机器和机器的整个集合两者都息息相关。

因为人类并不擅长记忆数字，所以一个称作 DNS(域名系统)的系统就出现了，DNS 把容易记忆的名字和 IP 地址联系在一起。例如，www.mcgraw-hill.com 这个名字就和一个单个 IP 地址相联。当然一个单个 DNS 名字也可以指向几个 IP 地址，而多个名字也可以指向同一个 IP 地址。还可以：一台单个机器拥有多个接口，每个接口有多个分配给它的 IP 地址。总之，不管是哪一种情况，如果接口以几种形式连接到因特网上，则每个接口的 IP 地址一定是惟一的。

13.1.3　网络端口

有关通信的规范远不止这些。许多不同应用程序都能在同一台机器上运行，因此通信不

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

仅仅要针对于机器，还要针对于与特有协议和最终的监听应用程序相联的机器端口。如果把IP地址和电话号码相比，那端口号就等价于分机号。

前1 024个端口号为知名的因特网协议而保留，并且每个协议都有拥其自己的端口号。例如，HTTP协议(超文本传输协议)，用于在Web浏览器和Web服务器之间传输信息，就拥有端口号80。要连接到一个服务器应用程序上，需要具备IP地址(或机器名)和正在监听的服务器端口号两者。总之，端口号仅仅是对IP地址的扩展。尽管端口80保留并默认用于进行HTTP通信，但这并不能说明在端口2000、8000或8081上运行HTTP服务程序就不可以。请注意，Unix为特定协议而保留了前1 024个端口号。如果创建自己的服务器，需要使用在这之上的端口号。除非拥有特权，否则不能使用这前1 024个端口号。

13.1.4 网络通信

任何形式的网络应用程序都需要交换信息，但是对交换幕后的机制要留点神。控制双向通信有许多不同的方法，因此没有任何系统完全可靠。最明显的就是“最佳猜测”状态，连接的每一端都处于这个“最佳猜测”状态。如果其中一端发送了一个信息段，然后假定信息传输是安全的而等待应答。如果另一端也作同样的假设，在接收到一些信息之后马上发送信息。

“最佳猜测”这个环境并非最可靠，其原因是，如果发送和接收两端在同一时刻决定等待信息，实际上连接的两端就无效了。另一个选择是，如果发送和接收两端在同一时刻决定发送信息，则这两个进程将不锁定。但是因为它们使用同一个发送——接收系统，一旦两者都发送了信息，则两者就都返回到等待状态，期待着一个应答。诚然，应增加超时来断掉连接再重新建立连接，但是这解决不了根本问题：两端于同一时刻互相对话、做同样的事。

对于此问题，一个较好的解决方法是使用一个协议，此协议对于通信方法和顺序都加以严格规定和限制。这就是简单邮件传输协议(SMTP)和类似协议工作的方式。客户端向服务器发送一个命令，服务器立即对客户端应答并告知客户端下一步做什么。应答可以包括数据，一点也不错，还包括数据的结束字符串。事实上，它与无线通信时所使用的技术类似。在每次通信的结束处，都要说一声“结束”来向接听者指明已经完成讲话。从根本上而言，仍然是使用最好猜测的相同方法通信。假定通信准确无误地被启动，并且每个通信端点都发送了通信结束信号，则通信应能正确地继续下去。

FTP协议使用一个有点不同的解决方法。文件传输协议用于在机器间传输文件，但是它还有一个交互接口，此交互接口允许从当前客户端导航远程机器的目录。FTP的交互元素叫作控制连接，并且通常使用端口25。当传输如文件或监听目录这样数量大的信息时，两端就打开一个数据连接。如果有什么原因导致数据连接失败，仍然还有一个控制连接，所以还能再次尝试进行数据传输。

13.1.5 BSD Socket 接口

BSD (Berkeley 系统部分，Unix的一个“特色”)套接字系统在BSD 4.2中引入，作为一个方法给不同有效协议提供了一个一致接口。套接字提供应用程序和网络之间的一个连接。要在机器间通信，在连接的每一端上就必须有一个套接字。在同一时刻，其中一端必须被设

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

置为接收数据，而另一端则在发送数据。只要套接字连接的每一方都知道自己是该发送信息还是该接收信息，通信就能是双向的。

现在，socket接口被普遍地接受，成为操作系统和底层硬件间的接口。大多数操作系统，包括Mac OS、Unix、Windows、BeOS和其他多数操作系统在内，都使用BSD socket函数库来支持其网络通信。

在Python中，关于所有网络通信的基本模块毫无疑问，就是socket模块。socket模块或多或少是底层C语言函数库的一个映像。您如果用其他语言编写过socket系统，由于socket模块采用了一贯的语法，所以利用Python下的socket模块，一定是轻车熟路。

除此之外，Python还支持许多模块，这些模块都建立在socket模块之上，处理诸如HTTP、FTP、SMTP、NNTP和其他一些这么重要的协议。

标准配置还包括一些服务器模块，这些模块提供普通的socket服务器或HTTP服务器。事实上，标准函数库包括一个用Python语言编写的完整Web服务器，叫作BaseHTTPServer，此程序支持BaseHTTPServer类。适合于CGI的版本，是CGIHTTPServer模块。

如果想更加详细地了解有关TCP、UDP和IP协议下套接字和数据流网络的知识，在此向读者推荐一本书《UNIX系统V4版本程序员指南：网络接口》(Prentice-Hall出版公司于1990年发行)，此书涵盖了网络的所有规定，还包括使网络起作用所要求的C语言源代码。

13.2 获取网络信息

在开始与远程机器通信之前，需要有一种方法能确定机器的IP地址。操作IP地址和机器名称就要使用socket模块提供的一组基本函数。表13-1列出用于地址解析的模块所支持的函数清单。

表13-1 Python网络信息函数

函　数	说　明
gethostbyname(hostname)	根据其hostname获取一个主机的IP地址
gethostbyname_ex(hostname)	根据其hostname获取有关一个主机的扩充信息。此函数返回一个元组，此元组包含主机名、一个关于主机名的别名列表和一个IP地址的列表
gethostbyaddr(address)	返回与gethostbyname_ex()函数相同的信息，但是是关于一个给定IP地址的，而不是一个名字的
getprotobyname(protocolname)	获取一个常量，此常量在socket模块中使用，与具体的协议名称相关联。例如，getprotobyname('icmp')返回IPPROTO_CMP。此函数可能不为所有操作系统支持
getservbyname(name, protocol)	根据协议和服务名字获取一给定协议号。例如，getprotobyname('tcp','http')就应返回80。此函数可能不为所有操作系统支持
gethostname()	返回当前机器的主机名

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

例如，能获取关于一给定主机的 IP 地址，所用命令如下所示：

```
address = gethostbyname('www.python.org')
```

Python 以字符串的形式返回 IP 地址，其格式是每隔 3 位添加一个句点(即 xxx.xxx.xxx.xxx，例如，192.168.1.1)，此信息不封装于一个单一结构里。

gethostbyname_ex()函数返回扩充信息，如果需要与远程机器对话，这个远程机器又有多个服务器和/或多个接口，当然也就有多个 IP 地址。gethostbyname_ex()函数返回的扩充信息就非常有用。下面就能清楚地看到这一点：www.python.org 是通过一个单一 IP 地址服务的，但是 www.altavista.com 被多个地址服务。如下所示：

```
>>> socket.gethostbyname_ex('www.python.org')
('parrot.python.org', ['www.python.org'], ['132.151.1.90'])
>>> socket.gethostbyname_ex('www.altavista.com')
('altavista.com', ['www.altavista.com'], ['209.73.164.96',
'209.73.164.97', '209.73.164.98', '209.73.164.99', '209.73.180.1',
'209.73.180.2', '209.73.180.3', '209.73.164.93', '209.73.164.94',
'209.73.164.95'])
```

13.3 基本套接字函数

实际上，使用服务器或客户端的套接字，需要一个特定的操作顺序。在打开与现存 TCP/IP 服务进行通信的套接字这个情况下，其操作顺序如下：

(1) 打开套接字(利用 socket())；

(2) 连接到远程机器上(利用 connect())；

(3) 使用 recv()和 send()来从服务器往外读取信息或往服务器里写入信息。

在 Python 中做此事，实际上很简单——socket.socket()函数创建一个新套接字。此函数接受两个参数，这两个参数就是套接字家族和套接字类型。它们是两种基本的套接字系列类型。Unix 套接字家族(通过常量 socket.AF_UNIX 有效)通过创建一个用于在两个系统间通过共享文件进行通信的套接字(但是仅在 Unix 系统下适用)。Unix 域套接字常常用于同一个系统上的两个进程之间通信的情况中，如，一些 Unix 打印系统和系统日志工具实际上就是通过 Unix 套接字实现其操作的。

因特网套接字家族(socket.AF_INET)是用于 TCP 和 UDP 通信的套接字家族，其中 TCP 和 UDP 通信被 FTP、SMTP 和大多数其他兼容因特网协议使用。在因特网套接字家族中有两个主要套接字类型。这两个主要套接字类型已经讲过了，就是 UDP 和 TCP。UDP 是个数据报套接字类型，能利用 socket.SOCK_DGRAM 常量来指定；TCP 套接字是个底层流，能用 socket.SOCK_STREAM 常量使其可用。总的来说，因特网套接字能用在几乎所有的通信中，除非与现存 Unix 套接字系统通信，否则大概只得使用因特网套接字。

例如，要打开一个 TCP/IP 套接字，应使用如下所示的命令：

```
mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

其中，socket()函数返回一个套接字对象，因此，通过此套接字对象上使用一系列方法来实现更进一步的选项。从上述清单可见，下一步要连接远程服务器，通过 connect()方法就能做到这一步。connect()方法接受一个单个参数，是个元组对，此元组对包含远程机器的主机名(或 IP 地址)和端口号。

要得到端口号，请在套接字对象上使用 getservbyname()方法。getservbyname()方法接受两个参数：服务名字和协议名。例如，要获取 TCP 服务上的 HTTP 协议的端口号，可使用如下所示的命令：

```
httpport = mysock.getservbyname('http','tcp')
```

现在，能用以下命令连接一个远程机器：

```
mysock.connect(('www.mcwords.com','httpport'))
```

到此为止，已经与远程机器连接上。能使用 recv()和 send()方法来接收远程服务器的信息和向远程服务器发送信息。

实际上，可以把这整个过程压缩到一个单一函数中。如下所示：

```
import socket

def open_tcp_socket(remotehost,servicename):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    portnumber = socket.getservbyname(servicename,'tcp')
    s.connect((remotehost,portnumber))
    return s
```

现在，能打开一个 TCP 套接字，向 HTTP 服务器发送一个请求，并能利用如下所示的命令把结果打印出来：

```
mysock = open_tcp_socket('www.mcwords.com','http')
mysock.send('GET http://www.mcwords.com\n\n')
while (1):
    data = mysock.recv(1024)
    if (data):
        print data
    else:
        break
mysock.close()
```

表 13-2 列出了有关 socket 对象所支持方法的完整清单。socket 模块导出的关于套接字选项和消息发送标志的常量分别列于表 13-3 和表 13-4 中。

表 13-2　socket 对象所支持的 socket 方法

方　　法	说　　明
S = socket(family, type)	用 family 和 type 打开一个套接字。其中，family 参数是个常量，或为 socket.AF_UNIX，或为 socket.AF_INET

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

（续表）

方 法	说 明
S.accept()	接受来自于远程主机的一个连接
S.bind(address)	把一个套接字绑定到一个特定地址上。在创建服务器的时候，或者在需要绑定到机器的特定地址上，而此机器的一个网络接口卡又支持多个 IP 地址的时候，需要使用 bind 方法
S.close()	关闭网络套接字
S.connect((address,serviceport))	在 serviceport 端口的 address 地址处打开一个到主机的连接
S.getpeername()	获取此套接字相连接的远程主机名
S.getsockname()	获取此套接字相连接的本地主机名
S.getsockopt(level,option [,buflen])	获取一个套接字选项。套接字选项设置一个可配置数值，此可配置数值是关于一个已经打开的套接字的。其中，level 参数应当是选项要应用于其上的层——socket.SOL_SOCKET 或 socket.IPPROTO_IP。其中，socket.SOL_SOCKET 是关于套接字层选项或协议号的，而 socket.IPPROTO_IP 是关于具体协议选项的。option 参数指定想要包含的选项值；buflen 参数指定返回信息的最大长度。如果不指定 buflen 参数的值，则所有数据都被返回。请参见表 13-3，表 13-3 是所有套接字所支持的选项数值的一个清单。请注意，根据操作系统的不同，所支持选项的实际清单会有所不同
S.listen(waitqueue)	启动对新连接的监听。waitqueue 参数是附加连接被拒绝前的连接队列号
message = S.recv(buflen [,flags])	从套接字中读出至多为 buflen 字节的数据。flags 参数定义怎样接收信息——请参见表 13-4，表 13-4 列出的是 socket 模块中的常量所支持的依赖于系统的数值清单
(message, address) =S.recv(buflen [, flags])	等价于 recv()，不同之处在于此方法的返回值是个元组，元组由从套接字中读出的数据和接收数据的地址组成。此函数尽管就要对 TCP 套接字有效，但现在仅真正适用于 UDP 套接字
S.send(message [, flags])	把 message 消息发送给远程机器。flags 参数与 recv()的 flags 参数等价
S.sendto(message [, flags],(address, port))	等价于 send()，不同之处在于此函数中定义的消息发送给(address,port)元组所指定的机器。此函数尽管就要对 TCP 套接字有效，但现在仅真正适用于 UDP 套接字
S.setsockopt(level, option,value)	设置套接字的 option 为 value。请参见表 13-3。表 13-3 是一个可以取值的清单。请注意，并非所有的套接字都支持所有选项。请查看一下套接字文档或者查看一下 Python 建立期间生成的 socketmodule.c 文件，看看到底支持什么选项

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

（续表）

方　法	说　明
S.shutdown(how)	关闭一个套接字。如果 how 参数为零，此套接字就不再接收信息；如果 how 参数为 1，此套接字就不再发送信息；如果 how 参数为 2，此套接字既不能用于发送信息也不能用于接收信息。这用于使用两个套接字与一台远程主机通信的情况下，每个套接字能分别用于发送信息或接收信息

表 13-3　可通过 socket 模块配置的套接字选项

选　项	有 效 值	说　明
SO_KEEPALIVE	0,1	如果设置为 1，套接字保持活动状态，定期询问连接的另一端。定期询问实际上不发送任何数据。在对多个请求进行服务的时候，这用于服务器套接字。利用一个套接字多次与利用多个套接字相比，前者提高了访问速度
SO_RCVBUF	整数	以字节为单位指定接收缓存的尺寸大小
SO_SNDBUF	整数	以字节为单位指定发送缓存的尺寸大小
SO_REUSEADDR	0,1	允许立即拒绝本地地址。如果值为零(默认设置)，端口将被封锁一小段时间。即使服务器已经终止了，也要封锁端口一小段时间。
SO_RCVLOWAIT	整数	在通知 select()调用之前读出的字节整数数目
SO_SNDLOWAIT	整数	在通知 select()调用之前发送到缓存中的有效字节整数数目
SO_RCVTIMEO	整数	以秒为单位的整数，接收数据的超时值
SO_SNDTIMEO	整数	以秒为单位的整数，发送数据的超时值
SO_OONINLINE	0,1	把带外数据放置到输入队列中
SO_LINGER	0,1	如果设置为 1，在仍然有数据等待发送的情况下，套接字在 close()之后继续拖延。默认设置为零
SO_DONTROUTE	0,1	如果设置为 1，数据包绕过普通路由表(仅仅适用于发送)
SO_ERROR	整数	返回错误状态
SO_BROADCAST	0,1	如果设置为 1，套接字将允许发送广播数据报文(仅仅适用于 UDP 套接字)
SO_TYPE	整数	返回套接字的类型
SO_USELOOPBACK	0,1	如果设置为 1，发送套接字将接收到它所发送数据的一个拷贝。这用于验证，验证实际发送的数据是否与所应发送的数据匹配

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

表 13-4 发送数据 / 接收数据的有效标志

消 息 标 志	说 明
MSG_PEEK	允许这样读取数据：不把读出的信息从缓存中删除
MSG_WAITALL	从套接字中读出 recv()调用所请求的字节数的全部数据，否则不返回
MSB_OOG	发送或接收带外数据
MSG_DONTROUTE	输出数据包绕过普通路由表(只适用于发送)

如果想能够像对文件那样对一个网络套接字进行读和写，请使用 os 模块的 fdopen()函数。fdopen()函数创建一个新的文件对象，新的文件对象是基于文件描述符数字之上的。文件描述符在大多数操作系统(Unix 和 Windows NT 变体系统，但不包括 BeOS 及 MacOS 系统)上为文件和套接字两者共享。利用 fileno()方法能获得任何文件的文件描述符数字或套接字对象。例如，能重新编写前面使用的代码，此代码从 SMTP 服务器读取数据。前面使用的代码如下所示：

```
mysock = open_tcp_socket('mail.mchome.com','smtp')
message = mysock.recv(1024)
print message
mysock.close()
```

重新编写为如下所示的代码：

```
mysock = open_tcp_socket('mail.mchome.com','smtp')
file = os.fdopen(mysock.fileno())
line = file.readline()
print line,
file.close()
mysock.close()
```

此段代码先打开套接字，然后创建一个基于套接字文件描述符数字的文件对象，接着用 readline()命令读取信息。还可以利用 makefile()方法自动完成上述操作(无需 os 模块)。如下所示：

```
mysock = open_tcp_socket('mail.mchome.com','smtp')
file = mysock.makefile()
line = file.readline()
print line,
mysock.close()
file.close()
```

如果与诸如 SMTP、HTTP 或 FTP 这样的一个标准因特网服务通信，请使用 Python 的一个模块，本章稍后就将讲述一些选择模块。socket 模块所支持的其他函数列于表 13-5 中。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

表 13-5　socket 模块所支持的其他函数

函　数	说　明
fromfd(fd, family,type [, proto])	利用指定的 family 和 type 创建基于 fd 中已打开文件描述符之上的一个套接字对象。文件描述符必须是个套接字，而不能是文件句柄。请注意，不是所有平台都支持此函数
ntohl(x)	把 32 位整数的字节顺序从网络字节顺序(big-endian)转换为主机字节顺序
ntohs(x)	把 16 位整数的字节顺序从网络字节顺序转换为主机字节顺序
htonl(x)	把 32 位整数的字节顺序从主机字节顺序转换为网络字节顺序
htons(x)	把 16 位整数的字节顺序从主机字节顺序转换为网络字节顺序

13.4　创建网络服务器

在创建网络服务器时，其过程略有些不同。对客户端套接字而言，既然是创建一个对已经存在机器的连接，那对服务端套接字而言，就需要创建一个套接字，然后监听输入的连接。一旦接收到连接，就能像客户端套接字那样读出和写入信息。对于 TCP/IP 套接字的实际操作顺序如下所示：

(1) 创建一个套接字对象；

(2) 把套接字绑定到一个本地地址上(利用 bind())；

(3) 监听连接(利用 listen())；

(4) 在接收到一个来自于客户端的连接请求时，接受此连接(利用 accept())；

(5) 利用 read()和 write()从服务器读出信息，再把读出的信息写到服务器里。

要想把服务器一侧的进程与客户端一侧联系在一起，请参见图 13-1。

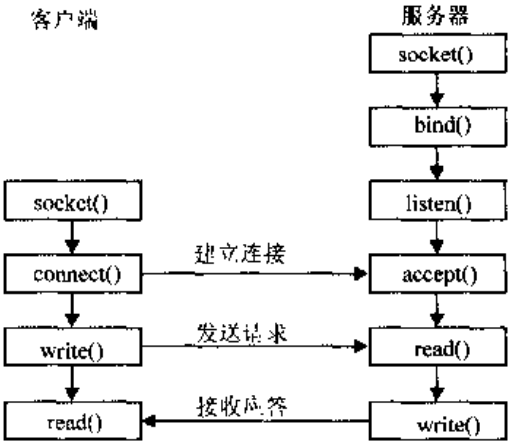

图 13-1　服务器和客户端两端的交互操作

最大不同之处在于第 2 步和第 3 步。bind()进程对照着具体的 IP 地址和端口来注册套接字，实际上采用的是与 connect()指定客户端一侧信息非常相近的方式。listen()只是把套接字置于这样的一个状态：套接字已经准备好可以接受连接。而对客户端套接字的连接实际上一直要到 accept()被调用之后才能完成。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

利用如下所示的代码能达到上述要求：

```
import socket

s = socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind('', 8000)
s.listen(5)

while 1:
    client,addr = s.accept()
    print "Accepted a connection from", addr
    data = client.recv(1024)
    client.send("You said: " + data)
    client.close()
```

但是这个例子还有些问题：第一，许多工作必须由自己动手来完成，一定有个更轻松的办法能完成这些工作；第二，只有在循环环境里，才处理多个客户的请求。如果同一时刻有 5 个客户要连接，只能依次顺序地一一处理所有这些请求。对简单的服务器来说，这还可以说得过去。但对现实世界的服务器而言，这样系统将产生严重的瓶颈问题。

利用 SocketServer 模块能解决上述那两个问题。SocketServer 模块提供一个基于类之上的用于支持网络服务的系统。

13.4.1　利用 SocketServer 模块

要支持大多数网络服务器所要求的多连接，可以使用如下常见方法：调用 os.fork()或使用 select 模块，甚至于使用线程。如果创建网络服务，那利用 SocketServer 模块会更好。SocketServer 模块提供多个不同的类，这些类被设计为用于处理网络通信。SocketServer 模块提供 8 个不同的类，这些类提供支持类似于 HTTP、SMTP 和其他服务的基本网络服务所要求的所有机制。有关所支持类的清单，请参见表 13-6。

表 13-6　SocketServer 模块所支持的网络服务器

类	说　明
TCPServer(address, handler)	一个基本的基于 TCP 协议之上的服务器。其中：address 参数应当是个元组，此元组包含用来建立服务的主机名和端口号。如果 hostname 参数设置为一个空字符串，则此服务器就绑定到当前主机的默认地址上。handler 参数应当是 BaseRequestHandler 类的一个实例——有关如何创建所需处理程序的信息，请参见主要文档
UDPServer(address, handler)	一个基本的基于 UDP 协议之上的服务器
UnixStreamServer(address, handler)	一个服务器，它使用 Unix 域套接字，以流模式进行通信
UnixDatagramServer(address, handler)	一个服务器，它使用 Unix 域套接字，以数据报模式进行通信
ForkingUDPServer(address, handler)	等价于 UDPServer，惟一不同之处在于此服务器利用 os.fork()处理多个请求

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

类	说　明
ForkingTCPServer(address, handler)	等价于 TCPServer，惟一不同之处在于此服务器利用 os.fork()来处理多个请求
ThreadingUDPServer(address,handler)	等价于 UDPServer，惟一不同之处在于此服务器利用线程建立多个客户端处理程序线程来处理多个请求
ThreadingTCPServer(address, handler)	等价于 TCPServer，惟一不同之处在于此服务器利用线程建立多个客户端处理程序线程来处理多个请求

为了使用这些基类创建一个服务器，需要创建一个 BaseRequestHandler 类的新子类，BaseRequestHandler 类是由 SocketServer 模块提供的。此类最起码至少要定义 handle()方法。handle()方法在客户端每次获得连接的时候都被调用。对象实例(self)包含多个不同属性，这些属性包括 socket 对象和客户端地址。其中 socket 对象如 request 一样，用来与服务器通信；而客户端地址如 client_address 一样，构成连接。

一旦新的子类创建了，所有需要做的就是创建一个想要的服务器类的新实例，并把刚刚创建的子类传递给此实例处理程序。最后，需要调用 handle_request()方法和 serve_forever()方法中任一个方法。其中，handle_request()方法处理一个单个请求；而 serve_forever()方法处理所有请求，一直到终止服务器进程为止。

例如，以下脚本创建一个叫做 TimeHandler 的处理程序，TimeHandler 处理程序返回数据、服务器的时间，或者返回客户端请求时服务器的新纪元时间值。如下所示：

```
import SocketServer, socket, string, time

class TimeHandler(SocketServer.BaseRequestHandler):
    def handle(self):
        socketfile = self.request.makefile()
self.request.send('Hello %s, What do you want?\r\n'
                  % (self.client_address,))
while 1:
    line = socketfile.readline()
    print "Got %s\n" % (line,)
    line = string.strip(line)
    if not line:
        break
    if line == 'time':
        self.request.send("The time is: %s\r\n"
                          % (time.strftime('%H:%M:%S',
                             time.localtime(time.time())),))
    if line == 'date':
        self.request.send("The time is: %s\r\n"
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
                                % (time.strftime('%d/%m/%Y',
                                    time.localtime(time.time())),))
        if line == 'datetime':
            self.request.send("The time is: %s\r\n"
                                % (time.strftime('%d/%m/%Y %H:%M:%S',
                                time.localtime(time.time())),))
        if line == 'rawtime':
            self.request.send('%d\r\n' % (time.time(),))
        socketfile.close()

server = SocketServer.TCPServer(('', 8000), TimeHandler)
server.serve_forever()
```

利用上述脚本，能很轻松地获得远程机器的时间，如下所示：

```
$ telnet twinsol 8000
Hello ('198.112.10.135', 43712), What do you want?
time
The time is: 15:09:39
date
The time is: 14/04/2001
datetime
The time is: 14/04/2001 15:09:43
rawtime
987257386
```

这是一个基本的例子，但是能把这个服务扩展成为所需要的任何基于网络的解决方案。

13.4.2　运行 HTTP 服务

如果想提供 HTTP 服务，Python 标准配置还包括几个用于支持 HTTP 服务的模块和相关基类。这些模块和相关基类实际上基于 SocketServer 模块之上。有 3 个模块和类，它们就是：BaseHTTPServer 模块、SimpleHTTPServer 模块和 CGIHTTPServer 模块 / 类。其中，BaseHTTPServer 模块涵盖接受请求的基本接口；而 SimpleHTTPServer 模块处理简单的 GET 请求——此模块适用于提供基本 Web 服务器服务。

最后的 CGIHTTPServer 模块 / 类建立在 SimpleHTTPServer 模块之上，也允许对 CGI 请求进行服务。使用这些模块最简单的方法是调用每一个模块的 test()函数。例如，要在当前目录下启动一个 HTTP 服务器，就很简单，如下所示：

```
Python 2.1 (#2, Apr 29 2001, 14:36:04)
[GCC 2.95.3 20010315 (release)] on sunos5
Type "copyright", "credits" or "license" for more information.
>>> import SimpleHTTPServer
>>> SimpleHTTPServer.test()
Serving HTTP on port 8000 ...
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

13.5　客户机模块

Python 有许多模块用于像客户机一样与网络服务器通信。关于所支持的协议和模块，请参见表 13-7。本部分就讲述某些客户机模块。

表 13-7　Python 网络客户机模块

客户机协议	Python 模块	说　明
SMTP(简单邮件传输协议)	smtplib	用于发送电子邮件
FTP(文件传输协议)	ftplib	用于上传或下载文件
NNTP(网络新闻协议)	nntplib	用于传输 Usenet 新闻发布
POP(Post Office Protocol)	poplib	用于访问电子邮件信箱
IMAP(因特网邮件访问协议)	imaplib	用于读取和保存电子邮件
HTTP/FTP	urllib	用于把统一资源定位符(URL)下载到本地磁盘上

13.5.1　使用 SMTP

SMTP(简单邮件传输协议)用于从您的机器上发送电子邮件，围绕网络分布此电子邮件，一直到最终抵达要发送到的用户信箱为止。利用 Python 的 SMTP 发送电子邮件要比调用像 mail 或 sendmail 这样的扩展应用程序有效，而且它具有能用在所有平台之上的优势。

Python 的 smtplib 类使用一个基于类的模块：一旦创建了一个 smtplib 类的新实例，就能使用方法名字与 SMTP 命令相同的方法来建立和发送电子邮件。一个比较快的方法是，如果所要做的就是发个快件，用 sendmail()方法。sendmail()方法接受 3 个参数：发送者、接听者和电子邮件消息。可以使用如下所示的代码：

```
import smtplib

server = smtplib.SMTP('localhost')
server.sendmail('mc@mcwords.com','mc@mcwords.com',
                "Subject: Reminder\nDon't Mail Self\n")

server.quit()
```

还有一个获取用户输入使用地更广泛的例子。此代码片段取自于 smtplib 模块的底层。如下所示：

```
import sys, rfc822
from smtplib import *

def prompt(prompt):
    sys.stdout.write(prompt + ": ")
    return string.strip(sys.stdin.readline())
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
fromaddr = prompt("From")
toaddrs = string.splitfields(prompt("To"), ',')
print "Enter message, end with ^D:"
msg = ''
while 1:
    line = sys.stdin.readline()
    if not line:
        break
    msg = msg + line
print "Message length is " + `len(msg)`

server = SMTP('localhost')
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

13.5.2 使用 FTP

ftplib 模块提供一个到远程 FTP 服务器的基于类的接口。能利用 FTP 类创建一个 ftplib 类的新实例，如下所示：

```
import ftplib
remote = ftplib.FTP('ftp.python.org')
```

现在，这个 FTP 类的新实例应通过提交想要发送给远程服务器的命令，利用指定期望接收的信息类型的方法而使用。例如，在获得远程文件列表的时候，可以使用 retrlines()方法接受多个输出行，如下所示：

```
remote.retrlines('LIST')
```

实际上，要接收一个文件，可以使用 retrbinary()方法。此方法要求给定命令和一个对象，其中对象要指向一个合适的文件对象方法。举例如下：

```
remote.retrbinary('RETR pytersrc.tgz',
open('pytersrc.tgz','w').write())
```

13.5.3 使用 HTTP

HTTP(超文本传输协议)为 Web 浏览器和 Web 服务器使用，用来传输 Web 页。HTTP 不像 FTP 那样，FTP 提供目录浏览和双向文件传输功能。HTTP 被设计为支持一个客户端简单请求，统一资源定位符或客户端所请求的 URL 的路径元素，然后简单地把文件返回给客户端，再没有其他处理了。对客户端而言，没有办法请求文件清单及其他信息，HTTP 服务器所能做的全部就是返回所请求的文件。

虽然有一个单独的 httplib 模块，用于在 HTTP 协议之上进行通信。但是十之八九，所想

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

做的是把一具体 URL 下载到您的机器上。Urllib 模块提供一个很简单的接口把一特定 URL 下载到本地文件中。如下所示：

```
import urllib

urllib.urlretrieve('http://www.python.org', 'pythonhomepage.html')
```

如果想处理正下载的信息，而不想把此信息临时存储到一个扩展文件当中，可以使用 urlopen 方法。此方法打开远程 URL，返回一个文件对象，因此能像是个本地文件一样处理服务器返回的信息。

例如，在最简单的情况下，重新编写 urlretrieve()函数为如下所示的命令：

```
import urllib

url = urllib.urlopen('http://www.python.org')
file = open('pythonhomepage.html','w')

while 1:
    line = url.readline()
    if len(line) == 0:
        break
    file.write(line)
```

实际上，想分析 HTML 下载页中的信息，需要使用 htmllib 模块。在本书的第 20 章里，将讲述分析 HTML。但是为了给出一个关于能做什么的粗略指南，以下脚本的功能是，先下载一页，然后处理其中的 HTML，抽取出图像列表，接着下载该页上的每一个图像。其命令如下所示：

```
import htmllib, sys,formatter, urllib, urlparse, os.path

# Create a new parser class to read handle the HTML
class ImgParser(htmllib.HTMLParser):
    def __init__ (self, formatter):
      htmllib.HTMLParser. __init__ (self, formatter)
      self.imglist = []
# Extract the information from an image tag when seen
    def handle_image(self, src, alt, ismap, align, width, height):
      self.imglist.append((src, alt, ismap, align, width, height))
# Get the URL from the command line
try:
    url = sys.argv[1]
except:
    print 'You must supply a URL'
    sys.exit(1)
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
# Open the URL as a file
try:
    urlfile = urllib.urlopen(url)
except IOError, msg:
    print "Error:", url, ":", msg
    sys.exit(1)

# Create a new parser based on our parser class, and then
# read each line from the file and pass it on to the parser
# for processing
parser = ImgParser(formatter.NullFormatter())
while(1):
    line = urlfile.readline()
    if line:
      parser.feed(line)
    else:
      break

urlfile.close
parser.close()

# Based on the list of images extracted from the file,
# get the full URL of each image, then the filename
# and download each to disk
for img in parser.imglist:
    imgurl = urlparse.urljoin(url,img[0])
    imgfile = os.path.basename(urlparse.urlparse(imgurl)[2])
    urllib.urlretrieve(imgurl,imgfile)
```

13.5.4 使用 IMAP

Python 的 imaplib 模块提供一个关于 IMAP(因特网邮件访问协议)服务器的基于类的接口。IMAP 是 POP3 的另一个选择，IMAP 不仅允许存储输入的电子邮件，还允许存储您在中央服务器上存储的电子邮件。就这一点而论，IMAP 协议比 POP3 协议更复杂。

下面的代码样本只是对作者在《Python Annotated Archives》一书当中包括的脚本略微修改了一下。此代码从一个单一电子邮件账户那里下载电子邮件，显示结果为 HTML 页，因此，能从浏览器检查到输入的电子邮件，而无需必须激发一个电子邮件客户请求。代码如下所示：

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

import imaplib
import sys,os,re,string

# Set up a function that we can use to call the corresponding
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
# method on a given imap connection when supplied with the
# name of an IMAP command
def run(cmd, args):
    typ, dat = apply(eval('imapcon.%s' % cmd), args)
    return dat

# Get the mail, displaying the output as HTML

def getmail(title,login,password):

# Login to the remote server, supplying the login and
# password supplied to the function
    run('login',(login,password))

# Use the select method to obtain the number of
# messages in the users mail account. The information is returned
# as a string, so we need to convert it to an integer
    nomsgs = run('select',())[0]
    nomsgs = string.atoi(nomsgs)

# Output a header for this email account
    print "<p><font size=+2><b>"+title+"</b></font></p>"

# Providing we've got some messages, download each message
# and display the sender and subject
    if nomsgs:
# Output a suitable table header row
        print "<table border=0 cellpadding=0 cellspacing=0>"
        print "<tr><td><b>Sender</b></td><td><b>Subject</b></td></tr>"
# Process each message
        for message in range(nomsgs,0,-1):
            subject,sender,status = ",",'U'
# Send the fetch command to the server to obtain the
# email's flags (read, deleted, etc.) and header from the email
            data = run('fetch', (message,'(FLAGS RFC822.HEADER)'))[0]
          meta,header = data
# Determine the email's flags and ignore a message if it's
# marked as deleted
          if string.find(meta,'Seen') > 0:
              status = ''
          if string.find(meta,'Deleted') > 0:
              continue
# Separate the header, which appears as one large string, into
# individual lines and then extract the subject and sender fields
          for line in string.split(header,'\n'):
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
                    if not line:
                        sender = re.sub(r'\<.*\>','',
                                        re.sub(r'\"','',sender))
# If the message is unread, then mark the subject and sender
# in red
                        if (string.find(status,'U') == 0):
                            subject = '<font color=red>'
                                      + subject + '</font>'
                            sender = '<font color=red>'
                                      + sender + '</font>'
                        print "<tr><td>%s</td><td>%s</td></tr>"
                            % (sender,subject)
                        break
# Extract the sender/subject information by looking for the field
# prefix
                    if line[:8] == 'Subject:':
                        subject = line[9:-1]
                    if line[:5] == 'From:':
                        sender = line[6:-1]
                print "</table>"
        else:
                print "No messages"
# Logout from the server
        run('logout',())
# Set the server information
server='imap'

# Print out a suitable HTTP header and HTML page header
print "Content-type: text/html\n\n"
print """
<head>
<title>Mail</title>
</head>
<body bgcolor="#ffffff" fgcolor="#000000">
"""

# Connect to the server, and then call getmail to get the mail
# from the server
try:
    imapcon = imaplib.IMAP4(server)
except:
    print "Can't open connection to ",server
    sys.exit(1)
getmail('MC','memcslp','PASSWORD')
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

此脚本的运行结果能在图 13-2 中看到。

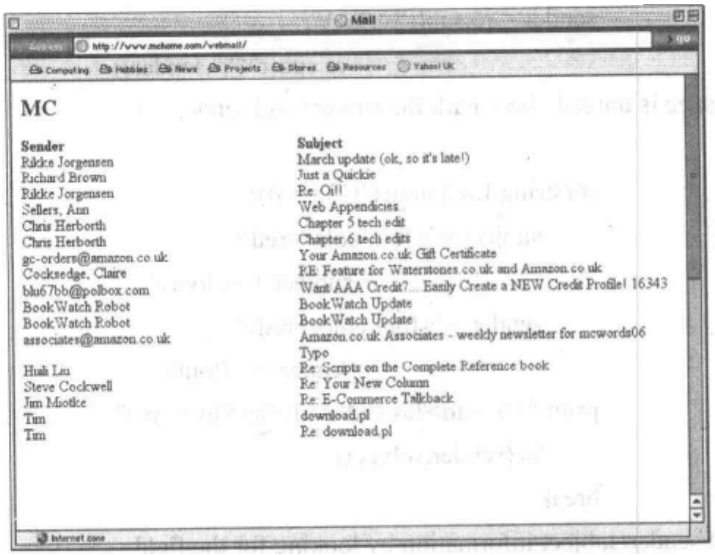

图 13-2　利用 IMAP 检查电子邮件

13.6　处理因特网数据

除了因特网所支持的标准文本和图像格式之外，还有几个附加的标准格式，用于表示多文件文档。其他格式用来把 8 位二进制数据编码为 7 位二进制数据格式，7 位二进制数据是传输电子邮件使用的格式。虽然实际上，在当代大多数电子邮件系统上，这并不是所要求的，但是这是从仅仅只支持 7 位字符集的 UUCP 老系统承接过来的。

13.6.1　base64

base64 模块用于利用 base64 编码格式对数据进编码和解码。这常用在 MIME 和其他多部分消息中，用来以跨平台方式对二进制信息进行编码。base64 模块所支持的函数列于表 13-8 中。

表 13-8　base64 模块的函数

函　数	说　明
decode(input, output)	对文件或 input 中的文件对象的 base64 格式的数据进行解码，解码后的结果存到文件或 output 文件对象里
decodestring(s)	对 base64 格式的 s 进行解码。返回解了码的数据流
encode(input, output)	对文件或 input 文件对象的数据进行编码为 base64 格式，编码后的结果存到文件或 output 文件对象里
encodestring(s)	对 s 进行编码为 base64 格式。返回编了码的数据流

13.6.2　binascii

此模块提供用于编码或解码几个不同 ASCII 码格式的函数。此模块所支持的函数列于表 13-9 中。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

表 13-9 binascii 模块所支持的函数

函 数	说 明
a2b_base64(string)	把 base64 格式编码的 ASCII 码字符串 string 转换为二进制格式
a2b_hqx(string)	把 binhex 格式的 ASCII 码字符串 string 转换为二进制格式
a2b_uu(string)	把未编码的 ASCII 码字符串 string 转换为二进制格式
b2a_base64(data)	把二进制数据 data 转换为 base64 ASCII 码格式
b2a_hqx(data)	把二进制数据 data 转换为 binhex ASCII 码格式
b2a_uu(data)	把二进制数据 data 转换为未编码的 ASCII 码格式
rlecode_hqx(data)	利用 RLE 算法压缩 data，以 binhex ASCII 码格式返回压缩过的字符串
rledecode_hqx(data)	利用 RLE 算法(Run Length Encoding)对以 binhex ASCII 码格式压缩的数据 data 进行解压缩
xcrc_hqx(data,crc)	创建一个关于数据 data 的 CRC(循环冗余校验)和。其中，crc 参数应当是用于 CRC 的开始值

此模块支持两个不同的异常，如表 13-10 中所示。

表 13-10 binascii 模块中的异常

异 常	说 明
Error	异常，一个为普通错误所引发的异常
Incomplete	异常，在存在编码不完整数据来结束解码时引发的异常

13.6.3 binhex

binhex 格式已经在 Mac 机上使用很长一段时间了，用于对文件进行编码和解码，原因是 binhex 模块对源码和给定文件的数据分支两者进行编码。在 Mac OS 系统之外的其他平台上，binhex 模块仅仅只对数据分支进行编码/解码。此模块所支持的函数列于表 13-11 中。

表 13-11 binhex 格式所支持的函数

函 数	说 明
binhex(input, output)	对二进制文件或 input 文件对象进行编码，编码结果存到此文件或 output 文件对象之中
hexbin(input [, output])	对 binhex 格式文件或 input 文件对象进行解码，解码结果存到此文件或文件对象 output 之中。如果没给出 output 参数的值，其文件名就由 binhex 格式文件来确定

13.6.4 mailcap

mailcap 模块提供用于读出 mailcap 文件中数据的函数(请参见表 13-12)。mailcap 文件通过比较文件扩展名和 MIME 类型是否和已知应用程序和操作程序匹配来告知电子邮件阅读器和 Web 浏览器如何处理不同文件类型的不同文件。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

表 13-12　mailcap 模块所支持的函数

函　　数	说　　明
getcaps()	读出所有有效的 mailcap 文件，并且返回一个字典。此字典映射 MIME 类型到一个 mailcap 入口上。在默认情况下，mailcap 文件从 $HOME/.mailcap、/etc/mailcap、/usr/etc/mailcap 和/usr/local/etc/mailcap 中读取
findmatch(caps, mimetype[, key [, filename [, plist]]])	搜索 caps 字典(如 getcaps()函数返回的字典)以查找与 mimetype 匹配的 MIME 类型。如果指定 key 参数的值，key 参数表示的是想要的行为；filename 参数是要处理的文件；plist 参数是关于操作程序的命名参数列表。此函数返回一个元组，此元组包含 mailcap 文件的命令和原始 mailcap 入口

13.6.5　mimetools

mimetools 模块提供用于操纵以 MIME 格式编码的消息的函数。MIME(多用途国际因特网电子邮件扩展)是一个被认可的在国际因特网电子邮件上发送多个文件的标准。MIME 还用于多部件请求和 Web 服务器的应答。关于 mimetools 模块所支持的函数，请见表 13-13。

表 13-13　mimetools 模块所支持的函数

函　　数	说　　明
Message(file [, seekable])	分析作为电子邮件消息的 file，返回一个 Message 对象。Message 对象本身是从 rfc822.Message 类派生的。seekable 参数指定 file 中的文件对象是个可搜索对象(如普通文件)还是个不可搜索对象(网络套接字)。请参见关于所支持方法清单的表 13-14
choose_boundary()	创建一个惟一字符串，此字符串能用做 MIME 文件中的部分边界。返回字符串的格式与 hostipaddr.uid.pid.timestamp.random 匹配
decode(input, output, encoding)	对 input 中的 MIME 数据进行解码，并且把解了码的数据写入到 output 中。其中，input 和 output 参数应是先前已打开的文件对象；encoding 参数应是用于对数据进行解码的编码方法，应当是"base64"、"quotedprintable"及"uuencode"中的一种方法
encode(input, output, encoding)	对 input 中的文件进行编码，并且把一个 MIME 格式化了的文件写入到 output 中。其中，input 和 output 参数应是先前已打开的文件对象；encoding 参数应是用于对数据进行解码的编码方法，应当是"base64"、"quoted printable"及"uuencode"中的一种方法
copyliteral(input, output)	从文件对象 input 中读出行，并且把这些行写入到文件对象 output 中，而不进行任何转换
copybinary(input, ouput)	从文件对象 input 中读出二进制数据块，并且把这些数据块写入到文件对象 output 中，而不进行任何转换

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

Messsage 对象(此模块的 Message()类返回的对象)拥有表 13-14 中所示的方法。

表 13-14 mimetools.Message 类所支持的方法

方 法	说 明
getplist()	返回消息的内容类型头中的参数(以 key=value 的格式)。请注意，仅仅是参数被返回，而不是实际的 MIME 类型。还请注意，返回的格式是字符串列表，不是字典，并且把 key 转换为小写
getparam(name)	返回内容类型头中最后一个参数的值
getencoding()	返回内容传输编码消息头中所指定的编码格式，如果这样的消息头不存在，则返回"7bit"
gettype()	返回内容类型头的消息类型。默认情况下，返回"text/plain"
getmaintype()	返回内容类型头的主类型。默认情况下，返回"text "
getsubtype()	返回内容类型头的子类型。默认情况下，返回" plain"

13.6.6 mimetypes

mimetypes 模块提供的函数力图根据 MIME 类型识别文件的类型和扩展。mimetypes 模块所支持的函数如表 13-15 所示。

表 13-15 mimetypes 模块所支持的函数/对象

函数/对象	说 明
guess_type(filename)	通过检查文件名来猜测此文件的 MIME 类型
guess_extension(type)	通过识别 MIME type 来猜测此文件的扩展
init([files])	通过读取 mimetype: extensions 格式的文件信息初始化 extension/MIME 类型表。扩展应是用空格字符分开的扩展列表，其第一个字符无需是句点字符。如，text/html:htm html
read_mime_types(filename)	读取 filename 的 MIME 类型
knownfiles	返回在默认情况下被 init()函数使用的 MIME 文件的公共名字的列表
suffix_map	返回一个把扩展名映射为相匹配扩展名的字典。如，".htm"应被映射为".html"
encodings_map	返回一个把文件名扩展映射为编码类型的字典
types_map	返回一个把编码类型映射为文件名扩展的字典

13.6.7 MimeWriter

MimeWriter 模块定义一个类，就是 MimeWriter，此类能用来生成 MIME 编码文件。要创建一个新对象，请使用如下所示的命令：

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

MimeWriter(file)

其中，file 是个已经存在的文件对象，输出就要写入其中。另一个选择是，使用 StringIO 对象。最终的结果对象具有表 13-16 所列出的方法。

表 13-16　MimeWriter 类所支持的方法

方　法	说　明
addheader(key, value [, prefix])	以"key:value"的格式向文件里添加一个头。其中，prefix 参数定义头应插入到的位置。如果 prefix 参数值为零(默认时)，要插入的头则插入到已经存在头的最末尾处；如果 prefix 参数值为 1，要插入的头则插入到已经存在头的最前端处
flushheaders()	把所有头写入到文件里
startbody(ctype [, plist [, prefix]])	返回一个类似于文件的对象，此对象能用于把信息插入到最终文件里。其中，ctype 参数指定内容类型；plist 参数列出关于文件内容的参数；而 prefix 参数与 addheader()函数对应参数的意义相同，惟一不同之处是其默认取值为 1
startmultipartbody(subtype [,boundary [, plist [, prefix]]])	识别多部件体元素的起点。返回一个类似于文件的对象，此对象能用来写入消息内容。其中，subtype 参数定义多部件子类型，如混合类型(mixed)。boundary 参数用作文件之间的分隔符；plist 参数是关于文件的一个参数列表；而 prefix 参数与 addheader()函数对应参数的意义完全相同
nextpart()	返回新实例 MimeWriter，它表示多部件消息的一个独立部分
lastpart()	识别多部件消息的最后部分

13.6.8　multifile

multifile 模块定义一个类，就是 MultiFile 类。MultiFile 类能用于读取像文本消息这样的多分文件。能利用如下所示的命令创建一个 MultiFile 类实例：

MultiFile(file [, seekable])

其中，file 是个已经存在的文件对象，能从其中读取信息。seekable 参数指定文件是可搜索的(普通文件)还是不可搜索的(网络套接字)。

关于类实例的方法列于表 13-17 中。

表 13-17　MultiFile 对象的方法/属性

方法/属性	说　明
push(str)	把边界字符串压入到阅读器中。此方法识别消息段的结束或消息的结束。能压入多个边界标记符。但是压入一个不是最近读取的边界值就会引发错误
readline()	读出文件里的一行文本。如果此行与最近压入的边界标记符匹配，就返回一个空字符串；如果边界标记符与结束标记符一致，则 last 属性就设置为 1

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

（续表）

方法/属性	说　　明
readlines()	以字符串列表的形式返回当前部分的剩余行
read()	以单个字符串的形式返回当前部分的剩余行
next()	跳到下一个消息段
pop()	弹出一个消息段边界，这个消息段边界表明一个新文件段的起点
seek(pos [, whence])	移动到当前段的新位置。其中，pos 参数应以字节为单位指定的；而 whence 参数表明执行相对移动的基位置。whence 参数值为 0，就意味着从段起点开始移动；而 whence 参数值为 1，就意味着从当前位置开始移动；而 whence 参数值为 2，就意味着从段终点移动
tell()	返回当前位置
level	当前部分的嵌套深度
last	如果最后的文件结束标记是消息结束标记，则为真

13.6.9　quopri

quopri 模块把文本转换为 quoted-printable 格式，或对 quoted-printable 格式的文本进行转换。quoted-printable 格式用于对兼容格式的文本文件编码。有关此模块的函数，请参见表 13-18。

表 13-18　quopri 模块所支持的函数

函　　数	说　　明
decode(input, output)	对 quoted-printable 格式的 input 文件对象解码，解码结果写入到 output 文件对象中
encode(input, output, quotetabs)	把 input 文件对象的原始文本编码为 quoted-printable 格式，编码结果写入到 output 文件对象中。如果 quotetabs 参数为 1，则制表符被引用；如果 quotetabs 参数为零，则制表符作为标准包含在内

13.6.10　rfc822

rfc822 模块提供一个基于类的系统，此系统读/写电子邮件消息头的内容。一个典型电子邮件消息就像下面这样：

```
Return-Path: <cyrus@twinsol>
X-Sieve: cmu-sieve 2.0
Return-Path: <mc@mcslp.com>
Received: from punt-21.mail.demon.net (punt-21.mail.demon.net        [194.217.242.6])
    by mcslp.pri (8.11.2/8.11.2) with SMTP id f6VHW7Z21877
    for <mcslp@prluk.demon.co.uk>; Tue, 31 Jul 2001 18:32:07 +0100 (BST)
Received: from punt-2.mail.demon.net by mailstore for mcslp@prluk.demon.co.uk
            id 996600381:20:25406:0; Tue, 31 Jul 2001 17:26:21 GMT
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
Received: from smaug.dreamhost.com ([216.240.148.26]) by punt-2.mail.demon.net
                id aa2025334; 31 Jul 2001 17:26 GMT
Received: from mcslp.pri (prluk.demon.co.uk [158.152.8.40])
          by smaug.dreamhost.com (8.12.0.Beta7/8.12.0.Beta7/Debian
8.12.0.Beta7-1) with ESMTP id f6VHPcKJ007697
          for <mc@mcslp.com>; Tue, 31 Jul 2001 10:25:38 -0700
Received: from [192.168.1.129] (atuin.mcslp.pri [192.168.1.129])
      by mcslp.pri (8.11.2/8.11.2) with ESMTP id f6VHPZZ21840
      for <mc@mcslp.com>; Tue, 31 Jul 2001 18:25:35 +0100 (BST)
User-Agent: Microsoft-Entourage/9.0.1.3108
Date: Tue, 31 Jul 2001 18:25:32 +0100
Subject: Reminder
From: Martin C Brown <mc@mcslp.com>
To: Martin C Brown <mc@mcslp.com>
Message-ID: <B78CA89C.32ACC%mc@mcslp.com>
Mime-version: 1.0
Content-type: text/plain; charset="US-ASCII"
Content-transfer-encoding: 7bit
Just a reminder to make that meeting...
```

从开始一直到第一个空行为止都被作为头来对待。其中，头的格式是"field: value"。例如，消息的主题被定义为"Subject: Reminder"。能通过创建 Message 类的一个新实例分析消息头，如下所示：

```
Message(file [, seekable])
```

其中，file 参数是个已经存在的文件对象，能从此文件对象读取信息；seekable 参数指定文件是可搜索的(普通文件)还是不可搜索的(网络套接字)。

关于此模块的行为有一个范例，请参见本章前面的“使用 SMTP”部分。最终结果对象实例的行为表现就像个字典一样，暂且不论注意事项，把附加方法列于表 13-19 中。

表 13-19　Message 类所支持的方法

方法 / 属性	说　明
m[name]	返回 name 头的值
m[name]=value	设置 name 头的值
m.keys()	列出消息中的所有头域
m.values()	列出消息中的所有头值
m.items()	返回一个两个元素元组的列表。其中每一个元组包含头字段名和值
m.has_key(name)	消息具有 name 头，则返回真
m.get(name [, default])	获得 name 头字段的值，而在头不能找到的时候且指定了 default 参数的值，就返回 default 参数值
len(m)	返回头的数目

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

方法 / 属性	说 明
str(m)	返回头的字符串表示。此头应是有效的，像兼容 RFC822 消息头一样
m.getallmatchingheaders (name)	返回与 name 匹配的所有头
m.getfirstmatchingheader (name)	返回与 name 匹配的第一个头
m.getrawheader(name)	返回一个原始字符串，此字符串源自于 name 字段的一个头行
m.getheader(name [,default])	像 getrawheader()一样，但是此方法删除所有的前置和后缀空格字符
m.getaddr(name)	返回一个元组。此元组由名字和电子邮件地址构成。其中，电子邮件地址源自于与 name 匹配的一个地址头
m.getaddrlist(name)	返回一个元组列表。其中，每一个元组包含与 name 匹配的头的地址
m.getdate(name)	返回一个 9 元素元组，此元组是关于在 name 头中找到的数据的
m.getdate_tz(name)	返回一个 9 元素元组。此元组包括偏差，偏差是 UTC 时区对于 name 头中找到的日期的偏差
m.headers	返回一个列表。此列表包含所有头行
m.fp	返回在对象实例创建时使用的像文件一样的对象

rfc822 模块所支持的其他函数列于表 13-20 中。

表 13-20 rfc822 模块所支持的其他函数

函 数	说 明
Parsedate(date)	分析诸如"Tue, 31 Jul 2001 18:34:03 +0000"的 RFC822 格式化了的日期，而返回一个包含已分析数据的 9 元素元组。此元组与 time.mktime()兼容
Parsedate_tz(date)	分析诸如"Tue, 31 Jul 2001 18:34:03 +0000"的 RFC822 格式化了的日期，而返回一个包含已分析数据的十元素元组。此表列包含对于 UTC 的时区偏差。此元组与 time.mktime()兼容
mktime_tz(tuple)	把十元素元组转换回为 UTC 时戳
AddressList(addrlist)	把包含一列电子邮件地址的字符串转换为一个 AddressList 对象。此新对象支持两个方法，即 len()和 str()方法。这两个方法分别返回对象的地址数和对象的字符串表示。+和–运算符则对两个 AddressList 对象的独立地址进行加和减

13.6.11 uu

uu 模块把文件编码为 UUCP uuencode 格式和 / 或把文件解码为 UUCP uuencode 格式。uu 模块所支持的函数如表 13-21 所示。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

表 13-21　uu 模块所支持的函数

函　　数	说　　明
encode(input, output [,name [, mode]])	把文件名或 input 文件对象编码为 uuencoded 格式，结果为此文件名或 output 文件对象。其中，name 参数应当是用来表示文件的名字；而 mode 参数应是文件在解码时所用文件的模式(八进制)
decode(input [, output [,mode]])	对 input(文件名或文件对象)中的 uuencoded 格式数据进行解码。输出名和模式由文件自动确定，或者被 output 参数和 mode 参数所重写

13.6.12　xdrlib

xdrlib 模块提供一整套函数，把数据转换为 Sun 的 XDR (外部数据表示)格式，或对 XDR 格式的数据进行逆转换。XDR 用在 RPC(远程过程调用)系统中，用于对机器 / 网络中性格式的数据进行编码，当然假定系统支持 XDR 和 RPC 函数库，则 XDR 也适用于系统之间交换信息。有关此模块的更多信息，请参见再线文档。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第14章　多媒体中使用Python

Python 可能看起来不像是多媒体开发所使用语言的最佳选择。但实际上，Python 是最适合于多媒体开发这项任务。Python 的强大数学能力，加之其面向对象和类型化的特性，使 Python 成为关于所有类型的多媒体数据的一个好选择。

本章，就将讲述 Python 标准配置包含的几个模块，这些模块能用来操纵音频文件和图像文件。

14.1　音频模块

标准函数库含有一些模块，这些模块能读写不同的音频格式。除此之外，标准函数库还有各种各样操纵声音信息的函数。

本书所讨论的所有音频格式，并且事实上，大多数普通音频格式都是基于一个相同的基础之上的。通过以特定时间为间隔记录声音的频率和音量，能把音频编码成为数字格式。为了表示一段声音，每个时间周期要记录两个数值。每一对数值，即频率和音量，称作一个取样。每个取样的长度和每秒钟取样的数目有助于确立音频的创建质量。比如说，一秒钟一次的取样频率就比用一秒钟表示一个单音要好得多，现在想像一下这样的旋律将是多么不堪忍受！通常情况下，声音在一秒钟里要被取样上千次。例如，音乐 CD 是以每秒钟 44100 次取样频率记录的。还将涉及的其他术语包括：

- Sampling rate(取样频率)或 frame rate(帧频) 是每秒钟对声音进行的取样次数。如，CD 音频每一秒钟就进行 441000 次取样，或者说 44.1 KHz。取样次数越多，声音的质量就越好。
- Bits per sample(位数 / 取样)是用于量化已经数字化了的声音的位数。CD 量化音频使用 16 位取样，而数字电话系统则使用 8 位取样。每个取样所用的位数越多，声音输出质量就越好。
- 音频中存储有一到多个 channel(声道)。立体声使用两个声道，单声道声音使用一个声道。有一些格式允许同时提供多个声道。这些格式适用于多声道数据系统或环绕声系统。多声道数据系统或环绕声系统使用 4、5.1 或 8.1 声道。如，DVD 格式，典型地使用 5.1 声道(左前方声道/右前方声道、左后方声道/右后方声道、中心声道和次低音声道)。
- frame(帧)是个单一数据块，此数据块用于存储一个取样数据。帧的尺寸能通过每个取样的位数和通道数相乘计算得到。例如，CD 音频使用 2 个声道和 16 位取样，因此，每一帧的尺寸大小为 4 个字节。

因而，能计算给出某些信息片段的特定文件的不同信息片段。如下所示：

- 假如给定了文件中帧的数目和取样频率，能利用如下所示的公式计算文件的时间长度：

  Duration = noofframes/samplerate

- 假如给定了帧数、取样频率、每个取样的位数和声道数，能利用如下所示的公式计算一

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

个文件里的音频单元的尺寸：

```
filesize = noofframes*samplerate*channels*(bitspersample/8)
```

● 最后，要计算传输音频所要求的数据频率，需要知道声道数、每个取样的位数和取样频率。计算公式如下所示：

```
datarate = samplerate*channels*(bitspersample/8)
```

14.1.1　sndhdr

要操纵任何种类的声音文件，应当使用其他模块。但是如果想知道某个特定文件使用的是哪种声音格式，除了通过识别文件的扩展之外，还可以使用 sndhdr 模块。

what()和 whathdr()函数执行相同的操作，都返回一个元组，在这个返回的元组里，包含着关于某个具体文件的信息。例如，下面的脚本：

```
import sys, sndhdr
print sndhdr.what(sys.argv[1])
```

将输出元组，此元组包含返回的关于命令行给定文件的信息。如果给其提供以 CD 唱片所记录的一个 AIFF 立体声文件，将得到如下所示的结果：

```
('aiff', 44100, 2, 10248840, 16)
```

请参见表 14-1 和表 14-2。表 14-1 是个有关返回元素的清单；而表 14-2 是个所支持音频格式的清单。

表 14-1　imghdr.what()和 imghdr.whathdr()返回的元素

元组元素	说　明
Type	从文件中识别出来的数据类型。请参见表 14-2。表 14-2 是返回格式的清单
samplingrate	以赫兹为单位的取样频率(每秒钟取样的次数)。如果取样频率不能被识别出来或难以确定，就可能返回 0
channels	声道数。如果是立体声，就为 2；如果是单声道，就为 1
frames	帧(数据块)数。如果帧数不能确定或者不能用指定的文件类型存储，则可能是-1
bitspersample	每一个取样的位数。或者是，对应于“A-LAW”格式的“A”，对应于“u-LAW”格式的“U”

表 14-2　sndhdr 所支持的声音格式

格　式	说　明
aifc	压缩的 AIFF 格式
aiff	音频互换文件格式(Audio Interchange File Format)
au	NeXT/Sun au 格式
hcom	Macintosh HCOM 格式
sndr	Amiga 8 位格式

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

格　式	说　明
sndt	Amiga 8 位格式
voc	Creative(SoundBlaster 卡制造厂商)声音格式
wav	Windows WAVE 格式
8svx	Amiga 声音取样格式，AIFF 格式的修订版
sb	纯有符号 8 位格式
ub	纯无符号 8 位格式
ul	US8 位格式——实际上使用每秒钟 8 000 个取样(用于电话)

14.1.2　aifc

aifc 模块允许读写 AIFF 格式和压缩 AIFC 格式的音频文件。open()函数打开一个音频文件(包括在需要时创建一个新文件)，其格式与标准 open()函数的格式相同，如下所示：

```
open(file, mode)
```

其中，file 参数应是要打开的或要创建的文件名，或者是给像文件一样的对象。mode 参数应取“r”、“rb”、“w”、“wb”之中的一个值。当 mode 参数值为“r”或“rb”时，就从一个已经存在的文件往外读取；而当 mode 参数值为“w”或“wb”的时候，就往文件里写入或创建一个新文件。例如，要打开一个用于读的文件，命令如下所示：

```
myaiff = open('myaiff.aiff', 'rb')
```

一旦以只读模式打开，就能对返回的对象使用表 14-3 所示的方法。

表 14-3　AIFF 文件的读取方法

方　法	说　明
getnchannels()	返回文件的音频声道数
getsampwidth()	返回以字节为单位的取样频率
getnframes()	返回音频流的帧数
getframerate()	返回帧频——每秒钟取样数
getcomptype()	获取压缩类型，此方法用于压缩 AIFF-C 或 AIFC 格式文件的数据
getcompname()	获取压缩类型的文本描述，此方法用于压缩 AIFF-C 或 AIFC 格式文件中的数据
getparams()	获取文件的参数。返回一个元组，元组的形式为(声道，取样宽度,帧数,帧频,压缩类型,压缩名)
getmarkers()	获取音频文件制造商的列表。返回一个元组列表，其中每个元组包含 3 个元素，分别是制造厂商标识符、制造厂商位置和制造厂商名字
getmark(id)	返回关于制造厂商 id 的元组
readframes(nframes)	从文件当前位置开始读出 nframes

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

方　法	说　明
rewind()	重新定位到文件的开始处
setpos(pos)	把文件的当前位置定位到 pos 处。请注意，pos 参数应当是以帧为单位指定的，而不应以字节为单位
tell()	返回从音频流的开始处计起的当前位置(以帧为单位)
close()	关闭文件

如果已经打开用于写入的文件，就能使用表 14-4 所示的方法。

如果试图把这些方法用到一个未打开的文件上，则将引发异常。

表 14-4　写 AIFF 文件数据所支持的方法

方　法	说　明
aiff()	创建一个 AIFF 文件。默认操作是创建一个 AIFF-C(或 AIFC)文件；除非 AIFF 文件在创建的时候以.aiff 为结尾
aifc()	创建一个 AIFF-C 文件。默认操作是创建一个 AIFF-C(或 AIFC)文件；除非 AIFF 文件在创建的时候以.aiff 为结尾
setnchannels(channels)	指定文件的声道数
setsampwidth(width)	指定文件的取样频率大小
setframerate(rate)	指定以帧为单位的每秒钟取样频率
setnframes(nframes)	指定要被写入到音频文件里的帧数
setcomptype(type, name)	指定用于压缩音频的压缩类型，仅仅只适用于 AIFF-C 文件。其中，name 参数应当是个关于压缩格式的可读性强的描述；而 type 参数应是由 4 个字符组成的字符串。此模块当前支持的 type 参数取值是“None”、“ULAW”、“ALAW”或“G722”
setparams(nchannels,sampwidth, framerate,comptype, compname)	一次设置文件的全部参数
setmark(id, pos, name)	在 pos 位置(tell()所返回的值)处利用 name 名字添加一个标识符 id 标记
tell()	返回文件的当前位置(以帧为单位)
writeframes(data)	把 data 中的帧写到文件里去。仅仅在设置给定文件参数时，此函数才支持
writeframesraw(data)	等价于 writeframes()，惟一不同之处在于此方法不更新文件头，不反映写到文件里附加帧这个事实
close()	关闭文件并更新文件头，以反映文件的真实尺寸

为了创建要写出去的帧，需要创建一个正弦波，这个正弦波是通过把频率和取样频率利用到几个帧周期上计算得来的。接下来，把计算得到的原始值转换为 2 字节值(对于每个取样音频

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

来说，就是 16 位)。如果是由音调求出的，而不是由频率求得的，就能使用表 14-5 当中的说明根据给定的音度和音调来求出频率。能利用 aifc 模块把这所有这些组合起来创建一个音频文件。下面就是根据著名 sci-fi 电影创建一个声音序列的例子。

表 14-5　音度、音调和相对应的频率

音调/音度	C,,	C,	C	C	C'	C''	C'''	C''''
C	16.35	32.70	65.41	130.8	261.7	523.3	1046.6	2093.2
C#	17.32	34.65	69.30	138.6	277.2	554.4	1108.8	2217.7
D	18.35	36.71	73.42	146.8	293.7	587.4	1174.8	2349.6
D#	19.44	38.89	77.79	155.6	311.2	622.3	1244.6	2489.3
E	20.60	41.20	82.41	164.8	329.7	659.3	1318.6	2637.3
F	21.82	43.65	87.31	174.6	349.2	698.5	1397.0	2794.0
F#	23.12	46.25	92.50	185.0	370.0	740.0	1480.0	2960.1
G	24.50	49.00	98.00	196.0	392.0	784.0	1568.0	3136.0
G#	25.95	51.91	103.8	207.6	415.3	830.6	1661.2	3322.5
A	27.50	55.00	110.0	220.0	440.0	880.0	1760.0	3520.0
A#	29.13	58.27	116.5	233.1	466.2	932.3	1864.6	3729.2
B	30.86	61.73	123.5	246.9	493.9	987.7	1975.5	3951.0

```
import aifc
import math

def write_note(freq, length, sample_rate, bitspersample, aifcfile):
    reallength = length*sample_rate
    max = (2**bitspersample) / 2
    pos = 0
    notedata = ''
    while (pos <= reallength):
        note = int(math.sin(
            math.pi*(float(pos)/
                (sample_rate/freq)))*max)
        notedata += chr((note >> 8) & 255) + chr(note & 255)
        pos += 1
    aifcfile.writeframesraw(notedata)

sample_rate = 44100
bitspersample = 16
channels = 1

aifcfile = aifc.open('ceottk.aiff','wb')
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
aifcfile.setframerate(sample_rate)
aifcfile.setsampwidth((bitspersample/8))
aifcfile.setnchannels(1)

write_note(493.9, 0.5, sample_rate, bitspersample, aifcfile)
write_note(261.7, 0.5, sample_rate, bitspersample, aifcfile)
write_note(440, 0.5, sample_rate, bitspersample, aifcfile)
write_note(220, 0.5, sample_rate, bitspersample, aifcfile)
write_note(261.7, 0.5, sample_rate, bitspersample, aifcfile)

aifcfile.close()
```

关于电影标题的线索就在执行时所创建的 ceottk.aiff 文件名里面。

14.1.3　audioop

audioop 模块是个普通工具，此工具用于在不同声音流所包含的帧之上进行操作。其中，不同的声音流是通过字符串表示的，而不同声音流是利用 8、16 或 32 位宽度取样的。audioop 模块所使用的格式与 al 和 sunaudiodev 模块所使用的格式一样。audioop 模块所提供的函数支持 u-LAW 和 Intel/DVI ADPCM 编码。在使用此模块的函数之前需要对 AIFF/AIFC 或 MP3 格式进行转换。

此模块所支持的函数清单如表 14-6 所示。请注意，几乎所有的函数都需要对其提供以数据流的取样宽度，这个取样宽度是以字节为单位，而不是以位为单位。

表 14-6　audioop 模块所支持的函数

函　　数	说　　明
add(fragment1, fragment2,width)	把 fragment1 声音段添加到 fragment2 声音段末尾处，返回一个单个声音段。这两个声音段的长度和宽度都应当相同，宽度为 width
adpcm2lin(adpcmfragment,width, state)	把 adpcmfragment 声音段里的 ADPCM 声音段转换为一个线性声音段，此线性声音段适合于用此模块中的例程操纵。返回一个元组，此元组包含取样数据流和返回码新的 state
adpcm32lin(adpcmfragment,width, state)	等价于 adpcm2lin()，但是此函数在 3 位 ADPCM 码上操作
avg(fragment, width)	返回声音段中所有取样的平均取样水平
avgpp(fragment, width)	返回声音段中所有取样的平均峰到峰水平
bias(fragment, width, bias)	返回对每个取样添加了 bias 的声音段 fragment
cross(fragment, width)	返回 fragment 中零相交的个数
findfactor(fragment,reference)	返回一个因子，此因子能使 fragment 声音段尽可能地逼近于 reference 声音段。两个声音段都应使用 16 位取样
findfit(fragment, reference)	努力找到 fragment 声音段中与 reference 声音段最匹配的那部分。返回一个元组。返回的元组包含 fragment 声音段中的位置偏移量和要求的因子

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

（续表）

函　　数	说　　明
findmax(fragment, length)	找到音频流 fragment 所包含能量(或音量)最高的那部分，其长度为 length 参数给定的数值。返回在此声音段中第一个取样的位置
getsample(fragment, width,index)	返回 fragment 段中索引为 index 取样的值
lin2lin(fragment, width,newwidth)	把 fragment 声音段从取样宽度为 width 转换为取样宽度为 newwidth。用于在不同的取样宽度间对声音段进行转换
lin2adpcm(fragment, width,state)	把取样转换为 4 位 Intel/DVI ADPCM 编码。其中，state 参数应当是个元组。此元组包含编码器的状态。最初传递给 state 参数的值应是 None。此函数的返回值是个元组。返回的元组包含 ADPCM 段和编码器的新状态
lin2adpcm3(fragment, width,state)	等价于 lin2adpcm(fragment, width, state)。惟一不同之处在于，此函数转换为 3 位而不是转换为 4 位 ADPCM 编码
lin2ulaw(fragment, width)	把 fragment 声音段的取样转换为 u-LAW 编码格式，并且以 Python 字符串的形式返回转换结果
minmax(fragment, width)	返回一个元组。返回的元组包含 fragment 声音段中最大值和小值在 fragment 声音段中的位置
max(fragment, width)	返回 fragment 声音段中所有数值的最大值
maxpp(fragment, width)	返回 fragment 声音段中的最大峰到峰值
mul(fragment, width, factor)	返回 fragment 声音段，其中 fragment 声音段中的每一个取样都被乘上一个因子 factor
ratecv(fragment, width,nchannels, inrate, outrate,start [, weighta [, weightb]])	把 fragment 声音段从帧频为 inrate 转换为帧频为 outrate。返回一个元组。返回的元组包含新的段和新状态。其中，weightA 参数和 weightB 参数是关于一个简单数字滤波器的参数，并且默认被分别设置为 0 和 1
reverse(fragment, width)	返回 fragment 声音段的所有取样，其中 fragment 声音段中所有取样的顺序与原来的顺序正好相反
rms(fragment, width)	返回一个段的均方根。这是对音频信号输出功率的一个度量(RMS 常常用于描述扩音器的输出)
tomono(fragment, width,lfactor, rfactor)	把立体声(两声道)声音段转换为单声道(单个声道)声音段。其中，lfactor 参数和 rfactor 参数是用来乘以总输出左右声道的因子。例如，tomono(fragment, width, 1, 1)将产生一个平衡的单声道段，此单声道声音段只包含立体声段的右侧声道，tomono(fragment, width, 0, 1)将返回单声道段
tostereo(fragment, width,lfactor, rfactor)	把单声道声音段转换为立体声声音段。其中，为把 lfactor 和 rfactor 应用于立体声最终输出的左右声道上，lfactor 参数和 rfactor 参数用作单声道总输出上的乘数

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

函　数	说　明
ulaw2lin(fragment, width)	把 U-LAW 格式的声音段转换为线性编码格式的声音段

14.1.4　chunk

chunk 模块读取 IFF 块中存储的数据，如 AIFF、AIFF-C 和 WAV 格式所使用的一样。IFF 块利用可变长度记录存储信息。此格式详细地列于表 14-7 中。

表 14-7　IFF 块格式

位　移	长　度	内　容
0	4	Chunk 标识符
4	4	Chunk 尺寸(以 big-endian 顺序)
8	n	Chunk 数据，其中 n 是存储在前面段的块尺寸
8+n	0 或 1	n 是个奇数时所需的和块对齐方式所使用的填充字节

要使用此模块，需要创建一个 Chunk 类的新实例，此新实例读取一给定文件对象的信息，就好像数据存储在 IFF 块中一样。创建一个新块的格式如下所示：

Chunk(file [, align, bigendian, inclheader])

其中 file 参数应是个类似文件对象，如前面通过 open()内置函数创建的对象一样。选项参数 align 指定块是否对齐到 2 字节边界上。如果 align 参数值为真，则块被对齐到 2 字节边界上(为此，要启用表 14-7 所示的填充字节)；而如果 align 参数值为假，则块就不需对齐；align 参数的默认取值是真。如果 bigendian 参数是假，就假定块尺寸不是 big-endian 顺序；bigendian 参数的默认取值是真。如果 inclheader 参数是真，块头中给出的尺寸数值就包括头本身的长度，此参数的默认取值是假。

最终得到的 Chunk 对象支持表 14-8 中所列方法。请注意，如果在 close()之后调用 getname()和 getsize()之外的方法将引发 error 异常。

表 14-8　Chunk 对象所支持的方法

方　法	说　明
getname()	返回块的标识符，就是块头文件中所存储的头 4 个字节
getsize()	以字节为单位，返回块的尺寸大小
close()	关闭并跳到块尾处，而不关闭底层文件
isatty()	返回 0
seek(pos[, whence])	设置块的当前位置。其中，pos 参数应是以字节为单位指定的位置；而 whence 参数，如果给定其值，就使用同底层文件对象一样的值(值为 0，就是绝对地址；值为 1，就是相对地址；值为 2，就是相对于尾部的相对地址)

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

方　法	说　明
tell()	返回设置在块中的当前位置
read([size])	从块中读出最多为 size 个字节。如果未给出 read 参数的值，则读出一直到块尾的全部字节。块数据以 Python 字符串的形式返回
skip()	跳到块尾处。接下来的 read()调用将返回一个空字符串

请注意，因为 IFF 块可能以多个块的形式存储在文件里，所以需要先打开文件并读出所有格式头文件(如，AIFF 格式所使用的头文件)，一直读到第一个块的指针为止，接下来再顺序调用来获得存储在给定文件里的所有 IFF 数据。即创建新的 Chunk 实例，接着调用 read()方法。举个例子看看 aifc 模块实际是怎样工作的。

14.1.5　sunau

sunau 模块提供一个接口。此接口是到 Sun AU 音频格式的接口，是个与 aifc 模块和 wave 模块兼容的接口。

所有 Sun AU 文件都具有表 14-9 中所示格式的文件头。关于文件头中的编码域和 magic word 域，可以使用表 14-10 中所列的常量。

表 14-9　Sun AU 文件头的格式

域	说　明
magic word	Sun AU 文件的 magic 头文件
header size	以字节为单位头文件的尺寸大小
data size	以字节为单位数据的尺寸大小
encoding	用在数据上的编码格式
sample rate	取样频率
# of channels	声道数
info	音频文件的说明

表 14-10　sunau 模块所定义的常量

常　量	说　明
AUDIO_FILE_MAGIC	整数，每一个 Sun AU 格式的文件都以此整数开始。此整数实际格式上是字符串.snd 的整数形式
AUDIO_FILE_ENCODING_MULAW_8 AUDIO_FILE_ENCODING_LINEAR_8 AUDIO_FILE_ENCODING_LINEAR_16 AUDIO_FILE_ENCODING_LINEAR_24 AUDIO_FILE_ENCODING_LINEAR_32 AUDIO_FILE_ENCODING_ALAW_8	此模块所支持的编码域的数值

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

常　量	说　明
AUDIO_FILE_ENCODING_FLOAT AUDIO_FILE_ENCODING_DOUBLE AUDIO_FILE_ENCODING_ADPCM_G721 AUDIO_FILE_ENCODING_ADPCM_G722 AUDIO_FILE_ENCODING_ADPCM_G723_3 AUDIO_FILE_ENCODING_ADPCM_G723_5	不为此模块所支持的编码域的数值

操作的基本模式是等价的：必须先打开一个已经存在的文件或者通过 open()函数创建一个新文件。这个已打开的对象支持与 aifc 模块和 wave 模块所支持方法相等价的方法。但是额外还要注意以下事项：

- readframes()方法返回一个线性格式的音频数据的字符串。u-LAW 格式的数据在返回过程中就被转换了。
- getmarkers()方法返回 None，而如果调用 getmark()方法则引发一个异常。

有关所支持方法的清单，请参见表 14-11 和表 14-12。

表 14-11　wave 对象的读方法

方　法	说　明
close()	关闭文件流。下一步对此文件上方法的任何调用将引发异常
getnchannels()	返回音频文件的声道数
getsampwidth()	返回每一个取样的宽度
getframerate()	返回帧频，帧频以每秒多少帧为单位
getnframes()	返回文件中全部音频帧的数目
getcomptype()	返回压缩类型。当前仅支持 None
getcompname()	返回具有可读性的压缩名。当前仅返回 None
getparams()	以元组的形式返回关于文件中音频的参数，返回的元组依次包含声道数、取样宽度、帧频、帧数、压缩类型和压缩名
readframes(n)	从文件里读出至多 n 个字节，并以字符串的形式返回读出的结果
rewind()	把文件指针重新定位到音频流的开始位置上
getmarkers()	返回 None。为兼容 aifc 模块而保留
getmark(id)	引发一个错误。为兼容 aifc 模块而保留
setpos(pos)	把文件指针定位到某个具体位置上
tell()	返回当前文件指针的位置

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

表 14-12　Wave 对象的写方法

方　法	说　明
close()	关闭文件并更新文件头中的帧信息。在一个 Wave 对象失效时，此方法被自动调用
setnchannels(n)	设置声道数
setsampwidth(n)	设置取样宽度为 n 字节
setframerate(n)	设置帧频为每秒钟 n 帧
setnframes(n)	设置文件中帧数为 n。在帧被写入的时候，自动更新
setcomptype(type, name)	设置压缩类型和描述
setparams(tuple)	一次设置关于文件的所有参数。给定的元组应依次包含声道数、取样宽度、帧频、帧数、压缩类型和压缩名
tell()	返回文件中的当前位置
writeframesraw(data)	把 data 中的帧写入到文件里，而不更正文件的帧数
writeframes(data)	把 data 中的帧写入到文件里，并更新整个文件的帧数

14.1.6　wave

wave 模块允许读写 Windows WAVE (WAV)文件。这儿所用的基本方法与本章前面部分论述的 aifc 模块所使用的方法等价。

用于打开一个已经存在的或需要新创建的 WAV 文件的基本方法是：open()函数。open()函数的格式如下所示：

open(file [, mode])

像 aifc 模块的 open()函数一样，file 参数应当是个已经存在的文件对象，或者是要打开或要创建的文件名。mode 参数定义打开模式，可取“r”、“rb”、“w”或“wb”。“r”或“rb”模式用于只读；而“w”或“wb”模式则用于写。

所有方法在发生错误的时候，都引发 wave.Error 异常。

在文件被打开用于只读时，打开对象所支持的方法列于表 14-11 中。

请注意，setpos()和 tell()方法返回的值仅仅对其自身兼容。不提倡使用返回的值，也不提倡把这些方法用在普通文件对象上。

其中，如果 wave 对象是打开用于写，wave 对象就能使用表 14-12 中所列方法。

关于怎样往文件里写入帧的例子，请参见本章前面 aifc 部分的例子，或者参见 Python 配置所带的 morse.py 例子。

14.2　图形模块

Python 支持少量的基本图像操纵和图像识别模块。这些模块中的大部分是为使用一些 SGI 的特定模块而开发的，而没有为本书囊括。总之，本书所论述的模块不是特定于某个平台的，因而能用在所有平台下。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

14.2.1　imghdr

imghdr 模块执行与 sndhdr 一样的操作，但不是在图像文件上。what()函数返回一个用来定义图像格式的单字符串。举例如下：

```
>>> import imghdr
>>> imghdr.what('mcslplogosm.gif')
'gif'
```

此模块识别的字符串和图像文件的完整清单如表 14-13 所示。

表 14-13　imghdr 模块所识别的文件格式

格　式	说　明
Rgb	SGI ImgLib 文件
Gif	GIF 87a 和 89a 文件
Pbm	Portable 位图文件
Pgm	Portable Graymap 文件
Ppm	Portable Pixmap 文件
Tiff	TIFF 文件
Rast	Sun Raster 文件
Xbm	X 位图文件
Jpeg	JFIF 格式的 JPEG 数据
Bmp	BMP 文件
Png	Portable 网络图像文件

14.2.2　colorsys

colorsys 模块允许在不同彩色空间定义间转换。

此模块允许在 RGB(红、绿、蓝)、YIQ(一个欧洲彩色空间格式)、HLS(色调、亮度、饱和度)和 HSV(色调、饱和度、值)格式间转换。

此模块所支持的函数清单如表 14-14 所示。

表 14-14　colorsys 模块中用于转换彩色空间值的函数

函　数	说　明
rgb_to_yiq(r, g, b)	把彩色值从 RGB 坐标系转换到 YIQ 坐标系下
yiq_to_rgb(y, i, q)	把彩色值从 YIQ 坐标系转换到 RGB 坐标系下
rgb_to_hls(r, g, b)	把彩色值从 RGB 坐标系转换到 HLS 坐标系下
hls_to_rgb(h, l, s)	把彩色值从 HLS 坐标系转换到 RGB 坐标系下
rgb_to_hsv(r, g, b)	把彩色值从 RGB 坐标系转换到 HSV 坐标系下
hsv_to_rgb(h, s, v)	把彩色值从 HSV 坐标系转换到 RGB 坐标系下

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

请注意，值应表示为 0 到 1 之间的一个浮点数。如果习惯于利用 8 位数值——0~255 范围的值——需要在将其提供给这些函数之前对此值进行转换，转换到 0 到 1 之间。

14.2.3　imageop

imageop 模块支持一些函数，用于操纵使用 8 或 32 位存储图像数据的图像。一旦有错误发生时，所有函数都返回 imageop.error 异常。imageop 模块所支持函数的清单如表 14-15 所示。

表 14-15　imageop 模块所支持的函数

函　　数	说　　明
crop(image, psize, width,height, x0, y0, x1, y1)	返回 image 图像或返回图像的大小为 psize 字节，高为 height，宽为 width。返回的图像是利用左下角(x0, y0)到右上角(x1, y1)定义的边界框剪切得到的。边界矩形包含于剪切的图像之内。如果 x0 比 x1 大，或者 y0 比 y1 大，图像就作垂直和/或水平镜像而返回
scale(image, psize, width,height, newwidth, newheight)	缩放(高为 height，宽为 width，大小为 psize 个字节的)image 图像为新的尺寸，宽为 newwidth 和高为 newheight。图像是通过复制或删除像素而进行缩放的，不进行内部填充
tovideo(image, psize, width,height)	过滤宽为 width，高为 height，大小为 psize 字节的 image 图像以适合于在隔行扫描的设备上输出
grey2mono(image, width,height, threshold)	把 8 位单色图像转换为 1 位单色图像
dither2mono(image, width,height)	利用抖动把 8 位单色图像转换为 1 位单色图像
mono2grey(image, width,height, p0, p1)	把 1 位单色图像转换为 8 位单色图像。p0 和 p1 两个参数分别用作 1 位单色图像像素值为 0 和 1 的转换取值。要得到一个灰度图像，请分别使用 0 和 255 这两个值
grey2grey4(image, width,height)	把 8 位灰度图像转换为 4 位灰度图像，不使用抖动
grey2grey2(image, width,height)	把 8 位灰度图像转换为 2 位灰度图像，不使用抖动
dither2grey2(image, width,height)	把 8 位灰度图像转换为 2 位灰度图像，利用抖动。就像关于 dither2mono()一样，目前的抖动算法都很简单
grey42grey(image, width,height)	把 4 位灰度图像转换为 8 位灰度图像
grey22grey(image, width,height)	把 2 位灰度图像转换为 8 位灰度图像

14.2.4　rgbimg

rgbimg 模块允许读写扩展名为.rgb 的 SGI imglib 文件。此函数仅仅只支持 4 字节 RGBA、3 字节 RGB 和 1 字节灰度图像。此模块支持的函数如表 14-16 所列。一旦发生错误，所有函数都引发 rgbimg.error 异常。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

表 14-16　rgbimg 模块所支持的函数

函　　数	说　　明
sizeofimage(file)	返回一个元组。返回的元组包含 file 中的图像的宽度和高度
longimagedata(file)	返回 file 的图像，此图像是以一个 Python 字符串的形式返回的。左下角像素是字符串的第一个元素，而且所有的像素都是以 4 字节 RGBA 格式表示
longstoimage(data,x, y, z, file)	把 data 中的 RGBA 图像写入到 file 文件里，所使用的宽度和高度分别为 x 和 y 参数所定义的值。而 z 参数则定义输出格式，如果为 1，就输出灰度图像；如果为 3，就输出 RGB 格式图像；如果为 4，就输出 RGBA 格式图像
ttob(flag)	设置此模块在对图像做处理时，是从最底行开始处理到最顶行结束(flag 参数为零，是默认设置)，还是从最顶行开始处理到最底行结束(flag 参数为 1)

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第15章　用Tk创建接口

要编写能实现想要功能的程序相对而言是简单的。但实际上要使程序好用,就要求创建一个合适的易于导航的用户接口。创建用户接口，有许多方法。对简单的实用程序而言，明显的解决方法是一组简单的命令行选项，这些选项使开发者能输入这样的命令：告知应用程序想要做的是什么。第 9 章已经讲述的 getopt 模块就是关于这方面的一个举例，此模块对命令行变量进行分析。

对更复杂的应用程序而言，特别是想在网络上和各种不同平台上使用共同的应用程序时，最好的解决方法是使用 HTML 和连接因特网网站或企业内部互联网网站的 CGI。本书第 19 章所关注的就是用 Python 制作网站的具体操作。

如果应用程序难度适中，不简单也不太复杂，那大概就要看看这个“桌面”解决方法。此方法是一种独立形式的应用程序，此应用程序不使用本地信息就使用从网络上许多远程源处收集而来的信息。

不论采用哪种方法，应用程序需要一个接口。此接口应当比较简单，比如说，一个允许数据记录到数据库中的良好形式的接口，或者更复杂些，如 Web 浏览器或办公应用程序。

不管应用程序多么复杂，都需要建立一个高效接口，此接口由窗口、按钮、文本输入框和其他部件组成，其他部件允许用户与之交互，并且允许用户通过相似的 GUI(图形用户接口)控制应用程序。自己是能设计和开发接口的，但是要兼容多个不同的系统就难了。甚至于获取对一个按钮单击这么简单的操作，应用程序代码也会变成复杂的一大序列。总之，非常有可能的是在已经具有 GUI 的平台上开发接口——那么需要知道的就仅仅是如何与那个 GUI 交互操作来开发自己的应用程序。

没有一个关于开发这些应用程序的标准工具包，但是使用 Tk 工具包是个好的选择。Tk 工具包大大简化了接口实现所需的复杂操作，而独立的设计又能很好地完成用户的要求。

Tk 工具包具有跨平台兼容特性。并且 Tk 工具包具有几个不同的版本，分别在 Unix 系统(通过 X Windows 系统)、Windows 系统和 MacOS 系统下实现相应的开发 Tk 工具包的工作。Python 支持一个叫作 Tkinter 的库。Tkinter 库是个对 Tk API 的接口。Tkinter 库作为标准配置，包含在 Python 的所有版本当中。

Tk 工具包最初是由 Dr. John Ousterhout 开发的。Dr. John Ousterhout 在去 Sun Microsystems 公司之前曾在加里福尼亚大学的伯克利分校就职。后来，Ousterhout 创建了一个叫 Scriptics 的新公司。Scriptics 公司的主要业务是辅助开发 Tk 工具包和 Tcl 语言，为其商业版作准备。Scriptics 公司现在已经被 Interwoven 公司购买，但是 Tcl 语言和 Tk 工具包的开发仍然延续使用 Scriptics 这个公司名字。最初的 Tcl 语言和 Tk 工具包现在仍然免费提供，当然 Scriptics 公司也开发了如 TclPro 这样的商业版产品。

Tk 工具包的主要目标是简化用户接口的设计过程。Tk 窗口化系统提供用于简化操作和简

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

化事件的方法和基础工具，简化诸如打开窗口、画线和接受来自键盘和鼠标的输入和动作这样的操作和事件。

甚至于，创建一个显示在屏幕上像按钮或简单的文本框这样单元的原始程序包含上百行代码。上百行代码的功能就是开发一个称作窗口小部件(widget)的 GUI 环境独立单元。一个单个窗口小部件能定义接口上的一个核心单元，如，一个按钮、一个滚动条，甚至于像刻度尺和分层列表这样更复杂的单元。这些窗口小部件自身还能由其他更简单的窗口小部件构成。Unix 系统和 X Windows 系统内，已经有许多不同关于窗口小部件的工具包产品，包括 Qt、Gnome、Motif、Athena、OpenWindows，当然也包括 Tk 了。

由于部件和对象之间的天然关系，脚本化语言开发 GUI 程序非常简单。Tk 工具包最初是为了和 Tcl 语言合作而开发的。Tcl(工具命令语言的缩写)实质上是个宏语言，此语言用于简化 shell 下复杂程序的开发。总之，与 Python 和其他脚本化语言相比，Tcl 语言本身很难使用。所以，要努力使 Tk 窗口小部件直接在这些语言内得到支持。

Tkinter 库工作于与 pTk 相同的基础之上。Tkinter 库是个位于 Tk 真实函数库之上的普通端口层。pTk 系统现在已被许多不同语言使用，这些语言包括 Perl、Guile、 Scheme，当然也包括 Python。Tkinter 提供一个对底层 Tk 函数库的完全接口。

如果对用 Tk 或其他系统开发接口这个问题非常关注，作者建议，为了您的和您的用户利益，最好读一读关于人机接口的合适的书。在此，作者力推 Apple 的所有文档。这些文档是其后出现的许多最好接口的基石。可以查看一下 Alan Cooper 编写的《关于接口：用户接口设计的基础》或者看看 Theo Mandel 的《用户接口设计元素》入门指南的精彩之作。

15.1　Unix 下安装 Python/Tk

第一步要获得最新版本的 Tcl 语言和 Tk 函数库。能从 Scriptics 网站 (www.scriptics.com) 下载到这些资源。如果运行在 Linux 系统上，就会发现 Tcl 和 Tk 已经安装在此系统上。需要做的是把这两个软件包解压到同一个目录中，然后编译 Tcl 库，这之后才能建立 Tk 库。在大多数系统上，安装命令是个像如下所示命令一样的序列：

```
$ gunzip -c /export/contrib/archive/tcl8.3.2.tar.gz |tar xf -
$ bunzip2 -c /export/contrib/archive/tk8.3.2.tar.gz |tar xf -
$ cd tcl8.3.2/unix
$ ./configure
$ make
$ make install
$ cd ../..
$ cd tcl8.3.2/unix
$ ./configure
$ make
$ make install
```

现在需要修改 Python 原版当中的模块安装信息，以便建立 Tkinter 模块和 Python 要与 Tk

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

库通信所要求的扩展模块。

进入到 Python 源目录中，打开 Modules 目录下的 Setup 文件。在 Setup 文件里，找到需要指正的必需配置行，修改这些行使 Tk 有效。如，在 Solaris 系统下利用 Tcl/Tk 8.3.2 所需要的配置行如下所示：

```
 tkinter _tkinter.c tkappinit.c -DWITH_APPINIT \
-L/usr/local/lib \
-I/usr/local/include \
-I/usr/openwin/include \
-ltk8.3 -ltcl8.3 \
-L/usr/openwin/lib \
-lX11
```

现在，返回到 Python 的基本目录下，运行 make 命令重新建立 Python(如果必需的话)并安装 Tk 库。运行一个 make install 命令，就一切准备就绪了。

15.2　Windows 下安装 Python/Tk

从 Python 主要网站得到的标准 Python 安装程序实际上包括 Tk 库，并且此安装程序在安装过程中自动为用户安装 Tk 工具包和 Tkinter 库。

15.3　MasOS 下安装 Python/Tk

MacOS 系统下的 Python 安装程序包括 MacOS 系统下使用 Tk 工具包所需的 Tcl/Tk 库。在 Python 的安装过程中，Tcl/Tk 库被自动安装。

15.4　Tk 简介

用 Tk 工具包开发用户接口是创建许多嵌套对象的一个事例。要创建的第一个对象是应用程序的主窗口。嵌套对象是构成用户接口的独立部件。一个部件就是一个按钮、一个文本框、一个菜单、或其他各种各样用于创建主窗口的接口的一个元素。

一旦定义了组成主窗口的部件，接下来的脚本就进入到一个叫做事件循环的循环里。脚本接受来自于用户的事件，并执行部件在创建时所定义的命令和操作。这与大多数其他 Python 脚本不同，其后紧跟一个逻辑进程。总之，不像许多 Python 脚本那样，此脚本根据在哪个按钮上、哪个文本框上或哪个其他部件上进行了操作，用户能控制具体执行，选择许多不同选项。

创建一个基于 Tk 之上的 GUI 应用程序，基本步骤如下所示：

(1) 创建一个保含所有对象的窗口。主窗口一般称作主要或顶层，当然也能使用其他任何名称。

(2) 创建几个部件，定义部件的内容、动作和其他元素。下例，创建了一个表示简单消息

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

的标签和一个按钮。在单击此按钮时，就退出此脚本。

(3) 显示和整理窗口里的部件。此项任务常常由 Pack 几何管理器(Pack geometry manager)处理，当然也可以使用其他几何管理器。几何管理器提供一个方法，此方法允许控制部件在窗口内的方向和占据空间的大小。虽然开发者要做些控制。但实际上，几何管理器已为开发者做了大量工作。几何管理器根据开发人员对如何布局独立单元的建议做决定。

(4) 启动事件循环。现在，脚本的主要执行工作已经完成。脚本的剩余部分被配置独立部件的事件驱动。

下面是个演示上述这些步骤的简短 Python/Tk 脚本：

```
import sys
from Tkinter import *

def main():
    root = Tk()
    button = Button(root,
                    text = 'Hello, world',
                    command = quit_callback)

    button.pack()
    root.mainloop()

def quit_callback():
    sys.exit(0)

main()
```

此脚本在 Windows 98 机器上运行时，其结果如图 15-1 所示。

图 15-1　在 Windows 98 机器上运行安装 Tk 库脚本的结果

能很清楚地看到此脚本的效果。作为一个对比，图 15-2 是在 Red Hat Linux 机器上运行同一个脚本的结果。

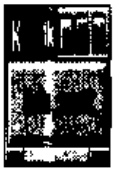

图 15-2　在 Red Hat Linux 机器上运行安装 Tk 库脚本的结果

最后，演示一下 Python 和 Tk 的真正跨平台本性，图 15-3 就是在 MacOS 系统上的运行结果。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

图 15-3　在 MacOs 机器上运行安装 Tk 库脚本的结果

这两个窗口的内容是等价的。只是窗口管理器放置在一起的装饰标志不同而已，这些不同的装饰标志是关于窗口尺寸重定义(resizing)、窗口最小化、窗口最大化或窗口关闭的。窗口装饰对于每个平台和窗口管理器而言是特定的。而且在 Tk 中创建的所有窗口都具备这些窗口装饰。

有 5 个重要的元素，在开发 Tk 接口的时候应当记住，它们如下所示：

- 窗口
- 部件
- 嵌套
- 尺寸管理
- 回调

15.4.1　窗口

窗口是所有部件的主要容器，并且是能用 Tk 在其中开发接口的惟一途径。没有了窗口，就不能创建部件。可以在同一应用程序里创建几个不同的主窗口。不能受限于：主窗口仅仅只能有一个。从应用开发的角度看，这使得 Tk 工具包更加实用，实际上，能开发出大多数所期望的基本 GUI 接口作品。这不仅包括基本的窗口，还包括浮动的调色板、弹出框和警告消息。

15.4.2　窗口小部件(Widgets)

主要要集中注意力关注一下独立窗口小部件是怎样创建的。不能创建一个位于窗口之外的小部件。窗口小部件必须创建在一些种类的容器里面。大多数容器是窗口，当然有些窗口小部件也可以是其他窗口小部件的容器。例如，Frame 这个窗口小部件就能包含其他窗口小部件。Frame 这个窗口小部件用于把一个或多个窗口小部件约束在窗口的某个区域里。更进一步讲，因为 Frame 这个窗口小部件自身就是个窗口小部件，所以能嵌套多个 Frame 窗口小部件产生复杂的布局。

15.4.3　嵌套

窗口小部件的嵌套是另一个重要原则问题。在微软 Windows 应用程序中，每个应用程序窗口一般是由两个主区域组成。窗口的最顶端包含菜单条，而窗口的剩余部分既可以包含一个由其他组件组成的单个帧，也可以包含一个允许多个窗口存在于一个较大帧中的接口。例如，在 Microsoft Word 中，就能打开多个文档，而这些打开的多个文档共享同一个菜单条。

注意:

每个窗口菜单条所包含的内容根据其所在环境的不同而不同。MacOS 系统是个首要例子，在屏幕的最顶端处有一个菜单条，所有应用程序都共享这个菜单条。在切换应用程序的时候，

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

菜单条的内容也随着改变以匹配当前活动的应用程序。这就使得菜单条是个要完全区别对待的独立项，几乎就像在其自己的窗口里一样。

Windows 应用程序中的菜单条所能包含的内容受到一些限制。尽管有一些应用程序创造了某些功能，但大多数 Windows 菜单还是限制为选项的简单列表。菜单条，事实上，就是个容器窗口小部件。而 MenuBar 对象也没有什么特殊的地方——事实上，MenuBar 对象在很大程度上以 Frame 窗口小部件为基础。可以把 MenuButton 放置于 MenuBar 窗口小部件内，每个 MenuButton 又可以由几个菜单项构成。总之，不像典型的 Windows 应用程序那样，基于 Tk 之上的应用程序能把任何部件放置于菜单项里：按钮、复选框、单选按钮，事实上，可以是任何所喜欢的小部件。

然而，因为 MenuBar 只是个窗口小部件，所以能把菜单放置于窗口中任何位置上——开发者不会被限制于只能生成位于窗口顶端的菜单。一个包含灵活性菜单和菜单中所嵌套的窗口小部件的组合是工具和调色板的主体，也是想在一个受限制区域内引入一个可能发生事件的复杂列表时的主体。

Tk 接口系统的嵌套能力可能是紧次于 Tk 跨平台兼容性的最强大特色性能。

15.4.4　几何管理

不要不考虑对几何管理器的需求。几何管理器实际所做的工作远远不止是组织独立窗口小部件在窗口中的布局这么一点。因为几何管理器主要还负责窗口小部件在屏幕上的显示(原因是只有几何管理器知道这些窗口小部件应当在屏幕的什么位置上显示)，所以实际上就是几何管理器在显示每一个窗口小部件。

如果不调用几何管理器，那么就不能显示任何窗口小部件。其原因是 Tk 本来不知道窗口小部件应当在屏幕的什么位置上显示。几何管理器根据开发者想配置的布局来设置这些信息。

15.4.5　回调

在本部分前面的演示脚本中，主要的小部件 Button 具有一个 command 属性。此属性指向 exit 方法，exit 方法则借助于一个匿名子例程。这个命令就是被称作为回调(callback)的命令——在执行某个操作的时候，此命令就从脚本的另一个部分调回一段代码。在上例中，如果用鼠标单击一下 Button 按钮，脚本就结束。

要充分理解回调及 Tk 窗口的其他单元是怎样工作的，首先需要弄懂事件循环。

15.4.6　事件循环

mainloop 方法执行一个简单的循环，此循环把来自于底层窗口系统的请求传输给相应的窗口小部件。对于每一个事件，都有一个在 command 属性中定义的方法要执行。总之，这就是被调方法的责任，其责任就是尽可能地快速执行其工作并放弃其控制权，为的是允许其他正在等待的事件执行其对应操作。

大量消耗 CPU 的复杂系统而言，还需要确保不同线程间多任务的高效执行，以便不锁定对早期事件循环提供服务的进程。对于一些应用程序和一些系统而言，这就要求手工把一个任务划分为几个可管理的大任务块，允许事件循环发送请求给其他回调方法。另一个可以选择的解

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

决方法是使用多线程应用程序模型。

阻塞的系统调用常常是 GUI 接口中较差的方法，其原因是处理位于事件堆栈中事件时系统阻塞。这是 Windows 上的一个特殊难题——进程阻塞实际上能冻结整个机器(即使这是不应该的)。最佳方法是使用如 select 这样的命令，select 为开发者完成打开文件句柄之间的多路通信工作。有所遗憾的是，这样的命令并不能为开发者解决处理 GUI 和文件事件这个难题，读者可能会想利用线程以达到解决此问题的目的(关于多线程的详细信息，请参见第 9 章)。

Mainloop 方法是不可配置的，不可能提供自己的版本。主循环仅在以下情况才退出：用户单击窗口化环境当中的关闭框时，或者当调用 sys.exit()时，或引发系统不能处理的异常时。

事件绑定

除了由 command 属性处理的基本事件绑定之外，还可以把诸如按键或单击鼠标这样具体的事件绑定到其他方法上。这就是如何把键盘按键和具体方法调用等同起来，如何提供命令和方法的快捷键问题。Tk 工具包提供了 bind 方法，此方法允许开发者把低层事件映射到相应的方法上。它也是定义 command 属性时被独立窗口小部件所运用的方法。此方法的使用格式如下所示：

```
widget.bind(event, callback)
```

其中：event 参数是要绑定的事件名，包括按键及单击鼠标(可以定义为按下和放开)。bind 方法能支持更加复杂的事件，如单击鼠标和拖动鼠标、鼠标普通的移动、鼠标指针进入或离开一个窗口，以及关于整个窗口的像尺寸重定义、图标化 / 最小化或隐藏这样的事件。

event 参数以字符串的形式定义，此字符串包含所要映射的序列。所要映射的序列可以由一个或多个独立事件构成，这个映射序列被称做事件序列(event sequence)。例如，如下所示的代码：

```
widget.bind("<z>", pressed_z);
```

映射用户按下不带任何修饰符的 CTRL 键，这个事件为 pressed z 方法。

关于 event 参数的其他可能取值如下所示：

```
widget.bind("<Control-z>", undo);
```

对上述命令而言，仅当 CTRL 键和 Z 键在同一时刻按下的时候，定义的事件才发生。而下面的代码：

```
widget.bind("<Escape><Control-z>", redo);
```

则在按下 ESC 键、然后按下 CTRL 键和 Z 键的时候，才调用 redo 事件。那么关于单击鼠标，可以使用如下所示的命令：

```
widget.bind("<Button1>", redo);
```

独立事件能被分组进称作 modifiers、types 及 details 的不同的类。一个修饰符就是一个特殊键，如 Escape 键、Control 键、Meta 键、Alt 及 Shift 键。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

鼠标按钮也能被分组到这个类中，因此能得到 Button1、Button2、Double(关于双击鼠标)及 Triple(关于三击鼠标，triple-click)。还有一个特殊修饰符，既 Any，此修饰符匹配任何一个修饰符，包括匹配零个修饰符。因为多按键鼠标在 Macintosh 系统上不是标准配置；而二键鼠标在 Unix 工作站上当然是普遍的，但是在 Windows 及 Macintosh 计算机上就不普遍。很显然，如果开发跨平台 Tk 环境，一定要小心谨慎。

事件的类型是 KeyPress、KeyRelease、ButtonPress、ButtonRelease、Enter、Leave 和 Motion 类型之一。请注意，能识别一个键的按下和释放两个事件。因此能配置游戏，比如说，接受某个键按下事件，仅当此键最终被释放才停止处理过程。这对按钮的按下和释放操作也是同样正确的。Leave 选项识别鼠标指针离开一个窗口小部件受限制区域的时候(用于撕下来的菜单和调色板)，而 Motion 识别鼠标指针移动，同时一个按钮和/或键盘组合按下的时候。

detail 类只用于键盘绑定，并且是定义被按下字符的字符串。除此之外，detail 类还支持 Enter、Right、Delete、Backspace、Escape、F1 和基本 ASCII 字符，如 A–Z 等等。

要使开发过程更简单，Tk 库还允许使用大多数普通按键的缩写形式，使得<KeyPress-z>能被指定为<z>，<Button1-ButtonPress>能被指定为<1>这么简单。

此外，Text 和 Canvas 窗口小部件甚至能允许独立绑定上的更精细粒度，允许把一个绑定附加于具体标签上。bind 的格式会相应改变：现在第一个参数定义要识别的标签；第二个参数和第三个参数定义要调用的绑定和方法。结果，能创建按下一段被定义标签的文本之上的第 2 个按钮的绑定，其命令如下所示：

```
text.bind('word', '<2>', synonym_menu);
```

获得事件细节

在事件发生的时候，传送给为此事件配置的回调一个单一参数。此参数就是一个 Event 对象，包含有关引发了的事件的信息。有关给定 Event 实例的属性如表 15-1 所示。

表 15-1　事 件 属 性

属　　性	说　　明
widget	产生此事件的窗口小部件。这是个有效的 Tkinter 部件实例，而不是个名字。此属性为所有事件而设置
X, y	鼠标的当前位置，以像素为单位
char	字符码(仅仅适用于键盘事件)
keysym	键符号(仅仅适用于键盘事件)
keycode	键码(仅仅适用于键盘事件)
width, height	窗口小部件的新尺寸，以像素为单位(仅仅适用于配置事件)

例如，可以通过访问 x 和 y 属性获得鼠标在窗口内的位置，因此能建立如下例所示的一个回调：

```
def callback(event):
    print "You clicked on", event.x, event.y
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

15.5　使用窗口小部件

为理解 Tk 系统是怎样工作的，让我们先简要地看看大多数普遍使用的窗口小部件。因篇幅所限，还有许多其他窗口小部件就没在本章列出。有关详细信息，请查阅 Fredrik Lundh 编写的《Tkinter 简介》，此书能从 Python 网站(www.python.org/topics/tkinter/doc.html)下载。

15.5.1　核心部件

Tk 库有许多预定义的窗口小部件。有一些是典型 GUI 应用程序的基本建立块，如，Button 和 Label 部件。其他则是其他窗口小部件的组合。表 15-2 列出 Tk 系统所支持的基本窗口小部件。

表 15-2　基本部件集

部　件　类	说　　明
BitmapImage	Image 窗口小部件的子类，用于显示位图图像
Button	简单的 push-button 窗口小部件，具有与 Label 窗口小部件类似的属性
Canvas	画图区域。在其中，可以放置圆、线、文本及其他图形元素(需要自己建立这些元素，或者把窗口小部件提供给最终用户，让最终用户为开发者做这些工作)
Checkbutton	多选择按钮窗口小部件，部件当中的每项能被单独选中
Entry	单行文本输入框
Frame	用于布置其他窗口小部件的容器
Image	用于显示位图、pixmap(彩色位图)和其他图形元素的简单窗口小部件
Label	简单文本框，能把消息文本放置于其中(不可编辑)
Listbox	一个能挑选选项的多行列表
Menu	菜单列表，此部件能由 Label、Message、Button 和其他窗口小部件构成
Menubutton	列出在 Menu 对象中指定选项的菜单(在一个单一菜单条里)
Message	多行 Label 对象(不可编辑)
OptionMenu	特殊类型的 Menu 窗口小部件，此部件提供在一个选择区域内的选择项的弹出列表
PhotoImage	Image 窗口小部件的子类，此部件用于显示真彩色图像
Radiobutton	多选按钮窗口小部件，其中只能选中多个选项中的一个选项值
Scale	滑动条，允许根据具体比例设置一个数值
Scrollbar	用于控制另一个窗口小部件的内容的滑动条，如 Text 或 Canvas 窗口小部件
Text	多行文本窗口小部件，此小部件支持能被标记以不同字体和颜色显示的可编辑的文本
Toplevel	窗口，此小部件能被父窗口管理器管理和调整

Tk 库的一大优势是，因为它支持基本水平的窗口小部件，所以能对这些基本窗口小部件进行组合和修改来建立其他的窗口小部件。例如，ScrolledText 窗口小部件就是 Scrollbar 和 Text

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

窗口小部件的一个组合。ScrolledText 窗口小部件允许根据 Scrollbar 窗口小部件的位置来控制 Text 窗口小部件的哪部分文本显示。

首先，这使得 Tk 库看起来远不如其他更富特色的工具包实用。如，不像 Windows 和一些基于 Unix 之上的工具包那样，Tk 库不支持标准对话框窗口小部件——必须自己动手制作。另一方面，因为必须自己动手制作，所以能制作出用户化版本的窗口小部件，可能包括一个错误或引用数字——预定义工具包不能支持的对象。开发进程的不足之处在于时间较长，要花费大量时间引入优秀的标准 GUI 作品，但是，最终灵活性会占上风。

让我们再仔细地看看一些更加普遍使用的窗口小部件。

15.5.2　普通部件属性

独立窗口小部件的配置是通过一系列属性(property)控制的。所有的窗口小部件都具有一组属性，定义从边框存在特性和颜色一直到字体样式和字体尺寸的所有特性。特殊化的窗口部件还具有关于构成此窗口小部件的惟一元素的属性。例如，MenuButton 窗口小部件就具有一个叫做 state 的属性，state 属性指明菜单是活动的还是禁用的。

所有窗口小部件能配置的普通属性如表 15-3 所示。

表 15-3　设置窗口小部件的普通属性

属　性	说　明
font	X 或 Windows 格式的字体名(请参见本章后面的“指定字体”部分)
background, bg	背景色，不是通过一个名字就是通过一个十六进制的 RGB 数值指定
foreground, fg	前景色，不是通过一个名字就是通过一个十六进制的 RGB 数值指定
text	显示于窗口小部件中的字符串，利用指定的前景色和字体数值显示此字符串
image, bitmap	要在窗口小部件内显示的图像或位图文件
relief	窗口小部件的边框样式，它应当是 raised、sunken、flat、ridge 或 groove 样式之一
borderwidth	relief 边框的宽度
height	窗口小部件的高度，对于标签、按钮和文本窗口小部件而言，以字符数指定，而对于所有其他窗口小部件而言，以像素数指定高度
width	窗口小部件的宽度，对于标签、按钮和文本窗口小部件而言，以字符数指定，而对于所有其他窗口小部件而言，以像素数指定宽度
textvariable	一个变量的名字，在窗口小部件改变的时候所使用的和 / 或所更新的变量名字
Anchor	定义窗口小部件在窗口中的位置，或窗口小部件内文本的位置。其有效值是 n、ne、e、se、s、sw、w、nw 和 center

1. 设置窗口小部件属性

在定义一个窗口小部件的时候，可以通过指定属性名字和属性值是所创建的窗口小部件类的关键字参数来设置属性，如下例所示：

```
button = Button(root, text = 'Hello, world', command = quit_callback)
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

另一个可以选择的是，在窗口小部件创建之后改变其属性，这又有两个选择。第一个选择是把每一个窗口小部件对象当作字典使用，所以能通过设置相应的属性直接更新属性值，如下例所示：

```
button['text'] = 'Hello, world'
```

第二个选择是使用 configure()方法，如下所示：

```
Button.configure(text = 'Hello, world',
                 command = quit_callback)
```

2. 获取窗口小部件属性

要获得为一给定窗口小部件而设置的属性，有两个选择。第一个选择是使用 cget() 方法，如下所示：

```
buttontext = button.cget('text')
```

但是，这第二个选择——直接获取字典元素大概更加简单，如下所示：

```
buttontext = button['text']
```

3. 指定字体

字体值按惯例以 XLFD(X 逻辑字体描述，X Logical Font Description)格式指定。字体值是由 14 个字段组成的一个复杂字符串，其中每两个字段之间由一个连字符分开。每个字段定义不同的属性。

例如，下面这个字体：

```
-sony-fixed-medium-r-normal--16-120-100-100-c-80-iso8859-1
```

定义一个这样的字体：来自于 sony 打造商、来自于 fixed 家族，中等重量。此字体是规则字体(不是斜体)，由“r”标识，字体宽度是正常值。字体是 16 像素，或 12 点高(点的尺寸被放大了 10 倍，因此，所定义的尺寸是 120 而不是 12)。接下来的两个字段指定分辨率，在本例当中，宽 100 个像素，高 100 个像素，和一个表示全体字符宽度为 80 的字符("c")。最后一个字段是注册名或字符的本地名字。

通常，不管怎样，在特定字段里能免除指定星号或问号作为通配符，而让 Tk 库和窗口化界面来确定合适的字体，所以能请求到一个更通用的字体。应当能免除指定 foundry、family、weight、slant 和 points 字段的操作。例如，要使用 12 点的 Helvetica 字体，　就可以使用如下所示的设置：

```
label.configure(font = '-adobe-helvetica-medium-r-*--*-120-*-*-*-*-*')
```

显然，这个设置过程很痛快。但是实际上这样的设置不能应用于更简单的 Windows 字体系统上，Tk 库也接受更简单的 Windows 和 MacOS 样式的定义，Windows 和 MacOS 样式定义与 Unix Tk 库也反向兼容。这种定义包括字体名字、点阵尺寸和重量，如下例所示：

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
label.configure(font = 'Helvetica 12 regular');
```

4. 指定颜色

X Windows 系统支持一个称作 rgb.txt 的文件，此文件把红、绿和蓝三源色映射为颜色名字。这允许用一个简单的名字指定一个颜色。

这儿有一小段内容，是从样本文件 rgb.txt 的开始处抽取出来的。如下所示：

```
255    250    250    Snow
248    248    255    ghost white
248    248    255    GhostWhite
47     79     79     DarkSlateGray
0      191    255    DeepSkyBlue
46     139    87     SeaGreen
178    34     34     firebrick
147    112    219    MediumPurple
```

显然，Windows 系统不使用 X Windows。但是，Windows 系统仍然能访问上要 X Windows 安装程序提供的核心颜色集合。如果想更具体地指定颜色，可以以#RGB、#RRGGBB、#RRRGGGBBB 和#RRRRGGGGBBBB 的形式严格地精确地指定 RGB 数值,，其中，R、G 和 B 代表的是一个对应于三源色的独立十六进制数字。

例如，能把 GhostWhite 颜色描述为#F8F8FF。对许多情况而言，使用 Python 格式运算符创建一个字符串可能更容易些，如下所示：

```
color = "#%02x%02x%02x" % (142,112,219)
```

5. 指定尺寸

在指定具体窗口小部件尺寸参数的时候，根据把此参数用在哪个窗口小部件上，有多个可以采用的选择。如果窗口小部件是基于图形，而不是基于文本的——比如说，Canvas——那么由 height 和 width 属性接受的尺寸指定方法以像素为单位。这个尺寸指定方法对具有图形，而不仅仅是文本的标签和按钮值也起作用。对于所有基于文本的窗口小部件，尺寸以字符为单位指定，这样窗口小部件的实际尺寸要依赖于用于显示文本字体的尺寸。

6. 图像和位图

某些窗口小部件支持使用图像，而不支持使用文本。例如，能在按钮上通常显示文本的位置使用一幅图像。从根本上而言，有两种类型的图像：一种是具有两个颜色的位图；一种是多色 pixmap。为有助于提高性能，Tk 库把一幅图像当作一个单独元素。如果需要在不止一处的位置显示此幅图像，对此图像进行一次渲染就可，所有要显示此幅图像的地方就使用这个渲染了的图像对象作为窗口小部件的图像源。这意味着在一个窗口小部件中使用图像要经过两个步骤。

第一步是创建要渲染的图像对象。尽管每个方法的返回值常常是同一种类型，但是渲染不同图像格式要使用不同的方法。要创建一个 X Bitmap (XBM) 图像对象，可以使用如下所示的

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

代码：

```
image = label.Bitmap(file = 'icon.xbm')
```

而要创建 X Pixmap (XPM) 图像对象，请使用下面这个代码：

```
image = label.Pixmap(file = 'icon.xpm')
```

对于 GIF 或 Portable Pixmap (PPM) 图像格式，需要使用 Photo 指令，如下所示：

```
image = label.Photo(file = 'icon.gif')
```

在想配置一个图像对象中的一个特定窗口小部件时，请使用 image 属性，如下所示：

```
label.configure(image = image)
```

关于位图，窗口小部件的 foreground 属性和 background 属性控制位图的前景色和后景色。

7. 窗口小部件变量

有一些窗口小部件使用一个单独的变量来保存其数值。对一些窗口小部件来说，这就是一个显示信息的方法。比如说可能想在一个 Label 窗口小部件内显示某个变量的值。而对于其他窗口小部件而言，此变量用作设置和接收特定窗口小部件数值的一个链接。关于这方面，一个比较好的例子是 Scale 窗口小部件，此部件给用户提供一个滑动条。当改变 Scale 窗口小部件所绑定的变量的值时，用户能改变此滑动条的位置。相反，改变滑动条的值也改变此变量的值。

遗憾的是，不能使用关于这些特定值的标准变量。而需要使用 Tkinter 类型类创建一个合适对象，如下例所示：

```
cbvar = IntVar()
cb = Checkbutton(master, text="Expand", variable=cbvar)
```

实际上，Tkinter 定义开发者能够使用的许多不同变量，这些变量包括 IntVar()、FloatVar()、StringVar()以及 BooleanVar()。不能直接访问或修改这些对象的值，而是需要使用 get()和 set()方法，如下所示：

```
cbvar.set(99)
print "The value of the checkbutton is",cbvar.get()
```

15.5.3　标签

Label 窗口小部件是个基本的窗口小部件，它提供一个简单方法，用来在窗口中显示一个小文本标签。Label 窗口小部件支持表 15-3 中列出的所有基本属性。

因为标签是个这样基本的单元，所以常是构成 Tk 工具包中其他许多窗口小部件的组成部分或基础。

15.5.4　按钮

Button 窗口小部件，从根本上讲，就是带有一个附加属性的标签。这个附加的属性就是

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

command，此属性是个指针，指向一个在按钮被单击时调用的方法。基本清单之外的附加属性和方法列于表 15-4 中。

表 15-4　按钮的属性和方法

属　性	说　明
command	一个对 Python 方法或函数的引用。在用鼠标按键 1 单击此按钮时调用对应的方法或函数
方　法	
flash	通过反转和重新设置按钮的前景色和背景色快速刷新按钮
invoke	启动 command 属性所定义的子例程

已经在本章前面关于"Hello, world"的脚本中见过了标签和按钮的举例。

15.5.5　单选按钮

Radiobutton 窗口小部件用于提供一个简单的开 / 关按钮，或用作几个不同选项之间的一个开关。对单选按钮有效的属性和方法列于表 15-5 中。

表 15-5　单选按钮的属性和方法

属　性	说　明
command	一个对 Python 方法或函数的引用。在用鼠标按键 1 单击此按钮时调用对应的方法或函数。与 variable 属性有关的变量在 command 变量引用的子例程激活之前，被更新为 value 属性的值
variable	把一个引用指向一个变量。在按钮单击的时候用 value 属性的值对此变量进行更新。当引用的变量值与 value 属性匹配的时候，按钮被自动选中
value	在按钮被选中的时候，指定其值存储在 variable 属性所指向的变量当中
方　法	
select	选中单选按钮，并且设置 variable 为 value
flash	通过反转和重新设置按钮的前景色和背景色快速刷新按钮
invoke	启动 command 属性所定义的子例程

请记住，必须使用一个 Tkinter 变量类型用来保存单选按钮的实际值。还请注意，单选按钮根据它们更新哪个变量而被分组。

例如，在下例当中，创建了两个独立的单选按钮组，一个单选按钮组用于表示名，而另一个单选按钮组用于表示姓，每一个单选按钮组更新自己的变量，如下所示：

```
from Tkinter import *

root = Tk();

firstname = StringVar()
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
lastname = StringVar()

radio1a = Radiobutton(root,text = 'Martin',
                      value = 'Martin', variable = firstname)
radio1b = Radiobutton(root, text = 'Sharon',
                      value = 'Sharon', variable = firstname)
radio1c = Radiobutton(root, text = 'Rikke',
                      value = 'Rikke', variable = firstname)
radio1a.pack(side = 'left')
radio1b.pack(side = 'left')
radio1c.pack(side = 'left')

radio2a = Radiobutton(root, text = 'Brown',
                      value = 'Brown', variable = lastname)
radio2b = Radiobutton(root, text = 'Penfold',
                      value = 'Penfold', variable = lastname)
radio2c = Radiobutton(root, text = 'Jorgensen',
                      value = 'Jorgensen', variable = lastname)
radio2a.pack(side = 'left')
radio2b.pack(side = 'left')
radio2c.pack(side = 'left')

root.mainloop()
```

请注意，在每个属性定义中使用着同样的变量，因此信息是共享的。对一个值的改变就更新相应的单选按钮组作出正确的选择。上述脚本的结果如图 15-4 所示。

图 15-4　单选按钮例子程序的运行结果

15.5.6　复选按钮

Checkbutton 窗口小部件，根据读者的不同背景，也被称作复选框(checkbox)，就像一个单选按钮。所不同的地方是复选框通常用于允许用户选中一个单个选项的多个复选按钮。有关 Checkbutton 窗口小部件的可能属性和方法如表 15-6 所示。

表 15-6　复选按钮的属性和方法

属　性	说　明
command	一个对 Perl 方法的引用。在用鼠标按键 1 单击此按钮时调用对应的方法。与 variable 属性有关的变量在 command 变量引用的子程序激活之前，被更新为 value 属性的值
variable	把一个引用指向一个变量。在复选按钮单击的时候用 value 属性的值对此变量进行更新。当引用的变量值与 value 属性匹配的时候，按钮被自动选中
onvalue	在按钮被选中的时候，指定其值存储于 variable 属性所指向的变量当中

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

（续表）

属　性	说　明
offvalue	在按钮未被选中的时候，指定其值存储于 variable 属性所指向的变量当中
indicatoron	如果是假(0)，则不显示复选按钮指示器。此属性切换整个窗口小部件的 relief 基本属性，实际上，使整个窗口小部件成为复选按钮
方　法	
select	选中复选按钮，并且设置 variable 为 value
flash	通过反转和重新设置按钮的前景颜色和背景颜色快速刷新按钮
invoke	启动 command 属性所定义的子程序
toggle	切换复选按钮的状态和值为开和关

15.5.7　文本框

Text 窗口小部件是个简单的文本框，用于显示多行文本。不像标签那样，标签实际上只用于在单行上显示少量的几个单词，而 Text 窗口小部件是个可编辑的信息输入框。Text 窗口小部件支持关于数据输入和文本框的来回移动的 emacs 快捷键。除了 Text 窗口小部件的可编辑特性之外，还能“标记”文本的独立部分以及改变其属性。这允许用多个字体、多个点阵尺寸和多个颜色支持来创建一个多特性的文本编辑器，而无需额外的编程。

Text 窗口小部件方法把一个或多个索引规范当作参数。一个参数可以是绝对数字(基)或是个相对数字(基和修饰符)，并且绝对数字和相对数字两者都可以以字符串形式指定。所支持的基本索引规范如下所示。第一列的项表明能修改的索引规范的单元。要不就是一个关键字。

line.char　　指明 line 行 char 处的字符。其中，char 指示字符在横向上的位置(从左至右)，而 line 指示字符在纵向上的位置(从上到下)。在一个文本框里，字符在一行(横向)上的位置定义从 0 开始，字符在纵向上的位置定义从 1 开始

end　　文本的末尾，就是最后换行符之后的那个字符

insert　　插入光标的位置

mark　　紧随其名字为 mark 的标记之后的字符

tag.first,tag.last　　用于指定一个标签的 first 和 last 字符

下面的索引规范还能被一个附加修饰符限定。

+count chars, – count chars, +count lines, – count lines　　用 count 字符或 lines 调整基本索引规范

wordstart, wordend, linestart, lineend　　调整索引来指向索引(wordstart, linestart)所指定的单词的第一个字符或行上的第一个字符，或者指向紧随单词或行(wordend，lineend)之后的那个字符

Text 窗口小部件所支持的一些属性和方法列于表 15-7 中。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

表 15-7　Text 窗口小部件所支持的属性和方法

属　　性	说　　明
tabs	用于 Text 窗口小部件的标签列表。其规范应当是对一个字符串列表的引用。每个字符串应当包含一个数字，此数字定义字符在行里的位置，此数字后面有一个后缀，指明特定标签对齐的方式。此后缀可以是 l、c、r 或分别指明为左对齐、中心对齐或右对齐
state	取值为 normal 或 disabled。如果取值为 normal，则是标准可编辑的文本框；如果取值为 disabled，则是不可修改的文本框
方　　法	
insert(index [, string [, tag]] ...)	用一个选项 tag 把 string 字符串插入到指定的 index 位置上
delete(index1 [,index2])	删除 index1 位置上的字符，或者删除从 index1 开始到 index2 为止的所有文本
get(index1 [,index2])	获取 index1 位置上的字符，或者获取从 index1 开始到 index2 为止的全部文本
index(index)	返回给定的 index 参数值所对应的绝对索引值
see(index)	如果 index 位置上的文本是可见的，则返回真
markSet(name, index)	把 index 位置上的文本的书标签名指定为 name
markUnset(name)	清除书签 name

如果要把一段文本插入到一个文本框的尾部，可以使用如下代码：

```
text.insert('Beginning!', 'end')
```

要把同样的文本段插入到第 5 行的第 20 个字符的位置上，可以使用如下代码：

```
text.insert('Beginning!', '5.20')
```

要指定和配置标签，需要使用表 15-8 中列出的方法和属性。

表 15-8　标签的方法和属性

属　　性	说　　明
-foreground, -background, -font	同基本属性
-justify	标了签的文本的对齐方式，其取值为 center、left 或 right 之一
-relief, -borderwidth	边框的宽度和凸纹的样式
-tabs	同基本文本窗口小部件的属性(请参见表 15-7)，但是仅应用于此行的第一个字符也属于同一个标签的时候。不能给一个标了签的块添加“子标签”
-underline	给标了签的文本加上下划线
方　　法	
tagAdd(name [,index1[,index2]] ...)	把 name 标签名添加在 index1 指定的位置上，或把 name 标签名添加在 index1 和 index2 约束的范围上

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

方　法	说　明
tagRemove(name [,index1[,index2]] ...)	把 name 标签名从 index1 和 index2 指定的字符或范围上删除，但是不删除此标签的实际定义
tagDelete(name)	移掉并删除 name 标签
tagConfigure	配置标签的一个或多个属性

如果要创建一个简单的标签，请使用如下代码：

```
text.tagAdd('tagged', '1.0', '3.0')
```

上述代码创建一个叫作 tagged 的标签，区域从第 1 行开始，到第 3 行结束，其中包含第 3 行。因为在配置独立标签选项的时候需要使用标签的名字，所以标签的名字必须惟一。

因而要把 tagged 名字标签的文本改变为 24 点粗体的字体，可以使用如下所示的代码：

```
text.tagConfigure('tagged', font = 'Times 24 Bold')
```

15.5.8　输入框

从根本上而言，Entry 窗口小部件是单行文本框。此小部件继承了 Text 窗口小部件的许多特性和方法。但是，由于 Entry 窗口小部件只是单个行，其索引和方法都非常简单。其索引选项如下所示：

number	窗口小部件内容的索引，从 0 开始 ，第一个字符的索引为 0
End	文本的结尾
insert	插入光标的位置，在光标之后
sel.first, sel.last	标签的第一个和最后一个字符

Entry 窗口小部件所支持的方法和属性列于表 15-9 中。

表 15-9　Entry 窗口小部件的属性和方法

属　性	说　明
show	一个简单的布尔值选项。如果设置了此属性，则把每个输入的字符都显示为*，这主要用于口令的输入。请注意，虽然字符以这种方式显示，复制和粘贴一个隐藏字段的内容将显露出其真实的内容
方　法	
get(index)	获取开始于 index 处的字符串
insert(index, string)	把 string 插入到 index 位置上
index(index)	返回一个相对索引值的绝对索引值
selectionFrom(index)	设置选中区域为从 index 开始一直到字段结尾处为止的区域
selectionTo(index)	设置选中区域为从字段开头开始一直到 index 为止的区域
selection(from, to)	设置字符的选中区域为从 index 开始一直到 to 为止的区域
selectionClear	清除选中区
selectionPresent	如果一个选中当前是激活的，则返回真

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

15.5.9　列表框

Listbox 窗口小部件使得能够创建列表，能从列表中选择一个独立项。此窗口小部件的每一行显示一个字符串列表，并且所有显示的字符串具有同样的特性。在创建列表时，填充此列表的最简便方法是：创建此列表小部件，然后使用 insert 方法往此列表里添加各个项。Listbox 窗口小部件的 width 属性和 height 属性以字符为单位定义列表框的宽和高。另一个可以选择的是指定这两个属性值为 0，这将导致列表框不断变化以显示所有对象。

这儿是个脚本，使用 Listbox 窗口小部件的一个举例，如下所示：

```
from Tkinter import *

root = Tk();

list = Listbox(root, height = 5, width = 0)
list.insert('end', 'Martin','Sharon','Wendy','Sharon','Chris')
list.pack()

root.mainloop()
```

上述脚本的执行结果如图 15-5 所示。

图 15-5　创建 Listbox 窗口小部件举例结果

请注意，需要使用本章前面部分已经举例演示的 bind 方法，bind 方法把一个特定的操作，如双击，绑定到一个方法上。在此方法中，需要使用 get 方法来获取到当前的选中区域。

能像在 Textbox 窗口小部件中选取文本一样指定 Listbox 窗口小部件中的独立元素。能指定一个这样的字符串：定义行、行选中或在列表中的相对位置。其细节如下所示：

number	行的索引值，从 0 开始，第一项的索引为 0
end	当前行的末尾
active	光标当前所在的位置；在列表中，激活的位置呈现为划上下划线
anchor	选中项的锚点

Listbox 窗口小部件所支持的属性和方法列于表 15-10 中。

表 15-10　Listbox 窗口小部件所支持的属性和方法

属　　性	说　　明
height, width	以行为单位列表的高度，以字符为单位列表的宽度。如果其中任一项属性是 0，窗口小部件调整其尺寸以将所有列表项集成进
selectMode	定义列表的选取模式。取值为 single、browse、multiple 和 extended 中的一个值

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

方　法	说　明
get(index)	获取开始于 index 处的字符串
insert(index, string)	把 string 字符串插入到 index 位置上
delete(index [, last])	删除 index 所在的行，或者删除从 index 开始到 last 为止的所有行
see(index)	把 index 元素放到当前视图中
selectionFrom(index)	选取从 index 处开始一直到列表结尾为止的所有行
selectionTo(index)	选取从列表开头处开始一直到 index 为止的所有行
selection(from, to)	选取从 from 开始一直到 to 为止的所有行
selectionClear	清除选取区域
selectionPresent	如果有一个激活的选中区域，就返回真
Curselection	所有选中项的索引值的列表

15.5.10　菜单

逻辑上，把菜单分离成为一个个 MenuButton 窗口小部件。其中，MenuButton 窗口小部件就是菜单的名字。MenuButton 窗口小部件就变成一个容器，此容器能保存独立的菜单项小部件。菜单项小部件又分离成为不同的类型，允许把普通菜单项(实际上就是标签)、按钮、复选按钮和单选按钮添加到菜单上。

创建一个菜单的正常方法如下：

(1) 利用 Frame 窗口小部件创建一个菜单条帧，用于保存独立的菜单按钮；

(2) 创建新帧内的独立菜单按钮。每一个菜单按钮定义一个单独的菜单，因此需要创建如 File、Edit 和 Help 这些菜单按钮来保持正常的菜单样式标准；

(3) 使用 MenuButton 窗口小部件之上的方法创建一个新 Menu 窗口小部件；

(4) 使用 Menu 窗口小部件之上的方法创建独立菜单选项；

(5) 一旦所有的独立菜单和按钮都创建完毕，调用菜单帧之上的 tk_menuBar()方法创建最后的菜单布局。

Menu 窗口小部件的属性和方法列于表 15-11 中。

表 15-11　Menu 窗口小部件的属性和方法

属　性	说　明
IndicatorOn	如果为真，在菜单的右端显示一个小菱形
State	菜单的状态，可取 normal、active 或 disabled 之中的一个值
方　法	
add_command(options)	创建相对应的菜单项类型；对于标准菜单输入框而言，是正常类型
add_radiobutton(options)	添加一个单选按钮菜单项
add_checkbutton(options)	添加一个复选按钮菜单项

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

方　法	说　明
add_cascade(options)	插入一个新的级联(分层)菜单项
add(type, options)	添加一个带有 options 的 type 类型的新菜单
delete(index1 [, index2])	删除 index1 的菜单项，或者删除从 index1 开始一直到 index2 为止的所有菜单项
entryconfig(index, options)	改变 index 所指菜单项的属性
index(item)	返回 item 字符串所指定的菜单项的 index 数目，item 字符串应当匹配于菜单项的标签

表 15-11 中所列方法支持的可配置选项 options 列于表 15-12 中，options 可配置选项工作方式与窗口小部件的属性相同。请注意，因为具有的是分层菜单，所以各个独立项能使用表 15-11 列出的方法。

单独的 Menu 输入框所支持的独立属性如表 15-12 所示。

表 15-12　普通菜单项属性

属　性	说　明
indicatorOn	如果为真，把一个小菱形放置于菜单选项处，这允许用一个菜单来把这个选项切换打开和关闭两种状态
selectColor	如果 indicatorOn 是真，就是指示器的颜色
tearOff	如果是真，菜单的第一个元素就是一个分隔符。单击此分隔符，就把此菜单“撕下来”成为一个单独的最高层窗口。通常，此特性不是所用工具都支持
label	菜单项使用的文本。正常情况下，这应当用在 text 属性的位置上
underline	划上下划线的字符的索引。这个属性与 accelerator 属性组合一起使用用来指明此菜单使用的快捷键
accelerator	显示字符串，这个字符串是要显示和调整的字符串，用作菜单选项的等价键。这并不把此键绑定到所使用的命令上，所以必须分别定义
state	状态，其取值为 normal、active 和 disabled 之一
command	在菜单项被选中的时候要调用的函数名
value	依附与单选按钮之上的值(请参见表 15-5)
variable	用于存储 value 的变量
onvalue,offvalue	等价于表 15-6 中的选项，表 15-6 中是关于单选按钮样式输入框的

如要创建一个简单的命令菜单，可以使用像如下所示脚本一样的代码：

```
from Tkinter import *

def new_file():
    print "opening new file"
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
def open_file():
    print "opening OLD file"

def print_something():
    print "picked a menu item"

def makeCommandMenu():
    Command_b = Menubutton(mBar,
                           text='Simple Button Commands',
                           underline=0)
    Command_b.pack(side=LEFT, padx="2m")
    Command_b.menu = Menu(Command_b)
    Command_b.menu.add_command(label="Undo")
    Command_b.menu.entryconfig(0, state=DISABLED)
    Command_b.menu.add_command(label='New...', underline=0,
                               command=new_file)
    Command_b.menu.add_command(label='Open...', underline=0,
                               command=open_file)
    Command_b.menu.add_command(
                        label='Different Font',
                        underline=0,
                        font='-*-helvetica-*-r-*-*-*-180-*-*-*-*-*-*',
                               command=print_something)
    Command_b.menu.add_command(bitmap="info")
    Command_b.menu.add('separator')
    Command_b.menu.add_command(label='Quit', underline=0,
                               background='red',
                               activebackground='green',
                               command=Command_b.quit)
    Command_b['menu'] = Command_b.menu

    return Command_b

root = Tk()

mBar = Frame(root, relief=RAISED, borderwidth=2)
mBar.pack(fill=X)

Command_b = makeCommandMenu()

mBar.tk_menuBar(Command_b)

root.title('menu demo')
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
root.iconname('menu demo')

root.mainloop()
```

15.5.11　帧

Frame 窗口小部件是关于其他窗口小部件的简单容器。Frame 窗口小部件用在需要创建一个复杂布局的时候。其中，此复杂布局要求的几何管理比正常使用有效工具的管理要高级得多。工作方式是把独立窗口区域划分成帧，把对象的集合封装到帧中。比如，可以创建一个新的帧，新帧可以包含贴到窗口顶端的菜单条。其中，菜单条中的菜单按钮实际上是水平分布的。在本章后面部分在学习 Scale 窗口小部件的时候，会看到关于 Frame 窗口小部件的一个例子。

15.5.12　滚动条

滚动条可以用做这样的情况下的独立窗口小部件：负责管理正在滚动的相应窗口小部件，也可以被自动地添加到所有合适的窗口小部件上。

让我们首先讨论一下自动滚动的滚动条。要创建一个自动滚动的窗口小部件，可以先使用特定 Scrolled 窗口小部件方法，然后指定要创建的带有一个滚动条的窗口小部件的类型。例如，下面就是从创建一个滚动的 Text 窗口小部件的文本浏览器复制过来的命令。

```
maintext = Scrolled(root, 'Text');
```

这创建了一个包含主 Text 窗口小部件和关于 Text 窗口小部件的水平(和垂直)滚动条的帧。实际上，其返回值指向这个新创建的 Frame 窗口小部件。

另一个可选择的方法是，利用表 15-13 和表 15-14 所列出的方法和属性来创建和管理自己的滚动条。表 15-13 所列方法允许把窗口小部件的当前视图设置为想要关联这个滚动条的那个视图。set 方法控制当前视图，而在滚动条被移动的时候 command 属性被调用。

表 15-13　滚动条的方法和属性

属　性	说　明
command	关于一个子程序的一个引用，这个子程序用于改变窗口小部件的视图
方　法	
set(first, last)	指明当前视图。first 和 last 元素应当是 0 到 1 之间的一个小数，如 0.1 和 0.2 这两个值指明窗口小部件的所有数据项 10%到 20%间的区域被显示
get	返回当前滚动条设置

所有可滚动的窗口小部件还支持如表 15-14 所示的方法和属性。属性定义方法并增加对滚动条的控制。滚动条窗口小部件自动调用合适的方法(xview 或 yview)来修改相链接窗口小部件的显示。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

表 15-14　可滚动窗口小部件的属性和方法

属　　性	说　　明
xscrollincrement, yscrollincrement	*x* 和 *y* 轴方向上的滚动是由给定增量确定的
xscrollcommand, yscrollcommand	对方法的一个引用，此方法用于在滚动条被移动的时候重新定位窗口小部件的位置
方　　法	
xview('moveto', fraction) yview('moveto', fraction)	把滚动条移动到 fraction 指定的位置上。这个新值指明滚动条标签的最左端或最顶端，字符或像素。请注意，第一个参数是个常量
xview('scroll', number, what) yview('scroll', number, what)	指明视图应当向上或向下移动，向左或向右移动，移动的增量为 number。如果 what 参数是"units"，根据 xscrollincrement 和属性 yscrollincrement 属性中指定的增量滚动；如果 what 参数是"pages"，窗口小部件滚动 number 页

15.5.13　刻度条

刻度条就像温度计一样。定义一个尺寸和范围，窗口小部件就显示一个水平的或垂直的滑动条。这个滑动条自动就具有一个标签(如果已经定义了一个标签)和刻度计来指明独立刻度。本部分，将看到用于建立这个滑动条的代码，一个用于把英尺转换为米的窗口小部件。这个窗口小部件如图 15-6 所示。

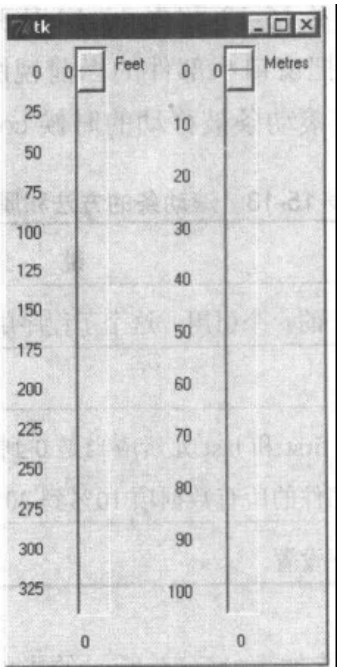

图 15-6　把英尺转换为米的刻度条窗口小部件

Scale 窗口小部件所支持的属性和方法列于表 15-15 中。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

表 15-15　Scale 窗口小部件所支持的属性和方法

属　性	说　明
command	对一个子程序的引用。此子程序在刻度条值改变的时候被调用
variable	对一个变量的引用，不管什么时候，只要滑动条移动，此变量就被更新。此属性的工作方式就像 variable 基本属性一样，更新这个值也设置滑动条的位置
Width, length	以像素为单位(不是字符)的刻度条的宽和长
orient	允许进行 horizontal 或 vertical 方向的选择
from, to	窗口小部件的 from 和 to 的真正值范围
resolution	所显示的值和 variable 内的设置常常是此属性值的几倍。默认设置为 1
tickinterval	刻度条上刻度计之间的间隔，是间隔的真实值
Label	显示于刻度条的顶端(水平)或左端(垂直)的标签
方　法	
set(value)	等同于修改 variable 的值

此处的脚本是产生图 15-6 的脚本。此脚本提供一个简单方法把英尺转换为米以及其逆转换。其代码如下所示：

```
from Tkinter import *

def main():
    main = Tk();

    global feetscale, metrescale
    feetscale = IntVar()
    metrescale = IntVar()
    feetframe = Frame(main)

    Scale(feetframe,
        command = update_feet,
        variable = feetscale,
        width = 20,
        length = 400,
        orient = 'vertical',
        from_ = 0,
        to = 328,
        resolution = 1,
        tickinterval = 25,
        label = 'Feet'
        ).pack(side = 'top')

    Label(feetframe,
        textvariable = feetscale
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
            ).pack(side = 'top',
                   pady = 5)

        feetframe.pack(side = 'left')

        metreframe = Frame(main)

        Scale(metreframe,
            command = update_metre,
            variable = metrescale,
            width = 20,
            length = 400,
            orient = 'vertical',
            from_ = 0,
            to = 100,
            resolution = 1,
            tickinterval = 10,
            label = 'Metres'
            ).pack(side = 'top');

        Label(metreframe,
            textvariable = metrescale
            ).pack(side = 'top',
            pady = 5)

        metreframe.pack(side = 'left');

        main.mainloop()

    def update_feet(self):
        global metrescale, feetscale
        metrescale.set(int(feetscale.get()/3.280839895))

    def update_metre(self):
        global metrescale, feetscale
        feetscale.set(int(metrescale.get()*3.280839895))

    main()
```

在本章前面部分已经看到了这段代码的 Windows 版本结果。图 15-7 就是在 MacOS 系统下运行此脚本的结果。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

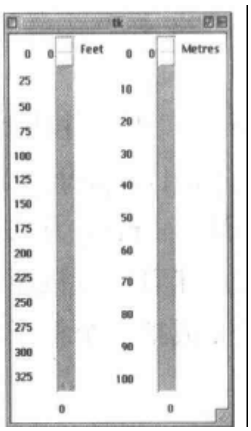

图 15-7　MacOS 系统下把英尺转换为米的刻度条窗口小部件

图 15-8 是在 Unix/X Windows 系统下运行此脚本的结果。

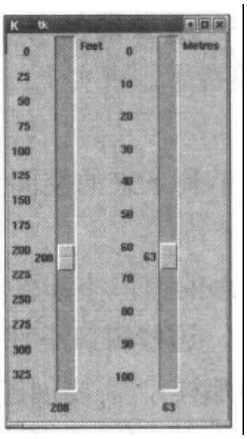

图 15-8　Unix/X Windows 系统下把英尺转换为米的刻度条窗口小部件

15.6　控制窗口几何

贯穿本章的是 pack 方法的范例。读者早已知道 pack 方法是窗口建立过程中一个必需的元素。但是，有一些使用 pack 的技巧，能辅助窗口中独立窗口小部件的布置。Tk 还支持两个布置窗口小部件的其他方法：定位器(placer)和栅格。虽然为适应所需可以混合和匹配一个单个窗口里的多个帧的独立几何管理器，但是在单个父窗口内必须使用同样的几何管理器。

因为必须利用 x 轴和 y 轴坐标指定每个窗口小部件的位置，所以为了正确地使用定位器，需要做一些细致的计划。这个系统是用于 Motif 和 Visual Basic 的 Bulletin Board 窗口小部件内的同样系统。因此，那些原来使用其他系统的开发人员用这个系统可能会备感轻松。

栅格几何管理器使用一个简单的表布局，如在文字处理器中所使用的一样，与用 HTML 设计网页时一样。每个窗口小部件放置于一个表单元里，通过定义窗口小部件应当显示于其中的列和行能指定其位置。如果需要，独立的窗口小部件能占据多个列和多个行。与使用定位器几何管理器一样，必须谨慎细致地考虑怎样布局此系统的窗口小部件。

包容器(Packer)几何管理器是本章使用的一个几何管理器。如果不想过多地关心几何管理过程，这是最实用的几何管理器。就这一点而言，它就是本章将最关注的内容。如果想更加细致地对这个系统加以了解，请参见 Tkinter 文档。Tkinter 文档从主要的 Python 网站能下载。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

15.6.1　包容器

包容器几何管理器与 Motif 的 Form 窗口小部件相似，它们都使用很简单的系统，用来定义窗口小部件在窗口的一个帧内的位置。请记住，pack 方法就是这个几何管理器——pack 方法仅仅只提供用于组织窗口小部件布局的算法。对 pack 方法的单独调用就把相应的窗口小部件组装到帧或窗口内的下一个有效空间里。这意味着可以按照窗口小部件被组装的顺序把它们添加到窗口或帧中，这与怎样把一个包塞进柜子或把一个柜子装满相似：首先从一个单点出发，添加窗口小部件直到空间全部使用完了为止。

这个算法的工作方式如下。

(1) 给定一个帧，包容器把一个窗口小部件附加到特定一侧(顶端、底端、左端或右端)；

(2) 窗口小部件所使用的空间占据帧中有效的空间，这个有效的空间区域称作包(parcel)。如果这个窗口小部件没有完全填满整个包(如果包空间比分离给这个窗口小部件的区域宽或高)，则从根本上而言，那个空间就被浪费了。事实上，这就是支持附加 Frame 窗口小部件的原因，为的是使空间充分所用。

(3) 然后，下一个窗口小部件放置于剩下来的空间里。窗口小部件再一次能把自己依附于顶端、底端或一侧来用尽所有可用空间。

(4) 请注意，所有指定一个特定锚点的窗口小部件将组装在一起，而共享同一个空间。那么，如果用左锚点指定多个窗口小部件，这些窗口小部件将从左至右分布于帧之中。如果想制作一个更加复杂的布局(如 Scale 例子中那样)，需要创建一个个独立的帧。

包容器方法的有效选项列于表 15-16 中。像 Tk 系统的其他元素一样，关于 pack 方法的选项以字典的形式指定。如果不指定任何选项，包容器尺寸管理器从顶端到底端把窗口小部件插入到帧里。

表 15-16　pack 方法的选项

属　性	说　明
Side	帧的一个侧面，就是窗口小部件要填加于其中的侧面。此属性的取值应当是 left、right、top 和 bottom 中的一个
Fill	指定窗口小部件是否填满包的 x 轴或 y 轴方向上的空间；也能指定此属性值为 both 来填满包 x 轴和 y 轴方向上的空间；或指定此属性值为 none 来阻止填满包的全部空间。ipadx 或 ipady 选项能用来指明包内围绕窗口小部件保留的一点附加的空白填充空间
Expand	指定窗口小部件在其他窗口小部件放置完毕是否扩展占掉所有剩余空间。此属性用于 Textbox 窗口小部件。此属性特别用于定义一个外部菜单和工具条而想让主要的窗口小部件占掉所有剩余空间的时候
padx, pady	窗口小部件之间的间隔空间，以像素、毫米、英寸或点为单位指定其值(还请参见表 15-17)
ipadx, ipady	包内围绕窗口小部件的“填充”空间，以像素、厘米、英寸、毫米或点阵为单位指定其值(还请参见表 15-17)

padx、pady、ipadx 和 ipady 属性接受一个字符串，而不是一个数字值。根据此参数值的后缀，把这个值解释为以像素、厘米、英寸、毫米或点为单位。对于像素之外的值，几何管理器

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

询问窗口管理器和确定屏幕的显示分辨率和密度来决定实际使用的像素数目。例如，在一个以 96dpi 密度显示的典型 Windows 屏幕上，有效后缀列于表 15-17 中。"1i"这个值的定义将采用 96 个像素来填充。

表 15-17　填充字符后缀

后　缀	说　明
none	尺寸以像素为单位计算
C	尺寸解释为屏幕上的厘米数
I	尺寸解释为屏幕上的英寸数
M	尺寸解释为屏幕上的毫米数
P	尺寸解释为打印机上的点数(一个点近似于 1/72 英寸)。这是指定字体尺寸时打印机所使用的点尺寸的同一个单位

15.6.2　栅格

栅格几何管理器工作方式与 HTML 中表的方式等价。单个窗口小部件放置于一个具体列和具体行的栅格中。单个窗口小部件被限制在栅格的每个单元内，但是即使在需要的时候，单个单元不能跨占地多于一列或一行空间。

grid 方法是关于栅格几何管理器的接口。根据每个窗口小部件应当显示于其中的列和行指定每个窗口小部件的位置。栅格的最终尺寸基于所指定的最大行和列数。grid 方法的属性列于表 15-18 中。

表 15-18　Grid 几何管理器的属性

属　性	值
column	把窗口小部件插入其中的列
columnspan	窗口小部件在栅格内要跨占的列数
row	把窗口小部件插入其中的行
rowspan	窗口小部件在栅格内要跨占的行数
sticky	定义窗口小部件将粘贴于其中的父窗口小部件的侧。此属性应当指定为 0 或 n、s、e 或 w 之中的多个字符。如果默认其值，窗口小部件就粘贴于栅格单元的中心处。如果指定为 n 和 s(或 e 和 w)，则窗口小部件伸缩以填满栅格单元的高(或宽)。如果属性值指定为所有 4 个字符，窗口小部件伸长填满整个栅格单元
padx, pady	窗口小部件之间的间隔空间，以像素、毫米、英寸或点为单位指定其值(请参见表 15-17)
ipadx, ipady	包内围绕窗口小部件的“填充”空间，以像素、毫米、英寸或点为单位指定其值(请参见表 15-17)

15.6.3　定位器

定位器的工作方式与其他两个几何管理器显著不同。既然，包容器和栅格的工作基础是根

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

据窗口页上的其他窗口小部件来对窗口小部件进行对齐，那定位器允许非常精确地指定想把窗口小部件放置于窗口上的位置。这个窗口小部件的位置指定以把窗口小部件要放置于其中的窗口的位置和尺寸为基础。如果想把每个窗口布置得像栅格几何管理器的每个栅格单元一样，必须要想点办法。这样，窗口小部件就被放置到通过此过程创建的窗口里。因为严格的位置定义，必须小心谨慎以确保不会疏忽大意覆盖了某个窗口小部件。

关于窗口自身尺寸的定义是与窗口小部件的父窗口(是 Window 或 Frame，或其他包容器窗口小部件)相关联定义的。在这个规则的武装下，能指定以下几种的一种。

(1) 父窗口内窗口的尺寸和位置(以像素为单位)；

(2) 相对于父窗口，窗口的尺寸和位置；

(3) 上述两者的组合，因此能拥有一个变动位置的固定尺寸或变动尺寸的固定位置。

这样，能使一个窗口小部件放置于父窗口的中心处，并且此窗口小部件随父窗口增长边框和尺寸。这些选项非常有用于 Canvas、Text 和其他窗口小部件，而对于这些窗口小部件，开发者想扩展其显示区域而不影响窗口内其他窗口小部件。

对定位器管理器的接口是借助于把 place 方法用在窗口小部件之上来实现的。place 方法接受的键 / 值对列于表 15-19 中。

表 15-19　Placer 几何管理器的属性

属　性	说　明
In	窗口小部件应当相对于其放置的窗口小部件(对象)。此属性的值必须是个有效的窗口小部件对象，并且必须是父窗口或是父窗口的一个子窗口。还必须保证窗口小部件和其父是同一个窗口的两个子窗口
x, y	用作窗口小部件锚点的 *x* 轴(水平)和 *y* 轴(垂直)坐标。请参见表 15-17。表 15-17 是这些数值的有效限定符的清单
relx, rely	父窗口中的 *x* 轴相对坐标。此数值应当定义为一个浮点数值。其中，0.0 指示父窗口的左边界，而 1.0 指示父窗口的右边界。这样，0.5 设置就应当把窗口小部件放置于父窗口的中心位置上
anchor	定义窗口中的哪一点应当作为锚点对待。使用正常的 n、ne、e、se、s、sw、w 和 nw 值
width,height	指定窗口的宽和高。请参见表 15-17。表 15-17 是这些值的有效限定符的一个清单。请注意，在两种情况里，测量值定义窗口的外部宽度，包括所有边框
relwidth,relheight	与父窗口尺寸相比，窗口的相对宽度或高度。其中，值为 0.5 意味着窗口是父窗口的一半大小；值为 1.0 意味着窗口具有与父窗口相同大小的宽和高
bordermode	取值为 inside(默认取值)、outside 或 ignore 三者之一。如果设置为 inside，则窗口区域计算结果比父窗口的所有边框都小；如果设置为 outside，则窗口区域计算结果就包括父窗口边框所占区域；如果设置为 ignore，则窗口区域的计算不考虑边框尺寸，使得整个父窗口都可以使用

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

注意

x、relx、y 和 rely 设置能组合在一起使用。在 relx 设置为 0.5，而 x 设置为 5 的时候，就把窗口小部件放置在父窗口小部件的中心靠右五个像素的位置上。这对于 width、relwidth、height 和 relheight,来说一样是正确的。其中，一个给定 relwidth 为 1.0，给定 width 为 5 的定义将产生一个比父窗口小五个像素的窗口。

有关利用 Tk 的更多范例，请参见《Python 注释文档》这本书的第 4 章。本书当中的代码可以从作者网站(www.mcwords.com)下载。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第 3 部分　应用程序开发

第16章　Python作为RAD工具使用

开发应用程序是一项竞争残酷的工作，尤其是因特网急剧发展的今天。就在我们有了想法正与他人商讨时，或许有人已经付诸行动了。将自己的想法越早付诸应用程序中，就能越快地进入市场，将其公布于众，并成为评价类似应用程序的标准。

问题就在于程序的设计是件复杂的工作。无论是多有能力的程序员，都需要设计算法，实现排序和流程，而所有这些的代码也都要由人去键入。

开发应用程序时似乎并不需要应付太多的事情，也就是要调试、优化代码，确保应用程序运行速度正常，不会当用户单击错误的位置、输入错误的值，或是忘记在磁盘上留下足够的空间以存储数据时发生崩溃。

没有什么简单的解决方法，我们还没有达到电影《星际旅行》中的地步，可以让计算机写段程序，重新装配好出故障的引擎以提高效率。但是今天已经出现了一些优秀的程序语言，开发人员通过使用这些语言的开发工具、扩展以及功能，可以使得整个开发过程更加容易。

16.1　何为 RAD

RAD(Rapid Application Dev elopment，快速应用程序开发)技术就是要在最短时间里，开发出应用程序，并且在产品的 α 测试、β 测试及最终交付过程中，保证应用程序稳定、有效并且运行良好。最根本的是，RAD 技术就是要使所有的应用程序开发更加简单，而这需要特定的语言和环境。

使用 RAD 技术开发应用程序，遵循的方法与典型的应用程序稍有不同。通常的开发过程采用组合的方法来进行，包括互相之间的商讨、对算法和过程的小型测试，以及使用尽可能多的可重用内容。尽管有许多工作是着眼于考虑开发的设计和进程，但多数时候不过是一张记录着注意事项的纸片，还有代码中对当前情况说明的不必要的注释。

无论如何这并非是说 RAD 技术开发的的应用程序是不合格或是开发失败的。多年来，从那些计划和设计周密的项目到那些使用 RAD 技术编码的项目中，通过这许多不同项目的工作经历，在达到最终产品的质量方面，作者倾向于使用 RAD 技术的开发方法。这或许是因为测试工作量的关系：遍历每一个单独的组件的测试工作量，要远大于对作为整体完成的应用程序的测试工作量(参见本章后面“开发生命周期”内容部分)。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

16.1.1　RAD 需求

如何或是什么能够使一个环境适于 RAD，这并没有什么固定又快速的方法或通用的标准，但是，以下的一些关键因素可以使得 RAD 程序开发更加简单，而可以不管底层语言。

- 可用的库和扩展：有许多可以用于提高开发速度的常用方法，其中之一就是通过使用外部库和模块。很明显，使用“smtplib”并相应地调整代码，比重新开发满足工作要求的 SMTP 接口要快得多。

同时，部分库/扩展的问题也与接口相关。当开发独立应用程序时，无论是需要自身的 GUI 接口(构建于 Tk/Gnome/Qt/X Windows/Cocoa/Carbon 等等之上)，还是通过 Web 界面方式，这个问题都不可忽略。界面开发得越快，产品就会越早走向市场。

- 面向对象：或许提及这一名词时许多人感到陌生。OO 不是所有人的程序开发时都必需的解决方法，同样 Python 也不是适合所有的工作。但是 OO 的确能够很好地实现 RAD，因为它能够显著地使得向应用程序中添加新方法、类、和扩展更加简便。OO 在更大层面上也使得代码重用十分简单，尤其是对于今后的 RAD 技术项目非常有帮助。

- 代码重用：日积月累，无数的函数、类和扩展被人编写，这些对于建立在最初曾开发过的应用程序之上的各种应用程序都十分有帮助。代码重用的简单实现，不必花费额外的时间将其转换为合适的模块或使其成为公用扩展或库来使用，这将有助于改善新项目的开发进度。

- 透明编程：许多语言当其开始管理应用程序执行的环境时，都提供基本的灵活性。开发人员可以控制所有事情，从对象和变量的破坏，到希望掌握的可分配的内存信息。所有这些都十分有用，但同时也成为一个主要的需求。获取内存分配的参数确实必不可少：内存耗费过多会降低应用程序的执行速度，太少又会增加程序崩溃的风险。

对于 RAD 技术开发的应用程序，需要一种语言可以掌控所有这些因素，使开发人员不用担心需要多少内存，或是当使用完变量后如何清除变量以释放内存(称作无用单元收集)。开发人员所惟一要关注的就是掌握并处理已有的信息，从而实现开发目标。

- 易用性：如果必须花费时间解决语言中如何实现各种动作的话，程序的设计不无枯燥。有经验的程序员显然有优势，但是即使是熟悉语言本身的程序员也会被古怪的界面或含糊不清的语义所困扰，这样的语义提供了太多的灵活性而缺乏足够的结构性。其他人也会因为要猜测用于获得结果的方法而减慢进度。一个经典的例子就是根据关键字进行数组排序或数据处理，这种操作在一些语言中速度很快，而在其他的语言中则会花费很多时间。相应的文档以及简单的注释会有所帮助的。

- 快速的开发生命周期：开发任何应用程序都是一个耗时的过程，因此在测试开发人员所做改动是否生效前，最后一件必需的事情就是：必须等待源代码被编译为应用程序。即使在运行速度很快的机器上，普通的应用程序也要花费几分钟时间进行编译。大型的应用程序可能会花费数小时时间，更大的例如“Microsoft Word”或“Mozilla”等应用程序会花费数天时间编译。能够设想在测试错误的更改生效前，还要等待好几天时间吗？

上述要点中的一些或是全部都可能有助于缩短开发时间，最终减轻普通程序员的负担。请记住，这里讲述的纯粹是对于从头脑中的主意到最终的产品的步骤的简化，以及减少开发过程中所有瓶颈影响的可能的方法。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

16.1.2　可选的 RAD 解决方案

许多语言都具有上一节中讨论的要素，也自然就成为可选的 RAD 解决方案。请不要略过本节内容。一个尺码不会适合所有人；同样，一种语言也不可能适合所有的程序员，更重要的是，不适合所有解决方案。本章的重点就在于说明：作为 RAD，为什么 Python 好于其他语言。下面将 Python 与其他一些顶尖的解决方案在性能和功能上进行如下比较。

● Pascal/Modula-2

由于 Pascal 的简单易学，它常常成为人们所学习的第一种语言。实际上，作为一种应用程序开发语言，Pascal 有着很好的历史：许多 Mac OS 的程序代码是用 Pascal 写成，而且当 Apple 取消对它的支持同时开发出 Mac OS 8 的时候，它依然是众多流行可选的编程语言之一。

Modula-2 则不太惹人注意。作为 Pascal 的高度面向对象的版本，它开发于 20 世纪 80 年代，同时进行了一些改动，包括实时的环境和多线程能力，它的多线程能力甚至能够在一般不支持多线程的平台上起作用。

这两种语言都具有大量的库，或多或少地具有面向对象的能力。在许多情况下，它们也能进行变量的管理和无用单元收集，这也是它们能被极力推荐给初学者的原因之一。

然而，这两种语言和其他编译语言一样，开发周期长，也不能像 C/ C++甚至 Perl 或 Python 一样，提供高级接口或系统编程能力。Delphi 是 Pascal 的一个成功的 RAD 环境，作为 Windows 平台下的开发环境，较之 Python 透明的跨平台开发显然存在局限性。

● C/C++/Objective C

我们真能将 C/C++/称作 RAD 的可选语言吗？不错，有人会赞成。环顾可用的开发工具，如微软公司的 Visual Studio 或 Cocoa(用于 Mac OS X 的新型界面开发工具)提供的新开发环境，就会发现有人将其作为可行选择的原因。

例如，使用 Cocoa 或 Visual Studio，可以建立可视化的界面，并在完成应用程序的后期添加必要的挂钩控件、回调函数和事件处理程序。实际上，借助于 Cocoa，可以只是简单地将组件拖放到窗口或对话框中，然后添加很少的代码，就能在很短的时间里开发出复杂的应用程序。无需写一行代码，Cocoa 及其开发环境就会添加所有的响应，包括尺寸可变的控件、自动的滚动条以及其他一些用户控件。

然而，时至今日，它仍然存在一些很明显的问题，在前面介绍 RAD 需求的要素时已经提到了这些问题。它的开发生命周期太长，使用 Microsoft、Apple、GNU 以及其他公司提供的工具时，开发人员还需要管理自己的变量并进行无用单元的收集。无论他们对于各种形态的 C 语言编程有多么精通，所有问题仍然会给开发至少带来 50%的额外负担。

● Perl

Perl 是那些非 Perl 程序员不喜欢的语言。先不考虑语义的细节，从理论上讲 Perl 和 Python 编程并没有太多差别。它们都是脚本语言，都允许开发时间上的快速变化，也都拥有很大的扩展、接口和其他增强功能的集合。

Perl 的缺陷在于，尽管所有的工作由许多开发人员提交，它在很大程度上仍然是一种系统的、管理员的以及处理文本的语言。它提供面向对象，但并不是完整的或是编程时所必需的。而且，由于语言本身的不易读性，随着时间的流逝，Perl 将会更加难以维护和支持。Perl 解释

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

器花费许多代价将其自身的思想标记到开发人员所做的工作上，当开发人员明白了这一点，Perl 的口号“条条大路通罗马”(TIMTOWTDI)将会变成灾难而不是优势。

● Tcl

即使有着广泛的追随者，Tcl 语言仍然经常被遗忘。Tcl 被开发出来，主要用作传统的 Unix 命令解释程序的扩展。它的结构实际上十分接近典型的命令解释脚本，如果需要使用 GUI(Graphical User Interface，图形用户接口)，可以使用 Tk 接口，这一接口也被 Python 使用并支持。

同时，和 Perl 和 Python 一样，Tcl 也能被嵌入传统的 C/C++应用程序中，用以提供脚本功能。然而，数据处理不是 Tcl 的强项。Python 所支持的强数据类型要强大灵活得多，而且 Tcl 也不直接提供面向对象的接口。

Tcl 还存在其自身独有的问题：它从未摆脱作为高级命令解释脚本图像的角色，而且由于语言的灵活性所限，它也不适于许多应用程序的实际开发。

● BASIC

BASIC 可能是最著名的一种语言，许多最初的计算机都内置有 BASIC 解释器，使得用户可以迅速开始编程。然而，尽管 BASIC 允许非常快速的开发进度，但其许多版本仍然不支持类、面向对象、强数据类型以及像 Python 所提供的高层的编程。由于 BASIC 最初的设计目的并不适于从外部重用代码，所以扩展 BASIC 也并不容易。

16.2　为何选用 Python

前面回顾了 RAD 环境的要求，还对一些适合或不适合要求的可选语言做了粗略的描述，但还是没有说明为何其中的一些语言，无论是采用传统的方法还是 RAD 技术，都无法用于开发应用程序。

在许多情况下，Python 实际上都是借用上述语言的许多特性和优点，来提供用户熟悉的功能，而这些特性与优点与相应的语言相比并不逊色。Python 并不总是适合要求：它从未被设计用来在所有条件下都能代替所有其他的语言。即使十分偏向 Python，仍然会使用到 Perl 作为开发项目中重要的部分，此外还有其他一些需要提供比 Perl 或 Python 更快速度的项目是使用 C 或 C++的。

然而 RAD 技术有着特别的要求，最终还是要着眼于简化编程过程，所有我们提及的语言几乎都无法完全符合要求。同样，Python 也不符合一些要求，但是它满足最主要的几点：

- 快速的开发生命周期
- 易于使用和编程
- 高层次、面向对象的方法
- 简单的代码重用和大量的扩展库

Python 也提供一些并非 RAD 技术所必需的额外能力，它们同样有助于开发过程。

16.2.1　开发生命周期

RAD 环境最重要的要素是开发应用程序的速度。略去我们已经或将要讨论的所有要素，耗费时间最多的就是语言本身。如果采用本身的开发周期就很耗费时间的语言和环境，那在进行

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

应用程序开发时就已经处于劣势了。

1. 编译型语言

类似于 C/C++、Pascal 以及其他许多种语言都是编译型语言，也就是要先编写源代码，再编译、连接后才可以执行。完整的过程如下：

(1) 编辑源代码；

(2) 将源代码编译为目标文件；

(3) 将目标文件和标准库连接成为可执行文件；

(4) 启动应用程序；

(5) 测试程序行为；

(6) 启动调试器；

(7) 调试应用程序；

(8) 中止应用程序；

(9) 回到第 1 步。

考虑到典型的应用程序由许多源文件组成，而且在能够进行最终应用程序的连接和执行之前，其中的一个或多个文件就必须被编译，上述过程会耗时更长。

同时，由于最终的应用程序是可执行的，因此如果应用程序中有错误，将不能被很快地找出和清除。可以在调试器中启动应用程序，但即使如此也不能在其运行时修改代码，在返回并编辑步骤 1 中最初的文件前，只能识别何处出错。

2. 脚本语言

Perl、Python、Tcl 以及其他许多的脚本语言提供了不同的解决方法。脚本语言名称的由来是因为存在一个执行脚本的解释器。从本质上讲，失去解释器的能力，脚本将一事无成，解释器提供所有的功能，从用户和系统的接口到变量和存储信息的基本指令。

脚本语言的主要优势是消除了编译环节，从而缩短了开发周期。实际上，脚本语言基本的开发过程如下所示：

(1) 编辑源代码；

(2) 启动应用程序；

(3) 测试程序行为；

(4) 中止应用程序；

(5) 回到第 1 步。

使用 Python 可以跨越开发的环节。在所有运行中的应用程序中，都可以使用 Python 的分析器。在分析器环境下，可以单步执行某一条命令语句，并且不用返回和编辑最初的代码，就可以编辑和检查语句。而且使用 eval，exec 或者 execfile，可以不必中止应用程序就能修改代码。

如果使用外部模块来支持应用程序，当应用程序还处于运行状态时，同样能够通过使用 reload 功能，重新加载外部模块。甚至所有对类、函数和方法的修改都不用中止应用程序的执行。对于单次运行的应用程序这并没有什么特别，但对于不间断运行的应用程序来说，例如网络服务器或 CGI 应用程序，就不用实际地停止服务以对其操作进行改动。

因此，在实际工作中，可以仅仅通过以下三个步骤，就能缩短 Python 应用程序的开发生命

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

周期：

(1) 启动应用程序；

(2) 测试程序行为；

(3) 编辑程序代码，返回到步骤 2。

通常 Python 会稍有些复杂，但是最终会使得开发更快，在进行快速应用程序开发时会带来很显著的帮助。

通过使用 Python，可以对不同的方法和技术进行交互的原型设计、试用并实现。在将单独的 Python 应用程序的组件集成到最终的应用程序之前，也可以在不同程度上独立地对其进行测试。由于进行基于对象的操作， Python 非常易于将组件集成为最终的应用程序，而不用担心其兼容性。

因此最终可以采用如下所示的两层的开发过程：

(1) 测试组件；

(2) 编辑程序代码；

(3) 如果此单元完成，转到步骤 4；否则转到步骤 1；

(4) 测试应用程序；返回到步骤 2。

16.2.2　高层编程

类似于 C/C++，广泛地讲还有 Perl 这样的语言，它们与操作系统的联系比与用户的联系更加紧密。它们允许用户十分贴近所在的操作系统，这对于进行底层操作开发的人很有好处。

对于开发应用程序的开发人员来说，不必要完全屏蔽掉底层，也不用对底层进行操作。举例而言，当需要根据模式来获得目录下文件的列表时，开发人员自己不用去处理文件的名称，只需要一个函数，用来准确地返回想要的匹配文件列表。

在顶层，Python 的简易性使其十分易于使用和理解。不用考虑变量的底层复杂性、无用单元的处理或是内存的管理。同样，可以访问大量的底层系统的扩展和接口的库，这使得 Python 解决问题非常简单。

代码生成得更加简单，使得程序也更短小。Python 程序的大小经常只是同样的 C/C++程序大小的零头，甚至与相应的 Perl 程序也是大小相当。更短小、更简洁的程序意味着更快的编程速度、更简单的技术支持和修改工作、以及更少的出错概率，所有这些都将进一步缩短开发时间。

16.2.3　方便的跨平台兼容性

跨平台能力不是所有人首要考虑的因素，但对于某些人来说，作为提供给用户的接口，跨平台能力与开发进度一样重要。许多语言都具有跨平台的兼容性：C/C++几乎在所有平台都可用，同样，在很多其他的平台上可以使用 Perl、Tcl 和 BASIC 的某一版本。

然而，大多数语言都存在各种形式的不相容性或限制。C/C++的连接显然十分脆弱。由于它生成二进制的可执行代码，对于每一种希望支持应用程序的平台，都必须分别进行编译软件。这还仅仅是部分工作，由于对特定的 GUI 也没有通用的接口来创建应用程序，因此几乎要为给定的平台重新开发应用程序，来利用相应平台的接口。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

上面提到的其他一些语言有天生的优势。BASIC、Perl 和 Tcl 都是解释型语言。因为不用处理二进制源文件，就可能在不同程度上，将应用程序从一个平台移到另一个平台上去执行。同样，也会遇上接口的问题，但是即使是 Python 也没有一个接口。Python 语言的 Tk 工具包在 Windows、Mac OS 和 Unix 上都能被支持，它适用于大多数平台，能提供比 Perl 更多的集成度，而 Perl 语言现在只能在 Windows 和 Unix 上支持 Tk。

Python 的优势就在于已经解决了那么多的问题。开发本地 UI(User Interface，用户接口)的时间和用 C/C++差不多，然而却不用考虑其他的问题。

16.2.4　Python 和 OOP

为了使用面向对象系统，Python 提供了最强大、最灵活、最重要而且最简单的方法。用 C++实现的 OOP(Object Oriented Programming，面向对象的程序设计)很强大，但是太复杂，尤其是对那些新用户，还在困扰于面向对象的基本概念，更不要说 C++实现的复杂性。Python 的 OOP 实现，相比 C++提供的实现方法简单得多，也就能既提供功能又免去了复杂性。

OOP 也是语言整体必要的部分，因此没有理由不在应用程序的所有地方使用。由于实现的简便，不必考虑如何用 Python 去达到 OOP，这也有助于进一步缩短开发时间。无需 C++带有的提供跨平台兼容性的包(baggage)，可以很容易地重用 Python 的对象。

16.2.5　Python 的嵌入扩展性

通过使用 C 和 C++来提供附加的功能，可以对 Python 进行扩展以提高速度，或者仅仅是提供不同于本地 Python 解释器的系统接口。Python 也可以被嵌入到 C/C++应用程序中，以提供与 Visual Basic 类似的脚本组件或是由某些应用程序所支持的宏语言。

这种 C/C++与 Python 的嵌入扩展性会带来一些特别的机会。举例来说，假如使用 RAD 技术来开发 Python 应用程序，将会遇上性能瓶颈问题。可以用 C 语言重写应用程序的一部分，将这些扩展部分插入应用程序中，从而无需做其他任何改动就能获得更快的执行速度性能。

16.2.6　Python：在 Steroids 层次的 RAD

本章到底讲述了什么呢？

作者很愿意将 Python 作为 RAD on steroids。它具有所有合适的条件，也具有合适的整体性能。使用 Python，可以快速开发出自动被许多平台所支持的应用程序，而只花费很少的时间来编写、更加少的时间来调试和后期的技术支持及更新。

对于那些需要提高执行速度的代码，可以将其用 C 语言重写。实际上如果需要这么做的话，可以很容易地将应用程序的主体移植到 C 或 C++上，而仍旧使用 Python 作为连接代码来集成所有部分，创建出独一无二的应用程序。

在随后的章里，将介绍使用 Python 作为 RAD 的一般和特殊方法。也会介绍如何决定将 Python 应用程序移植为 C/C++混合体。想了解更多关于扩展和嵌入 Python 的内容，请参见本书第 26 章和第 27 章。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第17章　使用Python开发应用程序

任何有助于加快开发进度的措施都值得采用——永远不要光说不做。无论是 RAD 特有的要求，还是能够将整个过程的进度安排缩短数分钟、数小时或数天，从而有益于整个过程的额外措施，抑或是可以防止疲劳和咖啡因的刺激的措施，它们都会减轻开发人员的负担。集成开发环境(Integrated Development Environment , IDE)提供统一的编辑、运行和调试环境，可以有助于减少在应用程序和测试各项指标的环境之间切换的时间。

来自于诸如“Vaults of Parnassus” 之类的资源库或“PythonWare”之类的商业性开发中心的各种扩展程序和附加模块也是有用的。如果它们不合适，可以自行开发扩展程序；如果立足于 C/C++语言，可以将 Python 嵌入到应用程序中以添加即时的功能。如果是开发基于 Web 或因特网的应用程序，还可以采用 Zope 或 Jython。

前一章中列出了 RAD 的目标和要求，并介绍了 Python 是如何解决一些瓶颈问题来加快应用程序开发的。本章将对一些可选的利用 Python 加快开发进度的解决方案进行更全面的讲述。

同样，有必要去考虑如何编程和开发的思想，是逐位地代码编程并在编程过程中组织材料，还是尝试着进行规划？为回答这问题，本章还将讨论这两种方式以及其他一些在编写 Python 应用程序时需要考虑的问题。

本章的目的就是为了揭示出典型 Python 程序的本质和相关的开发步骤，同时指出如需要时何处可以获得帮助。如果想迅速了解额外的 Python 信息，可以参见附录 B。

17.1　集成开发环境

任何应用程序可以见到的能够改善开发进度的措施，都可以作为 RAD 问题的解决方法。与所开发的应用程序中所包含的内容和使用的功能相比，如何进行编程同样重要。

时至今日，选择程序设计环境取决于个人的喜好，同时或多或少还与工作的平台有关。Unix/Linux 用户最常使用的环境是 EMACS 或 vi，有时还使用 shell 程序或内置的开发环境。在 Windows 平台上，许多人使用记事本(Notepad)或写字板(Wordpad)以及命令行窗口进行开发，这或许会让人感到奇怪。还有人使用程序员编辑器，如 Kedit 或 UltraEdit；另外一些人则使用 EMACS 的 Windows 版本。在 Mac 平台上，人们常使用 BBEdit、EMACS 或 Pepper。

所有这些可选的环境都缺少与 Python 本身的集成。当然，其中一些环境能够标记出 Python 脚本，并可以高亮显示出不同的组件。一些环境甚至允许不用切换到命令行窗口中或打开 Python 应用程序，就能在编辑器中直接执行 Python 脚本。然而，没有一种环境允许在运行时对代码进行交互修改，也不能提供以结构化的形式查看 Python 模块、类和对象的调试功能。

基于这些原因，就需要 IDE。有相当多的解决方案可以供选用，具体的选用决定于开发 Python 应用程序时所使用的操作系统平台种类。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

17.1.1　IDLE

IDLE 是一种完全用 Python 编写成的相当基本的 IDE，它使用第 15 章中介绍的 Tkinter GUI 来创建界面。由于 IDLE 是由 Python 和 Tk 编写的，所以可以在任何支持这两种系统的平台上运行。首先，IDLE 提供一个带有增强交互功能的命令行解释器窗口，它与命令行窗口相似，但是增加了回滚、剪切以及粘贴的功能。同时 IDLE 还提供一个完全的由 Python 识别的编辑器，具有代码高亮显示和完成功能，有查看模块中的类和类组件的类浏览器和交互的调试器。主要组件示例如图 17-1 所示。

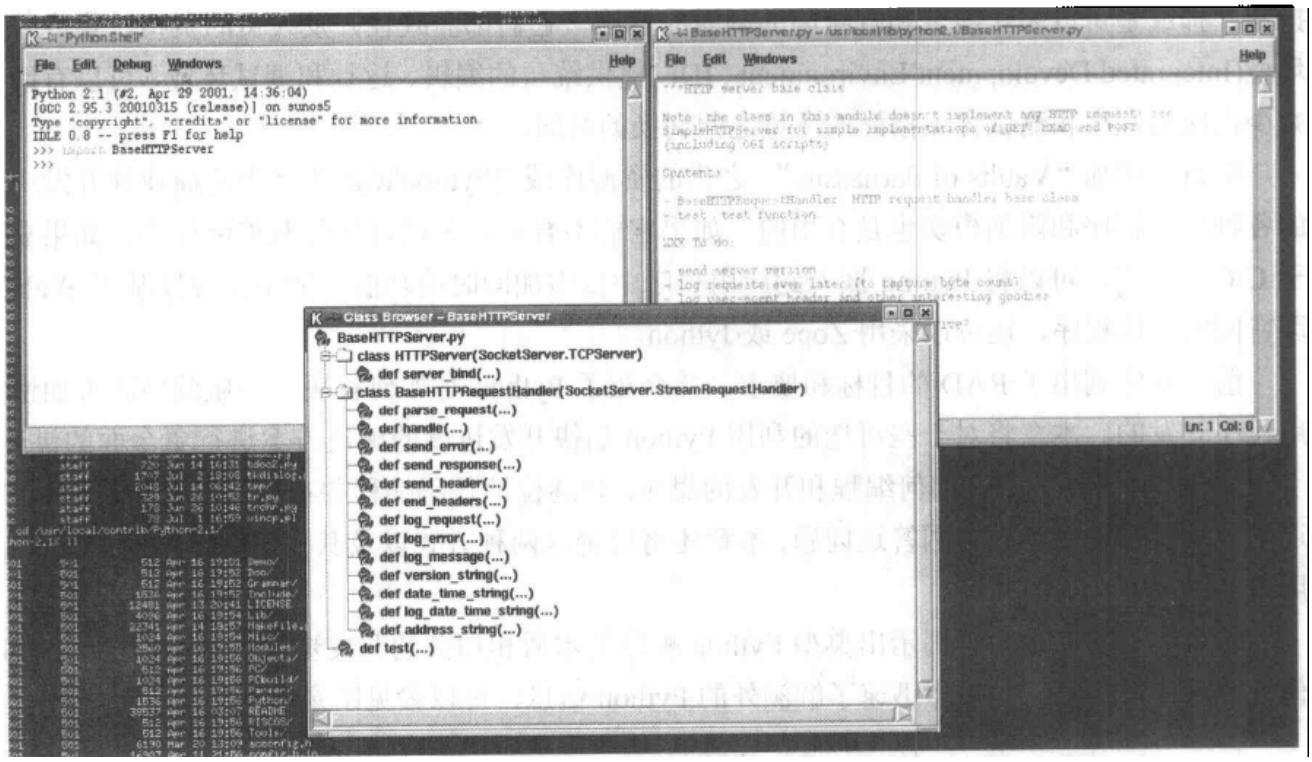

图 17-1　IDLE 集成开发环境工作状态

注意：

IDLE 界面中很好的一个特点是，它的菜单类似于 Tk 的“拆分型”菜单，可以收起调试器，再对菜单进行编辑将其去掉，这样就可以将菜单当作工具栏来使用。

IDLE 调试器提供一系列的特性，与交互调试器界面提供的核心调试器类似，所以可以进行单步调试、断点设置以及变量监视。IDLE 不能查看内存中的特定地址和变量的存储内容，也无法监视定时器或使用配置管理，但是这些都不过是小问题，

IDLE 并非毫无问题。当执行无法终止的脚本或应用程序时，IDE 很容易被锁死，即使按 Control+C 键也不能停止程序中使用的循环指令或 Web 服务器的执行。而且，当窗口以任何方式被隐藏后，有时会无法重绘；当正在执行的脚本被非正常中止时，窗口会完全无法更新。

17.1.2　PythonWin

PythonWin 是另一种推荐的 Python IDE，它只支持 Windows 平台。除了基本的调试器、类浏览器、交互窗口和编辑器外，PythonWin 还提供比 IDLE 更好的界面。PythonWin 的编辑器十

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

分有用，允许在键入时自动完成模块、类和函数的名称，这很像 Microsoft Word 提供的自动完成功能。图 17-2 所示为 PythonWin 的样例。

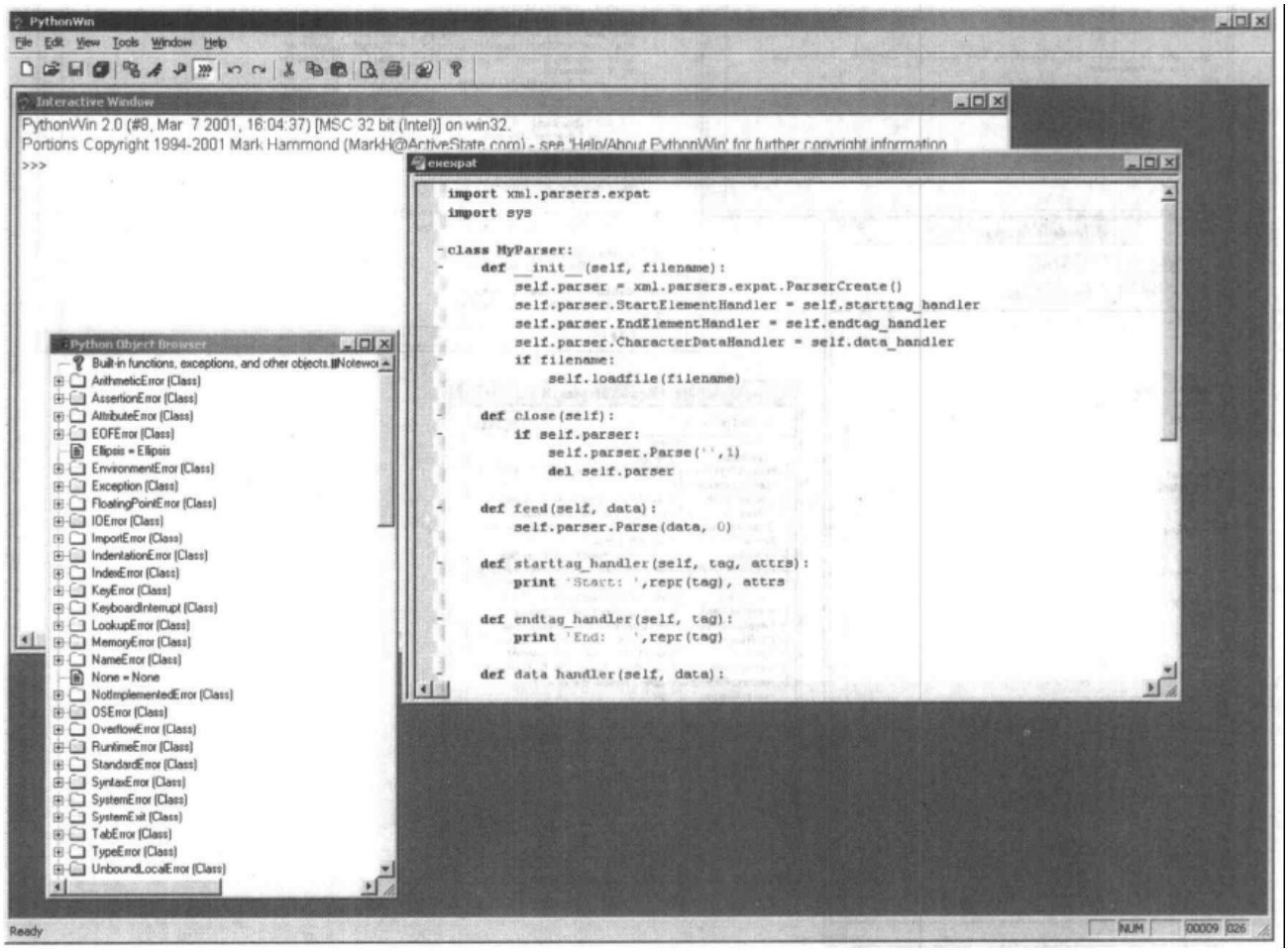

图 17-2　PythonWin 集成开发环境

PythonWin 的调试器提供的功能多于 IDLE，允许监视和代码的检查，而且在应用程序执行时，还有用于修改和改变程序组件的交互式窗口。然而，PythonWin 执行时仍然有问题，而且 IDE 也可能锁死，此时就需要强制结束以继续后续的工作。

17.1.3　MacPython IDE

MacPython IDE 显然是专用于 Mac OS 平台的，它符合标准 MacPython 分类。从字面上看，MacPython 由 Python 语言家族的另一成员编写成，它以“Just van Rossum”的形式，提供 Mac 样式的 IDLE 版本，同时保留 IDLE 的大部分功能。

这一 IDE 将基本的编辑器、调试器和代码配置管理整合成一个交互性的会话。其编辑器的确十分基本，提供简单的缩进和编辑功能，但不提供本节中介绍的其他 IDE 所具有的元素高亮、彩色显示或自动完成的功能。用惯 EMACS 和 BBEdit 之类编辑器的程序员将会感到有些失望，但是 MacPython IDE 的其他优点足可弥补这些缺点。尤其有用的是模块浏览器，这是 MacPython IDE 提供的最大优点之一。图 17-3 所示为 MacPython 的样例。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

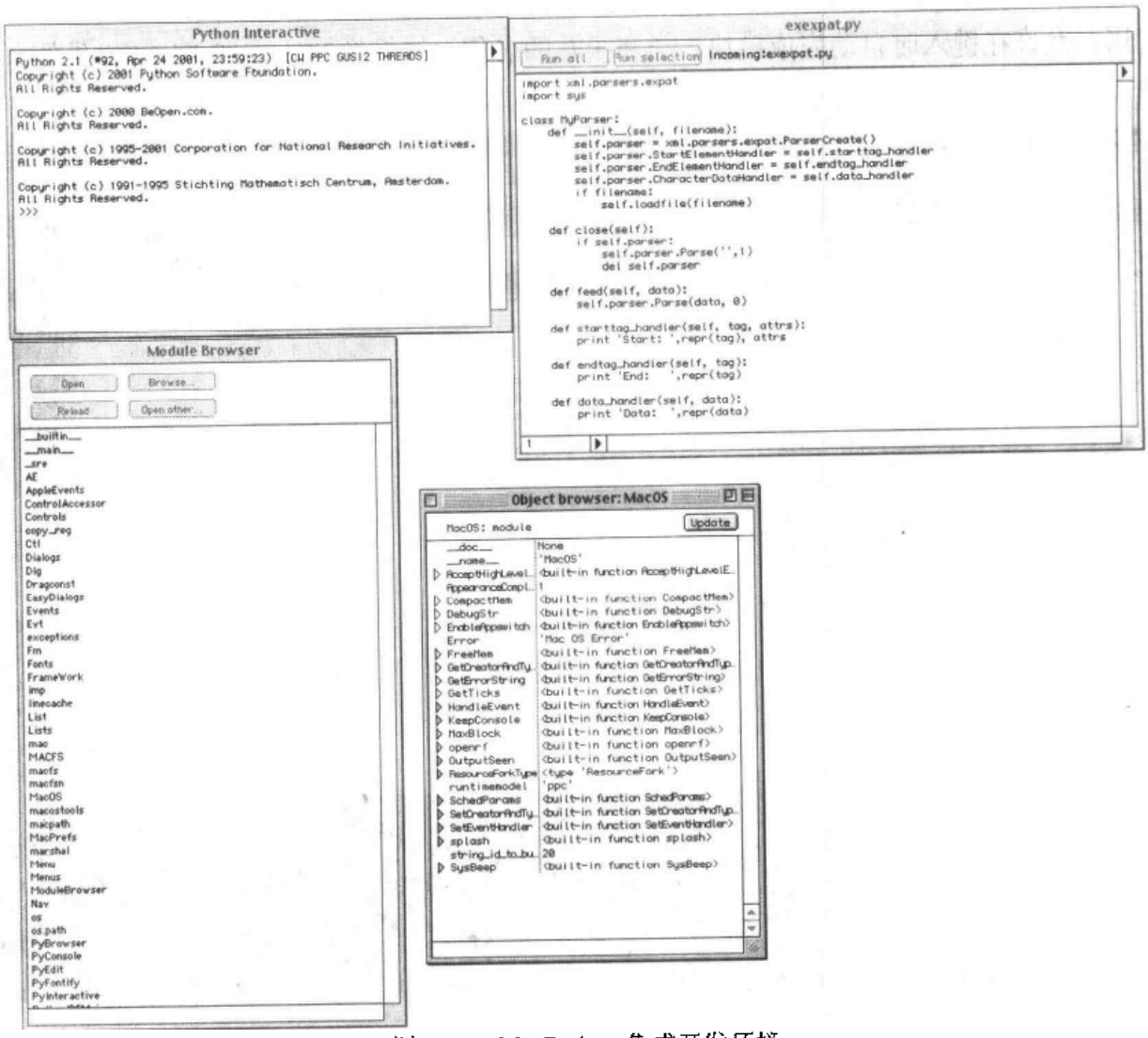

图 17-3　MacPython 集成开发环境

17.1.4　Komodo

Komodo 是 ActiveState 公司提供的商业性 IDE，与其他主要用于 Windows 平台的开发工具同样出色。ActiveState 公司的开发人员还曾经开发过 Perl to Windows、ActivePython 产品，以及其他 Perl 和 Python 工具。Komodo 本身可以用于 Windows、Linux 和 Solaris，不过必须获得 ActiveState Perl、Python 和 Tcl 的发布程序，以便在这一 IDE 中使用。

Komodo 通过提供适应多种语言的交互式的开发环境，以努力满足现代程序员的要求。Komodo1.0 支持 Perl、Python、JavaScript、XML 和 Tcl，拥有工程管理器、编辑器、调试器和运行环境。此外 Komodo 还支持对 HTML、XML、XSLT、C、C++、C#、Java、JavaScript、Lua、Makefile、PHP、Pascal 以及普通文本的智能编辑功能。除了一般的缩进、格式编排和代码高亮显示外，还可以在工作时标记可能出错的代码。就像 Microsoft Word 一样，Komodo 会用红下划线标出有疑问的区域，同时用户仍然可以继续键入，稍后再转回来进行修改。

正如图 17-4 所示，Komodo 提供了非常漂亮的界面，它的基于 Mozilla 浏览器的包则出自 Netscape 开发团队。浏览器也许不像是开发环境的首选，但是 Mozilla 的许多核心组件实际上正使得它成为好的界面。为了在交互环境中支持更多语言，Mozilla 也允许 Komodo 使用扩展和插件。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

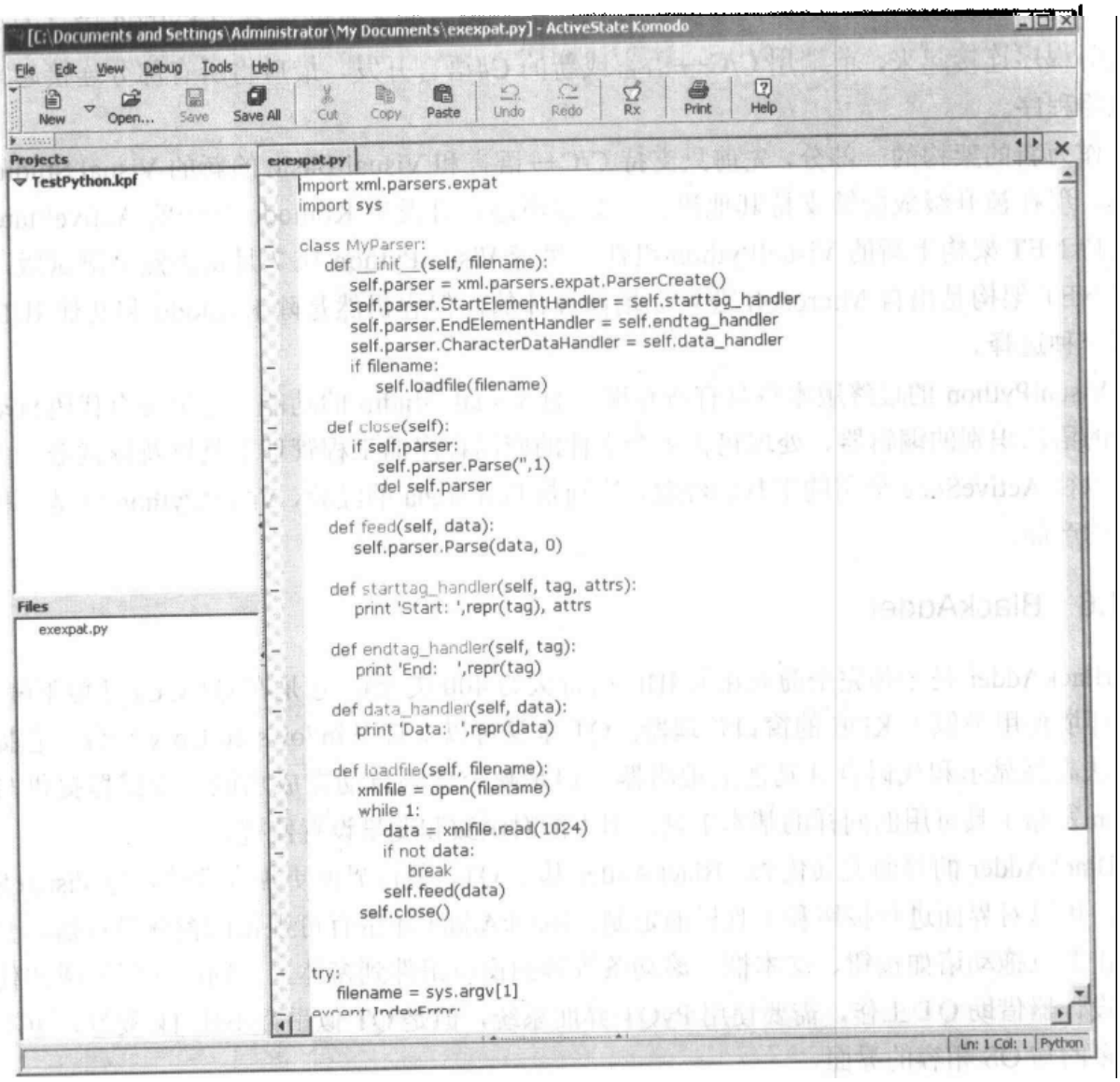

图 17-4　使用 Komodo

不幸的是，Komodo 是一种商业产品。尽管可以获得免费的许可号，但那只是用于非商业用途的。Komodo 的完整的开发许可号售价 295 美元，带有免费的升级，有一年的使用期，为了能够持续使用，每年都需要重新购买，还必须取得 ActivePython 和其他需要使用的语言的开发许可。这种情况可以理解：ActiveState 是一家商业性公司，它需要付费给 Mark Hammond(Python，PythonWin 的作者)和 Gurusamy Sarathy(Perl)等人，来开发不同语言的 Windows 扩展。

不过，尽管 Mozilla、Perl、Python、Tcl、XML 标准是免费的，但是使用 ActiveState 公司的工具的花费还是很大的。对于作者而言，由于 Komodo 提供更加清晰易懂的界面，所以使用 Komodo 进行 Windows 平台下的开发，但在 Unix/Linux 平台下则使用 EMACS 和 Interactive 公司的工具，在 Mac 平台下使用 BBEdit 或 Pepper。

17.1.5　Visual Studio/VisualPython

Microsoft 公司的 Visual Studio.NET 主动采用了大量创新性的特性，包括支持新型的媒介语言，这种语言的应用程序或多或少可以被编译成其他语言。应用程序中所有用这些语言编

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

写的组件都可以被共享。例如，可以将用 Perl 语言编写的文本处理器与 Python 语言编写的网络服务程序连接起来，前端用 C/C++语言或新的 C#语言开发，最后将所有部分连接起来创建出应用程序。

作为新的架构的一部分，先前只支持 C/C++语言和 Visual Basic 的新的 Visual Studio 开发环境，现在被升级成能够支持其他语言。如前所述，开发出 Komodo 系统的 ActiveState 公司也支持.NET 架构下新的 VisualPython 组件。尽管 VisualPython 在 7 月初仍然是测试版，而且许多.NET 架构是出自 Microsoft 公司立场而设计的，但它仍然是除 Komodo 和其他 IDE 之外的另一种选择。

VisualPython 的最终版本将具有所有现有的 Visual Studio 的功能，包括带有代码自动完成功能的语法识别的编辑器、处理包含多个文件的应用程序的工程管理工具以及调试器。由于吸取了以往 ActiveState 公司的工具的经验，特别是 Perl arena 的经验，VisualPython 将是一种值得关注的产品。

17.1.6　BlackAdder

BlackAdder 是一种完全商业化的 IDE(售价大约 400 美元)，它是在 QT GUI 环境下编写的，这一环境使用类似于 KDE 的窗口管理器。QT 本身可以支持 Windows 和 Unix 平台，它提供带有语法高亮显示和代码合并功能的编辑器，但是缺少代码自动完成功能。调试器提供与标准 Python 发布工具可用的同样的基本工具，但不能对函数和变量设置监视。

BlackAdder 的界面尤其优秀。BlackAdder 基于 QT，使得界面更加专业化，与 Visual Studio 一样，可以对界面进行按钮和工具栏的定制。BlackAdder 还带有可视化的窗体设计器，允许用户单击并且拖动诸如按钮、文本框、滚动条等等的窗口组件到窗体上，同时获得现成的代码。窗体设计器借助 QT 工作，需要使用 PyQT 界面系统，但是 QT 似乎并不比 Tk 要好，也不能提供更多的与 OS 相容的界面。

然而 BlackAdder 并非没有问题。它的界面尽管功能强大，但并不是完美的，一些古怪的问题会使人无法忍受。而且，不使用调试器就无法执行脚本，因此不能快速地运行应用程序并返回到编辑器。IDE 也偶尔会锁死，即使文件的确是有效的 Python 文件，编辑器也会莫名其妙地认为要加载的文件是无效的 Python 文件。

17.1.7　WingIDE

WingIDE 是另一种商业产品，它定位在与 Komodo 同一层次上。WingIDE 是支持多种语言的 IDE，包括带有命令自动完成和语法高亮显示功能的编辑器，以及用来直接定位到模块中各种函数的定义部分的工具，十分好用。还有用于在整个模块中定位实体的定义和文本串的模块浏览器，以及虽不带监视功能但性能不错的调试器。

WingIDE 当然是一种相对较好的工具，但有时会出现“窗口泛滥”现象——屏幕上充满了太多的窗口而无法对其进行控制。WingIDE 还有和 IDLE 同样的缺陷：一个简单的脚本就会锁死整个应用程序。不过，作为一种重要的开发环境，WingIDE 还是很有前途的。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

17.2　Python 标准库

除了解压缩和编译最新的 Python 发行版本外，您真正看过内容吗？Guido 和其他开发团队做了一件很好的事情：提供了丰富的扩展库的组合，其中的很多例子都已经介绍过了，此外还有大量范例、样本和附加的工具。

剽窃别人的成果是不可取的，但是参考他人代码常常能够很快解决特殊的问题。文档也同样有用，但也只限于查看不同的可用模块的范例和样本。如果他人已经做了很多有用的工作，就没有必要再从头做起了。

本节将对标准 Python 发布做快速的介绍，包括其示例和演示，还将介绍为了支持基本包中不支持的附加的扩展和模块的信息，如何在标准模块中设定和配置 Python 发布。

17.2.1　演示、范例和样本

要掌握更多信息，首先要从特定的模块开始。例如，如果想知道如何使用 IMAP 库，可以查看“imaplib”模块，该模块的尾部会给出一个测试函数。可以将模块中的测试函数作为自己脚本的基础。除了诸如“sys”和“os”等的基本模块之外，其他提供许多扩展的模块也都有测试部分。

在所有操作系统平台上的标准 Python 版本中，都能找到下列内容的演示和范例：

- 类(Demo/classes)：如何建立类、处理类的继承和重载运算符的范例。所有这些范例都提供对特殊数据类型操作的类，包括有理数、矢量和日期。
- Curses(Demo/curses)：如何使用基于文本的 Curses 系统创建界面。除了实用性外，大多数脚本还很有趣，十分有助于说明如何建立基于 Curses 的应用程序。此外，此目录下还以“life.py”的形式提供有用的转换器，以及以“xmas.py”的形式提供更多的周期性的演示。
- DNS 工具(Demo/dns)：DNS 客户端模块和一些范例脚本，用于从 DNS 系统检索信息。
- 远程过程调用(Remote Procedure Call，RPC)(Demo/rpc)：RPC 系统是用于从客户端调用函数的工具包，而被调用的函数实际上是在远程服务器上执行的。例如，使用 RPC，不用实际登录到服务器，就能查询远程服务器的状态，还可以检查信息。这部分的脚本提供了到 Sun RPC 系统的接口，如果要处理需要被串行化与其他系统进行交互的数据结构，这些接口将很有用处。

RPC 最初的格式是特定于平台和 OS 的；甚至当使用 XDR(eXternal Data Representation，外部数据表示)时，也会发生问题。SOAP(Simple Object Access Protocol，简单对象访问协议)和 XML-RPC 使用 XML 来本地化请求和响应，允许在任何平台上使用 SOAP 和 XML-RPC。

- 范例脚本(Demo/scripts)：汇集了各种脚本，能完成从因数的分解和整理收件箱，到用 Morse 编码从字符串中创建音频文件这些工作。
- 套接字(Sockets)(Demo/sockets)：套接字编程不仅是打开连接、发送数据和关闭连接。如果想与远程客户机实际交换信息，需要处理诸如线路终止问题，并要开发出通信的协议。此目录下提供的这些例子，可以使用套接字建立客户端和服务器端，同样也可以进行核心的网络操作，如进行广播甚至是 Telnet 协议的客户端。
- 线程(Thread)(Demo/threads)：回过头看看第 9 章中的线程部分。一旦理解这里所表达的

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

基本思想，就会觉得线程很简单。这里还有 Tim Peters 编写的有用的例子，可以进行乘法运算，通过使用线程获得结果。

• Tkinter(Demo/tkinter)：从基本的应用程序到菜单，甚至是 Unix 的页面查看器，这里都有。还可以得到经过改进的、完全用 HTML 写成的 Web 浏览器。

• XML 处理(Demo/xml)：如何分析和处理 XML 文档。这里的大部分信息都能用于对一般的、基于标记的语言的处理，包括 XML、HTML 和 SGML。在第 20 章中将介绍标记性语言的处理。

此外，还可以查看工具目录下的应用程序，它包括：

• Python 编译器(Tools/compiler)：如果查看此模块，就会对 Python 解释器的内在工作机制有所了解。此模块提供与内在操作的一般接口，允许在脚本中进行筛分和查找信息。例如，工具目录下的“demo.py”脚本可以遍历应用程序，提取出类和方法的列表。

• FAQ 向导(Tools/faqwiz)：一个基于 Web/CGI 的接口，用于建立和管理 FAQ(Frequently Asked Questions，经常被问到的问题)文档。接口提供了用于浏览和搜索的读者(终端用户) 接口，以及用于建立多层结构的管理接口。

• Python Freezer(Tools/freeze)：Python 工程中的“freeze”功能使得 Python 源程序具有可执行能力，可以建立能够发布给用户的、单独的、可执行文件，而用户却不用拥有 Python 解释器。如果在 Mac OS 下创建应用程序，可以使用 Python 发布中的“BuildApplication”小应用程序(Applet)。

• IDLE(Tools/idle)：在本章的前面已经介绍过，IDLE 是一种完全由 Python 驱动的 IDE。如果查看它的代码，会获得许多有关如何开发基于 Tk 的接口，以及如何执行和调试 Python 脚本的信息。

• Web 检查器(Tools/Webchecker)：Web 检查器工具遍历目录树或 URL，从文档中提取出文档的链接，并决定这一站点是否具有链接或其他的问题。它是遍历站点的很好的工具，可以进行合理的检查，但并不完全可靠，对于使用动态元素的站点，Web 检查器就无法正常工作。这个脚本给出了很好的背景，可以处理 HTML 以及访问远程站点。同时还给出了关于如何结构化嵌套的对象的例子，在此工具中记录不同页面的 URL 和错误信息。

如果没有找到合适的内容，可以查看 Vaults of Parnassus (参见后面部分)以及附录 B 的 Python 资源列表。

17.2.2　标准模块

Python 拥有大量的、包含所有内容的标准模块集，从与 OS 的基本接口到不同的因特网协议，还有文本和 XML 的处理系统。尽管肯定会使用到其中的一些模块，但仍然值得全部查看一遍这些被提供的模块，或许会有模块能够为您的问题提供现成的解决方法。

在别处已经介绍了大部分模块，本章后面还将介绍由 C 语言扩展支持的模块的列表。要查看 Python 支持的全部的模块列表，请参见附录 B。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

17.2.3　配置标准扩展

如果正在编译自己的标准发布，或许需要修改扩展模块的安装，来实现额外的功能。例如，标准 Python 发布提供必要的与 Tk 集成的接口，称为 Tkinter，但是默认状态下它是被禁用的。

可以使用 Python 发行版本中模块目录下的“setup”文件，来配置所需要包括的扩展。该文件是一种标准的文本文件，与“Makefile”格式类似。为了启用不同的扩展模块，需要取消注释行，有时还要提供模块所需要的库和文件所在的位置。例如，为了启用 GDBM 支持，就必须取消以“gdbm”开始的行的注释并且对其进行修改，如下所示：

```
# Anthony Baxter's gdbm module.      GNU dbm( 3) will require -lgdbm:
#
# First, look at Setup. config; configure may have set this for you.

gdbm gdbmmodule. c -I/ usr/ local/ include -L/ usr/ local/ lib -lgdbm
```

在创建处理过程中使用了“configure”脚本，这一脚本会预先配置许多模块，其他的则需要手工配置启动。

模块的确切列表随版本不同而不同，但是为便于参考，表 17-1 列出了通过这个文件配置的扩展模块，以及对这些模块功能的描述和对于启用这些模块有用的说明。

表 17-1　Python 发布中的可配置模块

模　块	说　明
Posix	提供 POSIX(Unix)系统调用。在兼容 POSIX 的操作系统上自动启用，包括 Windows。不能直接使用此模块，而应使用“os”模块，它提供与被支持的“posix”模块函数的链接
Sre	Fredrik Lundh 开发的 Secret Labs 正则表达式库
readline	提供对 GNU 的“readline”库的支持，它提供逐行编辑、输入、命令行历史，以及其他对于开发基于行分析输入或是使用行输入的应用程序有用的扩展。记住，“readline”由 GNU 通用公共许可(General Public License，GPL)包含，如果要提供某一版本的静态链接“readline”库的 Python，就必须遵守 GPL 许可。也可以动态地链接“readline”库或以二进制形式支持 Python 源代码。要获取更多信息，请参见 Python 文档
Array	支持固定数据类型的数组对象
Math	附加的数学功能(sin(),cos(),tan())。使用标准 C 数学库，通常默认可用
Cmath	附加的数学功能(sin(),cos(),tan())，支持复数。使用标准 C 数学库，通常默认可用
Struct	支持打包和拆分二进制结构，如由 C/C++的“struct”语句创建的结构
Time	标准时间访问和格式化函数。默认可用
operater	允许使用名称访问能够支持内部运算符的函数，如使用“add(a+b)”表示“a+b”
weakref	基本的弱引用支持
codecs	用于提供对于内置编码解码器及其编码解码器注册的访问，这些编码解码器用于将文本字符串在 Unicode、ASCII 以及其他字符格式之间进行转化

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

模　块	说　明
testcapi	Python 的 C 语言 API 测试模块
unicodedata	到 Unicode 注册数据库的接口，允许按名称查看 Unicode 字符，也用于将 Unicode 字符串转化成为 Unicode 字符描述
Locale	访问 ISO 的 C 本地支持函数，用于控制字符串排序算法和消息输出语言。
Fcntl	与 fcntl 和 ioctl 系统的文件和 I/O 控制接口。大多数平台支持
Pwd	通过 OS 的 pwd 调用，提供对 Unix/etc/passwd 数据库的访问。只在 Unix、BeOS 下，通过 Win32 和 MacOS X 上的“cygwin”被支持
Grp	通过 OS 的 grp 调用，提供对 Unix/etc/group{s}数据库的访问。只在 Unix、BeOS 下，通过 Win32 和 MacOS X 上的“cygwin”被支持
Errno	POSIX 错误码值和相应字符串。只在兼容 POSIX 的操作系统下被支持
select	使用 select()系统从不同的源进行多路 I/O 的模块。只被 Unix 的一些版本支持(不是所有的)
mmap	与 Unix 的 mmap()系统的接口，用于直接映射内存。在 Unix 和 Win32 下被支持
xreadlines	动态读入行库支持
socket	与 OS 的 socket()网络通信的接口。安装文件提供几行适合非 SSL 以及 SSL 支持的代码，但是需要有 ssl 和 crpto 库来使之工作。非 SSL 变量能在所有实现标准 BSD 的 socket 系统的平台上工作。Unix 也允许对 SSL 的支持
crypt	Unix 下对于 crypt()系统调用的接口，用于加密用户密码。在 Linux 和 FreeBSD 下的版本需要启用“crypt”库
nis	支持 Sun 的网络信息系统(Network Information System，NIS)，但不支持 NIS+
termios	支持 POSIX tty I/O 系统。只限于 Unix 的一些版本
resource	处理对于用于限制资源使用的“rlimit”系统的接口。只在特定的 Unix 版本下被支持
audioop	一般的音频处理库，应当在所有平台下被支持
imageop	一般的图像处理库，应当在所有平台下被支持
rgbimg	一般的读写 SGI RGB 图像文件的库。尽管支持的文件格式特定于 SGI，该库应当在所有平台下被支持
tkinter	与 Tk Gui 创建器的接口。需要先安装 Tcl 和 Tk 库，可以从 Scriptics(http://www.scriptics.com)下载这些库。必须先编译 Tcl，再编译 Tk。也可以添加对 Python Image Library(PIL)的支持
rotor	Enigma 机器加密算法
syslog	与 syslog 系统登录看护程序的接口。注意 syslog 是 Unix 特有的
dbm	与标准 DBM 库实现的接口。需要使用 dbm 或 ndbm 库。只对 Unix 有效
gdbm	与 GDBM 数据库的接口。需要 GNU 的 gdbm 库
zlib	与用于读写使用 gzip 或压缩工具压缩的文件的接口。需要 zlib 库，可以从 http://www.cdrom.com/pub/infozip/zlib/下获得

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

模　块	说　明
pyexpat	Expat XML 分析器。需要从 James Clark，http://expat.sourceforge.net 下载 expat 库。提供对于 xml.parsers.expat 模块的接口
bsddb	对于 BSD db 数据库的支持。注意目前只支持老的 v1.85 版本的 BSD 库。支持新的 v3.x 版本库的版本可以在 http://electricraint.com/greg/python/bsddb3/中得到

可以在 Python 主目录(常常是/usr/local/lib/python#.#)下的“Config”目录中，找到模块的当前副本。为了进行实际的改动，需要修改源代码版本中的“Setup”文件，再运行“Modules”目录下的“make”命令，或上一级目录中的“make”命令来重新生成 Python。

单个的模块和扩展建立后将会被并入主要的 Python 的库中。如果有动态的 Python 安装，模块就会为动态加载而重建。对于静态的平台，将会重建主要的 Python 库，然后 Python 的可执行程序将会重新链接到新的库。

17.3　Vaults of Parnassus

Vaults of Parnassus(http://www.vex.net/parnassus/)是一个站点，起初致力于整理所有发给 Python 新闻组(comp.lang.python)的模块、范例和扩展的邮件。从此它成为 Python 的新模块和扩展的焦点，也成为单独使用或联合其他工具使用 Python 的范例脚本和演示应用程序的资源。

Vaults 很适于展示特别的模块或扩展程序。例如，能够在 Vaults 上寻找与 MySQL 或 PostgreSQL、XML 分析器以及其他许多扩展的接口。Perl 程序员会发现所能找到的模块的范围比 CPAN 小一些，但作者认为这很容易用事实来反驳：Python 标准库已经包括许多模块，不这样的话就必须从 CPAN 下载这些模块。

然而，这并非是说它们并没有寻求新的提交。如果您拥有认为对其他人有用的模块，就请递交给 Vaults。

17.4　Zope 与 Jython

如果正在开发 Web 应用程序，可以通过使用 Zope 节省大量开发时间。Zope 发布对象，所有需要考虑的就是建立表达信息的类和方法。Zope 替您处理所有的 CGI 和数据库访问。根据经验，设计信息从数据库输入和输出的机制会占去整个应用程序大约 40%的开发过程时间，由此可以看出使用 Zope 的好处所在。

Jython 从不同的角度解决了同样的问题。Jython 是一种完全的 Python 解释器，但它是用 Java 写成。Jython 允许用 Java 创建 Python 应用程序，使其能在没有 Python 的客户端机器上执行，而且可以在同一个应用程序中混合 Python 和 Java 元素。

要想获得更多信息，请参见第 21 章有关部分，以及其他由 Python 写成或提供支持的 Web 解决方案。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第18章　发布Python模块

一旦完成了 Python 工程，无论是完整的应用程序还是简单的模块，都需要将其打包成某种格式，以便发布给他人。

有两种基本的终端用户类型：公共的和专有的。对于公共的版本，可以为公共的用户创建安装包，要么是免费，要么是公布源码工程和某一共享软件，或者是商业产品。此外，还可以创建安装包，它只能被有选择的数量的用户安装，或者作为定制(而不是大量生产的)的商业应用程序或是另一个大工程的一部分。例如，可以将 Python CGI 脚本作为网站的一部分提供。

对于专有的解决方案，借助于创建定制的安装文件，或是在一定程度上用手工进行安装，可以解决问题。实际上，当提交高度定制的解决方案时，很可能这是惟一的选择。

对于单独的应用程序，会有更多的灵活性。尽管在很大程度上，具体的解决方案会依赖于安装包的工作方式，但很可能的是，用户只需要将程序复制到计算机上的某一文件夹或目录就可以完成了。对于更复杂的安装，需要编写某种形式的安装脚本。这里存在第一个问题。首先，并非所有人都在“/usr/local”或“C:\Python”目录下安装 Python，所以必须考虑目标文件夹之类的问题，然后必须考虑 Python 本身的安装位置。而且情况还要更加复杂，但是如果需要在 Mac 上也能支持应用程序，可以使用不同的办法(尤其是目前还存在 OS 的 Mac OS 和 Mac OS X 等版本，其工作机制还十分不同)。

如果提供扩展 Python 功能的模块和扩展程序，情况将会更加复杂。对于标准(纯 Python)Python 模块，所有要做的只是将这一模块复制到正确的位置。寻找正确的位置就是个问题。当涉及到 C/C++扩展时，还需要在安装前编译，除了位置问题外还有一系列新的问题会出现。尽管存在可以在不同平台上编译扩展的技术(参见第 26 章)，但是它们都依赖于对在每台机器上都是独一无二的一个文件的访问。

实际上，很长一段时间来，许多这样的问题都被忽略了。对于熟悉 Perl 的开发人员，这或许是一个打击。相当早之前，Perl 就有了 CPAN(Comprehensive Perl Archive Network，综合 Perl 存档网络)以及相配的模块，它可以与这一语言的模块和扩展连接或进行下载。CPAN 本身是依靠一种称为“MakeMaker”的工具，它使用 Perl 脚本，写入了配置信息以建立传统的 Makefile。这种 Makefile 负责建立并最终将模块安装到 Perl 的发布中。

Python 拥有 Vaults of Parnassus，它提供 Python 模块和扩展的主要的知识库，但是仍然会需要手工下载包，经常还要进行手工安装。

今后这种情况将会好转。去年，由 dist-sig(special interest group，专门兴趣组)开始致力于开发出被称作“distutls”(Python Distribution Utilities，Python 发布工具)的系统。它是单独模块的包，配套使用时，只需要运行一个 Python 脚本，就能够在用户的系统上产生必要的步骤来建立模块和扩展，必要时还会将模块和扩展安装到用户的系统上。

直到 2001 年 5 月，此系统还在不断地进行开发，由于它所处的开发状态，本章特意避免深

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

入此系统的细节。要获取本书发行后此包的新的开发情况，请查看作者的网站(http://www.mcwords.com)或是 Python 网站(http://www.python.org)。

本章将简要地介绍一下此系统及其基本的工作机制，以及在未来的版本中可能实现的功能。

18.1　使用 distutils

完整的 distutils 系统使用一个简单的文件“setup.py”来工作，它包括将要安装的模块的所有信息。当用户下载包以后，只需要执行“setup.py”并给出命令来安装模块，所有其他的事情都会自动完成。

例如，给出简单的“setup.py”文件如下：

```
from distutils. core import setup
    setup( name=" mymodule",
           version=" 1.0",
           py_ modules=[" mymodule"])
```

通过使用下面的语句，用户可以安装上模块“mymodule”：

```
$ python setup. py install
```

这实在很简单！

真正的力量在于后端的“distutils”包以及编写“setup.py”文件的方式。

18.1.1　可支持的模块

“distutils”包被设计成用于处理以下三类基本的模块类型：

- 纯 Python 模块：这是完全用 Python 编写的模块，采用源文件的形式(.py)，还可以是.pyc 和.pyo 形式的文件。
- 扩展模块：这是 C/C++的扩展。通过 Java 类来支持 Jython 扩展的工作还在进行中。
- 包：某些 Python 模块被集合起来放在一个目录结构下。想了解有关包的更多信息，请参见第 5 章。

18.1.2　编写 setup.py

“setup.py”脚本是整个处理过程的核心。本质上讲，脚本能够做任何想做的事情。例如，能够向用户询问关于模块或应用程序的配置或其他信息的问题，但是最终整个过程只在对函数 setup()的单个调用中被处理，它是从模块“distutils.core”中被导入的。

setup()的参数被作为关键字参数的列表被提供，包括元数据和配置信息的混合体。例如，在前面的例子中，元数据包括版本号。其他被支持的元数据范围包括描述、作者、电子邮件地址和 URL，以及在任何时间被添加的更多的信息。

配置信息包括将要安装的模块和包的列表、扩展源文件的名称，甚至包括需要用于建立特定扩展的预处理器和链接器的信息。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

例如，当为安装 C 扩展编写安装脚本时，可能会如下这样编写：

```
from distutils. core import setup, Extension
setup( name=" mymodule",
        version=" 1.0",
        description=" Demonstration for C extensions",
        author=" Martin C Brown",
        author_ email=" mc@ mcwords. com",
        ext_ modules=[ Extension(" mymodule", [" mymodule. c"])])
```

Extension 类用于创建新的扩展实例，使用被提供的参数构建细节，用于创建扩展以及建立扩展所要求的源文件。Extension 的其他参数包括：include_dirs，用于目录的列表，在编译时使用；define_macros 和 undef_macros，用于在创建时定义和取消定义 C 的宏；以及 library_dirs 和 libraries，用于指定需要用于创建最终扩展的库模块。

18.2　未来的特征

我们仅仅对 distutils 现在的功能和此系统长期的计划进行了讲述。就在编写本章内容的时候，distutils 中加入了更多的特性。如果注意 dist-sig 邮件列表上的信息的话，可以看到，distutils 中每天都在添加新的特性、建议和扩展。希望到 2001 年底，distutils 包将会成为发布和安装所有 Python 扩展的标准系统。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第 4 部分　Web 开发

第19章　Web开发基础

很难想像，仅仅在几年前，世界还是一个没有因特网和 World Wide Web 的世界。早在 1993 年，作者就开始为网站编写 HTML(Hypertext markup Language，超文本标记语言)页面，当时距离发明 HTML 只过了很短一段时间。人们不需要花费太长时间就能了解 HTML 的能力，以及 HTML 除了为静态页面提供服务外的更加强大的动态能力。

本章在介绍用 Python 建立动态网站的细节之前，将会介绍生成 HTML 和使用 URL 的基础知识。本章还将介绍 cookies 的用法，它能够在用户的浏览器中存储信息。本章还对网站的安全性问题进行探讨，在为网站编写脚本时，需要注意这些知识。

在本章其余部分，将会介绍如何分析 HTML、XML(Extensible Markup Language，可扩展标记性语言)和 SGML(Standard Generalized Markup Language，标准通用标记性语言)文档，以便从中提取信息和数据。在第 22 章中还会介绍其他更高级的使用 Python 的网站开发解决方案。

19.1　编写 HTML

在直接介绍用 Python 编写动态脚本之前，有必要介绍一下如何编写页面，以及 Web 环境工作的机制。将从 HTML 开始。HTML 是一种在文本文档中标记不同元素来将文本文档格式化的语言。

如果没有 HTML，网站将会陷入困境：没有办法高亮显示或是格式化不同的元素，也没有办法将图像或链接合并入其他文档中。

HTML 通过在文本中使用“标记符(tag)”来工作。这些标记符对浏览器等的 HTML 查看器定义文本是如何在屏幕上格式化的。例如，使用“”标记符加黑一条语句：

```
<b> Hello World!</ b>
```

开头的标记符“”开始进行黑体字显示，“” 标记符结束黑体显示。其他的标记符包括建立表、改变字体、文本大小以及那些使用图像和“锚”的标记符，它提供与外部文档的链接，这使得这种语言体现出超文本元素来。

不过由于本书是关于 Python 的——如果想学习更多的关于 HTML 的知识，可以查看 World Wide Web Consortium 网站(www.w3c.org)，它包含所有基于 Web 的标准的指导和参考手册，包括 HTML。也可以查看 Thomas Powell(Osaborne/McGraw-Hill，2000)所著的《HTML 完全参考手册》(HTML: The Complete Reference)。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

在 Python 中，生成 HTML 十分简单，就像对一个打开的文件使用“print”语句或“write()”或其他方法一样。记住，HTML 只是一种标记性原始文本：它不是二进制文件格式，也没有严格的规则用于指定怎样输出信息。例如，我们能在 Python 中用下面的语句输出前面的 HTML 例子：

```
print "< b> Hello World!</ b>"
```

这种方法可以用于任何数量的 HTML，大部分 HTML 的开发方法几乎都采用这种方法。Python CGI 脚本中也能使用这种技术。

然而，使用这种方法并不是在所有情况下都是理想的。对于开始，输出的内容没有结构性：如果忘记打印黑体结束标记符，那会怎么样？更糟的是，忘记写上“</table>” 的结束符，会造成整个表格都消失。要避开这个问题，还没有快速的方法，只能对开发的 HTML 多加注意。

不过也可以通过使用“HtmlKit”模块简化这一过程。“HtmlKit”模块(可以从 http://www.dekorte.com/Software/Python/HtmlKit/中获得)提供创建大部分需要的标记符的方法，它使用标准的和结构化的接口，有助于生成有效的 HTML，而不用强迫人工检查编写的每一行 HTML 代码。

例如，使用“HtmlKit”在 HTML 中生成表，如下所示：

```
import HtmlKit

table = HtmlKit. Table()
row = table. newRow()
row. newColumn( 'Name')
row. newColumn( 'Title')
row. newColumn( 'Phone')
row = table. newRow()
row. newColumn( 'Martin')
row. newColumn( 'MD')
row. newColumn( '01234 567890')
print str( table)
```

可以看到，这里建立了一个“table”对象。首先作为“HtmlKit”中定义的“Table()”类的一个实例创建“table”对象，然后加入行，再加入列来创建表结构。一旦完成后，通过函数“str()”对“table”对象求值就能生成最终的表。上面的代码会生成如下所示的 HTML：

```
<table border=" 0" cellpadding=" 3" cellspacing=" 1" width=" 100%">
<tr>
<td nowrap bgcolor="# dddddd"> Name</ td>
<td nowrap bgcolor="# dddddd"> Title</ td>
<td nowrap bgcolor="# dddddd"> Phone</ td>
</ tr>
<tr>
<td nowrap bgcolor="# dddddd"> Martin</ td>
<td nowrap bgcolor="# dddddd"> MD</ td>
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
<td nowrap bgcolor="# dddddd"> 01234 567890</ td>
    </ tr>
    </ table>
```

可以看到，当用“HtmlKit”创建 HTML 时要小心，因为它自动为不同元素加入属性，这些属性或许并不是想要的。当然，在用“HtmlKit”创建表时，这些还是可以被修改的。需要更多信息，请查看有关文档。

19.2　统一资源定位符(URL，Uniform Resource Locator)

您可能已经使用过 URL(uniform resource locator，统一资源定位符)了，但却从未想过它究竟是什么。URL 是一个因特网上资源的地址，包括用到的协议、服务器的地址，以及要访问的文件的路径。例如，地址：

```
http:// www. mcwords. com/ index. shtml
```

表明需要使用 HTTP 协议，正在连接 www.mcwords.com 的机器，要检索文件“index.shtml”。

URL 也能包括进登录名、密码以及可选的服务端口号信息：

```
http:// anonymous: password@ ftp. mcwords. com: 1025/ cgi/ send. py? file= info. zip
```

前面的例子表明下载信息为：服务器是“ftp.mcwords.com”，使用端口“1025”，登录名为“anonymous”，密码是“password”。

这个例子也表明要访问服务器上名称为“send.py”的对象。问号后面的信息是作为请求的一部分而发送的数据。这个例子中，向“send.py”脚本发送一个键/值对。后面还将介绍如何处理这一信息。

分析 URL

如果要分析一个现有的 URL，要么提取出单独的组件，要么将相对 URL 和绝对 URL 连接起来，这就需要 Python 发布的“urlparse”模块。

该模块只提供三个函数：urlparse()、urlunparse()和 urljoin()。urlparse()函数接受一个 URL，返回一个定义了 URL 不同组件的元组。该函数基本的格式如下：

```
urlparse( urlstring[, default_ scheme[, allow_ fragments ] ])
```

urlstring 是需要作为 URL 进行分析的字符串的名字。例如，前面那个例子 URL 的组件的元组可以这样获得，如下所示：

```
>>>
urlparse. urlparse( 'http:// anonymous: password@ ftp. mcwords. com: 1025/ cgi/ send. p
    y? file= info. zip')
    ('http', 'anonymous: password@ ftp. mcwords. com: 1025', '/ cgi/ send. py', '',
    'file= info. zip', '')
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

返回的元组的格式为：

```
(scheme, host, path, parameters, query, fragment)
```

注意用户和密码没有提取出来，必须自己来提取，元组依次排列为：

```
scheme:// host/ path; parameters? query# fragment
```

当遇上不完全的URL(比如www.mcwords.com)，应当使用“default_scheme”变量，而且需要用给定的模式(协议)来转化这一URL。例如，www.mcwords.com这一URL实际上是一个Web地址，因此它必须用“http”模式来分析。某些模式不支持给定URL的所有元素，例如，FTP模式化的URL既不支持参数也不支持查询。

“allow_fragments”变量指定所提供的URL中是否允许分段。设置这个选项将会覆盖所提供的模式的设置。

“urlunparse()”方法接受与“urlparse()”返回的元组同样顺序的元组，它将给定的元素重新建立，生成单个的URL字符串。

```
>>> urlunparse(( 'scheme', 'host', 'path', 'parameters', 'query', 'fragment'))
'scheme:// host/ path; parameters? query# fragment'
```

“urljoin()”方法接受两个参数，一个绝对的URL和一个相对的URL，再将这两个URL连接成一个绝对URL。例如，作者网站的URL是“http://www.mcwords.com”，有关本书的内容所在的页面是“/projects/books/pytcr”。能够生成一个完整的URL指向这本书，如下所示：

```
>>> urljoin( 'http:// www. mcwords. com', '/ projects/ books/ pytcr')
    'http:// www. mcwords. com/ projects/ books/ pytcr'
```

对于更复杂的URL，它也同样有用。例如，《Perl完全参考手册》(《Perl Complete Refernce》)的页面在其站点的内部，为“http://www.mcwords.com/projects/books/pcr2e”，从这里指向本书所在内容的页面，其位置是“../pytcr”，可以将其连接起来，如下所示：

```
>>> urljoin( 'http:// www. mcwords. com/ projects/ books/ pcr2e/ ', '../ pytcr')
    'http:// www. mcwords. com/ projects/ books/ pytcr'
```

提示：

如果需要下载由URL定义的文件，应当使用“urllib”模块。更多信息请参见第13章。

19.3 使用CGI的动态网站

CGI(Common Gateway Interface，公共网关接口)，是一种用于在Web服务器软件和外部脚本之间交换信息的方法。当从浏览器向Web服务器提交表单时，填写的数据通过导管提供给外部的脚本，随后脚本提供的任何输入都被Web服务器沿原路返回到用户的浏览器中。

CGI定义了Web服务器与外部应用程序会话和执行外部应用程序的方法，以及用于在外部

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

脚本和服务器本身之间传输信息的方法。顺便提一下，外部脚本可以用任何语言写成。在这里当然使用 Python。与 AppleScript、C/C++应用程序或是 Java Servlet 相比，使用 Python 一样很简单。实际上有两种在 Web 服务器和需要调用的 CGI 脚本之间交换信息的方法。这两种方法是 GET 和 POST，使用哪种方法取决于需要传输的信息。GET 方法对于短小的请求很有用，优势在于字段和数据能够被附加到 URL 上。POST 方法最好用于大量的文本。下面将介绍这两种方法是如何影响 CGI 一边的。

对于提交数据给 CGI 脚本和取回信息的过程，这里进行了详细的描述：

(1) 用户浏览器(客户端)打开与服务器的连接；

(2) 用户浏览器(客户端)向服务器发出 URL 请求；

(3) 服务器分析 URL，并决定 URL 是指向静态页面文件，还是指向需要单独执行的有效 CGI 脚本。本例中当然假定属于后一种情况；

(4) 调用外部应用程序。此时任何有效的可执行文件都能使用；

(5) 所有数据(从表单得来的，或是从合适的格式化的 URL 中得来的)都被提供给外部应用程序。如果信息用 GET 请求被发送，那么在脚本必须读取的环境变量中，信息将会有效；如果信息使用 POST 方法提供，数据将被发送给标准的 CGI 应用程序输入；

(6) 现在，CGI 脚本处理信息；

(7) 任何由 CGI 应用程序产生并被发送到应用程序的标准输出的信息，都被 Web 服务器接受，并逐字地被发送回客户端浏览器。有效的应答必须包括 HTTP 头文件，它定义了应答的格式和实际需要返回的 HTML 页面。这两个元素应当用一个空行隔开。

这个流程十分简单，只是用来说明基本处理是如何进行的。在开发 CGI 脚本时，所关心的重点是步骤 4、步骤 5 和步骤 6。

在步骤 4 中，必须考虑应用程序执行的环境。环境定义了需要运行的 Python 脚本的物理和逻辑的限制。此外，除了标准的环境变量如 PATH 外，还有一些 Web 特有的信息。在步骤 5 中，需要提取从浏览器提供的所有信息，要么使用 GET 方法从某个环境变量中提取，要么使用 POST 方法从标准的输入中提取。“GET”和“POST”这些名称，是在向服务器发送表单数据时，从浏览器传输信息的方法，具体方法要根据表单的配置来选用。在步骤 6 中，需要知道如何将信息发送回用户浏览器。

在本章后面的节里，我们还将分别介绍每一种情况。

19.3.1　Web 环境

脚本执行的环境通常不影响脚本的操作，除非在 Python 的一般操作中被着重提出。例如，要执行的 Python 脚本的环境如果定义了可选的 PATH 变量，就会影响脚本将通过“exec*()”函数执行哪些程序。环境基本上不会改变脚本的执行方式，但它会影响脚本自身的某些操作。

大多数 Web 服务器生成 CGI 脚本环境，这一环境带有许多关于客户端及其浏览器的有用信息，以及关于 Web 服务器和文件位置的信息，还有其他一些信息。大多数运行在 Unix 下的 Web 服务器的可用环境变量的列表，可以参见表 19-1。

具体被支持的环境变量的列表，取决于 Web 服务器，也取决于被请求的 URL 的实例。对于那些作为引用结果显示的页面，也有一个“引用”信息的列表，生成了始于此站点对被请求

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

的 URL 的引用。

表 19-1　用于 CGI 脚本的 Web 服务器环境变量

环境变量	说　明
DOCUMENT_ROOT	Web 服务器文档的根目录
GATEWAY_INTERFACE	接口名称和版本号
HTTP_ACCEPT	浏览器接受的格式。这一信息是浏览器首次向服务器请求页面时的可选项。在正文中举出的例子中，默认的接受类型包括所有主要的图形类型(GIF，JPEG，X 位图)，还有所有其他的 MIME 类型(*/*)
HTTP_ACCEPT_CHARSET	浏览器接受的字符集
HTTP_ACCEPT_ENCODING	浏览器支持的特殊的编码格式。本例中，Netscape 支持 Gzip 编码文档，它们将在接收时被快速解码
HTTP_ACCEPT_LANGUAGE	浏览器接受的语言。如果服务器支持，将把特定语言的文档返回给浏览器
HTTP_CONNECTION	任何 HTTP 连接指令。典型的指令是“Keep Alive”，它强制服务器保持 Web 服务器的过程，使相关网络套接字专用于浏览器，直到定义的非活动时期
HTTP_HOST	服务器主机(不含域)
HTTP_USER_AGENT	远程浏览器的名称、版本号和平台。正文中例子的输出中，使用的浏览器是 Macintosh PPC 上的 Mozilla 4.0 兼容浏览器(实际的浏览器是 Microsoft Internet Explorer 5.0)，不要盲目地认为名称“Mozilla”只使用于 Netscape Navigator；包括 Microsoft Internet Explorer 5.0 在内的其他浏览器，也会报告为 Mozilla 浏览器—— 即使所有的浏览器在显示 HTML 时都不同，这仍然有助于兼容性识别
PATH	CGI 脚本的路径
CONTENT_LENGTH	查询信息的长度。只对于 POST 请求有效，有助于生成的脚本的安全性
QUERY_STRING	查询字符串，用于 GET 请求
REMOTE_ADDR	浏览器的 IP 地址
REMOTE_HOST	浏览器的解析名称
REMOTE_PORT	浏览器所在机器的远程端口
REQUEST_METHOD	请求的方法，如 GET 或 POST
REQUEST_URI	被请求的 URI(uniform resource identifier，统一资源鉴别符)
SCRIPT_FILENAME	CGI 脚本的完整路径
SCRIPT_NAME	CGI 脚本名称
SERVER_ADMIN	Web 服务器管理员的电子邮件地址
SERVER_NAME	服务器完整的合法域名
SERVER_PORT	服务器端口号

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

环 境 变 量	说　　明
SERVER_PROTOCOL	协议(通常为 HTTP)和版本号
SERVER_SOFTWARE	使用的服务器软件的名称和版本号。有助于使用利用了多 Web 服务器特性的单个脚本
TZ	Web 服务器的时区

可以获得使用 CGI 脚本的信息，如下所示。这个阶段不必考虑太多脚本的细节部分。

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

•     from os import environ

      print "Content- type: text/ htmlnn"
      print "< h1> Environment:</ h1>"

      for key in environ. keys():
        print "% s =& gt; %s< br>" % (key, environ[ key])
```

在作者的 Web 服务器上，脚本是 Solaris 8 上运行的 Apache 1.3.20，下面是在浏览器(Microsoft Internet Explorer 的 Mac5.01 版本)窗口结束显示的内容：

```
DOCUMENT_ ROOT => /export/ http/ Webs/ test
    SERVER_ ADDR => 192.168.1.1
    QUERY_ STRING =>
    SERVER_ PORT => 80
REMOTE_ ADDR => 192.168.1.1
    HTTP_ VIA => 1.0 http- proxy. mcslp. pri: 8080
    PATH =>
    /usr/ local/ bin:/ usr/ bin:/ usr/ games:/ export/ home/ root/ usr/ etc:/ export/ home/
    root/ usr/ bin:/ usr/ lib:/ usr/ ccs/ bin:/ usr/ sbin:/ usr/ local/ sbin:/ usr/ lib/
    lp/ postscript:/ export/ data/ bin:/ opt/ NSCPcom:/ usr/ openwin/ bin:.
    HTTP_ ACCEPT_ LANGUAGE => en
    GATEWAY_ INTERFACE => CGI/ 1.1
SERVER_ NAME => test. mcslp. pri
TZ => GB
    HTTP_ USER_ AGENT => Mozilla/ 4.0 (compatible; MSIE 5.0; Mac_ PowerPC)
    HTTP_ ACCEPT => */*
    REQUEST_ URI => /pyecho. cgi
    HTTP_ UA_ CPU => PPC
    HTTP_ EXTENSION => Security/ Remote- Passphrase
    SCRIPT_ FILENAME => /export/ http/ Webs/ test/ pyecho. cgi
    HTTP_ HOST => test. mcslp. pri
    REQUEST_ METHOD => GET
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
SERVER_SIGNATURE =>
Apache/ 1.3.20 Server at test. mcslp. pri Port 80

HTTP_IF_MODIFIED_SINCE => Tue, 10 Jul 2001 13: 37: 05 GMT
SCRIPT_NAME => /pyecho. cgi
SERVER_ADMIN => mc@ test. com
SERVER_SOFTWARE => Apache/ 1.3.20 (Unix) PHP/ 4.0.6 mod_perl/ 1.25
SERVER_PROTOCOL => HTTP/ 1.0
REMOTE_PORT => 51868
HTTP_UA_OS => MacOS
```

从这里可以获得很多有用的信息，可以用于自己的脚本。环境变量 SCRIPT_NAME 包含被客户端访问的 CGI 脚本的名称。然而，对于 CGI 程序来说，需要考虑的最重要的字段莫过于是 REQUEST_METHOD 了，它定义了从浏览器通过 Web 服务器向 CGI 应用程序传输信息(请求)使用的方法。

CONTENT_LENGTH 变量定义了使用 POST 方法查询所包含的字节数量。它主要用于确定已经提供了数据(因此也就需要进行处理)。并不是所有 Web 服务器都提供 CONTENT_LENGTH 环境变量，此变量也不是惟一的确定是否发送了查询的方法。然而，如果使用正确，它也能够帮助提高 Web 脚本的安全性。需要更多信息，请参见本章后面的“安全性”部分。当使用 GET 方法时，QUERY_STRING 是一个环境变量，用来存储客户端浏览器的数据。

19.3.2　提取表单数据

我们已经知道，表单提供的或是直接由 URL 定义的数据要能够有效，要么必须通过标准输入，要么必须通过生成 QUERY_STRING 环境变量。访问一两段信息不难，但是对信息解码并不容易。

来自表单的信息被作为一系列的键/值对提供，当提交表单时，键是表单字段的名称，值是字段的值。这些键/值对用等号分隔开，多个键/值对用符号“&”分隔开。

更糟的是，由于显然不能在字符串里包括任何等号和“&”符号，就需要经常编码键/值对信息，否则 CGI 脚本将会拒绝它。其他的字符也需要被编码——实际上，除了字母(大写和小写)和数字外，所有字符一般都会被翻译成它们对应的十六进制码，再加上“%”作为前缀。例如，字符串“Martin Brown”将会被翻译成“Martin%20Brown”。

听起来是不是很复杂？

获取表单信息的最简单的方法，就是使用标准 cgi 模块。它对提供给 CGI 脚本的表单数据进行分析，并通过一系列函数调用或单个的函数调用使得信息可用，所有的信息都可以直接被存放进一个字典中。

例如，下面的脚本提供了一个询问名称的表单的例子。当键入名称并提交表单后，脚本返回并显示“hello”。

```
from cgi import *

print "Content- type: text/ htmlnn"
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
    print """
    <html>
    <head>< title> Greeting</ title></ head>
    <body bgcolor=" White">
    <h1> Please enter your name</ h1>
    <form method= get action="/ pyform. cgi">
    <b> Name: </ b>< input type= text name= name size= 40></ br>
    <input type= submit>
    <hr>
    """

    form = FieldStorage()

    if (form. getvalue( 'name')):
    print "< h2> Hello", form. getvalue( 'name'),"</ h2>"

print "</ html>"
```

处理过程的主要部分是创建“FieldStorage”类的实例。结果对象是包含了字段及其值的结构，有许多不同的方法可以访问它。

如果在“form”对象中使用字典符号，它会返回“MiniFieldStorage”实例，其中包含一个匹配了给定名字的字段的列表。例如，下面这一行：

```
print form[ 'name']
```

产生出

```
MiniFieldStorage( 'name', 'MC')
```

当在一个表单中有许多字段向同一个字段名提供信息时，这种方法就会很有用处。例如，如果有一组复选框列出了可选项，当所有复选框都被选中时，就可以使用一个表单字段来保存所有值。

对于标准表单这更加有用，此时对“form”对象使用的方法是“getvalue()”方法。此方法以更熟悉的格式返回信息，本例中是一个字符串。如果使用复选框，“getvalue()”方法的返回值就会是这些值的列表。

这当然只是冰山一角：如何处理从表单检索到的数据，将完全取决于个人的实际情况。或许需要发送信息到数据库，发送给用户一封电子邮件，或是使用此信息根据用户喜好为其建立定制的新闻页面。

“cgi”模块支持的其他函数列在表 19-2 中。

表 19-2　cgi 模块支持的函数

函　　数	说　　明
parse(fp)	分析查询字符串(从 QUERY_STRING 环境变量中获得，如果为空，从 sys.stdin 中获得)

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

（续表）

函　数	说　明
parse_qs(string[,keep_blank_values, strict_parsing])	分析参数 string 中的查询字符串。结果字段和值作为字典返回给调用者，它带有作为字段名的键和作为字段值列表的值。如果 keep_blank_values 设为 true，它为空字段建立键。默认是忽略空字段。如果设置了 strict_parsing 参数，若给定的字符串存在任何错误，就会引发一个“ValueError”异常。默认操作是忽略任何错误
parse_qsl(string[,keep_blank_values, strict_parsing])	与 parse_qs()一样，但是返回字段名称和值作为字段名/值对的列表
parse_multipart(fp,pdict)	分析来自文件对象“fp”的输入，作为由多个部分组成的表单的数据。pdict 是在 Content-type 报头中由客户端提供的参数的字典。返回类似于 parse_qs()的字典
Parse_header(string)	将 MIME 报头分析为主值和参数的字典
test()	写回最小的 HTTP 报头，并且将环境和表单信息显示回客户端。用于调试
Print_environ()	在 HTML 格式中打印 shell 环境。用于调试
Print_form(form)	使用被支持的 form 对象以 HTML 格式打印表单数据。用于调试
Print_directory()	从以 HTML 格式执行的脚本中打印出当前目录
Print_environ_usage()	在 HTML 中打印有用的环境变量列表。注意，它显示的只是环境变量的名称，而不是数据
escape(s[,quote])	将字符串中特殊的 HTML 字符“&”、“<”、“>”转换为 HTML 的安全序列。如果 quote 为 true，也转换双引号。要了解关于翻译非安全字符的工具集的详细信息，请参见后面的“转义特殊字符”一节

19.3.3　发送信息

当需要将信息发送回客户端时，只用简单地将输出发送到脚本的标准输出中；Web 服务器将会提取这些信息，并通过提交请求的客户端使用的网络套接字，将信息发送回客户端。

应答分为两部分：HTTP 报头和需要发送回的文档。HTTP 报头用于告知客户端返回数据的信息。最少应当包括“Content-type”头部，它告知客户端返回信息的格式。对于 HTML，正如已经看到的，格式为“text/html”。例如：

```
print "Content- type: text/ html\n\n"
```

发送回的文档的报头和主体之间用一个换行符产生的空行隔开。有关 HTTP 报头及其使用方法的信息，下面将进行更多的介绍。

文档的主体可以包含任何需要的内容，不必限于 HTML，可以返回图像、声音文件甚至压缩的文档。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

1. HTTP 报头

返回的 HTTP 报头信息如下：

Field: data

Field 名字需要区分大小写，否则也可以在冒号和字段数据之间使用尽可能多的空格。HTTP 报头字段的示例列表如表 19-3 所示。

表 19-3　HTTP 报头字段

字　段	意　义
Allow: list	由请求的资源(脚本或程序)支持的 HTTP 请求方法的列表，用逗号分隔开。脚本通常支持 GET 和 POST 方法；其他方法包括 HEAD、POST、DELETE、LINK 和 UNLINK
Content-encoding: string	消息体使用的编码。目前只支持 Gzip 格式和压缩格式。如果需要编码数据，必须保证已经核对了环境变量中的 HTTP_ACCEPT_ENCODING 的值
Content-type: string	MIME 字符串，定义了所返回文件的格式。对于 HTML 页面，值为 text/html。更多信息请查看本章后面的内容
Content-length: string	返回的数据的长度，按字节计算。浏览器使用其值来报告文件的大概的下载时间
Date: string	发送消息的日期和时间，格式必须为：01 Jan 1998 12:00:00GMT。为使于参照，时区定为 GMT。如果必要，浏览器能够计算与当地时区的差值
Expires: string	信息失效的日期。浏览器使用其决定页面何时被刷新。应当与报头的 Date 字段的格式相同
Last-modified: string	资源最新修改的日期。浏览器使用其决定远程版本是否比在在缓存中更新了。应当与报头的 Date 字段的格式相同
Location: string	可以使用此字段定义浏览器将要访问的 URL，来代替继续读取被请求的 URL 的信息。用于将用户重定向到另外位置。必须是绝对 URL，相对 URL 无效
MIME-version: string	所支持的 MIME 协议的版本。更多信息请参见 MIME 的资源网页：http://www.oac.uci.edu/indiv/ehood/MIME/MIME.html
Server: string/string	Web 服务器应用程序和版本号
Title: string	资源的标题
URI: string	代替被请求的 URI 的返回的 URI

惟一必需的字段是 Content-type，它定义了返回文件的格式。如果不特别指定，浏览器假定发送回的预格式化了的是原始文本，而不是 HTML。文件格式由 MIME 字符串定义。MIME 是一个由斜线分隔的字符串，定义了原始格式及其中的子格式。其中，MIME 是多用途因特网邮件扩展(Multipurpose Internet Mail Extensions)的首字母缩写。例如，text/html 说明返回的信息是纯文本，使用 HTML 作为文件格式。Mac 用户会对文件所有者和类型的概念感到熟悉，这是

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

MIME 采用的一种基本的模型。MIME 正迅速成为一种通用的描述文件类型的方法。它被电子邮件和因特网文件传输使用，还在操作系统(特别是 BeOS，但是 Linux 和其他系统也即将采用)上被采用，用于描述文件系统中的文件的文件内容。

其他例子包括 application/pdf，它表示文件类型是应用程序特有的二进制，文件格式是 pdf，Adobe Acrobat 文件格式。其他熟悉的还有 image/gif，它表示文件是 GIF 文件；还有 application/zip，它是一个 Zip 文档。

浏览器使用 MIME 信息来决定如何处理文件。大多数浏览器具有处理 image/gif 类型的文件的映射，所以可以在页面内放置图形文件。还具有 application/pdf 的入口，可以调用外部的应用程序来打开收到的文件，或者将文件传递给一个可选的给用户显示文件的插件。

客户端和服务器都将文件扩展名映射到 MIME 类型。例如，.doc 扩展名指向 Microsoft Word 文档，因此也就指向 application/msword 的 MIME 类型。在服务器端，文件的扩展名用于将 MIME 类型发送给客户端。在客户端，使用映射来验证未附加于某种 MIME 类型的文件的类型。

如果要查看更多 MIME 类型的例子，可以查看 Apche Web 服务器提供的 MIME 类型文件段的内容。

application/ mac- binhex40	hqx
application/ mac- compactpro	cpt
application/ macwriteii	
application/ msword	doc
application/ news- message- id	
application/ news- transmission	
application/ octet- stream	bin dms lha lzh exe class
application/ oda	oda
application/ pdf	pdf
application/ postscript	ai eps ps
application/ powerpoint	ppt
application/ remote- printing	
application/ rtf	rtf
application/ slate	
application/ wita	
application/ wordperfect5.1	
application/ x- bcpio	bcpio
application/ x- cdlink	vcd
application/ x- compress	
application/ x- cpio	cpio
application/ x- csh	csh
application/ x- director	dcr dir dxr

表面上看，无足轻重，实际上它很重要。缺少了 Content-type 字段，浏览器将不知如何处理接收到的信息。通常 Web 服务器发送回 MIME 类型给浏览器，它使用一张对照表来映射 MIME 字符串到文件的扩展名。因此，当浏览器请求 myphoto.gif 文件时，服务器发送回值为 image/gif 的 Content-type 的字段。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

如果不能成功返回 HTTP 报头的 Content-type 字段，CGI 脚本产生的输出几乎肯定会当成纯文本，并作为纯文本显示在浏览器中。

2. 文档主体

文档的主体就是需要发送回的全部内容。如果是 HTML，就使用本章开头介绍的 print 语句或 HtmlKit 模块来格式化需要发送回的数据。

作为示例，这里给出一个脚本，它从 IMAP 服务器获取一个电子邮件列表，并在发送回客户端之前将列表格式化为 HTML 格式。

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

import imaplib
import sys, os, re, string

# Set up a function that we can use to call the corresponding
# method on a given imap connection when supplied with the
# name of an IMAP command
def run( cmd, args):
typ, dat = apply( eval( 'imapcon.% s' % cmd), args)
return dat

# Get the mail, displaying the output as HTML

def getmail( title, login, password):
# Login to the remote server, supplying the login and
# password supplied to the function
run( 'login',( login, password))

# Use the   select   method to obtain the number of
# messages in the users mail account. The information is returned
# as a string, so we need to convert it to an integer
nomsgs = run( 'select',())[ 0]
nomsgs = string. atoi( nomsgs)

# Output a header for this email account
print '< p>< font size="+ 2"><  b> '+ title+ '</ b></ font></ p> '

# Providing we've got some messages, download each message
# and display the sender and subject
if nomsgs:
# Output a suitable table header row
print '< table border=" 0" cellpadding=" 0" cellspacing=" 0"> '
print "< tr>< td><  b> Sender</ b></ td>< td><  b> Subject</ b></ td></ tr>"
# Process each message
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
for message in range( nomsgs, 0, - 1):
subject, sender, status = '', '', 'U'
# Send the fetch command to the server to obtain the
# email's flags (read, deleted, etc.) and header from the email
data = run( 'fetch', (message, '( FLAGS RFC822. HEADER) '))[ 0]
meta, header = data
# Determine the email's flags and ignore a message if it's
# marked as deleted
if string. find( meta, 'Seen') > 0:
status = ''
if string. find( meta, 'Deleted') > 0:
continue
# Separate the header, which appears as one large string, into
# individual lines and then extract the subject and sender fields
for line in string. split( header, 'n'):
if not line:
sender = re. sub( r'.* ', '',
re. sub( r' ', '', sender))
# If the message is unread, then mark the subject and sender
# in red
if (string. find( status, 'U') == 0):
subject = '< font color=" red"> '
+ subject + '</ font> '
sender = '< font color=" red"> '
+ sender + '</ font> '
print "< tr>< td>% s</ td>< td>% s</ td></ tr>"
% (sender, subject)
break
# Extract the sender/ subject information by looking for the field
# prefix
if line[: 8] == 'Subject: ':
subject = line[ 9: - 1]
if line[: 5] == 'From: ':
sender = line[ 6: - 1]
print "</ table>"
else:
print "No messages"
# Logout from the server
run( 'logout',())

# Set the server information
server= 'imap'
# Print out a suitable HTTP header and HTML page header
print "Content- type: text/ htmlnn"
print """
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
<head>
<title> Mail</ title>
</ head>
<body bgcolor="# ffffff" fgcolor="# 000000">
"""

# Connect to the server, and then call getmail to get the mail
# from the server
try:
imapcon = imaplib. IMAP4( server)
except:
print "Can't open connection to ", server
sys. exit( 1)
getmail( 'MC', 'mcmcslp', 'PASSWORD')
```

19.3.4　转义特殊字符

由于需要使用文本将数据传送给 CGI 脚本，所以在给定的连接上进行安全的传送就有一些限制。前面已经介绍过，当读取来自客户端的信息时，将对标准字母和数字之外的字符进行转换。如果需要提供使用这种方法进行编码的信息，就必须使用 urllib 模块中的 quote()函数。为了将所引用的字符串转换为原来的格式，使用 unquote()；注意这通常是由 cgi 模块处理的，作为分析过程的一部分，它自动将所引用的字符串解码为原版。在表 19-4 中将会看到这两个函数及其同类函数 quote_plus()和 unquote_plus()的详细信息。

表 19-4　URL 及其转义符之间的相互转换

函　　数	说　　明
quote(string)	用 URL 兼容的格式转义“string”中的字符。注意 Python 的 quote()函数默认时不能将空格转换为字符“+”。如果还需要转换加号，请使用 quote_plus()函数
unquote(string)	将含有 URL 转义符序列的字符串转换为普通的字符串。注意 unquote()函数不能将字符“+”转换为空格；请使用 unquote_plus()函数把空格编码为加号

例如，使用下面的语句来转换字符串“file=/path/to/an/odd file”：

```
import urllib
print urllib. quote_ plus( 'file=/ path/ to/ an/ odd file')
```

如果需要改变在这一过程中被转义的字符列表，可以提供函数的第二个参数，它定义了哪些字符是安全而不需要被转义的：

```
urllib. quote( url, 'a- zA- Z0- 9')
```

为了进行更复杂的转换，必须更加仔细地处理需要(更重要的是不需要)包括的字符。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

19.3.5　调试

调试各种 CGI 脚本时必须多加小心，因为许多地方都可能出错。主要的问题在于 CGI 脚本自身。如果脚本引发未被设置软中断的异常，脚本就会在执行时中止，而只向客户端返回很少的信息。

尽管将标准输出传递给了客户端，但并没有传递标准的错误，有关的错误信息被记录到了 Web 服务器的错误日志中。将标准错误重定向到标准输出文件处理，可以改善这种情况，如下所示：

```
import sys
sys. stderr = sys. stdout
```

除此之外，还需要保证异常尽可能完整，以正确追踪各种错误。

在第 24 章中将会介绍更具体的调试细节，还将介绍一些一般和特殊的 CGI 技术。

19.4　Cookie

cookie 是小的、离散的信息片，用于在 Web 服务器中存储信息。cookie 本身存储在客户端而不是服务器端，因此在同一个会话或是跨越了多个会话， cookie 可以用于存储浏览器的单独的访问之间的稳定信息。cookie 最简单的形式中，也许只用来存储姓名；在更复杂的系统中，它提供网站的登录和密码信息。Web 设计人员能够使用 cookie 为单独的用户提供定制的页面。

在其他系统中，cookie 用于存储从网上商店选取的商品信息。这种 cookie 扮演了“购物筐”的角色，来存储商品和其他选择的物品的信息。

这两种情况下，由于正是服务器使用这些信息来提供定制的 Web 页面，或是更新 Web“购物筐”中选择的商品，所以 cookie 的创建和如何访问 cookie 中存储的信息都是基于服务器的请求。cookie 有大小的限制，根据浏览器的不同而不同。通常，cookie 的大小应当最多不超过 1024 字节，但是有些浏览器会支持大到 16384 字节甚至更多字节数的 cookie。

cookie 被格式化为类似于 CGI 表单——字段的数据流的形式。cookie 由一系列由符号"&"隔开的字段/值对组成，每一个字段/值对又由符号“=”分隔开。在一般的交互过程中，cookie 的内容在服务器和客户机之间进行交换。服务器将更新作为 HTTP 报头的一部分发送回 cookie，浏览器则将 cookie 当前的内容作为请求的一部分发送给服务器。

除了字段/值之外，cookie 还有许多附加的属性。包括过期时间、域、路径以及可选的安全标志。

- 浏览器使用过期时间来决定何时从内部列表中删除 cookie。只要还没有到过期时间，每次访问服务器的页面时，cookie 就会被发送回正确的服务器。
- 域属性中存储有效服务器的定义。它是需要向其发送 cookie 的服务器的部分或完整的域名。例如，如果域属性的值是“.foo.bar”，那么对于每一次访问，cookie 数据将会发送给 foo.bar 域内的所有服务器。
- path 是与 Web 服务器的路径类似的一个部分的匹配。例如，路径为“/cgi-bin”表示只有

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

对于以这一路径开始的请求，cookie 数据才被发送。通常，规定“/”表示将 cookie 发送给所有的 CGI 脚本，但可以限制 cookie 数据只发送给以“/cgi-public”开始的脚本，而不发送给“/cgi-private”下的脚本。

- 安全属性限制浏览器向不安全的链接发送 cookie 数据。如果设置了这一属性，cookie 数据将只在安全的连接上传输，例如由 SSL 提供的连接。

在 Python 中提供一个第三方的模块“Cookie”，它建立适合于被发送回客户机的新的 cookie。例如，可以如下使用“Cookie”模块对前面的代码重写：

```
import Cookie
cookie = Cookie. SmartCookie()
cookie[ 'sample'] = 'login=% s; other= Other' % (login)
cookie[ 'sample'][ 'path'] = '/ '
cookie[ 'sample'][ 'domain'] = 'mcwords. mchome. com'
cookie[ 'sample'][ 'expires'] = 365* 24* 3600
print cookie
```

注意，不是大家所熟悉的在使用 Perl 或 JavaScript 时使用相关字符串，这里实际上是自己预先计算了值。cookie 作为结果，将被作为 HTTP 报头的一部分发送回客户端，应当被放在一个用于区分报头和主体的空行之前。

当浏览器访问与创建 cookie 时所定义的域和路径相匹配的 URL 时，cookie 自动被发送回浏览器。注意，客户端只发送回匹配域和路径的 cookies，它不会发送所有的 cookies。

然后，Web 服务器将 cookie 信息存放到 HTTP_COOKIE 环境变量中。要将客户端提供的 cookie 分析为内部的结构，最简单的方法就是使用 SmartCookie 对象的 load()方法：

```
import Cookie, os

cookie = Cookie. SmartCookie()
cookie. load( os. environ[ 'HTTP_ COOKIE'])
print cookie[ 'sample']
```

要存储与网站相关的客户端的信息，cookie 是一种很好的方法。例如，或许会使用 cookie 来保存一个引用值，此值用来指出用户在网站上的登录。这样，当用户访问您的站点时，您就能自动知道他们是谁，因此也就能向他们提供定制的视图。

但是请小心：不要在 cookie 中存储用户名和密码。尽管 cookie 并不被发送到任何不匹配其域和路径的 CGI 脚本，但用户仍然能够读取存储在计算机里的 cookie 中的数据，从而获得用户名和密码。更好的解决方法是，在数据库中存储一个随机生成的字符串，并将其用作 cookie 值。当用户访问网站时，从数据库中查询出这个随机的字符串，从而鉴别出用户。这使得某些人不可能通过查看用户的计算机中的 cookie 来获得该用户的登录名或密码。

当然，这并不能完全防止他们——他们可以复制 cookie，或是使用这台计算机访问网站。因此，作为双层保护，确定任何主要的操作，如更改用户优先权或提交/访问信用卡的细节，都需要用户输入完整的密码。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

19.5　安全性

因特网站上的攻击次数正在不断增长。无法弄清这是归究于计算机黑客的数目众多，还是仅仅因为太多的公司和主机未能重视。实际上，如果遵循一些简单的准则，要保证脚本的安全其实十分简单。在介绍解决方法之前，先介绍易受攻击的脚本类型：

- 所有将表单输入发送给邮件地址或邮件消息的脚本
- 所有发送将被用在子 shell 窗口或是要调用 system()或 exec*()的信息的脚本
- 所有在表单处理过程中盲目接收无限制数目信息的脚本

前两种危险的类型相对比较明显：如果攻击者提供了正确的信息，实际上就能在可以执行任何操作的命令行中为所欲为。例如，设想一个电子邮件地址直接传送到如下所示的 sendmail 语句：

```
me@ foo. bar;( mail me@ foo. bar </ etc/ passwd)
```

如果这条语句被当作 sendmail 行的一部分，在命令行中执行，分号后面的命令将会用电子邮件把密码文件发送到同一个用户，如果不经检查就会带来严重的灾难。捕获这种情况最简单的方法，就是总是检查提供给任何外部的各类命令的信息。

使用 CGI 脚本时可以遵循的简单规则是不要相信由对方提供的关于数据的大小、内容或是结构的信息。

下面列出了在编写安全的 CGI 脚本时，应当进行检查的注意事项：

- 使用前双击字段名称、值和关联项。例如，确保邮件地址确实是邮件地址，保证邮件地址确实是需要从表单中得到的正确字段的一部分。
- 不经检查不要自动处理字段值。作为一条规则：列出想接受的 ASCII 字符的一个清单，过滤掉不符合正则表达式的所有语句。
- 检查有效的信息，要比过滤掉坏的数据要更容易。使用正则表达式来匹配需要的，而不使用它来匹配不需要的。
- 检查输入的变量的大小，或者最好是表单数据。邮件地址不要超过 256 个字符，对于被配置为只支持 40 个字符的字段，如果有人向字段提供更多的数据，那么任何这样的人都有可能不怀好意。可以使用“CONTENT_ LENGTH”环境变量，由 Web 服务器计算出它的值，检查通过“POST”方法接受的数据的长度，有些 Web 服务器还提供使用“GET”请求获得的信息。
- 在使用之前不要假定字段数据存在或是有效的；空字段和填满坏的数据的字段一样会造成许多问题。
- 除非能够肯定是什么内容，不要总返回文件的内容。当以为用户在请求 HTML 打开文件时，随意地返回一个密码文件，会带来严重的权限滥用问题。要确保文件请求指向的文件是乐意返回的，或者可以使用 os. chroot( )方法来限制用户访问的对象。
- 不要接受发送给脚本的路径信息是自动有效。选择备用的能够信任的 PATH 环境值，将其硬编写为脚本的初始化参数。当在此路径上时，使用 del 方法删除任何不需要使用的环境变量。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

● 如果准备接受路径或文件名，要确保它们是相对路径，而不是绝对路径，而且不包括指向父级目录的“..”。入侵者能够很轻易地指定文件“../../../../../../../../../ etc/ passwd”，甚至在很深的目录路径下，这都会引用密码文件。

● 总是使用 os. system()方法、os. fork()方法或是 os. exec*()方法验证使用的信息。如果没有其他问题，要确保传递给这些函数的所有变量不包含这些字符：“;”、“|”、“(”或“)”。此外，要避免一起使用这些字符。

● 永远不要使用带有 os. popen()、os. exec*()以及其他有可能运行外部命令的函数的值，它们是不被处理的。

● 要确保 Web 服务器不是以“root”用户运行的，否则会使计算机很容易招致各种类型的攻击。要以“nobody”用户运行 Web 服务器，或是专门为 Web 服务器创建一个新用户，保证脚本只能被 Web 服务器用户读取和执行，任何人都不可写。

● 不要假定带有“hidden”类型的 HTML 字段真的就是被隐藏的。通过查看源文件，用户仍然能够看到它们。也不要依赖于自己的加密算法来加密在这些隐藏字段中提供的信息。要使用已有的、经过检验的、没有错误的系统。对于大多数加密系统，都会有相应的解决办法，可以查看 Vaults of Parnassus 来获取合适的模块。

如果遵循这些规则，就会在最大程度上减少被攻击的危险。但是，没有办法彻底保证安全性。一个顽强的攻击者会使用许多不同的工具和技巧来达到目的。

哦，再次提醒一句：不要相信对方所提供的关于数据的大小、内容和结构的信息！

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第20章　标准标记语言处理

有许多不同的标记语言可以用于建立文档：一些如 SGML 和 HTML 出现已经有一段时间了；其他系统，如 XML，也基于同样的原理。XML 相对比较新一些。从技术的角度讲，XML 和 HTML 都是 SGML 应用程序的范例。

标记语言提供结构化和布局文档的方法，它允许识别文档的独立的一部分，例如头部和尾部、章的标记和其他元素，以及标记单个单词或段落的能力。例如，一个十分简单的 HTML 文档可能会像下面这样，它包括文档标题、一些段落文本(其中会包含加粗的元素)以及项的列表：

```
<html>
<head>< title> Some Old Document</ title></ head>
<body bgcolor= '# ffffff' fgcolor= '# 000000'>
<p> Here's a list of the items that I'd like today from
<b> Sainbury's</ b>. Please remember your <i> credit cards</ i></ p>:
<ul>
<li> Sugar
<li> Redbush Tea
<li> Red Mountain Coffee
<li> Milk
</ ul>
</ body>
</ html>
```

文档中单个的组件和元素用标记符来标记，标记符是用尖括号括起来的元素。在 HTML 中它们有特殊的含义，例如标记符“”表示此标记符后的文本被标记为粗体显示，对应的结束标记是“”，它表示文本的粗体显示到此为止。其他一些标记只需要一个标记符，例如，标记符“<li>”表示一个列表项，但是不需要指定结束标记符(除非在 XHTML 中)。

SGML、XML 和 HTML 以同样的方法实现，它们使用相同形式的标记符。其他标记语言，例如 TeX 或*roff 一族的语言，使用类似的基于标记符的标记语言，但是它们不遵循相同的格式。由于文档是文本，所以很容易创建文档，所有需要做的就是使用“print”输出必要的信息。在第 19 章中已经介绍了一些例子。

然而，有时需要读取存放在这些文档中的信息，或是使用从其他地方获得的文档，从中提取不同的元素。例如，在第 19 章中介绍了“urllib”模块，它可以下载任何因特网对象。如果能够理解 HTML 页面中的信息，就能提取链接的列表，甚至是图像的列表，这样就能够下载用于在本地查看文档所需要的一切内容。

使用正则表达式能够提取出信息，但这会很杂乱。更好的解决方法是实际分析和理解文档，这样就能根据名称提取出需要的元素。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

标准 Python 库包括读取和分析 SGML、HTML 和 XML 文件的工具。“sgmllib”模块不具有完整的 SGML 语言那样的可扩展性。“sgmllib”能够分析基本的标记符和文档结构，但是不支持 SGML 语言中许多更高级的特征。“htmllib”模块提供更加通用目的的接口，用于从 HTML 文档中分析和提取信息。由于 HTML 只是 SGML 语言格式的一个应用程序，所以“htmllib”实际上继承了“sgmllib”模块的大部分功能。

Python 中对于 XML 的支持则更加广泛。“xmllib”是一个老的分析接口。我们可以访问它，现在它被 Expat 接口或 SAX(Simple API for XML 及 DOM(Document Object Model，文档对象模型)接口所取代。

本章中将对这些模块进行介绍，还将介绍使用这些标准模块能够创建的一些工具。

20.1　处理 SGML

所有这些标记语言的工作方式都是相同的。它们都起源于 SGML。SGML 最初于 20 世纪 70 年代由 IBM 发明。SGML 被设计用于允许(用标记)区分出文档中的技术信息，例如在 IBM、波音(Boeing)和 NASA 中使用的技术信息。这样就可能只提取出需要的信息，就能从完整的技术参考手册的快速参考指导中建立一切。

使用SGML可以用相同的标记符标记元素，还可以指向同一文档甚至不同文档的其他部分。由于 SGML 是原始格式的，所以可以输出任何需要的格式，从 HTML 文档到 Microsoft Word、帮助系统，甚至是 PostScript 或 PDF 格式的文件。

Python 的 SGML 实现的确有限。“sgmllib”模块提供的 SGML 分析器只能对 SGML 标记符进行分析和理解，也就是说，只能鉴别“<xxx>”标记符以及在起始和终止符之间包含的数据。因为“htmllib”模块建立在“sgmllib”模块的基础上，也因为大多数人都熟悉 HTML，所以下面就直接介绍 HTMLParser 类。

20.2　处理 HTML

SGML 最大的成功就在于 HTML，即超文本标记语言(Hypertext Markup Language)。HTML 是 Tim Berners-Lee 在 1990 年开发的，它实际上是 SGML 的一种应用的特例，尤其适合标记用于在屏幕上浏览的文档，它给 World Wide Web 带来了巨大的影响。HTML 不提供 SGML 的完整的能力：它只有一个特定的有特殊含义的标记符的集合，文档本身只用于在屏幕上浏览。为了简单起见，HTML 定义了文档的外观，而没有描述文档的内容。另一方面，SGML(以及 XML)使用标记符定义内容——分析器才能将其转换为屏幕显示的、打印的或其他格式的文档。尽管如此，这些缺点并没有影响 HTML 的使用。

处理 HTML 与处理 SGML 的基本步骤完全相同。然而由于许多 SGML 支持的更复杂的标记符样式，HTML 都没有采用，所以分析 HTML 要更加简单些。

HTML 处理方式

“htmllib”模块提供一个新的类“HTMLParser”，它本身基于“sgmllib”模块提供的

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

SGMLParser 类。这两种类的工作方式或多或少都是相同的：使用“feed()”方法向分析器中输入信息。可以对它进行多次调用，以支持新的需要分析器处理的信息。

在处理文档时，分析器每遇上一个标记符，都会尝试调用相应的方法。例如，当遇上“<img...>”标记符，分析器调用类中已定义的“do_img()”方法。此方法将一个在标记符中定义的属性的列表作为元组对的列表。例如，下面的标记符：

```
<img src= 'myimage. gif' width= 100 height= 200 alt= 'logo'
align= 'left'>
```

提供的属性列表如下所示：

```
[( 'src', 'myimage. gif'), ('width', '100'), ('height', '200'),
('alt', 'logo'), ('align', 'left')]
```

使用元组的列表可能不是最好的解决方法，字典或许更有用，但是它用于信息的处理已经足够了。通过使用这种方法，可以从标记符的属性和附加信息中提取任何有用的信息。对于需要识别的每个标记符采用不同的方法，分析器能够智能地识别匹配对中有效或无效的标记符。例如，使用“do_br()”方法处理“
”标记符，而“start_a()” 方法和“end_a()”方法则用来处理“<a...>...</ a>”的锚标记符。

查看“htmllib”模块的源代码可以看出处理工程是如何进行的。下面的一小段代码，展示了刚刚分析的用于锚和图像标记符处理的方法。

```
def start _ a( self, attrs):
        href = ''
        name = ''
        type = ''
for attrname, value in attrs:
      value = value. strip()
      if attrname == 'href':
         href = value
      if attrname == 'name':
         name = value
      if attrname == 'type':
         type = value. lower()
self. anchor_ bgn( href, name, type)


def end_ a( self):
self. anchor_ end()


# --- Image

def do_ img( self, attrs):
      align = ''
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
        alt = '( image) '
        ismap = ''
        src = ''
        width = 0
        height = 0
        for attrname, value in attrs:
        if attrname == 'align':
          align = value
        if attrname == 'alt':
          alt = value
        if attrname == 'ismap':
          ismap = value
        if attrname == 'src':
          src = value
        if attrname == 'width':
          try: width = int( value)
        except: pass
          if attrname == 'height':
        try: height = int( value)
    except: pass
    self. handle_ image( src, alt, ismap, align, width, height)
```

从这段程序中可以看到，在传递“anchor_end()”方法终止之前，“start_a()”方法从锚标记符中提取锚的信息。“end_a()”方法停止对锚标记符的处理。“do_img()” 在调用“handle_image()”方法终止之前，提取图像标记符中包含的一些典型的属性。

使用 htmllib

使用 HTML 分析器的主要的方式是创建一个新的类，这个类继承“htmllib”模块提供的“HTMLParser”父类的方法，然后重载需要自行处理的标记符的方法。例如，如果需要处理“<img...>”标记，就要定义自己的“do_img()”方法。

基本格式的 HTMLParser 实际上进行的处理很少。它被设计用来与“formatter”模块中定义的类一同工作，将 HTML 文件中的信息输出到屏幕。“formatter”模块允许根据触发的事件按结构化方式输出信息。需要更多信息，可以参看相关 Python 文档。

除此以外，HTMLParser 类提供的惟一有用的功能是建立在文档中查找到的锚标记符的列表。当文档被分析时，这些信息与 HTMLParser 类的实例的“anchorlist”属性中的元素一同被建立。例如，下面的脚本分析一个文档，然后显示找到的锚的列表：

```
import htmllib, sys, formatter
try:
file = sys. argv[ 1]
except:
print 'You must supply a file name'
sys. exit( 1)
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
try:
htmlfile = open( file, 'r')
except IOError, msg:
print "Error:", file, ":", msg
sys. exit( 1)

htmldata = htmlfile. read()
htmlfile. close()

parser = htmllib. HTMLParser( formatter. NullFormatter())
parser. feed( htmldata)
print parser. anchorlist
parser. close()
```

当在作者的网站的主页上使用时，脚本生成下面这样的列表：

```
['/ ', '/ legal/ ', '/ help/ ', '/ archive/ ', '/ about/ ', '/ projects/ articles',
'/ search/ ', '/ contact/ ', '/ projects/ books/ ', '/ downloads/ ',          '/ info/',
'/ projects/ ', '/ resources/ ', '/ elsewhere/ ',
'/ projects/ books/ pcr2e/ index. shtml', '/ downloads/ pcr2e/ pcr2esrc. zip',
'/ downloads/ pcr2esrc/ pcr2esrc. tgz', '/ projects/ books/ pdbg/ index. shtml',
'/ downloads/ pdb/ pdbsrc. zip', '/ downloads/ pdb/ pdbsrc. tgz',
'/ projects/ books/ pidk/ index. shtml', '/ projects/ books/ pidk/ index. shtml',
'/ projects/ books/ pdbg/ index. shtml', '/ projects/ books/ pdbg/ index. shtml',
'/ projects/ books/ imac/ index. shtml',
...
'/ downloads/ paa/ index. shtml', '/ downloads/ paa/ paa. zip',
'/ downloads/ paa/ paa. tgz', '/ downloads/ paa/ paa. hqx',
'http:// www. amazon. com/ exec/ obidos/ redirect- home/ mcwords? tag- id= mcwords&pl
acement= holiday- home- btn- 120x90. gif& site= amazon',
'http:// www. dreamhost. com/ rewards. cgi', 'mailto: mc@ mcwords. com']
```

可以使用这些信息建立网站上的链接的列表，甚至只用提取出链接并将其添加到需要下载的页面的列表中，就可以下载整个站点到本地磁盘。

除了使用“htmllib”返回锚标记符列表外，要实现更复杂的功能，需要创建新的类来从 HTMLParser 继承，同时还要重载需要自己处理的方法。

继续前面的例子，如果需要下载一个网站，还需要保证下载了页面上的图像。建立自己新的类，并重载 “do_img()”方法来建立需要下载的图像的 URL 列表，就可以很简单地达到目的。然而还有更简单的办法：从前面的程序段中，可以知道“do_img()”方法用属性列表调用了“handle_image()”方法，重载“handle_image()”方法会更直接一些。

新的类如下所示：

```
class ImgParser( htmllib. HTMLParser):
    def __ init__ ( self, formatter):
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
        htmllib. HTMLParser. __init__ ( self, formatter)
        self. imglist = []


    def handle_ image( self, src, alt, ismap, align, width, height):
        self. imglist. append(( src, alt, ismap, align, width, height))
```

由于 Python 没有调用 HTMLParser 类的初始化程序，所以“__init__ ()”方法必须手动地进行调用。“handle_image()”方法只是添加一个元组到包含所需信息的“ImgParser”实例的“imglist”属性中。为了重新取回信息，通过“feed()”方法向分析器提供完整的 HTML 文件，然后访问“imglist”属性：

```
parser = ImgParser(formatter. NullFormatter())
parser. feed(htmldata)
for img in parser. imglist:
    print img[0]
parser. close()
```

当然，还有更多的例子。假如需要编写合适的处理信息的方法，可以根据接收的 HTML 数据进行更多的处理。

20.3　处理 XML

XML 本质上是 SGML 的精简版本，它具有所有 SGML 的灵活性，但去除了其中许多复杂和无用(或是太复杂而无法实现)的特性。XML1.0 标准正式形成于 1998 年 2 月。XML 允许使用自己的标记符来标记文档，在不依赖于专有的二进制格式的同时，提供将文档布局为结构化格式的能力。

使用自己标记集的另一种方法是创建 XML 文档与 DTD(Document Type Definition，文档类型定义)的组合，DTD 定义了 XML 文档的标记符和结构，是控制 XML 信息的格式和布局的方法之一。因为 DTD 定义了 XML 文档中的信息如何写入和提取，所以尤其适合在两个不同的应用程序之间交换信息。

尽管这听起来可能与 HTML 没有什么区别，但是实际上这使得 XML 十分擅长于布局结构化格式的数据，这样就能分析和转换格式，从而在两个不同的应用程序之间共享信息。

例如，使用 XML 能够建立文档，为银行存放客户的信息。下面的例子显示了此文档如何被结构化成单一的带有子账户的客户，以及如何将子账户的信息存放进账户名称、提供者、余额以及交易的列表。

```
<client>
<clientname> Martin Brown</ clientname>
<account>
<accname> Checking</ accname>
<provider> HSBC</ provider>
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
<balance>$ 4567.00</ balance>
<transaction>
<payee> Rent</ payee>
<amount>$ 280.00</ amount>
<freq> monthly</ freq>
</ transaction>
<transaction>
<payee> Time Subscription</ payee>
<amount>$ 26.00</ amount>
<freq> yearly</ freq>
</ transaction>
</ account>


<account>
<accname> VISA</ accname>
<provider> Morgan Dean Stanley Witter</ provider>
<balance>$- 3485.00</ balance>
<transaction>
<payee> Supermarket</ payee>
<amount>$- 450.00</amount>
</ transaction>
<transaction>
<payee> Gas Station</ payee>
<amount>$- 18.00</amount>
</ transaction>
</ account>
</ client>
```

因为信息被作为简单的文本文件存储，所以不用担心文件的兼容性，就能简单地与其他应用程序交换信息。——文件只不过是文本形式的，所有的应用程序都会知道如何鉴别文件中需要的元素。可以非常容易地使用这种 XML 格式，作为在银行和财务管理应用程序(Quicken、Microsoft Money 以及 GNUcash)之间交换账户信息的方法。

XML 的确很灵活，可以用于重建所有类型的文档：

- Microsoft Word 实际上能够以 XML 格式存储文档。尽管标准文件格式不是基于文本的，并且还依赖于许多 Microsoft 专有的元素，但是只要愿意，是可以将任何 Word 文档保存为 XML 格式的。
- 矢量图形也能用 XML 重建，通过将不同的线和其他点的位置转换为一系列 XML 标记。
- Spreadsheets(电子数据表，一种电子制表软件)能够以 XML 格式存储信息，只要将行和列的内容转换为 XML 表达式，甚至能够嵌入进输出格式、计算式和其他数据中。

合同软件可以以 XML 格式存储合同数据。这方面的工作正在不断取得进展，以使我们能够使用 XML，这样就不再局限于一份合同中能够存放的合同数、地址或其他信息的数目了。更进一步，可以在桌面和手持设备之间使用 XML 文档交换信息。桌面计算机和手持设备将决定

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

哪些字段被存储和显示。

实际上，我们能在不同程度上使用 XML 来标记需要的任何信息，上面所有的例子都是可行的，许多已经在使用了。惟一不能轻易存储为 XML 格式的数据，就是原始的二进制数据，包括图片、声音、视频和基于位的长的数据流，例如应用程序和扩展库需要的机器代码。

20.3.1　XML 分析器

Python 支持四种不同的 XML 分析器："xmllib"、与基于 Expat C 的 XML 分析器的接口、SAX(Simple API for XML 和 DOM(Document Object Model)，所有这 4 种都能从 XML 文档中提取全部的信息。分析器通常符合两种流行的类型之一。要么是基于事件的，即当遇上特殊的标记符，就会触发对某一函数或方法的调用；要么是基于树形结构的，即访问 XML 文档中的信息时，就像整个文档是一棵树，逐个枝干地进行访问，从而从文档中访问到单个元素。

由于从来不会将整个文档存放在内存中，所以基于事件的模型适用于处理大量的文档。当需要将文档从最初的 XML 转化为其他形式时，例如转化为 HTML/XHTML，甚至要输入到数据库中，这种方法就很有用处。按照这种基于事件的模型的工作原理，必须按次序读取文档，从开头到结尾顺序地分析全部内容。

当按随机方式从 XML 文档中访问元素时，就应当使用基于树形结构的访问方法。基于树形结构的分析器检查整个 XML 文档，建立对象的层次结构，从而可以根据名称访问单独的标记符(以及它们相关联的数据元素)。

"xmllib"、Expat 和 SAX 分析器是基于事件的，也按照同样的基本原理工作。可以建立从相应模块中定义的基类继承方法的新类。然后可以注册(分配)或重载实际处理文档中标记符的方法。事件系统十分灵活，可以为不同的标记符注册已知和未知的处理程序。

"xml.dom"模块提供与 DOM 分析器的接口。主要的接口称为 minidom(由 xml. dom. minidom 模块提供)。使用 minidom 可以导入整个 XML 文档，然后通过调用 DOM 对象的方法访问标记符，返回当前分支上的标记符的列表。例如，从前面的银行客户的 XML 文档例子中，取得 VISA 账户中的事务标记符对象的列表。每一个事务的标记符还有子对象，具体描述收款人和金额。

关于 XML，最后还要注意的一点就是，无论是文档和标记符的编写方式，还是标记符中包含的数据的形式，XML 都采用的是 Unicode。遇上 Python 中的 Unicode 时，如果还不熟悉 Python 的工作原理，请参见第 12 章。

在 Python1.5 中引入的"xmllib"模块，尽管仍是标准发布的一部分，但已不再被直接支持。由于 SAX 系统(通过 xml.sax 包支持)可以扩展，其他语言的 SAX 的用户将不再会对它的支持感到失望了。不过，还是应当优先选用 Expat 基于事件的分析器，由于速度快，它很适合于生产环境；还有 minidom 的 DOM 分析器，它十分简单。

20.3.2　使用 Expat

Expat 是一种非验证性的 XML 分析器，是 James Clark 用 C 语言编写的。由于 Expat 在分析 XML 文档时不检查文档的格式，所以被归类为非验证性的 XML 分析器。Expat 是事件驱动的，与前面介绍的 htmllib 例子的工作方式是很相似的：它分析单独的 XML 构造，并使用回调

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

来初始化处理过程，处理单独的开始和结束标记符和数据内容。

为了使用 Python 中的 Expat，需要导入 xml. parsers. expat 模块。这一模块支持“ParserCreate()”主函数，它创建 Expat 分析器的一个实例，用于分析 XML 文档。

对于使用来说，可能最简单的就是建立新类，并在新类中放上所有需要使用的方法，包括那些由不同的 XML 构造触发的方法。这不是必要的，但的确保证了系统的良好和整齐。然而，不像“xmllib”，Expat 不能从父类继承方法，而只能直接使用这些方法；也不像“htmllib”，在新类中，可以将函数注册到基本的分析器而不是重载已有的方法。

```
import xml. parsers. expat
import sys

# Create a new class to hold all the methods that
# we want to use when parsing an XML document
class MyParser:
# Instance constructor. We create a new parser instance
# which we hold locally in parser, then we register
# the different methods which will handle the
# XML elements
def __ init__ ( self, filename):
self. parser = xml. parsers. expat. ParserCreate()
self. parser. StartElementHandler = self. starttag_ handler
self. parser. EndElementHandler = self. endtag_ handler
self. parser. CharacterDataHandler = self. data_ handler
if filename:
self. loadfile( filename)

# Kills off and deletes the parser instance once the
# processing of a given XML file is complete
# To ensure we get rid of circular references we must
# delete the parser reference
def close( self):
if self. parser:
self. parser. Parse( '', 1)
del self. parser


# Hand off some data to the parser
def feed( self, data):
self. parser. Parse( data, 0)

# Called when a start tag is found
def starttag_ handler( self, tag, attrs):
print 'Start: ', repr( tag), attrs
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
# Called when an end tag is found
def endtag_ handler( self, tag):
print 'End:          ', repr( tag)


# Called when a data portion is found
def data_ handler( self, data):
print 'Data:       ', repr( data)


# Load a file and supply the info to the parser
def loadfile( self, filename):
xmlfile = open( filename)
while 1:
data = xmlfile. read( 1024)
if not data:
break
self. feed( data)
self. close()


try:
filename = sys. argv[ 1]
except IndexError:
print "You must supply a filename"
sys. exit( 1)


try:
parser = MyParser( sys. argv[ 1])
except xml. parsers. expat. ExpatError:
print "Error in XML"
```

If we use this on a simple document we get a list of the start and end tags in the document, including any attributes:

```
$ python exexpat. py simple. xml
Start:      u'simple'
Data:       u'n'
Start:      u'paragraph'
Data:       u'and some data'
End:        u'paragraph'
Data:       u'n'
End:        u'simple'
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

注意：

任何数据或标记符都以 Unicode 字符串的形式返回，而不是 ASCII 字符串。可以使用“str()”函数将它们转换回 ASCII 字符串，或者可以使用“unicode()”内置函数将它们转换成其他的编码，或者使用“encode()”方法来返回 Unicode 对象。更多信息请参见第 12 章。

20.3.3　使用 DOM(minidom)

文档对象模型(Document Object Model)分析 XML 文档，然后将分析后的文档表示为树形结构。由于是将 XML 文档表示为一个整体，所以可以使用 DOM 来分析已有的文档或是创建新的文档。

在 Python 中，DOM 接口基于 W3C 发布的 IDL 版本规范。标准 Python2.x 发布符合基本的 DOM 分析系统，称为 minidom，复杂一些的称为 pulldom 系统，它不用将整个的 XML 文档读取进内存，就能从 DOM 树中提取单独的元素。

由于 Python 有灵活的对象系统，所以在 Python 对象中建立起 XML 镜像对等的树形结构就非常容易。有了易用的对象处理(尤其是列表处理)特性，就有了处理 XML 文档的很好的平台。

1. minidom 的工作方式

为了使用 minidom 将已有的 XML 文档分析为 DOM 对象，需要调用 parse()方法，它接受一个文件名或文件对象，然后处理其中的内容；或者是调用 parseString()，它分析纯字符串形式的信息，这些信息可能是从文件或是网络连接中单独读取的。实际上，这种方法十分简单，如下所示：

```
from xml. dom. minidom import parse, parseString

stringdoc = parseString('< para> Some text</ para> ')

xmlfile1 = open('myfile. xml')
filedoc = parse(xmlfile)

xmlfile2 = parse('myfile. xml')
```

一旦将 XML 流转换成为 DOM 对象，就能通过名称访问单个的标记符。再来看看银行客户的 XML 文档：

```
<client>
<clientname>Martin Brown</clientname>
<account>
    <accname>Checking</accname>
    <provider>HSBC</ provider>
    <balance>$4567.00</balance>
    <transaction>
        <payee> Rent</payee>
        <amount>$280.00</amount>
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
</transaction>
<transaction>
        <payee> Time Subscription</payee>
        <amount>$26.00</amount>
    </transaction>
</account>


<account>
    <accname> VISA</accname>
    <provider> Morgan Dean Stanley Witter</provider>
    <balance>$- 3485.00</balance>
    <transaction>
      <payee> Supermarket</payee>
      <amount>$- 450.00</amount>
    </transaction>
    <transaction>
      <payee> Gas Station</payee>
      <amount>$- 18.00</amount>
    </transaction>
</account>
</client>
```

此文档能够被表示为如图 20-1 所示的树形结构。我们将使用此图来帮助理解 Python 的 DOM 实现是如何工作的。

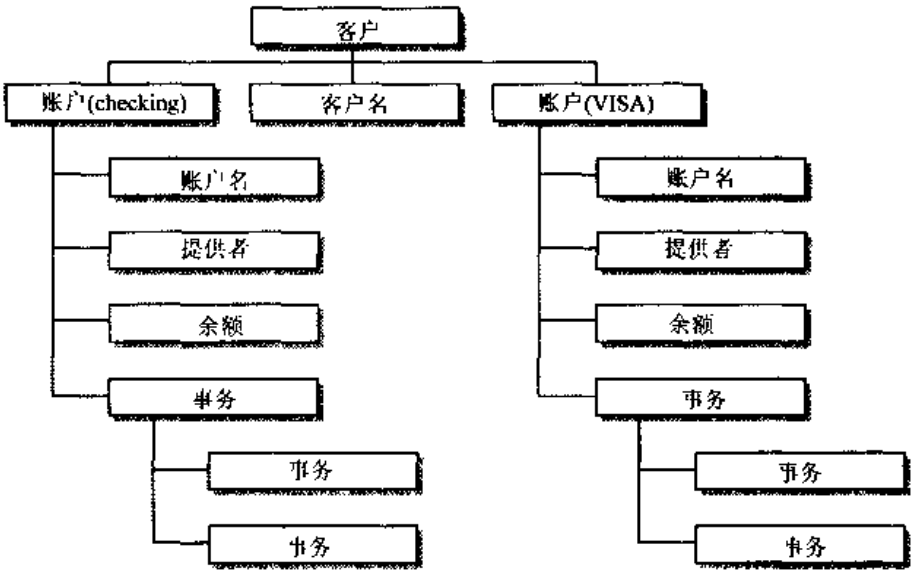

图 20-1　XML 树

使用下面的脚本，能够得到拥有账户信息的客户的姓名：

```
from xml. dom. minidom import parse
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
# Create a function to get the data between XML tags
# Information is held in nodes (discrete blocks)
# which we'll need to concenate together to get the
# full picture. We only need to add text nodes to the
# string
def getdata(nodes):
rc = ''
for node in nodes:
if node.nodeType == node.TEXT_ NODE:
rc = rc + node. data
return rc


# Parse the document
client = parse('client.xml')


# Get the first clientname tag from the document
clientname – client. getElementsByTagName("clientname")[0]


# Print out the data contained within the tags
# using getdata to extract the text from the nodes
# defined within the element
print 'Client name is', getdata(clientname. childNodes)
```

getElementsByTagName()方法返回所有的标记符元素及提供的名称的列表。结果对象包含标记符的信息，包括所有已提供的属性，以及标记在标记符中包含的数据的节点集。

注意对象返回的是图 20-1 所示的树形结构的枝(或叶子)。树的根等同于文档的根，因此为了访问文档中所有的元素，需要从在文档被分析时建立起的 client 对象中引用单个的枝。本例中是名为“clientname”的叶子，它是单独存在的。

如果这样使用：

```
accounts = client.getElementsByTagName("account")
```

账户对象现在就成为包含了两个账户分支的列表。图中的每一个元素都将引用账户分支中的一个。为了获得 checking 账户的事务列表，可以使用下面的语句：

```
checking = accounts[0]
trans = client.getElementsByTagName("transaction")
```

现在“trans”就包含了账户中的两个事务的信息。每一个元素都是事务分支之一。

2. 激活的 DOM

为了方便实际练习，可以建立一个脚本，它以结构化的格式输出 XML 文档中包含的信息。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

此脚本实际上是 Python 中基于树形结构的 XML 分析器的一个很好的例子。由于不用顺序地遍历树(就像基于事件的分析器一样)，所以可以在提取信息时少受一些限制：不用考虑记录的状态，也不用考虑遇上特殊的结束标记符输出格式是否应当改变。

```
from xml. dom. minidom import parse

def getdata(nodes):
    rc = ''
    for node in nodes:
        if node. nodeType == node. TEXT_ NODE:
                rc = rc + node. data
        return rc
def handleclient(client):
      clientname = client. getElementsByTagName("clientname")[ 0]
      print 'Client: ', getdata(clientname. childNodes)
      accounts = client. getElementsByTagName("account")
      handleaccounts(accounts)

def handleaccounts(accounts):
      print 'Accounts: '
      for account in accounts:
            handleaccount(account)

def handleaccount(account):
    accname = account. getElementsByTagName("accname")[ 0]
    provider = account. getElementsByTagName("provider")[ 0]
    print ' ' * 4, '% s (% s) ' % (getdata( accname. childNodes)

getdata( provider. childNodes))
      print ' ' * 4, 'Transactions: '
      trans = account. getElementsByTagName("transaction")
      for transaction in trans:
         handletransaction(transaction)
      balance = account. getElementsByTagName("balance")[ 0]
      print ' ' * 9, '%- 40s %s' % ('', '=======')
      print ' ' * 9, '%- 40s %s' % ('', getdata( balance. childNodes))
      print ''


def handletransaction(transaction):
    payee = transaction. getElementsByTagName("payee")[0]
    amount = transaction. getElementsByTagName("amount")[0]
    print ' ' * 9, '%- 40s %s' % (getdata(payee. childNodes),
getdata(amount. childNodes))
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
client = parse('client. xml')


handleclient(client)
```

如果在客户端的 XML 文档上运行这个脚本，就会得到下面的输出：

```
$ python exdom2. py client. xml
Client: Martin Brown
Accounts:
    Checking (HSBC)
    Transactions:
        Rent                                    $280.00
        Time Subscription                       $26.00
                                                ======
                                                $4567.00

VISA (Morgan Dean Stanley Witter)
Transactions:
    Supermarket                                 $- 450.00
    Gas Station                                 $- 18.00
                                                ======
                                                $- 3485.00
```

这里只是简单地将此文档转换为 HTML 或 XHTML，也可以提取出要写入数据库中单个表的信息。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第21章　Python的其他Web工具

如果询问大多数程序员：最流行的 Web 编程语言是哪一种，将会有很大一部分人选择 Perl。其他人可能会选择 PHP、Visual Basic 或 C++，很少一部分人可能会选择 Python。这很不幸，因为正如在前面的章里所介绍的，在大多数情况下，Python 和 perl 有着同样的功能，甚至在其他很多方面更加先进。

同样的问题，许多人不知道有 Jython(以前是 JPython)。Jython 是 Python 语言的 Java 的完全实现，而不是原先多数人所熟悉的 C 语言的形式。Jython 允许将 Python 和 Java 的代码混合在同一个脚本中使用。它实际上是将 Python 的源代码转换成 Java 的字节码，然后将其传递给 Java 虚拟机。

从服务器端，可选的使用 Python 的最好工具是 Zope(Z-Objects Publishing Environment，Z-对象发布环境)。Zope 允许展现出 Python 对象及其方法，这些方法就像是作为通常 Web 服务的一部分的 CGI 脚本编写的。使用简单的 URL 路径就可以访问对象和方法，所有附加上的键/值对被作为参数展示给方法。

本章将介绍用 Python 编写的或使用 Python 的一些 Web 和因特网工具。不幸的是，无法详细介绍所有这些不同的系统。如果需要更多的信息，可以使用本章中的链接和 Web 地址，还有附录 B 中列出的常用 Python 站点。

21.1　Zope(Z-对象发布环境)

大多数 Web 开发人员所要面对的主要问题之一，就是如何将应用程序作为 Web 站点实现。在最简单的层上，是将 HTML 文件和 CGI 脚本组合起来使用，从而对应用程序提供支持。当试图将两个组件紧密结合起来时，这种模型会导致问题。例如：无法使基于 HTML 和基于 CGI 的元素外观相同。CGI 组件需要导入和处理 CGI 和 HTTP 数据，并在向用户提供 HTML 格式化文档之前，根据这些信息做出决定。

Zope 却不同。在由 CGI 文档生成的 Web 页面上，由于大部分信息都来自 Python 脚本中的对象，所以 Zope 允许就在 HTML 页面的内容中嵌入 Python 对象，或是对象中包含的信息。不必再担心静态 HTML 和动态 HTML 组件的结合问题：HTML 文档包含了对于需要显示的对象和信息的引用。

更进一步，为便于多人同时开发同一站点，Zope 还提供了非常简单的方法。它使用特殊的标记语言，称作 DTML(Document Template Markup Language，文档模板标记语言)。DTML 系统允许整合对于 Python 对象的调用，并在 HTML 页面和诸如 SQL 数据库之类的外部数据源之间创建链接。

Zope 系统去除了 CGI 编程所有的复杂性，取而代之的是允许程序员集中精力开发与内部

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

系统的接口，允许 Web 开发人员集中精力开发合适的文档模板，允许内容主管集中精力填写网站内容，任何人都不会与别人的工作职责交叉。为了进一步帮助说明，下面将介绍 Zope 的组织方式以及对象发布的工作原理。

21.1.1　Zope 系统

Zope 由 4 个主要的组件组成，它们共同作用提供了 Zope 系统。它们包括 Zope ORB(Object Request Broker，对象请求代理)、Z 发布器(ZPublisher)、DTML 标记语言(DTML markup language)以及 ZODB(Zope Object Data Base，Zope 对象数据库)：

- Zope ORB 是 Zope 中的对象请求代理，它是 Zope 系统的核心。ORB 负责将客户端的请求转化成信息，并在对象实例中将其转化为对象和方法调用。更多信息请参见“Zope ORB 工作方式”部分的内容。
- ZPublisher 是一个公共的接口。它在 Web 服务器和请求、CGI 数据和 ZORB 之间交互，ZORB 实际上是 ZPublisher 的一个组件，而不是一个单独的实体。ZPublisher 是整个 Zope 系统的前端，可以与许多不同的 Web 服务器解决方案一同工作，包括 CGI、PCGI、FastCGI、Netscape 的 Web 应用程序接口(WAI，Web Application Interface)、COM、Medusa(参见本章后面的说明)，还包括 ZopeHTTPServer。

大多数人不知道 ZPublisher，把它当成是 ZORB，在大部分情况下这两个术语可以互换。

- DTML 提供简单的定义 HTML 模板的方式。在从任何对象(通过 ZORB 的代理)和外部数据源请求信息的期间，模板被分析。这就使单独的 Python 开发人员和 Web 程序员能够开发 Web 系统，任何人都不用担心如何将 Python 对象和 HTML 代码集成的问题。
- Zope 对象数据库(ZODB)使用“pickle”模块(参见第 12 章)来序列化 Python 对象，并存储数据结果流。除了存储对象的基本功能外，ZODB 还包含了对事务、对单个对象(类似于 RDBMS 中的行和表的锁定机制)的并行访问，以及对对象组件延期赋值的支持，允许用户除非必要，不需要花时间从数据库覆盖所有信息，就能访问对象。整个系统通过键来工作，以类似的方式将信息从 Python 的字典中提取出来。

除此之外，Zope 还提供许多辅助系统，用于帮助开发 Zope 的解决方案。例如，Zope 工具箱包括 HTTP 服务器模块，它使得 Zope 不用借助于现有的 Web 服务器，就能够自行代理所有的请求。其他组件包括管理站点的管理架构和内容管理系统，它与 CVS 系统一同工作，记录站点的变化，允许多个用户更新站点内容而不会干扰他人的工作。

21.1.2　Zope ORB 的工作方式

从 CGI 发布对象并非像第 19 章中介绍的，是一个自然的过程。如果要将包含在 Python 脚本中的信息输出给外部，比起这样的问题来，发布对象是更加简单的解决方案。

为了给出例子，设想已有一个 HTML 页面，它允许用户注册到站点上。先不考虑授权问题，只是着眼于 ZPublisher 和 ZORB 系统的机制。HTML 表单如下所示：

```
<form method= GET action=" http:// zope. mydomain. com/ cgi- bin/ authorize/ login">
Name: <input type= text size= 20 name= login>< br>
Password: <input type= password size= 20 name= passwd>< br>
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
<input type= submit><br>
</ form>
```

提交时，URL 请求的结果如下：

```
http:// zope. mydomain. com/ cgi-
bin/ authorize/ generic/ login? login= mc& passwd= password
```

当请求被 ZORB 接收时，此请求被映射到 zope.mydomain.com 服务器的对象/方法调用上，如下所示：

```
authorize. generic. login( login= 'mc', passwd= 'password')
```

从本质上讲，通常被识别为脚本位置(目录 authorize)的部分被识别为模块的名称；对象的名称作为下一个位置组件(generic)；调用的方法就是通常被当作脚本名称接受的部分。任何追加到这个请求的数据都被当作要调用的方法的参数提供。可以使用虚拟 URL 来更清楚地看到这一点：

```
http:// servername/ directory/ module/ object1/ object2/ method? args
```

它被转化为：

```
module. object1. object2. method( args)
```

来自脚本结果的返回值被插入到预先定义的 DTML 文档模板中，或者作为一个原始的页面。ZPublisher 处理所有的复杂过程，例如返回合适的 HTTP 报头，将信息格式化为合理的文档。

当然，这个简单的例子不能对 Zope 做出任何断定。请求中包含的信息可能非常复杂，要调用的对象和方法会完成所有任务：从返回数据库中的静态值到返回信息片。如果需要存储信息，可以使用喜欢的 SQL 数据库或 ZODB。

记住，引用的 Python 对象不必放置在同一台服务器上。可以很容易地创建页面，它向不同的服务器提出不同请求，这些服务器都运行 Zope，都将信息返回到单个页面。在这种情况下，Zope 的工作方式很像分布式发布环境。例如，假如有一组 Zope 服务器，每一个部门可以使用一台，同时还有单独的“重载”Zope 服务器，它显示关于当前所有部门站点状态的概述页面。

21.1.3　Zope 的特性

Zope 是由 Digital Creations 公司开发的，很多人认为它是下一代强有力的应用程序。Zope 经常被拿来与更著名的集成 Web 发布系统作比较，如 ColdFusion 或 IBM 的 WebSphere 系统。Zope 提供自由的可选择项，符合开放源码产品的所有要求，包括优秀的共用支持和极好的灵活性。

Zope 的未来似乎很光明。系统本身越来越强大，已经拥有许多高级配置的用户包括 ActiveState 公司、IDG Brazil 公司以及许多报纸和目录公司。假如这还不足以巩固 Zope 的成功地位，Digital Creations 公司还提供 Guido van Rossum、Python 开发人员以及其他 Python 开发团队的新主页。

如果需要更多信息和下载自己的 Zope，请访问 www.zope.org 站点。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

21.2　Jython

用户开发始终依赖 CGI 进行交互的 Web 站点时，会很快就感到厌倦，尤其是在企业内部互联网的环境下。要适合大多数人的费用要求，有许多解决方法可供选择：对于简单的回滚和表单处理有 Javascript；如果需要做更重要的工作，可以选用 Macromedia 的 Flash 系统来开发游戏和交互式的幻灯片和图像。

开发客户端应用程序，更常用的语言是 Java。尽管当 Sun 公司发布它时，并没有带来巨大的计算革命，但是如今无论是在开发客户端还是在开发使用 JSP 的服务器端时，Java 都已经成为开发应用程序的选择之一。此外 Java 还有自己的扩展，提供数据库连接(JDBC)和更友好的以 JavaBean 形式出现的 Java 对象模型。

Jython(从前的 JPython)是完全用 Java 写成的 Python 实现。除了使用 Java 写成的以外，它正是与大家所熟悉的 Python 解释器在同一层上，Python 解释器是由 C 语言写成的。这就允许在同一个应用程序中，将 Java 与 Python 组件进行混合和匹配。例如，可以在 Python 应用程序中集成 Java applet，使得客户端不安装 Python 解释器，就能执行 Python 代码。

对于一般的 Python 程序员来说，使用 Jython 不是轻而易举的事。为了使用 Jython，并充分利用系统的优势，需要学会使用 Java 编程。这里介绍了 Jython 基本的信息、工作方式以及能力。更详细的信息，请访问 Jython 站点(www.Jython.org)。

21.2.1　Jython 的工作原理

Jython 系统的主要功能是能将 Python 代码段编译为 Java 字节码，它能够被可以访问 Jython 类库的 Java 虚拟机执行。对于 Java 虚拟机(JVM，Java virtual machine)来说，翻译出的 Python/Jython 代码与任何其他 Java 字节码是一样的。

一旦抓住技术工作的核心，Jython 系统所有其他的特性立刻就非常简单了。Jython 代码继承了 Java 提供的所有功能，包括 Java 的无用单元收集和安全系统。此外，Jython 还为 Jython 程序员提供下列其他特性：

- Jython 的 Java 包装器：同样是能将现存脚本翻译成 Java 字节码的系统，Jython 提供了 Java 的“PythonInterpreter”类，通过常用的“exec”语句和“execfile()”函数，可以编译和执行 Python 代码。由于 Jython 将所有的对象都作为 Java 的 PyObject 的类实例显现出来，所以就能够访问 Java 的类实例中创建的对象。
- 将 Java 库显现在所有用 Jython 缩写的 Python 脚本。这就允许在 Jython 脚本中调用所有 Java 类，就像在使用 Python 类、对象和方法，这一集成过程是完全透明的。可以通过一般的“import”语句引入 Java 和 Jython 类，就像是引入 Python 模块到一个典型的脚本中一样。
- Jython 使用公共对象模型来平滑集成的过程。通过重新实现所有的 Python 内置类型，如基于 Java 的 PyObject 类的字典和列表，Jython 达到此目的。这使得 Jython 不必在两种语言之间不断转换对象；由于 Jython 将 Python 字节码转换为 Java 字节码，所有的对象自然都能被 JVM 访问了。
- Jython 实现中还包括了对接口的改进，这样，比起 Java 所需要的基于类的支持系统来说，

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

正如在 Tk 下开发所熟悉的，使用典型 Python 样式建立 GUI 应用程序要更加容易。例如，Java 中的回调需要被定义为真 Java 类的一部分，而 Jython 的回调则可以只是一个函数。

在开发用于在浏览器环境下使用的 Python 软件时，结果系统提供了很大的灵活性。首先，也是最重要的，开发 Python 应用程序比开发 Java 应用程序更快——使用 Jython 可以开发出在客户端运行的 Python 应用程序，同时仍然具有访问必要的基于 Java 的 GUI 接口库的能力。Jython 也在语言中增加了脚本组件，在具有更好的跨平台支持能力的情况下，可以被编译而且还具有 C 或 C++语言一样的坚固性。

此外，就像 Java 一样，由于 Python 是基于类和对象的，学习在基于 Java 或 Python 的工具上实现基于 Jython 的工具的过程，比起将 Python 解释器嵌入到 C/C++应用程序，要更加直接。

21.2.2　Jython 的局限

尽管 Jython 提供了基本 Java 机器所有的功能和改进，但它并不是完美的。Jython 还有许多限制其能做和不能做的约束。

- Jython 需要 Java。或许这是最明显的，但也是最重要的一点。在大多数地方，都能找到 Java，但是这并不说明所有地方都有 Java。一些公司和个人故意在他们的系统中禁用了 Java，使得 Jython 开发和部署成为不可能。对于其他一些系统，Java 实现可能不稳定或是在不断变化。例如，Microsoft 公司和 Sun 公司最近在 Java 实现上经历的纠纷，会使 Internet Explorer 暂时取消对 Java 的支持，还使 Mac OS X 最初版本的 Internet Explorer 完全不支持 Java。如果需要使用 Jython 进行开发，要保证用户具有支持 Java 的能力。
- 为了利用好 Jython 需要的 Java 知识。尽管能够用 Jython 和基本的 Python 脚本做许多事情，但为了支持 GUI 接口，或是与现有的 Java 组件(如 JDBC)一同工作，就需要会用 Java 编程。如果已经学会了 Python，学习 Java 不是主要的任务，但却是必需的额外步骤。
- Python 的兼容性不像 CPython 实现那样良好和完全。例如，它不能使用 CPython 中简单的结构，例如使用文件(特别是类似于 sys.stdin 之类的标准文件对象)；还有，它的异常系统使用字符串而不是基于类的异常，这更像 Python1.5，对于某些人而言这可能会是关键的问题。由于 CPython 开发不会总是领先于 Jython，所以这些问题也会有交叉。从新的 CPython 实现到同等功能 Jython 的发布，可能会花费一两个月时间。
- 由于总要使用一种被翻译成 Java 的语言，以替代本地的 Java 源代码，所以 Jython 比它的 CPython 版本要慢。这一附加的层自始至终都起作用，当然，根据运行脚本的不同，速度减慢的程度也不同，但是测试表明 Jython 脚本会比 CPython 版本慢上 2 到 100 倍。
- Jython 目前还不兼容 C/C++扩展。这不是 Jython 自身的局限，要使用这些接口进行工作，就必需将 Java 和 C/C++一同工作，或者还要能将 C/C++代码转换成 Java 字节码，——这在目前还是不可能的。不幸的是，这严重地限制了 Jython 在基于浏览器的应用程序之外使用。Jython 的解决方法是重新开发已经用 Java 写成的扩展模块，由于许多第三方模块来自 Vaults of Parnassus，就需要投入很大的工作量进行重新开发。当然，C/C++扩展中的许多元素，如数据库和网络访问，在 Java 类库都有了。

尽管存在所有这些问题，Jython 仍然是一种优秀的工具。如果需要开发基于 Python 的客户端应用程序的跨平台的解决方案，Jython 是最合适的。对于专用于 Windows 的解决方案，请参见本章后面关于“Python 和 ActiveScript”部分的内容。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

21.3　Python.NET

自 CD-ROM 的发明和因特网的飞速发展以来，Microsoft 首创的.NET 是在软件开发和部署方面最重大的变化之一。.NET 架构是基于组件的软件系统，它通过将 XML 和一种单一的连接语言 C#结合起来，允许在一个应用程序中混合和匹配多种语言。.NET 系统同时支持通常的基于主机的应用程序，使得部署分布式因特网应用程序变得更加简单，只要需要，就能够在不同主机之间利用 XML 和任何语言或语言的联合体传送信息。

通过使用 Perl 和可得到的测试版的 Python 实现，ActiveState 公司正在开发适合.NET 架构的组件。正如 Jython，Python.NET 将是 Python 的本机解释器。但是 Jython 将 Python 字节码翻译为 Java 字节码，Python.NET 将 Python 脚本翻译为 C#代码。随后，在本机的 C#、Perl、VisualBasic 甚至 C/C++应用程序中，就可以混合和匹配 Python 驱动的 C#的类和对象。无需专门将 Python 扩展或嵌入进应用程序，就能使用由 Python 写成的对象和方法创建基于 Perl 的应用程序；对于 Python 脚本而言，就能使用通常只能通过 C/C++使用的对象和类。

对于 Python 来说，最大的好处就是兼容性和创建单独的基于 Python 应用程序的能力。使用 Python.NET，可以通过 C#解释器将应用程序直接编译成为.exe 的可执行文件。对于许多 Windows 程序员来说，更有用的是，可以在 Microsoft 新版本的 VisualStudio 中进行.NET 开发，所以可以使用与开发 C/C++和 VisualBasic 同样的工具和 IDE，来开发 Python 应用程序。

Python.NET 还是一项不断发展的技术。尽管可以下载 ActiveState Python.NET 组件的测试版本，但在其翻译系统中还存在许多错误，而且通过 C#还不能支持 Python 所有的功能和模块。

ActiveState 公司自身也是 Python 工具的热心开发者。最初 ActiveState 公司只进行 Perl 的开发，但现在已经在开发 Perl、Python、Tcl、XML 开发人员的解决方案了。他们提供各种产品，从各种语言专用于 Windows 的扩展和安装包，到 Komodo 的 IDE 以及大量文档和样例。

.NET 仍旧停留在想像中，VisualStudio.NET 直到 2001 年 6 月也还是在第二版的测试中。VisualStudio.NET 组件的最终版本将会综合所有这些技术，不会早于 2001 年 11 月问世，Python.NET 和类似的工具将在之后不久才会出现。

如果需要更多的信息，请访问 ActiveState 的网站，具体请参见附录 B 中的资源列表。

21.4　Python 服务器页面(Python Server Page)

Python 服务器页面(PSP)与活动服务器页面(Active Server Pages，ASP)系统对等，它是基于 Python 的。在 Windows 下，由 ActiveState 的 Python 包和 Microsoft 的 IIS(Internet Information Server，因特网信息服务器)提供 PSP 的支持。还有人将 PSP 与 PHP 的服务器端脚本系统进行比较，PHP 使用类似于 Perl 的语言来处理网站服务器端的脚本。

PSP 不同于 ASP 或 PHP，它完全是用 Java 写成的，这就允许在任何支持 Java 的系统下部署 PSP。Python 的长处在于 PSP 使用 Jython，将 Jython 嵌入在 HTML 页面中，在同一工具中提供对于 Python、Jython 以及 Java 所有功能的访问。

不过，与 ASP 和 PHP 一样，PSP 的优点是可以很容易地将代码加入进 HTML 页面。对于

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

那些大量应用 CGI 脚本和把 HTML 嵌入脚本源代码的网站，最要紧的在于摆脱用 CGI 脚本编写 HTML 代码，代之以单纯地提供需要的信息。

关于 PSP 更多的信息，请参见 PSP 站点：www.ciobriefing.com/psp。

21.5 Python 与 ActiveScript

Microsoft 公司的因特网信息服务器(IIS，Internet Information Server)提供的服务称为 ActiveScript。ActiveScript 是 IIS 系统的扩展，允许在 HTML 页面中嵌入脚本语句。这些脚本元素可以被服务器或客户端执行，以在 Web 页面中提供额外的功能。

ActiveScript 不特定于语言。尽管过去主要支持 VisualBasic 脚本(VBScript)，ActiveScript 也能通过合适的扩展配置成支持 Python、Perl 以及许多其他语言。最初，Microsoft 公司向 ActiveState 公司提供了部分资助，用于开发使用 Perl 的 Windows 专用工具，这些工具能提供 Perl 和 Python 的合并 ActiveScript 功能的版本(分别是 ActivePerl 和 ActivePython)。

由于 ActiveScript 既是服务器端又是客户端的技术，所以具有很多用途。例如，可以将 Python 脚本嵌入到 HTML 页面，不需要 CGI 脚本，甚至不需要与浏览器进行任何通信，就能从表单中接受输入、处理信息和显示结果。

客户端 ActiveScript 的缺点是：需要有运行了 Internet Explorer 的 Windows 计算机，同时还要安装了 Python。这严重地限制了 ActiveScript 的使用，不过在封闭的环境，如在企业内联网环境中，ActiveScript 还是可以取代 CGI 脚本。

也可以将 ActiveScript 作为服务器端脚本的一种选择。这种模式下，除了是被嵌入在 HTML 中并在最终文档发送给客户端之前被分析之外，ActiveScript 与任何其他 CGI 脚本的工作方式都是类似的。

ActiveScript 最重要的特性就是：可以通过 COM(Common Object Model，通用对象模型)接口与 Windows 应用程序通信。COM 允许与其他应用程序提供的对象进行会话和交互。无论是作为 COM 客户端还是 COM 服务器，Python 都提供 Windows 下的支持。COM 是 Windows，尤其是 Microsoft 应用程序的主要组件。可以使用 COM 在 Python 中打开并打印一个 Word 文档，或者将 Python 作为应用程序服务器使用，利用应用程序专用的 Visual Basic(VBA，Visual Basic for Application，内置在 Microsoft Office 和其他应用程序中的宏语言)执行 Python 语句。

COM 本身内容很多，这里没有更多的篇幅来介绍其细节，不过在介绍 Python 用到的一些平台专有的扩展时，还会介绍到 COM 和 Python 的。更多的信息请参见第 22 章。

21.6 Mailman

检查您的电子邮件信箱，如果您正好属于某个邮件列表，可能会发现其中许多都是由 Mailman 管理的。Mailman 完全基于 Python，支持所有常用的邮件列表选项，例如摘要、基于邮件的讨论、以及简单的仅发送的声明列表。

Mailman 与其他许多实现的区别在于：Mailman 还包含一个基于 Python 的 Web 界面。这一 Web 界面能够被用户和邮件列表所有者使用，用于控制订阅情况以及邮件列表选项。似乎这还

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

不够，Mailman 还提供了垃圾邮件的过滤器，并且为了将邮件列表和新闻服务器集成在一起，还提供邮件到新闻的网关。

Mailman 可能是 Python 最具威力的用途之一了，许多人，包括 Python 的倡导者以及其他人，可能都不知不觉中使用着 Python。

21.7　Grail

Grail 是一种基于 Python 的 Web 浏览器，它使用 Tkinter GUI 系统来实现用户界面。除了可以查看网站外，Grail 还可以执行作为客户端 applet 的基于 Python 和 Tkinter 的脚本，它们可以直接嵌入 Web 页面中。在使用上，尽管由于 Grail 的开发已经停滞了很长时间，不能支持许多更新的标准，但是外观还不错，类似于 Netscape 或 Internet Explorer。

作为一种最早的完全用 Python 写成的大型应用程序，Grail 的历史很悠久。这归功于 Python 众多的因特网功能，包括一般的如“urllib”之类的下载模块、HTML 分析器以及一般的协议库如“ftplib”和“httplib”。

不过，真正的优势在于能够使用 Python 脚本，但更明显的是，可以在 Web 页面中使用基于 Tk 的窗口小部件(widget)，在 Macromedia 的 Flash 之类的系统或基于 Jython 的内容的应用之前很久，这些部件就已经被使用了。内容本身可以是一个简单的 widget，或者是从浏览器通过表单传来的信息。更进一步，由于 Grail 还具有访问整个 Python 语言的能力，所以可以在一定程度上，建立任何需要的通过浏览器来进行访问的应用程序。

很不幸，Grail 不再被正式地维护了，也不再是 Netscape 或 Internet Explorer 的竞争者了。如果需要基于 Python、不基于 Grail 或 ActiveScript 的客户端 Python 实现，应当使用 Jython。

21.8　Apache 与 Python

如果熟悉 Apache，而且已经使用 Apache Web 服务器来支持网站，那么会很高兴地听说很多人正致力于 Python 和 Apache 的集成。基本的 Apache 包已经能够通过标准 CGI 接口支持运行服务器端的 Python 脚本。

通过将 Python 解释器嵌入到 Apache 服务器中，PyApache 扩展努力提高了 Python 脚本的执行速度。由于不需要产生外部应用程序来处理脚本的执行，自然会提高 Python 脚本执行的速度。系统也能反向工作，允许在 Apache 中使用 Python 构造，这很适于处理配置信息和授权过程。例如，不用使用内置的授权方法，就能够通过数据库让 Python 给用户授权。

通过在 Apache 本身建立 Python 解释器，用于 Apache 的“mod_python”扩展模块将为 Apache 提供类似的功能。“mod_python”的区别在于：它在 Apache 的内存中以字节码格式保存 Python 脚本，从而将其优化并准备好执行格式。由于代码驻留在内存中，而且已经被分析和编译，所以会使速度有明显提高。“mod_python”扩展与“mod_perl”扩展是类似的，它以类似的方式使 Perl 脚本的执行速度提高到通常的 CGI 脚本速度的 100 倍。当编写本书时，Apache 的“mod_python”扩展还处于测试状态，但是有“mod_perl”成功的基础，相信“mod_python”扩展能够安全地带来速度上显著的提高。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

21.9　SocketServer 与 BaseHTTPServer

尽管不是每个人都需要，但是可能会有人要使用符合 Python 解释器的标准模块和类，创建自己的基于套接字(socket)和基于 HTTP 的网络服务器。实际上在第 13 章已经介绍了这样的一些例子。

监听请求并且提供服务实际上就是打开相应的端口，等待客户机的连接。然后，要决定是将连接通过“fork()”方法传递给新的进程，还是启动一个新的线程来处理请求，还是使用“select()”方法以循环的方式处理请求。线程显然优先考虑，但是不能被所有平台支持。

如果需要为专门的 HTTP 请求提供服务，使用“BaseHTTPServer”或者“CGIHTTPServer”可能是最简单的方法。如果需要的话，可以采用这些类并根据自己的使用情况进行修改。只需要一点点工作量，就能够创建单独的基于 Python 的 Web 服务器，它也具有内置的与数据库的连接，还为连接存放状态信息，比起通过通常的基于 CGI 的接口进行访问，这要更快一些。

使用这些模块格外简单，下面的脚本显示出使用 Python 和这些模块建立 Web 服务器是非常简单的：

```
import os, BaseHTTPServer, CGIHTTPServer

os. chdir( '/ export/ home/ mc/ html')
httpserver =
BaseHTTPServer. HTTPServer(( '', 8081), CGIHTTPServer. CGIHTTPRequestHandler)
httpserver. serve_ forever()
```

当心，在 Unix 下，为了启动端口号在 1000 以下的服务器，需要有超级用户(SuperUser)(root)特权。遇上这种情况时，在主页目录的 html 目录下，可以在端口 8081 启动 Web 服务器。要更详细地了解 BaseHTTPServer 和 CGIHTTPServer，可以查看作者的《Python 文档详解》(《Python Annotated Archives title》，见目录 B)，书中会对这些模块进行逐行的分析。

21.10　Medusa

为了支持建立高性能和高效用的基于套接字的网络服务器，Medusa 提供了必要的系统。从层次结构上看，Medusa 位于 SocketServer 和 select()之上，处理所有的支持多数据流和客户端的问题。

Medusa 起初是一对单独的模块：asyncore 和 asynchat，它们一同工作，提供异步的多路 I/O，但是没有使用线程和 fork()技术。代替这些模块的是使用建立在 select()系统调用上的事件的循环。由于 Medusa 使用 selelct()，所以只适合于快速短暂的事务。

从 v1.5.2 版本以来，Medusa 核心的组件就是标准库的一部分。完整的 Medusa 系统也包括 HTTP 和 FTP 服务器，对于非商业性用途，Medusa 是免费的，可以从“www. nightmare. com/ medusa”下载完整版本的 Medusa。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第 5 部分　跨平台开发

第22章　跨平台开发的路径

有大量平台支持 Python，因而 Python 经常用于跨平台开发。把所有支持 Python 的平台类型都一一列出，没有什么意义。但此处还是有必要指出：Python 已经在最常用的平台上使用，如 Unix、Windows、Mac OS(包括 Mac OS X)、OS/2、BeOS 等等许多平台，甚至于可以拥有一个在 Palm OS 系统上运行的版本，此版本叫作 Pippy。

要开发一个真正跨平台的应用程序并不容易。即使忽略像用户接口这样的一些问题，仍然还有一大堆围绕诸如行终止、环境变量和标准输入输出这些领域的后台问题。当然，平台不同，平台所支持的函数清单也不同，即使是，在 Unix 系统的不同变体间，受到支持的函数清单也不同。

实际上，Python 为程序员隐含地处理了跨平台开发的大量复杂问题。Python 只包含了很少数量的内置函数，主要的外部模块都为大多数平台所支持。在模块调用时，Python 对给定的函数不提供支持或提供准支持。

实际上，本书其他地方也同样存在着这样的问题。比如，在接口构造里， Tk 工具包当前只提供了兼容 Unix、Windows 和 Mac OS 系统的解决方案。详见本书第 15 章。有关对文件读写、数据和字符的支持情况，请查看本书的第 9 章、第 10 章和第 11 章。

本章的剩余部分将关注一些影响 Python 跨平台开发的主要问题。也可以访问 MC words 网站(http://www.mcwords.com)，在那里可以得到关于跨平台支持和技巧的最新信息。

22.1　基本平台支持

平台之间最明显的不同就是对不同函数的支持。Python 在外部模块里隐藏了许多关于跨平台开发的复杂操作。大多数情况下，在一个通常不支持跨平台开发选项的平台上，则采用另一个选择或者绕过此问题。事实上，如果坚持使用 Python 的大多数标准的模块，则可以很好地避免许多问题。

但是那样做并不能完全避免，下面列出几个要注意的项目：

- 一些查看 Unix 文件详细信息的函数，这些文件存放在 Unix 中心的/etc 中。这些文件包括网络信息例程，也包括一些这样的例程：与组和(或)密码信息(如通过 pwd 模块和 grp 模块所支持的)有关的例程。一些平台必须要使用 pwd 模块和 grp 模块，但是当 pwd 模块和 grp 模块在指定的平台上无效时，通常会有一些平台特定的等效模块可用。
- 所有的基本文件接口选项都能起作用。但是，其他如 stat()函数，就可能因为操作系统所

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

使用的底层文件系统而受到有限制的支持。比如说，并不是所有的平台支持修改和访问的次数。文件权限还需要严格查看—— chmod()函数和 chown()函数是特定于 Unix 系统。在非 Unix 平台上，大多数接受一个权限值(诸如 os.open())的函数能安全地忽略这个权限信息。

● 还应当记住：虽然 Mac OS 和 Windows 平台通过别名和快捷方式支持链接这个概念，但是 link()函数和 lstat()函数却常常不起作用。

● 其他许多平台并不支持对操作系统表的内部访问。特别是那些需要返回惟一进程和组 ID 的平台以及返回关于一个进程 ID 的父组和父组所有者信息的平台。

● 并不是在所有的平台上都能使用 exec*()函数或类似功能的函数，它们以这样的方式运行应用程序：应用程序的名字通过类似于接口的命令行提供。特别是在 Mac OS(不是 Mac OS X)系统上，没有命令行接口，内部就不支持像 os.system()这样的函数。而 os.popen()函数对 Mac 应用程序来说完全没有意义，因为 Mac 应用程序不理解允许获得第一手信息的标准输入输出的含义。

22.2　运行环境

Python 的运行环境对脚本有很大的影响。很多问题出现的原因是因为使用了 Python 外部的信息或外部性能，而这些外部的信息及外部性能完全可以直接在内部获取到。下面是这样的一些例子：

● 不要通过环境变量来获得有关用户、组、主机名、用户名或路径的信息。这些变量很容易被覆盖或错误地产生。在 Unix 之外的平台上，甚至于不能设置这样的变量。可以换一种方法，使用 os 模块，通过其中的 getuid()函数及其他函数来直接获得信息。确认在不支持用户 ID 的平台(Windows、Mac OS)上，有账户的备份计划。如果决定使用惟一 ID 或其他标识字符串中的一个变量，这个建议特别有用。有关环境变量和可选项的清单，请参见表 22-1。并请参见表 22-2，表 22-2 是特定于 Windows 系统的环境变量清单和支持对应环境变量的平台。

● 不要依赖于 PATH 环境变量中有效的路径执行的命令——自己设置路径，最好是给应用程序提供完整路径。请记住，根据不同平台的账户，使用与平台相适合的路径。

● 除非在必须的情况下，否则不要依赖信号(signal)。有些平台支持信号和信号处理程序，而其他一些平台却不支持，而另外一些平台可能仅仅支持相对于 Unix 环境下那些可用环境变量的一个减少了的集。

● 请使用共享文件或网络套接字在进程之间交换信息，或者使用线程(参见本书第 10 章)和共享变量在进程之间交换信息。因为并不是所有的平台和 Web 服务器支持同样范围的环境变量，所以在 Web/CGI 环境中使用环境变量时应当多加注意。要获取更多信息，请参见表 22-3。

表 22-1　Unix 机器中的环境变量

变　量	说　明	备　注
COLUMNS	当前显示的列数。在开发终端/文本接口时可以用来判断当前终端的大小	无

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

变　量	说　明	备　注
EDITOR	用户编辑器的首选。如果没有找到，则默认使用 vi 或 emacs，而在 Windows 下，默认使用 C:\Windows\Notepad.exe	无
EUID	当前进程的有效用户 ID。使用 os.geteuid()函数，此函数是正确地用 EUID 变量填充的	Os.geteuid()
HOME	用户的主目录。用 pwd.getpwuid()函数来获得这个信息	pwd.getpwuid()
HOST	当前主机名。socket.gethostname()函数提供了能在中性平台取得主机名的方法	socket.gethostname()
HOSTNAME	当前主机名	socket.gethostname()
LINES	当前显示器或是终端窗口支持的行数。参见本表的 COLUMNS	无
LOGNAME	用户登录。使用 getpass.getuser()函数，或更好的，联合使用 os.getpwuid()函数和 pwd.getpwuid()	getpass()、getuser()
MAIL	用户电子邮件文件的路径。如果没有找到，那它可能位于/var/mail/LOGNAME 或/var/spool/mail/LOGNA ME 下	无
PATH	用冒号分开的目录列表，此目录列表是查找要运行的应用程序时所搜索的目录列表。为了避免安全风险，最好利用对要运行的应用程序的完整路径，或者在脚本中生成路径 PATH	无
PPID	父进程的标识符(ID)。没有简单的方法来查找它，但有时候，却需要它	无
PWD	当前工作目录。可以使用 os.getcwd()函数	os.getcwd()
SHELL	用户使用的 Shell 的路径。此数值可能被误用，使得终止替代真实 Shell 的合适程序的运行。如果不能确定路径，最好的默认路径是/bin/sh	无
TERM	当前终端和仿真终端的名字/类型。见本表的 COLUMNS	无
UID	用户的真实 ID	os.getuid()
USER	用户登录名。见本表的 LOGNAME	getpass.getuser()
VISUAL	用户优先选择的可视化编辑器。见本表的 EDITOR	EDITOR
XSHELL	在 X Windows 系统中使用的 Shell。见本表的 SHELL	SHELL

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

表 22-2　Windows 系统的环境变量

变　量	平　台	说　明	备　注
ALLUSERSPROFILE	2000	当前使用的通用配置文件的位置。没有方法来确定这个信息。	无
CMDLINE	95/98	包含要运行的应用程序的名称的命令行。sys.argv 对象可以生成这个信息	sys.argv
COMPUTERNAME	NT，2000	计算机名。Socket.gethostname()函数返回当前主机的 DNS/因特网名字	无
COMSPEC	所有平台	命令解释器（通常是 COMMAND. COM)的路径，当打开命令提示时使用	无
HOMEDRIVE	NT，2000	用户的主驱动器的驱动器符号(带冒号)	无
HOMEPATH	NT，2000	用户主目录的路径	无
HOMESHARE	NT，2000	用户主目录的 UNC 名。如果它没有设置或设置为本地驱动器，那么这个值为空值	无
LOGONSERVER	NT，2000	验证用户身份的域名服务器	无
NUMBER_OF_ PROCESSORS	NT，2000	当前机器中激活的处理器数。	无
OS	NT，2000	操作系统的名字。sys.platform 对象保存这个平台字符串，通过它知道是运行在 Windows 3.1 (win16)下还是运行在 Windows95 - 2000(win32)下	sys.platform
OS2LIBPATH	NT，2000	与 OS/2 兼容的库的路径	无
PATH	所有平台	在命令提示符下，使用 os.system ()或 os.exec*()函数运行程序时，搜索应用程序的路径	无

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

变　量	平　台	说　明	备　注
PATHEXT	NT，2000	用来标识可执行程序的扩展名列表。最好不要修改这个值，但是如果需要自己手动定义时，.bat、.com.和.exe 是最重要的	无
PROCESSOR_ARCHITECTURE	NT，2000	当前机器的处理器的结构	无
PROCESSOR_IDENTIFIER	NT，2000	标识查询中央处理器(CPU)时返回的信息标记	无
PROCESSOR_LEVEL	NT，2000	处理器的级别：386 处理器是 3，486 处理器是 4，奔腾处理器是 5，MIPS 处理器在 3000 到 4000 之间，Alpha 处理器是 21064	无
PROCESSOR_REVISION	NT，2000	处理器的修订号	无
SYSTEMDRIVE	NT，2000	保存当前活动操作系统的驱动器。一般位于 C:	无
SYSTEMROOT	NT，2000	活动操作系统的根目录，可能是 Windows 或 Win	无
USERDOMAIN	NT，2000	当前用户连接的域。	无
USERNAME	NT，2000	当前用户名	无
USERPROFILE	NT，2000	用户配置文件的位置	无
WINBOOTDIR	NT，2000	用来引导机器的 Windows 操作系统的位置。见本表的 SYSTEMROOT	无
WINDIR	所有平台	活动的 Windows 操作系统的位置。当搜索动态连接库(DLL)和其他系统(OS)信息时使用这个目录。见本表的 SYSTEMROOT	无

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

表 22-3　当运行公用网关接口(CGI)脚本时定义的环境变量

环 境 变 量	平　台
CONTENT_LENGTH	Apache，IIS
DOCUMENT_ROOT	Apache
GATEWAY_INTERFACE	Apache
HTTPS	IIS
HTTP_ACCEPT	Apahce，IIS
HTTP_ACCEPT_CHARSET	Apache，IIS
HTTP_ACCEPT_ENCODING	Apache，IIS
HTTP_ACCEPT_LANGUAGE	Apache，IIS
HTTP_CONNECTION	Apache，IIS
HTTP_COOKIE	IIS
HTTP_EXTENSION	Apache
HTTP_HOST	Apache，IIS
HTTP_IF_MODIFIED_SINCE	Apache
HTTP_UA_CPU	Apache，IIS
HTTP_UA_OS	Apache，IIS
HTTP_USER_AGENT	Apache，IIS
LOCAL_ADDR	IIS
PATH	Apache
PATH_INFO	IIS
PATH_TRANSLATED	IIS
QUERY_STRING	Apache，IIS
REMOTE_ADDR	Apache，IIS
REMOTE_HOST	Apache，IIS
REQUEST_METHOD	Apache，IIS
REQUEST_URI	Apache
SCRIPT_FILENAME	Apache
SCRIPT_NAME	Apache，IIS
SERVER_ADDR	Apache
SERVER_ADMIN	Apache

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

环境变量	平台
SERVER_NAME	Apache，IIS
SERVER_PORT	Apache，IIS
SERVER_PORT_SECURE	IIS
SERVER_PROTOCOL	Apache，IIS
SERVER_SIGNATURE	Apache
SERVER_SOFTWARE	Apache，IIS
TZ	Apache

22.3 行终止

在多个平台上使用 Python 的最基本问题是在读写文件时的行终止。不同的操作系统为行终止提供了不同的字符。比如，Unix 使用换行符(\n 或\012 字符)作为行终止，而 Mac OS 使用回车符(\r 或\015 字符) 作为行终止。更复杂的是 DOS/Windows 使用的是回车符，换行符序列(\r\n 或\015\012)。

可以使用 os.linesep 变量来判断当前是否是行结束序列。在本地平台上，Python 自动使用向右的顺序来读写文件。但是，如果把 Python 脚本从 Mac 传送到 Unix 时，就有可能出错，因为 Python 解释器不能理解非当前平台的其他行结束标志。

在处理来自另一平台的文本文件时也一定要小心。如果对文本文件的内容和格式有任何怀疑，先打开文件再用 string.split()函数分离元素，从文件中提取出实际的行数。要知道，有些平台使用的文本文件可能来自于其他平台。比如，虽然 Unix 使用的是换行符，它也可以正常读取 Windows 的文本文件，但完全忽略 Mac 的行终止符。类似的，Macs 读 Windows 的文件时，只读到每一个回车符号处,把换行符作为每一新行的第一个字符。

读写二进制文件时，应注意二进制选项的习惯说明。虽然在 Unix，Mac OS 以及其他多数平台上都没有要求，但是在 Windows 和其他一些平台上，读写二进制文件时有所不同。有些平台在读取文件期间就自动截断行终止。

在使用网络套接字时，常常使用回车或换行符。虽然在多数条件下不要求套接字，但是所有描述协议的注释的请求(Request For Comments，RFC)都指定回车/换行符作为行终止符，使用完整序列将确保与期望使用完整序列的系统兼容。

22.4 字符集

另外一个比较普遍的错误是，认为所有平台都使用同一字符集。虽然大部分平台都使用 ASCII 字符集，但是您只能使用前 128 个相同的字符(从 0 到 127，或\00 到\0177)。出于这种考虑，在不同平台之间，没有必要依靠 chr()或 ord()函数返回的值。字符集里的真正值可能有很多

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

种字符，有可能有重音符号以及不同的顺序。

要考虑 Unicode(参见第 10 章)的使用，它可以避免大多数字符集所遇到的平台、字体和国家本地化问题。要确保读写的信息正确，必须小心谨慎。

如果关心本地化的问题，应考虑把规则的或常量字符串(消息，帮助文件等等)放置到外部文件里，这样可以根据不同的需要来加载指定语言的文件。添加新的语言就是添加一个新语言文件的情况。

22.5　文件和路径名

主要的 3 种平台显示用来分开组成文件路径的目录和文件的字符的范围。Unix(和 BeOS、QNX 和 Mac OS X)系统使用“/”，但是 DOS 和 Windows 系统使用“\”，而 Mac 系统则使用“:”，Windows 和 DOS 实现也允许使用 Unix 下的“/”来分开元素。更复杂的是，只有 Unix 和少数几种其他操作系统使用的是单独根目录的概论。

Windows 和 DOS 中的驱动器或分区，用一个字母和跟在后面的冒号来标识。在 Mac OS 中，每一个卷都有一个路径名，用标准冒号分开。os.path 模块可以用适当的字符集和分隔符创建和处理路径。不同的平台支持不同的文件名字和长度。下面是大致的差别：

- DOS 支持的文件名，长度不超过 8 个字符，扩展名不超过 3 个字符，并忽略大小写。
- Windows 95/NT 的定义稍微有些复杂：单独的路径组成部分最长可达 256 个字符，并区分大小写。
- 在 Mac OS 下，路径的任何元素可以长到 31 个字符，目录名与大小写无关，也就是说不能同时有一个文件叫“File”又有一个文件叫“file”。Mac OS X 的 HFS+和 ufs 分区支持长达 254 个字符的路径元素。但是，只有 ufs 分区支持区分大小写的文件名。在 Mac OS 和 Mac OS X 下的 HFS/HFS+分区中，“FileName.TXT”和“filename.txt”两个名字是等价的。在写入一个文件时，文件名相同但是大小写不同，可能会导致异常。
- 旧版本的 Unix 仅支持：每一个路径单元(也就是目录和文件名)长度不得超过 31 个字符。在新版本中，包括 Solaris、HP-UX 10.X 以上还有 Linux，支持：每一路径单元最多 256 个字符。注意，不管怎样，在不同的平台上，一个完整路径的最大长度是 2 048 或 4 096 个字符，

文件名应尽量使用标准字母字符。

22.6　数据不一致性

不同的物理系统和操作系统的组合使用不同序列来存储数字。这种特性对在不同系统之间，或在网络连接之间或文件里，存储和传送二进制数有影响。解决这个问题的方法就是使用标准字符串来保存信息(显然对大数字来说很浪费)，或者使用 pickle 或 struct 模块来把信息变成标准格式，以便很容易地进行组装和分解。

还应记住，考虑到诸如时间这样的数据。在不同的系统中处理的时间数据有可能不同。差不多在所有的系统中，新纪元时间为 1970 年 1 月 1 日 0:00:00。但是有的平台(如，Mac OS 平

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

台)定义的却是别的值。如果您想把日期存储成一种中间结构的格式，应使用一种不依靠新纪元值的格式，比如像 YYYYMMDDHHMMSS 一样的字符串。

22.7 性能和资源

表面上看来，并不是所有的平台都像 Unix 操作系统那样，对可用资源没有限制。虽然 Windows 提供了类似的内存接口，但是 Windows 机器和 Unix 相比，可用的内存要少于实际的物理和虚拟内存。

虽然内存的价格一直在降，1GB 的内存(RAM)已不到 150 美元，但是许多用户仍然只用比较小的内存，考虑到操作系统(OS)和其他应用程序对内存的占用，只有一小部分内存应用程序可以使用。例如，典型的 Windows2000 服务器的基本操作系统(OS)和后台服务大概要用到 223MB 内存。在 Mac 中，典型的操作系统(OS)要用 64MB，甚至达到 80MB。

遗憾的是，Python 很容易就使用了大量内存，而不考虑其他原因。可以很轻松地创建一个巨大的列表，元组和字典，甚至可以把一个文件内容全部读到内存中。

一般建议不要把整个文件内容输入到内存中，如果要用到大的列表和字典，考虑使用 DBM 数据库，Berkeley 数据库系统(Berkeley DB system)(在 bsddb 模块里)支持标准 DBM 系统的一个数组表单。

还应记住，其他操作系统不提供 Unix 的受保护内存空间或多任务特性。特别是 Mac OS(不是 Mac OS X)使用固定大小的应用程序分区。如果分配给 MacPython 应用程序的分区不够大，在程序载入或处理太多的信息时会遇到许多问题。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第 6 部分　深入 Python

第23章　Python体系结构

在把脚本名提供给 python 解释器时，对所发生的操作大概不会太在意——仅仅是等待应用程序运行完成的结果。事实上，在按下回车键和脚本开始实际操作这两个时刻之间已经发生相当大的改变。

许多解释性语言读取源文件，并一行一行地逐行运行读出的命令。这导致了许多麻烦和频繁的无效操作。被 BASIC 的大多数版本和许多 Unix shell 使用的逐行解释的解释器必须依赖在执行那个命令行时才了解的信息。如果引用了一个不存在的变量或函数，解释器就中止—— 不管前面已经运行了多少行。复制过的、改过名的，或在进程中生成的文件仍然存在着。不要太在意可能毁坏了的脚本。使得应用程序很难重新启动或者不可能重新启动的原因不是脚本的设计问题，而是因为在脚本的中间位置上中止了运行。

与 Perl、Rebol 以及其他一些语言一样，Python 采用了不同的方法。如果看过图 23-1，一定知道在 Python 编译器上上述脚本会立即通过。这只是实际所启动的 Python 应用程序的一个嵌入部分。

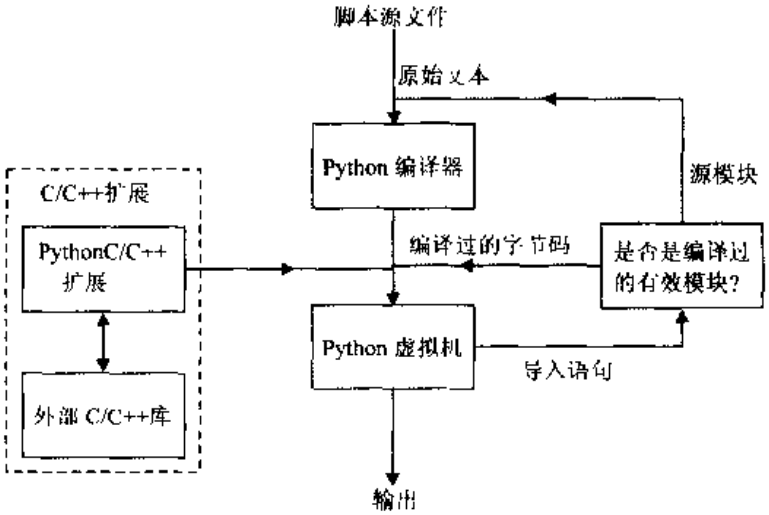

图 23-1　Python 运行结构示意图

Python 编译器把所有源素材转换为字节码。字节码是一种和计算机的 CPU 所使用的汇编码相似的内部格式码。转换后，字节码被送到运行字节码的 Python 虚拟机上以产生所期待的结果。

虽然这看来是一个复杂的步骤，但是这样做使 Python 生成可适合运行的高度优化脚本版本。而且因为整个脚本被编译成字节码，解释器在程序开始运行之前就可以对程序出问题做

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

标记。

除了所提供的语句之外，所用任何 import 语句只有在被编译了之后才运行。这个导入进程由第 5 章讲到的 Python 虚拟机触发一个检查。基本上，如果这个模块是最新的并且是字节码格式，那么解释器就会直接导入已编译的字节码并使用它。如果这个模块还不是字节码格式，或是这个源文件很接近字节文件，那么就被传送到编译器生成字节码再继续运行。

请注意，在左边的图 23-1 中，对利用外部 C/C++库的内部函数和模块的调用使用 C/C++扩展模块中所定义的字节码。这个扩展模块依次引用了所使用的 C/C++库中的底层对象、数据和函数。详见有关加载模块时使用的优先级和顺序的第 5 章。

进程的整个运行过程瞬间就完成了。Python 程序平均只需使用几百分之一秒就能把原始源码转换为字节码。一旦解释器拥有字节码，由于底层字节码的高度优化天性，运行过程能相当迅速地完成。

本章主要目的在于：介绍 Python 解释器的基本机制——单独语句是怎样被 Python 虚拟机执行的，用来保存 Python 脚本信息的内部结构，以及有关字节码自身的细节。

本章还提供了有助于理解本书后面部分信息的主架。例如，在第 24 章中，可以系统地学习 Python 调试器，调试器使用本书第 25 章所论述的属性(对象变量)，在第 26 和 27 章中会学习到怎样使用 C/C++扩展外部库来扩展 Python 解释器，和把 Python 嵌入到 C/C++应用程序中的方法。

23.1　名称空间、代码块和帧

忽略 Python 解释器使用的字节码的语义(其原因请参见本章后面的“字节码”部分)，了解在运行期间解释器处理源代码中的语句的方式是有用的。

就最简单的观点来讲，Python 脚本被分为不同的代码块。代码块在一个限定的运行帧中运行，代码块和帧都遵守名称空间制度，名称空间制度对运行的代码而言，控制哪个函数、对象和类是有效的。最后，当错误出现时，可以使用 traceback 对象中的信息。traceback 对象就是异常所使用的对象，它们报告导致异常的错误的位置。

让我们单独地分析每个元素，从而确定它们对 Python 脚本运行的影响。

23.1.1　代码块

在 Python 脚本里，每一个可运行的元素都由代码块组成。代码块是脚本里可执行的任何元素；代码块包含模块、类定义和函数体。另外，在输入到交互接口的语句也作为单独的独立代码块来对待。如，命令行中提供的脚本文件，以及在 exec 语句和 eval()、execfile()函数运行期间读入的字符串和文件。

这些代码块本身所包含的内容并不比编译过的字节码多什么，就是由要运行的语句组成。它们完全不能识别它们运行的上下文(context)，或是可能要访问的变量与其他的函数。

关于代码对象的更详细信息参见本章后面的“代码对象”部分。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

23.1.2　帧

代码块在一限定的帧内或是特殊的可执行帧中运行。一个可执行的帧由以下几部分组成：

- 要运行的代码块
- 代码块的位置(行数、源文件等等)
- 对下一个要运行的帧的引用
- 全局名称空间的内容
- 局部名称空间的内容

每个帧还有一个代码块堆栈，代码块堆栈包含其代码块中定义的代码块的列表。堆栈通过代码块中的不同的嵌套循环、try 语句和 if 语句来合并不同的代码块。在运行过程中，这个堆栈用来传递循环信息。例如，一个 for 循环给提供给它的序列的下一条目求值，然后把所有关于循环的信息(当前索引，序列和代码块)放到代码块堆栈上，代码块堆栈在每一次迭代期间再次出栈，这样又给下一条目求值。

在脚本中，帧是对代码的最小的有用的引用。帧在调试时用于验证。实际上，运行是在帧之间实际中断的，虽然是包含脚本可执行元素的代码块，但是也是包含代码块上下文的帧——包含了诸如从其中生成代码块的初始源码的行数的信息。

有关帧中所保存的数据信息请详见本章后面的“帧对象”部分。

23.1.3　名称空间

名称空间把名字——变量、函数和模块——映射为对象。典型的脚本由多个名称空间和可能引用同一个名称空间的多个帧组成。名称空间本身就是字典对象，字典对象的键引用名字，字典对象的值是对实际对象的引用。见第 3 章关于对象的介绍，以及在 Python 中怎样存储和引用对象。

如果还记得，第 5 章介绍了根据 LGB 规则来搜索的名称空间的概念。LGB 规则中的 L 指的是局部名称空间，是定义名字的名称空间以及相对于当前运行帧进行搜索的名称空间。LGB 规则中的 G 指的是全局名称空间，包含 global 语句声明的变量。全局名称空间变量的搜索不受当前运行帧的限制。LGB 规则中的 B 指的是内置函数和对象。如果通过 LGB 规则不能找到指定的名字，将引发 NameError 异常。

表 23-1 列出了不同的代码块类型及在代码块创建时，全局和本地名称空间从其中构造的位置。请记住，全局名称空间一般指封装模块或脚本的名称空间——全局名称空间不嵌套。

表 23-1　代码块的名称空间源/定义

代码块类型	全局名称空间	局部名称空间
模块	模块的名称空间	与全局相同
脚本	__main__用的名称空间	与全局相同
交互命令	__main__用的名称空间	与全局相同
类定义	包含代码块的全局名称空间	新的

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

代码块类型	全局名称空间	局部名称空间
函数体	包含代码块的全局名称空间 (可以覆盖)	新的
传送给 exec 语句的字符串	包含代码块的全局名称空间 (可以覆盖)	封装块的名称空间 (可以覆盖)
传送给 eval()函数的字符串	调用者的全局名称空间 (可以覆盖)	调用者的局部名称空间 (可以覆盖)
execfile()读取的文件	调用者的全局名称空间 (可以覆盖)	调用者的局部名称空间 (可以覆盖)
input()读取的表达式	调用者的全局名称空间	调用者的局部名称空间

23.1.4　回跟踪(tracebacks)

回跟踪对象在异常发生时创建。它包含对当前运行帧、当前行和异常发生时实际运行指令的引用。另外，当异常发生时，为了把单个的运行状态翻译成为正常重新生成的记录，回跟踪对象还包含对下一层堆栈记录的一个引用。遍历整个对象列表将得到回跟踪记录。

23.1.5　综合

综合很重要的原因是，因为它演示了 Python 是怎样运行脚本，怎样得到记录运行时当前脚本出错的错误和调试脚本的信息。

通过回跟踪(traceback)对象得到的信息，可以找出哪里发生的错误。当异常产生时，调用 sys.exc_info()函数可以得到回跟踪对象。实际上，函数调用返回的是包含对引发的异常类的引用、一个类实例和回跟踪对象的元组。

在回跟踪对象的武装下，可以和运行帧，因此，可以找到源代码中发生错误的行数。

在调试时，执行被分解为不同的帧；同时，行在这些帧中大多数用于其他语言的调试器是以这样的逐行处理为基础工作：在语句当中包含函数调用时，具有一个隐含“帧”。此隐含元素紧接着“进入”函数调用——在 Python 中，这只不过就是后随的运行帧树。

请注意，由于这个特性， Python 中的一个单行可能与多个帧相关；如果一条语句包含对几个独立函数的调用，那每一个函数调用就都有一个运行帧。例如，下列命令行：

```
hypot = math.sqrt(square(a) + square(b))
```

实际上由三个帧组成：第一帧用于运行代码对象与第一个 square()调用相关，第二帧是关于 square 的第二次调用，第三帧是关于 math.sqrt()的调用。在调试器中，这一行的运行分为 square(a)，然后是 square(b)，最后是 math.sqrt()。

相反的，在一帧内，调试器也可以根据独立的行来工作。例如，下面的函数有四行，每一

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

行都没有其他的帧。

```
def calc(a , b , c):
    resa = (a * b) / c
    resb = (c * b) / a
resc = (c * a) / b
return resa + resb + resc
```

23.2　内置类型

Python 解释器有 26 种不同的内置类型，并根据其基本类型分类。例如，所有属于 Number 类型都共享相同的基本属性，所有的顺序类型数据都使用相同的方法来访问。

表 23-2 列出了所有的对象类型和它们的类别。

表 23-2　Python 支持的对象类型和类别

类　别	类　型	说　明
None	NoneType	空对象(可以使用 None)
Numbers	IntType	整数型
	LongType	长整数型(任意)
	FloatType	浮点数型
	ComplexType	复数型
Sequences	StringType	字符串型
	ListType	数组和列表型
	TupleType	元组型
	XRangeType	由 xrange()内置函数创建
Mapping	DictType	字典
Callable	BuiltInFunctionType	内置函数
	BuiltinMethodType	内置方法
	ClassType	类对象
	Functiontype	用户自定义的函数
	InstanceType	类对象实例
	MethodType	约束类方法
	UnboundMethodType	非约束类方法
Modules	ModuleType	模块
Classes	ClassType	类定义
Class Instance	InstanceType	类实例

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

类　别	类　型	说　明
Files	FileType	文件
Internal	CodeType	编译过的字节码
	FrameType	运行帧
	TracebackType	异常的回跟踪堆栈
	SliceType	由扩展切片生成的类型
	EllipsisType	在扩展切片中使用的类型

虽然您已经学习了基本对象的类型和它们的属性，但是，还不知道怎样使用这些内部类型来保存信息，比如，本章前面介绍的模块、函数、代码、帧和回跟踪对象的特定结构。

这部分将对每一个对象类型和它们支持的属性加以描述。特别关注的是__doc__属性，每一个对象都要用它来保存实体文档。第 25 章将介绍怎样使用这个属性。

有关其他对象类型的信息，参见第 3 章。

23.2.1　可调用对象类型

可调用对象包括任何可以调用的对象，比如函数、方法和与特殊类相关联的方法。

1. 函数

用户自定义函数是在脚本的模块层中，使用 def 语句或 lambda 运算符创建的函数。因为它不只与类有关联，所以它在体结构上与方法完全不同。函数对象可以以任何标准结构，比如列表、字典和普通变量来存储。

表 23-3 列出了函数支持的属性。

表 23-3　函数支持的属性

属　性	说　明
__doc__或 func_doc	文档字符串
__name__或 func_name	声明的函数名字。由 lambda 定义的函数有这个名字(lambda)
func_code	对已编译字节码代码对象的引用
func_defaults	包含函数参数的默认值的元组。例如这样的定义：func(a=0,b=0)返回(0,0)
func_globals	定义全局名称空间的字典—— 就是说，这个对象可以在函数的全局名称空间用。在典型的脚本中，这意味着对脚本来说这个字典是全局的，从模块中导入函数，对模块来说这个字典是全局的

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

2. 方法(Method)

方法是只在对象实例上操作的函数。非约束方法(unbound method)对象是已经声明的，但没有与对象的实例关联的方法。类的方法属于非约束的方法。约束方法(bound method)是与类的实例连接的方法。表 23-4 列出了两种方法类型的属性。

表 23-4　方法的属性

属　性	说　明
__doc__	文档字符串
__name__	方法名
Im_class	用于定义方法的类
Im_func	实现方法的 Function 对象
Im_self	与方法关联的实例。对于未绑定的方法，这个值为 None

3. 内置函数和方法

内置函数和方法与，由 Python 语言直接定义的或在扩展模块中定义的和一般以 C/C++开发的函数和方法相关。表 23-5 列出了内置函数和方法的属性。

表 23-5　内置函数和方法的属性

属　性	说　明
__doc__	文档字符串
__name__	函数名或方法名
__self__	与方法关联的实例。对于内置函数来说，__self__值为 None。对于内置方法来说，__self__是指向激活这个方法的对象。比如，在 b = [1,2]中，b.append. __self__指向 b

23.2.2　模块

模块是通过 import 语句加载的其他对象集合的容器。如下列代码段：

```
import    foo
```

创建一个指向 foo 模块的新对象。

模块定义自己的名称空间，通过在__dict__属性里的字典来实现。在模块的名称空间内，可以调用对应的查找和/或分配方法来寻找变量或对象。例如，mymodule.list 对象如 mymodule.__dict__ ['list']一样有效。另一种方法是使用分配操作，但有隐含的开销。更多信息见第 24 章的“手动优化”部分。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

模块支持在表 23-6 中列出的属性。

表 23-6　模块对象支持的属性

属　性	说　明
__dict__	与模块名称空间相关联的字典
__doc__	模块文档字符串
__name__	模块名
__file__	模块从其中加载的文件

23.2.3　类

用 class 语句来创建类。类包含一个属性，__dict__，这个属性包含与类相关联的对象—— 包括方法在内。类的访问机制和模块的访问十分相似。但是，在访问不能通过__dict__完成的位置上，可能继续通过基类来完成查找；在定义类时所创建的基类列表中，查找按从左到右的顺序执行。要认识到，把值赋给一个属性时，更新或是必须时创建相对应的类的属性，而不是任何基类的属性。

表 23-7 列出了类所支持的属性的完整清单。

表 23-7　类支持的属性

属　性	说　明
__dict__	与类的名称空间相关联的字典
__doc__	类的文档字符串
__name__	类名
__module__	由类定义的模块名字
__bases__	包含对应于此类的基类列表的元组

23.2.4　类实例

在调用类时创建类的实例，即创建对象的一个新实例。每一个实例都有自己的名称空间，这个名称空间由字典定义。名字的查找通过检查__dict__来操作。如果没有找到名字，将继续通过__class__.__dict__来查找，然后进一步通过每一个基类的__dict__属性来查找。为类实例分配数值总是更新(或创建) __dict__中的入口。

表 23-8 列出了类实例所支持的属性。

表 23-8　类实例所支持的属性

属　性	说　明
__dict__	与类实例的名称空间相关联的字典
__class__	类的实例对象的名字

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

23.2.5　内部类型

已经学习了在 Python 下运行代码块和其他元素的内部类型的使用方法。本节主要讲述每个代码对象的具体属性。在建立一个新调试器时，或是对脚本在运行过程中的详细信息进行注解时，将要使用这些信息。详细信息请见第 24 章。

1. 代码对象

代码对象代表了任何一个可执行的元素。代码对象的所有属性都是只读的，表 23-9 列出了代码对象的所有属性。

表 23-9　代码对象的属性

属　性	说　明
co_name	函数名
co_argcount	函数定义的固定位置参数的个数
co_nlocals	函数使用的局部变量个数
co_varnames	包含局部变量名字的元组
co_code	表示原始字节码的字符串
co_consts	包含字节码使用的文字的元组
co_names	包含字节码使用的名字的元组
co_filename	在其中定义对象的文件名
co_firstlineno	在 co_filename 的文件里的代码对象的第一行
co_lnotab	包含字节码偏移的行数的字符串
co_stacksize	运行对象所需要的堆栈的大小
co_flags	整数型的解释器标志。如果对象使用有效参数列表，则设置其 2 位。如果对象接受关键字参数，则设置其 2 位

2. 帧对象

帧对象提供了运行代码对象的环境。表 23-10 列出所有帧对象的属性。

表 23-10　帧对象的属性

属　性	说　明
f_back	前一个堆栈帧(相对于调用者)。只读
f_code	本帧内运行的代码对象。只读

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

属　性	说　明
f_locals	局部变量使用的字典。只读
f_globals	全局变量使用的字典。只读
f_built-ins	内置名字使用的字典。只读
f_restricted	如果帧是在受限环境下运行，设为 1。只读
f_lineno	正在运行的代码的行数。只读
f_lasti	在运行的 f_code 里，指向最后一个字节码的索引。只读
f_trace	所调用函数的每一个源码行的起始处
f_exc_type	最新一个异常
f_exc_value	最新异常的值
f_exc_traceback	最新异常的回跟踪

3. 回跟踪(Traceback)对象

回跟踪对象保存用于判断当前的运行帧的结构信息，以及在当前帧前的所有帧。当异常发生时，回跟踪帧自动创建。表 23-11 列出了回跟踪对象支持的属性。

表 23-11　回跟踪对象支持的属性

属　性	说　明
tb_next	在堆栈跟踪里的下一个回跟踪对象(相对于发生异常的帧)
tb_frame	当前层的运行帧对象
tb_line	异常发生的行号
tb_lasti	异常发生前要执行的最后一条指令。这是那个时刻运行的代码对象的字节码的索引。等价于 frame.f_lasti

4. 切片对象和 Ellipsis 对象

当利用 slice 的扩展语法，比如 s[i:j:step]、s[a:b, c:d]或 s[..., i:j]选择了一个切片时，内部创建一个切片对象。切片对象也可以在使用内置函数 slice()时自动创建。表 23-12 列出了切片对象支持的属性。

Ellipsis 对象还用来指示省略号在切片中的存在。省略号对象没有属性，并且总是求值为真。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

表 23-12　切片对象支持的属性

属　性	说　明
start	切片的下边界
stop	切片的上边界
step	切片的跨距，如果默认，则为 None

23.3　字节码

计算机里的中央处理器(CPU)进行的运算，是根据它所接受的指令和数据来执行的，这些指令和数据是汇编码。不同的指令对应不同的操作，如 add 指令是将两个数字相加，mul 指令则是将两个数字相乘。

虽然不同的处理器支持不同的指令，但是基本的机制是相同的——无论何时运行应用程序，运行的都是这些处理器的指令。在编译 C/C++应用程序时，其中编译过程的一部分工作就是将 C/C++语句翻译成汇编码，接下来，汇编器又把汇编码翻译成为执行原来在 C/C++代码中定义的过程所需要的二进制码格式。

显然，应用程序比简单地将两个数相加复杂多了。被不同类型的处理器支持的指令数目是有限的，这些受支持的指令通常采用的是最最普遍的命名。例如，当在 C 语言中的将两个字符串相加的 strcat 函数，好像是很简单的事情。但是，这种情况下为了生成新的字符串，这个连接处理转换为很多条汇编指令，将信息从一个内存位置拷贝到另一个内存位置。一条单行语句可能要由很多的汇编指令组成。

传统的汇编器有一个问题，它非常受机器的限制。虽然可以在 486 或奔腾机上运行 386 的代码，但是并不能保证能完全有效。把信息从一种类型的处理器传送到另一种类型的处理器也是不可能的。比如，将 PC 机的二进制代码直接放在 Mac 上运行，这绝对不可能。PC 机使用的是 Intel x86 样式的处理器，而 Mac 使用的摩托罗拉 680x0 处理器或是 PowerPC 处理器。

在开发应用程序时，这种不同操作系统之间的不兼容是很头痛的事情。因为不同的处理器使用不同的汇编语言，为了得到正确的可运行的格式，应用程序需要被重新编译。当然，这还只是其中的一小部分，还需要根据运行不同的函数库和平台重新编译应用程序。因为底层硬件的环境、支持和接口完全不同，所以 Linux x86 应用程序不能在 Windows x86 平台上运行。

23.3.1　Python 字节码

Python 以一种非常类似的方法工作。本章前面的图 23-1，说明了 Python 脚本是怎样汇编成为 Python 字节码的。Python 字节码本质上和处理器的汇编语言是一样的——都有一系列在其上建立任何事物的低级指令。

这种情况下，和其他的任何应用程序一样，这些字节码指令实际就是高度优化的汇编语言。使用字节码给我们提供了许多好处：

- 因为源文本已经“编译”成字节码，而且经过了很大的优化，所以字节码最终得到灵活

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

地、快速地执行。

● 因为字节码是经过高度优化的，所以可以得到接近原始汇编器运行时间。但是，因为不得不把 Python 字节码翻译为本地处理器指令，所以也有一些时间开销。无论怎样，字节码处理程序都是优化和非常快速的。

● 因为字节码是特定于 Python 而不是特定于硬件，所以 Python 字节码可以在不同的平台之间共享(如 Mac、PC、基于 SPARC 的 Unix 系统等等)，并且仍是兼容的。

这种方式的不利之处是，在脚本运行前，Python 必须把脚本编译成为字节码。就是说，虽然运行 Python 字节码几乎和运行汇编过的应用程序一样快，但每次把原始源文本编译成为字节码也需要一定的时间开销。每一次以微秒来统计，但是，这并不意味着就可以忽略这个时间。

为了提高 Python 的执行速度，解释器试图载入模块的以前编译的版本。因为导入的是字节码，这样就没有编译开销，所花费的时间就仅仅只是载入代码和创建别名或名称空间里的挂钩的时间。

Python 并不是惟一使用解释器来编程的，Perl 的工作方式和 Python 十分相似，而 Java 的整个理论是为运行应用程序而生成跨平台兼容的字节码。

23.3.2　字节码的分解

虽然在大多数情况下，理解字节码并没有什么用，但是这样有助于您理解在 Python 应用程序运行时，后台进行了什么操作。dis 模块可以把 Python 字节码(来自于预编译代码对象)分解成为能读和能显示在屏幕上的文本格式。如果想查看 Python 内部是怎样工作的，这种方法很有用。

例如，下面的代码是将一个数字加到一个已经存在的变量中：

```
def simple(a, b):
    a += b
    return a
```

利用 dis 模块能分解利用如下设计的函数生成的字节码：

```
dis.dis(simple)
```

将输出以下的结果：

```
  0  SET_LINENO          1

  3  SET_LINENO          2
  6  LOAD_FAST           0 (a)
  9  LOAD_FAST           1 (b)
 12  INPLACE_ADD
 13  STORE_FAST          0 (a)

 16  SET_LINENO          3
 19  LOAD_FAST           0 (a)
 22  RETURN_VALUE
 23  LOAD_CONST          0 (None)
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
26   RETURN_VALUE
```

如果没有很快地得到详细信息，预输出过程显示先加载两个给定参数的字节码，然后，使用扩充的赋值(这里由 INPLACE_ADD 注解)将它们加在一起。和其他语言一样，Python 也使用堆栈来保存信息。LOAD_FAST 指令就是将两个变量放到堆栈中，接着， INPLACE_ADD 将堆栈中的这两个数加起来，再把结果放入堆栈中。然后，STORE_FAST 指令更新 a 的值。

堆栈开始是空的，因此在返回任何值前，需要把 a 的值加载回来，然后把堆栈上的值返回给调用者。最后的 LOAD_CONST/RETURN_VALUE 指令对，由调用 dis.dis 生成。

每一行的前面的数字表示指令出现的字节位置。因为每条指令占用一个字节，字节码的参数占用两个字节，所以要命名为字节码。

如果想使用脚本运行后的输出，就需要知道函数是怎么调用的。如果修改上例为如下所示：

```
def   simple(a, b):
      a += b

      sq(a)

      return   a


def   sq(a):

      a += a

      return a
```

可以看出，上例是在函数中调用了另一个函数 sq。dis 生成的输出和下面的一样：

```
0   SET_LINENO                1
3   SET_LINENO                2
6   LOAD_FAST                 0 (a)
9   LOAD_FAST                 1 (b)
12  INPLACE_ADD
13  STORE_FAST                0 (a)
16  SET_LINENO                3
19  LOAD_GLOBAL               2 (sq)
22  LOAD_FAST                 0 (a)
25  CALL_FUNCTION             1
28  POP_TOP
29  SET_LINENO                4
32  LOAD_FAST                 0 (a)
35  RETURN_VALUE
36  LOAD_CONST                0 (None)
INSID
39  RETURN_VALUE
```

字节码里的函数调用首先将 sq 函数载入堆栈，接着加载提供的参数，再调用函数。POP_TOP

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

指令将堆栈最上面的项删除(调用的函数名)。因为 sq 函数还没有返回任何值，这时堆栈还没有被无用信息占用。如果已经把 sq 的结果赋给 a，当然能访问从堆栈返回的值，就可以继续下面的操作。

23.3.3　字节码指令(Opcode)

dis 模块对多数程序来说有一定的限制值。如果想更好地了解 Python 的内部，请参见表 23-13，它列出所有的支持的操作码(即字节码指令)，这样使开发者能够跟踪脚本的执行。在入栈和出栈列中，无限定的数字表示有多少项移出或是压入堆栈中。如果数字用方括号括起来，数字表示那个压入或移出堆栈的数据项所在的序号。其中，1 表示栈顶的那一项。在描述列中，连续的字符表示移出堆栈的项，例如，a = 1(堆栈顶)，b = 2，c = 3 等。

从堆栈接受数据项而不把信息放回堆栈的操作码编辑适当位置上的变量，举例包括切片对象和序列上的删除操作。

表 23-13　Python 字节码解释器支持的操作码

字　节　码	入　　栈	出　　栈	说　　明
STOP_CODE			指示编译器代码结束
POP_TOP	1		从堆栈的顶部删除一项
ROT_TWO	[1,2]	[2,1]	交换堆栈顶端两项的位置
ROT_THREE	[1,2,3]	[2,3,1]	把第一项移动到第三项的位置上，用第二项和第三项替代第一项和第二项
ROT_FOUR	[1,2,3,4]	[2,3,4,1]	把第一项移动到第四项的位置上，用第二项、第三项和第四项替代第一项、第二项和第三项
DUP_TOP	[1]	[1,1]	复制堆栈顶端的数据项
UNARY_POSITIVE	1	1	执行 a = +a
UNARY_NEGATIVE	1	1	执行 a = -a
UNARY_NOT	1	1	执行 a = not a
UNARY_CONVERT	1	1	执行 a = 'a'
UNARY_INVERT	1	1	执行 a = ~a
BINARY_POWER	2	1	执行 a = a ** b
BINARY_MULTIPLY	2	1	执行 a = a * b
BINARY_DIVIDE	2	1	执行 a = a / b
BINARY_MODULO	2	1	执行 a = a % b

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

字 节 码	入 栈	出 栈	说 明
BINARY_ADD	2	1	执行 a = a + b
BINARY_SUBTRACT	2	1	执行 a = a - b
BINARY_SUBSCR	2	1	执行 a = a[a]
BINARY_LSHIFT	2	1	执行 a = a << b
BINARY_RSHIFT	2	1	执行 a = a >> b
BINARY_AND	2	1	执行 a = a & b
BINARY_XOR	2	1	执行 a = a ^ b
BINARY_OR	2	1	执行 a = a | a
INPLACE_POWER	2	1	在原位执行 a = a ** b
INPLACE_MULTIPLY	2	1	在原位执行 a = a * b
INPLACE_DIVIDE	2	1	在原位执行 a = a / b
INPLACE_MODULO	2	1	在原位执行 a = a % b
INPLACE_ADD	2	1	在原位执行 a = a + b
INPLACE_SUBTRACT	2	1	在原位执行 a = a - b
INPLACE_LSHIFT	2	1	在原位执行 a = a << b .
INPLACE_RSHIFT	2	1	在原位执行 a = a >> b .
INPLACE_AND	2	1	在原位执行 a = a & b .
INPLACE_XOR	2	1	在原位执行 a = a ^ b .
INPLACE_OR	2	1	在原位执行 a = a | b .
SLICE+0	1	1	执行 a = a
SLICE+1	2	1	执行 a = a[b:]
SLICE+2	2	1	执行 a = a[:b]
SLICE+3	3	1	执行 a = a[b:c]
STORE_SLICE+0	2	0	执行 a[:] = b
STORE_SLICE+1	3	0	执行 b[a:] = c

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

字 节 码	入 栈	出 栈	说 明
STORE_SLICE+ 2	3	0	执行 b[:a] = c
STORE_SLICE+ 3	4	0	执行 c[b:a] = d
DELETE_SLICE+ 0	1	0	执行 del a[:]
DELETE_SLICE+ 1	2	0	执行 del b[a:]
DELETE_SLICE+ 2	2	0	执行 del b[:a]
DELETE_SLICE+ 3	3	0	执行 del c[b:a]
STORE_SUBSCR	3	1	执行 b[a] = c
DELETE_SUBSCR	2	0	执行 b[a]
PRINT_EXPR	1	0	求 a 的值，并打印出来
PRINT_ITEM	1	0	求 a 的值，并打印到 sys.stdout 指向的文件对象中
PRINT_ITEM_TO	2	0	求 b 的值，并打印到文件对象 a 中(即 print file expr)
PRINT_NEWLINE	0	0	除非语句终止于逗号，否则打印由所有 print 语句生成的新行
PRINT_NEWLINE_TO	1	0	除非语句终止于逗号，否则把所有 print 语句生成的新行打印到文件对象 a 中
BREAK_LOOP	0	0	执行 break
LOAD_LOCALS	0	1	把一个局部定义的引用压入到堆栈的当前范围中
RETURN_VALUE	0	1	执行 return
IMPORT_STAR	1	0	执行 from a import*
EXEC_STMT	3	0	执行 exec c,b,a
POP_BLOCK	1	0	从块堆栈中删除一个块
END_FINALLY	0	0	中止 try 语句里的一个 finally 子句
BUILD_CLASS	3	0	执行 class c(b)，其中，a 是所支持方法的一个字典

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

字　节　码	入　　栈	出　　栈	说　　明
STORE_NAME namei	1	0	创建新的局部变量。执行 name =a ，其中，namei 是代码对象的 co_names 属性的 name 的索引
DELETE_NAME namei	0	0	执行 del name。其中，namei 是代码对象的 co_name 的索引
UNPACK_SEQUENCE count	1	count	以相反的顺序(从右到左),将 a 的值拆分别堆栈里的 count
DUP_TOPX count	count	count * 2	复制堆栈里的 count 项
STORE_ATTR namei	2	1	把值存储入一个属性中。执行 a.name = b，其中，namei 是代码对象的 co_ name 的索引
DELETE_ATTR namei	1	0	删除属性。执行 del a.name,其中，namei 是代码对象的 co_name 的索引
STORE_GLOBAL namei	1	0	创建一个新的全局变量
DELETE_GLOBAL namei	0	0	删除一个全局变量
LOAD_CONST consti	0	1	把常量压入堆栈，其中，consti 是代码对象的 co_const 属性中的常量的索引
LOAD_NAME namei	0	1	把变量 a 的值压入堆栈，其中，namei 是代码对象的 co_names 属性中的 name 的索引
BUILT_TUPLE count	count	1	创建一个元组，此元组包含了堆栈中 count 个数据项，再把此元组对象压入堆栈
BUILT_LIST count	count	1	创建一个列表，此列表包含了堆栈中 count 个数据项，再把此列表对象压入堆栈
BUILD_MAP zero	0	1	将一个空字典压入堆栈，忽略参数 zero
LOAD_ATTR namei	1	1	执行 a = getattr(a, co_names[namei])
COMPARE_OP opname	2	1	执行 a cmp_op[opname] b
IMPORT_NAME namei	0	1	导入 co_names[namei]模块，并将此模块对象压入堆栈

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

（续表）

字　节　码	入　栈	出　栈	说　明
IMPORT_FROM namei	1	1	从模块 i 中载入 co_names[namei]属性。结果对象被压入栈顶
JUMP_FORWARD delta	0	0	递增字节码计数器 delta
JUMP_IF_TRUE delta	1	1	如果 a 为真，增加字节码计数器 delta 字节。a 不从堆栈中删除
JUMP_IF_FALSE delta	1	1	如果 a 为假，增加字节码计数器 delta 字节。a 不从堆栈中删除
JUMP_ABSOLUTE target	0	0	设置字节码计数器为 target
FOR_LOOP delta	2	0(3)	执行一个 for 循环。其中，a 是序列 b 的当前索引。元素 b[a]被计算；如果序列取尽，字节码计数器加 1，否则序列和递增计数器和当前索引被压入堆栈
LOAD_GLOBAL namei	0	1	将名字为 co_names[namei]的全局变量压入堆栈
SETUP_LOOP delta	0	1	将一个循环块压到堆栈块中。块的范围是当前字节码指令向上 delta 个字节
SETUP_EXCEPT delta	0	1	将 try-except 子句压入块堆栈中，其中，delta 指向第一个 except 语句
SETUP_FINALLY delta	0	1	将 try-except 子句的 try 块压入块堆栈。其中，delta 指向 finally 语句
LOAD_FAST var_num	0	1	把对局部变量 co_varnames[var_num]的一个引用压入堆栈
STORE_FAST var_num	1	0	将 a 存入局部变量 co_varnames[var_num]
DELETE_FAST var_num	0	0	删除 co_varnames[var_num]
LOAD_CLOSURE i	0	1	将一个引用放到网格第 i 栏所含单元格里，并释放变量存储空间。如果 i 小于 co_cellvars 的长度，那么变量的名字在 co_cellvars[i]里，否则，在 co_freevars[i-leng(co_cellvars)里

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

字　节　码	入　　栈	出　　栈	说　　明
LOAD_DEREF i	0	1	将一个对象引用放到网格第 i 栏所含单元格里，并释放在堆栈上的变量存储空间
STORE_DEREF i	1	0	将 a 存储到网格第 i 栏所含单元格里，并释放变量存储空间
SET_LINENO lineno	0	0	将当前行号设为 lineno。当报告错误时，被异常和帧使用来决定当前的位置
RAISE_VARARGS argc	0 to 3	0	执行 raise 操作，其中，arc 是 raise 语句的参数个数。taceback 在 c 里能找到，参数在 b 里能找到，异常的名字在 a 中能找到
CALL_FUNCTION argc	argc	0	调用一个函数。argc 参数指定了位置参数(低位字节)和 keyword 参数(高位字节)的个数。参数从堆栈中删除。堆栈中最低的项应当是要调用的函数
MAKE_FUNCTION argc	1-argc	1	将函数对象 a 压入堆栈中。函数的声明中有参数 argc，argc 参数在堆栈的低位值中定义
MAKE_CLOSURE argc	1- argc	1	创建一个新的函数对象，其闭合开关，并将最终对象压入堆栈，a 是与此函数相关联的代码，argc 是使用默认参数的函数。堆栈里的后缀项是在函数里使用的变量所使用的单元
BUILD_SLICE argc	2-3	1	将切片对象压入堆栈。如果 argc 是 2，将 slice(b, a)放入堆栈。 如果 argc 是 3，将 slice(c, b, a)压入堆栈
EXTENDED_ARG_EXT	0	0	前置参数远远大于默认的两个字节的操作码
CALL_FUNCTION_VAR argc	?	?	除了在堆栈顶部的参数是变量参数元组，并且跟着关键字和位置参数外，与使用 CALL_FUNCTION 调用函数方法一样

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

字　节　码	入　　栈	出　　栈	说　　明
CALL_FUNCTION_KW argc	?	?	除了在堆栈顶部的元素是关键字参数字典，并且跟着关键字和位置参数外，与使用 CALL_FUNCTION 调用函数方法一样
CALL_FUNCTION_VAR_KW argc	?	?	除了在堆栈顶部的元素是关键字参数字典，并且跟着变量参数元组，元组后面又跟着关键字和位置参数外，与使用 CALL_FUNCTION 调用函数方法一样

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第24章　调试和调整

任何程序员都知道，编写程序并使之执行只是完成任务的一半。编写完成后，需要确认程序是按要求运行的，除了设计目的是这样之外，在运行过程中没有影响或扰乱计算机其他部分的操作。

调试是程序员开发过程当中最头痛的阶段。在理想情况下，查找和解决代码中的问题应占整个开发时间的 80%。但很少有人能在测试方面花如此多的时间，大多数人都没有足够的耐心。

但是，了解调试的需求和实际跟踪与记录 bug 是完全不同的两回事。所有的调试工作都需要深入地了解代码，而且有可能要对原代码逐行阅读，一定还有更简单的方法调试程序。

在实际开始前，应该先给 bug 做一个定义。bug 是指应用程序中的可能出现问题的位置，它能导致应用程序不稳定，不能完成预期任务，以其他方式影响或破坏运行。另外还有一种 bug：运行 bug，它不太好接受，但同样很重要。运行 bug 在没有明显原因的情况下使程序执行变慢，这虽不是致命错误，但它表明程序脚本中可能存在问题。

本章开始首先介绍了可能出现的不同 bug 类型，以及如何找出并隐含地改正 bug，然后介绍了如何通过改进设计和开发应用程序时所采用的方法，来避免 bug 的产生。

接下来，本章将介绍 Python 调试器，它是执行和监控程序脚本运行的交互式接口。Python 调试器有助于隔离运行问题和资源汇集，以及 Python 配置器(profiler)，它用于记录不同脚本组件的执行次数。

24.1　调试简介

在了解程序调试方法前，需要考虑软件中可能会出现哪些类型的错误。了解错误的类型并能识别它们，有助于选择正确的解决方法。

24.1.1　bug 类型

将程序的开发过程设想为分层的树形结构。在顶端是输入的值，底端是程序输出的值或输出的结果。遗憾的是，应用程序的开发经常是从底端开始向上进行，而不是从顶端开始向下进行。开发者使用自顶向下(top-down)的方式，就必须考虑一直到得到结果的所有步骤，而使用自底向上(bottom-up)的方法，可以尝试不同的方法，直到得到可以从顶端到底端运行得到的结果。

自底向上方法开发比较慢，但是从程序员的观点看更为实用，因为它有助于产生思路，在头脑中构思方法和过程，最后反映在最终的应用程序上。

选择不同的开发方法会对程序中出现的 bug 有一定影响。使用自底向上方法，在开发一个新的组件前，不能考虑到所有的因素，因此，相对于在开发程序时使用更注重实效的自顶向下

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

法，容易产生更多的错误。

忽略 bug 所产生的影响或者对 bug 及其影响分类不正确，是开发人员常犯的错误。虽然通常看这不算大问题—— bug 还是 bug—— 但它影响到解决 bug 问题的方法(这可能导致对其他 bug 的“撞击”性影响)，并且还可能导致其他 bug 完全被忽略。

就作者个人而言，在编程的时候，一般识别以下 3 种类型的 bug：

(1) 编排(typographical)bug

(2) 逻辑(logic)bug

(3) 运行(execution)bug

1. 编排 bug

编排 bug 是指编排过程中出现的错误，无论是录入错误或是因一时忘记语法造成的错误。在 Python 中，可能使用了错误的变量名，在计算中用&代替了*，或者调用 dispersal 时错用了 disposal。后两个错误虽然只是简单的录入失误，但对应用程序的行为来说，却能产生很大差别。

在脚本运行时，Python 可以捕捉到部分编排错误。使用错误的函数或方法，会产生 NameError 或其他异常。但混用&和*的错误会有些麻烦，Python 认为它们都是合法的，实际上对于某些值它们甚至可能得到同样的结果。

其他的编排 bug 就更不明显。例如去年在一个 Web 项目里，登录过程突然停止工作。登录系统的工作过程是先根据数据库检查用户名和密码，如果两项都通过，脚本会生成一个惟一的会话 ID，并且还把它存入数据库，以供 CGI 脚本使用来鉴别用户会话。问题是虽然允许登录并生成会话 ID，但有时用户无法利用新的会话 ID 进行连接。

解决这个问题花费了一点时间，不过通过对比返回给用户的会话 ID 和数据库中的会话 ID，可以很明显看到会话 ID 的长度是随机的，有时是 26 位，有时是 27 位，而数据库中会话 ID 字段的域宽是 26 位。因此会话 ID 系统 50%的时候工作正常，另 50%的时候会话 ID 超出字段的域宽，产生错误。这是另一种编排 bug：在创建数据库脚本时，输入了错误的字段域宽。

键入问题(typos)，除非它的错误会导致解释程序失效，否则很难跟踪。唯一找到它们的方法是通读每一行代码。如果键入问题导致了计算或操作的失败，至少可以找到一个跟踪的起始点，另外调试器可以帮助开发人员跟踪问题出现的准确位置。

2. 逻辑 bug

逻辑 bug 是指在程序的逻辑流程里含有错误。和编排 bug 一样，它很明显，比如，识别函数的一个可能返回值的失败，或错误地指定一个测试或其他操作。这不同于编排错误——因为程序员当时以为自己做的是正确操作。

还以前面提到过的问题为例，在登录代码做了一些优化以后，登录系统忽然开始返回错误登录。登录函数在出现错误时返回负数，登录或会话到期时返回零，成功时返回正值。对这个 bug 做了几小时的跟踪，终于发现发生问题的原因在于 if 语句把无应答的测试返回值当作是失败的，其他任何应答值都被解释为有效的。

反过来说，异常也能是产生逻辑 bug 的位置。比如捕捉到很多异常，但对异常处理却不够具体，或者只是简单地捕捉所有的异常都可能导致逻辑 bug。捕捉到没有预料到的异常可能会产生问题。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

有多种方法可以处理逻辑 bug。程序员很好地理解 Python，了解它如何工作，知道在语言中存在什么陷阱，这是最明显的解决方法。如果对语言很了解，就不会把逻辑 bug 带进核心语言。但是，这并不能把问题从自己的代码中隔离开来。下面粗略列出几个值得双倍注意的方面：

- 访问序列，尤其是对切片对象和连接的访问次序。
- 混合元组和列表。元组是不可变的，许多函数返回的都是元组而不是列表。
- 循环，尤其是处理 for、range()和 xrange()的时候。
- 函数，特别是函数的返回值。创建一个系统，以便在整个应用中保留它们。
- 异常，确信捕捉到想要识别的异常，异常的处理程序是按顺序执行的，符合第 7 章所示的异常树。首先处理最低级的异常，然后是组一级和通用(generic)异常。
- 跟踪逻辑 bug 的方法还有提取——将程序的组件转换成函数，模块和类帮助。如果能对一个独立的单元进行优化和调试，整个应用作为一个整体就可以不用改动了。提取还可以使代码更具有可移植性，更易于管理。

3. 运行 bug

运行 bug 最难发现，很多程序员可能根本不去考虑它们。运行 bug 影响程序的运行过程，不是因为键入问题或者逻辑问题，而是因为应用程序在设计方法上允许一系列致使应用程序执行过程变慢或者停止的事件。

运行 bug 致使应用程序执行变慢，但它们实际上可能不产生错误结果，也没有以任何其他形式表现出来。例如假设一个从数据库中提取记录的应用程序。典型的查找返回 200 条记录，脚本运行需几秒钟，达到了从数据库中提取信息的时间限制。但是当一种查找返回 1000 条记录时，脚本差不多要运行一分钟。

这不是逻辑问题—— 脚本是按照要求执行的；这也不是编排错误。程序确实显示了数据库里的 1000 条记录，而且信息格式也是正确的。那么为什么脚本运行了这么长时间？

这是一个运行问题—— 或者程序员需要创建一个限制器来减少因访问和显示大量记录造成的影响，或者应用程序的一部分花费了过多的时间来分析信息。这就是运行 bug，应用程序并没有错，只是没运行在所希望的层上。

运行 bug 很难跟踪，但发现并解决这些问题对程序很有益。清除了运行 bug，应用程序将比通常应用程序具有更好的健壮性，使用时也更加稳定。因为运行 bug 常常影响程序的性能，排除它们可以优化应用程序。

执行程序是发现运行 bug 的最简单方法，但隔离出现 bug 的位置需要一些工具和技巧。Python 提供的配置器，可以运行程序并监测程序不同部分的执行次数。详见本章后面的“优化 Python 应用程序”部分。配置器并不能解决问题，它只是帮助寻找问题所在位置的工具。而且，配置器提供的是单个函数级的信息，这些信息不一定是有用的，因为常常是整个算法和一系列函数调用导致问题产生。

24.1.2　基本调试原则

下面是解决问题的一些一般原则，它们适用于任何 Python 脚本，稍后学习 Python 中捕获并解决问题的方法和技巧。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

● 检查明显的错误：脚本出现问题时，不要寻找最复杂的原因，从简单的事情入手。如果计算出错，很可能是因为变量拼写错误或者遗漏了数值或运算符。在确认了不是简单的拼写错误以后，再在外部模块、函数或第三方扩展函数库里查找问题。

● 由外到内检查：发现问题后，在最初报告问题的地方开始，从最外层，最高层调用点逐步向里开始跟踪。因为表明问题的链接是从内部到外部，所以 Python 的异常是很有帮助的。但它们只能报告识别出来的错误，而不是 bug。异常一般不能指出问题的起源。在调用函数时出现的问题大多都与最初向函数提供的数据有关，而不是函数在过程中产生的。

● 在处理编排错误时从顶部开始：当 Python 报告错误时，尤其是有关括号，引号和其他成对使用的组件不匹配的错误，问题中发现错误的地方不一定是真正出错的位置。只有不该出现的地方出现了引号或括号，Python 才能识别其为匹配错误。

● 跟随异常处理程序树：当出现异常时，跟踪记录是寻找问题来源的最有效的工具。对于“从外到内检查”的原则来说，异常是一种特例。它可以反映问题的根源，程序员可以直接找到并改正错误。但也有一个警告：不要盲目地跟随异常或者期望异常能够给出精确的信息。

● 检查异常处理程序：很多时候 Python 的程序没有工作，也没有报告错误，都是因为程序员禁用(使用 pass)了某个异常，而“错误总会出现”。异常是将应用程序中有问题的地方高亮显示，不管它们似乎多么简单或者微不足道，忽略它们都很容易发生问题。特别是要注意跟踪那些解释程序正常捕获的异常。尤其是 SyntaxError 和 TypeError 异常，只有在处理用户提供的 Python 代码时才能捕获到。跟踪和/或忽略程序中这些异常可能会导致忽略了脚本中本应改正的错误。

24.1.3　预防 bug

显而易见，避免 bug 的发生是解决问题的最好方法，预防问题总是好于解决问题。难点在于预防产生 bug 有时比解决它们更加耗时，而且看起来影响了开发的进度。实际上，防止 bug 的发生比事后解决它们更有利于在最后期限前完成开发，因为程序写完后，在寻找和解决问题方面可以节省很多时间。

但是怎样在最初就避免 bug 的发生呢？因为依赖于实现设计目的的基本方法，而不是 Python 的具体特性，很多方法已经超出了本章的讨论范围。不过还是值得探讨一下一些可应用在任何设计工作当中的基本技巧。

1. 程序设计

开发应用程序时，至关重要的是要准确了解要实现的目的。对如何实现这一目的，有一个较好的设计思想，有利于确保软件不产生 bug。即使只是简单地写下函数及其功能的清单，也要优于编程时逐步整理。

建立程序设计的一个体系或风格并坚持采用这个风格也是大有益处的。比如使用标准参数序列来交换数据是容许的，但是对于大量的参数，尤其是如果其中还有可选参数的情况，容易引起混淆。幸运的是，Python 提供了一套灵活的参数接受系统，使用默认值，或者通过参数及关键字来提供参数，这种方式胜于使用固定列表顺序。即使如此，也应尽量坚持只使用一种方式或两种方式的组合，以使代码更容易使用。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

另外一个因素是设计和开发为从开始到结束这个顺序的第 n 层程序，最终得到这样的设计手册：运行多于所编写代码的行数。这一层的程序设计已找到了其用武之地——核电站软件和建立在现代飞机当中的有线航空(fly-by-wire)计算机都使用了这项技术。另一方面，开发用于转换数据文件格式的快速脚本的文档是完全没有必要的。

2. 编辑器

好的编辑器在任何程序设计工具中都是一个重要组成部分。大多数现代的编辑器都可以匹配括号、缩排代码以使程序更具可读性。Emacs 可能是最流行的程序设计编辑器，它的自动功能和为脚本提供的选项也同样是 Python 的基础。更高级的编辑器，如 BBEdit 、Pepper(用于 Mac Os)、EditPlus(用于 Windows)和 Emacs(用于 Unix)还可以根据代码独立组成给代码标记上颜色。

使用具有简单检查功能的编辑器可以减少很多简单的编排错误，从而减少程序 bug，避免编译失败。它还可以给问题识别以直观提示。

3. 格式编排

选择一个格式标准，并按照执行。这样程序员所写的代码整洁易读，一般来讲阅读时不会太费力。Python 在这方面是有所帮助的，它提供一种整洁的式样，通过少使用括号或在适当时候的缩进，使应用程序的各部分变得容易识别。

无论如何，一定要按样式去做，尤其是最好避免把 if 和 for 语句放在同一行上。如下所示：

```
if debug: print "Just about to try something..."
```

这样的语句与测试或 for 语句放在一行，因为这样整个语句都变得不清晰，所以一定避免这样情况的出现。因为在这样一个单行语句内，解释程序没有缩进关键字，想要在 if 程序块里添加语句会遇到问题。

因为好的编辑器可以为程序员处理缩进和其他格式化单元，所以使用好的编辑器对程序开发很有帮助，但好的编辑器仍有一些定制内容。大致上，应考虑以下基本原则：

- 每一个级别缩进四列。
- 在大多数运算符两边留有空格。
- 不同功能的文本块之间用空行隔开。
- 在函数名和其后的开括号之间不使用空格。
- 在每个逗号后使用空格。
- 较长的行要在运算符(and 和 or 除外)后的位置上断开。
- 纵向调整对应的项(多行的赋值和函数参数)。
- 在不影响明确表达性的情况下，省略多余的标点。

4. 注释

给代码加注释是掌握应用程序运行过程的最好方法，这样可以指出哪里可能有潜在的问题。假设同一个小程序段，在没有注释的情况下试图去调试，程序员就无法确切地了解那些为达到

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

要求的效果而使用的技术。虽然像跟踪 bug 系统和代码修订系统工程这样的工具会有所帮助，但它们不能提供所有想要知道的内容。

如果脚本没有正常工作，但它含有关于函数或者程序段的工作方式的简短说明，就可以根据这些在调试代码时跟踪执行。

本书第 25 章详细地讲述了文档和注释的创建及使用，但在这里还是有必要列出有关在脚本中加注释的一些基本注意事项：

- 不要只是重复代码行的内容——在含有 **for** 关键词的行里再讲“for 循环”是没有意义的，应“迭代源文件清单”，或者对正执行的任务加以解释。
- 编写输入变量的类型和函数需要什么信息的清单。
- 指定返回值以及什么情况下会返回这些值。
- 编写有关错误 / 失败的情况及其返回值的文档。
- 编写有关实施决策的文档，比如选择某个算法，而不选择其他算法的原因。
- 对于所有编写的“不易理解”代码都编写文档。有时需要处理与应用程序交互的其他程序或函数库的 bug，对这些做个记录是个好主意。

5. 跟踪 bug

找到 bug 是一件事，而记住 bug 所在的位置则是另外一件麻烦事。对已经发生的 bug 以及是否已经修正起码要做个文本记录。一定要确保包含 bug 是什么以及对 bug 怎样进行修正的这些信息，原因是做这样的记录有助于识别和跟踪后面的 bug。

发现 bug 时，应记录以下信息：

- 位置——文件号和行号，如果已知，或者起码是可能的函数故障所在。
- 所发生的 bug 说明，包括预期的结果和实际出现的结果。
- 发现 bug 的时间和日期，以及报告 bug 的用户名。
- 其他相关信息，比如平台、环境变量和其他任何可能导致 bug 的因素。

如果想使跟踪 bug 更加专业化，可以考虑使用类似 BugZilla 之类的跟踪软件，BugZilla 是 Mozilla 工程的一部分。实际上，BugZilla 系统是用 Perl 编写的，它是底层 MySQL 数据库的有效 Web 界面。BugZilla 可以从 http:// bugzilla. mozilla. Org.网址下载。

请记住，注释是为了帮助记忆，并向其他程序员指出执行顺序，而不是一种语言上的解说或者编程时遇到挫折后的发泄。实际上，在这最后一点上，如果利用这个机会，记录下最后选择某种方法的原因，那有时注释很有帮助。注释还是在深夜里处理问题的一个乐趣。

6. 使用代码修订

修订代码系统(Revision Code System ，RCS)和并发版本系统(Concurrent Versioning System ，CVS)，提供了用于跟踪不同版本代码之间差异的方法。最普通的操作方式是编写并运行一段代码，但有可能不必调试，然后将那个版本的代码登记到代码修订系统中。改变代码时，登记代码的版本修订序号。代码修订系统跟踪从一个版本到另一个版本的变化。如果修改代码失败，不必将代码中所有错误的修正再改回去，用前一个版本覆盖掉现在代码就可以了。

RCS(用作 CVS 的基础)允许存储每次修订的注释，并自动跟踪已登记文件的版本号。利用这点可以做一个状态报告，包括新的修订要达到的目的，对已定位的 bug 或问题给出指示等。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

从最终用户处接收到 bug 报告以及跟踪 bug 时，版本号会起到一定作用，那些用户报告的 bug 可能在新的修订中已经被修正了。

CVS 和 RCS 的区别在于同一时刻，RCS 只允许一个用户对某一独立的源文件进行编辑，而 CVS 允许许多程序员并行开发。另外 CVS 还有其他特性，比如它能够把树导出到因特网上，能够自动创建基于源文件当前版本号之上的包，即使这些文件仍在开发之中，也能创建这样的包。CVS 具有良好的接口，包括许多 GUI 客户程序和 Web 工具。

如果只是个程序员，RCS 就可以满足需求。从 http:// www. gnu. org/ software/ rcs/ rcs. html 网站可以得到更多的详细信息，并能下载到 RCS。如果需要 CVS，可以查看 http:// www. loria. fr/~ molli/ cvs- index. html 的 CVS Bubble 页。

24.2　调试技术

大多数人提到“调试”时，会想到调试器，并试图找到在脚本运行期间可以交互监控的方法。如果调试器很好用，对 Python 来说其中的很多功能都是用来查找 bug 的极好方法，要达到同样的结果有更简单的方法。

所有的程序终究都包括许多对变量和信息的更改。调试一段代码时，主要关注正在使用的变量的值、这些值在哪里改变，以及在特定位置变量是什么值。所有的调试器都提供几种变量监测机制，可以在程序运行时监控变量的内容。不过，要达到这样的监测一般还有更简单的方法。

小小的 print 语句就经常被忽略掉。但要显示脚本中的变量值，print 语句却是最快最简单的方法。只要知道怎样使用 print 语句，程序员就能根据要打印的值设计输出。而且对于在哪里能生成要监视的值，拥有极大的灵活性。

print 语句虽然不能解决所有问题，但如果应用程序是基于文本方式并且是交互式的，print 调试语句可以很好地中断程序流程和应用程序的接口。如果是 Tk 或者 GUI 的其他应用程序，则看不到输出。对于 CGI 脚本，输出信息可能会与脚本正常生成的 HTML 混淆。在所有这些情况下，较好的选择是把调试输出发送到一个单独的文件里，稍后再监控此输出。另外也可以写入诸如 syslog(更多信息参见 syslog 模块)的中心日志里。

对简短脚本或模块的调试，解释程序的交互式模式可能更简单好用。甚至可以把解释程序配置成为：脚本正常执行一结束就切换到交互模式。

当然，这里不是有意推荐这些技巧来替代调试器，但拥有一点点可选方法还是值得的，还要记住，调试器需要一个交互式的界面。如果要跟踪脚本的执行，而又无法交互式访问解释程序，或者想生成用于后处理的运行日志，后处理的运行日志可能由用户去做，而通过调试器将无法实现。

本节讨论不同调试技巧细节，包括 pdb 模块提供的 Python 调试器。

24.2.1　利用 print

print 语句是最简单最容易的脚本调试方法。因为可以快速地往脚本里添加 print 语句并监控输出，所以，使用 print 的情况甚至可能多于使用真正的调试器，尤其是在 Unix 或 Linux 下

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

编写程序时更是如此。因此，作者喜欢的环境是使用 Emacs 和命令行的组合，Emacs 用于编辑，命令行用于测试。

但是利用 print 不只是在脚本里为每一个想要输出的变量加单行语句，print 提供了很多有利的灵活性。所创建输出的格式是很重要的，只有这样才能判断在哪里，在什么时候出现了错误。

pring 语句还可以成为报告脚本最后版本中调试信息的一种方法。通常要和全局变量结合在一起来使用这种方法，可能要借助于脚本的命令行来设置启用或禁用一些简单调试。脚本以这种方式输出调试信息的好处在于，无论是用户还是程序员都可以事后检查应用程序的调试，而不需要使用交互式调试器。

print 语句唯一不常起作用的地方是在循环中，因为这样，在运行结束后会产生大量需要手工处理的信息。有时如果要在程序过程中持续不断地监控一单一变量，或者要监控对文件句柄的输入输出，循环机制会有一定作用。

print 的使用方法很简单，只需要在想输出数据的地方插入常规的 print 语句即可。如下所示：

```
for x in range( 100):
        # do some stuff
        print 'x is %d and result is %f' % (x, result)
```

这样就可以看到运行期间数值和计算的变化了。然而，这个例子最终生成的结果没有什么特别的用处。使用 print 命令来注解脚本的运行，并确保生成的信息在事件结束后有用。

1. 使用调试变量

如果想使用一个调试变量来激活这些消息的生成，只需要在 print 语句前加一个 if 语句。如下所示：

```
if debug:
    print "Value of the data after calculation is", data
```

不管怎样，只有在一种情况下这些信息才有用，那就是有简单的方法可以把 print 语句与脚本中的一个点连接起来。一般的习惯是在脚本中的许多地方剪切和粘贴 print 语句,可是除非更进一步限定输出，否则这样做只会减少 print 语句的用处。

在输出什么样的信息，为什么输出这些信息等方面，要确保包含一些有用指导。在开发某一部分脚本的时候，这个用处可能是很明显的，但对于在几个月或几年以后阅读它们的人来说，就不清楚了。

2. 引用信息

在输出中加入数据很简单——只需要在合并信息时加入变量或使用格式化运算符%。不管怎样，没有附加操作，在检查输出的时候这些语句就不会包含所有有用的信息。

一般情况，遇到的问题都与用到的内容有直接关系。比如一件很简单的事情，出现了通常并不想见的前导或后缀的空格，就是因为在输出中隐藏了对脚本产生影响的“不可见”字符。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

解决这个问题的方法就是引用您输出的信息，以便跟踪正在监控的变量的开始取值和最后取值。例如，不使用如下命令：

```
print "The name returned was %s" % (name,)
```

而使用如下命令：

```
print "The name returned was [% s]" % (name,)
```

将数据放在方括号内，可以很容易识别出隐藏的字符。

提示:

在 Web 脚本中，不要将输出放在尖括号(< >)内。因为浏览器会将其识别为未知的 HTML 标志符，所以将终止隐藏的输出。

引用输出并不能解决所有问题，仍然有隐藏的字符可能扰乱脚本的执行。尤其是序号较小的那些字符，它们包括如换行、回车、纵向制表符、横向制表符等控制字符，在输出中仍是不可见的。

使用 mapascii 函数可以避免这种情况，这个函数对控制字符进行转换，使它们具有可读性。实际上，可以向这个函数加入很多字符，如映射 CGI 脚本中的尖括号使之可读性更强。但最终的结果是一样的——包含控制字符的一个可见版本的字符串。如下所示：

```
def mapascii( string):
    retval = ''
    charmap = {
        '000' : '[ NUL] ',
        '001' : '[ SOH] ',
        '002' : '[ STX] ',
        '003' : '[ ETX] ',
        '004' : '[ EOT] ',
        '005' : '[ ENQ] ',
        '006' : '[ ACK] ',
        '007' : '\ a',
        '010' : '\ b',
        '011' : '\ t',
        '012' : '\ n',
        '013' : '\ v',
        '014' : '\ f',
        '015' : '\ r',
        '016' : '[ SO] ',
        '017' : '[ SI] ',
        '020' : '[ DCE] ',
        '021' : '[ DC1] ',
        '022' : '[ DC2] ',
        '023' : '[ DC3] ',
        '024' : '[ DC4] ',
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
        '025' : '[ NAK] ',
        '026' : '[ SYN] ',
        '027' : '[ ETB] ',
        '030' : '[ CAN] ',
        '031' : '[ EM] ',
        '032' : '[ SUB] ',
        '033' : '[ ESC] ',
        '034' : '[ FS] ',
        '035' : '[ GS] ',
        '036' : '[ RS] ',
        '037' : '[ US] ',
        '040' : '[ SP] ',
        }
    for char in string:
        if (charmap. has_ key( char)):
            retval += charmap[ char]
        else:
            retval += char
    return retval
```

要使用 mapascii，只需要给出字符串和函数，函数就会将其中的控制字符转换成可见的字符串，其他字符如正常情况下被附加到返回的字符串上。例如：

```
print mapascii( 'rn011004032027')
```

转换成为如下所示的字符：

```
\r\n\t[EOT][SUB][ETB]
```

3. 其他细节

有一个与 print 使用技巧相关的问题，当程序员回到脚本中寻找问题的位置时，依赖的是消息中包含的内容，怎样才能更容易地在输出中得到文件和行数等信息呢？

有一种方法可以使用，但要注意，除非是有经验的 Python 程序员，否则不建议使用这些函数。其实，这些函数一直存在，并且具有文档，有系统支持的特征。为使 print 语句更适用，需要两个信息：当前行所包括文件的文件名和当前行在该文件中的行数。

Sys.getframe() 返回的对象是当前执行的帧(frame)。如果给定一个数字，sys._getframe 可以返回以前执行的第 n 层帧。返回帧的 f_lineno 属性返回生成帧的模块的行数。inspect 模块提供了一系列关于对象的函数，其中包括 getfile()，这个函数返回定义对象的文件。利用上述信息，创建了 print_status()函数，该函数可以编写文件名、行号以及您调用的消息。如下所示：

```
def print_ status( data):
    import inspect, sys
    frame = sys._getframe(1)
    lineno = frame.f_lineno
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
        filename = inspect.getfile( frame)
        print "Status (% s:% s): %s" % (filename, str(lineno), str(data))
```

实际上调用 print_status()函数，如下所示：

```
print_status(" Nothing doin' here!")
```

很显然，只有 print_status()调用自己时，打印的有关文件和行号的信息才有效，但是要从出现问题的位置开始，这样就足够了。

24.2.2　保存日志

在脚本正运行，无法对其交互式访问时，可以写入外部日志文件并记录在脚本中的运行记录。比如对于基于 CGI 和 GUI 的脚本，就无法进行交互式访问。

在这种情况下最好的解决方法就是在脚本运行期间，将调试信息写入文件，备以后查看。最简单的方法是修改前一节提到的 print_status 函数，将信息发送到文件，而不是发送到标准输出中。修改的 print_status 函数如下：

```
def print_status_log(data):
        import inspect, sys
        frame = sys._getframe(1)
        lineno = frame.f_lineno
        filename = inspect.getfile(frame)
        try:
        logfile = open('debug. log', 'w+ ')
       except:
            print "Fatal, couldn't open the debug log"
        logfile. write("Status (% s:% s): %s"
                        % (filename, str(lineno), str(data)))
        logfile. close()
```

这不是写入日志的最有效方法。毕竟，每次调用这个函数时都要打开文件，但可以这样调试。在最后一版脚本中，将函数一起删除，或者至少利用一个变量限定函数调用来启用调试输出。这样只有在调试开关打开时，才会影响执行速度。如要真正有技巧地实现保存日志，就要创建一个用来控制打开日志文件的类，具备关于写入和更新日志的类实例的方法。

24.2.3　交互使用 Python

通过 Python 的交互式界面可以简单、快捷地测试和演示 Python 的部件。在扩展模块里也可以使用同样的界面测试语句和函数。只要激活 Python，就可以使用交互式界面。如下所示：

```
$ python
Python 2.1 (# 2, Apr 29 2001, 14: 36: 04)
[GCC 2.95.3 20010315 (release)] on sunos5
Type "copyright", "credits" or "license" for more information.
>>>
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

交互式界面的另一个好处在于执行脚本后，可以立即激活界面。在执行一系列自动操作之后继续手工处理时，这一点就很有用处。实现方式是在命令行中指定-i 参数。如下所示：

```
$ python -i myscript.py
        Some or other output...
>>>
```

一出现>>>提示符，就可以开始键入命令并继续手工处理脚本了。

24.2.4　使用 Python 调试器

作为标准配置的一部分，pdb 模块提供对 Python 脚本在源代码层调试的交互界面。可以通过调试器执行任何 Python 脚本，并立即进入调试器，以便单步执行源代码。也可以设置断点，当在脚本正常执行到某一特定行时，或者符合某一特定条件时，脚本代码进入调试器。

调用调试器有 3 种方法：通过命令行，交互式方法，或在出现异常后的交互式方法。使用命令行是最简单的方法。如下所示：

```
$ python /usr/ local/ lib/ python2.1/ pdb.py myscript. py
> /export/home/etc/mcslp/books/pytcr/ch24/<string>(0)?()
 (Pdb)
```

这样在脚本实际运行前，先进入调试器。注意必须特别指定 pdb.py 模块的位置，因为 Python 解释程序并不知道要用 pdb 调试文件。

交互式方法指在导入和执行脚本前手工导入 pdb 模块。如下所示：

```
>>> import pdb
>>> import mymodule
>>> pdb. run('mymodule. main()')
> <string>(0)?()
 (Pdb)
```

请注意，使用这种方法必须先导入自己的模块，然后再运行函数进行测试。如果导入的模块是立即开始执行语句的普通脚本，那么语句是在导入时执行——直到调用 pdb.run()为止，pdb 才进入并暂停脚本的运行过程。pdb 支持的其他函数参见本章后面的“调试器函数”一节。

如果知道脚本运行时会有异常出现，但不了解出现异常的原因，第三种方法是最好的。要以交互方式使用 Python，并且在导入和/或执行自己的模块前导入 pdb 模块。一旦发生异常并已经打印了前面的回跟踪信息，就需要调用 pdb.pm()对算后检查(post_mortem)调试器进行初始化。参见下例：

```
>>> import pdb
>>> import mymodule
Traceback (most recent call last):
      File "<stdin>", line 1, in ?
      File "./ mymodule. py", line 29, in ?
TypeError: source
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
>>> pdb. pm()
> ./ mymodule.py(29)
-> parse(source)
 (Pdb)
```

每种情况的结果都一样——直接进入调试器。

下节讲述利用调试器的机制。

1. 调试器界面

不管怎样调用调试器，最终会在调试器中结束并回到提示符(Pdb)状态。调试器支持很多不同的命令，这些命令执行脚本直到到达了断点为止或者基于逐行形式。

调试器界面遵循以下基本原则：

- 输入行首先被认为是调试器命令。如果输入的内容不能被识别为调试器命令，便被视为 Python 语句执行。是在被调试程序的环境中执行 Python 语句的，因此可以获得变量的值，更新变量，或者直接从调试器提示符中调用函数。如果发生异常，异常的名字将被打印出来，但调试器的状态不会改变，可以继续正常使用调试器。
- 自动把以感叹号(！)开始的命令识别为 Python 语句。
- 输入空行就表示再执行一遍刚刚输入执行的最后一条命令。
- 一行中可以输入多条命令，但要用两个分号(；；)隔开。注意对多条命令的隔开不是智能的，如果在字符串中出现了双分号，命令行也会在这个位置被分开。
- 调试器命令的参数要用空格或制表符(tab)隔开。

调试器第一次启动时会自动寻找称为.pdbrc 的文件，此文件用来加载一系列别名(简短的解释)或者用来运行其他命令。例如，可以在.pdbrc 文件中列出 5 个立即要求调试器执行的命令行。

调试器首先在用户主目录中查找.pdbrc 文件，然后在当前目录中查找。先加载的主目录中的版本，当前目录下的版本将覆盖所有初始定义。

下面就论述调试器所支持的命令。它们分为 3 类：一般命令、运行命令和断点命令。大多数命令都支持简写成为命令的首写字母。例如 h(elp)表示 help 命令，但可以缩写成 h。

一般命令　调试器支持许多一般命令，这些命令可以提供调试器或当前脚本的信息，或设置调试器选项。

Help　help 命令显示常规的帮助，包括调试器所支持命令的清单。如下所示：

```
h( elp) [command]
```

如果指定了 command 参数，将只显示相应的帮助信息。如果设置了 PAGER 环境变量，输出支持分页命令(如 more 或 less)。

Where　where 命令打印堆栈的跟踪，最近的帧显示在底部。如下所示：

```
w(here)
```

与发生异常时生成的输出相似，箭头指示的是当前帧。参见下例：

```
(Pdb) w
      /export/home/etc/mcslp/books/pytcr/ch24/<string>(1)?()
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
        /export/ home/ etc/ mcslp/ books/ pytcr/ ch24/ mymodule. py(12)?()
-> myfunc()
        /export/home/etc/mcslp/books/pytcr/ch24/mymodule.py(9) myfunc()
-> mysqrt(i)
> /export/home/etc/mcslp/ books/ pytcr/ ch24/ mymodule. py(3) mysqrt()
-> for i in xrange(iter):
```

在这个例子中，当前正在 mysqrt 帧内。

Up　up 命令把当前帧从现在的位置向上移动一层。如下所示：

```
u(p)
```

利用 where 命令的例子，当前帧将移到 myfunc。如果已经在最顶层，当前帧保持不变。

down　down 命令把当前帧从现在位置向下移一层。如下所示：

```
d(own)
```

利用 where 命令的例子，如果已经调用一次 up，该命令会将当前帧移回到 mysqrt。如果已经在最底层，则当前帧保持不变。

List　list 命令列出当前文件的源。默认情况下，列出环绕当前运行行的 11 行(当前行的前 5 行和后 5 行)。语法如下所示：

```
l(ist) [first [, last]]
```

如果给定 start，命令将列出环绕那一行的 11 行，如果指定 last 参数，命令将列出从 first 到 last 的所有行。

Args　args 命令列出当前函数的参数。如下所示：

```
a( args)
```

p　p 命令对 expression 求值，并打印结果，如下所示：

```
p expression
```

expression 可以访问当前帧的上下文中所有有效变量和函数。结果交给 print 语句输出。

Alias　alias 命令生成在用于调试器时运行 command 的别名 name。如下所示：

```
alias [name [command]]
```

command 参数不要加引号。可以用%1,%2……等指定的参数替换执行的命令，%*表示所有受支持的参数。下例：

```
alias dumpall for i in %*: print str(i)
```

创建一条新命令。此命令允许转出提供给命令的每个参数的字符串版本。如下所示：

```
(Pdb) dumpall string int float dict
```

执行任何其他调试器命令都可以使用别名。但它们大多数时像前面的例子一样，用来按一

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

定格式转出变量、对象或其他结构。别名可以嵌套使用，可以重写内置的调试器命令。在别名被删除(使用 unalias 命令)前，原来的命令被隐藏。

unalias　此命令删除别名 name。如下所示：

```
unalias name
```

quit　此命令退出调试器。正在执行的程序会中断。如下所示：

```
q(uit)
```

2. 单步执行命令

单步执行是执行 Python 语句的一个动作，可以按步单独执行，也可以按组执行(即运行一步执行了整个函数时)。单步执行基于逐行运行方式，监控程序的运行、语句所使用的变量，以及对语句的影响。尽管有些 Python 调试器提供一些其他辅助功能，但是下面是 3 种基本的单步命令：

- Step Into 运行当前行，后跟语句里所有函数或方法的运行。忽略变量的初始化，一直运行到调用函数或方法为止，停止于在被调用函数的第一条可执行语句处。
- Step Over 运行当前语句。所有调用的函数或方法被运行而不经过调试器处理，所以运行将停在当前文件的下一个可执行语句处。
- Step Out 继续运行直到当前函数或方法结束。运行停在下一个可执行语句处，即调用脚本的当前行的下一个函数调用，或者调用者的下一个语句。

单步执行通过断点可以独立监控每一行的执行。在研究执行顺序或者循环中的每一次迭代的细节时，这种方法尤其有用。

下面列出 Python 调试器所支持的单步命令。

Step　step 命令运行当前行，停在第一个可能停止的地方——当前行调用的第一个函数或者当前函数的下一行。如下所示：

```
s(tep)
```

从根本上而言，等价于前面论述的 Step Into。

Next　next 命令继续运行，直到当前函数的下一行或函数返回为止。如下所示：

```
n(ext)
```

从根本上而言，是 Step Out 和 Step Over 的组合。这条命令跳过当前行的函数调用，停在下一行开始执行前，或当前函数返回后的位置。

continue　continue 命令继续运行，直到遇到下一个断点为止。如下所示：

```
c(ont(inue))
```

3. 使用断点

断点是指脚本运行中合乎逻辑的中断。可以在指定的行号中设置断点，然后通过增加条件进一步限制是否真正中断运行。只有在条件返回真时才触发断点，否则不中断脚本的运行。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

Break　设置 lineno 参数时，break 命令在当前文件或者在参数 filename 指定文件的指定行中设置断点。如下所示：

```
b(reak) [[ filename:] lineno| function [, condition]]
```

请注意，filename 参数可以指定实际还没有加载的文件。如果给定 function 参数，命令将在指定函数的第一个可执行语句处设置断点。

如果给定 condition 参数，必须是正常的 Python 测试表达式的形式。比如在 parse 函数里设置只有 ignorenl 为 1 时触发断点，命令如下所示：

```
(Pdb) b parse, ignorenl == 1
```

如果默认所有参数，该命令列出当前所有断点。

Tbreak　tbreak 命令设置临时断点，这些断点功能与普通断点相同，只是在中断运行后，这些断点即被清除。如下所示：

```
tbreak [[ filename:] lineno| function [, condition]]
```

clear　clear 命令清除 bpnumber 指定的断点。如下所示：

```
cl(ear) [bpnumber [bpnumber ...]]
```

如果默认参数，该命令清除所有断点。

disable　disable 命令暂时禁用列出的断点。如下所示：

```
disable [bpnumber [bpnumber ...]]
```

如果默认参数，该命令禁用所有断点。

enable　enable 命令启用先前被禁用的断点。如下所示：

```
enable [bpnumber [bpnumber ...]]
```

如果默认参数，该命令启用所有断点。

ignore　在下一次迭代时，ignore 命令忽略 bpnumber 指定的断点。如下所示：

```
ignore　bunumber[count]
```

该命令禁用断点在下一次运行时起作用，然后立刻重新启用断点。如果指定了 count 参数，该命令在接下来的 count 次迭代中，忽略断点。

condition　无论在创建断点时是否已经设定了一个 condition，condition 命令设置当前断点的 condition。如下所示：

```
condition bpnumber [condition]
```

如果没有给出 condition，该命令将删除特定断点的 condition。

4. 调试器函数

相对于调试器的交互式界面，pdb 模式提供了一系列附加的函数。这些函数可以用来执行

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

一个代码段，然后再调用调试器，也可以从脚本中直接调用调试器。后一种方法有助于从异常处理程序中调用调试器。关于使用函数调用调试器的更多内容，参见本节的介绍。

表 24-1 列出了 pdb 模块所支持的函数。

表 24-1　调试器函数

函　　数	说　　明
run(statement [, globals [, locals]])	在调试器控制下运行 statement 语句。run 函数立即触发调试器，并在 statement 语句真正运行前中断脚本的运行。globals 参数和 locals 参数必须是在执行语句时使用的全局和局部变量的字典。默认操作是使用当前作用域定义的变量运行—— 比如_main_。除了在运行前立即调用调试器以外，run 函数在语法上与 exec 语句和 eval()函数相似
runeval(expression [, globals [, locals]])	除了按与 eval()相似的方式将字符串 expression 求值为 Python 语句以外，该函数等价于 run()函数。尽管 runeval()在表达式执行一结束就调用调试器，但函数与表达式返回同样的值
runcall(function[, argument, ...])	按所给定的参数调用 function 函数。在包含函数的语句执行前调用调试器。function 是一个代码对象，也就是说应该是一个函数或者方法的名字，而不是字符串
set_ trace()	进入当前帧的调试器。该函数可以从脚本中调用调试器，这与在交互式调试时设置断点的效果相同。无论脚本是否已调试完，该函数都会起作用
post_ mortem(traceback)	利用给定的 traceback 对象，开始脚本的算后检查(post-mortem)调试。最后一个 traceback 对象可以在 sys.last_traceback 中找到
pm()	等价于 post_ mortem(sys. last_traceback)

24.3　优化 Python 应用程序

影响应用程序运行的 bug 一般只是问题的一小部分。很多时候遇到的问题与逻辑无关，也不是录入拼写方面的错误。有时脚本看起来只是运行很慢，执行某一个函数比预想花费的时间多，或者数值较多时所用的时间与数值较少时所用的时间不成比例。

查找这些 bug 比较棘手。有时候问题很明显，大多数情况下要想找到解决问题的办法需要做一些研究。有一些明显的例子和经常突然出现的问题，将在下一节“手工优化”中详细讲述。这些问题仍依靠于手工阅读和检查代码。

查找这类问题的另一种方法是使用 Python 配置器。Python 配置器可以监控脚本运行并生成一个基于列表的配置文件(profile)，列表中包含每个函数的执行次数及其执行时间。对这些值进行综合考虑，就可以确定是哪些函数占用了过多的时间，因而知道哪些函数需要优化。“Python 配置器”一节将详细讲述如何使用这个工具。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

24.3.1　手工优化

脚本中有很多区域在不同单元中执行时可能要导致一些问题的产生。虽然不可能将它们一一列出或针对每个单独的脚本给出具体的建议，但仍可以指出一些在察觉有问题时值得注意的地方。

Python 语言具有很好的优化性能(更多细节请参见 23 章)，因此一般不会对 Phthon 本身产生怀疑，但这并不是说绝对没有错误。实际上使用了不够优化的算法，或者为了实现目的使用了错误的函数、模块及技巧都可能产生错误。

还要注意，有些脚本和函数无法做简单的优化或改进，还有些操作无论如何都要花费较长的时间。有一个工程，作者写的脚本执行了很长时间，不是因为脚本不够优化，而是它要处理的信息量太大了。它要删除一个有 19 000 条记录的自由文本数据库的内容，这些记录的大小约为 4k 到 16k。每条记录至多九个不同的关键字段，必须完全复制，并进行其他转换和计算。在大约十年前，这个脚本运行要将近 35 分钟。

计算机被习惯认为很快，其实它要完成一项指定的任务还是需要一定时间的。现在经常处理越来越多的数据集合。五年前，比较繁忙的站点的网络日志文件一天大约是运行 1~2MB，而现在同样的站点要创建大约 20~50MB，甚至可能更大的日志文件。计算机的速度在提高，但是数据量的增长速度要远远快于计算机的速度的提高，而且磁盘和存储器系统的速度仍然达不到 CPU 的要求。

下面列出了优化代码时应注意的方方面面和技巧：

- 在作用域较大时，使用 xrange()代替 range()，或者使用 while 循环。
- 对于存储连续的信息，使用数组或元组而不用字典。如果对数据的访问不是随机的，字典式存储会造成时间和空间的浪费。
- 对一系列值做收集、完全复制或汇总操作时，字典式存储比数组存储更有利。所有需要立即访问的数据都应存储于字典中，否则增加一个元素就要遍历整个数组。字典的关键字表使用散列算法优化，基本上可以瞬间完成对相应值的访问。
- 没有绝对必要，否则不要使用多后异常处理程序。例如创建一个打开并读取文件，并把读取的文件内容返回的函数，不要将异常放在函数中，或放在调用的附近。使用外部异常处理程序来处理可能出现的异常。
- 如果在循环中只有几条其他的语句，应尽量避免在较大的循环中调用复杂的子程序。取而代之，让函数自己处理循环，或者把函数的语句直接嵌入到循环块内。
- 使用循环时，尽可能将控制语句放在循环块的前面。例如，逐行处理文件的文本时，#符号一般忽略行，在循环块一开始就对它进行测试可以避免执行一些到最终不做任何事的语句。
- 可能的情况下，在函数中尽量使用列表而不使用连续的字符串。例如，在使用 print 语句打印多个值时用逗号将各个值分开。使用 + 符号来连接字符串需要做额外的处理。因为逗号在 print 语句中也表示空格，所以使用时要小心。另外还有一种方法，如果要将多个值连接在一起，可以优先选择使用 string.join()来生成最后的字符串，而不使用+操作。
- 在字符串里嵌入变量时，使用格式化操作符%来进行字符串连接。
- 在使用正则表达式时避免利用字符串切片来抽取值可能更快。例如，要从字符串提取日期，可以使用如下操作：

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
date = '20010723'
year = date[ 0: 4]
month = date[ 4: 6]
day = date[ 6: 8]
```

但如果使用正则表达式，可以把上面的操作缩减为以下一条语句。如下所示：

```
(year, month, day) = re. match( '(d 4)(d 2)(d 2) ', date). group( 1,2,3)
```

而且如果使用编译过了的正则表达式，对大量日期字符串操作的速度会更快。

```
datematch = compile( '(d 4)(d 2)(d 2) ')
(year, month, day) = datematch.match(date).group(1,2,3)
```

● 如果处理大量文本，使用正则表达式来进行替换操作，而不要用切片来抽取和连接来重组字符串。

● 使用临时变量来避免对序列、字典或属性的值做不必要的查找。例如，下面的脚本多次访问 self 的属性和数组。如下所示：

```
    def cumvolume(self):
PYTHON
        volume = 0.0
        for i in xrange(len(self.cubes) - 1):
            v = (self.cubes[i].x + self.cubes[i+1]. x) * \
                (self.cubes[i]. y + self.cubes[i+1]. y) * \
                (self.cubes[i]. z + self.cubes[i+1]. z)
            volume += v
    return volume
```

每次调用 self.cubes[i].x 都意味着要查找两个属性和一个数组，因此总共有 6 次循环，18 次操作。修改方法，使用临时值，操作可以将减半为 9 次。如下所示：

```
def cumvolume(self):
    volume = 0.0
    cubes = self.cubes
    for i in xrange(len(cubes) - 1):
        p1 = cubes[i]
        p2 = cubes[i+1]
        v = (p1.x + p2.x) * (p1.y + p2.y) * (p1.z * p2.z)
        volume += v
    return volume
```

尽管这样不会把速度提高 50%，但比原来的方式要改进 15—20%。

● 如果利用一个标准或扩展模块可以在 Python 中实现，就尽量避免通过 os. system()、os. exec*()、或 os. fork()来调用外部程序。启动一个外部程序包括要复制当前的过程或者创建新的过程，因此要增加许多额外的开销。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

● 当处理多个数据流时，使用 select 模块来确保没有正在等待发送或接收信息的数据流，或者使用线程会更好。

● 在系统中需要同时处理数据的地方使用线程。SMP 系统尤其适合使用这种方法，SMP 系统里允许使用多线程访问机器上所有的 CPU 的原始功能。

24.3.2　Python 配置器

Python 配置器是一种类似于调试器的工具，它有助于发现脚本里那些在应用程序执行时占用了大部分时间的函数和方法。和调试器一样，配置器所做的只是表明问题可能出现在哪里，它并不尝试去解决问题或者给出可选择的建议。

因为它要做额外的循环来收集第一手信息，所以使用配置器会增加一点运行脚本的开销，但这些额外的开销并不多，大约不到脚本真正运行时间的 10%。相对于这些信息的用处，收集它们所用的开销是值得的。

配置器实际分为两部分：profile 模块和 pstats 模块。profile 模块负责监控脚本不同部分的执行时间，并生成配置文件。然后 pstats 模块处理配置文件，生成关于脚本运行时间的报告。

1. 获取配置文件

本节例子中的脚本使用了两个函数：myfunc()和 mysqrt()。作为测试，没有必要担心实际完成了什么模块。在这个例子中，模块计算了一个数的平方然后是这个数的平方根，所有操作放在可变长的循环中。下面是脚本的源程序。如下所示：

```
def mysqrt(iter):
        import math
        for i in xrange(iter):
                sq = I * i
                rt = math.sqrt(sq)

def myfunc():
        for i in xrange( 00):
                mysqrt(i)
```

在命令行和交互模式中都可以创建配置文件。要在交互式模式下使用它，导入 profile 模块，然后在想要测试的模块里调用 run()执行函数。如下所示：

```
>>> import profile
>>> import mymodule
>>> profile. run('mymodule.myfunc() ')
                203 function calls in 0.350 ( PU seconds

        Ordered by: standard name

ncalls      tottime     percall     cumtime     percall     filename:lineno(function)
1           0.000       0.000       0.290       0.290       <string>:1(?)
200         0.270       0.001       0.270       0.001       mymodule. py: 1(mysqrt)
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

1	0.020	0.020	0.290	0.290	mymodule. py:65(myfunc)
1	0.060	0.060	0.350	0.350	profile:0(mymodule.myfunc())
0	0.000		0.000		profile:0(profiler)

以上报告的信息是新生成配置文件的概要，粗略地报告出执行的函数以及每个函数执行的时间。

对这些报告做进一步分析，会得到更多的信息。第一行显示的是调用函数的次数和执行提供给配置器的函数(或脚本)所用的时间，以及原始调用的次数，即不包括递归产生的调用。

“Ordered by”一行显示的是输出函数列表所使用的排序格式。在这里函数是按字母顺序列出的，也可以把顺序改为先列出最高层或最底层调用的函数。在本章后面“统计报告”一节中的 pstats 模块部分讲述如何改变输出顺序。

各列内容如下：

- ncalls 是每个函数被调用的次数。如果这个数字是以##/##形式显示的，那么第一个数字表示函数被调用的总次数，第二个数字表示原始调用的次数。
- tottime 报告函数执行的总时间，这个数值不包括本函数调用其他函数所用的时间。
- percall(1)报告每次调用函数的平均时间，不包括子函数所用的时间。
- cumtime 报告函数执行的总时间，包括子函数花费的时间。
- percall(2)报告每次调用函数的平均时间，包括子函数所用的时间。
- filename: lineno(function)显示的是所报告的文件名、行号和函数。

利用用于调试器的相同系统，通过命令行也可以得到关于整个脚本的同样报告。如下所示：

```
$ python /usr/ local/ lib/ python2.1/ profile. py mymodule. py
                    204 function calls in 0.340 CPU seconds
        Ordered by: standard name
ncalls    tottime    percall    cumtime    percall    filename:lineno(function)
1         0.020      0.020      0.280      0.280      <string>:1(?)
1         0.000      0.000      0.260      0.260      mymodule.py:1(?)
200       0.240      0.001      0.240      0.001      mymodule.py:1(mysqrt)
1         0.020      0.020      0.260      0.260      mymodule.py:65(myfunc)
1         0.060      0.060      0.340      0.340      profile:0(execfile('mymodule.py'))
0         0.000                 0.000                 profile:0(profiler)
```

上面两种方法唯一的问题是它们只产生关于要配置的脚本或函数的单次运行的信息。无法访问原始数据，因此无法改变报告的输出、计算或排序的方式。

要避免这一点，可以在交互模式下，使用配置器将原始数据保存为文件。要实现前述要求，在调用 run()时，给出要保存数据的文件名。如下所示：

```
>>> import profile
>>> import mymodule
>>> profile.run('mymodule.myfunc()', 'mymodule.prof')
>>>
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

这些命令会在当前目录下生成 mymodule.prof 文件，而不是立即生成报告。在前面的例子中，通过利用 pstats 模块来处理 mymodule.prof 文件。

2. 统计报告

pstats 模块提供一个单一的类：Stats，用于对配置文件内容处理和报告。要创建新的 Stats 类的实例，只需要把文件加载到构造器时给出文件名，如下所示：

```
import pstats
myprofile = pstats.Stats('mymodule.prof')
```

还有其他方法可以报告文件包含的信息。比如，上例中在配置器直接执行时将生成的输出重显，就请调用如下所示的命令：

```
myprofile.strip_dirs().sort_stats('stdname').print_stats()
```

Stats 类支持许多控制格式和报表输出的方法，这些方法汇总见表 24-2。表 24-3 列出 sort_stats()方法所支持的排序次序。

表 24-2　Stats 类支持的方法

方　法	说　明
strip_ dirs()	删除统计表中每个模块名的前缀路径名。注意该操作只有一定的破坏性，数据中的路径信息将从数据中被永久删除
add(filename [, ...])	在已存在的统计表中增加配置数据。该方法可用于合并同一脚本或模块中许多不同调用的信息，或者合并使用同一模块的多个脚本的信息。信息合并为每一个模块或函数组合生成的附加报告
sort_ stats(key [, ...])	按指定关键字对统计表排序，有效值列表见表 24-3。该命令可设多个值，可对多个关键字排序。注意该函数按给定的参数对统计表永久地排序。欲改变排序顺序，必需再次调用该函数
reverse_ order()	报告机制生成的统计表的顺序反转
print_ stats(restriction [, ...])	打印输出给定对象的所有统计表。输出格式与给出的示例相同，排序顺序由 sort_stats()方法指定。restriction 选项用于限制输出一组选择的函数。如果 restriction 是整数，只报告前 n 个函数；如果是 0 到 1(包括 1)间的浮点数，只报告一定百分比的函数(例如，0.25 = 25%)；如果 restriction 是字符串，则被解释为正则表达式，用于匹配使用 re 模块的文件名、行号或函数。 例如，print_ stats('foo')只匹配含有'foo'的文件名或函数，注意该命令是按 filename: function 严格匹配的，因此对于 foo: ，只匹配以 foo 结尾的文件。要匹配函数请使用:foo 。 如给出多个参数，则按顺序进行匹配。例如 print_ stats(0.25, ': foo') 先限制列表为前 25%，然后才是匹配 foo 的函数

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

方　法	说　明
print_ callers(restriction [, ...])	打印配置统计中调用每个函数的所有函数清单。输出同 print_ stats() 的输出一样，restriction 与 print_ stats 中的使用方式相同。注意对于每个函数，每个调用者附加的一个带括号数字表明这个调用的次数，没有被括起的数字表示在函数中所花的累积时间。参见本章后面的例子
print_ callees(restriction [, ...]	等价于 print_ callers(),但列出的是被调用的函数，而不是调用者

表 24-3　sort_stats()方法的排序参数

参　数	说　明
'calls'	报表按函数调用次数排序
'cumulative'	报表按每个函数的累计执行时间排序
'file'	报表按文件名排序
'module'	报表按模块(文件)名排序
'pcalls'	报表按每个函数的原始调用次数排序
'line'	报表按函数的行号排序
'name'	报表按函数名排序
'nfl'	报表按函数名/文件名/行号排序，行号按数值比较(例如：1，2，...10，11...23，24)。
'stdname'	报表按标准的函数名/文件名/行号排序，行号按字母顺序比较(例如：1,10,11,...19,2,20,21,...29,3,30)
'time'	报表按函数的平均执行时间排序

例如，使用下面的语句按时间顺序打印报表。

```
myprofile.strip_dirs().sort_stats('time').print_stats()
```

其输出如下所示：

```
Mon Jul 23 16: 11: 21 2001          mymodule. prof

          203 function calls in 0.310 CPU seconds

      Ordered by: internal time

ncalls     tottime    percall    cumtime    percall    filename:lineno(function)
200        0.300      0.002      0.300      0.002      mymodule.py:1(mysqrt)
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

1	0.010	0.010	0.310	0.310	mymodule.py:65(myfunc)
1	0.000	0.000	0.310	0.310	profile:0(mymodule.myfunc())
1	0.000	0.000	0.310	0.310	<string>:1(?)
0	0.000			0.000	profile:0(profiler)

也可以限制只输出 mymodule. py 文件使用的函数，如下所示：

```
myprofile.strip_dirs().sort_stats('time').print_stats('mymodule')
```

生成的报表如下所示：

```
Mon Jul 23 16: 11: 21 2001          mymodule. prof

          203 function calls in 0.310 CPU seconds

     Ordered by: internal time
     List reduced from 5 to 3 due to restriction <'mymodule'>
```

Ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
200	0.300	0.002	0.300	0.002	mymodule.py:1(mysqrt)
1	0.010	0.010	0.310	0.310	mymodule.py:65(myfunc)
1	0.000	0.000	0.310	0.310	profile:0(mymodule.myfunc())

print_ callees()生成的报表类似于如下所示：

```
Ordered by: internal time

Function                                called...
mymodule.py:1(mysqrt)                   --
mymodule.py:65(myfunc)                  mymodule.py:1(mysqrt)(200)      0.300
profile:0(mymodule.myfunc())            <string>:1(?)(1)                0.310
<string>:1(?)                           mymodule.py:65(myfunc)(1)       0.310
profile:0(profiler)                     profile:0(mymodule.myfunc())(1) 0.310
```

而 print_ callers()生成的报表如下所示：

```
Ordered by: internal time
Function                        was called by...
mymodule.py:1(mysqrt)           mymodule.py:65(myfunc)(200)     0.310
mymodule.py:65(myfunc)          <string>:1(?)(1)     0.310
profile:0(mymodule.myfunc())    profile:0(profiler)(1)    0.000
<string>:1(?)                   profile:0(mymodule.myfunc())(1)    0.310
profile:0(profiler)             --
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

3. 使用数据

了解了上述内容，应该知道如何处理这些信息。如本章开始部分所讲，配置器只是一个工具，它可以提供函数执行次数的信息。配置器所做的工作是帮助找出脚本中最费时的部分和需要优化的函数。

在本章的例子中，使用资源最多的是 mysqrt()函数，它花费了相当多的时间却没有有效地完成任何事。

在现实世界中，必须对调用的函数加以研究并确定它是否可以改进和优化。基本上说，符合以下一种或两种情况的函数需要研究：

- 被调用的次数多，而每次运行的时间短：这种函数可能是在循环中被调用的。尽管有时这种情况是不可避免的，但如果这样做占用了大量的时间，就应该检查一下是否可以将函数的内容直接放到循环中。调用函数需要向堆栈中放入并从中删除大量的额外信息，因此减少函数的调用次数有助于改善性能。
- 运行时间长，而调用次数少：这些函数需要手工优化以减少它们的运行时间。使用本章前面提到的手工优化技巧检查一下哪里可以改进。

最后，尽管配置器在处理中有所帮助，但调试和调整脚本仍需要手工干预。无论是否借助于配置器，多年得到的关于做什么、不做什么的经验，对优化脚本有很大帮助。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第25章　文档编制和文档

在阅读源文件代码时，无论代码是自己编写的或是他人编写的，如果代码没有任何注释说明某一部分完成的功能，将是件很糟糕的事。比如，在编写代码时，忘记利用注释注明代码使用了什么算法或是使用了什么特殊方法解决了问题。

注释是编制文档最基本的形式，但许多程序员常常忘记使用它。使用注释有助于提醒程序员，或在程序员之间交流代码是怎样工作的。

如果您想要在因特网上发布应用程序或模块，就需要编写相应的文档。可以有许多方法，比如，编写独立的 README 文档或创建 HTML、PDF 文档，或创建 Unix、troff/nroff 格式的文档。

也可以使用另外的解决方法，就是利用嵌入的文档字符串。这些属性依附于对象或源代码的其他实体上，接下来可以利用多种不同工具将其抽取成为合适的文档格式。因为文档绑定为脚本和模块的一个元素，可以知道文本来自哪里，什么与它相关联。仍然是比较好的方法，利用文本字符串来提供脚本的文档：想像一下显示应用程序里直接来自于对象的帮助文本。

本章主要讲述怎样编写好的注释和注释是怎样合并到 Python 脚本这方面的知识。本章还介绍了文档字符串和怎样创建，及以后把模块里的标志符抽取成为单独文档。关于文档编制的更多的信息请见附录 B。

关于编写好的文档举例，请访问下列 URL：

- http:// www. ibiblio. org/ mdw/ HOWTO/ Software- Release- Practice-HOWTO/documentation.html
- http:// www. techscribe. co. uk/ techw/ faq. htm
- http:// mindprod. com/ unmain. html
- http:// techwriting. about. com/ careers/ techwriting/ cs/ humour/
- http:// www. python. org/ sigs/ doc- sig/

25.1　注释

当您回过头来看自己的代码时可能会想到，“我在这儿做什么？”或是“我为什么要选择用这种方法？”。要想让自己或是他人能很好地读懂代码，使用注释是最好的方法，虽然您可能想到永远不会忘记某一部分代码，但经过 3 天、几周、甚至一个月或更长的时间您就需要帮助了。

显然，解决的方法就是使用注释，说明脚本是怎样工作的，说明脚本或模块里的不同函数和方法处理的是哪些参数以及返回值。写出好的注释是一门艺术，我们可以看到不同一些好的和差的例子。

一旦完成脚本，下一步就是把注释转换成为文档，这样其他人可以不读注释就能看懂脚本。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

请注意，如果将要编写正确的脚本文档，为什么要先写注释再编写文档呢？应当在一边编写注释一边编写文档。

25.1.1　写注释

众所周知，遗憾的事实是：大多数程序员都不写注释，或是只在软件完成后再写注释。这是一种很坏的习惯，不使用注释会使代码很难读懂，完成程序后再写注释有可能写出错误的注释，因为您可能已忘记了最初是怎样设计代码的。实际上，应该在编程时就写出注释，并保证注释信息是正确的。还有一个不错的主意，写下不能做的事情记录，这样就避免再去尝试不能做的工作。

通过在一行中插入符号（#），注释被引入到 Python 脚本中；从这个字符之后开始一直到这一行结束为止的所有字符串都被当作注释。如下所示：

```
print message # Output the message we received
```

在每行使用（#）符号来实现多行注释

```
for line in lines:          # Extract individual lines
    pline = parse( line)    # Parse the line to get the bits we want
    print pline             # Print out the extracted elements
```

这还是一个很好的经验，如上所示，在多行程序段内整齐排列注释，表明注释是一大段消息当中的一部分。

25.1.2　编写好的注释

在脚本中包含注释只是解决方法的一部分——必须确认注释有意义和真正有用。考虑下面程序段：

```
if message: print message      #   Prints a message if it exists
```

这一行没有给出任何有用的消息，而且也不知道这一行在做什么。应该写出这样的注释：说明为什么要这样做，做的是什么，而不是怎样做的，就像下面一样：

```
If message: print message      # Prints out the warning message from the
                               # get_ message() function, assuming a valid
                               # message was returned.
```

写注释时，要考虑包含下面几个元素：

- File headers　文件头应该包含文件内容、文件的目的，如果可能，还包含版权声明。例如，可以在模块的最上面包含一些像下面一样的信息：

```
# Module MyModule
# Copyright Me, 2001
# Provides functions and a class for manipulating time information
```

如果使用了修订版本控制系统，如 RCS 或 CVS，还应该包含修订号、版权及其他与工程

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

有关的字符串。比如，要引入修订号，可以通过 RCS 或 CVS 通过包含在文件头中的$Revision$来自动更新。

● Variable/object names　注释中应包含变量名和对象名，用来解释它们是怎样使用的，以及应该使用什么样的数据。为了有助于调试，还可以包含数据取值的有效范围。例如：

```
bankaccount = Account( 500) # Holds the basic bank account detail
errormsg = ''                # Stores the value of the error message
                             # returned during an exception
```

● Functions　应该被注释为其定义的一部分，至少应该解释函数使用的参数和要返回什么信息，当然能达到的信息也是个不错的主意。然而，如果想详细注释，大概要考虑使用 Python 的文档字符串。本章后面将介绍这方面的例子。在使用注释时，要确保注释缩排为一个封闭的块。如下所示：

```
def parser( line):
# Accepts a line as text, parses the contents to extract the
# individual words and elements and then returns a list of the
# elements to the caller
```

● Classes　应该进行说明，这样读者就可以理解类的整个作用域，但不用说明属性或方法。可以在声明时单独说明。比如：

```
class Account():
                # Base class for all accounts
                  def _ init_( self, balance):
                # Creates a new instance of Account, setting the balance attribute
               self. balance = balance # Holds the accounts current value
```

● In-line comments　用于高亮显示重要的特殊行或节，但不用每一行都标记。程序员在看代码时就应当知道 for 循环或 if 语句是什么意思。

最后，下面是一些在写注释时应当避免的问题：

● 不要每一行都注解。这对读者没有帮助，反而有可能使读者感到迷惑。

● 不要反复地用英语说明此行执行什么函数，对 print 语句说明“打印消息”没什么用，对程序员来说，这好像有点烦人。

● 不要“美化打印”注释。在注释中添加一些窗口修饰和空白经常会转移注释的真正意义，读者在读代码时不得不卷动窗口，当然读起来就不太方便。

如果考虑到这些约束，就可以写出比较好的注释来，其他程序员也能看懂。然而，既然要编写模块和脚本的文档，就应该考虑使用 doc 字符串。下面就将看到其功能。

25.2　嵌入文档字符串

所有模块、函数、类或方法，都隐含地拥有一个在声明期间附加在其上的文档字符串。其工作方式就是把语句定义之前出现的常量字符串作为此对象的文档。此字符串被设定为对象的

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

__doc__属性。例如，在下面的代码段中：

```
class Spam:
        """ Spam class documentation starts here..."""
        def method( self, arg):
            """ Method documentation starts here..."""
        def documentation():
            print "Class documentation:", Spam.__doc__
print "Method 'method' documentation:", Spam.method.__doc__
```

文档字符串可以利用任何引号、单引号、双引号或三重引号来指定，但是，三重引号是接受的标准。然而要记住，在文档字符串里使用引号时要注意引号使用的类型统一。

要得到信息，在下面的例子里可以看出，只须访问__doc__属性。如下所示：

```
print myfunc.__doc__
```

因为此信息是个属性，所以也可以在以后更新这个信息。在利用文档字符串作为帮助文本时，已经使用了这个属性。想像一下具有一个可以开或关的按钮，可以访问帮助来描述它的当前设置。更好的，实际上无需影响文档字符串而可以做到上述要求，文档字符串在其他时候可以用诸如 pydoc 的文档工具来抽取。例如：

```
class SetButton:
            """ Button """
            def __init__( self, name):
            self.value = 0
            self.name = name
            self.__doc__ = SetButton.__doc__ + self.name + " Off"
        def switchon( self):
            self.value = 1
            self.__doc__ = SetButton.__doc__ + self.name + " On"
        def switchoff( self):
            self.value = 0
self.__doc__ = SetButton.__doc__ + self.name + " Off"
```

使用文档字符串系统的最大好处是，它把文档附着到具体对象上作为此对象定义的一部分。因为文档可以用作对象的一个属性，就可以在运行期间访问脚本中的文档字符串，好像阅读源代码一样简单。

虽然文档字符串系统直到 Python 1.5 才开始广泛使用，由于文档字符串已被使用并被合并到大多数模块中，所以主要的 Python 开发小组和大多数第三方开发者已经将文档字符串作为脚本和模块中的方法使用。

虽然最近也推出了许多工具和可接受的文档字符串的开发标准，文档字符串仍然还没有广泛地使用。例如，Python 文档仍然利用传统的文档工具编写，大多数使用的仍是 SGML。这种情况在以后会得到逐步改善，如果想成为这种程序中的一员，考虑加入文档特别兴趣组(SIG, Documentation Special Interest Group)。可以在附录 B 中找到 SIG 的列表以及怎样加入它们。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

25.3　把嵌入字符串翻译为文档

文档 SIG 提供了一系列的叫 pythondoc(或 pydoc)的工具，从给定实体的模块文件中把文档字符串抽取出来。还有功能相似的其他工具，包括 HappyDoc。但是应然要坚持提供文档编制工具作为标准库的一部分。

25.3.1　pydoc 工具

主要的 pydoc 脚本是各种交易的插口。基本上，它从 Python 模块里把文档字符串抽取出来，显示为文本或是 HTML，后跟所有结构化的格式，并显示相应的函数、类和其他元素。

在创建和/或安装 Python 时，pydoc 脚本应该安装在正确的位置上。要运行，就从命令行上直接运行 pydoc 命令即可。如下所示：

```
$    pydoc
```

无需任何参数，将产生一些小的帮助消息。

默认情况下，pydoc 搜索这些模块所在的目录，就是说，目录定义在 sys.path 中。例如，要显示 random 模块的文档，只需简单键入如下所示的代码：

```
$ pydoc random
Python Library Documentation: module random

NAME
random - Random variable generators.
...
```

多线程注意：这里使用的随机数生成器不是线程安全；有可能两个调用返回同样的随机值。然而，可以在每个线程实例化不同的 Random()函数的实例来得到不共享状态的随机数生成器，然后使用.setstate()和.jumpahead()函数来移动生成器来将整个句点段分离出去。例如，

```
...
```

总之，如果在参数中使用了前斜杠（/）符号，那就假定参数是要为其显示文档的模块文件的路径。

1. 关键字搜索

通过使用-k 命令行选项，能搜索库目录中的所有 Python 文件的摘要行（模块文档中的第一行）的关键字。结果是搜索到的其中包含那个关键字的模块清单。例如，要得到创建随机数的模块清单，可以使用如下所示的命令：

```
$ pydoc -k random
    random - Random variable generators.
    whrandom - Wichman- Hill random number generator.
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

请注意，这个选项只对摘要行中的单词进行查找。模块当中的其他单词将被忽略。

2. Web 服务器

因为 pydoc 能将文档字符串转换成为 HTML 格式，那么就可以为 Python 文档提供一个 web 服务器。事实上，pydoc 包含了对 SimpleHTTPServer 类的支持，这个类可以将 HTML 版的文档直接发送到 Web 服务器上。

要使用这项功能，使用-p 选项来启动指定的 TCP/IP 端口上的 Web 服务器。例如，在 1001 端口启动服务器。如下所示：

```
$ ./ pydoc -p 1001
pydoc server ready at http:// localhost: 1001/
```

这样，在同一台机器上，就可以通过键入在 Web 浏览器中显示的 URL 来访问文档。也可以通过网络在其他机器上访问站点。作者使用这项功能作为从网络上任何机器访问文档的一种方法，而无需独立打开文件。图 25-1 显示了模块清单的结果。

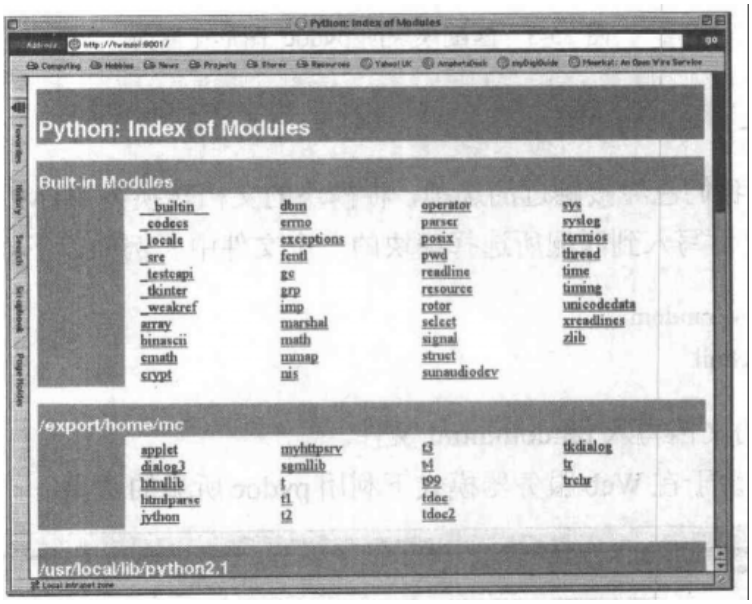

图 25-1　pydoc 的文档列表

3. 图形化界面

通过使用 Tk，pydoc 提供了一种对于文档库的图形化界面。遗憾的是，并非如所说的那么好：限定只能使用关键字（等价于-k 选项）来搜索模块。一旦找到所关注的内容，就启动内置 Web 服务器，并打开您的浏览器允许查看文档。

图 25-2 是简单的窗口，图 25-3 是显示所匹配模块的清单的扩展窗口。

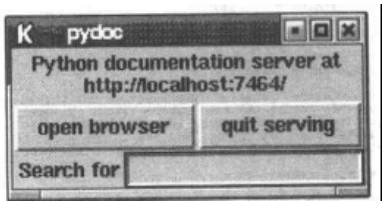

图 25-2　初始的 pydoc 图形化界面

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

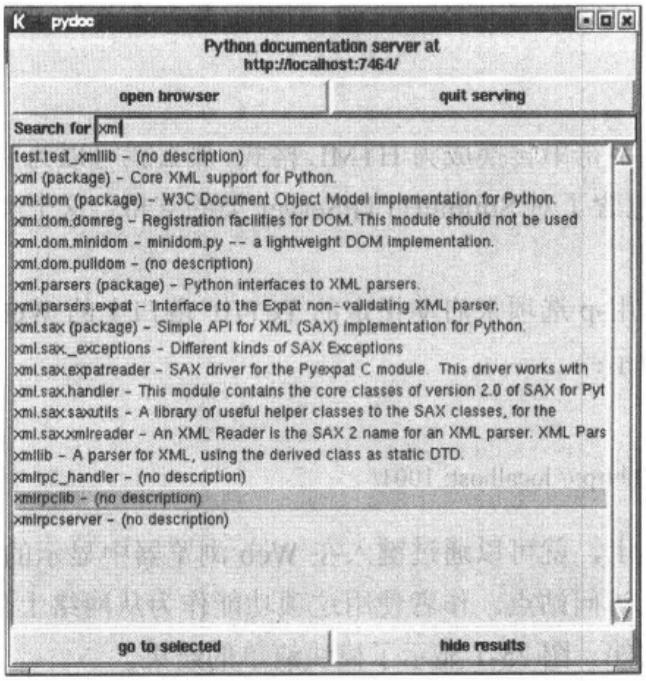

图 25-3　匹配模块的 pydoc 图形化界面

4. 输出为 HTML

最后一个选项是我们已经接触过的选项，将模块的文档转换为 HTML 文件。在命令行中使用-w 选项，将结果文件写入到匹配所选择模块的一个文件中。例如，下面的命令：

```
$ pydoc -w random
wrote random. Html
```

将 random 模块的文档写入 random.html 文件。

这个结果文件等价于在 Web 服务器模式下利用 pydoc 所见的结果。一个样本请参见图 25-4。

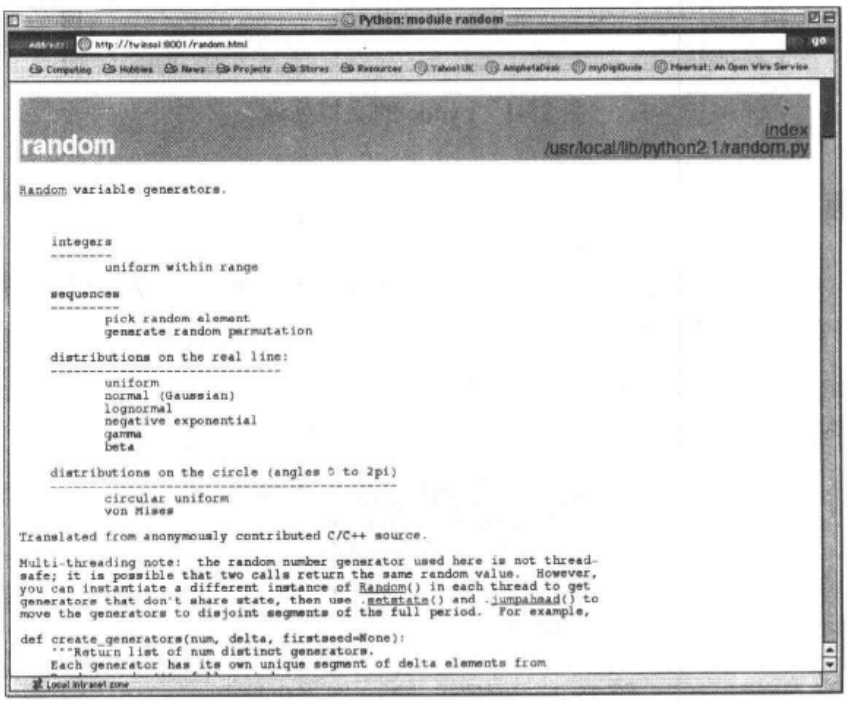

图 25-4　pydoc 生成的 HTML 格式的 random 模块

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

25.3.2　结构化的文本格式化规则

在关注 pydoc 工具的规范之前，需要检查一下实际上怎样编写文档字符串第一位置上包含的文本。“结构化的文本格式化规则”是一套规则，目的在于说明不同元素的，比如项目符号列表或数字列表和突出显示的单词在文档字符串中如何进行描述。

这个信息在开发过程中有很多，并不是每一个文档工具，包括 pydoc，都遵守这个规则。要了解具体定义了什么，可以加入文档 SIG，详见附录 B。

下面是一些基本规则：

- 结构化的文档字符串由几个被一个或是多个空行分开的段落组成。
- 每一段落都是一级，通过缩进来区分的，基本上等同于当前块的缩进。第 0 级是第一行后的第一个非空白行。大多数情况下，这意味着在源代码中，0 级不用缩排；第 1 级是第一个缩排；第 2 级是第二个缩排等等，依次类推。请看下面：

```
"""
        This is level 0 paragraph
            This is a level 1 paragraph
        Back to level 0"""
```

- 如果下一个最低级别上具有前置的段落，那么这个段落就是子段落，如上例所示，第二个段落是第一段落的子段落。同样的规则也适合于根据代码的结构而缩进的文档字符串，就是说，方法的文档字符串是整个类的 doc 字符串的子段落。

另外，一些符号有特殊的含义：

- 凡是由 - 、*或 o 开始的段落都作为无顺序的项目列表元素对待。例如：

```
""" Bulleted list:
* First bullet
    * Second bullet
    """
```

- 由数字序号开始并跟着空格的段落都作为有顺序的列表元素对待。请注意，元素不是为您而排序的。例如：

```
""" Numbered list:
    1 First bullet
    2 Second bullet
    3 Third bullet
    """
```

- 第一行有文本，后面跟着空格和--符号的段落都作为可描述的列表元素对待。前置文本作为元素标题对待。为了避免 pydoc 产生疑惑，应在段落中小心使用双横线来分开文本。例如，可以创建下面的可描述清单。如下所示：

```
""" Items:
The First item -- is just a lead in to the rest of the list
The Second item -- Looks slightly different
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
"""
```

● 由“example”或“examples”结束的子段落都作为代码举例对待，并且是无格式的输出。“example”或“examples”后的标点可以忽略。例如，下例就把一段样本代码引入 doc 字符串中：

```
""" To use this class, just create a new instance. For example:
        myparser = MyDocStringParser()
        """
```

● 由单引号封闭的文本（左边为空格加单引号，右边的单引号加空格或是标点）作为样本代码对待。例如，要引入单词“parser”作为样本代码的一部分，使用如下所示的代码：

```
""" When you call the method 'parser' to parse the document the
        information is then..."""
```

● 由单字符*前后包围并各有空格（或右边跟着标点）的文本都利用斜体来突出表示。例如，字符串“do *not* use this function,”中的单词“not”就是斜体。

● 由双字符*前后包围并各有空格（或右边跟着标点）的文本都使用粗体来突出表示，例如，字符串“do ** not** use this function,”中的单词“not”就是加粗表示。

预计这些规则会有所改变或改善，就像 pydoc 或是其他的文档工具被改善一样。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第26章　Python扩展

尽管 Python 语言有极强的处理各种任务的功能，但有时 Python 需要使用外部 C 语言库的功能。事实上，许多曾使用过的标准 Python 扩展，可能就直接或间接地使用了 C 语言的库。例如，socket 模块就使用了提供网络通信平台上的基本套接字库，同时 urllib 模板依赖 socket 模块才可从因特网上下载文件。

建造与 C 语言模块接口的 Python 扩展并不困难。一旦拥有了基本技巧，就能够与任何外部库或 C 函数建立接口。

本章将一步一步地解释在 Python 中构建扩展的过程，也将阐述在 Python 和 C 两者之间进行信息交换的方法与技巧。若想获得 Python API 的更多信息，可到 Python 网站(http://www.python.org)查看 API 文档资料。

26.1　基本接口

Python 的扩展接口非常简单。首先围绕要导出的 C 函数建立包装器(Wrapper)，包装器用来处理 Python 对象(变量)与底层 C 函数中的变量和所需值之间的转换。若需要的话，包装器将返回的数据转换为 Python 对象，Python 对象即可应用在 Python 应用程序中。

为进一步说明这个过程，先来看包含两个函数的一个简单 C 文件。其中一个函数利用 prinf() 函数打印原始字符串，而另一个函数接受两个浮点值计算圆锥的体积，并返回结果。

```
#include <math.h>
#include <stdio.h>
float volumecone(float radius, float height)
{
float volume;
volume = (1.0/3.0)*M_PI*(radius*radius)*height;
return volume;
}
void rawprint(char *string)
{
printf("%s", string);
}
```

包装器源程序实际上要执行两个任务：首先，它围绕要调用的独立函数建立，对每个要调用的外部 C 函数，都需要一个包装器。再者，必须注册 Python 函数名、支持接口的包装器和它要接受的参数类型；由此初始化模块并且开始整个任务。包装程序如下例所示：

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
#include <Python.h>
extern void rawprint(char *);
extern float volumecone(float, float);
PyObject *testex_volumecone(PyObject *self, PyObject *args)
{
float radius, height, volume;
if (!PyArg_ParseTuple(args, "ff", &radius, &height))
{
return NULL;
}
volume = volumecone(radius, height);
return Py_BuildValue("f", volume);
}
PyObject *testex_rawprint(PyObject *self, PyObject *args)
{
char *string;
if (!PyArg_ParseTuple(args,"s", &string))
{
return NULL;
}
rawprint(string);
Py_INCREF(Py_None);
return Py_None;
}
static PyMethodDef testexmethods[] = {
{"volumecone", testex_volumecone, METH_VARARGS,
"compute the volume of a cone given the radius and height" },
{"rawprint", testex_rawprint, METH_VARARGS, "print a raw string"
},
{ NULL, NULL },
};
void inittestex(void)
{
Py_InitModule("testex", testexmethods);
}
```

有关包装器每一部分的具体细节将在本章的下一个部分里论述，但是先浏览一下包装源程序本身，在这儿将能获知许多重要细节。

26.1.1　编写包装器

包装器有三部分：

- 第一部分提供必要的信息使包装器能调用外部 C 函数；
- 第二部分包含实际的包装器自身；
- 第三部分是注册信息。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

1. 设置背景

本例的背景信息是程序的前几行：

```
#include <Python.h>

extern void rawprint(char *);
extern float volumecone(float, float);
```

第一行导入关于 Python 函数的函数定义，其中，Python 函数用于支持和管理接口。下面两行为 C 源文件中的两个外部函数设置原型。如果要连接已存在的库，必须使用多个 include 说明语句，调入原型信息。

2. 包装器

第一个包装器是称为 testex_volumecone 的关于圆锥体积的 volumecone 函数。volumecone 函数接受两个参数：两个浮点值，一个是基圆的半径，另一个是锥体的高。包装器的定义如下所示：

```
PyObject *testex_volumecone(PyObject *self, PyObject *args)
{
```

因为包装器总是要返回一个值给调用者，所以 PyObject 返回要求的返回值。在本例中，返回计算结果。即使不返回任何值的函数也要返回一个有效 Python 对象。否则，解释器会得到 NULL 值并产生一个错误。

包装器 testex_volumecone 的两个参数也是 Python 对象。第一个参数是 self，是这个函数的挂钩(hook)，且此函数是被 Python 解释器内部标识为一个代码对象。args 是一列参数。当调用 volumecone 函数时，把 args 传给 volumecone 函数。请注意：此步骤无需标识参数类型和它们包含的数据。

下一行定义了 C 语言变量，可以用这些变量把参数传递给已存在的扩展函数并得到结果。如下所示：

```
float radius, height, volume;
```

PyArg_ParseTuple 函数把给定的参数列表转换为提供给包装器的参数列表，并从参数列表中抽取独立元素成为 C 语言局部类型。

```
if (!PyArg_ParseTuple(args, "ff", &radius, &height))
{
return NULL;
}
```

PyArg_ParseTuple 函数的第一个参数是已给定的参数列表；第二个参数是格式化的字符串，此参数指定应被抽取和转换的变量类型。在此例中，要抽取两个浮点值，而这两个浮点值通过地址插入到 C 语言局部变量中。此阶段结束时，应把 Python 函数调用所提供的两个浮点值抽取成为 C 语言局部变量。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

请注意，输出被捕获。如果 Python 调用没有包含正确类型的值，PyArg_ParseTuple 就返回 NULL，NULL 被解释器当作错误。此时，在返回 NULL 值给 Python 之前，可引发一个适当的 Python 异常——请参见本章后面的“异常”部分，可以获得更多的信息。

在下面一行，实际上调用 volumecone 函数。这个函数相当直观，提供两个浮点值就可获得结果。如下所示：

```
volume = volumecone(radius, height);
```

最后一步把返回值转换为 Python 对象，Python 对象接下来能被传送回 Python 解释器，最后回至曾首先调用此函数的 Python 脚本。用 Py_BuildValue 可做到这一点。Py_BuildValue 在语法上等价于 PyArg_ParseTuple；但 Py_BuildValue 不是把参数的元组转换为 C 变量，而将 C 变量转换为 Python 变量。如下所示：

```
        return Py_BuildValue("f", volume);
}
```

第二个包装器，testex_rawprint，比第一个包装器简单。它接受一个字符串而非两个浮点值。

```
    PyObject *testex_rawprint(PyObject *self, PyObject *args)
{
char *string;
if (!PyArg_ParseTuple(args,"s", &string))
{
return NULL;
}
rawprint(string);
Py_INCREF(Py_None);
return Py_None;
}
```

testex_rawprint 与第一个包装器之间的最大区别是，rawprint 函数不返回任何值。然而，为了防止解释器将 rawprint 函数作为错误标识为 NULL 值，仍须返回一个有效的 Python 对象。为了达到这种效果，需增加 Py_None 值的引用计数，且返回引用计数值。Py_None 是 Python None 值的 C 语言等价值。因此，从根本上而言，testex_rawprint 不返回任何值给调用者，但并不引发任何错误。

3. 初始化模块

包装模块的最后一部分是建立初始化函数。当加载模块时，为了注册有效函数和其他对象，其中这些对象是关于模块的名称空间的，Python 解释器将调用初始化函数。

为了保存信息，第一步是创建结构。此结构是包装模块支持的一列函数。每一个函数要列出 Python 函数名，列出支持 Python 函数调用的包装器，列出定义函数所支持的参数类型的一些选项。第 4 个可选参数，如下面的源程序中所示，还生成 doc 字符串(在_doc_属性中)。

```
static PyMethodDef testexmethods[] = {
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
{"volumecone", testex_volumecone, METH_VARARGS,
"compute the volume of a cone given the radius and height" },
{"rawprint", testex_rawprint, METH_VARARGS, "print a raw string" },
{ NULL, NULL },
};
```

前面的源程序使用包装器 testex_volumecone 来注册 volumecone 函数。volumecone 函数接受参数列表。于是就可使用 METH_VARARGS 选项。对于不接受任何值的函数，不必指定任何参数。如果想要函数支持关键字参数，可以使用 METH_KEYWORDS。如果既要支持参数列表，又要支持关键字参数，可以使用 METH_VARARGS 选项和 METH_KEYWORDS 选项(用“|”，请参见表 26-1)。

请注意，必须用一空输入项来终止配置列表。

最后，创建初始化函数。Py_InitModule4 函数用四元素格式(包括 dos 字符串)实现初始化。如果想省略 dos 字符串，则应使用 Py_InitModule 函数。初始化函数的名字很重要。例如，创建了 testex 模块，初始化函数名即为 inittestex。inittestex 一般通过调用函数来初始化模块，提供模块名、testex 以及前面论述的对象定义表。如下所示：

```
void inittestex(void)
{
Py_InitModule4("testex", testexmethods);
}
```

26.1.2　编译扩展

编译扩展要稍微复杂一点。编译前，需要编译源文件和包装文件，并确保包装文件中可以访问 Python.h 头文件。接着需要将后端模块和包装器编译成为共享目标文件。当导入文件时，解释器将加载共享目标文件。显然，如果要包装库函数，不必考虑另一 C 文件；但必须确定已经访问了所需的头文件和库文件。

如果这些说起来比较复杂，不必担心，还有更容易的方法。因为建造扩展的 Uinx 技术最为直观，本章将着重阐述这一问题。此技术对于 Uinx 类操作系统，例如：QNX，BeOS 和 Mac OSX，也是行之有效的。可以进入 Python 网站获取关于 Windows 和 MacOS 技术的有关细节。

如果已安装了 Python，安装的文件中含有 Makefile.pre.in 文件，可以在 Python 库目录的 config 目录中找到 Makefile.pre.in 文件。(典型为/usr/local/lib/python#.#，其中 #.#是 Python 的版本号)。

Makefile.pre.in 利用另外一个叫 Setup 的文件创建 Makefile；反回来，Makefile 构建模块。Setup 文件的格式遵循上 Setup 文件，于 Setup 文件用来建造来自 Python 配置的标准扩展模块。

Setup 文件从如下所示的语句开始：

```
*shared*
```

此说明语句建立系统来构建共享扩展。可通过使用带有 Makefile 的 static 目标文件,构建一个嵌有模块的静态版本 Python 解释器。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

后续行将定义模块名、需求的 C 文件，及其他构建扩展模块所必须的选项。例如,如果 C 语言文件是 testex.c，而包装文件是 testexwrapper.c，则 Setup 文件中的命令行如下所示：

```
testex testex.c testexmodule.c
```

有了 Setup 文件和 Makefile.pre.in 文件的副本，就能够用如下所示的命令创建 Makefile：

```
$ make -f Makefile.pre.in boot
```

要构建扩展，仅需键入如下所示的代码：

```
$ make
gcc -fPIC -g -O2 -Wall -Wstrict-prototypes -I/usr/local/include/python2.1 -
I/usr/local/include/python2.1 -DHAVE_CONFIG_H -c ././testex.c -o ./testex.o
gcc -fPIC -g -O2 -Wall -Wstrict-prototypes -I/usr/local/include/python2.1 -
I/usr/local/include/python2.1 -DHAVE_CONFIG_H -c ././testexmodule.c -o
./testexmodule.o
gcc -shared ./testex.o ./testexmodule.o -o ./testex.so
```

结果应是一共享对象文件，testex.so，该文件包含的内容可令扩展生效。

正如前面提到的，也可以运行 make static，make static 构建扩展；而不是把结果置入共享对象文件中。该共享对象文件将 Python 与对象文件及在当前 Python 版本中必需的其他对象结合起来，且在当前目录中创建可执行的新 Python 文件。

26.1.3 测试结果

为了测试模块，在扩展模块的同一目录下激活 Python，并试着导入模块，这样就可看到导入测试演示模块的结果，如下所示：

```
$ python
Python 2.1 (#1, Aug 2 2001, 20:52:30)
[GCC egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)] on linux2
Type "copyright", "credits" or "license" for more information.
>>> import testex
>>> testex.printraw("Hello this is a raw test")
Traceback (most recent call last):
File "<stdin>", line 1, in ?
AttributeError: 'testex' module has no attribute 'printraw'
>>> testex.rawprint("Hello this is a raw test")
Hello this is a raw test>>>
>>> testex.volumecone(4.5, 3.5)
74.220123291015625
>>
```

看来都生效了！

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

26.2　数据转换

函数 PyArg_ParseTuple()和函数 Py_BuildValue()被用来在 Python 对象与 C 数据类型之间转换信息，第三个函数 PyArg_ParseTupleAndKeywords()能与其他函数一起使用来接受关键字参数。关键字参数被置在以空字符结尾的一列字符串中，字符串包括所有参数名。每个函数的实际格式如下所示：

```
int PyArg_ParseTuple(PyObject *args, char *format, ...);
int PyArg_ParseTupleAndKeywords(PyObject *args, PyObject *kwdict
char *format, char **kwlist, ...);
PyObject *Py_BuildValue(format, ...);
```

每个函数都或多或少地共享同一个 format 选项，表 26-1 包括所支持的 format 选项的完整清单。

表 26-1　转换 Python 和 C 语言变量的格式选项

格　式	Py 类型	C 类型	描　述
s	String	char*	空字符结尾的字符串
s#	String	char*,int	字符串和长度。允许字符串包含空字节
z	String 或 None	char*	空字符结尾的字符串或 NULL
z#	string 或 None	char*,int	字符串和长度或 NULL
b	Integer	char	8 位整数
h	Integer	Short	16 位短整数
i	Integer	Int	整数
l	Integer	long	长整数
c	String	Char	单字符(字符的 Python 字符串)
f	Float	float	单精度浮点数
d	Float	Double	双精度浮点数
D	Complex	py_complex	复数
O	Any	PyObject*	任意 Python 对象
O!	Any	type,PyObject*	特定类型的 Python 对象
O&	Any	converter,any	由转换器函数 converter 处理的 Python 对象
S	String	PyStringObject*	Python 串对象

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

格　式	Py 类型	C 类型	描　述
(items)	Tuple	Vars	数据项元组。此处 items 确切的说是格式说明符的字符串(如本表中其他地方所用)。要把 Python 转换为 C，vars 应是一 C 变量地址的列表；要从 C 转换为 Python，vars 应是一 C 变量列表
[items]	List	Vars	仅从 C 转换为 Python。数据项列表，此处 items 确切是格式说明符的一个字符串，(本表中别处也使用)，vars 是一 C 变量列表
{items}	Dictionary	Vars	仅从 C 转换为 Python。数据项的字典，提供的变量列表应是关键字/值对中的关键字和值
|			可选参数的起点(仅适用于把 Python 转换为 C)
:			参数尾。附加文本作为错误报文中的函数名(仅适用于把 Python 转换为 C)
;			参数尾：附加文本作为错误报文中的错误文本(仅适用于把 Python 转换为 C)

把 Python 参数列表转换为 C 变量的一些例子如下：

```
/* Convert into two floats */
PyArg_ParseTuple(args, "ff", &floata, &floatb);
/* Convert a string and integer */
PyArg_ParseTuple(args, "si", &string, &inta);
/* Convert two tuples */
PyArg_ParseTuple(args, "(ss)(if)", &stringa, &stringb, &inta, &floata);
```

冒号(：)与分号(；)格式顺序根据具体情况而定。冒号(：)标识参数列表的结束，任何紧跟在冒号后的文本都被用作出错消息中的函数名；出错消息是在 Python 会话期转换时产生的。分号(；)也标识参数表的结束，任何紧跟在分号后的文本都被用作出错消息；但出错消息是在转换错误出现时产生。举个例子：如果一个函数接受两个参数，则可定义函数格式如下：

```
PyArg_ParseTuple(args, "ii;function accepts exactly two arguments",
&inta, &intb);
```

26.3　引用计数管理

也许已经猜到，PyObject*类型是用在 Python API 中保存 Python 对象的信息的。确切地说，

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

PyObject*是一种自定义正在存储的数据类型的特殊结构，这种特殊结构被用在 API 的整体当中，保存从模块数据到单独字符串、数组及其他类似的 Python 数据类型的所有数据。

作为 Python 中的解释器，在第 3 章、第 6 章、第 10 章已介绍过。这三章侧重于阐述处理 Python 对象时将要采取的必要措施。这是因为底层对象与符号表中的名字之间的关系仅是一个引用。还有，如数组这样的嵌套类型，实际存储对其他 Python 对象的引用的数组。

26.3.1 引用计数

虽然在正常 Python 使用时，使用引用不会引起问题。但当操纵来自 C API 的相同对象时，就必须注意引用问题。

概括地说，每一个引用有一个惟一的引用计数。引用计数告诉 Python 解释器有多少个其他对象正在引用该对象。例如，下列代码中：

```
numa = 45
numb = 56
lista = [numa, numb]
```

lista 对象有一个为“1”的引用计数，但保存 numa 和 numb 值的对象有一个为“2”的引用计数。当 numa 在当前模块的字典(例如，它的名称空间)或在列表中时，使用 lista 对象允许引用计数计数一次。如果删除 lista 的第一个元素，含有 numa 对象的引用计数将减 1。

如果一个对象的引用计数达到 0，Python 认定不再需求此对象，此对象占有的存储空间将被释放。在前面的例子中，此情况仅发生在 numa 最终跳出作用域、脚本可能结束时。

这里引用计数存在很大的潜在问题。假如对象的引用计数大于 0，甚至此时无对象引用此对象，不会引起问题。但是，当此对象仍在别的地方使用，却减少对象的引用计数到 0 的值将是非常危险的。

处理来自外部的 C 应用程序的相同对象时(或当与嵌入解释器中的 Python 对象会话时，如第 27 章所述)，应用的原则相同。需要确保解释器知道何时正在使用特定对象；因此，解释器期望能够访问命令中引用计数的值，同时解释器要知道何时完成对象的引用。

全部过程是由引用计数处理的。为了表明想使用一个对象引用，就递增其引用计数；要结束对象引用，就递减其引用计数。

26.3.2 引用类型

在 Python 里，对象引用实际上分为两种类型：拥有的(owned)和借用的(borrowed)。拥有的引用是指向 Python 对象的指针。如果该对象的引用计数增加，表明将使用该对象引用计数的值。借用的引用只是 Python 对象的裸指针，该对象的引用计数不能递增。

当函数创建新的 Python 对象时，拥有的引用通常被使用和创建。一旦创建了对象，则由创建它的函数将其引用归类为“拥有的”。于是必须更新引用计数，以表明该对象正在被使用。当访问诸如现存异常这样的已存对象的信息时或访问列表项时，将使用借用的引用。

26.3.3 更新引用计数

Python 提供 4 个宏来处理对象的引用计数。Py_INCREF()宏和 Py_DECREF()宏分别递增和

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

递减所提供对象的引用计数，但此对象必须是有效的 Python 对象。Py_XINCREF()宏和 Py_XDECREF()宏力图递增和递减所提供 Python 对象的引用计数，但如果引用计数为 NULL，对象将被忽略。

如果要理解哪种情况使用这些宏将是棘手的问题。拥有的引用规则相当直观：当要使用，尤其要更新对象时，将递增引用计数；结束对象使用时，将递减引用计数。例如，在包装器里创建临时对象，使用临时对象前递增引用计数，使用后递减引用计数。然而，正返回对象的引用计数不会减少；否则，在对象到达调用者之前将被释放。

借用的引用的规则更简单：除非想拥有对象，否则不必增加或减少引用计数。

作为一个大致指南，请遵循以下基本规则：

- 所有创建新对象的函数要构建拥有的引用，并且管理引用计数的值。
- 正在返回的对象的引用计数不应减小。
- 使用 Py_INCREF()使对象被存储。在 Python 里，保留对象的记录直到解释器结束，将是无害而有利的。
- 大多数序列和映射函数将拥有的引用返回给存储单元。

如要获取更多信息，查看 http：//www.python.org/dos/ext，Python CAPI 文档资料。

26.4　异常

尽管前节例子中的函数给调用者返回 NULL，但这表明在外部函数中出现了错误。Python 使用异常来突出问题，甚至在解释器的限制中也使用异常。为了支持与异常 API 同样的功能，需要一种方法来处理异常。

在一些实例中，可以使用像 PyErr_BadArgument()这样简单的函数，来引发 TypeError 异常给调用者。例如，可重写扩展函数 printraw() 如下：

```
PyObject *testex_rawprint(PyObject *self, PyObject *args)
{char *string;
if (!PyArg_ParseTuple(args,"s", &string))
{
PyErr_BadArgument();
return NULL;
}
rawprint(string);
Py_INCREF(Py_None);
return Py_None;
}
```

在 Python 里调试此版本函数，结果如下所示：

```
>>> rawprint(45)
Traceback (most recent call last):
File "<stdin>", line 1, in ?
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
TypeError: bad argument type for built-in operation
```

此例中报出的消息由 Python API 自动生成。若要得到自己定义的消息，须手工建立异常，可用下列命令行替代：

```
PyErr_SetString(PyExc_TypeError, "You gave me an invalid argument");
```

在解释器里将产生下述结果：

```
>>> rawprint(45)
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: bad argument type for built-in operation
```

其他种类异常的引发方式相同。请注意，前面每个例子中，PyErr_*()函数仅设定异常类型，一直到返回给调用者(此处为 Python 解释器)NULL，实际的异常才引发和处理，或生成一个回跟踪。本节中的所有函数仅设置或释放当前异常类型；假如没有异常生效，普通异常将由解释器产生，出现这种情况的原因仅仅是因为解释器接收到 NULL。

如果要创建一个特定于模块的异常，需要创建新的异常类型。可使用函数 PyErr_NewException()创建新异常类型。PyErr_NewException()函数接受 3 个参数。第一个参数关键，因为它定义要产生的异常的新名字。用户如果想能够访问这个异常，还能获取并处理异常，需要修改初始化函数，在模块字典里添加新的异常对象。

如要创建一个 testex 模块的具体 badarg 异常，请使用如下所示的代码：

```
PyObject *TestExExc;
void inittestex(void)
{
PyObject *mymod, *moddict;
mymod = Py_InitModule4("testex", testexmethods);
moddict = PyModule_GetDict(mymod);
TestExExc = PyErr_NewException('testex.badarg', NULL, NULL);
PyDict_SetItemString(moddict, "badarg", TextExExc);
}
```

现在可得到如下所示的错误：

```
import testex
try:
rawprint(45)
except testex.badarg:
print "Bad argument type supplied"
```

表 26-2 包括了设置和释放异常的函数的完全清单，表 26-3 列出内建 Python 异常类型所使用的 C 常量。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

表 26-2　异 常 函 数

函　　数	说　　明
void PyErr_Print()	打印标准回跟踪(如 Python 脚本里已引发的不能处理的异常)，接着清除错误指示器。如果异常没被设置，则会引发致命错误
PyObject* PyErr_Occurred( )	如果错误已被设置，则返回异常的类型(请参见表 26-3)；若无错误出现，返回 NULL。请注意，此时接收的是对异常类型的借用引用，所以无须管理引用。使用 PyErr_GivenExceptionMatches()来匹配返回值为真正异常
int PyErr_ExceptionMatches(PyObject *exc)	如果当前异常与 exc 匹配，则返回真。如无异常设置，则引发内存访问故障
int PyErr_GivenExceptionMatches (PyObject *given, PyObject *exc)	如 given 与异常 exc 匹配，则返回真。反转并比较元组与对象。若 given 为 NULL，则引发内存访问故障
void PyErr_NormalizeException (PyObject**exc, PyObject**val, PyObject**tb)	把异常 exc 正规化为 tb。当 PyErr_Fetch 返回不能实际匹配 exc 中的类的一个异常类(val)的实例时，需要调用此函数
void PyErr_Clear()	清除错误指示
void PyErr_Fetch(PyObject **ptype, PyObject **pvalue, PyObject **ptraceback)	把错误信息检索为错误类型、错误值、回跟踪对象。如果不存在错误条件，返回 NULL。可用此函数临时进行检索；如果需要，可用此函数清除异常状态。使用 PyErr_Restore()设置错误条件。请注意，所生成对象的引用计数需要管理
void PyErr_Restore(PyObject *type, PyObject *value, PyObject *traceback)	设置错误指示器。如果三个参数全为 NULL，则清除错误指示器。一旦函数已调用，需要清除每个参数的引用计数
void PyErr_SetString(PyObject *type, char *message)	利用 type 异常（如表 26-3 所定义)和 message 出错消息来设置错误指示器
void PyErr_SetObject(PyObject *type, PyObject *value)	利用 type 异常(如表 26-3 所定义)设置错误指示器。并非以字符串的格式来设置附加信息，此版本使用一个对象
PyObject* PyErr_Format(PyObject *exception, const char *format, . . .)	利用 exception 里的异常来设置错误指示器。深层参数的工作方式与 printf()相似，其中，format 是格式字符串，仅接受 4 格式类型：c(int 类型提供的字符)；d(int 类型提供的十进制数)；x(int 类型提供的十六进制数)；s(char* 类型提供的字符串)。请注意，width.precision 前缀也得到支持，但忽略 width
void PyErr_SetNone(PyObject *type)	PyErr_SetObject(type,Py_None)的缩写
int PyErr_BadArgument()	PyErr_SetString(PyExc_TypeError,message)的缩写，用于错误参数条件下

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

函　　数	说　　明
PyObject* PyErr_NoMemory()	PyErr_SetNone(PyExc_MemoryError)的缩写，用在内存溢出条件
PyObject*PyErr_SetFromErrno(PyObject *type)	便利函数。引发一个 type 类异常，其中异常数据是由 errno 值及与 errno 值相对应的错误消息(来自于 strerror())组成的元组对象。用于 C 库函数(自动设置 errno)所引发的错误
void PyErr_BadInternalCall()	PyErr_SetString(PyExc_TypeError,message) 的缩写
int PyErr_Warn(PyObject *category, char *message)	发出警告消息。category 参数应是有效 的 Warning 对象，message 是用来打印警告的消息。标准 warning 类型有 PyExc_Warning,PyExc_UserWarning PyExc_DeprecationWarning 、 PyExc_SyntaxWarning 和 PyExc_RuntimeWarning。输出被发送给 sys.stderr。如果用户指定警告作为错误对待，则引发异常。无异常引发，则返回 0；若有异常引发，则返回 1
int PyErr_WarnExplicit(PyObject *category, char *message, char*_lename, int lineno, char *module, PyObject *registry)	发出对所有警告属性的显式控制的警告消息
int PyErr_CheckSignals()	检查一个信号是否被发送给当前过程。若被发送，激活相应的基于 Python 的信号处理程序
PyObject*PyErr_NewException(char *name, PyObject *base, PyObject *dict)	创建新的异常对象。name 参数应是以 module.class 形式给出的新异常名。若使用 NULL 和 PyExc_Exception,则 base 参数指定基本异常类型。dic 参数用于指定类变量与方法的字典(如果 NULL 使用当前环境的话)
void PyErr_WriteUnraisable(PyObject *obj)	把警告消息打印到 sys.stderr 上，表明虽然已设置了异常，但解释器不可能引发异常

表 26-3　Python 异常类型的 C 常量

C 异常常量	Python 异常
PyExc_Exception	Exception
PyExc_StandardError	StandardError
PyExc_ArithmeticError	ArithmeticError
PyExc_LookupError	LookupError
PyExc_AssertionError	AssertionError
PyExc_AttributeError	AttributeError
PyExc_EOFError	EOFErro

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

C 异常常量	Python 异常
PyExc_EnvironmentError	EnvironmentError
PyExc_FloatingPointError	FloatingPointError
PyExc_IOError	IOError
PyExc_ImportError	ImportError
PyExc_IndexError	IndexError
PyExc_KeyError	KeyError
PyExc-KeyboardInterrupt	KeyboardInterrupt
PyExc_MemoryError	MemoryError
PyExc_NameError	NameError
PyExc_NotImplementedError	NotImplementedError
PyExc_OSError	OSError
PyExc_OverflowError	OverflowError
PyExc_RuntimeError	RuntimeError
PyExc_SyntaxError	SystemError
PyExc_SystemError	SyntaxError
PyExc_SystemExit	SystemExit
PyExc_TypeError	TypeError
PyExc_ValueError	ValueError
PyExc_WindowsError	WindowsError
PyExc_ZeroDivisionError	ZeroDivisionError

26.5　低层对象访问

Python C API 包括用于访问和操纵核心对象类型的一系列函数。本节中，从表 26-4 到表 26-17 描述了用于操纵不同对象类型的函数。这些表包含返回类型、函数名和参数。每一个函数应是自解释的。如需更多信息，请双击 Python/C API 来查看文档资料。

表 26-4　操纵通用 Python 对象的函数

返 回 类 型	函　　数
int	PyCallable_Check(PyObject*o)
PyObject *	PyObject_CallFunction(PyObject *callable_object,char *format, . . .)

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

（续表）

返 回 类 型	函　　数
PyObject *	PyObject_CallMethod(PyObject *o,Char *methodname, char *format, . . .)
PyObject *	PyObject_CallObject(PyObject *callable_object,PyObject *args)
int	PyObject_Cmp(PyObject *o1, PyObject *o2,int *result)
int	PyObject_Compare(PyObject *o1, PyObject *o2)
int	PyObject_DelAttr(PyObject *o, PyObject *attr_name)
int	PyObject_DelAttrString(PyObject *o, char *attr_name)
int	PyObject_Delltem(PyObject *o, PyObject *key)
PyObject *	PyObject_GetAttr(PyObject *o, PyObject *attr_name)
PyObject *	PyObject_GetAttrString(PyObject *o, char *attr_name)
PyObject *	PyObject Getitem(PyObject *o, PyObject *key)
int	PyObject_HasAttr(PyObject *o, PyObject *attr_name)
int	PyObject_HasAttrString(PyObject *o, char *attr_name)
int	PyObject_Hash(PyObject *o)
int	PyObject_lsTrue(PyObject *o)
int	PyObject_Length(PyObject *o)
int	PyObject_Print(PyObject *o, FILE *fp, int flags)
PyObject *	PyObject_Repr(PyObject *o)
int	PyObject_SetAttr(PyObject *o, PyObject *attr_name,PyObject *v)
int	PyObject_SetAttrString(PyObject *o, char *attr_name,PyObject *v)
int	PyObject_Setitem(PyObject *o, PyObject *key,PyObject *v)
PyObject *	PyObject_Str(PyObject *o)
PyObject *	PyObject_Type(PyObject *o)

表 26-5　操纵数值对象的函数

返 回 类 型	函　　数
PyObject *	PyNumber_Absolute(PyObject *o)
PyObject *	PyNumber_Add(PyObject *o1 PyObject *o2)
PyObject *	PyNumber_And(PyObject *o1, PyObject *o2)
int	PyNumber_Check(PyObject *o)
PyObject *	PyNumber_Coerce(PyObject **p1, PyObject **p2)

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

返回类型	函　数
PyObject *	PyNumber_Divide(PyObject *o1, PyObject *o2)
PyObject *	PyNumber_Divmod(PyObject *o1 PyObject*o2)
PyObject *	PyNumber_Float(PyObject *o)
PyObject *	PyNumber_Int(PyObject *o)
PyObject *	PyNumber_Invert(PyObject *o)
PyObject *	PyNumber_Long(PyObject *o)
PyObject *	PyNumber_Lshift(PyObject *o1 PyObject*o2)
PyObject *	PyNumber_Multiply(PyObject *o1, PyObject *o2)
PyObject *	PyNumber_Negative(PyObject *o)
PyObject *	PyNumber_Or(PyObject *o1, PyObject *o2)
PyObject *	PyNumber_Positive(PyObject *o)
PyObject *	PyNumber_Power(PyObject *o1, PyObject *o2,PyObject *o3)
PyObject *	PyNumber_Remainder(PyObject *o1,PyObject *o2)
PyObject *	PyNumber_Rshift(PyObject *o1, PyObject *o2)
PyObject *	PyNumber_Subtract(PyObject *o1, PyObject *o2)
PyObject *	PyNumber_Xor(PyObject *o1, PyObject *o2)

表 26-6　操纵序列对象的函数

返回类型	函　数
int	PySequence_Check(PyObject *o)
PyObject *	PySequence_Concat(PyObject *o1, PyObject *o2)
int	PySequence_Count(PyObject *o, PyObject * value)
int	PySequence_DelItem(PyObject *o, int i)
int	PySequence_DelSlice(PyObject *o, int i1, int i2)
PyObject *	PySequence_Getitem(PyObject *o, int i)
PyObject *	PySequence_GetSlice(PyObject *o, int i1, int i2)
int	PySequence_In(PyObject *o, PyObject *value)
int	PySequence_Index(PyObject *o, PyObject *value)
PyObject *	PySequence_Repeat(PyObject *o, int count)
int	PySequence_Setitem(PyObject *o, int i, PyObject *v)

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

（续表）

返回类型	函　数
int	PySequence_SetSlice(PyObject *o, int i1, int i2,PyObject *v)
PyObject *	PySequence_Tuple(PyObject *o)

表 26-7　操纵映射对象的函数

返回类型	函　数
int	PyMapping_Check(PyObject *o)
int	PyMapping_Clear(PyObject *o)
int	PyMapping_DelItem(PyObject *o, PyObject *key)
int	PyMapping_DelItemString(PyObject *o, char *key)
PyObject *	PyMapping_GetItemString(PyObject *o, char *key)
int	PyMapping_HasKey(PyObject *o, PyObject *key)
int	PyMapping_HasKeyString(PyObject *o, char *key)
PyObject *	PyMapping_Items(PyObject *o)
PyObject *	PyMapping_Keys(PyObject *o)
int	PyMapping_Length(PyObject *o)
int	PyMapping_SetitemString(PyObject *o, char *key, PyObject *v)
PyObject *	PyMapping_Values(PyObject *o)

表 26-8　创建和转换整数的函数

返回类型	函　数
long	PyInt_AsLong(PyObject ^iobj)
int	PyInt_Check(PyObject *obj)
PyObject*	PyInt_FromLong(long)
long	PyInt_GetMax()

表 26-9　创建和转换长整数的函数

返回类型	函　数
double	PyLong_AsDouble(PyObject *lobj)
long	PyLong_AsLong(PyObject *lobj)
long long	PyLong_AsLongLong(PyObject *lobj)
unsigned long	PyLong_AsUnsignedLong(PyObject *lobj)
unsigned long long	PyLong_AsUnsignedLongLong(PyObject *lobj)

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

返回类型	函　数
void *	PyLong_AsVoidPtr(PyObject *lobj)
int	PyLong_Check(PyObject *obj)
PyObject *	PyLong_FromDouble(double)
PyObject *	PyLong_FromLong(long)
PyObject *	PyLong_FromLongLong(long long)
PyObject *	PyLong_FromUnsignedLong(unsigned long)
PyObject *	PyLong_FromUnsignedLongLong(unsigned long long)
PyObject *	PyLong_FromVoidPtr(void *)

表 26-10　创建和转换浮点数的函数

返回类型	函　数
int	PyFloat_Check(PyObject *obj)
double	PyFloat_AsDouble(PyObject *fobj)
PyObject *	PyFloat_FromDouble(double)

表 26-11　创建和转换复数的函数

返回类型	函　数
Py_complex	PyComplex_AsCComplex(PyObject *cobj)
int	PyComplex_Check(PyObject *obj)
PyObject *	PyComplex_FromCComplex(Py_complex *cobj)
PyObject *	PyComplex_FromDoubles(double real, double I)
double	PyComplex_ImagAsDouble(PyObject *cobj)
double	PyComplex_RealAsDouble(PyObject *cobj)

表 26-12　创建和转换字符串对象的函数

返回类型	函　数
char *	PyString_AsString(PyObject *str)
int	PyString_Check(PyObject *obj)
PyObject *	PyString_FromString(char *str)
PyObject *	PyString_FromStringAndSize(char *str, int len)
int	PyString_Size(PyObject *str)

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

表 26-13　创建、更新和转换列表的函数

返回类型	函　数
int	PyList_Append(PyObject *list, PyObject *obj)
PyObject *	PyList_AsTuple(PyObject *list)
int	PyList_Check(PyObject *obj)
PyObject *	PyList_Getitem(PyObject *list, int index)
PyObject *	PyList_GetSlice(PyObject *list , int i, int j)
int	PyList_Insert(PyObject *list, int index,PyObject Aobj)
PyObject *	PyList_New(int size)
int	PyList_Reverse(PyObject *list)
int	PyList_Setitem(PyObject *list, int index,PyObject *obj)
int	PyList_SetSlice(PyObject *list, int i, int j,PyObject *slc)
int	PyList_Size(PyObject *list)
int	PyList_Sort(PyObject *list)

表 26-14　操纵元组的函数

返回类型	函　数
int	PyTuple_Check(PyObject *ob j)
PyObject *	PyTuple_Getitem(PyObject *tup, int index)
PyObject *	PyTuple_GetSlice(PyObject *tup, int i, int j)
PyObject *	PyTuple_New(int size)
int	PyTuple_Setitem(PyObject *tup, int index,PyObject *obj)
int	PyTuple_Size(PyObject *tup)

表 26-15　操纵字典的函数

返回类型	函　数
int	PyDict_Check(PyObject *obj)
void	PyDict_Clear(PyObject *dict)
int	PyDict_DelItem(PyObject *dict, PyObject *key)
int	PyDict_DelItemString(PyObject *dict, char *key)
PyObject*	PyDict_GetItem(PyObject *dict, PyObject *key)
PyObject *	PyDict_GetItemString(PyObject *dict, char *key)
PyObject *	PyDict_Items(PyObject *dict)

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

返回类型	函　数
PyObject *	PyDict_Keys(PyObject *dict)
PyObject *	PyDict_New()
int	PyDict_SetItem(PyObject *dict, PyObject *key,PyObject *val)
int	PyDict_SetItemString(PyObject *dict, char *key,PyObject *val)
int	PyDict_Size(PyObject *dict)
PyObject *	PyDict_Values(PyObject*dict)

表 26-16　操纵文件的函数

返回类型	函　数
FILE *	PyFile_AsFile(PyObject *file)
int	PyFile_Check(PyObject *obj)
PyObject *	PyFile_FromFile(FILE *, char *, char *, int (*)(FILE *))
PyObject *	PyFile_FromString(char *name, char *mode)
PyObject *	PyFile_GetLine(PyObject *file, int)
PyObject *	PyFile_Name(PyObject *file)
void	PyFile_SetBufSize(PyObject *file, int size)
int	PyFile_SoftSpace(PyObject *file, int)
int	PyFile_WriteObject(PyObject *file, PyObject *obj, int)
int	PyFile_WriteString(char *str, PyObject *file)

表 26-17　操纵模块的函数

返回类型	函　数
int	PyModule_Check(PyObject*obj)
PyObject*	PyModule_GetDict(PyObject*mod)
char*	PyModule_GetFilename(PyObject*mod)
char*	PyModule_GetName(PyObject*mod)
PyObject*	PyModule_New(char*name)

26.6　下一章的内容

现在读者已经拥有了足够的信息，在本章当中的各种关于数据转换的表的武装下，支持来

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

自于外部函数库的函数。大多数 Python 扩展模块实际上在同一个基本原则基础上工作，利用处理真正函数调用的包装器交换数据。

除了已看到的函数所提供的基本数据转换之外，还有更广泛的函数操纵对象及其属性，更新完全来自于 C 语言的核心 Python 对象类型。这些信息太多，本书里不可能囊括所有的信息。但在 Python API 文档资料里，可找到关于低层 Python 对象交互作用的函数的信息及其他细节。有关到哪儿去查找 Python 文档这一问题，请参见附录 B。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

第27章　Python嵌入

通过把对 C/C++函数的调用、对象和方法嵌入到 Python 应用程序里，将 C/C++和 Python 集成在一起，当然也能用其他方式完成此功能。

这种方式叫做嵌入。通过在 C/C++应用程序中嵌入 Python，可以把一个脚本元素添加到应用程序中(想像一下可以像脚本化 Microsoft Word 一样脚本化应用程序,此处用的是 Python)，或者只是提供对 Python 对象、类或系统的访问。用 C++通常不能做到这些。

当然，还有些合理化限制。使用本章阐述的嵌入技术来提供对通常为扩展库所支持的模块的访问，显然还做不到。可以直接从 C 里使用这些扩展函数，但是仍然有很多情况需要使用 Python。例如：因为通过用 urllib 从因特网上下载一个 C/C++程序中的文件比打开和管理 C 语言中的内部套接字完成此任务要简单地多，所以可以使用 Python 从因特网上下载一个 C/C++程序中的文件。

本章将解释如何把 Python 集成到 C/C++应用程序里，涵盖内容从嵌入过程的简单技巧一直到在 C 应用程序内运用不同 Python 实体的例子。

27.1　嵌入原则

在学习构造嵌入式 Python 应用程序的技巧之前，需要了解一些基本原则。首先，需要考虑从嵌入式接口执行不同类型的 Python 代码。事实上，有三种类型：Python 串、Python 对象和 Python 模块。

一个 Python 串只是一个任意的 Python 语句,Python 语句通常被包括在 Python 应用程序中，或传给 eval 函数调用或 exec 语句。嵌入 API 允许执行 Python 串，仿佛将 Python 脚本的行逐次传给 Python 解释器一样。事实上，这让我们联想到嵌入式 Python 解释器与 Python 本身提供的交互作用的接口相似。

一个 Python 对象只是一个对象，它可以是一个现存对象、函数、方法或类——几乎可以是 Python 自身中的所有可被调用的对象。实际上，可以在嵌入式 Python 解释器里调用任何类型的 Python 对象，这个对象可以在执行过程中由 C 语言应用程序定义，或者由一个外部模块导入。

Python 模块是在 Python 应用程序里导入的任何外部 Python 模块。例如，可以导入 urllib 模块，然后调用已加载模块中的 urlopen 函数。

27.1.1　用 Python 嵌入 API

用 Python 嵌入 API 很简单。要对系统进行初始化，只需调用 Py_Initialize()函数来加载所需库函数，然后启动 Python 解释器即可。拥有初始化了的 Python 解释器，通过执行必要的语句，即可得到一个 Python 实例。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

可用 Py_IsInitialized()函数检查嵌入式 Python 解释器的状态。如果解释器已经建立，此函数返回值为 1(真)，否则返回 0。一旦结束，必须调用 Py_Finalize()函数关闭解释器。Py_Finalize()函数释放解释器使用过的内存，并且应阻止应用程序的剩余部分访问已分配给嵌入式解释器的内存区域来避免问题的发生。请注意，尽管如此，如果使用嵌入式实例加载别的 C 模块，在释放内存时仍然可能出现问题。

表 27-1 列出了用来从 C 语言内部与 Python 解释器通信的主要函数。

表 27-1　嵌入 API 函数

C API 调用	Python 对应的 API 调用	说　明
PyImport_ImportModule	import module	导入模块到 Python 实例中
PyImport_ReloadModule	reload(module)	重载特定模块
PyImport_GetModuleDict	sys.modules	返回一个包含已加载模块清单的字典对象
PyModule_GetDict	module.__dict__	返回给定对象的字典
PyDict_GetItemString	dict[key]	获取对应字典键的值
PyDict_SetItemString	dict[key] = value	设置字典键的值
PyDict_New	dict = {}	生成新字典对象
PyObject_GetAttrString	getattr(obj, attr)	获取给定对象的属性
PyObject_SetAttrString	setattr(obj, attr, val)	设置对象的给定属性的值
PyEval_CallObject	apply(function, args)	用 args 里的参数调用函数
PyRunString	eval(expr), exec expr	将 expr 当作 Python 语句来执行
PyRun_File	execfile(filename)	执行文件 filename
PySetProgramName	sys.argv[0] = name	改变通常在命令行上设置的 Python 程序名
PyGetProgramName	sys.argv[0]	返回由 PySetProgramName()设定的 Python 程序名
PySys_SetArgv	sys.argv = list	设置通常在命令行上提供的参数，应该有 2 个参数： argc 和 argv，分别是参数的个数和字符串列表，这两个参数都从 0 开始

对于 C/C++数据类型和 Python 对象的其他通信和集成问题，请使用第 26 章所述的函数。

还有一些函数可以用在 C 语言中来获取关于已经嵌入到 Python 解释器中实例的信息。表 27-2 包含了 C 和 C++所支持函数的清单。

表 27-2　从 C/C++内获取 Python 解释器的信息

函　数	说　明
char* Py_GetPrefix()	返回独立于平台的文件的前缀
char* Py_GetExecPrefix()	返回已安装 Python 文件的执行前缀

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

函　数	说　明
char*Py_GetPath()	返回用于搜索模块的目录清单。在 Unix 下，目录用冒号分隔开，在 Windows 下用分号(；)分隔开，而在 Mac 下用换行字符分隔开
char*Py_GetProgramFullPath()	返回 Python 解释器的完整路径
const char*Py_GetVersion()	返回 Python 解释器的版本号
const char*Py_GetPlatForm()	返回当前平台的标识符
const char*Py_GetCopyright()	返回解释器的版权声明
const char*Py_GetCompiler()	返回编译器字符串(用来建立解释器的编译器的名字和版本号)
const char*Py_GetBuilderInfo()	返回解释器的建立信息(版本号和日期)

例如，利用下面的 C 源代码，就不能打印 Python 解释器信息。

```
#include <Python.h>
int main()
{
printf("Getting Python information\n");
Py_Initialize();
if( !Py_IsInitialized() ) {
puts( "Unable to initialize Python interpreter." );
return -1;
}
printf("Prefix: %s\nExec Prefix: %s\nPython Path: %s\n",
Py_GetPrefix(),
Py_GetExecPrefix(),
Py_GetProgramFullPath());
printf("Module Path: %s\n",
Py_GetPath());
printf("Version: %s\nPlatform: %s\nCopyright: %s\n",
Py_GetVersion(),
Py_GetPlatform(),
Py_GetCopyright());
printf("Compiler String: %s\nBuild Info: %s\n",
Py_GetCompiler(),
Py_GetBuildInfo());
Py_Finalize();
return 0;
}
```

实际上，要把这段程序编译和链接成为最终可运行的版本，需要在命令行中包含目录和库文件。下一节将阐述这一过程。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

27.1.2　编译与链接

当编译 Python 时，实际上有许多包含的文件和库用来建立应用程序。当建立一个嵌入 Python 解释器的应用程序时，需要链接库的相同的列表。也就是说，需要指定附加的包含目录，并且包含基本 Python.h 头文件的目录。

尽管可以通过手工确定这些信息，但是更容易的方法是利用 distutils.sysconfig 模块来获取在 Python 建立时所生成的配置参数。需要 3 条主要信息，如下所示：

- C 编译器的选项
- 头文件的包含目录清单
- 库清单

第一条信息设置于 OPT 配置参数中。可以用以下语句抽取此信息：

```
distutils.sysconfig.get_config_var('OPT')
```

库实际上存储在几个地方。系统库存储于 SYSLIBS 中，核心 Python 解释器所需的库存储于 LIBS 中。所有 Python 模块需求的库存储于 MODLIBS 中。这些库除了存储在默认指定位置中之外也可以存储在其他位置上，所以还需先确定一下库的目录。其中，库的目录存于 LIBDIR 中。最后一个选项确切地说是一个关于库目录的单独清单。为了包含于命令行里，需要对每个目录加上前缀-L。

对于存在于 INCLDIRSTOMAKE 中的包含目录，需要作同样处理。但是加上前缀-I 就指明是包含目录，而不是库目录。

到此，您可能已完全失去了信心。但是可以通过使用如下脚本加速整个过程。尽管这个脚本设计用来生成一个 Makefile，因此大概只能保证在 Unix 下运行。如果有一个合适的 make(dmake，nmake，或者类似的)，这个脚本也应该可以在 Windows 下运行。它产生的信息也能被用在 Mac 系统下的 CodeWarrior 上。以下 Python 脚本编写了一个标准 Makefile，该标准 Makefile 用于在命令行中给出的每个嵌入式应用程序。

```
#!/usr/local/bin/python
import distutils.sysconfig
import string, sys
configopts = {}
maketemplate = """
PYLIB=%(pythonlib)s
PYINC=-I%(pythoninc)s
LIBS=%(pylibs)s
OPTS=%(pyopt)s
PROGRAMS=%(programs)s
all: $(PROGRAMS)
"""
configopts['pythonlib'] =
distutils.sysconfig.get_config_var('LIBPL') \
+'/' +\
distutils.sysconfig.get_config_var('LIBRARY')
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
configopts['pythoninc'] = ''
configopts['pylibs'] = ''
for dir in
string.split(distutils.sysconfig.get_config_var('INCLDIRSTOMAKE')):
configopts['pythoninc'] += '-I%s ' % (dir,)
for dir in
string.split(distutils.sysconfig.get_config_var('LIBDIR')):
configopts['pylibs'] += '-L%s ' % (dir,)
configopts['pylibs'] +=
distutils.sysconfig.get_config_var('MODLIBS') \
+' ' +\
distutils.sysconfig.get_config_var('LIBS') \
+' ' +\
distutils.sysconfig.get_config_var('SYSLIBS')
configopts['pyopt'] = distutils.sysconfig.get_config_var('OPT')
targets = ''
for arg in sys.argv[1:]:
targets += arg +' '
configopts['programs'] = targets
print maketemplate % configopts
for arg in sys.argv[1:]:
print "%s: %s.o\n\tgcc %s.o $(LIBS) $(PYLIB) -o %s" \
% (arg, arg, arg, arg)
print "%s.o: %s.c\n\tgcc %s.c -c $(PYINC) $(OPTS)" \
% (arg, arg, arg)
print "clean:\n\trm -f $(PROGRAMS) *.o *.pyc core"
```

例如：为了建立一个关于前一节 Python 解释器信息示例的 Makefile，需要把 C 源文件命名为 pyinfo.c，然后运行如下所示的命令：

```
$ pymkfile pyinfo >Makefile
```

生成的 Makefile 应如下所示：

```
PYLIB=/usr/local/lib/python2.1/config/libpython2.1.a
PYINC=-I-I/usr/local/include -I/usr/local/include
-I/usr/local/include/python2.1 -I/usr/local/include/python2.1
LIBS=-L/usr/local/lib -lpthread -lsocket -lnsl -ldl -lthread -lm
OPTS=-g -O2 -Wall -Wstrict-prototypes
PROGRAMS=pyinfo
all: $(PROGRAMS)
pyinfo: pyinfo.o
gcc pyinfo.o $(LIBS) $(PYLIB) -o pyinfo
pyinfo.o: pyinfo.c
gcc pyinfo.c -c $(PYINC) $(OPTS)
clean:
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
rm -f $(PROGRAMS) *.o *.pyc core
```

现在可以用 make 编译应用程序(这是 Solaris 系统上演示的情况)，如下所示：

```
$ make
gcc pyinfo.c -c -I-I/usr/local/include -I/usr/local/include
-I/usr/local/include/python2.1 -I/usr/local/include/python2.1 -g
-O2 -W
all -Wstrict-prototypes
gcc pyinfo.o -L/usr/local/lib -lpthread -lsocket -lnsl -ldl
-lthread -lm /usr/local/lib/python2.1/config/libpython2.1.a -o
pyinfo
```

接下来，运行最终可执行文件，全部完成。这样就可从以下输出看到来自 Python 解释器信息的例子，使用 sys 模块能识别出来自 Python 本身内部的大部分信息。

```
% pyinfo
Getting Python information
Prefix: /usr/local
Exec Prefix: /usr/local
Python Path: /usr/local/bin/python
Module Path:
/usr/local/lib/python2.1/:/usr/local/lib/python2.1/plat-sunos5:/usr
/local/lib/python2.1/lib-tk:/usr/local/lib/python2.1
/lib-dynload
Version: 2.1 (#1, Aug 2 2001, 12:49:29)
[GCC 3.0]
Platform: sunos5
Copyright: Copyright (c) 2001 Python Software Foundation.
All Rights Reserved.
Copyright (c) 2000 BeOpen.com.
All Rights Reserved.
Copyright (c) 1995-2001 Corporation for National Research
Initiatives.
All Rights Reserved.
Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved.
Compiler String:
[GCC 3.0]
Build Info: #1 , Aug 2 2001,12:49:29
```

请注意，因为 pymkfile 命令是用 Python 编写的，并且使用了用于建立 Python 解释器的配置选项，应能在所有平台上使用此脚本，并且能用一个适合那个平台的 Makefile 文件结束。如果要改变配置或平台,当然需要再次运行这个命令

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

27.2　用 Python 嵌入类型

用嵌入式解释器产生一个应用程序可以有多种不同的方法。已经知道了嵌入式应用程序的基本布局，从本质上而言，如下所示：

```
#include <Python.h>
...
Py_Initialize();
# Do some stuff
Py_Finalize();
```

这就是现在应了解的中间丢失的代码。

注意：
在这里显示的原则同样适用于 C++，没有必要用 C++或不同接口重新编译 Python。

27.2.1　执行 Python 字符串

利用 PyRun_SimpleString() 函数，可以执行任意 Python 字符串。以其最简单的格式，此函数接受一个字符串来进行编译和执行。可以传递任何字符串，只要这个字符串是一个合法的 Python 语句就行。例如：

```
PyRun_SimpleString("print 'Hello World\n'")
```

执行 print 语句。因为执行 Python 解释器的一个单独实例的语句，为了建立一个应用程序可以添加更多的语句。如下所示：

```
#include <Python.h>
int main()
{
printf("String execution\n");
Py_Initialize();
PyRun_SimpleString("import string");
PyRun_SimpleString("words = string.split('rod jane freddy')");
PyRun_SimpleString("print string.join(words,', ')");
Py_Finalize();
return 0;
}
```

如果编译并执行这个应用程序，可以得到如下所示的输出：

```
String execution
rod, jane, freddy
```

这并非是最精彩的应用程序，但它显示了可以执行来自 C 语言的任意字符串。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

27.2.2 用 Python 对象工作

前面的例子是对典型嵌入式应用程序的陈述。因为刚刚起步，可能还没有真正了解 Python 脚本—— 逐行提供给嵌入式 Python 解释器的脚本很可能只能作为一个单独的 Python 应用程序。事实上，可以把那些每个 PyPun-SimpleString()函数调用提供的代码放置到一个独立的文件里，然后用 PyRun_File()函数执行整个模块。

大部分嵌入式应用程序需要在 C 和 Python 之间交换信息。其实在第 26 章已经接触到了关于创建扩展的一些例子。当进行嵌入操作时，往往需要将 Python 函数转换为 C 应用程序能够理解的形式。一般来说，嵌入更多地依赖于从 Python 到 C 的转换，而扩展依赖于从 C 到 Python 的转换。

为了演示的目的，使用一个称作 reverse 的模块。reverse 模块采用各种参数并且返回一个逻辑逆向值。这是一个数的逆向值、一个逆向顺序的字符串、一个逆向顺序的列表，和把值映射为键或相反映射的一个字典。模块本身如下所示：

```
def rstring(s):
i = len(s)-1
t = ''
while(i > -1):
t += s[i]
i -= 1
return t
def rnum(i):
return 1.0/float(i)
def rlist(l):
l.reverse()
return l
def rdict(d):
e = {}
for k in d.keys():
e[d[k]] = k
return e
```

所访问的对象无论是一个函数还是一个给定模块内的变量，几乎可以使用同样的方式利用 PyObject_GetAttrString()函数访问任何模块的任何对象。PyObject_GetAttrString()函数通过访问一个特定对象的属性表得到一个对象。例如，当访问函数 rstring()时，需要访问 reverse 模块的属性表中的 rstring 对象。用此作为一个例子来解释如何访问 C 应用程序的对象和调用 C 应用程序的函数。

获取一个函数调用的返回值，事实上包括 5 个阶段，可总结如下：

(1) 导入包含要调用函数的模块(如果有必要)，并获取输出。

(2) 通过访问模块的属性获取要调用的对象。

(3) 把要作为参数提供给函数的所有 C 变量建立为一个参数对象。

(4) 调用函数，给其提供参数。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(5) 把返回的参数对象转换为 C 变量。

要导入模块(阶段 1)，请使用 PyImport_ImportModule() 函数。如下所示：

```
PyObject *mymod = NULL;
mymod = PyImport_ImportModule("reverse");
```

为了得到想要调用的函数对象(阶段 2)，请使用 PyObject_GetAttrString()函数。 例如，为了保存 rstring 函数，请使用如下所示的语句：

```
PyObject *strfunc = NULL;
strfunc = PyObject_GetAttrString(mymod, "rstring");
```

请注意，如果这时访问的是一个变量而不是一个函数，可以把变量转换成为一个合适的 C 语言结构，并利用 PyArg_Parse()函数来打印它或使用它。

要建立函数的参数(阶段 3)，请使用 Py_BuildValue()函数。对于 rstring 函数，需要提供一个串。当然，既然把这个字符串提供给一个函数，需要准确地提供值的一个元组。如下所示：

```
PyObject *strargs = NULL;
strargs = Py_BuildValue("(s)", "Hello World");
```

关于转换格式的更多信息，请查看第 26 章“把 C 变量转换到 Python”这一节。

实际上要调用 rstring() 函数(阶段 4)、调用 PyEval_CallObject()，还需要一个变量来保存返回值，如下面例子中的 Strret：

```
PyObject *strret = NULL;
strret = PyEval_CallObject(strfunc, strargs);
```

最后阶段(阶段 5)获取返回值。在这种情况下，字符串，再次被转换为一个 C 变量并且显示出来。 PyEval_CallObject()的返回值是个 Python 对象，所以可以利用 PyArg_Parse()把信息抽取回为 C 变量值。如下所示：

```
char *cstrret = NULL;
PyArg_Parse(strret, "s", &cstrret);
printf("Reversed string: %s", cstrret);
```

关于把数据从 Python 对象转换为 C/C++类型和 / 或把 C/C++类型转换为 Python 对象的更多信息，请参见第 26 章的“数据转换”部分。

综上所述，可以得到如下所示的 C 源程序：

```
#include <Python.h>
int main()
{
PyObject *strret, *mymod, *strfunc, *strargs;
char *cstrret;
Py_Initialize();
mymod = PyImport_ImportModule("reverse");
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
strfunc = PyObject_GetAttrString(mymod, "rstring");
strargs = Py_BuildValue("(s)", "Hello World");
strret = PyEval_CallObject(strfunc, strargs);
PyArg_Parse(strret, "s", &cstrret);
printf("Reversed string: %s\n", cstrret);
Py_Finalize();
return 0;
}
```

如果建立并且执行了这个应用程序，将得到如下所示的输出：

```
Reversed string: dlroW olleH
```

成功了！

用本节学到的技巧并结合第 26 章描述的不同 Py_BuildValue()选项，可以进行关于数、列表和字典的同样过程。

注意：

请记住就像扩展一样，如果把 Python 对象重新用作新值，必须对引用计数器作递减操作以确保从内存中释放掉内存占用空间和引用。尽管在以上各例中，这并不必要，因为仅仅在 main()函数内部进行开发。如果将 Python 实例放到另外一个函数或者使用来自嵌入式 Python 实例的外部 C 模块，就必须应用以上原则。参看第 26 章“引用计数”这一节可得到更多信息。

27.2.3　利用 Python 类

从 C 中访问一个 Python 类听起来很复杂，但事实上是非常简单的。如果已经明白了前面的例子，就应知道怎么从 C 应用程序内访问对象。所需要做的就是进行同样的操作。当类被定义在模块内部时，为了访问类本身，需访问在模块内定义的类名，并加载模块，然后用 PyObject_GetAttrString()得到类名。

接下来为创建一个对象实例，需调用此类，提供所有初始化参数。为了执行一个对象方法，需要再用一次 PyObject_GetAttrString()函数，以对象实例为基础得到方法对象。

例如，给定以下模块：

```
class celsius:
def_init_(self, degrees):
self.degrees = degrees
def farenheit(self):
return ((self.degrees*9.0)/5.0)+32.0
```

在 Python 内部，可以这样使用 celsius 类。如下所示：

```
Python 2.1 (#1, Aug 2 2001, 12:49:29)
[GCC 3.0] on sunos5
Type "copyright", "credits" or "license" for more information.
>>> import celsius
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
>>> temp = celsius.celsius(100)
>>> temp.farenheit()
212.0
```

如果按着此顺序，就应该知道在 C 中也需跟随这个基本过程。首先，导入模块；然后在 celsius 模块内找到 celsius 类，创建一个 celsius 类的新实例；最后调用 temp 对象的 farenheit 方法，其中，temp 对象是 celsius 类的一个实例。

此 C 代码显示同样的过程，只不过这次是 C 语言，使用的是以前用过的相同技巧。然而，C 语言的这个例子也包含了检错代码，以确保当出错时程序可以及时地终止。此例在创建时也递减关于不同临时对象的引用计数。如下所示：

```
#include <Python.h>
/* Create a function to handle errors when they occur */
void error(char errstring)
{
printf("%s\n",errstring);
exit(1);
}
int main()
{
/* Set up the variables to hold methods, functions and class
instances. farenheit will hold our return value */
PyObject *ret, *mymod, *class, *method, *args, *object;
float farenheit;
Py_Initialize();
/* Load our module */
mymod = PyImport_ImportModule("celsius");
/* If we dont get a Python object back there was a problem */
if (mymod == NULL)
error("Can't open module");
/* Find the class */
class = PyObject_GetAttrString(mymod, "celsius");
/* If found the class we can dump mymod, since we wont use it
again */
Py_DECREF(mymod);
/* Check to make sure we got an object back */
if (class == NULL)
{
Py_DECREF(class);
error("Can't find class");
}
/* Build the argument call to our class - these are the arguments
that will be supplied when the object is created */
args = Py_BuildValue("(f)", 100.0);
if (args == NULL)
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
{
Py_DECREF(args);
error("Can't build argument list for class instance");
}
/* Create a new instance of our class by calling the class
with our argument list */
object = PyEval_CallObject(class, args);
if (object == NULL)
{
Py_DECREF(object);
error("Can't create object instance");
}
/* Decrement the argument counter as we'll be using this again */
Py_DECREF(args);
/* Get the object method - note we use the object as the object
from which we access the attribute by name, not the class */
method = PyObject_GetAttrString(object, "farenheit");
if (method == NULL)
{
Py_DECREF(method);
error("Can't find method");
}
/* Decrement the counter for our object, since we now just need
the method reference */
Py_DECREF(object);
/* Build our argument list - an empty tuple because there aren't
Composite Default screenany arguments */
args = Py_BuildValue("()");
if (args == NULL)
{
Py_DECREF(args);
error("Can't build argument list for method call");
}
/* Call our object method with arguments */
ret = PyEval_CallObject(method,args);
if (ret == NULL)
{
Py_DECREF(ret);
error("Couldn't call method");
}
/* Convert the return value back into a C variable and display it
*/
PyArg_Parse(ret, "f", &farenheit);
printf("Farenheit: %f\n", farenheit);
/* Kill the remaining objects we don't need */
```

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw


```
Py_DECREF(method);
Py_DECREF(ret);
/* Close off the interpreter and terminate */
Py_Finalize();
return 0;
}
```

如果编译并执行前面的代码，得到如下所示的结果：

```
$ exclass
Farenheit: 212.000000
```

成功了！

27.3　下一步的工作

如果已经进行了上面的实例，应该能结合在这一章演示的技巧和第 26 章描述的技巧与函数引用来用 Python 编写 C 应用程序。既然已经掌握了访问整个 Python 库的方法，那么几乎可以做任何事了！

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

附录A　Python库指南

本附录包含了组成标准 Python 库的所有模块清单，并且本附录之中的所有模块清单包含在 Python 所有版本的基本配置中。模块清单在表 A-2 中给出，但首先从关于模块分类说明的表 A-1 开始。

表 A-1　模块分类系统

模块的适用对象	说　明
Runtime	提供在 Python 和底层系统与环境之间的接口的服务
String	字符串操作和处理
Misc	其他服务
Generic	可能运行在所有平台上的普通操作系统服务
Optional	可选择的操作系统服务，通常特定于平台
Unix	特定于 Unix 的服务
Internet	特定于因特网的服务
Markup	结构化标记处理工具
Multimedia	产生声音和其他多媒体数据的模块
Crypto	用于编码和解码信息的加密的模块
Python	提供关于 Python 编程的服务和信息的模块
Restrict	在限定或限制的环境内产生和执行 Python 脚本的模块
SGI	特定于 SGI(以前的 Silicon Graphics，Inc.)平台的模块
SunOS	特定于 Sun OS (Solaris)平台的模块
Windows	特定于 Windows 平台的模块
Mac OS	特定于 Mac OS 平台的模块

表 A-2　Python 标准模块库

模　块	模块的适用对象	说　明	参看章节
builtin	Runtime	Python 解释器的内置(内部)函数集的模块引用	8
main	Runtime	保存主要脚本的模块名	5
AE	Mac OS	Apple Events 工具箱的接口	
aepack	Mac OS	压缩信息，在 Apple Events 结构和 Python 数据类型之间转换	

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

模　　块	模块的适用对象	说　　明	参看章节
aetypes	Mac OS	Apple Evernt 对象模型的 Python 接口	
aifc	Multimedia	读写 AIFF(Audio Interchange File Format)和 AIFC(compressed AIFF)音频文件	14
al	SGI	SGI 平台上的音频库函数的 接口(Irix)	
AL	SGI	al 模块使用的常量	
anydbm	Optional	用 DBM 格式化的数据库的*dbm 模块序列的简单接口	12
array	Misc	有效存储定长数字数组的类	10
asyncore	Internet	同步套接字处理服务的基类	13，21
atexit	Runtime	用于脚本中断时提供清空服务的注册函数	9
audioop	Multimedia	提供操作原始音频文件的函数	14
base64	Internet	MIME 文件的编码/解码例程	20
BaseHTTPServer	Internet	基本 HTTP 服务器类	13,21
Bastion	Restrict	提供对特定对象的限定访问	
binascii	Internet	用于在 ASCII 和二进制数据流之间转换的例程	13
binhex	Internet	Mac based binhex 格式的编码/解码例程	13
bisect	Misc	二进制搜索的数组二分法算法	
bsddb	Optional	Berkely DB 数据库(library)的接口	12
calendar	Misc	确定日历信息(天、周、月等)的函数	10
cd	SGI	SGI 系统的 CD-ROM 接口	
cgi	Internet	通过 Web 公共网关接口(Common Gateway Interface)发送的解码信息	19
CGIHTTPServer	Internet	提供一个能运行 CGI 脚本的 HTTP 服务器的请求处理程序	13,21
chunk	Multimedia	读 IFF 块的模块。(IFF 是一个标记过的二进制文件格式，原本用在 Amiga 的图形和文件上，它的子文件格式包括用在 Windows 下的 WAV 文件格式	14
Cm	Mac OS	Mac OS 组件管理程序的接口	
cmath	Misc	复数的数学函数	10
cmd	Misc	建立面向行的命令解释器的系统	
code	Runtime	交互式 Python 解释器的基类	23
codecs	String	编码与解码数据和流	21
codeop	Runtime	编译 Python 任意码	

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

模　　块	模块的适用对象	说　　明	参看章节
ColorPicker	Mac OS	用于选择颜色的颜色选择器接口	
colorsys	Multimedi-a	在 RGB 和其他颜色系统间转换的转换函数	14
commands	Unix	运行外部命令的实用函数，包括获取输出的工具	
compileall	Python	用于 1 字节编译在一个目录树中的所有 Python 源文件的工具	
ConfigParser	Misc	配置文件的分析程序	
Cookie	Internet	用于支持 HTTP 状态管理(cookies)	19
copy	Runtime	浅拷贝和深拷贝操作	10,12
copy_reg	Runtime	注册 pickle 支持函数	
cPickle	Runtime	更新版本的 pickle，不可子分类	12
crypt	Unix	crypt()函数，用于检测 Unix 口令	9
cStringIO	String	StringIO 的更新版本，但不可子分类	
ctb	Mac OS	通信工具箱支持	
Ctl	Mac OS	控制管理器(Control Manager)的接口	
curses	Generic	curses 终端接口库的接口	
curses.ascii	Generic	ASCII 字符的常量和成员关系设置函数	
curses.panel	Generic	给终端功能提供面板的 curses 基本模块的扩展	
curses.textpad	Generic	在 curses 窗口编辑的类似 Emacs 的输入	
curses.wrapper	Generic	curses 程序中使用的终端配置包装器	
dbhash	Optional	BSD 数据库库(library)的散列(字典)部分的接口	12
dbm	Unix	ndbm 散列(字典)数据库系统的接口	12
DEVICE	SGI	gl 模块用到的常量	
difflib	String	计算对象间差异的函数	
dircache	Generic	提供了一个基于高速缓存的目录清单机制。与 glob 比较，它对于大的目录清单有用	
dis	Python	Python 字节码的反汇编程序	23
dl	Unix	Unix(和类似 Unix 的操作系统，如 BeOS)下的动态库系统的接口，调用在共享对象中发现的 C 函数	
Dlg	Mac OS	Mac OS 对话管理器的接口	
doctest	Misc	在 docstrings 中验证示例的帧	25
dumbdbm	Optional	完全可以在 Python 上实现的可移植(跨平台)的类似 DBM 接口的实现	12

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

模　　块	模块的适用对象	说　　明	参看章节
EasyDialogs	Mac OS	基本的 Macintosh 对话	
errno	Generic	用于 errno 错误数的符号和文本常量	
Evt	Mac OS	事件管理器的接口	
fcntl	Unix	fcntl()和 ioctl()系统调用	10,11
filecmp	Generic	有效的文件比较例程	
fileinput	Misc	用于迭代多输入流的命令行的类似 Perl 的系统	
findertools	Mac OS	环绕 Finder's Apple 事件接口的包装器	
fl	SGI	关于 GUI 应用程序的 SGI FORMS 库的接口	
FL	SGI	fl 模块使用的常量	
flp	SGI	用于加载所存储的 FORMS 设计的函数	
Fm	Mac OS	字体管理器的接口	
fm	SGI	SGI 工作站的字体管理器接口	
fnmatch	Generic	提供用于匹配文件名的 Unix shell 类型模式	
formatter	Internet	一般的输出格式器和设备接口	20
fpectl	Runtime	浮点异常处理	
fpformat	String	浮点格式化函数	
FrameWork	Mac OS	交互式应用程序帧	
ftplib	Internet	FTP 协议客户机	13
gc	Runtime	循环检测无用单元收集器的接口	
gdbm	Unix	GNU dbm 数据库系统的接口	12
getopt	Generic	解释命令行选项的语法分析程序	9
getpass	Generic	获得用户名信息，提供交互接收用户口令的可移植的和安全的方式。只适用于在命令行中使用	9
gettext	Generic	多语种国际化服务	
gl	SGI	SGI 图形库的接口	
GL	SGI	gl 模块用到的常量	
glob	Generic	返回文件名的列表，使用 Unix shell 类型模式匹配系统	11
gopherlib	Internet	Gopher 协议客户机	13
grp	Unix	Unix/ect/group 文件的接口	9
gzip	Optional	通过文件对象执行 gzip 压缩和解压缩的接口	
htmlentitydefs	Markup	HTML 一般实体的定义	20
htmllib	Markup	HTML 文档的语法分析程序	20

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

模　　块	模块的适用对象	说　　明	参看章节
httplib	Internet	HTTP 协议客户机	13
ic	Mac OS	访问 Internet Config 配置应用程序	
imageop	Multimedia	支持原始图像的基本操作函数	14
imaplib	Internet	IMAP4(电子邮件)协议客户机	13
imgfile	SGI	用于支持 SGI imglib 文件	
imghdr	Multimedia	通过检查文件头信息确定图像类型	14
imp	Runtime	访问导入语句的实现	
inspect	Runtime	从活动对象中抽取信息和源代码	
jpeg	Multimedia	用 JPEG 格式读写文件	14
keyword 是	Python	确定一个字符串是否是 Python 中的关键字	
linecache	Runtime	支持随机访问文本文件的行	
List	Mac OS	Mac OS 列表管理器的接口	
locale	Generic	国际化服务	
mac	Mac OS	关于 os 模块 的特定于 Mac OS 的函数	
macdnr	Mac OS	Macintosh 域名分解器的接口	
macfs	Mac OS	用于支持 FSSpec ,管理器的别名，寻找器别名和标准文件包	
MacOS	Mac OS	访问特定于 Mac OS 的解释器的特征	
macostools	Mac OS	Mac OS 下的文件操作的例程	
macpath	Mac OS	Mac OS 路径操作函数	
macspeech	Mac OS	Macintosh Speech 管理器的接口	
mactcp	Mac OS	MacTCP　TCP/IP 系统的接口	
mailbox	Internet	解释各种邮箱格式	13
mailcap	Internet	处理 Mailcap 文件的函数	13
marshal	Runtime	将 Python 对象转换为字节流，或把字节流转换为 Python 对象	12
math	Misc	数学函数	10
md5	Crypto	编码 RSA 的 MD5 的消息摘要算法	
Menu	Mac OS	Mac OS 菜单管理器的接口	
mhlib	Internet	操纵 MH 格式化邮箱	13
mimetools	Internet	分析 MIME 样式的消息体的工具	
mimetypes	Internet	把文件扩展名映射到 MIME 类型	13
MimeWriter	Internet	普通 MIME 文件编写器	

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

模　块	模块的适用对象	说　明	参看章节
mimify	Internet	把电子邮件消息转换为 MIME 类型，或把 MIME 类型转换为电子邮件消息	13
MiniAEFrame	Mac OS	使得能够把自己的脚本设置为开放式脚本结构(OSA)服务器(可以应答 Apple Events)	
mmap	Optional	Unix 和 Windows 的内存映射文件的接口	
mpz	Crypto	任意精度算术的 GNU MP 库的接口	
msvcrt	Windows	MS VC++运行时库的有用的其他例程	
multifile	Internet	支持读/写利用多部件扩展解了码的文件，包括 MIME 文件和多部件 HTTP 请求	13
mutex	Generic	支持手工互斥锁定和排队系统	
netrc	Internet	解释 .netrc 文件的内容	
new	Runtime	对运行时实现对象所创建的接口	12
nis	Unix	Unix 网络信息系统(NIS)数据库的接口	
nntplib	Internet	NNTP 协议客户机	13
operator	Runtime	像内置函数一样的所有 Python 的标准运算符	10,12,13
os	Generic	杂项(多目标)OS 接口	9
os.path	Generic	公共路径名操作	9,11
parser	Python	Python 源代码的访问语法分析树	
pickle	Runtime	把 Python 对象转换为字节流，或把字节流转换为 Python 对象	12
pipes	Unix	Unix shell 管道线的 Python 接口	
popen2	Generic	通过标准 I/O 流从子进程的读接口和写到子进程的接口	9
poplib	Internet	POP3(电子邮件)协议客户机	13
posix	Unix	公共 POSIX 系统调用，一般在合适平台上由 os 模块自动导入	9,10,12
posixfile	Unix	可锁定文件对象接口	12
pprint	Runtime	数据对象的完美打印机	
pty	Unix	SGI/Linux 所处理的伪终端	
pwd	Unix	Unix/etc/password 文件的接口	9
py	Python	编译 Python 源文件为字节码文件	
pyclbr	Python	支持 Python 类浏览程序的信息提取	
Qd	Mac OS	Mac OS QuickDraw 工具箱的接口	

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

模　　块	模块的适用对象	说　　明	参看章节
Qt	Mac OS	Mac OS QuickTime 工具箱的接口	
Queue	Optional	同步队列类	
quopri	Internet	利用 MIME 引用可打印编码系统的编码/解码文件	13
random	Misc	生成各种不同的普通分布的伪随机数	10
re	String	Perl 样式的正则表达式匹配/替换例程	10
readline	Optional	GNU 读取命令行的基于行的接口	10
repr	Rumtime	支持限制长度的 repr()函数的另外选择	
Res	Mac OS	Mac OS 资源管理器和句柄(Handle)的接口	
resource	Unix	确定当前进程的资源使用信息	
rexec	Restrict	基础限制的执行帧	
rfc822	Internet	分析 RFC 822 类型的电子邮件头	
rgbimg	Multimedia	读和写 SGI RGB 格式的图像文件	13
rlcompleter	Optional	提供利用 GNU 阅读行库的 Python 标识符使用的完整系统	
robotparser	Internet	分析 Web 服务器 robots.tex 文件，接着建立一组表示信息的对象	
rotor	Crypto	类似 Enigma 的加密和解密	
sched	Generic	一般目的的事件调度程序	
Scrap	Mac os	Mac OS Scrap 管理器的接口	
select	Optional	多 I/O 流的多路复用调度程序	
sgmllib	Markup	SGML 文件的语法分析程序	20
sha	Crypto	支持 NIST 的安全散列算法(SHA)的函数	
shelve	Runtime	提供用于在外部文件中存储 Python 对象的函数(参见 marshal 和 pickle 模块)	12
shlex	Misc	类似 Unix shell 语言的简单词法分析	
shutil	Generic	高级别文件操作，包括复制	11
signal	Optional	Unix/POSIX 类型信号的接口	9
SimpleHTTPServer	Internet	简单 HTTP 服务器类	13,21
site	Rntime	支持关于引用特定于站点的模块的一个机制	
smtplib	Internet	SMTP 协议客户机	13
Snd	Mac os	Mac OS 声音管理器的接口	
sndhdr	Multimedia	通过检查文件头确定声音文件的格式	14

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

模　　块	模块的适用对象	说　　明	参看章节
socket	Optional	网络套接字的低层接口	13
Socketserver	Internet	用来创建基于套接字的网络服务器的帧	13,21
stat	Generic	用来解释 os.stat()、os.lstat()和 os.fstat()结果的函数和常量	11
statcache	Generic	获得文件统计并记忆结果	11
statvfs	Generic	用来解释 os.statvfs()结果的常量	
string	String	操纵和解释文本串的函数	10
StringIO	String	把串当作文件读写	
struct	String	压缩/解压缩二进制数据，尤其是基于 C 的结构	12
sunau	Multimedia	提供对 Sun AU 声音格式的接口	14
sunaudiodev	Sun	Sun 音频控制器	14
symbol	Python	代表语言分析程序树的内在节点的常量	
sys	Runtime	访问特定于系统的参数和函数	9
syslog	Unix	Unix syslog 库例程的接口	
tabnanny	Python	用于检测一个目录树下的 Python 源文件中的空格区域相关问题的工具	
TE	Mac OS	Mac OS 文本编辑类的接口	
telnetlib	Internet	Telnet 客户机类	13
tempfile	Generic	产生临时文件名	
termios	Unix	POSIX 类型的 tty 控制	
TERMIOS	Unix	使用 termios 模块所需的符号常量	
thread	Optional	主机 OS 线程生成函数的低层接口	9
threading	Optional	基本 thread 模块的高层接口	9
time	Generic	获取、转换和操纵时间和数据信息	10
token	Python	代表语言分析程序树上的终端节点的常量	
tokenize	Python	Python 源代码的词法扫描程序	
traceback	Runtime	打印或检索脚本的 Python 执行树的回跟踪	24
tty	Unix	执行一般终端控制操作的实用函数	
types	Runtime	所有内置类型的名字	23
unicodedata	String	访问 Unicode 数据库	10
unittest	Misc	为 Python 测试帧的单元	
urllib	Internet	打开或下载任意 URL	13,19

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

模　　块	模块的适用对象	说　　明	参看章节
urllib2	Internet	使用各种协议打开 URL 的可扩充库	13,19
urlparse	Internet	把 URL 分析为其组件部分	19
user	Runtime	引用特定用户模块的标准方法	
UserDict	Runtime	字典对象的类包装器	
UserList	Runtime	列表对象的类包装器	
UserString	Runtime	串对象的类包装器	
uu	Internet	编码和解码 uuencode 格式的文件	13
warnings	Runtime	发出警告消息，控制它们的配置	24
waste	Mac OS	Mac OS 世界认知类型化文本引擎 WorldScript-Aware StyledTextEngine (WASTE) 的接口	
wave	Multimedia	WAV 声音格式的接口	14
weakref	Runtime	弱引用和弱字典的支持	
webbrowser	Internet	Web 浏览器的易用控制器	
whichdb	Optional	确定哪种 DBM 类型模块创建了给定数据库	12
whrandom	Misc	浮点伪随机数生成器	10
Win	Mac OS	Mac OS 窗口管理器的接口	
winreg	Windows	用于操纵 Windows 注册的例程	14
winsound	Windows	对 Windows 声音播放工具的访问	
xdrlib	Internet	外部数据表征(XDR)系统的编码器和解码器(用在 C,尤其是 RPC 系统)	13
xml.dom	Markup	Python 的文档对象模型 API	20
xml.dom.minidom	Markup	轻型文档对象模型(DOM) 工具	20
xml.dom.pulldom	Markup	对从 SAX 事件建立部分 DOM 树的支持	20
xml.parsers.expat	Markup	Expat 无效 XML 语法分析程序的接口	
xml.sax	Markup	包含 SAX2 基类和便利函数的包	20
xml.sax.handler	Markup	SAX 事件处理程序的基类	20
xml.sax.saxutils	Markup	SAX 使用的便利函数和类	20
xml.sax.Xmlreader	Markup	符合 SAX 标准的 XML 语法分析程序必须执行的接口	20
xmllib	Markup	XML 文档的语法分析程序	20
xreadlines	Misc	文件数据行上的有效迭代	
zipfile	Optional	读写 ZIP 格式存档文件	
zlib	Optional	与 gzip 兼容的压缩和解压缩例程的低层接口	

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

附录B Python资源

像现代大多数脚本语言一样——确切像所有现代的开放资源开发项目，Python 主要受遍布因特网之上的编程集体支持。主网站是 www.python.org，此网站提供对 Python 语言的最全面的指导和联网或单机的有效资源。

B.1 Web 资源

Python 的成功主要源于用户，他们反馈信息给 Python 开发小组；对 Python 语义的新特性及变化提出宝贵的建议，帮助标识和修正 bug。尽管现在很多人致力于这种语言的开发，但来自用户的反馈信息则最后送至 Python 语言的设计和开发者 Guido Von Rossum 处。大多反馈信息是在因特网上通过邮件、新闻组或各种 Python 网站交换的。当然，因为因特网有很强的流动性，本附录不可能对所有有效信息加以说明。但同其他值得专门提及的网站一样,Python 主网站仍有一些精选区。下面所列条目可作为非常有用的资源的一个展示。

www.python.org 此网站对 Python 语言的一切信息进行了集中,且由 Python 软件小组(Python Software Activity ,PSA)的自愿者们管理。他们中许多人是致力于使脚本语言 Python 取得经久不衰的成功的狂热追随者。此网站也包含所有可在线搜索和下载的在线文档资料。此网站上拥有可用于个人或用于办公的 HTML、Postscript 和 Acrobat PDF 版本的文档资料。

www.sourceforge.net PythonLabs，负责 Python 开发的开发小组，于 2000 年 10 月加入到 SourceForge 。SourceForge 是一个免费软件机构。 尽管大量关于 Python 的开发和信息仍在 Python 主网站上发布，但关于 Python 的最新版本 、Python 当前开发版本的快讯，及其文档资料也出现在 SourceForge 上。

www.python.org/psa Python 软件小组(PSA)的主页。PSA 聚集资金，资助新项目的开发，举行会议，管理支持 Python 开发基于因特网的服务。PSA 向需求 Python 深层开发的会员收取会员费。写这本书时，PSA 会员费，个人是 50 美元，组织机构是 500 美元。

www.python.org/sigs 各种特别兴趣小组(SIG)是 Python 开发和改进的重要部分。虽然 SIG 能力有限，但不同的开发项目可在同时进行。表 B-1 列出在 2001 年 1 月初就存在的 SIG。若欲获得更新列表，请点击 URL www.python.org/sigs 。

表 B-1 Python PSA 特别兴趣小组(PSA,Special Interest Groups)

Name/Home	描 述	协 调 人	期 限
C++-sig	C++绑定开发	Geoffrey Furnish	2002.6
Catalog-sig	Python 软件分类，处理 Python 的模块、包和其他资源的分类	Andrew Kuchling	2002.6

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

(续表)

Name/Home	描　述	协 调 人	期　限
db-sig	数据库(目前正在创建公共列表数据库 API)	Andrew Kuchling	2002.6
distutils-sig	分布利用	Greg Ward	2002.6
do-sig	分布式对象技术	David Arnold	2002.6
doc-sig	文档资料(涵盖工具和内容两部分)	Fred Drake	2002.6
edu-sig	Python 教育(传授 Python 词汇，致力于把 Python 提升为适用于所有人的编程语言)	Guido van Rossum	2002.12
i18n-sig	国际化和区域化,包括 Unicode	Andy Robinson	2002.6
image-sig	图像处理	Fredrik Lundh	2002.6
import-sig	导入结构重设计	Gordon McMillan	2002.6
meta-sig	关于特别兴趣小组的 SIG	Guido van Rossum	无
plot-sig	规划和绘图	David Ascher	2002.6
pythonmac-sig	用在 Apple Macintosh 上的 Python	Jack Jansen	无
types-sig	静态键入法设计	Paul Prescod	2002.6
xml-sig	所有不同形式的 XML 处理	Andrew kuchling	2002.6

每一个 SIG 由邮件列表支持。加入 SIG 的任何人都可以参与到讨论中来，甚至可以涉足 Python 开发过程中特殊领域的开发。

www.python.org/cgi-bin/todo.py　指向 Python CGI 脚本的 URL，提供当前所有需求且急需完成的 Python 开发的项目清单。

www.jython.org　此网站是 Jython 项目的主页。它含有可下载的组件、文档、样本、和关于 Jython 及其用在何处的进一步信息指示。

www.mailman.org　Mailman 软件的主页。Mailman 完全用 Python 编写，具有发邮件列表管理器的所有普通特性，包括电子邮件申请、分类和申请人的安全验证。另外，它启动网络前端，允许申请人和发邮件列表管理者本人进入已有的列表。

www.zope.org　Zope 是网上公告和内容管理系统，允许多人管理网站内容。除了基本内容外，Zope 也提供导入外部资源信息的管道(称为工厂)。外部资源包括传统数据库、不太便利的 POP、IMAP、NNTP 源程序。

starship.python.net　Python Starship 网站是作为 www.pyrhon.org 的一个扩展而建立的，提供 Python 项目团队合作的中介场所。Starship Python 由 Digital Creations 支持。Digital Creations 是 Zope 的开发者，现在支持 Python 自身的开发。

www.pythonjournal.com　Python 月刊，是程序员的在线杂志。首刊已取得巨大成功，第二刊于 1999 年初出版。可是，以后没再出版。此网站集精力于当日资源上，如 O'Reilly Python 的更新及建设 LinuxProgramming.com 这样的站点。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

python.ora.com　O'Reilly Python 的成果网站。包括 Python 书刊信息，定期更新新闻和关于Python 开发和编程方面的文章。

www.pythonware.com　Python 的出现引起了许多商业公司将目标瞄准于提供更稳定更具有吸引力的 Python 解决方案。SecretLabs 就是这样的一家公司，Python Ware 是其网站。此公司当前正在开发称为 PythonWorks 的 Python RAD(快速应用程序开发)环境，此环境基于 Python 的核心语言之上，具有主要用于接口、图形、图像处理而开发的特殊扩展库。

www.activestate.com　ActiveState 可能因生产 Perl 的最初 Windows 端口而出名。过去几年，他们集中于扩展产品范围。这些产品不仅包括适用于 Windows 的扩展 Perl 安装程序(附有文档资料和特性)，也包括适用于 Linux 和 Solaris 的相似产品。

最近，ActiveState 获得许可能够发布关于 Python 的相似产品。除了 ActivePython 产品提供的核心性能外，ActiveState 也从事于关于 Python 的开发环境、调试器和其他适用于专用平台的扩展库。

非常有趣的是，ActiveState 和微软(资助 ActiveState)共同生产 VisualPerl 和 VisualPython 产品。这些产品允许 VisualStudio 开发环境的用户访问相同集成的开发环境、调试器和适用于 C、C++和 Visual Basic 开发人员的其他工具。

www.cdrom.com　Walnut Creek，是 www.cdrom.com 的运营者，生产出许多免费 CD，包括 Python 工具 CD。Python 工具 CD 不仅包括带有所有有效模块的不同平台版本的 Python 语言，还包括从 Python 主因特网网站上能访问到的海量在线文档资料。如果不想从网站上下载信息，这个 CD 是必须拥有的一张 CD。

B.2　邮件、新闻组和发邮件列表的资源

大多数关于 Python 的一般性的经常性的讨论和支持是由各种新闻组和发邮件列表处理的。除了本节所列之外，您还会发现不同 Python SIG 所讨论的特定论题。请参见上节当中关于特定 SIG 的信息。

comp.lang.python　它是关于 Python 一般性讨论的重要新闻组。任一感兴趣者都可以加入。可以在这里提出问题，而由其他成员回答。显然新闻组将产生大量的通信信息，但信息的内容是高层次的。若不想直接访问 Usenet 新闻组而仅愿通过邮件获得信息，也可使用 Usenet 新闻组。

发邮件列表本身是由 Mailman 维护的。Mailman 是由 GUN 认可的且完全是用 Python 编写的发邮件列表系统。若想加入新闻组可以访问 www.python.org/mailman/listinfo/python-list，或发送含有“申请”的电子邮件给 python-list-request@python.org。即使不是用户，也可通过发送电子邮件给 python-list@python.org 而与发邮件列表取得联系。

comp.lang.python.announce　窄带宽新闻组仅用来发布关于 Python 的声明而不进行进一步的讨论。虽然有人可以邮寄给新闻组发表有用的言论；但毕竟新闻组的工作量减轻了。

再次说明，可以通过网页 www.python.org/mailman/listinfo/python-announce-list 或发送含有“申请”的电子邮件给 python-announce-list-request@python.org 申请加入新闻组。如果不是作为用户申请，可发送电子邮件给 python-announce-list@pyhton.org 而与投递组取得联系。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

python-help@python.org　还以 help@python.org 而出名。实际上，邮件账户由一组 Python 志愿者处理。他们在封闭环境中回答用户的问题。邮至此地址的邮件会被送至专家组，但可通过电子邮件作进一步的交流。读邮件或访问邮件的归档文件是不可能的。

tutor@python.org　与 help@python.org 地址相似。邮件账户由打算学习 Python 编程的用户使用。与 help@python.org 不同之处是，tutor@python.org 是允许充分讨论的投递组，可以通过电子邮件交流思想，但此过程仍可被普通公众看到。通过网站 www.python. org/ mailman/ listinfo/tutor 可以加入此发邮件列表。

jpython-interest@python.org　如果对 Jython 产品(前面称为 JPython)函件投递专业组，完全用 Java 语言编写的 Python 产品感兴趣，此投递组将有助于使用 Jython 包，也能够使您可以与其他用户讨论问题。关于如何加入此发邮件列表的更多信息和细节，可访问www.python. org/mailman / listinfo/jpython-interest。

B.3　在线文档资料

Python 上网站有个文档资料区(www.python.org/doc/)，此处可以访问所有由 Fred Drake 编写并管理的 Python 在线文档资料。Fred Drake 是 PythonLab 小组的重要开发人员之一。可以下载 HTML、PostScript、Acrobat PDF 和 LaTeX 格式的在线文档资料。在线文档资料被分为表 B-2 中列出的几部分。

如果使用所有文档资料来创建网站，HTML 将是最实用的。仍然是最好的，如果已经拿到 Acrobat 全版(非免费读者)，可使用索引工具搜索所有独立的 Acrobat 文档以获得所需文档。

表 B-2 Python 文档资料部分。

表 B-2　Python 文档资料的部分

文档资料部分	说　明
api	本部分说明可用来扩展和 / 或嵌入 Python 的 C API
dist	本部分涵盖用于以开发者的角度分布 Python 模块的工具的开发
doc	本部分简述用于引证 Python 及 Python 程序的方法和工具
ext	本部分说明扩展和嵌入 Python 的方法学。 参看 api 文档，了解 API 本身的信息
inst	inst 部分解释从最终用户的角度安装三部分 Python 模块的方法
lib	本部分说明由 Python 提供的扩展和模块的标准库
mac	本部分解释特定于 Mac 的扩展库和 Python 的组件
ref	Python 参考手册涵盖所有 Python 解释器的核心组件，包括语言、运算符、表达式和语句的语义
tut	本部分是关于如何使用 Python 的一个指导

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

B.4 资源提示

Python 仍是一种相对较新的语言，还没被多家出版社出版。本节将列出关于 Python 的书的所有目录。由于本目录表目前很小，所以本目录中所列的书都值得购买。用户可用这些书来查取相应的参考信息(当然，要与本书相结合起来)。

Brown,M.C.Python Annotated Archives.Berkeley,CA：Osborne/McGraw-Hill,1999.

Python Annotated Archives 是一本关于 Python 编程的指南。这本书是从已完成的应用程序的角度而不是语言语义的角度讲述的。所有例子是逐行注解的，通篇附有指南和必需内容。

Harms,D.mcDonald,K.The Quick Python Book.Greenwich,CT:Manning Publication,1999.

The Quick Python Book 是一本关于 Python 语言基础的好书，适用于不了解 Python 语法结构的编程人员。

Beazley,D.Python Essential Reference.Indianpolis,In:New Riders,1999.

Python Essential Reference 是指导和参考手册综合书，是关于 Python 语言最精简且最全面的一本书。涉及到了用 Python 有效编程所必需的方方面面。

Lutz,M.,D.Ascher.Learning Python.Sebastopol,CA:O 'Reilyy,1999.

Learning Python 是一本好的教材，介绍了从 Python 基本特性和整体结构一直到复杂地使用 Python 作为完全面向对象环境的全部内容。它是学习 Python 语言语义的极佳起点。

Lutz,M.Programming Python.Sebastopol,CA:O 'Reilly,1996.

Programming Python 是 Learning Python 的姊妹篇，是 Python 语言的完全参考书。本书用大量细节论述了 Python 语言的全部内容。这本书里有其他语言的用户颇感头疼的组件；但总体上说，它是 Python 的“圣经”。

Watters,A.,G.van Rossum,J.C.Ahlstrom.Internet Programming with Python.New York:M&T Books,1996.

Internet Programming with Python 是一本好的入门书。集中精力介绍利用 Python 语言作为开发因特网应用程序的一种语言。本书不局限于仅用在 Web 服务器的典型 CGI 应用程序；也描述了如何使用 Python 联网编程(包括通信协议的深层例子)，以及如何把 Python 解释器嵌入到其他因特网应用程序中去。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

B.4 资源提示

Python 仍是一种相对较新的语言，还没被多家出版社出版。本节将列出关于 Python 的书的所有目录。由于本目录表目前很小，所以本目录中所列的书都值得购买。用户可用这些书来查取相应的参考信息(当然，要与本书相结合起来)。

Brown,M.C.Python Annotated Archives.Berkeley,CA：Osborne/McGraw-Hill,1999.

Python Annotated Archives 是一本关于 Python 编程的指南。这本书是从已完成的应用程序的角度而不是语言语义的角度讲述的。所有例子是逐行注解的，通篇附有指南和必需内容。

Harms,D.mcDonald,K.The Quick Python Book.Greenwich,CT:Manning Publication,1999.

The Quick Python Book 是一本关于 Python 语言基础的好书，适用于不了解 Python 语法结构的编程人员。

Beazley,D.Python Essential Reference.Indianpolis,In:New Riders,1999.

Python Essential Reference 是指导和参考手册综合书，是关于 Python 语言最精简且最全面的一本书。涉及到了用 Python 有效编程所必需的方方面面。

Lutz,M.,D.Ascher.Learning Python.Sebastopol,CA:O 'Reilyy,1999.

Learning Python 是一本好的教材，介绍了从 Python 基本特性和整体结构一直到复杂地使用 Python 作为完全面向对象环境的全部内容。它是学习 Python 语言语义的极佳起点。

Lutz,M.Programming Python.Sebastopol,CA:O 'Reilly,1996.

Programming Python 是 Learning Python 的姊妹篇，是 Python 语言的完全参考书。本书用大量细节论述了 Python 语言的全部内容。这本书里有其他语言的用户颇感头疼的组件；但总体上说，它是 Python 的“圣经”。

Watters,A.,G.van Rossum,J.C.Ahlstrom.Internet Programming with Python.New York:M&T Books,1996.

Internet Programming with Python 是一本好的入门书。集中精力介绍利用 Python 语言作为开发因特网应用程序的一种语言。本书不局限于仅用在 Web 服务器的典型 CGI 应用程序；也描述了如何使用 Python 联网编程(包括通信协议的深层例子)，以及如何把 Python 解释器嵌入到其他因特网应用程序中去。

更多IT类电子书，视频教程了请访问： www.guma.pw

更多IT类电子书，视频教程了请访问： www.guma.pw

B.4 资源提示

Python 仍是一种相对较新的语言，还没被多家出版社出版。本节将列出关于 Python 的书的所有目录。由于本目录表目前很小，所以本目录中所列的书都值得购买。用户可用这些书来查取相应的参考信息(当然，要与本书相结合起来)。

Brown,M.C.Python Annotated Archives.Berkeley,CA：Osborne/McGraw-Hill,1999.

Python Annotated Archives 是一本关于 Python 编程的指南。这本书是从已完成的应用程序的角度而不是语言语义的角度讲述的。所有例子是逐行注解的，通篇附有指南和必需内容。

Harms,D.mcDonald,K.The Quick Python Book.Greenwich,CT:Manning Publication,1999.

The Quick Python Book 是一本关于 Python 语言基础的好书，适用于不了解 Python 语法结构的编程人员。

Beazley,D.Python Essential Reference.Indianpolis,In:New Riders,1999.

Python Essential Reference 是指导和参考手册综合书，是关于 Python 语言最精简且最全面的一本书。涉及到了用 Python 有效编程所必需的方方面面。

Lutz,M.,D.Ascher.Learning Python.Sebastopol,CA:O 'Reilyy,1999.

Learning Python 是一本好的教材，介绍了从 Python 基本特性和整体结构一直到复杂地使用 Python 作为完全面向对象环境的全部内容。它是学习 Python 语言语义的极佳起点。

Lutz,M.Programming Python.Sebastopol,CA:O 'Reilly,1996.

Programming Python 是 Learning Python 的姊妹篇，是 Python 语言的完全参考书。本书用大量细节论述了 Python 语言的全部内容。这本书里有其他语言的用户颇感头疼的组件；但总体上说，它是 Python 的“圣经”。

Watters,A.,G.van Rossum,J.C.Ahlstrom.Internet Programming with Python.New York:M&T Books,1996.

Internet Programming with Python 是一本好的入门书。集中精力介绍利用 Python 语言作为开发因特网应用程序的一种语言。本书不局限于仅用在 Web 服务器的典型 CGI 应用程序；也描述了如何使用 Python 联网编程(包括通信协议的深层例子)，以及如何把 Python 解释器嵌入到其他因特网应用程序中去。

更多IT类电子书，视频教程了请访问： www.guma.pw