Learning Python

第3版
涵盖 Python 2.5

Python

学习手册

O'REILLY®
机械工业出版社
China Machine Press
HZ BOOKS

Mark Lutz 著
侯靖 等译

Python学习手册

www.TopSage.com

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

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

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库管理系统(DBMS)精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

平面设计优秀资源学习下载 | Flash 优秀资源学习下载 | 3D 动画优秀资源学习下载

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

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

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

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

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

>> 更多精品资料请访问大家论坛计算机区…

第三版

Python学习手册

Mark Lutz 著

侯靖 等译

O'REILLY®

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

O'Reilly Media, Inc.授权机械工业出版社出版

机械工业出版社

www.TopSage.com

图书在版编目（CIP）数据

Python学习手册（第三版）/（美）鲁特兹（Lutz, M.）著；侯靖等译. —北京：机械工业出版社，2009.8

书名原文：Learning Python, third edition

ISBN 978-7-111-26776-8

I. P... II. ①鲁... ②侯... III. 软件工具－程序设计－手册 IV. TP311.56-62

中国版本图书馆CIP数据核字（2009）第052979号

北京市版权局著作权合同登记

图字：01-2009-4017号

©2007 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Machine Press, 2007. Authorized translation of the English edition, 2009 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由O'Reilly Media, Inc. 出版2007。

简体中文版由机械工业出版社出版 2009。英文原版的翻译得到O'Reilly Media, Inc.的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc.的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

本书法律顾问
北京市展达律师事务所

书　　名/ Python学习手册（第三版）
书　　号/ ISBN 978-7-111-26776-8
责任编辑/ 陈佳媛
封面设计/ Karen Montgomery，张健
出版发行/ 机械工业出版社
地　　址/ 北京市西城区百万庄大街22号（邮政编码 100037）
印　　刷/ 北京京北印刷有限公司
开　　本/ 178毫米×233毫米　16开本　43印张
版　　次/ 2009年8月第1版　2009年8月第1次印刷
定　　价/ 89.00元（册）

凡购本书，如有倒页、脱页、缺页，由本社发行部调换
本社购书热线：(010)68326294

www.TopSage.com

大家网
TopSage.com

O'Reilly Media, Inc.介绍

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

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

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

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

www.TopSage.com

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

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

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库管理系统(DBMS)精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

平面设计优秀资源学习下载 | Flash 优秀资源学习下载 | 3D 动画优秀资源学习下载

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

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

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

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

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

>> 更多精品资料请访问大家论坛计算机区…

目录

前言 1

第一部分 使用入门

第1章 问答环节 19

人们为何使用Python 19
软件质量 20
开发者效率 21
Python是“脚本语言”吗 21
好吧，Python的缺点是什么呢 23
如今谁在使用Python 23
使用Python可以做些什么 24
系统编程 25
用户图形接口 25
Internet脚本 25
组件集成 26
数据库编程 26
快速原型 27
数值计算和科学计算编程 27
游戏、图像、人工智能、XML、机器人等 27

www.TopSage.com

Python有哪些技术上的优点 27
面向对象 28
免费 28
可移植 28
功能强大 29
可混合 30
使用简单 31
Python是工程，不是艺术 31
简单易学 33
名字来源于Monty Python 33
Python和其他语言比较起来怎么样 33
本章小结 34
头脑风暴 35
本章习题 35
习题解答 35

第2章 Python如何运行程序 37

Python解释器简介 37
程序执行 39
程序员的视角 39
Python的视角 40
执行模块的变种 42
Python实现的替代者 42
执行优化工具 44
冻结二进制文件 45
未来的可能性 46
本章小结 47
头脑风暴 48
本章习题 48
习题解答 48

六家网 TopSage.com

www.TopSage.com

第3章 如何运行程序 49
交互模式下编写代码 49
在交互提示模式下测试代码 51
使用交互提示模式 52
系统命令行和文件 52
使用命令行和文件 55
UNIX可执行脚本(#!) 56
UNIX env查找技巧 57
点击文件图标 57
在Windows中点击图标 58
raw_input的技巧 58
图标点击的其他限制 60
模块导入和重载 61
模块的显要特性：属性 62
import和reload的使用注意事项 65
IDLE用户界面 66
IDLE基础 66
使用IDLE 68
高级IDLE工具 70
其他的IDE 70
嵌入式调用 71
动付二进制的可执行性 72
文本编辑器启动的选择 72
其他的启动选择 73
未来的可能 73
我应该选用哪种 73
本章小结 74
头脑风暴 75
本章习题 75
习题解答 75
头脑风暴：第一部分 练习题 77

www.TopSage.com

第二部分 类型和运算

第4章 介绍Python对象类型 81

为什么使用内置类型 82

Python的核心数据类型 82

数字 84

字符串 85

序列的操作 85

不可变性 87

类型特定的方法 88

寻求帮助 89

编写字符串的其他方法 90

模式匹配 91

列表 91

序列操作 92

类型特定的操作 92

边界检查 93

嵌套 93

列表解析 94

字典 95

映射操作 95

重访嵌套 96

键的排序：for 循环 97

迭代和优化 99

不存在的键：if 测试 100

元组 101

为什么要用元组 101

文件 101

其他文件类工具 102

其他核心类型 103

如何破坏代码的灵活性 104

www.TopSage.com

用户定义的类 104
剩余的内容 105
本章小结 106
头脑风暴 107
本章习题 107
习题解答 107

第5章 数字 109
Python的数字类型 109
数字常量 109
内置数学工具和扩展 111
Python表达式操作符 112
混合操作所遵循的操作符优先级 113
括号分组的子表达式 113
混合类型自动升级 114
预习：运算符重载 115
在实际应用中的数字 115
变量和基本的表达式 115
数字显示的格式 117
str和repr显示格式 118
除法：传统除法、Floor除法和真除法 118
位操作 119
长整型数 120
复数 121
十六进制和八进制记数 121
其他的内置数学工具 122
其他数字类型 123
小数数字 124
集合 125
布尔型 126
第三方扩展 127
本章小结 127

大象网 TopSage.com

www.TopSage.com

头脑风暴 128
本章习题 128
习题解答 128

第6章 动态类型简介 129
缺少类型声明语句的情况 129
变量、对象和引用 129
类型属于对象，而不是变量 131
对象的垃圾收集 132
共享引用 133
共享引用和在原处修改 135
共享引用和相等 136
动态类型随处可见 138
本章小结 138
头脑风暴 139
本章习题 139
习题解答 139

第7章 字符串 140
字符串常量 141
单双引号字符串是一样的 142
用转义序列代表特殊字节 142
字符串抑制转义 145
三重引号编写多行字符串块 146
字符串编码更大的字符集 147
实际应用中的字符串 149
基本操作 149
索引和分片 151
为什么要在意：分片 154
字符串转换工具 154
修改字符串 157

www.TopSage.com

字符串格式化 158
更高级的字符串格式化 159
基于字典的字符串格式化 160
字符串方法 161
字符串方法实例：修改字符串 162
字符串方法实例：文本解析 164
实际应用中的其他常见字符串方法 165
最初的字符串模块 166
通常意义下的类型分类 168
同样分类的类型共享其操作集合 168
可变类型能够在原处修改 168
本章小结 169
头脑风暴 170
本章习题 170
习题解答 170

第8章 列表与字典 171

列表 171
实际应用中的列表 174
基本列表操作 174
索引、分片和矩阵 174
原处修改列表 175
字典 179
实际应用中的字典 181
字典的基本操作 181
原处修改字典 182
其他字典方法 183
语言表 184
字典用法注意事项 185
为什么要在意字典接口 189
本章小结 189

头脑风暴 190
本章习题 190
习题解答 190
第9章 元组、文件及其他 191
元组 191
实际应用中的元组 192
为什么有了列表还要元组 194
文件 195
打开文件 195
使用文件 196
实际应用中的文件 197
其他文件工具 201
重访类型分类 201
为什么要在意操作符重载 202
对象灵活性 202
引用 VS 拷贝 204
比较、相等性和真值 206
Python中真和假的含义 207
Python的类型层次 209
Python中的其他类型 210
内置类型陷阱 210
赋值生成引用，而不是拷贝 210
重复能够增加层次深度 211
留意循环数据结构 212
不可变类型不可以在原处改变 212
本章小结 213
头脑风暴 214
本章习题 214
习题解答 214
头脑风暴：第二部分练习题 215

www.TopSage.com

第三部分 语句和语法

第10章 Python语句简介 221

重访Python程序结构 221

Python的语句 222

两个if的故事 223

Python增加了什么 224

Python删除了什么 224

终止行就是终止语句 225

为什么使用缩进语法 226

几个特殊实例 228

简短实例：交互循环 230

一个简单的交互式循环 230

对用户输入数据做数学运算 231

用测试输入数据来处理错误 232

用try语句处理错误 233

嵌套代码三层 234

本章小结 235

头脑风暴 236

本章习题 236

习题解答 236

第11章 赋值、表达式和打印 237

赋值语句 237

赋值语句的形式 238

序列赋值 239

多目标赋值语句 242

增强赋值语句 243

变量命名规则 245

表达式语句 248

表达式语句和在原处的修改 249

打印语句 249
Python的“Hello World”程序 250
重定向输出流 251
print >> file扩展 252
本章小结 253
头脑风暴 255
本章习题 255
习题解答 255

第12章 if测试 256

if语句 256
通用格式 256
基本例子 257
多路分支 257
Python语法规则 259
代码块分隔符 260
语句的分隔符 261
一些特殊情况 262
真值测试 262
if/else三元表达式 264
为什么要在意布尔值 266
本章小结 267
头脑风暴 268
本章习题 268
习题解答 268

第13章 while和for循环 269

while循环 269
一般格式 269
例子 270
break、continue、pass和循环else 271
一般循环格式 271

www.TopSage.com

例子 271
为什么要在意“模拟C 语言的while循环” 275
for循环 275
一般格式 276
例子 276
为什么要在意“文件扫描” 279
迭代器：初探 280
文件迭代器 281
其他内置类型迭代器 283
其他迭代环境 284
用户定义的迭代器 285
编写循环的技巧 285
循环计数器：while和range 286
非完备遍历：range 287
修改列表：range 288
并行遍历：zip和map 289
产生偏移和元素：enumerate 291
列表解析：初探 292
列表解析基础 293
对文件使用列表解析 294
扩展列表解析语法 295
本章小结 296
头脑风暴 297
本章习题 297
习题解答 297

第14章 文档 299

Python文档资源 299
#注释 300
dir函数 300
文档字符串：__doc__ 301
PyDoc：help函数 304

大象网 TopSage.com

www.TopSage.com

PyDoc：HTML报表 306
标准手册集 309
网络资源 310
已出版的书籍 311
常见编写代码的陷阱 311
本章小结 313
头脑风暴 314
本章习题 314
习题解答 314
头脑风暴：第三部分练习题 315

第四部分 函数

第15章 函数基础 319

为何使用函数 320
编写函数 320
def语句 322
def语句是实时执行的 322
第一个例子：定义和调用 323
定义 323
调用 324
Python中的多态 324
第二个例子：寻找序列的交集 325
定义 326
调用 326
重访多态 327
本地变量 328
本章小结 328
头脑风暴 329
本章习题 329
习题解答 329

www.TopSage.com

第16章 作用域和参数 330
作用域法则 330
函数作用域基础 331
变量名解析：LEGB原则 332
作用域实例 334
内置作用域 334
global语句 336
最小化全局变量 337
最小化文件间的修改 338
其他访问全局变量的方法 339
作用域和嵌套函数 340
嵌套作用域的细节 341
嵌套作用域举例 341
传递参数 347
参数和共享引用 348
避免可变参数的修改 349
对参数输出进行模拟 350
特定的参数匹配模型 351
关键字参数和默认参数的实例 352
任意参数的实例 354
关键字参数和默认参数的混合 356
min调用 357
一个更有用的例子：通用set函数 359
参数匹配：细节 360
为什么要在意：关键字参数 361
本章小结 362
头脑风暴 363
本章习题 363
习题解答 364

第17章 函数的高级话题 365
匿名函数：lambda 365

www.TopSage.com

lambda表达式 365
为什么使用lambda 367
如何（不要）让Python代码变得晦涩难懂 368
嵌套lambda和作用域 369
作为参数来应用函数 370
内置函数apply 370
为什么要在意：回调 371
传入关键字参数 372
和apply类似的调用语法 372
在序列中映射函数：map 373
函数式编程工具：filter和reduce 374
重访列表解析：映射 376
列表解析基础 376
增加测试和嵌套循环 377
列表解析和矩阵 380
理解列表解析 381
为什么要在意：列表解析和map 382
重访迭代器：生成器 383
生成器函数实例 383
扩展生成器函数协议：send和next 385
迭代器和内置类型 386
生成器表达式：迭代器遇到列表解析 387
对迭代的各种方法进行计时 388
函数设计概念 390
函数是对象：简洁调用 392
函数陷阱 393
本地变量是静态检测的 393
默认和可变对象 395
没有return语句的函数 396
嵌套作用域的循环变量 397
本章小结 397

www.TopSage.com

头脑风暴 398
本章习题 398
习题解答 398
头脑风暴：第四部分 练习题 400

第五部分 模块

第18章 模块：宏伟蓝图 405
为什么使用模块 405
Python程序构架 406
如何组织一个程序 407
导入和属性 407
标准库模块 409
import如何工作 409
搜索 410
编译（可选） 414
运行 414
第三方工具：distutils 415
本章小结 415
头脑风暴 417
本章习题 417
习题解答 417

第19章 模块代码编写基础 418
模块的创建 418
模块的使用 419
import语句 419
from语句 420
from * 语句 420
导入只发生一次 420
import和from是赋值语句 421
文件间变量名的改变 422

import和from的对等性 422
from语句潜在的陷阱 423
模块命名空间 425
文件生成命名空间 425
属性名的点号运算 427
导入和作用域 427
命名空间的嵌套 428
重载模块 429
reload基础 430
reload实例 431
为什么要在意：模块重载 432
本章小结 432
头脑风暴 434
本章习题 434
习题解答 434

第20章 模块包 435

包导入基础 435
包和搜索路径设置 436
__init__.py包文件 436
包导入实例 438
包对应的from和import 439
为什么要使用包导入 440
三个系统的传说 441
为什么要在意：模块包 443
本章小结 443
头脑风暴 445
本章习题 445
习题解答 445

第21章 高级模块话题 446

在模块中隐藏数据 446
最小化from *的破坏：_X和__all__ 446

www.TopSage.com

启用以后的语言特性 447
混合用法模式：__name__和__main__ 447
以__name__进行单元测试 448
修改模块搜索路径 450
import as扩展 451
相对导入语法 451
为什么使用相对导入 452
模块设计理念 454
模块是对象：元程序 455
模块陷阱 457
顶层代码的语句次序的重要性 457
通过变量名字符串导入模块 458
from复制变量名，而不是连接 459
from *会让变量语义模糊 460
reload不会影响from导入 460
reload、from以及交互模式测试 461
reload的使用没有传递性 462
递归形式的from import无法工作 463
本章小结 464
头脑风暴 465
本章习题 465
习题解答 465
头脑风暴：第五部分练习题 466

第六部分 类和OOP

第22章 OOP：宏伟蓝图 471

为何使用类 472
概览OOP 473
属性继承搜索 473
类和实例 475
类方法调用 476

大家网 TopSage.com

编写类树 476
OOP是为了代码重用 479
本章小结 481
头脑风暴 483
本章习题 483
习题解答 483

第23章 类代码编写基础 485
类产生多个实例对象 485
类对象提供默认行为 486
实例对象是具体的元素 486
第一个例子 486
类通过继承进行定制 489
第二个例子 489
类是模块内的属性 491
类可以截获Python运算符 492
第三个例子 493
为什么要使用运算符重载 494
世界上最简单的Python类 495
本章小结 497
头脑风暴 499
本章习题 499
习题解答 499

第24章 类代码编写细节 501
class语句 501
一般形式 501
例子 502
方法 504
例子 505
调用超类的构造器 506
其他方法调用的可能性 506
继承 507

www.TopSage.com

属性树的构造 507
继承方法的专有化 508
类接口技术 509
抽象超类 510
运算符重载 511
常见的运算符重载方法 512
__getitem__拦截索引运算 513
__getitem__和__iter__实现迭代 513
用户定义的迭代器 514
__getattr__和__setattr__捕捉属性的引用 518
模拟实例属性的私有性 520
__repr__和__str__会返回字符串表达形式 520
__radd__处理右侧加法 522
__call__拦截调用 523
函数接口和回调代码 523
__del__是析构器 525
命名空间：完整的内容 526
简单变量名：如果赋值就不是全局变量 526
属性名称：对象命名空间 527
Python命名空间的“禅”：赋值将变量名分类 527
命名空间字典 529
命名空间链接 531
一个更实际的例子 533
本章小结 536
头脑风暴 537
本章习题 537
习题解答 537

第25章 类的设计 539

Python和OOP 539
通过调用标记进行重载（或不要） 540
类作为记录 540

类和继承：“是一个”关系 542
类和组合：“有一个”关系 544
重访流处理器 545
为什么要在意：类和持续性 548
OOP和委托 548
多重继承 549
类是对象：通用对象的工厂 552
为什么有工厂 554
方法是对象：绑定或无绑定 554
重访文档字符串 556
为什么要在意：绑定方法和回调函数 557
类和模块 558
本章小结 558
头脑风暴 559
本章习题 559
习题解答 559

第26章 类的高级主题 560

扩展内置类型 560
通过嵌入扩展类型 560
通过子类扩展类型 561
类的伪私有属性 563
变量名压缩概览 564
为什么使用伪私有属性 564
新式类 566
钻石继承变动 567
其他新式类的扩展 570
静态和类方法 573
使用静态和类方法 575
函数装饰器 576
装饰器例子 578

TopSage.com

www.TopSage.com

类陷阱 579
修改类属性的副作用 579
多重继承：顺序很重要 580
类、方法以及嵌套作用域 581
"过度包装" 583
本章小结 584
头脑风暴 585
本章习题 585
习题解答 585
头脑风暴：第六部分 练习题 586

第七部分 异常和工具

第27章 异常基础 595

为什么使用异常 596
异常的角色 596
异常处理：简明扼要 597
try/except/else语句 601
try语句分句 602
try/else分句 604
例子：默认行为 605
例子：捕捉内置异常 606
try/finally语句 607
例子：利用try/finally编写终止行为 608
统一try/except/finally 609
通过嵌套合并finally和except 610
合并try的例子 611
raise语句 612
例子：引发并捕捉用户定义的异常 613
例子：利用raise传入额外的数据 613
例子：利用raise传递异常 614

assert语句 614
例子：收集约束条件（但不是错误） 615
with/as环境管理器 616
基本使用 616
环境管理协议 617
为什么要在意：错误检查 618
本章小结 620
头脑风暴 621
本章习题 621
习题解答 621

第28章 异常对象 622
基于字符串的异常 623
字符串异常就要出局了 623
基于类的异常 624
类异常例子 624
为什么使用类异常 626
内置Exception类 629
定义异常文本 630
发送额外数据和实例行为 631
raise语句的一般形式 633
本章小结 634
头脑风暴 636
本章习题 636
习题解答 636

第29章 异常的设计 637
嵌套异常处理器 637
例子：控制流程嵌套 638
例子：语法嵌套化 639
异常的习惯用法 641
异常不总是错误 641
函数信号条件和raise 641

www.TopSage.com

在try外进行调试 642
运行进程中的测试 643
关于sys.exc_info 644
与异常有关的技巧 644
应该包装什么 644
捕捉太多：避免空except语句 645
捕捉过少：使用基于类的分类 647
异常陷阱 647
字符串异常匹配是通过同一性而不是通过值 648
捕捉到错误的异常 649
核心语言总结 649
Python工具集 650
大型项目的开发工具 651
本章小结 653
头脑风暴 655
本章习题 655
习题解答 655
头脑风暴：第七部分 练习题 656

第八部分 附录⊖

附录A 安装和配置

附录B 每部分练习题解答

⊖ 第八部分附录内容请到华章网站（*www.hzbook.com*）下载。

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

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

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库管理系统(DBMS)精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

平面设计优秀资源学习下载 | Flash 优秀资源学习下载 | 3D 动画优秀资源学习下载

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

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

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

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

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

>> 更多精品资料请访问大家论坛计算机区…

前言

本书是学习Python编程语言的入门书籍。Python是一种很流行的程序语言，可以作为独立的程序和脚本在各种领域中应用。Python免费、可移植、功能强大，而且使用起来相当容易。

无论你是编程初学者，还是专业开发人员，本书的目标是让你快速掌握核心Python语言基础。阅读本书后，会对Python有足够的了解，能够将其应用于你所要从事的应用领域中。

关于第三版

本书第二版于2003年末出版，从那时到现在的四年中，Python语言本身已经发生了实质性的变化，因此我在Python培训课程中介绍的话题也随之发生了改变。虽然我试着尽可能保留上一版的文字，但是本版书反映了Python语言和Python培训中发生的最新变化，此外还有一些结构性的改变。

本书中Python语言的变化

对于Python语言，本书进行了全面更新，反映了Python 2.5以及从第二版出版以来语言的所有变化（第二版大部分是基于Python 2.2，以及在Python 2.3版即将发布前提出的Python 2.3版的一些特性）。此外，还会在本书合适的地方讨论Python 3.0的预期的变化。下面是这门语言的一些主要的主题，读者将会发现这些主题在本书中是全新的内容或在原有基础上进行了扩展：

- 新的`B if A else C`条件表达式（第12章）。
- `with/as`环境管理器（第27章）。

www.TopSage.com

大家网
TopSage.com

- 统一try/except/finally（第27章）。
- 相对导入语法（第21章）。
- 生成器表达式（第17章）。
- 新的生成器函数的特性（第17章）。
- 函数装饰器（第26章）。
- 集合对象类型（第5章）。
- 新的内置函数：sorted、sum、any、all、enumerate（第4章和第13章）。
- 小数固定精度对象类型（第5章）。
- 文件、列表解析（list comprehension）和迭代器等新的扩展内容（第13章和第17章）。
- 新的开发工具：Eclipse、distutils、unittest和doctest、IDLE加强版、Shedskin等（第3章和第29章）。

贯穿本书始末都会讨论（Python语言的一些细微变化）（例如，广泛使用的True和False、用新的sys.exc_info来获取异常的细节、基于字符串的异常的退出、字符串方法、apply和reduce内置函数）。此外，有些在上一版中属于新的功能，也会在本书中进行扩展，包括第三个限制值的分片运算，以及书中apply的任意参数调用语法。

本书中Python培训的变化

除了Python语言变化外，本书又增加了我最近几年开设的Python培训课程中所介绍的新主题和例子。例如，读者会发现：

- 全新的一章介绍内置类型（第4章）。
- 全新的一章介绍语句语法（第10章）。
- 全新的一章以加强涵盖的方式，介绍动态定型（第6章）。
- 详细的OOP介绍（第22章）。
- 文件、作用域、嵌套语句、类以及异常等更多的新例子。

很多新增的内容和变化都是考虑到Python的初学者，有些主题则移到了培训课程中最恰当的地方，在那里最易于吸收所学知识。例如，列表解析和迭代器，最初是和for循环语句一起出现的，而不是和之后的函数工具一起出现的。

www.TopSage.com

读者也会发现，很多关于原本核心语言的讨论在本书中有了实质性的扩展，增加了新的讨论内容和例子。因为本书已成为学习核心Python语言的标准资源，介绍时我有意地让内容更加完整，使用了新的案例来进行扩展。

此外，本书也整合了新的Python技巧和窍门，这是我在过去10年中，从课堂上授课以及过去15年使用Python从事实际工作所收集来的。练习题已做了更新和扩充，来反映当前Python的最佳实践、新的语言特性以及课堂中所遇到的初学者常犯的错误。总之，本书所讨论的核心语言内容比之前的版本更多，不仅仅是因为Python变得更大，还因为我增加了一些在实际应用中证实了相当重要的脉络信息。

本书结构性变化

和上一版一样，为了适应本书当前更为完整的事实，书的内容经过了合理的划分。也就是说，我把核心语言题材组织成许多章节，让内容更易于理解。例如，类型和语句现在是两个顶层的部分，每种主要类型和语句话题都有一章。这种新结构的设计，是为了在本书中多讲一些内容，而又不会让读者产生恐惧。在此过程中，练习题和“陷阱”（常见的错误）则从每章的结尾移到了每部分结尾，现在它们出现在每部分最后一章的结尾处。

在第三版中，我也增强了每部分结尾处的练习题和每章结尾的小结以及每章结尾的习题，来帮助读者在做完那些题目后，可以达到复习章节的效果。每章的最后都有一组问题，来帮助读者复习和测试对该章内容的理解。练习题解答在附录B。即使读者保证能够正确回答问题，我还是鼓励读者去看一看解答，因为答案本身就是一种复习。

尽管有这些新主题，本书仍然是面向Python初学者的，其设计目标就是要作为程序员学习Python语言的首选书籍（注1）。本书保留前两版的许多题材、结构和焦点。在适当的地方，我会为初学者扩展简介的内容，同时把更高级的新话题和讨论的主线隔离出来，以免掩盖了基础知识。另外，因为本书绝大部分是采用经过时间检验的培训课程的经验和题材编写的，就像前两版那样，仍然可以作为自学Python的入门教材。

注1：所谓的“程序员”，是指的是过去以任何程序语言或脚本语言编写过一行程序代码的任何人。如果这不包括你在内，你还是可能会觉得这本书有些用处，但是，要注意一点，本书会花很多时间教授Python的知识，而不是谈论编程基础。

本书的范围变化

编写第三版的目的是将其作为核心Python语言的教程，仅此而已。首先要深入学习这门语言，然后学习应用层面的编程应用。本书的介绍是由下至上并且是循序渐进的，不过，本书会对整个语言进行概述，独立于其应用角色之外。

对某些人而言，“学习Python”就是花一两个小时在网络上看一看教程。对于已经是高级程序员的人来说，这样行得通，毕竟和其他语言相比，Python确实简单得多。这种走捷径的问题在于，这些实习生最终可能会在不寻常的情况中跌倒而被卡住：变量变了，可变默认参数值的变化难以理解等。本书的目标是提供扎实的Python基础知识，即使发生不寻常情况时，也能够理解。

这种范围应该慎重。我们把注意力集中在语言基础上，就能以更加令人满意的深度研究这些主题。其他书籍，例如，O'Reilly的《Programming Python》、《Python Cookbook》、《Python in a Nutshell》以及《Python Pocket Reference》，则补充了本书的遗漏，对应用层面的主题和参考材料提供更完整的说明。本书的目的是致力于教授Python，以便能够将其应用在读者所从事的任何领域中。

因为本书的侧重点有所变化，上一版中有些参考内容和更高级内容的章节（大约占上一版的15%）已被去掉，扩充了核心语言的章节。这样一来，本书的读者会发现对核心语言有了更完整的介绍，使得本书成了更实用的Python书籍。本版中加入了一些更高级的例子，作为读者自学的程序，也当作最终的练习题（参考第29章）。

关于本书

本部分强调了本书的一般性重点，和本书的版本无关。没有哪本书可以满足每一位潜在的读者，所以阅读之前了解编写本书的目标是很重要的。

事前准备

事实上，使用本书确实没有什么绝对的先决条件。初学者和功底深厚的编程高手都可以成功地使用本书。如果打算学习Python，本书可能就适合你。不过，一般来说，我发现在使用本书之前有过任何编程或脚本经验的读者，会有些帮助。但是，并不要求每位读者都得这样。

本书是作为程序员学习Python的入门书籍来设计的。对于那些从来没有接触过计算机的人，可能就不适合（例如，我不会花时间讨论计算机是什么）。但是，这并不意味你需要有编程的背景或教育。

www.TopSage.com

另一方面，我不会假设读者什么都不懂而冒犯了读者：使用Python来做有意义的事，这很容易，而本书就是要教读者怎样做。本书有时会用Python和C、C++、Java以及Pascal语言来做比较，但是如果读者过去没有使用过这些语言，则完全可以放心地忽略这些比较。

本书的范围和其他书籍

虽然本书涵盖了Python语言所有的基本内容，但基于速度和篇幅的考虑，我还是把本书的范围缩小了。为了让事情简单化，本书关注核心概念，使用小并且独立完备的例子来示范重点知识，并且有时省略了可以在参考手册中找到的细节。因此，把本书当作通往更高级应用的垫脚石和完整的入门书籍再好不过了。

例如，我们不会谈太多的Python/C集成，这个复杂话题显然是许多基于Python的系统的核心。我们也不会谈太多Python的历史或发展过程。对于流行的Python应用程序也只简单浏览而已，例如，GUI、系统工具以及网络脚本机制，而有的则根本不提。显然，这会漏掉整体内容的一部分。

从整体上来说，Python是为了让脚本世界的质量等级再提升几个级别。而Python的有些观念需要的背景环境，不是这里所能提供的，如果没有推荐读者读完此书后进行更深入地学习，那就是我的疏忽了。我希望本书的绝大多数读者最终都可以继续走下去，从其他书籍完整了解应用层面的编程技术。

因为本书关注的是初学者，设计上自然是和O'Reilly的其他Python书籍互补。例如，我编写的另一本书《Programming Python》，提供更大并且更完整的例子，还有应用程序编程技巧的教程，而且我有意将其设计为读完此书后的后续书籍。概括地说，本书的目前版本和《Programming Python》反映了作者培训内容的两部分：核心语言和应用程序程序设计。此外，O'Reilly的《Python Pocket Reference》也是搜索本书忽略的一些细节的快速参考手册。

其他后续的书籍也可提供参考、附加的例子或者于特定领域中（例如，《Web开发和GUI》）使用Python的细节。例如，O'Reilly的《Python in a Nutshell》以及Sams的《Python Essential Reference》提供了参考，而O'Reilly的《Python Cookbook》为那些已熟识应用程序设计技巧的人，提供了独立完备的例子。因为别人对书籍的评价是很主观的，我鼓励读者亲自浏览这些书，来选择满足自己需求的进阶书籍。不过，无论选择哪本书，要记住，Python其余内容所需学习的例子都相当实际，以致于这里没有空间能够容纳。

www.TopSage.com

尽管这么说，我想读者还是会发现本书是作为学Python的首先书籍，虽然范围有限（也许也正是因为如此）。你会学到初学阶段编写独立的Python程序和脚本所需要的一切。当你读完本书时，不仅学到了语言本身，也会学到如何将其合理地运用于日常工作中去。此外，当读者遇到更高级的主题和例子时，将会有足够能力去解决它们。

本书的风格和结构

本书是基于为期3天的Python课程的培训材料编写而成的。每章末尾有本章对应的习题，并且在每部分最后一章末尾有本部分对应的练习题。习题和练习题的解答在附录B。习题可以帮助读者复习的内容，而练习题引以帮助读者以正确的方式编写代码，而且这通常也是该课程的重点之一。

我强烈建议做一下习题和练习题，不仅是为了积累Python编程的经验，也是因为有些练习题会引出本书没有涉及的主题。如果碰上难题，附录B的解答应该可以帮助你（而且我也鼓励你尽量地阅读那些解答）。

本书的整体结构也是来自于课程的培训材料。因为本书是用来快速地介绍语言的基础的，我是以语言的主要功能进行组织并介绍的，而不是以例子为主。我们采用了由下至上的手法：从内置对象类型，到语句，再到程序单元等。每章都比较完备，但是后续的章节会利用到前面章节所介绍的概念（例如，谈到类时，我假定你已经知道如何编写函数），所以对多数读者来说，循序渐进应该是最合理的阅读方法。

一般来说，本书用由下至上的方式介绍Python语言，以一部分谈一种主要语言功能（类型以及函数等等）来组织结构，并且多数例子都很小，它们都是独立完备的（有些人可能会说本书的例子显得空洞，但是，这些例子都是为了说明知识点而设计的）。更确切地说，本书内容如下。

第1部分，使用入门

我们以概览Python作为开始，来回答一些常见的问题：为什么要使用这门语言、它有什么用处等。第1章介绍这门技术背后的主要思想，以及历史背景。然后，介绍本书技术方面的内容，我们会探索Python运行程序的方式。介绍这一部分的目标是让读者有足够的知识，可以跟上后面的例子和练习题的步伐。

第2部分，类型和运算

接着，我们开始Python语言之旅，深入研究Python的主要内置对象类型：数字、列表和字典等。使用这些工具，就可以用Python做很多事了。这是本书最重要的一部分，因为部分内容这是学习后续章节的基础。我们也会在此部分谈到动态定型和其引用值：这是掌握用Python的关键。

www.TopSage.com

第3部分，语句和语法

本部分开始介绍Python的语句：输入的代码会在Python中创建并处理对象。此外，本部分也会介绍Python的一般语法模型。虽然这一部分的重点是语法，但也会介绍相关的工具。例如，PyDoc系统，并探索其他一些编写代码的方法。

第4部分，函数

在这一部分开始讨论Python的更高层次的程序结构工具。函数是为重用而打包代码并避免代码冗余的简单方式。在这一部分内容中，我们将会探索Python的作用域法则、参数传递等技术。

第5部分，模块

Python模块把语句和函数组织成更大的组件，而这一部分会说明如何创建、使用并重载模块。我们也会在这里看到一些更高级的主题，例如，模块包、模块重载以及__name__变量。

第6部分，类和OOP

在一部分，我们探索了Python的面向对象编程（OOP）工具类 —— 类是选用的，但却是组织代码来定制和重用的强大工具。读者将会看到，类几乎是重复利用在本书中谈到的概念，而Python的OOP多数就是在链接的对象中查找变量名。读者也会了解到，OOP在Python中是选用的，但是可以减少大量的开发时间，尤其是对长期的策略性项目开发来说。

第7部分，异常和工具

本书最后一部分讨论Python异常处理模型和语句，加上对开发工具的简介（当读者开始编写较大的程序时，工具就会变得更实用。例如，调试和测试工具）。这一部分放在最后，是因为异常现在应该都是类了。

第8部分，附录（附录内容请到华章网站下载）

本书结尾是两个附录，介绍了在各种计算机上使用Python的与平台相关的技巧（附录A），并提供了每章结尾习题和每部分末尾的练习题的解答（附录B）。

注意：索引和目录可用于查找细节，但本书没有参考文献附录（本书是教程，而不是参考书）。就像之前提到的一样，读者可以参考《Python Pocket Reference》（O'Reilly）还有其他书籍，以及免费的Python参考手册（参看*http://www.python.org*）来了解语法和内置工具的细节。

大家网
TopSage.com

书籍更新

本书在不断地进行完善（输入错误也包括在内）。本书的更新、补充以及更正会在下列任一网站进行更新维护。

http://www.oreilly.com/catalog/9780596513986/（O'Reilly的本书的网页）
http://www.rmi.net/~lutz（作者的网站）
http://www.rmi.net/~lutz/about-lp.html（作者的关于本书的网页）

这三个URL中的最后一个是关于本书的网页，我会在此发布更新，如果链接失效了，一定要进行Web搜索。如果我更有洞察力，我会尽力，但网页的修改比印刷书籍要快得多。

关于本书的程序

本书和其中的所有程序的例子都是基于Python 2.5版的。此外，虽然我没有试着预测未来，但是，在本书的编写过程中会讨论Python 3.0版的一些概念。

然而，因为本书的重点是核心语言，可以相当肯定，多数内容在Python以后的版本中不会有太多的变化。本书多数内容也适用于早期的Python版本。当然，如果读者尝试使用在其所用版本之后增加的扩展功能，那当然行不通了。

原则就是，最新的Python就是最好的Python。因为本书重点是核心语言，多数内容也适用第2章提到的Jython（基于Java的Python语言实现）以及其他Python的实现。

本书例子的源代码以及练习题的解答都可从本书网站获取：*http://www.oreilly.com/catalog/9780596513986/*。那么读者该怎样运行例子呢？本书会在第3章介绍运行的细节，稍后即可了解更多细节。

迎接Python 3.0

Python 3.0的alpha版在本书还没有出版前已经问世。确切地说，本书是基于Python 2.x的版本系列（确切地说，就是2.5版），也增加了很多有关即将发布的Python 3.0版修改注解。

本书出版后的一年内，3.0版都不会正式推出，而且至少两年内，也不会广泛地使用。然而，如果读者在3.0广泛使用后挑选了本书，这一部分提供了读者将会遇到的这门语言的一些修改的简单说明，从而帮助读者顺利过渡。

www.TopSage.com

虽然会有些例外，但是Python 3.0版中的绝大部分内容都会和本书说明的相同，而且对典型和实际应用程序的冲击也会比较小。也就是说，本书介绍的Python基础不会随新版本的发布而改变，并且读者在处理版本特定的细节前，先学习这些基础知识，自然受益匪浅。

不过，为了帮助读者以学转换代码，下列清单列出了Python 3.0和之前版本相比的主要的差异。这一版的章节目标，不是介绍这些修改，就是介绍这些修改带来的影响，因此这份清单配合了相关的章节。有些修改已经可在Python 2.5中编写，但有些还不行。因为下面的清单对多数读者而言都没有意义，所以我建议先学习本书，掌握Python的基础知识，然后再回来看这份清单,从而了解Python 3.0中的修改之处。在Python 3.0中：

- 当前的execfile()内置函数已被移除；改用exec()（参考第3章）。
- reload()内置函数可能会被移除；它的替代函数尚未确定（参考第3章和第19章）。
- `X`反引号字符串转换表达式已被移除：使用repr(X)（参考第5章）。
- X<>Y多余的不相等表达式已被移除：使用X != Y（参考第5章）。
- 集合可能以新的常量语法{1, 3, 2}创建，相当于现在的set([1, 3, 2])（参考第5章）。
- 集合解析（set comprehension）可能写成：{f(x) for x in S if P(x)}，相当于现在的生成器表达式：set(f(x) for x in S if P(x))（参考第5章）。
- 已支持真除法：X / Y一定会传回保留小数部分的浮点数，即使是对于整数应用的除法；使用X // Y会启用当前的省略结果小数部分的除法（参考第5章）。
- 只有一种整数类型int，它支持当前长整型数类型的任意精度（参考第5章）。
- 八进制和二进制常量：当前八进制数0666会有错误：改用0o666。而oct()函数的结果也会随之发生改变；0b1010现在等于10，而bin(10)会传回"0b1010"（参考第5章）。
- 字符串类型str支持长字符Unicode文本。新的bytes类型会代表短字符字符串（例如，以二进制模式加载文件时）；bytes是小整数构成的可变序列，与str的接口稍有不同（参考第7章）。
- 增加新的、选用的字符串格式化技术："See {0}, {1} and {foo}".format("A", "B", foo="C")会创建"See A, B and C"（参考第7章）。
- 字典D.has_key(X)方法会被移除：改用X in D成员关系（参考第4章和第8章）。

www.TopSage.com

- 非数值的混合类型的比较（通过代理和排序函数）会引发异常，改用当前任意但固定的类型间次序（参考第8章和第9章）。
- 字典方法.keys()、.items()以及.values()会返回类似于可迭代的“view”对象，而不是列表；如果确实需要，使用list()来强制创建列表（参考第8章）。
- 因为上点的改变，这种编写代码的模式不能再使用：k = D.keys(); k.sort();，要改用k = sorted(D)（参考第4章和第8章）。
- file()内置函数可能会被移除；改用open()（参考第9章）。
- 内置函数raw_input()会更名为input()；使用eval(input())来实现现在的input()函数的行为（参考第10章）。
- exec程序代码字符串执行语句会再次变成内置函数（参考第10章）。
- as、with以及nonlocal会变成新的保留字；因为上一点的改变，exec不再是保留字（参考第11章）。
- print变成函数从而支持更多功能，不再是语句：使用print(x, y)，而不是print x, y，并且可以使用新函数的关键词参数，定制打印行为：file=sys.stdout, sep=" "以及end="\n"（参考第11章）。
- 扩展了迭代对象的拆包机制：现在可以用支持通用序列赋值的语句，例如a, b, *rest = some_sequence，就像*rest, a = stuff；这样赋值语句左右两侧的元素数目不再必须匹配（参考第11章）。
- 函数通过序列赋值而让元组参数自动解开的机制移除了；你不能再写def foo(a, (b, c)):，而必须使用def foo(a, bc): b, c = bc来明确进行序列赋值（参考第11章，注2）。
- 目前的内置函数xrange()更名为range()；也就是说，只有range()（参考第13章）。
- 在迭代器协议中，X.next()方法更名为X.__next__()，而新的内置函数next(X) 调用对象的X.__next__()方法（参考第13章和第17章）。
- 内置函数zip()、map()以及filter()会返回迭代器；使用list()会强制创建列表作为结果（参考第13章和第17章）。
- 函数可以包括批注说明参数和结果（选用）：def foo(x: "spam", y: list(range(3))) -> 42*2:会在运行时附加在函数对象的foo.func_annotations字典属性中：{'x': "spam", 'y': [0, 1, 2], "return": 84}（参考第15章）。

注2： Python3.0正式发布时，其语法应该是：def foo(a, b_c): b, c = b_c。

www.TopSage.com

- 新的nonlocal x, y语句可以让你写入读取嵌套函数作用域中的变量（参考第16章）。
- apply(func, args, kws)函数会被移除；改用func(*args, **kws)调用语法（参考第16章和第17章）。
- reduce()内置函数会被移除；改为使用本书所示的方法编写循环；lambda、map()以及filter()在3.0中保留（参考第17章）。
- 所有导入默认将变成绝对导入，而且会跳过包的自身目录：使用新语法from . import name将启用目前的相对导入（参考第21章）。
- 所有类都会是新式类，而且会支持目前的新式扩展（参考第26章）。
- 目前新式类所需要的类Spam(object)派生将不再需要；在3.0之中，当前独立的“经典”和衍生的”新式”类都会自动视为当前所谓的新式类（参考第26章）。
- 在try语句中，except name, value:形式，将变成except name as value:（参考第27章）。
- 在raise语句中，raise E, V必须写成raise E(V)，从而明确地创建了实例（参考第27章）。
- 本书描述的with/as异常环境管理器功能已经启用了（参考第27章）。
- 所有用户定义和内置的异常都被识别成类，不再是字符串（参考第28章）。
- 用户定义的异常必须派生自内置的BaseException，也就是异常层次的根（Exception是其子类，作为你的根类足够了）；内置的StandardException类已被移除（参考第28章）。
- 标准库的包结构发生实质性的重新组织（参考Python 3.0版的发布注解）。

虽然这份清单乍一看有些令人害怕，但记住，本书说明的多数核心Python语言与Python 3.0都是完全相同的。事实上，上面多数的内容都是相当琐碎的细节，不会对程序员有太多影响。

此外，注意在本书编写时，这份清单仍然是在不确定的，最终可能不完整也不准确，所以一定要看看Python 3.0的发布注解，从而了解官方的说明。如果你已经使用Python 2.x版本系列编写代码了，也可以看一看“2to3”这个自动把Python 2.x转换为Python 3.0代码的转换脚本（Python 3.0会提供）。

www.TopSage.com

关于此系列

O'Reilly学习丛书，是面向任何想通过系统的学习来掌握新技能的读者所编写和设计的。本系列的每本书都使用了我们（通过你们的帮助）认为最佳的学习原理，让读者能学到新项目的知识，从而能够完成经理要求的出乎意料的任务，或者迅速地学习新语言。

要善用学习手册系列的任何一本书，建议读者依次读完每一章，就会发现，阅读每章的标题就可以对这一章的内容有一个概览。读者也可以使用摘要，事先看一看每章的重点，并复习已学过的内容。最后，为了帮助读者精通每一章的内容，我们还附加了一节头脑风暴，其中包含了习题。每一部分也有实用的练习题。

学习丛书会伴随在读者身旁，就像你值得信赖的同事或导师一样，我们努力要让读者的学习体验更为愉快。请告诉我们你的评价，无论是赞赏、批评或改进的建议，都可寄给*learning@oreilly.com*。

使用代码示例

本书的目的是帮助读者把工作做好。一般来说，读者可以在程序和文档中使用本书的代码，不需要联系我们取得许可，除非是要重新发布大量的代码。例如，编写程序时，使用本书好几段代码，就需要许可。销售和分发O'Reilly范例光盘也需要许可。引用本书和例子程序来回答问题，不需要许可。把本书大量例子程序整合到自己的产品文档中则需要许可。

虽然并非必须，但我们会感谢那些标明所有权的行为。所有权通常包括标题、作者、出版社以及ISBN。例如，“Learning Python, Third Edition, by Mark Lutz. Copyright 2008 O'Reilly Media Inc., 978-0-596-51398-6.”。

如果读者觉得对程序例子的运用超出合理使用或者上列许可情况之外，可以与我们联系：*permissions@oreilly.com*。

体例

下面是本书关于印刷字体方面的一些约定：

斜体（*Italic*）

用于电子信箱、URL、文件名、路径名以及用于强调第一次介绍的新的术语。

www.TopSage.com

等宽字体（Constant width）

用于文件内容以及命令输出，来表示模块、方法、语句以及命令。

定宽粗体（**Constant width bold**）

用于程序代码段，来显示应该由用户输入的命令或文字，有时则用于强调程序代码的一部分。

定宽斜体（*Constant width italic*）

用于程序代码段中可替换的部分以及一些注释。

<等宽字体>（<Constant width>）

表示应该以真实程序代码取代的语法单元。

注意：表示和附近文字相关的技巧、建议或一般性注释。

警告：表示和附近文字相关的警告和注意事项。

本书例子中，系统命令行开头的%字符指的是系统提示符，这得取决于读者的机器（例如，DOS窗口是C:\Python25>）。不要自行输入%字符。同样，在解释器交互模式下所列出的内容中，也不要输入每行开头的>>>和...字符，这些是Python显示的提示符。只要输入这些提示符之后的文字就行了。为了帮助你记住这一点，本书中的用户输入都将以粗体显示。此外，一般也不需要输入列表中以#开头的文字，这些是注释，不是可执行的代码。

联系方式

关于本书的批评建议和相关问题请使用如下地址与出版社联系：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）
奥莱利技术咨询（北京）有限公司

www.TopSage.com

我们为本书提供了一个网页，在此页面中我们列出了勘误表、示例以及所有其他信息。读者可以登录以下网址访问该页面：

http://www.oreilly.com/catalog/9780596513986（英文版）

http://www.oreilly.com.cn/book.php?bn=978-7-111-26776-8（中文版）

要对本书发表评论或咨询相关技术问题，请发电子邮件至：

bookquestions@oreilly.com

要了解我们的书籍、会议、资源中心以及O'Reilly Network的详细信息，请访问以下网址：

http://www.oreilly.com

http://www.oreilly.com.cn

书籍链接的更新，请参考本书前言提到的相关链接。

致谢

当我在2007年写本书第三版时，我总是抱着某种“完成任务”的心态。我已经使用并推广Python 15年了，而且也已经有10年Python的培训经验了。尽管随着时间和世事的流逝，我依然惊讶于Python这些年来的成功。Python的成长，是我们多数人在1992年时难以想象的。所以，也许我得冒着被当成无可救药、固执己见的作者的风险，但是请你谅解一下，我得在这里说一说回忆、恭贺以及感谢的话。

这是漫长而崎岖的道路。今日回首，1992年当我第一次发现Python的时候，我根本不知道它对我未来15年的生活会有什么影响。1995年编写《Programming Python》第一版后的两年，我开始在全国和全世界旅行，为初学者和专家培训Python。从1999年完成本书第一版后，我成为了全职、独立的Python培训师和作家；这得益于Python指数级增长的受欢迎程度。

我在2007年写下这些话时，已经编写了9本Python书籍；我培训Python已经超过10年了，而且在美国、欧洲、加拿大以及墨西哥教过200多个Python短期培训课程，在此过程中遇到了超过3 000位以上的学生。除了频繁地累计飞行里程，这些课程也帮助我精炼本书以及其他Python书籍的内容。几年下来，教学磨练了书籍，而书籍又反过来磨练了教学。事实上，你读的这本书的大部分内容都源于我的课程。

www.TopSage.com

因此，我想要感谢过去10年来参加我课程的所有学生。除了Python本身的变化之外，你们的反馈对本书的出版，也扮演了重要角色（没有比看3 000位学生重复犯初学者的错误更有启发性的事了）。本版的修正主要根据2003年后的课程，不过，从1997年起的每堂课，都对本书的精炼有或多或少的帮助。我要特别感谢那些在都柏林、墨西哥城、巴塞罗那、伦敦、埃德蒙顿以及波多黎各举办课程的那些客户；无法想象有更好的培训地点了。

我也想对每一位参与本书制作的人表示感谢。参与这个项目的编辑：这一版的Tatiana Apandi，以及前几版的许多人。感谢Liza Daly负责本书的技术校对。感谢O'Reilly让我有这个机会写出这9本书，真的很有趣（感觉上有点像电影《偷天情缘（Groundhog Day）》）。

我想感谢最初的共同作者David Ascher对本书前几版的帮助。David在前几版中贡献了“外层”部分，但是在本版中，为了让新的核心语言素材多一点空间，我们不得不忍痛割爱了。我在本版中加了一些更高级的程序，作为自学的最终练习，但是，这无法弥补删去的所有内容的缺憾。如果你难以忘怀那些内容，可以参考序中前边关于应用层面的内容的相关说明。

创造了如此有趣和实用的语言，我得特别感谢Guido van Rossum和Python社区的人们。就像多数开源系统一样，Python是许多英雄努力的结果。拥有了15年的Python编程经验，我依然觉得Python相当有趣。我特别荣幸看到了Python从脚本语言的初级阶段成长为广泛使用的工具，几乎每个编写软件的组织都以某种方式在部署使用它。这是令人兴奋的结果，我想感谢并祝贺整个Python社区，你们做了件很美妙的事。

我也想感谢在O'Reilly原先的编辑：已故的Frank Willison。这本书大部分都是Frank的想法，反映出他那充满感染力的愿景。回首往事，Frank对我的职业生涯以及Python本身都有很深远的影响。Python刚出现时拥有这么多的乐趣并如此成功，可以说，这都归功于Frank，一点都不夸张。我们还是很想念他。

最后，还有些人要感谢。感谢OQO这么好的玩具。感谢已故的Carl Sagan，让来自威斯康星州的18岁小伙子得到了启发。感谢Jon Stewart和Michael Moore这些爱国者。感谢我这几年遇到的所有大型公司，提醒我自己有多么的幸运，可以为自己打工。

感谢我的孩子Mike、Sammy以及Roxy，无论他们将来选择做什么。我开始用Python时，你们都还小，而你们似乎也这样长大了，我以你们为荣。生活会让我们一路走下去，但是总会有回家的路。

www.TopSage.com

最需要感谢的是Vera，我最好的朋友、女友以及妻子。我美妙的日子就是我终于遇见你的那一天。我不知道接下来50年会怎样，但我知道，我想把所有时间都用来拥抱你。

—Mark Lutz
Berthoud，Colorado
2007年7月

www.TopSage.com

大家网
Topsage.com

第一部分

使用入门

www.TopSage.com

第一部分

門人開發

www.TopSage.com

第1章

问答环节

如果你已经购买了本书，你也许已经知道Python是什么，也知道为什么Python是一个值得学习的重要工具。如果你还不知道，那么通过学习本书并完成一两个项目之后，你将会迷上Python。在详细学习之前，本书首先将会简要介绍一下Python流行背后的一些主要原因。为了引入Python的定义，本章将采用一问一答的形式，其内容将涵盖新手可能提出的一些最常见的问题。

人们为何使用Python

目前有众多可选的编程语言，这往往是入门者首先面对的问题。鉴于目前大约有1百万Python用户，的确没有办法完全准确地回答这个问题。开发工具的选择有时取决于特定的约束条件或者个人喜好。

然而，在过去的十年中，在对近200个团体组织和3000名学生的Python培训过程中，作者发现这个问题的答案具有一些共性。Python用户反映，之所以选择Python的主要因素有以几个方面。

软件质量

在很大程度上，Python更注重可读性、一致性和软件质量，从而与脚本语言世界中的其他工具区别开来。Python代码的设计致力于可读性，因此具备了比传统脚本语言更优秀的可重用性和可维护性。即使代码并不是亲手所写，Python的一致性也保证了其代码易于理解。此外，Python支持软件开发的高级重用机制。例如面向对象程序设计（OOP）。

开发者效率

相对于C、C++和Java等编译/静态类型语言，Python的开发者效率提高了数倍。Python代码的大小往往只有C++或Java代码的五分之一到三分之一。这就意味着

可以录入更少的代码、调试更少的代码并在开发完成之后维护更少的代码。并且Python程序可以立即运行，无需传统编译/静态语言所必须的编译及链接等步骤，进一步提高了程序员的效率。

程序的可移植性

绝大多数的Python程序不做任何改变即可在所有主流计算机平台上运行。例如，在Linux和Windows之间移植Python代码，只需简单地在机器间复制代码即可。此外，Python提供了多种可选的独立程序，包括用户图形界面、数据库接入、基于Web系统等。甚至包括程序启动和文件夹处理等操作系统接口，Python尽可能地考虑了程序的可移植性。

标准库的支持

Python内置了众多预编译并可移植的功能模块，这些功能模块称作标准库（standavd library）。标准库支持一系列应用级的编程任务，涵盖了从字符模式到网络脚本编程的匹配等的方面。此外，Python可通过自行开发的库或众多第三方的应用支持软件进行扩展。Python的第三方支持工具包括网站开发、数值计算、串口读写、游戏开发等各个方面。例如，NumPy被描述为一个免费的、如同Matlab一样功能强大的数值计算开发平台。

组件集成

Python脚本可通过灵活的集成机制轻松地与应用程序的其他部分进行通信。这种集成使Python成为产品定制和扩展的工具。如今，Python可以使用C和C++的库，可以被C和C++的程序调用，可以与Java组件集成，可以与COM和.NET等框架进行通信，并且可以通过SOAP、XML-RPC和CORBA等接口与网络进行交互。Python绝不仅仅是一个独立的工具。

享受乐趣

Python的易用性和强大内置工具使编程成为一种乐趣而不是琐碎的重复劳动。尽管这是一个难以捉摸的优点，但这将对开发效率的提升有很重要的帮助。

以上因素中，对于绝大多数Python用户而言，前两项（质量和效率）也许是Python最具吸引力的两个优点。

软件质量

从设计来讲，Python秉承了一种独特的简洁和可读性高的语法，以及一种高度一致的编程模式。正如最近的一次Python会议标语所宣称的那样， Python确确实实是“符合大脑思维习惯”的，即Python的特性是以统一并有限的方式进行交互，可以在一套紧凑的

www.TopSage.com

核心思想基础上进行自由发挥。这使Python易于学习、理解和记忆。事实上，Python程序员在阅读和编写代码时无需经常查阅手册。Python是一个设计风格始终如一的开发平台，可以保证开发出相当规范的代码。

从哲学理念上讲，Python采取了一种所谓极简主义的设计理念。这意味着尽管实现某一编程任务通常有多种方法，往往只有一种方法是显而易见的，还有一些不是那么明显的方法，以及少量风格一致的解决方法。此外，Python并不强制约束用户，当交互含糊不清时，明了的解决办法要优于“魔术”般的方法。在Python的思维方式中，明了胜于晦涩，简洁胜于复杂（注1）。

除了以上的设计主旨，Python还采用模块化设计、OOP在内的一些工具来提升程序的可重用性。由于Python致力于精益求精，Python程序员也都自然而然地秉承了这一理念。

开发者效率

20世纪90年代中后期互联网带来的信息爆炸，使有限的程序员难以应付繁多的软件开发项目，开发者往往要求以互联网演变一样的速度去开发系统。时过境迁，后信息爆炸时代带来了公司裁员和经济衰退。今天，往往要求程序员以更少的人力去实现相同的开发任务。

无论在以上哪种背景下，Python作为开发工具均以付出更少的精力完成更多的任务而脱颖而出。Python致力于为开发速度的最优化：简洁的语法、动态类型、无需编译、内置工具包等特性使程序员能够快速完成项目开发，而使用其他开发语言则需要几倍的时间。其最终结果就是相对于传统的语言Python把开发者效率提高了数倍。不管处于欣欣向荣还是萧条不景气的时代，无论软件行业将向何处发展，这都是一件值得庆幸的事。

Python是“脚本语言”吗

Python是一门多种用途的编程语言，时常在扮演脚本语言的角色。一般来说，Python可定义为面向对象的脚本语言：这个定义把对面向对象的支持和全面的面向脚本语言的角色融合在一起。事实上，人们往往以“脚本”而不是“程序”描述Python的代码文件。本书中，“脚本”与“程序”是可以相互替代的，其中“脚本”往往倾向于描述简单的顶层代码文件，而“程序”则用来描述那些相对复杂一些的多文件应用。

注1：了解完整的Python哲学理念，可以在任意一个Python交互解释器中键入import this命令。这是Python隐藏的一个彩蛋：描述了一系列Python的设计原则。如今已是Python社区内流行的行话“EIBTI”就是“明了胜于晦涩”这条规则的简写。

由于“脚本语言”从不同的视角来观察有着众多不同的意义,对于Python来讲并不是所有的都适合。实际上，人们往往给Python冠以以下三个不同的角色，其中有些相对其余的更重要。

Shell 工具

当人们听到Python被描述为脚本语言时，他们往往会想到Python是一个面向系统的脚本语言代码工具。这些程序往往从命令行运行，实现诸如文本文件的处理以及调用其他程序等任务。

Python程序当然能够以这样的角色工作，但这仅仅是Python常规应用范围的很小一部分。

控制语言

对其他人而言，脚本可定义为控制或重定向其他应用程序组件的“粘接”层。Python经常部署于大型应用的场合。例如，测试硬件器件时，Python程序可调用相关组件，通过组件在底层和器件之间进行交互。类似地，在终端用户产品定制的过程中，应用程序可以在策略点调用一些Python代码，而无需分发或重新编译整个系统代码。

Python的简洁使其从本质上能够成为一个灵活的控制工具。从技术上来讲，这基本上就是Python的常规角色；许多Python代码作为独立的脚本执行时无需调用或者了解其他的集成组件。然而，Python不单单是一种控制语言而已。

使用快捷

对于“脚本语言”最好的解释也许就是应用于快速编程任务的一种简单语言。对于Python来说，这确实是实至名归，因为Python与C++等类似的编译语言相比，大大提高了程序开发速度。其快速开发周期促进了探索、递增的编程模式，而这些都是必须亲身体验之后才能体会得到的。

但是千万别被迷惑，误以为Python仅可以实现简单的任务。恰恰相反，Python的易用性和灵活性使编程任务变得简单。Python有着一些简洁的特性，但是它允许程序按照需求以尽可能优雅的方式扩展。也正是基于这一点，它通常应用于快速作业任务和长期战略开发。

所以，Python是不是脚本语言呢？这取决于你在问谁。一般意义上讲，“脚本语言”一词可能最适用于描述一种Python所支持的快速和灵活的开发模式，而不是特定的应用领域的概念。

www.TopSage.com

大家网 TopSage.com

好吧，Python的缺点是什么呢

在经过15年的Python使用和10年Python的教学之后，我们发现Python唯一的缺点就是，在目前现有的实现方式下，与C和C++这类编译语言相比，Python的执行速度还不够快。

本书后文将对实现方式的概念进行详细阐述。简而言之，目前Python的标准实现方式是将源代码的语句编译（或者说是转换）为字节码的形式，之后再将字节码解释出来。由于字节码是一种与平台无关的格式，字节码具有可移植性。然而，因为Python没有将代码编译成底层的二进制代码（例如，Intel芯片的指令），一些Python程序将会比像C这样的完全编译语言慢一些。

程序的类型决定了是否需要关注程序的执行速度。Python已经优化过很多次，并且Python代码在绝大多数应用领域运行的速度也足够快。此外，一旦使用Python脚本做一些“现实”世界的事情，程序实际上是以C语言的速度运行的，例如，处理某一个文件或构建一个用户图形界面（GUI）。因为在这样的任务中，Python代码会立即发送至Python解释器内部已经被编译的C代码。究其根源，Python开发速度带来的效益往往比执行速度带来的损失更为重要，特别是在现代计算机的处理速度情况下。

即使当今CPU的处理速度很快，在一些应用领域仍然需要优化程序的执行速度。例如，数值计算和动画，常常需要其核心数值处理单元至少以C语言的速度（或更快）执行。如果在以上领域工作，通过分离一部分需要优化速度的应用，将其转换为编译好的扩展，并在整个系统中使用Python脚本将这部分应用连接起来，仍然可以使用Python。

本书我们将不会再谈论这个扩展的问题，但这却是一个我们先前所提到过的Python作为控制语言角色的鲜活例子。NumPy一个采用双语言混编策略的重要例子：一个Python的数值计算扩展，NumPy将Python变为一个高效并简单易用的数值计算编程工具。你也许不会在你自己的Python工作中采用这种扩展的方式编程，但是如果需要的话，Python也是能够提供这种强大的优化机制的。

如今谁在使用Python

在2007年编写本书的时候，乐观估计，这个时候全球的Python用户将达到1百万（略微有些出入）。这个估计是基于各种数据的统计的。例如，下载率和用户抽样调查。因为Python开放源代码，没有注册许可证总数的统计，就很难得到精确的用户总数。此外，在Linux的各种发行版、Macintosh计算机和其他的一些硬件和产品中自动内置了Python，进一步模糊了用户数目。

总体来说，Python从广泛的用户基础和活跃的开发者社区中受益不少。由于Python有近15年的发展历史并得到了广泛的应用，Python保持了稳定并具有活力的发展趋势。除了个人用户使用之外，Python也被一些公司应用于商业产品的开发上。例如：

- Google在其网络搜索系统中广泛应用了Python，并且聘用了Python的创作者。
- YouTube视频分享服务大部分是由Python编写的。
- 流行的P2P文件分享系统Bittorrent是一个Python程序。
- Intel、Cisco、Hewlett-Packard、Seagate、Qualcomm和IBM使用Python进行硬件测试。
- Industrial Light & Magic、Pixar等公司使用Python制作动画电影。
- 在经济市场预测方面，JPMorgan Chase、UBS、Getco和Citadel使用Python。
- NASA、Los Alamos、Fermilab、JPL等使用Python实现科学计算任务。
- IRobot使用Python开发了商业机器人真空吸尘器。
- ESRI在其流行的GIS地图产品中使用Python作为终端用户的定制工具。
- NSA在加密和智能分析中使用Python。
- IronPort电子邮件服务器产品中使用了超过100万行的Python代码实现其作业。
- OLPC使用Python建立其用户界面和动作模块。

还有许多方面都有Python的身影。如今贯穿所有使用Python的公司的唯一共同思路也许就是：Python在所有的应用领域几乎无所不能。Python的通用性使其几乎能够应用于任何场合，而不是只能在一处使用。实际上，我们这样说也不为过：无论是短期策略任务（例如，测试或系统管理），还是长期战略上的产品开发，Python已经证明它是无所不能的。

想要了解更多有关Python公司的现状，请访问Python的官方网站 *http://www.python.org*。

使用Python可以做些什么

Python不仅仅是一个设计优秀的程序语言，它能够完成现实中的各种任务，包括开发者们日复一日所做的事情。作为编制其他组件、实现独立程序的工具，它通常应用于各种领域。实际上，作为一种通用语言，Python的应用角色几乎是无限的：你可以在任何场合应用Python,从网站和游戏开发到机器人和航天飞机控制。

www.TopSage.com

尽管如此，Python的应用领域分为如下几类。下文将介绍一些Python如今最常见的应用领域，以及每个应用领域内所用的一些工具。我们不会对各个工具进行深入探讨，如果你对这些话题感兴趣，请从Python网站或其他一些资源中获取更多的信息。

系统编程

Python对操作系统服务的内置接口，使其成为编写可移植的维护操作系统的管理工具和部件（有时也被称为Shell工具）的理想工具。Python程序可以搜索文件和目录树，可以运行其他程序，用进程或线程进行并行处理等等。

Python的标准库绑定了POSIX以及其他常规操作系统（OS）工具：环境变量、文件、套接字、管道、进程、多线程、正则表达式模式匹配、命令行参数、标准流接口、Shell命令启动器、文件名扩展等。此外，很多Python的系统工具设计时都考虑了其可移植性。例如，复制目录树的脚本无需做任何修改就可以在几乎所有的Python平台上运行。

用户图形接口

Python的简洁以及快速的开发周期十分适合开发GUI程序。Python内置了TKinter的标准面向对象接口Tk GUI API，使Python程序可以生成可移植的本地观感的GUI。Python/Tkinter GUI不做任何改变就可以运行在微软Windows、X Windows（UNIX和Linux）以及Mac OS（Classic和OS X都支持）等平台上。一个免费的扩展包PMW，为Tkinter工具包增加了一些高级部件。此外，基于C++平台的工具包wxPython GUI API可以使用Python构建可移植的GUI。

诸如PythonCard和Dabo等一些高级工具包是构建在wxPython和Tkinter的基础API之上的。通过适当的库，你可以使用其他的GUI工具包，例如，Qt、GTK、MFC和Swing等。对于运行于浏览器中的应用或在一些简单界面的需求驱动下，Jython（Java版本的Python，我们将会在第2章中进行介绍）和Python服务器端CGI脚本提供了其他一些用户界面的选择。

Internet脚本

Python提供了标准Internet模块，使Python能够广泛地在多种网络任务中发挥作用，无论是在服务器端还是在客户端都是如此。脚本可以通过套接字进行通信；从发给服务器端的CGI脚本的表单中解析信息；通过URL获取网页；从获取的网页中解析HTML和XML文件；通过XML-RPC、SOAP和Telnet通信等。Python的库使这一切变得相当简单。

不仅如此，从网络上还可以获得很多使用Python进行Internet编程的第三方工具。例如，*HTMLGen*可以从Python类的描述中生成HTML文件，*mod_python*包可以使在Apache服务器上运行的Python程序更具效率并支持Python Server Page这样的服务器端模板，而且支持客户端运行的服务器端Applet。此外，Python涌现了许多Web开发工具包，例如，Django、TurboGears、Pylons、Zope和WebWare，使Python能够快速构建功能完善和高质量的网站。

组件集成

在介绍Python作为控制语言时，曾涉及它的组件集成的角色。Python可以通过C/C++系统进行扩展，并能够嵌套C/C++系统的特性，使其能够作为一种灵活的粘合语言，脚本化处理其他系统和组件的行为。例如，将一个C库集成到Python中，能够利用Python进行测试并调用库中的其他组件；将Python嵌入到产品中，在不需要重新编译整个产品或分发源代码的情况下，能够进行产品的单独定制。

为了在脚本中使用，在Python连接编译好组件时，例如，SWIG和SIP这样的代码生成工具可以让这部分工作自动完成。更大一些的框架，例如，Python的微软Windows所支持的COM、基于Java实现的Jython、基于.NET实现的IronPython和各种CORBA工具包，提供了多种不同的脚本组件。例如，在Windows中，Python脚本可利用框架对微软Word和Excel文件进行脚本处理。

数据库编程

对于传统的数据库需求，Python提供了对所有主流关系数据库系统的接口，例如，Sybase、Oracle、Informix、ODBC、MySQL、PostgreSQL、SQLite。Python定义了一种通过Python脚本存取SQL数据库系统的可移植的数据库API，这个API对于各种底层应用的数据库系统都是统一的。例如，因为厂商的接口实现为可移植的API，所以一个写给自由软件MySQL系统的脚本在很大程度上不需改变就可以工作在其他系统上（例如，Oracle）——你仅需要将底层的厂商接口替换掉就可以实现。

Python标准的pickle模块提供了一个简单的对象持久化系统：它能够让程序轻松地将整个Python对象保存和恢复至文件和文件类的对象中。在网络上，同样可以找到名叫ZODB的第三方系统，它为Python脚本提供了完整的面向对象数据库系统，系统SQLObject可以将关系数据库映射至Python的类模块。并且，从Python 2.5版本开始，SQLite已经成为Python自带标准库的一部分了。

www.TopSage.com

快速原型

对于Python程序来说，使用Python或C编写的组件看起来都是一样的。正因为如此，我们可以在一开始利用Python做系统原型，之后再将组件移植到C或C++这样的编译语言上。和其他的原型工具不同，当原型确定后，Python不需要重写。系统中不需要像C++这样执行效率的部分可以保持不变，从而使维护和使用变得轻松起来。

数值计算和科学计算编程

我们之前提到过的NumPy数值编程扩展包括很多高级工具，例如，矩阵对象、标准数学库的接口等。通过将Python与出于速度考虑而使用编译语言编写的数值计算的常规代码进行集成，NumPy将Python变成一个缜密严谨并简单易用的数值计算工具，这个工具通常可以替代已有的代码，而这些代码都是用FORTRAN或C++等编译语言编写的。其他一些数值计算工具为Python提供了动画、3D 可视化、并行处理等功能的支持。

游戏、图像、人工智能、XML、机器人等

Python的应用领域很多，远比本书提到的多得多。例如，可以利用*pygame*系统使用Python对图形和游戏进行编程；用PIL和其他的一些工具进行图像处理；用PyRo工具包进行机器人控制编程；用`xml`库、`xmlrpclib`模块和其他一些第三方扩展进行XML解析；使用神经网络仿真器和专业的系统shell进行AI编程；使用*NLTK*包进行自然语言分析；甚至可以使用PySol程序下棋娱乐。可以从Vaults of Parnassus以及新的PyPI网站（请在Google或 *http://www.python.org*上获得具体链接）找到这些领域的更多支持。

一般来说，这些特定领域当中有许多在很大程度上都是Python组件集成角色的再次例证。采用C这样的编译语言编写库组件，增加Python至其前端，这样的方式使Python在不同领域广泛地发挥其自身价值。对于一种支持集成的通用型语言，Python的应用极其广泛。

Python有哪些技术上的优点

显然，这是开发者关心的问题。如果你目前还没有程序设计背景，接下来的这些章节可能会显得有些令人费解：别担心，在本书中我们将会对这些内容逐一做出详细解释。那么对于开发者来说，这将是对Python一些最优的技术特性的快速介绍。

www.TopSage.com

面向对象

从根本上讲，Python是一种面向对象的语言。它的类模块支持多态、操作符重载和多重继承等高级概念，并且以Python特有的简洁的语法和类型，OOP十分易于使用。事实上，即使你不懂这些术语，仍会发现学习Python比学习其他OOP语言要容易得多。

除了作为一种强大的代码构建和重用手段以外，Python的OOP特性使它成为面向对象系统语言如C++和Java的理想脚本工具。例如，通过适当的粘接代码，Python程序可以对C++、Java和C#的类进行子类的定制。

OOP是Python的一个选择而已，这一点非常重要。不必强迫自己立马成为一个面向对象高手，你同样可以继续深入学习。就像C++一样，Python既支持面向对象编程也支持面向过程编程的模式。如果条件允许的话，其面向对象的工具即刻生效。这对处于预先设计阶段的策略开发模式十分有用。

免费

Python的使用和分发是完全免费的。就像其他的开源软件一样，例如，Tcl、Perl、Linux和Apache。你可以从Internet上免费获得Python系统的源代码。复制Python，将其嵌入你的系统或者随产品一起发布都没有任何限制。实际上，如果你愿意的话，甚至可以销售它的源代码。

但请别误会："免费"并不代表"无支持"。恰恰相反，Python的在线社区对用户需求的响应和商业软件一样快。而且，由于Python完全开放源代码，提高了开发者的实力，并产生了一个很大的专家团队。尽管学习研究或改变一个程序语言的实现并不是对每一个人来说都那么有趣，但是当你知道还有源代码作为最终的帮助和无尽的文档资源是多么的令人欣慰。你不需要去依赖商业厂商。

Python的开发是由社区驱动的，是Internet大范围的协同合作努力的结果。这个团体包括Python的创始者Guido van Rossum：Python社区内公认的"终身的慈善独裁者"[Benevolent Dictator for Life (BDFL)]。Python语言的改变必须遵循一套规范的有约束力的程序（称作PEP流程），并需要经过规范的测试系统和BDFL进行彻底检查。值得庆幸的是，正是这样使得Python相对于其他语言可以保守地持续改进。

可移植

Python的标准实现是由可移植的ANSI C 编写的，可以在目前所有的主流平台上编译和

www.TopSage.com

运行。例如，如今从PDA到超级计算机，到处可以见到Python在运行。Python可以在下列平台上运行（这里只是部分列表）：

TopSage.com

- Linux和UNIX系统。
- 微软Windows和DOS（所有版本）。
- Mac OS（包括OS X 和 Classic）。
- BeOS、OS/2、VMS和QNX。
- 实时操作系统，例如，VxWorks。
- Cray超级计算机和IBM大型机。
- 运行Palm OS、PocketPC和Linux的PDA。
- 运行Windows Mobile和Symbian OS 的移动电话。
- 游戏终端和iPod。
- 还有更多。

除了语言解释器本身以外，Python发行时自带的标准库和模块在实现上也都尽可能地考虑到了跨平台的移植性。此外，Python程序自动编译成可移植的字节码，这些字节码在已安装兼容版本Python的平台上运行的结果都是相同的。

这些意味着Python程序的核心语言和标准库可以在Linux、Windows和其他带有Python解释器的平台无差别的运行。大多数Python 外围接口都有平台相关的扩展（例如，COM支持Windows），但是核心语言和库在任何平台都一样。就像之前我们提到的那样，Python还包含了一个叫做Tkinter的Tk GUI工具包，它可以使Python程序实现功能完整的无需做任何修改即可在所有主流GUI平台运行的用户图形界面。

功能强大

从特性的观点来看，Python是一个混合体。它丰富的工具集使它介于传统的脚本语言（例如，Tcl、Scheme和Perl）和系统语言(例如，C、C++和Java)之间。Python提供了所有脚本语言的简单和易用性，并且具有在编译语言中才能找到的高级软件工程工具。不像其他脚本语言，这种结合使Python在长期大型的开发项目中十分有用。下面是一些Python工具箱中的工具简介。

动态类型

Python在运行过程中随时跟踪对象的种类，不需要代码中关于复杂的类型和大小的

www.TopSage.com

声明。事实上，你将在第6章中看到，Python中没有类型或变量声明这回事。因为Python代码不是约束数据的类型，它往往自动地应用了一种广义上的对象。

自动内存管理

Python自动进行对象分配，当对象不再使用时将自动撤销对象（“垃圾回收”），当需要时自动扩展或收缩。Python能够代替你进行底层的内存管理。

大型程序支持

为了能够建立更大规模的系统，Python包含了模块、类和异常等工具。这些工具允许你组织系统为组件，使用OOP重用并定制代码，并以一种优雅的方式处理事件和错误。

内置对象类型

Python提供了常用的数据结构作为语言的基本组成部分。例如，列表（list）、字典（dictionary）、字符串（string）。我们将会看到，它们灵活并易于使用。例如，内置对象可以根据需求扩展或收缩，可以任意地组织复杂的信息等。

内置工具

为了对以上对象类型进行处理，Python自带了许多强大的标准操作，包括合并（concatenation）、分片（slice）、排序（sort）和映射（mapping）等。

库工具

为了完成更多特定的任务，Python预置了许多预编码的库工具，从正则表达式匹配到网络都支持。Python的库工具在很多应用级的操作中发挥作用。

第三方工具

由于Python是开放源代码的，它鼓励开发者提供Python内置工具之外的预编码工具。从网络上，可以找到COM、图像处理、CORBA ORB、XML、数据库等很多免费的支持工具。

除了这一系列的Python工具外，Python保持了相当简洁的语法和设计。综合这一切得到的就是一个具有脚本语言所有可用性的强大编程工具。

可混合

Python程序可以以多种方式轻易地与其他语言编写的组件“粘接”在一起。例如，Python的C语言API可以帮助Python程序灵活地调用C程序。这意味着可以根据需要给Python程序添加功能，或者在其他环境系统中使用Python。

www.TopSage.com

例如，将Python与C或者C++写成的库文件混合起来，使Python成为一个前端语言和定制工具。就像之前我们所提到过的那样，这使Python成为一个很好的快速原型工具；首先出于开发速度的考虑，系统可以先使用Python实现，之后转移至C，根据不同时期性能的需要逐步实现系统。

使用简单

运行Python程序，只需要简单地键入Python程序并运行就可以了。不需要其他语言（例如，C或C++）所必须的编译和链接等中间步骤。Python可立即执行程序，这形成了一种交互式编程体验和不同情况下快速调整的能力，往往在修改代码后能立即看到程序改变后的效果。

当然，开发周期短仅仅是Python易用性的一方面的体现。Python提供了简洁的语法和强大的内置工具。实际上，Python曾有种说法叫做“可执行的伪代码”。由于它减少了其他工具常见的复杂性，当实现相同的功能时，用Python程序比采用C、C++和Java编写的程序更为简单、小巧，也更灵活。

Python是工程，不是艺术

当Python于20世纪90年代初期出现在软件舞台上时，曾经引发其拥护者和另一个受欢迎脚本语言Perl的拥护者之间的冲突，但现今已成为经典的争论。我们认为今天这种争论令人厌倦，也没有根据，开发人员都很聪明，可以找到他们自己的结论。然而，这是我在培训课程上时常被问到的问题之一，所以在此对这个话题说几句话，似乎是合适的。

故事是这样的：你可以用Python做到一切用Perl能做到的事，但是，做好之后，还可以阅读自己的程序代码。就是因为这样，两者的领域大部分重叠，但是，Python更专注于产生可读性的代码。就大多数人而言，Python强化了可读性，转换为了代码可重用性和可维护性，使得Python更适合用于不是写一次就丢掉的程序。Perl程序代码很容易写，但是很难读。由于多数软件在最初的创建后都有较长的生命周期，所以很多人认为Python是一种更有效的工具。

这个故事反应出两个语言的设计者的背景，并体现出了人们选择使用Python的一些主要原因。Python的创立者所受的是数学家的训练，因此，他创造出来的语言具有高度的统一性，其语法和工具集都相当一致。再者，就像数学一样，其设计也具有

—待续—

正交性（orthogonal），也就是这门语言大多数组成部分都遵循一小组核心概念。例如，一旦掌握Python的多态，剩下的就只是细节而已。

与之相对比，Perl语言的创立者是语言学家，而其设计反应了这种传统。Perl中，相同任务有很多方式可以完成，并且语言材料的交互对背景环境敏感，有时还有相当微妙的方式，就像自然语言那样。就像著名的Perl所说的格言："完成的方法不止一种。"有了这种设计，Perl语言及其用户社群在编写代码时，就一直在鼓励表达式的自由化。一个人的Perl代码可能和另一个人的完全不同。事实上，编写独特、充满技巧性的代码，常常是Perl用户之间的骄傲来源。

但是，任何做过任何实质性的代码维护工作的人，应该都可以证实，表达式自由度是很棒的艺术，但是，对工程来说就令人厌恶了。在工程世界中，我们需要最小化功能集和可预测性。在工程世界中，表达式自由度会造成维护的噩梦。不止一位Perl用户向我们透漏过，太过于自由的结果通常就是程序很容易重头写起，但修改起来就不是那么容易了。

考虑一下：当人们在作画或雕塑时，他们是为自己做，为了纯粹美学考虑。其他人日后去修改图画或雕像的可能性很低。这是艺术和工程之间关键的差异。当人们在编写软件时，他们不是为自己写。事实上，他们甚至不是专门为计算机写的。而实际上，优秀的程序员知道，代码是为下一个会阅读它而进行维护或重用的人写的。如果那个人无法理解代码，在现实的开发场景中，就毫无用处了。

这就是很多人认为Python最有别于Perl这类描述语言的地方。因为Python的语法模型几乎会强迫用户编写可读的代码，所以Python程序会引导他们往完整的软件开发循环流程前进。此外，因为Python强调了诸如有限互动、统一性、规则性以及一致性这些概念，因此，会更进一步促进代码在首次编写后能够长期使用。

长期以来，Python本身专注于代码质量，提高了程序员的生产力，以及程序员的满意度。Python程序员也变得富有创意，以后就知道，语言本身的确对某些任务提供了多种解决办法。不过，本质上，Python鼓励优秀的工程的方式，是其他脚本语言通常所不具备的。

至少，这是许多采用Python的人之间所具有的共识。当然，你应该要自行判断这类说法，也就是通过了解Python提供了什么给你。为了帮助你们入门，让我们进行下一章的学习吧。

www.TopSage.com

简单易学

这一部分引出了本书的重点：相对于其他编程语言，Python语言的核心是惊人的简单易学。实际上，你可以在几天内（如果你是有经验的程序员，或许只需要几个小时）写出不错的Python代码。这对于那些想学习语言可以在工作中应用的专业人员来说是一个好消息，同样对于那些使用Python进行定制或控制系统的终端用户来说也是一个好消息。如今，许多系统依赖于终端用户可以很快地学会Python以便定制其代码的外围工具，从而提供较少的支持甚至不提供支持。尽管Python还是有很多高级编程工具，但不论对初学者还是行家高手来说，Python的核心语言仍是相当简单的。

名字来源于Monty Python

Python名字的来源这不算是一项技术，但是，这似乎是令人很惊讶、保护得很好的秘密，而我们希望把它全盘托出。尽管Python世界中都是蟒蛇的图标，但事实是，Python创立者Guido van Rossum是以BBC喜剧Monty Python s Flying Circus来命名的。他是Monty Python的大影迷，而很多软件开发人员也是（事实上，这两个领域似乎有种对称）。

这给Python代码的例子加入一种幽默的特质。比如，一般来说，传统常规的变量名为“foo”和“bar”，在Python的世界中变成了“spam”和“eggs”。而有时出现的“Brian”、“ni”、“shrubbery”等也是这样来的。这种方式甚至很大程度上影响了Python社区：Python会议上的演讲往往叫做“The Spanish Inquisition”。

当然，如果你熟悉这个幽默剧的话，所有这些你都会觉得很有趣，否则就没那么有意思了。你没有必要为了理解引自Monty Python（也许本书中你就会找到）的例子而刻意去熟悉这一串剧情，但是至少你现在应该知道它们的来源。

Python和其他语言比较起来怎么样

最后，用你也许已经知道的术语来说，人们往往将Python与诸如Perl、Tcl和Java这样的语言相比较。我们之前已经介绍过性能，那么这里我们重点谈一下功能。当其他语言也是我们所知道的并正在使用的有力工具的同时，人们认为Python：

- 比Tcl强大。Python支持“大规模编程”，使其适宜于开发大型系统。
- 有着比Perl更简洁的语法和更简单的设计，这使得Python更具可读性、更易于维护，有助于减少程序bug。

- 比Java更简单、更易于使用。Python是一种脚本语言，Java从C++这样的系统语言中继承了许多语法和复杂性。
- 比C++更简单、更易于使用，但通常也不与C++竞争。因为Python作为脚本语言，常常扮演多种不同的角色。
- 比Visual Basic更强大也更具备跨平台特性。由于Python是开源的，也就意味着它不可能被某一个公司所掌控。
- 比Ruby更成熟、语法更具可读性。与Ruby和Java不同的是，OOP对于Python是可选的：这意味着Python不会强制用户或项目选择OOP进行开发。
- 具备SmallTalk和Lisp等动态类型的特性，但是对开发者及定制系统的终端用户来说更简单，也更接近传统编程语言的语法。

特别对不仅仅做文本文件扫描还有也许未来会被人们读到（或者说你）的程序而言，很多人会发现Python比目前任何的可用的脚本或编程语言都划得来。不仅如此，除非你的应用要求最尖端的性能，Python往往是C、C++和Java等系统开发语言的一个不错的替代品：Python将会减少很多编写、调试和维护的麻烦。

当然，本书的作者从1992年就已经是Python的正式传道士了，所以尽可能接受这些意见吧。然而，所有这些的确反映出许多花费了时间精力来探索Python的开发者们的共同经验。

本章小结

以上是本书的概述部分。本章我们已经探索了人们为何选择Python完成他们编程任务的原因，也看到了它实现起来的效果以及当前一些具有代表性的使用Python的鲜活例子。然而我们的目标是教授Python，而不是销售它。最好的一种判断语言的方法就是在实践中使用它，所以本书的其余部分将把注意力集中到我们已经在这里简要介绍过的那些语言的细节之上。

为了入门，接下来两章将进行语言的技术介绍。我们将研究如何运行Python程序，窥视Python字节码执行的模式并介绍保存代码的模块文件的基本概念。目标就是让你能够运行本书其他部分的例子和练习。直到第4章我们才会开始真正的编程，但在此之前，请确保你已经掌握了继续深入学习的细节。

www.TopSage.com

大家网
TopSage.com

本章习题

本书的第三版，我们每一章都会以一个快速的小测验作为结束，测验包含了这一章介绍的内容，从而帮助你复习这些关键概念。而问题的答案见附录B，建议你独立完成测试后马上参考答案。除了这些每章结尾的测验以外，你还会在本书每一部分的结尾找到一些实验作业，这些作业是为了帮助你自己动手用Python进行编程而设计的。好了，这就是你的第一次测验。祝你好运！

1. 人们选择Python的六个主要原因是什么？
2. 请列举如今正在使用Python的四个著名的公司和组织的名称。
3. 出于什么样的原因会让你在应用中不使用Python呢？
4. 你可以用Python做什么？
5. 在Python中`import this`有什么意义？
6. 为什么“spam”在书中和网络上的Python例子中如此频繁地出现？
7. 你最喜欢哪个颜色？

习题解答

做的怎么样？这是本书提供的答案，当然，测验中一些问题的答案并不唯一。再一次强调，尽管你确定你的回答是正确的，本书还是建议你参考一下答案中提供的内容。如果这些答案对于你来说不合理的话，可以在本章节的内容中找到详细的信息。

1. 软件质量、开发者效率、程序的可移植性、标准库的支持、组件集成和享受乐趣。其中，质量和效率这两条是人们选择Python的主要原因。
2. Google、Industrial Light & Magic、Jet Propulsion Labs和ESRI等。做软件开发的所有组织几乎都流行使用Python,无论是长期战略产品开发还是测试或系统管理这样的短期策略任务都广泛采用了Python。
3. Python的缺点是它的性能：它不像C和C++这类常规的编译语言运行得那么快。另一方面，它对于绝大多数应用已经足够快了，并且典型的Python代码运行起来速度接近C，因为在Python解释器中调用链接了C代码。如果速度要求很苛刻的话，应用的数值处理部分可以采用编译好的扩展以满足应用要求。

www.TopSage.com

4. 你几乎可以在计算机上的任何方面使用Python：从网站和游戏开发到机器人和航天飞机控制。

5. `import this`会触发Python内部的一个彩蛋，它将显示Python语言层面之下的设计哲学。下一章你将会学习如何使用这条命令。

6. “Spam”引用自著名的幽默剧Monty Python，其中人们试着在露天咖啡馆点餐，却被维京人组成的一个合唱团唱着的有关spam的歌声给淹没了。哦，这也是Python脚本中的一个常规的变量名……

7. 蓝色。不，是黄色！

www.TopSage.com

第2章

Python如何运行程序

本章和下一章将给出程序执行的简要说明：应该如何开始编码代码以及Python如何运行代码。这一章我们将要学习Python解释器。第3章将会介绍如何编写程序以及如何运行。首先介绍的内容无疑与平台相关，本章有些内容也许并不适合你目前工作的平台，所以当你觉得所讲的内容与希望使用的平台不相关的话，你可以放心地跳过这些内容。同样，对于一些高端用户的读者，也许过去已经使用过类似的工具并希望快点尝尝Python的甜头，也许可以保留这一章的内容“以备以后参考”。对于其他的读者，让我们来学习如何运行程序代码吧。

Python解释器简介

迄今为止，我大多数时候都是将Python作为一门编程语言来介绍的。但是，从目前的实现上来讲，Python也是一个名为解释器的软件包。解释器是一种让其他程序运行起来的程序。当你编写了一段Python程序，Python解释器将读取程序，并按照其中的命令执行，得出结果。实际上，解释器是代码与机器的计算机硬件之间的软件逻辑层。

当Python包安装在机器上后，它包含了一些最小化的组件：一个解释器和支持的库。根据使用情况的不同，Python解释器可能采取可执行程序的形式，或是作为链接到另一个程序的一系列库。依照选用的Python版本的不同，解释器本身可以用C程序实现，或一些Java类实现，或者其他的形式。无论采取何种形式，编写的Python代码必须在解释器中运行。当然，为了实现这一点，首先必须要在计算机上安装Python解释器。

根据平台的不同Python的安装细节也不同，若想深入了解，请参照附录A。简而言之：

- Windows用户可通过获取并运行自安装的可执行文件，把Python安装到自己的机器上。双击后在所有的弹出提示框中选择“是”或“继续”即可。

www.TopSage.com

- 在Windows Vista，你也许需要使用Python 2.5 MSI 安装文件中的额外步骤；更多细节请参照附录A。
- Linux和Mac OS X 用户也许已经拥有了一个可用的Python预先安装在了电脑上：如今Python已成为这些平台的标准组件。
- 一些Linux用户（和大多数UNIX用户一样）可专门通过RPM文件安装Python或从源代码分发包中编译安装。
- 其他平台有着对应其平台的不同的安装技术。例如，Python可以在移动电话、游戏终端和iPod上应用，但是由于其安装方法的差异很大，在这里就不详细介绍了。

Python可以通过Python官方网站下载获得，也可以在其他的一些发布网站上找到。记住应该在安装Python之前确认Python是否已经安装。如果是在Windows上工作，一般可以在开始菜单中寻找Python，如图2-1所示（这些菜单的选项将在下一章进行讨论）。在UNIX或Linux上，Python也许在*/usr*目录下。

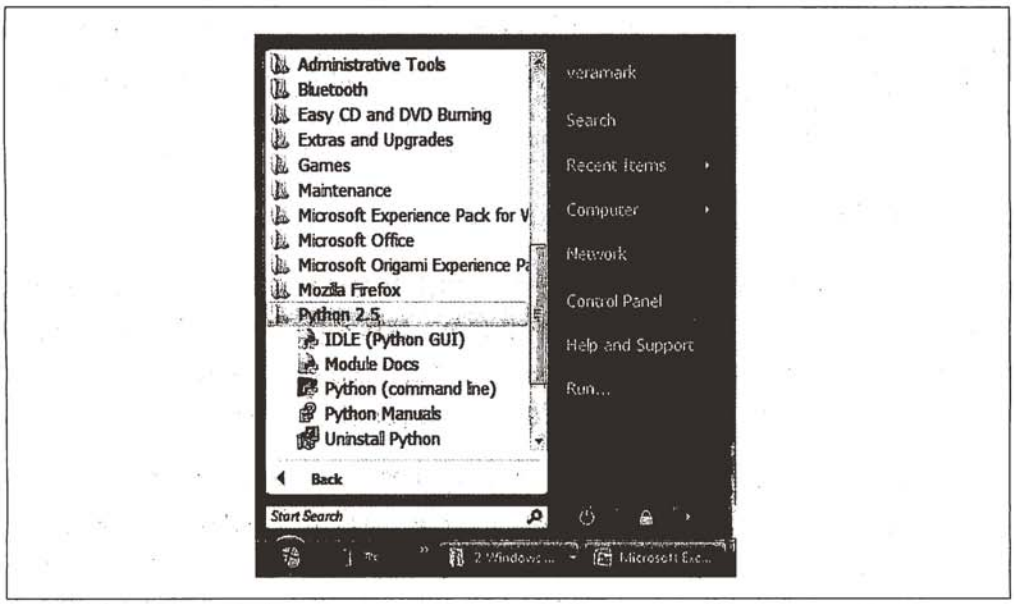


图2-1：当已经在Windows上安装好Python时，这就是Python在开始菜单中显示的情况。也许在不同的版本之间会有少许不同，但是IDLE可以提供开发的GUI，而Python可开始一个简单的交互会话。并且，这里有一些标准的手册，以及Pydoc的文档引擎（Docs模块）

由于安装细节具有很大的平台相关性，所以我们对这部分的讨论就此结束。若想获得更多安装过程的细节，请参考附录A。为了继续本章及下一章的内容，假设你已经安装好Python，并准备开始下一步的学习了。

www.TopSage.com

程序执行

编写或运行Python脚本的意义是什么呢？这取决于你是从一个程序员还是Python解释器的角度去看待这个问题。无论从哪一个角度看问题，这都会给你提供一个观察Python编程的重要视角。

程序员的视角

就最简单的形式而言，一个Python程序仅是一个包含Python语句的文本文件。例如，下面这个命名为*script1.py*的文件，是我们能够想到的最简单的Python脚本，但它算得上是一个典型的Python程序：

```
print 'hello world'
print 2 ** 100
```

这个文件包含了两个Python打印语句，在输出流中简单地打印一个字符串（引号中的文字）和一个数学表达式的结果（2的100次方）。不用为这段代码中的语法担心，我们这一章的重点只是程序的运行。本书的后面章节将会解释print语句，以及为什么可以计算2的100次方而不溢出。

你可以用任何自己喜欢的文本编辑器建立这样的文件语句。按照惯例，Python文件是以.py结尾的。从技术上来讲，这种命名方案在被“导入”时才是必须的，这也将在本书后边进行介绍，但是绝大多数Python文件为了统一都是以*.py*命名的。

当你将这些语句输入到文本文件后，你必须告诉Python去执行这个文件：也就是说，从头至尾按照顺序一个接一个地运行文件中的语句。正如下一章你将会看到的那样，可以通过命令行、点击其图标或者其他标准技术来运行Python程序。如果顺利的话，当执行文件时，将会看到这两个打印语句的结果显示在电脑屏幕的某处：一般默认是显示在运行程序的那个窗口。

```
hello world
1267650600228229401496703205376km
```

例如，这就是我在一个Windows笔记本电脑的DOS命令行（通常称为命令提示符窗口，可以在程序菜单的附件中找到）运行这个脚本的结果，需要保证不会有任何愚蠢的打字错误。

```
D:\temp> python script1.py
hello world
126765060022822940149670320537 6
```


www.TopSage.com

我们刚刚运行了一个打印字符串和数字的Python脚本。也许不会因为这个代码获得任何编程大奖，但是这对掌握一些程序执行的基本概念已经足够了。

Python的视角

前一节的简要介绍对于脚本语言来说，是相当标准的，并且往往绝大多数Python程序员只需要知道这些就足够了。在文本文件中输入代码，之后在解释器中运行这些代码。然而，当Python“运行”时，透过表面，还有一些事情发生。尽管了解Python内部并不是Python编程所必须的要求，然而对Python的运行实时结构有一些基本的了解可以帮助你从宏观上掌握程序的执行。

当Python运行脚本时，在代码开始进行处理之前，Python还会执行一些步骤。确切地说，第一步是编译成所谓的“字节码”，之后将其转发到所谓的“虚拟机”中。

字节码编译

当程序执行时，Python内部（对大多数用户是完全隐藏的）会先将源代码（文件中的语句）编译成所谓字节码的形式。编译是一个简单的翻译步骤，而且字节码是源代码底层的、与平台无关的表现形式。概括地说，Python通过把每一条源语句分解为单一步骤来将这些源语句翻译成一组字节码指令。这些字节码可以提高执行速度：比起文本文件中原始的源代码语句，字节码的运行速度要快得多。

你会注意到，前面一段所提到的这个过程对于你来说完全是隐藏起来的。如果Python进程在机器上拥有写入权限，那么它将把程序的字节码保存为一个以*.pyc*为扩展名的文件（“.pyc”就是编译过的“.py”源代码）。当程序运行之后，你会在那些源代码的附近（也就是说同一个目录下）看到这些文件。

Python这样保存字节码是作为一种启动速度的优化。下一次运行程序时，如果你在上次保存字节码之后没有修改过源代码的话，Python将会加载*.pyc*文件并跳过编译这个步骤。当Python必须重编译时，它会自动检查源文件和字节码文件的时间戳：如果你又保存了源代码，下次程序运行时，字节码将自动重新创建。

如果Python无法在机器上写入字节码，程序仍然可以工作：字节码将会在内存中生成并在程序结束时简单地丢弃（注1）。尽管这样，由于*.pyc*文件能够加速启动，你最好保

注1：从严格的意义上讲，只有文件导入的情况下字节码才保存，并不是对顶层文件。我们将会在第3章以及第5部分探讨有关导入的内容。当在交互提示模式下所录入的代码也不会保存为字节码，我们将在第3章予以说明。

www.TopSage.com

证在大型程序中可以写入。字节码文件同样是分发Python程序的方法之一：如果Python找到的都是.pyc文件，它也很乐意于运行这个程序，尽管这里没有原始的.py源代码文件（参考本章“冻结二进制”获得其他发布的选项）。

TopSage.com

Python虚拟机（PVM）

一旦程序编译成字节码（或字节码从已经存在的.pyc文件中载入），之后的节码被发送到通常称为Python虚拟机（Python Virtual Machine，简写为PVM）上来执行。PVM听起来比它本身给人的印象更深刻一些。实际上，它不是一个独立的程序，不需要安装。事实上，PVM就是迭代运行字节码指令的一个大循环，一个接一个地完成操作。PVM是Python的运行引擎，它时常表现为Python系统的一部分，并且它是实际运行脚本的组件。从技术上讲，它才是所谓“Python解释器”的最后步。

图2-2描述这里介绍的运行时的结构。请记住所有的这些复杂性都是有意识地对Python程序员隐藏起来的。字节码的编译是自动完成的，而且PVM也仅仅是安装在机器上的Python系统的一部分。再一次说明，程序员只需简单地编写代码并运行包含有语句的文件。

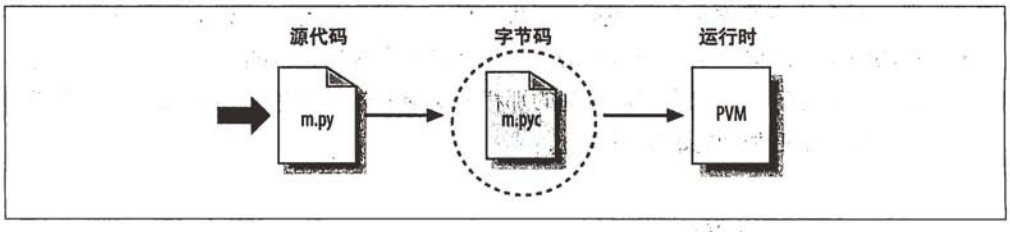


图2-2：Python的传统运行执行模式：录入的源代码转换为字节码，之后字节码在Python虚拟机中运行。代码自动被编译，之后再解释

性能的含义

熟悉C和C++这类完全编译语言的读者或许已经发现了Python模式中的一些不同之处。其中一个是，在Python的工作中通常没有“build”或“make”的步骤：代码在写好之后立即运行。另外一个就是，Python字节码不是机器的二进制代码(例如，Intel芯片的指令)。字节码是Python定义的一种表现形式。

这就是Python代码无法运行得像C或C++代码一样快的原因，就像第1章描述的那样：PVM循环（而不是CPU芯片）仍然须解释字节码，并且字节码指令与CPU指令相比需要更多的工作。另一方面，和其他经典的解释器不同，这里仍有内部的编译步骤：Python

并不需要反复地重分析和重分解每一行语句。实际的效果就是纯Python代码的运行速度介于传统的编译语言和传统的解释语言之间。更多关于Python性能的描述请参看第1章。

开发的含义

Python执行模块的另一个情况是其开发和执行的环境实际上并没有区别。也就是说，编译和执行源代码的系统是同一个系统。这种相似性对于拥有传统编译语言背景的读者来说，更有意义，然而在Python中，编译器总是实时出现，并且是运行程序系统的一部分。

使开发周期大大缩短。在程序开始执行之前不需要预编译和连接；只需要简单地输入并运行代码即可。这同样使Python具有更多的动态语言特性：在运行时，Python程序去构建并执行另一个Python程序是有可能的，而且往往是非常方便的。例如，eval和exec内置模块，能够接受并运行包含Python程序代码的字符串。这种结构是Python能够实现产品定制的原因：因为Python代码可以动态地修改，用户可以改进系统内部的Python部分，而不需要拥有或编译整个系统的代码。

从更基础的角度来说，牢记我们在Python中真正拥有的只有实时性：完全不需要初始的编译阶段，所有的事情都是在程序运行时发生的。这甚至还包括了建立函数和类的操作以及连接的模块。这些事情对于静态语言往往是发生在执行之前的，而在Python中是与程序的执行同时进行的。就像我们看到的那样，实际的效果就是Python比一些读者所用的程序语言带来了更加动态的编程体验。

执行模块的变种

在继续学习之前，应该指出前一节所介绍的内部执行流程反映了如今Python的标准实现形式，并且这实际上并不是Python语言本身所必需的。正是因为这一点，执行模块也在随时间而演变。事实上，从某种意义上讲有些系统已经改进了图2-2所描述的情况。让我们花些时间探索一下这些变化中最显著的改进吧。

Python实现的替代者

事实上，在编写本书的过程中，Python语言有三种主要实现方式（*CPython*、*Jython*和*IronPython*）以及一些次要的实现方式，例如，*Stackless Python*。简要地说，CPython是标准的实现；其他的都是有特定的目标和角色的。所有的这些都用来实现Python语言，只是通过不同的形式执行程序而已。

www.TopSage.com

CPython

和Python的其他两种实现方式相比，原始的、标准的Python实现方式通常称作CPython。这个名字根据它是由可移植的ANSI C语言代码编写而成的这个事实而来的。这就是你从*http://www.python.org*获取的、从ActivePython分发包中得到的以及从绝大多数Linux和Mac OS X机器上自动安装的Python。如果你在机器上发现有个预安装版本的Python，那么很有可能就是CPython，除非公司将Python用在相当特别的场合。

除非希望使用Python脚本化Java或.NET，你或许想要使用的就是标准的CPython系统。因为CPython是这门语言的参照实现方式，所以和其他的替代系统相比来说，它运行速度最快、最完整而且也最健全。图2-2反映了CPython的运行体系结构。

Jython

Jython系统（最初称为JPython）是一种Python语言的替代实现方式，其目的是为了与Java编程语言集成。Jython包含了Java类，这些类编译Python源代码、形成Java字节码，并将得到的字节码映射到Java虚拟机（JVM）上。程序员仍然可以像平常一样，在文本文件中编写Python语句；Jython系统的本质是将图2-2中的最右边两个方框中的内容替换为基于Java的等效实现。

Jython的目标是让Python代码能够脚本化Java应用程序，就好像CPython允许Python脚本化C和C++组件一样。它实现了与Java的无缝集成。因为Python代码被翻译成Java字节码，在运行时看起来就像一个真正的Java程序一样。Jython脚本可以应用于开发Web applet和servlet，建立基于Java的GUI。此外，Jython具有集成支持的功能，允许导入Python代码或使用Java的类（这些类就像是用Python编写的一样）。因为Jython要比CPython慢而且也不够健壮，它往往被看作是一个主要面向寻找Java代码前端脚本语言的Java开发者的一个有趣的工具。

IronPython

Python的第三种（截止到目前写本书时，从某种程度上来讲，Python的第三种实现方式仍然是新的）实现方式IronPython，设计它的目的是让Python程序可以与Windows平台上的.NET框架以及与之对应的Linux的上开源的Mono编写成的应用相集成。本着像微软早期的COM模型一样的精神，将.NET和C#程序语言的运行系统设计成与语言无关性的对象通信层。IronPython允许Python程序既可以用作客户端也可以用作服务器端的组件，还可以与其他.NET的语言进行通信。

在实现上，IronPython很像Jython（实际上两者都是由同一个创始人开发的）：它替换

了图2-2中最后的两个方框，将其换成.NET环境的等效执行方式。并且，就像Jython一样，IronPython有特定的目标：它主要为了满足在.NET组件中集成Python的开发者。因为它是由微软公司开发的，IronPython也许能够为了性能实现完成一些重要的优化工具。IronPython涉及到的应用范围就像本书所写的那样；如果想了解更多细节，请参考Python的线上资源，或者在网络上搜索相关内容（注2）。

执行优化工具

CPython、Jython和IronPython都是通过同样的方式实现Python语言的，即通过编译源代码到字节码，然后在适合的虚拟机上执行这些字节码。然而，其他的系统，包括Psyco即时编译器以及Shedskin C++转换器，则试着优化了基本执行模块。这些系统并不是现阶段学习Python所必备知识，但是简要地了解这些执行模块可以帮助你更轻松地掌握这些模块。

Psyco实时编译器

Psyco系统并不是Python的另一种实现方式，而是一个扩展字节码执行模块的组件，可以让程序运行得更快。如图2-2所示，Psyco是一个PVM的增强工具，这个工具收集并使用信息，在程序运行时，可以将部分程序的字节码转换成底层的真正的二进制机器代码，从而实现更快的执行速度。在开发的过程中，Psyco无需代码的修改或独立的编译步骤即可完成这一转换。

概括地讲，当程序运行时，Psyco收集了正在传递过程中的对象的类别信息，这些信息可以用来裁剪对象的类型，从而生成高效的机器代码。机器代码一旦生成后，就替代了对应的原始字节码，从而加快程序的整体执行速度。实际的效果就是，通过使用Psyco，使程序在整个运行过程中执行得更快。在理想的情况下，一些通过Psyco优化的Python代码的执行速度可以像编译好的C代码一样快。

因为字节码的转换与程序运行同时发生，所以Psyco往往被看作是一个即时编译器（JIT）。Pysco实际上与一些读者曾经在Java语言中了解的JIT编译器稍有不同。实际上，Psyco是一个专有的JIT编译器：它生成机器代码将数据类型精简至你程序实际上所使用的类型。例如，如果程序的一部分在不同的时候采用了不同的数据类型，Psyco可以生成不同版本的机器码用来支持每一个不同的类型组合。

注2：Jython和Python是完全独立的Python实现，可以为不同的运行构建编译Python源代码。这也使标准CPython程序能够获取Java以及.NET软件：例如，JPype以及.NET系统的Python，允许Python调用Java以及.NET的组件。

www.TopSage.com

Psyco已经证实能够大大提高Python代码的速度。根据其官方网站介绍，Psyco提供了“2倍至100倍的速度提升，典型值为4x，在没有改进的Python解释器和不修改的源代码基础上，仅仅依靠动态可加载的C扩展模块”。同等重要的是，最显著的提速是在以纯Python写成的算法代码上实现的。确切地讲，是那些为了优化往往需要迁徙到C的那部分代码。使用了Psyco后，这样的迁徙甚至没有必要了。

Psyco目前还不是标准Python的一部分，你也许需要单独获取并安装它。而且它仍是一个研究项目，所以需要在网上跟踪它的发展。事实上，目前写本书的时候，尽管Psyco本身仍可以获得并能够自动安装，但这个系统的大部分似乎最终将会被一个更新的项目“PyPy”（一个尝试用Python代码实现Python PVM的项目，能够像Psyco一样提供更好的优化）融合。

也许Psyco的最大缺点就是它实际上只能够为Intel x86构架的芯片生成机器代码，尽管包括了Windows、Linux以及最新的Mac。若想获得更多有关Psyco扩展的细节，以及JIT可能带来的效果，请参考 *http://www.python.org*。你也可以浏览Psyco的官方网站，目前的网址为 *http://psyco.sourceforge.net*。

Shedskin C++转换器

Shedskin是一个引擎系统，它采用了一种不同的Python程序执行方法：它尝试将Python代码变为C++代码，然后使用机器中的C++编译器将得到的C++代码编译为机器代码。正是如此，它以一种平台无关的方式来运行Python代码。在写本书的时候Shedskin仍是一个实验性质的项目，并且它给Python程序施加了一种隐晦的静态类型约束，而这在一般的Python代码中是不常见的，所以我们不再深入了解其中的一些细节了。不过初步结果显示它具有比标准Python代码以及使用Psyco扩展后的执行速度更快的潜质，并且它是一个前途光明的项目。请通过网络搜索以获得更多细节以及项目目前的发展状况。

冻结二进制文件

有时候人们需要一个“真正的”Python编译器，实际上他们真正需要的是得到一种能够让Python程序生成独立的可执行二进制代码的简单方法。这是一个比执行流程概念更接近于打包分发概念的东西，但是二者之间或多或少有些联系。通过从网络上获得的一些第三方工具，将Python程序转为可执行程序[在Python世界中被称作冻结二进制文件（Frozen Binary）]是有可能的。

冻结二进制文件能够将程序的字节码、PVM（解释器）以及任何程序所需要的Python支持文件捆绑在一起形成一个单独的文件包。过程会有一些不同，但是实际的结果将会是

一个单独的可执行二进制程序（例如，Windows系统中的*.exe*文件），这个程序可以很容易地向客户分发。如图2-2所示，这就好像将字节码和PVM混合在一起形成一个独立的组件——冻结二进制文件。

如今，主要有三种系统能够生成冻结二进制文件：*py2exe*（Windows下使用）、PyInstaller（和*py2exe*类似，它能够在Linux及UNIX上使用，并且能够生成自安装的二进制文件）以及*freeze*（最初始的版本）。你可以单独获得这些工具，它们也是免费的。它们处在持续的开发过程中，请参考*http://www.python.org*以及Vaults of Parnassus网站（*http://www.vex.net/parnassus/*）以便获得有关这些工具的更多信息。这里我们给出一些信息，方便你了解这些系统的应用范围，例如*py2exe*可以封装使用了Tkinter、Pmw、wxPython和PyGTK GUI库的独立程序；应用*pygame*进行游戏编程的程序；win32com客户端的程序等。

冻结二进制文件与真实的编译输出结果有所不同：它们通过虚拟机运行字节码。因此，如果离开了必要的初始改进，冻结二进制文件和最初的源代码程序运行速度完全相同。冻结二进制文件并不小（包括PVM），但是以目前的标准来衡量，它们的文件也不是特别的大。因为在冻结二进制文件中嵌入了Python，接收端并不需要安装Python来运行这些冻结二进制文件。此外，由于代码嵌入在冻结二进制代码之中，对于接收者来说，代码都是隐藏起来的。

对商业软件的开发者来说，单文件封装的构架特别有吸引力。例如，一个Python编码的基于Tkinter工具包的用户界面可以封装成一个可执行文件，并且可以作为一个CD中或网络上的独立程序进行发售。终端用户无需安装（甚至没有必要知道）Python去运行这些发售的程序。

未来的可能性

最后，值得注意的是这里所简要描述的运行时执行模块事实上是当前Python实现的产品，并不是语言本身。例如，或许在本书的销售过程中会出现一种完全的、传统的将Python源代码变为机器代码的编译器。未来也许会有新的字节码格式和实现方式的变种将被采用。例如：

- *Parrot*项目的目标就是提供一种对于多种编程语言通用的字节码格式、虚拟机以及优化技术（请参看*http://www.python.org*）。
- *Stackless Python*系统是一种标准CPython实现的变种，它不在C语言的栈上保存状态。这使得Python更容易移植到小栈的架构上，开创了一种新颖的编程可能性，例如，协同程序。

www.TopSage.com

- 新项目PyPy尝试在PVM上重新实现Python，以便使新的实现技术成为可能。

尽管未来实现的原理有可能从某种程度上改变Python运行的结构，但就未来的一个时期内来看，字节码编译仍然将会是一种标准。字节码的可移植性和运行的灵活性对于很多Python系统来说是很重要的特性。此外，为了实现静态编译，而增加类型约束声明将会破坏这种灵活、明了、简单以及所有代表了Python编码精神的特性。由于Python本身的高度动态性，以后的任何实现方式都可能保留许多当前的PVM产品。

本章小结

本章介绍了Python的执行模块（Python如何运行程序）并探索了这个模块的一些变种（即时编译器以及类似的工具）。尽管编写Python脚本并没有必要了解Python的内部实现，通过本章介绍的主题获得的知识会帮助你从一开始编码时就真正理解程序是如何运行的。下一章，我们将开始实际运行编写的代码。那么，首先让我们开始常规的章节测试吧。

www.TopSage.com

本章习题

1. 什么是Python解释器?
2. 什么是源代码?
3. 什么是字节码?
4. 什么是PVM?
5. 请列出两个Python标准执行模块的变种的名字。
6. CPython、Jython以及IronPython有什么不同?

习题解答

1. Python解释器是运行Python程序的程序。
2. 源代码是为程序所写的语句：它包括了文本文件（通常以*.py*为后缀名）的文本。
3. 字节码是Python将程序编译后所得到的底层形式。Python自动将字节码保存到后缀名为*.pyc*的文件中。
4. PVM是Python虚拟机：Python的运行时引擎解释编译得到的代码。
5. Psyco、Shedskin以及forzen binaries是执行模块的所有变种。
6. CPython是Python语言的标准实现。Jython和IronPython分别是Python程序的Java和.NET的实现，它们都是Python的编译器的替代实现。

www.TopSage.com

第3章

如何运行程序

好了，是开始编写程序的时候了。现在你已经掌握了程序执行的知识，终于可以准备开始一些真正的Python编程了。假设已经在电脑上安装好了Python；如果没有的话，请参照前一章或附录A来获得一些安装和配置的提示。

我们已经介绍了多种执行Python程序的方法。这一章我们讨论的内容都将是当前常用的启动技术。在这个过程中，将会学习如何交互地输入程序代码、如何将其保存至一个文件从而以后可以在系统命令行中运行、图标点击、模块导入、IDLE和Eclipse这样IDE GUI等内容。

如果你只想知道如何快速地运行Python程序，建议你阅读自己你的平台相关的内容并直接开始第4章。但是不要跳过模块导入的内容，因为这是你理解Python程序架构的基础。同时建议你至少浏览一下IDLE和其他IDE的部分，从而了解什么样的工具更适合你，帮助你开发出更为精致的Python程序。

交互模式下编写代码

也许最简单的运行Python程序的办法就是在Python交互命令行中输入这些程序。有多种办法能够开始这样的命令行：在IDE中、系统终端中等。假设解释器已经作为一个可执行程序安装在你的系统中，开始交互解释对话的最具有平台无关特性的方法，往往就是在操作系统的提示环境下输入python，不需要任何参数。例如：

```
% python
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

在系统提示环境下输入“python”后即可开始一个交互的Python会话（“%”字符代表

www.TopSage.com

了系统提示符，这个字符是不需要自己输入的）。注意这里的系统提示环境是通用的，而实际应用中根据平台的不同，获得的提示环境也是不同的：

- 在Windows中，可以在DOS终端窗口中输入**python**（称为命令提示符，通常可以从“开始”按钮的命令菜单中的附件中找到）或者在开始→运行…的对话框中输入也可以。
- 在UNIX、Linux以及Mac OS X中，可以在shell窗口或终端窗口中（例如，在xterm或终端中运行的*ksh*或*csh*这样的shell）输入python即可。
- 其他的系统可以采用类似的方法或平台特定的工具。例如，在PalmPilots上，可以点击Python home的图标启动一个交互的会话。

如果你没有设置系统中shell的PATH环境变量，使其包含了Python的安装目录，你也许需要将“python”改为机器上Python可执行文件的完整路径。在Windows上，可以尝试输入**C:\Python25\python**(对于2.5版本)；在UNIX或Linux上，可以输入**/usr/local/bin/python**（或**/usr/bin/python**）。或者，你可以将目录变换到Python的安装目录下（例如，可以在WIndows中尝试**cd c:\python25**）之后运行“python”。

Python交互对话刚开始时将会打印两行信息文本（为了节省章节内容在这里省略了这个例子），等待输入新的Python语句或表达式的提示符>>>。在交互模式下工作，输入代码的结果将会在>>>这一行之后显示。这是两行Python打印语句的输出结果：

```
% python
>>> print 'Hello world!'
Hello world!
>>> print 2 ** 8
256
```

现在还不需要为这里显示的print语句的细节担心（我们将会在下一章开始深入了解语法）。简而言之，这两行语句打印了一个Python的字符串和一个整数，和>>>输入的那行下边的输出行显示的那样。

像这样在交互模式下工作，想输入多少Python命令就输入多少，每一个命令在输入回车后都回立即运行。此外，由于交互式对话自动打印输入表达式的结果，在这个提示模式下，往往不需要每次都刻意地输入“print”：

```
>>> lumberjack = 'okay'
>>> lumberjack
'okay'
>>> 2 ** 8
256
```


www.TopSage.com

```
>>>                              # 使用Ctrl-D或者Ctrl-Z退出
%
```

此处，最后两行的输入为表达式（lumberjack和2**8），它们的结果是自动显示的。像这里一样退出交互对话并回到系统shell提示模式，在UNIX机器上输入Ctrl-D退出；在MS-DOS和Windows系统中输入Ctrl-Z退出。在我随后讨论到的IDLE GUI中，也可以输入Ctrl-D退出或简单地关闭窗口来退出。

现在，我们对这次会话中的代码并不是特别地了解：仅仅是输入一些Python的打印语句和变量赋值的语句，以及一些表达式，这些我们都会在稍后进行深入地学习。这里最重要的事情就是注意到解释器在每行代码输入完成后，也就是按下回车后立即执行。

例如，当在>>>提示符下输入第一条打印语句时，输出（一个Python字符串）立即回显出来。没有必要创建一个源代码文件，也没有必要在运行代码前先通过编译器和连接器，而这些是以往在使用类似C或C++语言时所必须的。

除了在shell窗口中输入**python**，也可以通过使用IDLE主窗口，或者在Windows上开始按钮上的Python菜单中通过选择“Python(command-line)”这个选项（如图2-1所示）开始简单的交互会话。这两种方法都会产生一个具有相同功能的>>>的提示符：代码在输入后立即运行。

在交互提示模式下测试代码

由于代码是立即执行的，交互提示模式变成了实验这个语言的绝佳的地方。这会在本书中示范较小的例子时常常用到。实际上，这也是需要牢记的第一条原则：当你对一段Python代码的运行有任何疑问的时候，马上打开交互命令行并实验代码，看看会发生什么。如果不会破坏什么东西的几率很大（在你变得危险之前，还是需要了解更多系统接口的知识）。

尽管你不会在交互会话中做大量的代码工作(因为输入的代码并不会被保存下来)，交互解释器仍是一个测试已经保存在文件中的代码的好地方。你可以通过交互的方式改善模块文件，并通过在交互提示模式中输入命令运用这些模块定义的工具来进行一些测试。更为常见的是，交互提示模式是一个测试程序组件的地方，不需要考虑其源代码（你可以通过输入命令来连接）是用C函数，还是Jython使用的Java类等。一部分是因为这种交互的本身特性，Python支持了一种实验性和探索性的编程风格，当开始使用的时候，你就会发现这种风格是很方便的。

www.TopSage.com

使用交互提示模式

尽管交互提示模式简单易用，这里还有那些初学者需要牢记的一些技巧。

- **只能够输入Python命令。**首先，记住只能在Python交互模式下输入Python代码，而不要输入系统的命令。这里有一些方法可以在Python代码中使用系统命令（例如，使用os.system），但是并不像简单的输入命令那么的直接。
- **在文件中打印语句是必须的。**在交互解释器中自动打印表达式的结果，不需要在交互模式下输入完整的打印语句。这是一个不错的特性，但是换成在文件中编写代码时，用户就会产生一些困惑：在文件中编写代码，必须使用print语句来进行输出，因为表达式的结果不会自动反应。记住，在文件中需要写print，在交互模式下则不需要。
- **在交互提示模式下不需要缩进（目前还不需要）。**当输入Python程序时，无论是在交互模式下还是在一个文本文件中，请确定所有没有嵌套的语句都在第一列（也就是说要在最左边）。如果不是这样，Python也许会打印“SyntaxError”的信息。在第10章以前，你所编写的所有的语句都不需要嵌套，所以这条法则目前都还适用。在介绍Python的初级课程时，这看起来也许会令人困惑。每行开头的空格也会产生错误的消息。
- **留意提示符的变换和复合语句。**我们在第10章之前不会见到复合（多行）语句，但是，为了预先有个准备，当在交换模式下输入二行或多行的复合语句时，提示符会发生变化。在简单的shell窗口界面中，交互提示符会在第二行及后边的行由>>>变成…；在IDLE界面中，第一行之后的行会被自动缩进。无论哪种情况，需要插入一个空行（在这行的起始位置点击回车键），告诉交互模式的Python，你已经完成了多行语句的输入；与之相比，文件中的空行将会被省略。

  在第10章中将看到这为什么如此重要。就目前而言，如果在代码中输入，偶然碰到…这个提示符或空行，这可能意味着让交互模式的Python误以为输入多行语句。试着点击回车键或Ctrl-C组合键来返回主提示模式。也可以改变>>>和…（它们在内置模块sys中定义），但是在本书的例子中，假定并没有改变过这两个提示符。

系统命令行和文件

尽管交互命令行对于实验和测试来说都很好，但是它也有一个很大的缺点：Python一旦执行了输入的程序之后，它们就消失了。在交互模式下输入的代码是不会保存在一个文件中的，所以为了能够重新运行，不得不从头开始输入。复制－粘帖和命令重调在这里也许有点用，但是帮助也不是很大，特别是当输入了相对较大的程序时。为了从交互对

www.TopSage.com

话模式中复制－粘帖代码，不得不重新编辑清理出Python提示符、程序输出以及其他的一些东西。

为了能够永久的保存程序，需要在文件中写入代码，这样的文件往往被看作模块。模块是一个包含了Python语句的简单文本文件。一旦编写完成，可以让Python解释器多次运行这样的文件中的语句，并且可以以多种方式去运行：通过系统命令行、通过点击图标、通过在IDLE用户界面中选择等方式。无论它是如何运行的，每一次当你运行模块文件时，Python都会从头至尾地执行模块文件中的每一条代码。

这一部分的术语可能会有某些变化。例如，模块文件常常作为Python写成的程序。也就是说，一个程序是由一系列预编写好的语句构成，保存在文件中，从而可以反复执行。可以直接运行的模块文件往往也称作脚本（一个顶层程序文件的非正式说法）。有些人将“模块”这个说法应用在被另一个文件所导入的文件（之后会为大家解释“顶层”和“导入”的含义）。

不论你怎样称呼它们，下面的几部分内容将会探索如何运行输入至模块文件的代码的方法。这一节，将会介绍如何以最基本的方法运行文件：通过在系统提示模式下的python命令行，列出它们的名字。在举例之前，打开文本编辑器（例如，*vi*、Notepad或IDLE编辑器），并在命名为*spam.py*的文件中输入两行Python语句：

```
print 2 ** 8                              # 求乘方
print 'the bright side ' + 'of life'      # “+”意思是连接
```

本文件包含了两条Python打印语句，并且在每一行的右边有一些Python注释（解释器会忽略#后的字符这只作为供读者阅读的注释，不是程序语句内容的一部分）。再次提示，目前请忽略掉这些文件代码的语法。此时需要注意的是将这些代码输入一个文件，而不是在交互提示模式下输入的。不知不觉中，你已经编写了一个功能完整的Python脚本了。

保存这个文本文件，在python命令后第一个参数的位置上列出文件的完整文件名，从而让Python运行这个文件，在系统shell提示模式下输入如下内容：

```
% python spam.py
256
the bright side of life
```

输入这样的系统shell命令，无论系统提供的命令行实体是Windows命令提示符窗口、xterm窗口还是其他等效的窗口。记住如果没有配置PATH这个参数的话，需要将“python”替换成完整的路径。在命令输入后显示这个脚本的输出：这是文本文件中的两行打印命令的结果。

www.TopSage.com

值得注意的是将模块文件命名为*spam.py*。作为顶层文件，它同样可以简单地命名为*spam*，但是希望导入至一个客户端的的代码文件必须是以*.py*为后缀的。我们将会在本章进行导入学习。

你也许想要导入一个文件，对于编写的绝大多数的Python文件使用后缀名.py是一个好主意。而且，有些文本编辑器可以通过后缀名*.py*识别出Python文件；如果不使用这样的后缀名，你也许不会得到像语法强调和自动缩进这样的特性。

因为这种方法是使用shell命令行开始Python程序，所以可以采用以往的shell的语法。例如，你可以重定向Python的输出至一个文件，以便以后使用，或者采用其他的一些特定shell语句对输出进行检查。

```
% python spam.py > saveit.txt
```

在这种情况下，之前运行输出的那两行保存在*saveit.txt*文件中。这通常称为流重定向；这可以应用在输入和输出文本中，可以在Windows或UNIX类系统中应用。这与Python稍有些关系（Python对其有简单的支持），所以我们这里不再对重定向的细节进行深入讨论。

如果你工作的系统为Windows平台，这个例子同样适用，但是往往系统的提示符发生了变化：

```
C:\Python25> python spam.py
256
the bright side of life
```

就像往常一样，如果没有设置PATH这个环境变量，请输入Python的完整路径，要么就切换到Python的安装目录：

```
D:\temp> C:\python25\python spam.py
256
the bright side of life
```

在较新的Windows版本上，可以直接输入脚本的文件名，不需要了解当前目录在哪里。因为较新的Windows系统使用Windows注册表会查找使用哪个程序运行这个文件，不需要在命令行中明确地列举其名字。之前的命令，例如，在最近的Windows上可以进行简化：

```
D:\temp> spam.py
```

最后，请记住如果脚本文件不在当前目录下，需要给出脚本文件的完整路径。例如，在

www.TopSage.com

下面的系统命令行中，当前目录为*D:\other*，假设Python在系统的路径中，但是要运行的文件存放在其他地方：

```
D:\other> python c:\code\myscript.py
```

使用命令行和文件

从系统命令行开始运行程序文件是相当直接明了的选择，特别是在通过你之前的日常工作已熟悉了命令行的使用时。对于初学者来说，我们提示大家注意这些新手陷阱：

- **注意Windows上的默认扩展名。**如果使用Windows系统的记事本编写程序文件，当保存文件时要注意选择所有文件类型，并指定文件后缀为*.py*。否则记事本会自动将文件保存成扩展名为*.txt*的文件（例如，保存成*spam.py.txt*），导致有些启动的方法运行程序困难。

  更糟糕的是，Windows默认隐藏文件扩展名，所以除非改变查看选项，否则你可能没有办法注意到你编写的文件是文本文件而不是Python文件。文件的图标可以给我们一些提示：如果图标上没有一条小蛇的话，你可能就有麻烦了。发生这样问题的其他一些症状还包括IDLE中的代码没有着色，以及点击时没有运行而变成了打开编辑文件。

  Microsoft Word默认文件扩展名为*.doc*；更糟糕的是，它增加了Python语法中不合乎语法的一些格式字符。作为一条简要的法则，当在Windows下保存文件时，永远要选择所有文件，或者使用对程序员更加友好的文本编辑器，例如，IDLE。IDLE不会自动添加*.py*后缀：这个特性程序员也许会喜欢，但是一般用户不会。

- **在系统提示模式下使用文件扩展名，但是在导入时别使用文件扩展名。**在系统命令行中别忘记输入文件的完整文件名。也就是说，使用**`python spam.py`**而不是**`python sapm`**。我们将会在本章后边提到Python的导入语句，忽略*.py*文件后缀名以及目录路径（例如，`import spam`）。这看起来简单，却是一个常见的错误。

  在系统提示模式下，你就是在一个系统的shell中，而不是Python中，所以Python的模块文件的搜索规则不再适用了。正是如此，必须包括*.py*后缀，并且可以在运行文件前包括其完整路径（例如，**`C:\python\25>python d:\tests\spam.py`**）。然而，在Python代码中，你可以只写**`import spam`**，并依靠Python模块搜索的路径定位文件，这将稍后进行介绍。

- **在文件中使用打印语句。**是的，我们已经这样说过了，但是这是一个常见错误，值得我们在这里重复说明。不像交互模式的编程，往往需要使用打印语句来看程序文件的输出。

UNIX可执行脚本(#!)

如果在Python、Linux及其他的UNIX类系统上使用Python，可以将Python代码编程为可执行程序，就像使用*csh*或*ksh*等的shell语言编写的程序一样。这样的脚本往往称为可执行脚本。简而言之，UNIX风格的可执行脚本包含了Python语句的一般文本文件，但是有两个特殊的属性。

- **它们的第一行是特定的。**脚本的第一行往往以字符#!开始（常常称作“hash bang”），其后紧跟着机器Python解释器的路径。
- **它们往往都拥有可执行的权限。**脚本文件往往通过告诉操作系统它们可以作为顶层程序执行，而拥有可执行的权限。在UNIX系统上，往往可以使用`chmod +x file.py`来实现这样的目的。

让我们看一个UNIX类系统的例子。使用文本编辑器创建一个名为*brian*的文件：

```
#!/usr/local/bin/python
print 'The Bright Side of Life...'          # 这的另一个注释
```

文件顶端的特定的一行告诉系统Python解释器保存在哪里。从技术上来看，第一行是Python注释。就像前之所介绍的一样，Python程序的注释都是以#开始并直到本行的结束为止；它们是为代码读者提供额外信息的地方。但是当第一行和这个文件一样的话，它就有特定的意义，因为操作系统使用它找到解释器来运行文件其他部分的程序代码。

并且，注意这个文件命名为*brian*，而没有像之前模块文件一样使用*.py*后缀。给文件名增加*.py*也没有关系（也许还会提醒你这是一个Python程序文件），但是因为这个文件中的代码并不打算被其他模块所导入，这个文件的文件名是没有关系的。如果通过使用`chmod +x brian`这条shell命令赋予了这个文件可执行的权限，你就能够在操作系统的shell中运行它，就好像这是一个二进制文件一样：

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

给Windows用户的另一个提示：这里介绍的方法是UNIX的一个技巧，也许它在你的平

www.TopSage.com

台上并不可行。但是别担心，可以使用我们刚才介绍的基本的命令行技术。在命令行中python后列出明确的文件名（注1）：

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

在这种情况下，不需要文件顶部的特定的#!注释（如果它还存在的话，Python会忽略它），并且这个文件不需要赋予可执行的权限。事实上，如果你可能想要在UNIX及微软Windows系统中都运行文件，如果经常采用基本的命令行的方法而不是UNIX风格的脚本去运行程序，你的生活或许会更简单一些。

UNIX env查找技巧

在一些UNIX系统上，也许可以避免硬编码Python解释器的路径，而可以在文件特定的第一行注释中像这样写：

```
#!/usr/bin/env python
...script goes here...
```

当这样编写代码的时候，*env*程序可以通过系统的搜索路径的设置（例如，在绝大多数的UNIX shell中，通过搜索PATH环境变量中的罗列出的所有目录）定位Python解释器。这种方法可以使代码更具可移植性，因为没有必要在所有的代码中的第一行都硬编码Python的安装路径。

假设在任何地方都能够使用*env*，无论Python安装在了系统的什么地方，你的脚本都可以照样运行：跨平台工作时所需要做的仅仅是改变PATH环境变量，而不是脚本中的第一行。当然，这是*env*在任何系统中都是相同的路径的前提下（有些机器，还有可能在*/sbin*、*/bin*或其他地方）；如果不是的话，这种可移植性也就无从谈起了。

点击文件图标

在Windows下，注册表使通过点击图标打开文件变得很容易。当Python程序文件点击打

注1：研究命令行时，我们讨论过，当前的Windows版本可在系统命令行上只输入*.py*文件的名称，因为Windows会使用注册表机制来确认该文件应该通过Python启动（例如，输入**brian.py**相当输入**python brian.py**）。这个命令行模式的实质类似于UNIX #!。注意在Windows上，有些程序会实际去解译并使用顶端的#!行，像UNIX那样，但是，Windows的DOS系统shell会完全忽略它。

www.TopSage.com

开时Python自动注册为所运行的那个程序。正是如此，你可以通过使用鼠标简单的点击（或双击）程序的图标来运行程序。

在非Windows系统中，也能够使用相似的技巧，但是图标、文件管理器、浏览的原理以及很多方面都有少许不同。例如，在一些UNIX系统上，也许需要在文件管理器的GUI中注册*.py*的扩展名，从而可以使用前一节介绍的#!技巧使脚本成为可执行的程序，或者使用应用程序关联文件的MIME类型或通过编辑文件、安装程序等命令，或者使用其他的工具。如果一开始点击后不能正常的工作，请参考文件管理器的文档以获得更多细节。

在Windows中点击图标

为了讲清楚，假设使用文本编辑器创建了如下的程序文件并将其保存为*script4.py*:

```
# 注释
import sys
print sys.platform
print 2 ** 100
```

这里没有太多新的东西：仅仅是一行导入和两行打印语句（sys.paltform是标示计算机种类的字符串；它在名为sys的模块中，所以必须导入来加载这个字符串）。可以在系统命令行中运行这个文件：

```
D:\LP3E\Examples> c:\python25\python script4.py
win32
1267650600228229401496703205376
```

然而，点击图标可以让你不需要任何输入即可运行文件。如果找到了这个文件的图标（例如，通过选择“我的电脑”，找到D驱动器的工作路径），将会得到如图3-1所示的文件管理器的截屏图（这里使用的是Windows XP）。在Python 2.5中，源文件在Windows中外观为白色背景的图标，字节码有黑色底色。一般你就会想要点击（或者说运行）源代码文件，为了观察最新修改后的结果。为了运行这里的文件，简单的点击*script4.py*的图标。

raw_input的技巧

不幸的是，在Windows中，点击文件图标的结果也许不是特别令人满意。事实上，就像刚才一样，这个例子的脚本在点击后产生了一个令人困惑的“一闪而过”的结果，而不是Python程序的入门者所期盼的结果反馈。这不是bug，但是需要做某种操作才能够让Windows处理打印的结果。

www.TopSage.com

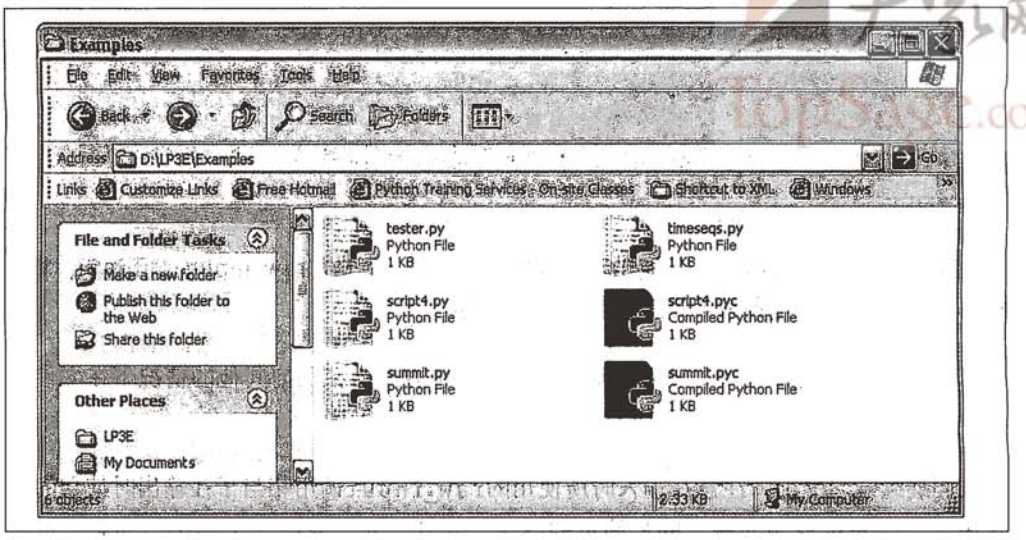


图3-1：在Windows中，Python程序在文件管理器窗口中显示为一个图标，并通过鼠标双击能够自动运行（尽管你采用这种办法也许不会看到打印的输出或错误的提示）

在默认情况下，Python会生成弹出一个黑色DOS终端窗口作为文件的输入或输出。如果脚本打印后退出了，也就是说，它仅是打印后退出终端窗口显示，然后文本在这里打印，但是在程序退出时，终端窗口关闭并消失。除非你反应非常快，或者是机器运行非常慢，否则看不到任何输出。尽管这是很正常的行为，但是这也许并不是你所想象的那样。

幸运的是，这样的问题很好解决。如果需要通过图标点击运行脚本时，脚本输出后暂停，可以简单地在脚本的最后添加内置raw_input函数的调用语句。例如：

```
# 注释
import sys
print sys.platform
print 2 ** 100
raw_input()             # ADDED
```

一般来说，raw_input读取标准输入的下一行，如果还没有得到的话一直等待输入。在这种情形下执行的实际效果就是让脚本暂停，因此能够显示如图3-2所示的输出窗口，直到按下回车键为止。

现在介绍的这个技巧，往往只在Windows中才是必要的，并且只是当脚本打印文本后退出或只是当通过点击文件图标运行脚本才是必要的。当且仅当以上这三个条件全部都生

www.TopSage.com

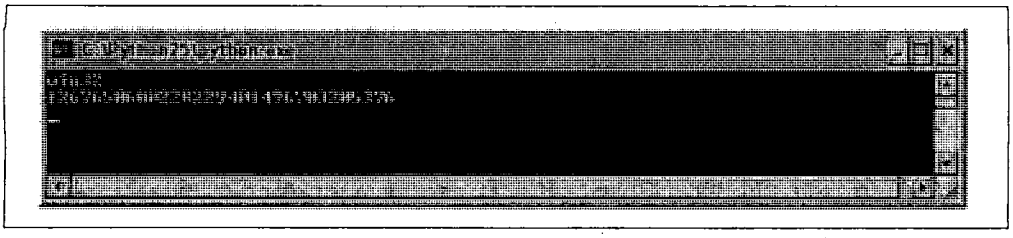


图3-2：当在Windows中点击程序图标时，如果在脚本的最后增加raw_input()调用语句，将会得到其输出的窗口。但是只需要在这种情况下这样做

效时，才应当在顶层文件的最后增加这个调用。没有理由在任何其他的情况下在文件中增加这个调用（注2）。

在我们继续学习之前，注意在输入时所使用的raw_input调用相当于在输出时使用的打印语句。这是读取用户输入的最简单的办法，并且实际上它比这个例子中的应用更全面。例如，raw_input。

- 可选的接受字符串，这些字符串将作为提示打印出来［例如，raw_input('Press Enter to exit')］。
- 以字符串的形式为脚本返回读入的文本［例如，nextinput = raw_input()］。
- 在系统shell的层面上支持输入流的重定向（例如，python spam.py < input.txt），就像输出时的打印语句一样。

在后面的章节中，我们将会以更复杂的方法使用raw_input。例如，第10章我们将会在交互循环中使用它。

图标点击的其他限制

即使使用了raw_input的技巧，点击文件图标仍有一定的风险。你可能看不到Python的错误信息。如果脚本出现了错误，错误信息的文字将会写在弹出的终端窗口上：这个窗口马上就会消失。更糟糕的是，这次即使是在文件中添加了对raw_input的调用也无济于事，因为早在调用raw_input之前脚本就已经终止。换句话说，你不会知道到底是哪里出了错误。

注2：在Windows中，还有一种完全阻止弹出DOS终端窗口的方法。以.pyw为扩展名的文件只显示由脚本构建的窗口，而不是默认的DOS终端窗口。.pyw文件是拥有这种特别的窗口操作行为的.py文件。它们常常应用于Python编码的用户界面，以及保存打印完成的输出和错误至文件中的情况。

www.TopSage.com

正是由于这些限制，最好将点击图标看作程序调试之后运行的一种方法。特别是初学的时候，请使用其他技术。例如，通过系统命令行或IDLE（本章将会介绍），以便能够看到生成的错误信息，并在不使用编码技巧的情况下，观察正常的输出结果。当我们在本书的后边讨论异常的时候，你将会认识到拦截并修复错误，以便错误不会终止程序。请留意本书后边对`try`语句的讨论，从中找到另一种发生了错误而不会关闭终端窗口的方法。

模块导入和重载

到现在为止，本书已经讲到了“导入模块”，而实际上没有介绍这个名词的意义。我们将会在第5部分深入学习模块和较大的程序架构，但是由于导入同时也是一种启动程序的方法，为了能够入门，这一节将会介绍一些模块的基础知识。

用简单的术语来讲，每一个以扩展名*.py*结尾的Python源代码文件都是一个模块。其他的文件可以通过导入一个模块读取这个模块的内容。导入从本质上来讲，就是载入另一个文件，并能够读取那个文件的内容。一个模块的内容通过这样的属性（这个术语我们将会在下一节定义）能够被外部世界使用。

这种基于模块的方式使模块变成了Python程序架构的一个核心概念。更大的程序往往以多个模块文件的形式出现，并且导入了其他模块文件的工具。其中的一个模块文件被设计成主文件，或叫做顶层文件（就是那个启动后能够运行整个程序的文件）。

我们将会对这样的架构问题有更深入的探索。本章最关心的是被载入的文件通过导入操作最终可运行代码。正是如此，导入文件是另一种运行文件的方法。

例如，如果开始一个交互对话（在IDLE中，从命令行或者其他），你可以运行之前创建的文件*script4.py*，通过简单的`import`来实现。

```
D:\LP3E\Examples> c:\python25\python
>>> import script4
win32
1267650600228229401496703205376
```

这可以运行，但是在默认情况下，只是在每次会话的第一次运行（真的，不信你可以试一下）。在第一次导入之后，其他的导入都不会再工作，甚至在另一个窗口中改变并保存了模块的源代码文件也不行。

```
>>> import script4
>>> import script4
```

这是有意设计的结果。导入是一个开销很大的操作以致于每个程序运行不能够重复多于一次。当你学习第18章时会了解，导入必须找到文件，将其编译成字节码，并且运行代码。

TopSage.com

但是如果真的想要Python在同一次会话中再次运行文件（不停止和重新启动会话），需要调用内置的reload（重载）函数。

```
>>> reload(script4)
win32
65536
<module 'script4' from 'script4.py'>
>>>
```

重载函数载入并运行了文件当前版本的代码，如果已经在另一个窗口中修改了它。这允许你在当前交互会话的过程中编辑并改进代码。例如，这次会话中，在第一个import和reload调用这段时间里，在*script4.py*中的第二个打印语句在另一个窗口中改成了2 ** 16。

reload函数希望获得的参数是一个已经加载了的模块对象的名称，所以如果在重载之前，请确保已经成功地导入了这个模块。值得注意的是，reload函数在模块对象的名称前还需要括号，import则不需要。reload是一个被调用的函数，而import是一个语句。这也就是为什么你必须传递模块名称给reload函数作为括号中的参数，并且这也是为何在重载时得到了额外的一行输出的原因。最后一行输出是reload调用后的返回值的打印显示，reload函数的返回值是一个Python模块对象。

函数将会在第15章介绍。

模块的显要特性：属性

导入和重载提供了一种自然的程序启动的选择，因为导入操作将会在最后一步执行文件。从更宏观的角度来看，模块扮演了一个工具库的角色，这将在第5部分学到。从一般意义上来说，模块往往就是变量名的封装，被认作是命名空间。在一个包中的变量名就是所谓的属性：也就是说，属性就是绑定在特定的对象上的变量名。

在典型的应用中，导入者得到了模块文件中在顶层所定义的所有变量名。这些变量名通常被赋值给通过模块函数、类、变量以及其他被导出的工具。这些往往都会在其他文件或程序中使用。表面上来看，一个模块文件的变量名可以通过两个Python语句读取——import和from，以及reload调用。

www.TopSage.com

为了讲清楚，请使用文本编辑器创建一个名为*myfile.py*的单行的Python模块文件，其内容如下所示：

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

这也许是世界上最简单的Python模块文件之一了（它只包含了一行赋值语句），但是它已经足够讲明白基本的要点。当文件导入时，它的代码运行并生成了模块的属性。这个赋值语句创建了一个名为title的模块的属性。

可以通过两种不同的办法从其他组件获得这个模块的title属性。第一种，你可以通过使用一个import语句将模块作为一个整体进行载入，并在启动Python使用模块名后跟一个属性名来获取它：

```
% python                          # 启动Python
>>> import myfile                 # 运行文件；载入模块作为一个整体
>>> print myfile.title            # 使用属性名字：来限定
The Meaning of Life
```

一般来说，这里的点号表达式代表了object.attribute的语法，可以从任何的object中取出其任意的属性，并且这是Python代码中的一个常用操作。在这里，我们已经使用了它去获取在模块*myfile*中的一个字符串变量title，即myfile.title。

作为替代方案，可以通过这样的语句从模块文件中获得（实际上是拷贝）变量名：

```
% python                          # 启动Python
>>> from myfile import title      # 运行文件；拷贝它的名字
>>> print title                   # 直接使用名字，不需限定
The Meaning of Life
```

就像今后看到的更多细节一样，from和import很相似，只不过增加了对载入组件的变量名的额外的赋值。从技术上讲，这是从模块的属性进行了拷贝，以便能够成为接收者的一个简单的变量。因此，能够简单的以title（一个变量）导入字符串而不是myfile.title（一个属性引用，注3）。

无论使用的是import还是from去执行导入操作，模块文件myfile.py的语句都会执行，并且导入的组件（对应这里是交互提示模式）在顶层文件中得到了变量名的读取权。也许在这个简单的例子中只有一个变量名（变量title被赋值给一个字符串），但是如果开

注3：注意，import和from列出模块名时，都是使用myfile，没有.py后缀。到了第5部分，你就会学到，当Python寻找实际文件时，知道在搜索程序中加上后缀名。然而，系统命令行中，一定要记得加上后缀名，但是import语句中则不用。

始在模块中定义对象，例如，函数和类时，这个概念将会很有用。这样一些对象就变成了可重用的组件，可以通过变量名被一个或多个客户端模块读取。

在实际应用中，模块文件往往定义了一个以上的可被外部文件使用的变量名。下面这个例子中定义了三个变量名：

```
a = 'dead'                    # 定义三个属性
b = 'parrot'                  # 导出到其他文件
c = 'sketch'
print a, b, c                 # 在这个文件中也要使用
```

文件*treenames.py*，给三个变量赋值，并对外部世界生成了三个属性。这个文件并且在一个print语句中使用它自有的三个变量，就像在将其作为顶层文件运行时看到的结果一样：

```
% python threenames.py
dead parrot sketch
```

所有的这个文件的代码运行起来就和第一次从其他地方导入（无论是通过import或者from）后一样。这个文件的客户端通过import得到了具有属性的模块，而客户端使用from时，则会获得文件变量名的拷贝。

```
% python
>>> import threenames              # 控制整个模块
dead parrot sketch
>>>
>>> threenames.b, threenames.c
('parrot', 'sketch')
>>>
>>> from threenames import a, b, c     # 拷贝多个名字
>>> b, c
('parrot', 'sketch')
```

这里的结果打印在括号中，（在本书的下一部分介绍的对象的一个种类）。

一旦你开始就像这里一样在模块文件编写多个变量名，内置的dir函数开始发挥作用了。你可以使用它来获得模块内部的可用的变量名的列表。

```
>>> dir(threenames)
['__builtins__', '__doc__', '__file__', '__name__', 'a', 'b', 'c']
```

当dir函数就像这个例子一样，通过把导入模块的名称传至括号里，进行调用后，它将返回这个模块内部的所有属性。其中返回的一些变量名是“免费”获得的：一些以双下划线开头并结尾的变量名，这些通常都是由Python预定义的内置变量名，对于解释器来说有特定的意义。那些通过代码赋值而定义的变量（a、b和c）在dir结果的最后显示。

www.TopSage.com

模块和命名空间

模块导入是一种运行代码文件的方法，但是就像稍后我们即将在本书中讨论的那样，模块同样是Python程序最大的程序结构。一般来说，Python程序往往由多个模块文件构成，通过import语句连接在一起。每个模块文件是一个独立完备的变量包装，即一个命名空间。一个模块文件不能看到其他文件定义的变量名，除非它明确地导入了那个文件，所以模块文件在代码文件中起到了最小化命名冲突的作用。因为每个文件都是一个独立完备的命名空间，即使在它们拼写相同的情况下，一个文件中的变量名是不会与另一个文件中的变量冲突的。

实际上，就像你将看到的那样，正是由于模块将变量封装为不同部分，Python具有了能够避免命名冲突的优点。我们将会在本书后面章节讨论模块和其他的命名空间结构（包括类和函数的作用域）。就目前而言，模块是一个不需要重复输入而可以反复运行代码的方法。

import和reload的使用注意事项

由于某种原因，一旦人们知道通过import和reload运行文件，有些人就会倾向于仅使用这个方法，而忽略了能够运行当前版本的代码的其他选择（例如，图标点击、IDLE菜单选项以及系统命令行）。这会让人变得困惑：需要记住是何时导入的，才能知道能不能够reload，需要记住当调用reload时需要使用括号，并且要记住让,代码的最新版本运行时首先要使用reload。

由于这些复杂的地方（并且我们将会在后边碰到其他的麻烦），从现在开始就要避免使用import和reload启动程序的冲动，这是一个好主意。例如，IDLE Run→Run模块的菜单选项，提供了一个简单并更少错误的运行文件的方法。另一方面，import和reload逐渐证明是一个在Python类中的一个流行的测试技术。你也许更喜欢这种实现，但是你发现自己走入一个死胡同的时候，停下来。

模块的故事不只是我们在这里所讲到的。例如，execfile('module.py')内置函数是另一个通过交互提示模式而不需要import或之后reload的运行文件的方法。它有类似的效果，但是从技术上来讲并没有导入模块。在默认情况下，每次调用execfile，都会运行一个新文件，就好像已经把它粘帖在了execfile调用的位置。正是由于这一点，execfile更像之前所提到过的from语句，它具有悄悄地覆盖正在使用中的变量的能力。基本的import语句，从另一个方面来说，每个进程只运行文件一次，并使该文件作为一个独立的命名空间，因此它不会改变作用域中的变量。

www.TopSage.com

此外，如果用不常见的方法使用模块，可能遇到麻烦。例如，如果想要导入一个模块文件，而该文件保存在其他的目录下而不是现在的工作目录，你必须跳到第18章并学习搜索路径的模块。从现在来说，如果必须导入，为了避免复杂性，请将所有的文件放在同一目录下，同时将这个目录作为你的工作目录。

注意： 如果你不想一直等到第18章才学习的话，那么简单地说，Python将会从列在`sys.path`（一个目录名的字符串的Python列表，在*sys*模块中，并通过PYTHONPATH这个环境变量进行初始化，并增加一系列的标准目录）中的所有目录中搜索被导入的模块。如果你想要导入一个不在当前工作目录下文件，这个目录必须在PYTHONPATH中列出。想了解更多细节，请阅读第18章。

IDLE用户界面

到目前为止，我们看到了如何通过交互提示模式、系统命令行、图标点击以及模块导入运行Python代码。如果你希望找到更可视化的方法，IDLE提供了做Python开发的用户图形界面（GUI），而且它是Python系统的一个标准并免费的部分。它往往被认为是一个集成开发环境（IDE），因为它在一个单独的界面中绑定了很多开发任务（注4）。

简而言之，IDLE是一个能够编辑、运行、浏览和调试Python程序的GUI，所有都能够在单独的界面实现。此外，由于IDLE是使用Tkinter GUI工具包开发的Python程序，可以在几乎任何Python平台上运行，包括微软Windows、X Windows（例如，Linux、UNIX以及Unix类平台）以及Mac OS（无论是Classic还是OS X）。对于很多人来说，IDLE代表了一种简单易用的命令行输入的替代方案，并且比点击图标出问题的可能性更小。

IDLE基础

让我们直接来看一个例子。在Windows中启动IDLE很容易：在开始按钮的Python菜单中进行启动（如图2-1所示），并且也能够通过右键点击Python程序图标进行选择。在UNIX类系统中，需要在命令行中启动IDLE的顶层脚本，另一种办法是通过点击位于

注4： IDLE是IDE的一个官方误用，但是其实际上是为了纪念Monty Python的成员Eirc Idle而命名的。

www.TopSage.com

Python的*Lib*目录下的*idlelib*子目录下的*idle.pyw*或*idle.py*运行（在Windows中，IDLE是位于*C:\Python25\Lib\idlelib*中的）（注5）。

图3-3显示了Windows下开始运行IDLE的场景。Python shell窗口是主窗口，一开始就会被打开，并运行交互会话（注意到>>>提示符）。这个工作起来就像完全的交互对话（在这里编写你输入的代码并能够在输入后马上运行）并且可以作为测试工具进行使用。

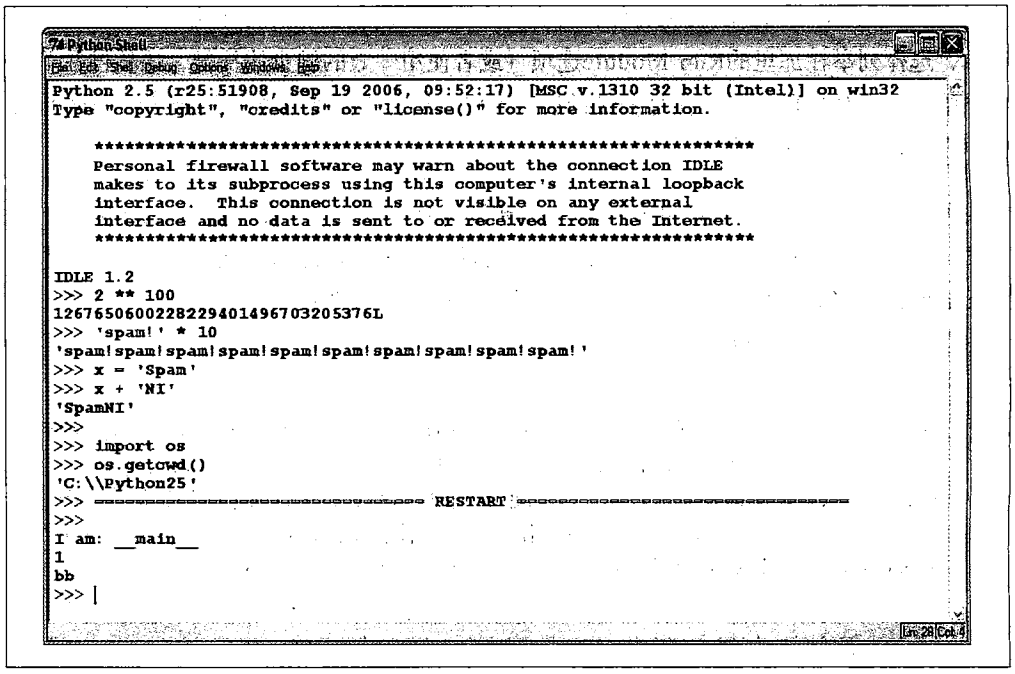


图3-3：IDLE开发GUI的主Python shell窗口，在Windows下进行。使用File菜单开始一个(新窗口)或改变（Open … ）一个源文件；使用文件编辑窗口的Run菜单去运行窗口的代码（Run Module）

IDLE可以使用友好的菜单并配合键盘快捷键进行绝大多数操作。为了在IDLE中创建（或编写）源文件，可以打开一个新的文本编辑窗口：在主窗口，选择File下拉菜单，并选择New Window来打开一个新的文本编辑窗口（或者Open…去编辑一个已存在的文

注5：IDLE是Python程序，是用标准库的Tkinter GUI工具集来创建的IDLE GUI。这使IDLE具有可移植性，但是，也意谓着你需要让Python支持Tkinter才能使用IDLE。Python的Windows版本默认支持IDLE，但有些Linux和UNIX用户可能需要安装适当的Tkinter支持工具集（yum tkinter命令在一些Linux发行版上就足够了，但是安装提示可参考附录A的细节）。Mac OS X可能已预先安装好你所需的一切。寻找机器上的idle命令或脚本。

件）。一个新窗口将会出现。这是一个IDLE文本编辑窗口，在这里创建或修改的文件的代码可以输入并显示出来。

尽管这不会在本书中进行详细的讲解，IDLE使用了语法导向的着色方法，对在主窗口输入的代码和文本编辑窗口的关键字使用的是一种颜色，常量使用的是另一种颜色。这能够帮助给代码中的组件一个更好的外观。

为了运行在IDLE中编辑的代码文件，首先选中文本编辑窗口，并点击窗口中的Run下拉菜单，选择列举在那里的Run Module选项（或者使用等效的键盘快捷键，快捷键在菜单中已给出）。如果已经在文件打开或最后一次保存后，你改变了文件的话，Python将会提醒你需要首先保存文件。

当按照这种方式运行时，脚本的输出结果或错误信息将可能在主交互窗口（Python shell窗口）生成。如图3-3所示，窗口中的最后三行就是在另外打开的独立编辑窗口的脚本的执行结果。“RESTART”信息告诉我们用户脚本的进程重新启动以运行编辑的脚本，并为独立的脚本输出做好准备。

注意： 建议：如果想要在IDLE的主窗口中重复前一条命令，可以使用Alt-P组合键回滚，找到命令行的历史记录，并用Alt-N向前寻找（在Mac上，可以试试使用Ctrl-P和Ctrl-N）。之前的命令可以重新调用并显示，并且可以编辑改变后运行。也可以通过使用游标指到命令上重新运行该命令，或使用复制粘帖的操作，但这些看起来需要花费更多力气。除了IDLE，Windows的交互模式对话环境中，可以使用方向键重新调用使用过的命令。

使用IDLE

IDLE是免费、简单易用、可移植并自动支持绝大多数平台的。本书通常向Python新手们推荐它，因为它对一些具体的细节包装起来并且不需要之前的系统命令行的经验。但是与一些更高级的商业IDE相比，它同样有一些局限。这里是一个IDLE新手应该在心中牢记的要点列表：

- **当保存文件时，必须明确地添加“.py”。**在讲到一般文件的时候提到过这一点，但是一般来讲，这是一个IDLE的障碍，特别是对于Windows用户来说。IDLE不会在文件保存时自动添加*.py*扩展名。当第一次保存文件时，需要亲自小心的输入*.py*扩展名。如果不这样的话，你仍可以从IDLE（以及系统命令行）运行文件，但是你将不再能以交互模式或从其他模块导入文件。
- **通过选择在文本编辑窗口Run→Run Module运行脚本，而不是通过交互模式的导入和重载。**本章前边，我们看到了通过交互模式导入运行一个文件是可能的。然

www.TopSage.com

而，这种机制会变得复杂，因为在文件发生改变后需要手动重载文件。与之相反，使用IDLE菜单选项的Run→Run Module总是运行文件的最新版本。如果必要的话，它同样会首先提醒你保存文件（另一个IDLE之外会发生的常见错误）。

- **你也许仍然需要重载嵌套的模块。**从技术上讲，IDLE的Run→Run Module菜单选项一般只是运行顶层文件的最新版本；当发生变化后导入的文件仍需要交互地重载。一般来说，尽管这样，Run→Run Modules还是避免了一些通常导入的一些麻烦。如果选择import和reload来替代的话，要记住使用Alt-P/Alt-N快捷键调用前一条命令。
- **可以对IDLE进行定制。**改变IDLE的字体及颜色，在任何一个IDLE窗口中选择Option菜单中的Configure选项。也可以配置快捷键组合的动作、缩进设置以及其他；参考IDLE的帮助下拉菜单以获得更多提示。
- **在IDLE中没有清屏选项。**这个选项看起来是一个经常提到的要求（也许是由于在其他的IDE中都有类似的功能选项），并且也有可能最终增加这个功能。尽管这样，目前还没有清空交互模式窗口文字的办法。如果想要清理掉窗口文字，可以一直按着回车键或者输入一个打印一系列空白行的Python循环。
- **Tkinter GUI和线程程序有可能不适用于IDLE。**因为IDLE是一个Python/Tkinter程序，如果使用它运行特定类型的高级Python/Tkinter程序，有可能会没有响应。这对于使用较新版本的IDLE在一个进程运行用户代码、在另一个进程运行IDLE GUI本身问题会变得小很多，但是一些程序仍然会发生GUI吸有响应的情况。你的代码也许不会碰到这样的问题，但是作为经验之谈，如果使用IDLE编辑GUI程序是永远安全的，最好使用其他的选项启动运行它们，例如，图标点击或系统命令行。有疑问时，如果代码在IDLE中发生错误，请在GUI外再试试。
- **如果发生了连接错误，试一下通过单个进程的模式启动IDLE。**由于IDLE要求在其独立的用户和GUI进程间通信，有时候它会在特定的平台上发生启动错误（特别是在一些Windows机器上，它会不时地出现启动错误）。如果运行时碰到了这样的连接错误，它常常可以通过系统命令行使IDLE运行在单一进程的模式下进行启动，从而避免了通信的问题：`-n`命令行标志位可以强制进入这种模式。例如，在Windows上，可以开启一个命令行提示窗口，并从*C:\Python25\Lib\idlelib*（如果必要的话，使用`cd`切换到这个目录下）运行系统命令行`idle.py -n`。
- **谨慎使用IDLE的一些可用的特性。**对于新手来说，IDLE让工作变得更容易，但是有些技巧在IDLE GUI之外并不能够使用。例如，IDLE可以在IDLE的环境中运行脚本，你代码中的变量自动在IDLE交互对话中显示：不需要总是运行`import`命令去

www.TopSage.com

获取已运行的顶层文件的变量名。这可以很方便，但是在IDLE之外的环境会让人很困惑，变量名只有在从文件中导入才能使用。

高级IDLE工具

除了基本的编辑和运行的功能，IDLE还提供了很多高级的特性，包括指向点击（point-and-click）程序调试和对象浏览器。IDLE的调试器是通过Debug菜单进行激活的，而对象浏览器是通过File菜单激活的。这个浏览器允许快速浏览模块搜索路径下的文件以及文件中的对象；通过点击文件或对象在文本编辑窗口打开对应的源代码。

IDLE调试通过选择主窗口中的“Debug→Debuger菜单选项”来启动，之后通过选择文本编辑窗口的“Run→Run Module 选项”开始运行脚本。一旦调试器生效后，你可以在文本编辑器的某一行点击右键，从而在代码中设置断点停止它的运行、显示变量值等。也可以在调试时查看程序的执行效果：在代码中执行该步时，当前运行的代码行就会被标注出来。

作为简单的调试操作，也可以使用鼠标，通过在错误信息的文字上进行右键点击来快速地跳到发生错误的那一行代码 —— 一个使得修改并重新运行代码变得简单快捷的小技巧。此外，IDLE的文本编辑器提供了丰富的、友好的工具集合，包括自动缩进、高级文本和文件搜索操作以及其他很多工具。由于IDLE使用了直观的GUI交互模式，你可以现场实验这个系统，来感受一下它的其他工具。

其他的IDE

由于IDLE是一个免费、可移植并且是Python的标准组件，如果希望使用IDE的话，它是一个不错的值得学习的首选开发工具。如果你是一个新人的话，本书建议你在本书的练习中使用IDLE，除非已经对基于命令行的开发模式非常熟悉了。尽管这样，这里有一把为Python开发者提供的IDE替代品， 与IDLE相比，其中一些工具相当强大和健全。这里是一些最常用的IDE：

Eclipse和PyDev

Eclipse是一个高级的开源IDE GUI。最初是用来作为Java IDE的，Eclipse在安装了PyDev（或类似的）插件后也能够支持Python开发。Eclipse是一个流行和强大的Python开发工具，它远远超过了IDLE的特性。其缺点就是看起来需要安装较大的系统，并且PyDev的插件需要一个共享扩展包来实现一些功能（包括集成交互终端），从严格意义上来说，它并不是开放源代码的。尽管这样，当你希望从IDLE升级时，Eclipse/PyDev这个组合是值得你注意的。

www.TopSage.com

Komodo

作为一个全功能的Python（及其他语言的）开发环境GUI，Komdodo包括了标准的语法着色、文本编辑、调试以及其他特性。此外，Komodo提供了很多IDLE所没有的高级特性，包括了项目文件、源代码控制集成、正则表达式调试和拖拽模式（drag-and-drop）的GUI构建器，可以生成Python/Tkinter代码从而交互地实现你所设计的GUI。在编写本书时，Komodo不是免费的；它在 *http://www.activestate.com* 可以下载。

PythonWin

PythonWin是一个免费的只能在Windows平台使用的免费的Python IDE，它是作为ActiveState的ActivePython版本的一部分来分发的（也可以独立从 *http://www.python.org*上获得）。大致来看，它很像IDLE，并增加了一些有用的Windows特定的扩展。例如，PythonWin支持CO对象。如今，IDLE也许比PythonWin更高级（例如，IDLE的双进程构架使其远离失去响应的现象）。尽管如此，PythonWin为Windows开发者提供了IDLE没有的工具。查看 *http://www.activestate.com* 以了解更多信息。

其他

据我所知概括计算有近十个著名的IDE（例如，WingIDE、PythonCard），还有更多也许还会不断随时涌现。事实上，目前几乎所有的程序员友好的文本编辑器对Python开发都有某种程度上的支持，无论这种支持已经预安装或是需要独立获取。例如，Emacs和Vim，都有基本的Python支持。不需要在这里罗列出全部的选择，查看 *http://www.python.org*的资源，或者在Google搜索“Python editors”（这也许会把你带到一个Wiki页面，那里包含了许多Python编程的IDE和文本编辑器的选择）。

嵌入式调用

到现在为止，我们已经看到了如何运行代码交互地输入，以及如何在系统命令行中运行保存在文件中的代码、Unix可执行脚本、图标点击、模块导入和像IDLE这样的IDE。这基本上包括了本书中提到的所有情况。

但是，在一些特定的领域，Python代码也许会在一个封闭的系统中运行。在这样的情况下，我们说Python程序被嵌入在其他程序中运行。Python代码可以保存到一个文本文件中、存储在数据库中、从一个HTML页面获取、从XML文件解析等。但是从执行的角度来看，另一个系统（而不是你）会告诉Python去运行你创建的代码。这样的嵌入执行

www.TopSage.com

模式一般用来支持终端用户定制的。例如，一个游戏程序，也许允许用户进行游戏定制（及时地在策略点存取Python代码）。

例如，从C程序中通过调用Python运行API（Python在机器上编译时创建的由库输出的一系列服务）的函数创建并运行Python代码是可行的：

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

C代码片段中，用C语言编写的程序通过连接Python解释器的库嵌入了Python解释器，并传递给Python解释器一行Python赋值语句字符串去运行。C程序也可以通过使用其他的Python API工具获取Python的对象，并处理或执行它们。

本书并不主要介绍Python/C集成，但是你应该意识到，按照你的组织打算使用Python的方式，你也许或者也许不会成为实际上运行你创建的Python程序的那个人（注6）。不管怎样，你仍将很有可能使用这里介绍过的交互和基于文件的启动技术去测试代码，那些被隔离在这些封闭系统中并最终有可能被这些系统使用的代码。

动付二进制的可执行性

如前一章所介绍的那样，冻结二进制的可执行性是集成了程序的字节码以及Python解释器为一个单个的可执行程序的包。通过这种方式，Python程序可以像其他启动的任何可执行程序一样（图标点击，命令行等）被启动。尽管这个选择对于产品的发售相当适合，但它并不是一个在程序开发阶段适宜使用的选择。一般是在发售前进行封装（在开发完成之后）。看上一章了解这种选择的更多信息。

文本编辑器启动的选择

像之前提到过的一样，尽管不是全套的IDE GUI，大多数程序员友好型文本编辑器都支持Python程序的编辑甚至运行等功能。这样的支持也许是内置的或这可通过网络获取。例如，你如果熟悉Emacs文本编辑器的话，可以在这个编辑器内部实现所有的Python编

注6：参考《Programming Python》（O'Reilly）来了解在C/C++中嵌入Python的细节。嵌入式API可以直接调用Python函数、加载模块等。此外，Jython系统可让Java程序使用基于Java的API（Python解释器类）来启用Python程序代码。

www.TopSage.com

辑以及启动功能。可以通过查看 *http://www.python.org/editors*或者在网络上搜索“Python editors”这个词来获得更多细节。

其他的启动选择

根据你所使用的平台，也许有其他的方法启动Python程序。例如，在一些Macintosh系统，你也许可以通过拖拽Python的程序文件至Python解释器的图标去让程序运行。在Windows中，你总是能够通过开始菜单中的运行…去启动Python脚本。最后，Python的标准库有一些工具允许Python程序启动其他的Python程序（例如，`execfile`、`os.popen`和`os.system`）。不过这些工具超出了本章的内容范围。

未来的可能

尽管本章反映的是当前的实际情况，其中很多都具有与平台和时间的特定性。确实，这里描述的执行和启动的细节是在本书几版的销售过程中提出来的。作为程序启动的选择，很有可能时不时的会有新的程序启动选择出现。

新的操作系统以及已存在系统的新版本，也许会提供超出这里列举的执行技术。一般来说，由于Python与这些变化保持同步，你可以通过任何对于你使用的机器合理的方式运行Python程序，无论现在还是将来 —— 通过在平板电脑或PDA上进行手写，在虚拟现实中拖拽图标，或者在与你的同事交谈中喊出脚本的名字。

实现的变换也会对启动原理有某种程度的影响。例如，一个全功能的编译器也许会生成一般的可执行文件，就像如今的frozen binary那样启动。

我应该选用哪种

这里所有的选择中，很自然就会提出一个问题：哪一种最适合我？一般来说，如果你是刚刚开始学习Python，应该使用IDLE界面做开发。它提供了一个用户友好的GUI环境，并能够隐藏一些底层配置细节。为编写脚本，它还提供了一个与平台无关的文本编辑器，而且它是Python系统中一个标准并免费的部分。

从另一面来说，如果你是一个有经验的程序员，你也许觉得这样的方式更惬意一些，简化成在一个窗口使用你选择的文本编辑器，在另一个窗口通过命令行或点击图标启动编写程序（事实上，这是作者如何开发Python程序，但是他有偏好UNIX的过去）。因为开

www.TopSage.com

发环境是很主观的选择，本书很难提供统一的标准。一般来说，你最喜欢使用的环境，往往就是最适合你用的环境。

TopSage.com

本章小结

在本章我们学习了启动Python程序的一般方法：通过交互地输入运行代码、通过系统命令行运行保存在文件中的代码、文件图标点击、模块导入以及像IDLE这样的IDE GUI。本章介绍了许多实际中入门的细节。本章的目标就是给你提供足够的信息，让你能够开工，完成我们将要开始的本书下一部分的代码。那里，我们将会以Python的核心数据类型作为开始。

首先，尽管这样，我们还会按照常规完成本章习题，练习本章学到的东西。因为这是本书这一部分的最后一章，在习题后边会紧跟着一些更完整的练习，测试你对本部分的主题的掌握程度。为了了解之后这些问题的答案，或者只是想换种思路，请查看附录B。

www.TopSage.com

本章习题

1. 怎样才能开始一个交互式解释器的会话?
2. 你应该在哪里输入系统命令行来启动一个脚本文件?
3. 指出两个在Windows下点击文件图标运行脚本的潜在的缺点。
4. 为什么需要重载模块?
5. 在IDLE中怎样运行一个脚本?
6. 列举2个使用IDLE的潜在的缺点。
7. 什么是命名空间，它和模块文件有什么关联?

习题解答

1. 在Windows下可以通过点击“开始”按钮，选择“程序”，点击“Python”，然后选择“Python (command line)”菜单选项来开始一个交互会话。在Windows下可以在系统终端窗口(在Windows下的一个命令提示窗口)输入**python**作为一条系统命令行来实现同样效果。另一种方法是启动IDLE，因为它的主Python shell窗口是一个交互式会话窗口。如果你没有设置系统的PATH变量来找到Python，你需要使用cd切换到Python安装的地方，或输入**python**的完整路径而不是仅仅python（例如，在Windows下输入**C:\Python25\python**）。
2. 在输入系统命令行的地方，也就是你所在的平台提供给作为系统终端的地方：在Windows下的系统提示符；在UNIX、Linux或Mac OS X上的xterm或终端窗口等。
3. 打印后退出的脚本会导致输出文件马上消失，在你能够看到输出之前（这也是raw_input这个技巧之所以有用的原因）；你的脚本产生的同样显示在输出窗口的错误信息,会在查看其内容前关闭（这也是对于大多数开发任务，系统命令和IDLE这类IDE之所以更好的原因）。
4. 在默认情况下，Python每个进程只会导入（载入）一个模块一次，所以如果你改变了它的源代码，并且希望在不停止或者重新启动Python的情况下运行其最新的版本，你将必须重载它。在你重载一个模块之前你至少已经导入了一次。在系统命令行中运行代码，或者通过图标点击，或者像使用IDLE这样的IDE，这不再是一个问题，因为这些启动机制往往每次都是运行源代码的最新版本。

5. 在你希望运行的文件所在的文件编辑窗口，选择窗口的Run→Run Module菜单选项。这可以将这个窗口的源代码作为顶层脚本文件运行，并在交互Python shell窗口显示其输出。

6. IDLE在运行某种程序时会失去响应 —— 特别是使用多线程（本书话题之外的一个高级技巧）的GUI程序。并且，IDLE有一些方便的特性在你一旦离开IDLE GUI时会伤害你：例如在IDLE中一个脚本的变量是自动导入到交互的作用域的，而通常的Python不是这样。

7. 命名空间就是变量（也就是变量名）的封装。它在Python中以一个带有属性的对象的形式出现。每个模块文件自动成为一个命名空间：也就是说，一个对变量的封装，这些变量对应了顶层文件的赋值。命名空间可以避免在Python程序中的命名冲突 —— 因为每个模块文件都是独立完备的命名空间，文件必须明确的导入其他的文件，才能使用这些文件的变量名。

www.TopSage.com

大家网
TopSage.com

第一部分 练习题

是自己开始编写程序的时候了。这第一部分的练习是比较简单的，但是这里的一些问题提示了未来章节的一些主题。一定要查看附录中的答案（附录B）“第一部分 使用入门程”，练习及其解答有时候包含了一些并没有在这部分主要内容中的补充信息，所以你应该看看解答，即使你能够独立回答所有的问题。

1. **交互**。使用系统命令行、IDLE或者其他的方法，开始Python交互命令行（>>>提示符），并输入表达式"Hello World!"（包括引号）。这行字符串将会回显。这个练习的目的是确保已配置Python运行的环境。在某些情况下，你也许需要首先运行一条cd shell命令，输入Python可执行文件的完整路经，或者增加Python可执行文件的路经至PATH环境变量。如果想要的话，你可以在*.cshrc*或*.kshrc*文件设置中设置PATH，使Python在Unix系统中永久可用；在Windows中，使用*setup.bat*、*autoexecb.bat*或者环境变量GUI。参照附录A获得环境变量设置的帮助。

2. **程序**。使用你选择的文本编辑器，写一个简单的包含了单个打印"Hello module world!"语句的模块文件，并将其保存为*module1.py*。现在，通过使用任何你喜欢的启动选项来运行这个文件：在IDLE中运行，点击其文件图标，在系统shell的命令行中将其传递给Python解释器程序（例如，python module1.py）等。实际上，尽可能地使用本章所讨论到的启动技术运行你的文件去实验。哪种技术看起来最简单？（这个问题没有正确的答案）

3. **模块**。紧接着，开始一个Python交互命令行（>>> prompt）并导入你在练习2中所写的模块。试着将这个文件移动到一个不同的目录并再次从其原始的目录导入（也就是说，在刚才导入的目录运行Python）。发生了什么？（提示：在原来的目录中仍然有一个*module1.pyc*的字节码文件吗？）

4. **脚本**。如果你的平台支持的话，在你的*module1.py*模块文件的顶行增加一行#!，赋予这个文件可执行的权限，并作为可执行文件直接运行它。在第一行需要包含什么？#!一般在UNIX、Linux以及Unix类平台如Mac OS X有意义；如果你在Windows平台工作，试着在DOS终端窗口不在其前边加“python”而直接列出其名字来运行这个文件（这在最近版本的Windows上有效），或者通过开始→运行… 对话框。

5. **错误**。在Python交互命令行中，试着输入数学表达式和赋值。首先输入1 / 0这个表达式。发生了什么？接下来，输入一个你还没有赋值的变量名。这次又发生了什么？

www.TopSage.com

你也许还不知道，但是你正在做的是异常处理（一个将会在第7部分探索的主题）。如同你将会在那里学到的，从技术上正在触发所谓的默认打印标准错误信息的异常处理逻辑。如果你没有获得错误信息，那么默认的处理模块获得了并作为回应打印了标准的错误信息。

作为一个完整的源代码调试工具，IDLE包括GUI调试界面（在本章的“高级IDLE工具”介绍过），并且又一个Python的名为pdb标准库模块提供了一个命令行调试界面（你可以在标准库手册中找到更多信息）。当开始以后，Python的默认错误信息将可能尽你所需求的那样提供更多的错误处理 —— 它们会提供导致错误的原因，并在错误发生时显示代码所在的那一行。

6. 中断。在Python命令行中，输入：

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

发生了什么？如果使用的是比1.5版更新的Python，你也许能够看到一个奇怪的输出，我们将会在本书的下一部分讲到。如果你用的Python版本比1.5.1还旧，在绝大多数平台上Ctrl-C组合键也许会有用。为什么会发生这种情况？当输入Ctrl-C组合键时Python提示了什么？

警告：如果你有一个比1.5.1版更老的Python，在运行这个测试之前，保证你的机器能够通过中断组合键中止一个程序，不然的话你得等很长时间。

---

7. 文档。在你继续感受标准库中可用的工具和文档集的结构之前，至少花17分钟浏览一下Python库和语言手册。为了熟悉手册集的主题，至少得花这么长的时间。一旦你这样做了，将很容易找到你所需要的东西。你可以在Windows的“开始”按钮的Python中，或者通过在IDLE的Help下拉菜单中的“Python Docs”选项，或者在网上 *http://www.python.org/doc*找到这个手册。本书将会在第14章用部分内容描述一些手册及其他可用文档资源中的内容（包括PyDoc以及help函数）。如果还有时间，去Python的网站以及Vaults of Parnassus 和PyPy第三方扩展的网站看看。特别留意Python.org的文档以及搜索页面，它们是至关重要的资源。

www.TopSage.com

第二部分

类型和运算

www.TopSage.com

大家网
TopSage.com

www.TopSage.com

第4章

介绍Python对象类型

本章我们将开始学习Python语言。从非正式的角度来说，在Python中，我们使用一些东西在做事情。“事情”采用的是像加法以及连接这样的操作形式，而“东西”指的就是我们操作的对象。本书这一部分，我们将注意力集中在“东西”，以及我们的程序用这些“东西”可以做的事情。

从更正式的角度来讲，在Python中，数据以对象的形式出现 —— 无论是Python提供的内置对象，还是使用Python或是像C扩展库这样的扩展语言工具创建的对象。尽管在以后才能确定这一概念，但对象无非是内存中的一部分，包含数值和相关操作的集合。

由于对象是Python中最基本的概念，从这一章开始我们将会全面地体验python的内置对象类型。

通过这样的介绍，让我们先建立一个图像，来简单地说明这一章如何符合Python全景。从一个更具体的视角来看，Python程序可以分解成模块、语句、表达式以及对象，如下所示。

1. 程序由模块构成。
2. 模块包含语句。
3. 语句包含表达式。
4. 表达式建立并处理对象。

在第3章中对模块的讨论介绍了这个等级层次中的最高一层。这部分的章节将会从底层开始，探索编程过程中使用的内置对象以及表达式。

www.TopSage.com

为什么使用内置类型

如果你使用过底层语言C或C++，应该知道很大一部分工作集中于用对象（或者称作数据结构）去表现应用领域的组件。需要部署内存结构、管理内存分配、实现搜索和读取例程等。这些工作听起来非常乏味（并容易出错），并且往往背离程序的真正目标。

在典型的Python程序中，这些令人头痛的大部分工作都消失了。因为Python提供了强大的对象类型作为语言的组成部分，在你开始解决问题之前往往没有必要编写对象的实现。事实上，除非你有内置类型无法提供的特殊对象要处理，最好总是使用内置对象而不是使用自己的实现。下面是其原因。

- **内置对象使程序更容易编写。**对于简单的任务，内置类型往往能够表现问题领域的所有结构。免费得到了如此强大的工具，例如，集合（列表）和搜索表（字典），可以马上使用它们。仅使用内置对象类就能够完成很多工作。
- **内置对象是扩展的组件。**对于较为复杂的任务，或许仍需要提供你自己的对象，使用Python的类或C语言的接口。但就像本书稍后要介绍的内容，人工实现的对象往往建立在像列表和字典这样的内置类型的基础之上。例如，堆栈数据结构也许会实现为管理和定制内置列表的类。
- **内置对象往往比定制的数据结构更有效率。**在速度方面，Python的内置类型优化了用C实现数据结构算法。尽管可以实现属于自己的类似的数据类型，但往往很难达到内置数据类型所提供的性能水平。
- **内置对象是语言的标准的一部分。**从某种程度上来说，Python不但借鉴了依靠内置工具的语言（例如，LISP），而且汲取了那些依靠程序员去提供自己实现的工具或框架的语言（例如，C++）的优点。尽管在Python中可以实现独一无二的对象类型，但在开始阶段并没有必要这样做。此外，因为Python的内置工具是标准的，它们一般都是一致的。另一方面，独创的框架则在不同的环境都有所不同。

换句话说，与我们毫无经验所创建的工具相比，内置对象类型不仅仅让编程变得更简单，而且它们也更强大和更高效。无论你是否创建新的对象类型，内置对象都构成了每一个Python程序的核心部分。

Python的核心数据类型

表4-1是Python的内置对象类型和一些编写其常量所使用到的语法，也就是能够生成这

www.TopSage.com

TopSage.com

些对象的表达式（注1）。如果你使用过其他语言，其中的一些类型也许对你来说很熟悉。例如，数字和字符串分别表示数学和文本的值，而文件提供了处理保存在计算机上的文件的接口。

表4-1：内置对象预览

对象类型	例子 常量/创建
数字	1234, 3.1415, 999L, 3+4j, Decimal
字符串	'spam', "guido's"
列表	[1, [2, 'three'], 4]
字典	{'food': 'spam', 'taste': 'yum'}
元组	(1,'spam', 4, 'U')
文件	myfile = open('eggs', 'r')
其他类型	集合、类型、None、布尔型

表4-1所列内容并不完整，因为在Python程序中处理的每样东西都是一种对象。例如，在Python中进行文本模式匹配时，创建了模式对象，还有进行网络脚本编程时，使用了套接字对象。其他的类型的对象往往都是通过导入或使用模块来建立的，而且它们都有各自的行为。

我们把在表4-1中的对象类型称作是核心类型，因为它们是在Python语言内部高效创建的，也就是说，有一些特定语法可以生成它们。例如，运行下面的代码：

```
>>> 'spam'
```

从技术上讲，你正在运行一个常量表达式，这里生成并返回了一个新的字符串对象。这是Python用来生成这个对象的一个特定语法。类似地，一个方括号中的表达式会生成一个列表，在大括号中会建立一个字典等。尽管这样，就像我们看到的那样，Python中没有类型声明，运行的表达式，决定了建立和使用的对象的类型。事实上，在Python语言中，就像在表4-1中的对象生成表达式一样是这些类型起源的地方。

同等重要的是，一旦创建了一个对象，它就和操作集合绑定了 —— 只可以对字符串进行字符串相关的操作，对列表进行列表相关的操作。就像你将会学到的，Python是动态类型的（它自动地跟踪你的类型而不是要求声明代码），但是它也是强类型语言（你只能对一个对象进行有效的操作）。

注1：本书中，常量（*literal*）是指其语法会生成对象的表达式，有时也叫做常数（*constant*）。值得注意的是，“常数”不是指不可变的对象或变量（这个术语与在C++中的const，或Python中的“不可变”这个概念没有什么关系；本章稍后会讨论这个概念）。

www.TopSage.com

在功能上，表4-1中的对象类型可能比你习惯的类型更常用，也更强大。例如，你会发现列表和字典就是强大的数据表现工具，省略了在使用底层语言的过程中，为了支持集合和搜索而引入的绝大部分工作。简而言之，列表提供了其他对象的有序集合，而字典是通过键存储对象的。列表和字典都可以嵌套，可以随需求扩展和删减，并能够包含任意类型的对象。

我们将会在下面章节学习表4-1中的每一个对象类型。在我们深入介绍细节之前，我们迅速地浏览在实际应用中的Python的核心对象。本章的其他部分将提供一些操作，而这些操作在本章以后的章节我们将会进行更深入地学习。别指望在这里能够找到所有的答案。本章的目的仅仅是激发你学习的欲望并介绍一些核心的概念。好了，最好的入门方法就是迈出第一步，所以让我们看一些真正的代码吧。

数字

如果你过去曾经编写过程序或脚本，表4-1中的一些对象类型看起来比较眼熟。即使你没有编程经验，数字也是比较直接的。Python的核心对象的设置包括常规的类型：整数（没有小数部分的数字）、浮点数（概括地讲，就是后边有小数部分的数字）以及更为少见的类型（无限精度“长”整型、有虚部的复数、固定精度的十进制数以及集合等）。

尽管提供了一些多样的选择，Python的基本数字类型还是相当基本的。Python中的数字支持一般的数学运算。例如，加号（+）代表加法，星号（*）在乘法中使用，双星号（**）在乘方运算中使用。

```
>>> 123 + 222                          # 整数加法
345
>>> 1.5 * 4                            # 浮点数乘法
6.0
>>> 2 ** 100                           # 2的100次方
1267650600228229401496703205376L
```

注意这里的最后一个运算结果末尾的那个L —— 当需要有额外的精度时，Python自动将整型变换升级成为长整型。例如，你可以在Python中计算2的1,000,000次幂（但是你也许不应该打印结果：有300，000个数字以上，你可得等一会了！）。注意当某个浮点数打印时发生的情况。

```
>>> 3.1415 * 2                         # repr: as code
6.2830000000000004
>>> print 3.1415 * 2                   # str: user-friendly
6.283
```


www.TopSage.com

第一个结果并不是bug；这是显示的问题。这证明有两种办法打印对象：全精度（就像这里的第一个结果显示的那样）以及用户友好的形式（就像第二个）。一般来说，第一种形式认作是对象的代码形式repr，第二种是它的用户友好形式str。当我们使用类时，这两者的区别将会表现出来。现在，如果有些东西看起来比较奇怪，试试使用打印语句显示它。

除了表达式，和Python一起分发的还有一些常用的数学模块。

```
>>> import math
>>> math.pi
3.1415926535897931
>>> math.sqrt(85)
9.2195444572928871
```

math模块包括了作为函数的更高级的数学工具，而random模块可以作为随机数字的生成器和随机选择（这里，从Python列表中选择，将会在本章介绍）。

```
>>> import random
>>> random.random()
0.59268735266273953
>>> random.choice([1, 2, 3, 4])
1
```

Python还包括了一些较为少见的数字对象。例如复数、固定精度十进制数和集合，第三方开源扩展领域甚至包含了更多（矩阵和向量）。在本书稍后部分我们将会对这些类型进行讨论。

到现在为止，我们已经把Python作为简单的计算器来使用。想要更好的利用内置类型的话，就让我们继续介绍字符串。

字符串

就像任意字符的集合一样，字符串是用来记录文本信息的。它们是在Python中作为序列(也就是说，一个包含其他对象的有序集合)提到的第一个例子。序列中的元素包含了一个从左到右的顺序 —— 序列中的元素根据它们的相对位置进行存储和读取。从严格意义上来说，字符串是一个单个字符的字符串的序列，其他类型的序列还包括了列表和元组（稍后介绍）。

序列的操作

作为序列，字符串支持假设其中各元素包含位置顺序的操作。例如，如果我们有一个含

有四个字符的字符串，我们通过内置的len函数验证其长度并通过索引操作得到其各个元素。

TopSage.com

```
>>> S = 'Spam'
>>> len(S)                  # Length
4
>>> S[0]                    # The first item in S, indexing by zero-based position
'S'
>>> S[1]                    # The second item from the left
'p'
```

在Python中，索引是按照从最前面的偏移量进行编码的，也就是从0开始，第一项索引为0，第二项索引为1,依此类推。在Python中，我们能够反向索引，从最后一个开始。

```
>>> S[-1]                   # The last item from the end in S
'm'
>>> S[-2]                   # The second to last item from the end
'a'
```

一般来说，一个负的索引号简单地与字符串的长度相加，得到两个操作是等效的（尽管第一个要更容易编写并不容易发生错误）：

```
>>> S[-1]                   # The last item in S
'm'
>>> S[len(S)-1]             # Negative indexing, the hard way
'm'
```

值得注意的是，我们能够在方括号中使用任意表达式，不仅仅可以使用数字常量——只要Python能够获取一个值，我们可以用一个常量、一个变量或任意表达式。Python的语法在这方面是完全通用的。

除了简单的从位置进行索引，序列也支持一种所谓分片（slice）的操作，这是一种一步就能够提取整个分片（slice）的方法。例如：

```
>>> S                       # A 4-character string
'Spam'
>>> S[1:3]                  # Slice of S from offsets 1 through 2 (not 3)
'pa'
```

也许认识分片最简单的办法就是把它们看作是从一个字符串中一步就提取出一部分的方法。它们的一般形式为X[I:J]，表示“取出在X中从偏移为I，直到但不包括J的内容”。结果就是返回一个新的对象。例如，上边的最后一个操作，给我们在字符串S中从偏移1到2（也就是，3-1）的所有字符作为一个新的字符串。效果就是切片或“分离出”中间的两个字符。

www.TopSage.com

在一个分片中，左边界默认为0，并且右边界默认为分片序列的长度。这引入了一些常用的变化：

```
>>> S[1:]                    # Everything past the first (1:len(S))
'pam'
>>> S                        # S itself hasn't changed
'Spam'
>>> S[0:3]                   # Everything but the last
'Spa'
>>> S[:3]                    # Same as S[0:3]
'Spa'
>>> S[:-1]                   # Everything but the last again, but simpler (0:-1)
'Spa'
>>> S[:]                     # All of S as a top-level copy (0:len(S))
'Spam'
```

注意负偏移量也可以用作分片的边界，并且在最后一个操作中有效地拷贝整个字符串。正像今后将学到的那样，没有必要拷贝一个字符串，但是这种操作形式在列表这样的序列中很有用。

最后，作为一个序列，字符串也支持使用加号进行合并（将连个字符串合成为一个新的字符串），或者重复（通过再重复一次创建一个新的字符串）：

```
>>> S
'Spam'
>>> S + 'xyz'                # Concatenation
'Spamxyz'
>>> S                        # S is unchanged
'Spam'
>>> S * 8                    # Repetition
'SpamSpamSpamSpamSpamSpamSpamSpam'
```

注意到加号（+）对于不同的对象有不同的意义：对于数字为加法，对于字符串为合并。这是Python的一般特性，也就是我们将会在本书的后面提到的多态。简而言之，一个操作的意义取决于被操作的对象。正如在学习动态类型时看到的那样，这种多态的特性给Python代码带来了很大的简洁性和灵活性。由于类型并不受约束， Python编写的操作通常可以自动地适用于不同类型的对象，只要它们支持一种兼容的接口（就像这里的+操作一样）。这成为Python中很重要的概念。关于这方面，你将会在后面的学习中学到更多的内容。

不可变性

注意：在之前的例子中，没有通过任何操作对原始的字符串进行改变。每个字符串都被定义为生成新的字符串作为其结果，因为字符串在Python中具有不可变性 —— 在其创建

www.TopSage.com

后值不能改变。例如，你不能通过对其某一位置进行赋值而改变字符串，但是你总是可以通过建立一个新的字符串并以同一个变量名对其进行赋值。因为Python在运行过程中会清理旧的对象（之后你将会看到），这并不像它听起来那么复杂：

```
>>> S
'Spam'
>>> S[0] = 'z'                  # Immutable objects cannot be changed
...error text omitted...
TypeError: 'str' object does not support item assignment

>>> S = 'z' + S[1:]             # But we can run expressions to make new objects
>>> S
'zpam'
```

在Python中的每一个对象都可以分为不可变性或者可变性。在核心类型中，数字、字符串和元组是不可变的；列表和字典不是这样（它们可以完全自由地改变）。在其他方面，这种不可变性可以用来保证在程序中保持一个对象固定不变。

类型特定的方法

目前我们学习过的每一个字符串操作都是一个真正的序列操作。也就是说，这些操作在Python中的其他序列中也会工作，包括列表和元组。尽管这样，除了一般的序列操作，字符串还有独有的一些操作作为方法存在（对象的函数，将会通过一个调用表达式触发）。

例如，字符串的`find`方法是一个基本的子字符串查找的操作（它将返回一个传入子字符串的偏移量，或者没有找到的情况下返回`-1`），而字符串的`replace`方法将会对全局进行搜索和替换。

```
>>> S.find('pa')                # Find the offset of a substring
1
>>> S
'Spam'
>>> S.replace('pa', 'XYZ')      # Replace occurrences of a substring with another
'SXYZm'
>>> S
'Spam'
```

尽管这些字符串方法的命名有改变的含义，但在这里我们都不会改变原始的字符串，而是会创建一个新的字符串作为结果——因为字符串具有不可变性，我们必须这样做。字符串方法将是Python中文本处理的头号工具。其他的方法还能够实现通过分隔符将字符串拆分为子字符串（作为一种解析的简单形式），大小写变换，测试字符串的内容（数字、字母或其他），去掉字符串组后的空格字符。

www.TopSage.com

```
>>> line = 'aaa,bbb,ccccc,dd'
>>> line.split(',')              # Split on a delimiter into a list of substrings
['aaa', 'bbb', 'ccccc', 'dd']

>>> S = 'spam'
>>> S.upper()                    # Upper- and lowercase conversions
'SPAM'

>>> S.isalpha()                  # Content tests: isalpha, isdigit, etc.
True

>>> line = 'aaa,bbb,ccccc,dd\n'
>>> line = line.rstrip()         # Remove whitespace characters on the right side
>>> line
'aaa,bbb,ccccc,dd'
```

注意：尽管序列操作是通用的，但方法不通用（字符串的方法只能用于字符串）。一条简明的法则是这样的：可作用于多种类型的通用型操作都是以内置函数或表达式的形式出现的［例如，`len(X)`，`X[0]`］，但是类型特定的操作是以方法调用的形式出现的［例如，`aString.upper()`］。如果经常使用Python，你会更顺利地从这些分类中找到你所需要的工具，下一节将会介绍一些马上能使用的技巧。

寻求帮助

上一节介绍的方法很具有代表性，但是仅仅是少数的字符串的例子而已。一般来说，这本书看起来并不是要详尽地介绍对象方法的。对于更多细节，你可以调用内置的`dir`函数，将会返回一个列表，其中包含了对象的所有属性。由于方法是函数属性，它们也会在这个列表中出现：

```
>>> dir(S)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__getslice__',
'__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__', '__mod__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__str__', 'capitalize', 'center',
'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'index',
'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper',
'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

也许只有在本书的稍后部分你才会对这个列表中变量名中有下划线的内容感兴趣，那时我们将在类中学习重载：它们代表了字符串对象的实现方式，并支持定制。一般来说，以双下划线开头并结尾的变量名是用来表示Python实现细节的命名模式。而这个列表中没有下划线的属性是字符串对象能够调用的方法。

www.TopSage.com

dir函数简单地给出了方法的名称。查询它们是做什么的，你可以将其传递给help函数。

```
>>> help(S.index)
Help on built-in function index:

index(...)
    S.index(sub [,start [,end]]) -> int

    Like S.find() but raise ValueError when the substring is not found.
```

就像PyDoc一样（一个从对象中提取文档的工具），help是一个随Python一起分发的面向系统代码的接口。本书后面你将会发现PyDoc也能够将其结果生成HTML格式。

你也能够对整个字符串提交帮助查询［例如，help(S)］，但是比起你所看到的也许需要更多帮助(例如，对于每一个字符串方法的详细信息)。一般最好去查询一个特定的方法，就像我们上边所做的那样。

想获得更多细节，你可以参考Python的标准库参考手册，或者商业出版的参考书，但是dir和help是Python文档的首要选择。

编写字符串的其他方法

到目前位置，我们学习了字符串的序列操作和类型特定的方法。Python还提供了各种方式的编写字符串的方法，我们将会在下面进行更深入的介绍（例如，反斜线转义序列表示特殊的字符）。

```
>>> S = 'A\nB\tC'          # \n is end-of-line, \t is tab
>>> len(S)                 # Each stands for just one character
5

>>> ord('\n')              # \n is a byte with the binary value 10 in ASCII
10

>>> S = 'A\0B\0C'          # \0, the binary zero byte, does not terminate the string
>>> len(S)
5
```

Python允许字符串包括在单引号或双引号中（它们代表着相同的东西）。它也能够在三个引号（单引号或双引号）中表示多行字符串的形式。当采用这种形式的时候，所有的行都合并在一起，并在每一行的末尾增加了换行符。这是一个微妙的语法上的便捷，但是在Python脚本中嵌入像HTML或XML这样的内容时，它是很方便的。

```
>>> msg = """
aaaaaaaaaaaaa
```


www.TopSage.com

```
bbb'''bbbbbbbbbb""bbbbbbb'bbbb
ccccccccccccc"""

>>> msg
'\naaaaaaaaaaaaa\nbbb\'\'\'bbbbbbbbbb""bbbbbbb\'bbbb\ncccccccccccccc'
```

Python也支持一种“raw”字符串常量的写法，可以去掉反斜线转义机制（它们是以字母“*r*”开头的），并支持Unicode字符串形式从而支持国际化（它们是以字母*u*开头并包含了多字节字符）。从技术上讲，Unicode字符串是一种特殊的数据类型而不是一般的字符串，但是它支持所有的字符串操作。我们将会在后边的章节学到这些特殊的字符串。

模式匹配

在我们继续学习之前，值得关注的一点就是字符串对象的方法能够支持基于模式的文本处理。文本的模式匹配是本书范围之外的一个高级工具，但是有其他脚本语言背景的读者也许对在Python中进行模式匹配很感兴趣，我们需要导入一个名为re的模块。这个模块包含了类似搜索、分割和替换等调用，但是因为使用模式去定义子字符串，可以更通用一些：

```
>>> import re
>>> match = re.match('Hello[ \t]*(.*)world', 'Hello    Python world')
>>> match.group(1)
'Python '
```

这个例子的目的搜索子字符串，这个子字符串以“Hello,”并包含了零个或几个制表符或空格，接着有任意字符并将其保存至匹配的group中，并最后以“world.”结尾。如果找到了这样的子字符串，与模式中括号包含的部分与匹配的子字符串的对应部分保存为组。例如，下面的模式取出了三个被斜线所分割的组：

```
>>> match = re.match('/(.*)/(.*)/(.*)', '/usr/home/lumberjack')
>>> match.groups()
('usr', 'home', 'lumberjack')
```

模式匹配本身是一个相当高级的文本处理工具，但是在Python中还支持了更高级的语言处理工具，包括自然语言处理等。不过，我们已经在这个教程中介绍了足够多的字符串了，所以让我们开始介绍下一个类型吧。

列表

Python的列表对象是这个语言提供的最通用的序列。列表是一个任意类型的对象的位置相关的有序集合，它没有固定的大小。不像字符串，其大小是可变的，通过对偏移量进行赋值以及其他各种列表的方法进行调用，列表确实能够修改其大小。

www.TopSage.com

序列操作

由于列表是序列中的一种，列表支持所有的我们对字符串所讨论过的序列操作。唯一的区别就是其结果往往是列表而不是字符串。例如，有一个有三个元素的列表：

```
>>> L = [123, 'spam', 1.23]          # A list of three different-type objects
>>> len(L)                           # Number of items in the list
3
```

能够对列表进行索引，切片等操作，就像对字符串所做的操作那样：

```
>>> L[0]                             # Indexing by position
123

>>> L[:-1]                           # Slicing a list returns a new list
[123, 'spam']

>>> L + [4, 5, 6]                    # Concatenation makes a new list too
[123, 'spam', 1.23, 4, 5, 6]

>>> L                                # We're not changing the original list
[123, 'spam', 1.23]
```

类型特定的操作

Python的列表与其他语言中的数组有些类似，但是列表要强大得多。其中一个方面就是，列表没有固定类型的约束。例如，上个例子中接触到的列表，包含了三个完全不同类型的对象（一个整数、一个字符串，以及一个浮点数）。此外，列表能够按照需要增加或减小列表大小，来响应其特定的操作：

```
>>> L.append('NI')                   # Growing: add object at end of list
>>> L
[123, 'spam', 1.23, 'NI']

>>> L.pop(2)                         # Shrinking: delete an item in the middle
1.23

>>> L                                # "del L[2]" deletes from a list too
[123, 'spam', 'NI']
```

这里，列表的append方法扩充了列表的大小并在列表的尾部插入一项；pop方法（或者等效的del语句）移除给定偏移的一项，从而让列表减小。其他的列表方法可以在任意位置插入（insert）元素,按照值移除（remove）元素等。因为列表是可变的，大多数列表的方法都会改变列表对象，而不是创建一个新的列表：

```
>>> M = ['bb', 'aa', 'cc']
>>> M.sort()
```


www.TopSage.com

```
>>> M
['aa', 'bb', 'cc']

>>> M.reverse()
>>> M
['cc', 'bb', 'aa']
```

例如，这里的列表sort方法，默认按照升序对列表进行排序，而reverse对列表进行了翻转。这种情况下，这些方法都直接对列表进行了改变。

边界检查

尽管列表没有固定的大小，Python仍不允许引用不存在的元素。

```
>>> L
[123, 'spam', 'NI']

>>> L[99]
...error text omitted...
IndexError: list index out of range

>>> L[99] = 1
...error text omitted...
IndexError: list assignment index out of range
```

这是有意而为之的，由于去给一个列表边界外的元素赋值，往往会得到一个错误（而在C语言中情况比较糟糕，因为它不会像Python这样进行错误检查）。在Python中，并不是默默地增大列表作为响应，而会提示错误。为了让一个列表增大，我们可以调用append这样的列表方法。

嵌套

Python核心数据类型的一个优秀的特性就是它们支持任意的嵌套。能够以任意的组合对其进行嵌套，并可以多深层次的嵌套都可以（例如，能够让一个列表包含一个字典，并在这个字典中包含另一个列表等）。这种特性的一个直接的应用就是实现矩阵，或者Python中的“多维数组”。一个嵌套列表的列表能够完成这个基本的操作：

```
>>> M = [[1, 2, 3],              # A 3 x 3 matrix, as nested lists
         [4, 5, 6],
         [7, 8, 9]]
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

这里，我们编写了一个包含了三个其他列表的列表。其效果就是表现了一个3×3的数字矩阵。这样的结构可以通过多种方法获取元素。

www.TopSage.com

```
>>> M[1]                          # Get row 2
[4, 5, 6]

>>> M[1][2]                       # Get row 2, then get item 3 within the row
6
```

这里的第一个操作读取了整个第二行，第二个操作读取了那行的第三个元素。串联起索引操作可以逐层深处地获取嵌套的对象结构（注2）。

列表解析

处理序列的操作和列表的方法中，Python还包括了一个更高级的操作,称作列表解析表达式（list comprehension expression），从而提供了一种处理像矩阵这样结构的强大工具。例如，假设我们需要从列举的矩阵中提取出第二列。因为矩阵是按照行进行存储的，所以可以通过简单的索引即可获取行，使用列表解析可以同样简单地获得列。

```
>>> col2 = [row[1] for row in M]          # Collect the items in column 2
>>> col2
[2, 5, 8]

>>> M                                     # The matrix is unchanged
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

列表解析源自集合的概念。它是一种通过在一个序列中运行一个表达式而创建的一个新列表，每次一个，从左至右。列表解析是编写在方括号中的（提醒你在创建列表这个事实），并且有表达式和循环结构都使用了一个变量名（这里是`row`）。之前的这个列表解析表达基本上就是它字面上所讲的："一把在矩阵M中的每个row中的row[1]，放在一个新的列表中"。其结果就是一个新的列表包含了矩阵的第二列。

实际应用中的列表解析可以更复杂：

```
>>> [row[1] + 1 for row in M]             # Add 1 to each item in column 2
[3, 6, 9]

>>> [row[1] for row in M if row[1] % 2 == 0]   # Filter out odd items
[2, 8]
```

注2：这种矩阵结构适用于小规模的任务，但对于更重要的数值运算而言，你可能会想要使用Python数值扩展包中的工具，例如开源NumPy系统。这样的工具能以更高效的方式储存并处理大型矩阵，胜过我们的嵌套列表结构。NumPy被称为把Python变成对等于MatLab系统对等的免费版本，而且功能更强大。此外，诸如NASA、Los Alamos以及JPMorgan Chase这些机构都使用这个工具以从事科学和金融工作。搜索互联网以了解更多细节。

www.TopSage.com

例如，这里的第一个操作，把它搜集到的每一个元素都加了1，第二个使用了一个if条件语句,通过使用%求余表达式（取余数）过滤了结果中的奇数。列表解析创建了新的列表作为结果，但是能够在任何可迭代的对象上进行迭代。例如，这里我们将会使用列表解析去步进一个硬编码的列表的坐标和一个字符串：

```
>>> diag = [M[i][i] for i in [0, 1, 2]]          # Collect a diagonal from matrix
>>> diag
[1, 5, 9]

>>> doubles = [c * 2 for c in 'spam']            # Repeat characters in a string
>>> doubles
['ss', 'pp', 'aa', 'mm']
```

列表解析比较复杂,本书不过多讲述了。这个简要说明的目的是描绘出Python中有简单的工具，也有高级的工具。列表解析是一个可选的特性，需要在实际中有用，并常常具有处理速度上的优势。它们也能够在Python的任何的序列类型中发挥作用，甚至一些不属于序列的类型。你将会在本书后面学到更多这方面的内容。

字典

Python中的字典是完全不同的东西（参考Monty Python）：它们不是序列，而是一种映射。映射是一个其他对象的集合，但是它们是通过键而不是相对位置来存储的。实际上，映射并没有任何可靠的从左至右的顺序。它们简单地将键映射到值。字典是Python核心对象集合中的唯一的一种映射类型，也具有可变性——可以改变，并可以随需求增大或减小，就像列表那样。

映射操作

作为常量编写时，字典编写在大括号中，并包含了一系列的“键:值”对。在我们需要将键与一系列值相关联（例如，为了表述某物的某属性）的时候，字典是很有用的。作为一个例子，请思考下面的包含三个元素的字典（键分别为“food”、“quantity”和“color”）：

```
>>> D = {'food': 'Spam', 'quantity': 4, 'color': 'pink'}
```

我们可以通过键对这个字典进行索引来读取或改变键所关联的值。字典的索引操作使用的是和序列相同的语法，但是在方括号中的元素是键，而不是相对位置。

```
>>> D['food']                                    # Fetch value of key 'food'
'Spam'
```

```
>>> D['quantity'] += 1                  # Add 1 to 'quantity' value
>>> D
{'food': 'Spam', 'color': 'pink', 'quantity': 5}
```

尽管可以使用大括号这种常量形式，最好还是见识一下不同的创建字典的方法。例如，下面开始一个空的字典，然后每次以一个键来填写它。不像在列表中的边界外的赋值是禁止的，对一个新的字典的键赋值会创建该键：

```
>>> D = {}
>>> D['name'] = 'Bob'                   # Create keys by assignment
>>> D['job']  = 'dev'
>>> D['age']  = 40

>>> D
{'age': 40, 'job': 'dev', 'name': 'Bob'}

>>> print D['name']
Bob
```

在这里，我们实际上是使用字典中的键，像名字的字段一样去记录描述某人。在另一个应用中，字典也可以用来执行搜索。通过键索引一个字典往往是Python中编写搜索的最快方法。

重访嵌套

在上个的例子中，我们使用字典去描述一个假设的人物，用了三个键。尽管这样，假设信息更复杂一些。也许我们需要去记录名（first name）和姓（last name），并有多个工作（job）的头衔。事实上这产生了另一个Python对象嵌套的应用。下边的这个字典，一次将所有内容编写进一个常量，将可以记录更多的结构信息。

```
>>> rec = {'name': {'first': 'Bob', 'last': 'Smith'},
           'job':  ['dev', 'mgr'],
           'age':  40.5}
```

在这里，在顶层再次使用了三个键的字典（键分别是“name”、“job”和“age”），但是值的情况变得复杂得多：一个嵌套的字典作为name的值，支持了多个部分，并用一个嵌套的列表作为job的值从而增加多个角色和未来的扩展。能够获取这个结构的组件，就像之前在矩阵中所做的那样，但是这次索引的是字典的键，而不是列表的偏移。

```
>>> rec['name']                         # 'Name' is a nested dictionary
{'last': 'Smith', 'first': 'Bob'}

>>> rec['name']['last']                 # Index the nested dictionary
'Smith'
```


www.TopSage.com

```
>>> rec['job']                          # 'Job' is a nested list
['dev', 'mgr']

>>> rec['job'][-1]                      # Index the nested list
'mgr'

>>> rec['job'].append('janitor')        # Expand Bob's job description in-place
>>> rec
{'age': 40.5, 'job': ['dev', 'mgr', 'janitor'], 'name': {'last': 'Smith', 'first':
'Bob'}}
```

注意这里的最后一个操作是如何扩展嵌入job列表的。因为job列表是字典所包含的一部分独立的内存，它可以自由地增加或减少（对象的内存部署将会在本书稍后部分进行讨论）。

介绍这个例子的真正原因是为了说明Python核心数据类型的灵活性。就像你所看到的那样，嵌套允许直接并轻松地建立复杂的信息结构。使用C这样的底层语言建立一个类似的结构，将会很枯燥复杂并会使用更多的代码 —— 我们将不得不去实现安排并且声明结构和数组，填写值，将每一个都连接起来等。在Python中，这所有的一切都是自动完成的 —— 运行表达式创建了整个的嵌套对象结构。事实上，这是Python这样的脚本语言的主要优点之一。

同样重要的是，在底层语言中，当我们不再需要该对象时，必须小心地去释放掉所有对象空间。在Python中，当最后一次引用对象后（例如，将这个变量用其他的值进行赋值），这个对象所有的占用的内存空间将会自动清理掉：

```
>>> rec = 0                             # Now the object's space is reclaimed
```

从技术来说，Python具有一种所谓垃圾收集的特性，在程序运行时可以清理不再使用的内存，并将你从必须管理代码中这样的细节中解放出来。在Python中，一旦一个对象的最后一次引用被移除，空间将会立即回收。我们将会在本书后边学习这是如何工作的。目前，知道能够自由地使用对象就足够了，不需要为创建它们的空间或不再使用时清理空间而担心（注3）。

键的排序：for 循环

就像我们将要看到的那样，作为映射，字典仅支持通过键获取元素。尽管这样，在各种常见的应用场合，通过调用方法，它们也支持类型特定的操作。

注3：补充说明：记住，当我们采用Python的对象持久化系统时（在文件或键值数据库中容易的保存Python原生对象的方式），我们刚刚创建的rec记录，很有可能是数据库记录。这里我们不会再深入讨论，你可以参考Python的pickle和shelve模块的细节。

本书之前提到过，因为字典不是序列，它们并不包含任何可靠的从左至右的顺序。这意味着如果我们建立一个字典，并将它打印出来，它的键也许会以与我们输入时的不同的顺序出现：

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D
{'a': 1, 'c': 3, 'b': 2}
```

那么，如果在一个字典的元素中，我们确实需要强调一个顺序的时候，应该怎样做呢？一个简单的解决办法就是通过字典的keys方法收集一个键的列表，并使用列表的sort方法，然后使用Python的for循环逐个进行显示结果：

```
>>> Ks = D.keys()                    # Unordered keys list
>>> Ks
['a', 'c', 'b']

>>> Ks.sort()                        # Sorted keys list
>>> Ks
['a', 'b', 'c']

>>> for key in Ks:                   # Iterate though sorted keys
        print key, '=>', D[key]

a => 1
b => 2
c => 3
```

这是一个有三个步骤的处理，然而，就像我们将会在稍后的章节中看到的那样，在最近版本的Python中，通过使用最新的sorted内置函数（sorted返回结果并对对象类型进行排序）可以一步完成：

```
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> for key in sorted(D):
        print key, '=>', D[key]

a => 1
b => 2
c => 3
```

这种情况是要学习Python 的for循环的理由。for循环是一个遍历一个序列中的所有元素并按顺序对每一元素运行一些代码的简单并有效的方法。一个用户定义的循环变量（这里是key）用作每次运行过程中当前元素的参考量。我们例子的实际效果就是打印这个自身是无序的字典的键和值，以排好序的键的顺序输出。

for循环以及与其作用相近的while循环，是在脚本中编写重复性任务语句的主要方法。

www.TopSage.com

事实上，尽管这样，for循环就像它的亲戚列表解析（我们刚见到的）一样是一个序列操作。它可以使用在任意一个序列对象，并且就像列表解析一样，甚至可以用在一些不是序列的对象中。例如，for循环可以步进循环字符串中的字符，打印每个字符的大写：

```
>>> for c in 'spam':
        print c.upper()

S
P
A
M
```

我们将会在本书稍后部分对循环语句进行讨论。

迭代和优化

如果for循环看起来就像之前介绍的列表解析表达式一样，那也没错。它们都是真正的通用迭代工具。事实上，它们都能够工作于遵守迭代协议（最近Python介绍的一个概念，实际上表示在内存中物理存储的序列，或一个在迭代操作情况下每次产生一个元素的对象）的任意对象。这也就是前一节所使用的sorted调用直接可以工作于字典的原因。我们没有必要调用keys方法得到一个序列，因为字典就是一个迭代对象。

本书稍后将会详细介绍迭代协议。现在记住任意的列表解析表达式，就像下边计算一个数字列表平方的例子：

```
>>> squares = [x ** 2 for x in [1, 2, 3, 4, 5]]
>>> squares
[1, 4, 9, 16, 25]
```

能够编写成一个等效的for循环，通过在运行时手动增加列表来创建最终的列表。

```
>>> squares = []
>>> for x in [1, 2, 3, 4, 5]:          # This is what a list comp does
        squares.append(x ** 2)

>>> squares
[1, 4, 9, 16, 25]
```

尽管这样，列表解析通常运行地更快（也许快了两倍）：这是大数据集合在程序中重要的原因之一。在Python中性能测试是一个很难应付的任务，因为它在反复地优化，也许版本和版本之间差别很大。

Python中的一个主要的原则就是，首先为了简单和可读性去编写代码，在程序运行后，并证明了确实有必要考虑性能后，再考虑该问题。更多的情况是代码本身就已经足够

快了。如果确实需要提高代码的性能，那么Python提供了帮助你实现的工具，包括time以及timeit模块和profile模块。你将会在本书稍后部分以及Python的手册中学到更多内容。

不存在的键：if 测试

在我们继续学习之前关于字典还有另一个要点：尽管我们能够通过给新的键赋值来扩展字典，但是获取一个不存在的键值仍然是一个错误。

```
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> D['e'] = 99                          # Assigning new keys grows dictionaries
>>> D
{'a': 1, 'c': 3, 'b': 2, 'e': 99}

>>> D['f']                               # Referencing one is an error
...error text omitted...
KeyError: 'f'
```

这就是我们所想要的：获取一个并不存在的东西往往是一个程序错误。但是，在一些通用程序中，我们编写程序时并不是总知道当前存在什么键。在这种情况下，我们如何处理并避免错误发生呢？一个技巧就是首先进行测试。字典的has_key方法允许我们查询一个键的存在性，并可以通过使用Python的if语句对结果进行分支处理：

```
>>> D.has_key('f')
False

>>> if not D.has_key('f'):
        print 'missing'

missing
```

本书稍后将对if语句及语句的通用语法进行更多的讲解，这里所使用的形式很直接：它包含关键字if，紧跟着一个其结果为真或假的表达式，如果测试的结果是真的话将运行一些代码。作为其完整的形式，在默认情况下，if语句有else分句，以及一个或多个elif(else if)分句进行其他的测试。它是Python的主要的选择工具，并且是在脚本中编写逻辑的方法。

这里有其他的方法来创建字典并避免获取不存在的字典键（包括get方法、成员表达式、以及try语句，一个捕获并从异常中恢复的工具，我们将在第10章首次学习），我们将会把这里省略了的细节留给后边的章节。现在，让我们开始来学习元组吧。

www.TopSage.com

大家网 TopSage.com

元组

元组对象（发音为“toople” 或“tuhple”，这取决于问的人是谁）基本上就像一个不可以改变的列表。就像列表一样，元组是序列，但是它具有不可变性，和字符串类似：

```
>>> T = (1, 2, 3, 4)                    # A 4-item tuple
>>> len(T)                              # Length
4

>> T + (5, 6)                           # Concatenation
(1, 2, 3, 4, 5, 6)

>>> T[0]                                # Indexing, slicing, and more
1
```

元组的真正的不同之处就在于一旦创建后就不能再改变。也就是说，元组是不可变的序列：

```
>>> T[0] = 2                            # Tuples are immutable
...error text omitted...
TypeError: 'tuple' object does not support item assignment
```

为什么要用元组

那么，为什么我们要用一种类似列表这样的类型，尽管它支持的操作很少，坦白地说，元组在实际中通常并不像列表这样常用，但是它的关键是不可变性。如果在程序中以列表的形式传递一个对象的集合，它能够在任何地方改变；如果使用元组的话，则不能。也就是说，元组提供了一种完整性的约束，这对于比我们这里所编写的更大型的程序来说是方便的。我们将会在本书稍后部分介绍更多元组的内容。让我们直接学习最后一个主要的核心类型 —— 文件。

文件

文件对象是Python代码对电脑上外部文件的主要接口。虽然文件是核心类型，但是它有些特殊：没有特定的常量语法创建文件。你调用内置的open函数创建一个文件对象，以字符串的形式传递给它一个外部的文件名以及一个处理模式的字符串。例如，创建一个输出文件，可以传递其文件名以及'w'处理模式字符串去写数据：

```
>>> f = open('data.txt', 'w')           # Make a new file in output mode
>>> f.write('Hello\n')                  # Write strings of bytes to it
>>> f.write('world\n')
>>> f.close()                           # Close to flush output buffers to disk
```


www.TopSage.com

这样就在当前文件夹下创建了一个文件，并向它写入文本（文件名可以是完整的路径，如果需要读取电脑上其他位置的文件）。为了读出刚才所写的内容，重新以‘r’处理模式打开文件，读取输入（如果在调用时忽略模式的话，这将是默认的）。之后将文件的内容读至一个名为bytes的字符串，并显示它。对于脚本，一个文件的内容总是字节字符串，无论文件包含的数据是什么类型：

```
>>> f = open('data.txt')                    # 'r' is the default processing mode
>>> bytes = f.read()                        # Read entire file into a string
>>> bytes
'Hello\nworld\n'

>>> print bytes                             # Print interprets control characters
Hello
world

>>> bytes.split()                           # File content is always a string
['Hello', 'world']
```

这里对其他的文件对象方法支持的特性不进行讨论。例如，文件对象提供了多种读和写的方法（read可以接受一个字节大小的选项，readline每次读一行等），以及其他的工具（搜索移动到一个新的文件位置）。我们在本书中会看到全部的文件方法，但是如果你现在想要预览一下的话，对file这个词（文件数据类型的变量名）运行dir调用，并可对任何输出的变量名使用help：

```
>>> dir(file)
['__class__', '__delattr__', '__doc__', '__enter__', '__exit__',
'__getattribute__', '__hash__', '__init__', '__iter__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__str__',
'close', 'closed', 'encoding', 'fileno', 'flush', 'isatty', 'mode',
'name', 'newlines', 'next', 'read', 'readinto', 'readline', 'readlines',
'seek', 'softspace', 'tell', 'truncate', 'write', 'writelines', 'xreadlines']

>>> help(file.seek)
...try it and see...
```

其他文件类工具

open函数能够实现在Python中编写的绝大多数文件处理。尽管这样，对于更高级的任务，Python还有额外的文件类工具：pipes、fifos、sockets、keyed-access files、对象持久、基于描述符的文件、关系数据库和面向对象数据库接口等。例如，文件描述符（descriptor file）支持文件锁定和其他的底层工具，而sockets提供网络和进程间通信的接口。本书中我们并不全部介绍这些话题，但是在开始使用Python编程时，一定会发现这些都很有用的。

www.TopSage.com

其他核心类型

到目前为止除了我们看到的核心类型，还有其他的或许能够称得上核心类型的类型，这取决于我们定义的分类有多大。例如，集合是最近增加到这门语言中的类型。集合是通过调用内置set函数而创建的对象的容器，它支持一般的数学集合操作：

```
>>> X = set('spam')
>>> Y = set(['h', 'a', 'm'])                    # Make 2 sets out of sequences
>>> X, Y
(set(['a', 'p', 's', 'm']), set(['a', 'h', 'm']))

>>> X & Y                                       # Intersection
set(['a', 'm'])

>>> X | Y                                       # Union
set(['a', 'p', 's', 'h', 'm'])

>>> X - Y                                       # Difference
set(['p', 's'])
```

此外，Python最近添加的十进制数（固定精度浮点数）和布尔值（预定义的true和false对象实际上是定制后以逻辑结果显示的整数1和0），以及长期以来一直支持的特殊的占位符对象None：

```
>>> import decimal                              # Decimals
>>> d = decimal.Decimal('3.141')
>>> d + 1
Decimal("4.141")

>>> 1 > 2, 1 < 2                                # Booleans
(False, True)
>>> bool('spam')
True

>>> X = None                                    # None placeholder
>>> print X
None
>>> L = [None] * 100                            # Initialize a list of 100 Nones
>>> L
[None, None, None, None, None, None, None, None, None, None, None, None, None,
...a list of 100 Nones...]

>>> type(L)                                     # Types
<type 'list'>
>>> type(type(L))                               # Even types are objects
<type 'type'>
```

如何破坏代码的灵活性

本书稍后将对所有的这些对象进行介绍，但是还有一点值得注意。对象类型可以让代码检验它所使用的对象的类型。事实上，在Python脚本中有至少三种方法可以这样做：

```
>>> if type(L) == type([]):                    # Type testing, if you must...
        print 'yes'

yes
>>> if type(L) == list:                        # Using the type name
        print 'yes'

yes
>>> if isinstance(L, list):                    # Object-oriented tests
        print 'yes'

yes
```

现在本书已经介绍了所有的对象检验的方法，尽管这样，我们不得不说，就像在本书后边看到的那样，在Python程序中这样做基本上都是错误的（也是一个前C程序员刚开始使用Python时的一个标志）。在代码中检验了特定的类型，实际上破坏了它的灵活性，即限制它只能使用一种类型工作。没有这样的检测，代码也许能够使用整个范围的类型工作。

这与前边我们讲到的多态的思想有些关联，它是由Python没有类型声明而发展出来的。就像你将会学到的那样，在Python中，我们编写对象接口（所支持的操作）而不是类型。不关注于特定的类型意味着代码会自动地适应于它们中的很多类型：任何具有兼容接口的对象均能够工作，而不管它是什么对象类型。尽管支持类型检测（即使在一些极少数的情况下，是必要的）你将会看到它并不是一个“Python式”的思维方法。事实上，你将会发现多态也是使用Python的一个关键思想。

用户定义的类

我们将深入学习Python中的面向对象编程（这门语言一个可选的但很强大的特性，它可以通过支持程序定制而节省开发时间）。尽管这样，用抽象的术语来说，类定义了新的对象类型，扩展了核心类型，所以本处对这些内容做一个概览。也就是说，假如你希望有一个对象类型对职员进行建模。尽管Python里没有这样特定的核心类型，下边这个用户定义的类或许符合你的需求：

```
>>> class Worker:
        def __init__(self, name, pay):          # Initialize when created
            self.name = name                    # Self is the new object
            self.pay  = pay
```


www.TopSage.com

```
    def lastName(self):
        return self.name.split()[-1]          # Split string on blanks
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)           # Update pay in-place
```

这个类定义了一个新的对象的种类，有name和pay两个属性（又称状态信息），也有两个小的行为编写为函数的形式。就像函数那样去调用类，会生成我们新类型的一个实例，并且类的方法调用时，类的方法自动接获取被处理的实例（其中的self参数）：

```
>>> bob = Worker('Bob Smith', 50000)      # Make two instances
>>> sue = Worker('Sue Jones', 60000)      # Each has name and pay
>>> bob.lastName()                        # Call method: bob is self
'Smith'
>>> sue.lastName()                        # Sue is the self subject
'Jones'
>>> sue.giveRaise(.10)                    # Updates sue's pay
>>> sue.pay
66000.0
```

采用的“self”对象是我们把这叫做面向对象模型的原因，即一个类中的函数总有一个隐含的对象。一般来说，尽管这样，基于类的类型是建立在并使用了核心类型的。例如，这里的一个用户定义的Worker对象，是一个字符串和数字（分别为name和pay）的集合，附加了用来处理这两个内置对象的函数。

关于类更多的知识及其继承机制使Python支持了软件层次，使其可以通过扩展进行定制。我们通过编写新的类来扩展软件，而不是改变目前工作的类。你应该知道类是Python可选的一个特性，并且与用户编写的类相比，像列表和字典这样的更简单的内置类型往往是更好的工具。到这里已经远超出了介绍面向对象教程的范围了，然而为了获取更多细节，你可以阅读后面的章节。

剩余的内容

就像之前说过的那样，Python脚本中能够处理的所有的事情都是某种类型的对象，而我们的对象类型介绍是不完整的。尽管这样，即使在Python中的每样东西都是一个“对象”，只有我们目前所见到的那些对象类型才被认为是Python核心类型集合中的一部分。其他Python中的对象往往是由模块函数实现的，而不是语言语法。它们也更倾向于特定的应用领域的角色，例如，文本模式、数据库接口、网络连接等。

此外，记住我们学过的对象仅是对象而已，并不一定是面向对象。面向对象是一种往往要求有继承和Python类声明的概念，我们将会在本书稍后部分学到。Python的核心对象是你可能碰到的每一个Python脚本的重点，它们往往是更多的非核心类型的基础。

www.TopSage.com

本章小结

这就是我们完成的简明的数据类型之旅。本章介绍了Python核心对象类型，以及可以对它们进行的一些操作。我们学习了一些能够用于许多对象类型的一般操作（例如，索引和分片这样的序列操作）。在学习的过程中已经定义了一些关键的术语。例如，不可变性、序列和多态。

在这个过程中，我们见证了Python的核心对象类型，比像C这样的底层语言中的对应部分有更好的灵活性也更强大。例如，列表和字典省去了在底层语言中为了支持集合和搜索所进行的绝大部分工作。列表是其他对象的有序集合，而字典是通过键而不是位置进行索引的其他对象的集合。字典和列表都能够被嵌套，能够根据需要增大或减小，以及可以包含任意类型的对象。此外，它们的内存空间在不再使用后也是自动清理的。

本书在这里尽量跳过了细节以便提供一个快速的介绍，所以别指望这一章所有的内容都讲透彻了。在紧接着的几章中，我们将会更深入地学习，填补我们这里所忽略了的Python核心对象类型的各种细节，以便你能够完全的理解。我们将会在下一章深入了解Python数字。那么，首先让我们进行来另一个测验来复习吧。

www.TopSage.com

大家网
TopSage.com

本章习题

未来的几章将会探索这章所介绍的概念的更多细节，所以这里我们将仅介绍一些大致的概念：

1. 列举4个Python核心数据类型的名称。
2. 为什么我们把它们称作是“核心”数据类型？
3. “不可变性”代表了什么，哪三种Python的核心类型被认为是具有不可变性的？
4. “序列”是什么意思，哪三种Python的核心类型被认为是这个分类中的？
5. “映射”是什么意思，哪种Python的核心类型是映射？
6. 什么是“多态”，为什么我们要关心多态？

习题解答

1. 数字、字符串、列表、字典、元组和文件一般被认为是核心对象（数据）类型。集合、类型、None和布尔型有时也被定义在这样的分类中。还有多种数字类型（整数，长整型数字，浮点数以及十进制数）和两种字符串类型（一般字符串和Unicode）。
2. 它们被认作是“核心”类型是因为它们是Python语言自身的一部分，并且总是有效的；为了建立其他的对象，你通常必须调用被导入模块的函数。大多数核心类型都有特定的语法去生成其对象：例如“spam”，是一个创建字符串的表达式，而且决定了可以被应用的操作的集合。正是因为这一点，核心类型与Python的语法紧密地结合在一起。与之相比较，你必须调用内置的open函数去创建一个文件对象。
3. 一个具有“不可变性”的对象是一个在其创建以后不能够被改变的对象。Python中的数字、字符串和元组都属于这个分类。尽管你无法改变一个不可变的对象，但是你总是可以通过运行一个表达式创建一个新的对象。
4. 一个“序列”是一个对位置进行排序的对象的集合。字符串、列表和元组是Python中所有的序列。它们共同拥有一般的序列操作，例如，索引、合并以及分片，但又各自有自己的类型特定的方法调用。
5. 术语“映射”，表示将键与值相互关联映射的对象。Python的字典是其核心类型集中唯一的映射类型。映射没有从左至右的位置顺序；它们支持通过键获取数据，并包含了类型特定的方法调用。

www.TopSage.com

6. “多态”意味着一个操作符（就像+）的意义取决于被操作的对象。这将变成使用好Python的关键思想之一（或许可以去掉之一吧）：不要把代码限制在特定的类型上，使代码自动的适用于多种类型。

www.TopSage.com

第5章

数字

本章我们将开始更深入的Python语言之旅。在Python中，数据采用了对象的形式 —— 无论是Python所提供的内置对象，还是使用Python的工具和像C这样的其他语言所创建的对象。事实上，编写的所有Python程序的基础就是对象。因为对象是Python编程中的最基本的概念，也是本书第一个关注的焦点。

在上一章，我们对Python的核心对象进行了概览。尽管在该章已经介绍了最核心的术语，但是由于篇幅有限，没有涉及过多的细节。本章将开始对数据类型概念进行更详尽的学习。本章将会探索与数字相关的类型，例如，集合和布尔型。让我们开始探索第一个数据类型的分类：Python数字。

Python的数字类型

Python的数字类型是相当典型的，如果你有其他语言的编程经验的话，也许会对比很熟悉。它能够保持跟踪你的银行余额、到火星的距离、访问网站的人数以及任何其他的数字特性。

在Python中，数字并不是一个真正的对象类型，而是一组类似类型的分类。Python不仅支持通常的数字类型（整数和浮点数），而且能够通过常量去直接创建数字并处理它的表达式。此外，Python提供了对高级数学编程的支持，包括复数类型、无穷精度整数类型、固定精度十进制数、集合和布尔型以及其他各种数字工具的库。下面的几节将会对Python数字进行概述。

数字常量

在基本类型中，Python提供了常用的数字类型：支持整数、浮点数（有小数部分）以及与之相关的语法和操作。就像C语言一样，Python允许使用十六进制数和八进制数常

www.TopSage.com

量。但是，和C语言不同的是，Python额外提供了复数类型以及有着无穷精度的长整数类型（只要内存空间允许，它可以增长成任意位数的数字）。表5-1展示了Python数字类型在程序中是如何显示的（换句话说，作为常量）。

TopSage.com

表 5-1：

数字	常量
1234, -24, 0	一般证书（C语言长整型）
99999999999999999L	长整型数（无限大小）
1.23, 3.14e-10, 4E210, 4.0e+210	浮点数（C语言双精度浮点数）
0177, 0x9ff, 0XFF	整数的八进制和十六进制数常量
3+4j, 3.0+4.0j, 3J	复数常量

一般来说，Python的数字类型是直接的，但是有些编程的概念需要在这里强调一下。

整数和浮点数常量

整数以十进制数字的字符串写法出现。浮点数带一个小数点，也可以加上一个科学计数标志e或者E。如果编写一个带有小数点或幂的数字，Python会将它变成一个浮点数对象，并且当这个对象用在表达式中时，将启用浮点数（而不是整数）的运算法则。Python中表示浮点数的方法和C语言中的是一样的。

数字精度和长整型数

一般的Python整数（表5-1中的第一行）在内部是以C语言的“长整型”来实现的（也就是说，至少有32位），而Python的浮点数是以C语言的“双精度浮点型”来实现的。因此Python数字的精度和建立Python解释器的C语言编译器使用的长整型和双精度浮点型的精度是一样的（注1）。

长整型数常量

然而，如果整数常量以l或L结尾，那么它就变成了Python的长整型数（不要和C语言的长整型混淆起来），而且可以任意地增大。在Python2.2和之后的版本中，因为当一个整数的值超过32位时，它会自动变换为长整型数，不需要自己输入字母L。当有额外的精度需求时，Python会自动将其升级为长整型数。

十六进制数和八进制数常量

在Python中编写十六进制（base 16）和八进制（base 8）整数的法则与C语言是一样的。八进制数常量以数字0开头，后面接数字0-7构成的字符串。十六进制数常量

注1：这就是Cpython的标准实现。而在以Java作为基础的Jpython的实现当中，Python的类型实际上就是Java的类。

www.TopSage.com

以0x或0X开头，后面接十六进制的数字0~9和A~F。在十六进制数常量中，十六进制的数字编写成大写或小写都可以。八进制数和十六进制数常量都会产生一个整数对象，它们仅仅是特定值的不同语法表示而已。

复数

Python的复数常量写成实部+虚部的写法，这里虚部是以j或J结尾。其中，实部从技术上讲可有可无，所以可能会单独表示虚部。从内部看来，复数都是通过一对浮点数来表示的，但是对复数的所有的数字操作都会按照复数的运算法则进行。

内置数学工具和扩展

除了在表5-1中显示的内置数字常量之外，Python还提供了一系列处理数字对象的工具：

表达式操作符

+、*、>>、**等。

内置数学函数

pow、abs等。

公用模块

random、math等。

随着学习的深入，所有这些我们都会见到。

最后，如果需要做一些严谨的数值运算，一个名为NumPy的Python扩展提供了高级的数值编程工具。例如，矩阵数据类型、向量处理以及高端的计算库。像Lawrence Livermore 和 NASA这些专业的科学程序设计团体都使用带有NumPy的Python去实现他们之前用C++、FORTRAN或Matlab编写的各种任务。

因为它太高端了，所以在本书中我们不会对NumPy进行深入讲解。你可以通过 Vaults of Parnassus这个站点获得Python进行高级数值编程的更多支持，或者在网络上搜索相关信息。另外需要注意NumPy当前是一个可选的扩展，并不包含在Python中，必须独立安装后才能使用。

注意： 在Python 3.0 中，当前的整数和长整型数将会合并，那么将会只有一个整数类型int。它将支持任意的精度，就像现在的长整型数一样。绝大多数的程序员会觉得这跟以前没什么区别。更多细节请查看Python 3.0 版本的升级说明。

---

www.TopSage.com

Python表达式操作符

表达式是处理数字的最基本的工具。当一个数字（或其他对象）与操作符相结合时，Python执行时将计算得到一个值。在Python中，表达式是使用通常的数学符号和操作符号写出来的。例如，让两个数字X和Y相加，写成X+Y，这就会告诉Python，对名为X和Y的变量值应用+的操作。这个表达式的结果就是X与Y的和，也就是另一个数字对象。

表5-2列举了Python所有的操作符表达式。有许多是一看就懂的。例如，支持一般的数学操作符（+、-、*、/等）。如果你曾经使用过C语言的话，其中一些操作符应该很眼熟：%是计算余数操作符；<<执行左移位，&计算位与的结果等。其他的则更Python化一些，并且不全都具备数值特征。例如，is操作符测试对象身份（也就是内存地址，严格意义上的相等），lambda创建匿名函数。我们稍后会介绍更多相关的内容。

表 5-2：Python表达式操作符及程序

操作符	描述
yield x	生成器函数发送协议（2.5版新增）
lambda args: expression	生成匿名函数
x if y else z	三元选择表达式（2.5版新增）
x or y	逻辑或（只有x为假，才会计算y）
x and y	逻辑与（只有x为真，才会计算y）
not x	逻辑非
x < y, x <= y, x > y, x >= y, x == y, x <> y, x != y, x is y, x is not y, x in y, x not in y	比较操作，值相等操作[a]，对象身份测试，序列成员测试
x | y	位或
x ^ y	Bitwise 位异或
x & y	位与
x << y, x >> y	x左移或右移y位
-x + y, x - y	加法/合并，减法
x * y, x % y, x / y, x // y	乘法/重复，余数/格式化，除法[b]
-x, +x, ~x, x ** y	一元减法，识别，按位求补，幂运算
x[i], x[i:j], x.attr, x(...)	索引，分片，点号取属性运算，函数调用
(...), [...], {...}, `...`	元组，列表[c]，字典，字符串转换[d]

a. 在Python 2.5版中，值不相等可以写成X != Y或X <> Y。在Python 3.0之中，后者会被移除，因其为它是多余的。值不相等测试使用X != Y就行了。

www.TopSage.com

b. Floor除法（X // Y）是在2.2版中引进的，会把余数小数部份去掉。“除法：传统除法，Floor除法，以及真除法”一节会进一步说明。

c. 从Python 2.0版起，列表语法（[...]）可以表示列表常量或列表解析表达式。后者是执行隐性循环，把表达式的结果收集到新的列表中。

d. 把对象转换成打印字符串也能以更具可读性的内建函数str和repr来完成，本章稍后“数字显示的格式“一节会进行说明。反引号表达式`X`因为难以理解，所以，计划在Python 3.0时予以移除。改为使用repr(X)。

混合操作所遵循的操作符优先级

就像大多数语言一样，在Python中，将表5.2中的操作符表达式像字符串一样结合在一起就能编写出很多较复杂的表达式。例如，两个乘法之和可以写成变量和操作符的结合：

```
A * B + C * D
```

那么，如何让Python知道先进行哪个操作呢？这个问题的答案就在于操作符的优先级。当编写含有一个操作符以上的表达式时，Python将按照所谓的优先级法则对其进行分组，这个分组决定了表达式中各部分的计算顺序。在表5.2中，表的操作符中越靠后的优先级越高，因此在混合表达式中要更加小心一些。

例如，计算表达式X + Y * Z，Python首先计算乘法（Y * Z），然后将其结果与X相加，因为“*”比“+”优先级高。类似地，在这一节的最初那个例子中，两个乘法（A * B和 C * D）将会在它们的结果相加之前进行。

括号分组的子表达式

如果用括号将表达式各部分进行分组的话，就可以完全忘掉优先级的事情了。当使用括号划分子表达式的时候，就会超越Python的优先级规则。Python总会先行计算括号中的表达式，然后再将结果用在整个表达式中。

例如，表达式X + Y * Z写成下边两个表达式中的任意一个以此来强制Python按照你想要的顺序去计算表达式：

```
(X + Y) * Z
X + (Y * Z)
```

在第一种情况下，“+”首先作用于X和Y，因为这个子表达式是包含在括号中的。在第二种情况下，首先使用“*”（即使这里没括号也会这样）。一般来说，在一个大型表达

www.TopSage.com

式中增加括号是个很好的方法，它不仅仅强制按照你想要的顺序进行计算，同时也增加了程序可读性。

混合类型自动升级

除了在表达式中混合操作符以外，也能够混合数字的类型。例如，可以把一个整数与一个浮点数相加：

```
40 + 3.14
```

但是这将引出另一个问题：它们的结果是什么类型？是整数还是浮点数？答案很简单，特别是如果你有使用其他语言经验的话：在混合类型的表达式中，Python首先将被操作的对象转换成其中最复杂的操作对象的类型，然后再对相同类型的操作对象进行数学运算。如果你使用过C语言，你会发现这个行为与C语言中的类型变换是很相似的。

Python是这样划分数字类型的复杂度的：整数比长整型数简单，长整型数要比浮点数简单，浮点数比复数简单。所以，当一个整数与浮点数混合时，就像前边的那个例子，整数首先会升级转为浮点数的值，之后通过浮点数的运算法则得到浮点数的结果。类似地，任何混合类型的表达式，其中一个操作对象是更为复杂的数字，则会导致其他的操作对象被升级为一个复杂的数字，使得表达式获得一个复杂的结果。就像本节后面内容介绍的那样，在2.2版中，只要一个整数对普通值来说过大的话，Python也会自动将一般的整数变换为长整型数。

可以通过手动调用内置函数来强制转换类型：

```
>>> int(3.1415)
3
>>> float(3)
3.0
>>> long(4)
4L
```

然而，通常是没有必要这样做的。因为Python在表达式中自动升级为更复杂的类型，其结果往往就是你所想要的。

再者，要记住所有这些混合类型变换仅仅在一个操作符或者比较操作周围将数字类型（例如，一个整数和一个浮点数）进行混合的时候才是有效的。一般来说，Python不会在其他的类型之间进行转换。例如，一个字符串和一个整数相加，会产生错误，除非你手动转换其中某个的类型。请注意第7章学习有关字符串的内容时将要看到的一个例子。

www.TopSage.com

预习：运算符重载

尽管我们目前把注意力集中在内置的数字上，要留心所有的Python操作符可以通过Python的类或C扩展类型被重载（即实现），让它也能工作于你所创建的对象中。例如，用类编写的对象代码也许可以使用+表达式做加法，以及使用[i]表达式进行索引等。

再者，Python自身自动重载了某些操作符，能够根据所处理的内置对象的类型而执行不同的操作。例如，“+”操作符应用于数字时是在做加法，而用于字符串或列表这样的序列对象时是在做合并运算。实际上，“+”应用在定义的类的对象上可以进行任何运算。

就像我们在前一章介绍过，这种特性通常称作多态。这个术语指操作的意义取决于所操作的对象的类型。第15章将会复习这个概念，因为在那时这已经成了更显著的特性了。

在实际应用中的数字

也许最好的理解数字对象和表达式的方法就是看看实际中它们的应用。所以让我们打开交互命令行，实验一些基本但是很能说明问题的操作吧（如果开始交互会话需要帮助，请在第3章找提示）。

变量和基本的表达式

首先，让我们练习一下基本的数学运算吧。在下面的交互中，首先把两个变量（a和b）赋值为整数，在更大的表达式中我们会使用它们。变量简单的就是变量名而已（由你或Python创建的），可以用来记录程序中信息。我们将会在下一章介绍更多内容，在Python中：

- 变量在它第一次赋值时创建。
- 变量在表达式中使用将被替换为它们的值。
- 变量在表达式中使用以前必须已赋值。
- 变量像对象一样不需要在一开始进行声明。

换句话说，这些赋值会让变量a和b自动生成：

```
% python
>>> a = 3                    # Name created
>>> b = 4
```

这里本书使用了注释。回忆一下在Python代码中，在#标记后直到本行末尾的文本会认作

是一个注释并忽略。注释是为代码编写读者可读文档的方法。因为在交互模式下编写的代码是暂时的，一般是不会在这种情形下编写注释的，但是本书中的一些例子中添加了注释，是为了有助于解释代码（注2）。本书的下一部分，我们将会看到一个类似的特性：文档字符串，对象上附加的注释的文本。

现在，让我们在表达式中使用新的整数对象。目前，a和b的值分别是3和4。一旦用在一个表达式中，就像这里的变量，都会被它们的值替换，当在交互模式下工作时，表达式的结果将马上显示出来：

```
>>> a + 1, a - 1              # Addition (3 + 1), subtraction (3 - 1)
(4, 2)
>>> b * 3, b / 2              # Multiplication (4 * 3), division (4 / 2)
(12, 2)
>>> a % 2, b ** 2             # Modulus (remainder), power
(1, 16)
>>> 2 + 4.0, 2.0 ** b         # Mixed-type conversions
(6.0, 16.0)
```

从技术上讲，这里的回显所得到的结果是有两个值的元组，因为输入在提示符的那行包含了两个被逗号分开的表达式。这也就是显示的结果包含在括号里的原因（稍后介绍更多关于元组的内容）。

注意表达式正常工作了的，因为变量a和b已经被赋值了。如果使用一个从未被赋值的不同的变量，Python将会报告有错误而不是赋给默认的值：

```
>>> c * 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'c' is not defined
```

在Python中，变量并不需要预声明，但是在使用之前，至少要被赋一次值。实际上，这意味着在对其进行加法运算时要计数器初始化为0，在列表后添加元素前，要首先初始化列表为一个空列表。

这里有两个稍长一些的表达式，阐明了操作符分组以及类型转换：

```
>>> b / 2 + a                 # Same as ((4 / 2) + 3)
5
>>> print b / (2.0 + a)       # Same as (4 / (2.0 + 3))
0.8
```

在第一个表达式中，没有括号，所以Python自动根据运算符的优先级法则将元件分组。

注2：　如果是跟着做的话，是不需要输入每个#后到行末的注释文本的；注释将会被Python忽略掉，并且在运行时注释部分是不需要的。

www.TopSage.com

因为在表5-2中“/”比“+”位置靠后，它的优先级更高，所以首先进行“/”运算。结果就像代码右边的注释中加了括号表达式运算的结果一样。并且，注意到第一个表达式中所有的数字都是整数。因为这一点，Python进行了整数的除法和加法。

在第二个表达式中，括号用在“+”的周围，强制使Python首先计算“+”（也就是说，先于“/”）。并且，通过增加小数点让其中的一个操作对象为浮点数2.0。因为是混合类型，Python在进行“+”之前首先将整数变换为浮点数的值（3.0）。它也会将b变换为一个浮点数的值（4.0），并进行浮点数的除法，（4.0 / 5.0）得到一个浮点数，结果为0.8。如果这个表达式中所有的数字都是整数，它将会变成整数除法（4/5），其结果将会截断后的整数0（至少在Python2.5中是这样，请阅读后面关于除法的讨论）。

数字显示的格式

注意我们在前边的最后一个例子使用的是打印语句。如果不使用打印语句，你首次看到一些结果可能会有些奇怪：

```
>>> b / (2.0 + a)          # Auto echo output: more digits
0.80000000000000004

>>> print b / (2.0 + a)    # Print rounds off digits
0.8
```

在这个奇怪结果背后的真正原因是浮点数的硬件限制，以及它无法精确地表现一些值。因为计算机架构不是本书能够涵盖的，那么，我们将简要解释成用第一个输出的所有数字都在计算机的浮点数硬件中，仅仅是你不习惯看它们而已。本书已经用过这个例子去说明在输出格式中存在的差异。交互提示模式下结果的自动回显会比打印语句显示更多的数字位数。如果你不想看到所有的位数，考虑一下print。

注意，尽管这样，并不是所有的值都有这么多的数字位数需要显示：

```
>>> 1 / 2.0
0.5
```

并且除了打印和自动回显之外，还有很多种方法显示电脑中的数字的位数：

```
>>> num = 1 / 3.0
>>> num                    # Echoes
0.33333333333333331
>>> print num              # Print rounds
0.333333333333

>>> "%e" % num             # String formatting
'3.333333e-001'
>>> "%2.2f" % num          # String formatting
'0.33'
```


www.TopSage.com

这些方法中的最后两个使用了字符串格式，灵活的进行格式化的表达式，我们将会在以后的关于字符串的章节介绍它（第7章）。

str和repr显示格式

从技术上来说，默认的交互模式回显和打印的区别就相当于内置repr和str函数的区别：

```
>>> repr(num)                  # Used by echoes: as-code form
'0.33333333333333331'
>>> str(num)                   # Used by print: user-friendly form
'0.333333333333'
```

这两个函数都会把任意对象变换成它们的字符串表达：repr（也就是默认的交互模式回显）产生的结果看起来就像是它们是代码。str（也就是打印语句）变换得到一种对用户更加友好的格式。这个概念将会为我们学习字符串做好铺垫，并且本书稍后会介绍关于这两个内置函数的更多内容。

除法：传统除法、Floor除法和真除法

现在，已经介绍了除法的工作方式，在以后的Python版本中会有些细微的改变（目前，Python 3.0打算在2.5版发行后采用）。在Python 2.5中，就像我们所描述的那样工作，但是实际上有两种不同的除法操作（其中一种将会改变）：

`X / Y`

传统除法。在Python 2.5或之前的版本中，这个操作对于整数会省去小数部分，对于浮点数会保持小数部分，就像这里描述的那样。这个操作在Python 3.0版本中将会变成真除法（无论任何类型都会保持小数部分）。

`X // Y`

Floor除法。在Python 2.2中新增的操作，这个操作不考虑操作对象的类型，总会省略掉结果的小数部分，剩下最小的能整除的整数部分。

新增Floor除法，是为了解决当前传统除法模型的结果取决于其操作对象的类型这一问题的，那样会让像Python这样的动态类型语言变得难以预料。

由于可能出现向前兼容的问题，Python中的除法如今处于不稳定的状态。为了统一，在2.5 版本中，“/除法”作为默认的除法进行工作，而“// Floor”除法能够用在省略掉结果中的小数部分，直到剩下最小的能整除的整数部分而不管它们的类型。

www.TopSage.com

大家网
TopSage.com

```
>>> (5 / 2), (5 / 2.0), (5 / -2.0), (5 / -2)
(2, 2.5, -2.5, -3)

>>> (5 // 2), (5 // 2.0), (5 // -2.0), (5 // -2)
(2, 2.0, -3.0, -3)

>>> (9 / 3), (9.0 / 3), (9 // 3), (9 // 3.0)
(3, 3.0, 3, 3.0)
```

在以后的Python版本中，“/除法”默认为真除法，总是保留小数部分，即使是应用在整数上。例如，1 / 2得到0.5，而不是0；而1//2仍然会是0。

在这个变换完全采用之前，可以通过特定的导入形式来使用这种未来将会采用的“/”操作：from __future__ import division。这会让“/”操作变成真除法（保留小数部分），但仍会让“//”保持原样。这就是“/”最终会变成的形式：

```
>>> from __future__ import division

>>> (5 / 2), (5 / 2.0), (5 / -2.0), (5 / -2)
(2.5, 2.5, -2.5, -2.5)

>>> (5 // 2), (5 // 2.0), (5 // -2.0), (5 // -2)
(2, 2.0, -3.0, -3)

>>> (9 / 3), (9.0 / 3), (9 // 3), (9 // 3.0)
(3.0, 3.0, 3, 3.0)
```

参照第13章中的while循环的求质数例子，以及在第4部分后面对应的练习题，它们将会显示出“/”变换后的影响。一般来说，任何依靠“/”来省略小数部分而成为整数的代码将会受到影响（使用新的“//”来替代）。就像本书介绍，这个变换将会在Python 3.0中采用，但是确保在你所使用的版本中实验过了，以便能够看到这样应用后的表现。别对这里使用到的from命令的细节太在意，它将会在第21章进一步讨论。

位操作

除了一般的数学运算（加法、减法等），Python也支持C语言中的大多数数学表达式。例如，还可以实现位移及布尔操作：

```
>>> x = 1        # 0001
>>> x << 2       # Shift left 2 bits: 0100
4
>>> x | 2        # bitwise OR: 0011
3
>>> x & 1        # bitwise AND: 0001
1
```

在第一个表达式中，二进制数1（逢2进位，0001）左移了两位，成为二进制数4（0100）。最后的两个操作一个是二进制或（0001|0010 = 0011），一个是二进制与（0001&0001 = 0001）。这样的按位进行掩码的运算，使我们可以对一个整数进行多个标志位和值进行编码。

我们在这里不会涉及更多的关于“位运算”的细节。如果需要的话，它是支持的，并且如果Python代码必须与由C程序生成的网络包或封装了的二进制数打交道的话，它是很实用的。尽管这样，注意位操作在Python这样的高级语言中并不像在C这样的底层语言中那么重要。作为一个简明的法则，如果你需要在Python中对位进行翻转，你应该想想现在使用的是哪一门语言。一般来说，在Python中有比位字符串更好的编码信息的方法。

长整型数

现在让我们看一些更少见的数据类型 —— 长整型数。当一个整数常量以字母L（或小写l）结尾，Python就会创建一个长整型数。在Python中，一个长整型数可以任意大。也就是说，你的内存空间有多大它就可以有多少位数字。

```
>>> 99999999999999999999999999999999999999L + 1
100000000000000000000000000000000000000L
```

数字字符串最后的L告诉Python去创建一个有无限精度的长整型数对象。实际上，在Python 2.2 中，字母L是可选的：如果对于一个一般的整数来说已经太大了的话（从技术上来讲，但它溢出一个正常的整数精度往往是32位），Python会自动将一个一般的整数升级为长整型数。也就是说，不需要自己编写L，如果程序需要的话，Python将会提供额外的精度。

```
>>> 9999999999999999999999999999999999999 + 1
100000000000000000000000000000000000000L
```

长整型数是一个方便的内置工具。例如，你可以使用它以便士为单位去直接计算国债（如果你有这种倾向，并且电脑有足够多的内存）。这也是它们能够用来计算第3章中2的如此高次幂的原因：

```
>>> 2L ** 200
1606938044258990275541962092341162602522202993782792835301376L
>>>
>>> 2 ** 200
1606938044258990275541962092341162602522202993782792835301376L
```

Python为了支持扩展的精度，需要做额外的工作，在实际应用中，长整型数的数学运算

www.TopSage.com

通常要比正常的整数运算（它是直接与硬件相关的）更慢。尽管这样，如果你确实需要精度，那么它是为你使用的是内置类型这个事实要比其性能的损失更重要一些吧。

复数

在Python中，复数是个不同的核心对象类型。如果你知道它是什么，你就会知道它的重要性；如果不知道的话，那么请把这一部分作为可选的阅读材料。复数表示为两个浮点数（实部和虚部）并接在虚部增加了j或J的后缀。我们能够把非零实部的复数写成由+连接起来的两部分。例如，一个复数的实部为2并且虚部为-3可以写成2 + -3j。这里是一些复数运算的例子。

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

复数允许我们分解出它的实部和虚部作为属性，并支持所有一般的数学表达式，以及可以通过标准的cmath模块（复数版的标准数学模块）中的工具进行处理。复数通常是在面向工程的程序中扮演了重要的角色。它是高级的工具，查看Python语言的参考手册来获取更多的信息。

十六进制和八进制记数

正如本章之前提到的那样，Python整数能够以十六进制（逢16进位）和八进制（逢8进位）的记数法作为一般的逢10进位的十进制编码的补充。

- 八进制常量以0开头，并紧跟着八进制数字0-7的字符串，每一个八进制数字都可由3个二进制位来表示。
- 十六进制常量以0x或0X开头，并跟着由十六进制数字0-9和大写或小写的A-F构成的字符串，每个十六进制数字能够由4个二进制位来表示。

记住这只是定义一个整数对象值的另一种语法而已。例如，下面的八进制和十六进制常量生成了特定的值的一般整数：

```
>>> 01, 010, 0100                 # Octal literals
(1, 8, 64)
>>> 0x01, 0x10, 0xFF              # Hex literals
(1, 16, 255)
```


www.TopSage.com

这里，八进制的0100就是十进制的64，而十六进制的0xFF就是十进制的255。Python默认会以十进制（逢10进位）进行打印，但是也提供了内置的函数允许将一个整数变为其八进制或十六进制的数字字符串。

TopSage.com

```
>>> oct(64), hex(64), hex(255)
('0100', '0x40', '0xff')
```

oct函数会将十进制数转换为八进制数，hex函数会将十进制转换为十六进制数。另一种方式，内置的int函数会将一个数字的字符串变换为一个整数，并可以通过定义的第二个参数来确定变换后的数字的进制：

```
>>> int('0100'), int('0100', 8), int('0x40', 16)
(100, 64, 64)
```

我们将在本书稍后介绍的eval函数，将会把字符串作为Python代码。因此，它也具有类似的效果（但往往运行得更慢：它实际上会作为程序的一个片段编译并运行这个字符串，并且它假设你能够信任运行的字符串的来源。一个耍小聪明的用户也许能够提交一个能够删除机器上文件的代码）：

```
>>> eval('100'), eval('0100'), eval('0x40')
(100, 64, 64)
```

最后，你能够将一个整数通过字符串格式表达式转换成八进制数和十六进制数的字符串：

```
>>> "%o %x %X" % (64, 64, 255)
'100 40 FF'
```

再次强调，字符串格式化将会在第7章介绍。

注意在Python中不要以使用0开头的数字字符串，除非你真正想要编写的是一个八进制数。Python将会把它当作八进制，将不会按照你原先设想的方式工作：010是十进制的8，而不是十进制的10（你是不是这样思考并不重要）。

其他的内置数学工具

除了核心对象类型以外，Python还支持用于数学处理的内置函数和内置模块。例如，内置函数int和round，分别省略浮点数的小数部分或省略保持小数点后若干位。这里有一些内置math模块（包含在C语言中math库中的绝大多数工具）的例子并有一些实际中的内置函数。

```
>>> import math
```


www.TopSage.com

```
>>> math.pi, math.e                          # Common constants
(3.1415926535897931, 2.7182818284590451)

>>> math.sin(2 * math.pi / 180)              # Sine, tangent, cosine
0.034899496702500969

>>> math.sqrt(144), math.sqrt(2)             # Square root
(12.0, 1.4142135623730951)

>>> abs(-42), 2**4, pow(2, 4)
(42, 16, 16)

>>> int(2.567), round(2.567), round(2.567, 2)   # Truncate, round
(2, 3.0, 2.5699999999999998)
```

就像之前描述的那样，如果我们使用**print**的话，这里最后的输出将会是**(2, 3.0, 2.57)**。

注意内置math这样的模块必须先导入，但是abs这样的内置函数不需要导入就可以直接使用。换句话说，模块是外部的组件，而内置函数位于一个隐性的命名空间内，Python自动搜索程序的变量名。这个命名空间对应于名为__builtin__的模块。在第4部分有更多关于变量名的解析。现在当你听到“模块”时，要想到“导入”。

使用标准库中的random模块时必须导入。这个模块提供了选出一个在0和1之间的任意浮点数、选择在两个数字之间的任意整数、在一个序列中任意挑选一项等功能：

```
>>> import random
>>> random.random()
0.49741978338014803
>>> random.random()
0.49354866439625611

>>> random.randint(1, 10)
5
>>> random.randint(1, 10)
4

>>> random.choice(['Life of Brian', 'Holy Grail', 'Meaning of Life'])
'Life of Brian'
>>> random.choice(['Life of Brian', 'Holy Grail', 'Meaning of Life'])
'Holy Grail'
```

Random模块很实用，在游戏中的发牌、在演示GUI中随机挑选图片、进行统计仿真等都需要用Random模块。参考Python库的手册来获取更多信息。

其他数字类型

本章已经介绍了Python的核心数字类型：整数、长整型数、浮点数和复数。这对于绝大

www.TopSage.com

多数程序员来说，需要进行的绝大多数数字处理都满足了。然而，Python还自带了一些更少见的数字类型，值得让我们在这里快速浏览一下。

小数数字

Python2.4介绍了一种新的核心数据类型：小数对象。比其他数据类型复杂一些，小数是通过一个导入的模块调用函数后创建的，而不是通过运行常量表达式创建的。从功能上来说，小数对象就像浮点数，有固定的小数位数，因此有固定的精度。例如，使用了小数对象，我们能够能使用一个只保留两位小数位精度的浮点数。此外，我们能够定义如何省略和截短额外的小数数字。尽管它对平常的浮点数类型来说带来了微小的性能损失，小数类型对表现固定精度的特性（例如，钱的总和）以及对实现更好的数字精度是个理想的工具。

浮点数对象的数学运算在精确方面有缺陷。例如，下面的计算应该得到零，但是结果却没有。结果接近零，但是却没有足够的位数去实现这样的精度：

```
>>> 0.1 + 0.1 + 0.1 - 0.3
5.5511151231257827e-017
```

打印结果将会产生一个用户友好的显示格式但并不能完全解决问题，因为与硬件相关的浮点数运算在精度方面有内在的缺陷。

```
>>> print 0.1 + 0.1 + 0.1 - 0.3
5.55111512313e-017
```

不过使用小数对象，结果能够改正：

```
>>> from decimal import Decimal
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') - Decimal('0.3')
Decimal("0.0")
```

正如这里显示的，我们能够通过调用在decimal模块中的Decimal的构造函数创建一个小数对象，并传入一个字符串，这个字符串有我们希望在结果中显示的小数位数。当不同精度的小数在表达式中混编时，Python自动升级为小数位数最多的：

```
>>> Decimal('0.1') + Decimal('0.10') + Decimal('0.10') - Decimal('0.30')
Decimal("0.00")
```

其他在decimal模块中的工具可以用于设置所有小数的精度等。例如，这个模块中的context对象可以设置特定的精度（小数位数），或者截短模式（舍去还是进位等）。

```
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal("0.14285714285714285714285714285714")
```


www.TopSage.com

大家网
TopSage.com

```
>>> decimal.getcontext().prec = 4
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal("0.1429")
```

因为小数类型在实际中的应用比较少见，要想获得更多的信息参见Python的标准库手册。

集合

Python 2.4引入了一种新的类型 —— 集合。因为它是其他对象的集合，集合照理说超出了本章的范围。但是因为它也支持数学的集合运算，这里简单介绍它的基本用法。创建一个集合对象，将一个序列或其他的迭代对象传递给内置set函数（在Python 2.4 版以前需要通过模块中的工具完成类似的功能，但是在Python 2.4版之后不需要再导入了）：

```
>>> x = set('abcde')
>>> y = set('bdxyz')
```

得到了一个集合对象，其中包含传递的对象的所有元素（注意集合并不包含位置顺序，序列却包含）：

```
>>> x
set(['a', 'c', 'b', 'e', 'd'])
```

集合通过表达式操作符支持一般的数学集合运算。不能在一般序列上应用这些操作，必须通过序列创建集合后才能使用这些工具。

```
>>> 'e' in x                                    # Set membership
True

>>> x - y                                       # Set difference
set(['a', 'c', 'e'])

>>> x | y                                       # Set union
set(['a', 'c', 'b', 'e', 'd', 'y', 'x', 'z'])

>>> x & y                                       # Set intersection
set(['b', 'd'])
```

当处理较大的数据集合时，这样的操作是很方便的。例如，两个集合的交集包含了两个集合共有的对象，并集则包含了两个集合所有的元素。下面是实际应用中关于集合操作的更现实的例子 —— 应用在一个假想公司的人员列表上：

```
>>> engineers = set(['bob', 'sue', 'ann', 'vic'])
>>> managers  = set(['tom', 'sue'])
>>>
>>> engineers & managers                        # Who is both engineer and manager?
```


www.TopSage.com

```
set(['sue'])
>>>
>>> engineers | managers                    # All people in either category
set(['vic', 'sue', 'tom', 'bob', 'ann'])
>>>
>>> engineers - managers                    # Engineers who are not managers
set(['vic', 'bob', 'ann'])
```

此外，集合对象提供的方法调用可以实现更为特殊的集合操作。尽管集合操作能够在Python中手动编写（并且在过去是常事），Python的内置集合还是提供了高效的算法和实现技巧，保证可以进行快速和标准的操作。

更多关于集合的细节，请看Python库的参考手册。

注意： 在Python 3.0 中，常量标记 {1, 2, 3}和当前调用set([1, 2, 3])具有相同的功能，并将会成为创建集合对象的另一种方法。这是一个以后的扩展，更多细节请参考3.0的发行报告。

布尔型

对于Python的布尔类型有一些争论，bool原本是一个数字，因为它有两个值True和False，不过是整数1和0以不同的形式打印后的定制版本而已。尽管这是大多数程序员应该知道的全部，我们还是要稍深入地探索这个类型。

Python 2.3 引入了一种新的明确的布尔型数据类型，称作bool，其值为True和False，并且其值True和False是预先定义的内置的变量名。在内部，新的变量名True和False是bool的实例，实际上仅仅是内置的整数类型int的子类（以面向对象的观点来看）。True和False的行为和整数1和0是一样的，除了它们有特定的逻辑打印形式:它们是作为关键字True和False打印的，而不是数字1和0（从技术上来讲，bool重新定义了str和repr的字符串格式）（注3）。

从Python 2.3开始，这个定制在交互提示模式的布尔表达式的输出就作为关键字True和False来打印，而不是曾经的1和0。此外，布尔型让真值更精确。例如，一个重定义的循环现在能够编写成while True:而不是while 1:。类似地，通过使用flag = False，可以更清楚地设置标志位。

注3： 有些人认为Python布尔型bool本质上是数字，因为两个值True和False只是整数1和0的定制版本而已（打印时显示不同的结果）。就多数实际用途而言，这就是对布尔值所需要知道的内容。

www.TopSage.com

还有对于其他实际的用途，你能够将True和False看作是预定义的设置为整数1和0的变量。大多数程序员都曾把True和False预先赋值为1和0，所以新的类型简单地让这个行为成为标准的技术。尽管它的实现能够导致奇怪的结果：因为True仅仅是定制了显示格式的整数1，在Python中True+3 得到了4!

我们将在第9章（去定义Python的真的概念）以及第12章（介绍像and和or这样的布尔操作符是如何工作的）重新介绍布尔型。

第三方扩展

除了Python自身的数字类型以外，你将会用到各种第三方开源扩展，支持更少见的数学工具。在网络上有提供像有理数、向量以及矩阵等类型。搜索一下获得更多的细节。

本章小结

本章进行了Python数字对象类型和能够应用于它们的操作的学习。在这个过程中，我们学习了标准的整数和浮点数类型，以及一些较少见和不常用的类型。例如，复数和小数类型。我们也学习了Python的表达式语法，类型转换、位操作以及各种在脚本中编写数字的常量形式。

接下来我们会学习下一个对象类型 —— 字符串的更多细节。尽管这样，在下一章，我们将会花一些时间去探索这里使用到的变量赋值机制的更多细节。这也许是Python中最基本的概念，所以在继续学习之前你要好好阅读下一章。下面我们照例进行章节测试。

www.TopSage.com

大家网 TopSage.com

本章习题

1. Python中表达式2 * (3 + 4)的值是多少?
2. Python中表达式2 * 3 + 4的值是多少?
3. Python中表达式2 + 3 * 4的值是多少?
4. 通过什么工具你可以找到一个数字的平方根以及它的平方?
5. 表达式 1 + 2.0 + 3的结果是什么类型?
6. 怎样能够截短或舍去浮点数的小数部分?
7. 怎样将一个整数转换为浮点数?
8. 如何将一个整数显示成八进制或十六进制的形式?
9. 如何将一个八进制或十六进制数的字符串转换成平常的整数?

习题解答

1. 结果的值将会是14, 即2*7的结果，因为括号强制让加法在乘法前进行运算。
2. 这里结果会是10，即6 + 4 的结果。Python的操作符优先级法则应用在没有括号存在的场合，根据表5-2，乘法的优先级要比加法的优先级高（先进行运算）。
3. 表达式将得到14，即2 + 12的结果，正如前一个问题一样是优先级的原因。
4. 为了得到平方根、*pi*以及正切等等函数，在导入math模块后即可使用。为了找到一个数字的平方根，import math后调用math.sqrt(N)。为了得到一个数字的平方，使用指数表达式X ** 2，或者内置函数pow(X, 2)。
5. 结果将是一个浮点数：整数将会变换升级成浮点数，这个表达式中最复杂的类型，然后使用浮点数的运算法则进行计算。
6. int(N)函数会省略小数部分，而round(N, 数字?)函数做四舍五入。
7. float(I)将整数转换为浮点数；在表达式中混合整数和浮点数也会实现转换。
8. 内置函数oct(I)和hex(I)会将整数以八进制数和十六进制数字符串的形式返回。%字符串表达式也会实现这样的目标。
9. int(S, base?)函数能够用来让一个八进制和十六进制数的字符串转换为正常的整数（传入以八和十六进制的数字）。eval(S)函数也能够用作这个目的，但是运行起来开销更大也会有可能导致安全问题。注意整数总是在计算机内存中以二进制保存的；这些只不过是显示的字符串格式的对话而已。

www.TopSage.com

第6章

动态类型简介

上一章通过学习Python数字类型，开始对Python的核心对象类型进行深入探索。我们将会在下一章继续我们的对象类型之旅，在我们继续学习之前，掌握Python编程中最基本的概念是很重要的。动态类型以及由它提供的多态性，这个概念无疑是Python语言的简洁性和灵活性的基础。

正像本书稍后介绍的那样，在Python中，我们并不会声明脚本中使用的对象的确切的类型。事实上，程序甚至可以不在意特定的类型；相反地，它们能够自然地适用于更广泛的场景下。因为动态类型是Python语言灵活性的根源，让我们先简要地看一下这个模块。

缺少类型声明语句的情况

如果你有学习静态编译类型语言C、C++或Java的背景，学到这里，你也许会有些困惑。到现在为止，我们使用变量时，都没有声明变量的存在和类型，并且变量还可以工作。例如，在交互会话模式或是程序文件中，当输入a = 3时，Python怎么知道那代表了一个整数呢？在这种情况下，Python怎么知道a是什么？

一旦你开始问这样的问题，就已经进入了Python动态类型模型的领域。在Python中，类型是在运行过程中自动决定的，而不是通过代码声明。这意味着没有必要事先声明变量（只要记住，这个概念实质上对变量、对象和它们之间的关系都适用，那么这个概念也就很容易理解并掌握了）。

变量、对象和引用

就像本书已使用过的很多例子一样，当在Python中运行赋值语句a = 3时，即使没有告诉Python将a作为一个变量来使用也能够工作，或者告诉Python，a应该作为一个整数类型对象。在Python语言中，这些都会以一种非常自然的方式完成，就像下边这样：

www.TopSage.com

变量创建

一个变量（也就是变量名），就像a，当代码第一次给它赋值时它就被创建了。之后的赋值将会改变已创建的变量名的值。从技术上来讲，Python在代码运行之前先检测变量名，可以当成是最初的赋值创建变量。

变量类型

变量永远不会有任何的和它关联的类型信息或约束。类型的概念是存在于对象中而不是变量名中。变量原本是通用的，它只是在一个特定的时间点，简单地引用了一个特定的对象而已。

变量使用

当变量出现在表达式中时，它会马上被当前引用的对象所代替，无论这个对象是什么类型。此外，所有的变量必须在其使用前明确地赋值，使用未赋值的变量会产生错误。

这种模式与传统语言相比有明显的不同。当入门时，如果清楚地将变量名和对象划分开来，动态类型是很容易理解的。例如，当我们这样说时：

```
>>> a = 3
```

至少从概念上来说，Python将会执行三个不同的步骤去完成这个请求。这些步骤反映了Python语言中所有赋值的操作：

1. 创建一个对象来代表值3。
2. 创建一个变量a，如果它还没有创建的话。
3. 将变量与新的对象3相连接。

实际的效果是如图6-1所示的一个在Python中的内部结构。如图6-1所示，变量和对象保存在内存中的不同部分，并通过连接相关联（这个连接在图6-1中显示为一个箭头）。变量总是连接到对象，并且绝不会连接到其他变量上，但是更大的对象可能连接到其他的对象（例如，一个列表对象能够连接到它所包含的对象）。

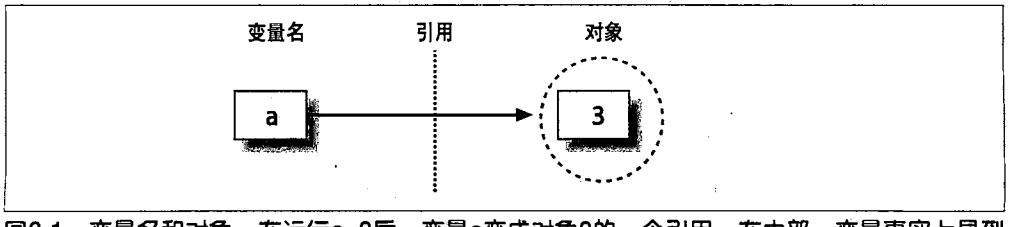


图6-1：变量名和对象，在运行a=3后。变量a变成对象3的一个引用。在内部，变量事实上是到对象内存空间(通过运行常量表达式3而创建)的一个指针

www.TopSage.com

在Python中从变量到对象的连接称作引用。也就是说，引用是一种关系，以内存中的指针的形式实现（注1）。一旦变量被使用（也就是说被引用），Python自动跟随这个变量到对象的连接。这实际上比术语所描述的要简单得多。以具体的术语来讲：

- **变量**是一个系统表的元素，拥有指向对象的连接的空间。
- **对象**是被分配的一块内存，有足够的空间去表现它们所代表的值。
- **引用**是自动形成的从变量到对象的指针。

至少从概念上讲，在脚本中，每一次通过运行一个表达式生成一个新的值，Python都创建了一个对象（换言之，一块内存）去表现这个值。从内部来看，作为一种优化，Python缓存了不变的对象并对其进行复用，例如，小的整数和字符串（每一个0都不是一块真正的新的内存块，稍后会介绍这种缓存行为）。但是，从逻辑的角度看，这工作起来就像每一个表达式结果的值都是一个不同的对象，而每一个对象都是不同的内存。

从技术上来讲，对象有更复杂的结构而不仅仅是有足够的空间表现它的值那么简单。每一个对象都有两个标准的头部信息：一个类型标志符去标示这个对象的类型，以及一个引用的计数器，用来决定是不是可以回收这个对象。为了理解这两个头部信息对模型的影响力，我们需要继续学习下去。

类型属于对象，而不是变量

为了理解对象类型是如何使用的，请看当我们对一个变量进行多次赋值后的结果：

```
>>> a = 3                # It's an integer
>>> a = 'spam'           # Now it's a string
>>> a = 1.23             # Now it's a floating point
```

这不是典型的Python代码，但是它是可行的。a刚开始是一个整数，然后变成一个字符串，最后变成一个浮点数。这个例子对于C程序员来说，可能看起来特别奇怪，因为当我们说a = 'span'时，a的类型似乎从整数变成了字符串。

事实并非如此。在Python中，情况很简单：变量名没有类型。就像前边所说的，类型属于对象，而不是变量名。就之前的例子而言，我们只是把a引用了不同的对象。因为变

注1：有C语言背景的读者可能会发现，Python的引用类似于C的指针（内存地址）。事实上，引用是以指针的形式实现的，通常也扮演着相同的角色，尤其是那些在实地修改的对象（我们将在稍后部分进行讨论）。然而由于在使用引用时会自动解除引用，你没有办法拿引用来做些什么：这种功能避免了许多C可能出现的bug。你可以把Python的引用想成C的void指针，每当使用时就会自动运行下去。

www.TopSage.com

量没有类型，我们实际上并没有改变变量a的类型，只是让变量引用了不同类型的对象而已。实际上，Python的变量就是在特定的时间引用了一个特定的对象。

从另一方面讲，对象知道自己的类型。每个对象都包含了一个头部信息，其中标记了这个对象的类型。例如，整数对象3，包含了值3以及一个头部信息，告诉Python，这是一个整数对象［从严格意义上讲，一个指向int（整数类型的名称）的对象的指针］。'spam'字符串的对象的标志符指向了一个字符串类型。因为对象记录了它们的类型，变量就没有必要了。

注意Python中的类型是与对象相关联的，而不是和变量关联。在典型的代码中，一个给定的变量往往只会引用一种类型的对象。尽管这样，因为这并不是必须的，你将会发现Python代码比你通常惯用的代码更加灵活：如果正确的使用Python，代码能够自动以多种类型进行工作。

本书提到的这个代码有两个头部信息，一个是类型标志符，另一个是引用计数器。为了了解后者，我们需要继续学习下面内容，并简要地介绍对象生命结束时发生了什么变化。

对象的垃圾收集

在上一节的例子中，我们把变量a赋值给了不同类型的对象。但是当重新给变量a赋值时，它前一个引用值发生了什么变化？例如，在下边的语句中，对象3发生了什么变化？

```
>>> a = 3
>>> a = 'spam'
```

答案就在Python中，每当一个变量名被赋与了一个新的对象，之前的那个对象占用的空间就会被回收（如果它没有被其他的变量名或对象所引用的话）。这种自动回收对象空间的技术称作垃圾收集。

为了讲清楚，思考下面的例子，其中每个语句，把变量名x赋值给了不同的对象：

```
>>> x = 42
>>> x = 'shrubbery'        # Reclaim 42 now (unless referenced elsewhere)
>>> x = 3.1415             # Reclaim 'shrubbery' now
>>> x = [1,2,3]            # Reclaim 3.1415 now
```

首先注意x每次被设置为不同的类型的对象。再者尽管这并不是真正的情况，效果却是x的类型每次都在改变。在Python中，类型属于对象，而不是变量名。由于变量名只是引用对象而已，这种代码自然行得通。

www.TopSage.com

第二，注意对象的引用值在此过程中逐个丢弃。每一次x被赋值给一个新的对象，Python都回收了对象的空间。例如，当它赋值为字符串'shrubbery'时，对象42马上被回收（假设它没有被其他对象引用）：对象的空间自动放入自由内存空间池，等待后来的对象使用。

在内部，Python是通过保持用每个对象中的计数器记录引用指到这个对象上的次数来完成这一功能的。一旦（并精确在同一时间）这个计数器被设置为零，这个对象的内存空间就会自动回收。在前面的介绍中，假设每次x都被赋值给一个新的对象，而前一个对象的引用计数器变为零，导致它的空间被回收。

垃圾收集最直接的、可感受得到的好处就是这意味着可以在脚本中任意使用对象而不需要考虑释放内存空间。在程序运行时，Python将会清理那些不再使用的空间。实际上，与C和C++这样的底层语言相比，省去了大量的基础代码。

共享引用

到现在为止，我们已经看到了单个变量被赋值引用了多个对象的情况。现在，在交互模式下，引入另一个变量，并看一下变量名和对象的变化：

```
>>> a = 3
>>> b = a
```

输入这两行语句后，生成如图6-2所示的结果。就像往常一样，第二行会使Python创建变量b。使用的是变量a，并且它在这里没有被赋值，所以它被替换成其引用的对象3，从而b也成为这个对象的一个引用。实际的效果就是变量a和b都引用了相同的对象（也就是说，指向了相同的内存空间）。这在Python中称作是共享引用 —— 多个变量名引用了同一个对象。

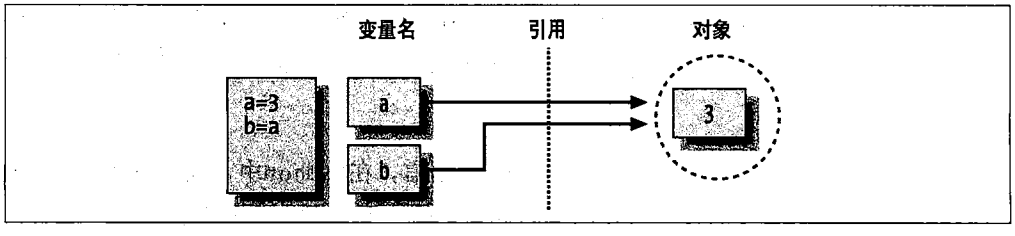


图6-2：变量名和对象，在运行赋值语句b = a之后。变量a成为对象3的一个引用。在内部，变量实际上是一个指针指向了对象的内存空间，该内存空间是通过运行常量表达式3创建的

下一步，假设运行另一个语句扩展了这样的情况。

www.TopSage.com

```
>>> a = 3
>>> b = a
>>> a = 'spam'
```

对于所有的Python赋值语句，这条语句简单地创建了一个新的对象（代表字符串值'spam'），并设置a对这个新的对象进行引用。尽管这样，这并不会改变b的值，b仍然引用原始的对象——整数3。最终的引用结构如图6-3所示。

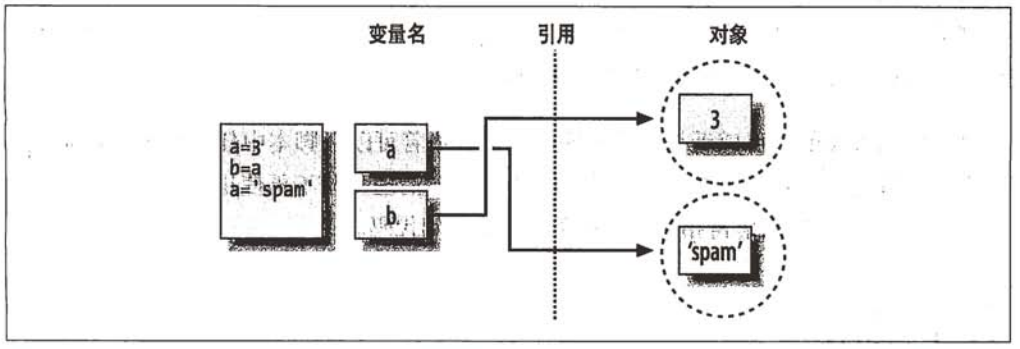


图6-3：变量名和对象，在最终运行完成赋值语句 a = 'spam' 后。变量a引用了由常量表达式'spam'所创建的新对象（例如，内存空间），但是变量b仍然引用原始的对象3。因为这个赋值运算改变的不是对象3，仅仅改变了变量a，变量b并没有发生改变

如果我们把变量b改成'spam'的话，也会发生同样的事情：赋值只会改变b，不会对a有影响。发生这种现象，跟没有类型差异一样。例如，思考下面这三条语句：

```
>>> a = 3
>>> b = a
>>> a = a + 2
```

在这里，产生了同样的结果：Python让变量a引用对象3，让b引用与a相同的对象，如图6-2所示。之前，最后的那个赋值将a设置为一个完全不同的对象（在这种情况下，整数5是表达式“+”的计算结果）。这并不会产生副作用，改变了b。事实上，是没有办法改变对象3的值的：就像第4章所介绍过的，整数是不可变的，因此没有方法再实地修改它。

认识这种现象的一种方法就是，不像其他的一些语言，在Python中，变量总是一个指向对象的指针，而不是可改变的内存区域的标签：给一变量赋一个新的值，并不是替换了原始的对象，而是让这个变量去引用完全不同的一个对象。实际的效果就是对一个变量赋值，仅仅会影响那个被赋值的变量。当可变的对象以及实地的改变进入这个场景的话，那么这个情形会有某种改变。想知道是怎样一种变化的话，请看继续学习。

www.TopSage.com

共享引用和在原处修改

正如你在这一部分后边的章节将会看到的那样，有一些对象和类型确实会在实地改变对象。例如，在一个列表中对一个偏移进行赋值确实会改变这个列表对象，而不是生成一个新的列表对象。对于支持这种在原处修改的对象，共享引用时的确需要加倍的小心，因为对一个变量名的修改会影响其他的变量。

为了进行深入地理解，让我们再看一看在第4章介绍过的列表对象。回忆一下列表，它在方括号中进行编写，是其他对象的简单集合，它支持对位置的在原处的赋值：

```
>>> L1 = [2, 3, 4]
>>> L2 = L1
```

L1是一个包含了2、3和4的对象。在列表中的元素是通过它们的位置进行读取的，所以L1[0]引用对象2，它是列表L1中的第一个元素。当然，列表自身也是对象，就像整数和字符串一样。在运行之前的两个赋值后，L1和L2引用了相同的对象，就像我们之前例子中的a和b一样（如图6-2所示）。如果我们现在像下面这样去扩展这个交互：

```
>>> L1 = 24
```

L1被简单地设置为一个不同的对象，L2仍是引用最初的列表。尽管这样，如果我们稍稍改变一下这个语句的内容，就会有明显不同的效果。

```
>>> L1 = [2, 3, 4]          # A mutable object
>>> L2 = L1                 # Make a reference to the same object
>>> L1[0] = 24              # An in-place change

>>> L1                      # L1 is different
[24, 3, 4]
>>> L2                      # But so is L2!
[24, 3, 4]
```

在这里，没有改变L1，改变了L1所引用的对象的一个元素。这类修改会覆盖列表对象中的某部分。因为这个列表对象是与其他对象共享的（被其他对象引用），那么一个像这样在原处的改变不仅仅会对L1有影响。也就是说，必须意识到当做了这样的修改，它会影响程序的其他部分。在这个例子中，也会对L2产生影响，因为它与L1都引用的相同的对象。另外，我们实际上并没有改变L2，但是它的值将发生变化。

这种行为通常来说就是你所想要的，应该了解它是如何运作的，让它按照预期去工作。这也是默认的。如果你不想要这样的现象发生，需要Python拷贝对象，而不是创建引用。有很多种拷贝一个列表的办法，包括内置列表函数以及标准库的copy模块。也许最常用的办法就是从头到尾的分片（请查阅第4章和第7章有关分片的更多内容）。

```
>>> L1 = [2, 3, 4]
>>> L2 = L1[:]          # Make a copy of L1
>>> L1[0] = 24

>>> L1
[24, 3, 4]
>>> L2                  # L2 is not changed
[2, 3, 4]
```

对L1的修改不会影响L2，因为L2引用的是L1所引用对象的一个拷贝。也就是说，两个变量指向了不同的内存区域。

注意这种分片技术不会应用在其他的可变的核心类型（字典，因为它们不是序列）上，对字典应该使用D.copy()方法。而且，注意标准库中的copy模块有一个通用的拷贝任意对象类型的调用，也有一个拷贝嵌套对象结构（例如，嵌套了列表的一个字典）的调用：

```
import copy
X = copy.copy(Y)          # Make a top-level "shallow" copy of any object Y
X = copy.deepcopy(Y)      # Make a deep copy of any object Y: copy all nested parts
```

我们将会在第8章和第9章更深入了解列表和字典，并复习共享引用和拷贝的概念。这里记住有些对象是可以在原处改变的（即可变的对象），这种对象往往对这些现象总是很开放。在Python中，这包括了列表、字典以及一些通过class语句定义的对象。如果这不是你期望的现象，可以根据需要，拷贝对象。

共享引用和相等

出于完整的考虑，本章前面介绍的垃圾收集的行为与常量相比，某些类型需要更多地思考。参照下边的语句：

```
>>> x = 42
>>> x = 'shrubbery'     # Reclaim 42 now?
```

因为Python缓存并复用了小的整数和小的字符串，就像前文提到的那样，这里的对象42也许并不像我们所说的被回收；相反地，它将可能仍被保存在一个系统表中，等待下一次你的代码生成另一个42来重复利用。尽管这样，大多数种类的对象都会在不再引用时马上回收；对于那些不会被回收的，缓存机制与代码并没有什么关系。

例如，由于Python的引用模型，在Python程序中有两种不同的方法去检查与否相等。让我们创建一个共享引用来说明：

```
>>> L = [1, 2, 3]
```


www.TopSage.com

```
>>> M = L                # M and L reference the same object
>>> L == M               # Same value
True
>>> L is M               # Same object
True
```

这里的第一种技术“==操作符”，测试两个被引用的对象是否有相同的值。这种方法往往在Python中用作相等的检查。第二种方法“is操作符”，是在检查对象的同一性。如果两个变量名精确地指向同一个对象，它会返回True，所以这是一种更严格形式的相等测试。

实际上，is只是比较实现引用的指针，所以如果必要的话是代码中检测共享引用的一种办法。如果变量名引用值相等，但是为不同的对象，它的返回值将是False，正如，当我们运行两个不同的常量表达式时：

```
>>> L = [1, 2, 3]
>>> M = [1, 2, 3]          # M and L reference different objects
>>> L == M                 # Same values
True
>>> L is M                 # Different objects
False
```

看看当我们对小的数字采用同样的操作时的结果：

```
>>> X = 42
>>> Y = 42                 # Should be two different objects
>>> X == Y
True
>>> X is Y                 # Same object anyhow: caching at work!
True
```

在这次交互中，X和Y应该是==的（具有相同的值），但不是is的（同一个对象），因为我们运行了两个不同的常量表达式。不过，因为小的整数和字符串被缓存并复用了，所以is告诉我们X和Y是引用了一个相同的对象。

实际上，如果你确实想刨根问底的话，你能够向Python查询对一个对象引用的次数：在sys模块中的getrefcount函数会返回对象的引用次数。例如，在IDLE GUI中查询整数对象1时，它会报告这个对象有837次重复引用（绝大多数都是IDLE系统代码所使用的）：

```
>>> import sys
>>> sys.getrefcount(1)     # 837 pointers to this shared piece of memory
837
```

这种对象缓存和复用的机制与代码是没有关系的（除非你运行这个检查）。因为不能改

变数字和字符串，所以无论对同一个对象有多少个引用都没有关系。然而，这种现象也反映了Python为了执行速度而优化其模块的众多方法中的一种。

动态类型随处可见

在使用Python的过程中，真的没有必要去用圆圈和箭头画变量名/对象的框图。尽管在刚入门的时候，它会帮助你跟踪它们的引用结构,理解不常见的情况。例如，如果在程序中，当传递过程中一个可变的对象发生了改变时，很有可能你就是本章话题的第一现场的见证者了。

此外，尽管目前来说动态类型看起来有些抽象，你最终还是需要关注它的。因为在Python中，任何东西看起来都是通过赋值和引用工作的，对这个模型的基本了解在不同的场合都是很有帮助的。就像将会看到的那样，它也会在赋值语句，变量参数，`for`循环变量，模块导入等很多场合发挥作用。值得高兴的是这是Python中唯一的赋值模型。一旦你对动态类型上手了，将会发现它在这门语言中任何地方都有效。

从最实际的角度来说，动态类型意味着你将写更少的代码。尽管这样，同等重要的是，动态类型也是Python中多态（我们在第4章介绍的一个概念，将会在本书后面再次见到）的根本。因为我们在Python代码中没有对类型进行约束，它具备了高度的灵活性。就像你将会看到的那样，如果使用正确的话，动态类型和多态产生的代码，可以自动地适应系统的新需求。

本章小结

这章对Python的动态类型（也就是Python自动为我们跟踪对象的类型，不需要我们在脚本中编写声明语句）进行了深入的学习。在这个过程中，我们学会了Python中变量是如何通过引用关联在一起的，还探索了垃圾收集的概念，学到了对象共享引用是如何影响多个变量的，并看到了Python中引用是如何影响相等的概念的。

因为在Python中只有一个赋值模型，并且赋值在这门语言中到处都能碰到，所以在我们继续学习之前掌握这个模型是很有必要的。接下来的章节测试会帮助你复习这一章的概念。这在下一章将会继续我们的学习对象之旅——学习字符串。

www.TopSage.com

本章习题

1. 思考下面三条语句。它们会改变A打印出的值吗？

```
A = "spam"
B = A
B = "shrubbery"
```

2. 思考下面三条语句。它们会改变A的值吗？

```
A = ["spam"]
B = A
B[0] = "shrubbery"
```

3. 这样如何，A会改变吗？

```
A = ["spam"]
B = A[:]
B[0] = "shrubbery"
```

习题解答

1. 不：A仍会作为"spam"进行打印。当B赋值为字符串"shrubbery"时，所发生的只是变量B被重新设置为指向了新的字符串对象。A和B最初共享（即引用或指向）了同一个字符串对象"spam"，但是在Python中这两个变量名从未连接在一起。因此，设置B为另一个不同的对象对A没有影响。如果这里最后的语句变为B = B + 'shrubbery'，也会发生同样的事情。另外，合并操作创建了一个新的对象作为其结果，并将这个值只赋值给了B。我们永远都不会在实地覆盖一个字符串（数字，或元组），因为字符串是不可变的。

2. 是：A现在打印为["shrubbery"]。从技术上讲，我们既没有改变A也没有改变B，我们改变的是这两个变量共同引用（指向）的对象的一部分，通过变量B在原处覆盖了这个对象的一部分内容。因为A像B一样引用了同一个对象，这个改变也会对A产生影响。

3. 不会：A仍然会打印为["spam"]。由于分片表达式语句会在被赋值给B前创建一个拷贝，这次对B在原处赋值就不会有影响了。在第二个赋值语句后，就有了两个拥有相同值的不同列表对象了（在Python中，我们说它们是==的，却不是is的）。第三条赋值语句会改变指向B的列表对象，而不会改变指向A的列表对象。

大家网
TopSage.com

第7章

字符串

我们内置对象之旅的下一个主要的类型为Python字符串 —— 一个有序的字符的集合，用来存储和表现基于文本的信息。我们曾在第4章对字符串进行过简单的介绍。这里我们将会更深入地再次学习，补充一些当时跳过的细节。

从功能的角度来看，字符串可以用来表现能够像文本那样编辑的任何信息：符号和词语（例如，你的名字）、载入到内存中的文本文件的内容、Internet网址和Python程序等。

你也许在其他语言中也用过字符串，Python当中的字符串与其他语言（例如，C语言）中的字符数组扮演着同样的角色，然而从某种程度上来说，它们是比数组更高层的工具。在Python中，字符串变成了一种强大的处理工具集，这一点与C语言不同。并且Python跟像C这样的语言不一样，没有单个字符的这种类型，取而代之的是可以使用一个字符的字符串。

严格地说，Python的字符串被划分为不可变序列这一类别，意味着这些字符串所包含的字符存在从左至右的位置顺序，并且他们不可以在原处修改。实际上，字符串是我们将学习的从属于稍大一些的对象类别 —— 序列的第一个代表。请格外留意本章所介绍的序列操作，因为它在今后要学习的其他序列类型（例如列表和元组）中同样也适用。

表7-1介绍了本章将要讨论到的常见的字符串常量和操作。空字符串表示为一对引号（单引号或双引号），其中什么都没有，还有许多方法编写字符串。处理字符串支持表达式的操作，例如，合并（组合字符串）、分片（抽取一部分）、索引（通过偏移获取）等。除了表达式，Python还提供了一系列的字符串方法，可以执行字符串常见的特定任务，还有用于执行如模式匹配这样的高级文本处理的任务模块。我们将会在本章学习这些内容。

www.TopSage.com

表 7-1：常见字符串常量和表达式

操作	解释
`s1 = ''`	空字符串
`s2 = "spam's"`	双引号
`block = """..."""`	三重引号块
`s3 = r'\temp\spam'`	Raw字符串
`s4 = u'spam'`	Unicode字符串
`s1 + s2` `s2 * 3`	合并，重复
`s2[i]` `s2[i:j]`	
`len(s2)`	索引，分片，求长度
`"a %s parrot" % type`	字符串格式化

`s2.find('pa')`
`s2.rstrip()`
`s2.replace('pa', 'xx')`
`s1.split(',')`
`s1.isdigit()`
`s1.lower()`
	字符串方法调用: 搜索、移除空格、替换、用展位符分隔、内容测试、短信息转换等

`for x in s2` `'spam' in s2`	迭代,成员关系

除了核心系列的字符串工具以外，Python通过标准库re模块（正则表达式）还支持更高级的基于模式的字符串处理，这在第4章介绍过。本章将会以字符串常量的形式以及基本的字符串操作作为开始，之后将会学习字符串方法和格式等更高级的工具。

字符串常量

从整体上来讲，Python中的字符串用起来还是相当的简单的。也许它们最复杂的事情就是在代码中有如此多的方法去编写它们：

- 单引号：'spa"m'。
- 双引号："spa'm"。
- 三引号：'''... spam ...''', """... spam ..."""。
- 转义字符："s\tp\na\0m"。

- Raw字符串：r"C:\new\test.spm"。
- Unicode字符串：u'eggs\u0020spam'。

单引号和双引号的形式尤其常见。其他的形式都是有特定角色的。下面依次介绍这些方法。

单双引号字符串是一样的

在Python字符串中，单引号和双引号字符是可以互换的。也就是说，字符串常量表达式可以用两个单引号或两个双引号来表示——两种形式同样有效并返回相同类型的对象。例如，程序一旦这样编写，就意味着二者是等效的：

```
>>> 'shrubbery', "shrubbery"
('shrubbery', 'shrubbery')
```

之所以这两种形式都能够使用是因为你不使用反斜线转义字符就可以实现在一个字符串中包含其余种类的引号。可以在一个双引号字符所包含的字符串中嵌入一个单引号字符，反之亦然：

```
>>> 'knight"s', "knight's"
('knight"s', "knight's")
```

此外，Python自动在任意的表达式中合并相邻的字符串常量，尽管可以简单地在它们之间增加+操作符来明确地表示这是一个合并操作。

```
>>> title = "Meaning " 'of' " Life"      # Implicit concatenation
>>> title
'Meaning of Life'
```

注意到在这些字符串之间增加逗号会创建一个元组，而不是一个字符串。并且Python倾向于打印所有这些形式的字符串为单引号，除非字符串内有了单引号了。你也能通过反斜线转义字符去嵌入引号：

```
>>> 'knight\'s', "knight\"s"
("knight's", 'knight"s')
```

想要了解原因，需要知道一般情况转义字符是如何工作的。

用转义序列代表特殊字节

上一个例子通过在引号前增加一个反斜线的方式可以在字符串内部嵌入一个引号。这是字符串中的一个常见的表现模式：反斜线用作是引入特殊的字节编码，是转义序列。

www.TopSage.com

转义序列让我们能够在字符串中嵌入不容易通过键盘输入的字节。字符串常量中字符“\”，以及在它后边的一个或多个字符，在最终的字符串对象中会被一个单个字符所替代，这个字符通过转义序列定义了一个二进制值。例如，这里有一个五个字符的字符串，其中嵌入了一个换行符和一个制表符：

```
>>> s = 'a\nb\tc'
```

其中两个字符“\n”表示一个单个字符 —— 在字符集中包含了换行字符这个值（通常来说，ASCII编码为10）的字节。类似的，序列“\t”替换为制表符。这个字符串打印时的格式取决于打印的方式。交互模式下是以转义字符的形式回显的，但是print会将其解释出来：

```
>>> s
'a\nb\tc'
>>> print s
a
b       c
```

为了清楚地了解这个字符串中到底有多少个字节，使用内置的len函数。它会返回一个字符串中到底有多少字节，无论它是如何显示的。

```
>>> len(s)
5
```

这个字符串长度为5个字节：分别包含了一个ASCII a字符，一个换行字符、一个ASCII b字符等。注意原始的反斜线字符并不真正和字符串一起存储在内存中。对于这些特殊字符的编写，Python提供了一整套转义字符序列，如表7-2所示。

表7-2：字符串反斜线字符

转义	意义
\newline	忽视（连续）
\\	反斜线（保留\）
\'	单引号（保留'）
\"	双引号（保留"）
\a	响铃
\b	倒退
\f	换页
\n	新行（换行）
\r	返回

www.TopSage.com

表7-2：字符串反斜线字符（续）

转义	意义
\t	水平制表符
\v	垂直制表符
\N{*id*}	Unicode数据库ID
\uhhhh	Unicode 16位的十六进制值
\Uhhhh...	Unicode 32位的十六进制值[a]
\xhh	十六进制值
\ooo	八进制值
\0	Null（不是字符串结尾）
other	不转义（保留）

a. \Uhhhh... 转义序列带有八个十六进制数字（h）；\u和\U只能使用于Unicode常量之中。

一些转义序列允许你一个字符串的字节中嵌入绝对的二进制值。例如，这里有一个五个字符的字符串，其中嵌入了两个二进制零字符。

```
>>> s = 'a\0b\0c'
>>> s
'a\x00b\x00c'
>>> len(s)
5
```

在Python中，零（空）字符不会像C语言那样去中断一个字符串。相反，Python在内存中保持了整个字符串的长度和文本。事实上，Python没有字符会中断一个字符串。这里有一个完全由绝对的二进制转义字符编码的字符串 —— 一个二进制1和2（以八进制编码）以及一个二进制3（以十六进制编码）。

```
>>> s = '\001\002\x03'
>>> s
'\x01\x02\x03'
>>> len(s)
3
```

对这有所了解在当使用Python处理二进制数据文件时显得格外重要。因为它的内容在脚本中是以字符串的形式表现的，处理包含了任意种类的二进制字符值的二进制文件也是完全可行的（在第9章有更多关于文件的细节）（注1）。

注1：如果你对二进制数据文件特别感兴趣的话，其主要的不同就在于它们是你在二进制模式下打开的（使用open模式的标志位b，例如'rb'，'wb'等）。请参照第9章中介绍的标准struct模块，它可以解析处理冲一个文件载入的二进制数据。

www.TopSage.com

最后，如表7-2最后一条所显示的，如果Python没有作为一个合法的转义编码识别出在“\”后的字符，它就会简单地在最终的字符串中保留反斜线。

```
>>> x = "C:\py\code"          # Keeps \ literally
>>> x
'C:\\py\\code'
>>> len(x)
10
```

除非你能够将表7-2中的所有内容都记住，这样你也许不会依赖这种现象（注2）。如果希望在脚本中编写明确的常量反斜线，将重复两个反斜线（“\\”是“\”的转义字符）或者使用raw字符串。raw将会在下一部分内容进行介绍。

字符串抑制转义

正如我们已经看到的，转义序列用来处理嵌入在字符串中的特殊字节编码是很合适的。尽管这样，有时候，由于为了引入转义字符而使用适应的反斜线的处理会带来一些麻烦。其实这相当常见，例如，Python新手有时会像下面这样使用文件名参数去尝试打开一个文件。

```
myfile = open('C:\new\text.dat', 'w')
```

他们会认为这将会打开一个在*C:\new*目录下名为*text.dat*的文件。问题是这里有“\n”，会被识别为一个换行字符，并且“\t”会被一个制表符所替代。结果就是，这个调用是尝试打开一个名为 *C:(*换行*)ew(*制表符*)ext.dat*的文件，而不是我们所期待的结果。

这正是使用raw字符串所要解决的问题。如果字母*r*（大写或小写）出现在字符串的第一引号的前面，它将会关闭转义机制。这个结果就是Python会将反斜线作为常量来保持，就像输入的那样。因此为了避免这种文件名的错误，记得在Windows中增加字幕*r*。

```
myfile = open(r'C:\new\text.dat', 'w')
```

还有一种办法：因为两个反斜线是一个反斜线的转义序列，能够通过简单地写两个反斜线去保留反斜线。

```
myfile = open('C:\\new\\text.dat', 'w')
```

实际上，当打印一个嵌入了反斜线字符串时，Python自身也会使用这种写两个反斜线的方法：

注2：在课堂上，我确实见过把这张表中的大多数或全部内容都记住的人；而我通常会认为这样是没有必要的，但是实际上我也把它们都记住了。

www.TopSage.com

```
>>> path = r'C:\new\text.dat'
>>> path                              # Show as Python code
'C:\\new\\text.dat'
>>> print path                        # User-friendly format
C:\new\text.dat
>>> len(path)                         # String length
15
```

大家网 TopSage.com

当与数字表示相同时，在交互提示打印结果的默认格式和编码一样，并且在输出中有转义的反斜线。打印语句提供了一种对用户更友好的格式，在每处实际上仅有一个反斜线。为了验证这种情况，你可以检查内置len函数的结果，这会返回这个字符串字节数，与其显示的格式没有关系。如果计算了整个路径的输出中的字符数，你会发现每个反斜线只占一个字符，所以总计15个字符。

除了在Windows下的文件夹路径，raw字符串也在正则表达式（文本模式匹配，通过在第4章介绍过的re模块支持）中常见。注意Python脚本会自动在Windows和Unix的路径中使用斜线表示字符串路径，因为Python试图以可移植的方法解释路径。尽管这样，如果你的编写的路径使用Windows的反斜线时raw字符串是很有用处的。

三重引号编写多行字符串块

到现在为止，你已经在实践中看到了单引号、双引号、转义字符以及raw字符串。Python还有一种三重引号内的字符串常量格式，有时候称作块字符串，这是一种对编写多行文本数据来说很便捷的语法。这个形式以三重引号开始（单引号和双引号都可以），并紧跟任意行数的文本，并且以开始时的同样的三重引号结尾。嵌入在这个字符串文本中的单引号和双引号也会但不是必须转义 —— 直到Python看到和这个常量开始时同样的三重引号，这个字符串才会结束。例如：

```
>>> mantra = """Always look
...  on the bright
... side of life."""
>>>
>>> mantra
'Always look\n on the bright\nside of life.'
```

这个字符串包含三行（在一些界面，对于连续的行来说交互提示符会变成…。IDLE就会简单地开始下一行）。Python收集了所有在三重引号之内的文本到一个单独的多行字符串中，并在代码折行处嵌入了换行字符（\n）。注意在这个常量中，结果中的第二行开头有一个空格，第三行却没有 —— 输入的是什么，得到的就是什么。

三重引号字符串在程序需要输入多行文本的的任何时候都是很有用的。例如，嵌入多行

www.TopSage.com

错误信息或在源文件中编写HTML或XML代码。能够直接嵌入这样的文本块而不需要求助于外部的文本文件，或者借助直接合并和换行字符。

三重引号字符串字符串常用于文档字符串，当它出现在文件的特定地点时，被当作注释一样的字符串常量（在本书的后面会介绍更多细节）。这并非只能使用三重引号的文本块，但是它们往往是可以用作多行注释的。

最后，三重引号字符串经常在开发过程中作为一种恐怖的黑客风格的方法去废除一些代码（好吧，这并不恐怖，并且它实际上是一种常规的惯例）。如果希望让一些行的代码不工作，之后再次运行代码，可以简单地在这几行前、后加入三重引号，就像这样。

```
X = 1
"""
import os
print os.getcwd()
"""
Y = 2
```

有些黑客风格因为Python确实并无意让字符串用这样的方法去废除一些行的代码，但是这也许对性能来说没有什么显著影响。对于大段的代码，这也比手动在每一行之间加入井号之后删除它们要容易得多。如果使用没有对编辑Python代码有特定支持的文本编辑器时尤其如此。在Python中，实用往往会胜过美观。

字符串编码更大的字符集

最后一种在脚本中编写字符串的方法也许是最特别的，而且它在web和XML处理之外是较为少见的。Unicode字符串有时称为“宽”字符串。因为每个字符也许在内存中会占用大于一个字节的空间，与一般字符串相比，Unicode字符串允许程序编码更丰富的字符集。

Unicode字符串典型地应用于支持国际化的应用（一些时候称为“i18”，为了压缩这个英文单词头尾之间的18个字母）。例如，它允许程序员在Python脚本中直接支持从欧洲到亚洲的字符集。因为这些字符集有很多字符，以至于不能够仅使用一个字节来表示，所以Unicode往往就是用来处理这些形式的文本的。

在Python中，可以在脚本中通过在开头的引号前增加字母U（大写或小写）编写一个Unicode字符串：

```
>>> u'spam'
u'spam'
```


www.TopSage.com

从技术上说，这种语法产生了一个Unicode字符串对象 —— 与一般的字符串类型不同的另一种数据类型。尽管这样，Python允许表达式中自由地混合Unicode字符串和一般的字符串，并作为混合类型的结果将结果转为Unicode（在下一部分的+合并操作中介绍更多信息）：

```
>>> 'ni' + u'spam'          # Mixed string types
u'nispam'
```

实际上，Unicode字符串是定义为支持下一节你将会看到的所有常见的字符串处理的操作，所以类型上的不同对于你的代码往往不重要。就像一般的字符串一样，Unicode字符串也可以被合并、索引、分片、通过re模块进行匹配，并且不能够在进行实地修改。

如果需要明确地在这两种类型间转换，你可以使用内置的str和unicode函数：

```
>>> str(u'spam')            # Unicode to normal
'spam'
>>> unicode('spam')         # Normal to Unicode
u'spam'
```

因为Unicode是用来处理多字节字符的，所以能够使用特殊的“\u”和“\U”转义字符去编码大于8bit的二进制值：

```
>>> u'ab\x20cd'             # 8-bit/1-byte characters
u'ab cd'
>>> u'ab\u0020cd'           # 2-byte characters
u'ab cd'
>>> u'ab\U00000020cd'       # 4-byte characters
u'ab cd'
```

这些语句的第一行嵌入了一个空格字符的二进制代码。它的二进制值以十六进制表示为“x20”。第二和第三行语句做了相同的事情，分别使用了2字节和4字节的Unicode转义字符。

甚至在不知道需要Unicode字符串的情况下，你也许都没有意识到使用了它。因为一些编程的接口（例如，Windows上的COM API以及一些XML解析）都是以Unicode字符串表现文本的，它也许会采用它的方法作为API的输入或结果进入脚本，而且也许有时候需要在一般字符串和Unicode字符串进行相互之间的类型转换。

因为Python在大多数环境中是等同对待这两种字符串类型的，然而当前的Unicode字符串对于你的代码来说是透明的 —— 你可以很大程度上忽略文本是使用Unicode对象进行传递的并可以使用一般的字符串操作。

www.TopSage.com

Unicode对于Python而言是个有益的补充，并且当需要时在脚本中有这样的数据可以容易处理。当然，对于Unicode还有很多可以讲的内容。例如：

- Unicode对象提供了一种编码方法，可以将Unicode字符串转换为一般的使用特定编码的8-bit字符串。
- 内置函数unicode和模块codecs支持注册“编码”（对“编码器和解码器来说”）。
- 在codecs模块中的open函数可以处理Unicode文本文件，这个文件中的每个字符都以大于一个字节的空间进行存储。
- unicodedata模块提供了对Unicode字符数据库的存取功能。
- sys模块包括了对默认Unicode编码方案（默认往往是ASCII）的获取及设置的调用。
- 你可以混合raw和Unicode字符串（例如，ur'a\b\c'）。

因为Unicode是一个相对较为高级的并且不是很常见的工具，我们不会在这个介绍中继续讨论它。参考Python标准手册以获得Unicode其他的内容。

注意： 在Python 3.0中，字符串将会有些改动：在3.0中目前的str字符串将总是Unicode，并且将会有一个新的“bytes”类型，作为一个可变的序列用来表现短字符的字符串。一些文件的read操作也许会返回bytes而不是str（例如，读取二进制文件）。这是仍然在规划之中的，所以请参考3.0的发行报告来获取更多细节。

实际应用中的字符串

一旦你已经使用我们刚刚见过的常量表达式创建了一个字符串，必定会很想用它去做些什么。这一部分以及后面的两部分将会介绍字符串的基础知识、格式，以及方法，这是Python语言中文本处理的首要工具。

基本操作

打开Python解释器，开始学习理解在表7-1中列举的字符串基本操作。字符串可以通过+操作符进行合并并且可以通过*操作符进行重复：

```
% python
>>> len('abc')          # Length: number of items
3
>>> 'abc' + 'def'       # Concatenation: a new string
'abcdef'
```


www.TopSage.com

```
>>> 'Ni!' * 4          # Repetition: like "Ni!" + "Ni!" + ...
'Ni!Ni!Ni!Ni!'
```

从形式上讲，两个字符串对象相加创建了一个新的字符串对象，这个对象就是两个操作的对象内容相连。重复就像在字符串后再增加一定数量的自身。无论是哪种情况，Python都创建了任意大小的字符串。在Python中没有必要去做任何声明，包括数据结构的大小（注3）。内置的len函数返回了一个字符串（或任意有长度的对象）的长度。

重复最初看起来有些费解，然而在相当多的场合使用起来十分顺手。例如，为了打印包含80个横线的一行，你可以一个一个数到80，或者让Python去帮你数：

```
>>> print '------- ...more... ---'      # 80 dashes, the hard way
>>> print '-'*80                        # 80 dashes, the easy way
```

注意操作符重载已经发挥了作用。这是使用了与在应用于数字时执行加法和乘法的相同的操作符+和*。Python执行了正确的操作因为它知道加和乘的对象的类型。但是小心：这个规则并不和你预计的那样一致。例如，Python不允许你在+表达式中混合数字和字符串：'abc'+9会抛出一个错误而不会自动地将9加载到个字符串上。

正如表7-1最后一行所示，可以使用for语句在一个字符串中进行迭代，并使用in表达式操作符进行成员关系的测试，这实际上是一种搜索。

```
>>> myjob = "hacker"
>>> for c in myjob: print c,          # Step through items
...
h a c k e r
>>> "k" in myjob                      # Found
True
>>> "z" in myjob                      # Not found
False
```

for循环指派了一个变量去获取一个序列（对应这里的是一个字符串）其中的元素，并对每一个元素执行一个或多个语句。实际上，这里变量c成为了一个在这个字符串中步进的指针。我们将会在本书稍后对类似的迭代工具进行讨论。

注3：与C字符数组不同的是，使用Python字符串时，你不用分配或管理储存数组，只要在需要时创建字符串对象，让Python去管理底层的内存空间。就像上一章所说的，Python会使用引用值计数的垃圾收集策略，自动回收无用对象的内存空间。每个对象都会记录引用其的变量名、数据结构等的次数，一旦计数值到了零，Python就会释放该对象的空间。这种方式意味着Python不用停下来扫描所有内存，从而寻找要释放的无用空间（一个额外的垃圾组件也会收集循环对象）。

www.TopSage.com

索引和分片

因为将字符串定义为字符的有序集合，所以我们能够通过其位置获得他们的元素。在Python中，字符串中的字符是通过索引（通过在字符串之后的方括号中提供所需要的元素的数字偏移量）提取的。你将获得在特定位置的一个字符的字符串。

就像在C语言中一样，Python偏移量是从0开始的，并比字符串的长度小1。与C语言不同，Python还支持类似在字符串中使用负偏移这样的方法从序列中获取元素。从技术上讲，一个负偏移与这个字符串的长度相加后得到这个字符串的正的偏移值。能够将负偏移认作是从结束处反向计数。下面的交互例子进行了说明。

```
>>> S = 'spam'
>>> S[0], S[-2]                    # Indexing from front or end
('s', 'a')
>>> S[1:3], S[1:], S[:-1]          # Slicing: extract a section
('pa', 'pam', 'spa')
```

第一行定义了有一个4个字符的字符串，并将其赋予变量名S。下一行用两种方法对其进行索引：S[0]获取了从最左边开始偏移量为0的元素（单字符的字符串's'），并且S[-2]获取了从尾部开始偏移量为2的元素［或等效的，在从头来算偏移量为(4 +−2)的元素］。偏移和分片的网格示意图如图7-1所示（注4）。

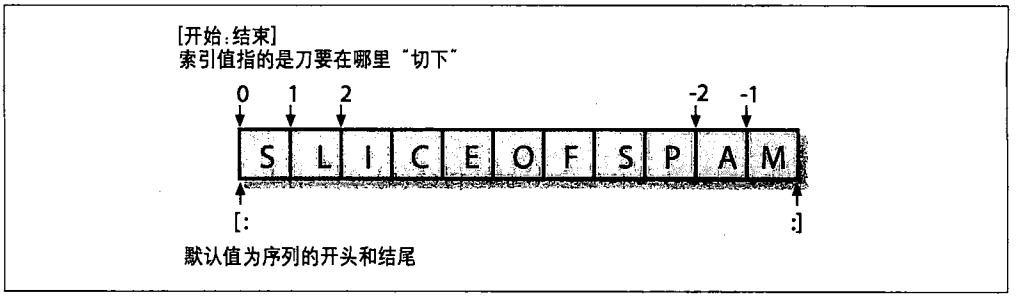


图 7-1：偏移和分片：位置偏移从左至右（偏移0为第一个元素），而负偏移是由末端右侧开始计算（偏移-1为最后一个元素）。这两种偏移均可以在索引及分片中作为所给出的位置

上边的例子中的最后一行对分片进行了演示。也许思考分片最好的办法就是将其看作解析（分析结构）的一种形式，特别是当你对字符串应用分片时——它让我们能够从一整个字符串中分离提取出一部分内容（子字符串）。分片可以用作提取部分数据，分离出前、后缀等场合。我们将会在本章稍后看到另一个分片用作解析的例子。

注4：更有数学思维的读者（以及我的课堂上的学生）有时会发现这里有些不对称：最左侧的元素的偏移为0，而最右侧的元素的偏移为−1。唉，在Python中是没有所谓−0这样的偏移啊。

这里将解释分片是如何运作的。当使用一对以冒号分隔的偏移索引字符串这样的序列对象时，Python将返回一个新的对象，其中包含了以这对偏移所标识的连续的内容。左边的偏移被取作是下边界（包含下边界在内），而右边的偏移被认作是上边界（不包含上边界在内）。Python将获取从下边界直到但不包括上边界的所有元素，并返回一个包含了所获取的元素的新的对象。如果被省略，上、下边界的默认值对应分别为0和分片的对象的长度。

例如，我们所看到的那个例子，S[1:3]提取出偏移为1和2的元素。也就是说，它抓取了第二个和第三个元素，并在偏移量为3的第四个元素前停止。下一个S[1:]得到了从第一个元素到上边界之间的所有元素，而上边界在未给出的情况下，默认值为字符串的长度。最后，S[:-1]获取了除了最后一个元素之外的所有元素 —— 下边界默认为0，而-1对应最后一项，不包含在内。

这在一开始学习看起来有些令人困惑，但是一旦你掌握了诀窍以后索引和分片就成为了简单易用的强大工具。记住，如果你不确定分片的意义，可以交互地试验一下。在下一章，将会介绍可以通过对一个分片进行赋值而改变一个特定对象的一部分内容。下面是用来参考的细节的总结：

- 索引（S[i]）获取特定偏移的元素：
  - 第一个元素的偏移为0。
  - 负偏移索引意味着从最后或右边反向进行计数。
  - S[0]获取了第一个元素。
  - S[-2]获取了倒数第二个元素（就像S[len(s)-2]一样）。
- 分片（S[i:j]）提取对应的部分作为一个序列：
  - 上边界并不包含在内。
  - 分片的边界默认为0和序列的长度，如果没有给出的话。
  - S[1:3]或去了从偏移为1的元素，直到但不包括偏移为3的元素。
  - S[1:]获取了从偏移为1直到末尾（偏移为序列长度）之间的元素。
  - S[:3]获取了从偏移为0直到但是不包括偏移为3之间的元素。
  - S[:-1]获取了从偏移为0直到但是不包括最后一个元素之间的元素。
  - S[:]获取了从偏移0到末尾之间的元素，这有效地实现顶层S拷贝。

上面列出的最后一项成为了一个非常常见的技巧：它实现了一个完全的顶层的序列对象

www.TopSage.com

的拷贝 —— 一个有相同值，但是是不同内存片区的对象（在第9章介绍更多关于拷贝的内容）。这对于像字符串这样的不可变对象并不是很有用，但是对于可以在实地修改的对象来说很实用，例如列表。在下一章，将会看到通过偏移进行索引（方括号）的语法也可以通过键对字典进行索引；操作看起来很相似，但是却有着不同的解释。

扩展分片：第三个限制值

在Python2.3中，分片表达式增加了一个可选的第三个索引，用作步进（有时称为是stride）。步进添加进每个提取的元素的索引中。完整形式的分片现在变成了X[I:J:K]，这表示“索引X对象中的元素，从偏移为I直到偏移为J－1，每隔K元素索引一次”。第三个限制，K，默认为1，这也就是通常在一个切片中从左至右提取每一个元素的原因。如果你定义了一个明确的值，那么能够使用这个第三个限制去跳过某些元素或反向排列它们的顺序。

例如，X[1:10:2]会取出X中，偏移值1~9之间，间隔了一个元素的元素，也就是收集偏移值1、3、5、7和9之处的元素 。如同往常，第一和第二限制值默认为0以及序列的长度，所以，X[::2]会取出序列从头到尾，每隔一个元素的元素：

```
>>> S = 'abcdefghijklmnop'
>>> S[1:10:2]
'bdfhj'
>>> S[::2]
'acegikmo'
```

也可以使用负数作为步进。例如，分片表达式"hello"[::-1]返回一个新的字符串"olleh" —— 前两个参数默认值分别为0和序列的长度，就像之前一样，步进－1表示分片将会从右至左进行而不是通常的从左至右。因此，实际效果就是将序列进行反转：

```
>>> S = 'hello'
>>> S[::-1]
'olleh'
```

通过一个负数步进，两个边界的意义实际上进行了反转。也就是说，分片S[5:1:-1]以反转的顺序获取从2到5的元素（结果是偏移为5、4、3和2的元素）。

```
>>> S = 'abcedfg'
>>> S[5:1:-1]
'fdec'
```

像这样使用三重限制的列表来跳过或者反序输出是很常见的情况，可以通过参看Python的标准库手册来获得更多细节，或者可以在交互模式下运行几个实例 —— 其中往往比这

里包含更多内容。我们将会在这本书稍后部分再次学习这种三重限制的分片，学习时会与for循环进行对比。

为什么要在意：分片

贯穿本书始末，经常会使用边框（就像现在的情况一样）这样的形式，让你对正在介绍的语言特性有个直观的认识，了解其在真实的程序中的典型应用。因为在看到绝大多数的Python内容之前，你将不会遇到很多的真实应用场景，这些边框中介绍的内容也许包含了许多尚未介绍的话题。不过你应该将这些内容看作是预览的途径，这样你就能够找到这些抽象的语言概念是如何在平常的编程任务中应用的。

例如，稍后你将会看到在一个系统命令行中启动Python程序时罗列出的参数，这使用了内置的sys模块中的argv属性：

```
# File echo.py
import sys
print sys.argv

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

通常，你只对跟随在程序名后边的参数感兴趣。这就是一个分片的典型应用：一个分片表达式能够返回除了第一项之外的所有元素的列表。这里，sys.argv[1:]返回所期待的列表['-a', '-b', '-c']。之后就能够不考虑最前边的程序名而只对这个列表进行处理。

分片也常常用作清理输入文件的内容。如果知道一行将会以行终止字符（\n换行字符标识）结束，你就能够通过一个简单的表达式，例如，line[:-1]，把这行除去最后一个字符之外的所有内容提取出来（默认的左边界为0）。无论是以上哪种情况，分片都表现出了比底层语言的实现更明确直接的逻辑关系。

值得注意的是，为了去掉离换行字符常常推荐采用line.rstrip方法，因为这个调用将会留下没有换行字符那行的最后一个字符，而这在一些文本编辑器工具中是很常见的。当你确定每一行都是通过换行字符终止时适宜使用分片。

字符串转换工具

Python的设计座右铭之一就是拒绝猜的诱惑。作为一个简单的例子，在Python中不能够让数字和字符串相加，甚至即使字符串看起来像是数字也不可以（例如，一个全数字的字符串）。

www.TopSage.com

大家网
TopSage.com

```
>>> "42" + 1
TypeError: cannot concatenate 'str' and 'int' objects
```

这是有意设计的。因为+既能够进行加法运算也能够进行合并操作，这种会话的选择会变得模棱两可。因此，Python将其作为错误来处理。在Python中，如果让操作变得更复杂，通常就要避免这样的语法。

下面，要做的就是，如果脚本从文件或用户界面得到了一个作为字符串出现的数字该怎么办？这里的技巧就是，需要使用转换工具预先处理，把字符串当作数字，或者把数字当作字符串。例如：

```
>>> int("42"), str(42)          # Convert from/to string
(42, '42')
>>> repr(42), `42`              # Convert to as-code string
('42', '42')
```

int函数将字符串转换为数字，而str将数字转换为字符串表达形式（实际上，它看起来和打印出来的效果是一样的）。repr函数以及之前的等效函数，反引号表达式，都能够将一个对象转换为其字符串形式，然而这些返回的对象将作为代码的字符串，可以重新创建对象（对于字符串来说，如果是使用print语句进行显示的话，其结果需要引号括起来）。请参考第5章相关内容，查看str与repr之间在这方面的区别。其中，int和str是通用的指定转换工具。

尽管你不能混合字符串和数字类型进行像+这样的加法，你能够在进行这样的操作之前手动进行转换：

```
>>> S = "42"
>>> I = 1
>>> S + I
TypeError: cannot concatenate 'str' and 'int' objects

>>> int(S) + I                  # Force addition
43

>>> S + str(I)                  # Force concatenation
'421'
```

类似的内置函数可以把浮点数转换成字符串，或者把字符串转换成浮点数：

```
>>> str(3.1415), float("1.5")
('3.1415', 1.5)

>>> text = "1.234E-10"
>>> float(text)
1.2340000000000001e-010
```


www.TopSage.com

之后，我们将会更深入地学习内置eval函数。它将会运行一个包含了Python表达式代码的字符串，并能够将一个字符串转换为任意类型的对象。函数int和float只能够对数字进行转换，然而这样的约束意味着它往往要更快一些（并且更安全，因为它不能够接受那些随意的表达式代码）。正如我们在第5章看到的那样，字符串格式表达式也能够提供一种数字到字符串的转换方法。我们将会在本章稍后进行格式的讨论。

字符串代码转换

同样是转换，单个的字符也可以通过将其传给内置的ord函数转换为其对应的ASCII码 —— 这个函数实际上返回的是这个字符在内存中对应的字符的二进制值。而chr函数将会执行相反的操作，获取ASCII码并将其转化为对应的字符：

```
>>> ord('s')
115
>>> chr(115)
's'
```

可以利用循环完成对字符串内所有字符的函数运算。这些工具也可以用来执行一种基于字符串的数学运算。例如，为了生成下一个字符，我们可以预先将当前字符转换为整型并进行这样的数学运算：

```
>>> S = '5'
>>> S = chr(ord(S) + 1)
>>> S
'6'
>>> S = chr(ord(S) + 1)
>>> S
'7'
```

至少对于单个字符的字符串来说，可通过调用内置函数int，将字符串转换为整数：

```
>>> int('5')
5
>>> ord('5') - ord('0')
5
```

这样的转换可以与循环语句一起配合使用，可以将一个表示二进制数的字符串转换为等值的整数 —— 每次都将当前的值乘以2，并加上下一位数字的整数值：

```
>>> B = '1101'
>>> I = 0
>>> while B:
...     I = I * 2 + (ord(B[0]) - ord('0'))
...     B = B[1:]
...
```


www.TopSage.com

```
>>> I
13
```

左移运算（I<<1）与在这里乘2的运算是一样的，由于我们还没有详细地介绍循环，那么建议把这部分程序当作一个试验来运行。

修改字符串

还记得“不可变序列”吗?不可变的意思就是不能在实地修改一个字符串（例如，给一个索引进行赋值）。

```
>>> S = 'spam'
>>> S[0] = "x"
Raises an error!
```

那么我们如何在Python中改变文本信息呢?若要改变一个字符串，需要利用合并、分片这样的工具来建立并赋值给一个新的字符串，倘若必要的话，之后将这个结果赋值给字符串最初的变量名。

```
>>> S = S + 'SPAM!'          # To change a string, make a new one
>>> S
'spamSPAM!'
>>> S = S[:4] + 'Burger' + S[-1]
>>> S
'spamBurger!'
```

第一个例子在S后面加了一个子字符串，这的确是通过合并创建了一个新的字符串并赋值给S，然而你也可以把它看作是对原字符串的“修改”。第二个例子通过分片、索引、合并将4个字符变为6个字符，正如本章稍后将介绍的一样，这一结果同样可以通过像replace这样的字符串方法来实现。

```
>>> S = 'splot'
>>> S = S.replace('pl', 'pamal')
>>> S
'spamalot'
```

像每次操作生成新的字符串的值那样，字符串方法都生成了新的字符串对象，如果愿意保留那些对象，你可以将其赋值给新的变量名。每修改一次字符串就生成一个新字符串对象并不像听起来效率那么低下 —— 记住，就像在前面的章节中我们讨论过的那样，Python在运行的过程中对不再使用的字符串对象自动进行垃圾收集（回收空间），所以新的对象重用了在前边的值所占用的空间。Python的效率往往超出了你的预期。

最后，可以通过字符串格式化表达式来创建新的文本值：

www.TopSage.com

```
>>> 'That is %d %s bird!' % (1, 'dead')     # Like C sprintf
That is 1 dead bird!
```

大家网 TopSage.com

这出现了一个功能强大的操作。下一部分将会解释它是如何运行的。

字符串格式化

Python在对字符串操作的时候定义了%二进制操作符(你可能还记得它在对数字应用时，是除法取余数的操作符)。当应用在字符串上的时候，它和C语言的`printf`函数作用相同：%提供了简单的方法对字符串的值进行格式化，这一操作取决于格式化定义的字符串。简而言之，%操作符为编写多字符串替换提供了一种简洁的方法。

格式化字符串：

1. 在%操作符的左侧放置一个需要进行格式化的字符串，这个字符串带有一个或多个嵌入的转换目标，都以%开头（例如，%d）。
2. 在%操作符右侧放置一个对象（或多个，在括号内），这些对象将会插入到左侧想让Python进行格式化字符串的（或多个）转换目标的位置上去。

例如，在上一个例子中，我们看一下前半部分，整数1插入到左边待格式化字符串%d这个位置，字符串`'dead'`插入到%s的位置。结果就得到了一个新的字符串，这个字符串就是这两个替换的结果。

从技术上来讲，字符串的格式化表达式往往是可选的 —— 通常你可以使用多次的多字符串的合并和转换达到类似的目的。然而格式化允许我们将多个步骤合并为一个简单的操作，这一功能相当强大，我们多举几个例子来看一看：

```
>>> exclamation = "Ni"
>>> "The knights who say %s!" % exclamation
'The knights who say Ni!'

>>> "%d %s %d you" % (1, 'spam', 4)
'1 spam 4 you'

>>> "%s -- %s -- %s" % (42, 3.14159, [1, 2, 3])
'42 -- 3.14159 -- [1, 2, 3]'
```

在第一个例子中，在左侧目标位置插入字符串`'Ni'`，代替%s。在第二个例子中，在目标字符串中插入三个值。需要注意的是当不止一个值待插入的时候，应该在右侧用括号把它们括起来（也就是说，把它们放到元组中去）。

第三个例子同样是插入三个值：一个整数、一个浮点数和一个表对象。但是注意到所有

www.TopSage.com

目标左侧都是%s，这就表示要把它们转换为字符串。由于任何对象都可以转换为字符串（打印时所使用的），每一个与%s一同参与操作的对象类型都可以转换代码。正因如此，除非你要做特殊的格式化，一般你只需要记得用%s这个代码来格式化表达式。

另外请记住格式化总是会返回新的字符串作为结果而不是对左侧的字符串进行修改；由于字符串是不可变的，所以只能这样操作。像前面说的一样，如果需要的话，你可以分配一个变量名来保存结果。

更高级的字符串格式化

对更高级的特定类型的格式化来说，你可以在格式化表达式中使用表7-3列出的任何一个转换代码。它们中的大部分都是C语言程序员所熟知的，因为Python字符串格式化支持C语言中所有常规的printf格式的代码（但是并不像printf那样显示结果，而是返回结果）。表中的一些格式化代码为同一类型的格式化提供了不同的选择。例如，%e和%g都可以用于浮点数的格式化。

表7-3：字符串格式化代码

代码	意义
%s	字符串（或任何对象）
%r	s，但使用repr，而不是str
%c	字符
%d	十进制（整数）
%i	整数
%u	无号（整数）
%o	八进位整数
%x	十六进制整数
%X	x，但打印大写
%e	浮点指数
%E	e，但打印大写
%f	浮点十进制
%g	浮点e或f
%G	浮点E或f
%%	常量%

事实上，在格式化字符串中，表达式左侧的转换目标支持多种转换操作，这些操作自有一套相当严谨的语法。转换目标的通用结构看上去是这样的：

```
%[(name)][flags][width][.precision]code
```

表7-3中的字符码出现在目标字符串的尾部，在%和字符码之间，你可以进行以下的任何操作：放置一个字典的键；罗列出左对齐（-）、正负号（+）和补零（0）的标志位；给出数字的整体长度和小数点后的位数等。

有关格式化目标的语法在Python的标准手册中都有完整的介绍，不过在这里，我们还是针对一般的用法举几个例子。下面这个例子首先是对整数进行默认格式化，随后进行了6位的左对齐格式化，最后进行了6位补零的格式化。

```
>>> x = 1234
>>> res = "integers: ...%d...%-6d...%06d" % (x, x, x)
>>> res
'integers: ...1234...1234  ...001234'
```

%e，%f和%g格式对浮点数的表示方法有所不同，正如下面的交互模式下所显示的那样：

```
>>> x = 1.23456789
>>> x
1.2345678899999999

>>> '%e | %f | %g' % (x, x, x)
'1.234568e+000 | 1.234568 | 1.23457'
```

对浮点数来讲，通过指定左对齐、补零、正负号、数字位数和小数点后的位数，你可以得到各种各样的格式化结果。对于较简单的任务来说，你可以通过利用简单的格式化表达式进行字符串转换或者我们早先提到的str内置函数来完成。

```
>>> '%-6.2f | %05.2f | %+06.1f' % (x, x, x)
'1.23   | 01.23 | +001.2'

>>> "%s" % x, str(x)
('1.23456789', '1.23456789')
```

基于字典的字符串格式化

字符串的格式化同时也允许左边的转换目标来引用右边字典中的键来提取对应的值。本书还没有详细介绍过字典，那么在这里举一个例子来说明其基本原理：

```
>>> "%(n)d %(x)s" % {"n":1, "x":"spam"}
'1 spam'
```

上例中，格式化字符串里（n）和（x）引用了右边字典中的键，并提取他们相应的值。生成类似HTML或XML的程序往往利用这一技术。你可以建立一个数值字典，并利用一个基于键的引用的格式化表达式一次性替换它们。

www.TopSage.com

大家网
TopSage.com

```
>>> reply = """
Greetings...
Hello %(name)s!
Your age squared is %(age)s
"""
>>> values = {'name': 'Bob', 'age': 40}
>>> print reply % values

Greetings...
Hello Bob!
Your age squared is 40
```

这样的小技巧也常与内置函数vars联起来一同使用，这个函数返回的字典包含了所有在本函数调用时存在的变量。

```
>>> food = 'spam'
>>> age = 40
>>> vars()
{'food': 'spam', 'age': 40, ...many more... }
```

当字典用在一个格式化操作的右边时，它会让格式化字符串通过变量名来访问变量（也就是说，通过字典中的键）：

```
>>> "%(age)d %(food)s" % vars()
'40 spam'
```

我们将在第8章更深入地学习字典。你还可以在第5章参考几个使用格式化目标代码%x和%o来转换十六进制和八进制的字符串的例子。

字符串方法

除表达式运算符之外，字符串还提供了一系列的方法去实现更复杂的文本处理任务。方法就是与特定的对象相关联在一起的函数。从技术的角度来讲，它们附属于对象的属性，而这些属性不过是些可调用函数罢了。在Python中，对不同的对象类型有不同的方法。拿字符串方法来说，它们仅限于字符串对象。

更详细一点，函数也就是代码包，方法调用同时进行了两次操作（一次获取属性和一次函数调用）。

属性读取

具有*object.attribute*格式的表达式可以理解为“读取*object*对象的属性*attribute*的值”。

www.TopSage.com

函数调用表达式

具有函数（参数）格式的表达式意味着“调用函数代码，传递零或者更多用逗号隔开的参数对象，最后返回函数的返回值”。

将两者合并可以让我们调用一个对象方法。方法调用表达式对象，方法（参数）从左至右运行，也就是说Python首先读取对象方法，然后调用它，传递参数。如果一个方法计算出一个结果，它将会作为整个方法调用表达式的结果被返回。

通过本书这一部分的介绍可知，绝大多数对象都有可调用的方法，而且所有对象都可以通过同样的方法调用的语法来访问。为了调用对象的方法，必须确保这个对象是存在的。让我们再通过几个例子来看看该怎样做。

字符串方法实例：修改字符串

表7-4总结了内置的字符串方法的调用方式（确定你已经通过Python的标准手册得到最新的列表，或者在交互模式下对不确认的字符串方法运行help函数）。表中的字符串方法适用于执行更高级的操作。例如，分解和连接、大小写转换、内容检测和子字符串搜索。

表7-4：字符串方法调用

```
S.capitalize()
S.center(width)
S.count(sub [, start [, end]])
S.encode([encoding [,errors]])
S.endswith(suffix [, start [, end]])
S.expandtabs([tabsize])
S.find(sub [, start [, end]])
S.index(sub [, start [, end]])
S.isalnum()
S.isalpha()
S.isdigit()
S.islower()
S.isspace()
S.istitle()
S.isupper()
S.join(seq)
S.ljust(width)
S.lower()
S.lstrip()
S.replace(old, new [, maxsplit])
S.rfind(sub [,start [,end]])
S.rindex(sub [, start [, end]])
S.rjust(width)
S.rstrip()
S.split([sep [,maxsplit]])
S.splitlines([keepends])
S.startswith(prefix [, start [, end]])
S.strip()
S.swapcase()
S.title()
S.translate(table [, delchars])
S.upper()
```


www.TopSage.com

现在让我们通过一些代码对使用中最常用的方法进行说明，并顺着这一方向对Python的文本处理的基础知识做一些阐述。我们已经知道，由于字符串是不可变的，所以不能在原处直接对其进行修改。为了在已存在的字符串中创建新的文本值，我们可以通过分片和合并这样的操作来建立新的字符串。举个例子，为了替换一个字符串中的两个字符，你可以用这样的代码来完成。

```
>>> S = 'spammy'
>>> S = S[:3] + 'xx' + S[5:]
>>> S
'spaxxy'
```

但是，如果仅为了替换一个子字符串的话，那么可以使用字符串的replace方法来实现。

```
>>> S = 'spammy'
>>> S = S.replace('mm', 'xx')
>>> S
'spaxxy'
```

replace方法比这一段代码所表现的更具有普遍性。它的第一个参数是原始子字符串（任意长度），替换原始子字符串的字符串（任意长度），之后进行全局搜索并替换：

```
>>> 'aa$bb$cc$dd'.replace('$', 'SPAM')
'aaSPAMbbSPAMccSPAMdd'
```

鉴于这样的角色，replace可以当作实现模板（例如，一定格式的信件）替换的工具来使用。注意到这次我们简单地打印了结果而不是赋值给一个变量名——只有当你想要保留结果为以后使用的时候才需要将它们赋值给变量名。

如果需要在任何偏移时都能替换一个固定长度的字符串，可以再做一次替换，或者使用字符串方法find搜索的子字符，之后使用分片：

```
>>> S = 'xxxxSPAMxxxxSPAMxxxx'
>>> where = S.find('SPAM')              # Search for position
>>> where                               # Occurs at offset 4
4
>>> S = S[:where] + 'EGGS' + S[(where+4):]
>>> S
'xxxxEGGSxxxxSPAMxxxx'
```

find方法返回在子字符串出现处的偏移（默认从前向后开始搜索）或者未找到时返回-1。另一种方式是用replace并插入第三个参数来限制只做一次代换：

```
>>> S = 'xxxxSPAMxxxxSPAMxxxx'
>>> S.replace('SPAM', 'EGGS')           # Replace all
'xxxxEGGSxxxxEGGSxxxx'
```


www.TopSage.com

```
>>> S.replace('SPAM', 'EGGS', 1)        # Replace one
'xxxxEGGSxxxxSPAMxxxx'
```

注意replace每次返回一个新的字符串对象。由于字符串是不可变的，因此每一种方法并不是真正在原处修改了字符串，尽管“replace”就是“替换”的意思！

合并操作和replace方法每次运行会产生新的字符串对象，实际上利用它们去修改字符串是一个潜在的缺陷。如果你不得不对一个超长字符串进行许多的修改，为了优化脚本的性能，可能需要将字符串转换为一个支持原处修改的对象。

```
>>> S = 'spammy'
>>> L = list(S)
>>> L
['s', 'p', 'a', 'm', 'm', 'y']
```

内置的list函数（或一个对象构造函数调用）以任意序列中的每一个元素的元素创立一个新的列表——在这个例子中，它将字符串“打散”做为一个列表。一旦字符串以这样的形式出现，你无需每次修改后进行拷贝就可以对其进行多次修改：

```
>>> L[3] = 'x'                          # Works for lists, not strings
>>> L[4] = 'x'
>>> L
['s', 'p', 'a', 'x', 'x', 'y']
```

修改之后，如果你需要将其变回一个字符串（例如，写入一个文件时），可以用字符串方法join将列表“合成”一个字符串：

```
>>> S = ''.join(L)
>>> S
'spaxxy'
```

可能第一眼看上去连接方法有些陌生。因为它是用于字符串的方法（并非用于列表），是通过设定的分隔符来调用。join将列表字符串连在一起，并用分隔符隔开。这个例子中使用一个空的字符串分隔符将列表转换为字符串。对更一般的情况来说，对任何字符串分隔符和字符串列表都会是这样的结果：

```
>>> 'SPAM'.join(['eggs', 'sausage', 'ham', 'toast'])
'eggsSPAMsausageSPAMhamSPAMtoast'
```

字符串方法实例：文本解析

另外一个字符串方法的常规角色是以简单的文本解析的形式出现的——也就是分析结构并提取子串。为了提取固定偏移的子串，我们可以利用分片技术：

www.TopSage.com

```
>>> line = 'aaa bbb ccc'
>>> col1 = line[0:3]
>>> col3 = line[8:]
>>> col1
'aaa'
>>> col3
'ccc'
```

在这里，那组数据出现在固定偏移处，因此有可能通过分片从原始字符串分出来。这一技术可以被认为是解析，只要你所需要的数据组件有固定的偏移。如果不是这样，而是有些分割符分开了数据组件，你就能够通过使用split提取出这些组件。在字符串中，数据出现在任意位置，这种方法都能够工作。

```
>>> line = 'aaa bbb  ccc'
>>> cols = line.split()
>>> cols
['aaa', 'bbb', 'ccc']
```

字符串的split方法将一个字符串分割为一个子字符串的列表，以分隔符字符串为标准。在上一个例子中，我们没有传递分隔符，所以默认的分隔符为空格 —— 这个字符串被一个或多个的空格、制表符或者换行符分成多个组，之后我们得到了一个最终子字符串的列表。在其他的应用中，可以使用更多的实际的分隔符分割数据。下面这个例子使用逗号分割（因此有时分解）一个字符串，这个字符串是使用某一数据库工具返回的由逗号分隔开的数据：

```
>>> line = 'bob,hacker,40'
>>> line.split(',')
['bob', 'hacker', '40']
```

分隔符也可以比单个字符更长，比如：

```
>>> line = "i'mSPAMaSPAMlumberjack"
>>> line.split("SPAM")
["i'm", 'a'; 'lumberjack']
```

尽管使用分片或split方法做数据解析的潜力有限，但是这两种方法运行都很快，并且能够胜任日常的基本字符串提取操作。

实际应用中的其他常见字符串方法

其他的字符串方法都有更专注的角色 —— 例如，为了清除每行末尾的空白，执行大小写转换，测试内容以及检测末尾的子字符串：

```
>>> line = "The knights who sy Ni!\n"
>>> line.rstrip()
```


www.TopSage.com

```
'The knights who sy Ni!'
>>> line.upper()
'THE KNIGHTS WHO SY NI!\n'
>>> line.isalpha()
False
>>> line.endswith('Ni!\n')
True
```

与字符串方法相比，其他的技术有时也能够达到相同的结果——例如，成员操作符in能够用来检测一个子字符串是否存在，并且length和分片操作能够用来做字符串末尾的检测：

```
>>> line
'The knights who sy Ni!\n'

>>> line.find('Ni') != -1          # Search via method call or expression
True
>>> 'Ni' in line
True

>>> sub = 'Ni!\n'
>>> line.endswith(sub)             # End test via method call or slice
True
>>> line[-len(sub):] == sub
True
```

因为字符串有很多方法可以使用，我们这里不会每一个都介绍。你会在本书后边见到一些其他的字符串例子，但是需要了解更多细节可以在Python库手册以及其他的文件中寻求帮助，或者在交互模式下自己动手来做些简单的实验。

注意没有字符串方法支持模式——对于基于模式的文本处理，必须使用Python的re标准库模块，这个模块是一个在第4章介绍过的高级工具，但是超出了本书的范围。尽管由于这方面的限制，字符串方法有时与re模块的工具比较起来，还是有运行速度方面的优势的。

最初的字符串模块

Python的字符串方法历史有些曲折。大约Python出现的前十年，只提供一个标准库模块，名为string，其中包含的函数大约相当于目前的字符串对象方法集。为了满足用户需求，在Python 2.0时，这些函数就变成字符串对象的方法了。然而，因为有那么多人写了如此多的代码都依赖最初的string模块，所以为了保持向后的兼容性它就一直被保留下来了。

如今，你应该只使用字符串方法，而不是最初的string模块。事实上，最初的模块调

www.TopSage.com

用形式打算在Python 3.0中删除，就在本书出版后不久。然而，因为你还是会在较旧的Python代码中看见这个模块，因此在这里要简单的看一下。

这种历史问题的结果就是，在Python 2.5中，从技术上来说，有两种方式可以启用高级的字符串运算：调用对象方法或者调用string模块函数，把对象当成自变量传递进去。例如，已知变量X为字符串对象，并调用对象方法：

```
X.method(arguments)
```

这样通常等效于通过string模块调用相同的运算（如果已导入该模块）：

```
string.method(X, arguments)
```

这里是一个在实际应用中方法机制的例子：

```
>>> S = 'a+b+c+'
>>> x = S.replace('+', 'spam')
>>> x
'aspambspamcspam'
```

要通过string模块获取相同的操作，你需要导入该模块（至少在进程中要导入一次）并传入对象：

```
>>> import string
>>> y = string.replace(S, '+', 'spam')
>>> y
'aspambspamcspam'
```

因为模块的实现方法是长久的标准，而且因为字符串是大多数程序的核心组件，你可能会在以后创建Python程序中看到两种调用模式。

不过，现在你应该使用方法调用而不是陈旧的模块调用。这样做有很好的理由，除了模块调用已经打算在3.0版移除以外，还有一个内容，那就是模块调用法需要你导入string模块（方法不需要导入）。此外，模块让调用在输入时需要多打几个字符（当你以import加载模块而不是使用from时）。最后，模块运行速度比方法慢（当前的模块把大多数调用对应到了方法，因此会导致占用额外的调用时间）。

最初的string模块可能会保留在Python 3.0中，因为它包含了其他的工具，包括预定义的字符串常数，以及模板对象系统（此处省略的一个高级工具，请参考Python库手册以了解模板对象的细节）。不过，除非你真的想在3.0问世时再修改代码，否则，就应该放弃基本字符串运算调用。

www.TopSage.com

通常意义下的类型分类

到现在，我们已经探索了第一个Python的集合对象 —— 字符串，让我们暂停一下，来定义一些通常意义下的类型观念，这些观念对于今后我们学到的大多数类型来说都有效。对于内置类型，对于相同分类的类型有很多操作工作起来都是一样的，所以绝大多数的概念我们只需要定义一次。到现在只检查了数字和字符串，但是由于它们代表了Python中的三大类型分类中的两个，对于其他的类型你已经了解了足够多的内容。

同样分类的类型共享其操作集合

正像我们所学习的那样，字符串是不可改变的序列：它们不能在原处进行改变（不可变部分），并且它们是位置相关排序好的集合，可以通过偏移量读取（序列部分）。现在，本书中我们学到的所有的序列都可以使用本章中对于字符串的序列操作 —— 合并、索引、迭代等。更正式的来说，在Python中有三个类型（以及操作）的分类：

数字

支持加法和乘法等。

序列

支持索引、分片和合并，等。

映射

支持通过键的索引等。

我们还没有深入地学习映射（字典将会在下一章讨论），但是我们遇到的其他类型将大部分都一致。例如，对于任意的序列对象X和Y：

- X+Y将会创建一个包含了两个操作对象内容的新的序列对象。
- X*N将会创建一个包含操作对象X内容N份拷贝的新的序列对象。

换句话说，这些操作工作起来对于任意一种序列对象都一样，包括字符串、列表、元组以及用户定义的对象类型。唯一的区别就是，你得到的新的最终对象是根据操作对象X和Y来决定的 —— 如果你合并的是列表，那么你将得到一个新的列表而不是字符串。索引、分片以及其他的序列操作对于所有的序列来说都是同样有效的，对象的类型将会告诉Python去执行什么样的任务。

可变类型能够在原处修改

不可变的分类是需要特别注意的约束，尽管对于新用户来说，还是有可能在这里犯糊涂

www.TopSage.com

的。如果一个对象是不可变的，你就不能在原处修改它的值；如果你这么做的话Python将会报错。替代的办法就是，你必须运行代码来创建一个新的对象来包含这个新的值。一般来说，不可变类型有某种完整性，保证这个对象不会被程序的其他部分改变。对于新人来说如果不知道这有什么要紧的话，请参考第6章关于共享对象引用的讨论。

本章小结

本章，我们深入学习了字符串这个对象类型。我们学习了如何编写字符串常量，探索了字符串操作，包括序列表达式、字符串格式化以及字符串方法调用。在这个过程中，我们深入学习了各种概念，例如，分片、方法调用、三重引号字符串。我们也定义了一些关于变量类型的核心概念。例如，序列，它会共享整个操作的集合。 在下一章，我们将会继续学习类型，集中学习Python中最常见的集合对象 —— 列表和字典。就像你发现的那样，这里你学到的很多东西在那两种类型中也能使用。下面通过章节测试来复习一下本章学到的内容。

大家网
TopSage.com

本章习题

1. 字符串find方法能用于搜索列表吗？
2. 字符串切片表达式能用于列表吗？
3. 你如何将字符转成其ASCII码？你如何反向转换，从整数转换成字符？
4. 在Python中，怎么修改字符串？
5. 已知字符串S的值为"s,pa,m"，提出两种从中间抽取两个字符的方式。
6. 字符串"a\nb\x1f\000d"之中有多少字符？
7. 你为什么要使用string模块，而不使用字符串方法调用？

习题解答

1. 不行，因为方法是类型特定的。也就是说，只能用于单一数据类型上。不过，表达式是通用的，能用于多种类型上。从这一点来说，例如，in成员关系表达式就有类似的效果，能够用于搜索字符串和列表。
2. 可以。和方法不同的是，表达式是通用的，可用于多种类型。就这一点来说，切片表达式其实是序列运算，可用于任何类型的序列对象上，包括字符串、列表以及元组。唯一的差别就是当你对列表进行切片时，你得到的是新列表。
3. 内置的ord(S)函数可将单个字符的字符串转换成整数字符编码。chr(I)则是从整数代码转换回字符串。
4. 字符串无法被修改，字符串是不可变的。尽管这样，你可以通过合并、切片运算、执行格式化表达式、方法调用（例如，replace）创建新的字符串，再将结果赋值给最初的变量名，从而达到相似的效果。
5. 你可以使用S[2:4]对字符串进行切片，或者使用S.split(',')[1]以逗号分割字符串，再进行索引运算。通过交互模式亲自试验，看一下结果。
6. 6个。字符串"a\nb\x1f\000d"包含一些字节a、新行（\n）、b、二进制值31（十六进制转义\x1f）、二进制值0（八进制转义\000）以及d。把字符串传给内置的len函数可以验证它，然后印出每个字符的ord结果，从而查看实际的字节值。参考表7-2的细节。
7. 如今，不应该使用string模块，而应该使用字符串对象方法调用。string模块将被弃用，Python 3.0将会移除它。使用string模块的唯一原因就是可以使用其他工具，例如，预定义的常数以及高级的模板对象。

www.TopSage.com

第8章

列表与字典

这一章里我们将要介绍列表和字典，这两个对象类型都是其他对象的集合。这两种类型几乎是Python所有脚本的主要工作组件。我们将看到这两种类型都相当灵活：它们都可以在原处进行修改，也可以按需求增长或缩短，而且可以包含任何种类的对象或者被嵌套。借助这些类型，可以在脚本中创建并处理任意的复杂的信息结构。

列表

我们的python内置对象之旅的下一站是列表（list），列表是Python中最具灵活性的有序集合对象类型。与字符串不同的是，列表可以包含任何种类的对象：数字、字符串甚至其他列表。同样，与字符串不同，列表都是可变对象，它们都支持在原处修改的操作，可以通过指定的偏移值和分片、列表方法调用、删除语句等方法来实现。

Python中的列表可以完成大多数集合体数据结构的工作，而这些在稍底层一些的语言中（例如，C语言）你不得不手工去实现。让我们快速地看一下它们的主要属性，Python列表是：

任意对象的有序集合

从功能上看，列表就是收集其他对象的地方，你可以把它们看作组。同时列表所包含的每一项都保持了从左到右的位置顺序（也就是说，它们是序列）。

通过偏移读取

就像字符串一样，你可以通过列表对象的偏移对其进行索引，从而读取对象的某一部分内容。由于列表的每一项都是有序的，那么你也可以执行诸如分片和合并之类的任务。

可变长度、异构以及任意嵌套

与字符串不同的是，列表可以实地的增长或者缩短（长度可变），并且可以包含任

何类型的对象而不仅仅是包含有单个字符的字符串（异构）。因为列表能够包含其他复杂的对象，又能够支持任意的嵌套。所以你可以创建列表的子列表的子列表等等。

属于可变序列的分类

就类型分类而言，列表支持在原处的修改（它们是可变的），也可以响应所有针对字符串序列的操作，例如,索引、分片以及合并。实际上，序列操作在列表与字符串中的工作方式相同。唯一的区别是：当合并和分片这样的操作当应用于列表时，返回新的列表而不是新的字符串。然而列表是可变的。因此它们也支持字符串不支持的其他操作（例如，删除和索引赋值操作，它们都是在原处修改列表）。

对象引用数组

从技术上来讲，Python列表包含了零或多个其他对象的引用。列表也许会让你想起指针（地址）数组，从Python的列表中读取一个项的速度与索引一个C语言数组差不多。实际上，在标准Python解释器内部，列表就是C数组而不是链接结构。我们曾在第6章学过，每当用到引用时，Python总是会将这个引用指向一个对象，所以程序只需处理对象的操作。当把一个对象赋给一个数据结构元素或变量名时，Python总是会存储对象的引用，而不是对象的一个拷贝（除非明确要求保存拷贝）。

表8-1总结了常见的和具有代表性的列表对象操作。和往常一样，为了得到更全面的信息，可以查阅Python的标准库手册，或者运行`help(list)`或`dir(list)`查看`list`方法的完整列表清单，你可以传入一个真正的列表，或者列表数据类型的名称——`list`这个单词。

表8-1：常用列表常量和操作

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

`L2[i]`
`L3[i][j]`
`L2[i:j]`
`len(L2)`
	索引，索引的索引，分片，求长度

`L1 + L2`	合并
`L2 * 3`	重复

www.TopSage.com

表8-1：常用列表常量和操作（续）

操作	解释
`for x in L2`	迭代，成员关系
`3 in L2`	
`L2.append(4)`	方法：增长，排序，搜索，插入，反转等等
`L2.extend([5,6,7])`	
`L2.sort()`	
`L2.index(1)`	
`L2.insert(I, X)`	
`L2.reverse()`	
`del L2[k]`	裁剪
`del L2[i:j]`	
`L2.pop()`	
`L2.remove(2)`	
`L2[i:j] = []`	
`L2[i] = 1`	索引赋值，分片赋值
`L2[i:j] = [4,5,6]`	
`range(4)`	生成整数列表/元组
`xrange(0, 4)`	
`L4 = [x**2 for x in range(5)]`	列表解析（见13章和17章）

当作为常量表达式编写时，列表会被写成系列对象（实际上是返回对象的表达式），这些对象被括在方括号中并用逗号隔开。例如，表8-1的第2行将变量L2赋给一个四项的列表。嵌套的列表写成一串嵌套的方括号（第3行）。空列表就是一对内部为空的方括号（第1行）（注1）。

表8-1中的大多数操作看上去应该很熟悉，因为它们都与我们先前在字符串上使用的序列操作相同，例如，索引、合并和迭代等。列表除了支持在原处的修改操作(删除项、赋值给索引和分片等)之外，还可以进行特定的列表方法调用（它们可完成排序、反转操作以及在结尾添加元素等任务），列表可以使用这些工具来进行修改操作是因为列表是可变的对象类型。

注1：在列表处理程序中，不会看到有很多列表写成这样。比较常见的代码是处理动态建立的列表（运行时）。实际上，虽然精通常量语法也很重要，但Python中多数数据结构的建立都是在运行时执行程序代码的。

www.TopSage.com

实际应用中的列表

理解列表最好的方法可能还是要在实践中体会它们是如何运作的，让我们再看几个简单的解释器交互的例子来说明表8-1中的操作。

基本列表操作

对列表进行“+”和“*”操作与字符串基本一致，两个操作的意思也是合并和重复，只不过结果是一个新的列表，而不是一个字符串。实际上，列表能够支持上一章里我们用于字符串的所有一般的序列操作。

```
% python
>>> len([1, 2, 3])                    # Length
3
>>> [1, 2, 3] + [4, 5, 6]             # Concatenation
[1, 2, 3, 4, 5, 6]
>>> ['Ni!'] * 4                       # Repetition
['Ni!', 'Ni!', 'Ni!', 'Ni!']
>>> 3 in [1, 2, 3]                    # Membership
True
>>> for x in [1, 2, 3]: print x,      # Iteration
...
1 2 3
```

由于与语法相关，我们将在第13章对迭代以及内置`range`函数进行介绍。简而言之，循环从左到右对序列中的每一项依次执行，每一项执行一条或多条语句。关于表8-1最后一行的列表解析部分的内容将在第13章介绍，并在第17章将做进一步介绍。正如第4章所介绍的一样，列表解析是创建列表的一种方法，它以单一步骤对序列中每一项应用一个表达式来建立列表。

尽管列表的“+”操作和字符串中的一样，然而值得重视的是“+”两边必须是相同类型的序列，否则运行时会出现类型错误。例如，不能将一个列表和一个字符串合并到一起，除非你先把列表转换为字符串（使用诸如反引号、`str`或者%格式这样的工具），或者把字符串转换为列表（列表内置函数能完成这一转换）。

```
>>> str([1, 2]) + "34"                # Same as "[1, 2]" + "34"
'[1, 2]34'
>>> [1, 2] + list("34")               # Same as [1, 2] + ["3", "4"]
[1, 2, '3', '4']
```

索引、分片和矩阵

由于列表都是序列，对于列表而言，索引和分片操作与字符串中的操作基本相同。然而

www.TopSage.com

对列表进行索引的结果就是你指定的偏移处的对象（不管是什么类型），而对列表进行分片时往往返回一个新的列表。

```
>>> L = ['spam', 'Spam', 'SPAM!']
>>> L[2]                          # Offsets start at zero
'SPAM!'
>>> L[-2]                         # Negative: count from the right
'Spam'
>>> L[1:]                         # Slicing fetches sections
['Spam', 'SPAM!']
```

注意：由于可以在列表中嵌套列表（和其他对象类型），有时需要将几次索引操作连在一起使用来深入到数据结构中去。举个例子，最简单的办法之一是将其表示为矩阵（多维数组），在Python中相当于嵌套了子列表的列表。在这里我们看一个基于列表的3×3的二维数组：

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

如果使用一次索引，会得到一整行（实际上，也就是嵌套的子列表），如果使用两次索引，你将会得到某一行里的其中一项：

```
>>> matrix[1]
[4, 5, 6]
>>> matrix[1][1]
5
>>> matrix[2][0]
7
>>> matrix = [[1, 2, 3],
...           [4, 5, 6],
...           [7, 8, 9]]
>>> matrix[1][1]
5
```

在之前的交互模式下可以让列表自然地横跨很多行，因为列表是以一对方括号括起来的（本书下一部分会对语法做更多的介绍）。本章稍后会介绍基于字典的矩阵表达形式。就高性能数值运算的工作来说，第5章所提到的NumPy扩展提供了处理矩阵的其他方式。

原处修改列表

由于列表是可变的，它们支持实地改变列表对象的操作。也就是说，本节中的操作都可以直接修改列表对象，而不会像字符串那样强迫你建立一个新的拷贝。因为Python只处理对象引用，所以需要将原处修改一个对象与生成一个新对象区分开来，这在第6章已经讨论过了，如果你在原处修改一个对象时，可能同时会影响一个以上指向它的引用。

大家网
TopSage.com

索引与分片的赋值

当使用列表的时候，可以将它赋值给一个特定项（偏移）或整个片段（分片）来改变它的内容。

```
>>> L = ['spam', 'Spam', 'SPAM!']
>>> L[1] = 'eggs'                    # Index assignment
>>> L
['spam', 'eggs', 'SPAM!']
>>> L[0:2] = ['eat', 'more']         # Slice assignment: delete+insert
>>> L                                # Replaces items 0,1
['eat', 'more', 'SPAM!']
```

索引和分片的赋值都是原地修改，它们对列表进行直接修改，而不是生成一个新的列表作为结果。Python中的索引赋值与C及大多其他语言极为相似 —— Python用一个新值取代指定偏移的对象引用。

上一个例子的最后一个操作是分片赋值，它仅仅用一步操作就能够将列表的整个片段替换掉。因为这可能有点复杂，所以分片赋值最好分成两步来理解：

1. **删除**。删除等号左边指定的分片。
2. **插入**。将包含在等号右边对象中的片段插入旧分片被删除的位置（注2）。

事实并不是这样的，但这有助于你理解为什么插入元素的数目不需要与删除的数目相匹配。例如，已知一个列表L的值为[1,2,3]，赋值操作L[1:2]=[4,5]会把L修改成列表[1,4,5,3]。Python会先删除2（单项分片），然后在删除2的地方插入4和5。这也解释了为什么L[1:2]=[]实际上是删除操作 —— Python删除分片（位于偏移为1的项）之后什么也不插入。

实际上，分片赋值是一次性代换整个片段或“栏”。因为被赋值的序列长度不一定要与被赋值的分片的长度相匹配，所以分片赋值能够用来替换（覆盖）、增长（插入）、缩短（删除）主列表。这是功能强大的操作，然而坦率地说，它也是在实践当中不常见的操作。Python通常还有更简单直接的方式实现替换、插入以及删除（例如，合并、insert、pop以及remove列表方法），那些才是Python程序员实际上比较喜欢用的。

列表方法调用

与字符串相同，Python列表对象也支持特定类型方法调用

注2：当赋值的值与分片重迭时，就需要详细的分析：例如，L[2:5]=L[3:6]是可行的，这是因为要被插入的值会在左侧删除发生前被取出。

www.TopSage.com

```
>>> L.append('please')              # Append method call
>>> L
['eat', 'more', 'SPAM!', 'please']
>>> L.sort()                        # Sort list items ('S' < 'e')
>>> L
['SPAM!', 'eat', 'more', 'please']
```

在第7章里我们介绍过方法。简而言之，方法就是附属于特定对象的函数（实际上是引用函数的属性）。方法提供特定类型的工具。例如，我们这里介绍的列表方法只适用于列表。

可能最常用的列表方法是append，它能够简单地将一个单项（对象引用）加至列表末端。与合并不同的是，append允许转入单一对象而不是列表。L.append(X)与L+[X]的结果类似，不同的是，前者会原地修改L，而后者会生成新的列表（注3）。另一个常见方法是sort，它原地对列表进行排序。sort是使用Python标准的比较检验作为默认值（在这里指字符串比较），而且以递增的顺序进行排序。你也可以将比较函数传给sort。

注意： 在Python 2.5和更早的版本中，不同类型的对象也是可进行比较的（例如，字符串和列表）。Python在不同类型之中都定义了一个固定的顺序，也许看上去并不那么赏心悦目，但它们是明确的。也就是说，这一次序是根据涉及的类型名称而定的。例如，所有整数都小于所有字符串，因为"int"比"str"小。比较运算时绝不会自动转换类型，除非是在比较数值类型对象。

在Python 3.0中，这一点可能就不一样了 —— 不同类型的比较不再依赖于固定的跨类型之间的排序，而是默认引发一个异常。因为排序是在内部进行比较，这意味着[1, 2, 'spam'].sort()在Python 2.x中是能行得通的，但是像在Python 3.0中则会发生异常。想要了解更多细节，请参考3.0版本发行报告的相关说明。

注意：要当心append和sort原处的修改相关的列表对象，而结果并没有返回列表（从技术上来讲，两者返回的值皆为None）。如果编辑类似L=L.append(X)的语句，将不会得到L修改后的值（实际上，会失去整个列表的引用）；当使用append和sort之类的属性时，对象的修改有点像副作用，所以没有理由再重新赋值。

与字符串相同，列表有其他方法可执行其他特定的操作。例如，reverse可原地的反转列表，extend和pop方法分别能够在末端插入多个元素、删除一个元素：

注3： 和“+”合并不同的是，append不用产生新对象，所以执行起来通常比较快。你也可以用聪明的分片运算模拟append：L[len(L):]=[X]就与L.append(X)一样，而L[:0]=[X]就好像在列表前端附加那样。两者都会删除空分片，并插入X，快速地在适当之处修改L，就像append一样。

www.TopSage.com

```
>>> L = [1, 2]
>>> L.extend([3,4,5])                    # Append multiple items
>>> L
[1, 2, 3, 4, 5]
>>> L.pop()                              # Delete and return last item
5
>>> L
[1, 2, 3, 4]
>>> L.reverse()                          # In-place reversal
>>> L
[4, 3, 2, 1]
```

在某些类型的应用程序中，往往会把这里用到的列表pop方法和append方法联用，来实现快速的后进先出（LIFO，last-in-first-out）堆栈结构。列表的末端作为堆栈的顶端：

```
>>> L = []
>>> L.append(1)                          # Push onto stack
>>> L.append(2)
>>> L
[1, 2]
>>> L.pop()                              # Pop off stack
2
>>> L
[1]
```

尽管在这里并没有显现出来，但pop方法也能够接受某一个即将删除并返回的元素的偏移（默认值为最后一个元素），这一偏移是可选的。其他列表方法可以通过值删除（remove）某元素，在偏移处插入（insert）某元素，查找某元素的偏移（index）等。可以参考其他说明源文件或者多练习这些调用进行更深入的学习。

其他常见列表操作

由于列表是可变的，你可以用del语句在原处删除某项或某片段：

```
>>> L
['SPAM!', 'eat', 'more', 'please']
>>> del L[0]                             # Delete one item
>>> L
['eat', 'more', 'please']
>>> del L[1:]                            # Delete an entire section
>>> L                                    # Same as L[1:] = []
['eat']
```

因为分片赋值是删除外加插入操作，也可以通过将空列表赋值给分片来删除列表片段（L[i:j]=[]）。Python会删除左侧的分片，然后什么也不插入。另一方面，将空列表赋值给一个索引只会在指定的位置存储空列表的引用，而不是删除。

```
>>> L = ['Already', 'got', 'one']
>>> L[1:] = []
```


www.TopSage.com

```
>>> L
['Already']
>>> L[0] = []
>>> L
[[]]
```

大家网 TopSage.com

虽然刚才讨论的所有操作都很典型，但是还有其他列表方法和操作并没有在这里列出（包括插入和搜索方法）。为了得到更全面的最新类型工具清单，你应该时常参考Python手册、Python的dir和help函数（我们在第4章首次提到过），《Python Pocket Reference》（O'Reilly）以及其他在序言中所提到的参考书籍。

我们这里讨论的原处修改操作都只适用于可变对象：无论你怎样绞尽脑汁，都是不能用在字符串上的（或者是下一章我们将要讨论的元组）。可变性是每个对象类型的固有属性。

字典

除了列表以外，字典（dictionary）也许是Python之中最灵活的内置数据结构类型。如果把列表看作是有序的对象集合，那么就可以把字典当成是无序的集合。它们主要的差别在于：字典当中的元素是通过键来存取的，而不是通过偏移存取。

作为内置类型，字典可以取代许多搜索算法和数据结构，而这些你可能在较低级语言中不得不通过手工来实现。对字典进行索引是非常快速的搜索操作。字典有时也能执行其他语言中的记录、符号表的功能，可以表示稀疏（多数为空）数据结构等等。Python字典的主要属性如下。

通过键而不是偏移量来读取

字典有时又被称为关联数组（associative array）或者是哈希表（hash）。它们通过键将一系列值联系起来，这样就可以使用键从字典中取出一项。就像列表那样，同样可以使用索引操作从字典中获取内容。但是索引采取键的形式，而不是相对偏移。

任意对象的无序集合

与列表不同，保存在字典中的项并没有特定的顺序。实际上，Python将各项从左到右随机排序，以便快速查找。键提供了字典中项的象征性（而非物理性的）位置。

可变长、异构、任意嵌套

与列表相似，字典可以在原处增长或是缩短（无需生成一份拷贝）。它们可以包含任何类型的对象，而且它们支持任意深度的嵌套（可以包含列表和其他的字典等）。

www.TopSage.com

属于可变映射类型

通过给索引赋值，字典可以在原处修改（可变），但不支持我们用于字符串和列表中的序列操作。实际上，因为字典是无序集合，而根据固定顺序进行操作是行不通的（例如，合并和分片操作）。相反，字典是唯一内置的映射类型（键映射到值的对象）。

对象引用表(哈希表)

如果说列表是支持位置读取的对象引用数组，那么字典就是支持键读取的无序对象引用表。从本质上讲，字典是作为哈希表（支持快速检索的数据结构）来实现的，一开始很小，并根据要求而增长。此外，Python采用最优化的哈希算法来寻找键，因此搜索是很快速的。和列表一样，字典存储的是对象引用（不是拷贝）。

表8-2总结了一些最为普通并具有代表性的字典操作［查看库手册或者运行dir（dict）或是help（dict）可以得到完整的清单，类型名为dict］。当写成常量表达式时，字典以一系列“键:值（key:value）”对形式写出的，用逗号隔开，用大括号括起来（注4）。一个空字典就是一对空的大括号，而字典可以作为另一个字典（列表、元组）中的某一个值被嵌套。

表8-2：常见字典常量和操作

操作	解释
`D1 = {}`	空字典
`D2 = {'spam': 2, 'eggs': 3}`	两项目字典
`D3 = {'food': {'ham': 1, 'egg': 2}}`	嵌套
`D2['eggs']` `D3['food']['ham']`	以键进行索引运算

`D2.has_key('eggs')`
`'eggs' in D2`
`D2.keys()`
`D2.values()`
`D2.copy()`
`D2.get(key, default)`
`D2.update(D1)`
`D2.pop(key)`
	方法：成员关系测试、键列表、值列表、复制、默认、合并、删除等

注4：与列表相似，不会经常用常量创建字典。不过，列表和字典增长的方式不同。下一节我们将会看到，我们常常通过在运行时对新的键赋值来建立字典。这种方法对列表是不起作用的（列表通过append语句增长）。

www.TopSage.com

表8-2：常见字典常量和操作（续）

操作	解释
`len(D1)`	长度（储存的元素的数目）
`D2[key] = 42` `del D2[key]`	新增/修改键，删除键

`D4 = dict.fromkeys(['a', 'b'])`
`D5 = dict(zip(keyslist, valslist))`
`D6 = dict(name='Bob', age=42)`
	其他构造技术

实际应用中的字典

如表8-2所示，字典通过键进行索引，被嵌套的字典项是由一系列索引（方括号中的键）表示的。当Python创建字典时，会按照任意所选从左到右的顺序来存储项。为了取回一个值，需要提供相应的键。让我们回过头来看一看解释器，感受一下表8-2列出的一些字典的操作。

字典的基本操作

通常情况下，创建字典并且通过键来存储、访问其中的某项：

```
% python
>>> d2 = {'spam': 2, 'ham': 1, 'eggs': 3}    # Make a dictionary
>>> d2['spam']                               # Fetch a value by key
2
>>> d2                                       # Order is scrambled
{'eggs': 3, 'ham': 1, 'spam': 2}
```

在这里，字典被赋值给一个变量d2，键'spam'的值为整数2等。像利用偏移索引列表一样，使用相同的方括号句法，用键对字典进行索引操作，只不过这里意味着用键来读取，而并不是用位置来读取。

注意这个例子的结尾，字典内键由左至右的次序几乎总是和原先输入的顺序不同。这样设计的目的是为了快速执行键查找（也就是哈希查找），键需在内存中随机设定。这就是为什么假设固定从左至右的顺序操作（例如，分片和合并）不适用于字典的原因所在。只能用键进行取值，而不是用位置来取值。

内置len函数也可用于字典，它能够返回储存在字典里的元素的数目，或者说是其keys列表的长度，这二者是等价的。字典的has_key方法以及in成员关系操作符提供了键存在与否的测试方法，keys方法则能够返回字典中所有的键，将它们收集在一个列表中。

www.TopSage.com

后者对于按顺序处理字典是非常有用的，但是你不应该依赖keys列表的次序。然而，因为keys结果是一个普通列表，如果次序要紧，随时都可以进行排序：

TopSage.com

```
>>> len(d2)                    # Number of entries in dictionary
3
>>> d2.has_key('ham')          # Key membership test
True
>>> 'ham' in d2                # Key membership test alternative
True
>>> d2.keys()                  # Create a new list of my keys
['eggs', 'ham', 'spam']
```

注意此列表中的第三个表达式。之前我们提到过，用于字符串和列表的in成员关系测试同样适用于字典。它能够检查某个键是否储存在字典内，如同上一行的has_key方法调用。从技术上来讲，这样做能行得通是因为字典定义了单步遍历keys列表的迭代器。其他类型也提供了反映它们共同用法的迭代器。例如，文件有逐行读取的迭代器。我们将会在第17章和第24章进一步讨论迭代器。

在本章以及本书的后面部分，你将会学到另外两种可选的创建字典的方式，如表8-2结尾所给的示范：我们可以把键/值元组或关键词函数自变量的列表打包传给新的dict调用（其实就是类型创建函数）。我们会在第16章探讨关键词自变量。我们也将会在第13章讨论zip函数；这是仅用单一调用就能完成通过键和值列表来创建字典的一种方式。如果你无法在程序代码中默认键和值，那么可以通过动态的方式将它们建立成列表，之后再将它们打包起来。

原处修改字典

让我们继续介绍交互模式会话。与列表相同，字典也是可变的，因此可以在原处对它们进行修改、扩展以及缩短而不需要生成新字典。简单地给一个键赋值就可以改变或者生成元素。del语句在这里也适用。它删除作为索引的键相关联的元素。此外，注意这个例子中字典所嵌套的列表（键'ham'的值）。Python中，所有集合数据类型都可以彼此任意嵌套。

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

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

>>> d2['brunch'] = 'Bacon'                        # Add new entry
>>> d2
{'brunch': 'Bacon', 'spam': 2, 'ham': ['grill', 'bake', 'fry']}
```


www.TopSage.com

与列表相同，向字典中已存在的索引赋值会改变与索引相关联的值。然而，与列表不同的是，每当对新字典键进行赋值（之前没有被赋值的键），就会在字典内生成一个新的元素，就像前一个例子里对'brunch'所做的那样。在列表中情况不同，因为Python会将超出列表末尾的偏移视为越界并报错。要想扩充列表，你需要使用append方法或分片赋值来实现。

其他字典方法

字典方法提供了多种工具。例如，字典values和items方法分别返回字典的值列表和(key,value)对元组。

```
>>> d2 = {'spam': 2, 'ham': 1, 'eggs': 3}
>>> d2.values()
 [3, 1, 2]
>>> d2.items()
 [('eggs', 3), ('ham', 1), ('spam', 2)]
```

这类列表在需要逐项遍历字典项的循环中是很有用的。读取不存在的键往往都会出错，然而键不存在时通过get方法能够返回默认值（None或者用户定义的默认值）。这是在当键不存在时为了避免missing-key错误而填入默认值的一个简单方法：

```
>>> d2.get('spam')                    # A key that is there
2
>>> d2.get('toast')                   # A key that is missing
None
>>> d2.get('toast', 88)
88
```

字典的update方法有点类似于合并，它把一个字典的键和值合并到另一个，盲目地覆盖相同键的值。

```
>>> d2
{'eggs': 3, 'ham': 1, 'spam': 2}
>>> d3 = {'toast':4, 'muffin':5}
>>> d2.update(d3)
>>> d2
{'toast': 4, 'muffin': 5, 'eggs': 3, 'ham': 1, 'spam': 2}
```

最后，字典pop方法能够从字典中删除一个键并返回它的值。这类似于列表的pop方法，只不过删除的是一个键而不是一个可选的位置。

```
# pop a dictionary by key

>>> d2
{'toast': 4, 'muffin': 5, 'eggs': 3, 'ham': 1, 'spam': 2}
```

```
>>> d2.pop('muffin')
5
>>> d2.pop('toast')                    # Delete and return from a key
4
>>> d2
{'eggs': 3, 'ham': 1, 'spam': 2}

# pop a list by position

>>> L = ['aa', 'bb', 'cc', 'dd']
>>> L.pop()                            # Delete and return from the end
'dd'
>>> L
['aa', 'bb', 'cc']
>>> L.pop(1)                           # Delete from a specific position
'bb'
>>> L
['aa', 'cc']
```

字典也能够提供copy方法。我们会在下一章对其进行讨论，因为它是避免相同字典共享引用潜在的副作用的一种方式。实际上，字典还有很多其他方法并没有在表8-2中列出，可以参考Python库手册，或者其他说明文件查看完整清单。

语言表

我们来看一个更实际的字典的例子。下面的例子能够生成一张表格，把程序语言名称（键）映射到它们的创造者（值）。你可以通过语言名称索引来读取语言创造者的名字。

```
>>> table = {'Python':  'Guido van Rossum',
...          'Perl':    'Larry Wall',
...          'Tcl':     'John Ousterhout' }
...
>>> language = 'Python'
>>> creator  = table[language]
>>> creator
'Guido van Rossum'

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

最后的命令使用了for循环，但我们还没有讨论过它的细节。如果你对for循环不熟悉，这里简单解释为这一命令只不过是通过迭代表中的每一个键，再打印出用tab分开的键及其值的表单而已。在第13章我们将对for循环做更多的介绍。

www.TopSage.com

因为字典并非序列，你无法像字符串和列表那样直接通过一个for语句迭代它们。但是，如果你需要遍历各项是很容易的：调用字典的keys方法，返回经过排序之后所有键的列表，再用for循环进行迭代。需要时，你可以像上面的代码中所做的那样在for循环中从键到值进行索引。

实际上，Python也能够让你遍历字典的键列表，而并不用在多数for循环中调用keys方法。就任何字典D而言，写成for key in D:和写成完整的key in D.keys():效果是一样的。这其实只是先前所提到的迭代器能够允许in成员关系操作符用于字典的另一个实例（有关迭代器更多内容我们稍后将进行介绍）。

字典用法注意事项

一旦你熟练掌握了字典，它将成为相当简单的工具，但是在使用字典时，有几点需要注意：

- 序列运算无效。字典是映射机制，不是序列。因为字典元素间没有顺序的概念，类似级联（有序合并）和分片（提取相邻片段）这样的运算是不能用的。实际上，如果你试着这样做，Python会在你的程序运行时报错。
- 对新索引赋值会添加项。当你编写字典常量时（此时的键是嵌套于常量本身的），或者向现有字典对象的新键赋值时，都会生成键。最终的结果是一样的。
- 键不一定总是字符串。我们的例子中都使用字符串作为键，但任何不可变对象（也就是说，不是列表）也是可以的。例如，你可以用整数作为键，这样让字典看起来很像列表（至少进行索引时很像）。元组偶尔允许合并键值时也可以用作字典键。只要它有合适的协议方法，类实例对象（我们将在本书第6部分进行讨论）也可以用作键。大体上来讲，它需要告诉Python其值不变，否则作为固定键将会毫无用处。

使用字典模拟灵活的列表

前面的表中的最后一点非常重要，我们应举些例子来说明一下。当使用列表的时候，对在列表末尾外的偏移赋值是非法的。

```
>>> L = []
>>> L[99] = 'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list assignment index out of range
```

虽然你可以使用重复按你所需预先分配足够大的列表（例如，[0]*100），但你也可以

www.TopSage.com

用字典来做类似的事情，这样就不需要这样的空间分配了。使用整数键时，字典可以效仿列表在偏移赋值时增长。

```
>>> D = {}
>>> D[99] = 'spam'
>>> D[99]
'spam'
>>> D
{99: 'spam'}
```

在这里，看起来似乎D是个有100项的列表，但其实是一个有单个元素的字典，键99的值是字符串'spam'。你可以像列表那样用偏移访问这一结构，但你不需要为将来可能会用到的会被赋值的所有位置都分配空间。像这样使用时，字典很像更具灵活性的列表。

字典用于稀疏数据结构

类似地，字典键也常用于实现稀疏数据结构。例如，多维数组中只有少数位置上有存储的值：

```
>>> Matrix = {}
>>> Matrix[(2, 3, 4)] = 88
>>> Matrix[(7, 8, 9)] = 99
>>>
>>> X = 2; Y = 3; Z = 4                    # ; separates statements
>>> Matrix[(X, Y, Z)]
88
>>> Matrix
{(2, 3, 4): 88, (7, 8, 9): 99}
```

在这里，我们用字典表示一个三维数组，这个数组中只有两个位置(2,3,4)和(7,8,9)有值，其他位置都为空。键是元组，它们记录非空元素的坐标。我们并不是分配一个庞大而几乎为空的三维矩阵，而是用一个简单的两个元素的字典。通过这一方式读取空元素时，会触发键不存在的异常，因为这些元素实质上并没有被存储。

```
>>> Matrix[(2,3,6)]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: (2, 3, 6)
```

避免missing-key错误

读取不存在的键的错误在稀疏矩阵中很常见，然而可能并不希望程序因为这一错误被关闭。在这里至少有三种方式可以让我们填入默认值而不会出现这样的错误提示：你可以在if语句中预先对键进行测试，也可以使用try语句明确地捕获并修复这一异常，还可以用我们前面介绍的get方法为不存在的键提供一个默认值。

www.TopSage.com

```
>>> if Matrix.has_key((2,3,6)):      # Check for key before fetch
...     print Matrix[(2,3,6)]
... else:
...     print 0
...
0
>>> try:
...     print Matrix[(2,3,6)]        # Try to index
... except KeyError:                 # Catch and recover
...     print 0
...
0
>>> Matrix.get((2,3,4), 0)           # Exists; fetch and return
88
>>> Matrix.get((2,3,6), 0)           # Doesn't exist; use default arg
0
```

从编程的需要方面来说，get方法是这三者中最简捷的。我们将在本书稍后部分详细介绍if和try语句。

使用字典作为“记录”

就像本书所介绍的，字典在Python中能够扮演多种角色。一般来说，字典可以取代搜索数据结构（因为用键进行索引是一种搜索操作），并且可以表示多种结构化信息的类型。例如，字典是在程序范围中多种描述某一项属性的方法之一。也就是说，它们能够扮演与其他语言中“记录”和“结构”相同的角色。

这是一个随时间通过向新键赋值来填写字典的例子。

```
>>> rec = {}
>>> rec['name'] = 'mel'
>>> rec['age']  = 45
>>> rec['job']  = 'trainer/writer'
>>>
>>> print rec['name']
mel
```

特别是在嵌套的时候，Python的内建数据类型可以很轻松地表达结构化信息。这个例子再一次使用字典来捕获对象的属性，但它是一次性写好（并没有对每个键分别赋值），而且嵌套了一个列表和一个字典来表达结构化属性的值。

```
>>> mel = {'name': 'Mark',
...        'jobs': ['trainer', 'writer'],
...        'web':  'www.rmi.net/~lutz',
...        'home': {'state': 'CO', 'zip':80513}}
...
```

当读取嵌套对象的元素时，只要简单地把索引操作串起来就可以了。

www.TopSage.com

大家网
TopSage.com

```
>>> mel['name']
'Mark'
>>> mel['jobs']
['trainer', 'writer']
>>> mel['jobs'][1]
'writer'
>>> mel['home']['zip']
80513
```

创建字典的其他方法

注意因为字典非常有用，先后出现了更多创建字典的方法。例如，在Python 2.3和后续的版本中，下面最后两次对`dict`创建函数的调用与它们上方文字和键赋值的形式具有相同的效果。

```
{'name': 'mel', 'age': 45}          # Traditional literal expression

D = {}                              # Assign by keys dynamically
D['name'] = 'mel'
D['age']  = 45

dict(name='mel', age=45)            # Keyword argument form

dict([('name', 'mel'), ('age', 45)])  # Key/value tuples form
```

这四种形式都会建立相同的两键字典：

- 如果你可以事先拼出整个字典，那么第一种是很方便的。
- 如果你需要一次动态地建立字典的一个字段，第二种比较合适。
- 第三种关键字形式所需的代码比常量少，但是键必须都是字符串才行。
- 如果你需要在程序运行时把键和值逐步建成序列，那么最后一种形式比较有用。

如表8-2的末端部分所示，最后一种形式通常也会与`zip`函数一起使用，把程序运行时动态获取的键和值列表合并在一起（例如，分析数据文件字段）。

如果所有键的值都相同，你也可以用这个特殊的形式对字典进行初始化 —— 简单地传入一个键列表，以及所有键的初始值（默认值为空）：

```
>>> dict.fromkeys(['a', 'b'], 0)
{'a': 0, 'b': 0}
```

虽然这个时候可以在Python中仅仅依靠常量和键赋值而蒙混过关，但是当开始将字典用在实际的、灵活性的以及动态的Python程序中时，你可能会发现所有这些字典创建形式的用处。

www.TopSage.com

为什么要在意字典接口

除了作为一种能够在程序中通过键存储信息的简便方法之外，有些Python的扩展程序也提供了外表类似并且实际工作都和字典一样的接口。例如，Python的DBM接口通过键来获取文件，它看上去特别像一个已经打开的字典。字符串的读取都使用键索引：

```
import anydbm
file = anydbm.open("filename")  # Link to file
file['key'] = 'data'            # Store data by key
data = file['key']              # Fetch data by key
\
```

稍后，我们将会看到如果你把刚才那段程序代码中的anydbm换成shelve（shelve是通过键来访问的Python持久对象的数据库），那么你也可以用这种方式储存整个Python对象。就互联网而言，Python的CGI脚本支持的一个接口看上去也跟字典类似。一个对cgi.FieldStorage范围的调用会产生一个类似字典的对象，在客户端网页上每个输入字段都有一项：

```
import cgi
form = cgi.FieldStorage()       # Parse form data
if form.has_key('name'):
    showReply('Hello, ' + form['name'].value)
```

所有这些（以及字典）都是映射的例子。一旦你学习了字典接口，你就会发现字典接口适用于Python各种内置工具。

本章小结

本章我们探讨了列表和字典类型 —— 这可能是在Python程序中所见到并使用的两种最常见、最具有灵活性而且功能最为强大的集合体类型。本章介绍了列表类型支持任意对象以位置排序的集合体，而且可以任意嵌套，按需要增长和缩短。字典类型也是如此，不过它是以键来储存元素而不是位置，并且不会保持元素之间任何可靠的由左至右的顺序。列表和字典都是可变的，所以它们支持各种不适用于字符串的原处修改操作。例如，列表可以通过append调用来进行增长，而字典则是通过赋值给新键的方法来实现。

下一章我们将要介绍元组和文件同时结束我们的深入核心对象类型之旅。随后我们将会介绍编写处理对象的逻辑语句，让我们向着编写完整程序的目标再前进一步。在那之前，还是让我们先做一些习题来巩固一下本章所学的知识。

www.TopSage.com

本章习题

1. 举出两种方式来创建内含五个整数零的列表。
2. 举出两种方式来创建一个字典，有两个键'a'和'b'，而每个键相关联的值都是0。
3. 举出四种在原处修改列表对象的运算。
4. 举出四种在原处修改字典对象的运算。

习题解答

1. 像[0, 0, 0, 0, 0]这种常量表达式以及[0] * 5这种重复表达式，都会创建五个零的列表。在实际应用中，你可能会通过循环创建这种列表。一开始是空列表，在每次迭代中附加0：L.append(0)。列表解析（[0 for i in range(5)]）在这里也可以用，但是，这种方法比较费功夫。
2. 像{'a': 0, 'b': 0}这种常量表达式，或者像D = {}, D['a'] = 0, D['b'] = 0这种一系列的赋值运算，都会创建所需要的字典。你也可以使用较新并且编写起来更简单的关键字形式dict(a=0, b=0)，或者更有弹性的dict([('a', 0), ('b', 0)])键/值序列形式。或者因为所有键的值都相同，你也可以使用特殊形式dict.fromkeys(['a', 'b'], 0)。
3. append和extend方法可让在原处增长列表，sort和reverse方法可以对列表进行排序或者翻转，insert方法可以在一个偏移值处插入一个元素，remove和pop方法会按照值和位置从列表中删除元素，del语句会删除一个元素或分片，而索引以及分片赋值语句则会取代一个元素或整个片段。本题可任意挑选其中的四个。
4. 字典的修改主要是赋值新的键或已存在的键，从而建立或修改键在表中的项目。此外，del语句会删除一个键的元素，字典update方法会把一个字典合并到另一个字典的适当的地方，而D.pop(key)则会移除一个键并返回它的值。字典也有其他更古怪的方法可以在原处进行修改，但在这一章中没有列出，例如，setdefault。查看参考资源来了解更多的细节。

www.TopSage.com

大学网
TopSage.com

第9章 元组、文件及其他

这一章我们将要探讨元组（无法修改的其他对象的集合）以及文件（计算机上外部文件的接口），这也将在我们深入了解Python的核心对象类型的部分画上一个圆满的句号。我们将会看到，元组是一个相关的简单对象，其实它的大部分执行操作在介绍字符串和列表的时候我们就已经学过了。文件对象是处理文件常用的并且全能的工具。对文件的基本概念将通过本书后续章节出现有关文件的例子时进行进一步的补充。

这一章我们也将总结本书介绍过的所有核心对象类型的共有属性来作为本书这一部分的结论：关于相等、比较、对象复制等的概念。我们也会简要探讨Python工具箱中其他的对象类型。正如你会看到的，尽管我们已经涵盖了所有主要的内置类型，但Python中的对象远比我们到目前为止所介绍的要多得多。最后，我们将会看到几个对象类型的常见错误，探讨一些能够巩固所学知识的练习题来结束这一章的内容。

元组

本书介绍的最后一个Python集合类型是元组（tuple）。元组由简单的对象组构成。元组与列表非常类似，只不过元组不能在原处修改（它们是不可变的），并且通常写成圆括号（而不是方括号）中的一系列项。虽然元组不支持任何方法调用，但元组具有列表的大多数属性。让我们快速了解一下它的属性。具体如下所示。

任意对象的有序集合

与字符串和列表类似，元组是一个位置有序的对象的集合（也就是其内容维持从左到右的顺序）。与列表相同，可以嵌入到任何类别的对象中。

通过偏移存取

同字符串、列表一样，在元组中的元素通过偏移（而不是键）来访问。它们支持所有基于偏移的操作。例如，索引和分片。

www.TopSage.com

属于不可变序列类型

类似于字符串，元组是不可变的，它们不支持应用在列表中任何原处修改操作。与字符串和列表类似，元组是序列，它们支持许多同样的操作。

固定长度、异构、任意嵌套

因为元组是不可变的，在不生成一个拷贝的情况下不能增长或缩短。另一方面，元组可以包含其他的复合对象（例如，列表、字典和其他元组等），因此支持嵌套。

对象引用的数组

与列表相似，元组最好被认为是对象引用的数组。元组存储指向其他对象的存取点（引用），并且对元组进行索引操作的速度相对较快。

表9-1中列出了常见的元组操作。元组编写为一系列对象（从技术上来讲，是生成对象的表达式），用逗号隔开，并且用圆括号括起来。一个空元组就是一对内空的括号。

表9-1：常见元组实字和运算

运算	解释
`()`	空元组
`t1 = (0,)`	单个元素的元组（非表达式）
`t2 = (0, 'Ni', 1.2, 3)`	四个元素的元组
`t2 = 0, 'Ni', 1.2, 3`	另一个四元素的元组（与前列相同）
`t3 = ('abc', ('def', 'ghi'))`	嵌套元组

`t1[i]`
`t3[i][j]`
`t1[i:j]`
`len(t1)`
	索引、索引的索引、分片、长度

`t1 + t2` `t2 * 3`	合并、重复
`for x in t` `'spam' in t2`	迭代、成员关系

实际应用中的元组

和往常一样，让我们开始以交互式会话的方式探索实际应用中的元组。在表9-1中，元组没有方法（例如，append调用在这是不可用的）。然而，元组的确支持字符串和列表的一般序列操作。

www.TopSage.com

```
>>> (1, 2) + (3, 4)                    # Concatenation
(1, 2, 3, 4)

>>> (1, 2) * 4                         # Repetition
(1, 2, 1, 2, 1, 2, 1, 2)

>>> T = (1, 2, 3, 4)                   # Indexing, slicing
>>> T[0], T[1:3]
(1, (2, 3))
```

元组的特殊语法：逗号和圆括号

表9-1中的第二和第四项应该做进一步说明。因为圆括号也可以把表达式括起来（参考第5章），如果圆括号里的单一对象是元组对象而不是一个简单的表达式时，需要对Python进行特别说明。如果确实想得到一个元组，只要在这一单个元素的之后、关闭圆括号之前加一个逗号就可以了。

```
>>> x = (40)                           # An integer
>>> x
40
>>> y = (40,)                          # A tuple containing an integer
>>> y
(40,)
```

作为特殊情况，在不会引起语法冲突的情况下，Python允许忽略元组的圆括号。例如，表中第四行简单列出了四个由逗号隔开的项。在赋值语句中，即使没有圆括号，Python也能够识别出这是一个元组。

现在，有些人会告诉你，元组中一定要使用圆括号，而有些人会告诉你不要用（其他人都有自己的生活，并不会告诉你该怎样对待元组）。仅当元组作为文字传给函数调用（圆括号很重要）以及当元组在`print`语句中列出（逗号很重要）的特殊情况时，圆括号才是必不可少的。

对初学者而言，最好的建议是一直使用圆括号，这可能会比弄明白什么时候可以省略圆括号要更简单一些。许多程序员也发现圆括号有助于增加脚本的可读性，因为这样可以使元组更加明确，尽管你的使用经验可能会有所不同。

转换以及不可变性

除了常量语法不同以外，元组的操作（表9-1最后三行）和字符串及列表是一致的。值得注意的区别在于“+”、“*”以及分片操作应用于元组时将返回新元组，并且元组不提供字符串、列表和字典中的方法。例如，如果你想对元组进行排序，通常先得将它转换为列表才能获得使用排序方法调用的权限，并将它变为一个可变对象。

www.TopSage.com

大家网 TopSage.com

```
>>> T = ('cc', 'aa', 'dd', 'bb')
>>> tmp = list(T)                    # Make a list from a tuple's items
>>> tmp.sort()                       # Sort the list
>>> tmp
['aa', 'bb', 'cc', 'dd']
>>> T = tuple(tmp)                   # Make a tuple from the list's items
>>> T
('aa', 'bb', 'cc', 'dd')
```

这里的列表和元组内置函数被用来转换为列表，之后返回为一个元组。实际上，这两个调用都会生成新的对象，但结果就像是转换。

列表解析 （list comprehension）也可用于元组的转换。例如，下面这个由元组生成的列表，过程中将每一项都加上20：

```
>>> T = (1, 2, 3, 4, 5)
>>> L = [x + 20 for x in T]
>>> L
[21, 22, 23, 24, 25]
```

列表解析是名副其实的序列操作 —— 它们总会创建新的列表，但也可以用于遍历包括元组、字符串以及其他列表在内的任何序列对象。我们将会看到，列表解析甚至可以用在某些并非实际储存的序列之上 —— 任何可遍历的对象都可以，包括可自动逐行读取的文件。

同样，注意元组的不可变性只适用于元组本身顶层而并非其内容。例如，元组内部的列表是可以像往常那样修改的。

```
>>> T = (1, [2, 3], 4)

>>> T[1] = 'spam'                    # This fails: can't change tuple itself
TypeError: object doesn't support item assignment

>>> T[1][0] = 'spam'                 # This works: can change mutables inside
>>> T
(1, ['spam', 3], 4)
```

对多数程序而言，这种单层深度的不可变性对一般元组角色来说已经足够了。这碰巧把我们引入下一节的内容。

为什么有了列表还要元组

初学者学习元组的时候，这似乎总是第一个出现的问题 —— 既然已经有列表了，为什么还需要元组？其中的某些原因可能是历史性的。Python的创造者接受过数学训练，并提到过把元组看作是简单的对象组合，把列表看成是随时间改变的数据结构。

www.TopSage.com

然而，最佳答案似乎是元组的不可变性提供了某种完整性。这样你可以确保元组在程序中不会被另一个引用修改，而列表就没有这样的保证了。因此，元组的角色类似于其他语言中的“常数”声明，然而这种常数概念在Python中是与对象相结合的，而不是变量。

元组也可以用在列表无法使用的地方。例如，作为字典键（参考第8章稀疏矩阵的例子）。一些内置操作可能也要求或暗示要使用元组而不是列表。凭经验来说，列表是定序集合的选择工具，可能需要进行修改，而元组能够处理其他固定关系的情况。

文件

想必大多数读者都熟悉文件的概念，也就是计算机中由操作系统管理的具有名字的存储区域。我们最后要讲的这个主要内置对象类型提供了一种可以存取Python程序内部文件的方法。

简而言之，内置open函数会创建一个Python文件对象，可以作为计算机上的一个文件链接。在调用open之后，你可以通过调用返回文件对象的方法来读写相关外部文件。内置名file是open的同义词，而且从技术上来讲，文件可以通过调用open或file来打开。open通常比file更常用，因为file几乎都是为面向对象程序设计量身打造的（本书稍后部分将予以说明）。

与我们目前见过的类型相比，文件对象多少有些不寻常。它们不是数字、序列也不是映射。相反，文件对象只是常见文件处理任务输出模块。多数文件方法都与执行外部文件相关的文件对象的输入和输出有关，但其他文件方法可让查找文件中的新位置、刷新输出缓存等。

打开文件

表9-2总结了常见的文件操作。为了打开一个文件，程序会调用内置open函数，首先是外部名，接着是处理模式。模式典型地用字符串'r'代表为输入打开文件（默认值），'w'代表为输出生成并打开文件，'a'代表为在文件尾部追加内容而打开文件。还有其他的选择我们在这里就先省略了。就更高级的角色而言，在模式字符串尾部加上b可以进行二进制数据处理（行末转换被关闭了），而加上“+”意味着同时为输入和输出打开文件（也就是说，我们可以对相同对象进行读写）。

www.TopSage.com

表9-2：常见文件运算

操作	解释
`output = open('/tmp/spam', 'w')`	创建输出文件（'w'是指写入）
`input = open('data', 'r')`	创建输入文件（'r'是指读写）
`input = open('data')`	与上一行相同（'r'是默认值）
`aString = input.read()`	把整个文件读进单一字符串
`aString = input.read(N)`	读取之后的N个字节（一或多个）到一个字符串
`aString = input.readline()`	读取下一行（包括行末标示符）到一个字符串
`aList = input.readlines()`	读取整个文件到字符串列表
`output.write(aString)`	写入字节字符串到文件
`output.writelines(aList)`	把列表内所有字符串写入文件
`output.close()`	手动关闭（当文件收集完成时会替你关闭文件）
`output.flush()`	把输出缓冲区刷到硬盘中，但不关闭文件
`anyFile.seek(N)`	修改文件位置到偏移量N处以便进行下一个操作

要打开的两个参数必须都是Python的字符串，第三个是可选参数，它能够用来控制输出缓存：传入"0"意味着输出无缓存（写入方法调用时立即传给外部文件）。外部文件名参量可能包含平台特定的以及绝对或相对目录路径前缀。没有目录路径时，文件假定存在当前的工作目录中（也就是脚本运行的地方）。

使用文件

一旦存在一个文件对象，就可以调用其方法来读写相关的外部文件。在任何情况下，Python程序中的文本文件采用字符串的形式。读取文件时会返回字符串形式的文本，文本作为字符串传递给write方法。读写方法有许多种，表9-2列出的方法是最常用的。

虽然表中的读写方法都是常用的，但是要记住，现在从文本文件读取文字行的最佳方式是根本不要读取该文件。我们在第13章将会看到，文件也有个迭代器会自动地在for循环、列表解析或者其他迭代语句中对文件进行逐行读取。

注意从文件读取的数据回到脚本时是一个字符串。所以如果字符串不是你所需的，就得将其转换成其他类型的Python对象。同样，与print语句不同的是，当你把数据写入文件时，Python不会自动把对象转换为字符串 —— 你必须传递一个已经格式化的字符串。因此，我们之前见过的处理文件时可以来回转换字符串和数字的工具迟早会派上用场（例如，int、float、str以及字符串格式表达式）。Python也包括一些高级标准库工

www.TopSage.com

具，它用来处理一般对象的存储（例如，pickle模块）以及处理文件中打包的二进制数据（例如，struct模块）。本章稍后部分我们会对这二者进行介绍。

调用文件close方法将会终止对外部文件的连接。我们在第6章曾经介绍过，在Python中，一旦对象不再被引用，则这个对象的内存空间就会自动被收回。当文件对象被收回的时候，如果需要的话，Python也会自动关闭该文件。这就意味着你不需要总是手动去关闭文件，尤其是对于不会运行很长时间的简单脚本。另一方面，手动关闭调用没有任何坏处，而且在大型程序中通常是个很不错的习惯。此外，严格地讲，文件的这个收集完成后自动关闭的特性不是语言定义的一部分，而且可能随时间而改变。因此，手动进行文件close方法调用是我们需要养成的一个好习惯。

实际应用中的文件

让我们看一个能够说明文件处理原理的简单例子。首先为输出而打开一个新文件，写入一个字符串（以行终止符\n结束），之后关闭文件。接下来，我们将会在输入模式下再一次打开同一文件，读取该行。注意第二个readline调用返回一个空字符串。这是Python文件方法告诉我们已经到达文件底部（文件的空行是含有新行符的字符串，而不是空字符串）。以下是完整的交互模式会话。

```
>>> myfile = open('myfile', 'w')              # Open for output (creates file)
>>> myfile.write('hello text file\n')         # Write a line of text
>>> myfile.close()                            # Flush output buffers to disk

>>> myfile = open('myfile')                   # Open for input: 'r' is default
>>> myfile.readline()                         # Read the line back
'hello text file\n'
>>> myfile.readline()                         # Empty string: end of file
''
```

这个例子把一行文本写成字符串，包括行终止符\n。写入方法不会为我们添加行终止符，所以程序必须包含它来严格地终止行（否则，下次写入时会简单地延长文件的当前行）。

在文件中存储并解析Python对象

现在，让我们创建一个较大的文件。下面这个例子将把多种Python对象占用多行写入文本文件。需要注意的是，我们必须使用转换工具把对象转成字符串。注意文件数据在脚本中一定是字符串，而写入方法不会自动地替我们做任何向字符串格式转换的工作。

```
>>> X, Y, Z = 43, 44, 45                      # Native Python objects
>>> S = 'Spam'                                # Must be strings to store in file
```


www.TopSage.com

```
>>> D = {'a': 1, 'b': 2}
>>> L = [1, 2, 3]
>>>
>>> F = open('datafile.txt', 'w')          # Create output file
>>> F.write(S + '\n')                      # Terminate lines with \n
>>> F.write('%s,%s,%s\n' % (X, Y, Z))      # Convert numbers to strings
>>> F.write(str(L) + '$' + str(D) + '\n')  # Convert and separate with $
>>> F.close()
```

一旦我们创建了文件就可以通过打开和读取字符串来查看文件的内容（单步操作）。注意：交互模式的回显给出了正确的字节内容，而print语句则会解释内嵌行终止符来给予用户满意的结果：

```
>>> bytes = open('datafile.txt').read()        # Raw bytes display
>>> bytes
"Spam\n43,44,45\n[1, 2, 3]${'a': 1, 'b': 2}\n"
>>> print bytes                                # User-friendly display
Spam
43,44,45
[1, 2, 3]${'a': 1, 'b': 2}
```

现在我们不得不使用其他转换工具，把文本文件中的字符串转换成真正的Python对象。鉴于Python不会自动把字符串转换为数字或其他类型的对象，如果我们需要使用诸如索引、加法等普通对象工具，就得这么做。

```
>>> F = open('datafile.txt')                   # Open again
>>> line = F.readline()                        # Read one line
>>> line
'Spam\n'
>>> line.rstrip()                              # Remove end-of-line
'Spam'
```

对第一行来说，我们使用字符串rstrip方法去掉多余的行终止符；line[:-1]分片也可以，但是只有确定所有行都含有“\n”的时候才行（文件中最后一行有时候会没有）。到目前为止，我们读取了包含字符串的行。现在，让我们读取包含数字的下一行，并解析出（抽取出）该行中的对象：

```
>>> line = F.readline()                        # Next line from file
>>> line                                       # It's a string here
'43,44,45\n'
>>> parts = line.split(',')                    # Split (parse) on commas
>>> parts
['43', '44', '45\n']
```

我们这里使用字符串split方法，从逗号分隔符的地方将整行断开，得到的结果就是含有个别数字的子字符串列表。如果我们想对这些数字做数学运算还是得把字符串转换为整数：

www.TopSage.com

```
>>> int(parts[1])                          # Convert from string to int
44
>>> numbers = [int(P) for P in parts]      # Convert all in list at once
>>> numbers
[43, 44, 45]
```

int能够把数字字符串转换为整数对象，我们在第4章所介绍的列表解析表达式也可以一次性对列表中的每个项使用调用（你可以在本书稍后的地方找到更多关于列表解析的介绍）。注意：我们不一定非要运行rstrip来删除最后部分的“\n”，int和一些其他转换方法会忽略数字旁边的空白。

最后，要转换文件第三行所储存的列表和字典，我们可以运行eval这一内置函数，eval能够把字符串当作可执行程序代码（从技术上来讲，就是一个含有Python表达式的字符串）。

```
>>> line = F.readline()
>>> line
"[1, 2, 3]${'a': 1, 'b': 2}\n"
>>> parts = line.split('$')                # Split (parse) on $
>>> parts
['[1, 2, 3]', "{'a': 1, 'b': 2}\n"]
>>> eval(parts[0])                         # Convert to any object type
[1, 2, 3]
>>> objects = [eval(P) for P in parts]     # Do same for all in list
>>> objects
[[1, 2, 3], {'a': 1, 'b': 2}]
```

因为所有这些解析和转换的最终结果是一个普通的Python对象列表，而不是字符串。现在我们可以在脚本内应用列表和字典操作了。

用pickle储存Python的原生对象

如前面程序所示，使用eval可以把字符串转换成对象，它是一个功能强大的工具。事实上，它有时太过于强大。eval会高高兴兴地执行Python的任何表达式，甚至是有可能会删除计算机上所有文件的表达式，只要给予必要的权限。如果你真的想储存Python原生对象，但又无法信赖文件的数据来源，Python标准库pickle模块会是个理想的选择。

pickle模块是能够让我们直接在文件中存储几乎任何Python对象的高级工具，也并不要求我们把字符串转换来转换去。它就像是超级通用的数据格式化和解析工具。例如，想要在文件中储存字典，就直接用pickle来储存。

```
>>> F = open('datafile.txt', 'w')
>>> import pickle
>>> pickle.dump(D, F)                      # Pickle any object to file
>>> F.close()
```


www.TopSage.com

之后，将来想要取回字典时，只要简单地再用一次pickle进行重建就可以了：

```
>>> F = open('datafile.txt')
>>> E = pickle.load(F)                    # Load any object from file
>>> E
{'a': 1, 'b': 2}
```

我们取回等价的字典对象，没有手动断开或转换的要求。pickle模块执行所谓的对象序列化（object serialization），也就是对象和字节字符串之间的相互转换。但我们要做的工作却很少。事实上，pickle内部将字典转成字符串形式，不过这其实没什么可看的（如果我们在其他模式下使用pickle会更难）：

```
>>> open('datafile.txt').read()
"(dp0\nS'a'\np1\nI1\nsS'b'\np2\nI2\ns."
```

因为pickle能够依靠这一格式重建对象，我们不必自己手动来处理。有关pickle模块的更多内容可以参考Python标准库手册，或者在交互模式下输入pickle传给help来查阅相关信息。于此同时你也可以顺便看一看shelve模块。shelve用pickle把Python对象存放到按键访问的文件系统中，不过这并不在我们讨论的范围之内。

文件中打包二进制数据的存储与解析

在我们继续学习下面的内容之前，还有一个和文件相关的细节需要注意：有些高级应用程序也需要处理打包的二进制数据，这些数据可能是C语言程序生成的。Python的标准库中包含一个能够在这一范围起作用的工具：struct模块能够构造并解析打包的二进制数据。从某种意义上说，它是另一个数据转换工具，它能够把文件中的字符串解读为二进制数据。

例如，要生成一个打包的二进制数据文件，用'wb'（写入二进制）模式打开它，并将一个格式化字符串和几个Python对象传给struct。这里用的格式化字符串是指一个4字节整数、一个包含4个字符的字符串以及一个2位整数的数据包，所有这些都按照高位在前（big-endian）的形式（其他格式代码能够处理补位字节、浮点数等）。

```
>>> F = open('data.bin', 'wb')            # Open binary output file
>>> import struct
>>> bytes = struct.pack('>i4sh', 7, 'spam', 8)    # Make packed binary data
>>> bytes
'\x00\x00\x00\x07spam\x00\x08'
>>> F.write(bytes)                        # Write byte string
>>> F.close()
```

Python生成一个我们通常写入文件的二进制数据字符串（主要由不可打印的字符组成，这些字符以十六进制转义的格式进行打印）。要将值解析为普通Python的对象，可以简

www.TopSage.com

单地读取字符串，并使用相同格式的字符串把它解压出来就可以了。Python能够把值提取出来转换为普通的Python对象（整数和字符串）。

```
>>> F = open('data.bin', 'rb')
>>> data = F.read()                          # Get packed binary data
>>> data
'\x00\x00\x00\x07spam\x00\x08'
>>> values = struct.unpack('>i4sh', data)    # Convert to Python objects
>>> values
(7, 'spam', 8)
```

二进制数据文件是高级而又有点低层次的工具，我们在这不再讲更多的细节了。如果需要更多帮助，可以参考Python库手册或者在交互模式下将struct传给help函数查阅相关的信息。同时需要注意的是，一般来说，二进制文件处理模式'wb'和'rb'可用于处理更简单的二进制文件。例如，图片或音频文件是不需要解压他的内容的。

其他文件工具

表9-2中列出了一些额外的更高级的文件方法，还有很多并没有列出。例如，seek函数能够复位你在文件中的当前位置（下次读写将应用在该位置上），flush能够强制性地将缓存输出写入磁盘（文件总会默认进行缓存）等。

Python标准库手册及序言中所提到的参考书提供了完整的文件方法清单，想要快速地浏览一下的话，可以在交互模式下运行dir或者调用help，并输入file这个单词（文件对象类型的名称）。更多有关文件处理的例子请参考第13章“为什么要在意文件扫描”。它反映了常见文件扫描循环代码模式，所用语法我们目前还没有完全涉及到，所以还不能在这里使用。

同样，需要注意的是，虽然open函数及其返回的文件对象是Python脚本中通向外部文件的主要接口，Python工具集中还有其他类似的文件工具。os模块中的描述文件（处理整数文件，支持诸如文件锁定之类的较低级工具）也是可用的。此外，还有sockets、pipes和FIFO文件（类文件对象，用于同步进程或者通过网络进行通信）。通过键来存取的文件（通过键直接存储的不变的Python对象）被认为是“shelve”等。要想了解更多这类文件工具的信息，你可以参考更高级的Python书籍。

重访类型分类

现在我们已经看到所有实际中的Python核心内置类型，让我们再看一看它们所共有的一些属性，以此来结束我们的对象类型之旅。

表9-3根据前面介绍的类型类别，对所有类型加以分类。对象根据它所属的类别共享操作。例如，字符串、列表和元组都共享序列操作。只有可变对象（列表和字典）才可能在原地修改，但是不可以改变数字、字符串或者元组。文件只能够输出方法，所以可变性不适用于文件——文件可能在写入的时候修改，但这与Python的类别约束有所不同。

为什么要在意操作符重载

本书稍后会介绍自己实现的类的对象可以在这些分类中任意选择。例如，如果你想提供一种新的特殊序列对象，它与内置序列一致，那么就写一个类，重载索引和合并等类似的操作。

```
class MySequence:
    def __getitem__(self, index):
        # Called on self[index], others
    def __add__(self, other):
        # Called on self + other
```

你还可以选择性地实现一些原处修改操作的方法调用来生成新的可变或不可变对象（例如，self[index]=value赋值操作中调用__setitem__）。尽管这不在本书的内容范围之内，但在C语言这类外部语言中实现新的对象作为C语言的扩展程序类型也是有可能的。就此而言，填上C函数指针位，在数字、序列和映射操作设置之中做选择。

表9-3：对象分类

对象类型	分类	是否可变
数字	数值	否
字符串	序列	否
列表	序列	是
字典	对应	是
元组	序列	否
文件	扩展	N/A

对象灵活性

本书这一部分介绍了一些复杂的对象类型（组件的集合）。一般来说：

- 列表、字典和元组可以包含任何种类的对象。

www.TopSage.com

大家网
TopSage.com

- 列表、字典和元组可以任意嵌套。
- 列表和字典可以动态地扩大和缩小。

因为Python的复合对象类型支持任意结构，因此对于表达程序中复杂的信息它们是相当拿手的。例如，字典的值可以是列表，这一列表可能包含了元组，而元组可能包含了字典，依此类推。只要能够满足创建待处理数据的模型的需要，嵌套多少层都是可以的。

让我们来看一个嵌套的例子。下面的交互式会话定义了一个嵌套复合序列对象的树，如图9-1所示。要存取它的内容时，我们需要按要求串起多个索引操作。Python从左到右计算这些索引，每一步取出一个更深层嵌套对象的引用。图9-1也许是过于复杂的莫名其妙的数据结构，但它描述了一般情况下，用于存取嵌套对象的语法。

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

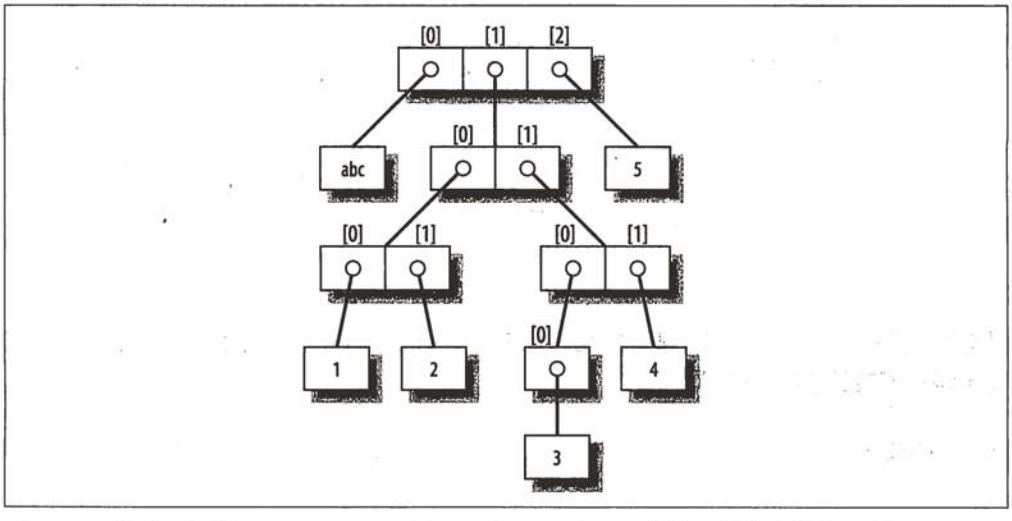


图9-1：一个由运行常量表达式[`abc', [(1, 2), ([3], 4)], 5]生成的元素偏移的嵌套对象树。从语法上来说，嵌套对象在内部被表示为对不同内存区域的引用（也就是指针）

www.TopSage.com

引用 VS 拷贝

我们在第6章曾经提到过，赋值操作总是储存对象的引用，而不是这些对象的拷贝。在实际应用中，这往往就是你想要的。不过，因为赋值操作会产生相同对象的多个引用，需要意识到在原处修改可变对象时可能会影响程序中其他地方对相同对象的其他引用，这一点很重要。如果你不想这样做，就需要明确地告诉Python复制该对象。

我们在第6章曾经研究过这种现象，但是当较大的对象参与时，就会变得更为严重。例如，下面这个例子生成一个列表并赋值为X，另一个列表赋值为L，L嵌套对列表X的引用。这一例子中还生成了一个字典D，含有另一个对列表X的引用。

```
>>> X = [1, 2, 3]
>>> L = ['a', X, 'b']          # Embed references to X's object
>>> D = {'x':X, 'y':2}
```

在这一问题上，对我们先前生成的列表有三个引用：来自名字X、来自赋值为L的列表内部以及来自赋值为D的字典内部。关系如图9-2所示。

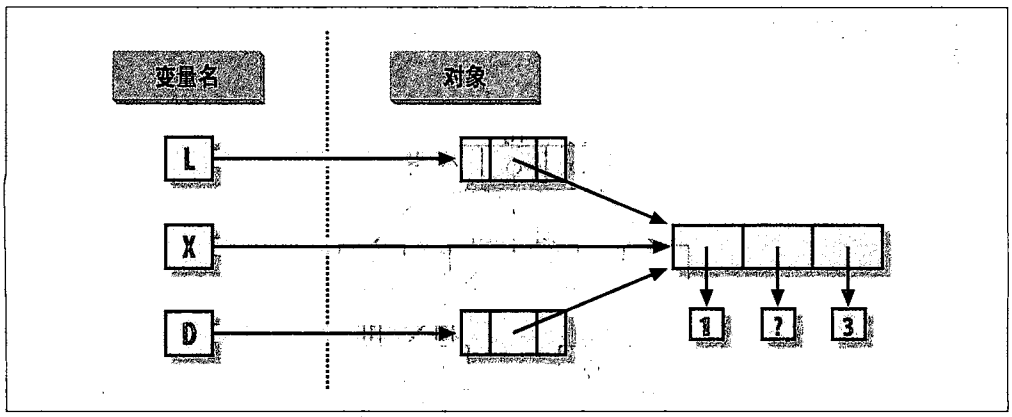


图9-2：共享对象引用：因为变量X引用的列表也在被L和D引用的对象内引用，修改X的共享列表与L和D的看起来也有所不同

由于列表是可变的，修改这三个引用中任意一个共享列表对象，也会改变另外两个引用的对象。

```
>>> X[1] = 'surprise'          # Changes all three references!
>>> L
['a', [1, 'surprise', 3], 'b']
>>> D
{'x': [1, 'surprise', 3], 'y': 2}
```

引用是其他语言中指针的更高级模拟。虽然你不能抓住引用本身，但在不止一个地方储

www.TopSage.com

存相同的引用（变量、列表等）是可能的。这是一大特点：你可以在程序范围内任何地方传递大型对象而不必在途中产生拷贝。然而，如果你的确需要拷贝，那么可以明确要求。

- 没有限制条件的分片表达式（L[:]）能够复制序列。
- 字典copy方法（D.copy()）能够复制字典。
- 有些内置函数（例如，list）能够生成拷贝（list(L)）。
- copy标准库模块能够生成完整拷贝。

举个例子，假如你有一个列表和一个字典，你又不想凭借其他变量来修改它们的值。

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

为了避免这一情况，可以简单地把拷贝赋值为其他变量，而不是相同对象的引用。

```
>>> A = L[:]                    # Instead of A = L (or list(L))
>>> B = D.copy()                # Instead of B = D
```

这样一来，由其他变量产生的改变将会修改拷贝，而不是原对象。

```
>>> A[1] = 'Ni'
>>> B['c'] = 'spam'
>>>
>>> L, D
([1, 2, 3], {'a': 1, 'b': 2})
>>> A, B
([1, 'Ni', 3], {'a': 1, 'c': 'spam', 'b': 2})
```

就我们原始的例子而言，你可以通过对原始列表进行分片而不是简单的命名操作来避免引用的副作用。

```
>>> X = [1, 2, 3]
>>> L = ['a', X[:], 'b']        # Embed copies of X's object
>>> D = {'x':X[:], 'y':2}
```

这样做将改变图9-2 —— L和D现在会指向不同的列表而不再是X。结果就是，凭借X所做的修改只能够影响X而不会再影响L和D。类似地，修改L或D不会影响X。

拷贝需要注意的是：无条件值的分片以及字典copy方法只能做顶层复制。也就是说，不能够复制嵌套的数据结构（如果有的话）。如果你需要一个深层嵌套的数据结构的完整的、完全独立的拷贝，那么就要使用标准的copy模块 —— 包括import copy语句，并编辑X = copy.deepcopy(Y)对任意嵌套对象Y做完整的复制。这一调用语句能够递归地遍

历对象来复制它们所有的组成部分。然而这是相当罕见的情况（这也是为什么这么做比较费劲的原因所在）。引用通常就是你想要的，然而当它们不是你所需要的时候，分片和copy方法通常就是你所需要的复制方法了。

TopSage.com

比较、相等性和真值

所有的Python对象也可以支持比较操作 —— 测试相等性、相对大小等。Python的比较总是检查复合对象的所有部分，直到可以得出结果为止。事实上，当嵌套对象存在时，Python能够自动遍历数据结构，并从左到右递归地应用比较，要多深就走多深。过程中首次发现的差值将决定比较的结果。

例如，在比较列表对象时将自动比较它的所有内容。

```
>>> L1 = [1, ('a', 3)]                # Same value, unique objects
>>> L2 = [1, ('a', 3)]
>>> L1 == L2, L1 is L2                # Equivalent? Same object?
(True, False)
```

在这里L1和L2被赋值为列表，虽然相等，却是不同的对象。因为Python的引用特性（我们在第6章曾经学过），有两种方法可以测试相等性：

- **“==”操作符测试值的相等性。**Python运行相等测试，递归地比较所有内嵌对象。
- **“is”表达式测试对象的一致性。**Python测试二者是否是同一个对象（也就是说，在同一个内存地址中）。

在上一个例子中，L1和L2通过了“==”测试（他们的值相等，因为它们的所有内容都是相等的），但是is测试却失败了（它们是两个不同的对象，因此有不同的内存区域）。我们注意一下短字符串会出现什么情况：

```
>>> S1 = 'spam'
>>> S2 = 'spam'
>>> S1 == S2, S1 is S2
(True, True)
```

在这里，我们应该能又一次得到两个截然不同的对象碰巧有着相同的值：“==”应该为真，而is应该为假。但是因为在Python内部暂时储存并重复使用短字符串作为最佳化，事实上内存里只有一个字符串'spam'供S1和S2分享。因此，“is”一致性测试结果为真。为了得到更一般的结果，我们需要使用更长的字符串：

```
>>> S1 = 'a longer string'
>>> S2 = 'a longer string'
```


www.TopSage.com

```
>>> S1 == S2, S1 is S2
(True, False)
```

当然，由于字符串是不可变的，对象缓存机制和程序代码无关 —— 无论有多少变量与它们有关，字符串是无法在原处修改的。如果一致性测试令你感觉到困惑，可以再回头看一看第6章的相关内容回忆一下对象引用的概念。

凭经验来说，“==”几乎是所有等值检验时都会用到的操作符；而is则保留了极为特殊的角色。你会在本书后面部分看到一些使用这些操作符的情况。

相对大小的比较也能够递归地应用于嵌套的数据结构。

```
>>> L1 = [1, ('a', 3)]
>>> L2 = [1, ('a', 2)]
>>> L1 < L2, L1 == L2, L1 > L2       # Less, equal, greater: tuple of results
(False, False, True)
```

因为嵌套的3大于2，这里的L1大于L2。上面最后一行的结果的确是一个含有三个对象的元组 —— 我们输入的三个表达式的结果（这是一个不带圆括号的元组的实例）。

一般来说，Python中不同的类型的比较方法如下：

- 数字通过相对大小进行比较。
- 字符串是按照字典顺序，一个字符接一个字符地对比进行比较（"abc" < "ac"）。
- 列表和元组从左到右对每部分的内容进行比较。
- 字典通过排序之后的（键、值）列表进行比较。

一般来说，结构化对象的比较就好像是你把对象写成文字，并从左到右一次一个地比较所有部分。在之后的章节中，我们将会看到其他可以改变比较方法的对象类型。

Python中真和假的含义

注意：上一节最后一个例子的元组返回的三个值，它们代表着真和假。它们被打印成True和False，但现在我们要认真地使用这种逻辑测试。

在Python中，与大多数程序设计语言一样，整数0代表假，整数1代表真。不过，除此之外，Python也把任意的空数据结构视为假，把任何非空数据结构视为真。更一般地，真和假的概念是Python中每个对象的固有属性：每个对象不是真就是假，如下所示：

- 数字如果非零，则为真。
- 其他对象如果非空，则为真。

www.TopSage.com

表9-4给出了Python中对象的真、假值的例子。

表 9-4：对象真值的例子

对象	值
"spam"	True
""	False
[]	False
{}	False
1	True
0.0	False
None	False

Python还有一个特殊对象：None（表9-4中最后一项），总被认为是假。我们曾经在第4章介绍过None，这是Python中一种特殊数据类型的唯一值，一般都起到一个空的占位作用，与C语言中的NULL指针类似。

例如，回想一下，对于列表来说，你是无法为偏移赋值的，除非这个偏移是已经存在的（如果你进行超出范围的赋值操作，列表是不会神奇地增长的）。要预先分配一个100项的列表，这样你可以在100个偏移的任何一个加上None对象：

```
>>> L = [None] * 100
>>>
>>> L
[None, None, None, None, None, None, None, ... ]
```

Python的布尔类型bool（在第5章里我们曾经介绍过）只不过是扩展了Python中真、假的概念。你可能已经注意到了，我们本章的测试结果显示为True和False。正如我们在第5章所学的，这些只是整数1和0的定制版本而已。由于这个新的类型的运行方式，这的确只是先前所说的真、假概念的较小扩展而已，这样的设计就是为了让真值更为明确。当明确地用在真值测试时，True和False这些文字就变成了真和假，因为它们的确只是整数1和0的定制版本而已。测试结果同样打印成True和False的字样，而不是1和0。

像if这样的逻辑语句中，没必要只用布尔类型。所有对象本质上依然是真或假，即使使用其他类型，本章所提到的所有布尔概念依然是可用的。等到我们在第12章研究逻辑语句的时候，将进一步探讨布尔值。

www.TopSage.com

Python的类型层次

大家网 TopSage.com

图9-3总结了Python允许的所有内置对象类型以及它们之间的关系。我们已经讨论了它们当中最主要的，图9-3中多数其他对象种类都相当于程序单位（例如，函数和模块），或者说明了解释器的内容（例如，堆栈和编译码）。

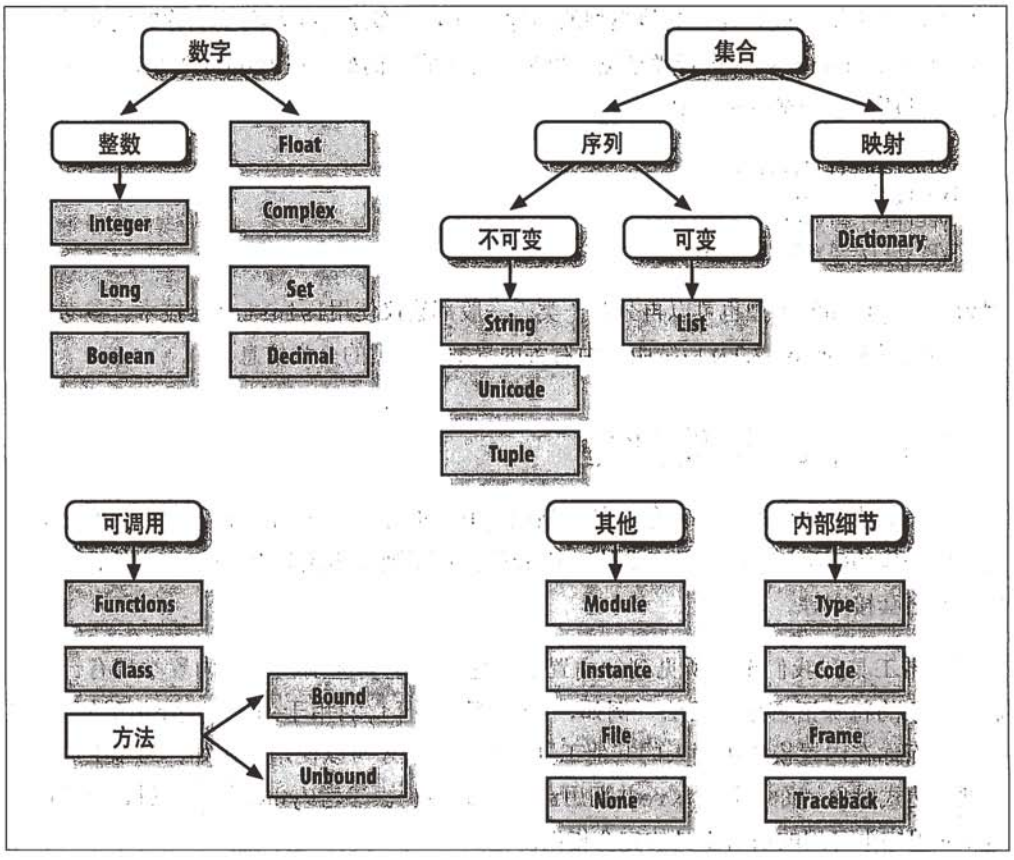


图9-3：按类别组织的Python的主要内置对象类型。Python中所有一切都是某种类型的对象，即便是某个对象的类型！任何对象的类型都是类型为"type"的对象

这里需要特别注意的是，Python系统中的任何东西都是对象类型，而且可以由Python程序来处理。例如，可以传递一个类给函数，赋值为一个变量，放入一个列表或是字典中等。

事实上，即使是类型本身在Python中也是对象类型 —— 调用内置函数type(X)能够返回对象X的类型对象。类型对象可以用在Python中的if语句来进行手动类型比较。然而，如第4章曾介绍过的原因，手动类型测试通常在Python中并不是个正确的选择。

www.TopSage.com

有关类型名称需要提醒一下：Python 2.2的每个核心类型都有个新的内置名来支持面向对象子类的类型定制：dict、list、str、tuple、int、long、float、complex、unicode、type和file（file是open的同义词）。调用这些名称事实上是对这些对象构造函数的调用，而不仅仅是转换函数，不过作为基本的使用来说，你还是可以把它们当作简单的函数的。

类型标准库模块也同样提供其他类型的名称（目前大部分是内置类型名称的同义词），而且用isinstance函数进行类型测试也是有可能的。例如，在Python 2.2以及后续版本中，下列所有类型测试都为真：

```
isinstance([1],list)
type([1])==list
type([1])==type([])
type([1])==types.ListType
```

因为目前Python的类型也可以再分为子类，一般都建议使用isinstance技术。参考第26章可得到更多Python 2.2及后续版本中有关内置类型子类的分类信息。

Python中的其他类型

除了这一章我们所学的核心对象之外，典型Python的安装还有几十种其他可用的对象类型，允许作为C语言的扩展程序或是Python的类：正则表达式对象、DBM文件、GUI组件、网络套接字等。

这些附带工具和我们至今所见到的内置类型之间的主要区别在于，内置类型有针对它们的对象的特殊语言生成语法（例如，4用于整数，[1,2]用于列表，open函数用于文件）。而你在内置模块中输入的其他工具必须先导入才可以使用。例如，为了生成一个正则表达式对象，你需要输入re并调用re.compile()。参考Python的库手册，可以得到Python程序中可用的所有工具的完全指南。

内置类型陷阱

这将是核心数据类型的最后一站。我们将一起讨论可能会困扰新手的（有时也会让专家感到头疼的）一些常见问题和相应的解决办法来完成这一章的内容。其中有些是我们已经讨论过的内容，但它们的确非常重要，值得我们再一次提醒读者。

赋值生成引用，而不是拷贝

因为这是非常核心的概念，我们在这里再提一次 —— 你需要理解程序中的共享引用是怎

www.TopSage.com

么回事。例如，下列这个例子中赋值为L的列表对象不但被L引用，也被赋值为M的内部列表引用。在原处修改了L的同时也修改了M的引用：

```
>>> L = [1, 2, 3]
>>> M = ['X', L, 'Y']              # Embed a reference to L
>>> M
['X', [1, 2, 3], 'Y']

>>> L[1] = 0                       # Changes M too
>>> M
['X', [1, 0, 3], 'Y']
```

这种影响通常只是在大型程序中才显得重要，而共享引用往往就是你真正想要的。如果不是这样，你可以明确地对它们进行拷贝以避免对象共享。就列表而言，你总能通过使用无限制条件的分片生成一个高级拷贝。

```
>>> L = [1, 2, 3]
>>> M = ['X', L[:], 'Y']           # Embed a copy of L
>>> L[1] = 0                       # Changes only L, not M
>>> L
[1, 0, 3]
>>> M
['X', [1, 2, 3], 'Y']
```

记住，分片限制的默认为0和所分片段的序列长度 —— 如果两者都省略的话，分片就会抽取序列中每一项，这样就生成了一个顶部拷贝（一个新的、无共享的对象）。

重复能够增加层次深度

序列重复就好像是多次将一个序列加到自己身上。这是事实，但是当可变序列被嵌套时效果就不见得总像你所想的那样。例如，下面这个例子中的X赋值给重复四次的L，而Y赋值给包含重复四次的L的列表：

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

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

由于L在第二次重复中是嵌套的，Y结束嵌套的引用，返回赋值为L的原始列表，所以出现了与上一节提到的相同类型的副作用：

```
>>> L[1] = 0                       # Impacts Y but not X
>>> X
```

```
[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
>>> Y
[[4, 0, 6], [4, 0, 6], [4, 0, 6], [4, 0, 6]]
```

大家网 TopSage.com

解决这一问题的方法与前面所讲过的一样，因为这的确是以另一种生成共享可变对象引用的方法。如果你还记得重复、合并以及分片只是在复制操作数对象的顶层的话，这类情况就比较好理解了。

留意循环数据结构

在前面的练习中遇到过这个问题：如果遇到一个复合对象包含指向自身的引用，就称之为循环对象。无论何时Python在对象中检测到循环，都会打印成[...]，而不会陷入无限循环。

```
>>> L = ['grail']                # Append reference to same object
>>> L.append(L)                  # Generates cycle in object: [...]
>>> L
['grail', [...]]
```

除了要了解方括号括起来的三个点代表对象中带有循环之外，有一种情况也应该知道，因为它会造成误区 —— 如果不小心，循环结构可能导致程序代码陷入无法预期的循环当中。例如，有些程序中保留了已经被访问过的列表或者字典，并通过检测来确定他们是否在循环当中。你可以参考第3章“第1部分练习题”得到有关这一问题的更多信息，也可以在第21章结尾的*reloadall.py*程序部分找到的相关的解决方法。

除非你真的需要，否则不要使用循环引用。虽然有很多创建循环的不错的理由，但除非你知道代码会如何处理，你可能不想让对象在实际中频繁地引用自身。

不可变类型不可以在原处改变

最后，你不能在实地改变不可变对象。如果需要的话，你得通过分片、合并等操作来创建一个新的对象，再向后赋值给原引用。

```
T = (1, 2, 3)

T[2] = 4                  # Error!

T = T[:2] + (4,)          # OK: (1, 2, 4)
```

这看起来可能像是多余的代码编辑工作，但是这样做有个好处，当你使用元组和字符串这样的不可变对象的时候，就不会发生前面提到的问题了。因为无法在原处修改，就不会产生列表的那种副作用。

www.TopSage.com

本章小结

我们在这一章学习了两种主要核心的对象类型：元组和文件。我们看到，元组支持所有一般的序列操作，但它们没有方法，因为是不可变的而不能进行任何在原处的修改。我们也看到，文件是由内置open函数返回的并且提供读写数据的方法。我们探讨为了存储到文件当中，如何让Python对象来回转换字符串，而我们也了解了pickle和struct模块的高级角色（对象序列化和二进制数据）。最后，我们复习一些所有对象类型共有的特性（例如，共享引用），讨论对象类型领域内常见的错误（“陷阱”）而完成这一章的内容。

接下来的部分，我们要转向讨论Python的语句语法。我们要在接下来几章里探讨Python所有的程序语句。下一章是这一部分的开头，我们将会介绍Python适用于所有语句类型的通用语法模型。不过，继续学习下面的内容之前，还是让我们先来做一做本章的习题，练习一下每章结尾的实验题来复习有关类型的概念。语句大体上就是生成并处理对象，所以在继续学习之前，你需要做这些练习，确定你已经熟练掌握本部分内容了。

本章习题

1. 你怎么确定元组有多大？
2. 写个表达式，修改元组中第一个元素。在此过程中，(4, 5, 6)应该变成(1, 5, 6)。
3. open文件调用中，默认的处理模式自变量是什么？
4. 你可能使用什么模块把Python对象储存在文件中，而不需要亲自将它们转换成字符串？
5. 你怎么一次复制嵌套结构的所有组成部分？
6. Python在什么时候会认为一个对象为真？
7. 你追求的是什么？

习题解答

1. 内置的len函数会传回Python中任何容器对象的长度（所含元素的数目），包括元组在内。这是内置函数，而不是类型方法，因为它适用于多种对象。
2. 因为它们是不可变的，你无法真正地在原处修改元组，但是你可以用所需要的值产生新元组。已知T = (4, 5, 6)，要修改第一个元素时，可以从其组成成份进行分片运算和合并来创建新元组：T = (1,) + T[1:]（回想一下，单个元素的元组需要多余的逗号）。你也可以把元组转成列表，在原处进行修改，再转换成元组，但是，这样做太麻烦了，在实际应用中也很少用。如果你知道对象需要在原处进行修改，就使用列表。
3. open文件调用中的处理模式参数默认值为'r'：读取输入。对于输入文本文件，只要传入外部文件名即可。
4. pickle模块可用于把Python对象储存在文件中，而不用刻意转成字符串。struct模块也是相关的，但是，那是要把数据打包成为二进制格式，从而保存在文件中。
5. 如果需要复制嵌套结构X的所有组成部分，就导入copy模块，然后调用copy.deepcopy(X)。在实际中，这也很罕见；引用往往就是所需要的行为，而浅层复制（例如，aList[:]、aDict.copy()）通常就足够满足大多数的复制。
6. 如果对象是非零数字或非空集合体对象，就被认作是真。内置的True和False关键字，从实质上来说，就是预先定义的整数1和0。
7. 可接受的答案包括“学习Python”、“继续学习本书下一部分内容”和“寻找圣杯”。

www.TopSage.com

大家网 TopSage.com

第二部分练习题

这里涉及内置的对象基础。就像往常一样，一些新概念会在这个过程中出现，所以当你做完时（或者做不完时），一定要翻到附录B看一看答案。如果你时间有限，建议你从练习题10和11开始（最实际的），然后在时间允许的情况下，再从头做到尾。不过这全都是基本材料，所以能做多少就做多少吧。

1. **基础。**以交互模式，实验第2部分表中的常见类型的表达式。首先，打开Python交互模式解释器，输入下列表达式，然后试着说明每种情况所产生的结果：

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

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

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

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

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

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

2. **索引运算和分片运算。**在交互模式提示符下，定义一个名为L的列表，内含四个字符串或数字（例如，`L=[0,1,2,3]`）。然后，实验一些边界情况：

   a. 索引值超过边界时，会发生何事（例如，`L[4]`）？

   b. 分片运算超出边界时又如何（例如，`L[-1000:100]`）？

   c. 最后，如果你试着反向抽取序列，也就是较低边界值大于较高边界

www.TopSage.com

值（例如，L[3:1]），Python会怎么处理？提示：试着赋值至此分片（L[3:1]=['?']），看看此值置于何处。你觉得这和分片超出边界是相同的现象吗？

3. **索引运算、分片运算以及del。**定义另一个列表L，有四个元素，然后赋值一个空列表给其偏移值之一（例如，L[2]=[]）。发生什么事？然后，赋值空列表给分片（L[2:3]=[]）。现在，发生了什么事？回想一下，分片赋值运算删除分片，并将新值插入分片曾经的地方。

   del叙述会删除偏移值、键、属性以及名称。将其用在列表上来删除一个元素（例如，del L[0]）。如果你删除整个分片，会发生何事（del L[1:]）？当你赋值非序列给分片时，会发生什么变化（L[1:2]=1）？

4. **元组赋值运算。**输入下面几行：

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

   当你输入这个序列时，你觉得X和Y会发生什么变化？

5. **字典键。**考虑下列代码片段：

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

   已知道字典不是按偏移值读取的，那么这里是在做什么？下面的例子让你看见了其主题吗？（提示：字符串、整数以及元组共享哪种类型分类？）

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

6. **字典索引运算。**创建一个字典，名为D，有三个元素，而键为'a'和'c'。如果你试着以不存在的键进行索引运算（D['d']），会发生何事？如果你试着赋值到不存在的键'd'（例如，D['d']='spam'），Python会怎么做？这一点和列表超出边界的赋值运算以及引用应该作何比较？这听起来是否像是变量名的规则？

7. **通用运算。**执行交互模式测试来回答下列问题：

   a. 使用“+”运算符在不同“/”混合的类型时（例如，字符串+列表和列表+元组），会发生何事？

   b. 当操作数之一是字典时，“+”还能工作吗？

www.TopSage.com

c. append方法能用于列表和字符串吗？可以对列表使用keys方法吗？（提示：append对其主体对象做了什么假设吗？）

d. 最后，当你对两个列表或两个字符串做分片或合并时，你会得到什么类型的对象？

8. **字符串索引运算**。定义一个S字符串，有四个字符：S = "spam"。然后输入下列表达式：S[0][0][0][0][0]。对这一次所发生的事，有任何线索吗？（提示：回想一下，字符串是字符集合体，但是，Python字符是一个字符的字符串）。如果你将此索引表达式施加于['s', 'p', 'a', 'm']这种列表，还能运行吗？为什么？

9. **不可变类型**。再次定义一个四字符的S字符串：S = "spam"。写个赋值运算，把字符串改成"slam"；只使用分片和合并运算。你可以只用索引运算和串接进行相同运算吗？索引赋值运算行吗？

10. **嵌套**。写个数据结构，表示你的个人信息：姓名（名、中间名、姓）、年龄、工作、住址、电子信箱以及电话号码。你可以用任何喜欢的内置对象类型组合创建这个数据结构（列表、元组、字典、字符串、数字）。然后，通过索引运算读取数据结构的个别元素。就这个对象而言，有些结构是否比其他的更有道理？

11. **文件**。写个脚本，创建名为*myfile.txt*的新的输出文件，把字符串"Hello file world!"写入。然后写另一个脚本，开启*myfile.txt*，把其内容读出来并将其打印。从系统命令行执行你的两个脚本。新文件是否出现在执行脚本所在的目录中？如果对开启的文件名新增不同的目录路径，又会怎样？附注：文件写入方法是不会新增换行字符到你的字符串的；如果你想在文件中完全终止这一行，就在字符串末尾明确的增加\n。

12. **再谈dir函数**。尝试在交互模式提示符下输入下列表达式。从1.5版起，dir函数就被通用化用来列出你可能感兴趣的任何Python对象的所有属性。如果你用的版本比1.5版还早，__methods__机制也有相同的效果。如果你在用Python 2.2，dir可能是唯一可用的方法。

```
[].__methods__              # 1.4 or 1.5
dir([])                     # 1.5 and later

{}.__methods__              # Dictionary
dir({})
```


www.TopSage.com

大家网
TopSage.com

www.TopSage.com

第三部分

语句和语法

www.TopSage.com

www.TopSage.com

大家网
TopSage.com

第10章 Python语句简介

现在我们已经熟悉了Python核心内置对象类型，在这一章里我们将探讨它的基本语句类型。和以前一样，我们在这里首先大概介绍一下语句的语法，在接下来的几章中我们将要讨论具体语句的更多细节。

简单地说，语句就是写出来要告诉Python，你的程序应该做什么的句子。如果程序是“用原料做事情”的话，那么语句就是你指定程序要做哪些事情的方式。Python是面向过程的、基于语句的语言。通过组合这些语句，可以指定一个过程，由Python实现程序的目标。

重访Python程序结构

另一种理解语句角色的方法就是再回头看一下我们在第4章曾介绍的概念层次，我们在该章曾讨论了内置对象以及相应的表达式。本章将深入学习：

1. 程序由模块构成。
2. 模块包含语句。
3. 语句包含表达式。
4. 表达式建立并处理对象。

Python的语法实质上是由语句和表达式组成的。表达式处理对象并嵌套在语句中。语句编辑实现程序操作中更大的逻辑关系——它们使用并引导表达式处理我们前几章所学的对象。此外，语句还是对象生成的地方（例如，赋值语句中的表达式），有些语句会完全生成新的对象类型（函数、类等）。语句总是存在于模块中的，而模块本身则又是由语句来管理的。

www.TopSage.com

Python的语句

表10-1总结了Python的语句集（注1）。本书这一部分将按照表格中从上到下的顺序进行讨论。我们曾经接触过表10-1的一部分语句。本书这一部分将会补充我们之前跳过的细节，介绍剩下的Python基本过程语句并讨论整体语法模型。表10-1中位置靠后部分的语句和较大程序单元有关 —— 函数、类、模块以及异常，这些语句会引入更大更复杂的程序设计思路，所以我们对每个概念将用一章的篇幅来进行阐述。更多的类似exec（编译和执行写成字符串的程序代码）这样独特的语句会在本书后面部分进行讨论，或者可以参考Python标准说明文件。

表10-1：Python语句

语句	角色	例子
赋值	创建引用值	a, b, c = 'good', 'bad', 'ugly'
调用	执行函数	log.write("spam, ham\n")
print	打印对象	print 'The Killer', joke
if/elif/else	选择动作	if "python" in text: print text
for/else	序列迭代	for x in mylist: print x
while/else	一般循环	while X > Y: print 'hello'
pass	空占位符	while True: pass
break, continue	循环跳跃	while True: if not line: break

try/except/ finally	捕捉异常	try:
		    action()
		except:
		    print 'action error'

注1：从技术角度上来讲，在Python 2.5中，yield变成表达式而不是语句，而try/except和try/finally语句则合并了（两者先前是单独的语句，但现在我们能在同一个try语句内写出except和finally）。此外，Python 2.6中新增了with/as语句来编写环境管理器概括地讲，这是代替try/finally异常相关操作的东西（在2.5版中，with/as是选用的扩展功能，除非执行语句from __future__ import with_statement，刻意将其打开，否则就无法使用）。参考Python手册相关细节。在3.0版中，print和exec会变成函数调用而不是语句，而新的nonlocal语句的用途则会类似于当前的global。

www.TopSage.com

表10-1：Python语句（续）

语句	角色	例子
raise	触发异常	raise endSearch, location
import, from	模块读取	import sys from sys import stdin

def, return, yield	创建函数	def f(a, b, c=1, *d):
		    return a+b+c+d[0]
		def gen(n):
		    for i in n, yield i*2

class	创建对象	class subclass(Superclass): staticData = []

global	命名空间空间	def function():
		    global x, y
		    x = 'new'

del	删除引用	del data[k] del data[i:j] del obj.attr del variable
exec	执行代码字符串	exec "import " + modName exec code in gdict, ldict
assert	调试检查	assert X > Y
with/as	环境管理器（2.6）	with open('data') as myfile: process(myfile)

两个if的故事

在我们深入探索表10-1中的任何一个具体语句之前，我们应该首先了解在Python代码中不能输入什么，之后开始Python语句语法的学习，弄明白这些你才能将它与以前可能见过的其他语法模型进行比较和对比。

考虑下面这个if语句，它是用类似C语言的语法写出来的：

```
if (x > y) {
    x = 1;
    y = 2;
}
```

这有可能是C、C++、Java、JavaScript或Perl的语句。现在，让我们看一看Python语言中与之等价的语句：

```
if x > y:
    x = 1
    y = 2
```

注意：等价的Python语句没有那么杂乱。也就是说，语法成分比较少。这是刻意设计的。作为脚本语言，Python的目标之一就是让程序员少打些字让生活轻松一点。

更明确地讲，当对照两种语法模型时，你会注意到Python多了一项新内容，而且存在于类C语言中的三个选项在Python程序中找不到。

Python增加了什么

Python中新的语法成分是冒号（:）。所有Python的复合语句（也就是语句中嵌套了语句）都有相同的一般形式，也就是首行以冒号结尾，首行下一行嵌套的代码往往按缩进的格式书写，如下所示。

```
Header line:
    Nested statement block
```

冒号是不可或缺的，遗漏掉冒号可能是Python新手最常犯的错误之一 —— 这绝对是我在Python培训课上见过无数次的错误。事实上，如果你刚刚开始学习Python，几乎很快就会忘记这个冒号。大多数和Python兼容的编辑器都会很容易被发现这一错误，而最终变成潜意识里的一种习惯（习惯到你也会在C++程序代码中输入冒号使C++编译器产生很多可笑的错误信息）。

Python删除了什么

虽然Python需要额外的冒号，但是你必须在类C语言程序中加入而通常不需要在Python中加入的语法成分却有三项。

括号是可选的

首先是语句顶端位于测试两侧的一对括号：

```
if (x < y)
```

许多类C语言的语法都需要这里的括号。而在Python中并非如此，我们可以省略括号而语句依然会正常工作：

www.TopSage.com

```
if x < y
```

从技术角度来讲，由于每个表达式都可以用括号括起来，在这里的Python程序中加上括号也没什么问题，不会被视为错误的if。但是不要这么做，你只会让你的键盘坏得更快，并且全世界都会知道你只是个正在学习Python的前C程序员。Python方式就是在这类语句中完全省略括号。

终止行就是终止语句

不会出现在Python程序代码中的第二个重要的语法成分就是分号。Python之中你不需要像类C语言那样用分号终止语句：

```
x = 1;
```

在Python中，一般原则是，一行的结束会自动终止出现在该行的语句。换句话说，就是你可以抛弃分号并且程序会正确工作：

```
x = 1
```

有些方式可以避开这一原则。但是一般来说，绝大多数Python程序代码都是每行一个语句，不需要分号。

如果你渴望像C语言一样进行程序设计（如果这种情况有可能出现的话），你还是可以在每个语句末使用分号的。当它们出现时，Python允许你侥幸通过。但是，别这么做（真的！）。同样，这么做是在告诉全世界，你依然是个C程序员，还没完全切换到Python中去。Python的风格就是完全不要分号。

缩进的结束就是代码块的结束

Python删除的第三个也是最后一个语法成分，也许对马上成为前C程序员的人来讲是最不寻常的一个（直到他们用上10分钟之后意识到它实际上是一大特色），也就是你不用刻意在程序代码中输入任何语法上用来标明嵌套代码块的开头和结尾的东西。你不需要像类C语言那样，在嵌套块前后输入begin/end、then/endif或者大括号：

```
if (x > y) {
    x = 1;
    y = 2;
}
```

取而代之的是，在Python中我们一致地把嵌套块里所有的语句向右缩进相同的距离，Python能够使用语句的实际缩进来确定代码块的开头与结尾：

www.TopSage.com

```
if x > y:
    x = 1
    y = 2
```

所谓缩进，是指这里的两个嵌套语句至左侧的所有空白。Python并不在乎你怎么缩进（你可以使用空格或制表符）或者你缩进多少（你可以使用任意个空格或是制表符）。实际上，两个嵌套代码块的缩进可以完全不同。语法规则只不过是给定一个单独的嵌套块中所有语句都必须缩进相同的距离。如果不是这样就会出现语法错误，而程序就无法运行了，直到把缩进修改一致。

为什么使用缩进语法

对于习惯了类C语言的程序员而言，缩进规则乍一看可能会有点特别，但是这正是Python精心设计的特点，是Python迫使程序员写出统一、整齐并具有可读性程序的主要方式之一。这就意味着你必须根据程序的逻辑结构，以垂直对齐的方式来组织程序代码。结果就是让程序更一致并具有可读性（不像类C语言所写的多数程序那样）。

更明确地讲，根据逻辑结构将代码对齐是令程序具有可读性的主要部分，因而具备了重用性和可维护性，对自己和他人都是如此。实际上，即使你在看过本书之后不使用Python，也应该在任何块结构的语言中对齐代码让程序更具可读性。Python将其设计为语法的一部分来强制程序的书写，但这也是在任何程序语言中都非常重要的一点并对代码起重要的作用。

大家的设计经历可能不同，但当我还在做全职基础开发的时候都是在处理许多程序员做过很多年的大而老的C++程序。几乎不可避免的是，每位程序员都有自己的缩进代码的风格。例如，别人总叫我修改用C++写的while循环，开头是这样的：

```
while (x > 0) {
```

在我们深入研究缩进之前，有三四种供程序员在类C语言程序中安排大括号的方式，组织通常有官方的争论以及编写标准手册说明相关选项（似乎距离我们需要用程序解决的问题有点太远了）。我们先看一下时常在C++代码中碰到的情况。第一个写代码的人的缩进为四个空格：

```
while (x > 0) {
    --------;
    --------;
```

这个人后来挤进管理层，只能由某个喜欢再往右缩进一点的人来接替他的位置：

www.TopSage.com

大家网
TopSage.com

```
while (x > 0) {
    --------;
    --------;
          --------;
          --------;
```

那个人后来又遇到了其他的机会，而某个接手这段代码的人喜欢少缩进一些：

```
while (x > 0) {
    --------;
    --------;
          --------;
          --------;
--------;
--------;
}
```

最后，这个代码块由关闭大括号（}）终止，大括号当然能让代码变成块结构了（他讽刺地说）。在任何代码块结构的语言中，无论是Python还是其他语言，如果嵌套代码块缩进的不一致，读者将很难解释、修改或者再使用。可读性是很重要的，缩进又是可读性的主要元素。

如果你用类C语言写过很多程序的话，可能你曾经为下面的例子头疼过。考虑下面这个C语言的语句：

```
if (x)
     if (y)
          statement1;
else
     statement2;
```

这个else是属于哪个if的呢？令人吃惊的是，这个else是属于嵌套的if语句[if (y)]，即使它看上去很像是属于外层if (x)的。这是C语言中经典的陷阱，而且可能导致读者完全误解代码并用不正确的方式进行修改还一直找不出原因，直到产生巨大的错误为止！

这种事Python中是不可能发生的：因为缩进很重要，程序看上去什么样就意味着它将如何运行。考虑一个等价的Python语句：

```
if x:
     if y:
          statement1
else:
     statement2
```

这个例子里，else垂直对齐的if就是其逻辑上if（外层的if x）。从某种意义上来说，

www.TopSage.com

Python是WYSIWYG语言 —— 所见即所得（what you see is what you get）。因为不管是谁写的，程序看上去的样子就是运行的方式。

如果这样还不足以突显Python语法的优点的话，来听听下面这个故事。在我职业生涯的早期，我在一家成功的用C语言开发系统软件的公司工作，C语言并不要求一致的缩进。即使是这样，当我们在下班之前把程序代码上传到源代码控制系统的时候，公司会运行一个自动化的脚本用来分析代码中的缩进。如果这个脚本检查到我们没有一致地缩进程序代码，我们将会在第二天早上收到自动发出的关于此事的电子邮件，同时我们的老板们也会收到！

我的意思是，即使是某个不要求这样做的语言，优秀的程序员都知道一致地使用缩进对于程序代码的可读性和质量有着至关重要的作用。Python将它升级到语法层次的事实，被绝大多数人视为是语言的特色。

最后，记住一点，目前几乎每个对程序员友好的文本编辑器都有对Python语法模型的内置支持。例如，在IDLE Python GUI中，当输入嵌套代码块时代码行会自动缩进，按下Backspace键会回到上一层的缩进，你可以在嵌套块里面调整IDLE将语句往右缩进多少。

缩进没有绝对的标准：常见的是每层四个空格或一个制表符，但是你想怎么缩进以及缩进多少都由你自己决定。嵌套越深的代码块向右缩进的越厉害，越浅就越靠近前一个块。此外，生成制表符来代替大括号，对于必须输出Python代码的工具而言在实践中不再是什么困难。总的来说，还是按照你在类C语言中该做的那样去做，不过无论如何都要去掉大括号，这样你的程序代码就满足Python的语法规则了。

几个特殊实例

正如我们曾经提到的，在Python的语法模型中：

- 一行的结束就是终止该行语句（没有分号）。
- 嵌套语句是代码块并且与实际的缩进相关（没有大括号）。

这些规则几乎涵盖了实际中你会写出或看到的所有Python程序。然而，Python也提供了一些特殊用途的规则来调整语句和嵌套语句的代码块。

语句规则的特殊情况

虽然语句一般都是一行一个，但是Python中也有可能出现某一行挤进多个语句的情况，这时它们由分号隔开：

www.TopSage.com

```
a = 1; b = 2; print a + b          # Three statements on one line
```

这是Python中唯一需要分号的地方——作为语句界定符。不过，只有当摆到一起的语句本身不是复合语句才行。换句话说，只能把简单语句放在一起。例如，赋值操作、打印和函数调用。复合语句还是必须出现在自己的行里（否则，你可以把整个程序挤在同一行上，这样很有可能你在团队里不会受到好评）。

语句的另一个特殊规则本质上就是反向——可以让一个语句的范围横跨很多。为了能实现这一操作，你只需要用一对括号把语句括起来就可以了：括号（()）、方括号（[]）或者字典的大括号（{}）。任何被括在这些符号里的程序代码都可横跨好几行。语句将一直运行，直到Python遇到包含闭合括号的那一行。例如，连续几行列表的常量：

```
mlist = [111,
         222,
         333]
```

由于程序被括在一对方括号里，Python就会接着运行下一行，直到遇见闭合的方括号为止。字典也可以用这个方法横跨数行，括号可以处理元组、函数调用和表达式。连续行的缩进是无所谓的，尽管常识告诉我们为了让程序具有可读性，那几行也应该对齐。

括号是可以包含一切——因为任何表达式都可以包含在内，只要插入一个左边括号，你就可以到下一行接着写你的语句。

```
X = (A + B +
     C + D)
```

顺便说一句，这种技巧也适用于复合语句。不管你在什么地方需要写一个大型的表达式，只要把它括在括号里，就可以在下一行接着写：

```
if (A == 1 and
    B == 2 and
    C == 3):
        print 'spam' * 3
```

有一条比较老的规则也允许我们跨越数行——当上一行以反斜线结束时，可以在下一行继续：

```
X = A + B + \
      C + D
```

但是这种方法已经过时了，目前从某种程度上来说，不再提倡使用这种方法，因为关注并维护反斜线比较困难，而且这种做法相当脆弱（反斜线之后可能没空格）。而且

这也是倒退回C语言的例子，因为反斜线时常在#define的宏里面使用。再强调一次，在Python中，做Python程序员该做的事，不要做C程序员做的事。

TopSage.com

代码块规则特殊实例

正如我们之前所提到的，嵌套代码块中的语句一般都与向右缩进相同的量相关联。这里给出一个特殊案例，说明复合语句的主体可以出现在Python的首行冒号之后。

```
if x > y: print x
```

这样我们就能够编辑单行if语句、单行循环等。不过，只有当复合语句本身不包含任何复合语句的时候，才能这样做。也就是说，只有简单语句可以跟在冒号后面，比如赋值操作、打印、函数调用等。较复杂的语句仍然必须单独编辑放在自己的行里。复合语句的附带部分（例如if的else部分，以后我们会看到）也必须在自己的行里。语句体可以由几个简单语句组成并用分号隔开，但这种做法已经越来越不受欢迎了。

总的来说，即使并不一定总是必须这样做，但是如果你将所有语句都分别放在不同的行里并总是将嵌套代码块缩进，那么程序代码会更容易读懂并且便于后期的修改。然而，实际中这些规则有一个非常重要而且很常见的例外（中断循环的单行if语句的使用）需要看一下，让我们继续学习下一个部分并编写一些实际的代码吧。

简短实例：交互循环

我们在后续几章学习Python具体的复合语句时，会看到所有这些实际应用中的语法规则，但是它们在Python语言中的工作方式都是相同的。为了入门，让我们看一个简单的实例来说明实际应用中语句语法和语句嵌套相结合的方式，并在其间介绍一些语句。

一个简单的交互式循环

假设有人要你写个Python程序，要求在控制窗口与用户交互。也许你要把输入数据传送到数据库，或者读取将参与计算的数字。不管是什么目的，你需要写一个能够读取用户键盘输入数据的循环并打印每次读取的结果。换句话说，你需要写一个标准的“读取/计算/打印”的循环程序。

在Python中，这种交互式循环的典型模板代码可能会像这样。

```
while True:
    reply = raw_input('Enter text:')
    if reply == 'stop': break
    print reply.upper()
```


www.TopSage.com

这段代码使用了一些新概念，如下所示。

- 这个程序利用了Python的while循环，它是Python最通用的循环语句。我们稍后会介绍while语句更多的细节，但简单地说，它的组成为：while这个单词，之后跟一个其结果为真或假的表达式，再接一个当顶端测试为真（这时的True看作是永远为真）时不停地迭代的嵌套代码块。
- 我们之前曾在本书中见过的raw_input内置函数，在这里的作用是通用控制台输入——打印可选的参数字符串作为提示信息并返回用户输入的回复字符串。
- 利用嵌套代码块特殊规则的单行if语句也在这里出现：if语句体出现在冒号之后的首行，而并不是在首行的下一行缩进。这两种方式哪一种都可以，但在这里我们就省了一行。
- 最后，Python的break语句用于立即退出循环。也就是完全跳出循环语句而程序会继续循环之后的部分。如果没有这个退出语句，while循环会因为测试总是真值而永远循环下去。

事实上，这样的语句组合实质上是指：从用户那里读取一行并用大写字母打印，直到用户输入“stop”为止。还有一些其他的方式可以编写这样的循环，但这里我们所采用的是在Python程序中很常见的一种形式。

需要注意的是，在while首行下面嵌套的三行的缩进是相同的。由于它们是以垂直的方式对齐的，所以它们是和while测试相关联的并重复运行的代码块。源文件的结束或是一个缩进较少的语句都能够终止这个循环体块。

当程序运行时，我们从这个程序取得的某种程度上的交互。

```
Enter text:spam
SPAM
Enter text:42
42
Enter text:stop
```

对用户输入数据做数学运算

脚本能够运行，但现在假设不是把文本字符串转换为大写字母，而是想对数值的输入做些数学运算。例如，求平方。也许会让新用户感到泄气。尝试使用下面的语句来达到想要的效果。

```
>>> reply = '20'
>>> reply ** 2
```

```
...error text omitted...
TypeError: unsupported operand type(s) for ** or pow(  ): 'str' and 'int'
```

然而这无法在脚本中运行，因为（我们在本书前一部分讨论过）除非表达式里的对象类型都是数字，否则Python不会在表达式中转换对象类型，而来自于用户的输入返回脚本时一定是一个字符串。我们无法使用数字的字符串求幂，除非我们手动地把它转换为整数：

```
>>> int(reply) ** 2
400
```

有了这个信息之后，现在我们可以重新编写循环来执行必要的数学运算。

```
while True:
    reply = raw_input('Enter text:')
    if reply == 'stop': break
    print int(reply) ** 2
print 'Bye'
```

像以前一样，这个脚本用了一个单行if语句在“stop”处退出，但是也能够转换输入来进行需要的数学运算。这个版本在底端加了一条结束信息。因为最后一行的print语句不像嵌套代码块那样缩进，不会看作是循环体的一部分，所以而只能退出循环之后运行一次。

```
Enter text:2
4
Enter text:40
1600
Enter text:stop
Bye
```

用测试输入数据来处理错误

到目前为止只需要注意当输入无效时会发生什么现象。

```
Enter text:xxx
...error text omitted...
ValueError: invalid literal for int() with base 10: 'xxx'
```

面对一个错误时，内置int函数会发生异常。如果想要我们的脚本够健全，可以事先用字符串对象的isdigit方法检查字符串的内容。

```
>>> S = '123'
>>> T = 'xxx'
>>> S.isdigit(), T.isdigit()
(True, False)
```


www.TopSage.com

这样就给我们一个在这个例子中进一步嵌套语句的理由。下面这个新版本的交互式脚本使用全方位的if语句来避免错误导致的异常。

TopSage.com

```
while True:
    reply = raw_input('Enter text:')
    if reply == 'stop':
        break
    elif not reply.isdigit():
        print 'Bad!' * 8
    else:
        print int(reply) ** 2
print 'Bye'
```

我们会在第12章进一步研究if语句，但这是在脚本中编写逻辑相当轻便的工具。其完整形式的构成是： if这个关键字后面接测试以及相配的代码块，一个或多个选用的elif（else if）测试以及代码块，以及一个选用的else部分和末尾的一个相配的代码块来作为默认行为。Python会执行首次测试为真所相配的代码块，按照由上至下的顺序，如果所有测试都是假，就执行else部分。

上一个例子中的if、elif以及else部分都属于相同语句的部分，因为它们都垂直对齐（也就是共享相同层次的缩进）。if语句从if这个字横跨至最后一列的print语句。接着，整个if区块是while循环的一部分，因为它们全部缩进在该循环首行下。一旦你了解了，语句嵌套就很自然了。

当我们运行新脚本时，程序会在错误发生前捕捉它，然后打印出（虽然不灵活）错误消息来进行说明。

```
Enter text:5
25
Enter text:xyz
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
Enter text:10
100
Enter text:stop
```

用try语句处理错误

之前的解法能够工作，但到本书后面就会知道，在Python中，处理错误最通用的方式是使用try语句，用它来捕捉并完全复原错误。本书最后一部分深入探索这个语句，在这里先看简单介绍一下。使用try会让有些人认为这要比上一个版本更简单一些。

```
while True:
    reply = raw_input('Enter text:')
    if reply == 'stop': break
```

```
    try:
        num = int(reply)
    except:
        print 'Bad!' * 8
    else:
        print int(reply) ** 2
print 'Bye'
```

这个版本的运作方式和上一个版本的相同，但是，本书把刻意进行错误检查的代码，换成了假设转换可工作的代码，然后把无法运作的情况，包含在异常处理器中。这个try语句的组成是：try关键字后面跟代码主要代码块（我们尝试运行的代码），再跟except部分，给异常处理器代码，再接else部分，如果try部分没有引发异常，就执行这一部分的代码。Python会先执行try部分，然后运行except部分（如果有异常发生）或else部分（如果没有异常发生）。

从语句嵌套观点来看，因为try、except以及else这些关键字全都缩进在同一层次上，它们全都被视为单个try语句的一部分。注意：else部分是和try结合，而不是和if结合。我们以后会知道，在Python中，else可出现在if语句中，也可以出现在try语句以及循环中——其缩进会告诉你它属于哪个语句。

我们会在本书后面重新介绍try语句。就目前而言，只需知到，因为try可用于拦截任何错误，于是可以减少你必须所编写的错误检查代码的数量，而且这也是处理不寻常的情况的通用办法。

嵌套代码三层

现在，我们来看脚本的最后一次改进。如果有必要的话，嵌套甚至可以让我们再深入一步。例如，我们可以根据有效输入资料的相对大小，分支到一组替代动作上。

```
while True:
    reply = raw_input('Enter text:')
    if reply == 'stop':
        break
    elif not reply.isdigit():
        print 'Bad!' * 8
    else:
        num = int(reply)
        if num < 20:
            print 'low'
        else:
            print num ** 2
print 'Bye'
```

这个版本包含一个if语句，嵌套在了另一个if语句（嵌套在while循环中）的else子句

www.TopSage.com

中。当代码是条件式时，或者像这样重复时，我们只要再往右缩进即可。结果就像前几版那样，不同的是我们现在可以为小于20的数字打印“low”。

```
Enter text:19
low
Enter text:20
400
Enter text:spam
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
Enter text:stop
Bye
```

本章小结

以上内容快速浏览了Python语句的语法。本章介绍语句和代码块代码编写的通用规则。就像你所学到的，在Python中，在一般情况下是每行编写一条语句，而嵌套代码块中的所有语句都缩进相同的量（缩进是Python语法的一部分）。然而，我们也看到这些规则的一些例外情况，包括连续行以及单行测试和循环。最后，我们把这些想法落实到一个交互模式下的脚本中，来示范一些语句并在实际中展示了语句的语法。

在下一章中，我们要开始深入探讨Python的每条基本的面向过程的语句。不过，所有语句都遵循这里介绍的通用规则。

大家网
TopSage.com

本章习题

1. 类C语言中需要哪三项在Python中省略了的语法成分？
2. Python中的语句一般是怎样终止的？
3. 在Python中，嵌套代码块内的语句一般是如何关联在一起的？
4. 你怎么让一条语句跨过多行？
5. 你怎么在单个行上编写复合语句？
6. 有什么理由要在Python语句末尾输入分号呢？
7. try语句是用来做什么的？
8. Python初学者最常犯的编写代码错误是什么？

习题解答

1. 类C语言需要在一些语句中的测试两侧使用圆括号，需要在每个语句末尾有分号，以及嵌套代码块周围有大括号。
2. 一行的结尾就是该行语句的终止。此外，如果一个以上的语句出现在同行上，可以使用分号终止；同样地，如果一个语句跨过数行，可以用语法上的闭合括号终止这一行。
3. 嵌套代码块中的语句都得缩进相同数目的制表符或空格。
4. 语句可以横跨多行，只要将其封闭在圆括号内、方括号内或大括号内即可。当Python遇到一行含有一对括号中的闭合括号，语句就会结束。
5. 复合语句的主体可以移到开头行的冒号后面，但前提是主体只由非复合语句构成。
6. 只有当你需要把一列以上的语句挤进行代码时。即使是这种情况下，也只有当所有语句都是非复合时，才行得通，此外因为这样会让程序代码难以阅读，所以不建议这么做。
7. try语句是用于在Python脚本中捕捉和恢复异常（错误）的。这通常是程序中自行检查错误的方法之一。
8. 忘记在复合语句开头行末尾输入冒号，是初学者最常犯的错误。

www.TopSage.com

第11章

赋值、表达式和打印

现在，我们已经快速地介绍了Python语句的语法，这一章要开始深入具体地学习Python语句。本章从基本着手 —— 赋值语句、表达式语句和打印。本书前文已经接触过这些语句的用法，不过本章要详细介绍之前跳过的重要细节。尽管它们都相当简单，但这些语句的类型都有可选的各种形式，一旦开始撰写实际的Python程序，就会觉得很方便。

赋值语句

我们已经使用Python的赋值语句，给对象命名。其基本形式是在等号左边写赋值语句的目标，而要赋值的对象则位于右侧。左侧的目标可以是变量名或对象元素，而右侧的对象可以是任何会计算得到的对象的表达式。绝大多数情况下，赋值语句都很简单，但有些特性要专门记性，如下所示。

- **赋值语句建立对象引用值。**正如第6章讨论过的，Python赋值语句会把对象引用值储存在变量名或数据结构的元素内。赋值语句总是建立对象的引用值，而不是复制对象。因此，Python变量更像是指针，而不是数据储存区域。
- **变量名在首次赋值时会被创建。**Python会在首次将值（即对象引用值）赋值给变量时创建其变量名。有些（并非全部）数据结构元素也会在赋值时被创建（例如，字典中的元素，一些对象属性）。一旦赋值了，每当这个变量名出现在表达式时，就会被其所引用值取代。
- **变量名在引用前必须先赋值。**使用尚未进行赋值的变量名是一种错误。如果你试着这么做，Python会引发异常，而不是返回某种模糊的默认值；如果返回默认值，就很难在程序中找出输入错误的地方。
- **隐式赋值语句：import、from、def、class、for、函数参数。**本节中，我们关心的是=语句，但在Python中，赋值语句会在许多情况下使用。例如，模块导入、函

数和类的定义、for循环变量以及函数参数全都是隐式赋值运算。因为赋值语句在任何出现的地方的工作原理都相同，所有这些环境都是在运行时把变量名和对象的引用值绑定起来而已。

赋值语句的形式

虽然赋值运算是Python中通用并且一般的概念，但本章的主要内容在于赋值语句。表11-1说明Python中不同的赋值语句的形式。

表11-1：赋值语句形式

运算	解释
`spam = 'Spam'`	基本形式
`spam, ham = 'yum', 'YUM'`	元组赋值运算（位置性）
`[spam, ham] = ['yum', 'YUM']`	列表赋值运算（位置性）
`a, b, c, d = 'spam'`	序列赋值运算，通用性
`spam = ham = 'lunch'`	多目标赋值运算
`spams += 42`	增强赋值运算（相当于`spams = spams + 42`）

表11-1中的第一种形式是至今最常见的 —— 把一个变量名（或数据结构元素）绑定到单个对象上。其他的表中的项目代表了程序员通常会觉得很方便的特定的和可选的形式。

元组及列表分解赋值

表中第二和第三种形式是相关的。当在“=”左边编写元组或列表时，Python会按照位置把右边的对象和左边的目标从左至右相配对。例如，表中第二行，变量名spam赋值给字符串'yum'，而变量名ham则绑定至字符串'YUM'。从内部实现上来看，Python会先在右边制作元素的元组，所以这通常被称为元组分解赋值语句。

序列赋值语句

在最新的Python版本中，元组和列表赋值语句已统一为现在所谓的序列赋值语句的实例 —— 任何变量名的序列都可赋值给任何值的序列，而Python会按位置一次赋值一个元素。实际上，我们可以混合和比对涉及的序列类型。例如，表11-1的第四行，把变量名的元组和字符的字符串对应起来：a赋值为's'，b赋值为'p'等。

多重目标赋值

表11-1的第五行指的是多重目标形式的赋值语句。在这种形式中，Python赋值相同对象的引用值（最右边的对象）给左边的所有目标。表11-1中，变量名spam和ham两者都赋值成相同的字符串对象'lunch'的引用值。效果就好像我们写成ham =

www.TopSage.com

'lunch'，而后面再写spam = ham，这是因为ham会得到原始的字符串对象（也就是说，它并不是这个对象的独立的拷贝）。

TopSage.com

增强赋值语句

表11-1的最后一列是增强赋值语句的例子 —— 以简洁的方式结合表达式和赋值语句的简写形式。例如，“spam += 42”相当于“spam = spam + 42”，然而增强形式输入较少，而且通常执行得更快。在Python中，每个二元表达式运算符都有增强赋值语句。

序列赋值

我们已在本书中使用过基本的赋值语句。以下是一些序列分解赋值语句的简单例子。

```
% python
>>> nudge = 1
>>> wink  = 2
>>> A, B = nudge, wink              # Tuple assignment
>>> A, B                           # Like A = nudge; B = wink
(1, 2)
>>> [C, D] = [nudge, wink]         # List assignment
>>> C, D
(1, 2)
```

注意：在这个交互模式下，我们在第三行写了两个元组。其中，只是省略了它们的括号。Python把赋值运算符右侧元组内的值和左侧元组内的变量互相匹配，然后每一次赋一个值。

元组赋值语句可以得到Python中一个常用的编写代码的技巧，我们在第2部分练习题的解法中介绍过。因为语句执行时，Python会建立临时的元组，来储存右侧变量原始的值，分解赋值语句也是一种交换两变量的值，却不需要自行创建临时变量的方式：右侧的元组会自动记住先前的变量的值。

```
>>> nudge = 1
>>> wink  = 2
>>> nudge, wink = wink, nudge      # Tuples: swaps values
>>> nudge, wink                    # Like T = nudge; nudge = wink; wink = T
(2, 1)
```

事实上，Python中原始的元组和列表赋值语句形式，最后已被通用化，以接受右侧可以是任何类型的序列，只要长度相等即可。你可以将含有一些值的元组赋值给含有一些变量的列表，字符串中的字符赋值给含有一些变量的元组。在通常情况下，Python会按位置，由左至右，把右侧序列中的元素赋值给左侧序列中的变量。

```
>>> [a, b, c] = (1, 2, 3)          # Assign tuple of values to list of names
>>> a, c
```


www.TopSage.com

```
(1, 3)
>>> (a, b, c) = "ABC"                    # Assign string of characters to tuple
>>> a, c
('A', 'C')
```

从技术的角度来讲，序列赋值语句实际上支持右侧任何可迭代的对象，而不仅局限于任何序列。这是更为通用的概念，我们会在第13章和第17章介绍。

高级序列赋值语句模式

注意：虽然可以在“=”符号两侧混合和匹配的序列类型，右边元素的数目还是要跟左边的变量的数目相同，不然会产生错误。

```
>>> string = 'SPAM'
>>> a, b, c, d = string                  # Same number on both sides
>>> a, d
('S', 'M')

>>> a, b, c = string                     # Error if not
...error text omitted...
ValueError: too many values to unpack
```

想要更通用的话，就需要使用分片了。这里有几种方式使用分片运算，可以使最后的情况正常工作。

```
>>> a, b, c = string[0], string[1], string[2:]   # Index and slice
>>> a, b, c
('S', 'P', 'AM')

>>> a, b, c = list(string[:2]) + [string[2:]]    # Slice and concatenate
>>> a, b, c
('S', 'P', 'AM')

>>> a, b = string[:2]                            # Same, but simpler
>>> c = string[2:]
>>> a, b, c
('S', 'P', 'AM')

>>> (a, b), c = string[:2], string[2:]           # Nested sequences
>>> a, b, c
('S', 'P', 'AM')
```

如最后例子所示，在这个交互模式下，甚至可以赋值嵌套序列，而Python会根据其情况分解其组成部分，就像预期的一样。就此而言，我们是赋值元组中的两个项目，而第一个项目是嵌套的序列（字符串），如下例编写的一样：

```
>>> ((a, b), c) = ('SP', 'AM')           # Paired up by shape and position
>>> a, b, c
('S', 'P', 'AM')
```


www.TopSage.com

Python先把右边的第一个字符串（'SP'）和左边的第一个元组（(a, b)）配对，然后一次赋值一个字符，接着再把第二个字符串（'AM'）一次赋值给变量c。在这次事件中，左边对象的序列嵌套的形状必须符合右边对象的形状。像这种嵌套序列赋值运算是比较高级，也很少见到，但是利用已知形状取出数据结构的组成成份，也是很方便的。例如，这个技术也可以用在函数参数列表中，因为函数参数是由赋值运算传递的（本书下一部分就会看见）。

序列分解赋值语句也会产生另一种Python常见的用法，也就是赋值一系列整数给一组变量。

```
>>> red, green, blue = range(3)
>>> red, blue
(0, 2)
```

这样是把三个变量名的初始值设为整数0、1以及2（这相当于你在其他语言所见的枚举数据类型）。range内置函数会产生连续整数列表：

```
>>> range(3)                    # Try list(range(3)) in Python 3.0
[0, 1, 2]
```

因为range一般用于循环中，我们会在第13章更多关注它。另一个你会看见元组赋值语句的地方就是，在循环中把序列分割为开头和剩余两部分，如下所示。

```
>>> L = [1, 2, 3, 4]
>>> while L:
...     front, L = L[0], L[1:]
...     print front, L
...
1 [2, 3, 4]
2 [3, 4]
3 [4]
4 []
```

在这里循环的元组的赋值语句也能编写成下列两行，但放在一起使用，往往要更方便一些。

```
...     front = L[0]
...     L = L[1:]
```

注意：这个程序使用列表作为一种堆栈的数据结构。这种事我们通常也能用列表对象的append和pop方法来实现。在这里，front = L.pop(0)和元组赋值语句有相当的效果，但这是实地进行的修改。我们会在第13章多学一些关于while循环的内容，以及其他通过for循环访问序列的方式。

www.TopSage.com

多目标赋值语句

多目标赋值语句就是简单的把所有提供的变量名都赋值给右侧的对象。例如，下面把三个变量a、b和c赋值给字符串'spam'：

```
>>> a = b = c = 'spam'
>>> a, b, c
('spam', 'spam', 'spam')
```

这种形式相当于这三个赋值语句，但更为简单：

```
>>> c = 'spam'
>>> b = c
>>> a = b
```

多目标赋值以及共享引用

记住，在这里只有一个对象，由三个变量共享（全都指向内存内同一对象）。这种行为对于不可变类型而言并没问题。例如，把一组计数器初始值设为零（回想一下，变量在Python中使用前，必须先赋值才行，所以你在增加值前，必须先把计数器初始值设为零）：

```
>>> a = b = 0
>>> b = b + 1
>>> a, b
(0, 1)
```

在这里，修改b只会对b发生修改，因为数字不支持在原处的修改。只要赋值的对象是不可变的，即使有一个以上的变量名使用该对象也无所谓。

不过，就像往常一样，把变量初始值设为空的可变对象时（诸如列表或字典），我们就得小心一点：

```
>>> a = b = []
>>> b.append(42)
>>> a, b
([42], [42])
```

这一次，因为a和b引用相同的对象，通过b附加值上去，而我们通过a也会看见所有效果。这其实是我们在第6章所见共享引用值现象的另一个例子而已。为避开这个话题，要通过单独的语句赋值可变对象的初始值，使其分别执行独立的常量表达式来创建独立的空对象：

```
>>> a = []
>>> b = []
```


www.TopSage.com

```
>>> b.append(42)
>>> a, b
([], [42])
```

增强赋值语句

从Python 2.0起，表11-2所列出的一组额外的赋值语句形式就能够使用了。这些称为增强赋值语句，这是从C语言借鉴而来的，而这些格式大多数只是简写而已。也就是二元表达式和赋值语句的组合。例如，下面的两种格式现在大值相等：

```
X = X + Y                    # Traditional form
X += Y                       # Newer augmented form
```

表11-2：增强赋值语句

X += Y	X &= Y	X -= Y	X |= Y
X *= Y	X ^= Y	X /= Y	X >>= Y
X %= Y	X <<= Y	X **= Y	X //= Y

增强赋值语句适用于任何支持隐式二元表达式的类型。例如，下面是两种让变量名增加1的方式。

```
>>> x = 1
>>> x = x + 1                # Traditional
>>> x
2
>>> x += 1                   # Augmented
>>> x
3
```

用于字符串时，增强形式会改为执行合并运算。于是，在这里第二行就相当于输入较长的S = S + "SPAM"：

```
>>> S = "spam"
>>> S += "SPAM"              # Implied concatenation
>>> S
'spamSPAM'
```

如表11-2所示，每个Python二元表达式的运算符（每个运算符在左右两侧都有值），都有对应的增强赋值形式。例如，X *= Y执行乘法并赋值，X >>= Y执行向右位移并赋值。X //= Y（floor除法）则是在2.2版新增加的赋值形式。

www.TopSage.com

增强赋值语句有三个优点（注1）。

- 程序员输入减少。
- 左侧只需计算一次。在X += Y中，X可以是复杂的对象表达式。在增强形式中，则只需计算一次。然而，在完整形式X = X + Y中，X出现两次，必须执行两次。因此，增强赋值语句通常执行得更快。
- 优化技术会自动选择。对于支持原处修改的对象而言，增强形式会自动执行原处的修改运算，而不是相比来说速度更慢的复制。

在这里的最后一点需要多一点的说明。就增强赋值语句而言，在原处的运算可作为一种优化而应用在可变对象上。回想一下，列表可以用各种方式扩展。要增加单个的元素至到列表末尾时，我们可以合并或调用append。

```
>>> L = [1, 2]
>>> L = L + [3]              # Concatenate: slower
>>> L
[1, 2, 3]
>>> L.append(4)              # Faster, but in-place
>>> L
[1, 2, 3, 4]
```

此外，要把一组元素增加到末尾，我们可以再次使用合并，或者调用列表的extend方法。

```
>>> L = L + [5, 6]           # Concatenate: slower
>>> L
[1, 2, 3, 4, 5, 6]
>>> L.extend([7, 8])         # Faster, but in-place
>>> L
[1, 2, 3, 4, 5, 6, 7, 8]
```

在这两种情况下，合并相对来说不会给共享对象的引用值带来更多的副作用，但是，于等效的在原处的修改相比，往往运行起来更慢一些。合并运算必须建立的新对象，复制左侧的列表，再复制右侧的列表。与之相对比的是，在原处的修改法只会在内存块的末尾增加元素。

注1：C/C++程序员要注意：虽然Python现在支持像X +=Y这类语句，但还没有C的自动递增/递减运算符（例如，X++和--X）。这些无法对应到Python中的对象模型，因为Python没有对不可变对象进行在原处的修改的概念，例如，数字。

www.TopSage.com

当我们使用增强赋值语句来扩展列表时，可以忘记这些细节。例如，Python会自动调用较快的extend方法，而不是使用较慢的“+”合并运算（注2）。

```
>>> L += [9, 10]          # Mapped to L.extend([9, 10])
>>> L
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

TopSage.com

增强赋值以及共享引用

这种行为通常就是我们想要的，但注意，这隐含了“+=”对列表是做原处修改的意思。于是，完全不像“+”合并，总是生成新对象。就所有的共享引用情况而言，只有其他变量名引用的对象被修改，其差别才可能体现出来：

```
>>> L = [1, 2]
>>> M = L                 # L and M reference the same object
>>> L = L + [3, 4]        # Concatenation makes a new object
>>> L, M                  # Changes L but not M
([1, 2, 3, 4], [1, 2])

>>> L = [1, 2]
>>> M = L
>>> L += [3, 4]           # But += really means extend
>>> L, M                  # M sees the in-place change too!
([1, 2, 3, 4], [1, 2, 3, 4])
```

这只对于列表和字典这类可变对象才重要，而且是相当罕见的情况（至少，直到影响程序代码时才算！）。就像往常一样，如果你需要打破共享引用值的结构，就要对可变对象进行拷贝。

变量命名规则

介绍了赋值语句后，可以对变量名的使用做更正式的介绍。在Python中，当为变量名赋值时，变量名就会存在。但是，为程序中事物选择变量名时，要遵循如下规则。

语法：（下划线或字母）+（任意数目的字母、数字或下划线）

变量名必须以下划线或字母开头，而后面接任意数目的字母、数字或下划线。_spam、spam、以及Spam_1都是合法的变量名，但1_Spam、spam$以及@#!则不是。

区分大小写：SPAM和spam并不同

Python程序中区分大小写，包括创建的变量名以及保留字。例如，变量名X和x指的

注2：就像第6章的建议，我们也可以使用分片赋值语句（例如，L[len(L):] = [11,12,13]），但这种方式和一些简单的extend方法相似。

www.TopSage.com

是两个不同的变量。就可移植性而言，大小写在导入的模块文件名中也很重要，即使是在文件系统不分大小写的平台上也是如此。

禁止使用保留字

定义的变量名不能和Python语言中有特殊意义的名称相同。例如，如果使用像class这样的变量名，Python会引发语法错误，但允许使用kclass和Class作为变量名。表11-3列出当前Python中的保留字。

表11-3：Python保留字

and	elif	if	pass
as (2.6及之后版本)	else	import	print
assert	except	in	raise
break	exec	is	return
class	finally	lambda	try
continue	for	nonlocal (3.0版)	while
def	from	not	with (2.6及以后版本)
del	global	or	yield
and	elif	if	pass
as （2.6及之后版本）	else	import	print
assert	except	in	raise
break	exec	is	return
class	finally	lambda	try
continue	for	nonlocal （3.0版）	while
def	from	not	with （2.6及之后版本）
del	global	or	yield

注意：yield是Python 2.2中选用的扩展功能，但是到2.3版时则成为标准的关键词。它是和生成器函数共同使用的，我们会在第17章讨论这个新功能。这也是Python破坏了向后兼容性的一些实例之一。不过，yield已酝酿多时，先在2.2版中引起注意，而一直到2.3版才启用。

同样地，在Python 2.6中，with和as将成为新的保留字，用于环境管理器（新的异常处理形式）。这两个变量名在2.5中并不是保留字，除非你手动打开环境管理器功能（参考第27章的细节）。用于2.5版时，with和as会产生有关接即将发生改变的警告。Python 2.5版的IDLE除外，因为它似乎已替你打开这个功能（也就是说，使用这些名称做为变量名在2.5版中会产生错误，但只在IDLE版本才是这样）。

---

www.TopSage.com

Python保留字都是小写的，而且确实是保留字，不像本书下一部分介绍的内置作用域中的变量名，你无法对保留字做赋值运算（例如，and = 1会造成语法错误）（注3）。

此外，因为import语句中的模块变量名会变成脚本中的变量，这种限制也会扩展到模块的文件名：你可以写*and.py*和*my-code.py*这类文件。但是你无法将其导入，因其变量名没有“.py”扩展名时，就会变成代码中的变量。因此必须遵循刚才所提到的所有变量规则（保留字是禁区，破折号不行，不过下划线可以）。第5部分我们会再次介绍这个概念。

命名惯例

除了这些规则外，还有一组命名惯例——这些并非是必要的规则，但一般在实际中都会遵守。例如，因为变量名前后有下划线时（例如，__name__），通常对Python解释器都有特殊意义，你应该避免让变量名使用这种样式。以下是Python遵循的一些惯例。

- 以单一下划线开头的变量名（_X）不会被from module import *语句导入的（第19章说明）。
- 前后有下划线的变量名（__X__）是系统定义的变量名，对解释器有特殊意义。
- 以两下划线开头、但结尾没有两个下划线的变量名（__X）是类的本地（“压缩”）变量（第26章说明）。
- 通过交互模式运行时，只有单个下划线的变量名（_）会保存最后表达式的结果。

除了这些Python解释器的惯例外，还有各种Python程序员通常会遵循的其他惯例。例如，本书后面会看见类变量名通常以一个大字母开头，而模块变量名以小写字母开头。此外，变量名self虽然并非保留字，但在类中一般都有特殊的角色。到了第4部分，我们会研究另一种更大类型的变量名，称为内置变量名，这些是预先定义的变量名，但并非保留字（所以，可以重新赋值：open = 42行得通，不过有时你可能会希望不能这样做）。

变量名没有类型，但对象有

本部分内容算是对前文的复习，这是让Python的变量名和对象保持鲜明差异的重点所在。如第6章所介绍的，对象有类型（例如，整数和列表），并且可能是可变的或不可变的。另一方面，变量名（变量）只是对象的引用值。没有不可变的观念，也没有相关联的类型信息，除了它们在特定时刻碰巧所引用的对象的类型。

注3：　不过，在Jython版的Python实现中，用户定义的变量名偶尔可以和Python的保留字相同。

在不同时刻把相同变量名赋值给不同类型的对象，程序允许这样做：

```
>>> x = 0          # x bound to an integer object
>>> x = "Hello"    # Now it's a string
>>> x = [1, 2, 3]  # And now it's a list
```

大家网 TopSage.com

在稍后的例子中，你会看到变量名的这种通用化的本质，是Python程序设计的具有决定性的优点（注4）。在第4部分时，会介绍变量名也会存在于所谓的作用域内，也就是定义的变量名可以适用在范围赋值的变量名的所在处会决定其在哪个范围内是可见的。

表达式语句

在Python中，你也可以使用表达式作为语句（本身只占一行）。但是，因为表达式结果不会储存，只有当表达式工作并作为附加的效果，这样才有意义。通常在两种情况下表达式用作语句。

调用函数和方法

有些函数和方法会做很多工作，而不会有返回值。这种函数在其他语言中有时称为流程。因为它们不会返回你可能想保留的值，所以你可以用表达式语句调用这些函数。

在交互模式提示符下打印值

Python会在交互模式命令行中响应输入的表达式的结果。从技术上来讲，这些也是表达式语句。作为输入print语句的简写方法。

表11-4列出Python中一些常见的表达式语句的形式。函数和方法的调用是写成在函数/方法变量名后的括号内，具有零或多个参数的对象（其实，这就是计算对象的表达式）。

表11-4：常见Python表达式语句

运算	解释
spam(eggs, ham)	函数调用
spam.ham(eggs)	方法调用
spam	在交互模式解释器内打印变量
spam < ham and ham != eggs	符合表达式
spam < ham < eggs	范围测试

注4：如果你用过C++，可能会想知道，Python中并没有所谓的C++的const声明的概念。有些对象是可变的，但变量名总是可以赋值的。Python也有些方式可以在类和模块内隐藏变量名，但是和C++的声明并不相同。

www.TopSage.com

表中最后一行是特殊形式：Python让我们把大小比较测试串在一起，从而写出这样比较链的范围测试。例如，表达式（A < B < C）测试B是否界于A和C之间，相当于布尔测试（A < B and B < C）。复合表达式通常都不写成语句，但是这在语法上是合法的，而且如果你不确定表达式结果，在交互模式提示符下也很适用。

注意：虽然表达式在Python中可作为语句出现，但语句不能用作表达式。例如，Python不让你把赋值语句（=）嵌入到其他表达式中。这样做的理由是为了避免常见的编码错误。当用“==”做相等测试时，不会打成“=”而意外修改变量的值。第13章介绍Python while循环时，你会看见编写代码时如何避开这样的事。

表达式语句和在原处的修改

这里会引出Python工作中常犯的错误。表达式语句通常用于执行可于原处修改列表的列表方法：

```
>>> L = [1, 2]
>>> L.append(3)          # Append is an in-place change
>>> L
[1, 2, 3]
```

然而，Python初学者时常把这种运算写成赋值语句，试着把L赋值给更大的列表：

```
>>> L = L.append(4)      # But append returns None, not L
>>> print L              # So we lose our list!
None
```

然而，这样做是行不通的。对列表调用append、sort或reverse这类在原处的修改的运算，一定是对列表做原处的修改，但这些方法在列表修改后并不会把列表返回。

事实上，它们返回的是None对象。如果你赋值这类运算的结果给该变量的变量名，只会丢失该列表（而且可能在此过程中被当成垃圾回收）。

所以不要这样做。我们会在本书这一部分的以后章节再提醒讨论这个现象，因为这种现象也会出现在之后几章我们要谈的一些循环语句的环境中。

打印语句

print语句可以实现打印 —— 只是对程序员友好的标准输出流的接口而已。从技术角度来讲，这是把对象转换为其文本表达形式，然后发送给标准输出。

标准输出流与C语言的stdout类似；通常对应到启动Python程序所在的窗口（除非系统shell上重定向到了文件或管道中）。

在第9章中，我们看到过一些写入文本的文件方法。print语句也类似，但更为专注——print把对象写入stdout流（以及一些默认的格式），但是文件写入方法则是把字符串写入任意的文件。因为标准输出流在Python中就是内置sys模块的stdout对象（即sys.stdout），通过文件写入来模拟print是有可能的，但print更容易使用。

表11-5：print语句形式

运算	解释
print spam, ham	把对象打印至sys.stdout，在元素间增加一个空格，以及在末尾增加换行字符
print spam, ham,	一样，但是在文本末尾没有加换行字符
print >> myfile, spam, ham	把文字传给myfile.write，而不是sys.stdout.write

我们已看过基本的print语句的用法。在默认的情况下，这会在逗号相隔的项目间增加一个空格，然后在当前输出行的末尾增加一个换行字符：

```
>>> x = 'a'
>>> y = 'b'
>>> print x, y
a b
```

这种格式只是默认的，你可以选择使用或不使用。要省略换行字符（使你能在当前行后增加更多文字），print语句后可以多个逗号，如表11-5第二行所示。要省略元素间的空格，就不要以下面这种方式打印，与之相对的是，使用第7章介绍过的字符串合并和格式化工具，自己创建输出的字符串，之后一次打印出字符串：

```
>>> print x + y
ab
>>> print '%s...%s' % (x, y)
a...b
```

Python的“Hello World”程序

要打印“hello world”信息，只需打印这个字符串：

```
>>> print 'hello world'                    # Print a string object
hello world
```

因为表达式结果会响应交互模式命令行，通常是连print语句都不需要使用，只要输入要打印的表达式，而其结果就会回显：

www.TopSage.com

大家网
TopSage.com

```
>>> 'hello world'                    # Interactive echoes
'hello world'
```

其实，print语句只是Python的人性化的特性，提供了sys.stdout对象的简单接口，再加上一些默认的格式设置。实际上，如果你想写的更复杂一些，也可以用下面这种方式编写打印操作。

```
>>> import sys                       # Printing the hard way
>>> sys.stdout.write('hello world\n')
hello world
```

这段程序有意调用了sys.stdout的write方法（当Python启动连接输出流的文件对象时，这个属性就会事先设置）。print语句隐藏大多数细节，提供了简单工具从而进行简单的打印任务。

重定向输出流

那么，为什么本书要教你复杂的打印方式呢？等效的sys.stdout打印方式可以说是Python中常用技术的基础。通常来说，print和sys.stdout的关系如下。

```
print X
```

等价于：

```
import sys
sys.stdout.write(str(X) + '\n')
```

这是先通过str执行字符串转换，再通过“+”增加一行字符，最后调用输出流的write方法。作为较长的打印书写形式本身并没有什么用处。不过了解这就是print语句所做的事是有用处，因为有可能把sys.stdout重新赋值给标准输出流以外的东西。换句话说，这种等效的方式提供了一种方法，可以让print语句将文字传送到其他地方。例如：

```
import sys
sys.stdout = open('log.txt', 'a')       # Redirects prints to file
...
print x, y, x                           # Shows up in log.txt
```

在这里，我们把sys.stdout重设成已打开的文件对象（采用附加模式）。重设之后，程序中任何地方的print语句都会将文字写至文件log.txt的末尾，而不是原始的输出流。print语句将会很乐意持续地调用sys.stdout的write方法，无论sys.stdout当时引用的是什么。因为你的进程中只有一个sys模块，通过这种方式赋值sys.stdout会把程序中任何地方的每个print都进行重新定向。

事实上，就像本章接下来有关print和stdout边栏文章所做的说明，你甚至可以将sys.stdout重设为非文件的对象，只要该对象有预期的协议（write方法）。当该对象是类时，打印的文字可以定位并通过任意方式进行处理。

这种重设输出列表的技巧主要用于程序原本是用print语句编写的情况。如果你一开始就知道应该输出到文件中去，就可以改为调用文件的write方法。不过，为了将基于print的程序重定向，sys.stdout重设提供了一种除了修改每个print语句以外的便利方式，或者使用系统shell重定向语法。

print >> file扩展

通过赋值sys.stdout而将打印文字重定向的技巧，在实际中非常常用。但是，上一节代码中有个潜在的问题，那就是没有直接的方式可以保存原始的输出流。在打印至文件后，可以切换回来。因为sys.stdout只是普通的文件对象，你可以储存它，需要时恢复它（注5）。

```
>>> import sys
>>> temp = sys.stdout                    # Save for restoring later
>>> sys.stdout = open('log.txt', 'a')    # Redirect prints to a file
>>> print 'spam'                         # Prints go to file, not here
>>> print 1, 2, 3
>>> sys.stdout.close()                   # Flush output to disk
>>> sys.stdout = temp                    # Restore original stream

>>> print 'back here'                    # Prints show up here again
back here
>>> print open('log.txt').read()         # Result of earlier prints
spam
1 2 3
```

这种需求会频繁地出现，而这种手动保存和恢复原始的输出流的方法已经够复杂了。因此增加了print的扩展功能，从而没有必要非得那么做。当print语句以>>开始，后面再跟着输出的文件对象（或其他对象）时，该print语句可以将其文字传给该对象的write方法，但是不用重设sys.stdout。因为这种重定向是暂时的，普通的print语句还是会继续打印到原始的输出流的。

```
log = open('log.txt', 'a')
print >> log, x, y, z                    # Print to a file-like object
print a, b, c                            # Print to original stdout
```

注5：你也可以使用sys模块中相对较新的__stdout__属性，指的就是程序启动时sys.stdout的原始值。不过，你还是应该把sys.stdout恢复成sys.__stdout__从而回到原始的流的值。参考库手册中sys模块的更多细节。

www.TopSage.com

如果你需要在同一个程序中打印到文件以及标准输出流，print的>>形式就很方便。然而，如果你使用这种形式，要确定提供一个文件对象（或者和文件对象一样有write方法的对象），而不是文件名字符串：

```
>>> log = open('log.txt', 'w')
>>> print >> log, 1, 2, 3
>>> print >> log, 4, 5, 6
>>> log.close()
>>> print 7, 8, 9
7 8 9
>>> print open('log.txt').read()
1 2 3
4 5 6
```

这种print的扩展形式通常也用于把错误讯息打印到标准错误流sys.stderr。你可以使用其文件write方法以及自行设置输出的格式，或者使用重定向语法打印：

```
>>> import sys
>>> sys.stderr.write(('Bad!' * 8) + '\n')
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!

>>> print >> sys.stderr, 'Bad!' * 8
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
```

注意： 在Python 3.0中，print语句将变为内置函数，功用相同，但语法略有不同。目标文件和列结尾行为是由关键词参数赋值的。例如，语句print x, y会变成调用print(x, y)，而print >> f, x会变成print(x, file=f, end=' ')。

这些都还是以后版本的内容，所以要参考3.0版相关的细节。当前3.0版的计划也将引入一个转换脚本（名为“2to3”），除了必须亲手修改的细节之外，可以自动把现有代码中的print语句转成print调用。如果发现自己介于2.X和3.X之间（如本书），这个办法应该会有所帮助的。

本章小结

在这一章，我们开始探索赋值语句、表达式以及打印从而深入研究Python语句。虽然这些一般都是很容易使用的语法，但都有些可选的替代形式，在实际应用中通常都有用。例如，增强赋值语句以及print语句的重定向形式，可让我们避免一些手动的编写代码的工作。在此过程中，我们也研究了变量名的语法、流重定向技术以及各种要避免的常见错误，诸如把append方法调用的结果指回给变量等。

下一章中，我们将会加入if语句（Python主要的选择工具）的细节，来继续我们的语句之旅。我们要深入讨论Python的语法模型，看一看布尔表达式的行为。不过，在继续学习之前，本章结尾的习题会测试你在本章所学到的知识。

为什么要注意print和stdout

print语句和sys.stdout之间的等效是很重要的。这样才有可能把sys.stdout重新赋值给用户定义的对象（提供和文件相同的方法，就像write）。因为print语句只是传送文本给sys.stdout.write方法，可以把sys.stdout赋值给一个对象，而由该对象的write方法通过任意方式处理文字，通过这个对象捕捉程序中打印的文本。

例如，你可以传送打印的文字给GUI窗口，或者定义一个有write方法的对象，会做所需要的发送工作，从而可以提供给多个目的地。本书后面介绍类的时候，你会看到这个技巧的例子，它基本上看起来如下面代码段。

```
class FileFaker:
    def write(self, string):
        # Do something with the string
import sys
sys.stdout = FileFaker()
print someObjects              # Sends to class write method
```

这样行得通是因为print是本书下一部分中的所谓的多态运算：sys.stdout是什么不重要，只要有个方法（接口）称为write即可。在最新的Python版本中，这种对象的重定向甚至以print的>>扩展形式而变得更简单，因为我们不再需要刻意重设sys.stdout：

```
myobj = FileFaker()            # Redirect to an object for one print
print >> myobj, someObjects    # Does not reset sys.stdout
```

Python内置的raw_input()函数会从sys.stdin文件读入，所以你可以用类似方式拦截对读取的请求：使用类来实现类似文件的read方法。参考第10章中关于raw_input和while循环的例子。

注意：因为打印的文字进入stdout流，这也是在CGI脚本中打印HTML的方式。这也可以让你在操作系统命令行中对Python脚本的输入和输出进行像往常一样的重定向：

```
python script.py < inputfile > outputfile
python script.py | filterProgram
```


www.TopSage.com

本章习题

1. 举出三种可以把三个变量赋值成相同值的方式。
2. 将三个变量赋值给可变对象时，你可能需要注意什么？
3. L = L.sort()有什么错误？
4. 怎么使用print语句来向外部文件发送文本？

习题解答

1. 你可以使用多重目标赋值语句（A = B = C = 0）、序列赋值语句（A, B, C = 0, 0, 0）或者单独行上的多重赋值语句（A = 0, B = 0, C = 0）。就后者的技术而言，就像第10章介绍过的，你也可以把三个单独的语句用分号合并在同一行上（A = 0; B = 0; C = 0）。
2. 如果你用这种方式赋值：

```
A = B = C = []
```

这三个变量名都会引用相同对象，所以对其中一个变量名进行在原处的修改［例如，A.append(99)］也会影响其他变量名。只有对列表或辞典这类可变对象进行在原处的修改时，才会如此。对不可变对象而言，诸如数字和字符串，则不涉及此问题。

3. 列表sort方法就像append方法，也是对主体列表进行在原处的修改：返回None，而不是其修改的列表。赋值给L，会把L设为None，而不是排序后的列表。我们会在本书这一部分后面看到，新的内建函数sorted会排序任何序列，并传回具有排序结果的新列表因为这并不是在原处的修改，将其结果赋值给变量名，就又能说得通了。
4. 你可以在打印前把sys.stdout赋值给手动打开的文件，或者对于单个print语句而言，可以使用扩展的print >> file语句形式来进行打印。你也可以用系统shell的特殊语法，把程序所有的打印文字重定向，但这是在Python的范围之外的内容了。

www.TopSage.com

第12章

if测试

本章介绍Python的if语句，也就是根据测试结果，从一些备选的操作中进行选择的主要语句。因为这是我们第一次深入探索复合语句（内嵌其他语句的语句），在此，我们也会比第10章更为详细地讨论Python语句语法模型的一般概念。此外，因为if语句引入了测试的概念，本章也会处理布尔表达式，加上一些通用的真值测试的细节。

if语句

简而言之，Python if语句是选取要执行的操作。这是Python中主要的选择工具，代表Python程序所拥有的大多数逻辑。此外，这也是我们首度讨论的复合语句一样。就像所有的Python复合语句一样，if语句可以包含其他语句，包括其他if在内。事实上，Python让你在程序中按照顺序结合语句（使其逐一执行），而且可以任意地扩展嵌套（使其只在特定情况下执行）。

通用格式

Python的if语句是多数面向过程语言中的典型的if语句。其形式是if测试，后面跟着一个或多个可选的elif（“else if”）测试，以及一个最终选用的else块。测试和else部分都可以结合嵌套语句块，缩进列在首行下面。当if语句执行时，Python会执行测试第一个计算结果为真的代码块，或者如果所有测试都为假时，就执行else块。if语句的一般形式如下。

```
if <test1>:                 # if test
    <statements1>           # Associated block
elif <test2>:               # Optional elifs
    <statements2>
else:                       # Optional else
    <statements3>
```


www.TopSage.com

基本例子

为了示范，我们来看一些if语句的例子。除了开头的if测试及其相关联的语句外，其他所有部分都是选用的。在最简单的情况下，其他部分都可省略：

```
>>> if 1:
...     print 'true'
...
true
```

注意：在基本接口中，这里使用的提示符在接下去的行中变成…（在IDLE中，你只会简单地向下到缩进的行，点击Backspace可以返回）。空白行将终止并执行整个语句。记住，1是布尔真值，所以这个测试永远会成功。要处理假值结果，改写成下面程序：

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

多路分支

下面是更为复杂的if语句例子，其所有选用的部分都存在：

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

这个多行语句从if行扩展到else块。执行时，Python会执行第一次测试为真的语句底下的嵌套语句，或者如果所有测试都为假时，就执行else部分（在这个例子中，就是这样）。实际上，elif和else部分都可以省略，而且每一段中可以嵌套一个以上的语句。注意if、elif以及else结合在一起的原因在于它们垂直对齐，具有相同的缩进。

如果你用过C或Pascal这类语言，可能想知道，Python中有没有switch或case语句，可以根据变量值选择动作。然而在Python，多路分支是写成一系列的if/elif测试（如上例所示），或者对字典进行索引运算或搜索列表。因为字典和列表可在运行时创建，有时会比硬编码的if逻辑更有灵活性。

www.TopSage.com

```
>>> choice = 'ham'
>>> print {'spam':  1.25,          # A dictionary-based 'switch'
...        'ham':   1.99,          # Use has_key or get for default
...        'eggs':  0.99,
...        'bacon': 1.10}[choice]
1.99
```

第一次接触上面的程序时，需要花点时间思考，但这个字典是多路分支：根据键的选择进行索引，再分支到一组值的其中一个，很像C语言的`switch`，二者基本等效但前者比较冗长。Python `if`语句表达如下：

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

注意：在没有键相符时，这里`if`的`else`分句，就按默认的情况处理。就像我们在第8章所见到的，字典默认值能通过`has_key`测试、`get`方法调用或异常捕捉来处理。在这里也能用这些相同的技术，在字典式的多路分支中用于编写默认动作。此处是通过`get`机制处理默认值的情况。

```
>>> branch = {'spam': 1.25,
...           'ham':  1.99,
...           'eggs': 0.99}
>>> print branch.get('spam', 'Bad choice')
1.25
>>> print branch.get('bacon', 'Bad choice')
Bad choice
```

字典适用于将值和键相关联，但能够通过`if`语句来编写的更复杂动作呢？在第4部分你会学到字典也可以包含函数，从而代表更为复杂的分支动作，并实现一般的跳跃表格。这类函数作为字典的值，通常写成`lambda`，通过增加括号调用来触发其动作。

虽然字典式多路分支在处理动态数据的程序中很有用，但多数程序员可能会发现，编写`if`语句是执行多路分支最直接的方式。编写代码时的原则是，有疑虑的时候，就遵循简易性和可读性。

www.TopSage.com

Python语法规则

第10章介绍过Python的语法模型。现在，我们要上升到像if这样更大的语句，而本节是对之前介绍过的语法概念进行复习并扩展。一般来说，Python都有简单和基于语句的语法。但是，有些特性是我们需要知道的。

- **语句是逐个运行的，除非你不这样编写。**Python一般都会按照次序从头到尾执行文件中嵌套块中的语句，但是像if（还有循环）这种语句会使得解释器在程序内跳跃。因为Python经过一个程序的路径称为控制流程，像if这类会对其产生影响的语句，通常称为控制流程语句。
- **块和语句的边界会自动被检测。**就像我们所见到的，Python的程序块中没有大括号或“being/end”的分隔字符；反之，Python使用首行下的语句缩进把嵌套块内的语句组合起来。同样地，Python语句一般是不以分号终止的，一行的末尾通常就是该行所写语句的结尾。
- **复合语句=首行+“：”+缩进语句。**Python中所有复合语句都遵循相同格式：首行会以冒号终止，再接一个或多个嵌套语句，而且通常都是在首行下缩进的。缩进语句称为块（有时被称为组）。在if语句中，elif和else分句是if的一部分，也是其本身嵌套块的首行。
- **空白行、空格以及注释通常都会被忽略。**文件中空白行将被忽略（但在交互模式提示符下不会）。语句和表达式中的空格几乎都被忽略（除了在字符串实字内，以及用在缩进时）。注释总是被忽略：它们以#字符开头（不是在字符串常量内才行），而且延伸至该行的末尾。
- **文档字符串（docstring）会被忽略，但会被保存并由工具显示。**Python支持的另一种注释，称为文档字符串（简称docstring）。和#注释不同的是，文档字符串会在运行时保留下来并便于查看。文档字符串只是出现在程序文件和一些语句顶端的字符串中。Python会忽略这些内容，但是，在运行时会自动将其附加在对象上，而且能由文档工具显示。文档字符串是Python更大型的文件策略的一部分，本书这一部分最后一章会讨论它。

就像你所见到的，Python没有变量类型声明。单就这一点而言，就让你拥有比以前用过的更为简单的语言语法。但是，对于大多数新用户而言，缺少许多其他语言用于标示块和语句的大括号和分号，似乎是Python最新颖的语法特点，所以让我们更详细地讨论这方面的意义。

www.TopSage.com

代码块分隔符

Python会自动以行缩进检测块的边界，也就是程序代码左侧的空白空间。缩进至右侧相同距离的所有语句属于同一块的代码。换句话说，块内的语句会垂直对齐，就好像在一栏之内。块会在文件末尾或者碰到缩进量较少的行时结束，而更深层的嵌套块就是比所在块的语句进一步向右缩进。

例如，图12-1示范了下列程序代码的块结构。

```
x = 1
if x:
    y = 2
    if y:
        print 'block2'
    print 'block1'
print 'block0'
```

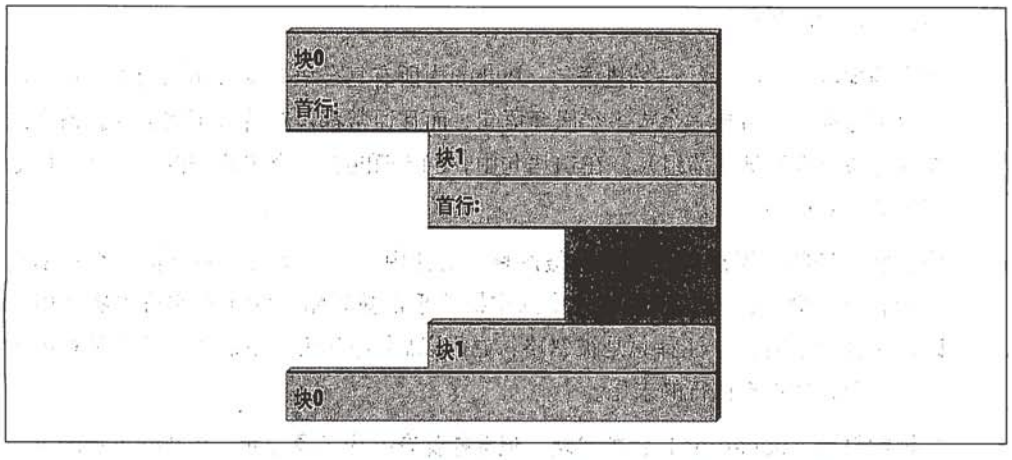


图12-1：嵌套块代码：嵌套块的语句会再往右缩进，碰到缩进量较少的语句或文件末尾时就结束

这段代码包含了三个模块：第一个（文件顶层代码）完全没有缩进，第二个（位于外层if语句内）则缩进四格，而第三个（位于嵌套if下的print语句）则缩进八格。

通常来说，顶层（无嵌套）代码必须于第1栏开始。嵌套块可以从任何栏开始。缩进可以由任意的空格和制表符组成，只要特定的单个块中的所有语句都相同即可。也就是说，Python不在乎你怎么缩进代码，只在乎缩进是否一致。从技术上来讲，制表符可看作足够的空格数，让当前栏数以8的倍数移动，但是在某一个块中混合制表符和空格，通常不是好的选择。使用其中一种就好。

www.TopSage.com

在Python中，空白重要的主要原因就在于它用于代码左侧作为缩进。在其他大多数环境中，可以在程序代码中加入空格或不加。尽管这样，缩进其实是Python语法中的一部分，而不仅仅是编程风格：任何特定单一块中的所有语句都必须缩进到相同的层次，否则Python会报告语法错误。这是有意而为之的，因为你不需明确标示嵌套代码块的开头和结尾，其他语言所见的一些语法上的杂乱无章，在Python中是看不见的。

把缩进变成语法模型一部分，也强化了一致性，这是Python这种结构化编程语言中可读性的重要组成部分。Python的语法偶尔被描述成是“所见即所得”—— 每行程序代码毫不含糊的缩进就告诉了读者它属于什么地方。这种一致的外观让Python程序更易于维护和重用。

一致性的缩进程序总是可以满足Python的规则。再者，多数文字编辑器（包括IDLE）会在输入时自动缩进程序代码来轻松地遵守Python的缩进模型。

语句的分隔符

Python的语句一般都是在其所在行的末尾结束的。不过，当语句太长、难以单放在一行时，有些特殊的规则可用于使其位于多行之中。

- **如果使用语法括号对，语句就可横跨数行。**如果在封闭的()、{}或[]这类配对中编写代码，Python就可让你在下一行继续输入语句。例如，括号中的表达式以及字典和列表常量，都可以横跨数行。语句不会结束，直到Python解释器到达（你输入闭合括号)、}或]所在的行。紧接着的行可在任何缩进层次开始，而且应该全部垂直对齐。
- **如果语句以反斜线结尾，就可横跨数行。**这是有点过时的功能，但是如果语句需要横跨数行，你也可以在前一行的末尾加上反斜线（\），以表示你要在下一行继续输入。因为也可以在较长结构两侧加上括号以便继续输入，反斜线几乎都已经不再使用了。
- **三重引号字符串常量可以横跨数行。**非常长的字符串常量可以横跨任意行数。实际上，我们在第7章看过的三重引号字符串块就是要设计成这样。
- **其他规则。**有关语句分隔字符，还有其他的重点要进行介绍。虽然不常见，但你可以用分号终止语句：这种惯例有时用于把一个以上的简单（非复合）语句挤进单个的行中。此外，注释和空白行也能出现在文件的任意之处。注释（以#字符开头）则在其出现的行的末尾终止。

www.TopSage.com

一些特殊情况

以下是使用括号配对规则让行保持连续的例子。你可以把受界线限制的内容放在任意数目的行中：

```
L = ["Good",
     "Bad",
     "Ugly"]                    # Open pairs may span lines
```

括号可以存放表达式、函数参数、函数的首行（参考第15章）等内容。如果你喜欢使用反斜线来使这一行继续也是可以的，但是这在实际的Python中并不是很常见。

```
if a == b and c == d and   \
   d == e and f == g:
   print 'olde'                 # Backslashes allow continuations...
```

因为任何表达式都可以包含在括号内，如果程序代码需要横跨数行，你通常可以改用这种技术。

```
if (a == b and c == d and
    d == e and e == f):
    print 'new'                 # But parentheses usually do too
```

另一种特殊情况是，Python允许在相同行上编写一个以上的非复合语句（语句内未嵌套其他语句），由分号隔开。有些程序员使用这种形式来节省程序文件的量，但是，如果你的多数工作都是让一个语句一行，会使程序更具可读性：

```
x = 1; y = 2; print x           # More than one simple statement
```

最后，Python可把复合语句的主体上移到首行，只要该主体只是简单（非复合）语句。简单if语句及单个测试和动作常常用到这种用法：

```
if 1: print 'hello'             # Simple statement on header line
```

你可以结合这些特殊情况来编写难读的代码，但本书不建议这么做。原则就是，试着让每条语句都在其自身的行上，除了最简单的块外，全都要缩进。六个月之后，你会庆幸你当时是这样做的。

真值测试

比较、相等以及真值的观点已经在第9章介绍过。因为if语句是我们第一次见到的实际中使用测试结果的语句，我们要在这里扩展一些概念。特别的是，Python的布尔运算符和C这类语言的布尔运算符有些不同。在Python中：

www.TopSage.com

大家网
TopSage.com

- 任何非零数字或非空对象都为真。
- 数字零、空对象以及特殊对象None都被认作是假。
- 比较和相等测试会递归地应用在数据结构中。
- 比较和相等测试会返回True或False（1和0的特殊版本）。
- 布尔and和or运算符会返回真或假的操作对象。

简而言之，布尔运算符是用于结合其他测试的结果。Python中有三种布尔表达式运算符：

`X and Y`
　　如果X和Y都为真，就是真。

`X or Y`
　　如果X或Y为真，就是真。

`not X`
　　如果X为假，那就是真（表达式返回True或False）。

此处，X和Y可以是任何真值或者返回真值的表达式（例如，相等测试、范围比较等）。布尔运算符在Python中是字（不是C的&&、||和!）。此外，布尔and和or运算符在Python中会返回真或假对象，而不是值True或False。我们来看一些例子，来了解它是怎样工作的。

```
>>> 2 < 3, 3 < 2          # Less-than: return 1 or 0
(True, False)
```

在Python中像这类值的比较会返回True或False作为其真值结果，我们已在第5章和第9章学过，但其实这些只是整数1和0的特殊版本（打印时不同，但其实完全一样）。

另一方面，and和or运算符总会返回对象，不是运算符左侧的对象，就是右侧的对象。如果我们在if或其他语句中测试其结果，总会如预期的结果那样（记住，每个对象本质上不是真就是假），但我们不会得到简单的True或False。

就or测试而言，Python会由左至右求算操作对象，然后返回第一个为真的操作对象。再者，Python会在其找到的第一个真值操作数的地方停止。这通常称为短路计算，因为求出结果后，就会使表达式其余部分短路（终止）：

```
>>> 2 or 3, 3 or 2        # Return left operand if true
(2, 3)                    # Else, return right operand (true or false)
>>> [ ] or 3
```

```
3
>>> [ ] or { }
{ }
```

上一个例子的第一行中，2和3两个操作数都是真（非零），所以Python总是在左边操作数停止并返回这个操作数。在另外两个测试中，左边的操作数为假（空对象），所以Python只会计算右边的操作数并将其返回（测试时，可能是真或假值）。

在结果知道时，and运算也会立刻停止。然而，就此而言，Python由左至右计算操作数，并且停在第一个为假的对象上：

```
>>> 2 and 3, 3 and 2      # Return left operand if false
(3, 2)                    # Else, return right operand (true or false)
>>> [ ] and { }
[ ]
>>> 3 and [ ]
[ ]
```

在这里，第一行的两个操作数都是真，所以Python会计算两侧，并返回右侧的对象。在第二个测试中，左侧的操作数为假（[]），所以Python会在该处停止并将其返回来作为测试结果。在最后测试中，左边为真（3），所以Python会计算右边的对象并将其返回（碰巧是假的[]）。

这些最终的结果都和C及其他多数语言相同：如果在if或while中测试时，你会得到逻辑真或假的值。然而，在Python中，布尔返回左边或右边的对象，而不是简单的整数标志位。

and和or这种行为，乍看之下似乎很难理解，但是，看一看本章的边框文章的例子，来了解它是如何对Python程序员的代码编写产生好处的。

if/else三元表达式

Python中布尔表达式的一种常见角色就是写个表达式，像if语句那样执行。考虑下列语句，根据X的真值把A设成Y或Z。

```
if X:
    A = Y
else:
    A = Z
```

就像这个例子所演示的，有时这类语句中涉及的元素相当简单，用在四行代码编写似乎太浪费了。在其他时候，我们可能想将这种内容嵌套在较大的语句内，而不是将其结果

www.TopSage.com

赋值给变量。因此（坦白地讲，因为C语言有类似工具）（注1），Python 2.5引入了新的表达式格式，让我们可以在一个表达式中编写出相同的结果：

```
A = Y if X else Z
```

这个表达式和先前边四行if语句的结果相同，但是更容易编写代码。就像这个语句的等效的语句，只有当X为真，Python才会执行表达式Y，而只有当X为假，才会执行表达式Z。也就是说，这是短路（short-circuit）运算，就像布尔运算符一样的行为。以下是其工作的一些例子：

```
>>> A = 't' if 'spam' else 'f'        # Nonempty is true
>>> A
't'
>>> A = 't' if '' else 'f'
>>> A
'f'
```

Python 2.5版前（以及2.5版以后，如果你坚持的话），相同的效果可以小心的用and和or运算符的结合实现，因为它们不是返回左边的对象就是返回右边的对象：

```
A = ((X and Y) or Z)
```

这样行得通，但有个问题：你得假定Y是布尔真值。如果是这样，效果就相同：and先执行，如果X为真，就返回Y；如果不是，就只返回Z。换句话说，我们得到的是“if X then Y else Z”。

第一次碰到时，这种and/or组合似乎需要“澄清时刻”才能理解，但是，2.5版时已不需要。如果你需要这种表达式，就使用更易于记忆的Y if X else Z，或者如果组成的成份并不琐碎，就使用完整的if语句。

此外，在Python中使用下列表达式也是类似的，因为bool函数会把X转换成对应的整数1或0，然后，就能用于从一个列表中挑选真假值：

```
A = [Z, Y][bool(X)]
```

例如：

```
>>> ['f', 't'][bool('')]
'f'
```

注1：事实上，Python的X if Y else Z和C的Y ? X : Z的顺序有点不同。据说这是分析Python程序代码常用的模式后的结果，但一部分也是为了减少前C程序员的过度滥用！记住，在Python中和其他地方，简单一定比复杂更好。

www.TopSage.com

```
>>> ['f', 't'][bool('spam')]
't'
```

然而，这并不完全相同，因为Python不会做短路运算，无论X值是什么，总是会执行Z和Y。因为这种复杂性，你最好还是使用Python 2.5更简单、更易懂的if/else表达式。不过，你还是应该少用，只有当组成成份都很简单时才用。否则，最好写完整的if语句，让以后的修改能简单一些。你的同事会很高兴你是这么做的。

然而，你还是会在2.5版以前的代码中看到and/or组合（以及一些还没忘记以前编写代码习惯的C程序员）。

为什么要在意布尔值

使用Python布尔运算符有些不寻常行为的常见方式就是，通过or从一组对象中做选择。像这样的语句：

```
X = A or B or C or None
```

会把X设为A、B以及C之中第一个非空（为真）的对象，或者如果所有对象都为空，就设为None。这样行得通是因为or运算符会返回两对象之一，这成为Python中相当常见的编写代码的手法：从一个固定大小的集合中选择非空的对象（只要将其串在一个or表达式中即可）。

了解短路计算也很重要，因为布尔运算符右侧的表达式可能会调用函数来执行实质或重要的工作，不然，如果短路规则生效，附加的效果就不会发生。

```
if f1() or f2(): ...
```

在这里，如果f1返回真值（非空），Python将不再会执行f2。为了保证两个函数都会执行，要在or之前调用它们。

```
tmp1, tmp2 = f1(), f2()
if tmp1 or tmp2: ...
```

你已在本章看过这个行为的另一种应用了：因为布尔运作方式，表达式((A and B) or C)几乎可用来模拟if/else语句。

此外，因为所有对象本质都是真或假，Python中，直接测试对象（if X:），而不是和空值比较（if X != ''），前者更为常见也更简单。就字符串而言，这两个测试是等效的。

www.TopSage.com

本章小结

在这一章，我们研究了Python if语句。因为这是第一个复合及逻辑语句，我们也复习了Python的一般语法规则，并比先前的更深入的探索了真值测试运算 。在此过程中，我们也看过如何在Python中编写多路分支，以及学习Python 2.5版中引进的if/else表达式。

下一章要扩展while和for循环的内容，继续探索面向过程的语句。我们会学习在Python中编写循环的各种方式，其中的一些方式胜过其他方式。不过，在那之前，先做一做本章习题吧。

本章习题

1. 在Python中怎样编写多路分支?
2. 在Python中怎样把if/else语句写成表达式?
3. 你是怎样使单个语句横跨多行的?
4. True和False这两个字代表了什么意义?

习题解答

1. if语句加多个elif分句通常是编写多路分支的最直接的方式，不过也许并不是最简明的。字典索引运算通常也能实现相同的结果，尤其是字典包含def语句或lambda表达式所写成的可调用函数。
2. 在Python 2.5中，表达式形式Y if X else Z在X为真时会返回Y，否则，则为Z。这相当于四行if语句。and/or组合((X and Y) or Z)也以相同方式工作，但更难懂，而且要求Y为真。
3. 把语句包裹在语法括号当中（()、[]、或{}），这样就可以按照需要横跨多行；当Python看见闭合括号时，语句就会结束。
4. True和False只不过分别是整数1和0的特殊版本而已。它们代表的就是Python中的布尔真假值。

www.TopSage.com

大家网
TopSage.com

第13章
while和for循环

在这一章中，我们将会遇到两个Python的主要循环结构：也就是不断重复动作的语句。首先是while语句，提供了编写通用循环的一种方法；而第二种是for语句，它是用来遍历序列对象内的元素，并对每个元素运行一个代码块。

Python中还有其他的循环运算，但这里所谈的两个语句是编写重复动作的主要语法。我们也会在这里研究一些不常用的语句（例如，break和continue），因为它们也会用在循环内。此外，本章也会探索Python迭代协议的相关概念，并增加了一些列表解析（与for循环的作用相近）的细节。

while循环

while语句是Python语言中最通用的迭代结构。简而言之，只要顶端测试一直计算到真值，就会重复执行一个语句块（通常有缩进）。称为“循环”是因为控制权会持续返回到语句的开头部分，直到测试为假。当测试变为假时，控制权会传给while块后的语句。结果就是循环主体在顶端测试为真时会重复执行，而如果测试一开始就是假，主体就绝不会执行。

while语句只是Python中两个循环语句中的一个，另一个是for语句。除了这些语句外，Python也提供一些工具，会进行隐性的迭代，包括map、reduce以及filter函数；in成员关系测试；列表解析等。我们会在第17章探讨其中的一些工具，因为它们与函数有关。

一般格式

while语句最完整的输写格式是：首行以及测试表达式、有一列或多列缩进语句的主体以及一个选用的else部分（控制权离开循环时而又没有碰到break语句时会执行）。Python会一直计算开头的测试，然后执行循环主体内的语句，直到测试返回假值为止。

```
while <test>:                    # Loop test
    <statements1>                # Loop body
else:                            # Optional else
    <statements2>                # Run if didn't exit loop with break
```

例子

为了讲清楚，我们来看一些实际中while循环的例子。第一个例子，while循环内有一个print语句，就是一直打印信息。回想一下，True只是整数1的特殊版本，总是指布尔真值；因为测试一直为真，Python会一直执行主体，或直到你停止执行为止。这种行为通常称为无限循环。

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

下个例子会不断切掉字符串第一个字符，直到字符串变空而成为假为止。像这样直接测试对象，而不是使用更冗长的等效写法（while x != ''：），可以说是一种很典型的用法。本章稍后，我们会看见用for循环更直接地遍历字符串内的元素另外的方式。注意：这里print末尾的逗号，就像第11章所学到的，会使所有输出都出现在同一行。

```
>>> x = 'spam'
>>> while x:                     # While x is not empty
...     print x,
...     x = x[1:]                # Strip first character off x
...
spam pam am m
```

下面的代码会从a的值向上计算到b的值（但不含b）。稍后，我们会看到以Python for循环和内置range函数来实现，编写更为简单：

```
>>> a=0; b=10
>>> while a < b:                 # One way to code counter loops
...     print a,
...     a += 1                   # Or, a = a + 1
...
0 1 2 3 4 5 6 7 8 9
```

注意：Python并没有其他语言中所谓的“do until”循环语句。不过我们可以在循环主体底部以一个测试和break来实现类似的功能。

```
while True:
    ...loop body...
    if exitTest(): break
```

为了完全了解这种结构的运作方式，我们进入下一节学习break语句。

www.TopSage.com

break、continue、pass和循环else

现在，我们已看过一些Python循环的例子，接下来是看两个简单的语句，只有嵌套在循环中时才有作用：break和continue语句。除了看这些不常用的语句外，我们也会在这里研究循环的else分句，因为这和break关连在一起。此外，还要学习Python的空占位语句pass。

break
: 跳出最近所在的循环（跳过整个循环语句）。

continue
: 跳到最近所在循环的开头处（来到循环的首行）。

pass
: 什么事也不做，只是空占位语句。

循环else块
: 只有当循环正常离开时才会执行（也就是没有碰到break语句）。

一般循环格式

加入break和continue语句后，while循环的一般格式如下所示。

```
while <test1>:
    <statements1>
    if <test2>: break              # Exit loop now, skip else
    if <test3>: continue           # Go to top of loop now, to test1
else:
    <statements2>                  # Run if we didn't hit a 'break'
```

break和continue可以出现在while（或for）循环主体的任何地方，但通常会进一步嵌套在if语句中，根据某些条件来采取对应的操作。

例子

我们举一些简单例子看一看在实际应用中这些语句是如何结合起来使用的吧。

pass

pass语句是无运算的占位语句，当语法需要语句并且还没有任何实用的语句可写时，就可以使用它。它通常用于为复合语句编写一个空的主体。例如，如果想写个无限循环，每次迭代时什么也不做，就写个pass。

www.TopSage.com

```
while 1: pass                    # Type Ctrl-C to stop me!
```

因为主体只是空语句，Python陷入死循环了。对语句而言的pass，差不多就像对象中的None一样，就是一个明确的什么也没有。注意：这里while循环主体和首行处在同一行上，就在冒号之后，如同if语句，只有当主体不是复合语句时，才可以这么做。

这个例子永远什么也不做。这可能不是什么有用的Python程序（除非你想在寒冷的冬天替你的笔记本暖暖机）。不过，坦率的讲，在这个时候，本书也只能举这样一个例子了。以后我们会看到它更意义的用处，例如，定义空类，而该类实现的对象行为就像其他语言的结构和记录。pass有时指的是“以后会填上”，只是暂时用于填充函数主体而已：

```
def func1():
    pass                         # Add real code here later

def func2():
    pass
```

continue

continue语句会立即跳到循环的顶端。此外，偶尔也避免语句的嵌套。下一个例子使用continue跳过奇数。这个程序代码会打印所有小于10以及大于或等于0的偶数。记住，0是假值，而%是求除法余数，所以，这个循环会倒数到0，跳过不是2的倍数的数字（它会打印出8 6 4 2 0）。

```
x = 10
while x:
    x = x-1                      # Or, x -= 1
    if x % 2 != 0: continue      # Odd? -- skip print
    print x,
```

因为continue会跳到循环的开头，所以不需要在if测试内置放print语句。只有当continue不执行时，才会运行到print。这听起来有点类似其他语言中的“goto”，的确如此。Python没有goto语句，但因为continue让程序执行时实现跳跃，有关使用goto所面临的许多关于可读性和可维护性的警告都适用。continue应该少用，尤其是刚开始使用Python的时候。例如，如果print是位于if底下，上个例子可能更清楚一些。

```
x = 10
while x:
    x = x-1
    if x % 2 == 0:               # Even? -- print
        print x,
break
```


www.TopSage.com

break语句会立刻离开循环。因为碰到break时，位于其后的循环代码都不会执行。所以有时可以引入break来避免嵌套化。例如，以下是简单的交互模式下的循环（第10章研究过的较大的例子的一个变种），通过raw_input输入数据，而当用户在请求的name处输入“stop”时就结束。

```
>>> while 1:
...     name = raw_input('Enter name:')
...     if name == 'stop': break
...     age  = raw_input('Enter age: ')
...     print 'Hello', name, '=>', int(age) ** 2
...
Enter name:mel
Enter age: 40
Hello mel => 1600
Enter name:bob
Enter age: 30
Hello bob => 900
Enter name:stop
```

注意：这个程序在计算平方前，先把年龄值通过int转换成整数。回想一下，这是有必要的。因为raw_input是以字符串返回用户输入的数据的。在第29章中，你会看到raw_input也会在文件结尾时（例如，如果用户按下Ctrl+Z或Ctrl+D）引发异常；如果这很重要，就以try语句将raw_input括起来。

else

和循环else分句结合时，break语句通常可以忽略其他语言中所需的搜索状态标志位。例如，下列程序搜索大于1的因子，来决定正整数y是否为质数。

```
x = y / 2                               # For some y > 1
while x > 1:
    if y % x == 0:                      # Remainder
        print y, 'has factor', x
        break                           # Skip else
    x = x-1
else:                                   # Normal exit
    print y, 'is prime'
```

除了设置标志位在循环结束时进行测试外，也可以在找到因子时插入break。这样一来，循环else分句可以视为只有当没有找到因子时才会执行。如果你没碰到break，该数就是质数（注2）。

注2：大概是这样。就严格的数学定义来讲，小于2的数字就不是质数了。确切地说，这个程序碰上负数和浮点数也会失败，而碰到第5章提过以后要改的/“真除法”也会有问题。如果你想实验这个程序代码，一定要看一看第4部分结尾的练习题，将其括在函数中。

www.TopSage.com

如果循环主体从没有执行过，循环else分句也会执行，因为你没在其中执行break语句。在while循环中，如果首行的测试一开始就是假，就会发生这种问题。因此，在上一个例子中，如果x一开始就小于或等于1（例如，如果y是2），你还是会得到“is prime”的信息。

关于循环else分句的更多内容

因为循环else分句是Python特有的，一些初学者容易产生困惑。简而言之，循环else分句提供了常见的编写代码的明确语法：这是编写代码的结构，让你捕捉循环的“另一条”出路，而不通过设定和检查标志位或条件。

例如，假设你要写个循环搜索列表的值，而需要知道在离开循环后该值是否已找到，可能会用这种方式编写该任务。

```
found = False
while x and not found:
    if match(x[0]):                    # Value at front?
        print 'Ni'
        found = True
    else:
        x = x[1:]                      # Slice off front and repeat
if not found:
    print 'not found'
```

在这里，我们对标志位进行初始化、设置以及稍后再进行测试，从而确认搜索是否成功。这是有效的Python程序，也的确可工作。然而这正是循环else分句所要处理的结构种类。以下是else的等效版本。

```
while x:                               # Exit when x empty
    if match(x[0]):
        print 'Ni'
        break                          # Exit, go around else
    x = x[1:]
else:
    print 'Not found'                  # Only here if exhausted x
```

这个版本要更简洁一些。标志位不见了，而我们在循环末尾使用else（和while这个关键字垂直对齐）取代了if测试。因为while主体内的break会离开循环并跳过else，因此可作为捕捉搜索失败的时况更为结构化的方式。

有些读者可能注意到，上一个例子中的else分句可以在循环后测试空x并将其取代（例如，if not x:）。虽然这个例子的确如此，但else提供了这种编码样式的明确语法（在这显然是一个搜索失败的分句），而且此种有意而为之的空测试在有些情况下并不适

www.TopSage.com

用。和for循环（下一节的主题）结合时，循环else分句甚至会变得更有用，因为序列迭代是不由你控制的。

TopSage.com

为什么要在意“模拟C 语言的while循环”

第11章讨论表达式语句那一节指出，Python不允许赋值这类语句出现在应该是表达式出现的场合。也就是说，这种常见的C语言编码样式在Python中是行不通的：

```
while ((x = next()) != NULL) {...process x...}
```

C赋值运算会返回赋值后的值，但Python赋值语句只是语句，不是表达式。这样就排除了一个众所周知的C的错误（当使用“==”时，在Python中会小心打成“=”的）。但是，如果你需要类似的行为，至少有三种方式可以在Python while循环中达到相同的效果，而不用在循环测试中嵌入赋值语句。你可以配合break，把赋值语句移到循环主体中来。

```
while True:
    x = next()
    if not x: break
    ...process x...
```

或者把赋值语句移进循环中再配合测试。

```
x = 1
while x:
    x = next()
    if x:
        ...process x...
```

或者把第一个赋值语句移出循环外。

```
x = next()
while x:
    ...process x...
    x = next()
```

这三种编码样式中，有些人认为第一种是最缺少结构化的方式，但是，这似乎是最简单的，而且也是最常用的（简单的Python for循环也可以取代一些C循环）。

for循环

for循环在Python中是一个通用的序列迭代器：可以遍历任何有序的序列对象内的元素。for语句可用于字符串、列表、元组、其他内置可迭代对象以及之后我们能够通过类所创建的新对象。

www.TopSage.com

一般格式

Python for循环的首行定义了一个赋值目标（或一些目标），以及你想遍历的对象。首行后面是你想重复的语句块（一般都有缩进）。

```
for <target> in <object>:          # Assign object items to target
    <statements>                   # Repeated loop body: use target
else:
    <statements>                   # If we didn't hit a 'break'
```

当Python运行for循环时，会逐个将序列对象中的元素赋值给目标，然后为每个元素执行循环主体。循环主体一般使用赋值的目标来引用序列中当前的元素，就好像那是遍历序列的游标。

for首行中用作赋值目标的变量名通常是for语句所在作用域中的变量（可能是新的）。这个变量名没什么特别的，甚至可以在循环主体中修改，但是，当控制权再次回到循环顶端时，就会自动被设成序列中的下一个元素。循环之后，这个变量一般都还是引用了最近所用过的元素，也就是序列中最后的元素，除非通过一个break语句退出了循环。

for语句也支持一个选用的else块，它的工作就像是在while循环中一样：如果循环离开时没有碰到break语句，就会执行（也就是序列所有元素都被访问过了）。之前介绍过的break和continue语句也可用在for循环中，就像while循环那样。for循环完整的格式如下。

```
for <target> in <object>:          # Assign object items to target
    <statements>
    if <test>: break               # Exit loop now, skip else
    if <test>: continue            # Go to top of loop now
else:
    <statements>                   # If we didn't hit a 'break'
```

例子

我们现在在交互模式下输入一些for循环，来看看在实际应用中它们是如何使用的。

基本应用

就像前边介绍的一样，for循环可以遍历任何一种序列对象。例如，在第一个例子中，我们把变量名x依序由左至右赋值给列表中三个元素的每一个，而print语句将会每个元素都执行一次。在print语句内（循环主体），变量名x引用的是列表中的当前元素。

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


www.TopSage.com

```
...
spam eggs ham
```

就像第11章所说的，print语句中末尾的逗号，可让所有这些字符串都出现在同一输出行中。

下面的两个例子会计算列表中所有元素的和与积。本章和本书后面，我们会遇见一些工具，可以自动对列表中的元素应用诸如“+”和“*”类似的运算，但是使用for循环通常也一样简单。

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

其他数据类型

任何序列都适用for循环，因它是通用的工具。例如，for循环可用于字符串和元组。

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

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

>>> for x in T: print x,                    # Iterate over a tuple
...
and I'm okay
```

实际上，稍后我们就会知道，for循环甚至可以应用在一些根本不是序列的对象上！

在for循环中的元组赋值

如果迭代元组序列，循环目标本身实际上可以是目标元组。这只是元组分解的赋值运算的另一个例子而已。记住，for循环把序列对象元素赋值给目标，而赋值运算在任何地方工作起来都是相同的。

```
>>> T = [(1, 2), (3, 4), (5, 6)]
>>> for (a, b) in T:                       # Tuple assignment at work
```


www.TopSage.com

```
...     print a, b
...
1 2
3 4
5 6
```

大家网 TopSage.com

在这里，第一次走过循环就像是编写(a,b) = (1,2)，而第二次就像是编写(a,b) = (3,4)，依次类推。这不是特殊情况，任何赋值目标在语法上都能用在for这个关键字之后。

嵌套for循环

现在，我们来学习复杂一点的for循环。下一个例子是在for中示范循环else分句以及语句嵌套。考虑到对象列表（元素）以及键列表（测试），这段代码会在对象列表中搜索每个键，然后报告其搜索结果。

```
>>> items = ["aaa", 111, (4, 5), 2.01]      # A set of objects
>>> tests = [(4, 5), 3.14]                  # Keys to search for
>>>
>>> for key in tests:                       # For all keys
...     for item in items:                  # For all items
...         if item == key:                 # Check for match
...             print key, "was found"
...             break
...     else:
...         print key, "not found!"
...
(4, 5) was found
3.14 not found!
```

因为这里的嵌套if会在找到相符结果时执行break，而循环else分句是认定如果来到此处，搜索就失败了。注意这里的嵌套。当这段代码执行时，同时有两个循环在运行：外层循环扫描键列表，而内层循环为每个键扫描元素列表。循环else分句的嵌套是很关键的，其缩进至和内层for循环首行相同的层次，所以是和内层循环相关联的（而不是if或外层for）。

注意：如果我们采用in运算符测试成员关系，这个就会比较易于编写。因为in会隐性的扫描列表来找到相符者，因此可以取代内层循环。

```
>>> for key in tests:                       # For all keys
...     if key in items:                    # Let Python check for a match
...         print key, "was found"
...     else:
...         print key, "not found!"
...
(4, 5) was found
3.14 not found!
```


www.TopSage.com

一般来说，基于对简洁和性能的考虑，让Python尽可能多做一点工作，这是个好主意，就像这个问题的解法中所展示的那样。

下一个例子以for执行典型的数据结构任务：收集两个序列（字符串）中相同元素。这差不多是简单的集合交集的例程。在循环执行后，res引用的列表中包含seq1和seq2中找到的所有元素。

```
>>> seq1 = "spam"
>>> seq2 = "scam"
>>>
>>> res = []                          # Start empty
>>> for x in seq1:                    # Scan first sequence
...     if x in seq2:                 # Common item?
...         res.append(x)             # Add to result end
...
>>> res
['s', 'a', 'm']
```

可惜的是，这个程序代码只能用在两个特定的变量上：seq1和seq2。如果这个循环可以通用化成为一种工具，可以使用多次，结果就会很棒。以后你就会知道，这个简单的想法会把我们引向函数，也就是本书下一部分的主题。

为什么要在意“文件扫描”

一般来说，每当你需要重复一个运算或重复处理某件事的时候，循环就很方便。因为文件包含许了多字符和行，它们也是循环常见的典型使用案例之一。要把文件内容一次加载至字符串，你可以调用read：

```
file = open('test.txt', 'r')
print file.read()
```

但是，要分块加载文件，通常要么是编写一个while循环，在文件结尾时使用break，要么写个for循环。要按字符读取时，下面的两种代码编写的方式都可行。

```
file = open('test.txt')
while True:
    char = file.read(1)          # Read by character
    if not char: break
    print char,

for char in open('test.txt').read():
    print char
```

这里的for也会处理每个字符，但是会一次把文件加载至内存。要以while循环按行或按块读取时，可以使用类似于下面的代码。

—待续—

www.TopSage.com

```
file = open('test.txt')
while True:
    line = file.readline()       # Read line by line
    if not line: break
    print line,

file = open('test.txt', 'rb')
while True:
    chunk = file.read(10)        # Read byte chunks
    if not chunk: break
    print chunk,
```

不过，如果是按行读取文本文件时，for循环是最易于编写以及执行最快的选择。

```
for line in open('test.txt').readlines():
    print line

for line in open('test.txt').xreadlines():
    print line

for line in open('test.txt'):
    print line-continued-
```

readlines会一次把文件载入到行字符串的列表，而xreadlines则是按需求加载文字列，从而避免大型文件导致内存溢出，最后的例子则按照文件迭代器来实现与xreadlines等效的结果（迭代器将会在下一节讨论）。在Python 2.2中，上面的变量名open也能被file取代。参考库手册中与之相关的内容。原则就是，每个步骤你所读的数据越多，你的程序就会运行得越快。

迭代器：初探

在上一节中介绍过，for循环可以用于Python中任何序列类型，包括列表、元组以及字符串，如下所示。

```
>>> for x in [1, 2, 3, 4]: print x ** 2,
...
1 4 9 16

>>> for x in (1, 2, 3, 4): print x ** 3,
...
1 8 27 64

>>> for x in 'spam': print x * 2,
...
ss pp aa mm
```

实际上，for循环甚至比这还要更为通用：可用于任何可迭代的对象。实际上，对Python

www.TopSage.com

中所有会从左至右扫描对象的迭代工具而言，这是真的，这些迭代工具包括了for循环、列表解析、in成员关系测试以及map内置函数等。

“可迭代对象”的概念在Python中是相当新颖的。基本上，这就是序列观念的通用化：如果对象是实际保存的序列，或者可以在迭代工具环境中（例如，for循环）一次产生一个结果的对象，就被看作是可迭代。总之，可迭代对象包括实际序列和按照需求而计算的虚拟序列。

文件迭代器

了解迭代器含义的最简单的方式之一就是，看一看它是如何与内置类型一起工作的，例如，文件。回想一下，已打开的文件对象有个方法名为readline，可以一次从一个文件中读取一行文本，每次调用readline方法时，就会前进到下一列。到达文件末尾时，就会返回空字符串，我们可通过它来检测，从而跳出循环。

```
>>> f = open('script1.py')
>>> f.readline()
'import sys\n'
>>> f.readline()
'print sys.path\n'
>>> f.readline()
'x = 2\n'
>>> f.readline()
'print 2 ** 33\n'
>>> f.readline()
''
```

如今，文件也有个方法，名为next，差不多有相同的效果：每次被调用时，就会返回文件中的下一行。唯一值得注意的区别在于，到达文件末尾时，next是引发内置的StopIteration异常，而不是返回空字符串。

```
>>> f = open('script1.py')
>>> f.next()
'import sys\n'
>>> f.next()
'print sys.path\n'
>>> f.next()
'x = 2\n'
>>> f.next()
'print 2 ** 33\n'
>>> f.next()
Traceback (most recent call last):
  File "<pyshell#330>", line 1, in <module>
    f.next()
StopIteration
```


www.TopSage.com

这个接口就是Python中所谓的迭代协议：有next方法的对象会前进到下一个结果，而在一系列结果的末尾时，则会引发StopIteration。在Python中，任何这类对象都认为是可迭代的。任何这类对象也能以for循环或其他迭代工具遍历，因为所有迭代工具内部工作起来都是在每次迭代中调用next，并且捕捉StopIteration异常来确定何时离开。

就像第9章所提到过的，这种魔法的效果就是，逐行读取文本文件的最佳方式就是根本不要去读取；其替代的办法就是，让for循环在每轮自动调用next从而前进到下一行。例如，下面是逐行读取文件（程序执行时打印每行的大写版本），但没有刻意从文件中读取内容：

```
>>> for line in open('script1.py'):          # Use file iterators
...     print line.upper(),
...
IMPORT SYS
PRINT SYS.PATH
X = 2
PRINT 2 ** 33
```

上例是读取文本文件的最佳方式，原因有三点：这是最简单的写法，运行最快，并且从内存使用情况来说也是最好的。相同效果的原始方式，是以for循环调用文件的readlines方法，将文件内容加载到内存，做成行字符串的列表。

```
>>> for line in open('script1.py').readlines():
...     print line.upper(),
...
IMPORT SYS
PRINT SYS.PATH
X = 2
PRINT 2 ** 33
```

这个readlines技术依然能用，但如今它已经不是最好的使用方法，而且从内存的使用情况来看，效果很差。实际上，因为这个版本其实是一次把整个文件加载到内存，如果文件太大，以至于计算机内存空间不够，甚至不能够工作。另一方面，因为一次读一行，迭代器版本对这类内存爆炸的问题就有了免疫能力。此外，基于迭代器的版本已在Python中进行了优化，所以它运行的也应该更快。

就像之前的边栏文章中所说的，也可以用while循环逐行读取文件。

```
>>> f = open('script1.py')
>>> while True:
...     line = f.readline()
...     if not line: break
...     print line.upper(),
...
...same output...
```


www.TopSage.com

尽管这样，比起迭代器for循环的版本，这可能运行得更慢一些，因为迭代器在Python中是以C语言的速度运行的，而while循环版本则是通过Python虚拟机器运行Python字节码的。任何时候，我们把Python代码换成C程序代码，速度都应该会变快。

其他内置类型迭代器

从技术角度来讲，迭代协议还有件事值得注意。当for循环开始时，会通过它传给iter内置函数，以便从可迭代对象中获得一个迭代器，返回的对象含有需要的next方法。如果我们看看for循环内部如何处理列表这类内置序列类型的话，就会变得一目了然了。

```
>>> L = [1, 2, 3]
>>> I = iter(L)                    # Obtain an iterator object
>>> I.next()                       # Call next to advance to next item
1
>>> I.next()
2
>>> I.next()
3
>>> I.next()
Traceback (most recent call last):
  File "<pyshell#343>", line 1, in <module>
    I.next()
StopIteration
```

除了文件以及列表中实际的序列外，其他类型也有其适用的迭代器。例如，遍历字典键的经典方法是明确的获取其键的列表。

```
>>> D = {'a':1, 'b':2, 'c':3}
>>> for key in D.keys():
...     print key, D[key]
...
a 1
c 3
b 2
```

不过，在最近的Python版本中，我们不再需要调用keys方法了：字典有一个迭代器，在迭代环境中，会自动一次返回一个键，所以完全不需要在内存中实际建立键列表。同样其效果就是最优化了运行速度、内存使用以及代码编写的效果。

```
>>> for key in D:
...     print key, D[key]
...
a 1
c 3
b 2
```


www.TopSage.com

其他迭代环境

到目前为止，本书是在for循环语句环境中示范的迭代器，而这也是本章的主题之一。不过，记住，每一种由左至右扫描对象的工具都会使用迭代协议。这也包括我们已见过的for循环。


```
>>> for line in open('script1.py'):          # Use file iterators
...     print line.upper(),
...
IMPORT SYS
PRINT SYS.PATH
X = 2
PRINT 2 ** 33
```

尽管如此，列表解析、in成员关系测试、map内置函数以及其他内置工具（例如，sorted和sum调用），也采用了迭代协议。


```
>>> uppers = [line.upper() for line in open('script1.py')]
>>> uppers
['IMPORT SYS\n', 'PRINT SYS.PATH\n', 'X = 2\n', 'PRINT 2 ** 33\n']

>>> map(str.upper, open('script1.py'))
['IMPORT SYS\n', 'PRINT SYS.PATH\n', 'X = 2\n', 'PRINT 2 ** 33\n']

>>> 'y = 2\n' in open('script1.py')
False
>>> 'x = 2\n' in open('script1.py')
True

>>> sorted(open('script1.py'))
['import sys\n', 'print 2 ** 33\n', 'print sys.path\n', 'x = 2\n']
```

这里所使用的map调用，我们会在本书下一部介绍，这是一种工具，对可迭代对象中的每个元素都应用一个函数调用。这类似于列表解析，但相对有些局限，因为它需要函数，而不是任意的表达式。因为列表解析和for循环有关，本章稍后还会再次对它进行讨论。

第4章看过此处所用的sorted函数。sorted是相对较新的内置函数，采用了迭代协议。这就像是原始的列表sort方法，不过是返回新的已排序列表来作为结果，而且可以应用任何可迭代的对象身上。其他较新的内置函数也支持迭代工具。例如，sum调用会计算任何可迭代对象内所有数字的和，而如果可迭代对象中的任何元素或全部元素为True、any和all内置函数就会返回True：


```
>>> sorted([3, 2, 4, 1, 5, 0])          # More iteration contexts
[0, 1, 2, 3, 4, 5]
>>> sum([3, 2, 4, 1, 5, 0])
15
```


www.TopSage.com

```
>>> any(['spam', '', 'ni'])
True
>>> all(['spam', '', 'ni'])
False
```

有趣的是，如今在Python中，迭代协议比前面例子更为普遍：Python内置工具集中每种由左至右扫描对象的工具，都定义为对主体对象使用迭代协议。这甚至还包括了比较难懂的工具，诸如list和tuple内置函数（从可迭代对象创建新对象）、字符串join方法（在可迭代对象内字符串之间放入子字符串）以及序列赋值语句等。因此，所有的这些也可以用在已打开的文件中，包括了一次自动读取一行：

```
>>> list(open('script1.py'))
['import sys\n', 'print sys.path\n', 'x = 2\n', 'print 2 ** 33\n']

>>> tuple(open('script1.py'))
('import sys\n', 'print sys.path\n', 'x = 2\n', 'print 2 ** 33\n')

>>> '&&'.join(open('script1.py'))
'import sys\n&&print sys.path\n&&x = 2\n&&print 2 ** 33\n'

>>> a, b, c, d = open('script1.py')
>>> a, d
('import sys\n', 'print 2 ** 33\n')
```

用户定义的迭代器

第17章还会配合函数再介绍迭代器，而第24章研究类时也会再谈。使用yield语句可以把用户定义的函数变成可迭代对象。如今的列表解析也可以通过生成器表达式支持此协议，而用户定义的类也能通过__iter__或__getitem__运算符重载方法而变成迭代对象。用户定义的迭代器可以在这里的任何迭代环境中使用任意对象和运算。

编写循环的技巧

for循环包括多数计数器式的循环。一般而言，for比while容易写，执行时也比较快。所以每当你需要遍历序列时，都应该把它作为首选的工具。但是，有些情况下，你需要以更为特定的方式来进行迭代。例如，如果你需要在列表中，每隔一个的元素或每隔两个元素，或者在过程中修改列表呢？如果在同一个for循环内，并行遍历一个以上的序列呢？

你可以用while循环以及手动索引运算编写这类独特的循环，但是python提供了两个内置函数，在for循环内定制迭代：

www.TopSage.com

- 内置range函数返回连续整数列表，可作为for中的索引（注3）。
- 内置zip函数返回并行的元素元组的列表，可用于在for中内遍历数个序列。

因为for循环一般都比while计数器循环运行得更快，因此如果可能的话，要尽量使用你能用的工具来获得优势。我们依次看一看这些内置函数吧。

循环计数器：while和range

range函数其实是通用的工具，可用在各种环境下。虽然range常用在for循环中来产生索引，但也可以用在任何需要整数列表的地方：

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

一个参数时，range会产生从零算起的整数列表，但其中不包括该参数的值。如果传进两个参数，第一个将视为下边界。第三个选用参数可以提供步进值。使用时，Python会对每个连续整数加上步进值从而得到结果（步进值默认为1）。range也可以是非正数或非递增的：

```
>>> range(-5, 5)
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]

>>> range(5, -5, -1)
[5, 4, 3, 2, 1, 0, -1, -2, -3, -4]
```

虽然range的结果本身都有用处，但是它们在for循环中是最常用的。至少，range提供一种简单的方法，重复特定次数的动作。例如，要打印三行时，可以使用range产生适当的整数数字。

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

注3：目前的Python也提供一个名为xrange的内置函数，会一次产生一个索引，而不是像range那样一次把全部的索引都存在列表中。xrange没有速度优势，但是如果得产生大量的数字，xrange有空间上优化，因此也有用处。然而，本书编写时，xrange似乎在Python 3.0中消失了，而range会变成生成器对象，支持迭代协议，一次产生一个元素，而不是一次全部产生放在列表中，查看3.0版的说明以了解未来的发展。

www.TopSage.com

range也常用来间接的迭代一个序列。遍历序列最简单并且是最快的方式就是使用简单的for，让Python为你处理大多数的细节。

TopSage.com

```
>>> X = 'spam'
>>> for item in X: print item,          # Simple iteration
...
s p a m
```

从内部实现上来看，for循环以这种方式使用时，会自动处理迭代的细节。如果你真的想要明确地掌控索引逻辑，可以用while循环来实现。

```
>>> i = 0
>>> while i < len(X):                   # while loop iteration
...     print X[i],; i += 1
...
s p a m
```

但是，你也可以使用for进行手动索引，也就是用range产生用于迭代的索引的列表。

```
>>> X
'spam'
>>> len(X)                              # Length of string
4
>>> range(len(X))                       # All legal offsets into X
[0, 1, 2, 3]
>>>
>>> for i in range(len(X)): print X[i],  # Manual for indexing
...
s p a m
```

这个例子是步进X的偏移值的列表，而不是X实际的元素。我们需要在循环中对X进行索引运算从而取出每个元素。

非完备遍历：range

上一节最后例子可以实现，但可能比它要求得更慢，同时也需要我们做更多的工作。除非你有特殊的索引需求，不然在可能的情况下，最好使用Python中的简单的for循环，不要用while，并且不要在for循环中使用range调用，只将其视为最后的手段。更简单的办法总是更好的。

```
>>> for item in X: print item,          # Simple iteration
...
```

然而，上一个例子中所用的编码样式可让我们做更特殊的遍历种类。例如，在遍历的过程中跳过一些元素。

www.TopSage.com

```
>>> S = 'abcdefghijk'
>>> range(0, len(S), 2)
[0, 2, 4, 6, 8, 10]

>>> for i in range(0, len(S), 2): print S[i],
...
a c e g i k
```

在这里，我们通过使用所产生的range列表，访问了字符串S中每隔一个的元素。要使用每隔两个的元素，可以把range的第三参数改为3，依此类推。实际上，通过这种方式使用range来跳过循环内的元素，依然保持了for循环的简单性。

然而，这可能不是今日Python中理想情况下最现实的技术。如果你真的想跳过序列中的元素，第7章介绍的扩展了的第三个限制值形式的分片表达式，提供了实现相同目标的更简单的办法。例如，要使用S中每隔一个的字符，可以用步进值2来分片：

```
>>> for x in S[::2]: print x
...
```

修改列表：range

可以使用range和for的组合的常见场合就是在循环中遍历列表时并对其进行修改。例如，假设你因某种理由要为列表中每个元素都加1。通过简单的for循环来做，确实会做某事，但可能并不是你想要的。

```
>>> L = [1, 2, 3, 4, 5]

>>> for x in L:
...     x += 1
...
>>> L
[1, 2, 3, 4, 5]
>>> x
6
```

这样并不行，因为修改的是循环变量x，而不是列表L。其原因有些微妙。每次经过循环时，x会引用已从列表中取出来的下一个整数。例如，第一轮中，x是整数1。下一轮中，循环主体把x设为不同对象，也就是整数2，但是并没有更新1所来自的那个列表。

要真的在我们遍历列表时对其进行修改，我们需要使用索引，让我们可以在遍历时，替每个位置赋值更新它的值。range/len组合可以替我们产生所需要的索引。

```
>>> L = [1, 2, 3, 4, 5]

>>> for i in range(len(L)):          # Add one to each item in L
...     L[i] += 1                    # Or L[i] = L[i] + 1
```


www.TopSage.com

```
...
>>> L
[2, 3, 4, 5, 6]
```

以这种方式编写时，列表会在执行循环中内容时被修改。没有办法用简单的for x in L:循环做相同的事，因为这种循环会遍历实际的元素，而不是列表的位置。但是，等效的while循环又如何呢？这种循环需要我们多做些工作，并且有可能运行得更慢。

```
>>> i = 0
>>> while i < len(L):
...     L[i] += 1
...     i += 1
...
>>> L
[3, 4, 5, 6, 7]
```

在这里，range的解法依然不理想。[x+1 for x in L]这种形式的列表解析也能做类似的工作，而且没有对最初的列表进行在原处的修改（我们可以把表达式的新列表对象赋值给L，但是这样不会更新原始列表的其他任何引用值）。因为这是循环的核心概念，本章稍后会再介绍列表解析。

并行遍历：zip和map

正如我们所见到过的，内置函数range允许我们在for循环中以非完备的方式遍历序列。本着同样的精神，内置的zip函数也让我们使用for循环来并行使用多个序列。在基本运算中，zip会取得一个或多个序列为参数，然后返回元组的列表，将这些序列中的并排的元素配成对。例如，假设我们使用两个列表：

```
>>> L1 = [1,2,3,4]
>>> L2 = [5,6,7,8]
```

要合并这些列表中的元素，我们可以使用zip来创建配对后元组的列表：

```
>>> zip(L1,L2)
[(1, 5), (2, 6), (3, 7), (4, 8)]
```

这样的结果在其他环境下也有用，然而搭配for循环时，它就会支持并行迭代。

```
>>> for (x, y) in zip(L1, L2):
...     print x, y, '--', x+y
...
1 5 -- 6
2 6 -- 8
3 7 -- 10
4 8 -- 12
```


www.TopSage.com

在这里，我们步进经过了zip调用的结果。也就是说，从两列表中提取出来的元素配对。注意：这个for循环在这里使用元组赋值运算以分解zip结果中的每个元组。第一次迭代时，就好像我们执行了赋值语句(x, y) = (1, 5)。

结果就是我们在循环中扫描L1和L2。我们也可以用while循环，手动处理索引，以达到类似的效果，但是需要更多的输入，而且可能始终比for/zip办法实现得慢。

zip函数比这个例子所建议的更为一般化。例如，zip可以接受任何类型的序列（其实就是任何可迭代的对象，包括文件），并且可以有两个以上的参数：

```
>>> T1, T2, T3 = (1,2,3), (4,5,6), (7,8,9)
>>> T3
(7, 8, 9)
>>> zip(T1,T2,T3)
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

当参数长度不同时，zip会以最短序列的长度为准来截断所得到的元组：

```
>>> S1 = 'abc'
>>> S2 = 'xyz123'
>>>
>>> zip(S1, S2)
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

相关（较旧）的内置map函数，用类似方式把序列的元素配对起来，但是如果参数长度不同，则会为较短的序列用None补齐：

```
>>> map(None, S1, S2)
[('a', 'x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None, '2'), (None,'3')]
```

这个例子其实是使用内置map函数的退化形式。正常来讲，map会带一个函数，以及一个或多个的序列参数，然后把调用该函数以及从序列中取出的并排的元素的结果收集起来。

当函数参数为None时（就像这里），就只是像zip那样把元素配对而已。map和类似的函数式工具会在第17章讨论。

使用zip构造字典

第8章介绍过，当键和值的集合必须在运行时计算时，这里所用的zip调用也可用于产生字典，并且使用非常方便。现在，我们已熟悉zip，我要说明它是如何与字典的创建有关联的。就像你所学到的，你可以编写字典常量或者不时的对键进行赋值来创建字典。

www.TopSage.com

```
>>> D1 = {'spam':1, 'eggs':3, 'toast':5}
>>> D1
{'toast': 5, 'eggs': 3, 'spam': 1}

>>> D1 = {}
>>> D1['spam']  = 1
>>> D1['eggs']  = 3
>>> D1['toast'] = 5
```

不过，如果你的程序是在脚本写好后，在运行时获得字典键和值的列表，那该怎么做呢？例如，假设你有下列的键和值的列表：

```
>>> keys = ['spam', 'eggs', 'toast']
>>> vals = [1, 3, 5]
```

将这些列表变成字典的一种做法就是将这些字符串zip起来，并通过for循环并行步进处理。

```
>>> zip(keys, vals)
[('spam', 1), ('eggs', 3), ('toast', 5)]

>>> D2 = {}
>>> for (k, v) in zip(keys, vals): D2[k] = v
...
>>> D2
{'toast': 5, 'eggs': 3, 'spam': 1}
```

不过，在Python 2.2和后续版本中，你可以完全跳过for循环，直接把zip过的键/值列表传给内置的dict构造函数。

```
>>> keys = ['spam', 'eggs', 'toast']
>>> vals = [1, 3, 5]

>>> D3 = dict(zip(keys, vals))
>>> D3
{'toast': 5, 'eggs': 3, 'spam': 1}
```

内置变量名dict其实是Python中的类型名称（在第26章，你会学到有关其他类型名称的内容，以及如何通过它们创建子类）。对它进行调用的时候，可以得到类似列表到字典的转换，但这其实是一个对象构造的请求。本章稍后，我们会探讨一个相关但更丰富的概念，也就是列表解析，来通过单个表达式建立列表。

产生偏移和元素：enumerate

之前，我们讨论过通过range来产生字符串中元素的偏移值，而不是那些偏移值处的元素。不过，在有些程序中，我们两者都需要：要用的元素以及这个元素的偏移值。从传统意义上来讲，这是简单的for循环，它同时也持有一个记录当前偏移值的计数器。

```
>>> S = 'spam'
>>> offset = 0
>>> for item in S:
...     print item, 'appears at offset', offset
...     offset += 1
...
s appears at offset 0
p appears at offset 1
a appears at offset 2
m appears at offset 3
```

可以按上面的例子做，但在最近更新的Python版本中，有个新的内置函数，名为enumerate，可以为我们做这件事。

```
>>> S = 'spam'
>>> for (offset, item) in enumerate(S):
...     print item, 'appears at offset', offset
...
s appears at offset 0
p appears at offset 1
a appears at offset 2
m appears at offset 3
```

enumerate函数返回一个生程器对象：这种对象支持本章稍早提过的迭代协议，本书下一部分会再深入讨论。这个对象有个next方法，每次遍历列表时，会返回一个(index, value)的元组 ，而我们能在for中通过元组赋值运算将其分解（很像是使用zip）：

```
>>> E = enumerate(S)
>>> E.next()
(0, 's')
>>> E.next()
(1, 'p')
```

就像往常一样，我们一般不会看到其作用的机制，这是因为迭代环境（包括列表解析，也就是下一节的主题）会自动执行迭代协议：

```
>>> [c * i for (i, c) in enumerate(S)]
['', 'p', 'aa', 'mmm']
```

列表解析：初探

在上一节中，我们学会了如何使用range，在遍历列表时对其进行修改。

```
>>> L = [1, 2, 3, 4, 5]
>>> for i in range(len(L)):
...     L[i] += 10
...
>>> L
[11, 12, 13, 14, 15]
```


www.TopSage.com

这样行得通，但就像本书所介绍的，这在Python中不是最优的“最佳实践”的实现办法。现在，列表解析使许多之前使用过的方法都过时了。例如，在这里我们可以用单个表达式取代循环，来产生所需要的结果列表。

```
>>> L = [x + 10 for x in L]
>>> L
[21, 22, 23, 24, 25]
```

结果相同，但是这种办法要求我们编写的代码更少，而且可能运行起来也快得很多。列表解析和for循环语句并不完全相同，因其创建新的列表对象（如果原始列表有多个引用值，这一点可能就很重要），但是对多数应用而言已经足够了，而且也是常见的方便的实现办法，值得在这里进一步介绍。

列表解析基础

在第4章接触过列表解析。从语法上讲，列表解析的语法是从集合理论表示法中的一种结构中衍生出来的，也就是对集合中的每个元素应用某一种运算，但不需要懂得集合理论就能运用。在Python中，多数人都会觉得列表解析看起来就像是倒过来的for循环。

我们仔细看一看上一节中的例子。列表解析是写在方括号中的，因为它毕竟是一种创建新的列表的方式。它是以我们所编写的任何的表达式开始，而该表达式中使用了一个我们所编写循环的变量（x + 10）。后面所接的就是你现在应该已经认识的for循环的首行，指出循环变量的名称以及可迭代的对象（for x in L）。

为了执行表达式，Python会在解释器内通过L来执行迭代，依次把x赋值给每个元素，然后通过左侧的表达式运行每一个元素并且存储其结果。我们所得到的结果列表就是列表解析所说的内容：新列表表，内含x + 10，而x是在L中的每个元素。

从技术角度来说，列表解析绝不是必需要求的，因为我们可以使用for循环，在遍历的过程中把表达式结果加在列表上。

```
>>> res = []
>>> for x in L:
...     res.append(x + 10)
...
>>> res
[21, 22, 23, 24, 25]
```

事实上，这正是列表解析内部所做的事。

尽管这样，列表解析写起来更简明，而且因为这种创建结果列表的编码模式在Python中是很常见的工作，因此在很多环境下使用起来都非常方便。再者，比起手动的for语

www.TopSage.com

句，列表解析运行更快（事实上，常常快大概两倍左右），因为列表解析的迭代在解释器内是以C语言的速度执行的，而不是手动编写的Python代码。对较大的数据集而言，尤其能够体现出使用它的强大的性能优势。

对文件使用列表解析

我们再看一个列表解析的常见的使用情形，从而进一步深入探索。回想一下，文件对象有一个readlines方法，会一次把文件加载成行字符串的列表。

```
>>> f = open('script1.py')
>>> lines = f.readlines()
>>> lines
['import sys\n', 'print sys.path\n', 'x = 2\n', 'print 2 ** 33\n']
```

这样可以，但是结果中的文本行在结尾处都包含了换行字符（\n）。就很多程序而言，换行字符很碍事，我们得在打印时，小心避免两行空行。如果我们可以一次去掉这些换行字符该有多好，难道不是吗？

任何时候，我们开始思考对序列中每个元素执行一种运算时，这就处在列表解析的领域内。例如，假设变量lines和之前的交互一样，则下列程序会通过字符串rstrip方法，对列表中的每一行运行rstrip，从而移除右侧的换行符，进而完成工作（line[:-1]也可以，但我们得确定所有文本行都是正确的终止才行）：

```
>>> lines = [line.rstrip() for line in lines]
>>> lines
['import sys', 'print sys.path', 'x = 2', 'print 2 ** 33']
```

这样可以，但是因为列表解析也是另一种迭代环境，就像简单的for循环，我们甚至不用事先打开文件。如果我们在表达式内开启它，则列表解析会自动使用本章之前提到过的迭代协议。也就是调用文件的next方法，一次从文件中读取一行，再通过rstrip表达式执行这行，再将其加到结果列表中。同样，我们会得到我们想要的文件中每行的rstrip结果：

```
>>> lines = [line.rstrip() for line in open('script1.py')]
>>> lines
['import sys', 'print sys.path', 'x = 2', 'print 2 ** 33']
```

这个表达式做很多隐性的工作，然而我们却毫不费力就做了很多事：Python会自动扫描文件并创建运算结果列表。这也是编写这种运算的高效的方式：因为大多数工作都是在Python解释器内做的，很有可能会比等效的for语句还要快。同样地，对大型文件而言，列表解析的速度优势尤其明显。

www.TopSage.com

扩展列表解析语法

列表解析在实际使用中可以更高级。表达式中嵌套的for循环可以结合一个if分句来过滤测试不为真的结果元素，这可是一项有用的扩展功能。

例如，我们想重复上一个例子，但我们只需收集开头为字母p的文字行（也许每行的第一个字符是某种行为代码）。加个if过滤分句到表达式中，就可以做到这一点。

```
>>> lines = [line.rstrip() for line in open('script1.py') if line[0] == 'p']
>>> lines
['print sys.path', 'print 2 ** 33']
```

在这里，if分句会检查从文件中读取到的每一行，并查看其第一个字符是否为p；如果不是，该行就会从结果列表中省略。这是相当大的表达式，但是如果我们将其转换为等效的简单for循环语句，就可轻易理解（通常来说，我们都可以把列表解析转换成for语句，也就是在通过结果附加到列表中，然后让每个连续的部分都进一步缩进）。

```
>>> res = []
>>> for line in open('script1.py'):
...     if line[0] == 'p':
...         res.append(line.rstrip())
...
>>> res
['print sys.path', 'print 2 ** 33']
```

这个for对等语句也能够工作，但是，占了四行而不是一行，而且可能慢很多。

有需要时，列表解析也能变得更复杂。例如，列表解析也可以含有嵌套循环，写成一系列for分句。事实上，其完整的语法可接纳任意数目的for分句，而每个分句都可以结合一个可选的if分句（第17章会正式讨论其语法）。

例如，下面的例子会对一个字符串中的每个x，以及另一个字符串中的每个y，创建x + y合并的列表。这其实就是在收集两个字符串中字符的排列组合：

```
>>> [x + y for x in 'abc' for y in 'lmn']
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']
```

了解这个表达式的一种方法，就是将其组成部分缩进从而形成语句形式。下面是达到相同结果的另一种等效方式，但是运行起来可能比较慢。

```
>>> res = []
>>> for x in 'abc':
...     for y in 'lmn':
...         res.append(x + y)
...
```


www.TopSage.com

```
>>> res
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']
```

不过除了它的复杂性外，列表解析表达式本身的优点也会变得过于紧凑。一般来说，列表解析都是用在简单的迭代类型上。对于涉及更广的工作来说，较简单的`for`语句结构可能理解起来更容易，而且今后修改起来也比较容易。就像往常的程序设计一样，如果有什么东西难以理解，可能就不是一个好的程序。

我们会在第17章的函数编程工具的环境中再次介绍迭代器和列表解析。与循环语句类似，它也和函数有关。

本章小结

本章，我们探索了Python的循环语句以及一些和Python循环有关的概念。我们深入讨论`while`和`for`循环语句，学习其相关的`else`分句。我们也研究过`break`和`continue`语句，而它们只在循环中才有意义。

此外，我们也对Python的迭代协议做了第一次的实质性的讨论：这是非序列对象参与迭代循环以及列表解析的方式。就像我们所见到的一样，列表解析类似于`for`循环，会将表达式施加到任何可迭代对象中的所有元素。

这就是我们对具体的面向过程的语句的学习。下一章要讨论Python代码中的文档选项，来结束本书这一部分。文档也是通用语法模型的一部分，而且是写好程序的重要元素。下一章中，我们也会做一下本书这一部分的练习题，然后再把注意力转向例如函数这样的较大的结构 。不过，就像往常一样，继续学习之前，先做一做这里的习题。

www.TopSage.com

本章习题

1. 一个循环的else分句何时执行？
2. 在Python中怎样编写一个基于计数器的循环？
3. for循环和迭代器之间有什么关系？
4. for循环和列表解析直接有什么关系？
5. 举出Python中的4种迭代环境。
6. 如今从一个文本文件逐行读取行的最好的方法是什么？
7. 你觉得 Spanish Inquisition所会采用哪些武器？

习题解答

1. while或for循环中的else分句会在循环离开时执行一次，但前提是循环是正常离开（没有运行break语句）。如果有的话，break会立刻离开循环，跳过else部分。
2. 计数器循环可以使用while语句编写，并手动记录索引值，或者以for循环写成，使用range内置函数来产生连续的整数偏移值。任何一种都不是Python中的推荐的做法，如果你只需要遍历序列中所有元素，可能时，就改用简单的for循环，不用range或计数器。这样做不仅更容易写，而且通常运行得更快。
3. for循环会使用迭代协议，步进被迭代的对象的每一个元素。for循环会在每次迭代中调用该对象的next方法，而且会捕捉StopIteration异常，从而决定何时停止循环。
4. 两者都是迭代工具。列表解析是执行常见for循环任务的简明并且高效的方法：对可迭代对象内所有元素应用一个表达式，并收集其结果。你可以把列表解析转换成for循环，而列表解析表达式的一部分的语法看起来就像是for循环的首行。
5. Python中的迭代环境包括for循环、列表解析、map内置函数、in成员关系测试表达式以及内置函数sorted、sum、any和all。这个分类也包括了内置函数list和tuple 、字符串join方法以及序列赋值运算。所有这些都使用了迭代协议（next方法）来一次一个元素逐个遍历可迭代对象 。
6. 如今从文本文件中读取文本行的最佳方式是不要刻意去读取：其替代方法是，在迭代环境中打开文件，诸如for循环或列表解析中，然后，让迭代工具在每次迭代中

www.TopSage.com

执行该文件的next方法，自动一次扫描一行。从代码编写的简易性、执行速度以及内存空间需求等方面来看，这种做法通常都是最佳方式。

7. 下列任何一种都有可能成为正确答案：使人恐惧、恐吓、漂亮的红色制服、舒适的沙发以及软枕头。

www.TopSage.com

TopSage.com

第14章 文档

本章是第3部分的最后一章，谈的是用于编写Python代码的文档的技术和工具。尽管Python代码具有可读性，但在合适的地方放一些人们可读的注释，也能很大程度上帮助其他人了解你的程序所做的工作。Python包含了可以使文档的编写变得更简单的语法和工具。

虽然这是和工具相关的概念，但这个主题会在这里介绍，一部分原因是它涉及了Python的语法模型，而另部分原因是这是努力想了解Python工具集的读者的资源。就后者这个原因而言，我会在这里说明第4章第一次所提到过的文档的向导。就像往常一样，本章的结尾包括一些常见陷阱的提醒、本章习题，以及这一部分的练习题。

Python文档资源

本书已经介绍过，Python预置的功能数量惊人：内置函数和异常、预先定义的对象属性和方法、标准库模块等。此外，我们只谈到了这几种类型的皮毛而已。

通常困扰初学者的头几个问题之一是怎么找到这些内置工具的信息。本节提示了一些Python可用的文档资源。此外，还会介绍文档字符串（docstring）以及使用它们的PyDoc系统。这些话题对核心语言本身算是外围的话题。但是，一旦你的代码到达本书这一部分的例子和练习题的水平的时候，这就变成是重要的知识了。

如表14-1所示，可以从很多地方查找Python信息，而且一般都是信息量逐渐增加。因为文档是实际编程中重要的工具，我们会在接下来几节中探讨这些类型中的每一种。

www.TopSage.com

表14-1：Python文档资源

形式	角色
#注释	文件中的文档
dir函数	对象中可用属性的列表
文档字符串：__doc__	附加在对象上的文件中的文档
PyDoc：help函数	对象的交互帮助
PyDoc：HTML报表	浏览器中的模块文档
标准手册	正式的语言和库的说明
网站资源	在线教程、例子等
出版的书籍	商业参考书籍

#注释

井字号注释是代码编写文档的最基本方式。Python会忽略#之后所有文字（只要#不是位于字符串常量中），所以你可以在这个字符之后插入一些对程序员有意义的文字和说明。不过，这类注释只能从源代码文件中看到。要编写能够更广泛的使用的注释，请使用文档字符串。

实际上，当前最佳的实践经验通常都表明，文档字符串最适于较大型功能的文档（例如，“我的文件做这些事”），而#注释最适用于较小功能的文档（例如，“这个奇怪的表达式做这些事”）。马上就会介绍文档字符串了。

dir函数

内置的dir函数是抓取对象内可用所有属性列表的简单方式（例如，对象的方法以及较简单的数据项）。它能够调用任何有属性的对象。例如，要找出标准库中的sys模块有什么可以用，可将其导入，并传给dir：

```
>>> import sys
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__',
'__stderr__', '__stdin__', '__stdout__', '_getframe', 'argv',
'builtin_module_names', 'byteorder', 'copyright', 'displayhook',
'dllhandle', 'exc_info', 'exc_type', 'excepthook',
...more names omitted...]
```

在这里只显示诸多变量名其中的一些而已。你可在机器上运行这些语句来查看完整的清单。

www.TopSage.com

要找出内置对象类型提供了哪些属性，可运行dir并传入所需要类型的常量。例如，要查看列表和字符串的属性，可传入空对象。

```
>>> dir([])
['__add__', '__class__', ...more...
'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']

>>> dir('')
['__add__', '__class__', ...more...
'capitalize', 'center', 'count', 'decode', 'encode', 'endswith',
'expandtabs', 'find', 'index', 'isalnum', 'isalpha', 'isdigit',
'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
...more names omitted...]
```

任何内置类型的dir结果都包含了一组属性，这些属性和该类型的实现相关（从技术角度来讲，就是运算符重载的方法）。它们的开头和结尾都是双下划线，从而保证了其独特性，而在本书这个时候你可以将其忽略掉。此外，你也可以把类型的名称传给dir（而不是常量），依然可以得到相同的结果。

```
>>> dir(str) == dir('')          # Same result as prior example
True
>>> dir(list) == dir([])
True
```

这样做行得通是因为像str和list这类函数以前曾经是类型转换器，而如今实际上已经是Python的类型的名称。调用其中的一个名称，会启用其建构函数，从而产生了该类型的实例。我会在第6部分讨论类时，再介绍构造函数和运算符重载方法。

dir函数可作为记忆提醒器，提供属性名称的列表，但并没有告诉那些名称的意义是什么。就这些额外的信息来说，我们需要继续学习下一个文档的资源。

文档字符串：__doc__

除了#注释外，Python也支持可自动附加在对象上的文档，而且在运行时还可保存查看。从语法上来说，这类注释是写成字符串，放在模块文件、函数以及类语句的顶端，就在任何可执行程序代码前（#注释在其前也没问题）。Python会自动封装这个字符串，也就是成为所谓的文档字符串，使其成为相应对象的__doc__属性。

用户定义的文档字符串

例如，考虑下面的文件*docstrings.py*。其文档字符串出现在文件开端以及其中的函数和类的开头。在这里，文件和函数多行注释使用的是三重引号块字符串，但是任何类型的

www.TopSage.com

字符串都能用。我们还没详细研究def或class语句，所以，除了它们顶端的字符串外，其他关于它们的内容都可以忽略。

TopSage.com

```
"""
Module documentation
Words Go Here
"""

spam = 40

def square(x):
    """
    function documentation
    can we have your liver then?
    """
    return x **2

class employee:
    "class documentation"
    pass

print square(4)
print square.__doc__
```

这个文档协议的重点在于，注释会保存在__doc__属性中以供查看（文件被导入之后）。因此，要显示这个模块以及其对象打算关联的文档字符串，我们只需要导入这个文件，简单的打印其__doc__属性（即Python储存文本的地方）即可：

```
>>> import docstrings
16

    function documentation
    can we have your liver then?

>>> print docstrings.__doc__

Module documentation
Words Go Here

>>> print docstrings.square.__doc__

    function documentation
    can we have your liver then?

>>> print docstrings.employee.__doc__
    class documentation
```

注意：一般都需要明确说出要打印的文档字符串，否则你会得到嵌有换行字符的单个字符串。你也可以把文档字符串附加到类的方法中（以后会谈），但是因为这些只是嵌套在类中的def语句，所以也不是什么特别的情况。要取出模块中类的方法函数的文档字符

www.TopSage.com

串，可以通过路径访问类：module.class.method.__doc__（参考第25章的方法的文档字符串的例子）。

文档字符串标准

文档字符串的文字应该有什么内容，并没有什么标准（不过有些公司有内部标准）。现在已经有各种标记语言和模板协议（例如，HTML或XML），但是，似乎没有在Python世界中流行起来。然而，坦率的讲，要说服程序员使用手动编写HTML为代码编写文档，那是不可能的！

通常来说，文档在程序员之间的优先级都偏低。而一般情况下，如果你看到文件中有任何注释，那都已经算是幸运了。不过，本书强烈建议你详细地为代码编写文档，这其实是写好代码的重要部分。这里的重点就是，目前文档字符串的结构没有标准。如果你想用，就别犹豫。

内置文档字符串

Python中的内置模块和对象都使用类似的技术，在dir返回的属性列表前后加上文档。例如，要查看内置模块的可读的说明时，可将其导入，并打印其__doc__字符串。

```
>>> import sys
>>> print sys.__doc__
This module provides access to some objects
used or maintained by the interpreter and to
...more text omitted...

Dynamic objects:

argv -- command line arguments; argv[0] is the script pathname if known
path -- module search path; path[0] is the script directory, else ''
modules -- dictionary of loaded modules
...more text omitted...
```

内置模块内的函数、类以及方法在其__doc__属性内也有附加的说明信息。

```
>>> print sys.getrefcount.__doc__
getrefcount(object) -> integer

Return the current reference count for the object.
...more text omitted...
```

也可以通过文档字符串读取内置函数的说明。

```
>>> print int.__doc__
int(x[, base]) -> integer
```

```
Convert a string or number to an integer, if possible.
...more text omitted...

>>> print open.__doc__
file(name[, mode[, buffering]]) -> file object

Open a file.  The mode can be 'r', 'w' or 'a' for reading
...more text omitted...
```

可以用这种方式查看其文档字符串，从而得到内置工具的大量信息，但是你不必这样做：下一节的主题help函数会为你自动做这件事。

PyDoc：help函数

文档字符串技术已被证明是实用的工具，Python现在配备了一个工具，使其更易于显示。标准PyDoc工具是Python程序代码，知道如何提取文档字符串并且自动提取其结构化的信息，并将其格式化成各种类型的排列友好的报表。

有很多种方式可以启动PyDoc，包括命令行脚本选项（参考Python库手册的细节）。也许两种最主要的PyDoc接口是内置的help函数，以及PyDoc GUI/HTML接口。help函数会启用PyDoc从而产生简单的文字报表（看起来就像是类UNIX系统上的“manpage”）。

```
>>> import sys
>>> help(sys.getrefcount)
Help on built-in function getrefcount:

getrefcount(...)
    getrefcount(object) -> integer

    Return the current reference count for the object.
    ...more omitted...
```

注意：调用help时，不会用导入sys，但是要取得sys的辅助信息时，就得导入sys，help期待有个对象的引用值传入。就较大对象而言，诸如，模块和类，help显示内容会分成几段，而其中有一些会在这里显示。通过交互模式运行它，来查看完整的报表。

```
>>> help(sys)
Help on built-in module sys:

NAME
    sys

FILE
    (built-in)

DESCRIPTION
    This module provides access to some objects used
```


www.TopSage.com

```
    or maintained by the interpreter and to functions
    ...more omitted...

FUNCTIONS
    __displayhook__ = displayhook(...)
        displayhook(object) -> None

        Print an object to sys.stdout and also save it
    ...more omitted...
DATA
    __name__ = 'sys'
    __stderr__ = <open file '<stderr>', mode 'w' at 0x0082BEC0>
    ...more omitted...
```

这个报表中的信息有些是文档字符串，而有些（例如，函数调用模式）是PyDoc自动查看对象内部而收集的结构化信息。你也可以对内置函数、方法以及类型使用help。要取得内置类型的help信息，就使用其类型名称（例如，字典为dict，字符串为str，列表为list）。你会得到大量的显示内容，说明该类型可用的方法。

```
>>> help(dict)
Help on class dict in module __builtin__:

class dict(object)
 |  dict() -> new empty dictionary.
 ...more omitted...

>>> help(str.replace)
Help on method_descriptor:

replace(...)
    S.replace (old, new[, maxsplit]) -> string

    Return a copy of string S with all occurrences
    ...more omitted...

>>> help(ord)
Help on built-in function ord:

ord(...)
    ord(c) -> integer

    Return the integer ordinal of a one-character string.
```

最后，help函数也能用在模块上，就像内置工具一样。在这里是对之前所写的docstrings.py文件生成报表。同样地，其中有些是文档字符串，而有些是查看对象的结构而自动取出的信息。

```
>>> help(docstrings.square)
Help on function square in module docstrings:
```


www.TopSage.com

```
square(x)
    function documentation
    can we have your liver then?

>>> help(docstrings.employee)
...more omitted...

>>> help(docstrings)
Help on module docstrings:

NAME
    docstrings

FILE
    c:\python22\docstrings.py

DESCRIPTION
    Module documentation
    Words Go Here

CLASSES
    employee
    ...more omitted...

FUNCTIONS
    square(x)
        function documentation
        can we have your liver then?

DATA
    __file__ = 'C:\\PYTHON22\\docstrings.pyc'
    __name__ = 'docstrings'
    spam = 40
```

PyDoc：HTML报表

在交互模式下工作时，help函数是获取文档的好帮手。然而，想要更宏伟的显示的话，PyDoc也提供GUI接口（简单并且可移植的Python/Tkinter脚本），可以将其报表通过HTML网页格式来呈现，可通过任何网页浏览器来查看。在这种模式下，PyDoc可以在本地运行，或者作为客户端/服务器模式中的远程服务器来运行。报表中会包含自动创建的超链接，让你能够点击应用程序中相关组件的文档。

要通过这种模式启动PyDoc，一般是先启动图14-1所示的搜索引擎GUI。你可以选择Windows Python的“Start”按钮中的“Module Docs”菜单来启动它，或者启动Python *Tools*目录下的*pydocgui.pyw*脚本（执行pydoc.py再带一个-g命令行参数也行）。输入你感兴趣的模块名称，然后按下Enter键。Python会深入到模块的导入搜索路径（sys.path）从而寻找所请求的模块的索引内容。

www.TopSage.com

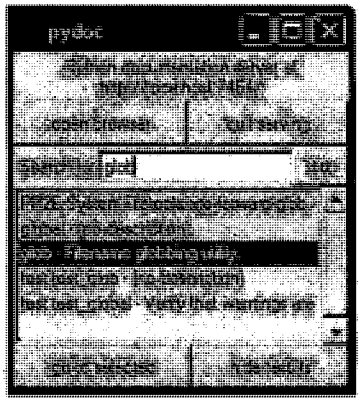


图14-1：Pydoc顶层搜索引擎GUI：输入你想找的模块名称，再按下Enter键，选择该模块，然后按下“go to selected”（或者不使用模块名称，而是按下“open browser”来查看所有可用的模块）

一旦你找到对象，选取它，再点击“go to selected”。Python会在机器上打开网页浏览器，以HTML格式显示报表。图14-2显示的是内置的glob模块的PyDoc信息。

注意这个网页“Module”部分中的超链接：你可以点击这些超链接从而跳到相关（已导入）模块的PyDoc网页。就较大的网页而言，PyDoc也会产生超链接从而指向网页的不同部分。

就像help函数接口，GUI接口也能用在用户定义的模块上。图14-3显示的是针对我们的*docstrings.py*模块文件所产生的网页。

PyDoc能以许多方式调整和启动，我们在这里不讨论。参考Python标准库手册中的内容来了解更多的细节。最后要提的是，PyDoc基本上是“免费”实现报表：如果你善于在文件中使用文档字符串，PyDoc会替你收集信息并排列其格式以便于显示。PyDoc只能帮助函数和模块这类东西，但是，提供一种简单的方式来读取这类工具的中级文档；其报表比单纯的属性列表更有用，但是，也比不上标准手册那么完整。

www.TopSage.com

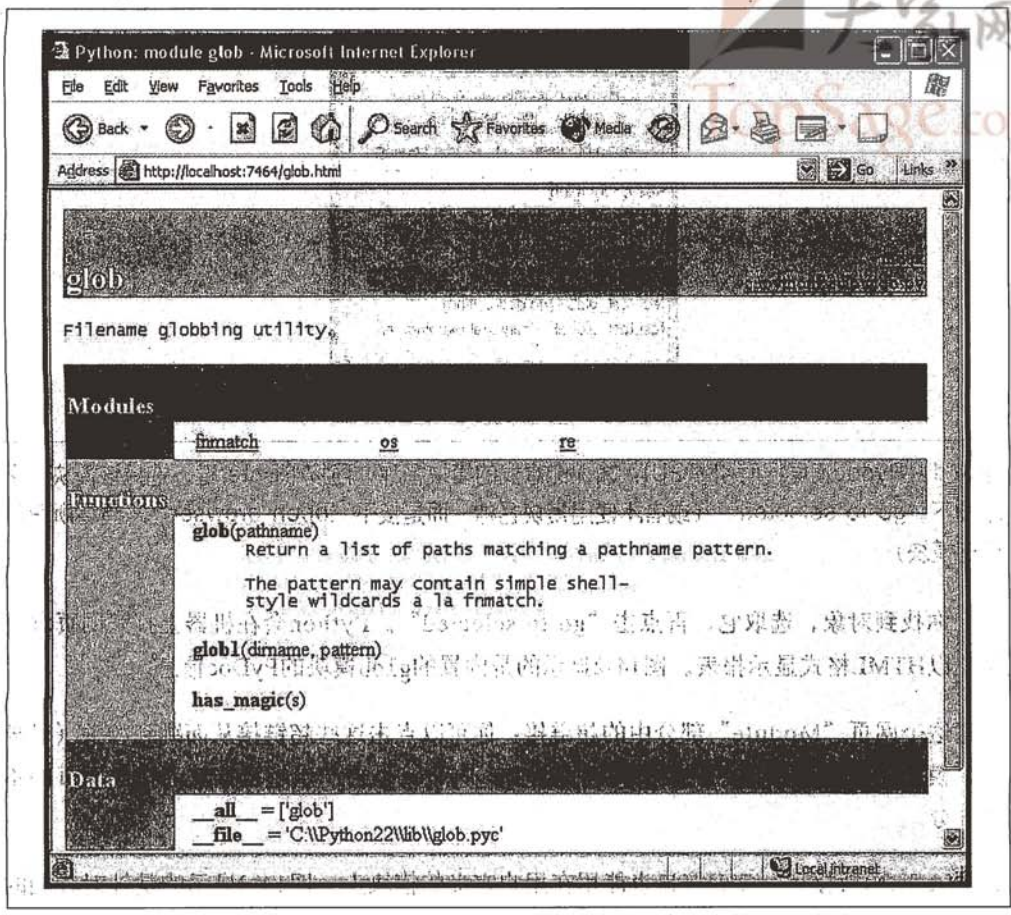


图14-2：当你在图14-1 GUI中找到一个模块并按下“go to selected”时，模块的文档会以HTML呈现，并显示在网页浏览器窗口中，就像这里所展示的：这是内置的标准库模块

注意：PyDoc技巧：如果你在图14-1窗口中的顶端输入字段中让模块名称留空，然后按下“Open Browser”按钮，PyDoc会产生一个网页，其中包含了可能在计算机上导入的每个模块的超链接。这包括Python标准库模块、已安装的第三方扩展模块、位于导入搜索路径上的用户定义模块以及静态或动态连结的C程序模块。如果没有编写程序去查看模块的源代码，是很难获得这类信息的。

PyDoc也能把模块的HTML文档保存在文件中，以便在今后查看或打印。参考其文档来了解如何使用。此外，注意：如果对象是从标准输入读取数据的脚本，PyDoc可能无法很好的运行。Python会导入目标模块来查看其内容，然而以GUI模式执行时，可能和标准输入文字没有连结。不过，可以导入但无需立即输入所需要的模块，也可以在PyDoc中运行得很好。

---

www.TopSage.com

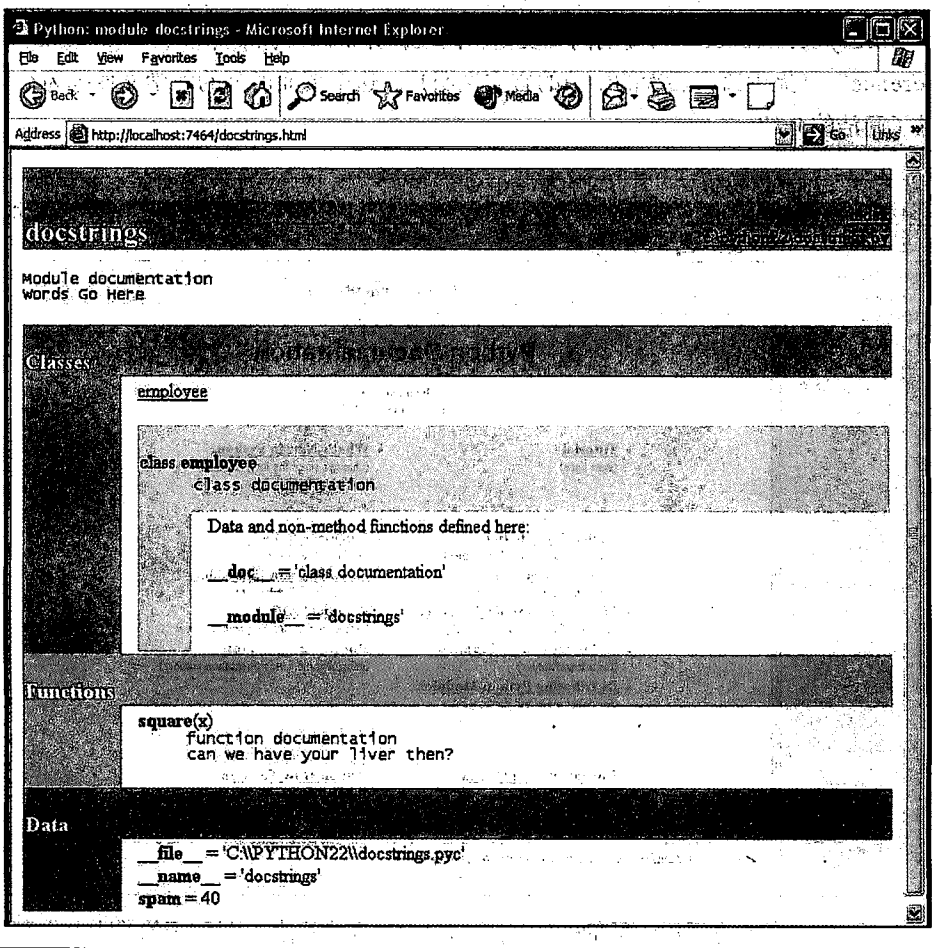


图14-3：PyDoc可作为显示内置和用户定义的模块的文档页。这里的网页是用户定义模块的说明网页，显示了它从源代码中提取出的所有文档字符串（文档字符串）

标准手册集

为了获得语言以及工具集最新的完整说明，Python标准手册随时可以提供支持。Python手册以HTML和其他格式来实现，在Windows上是随着Python系统安装：可以从“开始”按钮的Python选单中选取，而且也可以在IDLE的“Help”选项菜单中开启。你也可以从*http://www.python.org*获得不同格式的手册，或者在该网站上在线阅读（接着Documentation链接）。在Windows上，手册是编译了的帮助文件，支持搜索，而Python.org的在线版本还包括一个搜索页面。

www.TopSage.com

开启时， Windows格式的手册会显示像图14-4那样的根页面。这里最重要的两个项目是“Library Reference”（说明内置类型、函数、异常以及标准库模块）和“Language Reference”（提供语言层次的细节的官方说明）。这个页面所列的教学文件也为初学者提供了简洁的介绍。

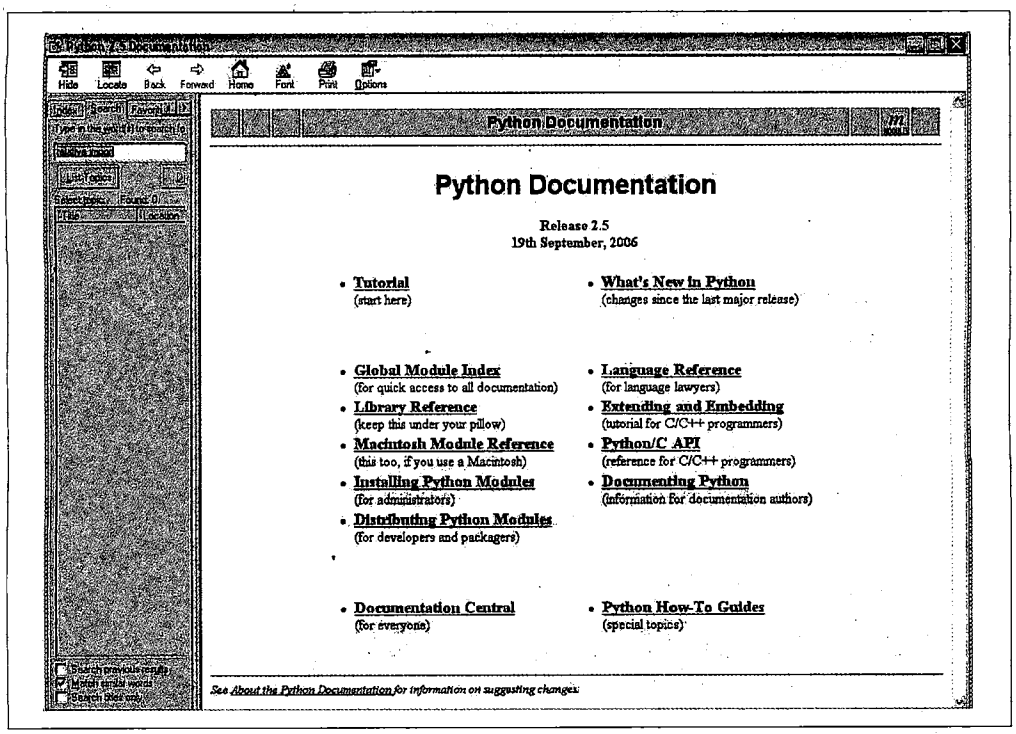


图14-4：Python的标准手册，可在www.python.org上在线阅读，从IDLE的“Help”菜单启动，以及从Windows的“开始”按钮菜单启动 。这是Windows上可搜索的辅助文件，而且在线版本也有搜索引擎。其中，Library手册是大多数实际最常使用的工具之一

网络资源

在官方的Python程序设计语言网站上（*http://www.python.org*），你会发现各种Python资源的链接，而其中一些涵盖了特定的主题或领域。点击Documentation链接可以获取在线教程以及“Beginners Guide to Python”。这个网站也列出了非英文的Python资源。

你会在如今的Web上发现各种Python维基、博客、网站以及其他许多资源。要寻找在线社群，可试着在Google中搜索“Python programming”。

www.TopSage.com

已出版的书籍

你可以从大量Python参考书籍中做选择，来作为最终的资源。记住，书籍会比Python最新的变动慢得多，一部分原因是因为这项工作涉及写作，另部分原因是因为出版的周期原本就比较迟缓。通常来说，当书籍问世时，会比当前Python的状态慢个三个月或更长时间。和标准手册不同的是，书籍一般也不是免费的。

然而，对多数人而言，专业出版的书籍的方便性和质量，是值得购买的。再者，Python的变动很慢，书籍在出版几年后，依然可用，特别是作者还在网站上更新的话。参考序文中有关其他Python书籍的指南。

常见编写代码的陷阱

在做本书这一部分的练习题前，我们来看一下初学者编写Python语句和程序最常犯的一些错误。很多都是本书之前已经提出过的警告，写在这里只是为了方便参考而已。一旦你有一些Python代码的编写经验后，就会懂得避开这些陷阱，但是现在一些话可能有帮助你避免一开始就掉在这些陷阱中。

- **别忘了冒号。**一定要记住在复合语句首行末尾输入“:”（if、while、for等的第一行）。你可能一开始会忘记（我就忘过，过去几年我那3,000位Python学生多数也会），但是，这很快就会变成无意识的习惯，所以你大可放心。
- **从第1行开始。**要确定顶层（无嵌套）程序代码从第1行开始。这包括在模块文件中输入的无嵌套的代码，以及在交互模式提示符下输入的无嵌套的代码。
- **空白行在交互模式提示符下很重要。**模块文件中复合语句内的空白行都会被忽视，但是，当你在交互模式提示符下输入代码时，空白行则是会结束语句。换句话来说，空白行是告诉交互模式命令行，你已完成复合语句；如果你想继续，就不要在…提示符下（或IDLE中）按Enter键，直到完成为止。
- **缩进要一致。**避免在块缩进中混合制表符和空格，除非你知道文字编辑器如何处理制表符。否则，如果编辑器把制表符也算成空格，你在编辑器中所见到的就不一定是Python所见到的。对任何块结构的语言来说都是如此，不仅仅是Python而已：如果下一位程序员对制表符有不同的设置，他就无法了解代码的结构。每个块全都使用制表符或空格，这样比较安全。
- **不要在Python中写C代码。**给C/C++程序员的提醒：在`if`和`while`首行，不用再测试两侧输入括号（例如，`if (X==1):`）。如果你喜欢，你也可以这么做（任何表达式都可包含在括号中），但是在这种环境下完全是多余的。此外，不要以分号终止

www.TopSage.com

所有的语句。在Python中，这么做在技术上也是合法的。但是完全没用，除非把一个以上的语句放在同一行中（每行的结尾通常就是该语句的终结）。此外，记住不要在while循环测试中嵌入赋值语句，而且不要在块周围使用{}（改为一致地缩进嵌套程序代码块）。

- **使用简单的for循环，而不是while或range。**另一件要提醒的事：比起while或者range式的计数器循环来讲，简单的for循环（例如，for x in seq:）总是比较容易写，运行起来也更快。因为Python会在内部为简单的for循环处理索引运算，因此有时会比等效的while快两倍。避免在Python中做计算的诱惑！

- **要注意赋值语句中的可变对象。**在第11章介绍过：在多重目标赋值语句中（a = b = []），以及在增强指定语句中（a += [1, 2]），使用可变对象时，要小心一点。在这两种情况下，在原处的修改会影响其他变量。参考第11章的细节。

- **不要期待进行在原处的修改的函数会返回结果。**我们以前也碰过这一点：像第8章介绍过的list.append和list.sort方法这种的修改运算，并不会有返回值（除了None）。所以在调用时不要对其结果进行赋值。初学者写出mylist = mylist.append(X)这样的语句，试着取得append的结果，结果却实际把mylist指定为None，而不是修改后的列表，这种事并非不常见（事实上，你会完全失去该列表的引用值）。

  当你尝试以排序的方式遍历字典元素的时候，会有更复杂的例子，例如，for k in D.keys().sort():这类代码。用：keys方法建立键列表，而sort方法可以用来排序。但是因为sort方法返回None，循环就失败了，因为最后变成一个None（而不是序列）的循环。要正确编写这段代码，可以使用较新的sorted内置函数，来返回排序后的列表，也可以把方法调用放在外边：先执行Ks = D.keys()，然后执行Ks.sort()，最后执行k in Ks:。这是你想明确调用keys方法来进行循环运算的一种情况，而不是依靠字典迭代器，迭代器不会排序。

- **一定要使用括号调用函数。**必须在函数名称后面加括号才能对它进行调用，无论它是否带有参数［例如，使用function()，而不是function］。在第4部分中，你会发现，函数也是对象，只是有特殊的运算——你通过括号触发对它的调用。

  从分类上看，这个问题似乎在文件上最常发生；初学者经常输入file.close来关闭文件，而不是file.close()。因为引用函数而不是对它调用也是合法的，第一个没有括号的版本也会成功，但是它并没有关闭文件。

- **不要在导入和重载中使用扩展名或路径。**在import语句中省略目录路径和文件字尾（例如，要写import mod，而不是import mod.py）。（我们在第3章讨论过模块的基础，而第5部分会继续研究模块）。因为模块可能有.py以外的其他后缀名（例

www.TopSage.com

如，*.pyc*），硬编码的后缀名不仅是不合法的语法，也说不通。任何平台特定的目录路径语法是属于模块搜索路径设置的，而不是import语句。

本章小结

本章带我们进行了程序的文档概念之旅：我们为程序编写的文档，以及内置工具的文档。我们见到了文档字符串，探索过Python的在线手册等参考资源，并且学习了PyDoc的help函数和网页接口是如何提供额外的文档来源的。因为这是本书这一部分的最后一章，我们也复习了常见的编写代码的错误，从而有助于你避开这些陷阱。

本书下一部分要把所学到的一切应用到较大程序结构：函数。然而，继续学习之前，先做一下本章结尾处第3部分的练习题。但在那之前，先来做本章习题。

www.TopSage.com

大家网
TopSage.com

本章习题

1. 在什么时候应该使用文档字符串而不是#字注释？
2. 举出三种查看文档字符串的方式。
3. 如何获得对象中可用属性的列表？
4. 如何获得计算机中所有可用模块的列表？
5. 本书之后，你应该买哪本Python书籍？

习题解答

1. 文档字符串（文件字符串）被认为最适用于较大、功能性的文档，用来描述程序中的模块、函数、类以及方法的使用。如今的#号注释最好只限于关于费解的表达式或语句的微型文档。一方面因为文件字符串在源代码文件中比较容易找到，另一方面也是因为PyDoc系统能将其取出并显示。
2. 你可以打印对象的`__doc__`属性，传给PyDoc的`help`函数，以及选取服务器/客户端模式下PyDoc GUI搜索引擎中的模块，查看文档字符串。此外，PyDoc可以把模块的文档储存在HTML文件中以便稍后查看或打印。
3. 内置的`dir(X)`函数会返回附加在任何对象上的所有属性的列表。
4. 执行PyDoc GUI接口，保持模块名称空白，然后选择“Open Browser”。这样会打开一个网页，其中包含了程序中每个可用模块的链接。
5. 当然，我的书（确切地说，前言列出一些我所推荐的一些进阶书籍，包括了参考书籍和应用程序开发的教程）。

www.TopSage.com

第三部分练习题

现在，你知道了怎样去编写基本程序逻辑，下面的练习题会要求你使用语句实现一些简单的任务。多数工作在练习题4，让你探索代码编写的各种替代方法。此外，还有很多种方式可以安排语句，并且学习Python的一部分内容就是学习什么样的安排要比其他的更好。

参考附录B“第3部分 语句和语法”的解答。

1. **编写基本循环。**

   a. 写个for循环，打印字符串S中每个字符的ASCII码。使用内置函数ord(character)把每个字符转换成ASCII整数（在交互模式下测试来观察其工作方式）。

   b. 接着，修改循环来计算字符串中所有字符的ASCII码的总和。

   c. 最后，再次修改代码，来返回一个新的列表，其中包含了字符串中每个字符的ASCII码。表达式map(ord, S)是否有类似的效果？（提示：参考第4部分）。

2. **反斜线字符。**当在交互模式下输入下面的代码时，你的机器上会发生什么？

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

要注意，如果是在IDLE接口外执行，这个例子可能会发出蜂鸣声，所以，你可能不想在有很多人的实验室里执行。IDLE会改为打印奇怪的字符（参考表7-2的反斜线转义字符）。

3. **排序字典。**在第8章中，我们知道字典是无序集合体。编写一个for循环来按照排序后（递增）顺序打印字典的项目。提示：使用字典keys和列表sort方法，或者较新的sorted内置函数。

4. **程序逻辑替代方案。**考虑下列代码，使用while循环以及found标志位来搜索2的幂值列表[到2的5次方（32）]。它保存在名为*power.py*的模块文件内。

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

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


www.TopSage.com

```
    else:
        i = i+1

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

C:\book\tests> python power.py
at index 5
```

这个例子并没有遵循一般的Python代码编写的技巧。遵循这里所提到的步骤来改进它（就所有的转变而言，你可以在交互模式下输入代码，或者将其保存在脚本文件中从系统命令行来运行，使用文件会让这个练习更容易一些）：

a. 首先，以while循环else分句重写这个代码来消除found标志位和最终的if语句。

b. 接着，使用for循环和else分句重写这个例子，去掉列表索引运算逻辑。提示：要取得元素的索引，可以使用列表index方法（L.index(X)返回列表L中第一个X的偏移值）。

c. 接着，重写这个例子，改用简单的in运算符成员关系表达式，从而完全移除循环（参考第8章的细节，或者以这种方式来测试：2 in [1,2,3]）。

d. 最后，使用for循环和列表append方法来产生2列表（L），而不是通过列表常量硬编码。

深入思考：

e. 把2 ** X表达式移到循环外，这样能够改善性能吗？你要怎样编写代码？

f. 就像我们练习题1中所看到过的，Python有一个map(function, list)工具也可以产生2次方值的列表：map(lambda x: 2 ** x, range(7))。试着在交互模式下输入这段代码；我们将会在第17章正式引入lambda。

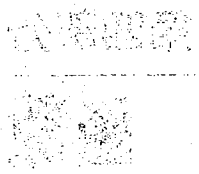
www.TopSage.com

大家网
TopSage.com

第四部分

函数

www.TopSage.com



www.TopSage.com

第15章

函数基础

在第三部分，我们学了Python中一些简单的流程语句。这里，我们将会继续学习更多的语句，让我们自己也能够创建函数。

简而言之，一个函数就是将一些语句集合在一起的部件，它们能够不止一次地在程序中运行。函数还能够计算出一个返回值，并能够改变作为函数输入的参数，而这些参数在代码运行时也许每次都不相同。以函数的形式去编写一个操作能够使它成为一个能够广泛应用的工具，让我们在不同的情形下都能够使用它。

从更一般的情况来说，函数是替代在编程过程中不断的拷贝粘贴的一个方法：而不是有多次的多余的拷贝来进行一个操作的代码，我们能够将其包括在一个函数之中。通过这样做，我们可以大大减少今后的工作：如果这个操作之后必须要修改，我们只需要修改其中的一份拷贝，而不是所有代码。

函数是Python为了代码最大程度的重用和最小化代码冗余而提供的最基本的程序结构。正如我们将看到的一样，函数也是一种设计工具，它让我们将复杂的系统分解为可管理的部件。表15-1总结了本书这一部分中我们将会学习到的主要的与函数相关的工具。

表 15-1：函数相关的语句和表达式

语句	例子
Calls	`myfunc("spam", "eggs", meat=ham)`
`def, return, yield`	`def adder(a, b=1, *c):` `  return a+b+c[0]`
`global`	`changer():` `  global x; x = 'new'`
`lambda`	`Funcs = [lambda x: x**2, lambda x: x*3]`

www.TopSage.com

为何使用函数

在学习具体内容之前，我们先了解函数的概念。函数是一个通用的程序结构部件。你也许已经在其他的编程语言中见到过，也常被称作子例程或过程。作为简明的介绍，函数主要扮演了两个角色。

最大化的代码重用和最小化代码冗余

和现代编程语言中一样，Python的函数是一种简单的办法去打包逻辑算法，使其能够在之后不止在一处不止一次地使用。直到现在，我们所写的代码都是立即运行的。函数允许整合以及通用化代码，以便这些代码能够在之后多次使用。因为它们允许一处去编写多处运行，Python的函数是这个语言中最基本的组成工具 —— 它让我们在程序中减少代码的冗余成为现实，并为代码的维护节省了不少的力气。

流程的分解

函数也提供了一种将一个系统分割为定义完好的不同部分的工具。例如，去做一份披萨，开始需要混合面粉，将面粉搅拌匀，增加顶部原料和烤等。如果你是在编写一个制作披萨的机器人的程序，函数将会将整个“做披萨”这个任务分割成为独立的函数来完成整个流程中的每个子任务。独立的实现较小的任务要比一次完成整个流程要容易得多。一般来说，函数讲的是流程：告诉你怎样去做某事，而不是让你使用它去做的事。我们将会在第4部分了解函数重要的原因。

在本部分，我们将会探索在Python中编写函数所使用到的工具：函数的基本概念，作用域以及参数传递，还有一些相关的概念，例如，生成器和函数式工具。由于多态的重要性在目前这个编程水平逐渐显得重要起来，我们也会重新回顾本书前面提到的多态。正如你将会看到的那样，函数并没有用到太多的新语法，但是它们带给了我们很多的编程方面的启示。

编写函数

尽管没有正式介绍，我们在前面已经接触了一些函数。例如，为了创建文件对象，我们调用了内置函数`open`。同样地，我们使用了内置函数`len`去得到一个集合对象的元素的数目。

在这一章，我们将会解释在Python中如何去编写一个函数。我们编写的函数使用起来就像内置函数一样：它们通过表达式进行调用，传入一些值，并返回结果。但是编写一个新的函数要求我们使用一些尚未介绍过的额外的概念。此外，函数在Python中同在像C

www.TopSage.com

这样的编译语言中表现非常不同。下面是一个关于Python函数背后的一些主要概念的简要介绍，我们都会在本书这一部分学习。

- **def是可执行的代码。**Python的函数是由一个新的语句编写的，即def。不像C这样的编译语言，def是一个可执行的语句 —— 函数并不存在，直到Python运行了def后才存在。事实上，在if语句、while循环甚至是其他的def中嵌套是合法的（甚至在某些场合还很有效）。在典型的操作中，def语句在模块文件中编写，并自然而然地在模块文件第一次被导入的时候生成定义的函数。
- **def创建了一个对象并将其赋值给某一变量名。**当Python运行到def语句时，它将会生成一个新的函数对象并将其赋值给这个函数名。就像所有的赋值一样，函数名变成了某一个函数的引用。函数名其实并没有什么神奇 —— 就像你将看到的那样，函数对象可以赋值给其他的变量名，保存在列表之中。函数也可以通过lambda表达式（在稍后章节会介绍的高级概念）来创建。
- **return将一个结果对象发送给调用者。**当函数被调用时，其调用者停止运行直到这个函数完成了它的工作，之后函数才将控制权返回调用者。函数是通过return语句将计算得到的值传递给调用者的，返回值成为函数调用的结果。像生成器这样的函数也可以通过yield语句来返回值，并挂起它们的状态以便稍后能够恢复状态。这是本书稍后要介绍的另一个高级话题。
- **函数是通过赋值（对象引用）传递的。**在Python中，参数通过赋值传递给了函数（也就是说，就像我们所学过的，使用对象引用）。正如你将看到的那样，Python的模块并不是真正的等效于C中的传递规则或是C++中的引用参数 —— 调用者以及函数通过引用共享对象，但是不需要别名。改变参数名并不会改变调用者中的变量名，但是改变传递的可变对象可以改变调用者共享的那个对象。
- **global声明了一个模块级的变量并被赋值。**在默认情况下，所有在一个函数中被赋值的对象，是这个函数的本地变量，并且仅在这个函数运行的过程中存在。为了分配一个可以在整个模块中都可以使用的变量名，函数需要在global语句中将它列举出来。在通常的情况下，变量名往往需要关注它的作用域（也就是说变量存储的地方），并且是通过实赋值语句将变量名绑定至作用域的。
- **参数、返回值以及变量并不是声明。**就像在Python中所有的一样，在函数中并没有类型约束。实际上，从一开始函数就不需要声明：可以传递任意类型的参数给函数，函数也可以返回任意类型的对象。其结果就是，函数常常可以用在很多类型的对象身上，任意支持兼容接口（方法和表达式）的对象都能使用，无论它们是什么类型。

www.TopSage.com

如果说前面有些内容介绍的不够深入的话，请别担心 —— 我们将会在本书的这一部分通过真实的代码去探索所有以上的这些概念。让我们开始展开介绍前面这些概念并看一些例子。

def语句

def语句将创建一个函数对象并将其赋值给一个变量名。Def语句一般的格式如下所示。

```
def <name>(arg1, arg2,... argN):
    <statements>
```

就像所有的多行Python语句一样，def包含了首行并有一个代码块跟随在后边，这个代码块通常都会缩进（或者就是在冒号后边简单的一句）。而这个代码块就成为了函数的主体 —— 也就是每当函数被调用时Python所执行的语句。

def的首行定义了函数名，赋值给了函数对象，并在括号中包含了0个或以上的参数（有些时候称为是形参）。在函数被调用的时候，在首行的参数名赋值给了括号中的传递来的对象。

函数主体往往都包含了一个return语句。

```
def <name>(arg1, arg2,... argN):
    ...
    return <value>
```

Python的return语句可以在函数主体中的任何地方出现。它表示函数调用的结束，并将结果返回至函数调用处。return语句包含一个对象表达式，这个对象给出的函数的结果。return语句是可选的。如果它没有出现，那么函数将会在控制流执行完函数主体时结束。从技术角度来讲，一个没有返回值的函数自动返回了none对象，但是这个值是往往被忽略掉的。

函数也许会有yield语句，这在每次都会产生一系列值时被用到，这在第17章我们研究函数的高级话题时才会讨论到。

def语句是实时执行的

Python的def语句实际上是一个可执行的语句：当它运行的时候，它创建并将 一个新的函数对象赋值给一个变量名。（请记住， Python中所有的语句都是实时运行的，没有像独立的编译时间这样的流程的）因为它是一个语句，一个def可以出现在任一语句可以出现的地方 —— 甚至是嵌套在其他的语句中。例如，尽管def往往是包含在模块文件中，

www.TopSage.com

并在模块被导入时运行，函数还是可以通过嵌套在if语句中去实现不同的函数定义，这样也是完全合法的。

TopSage.com

```
if test:
    def func():          # Define func this way
        ...
else:
    def func():          # Or else this way
        ...
...
func()                   # Call the version selected and built
```

它在运行时简单的给一个变量名进行赋值。与C这样的编译语言不同，Python函数在程序运行之前并不需要全部定义。更确切地讲，def在运行时才进行评估，而在def之中的代码在函数调用后才会评估。

因为函数定义是实时发生的，所以对于函数名来说并没有什么特别之处。关键之处在于函数名所引用的那个对象。

```
othername = func          # Assign function object
othername()               # Call func again
```

这里，将函数赋值给一个不同的变量名，并通过新的变量名进行了调用。就像Python中其他语句的一样，函数仅仅是对象，在程序执行时它清楚地记录在了内存之中。

第一个例子：定义和调用

除了像运行概念之外（对于有着传统编译语言的程序员来说，这看起来很比较特殊），Python函数用起来还是很直接的。让我们编写第一个真实的例子来说明这些基础知识。正如你将看到的，函数描绘了两个方面：定义（def创建了一个函数）以及调用（表达式告诉Python去运行函数主体）。

定义

这是一个在交互模式下输入的定义语句，它定义了一个名为times的函数，这个函数将返回两个参数的乘积。

```
>>> def times(x, y):          # Create and assign function
...     return x * y          # Body executed when called
...
```

当Python运行到这里并执行了def语句时，它将会创建一个新的函数对象，封装这个函数的代码并将这个对象赋值给变量名times。很典型，这样一个语句编写在一个模块文件

www.TopSage.com

之中，当这个文件导入的时候运行。在这里，对于这么小的一个程序，用交互提示模式已经足够了。

调用

在def运行之后，可以在程序中通过在函数名后增加括号调用（运行）这个函数。括号中可以包含一个或多个对象参数，这些参数将会传递（赋值）给函数头部的参数名。

```
>>> times(2, 4)                # Arguments in parentheses
8
```

这个表达式传递了两个参数给times函数。就像在前边提到过的那样，参数是通过赋值传递的。因此，在这个例子中，在函数头部的变量x被赋值为2，y被赋值为4，之后函数的主体开始运行。对于这个函数，其主体仅仅是一条return语句，这条语句将会返回结果作为函数调用表达式的值。在这里返回的对象将会自动打印出来（就像在大多数语言一样，在Python中2*4的结果为8），但是，如果稍后需要使用这个值，我们可以将其赋值给另一个变量。例如：

```
>>> x = times(3.14, 4)         # Save the result object
>>> x
12.56
```

现在，看看函数在第三次被调用时将会发生什么吧，这次我们将会传递两个完全不同种类的对象：

```
>>> times('Ni', 4)             # Functions are "typeless"
'NiNiNiNi'
```

这次，函数的作用完全不同（Monty Python 再次被引用）。在这第三次调用中，将一个字符串和一个整数传递给x和y，而不是两个数字。“*”对数字和序列都有效。因为在Python中，我们从未对变量、参数或者返回值有过类似的声明，我们可以把times用作数字的乘法或是序列的重复。

换句话说，函数times的作用取决于传递给它的值。这是Python中的核心概念之一（也是使用Python的诀窍之一），下一部分再学习这些内容。

Python中的多态

就像我们看到的那样，times函数中表达式x*y的意义完全取决于x和y的对象类型，同样的函数，在一个实例下执行的是乘法，在另一个实例下执行的却是赋值。Python将对某

www.TopSage.com

一对象在某种语法的合理性交由对象自身来判断。实际上，“*”在针对正被处理的对象进行了随机应变。

这种依赖类型的行为称为多态，我们在第4章介绍过这个术语，其含义就是一个操作的意义取决于被操作对象的类型。因为Python是动态类型语言，所以多态在Python中随处可见。实际上，在Python中每个操作都是多态的操作：print、index、*操作符，还有很多。

这实际上是有意而为的，并且从很大程度上算作是这门语言的简易性和灵活性的一个表现。作为函数，例如，它可以自动地适用于所有类别 的对象类型。只要对象支持所预期的接口（a.k.a. protocol），那么函数就能处理。也就是说，如果对象传给函数的对象有预期的方法和表达式操作符，那么它们对于函数的逻辑来说就是有着即插即用的兼容性的。

即使是简单的`times`函数，任意两个支持*的对象都可以执行，无论它是哪种类型，也不管它是何时编写的。这个函数对于数字来说是有效的（执行乘法），两个字符串或者一个字符串和一个数字（执行重复），或者任意其他支持扩展接口的兼容对象 —— 甚至是我们尚未编写过的基于类的对象。

除此之外，如果传递的对象不支持这种预期的接口，Python将会在*表达式运行时检测到错误，并自动抛出一个异常。因此编写代码错误进行检查是没有意义的。实际上，这样做会限制函数的功能，因为这会让函数限制在测试过的那些类型上才有效。

这也是Python和静态类型语言如C++和Java至关重要的根本的不同：在Python中，代码不应该关心特定的数据类型。如果不是这样，那么代码将只对编写时你所关心的那些类型有效，对以后的那些可能会被编写的兼容对象类型并不支持，这样做会打乱代码的灵活性。大体上来说，我们在Python中为对象编写接口，而不是数据类型。

当然，这种多态的编程模型意味着必须测试代码去检测错误，而不是开始提供编译器用来为我们检测类型错误的类型声明。那么，以最初做些测试作为代价，我们马上减少了我们必须去编写的代码，让代码可以灵活使用。正如你将学到的，这是一种实实在在的好处。

第二个例子：寻找序列的交集

让我们看一下第二个函数的例子，这个例子要比将参数相乘更有用一些，也能够进一步的解释函数的基本概念。

www.TopSage.com

在第13章中，我们编写了一个loop循环，搜索两个字符串公共元素。我们发现那段代码并不是想象的那么有用，因为这个程序被设置为只能列出定义好的变量并且不能继续使用。当然，我们可以在需要它的每一个地方都使用拷贝粘贴的方法，但是这样的解决方案既不好也不通用 —— 我们还是得编辑每一份拷贝的内容，将它换成不同的序列名称，并且改变不同拷贝所需要的算法。

定义

到现在,你也许已经猜到了解决这种困境的办法：将这个for循环封装在一个函数之中。这样做的好处如下。

- 把代码放在函数中让它能够成为一个想运行多少次就运行多少次的工具。
- 因为调用者可以传递任意类型的参数，函数对于任意两个希望寻找其交集的序列（或者其他可迭代的类型）都是通用的。
- 当逻辑由一个函数进行封装的时候，如果一旦需要修改重复性的任务时候，只需要在函数里进行修改搜索交集的方式就可以了。
- 在模块文件中编写函数意味着它可以被在电脑中的任意程序来导入和重用。

实际效果就是，将代码封装在函数中，使它成为一个通用搜索交集的工具。

```
def intersect(seq1, seq2):
    res = []                    # Start empty
    for x in seq1:              # Scan seq1
        if x in seq2:           # Common item?
            res.append(x)       # Add to end
    return res
```

将第13章的简单代码转化为这样的函数是很直接的。我们就是把原先的逻辑编写在def头部之后，并且让被操作的对象变成被传递进入的参数。为了实现这个函数的功能，我们增加了一个return语句来将最终结果的对象返回给调用者。

调用

在你能够调用函数之前，必须先创建它。你可以先运行def语句，要么就是通过在交互模式下输入，要么就是通过在一个模块文件中编写好它然后导入这个文件。一旦运行了def，就可以通过在括号中传递两个序列对象从而调用这个函数：

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


www.TopSage.com

```
>>> intersect(s1, s2)                    # Strings
['S', 'A', 'M']
```

这里，我们传递了两个字符串，并且得到了一个包含着用逗号分隔的字符的列表。这个函数的算法相当的简单："对于第一个参数中的所有元素，如果也出现在第二个参数之中，将它增加至结果之中"。在Python中表达这样的意思要比用英语表达简单一些，但作用是一样的。

重访多态

和所有的Python中的函数一样，intersect是多态的。也就是说，它可以支持多种类型，只要其支持扩展对象接口：

```
>>> x = intersect([1, 2, 3], (1, 4))     # Mixed types
>>> x                                    # Saved result object
[1]
```

这次，我们给的函数传递了不同类型的对象［一个列表和一个元组（混合类型）］，并且仍然是选择出共有的元素。因为你没有必要去定义预先定义参数的类型，这个intersect函数很容易对传递给它的任何序列对象进行迭代，只要这些序列支持预期的接口就行了。

对于intersect函数，这意味着第一个参数必须支持for循环，并且第二个参数支持成员测试。所有满足这两点的对象都能够正常工作，与它们的类型无关——这包括了物理存储的序列，例如，字符串和列表。所有在第13章见到过的迭代对象，包括文件和字典；甚至我们编写的支持操作符重载技术的任意基于类的对象（之后我们将会在第6部分讨论这一点）（注1）。

这里再一次强调，如果我们传入了不支持这些接口的对象（例如，数字），Python将会自动检测出不匹配，并抛出一个异常——这正是我们所想要的，如果我们希望明确地编

注1：这里有两点细节。其一，从技术上来讲，文件也可作为intersect函数的第一个参数，但不能是第二个，因为文件在第一个in成员关系测试执行后，就会被扫描至文件尾端。例如，像intersect(open('data1.txt'), ['line1\n', 'line2\n', 'line3\n'])这样的调用可行，但像intersect(open('data1.txt'), open('data2.txt'))这样的调用就不行了，除非第一个调用中只含一行；就实际序列而言，由迭代环境所做的iter调用来取得迭代器，每次总是会重头开始，但是，一旦文件已被打开或读取，其迭代器实质上已被耗尽。其二，注意到，对类而言，我们可能使用第24章所谈到的更新__iter__或较旧__getitem__运算符重载方法来支持预期的迭代协议。如果我们这么做，就能定义并控制迭代对数据的意义是什么。

写类型检测的话，我们利用它来自己实现。通过不编写类型测试，并且允许Python检测不匹配，我们都减少了自己动手编写代码的数量，并且增强了代码的灵活性。

本地变量

intersect函数中的res变量在Python中称为本地变量 —— 这个变量只是在def内的函数中是可见的，并且仅在函数运行时是存在的。实际上，由于所有的在函数内部进行赋值的变量名都默认为本地变量，所以intersect函数内的所有的变量均为本地变量。

- res是明显的被赋值过的，所以它是一个本地变量。
- 参数也是通过赋值被传入的，所以seq1和seq2也是本地变量。
- for循环将元素赋值给了一个变量，所以变量x也是本地变量。

所有的本地变量都会在函数调用时出现，并在函数退出时消失 —— intersect函数末尾的return语句返回结果对象，但是变量res却消失了。为了完整地介绍本地变量的概念，我们需要继续学习第16章。

本章小结

这一章节介绍了函数定义的核心概念 —— 语法以及def和return语句的操作，函数调用表达式的行为，以及Python函数中多态的概念和优点。正如我们见到的那样，def语句是实时创建函数对象的可执行代码。当一个函数稍后被调用时，对象通过赋值被传递给函数（请回忆一下在Python中赋值表示对象引用，我们在第6章学习过，内部真实含义就是指针），并且将计算得到的值通过return返回。我们也开始在这一章对本地变量和作用域的概念进行了探索，而我们会将所有这些主题的细节留在第16章进行介绍。那么，下面让我们来做个简单测试吧。

www.TopSage.com

大家网
TopSage.com

本章习题

1. 编写函数有什么意义？
2. 什么时候Python将会创建函数？
3. 当一个函数没有return语句时，它将返回什么？
4. 在函数定义内部的语句什么时候运行？
5. 检查传入函数的对象类型有什么错误？

习题解答

1. 函数是Python避免程序代码冗余的最基本方式：把代码分解成函数，意味着未来只有一个运算的代码的拷贝需要更新。函数是Python中代码重用的基本单位：在函数中包装代码，就使其成为可再利用的工具，可在许多程序中调用它。最后，函数可让我们把复杂系统分割为可管理的部分，而每一部分都可独立进行开发。
2. 当Python运行到并执行def语句时，函数就会被创建。这个语句会创建函数对象，并将其赋值给函数名。当函数所在模块文件被另一个模块导入时，通常就会发生这种事（回想一下，导入会从头到尾运行文件中的代码，包括任何的def），但是，当def通过交互模式输入，或者嵌套在其他语句中时（例如，if），也会发生这件事。
3. 如果控制流程来到函数主体末尾并没有运行return语句，函数就会传回None对象。这类函数通常是通过表达式语句调用，并将其None结果赋值给变量通常是没有意义的。
4. 函数主体（嵌套在函数定义语句中的代码）在函数稍后通过一个调用表达式调用时就会执行。函数每次被调用，主体都会全新运行一次。
5. 检查传入函数的对象类型，实质上就是破坏函数的灵活性、把函数限制在特定的类型上。没有这类检查时，函数可能可以处理所有的对象类型：任何支持函数所预期的接口的对象都能用（接口一词是指函数所执行的一组方法和表达式运算符）。

www.TopSage.com

第16章

作用域和参数

第15章介绍了函数定义和调用。正如我们所知，Python的基本函数模型是易用的。这一章将深入介绍Python作用域（变量定义以及查找的地方）以及参数传递（传递给函数作为其输入对象的方式）背后的细节。

作用域法则

既然现在你已经准备编写函数了，那么我们需要更正式的了解Python中变量名的含义。当你在一个程序中使用变量名时，Python创建、改变或查找变量名都是在所谓的命名空间（一个保存变量名的地方）中进行的。当我们谈论到搜索变量名对应于代码的值的时候，作用域这个术语指的就是命名空间。也就是说，在代码中变量名被赋值的位置决定了这个变量名能被访问到的范围。

关于所有变量名，包括作用域的定义在内，都是在Python赋值的时候生成的。正如我们所知，Python中的变量名在第一次被赋值时已经创建，并且必须经过赋值后才能够使用。由于变量名最初没有声明，Python将一个变量名被赋值的地点关联为（绑定给）一个特定的命名空间。换句话说，在代码中给一个变量赋值的地方决定了这个变量将存在于哪个命名空间，也就是它可见的范围。

除打包代码之外，函数还为程序增加了一个额外的命名空间层:在默认的情况下，一个函数的所有变量名都是与函数的命名空间相关联的。这意味着：

- 一个在def内定义的变量名能够被def内的代码使用。不能在函数的外部引用这样的变量名。
- def之中的变量名与def之外的变量名并不发生冲突，即使是使用在别处的相同的变量名。一个在def之外被赋值（例如，在另外一个def之中或者在模块文件的顶层）的变量X与在这个def之中的赋值的变量X是完全不同的变量。

www.TopSage.com

在任何情况下，一个变量的作用域（它所使用的地方）总是由在代码中被赋值的地方所决定，并且与函数调用完全没有关系。如果一个变量在def内被赋值，它被定位在这个函数之内；如果在def之外被赋值，它就是整个文件全局的。我们将其称为语义作用域，因为变量的作用域完全是由变量在程序文件中源代码的位置而决定的，而不是由函数调用决定。

例如，在下面的模块文件中，X = 99这个赋值语句创建了一个名为X的全局变量（在这个文件中可见），但是X = 88这个赋值语句创建了一个本地变量X（只是在def语句内是可见的）。

```
X = 99

def func():
    X = 88
```

尽管这两个变量名都是X，但是它们作用域可以把它们区别开来。实际上，函数的作用域有助于防止程序之中变量名的冲突，并且有助于函数成为更加独立的程序单元。

函数作用域基础

在开始编写函数之前，我们编写的所有的代码都是位于一个模块的顶层（也就是说，并不是嵌套在def之中），所以我们使用的变量名要么就是存在于模块文件本身，要么就是Python内置预先定义好的（例如，open）（注1）。函数提供了嵌套的命名空间（作用域），使其内部使用的变量名本地化，以便函数内部使用的变量名不会与函数外（在一个模块或是其他的函数中）的变量名产生冲突。再一次说明，函数定义了本地作用域，而模块定义的是全局作用域。这两个作用域有如下的关系。

- **内嵌的模块是全局作用域。**每个模块都是一个全局作用域（也就是说，一个创建于模块文件顶层的变量的命名空间）。对于外部的全局变量就成为一个模块对象的属性，但是在一个模块中能够像简单的变量一样使用。
- **全局作用域的作用范围仅限于单个文件。**别被这里的“全局”所迷惑:这里的全局指的是在一个文件的顶层的变量名仅对于这个文件内部的代码而言是全局的。在Python中是没有基于一个单个的、无所不包的情景文件的全局作用域的。替代这种方法的是，变量名由模块文件隔开，并且必须精确地导入一个模块文件才能够使用这个文件中定义的变量名。

注1：在交互模式提示符下输入的代码，其实就是输入在了__main__内置模块，所以，交互模式建立的变量名也会存在于模块中，因此，也遵循普通的变量名法则。你会在第5部分进一步学习模块。

www.TopSage.com

- **每次对函数的调用都创建了一个新的本地作用域。**每次调用函数，都创建了一个新的本地作用域。也就是说，将会存在由那个函数创建的变量的命名空间。可以认为每一个def语句（以及lambda表达式）都定义了一个新的本地作用域，但是因为Python允许函数在循环中调用自身（一个被称作递归的高级技术），所以从技术上讲本地作用域实际上对应的是函数的调用。换句话说，每一个函数调用都创建了一个新的本地命名空间。递归在处理以前不了解的流程结构时是一个有用工具。
- **赋值的变量名除非声明为全局变量，否则均为本地变量。**在默认情况下，所有函数定义内部的的变量名是位于本地作用域（与函数调用相关的）内的。如果需要给一个在函数内部却位于模块文件顶层的变量名赋值，需要在函数内部通过global语句声明。
- **所有的变量名都可以归纳为本地、全局或者内置的。**在函数定义内部的尚未赋值的变量名是一个在一定范围内（在这个def内部）的本地变量、全局（在一个模块的命名空间内部）或者内置（由Python的预定义__builtin__模块提供的）变量。

在函数内部定义的任意的赋值操作定义的变量名都将成为本地变量：=语句、import语句、def语句、参数传递。此外，注意实地改变对象并不会把变量划分为本地变量，实际上只有对变量名赋值才可以。例如，如果变量名L在模块的顶层被赋值为一个列表，在函数内部的像L.append(X)这样的语句并不会将L划分为本地变量，而L = X却可以。前者L像往常一样会出现在全局变量中，而像后者那样的语句会改变全局变量的列表。

变量名解析：LEGB原则

如果上一节内容看起来有些令人困惑的话，那么让我们总结这样三条简单的原则。对于一个def语句：

- 变量名引用分为三个作用域进行查找：首先是本地，之后是函数内（如果有的话），之后全局，最后是内置。
- 在默认情况下，变量名赋值会创建或者改变本地变量。
- 全局声明将赋值变量名映射到模块文件内部的作用域。

换句话说，所有在函数def语句（或者lambda，我们稍后会学习的一个表达式）内赋值的变量名默认均为本地变量。函数能够在函数内部以及全局作用域 (也就是物理上)直接使用变量名，但是必须声明为全局变量去改变其属性。Python的变量名解析机制有时称为*LEGB*法则，这也是由作用域的命令而来的。

- 当在函数中使用未认证的变量名时，Python搜索4个作用域［本地作用域（*L*），之

www.TopSage.com

后是上一层结构中def或lambda的本地作用域（*E*），之后是全局作用域（*G*），最后是内置作用域（*B*）] 并且在第一处能够找到这个变量名的地方停下来。如果变量名在这次搜索中没有找到，Python会报错。正如我们在第6章学到的那样，变量名在使用前首先必须被赋值过。

- 当在函数中给一个变量名赋值时（而不是在一个表达式中对其进行引用），Python总是创建或改变本地作用域的变量名，除非它已经在那个函数中声明为全局变量。
- 当在函数之外给一个变量名赋值时（也就是，在一个模块文件的顶层，或者是在交互提示模式下），本地作用域与全局作用域（这个模块的命名空间）是相同的。

图16-1描述了Python的四个作用域的关系。注意到第二个E作用域的查找层次（上层def和lambda的作用域）从技术上来说可能不仅是一层查找的层次。当你在函数中嵌套函数时这个层次才需要考虑（注2）。

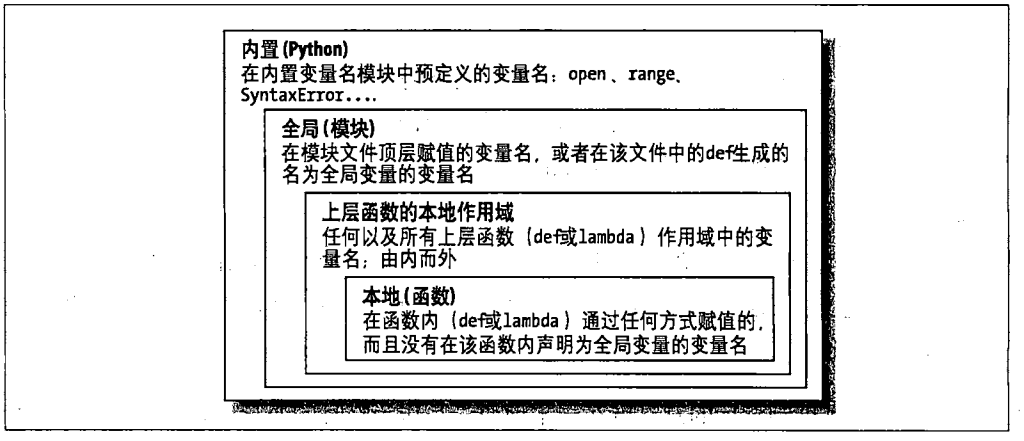


图16-1：LEGB作用域查找原则。当引用一个变量时，Python按以下顺序依次进行查找：从本地变量中，在任意上层函数的作用域，在全局作用域，最后在内置作用域中查找。第一个能够完成查找的就算成功。变量在代码中被赋值的位置通常就决定了它的作用域

此外，记住这些规则仅对简单的变量名有效（例如，spam）。在第5部分和第6部分中，我们将会看到被验证的属性变量名（例如，object.spam）会存在于特定的对象中，并遵循一种完全不同的查找规则，而不止我们这里提到的作用域的概念。属性引用（变量名跟着点号）搜索一个或多个对象，而不是作用域，并且有可能涉及到所谓的“继承”的概念（将在第6部分讨论）。

注2：本书第一版时，作用域搜索规则称为LGB原则。“嵌套def层”是后来Python新增的，从而能够消除需要刻意传递所在作用域变量名的任务；这种话题对Python初学者而言通常不感兴趣，所以，我们会将其放到本章后面再谈。

www.TopSage.com

作用域实例

让我们看一个稍大点的例子来说明作用域的概念。假设我们在一个模块文件中编写了下面这个模块文件。

```
# Global scope
X = 99                  # X and func assigned in module: global

def func(Y):            # Y and Z assigned in function: locals
    # Local scope
    Z = X + Y           # X is a global
    return Z

func(1)                 # func in module: result=100
```

这个模块和函数包含了一些变量名去完成其功能。通过使用Python的作用域法则，我们能够将这些变量名进行如下定义。

全局变量名：X，func

因为X是在模块文件顶层注册的，所以它是全局变量；它能够在函数内部进行引用而不需要特意声明为全局变量。因为同样的原因func也是全局变量；def语句在这个模块文件顶层将一个函数对象赋值给了变量名func。

本地变量名：Y，Z

对于这个函数来说，Y和Z是本地变量（并且只在函数运行时存在），因为他们都是在函数定义内部进行赋值的：Z是通过=语句赋值的，而Y是由于参数总是通过赋值来进行传递的。

这种变量名隔离机制背后的意义就在于本地变量是作为临时的变量名，只有在函数运行时才需要它们。例如，在上一个例子中，参数Y和加法的结果Z只存在于函数内部。这些变量名不会与模块命名空间内的变量名（同理，与其他函数内的变量名）产生冲突。

本地变量/全局变量的区别也使函数变得更容易理解，因为一个函数使用的绝大多数变量名只会在函数自身内部出现，而不是这个模块文件的任意其他地方。此外，因为本地变量名不会改变程序中的其他函数，这会让程序调试起来更加容易。

内置作用域

我们已经简单的介绍了内置作用域，但是可能要比你想象的还要简单。实际上，内置作用域仅仅是一个名为__builtin__的内置模块，但是必须要import __builtin__之后才能使用内置作用域，因为变量名builtin本身并没有预先内置。

www.TopSage.com

内置作用域是通过一个名为__builtin__的标准库模块来实现的，但是这个变量名自身并没有放入内置作用域内，所以必须导入这个文件才能够使用它。一旦这样做，就能够运行dir调用，来看看其中预定义了那些变量名。

TopSage.com

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError',
'DeprecationWarning', 'EOFError', 'Ellipsis',
    ...many more names omitted...
'str', 'super', 'tuple', 'type', 'unichr', 'unicode',
'vars', 'xrange', 'zip']
```

这个列表中的变量名组成了Python中的内置作用域。概括地讲，前一半是内置的异常，而后一半是内置函数。由于LEGB法则Python最后将自动搜索这个模块，将会自动得到这个列表中的所有变量名。也就是说，你能够使用这些变量名而不需要导入任何模块。因此，有两种方法引用一个内置函数:通过LEBG法则带来的好处，或者手动导入__builtin__模块。

```
>>> zip                          # The normal way
<built-in function zip>

>>> import __builtin__           # The hard way
>>> __builtin__.zip
<built-in function zip>
```

其中的第二种实现方法有时在更复杂的任务中是很有用的。细心的读者也许注意到了由于LEGB查找的流程，会使它找到第一处变量名的地方生效。也就是说，在本地作用域的变量名可能会覆盖在全局作用域和内置作用域的有着相同变量名的变量，而全局变量名有可能覆盖内置的变量名。举个例子，一个函数创建了一个名为open的本地变量并将其进行了赋值：

```
def hider():
    open = 'spam'                 # Local variable, hides built-in
    ...
    open('data.txt')              # This won't open a file now in this scope!
```

这样的话，就会将存储于内置（外部）作用域的名为open的内置函数隐藏起来。这也往往是个Bug，并且让人头疼的是，因为Python对于这个问题并不会处理为警告消息

www.TopSage.com

（在高级编程的场合你可能会很想通过在代码中预定义变量名来替代内置的变量名）（注3）。

函数也能够简单地使用本地变量名隐藏同名的全局变量。

```
X = 88                          # Global X

def func():
    X = 99                      # Local X: hides global

func()
print X                         # Prints 88: unchanged
```

这里，函数内部的赋值语句创建了一个本地变量X，它与函数外部模块文件的全局变量X是完全不同的变量。正是由于这一点，如果在def内不增加global声明的话，是没有办法在函数内改变函数外部的变量的。

global语句

global语句是Python中唯一看起来有些像声明语句的语句。但是，它并不是一个类型或大小的声明，它是一个命名空间的声明。它告诉Python函数打算生成一个或多个全局变量名。也就是说，存在于整个模块内部作用域（命名空间）的变量名。我们已经在前边讲过了全局变量名。这里只是作一个总结。

- 全局变量是位于模块文件内部的顶层的变量名。
- 全局变量如果是在函数内被赋值的话，必须经过声明。
- 全局变量名在函数的内部不经过声明也可以被引用。

global语句包含了关键字global，其后跟着一个或多个由逗号分开的变量名。当在函数主体被赋值或引用时，所有列出来的变量名将被映射到整个模块的作用域内。例如：

注3：这是你在Python中可以做的另一件事，但却不应该这么做：因为名称True和False只是内置作用域中的变量，有可以通过True = False这类语句进行重新赋值。这么做时，也不会破坏整体逻辑一致性。这个语句只是替单一作用域中所出现的True这个字重新定义而已。不过，想要更有乐趣的话，你可以写__builtin__.True = False，整个Python行程都把True设成False！未来将不能这么做（而且会让IDLE进入奇怪的恐慌状态，而重设用户程序代码行程）。然而，这种技术对于工具的编写者而言却是有用的，例如，修改open这类内建变量名从而可以定制函数。此外，注意到，像PyChecker这类第三方工具会警告常见的程序错误，包括对内置变量名预期之外所做的赋值运算（这在PyChecker之中称为对内置变量名“遮影”〔shadowing〕）。

www.TopSage.com

```
X = 88                          # Global X

def func():
    global X
    X = 99                      # Global X: outside def

func()
print X                         # Prints 99
```

这个例子中我们增加了一个global声明，以便在def之内的X能够引用在def之外的X，这次它们有相同的值。这里有一个global使用的例子：

```
y, z = 1, 2                     # Global variables in module

def all_global():
    global x                    # Declare globals assigned
    x = y + z                   # No need to declare y, z: LEGB rule
```

这里，x、y和z都是all_global函数内的全局变量。y和z是全局变量，因为它们不是在函数内赋值的；x是全局变量，因为它通过global语句使自己明确地映射到了模块的作用域。如果不使用global语句的话，x将会由于赋值而被认为是本地变量。

注意：y和z并没有进行global声明。Python的LEGB查找法则将会自动从模块中找到它们。此外，注意x在函数运行前可能并不存在。如果这样的话，函数内的赋值语句将自动在模块中创建x这个变量。

最小化全局变量

在默认情况下，函数内部注册的变量名是本地变量，所以如果希望在函数外部对变量进行改变，必须编写额外的代码（global语句）。这是有意而为的。这似乎已成为Python中的一种惯例，如果想做些“错误”的事情，就得多编写代码。尽管有些时候global语句是有用的，然而在def内部赋值的变量名默认为本地变量，通常这都是最好的约定。将其改为全局变量会引发一些软件工程问题：由于变量的值取决于函数调用的顺序，而函数自身是任意顺序进行排列的，导致了程序调试起来变得很困难。

作为例子，思考一下这个模块文件。

```
X = 99

def func1():
    global X
    X = 88

def func2():
    global X
    X = 77
```


www.TopSage.com

现在， 假设你的任务就是修改或重用这个模块文件。这里X的值将会是什么？确切地说，如果不确定引用的时间，这个问题就是毫无意义的。X的值与时间相关联，因为它的值取决于哪个函数是最后进行调用的（有时我们是无法单从这个文件就能说明白的）。

实际的结果就是，为了理解这个代码，你必须去跟踪整个程序的控制流程。此外，如果重用或修改了代码，你必须随时记住整个程序。在这种情况下，如果使用这两个函数中的一个的话，必须要确保没有在使用另一个函数。它们通过全局变量而变得具有相关性（也就是说，是耦合在一起的）。这就是使用全局变量的问题：不像那些依赖于本地变量的由自包含的函数构成的代码，全局变量使得程序更难理解和使用。

另一方面，不使用面向对象的编程方法以及类的话，全局变量也许就是Python中最直接去住状态信息的方法（函数在其下次被调用时需要记住的信息）：本地变量在函数返回时将会消失，而全局变量不是这样。另一种技术，例如，默认可变参数以及嵌套函数作用域，也能够实现这一点，但是它们与将值推向全局作用域来记忆这种方法相比过于复杂了。

一些程序委任一个单个的模块文件去定义所有的全局变量。只要这样考虑，那么就没有什么不利的因素了。此外，在Python中使用多线程进行并行计算程序实际上是要依靠全局变量的：因为全局变量在并行线程中在不同的函数之间成为了共享内存，所以扮演了通信工具的角色（线程超出了本书讲解的范围，更多的细节请参考序言中后边提到的内容）。

那么，到现在为止，特别是编程还不是很熟悉的时候，最好尽可能地避免使用全局变量（试试通过传递函数然后返回值来替代一下）。六个月以后，你和你的合作者都会感谢你没有使用那么多全局变量的。

最小化文件间的修改

这是另一个和作用域相关的问题：尽管我们能够直接修改另一个文件中的变量，但是往往我们都不这样做。思考一下下面两个模块文件。

```
# first.py
X = 99

# second.py
import first
first.X = 88
```

第一个模块文件定义了变量X，这个变量在第二个文件中通过赋值被修改了。注意到我

www.TopSage.com

们必须在第二个文件中导入第一个模块才能够得到它的值：就像我们学到的那样，每个模块都是自包含的命名空间（变量名的封装），而且我们必须导入一个模块才能在从另一个模块中看到它内部的变量。事实上，按照本章的主题来讲，一个模块文件的全局变量一旦被导入就成为了这个模块对象的一个属性：导入者自动得到了这个被导入的模块文件的所有全局变量的访问权，所以在一个文件被导入后，它的全局作用域实际上就构成了一个对象的属性。

在导入第一个模块文件后，第二个模块就将其变量赋了一个新的值。那么，这个赋值的问题就在于，这样的做法过于含糊了： 无论是谁负责维护或重用第一个模块，都不一定知道有个不知道在哪的模块位于导入链上可以修改X。实际上，第二个模块可能在完全不同的一个目录下，而且很难找到。再者，这回让两个文件有过于强的相关性：因为它们都与变量X的值相关，如果没有其中一个文件的话很难理解或重用另一个文件。

这里再说一次，最好的解决办法就是别这样做：在文件间进行通信最好的办法就是通过调用函数，传递参数，然后得到其返回值。在这个特定的情况下，我们最好使用accessor函数去管理这种变化。

```
# first.py
X = 99

def setX(new):
    global X
    X = new

# second.py
import first
first.setX(88)
```

这需要更多的代码，但是这在可读性和可维护性上有着天壤之别：当人们仅阅读第一个模块文件时看到这个函数，他会知道这是一个接入点，并且知道这将可以改变变量X。尽管我们无法改变文件间变量修改的发生，除非已经被广泛接受，我们通常的做法就是最小化文件间变量的修改。

其他访问全局变量的方法

有意思的是，由于全局变量构成了一个被导入的对象的属性，我们能够通过使用导入嵌入的模块并对其属性进行赋值来仿造出一个global语句，就像下边这个模块文件的例子一样。这个文件中的代码先通过变量名然后通过索引sys.modules导入了嵌套的模块，其中包含了已载入的表（关于这个表的更多内容在第21章介绍）：

```
# thismod.py
```


www.TopSage.com

```
var = 99                          # Global variable == module attribute

def local():
    var = 0                       # Change local var

def glob1():
    global var                    # Declare global (normal)
    var += 1                      # Change global var

def glob2():
    var = 0                       # Change local var
    import thismod                # Import myself
    thismod.var += 1              # Change global var

def glob3():
    var = 0                       # Change local var
    import sys                    # Import system table
    glob = sys.modules['thismod'] # Get module object (or use __name__)
    glob.var += 1                 # Change global var

def test():
    print var
    local(); glob1(); glob2(); glob3()
    print var
```

运行时这将会给全局变量加3（只有第一个函数不会影响全局变量）：

```
>>> import thismod
>>> thismod.test()
99
102
>>> thismod.var
102
```

这很有效，并且这表明全局变量与模块的属性是等效的，但是为了清晰的表达你的想法，这种方法要比直接使用global语句需要做更多的工作。

作用域和嵌套函数

到现在为止，忽略了Python的作用法则中的一部分（是有意而为的，因为它在实际情景中很少见到）。但是，现在到了深入学习一下LEGB查找法则中*E*这个字母的时候了。*E*这一层是新内容（是Python 2.2才增加的），它包括了任意嵌套函数内部的本地作用域。嵌套作用域有时也称为静态嵌套作用域。实际上，嵌套是一个语法上嵌套的作用域，它是对应于程序源代码的物理结构上的嵌套结构。

www.TopSage.com

注意：在Python 3.0中，设想有nonlocal语句，它计划用做声明允许对嵌套函数作用和内部的变量有写的权限，这一点很像现在的global语句对整个模块作用域内的变量所做的一样。从语法上来看，这条语句就像global语句一样，仅仅是会使用nonlocal关键字而已。这还是以后的一个设想，所以具体内容需要看3.0的发行报告。

TopSage.com

嵌套作用域的细节

在增加了嵌套的函数作用域后，变量的查找法则变得稍微复杂了一些。对于一个函数：

- 在默认情况下，一个赋值（X = value）创建或改变了变量名X的当前作用域。如果X在函数内部声明为全局变量，它将会创建或改变变量名X为整个模块的作用域。
- 一个引用（X）首先在本地（函数内）作用域查找变量名X；之后会在代码的语法上嵌套了的函数中的本地作用域，从内到外；之后查找当前的全局作用域（模块文件）；最后再内置作用域内（模块__builtin__）。全局声明将会直接从全局（模块文件）作用域进行搜索。

注意：全局声明将会将变量映射至整个模块。当嵌套函数存在时，整个函数的变量也许仅仅是引用，而不会改变。为了讲明白这几点，让我们看看真实的代码。

嵌套作用域举例

下面是一个嵌套作用域的例子。

```
def f1():
    x = 88
    def f2():
        print x
    f2()

f1()                                    # Prints 88
```

首先，这是一个合法的Python代码。def是一个简单的可执行语句，可以出现在任意其他语句能够出现的地方，包括嵌套在另一个def之中。这里，嵌套的def在函数f1调用时运行；这个def生成了一个函数，并将其赋值给变量名f2，f2是f1的本地作用域内的一个本地变量。在此情况下，f2是一个临时函数，仅在f1内部执行的过程中存在。

但是，值得注意的是f2内部发生了什么。当打印变量x时，x引用了存在于函数f1整个本地作用域内的变量x的值。因为函数能够在整个def声明内获取变量名，通过LEGB查找法则，f2内的x自动映射到了f1的x。

www.TopSage.com

这个嵌套作用域查找在嵌套的函数已经返回后也是有效的。例如，下面的代码定义了一个函数创建并返回了另一个函数。

```
def f1():
    x = 88
    def f2():
        print x
    return f2

action = f1()                          # Make, return function
action()                               # Call it now: prints 88
```

在这个代码中，我们命名为f2的函数的调用动作的运行是在f1运行后发生的。f2记住了在f1中嵌套作用域中的x，尽管f1已经不处于激活状态。

工厂函数

根据要求的对象，这种行为有时也叫做闭合（closure）或者工厂函数——一个能够记住嵌套作用域的变量值的函数，尽管那个作用域或许已经不存在了。尽管类（将在第6部分介绍）是最适合用作记忆状态的，因为它们通过属性赋值让这个过程变得很明了，像这样的函数也提供了一种替代的解决方法。

例如，工厂函数有时用于需要及时生成事件处理，实时对不同情况进行反馈的程序中（例如，用户的输入是无法进行预测的）。作为例子，请看下面的这个函数：

```
>>> def maker(N):
...     def action(X):
...         return X ** N
...     [21]return action
...
```

这定义了一个外部的函数，这个函数简单的生成并返回了一个嵌套的函数，却并不调用这个内嵌的函数。如果我们调用外部的函数：

```
>>> f = maker(2)                       # Pass 2 to N
>>> f
<function action at 0x014720B0>
```

我们得到的是生成的内嵌函数的一个引用。这个内嵌函数是通过运行内嵌的def而创建的。如果现在调用从外部得到的那个函数的话：

```
>>> f(3)                               # Pass 3 to X, N remembers 2
9
>>> f(4)                               # 4 ** 2
16
```


www.TopSage.com

它将会调用内嵌的函数。也就是说，maker函数内部的名为action的函数。这一部分最不平常用的就是，内嵌的函数记住了整数2，即maker函数内部的变量N的值，尽管在调用执行f时maker已经返回了值并退出。实际上，在本地作用域内的N被作为执行的状态保留了下来：我们可以使用参数3代替2做立方运算，但是最初的函数f仍然将是像从前一样做平方运算。

现在，如果再调用外层的函数，将得到一个新的有不同状态信息的嵌套函数 —— 得到了参数的三次方而不是平方，但是最初的仍像往常一样是平方。

```
>>> g = maker(3)
>>> g(3)                                # 3 ** 3
27
>>> f(3)                                # 3 ** 2
9
```

这是一个相当高级的技术，除了那些拥有函数编程背景的程序员们（以及优势在lambda中，就像我们谈到过的那样），以后在实际使用中也不会常常见到。通常来说，类（将会在本书稍后进行讨论）是一个更好的像这样进行“记忆”的选择，因为它们让状态变得很明确。不使用类的话，全局变量、像这样的嵌套作用域以及默认的参数就是Python的函数能够保留状态信息的主要方法了。很巧的是，默认参数就是下一部分我们的话题。

使用默认参数来保留嵌套作用域的状态

在较早版本的Python中，上一节中的代码执行会失败，因为嵌套的def与作用域没有一点关系 —— 一个f2中的变量的引用只会搜索f2的本地作用域、全局作用域（f1函数以外）以及内置作用域。因为它将会跳过内嵌函数的作用域，从而会引发错误。为了解决这一问题，程序员一般都会将默认参数值传递给（记住）一个内嵌作用域内的对象：

```
def f1():
    x = 88
    def f2(x=x):
        print x
    f2()

f1()                                 # Prints 88
```

这段代码会在任意版本的Python中工作，而且你也仍会在一些现存的Python代码中看到这样的例子。我们将会在本章后面见到更多默认参数的例子。简而言之，出现在def头部的arg = val的语句表示参数arg在调用时没有值传入进来的时候，默认会使用值val。

通过修改了f2，x=x意味着参数x将会默认使用嵌套作用域中x的值：因为第二个x在Python进入内嵌的def之前是验证过的，所以它仍将引用f1中的x。实际上，默认参数记住了f1中x的值（也就是，对象88）。

上面这些都相当的复杂，而且它完全取决于默认值进行验证的时刻。实际上，嵌套作用域查找法则之所以加入到Python中就是为了让默认参数不再扮演这种角色。如今，Python自动记住了所需要的上层作用域的任意值，为了能够在内嵌的def中使用。

当然，最好的处方就是简单地避免在def中嵌套def，这会让程序更加的简单。下面的代码就是前边例子的等效性形式，这段代码就避免了使用嵌套。注意到，就像这个例子一样，在某一个函数内部就调用一个之后才定义的函数是可行的，只要第二个函数定义的运行是在第一个函数调用前就行，在def内部的代码直到这个函数运行时才会被验证。

```
>>> def f1():
...     x = 88
...     f2(x)
...
>>> def f2(x):
...     print x
...
>>> f1()
88
```

如果使用这样的办法避免嵌套，你几乎都可以忘记在Python中的嵌套作用域的观点了，除非需要编写之前讨论过的工厂函数风格的代码，至少对于def是这样。lambda对于def的嵌套是十分自然的，它常常依赖于嵌套作用域，正像下一节将会介绍的那样。

嵌套作用域和lambda

尽管对于def本身来说、嵌套作用域很少使用，但是当开始编写lambda表达式时，就要注意了。我们到第17章才会深入学习lambda，但是简短地说，它就是一个表达式，将会生成后面会被调用的一个新的函数，与def语句很相似。由于它是一个表达式，尽管能够使用在def中不能使用的地方，例如，在一个列表或是字典常量之中。

就像def，lambda表达式引入了新的本地作用域。多亏了嵌套作用域查找层，lambda能够看到所有生存在所编写的函数的变量。因此，以下的代码现在能够运行，但仅仅是因为如今能够使用嵌套作用域法则了。

```
def func():
    x = 4
    action = (lambda n: x ** n)          # x remembered from enclosing def
    return action
```


www.TopSage.com

```
x = func()
print x(2)                                  # Prints 16, 4 ** 2
```

参考之前对嵌套作用域的介绍，程序员需要使用默认参数从上层作用域传递值给lambda，就像为def做过的那样。例如，下面的代码对于所有版本的Python都可以工作。

```
def func():
    x = 4
    action = (lambda n, x=x: x ** n)     # Pass x in manually
```

由于lambda是表达式，所以它们自然而然的（或者更一般的）嵌套在了def中。因此，它们也就成为了后来在查找原则中增补嵌套函数作用域的最大受益者。在大多数情况下，给lambda函数通过默认参数传递值也就没有什么必要了。

作用域与带有循环变量的默认参数相比较

在已给出的法则中有个值得注意的特例：如果lambda或者def在函数中定义，嵌套在一个循环之中，并且嵌套的函数引用了一个上层作用域的变量，该变量被循环所改变，所有在这个循环中产生的函数将会有相同的值 —— 在最后一次循环中完成时被引用变量的值。

例如，下面的程序试图创建一个函数的列表，其中每个函数都记住嵌套作用域中当前变量i的值。

```
>>> def makeActions():
...     acts = []
...     for i in range(5):                  # Tries to remember each i
...         acts.append(lambda x: i ** x)   # All remember same last i!
...     return acts
...
>>> acts = makeActions()
>>> acts[0]
<function <lambda> at 0x012B16B0>
```

尽管这样，这并不怎么有效：因为嵌套作用域中的变量在嵌套的的函数被调用时才进行查找，所以它们实际上记住的是同样的值（在最后一次循环迭代中循环变量的值）。也就是说，我们将从列表中的每个函数中会得到4的平方的函数，因为i对于在每一个列表中的函数都是相同的值4。

```
>>> acts[0](2)                          # All are 4 ** 2, value of last i
16
>>> acts[2](2)                          # This should be 2 ** 2
16
>>> acts[4](2)                          # This should be 4 ** 2
16
```


www.TopSage.com

这是在嵌套作用域的值和默认参数方面遗留的一种仍需要解释清楚的情况，而不是引用所在的嵌套作用域的值。也就是说，为了让这类代码能够工作，必须使用默认参数把当前的值传递给嵌套作用域的变量。因为默认参数是在嵌套函数创建时评估的（而不是在其稍后调用时），每一个函数记住了自己的变量i的值。

```
>>> def makeActions():
...     acts = []
...     for i in range(5):                      # Use defaults instead
...         acts.append(lambda x, i=i: i ** x)  # Remember current i
...     return acts
...
>>> acts = makeActions()
>>> acts[0](2)                                  # 0 ** 2
0
>>> acts[2](2)                                  # 2 ** 2
4
>>> acts[4](2)                                  # 4 ** 2
16
```

这是一种相当隐晦的情况，但是它会在实际情况中发生，特变是在生成应用于GUI一些部件的回调处理函数的代码中（例如，按钮的事件处理）。我们将会在下一章对默认参数和lambda做更详细的介绍，所以也许稍后会回来重新复习这一部分的内容（注4）。

任意作用域的嵌套

在结束这个话题时，应该提醒大家作用域可以做任意的嵌套，但是只有内嵌的函数（而不是类，将在第6部分介绍）会被搜索：

```
>>> def f1():
...     x = 99
...     def f2():
...         def f3():
...             print x                          # Found in f1's local scope!
...         f3()
...     f2()
...
>>> f1()
99
```

Python将会在所有的内嵌的def中搜索本地作用域，从内至外，在引用过函数的本地作用

注4：在本部分下一章结尾的“函数陷阱”一节中，我们会看到一个和默认值自变量使用列表和字典这类可变对象有关的话题（例如，def f(a=[])：因为默认值是以单一对象来实现的，可变默认值会在调用过程中保留状态，而不是每次调用时都重新设定初始值。这根据你问的人是谁而定，它被视为支持状态保留的功能，或者是这门语言奇怪的瑕疵。下一章会再谈这个话题。

www.TopSage.com

域之后，并在搜索模块的全局作用域之前进行这一过程。尽管如此，这种代码不可能会在实际中这样使用。在Python中，我们说过平坦要优于嵌套：如果你尽可能的少定义嵌套函数的话，你以及同事的生活，都会变得更美好。

TopSage.com

传递参数

本书在前面介绍过参数是通过赋值来传递的。对于初学者来说，这稍有些混乱而不够清晰，这将会在这一部分进行详尽的阐述。下面是对于给函数传递参数时一些简要的关键点。

- **参数的传递是通过自动将对象赋值给本地变量来实现的。**函数参数［调用者（可能的）的共享对象引用值］在实际中只是Python赋值的另一个实例而已。因为应用是以指针的形式实现的，所有的参数实际上都是通过指针进行传递的。作为参数被传递的对象从来不自动拷贝。
- **在函数内部的参数名的赋值不会影响调用者。**在函数运行时，在函数头部的参数名是一个新的、本地的变量名，这个变量名是在函数的本地作用域内的。函数参数名和调用者的变量名是没有别名的。
- **改变函数的可变对象参数的值也许会对调用者有影响。**换句话说，因为参数是简单的通过赋值进行对象的传递的，函数能够改变传入的可变对象，因此其结果会影响调用者。可变参数对于函数来说是可以做输入和输出的。

更多细节，请参看第6章。我们这里所学的对于参数来说也适用，尽管对参数变量名的赋值时是自动并且是隐性的。

Python的通过赋值进行传递的机制与C++的引用参数选项并不是特别相像，但是在实际中，它与C语言的参数传递模型相当相似。

- **不可变参数是“通过值”进行传递。**像整数和字符串这样的对象是通过对象引用而不是拷贝进行传递的，但是因为你无论怎样都不可能在原处改变不可变对象，实际的效果就很像创建了一份拷贝。
- **可变对象是通过“指针”进行传递的。**例如，列表和字典这样的对象也是通过对象引用进行传递的，这一点与C语言使用指针传递数组很相似：可变对象能够在函数内部进行原处的改变，这一点和C数组很像。

当然，如果你从来没有使用过C，Python的参数传递模型看起来也会比较简单：它仅仅是将对象赋值给变量名，并且无论对于可变对象或不可变对象都是这样的。

www.TopSage.com

参数和共享引用

这是一个在实际情况下能够展示以上一些特性的例子。

```
>>> def changer(a, b):              # Function
...     a = 2                       # Changes local name's value only
...     b[0] = 'spam'               # Changes shared object in-place
...
>>> X = 1
>>> L = [1, 2]                      # Caller
>>> changer(X, L)                   # Pass immutable and mutable objects
>>> X, L                            # X is unchanged, L is different
(1, ['spam', 2])
```

在这段代码中，changer函数给参数a赋值，以及参数b所引用的一个对象元素。这两个函数内的赋值从语法上仅有一点不同，但是从结构上看却大相径庭：

- 因为a是在函数作用域内的本地变量名，第一个赋值对函数调用者没有影响：它仅仅是简单地修改了本地变量名a，并没有改变参数a绑定的函数调用者的变量名X。
- b也是一个本地变量名，但是它被传给了一个可变对象（在调用者中称作是L的列表）。因为第二个赋值是一个在原处发生的对象改变，对函数中b[0]进行赋值的结果会在函数返回后影响L的值。实际上，我们没有修改b，我们修改的是b当前所引用的对象的一部分，并且这个改变将会影响调用者。

图16-2 表明了在函数被调用后，函数代码在运行前，变量名/对象所存在的绑定关系。

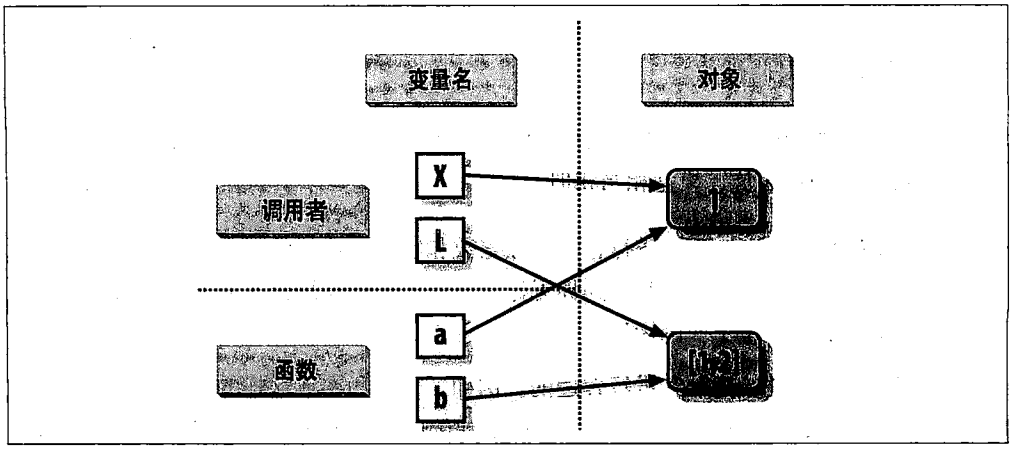


图16-2：引用：参数。因为参数是通过赋值传递的，参数名可以在调用时通过变量实现共享对象。因此，一个函数中对可变对象参数的在原处的修改能够影响调用者。这里，函数中的a和b在函数一开始调用时最初通过变量X和L进行了对象引用。通过变量b对列表的改变在函数调用返回后，L也会发生改变

www.TopSage.com

如果这个例子还是令人困惑的话，这个提醒或许有些帮助。也就是说，自动对传入的参数进行赋值的效果与运行一系列简单的赋值语句是相同的。对于第一个参数，参数赋值对于调用者来说没有什么影响。

```
>>> X = 1
>>> a = X                    # They share the same object
>>> a = 2                    # Resets 'a' only, 'X' is still 1
>>> print X
1
```

但是，对第二个参数的赋值就会影响调用的变量，因为它对对象进行了在原处的修改。

```
>>> L = [1, 2]
>>> b = L                    # They share the same object
>>> b[0] = 'spam'            # In-place change: 'L' sees the change too
>>> print L
['spam', 2]
```

在第6章和第9章中对于共享可变对象的争论可以说明这种现象：对可变对象的在原处的修改会影响其他引用了该对象的变量。这里，实际效果就是使其中的一个参数表现的就像函数的输出。

避免可变参数的修改

在Python中，默认通过引用（也就是指针）进行函数的参数传递，是因为这通常是我们所想要的：这意味着能够在程序中随着需要通过创建多个拷贝传递对象，并且能够按照需要方便的更新这些对象。如果不想要在函数内部在原处的修改影响传递给它的对象，那么，能够简单的创建一个明确的可变对象的拷贝,正如我们在第6章学到的那样。对于函数参数，我们总是能够在调用时对列表进行拷贝。

```
L = [1, 2]
changer(X, L[:])            # Pass a copy, so our 'L' does not change
```

如果不想改变传入的对象，无论函数是如何调用的，我们同样可以在函数内部进行拷贝。

```
def changer(a, b):
    b = b[:]                # Copy input list so we don't impact caller
    a = 2
    b[0] = 'spam'           # Changes our list copy only
```

这两种拷贝的机制都不会阻止函数改变对象：这样做仅仅是防止了这些改变会影响调用者。为了真正意义上防止这些改变，我们总是能够将可变对象转换为不可变对象来杜绝这种问题。例如，元组，在试图改变时会抛出一个异常。

www.TopSage.com

```
L = [1, 2]
changer(X, tuple(L))          # Pass a tuple, so changes are errors
```

这种原理会使用到内置tuple函数，将会以一个序列（任意可迭代对象）中的所用元素为基础创建一个新的元组。这种方法从某种意义上来说有些过于极端：因为这种方法强制函数写成绝不改变传入参数的样子，这种办法强制对函数比原本应该的进行了更多的限制，所以通常意义下应该避免出现。或许将来你会发现对于一些调用来说改变参数是有用的一件事。使用这种技术会让函数失去一种参数能够调用任意列表特定方法的能力，包括了那些不会在原处改变对象的那些方法都不再能够使用。

这里最需要记住的就是，函数能够升级为传入可变对象（例如，列表和字典）的形式。这不会是一个问题，并且有时候这对于有些用途很有用处。但是，你必须要意识到这个属性：如果在你没有预期的情况下对象在外部发生了改变，检查一下是不是一个调用了的函数引起的，并且有必要的话当传入对象时进行拷贝。

对参数输出进行模拟

我们已经讨论了return语句并在例子中使用了它。这里有一个较为纯粹的技巧：因为return能够返回任意种类的对象，所以它也能够返回多个值，如果这些值被封装进一个元组或其他的集合类型。实际上，尽管Python不支持一些其他语言所谓的“通过引用进行调用”的参数传递，我们通常能够通过返回元组并将结果赋值给最初的调用者的参数变量名来进行模拟。

```
>>> def multiple(x, y):
...     x = 2                    # Changes local names only
...     y = [3, 4]
...     return x, y              # Return new values in a tuple
...
>>> X = 1
>>> L = [1, 2]
>>> X, L = multiple(X, L)        # Assign results to caller's names
>>> X, L
(2, [3, 4])
```

看起来这里的代码好像返回了两个值，但是实际上只有一个：一个包含有2个元素的元组，它的圆括号是可选的，这里省略了。在调用返回之后，我们能够使用元组赋值去分解这个返回元组的组成部分。（如果你忘记怎么去做，阅读第4章“元组”以及第11章的“赋值语句”）。这段代码的实际效果就是通过明确的赋值模拟了其他语言中的输出参数。X和L在调用后发生了改变，但是这仅仅是因为代码编写而已。

www.TopSage.com

特定的参数匹配模型

正如我们看到的，参数在Python中总是通过赋值进行传递的。传入的对象赋值给了在def头部的变量名。尽管这样，在模型的上层，Python提供了额外的工具，该工具改变了调用过程中，赋值时参数对象匹配在头部的参数名的优先级。这些工具都是可选的，但是允许编写使用更复杂的调用模式的函数。

在默认情况下，参数是通过其位置进行匹配的，从左至右，而且必须精确地传递和函数头部参数名一样多的参数。还能够通过定义变量名进行匹配，默认参数值，以及对于额外参数的容器。

这一部分的有些内容变得比较复杂，并且在学习语法的细节之前，这些特定的模型是可选的，并且必须要根据变量名匹配对象，匹配完成后在传递机制的底层依然是赋值。实际上，这些工具对于编写库文件的人来说，要对比应用程序开发者更有用。但是因为尽管你不会自己动手编写这些模型你还是有可能在这里犯错误，这里是一些关于匹配模型的大纲。

位置：从左至右进行匹配

一般情况下，也是我们迄今为止使用的那种方法，是通过位置进行参数的匹配。

关键字参数：通过参数名进行匹配

调用者可以定义哪一个函数接受这个值，通过在调用时使用参数的变量名，使用name=value这种语法。

默认参数：为没有传入值的参数定义参数值

如果调用时传入的值过于少的话，函数能够为参数定义接受的默认值，再一次使用语法name=value。

可变参数：收集任意多基于位置或关键字的参数

函数能够使用特定的参数，它们是以字符*开头，收集任意多的额外参数（这个特性常常称作是可变参数，类似C语言中的可变参数特性，也能够支持可变长度参数的列表）。

可变参数：传递任意多的基于位置或关键字的参数

调用者能够再使用*语法去将参数集合打散，分成参数。这个“*”与在函数头部的“*”恰恰相反：在函数头部它意味着收集任意多的参数，而在调用者中意味着传递任意多的参数。

表16-1总结了与特定匹配模式有关的语法。

www.TopSage.com

表16-1：函数参数匹配表

语法	位置	解释
func(value)	调用者	常规参数：通过位置进行匹配
func(name=value)	调用者	关键字参数：通过变量名匹配
func(*name)	调用者	以name传递所有的对象，并作为独立的基于位置的参数
func(**name)	调用者	以name成对的传递所有的关键字/值，并作为独立的关键字参数
def func(name)	函数	常规参数：通过位置或变量名进行匹配
def func(name=value)	函数	默认参数值，如果没有在调用中传递的话
def func(*name)	函数	匹配并收集（在元组中）所有包含位置的参数
def func(**name)	函数	匹配并收集（在字典中）所有包含位置的参数

在调用中（在表中的前4行），简单的通过变量名位置进行匹配，但是使用name=value的形式告诉Python依照变量名进行匹配，这些叫做关键字参数。在调用中使用“*”或者“**”允许我们在一个序列或字典中相应地封装任意多的位置相关或者关键字的对象。

在函数的头部，一个简单的变量名是通过位置或变量名进行匹配的（取决于调用者是如何传递给它参数的），但是name=value的形式定义了默认的参数值。*name的形式收集了任意的额外不匹配的参数到元组中，并且**name的形式将会收集额外的关键字参数到字典之中。

在这其中，关键字参数和默认参数也许是在Python代码中最常见的了。关键字允许使用其变量名去标记参数，让调用变得更有意义。我们之前见过默认参数，作为一种从内嵌函数作用域传递值的办法，但是它们实际上比这更通用：它们允许创建任意可选的参数，并在函数定义中提供了默认值。

特定匹配模式可以自由地确认有多少参数是必须传递给函数的。如果函数定义了默认参数，如果你传递太少的参数它们就会被使用。如果一个函数使用*可变参数列表的形式，你能够传入任意多的参数；*变量名会将额外的参数收集到一个数据结构中去。

关键字参数和默认参数的实例

使用代码来解释要比前文的描述所暗含的意思更简单。像其他大多数语言一样，Python默认会通过位置匹配变量名。例如，如果定义了一个需要三个参数的函数，必须使用三个参数对它进行调用。

www.TopSage.com

```
>>> def f(a, b, c): print a, b, c
...
```

这里，我们依照位置传递值：a匹配到1，b匹配到2，依次类推。

大家网 TopSage.com

```
>>> f(1, 2, 3)
1 2 3
```

关键字参数

在Python中，调用函数的时候，能够更详尽的定义内容传递的位置。关键字参数允许通过变量名进行匹配，而不是通过位置。

```
>>> f(c=3, b=2, a=1)
1 2 3
```

例如，这个调用中c=3，意味着将3传递给参数c。更准确地讲，Python将调用中的变量名c匹配给在函数定义头部的名为c的参数，并将值3传递给了那个参数。实际的效果就是这个调用与上一个调用的效果一样，但是注意到，当关键字参数使用时参数从左至右的关系不再重要了，因为参数是通过变量名进行传递的，而不是根据其位置。甚至在一个调用中混合使用基于位置的参数和基于关键字的参数都可以。在这种情况下，所有基于位置的参数首先依照从左至右的顺序匹配头部的参数，之后再进行基于变量名进行关键字的匹配。

```
>>> f(1, c=3, b=2)
1 2 3
```

当人们第一次看到这种形式的时候，他们都想知道为什么使用这样的工具。关键字在Python中扮演了两个典型的角色。首先，他们使调用显得更文档化一些（假设使用了比a、b和c更好的参数名）。例如，下面这种形式的调用：

```
func(name='Bob', age=40, job='dev')
```

这种形式的调用要比直接进行一个由逗号分隔的三个值的调用明了得多：关键字参数在调用中起到了数据标签的作用。第二个主要的角色就是与使用的默认参数进行配对，我们将在下一部介绍。

默认参数

我们讨论嵌套作用域时，涉及了一些默认参数的内容。简而言之，默认参数允许创建函数可选的参数。如果没有传入值的话，在函数运行前，参数就被赋了默认值。例如，这里有个函数需要一个参数，和两个默认参数。

www.TopSage.com

```
>>> def f(a, b=2, c=3): print a, b, c
...
```

当调用这个函数的时候，我们必须为a提供值，无论是通过位置参数或者关键字参数来实现。然而，为b和c提供值是可选的。如果我们不给b和c传递值，它们会默认分别赋值为2和3：

```
>>> f(1)
1 2 3
>>> f(a=1)
1 2 3
```

当给函数传递两个值的时候，之后c得到默认值，并且当有三个值传递时，不会使用默认值：

```
>>> f(1, 4)
1 4 3
>>> f(1, 4, 5)
1 4 5
```

最后，这里关键字和默认参数一起使用后的情况。因为它们都破坏了通常的的从左至右的位置映射，关键字参数从本质上允许我们跳过有默认值的参数：

```
>>> f(1, c=6)
1 2 6
```

这里，a通过位置得到了1，c通过关键字得到了6，而b，在两者之间，通过默认值获得2。

小心不要被在一个函数头部和一个函数调用中的特定的name=value语法搞糊涂。在调用中，这意味着通过变量名进行匹配的关键字，而在函数头部，它为一个可选的参数定义了默认值。无论是哪种情况，这都不是一个赋值语句。它是在这两种情况下的特定语法，改变了默认的参数匹配机制。

任意参数的实例

最后两种匹配扩展，*和**，是让函数支持接受任意数目的参数的。它们都可以出现在函数定义或是函数调用中，并且它们在两种场合下有着相关的目的。

收集参数

第一种用法：在函数定义中，在元组中收集不匹配的位置参数。

```
>>> def f(*args): print args
...
```


www.TopSage.com

当这个函数调用时，Python将所有位置相关的参数收集到一个新的元组中，并将这个元组赋值给变量args 。因为它是一个一般的元组对象，能够索引或在一个for循环中进行步进。

```
>>> f()
()
>>> f(1)
(1,)
>>> f(1,2,3,4)
(1, 2, 3, 4)
```

特性与类似，但是它只对关键字参数有效。将这些关键字参数传递给一个新的字典，这个字典之后将能够通过一般的字典工具进行处理。在这种情况下，允许将关键字参数转换为字典，你能够在之后使用键调用进行步进或字典迭代，如下段程序所示。

```
>>> def f(**args): print args
...
>>> f()
{ }
>>> f(a=1, b=2)
{'a': 1, 'b': 2}
```

最后，函数头部能够混合一般参数、*参数以及**去实现更加灵活的调用方式。

```
>>> def f(a, *pargs, **kargs): print a, pargs, kargs
...
>>> f(1, 2, 3, x=1, y=2)
1 (2, 3) {'y': 2, 'x': 1}
```

实际上，这种特性能够混合成更复杂的形式，以至于刚开始看上去有些糊涂，我们将会在本章稍后对这个概念进行复习。尽管如此，让我们先看一下在函数调用时而不是定义时使用*和**发生了什么吧。

分解参数

在最新的Python版本中，我们在调用函数时能够使用*语法。在这种情况下，它与函数定义的意思相反。它会分解参数的集合，而不是创建参数的集合。例如，我们能够通过一个元组给一个函数传递四个参数，并且让Python将它们分解成不同的参数。

```
>>> def func(a, b, c, d): print a, b, c, d
...
>>> args = (1, 2)
>>> args += (3, 4)
>>> func(*args)
1 2 3 4
```

相似地，在函数调用时，**会以键/值对的形式分解一个字典，使其成为独立的关键字参数。

www.TopSage.com

```
>>> args = {'a': 1, 'b': 2, 'c': 3}
>>> args['d'] = 4
>>> func(**args)
1 2 3 4
```

大家网 TopSage.com

另外，我们在调用中能够以非常灵活的方式混合普通的参数、基于位置的参数、以及关键字参数。

```
>>> func(*(1, 2), **{'d': 4, 'c': 4})
1 2 4 4

>>> func(1, *(2, 3), **{'d': 4})
1 2 3 4

>>> func(1, c=3, *(2,), **{'d': 4})
1 2 3 4
```

这种代码在编写脚本时，不能预测将要传入函数的参数的数目的时候是很方便的。作为替代方法，能够在运行时创建一个参数的集合，并且可以统一使用这种方法进行函数的调用。另外，别混淆函数头部或是函数调用时*/**的语法：在头部，它意味着收集任意数目的参数，而在调用时，它解包任意数目的参数。

当我们在下一章接触内置函数apply的时候，会复习这种形式（一种特定调用语法,可以当作替代和包含的工具）。

关键字参数和默认参数的混合

下面是一个介绍关键字和默认参数在实际应用中稍复杂的例子。在这个的例子中，调用者必须至少传递两个参数（去匹配spam和eggs），其他的是可选的。如果忽略它们，Python将会分配头部定义的默认值给toast和ham。

```
def func(spam, eggs, toast=0, ham=0):    # First 2 required
    print (spam, eggs, toast, ham)

func(1, 2)                               # Output: (1, 2, 0, 0)
func(1, ham=1, eggs=0)                   # Output: (1, 0, 0, 1)
func(spam=1, eggs=0)                     # Output: (1, 0, 0, 0)
func(toast=1, eggs=2, spam=3)            # Output: (3, 2, 1, 0)
func(1, 2, 3, 4)                         # Output: (1, 2, 3, 4)
```

再次强调当关键字参数在调用过程中使用时，参数排列的位置并没有关系，Python通过变量名进行匹配，而不是位置。调用者必须提供spam和eggs的值，而它们可以通过位置或变量名进行匹配。另外，注意：name=value的形式在调用时和def中有两种不同的含义（在调用时代表关键字参数而在函数头部代表默认参数）。

www.TopSage.com

min调用

为了更加详细的介绍这一部分内容，让我们通过一个练习来说明实际应用中的一个参数匹配工具。假设你想要编写一个函数，这个函数能够计算任意参数集合和任意对象数据类型集合中的最小值。也就是说，这个函数应该接受零个或多个参数：希望传递多少就可以传递多少。此外，这个函数应该能够使用所有的Python对象类型：数字、字符串、列表、字典的列表、文件甚至None。

第一个要求提供了一个能够充分展示*的特性的自然样本：我们能够将参数收集到一个元组中，并且可以通过简单的loop依次步进处理每一个参数。第二部分的问题定义很简单：因为每个对象类型支持对比，没有必要对每种类型都创建一个函数（一个多态的应用）。我们能够不论其类型进行简单地比较，并且让Python执行正确的比较。

满分

下面文件介绍了编写这个操作的三种方法，从某种程序上讲：其中至少一个是学生在学习的过程中提出来的

- 第一个函数获取了第一个参数（args是一个元组），并且使用分片去掉第一个得到了剩余的参数（一个对象同自己比较是没有意义的，特别是这个对象是一个较大的结构时）。
- 第二个版本让Python自动获取第一个参数以及其余的参数，因此避免了进行一次索引和分片。
- 第三个版本通过对内置函数list的调用让一个元组转换为一个列表，之后调用list内置的sort方法来实现比较。

Sort方法是用C语言进行编写的，所以有时它要比其他的程序运行地快，而头两种办法的线性搜索将会让它们在绝大多数时间都要更快（注5）。文件*mins.py*包含了所有三种解决办法的代码。

注5：其实，这相当复杂。Python sort例程是以C写成，使用高度优化的算法，试着利用被排序元素间的部份次序。这种排序称为“timsort”，以其创造者Tim Peter命名，而在其文档中，声称有些时候有“超自然的性能”（对排序来讲，这真的很好）。不过，排序本质上依然是指数型的（必须多次切开序列，再重组起来），而其他版本只是执行线性、由左至右的扫描。结果就是，如果参数有部份定序，排序就会快一点，否则可能会慢一点。即使这样，Python的性能还是会不时的发生变化，而排序以C来实现的确有很大的帮助；有关精确的分析，你应该以下一章会碰到的time和timeit模块进行计时。

www.TopSage.com

```
def min1(*args):
    res = args[0]
    for arg in args[1:]:
        if arg < res:
            res = arg
    return res

def min2(first, *rest):
    for arg in rest:
        if arg < first:
            first = arg
    return first

def min3(*args):
    tmp = list(args)            # Or, in Python 2.4+: return sorted(args)[0]
    tmp.sort()
    return tmp[0]

print min1(3,4,1,2)
print min2("bb", "aa")
print min3([2,2], [1,1], [3,3])
```

所有的这三种解决办法在文件运行时都产生了相同的结果。试着在交互模式下输入一些调用来测试这些方法。

```
% python mins.py
1
aa
[1, 1]
```

注意：上边这三种方法都没有做没有参数传入时的测试。它们可以做这样的测试，但是在这里做是没有意义的。所有的这三种解决办法，如果没有参数传入的话，Python都会自动的抛出一个异常。当尝试获取元素0时，第一种方案会发生异常；当Python检测到参数列表不匹配时，第二种方案会发生异常；在尝试最后返回元素0时，第三种方案会发生异常。

这就是我们所希望得到的结果：因为函数支持任何数据类型，其中没有有效的信号值能够传回标记一个错误。有异常来做这种规则（例如，在运行到错误发生时，不得不有很复杂的运行开销），通常来说，最好假设参数在函数代码中有效，并且当它们不是这样时可以让Python来抛出一个错误。

加分点

如果学生和读者能够使用这些函数来计算最大值，而不是最小值的话，那么他们能够在这里得到加分。这算是简单的：头两个函数只需要改为<to>，而第三个只需要返回

www.TopSage.com

tmp[-1]而不是tmp[0]。对于加分点，请确认函数名也修改成了max（尽管这从严格意义上讲是可选的）。

通用化单个的函数计算无论最小值还是最大值都可以，也是可能的，这样的函数需要使用到评估对比表达式。例如，内置函数eval（参考库手册）或者传入一个任意的比较函数。文件*minmax.py*显示了后者的原理是如何实现的。

```
def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res

def lessthan(x, y): return x < y                    # See also: lambda
def grtrthan(x, y): return x > y

print minmax(lessthan, 4, 2, 1, 5, 6, 3)            # Self-test code
print minmax(grtrthan, 4, 2, 1, 5, 6, 3)

% python minmax.py
1
6
```

和这里一样，函数作为另一种参数对象可以传入一个函数。例如，为了创建max（或者其他）函数，我们能够简单地传入正确种类的比较函数。这看起来像是附加的工作，但是这种通用化函数（而不是剪切和粘帖来改变一个字符）核心的一点就是在未来我们只需要修改一个版本就可以了，而不是两个。

结论

当然，所有这些不过是编写代码练习而已。没有理由编写min或max函数，因为这两个都是Python内置的函数！内置版本的函数工作起来基本上很像我们自己编写的函数，不过它们是用C编写的，目的是为了优化运行速度。

一个更有用的例子：通用set函数

现在，让我们看一个实际中常用的使用特定参数匹配模式的例子吧。在前一章的末尾，我们编写了一个函数返回了两个序列的公共部分（它将挑选出在两个序列中都出现的元素）。这里是一个能够对任意数目的序列（一个或多个）进行公共部分挑选的函数，通过使用可变参数的匹配形式*args去收集传入的参数。因为参数是作为一个元组传入的，我们能够通过一个简单的for循环对它们进行处理。我们编写一个union函数，来从任意多的参数中收集所有曾经在任意操作对象中出现过的元素。

```
def intersect(*args):
    res = []
    for x in args[0]:                    # Scan first sequence
        for other in args[1:]:           # For all other args
            if x not in other: break     # Item in each one?
        else:                            # No: break out of loop
            res.append(x)                # Yes: add items to end
    return res

def union(*args):
    res = []
    for seq in args:                     # For all args
        for x in seq:                    # For all nodes
            if not x in res:
                res.append(x)            # Add new items to result
    return res
```

因为这些工具是值得重用的（并且在交互模式下它们重新输入有些太大了），我们将会把这些函数保存为一个名为*inter2.py*的模块文件（在第5部分有更多关于模块的内容）。无论是哪个函数，参数在调用时都是作为元组`args`传入的。就像原始的`intersect`函数一样，这些函数也都对任意类型的序列有效。在这里，他们处理了字符串、混合类型以及两个以上的序列。

```
% python
>>> from inter2 import intersect, union
>>> s1, s2, s3 = "SPAM", "SCAM", "SLAM"

>>> intersect(s1, s2), union(s1, s2)          # Two operands
(['S', 'A', 'M'], ['S', 'P', 'A', 'M', 'C'])

>>> intersect([1,2,3], (1,4))                 # Mixed types
[1]

>>> intersect(s1, s2, s3)                     # Three operands
['S', 'A', 'M']

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

注意： 因为Python有一个新的`set`对象类型（在第5章介绍过），这本书中所有关于集合处理的例子都没有严格存在的必要。之所以介绍它们，不过是用来说明如何编写函数（因为Python在不断的改进，这本书中的例子有时会显得过时）。

参数匹配：细节

如果决定使用并混合特定的参数匹配模型，Python将会遵循下面有关顺序的法则。

- 在函数调用中，所有的非关键字参数（`name`）必须首先出现，其后跟随所有的关

www.TopSage.com

键字参数（name=value），后边跟着*name的形式，并且如果需要的话，最后是**name的形式。

- 在函数头部，参数必须以相同的顺序出现：一般参数（name），紧跟着默认参数（name=value），后面如果出现了的话是*name的形式，如果使用了的话最后是**name。

如果你使用任何其他的顺序混合了参数，你将会得到一个语法错误，因为其他顺序的混合会产生歧义。Python内部是使用以下的步骤来在赋值前进行参数匹配的。

1. 通过位置分配非关键字参数。
2. 通过匹配变量名分配关键字参数。
3. 其他额外的非关键字参数分配到*name元组中。
4. 其他额外ide关键字参数分配到**name字典中。
5. 用默认值分配给在头部未得到分配的参数。

在这之后，Python检测来确保每个参数只传入了一个值。如果不是这样的话，将会发生错误。这实际上和它看起来一样的复杂，但是弄清楚Python的匹配算法有助于理解一些复杂费解的情况，特别是当使用了混合模式进行参数匹配的时候。我们知道第4部分结尾的练习中会看到附加的关于这些特定匹配模式的例子。

正如你看到的，高级参数匹配模型可以变得很复杂。它们也可以完全是可选的。能够仅使用简单的基于位置的匹配，并且也许在初学的时候这是一个不错的选择。尽管如此，因为有些Python的工具使用了这些模式，一些使用常规还是很重要的。

为什么要在意：关键字参数

关键字参数在Tkinter中扮演很重要的角色，Tkinter是Python中的标准GUI APi。我们稍后会见到Tkinter，这里仅作为预习，关键字参数设置了GUI组件创建时的配置选项。例如，界面的调用。

```
from Tkinter import *
widget = Button(text="Press me", command=someFunction)
```

创建了一个新的按钮并定义了它的文字以及回调函数，使用了text和command关键字参数。因为对于一个部件的设置选项的数目可能很多，关键字参数能够从中进行选择。如果不是这样的话，也许必须根据位置列举出所有可能的选项，要么期待一个明智的基于位置的参数的默认协议来处理每一个可能选项的设置。

www.TopSage.com

本章小结

这一章，我们学习了关于函数的两个关键概念：作用域（当使用时变量如何查找）和参数（对象是如何传入函数的）。正如我们所学的那样，变量作为它所赋值的函数定义的本地变量，除非他们特定地声明为全局变量。和我们所看到的一样，参数是通过赋值传入函数的，这也就意味着是通过对象引用，其真实含义就是通过指针。

对于作用域和参数，我们也都学过了一些更高级的扩展。例如，嵌套函数作用域、默认参数以及关键字参数。最后，我们学习了一些通用的设计观点（避免使用全局变量和跨文件间的改变），并看到了可变类型的参数能够像其他共享引用一样，展示出共同的特性。除非在传入时，对象是一份明确的拷贝，在函数内对传入的可变对象的改变会影响调用者。

下一章将结束函数的学习，该章将会探索关于函数更高级的概念：lambda、generator、iterator以及如map这样的函数工具。这些很多基于函数的概念是Python中一种普通的对象的延伸，并且因此能够支持一些高级和相当灵活的处理模型。

www.TopSage.com

本章习题

1. 下面的代码会输出什么？为什么？

```
>>> X = 'Spam'
>>> def func():
...     print X
...
>>> func()
```

2. 下面的代码会输出什么？为什么？

```
>>> X = 'Spam'
>>> def func( ):
...     X = 'NI!'
...
>>> func( )
>>> print X
```

3. 下面的代码会打印什么内容？为什么？

```
>>> X = 'Spam'
>>> def func( ):
...     X = 'NI'
...     print X
...
>>> func( )
>>> print X
```

4. 下面的代码会产生什么输出？为什么？

```
>>> X = 'Spam'
>>> def func( ):
...     global X
...     X = 'NI'
...
>>> func( )
>>> print X
```

5. 下面的代码会输出什么呢？为什么？

```
>>> X = 'Spam'
>>> def func( ):
...     X = 'NI'
...     def nested( ):
...         print X
...     nested( )
...
>>> func( )
>>> X
```

6. 这段代码会输出什么？为什么？

www.TopSage.com

```
>>> def func(a, b, c=3, d=4): print a, b, c, d
...
>>> func(1, *(5,6))
```

7. 举出三种或四种Python函数中保存状态信息的方法。

8. 举出三种函数和调用者能够交流结果的方法。

习题解答

1. 这里的输出是'Spam'，因为函数引用的是所在模块中的全局变量（因为不是在函数中赋值的，所以被当作是全局变量）。

2. 这里的输出也是'Spam'，因为在函数中赋值变量会将其变成本地遍历，从而隐藏了同名的全局变量。print语句会找到没有发生改变的全局（模块）作用域中的变量。

3. 这会在一行上打印'NI'，在另一行打印'Spam'，因为函数中引用的变量会找到其本地变量，而print中引用的变量会找到其全局变量。

4. 这次只打印了'NI'，因为全局声明会强制函数中赋值的变量引用其所在的全局作用域中的变量。

5. 这个例子的输出还是'NI'一行，而'Spam'在另一行，因为嵌套函数中的print语句会在所在的函数本地作用域中发现变量名，而末尾的print会在全局作用域中发现这个变量。

6. 这里的输出为"1 5 6 4"：1是通过位置进行匹配的，5和6是以*name位置匹配b和c（6覆盖了c的默认值），而d默认为4，因为没有传入它的值。

7. 虽然函数返回后本地变量的值就会消失，但Python函数可以利用全局变量、嵌套函数中引用所在函数作用域的变量或默认参数值，通过它们来保存状态信息。另一种方式是使用OOP和类，它所支持的状态保存比前三种技术都要好，因为它是利用属性赋值运算明确进行状态保存的。

8. 函数可以用return语句、修改传入的可变参数以及通过设置全局变量来返回其结果。全局变量一般都很少应用（除了很特殊的情况，例如,多线程编程），因为这会让代码难以理解和使用。return语句通常是最好的选择，但是，在有准备的情况下，修改可变对象也是可以的。函数也可以和系统组件进行通信，例如文件和套接字，但这些已经不在本书讨论的范围之内了。

www.TopSage.com

TopSage.com

第17章 函数的高级话题

这一章将会介绍一系列更高级的与函数相关的话题：lambda表达式、如map和列表解析这样的函数式编程工具以及生成器函数和表达式等。使用函数的魅丽部分原因在于函数的接口，所以我们在这里也探索了一些通用的函数设计原则。因为这是第4部分的最后一章，我们将会以一系列常见的陷阱和练习题来结束这一部分，来帮助你开始使用已经学习过的那些概念，来编写程序。

匿名函数：lambda

前文介绍过，在Python中，基本的函数该如何编写。下一部分介绍一些更高级的与函数相关的概念。其中大多数都是可选的特性，但是在使用得当的时候，它们能够简化编程任务。

除了def语句之外，Python还提供了一种生成函数对象的表达式形式。由于它与LISP语言中的一个工具很相似，所以称为lambda。就像def一样，这个表达式创建了一个之后能够调用的函数，但是它返回了一个函数而不是将这个函数赋值给一个变量名。这也就是lambda有时被称作匿名（也就是没有函数名）的函数的原因（注1）。实际上，它们常常以一种行内进行函数定义的形式被使用，或者用作推迟执行一些代码。

lambda表达式

lambda的一般形式是关键字lambda，之后是一个或多个参数（与在一个def头部在括号内的参数列表极其相似），紧跟的是一个冒号，之后是一个表达式：

```
lambda argument1, argument2,... argumentN : expression using arguments
```

注1：“lambda”这个名称似乎常常会让人害怕，但没那么严重。这个名称来自于LISP，而LISP则是从lambda calculus（一种符号逻辑形式）取得这个名称的。不过，在Python中，这其实只是一个关键词，作为引入表达式的语法而已。

www.TopSage.com

有lambda表达式所返回的函数对象与由def创建并赋值后的函数对象工作起来是完全一样的，但是lambda一些不同之处让其在扮演特定的角色时很有用。

- **lambda是一个表达式，而不是一个语句。**因为这一点，lambda能够出现在Python语法不允许def出现的地方 —— 例如，在一个列表常量中或者函数调用中。此外，作为一个表达式，lambda返回了一个值（一个新的函数），可以选择性地赋值给一个变量名。相反，def语句总是得在头部将一个新的函数赋值给一个变量名，而不是将这个函数作为结果返回。
- **lambda的主体是一个单个的表达式，而不是一个代码块。**这个lambda的主体简单得就好像放在def主体的return语句中的代码一样。简单地将结果写成一个顺畅的表达式，而不是明确的返回。因为它仅限于表达式，lambda通常要比def功能要小：你仅能够在lambda主体中封装有限的逻辑进去，连if这样的语句都不能够使用。这是有意设计的 —— 它限制了程序的嵌套：lambda是一个为编写简单的函数而设计的，而def用来处理更大的任务。

除了这些差别，def和lambda都能够做同样种类的工作。例如，我们见到了如何使用def语句创建函数。

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

但是，能够使用lambda表达式达到相同的效果，通过明确地将结果赋值给一个变量名，之后就能够通过这个变量名调用这个函数。

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

这里的f被赋值给一个lambda表达式创建的函数对象。这也就是def所完成的任务，只不过def的赋值是自动进行的。

默认参数也能够在lambda参数中使用，就像在def中使用一样。

```
>>> x = (lambda a="fee", b="fie", c="foe": a + b + c)
>>> x("wee")
'weefiefoe'
```

在lambda主体中的代码想在def内的代码一样都遵循相同的作用域查找法则。Lambda表达式引入的一个本地作用域更像一个嵌套的def语句，将会自动从上层函数中、模块中以及内置作用域中（通过LEGB法则）查找变量名。

www.TopSage.com

```
>>> def knights():
...     title = 'Sir'
...     action = (lambda x: title + ' ' + x)   # Title in enclosing def
...     return action                          # Return a function
...
>>> act = knights()
>>> act('robin')
'Sir robin'
```

在Python 2.2中，变量名title的值通常会修改为通过默认参数的值传入。如果忘记其中的原因，请复习第16章中的相关内容。

为什么使用lambda

通常来说，lambda起到了一种函数速写的作用，允许在使用的代码内嵌入一个函数的定义。它们完全是可选的（你总是能够使用def来替代它们），但是在你仅需要嵌入小段可执行代码的情况下它们会带来一个更简洁的代码结构。

例如，我们在稍后会看到回调处理，它常常在一个注册调用（registration call）的参数列表中编写成单行的lambda表达式，而不是使用在文件其他地方的一个def来定义，之后引用那个变量名（请看本章稍后的边栏内容中的例子）。

lambda通常用来编写跳转表（jump table），也就是行为的列表或字典，能够按照需要执行相应的动作。如下段代码所示。

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

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

print L[0](3)                           # Prints 9
```

当需要封装把小段的可执行代码编写进def语句从语法上不能编写进的地方时，lambda表达式作为def的一种速写来说是最为有用的。 例如，这种代码片段，可以通过在列表常量中嵌入lambda表达式创建一个含有三个函数的列表。一个def是不会在列表常量中工作的，因为它是一个语句，而不是一个表达式。

能够使相同的办法用Python语言，在字典或其他的数据结构中创建一个行为表。

```
>>> key = 'got'
>>> {'already': (lambda: 2 + 2),
...  'got':     (lambda: 2 * 4),
...  'one':     (lambda: 2 ** 6)
... }[key]()
8
```


www.TopSage.com

这里，当Python创建这个字典的时候，每个嵌套的lambda都生成并留下了一个在之后能够调用的函数。通过键索引来取回其中一个函数，而括号使取出的函数被调用。当这样编写的时候，与在第12章中向你展示的if语句的扩展用法相比来说，一个字典也就变成一个更加通用的多路分支工具。

如果不是用lambda做这种工作，需要使用三个文件中其他地方出现过的def语句来替代，也就是在这些函数将会使用的那个字典外的某处需要定义这些函数。

```
def f1(): return 2 + 2
def f2(): return 2 * 4
def f3(): return 2 ** 6
...

key = 'one'
{'already': f1, 'got': f2, 'one': f3}[key]()
```

同样，会实现相同的功能，但是def也许会出现在文件中的任意位置，即使它们只有很少的的代码。类似刚才lambda的代码，提供了一种特别有用的可以在单个情况出现的函数：如果这里的三个函数不会在其他的地方使用到，那么将它们的定义作为lambda嵌入在字典中就是很合理的了。不仅如此，def格式要求为这些小函数创建量名，这些变量名也许会与这个文件中的其他变量名发生冲突。

lambda在函数参数里作为行内临时函数的定义，并且该函数在程序中不再其他地方使用时也是很方便的。在本章稍后学习map时，会介绍一些例子。

如何（不要）让Python代码变得晦涩难懂

由于lambda的主体必须是单个表达式（而不是一些语句），由此可见仅能将有限的逻辑封装到一个lambda中。如果你知道在做什么，那么你就能在Python中作为基于表达式等效的写法编写足够多的语句。

例如，如果你希望在lambda函数中进行print，简单的编写sys.stdout.write(str(x)+'\n')这个表达式，而不是使用print x这样的语句（回忆第11章，其实这就是print实际上所做的）。简单的在一个lambda中映射逻辑，能够使用在第13章中曾介绍过的if/else三元表达式。

```
if a:
    b
else:
    c
```

能够由以下的概括等效的表达式来模拟：

www.TopSage.com

```
b if a else c

((a and b) or c)
```

因为这样类似的表达式能够放在lambda中，所以它们能够在lambda函数中来实现选择逻辑。

```
>>> lower = (lambda x, y: x if x < y else y)
>>> lower('bb', 'aa')
'aa'
>>> lower('aa', 'bb')
'aa'
```

此外，如果需要在lambda函数中执行循环，能够嵌入map调用或列表解析表达式（我们之前在第13章见过的工具，将会在这章稍后进行复习）这样的工具来实现。

```
>>> import sys
>>> showall = (lambda x: map(sys.stdout.write, x))

>>> t = showall(['spam\n', 'toast\n', 'eggs\n'])
spam
toast
eggs

>>> showall = lambda x: [sys.stdout.write(line) for line in x]

>>> t = showall(('bright\n', 'side\n', 'of\n', 'life\n'))
bright
side
of
life
```

这些技巧必须在万不得已的情况下才使用。一不小心，它们就会导致不可读（也称为晦涩难懂）的Python代码。一般来说，简洁优于复杂，明确优于晦涩，而且一个完整的语句要比神秘的表达式要好。从另一个方面来说，你也会发现适度的使用这些技术是很有用处的。

嵌套lambda和作用域

lambda是嵌套函数作用域查找（我们在第16章见到的LEGB原则中的E）的最大受益者。例如，在下面的例子中，lambda出现在def中（很典型的情况），并且在上层函数调用的时候，嵌套的lambda能够获取到在上层函数作用域中的变量名x的值。

```
>>> def action(x):
...     return (lambda y: x + y)      # Make and return function, remember x
...
>>> act = action(99)
```


www.TopSage.com

```
>>> act
<function <lambda> at 0x00A16A88>
>>> act(2)
101
```

在上一章中关于嵌套函数作用域的讨论没有表明的就是lambda也能够获取任意上层lambda中的变量名。这种情况有些隐晦，但是想象一下，如果我们把上一个例子中的def换成一个lambda。

```
>>> action = (lambda x: (lambda y: x + y))
>>> act = action(99)
>>> act(3)
102
>>> ((lambda x: (lambda y: x + y))(99))(4)
103
```

这里嵌套结构让函数在调用时创建了一个函数。无论以上哪种情况，嵌套的lambda代码都能够获取在上层lambda函数中的变量x。这可以工作，但是这种代码让人相当费解。出于对可读性的要求，通常来说，最好避免使用嵌套的lambda。

作为参数来应用函数

一些程序需要以一种更通用的样子来调用任意的函数，而不需要一开始就知道它们的函数名或参数（我们将在后边举例介绍这有用的原因）。内置函数apply以及Python中能够使用的一些特定的调用语法，可以完成这样的任务。

注意： 在编写这本书的时候，在Python 2.5中，在这一部分提到的无论是apply还是特定的调用语法都可以自由的使用，但是看起来apply将会在Python 3.0 中消失。如果你希望得到面向未来的代码，使用等效的特定调用语法，别使用apply。

内置函数apply

当需要变得更加动态的话，可以通过将一个函数作为一个参数传递给apply来调用一个生成的函数，并且也将传给那个函数的参数作为一个元组传递给apply函数。

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


www.TopSage.com

为什么要在意：回调

lambda的另一个常见的应用就是为Python的Tkinter GUI API定义行内的回调函数。例如，如下的代码创建了一个按钮，这个按钮在按下的时候会打印一行信息。

```
import sys
x = Button(
    text ='Press me',
    command=(lambda:sys.stdout.write('Spam\n')))
```

这里，回调处理是通过传递一个用lambda所生产的函数作为command的关键字参数。与def相比lambda的优点就是处理按钮动作的代码都在这里，嵌入了按钮创建的调用中。

实际上，lambda知道时间发生时才会调用处理器执行。在按钮按下时，编写的调用才发生，而不是在按钮创建时发生。

因为嵌套的函数作用域法则对lambda也有效，它们也使回调处理变得更简单易用，自Python2.2之后，它们自动查找编写时所在的函数中的变量名，并且在绝大多数情况下，都不再需要传入参数默认参数。这对于获取特定的self实例参数是很方便的，这些参数是在上层的类方法函数中的本地变量（关于类的更多内容在第6部分介绍）。

```
class MyGui:
    def makewidgets(self):
        Button(command=(lambda: self.display("spam")))
    def display(self, message):
        ...use message...
```

在上一个发布版本中，self必须要作为默认参数来传入到函数中。

apply函数简单地调用了在第一个参数位置上传入的函数，并将传入的元组作为函数预期的参数。因为参数列表可以作为一个元组（也就是一个数据结构）传入，一个程序能够在运行时构建这个参数列表（注2）。

apply真正的力量在于它并不需要知道一个函数调用时需要多少参数。例如，你可以使用逻辑去选择一些函数和参数列表，并且使用apply以任意组合来调用。

注2：不要把apply和下一节的主题map搞混了。pply是执行单个函数的调用，把参数传入该函数，这样只进行一次。map会替序列中每个元素都调用函数，这样进行多次。

www.TopSage.com

```
if <test>:
    action, args = func1, (1,)
else:
    action, args = func2, (1, 2, 3)
...
apply(action, args)
```

一般来说，在你无法预测参数列表的任何时候，apply都是有用的。例如，如果用户通过用户接口选择了任意的函数，当编写代码的时候可能不能实在地编写一个函数调用。为了能够解决类似的问题，简单地使用元组操作来构建一个参数的列表，之后间接的通过apply来调用这个函数。

```
>>> args = (2,3) + (4,)
>>> args
(2, 3, 4)
>>> apply(func, args)
9
```

传入关键字参数

apply调用还支持了可选的第三参数，其中能够传入一个字典、来代表传给这个函数的关键字参数。

```
>>> def echo(*args, **kwargs): print args, kwargs
...
>>> echo(1, 2, a=3, b=4)
(1, 2) {'a': 3, 'b': 4}
```

在运行时，允许创建位置和关键字参数。

```
>>> pargs = (1, 2)
>>> kargs = {'a':3, 'b':4}
>>> apply(echo, pargs, kargs)
(1, 2) {'a': 3, 'b': 4}
```

和apply类似的调用语法

Python还允许在调用中使用特定的语法，从而完成与apply调用相同功效的工作。这种语法就是镜像了我们在第16章曾见到过的在def头部任意参数。例如，假设下一个例子中的变量名仍像前面一样进行了赋值。

```
>>> apply(func, args)                  # Traditional: tuple
9
>>> func(*args)                        # New apply-like syntax
9
>>> echo(*pargs, **kargs)              # Keyword dictionaries too
(1, 2) {'a': 3, 'b': 4}
```


www.TopSage.com

这种特定的调用语法比apply函数更新，并且如今它更趋向于应用。除了它与def头部是对称的并可以少敲几下键盘以外，它相对于一个明确的apply调用来说，并没有什么明显的优点。尽管如此，这种新的调用语法的一种形式允许我们传递真正的附加参数，并且这样也是更通用的。


```
>>> echo(0, *pargs, **kargs)          # Normal, *tuple, **dictionary
(0, 1, 2) {'a': 3, 'b': 4}
```

在序列中映射函数：map

程序对列表和其他序列常常要做的一件事就是对每一个元素进行一个操作并把其结果集合起来。例如，在一个列表counter中更新所有的数字，可以简单地通过一个for循环来实现：

```
>>> counters = [1, 2, 3, 4]
>>>
>>> updated = []
>>> for x in counters:
...     updated.append(x + 10)          # Add 10 to each item
...
>>> updated
[11, 12, 13, 14]
```

因为这是一个如此常见的操作，Python实际上提供了一个内置的工具，为你做了大部分的工作。map函数会对一个序列对象中的每一个元素应用被传入的函数，并且返回一个包含了所有函数调用结果的一个列表。如下所示。

```
>>> def inc(x): return x + 10          # Function to be run
...
>>> map(inc, counters)                 # Collect results
[11, 12, 13, 14]
```

map在第13章中是作为一个并行loop遍历处理工具进行介绍的。也许你回忆起来了，你为函数参数传入了None这个值，为的是对元素进行配对。这里，我们将会传入一个真正的函数来对它进行充分的利用，从而可以对在列表中的每一个元素应用这个函数：map对每个列表中的元素动调用了inc函数，并将所有的返回值收集到一个列表中。

```
>>> map((lambda x: x + 3), counters)   # Function expression
[4, 5, 6, 7]
```

这里，函数将会为counters列表中的每一个元素加3。因为这个函数不会在其他的地方用到，所以将它写成了一行的lambda。因为这样使用map与for循环是等效的，在多编写一些的代码后，你就能够得到自己编写一个一般的映射工具了。

www.TopSage.com

大家网
TopSage.com

```
>>> def mymap(func, seq):
...     res = []
...     for x in seq: res.append(func(x))
...     return res
...
>>> map(inc, [1, 2, 3])
[11, 12, 13]
>>> mymap(inc, [1, 2, 3])
[11, 12, 13]
```

尽管如此，因为map是内置函数，它总是可用的，并总是以同样的方式工作，还有一些性能方面的优势（简而言之，它要比自己编写的for循环更快）。此外，map还有比这里介绍的更高级的使用方法。例如，提供了多个序列作为参数，它能够并行返回分别以每个序列中的元素作为函数对应参数得到的结果的列表。

```
>>> pow(3, 4)
81
>>> map(pow, [1, 2, 3], [2, 3, 4])         # 1**2, 2**3, 3**4
[1, 8, 81]
```

这里，pow函数在每次调用中都使用了两个参数：每个传入map的序列中都取一个。尽管我们大概也能够来模拟这样做，但是当有速度优势的内置函数已经提供了这样的功能，再去模拟，意义不是很大。

注意： map调用与在第13章中学到过的列表解析很相似，但是map对每一个元素都应用了函数调用而不是任意的表达式。因为这点限制，从某种意义上来说，它成为了不太通用的工具。尽管如此，在某些情况下，目前map比列表解析运行起来更快（也就是说，当映射一个内置函数时），并且它所编写的代码也较少。这是因为这一点，它很有可能仍然在Python 3.0中还可用。尽管如此，最新的Python 3.0文档中提议从内置命名空间中（取而代之的是，它们也许会在某一模块中出现）去掉map调用，一并去掉的还有本节稍后会提到的reduce和filter调用。

尽管map也许还会留下来，而reduce和filter也许就会从3.0中去除，从某种程度上来说，是因为它们对于列表解析是多余的（filter可当作是列表解析if从句的一种），而且部分是由于它们的复杂性（reduce是Python中最为复杂的工具之一，而且并不直接）。对于这个话题我无法预测未来，那么，想要确认更多可能的改变的话就看看Python 3.0的发行报告吧。这些工具仍包括在这版书之中，因为它们是当前版本的Python的一部分，而且在未来的某个时候也许会在你所见到的Python代码中出现。

函数式编程工具：filter和reduce

在Python内置函数中，map函数用来进行函数式编程的这类工具中最简单的函数代表：函数式编程的意思就是对序列应用一些函数的工具。例如，基于某一测试函数过滤出一些

www.TopSage.com

元素（filter），以及对每对元素都应用函数并运行到最后结果（reduce）。例如，下面这个filter的调用实现了从一个序列中挑选出大于5的元素。

```
>>> range(-5, 5)
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]

>>> filter((lambda x: x > 0), range(-5, 5))
[1, 2, 3, 4]
```

序列中的元素若其返回值为真的话，将会被键入到结果的列表中。就像map，这个函数也能够概括地用一个for循环来等效，但是它也是内置的，运行比较快。

```
>>> res = []
>>> for x in range(-5, 5):
...     if x > 0:
...         res.append(x)
...
>>> res
[1, 2, 3, 4]
```

reduce要更复杂一些。这里是两个reduce调用，计算了在一个列表中所有元素加起来的和以及乘起来的乘积。

```
>>> reduce((lambda x, y: x + y), [1, 2, 3, 4])
10
>>> reduce((lambda x, y: x * y), [1, 2, 3, 4])
24
```

每一步，reduce传递了当前的和或乘积以及下一个列表中的元素，传给列出的lambda函数。默认，系列中的第一个元素初始化了起始值。这里是一个对第一个调用的for循环的等效，在循环中使用了额外的代码。

```
>>> L = [1,2,3,4]
>>> res = L[0]
>>> for x in L[1:]:
...     res = res + x
...
>>> res
10
```

自己编写reduce（例如，如果它确实在Python 3.0 中被移除），实际上相当直接。

```
>>> def myreduce(function, sequence):
...     tally = sequence[0]
...     for next in sequence[1:]:
...         tally = function(tally, next)
...     return tally
...
>>> myreduce((lambda x, y: x + y), [1, 2, 3, 4, 5])
```


www.TopSage.com

```
15
>>> myreduce((lambda x, y: x * y), [1, 2, 3, 4, 5])
120
```

如果这引起了你的兴趣，再看看内置的operator模块，其中提供了内置表达式对应的函数，并且对于函数式工具来说，它使用起来是很方便的。

```
>>> import operator
>>> reduce(operator.add, [2, 4, 6])      # Function-based +
12
>>> reduce((lambda x, y: x + y), [2, 4, 6])
12
```

与map一样，filter和reduce支持了强大的函数式编程的技术。一些观察家也将lambda、apply以及列表解析扩展进了Python中函数式工具集中，我们将在下一部分讨论列表解析的内容。

重访列表解析：映射

因为对序列以及集合的映射操作是Python编程中一个常见的任务，Python 2.0引入了一个新的特性（列表解析表达式），它的使用要比我们刚刚学习的工具简单得多。我们在第13章见到了列表解析，但是因为它们与map和filter这样的函数式编程工具相关，所以我们将会在这里复习补充这个话题的内容作为最后的一个回顾。从技术上讲，这个特性与函数并没有绑定在一起：正如我们所见到的，列表解析可以成为一个比map和filter更通用的工具，有时候通过基于函数的另类视角进行分析，有助于深入理解它。

列表解析基础

让我们举一个例子来说明基础知识吧。正如我们在第7章所见到过的，Python的内置ord函数会返回一个单个字符的ASCII整数编码（chr内置函数是它的逆过程，它将一个ASCII整数编码转为字符）。

```
>>> ord('s')
115
```

现在，假设我们希望收集整个字符串中的所有字符的ASCII编码。也许最直接的方法就是使用一个简单的for循环，并将结果附加在列表中。

```
>>> res = []
>>> for x in 'spam':
...     res.append(ord(x))
...
```


www.TopSage.com

```
>>> res
[115, 112, 97, 109]
```

然而，现在我们知道了map，我们能够使用一个单个的函数调用，而不必关心代码中列表的结构，从而实现起来更简单。

```
>>> res = map(ord, 'spam')                    # Apply function to sequence
>>> res
[115, 112, 97, 109]
```

尽管如此，从Python 2.0开始，我们能够通过列表解析表达式得到相同的结果。

```
>>> res = [ord(x) for x in 'spam']            # Apply expression to sequence
>>> res
[115, 112, 97, 109]
```

列表解析在一个序列的值上应用一个任意表达式，将其结果收集到一个新的列表中并返回。从语法上来说，列表解析是由方括号封装起来的（为了提醒它们构造了一个列表）。它们的简单形式是在方括号中编写一个表达式，其中的变量，在后边跟随着的看起来就像一个for循环的头部一样的语句，有着相同的变量名的变量。Python之后将这个表达式的应用循环中每次迭代的结果收集起来。

上一个例子的效果与手动进行for循环和map调用相比，没有什么不同。然而，列表解析可以变得更方便，当我们希望对一个序列应用一个任意表达式的时候。

```
>>> [x ** 2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

这里，我们收集了从0到9数字的平方（我们只是让它在交互模式下打印了结果，如果你需要保留它的话，请将其赋值给一个变量）。和map调用差不多，我们也许能够创建一个小函数来实现平方操作。因为在其他的地方不需要这个函数，通常（但不是必须）在行内编写，使用lambda，而不是使用其他地方的def语句：

```
>>> map((lambda x: x ** 2), range(10))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

这同样也有效，并且它要比等效的列表解析编写更少的代码。它只是稍有一点复杂（至少，一旦理解了lambda后）。对于更高级种类的表达式，那么，通常列表解析将会被认为是输入较少的。下一部分将会告诉你为什么会这样。

增加测试和嵌套循环

列表解析甚至要比现在所介绍的更通用。例如，能够在for之后编写一个if分支，用

www.TopSage.com

来增加选择逻辑。使用了if分支的列表解析能够当成一种与上一部分讨论过的内置的filter类似的工具：它们会在分支不是真的情况下跳过一些序列的元素。这里举一个选择出从0到4的偶数的例子。就像我们刚刚看到过的map可以替代列表解析，为了测试表达式,这里的filter版本创建了一个小的lambda函数。为了对比，在这里也显示了等效的for循环。

```
>>> [x for x in range(5) if x % 2 == 0]
[0, 2, 4]

>>> filter((lambda x: x % 2 == 0), range(5))
[0, 2, 4]

>>> res = [ ]
>>> for x in range(5):
...     if x % 2 == 0:
...         res.append(x)
...
>>> res
[0, 2, 4]
```

所有的这些都是用了求余（求除法的余数）操作符%，用来检测该数是否是偶数。如果一个数字除以2以后没有余数，它就一定是偶数。filter调用与这里的列表解析相比也更短。尽管如此，在列表解析中能够混合一个if分支以及任意的表达式，从而赋予了它通过一个单个表达式，完成了一个filter和一个map相同的功效。

```
>>> [x ** 2 for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]
```

这次，我们收集了从0到9的偶数的平方。若在右边的if中得到的是假的话，for循环就会跳过这些数字，并且用左边的表达式来计算值。这个等效的map调用将需要更多的工作来完成这一部分。我们需要在map迭代中混合filter选择过程，这使得表达式明显复杂得多。

```
>>> map((lambda x: x**2), filter((lambda x: x % 2 == 0), range(10)))
[0, 4, 16, 36, 64]
```

实际上，列表解析还能够更加通用。你可以在一个列表解析中编写任意数量的嵌套的for循环，并且每一个都有可选的关联的if测试。通用的列表解析的结构如下所示。

```
[ expression for target1 in sequence1 [if condition]
              for target2 in sequence2 [if condition] ...
              for targetN in sequenceN [if condition] ]
```

当for分句嵌套在列表解析中时，它们工作起来就像等效的嵌套的for循环语句。例如，如下代码。

www.TopSage.com

```
>>> res = [x + y for x in [0, 1, 2] for y in [100, 200, 300]]
>>> res
[100, 200, 300, 101, 201, 301, 102, 202, 302]
```

与下文如此冗长的代码有相同的效果。

```
>>> res = []
>>> for x in [0, 1, 2]:
...     for y in [100, 200, 300]:
...         res.append(x + y)
...
>>> res
[100, 200, 300, 101, 201, 301, 102, 202, 302]
```

尽管列表解析创建了列表，记住它们能够像任意的序列和其他迭代类型一样进行迭代。这里有个小巧简单的代码，能够不使用列表的数字索引遍历字符串，并收集它们合并后的结果。

```
>>> [x + y for x in 'spam' for y in 'SPAM']
['sS', 'sP', 'sA', 'sM', 'pS', 'pP', 'pA', 'pM',
'aS', 'aP', 'aA', 'aM', 'mS', 'mP', 'mA', 'mM']
```

最后，这里有个复杂得多的列表解析工具，表明了在嵌套的`for`从句中附加`if`选择的作用。

```
>>> [(x, y) for x in range(5) if x % 2 == 0 for y in range(5) if y % 2 == 1]
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```

这个表达式排列了从0到4的偶数与从0到4的奇数的组合。其中`if`分句过滤出了每个系列中需要进行迭代的元素。这里是一个等效的用语句编写而成的代码。

```
>>> res = []
>>> for x in range(5):
...     if x % 2 == 0:
...         for y in range(5):
...             if y % 2 == 1:
...                 res.append((x, y))
...
>>> res
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```

注意到如果你对一个复杂的列表解析有什么困惑的话，你总是能够将列表解析的`for`和`if`分句在其中进行嵌套（将后来的分句缩进到右边），从而得到等效的语句。得到的结果要长得多，但是也许更清晰。

而`map`和`filter`的等效形式往往将会更复杂也会有深层的嵌套，这里不进行说明，将这部分代码留给禅师、前LISP程序员以及犯罪神经病作为练习。

www.TopSage.com

列表解析和矩阵

让我们看一个更高级的列表解析应用，来进一步学习。使用Python编写矩阵（也被称为多维数组）的一个基本的方法就是使用嵌套的列表结构。例如，如下代码使用了嵌套了列表的列表定义了两个3×3的矩阵。

```
>>> M = [[1, 2, 3],
...      [4, 5, 6],
...      [7, 8, 9]]

>>> N = [[2, 2, 2],
...      [3, 3, 3],
...      [4, 4, 4]]
```

使用这样的结构，我们总是能够索引行，以及索引行中的列，使用通常的索引操作。

```
>>> M[1]
[4, 5, 6]

>>> M[1][2]
6
```

那么，列表解析也是处理这样结构的强大的工具，因为它将会自动为我们扫描行和列。例如，尽管这种结构通过行存储了矩阵，为了选择第二列，我们能够简单地通过对行进行迭代，之后从所需要的列中提取出元素，或者就像下面一样通过在行内的位置进行迭代：

```
>>> [row[1] for row in M]
[2, 5, 8]

>>> [M[row][1] for row in (0, 1, 2)]
[2, 5, 8]
```

给出了位置的话，我们能够简单地执行像提取出对角线位置的元素的任务。下面的表达式使用range来生成列表的偏移量，并且之后使用相同的行和列来进行索引，取出了M[0][0]，之后是 M[1][1]（我们假设矩阵有相同数目的行和列）。

```
>>> [M[i][i] for i in range(len(M))]
[1, 5, 9]
```

最后，我们使用列表解析来混合多个矩阵。下面的首行代码创建了一个单层的列表，其中包含了矩阵对元素的乘积。

```
>>> [M[row][col] * N[row][col] for row in range(3) for col in range(3)]
[2, 4, 6, 12, 15, 18, 28, 32, 36]
```


www.TopSage.com

```
>>> [[M[row][col] * N[row][col] for col in range(3)] for row in range(3)]
[[2, 4, 6], [12, 15, 18], [28, 32, 36]]
```

最后一个表达式是有效的，因为row迭代是外层的循环：对于每个row，它运行嵌套的列的迭代来创建矩阵每一行的结果。

```
>>> res = []
>>> for row in range(3):
...     tmp = []
...     for col in range(3):
...         tmp.append(M[row][col] * N[row][col])
...     res.append(tmp)
...
>>> res
[[2, 4, 6], [12, 15, 18], [28, 32, 36]]
```

与这些语句相比，列表解析这个版本只需要一行代码，而且可能对于大型矩阵来说，运行相当快，我们对这些可能会糊涂，那么请进行下一部分的学习。

理解列表解析

拥有了这样的通用性，列表解析变得难以理解，特别是在嵌套的时候。因此，建议对于刚开始使用Python的编程者，通常使用简单的for循环，在其他大多数情况下，使用map调用（除非它们会变得过于复杂）。“保持简洁”法则就在这里生效了，就像往常一样：实现代码的精简与代码的可读性相比，就没有那么重要了。

尽管如此，在这种情况下，对当前额外的复杂度来说有可观的性能优势：基于对运行在当前Python下的测试，map调用比等效的for循环要快两倍，而列表解析往往比map调用要稍快一些（注3）。速度上的差距是来自于底层实现上，map和列表解析是在解释器中以C语言的速度来运行的，比Python的for循环代码在PVM中步进运行要快得多。

因为for循环让逻辑变得更清晰，基于简单性我们通常推荐使用。尽管如此，map和列表解析作为一种简单的迭代是容易理解和使用的，而且如果应用对速度特别重视的话。此外，因为map和列表解析都是表达式，从语法上来说，它们能够在for循环语句不能够出现的地方使用。例如，在一个lambda函数的主体中或者是在一个列表或字典常量中。

注3：这种通常意义上的性能差异取决于调用方式，以及Python本身的变动和优化。例如，最近的Python版本使for循环加速。不过，一般来说，列表解析还是比for循环快很多，甚至也比map快（不过，对于内置函数来说map还是赢家）。要自行测试这些方案的速度，可以参考标准库time模块的time.clock和time.time调用，与2.4版新增的timeit模块，或者本章接下来的“对迭代各种方法进行计时”一节。

www.TopSage.com

然而应该尝试让map调用和列表解析保持简单。对于更复杂的任务，用完整的语句来替代。

为什么要在意：列表解析和map

这里介绍一个实际应用中更现实的列表和map的例子（我们在第13章的列表解析中解决过这个问题，在这里复习它并增加了基于map的替代方案）。回顾文件的readlines方法将返回以换行符\n结束的行：

```
>>> open('myfile').readlines()
['aaa\n', 'bbb\n', 'ccc\n']
```

如果不想要换行符，可以使用列表解析和map调用通过一个步骤从所有的行中将它们都去掉。

```
>>> [line.rstrip() for line in open('myfile').readlines()]
['aaa', 'bbb', 'ccc']

>>> [line.rstrip() for line in open('myfile')]
['aaa', 'bbb', 'ccc']

>>> map((lambda line: line.rstrip()), open('myfile'))
['aaa', 'bbb', 'ccc']
```

这里最后两个使用了文件迭代器（这里实际上是指不需要一个方法调用就能够在迭代中获取所有的行）。map调用要比列表解析稍长一些，但是无论哪种方法都没有必要明确地管理结果列表的构造。

列表解析还能作为一种列选择操作来使用。Python的标准SQL数据库API将返回查询结果保存为与下边类似的元组的列表：列表就是表，而元组为行，元组中的元素的就是列的值：

```
listoftuple = [('bob', 35, 'mgr'), ('mel', 40, 'dev')]
```

一个for循环能够手动从选定的列中提取出所有的值，但是map和列表解析能够一步就做到这一点，并且更快。

```
>>> [age for (name, age, job) in listoftuple]
[35, 40]

>>> map((lambda (name, age, job): age), listoftuple)
[35, 40]
```

这两种方法都能够利用元组赋值来分解列表中作为行的元组。

更多关于Python的数据库API请参考其他的书籍以及资源。

www.TopSage.com

重访迭代器：生成器

本书的这一部分我们学到了编写普通的函数，这种函数收到输入的参数并马上送回一个结果。尽管如此，编写函数能够返回一个值，并且稍后还可以从它刚才离开的地方仍然返回值。这样的函数被认作是生成器，因为它们随时间生成一个序列的值。

从大多数方面来看生成器函数就像一般的函数，但是在Python中，它们被自动用作实现迭代协议，因此它只能够在迭代的语境中出现。我们在第13章学习了迭代器。这里，我们将会对它进行复习来看一看它们与生成器之间有什么关系。

不像一般的函数会生成值后退出，生成器函数在生成值后自动挂起并暂停它们的执行和状态。正是因为这一点，无论是在从头计算整个系列的值，或者手动保存和恢复类中的状态时，它们都常常作为一种实用的替代解决方案。生成器在被挂起时自动地保存状态，它的本地变量将保存状态信息，这些信息在函数恢复时将再度有效。

生成器和一般的函数之间代码上最大的不同就是一个生成器`yield`一个值，而不是`return`一个值。`yield`语句将会将函数关起，并向它的调用者返回一个值，但是保存足够的状态信息为了让其能够在函数从它挂起的地方恢复。这能够允许这些函数不断的产生一系列的值，而不是一次计算所有的值，之后将值以类似列表之类的形式来返回。

生成器函数在Python中与迭代器协议的概念联系在一起。简而言之，包含了`yield`语句的函数将会特地编译为生成器。当调用时，它们返回了一个生成器对象，这个生成器对象支持迭代器对象接口。生成器函数也许也有一个`return`语句，这个语句就是用来终止产生值的。

迭代器对象，依次定义了下一个方法，这些方法既要返回在迭代中的下一个元素，又要抛出一个特定的异常（`StopIteration`）来终结这个迭代。迭代器是通过内置的`iter`函数获取的。如果协议支持的话，Python的`for`循环使用了这个迭代接口协议来步进处理一个序列（或者序列生成器）；如果不支持的话，`for`将会重复对序列进行索引运算。

生成器函数实例

生成器和迭代器都是高级语言特性，所以关于它们所有的故事请查看Python的库手册。

www.TopSage.com

为了讲清楚基础知识，请看下面代码，它定义了一个生成器函数，这个函数将会用来不断地生成一系列的数字的平方（注4）。

```
>>> def gensquares(N):
...     for i in range(N):
...         yield i ** 2          # Resume here later
...
```

这个函数在每次循环时都会产生一个值，之后将其返还给它的调用者。当它被暂停后，它的上一个状态被保存了下来，并且在yield语句之后控制器马上被回收。例如，当用在一个for循环中时，在循环中每一次完成函数的yield语句后，控制权都会返还给函数。

```
>>> for i in gensquares(5):          # Resume the function
...     print i, ':',                # Print last yielded value
...
0 : 1 : 4 : 9 : 16 :
>>>
```

为了终止生成值，函数可以使用一个无值的返回语句，或者在函数主体最后简单地让控制器脱离。

如果想要看看在for里面发生了什么，直接调用一个生成器函数：

```
>>> x = gensquares(4)
>>> x
<generator object at 0x0086C378>
```

得到的是一个生成器对象，它支持迭代器协议（也就是说，next方法可以开始这个函数，或者从它上次yield值后的地方恢复，以及在得到一系列的值的最后一个时，产生StopIteration异常）。

```
>>> x.next()
0
>>> x.next()
1
>>> x.next()
4
>>> x.next()
9
>>> x.next()

Traceback (most recent call last):
  File "<pyshell#453>", line 1, in <module>
```

注4：生成器在Python 2.2之后才能用；在2.2中，必须以特殊的import语句才能开启它：from __future__ import generators（参考第18章有关这条语句的形式）。迭代器已经能够在2.2中使用，大部份是因为底层协议不需要过去不兼容的新语句yield。

www.TopSage.com

```
    x.next()
StopIteration
```

for循环与生成器工作起来是一样的：通过重复调用next方法，直到捕获一个异常。如果一个不支持这种协议的对象进行这样迭代，for循环会使用索引协议进行迭代。

注意在这个例子中，我们能够简单地一次就构建一个所获得的值的列表。

```
>>> def buildsquares(n):
...     res = []
...     for i in range(n): res.append(i**2)
...     return res
...
>>> for x in buildsquares(5): print x, ':',
...
0 : 1 : 4 : 9 : 16 :
```

对于这样的例子，我们还能够是使用for循环、map或者列表解析的技术来实现。

```
>>> for x in [n**2 for n in range(5)]:
...     print x, ':',
...
0 : 1 : 4 : 9 : 16 :

>>> for x in map((lambda x:x**2), range(5)):
...     print x, ':',
...
0 : 1 : 4 : 9 : 16 :
```

尽管如此，生成器能够运行函数在第一线做了所有的工作，当结果的列表很大或者在处理每一个结果都需要很多时间的时候，这一点尤其有用。生成器将在loop迭代中处理一系列值的时间分布开来。尽管如此，对于更多高级的应用，它们提供了一个更简单的替代方案来手动将类的对象保存到迭代中的状态（更多关于类的内容在稍后的第6部分介绍）。有了生成器，函数变量进行了自动的保存和恢复。

扩展生成器函数协议：send和next

在Python 2.5中，生成器函数协议中增加了一个send的方法。send方法生成一系列结果的下一个元素，这一点就像next方法一样，但是它也提供了一种调用者与生成器之间进行通信的方法，从而能够影响它的操作。

从技术上来说，yield现在是一个表达式的形式，可以返回传入的元素来发送，而不是一个语句［尽管无论哪种叫法都可以：作为yield X或者A = (yield X)］。值是通过调用本身的send(value)方法传给生成器的。之后恢复生成器的代码，并且yield表达式返回了为了发送而传入的值。如果调用了正常的next()方法，yield返回None。

www.TopSage.com

例如，用sedn方法，编写一个能够被它的调用者终止的生成器。此外，在2.5版中，生成器还支持throw(type)的方法，它将在生成器内部最后一个yield时产生一个异常以及一个close()方法，它会在生成器内部产生一个终止迭代的新的GeneratorExit异常。这些都是我们这里不会深入学习的一些高级特性；请查看Python的标准库来获得更多的细节。

迭代器和内置类型

正如我们在第13章看到的一样，内置的数据类型设计了对应于内置函数iter的迭代器对象。例如，字典迭代器在每次迭代中产生关键字列表元素。

```
>>> D = {'a':1, 'b':2, 'c':3}
>>> x = iter(D)
>>> x.next()
'a'
>>> x.next()
'c'
```

此外，所有的迭代内容（包括for循环、map调用、列表解析以及很多我们在第13章见到过的其他内容）依次设计用来自动调用iter函数，来看看是不是都支持了迭代协议。这就是为什么我们可以在字典的键中进行循环而不用调用它的关键字方法，步进一个文件中的所有行而不需要调用readlines或xreadline。

```
>>> for key in D:
...     print key, D[key]
...
a 1
c 3
b 2
```

正如我们所看到的，在文件迭代器中，Python简单地载入了一个文件的行。

```
>>> for line in open('temp.txt'):
...     print line,
...
Tis but
a flesh wound.
```

通过支持迭代协议的类来实现任意的生成器对象是可能的，并且已经有很多这样的对象在for循环和其他的迭代环境中使用。这样的类定义了一个特别的__iter__方法，它将返回一个迭代对象（更倾向于__getitem__索引方法）。尽管如此，这已经超出了我们这一章所涉及的内容。请参考第6章获得关于类内容，并且在第24章看一个实现迭代器的类的例子。

www.TopSage.com

生成器表达式：迭代器遇到列表解析

在最新版本的Python中，迭代器和列表解析的概念形成了这个语言的一个新的特性，生成器表达式。从语法上来讲，生成器表达式就像一般的列表解析一样，但是它们是括在圆括号中而不是方括号中的。

```
>>> [x ** 2 for x in range(4)]        # List comprehension: build a list
[0, 1, 4, 9]

>>> (x ** 2 for x in range(4))        # Generator expression: make an iterable
<generator object at 0x011DC648>
```

尽管如此，从执行过程上来讲，生成器表达式很不相同：不是在内存中构建结果，而是返回一个生成器对象，这个对象将会支持迭代协议并在任意的迭代语境的操作中，获得最终结果列表中的一部分。

```
>>> G = (x ** 2 for x in range(4))
>>> G.next()
0
>>> G.next()
1
>>> G.next()
4
>>> G.next()
9
>>> G.next()

Traceback (most recent call last):
  File "<pyshell#410>", line 1, in <module>
    G.next()
StopIteration
```

我们一般不会机械地使用next迭代器来操作生成器表达式，因为for循环会自动触发。

```
>>> for num in (x ** 2 for x in range(4)):
...     print '%s, %s' % (num, num / 2.0)
...
0, 0.0
1, 0.5
4, 2.0
9, 4.5
```

实际上，每一个迭代的语境都会这样，包括sum、map和sorted等内置函数，以及在第13章中我们学到过的其他迭代语境，例如any、all和list内置函数等。

注意：如果生成器表达式是在其他的括号之内的话，就像在那些函数调用之中，在这种情况下，生成器自身的括号就不是必须的了。尽管这样，在下面第二个sorted调用中，还是需要额外的括号。

www.TopSage.com

```
>>> sum(x ** 2 for x in range(4))
14

>>> sorted(x ** 2 for x in range(4))
[0, 1, 4, 9]

>>> sorted((x ** 2 for x in range(4)), reverse=True)
[9, 4, 1, 0]

>>> import math
>>> map(math.sqrt, (x ** 2 for x in range(4)))
[0.0, 1.0, 2.0, 3.0]
```

生成器表达式大体上可以认为是对内存空间的优化：它们不需要像方括号的列表解析一样，一次构造出整个结果列表。它们在实际中运行起来可能稍慢一些，所以它们可能只对于非常大的结果集合的运算来说是最优的选择，这为我们学习下一部分内容做了自然的过渡。

对迭代的各种方法进行计时

本书已经介绍了一些迭代的替代方案。让我们简要的看一下这个实例，来学习融和了所有我们所学过的关于迭代和函数的内容。

列表解析要比for循环语句有速度方面的性能优势，而且map会依据调用方法的不同表现出更好或更差的性能。上一节介绍的生成器表达式看起来比列表解析速度更慢一些，但是它们把内存需求降到了最小。

所有这些今天都是真实的，但是随着时间的不同其相对的性能也有所不同（Python还在不断的优化中）。如果你想要自己测试它们的话，试试在自己的电脑上，用现有的Python版本来运行下面脚本。

```
# file timerseqs.py

import time, sys
reps = 1000
size = 10000

def tester(func, *args):
    startTime = time.time()
    for i in range(reps):
        func(*args)
    elapsed = time.time() - startTime
    return elapsed

def forStatement():
    res = []
```


www.TopSage.com

大家网
TopSage.com

```
        for x in range(size):
            res.append(abs(x))

    def listComprehension():
        res = [abs(x) for x in range(size)]

    def mapFunction():
        res = map(abs, range(size))

    def generatorExpression():
        res = list(abs(x) for x in range(size))

    print sys.version
    tests = (forStatement, listComprehension, mapFunction, generatorExpression)
    for testfunc in tests:
        print testfunc.__name__.ljust(20), '=>', tester(testfunc)
```

这个脚本测试了所有构造结果列表可能的方法，并且就像介绍的那样，每一个都依次执行了一千万步。也就是说，这四个测试分别创建了一千次结果列表，而这个列表中含有一万个元素。

值得注意的是，我们是通过内置的`list`调用来运行生成器表达式，使它得到所有的结果。如果我们不这样的话，我们将会得到一个不做任何实际工作的生成器。此外，注意后边步骤中的代码是如何使用一个包含有四个函数对象的元组，并打印出每一个函数对象的`__name__`属性的——这是一个内置属性，它会提供函数的函数名。

当在Winodws XP上安装了Python 2.5的IDLE中运行这段代码的时候，运行的情况如下：列表解析大概比`for`循环语句快两倍，而`map`在映射像`abs`（计算绝对值）这样的内置函时数比列表解析稍快一些。

```
2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)]
forStatement         => 6.10899996758
listComprehension    => 3.51499986649
mapFunction          => 2.73399996758
generatorExpression  => 4.11600017548
```

但是当我们改变这段代码来每个迭代中执行一个真正的操作时会发生什么情况呢？例如，做加法运算时。

```
...
...
def forStatement():
    res = []
    for x in range(size):
        res.append(x + 10)

def listComprehension():
    res = [x + 10 for x in range(size)]
```


www.TopSage.com

```
def mapFunction():
    res = map((lambda x: x + 10), range(size))

def generatorExpression():
    res = list(x + 10 for x in range(size))
...
...
```

map调用需要进行函数调用使得它变得和for循环语句一样慢，尽管for循环语句的版本使用了更多的代码。

```
2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)]
forStatement          => 5.25699996948
listComprehension     => 2.68400001526
mapFunction           => 5.96900010109
generatorExpression   => 3.37400007248
```

因为解释器的优化是相当内在化的，像这里的Python代码的性能分析是很难处理的事情。猜出哪一种方法的性能是最佳的几乎是不可能的。你所能做到的就是使用Python，计算代码的运行时间。在这种情况下，我们所能确定的就是对于目前的Python，在map调用中使用用户定义的函数至少会导致map慢2倍以上，而列表解析对这样的测试运行最快。

正如前文所提到的，尽管如此，在编写Python代码的时候，性能不应该是最优先考虑的。首先要为可读性和简洁性的标准编写，如果有必要的话，之后再进行优化。如果这四种方法中的任意一种对于程序需要处理的数据集合都足够快的话就很好了，程序的清晰将会成为主要的目标。

为了加深理解，可以尝试改变在这段代码前的重复次数，或者看看更新的timeit模块，这个模块将会自动为代码进行计时，或者完善一些平台相关的问题（例如，在一些平台上， time.time由于time.clock）。此外，参考profile标准库模块可以获得一个完整的源代码配置工具。

函数设计概念

当你开始使用函数时，就开始面对如何将组件聚合在一起的选择了。例如，如何将任务分解成为更有针对性的函数（导致了聚合性）、函数将如何通信（耦合性）等。你需要深入考虑诸如聚合性、耦合性以及函数的大小等性质其中一些可以归类于结构分析和设计的范畴。我们在前一章中介绍过了关于函数和模块耦合性的观念，但是这里是一个对Python的初学者的一些通用的指导方针的复习。

www.TopSage.com

- **耦合性：对于输入使用参数并且对于输出使用return语句。**一般来讲，你需要力求让函数独立于它外部的东西。参数和return语句通常就是隔离对外部的依赖关系的最好的办法，从而让代码中只剩少量醒目位置。
- **耦合性：只有在真正必要的情况下使用全局变量。**全局变量（也就是说，在整个模块中的变量名）通常是一种蹩脚的函数间进行通信的办法。它们引发了依赖关系和计时的问题，会导致程序调试和修改的困难。
- **耦合性：不要改变可变类型的参数，除非调用者希望这样做。**函数会改变传入的可变类型对象，但是就像全局变量一样，这会导致很对调用者和被调用者之间的耦合性，这种耦合性会导致一个函数过于特殊和不友好。
- **聚合性：每一个函数都应该有一个单一的、统一的目标。**在设计完美的情况下，每一个函数中都应该做一件事：这件事可以用一个简单说明句来总结。如果这个句子很宽泛（例如，“这个函数实现了整个程序”），或者包含了很多的排比（例如，“这个函数让员工产生并提交了一个批萨订单”），你也许就应该想想是不是要将它分解成多个更简单的函数了。否则，是无法重用在一个函数中把所有步骤都混合在一起的代码。
- **大小：每一个函数应该相对较小。**从前面的目标延伸而来，这就比较自然，但是如果函数在显示器上需要翻几页才能看完，也许就到了应该把它分开的时候了。特别是Python代码是以简单明了而著称，一个过长或者有着深层嵌套的函数往往就成为设计缺陷的征兆。保持简单，保持简短。
- **耦合：避免直接改变在另一个模块文件中的变量。**我们在前一章中介绍过了这个概念，下面我们将会在本书的下一部分学习模块时重新复习它。作为参考，记住在文件间改变变量会导致模块文件间的耦合性，就像全局变量产生了函数间的耦合一样：模块难于理解和重用。在可能的时候使用读取函数，而不是直接进行赋值语句。

图17-1总结了函数与外部世界通信的方法。输入可能来自于左侧的元素，而结果能以右侧的任意一种形式输出。很多函数设计者倾向于只使用参数作为输入，return语句作为输出。

当然，前面设计的法则有很多特例，包括一些与Python的OOP支持相关的内容。就像将在第6部分看到的那样，Python的类依赖于修改传入的可变对象：类的函数会自动设置传入参数self的属性，从而修改每个对象的状态信息（例如，self.name='bob'）。另外，如果没有使用类，全局变量通常是模块中函数保留调用中状态的最佳方式。如果都在预料之中，副作用就没什么危险。

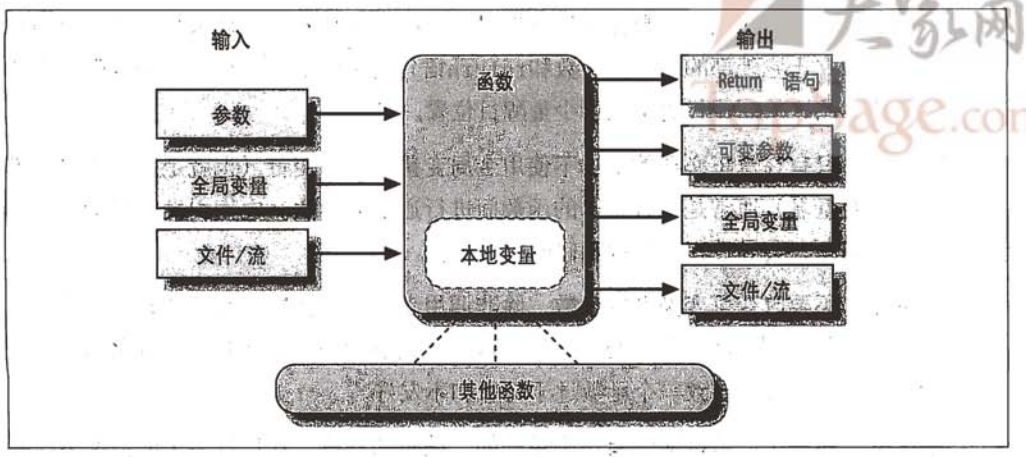


图17-1：函数执行环境。函数可以通过多种办法获得输入产生输出，尽管使用参数作为输入，return语句并配合可变参数的改变作为输出时，函数往往更容易理解和维护

函数是对象：简洁调用

由于函数在运行时是对象，你可以编写通用化程序来处理它们。函数对象能够进行赋值、传递给其他的函数以及在数据结构中排序，这和简单的数字和字符串一样。这些内容在前边例子中已经使用过了。函数对象还支持特殊的操作。

例如，对于def语句中使用的变量名来说，并没有什么特别的。它仅仅是一个分配在当前作用域内的一个变量而已，就像它出现在=符号左边一样。在def运行后，函数名就是一个简单的对象的引用，而且能够将这个对象重新分配给其他的变量名，以及通过任何引用对它进行调用。

```
>>> def echo(message):                    # echo assigned to a function object
...     print message
...
>>> x = echo                              # Now x references it too
>>> x('Hello world!')                     # Call the object by adding ()
Hello world!
```

因为参数是通过赋值传递的，所以给其他函数以参数的形式传入函数也很简单。通过给传入的参数加括号后，被调用者就可以调用传入的函数了。

```
>>> def indirect(func, arg):
...     func(arg)                          # Call the object by adding ()
...
>>> indirect(echo, 'Hello jello!')        # Pass the function to a function
Hello jello!
```


www.TopSage.com

甚至可以将函数对象封装在数据结构中，就像它是整数或字符串一样。因为Python混合类型能够包含任意类型的对象，这里也不是什么特别情况。

```
>>> schedule = [ (echo, 'Spam!'), (echo, 'Ham!') ]
>>> for (func, arg) in schedule:
...     func(arg)
...
Spam!
Ham!
```

这段代码简单地步进到schedule列表里，每次都使用了一个参数调用了echo函数（注意在for循环头部的元组解包赋值，在第13章中介绍过）。Python省略了类型检测创造出了一种有着难以置信的灵活性的编程语言。

函数陷阱

函数有些你想不到的陷阱。它们都很少见，而有些在最新版本中已经从语言中完全消失，但多数都会让新的用户栽跟头。

本地变量是静态检测的

正如我们所知道的一样，Python定义的在一个函数中进行分配的变量名是默认为本地变量的，它们存在于函数的作用域并只在函数运行时存在。Python是静态检测Python的本地变量的，当编译def代码时，不是通过发现赋值语句在运行时进行检测的。这导致了在Python新闻组中有入门者最为常见的陷阱。

一般来说，没有在函数中赋值的变量名会在整个模块文件中查找。

```
>>> X = 99
>>> def selector():          # X used but not assigned
...     print X              # X found in global scope
...
>>> selector()
99
```

这里，函数中的X被解析为模块中的X。但是如果在引用之后增加了一个赋值语句，看看会发生什么。

```
>>> def selector():
...     print X              # Does not yet exist!
...     X = 88               # X classified as a local name (everywhere)
...                          # Can also happen if "import X", "def X" ...
>>> selector()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
```


www.TopSage.com

```
  File "<stdin>", line 2, in selector
UnboundLocalError: local variable 'X' referenced before assignment
```

你得到了一个未定义变量名的错误，但其原因是微妙的。在交互模式下输入或从一个模块文件中导入时，Python读入并编译这级代码。在编译时，Python看到了对X的赋值语句，并且决定了X将会在函数中的任一地方都将是本地变量名。但是，当函数实际运行时，因为在print执行时赋值语句并没有发生，Python告诉你正在使用一个未定义的变量名。根据其变量名规则，本地变量X是在其被赋值前就使用了。实际上，任何在函数体内的赋值将会使其成为一个本地变量名。Import、=、嵌套def、嵌套类等，都会受这种行为的影响。

产生这种问题的原因在于被赋值的变量名在函数内部是当作本地变量来对待的，而不是仅仅在赋值以后的语句中才被当作是本地变量。实际上，前一个例子是最含糊不清的：是希望打印一个全局变量X之后创建一个本地变量X，还是这真的是一个程序错误？因为Python会在函数中将X作为本地变量，它就是一个错误。如果你真的想要打印全局变量X，需要在一个global语句中声明这一点。

```
>>> def selector():
...     global X                # Force X to be global (everywhere)
...     print X
...     X = 88
...
>>> selector()
99
```

记住，尽管这样，这一位置的赋值语句同样会改变全局变量X，而不是一个本地变量。在函数中，不可能同时使用同一个简单变量名的本地变量和全局变量。如果真的是希望打印全局变量，并在之后设置一个有着相同变量名的本地变量，导入上层的模块，并使用模块的属性标记来获得其全局变量。

```
>>> X = 99
>>> def selector():
...     import __main__         # Import enclosing module
...     print __main__.X        # Qualify to get to global version of name
...     X = 88                  # Unqualified X classified as local
...     print X                 # Prints local version of name
...
>>> selector()
99
88
```

点号运算（.X这部分）从命名空间对象中获取了变量的值。交互模式下的命名空间是一

www.TopSage.com

个名为__main__的命名空间，所以__main__.X得到了全局变量版本的X。如果还不够清楚的话，请查看第5部分（注5）。

TopSage.com

默认和可变对象

默认参数是在def语句运行时被评估并保存的，而不是在这个函数调用时。从内部来讲，Python会将每一个默认参数保存成一个对象，附加在这个函数本身。

这也就是通常我们所想要的：因为默认参数是在def时被评估的，如果必要的话，它能够从整个作用域内保存值，但是因为默认参数在调用之间都保存了一个对象，必须对修改可变的默认参数十分小心。例如，下面的函数使用了一个空列表作为默认参数，并在函数每次调用时都对它进行了改变。

```
>>> def saver(x=[]):              # Saves away a list object
...     x.append(1)               # Changes same object each time!
...     print x
...
>>> saver([2])                    # Default not used
[2, 1]
>>> saver()                       # Default used
[1]
>>> saver()                       # Grows on each call!
[1, 1]
>>> saver()
[1, 1, 1]
```

有些人把这种行为当作一种特性。因为可变类型的默认参数在函数调用之间保存了它们的状态，从某种意义上讲它们能够充当C语言中的静态本地函数变量的角色。在一定程度上，它们工作起来就像全局变量，但是它们的变量名对于函数来说是本地变量，而且不会与程序中的其他变量名发生冲突。

尽管这样，对于大多数人来说，这看起来就像是一个陷阱，特别第一次遇到这样的情况的时候。在Python中有更好的办法在调用之间保存状态（例如，使用类，这将在第6部分进行讨论）。

此外，可变类型默认参数记忆起来比较困难（理解起来也不容易）。它们的值取决于默认对象构建的时间。在上一个例子中，其中只有一个列表对象作为默认值，这个列表对象是在def语句执行时被创建的。不会每次函数调用时都得到一个新的列表，所以每次新的元素加入后，列表会变大，对于每次调用，它都没有重置为空列表。

注5：Python已经对此进行了改进，呈现例子中所显示的“UnboundLocalError”错误信息（发生了一般的变量名错误）；不过，这个陷阱依然存在。

如果这不是你想要的行为的话，在函数主体的开始对默认参数进行简单的拷贝，或者将默认参数值的表达式移至函数体内部。只要值是存在于在代码中，而这部分代码在函数每次运行时都会执行的话，你就会每次都得到一个新的对象。

```
>>> def saver(x=None):
...     if x is None:               # No argument passed?
...         x = []                  # Run code to make a new list
...     x.append(1)                 # Changes new list object
...     print x
...
>>> saver([2])
[2, 1]
>>> saver()                         # Doesn't grow here
[1]
>>> saver()
[1]
```

顺便提一下，这个例子中的if语句可以被赋值语句x = x or []来代替，这个赋值语句将会利用Python的or语句返回操作符对象中的一个的特性：如果没有参数传入的话，x将会默认为None，所以or将会返回右边的空列表。

尽管如此，这还不是完全相同的。如果是一个空列表被传入，or表达式将会导致函数扩展并返回一个新创建的列表，而不是像if版本那样扩展并返回传入的列表。（表达式变成[] or []，将会设置为右边空列表的值。参看第12章中的相关内容，如果你想不起来为什么的话）。真正的程序也许都需要这两种行为。

没有return语句的函数

在Python函数中，return（以及yield）语句是可选的。当一个没有精确的返回值的时候，函数在控制权从函数主体脱离时，函数将会推迟。从技术上来讲，所有的函数都返回了一个值，如果没有提供return语句，函数将自动返回None对象：

```
>>> def proc(x):
...     print x                     # No return is a None return
...
>>> x = proc('testing 123...')
testing 123...
>>> print x
None
```

没有return语句的函数与Python对应于一些其他语言中所谓的“过程”是等效的。它们常被当作语句，并且None这个结果被忽略了，就像它们只是执行任务而不需要计算有用的结果一样。

www.TopSage.com

了解这些内容是值得的，因为如果你想要尝试使用一个没有返回值的函数的结果时，Python不会告诉你。例如，将一个列表添加方法的结果赋值不会导致错误，但是得到的会是None，而不是改变后的列表。

```
>>> list = [1, 2, 3]
>>> list = list.append(4)       # append is a "procedure"
>>> print list                  # append changes list in-place
None
```

就像在第14章中提到的“常见编写代码的陷阱”，这样的函数执行任务也会有副作用，就是它们往往设计成语句来运行，而不是表达式。

嵌套作用域的循环变量

我们在第16章中对嵌套函数作用域的讨论中，曾经介绍过这个容易犯错误的地方。但是，作为提醒，在进行嵌套函数作用域查找时，处理嵌套的循环改变了的变量时要小心。所有的引用将会使用在最后的循环迭代中对应的值。作为替代，请使用默认参数来保持循环变量的值（参照第16章中关于这个话题的更多内容）。

本章小结

这一章我们对函数相关的高级话题进行了学习：lambda表达式函数；带有yield语句的生成器函数；生成器表达式；apply调用语法；像map、filter以及reduce函数式编程工具；通用的函数设计观点。在这里我们还复习了迭代器和列表解析，像循环语句一样，它们与函数式编程有关联。作为迭代器观点的总结，我们也测试了所有迭代解决方案的性能。最后，我们复习了与函数相关的一般错误，来帮助你躲开那些潜在的陷阱。

这就是本书关于函数的内容。在下一部分内容中，我们将会学习模块 —— Python中最顶端的组织结构，而且这种结构组织了函数存在的空间。在这之后，我们将会探索类，这个工具是有着特定的第一个参数的函数的封装。就像我们将要看到的一样，在这里所学到的一切将会在本书中函数出现的地方有所应用。

在继续学习之前，通过做本章的测试以及这一部分的练习来确认你已经充分掌握了函数的基础知识。

大家网
TopSage.com

本章习题

1. 列表解析放在方括号和圆括号中有什么区别?
2. 生成器和迭代器有什么关系?
3. 如何分辨函数是否为生成器函数?
4. yield语句是做什么的?
5. 已知函数对象以及内含参数的元组，你可以怎么调用这个函数?
6. map调用和list comprehension有什么关系? 比较并对比两者。
7. lambda表达式和def语句有什么关系? 比较和对比两者。

习题解答

1. 方括号中的列表解析会一次在内存中产生结果列表。当位于圆括号中时，实际上是生成器表达式：它们有类似的意义，但不会一次产生结果列表。与之相对比的是，生成器表达式会返回一个生成器对象，用在迭代环境中时，一次产生结果中的一个元素。
2. 生成器是支持迭代协议的对象：它们有next方法，重复前进到系列结果中的下个元素，以及到系列尾端时引发例外事件。在Python中，我们可以用def、加圆括号的列表解析的生成器表达式以及以类定义特殊方法__iter__来创建生成器对象（本书稍后讨论），通过它们来编写生成器函数。
3. 生成器函数在其代码中的某处会有一个yield语句。除此之外，生成器函数和普通函数相同。
4. 当有了主题语句时，这个语句会让Python把函数特定的编译成生成器；当调用时，会返回生成器对象，支持迭代协议。当yield语句运行时，会把结果返回给调用者，让函数的状态挂起。然后，当调用者再调用next方法时，这个函数就可以重新在上次yield语句后继续运行。生成器也可以有return语句，用来终止生成器。
5. 可以通过类似于apply的调用语法调用函数：function(*argstuple)。也可以使用内置函数apply(function, args)，但是，这个内置函数可能会在未来Python版本中移除，因此，不是一种通用的方法。
6. map调用类似于列表解析，两者都会收集对序列或其他可迭代对象中每个元素应用运算后的结果（一次一个项目），从而创建新列表。其主要差异在于，map会对每

www.TopSage.com

个元素应用函数，而列表解析则是应用任意的表达式。因此，列表解析更通用一些，可以像map那样应用函数调用表达式，但是，map需要一个函数才能应用其他种类的表达式。列表解析也支持扩展语法，例如，嵌套for循环和if分句从而可以包含内置函数filter的功能。

7. lambda和def都会创建函数对象，以便稍后调用。不过，因为lambda是表达式，可以嵌入函数定义中def语法上无法出现的地方。lambda的使用，总是可以用def来替代，并且通过变量名来引用函数。不过，lambda用于嵌入推迟执行的小段程序，并且不太可能在程序其他地方用到时，就相当方便。从语法上来讲，lambda只允许单个的返回值表达式，因为它不支持语句代码块，因此，不适用于较大的函数。

www.TopSage.com

第四部分 练习题

在这些练习中，你要开始编写更为成熟的程序。一定要看一看附录B中的解答，而且一定要在模块文件中编写代码。如果发生了错误的话，你是不会想从头输入这些练习的。

1. **基础。**在Python提示符下，编写一个函数将一个单独的参数打印至屏幕上，并以交互模式进行调用，传递各种对象类型：字符串、整数、列表、字典。然后，试着不传递任何参数进行调用。发生了什么？当你传两个参数时，发生了什么？

2. **参数。**在Python模块文件中编写一个名为adder的函数。这个函数应该接受两个参数，返回两者的和（或合并后的结果）。然后，在文件末尾增加代码，使用各种对象类型调用adder函数（两个字符串、两个列表、两个浮点数），然后，将这个文件当作脚本，从系统命令行运行。你是不是必须得打印调用语句的结果，才能在屏幕上查看结果？

3. **可变参数。**把上个练习题所编写的adder函数通用化，来计算任意数目的参数的和，然后修改调用方式，来传递两个以上或以下的参数。返回值的和的类型是什么？（提示：诸如S[:0]的切片会返回和S相同类型的空序列，而type内置函数可以测试类型；但是，可以参考第16章的例子min从而了解更简单的做法）如果传入不同类型的参数时，会发生什么？传入字典又是什么情况？

4. **关键字参数。**修改练习题2的adder函数，使其可以接受三个参数，并求其和/合并值：def adder(good, bad, ugly)。现在，为每个参数提供默认值，通过交互模式调用这个函数进行实验。试着传入一个、两个、三个以及四个参数。然后，试着传入关键字参数。adder(ugly=1, good=2)这样的调用方式能用吗？为什么？最后，把新的adder再通用化，从而可以接受任意数目的关键字参数，并求其和/合并值。这类似于你在练习题3所做的事，但是，你得遍历字典，而不是元组。（提示：dict.keys()方法会返回一个列表，你可以用for或while遍历）

5. **编写一个名为copyDict(dict)的函数，来赋值其字典参数。**这个函数应该返回新的字典，其中包含了其参数内所有的元素。使用字典keys方法来进行迭代（或者，在Python 2.2中，遍历字典的键时，不需要调用keys）。复制序列很简单（X[:]是做顶层复制）；字典也可以这么做吗？

6. **编写一个名为addDict(dict1, dict2)的函数，计算两个字典的并集。**这个函数应该返回新的字典，其中包含了它的两个参数中（假设为字典）所有的元素。如果两参数中有相同的键，可从其中之一挑选这个键的值。在文件中编写函数并将其作为脚本来执行，从而测试函数。如果你传入列表而不是字典的时候，会发生什么？你

www.TopSage.com

大家网
TopSage.com

怎么样把函数通用化从而能够处理这种情况？（提示：参考之前使用的type内置函数）传入参数的次序重要吗？

7. **更多关于参数匹配的例子。**首先，定义下面的六个函数（通过交互模式或者在可以导入的模块文件）。

```
def f1(a, b): print a, b              # Normal args

def f2(a, *b): print a, b             # Positional varargs

def f3(a, **b): print a, b            # Keyword varargs

def f4(a, *b, **c): print a, b, c     # Mixed modes

def f5(a, b=2, c=3): print a, b, c    # Defaults

def f6(a, b=2, *c): print a, b, c     # Defaults and positional varargs
```

现在，通过交互模式测试下面的调用，并尝试着来说明每个结果：在某些情况下，可能需要复习第16章的匹配算法。通常来说，混合匹配模式是个好主意吗？你能想到它所适用的情况吗？

```
>>> f1(1, 2)
>>> f1(b=2, a=1)

>>> f2(1, 2, 3)
>>> f3(1, x=2, y=3)
>>> f4(1, 2, 3, x=2, y=3)

>>> f5(1)
>>> f5(1, 4)

>>> f6(1)
>>> f6(1, 3, 4)
```

8. **重访质数。**回想第13章下列代码片段，它是用来简单的判断正整数是否为质数。

```
x = y / 2                             # For some y > 1
while x > 1:
    if y % x == 0:                    # Remainder
        print y, 'has factor', x
        break                         # Skip else
    x = x-1
else:                                 # Normal exit
    print y, 'is prime'
```

把这个代码封装成模块文件中可重用的函数，并在文件末尾增加一些函数的调用。写好后，把第一行的/运算符换为//，使其也能处理浮点数，并避开Python 3.0对

www.TopSage.com

/运算符计划中的真除法的改变（第5章所介绍的）。负数应该怎么做？0和1呢？如何加快其运行速度？输出如下所示。

```
13 is prime
13.0 is prime
15 has factor 5
15.0 has factor 5.0
```

9. **列表解析**。编写代码来创建新列表，其中包含了这个列表中所有数字的平方根：[2, 4, 9, 16, 25]。将它先写成for循环，然后是map调用，最后是列表解析。使用内置math模块中的sqrt函数来进行计算[导入math，编写math.sqrt(x)]。这三种做法中，你最喜欢哪一种？

www.TopSage.com

第五部分

模块

www.TopSage.com

大家网
TopSage.com

www.TopSage.com

第18章

模块：宏伟蓝图

从这一章开始，我们开始深入学习Python模块，模块是最高级别的程序组织单元，它将程序代码和数据封装起来以便重用。从实际的角度来看，模块往往对应于Python程序文件（或是用外部语言如C、Java或C#编写而成的扩展）。每一个文件都是一个模块，并且模块导入其他模块之后就可以使用导入模块定义的变量名。模块可以由两个语句和一个重要的内置函数进行处理。

`import`
: 使客户端（导入者）以一个整体获取一个模块。

`from`
: 允许客户端从一个模块文件中获取特定的变量名。

`reload`
: 在不中止Python程序的情况下，提供了一种重新载入模块文件代码的方法。

第3章介绍了模块文件的基础知识，并且之前也已经使用过这些知识。第4部分开始扩展了核心的模块文件的概念，之后开始探索更高级的模块应用。这一部分的第一章提供了一个模块文件在整个程序结构中扮演的角色的概览。在下一章及后边的章节，我们将会深入到理论背后的代码编写的细节。

在此过程中，我们将会得到曾经忽略了的模块的细节：你将会学到`reload`、`_name_`和`__all__`、封装`import`。因为模块和类实际上就是一个重要的命名空间，我们也会在这里正式介绍关于命名空间的概念。

为什么使用模块

简而言之，模块通过使用自包含的变量的包，也就是所谓的命名空间提供了将部件组织为系统的简单的方法。在一个模块文件的顶层定义的所有的变量名都成为了被导入的模

www.TopSage.com

块对象的属性。正如我们本书中前一部分中见到的那样，导入给予了对模块的全局作用域中的变量名的读取权。也就是说，在模块导入时，模块文件的全局作用域变成了模块对象的对象的命名空间。最后，Python的模块允许将独立的文件连接成一个更大的程序系统。

更确切地说，从抽象的视角来看，模块至少有三个角色，如下所示。

代码重用

就像在第3章中讨论过的那样，模块可以在文件中永久保存代码。不像在Python交互提示模式输入的代码，当退出Python时，代码就会消失，而在模块文件中的代码是永久的。你可以按照需要任意次数地重新载入和重新运行模块。除了这一点之外，模块还是定义变量名的空间，被认作是属性，可以被多个外部的客户端引用。

系统命名空间的划分

模块还是在Python中最高级别的程序组织单元。从根本上讲，它们不过是变量名的软件包。模块将变量名封装进了自包含的软件包，这一点对避免变量名的冲突很有帮助。如果不精确导入文件的话，我们是不可能看到另一个文件中的变量名。事实上，所有的一切都“存在于”模块文件中，执行的代码以及创建的对象都毫无疑问地封装在模块之中。正是由于这一点，模块是组织系统组件的天然的工具。

实现共享服务和数据

从操作的角度来看，模块对实现跨系统共享的组件是很方便的，而且只需要一个拷贝即可。例如，如果你需要一个全局对象，这个对象会被一个以上的函数或文件使用，你可以将它编写在一个模块中以便能够被多个客户端导入。

为了真正理解Python系统中模块的角色，那么，我们需要偏离主题来探索Python程序的通用结构。

Python程序构架

到现在为止,在本书中一些Python程序的复杂性,我们对之进行了简化。实际上，程序通常都不仅仅涉及到一个文件，除了最简单的脚本之外，程序都会采用多文件系统的形式。而且即使能够自己编写单个文件，几乎一定会使用到其他人已经写好的外部文件。这一部分介绍了通用的Python程序的构架:这种构架是将一个程序分割为源代码文件的集合以及将这些部分连接在一起的方法。在这个过程中，我们也将会探索Python模块、导入以及对象属性的一些核心概念。

www.TopSage.com

大家网 TopSage.com

如何组织一个程序

一般来讲，一个Python程序包括了多个含有Python语句的文本文件。程序是作为一个主体的、顶层的文件来构造的，配合有零个或多个支持的文件，在Python中这些文件称作模块。

在Python中，顶层文件包含了程序的主要的控制流程：这就是你需要运行来启动应用的文件。模块文件就是工具的库，这些工具是用来收集顶层文件（或者其他可能的地方）使用的组件。顶层文件使用了在模块文件中定义的工具，而这些模块使用了其他模块所定义的工具。

模块文件通常在运行时不需直接做任何事。然而，它们定义的工具会在其他文件中使用。在Python中，一个文件导入了一个模块来获得这个模块定义的工具的访问权，这些工具被认作是这个模块的属性（也就是说，附加类似于函数这样的对象上的变量名）。总而言之，我们导入了模块、获取它的属性并使用它的工具。

导入和属性

下面进行详细的介绍。图18-1一个包含有三个文件的Python程序的草图：*a.py*、*b.py*和*c.py*。文件a是顶层文件，它是一个含有语句的简单文本文件，在运行时这些语句将会从上至下执行。文件*b.py*和*c.py*是模块，它们也是含有语句的简单文本文件，但是它们通常并不是直接运行。就像之前解释的那样，取而代之的是，模块通常是被其他的文件导入的，这些文件想要使用这些模块所定义的工具。

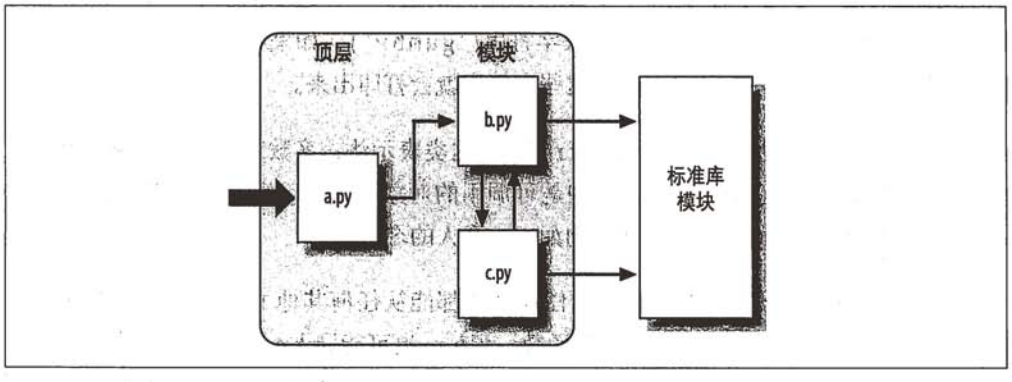


图18-1：Python的程序构架。一个程序是一个模块的系统。它有一个顶层脚本文件（启动后可运行程序）以及多个模块文件（用来导入工具库）。脚本和模块都是包含了Python语句的文本文件，尽管在模块中的语句通常都是创建在之后使用的对象。Python的标准库提供了一系列的预先编写好的模块

例如，图18-1中的文件*b.py*定义了一个名为spam的函数，供外部来使用。就像我们在第4部分提到的那样，*b.py*包含一个Python的def语句来生成函数，这个函数将会在之后通过给函数名后的括号中传入零个或更多的值来运行。

```
def spam(text):
    print text, 'spam'
```

现在，假设*a.py*想要使用spam。为了实现这个目标，*a.py*中也许就要包含如下这样的Python语句。

```
import b
b.spam('gumby')
```

首先，一个Python import语句，给文件*a.py*提供了由文件*b.py*在顶层所定义的所有对象的访问权限。概括来讲，这也就是意味着“载入文件*b.py*（除非它已经被载入了），并能够通过变量名b获取它的所有的属性”。import（以及，我们之后会见到的from）语句会在运行时并载入其他的文件。

在Python中；交叉文件的模块连接在导入语句执行时才会进行解析。实际效果就是，将模块名（简单的认为是变量名）赋值给载入的模块对象。事实上，在一个导入语句中的模块名起到两个作用：识别加载的外部文档，但是它也会变成赋值给被载入模块的变量。模块定义的对象也会在执行时创建，就在import执行时；import会一次运行在目标文档中的语句从而建立其中的内容。

*a.py*中的第二行语句调用了模块b中所定义的函数spam，使用了对象属性语法。代码b.spam指的是“取出存储对象b中变量名为spam的值。”在这个例子中，碰巧是个可调用的函数，所以，我们在小括号内传入字符串('gumby')。如果实际输入这些文件，保存之后，执行*a.py*，“gumby spam”这些字符串就会打印出来。

在Python脚本中随处见到object.attribute这类表示法：多数对象都有一些可用的属性，可以通过“.”运算符取出。有些是可调用的对象。例如，函数，而其他的则是简单数据数值，给予对象特定的属性（例如，一个人的名称）。

导入的概念在Python之中贯穿始末。任何文件都能从任何其他文件中导入其工具。例如，文件*a.py*可以导入*b.py*从而调用其函数，但*b.py*也可能导入*c.py*以利用在其中定义了的不同工具。导入链要多深就有多深：在这个例子中，模块a可导入b，而b可导入c，c可再导入b，诸如此类。

除了作为最高级别的组织结构外，模块（以及模块包，将在第20章中说明）也是Python中程序代码重用的最高层次。在模块文件中编写组件，可让原先的程序以及其他可能

www.TopSage.com

编写的任何程序得以使用。例如，编写图18-1程序后，我们发现函数b.spam是通用的工具，可在完全不同的程序中再次使用。我们所需要做的，就是从其他程序文件中再次导入文件*b.py*。

标准库模块

注意：图18-1最右侧位置。程序导入的模块是由Python自身提供的，而并非是程序员所编写的文件。

Python自带了很多实用的模块，称为标准链接库。这个集合体大约有200个模块，包含与平台不相关的常见程序设计任务：操作系统接口、对象永久保存、文字模式匹配、网络和Internet脚本、GUI建构等。这些工具都不是Python语言的组成部分，但是，你可以在任何安装了标准Python的情况下，导入适当的模块来使用。因为这些都是标准库模块，可以理所当然地认为它们一定可以用，而且在执行Python的绝大多数平台上都可运行。

本书例子中，你会看到一些标准库模块的运用，但是就整体而言，你应该查看Python标准库参考手册，这份手册在Python安装后就可以看到（通过Windows上的IDLE或“Python Start”按钮），或者也可以使用*http://www.python.org*的在线版本。

因为有如此繁多的模块，这是唯一了解有哪些工具可以使用的方式。你也可以在涉及应用级程序设计的商业书籍中找到Python链接库工具的教学，例如《Programming Python》，不过手册是免费的，可以用任何网页浏览器查看（属于HTML格式），而且每次Python发行时都会更新。

import如何工作

上一节谈到了导入模块，然而并没有实际解释当你这么做时都发生了什么。因为导入是Python中程序结构的重点所在，本节要深入讨论导入这个操作，让这个流程尽量不再那么抽象。

有些C程序设计者喜欢把Python的模块导入操作比作C语言中的#include，但其实不应该这么比较：在Python中，导入并非只是把一个文件文本插入另一个文件而已。导入其实是运行时的运算，程序第一次导入指定文件时，会执行三个步骤。

1. 找到模块文件。
2. 编译成位码（需要时）。
3. 执行模块的代码来创建其所定义的对象。

要对模块导入有更好的理解，我们将会逐一探索这些步骤。记住，这三个步骤只在程序执行时，模块第一次导入时才会进行。在这之后，导入相同模块时，会跳过这三个步骤，而只提取内存中已加载的模块对象。

TopSage.com

搜索

首先，Python必须查找到import语句所引用的模块文件。注意：上一节例子中的import语句所使用的文件名中没有.py，也没有目录路径，只有import b，而不是import c:\dir1\b.py。事实上，只能列出简单名称。路径和字尾是刻意省略掉的，因为Python使用了标准模块搜索路径来找出import语句所对应的模块文件（注1）。因为这是程序员对于import操作所必须了解的主要部分，我们就仔细研究这个步骤吧。

模块搜索路径

在大多数情况下，可以依赖模块导入搜索路径的自动特性，完全不需要配置这些路径。不过，如果你想在用户间定义目录边界来导入文件，就需要知道搜索路径是如何运作的，并予以调整。概括地讲，Python的模块搜索路径是这些主要组件组合而成的结果。其中有些进行了预先定义，而其中有些你可以进行调整来告诉Python去哪里搜索。

1. 程序的主目录。
2. PYTHONPATH目录（如果已经进行了设置）。
3. 标准链接库目录。
4. 任何*.pth*文件的内容（如果存在的话）。

最后，这四个组件组合起来就变成了sys.path，它是下一部分详细介绍的的目录名称字符串的列表。搜索路径的第一和第三元素是自动定义的，但是因为Python会从头到尾搜索这些组件组合的结果，第二和第四元素，就可以用于拓展路径，从而包含你自己的源代码目录。以下是Python使用路径组件的方式。

注1：在标准的import中引入路径和后缀名等细节，从语法上讲是非法的。第20章会讨论的包导入，可以让import语句在文件开头引入目录路径部分，成为一组以点号相隔的变量名。然而，包的导入依然依赖普通模块搜索路径，来找出包路径最左侧的目录（也就是相对于搜索路径中的目录）。包导入也不能在import语句中使用平台特定的目录语法；这类语法只能使用在搜索路径上。此外，值得注意的是，当冻结可执行文件时（在第2章讨论过），就不关心模块文件搜索路径的问题了，因为这种可执行文件通常是把字节码嵌入到二进制代码中。

www.TopSage.com

主目录

Python首先会在主目录内搜索导入的文件。根据启动代码的方式，这可能是包含程序的顶层文件的目录，或者是使用交互模式所在的目录。因为这个目录总是先被搜索，如果程序完全位于单一目录，所有导入都会自动工作，而并不需要配置路径。

PYTHONPATH目录

之后，Python会从左至右（假设你的设置了的话）搜索PYTHONPATH环境变量设置中罗列出的所有目录。简而言之，PYTHONPATH是设置包含Python程序文件的目录的列表,这些目录可以是用户定义的或平台特定的目录名。你可以把想导入的目录都加进来，而Python会使用你的设置来扩展模块搜索的路径。

因为Python会先搜索主目录，当导入的文件跨目录时，这个设置值才显得格外重要。也就是说，如果你需要被导入的文件与进行导入的文件处在不同目录时。当开始编写具体的程序时，也许就想设值PYTHONPATH变量了，但是刚刚入门时，只要把所有模块文件放在交互模式所使用的目录下就行了（也就是说，主目录），如此一来，导入都可运作，而你不需要去担心如何进行这个设置。

标准库目录

接着，Python会自动搜索标准库模块安装在机器上的那些目录。因为这些一定会被搜索，通常是不需要添加到PYTHONPATH之中的。

.pth 文件目录

最后，Python有个相当新的功能，允许用户把有效的目录添加到模块搜索路径中去，也就是在后缀名为*.pth*（路径的意思）的文本文件中一行一行地列出目录。这些路径配置文件是和安装相关的高级功能，我们不会在这里进行全面的讨论。

简而言之，当内含目录名称的文本文件放在适当目录中时，也可以概括地地扮演与PYTHONPATH环境变量设置相同的角色。例如，名为*myconfig.pth*的文件可以放在Windows中Python安装目录的顶层（例如，在*C:\Python25*或*C:\Python25\Lib\site-packages*中），来扩展模块搜索路径。Python会把文件每行所罗列的目录，从头至尾加至模块搜索路径列表的最后。因为这些是文件，而不是shell设置值，路径文件可以适用于所安装系统的所有用户，而并非仅限于一个用户或者一个shell。

这个特性比这里介绍的更为复杂。相关细节，可参考Python标准库手册（尤其是标准库模块网站的说明文档）。建议初学者使用PYTHONPATH或单个的*.pth*文件，并且只有进行跨目录的导入时才使用。参考附录A有关PYTHONPATH或*.pth*文件在各种平台上扩展模块搜索路径的常见方式。

这里对模块搜索路径的说明已很精确，但只算一般性说明：搜索路径的配置可能随平台

以及Python版本而异。取决于你所使用的平台，附加的目录也可能自动加入模块搜索路径。

例如，Python可能会把当前的工作目录也加进来（也就是启动程序所在的目录），放在搜索路径PYTHONPATH之后，并在标准库这项之前。在从命令行启动时，当前工作目录和顶层文件的主目录（也就是程序文件所在的目录）不一定相同（注2）。因为每次程序执行时，当前工作目录可能都会变，一般来说，不应该依赖这个值进行导入（注3）。

sys.path列表

如果你想看看模块搜索路径在机器上的实际配置，可以通过打印内置的sys.path列表（也就是标准库模块sys的path属性）来查看这个路径,就好像Python知道一样。目录名称字符串列表就是Python内部实际的搜索路径。导入时，Python会由左至右搜索这个列表中的每个目录。

其实，sys.path是模块搜索的路径。Python在程序启动时进行配置，自动将PYTHONPATH和.pth文件的路径设置值合并到这个列表中，并设置第一项作为顶层文件的主目录（可能是一个空字符串）。

Python出于两种合理的理由来描述这个列表。首先，提供一种方式来确认你所做的搜索路径的设置值：如果在清单中看不到设置值，就需要重新检查你的设置。其次，如果你知道在做什么，这个列表也提供一种方式，让脚本手动调整其搜索路径。本书这一部分后面就会知道，通过修改sys.path这个列表，你可以修改将来的导入的搜索路径。然而，这种修改只会在脚本存在期间保持而已。PYTHONPATH和*.pth*文件提供了更持久的路径修改方法（注4）。

模块文件选择

记住，文件名的后缀（例如，*.py*）是刻意从import语句中省略的。Python会选择在搜索路径中第一个符合导入文件名的文件。例如，import b形式的import叙述可能会加载。

注2：更多内容请参考第3章有关从命令行启动程序。

注3：参考第21章对Python 2.5新的相对导入语法的讨论。使用“.”字符时（例如，from .import string），将会修改from语句的搜索路径。

注4：不过，有些程序确实需要修改sys.path。例如，在网站服务器上执行的脚本通常会以用户“nobody”的身份执行，从而限制机器的读取权限。因为这类脚本通常用“nobody”以特定方式来设置PYTHONPATH，因此，在执行任何import语句前，通常会手动设置sys.path来将所需的源代码目录包括在内。通常使用sys.path.append(dirname)就可以了。

www.TopSage.com

大家网
TopSage.com

- 源代码文件*b.py*。
- 字节码文件*b.pyc*。
- 目录*b*（包导入；在第20章说明）。
- 编译扩展模块（通常用C或C++编写），导入时使用动态连接（例如，Linux的*b.so*以及Cygwin和Windows的*b.dll*或*b.pyd*）。
- 用C编写的编译好的内置模块，并通过静态连接至Python。
- ZIP文件组件，导入时会自动解压缩。
- 内存内映像，对于frozen可执行文件。
- Java类，在Jython版本的Python中。
- .NET组件，在IronPython版本的Python中。

C扩展、Jython以及包导入，都不仅仅是简单文件的导入机制的延伸。不过，对导入者来说，完全忽略了需要加载的文件类型之间的差异，无论是在导入时或者是在读取模块的属性时都是这样。例如，`import b`就是读取模块b，根据模块搜索路径，b是什么就是什么，而`b.attr`则是取出模块中的一个元素，可能是Python变量或连接的C函数。本书所用的某些标准模块实际上是用C编写的而不是Python。正是因为这种透明度，客户端并不在乎文件是什么。

如果在不同目录中有*b.py*和*b.so*，Python总是在由左至右搜索`sys.path`时，加载模块搜索路径那些目录中最先出现（最左边的）的相符文件。但是，如果是在相同目录中找到*b.py*和*b.so*，会发生什么事情呢？在这种情况下，Python遵循一个标准的挑选顺序，不过这种顺序不保证永远保持不变。通常来说，你不应该依赖Python会在给定的目录中选择何种的文件类型：让模块名独特一些，或者设置模块搜索路径，让模块选择的特性更明显一些。

高级的模块选择概念

一般来说，导入工作起来就像这一部分所介绍的那样：在机器上搜索并载入文件。然而，重新定义Python中import操作所做的事也是可能的，也就是使用所谓的导入钩子（import hook）。这些钩子可以让导入做各种有用的事，例如，从归档中加载文件，执行解密等。事实上，Python使用这些钩子让文件可直接从ZIP文件库中导入：当选择一个在搜索路径中的*.zip*文件后，归档后的文件会在导入时自动解压缩。更多细节，可以参考Python标准库手册中关于内置的`__import__`函数的说明，这个函数是实际执行的`import`语句的可定制的工具。

Python也支持最佳化字节码文件.*pyo*，这种文件在创建和执行时要加上`-O` 这个Python标志位。因为这些文件执行时会比普通的.*pyc*文件快一点（一般快5%），然而，它们并没有频繁的使用。Psyco系统（参考第2章）提供更实质性的加速效果。

编译（可选）

遍历模块搜索路径，找到符合`import`语句的源代码文件后，如果必要的话，Python接下来会将其编译成字节码（我们在第2章讨论过字节码）。

Python会检查文件的时间戳，如果发现.*pyc*字节码文件不比对应的.*py*源代码文件旧，就会跳过源代码到字节码的编译步骤。此外，如果Python在搜索路径上只发现了字节码文件，而没有源代码，就会直接加载字节码。换句话说，如果有可能使程序的启动提速，就会跳过编译步骤。如果修改了源代码，下次执行程序时，Python会自动重新生成字节码。再者，你可以只以字节码文件的形式分发程序，从而避免公开源代码。

注意：当文件导入时，就会进行编译。因此，通常不会看见程序顶层文件的.*pyc*字节码文件，除非这个文件也被其他文件导入：只有被导入的文件才会在机器上留下.*pyc*。顶层文件的字节码是在内部使用后就丢弃了；被导入文件的字节码则保存在文件中从而可以提高之后导入的速度。

顶层文件通常是设计成直接执行，而不是被导入的。稍后我们将会看到，设计一个文件，使其作为程序的顶层文件，并同时扮演被导入的模块工具的角色也是有可能的。这类文件既能执行也能导入，因此，的确会产生.*pyc*。要了解其运作方式，可参考第21章关于特定的`__name__`属性以及`__main__`的讨论。

运行

`import`操作最后步骤是执行模块的字节码。文件中所有语句会依次执行，从头至尾，而此步骤中任何对变量名的赋值运算，都会产生所得到的模块文件的属性。因此，这个执行步骤会生成模块代码所定义的所有工具。例如，文件中的`def`语句会在导入时执行，来创建函数，并将模块内的属性赋值给那些函数。之后，函数就能被程序中这个文件的导入者来调用。

因为最后的导入步骤实际上是执行文件的程序代码，如果模块文件中任何顶层代码确实做了什么实际的工作，你就会在导入时看见其结果。例如，当一个模块导入时，该模块内顶层的`print`语句就会显示其输出。函数的`def`语句只是简单地定义了稍后使用的对象。

www.TopSage.com

正如你所见到的那样，import操作包括了不少的操作：搜索文件、或许会运行一个编译器以及执行Python代码。因此，任何给定的模块在默认情况下每个进程中只会导入一次。未来的导入会跳过导入的这三个步骤，重用已加载内存内的模块（注5）。如果你在模块已加载后还需要再次导入（例如，为了支持终端用户的定制），你就得通过调用reload（下一章我们将会学到的一个工具）强制处理这个问题。

第三方工具：distutils

本章对模块搜索路径设置的说明，主要是针对自己编写的用户定义的源代码。Python的第三方扩展，通常使用标准链接库中的distutils工具来自动安装，所以不需要路径设置，就能使用它们的代码。

使用distutils的系统一般都附带*setup.py*脚本，执行这个脚本可以进行程序的安装。这个脚本会导入并使用distutils模块，将这种系统放在属于模块自动搜索路径一部分的目录内（通常是在Python安装目录树下的*Lib\site-packages*子目录中，而不管Python安装到了目标机器的哪一部分）。

更多关于distutils分发和安装的细节，可参考Python标准手册集。它的使用不在本书范围之内（例如，此工具还提供一些方式，可在目标机器上自动编译C所编写的扩展）。此外，参考最近涌现出的第三方开源*eggs*系统，它增加了对已安装的Python软件的依存关系的检查。

本章小结

在这一章中，我们介绍了模块、属性以及导入的基础知识，并探索了import语句的操作。我们也学到，导入会在模块搜索路径上寻找指定的文件，将其编译成字节码，并执行其中的所有语句从而产生其内容。我们也学到如何配置搜索路径，以便于从主目录和标准库目录以外的其他目录进行导入，主要是通过对PYTHONPATH的设置来实现的。

正如本章所展示过的，import操作和模块是Python之中程序架构的核心。较大程序可分成几个文件，利用导入在运行时连结在一起。而导入会使用模块搜索路径来寻找文件，并且模块定义了属性，供外部使用。

注5：从技术角度来讲，Python把已加载的模块放在内置的sys.modules字典内，在import操作开始时会确认引用的模块是否已加载。如果你想看看有哪些模块已加载，可以导入sys，打印sys.modules.keys()。第21章会再次介绍这张内部的表格。

www.TopSage.com

当然，导入和模块就是为程序提供结构，让程序将其逻辑分割成一些独立完备的软件组件。一个模块中的程序代码和另一个的程序代码彼此隔离。事实上，没有文件可以看到另一个文件中定义的变量名，除非明确地运行`import`语句。因此，模块最小化了程序内不同部分之间的变量名冲突。

下一章从实际的代码的观点看，就会了解其中的含义。但在继续之前，让我们先做完本章的习题吧。

www.TopSage.com

本章习题

1. 模块源代码文件是怎样变成模块对象的?
2. 为什么需要设置PYTHONPATH环境变量?
3. 举出模块导入搜索路径的四个主要组件。
4. 举出Python可能载入的能够响应import操作的四种文件类型。
5. 什么是命名空间? 模块的命名空间包含了什么?

习题解答

1. 模块的源代码文件在模块导入时，就会自动生成模块对象。从技术角度来讲，模块的源代码会在导入时运行，一次一条语句，而在这个过程中赋值的所有变量名都会生成模块对象的属性。
2. 只需设置PYTHONPATH，从而可以从正在用的目录（也就是正在交互模式下使用的当前目录，或者包含顶层文件的目录）以外的其他目录进行导入。
3. 模块导入搜索路径的四个主要组件是顶层脚本的主目录（包含该文件的目录）、列在PYTHONPATH环境变量中的所有目录、标准链接库目录以及位于标准位置中*.pth*路径文件中的所有目录。其中，程序员可以定制PYTHONPATH和*.pth*文件。
4. Python可能载入源代码文件（*.py*）、字节码文件（*.pyc*）、C扩展模块（例如，Linux的*.so*文件，以及Windows的*.dll*或*.pyd*）以及相同变量名的目录（用于包导入）。导入也可以加载更罕见的东西，例如，ZIP文件组件、Python Jython版的Java类、IronPython的.NET组件以及没有文件形式的静态连结C扩展。有了导入钩子，导入可以加载任何东西。
5. 命名空间是一种独立完备的变量包，而变量就是命名空间对象的属性。模块的命名空间包含了代码在模块文件顶层赋值的所有变量名（也就是没有嵌套于def或class语句中）。从技术角度上来讲，模块的全局作用域会变成模块对象的属性命名空间。模块的命名空间也会将其导入的其他文件中所做的赋值运算而发生变化，不过这不值得鼓励（参考第16章的相关内容）。

www.TopSage.com

第19章

模块代码编写基础

现在，我们已经接触了一些模块的重要概念，下面我们来看一个简单的模块实例吧。Python模块的创建很简单，只不过是用文本编辑器创建的Python程序代码档案而已。不需要编写特殊语法去告诉Python现在正在编写模块，几乎任何文本文件都可以。因为Python会处理寻找并加载模块的所有细节，所以模块很容易使用。客户端只需导入模块，就能使用模块定义的变量名以及变量名所引用的对象。

模块的创建

定义模块，只要使用文本编辑器，把一些Python代码输入至文本文件中，然后以“.py”为后缀名进行保存，任何此类文件都会被自动认为是Python模块。在模块顶层指定的所有变量名都会变成其属性（与模块对象结合的变量名），并且可以导出供客户端来使用。

例如，如果在名为*module1.py*的文件中输入下面的`def`语句，并将这个文件导入，就会创建一个拥有一个属性的模块对象：变量名`printer`，而这个变量名恰巧引用了一个函数对象。

```
def printer(x):                    # Module attribute
    print x
```

在继续学习之前，我们应该对模块文件名再多介绍一下。模块怎么命名都可以，但是如果打算将其导入，模块文件名就应该以*.py*结尾。对于会执行但不会被导入的顶层文件而言，*.py*从技术上来讲是可有可无的，但是每次都加上去，可以确保文件类型更醒目，并允许以后可以导入任何文件。

因为模块名在Python程序中会变成变量名（没有*.py*）。因此，应该遵循第11章所提到的普通变量名的命名规则。例如，你可以建立名为*if.py*的模块文件，但是无法将其导入，

www.TopSage.com

因为if是保留字。当尝试执行import if时，会得到语法错误。事实上，包导入中所用的模块的文件名和目录名（下一章讨论），都必须遵循第11章所介绍的变量名规则。例如，只能包含字母、数字以及下划线。包的目录也不能包含平台特定的语法，例如，名称中有空格。

当一个模块被导入时，Python会把内部模块名映射到外部文件名，也就是通过把模块搜索路径中的目录路径加在前边，而.py或其他后缀名添加在后边。例如，名为M的模块最后会映射到某个包含模块程序代码的外部文件：*<directory>\M.<extension>*。

正像上一张所提到的那样，也有可能使用C或C++（或Java，Python这门语言的Jython实现）这类外部语言编写代码来创建Python模块。这类模块称做扩展模块，一般都是在Python脚本中作为包含外部扩展库来使用的。当被Python代码导入时，扩展模块的外观和用法与Python源代码文件所编写的模块一样：也是通过import语句进行读取，并提供函数和对象作为模块属性。扩展模块不在本书讨论范围之内。参考Python的标准手册，或者参考更高级的书籍，如《Programming Python》来获得更多细节。

模块的使用

客户端可以执行import或from语句，以使用我们刚才编写的简单模块文件。如果模块还没有加载，这两个语句就会去搜索、编译以及执行模块文件程序。主要的差别在于，import会读取整个模块，所以必须进行定义后才能读取它的变量名；from将获取（或者说是复制）模块特定的变量名。

让我们从代码的角度来看这意味着什么吧。下面所有的范例最后都是调用外部模块文件*module1.py*所定义的printer函数，只不过使用了不同的方法。

import语句

在第一个例子中，变量名module1有两个不同目的：识别要被载入的外部文件，同时会生成脚本中的变量，在文件加载后，用来引用模块对象。

```
>>> import module1                  # Get module as a whole
>>> module1.printer('Hello world!')  # Qualify to get names
Hello world!
```

因为import使一个变量名引用整个模块对象，我们必须通过模块名称来得到该模块的属性（例如，module1.printer）。

www.TopSage.com

from语句

因为from会把变量名复制到另一个作用域，所以它就可以让我们直接在脚本中使用复制后的变量名，而不需要通过模块（例如，printer）。

```
>>> from module1 import printer        # Copy out one variable
>>> printer('Hello world!')            # No need to qualify name
Hello world!
```

这和上一个例子有着相同的效果，但是from语句出现时，导入的变量名会复制到作用域内，在脚本中使用该变量名就可少输入一些：我们可直接使用变量名，而无须在嵌套模块名称之后。

稍后我们会更详细地介绍，from语句其实只是稍稍扩展了import语句而已。它照常导入了模块文件，但是多了一个步骤，将文件中的一个或多个变量名从文件中复制了出来。

from *语句

最后，下一个例子使用特殊的from形式：当我们使用*时，会取得模块顶层所有赋了值的变量名的拷贝。在这里，我们还是在脚本中使用复制后得到的变量名printer，而不需要通过模块名。

```
>>> from module1 import *              # Copy out all variables
>>> printer('Hello world!')
Hello world!
```

从技术角度来说，import和from语句都会使用相同的导入操作。from *形式只是多加个步骤，把模块中所有变量名复制到了进行导入的作用域之内。从根本上来说，这就是把一个模块的命名空间融入另一个模块之中；同样地，实际效果就是可以让我们少输入一些。

就是这样，模块使用起来其实很容易。不过，为了进一步了解定义和使用模块时，究竟会发生什么，我们详细地看一下它们的某些特性吧。

导入只发生一次

使用模块时，初学者最常问的问题之一似乎就是："为什么我的导入不是一直有效?"他们时常报告说，第一次导入运作良好，但是在交互式会话模式（或程序运行）期间,之后的导入似乎就没有效果。事实上，本来就应该如此，原因如下。

模块会在第一次import或from时载入并执行，并且只在第一次如此。这是有意而为之

www.TopSage.com

的，因为该操作开销较大。在默认的情况下，Python只对每个文件的每个进程做一次操作。之后的导入操作都只会取出已加载的模块对象。

结果，因为模块文件中的顶层程序代码通常只执行一次，你可以凭借这种特性对变量进行初始化。例如，考虑下面文件*simple.py*。

```
print 'hello'
spam = 1                    # Initialize variable
```

此例中，print和=语句在模块第一次导入时执行，而变量spam也在导入时初始化：

```
% python
>>> import simple           # First import: loads and runs file's code
hello
>>> simple.spam             # Assignment makes an attribute
1
```

第二次和其后的导入并不会重新执行此模块的代码，只是从Python内部模块表中取出已创建的模块对象。因此，变量spam不会再进行初始化：

```
>>> simple.spam = 2         # Change attribute in module
>>> import simple           # Just fetches already loaded module
>>> simple.spam             # Code wasn't rerun: attribute unchanged
2
```

当然，有时需要一个模块的代码通过某种导入后再一次运行。我们将会在本章稍后介绍如何使用内置函数reload实现这种操作。

import和from是赋值语句

就像def一样，import和from是可执行的语句，而不是编译期间的声明，而且它们可以嵌套在if测试中，出现在函数def之中等等，直到执行程序时，Python执行到这些语句，才会进行解析。换句话来说，被导入的模块和变量名，直到它们所对应的import或from语句执行后，才可以使用。此外，就像def一样，import和from都是隐性的赋值语句。

- import将整个模块对象赋值给一个变量名。
- from将一个或多个变量名赋值给另一个模块中同名的对象。

我们谈过的关于赋值语句方面的内容，也适用于模块的读取。例如，以from复制的变量名会变成对共享对象的引用。就像函数的参数，对已取出的变量名重新赋值，对于其复制之处的模块并没有影响，但是修改一个已取出的可变对象，则会影响导入的模块内的对象。为了解释清楚，思考一下下面的文件*small.py*。

www.TopSage.com

```
x = 1
y = [1, 2]

% python
>>> from small import x, y          # Copy two names out
>>> x = 42                          # Changes local x only
>>> y[0] = 42                       # Changes shared mutable in-place
```

此处，x并不是一个共享的可变对象，但y是。导入者中的变量名y和被导入者都引用相同的列表对象，所以在其中一个地方的修改，也会影响另一个地方的这个对象。

```
>>> import small                    # Get module name (from doesn't)
>>> small.x                         # Small's x is not my x
1
>>> small.y                         # But we share a changed mutable
[42, 2]
```

对于赋值语句和引用之间的图形关系，可以重新查看图16-2（函数参数传递），只要在心中把“调用者”和“函数”换成“被导入模块”和“导入者”即可。实际效果是相同的，只不过我们现在面对的是模块内的变量名，而不是函数。在Python中，赋值语句工作起来都是一样的。

文件间变量名的改变

回忆前边的例子中，在交互会话模式下对x的赋值运算，只会修改该作用域内的变量x，而不是这个文件内的x。以from复制而来的变量名和其来源的文件之间并没有联系。为了实际修改另一个文件中的全局变量名，必须使用import。

```
% python
>>> from small import x, y          # Copy two names out
>>> x = 42                          # Changes my x only

>>> import small                    # Get module name
>>> small.x = 42                    # Changes x in other module
```

这种现象已在第16章介绍过。因为像这样修改其他模块内的变量是常常困惑开发人员的原因之一（通常也是不良设计的选择），本书这一部分稍后会再谈到这个技巧。注意：前一个会话中对y[0]的修改是不同的。这是修改了一个对象，而不是一个变量名。

import和from的对等性

注意：在上一个例子中，我们需要在from后执行import语句，来获取small模块的变量。from只是把变量名从一个模块复制到另一个模块，并不会对模块名本身进行赋值。至少从概念上来说，一个像这样的from语句：

www.TopSage.com

```
from module import name1, name2      # Copy these two names out (only)
```

与下面这些语句就是等效的：

```
import module                        # Fetch the module object
name1 = module.name1                 # Copy names out by assignment
name2 = module.name2
del module                           # Get rid of the module name
```

就像所有赋值语句一样，from语句会在导入者中创建新变量，而那些变量初始化时引用了导入文件中的同名对象。不过，只有变量名被复制出来，而非模块本身。当我们使用语句from *这种形式时（from module import *），等效的写法是一样的，只不过是模块中所有的顶层变量名都会以这种方式复制到进行导入的作用域中。

注意：from的第一步骤也是普通的导入操作。因此，from总是会把整个模块导入到内存中（如果还没被导入的话），无论是从这个文件中复制出多少变量名。只加载模块文件的一部分（例如，一个函数）是不可能的，但是因为模块在Python之中是字节码而不是机器码，通常可以忽略效率的问题。

from语句潜在的陷阱

因为from语句会让变量位置更隐密和模糊（与module.name相比，name对读者而言意义不大），有些Python用户多数时候推荐使用import而不是from。不过不确定这种建议是否有根据。from得到了广泛的应用，也没太多可怕的结果。实际上，在现实的程序中，每次想使用模块的工具时，省略输入模块的变量名，通常是很方便的。对于提供很多属性的大型模块而言更是如此。例如，标准库中的Tkinter GUI模块。

from语句有破坏命名空间的潜质，这是真的，至少从理论上讲是这样的。如果使用from导入变量，而那些变量碰巧和作用域中现有变量同名，变量就会被悄悄地覆盖掉。使用简单import语句时就不存在这种问题，因为你一定得通过模块名才能获取其内容（module.attr不会和你的作用域内的名为attr的变量相冲突）。不过，使用from时，只要你了解并预料到可能发生这种事，在实际情况下这就不是一个大问题了，尤其当你明确列出导入的变量名时（例如，from module import x, y, z）。

另一方面，和reload调用同时使用时，from语句有比较严重的问题，因为导入的变量名可能引用之前版本的对象。再者，from module import *形式的确可能破坏命名空间，让变量名难以理解，尤其是在导入一个以上的文件时。在这种情况下，没有办法看出一个变量名来自哪个模块，只能搜索外部的源代码文件。事实上，from *形式会把一个命

名空间融入到另一个，所以会使得模块的命名空间的分割特性失效。我们会在本书这一部分的最后的“模块陷阱”这一部分再深入探讨这些话题（参考第21章）。

此处，也许真正务实的建议就是：简单模块一般倾向于使用import，而不是from。多数的from语句是用于明确列举出想要的变量，而且限制在每个文件中只用一次from *形式。这样一来，任何无定义的变量名都可认为是存在于from *所引用的模块内。使用from语句时，的确要小心一点，但是只要有些常识，多数程序员都会发现这是一种方便的存取模块的方式。

何时使用import

当你必须使用两个不同模块内定义的相同变量名的变量时，才真的必须使用import，这种情况下不能用from。例如，如果两个文件以不同方式定义相同变量名。

```
# M.py

def func():
    ...do something...

# N.py

def func():
    ...do something else...
```

而你必须在程序中使用这两个版本的变量名时，from语句就不能用了。作用域内一个变量名只能有一个赋值语句。

```
# O.py

from M import func
from N import func          # This overwites the one we got from M
func()                      # Calls N.func only
```

不过，只用一个import就可以，因为把所在模块变量名加进来，会让两个变量名都是唯一的。

```
# O.py

import M, N                 # Get the whole modules, not their names
M.func()                    # We can call both names now
N.func()                    # The module names make them unique
```

这种情况很少见，在实际情况中，不太可能遇见。

www.TopSage.com

模块命名空间

模块最好理解为变量名的封装，也就是定义想让系统其余部分看见变量名的场所。从技术上来讲，模块通常相应于文件，而Python会建立模块对象，以包含模块文件内所赋值的所有变量名。但是，简而言之，模块就是命名空间（变量名建立所在的场所），而存在于模块之内的变量名就是模块对象的属性。我们会在本节讨论这是如何运作的。

文件生成命名空间

那么，文件如何变成命名空间的呢？简而言之，在模块文件顶层（也就是不在函数或类的主体内）每一个赋值了的变量名都会变成该模块的属性。

例如，假设模块文件*M.py*的顶层有一个像`X = 1`这样的赋值语句，而变量名`X`会变成`M`的属性，我们可在模块外以`M.X`的方式对它进行引用。变量名`X`对*M.py*内其他程序而言也会变成全局变量，但是，我们需要更正式地说明模块加载和作用域的概念以了解其原因。

- **模块语句会在首次导入时执行。**系统中，模块在第一次导入时无论在什么地方，Python都会建立空的模块对象，并逐一执行该模块文件内的语句，依照文件从头到尾的顺序。
- **顶层的赋值语句会创建模块属性。**在导入时，文件顶层（不在`def`或`class`之内）赋值变量的语句（例如，`=`和`def`），会建立模块对象的属性；赋值的变量名会储存在模块的命名空间内。
- **模块的命名空间能通过属性`__dict__`或`dir(M)`获取。**由导入而建立的模块的命名空间是字典；可通过模块对象相关联的内置的`__dict__`属性来读取，而且能通过`dir`函数查看。`dir`函数大至与对象的`__dict__`属性的键排序后的列表相等，但是它还包含了类继承的变量名。也许不完整，而且会随版本而异。
- **模块是一个独立的作用域（本地变量就是全局变量）。**正如第16章所显示的，模块顶层变量名遵循和函数内变量名相同的引用/赋值规则，但是，本地作用域和全局作用域相同（更正式的说法是，遵循我们于第16章提及的LEGB范围规则，但是没有L和E搜索层次）。但是，在模块中，模块范围会在模块加载后变成模块对象的属性辞典。和函数不同的是（本地变量名只在函数执行时才存在），导入后，模块文件的作用域就变成了模块对象的属性的命名空间。

以下是这些概念的示范说明。假设我们在文本编辑器中建立如下的模块文件，并将其命名为*module2.py*。

www.TopSage.com

```
print 'starting to load...'

import sys
name = 42

def func(): pass

class klass: pass

print 'done loading.'
```

这个模块首次导入时（或者作为程序执行时），Python会从头到尾执行其中的语句。有些语句会在模块命名空间内创建变量名，也就是副作用，而其他的语句在导入进行时则会做些实际工作。例如，此文件中的两个print语句会在导入时执行。

```
>>> import module2
starting to load...
done loading.
```

但是，一旦模块加载后，它的作用域就变成模块对象（由import取得）的属性的命名空间。然后，我们可以结合其模块名，通过它来获取命名空间内的属性。

```
>>> module2.sys
<module 'sys' (built-in)>

>>> module2.name
42

>>> module2.func
<function func at 0x012B1830>

>>> module2.klass
<class module2.klass at 0x011C0BA0>
```

此处，sys、name、func以及klass都是在模块语句执行时赋值的，所以在导入后都变成了属性。我们会在第6部分讨论类，但是请注意sys属性：import语句其实是把模块对象赋值给变量名，而文件顶层对任意类型赋值了的变量名，都会产生模块属性。

在内部模块命名空间是作为辞典对象进行储存的。它们只是普通字典对象，有通用的字典方法可以使用。我们可以通过模块的__dict__属性获取模块命名空间字典：

```
>>> module2.__dict__.keys()
['__file__', 'name', '__name__', 'sys', '__doc__', '__builtins__',
'klass', 'func']
```

我们在模块文件中赋值的变量名，在内部成为字典的键。因此这里多数的变量名都反映了文件中的顶层的赋值语句。然而，Python也会在模块命名空间内加一些变量名。例

www.TopSage.com

如，__file__指明模块从哪个文件加载，而__name__则指明导入者的名称（没有.py扩展名和目录路径）。

属性名的点号运算

现在，熟悉了模块的基本知识，我们应该深入探讨变量名点号运算（notion of name qualification）的概念。在Python之中，可以使用点号运算语法object.attribute获取任意的object的attribute属性。

点号运算其实就是表达式，传回和对象相配的属性名的值。例如，上一个例子中，表达式module2.sys会取出module2中赋值给sys的值。同样地，如果我们有内置的列表对象L，而L.append会返回和该列表相关联的append方法对象。

那么，属性的点号运算和第16章学过的作用域法则有什么关系呢？其实，二者没有关系 —— 这是不相关的概念。当使用点号运算来读取变量名时，就把明确的对象提供给Python，来从其中取出赋值的变量名。LEGB规则只适用于无点号运算的纯变量名。以下是其规则。

简单变量

X是指在当前作用域内搜索变量名X（遵循LEGB规则）。

点号运算

X.Y是指在当前范围内搜索X，然后搜索对象X之中的属性Y（而非在作用域内）。

多层点号运算

X.Y.Z指的是寻找对象X之中的变量名Y，然后再找对象X.Y之中的Z。

通用性

点号运算可用于任何具有属性的对象：模块、类、C扩展类型等。

在第6部分中，我们会看到点号运算对类（这也是继承发生的地方）的意义还要多一点，但是，一般而言，此处所列举的规则适用于Python中所有的变量名。

导入和作用域

正如我们所学过的，不导入一个文件，就无法存取该文件内所定义的变量名。也就是说，你不可能自动看见另一个文件内的变量名，无论程序中的导入结构或函数调用的结构是什么情况。变量的含义一定是由源代码中的赋值语句的位置决定的，而属性总是伴随着对对象的请求。

www.TopSage.com

例如，考虑以下两个简单模块。第一个模块*moda.py*只在其文件中定义一个全局变量X，以及一个可修改全局变量X的函数。

```
X = 88                          # My X: global to this file only

def f():
    global X                    # Change this file's X
    X = 99                      # Cannot see names in other modules
```

第二个模块*modb.py*定义自己的全局变量X，导入并调用了第一个模块的函数。

```
X = 11                          # My X: global to this file only

import moda                     # Gain access to names in moda
moda.f()                        # Sets moda.X, not this file's X
print X, moda.X
```

执行时，moda.f修改moda中的X，而不是modb中的X。moda.f的全局作用域一定是其所在的文件，无论这个函数是由哪个文件调用的：

```
% python modb.py
11 99
```

换句话说，导入操作不会赋予被导入文件中的代码对上层代码的可见度：被导入文件无法看见进行导入的文件内的变量名。更确切的说法是：

- 函数绝对无法看见其他函数内的变量名，除非它们从物理上处于这个函数内。
- 模块程序代码绝对无法看见其他模块内的变量名，除非明确的进行了导入。

这类行为是语法作用域范畴的一部分：在Python中，一段程序的作用域完全由程序所处的文件中实际位置决定。作用域绝不会被函数调用或模块导入影响（注1）。

命名空间的嵌套

就某种意义而言，虽然导入不会使命名空间发生向上的嵌套，但确实会发生向下的嵌套。利用属性的点号运算路径，有可能深入到任意嵌套的模块中并读取其属性。例如，考虑下列三个文件。*mod3.py*以赋值语句定义了一个全局变量名和属性：

```
X = 3
```

接着，*mod2.py*定义本文件内的X，然后导入mod3，使用点号运算来取所导入的模块的属性。

注1：有些语言的行为不同，提供所谓的动态作用域，也就是作用域其实依赖于运行期间的调用。然而，这样会让程序代码变得很难处理，因为变量的含义会随时间而发生变化。

www.TopSage.com

```
X = 2
import mod3

print X,                    # My global X
print mod3.X                # mod3's X
```

*mod1.py*也定义本文件内的X，然后导入mod2，并取出第一和第二个文件内的属性。

```
X = 1
import mod2

print X,                    # My global X
print mod2.X,               # mod2's X
print mod2.mod3.X           # Nested mod3's X
```

实际上，当这里的mod1导入mod2时，会创建一个两层的命名空间的嵌套。利用mod2.mod3.X变量名路径，就可深入到所导入的mod2内嵌套了的mod3。结果就是mod1可以看见三个文件内的X，因此，可以读取这三个全局范围。

```
% python mod1.py
2 3
1 2 3
```

然而，反过来讲，就没这回事了：mod3无法看见mod2内的变量名，而mod2无法看见mod1内的变量名。如果你不以命名空间和作用域的观点思考，而是把焦点集中在牵涉到的对象，这个例子就会比较容易掌握。在mod1中，mod2只是变量名，引用带有属性的对象，而该对象的某些属性可能又引用其他带有属性的对象（import是赋值语句）。对于mod2.mod3.X这类路径而言，Python只会由左至右进行计算，沿着这样的路径取出对象的属性。

注意到：mod1可以说import mod2，然后mod2.mod3.X，但是，无法说import mod2.mod3。这个语法牵涉到所谓的包（目录）导入，这将在下一章介绍。包导入也会形成模块命名空间嵌套，但是，其导入语句会反映目录树结构，而非简单的导入链。

重载模块

正如我们所见到过的，模块程序代码默认只对每个过程执行一次。要强制使模块代码重新载入并重新运行，你得刻意要求Python这么做，也就是调用reload内置函数。本节中，我们要探索如何使用reload让系统变得更加动态。简而言之：

- 导入（无论是通过import或from语句）只会模块在流程中第一次导入时，加载和执行该模块的代码。

www.TopSage.com

- 之后的导入只会使用已加载的模块对象，而不会重载或重新执行文件的代码。
- reload函数会强制使已加载的模块的代码重新载入并重新执行。此文件中新的代码的赋值语句会在适当的地方修改现有的模块对象。

为什么要这么麻烦去重载模块？reload函数可以修改程序的一些部分，而无须停止整个程序。因此，利用reload，可以立即看到对组件的修改的效果。重载无法用于每种情况，但是能用时，可缩短开发的流程。例如，想象一下，数据库程序必在启动时连接服务器，因为程序修改或调整可在重载后立即测试，在调试时，只需连接一次就可以了。

因为Python是解释性的（或多或少），其实已经避免了类似C语言程序执行时所需的编译/连结步骤：在执行程序导入时，模块会动态加载。重载进一步地提供了性能优势，让你可以修改执行中的程序的一部分，而不需要中止。注意：reload当前只能用在用Python编写的模块；用C这类语言编写的编译后的扩展模块也可在执行中动态加载，但无法重载。

reload基础

与import和from不同的是：

- reload是Python中的内置函数，而不是语句。
- 传给reload的是已经存在的模块对象，而不是变量名。

因为reload期望得到的是对象，在重载之前，模块一定是已经预先成功导入了（如果因为语法或其他错误使得导入没成功，你得继续试下去，否则将无法重载）。此外，import语句和reload调用的语法并不相同：reload需要小括号，但import不需要。重载看起来如下所示。

```
import module                    # Initial import
...use module.attributes...
...                              # Now, go change the module file
...
reload(module)                   # Get updated exports
...use module.attributes...
```

一般的用法是：导入一个模块，在文本编辑器内修改其原代码，然后将其重载。当调用reload时，Python会重读模块文件的源代码，重新执行其顶层语句。也许有关reload所需要知道的最重要的事情就是，reload会在适当的地方修改模块对象，reload并不会删除并重建模块对象。因此，程序中任何引用该模块对象的地方，自动会受到reload的影响。下面是一些细节。

www.TopSage.com

- **reload会在模块当前命名空间内执行模块文件的新代码。**重新执行模块文件的代码会覆盖其现有的命名空间，并非进行删除而进行重建。
- **文件中顶层赋值语句会使得变量名换成新值。**例如，重新执行的def语句会因重新赋值函数变量名而取代模块命名空间内该函数之前的版本。
- **重载会影响所有使用import读取了模块的客户端。**因为使用import的客户端需要通过点号运算取出属性，在重载后，它们会发现模块对象中变成了新的值。
- **重载只会对以后使用from的客户端造成影响。**之前使用from来读取属性的客户端并不会受到重载的影响，那些客户端引用的依然是重载前所取出的旧对象。

reload实例

这里是一个更具体的reload的例子。在下面这个例子中，我们要修改并重载一个模块文件，但是不会中止交互模式的Python会话。重载也在很多场景中使用（参考边栏内容），但是，为了解释清楚，我们举一个简单的例子。首先，在选择的文本编辑器中，编写一个名为*changer.py*的模块文件，其内容如下。

```
message = "First version"

def printer():
    print message
```

这个模块会建立并导入两个变量名：一个是字符串，另一个是函数。现在，启动Python解释器，导入该模块，然后调用其导出的函数。此函数会打印出全局变量message的值。

```
% python
>>> import changer
>>> changer.printer()
First version
```

不要关掉解释器，现在，在另一个窗口中编辑该模块文件。

```
...modify changer.py without stopping Python...
% vi changer.py
```

改变message变量和printer函数体：

```
message = "After editing"

def printer():
    print 'reloaded:', message
```

然后，回到Python窗口，重载该模块从而获得新的代码。注意下面的交互模式，再次导

www.TopSage.com

入该模块并没有效果。我们得到原始的message，即使文件已经修改过了。我们得调用reload，才能够获取新的版本。

```
...back to the Python interpreter/program...

>>> import changer
>>> changer.printer()          # No effect: uses loaded module
First version

>>> reload(changer)            # Forces new code to load/run
<module 'changer'>
>>> changer.printer()          # Runs the new version now
reloaded: After editing
```

注意：reload实际是为我们返回了模块对象：其结果通常被忽略，但是，因为表达式结果会在交互模式提示符下打印出来，Python会打印默认的<module 'name'>表现形式。

为什么要在意：模块重载

除了可以在交互式提示符号下重载（以及重新执行）模块外，模块重载在较大系统中也有用处，在重新启动整个应用程序的代价太大时尤其如此。例如，必须在启动时通过网络连接服务器的系统，就是动态重载的一个非常重要的应用场景。

重载在GUI工作中也很有用（组件的回调行为可以在GUI保持活动的状态下进行修改）。此外，当Python作为C或C++程序的嵌入式语言时，也有用处（C/C++程序可以请求重载其所执行的Python代码而无须停止）。参考《Programming Python》有关重载GUI回调函数和嵌入式Python程序代码的更多细节。

通常情况下，重载使程序能够提供高度动态的接口。例如，Python通常作为较大系统的定制语言：用户可以在系统运作时通过编写Python程序定制产品，而不用重新编译整个产品（或者甚至获取整个源代码）。这样，Python程序代码本身就增加了一种动态本质了。

不过，为了更具动态性，这样的系统可以在执行期间定期自动重载Python定制的程序代码。这样一来，当系统正在执行时，就可采用用户的修改；每次Python代码修改时，都不需要停止并重启。并非所有系统都需要这种动态的实现，但对那些需要的系统而言，模块重载就提供了一种易于使用的动态定制工具。

本章小结

本章深入讨论了模块编码工具的基础知识：import和from语句，以及reload调用。我们

www.TopSage.com

知道from语句只多加了一个步骤，在文件导入之后，将文件的变量名复制出来，也学习了reload如何不停止并重启Python而使文件再次导入。我们也研究了命名空间的概念，学习了当导入嵌套时会发生什么，探索了文件转换为模块的命名空间的过程，并学习了from语句潜在的一些陷阱。

虽然我们已在程序中见过很多处理模块文件的例子，但下一章要通过介绍包导入来扩展导入模块的相关内容。包导入是import语句赋值目录路径的部分的方式，以获取所需的模块。正如我们所看到的，包导入赋予我们一种层次架构，在较大型的系统中会有用处，而且可以避免同名模块间的冲突。不过，我们先做一做习题来复习本章介绍的概念。

www.TopSage.com

大家网
TopSage.com

本章习题

1. 怎样创建模块？
2. from语句和import语句有什么关系？
3. reload函数和导入有什么关系？
4. 什么时候必须使用import，不能使用from？
5. 请列举出from语句三种潜在陷阱。
6. 空载的燕子飞行速度是多少？

习题解答

1. 要创建模块时，只需编写一个包含Python语句的文本文件就可以了；每个原代码文件都会自动成为模块，而且也没有语法用来声明模块。导入操作会把模块文件加载到内存中使其成为模块对象。你以可以用C或Java这类外部语言编写代码来创建模块，但是这类扩展模块不在本书讨论范围之内。
2. from语句是导入整个模块，就像import语句那样，但是还有个步骤，就是会从被导入的模块中，复制一或多个变量到from语句所在的作用域中。这样可以让你直接使用被导入的变量名（name），而不是通过模块来使用变量名（module.name）。
3. 默认，模块是每个进程只导入一次。reload函数会强制模块再次被导入。这基本上都是用于开发过程中选取模块源代码的新版本，或者用在动态定制的场景中。
4. 当需要读取两个不同模块中的相同变量名时，就必须使用import，而不能用from；因为你必须制定变量名所在模块，从而保证这两个变量名是独特的。
5. from语句会让变量含义模糊（究竟是哪个模块定义的），通过reload调用时会有问题（变量名还是引用对象之前的版本），而且会破坏命名空间（可能悄悄覆盖正在作用域中使用的变量名）。from *形式在多数情况下都很糟糕：它会严重的污染命名空间，让变量意义变的模糊，最好少用为妙。
6. 你是什么意思？是非洲还是欧洲的燕子？

www.TopSage.com

第20章

模块包

到目前为止，我们已导入过模块，加载了文件。这是一般性模块用法，可能也是你早期Python职业生涯中多数导入会使用的技巧。然而，模块导入故事比目前所提到的还要丰富一点。

除了模块名之外，导入也可以指定目录路径。Python代码的目录就称为包，因此，这类导入就称为包导入。事实上，包导入是把计算机上的目录变成另一个Python命名空间，而属性则对应于目录中所包含的子目录和模块文件。

这是有点高级的特性，但是它所提供的层次，对于组织大型系统内的文件会很方便，而且可以简化模块搜索路径的设置。我们将知道，当多个相同名称的程序文件安装在某一机器上时，包导入也可以偶尔用来解决导入的不确定性。

包导入基础

那么，包导入是如何运作的呢？在`import`语句中列举简单文件名的地方，可以改成列出路径的名称，彼此以点号相隔。

```
import dir1.dir2.mod
```

`from`语句也是一样的：

```
from dir1.dir2.mod import x
```

这些语句中的“点号”路径是对应于机器上目录层次的路径，通过这个路径可以获得到文件*mod.py*（或类似文件，扩展名可能会有变化）。也就是说，上面的语句是表明了机器上有个目录*dir1*，而*dir1*里有子目录*dir2*，而*dir2*内包含有一个名为*mod.py*（或类似文件）的模块文件。

www.TopSage.com

此外，这些导入意味着，*dir1*位在某个容器目录*dir0*中，这个目录可以在Python模块搜索路径中找到。换句话说，这两个import语句代表了这样的目录结构（以DOS反斜线分隔字符显示）。

```
dir0\dir1\dir2\mod.py                # Or mod.pyc, mod.so, etc.
```

容器目录*dir0*需要添加在模块搜索路径中（除非这是顶层文件的主目录），就好像*dir1*是模块文件那样。从此以后，脚本内的import语句所指出能够找到模块的目录路径。

包和搜索路径设置

如果使用这个特性，要记住，import语句中的目录路径只能是以点号间隔的变量。你不能在import语句中使用任何平台特定的路径语法。例如，C:\dir1、My Documents.dir2或../dir1：这些从语法上讲是行不通的。你所需要做的就是，在模块搜索路径设置中使用平台特定的语法，来定义容器的目录。

例如，上一个例子中，*dir0*（加在模块搜索路径中的的目录名）可以是任意长度而且是与平台相关的目录路径，在其下能够找到*dir1*。而不是使用像这样的无效的语句。

```
import C:\mycode\dir1\dir2\mod       # Error: illegal syntax
```

增加*C:\mycode*在PYTHONPATH系统变量中或是*.path*文件中（假设它不是这个程序的主目录，这样的话就不需要这个步骤了），并且这样描述。

```
import dir1.dir2.mod
```

实际上，模块搜索路径上的项目提供了平台特定的目录路径前缀，之后再在import的那些路径左边添加了这些路径。import语句以与平台不相关的方式，提供了目录路径写法（注1）。

__init__.py包文件

如果选择使用包导入，就必须多遵循一条约束：包导入语句的路径内的每个目录内都必须有*__init__.py*这个文件，否则导入包会失败。也就是说，在我们所采用的例子中，

注1：选择点号语法，一部分是考虑到跨平台，但也是因为import语句中的路径会变成实际的嵌套对象路径。这种语法也意味着，如果你忘了在import语句中省略*.py*，就会得到奇怪的错误信息。例如，import mod.py会被看成是目录路径导入：这是要载入*mod.py*，但解释器却试着载入*mod\py.py*，而最终就是发出可能令人困惑的错误信息。

www.TopSage.com

*dir1*和*dir2*内都必须包含*__init__.py*这个文件。容器目录*dir0*不需要这类文件，因为其本身没列在import语句之中。更正式说法是，像这样的目录结构：

```
dir0\dir1\dir2\mod.py
```

以及这样形式的import语句：

```
import dir1.dir2.mod
```

必须遵循下列规则：

- *dir1*和*dir2*中必须都含有一个*__init__.py*文件。
- *dir0*是容器，不需要*__init__.py*文件；如果有的话，这个文件也会被忽略。
- *dir0*（而非*dir0\dir1*）必须列在模块搜索路径上（也就是此目录必须是主目录，或者列在PYTHONPATH之中）。

结果就是，这个例子的目录结构应该是这样（以缩进表示目录嵌套结果）。

```
dir0\                          # Container on module search path
    dir1\
        __init__.py
        dir2\
            __init__.py
            mod.py
```

*__init__.py*可以包含Python程序代码，就像普通模块文件。这类文件从某种程度上讲就像是Python的一种声明，尽管如此，也可以完全是空的。作为声明，这些文件可以防止有相同名称的目录不小心隐藏在模块搜索路径中，而之后才出现真正所需要的模块。没有这层保护，Python可能会挑选出和程序代码无关的目录，只是因为有一个同名的目录刚好出现在搜索路径上位置较前的目录内。

更通常的情况下，*__init__.py*文件扮演了包初始化的挂勾、替目录产生模块命名空间以及使用目录导入时实现from *（也就是from ... import *）行为的角色。

包的初始化

Python首次导入某个目录时，会自动执行该目录下*__init__.py*文件中的所有程序代码。因此，这类文件自然就是放置包内文件所需要初始化的代码的场所。例如，包可以使用其初始化文件，来创建所需要的数据文件、连接数据库等。一般而言，如果直接执行，*__init__.py*文件没什么用，当包首次读取时，就会自动运行。

模块命名空间的初始化

在包导入的模型中，脚本内的目录路径，在导入后会变成真实的嵌套对象路径。例如，上一个例子中，导入后，表达式dir1.dir2会运行，并返回一个模块对象，而

www.TopSage.com

此对象的命名空间包含了*dir2*的*__init__.py*文件所赋值的所有变量名。这类文件为目录（没有实际相配的模块文件）所创建的模块对象提供了命名空间。

*from *语句的行为*

作为一个高级功能，你可以在*__init__.py*文件内使用__all__列表（我们会在第21章碰到__all__）来定义目录以from *语句形式导入时，需要导出什么。在*__init__.py*文件中，__all__列表是指当包（目录）名称使用from *的时候，应该导入的子模块的名称清单。如果没有设定__all__，from *语句不会自动加载嵌套于该目录内的子模块。取而代之的是，只加载该目录的*__init__.py*文件中赋值语句定义的变量名，包括该文件中程序代码明确导入的任何子模块。例如，某目录中*__init__.py*内的语句from submodule import X，会让变量名X可在该目录的命名空间内使用。

如果你用不着这类文件，也可以让这类文件保持空白。不过，为了让目录导入完全运作，这类文件就得存在。

包导入实例

让我们实际编写刚才所谈的例子，来说明初始化文件和路径是如何运作的吧。下列三个文件分别位于目录*dir1*以及*dir1*的子目录*dir2*中。

```
# File: dir1\__init__.py
print 'dir1 init'
x = 1

# File: dir1\dir2\__init__.py
print 'dir2 init'
y = 2

# File: dir1\dir2\mod.py
print 'in mod.py'
z = 3
```

这里，*dir1*要么是我们工作所在目录（也就是主目录）的子目录，要么就是位于模块搜索路径中（技术上就是sys.path）的一个目录的子目录。无论哪一种，*dir1*的容器都不需要*__init__.py*文件。

当Python向下搜索路径时，import语句会在每个目录首次遍历时，执行该目录的初始化文件。print语句加在这里，用来跟踪它们的执行。此外，就像模块文件一样，任何已导入的目录也可以传递给reload，来强制该项目重新执行。就像这里展示的那样，reload可以接受点号路径名称，来重载嵌套的目录和文件。

www.TopSage.com

```
% python
>>> import dir1.dir2.mod       # First imports run init files
dir1 init
dir2 init
in mod.py
>>>
>>> import dir1.dir2.mod       # Later imports do not
>>>
>>> reload(dir1)
dir1 init
<module 'dir1' from 'dir1\__init__.pyc'>
>>>
>>> reload(dir1.dir2)
dir2 init
<module 'dir1.dir2' from 'dir1\dir2\__init__.pyc'>
```

导入后，import语句内的路径会变成脚本的嵌套对象路径。在这里，mod是对象，嵌套在对象dir2中，而dir2又嵌套在对象dir1中。

```
>>> dir1
<module 'dir1' from 'dir1\__init__.pyc'>
>>> dir1.dir2
<module 'dir1.dir2' from 'dir1\dir2\__init__.pyc'>
>>> dir1.dir2.mod
<module 'dir1.dir2.mod' from 'dir1\dir2\mod.pyc'>
```

实际上，路径中的每个目录名称都会变成赋值了模块对象的变量，而模块对象的命名空间则是由该目录内，*__init__.py*文件中所有赋值语句进行初始化的。dir1.x引用了变量x，x是在*dir1__init__.py*中赋值的，而mod.z易用的变量z则是在*mod.py*内赋值的。

```
>>> dir1.x
1
>>> dir1.dir2.y
2
>>> dir1.dir2.mod.z
3
```

包对应的from和import

import语句和包一起使用时，有些不方便，因为你必须经常在程序中输入路径。例如，上一节的例子中，每次要得到z时，就得从dir1开始重新输入完整路径，并且每次都要重新执行整个路径。如果你想要尝试直接读取dir2或mod，就会得到一个错误。

```
>>> dir2.mod
NameError: name 'dir2' is not defined
>>> mod.z
NameError: name 'mod' is not defined
```


www.TopSage.com

因此，让包使用from语句，来避免每次读取时都得重新输入路径，通常这样比较方便。也许更重要的是，如果你重新改变目录树结构，from语句只需在程序代码中更新一次路径，而import则需要修改很多地方。import作为一个扩展功能（下一章讨论），在这里也有一定的帮助，它提供一个完整路径较短的同义词：

```
% python
>>> from dir1.dir2 import mod       # Code path here only
dir1 init
dir2 init
in mod.py
>>> mod.z                           # Don't repeat path
3
>>> from dir1.dir2.mod import z
>>> z
3
>>> import dir1.dir2.mod as mod     # Use shorter name
>>> mod.z
3
```

为什么要使用包导入

如果刚学Python，确认已经精通了简单的模块，才能进入包的领域，因为这里有些高级的功能。然而，包也扮演了重要的角色，尤其是在较大程序中：包让导入更具信息性，并可以作为组织工具，简化模块的搜索路径，而且可以解决模糊性。

首先，因为包导入提供了程序文件的目录信息，因此可以轻松地找到文件，从而可以作为组织工具来使用。没有包导入时，通常得通过查看模块搜索路径才能找出文件。再者，如果根据功能把文件组织成子目录，包导入会让模块扮演的角色更为明显，也使代码更具可读性。例如，正常导入模块搜索路径上某个目录内的文件时，就像这样：

```
import utilities
```

与下面包含路径的导入相比，提供的信息就更少：

```
import database.client.utilities
```

包导入也可以大幅减化PYTHONPATH和*.pth*文件搜索路径设置。实际上，如果所有跨目录的导入，都使用包导入，并且让这些包导入都相对于一个共同的根目录，把所有Python程序代码都存在其中，在搜索路径上就只需一个单独的接入点：通用的根目录。最后，包导入让你想导入的文件更明确，从而解决了模糊性。下一节要深入探索包导入所扮演的角色。

www.TopSage.com

三个系统的传说

实际中需要包导入的场和，就是解决当多个同名程序文件安装在同一个机器上时，所引发的模糊性。这是与安装相关的问题，也是通常实际中所要留意的地方。让我们用一个假设的场景来说明。

假设程序员开发了一个Python程序，它包含了一个文件*utilities.py*，其中包含了通用的工具代码，还有一个顶层文件*main.py*让用户来启动程序。这个程序的文件会以`import utilities`加载并使用通用的代码。当程序分发给用户时，采用的是*.tar*或*.zip*文件的形式，其中包含了该程序的所有文件，而当它在安装时，会把所有文件解压放进目标机器上的某一个名为*system1*的目录中。

```
system1\
    utilities.py        # Common utility functions, classes
    main.py             # Launch this to start the program
    other.py            # Import utilities to load my tools
```

现在，假设有第二位程序员开发了另一个不同的程序，而其文件也命名为*utilities.py*和*main.py*，而且同样也在程序文件中使用`import utilities`来加载一般的代码文件。当获得第二个系统并安装在和第一个系统相同的计算机上时，它的文件会解压并安装至接收机器上某处文件夹名为*system2*的新目录内，使其不会覆写第一个系统的同名文件。

```
system2\
    utilities.py        # Common utilities
    main.py             # Launch this to run
    other.py            # Imports utilities
```

到目前为止，都没有什么问题：两个系统可共存，在同一台机器上运行。而实际上，不需要配置模块搜索路径，来使用计算机上的这些程序。因为Python总是先搜索主目录（也就是包含顶层文件的目录），任一个系统内的文件的导入，都会自动看见该系统目录内的所有文件。例如，如果点击*system1\main.py*，所有的导入都会先搜索*system1*。同样地，如果启动*system2\main.py*，则会改为先搜索*system2*。记住，只有在跨目录进行导入时才需要模块搜索路径的设置。

尽管如此，假设在机器上安装这两套程序之后，决定在自己的系统中使用每一个*utilities.py*文件内的一些程序代码。毕竟，这是通用工具的代码，而Python代码的本质都是想再利用的。就此而言，想在第三个目录内所编写的文件内写下了下面的代码，来载入两个文件其中的一个：

```
import utilities
utilities.func('spam')
```


www.TopSage.com

现在，问题开始出现了。为了让这能够工作，得设置模块搜索路径，引入包含*utilities.py*文件的目录。但是，要在路径内先放哪个目录呢：*system1*还是*system2*？

这个问题在于搜索路径本质上是线性的。搜索总是从左至右扫描，所以不管这个困境你想多久，一定会得到搜索路径上最左侧（最先列出）的目录内的*utilities.py*。也就是说，永远无法导入另一个目录的那个文件。每次导入操作时，可以试着在脚本内修改`sys.path`，但那是外部的工作，很容易出错。在默认情况下，可以说你走到了一个死胡同。

这个问题正是包所能够解决的。不要在单独的目录内把文件安装成单纯的文件列表，而是将它们打包，在共同根目录之下，安装成子目录。例如，可能想组织这个例子中的所有代码，变成下面这样的安装层次。

```
root\
    system1\
        __init__.py
        utilities.py
        main.py
        other.py
    system2\
        __init__.py
        utilities.py
        main.py
        other.py
    system3\                    # Here or elsewhere
        __init__.py             # Your new code here
        myfile.py
```

现在，就是把共同根目录添加到搜索路径中。如果程序代码的导入就相对于这个通用的根目录，就能以包导入，导入任何一个系统的工具文件：该文件所在目录名称会使其路径具有唯一性（因此，引用的模块也是这样）。事实上，只要使用`import`语句，就可以在同一个模块内导入这两个工具文件，而每次引用工具模块时，都要重复其完整的路径。

```
import system1.utilities
import system2.utilities
system1.utilities.function('spam')
system2.utilities.function('eggs')
```

在这里，所在目录名称让模块的引用变得具有唯一性。

注意：如果需要读取两个或两个以上路径内的同名属性时，才须要使用`import`，在这种情况下不能用`from`。如果被调用的函数名称在每个路径内都不同，`from`语句就可以避免每当调用其中一个函数时，就得重复完整的包的路径的问题，这一点先前已经说过。

www.TopSage.com

此外，注意到，在前边所展示的安装层次中，*__init__.py*文件已加入到了*system1*和*system2*目录中来使其工作，但是不需要在根目录内增加。只有在程序代码内，import语句所列的目录才需要这些文件。回想一下，Python首次通过包的目录处理导入时，这些文件就会自动运行了。

从技术上来讲，在这种情况下，*system3*目录不需要放在根目录下：只有被导入的代码包需要。然而，因为不知道何时模块可能在其他程序中有用，你可能还是想将其放在通用的根目录下，来避免以后类似的变量名冲突问题。

最后，注意两个初始系统的导入依然正常运作。因为它们的主目录都会先搜索，在搜索路径上多余的通用根目录，对于*system1*和*system2*内的代码是不相关的。它们只需写import utilities，就可以找到自己的文件。再者，如果在通用的根目录下解开所有的Python系统，路径配置就会变得很简单：只需要一次添加通用的根目录就可以了。

为什么要在意：模块包

现在，包是Python的标准组成部分，常常见到较大的第三方扩展都是以包目录的形式分发给用户，而不是单纯的模块列表。例如，*win32all*这个Python的Windows扩展包，是首先采用包这种潮流的扩展之一。其许多工具模块都位于包内，并且通过路径来导入。例如，需要加载客户端COM工具，就需要使用像这样的语句：

```
from win32com.client import constants, Dispatch
```

这一行代码会从win32com包（一个安装子目录）的client模块取出一些变量名。

包导入在Jython（Python的Java实现版本）所运行的代码中也很普遍，因为Java库也是组织为层次结构的。在最近的Python版本中，email和XML工具，也像这样组织了标准库内的子目录。无论你是否将要建立包的目录，最终还是可能从包的目录导入的。

本章小结

本章介绍了Python的包导入模型：这是可选的但确实相当有用的方式，可以明确列出目录路径从而找到模块。包导入依然与模块导入搜索路径上的一个目录有一定的关系，但是，不是依赖Python去遍历搜索路径，而是由脚本明确指出模块路径的其余部分。

正如我们所见到的，包不仅让导入在较大系统中更有意义，也可以简化了导入搜索路径

www.TopSage.com

设置（如果所有跨目录的导入都在共同根目录下的话），而且当同名的模块有一个以上时，也可解决模糊性（通过包导入所引入的模块所在目录名称来区分）。

下一章中，我们要研究一些更高级的模块的相关话题，例如，相对导入语法以及__name__模式变量的用法。不过，就像平常一样，我们要以习题结束本章，来测试你在本章所学到的内容。

www.TopSage.com

本章习题

1. 模块包目录内的*__init__.py*文件有何用途?
2. 每次引用包的内容时，如何避免重复包的完整路径?
3. 哪些目录需 要*__init__.py*文件?
4. 在什么情况下必须通过import而不能通过from使用包?

习题解答

1. *__init__.py*文件是用于声明和初始化模块包的。第一次在进程中导入某目录时，Python会自动运行这个文件中的代码。其赋值的变量会变成对应于该目录在内存中所创建的模块对象的属性。它不是选用的：如果一个目录中没有包含这个文件的话，是无法通过包语法导入目录的。
2. 通过from语句使用包，直接把包的变量名复制出来，或者使用import语句的as扩展功能，把路径改为较短的别名。在这种情况下，路径只出现在了一个地方，就在from或import语句中。
3. import或from语句中所列出的每个目录都必须含有*__init__.py*文件。其他目录则不需要包含这个文件，包括含有包路径最左侧组件的目录。
4. 只有在你需要读取定义在一个以上路径的相同变量名时，才必须通过import来使用包，而不能使用from。使用import，路径可让引用独特化，然而，from却让任何变量名只有一个版本。

www.TopSage.com

大家网
TopSage.com

第21章

高级模块话题

本章以一些更高级的模块相关话题作为第5部分的结尾：相对导入语法、数据隐藏、__future__模块、__name__变量和sys.path修改等，此外还有本书这一部分所提到的相关陷阱和练习题。就像函数一样，当接口定义良好时，模块会更加高效。所以本章简要复习模块设计的概念，而其中有些概念已在前几章中讨论过。

尽管本章标题有“高级”二字，这里所讨论的有些话题（例如，__name__技巧）都广泛得到了使用，所以，在学习本书下一部分内容——类之前，要确定学过了这些内容。

在模块中隐藏数据

正如我们所见到过的，Python模块会导出其文件顶层所赋值的所有变量名。没有对某一个变量名进行声明，使其在模块内可见或不可见这种概念。实际上，如果客户想的话，是没有防止客户端修改模块内变量名的办法的。

在Python中，模块内的数据隐藏是一种惯例，而不是一种语法约束。的确可以通过破坏模块名称使这个模块不能工作，但值得庆幸的是，我们还没遇到过这种程序员。有些纯粹主义者对Python资料隐藏采取的这种开放态度不以为然，并宣称这表明Python无法实现封装。然而，Python的封装更像是打包，而不是约束。

最小化from *的破坏：_X和__all__

有种特定的情况，可以把下划线放在变量名前面（例如，_X），可以防止客户端使用from *语句导入模块名时，把其中的那些变量名复制出去。这其实是为了把对命名空间的破坏最小化而已。因为from *会把所有变量名复制出去，导入者可能得到超出它所需的部分（包括会覆盖导入者内的变量名的变量名）。下划线不是“私有”声明：你还是可以使用其他导入形式看见并修改这类变量名。例如，使用import语句。

www.TopSage.com

此外，也可以在模块顶层把变量名的字符串列表赋值给变量__all__，以达到类似于_X命名惯例的隐藏效果。例如：

```
__all__ = ["Error", "encode", "decode"]      # Export these only
```

使用此功能时，from *语句只会把列在__all__列表中的这些变量名复制出来。事实上，这和_X惯例相反：__all__是指出要复制的变量名，而_X是指出不被复制的变量名。Python会先寻找模块内的__all__列表，如果没有定义的话，from *就会复制出开头没有单下划线的所有变量名。

就像_X惯例一样，__all__列表只对from *语句这种形式有效，它并不是私有声明。模块编写者可以使用任何一种技巧实现模块，在碰上from *时，能良好的运行（参考第20章中有关包*__init__.py*文件内__all__列表的讨论，那些列表中声明了由from *所加载的子模块）。

启用以后的语言特性

可能破坏现有代码语言方面的变动会不断引进。一开始，是以选用扩展功能的方式出现，默认是关闭的。要开启这类扩展功能，可以使用像以下形式的特定的import语句：

```
from __future__ import featurename
```

这个语句一般应该出现在模块文件的顶端（也许在docstring之后），因为这是以每个模块为基础，开启特殊的代码编译。此外，在交互模式提示符下提交这个语句也是可以的，从而能够实验今后的语言变化。于是，接下来的交互会话过程中，就可以使用这些功能了。

例如，本书的上一个版本中，我们必须使用这条语句来示范生成器函数，在默认情况下还不能使用这类函数需要的关键词（它们使用的是generators这个功能名称）。在第5章使用这条语句启用数字的真除法，本章将再次使用，来开启绝对导入的功能，以及稍后在第7部分示范环境管理器的用法。

这些变动都有可能破坏现有的程序代码，因此以这种特定的导入形式，当作是可选的功能，逐步熟悉。

混合用法模式：__name__和__main__

这是一个特殊的与模块相关的技巧，可把文件作为成模块导入，并以独立式程序的形式运行。每个模块都有个名为__name__的内置属性，Python会自动设置该属性：

www.TopSage.com

- 如果文件是以顶层程序文件执行，在启动时，__name__就会设置为字符串"__main__"。
- 如果文件被导入，__name__就会改设成客户端所了解的模块名。

结果就是模块可以检测自己的__name__，来确定它是在执行还是在导入。例如，假设我们建立下面的模块文件，名为*runme.py*，它只导出了一个名为tester的函数。

```
def tester():
    print "It's Christmas in Heaven..."

if __name__ == '__main__':          # Only when run
    tester()                        # Not when imported
```

这个模块定义了一个函数，让用户可以正常的导入并使用：

```
% python
>>> import runme
>>> runme.tester()
It's Christmas in Heaven...
```

然而，这个模块也在末尾包含了当此文件以程序执行时，就会调用该函数的代码：

```
% python runme.py
It's Christmas in Heaven...
```

也许使用__name__测试最常见的就是自我测试代码。简而言之，可以在文件末尾加个__name__测试，把测试模块导出的程序代码放在模块中。如此一来，你可以继续导入，在客户端使用该文件，而且可以通过检测其逻辑在系统shell中（或其他启动方式）运行它。实际上，在文件末端的__name__测试中的自我测试程序代码，可能是Python中最常见并且是最简单的单元测试协议（第29章讨论其他用于测试Python程序代码的常用选项，要知道，unittest和doctest标准库模块，提供更为高级的测试工具）。

编写既可以作为命令行工具也可以作为工具库使用的文件时，__name__技巧也很好用。例如，假设用Python编写了一个文件寻找脚本。如果将其打包成一些函数，而且在文件中加入__name__测试，当此文件独立执行时，就自动调用这些函数，这样就能提高代码的利用效率。如此一来，脚本的代码就可以在其他程序中再利用了。

以__name__进行单元测试

实际上，我们已经在本书的一个实例中看到了__name__检查的有用之处。第16章讨论参数那一节，我们编写了一个脚本，从一组传进来的参数中计算出其最小值。

www.TopSage.com

大家网
TopSage.com

```
def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res

def lessthan(x, y): return x < y
def grtrthan(x, y): return x > y

print minmax(lessthan, 4, 2, 1, 5, 6, 3)          # Self-test code
print minmax(grtrthan, 4, 2, 1, 5, 6, 3)
```

这个脚本在末端包含了自我测试程序代码。所以不用每次执行时，都得在交互模式命令行中重新输入所有代码就可以进行测试。然而，这种写法的问题在于，每次这个文件被另一个文件作为工具导入时，都会出现调用自我测试所得到的输出：这可不是用户友好的特性！改进之后，我们在__name__检查区块内封装了自我测试的调用，使其在文件作为顶层脚本执行时才会启动，而导入时则不会。

```
print 'I am:', __name__

def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res

def lessthan(x, y): return x < y
def grtrthan(x, y): return x > y

if __name__ == '__main__':
    print minmax(lessthan, 4, 2, 1, 5, 6, 3)      # Self-test code
    print minmax(grtrthan, 4, 2, 1, 5, 6, 3)
```

我们也在顶端打印__name__的值，目的是来跟踪它的值。Python开始加载文件时，就创建了这个用法模式的变量并对其赋值。当以顶层脚本执行这个文件的时候，它的名称就会设置为__main__，所以，它的自我测试程序代码会自动执行。

```
% python min.py
I am: __main__
1
6
```

但是，如果我们导入这个文件，其名称就不是__main__，就必须明确地调用这个函数来执行。

```
>>> import min
I am: min
```


www.TopSage.com

```
>>> min.minmax(min.lessthan, 's', 'p', 'a', 'm')
'a'
```

同样地，无论这是否用于测试，结果都是让代码有两种不同的角色：作为工具的库模块，或者是作为可执行的程序。

修改模块搜索路径

在第18章中已经介绍过，模块搜索路径是一个目录列表，可以通过环境变量PYTHONPATH以及可能的*.pth*路径文件进行定制。还没有介绍的是，实际上Python程序本身是如何修改搜索路径的，也就是修改名为`sys.path`（内置模块`sys`的`path`属性）的内置列表。`sys.path`在程序启动时就会进行初始化，但在那之后，可以随意对其元素进行删除、附加和重设。

```
>>> import sys
>>> sys.path
['', 'D:\\PP3ECD-Partial\\Examples', 'C:\\Python25', ...more deleted...]

>>> sys.path.append('C:\\sourcedir')          # Extend module search path
>>> import string                             # All imports search the new dir
```

一旦做了这类修改，就会对Python程序中将要导入的地方产生影响，因为所有导入和文件都共享了同一个`sys.path`列表。事实上，这个列表可以任意修改。

```
>>> sys.path = [r'd:\temp']                   # Change module search path
>>> sys.path.append('c:\\lp3e\\examples')     # For this process only
>>> sys.path
['d:\\temp', 'c:\\lp3e\\examples']

>>> import string
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ImportError: No module named string
```

因此，可以使用这个技巧，在Python程序中动态配置搜索路径。不过，要小心：如果从路径中删除重要目录，就无法获取一些关键的工具了。例如，上一个例子中，我们从路径中删除Python的源代码库目录的话，我们就再也无法获取`string`模块。

此外，记住`sys.path`的设置方法只在修改的Python会话或程序（即进程）中才会存续。在Python结束后，不会保留下来。PYTHONPATH和*.pth*文件路径配置是保存在操作系统中，而不是执行中的Python程序。因此使用这种配置方法更全局一些：机器上的每个程序都会去查找PATHONPATH和*.pth*，而且在程序结束后，它们还存在着。

www.TopSage.com

import as扩展

import和from语句都可以扩展，让模块可以在脚本中给予不同的变量名。下面的import语句：

```
import longmodulename as name
```

相当于：

```
import longmodulename
name = longmodulename
del longmodulename                          # Don't keep original name
```

在这类import之后，可以（事实上是必须）使用列在as之后的变量名来引用该模块。from语句也可以这么用，把从某个文件导入的变量密码那个，赋值给脚本中的不同的变量名：

```
from module import longname as name
```

这个扩展功能很常用，替变量名较长的变量提供简短一些的同义词，而且当已在脚本中使用一个变量名使得执行普通import语句会被覆盖时，使用as，就可避免变量名冲突。此外，使用第20章所提到的包导入功能时，也可替整个目录路径提供简短、简单的名称，十分方便。

相对导入语法

Python 2.5修改一些from语句，用在上一章所提到的模块包时的导入搜索路径语义上。这个变动的某些影响在稍后的Python版本中才会显现（目前预计在2.7版和3.0版出现），不过，有些如今已呈现出来了。

简而言之，from语句现在可以使用点号（“.”），更倾向于定义位于同一个包内的模块（称为包相对导入），而不是位于模块导入搜索路径上其他地方的模块（所谓的绝对导入）。

- 现在，可以使用点号指出该导入应该与其所在的包想关联：这类导入倾向于导入位于该包内的模块，而不是导入搜索路径sys.path上其他地方的同名模块。
- 在软件包内的代码的正常导入（没有点号）目前默认的搜索路径次序是：“相对”之后“绝对”。然而以后Python将会以绝对导入为默认情况：缺少任何特殊点号语法时，导入会跳过所在的包，而去sys.path搜索路径上寻找。

www.TopSage.com

例如，目前的一条form语句：

```
from .spam import name
```

的意思是“从这条语句所在同一个包内名为spam的模块，导入name这个变量。”没有开头点号的类似语句，依然默认为是当前的“相对”之后“绝对”的路径搜索顺序，除非在进行导入的文件中包含了下列形式的语句：

```
from __future__ import  absolute_import    # Required until 2.7?
```

如果存在，这条语句会改变为启用以后的默认绝对路径。这会使得没有附加点号的所有导入都跳过模块导入搜索路径的相对元素，并改为首先寻找sys.path所含的绝对目录。例如，当绝对导入开启时，下列形式的语句总是会去找标准库的string模块，而不是包中同名的模块：

```
import string                              # Always finds standard lib's version
```

没有from __future__语句时，如果包中有个字符串模块，就会改为导入该模块。当未来绝对导入变化启用后，要达到相同的行为的效果，就需要执行下面形式的语句（当前的Python也适用），来进行强制性的相对导入。

```
from . import string                       # Searches this package first
```

注意：开头的点号能用在from语句却不能用在import语句中。import modname语句的形式依然进行相对导入，但是Python 2.7版时会变成绝对导入。

还有其他的基于点号相对引用的样式可以采用。已知包的名称为mypkg，下列由包内代码所采用的替代的导入形式，也会像注释所说的那样运行：

```
from .string import name1, name2          # Imports names from mypkg.string
from . import string                      # Imports mypkg.string
from .. import string                     # Imports string from parent directory
```

要进一步了解这些形式，需要了解这种变化内含的原理。

为什么使用相对导入

这个功能的设计初衷是，让当脚本在同名文件出现在模块搜索路径上许多地方时，可以解决模糊性。考虑下面的包的目录：

```
mypkg\
    __init__.py
    main.py
    string.py
```


www.TopSage.com

这定义了一个名为mypkg的包，其中含有名为mypkg.main和mypkg.string的模块。现在，假设模块main试图导入名为string的模块。在Python 2.4和稍早版本中，Python会先寻找*mypkg*目录以执行相对导入。这会找到并导入位于该处的*string.py*文件，将其赋值给mypkg.main模块命名空间内的变量名string。

不过，import的原意可能是要导入Python标准库的string模块。可惜的是，在这些Python的版本中，没有办法直接忽略mypkg.string而去寻找位于模块搜索路径更右侧的标准库中的string模块。我们无法依赖在每台机器上都存在标准链接库以上的额外包的目录结构。

换句话说，包中的导入可能是模糊的：在包内，import spam语句是指包内或包外的模块，这并不明确。更准确地讲，一个局部的模块或包可能会隐藏sys.path上的另一个模块，无论是有意或无意的。

实际上，Python用户可以避免为他们自己的模块重复使用标准库模块的名称（如果需要标准string，就不要把新的模块命名为string）。但是，这样对包是否会不小心隐藏标准模块没有什么帮助。再者，Python以后可能新增标准库模块，而其名称刚好和自己的一个模块同名。依赖相对导入的程序代码比较不容易理解，因为读者对于期望使用哪个模块，可能会感到困惑。如果程序代码中能明确地进行分辨，就比较好了。

在Python 2.5中，可以使用先前提到的__future__导入命令，强迫导入的绝对性，来控制导入的行为。同样请记住这种绝对导入行为在未来版本中会变成默认（目前计划在Python 2.7中）。当绝对导入开启时，范例文件*mypkg/main.py*中下列形式的语句，总是会通过绝对导入搜索，而去寻找标准库版本的string：

```
import string                    # Imports standard lib string
```

现在应该习惯使用绝对导入，当变化生效时，才有心理准备。也就是说，如果真的想从包中导入模块，为了使导入变得明确和绝对，应该开始在程序代码中编写这样的语句（mypkg会在sys.path上的绝对目录内找到）：

```
from mypkg import string         # Imports mypkg.string (absolute)
```

在from语句中使用点号样式，依然可以使用相对导入：

```
from . import string             # Imports mypkg.string (relative)
```

这种形式会导入相对于当前包的string模块，而且是上一个例子中绝对形式的相对等效的版本（包的目录会首先搜索到）。

www.TopSage.com

我们也可以使用相对语法从模块中复制特定变量名：

```
from .string import name1, name2        # Imports names from mypkg.string
```

这条语句仍然引用相对于当前包的string模块。例如，如果此程序代码出现在mypkg.main模块内，就会从mypkg.string导入name1和name2。多加的开头点号会从当前包的上层目录开始执行相对导入。例如：

```
from .. import spam                     # Imports a sibling of mypkg
```

会加载mypkg的兄弟模块，也就是说，spam位于上层目录，和mypkg相邻。更通畅的情况是，位于某个模块A.B.C之内的程序代码可以做下列任何之事：

```
from . import D                         # Imports A.B.D
from .. import E                        # Imports A.E
from ..F import G                       # Imports A.F.G
```

相对导入语法和提到的默认绝对导入这个改变都是高级的概念，而且这些功能在Python 2.4中只存在了一部分而已。因此，我们在这里省略以后的细节。请参考Python标准手册以了解更多信息。

模块设计理念

就像函数一样，模块也有设计方面的折衷考虑：需要思考哪些函数要放进模块、模块通讯机制等。当开始编写较大的Python系统时，这些就会变得明朗起来。但是要记住以下是一些通用的概念。

- **总是在Python的模块内编写代码。**没有办法写出不在某个模块之内的程序代码。事实上，在交互模式提示符下输入的程序代码，其实是存在于内置模块__main__之内。交互模式提示符独特之处就在于程序是执行后就立刻丢弃，以及表达式结果是自动打印的。
- **模块藕合要降到最低：全局变量。**就像函数一样，如果编写成闭合的盒子，模块运行得最好。原则就是，模块应该尽可能和其他模块的全局变量无关。
- **最大化模块的粘合性：统一目标。**可以通过最大化模块的粘合性来最小化模块的藕合性。如果模块的所有元素都享有共同的目的，就不太可能依赖外部的变量名。
- **模块应该少去修改其他模块的变量。**我们在第16章中以代码做过说明，但值得在这里重复：使用另一个模块定义的全局变量，这完全是可以的（毕竟这就是客户端导入服务的方式），但是，修改另一个模块内的全局变量，通常是出现设计问题的征

www.TopSage.com

兆。当然，也有些例外，但是应该试着通过函数参数返回值这类机制去传递结果，而不是进行跨模块的修改。否则，全局变量的值会变成依赖于其他文件内的任意远程赋值语句的顺序，而模块会变得难以理解和再利用。

TopSage.com

总之，图21-1描绘了模块操作的环境。模块包含变量、函数、类以及其他的模块（如果导入了的话）。函数有自己的本地变量。在第22章会介绍类（模块中的另一种对象）。

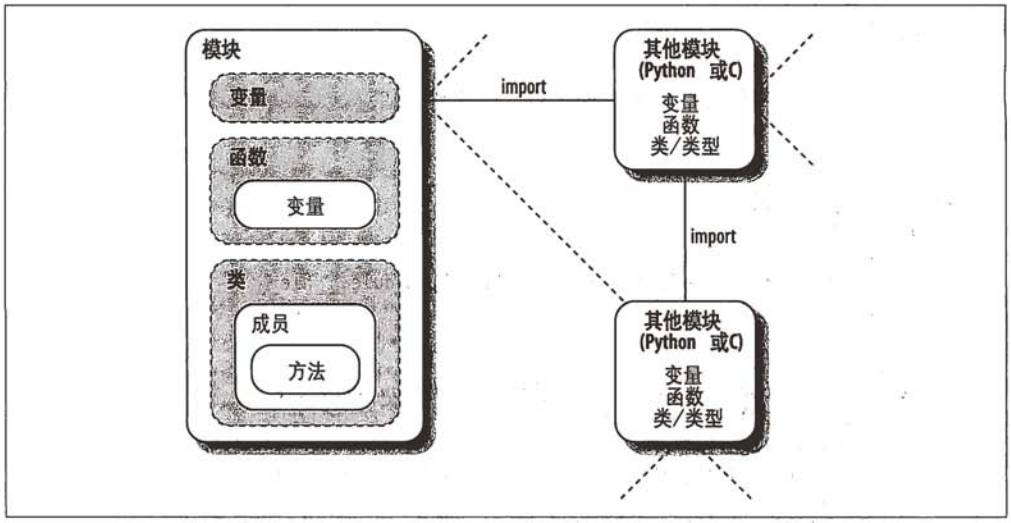


图21-1：模块的执行环境。模块是被导入的，但模块也会导入和使用其他模块，这些模块可以用Python或其他语言（例如，C语言）写成。模块可内含变量、函数以及类来进行其工作，而函数和类可包含变量和其他元素。不过，从最顶端来看，程序也只是一个模块的集合而已

模块是对象：元程序

因为模块通过内置属性显示了它们的大多数有趣的特性，因此，可很容易的编写程序来管理其他程序。我们通常称这类管理程序为元程序（metaprogram），因为他们是在其他系统之上工作。这也称为内省（introspection），因为程序能看见和处理对象的内部。内省是高级功能，但是，它可以用做创建程序的工具。

例如，要取得M模块内名为name的属性，可以使用结合点号运算，或者对模块的属性字典进行索引运算（在内置__dict__属性中显示）。Python也在sys.modules字典中导出所有已加载的模块的列表（也就是sys模块的modules属性），并提供一个内置函数getattr，让我们以字符串名来取出属性（就好像是object.attr，而attr是运行时的字符串）。因此，下列所有表达式都会得到相同的属性和对象。

www.TopSage.com

```
M.name                          # Qualify object
M.__dict__['name']              # Index namespace dictionary manually
sys.modules['M'].name           # Index loaded-modules table manually
getattr(M, 'name')              # Call built-in fetch function
```

通过像这样揭示了模块的内部机制，Python可帮助你建立关于程序的程序（注1）。例如，以下是名为*mydir.py*的模块，运用这些概念，可以实现定制版本的内置函数dir。它定义并导出了一个名为listing的函数，这个函数以模块对象为参数，打印该模块命名空间的格式化列表。

```
# A module that lists the namespaces of other modules

verbose = 1

def listing(module):
    if verbose:
        print "-"*30
        print "name:", module.__name__, "file:", module.__file__
        print "-"*30

    count = 0
    for attr in module.__dict__.keys():        # Scan namespace
        print "%02d) %s" % (count, attr),
        if attr[0:2] == "__":
            print "<built-in name>"           # Skip __file__, etc.
        else:
            print getattr(module, attr)       # Same as __dict__[attr]
        count = count+1

    if verbose:
        print "-"*30
        print module.__name__, "has %d names" % count
        print "-"*30

if __name__ == "__main__":
    import mydir
    listing(mydir)                            # Self-test code: list myself
```

我们在模块末端提供自我测试逻辑，导入自身并列举出自身的内容。以下是得到的输出结果。

注1：如第16章所看到的，因为函数可以像这里一样通过sys.modules表来获得它所在模块，所以，模拟global语句的效果是有可能的。例如，global X。X=0的效果可在函数内这样写以进行模拟（只不过要输入比较多的字）：import sys; glob=sys.modules[__name__]; glob.X=0。记住，每个模块都可取得__name__属性，在模块内的函数中，这是可见的全局变量。这个技巧还有另一种方式，可以修改函数内同名的局部变量和全局变量。

www.TopSage.com

```
C:\python> python mydir.py
------------------------------
name: mydir file: mydir.py
------------------------------
00) __file__ <built-in name>
01) __name__ <built-in name>
02) listing <function listing at 885450>
03) __doc__ <built-in name>
04) __builtins__ <built-in name>
05) verbose 1
------------------------------
mydir has 6 names
------------------------------
```

稍后我们会再遇见getattr以及与它作用相似的操作。重点就在于mydir是一个可以浏览其他程序的程序。因为Python能够展现其内部，通常可以像这样处理各种对象（注2）。

模块陷阱

本节中，我们要看一看会让Python初学者生活多点乐趣的常见的极端案例。有些很罕见，很难举例说明，但大多数都示范了语言中重要的部分。

顶层代码的语句次序的重要性

当模块首次导入（或重载）时，Python会从头到尾执行语句。这里有些和前向引用（forward reference）相关的涵义，值得在此强调：

- 在导入时，模块文件顶层的程序代码（不在函数内）一旦Python运行到时，就会立刻执行。因此，该语句是无法引用文件后面位置赋值的变量名。
- 位于函数主体内的代码直到函数被调用后才会运行。因为函数内的变量名在函数实际执行前都不会解析，通常可以引用文件内任意地方的变量。

一般来说，前向引用只对立即执行的顶层模块代码有影响，函数可以任意引用变量名。以下是示范前向引用的例子。

```
func1()                          # Error: "func1" not yet assigned
```

注2：像mydir.listing这类工具可以由PYTHONSTARTIP环境变量所引用的文件进行导入，预先在交互模式的命名空间中加载。因为在启动文件内的程序代码会在交互模式命名空间内（模块__main__）执行，在启动文件内导入常用工具，可以节省一些输入。参考附录A以获得更多细节。

www.TopSage.com

大家网
TopSage.com

```
def func1():
    print func2()                    # OK: "func2" looked up later

func1()                              # Error: "func2" not yet assigned

def func2():
    return "Hello"

func1()                              # Okay: "func1" and "func2" assigned
```

当这个文件导入时（或者作为独立程序运行时），Python会从头到尾运行它的语句。对func1的首次调用失败，因为func1 def尚未执行。只要func1调用时，func2的def已运行过，在func1内对func2的调用就没有问题 （当第二个顶层func1调用执行时fun2的def还没有运行）。文件末尾最后对func1的调用可以工作，因为func1和func2都已经赋值了。

在顶层程序内混用def不仅难读，也造成了对语句顺序的依赖性。作为一条原则，如果需要把立即执行的代码和def一起混用，就要把def放在文件前面，把顶层代码放在后面。这样的话，你的函数在使用的代码运行的时候，可以保证它们都已定义并赋值过了。

通过变量名字符串导入模块

import或from语句内的模块名是“硬编码”的变量名。不过，尽管这样，程序会在运行期间获得要导入的模块名的字符串（例如，如果用户从GUI中选择一个模块名）。可惜，你无法使用import语句直接以字符串的形式载入模块：Python期望得到的是变量名，而不是字符串。如下所示。

```
>>> import "string"
  File "<stdin>", line 1
    import "string"
                  ^
SyntaxError: invalid syntax
```

把字符串赋值给变量名也不行：

```
x = "string"
import x
```

在这里，Python会试着导入文件*x.py*，而不是string模块。

为了避免发生这样的问题，得使用特定的工具，才能够通过运行时产生的字符串动态加载模块。最通常的做法就是把import语句构造成Python代码的字符串，再传给exec语句执行：

www.TopSage.com

```
>>> modname = "string"
>>> exec "import " + modname      # Run a string of code
>>> string                        # Imported in this namespace
<module 'string'>
```

exec语句（以及与它作用相近的eval表达式函数）会编译一段字符串代码，将其传给Python解释器来执行。在Python中，字节码编译器可在运行时使用。所以可以像这里一样编写程序，构造并执行其他程序。在默认情况下，exec会在当前作用域内运行代码，但是，可以传入可选的命名空间字典来进行定义。

exec唯一的缺点就是，每次执行时都必须编译import语句。要执行多次时，如果能使用内置的__import__函数，改为从变量名字符串加载，代码可能会更快一些。其效果类似，但__import__会返回模块对象，所以在这里可以将其赋值给变量名，从而将其保留下来。

```
>>> modname = "string"
>>> string = __import__(modname)
>>> string
<module 'string'>
```

from复制变量名，而不是连接

尽管常用，但from语句也是Python中各种潜在陷阱的源头。from语句其实是在导入者的作用域内对变量名的赋值语句，也就是变量名拷贝运算，而不是变量名的别名机制。它的实现和Python所有赋值运算都一样，但是其微妙之处在于，共享对象的代码存在于不同的文件中。例如，假设我们定义了下列模块（*nested1.py*）。

```
X = 99
def printer(): print X
```

如果我们在另一个模块内（*nested2.py*）使用from导入两个变量名，就会得到两个变量名的拷贝，而不是对两个变量名的连接。在导入者内修改变量名，只会重设该变量名在本地作用域版本的绑定值，而不是*nested1.py*中的变量名：

```
from nested1 import X, printer      # Copy names out
X = 88                              # Changes my "X" only!
printer()                           # nested1's X is still 99

% python nested2.py
99
```

然而，如果我们使用import获得了整个模块，然后赋值某个点号运算的变量名，就会修改*nested1.py*中的变量名。点号运算把Python定向到了模块对象内的变量名，而不是导入者的变量名（*nested3.py*）。

www.TopSage.com

```
import nested1                    # Get module as a whole
nested1.X = 88                    # OK: change nested1's X
nested1.printer()

% python nested3.py
88
```

from *会让变量语义模糊

在第19章提到过这些内容，但把细节留在这里描述。使用from module import *语句形式时，因为你不会列出想要的变量，可能会意外覆盖了作用域内已使用了的变量名。更糟的是，这将很难确认变量来自何处。如果有一个以上的被导入文件使用了from *形式，就更是如此了。

例如，如果在三个模块上使用from *，没有办法知道简单的函数调用真正含义，除非去搜索这三个外部的模块文件（三个可能都在其他目录内）。

```
>>> from module1 import *         # Bad: may overwrite my names silently
>>> from module2 import *         # Worse: no way to tell what we get!
>>> from module3 import *
>>> . . .

>>> func()                        # Huh???
```

解决办法就是不要这么做：试着在from语句中明确列出想要的属性，而且限制在每个文件中最多只有一个被导入的模块使用from *这种形式。如此一来，任何未定义的变量名一定可以减少到某一个from *所代表的模块。如果你总是使用import而不是from，就可完全避开这个问题，但这样的建议过于苛刻。就像其他大多数程序设计中，如果合理使用的话，from也是一种很方便的工具。

reload不会影响from导入

这是另一个和from相关的陷阱：正如前边讨论过的那样，因为from在执行时会复制（赋值）变量名，所以不会连结到变量名的那个模块。通过from导入的变量名就简单的变成了对象的引用，当from运行时这个对象恰巧在被导入者内由相同的变量名引用。

正是由于这种行为，重载被导入者对于使用from导入模块变量名的客户端没有影响。也就是说，客户端的变量名依然引用了通过from取出的原始对象，即使之后原始模块中的变量名进行了重新设置：

```
from module import X              # X may not reflect any module reloads!
 . . .
reload(module)                    # Changes module, but not my names
X                                 # Still references old object
```


www.TopSage.com

为了保证重载更有效，可以使用import以及点号运算，来取代from。因为点号运算总是会回到模块，这样就会找到模块重载后变量名的新的绑定值。

```
import module                    # Get module, not names
 . . .
reload(module)                   # Changes module in-place
module.X                         # Get current X: reflects module reloads
```

reload、from以及交互模式测试

第3章曾经提到过，通常情况下，最好不要通过导入或重载来启动程序，因为其中牵涉到了许多复杂的问题。当引入from之后，事情就变得更糟了。Python初学者常常会遇到这里所提到的陷阱。在文本编辑窗口开启一个模块文件后，假设你启动一个交互模式会话，通过from加载并测试模块：

```
from module import function
function(1, 2, 3)
```

发现了一个bug，跳回编辑窗口，做了修改，并试着重载模块：

```
reload(module)
```

但是这样行不通：from语句赋值的是变量名function，而不是module。要在reload中引用模块，得先通过import至少将其加载一次：

```
import module
reload(module)
function(1, 2, 3)
```

然而，这样也无法运行：reload更新了模块对象，但是就像上一节的讨论，像function这样的变量名之前从模块复制出来，依然引用了旧的对象（在这个例子中，就是function的原始版本）。要确实获得新的function，必须在reload之后调用module.function，或者重新执行from：

```
import module
reload(module)
from module import function
function(1, 2, 3)
```

现在，新版本的function终于可以执行了。

正如见到的那样，使用reload和from有些本质上的问题：不但得记住导入后要重载，还得记住在重载后重新执行from语句。即使是专家，其复杂度也让人够头疼。

www.TopSage.com

不应该对reload和from能完美的合作抱有幻想。最佳的原则就是不要将它们结合起来使用：使用reload和import，或者以其他方式启动程序，如第3章的建议（例如，使用IDLE中的“Run”/“Run Module”菜单选项、点击文件图标或者系统命令行）。

reload的使用没有传递性

当重载一个模块时，Python只会重载那个模块的文件，不会自动重载该文件重载时碰巧还要导入的模块。例如，如果重载某个模块A，而A导入模块B和C，则重载只适用于A，不包括B和C。A中导入B和C的语句在重载时会重新执行，但只获得了已加载的B和C模块对象（假设先前已经导入过了）。以下是*A.py*文件的实际的代码：

```
import B                    # Not reloaded when A is
import C                    # Just an import of an already loaded module

% python
>>> . . .
>>> reload(A)
```

不要依赖模块重载的传递性。相反的，应该使用多个reload调用来更新子组件。依照需要，可以把系统设计成自动重载子组件，也就是在像A这样的上层模块中添加reload调用。

更好的一点就是，你可以编写一个通用工具，扫描模块的__dict__属性，自动进行传递性的重载（参考本章之前“模块是对象：元程序”一节），然后检查每个元素的类型（参考第9章），找出嵌套的模块，从而进行递归地重载。这类工具函数可递归调用自身，从而可以任意访问导入依存关系链上的模块来进行重载。

例如，下面的模块*reloadall.py*有个reload_all函数可自动重载一个模块和每个该模块导入的模块等等，一直到每个导入链的末尾。它使用了一个字典来跟踪已重载的模块、递归遍历的导入链以及标准链接库的types模块（第9章末尾介绍过），其中只是为内置类型预定义的类型结果而已。

要使用这个工具时，导入其reload_all函数，将已加载的模块名称传进去（就像内置reload函数那样）。当文件独立执行时，其自我测试代码会自我测试：必须自行导入，因为不导入有些变量名不会在文件中定义。建议亲自用这个例子进行研究和实验。

```
import types
def status(module):
    print 'reloading', module.__name__

def transitive_reload(module, visited):
    if not visited.has_key(module):                # Trap cycles, dups
```


www.TopSage.com

```
        status(module)                              # Reload this module
        reload(module)                              # And visit children
        visited[module] = None
        for attrobj in module.__dict__.values():    # For all attrs
            if type(attrobj) == types.ModuleType:   # Recur if module
                transitive_reload(attrobj, visited)

def reload_all(*args):
    visited = {  }
    for arg in args:
        if type(arg) == types.Module Type:
            transitive_reload(arg, visited)

if __name__ == '__main__':
    import reloadall                    # Test code: reload myself
    reload_all(reloadall)               # Should reload this, types[24]
```

递归形式的from import无法工作

把最诡异（值得庆幸的事，并且也是最罕见）的陷阱留到最后。因为导入会从头到尾执行一个文件的语句，使用相互导入的模块时，需要十分小心（称为递归导入）。因为一个模块内的语句在其导入另一个模块时不会全都执行，有些变量名可能还不存在。

如果使用import取出整个模块，这也许重要，也许不重要。模块的变量名在稍后使用点号运算，在获得值之前都不会读取。但是，如果使用from来取出特定的变量名，必须记住，只能读取在模块中已经赋值的变量名。

例如，考虑下列模块recur1和recur2。recur1给变量名X赋了值，然后在赋值变量名Y之前导入recur2。这时，recur2可以用import把recur1整个取出（recur1已经存在于Python的内部的模块表中了），但是，如果使用from，就只能看见变量名X。变量名Y是在导入recur1后赋值的，现在不存在，所以会产生错误。

```
# File: recur1.py
X = 1
import recur2                       # Run recur2 now if it doesn't exist
Y = 2


# File: recur2.py
from recur1 import X                # OK: "X" already assigned
from recur1 import Y                # Error: "Y" not yet assigned

>>> import recur1
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "recur1.py", line 2, in ?
    import recur2
  File "recur2.py", line 2, in ?
```


www.TopSage.com

```
    from recur1 import Y                    # Error: "Y" not yet assigned
ImportError: cannot import name Y
```

大家网
TopSage.com

当recur1的语句由recur2递归导入时，Python会避免重新执行（否则导入会让脚本变成死循环），但是，被recur2导入时，recur1的命名空间还不完整。

解决办法就是，不要在递归导入中使用from（真的，不要）。如果这么做，Python不会卡在死循环中，但是，程序又会依赖于模块中语句的顺序。

有两种方式可避开这个陷阱

- 小心设计，通常可以避免这种导入循环：最大化模块的聚合性，同时最小化模块间的藕合性，是一个很好的开始。
- 如果无法完全断开循环，就要使用import和点号运算（而不是from），将模块变量名的读取放在后边，要么就是在函数中；或者在文件末尾附近去执行from（而不是在模块顶层），以延迟其执行。

本章小结

本章研究了一些模块相关的高级概念。我们研究了数据隐藏的技巧、通过__future__模块启用新的语言特性、__name__使用模式变量和包相对导入语法等。我们也探索和总结模块设计的话题，并见到模块相关的常见错误，从而在代码中避免发生类似的错误。

下一章要开始讨论Python的面向对象程序设计工具：类。前几章我们所涉及的内容多数也都适用于类：类存在于模块内，命名空间也是。但是类对属性查找多加了一个额外的组件，称为“继承搜索”。然而，因为这是本书此部分最后一章，在深入下一个主题之前，确认已经做过这一部分的练习题。我们先做本章习题来复习一下这里所讨论的话题。

www.TopSage.com

本章习题

1. 模块顶层以下划线开头的变量名的重要性是什么？
2. 当模块的__name__变量是字符串"__main__"时，代表了什么意义？
3. from mypkg import spam和from . import spam有什么差别？
4. 如果用户通过交互模式输入模块的变量名进行测试，你该怎样进行导入？
5. 改变sys.path和设置PYTHONPATH来修改模块搜索路径有什么不同？
6. 如果模块__future__可让我们导入未来，那我们也能导入过去吗？

习题解答

1. 模块顶层变量名以单个下划线开头时，当使用from *语句形式导入，这些变量名不会被复制到进行导入的作用域中。不过，这类变量名还是可通过import或者普通的from语句形式来导入。
2. 如果模块的__name__变量是字符串"__main__"，代表了该文件是作为顶层脚本运行的，而不是被程序中另一个文件所导入的。也就是说，这个文件作为程序在使用，而不是一个库。
3. from mypkg import spam是绝对导入：mypkg位于sys.path中的一个绝对目录中。另一方面，from . import spam是相对导入：spam的查找是相对于该语句所在的包，然后才会去搜索sys.path。
4. 用户输入脚本时通常作为字符串。要通过字符串名导入所引用的模块，你可以创建import语句并通过exec执行，或把字符串名传给__import__函数进行调用。
5. 修改sys.path只会影响一个正在运行的程序，是暂时的，当程序结束时，修改就会消失。PYTHONPATH设置是存在于操作系统中的，机器上所有程序都会使用，而且对这些设置的修改在程序离开后还会保存。
6. 不行，我们无法在Python中导入过去。我们可以安装（或顽固地使用）这门语言的旧版本，但是，最新的Python往往是最好的Python。

www.TopSage.com

第五部分练习题

参考附录B的“第五部分 模块”的解答。

1. **导入基础。**编写一个程序计算文件中行数和字符数（类似于UNIX的wc）。利用文本编辑器，编写一个名为*mymod.py*的Python模块，导出三个顶层变量名。

   - countLines(name)函数：读取输入文件，计算其中的行数（提示：file.readlines可为你做大多数工作，而剩下的事可以交给len来做）。
   - countChars(name)函数：读取输入文件，计算其中的字符数（提示：file.read返回单个字符的字符串）。
   - test(name)函数：调用两个计算的函数并给出输入文件名。这类文件名一般是通过传递进来、直接写出来、通过raw_input输入或者通过sys.argv列表从命令行获得。就目前而言，假设这是传递进来的函数参数。

   这三个mymod函数预期应该有个文件名字符串会传入。如果每个函数输入的代码多于两三行，那就太费劲了：使用本书提示！

   接着，在交互模式下测试模块，使用导入和变量名点号运算来读取导出的变量名。PYTHONPATH需要引入创建的*mymod.py*所在的目录吗？试着让模块对自身运行。例如，test("mymod.py")。注意：这个测试打开了文件两次；如果你志向远大，可以通过传入一个已开启的文件对象给那两个计算函数来改进它［提示：file.seek(0)是文件回滚］。

2. from/from *。使用from直接加载导出变量名，通过交互模式测试练习题1的mymod模块，先通过变量名导入，然后使用from *来获取所有。

3. __main__。在mymod模块中加入一行，只有在模块通过脚本运行时（而不是在其被导入时），才自动调用test函数。你加入的行可能会测试__name__的值是否为字符串"__main__"，就像这一章所演示过的一样。试着从系统命令行运行模块。然后导入该模块，以交互模式测试其函数。两种模式都能用吗？

4. **嵌套导入。**写第二个模块*myclient.py*导入mymod，并测试其函数。然后，从系统命令行执行myclient。如果myclient使用from取出mymod，mymod的函数可以从myclient的顶层存取吗？如果改用import导入是什么情况呢？试着在myclient中编写这两种形式，并导入myclient，在交互模式下查看其__dict__属性。

5. **包导入。**从包导入文件。在模块导入搜索路径上的一个目录中创建名为mypkg的子

www.TopSage.com

大家网
TopSage.com

目录，把练习题1或3所创建的*mymod.py*模块文件移到这个新目录下，然后试着以`import mypkg.mymod`形式导入。

你需要在模块移入的目录中添加*__init__.py*文件才行，但是在所有主要Python平台上应该都能工作（这是Python使用“.”作为路径分隔字符的一部分原因）。创建的包目录可以是正在运行的目录底下的子目录。如果是这样，就可通过搜索路径的主目录元素找到，而不用去配置路径。加一些代码到*__init__.py*，并观察这些代码在每次导入时是不是都会运行。

6. **重载。**用模块实验重载：运行第19章*changer.py*例子的测试，重复修改被调用的函数的消息和行为，而不停止Python解释器。这取决于你的系统，你可能可以在另一个窗口编辑changer，或者暂停Python解释器，在相同的窗口中编辑（在UNIX上，Ctrl+Z通常会挂起当前的进程，而fg命令可使其重新恢复）。

7. **循环导入**（注3）。在递归导入陷阱那一节中，导入`recur1`会引发错误。但是，如果重启Python，通过交互模式导入`recur2`，则不会发生错误，自行测试并查看结果。为什么导入recur2正常工作，而recur1则行不通？［提示：Python在运行新模块的代码前，会先将新模块保存在内置的`sys.modules`表内（一个字典）；稍后的导入会先从这个表中取出该模块，无论该模块是否“完整”。］现在，试着以顶层脚本执行recur1：**`python recur1.py`**。你是否得到和交互模式导入recur1时相同的错误？为什么？（提示：当模块以程序执行时，不是被导入，所以这种情况下和通过交互模式导入`recur2`是一样的效果；`recur2`是最先导入的模块）当你把`recur2`当成脚本运行时，发生了什么？

注3：注意：循环导入在实际中很罕见。事实上，在作者十年Python代码编写的经验中，从未编写过或遇到过循环导入。另一方面，如果你了解为什么这是潜在的问题，那么你已经很好地掌握了Python的导入含义了。

www.TopSage.com

www.TopSage.com

第六部分

类和OOP

www.TopSage.com

大家网
TopSage.com

www.TopSage.com

第22章

OOP：宏伟蓝图

到目前为止，本书经常使用“对象”这个术语。其实，编写代码达到现在这个水平，都是以对象为基础。我们在脚本中传递对象、用在表达式中和调用对象的方法等。不过，要让代码真正归类于面向对象（OO），我们对象一般也需要参与到所谓的继承层次中来。

本章要开始我们对Python类的探索：类是在Python实现支持继承的新种类的对象的部件。类是Python面向对象程序设计（OOP）的主要工具，而本书这一部分内容中，我们将会一直讨论OOP的基础内容。OOP提供了一种不同寻常而往往更有效的检查程序的方式，利用这种设计方法，我们分解代码，把代码的冗余度降至最低，并且通过定制现有的代码来编写新的程序，而不是在原有代码中的适当之处进行修改。

在Python中，类的建立使用了一条新的语句：`class`语句。正如你将看到的那样，通过`class`定义的对象，看起来很像本书之前研究过的内置类型。事实上，类其实是只运用并扩展了我们谈到过的一些想法。概括地讲，类就是一些函数的包，这些函数大量使用并处理内置对象类型。不过，类的设计是为了创建和管理新的对象，并且它们也支持继承。这是一种代码定制和复用的机制，到现在为止，我们还没有见过。

还有件事要注意：在Python中，OOP完全是可选的，并且在初学阶段不需要使用类。实际上，可以用较简单的结构，例如函数，甚至简单顶层脚本代码，这样就可以做很多事。因为妥善使用类需要一些预先的规划。因此和那些采用战术模式工作的人相比（时间有限），采用战略模式工作的人（做长期产品开发）对类会更感兴趣一些。

然而，在本书这一部分你会知道，类是Python所提供最有用的工具之一。合理使用时，类实际上可以大量减少开发的时间。类也在流行的Python工具中使用，例如，Tkinter GUI API。所以大多数Python程序员往往都会发现，至少有些类基础知识是很有帮助的。

www.TopSage.com

为何使用类

还记得本文曾介绍过，程序就是“用一些东西来做事”吗？简而言之，类就是一种定义新种类的东西的方式，它反映了在程序领域中的真实对象。例如，假设要实现第15章作为例子的虚拟的制作披萨的机器人。如果通过类来实现，就可以建立其实际结构和关系的模型。从两个方面来讲OOP都证明很有用处。

继承

制作披萨的机器人就是某种机器人，它拥有一般机器人属性。从OOP术语来看，制作披萨的机器人继承了所有机器人的通用类型的属性。这些通用的属性只需要在通用的情况下实现一次，就能让未来我们所创建的所有种类的机器人重用。

组合

制作披萨机器人其实是一些组件的集合，这些组件以团队的形式共同工作。例如，机器人要成功，也许会需要机器臂滚面团，马达起动烤箱等。以OOP的术语来讲，机器人是一个组合的实例，它包含其他对象，这些对象来运作完成相应的指令。每个组件都可以写成类，定义自己的行为以及关系。

像继承和组合这些一般性OOP概念，适用于能够分解成一组对象的任何应用程序。例如，在一般GUI系统中，接口是写成图形组件的集合（按钮、标签等），当绘制图形组件的容器时，图形组件也会跟着绘制（组合）。此外，我们可以编写定制的图形组件——有独特字体的按钮、有新的配色的标签等，这些都是更通用接口机制（继承）的特定化的版本。

从更具体的程序设计观点来看，类是Python的程序组成单元，就像函数和模块一样：类是封装逻辑和数据的另一种方式。实际上，类也定义新的命名空间，在很大程度上就像模块。但是，和我们已见过的其他程序组成单元相比，类有三个重要的独到之处，使其在建立新对象时更为有用。

多重实例

类基本上就是产生对象的工厂。每次调用一个类，就会产生一个有独立命名空间的新对象。每个由类产生的对象都可读取类的属性，并获得自己的命名空间来储存数据，这些数据对于每个对象来说都不同。

通过继承进行定制

类也支持OOP的继承的概念。我们可以在类的外部重新定义其属性从而扩充这个类。更通用的是，类可以建立命名空间的层次结构，而这种层次结构可以定义该结构中类创建的对象所使用的变量名。

www.TopSage.com

运算符重载

通过提供特定的协议方法，类可以定义对象来响应在内置类型上的几种运算。例如，通过类创建的对象可以进行切片、级联和索引等运算。Python提供了一些可以由类使用的钩子，从而能够中断并实现任何的内置类型运算。

大家网 TopSage.com

概览OOP

在我们通过代码了解这些概念的意义之前，对OOP的一般概念再做一些说明。如果你以前从未做过任何面向对象方面的工作，本书一些术语乍看起来可能有点令人困惑。此外，直到你有机会研究程序员如何将这些术语运用于较大系统之前，这些术语的动机也可能难以理解。OOP不仅仅是一门技术，更是一种经验。

属性继承搜索

值得庆幸的是，比起C++或Java等其他语言，Python中OOP的理解和使用都很简单。作为动态类型脚本语言，Python把其他工具中让OOP隐藏的语法杂质和复杂性都去掉了。实际上，Python中大多数OOP故事，都可简化成这个表达式：

```
object.attribute
```

本书一直使用这个表达式读取模块的属性，调用对象的方法等。然而，当我们对`class`语句产生的对象使用这种方式时，这个表达式会在Python中启动搜索 —— 搜索对象连结的树，来寻找`attribute`首次出现的对象。当类启用时，上边的Python表达式实际上等于下列自然语言。

> 找出attribute首次出现的地方，先搜索object，然后是该对象之上的所有类，由下至上，由左至右。

换句话来说，取出属性只是简单的搜索“树”而已。我们称这种搜索程序为继承，因为树中位置较低的对象继承了树中位置较高的对象拥有的属性。当从下至上进行搜索时，连结至树中的对象就是树中所有上层对象所定义的所有属性的集合体，直到树的最顶端。

在Python中实际上就是这样。我们通过代码建立连结对象树，而每次使用`object.attribute`表达式时，Python确实会在运行期间去“爬树”，来搜索属性。为了更具体的说明，图22-1是这种树的一个例子。

www.TopSage.com

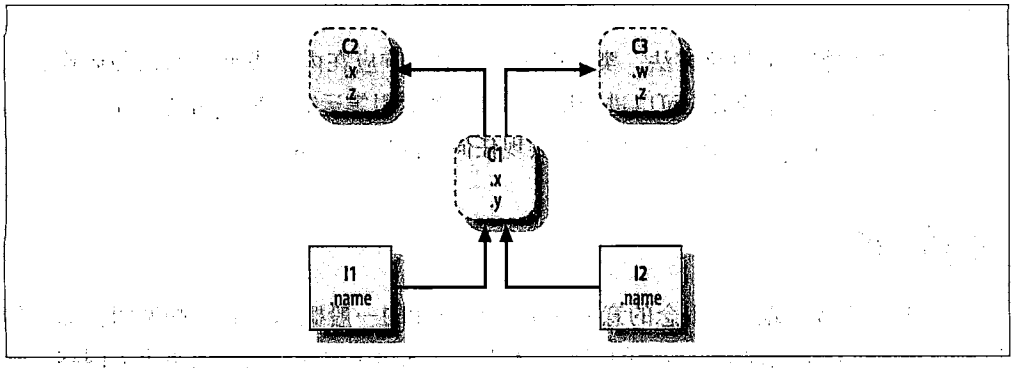


图22-1：类树，底端有两个实例（I1和I2），在它上有个类（C1），而顶端有两个超类（C2和C3）。所有这些对象都是命名空间（变量的封装），而继承就是由下至上搜索此树，来寻找属性名称所出现的最低的地方。代码隐含了这种树的形状

图22-1中，包含了五个对象树，而对象都标识为变量，这些对象全都有相应的属性，可进行搜索。更明确地讲，此树把三个类的对象（椭圆的C1、C2以及C3）以及两个实例对象（矩形的I1和I2）连结至继承搜索树。注意：在Python对象模型中，类和通过类产生的实例是两种不同的对象类型。

类

类是实例工厂。类的属性提供了行为（数据以及函数），所有从类产生的实例都继承该类的属性（例如，通过时薪和小时数计算员工薪水的函数）。

实例

代表程序领域中具体的元素。实例属性记录数据，而每个特定对象的数据都不同（例如，一个员工的社会安全号码）。

就搜索树来看，实例从它的类继承属性，而类是从搜索树中所有比它更上层的类中继承属性。

在图22-1中，我们可以按照椭圆形在树中相对的位置再进一步分类。我们通常把树中位置较高的类称为超类（superclass）（就像C2和C3）。树中位置较低的类则称为子类（就像C1）（注1）。这些术语指的就是相对于树中位置和角色。超类提供了所有子类共享的行为，但是因为搜索是由下而上，子类可能会在树中较低位置重新定义超类的变量名，从而覆盖超类定义的行为。

注1：在其他书籍中，偶尔也会遇到基础类（base class）和衍生类（derived class）来描述超类和子类。

www.TopSage.com

最后几句话其实就OOP软件定制的关键之处，让我们扩展这个概念。假设我们创建了图22-1的树，然后说：

```
I2.w
```

这个代码会立即启用继承。因为这是一个object.attribute表达式，于是会触发图22-1中对树的搜索：Python会查看I2和其上的对象来搜索属性w。确切地讲，就是以下面这个顺序搜索连结的对象：

```
I2, C1, C2, C3
```

找到首个w之后就会停止搜索（或者如果找不到w，就发生一个错误）。此例中，直到搜索C3时才会找到w，因为w只出现在了该对象内。也就是说，通过自动搜索，I2.w会解析为C3.w。就OOP术语而言，I2从C3“继承”了属性w。

最后，这两个实例从如下所示。类中继承了四个属性：w、x、y和z。其他属性的引用则会循着树中其他路径进行。如下所示。

- I1.x和I2.x两者都会在C1中找到x并停止搜索，因为C1比C2位置更低。
- I1.y和I2.y两者都会在C1中找到y，因为这里是y唯一出现的地方。
- I1.z和I2.z两者都会在C2中找到z，因为C2比C3更靠左侧。
- I2.name会找到I2中的name，不需要“爬树”。

通过图22-1的树来跟踪这些搜索，从而可以了解Python中继承搜索的工作方式。

前面的列表中的第一项也许是最需要注意的：因为C1在树中较低的地方重新定义了属性x，相当于有效的取代其上C2中的版本。稍后就会知道，这类重新定义就是OOP中软件定制的重点。通过重新定义和取代属性，C1有效地定制了它从超类中所继承的属性。

类和实例

虽然在Python模型中，类和实例是两种不同的对象类型，但放在这些树中时，它们几乎完全相同：每种类型的主要用途都是用来作为另一种类型的命名空间（变量的封装，也就是我们可以附加属性的地方）。因此，如果类和实例听起来像模块，那也应该如此。然而，类树中的对象也有对其他命名空间对象的自动搜索连结，而类是对应语句，并不是整个文件。

类和实例的主要差异在于，类是一种产生实例的工厂。例如，在现实的应用中，我们可能会有一个Employee类，定义所谓的员工。通过这个类，我们可以产生实际的Employee

www.TopSage.com

实例。这是类和模块的另一个差异：内存中特定模块只有一个实例（所以我们得重载模块以取得其新代码），但是，对类而言，只要有需要，制作多少实例都可以。

从操作的角度来说，类通常都有函数（例如，computeSalary），而实例有其他基本的数据项，类的函数中使用了这些数据（例如，hoursWorked）。事实上，面向对象模型与经典的程序加记录的数据处理模型相比，并没有太多的差异。在OOP中，实例就像是带有“数据”的记录，而类是处理这些记录的“程序”。不过，在OOP中，还有继承层次的概念，和之前的模型相比，更好的支持了软件定制。

类方法调用

在上一节中，例子中的类树介绍了属性的引用I2.w是怎样通过Python中的继承搜索传到C3.w的。不过，也许与了解属性继承同样重要的是，当我们试着调用方法（也就是附属于类的函数属性）时会发生什么。

如果这个I2.w是引用函数调用，它代表的就是“调用C3.w函数来处理I2”。也就是说，Python会自动把I2.w()调用对应为C3.w(I2)调用，将实例当成第一个参数传给继承的函数。

事实上，每当我们调用附属于类的函数时，总会隐含着这个类的实例。这个隐含的主体或环境就是将其称之为面向对象模型的一部分原因：当运算执行时，总是有个主体对象。在更现实的例子中，我们可能会启用名为giveRaise的方法（附属于Employee类的属性）。除非和应该加薪的员工结合在一起使用，不然这种调用是没有意义的。

就像之后我们会看到的那样，Python把隐含的实例传进方法中的第一个特殊的参数，习惯上将其称为self。本书稍后会介绍，方法能通过实例［例如，bob.giveRaise()］或类［例如，Employee.giveRaise(bob)］进行调用，而两种形式在脚本中都有各自的用途。不过，要了解方法如何接收其主体，我们需要写些代码。

编写类树

虽然这里的描述很抽象，这些概念内都有具体的代码。我们以class语句和类调用来构造一些树和对象，这些内容稍后我们会详细介绍。简而言之，内容如下所示。

- 每个class语句会生成一个新的类对象。
- 每次类调用时，就会生成一个新的实例对象。
- 实例自动连结至创建了这些实例的类。

www.TopSage.com

- 类连结至其超类的方式是，将超类列在类头部的括号内。其从左至右的顺序会决定树中的次序。

例如，要建立图22-1的树，我们可以运行这种形式的Python代码（在这里省略了class语句中的内容）。

```
class C2: ...                    # Make class objects (ovals)
class C3: ...
class C1(C2, C3): ...            # Linked to superclasses

I1 = C1()                        # Make instance objects (rectangles)
I2 = C1()                        # Linked to their classes
```

在这里，通过运行三个class语句创建了三个类对象，然后通过两次调用类C1，创建两个实例对象，这就好像它是一个函数一样。这些实例记住了它们来自哪个类，而类C1也记住了它所列出的超类。

从技术角度来讲，这个例子使用的是所谓的多重继承。也就是说，在类树中，类有一个以上的超类。在Python中，如果class语句中的小括号内有一个以上的超类（像这里的C1），它们由左至右的次序会决定超类搜索的顺序。

因为继承搜索以这种方式进行，你要进行属性附加的对象就变得重要起来：这决定了变量名的作用域。附加在实例上的属性只属于那些实例，但附加在类上的属性则由所有子类及其实例共享。稍后，我们将会深入研究把属性增加在这些对象上的代码。我们将会发现：

- 属性通常是在class语句中通过赋值语句添加在类中，而不是嵌入在函数的def语句内。
- 属性通常是在类内，对传给函数的特殊参数（也就是self），做赋值运算而添加在实例中的。

例如，类通过函数（在class语句内由def语句编写而成）为实例提供行为。因为这类嵌套的def会在类中对变量名进行赋值，实际效果就是把属性添加在了类对象之中，从而可以由所有实例和子类继承。

```
class C1(C2, C3):                # Make and link class C1
    def setname(self, who):      # Assign name: C1.setname
        self.name = who          # Self is either I1 or I2

I1 = C1()                        # Make two instances
I2 = C1()
I1.setname('bob')                # Sets I1.name to 'bob'
I2.setname('mel')                # Sets I2.name to 'mel'
print I1.name                    # Prints 'bob'
```


www.TopSage.com

在这样的环境下，def的语法没有什么特别之处。从操作的角度来看，当def出现在像这样的类内部的时候，通常称为方法，而且会自动接收第一个特殊参数（通常称为self），这个参数提供了被处理的实例的参照值（注2）。

因为类是多个实例的工厂，每当需要取出或设定正由某个方法调用所处理的特定的实例的属性时，那些方法通常都会通过这个自动传入的参数self。在之前的代码中，self是用来储存两个实例之一的内部变量名的。

就像简单变量一样，类和实例属性并没有事先声明，而是在首次赋值时它的值才会存在。当方法对self属性进行赋值时，会创建或修改类树底端实例（也就是其中一个矩形）内的属性，因为self自动引用正在处理的实例。

事实上，因为类树中所有对象都不过是命名空间对象，我们可以通过恰当的变量名读取或设置其任何属性。只要变量名C1和I1都位于代码的作用域内，写C1.setname和写I1.setname同样都是有效的。

就目前编写的代码而言，直到setname方法调用前，C1类都不会把name属性附加在实例之上。事实上，调用I1.setname前引用I1.name会产生未定义变量名的错误。如果类想确保像name这样的变量名一定会在其实例中设置，通常都会在构造时填好这个属性。

```
class C1(C2, C3):
    def __init__(self, who):      # Set name when constructed
        self.name = who           # Self is either I1 or I2

I1 = C1('bob')                    # Sets I1.name to 'bob'
I2 = C1('mel')                    # Sets I2.name to 'mel'
print I1.name                     # Prints 'bob'
```

写好并继承后，每次从类产生实例时，Python会自动调用名为__init__的方法。新实例会如往常那样传入__init__的self参数，而列在类调用小括号内的任何值会成为第二以及其后的参数。其效果就是在创建实例时初始化了这个实例，而不需要额外的方法调用。

由于__init__方法的运行时机，它也称作是构造器。这是所谓的运算符重载方法这种较大类型方法中最常用的代表，我们会在接下来几章详细介绍这种方法。这种方法会像往常一样在类树中被继承，而且在变量名开头和结尾都有两个下划线以使其变得特别。当能够支持通信操作的实例出现在对应的运算时，Python就会自动运行它们，而且它们

注2：如果你使用过C++或Java，就知道Python的self相当于this，但是Python中的self一定是明确写出的，这样使属性的读取更为明显。

www.TopSage.com

是使用简单方法调用最常用的替代方法。这类方法也是可选的：省略时，不支持这类运算。

例如，要实现集合交集，类可能会提供名为intersect的方法，或者重载表达式运算符&，也就是编写名为__and__的方法来处理所需要的逻辑。因为运算符机制让实例的用法和外观类似于内置类型，可以让有些类提供一致而自然的接口，从而可以与预期的内置类型的代码兼容。

OOP是为了代码重用

这就是Python中OOP的大部分内容，此外，就是一些语法细节了。当然，除了继承之外，还有些其他的事。例如，运算符重载比这里所说的更为常见：类也可以提供自己实现的运算，例如，索引运算、取出属性和打印等。不过，大体而言，OOP就是在树中搜索属性。

那么，我们为什么对建立和搜索对象树感兴趣？虽然这得具备一些经验后才能了解的，但是妥善使用时，类所支持的代码重用的方式，是Python其他程序组件难以提供的。通过类，我们可以定制现有的软件来编写代码，而不是对现有代码进行在原处的修改，或者每个新项目都从头开始。

从基本的角度来说，类其实就是由函数和其他变量名所构成的包，很像模块。然而，我们从类得到的自动属性继承搜索，支持了软件的高层次的定制，而这是我们通过模块和函数做不到的。此外，类提供了自然的结构，让代码可以把逻辑和变量名区域化，这样也有助于程序的调试。

例如，因为方法只是有特殊第一参数的函数，我们可以把要处理的对象传给简单函数，来模拟其行为。不过，方法参与了类的继承，可让我们自然地通过新方法定义编写子类，通过这样定制现有的软件，而不需要对现有的代码进行在原处的修改。在模块及函数中是没有类似的概念的。

举个例子，假设任务是实现员工的数据库应用程序。作为一个Python OOP程序员，可能会先写个通用的超类，来定义组织中所有员工的默认的通用行为。

```
class Employee:                          # General superclass
    def computeSalary(self): ...        # Common or default behavior
    def giveRaise(self): ...
    def promote(self): ...
    def retire(self): ...
```

一旦你编写了这样的通用行为，就可以针对每个特定种类的员工进行定制，来体现各种

www.TopSage.com

不同类型和一般情况的差异。也就是说，可以编写子类，定制每个类型的员工中不同的行为。这个类型的员工的其他行为则会继承那个通用化的类。例如，如果工程师有独特的薪资计算规则（并非以小时计算），就可以在子类中只取代这一个方法。

```
class Engineer(Employee):          # Specialized subclass
    def computeSalary(self): ...   # Something custom here
```

因为这里的computeSalary版本在类树下面出现，所以会取代（覆盖）Employee中的通用版本。然后，你可以建立员工所属的员工类种类的实例，从而使其获得正确的行为：

```
bob = Employee()                   # Default behavior
mel = Engineer()                   # Custom salary calculator
```

注意：可以对树中任何类创建实例，而不是只有底端的类可以：创建的实例所用的类会决定其属性搜索从哪个层次开始。最后，这两个实例对象可能会嵌入到一个更大的容器对象中（例如，列表或另一个类的实例），利用本章开头所提到的组合概念，从而可以代表部门或公司。

当稍后要取得这些员工的薪资时，可根据创建这个对象的类来计算，这也是基于继承搜索的原理（注3）。

```
company = [bob, mel]               # A composite object
for emp in company:
    print emp.computeSalary()      # Run this object's version
```

这是第4章以及第15章介绍过的多态概点的又一实例。回想一下，多态是指运算的意义取决于运算对象。在这里，computeSalary方法在调用前，会通过继承搜索在每个对象中找到。在其他应用中，多态可用于隐藏（封装）接口差异性。例如，处理数据流的程序可以写成预期有输入和输出方法的对象，而不关心那些方法实际在做的是什么。

```
def processor(reader, converter, writer):
    while 1:
        data = reader.read()
        if not data: break
        data = converter(data)
        writer.write(data)
```

把针对各种数据来源所需读取和写入方法接口定制后的子类的实例传入后，都可以重用这个processor函数，无论什么时候都可以让它来处理所需使用的任何数据来源：

注3：注意：这个例子中的company列表可以使用Python对象的pickle功能（我们在第9章学习文件时介绍过）储存在文件中，以产生永久保存的员工数据库。Python有一个名为shelve的模块，可以把类实例的pickle形式储存在以键读取的文件系统内。第三方开源ZODB系统也是做的相同的事，只不过它对商业级面向对象数据库有着更好的支持。

www.TopSage.com

```
class Reader:
    def read(self): ...                 # Default behavior and tools
    def other(self): ...
class FileReader(Reader):
    def read(self): ...                 # Read from a local file
class SocketReader(Reader):
    def read(self): ...                 # Read from a network socket
...
processor(FileReader(...),   Converter,  FileWriter(...))
processor(SocketReader(...), Converter,  TapeWriter(...))
processor(FtpReader(...),    Converter,  XmlWriter(...))
```

此外，因为读取和写入方法的内部实现已分解至某个独立的位置，修改这些代码是不会与正在使用的代码产生冲突的。实际上，processor函数本身也可以是类，让转换器的转换逻辑通过继承添加，并让读取器和写入器能够通过组合方式嵌入（本书这一步分稍后会说明如何实现）。

一旦习惯了使用这种方式进行程序设计（通过软件定制），你就会发现，当要写新程序时，很多工作早已做好。你的任务大部分就是把已实现的程序所需行为的现有超类混合起来。例如，某人已写好了这个例子中的Employee、Reader以及Writer类，用在完全不同的程序中。如果是这样，你就可以“毫不费力”地采用那个人的所有代码。

事实上，在很多应用领域中，你可以取得或购买超类集合体，也就是所谓的软件框架（framework），把常见程序设计任务实现成类，可以让你在应用程序中混合。这些软件框架可能提供一些数据库接口、测试协议、GUI工具箱等。利用软件框架，只需编写子类，填入所需的一两个方法。树中较高位置的框架类会替你做绝大多数的工作。在OOP中写程序，所需要做的就是通过编写自己的子类，结合和定制已调试的代码。

当然，需要花点时间学习如何充分利用这些类，从而实现这种OOP是理想化。实际应用中，面向对象工作也需要有实质性的设计工作，来全面实现类的代码重用的优点。结果，程序员开始将常见的OOP结构归类，称为设计模式（design pattern），来协助解决设计中的问题。不过，在Python中所编写的用于OOP的实际代码是如此的简单，在探索OOP时，不会增加额外的障碍。想了解原因，请继续学习第23章。

本章小结

本章对类和OOP进行了抽象的学习，在深入学习细节前需要先看一看蓝图。正如我们所见到的，OOP差不多就是在查找连结对象树中的属性。我们将这样的查找称为继承搜索。对象树底端的对象会继承树中较高对象的属性。这种功能让我们可以通过定制代码

www.TopSage.com

来编写程序，而不是进行修改或是从头开始。合理使用时，这种程序设计模型可以大幅减少开发时间。

下一章要开始完成蓝图编码细节。不过，当深入学习Python类时，要记住Python的OOP模型非常简单。就像我们所说的，这其实就是在对象树中搜索属性。在继续之前，先做一做习题，复习一下我们所讲的内容。

www.TopSage.com

本章习题

1. Python的OOP的重要的意义是什么?
2. 继承搜索在哪里查找属性?
3. 类对象和实例对象有什么不同?
4. 为什么类方法函数中的第一个参数特殊?
5. `__init__`方法是做什么用的?
6. 怎样创建类实例?
7. 怎样创建类?
8. 怎样定义类的超类?

习题解答

1. OOP就是代码的重用：分解代码、最小化代码的冗余以及对现存的代码进行定制来编写程序，而不是实地地修改代码，或者从头开始。
2. 继承搜索会先在实例对象中寻找属性，然后才是创建实例的类，之后是所有较高的超类，由对象树底端到顶端，并且从左侧至右侧（默认）。当属性首次找到时，搜索就会停止。因为在此过程中变量名的最低的版本会获胜，类的层次自然而然地支持了通过扩展进行代码的定制。
3. 类和实例对象都是命名空间（由作为属性的变量的包）。两者间主要差别是，类是建立多个实例的工厂。类也支持运算符重载方法，由实例继承，而且把其中的任何函数视为处理实例的特殊的方法。
4. 类方法函数中的第一个参数之所以特殊，是因为它总是接受将方法调用视为隐含主体的实例对象。按惯例，通常称为`self`。因为方法函数默认总是有这个隐含的主体对象环境，所以我们说这是“面向对象”，也就是设计用来处理或修改对象的。
5. 如果类中编写了或继承了`__init__`方法，每次类实例创建时，Python会自动调用它。这也称为构造器。除了明确传入类的名称的任何参数外，还会隐性的传入新实例。这也是最常见的运算符重载方法。如果没有`__init__`方法，实例刚创建时就是一个简单的空的命名空间。
6. 你可以调用类名称（就好像函数一样）来创建类实例。任何传给类名称的参数都要

出现在__init__构造器中第二和其后的参数。新的实例会记得创建它的类，从而可以实现继承目的。

7. 你可以运行 class语句来创建类。就像函数定义一样，这些语句在所在的模块文件导入时，一般就会运行（下一章会再谈）。

8. 定义一个类的超类是通过在class语句的圆括号中将其列出，也就是在新的类名称后。类在圆括号中由左至右列出的顺序，会决定其在类树中由左至右的搜索的顺序。

www.TopSage.com

大象网
TopSage.com

第23章

类代码编写基础

现在，我们已经抽象地学习了OOP，接下来要看的是怎样转换为实际的代码。本章和下一章会介绍Python中类模型的语法细节。

如果过去没用过OOP，若是囫囵吞枣的话，类看起来就有些复杂。为了更容易接受类的编写方法，本章要先看一些实际应用中的基本的类，从而可详细探讨OOP。我们要在本书这一部分后续章节扩展这里介绍的细节，但是就基本形式而言，Python的类是很容易理解的。

类有三个主要的不同之处。从最底层来看，类几乎就是命名空间，很像第5部分研究过的模块。但是，和模块不同的是，类也支持多个对象的产生、命名空间继承以及运算符重载。让我们逐一探索这三种不同之处，开始我们学习class语句之旅吧。

类产生多个实例对象

要了解多个对象的概念是如何工作的，得先了解Python的OOP模型中的两种对象：类对象和实例对象。类对象提供默认行为，是实例对象的工厂。实例对象是程序处理的实际对象：各自都有独立的命名空间，但是继承（可自动存取）创建该实例的类中的变量名。类对象来自于语句，而实例来自于调用。每次调用一个类，就会得到这个类的新的实例。

这种对象生成的概念和我们在本书至今所见的其他任何程序构架有着很大的不同。实际上，类是产生多个实例的工厂。反之，每个模块只有一个副本会导入到某一个程序中（事实上，我们必须调用reload来更新单个模块对象，反应出来对该模块的修改，这就是原因之一）。

下面就是Python OOP本质的简介。正如我们将看到的，从某种程度上来说，Python的类和def及模块很相似，但是它和在其他语言用过的相比可能就大不相同了。

www.TopSage.com

类对象提供默认行为

执行class语句，就会得到类对象。以下是Python类的主要特性的要点。

- **class语句创建类对象并将其赋值给变量名。**就像函数def语句，Python class语句也是可执行语句。执行时，会产生新的类对象，并将其赋值给class头部的变量名。此外，就像def应用，class语句一般是在其所在文件导入时执行的。
- **class语句内的赋值语句会创建类的属性。**就像模块文件一样，class语句内的顶层的赋值语句（不是在def之内）会产生类对象中的属性。从技术角度来讲，class语句的作用域会变成类对象的属性的命名空间，就像模块的全局作用域一样。执行class语句后，类的属性可由变量名点号运算获取object.name。
- **类属性提供对象的状态和行为。**类对象的属性记录状态信息和行为，可由这个类所创建的所有实例共享。位于类中的函数def语句会生成方法，方法将会处理实例。

实例对象是具体的元素

当调用类对象时，我们得到了实例对象。以下是类的实例内含的重点概要。

- **像函数那样调用类对象会创建新的实例对象。**每次类调用时，都会建立并返回新的实例对象。实例代表了程序领域中的具体元素。
- **每个实例对象继承类的属性并获得了自己的命名空间。**由类所创建的实例对象是新命名空间。一开始是空的，但是会继承创建该实例的类对象内的属性。
- **在方法内对self属性做赋值运算会产生每个实例自己的属性。**在类方法函数内，第一个参数（按惯例称为self）会引用正处理的实例对象。对self的属性做赋值运算，会创建或修改实例内的数据，而不是类的数据。

第一个例子

下面看一个真实的例子，注意这些概念在实际中是如何工作的。首先，定义一个名为FirstClass的类，通过交互模式运行Python class语句。

```
>>> class FirstClass:                    # Define a class object
...     def setdata(self, value):        # Define class methods
...         self.data = value            # self is the instance
...     def display(self):
...         print self.data              # self.data: per instance
...
```


www.TopSage.com

这里是在交互模式下工作，但一般来说，这种语句应该是当其所在的模块文件导入时运行的。就像通过def建立的函数，这个类在Python抵达并执行语句前是不会存在的。

就像所有复合语句一样，class开头一行会列出类的名称，后面再接一或多个内嵌并且（通常）缩进的语句的主体。在这里，嵌套的语句是def，定义类要实现导入的行为的函数。就像我们学到的，def其实是赋值运算。在这里是把函数对象赋值给变量名setdata，而且display位于class语句范围内，因此会产生附加在类上的属性：FirstClass.setdata和FirstClass.display。事实上，在类嵌套的代码块中的顶层的赋值的任何变量名，都会变成类的属性。

位于类中的函数通常称为方法。方法是普通def，支持先前学过的函数的所有内容（可以有默认参数、返回值等）。在方法函数中，调用时，第一个参数自动接收隐含的实例对象：调用的主体。我们需要建立一些实例来理解它是如何工作的：

```
>>> x = FirstClass()                # Make two instances
>>> y = FirstClass()                # Each is a new namespace
```

以此方式调用类时（注意小括号），会产生实例对象，也就是可读取类属性的命名空间。确切地讲，此时有三个对象：两个实例和一个类。其实是有三个连结命名空间，如图23-1所示。以OOP观点来看，我们说x是一个FirstClass对象， y也是。

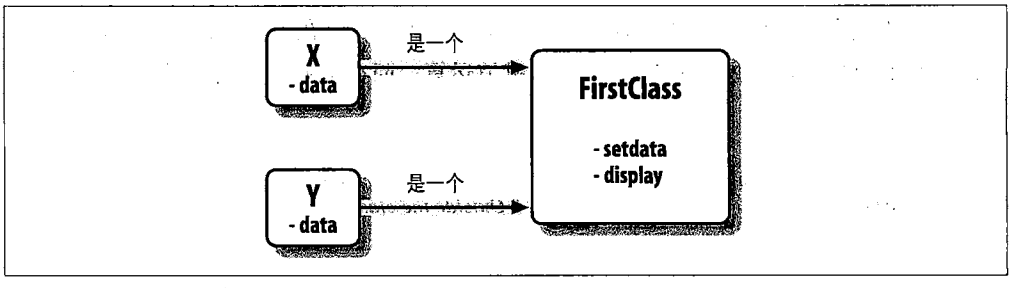


图23-1：类和实例是类树中通过继承搜索的相连的命名空间。这里，“data”属性会在实例内找到，但是“setdata”和“display”则是在它们之上的类中找到

这两个实例一开始是空的，但是它们被连接到创建它们类。如果对实例以及类对象内的属性名称进行点号运算，Python会通过继承搜索从类取得变量名（除非该变量名位于实例内）：

```
>>> x.setdata("King Arthur")        # Call methods: self is x
>>> y.setdata(3.14159)              # Runs: FirstClass.setdata(y, 3.14159)
```

x或y本身都没有setdata属性，为了寻找这个属性，Python会顺着实例到类的连结搜

www.TopSage.com

索。而这就是所谓的Python的继承：继承是在属性点号运算时发生的，而且只与查找连结对象内的变量名（例如，图23-1的“是一个”连结）有关。

在FirstClass的setdata函数中，传入的值会赋值给self.data。在方法中，self（按惯例，这是最左侧参数的名称）会自动引用正在处理的实例（x或y），所以赋值语句会把值储存在实例的命名空间，而不是类的命名空间（这是图23-1中变量名data的创建的方式）。

因为类会产生多个实例，方法必须经过self参数才能获取正在处理的实例。当调用类的display方法来打印self.data时，会发现每个实例的值都不同。另外，变量名display在x和y之内都相同，因为它是来自于（继承自）类的：

```
>>> x.display()                    # self.data differs in each instance
King Arthur
>>> y.display()
3.14159
```

注意：在每个实例内的data成员储存了不同对象类型（字符串和浮点数）。就像Python中的其他事物，实例属性（有时被称做成员）并没有声明。首次赋值后，就会存在，就像简单的变量。事实上，如果在调用setdata之前，就对某一实例调用display，就会触发未定义变量名的错误：data属性以setdata方法赋值前，是不会在内存中存在的。

另一种喜欢这个模型的动态性的方式是，考虑一下我们可以在类的内部或外部修改实例属性。在类内时，通过方法内对self进行赋值运算；而在类外时，则可以通过对实例对象进行赋值运算：

```
>>> x.data = "New value"           # Can get/set attributes
>>> x.display()                    # Outside the class too
New value
```

虽然比较少见，通过在类方法函数外对变量名进行赋值运算，我们甚至可以在实例命名空间内产生全新的属性：

```
>>> x.anothername = "spam"         # Can set new attributes here too
```

这样会增加一个名为anothername的新属性；实例对象x的任何类方法都可使用它，也可不使用它。类通常是通过对self参数进行赋值运算从而建立实例的所有属性的，但不是必须如此。程序可以取出、修改或创建其所引用的任何对象的属性。

www.TopSage.com

类通过继承进行定制

除了作为工厂来生成多个实例对象之外，类也可引入新组件（子类）来进行修改，而不对现有组件进行实地的修改。由类产生的实例对象会继承该类的属性。Python也可让类继承其他类，因而开启了编写类层次结构的大门，在阶层较低的地方覆盖现有的属性，让行为特定化。在这里和模块并不一致：模块的属性存在于一个单一、平坦的命名空间之内。

在Python中，实例从类中继承，而类继承于超类。以下是属性继承机制的核心观点。

- **超类列在了类开头的括号中。**要继承另一个类的属性，把该类列在`class`语句开头的括号中就可以了。含有继承的类称为子类，而子类所继承的类就是其超类。
- **类从其超类中继承属性。**就像实例继承其类中所定义的属性名一样，类也会继承其超类中定义的所有属性名称。当读取属性时，如果它不存在于子类中，Python会自动搜索这个属性。
- **实例会继承所有可读取类的属性。**每个实例会从创建它的类中获取变量名，此外，还有该类的超类。寻找变量名时，Python会检查实例，然后是它的类，最后是所有超类。
- **每个object.attribute都会开启新的独立搜索。**Python会对每个属性取出表达式进行对类树的独立搜索。这包括在`class`语句外对实例和类的引用（例如，`X.attr`），以及在类方法函数内对`self`实例参数属性的引用。方法中的每个`self.attr`表达式都会开启对`self`及其上层的类的`attr`属性的搜索。
- **逻辑的修改是通过创建子类，而不是修改超类。**在树中层次较低的子类中重新定义超类的变量名，子类就可取代并定制所继承的行为。

这种搜索的结果和主要目的就是，类支持了程序的分解和定制，比迄今为止所见到的其他任何语言工具都要好。另外，这样可以把程序的冗余度降到最低（减少维护成本），也就是把操作分解为单一、共享的实现。此外，这样写程序时，也可让我们对现有的程序代码进行定制，而不是在实地进行修改或是从头开始。

第二个例子

下个例子是建立在上一个例子基础之上的。首先，我们会定义一个新的类`SecondClass`，继承`FirstClass`所有变量名，并提供其自己的一个变量名。

```
>>> class SecondClass(FirstClass):                # Inherits setdata
...     def display(self):                        # Changes display
```


www.TopSage.com

```
...         print 'Current value = "%s"' % self.data
...
```

SecondClass定义display方法以不同格式打印。定义一个和FirstClass中的属性同名的属性，SecondClass有效的取代其超类内的display属性。

回想一下，继承搜索会从实例往上进行，之后到子类，然后到超类，直到所找的属性名称首次出现为止。在这个例子中，因为SecondClass中的变量名display会在FirstClass内首先被找到， SecondClass覆盖了FirstClass中的display。有时候，我们把这种在树中较低处发生的重新定义，取代属性的动作称为重载。

结果就是SecondClass改变了方法display的行为，把FirstClass特定化了。另外，SecondClass（以及其任何实例）依然会继承FirstClass的setdata方法。用一个例子来说明：

```
>>> z = SecondClass()
>>> z.setdata(42)                    # Finds setdata in FirstClass
>>> z.display()                      # Finds overridden method in SecondClass
Current value = "42"
```

就像往常一样，我们调用SecondClass创建了其实例对象。setdata依然是执行FirstClass中的版本，但是这一次display属性是来自SecondClass，并打印定制的内容。图23-2描绘了其中涉及到的命名空间。

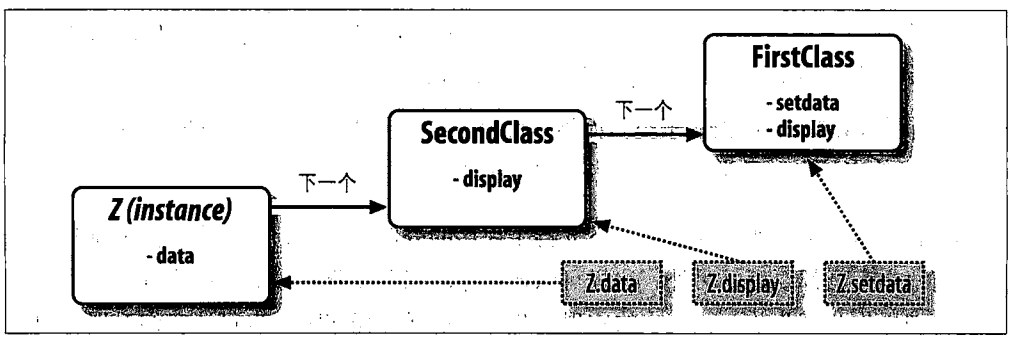


图23-2：在类树中较低的扩展类中重新定义变量名，从而覆盖了继承的变量名并将其专有化。在这里，SecondClass重新定义了方法display，从而定制了它的实例的display方法

这里有一件和OOP相关的很重要的事要留意：SecondClass引入的专有化完全是在FirstClass外部完成的。也就是说，不会影响当前存在的或未来的FirstClass对象，就像上一个例子中的x：

```
>>> x.display()                      # x is still a FirstClass instance (old message)
New value
```


www.TopSage.com

我们不是修改FirstClass，而是对它进行了定制。很自然，这是有意而为之的例子，但是作为一条规则，因为继承可以让我们像这样在外部组件内（也就是在子类内）进行修改，类所支持的扩展和重用通常比函数或模块更好。

TopSage.com

类是模块内的属性

在继续学习之前，请记住类的名称没有什么神奇之处。当class语句执行时，这只是赋值给对象的变量，而对象可以用任何普通表达式引用。例如，如果FirstClass是写在模块文件内，而不是在交互模式下输入的，就可将其导入，在类开头的那行可以正常的使用它的名称。

```
from modulename import FirstClass          # Copy name into my scope
class SecondClass(FirstClass):             # Use class name directly
    def display(self): ...
```

或者，其等效写法如下。

```
import modulename                          # Access the whole module
class SecondClass(modulename.FirstClass):  # Qualify to reference
    def display(self): ...
```

就像其他一切事物一样，类名称总是存在于模块中，所以必须遵循第5部分学到的所有规则。例如，单一模块文件内可以有一个以上的类，就像模块内其他语句，class语句会在导入时执行已定义的变量名，而这些变量名会变成独立的模块属性。更通用的情况是，每个模块可以任意混合任意数量的变量、函数以及类，而模块内的所有变量名的行为都相同。文件*food.py*示范如下。

```
# food.py

var = 1                                    # food.var
def func():                                # food.func
    ...
class spam:                                # food.spam
    ...
class ham:                                 # food.ham
    ...
class eggs:                                # food.eggs
...
```

如果模块和类碰巧有相同名称，也是如此。例如，文件*person.py*，写法如下。

```
class person:
    ...
```

需要像往常一样通过模块获取类：

www.TopSage.com

```
import person                    # Import module
x = person.person()              # Class within module
```

虽然这个路径看起来是多余的，但却是必须的：person.person指的是person模块内的person类。只写person只会取得模块，而不是类，除非使用from语句。

```
from person import person        # Get class from module
x = person()                     # Use class name
```

就像其他的变量一样，没有预先导入，并且从其所在文件中将其取出，我们是无法看见文件中的类的。如果这看起来令人困惑，就别让模块和该模块内的类使用相同名称。

此外，虽然类和模块都是附加属性的命名空间，它们是非常不同的源代码结构：模块反应了整个文件，而类只是文件内的语句。我们会在这一部分稍后介绍这种区别。

类可以截获Python运算符

现在，让我们来看类和模块的第三个主要差别：运算符重载。简而言之，运算符重载就是让用类写成的对象，可截获并响应用在内置类型上的运算：加法、切片、打印和点号运算等。这只是自动分发机制：表达式和其他内置运算流程要经过类的实现来控制。这里也和模块没有什么相似之处：模块可以实现函数调用，而不是表达式的行为。

虽然我们可以把所有类行为实现为方法函数，运算符重载则让对象和Python的对象模型更紧密地结合起来。此外，因为运算符重载，让我们自己的对象行为就像内置对象那样，这可促进对象接口更为一致并更易于学习，而且可让类对象由预期的内置类型接口的代码处理。以下是重载运算符主要概念的概要。

- **以双下划线命名的方法（__X__）是特殊钩子。**Python运算符重载的实现是提供特殊命名的方法来拦截运算。Python语言替每种运算和特殊命名的方法之间，定义了固定不变的映射关系。
- **当实例出现在内置运算时，这类方法会自动调用。**例如，如果实例对象继承了__add__方法，当对象出现在+表达式内时，该方法就会调用。该方法的返回值会变成相应表达式的结果。
- **类可覆盖多数内置类型运算。**有几十种特殊运算符重载的方法的名称，几乎可截获并实现内置类型的所有运算。它不仅包括了表达式，而且像打印和对象建立这类基本运算也包括在内。
- **运算符覆盖方法没有默认值，而且也不需要。**如果类没有定义或继承运算符重载方

www.TopSage.com

法，就是说相应的运算在类实例中并不支持。例如，如果没有__add__，+表达式就会引发异常。

- **运算符可让类与Python的对象模型相集成。**重载类型运算时，以类实现的用户定义对象的行为就会像内置对象一样，因此，提供了一致性，以及与预期的接口的兼容性。

运算符重载是可选的功能。主要是替其他Python程序员开发工具的人在使用它，而不是那些应用程序开发人员在使用。此外，不客气地讲，不要因为这看起来很“酷”就随便去试用。除非类需要模仿内置类型接口，不然应该使用更简单的命名的方法。例如，员工数据库应用程序为什么要支持像*和+这类表达式呢？通常来说，像giveRaise和promote这类名称的方法更有意义。

因此，我们不会在本书中深入讨论Python每个可用的运算符重载方法。不过，有个运算符重载方法，你可能会在每个现实的Python类中遇见：__init__方法，也称为构造器方法，它是用于初始化对象的状态的。你应该特别注意这个方法，因为__init__和self参数是了解Python的OOP程序代码的关键之一。

第三个例子

这是另一个例子。这一次，我们要定义SecondClass的子类，实现三个特殊名称的属性，让Python自动进行调用：当新的实例构造时，会调用__init__（self是新的ThirdClass对象），而当ThirdClass实例出现在+或*表达式中时，则分别会调用__add__和__mul__。以下是新子类。

```
>>> class ThirdClass(SecondClass):                # Is a SecondClass
...     def __init__(self, value):                # On "ThirdClass(value)"
...         self.data = value
...     def __add__(self, other):                  # On "self + other"
...         return ThirdClass(self.data + other)
...     def __mul__(self, other):
...         self.data = self.data * other          # On "self * other"
...
>>> a = ThirdClass("abc")        # New __init__ called
>>> a.display()                  # Inherited method
Current value = "abc"

>>> b = a + 'xyz'                # New __add__: makes a new instance
>>> b.display()
Current value = "abcxyz"

>>> a * 3                        # New __mul__: changes instance in-place
>>> a.display()
Current value = "abcabcabc"
```


www.TopSage.com

ThirdClass“是一个”SecondClass对象，所以其实例会继承SecondClass的display方法。但是，ThirdClass生成的调用现在会传递一个参数（例如，“abc”），这是传给__init__构造器内的参数value的，并将其赋值给self.data。此外，ThirdClass对象现在可以出现在+和*表达式中。Python把左侧的实例对象传给self参数，而右边的值传给other，如图23-3所示。

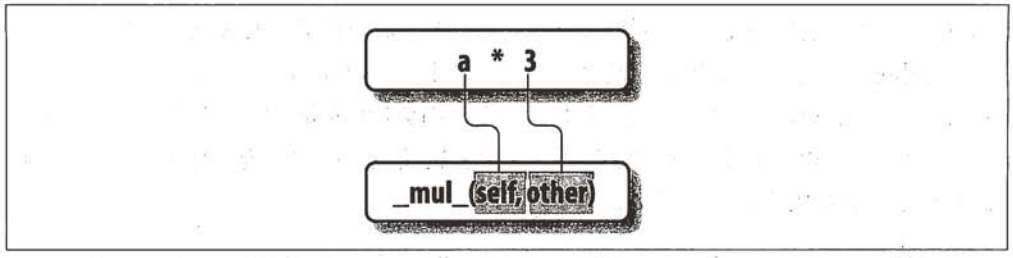


图23-3：在运算符重载中，在类的实例上执行的表达式运算符，和其他内置运算都会对应到类中特殊名称的方法。这些特殊方法是选用的，也可像平常那样继承。在这里，“*”表达式会触发“__mul__”方法

__init__和__add__这样的特殊命名的方法会由子类和实例继承，就像这个类中赋值的其他变量名。如果方法没有在类中编写，Python会在其所有超类内寻找这类变量名。运算符重载方法的名称并不是内置变量或保留字。只是当对象出现在不同的环境时，Python会去搜索的属性。Python通常会自动进行调用，但偶尔也能由程序代码调用（稍后会谈到这个问题。例如，__init__通常可手动调用来触发超类的构造器）。

注意：__add__方法创建并返回这个类的新的实例对象（通过它的结果值调用ThirdClass），但是，__mul__会在原处修改当前的实例对象（通过重新赋值self属性）。这一点和内置类型的行为不同，就像数字和字符串，总是替*运算符制作新对象。因为运算符重载其实只是表达式对方法的分发机制，可以在自己的类对象中以任何喜欢的方式解释运算符（注1）。

为什么要使用运算符重载

作为一名类的设计者，你可以选择使用或不使用运算符重载。你的抉择取决于有多想让对象的用法和外观看起来更像内置类型。就像前边提到的那样，如果省略运算符重载方

注1：　但是，你可能不应该这么做。一般情况下，重载的运算符应该和内置运算符的实现的运作方式相同。就此而言，这意谓着__mul__方法应该返回新对象来作为其结果，而不是在实地修改实例（self）。要做原处的修改，比起*重载，mul方法调用会是比较好的形式[例如，mul(3)，而不是a * 3]。

www.TopSage.com

法，并且不从超类中继承该方法，实例就不支持相应的运算；如果试着使用这个实例进行运算，就会抛出异常（或者使用标准的默认值）。

坦白地讲，只有在实现本质为数学的对象时，才会用到许多运算符重载方法。例如，向量或矩阵类可以重载加法运算符，但员工类可能就不用。就较简单的类而言，可能根本不会用到重载，而应该利用明确的方法调用来实现对象的行为。

另外，如果需要传递用户定义的对象给预期的内置类型（例如，列表或字典）可用的运算符的函数，可能就会决定使用运算符重载。在类内实现同一组运算符，可以保证对象会支持相同的预期的对象接口，因此会与这个函数兼容。

几乎每个实际的类似乎都会出现的一个重载方法是：`__init__`构造器方法。因为这可让类立即在其新建的实例内添加属性，对于每种你可能会写的类而言，构造器方法都是有用的。事实上，虽然Python不会对实例的属性进行声明，通常也可以找到类的`__init__`方法的代码，以了解它实例有哪些属性。本书不会谈太多这种高级功能，但是第24章会介绍一些其他的继承和运算符重载的技术。

世界上最简单的Python类

我们在本章详细研究过`class`语句语法，不过类产生的基本的继承模型其实非常简单：所涉及的就是在连结的对象树中搜索属性。实际上，我们建立的类中可以什么东西都没有。下列语句建立一个类，其内完全没有附加的属性（空的命名空间对象）。

```
>>> class rec: pass                    # Empty namespace object
```

因为没有写任何方法，所以我们需要无操作的pass语句（第13章讨论过）。以交互模式执行此语句，建立这个类后，就可以完全在最初的class语句外，通过赋值变量名给这个类增加属性：

```
>>> rec.name = 'Bob'                   # Just objects with attributes
>>> rec.age  = 40
```

通过赋值语句创建这些属性后，就可以用一般的语法将它们取出。这样用时，类差不多就像C的struct或者Pascal的record：这种对象就是有字段附加在它的上边（我们也可以用字典的键做类似的事情，但是需要额外的字符）。

```
>>> print rec.name                     # Like a C struct or a record
Bob
```

注意：其实该类还没有实例，也能这样用。类本身也是对象，也是没有实例。事实上，

类只是独立完备的命名空间，只要有类的引用值，就可以在任何时刻设定或修改其属性。不过，当建立两个实例时，看看会发生什么事情：

```
>>> x = rec()                    # Instances inherit class names
>>> y = rec()
```

这些实例最初完全是空的命名空间对象。不过，因为它们知道创建它们的类，所以会因继承并获取附加在类上的属性：

```
>>> x.name, y.name               # Name is stored on the class only here
('Bob', 'Bob')
```

其实，这些实例本身没有属性。它们只是从类对象那里取出name属性。不过，如果把一个属性赋值给一个实例，就会在该对象内创建（或修改）该属性，而不会因属性的引用而启动继承搜索，因为属性赋值运算只会影响属性赋值所在的对象。在这里，x得到自己的name，但y依然继承附加在它的类上的name：

```
>>> x.name = 'Sue'               # But assignment changes x only
>>> rec.name, x.name, y.name
('Bob', 'Sue', 'Bob')
```

事实上，当进行下一章的深入探索时，命名空间对象的属性通常都是以字典的形式实现的，而类继承树（一般而言）只是连结至其他字典的字典而已。如果知道在哪里去搜索，的确会看到这一点。__dict__属性是大多数基于类的对象的命名空间字典。

```
>>> rec.__dict__.keys()
['age', '__module__', '__doc__', 'name']

>>> x.__dict__.keys()
['name']

>>> y.__dict__.keys()
[]
```

在这里，类的字典显示出我们进行赋值了的name和age属性，x有自己的name，而y依然是空的。不过，每个实例都连结至其类以便于继承，如果你想查看的话，这个连结叫做__class__：

```
>>> x.__class__
<class __main__.rec at 0x00BAFF60>
```

类也有__bases__属性，也就是其超类构成的元组。这两个属性是Python在内存中类树常量的表示方式。

揭开其奥秘的重点就是，Python的类模型相当动态。类和实例只是命名空间对象，属性

www.TopSage.com

是通过赋值语句动态建立。恰巧这些赋值语句往往在class语句内。只要能引用树中任何一个对象的任意地方，都可以发生。

即使是方法（通常是在类中通过def创建）也可以完全独立的在任意类对象的外部创建。例如，下列在任意类之外定义了一个简单函数，并带有一个参数。

```
>>> def upperName(self):
...     return self.name.upper()    # Still needs a self
```

这里与类完全没有什么关系：这是一个简单函数，在此时就能予以调用，只要我们传进一个带有name属性的对象（变量名self并没有使这变得特别）。不过，如果我们把这个简单函数的赋值成类的属性，就会变成方法，可以由任何实例调用（并且通过类名称本身，只要我们手动传入一个实例）（注2）。

```
>>> rec.method = upperName

>>> x.method()                      # Run  method to process x
'SUE'

>>> y.method()                      # Same, but  pass y to self
'BOB'

>>> rec.method(x)                   # Can call through instance or class
'SUE'
```

在通常情况下，类是由class语句填充的，而实例的属性则是通过在方法函数内对self属性进行赋值运算而创建的。不过，重点在于并不是必须如此。Python中的OOP其实就是在已连结命名空间对象内寻找属性而已。

本章小结

本章介绍Python编写类的基础知识。我们研究过class语句的语法，了解了它是如何用于创建类的继承树的。我们也研究了Python如何自动添加方法函数内的第一个参数，属性如何通过简单赋值语句，而把属性加到类树中的对象，以及特殊名称运算符重载方法，如何替实例截获并实现内置运算（例如，表达式和打印）。

注2：实际上，这正是self参数必须在Python方法中明确列出的原因之一：因为方法可以独立于类之外，创建为一个简单函数。因此，必须让隐含的实例参数明确化才行。否则，Python无法猜测简单函数是否最终会变成类的方法。不过，让self参数明确化的主要原因是为了让变量名的意义更为明确：没有通过self而引用的变量名是简单变量，而通过self引用的变量名则显然是实例的属性。

www.TopSage.com

下一章我们要继续讨论类的编写，再次介绍这个模型，填加一些为了让事情保持简单而在这里省略的细节。我们也会开始探索一些较大型以及更现实的类。不过，首先得做一做习题，复习本章介绍的基础知识。

www.TopSage.com

本章习题

1. 类和模块之间有什么关系？
2. 实例和类是如何创建的？
3. 类属性是在哪里创建的？是怎样创建的？
4. 实例属性是在哪里创建的？是怎样创建的？
5. Python类中的self有什么意义？
6. Python类中如何编写运算符重载？
7. 什么时候可能在类中支持运算符重载？
8. 哪个运算符重载方法是最常用的？
9. Python OOP程序代码中最重要的两个概念是什么？

习题解答

1. 类总是位于模块中；类是模块对象的属性。类和模块都是命名空间，但类对应于语句（而不是整个文件），而且支持多个实例、继承以及运算符重载这些OOP概念。总之，模块就像是一个单个的实例类，没有继承，而且模块对应于整个文件的代码。
2. 类是通过运行class语句创建的；实例是像函数那样调用类来创建的。
3. 类属性的创建是通过把属性赋值给类对象实现的。类属性通常是由class语句中的顶层赋值语句而产生的：每个在class语句代码区中赋值的变量名，会变成类对象的属性（从技术角度来讲，class语句的作用域会变成类对象的属性的命名空间）。不过，也可以于任何引用类对象的地方（在class语句外）对其属性赋值，从而也可以创建类属性。
4. 实例属性是通过对实例对象赋值属性来创建的。实例属性一般是在class语句中的类方法函数中对self参数（永远是隐含实例）赋值属性而创建的。不过，你也可以在任何地方引用实例通过赋值语句来创建属性，即使是在class语句外。一般来说，所有实例属性都是在__init__构造器中初始化的。这样的话，之后的方法调用都可假设属性已经存在。
5. self通常是给与类方法函数中的第一个（最左侧）参数的名称；Python会自动填入实例对象（也就是方法调用的隐含的主体）。这个参数不必叫self。其位置才是重

www.TopSage.com

大家网
TopSage.com

点（C++或Java程序员可能更喜欢把它称作this，因为在这些语言中，该名称反应的是相同的概念。不过，在Python中，这个参数总是需要明确的）。

6. Python类中的运算符重载是有特定名称的方法写成的。这些方法的开头和结尾都是双下划线，通过这种办法使其变得独特。这些不是内置或保留字。当实例出现在相应的运算中时，Python就会自动执行它们。Python为这些运算和特殊方法的名称定义了对应关系。

7. 运算符重载可用于实现模拟内置类型的对象（例如，序列或像矩阵这样的数值对象），以及模拟代码中所预期的内置类型接口。模拟内置类型的接口可让你传入具有状态信息（也就是记住操作调用之间的数据的属性）的类实例。不过，当简单命名的方法就够用时，不应该使用运算符重载。

8. __init__构造器是最常用的。几乎每个类都使用这个方法为实例属性进行初始化，以及执行其他的启动任务。

9. 方法函数中的特殊self参数和__init__构造器方法是Python中OOP的两个基石。

www.TopSage.com

第24章

类代码编写细节

如果你没有全搞懂第23章的内容，请别担心，我们已经很快浏览了类编码，现在将会更深入的研究之前介绍过的概念。在这一章中，我们要从另一个角度看待类、方法、继承以及运算符重载，正式讲解第23章介绍的一些编写类概念，并进行扩展。因为类是最后一个命名空间工具，我们也要在这里总结Python中命名空间的概念。本章还会介绍一些比以前所见更大规模并且更现实的类，包括用一个例子把我们所学到有关OOP的事结合起来。

class语句

虽然Python class语句表面上看起来与其他OOP语言的工具类似，但仔细观察时，和一些程序员习惯的东西其实有这很大的不同。例如，class语句是Python主要的OOP工具，但与C++不同的是，Python的class并不是声明式的。就像def一样，class语句是对象的创建者并且是一个隐含的赋值运算。执行时，会产生类对象，把其引用值储存在开头的行中所使用的变量名。此外，像def一样，class语句也是真正的可执行代码。直到Python抵达并运行定义的class语句前，你的类都不存在（一般都是在其所在模块被导入时，在这之前都不会存在）。

一般形式

class是复合语句，其缩进语句的主体一般都出现在头一行下边的。在头一行中，超类列在类名称之后的括号内，由逗号相隔。列出一个以上的超类会引起多重继承（下一章会进一步讨论）。以下是class语句的一般形式。

```
class <name>(superclass,...):          # Assign to name
    data = value                       # Shared class data
    def method(self,...):              # Methods
        self.member = value            # Per-instance data
```


www.TopSage.com

在class语句内，任何赋值语句都会产生类属性，而且还有特殊名称方法重载运算符。例如，名为__init__的函数会在实例对象构造时调用（如果定义过的话）。

例子

就像我们见过的那样，类几乎就是命名空间，也就是定义变量名（属性）的工具，把数据和逻辑导出给客户端。那么，怎样从class语句得到命名空间的呢？

过程如下。就像模块文件，位于class语句主体中的语句会建立其属性。当Python执行class语句时（不是调用类），会从头至尾执行其主体内的所有语句。在这个过程中，进行的赋值运算会在这个类作用域中创建变量名，从而成为对应的类对象内的属性。因此，类就像模块和函数：

- 就像函数一样，class语句是作用域，由内嵌的赋值语句建立的变量名，就存在在这个本地作用域内。
- 就像模块内的变量名，在class语句内赋值的变量名会变成类对象中的属性。

类的主要的不同之处在于其命名空间也是Python继承的基础。在类或实例对象中找不到的所引用的属性，就会从其他类中获取。

因为class是复合语句，任何种类的语句都可位于其主体内：print、=、if、def等。当class语句自身运行时（不是稍后调用类来创建实例的时候），class语句内的所有语句都会执行。在class语句内赋值的变量名，会创建类属性，而内嵌的def则会创建类方法，但是，其他赋值语句也可制作属性。

例如，把简单的非函数的对象赋值给类属性，就会产生数据属性，由所有实例共享。

```
>>> class SharedData:
...     spam = 42                    # Generates a class data attribute
...
>>> x = SharedData()                 # Make two instances
>>> y = SharedData()
>>> x.spam, y.spam                   # They inherit and share spam
(42, 42)
```

在这里，因为变量名spam是在class语句的顶层进行赋值的，因此会被附加在这个类

www.TopSage.com

中，从而被所有的实例共享。我们可通过类名称修改它，或者是通过实例或类引用它（注1）。

```
>>> SharedData.spam = 99
>>> x.spam, y.spam, SharedData.spam
(99, 99, 99)
```

这种类属性可以用于管理贯穿所有实例的信息。例如，所产生的实例的数目的计数器（我们会在第26章进一步扩展这个概念）。现在，如果我们通过实例而不是类来给变量名spam赋值时，看看会发生什么：

```
>>> x.spam = 88
>>> x.spam, y.spam, SharedData.spam
(88, 99, 99)
```

对实例的属性进行赋值运算会在该实例内创建或修改变量名，而不是在共享的类中。通常的情况下，继承搜索只会在属性引用时发生，而不是在赋值运算时发生：对对象属性进行赋值总是会修改该对象，除此之外没有其他的影响（注2）。例如，`y.spam`会通过继承而在类中查找，但是，对`x.spam`进行赋值运算则会把该变量名附加在x本身上。

下面这个例子，可以更容易理解这种行为，把相同的变量名储存在两个位置。假设我们执行下列类。

```
class MixedNames:                          # Define class
    data = 'spam'                          # Assign class attr
    def __init__(self, value):            # Assign method name
        self.data = value                  # Assign instance attr
    def display(self):
        print self.data, MixedNames.data   # Instance attr, class attr
```

这个类有两个`def`，把类属性与方法函数绑定在一起。此外，也包含一个=赋值语句。因为赋值语句是在类中赋值变量名`data`，该变量名会在这个类的作用域内存在，变成类对象的属性。就像所有类属性，这个`data`会被继承，从而被所有没有自己的`data`属性的类的实例所共享。

当创建这个类的实例的时候，变量名`data`会在构造器方法内对`self.data`进行赋值运算，从而把`data`附加在这些实例上。

注1：如果你用过C++，大概会认出这与C++的静态数据成员的概念有些类似：也就是储存在类中的成员，与实例是不相关的。在Python中，这没有什么特别的：所有类属性都是在class语句中的赋值的变量名，无论它们是否恰巧引用的是函数（C++的方法）或其他东西（C++的成员）。

注2：除非该类用`__setattr__`运算符重载方法重新定义了属性的赋值运算，去做其他的事。

www.TopSage.com

```
>>> x = MixedNames(1)          # Make two instance objects
>>> y = MixedNames(2)          # Each has its own data
>>> x.display(); y.display()   # self.data differs, Subclass.data is the same
1 spam
2 spam
```

结果就是，data存在于两个地方：在实例对象内（由__init__中的self.data赋值运算所创建）以及在实例继承变量名的类中（由类中的data赋值运算所创建）。类的display方法打印了这两个版本，先以点号运算得到self实例的属性，然后才是类。

利用这些技术把属性储存在不同对象内，我们可以决定其可见范围。附加在类上时，变量名是共享的；附加在实例上时，变量名是属于每个实例的数据，而不是共享的行为或数据。虽然继承搜索会查找变量名，但总是可以通过直接读取所需要的对象，而获得树中任何地方的属性。

例如，在上一个例子中，明确了x.data或self.data，都会返回实例的名称（通常是隐藏在类中的相同名称）；然而，MixedNames.data则是明确地找出类的名称。我们之后将会看到这种编程模式的各种角色。下一节会说明其中最常用的一种。

方法

因为你已经了解了函数，那么你就了解了类中的方法。方法位于class语句的主体内，是由def语句建立的函数对象。从抽象的视角来看，方法替实例对象提供了要继承的行为。从程序设计的角度来看，方法的工作方式与简单函数完全一致，只是有个重要差异：方法的第一个参数总是接收方法调用的隐性主体，也就是实例对象。

换句话说，Python会自动把实例方法的调用对应到类方法函数，如下所示。方法调用需通过实例，就像这样：

```
instance.method(args...)
```

这会自动翻译成以下形式的类方法函数调用：

```
class.method(instance, args...)
```

class通过Python继承搜索流程找出方法名称所在之处。事实上，两种调用形式在Python中都有效。

除了方法属性名称是正常的继承外，第一个参数就是方法调用背后唯一的神奇之处。在类方法中，按惯例第一个参数通常都称为self（严格地说，只有其位置重要，而不是它

www.TopSage.com

的名称）。这个参数提供方法一个钩子，从而返回调用的主体，也就是实例对象：因为类可以产生许多实例对象，所以需要这个参数来管理每个实例彼此各不相同的数据。

C++程序员会发现，Python的self参数与C++的this指针很相似。不过，在Python中，self一定要在程序代码中明确地写出：方法一定要通过self来取出或修改由当前方法调用或正在处理的实例的属性。这种让self明确化的本质是有意设计的：这个变量名存在，会让你明确脚本中使用的是实例属性名称，而不是本地作用域或全局作用域中的变量名。

例子

为了让这些概念更清晰，我们举个例子来说明。假设我们定义了下面的类。

```
class NextClass:                          # Define class
    def printer(self, text):              # Define method
        self.message = text               # Change instance
        print self.message                # Access instance
```

变量名printer引用了一个函数对象。因为这是在class语句的作用域中赋值的，就会变成类对象的属性，被每个由这个类创建的实例所继承。通常，因为像printer这类方法都是设计成处理实例的，所以我们得通过实例予以调用。

```
>>> x = NextClass()                       # Make instance

>>> x.printer('instance call')            # Call its method
instance call

>>> x.message                             # Instance changed
'instance call'
```

当通过对实例进行点号运算调用它时，printer会先通过继承将其定位，然后它的self参数会自动赋值为实例对象（x）。text参数会获得在调用时传入的字符串（'instance call'）。注意：因为Python会自动传递第一个参数给self，实际上只需传递一个参数。在printer中，变量名self是用于读取或设置每个实例的数据的，因为self引用的是当前正在处理的实例。

方法能通过实例或类本身两种方法其中的任意一种进行调用。例如，我们可以通过类的名称调用printer，只要明确地传递了一个实例给self参数。

```
>>> NextClass.printer(x, 'class call')    # Direct class call
class call

>>> x.message                             # Instance changed again
'class call'
```


www.TopSage.com

通过实例和类的调用具有相同的效果，只要在类形式中传递了相同的实例对象。实际上，在默认的情况下，如果尝试不带任何实例调用的方法时，就会得到错误信息。

```
>>> NextClass.printer('bad call')
TypeError: unbound method printer() must be called with NextClass instance...
```

调用超类的构造器

方法一般是通过实例调用的。不过，通过类调用方法也扮演了一些特殊的角色。常见的场景涉及到了构造器方法。就像所有属性__init__方法是由继承进行查找的。也就是说，在构造时，Python会找出并且只调用一个__init__。如果要保证子类的构造方法也会执行超类构造时的逻辑，一般都必须通过类明确地调用超类的__init__方法。

```
class Super:
    def __init__(self, x):
        ...default code...

class Sub(Super):
    def __init__(self, x, y):
        Super.__init__(self, x)              # Run superclass __init__
        ...custom code...                    # Do my init actions

I = Sub(1, 2)
```

这是代码有可能直接调用运算符重载方法的环境之一。如果真的想运行超类的构造方法，自然只能用这种方式进行调用：没有这样的调用，子类会完全取代超类的构造器。这门技术在实际中更现实的介绍，可以参见本章最后的例子（注3）。

其他方法调用的可能性

这种通过类调用方法的模式，是扩展继承方法行为（而不是完全取代）的一般基础。在第26章中，我们也会遇到Python 2.2新增的选项：静态方法，可让你编写不预期第一参数为实例对象的方法。这类方法可像简单的无实例的函数那样运作，其变量名属于其所在类的作用域。不过，这是高级的选用扩展功能。通常来说，你一定要为方法传入实例，无论通过实例还是类调用。

注3：有个相关的注意事项：你也可以在相同类中写几个__init__方法，但只会使用最后的定义，参考第25章以获得更多细节。

www.TopSage.com

继承

像class语句这样的命名空间工具的重点就是支持变量名继承。本节扩展了Python中关于属性继承的一些机制和角色。

在Python中，当对对象进行点号运算时，就会发生继承，而且涉及到了搜索属性定义树（一或多个命名空间）。每次使用object.attr形式的表达式时（object是实例或类对象），Python会从头至尾搜索命名空间树，先从对象开始，找到第一个attr为止。这包括在方法中对self属性的引用。因为树中较低的定义会覆盖较高的定义，继承构成了专有化的基础。

属性树的构造

图24-1总结命名空间树构造以及填入变量名的方式。通常来说：

- 实例属性是由对方法内self属性进行赋值运算而生成的。
- 类属性是通过class语句内的语句（赋值语句）而生成的。
- 超类的链接是通过class语句首行的括号内列出类而生成的。

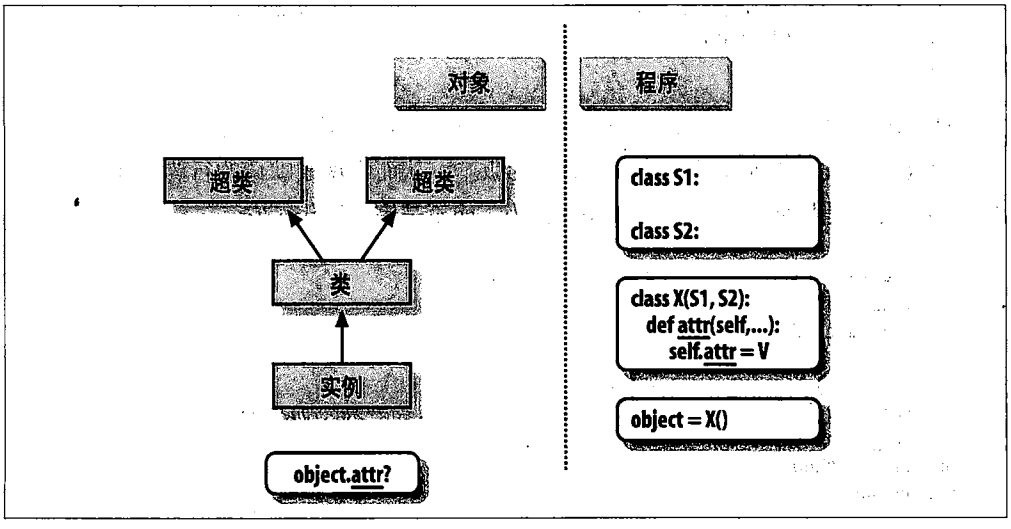


图24-1：程序代码会在内存中创建对象树，这个树是通过属性继承搜索的。调用类会创建记忆了这个类的新的实例。执行class语句会创建新的类，而列在class语句首行括号内的类则成为超类。即使self属性位于类的方法内每个属性引用都会触发由下至上的树搜索

结果就是属性命名空间树、连结实例、到产生它的类、再到类首行中所列出的所有超

www.TopSage.com

类。每次以点号运算从实例对象取出属性名称时，Python会向上搜索树，从实例直到超类（注4）。

继承方法的专有化

刚才谈到了继承树搜索模式，变成了将系统专有化的最好的方式。因为继承会先在子类寻找变量名，然后才查找超类，子类就可以对超类的属性重新定义来取代默认的行为。实际上，你可以把整个系统做成类的层次，再新增外部的子类来对其进行扩展，而不是在原处修改已经存在的逻辑。

重新定义继承变量名的概念引出了各种专有化技术。例如，子类可以完全取代继承的属性，提供超类可以找到的属性，并且通过已覆盖的方法回调超类来扩展超类的方法。我们已经看到过实际中取代的做法。下面是如何进行扩展的例子。

```
>>> class Super:
...     def method(self):
...         print 'in Super.method'
...
>>> class Sub(Super):
...     def method(self):                    # Override method
...         print 'starting Sub.method'      # Add actions here
...         Super.method(self)               # Run default action
...         print 'ending Sub.method'
...
```

直接调用超类方法是这里的重点。Sub类以其专有化的版本取代了Super的方法函数。但是，取代时，Sub又回调了Super所导出的版本，从而实现了默认的行为。换句话说，Sub.method只是扩展了Super.method的行为，而不是完全取代了它：

```
>>> x = Super()              # Make a Super instance
>>> x.method()               # Runs Super.method
in Super.method

>>> x = Sub()                # Make a Sub instance
>>> x.method()               # Runs Sub.method, which calls Super.method
starting Sub.method
in Super.method
ending Sub.method
```

这种扩展编码模式常常用于构造器方法。例如，参考之前的“方法”一节。

注4：这样的说明并不完整，因为我们也可以在class语句外，给对象做赋值来创建实例和类属性，但是，这么做比较少见，也易于出错（修改并非与class语句无关）。在Python中，默认所有的属性都是可读取的。我们会在第26章再谈变量名的私有性。

www.TopSage.com

类接口技术

扩展只是一种同超类接口的方式。下面所展示的*specialize.py*文件定义了多个类，示范了一些常用技巧。

Super
: 定义一个method函数以及一个delegate函数。

Inheritor
: 没有提供任何新的变量名，因此会获得Super中定义的一切内容。

Replacer
: 用自己的版本覆盖Super的method。

Extender
: 覆盖并回调默认method，从而定制Super的method。

Provider
: 实现Super的delegate方法预期的action方法。

研究这些子类来了解它们定制的共同的超类的不同途径。下面就是这个文件。

```
class Super:
    def method(self):
        print 'in Super.method'            # Default behavior
    def delegate(self):
        self.action()                      # Expected to be defined

class Inheritor(Super):                    # Inherit method verbatim
    pass

class Replacer(Super):                     # Replace method completely
    def method(self):
        print 'in Replacer.method'

class Extender(Super):                     # Extend method behavior
    def method(self):
        print 'starting Extender.method'
        Super.method(self)
        print 'ending Extender.method'

class Provider(Super):                     # Fill in a required method
    def action(self):
        print 'in Provider.action'

if __name__ == '__main__':
    for klass in (Inheritor, Replacer, Extender):
        print '\n' + klass.__name__ + '...'
        klass().method()
```


www.TopSage.com

```
print '\nProvider...'
x = Provider()
x.delegate()
```

有些事值得在这里讲一下。首先，这个例子末尾的自我测试程序代码会在for循环中建立三个不同类实例。因为类是对象，你可将它们放在元组中，并可以通过通用方式创建实例（稍后会再谈这个概念）。类也有特殊的__name__属性，就像模块。它默认为类首行中的类名称的字符串。以下是执行这个文件时的结果。

```
% python specialize.py

Inheritor...
in Super.method

Replacer...
in Replacer.method

Extender...
starting Extender.method
in Super.method
ending Extender.method

Provider...
in Provider.action
```

抽象超类

注意上一个例子中的Provider类是如何工作的。当通过Provider实例调用delegate方法时，有两个独立的继承搜索会发生：

1. 在最初x.delegate的调用中，Python会搜索Provider实例和它上层的对象，知道在Super中找到delegate的方法。实例x会像往常一样传递给这个方法的self参数。
2. 在Super.delegate方法中，self.action会对self以及它上层的对象启动新的独立继承搜索。因为self指的是Provider实例，就会找到Provider子类中的action方法。

这种“填空”的代码结构一般就是OOP的软件框架。至少，从delegate方法的角度来看，这个例子中的超类有时也称作是抽象类——也就是类的部分行为默认是由其子类所提供的。如果预期的方法没有在子类中定义，当继承搜索失败时，Python会引发未定义变量名的异常。类的编写者偶尔会使用assert语句，使这种子类需求更为明显，或者引发内置的异常NotImplementedError：

```
class Super:
    def method(self):
```


www.TopSage.com

```
        print 'in Super.method'
    def delegate(self):
        self.action()
    def action(self):
        assert 0, 'action must be defined!'
```

我们将会在第27章介绍assert。简而言之，如果其表达式运算结构为假，就会引发带有错误信息的异常。在这里，表达式总是为假（0）。因此，如果没有方法重新定义， 继承就会找到这里的版本，触发错误信息 。此外，有些类只在该类的不完整方法中直接产生NotImplemented异常。我们会在第27章介绍raise语句。

要查看这一节中的概念的更实际的例子，可参考第26章结尾的练习题8以及第6部分中的解答（在附录B）。这种分类法是介绍OOP的传统方式，但是多数开发人员的职务说明中已经或多或少的将它去掉了。

运算符重载

上一章中简短介绍了运算符重载。在这里，我们将会介绍更多的细节，以及一些其他常用的重载方法。以下是对重载的关键概念的复习：

- 运算符重载让类拦截常规的Python运算。
- 类可重载所有Python表达式运算符。
- 类可重载打印、函数调用、属性点号运算等运算。
- 重载使类实例的行为像内置类型。
- 重载是通过提供特殊名称的类方法来实现的。

让我们看一个简单的重载例子吧。如果类中提供了某些特殊名称的方法，当类实例出现在和运算有关的表达式的时候，Python就会自动调用这些方法。例如，下列文件*number.py*内的Number类提供一个方法来拦截实例的构造器（__init__），此外还有一个方法捕捉减法表达式（__sub__）。这种特殊的方法是钩子，可与内置运算相绑定。

```
class Number:
    def __init__(self, start):                  # On Number(start)
        self.data = start
    def __sub__(self, other):                   # On instance - other
        return Number(self.data - other)        # Result is a new instance

>>> from number import Number                   # Fetch class from module
>>> X = Number(5)                               # Number.__init__(X, 5)
>>> Y = X - 2                                   # Number.__sub__(X, 2)
>>> Y.data                                      # Y is new Number instance
3
```


www.TopSage.com

就像前边讨论过的一样，该代码中所见到的__init__构造器方法是Python中最常用的运算符重载方法，它存在于绝大多数类中。在本节中，我们会举例说明这个领域中其他一些可用的工具，并看一看这些工具常用的例程。

TopSage.com

常见的运算符重载方法

在类中，对内置对象（例如，整数和列表）所能做的事，几乎都有相应的特殊名称的重载方法。表24-1列出其中一些最常用的重载方法。事实上，很多重载方法有好几个版本（例如，加法就有__add__、__radd__和__iadd__）。参考其他Python书籍，或者Python语言参考手册，来了解完整的特殊方法名的清单。

表24-1：常见运算符重载方法

方法	重载	调用
__init__	构造器方法	对象建立：X = Class()
__del__	析构方法	对象收回
__add__	运算符+	X + Y, X += Y
__or__	运算符|（位OR）	X | Y, X |= Y
__repr__,__str__	打印，转换	print X、 repr(X)、 str(X)
__call__	函数调用	X()
__getattr__	点号运算	X.undefined
__setattr__	属性赋值语句	X.any = value
__getitem__	索引运算	X[key]， 没__iter__时的for循环和其他迭代器
__setitem__	索引赋值语句	X[key] = value
__len__	长度	len(X)，真值测试
__cmp__	比较	X == Y, X
__lt__	特定的比较	X < Y (or else __cmp__)
__eq__	特定的比较	X == Y (or else __cmp__)
__radd__	左侧加法 +	Noninstance + X
__iadd__	实地（增强的）加法	X += Y (or else __add__)
__iter__	迭代环境	用于循环、测试、理解、列表、映射及其他

所有重载方法的名称前后都有两个下划线字符，以便把同类中定义的变量名区别开来。特殊方法名称和表达式或运算的映射关系，是由Python语言预先定义好的（在标准语言

www.TopSage.com

手册中有说明）。例如，名称__add__按照Python语言的定义，无论__add__方法的代码实际在做些什么，总是对应到了表达式+。

所有运算符重载方法都是选用的：如果没有写某个方法，你的类就是不支持该运算（如果尝试使用该运算，就会引发异常）。多数重载方法只用在需要对象行为表现得就像内置类型一样的高级程序中。然而，__init__构造方法常出现在绝大多数类中。我们已见到过__init__初始设定构造方法，以及表24-1中一些其他的方法。让我们通过例子来说明表中的其他方法吧。

__getitem__拦截索引运算

__getitem__方法拦截实例的索引运算。当实例X出现在X[i]这样的索引运算中时，Python会调用这个实例继承的__getitem__方法（如果有的话），把X作为第一个参数传递，并且方括号内的索引值传给第二个参数。例如，下面的类将返回索引值的平方。

```
>>> class indexer:
...     def __getitem__(self, index):
...         return index ** 2
...
>>> X = indexer()
>>> X[2]                                # X[i] calls __getitem__(X, i).4
>>> for i in range(5):
...     print X[i],
...
0 1 4 9 16
```

__getitem__和__iter__实现迭代

初学者可能不见得马上就能领会这里的技巧，但这些技巧都是非常有用的。for语句的作用是从0到更大的索引值，重复对序列进行索引运算，直到检测到超出边界的异常。因此，__getitem__也可以是Python中一种重载迭代的方式。如果定义了这个方法，for循环每次循环时都会调用类的__getitem__，并持续搭配有更高的偏移值。这是一种“买一送一”的情况：任何会响应索引运算的内置或用户定义的对象，同样会响应迭代。

```
>>> class stepper:
...     def __getitem__(self, i):
...         return self.data[i]
...
>>> X = stepper()                       # X is a stepper object
>>> X.data = "Spam"
>>>
>>> X[1]                                # Indexing calls __getitem__
```


www.TopSage.com

```
'p'
>>> for item in X:                    # for loops call __getitem__
...     print item,                    # for indexes items 0..N
...
S p a m
```

事实上，这其实是“买一送一”的情况。任何支持for循环的类也会自动支持Python所有迭代环境，而其中多种环境我们已在前几章看过了（参考第13章的其他迭代环境）。例如，成员关系测试in、列表解析、内置函数map、列表和元组赋值运算以及类型构造方法也会自动调用__getitem__（如果定义了的话）。

```
>>> 'p' in X                          # All call __getitem__ too
1

>>> [c for c in X]                    # List comprehension
['S', 'p', 'a', 'm']

>>> map(None, X)                      # map calls
['S', 'p', 'a', 'm']

>>> (a, b, c, d) = X                  # Sequence assignments
>>> a, c, d
('S', 'a', 'm')

>>> list(X), tuple(X), ''.join(X)
(['S', 'p', 'a', 'm'], ('S', 'p', 'a', 'm'), 'Spam')

>>> X
<__main__.stepper instance at 0x00A8D5D0>
```

在实际应用中，这个技巧可用于建立提供序列接口的对象，并新增逻辑到内置的序列类型运算。第26章扩展内置类型时，我们会再谈这个观点。

用户定义的迭代器

如今，Python中所有的迭代环境都会先尝试__iter__方法，再尝试__getitem__。也就是说，它们宁愿使用第13章所学到的迭代协议，然后才是重复对对象进行索引运算。如果对象不支持迭代协议，就会尝试索引运算。

从技术角度来讲，迭代环境是通过调用内置函数iter去尝试寻找__iter__方法来实现的，而这种方法应该返回一个迭代器对象。如果已经提供了，Python就会重复调用这个迭代器对象的next方法，直到发生StopIteration异常。如果没找到这类__iter__方法，Python会改用__getitem__机制，就像之前那样通过偏移量重复索引，直到引发IndexError异常。

www.TopSage.com

在新机制中，类就是通过实现第13章和第17章介绍的迭代器协议（回头去看那几章以了解迭代器的背景细节），来实现用户定义的迭代器的。例如，下面的文件*iters.py*，定义了用户定义的迭代器类来生成平方值。

```
class Squares:
    def __init__(self, start, stop):  # Save state when created
        self.value = start - 1
        self.stop  = stop
    def __iter__(self):                # Get iterator object on iter()
        return self
    def next(self):                    # Return a square on each iteration
        if self.value == self.stop:
            raise StopIteration
        self.value += 1
        return self.value ** 2

% python
>>> from iters import Squares
>>> for i in Squares(1, 5):           # for calls iter(), which calls __iter__()
...     print i,                      # Each iteration calls next()
...
1 4 9 16 25
```

在这里，迭代器对象就是实例self，因为next方法是这个类的一部分。在较为复杂的场景中，迭代器对象可定义为个别的类或对象，有自己的状态信息，对相同数据支持多种迭代（下面会看到这种例子）。以Python raise语句发出信号表示迭代结束（本书下一部分会谈到引发异常的内容）。

__getitem__所写的等效代码可能不是很自然，因为for会对所有的0和较高值的偏移值进行迭代。传入的偏移值和所产生的值的范围只有间接的关系（0..N需要映射为start..stop）。因为__iter__对象会在调用过程中明确地保留状态信息。所以比__getitem__具有更好的通用性。

另外，有时__iter__迭代器会比__getitem__更复杂和难用。迭代器是用来迭代，不是随机的索引运算。事实上，迭代器根本没有重载索引表达式：

```
>>> X = Squares(1, 5)
>>> X[1]
AttributeError: Squares instance has no attribute '__getitem__'
```

__iter__机制也是我们在__getitem__中所见到的其他所有迭代环境的实现方式（成员关系测试、类型构造器、序列赋值运算等）。然而，和__getitem__不同的是，__iter__只循环一次，而不是循环多次。例如，Squares类只循环一次，循环之后就变为空。每次新的循环，都得创建一个新的迭代器对象。

www.TopSage.com

```
>>> X = Squares(1, 5)
>>> [n for n in X]                    # Exhausts items
[1, 4, 9, 16, 25]
>>> [n for n in X]                    # Now it's empty
[]
>>> [n for n in Squares(1, 5)]        # Make a new iterator object
[1, 4, 9, 16, 25]
>>> list(Squares(1, 3))
[1, 4, 9]
```

注意：如果用生成器函数编写（第17章介绍过迭代器相关的一个话题），这个例子可能更简单一些。

```
>>> from __future__ import generators      # Needed in Python 2.2, but not later
>>>
>>> def gsquares(start, stop):
...     for i in range(start, stop+1):
...         yield i ** 2
...
>>> for i in gsquares(1, 5):
...     print i,
...
1 4 9 16 25
```

和类不同的是，这个函数会自动在迭代中储存其状态。当然，这是假设的例子。实际上，可以跳过这两种技术，只用for循环、map或是列表解析，一次创建这个列表。在Python中，完成任务最佳而且是最快的方式通常也是最简单的方式：

```
>>> [x ** 2 for x in range(1, 6)]
[1, 4, 9, 16, 25]
```

然而，在模拟更复杂的迭代时，类会比较好用，特别是能够获益于状态信息和继承层次。下一节要探索这种情况下的使用例子。

有多个迭代器的对象

之前，提到过迭代器对象可以定义成一个独立的类，有其自己的状态信息，从而能够支持相同数据的多个迭代。考虑一下，当步进到字符串这类内置类型时，会发生什么事情。

```
>>> S = 'ace'
>>> for x in S:
...     for y in S:
...         print x + y,
...
aa ac ae ca cc ce ea ec ee
```

在这里，外层循环调用iter从字符串中取得迭代器，而每个嵌套的循环也做相同的事来

www.TopSage.com

获得独立的迭代器。因为每个激活状态下的迭代器都有自己的状态信息，不管其他激活状态下的循环是什么状态。每个循环可维持它在字符串内的位置信息。要用用户定义的迭代器达到相同效果，__iter__只需替迭代器定义新的状态对象，而不是返回self。

例如，下面定义了一个迭代器类，迭代时，跳过下一个元素。因为迭代器对象会在每次循环时都重新创建，所以能够支持多个处于激活状态下的循环。

```
class SkipIterator:
    def __init__(self, wrapped):
        self.wrapped = wrapped                    # Iterator state information
        self.offset  = 0
    def next(self):
        if self.offset >= len(self.wrapped):      # Terminate iterations
            raise StopIteration
        else:
            item = self.wrapped[self.offset]      # else return and skip
            self.offset += 2
            return item

class SkipObject:
    def __init__(self, wrapped):                  # Save item to be used
        self.wrapped = wrapped
    def __iter__(self):
        return SkipIterator(self.wrapped)         # New iterator each time

if __name__ == '__main__':
    alpha = 'abcdef'
    skipper = SkipObject(alpha)                   # Make container object
    I = iter(skipper)                             # Make an iterator on it
    print I.next(), I.next(), I.next()            # Visit offsets 0, 2, 4

    for x in skipper:                 # for calls __iter__ automatically
        for y in skipper:             # Nested fors call __iter__ again each time
            print x + y,              # Each iterator has its own state, offset
```

运行时，这个例子工作起来就像是对内置字符串进行嵌套循环一样，因为每个循环都会获得独立的迭代器对象来记录自己的状态信息；所以每个激活状态下的循环都有自己在字符串中的位置。

```
% python skipper.py
a c e
aa ac ae ca cc ce ea ec ee
```

作为对比，除非我们在嵌套循环中再次调用Squares来获得新的迭代对象，否则之前的Squares例子只支持一个激活状态下的迭代。在这里，只有SkipObject，但从该对象中创建了许多的迭代器对象。

就像往常一样，我们可以用内置工具达到类似的效果。例如，用第三参数边界值进行分片运算来跳过元素。

大家网
TopSage.com

```
>>> S = 'abcdef'
>>> for x in S[::2]:
...     for y in S[::2]:              # New objects on each iteration
...         print x + y,
...
aa ac ae ca cc ce ea ec ee
```

然而，这并不完全相同，主要有两个原因。首先，这里的每个分片表达式，实质上都是一次把结果列表储存在内存中；另一方面，迭代器则是一次产生一个值，这样使大型结果列表节省了实际的空间。其次，分片产生的新对象，其实我们没有对同一个对象进行多处的循环。更接近于类，我们需要事先创建一个独立的对象通过分片运算进行步进。

```
>>> S = 'abcdef'
>>> S = S[::2]
>>> S
'ace'
>>> for x in S:
...     for y in S:                   # Same object, new iterators
...         print x + y,
...
aa ac ae ca cc ce ea ec ee
```

这样与类的解决办法更相似一些，但是，它仍是一次性把分片结果储存在内存中（目前分片运算并没有生成器），并且只等效于这里跳过一个元素的特殊情况。

因为迭代器能够做类能做的任何事，所以它比这个例子所展示出来的更通用。无论我们的应用程序是否需要这种通用性，用户定义的迭代器都是强大的工具，可让我们把任意对象的外观和用法变得很像本书所遇到过的其他序列和可迭代对象。例如，我们可将这项技术在数据库对象中运用，通过迭代进行数据库的读取，让多个游标进入同一个查询结果。

__getattr__和__setattr__捕捉属性的引用

__getattr__方法是拦截属性点号运算。更确切地说，当通过对未定义（不存在）属性名称和实例进行点号运算时，就会用属性名称为字符串调用这个方法。如果Python可通过其继承树搜索流程找到这个属性，该方法就不会被调用。因为有这种情况，__getattr__可以作为钩子来通过通用的方式响应属性请求。例子如下。

```
>>> class empty:
...     def __getattr__(self, attrname):
...         if attrname == "age":
...             return 40
...         else:
...             raise AttributeError, attrname
...
```


www.TopSage.com

```
>>> X = empty()
>>> X.age
40
>>> X.name
...error text omitted...
AttributeError: name
```

在这里，empty类和其实例X本身并没有属性，所以对X.age的存取会转至__getattr__方法，self则赋值为实例（X），而attrname则赋值为未定义的属性名称字符串（"age"）。这个类传回一个实际值作为X.age点号表达式的结果（40），让age看起来像实际的属性。实际上，age变成了动态计算的属性。

对于类不知道该如何处理的属性而言，这个__getattr__会引发内置的AttributeError异常，告诉Python，那真的是未定义属性名。请求X.name时，会引发错误。当我们在后两章看到实际的委托和内容属性时，你会再看到__getattr__，而在第7部分会再介绍关于异常的更多细节。

有个相关的重载方法__setattr__会拦截所有属性的赋值语句。如果定义了这个方法，self.attr = value会变成self.__setattr__('attr', value)。这一点技巧性很高，因为在__setattr__中对任何self属性做赋值，都会再调用__setattr__，导致了无穷递归循环（最后就是堆栈溢出异常）。如果想使用这个方法，要确定是对属性字典做索引运算，来赋值任何实例属性的（下一节讨论）。使用self.__dict__['name'] = x，而不是self.name = x。

```
>>> class accesscontrol:
...     def __setattr__(self, attr, value):
...         if attr == 'age':
...             self.__dict__[attr] = value
...         else:
...             raise AttributeError, attr + ' not allowed'
...
>>> X = accesscontrol()
>>> X.age = 40                                    # Calls __setattr__
>>> X.age
40
>>> X.name = 'mel'
...text omitted...
AttributeError: name not allowed
```

这两个属性读取的重载方法，可控制或专有化对象属性的读取。这两个方法一般都在特定的场合扮演角色，我们会在本书后面介绍其中的一些内容。

www.TopSage.com

模拟实例属性的私有性

下列程序代码把上一个例子通用化了，让每个子类都有自己的私有变量名，这些变量名无法通过其实例进行赋值。

```
class PrivateExc(Exception): pass                    # More on exceptions later

class Privacy:
    def __setattr__(self, attrname, value):         # On self.attrname = value
        if attrname in self.privates:
            raise PrivateExc(attrname, self)
        else:
            self.__dict__[attrname] = value          # Self.attrname = value loops!

class Test1(Privacy):
    privates = ['age']

class Test2(Privacy):
    privates = ['name', 'pay']
    def __init__(self):
        self.__dict__['name'] = 'Tom'

x = Test1()
y = Test2()

x.name = 'Bob'
y.name = 'Sue'   # <== fails

y.age  = 30
x.age  = 40      # <== fails
```

实际上，这是Python中实现属性私有性（也就是无法在类外对属性名进行修改）的首选方法。虽然Python不支持`private`声明，但类似这种技术可以模拟器主要的目的。不过，这只是一部分的解决方案。为使其更有效，必须增强它的功能，让子类也能够设置私有属性，并且使用`__getattr__`和包装（有时称为代理）来检测对私有属性的读取。

完整的解法留给练习题，因为即使能以这种方式模拟私有性，但实际应用中几乎不会这么做。 不用`private`声明，Python程序员就可以编写大型的OOP软件框架和应用程序：这是一般意义上的读取控制的有趣的发现，偏离了我们的这里的介绍范围。

捕捉属性引用值和赋值，往往是很有用的技术。这可支持委托，也是一种设计技术，可以让控制器对象包裹内嵌的对象，增加新行为，并且把其他运算传回包装的对象（下一章会再谈委托和包装）。

`__repr__`和`__str__`会返回字符串表达形式

下一个例子是已经见过的`__init__`构造方法和`__add__`重载方法，本例也会定义返回实

www.TopSage.com

例的字符串表达形式的__repr__方法。字符串格式把self.data对象转换为字符串。如果定义了的话，当类的实例打印或转换成字符串时，__repr__（或其近亲__str__）就会自动调用。这些方法可替对象定义更好的显示格式，而不是使用默认的实例显示。

```
>>> class adder:
...     def __init__(self, value=0):
...         self.data = value                    # Initialize data
...     def __add__(self, other):
...         self.data += other                   # Add other in-place
...
>>> class addrepr(adder):                        # Inherit __init__, __add__
...     def __repr__(self):                      # Add string representation
...         return 'addrepr(%s)' % self.data     # Convert to string as code
...
>>> x = addrepr(2)                               # Runs __init__
>>> x + 1                                        # Runs __add__
>>> x                                            # Runs __repr__
addrepr(3)
>>> print x                                      # Runs __repr__
addrepr(3)
>>> str(x), repr(x)                              # Runs __repr__
('addrepr(3)', 'addrepr(3)')
```

那么，为什么要两个显示方法？概括地讲，是为了进行用户友好的显示，就像print语句和str内置函数一样，会先尝试__str__。__repr__方法从原则上应该返回一个字符串，这个字符串可以作为可执行代码去重建该对象，这是用于交互模式下提示回应以及repr函数的。如果__str__不存在，Python会转向__repr__（反过来就不会）。

```
>>> class addstr(adder):
...     def __str__(self):                       # __str__ but no __repr__
...         return '[Value: %s]' % self.data     # Convert to nice string
...
>>> x = addstr(3)
>>> x + 1
>>> x                                            # Default repr
<__main__.addstr instance at 0x00B35EF0>
>>> print x                                      # Runs __str__
[Value: 4]
>>> str(x), repr(x)
('[Value: 4]', '<__main__.addstr instance at 0x00B35EF0>')
```

正是由于这一点，如果想让所有环境都有统一的显示，__repr__是最佳选择。不过，分别定义这两个方法的话，就可在不同环境内支持不同显示。例如，终端用户显示使用__str__，而程序员在开发期间则使用底层的__repr__来显示。

```
>>> class addboth(adder):
...     def __str__(self):
...         return '[Value: %s]' % self.data     # User-friendly string
...     def __repr__(self):
```


www.TopSage.com

```
...         return 'addboth(%s)' % self.data      # As-code string
...
>>> x = addboth(4)
>>> x + 1
>>> x                                             # Runs __repr__
addboth(5)
>>> print x                                       # Runs __str__
[Value: 5]
>>> str(x), repr(x)
('[Value: 5]', 'addboth(5)')
```

在实际应用中，除了__init__以外，__str__（或其底层的近亲__repr__）似乎是Python脚本中第二个最常用的运算符重载方法。在任何时候，都可以打印对象而看见定制过的显示的时候，可能就是使用了这两个工具中的一个。

__radd__处理右侧加法

从严格意义上来讲，前边例子中出现的__add__方法并不支持+运算符右侧使用实例对象。要实现这类表达式，而支持可互换的运算符，可以一并编写__radd__方法。只有当+右侧的对象是类实例，而左边对象不是类实例时，Python才会调用__radd__。在其他所有情况下，则由左侧对象调用__add__方法。

```
>>> class Commuter:
...     def __init__(self, val):
...         self.val = val
...     def __add__(self, other):
...         print 'add', self.val, other
...     def __radd__(self, other):
...         print 'radd', self.val, other
...
>>> x = Commuter(88)
>>> y = Commuter(99)
>>> x + 1                          # __add__: instance + noninstance
add 88 1
>>> 1 + y                          # __radd__: noninstance + instance
radd 99 1
>>> x + y                          # __add__: instance + instance
add 88 <__main__.Commuter instance at 0x0086C3D8>
```

注意：__radd__中的顺序与之相反：self是在+的右侧，而other是在左侧。每个二元运算符都有类似的右侧重载方法（例如，__mul__和__rmul__）。一般而言，像__radd__这类右侧方法只在需要时才进行转换，而且会返回+来触发__add__（主逻辑写在这里）。此外，注意x和y是同一个类的实例。当不同类的实例混合出现在表达式时，Python优先选择左侧的那个类。

右侧方法是高级话题，在实际中很少使用。当你需要运算符具有互换性，而且只有当

www.TopSage.com

需要支持这类运算符时，才需要写这种方法。例如，Vector类会需要这些工具，但Employee或Button类可能就不需要。

__call__拦截调用

当实例调用时，使用__call__方法。不，这不是循环定义：如果定义了，Python就会为实例应用函数调用表达式运行__call__方法。这样可以让类实例的外观和用法类似于函数。

```
>>> class Prod:
...     def __init__(self, value):
...         self.value = value
...     def __call__(self, other):
...         return self.value * other
...
>>> x = Prod(2)
>>> x(3)
6
>>> x(4)
8
```

在这个例子中，__call__似乎没什么特别的意义。下面这个简单的方法也能提供类似功能。

```
>>> class Prod:
...     def __init__(self, value):
...         self.value = value
...     def comp(self, other):
...         return self.value * other
...
>>> x = Prod(3)
>>> x.comp(3)
9
>>> x.comp(4)
12
```

然而，当需要为函数的API偏写接口时，__call__就变得很有用：这可以编写遵循所需要的函数来调用接口对象，同时又能保留状态信息。事实上，这可能是除了__init__构造方法以及__str__和__repr__显示格式方法外，第三个最常用的运算符重载方法了。

函数接口和回调代码

作为例子，Tkinter GUI工具箱（本书稍后会谈到）可以把函数注册成事件处理器（也就是回调函数callback）。当事件发生时，Tkinter会调用已注册的对象。如果想让事件处理器保存事件之间的状态，可以注册类的绑定方法（bound method）或者遵循所需接口

www.TopSage.com

的实例（使用__call__）。在这一节的代码中，第二个例子中的x.comp和第一个例子中的x都可以用这种方式作为类似于函数的对象进行传递。

下一章会再介绍绑定方法，这里仅是举一个假设的__call__例子，应用于GUI领域。下列类定义了一个对象，支持函数调用接口，但是也有状态信息，可记住按钮稍后按下后应该变成什么颜色。

```
class Callback:
    def __init__(self, color):                  # Function + state information
        self.color = color
    def __call__(self):                         # Support calls with no arguments
        print 'turn', self.color
```

现在，在GUI环境中，即使这个GUI期待的事件处理器是无参数的简单函数，我们还是可以为按钮把这个类的实例注册成事件处理器。

```
cb1 = Callback('blue')                          # 'Remember' blue
cb2 = Callback('green')

B1 = Button(command=cb1)                        # Register handlers
B2 = Button(command=cb2)                        # Register handlers
```

当这个按钮按下时，会把实例对象当成简单的函数来调用，就像下面的调用一样。不过，因它把状态保留成实例的属性，所以知道应该做什么。

```
cb1()                                           # On events: prints 'blue'
cb2()                                           # Prints 'green'
```

实际上，这可能是Python语言中保留状态信息的最好的方式，比之前针对函数所讨论的技术更好（全局变量、嵌套函数作用域引用以及默认可变参数等）。利用OOP，状态的记忆是明确地使用属性赋值运算而实现的。

在继续之前，Python程序员偶尔还会用两种其他的方式，把信息和回调函数联系起来。其中一个选项是使用lambda函数的默认参数：

```
cb3 = (lambda color='red': 'turn ' + color)  # Or: defaults
print cb3()
```

另一种是使用类的绑定方法：这种对象记住了self实例以及所引用的函数，使其可以在稍后通过简单的函数调用而不需要实例来实现。

```
class Callback:
    def __init__(self, color):                  # Class with state information
        self.color = color
    def changeColor(self):                      # A normal named method
        print 'turn', self.color
```


www.TopSage.com

```
cb1 = Callback('blue')
cb2 = Callback('yellow')

B1 = Button(command=cb1.changeColor)          # Reference, but don't call
B2 = Button(command=cb2.changeColor)          # Remembers function+self
```

当按钮按下时，就好像是GUI这么做的，启用changeColor方法来处理对象的状态信息：

```
object = Callback('blue')
cb = object.changeColor                       # Registered event handler
cb()                                          # On event prints 'blue'
```

这种技巧较为简单，但是比起__call__重载调用而言就不通用了；同样地，有关绑定方法可参考下一章的内容。

在第26章你会看到另一个__call__例子，我们会通过它来实现所谓的函数装饰器的概念：它是可调用对象，在嵌入的函数上多加一层逻辑。因为__call__可让我们把状态信息附加在可调用对象上，自然而然地成为了被一个函数记住并调用了另一函数的实现技术。

__del__是析构器

每当实例产生时，就会调用__init__构造方法。每当实例空间被收回时（在垃圾收集时），它的对立面__del__，也就是析构方法（destructor method），就会自动执行。

```
>>> class Life:
...     def __init__(self, name='unknown'):
...         print 'Hello', name
...         self.name = name
...     def __del__(self):
...         print 'Goodbye', self.name
...
>>> brian = Life('Brian')
Hello Brian
>>> brian = 'loretta'
Goodbye Brian
```

在这里，当brian赋值为字符串时，我们会失去Life实例的最后一个引用。因此会触发其析构方法。这样行得通，可用于完成一些清理行为（例如，中断服务器的连接）。然而，基于某些原因，在Python中，析构方法不像其他OOP语言那么常用。

原因之一就是，因为Python在实例收回时，会自动收回该实例所拥有的所有空间，对

www.TopSage.com

于空间管理来说是不需要析构方法的（注5）。原因之二是无法轻易地预测实例何时收回，通常最好是在有意调用的方法中（或者try/finally语句，本书下一部分会说明）编写代码去终止活动。在某种情况下，系统表中可能还在引用该对象，使析构方法无法执行。

我们在篇幅内所能介绍的重载的例子就只有这么多了。其他大多数运算符重载方法的运行方式与我们讨论过的很相似，并且这些全都是钩子而已，是用来拦截内置类型运算的。例如，有些重载方法有独特的参数列表或传回值。本书后面内容会介绍一些其他方法，但是对于它的完整解释，只能参考其他的文档资源。

命名空间：完整的内容

现在，我们已谈过了类和实例对象，Python命名空间内容已经完整。作为学习参考，本书将用于解析变量名的所有规则在这里做个总结。首先要记住的是，点号和无点号的变量名，会用不同的方式处理，而有些作用域是用于对对象命名空间做初始设定的。

- 无点号运算的变量名（例如，X）与作用域相对应。
- 点号的属性名（例如，object.X）使用的是对象的命名空间。
- 有些作用域会对对象的命名空间进行初始化（模块和类）。

简单变量名：如果赋值就不是全局变量

无点号的简单变量名遵循第16章中的函数LEGB作用域法则，具体如下。

赋值语句 (X = value)

使变量名成为本地变量：在当前作用域内，创建或改变变量名X，除非声明它是全局变量。

引用 (X)

在当前作用域内搜索变量名X，之后是在任何以及所有的嵌套的函数中，然后是在当前的全局作用域中搜索，最后在内置作用域中搜索。

注5：在当前C实现的Python中，不需要在析构方法中关闭由实例打开的文件，因为那些文件在被收回时也会自动关闭。然而，就像第9章介绍的，最好明确地调用文件的关闭方法，因为在收回时自动关闭是最终的表现，而不是语言本身的特性（这种行为在Jython底下也许就有所不同）。

www.TopSage.com

属性名称：对象命名空间

点号的属性名指的是特定对象的属性，并且遵循模块和类的规则。就类和实例对象而言，引用规则增加了继承搜索这个流程。

赋值语句（object.X = value）

在进行点号运算的对象的命名空间内创建或修改属性名X，并没有其他作用。继承树的搜索只发生在属性引用时，而不是属性的赋值运算时。

引用（object.X）

就基于类的对象而言，会在对象内搜索属性名X，然后是其上所有可读取的类（使用继承搜索流程）。对于不是基于类的对象而言，例如，模块，则是从对象中直接读取X。

Python命名空间的“禅”：赋值将变量名分类

点号和无点号的变量名有不同的搜索流程，再加上两者都有多个搜索层次，有时很难看出变量名最终属于何处。在Python中，赋值变量名的场所相当重要：这完全决定了变量名所在的作用域或对象。文件*manynames.py*示范了这条原则是如何变成代码的，并总结了本书遇到的命名空间的概念。

```
# manynames.py

X = 11                      # Global (module) name/attribute (X, or manynames.X)

def f():
    print X                 # Access global X (11)

def g():
    X = 22                  # Local (function) variable (X, hides module X)
    print X

class C:
    X = 33                  # Class attribute (C.X)
    def m(self):
        X = 44              # Local variable in method (X)
        self.X = 55         # Instance attribute (instance.X)
```

这个文件分别五次给相同的变量名X赋值。不过，因为这个名称是在五个不同地方进行赋值的，这个程序中的五个X是完全不同的变量。从上至下，这里对X的赋值语句会产生：模块属性（11）、函数内的本地变量（22）、类属性（33）、方法中的本地变量（44）、以及实例属性（55）。虽然这五个都称为X，但事实上它们都是在原代码内的不同位置进行赋值的，或者说是赋值到了不同的对象，因此，使得这些变量名都是独特的变量。

www.TopSage.com

你应该花时间仔细研究一下这个例子，因为这个例子把本书前几部分探索过的概念都集合起来了。当你看懂时，就完成了Python命名空间的涅盘重生。当然，另一条达到涅盘状态的途径就是运行这个程序，看看运行结果是什么。以下是这个源代码的其余部分，也就是制作实例，打印其能读取的所有的X。

```
# manynames.py, continued

if __name__ == '__main__':
    print X                 # 11: module (a.k.a. manynames.X outside file)
    f()                     # 11: global
    g()                     # 22: local
    print X                 # 11: module name unchanged

    obj = C()               # Make instance
    print obj.X             # 33: class name inherited by instance

    obj.m()                 # Attach attribute name X to instance now
    print obj.X             # 55: instance
    print C.X               # 33: class (a.k.a. obj.X if no X in instance)

    #print C.m.X[26]        # FAILS: only visible in method
    #print f.X              # FAILS: only visible in function
```

文件执行时所打印的输出就在程序代码的注释中。可以跟踪这些输出来了解每次读取的变量X是哪一个。注意：可以通过类来读取其属性（C.X），但无法从def语句外读取函数或方法内的局部变量。局部变量对于在def内的其余代码才是可见的。而事实上，也只有当函数调用或方法执行时，才会存在于内存中。

这个文件定义的其中一些变量名可以让文件以外的其他模块看见。但是，回想一下，我们在另一个文件内读取这些变量名前，总是需要先进行导入。这就是模块的重点所在。

```
# otherfile.py

import manynames

X = 66
print X                     # 66: the global here
print manynames.X           # 11: globals become attributes after imports

manynames.f()               # 11: manynames's X, not the one here!
manynames.g()               # 22: local in other file's function

print manynames.C.X         # 33: attribute of class in other module
I = manynames.C()
print I.X                   # 33: still from class here
I.m()
print I.X                   # 55: now from instance!
```

注意：manynames.f()是怎样打印manynames中的X的，而不是打印本文件中赋值的X。

www.TopSage.com

作用域总是由源代码中的赋值语句位置来决定的（也就是语句），而且绝不会受到其导入关系的影响。此外，直到我们调用I.m()前实例的X都不会创建：属性就像是变量，在赋值之后才会存在，而不是在赋值前。在通常情况下，创建实例属性的方法是在类的__init__构造方法内进行赋值的，但这并不是唯一的选择。

当然，通常来说，在脚本内让每个变量不应该都使用相同的变量名！但是，就像这个例子所表示的那样，即使这么做，Python的命名空间还是会工作，防止在一个环境中所用的变量名无意中和另一个环境中所使用的变量名发生冲突。

命名空间字典

第19章中，我们学到了模块的命名空间实际上是以字典的形式实现的，并且可以由内置属性__dict__显示这一点。类和实例对象也是如此：属性点号运算其实内部就是字典的索引运算，而属性继承其实就是搜索连结的字典而已。实际上，实例和类对象就是Python中带有链接的字典而已。Python揭露这些字典，还有字典间的链接，以便于在高级角色中使用（例如，编码工具）。

为了了解Python内部属性的工作方式，我们通过交互模式会话加入类，来跟踪命名空间字典的增长的方式。首先，我们定义一个超类和子类，而它们的方法会在实例中保存数据。

```
>>> class super:
...     def hello(self):
...         self.data1 = 'spam'
...
>>> class sub(super):
...     def hola(self):
...         self.data2 = 'eggs'
...
```

当我们制作子类的实例时，该实例一开始会是空的命名空间字典，但是有链接会指向它的类，让继承搜索能顺着寻找。实际上，继承树可在特殊的属性中看到，你可以进行查看。实例中有个__class__属性连结到了它的类，而类有个__bases__属性，是一个元组，其中包含了通往更高的超类的链接。

```
>>> X = sub()
>>> X.__dict__
{ }

>>> X.__class__
<class __main__.sub at 0x00A48448>
```


www.TopSage.com

```
>>> sub.__bases__
(<class __main__.super at 0x00A3E1C8>,)

>>> super.__bases__
()
```

大家网 TopSage.com

当类为self属性赋值时，会填入实例对象。也就是说，属性最后会位于实例的属性命名空间字典内，而不是类的。实例对象的命名空间保存了数据，会随实例的不同而不同，而self正是进入其命名空间的钩子。

```
>>> Y = sub()

>>> X.hello()
>>> X.__dict__
{'data1': 'spam'}

>>> X.hola()
>>> X.__dict__
{'data1': 'spam', 'data2': 'eggs'}

>>> sub.__dict__
{'__module__': '__main__', '__doc__': None, 'hola': <function hola at
 0x00A47048>}

>>> super.__dict__
{'__module__': '__main__', 'hello': <function hello at 0x00A3C5A8>,
 '__doc__': None}

>>> sub.__dict__.keys(), super.__dict__.keys()
(['__module__', '__doc__', 'hola'], ['__module__', 'hello', '__doc__'])

>>> Y.__dict__
{ }
```

注意：类字典内的其他含有下划线变量名。Python会自动设置这些变量，它们中的大多数都不会在一般程序中使用到，但是有些工具会使用其中的一些变量（例如，__doc__控制第14章讨论过的的文档字符串）。

此外，Y是这些语句中创建的第二个实例，即使X的字典已由方法内的赋值语句做了填充，Y最后还是个空的命名空间字典。同样地，每个实例都有独立的命名空间字典，一开始是空的，可以记录和相同类的其他实例命名空间字典中的属性，完全不同的属性。

因为属性实际上是Python的字典键，其实有两种方式可以读取并对其进行赋值：通过点号运算或者通过键索引运算。

```
>>> X.data1, X.__dict__['data1']
('spam', 'spam')
```


www.TopSage.com

```
>>> X.data3 = 'toast'
>>> X.__dict__
{'data1': 'spam', 'data3': 'toast', 'data2': 'eggs'}

>>> X.__dict__['data3'] = 'ham'
>>> X.data3
'ham'
```

不过，这种等效关系只适用于实际中附加在实例上的属性。因为属性点号运算也会执行继承搜索，所以可以存取命名空间字典索引运算无法读取的属性。例如，继承的属性X.hello无法由X.__dict__['hello']读取。

最后，下面是在第4章和第14章介绍过的内置函数dir用在类和实例对象上的情况。这个函数能用在任何带有属性的对象上：dir(object)类似于object.__dict__.keys()调用。不过，dir会排序其列表并引入一些系统属性。在Python 2.2中，dir也会自动收集继承的属性（注6）。

```
>>> X.__dict__
{'data1': 'spam', 'data3': 'ham', 'data2': 'eggs'}
>>> X.__dict__.keys()
['data1', 'data3', 'data2']

>>>> dir(X)
['__doc__', '__module__', 'data1', 'data2', 'data3', 'hello', 'hola']
>>> dir(sub)
['__doc__', '__module__', 'hello', 'hola']
>>> dir(super)
['__doc__', '__module__', 'hello']
```

亲自动手实验一下这些特殊的属性，来了解命名空间实际上怎么处理它们储存的属性的。即使你绝不会在程序内使用这些特殊属性，但是知道它们只是普通的字典，也有助于弄清楚命名空间的一般概念。

命名空间链接

上一节介绍了“实例和类的特殊属性__class__和__bases__”，但是没有例子说明为什么留意这些属性。简而言之，这些属性可以在程序代码内查看继承层次。例如，可以用它们来显示类树，就像下面的例子所展示的那样。

注6：属性字典内容和dir调用结果会随时间而变。例如，因为Python现在可让内置类型也像类那样可制作子类，dir对内置类型的结果的内容会受到扩展，以包含运算符重载方法。一般而言，前后有双下划线的属性名称是直译器专属的属性。类型子类会在第26章再讨论。

www.TopSage.com

```
# classtree.py

def classtree(cls, indent):
    print '.'*indent, cls.__name__          # Print class name here
    for supercls in cls.__bases__:          # Recur to all superclasses
        classtree(supercls, indent+3)       # May visit super > once

def instancetree(inst):
    print 'Tree of', inst                   # Show instance
    classtree(inst.__class__, 3)            # Climb to its class

def selftest():
    class A: pass
    class B(A): pass
    class C(A): pass
    class D(B,C): pass
    class E: pass
    class F(D,E): pass
    instancetree(B())
    instancetree(F())

if __name__ == '__main__': selftest()
```

此脚本中的classtree函数是递归的：这会使用__name__打印类的名称，然后调用自身从而运行到超类。这样可让函数遍历任意形状的类树。递归会运行到顶端，然后在具有空的__bases__属性组超类停止。这个文件的大部分内容都是自我测试程序代码。独立执行时，会创建空的类树，从中制作两个实例，并打印其类树结构。

```
% python classtree.py
Tree of <__main__.B instance at 0x00ACB438>
... B
...... A
Tree of <__main__.F instance at 0x00AC4DA8>
... F
...... D
......... B
............ A
......... C
............ A
...... E
```

在这里，由点号所表示的缩进是用来代表类树的高度的。当然，也可以改进这种输出格式，也许还可以在GUI中显示出来。

可以在任何想很快得到类树显示的地方导入这些函数。

```
>>> class Emp: pass
...
>>> class Person(Emp): pass
...
>>> bob = Person()
```


www.TopSage.com

```
>>> import classtree
>>> classtree.instancetree(bob)
Tree of <__main__.Person instance at 0x00AD34E8>
... Person
...... Emp
```

大家网 TopSage.com

无论是否会编写或使用这类工具，这个例子示范了能够利用的多种特殊属性中的一种，而这些属性显示出解释器内部细节。当我们在第25章中编写通用属性列表类时，还会看到另一种形式。

一个更实际的例子

到目前为止，我们所看的大多数例子都是人为创造而且是独立完备的，其目的是为了帮助你把注意力集中在基础知识上。然而，本章的结尾是一个较大的例子，把我们所学的大多数概念都聚合在这里。这个例子几乎是需要自行研究的练习题：试着看这个例子的程序代码，来了解方法调用是如何解析的。

简而言之，下列模块*person.py*定义了三个类：

- GenericDisplay是混合类，提供了通用的__str__方法。对任何继承了它的类来说，这个方法会返回字符串，给出创建该实例的类的名称，以及实例内每个属性的“name=value”对。它使用__dict__属性命名空间字典替代了类实例中每个属性创建的“name=value”配对列表，并且使用实例内置的__class__中的内置__name__来确认类的名称。因为print语句会触发__str__，这个类的结果，会替这个类所衍生的所有实例都显示这个专有化打印格式。这是通用的工具。
- Person会记录人们的一般信息，提供两个处理方法来使用并修改实例对象的状态信息。此外，也会从其超类继承专有的打印格式逻辑。人物对象有两个属性和两个由这个类管理的方法。
- Employee是对Person定制的类，继承了读取姓氏和专有的打印格式方法。但是也增加了一个方法来实现加薪，并重新定义生日运算从而进行了定制（显然，员工要比其他人都老得快）。注意：超类构造方法如何手动启用的。我们需要执行上层的超类版本，来添加名字和年龄。

当你研究这个模块的代码时，你会看见每个实例都有自己的状态信息。注意继承是如何应用于混合并对行为定进行制，以及运算符重载是如何用于对实例进行初始化和打印实例的。

```
# person.py
```


www.TopSage.com

```
class GenericDisplay:
    def gatherAttrs(self):
        attrs = '\n'
        for key in self.__dict__:
            attrs += '\t%s=%s\n' % (key, self.__dict__[key])
        return attrs
    def __str__(self):
        return '<%s: %s>' % (self.__class__.__name__, self.gatherAttrs())

class Person(GenericDisplay):
    def __init__(self, name, age):
        self.name = name
        self.age  = age
    def lastName(self):
        return self.name.split()[-1]
    def birthDay(self):
        self.age += 1

class Employee(Person):
    def __init__(self, name, age, job=None, pay=0):
        Person.__init__(self, name, age)
        self.job  = job
        self.pay  = pay
    def birthDay(self):
        self.age += 2
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)

if __name__ == '__main__':
    bob = Person('Bob Smith', 40)
    print bob
    print bob.lastName()
    bob.birthDay()
    print bob

    sue = Employee('Sue Jones', 44, job='dev', pay=100000)
    print sue
    print sue.lastName()
    sue.birthDay()
    sue.giveRaise(.10)
    print sue
```

要测试此程序代码时，我们可以在交互模式下导入这个模块并创建实例。例如，下面是对Person类的运用。建立实例时会触发__init__，接着调用一些方法，来使用或修改实例状态信息（属性），并在打印实例时，运行继承的__str__从而打印了所有的属性。

```
>>> from person import Person
>>> ann = Person('Ann Smith', 45)
>>> ann.lastName()
'Smith'
>>> ann.birthDay()
>>> ann.age
46
```


www.TopSage.com

```
>>> print ann
<Person:
    age=46
    name=Ann Smith
>
```

最后，下面是文件自我测试逻辑（位于最后的代码，就用__name__测试下面的代码）的输出，它创建一个人物和一个员工，并对其进行修改。就像往常一样，当文件以顶层脚本执行时，这个自我测试程序代码才会运行，当作为库模块导入时则不会。注意员工是如何继承打印格式和姓氏提取的，有更多的状态信息，还有另一个方法用来加薪，而且可以执行专有化版本的生日方法（过生日老两岁）。

```
% python person.py
<Person:
    age=40
    name=Bob Smith
>
Smith
<Person:
    age=41
    name=Bob Smith
>
<Employee:
    job=dev
    pay=100000
    age=44
    name=Sue Jones
>
Jones
<Employee:
    job=dev
    pay=110000.0
    age=46
    name=Sue Jones
>
```

跟踪这个例子中的代码，从而了解这些输出所反映的方法调用。这个例子涵盖了Python中OOP机制的大多数概念。

现在，已经了解了Python类，可能意识到，这里所用的类几乎就像是函数包，嵌套附加在实例上的属性的内置对象，作为状态信息，并对其进行管理。例如，当lastName方法切割并进行索引运算时，只是在对类所管理的对象进行内置的字符串和列表处理运算。

运算符重载和继承（在类树中自动查找属性）是蓝图中主要的OOP工具。最后，这可以让类树底端的Employee获得得不少“免费”的行为，而这也是OOP内含主要概念。

www.TopSage.com

本章小结

本章对Python语言的OOP机制做更为深入的探索。我们学到更深入的类和方法、继承以及其他运算符重载方法。我们把Python命名空间内容扩展到类，使其更完整。学到这里，我们看过一些更高级的概念，例如，抽象超类、类数据属性以及对超类方法和构造方法手动调用。最后，我们研究一个较大的例子，把至今所学到的大部分OOP的概念都结合了起来。

现在，我们已经学会了Python中编写类的机制，下一章要转入的常见设计模式，也就是类常用和结合的方式，最优惠程序代码的重用。下一章有些题材并非局限于Python语言，但是，对于使用好类来说，可是相当重要的。不过继续学习之前，一定要做一下本章的习题，复习学过的内容。

www.TopSage.com

本章习题

1. 什么是抽象超类?
2. 类中有哪两个运算符重载方法可用于支持迭代?
3. 当简单赋值语句出现在class语句顶层时，会发生什么?
4. 类为什么可能会需要手动调用超类中的__init__方法?
5. 怎样增强（而不是完全取代）继承的方法?
6. 本章最后的例子中，当sue Employee实例打印时，哪些方法会运行?
7. Assyria 的首都…在哪?

习题解答

1. 抽象类是会调用方法的类，但没有继承或定义该方法，而是期待该方法由子类填补。当行为无法预测，非得等到更为具体的子类编写时才知道，通常可用这种方式把类通用化。OOP软件框架也使用这种方式做为客户端定义、可定制的运算的实现方法。
2. 类可定义（或继承）__getitem__或__iter__从而支持迭代。在所有迭代环境中，Python会先试着使用__iter__（返回的对象通过next方法支持迭代协议）：如果通过继承搜索没有找到__iter__，Python就会改用__getitem__索引方法（通过连续更高的索引值反复调用）。
3. 当简单赋值语句（X = Y）出现在类语句的顶层时，就会把数据属性附加在这个类上（Class.X）。就像所有的类属性，这会由所有的实例共享。不过，数据属性并不是可调用的方法函数。
4. 如果类定义自身的__init__构造方法，但是也必须启用超类的构建其代码，就必须手动调用超类的__init__方法。Python本身只会自动执行一个构造器方法：树中最低的那个。超类的构造器方法是通过类名称来调用，手动传入到self实例：Superclass.__init__(self, ...)。
5. 要增强继承的方法而不是完全替代，还得在子类中进行重新定义，但是要从子类的新版方法中，手动回调超类版本的这个方法。也就是，把self实例手动传给超类的版本的这个方法：Superclass.method(self, ...)。

www.TopSage.com

6. 打印sue最终会运行GenericDisplay.__str__方法和其所调用的GenericDisplay.gatherAttrs方法。更确切地讲，要打印sue时，print语句会将它传给内置的str函数，从而转换成用户友好的显示字符串。在类中，这也意谓着通过继承搜索来寻找__str__运算符重载方法，而如果找到时就会执行它。sue的类Employee并没有__str__方法，接着会搜索Person，而最终在GenericDisplay类中发现了__str__。

7. Ashur（或Qalat Sherqat）、Calah（或Nimrud）、短暂的Dur Sharrukin（或Khorsabad）以及最终的Nineveh。

www.TopSage.com

TopSage.com

第25章 类的设计

到目前为止，我们都将注意力集中在使用Python的OOP工具：类上。但是，OOP也有设计的问题，也就是如何使用类来对有用的对象进行建模。本章会介绍一些核心的OOP概念，以及一些额外的比目前展示过的例子更实际的例子。在这里会提到许多设计术语（委托、组合和工厂等），本书对这些术语进行了简单说明。如果想了解更多内容，建议查询一些OOP设计（或设计模式）方面的书籍。

Python和OOP

Python的OOP实现可以概括为三个概念，如下所示。

继承

继承是基于Python中的属性查找的（在`X.name`表达式中）。

多态

在`X.method`方法中，`method`的意义取决于`X`的类型（类）。

封装

方法和运算符实现行为，数据隐藏默认是一种惯例。

现在，你应该已经很好地掌握了Python中所谓的继承。本书也多次介绍了Python中的多态；这是因Python没有类型声明而出现的。因为属性总是在运行期解析，实现相同接口的对象是可互相交换的。客户端不需要知道实现它们调用的方法的对象种类。

封装指的是在Python中打包，也就是把实现的细节隐藏在对象接口之后。这并不代表有强制的私有性，就像第26章介绍的一样。封装可让对象接口的实现出现变动时，不影响这个对象的用户。

www.TopSage.com

通过调用标记进行重载（或不要）

有些OOP语言把多态定义成基于参数类型标记（type signature）的重载函数。但是，因为Python中没有类型声明，这种概念其实是行不通的。Python中的多态是基于对象接口的，而不是类型。

可以试一下通过参数列表进行重载方法，如下所示。

```
class C:
    def meth(self, x):
        ...
    def meth(self, x, y, z):
        ...
```

这样的代码是会执行的，但是，因为def只是在类的作用域中把对象赋值给变量名，这个方法函数的最后一个定义才是唯一保留的（就好像X = 1，然后X = 2，结果X将是2）。

基于类型的选择都能以第4章和第9章所见到过的类型测试的想法去编写代码，或者使用第16章的参数列表工具。

```
class C:
    def meth(self, *args):
        if len(args) == 1:
            ...
        elif type(arg[0]) == int:
            ...
```

不过，通常来讲，不应该这么做：就像第15章所描述的那样，应该把程序代码写成预期的对象接口，而不是特定的数据类型。这样一来，不论现在还是以后，都可在更多的类型和应用上使用。

```
class C:
    def meth(self, x):
        x.operation()          # Assume x does the right thing
```

通常来说，独特的运算使用独特的方法名称，不要依赖于调用标记，这样做才是更好的选择（无论使用的是哪种语言）。

类作为记录

第8章向我们展示了如何使用字典来记录程序中事物的内容属性的。让我们更详细地探索这方面内容。这里是之前使用过的基于字典的记录的例子。

www.TopSage.com

```
>>> rec = {}
>>> rec['name'] = 'mel'
>>> rec['age']  = 40
>>> rec['job']  = 'trainer/writer'
>>>
>>> print rec['name']
mel
```

这段代码模拟了其他语言中类似于记录和struct工具。不过，正如第23章所见到过的一样，还有其他许多方式能够做类似的事请。最简单如下所示。

```
>>> class rec: pass
...
>>> rec.name = 'mel'
>>> rec.age  = 40
>>> rec.job  = 'trainer/writer'
>>>
>>> print rec.age
40
```

这个程序代码的语法实质上比等价的字典版本少很多。这是使用空的class语句来产生空的命名空间对象（注意pass语句，从语法上讲，即使这个例子中没有逻辑要写，我们还是需要一个语句）。一旦创建好空类后，就可通过为类属性赋值来随时添加。

这样行得通，但是我们所需的每个记录，都需要新的class语句。也许更典型的情况是，我们可以改为产生空的类的实例，代表每个独特的事物。

```
>>> class rec: pass
...
>>> pers1 = rec()
>>> pers1.name = 'mel'
>>> pers1.job  = 'trainer'
>>> pers1.age  = 40
>>>
>>> pers2 = rec()
>>> pers2.name = 'dave'
>>> pers2.job  = 'developer'
>>>
>>> pers1.name, pers2.name
('mel', 'dave')
```

此处，用同一个类创建了两个记录：实例开始是空的，就像类一样。然后，用赋值属性来填写记录。不过，这一次有两个不同对象，因此有两个不同的name属性。事实上，相同类的实例不必拥有同一组属性名。在这个例子中，一个对象有独特的age变量名。实例其实是独立的命名空间：每个都有独立的属性字典。虽然它们一般都是由类方法用统一的方式填写的，但是，这比你预计的更加灵活。

最后，我们可能改为编写完整的类来实现记录。

www.TopSage.com

```
>>> class Person:
...     def __init__(self, name, job):
...         self.name = name
...         self.job  = job
...     def info(self):
...         return (self.name, self.job)
...
>>> mark = Person('ml', 'trainer')
>>> dave = Person('da', 'developer')
>>>
>>> mark.job, dave.info()
('trainer', ('da', 'developer'))
```

大家网 TopSage.com

这种机制也可以创建多个实例，但是，这次的类不是空的：在构造时我们新增了逻辑（方法）对实例进行初始化，并且属性收集到了一个元组。构造方法会设定name和job属性，使实例加强了一致性。

今后我们还可以增加逻辑来计算薪资、分解名字等，来进一步扩展这个代码（参考第24章最后关于这方面内容的例子）。最后，我们可能把此类连结到更大的层次，通过类自动属性搜索来继承现有的一组方法，或者使用Python对象的pickle机制把类实例存储在文件中，使其永久保存（边栏文章会再谈到pickle和永久保存，而本书后面也会再介绍这方面内容）。最后，虽然像字典这类类型使用起来很灵活，但类可让我们为对象增加不同方式的行为，这是内置类型和简单函数无法直接支持的。

类和继承："是一个"关系

本书已经深入探索了继承的机制，这里举个例子来说明它是如何用于模拟真实世界的关系的。从程序员的角度来看，继承是由属性点号运算启动的，由此触发实例、类以及任何超类中的变量名搜索。从设计师的角度来看，继承是一种定义集合成员关系的方式：类定义了一组内容属性，可由更具体的集合（子类）继承和定制。

为了说明，再看前面提到过的制作披萨的机器人的例子。假设我们决定探索另一条路径，开一家披萨餐厅。我们需要做的其中一件事就是聘请员工为顾客服务，准备食物等等。工程师是核心，我们决定创造一个机器人制作披萨，但是，为了让行政和网络关系正确，我们也决定把机器人做成有薪水的功能齐全的员工。

披萨店团队可以通过文件*employees.py*中的四个类来定义。最通用的类Employee提供共同行为，例如，加薪（giveRaise）和打印（__repr__）。员工有两种，所以Employee有两个子类：Chef和Server。这两个都会覆盖继承的work方法来打印更具体的信息。最后，我们的披萨机器人是由更具体的类来模拟：PizzaRobot是一种Chef，也是一种

www.TopSage.com

Employee。以OOP术语来看，我们称这些关系为“是一个”（is-a）链接：机器人是一个主厨，而主厨是一个员工。以下是*employees.py*文件。

```
class Employee:
    def __init__(self, name, salary=0):
        self.name   = name
        self.salary = salary
    def giveRaise(self, percent):
        self.salary = self.salary + (self.salary * percent)
    def work(self):
        print self.name, "does stuff"
    def __repr__(self):
        return "<Employee: name=%s, salary=%s>" % (self.name, self.salary)

class Chef(Employee):
    def __init__(self, name):
        Employee.__init__(self, name, 50000)
    def work(self):
        print self.name, "makes food"

class Server(Employee):
    def __init__(self, name):
        Employee.__init__(self, name, 40000)
    def work(self):
        print self.name, "interfaces with customer"

class PizzaRobot(Chef):
    def __init__(self, name):
        Chef.__init__(self, name)
    def work(self):
        print self.name, "makes pizza"

if __name__ == "__main__":
    bob = PizzaRobot('bob')          # Make a robot named bob
    print bob                        # Run inherited __repr__
    bob.work()                       # Run type-specific action
    bob.giveRaise(0.20)              # Give bob a 20% raise
    print bob; print

    for klass in Employee, Chef, Server, PizzaRobot:
        obj = klass(klass.__name__)
        obj.work()
```

当我们执行此模块中的自我测试代码时，会创建一个名为bob的制作披萨机器人，从三个类继承变量名：PizzaRobot、Chef以及Employee。例如，打印bob会执行Employee.__repr__方法，而给与bob加薪，则会运行Employee.giveRaise，因为会在这里继承找到这个方法。

```
C:\python\examples> python employees.py
<Employee: name=bob, salary=50000>
bob makes pizza
<Employee: name=bob, salary=60000.0>
```


www.TopSage.com

```
Employee does stuff
Chef makes food
Server interfaces with customer
PizzaRobot makes pizza
```

在这样的类的层次中，通常可以创建任何类的实例，而不只是底部的类。例如，这个模块中自我测试程序代码的for循环，创建了四个类的实例。要求工作时，每一个的反应都不同，因为work方法都各不相同。其实，这些类只是仿真真实世界的对象。work在这里只打印信息，稍后可以扩展它使其能够做实际的工作。

类和组合："有一个"关系

在第22章中介绍过组合的概念。从程序员的角度来看，组合涉及到把其他对象嵌入到容器对象内，并使其实现容器方法。对设计师来说，组合是另一种表示问题领域中的关系的方式。但是，组合不是集合的成员关系，而是组件，也就是整体的组成部分。

组合也反应了各组成部分之间的关系，通常称为"有一个"（has-a）关系。有些OOP设计书籍把组合称为聚合（aggregation），或者使用聚合指称容器和所含的物之间较弱的依赖关系，来区分这两个术语。本书中，"组合"就是指内嵌对象集合体。组合类一般都提供自己的接口，并通过内嵌的对象来实现接口。

现在，我们已经有了员工，把他们放到披萨店，开始忙吧。我们的披萨店是一个组合对象，有个烤炉，也有服务生和主厨这些员工。当顾客来店下单时，店里的组件就会开始行动：服务生接下订单，主厨制作披萨等等。下面的例子（文件*pizzashop.py*）模拟了这个场景中所有的对象和关系。

```
from employees import PizzaRobot, Server

class Customer:
    def __init__(self, name):
        self.name = name
    def order(self, server):
        print self.name, "orders from", server
    def pay(self, server):
        print self.name, "pays for item to", server

class Oven:
    def bake(self):
        print "oven bakes"

class PizzaShop:
    def __init__(self):
        self.server = Server('Pat')          # Embed other objects
        self.chef   = PizzaRobot('Bob')      # A robot named bob
        self.oven   = Oven()
```


www.TopSage.com

```
    def order(self, name):
        customer = Customer(name)           # Activate other objects
        customer.order(self.server)         # Customer orders from server
        self.chef.work()
        self.oven.bake()
        customer.pay(self.server)

if __name__ == "__main__":
    scene = PizzaShop()                     # Make the composite
    scene.order('Homer')                    # Simulate Homer's order
    print '...'
    scene.order('Shaggy')                   # Simulate Shaggy's order
```

PizzaShop类是容器和控制器，其构造器会创建上一节所编写的员工类实例并将其嵌入。此外，Oven类也是在这里定义的。当此模块的自我测试程序代码调用PizzaShop order方法时，内嵌对象会按照顺序进行工作。注意：每份订单创建了新的Customer对象，而且把内嵌的Server对象传给Customer方法。顾客是流动的，但是，服务生是披萨店的组合成分。另外，员工也涉及了继承关系，组合和继承是互补的工具。当执行这个模块时，我们的披萨店处理了两份订单：一份来自Homer，另一份来自Shaggy。

```
C:\python\examples> python pizzashop.py
Homer orders from <Employee: name=Pat, salary=40000>
Bob makes pizza
oven bakes
Homer pays for item to <Employee: name=Pat, salary=40000>
...
Shaggy orders from <Employee: name=Pat, salary=40000>
Bob makes pizza
oven bakes
Shaggy pays for item to <Employee: name=Pat, salary=40000>
```

同样地，这只是一个用来模拟的例子，但是，对象和交互足以代表组合的工作。其简明的原则就是，类可以表示任何用一句话表达的对象和关系。只要用类取代名词，用方法取代动词，就有第一手的设计方案了。

重访流处理器

就更为现实的组合范例而言，可以回忆第22章介绍OOP时，写的通用数据流处理器函数的部分代码。

```
def processor(reader, converter, writer):
while 1:
        data = reader.read()
        if not data: break
        data = converter(data)
        writer.write(data)
```


www.TopSage.com

在这里，不是使用简单函数，而是编写类，使用组合机制工作，来提供更强大的结构并支持继承。下面的文件*streams.py*示范了一种编写类的方式。

```
class Processor:
    def __init__(self, reader, writer):
        self.reader = reader
        self.writer = writer
    def process(self):
        while 1:
            data = self.reader.readline()
            if not data: break
            data = self.converter(data)
            self.writer.write(data)
    def converter(self, data):
        assert 0, 'converter must be defined'
```

以这种方式编写代码，读取器和写入器对象会内嵌在类实例当中（组合），而我们是在子类内提供转换器的逻辑，而不是传入一个转换器函数（继承）。文件*converters.py*显示了这种方法。

```
from streams import Processor

class Uppercase(Processor):
    def converter(self, data):
        return data.upper()

if __name__ == '__main__':
    import sys
    Uppercase(open('spam.txt'), sys.stdout).process()
```

在这里，Uppercase类继承了类处理的循环逻辑（以及其超类内所写的其他任何事情）。它只需定义其所特有的事件：数据转换逻辑。当这个文件执行时，会创建并执行实例，而该实例再从文件*spam.txt*中读取，把该文件对应的大写版本输出到stdout流。

```
C:\lp3e> type spam.txt
spam
Spam
SPAM!

C:\lp3e> python converters.py
SPAM
SPAM
SPAM!
```

要处理不同种类的流，可以把不同种类的对象传入类的构造调用中。在这里，我们使用了输出文件，而不是流。

```
C:\lp3e> python
>>> import converters
```


www.TopSage.com

```
>>> prog = converters.Uppercase(open('spam.txt'), open('spamup.txt', 'w'))
>>> prog.process()

C:\lp3e> type spamup.txt
SPAM
SPAM
SPAM!
```

但是，就像之前所建议的，我们可以传入包装在类中的任何对象（该对象定义了所需要的输入和输出方法接口）。以下是简单例子，传入写入器类（把文字嵌HTML标签中）。

```
C:\lp3e> python
>>> from converters import Uppercase
>>>
>>> class HTMLize:
...     def write(self, line):
...         print '<PRE>%s</PRE>' % line[:-1]
...
>>> Uppercase(open('spam.txt'), HTMLize()).process()
<PRE>SPAM</PRE>
<PRE>SPAM</PRE>
<PRE>SPAM!</PRE>
```

即使原始的Processor超类内的核心处理逻辑什么也不知道，如果跟随这个例子的控制流程，就会发现得到了大写转换（通过继承）以及HTML格式（通过组合）。处理代码只在意写入器的write方法，而且又定义一个名为convert的方法。并不在意这些调用在做什么。这种逻辑的多态和封装远超过类的威力。

Processor超类只提供文件扫描循环。在更实际的工作中，我们可能会对它进行扩充，使其子类能支持其他程序设计工具，而且在这个流程中，将其变为成熟的软件框架。在超类中编写一次这种工具，就可以在所有程序中重复使用。即使是这个简单的例子，因为类打包了不少东西，可通过继承，我们所需做的编写的就是HTML格式这一步。其余都是免费的。

看看组合的另一个例子，可以参考第26章结尾的练习题9以及其附录B的解法。这个例子类似于披萨店的例子。本书中把焦点放在继承上，因为这是Python语言本身提供的OOP的主要工具。但是，在实际中，组合和继承用的一样多，都是组织类结构的方式，尤其是在较大型系统中。正如我们所见的，继承和组合通常是互补的（偶尔是互换的）技术。不过，因为组合是设计的话题，不在Python语言和本书的范围内，这个话题的其他内容引以参考其他资源。

www.TopSage.com

大家网 TopSage.com

为什么要在意：类和持续性

本书这一部分内容中几次提到了pickle机制，因为它和类实例合起来使用效果很好。例如，除了可以模拟真实世界的交互外，在这里开发的披萨店类，也可以作为持续保存餐馆数据库的基础。类实例可以利用Python的pickle或shelve模块，通过单个步骤储存到磁盘上。对象的pickle接口很容易使用。

```
import pickle
object = someClass()
file   = open(filename, 'wb')     # Create external file
pickle.dump(object, file)         # Save object in file

import pickle
file   = open(filename, 'rb')
object = pickle.load(file)        # Fetch it back later
```

pickle机制把内存中的对象转换成序列化的字节流，可以保存在文件中，也可通过网络发送出去。解除pickle状态则是从字节流转换回同一个内存中的对象，Shelve也类似。但是它会自动把对象pickle生成按键读取的数据库，而此数据库会导出类似于字典的接口。

```
import shelve
object = someClass()
dbase  = shelve.open('filename')
dbase['key'] = object             # Save under key

import shelve
dbase  = shelve.open('filename')
object = dbase['key']             # Fetch it back later
```

上例中，使用类来模拟员工意味着我们只需做一点工作，就可以得到员工和商店的简单数据库：把这中实例对象pickle至文件，使其在Python程序执行时都能够永续保存。参考标准链接库手册和之后的范例，进一步了解关于pickle的内容。

OOP和委托

面向对象程序员时常会谈到所谓的委托 （delegation），通常就是指控制器对象内嵌其他对象，而把运算请求传给那些对象。控制器负责管理工作，例如，记录存取等。在Python中，委托通常是以__getattr__钩子方法实现的，因为这个方法会拦截对不存在属性的读取，包装类（代理类）可以使用__getattr__把任意读取转发给被包装的对象。包装类包有被包装对象的接口，而且自己也可以增加其他运算。

例如，考虑文件*trace.py*。

www.TopSage.com

```
class wrapper:
    def __init__(self, object):
        self.wrapped = object                    # Save object
    def __getattr__(self, attrname):
        print 'Trace:', attrname                 # Trace fetch
        return getattr(self.wrapped, attrname)   # Delegate fetch
```

回忆第24章，__getattr__会获得属性名称字符串。这个程序代码利用getattr内置函数，以变量名字符串从包裹对象取出属性：getattr(X,N)就像是X.N，只不过N是表达式，可在运行时计算出字符串，而不是变量。事实上，getattr(X,N)类似于X.__dict__[N]，但前者也会执行继承搜索，就像X.N，而getattr(X，N)则不会（参考第24章关于__dict__属性的内容）。

你可以使用这个模块包装类的做法，管理任何带有属性的对象的存取：列表、字典甚至是类和实例。在这里，wrapper类只是在每个属性读取时打印跟踪讯息，并把属性请求委托给嵌入的wrapped对象。

```
>>> from trace import wrapper
>>> x = wrapper([1,2,3])                    # Wrap a list
>>> x.append(4)                             # Delegate to list method
Trace: append
>>> x.wrapped                               # Print my member
[1, 2, 3, 4]

>>> x = wrapper({"a": 1, "b": 2})           # Wrap a dictionary
>>> x.keys()                                # Delegate to dictionary method
Trace: keys
['a', 'b']
```

实际效果就是以包装类内额外的代码增强被包装的对象的整个接口。我们可以利用这种方式记录方法调用，把方法调用转给其他或定制的逻辑等等。

第26章会重谈被包装对象和委托操作的概念，作为扩展内置类型的一种方式。如果你对委托设计模式感兴趣，也可以参看第26章有关函数装饰器的讨论：这是关联性很强的概念，其设计就是用于增加特定函数或方法调用，而不是对对象的整个接口。

多重继承

在class语句中，首行括号内可以列出一个以上的超类。当这么做时，就是在使用所谓的多重继承：类和其实例继承了列出的所有超类的变量名。

搜索属性时，Python会由左至右搜索类首行中的超类，直到找到相符者。从严格意义上来讲，搜索以深度优先法进行，一路走向继承树顶端，而且由左至右，因为任何超类本身可能还有一些其他的超类。

www.TopSage.com

通常意义上讲，多重继承是模拟属于一个集合以上的对象的好办法。例如，一个人可以是工程师、作家、音乐家等，因此，可继承这些集合的特性。

也许多重继承最常见的用法是作为“混合”超类的通用方法。这类超类一般都称为混合类：它们提供方法，你可通过继承将其加入应用类。例如，Python打印类实例对象的默认方式并不是很好用。

```
>>> class Spam:
...     def __init__(self):                    # No __repr__
...         self.data1 = "food"
...
>>> X = Spam()
>>> print X                                    # Default: class, address
<__main__.Spam instance at 0x00864818>
```

就像上一节见到过的运算符重载，你可以提供__repr__方法，以实现制定后的字符串表达形式。但是，如果不在每个你想打印的类中编写__repr__，为什么不在一个通用工具类中编写一次，然后在所有的类中继承呢？

这就是混合的用法。下列文件*mytools.py*定义一个混合类，名为Lister，重载了__repr__方法，让每个将Lister列在首行的类都可以受益。这个方法只是扫描实例的属性字典（记住，它是由__dict__导出的）从而创建字符串，显示所有实例的属性的名称和值。因为类是对象，Lister的格式逻辑适用于任何子类的实例，这是通用工具（注1）。

Lister使用两个特殊技巧来提取实例的类名称和地址。每个实例都有内置的__class__属性，引用了它所继承的类，而每个类都有__name__属性，引用了首行中的变量名，所以self.__class__.__name__是取出实例的类的名称。你可以调用内置id函数传回任意对象的地址（从定义上来讲，这就是唯一的对象识别码），从而获得实例的内存地址。

```
##########################################
# Lister can be mixed into any class to
# provide a formatted print of instances
# via inheritance of __repr__ coded here;
# self is the instance of the lowest class.
##########################################

class Lister:
   def __repr__(self):
       return ("<Instance of %s, address %s:\n%s>" %
                        (self.__class__.__name__,      # My class's name
                         id(self),                     # My address
                         self.attrnames()) )           # name=value list
```

注1：另一种做法可参考第24章例子中的*person.py*模块。这也会扫描属性命名空间字典，不过是假设不跳过双下划线的变量名。

www.TopSage.com

```
    def attrnames(self):
        result = ''
        for attr in self.__dict__.keys():                # Instance namespace dict
            if attr[:2] == '__':
                result = result + "\tname %s=<built-in>\n" % attr
            else:
                result = result + "\tname %s=%s\n" % (attr, self.__dict__ [attr])
        return result
```

从这个类衍生的实例会在打印时自动显示其属性，比起简单的地址，它提供了更多的信息。

```
>>> from mytools import Lister
>>> class Spam(Lister):
...     def __init__(self):
...         self.data1 = 'food'
...
>>> x = Spam()
>>> x
<Instance of Spam, address 8821568:
        name data1=food
>
```

Lister类对你写的任何类都有用：即使是已经有超类的类。这里就是多重继承方便之处：把Lister加至类首行的超类列表中（将其混合进来），就可免费获得其__repr__，同时依然继承现有的超类。文件*testmixin.py*做了示范。

```
from mytools import Lister              # Get tool class

class Super:
    def __init__(self):                 # superclass __init__
        self.data1 = "spam"

class Sub(Super, Lister):               # Mix in a __repr__
    def __init__(self):                 # Lister has access to self
        Super.__init__(self)
        self.data2 = "eggs"             # More instance attrs
        self.data3 = 42

if __name__ == "__main__":
    X = Sub()
    print X                             # Mixed-in repr
```

在这里，Sub从Super和Lister继承了变量名。它是自己的变量名和两个超类内的变量名的组合体。当创建Sub实例打印它的时候，就会自动取得Lister混合进来的制定后的表达形式。

```
C:\lp3e> python testmixin.py
<Instance of Sub, address 7833392:
        name data3=42
```

www.TopSage.com

```
        name data2=eggs
        name data1=spam
>
```

Lister可在其混入的任何类中工作，因为self引用的就是把Lister拉进来的子类的实例，无论它是什么。如果稍后你决定扩展Lister的__repr__，也要打印实例继承的所有类属性，那也很安全。因为这是继承的方法，修改Lister.__repr__会自动更新每个导入类，并将显示其混合进来的子类的情况（注2）。

总之，混合类相当于模块：可用在各种客户端的方法包。以下是Lister用在不同类实例上，采用单个继承模式的情况。

```
>>> from mytools import Lister
>>> class x(Lister):
...     pass
...
>>> t = x()
>>> t.a = 1; t.b = 2; t.c = 3
>>> t
<Instance of x, address 7797696:
        name b=2
        name a=1
        name c=3
>
```

OOP其实就是代码重用，而混合类是强大的工具。就像其他的程序设计一样，多重继承运用妥当时就是有力的工具。然而，实际上，这是高级的功能，如果用得过度或太随意，就会变得很复杂。我们会在下一章结尾时再介绍这个话题的陷阱。在下一章，我们也会碰到一种选择（新类），用来为特定的多重继承情况修改搜索的顺序。

类是对象：通用对象的工厂

类是对象，因此它很容易在程序中进行传递，保存在数据结构中。也可以把类传给会产生任意种类对象的函数。这类函数在OOP设计领域中偶尔称为工厂。这些函数是C++这

注2：如果对过程好奇，可以参看第24章的的相关内容。每个类都有__bases__内置属性，也就是该类的超类对象构成的元组。通用类层次的罗列工具或浏览器可以从实例的__class__遍历继承树走到它的类，然后再从类的__bases__递归地走到器所有超类，就像之前所示的例子*classtree.py*。在Python 2.2和后续版本中，甚至更为简单，因为内置dir函数现在自动包含了继承的属性名称。如果你不在乎树结构的显示，可以只扫描dir列表，而不是字典键列表，然后使用getattr按变量名字符串取出属性，而不是字典键索引运算。我们会在这一部分结尾练习题中再谈这个概念。

www.TopSage.com

类强类型语言的主要工作。但是在Python中进行实现，几乎是轻而易举的一件事。第17章介绍的apply函数和更新的替代语法，可以用一步调用带有任意构造方法参数的类，从而产生任意种类的实例（注3）。

```
def factory(aClass, *args):                    # varargs tuple
    return apply(aClass, args)                 # Call aClass, or: aClass(*args)

class Spam:
    def doit(self, message):
        print message

class Person:
    def __init__(self, name, job):
        self.name = name
        self.job  = job

object1 = factory(Spam)                        # Make a Spam object
object2 = factory(Person, "Guido", "guru")     # Make a Person object
```

在这段代码中，我们定义了一个对象生成器函数，称为factory。它预期传入的是类对象（任何对象都行），还有该类构造器的一个或多个参数。这个函数使用apply调用该函数并返回实例。

这个例子的其余部分只是定义了两个类，并将其传给factory函数以产生两者的实例。而这就是在Python中编写的工厂函数所需要做的事。它适用于任何类以及任何构造器参数。

可能的改进之处就是，在构造器调用中支持关键词参数。函数factory能够通过**args参数收集参数，当第三个参数传给apply时：

```
def factory(aClass, *args, **kwargs):          # +kwargs dict
    return apply(aClass, args, kwargs)         # Call aClass
```

现在，你应该知道，在Python中一切都是“对象”，包括类（类在C++这类语言中仅仅是编译器的输入而已）。然而，就像第6部分一开始所说的，只有从类衍生的对象才是Python中的OOP对象。

注3：其实，apply可以调用任何可调用的对象，包括函数、类和方法。这里的factory函数也会运行任何可调用的对象，而不仅仅是类（尽管参数名称是这样）。此外，注意到，最近的Python版本中，通常都更倾向于使用aClass(*args)调用语法而不是apply(aClass, args)内置调用。

www.TopSage.com

为什么有工厂

工厂函数有什么优势（除了作为本书示范类对象的原因外）呢？可惜，如果没有列出超出篇幅以外的代码的话，是很难展现出这种设计模式的应用的。不过，一般而言，这类工厂可以将代码和动态配置对象的构造细节隔离开。

例如，回想第22章以抽象方式介绍的例子processor，以及本章中再次作为“有一个”关系的组合的例子。这个程序接受读取器和写入器对象来处理任意的数据流。

这个例子的原始版本可以手动传入特定的类的实例，例如，FileWriter和SocketReader，来调整正被处理的数据流。稍后，我们传入硬编码的文件、流以及格式对象。在更为动态的场合下，像配置文件或GUI这类外部工具可能用来配置流。

在这种动态世界中，我们可能无法在脚本中把流的接口对象的建立方式固定地编写好。但是有可能根据配置文件的内容在运行期间创建它。

例如，这个文件可能会提供从模块导入的流的类的字符串名称，以及选用构造器的调用参数。工厂式的函数或程序代码在这里可能很方便，因为它们可以让我们取出并传入没有预先在程序中硬编码的类。实际上，这些类在编写程序时可能还不存在。

```
classname = ...parse from config file...
classarg  = ...parse from config file...

import streamtypes                          # Customizable code
aclass = getattr(streamtypes, classname)    # Fetch from module
reader = factory(aclass, classarg)          # Or aclass(classarg)
processor(reader, ...)
```

在这里，getattr内置函数依然用于取出特定字符串名称的模块属性（很像obj.attr，但attr是字符串）。因为这个程序代码片段是假设的单独的构造器参数，因此并不见得需要factory或apply：我们能够使用aclass(classarg)直接创建其实例。然而，存在未知的参数列表时，它们可能就更有用了，而通用的工厂编码模式可以改进代码的灵活性。有关这个话题的更多细节，请参考有关OOP设计和设计模式的书籍。

方法是对象：绑定或无绑定

方法也是一种对象，很像函数。类方法能由实例或类来读取，实际上在Python中就有两种方式，如下所示。

www.TopSage.com

无绑定类方法对象：无self

通过对类进行点号运算从而获取类的函数属性，会传回无绑定（unboud）方法对象。调用该方法时，必须明确提供实例对象作为第一个参数。

绑定实例方法对象：self+函数对

通过对实例进行全运算从而获取类的函数属性，会传回绑定（bound）方法对象。Python在绑定方法对象中自动把实例和函数打包，所以，不用传递实例去调用该方法。

这两种方法都是功能齐全的对象，可四处传递，保持在列表内等。执行时，两者需要在第一参数的实例（也就是self的值）。这也就是为什么我们上一章中在子类方法调用超类方法时，要刻意传入实例的原因。从严格意义上来说，这类调用会产生无绑定方法对象。

调用绑定方法对象时，Python会自动提供实例，来创建绑定方法对象的实例。也就是说，绑定方法对象通常都可和简单函数对象互换，而且对于原本就是针对函数而编写的接口而言，就相当有用了（参考边栏文章的例子）。

为了解释清楚，假设我们定义下面的类。

```
class Spam:
    def doit(self, message):
        print message
```

现在，在正常操作中，创建了一个实例，以一步调用了它的方法，从而打印出传入的参数：

```
object1 = Spam()
object1.doit('hello world')
```

不过，其实，绑定方法对象是在过程中产生的，就在方法调用的括号前。事实上，我们可以获取绑定方法，而不用实际进行调用。object.name点号结合运算是一个对象表达式。在下列代码中，会传回绑定方法对象，把实例（object1）和方法函数（Spam.doit）打包起来。我们可以把这个绑定方法赋值给另一个变量名，然后像简单函数那样进行调用。

```
object1 = Spam()
x = object1.doit          # Bound method object: instance+function
x('hello world')          # Same effect as object1.doit('...')
```

另一方面，如果对类进行点号运算来获得doit，就会得到无绑定方法对象，也就是函数对象的引用值。要调用这类方法时，必须传入实例作为最左侧参数。

www.TopSage.com

```
object1 = Spam()
t = Spam.doit            # Unbound method object
t(object1, 'howdy')      # Pass in instance
```

扩展一下，如果我们引用的self的属性是引用类中的函数，相同规则也适用于类的方法。self.method表达式是绑定方法对象，因为self是实例对象。

```
class Eggs:
    def m1(self, n):
        print n
    def m2(self):
        x = self.m1      # Another bound method object
        x(42)            # Looks like a simple function

Eggs().m2()              # Prints 42
```

大多数时候，通过点号运算取出方法后，就是立即调用，所以你不会注意到这个过程中产生的方法对象。但是，如果开始编写以通用方式调用对象的程序代码时，就得消息了，特别要注意无绑定方法：无绑定方法一般需要传入明确的实例对象（注4）。

现在，了解了方法对象模型，要看其他绑定方法的范例，可参考本章边栏文章，以及上一章中有关回调函处理器的讨论。

重访文档字符串

文档字符串（第14章详细探讨过）是字符串常量，在各种结构的顶端显示，由Python自动储存在相应对象的__doc__属性内。模块文件、函数def以及类和方法都能够使用。现在，对类和方法了解的都深入了，*docstr.py*文件是完整的例子，总结了代码中文档字符串出现的场合。这些都可以是三引号代码块。

```
"I am: docstr.__doc__"

class spam:
    "I am: spam.__doc__ or docstr.spam.__doc__"

    def method(self, arg):
        "I am: spam.method.__doc__ or self.method.__doc__"
        pass

def func(args):
    "I am: docstr.func.__doc__"
    pass
```

注4：参考第26章有关静态和类方法的讨论，从而了解这个规则的例外。就像绑定方法一样，两者都可以冒充基本的函数，因为它们在调用时不需要实例。

www.TopSage.com

大家网 TopSage.com

为什么要在意：绑定方法和回调函数

因为绑定方法会自动让实例和类方法函数配对，可以在任何希望得到简单函数的地方使用。最常见的使用，就是把方法注册成Tkinter GUI接口中事件回调处理器的代码。下面是一个简单的例子。

```
def handler():
    ...use globals for state...
...
widget = Button(text='spam', command=handler)
```

要为按钮点击事件注册一个处理器时，通常是传递一个不带参数的可调用对象给command关键词参数。函数名（和lambda）都可以使用，而类方法只要是绑定方法也可以使用。

```
class MyWidget:
    def handler(self):
        ...use self.attr for state...
    def makewidgets(self):
        b = Button(text='spam', command=self.handler)
```

在这里，事件处理器是self.handler：绑定方法对象，它记住self和MyGui.handler。因为handler稍后因事件而启用时，self会引用原始实例。因此这个方法可以读取在事件间用于保留状态信息的实例的属性。如果利用简单函数，状态信息一般都必须通过全局变量保存。此外，也可参考第24章有关__call__运算符重载的讨论，来了解另一种让类和函数API相容的方式。

说明文档字符串的主要优点是运行期间也会存在。因此，写成文档字符串时，就可以对一个对象和它的__doc__属性使用点号运算，来获得其文档。

```
>>> import docstr
>>> docstr.__doc__
'I am: docstr.__doc__'

>>> docstr.spam.__doc__
'I am: spam.__doc__ or docstr.spam.__doc__'

>>> docstr.spam.method.__doc__
'I am: spam.method.__doc__ or self.method.__doc__'

>>> docstr.func.__doc__
'I am: docstr.func.__doc__'
```

第14章讨论过的PyDoc工具，知道如何将这些字符串格式化为报告。

文档字符串可以在运行时使用，但是，语法上的灵活性却不如#注释（它可以出现在程

www.TopSage.com

序中任何地方）。两种形式都是有用的工具，而任何程序的文档都是有益的（只要足够准确）。

类和模块

本章最后要简单比较本书前两部分的话题：模块和类。因为它们都是命名空间，其区别会让人感到困惑。简而言之。

模块

- 是数据/逻辑套件。
- 由Python文件或C扩展编写而成。
- 通过导入使用。

类

- 实现新的对象。
- 由class语句创建。
- 通过调用使用。
- 总是存在于模块中。

类也支持其他模块不支持的功能，例如，运算符重载、产生多个实例以及继承。虽然类和模块都是命名空间，现在你应该看得出来，它们其实区别是很大的。

本章小结

本章中，我们列举了使用和混合类的常见方式，使其重用性和优点得以最优化：这些通常被认为是设计的话题，通常和具体的程序语言无关（不过Python让实现变得更为容易）。我们研究过委托（把对象包装在代理类内）、组合（控制嵌入的对象）、继承（从其他类中获取行为）以及一些比较少见的概念。例如，多重继承、绑定方法以及工厂函数。

下一章要研究更高级的类的相关话题，来结束我们对类和OOP的讨论。与编写应用的程序员相比，其中有些题材对工具编写者更有用处，不过大多数要使用Python做OOP的人，还是值得看一看的。下面首先做一下本章的习题。

www.TopSage.com

本章习题

1. 什么是多重继承？
2. 什么是委托？
3. 什么是组合？
4. 什么是绑定方法？

习题解答

1. 当类从一个以上超类继承时，就发生了多重继承。把多个类代码的包混合在一起是十分有用的。
2. 委托涉及到把对象包装在代理类中，这样代理类会增加额外的行为，而把其他运算传给被包装的对象。代理类包含了被包装的对象的接口。
3. 组合是一种技术，让控制器类嵌入和引导一群对象，并自行提供接口。这是利用类创建较大结构的方式。
4. 绑定方法结合实例和方法函数；调用时，不用刻意传入实例对象，因为原始的实例依然可用。

www.TopSage.com

大家网
TopSage.com

第26章

类的高级主题

本章将介绍一些与类相关的高级主题，作为第6部分和Python OOP讨论的结束：我们要研究如何建立内置类型的子类、伪私有属性、新式类、静态方法和函数装饰器等。

正像我们见到的那样，Python的OOP模型核心非常简单，而本章所介绍的是一些高级主题，而且是可选的，因此在Python应用程序设计中，不会经常遇到。不过，出于完整的考虑，我们还是简单看一看这些用于高级OOP工作的高级工具，来结束类的讨论。

就像往常一样，因为这是本书这一部分的最后一章，最后一节是介绍类与相关陷阱，还有这一部分的实验练习题。鼓励读者做一下练习题，来强化这里所学到的概念。也建议读者从事或研究较大的Python OOP项目，作为本书的补充。计算机领域一向如此，通过实践OOP的优点会越来越明显。

扩展内置类型

除了实现新的种类的对象以外，类偶尔也用于扩展Python的内置类型的功能，从而支持更另类的数据结构。例如，要为列表增加队列插入和删除方法，你可以写些类，包装（嵌入）列表对象，然后导出能够以特殊方式处理该列表的插入和删除的方法，同第25章所学习过的委托技术。在Python 2.2时，你也可以使用继承把内置类型专有化。下面的两节会说明这两种技术。

通过嵌入扩展类型

还记得我们在第4部分所写的那些集合函数吗？下面是以Python类的形式重生的样子。下面的例子（文件*setwrapper.py*）把一些集合函数变成方法，而且新增了一些基本运算符重载，实现了新的集合对象。对于多数类而言，这个类只是包装了Python列表，以及附加的集合运算。因为这是类，所以也支持多个实例和子类继承的定制。

www.TopSage.com

```
class Set:
    def __init__(self, value = []):    # Constructor
        self.data = []                  # Manages a list
        self.concat(value)

    def intersect(self, other):         # other is any sequence
        res = []                        # self is the subject
        for x in self.data:
            if x in other:              # Pick common items
                res.append(x)
        return Set(res)                 # Return a new Set

    def union(self, other):             # other is any sequence
        res = self.data[:]              # Copy of my list
        for x in other:                 # Add items in other
            if not x in res:
                res.append(x)
        return Set(res)

    def concat(self, value):            # value: list, Set...
        for x in value:                 # Removes duplicates
           if not x in self.data:
                self.data.append(x)

    def __len__(self):          return len(self.data)        # len(self)
    def __getitem__(self, key): return self.data[key]        # self[i]
    def __and__(self, other):   return self.intersect(other) # self & other
    def __or__(self, other):    return self.union(other)     # self | other
    def __repr__(self):         return 'Set:' + `self.data` # Print
```

重载索引运算让Set类的实例可以充当真正的列表。本章结尾的练习题中会碰见这个类并扩展它，在附录B中将进一步解释这些代码。

通过子类扩展类型

从Python 2.2起，所有内置类型现在都能直接创建子类。像list、str、dict以及tuple这些类型转换函数都变成内置类型的名称：虽然脚本看不见，但类型转换调用［例如，list('spam')］其实是启用了类型的对象构造方法。

这样的改变让你可以通过用户定义的class语句，定制或扩展内置类型的行为：建立类型名称的子类并对其进行定制。类型的子类实例，可用在原始的内置类型能够出现的任何地方。例如，假设你对Python列表偏移值以0开始计算而不是1开始一直很困扰，不用担心，你可以编写自己的子类，定制列表的核心行为。文件*typesubclass.py*说明了如何去做。

```
# Subclass built-in list type/class.
# Map 1..N to 0..N-1; call back to built-in version.
```


www.TopSage.com

```
class MyList(list):
    def __getitem__(self, offset):
        print '(indexing %s at %s)' % (self, offset)
        return list.__getitem__(self, offset - 1)

if __name__ == '__main__':
    print list('abc')
    x = MyList('abc')              # __init__ inherited from list
    print x                        # __repr__ inherited from list

    print x[1]                     # MyList.__getitem__
    print x[3]                     # Customizes list superclass method

    x.append('spam'); print x      # Attributes from list superclass
    x.reverse();      print x
```

在这个文件中，MyList子类扩展了内置list类型的__getitem__索引运算方法，把索引1到N映射至实际的0到N−1。它所做的其实就是把提交的索引值减1，之后继续调用超类版本的索引运算，但是，这样做足够了。

```
% python typesubclass.py
['a', 'b', 'c']
['a', 'b', 'c']
(indexing ['a', 'b', 'c'] at 1)
a
(indexing ['a', 'b', 'c'] at 3)
c
['a', 'b', 'c', 'spam']
['spam', 'c', 'b', 'a']
```

此输出包括打印类索引运算的过程。像这样改变索引运算是否是好事是另一个话题：MyList类的使用者，对于这种和Python序列行为有所偏离的困惑程度可能也都不同。一般来说，用这种方式定制内置类型，可说是很强大的工具。

例如，这样的编码模式会产生编写集合的另一种方式：作为内置list类型的子类，而不是管理内嵌列表对象的独立类。下面的类位于文件*setsubclass.py*内，通过定制list来增加和集合处理相关的方法和运算符。因为其他所有行为都是从内置list超类继承而来的，这样可以得到较短和较简单的替代做法。

```
class Set(list):
    def __init__(self, value = []):     # Constructor
        list.__init__([])                # Customizes list
        self.concat(value)               # Copies mutable defaults

    def intersect(self, other):          # other is any sequence
        res = []                         # self is the subject
        for x in self:
            if x in other:               # Pick common items
                res.append(x)
        return Set(res)                  # Return a new Set
```


www.TopSage.com

大家网
TopSage.com

```
    def union(self, other):                 # other is any sequence
        res = Set(self)                     # Copy me and my list
        res.concat(other)
        return res

    def concat(self, value):                # value: list, Set . . .
        for x in value:                     # Removes duplicates
            if not x in self:
                self.append(x)

    def __and__(self, other): return self.intersect(other)
    def __or__(self, other):  return self.union(other)
    def __repr__(self):       return 'Set:' + list.__repr__(self)

if __name__ == '__main__':
    x = Set([1,3,5,7])
    y = Set([2,1,4,5,6])
    print x, y, len(x)
    print x.intersect(y), y.union(x)
    print x & y, x | y
    x.reverse(); print x
```

以下是文件末尾自我测试代码的输出。因为创建核心类型的子类是高级功能，在这里要省略其他的细节，建议参看程序代码的结果来研究其行为。

```
% python setsubclass.py
Set:[1, 3, 5, 7] Set:[2, 1, 4, 5, 6] 4
Set:[1, 5] Set:[2, 1, 4, 5, 6, 3, 7]
Set:[1, 5] Set:[1, 3, 5, 7, 2, 4, 6]
Set:[7, 5, 3, 1]
```

Python中还有更有效率的方式，也就是通过字典实现集合：把这里的集合实现中的线性扫描换成字典索引运算（哈希），因此运行时会快很多（相关细节，可参考《Programming Python》）。如果你对集合感兴趣，也可以看一看第5章探索过的集合对象类型。这种类型将集合运算作为内置工具。实验集合的实现很有趣，但是在如今的Python中，已经不再有那么迫切的需要了。

有关另一个类型子类的例子，可参考Python 2.3中新的bool类型的实现。就像本书之前所提到的，bool是int的子类，有两个实例（True和False），行为就像整数1和0，但是继承了定制后的字符串表达方法来显示其变量名。

类的伪私有属性

在第4部分中，我们学到了每个在模块文件顶层赋值的变量名都会导出。在默认情况下，类也是这样：数据隐藏是一个惯例，客户端可以读取或修改任何它们想要的类或实

www.TopSage.com

例的属性。事实上，用C++术语来讲，属性都是“public”和“virtual”，在任意地方都可进行读取，并且在运行时进行动态查找（注1）。

如今依然如此。然而，Python也支持变量名压缩（mangling，相当于扩张）的概念，让类内某些变量局部化。压缩后的变量名有时会被误认为是“私有属性”，但这其实只是一种把类所创建的变量名局部化的方式而已：名称压缩并无法阻止类外代码对它的读取。这种功能主要是为了避免实例内的命名空间的冲突，而不是限制变量名的读取。因此，压缩的变量名最好称为“伪私有”而不是“私有”。

伪私有变量名是高级且完全可选的功能，除非你开始在多人的项目中编写大型的类的层次，否则可能不会觉得有什么用处。但是，可能在其他人的代码中看见这个功能，所以即使不用，多少还是得留意。

变量名压缩概览

下面是变量名压缩的工作方式：`class`语句内开头有两个下划线但结尾没有两个下划线的变量名，会自动扩张从而包含了所在类的名称。例如，像`Spam`类内`__X`这样的变量名会自动变成`_Spam__X`：原始的变量名会在头部加入一个下划线，然后是所在类名称。因为修改后的变量名包含了所在类的名称，相当于变得独特。不会和同一层次中其他类所创建的类似变量名相冲突。

变量名压缩只发生在`class`语句内，而且只针对开头有两个下划线的变量名。然而，每个开头有两个下划线的变量名都会发生这件事，包括方法名称和实例属性名称（例如，在`Spam`类内，引用的`self.__X`实例属性会变成`self._Spam__X`）。因为不止有一个类在给一个实例新增属性，所以这种办法是有助于避免变量名冲突的。我们需要用一个例子来了解它是如何工作的。

为什么使用伪私有属性

伪私有属性功能是为了缓和与实例属性储存方式有关的问题。在Python中，所有实例属性最后都会在类树底部的单个实例对象内。这一点和C++模型大不相同，C++模型的每个类都有自己的空间来储存其所定义的数据成员。

注1：这会让使用C++的人产生不必要的恐慌。在Python中，甚至有可能在运行时修改或完全删除类的方法。另一方面，在实际的程序中，几乎没人会这样做。作为描述性语言，Python更关心的是开放而不是约束。此外，回想一下第24章讨论的运算符重载，`__getattr__`和`__setattr__`可用于模拟私有性，但是，实际中也很少使用。

www.TopSage.com

在Python的类方法内，每当方法赋值self的属性时（例如，self.attr = value），就会在该实例内修改或创建该属性（继承搜索只发生在引用时，而不是赋值时）。即使在这个层次中有多个类赋值相同的属性，也是如此，因此有可能发生冲突。

例如，假设当一位程序员编写一个类时，他认为属性名称X是在该实例中。在此类的方法内，变量名被设定，然后取出。

```
class C1:
    def meth1(self): self.X = 88          # Assume X is mine
    def meth2(self): print self.X
```

假设另一位程序员独立作业，对他写的类也有同样的假设。

```
class C2:
    def metha(self): self.X = 99          # Me too
    def methb(self): print self.X
```

这两个类都在各行其事。如果这两个类混合在相同类树中时，问题就产生了。

```
class C3(C1, C2): ...
I = C3()                                  # Only 1 X in I!
```

现在，当每个类说self.X时所得到的值，取决于最后一个赋值是哪个类。因为所有对self.X的赋值语句都是引用一个相同实例，而X属性只有一个（I.X），无论有多少类使用了这个属性名。

为了保证属性会属于使用它的类，可在类中任何地方使用，将变量名前加上两个下划线，如*private.py*这个文件所示。

```
class C1:
    def meth1(self): self.__X = 88        # Now X is mine
    def meth2(self): print self.__X       # Becomes _C1__X in I

class C2:
    def metha(self): self.__X = 99        # Me too
    def methb(self): print self.__X       # Becomes _C2__X in I

class C3(C1, C2): pass
I = C3()                                  # Two X names in I

I.meth1(); I.metha()
print I.__dict__
I.meth2(); I.methb()
```

当加上了这样的前缀时，X属性会扩张，从而包含它的类的名称，然后才加到实例中。如果对I执行dir，或者在属性赋值后查看其命名空间字典，就会看见扩张后的变量名

www.TopSage.com

_C1__X和_C2__X，而不是X。因为扩张让变量名在实例内变得独特，类的编码者可以安全地假设，他们真的拥有任何带有两个下划线的变量名。

TopSage.com

```
% python private.py
{'_C2__X': 99, '_C1__X': 88}
88
99
```

这个技巧可避免实例中潜在的变量名冲突，但是，这并不是真正的私有。如果知道所在类的名称，依然可以使用扩张后的变量名（例如，I._C1__X = 77），在能够引用实例的地方，读取这些属性。另外一方面，这个功能也保证不太可能意外地访问到类的名称。

同样地，这个功能只对较大型的多人项目有用，而且只用于已选定的变量名。别把程序代码弄得难以置信的混乱。只当单个类真的需要控制某些变量名时，才使用这个功能。就较为简单的程序而言，这么做可能过头了。

注意： 此外，看一看第24章所提到的模拟私有实例属性的内容，在__getattr__一节。虽然有可能在Python类中模拟读取控制，但实际中很少这么做，即使是大型的系统也是如此。

新式类

在Python 2.2中，引入一种新的类，称为“新式”（new-style）类。本书这一部分至今为止所谈到的类和新的类相比时，就称为“经典”（classic）类。

新式类和经典类有点不同，然而其差异性对于大多数使用Python的人来说是无关紧要的。再者，经典类模型已陪伴Python 15年的时间了，依然像本书之前描述的一样在使用。

新式类在语法和行为上，几乎完全和经典类兼容。差不多只是多了一些新的高级功能。然而，因为修改了继承的一种特定情况，所以必须当成独特工具来介绍，以免与依赖于之前的行为的任何现有程序发生冲突。

新式类也是使用之前学过的正常类的语法编写。主要的编码差异在于是从内置类型（例如，list）创建子类，从而产生新式类。如果没有恰当的内置类型可用，新的内置名称object就可以作为新式类的超类。

```
class newstyle(object):
    ...normal code...
```


www.TopSage.com

通常情况下，任何从object或其他内置类型衍生的类，都会自动视为新式类（所谓的衍生，指的是包括object的子类、object的子类的子类以及那些只要有内置类型位于超类树的某个地方，新的类就会被当作是新式类）。不是从内置类型衍生出来的类，就会当作经典类来对待。

注意： Python创立者Guido van Rossum说，在Python 3.0中，所有类都自动成为新式类，所以必须从内置超类衍生出来的条件将不存在。因为即使是独立的类都会当作是新式类，而且因为新式类几乎和经典类完全兼容，所以对多数程序员而言，这样的变动将是透明的。

过去曾经有些顾虑，认为Python 3.0中的顶层类可能需要从object衍生出来，但是Guido最近指出，这是没有必要的。对多数程序员而言，Python 3.0中的所有类将同本书所介绍的类一样工作，并且还多了额外的新功能可以使用。不过，无法完全预测以后的发展，所以一定要查看Python 3.0的更新消息来了解前沿的技术。

钻石继承变动

也许新式类中最显著的变化就是，对于所谓的多重继承树的钻石模式（diamond pattern）的继承（也就是有一个以上的超类会通往相同更高的超类）处理方式有点不同。钻石模式是高级设计概念，对于普通类我们没有必要讨论。

简而言之，对经典类而言，继承搜索程序是绝对深度优先，然后才是由左至右。Python一路往上搜索，深入树的左侧，返回后，才开始找右侧。在新式类中，在这类情况下，搜索相对来说是宽度优先的。Python先寻找第一个搜索的右侧的所有超类，然后才一路往上搜索至顶端共同的超类。因为有这样的变动，较低超类可以重载较高超类的属性，无论它们混入的是哪种多重继承树。

钻石继承例子

为了说明起见，举一个经典类构成的简单钻石继承模式的例子。

```
>>> class A:      attr = 1          # Classic
>>> class B(A):   pass
>>> class C(A):   attr = 2
>>> class D(B,C): pass              # Tries A before C
>>> x = D()
>>> x.attr
1
```

此处是在超类A中内找到属性的。因为对经典类来说，继承搜索是先往上搜索到最高，然后返回再往右搜索：Python会先搜索D、B、A，然后才是C（但是，当attr在A找到时，B之上的就会停止）。不过，对于从类似的内置对象衍生出来的新式类而言，Python

www.TopSage.com

会先搜索C（B的右侧），然后才是A（B之上）：也就是先搜索D、B、C，然后才是A（在这个例子中，则会停在C处）。

```
>>> class A(object): attr = 1          # New style
>>> class B(A):      pass
>>> class C(A):      attr = 2
>>> class D(B,C):    pass              # Tries C before A
>>> x = D()
>>> x.attr
2
```

这种继承搜索流程的变化是基于这样的假设：如果在树中较低处混入C，和A相比，可能会比较想获取C的属性。此外，这也是假设C总是要覆盖A的属性：当C独立使用时，可能是真的，但是当C混入经典类钻石模式时，可能就不是这样了。当编写C时，可能根本不知道C会以这样的方式混入。

明确解决冲突

当然，假设的问题就是这是假设的。如果难以记住这种搜索顺序的偏好，或者如果你想对搜索流程有更多的控制，都可以在树中任何地方强迫属性的选择：通过赋值或者在类混合处指出你想要的变量名。

```
>>> class A:      attr = 1             # Classic
>>> class B(A):   pass
>>> class C(A):   attr = 2
>>> class D(B,C): attr = C.attr        # Choose C, to the right
>>> x = D()
>>> x.attr                             # Works like new style
2
```

在这里，经典类树模拟了新式类的搜索顺序：在D中对为属性赋值，使其挑选C中的版本，因而改变了正常的继承搜索路径（D.attr位于树中最低的位置）。新式类也能选择类混合处以上的属性来模拟经典类。

```
>>> class A(object): attr = 1          # New style
>>> class B(A):      pass
>>> class C(A):      attr = 2
>>> class D(B,C):    attr = B.attr     # Choose A.attr, above
>>> x = D()
>>> x.attr                             # Works like classic
1
```

如果愿意以这样的方式解决这种冲突，大致上就能忽略搜索顺序的差异，而不依赖假设来决定所编写的类的意义。自然地，以这种方式挑选的属性也可以是方法函数（方法是正常可赋值的对象）。

www.TopSage.com

大家网
TopSage.com

```
>>> class A:
...     def meth(s): print 'A.meth'
>>> class C(A):
...     def meth(s): print 'C.meth'
>>> class B(A):
...     pass
>>> class D(B,C): pass                  # Use default search order
>>> x = D()                             # Will vary per class type
>>> x.meth()                            # Defaults to classic order
A.meth

>>> class D(B,C): meth = C.meth         # Pick C's method: new style
>>> x = D()
>>> x.meth()
C.meth

>>> class D(B,C): meth = B.meth         # Pick B's method: classic
>>> x = D()
>>> x.meth()
A.meth
```

在这里，我们明确在树中较低处赋值变量名以选取方法。我们也可以明确调用所需要的类。在实际应用中，这种模式可能更为常用，尤其是构造方法。

```
class D(B,C):
    def meth(self):                  # Redefine lower
        ...
        C.meth(self)                 # Pick C's method by calling
```

这类在混合点进行赋值运算或调用而做的选择，可以有效地把代码从类中的差异性隔离出代码。通过这种方式明确地解决冲突，可以确保你的代码不会因以后更新 的Python版本而有所变化（除了新式类需要从内置类型衍生类之外）（注2）。

总之，在默认情况下，钻石模式的搜索对于经典类和新式类是不同的，而这是和旧版不兼容的改变。然而，这种改变只影响钻石模式。就其他继承树结构而言，新式类继承工作起来并没有变化。此外，这里整个问题其实有可能已经比较理论化，在实际应用中重要性不高，因为在Python 2.2之前，都没必要去修改，似乎不太可能影响很多的Python程序。

注2：即使没有经典/新式类的差异，这种技术在一般多重继承场合中也很方便。如果你想要左侧超类的一部分以及右侧超类的一部分，可能就需要在子类中明确使用赋值语句，告诉Python要选择哪个同名属性。我们会在本章结尾的陷阱中再介绍这个概念。此外，钻石继承模式在有些情况下的问题，比此处所提到的还要多（例如，如果B和C都有构造方法会调用A的，那该怎么办？），不过这已不是本书范围之内。

www.TopSage.com

其他新式类的扩展

除了钻石继承搜索模式中的这个改变以外（过于罕见，不需要大多数读者留意），新式类还启用了一些更为高级的可能性。这里逐一进行简要的介绍。

静态和类方法

在Python 2.2中，在类中定义方法是可能的，不需实例就能够调用它：静态方法的运作差不多就像类中的简单无实例函数，而类方法传递的是类而不是实例。名为staticmethod和classmethod的特定的内置函数，必须在类中调用，才能使这些方法模式有效。虽然这个功能是伴随新式类增加的，但静态和类方法也能用于经典类。因此，下一节要谈的就是这个话题。

slots实例

赋值字符串属性名称列表给特殊的__slots__类属性，新式类就有可能限制类的实例能有的合法属性集。这个特殊属性一般是在class语句顶层内对变量__slots__做设置：只有__slots__列表内的这些变量名可赋值为实例属性。然而，就像Python中的所有变量名，实例属性名必须在引用前赋值，即使是列在__slots__中也是这样。以下是说明的例子。

```
>>> class limiter(object):
...     __slots__ = ['age', 'name', 'job']
...
>>> x = limiter()
>>> x.age                                    # Must assign before use
AttributeError: age

>>> x.age = 40
>>> x.age
40
>>> x.ape = 1000                             # Illegal: not in slots
AttributeError: 'limiter' object has no attribute 'ape'
```

这个功能看作是捕捉“打字错误”的方式（对于不在__slots__内的非法属性名做赋值运算，就会侦测出来），而且也是最优化机制（__slots__内的属性可以储存在元组内，而不是字典中，可以更快地进行查找）。

__slots__会破坏Python的动态本质，也就是硬性规定了能够由赋值语句创建的变量名。此外，还有些其他限制和含义，这些过于复杂，不适合在此讨论。例如，有些带有__slots__的实例也许没有__dict__属性字典，使得有些本书中所写的元程序过于复杂。例如，以通用方式列出属性的工具，可能去查看两个来源，而不是一个。参考Python 2.2中的文件和Python标准手册来获得更多细节。

www.TopSage.com

类内容属性

有种称为内容属性（property）的机制，提供另一种方式让新式类定义自动调用的方法，来读取或赋值实例属性。这种功能是第24章谈过、目前用得很多的__getattr__和__setattr__重载方法的替代做法。内容属性和这两个方法有类似效果，但是只在读取所需要的动态计算的变量名时，才会发生额外的方法调用。内容属性（和slots）都是基于属性描述器（attribute descriptor）的新概念（太高级，不适合在这里说明）。

简而言之，内容属性是一种对象，赋值给类属性名称。内容属性的产生是以三种方法（获得、设置以及删除运算的处理器）以及通过文档字符串调用内置函数property。如果任何参数以None传递或省略，该运算就不能支持。内容属性一般都是在class语句顶层赋值［例如，name = property(...)］。这样赋值时，对类属性本身的读取（例如，obj.name），就会自动传给property的一个读取方法。例如，__getattr__方法可让类拦截未定义属性的引用。

```
>>> class classic:
...     def __getattr__(self, name):
...         if name == 'age':
...             return 40
...         else:
...             raise AttributeError
...
>>> x = classic()
>>> x.age                                   # Runs __getattr__
40
>>> x.name                                  # Runs __getattr__
AttributeError
```

下面是相同的例子，改用内容属性来编写。

```
>>> class newprops(object):
...     def getage(self):
...         return 40
...     age = property(getage, None, None, None)  # get,set,del,docs
...
>>> x = newprops()
>>> x.age                                   # Runs getage
40
>>> x.name                                  # Normal fetch
AttributeError: newprops instance has no attribute 'name'
```

就某些编码任务而言，比起传统技术，内容属性不是那么的复杂，而且运行起来比较快。例如，当我们新增属性赋值运算支持时，内容属性就变得更有吸引力：输入的代码比较少，对我们不希望动态计算的属性进行赋值运算时，不会发生额外的方法调用。

```
>>> class newprops(object):
...     def getage(self):
```


www.TopSage.com

```
...         return 40
...     def setage(self, value):
...         print 'set age:', value
...         self._age = value
...     age = property(getage, setage, None, None)
...
>>> x = newprops()
>>> x.age                                   # Runs getage
40
>>> x.age = 42                              # Runs setage
set age: 42
>>> x._age                                  # Normal fetch; no getage call
42
>>> x.job = 'trainer'                       # Normal assign; no setage call
>>> x.job                                   # Normal fetch; no getage call
'trainer'
```

等效的经典类可能会引发额外的方法调用，而且需要通过属性字典传递属性赋值语句以避免死循环。

```
>>> class classic:
...     def __getattr__(self, name):            # On undefined reference
...         if name == 'age':
...             return 40
...         else:
...             raise AttributeError
...     def __setattr__(self, name, value):     # On all assignments
...         print 'set:', name, value
...         if name == 'age':
...             self.__dict__['_age'] = value
...         else:
...             self.__dict__[name] = value
...
>>> x = classic()
>>> x.age                                   # Runs __getattr__
40
>>> x.age = 41                              # Runs __setattr__
set: age 41
>>> x._age                                  # Defined: no __getattr__ call
41
>>> x.job = 'trainer'                       # Runs __setattr__ again
>>> x.job                                   # Defined: no __getattr__ call
```

就这个简单的例子而言，内容属性似乎是赢家。然而，__getattr__和__setattr__的某些应用依然需要更为动态或通用的接口，超出内容属性所能直接提供的范围。例如，在大多数情况下，当类编写时，要支持的属性集无法确认，而且甚至无法以任何具体形式存在（例如，委托任意方法的引用给被包装/嵌入对象时）。在这种情况下，通用的__getattr__或__setattr__属性处理器外加传入的属性名，会是更好的选择。因为这类通用处理器也能处理较简单情况，内容属性大致上就只是选用的扩展功能了。

www.TopSage.com

新的__getattribute__重载方法

__getattribute__方法只适用于新式类，可以让类拦截所有属性的引用，而不局限于未定义的引用（如同__getattr__）。但是，远比__getattr__和__setattr__难用（易于造成死循环）。此方法的细节请参看Python的标准文档。

除了这些新增的功能以外，新式类也整合了本章之前所提到的可建子类的类型的概念。在Python 2.2以后，可建子类的类型和新式类引入时，将类型/类的对立消除。

因为新式类功能全部都是高级话题，我们不会在这本入门书籍中讨论细节。请参考Python 2.2的说明文件和语言参考文献来了解更多的信息。

静态和类方法

在Python 2.2之前，类方法函数调用时绝不能没有实例。在Python 2.2之后，这是默认行为，不过可以通过静态方法调整这个新的选用功能：也就是出现在类中不带self参数的简单函数，并且其设计用于类属性，而不是实例属性。这类方法通常是记录贯穿所有实例的信息（例如，已创建的实例数目），而不是用来给实例提供行为。

前文介绍过无绑定方法：当我们对类进行点号运算（而不是实例）取出方法函数时，就会取得无绑定方法对象。即使都是通过def定义的，无绑定方法对象和简单函数并不相同。它们没有实例时，是无法调用的。

例如，假设我们想使用类属性去计算从一个类产生了多少实例（如下面的文件*spam.py*所示）。记住，类属性是由所有实例共享的，所以我们可以把计数器放在类对象内。

```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1
    def printNumInstances():
        print "Number of instances created: ", Spam.numInstances
```

这样行不通：printNumInstances方法在调用时，还是需要传入一个实例，因为这个函数是和类相结合的（即使def首行中没有参数）。

```
>>> from spam import *
>>> a = Spam()
>>> b = Spam()
>>> c = Spam()
>>> Spam.printNumInstances()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: unbound method must be called with class instance 1st argument
```

这里的问题在于无绑定实例的方法并不完全等同于简单函数。这基本上是知识问题。但是如果不想通过实例调用读取类成员的函数，最直接的想法可能就是将函数变成简单函数，而不是类方法。这样一来，调用时就不需要实例。

```
def printNumInstances():
    print "Number of instances created: ", Spam.numInstances

class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1

>>> import spam
>>> a = spam.Spam()
>>> b = spam.Spam()
>>> c = spam.Spam()
>>> spam.printNumInstances()
Number of instances created:  3
>>> spam.Spam.numInstances
3
```

因为类名称对简单函数而言是可读取的全局变量，这样可正常工作。此外，函数名变成了全局变量，这仅适用于这个单一的模块而已。它不会和程序其他文件中的变量名冲突。

就像往常一样，我们也可以通过实例调用函数，不过，如果创建实例会修改类的数据，不是太方便：

```
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1
    def printNumInstances(self):
        print "Number of instances created: ", Spam.numInstances

>>> from spam import Spam
>>> a, b, c = Spam(), Spam(), Spam()
>>> a.printNumInstances()
Number of instances created:  3
>>> b.printNumInstances()
Number of instances created:  3
>>> Spam().printNumInstances()
Number of instances created:  4
```

在Python 2.2的静态方法扩展功能之前，有些语言理论家宣称，这种技术的存在意味着Python没有类方法，只有实例方法。他们的意思是指，Python类和其他语言的类并不相同。Python真正有的是绑定和无绑定方法对象，语义相当明确。以点号结合类时，就得到无绑定方法，也就是特定种类的函数。Python的确有类属性，但类中的函数需要实例参数。

www.TopSage.com

此外，因为Python通过提供模块作为命名空间的分割工具，通常没有必要把函数打包在类中，除非是为了实现对象行为。模块内的简单函数通常可做到无实例类方法能做的多数事情。例如，本节第一个例子中，printNumInstances其实已和类匹配，因为它存在于相同模块内。唯一失去的特性就是函数名有较宽的作用域：整个模块，而不是类。

使用静态和类方法

现在，还有另一个选择，可以编写和类相关联的简单函数。在Python 2.2中，可以用静态和类方法编写类，两者都不需要在启用时传入实例参数。要设计这个类的方法时，类要调用内置函数staticmethod和classmethod，就像之前讨论过的新式类中提到的那样。

```
class Multi:
    def imeth(self, x):              # Normal instance method
        print self, x
    def smeth(x):                    # Static: no instance passed
        print x
    def cmeth(cls, x):               # Class: gets class, not instance
        print cls, x
    smeth = staticmethod(smeth)      # Make smeth a static method
    cmeth = classmethod(cmeth)       # Make cmeth a class method
```

注意：程序代码中最后两个赋值语句只是重新赋值方法名称smeth和cmeth而已。在class语句中，通过赋值语句进行属性的建立和修改，所以这些最后的赋值语句会覆盖稍早由def所做的赋值。

从严格意义上来讲，Python现在支持三种类相关方法：实例、静态和类。实例方法是本书所见的正常（默认）情况。一定要用实例对象调用实例方法。通过实例调用时，Python会把实例自动传给第一个（最左侧）参数。类调用时，需要手动传入实例。

```
>>> obj = Multi()                  # Make an instance
>>> obj.imeth(1)                   # Normal call, through instance
<__main__.Multi instance...> 1
>>> Multi.imeth(obj, 2)            # Normal call, through class
<__main__.Multi instance...> 2
```

反之，静态方法调用时不需要实例参数，其变量名位于定义所在类的范围内，属于局部变量，而且可以通过继承查找。概括地讲，就是简单函数，只是碰巧写在类中。

```
>>> Multi.smeth(3)                 # Static call, through class
3
>>> obj.smeth(4)                   # Static call, through instance
4
```

类方法类似，但Python自动把类（而不是实例）传入类方法第一个（最左侧）参数中：

```
>>> Multi.cmeth(5)                    # Class call, through class
__main__.Multi 5
>>> obj.cmeth(6)                      # Class call, through instance
__main__.Multi 6
```

静态和类方法是新颖而高级的语言特性，属于相当特定场合应用的角色，这里没有用完整篇幅进行说明。静态方法一般是和类属性合作，来管理贯穿该类产生的所有实例的信息。例如，要记录类生成的实例数目（就像之前的例子），我们可以使用静态方法管理，作为类属性的计数器。因为这类计数和任何特定的实例无关，通过实例读取对其处理的方法并不方便（尤其是制作实例来读取这个计数器，可能会修改计数器的值）。再者，静态方法接近于类，比起在类外编写面向类的函数，提供了更自然的解决方案。

以下是和本节最初的例子等效的静态方法。

```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances += 1
    def printNumInstances():
        print "Number of instances:", Spam.numInstances
    printNumInstances = staticmethod(printNumInstances)

>>> a = Spam()
>>> b = Spam()
>>> c = Spam()
>>> Spam.printNumInstances()
Number of instances: 3
>>> a.printNumInstances()
Number of instances: 3
```

比较把`printNumInstances`移到类外的做法（如之前所做的），这个版本还需要额外的`staticmethod`调用。然而，这样做把函数名称变成类作用域内的局部变量（不会和模块内的其他变量名碰撞），而且把函数程序代码移到靠近其使用的地方（位于`class`语句中）。你应该自己判断这样做是不是有所改善。

在最新的Python版本中，静态方法的设计甚至变得更加简单，这在下一节将做说明。

函数装饰器

因为上一节所说的`staticmethod`调用技术，对有些人来讲似乎很奇怪。因此新增了一个功能，要让这个运算变得简单一点。函数装饰器（function decorator）提供了一种方式，替函数明确了特定的运算模式，也就是将函数包裹了另一层，在另一函数的逻辑内实现。

www.TopSage.com

函数装饰器变成了通用的工具：除了静态方法用法外，也可用于新增多种逻辑的函数。例如，可以用来记录函数调用的信息和在出错时检查传入的参数类型等。从某种程度上来说，函数装饰器类似于第25章讨论过的委托设计模式，但是其设计是为了增强特定的函数或方法调用，而不是整个对象接口。

Python提供一些内置函数装饰器，来做一些运算，例如，标示静态方法，但是程序员也可以编写自己的任意的装饰器。虽然不限于使用类，但用户定义的函数装饰器通常也写成类，把原始函数和其他数据当成状态信息。

从语法上来讲，函数装饰器是一种它后边的函数的运行时的声明。函数装饰器是写成一行，就在定义函数或方法的`def`语句之前，而且由`@`符号，后面跟着所谓的元函数（metafunction）组成：也就是管理另一函数（或其他可调用对象）的函数。例如，如今的静态方法可以用下面的的装饰器语法编写。

```
class C:
    @staticmethod
    def meth():
        ...
```

从内部来看，这个语法和下面的写法有相同效果（把函数传递给装饰器，再赋值给最初的变量名）。

```
class C:
    def meth():
        ...
    meth = staticmethod(meth)                    # Rebind name
```

结果就是，调用方法函数的名称，实际上是触发了它`staticmethod`装饰器的结果。因为装饰器会传回任何种类的对象，这也可以让装饰器在每次调用上增加一层逻辑。装饰器函数可返回原始函数，或者新对象（保存传给装饰器的原始函数，这个函数将会在额外逻辑层执行后间接的运行）。

事实上，装饰器语法可让你对被装饰的函数或方法新增几层包裹的逻辑。如下面形式的装饰器行。

```
@A @B @C
def f():
    ...
```

运行时如下所示。

```
def f():
    ...
f = A(B(C(f)))
```


www.TopSage.com

这是把最初的函数传递给三个不同的装饰器，并把结果赋值给最初的变量名。同样地，结果就是当调用最初函数的名称时，有三层逻辑会启用从而加强了最初的函数。

TopSage.com

装饰器例子

以下是用户定义的装饰器的例子。回想第24章，__call__运算符重载方法替类实例实现函数调用接口。下面的程序通过这种方法定义类，在实例中储存装饰的函数，并捕捉对最初变量名的调用。因为这是类，也有状态信息（记录所做调用的计数器）。

```
class tracer:
    def __init__(self, func):
        self.calls = 0
        self.func  = func
    def __call__(self, *args):
        self.calls += 1
        print 'call %s to %s' % (self.calls, self.func.__name__)
        self.func(*args)

@tracer
def spam(a, b, c):            # Wrap spam in a decorator object
    print a, b, c

spam(1, 2, 3)                 # Really calls the tracer wrapper object
spam('a', 'b', 'c')           # Invokes __call__ in class
spam(4, 5, 6)                 # __call__ adds logic and runs original object
```

因为spam函数是通过tracer装饰器执行的，当最初的变量名spam调用时，实际上触发的是类中的__call__方法。这个方法会计算和记录该次调用，然后委托给原始的包裹的函数。注意*name参数语法是如何打包并解开传入的参数的。因此，此装饰器可用于包裹携带任意数目参数的任何函数。

结果就是新增一层逻辑至原始的spam函数。以下是此脚本的输出：第一列来自tracer类，第二列来自spam函数。

```
call 1 to spam
1 2 3
call 2 to spam
a b c
call 3 to spam
4 5 6
```

仔细学习一下这个例子的代码来加深理解。

虽然是通用的机制，但函数装饰器是高级功能，其主要的受益者是工具的编写者，而不是应用程序员，所以要了解这个主题的细节。

www.TopSage.com

类陷阱

大多数类的问题通常都可以浓缩为命名空间的问题（这是有道理的，因为类只是多了一些技巧的命名空间而已）。本节所谈的有些问题更像是高级类的用法研究，而不是问题，而其中有一两个陷阱已随着最新Python版本的发布而改进了。

修改类属性的副作用

从理论的角度讲，类（和类实例）是可改变的对象。就像内置列表和字典一样，可以给类属性赋值，并且进行在原处的修改，同时意味着修改类或实例对象，也会与影响对它的多处引用。

这通常就是我们想要的（也是对象一般修改其状态的方式），修改类属性时，了解这一点特别重要。因为所有从类产生的实例都共享这个类的命名空间，任何在类层次所做的修改都会反应在所有实例中，除非实例拥有自己的被修改的类属性版本。

因为类、模块以及实例都只是属性命名空间内的对象，一般可通过赋值语句在运行时修改它们的属性。在类主体中，对变量名a的赋值语句会产生属性X.a，在运行时存在于类的对象内，而且会由所有X的实例继承。

```
>>> class X:
...     a = 1          # Class attribute
...
>>> I = X()
>>> I.a               # Inherited by instance
1
>>> X.a
1
```

到目前为止，都不错，这是正常的情况。但注意到，当我们在class语句外动态修改类属性时，将发生什么事情：这也会修改每个对象从该类继承而来的这个属性。再者，在这个进程或程序执行时，由类所创建的新实例会得到这个动态设置值，无论该类的源代码是怎样的情况。

```
>>> X.a = 2           # May change more than X
>>> I.a               # I changes too
2
>>> J = X()           # J inherits from X's runtime values
>>> J.a               # (but assigning to J.a changes a in J, not X or I)
2
```

这是有用的功能还是危险的陷阱？自己判断。你可以修改类的属性，不修改实例就可以

www.TopSage.com

达到相同的目的。这种技术可以模拟其他语言的“记录”或“结构体”（struct）。考虑下面的不常见但是合法的Python程序。

```
class X: pass                          # Make a few attribute namespaces
class Y: pass

X.a = 1                                # Use class attributes as variables
X.b = 2                                # No instances anywhere to be found
X.c = 3
Y.a = X.a + X.b + X.c

for X.i in range(Y.a): print X.i       # Prints 0..5
```

在这里，类X和Y就像“无文件”模块：储存我们不想发生冲突的变量的命名空间。这是完全合法的Python程序设计技巧，但是使用其他人编写的类就不合适了。你永远无法知道，修改的类属性会不会对类内部行为产生重要影响。如果你要仿真C的结构体，最好是修改实例而不是类，这样的话，只有影响一个对象。

```
class Record: pass
X = Record()
X.name = 'bob'
X.job  = 'Pizza maker'
```

多重继承：顺序很重要

这很明显，但还是需要强调一下：如果使用多重继承，超类列在class语句首行内的顺序就很重要。Python总是会根据超类在首行的顺序，由左至右搜索超类。

例如，在第25章的多重继承的例子中，假设Super类也实现了__repr__方法。我们想要继承Lister的还是Super的？我们会从先列在Sub类首行的那个类取得该方法，因为继承搜索是从左至右进行的。假设，我们先编写Lister，因为这个类的整个目的就是其定制了的__repr__。

```
class Lister:

    def __repr__(self): ...

class Super:
    def __repr__(self): ...

class Sub(Lister, Super):      # Get Lister's __repr__ by listing it first
```

但现在，假设Super和Lister各自有其他的同名属性的版本。如果我们想要使用Super的变量名，也想要使用Lister的变量名，在类首行的编写顺序就没什么帮助：我们得手动对Sub类内的属性名赋值来覆盖继承。

www.TopSage.com

```
class Lister:
    def __repr__(self): ...
    def other(self): ...

class Super:
    def __repr__(self): ...
    def other(self): ...

class Sub(Lister, Super):      # Get Lister's __repr__ by listing it first
    other = Super.other        # But explicitly pick Super's version of other
    def __init__(self):
        ...

x = Sub()                      # Inheritance searches Sub before Super/Lister
```

在这里，对Sub类中的other做赋值运算，会建立Sub.other：对Super.other对象的引用值。因它在树中的位置较低，Sub.other实际上会隐藏Lister.other（继承搜索时正常会找到的属性）。同样地，如果在类首行中先编写Super来挑选其中other，就需要刻意地选取Lister中的方法。

```
class Sub(Super, Lister):          # Get Super's other by order
    __repr__ = Lister.__repr__     # Explicitly pick Lister.__repr__
```

多重继承是高级工具。即使你掌握了上一段所讲的内容，谨小慎微的使用依然是个不错的主意。否则，对于任意远的子类中的变量的含义，将会取决于混入的类的顺序（这里所示技术的另一个例子，可以参考本章之前讨论新式类时所提到明确解决冲突）。

一条简明的原则就是，当混合类尽可能的独立完备时，多重继承的工作状况最好，因为混合类可以应用在各种环境中，因此不应该对树中其他类相关的变量名有任何假设。之前研究过的伪私有属性功能可以把类仰赖的变量名本地化，限制混合类可以混入的名称，因此会有所帮助。例如，在这个例子中，如果Lister只是要导出特殊的__repr__，就可将其另一个方法命名为__other，从而避免和其他类发生冲突。

类、方法以及嵌套作用域

这个陷阱在Python 2.2引入嵌套函数作用域后就消失了，不过本书对此进行了保留，只是作为历史回顾，也是为旧版本Python的用户着想，因为这可以示范当一层嵌套是类时，新的嵌套函数作用域会发生什么事情。

类引入了本地作用域，就像函数一样。所以相同的作用域行为也会发生在class语句的主体中。此外，方法是嵌套函数，也有相同的问题。当类进行嵌套时，看起来令人困惑就比较常见了。下面的例子中（文件*nester.py*），generate函数返回嵌套的Spam类的实例。在其代码中，类名称Spam是在generate函数的本地作用域中赋值的。但是，在

Python 2.2以前，在类的方法函数中，是看不见类名称Spam的。方法只能读取其自己的本地作用域、generate所在的模块以及内置变量名。

```
def generate():
    class Spam:
        count = 1
        def method(self):          # Name Spam not visible:
            print Spam.count       # Not local(def), global(module), built-in
    return Spam()

generate().method()

C:\python\examples> python nester.py
Traceback (innermost last):
  File "nester.py", line 8, in ?
    generate().method()
  File "nester.py", line 5, in method
    print Spam.count               # Not local(def), global(module), built-in
NameError: Spam
```

这个例子可在Python 2.2和以后的版本中执行，因为任何所在函数def的本地作用域都会自动被嵌套的def中看见（包括嵌套的方法def，就像这个例子所演示的那样）。但是，Python 2.2之前的版本就行不通了（参考之后的可能解法）。

注意：即使是在2.2版中，方法def还是无法看见所在类的局部作用域。方法def只看得见所在def的局部作用域。这就是为什么方法得通过self实例，或类名称去引用所在类语句中定义的方法和其他属性。例如，方法中的程序代码必须使用self.count或Spam.count，不能只是count。

如果你正在使用2.2版以前的版本，有很多方式可以使用上一个例子。其中一种最简单的方式，就是以全局声明，把名称Spam放在所在模块的作用域中。因为方法看得见所在模块中的全局变量名，就能够引用Spam。

```
def generate():
    global Spam                    # Force Spam to module scope
    class Spam:
        count = 1
        def method(self):
            print Spam.count       # Works: in global (enclosing module)
    return Spam()

generate().method()                # Prints 1
```

更好的替代做法是重构代码，使得Spam定义在模块顶层，而不是使用全局声明。嵌套方法函数和顶层generate就会在全局作用域中找到Spam。

www.TopSage.com

```
def generate():
    return Spam()

class Spam:                         # Define at top level of module
    count = 1
    def method(self):
        print Spam.count            # Works: in global (enclosing module)

generate().method()
```

事实上，这种做法适用于所有Python版本。一般而言，如果避免嵌套类和函数，代码都会比较简单。

如果想做得既复杂又难懂，也可以完全放弃在方法中引用Spam，而是改用特殊的__class__属性，来返回实例的类对象。

```
def generate():
    class Spam:
        count = 1
        def method(self):
            print self.__class__.count        # Works: qualify to get class
    return Spam()

generate().method()
```

“过度包装”

如果运用得当的话，OOP的程序代码重用功能会在开发的攻坚阶段发挥其优越性。不过，有时候，OOP的抽象潜质会被过度使用，使代码晦涩难懂。如果类层次太深，程序就变得晦涩难懂。你得搜索许多类，才能找到某个运算是在做什么。

例如，我曾在一家C++公司碰到过数千个类（有些由机器产生），多达15层的继承。在这种复杂系统中，要解读方法调用，往往是一项很艰难的任务：即使是最基本的运算，都得看好几个类才行。实际上，系统的逻辑封装的太深，以至于在有些情况下，了解一段程序需要好几天时间去查找相关的文件才行。

Python程序设计最通用原则也适用于此：除非真的有必要，否则不要把事情弄得很复杂。把程序代码包裹很多层类直到人们难以理解为止，这绝对是个坏主意。抽象是多态和封装的基础，只要恰当地使用就会成为非常高效的工具。然而，如果要类接口保持直观性，避免代码过于抽象，如果没有很好的理由就保持类层次的简短和平坦，这样的话，就能够让调试变得简单，也有助于提高代码的可维护性。

www.TopSage.com

本章小结

本章介绍了一些与类相关的高级话题，包括了创建内置类型的子类、伪私有属性、新式类、静态方法以及函数装饰器。多数都是Python OOP模型中可选的扩展功能，当你开始编写较大的面向对象程序时，这些会比较有用。

这是这一部分的最后一章，在本章末尾有实验练习题。一定要做一下，做一些真正的类的实际编程工作。下一章中，我们要开始探讨最后的核心语言话题：异常。异常是Python用于错误和其他情况下于代码的通信机制。这是相当轻松的议题，留到最后是因为异常如今也编写成类。不过，我们完成最后主题前，先看一看本章习题和实验练习题。

www.TopSage.com

本章习题

1. 列举出两种能够扩展内置对象类型的方法？
2. 函数修饰器是用来做什么的？
3. 怎样编写新式类？
4. 新式类与经典类有何不同？
5. 正常方法和静态方法有何不同？
6. 你等多长时间才能得到一个“Holy Hand Grenade”？

习题解答

1. 你可以在包装类中内嵌内置对象，或者直接做内置类型的子类。后者显得更简单，因为大多数原始的行为都被自动的继承了。
2. 函数修饰器通常是用来给现存的函数增加每次函数被调用是都会运行的一层逻辑。它们可以用来记录函数的日志或调用次数，检查参数的类型等。它们同样可以用做“静态方法”（一个在类中的函数，不需要传入实例）。
3. 可以通过对对象的内置类（或者其他的内置类型）继承来编写新式类。在Python 3.0中可能就不需要这样做了，所有的类都将会默认为新式类。
4. 新式类与多重继承树中的钻石搜索模式有所不同，它们实际上是以广度优先（横向）进行搜索的，而不是深度优先（向上）。新式类还支持一系列额外的高级工具，包括内容属性和`__slots__`实例属性列表。
5. 正常（实例）方法会接受第一个`self`参数（隐含的实例），但是静态方法不是这样。静态方法只是嵌套在类对象中的简单函数。为了使一个方法成为静态方法，它必须可以通过特殊的内置函数运行，或者使用装饰器进行装饰。
6. 三秒。（或者，更准确的说：“And the Lord spake, saying, `First shalt thou take out the Holy Pin. Then, shalt thou count to three, no more, no less. Three shalt be the number thou shalt count, and the number of the counting shall be three. Four shalt thou not count, nor either count thou two, excepting that thou then proceed to three. Five is right out. Once the number three, being the third number, be reached, then lobbest thou thy Holy Hand Grenade of Antioch towards thy foe, who, being naughty in my sight, shall snuff it.'”）（注3）。

注3：引文来自*Monty Python和Holy Grail*上。

www.TopSage.com

第六部分 练习题

这些练习题会让你编写一些类，以及对一些现有的代码做些实验。当然在现有代码中出现的问题是必须存在的。为了运行练习题5，要么从Internet上找出类的代码来下载，要么手动输入它（相当清楚）。这些程序开始变得复杂起来，所以要确认查看了本书末尾的解答，这些解答可以作为向导。你可以在附录B中找到解答。

1. **继承**。编写一个名为Adder的类，导出方法add(self, x, y)，作用是打印“Not Implemented”的消息。之后，定义两个Adder的子类，来实现其中的add方法：

   ListAdder

   有一个add方法，它会返回两个列表参数合并的结果。

   DictAdder

   有一个add方法，可以返回一个新的字典，该类包含两个字典参数所包含的所有元素（任意加法的定义都行）。

   通过创建三个类的实例并且调用其方法来做实验。

   现在，扩展Adder超类，使其在实例中通过一个构造器保存一个对象（例如，将self.data赋值为一个列表或一个字典），并且通过__add__方法重载+运算符为add方法打补丁［例如，X + Y触发X.add(X.dada, Y)］。哪里是最适合放置构造器和操作符重载方法的地方（也就是说，在哪一个类里）？哪种对象可以增加在类实例中？

   实际上，你可能已经发现了编写方法只接受一个真正的参数更简单［例如，add(self, y)］，并且add将那个参数加载实例当前的data属性上（例如，self.data + y）。这是不是比给add传入两个参数更合理？你会说这会让你的类更“面向对象”吗？

2. **运算符重载**。编写一个名为Mylist的类遮住（“包装”）了Python的列表。它应该重载大多数的列表操作符和运算，包括+、索引、迭代、分片以及列表方法（例如，append和sort）。查看Python参考手册以获得所有能够支持的方法的列表。并且，为类提供一个构造器接受现有的列表（或者一个Mylist实例）并且将其元素拷贝到实例成员中。在交互模式下测试这个类。下列是需要探讨的问题：

   a. 为什么在这里拷贝初始值很重要？

   b. 你能使用一个空分片（例如，start[:]）来拷贝Mylist实例的初始值吗？

   c. 有一种通用的方法把列表方法调用部署到被包装的列表吗？

www.TopSage.com

大家网 TopSage.com

d. 你可以让Mylist加正常的列表吗？如果是列表加Mylist实例呢？

e. 像+和分片这样的倡导做的返回值应该是什么类型的对象呢？如果是索引操作的返回值呢？

f. 如果你使用的是最新的Python版本（2.2版或之后的版本），你可以通过嵌入一个真正的列表在一个单独的类中来实现封装类，或者通过一个子类来扩展内置的列表类型。哪一种方法更简单?为什么?

3. **子类**。创建一个名为MylistSub的习题2中Mylist的一个子类，让它扩展Mylist，能够在重载运算调用前通过stdout打印一条信息，并且计算调用的次数。MylistSub应该在congMylist中继承了基本的方法行为。增加了一个序列给MylistSub应该打印一条信息，增加了对+调用的计数器，并且执行了超类的方法。此外，引入了一个新的打印操作计数器到stdout的方法，并且在交互模式下实验你编写的类。你是对每个实例都计算了调用的次数，还是对每个类（对这个类的所有的实例）？要是你的程序两种都可以的话该如何编写？（提示：这取决于计数成员是赋值给了哪个对象：类成员是由所有的实例所共享的，而self的成员是每个实例的数据）。

4. **元类方法**。编写一个名为Meta的类，有一个能够截获所有的属性点号运算的方法（包括读取和复制），并且打印其参数在stdout中列出。创建一个Meta的实例，并且在交互模式下通过对它进行点号运算来实验。当你尝试在表达式中使用这个实例的时候会发生什么？用你编写的类试试加法、索引以及分片运算。

5. **集合对象**。使用在“通过嵌入扩展类型”中的集合类，运行下面的命令来做如下的操作。

a. 创建两个整数的集合，并且通过&和|操作符表达式来计算它们的交集和并集。

b. 从字符串创建一个集合，并且试着对集合进行索引运算。在类的内部调用的是哪个方法?

c. 试着使用for循环迭代字符串集合中的每个元素。这次运行的是哪个方法?

d. 尝试为字符串集合和一个简单的Python字符串进行交集和并集计算。这样可行吗?

e. 现在，通过子类扩展集合，使其通过使用*arg的参数形式从而能够处理任意多的操作对象。（提示：参考第16章中类似的算法）。使用集合子类来计

www.TopSage.com

算多个操作对象的交集和并集。该如何对三个或更多的操作对象进行交集计算，因为&操作符只有左右两边？

f. 怎样才能在集合类中模拟其他的列表操作？（提示：__add__能够捕获合并运算，而 __getattr__可以传递给被包装的列表大多数的列表方法调用）

6. **类树链接**。在第24章和第25章都提到了类有一个__bases__属性，它会返回它们的超类对象的元组（在类的首行中的括号中的对象）。使用__bases__来扩张Lister混合类（参考25章），以便能够打印实例的类的超类的名称。当完成的时候，第一行的字符串表现形式看起来应该如下所示（地址可能不尽相同）。

```
<Instance of Sub(Super, Lister), address 7841200:
```

怎样才能列举出从类属性中继承的属性呢？（提示：类有一个__dict__属性。）试着扩展Lister类来显示所有的可读取的超类以及它们的属性；对于爬类树的更多提示，请参考第24章的例子*classtreee.py*以及第25章对在Python 2.2中使用dir和getattr的介绍。

7. **组合**。通过定义4个类来模拟一个快餐订餐的场景：

Lunch
: 一个容器和控制器的类。

Customer
: 扮演顾客。

Employee
: 顾客从他那里订餐。

Food
: 顾客买的东西。

下面是你将要定义的类和方法。

```
class Lunch:
    def __init__(self)                  # Make/embed Customer and Employee
    def order(self, foodName)           # Start a Customer order simulation
    def result(self)                    # Ask the Customer what kind of Food it has

class Customer:
    def __init__(self)                                  # Initialize my food to None
    def placeOrder(self, foodName, employee)            # Place order with an Employee
    def printFood(self)                                 # Print the name of my food

class Employee:
    def takeOrder(self, foodName)       # Return a Food, with requested name
```


www.TopSage.com

```
class Food:
    def __init__(self, name)          # Store food name
```

模拟的订单流程如下。

a. Lunch类的构造器应该创建并嵌入一个Customer的实例和一个Employee的实例，并且应该导入一个名为order的方法。当其被调用时，这个order方法应该要求Customer实例通过调用自身的placeOrder要一个订单。Customer的placeOrder方法应该转向要求Employee对象一个新的Food对象，通过调用Empmloyee的takeOrder方法。

b. Food对象应该保存了一个食物名的字符串（例如，“burritos”），从Lunch.order传递到Customer.placeOrder,再到Employee.takeOrder，最后到Food的构造器。顶层的Lunch类应该也到处一个名为result的方法，它要求顾客打印从Emplopee通过订单收到的食物的名字（这能够用来测试你的模拟）。

注意，Lunch需要传入Employee或者自身给Customer，才能让Customer去调用Employee的方法。

在交互模式下，用已编写的类做实验，导入Lunch类，调用它的order方法来运行一个交互，之后调用它的result方法来验证Customer得到了他或她所定的食物.如果你愿意的话，也可以在定义类的文件中简单地编写一个测试案例作为自我测试的代码，编写代码时，使用在第21章中用过的__name__技巧。在这次模拟中，Customer是一个实际的作用者。如果改成由Employee在顾客/员工的交互中进行主导，你又该如何修改你的类呢？

8. **动物园动物的继承层次。**思考如图26-1所示的类树。在一个模块中编写一个包含六个类语句的集合，用Python的继承为这个生物分类建模。之后，为每一个类都增加一个speak方法，以及在顶层的Animal超类中增加一个reply方法，从而可以简单地调用self.speak来引入下面的子类中的不同分类的信息（这将开始一个独立的继承搜索）。最后，在Hacker类中去掉speak方法，从而可以让它使用默认的方法。当你完成以后，你的类应该是这个样子，如图26-1所示。

```
% python
>>> from zoo import Cat, Hacker
>>> spot = Cat()
>>> spot.reply()                    # Animal.reply; calls Cat.speak
meow
>>> data = Hacker()                 # Animal.reply; calls Primate.speak
>>> data.reply()
Hello world!
```


www.TopSage.com

大家网
TopSage.com

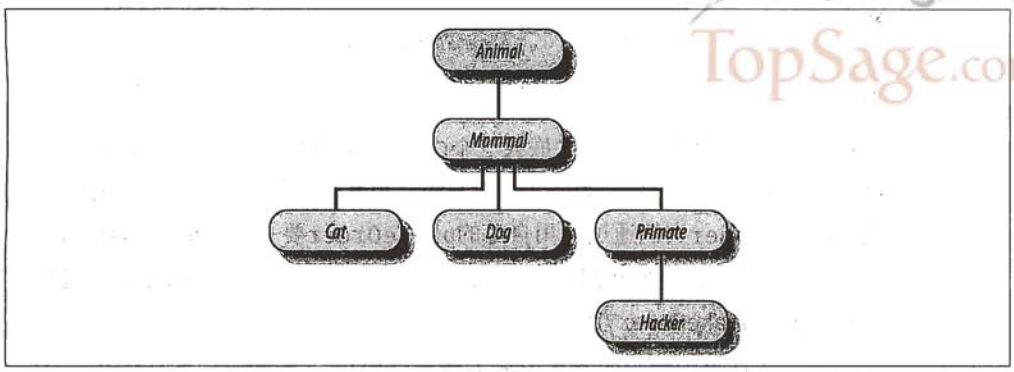


图26-1：动物园的继承层次是由连接在属性继承搜索树上的类构成的。Animal有一个通用的“reply”方法，但是每个类可能都有自己的由“reply”所调用的“speak”方法

9. **描绘死鹦鹉。**思考如图26-2所示结构的主题。编写一系列Python的类并通过组合来实现这个结构。编写你的场景对象来定义个动作方法，并且嵌入Customer实例、Clerk和rrot —— 所有的都应该定义一个line方法来打印出独特的消息。嵌入的对象可以继承一个通用的超类，其中定义了line并提供了简单的文本信息，或者让它们自己定义line。最后，你的类运行起来如下所示。

```
% python
>>> import parrot
>>> parrot.Scene().action()          # Activate nested objects
customer: "that's one ex-bird!"
clerk: "no it isn't..."
parrot: None
```

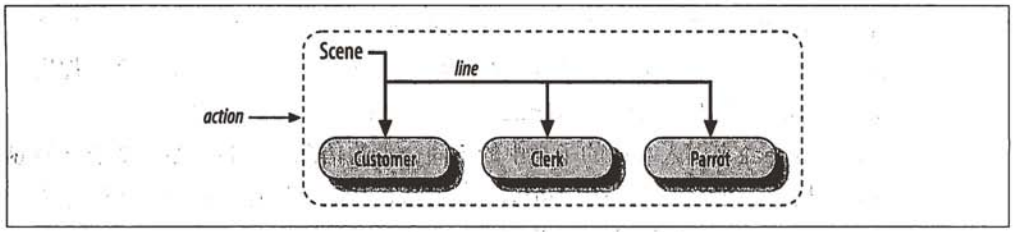


图26-2：这个场景有一个嵌入并指导其他三个类的实例（Customer、Clerk和Parrot）的控制器类(Scene)构成。嵌入的实例的类可以参与到继承层次中来。组合和继承常常是为了代码宠儿而组织类的相当有用的方法

www.TopSage.com

为什么要在意：大师眼中的OOP

当我开始教Python类的时候，无一例外的发现班上有两种平分秋色的现象。那些曾经使用过OOP的人很强烈地表示了他们的赞同，而那些没有OOP经验的人则开始眼神呆滞（要么就是开始打盹）。

像这样的书籍，我大费笔墨地介绍了很多内容，就像第22章新的宏伟蓝图这样的概览，如果你已经开始觉得OOP只不过是计算机科学中毫无意义的崇拜对象的话，那么你也许应该开始重新复习这一部分了。

在真实的课堂上，为了帮助那些新手上路（并且让他们保持清醒），我已经知道了不能再停止向听众中的专家询问他们为何使用OOP这样的问题了。如果这个话题对于你来说是新的话，他们提供的结果或许会遮住一些OOP目的的一些光芒。

我进行了少许的加工，总结了多年来我的学生所举出的使用OOP的最常见的原因如下所示。

代码重用

这很简单（并且是使用OOP最主要的原因）。通过支持继承，类允许通过定制来编程，而不是每次都从头开始一个项目。

封装

在对象接口后包装其实现的细节，从而隔离了代码的修改对用户产生的影响。

结构

类提供了一个新的本地作用域，最小化了变量名冲突。它们还提供了一种编写和查找实现代码，以及去管理对象状态的自然的场所。

维护性

类自然而然地促进了代码的分解，这让减少了冗余。多亏支持类的结构以及代码重用，这样每次只需要修改代码中一个拷贝就可以了。

一致性

类和继承可以实现通用的接口。这样你的代码有了统一的外表和观感。这简化了代码的调试、理解以及维护。

— 待续 —

www.TopSage.com

多态

这更像是一个OOP的属性而不是一条使用它的理由，但是通过广泛的支持代码，多态让代码更灵活和有了广泛的适用性，因此有了更好的可重用性。

其他

此外，学生们给出的使用OOP的最重要的理由就是：这在一份简历上看起来棒极了（好吧，我把这个当成一个笑话，但是如果你打算在如今的软件领域工作的话，熟悉OOP是相当重要的）。

最后，记住我在第6部分开始说过的：在你使用OOP一段时间以后，你才会完完全全地感激它。选择一个项目，研究更大的例子，通过练习来实现（做你觉得适合OO代码的一切）；它值得你努力。

www.TopSage.com

大家网
TopSage.com

第七部分

异常和工具

www.TopSage.com

www.TopSage.com

第27章

异常基础

本书最后一部分将要面对的是异常，也就是可以改变程序中控制流程的事件。在Python中，异常会被错误自动地触发，也能由代码触发和截获。异常由四个语句处理，这一部分会对它们进行介绍。第一种有两种变异（在这里分开列举），而最后一种在Python 2.6之前都是可选的扩展功能。

`try/except`
: 捕捉由Python或你引起的异常并恢复。

`try/finally`
: 无论异常是否发生，执行清理行为。

`raise`
: 手动在代码中触发异常。

`assert`
: 有条件地在程序代码中触发异常。

`with/as`
: 在Python 2.6和后续版本中实现环境管理器（在2.5版中是可选功能）。

这个话题留到本书最后一部分是因为需要了解类，才能编写异常。不过，也存在一些例外，Python的异常处理相当简单，因它为已经整合到了语言本身中，成为另一个高级工具。

有件事从一开始就需要提醒大家：从本书首度出版以来，异常已经经历了两次主要的变动：`finally`分句现在可以出现在和`except`及`else`分句相同的`try`语句内，而用户定义的异常现在应该要写成类的实例，而不是字符串。本书会说明新旧的方式，因为在未来一段时间内，还是有可能看到使用最初的技术编写而成的程序的。在这个过程中，本书会介绍这个领域的演变经过。还会说明新的`with`语句，虽然其正式用法要到以后的版本中才会揭晓。

www.TopSage.com

为什么使用异常

简而言之，异常让我们从一个程序中任意大的代码块中跳出来。考虑本书之前提到过的制作披萨机器人的例子。假设认真对待这个想法，并且实际创造出这样的机器。要制作披萨时，厨房机器人需要执行计划，而我们在这里实现成Python程序：接订单、准备面团、加上饼料、烘烤等。

现在，假设在烘烤阶段，有的地方出错了。也许是烤炉坏掉，或者是机器人算错时间，结果起火燃烧。显然，我们需要能很快地跳到处理这类情况的代码。此外，在这些罕见的情况下，我们无法完成披萨，只能放弃整个计划。

这正是异常做的事：可以在一个步骤内跳至异常处理器，放弃之后所有暂停的函数调用。异常是一种结构化的“超级goto”（注1）。异常处理器（try语句）会留下标示，并可执行一些代码。程序前进到以后的某处代码时，引发异常，会使Python立即跳回到那个标示，丢弃任何该标示留下后所调用的激活的函数。异常处理器中的代码，可以恰当地响应被引发的异常（例如，调用消防部门）。再者，因为Python会立即跳到处理器的语句，对于可能会发生失败的函数，通常就没有必要每次调用时检查这些函数的状态码了。

异常的角色

在Python中，异常通常可以用于各种用途。下面是它最常见的几种角色。

错误处理

每当在运行时检测到程序错误时，Python就会引发异常。可以在程序代码中捕捉和响应错误，或者忽略已发生的异常。如果忽略错误，Python默认的异常处理行为将启动：停止程序，打印错误消息。如果不想启动这种默认行为，就要写try语句来捕捉异常并从异常中恢复：当检测到错误时，Python会跳到try处理器，而程序在try之后会重新继续执行。

事件通知

异常也可用于发出有效状态的信号，而不需在程序间传递结果标志位，或者刻意对其进行测试。例如，搜索的程序可能在失败时引发异常，而不是返回一个整数结果代码（而且这段代码很有可能不会有一个有效的结果）。

注1：如果用过C语言，可能想知道，Python异常类似C的setjmp/logjmp标准函数对：try语句就像setjmp，而raise就像logjmp。但是，在Python中，异常是基于对象的，是执行模型的标准组成成份。

www.TopSage.com

特殊情况处理

有时，发生了某种很罕见的情况，很难调整代码去处理。通常会在异常处理器中处理这些罕见的情况，从而省去编写应对特殊情况的代码。

终止行为

正如将要看到的一样，try/finally语句可确保一定会进行需要的结束运算，无论程序中是否有异常。

非常规控制流程

最后，因为异常是一种高级的“goto”，它可以作为实现非常规的控制流程的基础。例如，虽然反向跟踪（backtracking）并不是语言本身的一部分，但它能够通过Python的异常来实现，此外需要一些辅助逻辑来退回赋值语句（注2）。

这部分稍后会介绍异常的一些典型用法。现在，让我们先看一看Python的异常处理工具。

异常处理：简明扼要

和本书介绍过的其他核心语言话题相比，异常对Python而言是相当简单的工具。因为它们是如此简单，那么我们就马上看第一个例子吧。假设编写了下面的函数。

```
>>> def fetcher(obj, index):
...     return obj[index]
...
```

这个函数没什么特别的，只是通过传入的索引值对对象进行索引运算。在正常运算中，它将返回合法的索引值的结果。

```
>>> x = 'spam'
>>> fetcher(x, 3)           # Like x[3]
'm'
```

然而，如果要求这个函数对字符串末尾以后的位置做索引运算，当函数尝试执行obj[index]时，就会触发异常。Python会替序列检测到超出边界的索引运算，并通过抛出（触发）内置的IndexError异常进行报告。

注2：反向跟踪是高级话题，并不是Python语言的一部分（即使2.2版新增了生成器函数），所以本书在这里不会多谈。概括地讲，反向跟踪撤销其跳跃前所有的计算结果，Python异常则不是这样（也就是说，在进入try语句以及异常引发这段时间内，赋值的变量不会重设为之前的值）。如果好奇的话，可以参考有关人工智能、Prolog或Icon编程语言的书籍。

www.TopSage.com

```
>>> fetcher(x, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in fetcher
IndexError: string index out of range
```

因为我们的代码没有刻意捕捉这个异常，所以它将会一直向上返回到程序顶层，并启用默认的异常处理器：就是打印标准错误消息。此时，你也许已经熟悉了标准错误消息。这些消息包括引发的异常还有堆栈跟踪：也就是异常发生时激活的程序行和函数清单。通过交互模式编写代码时，文件就是“stdin”（标准输入流）或“pyshell”（在IDLE中），所以文件的行号在这里并没有太大的意义。

在交互模式提示符环境外启动的更为现实的程序中，顶层的默认处理器就是立刻终止程序。对简单的脚本而言，这种行为很有道理。错误通常应该是致命错误，而当其发生时，所能做的就是查看标准错误消息。不过，在有些情况下，这并不是我们想要的。例如，服务器程序一般需要在内部错误发生时依然保持工作。如果你不想要默认的异常行为，就需要把调用包装在try语句内，自行捕捉异常。

```
>>> try:
...     fetcher(x, 4)
... except IndexError:
...     print 'got exception'
...
got exception
>>>
```

现在，当try代码块执行时触发异常，Python会自动跳至处理器（指出引发的异常名称的except分句下面的代码块）。像这样以交互模式进行时，在except分句执行后，我们就会回到Python提示符下。在更真实的程序中，try语句不仅会捕捉异常，也会从中恢复执行。

```
>>> def catcher():
...     try:
...         fetcher(x, 4)
...     except IndexError:
...         print 'got exception'
...     print 'continuing'
...
>>> catcher()
got exception
continuing
>>>
```

这次，在异常捕捉和处理后，程序在捕捉了整个try语句后继续执行：这就是我们之所以得到“continuing”消息的原因。我们没有看见标准错误消息，而程序也将正常运行下去。

www.TopSage.com

异常能由Python或程序引发，也能捕捉或忽略。要手动触发异常，就是执行raise语句（或assert，这是有条件的raise）。用户定义的异常的捕捉方式和内置异常一样，如下所示。

```
>>> bad = 'bad'
>>> try:
...     raise bad
... except bad:
...     print 'got bad'
...
got bad
```

如果没捕捉到异常，用户定义的异常就会向上传递直到顶层默认的异常处理器，并通过标准错误消息终止该程序。在这个情况下，标准消息包括用于识别异常的字符串文本。

```
>>> raise bad
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in ?
    raise bad
bad
```

在其他情况下，错误消息也许会包含类提供的文字，用于识别异常。就像下一章你将看到的那样，用户定义的异常能够通过字符串或类定义，但基于类的异常可以让脚本建立异常类型、继承行为以及附加状态信息。现在，基于类的异常比字符串更好，到了Python 3.0这会变成必须的。

```
>>> class Bad(Exception): pass
...
>>> def doomed(): raise Bad()
...
>>> try:
...     doomed()
... except Bad:
...     print 'got Bad'
...
got Bad
>>>
```

最后，try语句可以包含finally代码块。try/finally的组合，可以定义一定会在最后执行时的收尾行为，无论try代码块中是否发生了异常。

```
>>> try:
...     fetcher(x, 3)
... finally:
...     print 'after fetch'
...
'm'
after fetch
```


www.TopSage.com

在这里，如果try代码块完成后没有异常，finally代码块就会执行，而程序会在整个try后继续下去。在这个例子中，这条语句似乎有点笨：我们似乎也可以直接在函数调用后输入print，从而完全跳过try：

```
fetcher(x, 3)
print 'after fetch'
```

不过，这样编写会存在一个问题：如果函数调用引发了异常，就永远到不了print。try/finally组合可避免这种缺点：一旦异常确实在try代码块中发生时，当程序被层层剥开，将会执行finally代码块。

```
>>> def after():
...     try:
...         fetcher(x, 4)
...     finally:
...         print 'after fetch'
...     print 'after try?'
...
>>> after()
after fetch
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in after
  File "<stdin>", line 2, in fetcher
IndexError: string index out of range
```

在这里，我们没有看到“after try?”消息，因为当异常发生时，控制权在try/finally代码块后中断了。与其相对比的是，Python跳回去执行finally的行为，然后把异常向上传播到前一个处理器（在这个例子中，就是顶层的默认处理器）。如果我们修改这个函数中的调用，使其不触发异常，则finally程序代码依然会执行，但程序就会在try后继续运行。

```
>>> def after():
...     try:
...         fetcher(x, 3)
...     finally:
...         print 'after fetch'
...     print 'after try?'
...
>>> after()
after fetch
after try?
>>>
```

在实际应用中，try/except的组合可用于捕捉异常并从中恢复，而try/finally的组合则很方便，可以确保无论try代码块内的代码是否发生了任何异常，终止行为一定会运行。例如，可能使用try/except来捕捉从第三方库导入的代码所引发的错误，然后以

www.TopSage.com

try/finally来确保关闭文件，或者终止服务器连接的调用等行为一定会执行。这一部分稍后会看到实际应用的例子。

虽然从概念上讲是用于不同的用途，但是，在Python 2.5中，我们可以在同一个try语句内混合except和finally分句：finally一定会执行，无论是否有异常引发，而且也不管异常是否被except分句捕捉到。

这就是异常的主线，它其实是一个简单的工具。在这一部分的其余内容中，我们会填加所涉及到的语句的一些细节，查看其他可出现在try中的分句种类，以及讨论基于字符串的和基于类的异常对象。

Python异常是高级的控制流程机制，可以由Python或程序引发。无论是哪种情况，它们也是可以忽略的（触发默认错误消息），或者由try语句进行捕捉（由程序代码处理）。try语句有两种逻辑格式，在Python 2.5时，可合并成一个：一种是处理异常，而另一种是执行最终的代码（无论异常是否发生）。Python的raise和assert语句可按照需要触发异常。头脑中有了这样的宏观概念后，让我们深入探讨这些语句的一般形式吧。

try/except/else语句

下列讨论中，本书把try/except/else和try/finally当成独立的语句进行介绍，因为它们是不同角色，在Python 2.5以前都无法合并。就像你所见到的一样，在Python 2.5中，except和finally可以混在一个try语句中，分别探索过这两种原始形式后，再说明这种改变的含义。

try是复合语句，它的最完整的形式如下所示。首先是以try作为首行，后面紧跟着（通常）缩进的语句代码，然后是一或多个except分句来识别要捕捉的异常，最后是一个可选的else分句。try、except以及else这些关键字会缩进在相同的层次（也就是垂直对齐）。为了方便参考，以下是其一般格式。

```
try:
    <statements>          # Run this action first
except <name1>:
    <statements>          # Run if name1 is raised during try block
except <name2>, <data>:
    <statements>          # Run if name2 is raised, and get extra data
except (name3, name4):
    <statements>          # Run if any of these exceptions occur
except:
    <statements>          # Run for all (other) exceptions raised
else:
    <statements>          # Run if no exception was raised during try block
```


www.TopSage.com

在这个语句中，try首行底下的代码块代表此语句的主要动作：试着执行的程序代码。except分句定义try代码块内引发的异常的处理器，而else分句（如果编写了的话）则是提供没发生异常时要执行的处理器。在这里的<data>元素和raise语句功能有关，本章稍后会进行讨论。

以下是try语句的运行方式。当try语句启动时，Python会标识当前的程序环境，这样一来，如果有异常发生时，才能返回这里。try首行下的语句会先执行。接下来会发生什么事情，取决于try代码块语句执行时是否引发异常。

- 如果try代码块语句执行时发生了异常，Python就跳回try，执行第一个符合引发的异常的except分句下面的语句。当except代码块执行后（除非except代码块引发了另一异常），控制权就会到整个try语句后继续执行。
- 如果异常发生在try代码块内，没有符合的except分句，异常就会向上传递到程序中的之前进入的try中，或者到这个进程的顶层（这会使Python终止这个程序并打印默认的错误消息）。
- 如果try首行底下执行的语句没有发生异常，Python就会执行else行下的语句（如果有的话），控制权会在整个try语句下继续。

换句话说，except分句会捕捉try代码块执行时所发生的任何异常，而else分句只在try代码块执行时不发生异常才会执行。

except分句是专注于异常处理器的：捕捉只在相关try代码块中的语句所发生的异常。尽管这样，因为try代码块语句可以调用写在程序其他地方的函数，异常的来源可能在try语句自身之外。第29章探索try嵌套化时，会再多介绍一些关于这方面的内容。

try语句分句

编写try语句时，有一些分句可以在try语句代码块后出现。表27-1列出所有可能形式：至少会使用其中一种。本书已经介绍过一些表27-1列出的形式：正如你所知道的那样，except分句会捕捉异常，finally分句最后一定会执行，而如果没遇上异常，else分句就会执行。从语法上来讲，except分句数目没有限制，但是应该只有一个else。在Python 2.4中，finally必须单独出现（没有else或except），其实这是不同的语句。然而，从Python 2.5开始，finally可出现在except和else所在相同try语句中了。

www.TopSage.com

表27-1：try语句分句形式

分句形式	说明
except:	捕捉所有（其他）异常类型
except name:	只捕捉特定的异常
except name, value:	捕捉所列的异常和其额外的数据（或实例）
except (name1, name2):	捕捉任何列出的异常
except (name1, name2), value:	捕捉任何列出的异常，并取得其额外数据
else:	如果没有引发异常，就运行
finally:	总是会运行此代码块

当我们见到raise语句时，就会探索具有额外数据的部分。表27-1中第一和第四项是新的。

- except分句没列出异常名称（except:）时，捕捉没在try语句内预先列出的所有异常。
- except分句以括号列出一组异常[except (e1, e2, e3):]会捕捉任何所列出异常。

因为Python会从头到尾查找except分句，在某个try中寻找是否有相符者，括号版本就像是每个异常列在其except分句内，但是语句主体只需编写一次而已。以下是多个except分句的例子，示范处理器的具体化。

```
try:
    action()
except NameError:
    ...
except IndexError
    ...
except KeyError:
    ...
except (AttributeError, TypeError, SyntaxError):
    ...
else:
    ...
```

在这个例子中，如果action函数执行时，引发了异常，Python会回到try，并搜索第一个和异常名称相符的except。Python会从头到尾以及由左至右查看except分句，然后执行第一个相符的except下的语句。如果没有符合的，异常会向这个try外传递。注意：只有当action中没有发生异常时，else才会执行，当没有相符except的异常发生时，则不会执行。

如果想要编写通用的“捕捉一切”分句，空的except就可以做到。

www.TopSage.com

```
try:
    action()
except NameError:
    ...                     # Handle NameError
except IndexError:
    ...                     # Handle IndexError
except:
    ...                     # Handle all other exceptions
else:
    ...                     # Handle the no-exception case
```

空的except分句是一种通用功能：因为这是捕捉任何东西，可让处理器通用化或具体化。在某些场合下，比起列出的try中所有可能异常来说，这种形式反而更方便一些。例如，下面是捕捉一切，但没列出任何事件的例子。

```
try:
    action()
except:
    ...                     # Catch all possible exceptions
```

不过，空except也会引发一些设计的问题：尽管方便，也可能捕捉和程序代码无关、意料之外的系统异常，而且可能意外拦截其他处理器的异常。例如，在Python中，即便是系统离开调用，也会触发异常，而你通常会想让这些事件通过。这一部分末尾会再谈这个陷阱。就目前而言，要小心使用。

注意：在Python 3.0中，表27-1的第三行会改变——except name, value:会改写成except name as value:。这样的变动是为了排除使用tuple写出的各种异常时所引发的语法困扰：表27-1的第四行在3.0时不再需要括号了。这样的变动也修改了作用域规则：有了新的as语法，except代码块末尾的value变量会被删除。

此外，在Python 3.0中，raise语句的raise E, V需要改写为raise E(V)，来生成要引发的类实例。前者形式保留下来只是为了和Python 2.x的基于字符串的异常兼容而已（参考本章稍后有关raise的更多细节以及下一章对基于类的异常的讨论）。

虽然无法在Python 2.x中使用as形式的except来保证程序在以后适用，但是Python 3.0分发时带有的“2to3”转换工具，会自动为已有的2.x程序代码转换except和raise。

---

try/else分句

Python新手无法一眼看出else分句的用途。不过，如果没有else，是无法知道控制流程通过try语句时，是因为没有异常引发，还是因为异常发生且已被处理过。

```
try:
    ...run code...
except IndexError:
    ,
```


www.TopSage.com

大家网
TopSage.com

```
    ...handle exception...
# Did we get here because the try failed or not?
```

就像循环内的else分句让离开原因更为明显，else分句也为try中提供了让所发生的事情更为明确而不模糊的语法。

```
try:
    ...run code...
except IndexError:
    ...handle exception...
else:
    ...no exception occurred...
```

把程序移进try代码块中，也能模拟else分句。

```
try:
    ...run code...
    ...no exception occurred...
except IndexError:
    ...handle exception...
```

不过，这可能造成不正确的异常分类。如果“没有异常发生”这个行为触发了IndexError，就会视为try代码块的失败，因此错误地触发try底下的异常处理器（微妙，但是真实）改为使用明确的else分句，你可以让逻辑更为明确，保证except处理器只会因包装在try中的代码真正的失败而执行，而不是为else情况中的行为失败而执行。

例子：默认行为

因为Python中通过一个程序的控制流程，比英语更容易掌握，让我们运行一些例子，进一步示范异常的基础知识。前文已经提到过，try语句没有捕捉的异常会向上传递到Python进程的顶层，并执行Python默认异常处理逻辑（也就是说，Python终止执行中的程序，并打印标准错误消息）。让我们看一个例子。执行下列模块*bad.py*来产生一个除以零的异常。

```
def gobad(x, y):
    return x / y

def gosouth(x):
    print gobad(x, 0)

gosouth(1)
```


www.TopSage.com

因为程序忽略它触发的异常，Python会终止这个程序，打印一个消息（注3）。

```
% python bad.py
Traceback (most recent call last):
  File "bad.py", line 7, in <module>
    gosouth(1)
  File "bad.py", line 5, in gosouth
    print gobad(x, 0)
  File "bad.py", line 2, in gobad
    return x / y
ZeroDivisionError: integer division or modulo by zero
```

消息中包括了堆栈跟踪和抛出的异常名称（以及任何额外的数据）。堆栈跟踪按照从旧到新的顺序列出异常发生时激活状态下的所有程序的行。因为我们不是在交互模式提示符下工作，这个例子中，文件和行号信息都有用。例如，在这里我们可以看见跟踪中最后一项发生了错误的除法：文件*bad.py*的第2行，即return语句。

因为Python会在运行时检测所有错误，引发异常并报告，一般来说，异常和错误处理及调试的想法紧密结合起来。如果你做过本书例子，在过程中显然会看到过一两个异常：当文件导入或执行时（当编译器在执行时），即使是输入错误通常也会产生SyntaxError或其他异常。默认情况下，你会得到像上面那样有用的错误显示，有助于跟踪问题。

通常来说，这个标准错误消息，就是解决程序代码中问题所需的一切。就更大型的调试工作而言，可以用try语句捕捉异常，或者使用第29章介绍的调试工具，例如，pdb标准库模块。

例子：捕捉内置异常

Python的默认异常处理通常就是你想要的：尤其是对顶层脚本文件内的代码，错误通常应该会立刻终止程序。就许多程序而言，没有必要再更加地明确代码中的错误。

尽管这样，你偶尔会想捕捉错误并从中恢复。如果不想在Python引发异常时造成程序终止，只要把程序逻辑包装在try中进行捕捉就行了。这是网络服务器这类程序很重要的功能，因它们必须不断持续运行下去。例如，下列程序代码在Python引发TypeError时就立刻予以捕捉并从中恢复，当时正试着把列表和字符串给链接起来（+运算符预期的是两边都是相同类型的序列）。

注3：　错误消息和堆栈跟踪的文字可能随时间不同而略有不同。如果你的错误消息和本书的不同，也别害怕。例如，在Python 2.5的IDLE中执行这个例子时，错误消息正文在文件名中显示完整的目录路径。

www.TopSage.com

```
def kaboom(x, y):
    print x + y                 # Trigger TypeError

try:
    kaboom([0,1,2], "spam")
except TypeError:               # Catch and recover here
    print 'Hello world!'
print 'resuming here'           # Continue here if exception or not
```

当异常在函数kaboom中发生时，控制权会跳至try语句的except分句，来打印消息。因为像这样异常捕捉后就“死”了，程序会继续在try后运行，而不是被Python终止。事实上，程序代码处理并清理了错误。

注意：一旦捕捉了错误，控制权会在捕捉的地方继续下去（也就是在try之后），没有直接的方式可以回到异常发生的地方（在这里，就是函数kaboom中）。总之，这会让异常更像是简单的跳跃，而不是函数调用：没有办法回到触发错误的代码。

try/finally语句

try语句的另一种形式是特定的形式，和最终动作有关。如果在try中包含了finally分句，Python一定会在try语句后执行其语句代码块，无论try代码块执行时发生了异常没有。其一般形式如下所示。

```
try:
    <statements>                # Run this action first
finally:
    <statements>                # Always run this code on the way out
```

利用这个变种，Python可先执行try首行下的语句代码块。接下来发生的事情，取决于try代码块中是否发生异常。

- 如果try代码块运行时没有异常发生，Python会跳至执行finally代码块，然后在整个try语句后继续执行下去。
- 如果try代码块运行时有发生异常，Python依然会回来运行finally代码块，但是接着会把异常向上传递到较高的try语句或顶层默认处理器。程序不会在try语句下继续执行。也就是说，即使发生了异常，finally代码块还是会执行的，和except不同的是，finally不会终止异常，而是在finally代码块执行后，一直处于发生状态。

当想确定某些程序代码执行后，无论程序的异常行为如何，有个动作一定会发生，那么，try/finally形式就很有用。在实际应用中，这可以让你定义一定会发生的清理动作，例如，文件关闭以及服务器断开连接等。

www.TopSage.com

在Python 2.4和更早版本中，finally分句无法和except、else一起用在相同的try语句内，所以，如果用的是旧版，最好把try/finally想成是独特的语句形式。然而，到了Python 2.5，finally可以和except及else出现在相同语句内，所以现在其实只有一个try语句，但是有许多选用的分句（等一下会介绍）。不过，无论选用哪个版本，finally分句依然具有相同的用途：指明一定要执行的“清理”动作，无论异常发生了没有。

注意： 在Python 2.6中，with语句及其环境管理器提供的基于对象式的方式，可替推出动作做类似的事。这个语句也支持进入动作（entry action）。

例子：利用try/finally编写终止行为

我们之前看过了简单的try/finally的例子。以下是更为实际的例子，示范了这个语句的典型角色。

```
class MyError(Exception): pass

def stuff(file):
    raise MyError()

file = open('data', 'w')      # Open an output file
try:
    stuff(file)               # Raises exception
finally:
    file.close()              # Always close file to flush output buffers
...                           # Continue here only if no exception
```

在这段代码中，在带有finally分句的try中包装了一个文件处理函数的调用，以确保无论函数是否触发异常，该文件总是会关闭。这样以后的代码就可确定文件的输出缓存区的内容已经从内存转移至磁盘了。类似的代码结构可以保证服务器连接已关闭了。

这个特定的函数并没那么有用（只是引发异常），但是把调用包装在try/finally语句中是确保关闭活动（即终止）一定会执行的绝佳方式。同样地，Python一定会执行finally代码块的代码，无论try代码块中是否发生异常（注4）。

当这里的函数引发异常时，控制流程会跳回，执行finally代码块并关闭文件。然后，异常要么会传递到另一个try，要么就是传递至默认的顶层处理器（打印标准错误消息并关闭程序）；绝不会运行到try后的语句。如果在这里的函数没有引发异常，程序依

注4： 当然，除非Python彻底崩溃。Python努力避免这种事的发生，在程序执行时会检查所有可能的错误。当一个程序崩溃，通常是因为连结的C扩展代码中的bug，而这已在Python范围之外了。

www.TopSage.com

然会执行finally代码块来关闭文件，但是，接着就是继续运行整个try语句之后的语句了。

此外，在这里的用户定义异常依然是通过类定义的：就像下一章将要见到的，如今的异常应该都是类的实例。

统一try/except/finally

在Python 2.5发布以前的（离它的第一个版本差不多有15年左右的时间了）所有Python版本中，try语句都有两种形式，而且是独立的两种语句：我们可以使用finally来确保清理代码一定会执行，或者编写except代码块来捕捉和恢复特定的异常，此外，如果没有异常发生的话，还能定义选用的else分句，执行其中的语句。

也就是说，finally分句无法和except和else混合。一部分原因是实现的问题，而一部分原因是合并两者的意义似乎令人费解：捕捉和恢复异常并执行清理动作似乎是毫不相关的概念。

不过，在Python 2.5中（本书所用的Python版本），这两个语句已经合并。现在，我们可以在同一个try语句中混合finally、except以及else分句。也就是说，我们现在可以编写下列形式的语句：

```
try:
    main-action
except Exception1:
    handler1
except Exception2:
    handler2
...
else:
    else-block
finally:
    finally-block
```

就像往常一样，这个语句中的*main-action*代码块会先执行。如果该程序代码引发异常，那么所有except代码块都会逐一测试，寻找与抛出的异常相符的语句。如果引发的异常是Exception1，则会执行*handler1*代码块；如果引发的异常是*Exception2*，则会执行*handler2*代码块，以此类推。如果没有引发异常，将会执行*else-block*。

无论之前发生了什么，当*main-action*代码块完成时，而任何引发的异常都已处理后，*finally-block*就会执行。事实上，即使异常处理器或者*else-block*内有错误发生而引发了新的异常，*finally-block*内的程序代码依然会执行。

www.TopSage.com

就像往常一样，finally分句并没有终止异常：当*finally-block*执行时，如果异常还存在，就会在*finally-block*代码块执行后继续传递，而控制权会跳至程序其他地方（到另一个try，或者默认的顶层处理器）。如果finally执行时，没有异常处于激活状态，控制权就会在整个try语句之后继续下去。结果就是无论发生如下哪种情况，finally一定会执行。

- main-action中是否发生异常并处理过。
- main-action中是否发生异常并没有处理过。
- main-action中是否没有发生异常。
- 任意的处理器中是否引发新的异常。

finally用于定义清理动作，无论异常是否引发或受到处理，都一定会在离开try前运行。

通过嵌套合并finally和except

在Python 2.5之前，实际上在try中合并finally和except分句是可能的，也就是在try/finally语句的try代码块嵌套try/except（第29章会更全面的探索这门技术）。实际上，下列写法和上一节所展示的合并后的新形式有相同的效果。

```
try:
    try:
        main-action
    except Exception1:
        handler1
    except Exception2:
        handler2
    ...
    else:
        no-error
finally:
    clean-up
```

异地时，finally代码块一定会执行，无论main-action发生什么，也无论嵌套的try中执行了什么样的异常处理器（看一看前四种情况来了解为什么执行的结果相同）。然而，这种对等的形式比较难懂，而且与新的合并形式相比需要更多的代码。在同一个try语句中合并比较容易编写和读，因此，更倾向于使用目前的技术。

www.TopSage.com

合并try的例子

以下示范了合并的try语句的执行情况。下面编写了四种常见场景，通过print语句来说明其意义。

```
print '-' * 30, '\nEXCEPTION RAISED AND CAUGHT'
try:
    x = 'spam'[99]
except IndexError:
    print 'except run'
finally:
    print 'finally run'
print 'after run'

print '-' * 30, '\nNO EXCEPTION RAISED'
try:
    x = 'spam'[3]
except IndexError:
    print 'except run'
finally:
    print 'finally run'
print 'after run'

print '-' * 30, '\nNO EXCEPTION RAISED, ELSE RUN'
try:
    x = 'spam'[3]
except IndexError:
    print 'except run'
else:
    print 'else run'
finally:
    print 'finally run'
print 'after run'

print '-' * 30, '\nEXCEPTION RAISED BUT NOT CAUGHT'
try:
    x = 1 / 0
except IndexError:
    print 'except run'
finally:
    print 'finally run'
print 'after run'
```

当这段代码执行时，会产生下面的输出。看一看代码，来了解异常处理是如何产生这个测试的每种输出的。

```
------------------------------
EXCEPTION RAISED AND CAUGHT
except run
finally run
after run
------------------------------
```


www.TopSage.com

```
NO EXCEPTION RAISED
finally run
after run
------------------------------
NO EXCEPTION RAISED, ELSE RUN
else run
finally run
after run
------------------------------
EXCEPTION RAISED BUT NOT CAUGHT
finally run

Traceback (most recent call last):
  File "C:/Python25/mergedexc.py", line 32, in <module>
    x = 1 / 0
ZeroDivisionError: integer division or modulo by zero
```

这个例子使用main-action里的内置表达式，来触发异常（或不触发），而且利用了Python总是会在代码运行时检查错误的事实。下一节说明如何手动引发异常。

raise语句

要故意触发异常，可以使用raise语句，其一般形式相当简单。raise语句的组成是：raise关键字，后面跟着要引发的异常名称（选用），以及一个可选的额外数据项，可随着异常传递：

```
raise <name>                # Manually trigger an exception
raise <name>, <data>        # Pass extra data to catcher too
raise                       # Re-raise the most recent exception
```

第二种形式可随着异常传递额外的数据项，从而为处理器提供细节。在raise语句中，数据是列在异常名称的后面的；回try语句中，取得该数据是通过引入一个进行接收它的变量实现的。例如，如果try引入一个except name, X:语句，则变量X就会被赋值为raise内所列出的额外的数据项。第三种raise形式只是重新引发当前的异常。如果你想把刚捕捉的异常传递给另一个处理器，它就很方便了。

那么，异常名称是什么呢？它可以是内置作用域中的内置异常的名称（例如，IndexError），或者在程序中赋值的任意字符串对象的变量名。此外，也可以引用用户定义的类或类实例：可进一步让raise语句格式通用化的一种可能性。下一章介绍基于类的异常后，再谈论这个通用化的细节。

无论如何命名异常，它都是被普通对象识别的，而且任何时刻最多只有一个会处于激活状态。一旦被程序中任意的except分句捕捉，异常就死了（也就是说，不会传递给另一个try），除非又被另一个raise语句或错误所引发。

www.TopSage.com

例子：引发并捕捉用户定义的异常

Python程序可以使用raise语句触发内置和用户定义的异常。用户定义的异常现在应该是类实例对象，例如，下列代码中调用MyBad所创建的对象。

```
class MyBad: pass

def stuff():
    raise MyBad()              # Trigger exception manually

try:
    stuff()                    # Raises exception
except MyBad:
    print 'got it'             # Handle exception here
...                            # Resume execution here
```

这一次，raise是发生在函数内的，但是这并没有差别：控制权立刻跳回到except代码块。用户定义的异常是由try语句捕捉，就像内置的异常一样。

例子：利用raise传入额外的数据

就像之前描述过的一样，raise语句可以随异常一起传递额外的数据项给处理器使用。一般而言，额外数据可以向处理器传递关于异常的背景信息。例如，如果正在写数据文件分析器，可能会在遇到错误时引发语法错误异常，此外也传入一个对象，把行号和文件信息提供给处理器（我们会在第28章遇见这样的例子）。

这是有用的，因为异常引发时，也许会跨越任意的文件边界：触发异常的raise语句以及进行捕捉的try语句，也许位于完全不同的文件中。通常来说，无法把额外细节储存在广域变量内，因为try语句可能不知道广域变量位于哪个文件中。和异常本身一起把额外的数据传递进去，可让try语句读取时，更有把握。从严格意义上来讲，每个异常都有这个额外的数据：就像函数返回值，如果没有刻意的传递什么时，就默认为特殊的None对象。下面的*raisedata.py*代码通过基于字符串的异常示范了这个概念。

```
myException = 'Error'                  # String object

def raiser1():
    raise myException, "hello"         # Raise, pass data

def raiser2():
    raise myException                  # Raise, None implied

def tryer(func):
    try:
        func()
    except myException, extraInfo:     # Run func; catch exception + data
        print 'got this:', extraInfo
```


www.TopSage.com

```
% python
>>> from raisedata import *
>>> tryer(raiser1)                    # Explicitly passed extra data
got this: hello
>>> tryer(raiser2)                    # Extra data is None by default
got this: None
```

大家网 TopSage.com

在这里，tryer函数一定会请求额外数据对象。从raiser1返回的是字符串，raiser2的raise语句则默认为None。

下一章，我们会见到同样的钩子可用于读取随基于类的异常所引发的实例：except中的变量就会赋值给引发的实例，来读取附加的状态信息，以及可调用的类方法。

例子：利用raise传递异常

raise语句不包括异常名称或额外资料值时，就是重新引发当前异常。如果需要捕捉和处理一个异常，又不希望异常在程序代码中死掉时，一般就会使用这种形式。

```
>>> try:
...     raise IndexError, 'spam'
... except IndexError:
...     print 'propagating'
...     raise
...
propagating
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
IndexError: spam
```

通过这种方式执行raise时，会重新引发异常，并将其传递给更高层的处理器，或者顶层的默认处理器，也就是停止程序，打印标准错误消息。

assert语句

Python还包括了assert语句，这种情况有些特殊。这是raise常见使用模式的语法简写，可视为条件式的raise语句。该语句形式为：

```
assert <test>, <data>          # The <data> part is optional
```

执行起来就像如下的代码。

```
if __debug__:
    if not <test>:
        raise AssertionError, <data>
```


www.TopSage.com

换句话说，如果test计算为假，Python就会引发异常：data项（如果提供了的话）是异常的额外数据。就像所有异常，引发的AssertionError异常如果没被try捕捉，就会终止程序。

assert语句是附加的功能，如果使用-O Python命令行标志位，就会从程序编译后的字节码中移除，从而优化程序。AssertionError是内置异常，而__debug__标志位是内置变量名，除非有使用-O旗号，否则自动设为1（真值）。

例子：收集约束条件（但不是错误）

Assert语句通常是用于验证开发期间程序状况的。显示时，其错误消息正文会自动包括源代码的行信息，以及列在assert语句中的值。考虑文件*asserter.py*。

```
def f(x):
    assert x < 0, 'x must be negative'
    return x ** 2

% python
>>> import asserter
>>> asserter.f(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "asserter.py", line 2, in f
    assert x < 0, 'x must be negative'
AssertionError: x must be negative
```

牢记这一点很重要：assert几乎都是用来收集用户定义的约束条件，而不是捕捉内在的程序设计错误。因为Python会自行收集程序的设计错误，通常来说是没必要写assert去捕捉超出索引值、类型不匹配以及除数为零之类的事情。

```
def reciprocal(x):
    assert x != 0                # A useless assert!
    return 1 / x                 # Python checks for zero automatically
```

这类assert一般都是多余的：因为Python会在遇见错误时自动引发异常，让Python替你把事情做好就行了（注5）。另一个assert常见用法例子，可以参考第24章的抽象超类例子。在那里，我们使用assert让未定义方法的调用失败并打印消息。

注5：至少，多数情况下是这样。就像第4部分建议的那样，如果函数必须响应不及时或无法恢复的动作，才能到达异常会被触发的地方，你可能会想亲自测试错误。不过，即使是这种情况，也要小心，别让测试过于具体或严格，不然，就会限制程序代码的用处。

www.TopSage.com

with/as环境管理器

Python 2.6（本书编写时，依然是以后版本）会引入新的异常相关的语句：with及其可选的as分句。这个语句的设计是为了和环境管理器对象（支持新的方法协议）一起工作。

简而言之，with/as语句的设计是作为常见try/finally用法模式的替代方案。就像try/finally语句，with/as语句也是用于定义必须执行的终止或“清理”行为，无论处理步骤中是否发生异常。不过，和try/finally不同的是，with语句支持更丰富的基于对象的协议，可以为代码块定义支持进入和离开动作。

Python以环境管理器强化一些内置工具，例如，自动自行关闭的文件，以及对锁的自动上锁和开锁，程序员也可以用类编写自己的环境管理器。

基本使用

这个功能一直到2.6版才成为Python正式的一部分。在Python 2.5中，默认并没有开启；必须通过本书模块部分所提到的特定的以后导入语句，才能将其开启（因为新的两个保留字with和as， 就像往常一样，这个功能也是逐步引入）：

```
from __future__ import with_statement
```

当在2.5版中执行这个导入语句时，就可以开启新的with语句，以及它的两个保留字。with语句的基本格式如下。

```
with expression [as variable]:
    with-block
```

在这里的*expression*要返回一个对象，从而支持环境管理协议（稍后会谈到这个协议的更多内容）。如果选用的as分句存在时，此对象也可返回一个值，赋值给变量名*variable*。

注意：*variable*并非赋值为*expression*的结果。*expression*的结果是支持环境协议的对象，而*variable*则是赋值为其他的东西。然后，*expression*返回的对象可在*with-block*开始前，先执行启动程序，并且在该代码块完成后，执行终止程序代码，无论该代码块是否引发异常。

有些内置的Python对象已得到强化，支持了环境管理协议，因此可以用于with语句。例如，文件对象有环境管理器，可在with代码块后自动关闭文件，无论是否引发异常。

```
with open(r'C:\python\scripts') as myfile:
    for line in myfile:
```


www.TopSage.com

```
print line
line = line.replace('spam', 'SPAM')
...more code here...
```

在这里，对open的调用，会返回一个简单文件对象，赋值给变量名myfile。我们可以用一般的文件工具来使用myfile：就此而言，文件迭代器会在for循环内逐行读取。

然而，此对象也支持with语句所使用的环境管理协议。在这个with语句执行后，环境管理机制保证由myfile所引用的文件对象会自动关闭，即使处理该文件时，for循环引发了异常。

我们不会在本书讨论Python的多进程模块（有关这个话题的更多内容，可以参考后续应用书籍，例如，《Programming Python》），那些模块所定义的锁和条件变量同步工具也支持with语句所需的环境管理协议。

```
lock = threading.Lock()
with lock:
    # critical section of code
    ...access shared resources...
```

在这里，环境管理机制保证锁会在代码块执行前获得，以及在代码块完成时释放。

decimal模块（参考第5章有关小数类型的事）也使用环境管理器来简化储存和保存当前小数配置环境（定义了赋值计算时的精度和取整的方式）。

环境管理协议

用在with语句中对象所需的接口有点复杂，而多数程序员只需知道如何使用现有的环境管理器。不过，对那些可能想写新的环境管理器的工具创造者而言，我们快速浏览其中细节吧。以下是with语句实际的工作方式。

1. 计算表达式，所得到的对象称为环境管理器，它必须有__enter__和__exit__方法。
2. 环境管理器的__enter__方法会被调用。如果as分句存在，其返回值会赋值给一个as后的变量，否则，就被丢弃。
3. 代码块中嵌套的代码会执行。
4. 如果with代码块引发异常，__exit__(*type*, *value*, *traceback*)方法就会被调用（带有异常细节）。这些也是由sys.exc_info返回的相同值（Python手册和本书这部分稍后会做说明）。如果此方法返回值为假，则异常会重新引发。否则，异常会终止。正常情况下异常是应该被重新引发，这样的话才能传递到with语句外。

www.TopSage.com

5. 如果with代码块没有引发异常，__exit__方法依然会被调用，其*type*、*value*以及*traceback*参数都会以None传递。

为什么要在意：错误检查

了解异常是有多么有用的方法之一就是，比较Python以及没有异常的语言的代码风格。例如，如果想以C语言编写健壮的程序，一般得在每个可能出错的运算之后测试返回值或状态码，然后在程序执行时传递测试结果。

```
doStuff()
{                                       # C program
    if (doFirstThing() == ERROR)        # Detect errors everywhere
        return ERROR;                   # even if not handled here
    if (doNextThing() == ERROR)
        return ERROR;
    ...
    return doLastThing();
}

main()
{
    if (doStuff() == ERROR)
        badEnding();
    else
        goodEnding();
}
```

实际上，现实的C程序中，通常用于处理错误检测和用于实际工作的代码数量相当。但是，在Python中，你就不用那么谨小慎微和神经质。你可以把程序的任意片段包装在异常处理器内，然后编写从事实际工作的部分，假设一切都工作正常。

```
def doStuff():          # Python code
    doFirstThing()      # We don't care about exceptions here
    doNextThing()       # so we don't need to detect them
    ...
    doLastThing()

if __name__ == '__main__':
    try:
        doStuff()       # This is where we care about results
    except:             # so it's the only place we must check
        badEnding()
    else:
        goodEnding()
```

因为控制权在异常发生时就会立刻跳到处理器，没必要让所有代码都去预防错误的发生。再者，因为Python会自动检测错误，程序代码通常不需要事先检查错误。重点在于，异常让你大致上可以忽略罕见情况，并避免编写错误检查程序代码。

www.TopSage.com

让我们来看这个协议的示范。下面定义一个环境管理器对象，跟踪其所用的任意一个with语句内with代码块的进入和退出。

```
from __future__ import with_statement          # Required in Python 2.5

class TraceBlock:
    def message(self, arg):
        print 'running', arg
    def __enter__(self):
        print 'starting with block'
        return self
    def __exit__(self, exc_type, exc_value, exc_tb):
        if exc_type is None:
            print 'exited normally\n'
        else:
            print 'raise an exception!', exc_type
            return False  # propagate

with TraceBlock() as action:
    action.message('test 1')
    print 'reached'

with TraceBlock() as action:
    action.message('test 2')
    raise TypeError
    print 'not reached'
```

注意：这个类的__exit__方法返回Flase来传播该异常。删除那里的return语句也有相同效果，因为默认的函数返回值None，按定义也是False。此外，__enter__方法返回self，作为赋值给as变量的对象。在其他情况下，这里可能会返回完全不同的对象。

运行时，环境管理器会以__enter__和__exit__跟踪with语句代码块的进入和离开。

```
% python withas.py
starting with block
running test 1
reached
exited normally

starting with block
running test 2
raise an exception! <type 'exceptions.TypeError'>

Traceback (most recent call last):
  File "C:/Python25/withas.py", line 22, in <module>
    raise TypeError
TypeError
```

环境管理器是有些高级的机制，还不是Python的正式组成部分，所以我们在这里跳过了其他细节（参考Python的标准手册来了解细节。例如，新的contextlib标准模块提供其

www.TopSage.com

他的工具来编写环境管理器）。就较为简单的用途来说，try/finally语句可对终止活动提供足够的支持。

TopSage.com

本章小结

在这一章，我们开始讨论了异常的处理，探索Python中有关异常的语句：try是捕捉，raise是触发，assert是条件式引发，而with是把代码块包装在环境管理器中（定义了进入和离开的行为）。

到目前为止，异常看起来可能是相当简单的工具，然而事实上，的确是如此。唯一真正复杂的事就是如何去识别。下一章要说明如何自行实现异常对象，就像你将见到的那样，类可以编写比简单字符串更为有用的异常。不过，继续学习之前，先做一做本章的习题。

www.TopSage.com

本章习题

1. try语句有什么用途?
2. try语句的两个常见变种是什么?
3. raise语句有什么用途?
4. assert语句的设计来是做什么用的? 和其他哪些语句相像?
5. with/as语句设计来是做什么用的? 和其他哪些语句相像?

习题解答

1. try语句可以捕捉异常并从中恢复：定义要运行的程序代码块，一个或多个处理器，用来处理代码块运行时可能引发的各种异常。
2. try语句两个常见变种就是try/except/else（捕捉异常）以及try/finally（指明清理动作，无论异常是否发生都必须运行）。在Python 2.4中，这些是独立的语句，只能够通过语法嵌套统一起来。在2.5和后续版本中，except和finally代码块可在同一个的try语句中混合，所以这两个语句形式合并了。在合并后的形式中，finally在try结束前执行，无论是否发生了异常或是否处理了异常。
3. raise语句引发（触发）异常。Python内部会在发生错误时引发内置异常。脚本也能通过raise触发内值或用户定义的异常。
4. assert语句在条件为假时，会引发AssertionError异常。这就像是包裹在if语句中的条件式raise语句。
5. with/as语句的设计，是为了让必须在程序代码块周围发生的启动和终止活动一定会发生。和try/finally语句（无论异常是否发生其离开动作都会执行）类似，但是with/as有更丰富的对象协议，可以定义进入和离开的动作。

www.TopSage.com

第28章

异常对象

到目前为止，本书有意模糊了异常这个概念。Python把异常的概念一般化。就像上一章中所描述的，异常可以是字符串对象或类实例对象。如今更倾向于类实例对象，而且很快就会成为一种规定。两种做法都有优点，不过从维护异常层次的角度来说，类能提供更好的解决方案。

简而言之，基于类的异常可以创建各种异常的分类，有附加的状态信息，而且支持继承。详细一点讲，和旧的字符串异常模型相比，类异常有如下特点。

- 提供类型分类，对今后的修改有更好的支持：以后增加新异常时，通常不需要在try语句中进行修改。
- 提供了储存在try处理器中所使用的环境信息的合理地点：这样的话，可以拥有状态信息以及可调用的方法，并且可以通过实例进行读取。
- 允许异常参与继承层次，从而可以获得共同的行为。例如，继承的显示方法可提供通用的错误消息的外观。

因为有这些差异，基于类的异常支持了程序的演进和较大系统，从这方面讲，它远优于基于字符串的异常。当程序规模小时，字符串异常也许看起来似乎比较容易使用，但是随着程序逐步增长时，就变得越来越难用了。事实上，所有内置异常都是类组织成继承树，其原因就是刚才所说的。也可以对用户定义的异常做相同的事。

为了版本向下兼容，这里介绍字符串异常和类异常。这两种目前都能使用，但在当前的Python 2.5版，字符串异常会产生“deprecation（不建议使用）”警告，而且Python 3.0将不再支持它。我们会在此提到，是因为还是有可能在现有的代码中碰到字符串异常，今后定义的新异常应该是用类来编写。一部分原因是因为类更好，也是因为当Python 3.0问世时，可能就不用再修改的异常代码了。

www.TopSage.com

大家网
TopSage.com

基于字符串的异常

到目前为止，我们所见的例子中，用户定义的异常都是用字符串编写的。这是编写异常的较为简单的方式，如下所示。

```
>>> myexc = "My exception string"
>>> try:
...     raise myexc
... except myexc:
...     print 'caught'
...
caught
```

任何字符串值都可以识别为异常。从技术角度来说，异常是字符串对象，而不是字符串值：必须使用相同的变量引发和捕捉异常（在第7部分总结陷阱时，再扩展这个概念）。在这里，异常名称myexc只是普通变量：可通过导入某个模块等方式来实现。字符串的文本几乎是无关紧要的，只不过是作为异常消息而打印出来而已。

```
>>> raise myexc
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
My exception string
```

注意： 如果字符串异常像这样打印出来的，可能想要使用比本书大多数例子更有意义的文字。

字符串异常就要出局了

就像前边提到的那样，基于字符串的异常依然可以使用，在Python 2.5时，会产生警告，而且计划在Python 3.0时完全消失，如果不是更早的话。实际上，在Python 2.5 IDLE下执行时，上一个程序的实际输出如下所示。

```
>>> myexc = 'My exception string'
>>> try:
        raise myexc
    except myexc:
        print 'caught'

Warning (from warnings module):
  File "__main__", line 2
DeprecationWarning: raising a string exception is deprecated
caught
```

可以关闭这类警告，不过产生这类消息是让我们知道，以后使用字符串异常时会当成错误。因此，将会完全禁止使用。本书介绍的字符串异常的内容只是要了解以前所编写的

www.TopSage.com

代码。如今，所有的内置异常都是类实例，而创建的所有的用户定义的异常，也应该是基于类的。下一节会解释其原因。

基于类的异常

字符串是定义异常的简单方式。然而，就像前边描述的一样，类多了一些优点。最主要的是，类可让你组织的异常分类，比起简单的字符串而言，使用和维护起来更灵活。再者，类可附加异常的细节，而且支持继承。因为类是更好的办法，很快也会变成规定的做法。

先不管编写代码的细节，字符串异常和类异常的主要差别在于，引发的异常在try语句中的except分句匹配时的方式不同。

- 字符串异常是以简单对象识别来匹配的：引发的异常是由Python的is测试（不是==）来匹配except分句的。
- 类异常是由超类关系进行匹配的：只要except分句列举了异常的类，或其任何超类名，引发的异常就会匹配该分句。

也就是说，当try语句的except分句列出一个超类时，就可以捕捉该超类的实例，以及类树中所有较低位置的子类的实例。结果就是，类异常支持异常的层次的架构：超类变成分类的名称，而子类变成这个分类中定种类的异常。except分句列出一个通用的异常超类，就可捕捉整个分类中的各种异常：任何特定的子类都可匹配。

除了这种类型想法外，基于类的异常也更好地支持了异常状态信息（附加在实例上），而且可以让异常参与继承层次（从而获得通用的行为）。比起基于字符串的异常来说，只需多一点代码，就可换得更为强大的替代方案。

类异常例子

让我们看一个例子，看看在代码中类异常是如何应用的。下列*classexc.py*文件中，我们定义一个名为General的超类，以及两个子类Specific1和Specific2。这个例子说明异常分类的概念：General是分类的名称，而其两个子类是这个分类中特定种类的异常。捕捉General的处理器也会捕捉其任何子类，包括Specific1和Specific2。

```
class General:           pass
class Specific1(General): pass
class Specific2(General): pass
```


www.TopSage.com

```
def raiser0():
    X = General()           # Raise superclass instance
    raise X

def raiser1():
    X = Specific1()         # Raise subclass instance
    raise X

def raiser2():
    X = Specific2()         # Raise different subclass instance
    raise X

for func in (raiser0, raiser1, raiser2):
    try:
        func()
    except General:         # Match General or any subclass of it
        import sys
        print 'caught:', sys.exc_info()[0]

C:\python> python classexc.py
caught: __main__.General
caught: __main__.Specific1
caught: __main__.Specific2
```

我们会在下一章再谈这里所用到的sys.exc_info调用：这是一种抓取最近发生异常的常用的方式。简而言之，对基于类的异常而言，其结果中的第一个元素就是引发异常类，而第二个是实际引发的实例。除了用此方法能唯一确保捕捉所有空except分句内发生的事的方法。

我们调用类来创建raise语句的实例。本节稍后提出raise语句形式规则后，引发基于类的异常时，一定要有一个实例。这段代码也包含一些函数，引发三个类实例使其成为异常，此外，有个顶层try会调用那些函数，并捕捉General异常（同一个try也会捕捉两个特定的异常，因为他们是General的子类）。

注意：目前Python的说明文件指出，用户定义的异常类最好继承自Exception内置的异常（但并不是必须要求的）。为此，我们可以把*classexc.py*文件的第一行改写如下。

```
class General(Exception): pass
class Specific1(General): pass
class Specific2(General): pass
```

虽然这并不是必须要求的，而独立的异常类如今也能够正常工作，但是在Python中的倾向都会随时间的推移逐步变成必须要求的。如果想要代码向后兼容，还是根超类继承Exception，就像这里展示的那样。通过继承，也让类获得一些免费的有用接口和工具。例如，Exception类有__init__构造器逻辑，自动把构造参数附加到类实例上去。

www.TopSage.com

为什么使用类异常

因为上一节例子中，只有三种可能的异常，其实无法说明类异常的用处。事实上，我们可以在except分句的括号内，字符串异常名称的清单，从而可以达到相同效果。文件*stringexc.py*示范了其做法。

```
General  = 'general'
Specific1 = 'specific1'
Specific2 = 'specific2'

def raiser0(): raise General
def raiser1(): raise Specific1
def raiser2(): raise Specific2

for func in (raiser0, raiser1, raiser2):
    try:
        func()
    except (General, Specific1, Specific2):      # Catch any of these
        import sys
        print 'caught:', sys.exc_info()[0]

C:\python> python stringexc.py
caught: general
caught: specific1
caught: specific2
```

然而，就大型或多层次的异常而言，在一个except分句中使用类捕捉分类，会比列出一个分类中的每个成员更为简单。此外，可以新增子类扩展异常层次，而不会破坏现有的代码。

假设用Python编写了一个数值计算库，可供许多人使用。当编写库时，有两件事会让代码中的数值出错：除数为零以及数值溢出。在文档中指出这些是异常，可以通过库引发这两个异常，同时在代码中将异常定义为简单字符串。

```
# mathlib.py

divzero = 'Division by zero error in library'
oflow   = 'Numeric overflow error in library'
...
def func():
    ...
    raise divzero
```

现在，当人们使用库的时候，他们一般会在try语句内，把对函数或类的调用包装起来，从而捕捉两个异常（如果他们没捕捉异常，库的异常会终止代码）。

```
# client.py
```


www.TopSage.com

```
import mathlib
...
try:
    mathlib.func(...)
except (mathlib.divzero, mathlib.oflow):
    ...report and recover...
```

这样运作起来没有什么问题，许多人开始使用你的库。然而，发布了六个月后，你做了些修改。在这个过程中，你发现有新的情况也会出错：退位（underflow），于是，新增了一个字符串异常。

```
# mathlib.py

divzero = 'Division by zero error in library'
oflow   = 'Numeric overflow error in library'
uflow   = 'Numeric underflow error in library'
```

不幸的是，当你发布代码时，就给用户创造了一个维护问题。如果他们要明确地列出你的异常，现在就得回去修改每处调用你的库的地方，来引入新增的异常名。

```
# client.py

try:
    mathlib.func(...)
except (mathlib.divzero, mathlib.oflow, mathlib.uflow):
    ...report and recover...
```

这也许不是世界末日。如果你的库只是给自己使用，可以自己修改。你也可以发布一个Python脚本，试着自动修改这类代码（可能只有几十行，至少在一段时间内能猜测正确）。不过，如果每次修改异常集后，就有许多人得修改他们的代码，这就不是一个最合理的升级策略了。

用户可能为了避免这种麻烦，而编写了空的except分句来捕捉所有的可能的异常。

```
# client.py

try:
    mathlib.func(...)
except:                              # Catch everything here
    ...report and recover...
```

但是，这种权宜之计可能捕捉到不想要捕捉的异常：像变量名输入错误、内存错误以及系统离开时触发的异常，然而，你想让这些异常通过，而非得捕捉到误以为是库错误。

www.TopSage.com

基本原则就是，在异常处理器中，通常来说具体要优于一般（下一章陷阱一节会再谈这个概念）（注1）。

那么，该怎么做？类异常可以完全修复这种难题。不是把你的库的异常定义成简单的一组字符串，而是安排到类树中，有个共同的超类来包含整个类型。

```
# mathlib.py

class NumErr(Exception): pass
class Divzero(NumErr): pass
class Oflow(NumErr): pass
...
def func():
    ...
    raise DivZero()
```

这样的话，你的库用户只需列出共同的超类（也就是分类），来捕捉库的所有异常，无论是现在还是以后。

```
# client.py

import mathlib
...
try:
    mathlib.func(...)
except mathlib.NumErr:
    ...report and recover...
```

当你回来修改代码时，作为共同超类的新的子类来增加的新异常。

```
# mathlib.py

...
class Uflow(NumErr): pass
```

结果就是用户代码捕捉库的异常依然保持正常工作。事实上，你可在未来任意新增、删除以及修改异常，只要客户端使用的是超类的名称，就和异常集中的修改无关。换句话说，比起字符串，对于维护的问题来说，类异常提供了更好的答案。再者，基于类的异

注1：有一位聪明的学生建议，库模块也可以提供元组对象，包含库可能引发的所有异常：然后，客户端可以导入元组，在except分句中上填入该元组的名称，从而捕捉库所有的异常（回想一下，except的元组指的是捕捉任何一个异常）。当稍后增加新异常时，库只需扩充导出的元组。这样也行得通，但是，你还是得更新元组，使其跟上库模块内可能引发的异常。此外，基于类的异常提供的优点不仅止于分类而已，也支持附加的状态信息、方法调用以及继承，而这些是字符串异常无法提供的。

www.TopSage.com

常可支持状态保留和继承，而这是字符串办不到的 —— 本章稍后会用例子说明这个概念。

TopSage.com

内置Exception类

前一节的例子并没有带来什么新的东西。尽管用户可以通过字符串或类对象定义异常，但Python本身可能引发的所有的内置异常，都预定义为类对象，而不是字符串。此外，内置异常通常通过一般的超类分类以及具体的子类形态组织成的层次，很像我们之前学习过的异常类树。

所有熟悉的异常（例如，SyntaxError）其实都是预定义的类，可以作为内置变量名（在模块__builtin__中），以及作为标准库模块exceptions的属性。此外，Python把内置异常组织成层次，来支持各种捕捉模式。

Exception
: 异常的顶层根超类。

StandardError
: 所有内置错误异常的超类。

ArithmeticError
: 所有数值错误的超类。

OverflowError
: 识别特定的数值错误的子类。

你可以在Python库手册或exceptions模块的帮助文本中（参考第4章和第14章有关help的内容）进一步阅读关于这个结构的内容。

```
>>> import exceptions
>>> help(exceptions)
...lots of text omitted...
```

内置类树可让你选择处理器具体或通用的程度。例如，内置异常ArithmeticError是如OverflowError和ZeroDivisionError这样的更为具体的异常的超类。只列出OverflowError时，就只会拦截这种特定类型的错误，而不能捕捉其他的异常。

与之相类似的是，因为StandardError是所有内置错误异常的超类，通常可以通过它在try中选择内置错误和用户定义的异常：

```
try:
    action()
```


www.TopSage.com

```
except StandardError:
    ...handle Python errors...
except:
    ...handle user exceptions...
else:
    ...handle no-exception case...
```

捕捉了根类Exception，几乎都可以模拟空except分句（捕捉一切）。然而，这样行不通，因为这样做是无法捕捉字符串异常的，而且用户定义的独立异常，目前并没有必须要求要继承Exception根类。

无论你是否使用内置类树内的分类，这都是个不错的例子。在代码中通过类异常使用相似的技术，就可提供非常灵活并且修改方便的异常集合。

除了我们所谈到的这些方面以外，内置异常和原始的基于字符串的模型几乎难以区分。事实上，通常不用在意它们是否是类，除非假设内置异常是字符串并且试着不通过转换就进行合并（例如，KeyError + "spam"会失败，但str(KeyError) + "spam"就可以）。

定义异常文本

本章开头见到是基于字符串的异常，当异常没被捕捉到时，字符串文字会显示在标准错误消息中（也就是往上传递到顶层默认的异常处理器）。但是，对于未被捕捉的类异常，这个消息包含什么呢？在默认情况下，得到的是类的名称以及被抛出的实例对象不太漂亮的显示。

```
>>> class MyBad: pass
>>> raise MyBad()

Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    raise MyBad()
MyBad: <__main__.MyBad instance at 0x00BB5468>
```

为了改进这个显示，可在类中定义__repr__或__str__显示字符串的重载方法，从而返回异常到达时想要默认处理器显示的字符串。

```
>>> class MyBad:
...     def __repr__(self):
...         return "Sorry--my mistake!"
...
>>> raise MyBad()

Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    raise MyBad()
MyBad: Sorry--my mistake!
```


www.TopSage.com

正如之前我们学到过的，这里所用的__repr__运算符重载方法，会因类实例的打印和字符串转换请求而被调用。__str__则定义了print语句的用户友好的显示（参考第24章有关显示字符串方法的内容）。

注意：如果我们继承自内置异常类，如之前所建议的那样，错误测试会有细微的改变，构造方法参数会自动储存并显示在消息中。

```
>>> class MyBad(Exception): pass
>>> raise MyBad()

Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    raise MyBad()
MyBad

>>> class MyBad(Exception): pass
>>> raise MyBad('the', 'bright', 'side', 'of', 'life')

Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    raise MyBad('the', 'bright', 'side', 'of', 'life')
MyBad: ('the', 'bright', 'side', 'of', 'life')
```

如果终端用户看到异常错误消息，你可能就想通过运算符重载定义自己的显示格式方法，就像这里所演示的那样。不过，像这样能够自动把状态信息附加在实例上，通常来说是一项实用的功能，下一节将会展开介绍。

发送额外数据和实例行为

除了支持灵活的层次外，类异常也为额外的状态信息提供了储存空间，作为实例的属性。当基于类的异常发生时，Python会自动传递异常以及类实例对象（作为额外数据项）。至于字符串异常，你可以在try语句中列出额外的变量，来读取引发的实例。这样可提供自然的钩子，从而为处理器提供了资料和行为。

例子：类和字符串的额外数据

让我们通过一个例子来探索传递额外数据的概念，并且顺便比较字符串和类这两种做法。分析数据文件的程序可能引发异常实例（填入有关错误的额外细节）从而发出格式错误的信号。

```
>>> class FormatError:
...     def __init__(self, line, file):
...         self.line = line
...         self.file = file
...
```


www.TopSage.com

```
>>> def parser():
...     # when error found
...     raise FormatError(42, file='spam.txt')
...
>>> try:
...     parser()
... except FormatError, X:
...     print 'Error at', X.file, X.line
...
Error at spam.txt 42
```

在except分句中，变量X赋值为异常引发时所产生的实例（注2）。不过，在实际应用中，这并没有比字符串异常传递复合对象（例如，元组、列表或字典）作为额外数据方便多少，而且也不是非使用基于类的异常不可的理由。以下是等效的基于字符串的异常。

```
>>> formatError = 'formatError'

>>> def parser():
...     # when error found
...     raise formatError, {'line':42, 'file':'spam.txt'}
...
>>> try:
...     parser()
... except formatError, X:
...     print 'Error at', X['file'], X['line']
...
Error at spam.txt 42
```

这一次，except分句中的变量X是指定为raise语句中列出的字典（内含额外的细节）。结果类似，但是，我们不用编写类。然而，如果异常应该有行为时，类这种方法可能就更方便些了。异常类也可定义方法，在处理器中调用。

```
class FormatError:
    def __init__(self, line, file):
        self.line = line
        self.file = file
    def logerror(self):
        log = open('formaterror.txt', 'a')
        print >> log, 'Error at', self.file, self.line

def parser():
    raise FormatError(40, 'spam.txt')
```

注2：下一章就会知道，引发的实例对象也可通过一般的sys.exc_info调用所得到的结果元组的第二项获得。这个工具会传回最近引发的异常的信息。如果在except分句中没有列出异常名称，但又得读取所发生的异常，或者其附加的状态信息或方法，就一定得使用这个接口。

www.TopSage.com

```
try:
    parser()
except FormatError, exc:
    exc.logerror()
```

在这种类内，方法（像logerror）也会继承自超类，而实例属性（像line和file）也提供了空间来储存这些提供的额外背景的状态信息，以便于在稍后的方法调用中使用。我们可以用字符串法传递简单函数来模拟这种效果，但是程序确实会增加代码的复杂度。

```
formatError = "formatError"

def logerror(line, file):
    log = open('formaterror.txt', 'a')
    print >> log, 'Error at', file, line

def parser():
    raise formatError, (41, 'spam.txt', logerror)

try:
    parser()
except formatError, data:
    data[2](data[0], data[1])          # Or simply: logerror()
```

自然，这中函数不会参与像类方法那样的继承，而且也无法在实例属性中保留状态（lambda和广域变量，通常就是我们能对于状态函数所做的最好的办法了）。当然，我们可以在基于字符串的异常的额外资料中，传入类实例来达到相同效果。如果我们费那么大的力气来模拟基于类的异常，还不如就使用类异常：不管怎样，最好还是编写类。

正如前边描述的那样，在Python以后版本中，需要类异常。但是，即使不是这样，现在进行使用，还是有很多好理由来使用它。通常意义上，基于字符串的异常用于简单任务更为简单。然而，基于类的异常可用于定义分类，而且对于得益于状态保留和属性继承的高级应用程序而言，也更倾向于使用它们。并非每个应用程序都需要OOP的力量，但是，当系统演进并扩展时，类异常的好处会变得更加明显。

raise语句的一般形式

因为多了基于类的异常，raise语句可以有下列五种形式。前两个是引发字符串异常，后两个是引发类异常，而最后一个是重新引发当前异常（如果需要传递任意的异常时，就有用处）。

```
raise string            # Matches except with same string object
raise string, data      # Passes optional extra data (default=None)

raise instance          # Same as: raise instance.__class__, instance
raise class, instance   # Matches except with this class or its superclass
```


www.TopSage.com

```
raise                    # Reraises the current exception
```

目前，第三个是最常用的。对基于类的异常而言，Python总是需要类实例。引发实例，其实就是引发该实例的类。实例是随着类一起传递的，它是作为额外数据项（正像我们见到的，这是储存处理器所需要信息的绝佳场合）。为了和内置异常为字符串的Python旧版兼容，你也可以使用下列raise语句形式。

```
raise class                  # Same as: raise class()
raise class, arg             # Same as: raise class(arg)
raise class, (arg, arg, ...) # Same as: raise class(arg, arg, ...)
```

这些都相当于raise class(arg...)，等效于之前的raise instance形式。明确地讲，如果列出的是类而不是实例，则额外数据项就不是所列类的实例，Python会自动调用类，并通过额外数据项作为构造参数，创建并引发实例。

例如，可以只写raise KeyError引发内置KeyError异常的实例，即使KeyError现在是类。Python会调用KeyError创建实例。事实上，可以使用各种方式引发KeyError以及其他基于类的异常。

```
raise KeyError()             # Normal form: raise an instance
raise KeyError, KeyError()   # Class, instance: use instance
raise KeyError               # Class: an instance will be generated
raise KeyError, "bad spam"   # Class, arg: an instance will be generated
```

就这些raise形式而言，下列try语句形式，将X赋值给了引发的KeyError实例对象。

```
try:
    ...
except KeyError, X:
    ...
```

如果这么做令人困惑，只要记住异常可作为字符串或类实例对象就可以了。对于字符串而言，你可以随着异常传递额外数据，但不传也可以。对类而言，如果raise语句中没有实例对象，Python就会创建实例。

在Python 2.5中，几乎可以忽略raise的字符串形式，因为基于字符串的异常会产生警告，在以后版本中将无法再使用。但是，作为目前为超过百万人教授如何使用一门编程语言的书来说，为了考虑与之前版本的兼容性，还是值得在这里讲一下的。

本章小结

在这一章中，介绍的是编写用户定义的异常。正如我们所学到的，异常可以由字符串对象或类实例对象实现。然而，如今更倾向于使用类实例，而这对于未来的Python版本也

www.TopSage.com

是要求的。更倾向于使用类实例是因为它支持异常的层次概念（更好的可维护性），可以让数据和行为附加在异常上作为实例的属性以及方法，而且可以让异常继承超类的数据和行为。

TopSage.com

我们看到过，在try语句中，捕捉其超类就会捕捉这个类，以及类树中超类下的所有子类：超类会变成异常分类的名称，而子类会变成该分类中特定的异常类型。我们也看过raise语句已经通用化，从而支持了各种格式，只不过如今大多数程序只会产生并引发类实例。

虽然我们在本章探索了字符串和类的比较，如果把范围锁定在Python如今鼓励的基于类的模型上，异常对象就更容易记住：每个异常都写成类，从异常树顶层继承Exception，然后，你就能忘记陈旧的基于字符串的模型。

下一章是本书以及这一部分的结尾，大部分都是探索一些异常的常见情况，以及研究Python程序员常用的工具。在学习这些内容前，做一下本章的习题。

www.TopSage.com

本章习题

1. 基于字符串的异常是怎样与处理器匹配的?
2. 基于类的异常是怎样与处理器匹配的?
3. 怎样把环境信息附加在基于类的异常上，并在处理器中使用的?
4. 怎样在基于类的异常中定义错误消息文本?
5. 如今为何不再使用基于字符串的异常?

习题解答

1. 基于字符串的异常的匹配是通过对象本身（从技术角度来讲，就是通过is表达式），而不是通过对象值（==表达式）。因此，命名为相同的字符串值是有可能行不通的；你需要编写的是相同的对象引用值（通常就是变量）。在Python中，简单字符串会被缓存并且重用，所以编写出相同的值，有时候可能可行，但是你不能依赖它（下一章末尾的陷阱会再谈这一点）。
2. 基于类的异常是由超类的关系匹配的：在异常处理器中指定超类，就会捕捉该类的实例，以及类树中任何更低的子类的实例。因此，你可以把超类想成是一般异常的分类，而子类是该分类中更具体的异常类型。
3. 把环境信息附加在基于类的异常的办法是：在引发的实例对象中填写实例的属性，通常是在类的构造器方法中。在异常处理器中，是列出要赋值为引发的实例的变量，然后通过这个变量名来读取附加的状态信息，并且调用任何继承的类方法。
4. 基于类的异常内的错误消息文本能够通过__repr__或__str__运算符重载方法定义。如果继承了内置的Exception类，传给类构造器的任何东西都会自动显示。
5. 因为Guido这么说：未来Python版本中就会消失。其实，这么做有不少好的理由：基于字符串的异常不支持分类、状态信息或行为继承，不像基于类的异常。在实际中，这使得基于字符串的异常在开始阶段更易于使用，但那是程序规模小时，一旦程序规模变大，就变得难以使用了。

www.TopSage.com

第29章

异常的设计

这一章包括了异常设计的话题以及常用例子的集合，再加上这一部分的陷阱和练习题，作为本书这一部分的收尾。因为这一章也是本书最后一章，因此也会简单介绍一下开发工具，帮助你从Python初学者转变成为Python应用开发者。

嵌套异常处理器

到目前为止，我们的例子都只使用了单一的try语句来捕捉异常，如果try中还有try，那会发生什么事情呢？就此而言，如果try调用一个会执行另一个try的函数，这代表了什么意思？从技术角度上来讲，从语法和代码运行时的控制流程来看，try语句是可以嵌套的。

如果你知道Python会在运行时将try语句放入堆栈，这两种情况就可以理解了。当发生异常时，Python会回到最近进入、具有相符except分句的try语句。因为每个try语句都会留下标识，Python可检查堆栈的标识，从而跳回到较早的try。这种处理器的嵌套化，就是我们所谈到的异常向上传递至较高的处理器的意思：这类处理器就是在程序执行流程中较早进入的try语句。

图29-1说明嵌套的try/except语句在运行时所发生的事情。进入try代码块的代码量可能很大（例如，它可能包含了函数调用），而且通常会启用正在监视相同异常的其他代码。当异常最终引发时，Python会跳回到匹配该异常、最近进入的try语句，执行该语句的except分句，然后在try语句后继续下去。

一旦异常被捕捉，其生命就结束：控制权不会跳回所有匹配这个异常、相符的try语句；只有第一个try有机会对它进行处理。如图29-1所示，函数func2中的raise语句会把控制权返还func1中的处理器，然后程序再在func1中继续下去。

www.TopSage.com

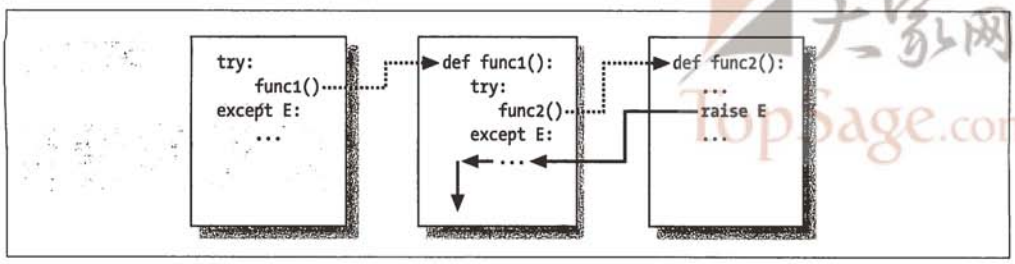


图29-1：嵌套的try/except语句：当异常引发时（由你或由Python引起），控制权会跳回具有相符的except分句、最近进入的try语句，而程序会在try语句后继续执行下去。except分句会拦截并停止异常，这里就是你处理异常并从中恢复的地方

与之相对比的是，当try/finally语句嵌套且异常发生时，每个finally代码块都会执行：Python会持续把异常往上传递到其他try语句上，而最终可能达到顶层默认处理器（标准错误消息打印器）。如图29-2所示，finally分句不会终止异常，而是指明异常传播过程中，离开每个try语句之前要执行的代码。如果异常发生时，有很多try/finally都在作用，就都会执行，除非有个try/except在这个过程中某处捕捉该异常。

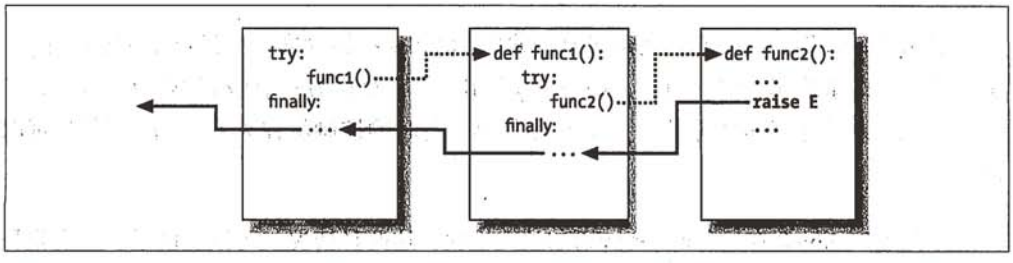


图29-2：嵌套的try/finally：当异常在这里引发时，控制权会回到最近进入的try去执行其finally语句，异常会持续传播到所有激活状态下的try语句的finally，至到最终抵达默认顶层处理器，在那里打印错误消息。finally分句会拦截（但不会停止）异常：只是定义了离开前要执行的动作而已

换句话说，引发异常时，程序去向何方完全取决于异常在何处发生：这是脚本运行时控制流程的函数，而不仅仅是其语法。异常的传递，基本上就是回到处理先前进入但尚未离开的try。只要控制权碰到相符合except分句，传递就会停止，而通过finally分句时就不会。

例子：控制流程嵌套

让我们分析一个例子，让这个嵌套的概念更为具体。下面的模块文件*nestexc.py*定义了两

www.TopSage.com

个函数。action2是写成要触发异常（做数字和序列的加法），而action1把action2调用封装在try处理器内，以捕捉异常。

```
def action2():
    print 1 + [ ]            # Generate TypeError

def action1():
    try:
        action2()
    except TypeError:        # Most recent matching try
        print 'inner try'

try:
    action1()
except TypeError:            # Here, only if action1 re-raises
    print 'outer try'

% python nestexc.py
inner try
```

那么，文件底端的顶层模块代码，也在try处理器中包装了action1调用。当action2触发TypeError异常时，就有两个激活的try语句：一个在action1内，另一个在模块文件顶层。Python会挑选并执行具有相符except、最近的try，而在这个例子中就是action1中的try。

正如我所提到过的，异常最后的所在之处，取决于程序运行时的控制流程。因此，想要知道你要去哪里，就需要知道你在哪里。就这个例子而言，异常在哪里进行处理是控制流程的函数，而不是语句的语法。然而，我们也可以用语法把异常处理器嵌套化：等一下会看一下与其等效的情况。

例子：语法嵌套化

第27章讨论新的统一后的try/except/finally语句时，就像我提到的那样，从语法上有可能让try语句通过其源代码中的位置来实现嵌套。

```
try:
    try:
        action2()
    except TypeError:        # Most recent matching try
        print 'inner try'
except TypeError:            # Here, only if nested handler re-raises
    print 'outer try'
```

其实，这段代码只是像之前的那个例子一样（行为也相同），设置了相同的处理器嵌套结构。实际上，语法嵌套的工作就像图29-1和29-2所描绘的情况一样。唯一的差别就在于嵌套处理器实际上是嵌入try代码块中，而不是写在其他被调用的函数中。例如，嵌

www.TopSage.com

套的finally处理器会因一个异常而全部启动，无论是语法上的嵌套，或者因运行时流程经过代码中某个的部分。

```
>>> try:
...     try:
...         raise IndexError
...     finally:
...         print 'spam'
... finally:
...     print 'SPAM'
...
spam
SPAM
Traceback (most recent call last):
  File "<stdin>", line 3, in ?
IndexError
```

参考图29-2有关这段代码运行的图形说明。效果是相同的，不过函数的逻辑变成了嵌套语句。有关语法嵌套更有用的例子，可以考虑下面的文件*except-finally.py*。

```
def raise1():  raise IndexError
def noraise(): return
def raise2():  raise SyntaxError

for func in (raise1, noraise, raise2):
    print '\n', func
    try:
        try:
            func()
        except IndexError:
            print 'caught IndexError'
    finally:
        print 'finally run'
```

此代码在异常引发时，会对其进行捕捉，而且无论是否发生异常，都会执行finally终止动作。这需要一点时间去掌握，但是其效果很像如今在单个try语句内结合except和finally（回想一下，在Python 2.5之前，这类组合在语法上是非法的）。

```
% python except-finally.py

<function raise1 at 0x00BA2770>
caught IndexError
finally run

<function noraise at 0x00BB47F0>
finally run

<function raise2 at 0x00BB4830>
finally run
```


www.TopSage.com

```
Traceback (most recent call last):
  File "C:/Python25/except-finally.py", line 9, in <module>
    func()
  File "C:/Python25/except-finally.py", line 3, in raise2
    def raise2():  raise SyntaxError
SyntaxError: None
```

大家网 TopSage.com

就像我们在第27章见到过的，Python 2.5时，except和finally分句可以混合在相同try语句中。这使本节所讲的某些语法嵌套变得不再必要，虽然依然可用，但可能是出现在Python 2.5版以前的代码中，而且可作为执行其他的异常处理行为的技术。

异常的习惯用法

我们已看过异常背后的机制。现在，让我们看看它们的其他常见的用法。

异常不总是错误

在Python中，所有错误都是异常，但并非所有异常都是错误。例如，我们在第9章看过，文件对象读取方法会在文件末尾时返回空字符串。与之相对比的是，内置的raw_input函数（我们在第3章第一次见到，在第10章时用于交互模式的循环中）在每次调用时，则是从标准输入串流sys.stdin读取一行文字，并且在文件末尾时引发内置的EOFError。和文件方法不同的是，这个函数并不返回空字符串：raw_input的空字符串是指空行。除了EOFError的名称，这个异常在这种环境下也只是信号而已，不是错误。因为有这种行为，除非文档末尾应该终止脚本，否则，raw_input通常会出现在try处理器内，并嵌入循环内，如下列代码所示。

```
while 1:
    try:
        line = raw_input()          # Read line from stdin
    except EOFError:
        break                       # Exit loop at end-of-file
    else:
        ...process next line here...
```

其他内置异常都是类似的信号，而不是错误。Python也有一组内置异常，代表警告，而不是错误。其中有些代表了正在使用不推荐的（即将退出的）语言功能的信号。请参考库手册有关内置异常的说明，以及warnings模块相关的警告。

函数信号条件和raise

用户定义的异常也可引发非错误的情况。例如，搜索程序可以写成找到相符者时引发异

www.TopSage.com

常，而不是为调用者返回状态标志来拦截。在下面的代码中，try/except/else处理器做的就是if/else返回值的测试工作。

```
class Found(Exception): pass

def searcher():
    if ...success...:
        raise Found()
    else:
        return

try:
    searcher()
except Found:                    # Exception if item was found
    ...success...
else:                            # else returned: not found
    ...failure...
```

更通常的情况是，这种代码结构，可用于任何无法返回警示值（sentinel value）以表明成功或失败的函数。例如，如果所有对象都是可能的有效返回值，就不可能以任何返回值来代表不寻常的情况。异常提供一种方式来传达结果讯号，而不使用返回值。

```
class Failure(Exception): pass

def searcher():
    if ...success...:
        return ...founditem...
    else:
        raise Failure()

try:
    item = searcher()
except Failure:
    ...report...
else:
    ...use item here...
```

因为Python核心是动态类型和多态的，通常更倾向于使用异常来发出这类情况的信号，而不是警示性的返回值。

在try外进行调试

也可以利用异常处理器，取代Python的默认顶层异常处理行为。在顶层代码中的外层try中包装整个程序（或对它调用），就可以捕捉任何程序执行时会发生的异常，因此可破坏默认的程序终止行为。

在下面的代码中，空的except分句会捕捉任何程序执行时所引发的而未被捕捉到的异常。要取得所发生的实际异常，可以从内置sys模块取出sys.exc_info函数的调用结

www.TopSage.com

果。这会返回一个元组，而元组之前两个元素会自动包含当前异常的名称，以及其相关的额外数据（如果有的话）。就基于类的异常而言，这两个元素分别对应的是异常的类以及引发的类实例（关于sys.exc_info的更多内容稍后介绍）。

```
try:
    ...run program...
except:                              # All uncaught exceptions come here
    import sys
    print 'uncaught!', sys.exc_info()[0], sys.exc_info()[1]
```

这种结构在开发期间会经常使用，在错误发生时，仍保持程序处于激活状态：这样可以执行其他的测试，而不用重新开始。测试其他程序时，也会用到它，就像下一节所描述的那样。

运行进程中的测试

可以在测试驱动的应用中结合刚才所见到的一些编码模式，在同一进程中测试其他代码。

```
import sys
log = open('testlog', 'a')
from testapi import moreTests, runNextTest, testName

def testdriver():
    while moreTests():
        try:
            runNextTest()
        except:
            print >> log, 'FAILED', testName(), sys.exc_info()[:2]
        else:
            print >> log, 'PASSED', testName()

testdriver()
```

在这里的testdriver函数会循环进行一组测试调用（在这个例子中，模块testapi是抽象的）。因为测试案例中未被捕捉的异常，一般都会终止这个测试驱动程序，如果你想在测试失败后让测试进程继续下去，就需要try中包装测试案例的调用。就像往常一样，空的except会捕捉由测试案例所产生的没有被捕捉的异常，而其使用sys.exc_info把该异常记录到文件内。没有异常发生时（测试成功情况），else分句就会执行。

对于作为测试驱动运行在同一个进程的函数、模块以及类，而进行测试的系统而言，这种形式固定的代码是很典型的。然而，在实际应用中，测试可能会比这里所演示的更为精致复杂。例如，要测试外部程序时，你要改为检查程序启动工具所产生的状态代码或

www.TopSage.com

输出，例如，标准库手册所谈到的os.system和os.popen（这类工具一般不会替外部程序中的错误引发异常。事实上，测试案例可能会和测试驱动并行运行）。

本章结尾时，我们将会遇到Python提供的其他更完整的测试框架，例如，Doctest和PyUnit，它们都提供了比较实际结果和预期的输出的工具。

关于sys.exc_info

通常来说，前两节中所用的sys.exc_info结果是获得最近引发的异常的更好的方式。如果没有处理器正在处理，就返回包含了三个None值的元组。否则，将会返回（*type*、*value*和*traceback*）：

- *type*是正在处理的异常的异常类型（一个基于类的异常的类对象）。
- *value*是异常参数（它的关联值或raise的第二参数，如果异常类型为类对象，就一定是类实例）。
- *traceback*是一个traceback对象，代表异常最初发生时的所调用的堆栈（参考标准traceback模块到文档，来获得可以和这个对象共同使用的对象工具从而可以手动产生错误消息的工具）。

旧的工具sys.exc_type和sys.exc_value依然可用于获得最近异常的类型和值，但是只能替整个进程管理一个全局的异常。更倾向于使用的sys.exc_info可记录每个线程的异常信息，因此是线程专有的方式。当然，当你在Python程序中使用多线程时，这种区别才显得重要（这个主题不在本书范围内）。参考Python库手册或后续的书籍的来获得更多的细节。

与异常有关的技巧

大致来说，Python的异常在使用上都很简单。异常背后真正的技巧在于确定except分句要多具体或多通用，以及try语句中要包装多少代码。我们先介绍第二项。

应该包装什么

从理论上讲，可以在脚本中把所有的语句都包装在try中，但这样做不够明智（这样的话try语句也需要包装在try语句中）。这其实是设计的问题，不在语言本身的范围内，而且实际运用时，就会更明显。以下是一些简要的原则。

www.TopSage.com

- 经常会失败的运算一般都应该包装在try语句内。例如，和系统状态衔接的运算（文件开启、套接字调用等等）就是try的主要候选者。
- 尽管这样，上边的规则有些特例：在简单的脚本中，你会希望这类运算失败时终止你的程序，而不是被捕捉或是被忽略。如果是一个重大的错误，更是如此。Python中的错误会产生有用的错误消息（如果不是崩溃的话），而且这通常就是所期望的最好的结果。
- 应该在try/finally中实现终止动作，从而保证它们的执行。这个语句的形式可以执行代码，无论异常是否发生。
- 偶尔，把对大型函数的调用包装在单个try语句内，而不是让函数本身零散着放入若干try语句中，这样会更方便。这样的话，函数中的异常就会往上传递到调用周围的try，而你也可以减少函数中的代码量。

你所编写的程序种类可能会影响处理异常的代码的量。例如，服务器一般都必须持久性地运行，所以，可能需要try语句捕捉异常并从中恢复。本章所见的进程内的测试程序可能也需要处理异常。不过，较简单的一次性的脚本，通常会完全忽略异常的处理，因为任何步骤的失败都要求关闭脚本。

捕捉太多：避免空except语句

另一个问题是处理器的通用性的问题。Python可选择要捕捉哪些异常，有时候必须小心，不要涵盖太广。例如，空except分句会捕捉try代码块中代码执行时所引发的每个异常。

这样很容易写，并且有时候也是我们想要的结果，但是也可能拦截到异常嵌套结构中较高层的try处理器所期待的事件。例如，下列的异常处理器，会捕捉每个到达的异常并使其停止，无论是否有另一个处理器在等待该事件。

```
def func():
    try:
        ...                         # IndexError is raised in here
    except:
        ...                         # But everything comes here and dies!

try:
    func()
except IndexError:                  # Exception should be processed here
    ...
```

也许更糟，这类代码可能会捕捉无关的系统异常。例如，内存错误、一般程序错误、迭代停止以及系统退出等等，都会在Python中引发异常。这类异常通常是不应该拦截的。

www.TopSage.com

例如，当控制权到达顶层文件末尾时，脚本正常都是结束。然而，Python也提供内置sys.exit(statuscode)调用来提前终止。这实际上是引发内置的SystemExit异常来终止程序，使try/finally可以在离开前执行，而程序的特殊类型可拦截该事件（注1）。因此，try带空except时，可能会不知不觉阻止重要的结束，如下面文件所示（*exiter.py*）：

```
import sys

def bye():
    sys.exit(40)                    # Crucial error: abort now!

try:
    bye()
except:
    print 'got it'                  # Oops-we ignored the exit
print 'continuing...'


% python exiter.py
got it
continuing...
```

可能无法预期运算中可能发生的所有的异常种类。

最糟的情况是，空except也会捕捉一般程序设计错误，但这类错误多数时候都应让其通过。事实上，空except分句会有效关闭Python的错误报告机制，使得代码中的错误难以发现。例如，考虑下面的代码。

```
mydictionary = {...}
...
try:
    x = myditctionary['spam']     # Oops: misspelled
except:
    x = None                      # Assume we got KeyError
...continue here with x...
```

在这里代码的编写者假设，对字典做字典运算时，唯一可能发生的错误就是键错误。但是，因为变量名myditctionary拼写错误了（应该是mydictionary），Python会为未定义的变量名的引用引发NameError，但处理器会默默的捕捉并且忽略了这个异常。事件处理器错误填写了字典错误的默认值，导致了程序出错。如果这件事是发生在离读取值的使用很远的地方，就会变成一项很有趣的调试任务！

注1：一个相关的调用os._exit也会结束程序，是立即终止：跳过清理动作，无法被try/except或try/finally代码块拦截。这通常只用在衍生的子进程中，但这种话题不在本书讨论的范围之内。参考库手册或后续的书籍以获得更多细节。

www.TopSage.com

原则就是，尽量让处理器具体化：空except分句很方便，但是可能容易出错。例如，上一个例子中，最好是写except KeyError:，意图明确，避免拦截无关的事件。在较简单脚本中，潜在的问题可能不如全部捕获所带来的便利，通常来说，通用化的处理器一般都是麻烦的源头。

TopSage.com

捕捉过少：使用基于类的分类

另一方面，处理器也不应过于具体化。当在try中列出特定的异常时，就只捕捉实际所列出的事件。这不见得是坏事，如果系统演进发展，以后会引发其他的异常，就得回头在代码其他地方，把这些新的异常加入到异常的列表中去。

例如，下列处理器是把myerror1和myerror2看作是正常的情况，并把其他的一切视为错误。如果未来增加了myerror3 ，就会视为错误并对其进行处理，除非更新异常列表。

```
try:
    ...
except (myerror1, myerror2):      # Breaks if I add a myerror3
    ...                           # Nonerrors
else:
    ...                           # Assumed to be an error
```

值得庆幸的是，小心使用第28章讨论过的基于类的异常，可让这种陷阱消失。就像我们所见到的，如果你捕捉一般的超类，就可以在未来新增和引发更为特定的子类，而不用手动扩展except分句的列表：超类会变成可扩展的异常分类。

```
try:
    ...
except SuccessCategoryName:       # OK if I add a myerror3 subclass
    ...                           # Nonerrors
else:
    ...                           # Assumed to be an error
```

无论你是否使用基于类的异常的分类层次，采用一点细微的设计，就可以走得长远。这个故事的寓意是，异常处理器不要过于一般化，也不要过于太具体化，而且要明智选择try语句所包装的代码的量。特别是在较大系统中，异常规则也应该是整体设计的一部分。

异常陷阱

在异常这里出错的情况不多，有两个通用的使用指导（其中一个的总体概念，我们已经见到过了）。

www.TopSage.com

字符串异常匹配是通过同一性而不是通过值

现在假设异常是类而不是字符串，所以这个陷阱是老问题。然而，因为你还是可能在现有的代码中遇到基于字符串的异常，它还是值得了解的。当异常引发时（由你或由Python本身），Python会搜索最近进入的、具有相符的except分句的try语句（对字符串异常而言，也就是命名了相同字符串对象的except分句；对基于类的异常而言，就是相同类或其超类）。对字符串异常而言，注意到匹配是通过同一性而不是值相等，这一点很重要。

例如，假设我们定义两个字符串对象，我们想用它们引发异常。

```
>>> ex1 = 'The Spanish Inquisition'
>>> ex2 = 'The Spanish Inquisition'

>>> ex1 == ex2, ex1 is ex2
(True, False)
```

应用==测试会返回True，因为它们的值相等，is返回False，因为它们在内存内，是两个不同的字符串对象（假设它们足够长，可以突破Python字符串的内部缓存机制，第6章曾说明过）。于是，引用相同字符串对象的except分句就将总会匹配。

```
>>> try:
...     raise ex1
... except ex1:
...     print 'got it'
...
got it
```

但列出相等值而并非相同对象的语句，将会失败。

```
>>> try:
...     raise ex1
... except ex2:
...     print 'Got it'
...
Traceback (most recent call last):
  File "<pyshell#43>", line 2, in <module>
    raise ex1
The Spanish Inquisition
```

在这里，没有捕捉异常，因为对象的同一性不匹配，所以Python往上返回进程的顶层，并自动打印堆栈跟踪和异常文本。尽管如此，就像前文说过的，有些不同的字符串对象碰巧有相同值，实际上会匹配，这是因为Python会缓存小字符串并对其重用（第6章有说明过）。

www.TopSage.com

```
>>> try:
...     raise 'spam'
... except 'spam':
...     print 'got it'
...
got it
```

这样行得通是因为两个字符串因缓存机制而对应到了相同对象。反之，下列例子的字符串太长，以至于都不在缓存范围之内。

```
>>> try:
...     raise 'spam' * 8
... except 'spam' * 8:
...     print 'got it'
...
Traceback (most recent call last):
  File "<pyshell#58>", line 2, in <module>
    raise 'spam' * 8
spamspamspamspamspamspamspamspam
```

如果看来有些难懂，你所需记住的就是：对于字符串而言，要确定在raise和try中使用相同对象，而不是相同的值。

对基于类的异常而言（如今推荐使用的技术），虽然其整体行为类似，但是Python把异常配对的概念一般化，从而引入了超类关系，使得这个陷阱不再存在，而这也是另一个现在要使用类异常的原因。

捕捉到错误的异常

也许和异常有关，最常见的陷阱与上一节讨论的设计原则有关。记住，要尝试避免使用空except分句（否则会捕捉到系统离开这类异常），以及过度具体的except分句（使用超类分类，从而避免了未来增加的新异常时的维护问题）。

核心语言总结

这里要总结核心Python程序设计语言。如果你学习到这里，就可以把自己当成正式的Python程序员了（下次填简历时，不妨把Python也加进去）。你已经看过了语言本身的大部分内容，而且比实际中的很多Python程序员一开始所需做的还要更深入。你已研究过内置类型、语句以及异常，而且还有用于创建较大程序单元的工具（函数、模块以及类）。你甚至探索过重要的设计问题、OOP和程序架构等。

www.TopSage.com

Python工具集

从这里开始，你以后的Python生涯大部分就是在熟练掌握应用级的Python编程的工具集。你将发现这是一项持续不断的任务。例如，标准库包含了几百个模块，而公开领域提供了更多的工具。你有可能花个十年甚至更多的时间去研究所有这些工具，尤其是新的工具还在不断地涌现出来（这一点你可以相信我）

一般而言，Python提供了一个有层次的工具集。

内置工具

像字符串、列表以及字典这些内置类型，会让编写简单的程序更为迅速。

Python扩展

就更重要的任务来说，你可以编写自己的函数、模块以及类，来扩展Python。

已编译的扩展

虽然我们在本书中没介绍这一话题，Python也可以使用C或C++这样的外部语言所编写的模块进行扩展。

因为Python将其工具集分层，可以决定程序任务要多么的深入这个层次：你可以让简单脚本使用内置工具，替较大系统新增Python所编写的扩展工具，并且为高级的工作编写编译好的扩展工具。我们已在本书谈到过前两种类型，而这些已经足够开始编写实际的Python程序。

表29-1总结Python程序员可用的内置或现有的功能的来源，而有些话题你可能会用剩余的Python生涯时间来探索。到目前为止，我们多数例子都很小，独立完备。它们是有意这样编写的，也就是为了帮助你掌握基础知识。现在，了解核心语言知识的，该是学习如何使用Python内置接口来进行实际工作的时候了。你会发现，利用Python这种简单的语言，常见任务会比你想象的更为简单。

表29-1：Python的工具箱类型

分类	例子
对象类型	列表、字典、文件和字符串
函数	len、range、apply、open
异常	IndexError、KeyError
模块	os、Tkinter、pickle、re
属性	__dict__、__name__、__class__
外部工具	NumPy、SWIG、Jython、IronPython等

www.TopSage.com

大型项目的开发工具

一旦精通了Python基础知识，就会发现Python程序变得比你至今体验过的例子还要大。对于开发大型系统而言，Python和公开领域有一批开发工具可以使用。你已看过其中几种工具的用法，而且本书也提过一些。为了帮助你开始编程，以下是此领域中一些最常用的工具的摘要。

PyDoc和文档字符串

PyDoc的help函数和HTML界面在第14章介绍过。PyDoc为模块和对象提供了一个文档系统，并整合了Python的文档字符串功能。这是Python系统的标准部分，参考库手册以获得更多细节。请确认返回查看了第4章所列举的文档资源的提示，以了解其他Python资源的信息。

PyChecker

因为Python是的一门动态语言，有些程序设计的错误在程序运行前不会报错（例如，当文件执行或导入时，语法错误会被捕捉）。这不是什么大的缺点：就像大多数语言一样，这只是代表这把产品传送给客户前，需要测试你的Python程序。此外，Python的动态本质、自动错误消息以及异常模型，使你很容易在Python中寻找和修正错误，远胜过其他语言（例如，和C不同的是，Python不会因错误而崩溃）。

PyChecker系统可以在脚本运行前把大量的常见错误预先缓存起来。这和C开发领域中的“lint”程序扮演了类似的角色。有些Python社区会在测试或者分发给客户前，先用PyChecker执行其代码，来捕捉任何潜在的问题。事实上，Python标准库在发布前都会定期用PyChecker执行。PyChecker是第三方开方源代码包。你可以在*http://www.python.org*或者Vaults of Parnassus网站上找到。

PyUnit（也就是unittest）

在第五部分中，我们看过如何在文件末尾使用__name__ =='__main__'技巧，把自我测试代码加到Python文件中。就更高级的测试目的而言，Python有两个测试辅助工具。第一个是PyUnit（在库手册中称为unittest），提供了一个面现对象的类框架，来定义和定制测试案例以及预期的结果。这是模拟Java的JUnit框架的。这是个精致的类系统。更多细节请参考Python库手册。

doctest

doctest标准库模块，提供第二个并且更为简单的做法来进行回归测试。这是基于Python的文档字符串功能的。概括地讲，要使用doctest时，把交互模式测试会话的记录复制粘贴到源代码文件中的文档字符串中。然后，doctest会抽取出你的

www.TopSage.com

文档字符串，分解出测试案例和结果，然后重新执行测试，并验证预期的结果。doctest的操作可以用多种方式剪裁。更多细节请参考库手册。

IDE

我们在第3章讨论过Python的IDE。例如IDLE这类IDE，提供了图形环境，来编辑、运行、调试以及浏览Python程序。有些高级的IDE，例如，Eclipse和Komodo，支持其他的开发任务，包括源代码控制整合、GUI交互构建工具和项目文件等。参考第3章、*http://www.python.org*中text editor的网页以及Vaults of Parnassus网站有关Python可用的IDE和GUI构建工具。

配置工具

因为Python是高级和动态的，从其他语言学习得到的直接经验，通常不适用于Python代码。为了真正把代码中的性能的瓶颈隔离出来需要通过time或timeit模块内的时钟工具新增计时逻辑，或者在profile模块下运行代码。在第17章比较迭代工具的速度时，我们看过time模块的用法。

profile是标准库模块，为Python实现源代码配置工具。它会运行你所提供的代码的字符串（例如，脚本文件导入，或者对函数的调用）。在默认情况下，它会打印报告到标准输出流，从而给出性能的统计结果：每个函数的调用次数、每个函数所花时间等。profile模块能以多种方式进行定制。例如，可以把统计资料保存到文件中，稍后使用pstats模块进行分析。

调试器

Python标准库包含了一个命令行源代码调试器模块，称为pdb。这个模块很像C语言的命令行调试器（例如，*dbx*、*gdb*）：导入这个模块，调用pdb函数开始执行代码[例如，pdb.run("main()")]，然后在交互模式提示符下输入调试命令。pdb包括了实用的事后分析，调用pdb.pm可在遇到异常后启动调试器。因为IDLE这类IDE包括“指针并点击”的调试接口，pdb其实现在很少有人使用。参考第3章有关使用IDLE的调试GUI接口的技巧（注2）。

发布的选择

在第2章中，我们介绍打包Python程序的常见工具。*py2exe*、PyInstaller以及freeze都可打包字节码以及Python虚拟机，从而成为“冻结二进制”的独立的可执行文

注2：老实说，IDLE的调试器也很少使用。多数实务的Python程序员都是插入策略性的print语句，并执行它来对代码进行调试的。因为在Python中，修改到执行的时间很短，加上print语句通常比输入pdb调试器命令，或者启动GUI调试进程更为快速。另一个有效的调试技巧是什么也不做。因为Python会打印有用的错误消息，而不是遇见程序错误时就崩溃，因此你通常会得到足够的信息来分析和修复错误。

www.TopSage.com

件，也就是不需要目标机器上有安装Python，完全可以隐藏系统的代码。此外，第2章和第5部分学过，当Python程序分发给客户时，可以采用源代码的形式（*.py*）或字节码的形式（*.pyc*），此外，导入钩子支持特殊的打包技术，例如，*.zip*文件自动解压缩以及字节码加密。我们也简单谈过标准库的`distutils`模块，为Python模块和套件以及C编写而成的扩展工具提供了各种打包选项，更多细节请参考Python手册。后起之秀Python eggs打包系统提供另一种做法，也可解决包的相互依性，更多细节请在互联网上搜索。

优化选项

优化程序时，第2章所提到的Psyco系统提供了实时的编译器，可以把Python字节码翻译成二进制机码，而Shedskin提供了Python对C++的翻译器。偶尔会看见*.pyo*优化后的字节码文件，这是以`-O` Python命令标志位运行所产生的文件（第18章讨论过），这只提供了有限的性能提升，并不常用。最后，也可以把程序的一些部分改为用C这类编译型语言编写，从而提升程序性能，参考《Programming Python》以及Python标准手册来了解C扩展的更多细节。一般来说，Python的速度也会随时间不断改善，要尽可能升级到最新的版本（例如，2.3版的速度就比2.2版快15~20%）。

对于大型项目的提示

我们在本书中遇过各种语言特性，当开始编写大型项目时，就会变得更有用。其中，模块包（第20章）、基于类的异常（第28章）、类的伪私有属性（第25章）、文档字符串（第14章）、模块路径配置文件（第18章）、从`from *`以及`__all__`列表到`_X`风格的变量名来隐藏变量名（第21章）、以`__name__ =='__main__'`技巧来增加自我测试代码（第21章）和使用函数和模块的常见设计规则（第16、17以及21章）等。

要学习公开场合的其他的大型的Python开发工具，一定要去浏览Vaults of Parnassus网站的相关网页。

本章小结

本章是异常这一部分内容的结尾，也是本书的结尾。我们看了常见的异常使用实例，以及简要地介绍了常用的开发工具。因为这是本书结尾，习题少一些，这次只有一个问题。不过，一定要做一做本部分结尾的练习题，来强化前几章学到的概念。接下来的附录，提供了安装提示以及习题和练习题的答案 。

www.TopSage.com

有关本书之后的阅读方向，可参考序文中所列的后续书籍列表。你现在已经开始觉得Python确实有趣了，不过本书也结束了。你现在有了很好的编程基础，可借助其他书籍和资源，来帮助你进行真正的应用程序的设计工作，例如，创建GUI、网站和数据库接口等。希望你的旅程顺利，“要留意生命中美好的那一面！”

www.TopSage.com

本章习题

1. （这个问题出自第1章的习题；你看，我告诉你这很简单：-））为什么这本书很多例子中都出现了spam?

习题解答

1. 因为Python这个名字是从英国喜剧团体Monty Python（根据我在课堂上的调查，这是Python世界中向来保持和很好的一个秘密！）spam这个词是来自Monty Python喜剧，有人试着在露天咖啡馆点餐，却被维京人合唱团唱着有关spam的歌声给淹没掉。如果我可以在这里插播那首歌的音频剪辑，作为本书谢幕的幕后工作人员名单的片尾曲，我一定会这么做的……

www.TopSage.com

大家网 TopSage.com

第七部分 练习题

我们已经学到了这一部分的结尾，是该做一做与异常相关的练习题，来复习其基础知识。异常其实是相当简单的工具。如果你能把这些弄懂，那么你就算精通了。

参考附录B的解答。

1. `try/except`。写个名为`oops`的函数，在调用时，故意抛出`IndexError`异常。之后，编写另外一个函数在一个`try/except`语句中调用`oops`并捕获这个异常。如果你修改`oops`引发的是`KeyError`，而不是`IndexError`，会发生什么情况？`KeyError`和`IndexError`是从哪里来的？（提示：回想一下，根据LEGB法则，没有点号的变量都来自四个作用域之一。）

2. **异常对象和列表**。修改你刚写的`oops`函数，来引发你自己定义的`MyError`异常，然后随着这个异常传递额外的数据元素。你可以使用字符串或类来识别异常。然后，扩展捕捉者函数中的`try`语句，来捕捉这个异常及其数据（除了`IndexError`外），以及打印其额外的数据元素。最后，如果你使用字符串异常，就回头将其修改成类实例。现在，传给处理器的额外数据是什么？

3. **错误处理**。编写一个名为`safe(func, *args)`的函数，使用`apply`执行任意函数（或者较新的`*name`调用语法），当这个函数执行时捕捉任意引发的异常，以及使用`sys`模块中的`exc_type`和`exc_value`属性来打印异常（或者更新的`sys.exc_info`调用结果）。然后，使用`safe`函数来执行练习题1或2的`oops`函数。把`safe`放在名为*tools.py*的模块文件中，通过交互模式将其传给`oops`函数。你会得到哪种错误讯息？最后，扩展`safe`，在错误发生时，通过调用标准`traceback`模块的支持`print_exc`函数来打印Python堆栈跟踪（参考Python库参考手册的细节）。

4. **自学例子**。在附录B末尾，列举了一组练习题例子的脚本，编写成Python类，可对应参考Python标准手册集来研究和运行。这些例子没有说明，而且使用了Python标准链中的工具，你得自己去搜索。但是，对多数读者而言，这有助于通过实际的程序了解本书所讨论的概念。如果这些会勾起你更大的兴趣，可以在《Programming Python》后续书籍中，或者用喜欢的浏览器搜索网络，找到更有价值的也更实际的Python应用程序的例子。

www.TopSage.com

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

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

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库管理系统(DBMS)精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

平面设计优秀资源学习下载 | Flash 优秀资源学习下载 | 3D 动画优秀资源学习下载

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

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

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

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

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

>> 更多精品资料请访问大家论坛计算机区…

作者简介

作为全球Python培训界的领军人物。**Mark Lutz**是最早出版的Python最畅销书籍的作者，也是Python社区的先驱。

Mark 是O'Reilly出版的《Programming Python》和《Python Pocket Reference》的作者，目前这两本书都已经出版了第三版。Mark自1992年开始接触Python，1995年开始撰写有关Python的书籍，从1997年开始教授Python课程，截止到2007年中，他已经开办了200多个Python短期培训课程。

此外，Mark拥有威斯康星大学计算机科学学士和硕士学位，在过去的25年中，他主要从事编译器、编程工具、脚本程序以及各种客户端/服务器系统方面的工作。

每当Mark休假的时候，他都会在科罗拉多过着普通的生活。Mark的电子邮箱是：*lutz@rmi.net*，你也可以通过访问*http://www.rmi.net/~lutz* 或*http://home.earthlink.net/~python-training*与他取得联系。

封面介绍

《学习Python》第三版的封面动物为林鼠（wood rat，鼠科林鼠属），林鼠能够居住于各种环境（大多是在多岩石、灌木丛或是沙地），遍布北美洲和中美洲，一般会远离人类。林鼠善于攀爬，巢居在离地面大约六公尺的树上或是灌木上，有些种类的林鼠会居住在地洞或是岩石的缝隙中，也有时会住在其他动物放弃的洞穴里。

这些灰色中型啮齿类动物又称为收集鼠（pack rat）：它们喜欢把各种各样的东西运回自己的巢穴，无论是否有用，它们对闪闪发亮的东西尤其感兴趣，比如易拉罐、玻璃或者银器。

封面图来自19世纪Cuvier's Animals的雕刻板画。

www.TopSage.com